# The

# Programming Language

## Concurrent Programming in an Extended Java

by
Ronald A. Olsson
Aaron W. Keen

Kluwer Academic Publishers

# THE JR PROGRAMMING LANGUAGE

*Concurrent Programming in an Extended Java*

**THE KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE**

# THE JR PROGRAMMING LANGUAGE

## *Concurrent Programming in an Extended Java*

*by*

**Ronald A. Olsson**
*University of California, Davis*
*U.S.A.*

**Aaron W. Keen**
*California Polytechnic State University*
*U.S.A.*

**KLUWER ACADEMIC PUBLISHERS**
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

eBook ISBN: 1-4020-8086-7
Print ISBN: 1-4020-8085-9



Created in the United States of America

Visit Springer's eBookstore at: http://www.ebooks.kluweronline.com
and the Springer Global Website Online at: http://www.springeronline.com

*To the memory of my parents, Dorothy and Ronald* ***RAO***

*To all who have touched my life* ***AWK***

*This page intentionally left blank*

# Contents

*This page intentionally left blank*

# List of Figures

# List of Tables

*This page intentionally left blank*

# Preface

JR is a language for concurrent programming. It is an imperative language that provides explicit mechanisms for concurrency, communication, and synchronization. JR is an extension of the Java programming language with additional concurrency mechanisms based on those in the SR (Synchronizing Resources) programming language. It is suitable for writing programs for *both* shared- and distributed-memory applications and machines; it is, of course, also suitable for writing sequential programs. JR can be used in applications such as parallel computation, distributed systems, simulation, and many others.

JR supports many "features" useful for concurrent programming. However, our goals have always been keeping the language simple and easy to learn and use. We have achieved these goals by integrating common notions, both sequential and concurrent, into a few powerful mechanisms. We have implemented these mechanisms as part of a complete language to determine their feasibility and cost, to gain hands-on experience, and to provide a tool that can be used for research and teaching. The introduction to Chapter 1 expands on how JR has realized our design goals.

As noted above, JR is based on Java and SR. Java itself provides concurrency via threads and a monitor-like mechanism. Java also provides RMI for distributed programming. However, these mechanisms are low-level and not easy to use (especially RMI). In contrast, JR provides higher-level abstractions that are much simpler and more flexible to learn and use. (For an illustrative example, see Reference [33]). JR is a more modern language than SR, e.g., it is object-oriented. Being an extension of Java, JR should be easier for students who already know Java to learn than it would be for them to learn SR, which is an entirely different language. That is, students' attention can be focused on learning the concurrent extensions, not learning an entirely new language (both sequential and concurrent mechanisms). (See Appendix E for a detailed comparison of SR and JR.) JR programs also should run on any platform that

supports Java (and the fairly standard tools used within the JR implementation) and can use Java's packages.

The JR implementation comes with three preprocessors that convert notations for CCRs, monitors, and CSP (Communicating Sequential Processes) into JR code. These allow students to get hands-on experience with those mechanisms. Together with JR, the three preprocessors provide a complete teaching tool for a spectrum of synchronization mechanisms: shared variables, semaphores, CCRs, monitors, asynchronous message passing, synchronous message passing (including output commands in guards, as in extended CSP), RPC, and rendezvous. JR itself directly contains the mechanisms other than CCRs, monitors, and CSP.

## Online Resources

The JR webpage is

```
http://www.cs.ucdavis.edu/~olsson/research/jr
```

The JR implementation is in the public domain and is available from the JR webpage. The JR implementation executes on UNIX-based systems (Linux, Mac OS X, and Solaris) and Windows-based systems. JR code is translated to native Java code, which executes using the JR run-time system (RTS). The implementation also uses true multiprocessing when run on a multiprocessor. The implementation includes documentation and many example programs. We can't provide a warranty with JR; it's up to you to determine its suitability and reliability for your needs. We do intend to continue to develop and maintain JR as resources permit, and would like to hear of any problems (or successes!) and suggestions for improvements. Via email, contact `jr-project@cs.ucdavis.edu`.

Complete source code for all programming examples and the "given" parts of all programming exercises in the book are also available on the JR webpage. This source code is organized so that we can easily test all programs and program fragments to ensure that they work as advertised. As a result, we hope that there will be very few bugs in the programs (a common source of annoyance in programming language books).

## Content Overview

This book contains 21 chapters. The first chapter gives an overview of JR and includes a few sample programs. The remaining chapters are organized into two parts: extensions for concurrency and applications. In addition, the appendices contain language reference material, describe how to develop and execute programs, present an overview of JR's implementation and performance, and trace JR's historical roots.

The introduction to Part I summarizes the key language mechanisms. The introduction to Part II describes how the applications relate to the material in Part I. Each chapter in Part I (except for one) introduces new language mechanisms and develops solutions to several problems. Some problems are solved in more than one chapter to illustrate the tradeoffs between different language mechanisms. The problems include the "classic" concurrent programming problems—e.g., critical sections, producers and consumers, readers and writers, the dining philosophers, and resource allocation—as well as many important parallel and distributed programming problems. Each chapter in Part II describes an application, presents (typically) several solutions, and describes the tradeoffs between the solutions. (However, the last two chapters of Part II deal with graphical user interfaces and other concurrency notations.) The end of each chapter contains numerous exercises, including several that introduce additional material.

Part I describes how JR extends Java with mechanisms for concurrency. Chapter 2 gives an overview of these extensions. Chapter 3 introduces the operation; because this mechanism is so fundamental to JR, this chapter focuses on just its sequential aspects. Chapter 4 introduces the language mechanisms for creating concurrently executing processes. Chapter 5 presents synchronization using shared variables; although this kind of synchronization requires no additional language mechanisms, it does show one low-level way in which processes can interact. Chapters 6, 7, 8, and 9 show how processes can synchronize and communicate using semaphores, asynchronous message passing, remote procedure call, and rendezvous, respectively. All these mechanisms are variations on JR's operations. Chapter 10 describes how to distribute a program so that it can execute in multiple address spaces, potentially on multiple physical machines such as a network of workstations. Chapter 11 describes the classic dining philosophers problem to show how many of JR's concurrency features can be used with one another. Chapter 12 describes how JR's mechanisms for operation invocation and servicing deal with exceptions. Chapter 13 defines and illustrates how operations can be inherited. Finally, Chapter 14 presents additional mechanisms for servicing operation invocations in more flexible ways.

Part II describes several realistic applications for JR. Chapter 15 gives four solutions to matrix multiplication. It includes solutions appropriate for both shared- and distributed-memory environments. Chapter 16 describes grid computations for solving partial differential equations. It too provides both shared- and distributed-memory solutions. Chapter 17 presents solutions to the traveling salesman problem that employ two important paradigms: bag of tasks and manager/workers. Chapter 18 describes a prototype distributed file system. Chapter 19 shows how to program a discrete event simulation in JR. Finally, Chapter 20 describes how JR programs can interact with the Java GUI (graph-

ical user interface) packages AWT and Swing. Finally, Chapter 21 describes other concurrency notations, which preprocessors convert into JR programs.

The first three appendices contain material in quick-reference format. They are handy when actually programming in JR. Appendix A summarizes the syntax for the JR extensions. Appendix B provides the details of the classes and methods used with the inter-operation invocation selection mechanism described in Chapter 14. Appendix C describes how to develop, translate, and execute JR programs. Appendix D gives an overview of the implementation and describes the performance of JR code. Finally, Appendix E gives a short history of the JR language, mentions other JR-related work, and cites papers published on JR.

## Classroom Use

Drafts of this text have been used over the last few years in a variety of undergraduate and graduate courses (formal classes and independent studies) at the University of California, Davis, and a few other universities. These courses cover topics such as programming languages, operating systems, concurrent programming, parallel processing, and distributed systems.

This text can serve as a stand-alone introduction to one particular concurrent programming language or as a supplement to a more general concurrent programming course. For example, the text can be used to teach a section on concurrent programming in an undergraduate programming language course. Indeed, SR is listed as one of the languages in the proposed knowledge units for programming languages in ACM's Curriculum 2001 (SIGPLAN Notices, April 2000); JR can serve that purpose, too, and is, as already noted, a more modern and easier-to-learn language. In course ECS 140B course at UC Davis, we spend about three and a half weeks in lecture on JR. Lectures cover all of Part I, although they only touch on the more advanced topics in Chapters 12–14, and most of the applications in Part II. Students write about a dozen small programs, mostly based on exercises in the book and do a small, group term project using distributed programming. The project requires that the program run on several physical machines and uses a GUI (Swing or AWT, as in Chapter 20) to show some visualization of the program's execution. This project has been very successful. Since JR is an extension to Java, JR can be used with Swing or AWT without trouble. Students can focus on the distributed aspects of the project, which JR makes easy with its notions of virtual machines and interprocess communication. A course could spend less time, yet still provide a good introduction to concurrent programming, by covering most of Part I, and just one or two of the applications from Part II.

As another example, the text forms a natural supplement for a course that uses Greg Andrews's text entitled *Concurrent Programming: Principles and Practice,* published by Benjamin/Cummings. That text explores the concepts of

concurrent programming, various synchronization and communication mechanisms, programming paradigms, implementation issues, and techniques to understand and develop correct programs. The notation used there is fairly close to JR's notation. In course ECS 244 at UC Davis, students implement as JR programs some of their solutions to exercises in Andrews's text. The students use both native JR and the preprocessors that turn CCR, monitor, and CSP notation into JR code. The JR text can also serve as a supplement to Andrews's text entitled *Foundations of Multithreaded, Parallel, and Distributed Programming,* published by Addison-Wesley (the MPD notation, being based on SR, is fairly close to JR's notation) or other texts on concurrent programming.

JR and the preprocessors are also appropriate for undergraduate or graduate operating systems courses. JR's notation for processes, semaphores, and monitors is straightforward and is close to what is often used in lectures and texts. Instead of just writing their homework solutions on paper, students can write some small programs using shared variables, semaphores, and monitors, for which they can use JR and the preprocessors.

This book is aimed at junior or senior level undergraduate students and at graduate students. Knowledge of Java is recommended and assumed, but knowledge of C++ or another object-oriented language should suffice. The additional maturity and knowledge gained via courses in data structures, programming languages, or operating systems will be beneficial, although not essential, in understanding the material. The specific prerequisite courses depend on how the book is to be used. The following is a typical use of this book: Read Chapters 1 and 2 to get a feel for the language; read Chapter 3 very carefully to understand the pervasive concepts of operations and operation capabilities; read the rest of Part I to understand JR's concurrent aspects; and then read Part II to see how to apply JR in a number of application areas.

Each chapter contains exercises dealing with the concepts and examples presented in the chapter. They range from simple to more difficult ones, including suggestions for a number of larger projects, especially in Part II. A number of other exercises and projects can be found in general concurrent programming books. As noted above under "Online Resources", to save readers typing for some of the exercises, complete programs that appear in this text are available online.

*This page intentionally left blank*

# Acknowledgments

Developing a new programming language and writing a book on it is a multi-year project (more "multi" than even those who have completed similar projects expect at the onset). Numerous people have made contributions.

First and foremost, we acknowledge the influence of the SR language and its implementation. We thank the SR Project at The University of Arizona (`http://www.cs.arizona.edu/sr/`), from whom we borrowed ideas, prose, and code. We especially thank Greg Andrews, the founding father of SR, for his continuing encouragement and for his (and Addison-Wesley Publishing, Inc.'s) graciously allowing us to reuse portions of the SR text in this text. We also thank Gregg Townsend, who over the years has contributed key ideas to SR and has done a superb job with the SR implementation.

We received much help with the JR implementation from many great UC Davis undergraduate students and graduate students (names marked with †).

- Tingjian Ge† worked on an initial prototype of JR.
- Justin Maris wrote a tool to convert the programs in the SR test suite to JR.
- Greg Benson† contributed some ideas on invocation servicing.
- Andrew Arcilla, Ezequiel Cervante, Alan Ngai, and Cindy Truong bravely used an early JR prototype in independent study work.
- Ben Ahlborn, Brett Groel, Dan Simon, and Mike Weaver worked on porting the JR implementation to Windows; Hiu Ning (Angela) Chan, Esteban Pauli, Nija Shi†, and Erik Staab helped apply the finishing touches.
- Hiu Ning (Angela) Chan and Erik Staab added quantifiers to the input statement.
- Hiu Ning (Angela) Chan added parameterized virtual machines as part of her undergraduate honors thesis work.

- Steven Chau, Andre Nash, and Esteban Pauli did the initial work on adding the concurrent invocation statement. Esteban Pauli, as part of his undergraduate honors thesis work, and Hiu Ning (Angela) Chan are continuing that work and are nearing completion of a working prototype.
- Erik Staab is looking into ways to improve JR's performance (both compile time and run-time) and at porting JR to use Java 1.5.
- Alex Wen and Ingwar Wirjawan are adding a timeout arm to the input statement.

We appreciate the feedback and patience of students at UC Davis who used early versions of JR and the JR book in ECS 140B and ECS 244 classes over the last few years. The System Support Group of the UC Davis Department of Computer Science, especially Babak Moghadam, did a great job in keeping our systems up and up-to-date!

We thank Sun Microsystems, Inc. for making public the source code for their Java translator and virtual machine. JR's translator is built using Sun's Java translator.

The staff of Kluwer Academic Publishers has been great in helping us get this book into print. Susan Lagerstrom-Fife is the Publishing Editor and Sharon Palleschi is the Editorial Assistant. They have been very responsive and helpful with all our large and small questions. The Kluwer Author Support Help Desk was also great in answering our detailed formatting questions. Amy Hendrickson of TEXnology, Inc. (Kluwer's consultant) nicely "tweaked" the formatting macros to fit our specific needs. (Camera-ready copy was produced by Olsson.)

The National Science Foundation supported our early JR-related research through grant CCR-9527295 at UC Davis. The NSF also partially supported the computing equipment used in our work through grant EIA-0224469 at UC Davis.

Many other people have made less technical, but not less meaningful contributions to this book. Ron thanks his family and friends (especially Nancy Wilson) for their support, understanding, and encouragement over the years; he also thanks his present and former students, including his keen co-author, for their continuing inspiration. Aaron thanks his family and friends for their continued encouragement and support, his colleagues at Cal Poly for providing a supportive environment in which to pursue this undertaking, and Ron, without whom this book would not exist.

Chapter 1

# INTRODUCTION

Concurrent programming is concerned with writing programs having multiple processes that may execute in parallel. The topic originated in the 1960s when the invention of independent device controllers (channels) led people to organize operating systems as concurrent programs, even for single-processor machines. Since then, rapid developments in computer architecture have led to an increasingly large number of multiprocessor architectures, such as shared-memory multiprocessors, multicomputers, and networks of workstations. The operating systems for these architectures are all instances of concurrent programs. More importantly, multiprocessor architectures make it possible to write application programs that exploit the concurrency inherent in the hardware. Both distributed systems, multiprocessor systems, and hybrids (e.g., distributed systems that include some multiprocessors) are prevalent today and they are likely to remain so.

A concurrent program specifies two or more processes that cooperate in performing a task. Each process consists of a sequential program. The processes cooperate by communicating, which in turn gives rise to the need for synchronization. Communication and synchronization are programmed by reading and writing shared variables or by sending and receiving messages. Shared variables are most appropriate for concurrent programs that execute on a single processor or a shared-memory multiprocessor. Message passing is most appropriate for distributed programs that execute on multicomputers or networks of workstations. (Message passing can also be used on shared-memory machines.)

This book describes the JR programming language and shows how it can be used to write concurrent programs for a variety of hardware architectures and software applications. JR is an extension of the Java programming lan-

guage [28] with additional concurrency mechanisms based on those in the SR (Synchronizing Resources) programming language [6, 9].

Java has proven to be a clean and simple (and popular) language for object-oriented programming. Even so, the standard Java concurrency model is rather limited. It provides threads, a primitive monitor-like mechanism, and remote method invocation (RMI). Although these features are useful, they offer little flexibility in the design and implementation of concurrent programs.

JR provides a richer and more flexible concurrent programming model than Java. JR adapts the following features from SR: dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, dynamic process creation, support for rendezvous, asynchronous message passing, semaphores, and shared variables. JR takes a novel object-oriented approach to synchronization whereas SR is not object-oriented.

Thus, JR inherits and extends one of SR's distinguishing attributes: its expressive power. The communication and synchronization mechanisms listed above include most of the ones that have proven popular and useful. This makes JR suitable for writing concurrent programs for *both* shared- and distributed-memory applications and machines.

In addition to being expressive, JR is easy to learn and use for someone who has some background with Java. Its variety of concurrent programming mechanisms is based on only a few underlying concepts. Moreover, these concepts are generalizations of ones that have been found useful in sequential programs. The concurrent programming mechanisms are also integrated with the sequential ones, so that similar things are expressed in similar ways. An important design goal has been to retain the "feel" of Java while providing a richer concurrency model.

Part I of this book describes the concurrent aspects of JR in detail and gives numerous, smaller examples. Part II develops complete programs for several larger applications: matrix multiplication, partial differential equations, the traveling salesman problem, a distributed file system, and discrete event simulation. These illustrate the use of JR for distributed programming using message passing and parallel programming using shared variables. JR is implemented on top of Java, so, in principle, it can run on any platform that supports Java, including networks of workstations and shared-memory multiprocessors. JR programs can also be executed on single processor machines, in which case process execution is interleaved. The current JR implementation runs on UNIX-based (Linux, Mac OS X, and Solaris) and Windows-based systems.

The remainder of this chapter gives a brief overview of JR. First we describe the main components of the language. Then we present complete programs that solve several familiar problems. The solutions illustrate the structure of JR programs and some—but by no means all—of the language's power and flexibility. Finally, we describe how to create and execute JR programs.

## 1.1 Key JR Components

As noted above, JR extends Java with additional mechanisms for supporting concurrency. The key new features are virtual machines, remote objects, and operations.

A JR virtual machine represents an address space, which is located entirely on one physical machine. These virtual machines can be created dynamically during program execution in a way similar to how objects are created. JR virtual machines can be "populated" with remote objects, which are essentially the usual instances of classes. In JR, a remote object is simply a Java object that has been created in a way slightly different from the usual Java new. Thus, JR object creation is dynamic, as in Java. A class in Java serves as the unit of compilation and encapsulation; a class in JR serves a similar role. A JR class may contain anything that a Java class may contain plus it may contain additional JR features. The one difference in the use of classes is that in JR all classes must be compiled together.

One such feature is the process, which represents a separate thread of control.[1] JR provides a process abbreviation. Processes can be created dynamically and can share variables in the same object, in the same class (static variables), and in other classes on the same virtual machine (public static variables). Processes can also communicate and synchronize by means of operations.

An operation can be considered a generalization of a method: It has a name and can take parameters and return a result. An operation can be invoked in two ways: synchronously by means of a call statement or asynchronously by means of a send statement. An operation can also be serviced in two ways: by a method or by input statements. These ways of servicing an operation support local and remote method calls and rendezvous. As we shall see in Part I, this variety of possibilities provides a great deal of flexibility and power for solving concurrent programming problems.

JR contains several mechanisms that are abbreviations for common uses of operations; these can be used to simplify many programs. Abbreviations include process declarations, op-method declarations, receive statements, and semaphores. JR also provides a few additional statements that are useful for concurrent programming. The reply and forward statements provide additional ways to use operations.

JR also provides a means to deal with program quiescence. A JR program becomes *quiescent* when all of its processes have terminated or deadlocked. At that point, the JR implementation will normally terminate the program's execution. Instead, however, JR allows an operation to be registered as the "qui-

[1] JR uses the traditional term "process" to represent this abstraction. As we will see in later chapters, JR processes are actually mapped to Java threads. To further confuse matters, the term "process" is often used to represent an operating system process, which might contain multiple threads of execution.

escence operation"; this operation will be invoked when the program becomes quiescent. This feature is useful to avoid having to write code to determine when processes have terminated.

JR programs can use all of the many packages provided for Java. For example, these include common math functions and a variety of input/output functions. JR programs can also interact with Java packages for building GUIs (graphical user interfaces), such as AWT and Swing; Chapter 20 show some examples of such interaction.

## 1.2 Two Simple Examples

One of the best ways to learn a new programming language is to start writing programs. To do so, it helps to look at examples.

A standard first example in a programming language text is a program that writes the message "Hello World!" on the standard output file. In JR, the following program does the trick:

```
import edu.ucdavis.jr.JR;
public class HelloWorld {
  public static void main(String [] args) {
    System.out.println("Hello World!");
  }
}
```

It is nearly identical to the equivalent program in Java. The first difference is that JR programs must import the JR package. However, to save space, most examples in this book will omit that line; be sure to include it in any programs that you actually try to compile, though! For the same reason, our code in this book generally does not check for errors in input data or command-line arguments. The second difference is that the JR program's main method must appear in a `public` class.

As noted earlier, the sequential aspects of JR are identical to those of Java (with the exception of one extension seen in Chapter 3). However, JR provides extensions to Java to simplify the writing of concurrent programs, as the next example illustrates.

This program uses two processes to perform two independent computations:

```
public class TwoProcesses {
  private static final int [] A = { 8,  4, 11, 19};
  private static final int [] B = {14, 17,  9,  3};
  private static final int N = A.length;
  private static process p1 {
    int sum = 0;
    for (int i = 0; i < N; i++) {
      sum += A[i];
    }
```

```
    System.out.println("sum of A is " + sum);
  }
  private static process p2 {
    int sum = 0;
    for (int i = 0; i < N; i++) {
      sum += A[i] * B[i];
    }
    System.out.println("inner product of A with B is " + sum);
  }
  public static void main(String [] args) {
  }
}
```

Process `p1` computes the sum of the elements in array `A` and outputs the result; process `p2` computes the inner product of the elements in array `A` with those in array `B` and outputs the result.

This program illustrates four important aspects of JR. Chapter 4 discusses these aspects in detail.

The first aspect to note is that JR programs use the same scoping as Java programs. Consequently, each process gets its own copy of variables declared local to it (such as `sum` and `i`), but the processes *share* variables and constants (such as `A` and `B`) declared at the class level.

In this program, since the processes only read shared constants, there is no potential for both processes updating a shared variable at about the same time and interfering with each other in doing so. Such a *race condition* (or *data race*) can occur with shared variables. An example illustrating a race condition is given in Section 4.1. Processes can use synchronization to protect access to shared variables. One such technique is demonstrated in Section 1.5. Others are demonstrated in subsequent chapters; e.g., see Section 5.5 for an example of how to use only shared variables to program synchronization and see Section 6.1 for an example of how to use semaphores.

The second important aspect of JR illustrated by the `TwoProcesses` program involves the program's output. It outputs two lines, one from each process, but the order in which the lines appear is non-deterministic. The output might be `p1`'s output followed by `p2`'s output, or vice versa. Which ordering occurs depends on the order in which the two processes execute, which is also non-deterministic.

The third aspect illustrated by the `TwoProcesses` program is that the processes were declared to be `static`. Non-static processes are also allowed, but static processes are slightly simpler to use, so we use them in many of the examples in this book.

The final aspect deals with program termination. As noted in Section 1.1, a JR program terminates when all of its processes have terminated or deadlocked; it will also terminate when it has executed a `JR.exit`.

## 1.3 Matrix Multiplication

Now consider the problem of multiplying two `N` × `N` real matrices `A` and `B`. We first present a sequential program to solve this problem and then show how to modify the program to compute all $N^2$ inner products in parallel.

The following program first reads in the source matrices, then computes the matrix product, and finally prints the result matrix. (The code omits the details of reading in the matrices as that code just uses standard Java features.) The main method reads in the arrays, instantiates a `MMMultiplier` object to do the actual computation, and then invokes the `print` method in that object.

```
public class MMMain {
  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    // read in NxN arrays A and B
    ...
    MMMultiplier m = new MMMultiplier(A, B, N);
    m.print();
  }
}


public class MMMultiplier {
  int N; // A and B are NxN
  double [][] C;
  public MMMultiplier(double [][] A, double [][] B, int N) {
    this.N = N;
    C = new double [N][N];
    // compute NxN inner products
    for (int r = 0; r < N; r++) {
      for (int c = 0; c < N; c++) {
        C[r][c] = 0.0;
        for (int k = 0; k < N; k++) {
          C[r][c]+= A[r][k] * B[k][c];
        }
      }
    }
  }
  public void print() {
    // output C
    for (int r = 0; r < N; r++) {
      for (int c = 0; c < N; c++) {
        System.out.print(C[r][c]+" ");
      }
      System.out.println();
    }
  }
}
```

The code in `MMMultiplier`'s constructor computes $n^2$ inner products using nested for statements. The inner for statement computes the inner product of row `r` of `A` and column `c` of `B` and stores the result in `C [r] [c]`. The code in the `print` method prints matrix `C`, with each row printed on a separate line.

Since the inner products are independent of each other, we can compute all $N^2$ in parallel, as shown below. This program will not be very efficient, since each process does very little computation, but we could readily modify it to use fewer processes (see Exercise 1.2 and also Chapter 15). The main class the same as the previous main class, except it uses a quiescence operation to print the result, as described later below.

```
public class MMMain {
  private static MMMultiplier m;
  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    // read in NxN arrays A and B
    ...
    m = new MMMultiplier(A, B, N);
    // register done as the quiescence operation
    try {
      JR.registerQuiescenceAction(done);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
  }
  private static op void done() {
    m.print();
  }
}
```

The `MMMultiplier` code now performs the matrix multiplication by using `compute` processes.

```
public class MMMultiplier {
  int N; // A and B are NxN
  double [][] A, B, C;
  public MMMultiplier(double [][] A, double [][] B, int N) {
    this.A = A; this.B = B; this.N = N;
    C = new double [N][N];
  }
  process compute ( (int r = 0; r < N; r++),
                    (int c = 0; c < N; c++) ) {
    // compute the inner product for C[r,c]
    C[r][c] = 0.0;
    for (int k = 0; k < N; k++) {
      C[r][c]+= A[r][k] * B[k][c];
    }
```

```
  }
  public void print() {
    // output C --- code same as before
  }
}
```

The heading on `compute` contains two quantifiers, so $N^2$ processes are created, one for each combination of values for `r` and `c`. In fact, `r` and `c` are parameters to each instance of `compute` and are available in `compute`'s body. Each process computes one inner product, just as each iteration of the innermost loop does in the sequential program. The `compute` processes are created at the end of execution of `MMMultiplier`'s constructor.

When inner products are computed in parallel, `C` should not be printed out until all processes have terminated. As mentioned in Section 1.1, a program may register a quiescence operation, which is invoked when JR has detected that the program has finished computation and is about to terminate. Hence, the code associated with the quiescence operation is executed *after* the rest of the computation terminates. In the program above, the main method registers `done` as the program's quiescence operation. Once the `compute` processes terminate, the code in `done` is executed to print out `C`. By using a quiescence operation, we do not need to add synchronization code to the rest of the program to determine when all the `compute` processes have terminated. This feature of JR makes many programs, including this one, easy to write. Chapter 15 describes how to structure solutions to this problem in ways that do not require using a quiescence operation.

## 1.4 Concurrent File Search

The programs given so far are very short, so they consist of a single class. Often it is best to employ multiple classes. The last two examples in this chapter illustrate how to do so.

The `grep` family of UNIX commands is commonly used to search for patterns in files. For example, the following command searches each of the named files:

```
grep string filename1 filename2 ...
```

Each line containing `string` is printed on standard output. (If there is more than one file, each line of output begins with the name of the file.) The grep command searches each file sequentially.

The following JR program gives a simplified, concurrent implementation of the above command. In particular, it searches the files in parallel, one process for each file. The program has the same arguments as `grep` above: a pattern string and one or more file names. (It does not implement the `grep` command's other useful features, such as searching for strings matching patterns specified by regular expressions; see Exercise 1.5.) Like `grep`, the program prints all

lines that contain the pattern string on the standard output. A string containing the file name concatenated with a colon is printed at the front of each line. Since searching and printing proceed in parallel, however, lines from different files will be interleaved.

The program consists of two classes. Execution begins in the `grepmain` class, which creates a `grepworker` object for each filename given on the command line.

```
public class GrepMain {
  public static void main(String [] args) {
    if (args.length < 2) {
      System.err.println(
        "needs arguments: pattern filename {filename}");
      JR.exit(1);
    }
    String pattern = args[0];
    // create a GrepWorker object for each filename
    for (int k = 1; k < args.length; k++) {
      new GrepWorker(pattern, args[k]);
    }
  }
}
```

The constructor for class `grepworker` has two parameters: `pattern` and `filename`. It saves the parameters into object variables. When the constructor is done executing, the new in `grepmain` completes and an instance of process `search` in the newly instantiated `grepworker` object is created implicitly. The `search` process finds all instances of `pattern` in `filename` and writes them out; the file name and a colon are printed at the front of each line.

```
import java.io.*;
public class GrepWorker {
  String pattern, filename;
  public GrepWorker(String pattern, String filename) {
    this.pattern = pattern;
    this.filename = filename;
  }
  private process search {
    try {
      FileReader fr = new FileReader(filename);
      BufferedReader br = new BufferedReader(fr);
      String line;
      while ((line = br.readLine()) != null) { // get null on EOF
        if (line.indexOf(pattern) >= 0) {
          System.out.println(filename + ":" + line);
        }
      }
      fr.close();
```

```
    } catch (FileNotFoundException fe) {
      System.err.println("can't open " + filename);
    } catch (IOException ioe) {
      System.err.println("IO Exception for " + filename);
    }
  }
}
```

All objects in the above program execute on the same machine. However, we can readily modify the program so that different instances of `grepworker` execute on potentially different machines. For example, suppose a file name is specified on the command line as `machine:filename`. Also, suppose that `main` separates `machine` from `filename` and stores the values in string variables with those names. Then `main` can create a `grepworker` object on machine by executing

```
vm vmcap;
vmcap = new vm() on machine;
new remote GrepWorker(pattern, filename) on vmcap;
```

A `vm` in JR is a virtual machine (address space). The first line declares a reference for a `vm`. The second line creates a new `vm` on the machine whose name is stored in variable `machine`. The third line creates an instance of `grepworker` on the newly created `vm`, and hence on a potentially remote machine (as indicated by the `remote` keyword). The effect of making the above changes is that each `grep` object will open `filename` on the machine on which it is executing. (This program assumes that, for reasons explained in Section 10.8, the names of the files to be searched are specified as relative to home directory or are specified as absolute pathnames on the remote machine.)

## 1.5 Critical Section Simulation

As a final example, we present a program that illustrates a few of the numerous message-passing mechanisms available in JR. The program also illustrates how one can construct a simple simulation of a solution to a synchronization problem.

The following program contains `numusers` instances of a `user` process, each of which repeatedly executes a critical section of code and then a non-critical section. At most one process at a time is permitted to execute its critical section. If more than one process wants to enter its critical section at the same time, the one with the highest priority is permitted to do so. Each `user` process has an index `i`; the lower the index value, the higher the priority of the process. We simulate the duration of critical and non-critical sections of code by having each `user` process "nap" for a random number of milliseconds.

```
import java.util.Random;
public class CSS {
```

```
  private static op void CSenter(int);
  private static op void CSexit();
  private static process arbitrator {
    while (true) {
      inni void CSenter(int id) by id {
        System.out.println("user " + id + " in its CS at " +
                           System.currentTimeMillis());
      }
      receive CSexit();
    }
  }
  private static final int numusers = 3, rounds = 4;
  public static void main(String [] args) {
  }
  private static process user( (int i = 1; i <= numusers; i++) ) {
    Random r = new Random();  // seed with system time
    for (int j = 1; j <= rounds; j++) {
      call CSenter(i);                 // enter critical section
      try {
        Thread.sleep(r.nextInt(100));  // delay up to 100 msec
      } catch (Exception e) {e.printStackTrace();}
      send CSexit();                   // exit critical section
      try {
        Thread.sleep(r.nextInt(1000)); // delay up to 1 second
      } catch (Exception e) {e.printStackTrace();}
    }
  }
}
```

The `CSS` class contains an `arbitrator` process that implements two operations: `CSenter` and `CSexit`. It first uses an input statement (`inni`) to wait for an invocation of `CSenter`. This is JR's rendezvous mechanism. If there is more than one invocation of `CSenter`, the one that has the smallest value for parameter `id` is selected, and a message is then printed. Next the `arbitrator` uses a receive statement to wait for an invocation of `CSexit`. Receive is a special case of `inni` that can be used when one just needs to receive a message or, in this case, simply a signal.

Each `user` process calls the `CSenter` operation to get permission to enter its critical section, passing its index `i` as an argument. After "napping" the process then invokes the `CSexit` operation. The `CSenter` operation must be invoked by a synchronous call statement because the `user` process has to wait to get permission. However, since a `user` process does not need to delay when leaving its critical section, it invokes the `CSexit` operation by means of the asynchronous send statement.

The program employs several methods in Java packages. The `System.currentTimeMillis` method in the print statement returns the number of milliseconds since a particular epoch. The `Thread.sleep` method causes

a process to "nap" for the number of milliseconds specified by its argument. The `nextInt` method in the `Random` class returns a pseudo-random integer between 0 and its argument.

## 1.6 Translating and Executing JR Programs

To execute a JR program, one must first create one or more files containing the program text. The names of these files must end with `.jr`. Following Java requirements, the JR class `x` must be placed in the file `x.jr` if `x` is public. For example, the "Hello world" program must be placed in a file named `HelloWorld.jr`.

The standard tool to translate and execute a JR program is `jr`. Assuming the directory containing `jr` is in a particular user's search path, the user can compile and execute the program in `HelloWorld.jr` by using the command

```
jr HelloWorld
```

Note that the `.jr` suffix does *not* appear in this command; only the name of the class containing the main method does. The `jr` command assumes that all `.jr` files in the current directory are part of the program. After the main class name, additional arguments to `jr` are the command-line arguments to be passed to the main method.

The `jr` command performs several actions. Assuming no errors, it invokes each of the following:

- the JR compiler (`jrc`) to generate Java code for each `.jr` file (it also generates additional Java classes as needed by the program);
- the Java compiler to translate that code to bytecode;
- the RMI compiler to adapt the translated code to execute with RMI (which JR uses to distribute programs); and
- the JVM (Java Virtual Machine) to run the translated bytecode.

The `jr` command creates in the current directory a new subdirectory named `jrGen` (first deleting the old one if it already exists). It uses this directory for all the files created by the above steps, e.g., `.java` and `.class` files. The programmer should have no need to be concerned with the contents of these files but also should not modify them. However, `jrGen` is needed to run a program, so it should not be deleted by the user if the program is to be executed several times.

Several other JR tools provide flexibility in applying the above steps. The following table summarizes these tools:

| | |
|---|---|
| `jr` | translates and executes a JR program |
| `jrc` | translates a JR program |
| `jr_rmic` | adapts JR-translated Java code to execute with RMI |
| `jrrun` | executes an already-translated JR program |
| `jrgo` | like `jr`, but tries to determine the name of main class |
| `jrgox` | like `jrrun`, but tries to determine the name of main class |

See Appendix C for further details on developing and executing JR programs.

## 1.7 Vocabulary and Notation

We begin by explaining the notation and conventions we will be using in the remainder of the book. As already seen in this chapter, we typeset JR syntactic tokens and programs in the Courier (typewriter) typeface.

JR's syntax extends Java's syntax with additional statements and forms of declarations and expressions; these extensions introduce several new keywords. The specific keywords and syntax will be introduced as we describe the various language mechanisms. The exact syntax and the complete set of keywords is given in Appendix A.

We will present the syntax of the JR extensions in a form in which each syntax display conveys what an element of the JR grammar looks like in a program. For example, consider how we might describe the syntax of a simplified version of JR's receive statement. (Chapter 7 gives the full description.) A receive statement names an operation and gives a list of zero or more variables separated by commas. It has the following general form:

**`receive`** *op_id* **`(`** *variable*, *variable*, ... **`)`**

The keyword `receive`, the parentheses, and the commas are JR tokens, so they are typeset in Courier. The items *op_id* and *variable* are non-terminals in the JR grammar. When an item such as *variable* can be repeated, we will always list two instances and two separators and follow them with an ellipsis. We will also say whether there must be zero or more or one or more instances of the item.

We will when possible follow common Java terminology in presenting JR syntax. The key syntactic items we will use are summarized in the following table.

| | |
|---|---|
| *block* | a block of zero or more statements enclosed within { and } |
| *expr* | an expression |
| *id* | an identifier |
| *variable* | a variable |

## Exercises

⇒ As noted in the Preface, source code for all programming examples and the "given" parts of the programming exercises are available on the JR webpage.

1.1 Execute the `TwoProcesses` program several times to see whether the order of output differs between executions. If not, then add an invocation of `Thread.sleep` to force the other order of output.

1.2 Add to the `TwoProcesses` program a third process, which is to find the maximum element in both of the arrays.

1.3 (a) Compare the execution times of the sequential and parallel matrix multiplication programs for various size matrices. Which is more efficient?

(b) Modify the parallel program so that it uses only `N` processes, each of which computes one row of result matrix `C`. Compare the performance of this program to your answers to part (a).

1.4 (a) Execute the concurrent file search program using different patterns and files on a UNIX system. Compare the output to that of the `grep` command. Now try piping the output of your JR program through the `sort` command, and compare the output to that of `grep`. What happens if the file-name arguments to your JR program are given in alphabetical order?

(b) Modify the program to create instances of `grep` on different machines, as described in Section 1.4. Experiment with this version of the program.

1.5 Modify the concurrent file search program so that it allows the search string to be a regular expression. To save yourself a lot of work, use an existing Java regular expression package like `gnu.regexp`.

1.6 Execute the critical section simulation program several times and examine the results. Also experiment with different nap intervals by modifying the argument to the `nextInt` method. Modify the program by deleting the phrase `by id` in the `arbitrator` process, and execute this version of the program several times. How do the results compare to that of the original program? What if `by id` is replaced by `by -id`?

PART I

# EXTENSIONS FOR CONCURRENCY

This part of the text introduces JR's mechanisms for concurrent programming. JR extends Java with SR-like [9] concurrency mechanisms. (Much of what we say about JR below applies equally well to SR; Appendix E summarizes the differences.) JR is rich in the functionality it provides: dynamic process creation, semaphores, message passing, remote procedure call, and rendezvous. All are variations on ways to invoke and service operations. JR also provides easy-to-use ways to construct distributed programs.

We describe the concurrent aspects of JR in a bottom-up manner, from simpler mechanisms to more powerful ones. This also follows the historical order in which the various concurrent programming mechanisms that appear in JR were first developed. While reading these chapters, keep in mind that all the process interaction mechanisms are based on invoking and servicing operations. Chapter 2 first gives a brief overview of JR's extensions for concurrency. Chapter 3 introduces op-methods, operation declarations, and operation capabilities; because these mechanisms are so fundamental to JR, it focuses on just their sequential aspects. Chapter 4 describes process creation and execution. Chapter 5 presents synchronization using shared variables; although this kind of synchronization requires no additional language mechanisms, it does show one low-level way in which processes can interact. Chapter 6 discusses how semaphores are declared and used. Chapter 7 introduces the mechanisms for asynchronous message passing. Chapter 8 describes remote procedure call, and Chapter 9 describes rendezvous. Chapter 10 presents the notion of a virtual machine as an address space and shows how to create and use virtual machines. Chapter 11 describes three ways to solve the classic Dining Philosophers Problem; the solutions illustrate several combinations of uses of the mechanisms presented in the previous chapters in this part. Chapter 12 describes JR's exception handling mechanism. Chapter 13 defines and illustrates how operations can be inherited. Finally, Chapter 14 presents additional mechanisms for servicing operation invocations in more flexible ways.

*This page intentionally left blank*

Chapter 2

# OVERVIEW OF EXTENSIONS

As noted in Part I, the extensions to JR include mechanisms for processes to interact with one another and mechanisms to distribute a program across a network of machines. Below, we give an overview of these extensions. Subsequent chapters explore these topics in details.

## 2.1 Process Interactions via Operations

JR's concurrency mechanisms are variations on ways to invoke and service operations. An operation defines a communication interface; an op-method defines how invocations of that operation are to be serviced. We will see in Chapter 3 that an op-method is merely an abbreviation for an operation declaration specifying the parameterization and return value plus a method for the method body. An op-method is invoked by a call statement or function invocation. Capabilities act as pointers or references to operations.

Operations, methods, and calls are three of the bases for JR's concurrent programming mechanisms. To these we add send invocations and input statements.

JR allows construction of distributed programs in which objects can be placed on two or more machines in a network. Hence the caller of a method might be in an object on one machine, and the method itself might be in an object on another machine. In this case the call of a method is termed a *remote procedure call* (or *remote method invocation*).

When a method is called, the caller waits until the method returns. JR also provides the send statement, which can be used to *fork* a new instance of a method. Whereas a call is synchronous—the caller waits—a send is asynchronous—the sender continues. In particular, if one process invokes a method by sending to the corresponding operation, a new process is created to execute the body of the method, and then the sender and new process execute

concurrently. JR also provides process declarations, which are the concurrent programming analog of op-method declarations. A process declaration is an abbreviation for an operation declaration, a method, and a send invocation of that operation.

Processes in a concurrent program need to be able to communicate and synchronize. In JR, processes in the same object or same class can share variables and operations declared in that object or class. Processes in the same address space can also share variables and operations. Processes can also communicate by means of the input statement, which services one or more operations. A process executing an input statement delays until one of these operations is invoked, services an invocation, optionally returns results, and then continues. An invocation can either be synchronous (`call`) or asynchronous (`send`). A call produces a two-way communication plus synchronization—a *rendezvous*—between the caller and the process executing an input statement.[1] A send produces a one-way communication—i.e., *asynchronous message passing.*

To summarize, the bases for JR's concurrent programming mechanisms are operations and different ways to invoke and service them. Operations can be invoked synchronously (`call`) or asynchronously (`send`), and they can be serviced by a method or by input statements (`inni`). This yields the following four combinations:

| *Invocation* | *Service* | *Effect* |
|---|---|---|
| `call` | method | procedure (method) call (possibly remote) |
| `call` | `inni` | rendezvous |
| `send` | method | dynamic process creation |
| `send` | `inni` | asynchronous message passing |

These combinations are illustrated by the four diagrams in Figure 2.1. The squiggly lines in the diagrams indicate when a process is executing; the arrows indicate when an explicit invocation message or implicit reply message is sent.

Further discussion of most of these concurrent programming mechanisms, in a more general context, appears in Reference [7].

One virtue of JR's approach is that it supports abstraction of interfaces. In particular, JR allows the declaration of an operation to be separated from the code that services it. This allows classes to be written and used without concern for how an operation is serviced.

Another attribute of JR is that it provides abbreviations for common uses of the above interaction possibilities. We have already mentioned the op-method declaration and the process declaration, which abbreviates a common pattern of creating background processes. The receive statement abbreviates a common use of an input statement to receive a message. Semaphore declarations and

[1] For readers familiar with Ada, the input statement combines and generalizes aspects of Ada's `accept` and `select` statements.

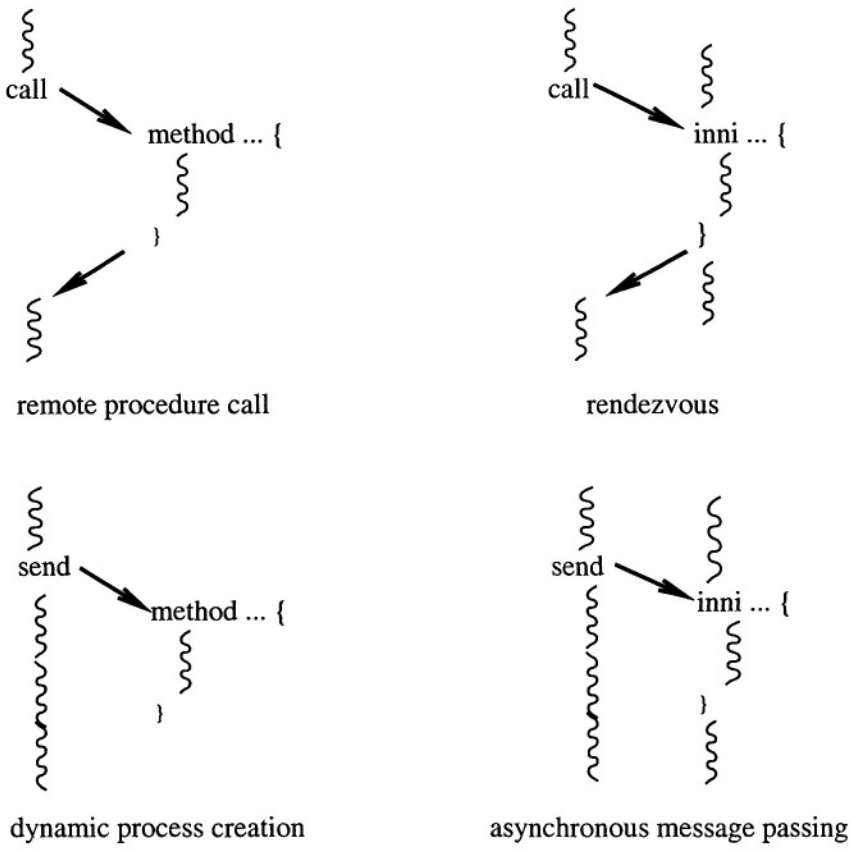


*Figure 2.1.* Process interaction mechanisms in JR

`V` and `P` statements abbreviate operations and send and receive statements that are used merely to exchange synchronization signals. In addition to these abbreviations, JR provides two additional kinds of statements that also deal with operations: forward and reply.

## 2.2 Distributing JR Programs

JR also allows the programmer to control the large-scale issues associated with concurrent programming. For constructing distributed programs, JR provides what is called a *virtual machine*—a named address space in which remote objects can be created and variables and operations can be shared. A JR program consists of one or more virtual machines. Virtual machines, like objects of classes, are created dynamically; each can be placed on a different physical machine. Communication between parts of a program located on different virtual machines is handled transparently.

Processes in a distributed program need to be able to communicate, and in many applications communication paths vary dynamically. This is supported in JR by operation capabilities, which are introduced in Chapter 3, and remote object references, which were introduced in Chapter 1. An operation capability is a pointer to a specific operation; a remote object reference is a pointer to all the operations made public by the object. These can be passed as parameters and hence included in messages.

Chapter 3

# OP-METHODS, OPERATIONS, AND CAPABILITIES

This chapter examines how *op-methods* are declared and invoked. We shall see that the mechanism for defining an op-method is really an abbreviation that involves two more general mechanisms: an operation declaration and a method. This chapter also introduces operation capabilities, which serve as pointers or references to operations. The general mechanisms introduced in this chapter—i.e., operation declarations, op-methods, and operation capabilities—are also used in concurrent programming. Because these mechanisms are so fundamental to JR, however, this chapter focuses on just their sequential aspects; later chapters extend these mechanisms by examining their concurrent aspects.

## 3.1 Op-methods

An op-method declaration in JR has the same form as a method declaration in Java, except the former includes the extra keyword `op`. An op-method can be invoked in the same ways as a method can in Java, either as a separate expression or part of a larger expression. In addition, an op-method can be invoked via a `call` statement. All of these kinds of invocations are known as *call invocations.* (Later chapters will introduce a `send` statement, which is used in *send invocations.*) A call invocation is, for the present chapter, equivalent to a regular Java method invocation; later chapters will describe the additional semantics for call invocations when they are used in concurrent programming. As in Java, invocation parameters are evaluated in left-to-right order.

As a basic example of an op-method and its use, consider the following code:

```
public class Basic {
  private static op int square(int x) {
    System.out.println("in square "+x);
    return x*x;
  }
```

```
  public static void main(String [] args) {
    System.out.println(square(23));
    square(41);
    call square(41);
  }
}
```

`main` makes three call invocations. The first invocation's value is used as an argument to the print method. The other two invocations discard the return value; they are equivalent.

## 3.2 Operation and Method Declarations

An op-method declaration is really an abbreviation for an operation declaration and an ordinary Java method. An op-method declaration can be used in all cases, but it is helpful in understanding the material in later chapters to see the underlying mechanism here.

An operation declaration essentially gives the types of the parameters and the return value. So, the `square` op-method from the previous section can be written equivalently as

```
private static op int square(int);
private static int square(int x) {
  System.out.println("in square "+x);
  return x*x;
}
```

The method is said to *service* invocations of the operation.

The reason for having a separate operation declaration is that, as introduced in Part I, invocations can be serviced in an additional way, with `inni` statements. This additional form of servicing requires the declaration to be visible to invokers, even though the servicing statements are not. Also, arrays of operations are permitted. (See Chapter 9.)

## 3.3 Operation Capabilities

An operation capability is a pointer to (or reference to) an operation.[1] Such pointers can be assigned to variables, passed as parameters, and used in invocation statements; invoking a capability has the effect of invoking the operation to which it points. A variable or parameter is defined to be an operation capability by declaring its type in the following way:

`cap` *operation_specification cap_id*

The capability is defined to have the parameterization and return type in the operation specification. The operation specification is similar to the method

[1]Java has no function pointers or references, but the effect can be simulated via using inner classes.

return type and signature parts of a method header in Java, but it omits the name of the method. (It may also contain `throws` clauses; see Chapter 12.)

An operation capability can be bound to any user-defined operation having the same parameterization[2]. When parameterization is compared, only the signatures of formals and return values matter; formal and return identifiers are ignored. Capabilities can also be compared using the == and ! = relational operators; however, the other relational operators (e.g., <) are not allowed for capabilities since no ordering is defined among them.

Some simple examples will illustrate the declaration and use of operation capabilities. The following partial program shows examples of how to declare and use capability variables:

```
public class Simple {
  // declare a few operations
  private static op void d(int);
  private static op int e(int);
  private static op double f(double);
  private static op double g(double);
  // declare a few capabilities
  private static cap void (int) x, z;
  private static cap double (double) y;

  public static void main(String [] args) {
    x = d;  // x now points to operation d
    x(387); // invoke operation d with argument 387
    // make y point to one of f or g
    if (e(9) > 0) { y = f; }
    else          { y = g; }
    // invoke what y points to
    System.out.println(y(4.351));
    // capabilities can be assigned and compared
    z = x;
    if (y == f) { System.out.println("y is f"); }
    else        { System.out.println("y is g"); }
  }
}
```

However, the following assignments are illegal for the reasons indicated:

```
  x = e;   // e returns an int; x is a cap for a void
  f = g;   // f is an operation; cannot assign to it
  z(3,45); // z requires a single argument
```

[2]Some matching on `throw` clauses (for exception handling) is also required, but we will defer discussing that topic until Chapter 12.

Note how the name of an operation in effect acts as a capability constant. It cannot be assigned to, but its value can be used to invoke an operation, assigned to capability variables, and compared with other capabilities.

As a more realistic example, consider the following program, which defines an op-method, `trapezoidal`, that approximates the area under a curve (function) by means of the trapezoidal rule. The op-method has four parameters. The first three specify the end points and number of intervals to use. The fourth is a capability for the function that defines the curve. This op-method might be used as follows:

```
public class TrapRule {
  // return area under the curve f(x) for a <= x <= b
  // using trapezoidal rule with n intervals
  private static op double trapezoidal(double a, double b, int n,
                                       cap double (double) f) {
    double area = 0;
    double x = a;
    double h = (b-a)/n;
    area = (f(a)+f(b))/2;
    for (int i = 1; i <= n-1; i++) {
      x += h; area += f(x);
    }
    area *= h;
    return area;
  }
  private static op double fun1(double x) {
    return x*x + 2*x +4;
  }
  private static op double fun2(double x) {
    return Math.sin(2*x);
  }
  public static void main(String [] args) {
    // output area under some curves
    System.out.println(trapezoidal(0,1,200,fun1));
    System.out.println(trapezoidal(0,Math.PI/2,1000,fun2));
  }
}
```

The first invocation of `trapezoidal` will find the area under `fun1` between 0 and 1 using 200 intervals. The second will find the area under `fun2` between 0 and $\pi/2$ using 1000 intervals.

Arrays of capabilities can also be declared, following the normal Java style for declaring arrays. However, we defer the discussion of such arrays until later chapters where they are needed. See Section 6.1 and Section 9.8.

Capability variables can also take on two special values: `null` and `noop`. Invocation of a capability variable whose value is `null` causes a run-time error. In general, invocation of a capability variable whose value is `noop` has no effect.

However, arguments in the invocation are evaluated so any side effects they have will occur. Also, if the invocation appears in an expression, the return value is undefined.

As an example of the use of `noop`, suppose we are writing a terminal-independent program that has a screen repaint command. We want the user to be able to invoke the repaint command by simply executing

```
repaintcap();
```

For terminals with screens, we would set `repaintcap` to the name of the op-method that will actually repaint the screen. For a paper terminal, however, invocations of the repaint command should be ignored, so `repaintcap` would be set to `noop`. The advantage of using `noop` is that it simplifies code by eliminating testing for special cases, which might depend on information that is not readily available where it is required.

We shall see later that `null` is useful if we want to make sure that a capability variable has a value. Operation capabilities are also useful in programs containing multiple classes and in distributed programs, as we shall see in later chapters.

## Exercises

3.1 Consider the following capability declarations and which operations can be assigned to each.

```
private static cap int (char) z;
private static cap void () a;
private static cap double (int) b;
private static cap int (char, double) c;
private static cap int (cap double (char)) d;
private static cap cap void () (int) e;
private static cap cap int (boolean) (cap double (char)) f;
```

For example, `z` can be assigned any operation that takes a single character parameter and returns an integer. In a similar style, describe each of the other capabilities.

3.2 Show the output from program `Simple` in Section 3.3 given the following method declarations:

```
private static void d(int x) {System.out.println("d "+x); }
private static int e(int x) { return -x;}
private static double f(double x) { return x*10; }
private static double g(double x) { return 10000-x; }
```

3.3 Suppose we are writing an array-sorting program. For large arrays we want to use quicksort, but for small arrays we want to use selection sort. An outline for the program follows:

```
public class Sort {
  private static op int [] selectionsort(int [] a) {
    // selection sort code
    return a;
  }
  private static op int [] quicksort(int [] a) {
    // quicksort code
    return a;
  }
  public static void main(String [] args) {
    // read values for n and x
    int n;    // size of x
    int [] x; // array to sort
    ...
    final int threshold = ...; // some value
    int [] y; // array to hold sorted x
    if (n <= threshold) y = selectionsort(x);
    else y = quicksort(x);
    // print results (y)
    ...
  }
}
```

Rewrite this program to use a capability variable to invoke the appropriate sorting routine. In particular, there should be only one call invocation.

3.4 Write a sort op-method that takes an array of integers and an operation capability for the comparison method that sort is to use. For example, the sort method might be invoked as:

```
y = sort(a,greaterthan);
z = sort(b,lessthan);
```

The second parameters in these invocations, `greaterthan` and `lessthan`, are operations whose names indicate the "sense" of the comparison that `sort` is to use. Provide a complete program that tests `sort` on a few sample invocations.

Chapter 4

# CONCURRENT EXECUTION

Processes are at the heart of concurrent programming. They represent independent threads of control, each of which executes sequential code. Sequential programs contain just a single thread of control; concurrent programs contain multiple threads of control. Most of the programs in this book are concurrent.

This chapter describes the JR mechanisms for creating processes, all of which are based on operations. We first describe process declarations, which provide a simple way to create single processes and families (arrays) of processes. Process declarations are actually an abbreviation for JR's more general process-creation mechanism: sending to an operation that is implemented by a method. JR provides both static and non-static processes; for brevity of exposition, the examples in this book generally use static processes where possible, although non-static processes are also very useful. The next section describes process scheduling and priorities. The final section describes JR's automatic program termination detection.

## 4.1 Process Declarations

We have already seen that a class can contain methods and op-methods. It can also contain processes. The following is the simplest form of a process:

**process** *process_id block*

As usual, a *block* consists of zero or more statements enclosed within { and }.

A process declaration can also contain a list of one or more quantifiers to specify multiple instances of the same process. Such a family (array) of processes has the form

**process** *process_id* **(** *(quantifier)* **,** *(quantifier)* **,** ... **)** *block*

Note that the set of quantifiers is enclosed by one pair of parentheses and that each quantifier is enclosed in another pair. A quantifier has the form

( *initial_expr* ; *continue_expr* ; *increment_expr* )
or
( *initial_expr* ; *continue_expr* ; *increment_expr* ; *such-that_expr* )

Thus, a quantifier has the same form as the group of control expressions that appear in `for` statements, except it may contain a fourth expression. Also, the first expression is required and must specify a new variable, e.g., `int i = 0` (see Exercise 4.13). The quantifiers imply loops to create the specified processes. The extra, boolean *such-that_expr* is evaluated on each iteration of this implied loop and a process is created on that iteration only if the expression evaluated to true. This section and the next give examples.

All processes are created automatically, but when a particular process is created depends on whether or not the process's declaration is static. A static process is created when its enclosing class is created. A non-static process is created when an object of the class is created. Details appear in Section 4.2. For process declarations that contain quantifiers, one instance of the process is created for each combination of values of the bound variables. Each instance of the process has access to the associated values of the bound variables; these are implicitly passed as arguments to the instance.

A JR program terminates when its processes have terminated or deadlocked or when it has executed a `JR.exit`. Although Java's `System.exit` also shuts down program execution for single virtual machine JR programs, such as those seen in this chapter, it will not cleanly shut down program execution for multiple virtual machine JR programs. So using `JR.exit`, when explicit program termination is needed, is a good practice to get into.

As a simple example of a program that uses processes, consider the following:

```
public class Foo {
  private static int x = 0;
  private static process p1 {
    x += 3;
    System.out.println("in p1");
  }
  private static process p2 {
    x += 4;
    System.out.println("in p2");
  }
  public static void main(String [] args) {
  }
}
```

This program contains two processes, `p1` and `p2`. When the program begins execution, first `x` is initialized, then `p1` and `p2` are created. The processes execute at the same time, at least conceptually. Above, each process unsafely accesses variable `x`. Such shared variables are not automatically protected from concurrent access; mutual exclusion must be programmed explicitly, e.g., using

shared variable programming techniques (see Chapter 5). or using semaphores (see Chapter 6).

The order in which processes execute is non-deterministic. Thus in the above example, the order in which processes `p1` and `p2` execute their assignments is not known. Similarly, the order in which they execute their prints is also non-deterministic. However, the output from one print will not be interleaved with the output from the other.

The above program has a potential *race condition* in its access to shared variable `x`. The two processes can access `x` at about the same time in such a way as to not affect its value as desired. For example, at the end of execution, `x`'s value can be 3, 4, or 7. To see how, consider a variant of the above program:

```
public class Race {
  private static int x = 0;
  private static process p1 {
    int y = x;
    y += 3;
    x = y;
    System.out.println("in p1");
  }
  private static process p2 {
    int y = x;
    y += 4;
    x = y;
    System.out.println("in p2");
  }
  public static void main(String [] args) {
  }
}
```

It is, in effect, equivalent to the original program. Suppose that the program is run on a single processor system, so that the executions of `p1` and `p2` are interleaved. Then, the following execution ordering can occur:

1 `p1` reads `x`'s value (0) and stores 0 into its copy of `y`

2 `p1` increments its `y` by 3

3 `p2` reads `x`'s value (0) and stores 0 into its copy of `y`

4 `p2` increments its `y` by 4

5 `p2` stores its `y` (4) to `x` (now 4)

6 `p1` stores its `y` (3) `x` (now 3)

The `Race` program mirrors closely what can happen at the machine level representation of the program when the original program is run on a single processor

system. The extra, local variable `y` corresponds to a register. The three assignment statements correspond, respectively, to reading into a register from memory, incrementing the register, and writing from the register back to memory. Each of these activities is *atomic,* i.e., indivisible. The interleaving of the processes' execution is accomplished by a *context switch*, which saves the current process's state (which includes values in registers) and restores another process's state. In the above, that occurs between steps 2 and 3.

Processes can also be declared as non-static as shown in the following variant of the above program:

```
public class MainFoo1 {
  public static void main(String [] args) {
    new Foo1();
  }
}

public class Foo1 {
  private int x = 0;
  private process p1 {
    x += 3;
    System.out.println("in p1");
  }
  private process p2 {
    x += 4;
    System.out.println("in p2");
  }
}
```

Here, the main program instantiates a `Foo1` object. The output from this program is the same as the earlier program. Of course, multiple `Foo1` objects can be created. Each `Foo1` object has its own instances of variable `x` and processes `p1` and `p2`. If variable `x` were declared to be static, then all processes would share that single variable. (See Exercise 4.6.)

As an example of a process declaration with quantifiers, consider the following program:

```
public class Quant {
  private static process p( (int i = 0; i < 8; i++) ) {
    System.out.println("in p "+i);
  }
  private static process q( (int i = 1; i <= 5; i++; i != 3) ) {
    System.out.println("in q "+i);
  }
  public static void main(String [] args) {
  }
}
```

It creates eight instances of process `p` and four instances of process `q`. Each instance of process `p` is given its own local variable `i`, whose initial values are

from 0 through 7. Each instance of process q is given its own local variable i, whose initial values are 1, 2, 4, and 5.

As another example of process declarations with quantifiers, consider again the matrix multiplication example presented in Section 1.3. Here is the relevant portion of that code:

```
process compute ( (int r = 0; r < N; r++),
                  (int c = 0; c < N; c++) ) {
  // compute the inner product for C[r,c]
  C[r][c] = 0.0;
  for (int k = 0; k < N; k++) {
    C[r][c]+= A[r][k] * B[k][c];
  }
}
```

It employs an N × N array-like family of processes to perform matrix multiplication. Each process computes one element in the result matrix. The N*N instances of compute are created as part of creating the enclosing object. Each instance of compute can determine its own "identity" through r and c; these are different in each instance. Since instances of compute only read values from A and B and write disjoint parts of C, these variables can safely be accessed concurrently.

## 4.2 The Unabbreviated Form of Processes

The process abbreviation given in the previous section is handy when the number of instances to create is known in advance and when processes are to be created at the same time the class is loaded or the object is created. In some cases, though, processes need to be created as a program executes, say in response to input or to actions occurring in other parts of the program. To understand the general mechanism for creating processes, it is first useful to examine the constituent pieces of the process abbreviation.

A process declaration is really an abbreviation for an operation declaration, a method, and a send invocation. The difference between a send invocation and a call invocation is that a send is asynchronous (i.e., non-blocking) whereas a call is synchronous (i.e., blocking). That is, a send does not wait for the invoked method to return any results; it terminates immediately after passing the arguments to the method. A new process is created to execute the method; it executes in parallel with the process that executed send. Figure 2.1 summarizes these actions.

Program Foo in Section 4.1 can be written equivalently, without employing process declarations, as follows:

```
public class FooUnabbrev {
  private static int x = 0;
  private static op void p1();
```

```
  static {
    send p1();
  }
  private static op void p2();
  static {
    send p2();
  }
  private static void p1() {
    x += 3;
    System.out.println("in p1");
  }
  private static void p2() {
    x += 4;
    System.out.println("in p2");
  }
  public static void main(String [] args) {
  }
}
```

The changes are that `p1` and `p2` are now declared as operations, and their code is now written as methods. (Note that `p1` and `p2` can be written using the op-method abbreviation, but the expanded form is more fundamental.) Also, `Foo` now contains static initializers with explicit sends to create an instance of each process.

If the programmer is writing code to simulate the process abbreviation, the creation code does not need to be placed in static initializers. For example, the above program is equivalent to one without static initializers but in which the sends are placed at the beginning of the main method. (See Exercise 4.8).

Programs with non-static processes are transformed similarly, but the details differ. Program `Foo1` in Section 4.1 can be written equivalently as follows:

```
public class Foo1Unabbrev {
  private int x = 0;
  private op void p1();
  private op void p2();
  public Foo1Unabbrev() { // new constructor
    send p1();
    send p2();
  }
  private void p1() {
    x += 3;
    System.out.println("in p1");
  }
  private void p2() {
    x += 4;
    System.out.println("in p2");
  }
}
```

The code for main is identical to the code before, except it now creates a `Foo1Unabbrev` object. As before, `p1` and `p2` are now declared as operations, and their code is now written as methods. However, a new constructor now contains explicit sends to create an instance of each process. (Those sends are placed at the end of each constructor, if any exist, or, as above, placed within the implicit constructor, if no other constructors exist.)

Programs with quantified processes use loops to create the process instances. Program `Quant` in Section 4.1 can be written equivalently as follows:

```
public class QuantUnabbrev {
  private static op void p(int);
  static {
    for (int i = 0; i < 8; i++) {
      send p(i);
    }
  }
  private static void p(int i) {
    System.out.println("in p "+i);
  }
  private static op void q(int);
  static {
    for (int i = 1; i <= 5; i++) {
      if (i != 3) send q(i);
    }
  }
  private static void q(int i) {
    System.out.println("in q "+i);
  }
  public static void main(String [] args) {
  }
}
```

Note how the loop for creating instances of process `q` includes an if statement implied by the fourth expression in the quantifier. Clearly, it is simpler to use the process abbreviation whenever possible!

As another example of how processes can be created dynamically, consider the following class:

```
public class Compressor {
  public op void compress(String);
  public void compress(String filename) {
    // open file filename, compress its contents
    // and then close it.
  }
}
```

It declares a public operation, `compress`, which is to be used from another class. A `compressor` object can be created and the `compress` operation may be invoked by executing

```
Compressor c = new Compressor();
send c.compress("data1");
send c.compress("data2");
```

Each invocation of compress causes a new process to be created within `Compressor`. The invoker does not wait for that process to terminate before it continues executing. A process declaration could not be used in cases like this because the invocation that causes processes to be created occurs outside the class.

Since a process declaration is actually an abbreviation for an operation declaration and a method, as seen in the above examples, additional processes may be created dynamically by explicit send invocations. For example, suppose the programmer initially wants two instances of a process `p`. This can be specified by a process declaration:

```
public process p( (int id = 1; id <= 2; id++) ) {
  // body of p
}
```

Later, an additional instance of `p` can be created by, for example, executing

```
send p(17);
```

The value of `id` in the new instance will be 17.

## 4.3 Static and Non-static Processes

As seen in the previous sections, processes can be declared as static or non-static. How processes are declared can be mixed within the same class. For example, consider the following program:

```
public class Mixed {
  private static process server {
    System.out.println("in server");
  }
  private process client {
    System.out.println("in client " + id);
  }
  private int id;
  Mixed(int id) { this.id = id; }
  public static void main(String [] args) {
    // create two objects, each with its own client process
    new Mixed(1);
    new Mixed(2);
  }
}
```

One instance of `Mixed`'s `server` process is created statically; one instance of `Mixed`'s `client` process is created for each `Mixed` object.

This structure is useful for some client-server interactions. The client and server can communicate, e.g., via operations as seen later in Chapter 9. This flexibility in mixing static and non-static processes can be useful. However, the programmer should be careful to be sure to specify static whenever that is intended. For example, consider a variant of the above program in which the programmer intends process `client` to be static—but mistakenly omits the `static` keyword—and creates no `Mixed` objects. Then, only the `server` process is created and the program is unlikely to work as the programmer intends! Such a scenario might be hard to debug because the programmer is likely to assume that the `client` process has been created.

As described in Section 4.2, static processes are created by static initializers, which execute before any other code in a class. Consider, for example, the following program;

```
public class BadStatic {
  private static int N;
  private static process p( (int i = 0; i < N; i++) ) {
    System.out.println("in p "+i);
  }
  public static void main(String [] args) {
    N = Integer.parseInt(args[0]);
  }
}
```

Suppose the command-line argument is `10`. On first inspection, it appears as though ten instances of process `p` will be created. In fact, no instances of process `p` are created. The problem is that `N` is not set to ten until after the static initializer that contains the code that creates the instances of `p` executes; when the static initializer executes, `N` is zero.

To get the intended behavior, the above program can use the unabbreviated form of process creation (Section 4.2) with the sends appearing in `main` *after* `N` has been set. Or, the above program can be written using non-static processes, for example, as shown in the matrix multiplication example in Section 1.3.

## 4.4 Process Scheduling and Priorities

Processes in a JR program execute concurrently, at least conceptually. Consequently, statements that access shared variables may need to be protected by critical sections to insure they execute with mutual exclusion. This can be implemented using semaphores (see Chapter 6) or the input statement (see Chapter 9).

JR processes are mapped to Java threads, so how JR processes execute is determined by the semantics for Java threads. The Java Language Specification [28] neither requires nor prohibits that Java threads execute fairly. Consider an implementation for a single processor system. It may timeslice execution

among threads, thus providing fairness. Most of the implementations of Java that the authors have used, including all of the recent implementations, use this approach. Alternatively, a Java implementation may allow a thread to run until the thread explicitly blocks on synchronization or waiting for I/O to complete, or until it yields the processor. (Java's `Thread.yield()` is not guaranteed to actually do anything, but the implementations that the authors have used do yield as expected.)

JR processes can determine and specify priorities by using the underlying Java thread mechanisms: `Thread.getPriority()` and `Thread.setPriority()`. Each JR process executes at the default thread priority level unless it explicitly sets its priority to another level. Similar to the above, the Java language specification does not require that higher-priority threads actually be given preference over lower-priority threads. Many implementations, though, do honor thread priorities.

As seen in the critical section simulation example in Section 1.5, JR processes can also use `Thread.sleep()` to yield the processor for a specified amount of time. Doing so is useful in simulations, as in the above example, and in some other situations, such as when a thread needs to update a display periodically.

Although the above Java thread methods can be used safely with the current JR implementation, other Java features dealing with threads cannot. In particular, using Java's thread synchronization mechanisms (e.g., `synchronized`, `wait`, and `notify`) or RMI within a JR program might interact oddly with the JR implementation and cause undesirable results. (In the current implementation, the only such effect is that the automatic termination detection algorithm might not work.)

## 4.5 Automatic Termination Detection

As described in Section 1.1, JR provides a means to deal with program quiescence, i.e., all processes in a program have terminated or deadlocked. Normally, JR just terminates the program. But, it also allows an operation to be registered as the "quiescence operation". This operation must have no parameters and have a `void` return type. Once quiescence is reached, the registered operation is implicitly invoked[1] and the code associated with the quiescence operation is executed, at which point the quiescence operation is implicitly unregistered. If no operation has been registered (or if `null` has been registered), then the program is simply terminated.

Note that a program in which all processes are sleeping is not considered to be quiescent. A process will eventually awaken from sleep1ng and the program

[1] For completeness, the invocation is invoked via a `send`, which was described briefly in Chapter 1 and will be described fully in Chapter 7.

will continue (whereas a process will not awaken from deadlock or termination).

The code associated with the quiescence operation may initiate other activity; e.g., it may start up new processes. If desired, the code may also reregister an operation to be the quiescence operation, which will be invoked when this new activity becomes quiescent; the operation can be the same one or a different one. Otherwise, when this new activity becomes quiescent, the program will just terminate.

We saw in the example in Section 1.3 that automatic quiescence detection is useful to avoid having to write code to determine when processes have terminated. Here's another, simpler example. It's a version of the `Race` program. The only difference is that it outputs `x`'s value at completion.

```
public class RaceQuiescent {
  private static int x = 0;
  private static process p1 {
    int y = x;
    y += 3;
    x = y;
    System.out.println("in p1");
  }
  private static process p2 {
    int y = x;
    y += 4;
    x = y;
    System.out.println("in p2");
  }
  public static void main(String [] args) {
    try {
      JR.registerQuiescenceAction(printx);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
  }
  public static op void printx() {
    System.out.println("x="+x);
  }
}
```

Note that the main method registers `printx` as the quiescence operation. That operation is implicitly invoked when both `p1` and `p2` have completed.

As a simple example of reregistering the quiescence operation, consider the following code:

```
public class FactQuiescent {
  public static int num, result = 1;
  public static void main(String [] args) {
    try {
```

```
      num = 10;
      System.out.print(num + "! = ");
      JR.registerQuiescenceAction(foo);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
  }
  public static op void foo() {
    if (num <= 1) {
      System.out.println(result);
    }
    else {
      try {
        result *= num;
        num--;
        JR.registerQuiescenceAction(foo);
      } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Its main method registers the operation `foo`. When the program becomes quiescent, `foo` is invoked; its code updates `result` and `num`, and reregisters itself (`foo`). This activity continues until `num` is one, at which point the program actually terminates. By the way, this program is *not* the recommended way to write a factorial program!

In the two programs above, the quiescence operation is serviced by a method. It can instead be serviced by JR's receive or input statements, which are described in Chapters 7 and 9. The overall effect is nearly identical.[2] Examples of this kind of quiescence operation appear in Chapter 17.

As noted above, JR's normal behavior is to terminate a program once the program becomes quiescent. This default behavior, though, can be changed via a command-line option. For programs run under the non-default behavior, it is the programmer's responsibility to terminate program execution using `JR.exit`; registering a quiescence operation has no effect on program execution. (See Appendix C for details.)

## Exercises

4.1 Consider the code for the `Foo` program. Suppose the two prints are changed as follows:

[2]The difference is that for a quiescence operation serviced by a receive or input statements, the code associated with the operation is executed only if the servicing process is already waiting for the quiescence operation to be invoked.

```
System.out.println("in p1 " + x); // changed in process p1
System.out.println("in p2 " + x); // changed in process p2
```

Give all possible outputs (including all possible output orderings) from this program.

4.2 (a) Give a step by step execution ordering (as in Section 4.1) to show how the `Race` program can end with the value 4. Give a step by step execution ordering (different from the one in Section 4.1) to show how the `Race` program can end with the value 3. Also give two different such orderings for the value 7.

(b) Run the code for the `Race` program several times to see whether a race condition actually occurs. (It may or may not depending on implementation factors.)

(c) Modify the code for the `Race` program to, in effect, force a race condition to occur and have `x` end with the value 4. Do so by placing one or more calls to `Thread.sleep` in the code.

4.3 Consider the following program.

```
public class Foo {
  private static int x = 0;
  private static process p ( (int i = 1; i <= 3; i++) ) {
    x += i+3;
  }
  public static void main(String [] args) {
    try {
      JR.registerQuiescenceAction(printx);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
  }
  public static op void printx() {
    System.out.println("x="+x);
  }
}
```

(a) Give all possible outputs from executing the given program.

(b) Suppose each process executes its body twice. Describe the set of all possible outputs in terms of your answers to the previous part.

4.4 Read in integer `n` (`n>=2`) and integer array `a`, which has `n` elements. Calculate `evenbig` and `oddbig`: the largest number in a occurring in an even position and the largest number in a occurring in an odd position, respectively. Use one process to calculate `evenbig` and another process

to calculate `oddbig`. Use quiescence to detect when those processes have calculated the two values and have completed; then calculate (from `evenbig` and `oddbig`) `anybig`, the largest in any position, and output all three values.

4.5 Read in integer `n` (`n >= 2` and `n` is even) and integer array `a`, which has `n` elements. Calculate `small1` and `small2`: the smallest number in `a` occurring in positions 0 through `(n-2)/2` and the smallest number in `a` occurring in positions `n/2` through `n-1`, respectively. Use one process to calculate `small1` and another process to calculate `small2`. Use quiescence to detect when those processes have calculated the two values and have completed; then calculate (from `small1` and `small2`) `smallest`, the smallest in any position, and output all three values.

4.6 Suppose that `MainFoo1` creates three `Foo1` objects.

(a) Show all possible outputs (including all possible output orderings). What are the values of the `x` variables right before the program terminates?

(b) Suppose variable `x` is declared as static. What is its value or its possible values right before the program terminates?

4.7 Consider the code for the `Foo` program. Show how to rewrite it using a family of two processes specified in a single quantified `process`.

4.8 Section 4.2 describes how a programmer simulating the process abbreviation may place the explicit sends in the main method versus in static initializers. Give a specific example of where putting the explicit sends at the *end* of the main method does *not* yield an equivalent program.

4.9 Write the equivalent of the declaration of the `compute` processes in Section 4.1 without using the process abbreviation.

4.10 Section 4.1 shows an excerpt of the matrix multiplication example from Section 1.3. Suppose we eliminate the `print` method (and its invocation) and instead move its code to the end of the constructor. Would the new program be equivalent to the original? Explain.

4.11 In the `Foo1Unabbrev` program in Section 4.2, suppose the `send` in the for loop were replaced by `call`. Would the program's behavior be affected? Explain.

4.12 Suppose quantifiers were not allowed to include the fourth, boolean expression.

(a) Show how to rewrite the `Quant` program (Section 4.1) so that it still uses a single `process` abbreviation for creating all of the `q` processes.

(b) Give a general technique for rewriting process declarations with four-part quantifiers using only three-part quantifiers (with possibly some changes to the body of the process).

(c) Is there any disadvantage to this new style?

4.13 As described in Section 4.1, the first expression in a quantifier must specify a new variable. The intent is that each process specified by the quantifier will get its own local variable as its "process identity", which was seen to be useful in the examples in this chapter. However, this intent can be defeated. For example, consider the following program

```
public class WeirdQuant {
  public static void main(String [] args) {}
  static int i = 0;
  // note j vs. i in quantifier
  static process worker( (int j = 0; i < 5; i++) ) {
    System.out.println("hi "+i);
  }
}
```

(a) Explain how processes are created. What variable represent each process's identity and what is that value in each process?

(b) Describe the output of the program. Be as specific as possible.

4.14 Modify program `Quant` so that

(a) it includes a quiescence operation that outputs "all done" just before the program terminates.

(b) it includes a quiescence operation that outputs (only) the indices of the `p` process and the `q` process that terminated last. (Hint: use two class variables. You may assume that a process actually terminates immediately after executing its last statement.)

4.15 The eight-queens problem is concerned with placing eight queens on a chess board in such a way that none can attack another. One queen can attack another if they are in the same row or column or are on the same diagonal.

Write a parallel program to generate all 92 solutions to the eight-queens problem. (Hint: Use a recursive procedure to try queen placements and a second procedure to check whether a given placement is acceptable.)

4.16 The quadrature problem is to approximate the area under a curve, i.e., to approximate the integral of a function. Given is a continuous, non-negative function $f(x)$ and two endpoints $l$ and $r$. The problem is to compute the area of the region bounded by $f(x)$, the $x$ axis, and the vertical lines through $l$ and $r$. The typical way to solve the problem is to subdivide the regions into a number of smaller ones, use something like a trapezoid to approximate the area of each smaller region, and then sum the areas of the smaller regions.

Write a recursive function that implements a parallel, adaptive solution to the quadrature problem. The function should have four arguments: two points $a$ and $b$ and two function values $f(a)$ and $f(b)$. It first computes the midpoint between $a$ and $b$, then computes three areas: from $a$ to $m$, $m$ to $b$, and $a$ to $b$. If the sum of the smaller two areas is within $\epsilon$ of the larger, the function returns the area. Otherwise it recursively and in parallel computes the areas of the smaller regions. Assume $\epsilon$ is a static value.

4.17 Gaussian elimination with partial pivoting is a method for reducing real matrix $m[n, n]$ to upper-triangular form. It involves iterating across the columns of $m$ and zeroing out the elements in the column below the diagonal element $m[d, d]$. This is done by performing the following three steps for each column. First, select a pivot element, which is the element in column $d$ having the largest absolute value. Second, swap row $d$ and the row containing the pivot element. Finally, for each row $r$ below the new diagonal row, subtract a multiple of row $d$ from row $r$. The multiple to use for row $r$ is $m[r, d]/m[d, d]$; subtracting this multiple of row $d$ has the effect of setting $m[r, d]$ to zero.

Write a program that implements the above algorithm. Use parallelism whenever possible. Assume every divisor is non-zero; i.e., assume the matrix is non-singular.

Chapter 5

# SYNCHRONIZATION USING SHARED VARIABLES

This chapter illustrates a primitive way to provide synchronization between processes. The technique introduced in this chapter can, for example, be used to avoid race conditions such as the one discussed in Section 4.1. The synchronization is accomplished by using only shared variables. That is, no special language mechanisms are required. In contrast, Chapter 6, for example, introduces JR's semaphore mechanism, which requires new JR declarations and statements that manipulate semaphores. The only assumption made for JR programs that use shared variables for synchronization is that reading or writing a word of memory is atomic, as described in Section 4.1.

In this kind of synchronization, processes set and test shared variables indicating their states. If one process finds that the state of another process should prevent the first process from proceeding, then it waits until the state changes. The process accomplishes such waiting by looping, repeatedly testing whether the state has changed. This kind of waiting is termed *busy waiting.* (Implementations of synchronization mechanisms such as semaphores and message passing do not require such busy waiting loops; instead they can place waiting processes on queues.)

The rest of this chapter addresses the fundamental critical section problem. This problem restricts execution of some piece of code to at most one process at a time. Section 1.5 presented a version of this problem using JR's message-passing mechanisms; the version in this chapter uses only shared variables. Further details on synchronization using shared variables and additional algorithms appear in [7], [42], and [43].

## 5.1 The Critical Section Problem

The critical section problem consists of `N` processes, each repeatedly executing the same code. The code includes a *critical section,* in which the process

might be updating a shared variable, using a printer, writing a file, etc. In general, the process is using some shared resource that can be properly used by only one process at a time. The code also includes a *non-critical section,* in which the process does not access any shared resources.

Here is an outline of the general form of a solution to the critical section problem.

```
public class GeneralForm {
  private static final int S = 4; // number of "sessions"
  private static final int N = 6; // number of processes
  private static int x = 0;
  private static process p ( (int i = 0; i < N; i++) ) {
    for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      ...
      // critical section
      x += 3;
      // exit protocol
      ...
    }
  }
  public static void main(String [] args) {
    try {
      JR.registerQuiescenceAction(done);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
  }
  private static op void done() {
    System.out.println("done: x = " + x);
  }
}
```

In this code, each process executes `S` sessions. Each session consists of non-critical section code and critical section code. The critical section code is preceded by code that obtains access to the critical section (the *entry protocol*) and is followed by code that releases access to the critical section (the *exit protocol*).

To make the examples in this chapter more concrete, the code includes a shared variable, `x`. The critical section code simply increments `x` by 3. As seen in Chapter 4, uncontrolled access to `x` can result in race conditions. Hence, access to `x` is "protected" within a critical section. (Presumably, the non-critical section does not access `x`.)

The desired properties of solutions to the critical section are, as given in Reference [7]:

CS1 Mutual exclusion: At any time, at most one process is executing in its critical section.

CS2 Absence of livelock: If two or more processes are attempting to enter their critical sections, then one will succeed.

CS3 Absence of unnecessary delay: A process attempting to enter its critical section is prevented from doing so only by another process that is executing its critical section or by other processes attempting to enter their critical sections.

CS4 Eventual entry: A process that is attempting to enter its critical section will eventually be allowed to.

The term *livelock* (CS2) is used in this chapter because the processes will be executing, but not progressing beyond their entry protocol. The term *deadlock* is used instead for synchronization mechanisms, such as semaphores (Chapter 6), for which the processes do not execute continuously, but instead block (i.e., suspend waiting to be resumed).

The careful reader will have noted a slight oddity regarding CS4 and the code outline given in the `GeneralForm` program. CS4 can be satisfied for the `GeneralForm` program by having process 0 execute its critical section `S` times in a row, then having process 1 execute its critical section `S` times in a row, etc. (See Exercise 5.3.) However, this kind of execution is likely to violate CS3 if the non-critical sections take any time to execute. In any case, the critical section problem is often stated so that processes execute infinitely, which means that the suggested execution ordering above would then violate CS4. In this chapter, we place a bound on execution (`S` sessions) so that the programs will terminate (we hope!) when we actually run them.

Section 5.5 gives a solution to the critical section problem for `N` processes. First, though, the following sections explore several other solutions as a means of better understanding this problem and the requirements of its solution. For simplicity, these proposed solutions use just two processes. To aid in the exposition, in these solutions, one process increments `x` by 3 and the other increments `x` by 4.

## 5.2 An Incorrect Solution

The following simple solution might appear on first look to be correct, but it is not.

```
public class BadCSFlag {
  private static final int S = 4; // number of "sessions"
  private static boolean busy = false;
  private static int x = 0;
```

```
  private static process p0 {
    for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      while (busy) /* do nothing */ ;
      busy = true;
      // critical section
      x += 3;
      // exit protocol
      busy = false;
    }
  }
  private static process p1 {
    for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      while (busy) /* do nothing */ ;
      busy = true;
      // critical section
      x += 4;
      // exit protocol
      busy = false;
    }
  }
  // main and done methods same as in GeneralForm
}
```

It potentially violates property CS1. Specifically, each process could find busy to be false before either sets busy to be true. So, both processes could be executing their critical sections at the same time.

The general difficulty here is that each process tests and sets the flag, but those two actions are not performed atomically. Consequently, one process might execute between the other process's two actions. Some architectures provide an atomic "test and set" instruction (or other similar instructions, such as "fetch and add") that can be used to avoid the problem with this proposed solution.

## 5.3 An Alternating Solution

The following solution does provide mutual exclusion (CS1).

```
public class SoSoCSTurns {
  private static final int S = 4; // number of "sessions"
  private static int turn = 0; // or 1
  private static int x = 0;
  private static process p0 {
    for (int s = 1; s <= S; s++) {
```

```
      // non-critical section
      ...
      // entry protocol
      while (turn != 0) /* do nothing */ ;
      // critical section
      x += 3;
      // exit protocol
      turn = 1;
    }
  }
  private static process p1 {
    for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      while (turn != 1) /* do nothing */ ;
      // critical section
      x += 4;
      // exit protocol
      turn = 0;
    }
  }
  // main and done methods same as in GeneralForm
}
```

The value of `turn` is always either 0 or 1. If `p0` is in its critical section, then `turn` is 0, which prevents `p1` from entering its critical section. Unfortunately, this solution potentially violates property CS3. For example, suppose that `p0`'s non-critical section takes a long time to execute and `p1`'s non-critical section takes a very short time. When the program starts, `turn` will be 0 and `p0` will be in its non-critical section when `p1` reaches its entry protocol. Hence, `p1` will delay unnecessarily until after `p0` has completed its critical section and exit protocol. The general behavior is that the two processes execute their critical sections alternately.

## 5.4 The Bakery Algorithm for Two Processes

This section presents a correct solution to the critical section problem for two processes. We defer explaining the motivation behind the name of this algorithm until the next section, where it is clearer for the solution for `N` processes. The key to avoid the alternating nature of the previous solution is to use two different `turn` variables. Here is the code:

```
public class Bakery2 {
  private static final int S = 4; // number of "sessions"
  private static int turn0 = 0;
  private static int turn1 = 0;
  private static int x = 0;
```

```
  private static process p0 {
    for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      turn0 = 1;
      turn0 = turn1 + 1;
      while(turn1 != 0 && turn0 > turn1) /* do nothing */ ;
      // critical section
      x += 3;
      // exit protocol
      turn0 = 0;
    }
  }
  private static process p1 {
    for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      turn1 = 1;
      turn1 = turn0 + 1;
        // note: >= comparison below, but just > in p0
      while(turn0 != 0 && turn1 >= turn0) /* do nothing */ ;
      // critical section
      x += 4;
      // exit protocol
      turn1 = 0;
    }
  }
  // main and done methods same as in GeneralForm
}
```

We argue informally that this program solves the critical section problem by considering the key cases that arise in program execution. Although we cast `p0` and `p1` in particular roles below, they are generally symmetric.

- Suppose `p0` is attempting to enter its critical section while `p1` is in its non-critical section. Then, `p0` gains access because `turn1` is 0.
- Suppose `p0` is attempting to enter its critical section while `p1` is in its critical section. Then, `p0` will busy wait because `turn1` is not 0 and `turn0` was set to be greater than `turn1`. When `p1` finishes its critical section, it sets `turn1` to 0. There are two subcases. In each, `p0` will enter its critical section.
  - If `p1` is executing in its non-critical section when `p0` tests `turn1` (i.e., `turn1` is 0), then `p0` will enter its critical section.
  - On the other hand, `p1` might finish its non-critical section and set `turn1` to `turn0+1` as part of its entry protocol. When `p0` retests its condition,

it will find it false because `turn1` is greater than `turn0`. So, `p0` will enter its critical section.

- Suppose that `p0` and `p1` begin execution of their entry protocols at about the same time and each completes their first assignment statement; so, each turn variable is 1. Suppose further that, say, `p0` completes its second assignment statement and evaluates its condition before `p1` executes its second assignment statement; so, `turn0` is 2 and `turn1` is 1. Then, `p0` will wait until `p1` continues execution and sets `turn1` to 3. Note how one process setting its turn variable to 1 forces the other process to wait until the first process has finished picking its value for the turn variable.
- Suppose `p0` and `p1`, as in the previous case, complete their first assignment statements, which set each turn variable to 1. Suppose now that execution of the second assignment statements is interleaved at the machine instruction level, similar to what was discussed for the `Race` program in Section 4.1. That is, the following execution ordering can occur:

  1 `p0` reads `turn1`'s value (1)

  2 `p1` reads `turn0`'s value (1)

  3 `p0` adds 1 and stores the result (2) into `turn0`

  4 `p1` adds 1 and stores the result (2) into `turn1`

  Although the turn variables have equal values (2), `p1` will defer to `p0` because its condition uses >= whereas `p0`'s uses >.

## 5.5 The Bakery Algorithm for N Processes

The following solution generalizes the two-process solution in the previous section.

```
public class BakeryN {
  private static final int S = 4; // number of "sessions"
  private static final int N = 6; // number of processes
  private static int [] turn = new int [N]; // initialized to 0
  private static int x = 0;
  private static process p ( (int i = 0; i < N; i++) ) {
    for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      turn[i] = 1;
      turn[i] = maxturn()+1;
      for (int j = 0; j < N; j++) {
        if (j != i) {
          while(turn[j] != 0 &&
```

```
                greaterthan(turn[i],i,turn[j],j)) /* do nothing */ ;
          }
        }
        // critical section
        x += 3;
        // exit protocol
        turn[i] = 0;
      }
    }
    public static int maxturn() {
      int max = turn[0];
      for (int k = 1; k < N; k++) {
        max = Math.max(max, turn[k]);
      }
      return max;
    }
    // pairwise >.  i.e., (turni, i) > (turnj, j)
    public static boolean greaterthan(int turni, int i,
                                      int turnj, int j ) {
      return (turni > turnj) || (turni == turnj && i > j);
    }
    // main and done methods same as in GeneralForm
}
```

The overall structure is similar to the previous solution. Each process sets its turn variable to be one more than the maximum of the others. As in the previous solution, more than one process can pick the same value for its turn. Here again, the higher numbered process will defer to the lower numbered process. Specifically, the code uses a loop to check whether to defer to each other process. This checking uses a comparison method that performs a pairwise "greater than" operator between pairs of *(turn, process index).* It orders pairs by the value of turns, but within pairs with equal turns, it orders the pairs by process index.

The name of the bakery algorithm is motivated by how each entering customer (process) in a bakery takes a number just higher than the numbers already taken by waiting customers. As noted above, though, if two customers enter the bakery at about the same time, then they might both pick the same number, so one needs to defer to the other. Another algorithm, the *ticket algorithm,* is motivated by the ticket machines that are often used in bakeries, which dispense a sequence of increasing numbers to customers. However, it is more complicated to write or it requires the use of a special hardware instruction.

## Exercises

5.1 Give all possible outputs from the `GeneralForm` program assuming that the entry and exit protocols are empty and using the given values for `S` and `N`.

5.2 Consider a variant of the `GeneralForm` program (with empty entry and exit protocols) in which `N` is 3, `S` is 1, and the critical section code is

```
x += i+3; // recall: i is each process's ID
```

Give all possible outputs.

5.3 Give a solution to the `N` process critical section problem in which, as suggested in Section 5.1, process 0 executes its critical section `S` times in a row, then process 1 executes its critical section `S` times in a row, etc.

5.4 Run the code for the `BadCSFlag` program several times to see whether a race condition actually occurs. It may or may not depending on implementation factors. If it does not, then modify the code to, in effect, force a race condition to occur. Do so by placing one or more calls to `Thread.sleep` in the code. (Hint: first rewrite the code in the style of the `Race` program in Section 4.1.)

5.5 Gather evidence that the processes in the `SoSoCSTurns` program actually alternate execution of their critical sections. First, add print statements to the program and run the modified program. Then, place several calls to `Thread.sleep` in the code and verify that the output still shows the desired execution order.

5.6 Consider again the `SoSoCSTurns` program. What effect, if any, does initializing `turn` to 1 instead of 0 have?

5.7 Rewrite the `SoSoCSTurns` program using a family of two processes specified in a single quantified process.

5.8 Suppose each process in the `SoSoCSTurns` program is changed to output `x`'s value after its for loop. Give all possible output values from this modified program.

5.9 Gather evidence that the `Bakery2` program executes as it should. Insert print statements whenever a process enters or leaves its critical section, and place a call to `Thread.sleep` within each process's critical section. Then, examine the output of the program to verify that it does not show that two processes were in their critical sections at the same time.

5.10 Repeat the previous exercise for the `BakeryN` program.

5.11 Consider the `Bakery2` program. What effect, if any, does each of the following have?

(a) initializing `turn0` and `turn1` to 1 instead of 0.

(b) initializing `turn0` to 1 instead of 0 (but still initializing `turn1` to 0).

5.12 Consider the effect of deleting the assignments `turn0 = 1` and `turn1 = 1` from the `Bakery2` program. Does this modified program still provide mutual exclusion? If so, give a convincing argument; if not, give a specific execution ordering showing how both processes can be executing in their critical sections at the same time.

5.13 Consider the values taken on by the turn variables in the `Bakery2` program. Are these values bounded or can they become arbitrarily large? If they are bounded, give a specific bound and explain why that bound holds; otherwise, give a specific execution ordering showing how they can grow arbitrarily large. Give answers for when the program executes `S` sessions and for when the program executes infinitely.

5.14 Consider the code for `BakeryN` when `N` is 2. Show that it really is equivalent to the code for `Bakery2`.

5.15 Consider the code for `BakeryN`. What effect, if any, does eliminating the if statement (but keeping its while loop) have? Explain your answer.

Chapter 6

# SEMAPHORES

Semaphores are a low-level but efficient synchronization mechanism. They are used in JR programs to synchronize the activities of processes. For example, they can be used to implement mutually exclusive access to shared data. Compared with synchronization using shared variables seen in the previous chapter, semaphores are generally easier to use and more efficient because their implementation does not require busy waiting.

A semaphore is a non-negative integer that is accessed by means of two special operations, `P` and `V`.[1] If `s` is a semaphore, `V(s)` increments the value of `s`, and `P(s)` delays its caller until `s` is positive and then decrements `s`. A `V` is used to signal the occurrence of an event, and a `P` is used to delay until an event has occurred.

This chapter describes how semaphores are declared and shows how they can be used. As mentioned in the introduction to Part I, JR's semaphores are actually an abbreviation of a particular form of message passing. Section 7.5 discusses this concept in detail. JR supports both simple semaphores and arrays of semaphores.

## 6.1 Semaphore Declarations and Operations

A semaphore declaration contains a list of one or more semaphore definitions separated by commas:

**sem** *sem_definition*, *sem_definition*, ...

[1] Semaphores were invented by Dijkstra [18], who is Dutch. The operations `P` and `V` are mnemonics for the Dutch words *passeren* and *vrygeven,* which mean "to pass" and "to release," respectively.

Each semaphore definition specifies a single semaphore and optionally gives the initial value of the semaphore. A semaphore definition has the following general form:

*sem_id* or *sem_id = expr*

As shown, the initialization clause is optional. Its value must be non-negative since semaphores are non-negative. The default initial value of each semaphore is zero.

JR uses traditional notation for the two standard semaphore operations, `P` and `V`. They have the general forms

`P(` *sem_reference* `)`
`V(` *sem_reference* `)`

For simple semaphores, a *sem_reference* is just the name of the semaphore, i.e., a *sem_id.*

To illustrate one use of semaphores, consider the following instance of the standard critical section problem seen in Chapter 5. Suppose `N` processes share a class variable, e.g., a counter. Access to the counter is to be restricted to one process at a time to ensure that it is updated atomically. An outline of a JR solution follows:

```
public class CS {
  private static final int N = 20; // number of processes
  private static int x = 0;        // shared variable
  private static sem mutex = 1;    // mutual exlusion for x

  private static process p( (int i = 0; i < N; i++) ) {
    // non-critical section
      ...
    // critical section
    P(mutex); // enter critical section
    x = x + 1;
    V(mutex); // leave critical section
    // non-critical section
      ...
    }
  public static void main(String [] args) {
  }
}
```

The mutex semaphore is initialized to 1 so that only a single process at a time can modify `x`.

Processes wait on semaphores in first-come, first-served order based on the order in which they execute `P` operations. Thus waiting processes are treated fairly: A process waiting on a semaphore will eventually be able to proceed after it executes a `P`, assuming a sufficient number of `V`s are executed on that semaphore.

As mentioned, JR supports arrays of semaphores. Because a semaphore in JR is an object, the declaration of an array of semaphores follows the style of declarations of arrays of other objects in Java. Here, a reference to a semaphore is an operation capability and so the *sem_reference* that appears within `P` or `V` is also an operation capability. To obtain an array of semaphores, an array of capabilities must be declared and each element of the array must be explicitly instantiated. For example, an array of five semaphores, `t`, can be declared and instantiated as follows:

```
cap void () t[] = new cap void()[5];
for (int i = 0; i < 5 ; i++ ) {
  t[i] = new sem;
}
```

This code might appear within a method or, in general, within a block of code. Other examples below show how to declare semaphores at the class level. The semaphore operations can be applied to individual elements of the array, e.g., a `V` on the second element of `t` is written `V(t[1])`. In the above, each element of t is initialized to zero, the default initial value for semaphores. An element can be initialized to other, non-negative values by passing the value as the parameter to the `sem` constructor, e.g.,

```
cap void () t[] = new cap void()[5];
int [] tinit = {2,7,1,8,3};
for (int i = 0; i < 5 ; i++ ) {
  t[i] = new sem(tinit[i]);
}
```

Arrays of semaphores are often used in managing collections of computing resources (e.g., printers or memory blocks) or in controlling families of processes. Typically one semaphore is associated with each computing resource or each process. For example, suppose that `N` processes are to enter their critical sections in circular order according to their process identities, i.e., first process 0, then process 1, and so on up to process `N-1`, with the cycle repeating four times. This can be expressed as follows:

```
public class CSOrdered {
  private static final int N = 20; // number of processes
  private static final int C =  4; // number of cycles
  private static cap void () mutex[] = new cap void()[N];
  static { // use static initializer to initialize mutex array
    mutex[0] = new sem(1);
    for (int i = 1; i < N ; i++ ) {
      mutex[i] = new sem;
    }
  }
  private static process p( (int i = 0; i < N; i++) ) {
```

```
    for (int cycle = 1; cycle <= C; cycle++) {
      // non-critical section
        ...
      // critical section
      P(mutex[i]);       // enter critical section
                         // actual critical section
      V(mutex[(i+1)%N]); // leave critical section
      // non-critical section
      ...
    }
  }
  public static void main(String [] args) {
  }
}
```

The array of semaphores, `mutex`, has one element for each process `p`. It acts as a *split binary semaphore* [7]: At most one of the semaphores in the array is 1, the rest are 0. That corresponds to the desired property that only one process at a time can be in its critical section. The element of `mutex` that is 1 indicates which process has permission to enter its critical section. As process `i` leaves its critical section, it passes permission to enter the critical section to the next process by signaling `mutex[(i+1) % N]`.

Because semaphores are objects, they can be created anywhere in a program, not just as part of initialization. Section 9.9 discusses such creation in a more general setting.

## 6.2 The Dining Philosophers Problem

This section presents a semaphore solution to the classic Dining Philosophers Problem [17]. In this problem, `N` philosophers (typically five) sit around a circular table set with `N` chopsticks, one between each pair of philosophers. Each philosopher alternately thinks and eats from a bowl of rice. To eat, a philosopher must acquire the chopsticks to its immediate left and right. After eating, a philosopher places the chopsticks back on the table.

The usual statement of this problem is that philosophers use two forks to eat spaghetti rather than two chopsticks to eat rice. Apparently, the spaghetti is so tangled that a philosopher needs two forks! Although we think the chopstick analogy is more fitting, our explanations and code use the more traditional forks.

This initial table setting for five philosophers is illustrated in Figure 6.1. In the figure a P represents a philosopher and an F represents a fork.

In our solution to this problem, we represent philosophers by processes. Because philosophers (processes) compete for forks, their use needs to be synchronized. So, we represent each fork as a semaphore. Here is the code:

```
public class DiningPhilosophers {
  private static final int N =  5; // number of philosopers
```

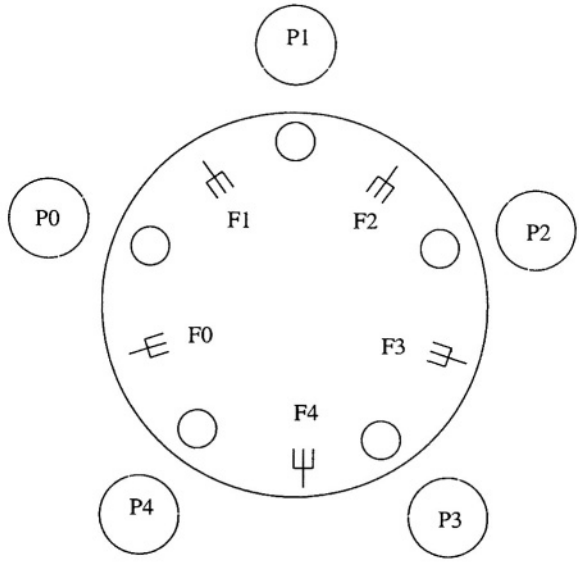


*Figure 6.1.* Initial table setting for Dining Philosophers

```
  private static final int T = 20; // number of sessions
  private static cap void () fork[] = new cap void()[N];
  static {
    for (int i = 0; i < N ; i++ ) {
      fork[i] = new sem(1);
    }
  }
  public static void main(String [] args) {
  }
  private static process phil ( (int id = 0; id < N; id++) ) {
    // determine this philosopher's forks
    int right = id, left = (id+1>N-1?0:id+1);
    // introduce asymmetry to avoid deadlock
    if (id == 0) {
      int temp = left; left = right; right = temp;
    }
    for (int i = 1; i <= T; i++) {
      P(fork[left]);  P(fork[right]);
      System.out.println("Philosopher "+id+" is eating");
      V(fork[left]);  V(fork[right]);
      System.out.println("Philosopher "+id+" is thinking");
    }
  }
}
```

It declares the forks as an array of semaphores and initializes each fork to indicate that it is available, i.e., on the table. Each philosopher determines the

indices of its left and right forks. It then executes its loop for `T` sessions of eating and thinking. Before it eats, it acquires each fork by using a `P`; it will wait if its neighboring philosopher is eating. When it finishes eating, it puts down each fork by using a `V`.

Our solution could deadlock if not for the if statement in the code for the philosopher. Deadlock (see Section 5.1) in this problem means that all philosophers could be attempting to eat, but none would be allowed to. That can happen (in code without the if statement) if each philosopher grabs its left fork. Then each would try to grab its right fork, which is already held by another philosopher! So, unfortunately, the philosophers could make no further progress. Using that if statement avoids this problem by making one philosopher act slightly differently. In particular, the effect of the if statement is that philosopher 0 acquires its forks in the opposite order. Hence, bad scenarios such as the one described above cannot occur.

Chapter 11 contains solutions to the dining philosophers problem in a distributed environment. The solutions presented there use the other JR synchronization mechanisms developed in the remaining chapters in this part.

## 6.3 Barrier Synchronization

A barrier is a common synchronization tool used in parallel algorithms. It is used in iterative algorithms—such as some techniques for finding solutions to partial differential equations—that require all tasks to complete one iteration before they begin the next iteration. (A barrier might also be used to synchronize stages within an iteration.) This is called barrier synchronization since the end of each iteration represents a barrier at which all processes have to arrive before any are allowed to pass. (See Reference [7] for further discussion and specific applications.)

One possible structure for a parallel iterative algorithm is to employ several worker processes and one coordinator process. The workers solve parts of a problem in parallel. They interact with the coordinator to ensure the necessary barrier synchronization. This kind of algorithm can be programmed as follows:

```
public class Barrier {
  private static final int N = 20; // number of processes
  private static sem done = 0;
  private static cap void () proceed[] = new cap void()[N];
  static {
    for (int i = 0; i < N ; i++ ) {
      proceed[i] = new sem;
    }
  }

  private static process worker( (int i = 0; i < N; i++) ) {
    while (true) {
```

```
      // code to implement one iteration of task i
      ...
      // barrier
      V(done);       // tell coordinator "I did iteration i"
      P(proceed[i]); // wait for coordinator to say "continue"
    }
  }
  private static process coordinator {
    while (true) {
      for (int w = 0; w < N; w++) { P(done); }
      for (int w = 0; w < N; w++) { V(proceed[w]); }
    }
  }
  public static void main(String [] args) {
  }
}
```

Each worker first performs some action; typically the action involves accessing part of an array determined by the process's subscript `i`. Then each worker executes a `V` and a `P`, in that order. The `V` signals the coordinator that the worker has finished its iteration; the `P` delays the worker until the coordinator informs it that all other workers have completed their iterations. The coordinator consists of two for loops. The first loop waits for each worker to signal that it is done. The second loop signals each worker that it can continue.

The above implementation of barrier synchronization employs an extra coordinator process and has execution time that is linear in the number of workers. It is more efficient to use a symmetric barrier with logarithmic execution time (see Reference [7] and Exercise 6.15).

So far, the examples in this chapter have declared semaphores to be private and static. They can also be declared to be public. In that role, semaphores provide synchronization for processes executing inside or outside the class. A semaphore can also be declared as non-static, which, as with other Java objects, makes the semaphore specific to each instance of the class, rather than to the entire class.

As an example, we can rewrite the code in the previous barrier example as follows to put the semaphores and coordinator process in one class, `Barrier`, and the workers in another, `Workers`. The `Main` class creates an instance of `Barrier` and an instance of `Workers`, to which it passes a reference for the former.

```
public class Main {
  private static final int N = 20; // number of processes

  public static void main(String [] args) {
    // create a barrier
    Barrier bar = new Barrier(N);
```

```
    // create the workers and pass them their barrier
    Workers ws = new Workers(N, bar);
  }
}
```

The `Barrier` class declares the semaphores to be public and contains the coordinator process.

```
public class Barrier {
  private int N; // number of processes
  public sem done = 0;
  public cap void () proceed[];

  Barrier (int N) {
    this.N = N;
    proceed = new cap void()[N];
    for (int i = 0; i < N ; i++ ) {
      proceed[i] = new sem;
    }
  }

  private process coordinator {
    while (true) {
      for (int w = 0; w < N; w++) { P(done); }
      for (int w = 0; w < N; w++) { V(proceed[w]); }
    }
  }
}
```

The `Workers` class contains the worker processes, which use the barrier that the class is passed.

```
public class Workers {
  private int N;
  private Barrier mybar;

  Workers(int N, Barrier mybar) {
    this.N = N;
    this.mybar = mybar;
  }

  private process worker( (int i = 0; i < N; i++) ) {
    while (true) {
      // code to implement one iteration of task i
      ...
      // barrier
      // tell coordinator "I did iteration i"
      V(mybar.done);
      // wait for coordinator to say "continue"
      P(mybar.proceed[i]);
```

```
    }
  }
}
```

The advantage of this structure is that it separates the details of the coordinator process from the details of the workers. In fact, the workers could themselves be in different classes. By making the semaphores non-static, this structure also makes it easy to create multiple instances of barriers within the same program. Other structures for barriers are also possible; for example, see Section 16.2 and Exercise 6.14.

## Exercises

6.1 Write a program that contains two processes, `p1` and `p2`. `p1` outputs two lines: "hello" and "goodbye". `p2` outputs one line: "howdy". Use one or more semaphores to ensure that `p2`'s line is output between `p1`'s two lines.

6.2 Consider the code in the `CS` program in Section 6.1. What are the minimum and maximum values that the `mutex` semaphore can take on during execution of the program for any possible execution ordering of processes? What is the maximum number of processes that might be waiting to enter their critical sections? What is the minimum number of times that processes might need to wait to enter their critical sections? Explain your answers.

6.3 Consider the code in the `CSOrdered` program in Section 6.1. Suppose the semaphore initialization code in the static initializer is moved to the main method. The modified program is not guaranteed to be equivalent to the original program because the processes might execute before the semaphores are initialized. Show how to further modify this program so that it is guaranteed to be equivalent. (Hint: add a single "go ahead" semaphore on which processes wait until the main method indicates that it has completed initialization.)

6.4 Complete the code in the `CSOrdered` program in Section 6.1 so that each process outputs its process id (i.e., `i`) within its critical section. Also, modify the order in which processes execute their critical sections to be one of the following:

(a) on each cycle: 0, 2, 4, ..., *X*, 1, 3, 5, ..., *Y* where *X* is the largest even number < `N` and *Y* is the largest odd number < `N`.

(b) for the overall program execution: 0, 0, 1, 1, 2, 2, ..., `N-1`, `N-1`, 0, 0, 1, 1, 2, 2, ..., `N-1`, `N-1`.

6.5 Consider Exercise 5.2 but with the critical section code (the assignment to `x`) enclosed within `P(t)` and `V(t)`. For each initial value 0, 1, 2, 3, and 4 for the semaphore `t`, give all possible outputs from the program.

6.6 This exercise builds on Exercise 4.4.

(a) Modify the solution to Exercise 4.4 so that

- Both processes modify `anybig` directly and on each iteration. (So, eliminate variables `evenbig` and `oddbig`.) Use a semaphore to provide the needed synchronization. Explain why synchronization is needed here.
- The program outputs only `anybig` at the end.

(b) Modify the program from part (a) by removing all synchronization. Run the modified program several times. If the output ever differs from the output of the program in part (a), explain why. If it does not, explain why not. (Note: whether or not any difference appears depends on implementation factors. See also the next part.)

(c) Modify the program from part (b) so that it outputs an incorrect result due to a race condition on a particular set of input data. (Include as a comment in the code the input data.) Hint: Use `Thread.sleep` to force context switches at strategic points of execution; have one process sleep and the other not.

6.7 Repeat the previous exercise, but for Exercise 4.5 (and for variables `small1`, `small2`, and `smallest`).

6.8 Consider the `DiningPhilosophers` program. First, remove the if statement and run the program several times. Does the program deadlock? (Whether or not it does depends on implementation factors.) Then, modify the program to force it to deadlock. Do so by adding `Thread.sleep` to force context switches at strategic points of execution.

6.9 Another classic concurrent programming problem is the producers/consumers problem. In this problem, two kinds of processes communicate via a single buffer. Producers deposit messages into the buffer and consumers fetch messages from the buffer. Here is an outline of the code

```
public class PC {
  private static final int N =  5; // number of Ps and Cs
  private static final int T = 20; // number of sessions
  private static int buffer; // the shared buffer
  public static void main(String [] args) {
  }
```

```
  private static process prod ( (int id = 0; id < N; id++) ) {
    for (int i = 1; i <= T; i++) {
      int m;
      // produce a message, m
      ...
      // deposit the message
     buffer = m;
      ...
    }
  }
  private static process cons ( (int id = 0; id < N; id++) ) {
    for (int i = 1; i <= T; i++) {
      int m;
      // fetch a message
      m = buffer;
      // consume the message m
      ...
    }
  }
}
```

Obviously, this outline is missing synchronization to ensure that a consumer does not take a message from an empty buffer or take a message that another consumer has already taken, and that a producer does not overwrite a message before a consumer consumes it. (Each message is to be read by one consumer, not by all consumers.)

Complete the above outline with the necessary synchronization. Hint: use two semaphores. (Section 9.2 reexamines this problem using the rendezvous synchronization mechanism. The bounded buffer problem, which is a generalization of this problem, is presented in Section 9.3 and used in examples in Chapter 21.)

6.10 Consider the code in the `Barrier` class (see the start of Section 6.3). Can the array of semaphores, `proceed`, be replaced by a single semaphore? Explain.

6.11 Consider the code in the `Barrier` class (see the start of Section 6.3). Can the done semaphore be replaced by an array of `N` semaphores, with one element for each worker? Explain.

6.12 Eliminate the coordinator process from the code in the `Barrier` class (see the start of Section 6.3) by having the last worker that arrives at the barrier signal the other workers. Hint: Use a counter protected by a critical section.

6.13 In the `Barrier` class (see the end of Section 6.3), why does the body contain a process? What would happen if the code for the coordinator process were moved to the end of `Barrier`'s constructor?

6.14 Rewrite the code in the `Barrier` and `Workers` classes (see the end of Section 6.3) to hide all details about the implementation of the barrier in the `Barrier` class. The `Barrier` class should make public a single method that the workers call when they arrive at the barrier. The `Barrier` class should declare the semaphores as private so they cannot be used directly by the workers.

6.15 A *dissemination barrier* [21] is much more efficient than one implemented using a coordinator process. It consists of a series of stages in which each worker interacts with two others. Workers first interact with others that are distance 1 away, then distance 2 away, then distance 4 away, and so on. If there are $n$ workers, the number of stages is the (ceiling of the) logarithm of $n$.

For example, suppose there are eight workers. In the first stage, worker 1 signals worker 2 then waits for worker 8, worker 2 signals worker 3 then waits for worker 1, and so on. In the second stage, worker 1 signals worker 3 then waits for worker 7, worker 2 signals worker 4 then waits for worker 8, and so on. In the third stage, worker 1 signals worker 5 and waits for worker 5, worker 2 signals worker 6 and waits for worker 6, and so on. At the end of the third stage, the workers can proceed past the barrier since each will know that all others have arrived.

Implement a dissemination barrier for 20 processes; use semaphores for synchronization. Compare its performance to either of the coordinator barriers in Section 6.3.

Chapter 7

# ASYNCHRONOUS MESSAGE PASSING

Asynchronous message passing is higher-level and more powerful than semaphores. As its name implies, it allows processes to communicate as well as synchronize by using operations to exchange messages.

Message passing in JR is accomplished by having processes send messages to and receive messages from operations. In this role, operations serve as queues that hold messages for receivers. The sender of a message continues immediately after the message has been sent. The receiver of a message delays until there is a message on the queue and then removes one. Thus the send statement is asynchronous (non-blocking) and the receive statement is synchronous (blocking).

This chapter first describes this new use of operations as message queues. We then show how the semaphore primitives described in the previous chapter are actually abbreviations for a specific form of asynchronous message passing. We also describe the use of data-containing semaphores, which are a generalization of standard semaphores. Finally, we describe how multiple processes can receive messages from the same operation and discuss the additional flexibility that provides.

JR's receive statement is actually just an abbreviation for a more general mechanism for receiving messages. That more general mechanism—the input statement—is discussed in Chapter 9.

## 7.1 Operations as Message Queues

As we saw in Chapters 3 and 6, operations can be serviced by methods. In this case an operation definition specifies the method's parameterization. Each invocation of the operation results in a new instance of the method's code being executed to service the invocation. Specifically, a call invocation of a method

is like a procedure call (Chapter 3); a send invocation of a method results in a new process being created (Chapter 6).

An alternative role of an operation is to define a message queue. In this role the operation has no corresponding method. Instead, invocations of the operation (i.e., messages) are serviced by receive statements within one or more processes. A receive statement removes an invocation from the message queue; the executing process delays if no invocation is present.

A send invocation of an operation serviced by receive statements causes the invocation to be appended to the message queue. The invoker continues immediately after the invocation has been sent. Figure 2.1 summarizes these actions. Note that this is consistent with send invocations to methods being asynchronous. Call invocations to operations serviced by receive statements are also allowed; this provides a synchronous form of message passing, which we discuss in Chapter 9.

A receive statement specifies an operation and gives a list of zero or more variables separated by commas. It has the following general form:

`receive` *op_expr* `(` *variable*`,` *variable*`, ...` `)`

The *op_expr* is any expression that evaluates to an operation: it can specify an operation directly by the operation's name or indirectly via an operation capability. The former is most common, but the latter is also very useful as will be seen in Section 7.2 and later chapters. A receive statement specifies one variable for each parameter in the operation's definition; it must match the corresponding parameter's type.

As mentioned earlier, execution of receive removes an invocation from the message queue associated with the operation. The values of the arguments of that invocation are assigned to the variables in the receive statement. (These variables must already have been declared in the current scope.)

As an example, consider a three-process system. Each of two processes sends an ordered stream of messages to the third, which outputs the merge of the two streams. For simplicity, we assume that the messages contain just integers and that the end of each stream is indicated by a value greater than any possible value in the stream. Following is an outline of a solution:

```
public class StreamMerge {
  // end of stream marker:
  private static final int EOS = Integer.MAX_VALUE;
  private static op void stream1(int);
  private static op void stream2(int);

  private static process one {
    ...
    send stream1(y);
    ...
    send stream1(EOS);
```

```
  }
  private static process two {
    ...
    send stream2(y);
    ...
    send stream2(EOS);
  }
  private static process merge {
    int v1, v2;
    receive stream1(v1);  receive stream2(v2);
    while (v1 < EOS || v2 < EOS) {
      if (v1 <= v2) {
        System.out.println(v1);
        receive stream1(v1);
      }
      else { // v1 > v2
        System.out.println(v2);
        receive stream2(v2);
      }
    }
    System.out.println(EOS);
  }

  public static void main(String [] args) {
  }
}
```

This program uses two operations, `stream1` and `stream2`. The first process sends its numbers, including the end of stream marker `EOS`, to `stream1`, the second sends to `stream2`. The `merge` process first gets a number from each stream. It executes the body of the loop as long as one of the two numbers is not `EOS`. The if statement compares the two numbers, outputs the smaller, and receives the next number in the stream from which the smaller number came. If one stream "dries up" before the other, `v1` or `v2` will be `EOS`. Since `EOS` is larger than any other number in the stream, numbers from the non-dry stream will be consumed until it too is dry. The loop terminates when both streams have been entirely consumed.

As indicated for the above program, messages sent from one process to another are delivered in the order in which they are sent. However, in other cases, non-deterministic process execution can affect message ordering. For example, consider the following program:

```
public class Order {
  private static op void a(int);
  private static op void b(int);
  private static process one {
    send a(1);
    send b(2);
```

```
  }
  private static process two {
    int x;
    receive a(x);
    send b(3);
  }
  private static process three {
    int x;
    receive b(x);
    receive b(x);
  }
  public static void main(String [] args) {
  }
}
```

The order in which process `three` receives the two `b` messages depends on the order in which the other two processes execute.

## 7.2 Invoking and Servicing via Capabilities

Section 3.3 showed how methods can be invoked indirectly via capabilities. An operation serviced by a receive statement can too. As a simple example, consider the following program.

```
public class Cap1 {
  private static op void f(double);
  private static op void g(double);
  public static void main(String [] args) {
    cap void (double) y, z;
    // make y point to one of f or g, and z point to other
    if (args.length > 0) { y = f; z = g; }
    else                 { y = g; z = f; }
    // invoke what y points to and then what z points to
    send y(4.351);
    send z(8.21);
  }
  private static process pf {
    double d;
    receive f(d);
    System.out.println("pf got "+d);
  }
  private static process pg {
    double d;
    receive g(d);
    System.out.println("pg got "+d);
  }
}
```

The main method declares two capabilities and assigns them the values of `f` and `g`, although which capability gets which value depends on whether any

command-line arguments are present. It then sends to the operations associated with the capabilities. A more realistic example of the use of capabilities appears in Section 7.3.

As seen in Section 7.1, a receive statement can also name a capability as a way of indirectly specifying the operation from which to receive. The capability is evaluated to determine the operation from which to receive. As a simple example, consider the following program:

```
public class Cap2 {
  private static op void f(double);
  private static op void g(double);
  public static void main(String [] args) {
    cap void (double) y, z;
    // make y point to one of f or g, and z point to other
    if (args.length > 0) { y = f; z = g; }
    else                 { y = g; z = f; }
    send f(4.351);
    send g(8.21);
    double d;
    receive y(d);
    System.out.println("rcv1 got "+d);
    receive z(d);
    System.out.println("rcv2 got "+d);
  }
}
```

As in the previous program, the main method declares and assigns values to two capabilities, `y` and `z`. It then sends to the two operations and receives from the operation associated with `y` and then from the operation associated with `z`. More realistic examples of this use of capabilities appear in later chapters (e.g., Section 20.2.4).

Capabilities can be used to circumvent normal scoping rules. Consider, for example, the following program

```
public class Cap3 {
  public static op void getcap(cap void (double));
  public static process p {
    op void f(double);
    send getcap(f);
    double d;
    receive f(d);
    System.out.println("rcv got "+d);
  }
  public static process q {
    cap void (double) y;
    receive getcap(y);
    send y(5.78);
  }
```

```
  public static void main(String [] args) {
  }
}
```

Operation `f` is local to process `p`. However, a capability for it is passed outside of `p` and used by process `q`. The capability is passed via the operation `getcap`, which is known to both `p` and `q`. Section 7.3 contains a similarly structured example. A similar technique can be used to make an operation that is private in one class available in another class (see Exercise 7.5).

The above examples illustrate capabilities that are assigned operations serviced by receive statements. As seen in Section 3.3, capabilities can also be assigned operations serviced by op-methods. In fact, capabilities can be assigned to either kind of operation. Thus, how the operation being invoked via the capability is actually serviced is transparent to the invoker. This flexibility is often useful, as will be seen in later chapters. (See Exercise 7.6 for a simple example.)

When an operation goes out of scope, it continues to exist if there are any capabilities for it. That is, an operation is an object and capabilities act as references to it. To demonstrate this effect, suppose that in the above program process `q` executes a second send to operation `f` (indirectly via `y`). Process `p` may or may not have terminated when this second invocation occurs. In either case, though, the invocation is legal, but it has no effect in this program. (In general, evaluation of its parameters might have side effects, another process might service the operation, etc.) The invocation simply will not be serviced. Similarly, any pending invocations of an operation when the operation ceases to exist (i.e., due to there being no more references for it) will just be ignored.

The capability specified in a send or receive statement can take on the capability values `null` and `noop`. Their meanings in these contexts are consistent with their meanings in invoking a method as described in Section 3.3. Sending to or receiving from `null` causes a run-time exception. Sending to `noop` has no effect (except for any side effects of argument evaluation). Receiving from `noop` causes the program to hang (because there will never be an invocation associated with `noop`).

## 7.3 Simple Client-Server Models

As further examples of asynchronous message passing, we now consider how to program several simple client-server models. A server is a process that repeatedly handles requests from client processes. For example, a disk server might read information from a disk; its clients might pass it requests for disk blocks and then wait for results.

We first consider the case of one client process and one server process. An outline of possible interactions between client and server is shown in program `Model1` below. The processes share two operations: `request` and `results`.

The client sends to the `request` operation and waits for results to be returned by receiving from the `results` operation; between the send and receive, the client can perform other work. The server waits for a request, performs the requested action, and sends the results back. To make the examples in this section more specific, our code shows the type of the request data as a character and the type of the results data as a double.

```
public class Model1 {
  private static op void request(char);
  private static op void results(double);

  private static process client {
    ...
    send request('w');
    // possibly perform some other work
    double d;
    receive results(d);
    ...
  }
  private static process server {
    while (true) {
      char data;  double ans;
      receive request(data);
      // handle request; put answer in ans
      ...
      send results(ans);
    }
  }
  public static void main(String [] args) {
  }
}
```

Unfortunately, the above code does not generalize directly if more than one client process is present. Specifically, one client can obtain the results intended for the other because the `results` operation would be shared by both of them. The processes can execute so that results intended for one process are sent to `results` first, but another process is first to execute its receive statement. One way to generalize the code is to use an array of result operations, one for each client process. An outline of that kind of solution follows:

```
public class Model2 {
  private static final int N = 20; // number of client processes
  private static op void request(int, char);
  private static cap void (double) results[] =
    new cap void (double)[N];
  static {
    for (int i = 0; i < N ; i++ ) {
      results[i] = new op void (double);
```

```
    }
  }

  private static process client( (int i = 0; i < N; i++) ) {
    ...
    send request(i, 'w');
    // possibly perform some other work
    double d;
    receive results[i](d);
    ...
  }
  private static process server {
    while (true) {
      int id;  char data;  double ans;
      receive request(id, data);
      // handle request; put answer in ans
      ...
      send results[id](ans);
    }
  }
  public static void main(String [] args) {
  }
}
```

Here each client process passes its identity as part of its request message. The identity is used by the server to send the results of a request back to the client that initiated that request. Each client process receives from the one element of the `results` operation that corresponds to its identity.

An obvious drawback of the code in resource `Model2` is that it requires the number of clients to be known in advance. That requirement is not reasonable in many situations, such as when the server process is in a library. The `results` array in `Model2` provides a simple means to associate an operation with each client process. Another way to achieve the same effect is to declare an operation local to each client process:

```
public class Model3 {
  private static final int N = 20; // number of client processes
  private static op void request(cap void (double), char);

  private static process client( (int i = 0; i < N; i++) ) {
    op void results(double);
    ...
    send request(results, 'w');
    // possibly perform some other work
    double d;
    receive results(d);
    ...
  }
  private static process server {
```

```
    while (true) {
      cap void (double) results_cap;  char data;  double ans;
      receive request(results_cap, data);
      // handle request; put answer in ans
      ...
      send results_cap(ans);
    }
  }
  public static void main(String [] args) {
  }
}
```

Each client declares a local operation, `results`, whose parameterization is that of the result messages. It passes a capability for that operation as the first parameter to `request`. The server receives that capability in local variable `results_cap` and uses it to send back results to the operation to which the capability points.

An important advantage of the above structure is that it permits any client process to interact with the server. All the process needs is a results operation to pass to the server. Clients can even be in different resources than the server—even different virtual machines—as long as the `request` operation is made visible by declaring it in the spec of the server resource.

Another variant of the client-server model is to have multiple servers. Consider the case where a new server is created for each client's request. The following outlines a solution:

```
public class Model4  {
  private static final int N = 20; // number of client processes
  private static op void request(cap void (double), char);

  private static process client( (int i = 0; i < N; i++) ) {
    op void results(double);
    ...
    send request(results, 'w');
    // possibly perform some other work
    double d;
    receive results(d);
    ...
  }
  private static void request (cap void (double) results_cap,
                               char data) {
    double ans;
    // handle request; put answer in ans
    ...
    send results_cap(ans);
  }
  public static void main(String [] args) {
  }
```

```
}
```

The key difference between this code and that in resource `Model3` is that the `request` operation is now serviced by a method. Thus a new server process is created for each invocation of `request`. The parameterization of `request` is unchanged. Each server process uses the capability technique from `Model3` to send results back to its client.

It is worth emphasizing that the only difference between programs `Model3` and `Model4` is the way the `request` operation is serviced. The client processes execute exactly the same code. This is significant since clients can invoke `request` without concerning themselves with how it is serviced—whether by a method or by a receive statement. Section 8.2 shows a simpler way to package servers like the ones in `Model4`.

Section 9.9 presents another client-server model. It uses dynamically created operations, which can also be used with send and receive statements.

## 7.4 Resource Allocation

As a special, but important, kind of client-server model, consider the problem of resource allocation. Here, the server controls one or more units of some resource, which it gives out to the clients in response to their requests. As a concrete example, the server might manage a group of indistinguishable printers.

To start, consider the case where the server manages only a single unit of the resource. A client requests one unit of the resource, uses it, and then informs the server that it has finished using the resource.

```
public class ResourceAllocatorSingle {
  private static final int N = 20; // number of client processes
  private static op void request(int);
  // parameter in a request is client id.
  private static op void release();
  private static cap void () creply[] =
    new cap void ()[N];
  static {
    for (int i = 0; i < N ; i++ ) {
      creply[i] = new op void ();
    }
  }

  private static process client( (int i = 0; i < N; i++) ) {
    ...
    send request(i);
    // possibly perform some other work
    receive creply[i]();
    // use resource, then release it
    send release();
```

```
    ...
  }
  private static process server {
    while (true) {
      int id;
      receive request(id);
      send creply[id]();
      receive release();
    }
  }
  public static void main(String [] args) {
  }
}
```

The code is similar to that for client-servers in the previous section, but now clients and servers interact via two kinds of operations. Client request the resource via the `request` operation and release the resource via the `release` operation. Note that clients wait for a reply only after a request. The server code is simple: it uses "flow of control" to alternately receive a request message and then a release message. Section 9.2 will show this same example, but packaged in a simpler way.

Now, consider the case where there are multiple units of the resource. The code for the single unit case will clearly not work here and requires substantial modification. The fundamental reason is that now the server, in some cases, needs to be able to service *either* a request message or a release message. That is, the server will not be able to service immediately all request messages. It must somehow defer handling those until it has received release messages. The following code solves this problem.

```
public class ResourceAllocatorMultiple {
  private static final int N = 20; // number of client processes
  private static op void action(int, int, int);
  // 1st parameter in a message is client id.
  // values for message kind (2nd parameter) in a message:
  private static final int REQUEST = 1;
  private static final int RELEASE = 2;
  // value for unused unitId (3rd parameter) in a message:
  private static final int UNUSED = -999;
  private static cap void (int) creply[] =
    new cap void (int)[N];
  static {
    for (int i = 0; i < N ; i++ ) {
      creply[i] = new op void (int);
    }
  }

  private static process client( (int i = 0; i < N; i++) ) {
    ...
```

```
    send action(i, REQUEST, UNUSED);
    // possibly perform some other work
    int unitId;
    receive creply[i](unitId);
    // use resource unitId, then release it
    send action(i, RELEASE, unitId);
    ...
  }
  private static process server {
  final int MaxUnits = 8; // maximum units available
    int avail = MaxUnits;
    // save identities of all units
    List units = new ArrayList();
    for (int u = 0; u < MaxUnits; u++) {
      units.add(new Integer(u));
    }
    // process ids of pending requests
    List pending = new ArrayList();
    while (true) {
      int id;  int k;  int unitId;
      receive action(id, k, unitId);
      if (k == REQUEST) {
        if (avail > 0) {
          avail--;
          unitId = ((Integer) units.remove(0)).intValue();
          send creply[id](unitId);
        }
        else { // avail == 0; defer request
          pending.add(new Integer(id));
        }
      }
      else { // i.e., k == RELEASE
        if (pending.isEmpty()) {
          // no pending requests, save unit for later use
          avail++;
          units.add(new Integer(unitId));
        }
        else {
          // pass this just released unitId to first pending request
          id = ((Integer) pending.remove(0)).intValue();
          send creply[id](unitId);
        }
      }
    }
  }
  public static void main(String [] args) {
  }
}
```

The interface between clients and server has changed. Now, a single operation, `action`, is used for both request and release messages. The kind of message is encoded as a parameter to `action`. The server code is considerably more involved than it was for the previous example. It receives a message. For a request message, if the server has any resources available, then it can handle that request immediately, so it replies to the client; otherwise, it defers replying to the request until after a resource is returned. It handles deferring by saving the process's id in a queue of pending requests. For a release message, if no request is presently pending, then the server just returns the unit of resource to its available pool; otherwise, it passes this unit to the first client that is waiting on the queue.

This example shows a need for a multi-way receive. Although it can be accomplished as shown above, Chapter 9 discusses the input statement, which provides direct support for multi-way receive. Section 9.3 will show this same example, but solved more directly using the input statement.

## 7.5 Semaphores Revisited

JR's semaphores and `P` and `V` statements, as described in Chapter 6, are actually just abbreviations for operations and send and receive statements. Specifically, a semaphore is a parameterless operation. A `V` on a semaphore corresponds to a send to the operation; a `P` corresponds to a receive on the operation. Initialization of the semaphore to *expr* corresponds to a for loop that sends to the operation *expr* times.

*Table 7.1.* Correspondence between semaphores and message passing

| *Semaphore Primitive* | *Corresponding Message Passing Primitive* |
|---|---|
| `sem s` | `op void s()` |
| `P(s)` | `receive s()` |
| `V(s)` | `send s()` |
| `sem s = expr` | `op void s();`<br>`{ int v = expr;`<br>`  for (int i = 1; i <= v; i++) { send s(); }`<br>`}` |

The mapping of semaphore primitives to their general unabbreviated forms is summarized in Table 7.1. A new variable, `v`, is introduced in case the semaphore declaration contains an initialization expression. Its use ensures that *expr* is evaluated just once. If the semaphore is a class semaphore, the code is executed within a static initializer. If the semaphore is an instance semaphore, the code

is executed within a non-static initializer. If the semaphore is local, the code is execute in-line within the block.

To illustrate, recall the solution to the critical section problem given in resource `CS` in Section 6.1. It can be written equivalently using asynchronous message passing as follows:

```
public class CS {
  private static final int N = 20; // number of processes
  private static int x = 0;        // shared variable
  private static op void mutex();  // mutual exlusion for x

  static {
    send mutex(); // initialize mutex to "1"
  }

  private static process p( (int i = 0; i < N; i++) ) {
    // non-critical section
      ...
    // critical section
    receive mutex(); // enter critical section
    x = x + 1;
    send mutex();    // leave critical section
    // non-critical section
      ...
    }
  public static void main(String [] args) {
  }
}
```

The initialization of `mutex` consists of a single send, which places one message on `mutex`'s message queue. When a process attempts to enter its critical section, it attempts to remove an invocation from the message queue. If successful, it continues—`mutex`'s message queue remains empty until that process completes its critical section and sends an invocation to `mutex`. If a process is unsuccessful in its attempt to remove an invocation, it delays until such an invocation arrives and the process is the first process waiting for the invocation. (As with semaphores, processes access message queues in first-come, first-served order.) The message queue of `mutex` will contain at most one invocation; that corresponds to `mutex`'s value as a semaphore being at most one.

Using semaphores rather than the corresponding message passing primitives makes programs somewhat more concise and readable. However, in our implementation of JR, the two classes of mechanisms are equally efficient, and both are more efficient than general message passing. That is, our current JR implementation does not optimize parameterless operations as shown in Table 7.1 into the equivalent semaphore primitive.

## 7.6 Data-Containing Semaphores

A data-containing semaphore is conceptually a semaphore that contains data as well as a synchronization signal. It is, in essence, an unbounded buffer of messages that have been produced and not yet consumed. Such a semaphore is represented by a operation that is shared by several processes. The data is passed as a parameter of the operation. Processes use send and receive statements to append data to or remove data from the operation's message queue; synchronization between processes accessing the queue is implicit through their use of send and receive statements.

A data-containing semaphore can be used, for example, to provide a pool of buffers shared by a group of processes. Consider the following code outline:

```
public class DataContaining {
  private static final int B = 10;  // number of buffers
  private static final int N = 20;  // number of processes
  private static char buffer[] = new char [B]; // char buffers
  private static op void pool(int); // pool of buffer indices
  static {
    for (int i = 0; i < B; i++) { send pool(i); }
  }

  private static process p( (int i = 0; i < N; i++) ) {
    ...
    int x;
    receive pool(x); // request a buffer
    // use buffer[x]
    ...
    send pool(x);    // release the buffer
    ...
    }
  public static void main(String [] args) {
  }
}
```

Operation `pool` represents the buffer pool. The for loop in the static initializer places `N` invocations on `pool`'s message queue; each invocation contains the index of a free buffer. As shown in the above code, a process obtains a buffer by receiving the index of a free buffer from `pool`. When done with the buffer, a process returns it to the buffer pool by sending the index to `pool`. When used in this way, operation `pool` is a bounded buffer. In general, however, such an operation can contain an unbounded number of messages.

The advantage of using an operation's message queue to represent a buffer pool is that it saves the programmer from having to write code that explicitly implements a list of free buffers and code that synchronizes access to the list. The disadvantage is that such use of an operation queue is somewhat less efficient [39].

## 7.7 Shared Operations

Regular semaphores and data-containing semaphores are both examples of operations that are shared by more than one process. That is, more than one process can receive invocations from the operation's message queue. In these cases, the shared operations are declared at the top level within a class as opposed to being declared within processes as in class `Model3` (see Section 7.3). When an operation is shared, servicing processes compete for the operation's invocations. Processes obtain access to pending invocations in a first-come, first-served order.

Shared operations are almost a necessity given that multiple instances of a process can service the same operation. The code in class `CS` (see Section 7.5), for example, demonstrates this point: an invocation of `mutex` can be received by any instance of process `p`. The code in class `Model2` (see Section 7.3) also contains a shared operation, `results`. By convention, however, each client process accesses only its own element of that operation array. Thus, *in effect,* the array elements are not shared in that code. In general, though, even array elements can be shared.

Another useful application of shared operations is for server work queues. In particular, a shared operation can be used to permit multiple servers to service the same work queue. Clients request service by invoking a shared operation. Server processes wait for invocations of the shared operation; which server actually receives and services a particular invocation is transparent to the clients.

As an example of using shared operations for a server work queue, consider the adaptive quadrature method for finding the area under a curve (i.e., a function). Given are a continuous, non-negative function `f(x)` and two values `l` and `r`, with `l < r`. The problem is to compute the area bounded by `f(x)`, the `x` axis, and the vertical lines through `l` and `r`. This corresponds to approximating the integral of `f(x)` from `l` to `r`. The following code outlines a solution. It employs a shared operation, `bag`, which contains a bag of tasks. Each task represents a sub-interval over which the integral of `f` is to be approximated.

```
public class AQ {
  private static final int N = 20; // number of workers
  private static op void bag(double, double, double, double);
  private static op void result(double);
  private static final double Epsilon = 0.001; // convergence test

  private static double f(double x) {
    return Math.pow(x,3.0);
  }

  private static double area = 0.0;
  private static process administrator {
```

```
    double part;
    double l = 0.0, r = 4.0;
    send bag(l,r,f(l),f(r));
    while (true) {
      receive result(part);
      area += part;
    }
  }

  private static process worker( (int i = 1; i <= N; i++) ) {
    double a, b, m, fofa, fofb, fofm;
    double larea, rarea, tarea, diff;
    while (true) {
      receive bag(a,b,fofa,fofb);
      m = (a+b)/2;  fofm = f(m);
      // compute larea, rarea, and tarea
      // using trapezoidal rule
      larea = (m - a) * (fofa + fofm) / 2.0;
      rarea = (b - m) * (fofm + fofb) / 2.0;
      tarea = (b - a) * (fofa + fofb) / 2.0;
      diff  = Math.abs(tarea - (larea + rarea));
      if (diff <= Epsilon) { /* diff small enough */
        send result(larea + rarea);
      }
      else { // diff > Epsilon /* diff too large */
        send bag(a, m, fofa, fofm);
        send bag(m, b, fofm, fofb);
      }
    }
  }

  public static void main(String [] args) {
    // register done as the quiescence operation
    try {
      JR.registerQuiescenceAction(done);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
  }
  private static op void done() {
    System.out.println("computed area = "+area);
  }
}
```

Initially, the administrator places in the bag one task corresponding to the entire problem to be solved. Multiple worker processes take tasks from the bag and service them, often generating two new tasks—corresponding to subproblems—that are put into the bag. Specifically, a worker takes a task—representing the interval `[a, b]`—from the bag, computes its midpoint `m`, and calculates three

areas. The three areas are those of the three trapezoids defined by the points `a`, `m`, and `b` and the value of `f` at these three points. The worker then compares the area of the larger trapezoid with the sum of the areas of the two smaller ones. If these are sufficiently close (within `Epsilon`), the sum of the areas of the smaller trapezoids is taken as an acceptable approximation of the area under `f`, and the worker sends it to the administrator using operation `result`. Otherwise, the worker adds to the bag the two subproblems of computing the area from `a` to `m` and from `m` to `b`.

After initializing the bag of tasks, the administrator repeatedly receives results, which are parts of the total area. It adds these to `area`. The computation terminates (in deadlock) when all message queues are empty and all processes are blocked; i.e., all tasks have been processed by workers, and all results have been received by the administrator. At this point the quiescence operation `done` is executed. `area` is declared as a class variable, so it is accessible to both the administrator process and `done`'s code.

An especially interesting aspect of the above algorithm is that it permits any number of worker processes! If there is only one, the algorithm is essentially an iterative, sequential algorithm. If there are more workers, subproblems can be solved in parallel. Thus the number of workers can be tuned to match the hardware on which the algorithm executes.

Another interesting aspect of the algorithm is that it uses a quiescence operation to print the result. For this problem and many similar ones, it would be difficult for the processes themselves to determine when all tasks had been computed. Here the administrator would have to keep track of which parts of the area had been computed. (It is not sufficient for the administrator to wait for the bag to be empty because a worker may be about to put two new tasks in it.) Letting the JR implementation detect termination, and using a quiescence operation to print the result, yields a much simpler algorithm. Exercise 7.14 explores the above termination issues.

The above example demonstrated the use of a shared class (i.e., static) operation. To examine in more detail that and other ways in which operations can be shared, consider the following program.

```
public class MainSharing {
  public static void main(String [] args) {
    Sharing s1 = new Sharing();
    Sharing s2 = new Sharing();
    send Sharing.f(11);
    send s2.g(22);
    send s2.g(33);
    send s1.g(44);
    send Sharing.f(55);
    send s2.g(66);
    send s1.g(77);
```

```
    int z;
    receive Sharing.f(z);
    receive s1.g(z);
  }
}

public class Sharing {
  public static op void f(int);
  public op void g(int);
  private static process p( (int id = 1; id <= 2; id++) ) {
    int x;
    receive f(x);
  }
  private process q( (int id = 1; id <= 2; id++) ) {
    int x, y;
    receive f(x);
    receive g(y);
  }
}
```

The main method creates two instances of `Sharing`. The `Sharing` class contains two operations, `f` and `g`.

Operation `f` is static, so it can be shared directly by all static and all non-static processes in `Sharing`. In this particular code, there are two static processes (instances of `p`) and four non-static processes (two instances of `q` in each instance of `Sharing`) that can share `f`. Operation `g` is non-static, so an instance of `g` exists within each instance of `Sharing`. Each instance of `g` can be shared directly by only the non-static processes in the same instance.

In addition, operations can be shared indirectly. As shown in the above program, the main method may receive an invocation of `Sharing.f` and may receive an invocation of `s1.g`. Such servicing is permitted here because `f` and `g` are declared in `Sharing` as public. As described in Section 7.2, capabilities for operations—even those operations declared within a process or declared as private—can be passed around a program and serviced anywhere. (Servicing an operation declared in a different virtual machine is also permitted, but doing so incurs additional overhead; see Chapter 10.) Later chapters will illustrate this powerful mechanism. One example uses such servicing to provide a bag of tasks paradigm in a distributed fashion (Section 17.3).

## 7.8 Parameter Passing Details

Parameter passing in JR operation invocations on the same virtual machine follow the same rules as parameter passing in Java method invocations. That is, parameters are passed "by value". Even object references are passed by value. Consider, for example, the following program.

```
public class BasicArraySend {
```

```
  public static void main(String [] args) {
    op void f(int []);
    // generate two invocations of f
    int [] b = new int [2];
    b[0] = 11; b[1] = 34;
    send f(b);
    b[0] = 65; b[1] = 87;
    send f(b);
    // service two invocations of f
    for (int k = 1; k <= 2; k++) {
      inni void f(int [] a) {
        for (int i = 0; i < 2; i++) {
          System.out.println(k +" a["+i+"] "+a[i]);
        }
      }
    }
  }
}
```

Its output is 65 and 87, twice. The first invocation contains (the value of) a reference to an object, the array that `b` was assigned. However, the *contents* of that object are changed before the first invocation is serviced. The above program can be modified to output the two different values for `b` by adding the following statement after the first send statement.

```
b = new int [2];
```

Doing so makes `b` reference a different array.

Parameter passing in JR operation invocations on the different virtual machines follows slightly different rules. See Section 10.7 for details.

## Exercises

7.1 Consider the code in `StreamMerge` (see Section 7.1). Explain the effect, if any, on the program's execution if process two's last send is changed to `send stream1(EOS)`.

7.2 The code in `StreamMerge` (see Section 7.1) assumes that `EOS` is larger than the other values in the stream. This exercise explores removing that assumption.

(a) Run the original program. To make the program concrete, have process one send the stream 1, 3, 5, 7, 9, `EOS` and process two send the stream 4, 8, `EOS`.

(b) Run the program from part (a) with the following values of `EOS`: 99, -99, and 6. Compare each of the outputs from these runs with the output from the program in part (a); explain any differences.

(c) The program in part (b) implicitly assumes that `EOS` appears only at the end of the stream. Or, if `EOS` does appear midstream, it is still interpreted as `EOS`, and thus terminates the stream; subsequent numbers are just ignored. Confirm this behavior by running the program with `EOS` set to 5.

(d) Rewrite the code in `StreamMerge` so `EOS` can be an arbitrary integer value that is not necessarily larger than the other values in the stream. (Assume that `EOS` appears only as stated in part (c).) This program's behavior should be identical to the original program's behavior except for the value of `EOS` it outputs. Run the modified program for the values of `EOS` given in part (b).

7.3 Rewrite the code in `StreamMerge` (see Section 7.1) so it uses a family of two processes to represent the processes that produce the streams and an array of operations to represent the streams.

To make the program concrete, have the first process send the stream 1, 3, 5, 7, 9, `EOS` and the second process send the stream 4, 6, 8, 10, 12, 14, 16, `EOS`.

7.4 Generalize the previous exercise to have `N` processes producing streams.

7.5 Consider the code in class `Cap3` (see Section 7.2).

(a) Rewrite the code so that the two processes are in different classes and `f` is still declared local to `p`.

(b) Rewrite the code so that the two processes are in different classes and `f` is now declared private to the class containing `p`.

(c) Would either of the above two programs or the original program work if `getcap` were declared private? Explain.

7.6 Show the output from the following program on input of 1 and on input of 87.

```
public class CapTest {
  private static op void g(int);
  private static op void f(int x) {
    System.out.println("in f "+x);
    send g(1000+x);
  }
  private static process p {
    int x;
    receive g(x);
    System.out.println("in p "+x);
```

```
  }
  public static void main(String [] args) {
    cap void (int) c;
    int z;
    // read in a value for z
    ...
    if (z < 3) { c = f; }
    else { c = g; }
    c(22); // invokes either f or g
  }
}
```

7.7 Rewrite the code in program `Model4` (see Section 7.3) so it uses arrays of result operations as in program `Model2`.

7.8 The `ResourceAllocatorMultiple` program in Section 7.4 allows clients to request or release resources only one unit at a time. Generalize the program to allow clients to request or release resources multiple units at a time. For simplicity, assume that clients release only those units it currently possesses, as was assumed implicitly in the `ResourceAllocatorMultiple` program. However, clients need not release units in the same groups as they acquired them.

7.9 The `ResourceAllocatorMultiple` program in Section 7.4 could use a more object-oriented style in how it uses messages.

(a) Change `action`'s second parameter from an integer to an instance of a new `MsgKind` class, which encapsulates the different kinds of messages. (That is, simulate an enumeration type using `MsgKind`.) Define the new class and change `ResourceAllocatorMultiple` as needed.

(b) Replace `action`'s second and third parameters with an instance of a new, abstract `Msg` class. Define a subclass, derived from `Msg`, for each kind of message. (The subclass corresponding to the release message will contain the unit identifier.) Change `ResourceAllocatorMultiple` as needed.

7.10 *Atomic Broadcast Problem* (based roughly on Exercise 6.16 in Reference [7]). A single slot integer buffer is shared by one producer process and $N$ consumer processes. The producer repeatedly deposits messages into the buffer; consumers repeatedly fetch messages from the buffer.

(a) A given message is fetched by any $N$ processes, not necessarily all $N$ processes. That is, a given message might be fetched multiple times by some processes and not at all by other processes.

Does your solution also work if there are multiple producers? Explain. (Address multiple producers only in this particular part; do not use multiple producers in your solutions to any other parts of this question.)

(b) Modify your solution to the previous part so that each consumer process fetches exactly once each message the producer deposits.

(c) Modify your solution to the previous part so that only the consumer processes waiting (if any) when the producer deposits a message get to fetch that message.

Your solution must be structurally similar to the `ResourceAllocatorMultiple` code in Section 7.4. This problem requires three kinds of processes that interact with only the indicated interfaces:

- producer — similar to a client. It interfaces to the coordinator by

```
send request(DEPOSIT, UNUSED, somevalue);
receive preply();
```

- consumer, with identity `i` between 0 and $N-1$ — similar to a client. It interfaces to the coordinator by

```
send request(FETCH, i, UNUSED);
receive creply[i](somevariable);
```

- coordinator — similar to the server; it holds the buffer and synchronizes access to it.

Use only `send` and `receive` (no `inni`, `call`, `reply`, `forward`, etc.).

7.11 *Savings Account Problem.* A savings account is shared by several customers of a bank. Each customer may deposit or withdraw funds from the account. Assume that the amount of each deposit or withdrawal is positive. The current balance in the account is the sum of all deposits to date minus the sum of all withdrawals to date. The balance must never become negative, which implies that some withdrawals might need to be delayed.

Your solution must be structurally similar to the `ResourceAllocatorMultiple` code in Section 7.4. Each customer is a client and the bank is the server. On a deposit, your solution must service any waiting withdrawal request(s) that can be serviced. (I.e., it is non-FCFS in part.)

7.12 Copy the outline for the adaptive quadrature program (see Section 7.7) into a file.

(a) Modify the program so that $f(x)$ is $x^2 + 3x$, `Epsilon` is `0.0001`, and the interval is `[1,10]`. Execute the program and compare the computed area with the exact mathematical result.

(b) Modify the original program so that the number of workers, `N`, is a command-line argument. Execute your program using different numbers of workers. How does the performance differ?

(c) Repeat part (b) but for the program from part (a).

7.13 Copy the outline for the adaptive quadrature program (see Section 7.7) into a file.

(a) Modify the program so that $f(x)$ is $x^3 + 2x + 1$, `Epsilon` is `0.0001`, and the interval is `[0,10]`. Execute the program and compare the computed area with the exact mathematical result.

(b) Modify the above program so that `f` is declared as a public static method in a different class.

(c) Modify the program again so that `f` is declared as a public non-static method in a different class, named `fun`.

(d) Modify the part (c) program so that the main invokes `f(0.5)` $T$ times (instead of computing the area). Do not have the program perform any I/O. Time the results ten times using the UNIX csh command "repeat 10 time jrrun AQ" (or the equivalent in another shell).[1] (Also see Exercise 10.10.)

Pick $T$ so that the program runs at least 10 seconds but less than 30 seconds. The specific value for $T$ will depend on the speed of the processor on which the program is run. Start with 1,000 invocations and increase that until a suitable value for $T$ is determined.

Report the value of $T$ and the ten execution times.

7.14 Recall how the adaptive quadrature program (see Section 7.7) depends on JR's automatic program termination detection for its termination and to output its result. Suppose JR did not provide automatic program termination detection. Modify the program so it terminates (and still outputs its result) using

[1] Windows does not provide a direct equivalent of this command. It can be roughly simulated by typing ten times the sequence "time", "jrrun AQ", "time" and calculating the differences in time. Alternatively, one can write a Perl script to measure elapsed time, as suggested in `http://www.perldoc.com/perl5.8.0/lib/Time/HiRes.html`.

(a) The technique suggested earlier in which the administrator keeps track of which parts still need to be computed.

(b) A different technique where the administrator counts how many outstanding tasks need to be completed. Your solution might need to use additional messages between the administrator and the workers.

7.15 Write a program that implements the quicksort method of sorting using a shared bag of tasks. The tasks are slices of the data that need to be sorted. The administrator should put the initial data in the bag. A worker should remove a task, partition it into smaller tasks and put them back in the bag. If a task is small enough, say eight data elements, a worker should instead sort the data and send it to the administrator.

Execute your program using different input data and different numbers of workers. How does the performance differ?

7.16 Develop a distributed program to implement a compare/exchange sorting algorithm. Use `W` worker processes laid out in a line; each worker should communicate only with its one or two neighbors. If there are `N` items to sort, initially each worker should be given `N/W` items. Each worker should sort its items, exchange high and low elements with its neighbors, and repeat until all `N` items are sorted.

(a) Execute your program using different input data and different numbers of workers. How does the performance differ?

(b) Compare the performance of this program to your program for Exercise 7.15.

7.17 Develop a program to generate prime numbers using a shared bag of tasks. The tasks are odd numbers that should be checked for primality. The workers check different candidates. Each worker should have a local table of primes that it uses to check candidates. When a worker finds a new prime, it should send it to all the other workers.

Execute your program for different ranges of primes and different numbers of workers. How does the performance differ?

*This page intentionally left blank*

Chapter 8

# REMOTE PROCEDURE CALL

Remote procedure call (RPC) is another mechanism that is used in many applications. Two processes are involved: the process doing the call (the invoker or client) and the process created to service the call (the server). These processes are typically in different objects and might even be on different virtual or physical machines. The invoking process waits for results to be returned from the call. Thus remote procedure call is synchronous from the client's perspective.

Remote procedure call (or remote method invocation) is accomplished in JR through yet another use of operations. To initiate a remote procedure call, the invoking process calls an operation. The operation is one that is serviced by a method. A call invocation of a remote method results in a process being created to service the invocation. A remote procedure call resembles a sequential procedure call both syntactically and semantically. The fact that the method that services a call might be located on a different virtual or physical machine is transparent to the caller. (However, a remote procedure call takes longer than a local, sequential procedure call.)

This chapter first presents the JR mechanisms for remote procedure call. We then discuss the equivalence between remote procedure call primitives and send and receive primitives. We also describe three statements—return, reply, and forward—that can be used with remote procedure call to obtain additional flexibility. (The use of these statements with rendezvous is discussed in Chapter 9.)

## 8.1 Mechanisms for Remote Procedure Call

The mechanisms for remote procedure call are op-methods (or in their unabbreviated form: operation declarations and methods) and call invocations. Chapter 3 introduced these mechanisms and gave examples of how they are used.

In all cases the semantics of a call invocation to a method is that a new process is created to execute the method's code for the invocation. After initiating the call, the invoking process delays until the invoked process returns results to it. This semantics can often be implemented as a conventional procedure call (see below), but at least conceptually a new process is created to service each call. Figure 2.1 summarizes these actions.

This view of call invocations is useful since the invoked method can be located in another object, which might be located on another virtual or physical machine. (Chapter 10 describes virtual machines and how they are created and placed on physical machines.) This view also covers purely sequential method calls, such as those described in Chapter 3. It is worth noting that process creation is considerably more costly than executing a standard sequential method call. An implementation can optimize many call invocations of methods so they use a less expensive form of procedure call (e.g., see Reference [9]), and the current JR implementation does so to some extent.

To illustrate remote procedure call, consider a (very) simple bank example, with bank accounts and a single bank customer. The bank account is represented by the following code:

```
public class BankAccount {
  public op void deposit(int);
  public op void withdraw(int);
  private int balance = 0;
  public void deposit(int amount) {
    if (amount < 0)
      System.out.println("can't deposit a negative amount");
    else
      balance += amount;
  }
  public void withdraw(int amount) {
    if (amount < 0)
      System.out.println("can't withdraw a negative amount");
    else if (amount > balance)
      System.out.println("can't withdraw more than in account");
    else
      balance -= amount;
  }
}
```

It provides `deposit` and `withdraw` operations. The bank customer is represented by the following code:

```
public class BankCustomer {

  public static void main(String [] args) {
    BankAccount b1 = new BankAccount();
    BankAccount b2 = new BankAccount();
```

```
    b1.deposit(100);  b2.deposit(200);
    // transfer money from b2 to b1
    b2.withdraw(40);  b1.deposit(40);
    ...
  }
}
```

The bank customer instantiates two bank accounts, `bl`and `b2`, and invokes the `deposit` and `withdraw` operations in those instances. Those invocations are call invocations to operations serviced as methods in a different class, `BankAccount`. As coded above, the `BankAccount` objects created by `BankCustomer` will be located on the same virtual machine as `BankCustomer`. Hence the calls of `deposit` and `withdraw` will be local. However, the instances of `BankAccount` could be placed on one or two different virtual machines (see Chapter 10). In that case the calls would be remote. The syntax and semantics of the calls are the same, but the `BankAccount` objects would need to be created using the `remote` keyword (see Chapter 10). Besides that, only the implementation, and consequently the performance, is different.

As mentioned earlier, each call invocation of a method results in a new process, at least conceptually. Suppose, for example, that `BankCustomer` contains several processes, each of which invokes `bl`'s and `b2`'s `deposit` and `withdraw` operations. Each invocation of `deposit` and `withdraw` causes a new process to be created to execute the corresponding code. That can lead to more than one process manipulating the shared bank account variables at the same time: Two processes, each executing on behalf of an invocation of `deposit`, can both attempt to update `balance` at about the same time—i.e., a race condition can result.

To avoid problems of accessing shared data, processes need to synchronize. In the `BankAccount` class, a semaphore can be used to allow at most one process at a time to execute either `deposit` or `withdraw`. In the next chapter, we will see another way to solve this problem using a process and an input statement.

## 8.2 Equivalence to Send/Receive Pairs

A (remote) procedure call can be written equivalently as a send to a method to create the process plus a receive to get back results when the process has completed. Consider, for example, the following simple code outline:

```
public class RPC1 {
  private static op int p(int);
  private static process q {
    int a, b;
    ...
    b = p(a);
    ...
  }
```

```
  private static int p(int x) {
    return x+4;
  }

  public static void main(String [] args) {
  }
}
```

The above code can be written equivalently as follows:

```
public class SRPair1 {
  private static op void p(int);
  private static op void r(int);
  private static process q {
    int a, b;
    ...
    send p(a);
    receive r(b);
    ...
  }
  private static void p(int x) {
    send r(x+4);
  }

  public static void main(String [] args) {
  }
}
```

The operation `p` has been replaced by a new version of `p` and a new results operation, `r`. Note that `p` is now declared as having no return type (i.e., `void`); the return value is now passed back as the parameter of `r`. The call to the method has been replaced by a send/receive pair. The send passes the parameter to `p`; the receive gets the result back from `p`.

The above code works as long as only one process is invoking `p`. However, if more than one process invokes `p`, each needs its own local results operation. For this, we can use local operations and capabilities, as we did in class `Model4` in Section 7.3. For example, suppose `q` is now a family of processes, each of which invokes `p` :

```
public class RPC2 {
  private static op int p(int);
  private static process q( (int i = 1; i <= 20; i++) ) {
    int a, b;
    ...
    b = p(a);
    ...
  }
  private static int p(int x) {
    return x+4;
```

```
  }

  public static void main(String [] args) {
  }
}
```

This code can be written equivalently as follows:

```
public class SRPair2 {
  private static op void p(int, cap void (int));
  private static process q( (int i = 1; i <= 20; i++) ) {
    int a, b;
    ...
    op void r(int);
    send p(a, r);
    receive r(b);
    ...
  }
  private static void p(int x, cap void (int) r) {
    send r(x+4);
  }

  public static void main(String [] args) {
  }
}
```

Here the operation `p` has been replaced by a new version; the new second argument is a capability for an operation that is used to return the result. Each invoking process now declares a local operation, `r`. Again the call to the method has been replaced by a send/receive pair. A sending process passes the capability for its `r` to `p`; `p` sends results back to that operation.

The structure of the above code is very similar to what we saw earlier in the client-server code in class `Model4` (see Section 7.3). In fact, that code can be rewritten to use a call invocation instead of a send/receive pair (see Exercise 8.2). Using a call invocation, though, precludes clients from performing other work while waiting for the server to give back results.

In general, a call invocation provides a cleaner interface than does a send/receive pair. In particular, the passing of results back to the invoker is an implicit part of a call invocation. Using a send/receive pair, on the other hand, requires declaring a local operation to which results get sent and passing a capability for that operation. Using a call invocation also allows invocations of value-returning operations within expressions. However, send/receive pairs are useful when a client wants to do some work between initiating a request for service and picking up results from that request.

## 8.3 Return, Reply, and Forward Statements

The return, reply, and forward statements provide additional flexibility in handling remote procedure calls. They appear in the body of a method and alter the way results are passed back to the invoking process. They can also be used, with similar meanings, as part of the rendezvous mechanism, as discussed later in Chapter 9. (As in Java, the return statement can appear within constructors, but not within initializers. The reply and forward statements are not allowed within constructors or initializers.)

### Return

It has the form

`return` or `return` *expr*

The first form is used for `void` methods and the second for `non-void` methods.

We have already seen the return statement in the context of sequential methods (see Chapter 3). Its meaning there is consistent with its more general meaning, which we give here.

Recall from Section 8.1 the view of a call invocation to a method: it causes a new process to be created to service the invocation. A return statement, then, terminates both the call invocation and the process that executes the return. Any results of the invocation—i.e., the return value—are returned to the invoker.

As a simple example, consider the following code

```
public class Ret {
  private static process p {
    int z;
    ...
    z = f(10);
    ...
  }
  private static op int f(int x) {
    return x*8;
  }

  public static void main(String [] args) {
  }
}
```

Its execution is depicted in Figure 8.1. Process `p` invokes `f` and waits for it to complete. A new process is created to execute the body of `f`. When it reaches the return statement, it evaluates the expression in the return statement, and then terminates. The return value, `80`, is copied back and assigned to `z`, and process `p` continues execution with its next statement.

Sometimes it is useful to allow a method to be invoked by either call or send. For example, the method might update a display screen. In some cases invoking

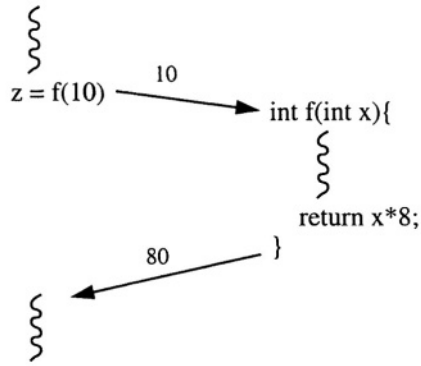


*Figure 8.1.* Execution of simple return program

processes will want to wait for the update to complete; in others they will not. Such a method might execute a return statement. If the method was invoked by send, the return statement just terminates the process that executes the return. Since a send invocation terminates immediately after its parameters are sent to the method, any results from the method are not actually returned.

## Reply

The reply statement is used by a method to continue execution after servicing an invocation of the method. It has the form

**reply** or **reply** *expr*

The first form is used for `void` methods and the second for `non-void` methods. Like a return statement, a reply statement terminates the invocation being serviced by the enclosing method, if it was called. However, a process that executes a reply statement continues executing with the statement following the reply. Such a process may continue to reference the formals until it leaves their scope; however, no subsequent change to formal parameters or to the return value is reflected back to the caller. A reply to a send invocation has no effect; a subsequent reply to an invocation for which a reply has already been executed also has no effect.

As a simple example of reply, consider the following code:

```
public class Rep {
  private static process p {
    int z;
    ...
    z = f(10);
    ...
  }
  private static op int f(int x) {
```

```
    reply x*8;
    x++;
    return x*100;
  }

  public static void main(String [] args) {
  }
}
```

Its execution is depicted in Figure 8.2. The reply statement in `f` terminates

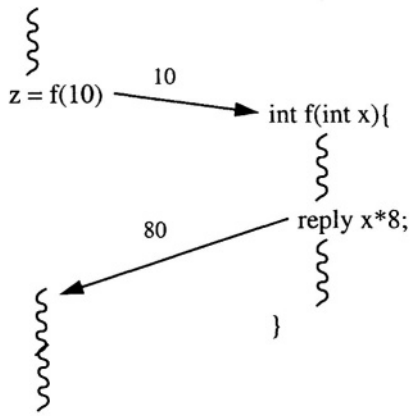


*Figure 8.2.* Execution of simple reply program

the invocation from `p`. At that point, the return value, `80`, is copied back and assigned to `z`, and process `p` continues execution with its next statement. The process executing `f` continues execution, too, with the statements following `reply`. It modifies the value of `x`, evaluates the expression in the return statement, and then terminates. None of those actions has an effect on the caller (but see Exercise 8.6); in particular, it does not change `z`.

The following, more realistic example of a reply also demonstrates one of its common uses—programming what is called *conversational continuity* [7]. A client process creates a server process and wishes to carry out a private conversation with it, i.e., send further requests for work to it. This interaction can be accomplished by having the server process execute `reply`, passing back a capability (or capabilities) for local operations. The following code outline illustrates this technique:

```
public class Conversation {
  private static process client( (int i = 1; i <= 10; i++) ) {
    final int N = 20;
    String t[] = new String [N];
    ...
```

```
    cap void (String) c = server(N);
    for (int k = 0; k < N; k++) {
      send c(t[k]);
    }
  }
  private static op cap void (String) server(int n) {
    op void line (String);
    reply line;
    for (int k = 0; k < n; k++) {
      String s;
      receive line(s);
      // do something with s, e.g., print it
    }
  }

  public static void main(String [] args) {
  }
}
```

The declaration of operation `server` indicates that the server takes an integer parameter and returns a value of `type cap void (String)`, i.e., a capability for an operation that takes a string parameter and has no return value. Here a client process creates a server process, passing it `n`, the number of lines that the client will later send it. The server first assigns a capability for its local operation, `line`, and returns that to its client by executing a reply. The reply allows both the client and the server to continue execution. The client sends `n` messages to its server; the server receives `n` messages from its client. The use of a local operation here ensures that only a server's client can send it messages. Section 18.2 presents another example of the use of `reply` to effect a conversation between components of a distributed file system.

A communication structure similar to the one used above is employed in the following example to implement a parallel sorting algorithm. Sorting is performed by an array of worker processes connected in a pipeline fashion. Each worker keeps the smallest value it sees, and passes all others on to the next worker. For `n` input values, a total of `n` workers are eventually executing. The first worker sees all `n` input values; the last sees just one value. After seeing all the values it will receive, each worker passes back the smallest value it saw.

```
public class PipelineSort {
  public static void main(String [] args) {
    int n;       // number of numbers to sort
    int [] nums = null; // array to sort
    // read in the numbers (one per line to keep it simple)
    try {
      System.out.println("number of numbers?");
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
```

```
    String s = br.readLine();
    n = Integer.parseInt(s);
    nums = new int[n];
    for (int i = 0; i < n; i++) {
      System.out.println("?");
      s = br.readLine();
      nums[i] = Integer.parseInt(s);
    }
  } catch (Exception e) {e.printStackTrace();}
  System.out.println("before sorting");
  call print(nums);
  sort(nums);
  System.out.println("after sorting");
  call print(nums);
}
// Print elements of array a
private static op void print(int[] a) {
  for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
  }
}
private static op void result (int, int); // used by sort
// Sort array a into non-decreasing order
private static op void sort(int[] a) {
  if (a.length > 0){
    cap void (int) first_worker;
    // Call worker; get back a capability for its
    // pipe operation; use the pipe to send all values
    // in a to the worker.
    first_worker = worker(a.length);
    for (int i = 0; i < a.length; i++) {
      send first_worker(a[i]);
    }
    // Gather the results into the right places in a
    for (int i = 0; i < a.length; i++) {
      int pos, value;
      receive result(pos, value);
      a[a.length - pos] = value;
    }
  }
}
// Worker receives m integers on mypipe from its
// predecessor.  It keeps smallest and sends
// others on to the next worker.  After seeing all
// m integers, worker sends smallest to sort,
// together with the position (m) from right in
// array a in which smallest is to be placed.
private static op cap void (int) worker (int m) {
  int smallest; // the smallest seen so far
  op void mypipe(int);
```

```
    reply mypipe; // invoker now has a capability for mypipe
    receive mypipe(smallest);
    if (m > 1){
      cap void (int) next_worker; // pipe to next worker
      // create next instance of worker
      next_worker = worker(m - 1);
      for (int i = m - 1; i >= 1; i--) {
        int candidate;
        // save new value if it is smallest
        // so far; send other values on
        receive mypipe(candidate);
        if (candidate < smallest) {
          int tmp = smallest;
          smallest = candidate;
          candidate = tmp;
        }
        send next_worker(candidate);
      }
    }
    send result(m, smallest); // send smallest back to sort
  }
}
```

Each worker, other than the last one, creates the next worker in the pipeline. A worker uses a reply statement to pass a capability for its `mypipe` operation back to its invoker. The first worker passes the capability back to the process executing `sort`; each other worker passes it back to the previous worker.

Above, the worker processes are created dynamically, so exactly as many as are required (`n`) are created. This necessitates the use of local operations (`mypipe`) and capabilities for these operations. Exercise 8.8 explores a more static version of this problem.

## Forward

The forward statement defers replying to a called invocation and instead passes on this responsibility. The fact that an invocation was forwarded is transparent to the original invoker.

The forward statement names an operation and contains a list of expressions separated by commas:

`forward` *operation* `(` *expr*`,` *expr*`,` ... `)`

Execution of forward takes the operation invocation currently being serviced, evaluates a possibly new set of arguments, and invokes the named operation. An invocation can be forwarded to any operation having the same return type, including the operation being serviced! (See Exercise 8.13.) If the invocation being serviced was called, the caller remains blocked until the new invocation has been serviced (to completion).

After executing forward, the forwarding process continues with the next statement just as if it had replied to the forwarded invocation. During the continued servicing of the invocation after the forward statement, a subsequent forward or reply behave differently from usual. A subsequent forward is treated as if it were a send invocation. A subsequent reply to a forwarded invocation has no effect. Similarly, a subsequent return from within the block handling the invocation has no effect on the caller if the invocation was called, although it has the usual effect of causing the executing process to exit that block.

While within the block handling a forwarded invocation, the forwarding process may still reference formal parameters. However, no subsequent changes to parameters or to the return value will be seen by any other process.

As a simple (contrived) example, consider the following program fragment:

```
public class Fwd {
  private static process p {
    int a = f(1); System.out.println(a);
  }
  private static op int f(int x) {
    forward g(2*x);
    ... // continue execution, perhaps changing x
    return 0;
  }
  private static op int g(int y) {
    return y+10;
  }

  public static void main(String [] args) {
  }
}
```

Its execution is depicted in Figure 8.3. First, process `p` invokes `f`. The process executing `f` doubles its argument `x`, forwards the invocation to `g`, and then continues executing; the process may change `x` and execute a return statement, but this has no effect on the result returned to process `p`. Forwarding to `g` causes a new process to be created; it adds 10 to its argument and returns the value to `p`, which is waiting for its invocation of `f` to return. The overall effect, then, is that variable `a` is assigned 12.

A more realistic example of the use of `forward` is the following. Client processes make requests for service to a central allocator process. The allocator assigns a server process to the request by forwarding the invocation to it. To be more concrete, the allocator might represent a file server to which clients pass the names of files. The allocator determines on which server the requested file is located and forwards the client's request to the server, which typically would be located on a different machine. Section 18.2 explores this example and its use of `forward` in detail. Because only the return types of the original

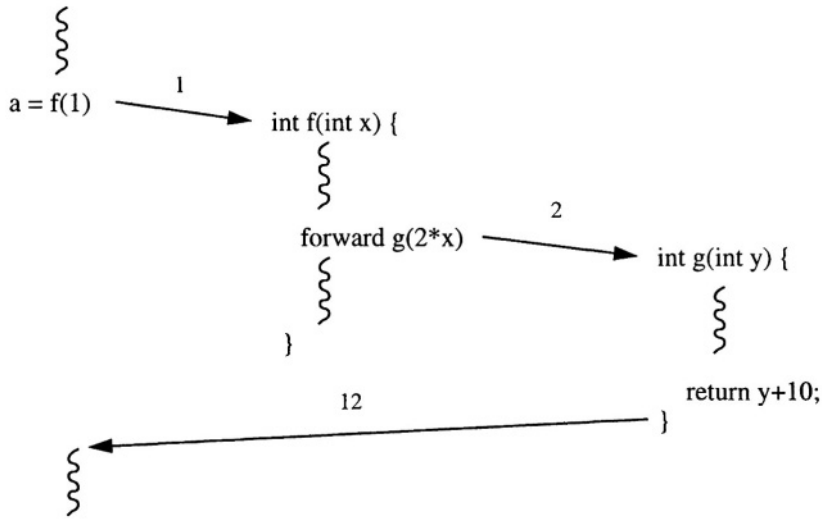


*Figure 8.3.* Execution of simple forward program

operation and of the operation to which the invocation is being forwarded must be identical, the allocator can forward a different set of parameters to the server. It may add parameters — e.g., representing the results of its computation on the invocation — or omit parameters — e.g., dropping those needed only by the allocator.

## Exercises

8.1 Consider the bank example in Section 8.1. Give a detailed explanation (in the style of Section 4.1) of how two processes, each executing on behalf of an invocation of `deposit`, can cause a race condition on the account's balance.

8.2 Show how to rewrite the client-server model in class `Model4` (see the end of Section 7.3) using a call statement instead of send and receive statements. Assume clients do no work between the send and receive.

8.3 Suppose method `p` is declared as

```
private static op int p(int x) {
  return x*x;
}
```

Show how to rewrite `p` and calls to it using send/receive pairs.

8.4 Suppose method `p` is declared as

```
static int z = 0;
private static op int p(int x) {
  int y = x*x;
  reply y;
  z += y;
  return 0;
}
```

Show how to rewrite `p` and calls to it using send/receive pairs. Discuss the need and desire for a reply statement in JR.

8.5 Suppose method `p` is declared and invoked as

```
private static op void p(cap void (int) rep, int n) {
  reply;
  send rep(n+1000);
}
private static process q {
  ...
  op void rep(int);
  p(rep, 33);
  int z;
  receive rep(z);
  ...
}
```

Show how to rewrite `p` and calls to it using send/receive pairs.

8.6 In the `Rep` program (see Section 8.3), the actions of the process executing `f` after it replies do not affect the caller. However, the actions might affect the caller if the statements following the reply have side effects that are visible globally. Modify the `Rep` program to demonstrate that.

8.7 Recode the `Conversation` class (see Section 8.3) so that clients are in a different class than servers.

8.8 Show how to code the `PipelineSort` program (see Section 8.3) using an array of operations instead of local operations. Assume exactly `N` values are to be sorted, where `N` is a declared constant rather than an input value.

8.9 Recode the `PipelineSort` program (see Section 8.3) so that workers are not passed the number of numbers to expect. Instead they are passed a sentinel (e.g., `0`) after all legitimate numbers.

Do not pass anything as a parameter to the worker. Pass the `mypipe` operation only the numbers in the stream followed by the sentinel. Do not pass any additional messages. Use no global variables.

8.10 Repeat the previous exercise, but assume each worker is sent a special done message (represented by a new operation declared local to the worker) after all legitimate numbers.

The solution requires an input statement, which was introduced in Chapter 1 and is discussed in detail in Chapter 9. (Hint: the worker needs to return two values; see the `Sieve` program in Section 9.10.)

8.11 Consider the `PipelineSort` program (see Section 8.3). In gathering back the results, the sort method uses the statements

```
receive result(pos, value);
a[a.length - pos] = value;
```

Can those be combined into a single statement? Explain your answer.

8.12 Consider the `PipelineSort` program (see Section 8.3). Each worker passes back the smallest number it has seen and its worker id, so sort can place the number in the correct position of the array. Show how to eliminate passing back the worker id. In particular, each worker now passes back only its value of `smallest`. (Hint: Have each worker, except the last, pass back its value of `smallest` just before it passes the last number to the next worker.)

8.13 Write a factorial procedure that is (technically) not iterative or recursive. Use a forward statement to achieve the effect of a loop or a recursive procedure call. Do not use a loop or a recursive call.

8.14 A monitor is a modular synchronization mechanism (see Reference [25], Reference [7], and Chapter 21). It exports a collection of procedures that are called by processes that want to communication and synchronize. The procedures execute with mutual exclusion and use what are called condition variables for condition synchronization.

Show how to simulate a monitor using RPC and semaphores. Illustrate your simulation by converting one of the monitors in Reference [25], Reference [7], or Chapter 21 into a JR class.

*This page intentionally left blank*

Chapter 9

# RENDEZVOUS

A rendezvous, like a remote procedure call, involves two processes: an invoking process and a process that handles the invocation. However, the invocation is handled by an existing process; a new process is *not* created as a result of the invocation. As with remote procedure call, rendezvous is synchronous from the invoking process's perspective, and the two processes can be located on different virtual or physical machines. Figure 2.1 summarizes these actions.

Rendezvous, like JR's other synchronization mechanisms, is accomplished through the use of operations. The invoking process uses a call invocation of an operation to initiate a rendezvous. The operation is one that is serviced by an existing process executing what is called an input statement.[1] JR's input statement allows a process to wait for one of several operations to be invoked. It also allows a process to base its decision as to which invocation to service on the values of the invocation parameters.

This chapter first discusses the general form of the input statement and gives a few simple examples. We then show how the receive statement is an abbreviation for a common form of input statement. In Chapter 7 the receive statement was used to service send invocations; `receive` can also be used to service call invocations, which results in synchronous message passing. The remainder of the chapter describes other aspects of the input statement—synchronization expressions, scheduling expressions, conditional input, and servicing arrays of operations—and its use with other statements such as break and reply. (Although we focus on rendezvous, we also show how the input statement can be used to handle send invocations.) These various aspects give the input statement much flexibility in handling invocations. However, there are some applications

[1] This kind of rendezvous is sometimes called an extended rendezvous to contrast it with the simple rendezvous of synchronous message passing, as in Communicating Sequential Processes (CSP) [26].

for which it does not provide sufficient flexibility. Chapter 14 discusses these applications and additional, more flexible invocation selection mechanisms.

## 9.1 The Input Statement

The input statement is JR's most complicated statement in that it has many optional parts that can affect its execution. On the other hand, it is also JR's most powerful statement, as we shall see. In this section we describe the input statement's general form and give a few simple examples. Subsequent sections explore the parts in detail and give numerous additional examples.

### 9.1.1 General Form and Semantics

An input statement[2] contains one or more operation commands separated by brackets that form boxes:

`inni` *op_command* `[]` *op_command* `[]` ...

Each operation command specifies an operation to service, an optional synchronization clause, an optional scheduling clause, and a block of code. An operation command has the general form

*return_type op_expr* `(` *formal_list* `)` `st` *synch_expr* `by` *sched_expr block*

The *op_expr* is any expression that evaluates to an operation (e.g., name or capability), just as it was for the receive statement (Section 7.1). The formal list contains types and names for the parameters, just as in Java's method headers. The *op_expr* may also contain a `throws` clause, just as in Java's method headers; we discuss exception handling for operations in Chapter 12. The keyword `st` (such-that) introduces the *synchronization expression.* It specifies which invocations of the operation are acceptable. The keyword by introduces the *scheduling expression.* It dictates the order in which invocations are serviced.

In general, an input statement can service any operation declared in a scope that includes the statement. The same operation can even appear in more than one operation command (see Exercises 9.8, 9.9, and 9.10). An input statement can also service an operation that is not in scope by specifying a capability for the operation, similar to what was shown for receive statements in Sections 7.2 and 7.7. Thus, the servicing of a given operation can be shared by more than one process, if desired.

Although not shown above, the operation in an operation command can be an element of an array of operations, in which case it must be subscripted to indicate the particular operation to be serviced. Furthermore, a quantifier can

[2] The keyword `inni` combines the keywords delimiting the start of and end of SR's input statement: `in` and `ni`. An early version of JR used `in`, but that turned out to conflict with the use of that identifier by some Java libraries.

be used to specify that any one of a group of elements of an operation array is to be serviced. Arrays of operations and quantifiers are discussed in Section 9.8.

A process executing an input statement is in general delayed until some invocation is selectable. An invocation is *selectable* if the boolean-valued synchronization expression for the corresponding operation is true (or is omitted). In general, the oldest selectable invocation is serviced. However, if the corresponding operation command contains a scheduling expression, the invocation that is serviced is the oldest one that is selectable and that also minimizes the scheduling expression. Both synchronization and scheduling expressions can reference invocation parameters, thereby allowing selection to be based on their values. If no invocations are pending for an input statement, the process executing that statement delays until a selectable invocation is received.

An invocation is serviced by executing the corresponding block. The input statement terminates when that block terminates. If the invocation was called, the corresponding call statement also terminates at that time.

The operation commands in an input statement can be followed by an else command, which has the form

**[] else** *block*

This block of code is executed if no invocation is selectable. Thus a process will never delay when it executes an input statement containing an else command.

For readers familiar with Ada, JR's input statement combines and generalizes aspects of Ada's `accept` and `select` statements. In particular, an input statement is like a `select` statement in which each arm is an `accept` statement. Also, JR's else command is similar to Ada's `otherwise` clause in a `select` statement. The notable differences between JR's and Ada's rendezvous mechanisms are: JR allows synchronization expressions to reference parameters of operation invocations, whereas Ada does not; and JR includes a scheduling expression, which Ada does not. These make the input statement much more expressive and powerful, but they do add some implementation overhead when used (see Appendix D).

### 9.1.2 Simple Input Statements

As a first example of an input statement, consider the code outline below. The two processes, `p` and `q`, interact via a rendezvous through operation `f`. Process `p` calls `f(y)` and then waits until process `q` receives `y` and increments its local variable `z`; that is, `p` delays until `q` reaches the end of the command block associated with `f`. If process `q` arrives at its input statement and finds no pending invocations of `f`, it delays until `p` invokes `f`.

```
public class Basic {
  private static op void f(int);
  private static process p {
```

```
    int y;
    ...
    call f(y);
    ...
  }
  private static process q {
    int z;
    ...
    inni void f(int x) { z += x; }
    ...
  }
  public static void main(String [] args) {
  }
}
```

The following code outline presents a slightly more complicated example of an input statement:

```
public class More {
  private static op void f(int);
  private static op double g(double);
  private static process p1 {
    int y;
    ...
    call f(y);
    ...
  }
  private static process p2 {
    double w;
    ...
    w = g(3.8);
    ...
  }
  private static process q {
    int z;
    ...
    inni void f(int x) { z += x; }
    [] double g(double u) { return u*u-9.3; }
    ...
  }

  public static void main(String [] args) {
  }
}
```

Here three processes interact via two rendezvous: `p1` and `q` interact through operation `f`, and `p2` and `q` interact through operation `g`, which returns a value. The input statement allows process `q` to service either an invocation of `f` or an invocation of `g`. When `q` reaches its input statement, it finds one of three states:

- An invocation of only one of `f` or `g` is pending—`q` will service the operation that has pending invocations.
- Invocations of both `f` and `g` are pending—`q` will service the invocation that arrived first.
- No invocations are pending—`q` delays until either `f` or `g` is invoked.

After invoking `f`, process `p1` delays until its invocation is serviced by process `q`, i.e., until `q` reaches the end of the block in the corresponding operation command. Similarly, after invoking `g`, process `p2` delays until its invocation is serviced by process `q`. As programmed above, only one rendezvous will actually occur; if the input statement were embedded in a loop and executed twice, both would occur but in an unpredictable order.

The next example shows how input statements can be nested, even when they service the same operation. It also demonstrates that the formal identifiers used in input statements for a particular operation can be named differently in different input statements for that operation, and they must be in this particular case. Consider the following code outline:

```
public class Nested {
  private static op String swap(String);
  private static process p1 {
    String y;
    ...
    y = swap(y);
    ...
  }
  private static process p2 {
    String z;
    ...
    z = swap(z);
    ...
  }
  private static process q {
    ...
    inni String swap(String x1) {
      String r1 = null; // return value for outer swap
      inni String swap(String x2) {
        r1 = x2;
        return x1;
      }
      return r1;
    }
    ...
  }

  public static void main(String [] args) {
```

```
  }
}
```

Processes `p1` and `p2` both invoke `swap` to exchange values. Process `q` uses a nested input statement to service one invocation of `swap` within another. It services one of the calls of `swap` with the outer input statement and another with the nested input statement. In the innermost block, `q` has access to the parameters of both invocations of `swap` because they have been given different local names; as in Java, local names in nested blocks must be unique.

## 9.2 Receive Statement Revisited

The receive statement, introduced in Chapter 7, is an abbreviation for a simple form of input statement. For example, if operation `f` is declared with two `int` parameters, then

```
receive f(v1, v2);
```

is equivalent to the following input statement:[3]

```
inni void f(int p1, int p2) { v1 = p1; v2 = p2; }
```

In the input statement, each variable that appears in the receive statement is assigned the value of the corresponding parameter. Variables are assigned to in left-to-right order. If the same variable appears more than once (which is probably bad programming practice), only the last assignment is visible.

The receive statement was used in Chapter 7 to service send invocations. It can also service call invocations. The effect is what is called *synchronous message passing*: The calling process delays until a receiving process accepts its message. For example, suppose `foo` is an operation used by two processes as follows:

```
private static op void foo(int);
private static process p {
  ...
  call foo(v);
  ...
}
private static process q {
  int x;
  ...
  receive foo(x);
  ...
}
```

[3]The two variables must also have been initialized before the input statement.

Assume `foo` is used only by these two processes and that there are no pending invocations. Then the first process to arrive at its communication statement delays until the other arrives; the value of `v` is then assigned to variable `x`, and both processes continue execution. When `call` and `receive` are used as above, they are like the output and input commands, respectively, in CSP [26] and occam [14, 46].

The input statement has been used thus far in this chapter to service call invocations. It can also service send invocations (even if the block in the operation command is not empty). As an example of the use of send invocations with the receive form of input statements, consider the following code for a process that allocates a single unit of some resource:

```
// single unit allocator
while (true) {
  receive request(); receive release();
}
```

It repeatedly services two operations, `request` and `release`, in that order. To gain access to the resource, a process calls `request`. To release its access to the resource, a process could call `release`. However, there is no need for a releasing process to wait for the allocator to service its release (unlike the request). Therefore, sending to `release` is better. This example was seen as `ResourceAllocatorSingle` in Section 7.4, where only `send` and `receive` were used. The use of `call` here obviates the need for an explicit reply operation.

The receive statement can be used for simple forms of process interaction, but the input statement must be used for more complicated forms. A receive statement does not provide a process the means to wait for one of several operations to be invoked, in contrast to the input statement in the `More` class in Section 9.1. It also does not allow synchronization and scheduling expressions, which are discussed in subsequent subsections. Finally, receive supports only one-way information flow from the invoker to the receiving process; it cannot be used to return a value from a non-void operation. [4]

Consider, for example, the following code fragment that implements a single slot buffer (mailbox):

```
private static op void deposit(int);
private static op int fetch();
private static process manager {
```

[4]The receive statement can still service such an operation, but the value of the result returned is undefined. For a send invocation, the undefined value does not cause problems because the return value is not accessible to the invoker. For a call invocation, though, the undefined value can cause problems because the return value is accessible. The current JR implementation issues a translation-time warning for a receive statement for a non-void operation.

```
int buffer = 0;
while (true) {
  inni void deposit(int item) { buffer = item; }
  inni int fetch() { return buffer; }
}
```

The `manager` process repeatedly services a `deposit` operation and then a `fetch` operation. Thus synchronization to the single slot buffer is provided by "flow of control." For example, if an item has just been deposited into the buffer, the manager can service only a `fetch`, not another `deposit`. The first input statement above can be written more concisely with a `receive` statement:

```
receive deposit(buffer);
```

The second input statement cannot be written in this way, however, since `fetch` has a return value.

Using a call invocation serviced by an input statement can simplify some send/receive interactions. For example, some of the client-server models in Section 7.3 can be simplified. In particular, both the code that used an array of operations (class `Model2`) and the code that used capabilities (class `Model3`) can be rewritten as follows:

```
public class Model {
  private static final int N = 20; // number of client processes
  private static op double request(int, char);

  private static process client( (int i = 0; i < N; i++) ) {
    ...
    double d;
    d = request(i, 'w');
    ...
  }
  private static process server {
    while (true) {
      inni double request(int id, char data) {
        // handle request; put answer in ans
        double ans;
        ...
        return ans;
      }
    }
  }
  public static void main(String [] args) {
  }
}
```

The send/receive pair in the client has been replaced by a call invocation of `request`. The `request` operation now returns a value. The server uses an

input statement to service each request. When the server reaches the end of the input statement, it passes results back. This structure, though, precludes clients from performing other work while waiting for the server to give back results.

The `request` operation above is serviced by a single server process. Hence invocations of `request` are serviced one at a time, i.e., with mutual exclusion. If `request` had been implemented by a method instead of an input statement, instances of the method could execute concurrently but then the programmer would have to program mutual exclusion, using semaphores, for example.

Section 7.7 discussed how shared operations can be used with send and receive statements. The operations that are invoked as part of rendezvous—or more generally, serviced by input statements—follow the same rules.

## 9.3 Synchronization Expressions

A boolean-valued synchronization expression can be used to control which invocation an input statement services next. Consider the following input statements:

```
inni void a(int x) st c > 0 { ... }

inni void a(int x) st x == 3 { ... }

inni void a(int x) st x == 3 { ... }
[] void b(int y, int z) st y == f(z) {...}
```

The first input statement services an invocation of `a` only when the value of variable `c` is positive. The second input statement services only invocations of a whose parameter `x` is equal to `3`. The third input statement services the same invocations as the second, as well as invocations of `b` whose parameters satisfy the condition `y == f(z)`, where `f` represents a user-defined or predefined operation. The exact order in which such invocations are serviced is defined in Section 9.5.

It is important to emphasize that synchronization expressions can reference invocation parameters. That ability leads to straightforward solutions to many synchronization problems, as we shall see later in this chapter and in Part I. However, it is more expensive to implement such synchronization expressions since it requires searching the queue of pending invocations (see Appendix D). This tradeoff between expressive power and implementation cost is a fundamental aspect of language design.

As a more realistic application of a synchronization expression, consider a process that manages a multiple unit resource. This example was first seen in Section 7.4. Each client process requests or releases a single unit at a time. The single-unit allocator code given in Section 9.2 would not work here because the manager now needs to be able to service either a request operation or a release

operation. A two-arm input statement with a synchronization expression can be used for that purpose:

```
// multiple unit allocator for M units
int avail = M;
while (true) {
  inni void request() st avail > 0 { avail--; }
  [] void release() { avail++; }
}
```

The synchronization expression with `request, avail>0`, ensures that a resource is only allocated to a requesting process if at least one unit of the resource is available. If a process requests a resource and none is available, the invocation is not serviced—and the caller delays—until a resource becomes available. The `ResourceAllocatorMultiple` program in Section 7.4 solved this problem using only `send` and `receive`. The use of the input statement and `call` here significantly simplifies the solution.

The next example presents a solution to the classic bounded buffer problem; it generalizes the single slot buffer example in Section 9.2 The `BoundedBuffer` program provides two operations: `deposit` and `fetch`. A producer process calls `deposit` to insert an item into the buffer; a consumer process calls `fetch` to retrieve an item from the buffer. An input statement synchronizes how invocations of `deposit` and `fetch` are serviced to ensure that messages are fetched in the order in which they were deposited, are not fetched until deposited, and are not overwritten. The code is as follows:

```
private static op void deposit(int);
private static op int fetch();
private static process manager {
  int [] buf = new int [size];
  int count = 0, front = 0, rear = 0;
  while (true) {
    inni void deposit(int item) st count < size {
      buf[rear] = item;
      rear = (rear+1) % size;
      count++;
    }
    [] int fetch() st count > 0 {
      int item = buf[front];
      front = (front+1) % size;
      count--;
      return item;
    }
  }
```

The manager process loops around a single input statement, which services `deposit` and `fetch`. The synchronization expressions in the input statement

ensure that the buffer does not overflow or underflow. For example, a producer is delayed if the buffer is full and a consumer is delayed if the buffer is empty.

A bounded buffer provides functionality somewhat similar to a stack. Both provide similar operations: deposit and fetch versus push and pop. They differ, though, in that the stack operations enforce a last-in, first-out discipline whereas the bounded buffer operations enforce a first-in, first-out discipline. The implementations of these operations also differ in how overflow and underflow are handled. For the stack those invocations are handled as errors; for the bounded buffer those invocations are not serviced until they can be handled without error. Thus a bounded buffer is a synchronized queue. The same technique used for programming the bounded buffer can be used to program a synchronized stack (see Exercise 9.12).

The readers/writers problem is another classic synchronization problem [16]. Two classes of processes want to access a database (or file or set of shared variables). Reader processes only examine the database; hence they can execute concurrently with each other. Writer processes update the database; to keep the database consistent, they must have exclusive access to it. Suppose that reader processes call operation `start_read` before reading the database and call (or send to) operation `end_read` when done. Similarly, writer processes call `start_write` before and call (or send to) `end_write` after writing to the database. The following process implements the specified reader/writer exclusion:

```
private static op void start_read();
private static op void end_read();
private static op void start_write();
private static op void end_write();

private static process RWAllocator() {
  int nr = 0, nw = 0;
  while (true) {
    inni void start_read()  st nw == 0            { nr++; }
    []   void end_read()                          { nr--; }
    []   void start_write() st nw == 0 && nr == 0 { nw++; }
    []   void end_write()                         { nw--; }
  }
}
```

Variables `nr` and `nw` count the number of active readers and writers, respectively. Reader processes can start reading when there are no active writers; a writer process can start writing when there are no active readers or writers. The process above implements what is called a *readers' preference* solution. Namely, if there is a steady stream of readers, the value of `nr` will always be positive, and hence a writer may never get to access the database. For example, suppose start requests arrive in the order

R1 R2 W1 R3

Suppose that R1 does not finish reading until after R3's request arrives. Then, R3 would be serviced and W1 would be made to wait until there are no active readers, which as noted above will not occur if there is a steady stream of readers. The solution can be modified to give writers preference or to guarantee eventual access to all processes (see Section 9.10 and Exercise 9.13).

One predefined method dealing with operations is particularly useful in synchronization expressions. The expression `f.length()` returns the number of pending invocations of operation `f`. This method can, for example, be used to give preference to servicing invocations of one operation over another:

```
// preference to servicing invocations of f over g
inni void f() { ... }
[] void g() st f.length() == 0 { ... }
```

The synchronization expression on the second arm of the input statement says that an invocation of `g` should be serviced only if no invocations of `f` are pending. Thus the above input statement gives preferential service to invocations of `f` over those of `g`.

## 9.4 Scheduling Expressions

A scheduling expression can be used to control which invocation—of those whose synchronization expression is true—an input statement services next. Preference is given to the selectable invocation that minimizes the value of the scheduling expression; if there is more than one such invocation, the oldest is selected. The type of a scheduling expression may be any ordered type—such as `int`—or `double`. Scheduling expressions, like synchronization expressions, make it easy to solve many problems, but they do incur an implementation cost since all pending invocations have to be examined.

Consider the following three input statements:

```
inni void a(int x) by x { ... }

inni void a(int x) st x%2 == 0 by -x { ... }

inni void a(int x) st x%2 == 0 by -x { ... }
[] void b(int y, int z) by y+z { ... }
```

The first input statement services invocations of `a`, giving preference to those with smaller values of `x`. The second services only invocations of `a` whose parameter `x` is even, giving preference to those with larger values of `x`. The third input statement services the same invocations of `a`, in the same order, as the second input statement as well as services invocations of `b`, giving preference to those with smaller values of `y+z`. The exact order in which such invocations are serviced is defined in Section 9.5.

A more realistic example of the use of a scheduling expression is a variant of the single unit allocator given in Section 9.2. Suppose each requesting process includes in its request the length of time it expects to use the resource; the allocator gives preference to the request with the smallest time. The following implements this shortest-job-next (SJN) allocation scheme:

```
// single unit allocator; SJN variant
while (true) {
  inni void request(int usage_time) by usage_time {}
  receive release();
}
```

The scheduling expression of the input statement causes invocations of `request` to be serviced in increasing order of the value of `usage_time`.

## 9.5 More Precise Semantics

As seen already in this chapter, the input statement generally services invocations in FCFS order based on their time of arrival. This order, however, can be altered if the input statement contains a synchronization or scheduling expression. Exactly which invocation an input statement services depends on what invocations are pending and when they arrived.

The general rule is as follows. The operation with the oldest pending invocation (regardless of whether the synchronization expression is true for that invocation) is checked first to determine if any of its invocations satisfy the synchronization expression. If so then that invocation is serviced or, if a scheduling expression is present, the oldest invocation that satisfies the synchronization expression and that minimizes the scheduling expression is serviced. Otherwise the operation (of those remaining) with the oldest pending invocation is checked next. The above process repeats until a suitable invocation is serviced or none is found.

Consider again the third input statement at the beginning of Section 9.3. Suppose that `f(z)` returns `2*z` and that the following invocations are pending (in order of arrival):

```
b(8,4)  b(0,9)  a(3)  a(4)  b(4,2)
```

Then the input statement will service the invocation `b(8,4)`. If the same input statement is executed a second time, it will service the invocation `b(4,2)`. In this second execution the invocation `b(0,9)` is the oldest, so the input statement first tries to find any `b` invocations that satisfy the synchronization expression, even though the invocation `b(0,9)` itself does not.

Now consider again the third input statement at the beginning of Section 9.4 with the above group of pending invocations. The first execution of the input statement will service the invocation `b(4,2)`. Operation `b` is chosen because it has the oldest invocation; invocation `b(4,2)` is chosen because it minimizes the

scheduling expression. The second execution will service invocation `b(0,9)` and the third will service invocation `b(8,4)`. Then, the fourth will service invocation `a(4)`. (The above assumes that no new invocations arrive because they could be selected.)

Synchronization and scheduling expressions should not have side effects. Although such is allowed, whether such side effects will occur and how often is not defined. For example, the output from the following program fragment under the current JR implementation is "2000 2001".

```
send f(2000);
send f(2001);
int z = 9999;
inni void f(int x) st ((z=x) % 2 == 0) by (z=-x) {
  System.out.println(x + " " + z);
}
```

Note how, perhaps unexpectedly, the second value output is positive and is the value of x in the second invocation, not the value of x in the first invocation, which is the invocation actually selected for servicing. If predictable side-effects are desired, then the behavior of `st` and `by` can be directly emulated using the features described in Chapter 14.

## 9.6 Break And Continue Statements

As in Java, the break statement forces early termination of a loop. The continue statement forces return to the loop control in such a statement. Break and continue statements may also appear within an input statement that is nested within a loop. In these cases execution of a break or continue statement also forces the input statement to terminate.

For example, consider the following code outline, which illustrates a common server structure:

```
private static process server {
  while (true) {
    inni void work(...) { ... }
    [] void done(...) { break; }
  }
}
```

A client process repeatedly makes requests for service by invoking work, usually by a call invocation. When a client is through using the server, it so informs the server by invoking `done`. The server process repeatedly services invocations of `work`. When it services an invocation of `done`, it exits the loop; the break statement also terminates the invocation of `done`. We will see this structure again in later examples.

If a break or continue statement is executed within the arm of an input statement whose operation returns a value, a return value must be passed back to

the invoker. A reply statement can be used for that purpose; see Section 9.10 for an example.

## 9.7 Conditional Input

An input statement's last operation command can be an else command. As described earlier, the block of code associated with the else command is executed if none of the input statement's operation guards is true; when that block terminates, so does the input statement. The use of an else command supports *conditional input* since the executing process has control over whether it waits for an operation to be invoked.

For example, consider the following program fragment:

```
inni void a() { x = 1; } [] else { x = 2; }
```

If an invocation of a is present, the executing process executes the first assignment; otherwise it executes the second assignment. If only one process services invocations of a, the above input statement is equivalent to[5]

```
if (a.length() > 0 ) { receive a(); x = 1; }
else { x = 2; }
```

However, if more than one process receives invocations of a, then the above two input statements are not equivalent because a process executing the second statement could evaluate `a.length()`, find it positive, and yet block at the receive statement. (Other processes could grab all pending invocations after `a. length()` is evaluated.) In contrast, an input statement with an else command never causes a process to block.

Conditional input is useful in a number of situations. One is illustrated by the following code outline:

```
while (true) {
  // do some work
  ...
  // terminate loop if told; otherwise, iterate
  inni void done() { break; } [] else {}
}
```

Here a process repeatedly does some work and then checks for an invocation of the `done` operation, indicating that it should terminate. If no such invocation is present, the process just continues with the next loop iteration.

A second situation in which conditional input is useful is typified by the following code fragment:

[5]It is equivalent if a is invoked by `send`. However, if a is invoked by `call`, the caller continues before `1` is actually assigned to `x`. This difference does not matter if `x` is local to the process executing the receive statement; it might matter if `x` is shared.

```
// service all invocations of a for which x == t
while (true) {
  inni void a(int x) st x == t { ... } [] else { break; }
}
```

The overall effect of the loop is to service all pending invocations of a whose parameter `x` is equal to `t`. On each iteration of the loop, the input statement services one such invocation, if there is one, or exits the loop.

## 9.8 Arrays of Operations

Input statements (or semaphore `P` statements or receive statements) can also be used to service arrays of operations. A particular element of an array of operations is serviced by specifying its index. For example, suppose `N` is a constant and consider the following program outline:

```
private static cap void (int) f[] = new cap void (int)[N];
static {
  for (int i = 0; i < N ; i++ ) {
    f[i] = new op void (int);
  }
}
private static process q( (int i = 0; i < N; i++) ) {
  inni void f[i](int x) { ... }
}
```

Each process services one element of the array `f`.

An input statement can service any one of a group of elements of an operation array by specifying a quantifier. For example, consider the following input statement:

```
// wait for any one of signal[0:3] to be invoked
inni ( (int i = 0; i < 4; i++) ) void signal[i](int x) {
  System.out.println(i + " got " + x);
}
```

Note how the scope of the quantifier variable extends to the end of the command block. Thus in the above, `i` can be used within the command block. Its value indicates which element of `signal` is being serviced.

When a quantifier on an arm of an input statement specifies an empty range of index values, then that entire arm of the input statement is ignored. The behavior of the process executing the input statement is consistent with what was described in Section 9.1. For example, suppose the empty arm is the only arm in the input statement. Then, the process executing the statement will delay forever, unless the input statement contains an else command.

## 9.9 Dynamic Operations

Sections6.1, 7.3, and 9.8. showed how arrays of operations (and semaphores, a specific kind of operation) can be declared and initialized. Each element of an array of operations is just a capability initialized with its own operation object, e.g.,

```
new op void (int)
```

Operations can be created anywhere in a program, not just as part of initialization of an array. The following program illustrates this technique in a variant of the client-server programs from Section 7.3.

```
public class Model5 {
  private static final int N = 20; // number of client processes
  private static op cap double (int) get_worker();

  private static process manager {
    while (true) {
      inni cap double (int) get_worker() {
        cap double (int) ret = new op double (int);
        send worker(ret);
        return ret;
      }
    }
  }
  private static op void worker(cap double (int) request) {
    inni double request(int data) {
      return data*.5;
    }
  }

  private static process client( (int i = 0; i < N; i++) ) {
    ...
    cap double (int) my_worker = get_worker();
    double d;
    d = my_worker(i);
    ...
  }
  public static void main(String [] args) {
  }
}
```

The client invokes the `get_worker` operation to get a capability, `my_worker`, for its own worker. It then invokes its worker and prints out the result of the worker's computation. The `get_worker` operation is serviced by a manager process. It creates a new operation and gets a capability for it, `ret`. The manager then makes `ret` available to both the client and a new worker, which it creates just to handle this request. Specifically, it returns `ret` to the client

and it passes `ret` as a parameter to a new worker process, which it creates via the send statement.

A similar structure can be used to return semaphores to processes, since semaphores are just a specific kind of operation. Also, the manager process can be tailored to fit different applications. For example, the manager might create new processes up to a certain limit; see Exercise 9.21.

## 9.10 Return, Reply, and Forward Statements

Return, reply, and forward statements can be used within input statements. The meaning of return is similar to its meaning within a method (see Section 8.3); the only difference is what happens to the process executing return. The meanings of reply and forward are identical to their meanings within a method. Exercise 9.22 highlights these similarities and differences.

A return statement within an input statement transmits results to the invoker of the operation being serviced—if it was called—and terminates the input statement. The process executing `return` continues execution with the statement following the input statement. (The process executing `return` terminates when the return statement is not within an input statement.)

A reply statement within an operation command of an input statement transmits results to the invoker, if the invocation was called. The replying process continues execution with the statement following the reply. A forward statement defers replying to a called invocation and instead passes on this responsibility to another operation, which may be serviced by a method or by input statements. The forwarding process continues execution with the statement following the forward.

The following three examples illustrates the use of a reply statement within an input statement.

The first example provides a fair solution to the readers/writers problem. Section 9.3 presented a reader's preference solution, which as discussed could starve writers. The idea here is to service start invocations in first-come, first-served order. For example, consider again the scenario from Section 9.3, which involved the following sequence of start requests

```
R1 R2 W1 R3
```

Here, we will service the R1 and R2 requests. However, when W1 arrives, we cannot service it until R1 and R2 have finished. When R3 arrives, we will not service that because W1 has not yet been serviced. (In the earlier algorithm, R3 would be serviced and W1 would be made to wait until there were no readers.) Thus, to ensure fairness, any read requests that arrive before a write request will be serviced, but any read requests that arrive after a write request will not be serviced until after the write request finishes.

Here is the code that implements this fair algorithm:

```
private static process FairRWAllocator() {
  int nr = 0, nw = 0;
  while (true) {
    inni void start_read() {
      reply;
      nr++;
      while (nr > 0) {
        inni void start_read() { nr++; }
        [] void start_write() {
          // the tricky case: don't let this writer
          // proceed until all reads have finished
          while (nr > 0) {
            receive end_read();
            nr--;
          }
          // nr==0, so let writer proceed
          // and then wait for it to finish
          reply;
          receive end_write();
          // since nr==0, enclosing while terminates
        }
        [] void end_read() { nr--; }
      }
    }
    [] void start_write() {
      // easy case: let writer go; then, wait for it.
      reply;
      receive end_write();
    }
  }
}
```

This solution is interesting in that, unlike the previous solution, it uses no synchronization expressions. Its top-level `inni` services a read or write request, which is fine because at that point of execution both `nr` and `nw` are guaranteed to be 0. The arm for `start_write` simply tells the writer to proceed, by using a reply statement, and then waits for the writer to finish, by using a receive statement on `end_write`. The arm for `start_read` is more complicated because, unlike the arm for `start_write`, it needs to allow multiple readers. This code enters a loop, which contains an `inni` that accepts invocations of `start_read`, `start_write`, and `end_read`. It lets any new `start_read` invocations proceed. It also handles `end_read` invocations. The interesting case is when, during this loop, a `start_write` arrives. Such a request corresponds to the example scenario above. So, the code waits for all current readers to finish their accesses to the database and then it lets the writer start and waits for it to finish. Thus, the reply statement is not executed until after the readers have finished. This code takes advantage of the FCFS servicing of invocations. For

example, in the scenario given above, the nested `inni` is guaranteed to service W1's invocation before R3's invocation.

The second use of a reply statement within an input statement provides another example of conversational continuity, which was introduced in the `Conversation` class in Section 8.3. The problem considered here uses a sieve of processes to find prime numbers. The solution is structurally similar to the `PipelineSort` class in Section 8.3. The algorithm is a parallel implementation of what is called the sieve of Eratosthenes. Worker processes pass numbers down a pipeline. Each worker filters out multiples of its prime number and passes the next prime number on to the next worker, which it must create. In this example, however, a given worker does not know in advance how many numbers it will receive. Therefore, each worker also provides a termination operation, which its invoker uses to inform it to terminate.

```
// finds primes from 2 through n
public class Sieve {
  private static op Wrets worker();

  public static void main(String [] args) {
    int n;
    // get maximum number, n, from command line
    n = Integer.parseInt(args[0]);
    // starts up pipeline
    // create first worker instance and feed to it numbers.
    Wrets c = worker();
    for (int i = 2; i <= n; i++) {
      send c.feed(i);
    }
    send c.done();
  }
  // Each worker receives a stream of integers.
  // It keeps the first, discards multiples, and
  // passes the rest onto the next worker, its child.
  private static Wrets worker() {
    op void filter(int);
    op void done();
    int n;
    reply new Wrets(filter, done);
    receive filter(n);
    System.out.println(n);
    Wrets child = null; // child doesn't exist (yet)
    while( true ) {
      inni void filter(int y) {
        if (y % n != 0) {
          if (child == null) { // no child yet; create it
            child = worker();
          }
          send child.feed(y);
```

```
        }
      }
      [] void done() {
        break;
      }
    }
    if (child != null) {
      send child.done();
    }
  }
}

// just a container for two caps
public class Wrets {
  public cap void (int) feed;
  public cap void () done;
  public Wrets( cap void (int) feed,
                cap void () done ) {
    this.feed = feed;  this.done = done;
  }
}
```

Each worker provides two local operations, `filter` and `done`; it returns capabilities for these to its creator via the reply statement. Because each worker needs to return two values, it uses `Wrets` to do so. A worker services invocations of `filter` from its creator until an invocation of `done` is sent, at which point it exits its loop and informs its child, if any, to terminate. The use of the termination message is another example of the structure we saw in Section 9.6.

Another use of a reply statement is to return a value before a break or continue statement executes. For example, suppose that the `done` operation in the example in Section 9.6 returns a value. Then, a reply statement can be used to do so as in the following:

```
private static process server {
  while (true) {
    inni void work() {
      System.out.println("work");
    }
    [] int done() {
      System.out.println("done");
      reply 73;
      break;
    }
  }
  System.out.println("out");
}
```

## Exercises

9.1 Semaphores in JR are provided using abbreviations for operations and for send and receive statements. In turn, receive statements are abbreviations for input statements. Rewrite each program below to use operations and send and input statements.

(a) the critical section program `CS` given in Section 6.1.

(b) the barrier synchronization program `Barrier` given in Section 6.3.

(c) the program developed as a solution to Exercise 6.4(a).

(d) the program developed as a solution to Exercise 6.4(b).

9.2 Consider the nested input statements used for swapping in Section 9.1. Show how processes `p1` and `p2` can exchange values without involving a third process.

9.3 Rewrite the `PipelineSort` class given in Section 8.3 so it uses input statements instead of receive statements.

9.4 (a) Consider the single unit allocator given in Section 9.2, which repeatedly services `request` and `release` operations in that order. Explain why the following is an incorrect solution to the problem:

```
while (true) {
  inni void request() { inni void release() {} }
}
```

(b) Modify the code in part (a) so it keeps the same nested structure but correctly solves the problem.

9.5 Consider again the single unit allocator given in Section 9.2 Explain why the following is an incorrect solution to the problem:

```
while (true) {
  inni void request() {}
  [] void release() {}
}
```

9.6 Show how to rewrite the second input statement in the single slot buffer code (see Section 9.2) to use receive (and send). Also show what an invocation of `fetch` must now look like.

9.7 Rewrite the code in `StreamMerge` (see Section 7.1) so that it does not use an end of stream marker. Instead, have each sending process invoke an additional operation that signifies end of stream.

9.8 An operation may appear in more than one operation command in an input statement. Consider the following input statement:

```
inni void a(int i) st i >= 0 {
  System.out.println("arm 1 "+i);
}
[]   void a(int i) st i <= 0 {
  System.out.println("arm 2 "+i);
}
```

What will be printed if `a(-1)` is invoked? What if `a(0)` is invoked? What if `a(1)` is invoked? Explain your answers.

9.9 Let `S1` and `S2` denote statement lists, and let `B1` and `B2` denote boolean expressions. Are the following two input statements equivalent?

```
inni void f(int x) st B1 {
  S1
}
[] void f(int x) st B2 {
  S2
}

inni void f(int x) {
  if (B1) {
    S1
  }
  else if (B2) {
    S2
  }
}
```

Give a convincing argument why they are, or a specific example demonstrating why they are not.

9.10 Consider the multiple unit allocator given in Section 9.3. Initially there are `M` units of the resource, as shown. Suppose there are `W` worker processes that call `request` and `release`. The units could represent a bag of tasks for the workers to process. In this case, a worker calls `request` to get a task from the bag and calls `release` to put a new task in the bag. Also, the allocator is an administrator process that manages the bag of tasks.

Suppose the workers and allocator are the only processes and that each worker has the following code outline:

```
while (true) {
```

```
    request();
    release(); // now called 0 or more times per request
  }
```

Assume that eventually `request` is called `M` more times than `release`, i.e., that eventually the bag is empty and all tasks have been processed.

As programmed, the processes will deadlock. Modify the code in the multiple unit allocator so that it terminates the computation by invoking `JRexit`. Hint: Service an operation in more than one operation command.

9.11 Modify the `BoundedBuffer` example (see Section 9.3) so that it also provides `current_size` and `query` operations. The former returns the current number of elements in the buffer; the latter returns a boolean indicating whether a specified number is an element of the buffer.

Develop a complete test program as well. It must contain multiple user processes and use `Thread.sleep` to make the output more interesting.

9.12 Program a stack class that provides synchronized push and pop operations in the same spirit as the `BoundedBuffer` example (see Section 9.3).

9.13 The readers/writers scheduling process `RWAllocator` (see Section 9.3) gives preference to readers.

(a) Modify the process to give preference to writers; i.e., if a writer wants to start writing, it gets to do so before a reader gets to start reading. Hint: Use the `length()` method.

(b) Modify the process to give further preference to readers. The solution in Section 9.3 already gives readers preference to some extent, but not in the following scenario. Suppose writer1 is writing the database, and writer2 arrives, and then reader1 arrives. In the current solution, when writer1 finishes writing the database, writer2 will be allowed to go, because JR services the oldest invocations first when possible. Modify the given solution so that in scenarios like this one, reader1 instead should be allowed to go. Hint: Use the `length()` method.

(c) Modify the process so scheduling is fair; i.e., any reader or writer that wants to access the database is able to do so eventually, assuming every reading or writing process eventually calls `end_read` or `end_write`. Use the approach of allowing at most $X$ readers to proceed when a writer is waiting before switching to writing, and vice versa. (Do not use the approach given in Section 9.10!)

For each of the above, develop a complete test program as well. It must contain multiple reader processes and multiple writer processes; it must use `Thread.sleep` to make the output more interesting. The program's output must demonstrate that the program handles readers and writers in the correct order.

9.14 The readers/writers scheduling process `RWAllocator` (see Section 9.3) uses a four-arm input statement and synchronization expressions. Show how to convert that input statement into code that has the same effect but that does not use synchronization expressions. Hint: Use nested input statements plus additional statements.

9.15 *Atomic Broadcast Problem.* Solve Exercise 7.10, but change the interface so that producers and consumers use call invocations and the coordinator uses one (or more) input statements possibly using synchronization expressions.

9.16 *Savings Account Problem.* Solve Exercise 7.11, but change the interface between customers and the bank so that customers making a withdrawal use a call invocation and customers making a deposit use a send invocation. Also, change the bank (server) so that it uses an input statement with a synchronization expression to defer withdrawals when necessary.

9.17 *One-Lane Bridge Problem.* Cars coming from the north and the south arrive at a one-lane bridge. Cars heading in the same direction can cross the bridge at the same time, but cars heading in opposite directions cannot. Assume that a maximum of $N$ cars are in the system at any time, that each car has some unique identifier, and that the capacity of the bridge is unbounded (although one would hope no more than $N$ cars are on the bridge at any time!).

(a) Write a deadlock-free program that solves this problem.

(b) Modify your program so that it also ensures fairness. (Hint: allow at most $C$ cars to cross in one direction if cars are waiting to cross in the other direction.)

Have the programs give meaningful output. For example, at each interesting "event" (e.g., each time a car gets on or off the bridge, etc.), print out the "state" of the entire system, i.e., what each car is doing.

9.18 *Bus Problem.* Two kinds of processes–busses and passengers—arrive at a bus stop. A passenger cannot leave the bus stop until it has boarded a bus and its bus leaves. Each bus holds $N > 0$ passengers. A bus cannot leave the bus stop until it has filled all its seats.

The following hints should lead to a fairly straightforward solution. A passenger waits until its bus leaves. A bus waits until there are at least $N$ passengers; it then, in effect, boards $N$ passengers at once and leaves. The key events, then, are when passengers arrive, when passengers leave, and when busses leave. Boarding is really only used for descriptive purposes; it does not require additional synchronization. Note that a passenger does not really leave "on" a bus; processes, after synchronizing as described above, continue execution independently. Your program can use an extra, manager process to coordinate the activities of the busses and passengers.

Have the programs give meaningful output. For example, at each interesting "event" (e.g., each time a bus arrives at a bus stop, a passenger arrives, etc.), print out the "state" of the entire system, i.e., what each bus or passenger is doing.

9.19 (a) Modify your solution to the previous question so that a bus leaves the bus stop after boarding *at most N* passengers. Specifically, a bus will board as many of those passengers waiting that it can, up to $N$, when it arrives. If no passengers are waiting, the bus departs empty.

(b) Modify your solution to the previous part so that a bus waits for one passenger to arrive and board if it arrives at a bus stop and finds no passengers waiting.

9.20 Consider again the input statement that uses a quantifier given in Section 9.8. First, show how that code can be rewritten without using a quantifier. Now suppose that `signal` is declared with an upper bound of `N`, which is read from input, and that the input statement is to service any element of the `signal` array. Does your rewriting technique used above generalize to this case? Explain why or why not.

9.21 Modify the `Model5` program in Section 9.9 so that at most $W$ worker processes can co-exist. (Use, say, $W = 6$ when running your program.) Thus, when $W$ worker processes do exist, a request for a new worker will need to be deferred until one of the worker processes has finished.

9.22 For the parts below, recall that Figure 2.1 depicts the execution of a process executing a basic rendezvous.

(a) Give a figure similar to Figure 8.1 (and give the corresponding code outline) for when `f` is serviced by an input statement.

(b) Give a figure similar to Figure 8.2 (and give the corresponding code outline) for when `f` is serviced by an input statement.

(c) Give a figure similar to Figure 8.3 (and give the corresponding code outline) for when `f` and `g` are serviced by input statements.

(d) Repeat part (c) for when `f` is serviced by a method and `g` is serviced by an input statement.

(e) Repeat part (c) for when `f` is serviced by an input statement and `g` is serviced by a method.

9.23 Consider the output of the code in the prime sieve algorithm (see Section 9.10).

(a) Is it printed in order or might it be interleaved? Explain.

(b) Suppose the main process wants to output "`Done`" *after* all other processes have output their numbers. Explain why adding just a print statement to the end of the main process is not guaranteed to work. Show how to modify the code to get the desired effect.

9.24 Consider again the prime sieve algorithm (see Section 9.10). First modify the code so that each worker is passed, when it is created, its position within the pipeline (i.e., 1, 2, ...). Then, modify the code so that after the pipeline is set up, it can be searched multiple times; each search request specifies a single number for which to search. Use a separate local operation, `search`, within each worker to service a single request for searching. Also use a separate local operation, `search_done`, within each worker; it is called after all searches to terminate the worker. The output from a search indicates whether the number was found in the pipeline and the position of the worker that made that determination.

Searching is to be done following the pipeline. That is, each worker tests to see if it holds the given number. If it does, it returns that the number was found and this worker's position. If not, the worker determines whether it can terminate the search early (hint: the pipeline is sorted). If so, it returns failure and this worker's position; if not, it passes the request on to the next worker in the pipeline (with the last worker doing something special); Each worker simply compares the values of its prime number and the given search number — it does not test whether the given number is a multiple of its prime number; e.g., when searching for 9, the worker holding 3 will pass the request on to the next worker.

9.25 Repeat the previous exercise for the pipeline sort algorithm introduced in Section 8.3.

9.26 Consider again the prime sieve algorithm (see Section 9.10). Modify the code so that an explicit done message is not needed. Instead pass a

special number (e.g., zero) to `filter`. Which technique is more general and cleaner?

9.27 Consider the following operation declarations and input statement:

```
op void a(double);  op void b(char);
inni void a(double d) {
  ...
}
[] void b(char c) {
  ...
}
```

Assume that these operations are invoked only by call. Show how the input statement can be replaced by a receive statement. Also show how `a` and `b` must now be invoked.

9.28 Consider the following input statement from the prime sieve algorithm (see Section 9.10):

```
inni void filter(int y) {
  ...
}
[] void done() {
  break;
}
```

Within its containing loop, it services all invocations of `filter` and then an invocation of `done`. The reason for that ordering of invocation servicing is that the semantics of JR's input statement picks which operation to service based on the arrival times of the invocations. Suppose that the semantics of JR's input statement were that, instead, the operation for which to service an invocation is picked in a non-deterministic order. Show how the above input statement would need to be written so that all invocations are handled in the same order as they are now.

9.29 A *binary search tree.* Write a JR program that reads in a list of numbers, builds a binary search tree from processes, and then responds to print (in-order) and search commands. Your solution is to be similar in spirit to the pipeline sort program (see Section 8.3) and the prime sieve algorithm (see Section 9.10).

Specifically, use one process for each node in the tree, which holds one number from the input. You do not know in advance how many node processes are needed, which means that they must be created on demand.

All processes must be created within a single object. Use local operations, capabilities, and the `reply` statement; do *not* use an array of operations. Your program should terminate normally, not in deadlock. For a search, output whether the number exists in the tree or, if it was not found, the number in the node that determined it was not in the tree.

Source files containing parts of this program come with the JR distribution. Run your program on each of the supplied data files.

9.30 The following sort method uses an operation to sort an array:

```
int [] sort(int [] a) {
  op void list(int);
  for (int i = 0; i < a.length; i++) { send list(a[i]); }
  for (int i = 0; i < a.length; i++) {
    inni void list(int x) by x { a[i] = x; }
  }
  return a;
}
```

Explain how it works. Its running time appears to be linear, i.e., order `n`, where `n` is the size of `a`. Of course, a general linear-time sorting algorithm is not possible. Explain the discrepancy.

9.31 In extended forms of CSP [26] (also see Chapter 21), guards in if and do statements can contain both input and output commands; thus a process can be waiting either to receive input or to send output. (CSP's do statement is similar to Java's while statement, but the do statement, like an if statement, allows multiple arms, each containing a guard and statement list.) JR does not allow invocations to appear in guards of input statements; on the other hand, `send` is non-blocking. Ada's `select` statement allows either invocation statements (calls) or `accept` statements (input), but not both; Ada does not have an asynchronous invocation statement.

Discuss the tradeoffs between these three approaches. Are there differences in expressive power? In implementation cost?

9.32 *Set partition* [36]. Process *A* has a set of integers, *S*. Process *B* has a set of integers, *B*. The processes are to exchange values one at a time until all elements of *S* are less than all elements of *T*. Note that after any exchange, *S* has the same number of elements in it as it did at the beginning; the same applies to *T*.

Assume that *S* is not empty and that *S* and *T* are disjoint. Do not use shared variables or additional processes.

Source files containing parts of this program come with the JR distribution. Run your program on each of the supplied data files.

9.33 *Dutch National Flag.* A collection of colored balls is distributed among $n$ processes. There are at most $n$ different colors of balls. The goal is for the processes to exchange balls so that eventually, for all $i$, process $i$ holds all balls of color $i$. Assume process identities and colors are integers between 0 and $n-1$.

The number of balls in the collection is unknown to the processes. A process might start holding no balls if that is how the balls were initially distributed. Process $i$ will finish holding no balls if no balls of color $i$ appear in the collection.

Write code for the processes. Assume interprocess communication forms a ring: Process $i$ is allowed to give a ball only to process $i+1$ (with wrap-around from process $n-1$ to process 0). Processes are allowed to pass only messages that contain a single ball or control information (but not counts of the number of balls). The processes should terminate normally, not in deadlock.

Process $i$ can access only its source and target bags, not those of other processes. Do not use shared variables or additional processes. The solution must be symmetric, i.e., do not have processes execute special case code based on their process indices. (Of course, a process will use its process index in determining who to send to and receive from, and to determine whether a ball belongs with the process.)

Source files containing parts of this program come with the JR distribution. Run your program on each of the supplied data files.

9.34 Give the output from the following program and explain how it works.

```
public class Play {
  private static op void f(int);
  private static op void g();
  private static process p1 {
    send f(10);
    send g();
    send f(4);
    send f(3);
    send f(8);
    while (f.length() + g.length() > 0) {
      inni void f(int x) by x {
        System.out.println("got f "+x);
      }
      [] void g() {
         System.out.println("got g");
      }
    }
    System.out.println("done");
```

```
  }
  public static void main(String [] args) {
  }
}
```

9.35 Give the output from the following program and explain how it works.

```
public class Main {
  public static void main(String [] args) {}
  static op Object f(int);
  static op Object g(int);
  static cap Object (int) c;
  static process go {
    c = f;
    c = (cap Object (int)) c(3);
    c = (cap Object (int)) c(12);
    c = (cap Object (int)) c(19);
    c = (cap Object (int)) c(28);
  }
  static process gogo{
    while(true) {
      inni Object f(int z) {
        System.out.println("ffff "+z);
        return (cap Object (int)) g;
      }
      [] Object g(int z) {
        System.out.println("gggg "+z);
        return (cap Object (int)) f;
      }
    }
  }
}
```

*This page intentionally left blank*

Chapter 10

# VIRTUAL MACHINES

So far we have implicitly assumed that programs execute within a single address space on a single physical machine. This chapter describes the JR mechanisms that allow programs to contain multiple address spaces, which can execute on multiple physical machines. These additional mechanisms thus support truly distributed programming.

The JR model of computation allows a program to be split into one or more address spaces called *virtual machines.* Each virtual machine defines an address space on one physical machine. (A JR virtual machine includes a Java virtual machine and an additional layer that supports the JR concurrency extensions.) Virtual machines are created dynamically, in a way similar to the way objects are created. When a virtual machine is created, it can be placed on a specific physical machine.

How a virtual machine is used is reflected in how its objects are created. An object is created using a slight variant of the usual `new` operator; an optional clause specifies the virtual machine on which the new instance is to be located. Processes, variables, and operations in an instance exist entirely within a single virtual machine. As in Java, the static parts of a class are created automatically as needed; however, a separate instance of the static parts is created on *each* virtual machine that needs them.

Communication between virtual machines is transparent. For example, a send invocation from one virtual machine to an operation serviced within an object located on a different virtual machine has the same syntax and nearly identical semantics as a send invocation to an operation serviced on the same virtual machine. The same applies to the syntax and semantics of call invocations between different virtual machines.

This chapter describes the JR mechanisms for creating virtual machines, including how to place them on different physical machines. It also describes

how remote objects are created on virtual machines. Finally, we discuss several practical issues that arise in programs that employ multiple virtual machines. As usual, we present several example programs that employ multiple virtual and physical machines. Additional, more realistic examples appear in Chapter 11 and Part II. Some of the rules for the mechanisms described in this chapter are motivated by implementation concerns (see Appendix D).

## 10.1 Program Start-Up and Execution Overview

Recall that execution of Java programs begins in the `main` method in the designated "main" class. However, before that code actually executes, some static initializers may execute: those in the "main" class and those in other classes on which those in the "main" class depend. For example, if a static initializer in the "main" class invokes a method in another class, then the static initializers in the other class are executed before the method call.

Execution of JR programs is similar, but extends to programs with multiple virtual machines. JR program execution begins with the implicit creation of a "main" virtual machine, on which the "main" class begins execution. As noted earlier, execution begins by executing any needed static initializers (such as those associated with implicit process creation, as described in Section 4.2); after that, execution continues in the main method. The main virtual machine executes on the physical machine from which program execution was initiated.

When a virtual machine is created, it is, by default, essentially empty: It contains no objects. (See Appendix D for details.) The main virtual machine is empty when it is first created; however, it is special in that the static parts of the "main" class are immediately and automatically created on it, as described above. Code in the main method or in other methods, objects, or classes invoked or created directly or indirectly by the main method, can create additional virtual machines, which can be located on other physical machines. Objects can then be created on those virtual machines. Consider a particular class `C` and what happens the first time a `C` object is created or a static method in `C` is invoked on a particular virtual machine. Any static initializers in `C` and any static initializers in other classes on which `C`'s static initializers depend are executed. A given virtual machine might extend the default virtual machine class, so it too can contain variables, methods, and objects; see Section 10.6 for details.

As noted earlier, each virtual machine contains its own instance of static parts of classes. Static variables are local to that virtual machine. Variables in objects created on a given virtual machine are local to the scope in which they are created, so they too are accessible only within (a limited part of) a single virtual machine. Access to operations follows similar rules, except operations can be shared across virtual machines by using capabilities (see Section 7.7). Recall that capabilities can be used to specify the operation in a receive statement (Chapter 7) or in an input statement (Chapter 9). The specified operation can

be located on a different virtual machine from the servicing statement. However, such servicing will generally incur additional execution cost. A typical implementation will likely keep the invocations for an operation on the virtual machine containing the operation. Access to the invocations from a different virtual machine will require messages to be exchanged. See Appendix D for details. The examples in Section 10.4 illustrate the sharing of variables and operations.

Termination of a program with multiple virtual machines is similar to that for a program with only a single virtual machine. As before, a program terminates when all processes have terminated, a deadlock has occurred, or `JR.exit` is executed. Note that the effect of `JR.exit` is not instantaneous; i.e., all parts of a distributed program do not halt immediately. As for a program with only a single virtual machine, a program with multiple virtual machines can use a quiescence operation. Similar to what was discussed in Section 4.5, only the operation most recently registered is invoked. Thus, the quiescence operation is global across all virtual machines.

## 10.2 Creating Virtual Machines

A virtual machine is created by creating an instance of the `vm` pseudo-class, which is a special, predefined class. In this case, the value returned by new is a reference of type `vm`. For example, consider

```
vm c;
c = new vm();
```

This code fragment creates a new virtual machine and assigns a reference for that virtual machine to variable `c`.

By default, a new virtual machine is placed on the same physical machine as its creator. A new virtual machine instance can be placed on a specific physical machine (network node) by using the following form of creation:

```
new vm() on expr
```

The expression specifies the name of a physical machine as a string or specifies a virtual machine reference. When the expression is a string, it is the name of a physical machine (which is, of course, installation dependent) on which to create the new virtual machine. When *expr* is a virtual machine reference, it indicates that the new virtual machine is to be located on the same physical machine as the specified virtual machine.

As an example, suppose the following code fragment is executed on a physical machine named `magic`:

```
vm c1, c2, c3, c4;
c1 = new vm();
c2 = new vm() on "camelot";
```

```
  c3 = new vm() on "excalibur";
  c4 = new vm() on "localhost";
```

It creates three virtual machines and assigns references for them to `c1`, `c2`, and `c3`. The first virtual machine is created on `magic` since no explicit physical machine was specified. The second is created on a physical machine named `camelot` and the third on `excalibur`. The fourth virtual machine is created on `magic`; the name `localhost` is defined by the operating system as an alias for the current machine.

The use of `localhost` is convenient as a system-independent way to specify the present host name. For example, it is common to specify on the command line the names of the physical machines on which to create virtual machines. But, as a default, if no names are specified, then the program should create a single virtual machine on the present host. The following code shows how to do that.

```
public class Main {
  public static void main(String [] args) {
    String [] machs;
    if (args.length > 0) {
      machs = args;
    }
    else {
      // default: one machine, the local host.
      machs = new String [1];
      machs[0] = "localhost";
    }
    vm [] vmachs = new vm [machs.length];
    for (int i = 0; i < machs.length; i++) {
      vmachs[i] = new vm() on machs[i];
    }
  }
}
```

Suppose this program is run on machine `magic`. If the command-line arguments are `camelot`, `excalibur`, and `camelot`, then the program creates two virtual machines on `camelot` and one on `excalibur`. If no command-line arguments are given, then the program creates a single virtual machine on `magic`. Notice how the loop handles creation in either case. Of course, this example only creates virtual machines, but does not "populate" them with any objects. However, this pattern is simpler than the alternative (see Exercise 10.1) and is quite useful; e.g., it is used in Section 20.2.1.

JR has no explicit statement to destroy a virtual machine, which is consistent with Java's implicit object destruction. Conceptually, a virtual machine will become "garbage" and can be garbage collected when the rest of the program has no references to it and the virtual machine has become idle. Becoming

idle means that all processes are blocked waiting for messages, i.e., no process is executing, waiting for input/output to complete, or waiting to be awakened from sleeping. In our current JR implementation, however, the JR execution manager always holds a reference to each virtual machines created, so they are not garbage collected.

## 10.3 Creating Remote Objects

Objects in standard Java programs are created using `new` and objects in JR can also be created in the same way. However, objects in JR programs that are to be placed on virtual machines must be declared and created slightly differently. Specifically, such an object must be specified as being `remote` both when it is declared and created, as indicated in the following code example

```
remote Foo f;
f = new remote Foo(11);
```

In addition, any class from which remote objects are instantiated (e.g., `Foo` above) must be declared as `public`.

By default, a remote object is created on the same virtual machine as its creator. The following form of remote object creation instantiates the specified object on the specified (existing) virtual machine:

**new remote** *class_name* **(** *arguments* **) on** *expr*

The value of the expression is a reference for the virtual machine on which the object is to be created.

For example, consider the following, which uses `c1` and `c2` from above:

```
new remote Foo(34) on c1;
new remote Foo(22) on c2;
new remote Foo(70);
```

This code fragment creates three `Foo` objects. The first is created on virtual machine `c1`, the second on `c2`, and the third on the same virtual machine as the creator.

The operations in a remote object are accessed via a remote reference to the object. Continuing the above example, where `f` is a remote reference for a `Foo` object, suppose that class `foo` declares operation `g`, then `f.g()` invokes operation `g` in the `f` remote object. A remote object reference is, in effect, a collection of individual capabilities for the remote object's operations. Thus, the individual fields of a remote reference can be manipulated independently. As seen earlier, a field can be used to invoke an operation. A field can also be assigned to with a different capability; see Exercise 10.3.

Remote object references can be assigned two special values: `null` and `noop`. These values have meanings similar to their use with capability variables

(Section 3.3). The effect is to set each of the remote reference's individual capabilities to the particular special value. A remote interface reference cannot be assigned `noop` unless the reference is cast to a valid remote implementing type for that interface. Otherwise, the type of noop could not be determined. See Exercise 10.4 for examples.

Given that a remote reference is really a collection of individual operation capabilities, non-static variables and methods are *not* accessible via a remote reference to the object. If such access to a non-static method is desired, the method must instead be made an operation. Static variables and static methods in a class from which remote objects are instantiated can be accessed, as usual, via the name of the class.[1] Such a reference refers to a static member within the local virtual machine (similar to the `Main2` example in the next section). From one virtual machine, the static members in another virtual machine cannot be accessed directly.

One notable difference between remote objects and regular objects is that a remote object is *not* garbage collected when the JR code no longer holds any references for it. The reason is that our current JR implementation maintains additional, internal references for remote objects.

## 10.4 Examples of Multiple Machine Programs

The following examples present programs composed from a class containing only static variables and a semaphore, a class that uses those static fields, and a class that contains the main method. Only the "main" classes differ between the examples.

The first class is as follows:

```
public class Glob {
  public static int x = 0;
  public static sem mutex = 1; // for exclusive access to x
}
```

It provides a shared variable, `x`, plus a semaphore, `mutex` that can be used to protect accesses to it.

The second class is:

```
public class Foo {
  int N, n;
  cap void () c;
  public Foo(int N, int n, cap void() c) {
    this.N = N; this.n = n; this.c = c;
  }
```

[1]The current JR implementation does not allow a static member to be accessed via a remote object reference, as Java allows for its object references.

```
  private process p( (int i = 1; i <= N; i++) ) {
    P(Glob.mutex); Glob.x += n; V(Glob.mutex);
    send c();
  }
  public op void writex() {
    System.out.println(n + " sees " + Glob.x);
  }
}
```

Each of the `N` instances of process `p` adds `n` to the shared variable `x`; the update is protected by using `mutex`, the semaphore (shared operation) declared in `Glob`. Each process then sends a message to the operation pointed to by capability `c`, which was passed to `Foo`'s constructor.

As a first example, consider the following main class:

```
public class Main1 {
  public static void main(String [] args) {
    final int N = 5;
    op void done();
    remote Foo foo1, foo2;
    foo1 = new remote Foo(N, 1, done);
    foo2 = new remote Foo(N, 2, done);
    for(int i = 1; i <= 2*N ; i++) {
      receive done();
    }
    foo1.writex();
    foo2.writex();
  }
}
```

It creates two `Foo` objects and then gathers `done` messages from every process `p` in each instance. It then calls `writex` in the two `Foo` objects. At this point the program terminates.

The above program executes on a single virtual machine. Therefore, only one instance of `Glob` is created. The invocations of `writex` in both `Foo` objects refer to the same value, `15`, which is output twice. (The five processes in `foo1` each add 1 to `x`; the five in `foo2` each add 2 to `x`.)

As a second example, consider the following main class:

```
public class Main2 {
  public static void main(String [] args) {
    final int N = 5;
    op void done();
    remote Foo foo1, foo2;
    foo1 = new remote Foo(N, 1, done);
    vm vmref = new vm();
    foo2 = new remote Foo(N, 2, done) on vmref;
    for(int i = 1; i <= 2*N ; i++) {
```

```
      receive done();
    }
    foo1.writex();
    foo2.writex();
  }
}
```

The code here differs from `Main1` in that it creates a second virtual machine on which it places the second instance of `Foo`. Those two lines of code can be written more compactly as:

```
foo2 = new remote Foo(N, 2, done) on new vm();
```

Since `Main2` executes on two virtual machines, an instance of `Glob` is created on each. The effect is to create separate instances of `x` and mutex on each virtual machine. The program outputs first the number 5—for the first `writex` (on the main virtual machine)—and then the number 10—for the second `writex` (on the second virtual machine).

This program executes on a single physical machine. If desired, the second virtual machine can be placed on a different physical machine by changing the statement that creates it as follows:

```
vm vmref = new vm() on "camelot";
```

However, the program's output remains the same as above. As before, the above line of code and the line that creates `foo2` can be combined as:

```
foo2 = new remote Foo(N, 2, done) on new vm() on "camelot";
```

Virtual machines can be distributed easily over a group of physical machines. Section 10.2 showed a basic example and Section 11.1 contains a complete example in the context of a more realistic program.

In the examples, parameter `c` of `Foo` is a capability for the operation done declared in the main class. It is worth noting that invocations of `c` might cross virtual and physical machine boundaries. In `Main2`, for example, the invocations of `c` from the second instance of `Foo` go from the explicitly created virtual machine (pointed at by `vmref`) to the original one. Those invocations also go from one physical machine to another if the second virtual machine is on a different physical machine, which can be accomplished as shown above. No change to the send or receive statements is required for such intermachine invocations.

## 10.5 Predefined Fields

Just as it is sometimes convenient to have an object reference for the current object (i.e., `this`), it is also sometimes convenient to have a remote object

reference for the current object. The latter can be obtained by applying the `remote` field to `this`, i.e., `this.remote`. The following (artificial) program demonstrates; a more realistic example appears in Section 17.3. The main method creates a remote instance of `A` and invokes the `h` operation therein.

```
public class Main {
  public static void main(String [] args) {
    remote A a = new remote A();
    System.out.println(a.h(2));
  }
}
```

`A`'s constructor creates a remote instance of `B`, to which it passes as a constructor parameter a remote object reference for itself.

```
public class A{
  remote B b;
  public A () {
    b = new remote B(this.remote);
  }
  public op double f(double x) {
    return x*x;
  }
  public op int g(int x) {
    return x*100;
  }
  public op int h(int x) {
    return b.e(20*x);
  }
}
```

`B` is then able to use any of the public operations in `A` through its remote reference `a`, which it does within the body of `e`'s op-method.

```
public class B{
  remote A a;
  public B (remote A a) {
    this.a = a;
  }
  public op int e(int x) {
    System.out.println(a.f(12));
    System.out.println(a.g(14));
    return x*3;
  }
}
```

Note that the same effect could be achieved by instead passing each of `A`'s operations as parameters to `B`'s constructor, but that will generally be more

verbose. The `remote` field can also be applied to any object reference, not just `this`.

One other predefined field deals with virtual machines. The field `vm.thisvm` returns a reference for the virtual machine on which it is executed. The following program demonstrates the use of this field. The main program creates `N` instances of `Foo` on separate virtual machines.

```
public class thisvmDemo {
  public static void main(String [] args) {
    final int N = 2;
    remote Foo foo [] = new remote Foo[N];
    vm whichvm [] = new vm [N];
    for (int i = 0; i < N ; i++) {
      foo[i] = new remote Foo(100+i) on new vm();
    }
    for (int i = 0; i < N ; i++) {
      whichvm[i] = foo[i].go();
    }
    for (int i = 0; i < N ; i++) {
      new remote Goo(i) on whichvm[i];
    }
  }
}
```

Each instance of `Foo` uses a static variable, `x`, defined in `Glob`, so a separate instance of `Glob` is created on each virtual machine.

```
public class Glob {
  public static int x = 0;
}
```

Each instance of `Foo` returns via its `go` operation a reference for its virtual machine.

```
public class Foo {
  int n;
  public Foo(int n) {
    this.n = n;
  }
  public op vm go() {
    Glob.x = n;
    System.out.println("Foo "+n+" went");
    return vm.thisvm;
  }
}
```

The main program uses those virtual machine references to create instances of `Goo` on the separate virtual machines.

```
public class Goo {
  public Goo(int i) {
    System.out.println("Goo "+i+" sees "+Glob.x);
  }
}
```

Goo's constructor simply prints out its i and Glob.x. The exact output of the program is left as an exercise (see Exercise 10.6).

## 10.6 Parameterized Virtual Machines

The examples in the previous sections illustrate one common set up activity when using multiple virtual machines: Each virtual machine is given its own, application-defined identifier. For example, in the Main2 program in Section 10.4, class Foo uses n, in effect, to identify the virtual machine on which it resides; in the thisvmDemo program in Section 10.5, class Goo uses i similarly. If several classes each want access to the virtual machine identifier, then each class needs the identifier passed to its constructor, which is somewhat cumbersome.

A nicer solution is to use *parameterized virtual machines.* A parameterized virtual machine is a class that extends the predefined vm class. The extended class can then define operations accessible to all objects executing within a virtual machine. These operations are accessible indirectly via the vm.thisvm reference. (This access via operations is similar to the kind of access provided by remote objects.)

As an example, here is how to rewrite a slight variant of the Main2 program in Section 10.4 to use parameterized virtual machines. The slight difference is that each instance of Foo is created on its own virtual machine. (The reason for this slight difference is discussed at the end of this section.)

The major change in the code, of course, is the new class Myvm. It stores a virtual machine identifier (an integer, although any type can be used) and provides an access operation for it.

```
public class Myvm extends vm {
  private int id;
  public Myvm(int id) {
    this.id = id;
  }
  public op int GetID() {
    return id;
  }
}
```

The main class changes only in the statements that create instances of Foo on instances of the virtual machine Myvm. Notice how the virtual machine number is now passed to Myvm's constructor; previously it was passed to Foo's constructor.

```
  foo1 = new remote Foo(N, done) on new Myvm(1);
  foo2 = new remote Foo(N, done) on new Myvm(2);
```

The code for class `Foo` now needs to access the identifier from `Myvm`. To do so, it downcasts `vm.thisvm` to `Myvm` and invokes the `GetID` operation in its instance of `Myvm`. (This downcast would result in a run-time exception if the current virtual machine were not a subclass of `Myvm`.)

```
public class Foo {
  int N, n;
  cap void () c;
  public Foo(int N, cap void() c) {
    this.N = N; this.c = c;
  }
  private process p( (int i = 1; i <= N; i++) ) {
    P(Glob.mutex);
    Glob.x += ((Myvm)(vm.thisvm)).GetID();
    V(Glob.mutex);
    send c();
  }
  public op void writex() {
    System.out.println(((Myvm)(vm.thisvm)).GetID() +
                       " sees " + Glob.x);
  }
}
```

Class `Glob` remains unchanged.

A parameterized virtual machine class (i.e., one that extends the predefined `vm` class) can also declare static members, which are accessible via the class name. Such a static variable can be used to achieve the effect of the virtual machine identifier seen in the above example. In this case, the resultant code is a bit simpler than the above code (see Exercise 10.8), but in general the above code is more flexible.

Consider rewriting the *original* `Main2` program in Section 10.4 so it uses parameterized virtual machines, say `Myvm` above. The difficulty is that the `foo2` would be placed on a virtual machine of type `Myvm`, whereas `fool` would be placed on the main virtual machine, which is of type `vm`. If `fool` attempts to access the `GetID` operation, its downcast of `vm.thisvm` to `Myvm` would result in a run-time exception. Moreover, JR does not provide a way to change the virtual machine after it is created. Here, for example, that would be useful to allow the main virtual machine to become an instance of `Myvm`.

A more realistic example of the use of parameterized virtual machines appears in Section 18.3.

## 10.7 Parameter Passing Details

As noted in Section 7.8, parameter passing in JR invocations on the same virtual machine is "by value". Consider a variant of the program in Section 7.8.

```
public class SameVMArraySend {
  public static void main(String [] args) {
    remote R rr = new remote R();
    int [] b = new int [2];
    b[0] = 11; b[1] = 34;
    send rr.f(b);
    b[0] = 65; b[1] = 87;
    send rr.f(b);
  }
}

public class R {
  public op void f(int []);
  process p {
    for (int k = 1; k <= 2; k++) {
      inni void f(int [] a) {
        for (int i = 0; i < 2; i++) {
          System.out.println(k +" a["+i+"] "+a[i]);
        }
      }
    }
  }
}
```

The difference is that now a separate process services invocations of `f`. The program's output is non-deterministic depending on the order in which processes execute. For example, suppose process `p` executes its input statement before the assignments after the first send statement in the main method. In this case, the output will be 11, 34, 65, and 87. In other cases, the output can be different.

Parameter passing across virtual machines is slightly different. Passing just an object reference would not work since the object being referenced is in a different address space and so would not be directly accessible. Instead, a copy of the object is passed too. More precisely, the JR implementation uses RMI. RMI "serializes" objects that it passes between different virtual machines.

Consider again the above program. Suppose we modify it so that the `R` object is created on a different virtual machine, i.e.,

```
remote R rr = new remote R() on new vm();
```

Then the program's output will *always* be 11, 34, 65, and 87. The reason is that, in effect, each invocation of `f` now contains a copy of the `b` object at the time the send statement is executed.

One other related effect is that parameters (and return values) for invocations between virtual machines must be serializable. In many cases, that

will cause no difficulty since a parameter's type will already be serializable. In some cases, the programmer might need to specify that a class `implements java.io.Serializable`. In a few cases, the programmer might need to write code to make a class serializable (or change the interface). Chapters 18 and 20 present examples and exercises where a class must be specified as being serializable and further discusses this topic.

Another issue related to serializability involves comparisons of object references. See Exercise 10.12.

## 10.8 Other Aspects of Virtual Machines

This section describes several aspects of virtual machines that arise in practice. Some of these aspects, as will be noted, are implementation dependent. Moreover, some may cause different behaviors in multiple virtual machine programs than in single virtual machine programs.

The initially created virtual machine inherits the standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`) streams from the command that starts execution of an JR program. Other virtual machines created by the program inherit these streams from the initial virtual machine. If several machines print to these shared `stdout` or `stderr` streams, the ordering of output is deterministic only if the prints are properly synchronized within the JR program. Having multiple virtual machines read from `stdin` at about the same time is usually not useful: they will compete for individual characters from `stdin` and obtain them non-deterministically.

Except for this initial duplication of the standard streams, input/output is virtual machine specific. In particular, Java file-related objects (e.g., such as `FileReader` or `PrintWriter` objects) that have been created on one virtual machine cannot be passed to other virtual machines. Attempting to do so will result in a "not serializable" exception.

Command-line arguments are passed as arguments to the main method. As in Java, if their values are needed elsewhere, they need to be made available, e.g., via passing them as parameters. The same holds in JR programs. In particular, command-line arguments are not available on other virtual machines unless they are passed to it, e.g., as parameters to a constructor or method on the other machine.

The main virtual machine begins execution in the directory in which its execution is initiated. However, in our implementation, other virtual machines begin execution in the user's home directory. This difference can have an effect on programs that open files. If a filename is absolute, then there is no problem. However, if a filename is relative to the current directory and code on a non-main virtual machine attempts to open that file, then it will not find it. (Or, worse, it will open the wrong file: the one with the same name relative to the home directory!) Beside using absolute filenames, the above problem can be

avoided by having code prepend the pathname of the current directory to all the names of any files that it opens. The current directory can be obtained via:

```
System.getProperty("user.dir");  // returns a String
```

The above discussion applies to executing virtual machines on the same physical machine as the main virtual machine or on a different physical machine but with a common NFS (network file system) environment, in which files and their names are identical across a collection of machines. For non-NFS environments, the user might need to account for differences in file systems in other ways, e.g., by mapping file names on one system to those on the other.

## Exercises

10.1 Modify Main (see Section 10.2) so that it does not use `localhost`. Do *not* modify `Main`'s creation loop, only modify the code above it. Hint: What does the following code output?

```
import java.net.*;
public class Main {
  public static void main(String [] args) {
    try {
      InetAddress addr = InetAddress.getLocalHost();
      System.out.println(addr.getHostName());
    } catch (Exception e) {e.printStackTrace();}
  }
}
```

10.2 Extend `Main2` (see Section 10.4) so that it places the first instance of `Foo` on a new virtual machine located on a physical machine different from those already used. How many instances of `Glob` are now created, and what does the program output?

10.3 Show the output from the following program. Assume that an invocation of `noop` returns the value 0.

```
public class Main {
  static remote Worker wref[] = new remote Worker[2];
  public static void main(String [] args) {
    for (int i = 0; i < wref.length; i++) {
      wref[i] = new remote Worker(i) on new vm();
    }
    print();
    remote Worker temp = wref[0];
    wref[0] = noop;
    print();
    wref[0] = temp;
```

```
    print();
    cap int() tc = wref[0].g;
    wref[0].g = wref[1].g;
    wref[1].g = tc;
    print();
    wref[0].f = noop;
    wref[1].g = fours;
    print();
    wref[1] = noop;
    wref[1].g = null;
    print();
  }
  public static op int fours() { return 444; }
  public static void print() {
    System.out.println("----");
    for (int i = 0; i < wref.length; i++) {
      System.out.println(wref[i].f(8));
      System.out.println(wref[i].g());
    }
  }
}

public class Worker {
    private final int me;
    public Worker(int me) {
        this.me = me+1;
    }
    public op int f(int i) {
        return me*1000+i;
    }
    public op int g() {
        return me*1000;
    }
}
```

10.4 Consider the following classes and interface.

```
public class A {
    ...
}

public interface B {
    ...
}

public class C implements B {
    ...
}
```

For each assignment below, indicate whether it is legal and briefly explain.

```
remote A a1 = noop;
remote A a2 = null;
remote B b1 = noop;
remote B b2 = null;
remote B b3 = (remote B) noop;
remote B b4 = (remote B) null;
remote B b5 = (remote C) noop;
remote B b6 = (remote C) null;
remote C c1 = noop;
remote C c2 = null;
remote C c3 = (remote B) noop;
remote C c4 = (remote B) null;
remote C c5 = (remote C) noop;
remote C c6 = (remote C) null;
```

10.5 Consider the `Main1` and `Main2` programs in Section 10.4. Each uses `done` messages to inform the main method that all processes have finished. Rewrite each program so that it instead uses a quiescence operation for that purpose.

10.6 Show the output from the `thisvmDemo` program (Section 10.5).

10.7 Rewrite the `thisvmDemo` program (Section 10.5) so it uses parameterized virtual machines.

10.8 Rewrite the `Main2` program from Section 10.6 so it uses parameterized virtual machines using a static variable in the `Myvm` class for the virtual machine identifier, as suggested in Section 10.6.

10.9 Modify the adaptive quadrature program developed for Exercise 7.13(c) so that `AQ` and `fun` are

(a) in different virtual machines on the same physical machine.

(b) in different virtual machines on different physical machines.

10.10 Modify the program developed for Exercise 7.13(d) so that `AQ` and `fun` are

(a) in different virtual machines on the same physical machine.

(b) in different virtual machines on different physical machines.

Time the results as per the instructions in Exercise 7.13(d).

Compare the results and explain any significant differences in the average invocation times for the three programs:

- Exercise 7.13(d).
- Exercise 7.13(d) modified as per (a).
- Exercise 7.13(d) modified as per (b).

10.11 In a multiple virtual machine program, standard output from all virtual machines appears by default on the main virtual machine's `stdout` file. Thus output can be interleaved and therefore difficult to comprehend. In a multiple window UNIX environment, output from each virtual machine can instead be directed to its own window (e.g., running `xterm`).

(a) Demonstrate how to do so in your environment. As a concrete example, use `Main2` or a variant that does more output.

(b) Adapt your solution so that each virtual machine takes its input from its own window.

10.12 Consider the two classes:

```
public class Color implements java.io.Serializable {
  private final String name;
  // declaring constructor as private prevents outsiders
  // from creating new colors;
  // and so can test equality using ==.
  private Color(String name) {
    this.name = name;
  }
  public String toString() { // make it printable for debugging
    return name;
  }

  public static Color Blue = new Color("Blue");
  public static Color Red  = new Color("Red");
}

public class Compare {
  public Compare (Color a, Color b, Color c) {
    System.out.println( (a == b) +" "+ (a == c) );
    Color r = Color.Red;
    System.out.println( (a == r) +" "+ (b == r) +" "+ (c == r) );
  }
}
```

Show the output from each of the following main programs:

(a)
```
public class main {
  public static void main(String [] args) {
    Color a, b, c;
    a = Color.Red;
    b = Color.Blue;
    c = Color.Red;
    new Compare(a, b, c);
  }
}
```

(b)
```
public class main {
  public static void main(String [] args) {
    Color a, b, c;
    a = Color.Red;
    b = Color.Blue;
    c = Color.Red;
    new remote Compare(a, b, c);
  }
}
```

(c)
```
public class main {
  public static void main(String [] args) {
    Color a, b, c;
    a = Color.Red;
    b = Color.Blue;
    c = Color.Red;
    new remote Compare(a, b, c) on new vm();
  }
}
```

Explain any differences in the outputs.

*This page intentionally left blank*

Chapter 11

# THE DINING PHILOSOPHERS

This chapter presents three solutions to the classic Dining Philosophers Problem [17]. The problem was described in Section 6.2. This problem is interesting because it raises aspects of resource allocation problems that real operating and distributed systems must deal with. In particular, avoiding deadlock and starvation (or lack of fairness) are important goals in solutions to this and similar problems.

Our three solutions employ many JR communication mechanisms and illustrate different ways to structure solutions to synchronization problems. They also illustrate how to use virtual machines so that a program can execute on several physical machines. Unlike the other chapters in this part, this chapter introduces no new language mechanisms. It does, however, show mechanisms used in different combinations than seen earlier.

This problem can be solved in several ways in JR. A semaphore solution was already given in Section 6.2. In contrast, the solutions in this chapter distribute the philosophers and servant(s) onto several virtual machines and use other synchronization mechanisms. In our three solutions, philosophers are represented by processes. The solutions differ in how forks are managed, as shown in the following table:

| *Approach* | *Servant(s)* |
|---|---|
| centralized | one servant |
| distributed | one servant per fork |
| decentralized | one servant per philosopher |

The first approach is to have a single, centralized servant process that manages all `n` forks. The second approach is to distribute the forks among `n` servant processes, with each servant managing one fork. The third approach is to decentralize control but employ one servant per philosopher instead of one servant per fork.

## 11.1 Centralized Solution

This approach employs a single servant process that manages all `n` forks. Each philosopher requests two forks from the servant, eats, and then releases the forks back to the servant. This interaction for `n` equals 5 is illustrated in Figure 11.1. In the figure a P represents a philosopher, the S represents the servant, and the lines represent communication.

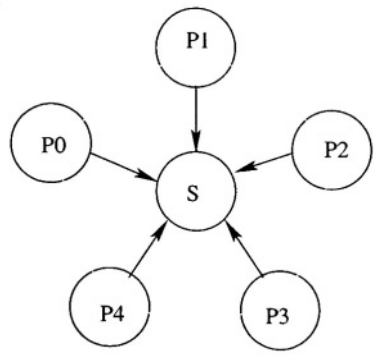


*Figure 11.1.* Structure of centralized solution

Our solution employs three classes: `Servant`, `Philosopher`, and `Main`. Execution begins in `Main`. It first reads command-line arguments specifying the number of philosophers (`n`) and the number of "sessions" each philosopher is to execute (`t`). Main then creates one instance of `Servant` and `n` instances of `Philosopher`. The instance of `Servant` is passed the number of philosophers. Each instance of `Philosopher` is passed a capability for the instance of Servant, the philosopher's identity, and the number of sessions.

```
public class Main {
  public static void main(String [] args) {
    // get number of philosophers and sessions from args
    int n = Integer.parseInt(args[0]);
    int t = Integer.parseInt(args[1]);
    // create the Servant and Philosophers
    remote Servant s = new remote Servant(n);
    for(int i = 0; i < n; i++) {
      new remote Philosopher(s, i, t);
    }
  }
}
```

Each philosopher alternately eats and thinks for `t` sessions. Before eating, a philosopher calls the servant's `getforks` operation; after eating, it calls the servant's `relforks` operation. The `Servant` object services invocations of `getforks` and `relforks` from all instances of `Philosopher`. Each

`Philosopher` object passes its `id` to these operations to allow the servant to distinguish between philosophers. A philosopher is permitted to eat when neither of its neighbors is eating.

```
public class Philosopher {
  remote Servant s;
  int id, t;
  public Philosopher(remote Servant s, int id, int t) {
    this.s = s; this.id = id; this.t = t;
  }
  private process phil {
    for (int i = 1; i <= t; i++) {
      call s.getforks(id);
      System.out.println("Philosopher "+id+" is eating");
      call s.relforks(id);
      System.out.println("Philosopher "+id+" is thinking");
    }
  }
}
```

The `Servant` constructor is passed the number of philosophers (`n`) as a parameter. It uses this value to allocate the array `eating`, which indicates the status of each philosopher. The `server` process continually services the operations `getforks` and `relforks`. The synchronization expression on `getforks` uses the invocation parameter `id` together with `n` to determine, using modular arithmetic, whether either of a philosopher's neighbors is eating. If neither neighboring philosopher is eating, `server` grants the philosopher requesting forks permission to eat and updates the philosopher's entry in `eating`.

```
public class Servant {
  // operations invoked by philosophers
  public op void getforks(int);
  public op void relforks(int);
  int n;
  boolean [] eating; // status of philosophers
  public Servant(int n) {
    this.n = n;
    eating = new boolean [n];
    for (int i = 0; i < n ; i++) {
      eating[i] = false;
    }
  }
  process server {
    while (true) {
      inni void getforks(int id) st oktoeat(id) {
        eating[id] = true;
      }
      [] void relforks(int id) {
        eating[id] = false;
```

```
      }
    }
  }
  // check whether either philosopher that neighbors
  //  philosopher id is eating.  (add in n so that
  // remainder (%) returns non-negative result.)
  private boolean oktoeat(int id) {
    return !eating[((id+n)+1)%n] && !eating[((id+n)-1)%n];
  }
}
```

The above solution is deadlock-free since, in effect, `getforks` allocates both forks at the same time. However, a philosopher can starve if its two neighbors "conspire" against it, i.e., if at any time at least one of them is eating.

The above program executes on a single virtual machine and, therefore, on a single physical machine. It can be easily modified, though, so that each philosopher executes on a different virtual machine. Only `Main`'s loop needs to be changed as follows:

```
for(int i = 0; i < n; i++) {
  vm vmcap = new vm();
  new remote Philosopher(s, i, t) on vmcap;
}
```

As seen in Section 10.2, the creation of a virtual machine can also specify the physical machine on which to create the virtual machine. So, the above loop can be further modified so as distribute the philosophers somewhat evenly over a collection of physical machines (an attempt at primitive load balancing).

```
String [] hosts = {"camelot", "excalibur", "magic"};
for(int i = 0; i < n; i++) {
  vm vmcap = new vm() on hosts[i % hosts.length];
  new remote Philosopher(s, i, t) on vmcap;
}
```

This code uses an array of machine names (each element is a string). To port a program containing code such as that in the above loops to another installation, only the `hosts` array needs to be changed to contain the local machine names. To make the program more portable, the array can be read from a file or from command-line arguments. In fact, that array might be defined in a separate class and accessed as needed.

## 11.2 Distributed Solution

The centralized solution to the dining philosopher's problem is deadlock-free, but it is not fair. Also, the single servant could act as a bottleneck because all philosophers need to interact with it. In contrast, the distributed solution

employs one servant per fork, and it is deadlock-free and fair. Each philosopher interacts with two servants to obtain the forks it needs. A philosopher that is hungry may eat after it obtains a fork from each of the servants. This interaction is illustrated in Figure 11.2.

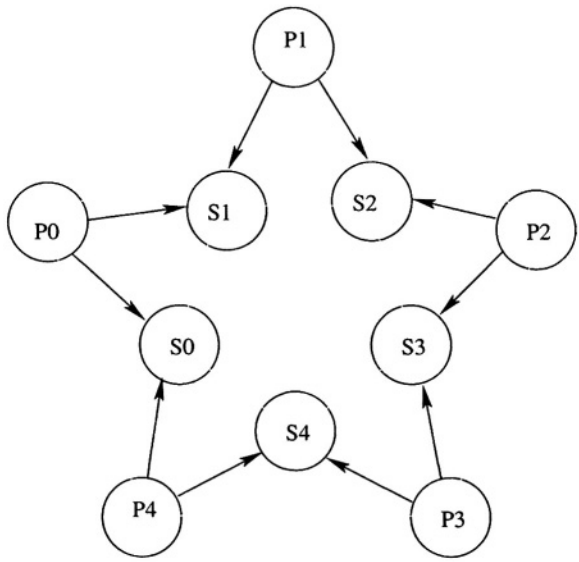


*Figure 11.2.* Structure of distributed solution

Our solution for this approach again employs three classes: `Servant`, `Philosopher`, and `Main`. The `Main` class is similar to `Main` in the centralized solution. It differs in that `Main` now creates `n` instances each of `Philosopher` and `Servant` and passes references for the latter to the former so they can communicate with each other.

```
public class Main {
  public static void main(String [] args) {
    // get number of philosophers and sessions from args
    int n = Integer.parseInt(args[0]);
    int t = Integer.parseInt(args[1]);
    // create the Servants
    remote Servant s[] = new remote Servant[n];
    for(int i = 0; i < n; i++) {
      s[i] = new remote Servant(i);
    }
    // create the Philosophers; to prevent deadlock,
    // they are passed capabilities for their servants
    // in an asymmetric fashion
    for(int i = 0; i < n-1; i++) {
      new remote Philosopher(s[i], s[i+1], i, t);
```

```
    }
    new remote Philosopher(s[0], s[n-1], n-1, t);
  }
}
```

The asymmetric way in which references for servants are passed as constructor parameters to instances of `Philosopher` makes deadlock easy to avoid, as discussed later.

The `Philosopher` class is also similar to its counterpart in the centralized solution. The differences are that it is now passed references for two `Servants`, and it now invokes `getfork` and `relfork` in each of those two `Servants`.

```
public class Philosopher {
  remote Servant l, r;
  int id, t;
  public Philosopher(remote Servant l, remote Servant r,
                     int id, int t) {
    this.l = l; this.r = r; this.id = id; this.t = t;
  }
  private process phil {
    for (int i = 1; i <= t; i++) {
      call l.getfork(); call r.getfork();
      System.out.println("Philosopher "+id+" is eating");
      call l.relfork(); call r.relfork();
      System.out.println("Philosopher "+id+" is thinking");
    }
  }
}
```

The `server` process in each `Servant` object continually services invocations of first `getfork` and then `relfork` from its two associated `Philosopher` objects. This ensures that the servant's fork is allocated to at most one philosopher at a time. A philosopher is permitted to eat when it obtains a fork from each of its two servants.

```
public class Servant {
  public op void getfork();
  public op void relfork();
  int id;
  public Servant(int id) {
    this.id = id;
  }
  process server {
    while (true) {
      receive getfork(); receive relfork();
    }
  }
}
```

The distributed solution is deadlock-free. When `Main` creates instances of `Philosopher`, it passes them references for their left and right servants. The order of these references is switched for the last philosopher (i.e., the `Philosopher` passed an id of `n`). Thus the last philosopher requests its right fork first, whereas each other philosopher requests its left fork first. This avoids the typical deadlock scenario in which each philosopher picks up one of its forks and then requests its other fork. A more formal way to state this property, as defined in operating systems texts, is that requests from philosophers cannot form a cycle in the resource (i.e., fork) allocation graph. Unlike the centralized solution, the distributed solution also prevents starvation since forks are allocated one at a time and invocations of `getfork` are serviced in order of their arrival.

## 11.3 Decentralized Solution

The decentralized solution employs one servant per philosopher. Each philosopher interacts with its own personal servant; that servant interacts with its two neighboring servants. Each individual fork either is held by one of the two servants that might need it or is in transit between them. A philosopher that is hungry may eat when its servant holds two forks. This interaction is illustrated in Figure 11.3.

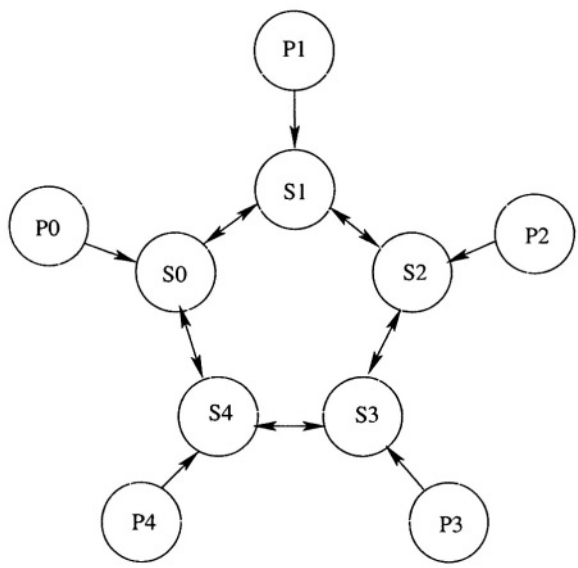


*Figure 11.3.* Structure of decentralized solution

The specific algorithm that the servants employ is adapted from Reference [15]. It has the desirable properties of being deadlock-free and fair. The

basic solution strategy also has applications to other, realistic problems such as file replication, distributed database consistency, and distributed mutual exclusion.

Our decentralized solution once again employs `Servant`, `Philosopher`, and `Main` classes. `Main` is similar to its counterpart used in the distributed solution. The differences are that `Main` in this solution passes different combinations of references—to support the communication structure in Figure 11.3—and it sends each instance of `Servant` the initial values for its local variables.

```
public class Main {
  public static void main(String [] args) {
    // get number of philosophers and sessions from args
    int n = Integer.parseInt(args[0]);
    int t = Integer.parseInt(args[1]);
    // create the Servant and Philosophers
    remote Servant s[] = new remote Servant[n];
    for(int i = 0; i < n; i++) {
      s[i] = new remote Servant(i);
      new remote Philosopher(s[i], i, t);
    }
    // give each Servant capabilities for
    // its two neighboring Servants
    for (int i = 0; i < n; i++) {
      int left = (i-1<0?n-1:i-1), right = (i+1>n-1?0:i+1);
      send s[i].links(s[left],s[right]);
    }
    // initialize each Servant's forks;
    // initialization is asymmetric to prevent deadlock;
    // see Servant class for parameters' meanings.
    send s[0].forks(true, true, true, true);
    for (int i = 1; i < n-1; i++) {
              send s[i].forks(false, false, true, false);
    }
    send s[n-1].forks(false, false, false, false);
  }
}
```

References for a philosopher's servant are passed to the philosopher's constructor as parameters. On the other hand, references for a servant's neighbor are passed to it via a separate operation, `links`. This dissimilarity results from the fact that `Main` has to create the instances of `Servant` before it passes them the references for each other. Here the servants are arranged circularly; therefore, no order of creation would allow this code to pass references for servants as parameters to instances of `Servant`.

As in the previous solutions, each philosopher alternately eats and thinks for `t` sessions. Before eating, a philosopher calls the `getforks` operation in its personal instance of `Servant`; after eating, it calls the `relforks` operation in

that `Servant`. Thus `Philosopher` here is almost identical to `Philosopher` in the centralized solution. The only difference is that a philosopher no longer passes its identity to `getforks` or `relforks`. Each philosopher now interacts with a single servant, whereas in the centralized solution, the one servant is shared by all philosophers.

```
public class Philosopher {
  remote Servant s;
  int id, t;
  public Philosopher(remote Servant s, int id, int t) {
    this.s = s; this.id = id; this.t = t;
  }
  private process phil {
    for (int i = 1; i <= t; i++) {
      call s.getforks();
      System.out.println("Philosopher "+id+" is eating");
      call s.relforks();
      System.out.println("Philosopher "+id+" is thinking");
    }
  }
}
```

`Servant` objects service invocations of `getforks` and `relforks` from their associated `Philosopher` objects. They communicate with neighboring `Servant` objects using the `needL`, `needR`, `passL`, and `passR` operations. A philosopher is permitted to eat when its servant has acquired two forks. A servant may already have both forks when `getforks` is called, or it may need to request one or both from the appropriate neighbor.

Two variables are used to record the status of each fork: `haveL` (`haveR`) and `dirtyL` (`dirtyR`). Starvation is avoided by having servants give up forks that are dirty; a fork becomes dirty when it is used by a philosopher. Further details on the servant's algorithm are in Reference [15].

```
public class Servant {
  // operations invoked by associated philosopher
  public op void getforks();
  public op void relforks();
  // operations invoked by neighboring servants
  public op void needL();
  public op void needR();
  public op void passL();
  public op void passR();
  // initialization operations invoked by Main
  public op void links(remote Servant l, remote Servant r);
  public op void forks(boolean haveL,  boolean haveR,
                       boolean dirtyL, boolean dirtyR);
  int id;
  public Servant(int id) {
```

```
  this.id = id;
}
private op void hungry();
private op void eat();
public void getforks() {
  send hungry(); // tell server Philosopher is hungry
  receive eat(); // wait for permission to eat
}
process server {
  remote Servant l, r;
  receive links(l,r);
  boolean haveL, dirtyL, haveR, dirtyR;
  receive forks(haveL, dirtyL, haveR, dirtyR);
  while (true) {
    inni void hungry() {
      // ask for forks I don't have; I ask my
      // right neighbor for its left fork,
      // and my left neighbor for its right fork
      if (!haveR) { send r.needL(); }
      if (!haveL) { send l.needR(); }
      // wait until I have both forks
      while( !(haveL && haveR) ) {
        inni void passR() {
          haveR = true; dirtyR = false;
        }
        [] void passL() {
          haveL = true; dirtyL = false;
        }
        [] void needR() st dirtyR {
          haveR = false; dirtyR = false;
          send r.passL(); send r.needL();
        }
        [] void needL() st dirtyL {
          haveL = false; dirtyL = false;
          send l.passR(); send l.needR();
        }
      }
      // let my Philosopher eat;
      // then wait for it to finish
      send eat(); dirtyL = true; dirtyR = true;
      receive relforks();
    }
    [] void needR() {
      // neighbor needs my right fork (its left)
      haveR = false; dirtyR = false;
      send r.passL();
    }
    [] void needL() {
      // neighbor needs my left fork (its right)
      haveL = false; dirtyL = false;
```

```
        send l.passR();
      }
    }
  }
}
```

Notice the various combinations of operation invocation and servicing that are employed. For example, `getforks` provides a procedural interface and hides the fact that getting forks requires sending a `hungry` message and receiving an `eat` message. Also, server processes use a send to invoke the need and pass operations serviced by neighboring servers; a call cannot be used for this because deadlock could result if two neighboring servers invoked each other's operations at the same time.

A philosopher and its servant are represented as separate classes. This structure provides a clean separation of their functionalities. However, they can be combined into a single class (see Exercise 11.6).

The structure of the servants and their interaction in the above example is typical of that found in some distributed programs, such as those that implement distributed voting schemes. Such a program might contain a collection of voter processes, each of which can initiate an election. After a voter process initiates an election, it tallies the votes from the other voters. While a voter process is waiting for votes to arrive for its election, it must also be able to vote in elections initiated by other voter processes. This kind of communication can be accomplished easily only by using an asynchronous send.

In the above example, when a servant is attempting to acquire both forks for its philosopher, it might give up a fork it already possesses (because the fork is dirty). In this case it passes the fork to its neighbor and then immediately requests the fork's return. To reduce the number of messages exchanged, the request for the fork's return could be combined with the passing of the fork. In particular, the pass operations could be parameterized with a boolean that indicates whether or not the servant, when its philosopher has finished eating, should automatically pass the fork back to its neighbor (see Exercise 11.7).

## Exercises

11.1 Give a solution to the dining philosophers problem in which all processes reside in a single object and use semaphores to synchronize. (No servant process is needed.)

11.2 Repeat the previous exercise, but define the semaphores within a separate class and represent each philosopher as its own object.

11.3 Modify the code in the distributed solution so that each philosopher is created on its own virtual machine and the one servant used by that philosopher is created on the same virtual machine. Show how to modify

your solution so that each virtual machine executes on its own physical machine.

11.4 Modify the code in the decentralized solution so that each philosopher and its servant is created on a separate virtual machine. Show how to modify your solution so that each virtual machine executes on its own physical machine.

11.5 Rewrite the code for the decentralized solution so that it uses only call invocations. (Good luck!)

11.6 Show how, in the decentralized solution, to combine philosophers and servants into a single class, as suggested at the end of Section 11.3. Can this also be done in the distributed solution?

11.7 Show how to write the variant of the decentralized solution suggested at the end of Section 11.3 to reduce message passing.

11.8 A philosopher and its servant are represented as separate processes in the decentralized solution. Can they be combined into a single process? If so, show how. If not, explain why not.

11.9 Modify the decentralized solution so that servants terminate. A servant can terminate once its philosopher has terminated and its neighboring servants will no longer request a common fork from it.

11.10 In all the dining philosopher solutions, the operations to release one or both forks have been invoked using calls. For each solution, state whether it will work if those calls are replaced by sends. Justify your answer.

11.11 Run each solution to the dining philosophers problem.

(a) Evaluate the performance of the different solutions. Which is fastest? Which is slowest? How much does performance differ? Explain the reasons for your answers.

(b) Does the order in which philosophers eat differ? If so, explain why. If not, explain why not.

11.12 *Distributed mutual exclusion.* Given are $n$ user processes, each of which repeatedly executes a critical section of code and then a non-critical section. At most one process at a time is permitted to execute its critical section of code.

Develop centralized, distributed, and decentralized solutions to this problem. The structures of your solutions should be similar to those for the

dining philosophers problem. The centralized solution should have one "servant" process with which all $n$ users interact. The distributed solution should have $n$ servant processes; each user process should interact with all $n$ servants. The decentralized solution should also have $n$ servant processes, but each user should interact with just one servant; the servants should also interact with each other.

*This page intentionally left blank*

Chapter 12

# EXCEPTIONS

The Java programming languages provides an exception handling model to support the reporting of errors and the handling exceptional conditions (e.g., I/O errors). To fully support its extensions to Java, JR adapts and extends Java's exception handling model. This chapter first discusses the support for exceptions with respect to operations and input statements. The chapter then discusses the extension of the exception handling model with **handler** objects to support throwing exceptions from asynchronously invoked operations.

## 12.1 Operations and Capabilities

An operation (or op-method) declaration is very similar to a method declaration in Java (see Chapter 3); an operation declaration specifies both its return type and parameter types (and body for an op-method). This similarity extends, as well, to the `throws` clauses used in Java to specify which exceptions can be thrown from a method. The addition of the `throws` clause completes the general form of an operation declaration, which is now

[*modifiers*] `op` *return_type id* (*parameter_types*) [throws *exception_list*]

An example of an operation that may throw an exception is:

```
op String readFile() throws IOException;
```

An op-method declaration, of course, would include a method body.

Operation capabilities are similarly extended. The complete general form of an operation capabilities declaration is:

`cap` *return_type* (*parameter_types*) [throws *exception_list*] *cap_id*

In addition to the parameterization requirements discussed in Section 3.3, an operation capability can be bound only to an operation with a matching `throws`

clause. This requirement prevents attempts to propagate out of invocations of capabilities and operations exceptions not listed in the respective `throws` clauses.

## 12.2 Input Statements

Input statements now support exception handling in a manner similar to operations. For each operation or capability serviced by an input statement, a `throws` clause must list all exceptions that may be thrown from the body of the respective operation command. For instance, if the servicing code for an operation includes file I/O, then an `IOException` may be thrown. If this exception is propagated out of the body of the operation command, then it must be included in the `throws` clause as demonstrated by the following code:

```
inni String readFile() throws IOException { ... }
[] void writeFile(String data) throws IOException { ... }
```

The general form of the input statement discussed in Section 9.1.1 now includes an optional `throws` clause as part of each operation command. An operation command now has the general form

*return_type op_expr* ( *formal_list* ) [`throws` *exception_list*] `st` *synch_expr* `by` *sched_expr*
*block*

The exceptions listed in the `throws` clause of an operation command must match those listed in the `throws` clause of the operation or capability specified by the *op-expr* of the operation command.

An exception thrown from the body of an operation command is propagated (except as discussed in Section 12.3) back to the invoker. Such an exception is never propagated into the enclosing scope of the offending input statement. As such, the enclosing scope does not require a `catch` clauses for the exceptions listed in the operation commands.

## 12.3 Asynchronous Invocation

The exception handling mechanisms provided by sequential programming languages rely upon the call chain for the propagation of exceptions. A thrown exception is propagated (either implicitly or explicitly) up the call chain until an appropriate handler is found. Figure 12.1 depicts such a propagation in a parser program. Method `read` throws an exception because of an I/O error. The exception is propagated through method `parse1` and into method `parse`, where it is finally handled.

Asynchronous invocations, however, "break" links in the call chain and render such propagations impossible. Figure 12.2 depicts the same program as before, but with the method `parse1` invoked asynchronously (in this modified program each file is parsed concurrently). Again, an exception is thrown from

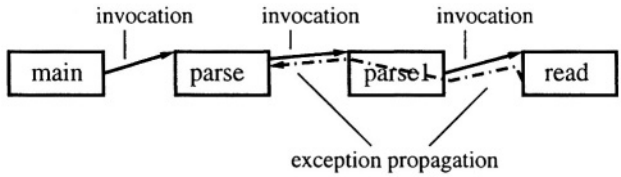


*Figure 12.1.* Exception propagated through call chain

method `read` and propagated to method `parse1`. If method `parse1` does not have an appropriate exception handler, then the exception must be further propagated. But, since method `parse1` was invoked asynchronously, the preceding link in the call chain is not accessible. In fact, the method that invoked `parse1` (in this example, `parse`) may no longer be executing. Therefore, the call chain cannot be used to propagate exceptions from methods invoked asynchronously.

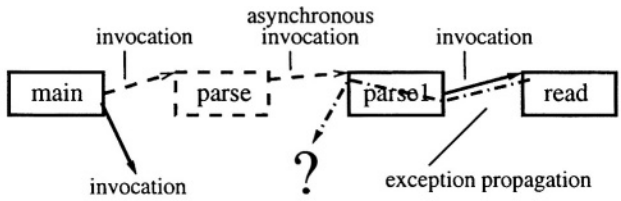


*Figure 12.2.* Exception propagated from method invoked asynchronously

JR addresses this problem by providing **handler** objects and requiring the specification of handler objects for exceptions thrown from asynchronously invoked operations. More precisely, a handler object must be specified at each point that a link in the call chain can be broken. These points are at each send, `reply`, or `forward` statement. Handler objects and their specification for each of the asynchronous statements are discussed in the sections that follow.

### 12.3.1 Handler Objects

A **handler** object is an instance of a class that (1) implements the `Handler` interface in the `edu.ucdavis.jr` package and (2) defines a set of **handler** methods. A method is defined as a **handler** through the use of the `handler` modifier (much like the use of the `public` modifier). A **handler** method takes only a single argument: a reference to an exception object. Each **handler** method specifies the exact exception type that it can handle (e.g., `FileNotFoundException`). When an exception is delivered to a **handler** object, it is handled by the **handler** method of the appropriate type (which is synchronously invoked). Operations can also be defined as **handlers** using the same modifier (see Section 12.5 for further discussion).

An example definition of a **handler** object's class follows:

```
class IOHandler implements edu.ucdavis.jr.Handler {
  public handler void handleEOF(EOFException e) {
    ... // handle exception
  }
  public handler void handleNotFound(FileNotFoundException e) {
    ... // handle exception
  }
}
```

In this example, **handler** objects of type `IOHandler` can handle end-of-file and file-not-found exceptions. An exception of type `EOFException` directed to such a **handler** object will be handled by the `handleEOF` method; exceptions of type `FileNotFoundException` will be handled by the `handleNotFound` method.

### 12.3.2 Send

A `send` statement must specify, using a `handler` clause, the **handler** object that is to be used to handle any exceptions propagated from the asynchronously invoked operation. (Of course, if the operation has no `throws` clause, then a **handler** object is unnecessary.) The following code demonstrates the specification of an instance of the `IOHandler` class described above:

```
op void read(String filename) throws FileNotFoundException;
...
IOHandler ioHandler = new IOHandler();
send read(filename) handler ioHandler;
```

A **handler** object must be capable of handling any exceptions thrown from the operation. Such an object, defined as discussed above, must define a **handler** method for each exception thrown[1]. For the operation read in the example, the **handler** object must support a **handler** method for `FileNotFoundException` (which it does).

To illustrate how **handler** objects work, suppose the above examples are combined into a complete, but simple program

```
import java.io.FileNotFoundException;
import java.io.EOFException;
public class simple {
  static op void read(String filename)
   throws FileNotFoundException {
```

[1] More precisely, the **handler** methods may handle exceptions that are supertypes of the thrown exceptions, resulting in potentially fewer **handler** methods than exceptions. In addition, a **handler** may actually be defined as an operation (see Section 11.6).

```
    ... // open and read from file
  }

  public static void main(String [] args) {
    try {
      IOHandler ioHandler = new IOHandler();
      send read(args[0]) handler ioHandler;
    }
    catch (Exception e) {
       e.printStackTrace();
    }
  }
}

class IOHandler implements edu.ucdavis.jr.Handler {
  public handler void handleEOF(EOFException e) {
    ... // handle exception
  }
  public handler void handleNotFound(FileNotFoundException e) {
    ... // handle exception
  }
}
```

Now, suppose that the `read` operation attempts to open its `filename` argument and a `FileNotFoundException` exception is raised. Since read was invoked asynchronously, the raised exception is propagated to the **handler** object specified as part of the invocation, `ioHandler` in this example. `ioHandler` refers to an instance of `IOHandler`, which supports two **handler** methods: `handleEOF` and `handleNotFound`. Since the propaged exception matches the argument of the `handleNotFound` method, this method is selected and invoked to handle the raised exception.

## 12.4 Additional Sources of Asynchrony

During a *synchronous* method invocation, asynchrony can arise if the invoked operation executes

- an early reply.
- a forward.

In these cases, we can use handler objects to capture exceptions that occur during execution of asynchronous activity.

### 12.4.1 Exceptions After Reply

Much like an asynchronous invocation, an early reply breaks the call chain link between the invoking method and the invoked operation (recall that a reply can only be used within an operation's servicing code). As such, exceptions

thrown *after* a reply cannot be propagated back to the invoking method. Therefore, a `reply` statement must specify a **handler** object that is capable of handling any exceptions potentially thrown by the replying method as demonstrated by the following code:

```
static op int read(String filename) throws EOFException {
  int retval = 0;
  ...  // calculate retval
  IOHandler ioHandler = new IOHandler();
  reply retval handler ioHandler;
  ... // continue executing, directing any exceptions to ioHandler
}

public static void main(String [] args) {
  ...
  int value = read(args[0]);
  // execute concurrently with read after read replies
  ...
}
```

Exceptions thrown *before* a `reply` statement are propagated as appropriate for the manner in which the operation was invoked (i.e., through the call chain if invoked synchronously and to a **handler** object if invoked asynchronously). Exceptions thrown *after* a reply are directed to the **handler** object specified as part of the reply statement. Note that a subsequent reply to an invocation for which a reply has already been executed does not return a value to the invoker, but it may change the **handler** object to which exceptions are directed.

### 12.4.2 Exceptions After Forward

A `forward` statement also causes a break in the call chain. Unlike the send and `reply` statements, however, the `forward` statement replaces the broken link with a link to the newly invoked operation. As such, the newly invoked operation can propagate exceptions up the call chain. The forwarding operation, however, must handle exception thrown after forwarding locally or direct them to a **handler** object. The following code demonstrates the specification of a **handler** object as part of a `forward` statement:

```
static op String readFile(File file) throws EOFException {
  ...  // read file
}
static op String read(String filename) throws EOFException {
  ...  // check access rights and open file for reading
  IOHandler ioHandler = new IOHandler();
  forward readFile(file) handler ioHandler;
  ... // continue executing, directing any exceptions to ioHandler
}
public static void main(String [] args) {
```

```
    ...
    String value = read(args[0]);
    // execute concurrently with read after read forwards
    // wait for reply from readFile
    ...
}
```

The method to which responsibility is forwarded, `readFile` in the example, inherits the call chain link from the forwarding method, `read`. In the example, method `read` is synchronously invoked. After forwarding, the call chain link that had been between `main` and `read` is changed to link `main` with `readFile`. As such, exceptions thrown from method `readFile` will be propagated through the call chain back to method `main` (the original invoker of method `read`).

Had the method `read` been invoked asynchronously, as demonstrated by the following code, any exceptions thrown from method `readFile` would be directed to the **handler** object specified as part of the asynchronous invocation of method `read` (the object referred to by `asynchHandler` in the following code).

```
static op void readFile(File file) throws EOFException {
  ... // read file
}
static op void read(String filename) throws EOFException {
  ... // check access rights and open file for reading
  IOHandler ioHandler = new IOHandler();
  forward readFile(file) handler ioHandler;
  ... // continue executing, directing any exceptions to ioHandler
}
public static void main(String [] args) {
  ...
  IOHandler asynchHandler = new IOHandler();
  send read(args[0]) handler asynchHandler;
  // execute concurrently with read
  ...
}
```

## 12.5 Exceptions and Operations

Section 12.3.1 discusses the definition of **handler** objects. A **handler** object must define a set of **handler** methods, one for each exception type that is to be handled. These **handler** methods may, however, be operations. The use of operations provides additional flexibility in handling exceptions. In particular, the use of operations serviced by input statements allows the handling of exceptions to be distributed among a number of threads (each executing an input statement servicing the operation) or serialized by a single thread servicing multiple **handler** operations with a single input statement. Distributing exception handling

allows exceptions to be routed to the appropriate components of the system. Serializing exception handling can provide a means for prioritizing exception handling.

The following code shows the class definition of a **handler** object that handles exceptions using an input statement. The `server` thread repeatedly services exceptions (one at a time) as they are propagated to the **handler** object.

```
class IOHandler implements edu.ucdavis.jr.Handler {
  public handler op void handleEOF(EOFException e);

  public handler op void handleNotFound(FileNotFoundException e);

  process server {
    try {
      while (true) {
        inni void handleEOF(EOFException e) {
          ...
        }
        [] void handleNotFound(FileNotFoundException e) {
          ...
        }
      }
    } catch (Exception e) {
        e.printStackTrace();
    }
  }
}
```

If this example were combined with code similar to that in Section 12.3.2, but that executes two sends with the same **handler** object (e.g., `ioHandler` is reused), and exceptions are raised from each, then the `inni` handles the exceptions one at a time in the order in which the **handler** object received notification that they were raised.

Operations in a **handler** object can also be used to provide a means for an invoking method to monitor the exception status of an asynchronously invoked operation with which it is concurrently executing. Instead of the **handler** object servicing the **handler** operations (as in the above example), the invoker (or any other thread) can check for and handle any raised exceptions by executing an `inni` statement (similar to that in the above example).

## Exercises

12.1 What is the output for the following program that uses a `reply`? (Assume that the `reply` statement itself does not raise an `Exception`.)

```
import java.io.FileNotFoundException;
import java.io.EOFException;
```

```
public class simple {
  static op int read(String filename) throws EOFException {
    try {
      reply filename.length() handler new IOHandler();
    } catch (Exception e) {e.printStackTrace();}
    throw new EOFException();
  }

  public static void main(String [] args) {
    try {
      int val = read("abc");
      System.out.println(val);
    }
    catch (EOFException e) {
      System.out.println("Caught EOFException with try/catch");
    }
    catch (Exception e) {
       System.out.println("error");
    }
  }
}

class IOHandler implements edu.ucdavis.jr.Handler {
  public handler void method(EOFException e) {
    System.out.println("Caught EOFException");
  }
  public handler void method(FileNotFoundException e) {
    System.out.println("Caught FileNotFoundException");
  }
}
```

12.2 What is the output for the following program that uses a `forward`? (Assume that the `forward` statement itself does not raise a `Exception`.)

```
import java.io.FileNotFoundException;
import java.io.EOFException;
import java.io.File;

public class simple {
  static op String readFile(File file) throws EOFException {
    throw new EOFException();
  }
  static op String read(String filename) throws EOFException {
    try {
      forward readFile(null) handler new IOHandler();
    } catch (Exception e) {e.printStackTrace();}
    throw new EOFException();
  }
```

```
  public static void main(String [] args) {
    try {
      String val = read("abc");
      System.out.println(val);
    }
    catch (EOFException e) {
      System.out.println("Caught EOFException with try/catch");
    }
    catch (Exception e) {
      System.out.println("error");
    }
  }
}

class IOHandler implements edu.ucdavis.jr.Handler {
  public handler void method(EOFException e) {
    System.out.println("Caught EOFException");
  }
  public handler void method(FileNotFoundException e) {
    System.out.println("Caught FileNotFoundException");
  }
}
```

12.3 What is the output for the following program that uses a `forward` and a `send`? (Assume that neither the `forward` nor the `send` raises an `Exception`.)

```
import java.io.FileNotFoundException;
import java.io.EOFException;
import java.io.File;

public class simple {
  static op void readFile(File file) throws EOFException {
    throw new EOFException();
  }
  static op void read(String filename) throws EOFException {
    try {
      forward readFile(null) handler new IOHandler("forward");
    }
    catch (Exception e) {
      System.out.println("error");
    }
    throw new EOFException();
  }

  public static void main(String [] args) {
    try {
      send read("abc") handler new IOHandler("send");
```

```
      }
      catch (Exception e) {
        System.out.println("error");
      }
    }
  }

  class IOHandler implements edu.ucdavis.jr.Handler {
    String _str;
    public IOHandler(String str) {
      this._str = str;
    }
    public handler void method(EOFException e) {
      System.out.println("Caught EOFException from " + _str);
    }
    public handler void method(FileNotFoundException e) {
      System.out.println("Caught FileNotFoundException from " +
                         _str);
    }
  }
```

12.4 Modify the Bounded Buffer example from Chapter 9 so that an exception (`OutOfRangeException`) is raised if the item (integer) that is about to be deposited into the buffer is not within a valid range. Define the `OutOfRangeException` and test your program with a valid range of 20 to 2000, inclusive.

12.5 (a) Modify the Readers/Writers example from Chapter 9 so that occasionally, when reading the database, a reader process terminates. Have this termination generate an exception (i.e., the reader terminates by raising an exception). Use a single exception handler object for all readers; in handling the exception, the handler object needs to invoke the end_read operation on behalf of the terminating reader. Note that the `process` abbreviation does not allow a `throws` clause, so each reader process must be started with an explicit `send`. To test your program, have the reader raise an exception based on a random value.

(b) Modify the program in part (a) so that a writer process may also terminate with an exception while accessing the database. Use a single exception handler object, separate from that for readers, for all writers.

(c) Modify the program in part (b) to use a single exception handler object for both readers and writers.

12.6 The **handler** clause for the `send`, `forward`, and `reply` statements allows only a single **handler** object to be specified. Discuss any limitations

this might impose. Discuss how one might be able circumvent these limitations (consider an aggregate object).

Chapter 13

# INHERITANCE OF OPERATIONS

This chapter defines and illustrates how operations can be inherited in JR. Overriding inherited operations allows a subclass to specialize the implementation of those operations. Such specialization also facilitates changing the manner in which an operation is serviced. This change in servicing enables, among other things, the distribution of the servicing of an operation and the filtering of invocations of a distributed operation.

This chapter first introduces the general notion of operation inheritance and possibilities for redefinition in a subclass. It then illustrates the use of operations inheritance through two examples. Finally, it presents some fine points that one must consider when redefining the manner in which an operation is serviced.

This chapter assumes an understanding of inheritance in Java, but it is briefly discussed again here to highlight some important points for the ensuing discussion of operation inheritance. A Java class definition consists of a specification and an implementation (a Java interface consists of only a specification). The specification of a class defines the external interface "exported" by instances of the class. An instance of a class is used by invoking instance methods declared in the class' specification. The implementation of a class defines the actions taken when a method in the specification is invoked (i.e., the implementation defines the statements that are executed upon invocation).

In Java, a new class may be derived from an existing class to create a subclass. Through this derivation, the subclass inherits the specification of its parent class. By default, a subclass also inherits its parent's implementation of the specification. A subclass can extend the inherited specification through the addition of methods. Similarly, a subclass can modify the implementation of its inherited specification by redefining the inherited methods.

## 13.1 Operation Inheritance

In JR, a derived class may modify the implementation of its inherited specification by redefining the implementation of its inherited methods and operations. An inherited method's implementation is modified, as in standard Java, by redefining the method. Operations are classified according to their implementation. An operation that is associated with a method is termed a ProcOp. An operation that is serviced by an `inni` statement is termed an InOp. In general, JR allows a subclass to redefine the implementation of an inherited operation as either a ProcOp or an InOp, regardless of the operation's implementation in the superclass.

The different combinations of operation redefinition are discussed below. In brief, an operation's implementation is redefined by either redefining the servicing code or switching the manner in which the operation is serviced. Redefinition of an operation's implementation requires an explicit redeclaration of the operation in the subclass only if the redefinition changes the operation from an InOp to a ProcOp or vice-versa. Otherwise, an explicit redeclaration of the operation is not required.

The notation $\mathbf{Op_1} \rightarrow \mathbf{Op_2}$ means that the superclass defines the operation as an $\mathbf{Op_1}$ and the subclass is redefining the operation to be an $\mathbf{Op_2}$.

1. ProcOp → ProcOp

   A redefinition from a ProcOp to a ProcOp corresponds directly to a method redefinition in standard Java. The subclass can simply redefine the method associated with the operation. Such a redefinition allows a subclass to specialize the operation implementation.

2. InOp → InOp

   The implementation of an InOp is not actually redefined but rather extended. Any `inni` statements that "service" an inherited InOp are added to the set of `inni` statements that implement the operation. A subclass may explicitly redeclare an InOp as an InOp by explicitly redeclaring the operation. This redeclaration allows a subclass to relax access restrictions (i.e., public, private, and protected) but does not create a separate invocation queue.

3. ProcOp → InOp

   A ProcOp may be redefined as an InOp in a subclass by explicitly redeclaring the operation and not defining a signature-compatible method. The signature-compatible method that would have been inherited from the superclass is ignored.

4. InOp → ProcOp

   An InOp may be redefined as a ProcOp in a subclass by both redeclaring the operation and defining a signature-compatible method.

## 13.2 Example: Distributing Operation Servicing

Redefinition of a ProcOp to be an InOp can be used to distribute the servicing of the operation's invocations without changing the client. Figure 13.1 graphically depicts both the original client-server structure, which uses a ProcOp, and the new bag of tasks structure that results from redefining the inherited operation to be an InOp. With the original operation, each invocation results in a new thread being created (at the server) to service the invocation.

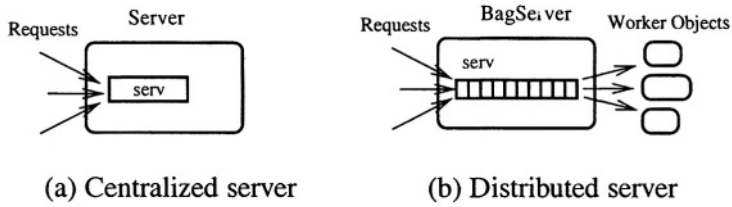


*Figure 13.1.* Distribution of servicing through redefinition of operation in subclass `BagServer`

The following code segment defines the original centralized server. Clients invoke the `serv` operation and wait for the result from the single server.

```
public class Server {
  public op int serv(int i) {
    return i + 1;
  }
}
```

In the following code, the `BagServer` class extends the original `Server` class and redefines the `serv` operation to be an InOp (by redeclaring the operation without defining a signature-compatible method).

```
public class BagServer extends Server {
  // redeclaration of operation as InOp,
  // no signature-compatible method
  public op int serv(int i);

  public BagServer(vm [] hosts) {
    // initialize Worker objects
    for (int i = 0; i < hosts.length; i++) {
      new remote Worker(serv) on hosts[i];
    }
  }
}

public class Worker {
  private cap int (int) srvr;
  public Worker(cap int (int) srvr) {
    this.srvr = srvr;
  }
```

```
  process work() {
    while (true) {
      inni int srvr(int i) {
        return i + 1;
      }
    }
  }
}
```

With the redefined operation, each invocation is handled by an extant `Worker` object, which was created at program startup and which may be located on a separate host. A capability for `serv` is passed to the constructor of each `Worker` object. Each `Worker` object repeatedly executes an `inni` statement to service invocations on `serv`'s queue. These `Worker` objects can be located on an arbitrary set of physical machines as specified by `hosts`.

## 13.3 Example: Filtering Operation Servicing

Redefinition of an InOp to be a ProcOp can be used to filter the invocations of an operation. Figure 13.2 graphically depicts each server configuration. With the original operation, as shown in Figure 13.2 (a), each invocation is serviced by an extant Worker object. Redefinition of the operation, as shown in Figure 13.2 (b), allows for the filtering of invocations in order to reduce the amount of work done by the Worker objects. The following code segment shows the definition of the subclass `FilterServer` that redefines an InOp to be a ProcOp. In the original `Server` class, the `serv` operation is defined as an InOp. `FilterServer` redefines the `serv` operation to be a ProcOp by explicitly redeclaring the operation and defining a signature-compatible method.

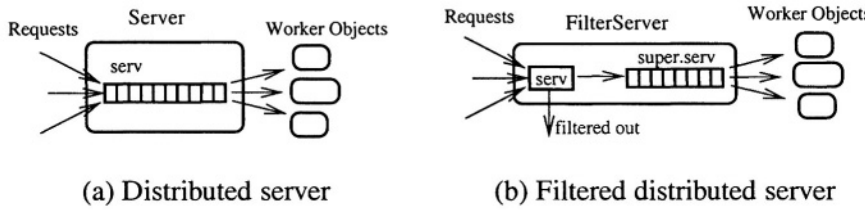


*Figure 13.2.* Filtering of invocations through redefinition of operation in subclass `FilterServer`

```
public class Server {
  public op int serv(int i);

  public Server() {} // empty constructor
  public Server(vm [] hosts) {
    // initialize Worker objects
    for (int i = 0; i < hosts.length; i++) {
      new remote Worker(serv) on hosts[i];
```

```
    }
  }
}

public class Worker {
  private cap int (int) srvr;
  public Worker(cap int (int) srvr) {
     this.srvr = srvr;
  }
  process work() {
    while (true) {
      inni int srvr(int i) {
        return i + 1;
      }
    }
  }
}

public class FilterServer extends Server {
  public static final int DEFAULT = -1;
  // redeclaration of operation
  public op int serv(int i) {
    if (filter(i))
      return DEFAULT;
    else     // forward to Worker objects
      forward super.serv(i);
  }

  // simple filter
  boolean filter(int i) {
    return (i < 0);
  }
  public FilterServer(vm [] hosts) {
    super();
    // initialize Worker objects
    for (int i = 0; i < hosts.length; i++) {
      new remote Worker(super.serv) on hosts[i];
    }
  }
}
```

Each invocation of `serv` defined in the subclass is routed through the method associated with the ProcOp to determine whether or not the invocation will be passed on to a `Worker` object. If the invocation is not rejected by the filter, then the subclass uses a `forward` statement to pass responsibility for servicing the invocation to the InOp `serv` defined in the parent class (`Server`). Each `Worker` object repeatedly executes an `inni` statement to service the InOp `serv` defined in the `Server` class; this operation capability was passed to the constructor of each `Worker` object during initialization in `FilterServer`'s constructor.

## 13.4 Redefinition Considerations

As discussed above, JR requires that a subclass explicitly redeclare an operation if the subclass redefines an inherited InOp as a ProcOp or an inherited ProcOp as an InOp. Such a redeclaration is required to statically determine that the operation has been redefined and to reduce the potential for erroneous code. The redeclaration also serves as a statement of intent.

Imagine that an explicit redeclaration were not required to redefine a ProcOp as an InOp. Instead, assume that an operation is implicitly redefined as an InOp if the operation is serviced by an `inni` statement. Full program analysis would be required to statically determine that the operation has been redefined. However, such an analysis is not sufficient when capabilities are used within `inni` statements. This "redefinition" approach is unsatisfactory because all such redefinitions cannot be discovered until run-time. Furthermore, allowing such a "redefinition" could lead to hard-to-find errors if a programmer accidentally "redefines" the wrong operation in an `inni` statement.

An InOp may be redefined to be a ProcOp if the subclass explicitly redeclares the operation and defines a signature-compatible method. If an `inni` statement attempts to service the operation through a reference to the subclass, then a compile-time error will be raised. If an `inni` statement attempts to service the operation through a reference to the superclass, then a run-time error will be generated. These two cases are illustrated in the following code segment.

```
class A {
  public op void foo();
}

class B extends A {
  public op void foo() {
     ...
  }
}

public class C {
  public static void main(String [] args) {
    ...
    // use of operations
    A a = new B();
    B b = new B();

    inni void b.foo() { // compile-time error - b.foo is a ProcOp
      ...
    }
    [] void a.foo() {   // run-time error - a refers to a B object
      ...
    }
  }
```

JR makes a strict distinction between operations and methods. An inherited operation may not be redefined as a method and an inherited method may not be redefined as an operation. The interested reader can find a detailed discussion of this topic in [30].

## Exercises

13.1 Consider the following interface

```
public interface Intf {
  public op void foo(int);
}
```

Define two classes that implement this interface and a main method that creates an instance of each class and invokes the operation `foo` on these instances.

13.2 Consider the following abstract class

```
public abstract class Abst {
  public abstract op void foo(int);
}
```

Define two classes that extend this class and a main method that creates an instance of each class and invokes the operation `foo` on these instances.

13.3 Starting with the code in Section 13.2, create a centralized server program and a decentralized server program.

(a) Measure the performance of each program (at a minimum you will want to simulate some real computation in the servicing of `serv`).

(b) Discuss the circumstances under which each configuration is preferable.

13.4 Starting with the code in Section 13.3, create a program where the `Worker` objects filter invocations and a program where filtering is done at the `server`.

(a) Measure the performance of each program (at a minimum you will want to simulate some real computation in the servicing of `serv`).

(b) Discuss the circumstances under which each configuration is preferable.

*This page intentionally left blank*

Chapter 14

# INTER-OPERATION INVOCATION SELECTION MECHANISM

Chapter 9 introduced the use of synchronization and scheduling expressions to alter the default invocation servicing semantics of the input (`inni`) statement, which is used for rendezvous. Though powerful, these expressions cannot be used to specify an important set of selection algorithms. Imagine a bank that gives priority service to preferred customers (e.g., businesses or individuals with large balances). When there is a line, the customers are serviced in order of priority based on their status (e.g., large businesses have higher priority than small businesses). Consider the following input statement that attempts to enforce this priority scheduling:

```
inni void withdraw(int priority, double amt) by priority { ... }
[] void deposit(int priority, double amt) by priority { ... }
```

A single execution of this input statement services either an invocation of `deposit` or an invocation of `withdraw`. The scheduling expressions enforce priority scheduling in servicing invocations over each arm. Invocations of `deposit` are serviced in order of their `priority` argument, as are invocations of `withdraw`.

A scheduling expression applies only to the arm on which it is specified. As such, the priority scheduling defined above does not hold over both arms as is demonstrated by the following program segment.

```
send withdraw(100, 23.10);
send deposit(1, 13.75);
send deposit(2, 5.23);
send withdraw(30, 356.73);
send withdraw(40, 6002.11);
while (true)
{
```

```
  inni void withdraw(int priority, double amt) by priority
     { System.out.println("withdraw: " + priority); }
  [] void deposit(int priority, double amt) by priority
     { System.out.println("deposit: " + priority); }
  [] else break;
}
```

This program segment prints:

```
withdraw : 30
withdraw : 40
withdraw : 100
deposit : 1
deposit : 2
```

Notice, however, that the invocations of `withdraw` *are* serviced in order of priority, as are the invocations of `deposit`. (Recall that invocations are normally serviced in order of arrival, but that this order is affected by synchronization and scheduling expressions; see Section 9.5 for details.) Presentation of a solution to priority scheduling over multiple arms is postponed until Section 14.4.

Scheduling and synchronization expressions apply only to the arm on which they are specified and, therefore, cannot be used to implement an invocation selection algorithm that applies to all arms on an `input` statement. To support such selection algorithms, the input statement provides additional features to specify a selection method and to examine pending invocations.

This chapter presents additional features of the input statement that can be used to implement advanced invocation selection algorithms. The presentation begins with a discussion of selection methods and the `view` statement. Next, an overview of some predefined support classes is given. Finally, the chapter concludes with some examples. Additional discussion appears in Reference [32].

## 14.1 Selection Method Expression

To implement an invocation selection algorithm that selects over all arms instead of over a single arm, or that modifies the selection semantics within a single arm, one must specify a *selection method.* A selection method expression is specified for an input statement by a `with/over` clause. Consider the following input statement:

```
inni with selMethod over
   void a(int x) { ... }
[] void b(int y, int z) { ... }
```

This input statement uses a `with/over` clause to specify that the invocation to be serviced should be selected using the `selMethod` method, which the programmer writes. This method may be a normal Java method, an operation, or an expression that evaluates to a capability.

When an input statement with a `with/over` clause is executed, an enumeration of the pending invocations for all operations being serviced by the input statement is passed as an argument to the selection method. From the enumeration, the selection method picks a single invocation to service and returns the selected invocation. As such, the selection method's argument type must be `edu.ucdavis.jr.ArmEnumeration` and its return type must be `edu.ucdavis.jr.Invocation`. If there is no suitable invocation to service, then returning `null` will force the input statement to wait until another invocation arrives. The declaration of the selection method used in the above input statement (with the body elided) is:

```
Invocation selMethod(ArmEnumeration a) {
  ...
}
```

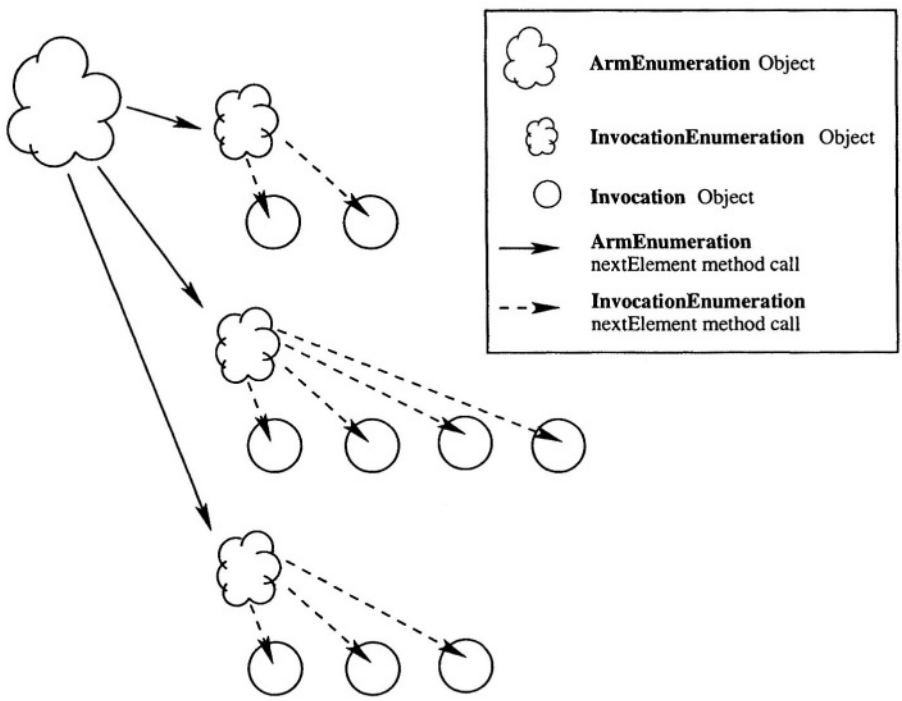


*Figure 14.1.* Pictorial representation of the structure of `ArmEnumeration`

Figure 14.1 portrays the structure of the `ArmEnumeration` argument passed to the selection method. Each `ArmEnumeration` object is associated with a specific input statement. `ArmEnumeration` objects enumerate (through calls to the `nextElement` method) a number of `InvocationEnumeration` objects, one for each arm of the associated input statement, in lexical order of the arms

of the associated input statement. Each `InvocationEnumeration` object, in turn, enumerates (through calls to the `nextElement` method) the `Invocation` objects within a specific arm in order of oldest pending invocation.

The following selection method demonstrates the use of the enumerations:

```
Invocation getFirst(ArmEnumeration armEnum) {
  while (armEnum.hasMoreElements()) {
    InvocationEnumeration invocEnum =
      (InvocationEnumeration) armEnum.nextElement();
    if (invocEnum.hasMoreElements()) {
      return (Invocation) invocEnum.nextElement();
    }
  }
return null;
}
```

This selection method checks each arm for invocations and returns the first invocation found (which need not be the oldest pending). So, for example for the `inni` statement

```
inni with getFirst over
   void one(int x) { ... }
[] void two(int y) { ... }
[] void three(int z) { ... }
```

with the following invocations

```
send three(32);
send two(5);
send one(100);
```

it would return the invocation of one with argument `100`. If executed a second time (assuming no new invocations arrive) it would return the invocation of `two` with argument `5`. Further details of the methods provided by the `ArmEnumeration`, `InvocationEnumeration`, and `Invocation` classes are discussed in Section 14.3.

Selection methods provide access to the entire set of pending invocations during invocation selection. With such access, it is possible to compare individual invocations from different arms to determine which to service. This allows for the implementation of selection algorithms, for example, that enforce priority or that show preference for one arm over another under certain conditions (e.g., give preference to a writer over a reader if the writer is participating in an interactive session). Synchronization and scheduling expressions, however, only provide access to a single invocation at a time.

While a selection method is executing, the queues for the operations in the associated input statement are locked. This locking prevents additional invocations from arriving in the middle of a selection algorithm, which simplifies

the implementation of such algorithms. The locking, however, also prevents any other input statement from servicing one of the locked operations (or any operation in an input statement that even refers to one of the locked operations). As such, selection methods should, when possible, complete quickly.

## 14.2 View Statement

The previous section introduced the use of the `with/over` clause and the implementation of selection methods. The presented features alone, however, are insufficient for implementing the types of selection algorithms discussed in the introduction to this chapter. For example, the priority scheduling selection algorithm requires access to the arguments of the pending invocations. This section presents the `view` statement, which provides access to the arguments of an invocation.

### 14.2.1 General Form and Semantics

A `view` statement is similar, in function, to a `switch` statement; a `view` statement attempts to match an invocation against a number of different cases. The `view` statement differs, however, in that its cases are types rather than values. A `view` statement contains an `Invocation` object and one or more as commands separated by the keyword as:

**`view`** *invocation_expr* **`as`** *formal_list block* **`as`** *formal_list block* ...

The *invocation_expr* is any expression that evaluates to an `Invocation` object. Each `as` command specifies a formal argument list and a block of code. The formal list contains types and names for the parameters, just as in Java's method headers. The values of a matching invocation's arguments are bound to the formal parameters during execution of the associated block of code. After the block completes, execution continues at the first statement following the `view` statement.

A `view` statement evaluates the invocation expression to obtain a reference to a specific `Invocation` object. The types of the arguments for the referenced invocation are then matched against those in the different as commands. If a match is found, then the as command's block is executed with access to the invocation's values through the formal arguments.

The `as` commands in a `view` statement can be followed by an `else` command, which has the form

**`else`** *block*

This block of code is executed if the invocation type does not match a formal argument list in any as command. If no match is found and the `view` statement does not contain an `else` command, then execution continues at the first statement following the `view` statement.

A `view` statement provides a "view" of the contents of an `Invocation` object. Using `view` statements, a selection method can be written that examines and compares the contents of the pending invocations. Once the selection method has determined which invocation to service, it returns that invocation.

### 14.2.2 Simple View Statement

The view statement is specifically intended to provide access to an invocation's arguments within the implementation of a selection method. Typically, a `view` statement will closely mirror the structure of the input statement with which the selection method is associated. As such, consider the following input statement originally discussed above:

```
inni with selMethod over
   void a(int x) { ... }
[] void b(int y, int z) { ... }
```

This input statement services two operations, `a` and `b`, and delegates invocation selection to a selection method, `selMethod`.

A simple `view` statement that may appear in `selMethod` would look as follows:

```
Invocation invocToReturn = null;
view invoc
  as (int m) {
    if (m > 3) // code for a
      invocToReturn = invoc;
  }
  as (int n, int o) {
    if (n > o)
      invocToReturn = invoc;
  }
```

This `view` statement attempts to match the `Invocation` object, `invoc`, against the formal argument lists that correspond to the argument lists of the operations being serviced. A successful match with the first argument list, `(int m)`, indicates a match with an invocation of operation `a`. The value of the first argument of the invocation is bound to parameter `m` during the execution of the block of code labeled "`code for a`". Additional examples of the use of the `with/over` clause, selection methods, and `view` statements are given in Section 14.4.

## 14.3 Selection Method Support Classes

Selection methods make use of `ArmEnumeration`, `InvocationEnumeration`, and `Invocation` objects to implement invocation selection algorithms. Each of the following classes is a member of

the `edu.ucdavis.jr` package. These classes provide methods to facilitate the implementation of different selection algorithms. These methods are summarized below. Detailed descriptions of the methods are provided in Appendix B.

### 14.3.1 ArmEnumeration Methods

The `ArmEnumeration` class provides access to the invocations of an associated input statement through the enumeration of a number of `InvocationEnumeration` objects (one per arm of the input statement). The `ArmEnumeration` class supports the following methods.

| method | description |
|---|---|
| `hasMoreElements` | Tests if this enumeration contains more elements. |
| `nextElement` | Returns the next element of this enumeration. |
| `reset` | Resets the enumeration to the beginning. |
| `size` | Returns the number of elements. |

### 14.3.2 InvocationEnumeration Methods

The `InvocationEnumeration` class provides access to the invocations of a specific arm of an input statement through the enumeration of its pending invocations (represented by `Invocation` objects). The pending invocations are ordered by logical timestamp (oldest pending first). The logical timestamp is an implementation specific data structure that ensures causal ordering of messages. Logical timestamps need not (and do not) correspond to actual time. The `InvocationEnumeration` class supports the following methods.

| method | description |
|---|---|
| `hasMoreElements` | Tests if this enumeration contains more elements. |
| `nextElement` | Returns the next element of this enumeration. |
| `reset` | Resets the enumeration to the beginning. |
| `size` | Returns the number of elements. |

### 14.3.3 Invocation Methods

The `Invocation` class provides access to a single pending invocation of a specific arm of an input statement. The values of the actual arguments within an invocation are accessed using a `view` statement as discussed in Section 14.2. The `Invocation` class supports the following method.

| method | description |
|---|---|
| `getTimestamp` | Returns the logical timestamp of the invocation. |

### 14.3.4 Timestamp Methods

The `Timestamp` class stores a logical timestamp for an invocation. The `Timestamp` class implements `java.lang.Comparable` and supports the following methods.

| method | description |
|---|---|
| `equals` | Tests two `Timestamp` objects for numerical equality. |
| `compareTo` | Compares two `Timestamp` objects numerically. |

## 14.4 Examples

The following examples implement advanced selection algorithms using invocation selection methods and `view` statements. These examples demonstrate the use of a number of the methods discussed for the enumeration and invocation classes.

### 14.4.1 Priority Scheduling

This example revisits the priority scheduling problem presented in the introduction to this chapter. This selection algorithm services the following input statement in order of priority over all invocations.

```
inni with prioritySelect over
   void withdraw(int priority, double amt) { ... }
[] void deposit(int priority, double amt) { ... }
```

To select an invocation based on all pending invocations a `with/over` clause is required and a selection method must be implemented.

The priority scheduling selection algorithm for the above input statement can be implemented as follows.

```
static Invocation prioritySelect(ArmEnumeration armEnum) {
  Invocation cur = null;  int best_prio = 0;

  // iterate over arms
  while (armEnum.hasMoreElements()) {
    InvocationEnumeration invocEnum =
      (InvocationEnumeration)armEnum.nextElement();

    // iterate over invocations for given arm
    while (invocEnum.hasMoreElements()) {
      Invocation invoc = (Invocation)invocEnum.nextElement();

      // determine invocation type and get arguments
      view invoc as (int priority, double amt) {
        if ((cur == null) || (priority < best_prio)) {
          cur = invoc;
          best_prio = priority;
        }
      }
    }
  }
  return cur;
}
```

This selection method iterates through the ArmEnumeration and, in turn, iterates through each InvocationEnumeration to access the pending invocations. The arguments of each pending invocation are accessed through the use of a view statement to allow comparison between an invocation's priority and that of the currently believed highest priority invocation, cur (which is initially null at the beginning). Once all invocations have been examined, cur will refer to the highest priority invocation and it will be returned. If there are no pending invocations, the method returns null and the input statement blocks until an invocation arrives.

Note that the operations being serviced have matching signatures (i.e., both deposit and withdraw take an integer priority and a double amount). Special attention must be paid when writing a selection method for operations with matching signatures. In such cases, it is only possible to distinguish between the invocations of different operations by the order of the InvocationEnumerations as returned by the ArmEnumeration (they are returned in lexical order of the input statement being serviced). In this example, however, such a distinction is unnecessary since selection is based solely on the priority argument.

### 14.4.2 Random Scheduling

Consider, again, the input statement discussed above. In this example, however, the invocation to service will be selected at random rather than according to priority. Selecting an invocation at random, in essence, gives priority to those clients with many pending requests, but still gives those with only a few requests a chance. (The following algorithm can be modified to give a specific request a greater chance for servicing by implementing lottery-scheduling [47].) The following code defines a random scheduling selection method and demonstrates the use of the reset and size methods discussed in Section 14.3.

```
protected static Random rand = new Random();
static op Invocation randomSelect(ArmEnumeration armEnum) {
  int num = 0;

  // iterate over arms and sum number of invocations
  while (armEnum.hasMoreElements()) {
    InvocationEnumeration invocEnum =
      (InvocationEnumeration) armEnum.nextElement();
    num += invocEnum.size();
  }

  int rand = rand.nextInt(num);

  armEnum.reset();
  // iterate over arms
  while (armEnum.hasMoreElements()) {
```

```
    InvocationEnumeration invocEnum =
      (InvocationEnumeration) armEnum.nextElement();
    // determine if the random invocation is in arm
    if (rand >= invocEnum.size())
      rand -= invocEnum.size();
    else {   // it is in this arm
      while (rand > 0) {
        invocEnum.nextElement();
        rand--;
      }
      return (Invocation)invocEnum.nextElement();
    }
  }
  return null;
}
```

This selection method works in two phases. In the first phase, the total number of pending invocations is calculated by iterating through the `ArmEnumeration` and summing the sizes of the individual `InvocationEnumerations` using the `size` method. Once the number of invocations has been tallied a random number in the range from 0 to the total is calculated.

In the second phase, the `ArmEnumeration` is reset and iterated over again until the `InvocationEnumeration` that contains the randomly selected invocation is found. The selected `InvocationEnumeration` is then iterated over until the invocation to service is found and returned.

Notice that the above selection method is an op-method. In this example, there is no difference between defining the selection method as a normal method or as an op-method; it works the same either way. In general, however, using an operation as a selection method allows invocation selection to be implemented by an input statement. Of course, if the input statement servicing the selection operation also services an operation in the original input statement, then deadlock will result due to locking (see Section 14.1), for example, as in the following code:

```
private static process p1 {
  inni with selMethod over
     void a(int x) { ... }
  [] void b(int y, int z) { ... }
}
private static process serving {
                  // selection method
  inni Invocation selMethod(ArmEnumeration armEnum) { ... }
  [] void b(int y, int z) { ... }
}
```

### 14.4.3 Median Scheduling

The final example demonstrates a more complicated selection algorithm in which all invocations must be compared, in a sense, to each other. In this example, the invocation to service is selected according to the median value of the first parameter of the pending invocations. Again, assume the input statement shown above in which there are two operations. The following code implements the median scheduling selection algorithm.

```
static Invocation medianSelect(ArmEnumeration armEnum) {
  Vector v = new Vector();
  // iterate over arms
  while (armEnum.hasMoreElements()) {
    InvocationEnumeration invocEnum =
      (InvocationEnumeration) armEnum.nextElement();
    // iterate over invocations for a given arm
    while (invocEnum.hasMoreElements()) {
      Invocation invoc = (Invocation) invocEnum.nextElement();
      // determine type of given invocation and get arguments
      view invoc
      as (int val) {
        v.add(new Element(invoc, val));
      }
      as (int val, int i) {
        v.add(new Element(invoc, val));
      }
    }
  }

  Object invocations [];
  invocations = v.toArray();
  if (invocations.length == 0) return null;

  // select the median invocation from the array and return it
  Arrays.sort(invocations, Element.comparator);
  return ((Element)invocations[invocations.length/2]).getInvocation();
}
static class Element {
  private Invocation invoc;
  private int i;

  public static final Comparator comparator = new Comparator() {
    public int compare(Object o1, Object o2) {
      return ((Element)o1).i - ((Element)o2).i;
    }

    public boolean equals(Object o1, Object o2) {
      return o1.equals(o2);
    }
```

```
  };

  public Element(Invocation invoc, int i) {
    this.invoc = invoc;
    this.i = i;
  }

  public Invocation getInvocation() {
    return invoc;
  }

  public boolean equals(Object o2) {
    return ((o2 instanceof Element) &&
            (this.i == ((Element)o2).i) &&
            (this.invoc.equals(((Element)o2).invoc)));
  }
}
```

The selection method first iterates over all of the invocations and creates an `Element` object containing each invocation and the invocation's first argument. As the `Element` objects are created, they are stored in a `Vector`. Once all of the invocations have been gathered, the vector is converted into an array and sorted (using the `Comparator` object in the `Element` class). Finally, the median invocation is selected and returned for servicing.

## Exercises

14.1 Write a selection method to return

(a) the first invocation of the lexically last arm that has one.

(b) the last invocation of the lexically first arm that has one.

14.2 The following input statement uses a synchronization (`st`) expression to select which invocations to service.

```
inni void a(int x) st x > 3 { ... }
```

Using a `with/over` clause instead of the synchronization expression, write a selection method that simulates the above `st` expression (i.e., that selects the same invocations to service in the same order).

14.3 The following input statement uses a scheduling (`by`) expression to alter the order in which invocations are serviced.

```
inni void a(double x) by Math.sin(x) { ... }
```

Using a `with/over` clause instead of the scheduling expression, write a selection method that simulates the above by expression (i.e., that selects the same invocations to service in the same order).

14.4 This exercise asks that you write a selection method that simulates the default JR invocation selection semantics for a specific input statement. JR's invocation selection semantics are as follows (see Section 9.5 for further discussion):

(a) Select the arm with the oldest pending invocation.

(b) Search the selected arm for a serviceable invocation (i.e., for which the synchronization expression is true).

(c) If this arm contains a scheduling expression, then continue searching this arm for a serviceable invocation that minimizes the scheduling expression.

(d) If no serviceable invocation is found, then select the arm with the next oldest pending invocation and repeat the process. If there are no unchecked arms, execute the else arm (if present) or block.

Write a selection method that simulates JR's semantics for this input statement:

```
inni void a(int x) st x > 3 by -x {
  System.out.println("a: " + x);
}
[] void b(int x, int y) st (x + y) < 10 by y {
  System.out.println("b: " + x + ":" + y);
}
```

Just to be clear, for the following set of asynchronous invocations

```
send a(2);
send b(3, 4);
send a(11);
send b(13, -5);
send a(444444);
```

the output of repeated executions of the above input statement (assuming that all invocations had already arrived before executing the input statement) is:

```
a: 444444
a: 11
b: 13:-5
b: 3:4
```

14.5 This exercise asks that you repeat Exercise 14.4 but write a selection method that simulates the default SR invocation selection semantics instead of JR's semantics. SR's invocation selection semantics are as follows [9]:

(a) Examine the invocations in order of oldest pending.

(b) Find the oldest pending serviceable invocation (i.e., for which the synchronization expression is true).

(c) If the arm containing the oldest pending serviceable invocation has a scheduling expression, then continue searching that arm for a serviceable invocation that minimizes the scheduling expression.

(d) If no serviceable invocation is found, then execute the else arm (if present) or block.

Just to be clear, for the following set of asynchronous invocations

```
send a(2);
send b(3, 4);
send a(11);
send b(13, -5);
send a(444444);
```

the output of repeated executions of the above input statement (assuming that all invocations had already arrived before executing) is:

```
b 13 -5
b 3 4
a 444444
a 11
```

14.6 Synchronization and scheduling clauses can access not only the parameters of an invocation, but also variables in the local scope (i.e., local, instance, and class variables). Selection methods, by virtue of being invoked as a method, do not have access to variables local to the scope in which the associated input statement resides. Furthermore, selection methods may not have access to the same set of instance and class variables.

Discuss the potential implications of these differing access rights. Is it possible to provide a selection method with access to the same set of variables? If so, how?

14.7 Discuss implementing a general selection method (or an object with a selection method) that can be used to replace the use of the built-in synchronization and scheduling expressions. Consider passing capabilities to the object's constructor. As a first step, verify that your solution works on the examples in Exercises 14.2 and 14.3. Next, discuss any limitations relating to Exercise 14.6.

*This page intentionally left blank*

PART II

# APPLICATIONS

Part I described JR's concurrent programming mechanisms and gave numerous, mostly small examples. In this part we examine several larger applications. These are representative of the kinds of parallel and distributed programming problems JR can be used to solve. The solutions also illustrate several process interaction paradigms that occur in concurrent programs. Each interaction paradigm is an example or model of an interprocess communication pattern and associated programming technique that can be used to solve a variety of problems. The exercises at the end of each chapter explore additional problems that can be solved using these paradigms.

Chapters 15 and 16 examine two problems that are representative of those that arise in scientific computing: matrix multiplication and iterative solutions to partial differential equations (PDEs). In both cases we develop solutions that use shared variables and ones that use message passing. We also discuss performance issues, including the effect of memory caches and the tradeoffs between task size and communication and synchronization overhead. In Chapter 17 we examine the classic traveling salesman problem, which is representative of combinatorial problems. Again we present both centralized and distributed solutions and discuss performance tradeoffs. The next two chapters in Part II present further examples of concurrent programs. Chapter 18 develops a simple command interpreter and distributed file system. Chapter 19 develops an implementation of a discrete event simulation package. Chapter 20 describes how JR programs can interact with the Java GUI (graphical user interface) packages AWT and Swing. Finally, Chapter 21 describes other concurrency notations, which preprocessors convert into JR programs.

*This page intentionally left blank*

Chapter 15

# PARALLEL MATRIX MULTIPLICATION

Matrix computations lie at the heart of most scientific computing problems. Matrix multiplication is one of the most basic of these computations. In Chapter 1 we presented a simple, but inefficient, parallel algorithm for matrix multiplication. Here we develop four realistic algorithms. Two employ shared variables and hence are suitable for execution on shared-memory multiprocessors. The other two algorithms employ message passing and hence are suitable for execution on distributed-memory systems. Each algorithm also illustrates a different programming technique and a different combination of JR mechanisms.

As in Section 1.3, the problem is to compute the product of two $N \times N$ real matrices $A$ and $B$. This requires computing $N^2$ inner products, one for each combination of a row of $A$ and a column of $B$. On a massively parallel, synchronous multiprocessor, all inner products could be computed in parallel with reasonable efficiency since, by default, every processor executes the same sequence of instructions at the same time. However, on an asynchronous multiprocessor each process has to be created and destroyed explicitly, and each inner product requires relatively little computation. In fact, the parallel program in Section 1.3 would be much slower than a sequential program since the cost of creating and destroying processes would far outweigh any benefits derived from parallel execution.

To execute efficiently on an asynchronous multiprocessor, each process in a parallel program must perform quite a bit of work relative to the amount of time it takes to create the process and the amount of time the process spends communicating and synchronizing with other processes. A common way to describe the amount of sequential work that a process performs is in terms of the number of basic steps—or *grains*—of the parallel computation. Choosing an appropriate grain size is a ubiquitous and important problem in parallel computing because

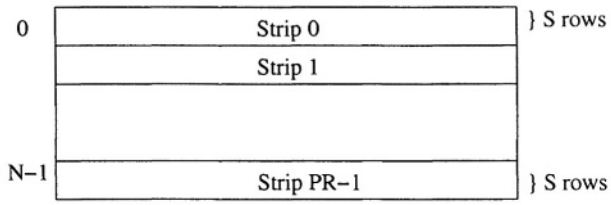


*Figure 15.1.* Assigning processes to strips

the grain size determines each process's sequential execution time, which must be much greater than the concurrency and communication overhead. The exact balance depends, of course, on the underlying hardware and on the concurrent programming mechanisms that are employed. This chapter develops four matrix multiplication algorithms that employ different combinations of communication and synchronization mechanisms. Each can readily be modified to alter the balance between sequential execution time and concurrency overhead.

## 15.1 Prescheduled Strips

Our first algorithm uses `N` × `N` real matrices `A`, `B`, and `C`. Assume that these are shared variables, and we wish to use `PR` processes to compute the product of `A` and `B` and store it in `C`. For simplicity we also assume that `N` is a multiple of `PR`; for example, `N` might be 100 and `PR` might be 10.

To balance the amount of computation performed by each process, each should compute $N^2/PR$ inner products. The simplest way to do this is to assign each process responsibility for computing the values for all elements in a strip of matrix `C`, as shown in Figure 15.1. In particular, let `S` be `N/PR`. Then the first process computes the values of the first `S` rows of `C`, the second computes the values of the next `S` rows of `C`, and so on. This kind of approach is sometimes called prescheduling because each process is assigned in advance a certain number of "chores," i.e., inner products in this case.

To implement this algorithm in JR, we use a main class and a multiplier class. The main class reads values for `N`, `PR`, `A`, and `B` from input files or the command line (straightforward code not shown). It creates a multiplier object and then calls the object's `compute` method, which does the actual computation and returns the results. The main class then outputs the results.

```
public class MMMain {
  public static void main(String [] args) {
    int N;  // A and B are NxN
    int PR; // number of processes
    double [][] A, B;
    // read in NxN arrays A and B and value for PR
```

```
    ...
    MMMultiplier m = new MMMultiplier(PR);
    double [][] C = m.compute(A, B, N);
    print(C);
  }
  private static void print(double [][] C) {
    // output C
    for (int r = 0; r < C.length; r++) {
      for (int c = 0; c < C[0].length; c++) {
        System.out.print(C[r][c]+" ");
      }
      System.out.println();
    }
  }
}
```

The multiplier class declares `N`, `PR`, `S`, and the matrices `A`, `B`, and `C`. Its `compute` method creates `PR` processes to compute the inner products. These processes share the three matrices, but each `strip` process computes a separate strip of `C` so no synchronization is needed. The `compute` method starts up these processes and then waits, using the `done` semaphore, for all processes to finish computing their strips.

```
public class MMMultiplier {
  int N;  // A and B are NxN
  int PR; // number of processes
  int S;  // strip size
  double [][] A, B, C;
  public MMMultiplier(int PR) {
    this.PR = PR;
  }
  sem done;
  public double [][] compute(double [][] A,
                             double [][] B, int N) {
    this.A = A;
    this.B = B;
    this.N = N;
    S = N/PR;
    C = new double [N][N];
    // start PR processes
    for (int p = 0; p < PR; p++) {
      send strip(p);
    }
    // wait for all to complete
    for (int p = 0; p < PR; p++) {
      P(done);
    }
    return C;
  }
  // one strip process for each strip of C.
```

```
  private op void strip(int p) {
    final int R = p * S; // starting row
    // compute S*N inner products
    for (int r = R; r < R+S; r++) {
      for (int c = 0; c < N; c++) {
        double inner_prod = 0.0; // local accumulator
        for (int k = 0; k < N; k++) {
          inner_prod += A[r][k] * B[k][c];
        }
        C[r][c] = inner_prod;
      }
    }
    V(done);
  }
}
```

The `compute` method creates the `strip` processes using send invocations and then waits for them to complete using a semaphore. That code could not be replaced (only) by declaring the equivalent family of processes (using the `process` abbreviation) because those processes might execute before the code in the `compute` method initializes instance variables used within `strip`. (See Exercise 15.3.)

Many shared-memory multiprocessors employ caches, with one cache per processor. Each cache contains the memory blocks most recently referenced by the processor. (A block is typically a few contiguous words.) The purpose of caches is to increase performance, but they have to be used with care by the programmer or they can actually decrease performance (due to cache conflicts). Reference [22] gives three rules-of-thumb programmers need to keep in mind:

- Perform all operations on a variable, especially updates, in one process.
- Align data so that variables updated by different processors are in different cache blocks.
- Reuse data quickly when possible so it remains in the cache and does not get "spilled" back to main memory.

A two-dimensional array in Java is an array of references to single-dimensional arrays. So, a matrix is stored in row-major order (i.e., by rows), although adjacent rows are not necessarily contiguous. The above program, therefore, uses caches well. Each `strip` process reads one distinct strip of `A` and writes one distinct strip of `C`, and it references elements of `A` and `C` by sweeping across rows. Every process references all elements of `B`, but that is unavoidable. (If `B` were transposed, so that columns were actually stored in rows, it too could be referenced efficiently.)

## 15.2 Dynamic Scheduling: A Bag of Tasks

The algorithm in the previous section statically assigned an equal amount of work to each `strip` process. If the processes execute on homogeneous processors without interruption, they would be likely to finish at about the same time. However, if the processes execute on different-speed processors, or if they can be interrupted—e.g., in a timesharing system—then different processes might complete at different times. To dynamically assign work to processes, we can employ a shared bag of tasks, as in the solution to the adaptive quadrature problem in Section 7.7. Here we present a matrix multiplication program that implements such a solution. The structure of the solution is illustrated in Figure 15.2.

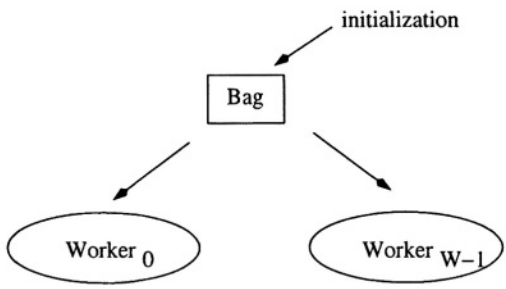


*Figure 15.2.* Replicated workers and bag of tasks

As in the previous program, we employ two classes. The main class is identical to that in the previous section: it again creates a multiplier object, calls the object's `compute` method, and then prints out results.

The multiplier class is similar to that in the previous section in that it declares `N`, `A`, and `B`. It also declares and initializes `W`, the number of worker processes. The class declares an operation, `bag`, which is shared by the `worker` processes.

The code in method `compute` sends each row index to `bag`. It then creates the worker processes, waits for them to terminate, and returns results to the invoker. Each worker process repeatedly receives a row index `r` from `bag` and computes `N` inner products, one for each element of row `r` of result matrix `C`. However, if the bag is empty, then the worker process notifies the compute method that it has completed and terminates itself. (See Exercises 15.5 and 15.6.)

```
public class MMMultiplier {
  int N;  // A and B are NxN
  int W;  // number of worker processes
  op void bag(int);
  double [][] A, B, C;
  public MMMultiplier(int PR) {
    W = PR;
  }
```

```
  sem done;
  public double [][] compute(double [][] A,
                             double [][] B, int N) {
    this.A = A;
    this.B = B;
    this.N = N;
    C = new double [N][N];
    // initialize bag of tasks
    for (int r = 0; r < N; r++) {
      send bag(r);
    }
    // start W worker processes
    for (int p = 0; p < W; p++) {
      send worker();
    }
    // wait for all workers to complete
    for (int p = 0; p < W; p++) {
      P(done);
    }
    return C;
  }
  private op void worker () { // worker process
    while (true) {
      inni void bag(int r) { // compute r-th row of C
        for (int c = 0; c < N; c++) {
          double inner_prod = 0.0; // local accumulator
          for (int k = 0; k < N; k++) {
            inner_prod += A[r][k] * B[k][c];
          }
          C[r][c] = inner_prod;
        }
      }
      [] else {
        break;
      }
    }
    V(done);
  }
}
```

This way of detecting when to terminate works here because once the bag becomes empty, no new tasks are added to it; this way would not work for other problems where the bag might become empty before additional tasks are placed into it. For examples, see the adaptive quadrature example in Section 7.7, and the two solutions to the traveling salesman problem in Sections 17.2 and 17.3.

This program should show nearly perfect speedup—over the one worker and one processor case—for reasonable-size matrices, e.g., when `N` is 100 or more. In this case the amount of computation per iteration of a worker process far outweighs the overhead of receiving a message from the bag. Like the previous

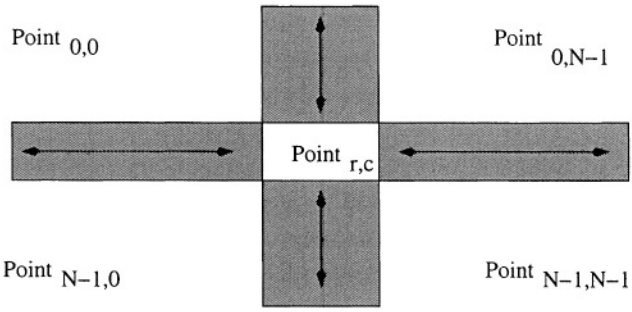


*Figure 15.3.* Broadcast algorithm interaction pattern

program, this one uses caches well since JR stores matrices in row-major order, and each worker fills in an entire row of `C`. If the bag of tasks contained column indices instead of row indices, performance would be much worse because workers would encounter cache update conflicts.

## 15.3 A Distributed Broadcast Algorithm

The program in the previous section can be modified so that the workers do not share the matrices or bag of tasks. In particular, each worker (or address space) could be given a copy of `A` and `B`, and an administrator process could dispense tasks and collect results (see Exercise 15.4). With these changes, the program could execute on a distributed-memory machine.

This section and the next present two additional distributed algorithms. To simplify the presentation, we use $N^2$ processes, one to compute each element `C[r] [c]`. Initially each such process also has the corresponding values of `A` and `B`, i.e., `A[r] [c]` and `B [r] [c]`. In this section we have each process broadcast its value of `A` to other processes on the same row and broadcast its value of `B` to other processes on the same column. In the next section we have each process interact only with its four neighbors. Both algorithms are inefficient as given since the grain size is way too small to compensate for communication overhead. However, the algorithms can readily be generalized to use fewer processes, each of which is responsible for a block of matrix `C` (see Exercises 15.11 and 15.12).

Our broadcast implementation of matrix multiplication uses three classes: a main class, a multiplier class, and a point class. The main class is identical to those in the previous sections.

Instances of class `Point` carry out the computation. The multiplier class creates one instance for each value of `C[r][c]`. Each instance provides three public operations: one to start the computation, one to exchange row values, and one to exchange column values. Operation `compute` is serviced by a method; it is invoked by a send statement in the multiplier class and hence executes as a process. The arguments of the `compute` operation are references for other

instances of Point. Operations `rowval` and `colval` are serviced by receive statements; they are invoked by other instances of `Point` in the same row `r` and column `c`, respectively.

The $N^2$ instances of `Point` interact as shown in Figure 15.3. The `compute` process in `Point` first sends its value of `Arc` to the other instances of `Point` in the same row and receives their elements of `A`. The `compute` process then sends its value of `Brc` to other instances of `Point` in the same column and receives their elements of `B`. After these two data exchanges, `Point (r, c)` now has row `r` of `A` and column `c` of `B`. It then computes the inner product of these two vectors. Finally, it sends its value of `Crc` back to the multiplier class.

```
public class Point {
  public op void compute(remote Point [], remote Point []);
  public op void rowval(int, double);
  public op void colval(int, double);
  private int N, r, c;
  private double Arc, Brc, Crc = 0.0;
  private cap void (int, int, double) gather;
  public Point(int N, int r, int c, double Arc, double Brc,
               cap void(int, int, double) gather ) {
    this.N = N;
    this.r = r;
    this.c = c;
    this.Arc = Arc;
    this.Brc = Brc;
    this.gather = gather;
  }
  public void compute(remote Point [] rlinks,
                      remote Point [] clinks) {
    double [] row = new double [N];
    double [] col = new double [N];
    int sender;
    row[c] = Arc;
    col[r] = Brc;
    // broadcast Arc to points on same row
    for (int k = 0; k < N; k++) {
      if (k != c) {
        send rlinks[k].rowval(c, Arc);
      }
    }
    // acquire other point from same row
    for (int k = 0; k < N; k++) {
      if (k != c) {
        receive rowval(sender, row[sender]);
      }
    }
    // broadcast Brc to points on same col
    for (int k = 0; k < N; k++) {
```

```
      if (k != r) {
        send clinks[k].colval(r, Brc);
      }
    }
    // acquire other point from same col
    for (int k = 0; k < N; k++) {
      if (k != r) {
        receive colval(sender, col[sender]);
      }
    }
    // compute inner product of row and col
    for (int k = 0; k < N; k++) {
      Crc += row[k] * col[k];
    }
    // send back result to invoker
    send gather(r, c, Crc);
  }
}
```

The multiplier class creates $N^2$ instances of `Point` and gets back a reference for each, which it stores in matrix `pref`. It then invokes the `compute` operations, passing each instance of `Point` references for other instances in the same row and column. We use `pref[r]` to pass row `r` of `pref` to `compute`. But, we must extract the elements in column `c` of `pref` and store them in a new array, `cpref`, which we then pass to `compute`. It then waits for all points to finish their computations and gathers the results, which it returns to the invoker.

```
public class MMMultiplier {
  public double [][] compute(double [][] A,
                             double [][] B, int N) {
    op void gather(int, int, double);
    double [][] C = new double [N][N]; // result
    remote Point [][] pref = new remote Point [N][N];
    // create points
    for (int r = 0; r < N; r++) {
      for (int c = 0; c < N; c++) {
        pref[r][c] =
          new remote Point(N, r, c, A[r][c], B[r][c], gather);
      }
    }
    // give each point references for its neighbors
    for (int r = 0; r < N; r++) {
      for (int c = 0; c < N; c++) {
        // cpref is column c of pref
        remote Point [] cpref = new remote Point[N];
        for (int k = 0; k < N; k++) {
          cpref[k] = pref[k][c];
        }
        send pref[r][c].compute(pref[r], cpref);
```

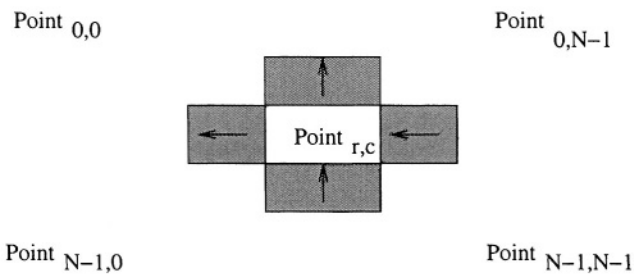


*Figure 15.4.* Heartbeat algorithm interaction pattern

```
      }
    }
    // gather results from all points
    for (int k = 1; k <= N*N; k++) {
      int r, c;
      receive gather(r, c, C[r][c]);
    }
    return C;
  }
}
```

As noted, this program can readily be modified to have each instance of `Point` start with a block of `A` and a block of `B` and compute all elements of a block of `C`. It also can be modified so that the blocks are not square, i.e., strips can be used. In either case the basic algorithmic structure and communication pattern is identical. The program can also be modified to execute on multiple virtual machines: The multiplier class first creates the virtual machines and then creates instances of `Point` on them.

## 15.4 A Distributed Heartbeat Algorithm

In the broadcast algorithm, each instance of `Point` acquires an entire row of `A` and an entire column of `B` and then computes their inner product. Also, each instance of `Point` communicates with all other instances on the same row and same column. Here we present a matrix multiplication algorithm that employs the same number of instances of a `Point` class. However, each instance holds only one value of `A` and one of `B` at a time. Also, each instance of `Point` communicates only with its four neighbors, as shown in Figure 15.4. Again, the algorithm can readily be generalized to work on blocks of points and to execute on multiple virtual machines.

As in the broadcast algorithm, we will use $\mathtt{N}^2$ processes, one to compute each element of matrix `C`. Again, each initially also has the corresponding elements of `A` and `B`. The algorithm consists of three stages [37]. In the first stage, processes shift values in `A` circularly to the left; values in row `r` of `A` are shifted left `r`

columns. Second, processes shift values in B circularly up; values in column c of B are shifted up c rows. The result of the initial rearrangement of the values of A and B for a 3 × 3 matrix is shown in Figure 15.5. (Other initial

```
a[0][0], b[0][0]   a[0][1], b[1][1]   a[0][2], b[2][2]

a[1][1], b[1][0]   a[1][2], b[2][1]   a[1][0], b[0][2]

a[2][2], b[2][0]   a[2][0], b[0][1]   a[2][1], b[1][2]
```

*Figure 15.5.* Initial rearrangement of 3 × 3 matrices A and B

rearrangements are possible; see Exercise 15.9.) In the third stage, each process multiplies one element of A and one of B, adds the product to its element of C, shifts the element of A circularly left one column, and shifts the element of B circularly up one row. This compute-and-shift sequence is repeated N-1 times, at which point the matrix product has been computed.

We call this kind of algorithm a *heartbeat algorithm* because the actions of each process are like the beating of a heart: first send data out to neighbors, then bring data in from neighbors and use it. To implement the algorithm in JR, we again use three classes, as in the broadcast algorithm. Once again, the main class is identical to those in the previous sections.

The computation is carried out by $N^2$ instances of a Point class, which provides three public operations as in the broadcast algorithm. However, here the compute operation is passed references for only the left and upward neighbors, and the rowval and colval operations are invoked by only one neighbor. Also, the body of Point implements a different algorithm, as seen in the following.

```
public class Point {
  public op void compute(remote Point, remote Point);
  public op void rowval(double);
  public op void colval(double);
  private int N, r, c;
  private double Arc, Brc, Crc;
  private cap void(int, int, double) gather;
  public Point(int N, int r, int c, double Arc, double Brc,
               cap void(int, int, double) gather ) {
    this.N = N;
    this.r = r;
    this.c = c;
    this.Arc = Arc;
    this.Brc = Brc;
    this.gather = gather;
  }
  public void compute(remote Point left,
```

```
                    remote Point up) {
    // shift values in Arc circularly left r columns
    for (int k = 0; k < r; k++) {
      send left.rowval(Arc); receive rowval(Arc);
    }
    // shift values in Brc circularly up c rows
    for (int k = 0; k < c; k++) {
      send up.colval(Brc); receive colval(Brc);
    }
    Crc = Arc * Brc;
    for (int k = 1; k <= N-1; k++) {
      // shift Arc left, Brc up, then multiply
      send left.rowval(Arc); send up.colval(Brc);
      receive rowval(Arc); receive colval(Brc);
      Crc += Arc * Brc;
    }
    // send back result to invoker
    send gather(r, c, Crc);
  }
}
```

Method `compute` in the multiplier class creates instances of `Point` and passes each references for its left and upward neighbors. The `compute` method starts up the computation in the `Point` objects and gathers the results from all the points.

```
public class MMMultiplier {
  public double [][] compute(double [][] A,
                             double [][] B, int N) {
    op void gather(int, int, double);
    double [][] C = new double [N][N]; // result
    remote Point [][] pref = new remote Point [N][N];
    pref = new remote Point [N][N];
    // create points
    for (int r = 0; r < N; r++) {
      for (int c = 0; c < N; c++) {
        pref[r][c] =
          new remote Point(N, r, c, A[r][c], B[r][c], gather);
      }
    }
    // give each point references for
    // its left and upward neighbors
    for (int r = 0; r < N; r++) {
      for (int c = 0; c < N; c++) {
        send pref[r][c].compute(pref[r][prev(c,N)],
                                pref[prev(r,N)][c]);
      }
    }
    // gather results from all points
    for (int k = 1; k <= N*N; k++) {
```

```
      int r, c;
      receive gather(r, c, C[r][c]);
    }
    return C;
  }
  private int prev(int k, int N) {
    return (k-1+N)%N; // +N prevents negative results from %
  }
}
```

The `prev` method uses modular arithmetic so that instances of `Point` on the left and top borders communicate with instances on the right and bottom borders, respectively.

## Exercises

15.1 Determine the execution times of the programs in this chapter. To do so, place an invocation of `System.currentTimeMillis()` just before the computation begins and another just after the computation completes. The difference between the two values returned by this method is the time, in milliseconds, that the JR program has been executing.

15.2 Modify the prescheduled strip algorithm so that `N` does not have to be a multiple of `PR`.

15.3 Rewrite the `MMMultiplier` class in Section 15.1 so that the `strip` processes are declared as a family of processes using the `process` abbreviation. Be sure your solution prevents the potential problem mentioned in the text; i.e., it prevents these processes from executing before instance variables have been initialized.

15.4 Change the bag of tasks program so that it does not use shared variables.

15.5 Suppose we change the code in the `MMMultiplier` class in Section 15.2 so that the compute method does not create the processes. Instead they are created using the `process` abbreviation:

```
private process worker ( (int p = 0; p < W; p++) ) {
```

Is this new program correct?

15.6 Suppose we change the code in the `MMMultiplier` class in Section 15.2 so that the worker process executes the following code

```
private op void worker () { // worker process
  while (bag.length() > 0) {
    int r;
```

```
        receive bag(r);
        // compute r-th row of C
        // same code as in original
        ...
      }
      V(done);
    }
```

Is this new program correct?

15.7 The `compute` process in class `Point` in Section 15.3. contains the following receive statement:

```
receive rowval(sender, row[sender]);
```

This statement is within a for statement.

(a) Write an equivalent input statement for the receive statement.

(b) Explain why the receive statement cannot be simplified to the following, assuming the declaration of `rowval` is changed to omit the `sender` field:

```
receive rowval(row[k]);
```

15.8 Suppose `A` and `B` are $5 \times 5$ matrices. Determine the location of each value of `A` and `B` after the two shift stages of the heartbeat algorithm in Section 15.4.

15.9 Reconsider the initial rearrangement phase of the heartbeat algorithm in Figure 15.5. Suppose instead that each row `r` of `A` is shifted left `r+1` columns and that each column `c` of `B` is shifted `up c+1` rows. Show this initial rearrangement for when `A` and `B` are $3 \times 3$ matrices. Will the heartbeat algorithm still multiply arrays correctly?

Now suppose that each row `r` of `A` is shifted left `r+2` columns and that each column `c` of `B` is shifted up `c+2` rows. Repeat the above questions,

If possible, generalize the above results.

15.10 Determine the total number of messages that are sent in the distributed broadcast algorithm and the size of each. Do the same for the distributed heartbeat algorithm. Explain the differences.

15.11 Modify the broadcast algorithm so that each instance of `Point` is responsible for a block of points. Use $\mathtt{PR}^2$ processes, where `N` is a multiple of `PR`.

15.12 Modify the heartbeat algorithm so that each instance of `Point` is responsible for a block of points. Use $PR^2$ processes, where `N` is a multiple of `PR`.

15.13 Compare the performance of the various programs presented in this chapter or those that you developed in answering the above exercises.

15.14 Implement matrix multiplication using a grid of filter processes [26, 7].

15.15 Implement Gaussian elimination (see Exercise 4.17) using the techniques illustrated in this chapter.

*This page intentionally left blank*

Chapter 16

# SOLVING PDEs: GRID COMPUTATIONS

Partial differential equations (PDEs) are used to model a variety of different kinds of physical systems: weather, airflow over a wing, turbulence in fluids, and so on. Some simple PDEs can be solved directly, but in general it is necessary to approximate the solution at a finite number of points using iterative numerical methods. In this chapter we show how to solve one specific PDE—Laplace's equation in two dimensions—by means of a grid computation, which employs what is called a finite-difference method. As in the previous chapter, we present several solutions that illustrate a variety of programming techniques and their realizations in JR.

Laplace's equation is an example of what is called an elliptic partial differential equation. The equation for two dimensions is the following:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

Function $\phi$ represents some unknown potential, such as heat or stress.

Given a fixed spatial region and solution values for points on the boundaries of the region, our task is to approximate the steady-state solution for points within the interior. We can do this by covering the region with an evenly spaced grid of points, as shown in Figure 16.1. Each interior point is initialized to some value. The steady-state values of the interior points are then calculated by repeated iterations. On each iteration the new value of a point is set to a combination of the old and/or new values of neighboring points. The computation terminates when every new value is within some acceptable difference $\epsilon$ of every old value.

There are several stationary iterative methods for solving Laplace's equation—Jacobi iteration, Gauss-Seidel, and successive over-relaxation (SOR). In Jacobi iteration, the new value for each point is set to the average of the old values of the four neighboring points. Jacobi iteration can be paral-

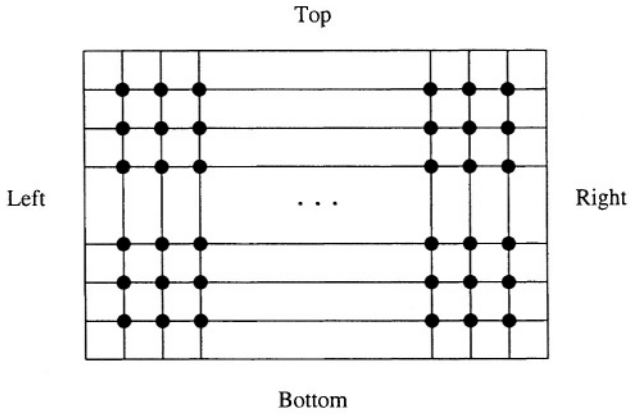


*Figure 16.1.* Approximating Laplace's equation using a grid

lelized readily because each new value is independent of the others. Although Jacobi iteration converges more slowly than other methods, we will use it in this chapter since it is easier to program. In any event, parallel computations that use other iterative methods employ basically the same communication and synchronization patterns.

## 16.1 A Data Parallel Algorithm

A data parallel algorithm is an iterative algorithm that repeatedly and in parallel manipulates a shared array [23]. This kind of algorithm is most closely associated with synchronous (SIMD) multiprocessors, but it can also be used on asynchronous multiprocessors. Here we present a data parallel implementation of Jacobi iteration.

The main loop in a data parallel implementation of Jacobi iteration repeatedly executes three phases:

```
while (true) {
  compute new values for all grid points
  replace old values by new values
  check for convergence; exit loop if converged
}
```

To implement this computation in JR, we use a main class, a Jacobi class, and a results class.

The results class is the simplest. It is simply a container for the results of the computation; it provides a `print` method to `print` out the results.

```
public class Results {
  // return info from approximating solution to PDE
  private int iterations; // number of iterations
  private double [][] matrix; // computed grid
```

```
  public Results(int iterations, double [][] matrix) {
    this.iterations = iterations;
    this.matrix = matrix;
  }
  public void print() {
   System.out.println("convergence after " +
                      iterations + " iterations");
    for( int r = 1; r <= matrix.length-2; r++ ) {
      for( int c = 1; c <= matrix[0].length-2; c++ ) {
        System.out.print(matrix[r][c] + " ");
     }
     System.out.println();
   }
  }
}
```

The main class declares the grid size `N`, border values (left, top, right, and bottom), and the convergence criterion `epsilon`. `N` is the number of rows and columns in the grid of interior points, i.e., points whose steady-state value is to be computed. The main method reads these values from input or as command-line arguments (not shown in the code). It then creates a `Jacobi` object, which will be used for the actual computation. The main method then invokes the `compute` in the `Jacobi` object, gets back the results, and prints them out.

```
public class Main {
  private static int N;
  private static double xl, xt, xr, xb, epsilon;
  public static void main(String [] args) {
    // read N, xl, xt, xr, xb, and epsilon
    // as input or command-line args
    ...
    Jacobi j = new Jacobi();
    Results results = j.compute(N, xl, xt, xr, xb, epsilon);
    results.print();
  }
}
```

The Jacobi class provides the `compute` method. This method is passed the grid size `N`, border values (left, top, right, and bottom), and the convergence criterion `epsilon`. It initializes an array that contains old and new grid values and two variables that are used to index `grid`. The current (old) grid values are `grid[cur]` and the next (new) grid values are `grid[nxt]`. The code later in this section reads values from the old grid and writes values to the new grid on each iteration. At the end of each iteration, the code replaces the old values by the new values by simply swapping `nxt` and `cur`, which is more efficient than copying `grid[nxt]` to `grid[cur]` element by element. (Exercise 16.5 explores this issue in further detail.)

Array `grid` consists of two matrices. Each matrix has `N+2` rows and columns so the boundary values can be stored within the matrix. This avoids having to treat the boundaries as special cases in the main computation. For simplicity each interior grid point is initialized to zero; for faster convergence each should be initialized to a value that somewhat approximates the expected final value.

After initializing the grid, the `compute` method performs the actual iterative computation. It initializes the array `diff`, which is used to record differences for each point between successive iterations. It then invokes the `compute` method. The main loop in `compute` has three phases, as outlined above. The first phase is implemented by a `co` statement that makes $N^2$ calls of `update`. The second phase is implemented by swapping the two indices, `which` switches the roles of the two grids. The third phase is implemented by a second `co` statement that makes `N` calls of `check_diff`s and by an if statement that exits the loop if the grid values have converged.

```
public class Jacobi {
  private int N;
  private double grid[][][];
  private int cur = 0, nxt = 1;
  private double [][] diff;
  private sem done;
  public Results compute(int N, double xl, double xt,
                         double xr, double xb, double epsilon) {
    this.N = N;
    grid = new double[2][N+2][N+2];
    // initialize grid
    for( int g = 0; g <= 1; g++ ) {
      for( int r = 0; r <= N+1; r++ ) {
        grid[g][r][0] = xl;
        grid[g][r][N+1] = xr;
        for( int c = 1; c <= N; c++ ) {
          grid[g][r][c] = 0.0;
        }
      }
    }
    // top row
    for( int c = 0; c <= N+1; c++ ) {
      grid[cur][0][c] = grid[nxt][0][c] = xt;
    }
    // bottom row
    for( int c = 0; c <= N+1; c++ ) {
      grid[cur][N+1][c] = grid[nxt][N+1][c] = xb;
    }

    // diff is N+2 by N+2 to be consistent with grid;
    // 0, N+1 rows and cols are not used.
    diff = new double [N+2][N+2];
    int iters = 0;
```

```
    while (true) {
      iters++;
      // compute new values for all grid points
      // start N*N processes
      for (int r = 1; r <= N; r++ ) {
        for (int c = 1; c <= N; c++ ) {
          send update(r,c);
        }
      }
      // wait for all to complete
     for (int r = 1; r <= N; r++ ) {
       for (int c = 1; c <= N; c++ ) {
         P(done);
       }
     }
     // replace old values by new values
     int tmp = cur;
     cur = nxt;
     nxt = tmp;
     // check for convergence; exit loop if converged
     double maxdiff = 0.0;
     // start N processes
     for (int r = 1; r <= N; r++ ) {
       send check_diffs(r);
     }
     // wait for all to complete
     for (int r = 1; r <= N; r++ ) {
       P(done);
     }
     for (int r = 1; r <= N; r++ ) {
       maxdiff = Math.max(maxdiff, diff[r][1]);
     }
     if (maxdiff <= epsilon) break;
   }
   return new Results(iters, grid[cur]);
  }
  private op void update(int r, int c) {
    grid[nxt][r][c] =
                      ( grid[cur][r-1][c]
                       +grid[cur][r][c-1]
                       +grid[cur][r+1][c]
                       +grid[cur][r][c+1]
                      ) / 4.0;
    diff[r][c] = Math.abs(grid[nxt][r][c]-grid[cur][r][c]);
    V(done);
  }
  private op void check_diffs(int r) {
    for (int c = 2; c <= N; c++) {
      diff[r][1] = Math.max(diff[r][1], diff[r][c]);
    }
```

```
    V(done);
  }
}
```

After the first group of processes have completed, matrix `diff` contains the differences between all old and new grid points. Then, a second group of processes determines the maximum difference: `N` instances of `check_diffs` run in parallel, one for each row `i`. Each instance of `check_diffs` stores the maximum difference of the elements in its row in `diff[i][1]`. The code then updates local variable `maxdiff`, which contains the maximum of all the differences. If this value is at most `epsilon`, we exit the loop and return the results.

## 16.2 Prescheduled Strips

The main loop in the algorithm in the previous section repeatedly creates numerous processes and then waits for them to terminate. Process creation/destruction is much more time consuming than most forms of interprocess synchronization, especially when processes are repeatedly created and destroyed. Hence we can implement a data parallel algorithm much more efficiently on an asynchronous multiprocessor by creating processes once and using barriers to synchronize execution phases. (See Reference [7] for further discussion of this topic.) We can make the implementation even more efficient by having each process handle several points of the grid, not just one.

This section presents a parallel algorithm for Jacobi iteration that uses a fixed number of processes. As in the matrix multiplication algorithm in Section 15.1, each process is responsible for a strip of the grid. In particular, for an $N \times N$ grid, we use `PR` processes, with each responsible for `S` rows of the grid. The solution also illustrates one way to implement a monitor [25] in JR.

Our program employs four classes: a main class, a barrier class, a Jacobi class, and a results class. The results class is the same as in the previous section. The main class is similar to the one in the previous section. The differences are that it now also reads in `PR`, computes the strip size, and passes both to Jacobi's constructor.

```
public class Main {
  private static int N, PR, S;
  private static double xl, xt, xr, xb, epsilon;
  public static void main(String [] args) {
    // read N, PR, xl, xt, xr, xb, and epsilon
    // as input or command-line args
    ...
    S = N/PR;
    Jacobi j = new Jacobi(PR, S);
    Results results = j.compute(N, xl, xt, xr, xb, epsilon);
    results.print();
```

```
  }
}
```

The BarrierSynch class implements a barrier synchronization point for PR processes. It is essentially a monitor, with mutual exclusion and condition synchronization implemented using semaphores. In particular, the class provides one public operation barrier. Processes call barrier when they reach a barrier synchronization point. All but the last to arrive block on a semaphore. The last to arrive awakens those that are sleeping and resets the local variables.

```
public class BarrierSynch {
  public op void barrier();
  private int PR; // number of processes
  private int cnt = 0, sleep = 0;
  private sem mutex = 1;
  private cap void () delay[];

  BarrierSynch (int PR) {
    this.PR = PR;
    delay = new cap void()[2];
    for (int i = 0; i < 2 ; i++ ) {
      delay[i] = new sem;
    }
  }
  public void barrier() {
    P(mutex);
    cnt++;
    if (cnt < PR) { // i.e., all but last to reach barrier
      int mysleep = sleep;
      V(mutex);
      P(delay[mysleep]);
    }
    else { // so cnt == PR, i.e., last to reach barrier
      cnt = 0;
      for (int i = 1; i <= PR-1; i++) { V(delay[sleep]); }
      sleep = 1-sleep; // switch delay semaphores
      V(mutex);
    }
  }
}
```

Two delay semaphores are needed to prevent processes that are quick to arrive at the next barrier synchronization point from "stealing" signals intended for processes that have not yet left the previous barrier. Their use is analogous to the use of an array of condition variables in a monitor. Variable sleep indicates which element of delay a process is to block on; its value alternates between 0 and 1. Before blocking, a process copies the current value of sleep into a local variable; this is necessary since the value of sleep could otherwise change before the process blocks (see Exercise 16.6).

The `Jacobi` class implements parallel Jacobi iteration using `PR` processes. As before, each process repeatedly executes three phases: computing new values, replacing old values by new ones, and checking for convergence. These are implemented in pretty much the same way as in the data parallel algorithm. To ensure that all processes complete one phase before beginning the next, the first and last are followed by a barrier synchronization point.

```
public class Jacobi {
  private int N, PR, S;
  private double grid[][][];
  private double epsilon;
  private int cur = 0, nxt = 1;
  private double [] maxdiff;;
  private int iters = 0;
  private BarrierSynch bar;
  private sem done;

  public Jacobi(int PR, int S) {
    this.PR = PR;
    this.S = S;
    bar = new BarrierSynch(PR);
  }

  public Results compute(int N, double xl, double xt,
                         double xr, double xb, double epsilon) {
    this.N = N;
    this.epsilon = epsilon;
    grid = new double[2][N+2][N+2];
    // initialize grid;
    // identical to that in previous section.
    ...
    maxdiff = new double [PR];
    // start PR processes
    for (int p = 0; p < PR; p++) {
      send strip(p);
    }
    // wait for all to complete
    for (int p = 0; p < PR; p++) {
      P(done);
    }
    return new Results(iters, grid[cur]);
  }
  // one strip process for each strip of the grid.
  private op void strip(int p) {
    final int sr = p * S + 1; // starting row of strip
    double mdiff;
    while( true ) {
      // if process 0, count iterations
      if (p == 0) iters++;
```

```
      // compute new values for strip of grid
      // use local variable to hold maximum difference
      mdiff = 0.0;
      for (int r = sr; r <= sr+S-1; r++) {
        for (int c = 1; c <= N; c++) {
          grid[nxt][r][c] =
                    ( grid[cur][r-1][c]
                     +grid[cur][r][c-1]
                     +grid[cur][r+1][c]
                     +grid[cur][r][c+1]
                    ) / 4.0;
           mdiff = Math.max(mdiff,
                       Math.abs(grid[nxt][r][c]-grid[cur][r][c]));
        }
      }
      maxdiff[p] = mdiff;
      call bar.barrier();
      // if process 0, swap roles of grids
      if (p == 0) {
        int tmp = cur;
        cur = nxt;
        nxt = tmp;
      }
      // check for convergence and possibly terminate
      for (int i = 0; i < PR; i++) {
        mdiff = Math.max(mdiff, maxdiff[i]);
      }
      if (mdiff <= epsilon) break;
      call bar.barrier();
    }
    V(done);
  }
}
```

As commented in the code, only the first process counts the number of iterations (all will execute the same number since all use the same convergence criterion). Also, only the first process executes the swap statement that switches the roles of the grids. Variable `iters` is global to the processes so that it is accessible to the compute method. The `strip` processes are not created using the `process` abbreviation for the same reason discussed in Section 15.1. (See Exercise 16.7.)

To avoid cache update conflicts, each `strip` process uses a local variable to accumulate the maximum difference between old and new values in its strip of the grid. Only at the end of phase one of its main loop does a process store into shared array `maxdiff`. In the second phase, each process then reads these shared values to determine the maximum difference in the entire grid; again it uses local variable `mdiff` to avoid writing into shared variables.

## 16.3 A Distributed Heartbeat Algorithm

The previous two programs for Jacobi iteration use shared variables. In this section we present a distributed program that uses message passing. The program again employs PR processes, and each is responsible for a strip of S rows of the grid. Also, each process repeatedly executes the same three phases: updating grid points, copying new values into old, and checking for termination. These phases are realized differently, however, as is end-of-phase synchronization.

In a distributed JR program that might execute on multiple virtual machines, we cannot use shared variables to store shared data since each virtual machine gets a distinct copy of static variables.

Hence we will use four classes—main, results, Jacobi, and worker classes—with no public static variables. The main class and the result class are identical to those in the previous section.

The Worker class implements the computation proper. The Jacobi class creates PR instances of Worker and then starts the computation in each instance. During each iteration of the computation, instances of Worker exchange the boundaries of their strip of the grid.

The Worker class provides three public operations: toprow, which is used to acquire a new top boundary; bottomrow, which is used to acquire a new bottom boundary; and compute, which is used to start the computation. Each instance of Worker is also parameterized with several values.

```
public class Worker {
  public op void compute( remote Worker up,
                          remote Worker down,
                          cap boolean (double) terminate,
                          cap void (int, double [][]) gather);
  public op void toprow   (double []);
  public op void bottomrow(double []);

  private int id, N, PR, S;
  private double grid[][][];
  private int cur = 0, nxt = 1;
  private double diff;
  public Worker(int id, int N, int PR, int S,
                double xl, double xt, double xr, double xb) {
    this.id = id;
    this.N = N;
    this.PR = PR;
    this.S = S;
    grid = new double[2][S+2][N+2];

    // initialize grid
    for (int g = 0; g <= 1; g++) {
      for (int r = 0; r <= S+1; r++) {
```

```
        grid[g][r][0] = xl;
        grid[g][r][N+1] = xr;
        for (int c = 1; c <= N; c++) {
          grid[g][r][c] = 0.0;
        }
      }
    }
    if (id==1) {// top for process 1
      for (int c = 0; c <= N+1; c++) {
        grid[cur][0][c] = grid[nxt][0][c] = xt;
      }
    }
    if (id==PR) {// bottom for process PR
      for (int c = 0; c <= N+1; c++) {
        grid[cur][S+1][c] = grid[nxt][S+1][c] = xb;
      }
    }
  }
  public void compute( remote Worker up,
                       remote Worker down,
                       cap boolean (double) terminate,
                       cap void (int, double [][]) gather) {
    while (true) {
      // compute new values for grid points
      diff = 0.0;
      for (int r = 1; r <= S; r++) {
        for (int c = 1; c <= N; c++) {
          grid[nxt][r][c] =
              ( grid[cur][r-1][c]
               +grid[cur][r][c-1]
               +grid[cur][r+1][c]
               +grid[cur][r][c+1]
               ) / 4.0;
          diff = Math.max(diff,
                  Math.abs(grid[nxt][r][c]-grid[cur][r][c]));
        }
      }
      // replace old values by new ones, and
      // exchange top and bottom rows with neighbors
      int tmp;
      tmp = cur; cur = nxt; nxt = tmp;
      // for top worker, next is noop
      send up.bottomrow(grid[cur][1]);
      // for bottom worker, next is noop
      send down.toprow(grid[cur][S]);
      if (id != 1) {
        receive toprow(grid[cur][0]);
      }
      if (id != PR) {
        receive bottomrow(grid[cur][S+1]);
```

```
      }
      // check for termination
      if (terminate(diff)) break;
    }
    send gather(id, grid[cur]);
  }
}
```

Each instance of `Worker` is responsible for a strip of `S` rows and `N` columns of the grid. As before, array `grid` contains two matrices, and each has two extra rows and columns to hold the values on the edges of the strip. The extra columns contain boundary values, which do not change. However, the extra rows contain values computed on each iteration by the instances of `Worker` responsible for adjacent strips. Hence each instance exchanges these rows with its neighbors on each iteration. The instances responsible for the topmost strip and bottommost strip have only one neighbor, so they exchange only one row. As we shall see, remote reference `up` is set to `noop` for the topmost strip, and down is set to `noop` for the bottommost strip; this makes the corresponding send statements have no effect.

The Jacobi class provides one public method, `compute`, which controls the computation. The `compute` method creates instances of `Worker`, starts the computation, checks for termination, gathers the results from the workers, and returns the overall result to the invoker.

```
public class Jacobi {
  private int PR, S;
  public Jacobi(int PR, int S) {
    this.PR = PR;
    this.S = S;
  }
  public Results compute(int N, double xl, double xt,
                         double xr, double xb, double epsilon) {
    op boolean terminate(double);
    op void gather(int, double [][]);
    // create instances of Worker
    remote Worker wref[] = new remote Worker[PR];
    double [][] matrix = new double [N+2][N+2];
    int iters = 0;
    wref[0] = new remote Worker(1,N,PR,S,xl,xt,xr,0.0);
    for (int i = 1; i <= PR-2; i++) {
      wref[i] = new remote Worker(i+1,N,PR,S,xl,0.0,xr,0.0);
    }
    wref[PR-1] = new remote Worker(PR,N,PR,S,xl,0.0,xr,xb);
    // start the computation
    send wref[0].compute(noop, wref[1], terminate, gather);
    for (int i = 1; i <= PR-2; i++) {
      send wref[i].compute(wref[i-1],wref[i+1],terminate, gather);
    }
```

```
    send wref[PR-1].compute(wref[PR-2], noop, terminate, gather);
    // do termination checks until convergence
    boolean ans = false;
    while (!ans) {
      iters++;
      inni boolean terminate(double diff)
       st terminate.length() == PR by -diff {
        ans = (diff <= epsilon);
        for (int i = 1; i <= PR-1; i++) {
          inni boolean terminate(double diff_unused) {
            return ans;
          }
        }
        return ans;
      }
    }
    // accumulate results from each Worker.
    // Worker sends back all its S+2 rows;
    // copy middle S rows from all Workers,
    // but also topmost row from top Worker
    // and bottommost row from bottom Worker.
    for (int i = 0; i < PR; i++) {
      inni void gather(int id, double [][] chunk) {
        // copy chunk into matrix
        for (int r = (id==0?0:1); r <= (id==PR?S+1:S); r++) {
          matrix[r+(id-1)*S] = chunk[r];
        }
      }
    }
    return new Results(iters, matrix);
  }
}
```

The `compute` method provides two operations. Its `terminate` operation is called by instances of `Worker` to check for termination. Its `gather` operation is invoked by instances of `Worker` to return their part of the overall result.

The code that creates instances of `Worker` passes each worker its `id`, and appropriate values for the worker to initialize its part of the grid. It saves references for the workers for use when starting the computation. (The code assumes that `PR` > 1; see Exercise 16.12.)

The code that starts the computation invokes the `compute` operation in each of the `PR` instances of `Worker`. Each worker instance is passed references for its two neighboring worker instances. Since the first and last instances of `Jacobi` have only one neighbor, each is passed one `noop` capability. Each worker is also passed two capabilities: one for the terminate operation and one for the gather operation.

The loop in `Jacobi`'s compute method that checks for termination illustrates an interesting use of nested input statements. On each iteration of its computational loop, each instance of `Worker` calls `terminate(diff)`, where `diff` is the maximum difference it found. The outer input statement here uses a synchronization expression `(terminate.length() == PR)` to wait for all `PR` instances of `Worker` to call `terminate`. It then uses a scheduling expression `(-diff)` to service the invocation that has the maximum value for parameter `diff`. If this maximum is at most `epsilon`, the computation has converged, so ans is set to true; otherwise ans is set to false.

The inner input statement is used to service and return the value of `ans` to the `PR-1` other invocations of `terminate`. It ignores the parameter passed in because that value is smaller than the maximum difference, as described above.

After the computation has converged, the `compute` method receives results from each worker. It combines them into a single result, which it returns to its invoker.

## 16.4 Using Multiple Virtual Machines

The program in the previous section will execute on one virtual machine. Here we show how to extend it to employ multiple virtual machines, which can be on multiple physical machines.

To use multiple virtual machines, we need to make only a few changes to the Jacobi class; the other classes do not need to change at all. In particular, we need to create virtual machines before we create instances of `Worker`. Let `hosts` be the name of a file that contains a list of at least `PR` strings, each of which is the name of a physical machine. Then the following code will create a virtual machine on each of the named host machines and then create an instance of `Worker` on each:

```
// create virtual machines;
// create instances of Worker, one per virtual machine
remote Worker wref[] = new remote Worker[PR];
double [][] matrix = new double [N+2][N+2];
int iters = 0;
vm [] vmref = new vm[PR];
FileReader fr = null;
try {
  fr = new FileReader("hosts");
} catch (FileNotFoundException fe) {
  System.err.println("can't open hosts");
}
BufferedReader br = new BufferedReader(fr);
for (int i = 0; i < PR; i++) {
  String hostname = null;
  try {
    hostname = br.readLine();
```

```
    } catch (IOException ioe) {
      System.err.println("IO Exception on getting hostname");
    }
    vmref[i] = new vm() on hostname;
  }
  try {
    fr.close();
  } catch (IOException ioe) {
    System.err.println("IO Exception on close");
  }
  wref[0] = new remote Worker(1,N,PR,S,xl,xt,xr,0.0)
            on vmref[0];
  for (int i = 1; i <= PR-2; i++) {
    wref[i] = new remote Worker(i+1,N,PR,S,xl,0.0,xr,0.0)
              on vmref[i];
  }
  wref[PR-1] = new remote Worker(PR,N,PR,S,xl,0.0,xr,xb)
               on vmref[PR-1];
```

No further changes to the distributed program are required.

## Exercises

16.1 Copy the four programs in this chapter into files and compile them. (Source files containing the programs come with the JR distribution.) Construct a set of experiments to determine the relative performance of the four programs. Experiment with different problem sizes, numbers of processes, and numbers of processors. Explain the results you observe.

16.2 (a) Develop a program for Jacobi iteration that uses a bag of tasks and replicated workers, as illustrated in Section 15.2. Experiment with different problem sizes, tasks sizes, and numbers of workers.

(b) Compare the results of your experiments to the results of the experiments conducted in Exercise 16.1. Explain any differences you observe.

16.3 Consider the data parallel program in Section 16.1.

(a) Modify the program to employ only `PR` processes in `compute`'s main loop instead of `N*N` or `N`. Assume that `N` is a multiple of `PR`.

(b) Compare the performance of your answer to (a) to the performance of the original program. Experiment with different values of `PR`. Explain your results.

16.4 Rewrite the convergence-checking code in `Jacobi` in Section 16.1 so that only one process checks for convergence. If that process finds that the maximum difference is at most `epsilon`, it should inform the others.

16.5 Consider again the `Jacobi` program in Section 16.1. It uses an array consisting of two grids of values—a current grid and a next grid—to avoid having to copy values from the next grid to the current grid. Recall that arrays in Java (and so in JR) are object references.

Is the above technique any more efficient than having two separate grid variables, say `gridcur` and `gridnxt`, and swapping them by

```
temp    = gridcur;
gridcur = gridnxt;
gridnxt = temp;
```

First, answer the above question based only on what you expect to happen based on your understanding of Java. Then, modify the program as suggested and run several tests on different input data. Explain any differences in what you expected and what you observed.

16.6 Consider the `BarrierSynch` class (see Section 16.2), which implements a barrier.

(a) Suppose local variable `mysleep` is not used. In particular, delete its declaration and change the `P` operation to `P(delay[sleep])`. Explain why the resulting code is incorrect. Hint: When could context switches occur?

(b) Suppose the array of `delay` semaphores is replaced by a single semaphore (and variable `sleep` is deleted). Explain why the resulting code is incorrect when used by the code in `Jacobi` (see Section 16.2). Is there any situation in which the modified code would correctly implement a barrier?

16.7 Rewrite the Jacobi class in Section 16.2 so that the `strip` processes are declared as a family of processes using the `process` abbreviation. Be sure your solution prevents the potential problem mentioned in the text; i.e., it prevents these processes from starting their computations before instance variables have been initialized.

16.8 In the convergence-checking code in `Jacobi` in Section 16.2, every `strip` process calculates the maximum value in the array `maxdiff`.

(a) Modify the program so that only one process computes the maximum and other processes wait until that value has been computed.

(b) Compare the performance of your answer to part (a) with the original code for various numbers of `strip` processes. Which is faster? Why? How does performance depend on the number of processes?

16.9 The distributed program in Section 16.3 partitions the grid into strips and assigns one instance of `Worker` to each strip. Suppose instead that the grid is partitioned into blocks; e.g., a 100 × 100 point grid is partitioned into 16 blocks of 25 × 25 points each. Modify the program to implement this approach.

16.10 The `Jacobi` class in Section 16.3 implements convergence checking. Modify the program in Section 16.3 so that the instances of `Worker` interact only with each other to check for convergence. In particular, delete the block of code in `Jacobi` between the comment `// do termination ...` and the code that accumulates the results.

16.11 The termination checking code in `Jacobi` in Section 16.3 uses the `length` method. Show how to rewrite the code in the following ways so as not to use the `length` method.

(a) Change the interface to the `terminate` operation and/or introduce another operation.

(b) Do not change the interface to the `terminate` operation. Instead use the forward statement.

(c) Do not change the interface to the `terminate` operation or use the forward statment. Instead use additional processes.

(d) Use recursion to simulate nested input statements, but do not use any of the above "tricks."

16.12 The code in `Jacobi` in Section 16.3 (and Section 16.4) assumes that `PR` > 1. Modify the code so it works when `PR = 1`.

16.13 The code in `Jacobi` in Sections 16.3 and 16.4 uses the `noop` remote reference. Show how to rewrite the code in Section 16.4 so it does not use `noop`.

16.14 Gauss-Seidel and successive over-relaxation (SOR) are two additional iterative methods for solving Laplace's equation. With Gauss-Seidel, on each iteration, new values for points are computed sequentially; each new value is the average of two values from the current iteration and two from the previous iteration. In particular, new values are computed by the following loop:

```
for (int r = 1; r <= N; r++ ) {
  for (int c = 1; c <= N; c++ ) {
    grid[r][c] = ( grid[r-1][c] + grid[r][c-1] +
                   grid[r+1][c] + grid[r][c+1] ) / 4.0;
  }
```

```
}
```

Note that `grid` is updated in place, unlike in Jacobi iteration.

SOR is a generalization of Gauss-Seidel that also averages in the previous value of a point. With SOR, new values are computed by

```
for (int r = 1; r <= N; r++ ) {
  for (int c = 1; c <= N; c++ ) {
    grid[r][c] = omega * ( grid[r-1][c] + grid[r][c-1] +
                           grid[r+1][c] + grid[r][c+1] ) / 4.0
               + (1-omega)*grid[r][c];
  }
}
```

Variable `omega` is called the over-relaxation parameter. For optimum convergence it usually is chosen to be between 1 and 2. (If `omega` is 1, SOR simplifies to Gauss-Seidel.)

Write sequential JR programs to implement Jacobi iteration, Gauss-Seidel, and SOR. Compare the performance of your programs. For a given set of initial values, which converges most rapidly? How does the rate of convergence depend on the initial values? How does the performance of SOR depend on the value of `omega`?

16.15 The Gauss-Seidel and SOR methods defined in the previous problem have to be applied sequentially in order to converge. Both methods update points in place, which can lead to chaos if the points were all updated concurrently. However, both can be parallelized by using a red/black (checkerboard) partitioning scheme that partitions the grid of points into blocks. For example, partition a 100 × 100 point grid into 16 blocks of 25 × 25 points each. Next color each block red or black in a checkerboard fashion; i.e., adjacent blocks have different colors. Assign a process to each block or, better yet, to each set of two or four blocks. On each iteration of the main computation, concurrently update all red blocks, then concurrently update all black blocks. Within a block use Gauss-Seidel or SOR to update points sequentially.

(a) Implement a parallel algorithm for red/black SOR. Use shared variables and a value of 1.5 for `omega`.

(b) Implement a distributed algorithm for red/black SOR. Use message passing and a value of 1.5 for `omega`.

(c) Construct a set of experiments to determine the performance of your two programs. Compare their performance to the corresponding programs in this chapter that use Jacobi iteration.

16.16 The following region-labeling problem arises in image processing. Given is an `n` × `n` integer array `image`. The value of each entry is the intensity of a pixel. The neighbors of a pixel are the four pixels that surround it, i.e., the elements of `image` to the left, right, above, and below it. Two pixels belong to the same region if they are neighbors and they have the same value. Thus a region is a maximal set of pixels that are connected and that all have the same value.

The problem is to find all regions and assign every pixel in each region a unique label. In particular, let `label` be a second `n` × `n` matrix, and assume that the initial value of `label[i][j]` is `n*i+j`. The final value of `label[i][j]` is to be the largest of the initial labels in the region to which pixel `[i][j]` belongs.

(a) Write a shared-variable program to solve this problem. Divide the image into fixed-size sub-images, and assign one process to each sub-image.

(b) Write a recursive, divide-and-conquer algorithm to solve this problem. Start with the entire image and recursively fork processes to assign labels to sub-images; stop recursing when you reach a sub-image of some prespecified size. When subprocesses terminate, combine their labeled sub-images into a larger labeled image.

(c) Write a distributed program to solve this problem; use a heartbeat algorithm. Divide the image into sub-images of some prespecified size, and assign one process to each sub-image.

(d) Write a program to solve this problem using the bag-of-tasks paradigm. Divide the image into sub-images of some prespecified size, and put these "tasks" into a bag. Worker processes repeatedly take tasks from this bag, label the corresponding sub-image, and put the labeled sub-images into a second bag. Other workers repeatedly take pairs of adjacent sub-images from the second bag, combine them into a larger labeled image, and put the combined image back into the second bag. The computation terminates when the entire image has been properly labeled. Implement the bags by means of operations that are shared by the worker processes.

(e) Repeat part (d), but do not use shared bags of tasks. Instead, have one or two administrator processes implement the bags.

(f) Write a program to solve this problem using a data parallel algorithm. First, for each pixel determine whether it is on the boundary of a region. Second, have each boundary pixel determine which neighbors are also on the boundary; in essence, for each region this produces a doubly linked list connecting all pixels that are on the boundary of

that region. Third, using the lists, propagate the largest label of any of the boundary pixels to the others that are on the boundary. (The pixel in a region that has the largest label will be on its boundary.) Finally, propagate the label for each region to pixels in the interior of the region.

(g) Construct a set of experiments to compare the performance of the programs you wrote for previous parts of this problem. Experiment with different image sizes and numbers of processes. Explain the results you observe.

Chapter 17

# THE TRAVELING SALESMAN PROBLEM

The traveling salesman problem is the classic "hard" combinatorial search problem. Given are `n` cities and an `n` × `n` matrix `dist` of intercity distances. The value in `dist[i][j]` is the distance from city `i` to city `j`, e.g., the airline miles. We assume there is a direct connection from each city to every other.

A salesman starts in city 1 and wishes to visit every city exactly once, ending back in city 1. The problem is to determine a path that minimizes the distance the salesman must travel. Thus we need to find a permutation of integers 1 to `n` such that the sum of the distances between adjacent pairs of cities—plus the distance back to city 1—is minimized.

For `n` cities, there are `(n-1)`! different paths starting and ending in city 1. Unless `n` is small, this number is, of course, very large. Thus we need to look for ways to reduce the amount of computation that has to be performed and for ways to use parallelism to speed up the computation.

This chapter presents three solutions to the traveling salesman problem. To simplify the programs, we develop exact solutions, i.e., ones that find a minimum cost tour. In practice, finding an exact solution is infeasible except for small values of `n` (e.g., 15 or so). Consequently, many heuristics have been developed to generate approximate solutions to the traveling salesman and similar optimization problems. A few are considered in the exercises; see References [34] and [27] for descriptions of those heuristics and numerous others.

The first solution is a sequential program that uses depth-first search to examine all feasible paths. A path is feasible if it is not (yet) longer than the best complete path that has been computed so far. The second solution is a parallel program that uses the bag-of-tasks paradigm. In particular, partial paths are stored in an operation queue shared by several worker processes; each worker repeatedly extracts a partial path and extends it with cities that have not yet been

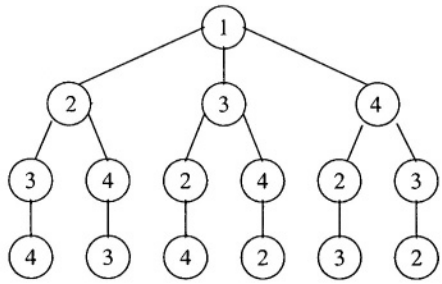


*Figure 17.1.* Search tree for four cities

visited. The final, distributed solution modifies the second program to use only message passing.

These last two programs are not as reusable as the programs in the previous two chapters. That is, they depend on JR's automatic termination detection to terminate. To make them reusable, they can be rewritten without relying on such a feature. Exercises 17.5 and 17.6 explore such explicit termination.

All three solutions illustrate techniques for solving branch-and-bound algorithms, which arise in numerous applications such as searching game trees and solving optimization problems.

## 17.1 Sequential Solution

To find the shortest path that visits all cities exactly once, we have to consider every possible tour. If we start in city 1, there are `n-1` possible cities we could visit next. From each of these, there are `n-2` possible cities to visit third, and so on. We can thus represent all possible tours by a tree, with city 1 at the root. The tree has depth `n` (the number of cities) and `(n-1)`! leaves (the number of different tours). Figure 17.1 illustrates a search tree for four cities.

The standard way to examine all paths in a tree such as this is to use depth-first search, which is realized by a recursive, backtracking algorithm. We must follow a path all the way to a leaf node, then go part way back up the tree and follow a path to a different leaf node, and so on. For example, if we search from left to right in the tree in Figure 17.1, we would visit the four cities in the following order:

(1, 2, 3, 4), (1, 2, 4, 3), (1, 3, 2, 4),
(1, 3, 4, 2), (1, 4, 2, 3), (1, 4, 3, 2)

In the traveling salesman problem, the goal is to find the shortest tour. It is not necessary to consider a tour that is known to be longer than the shortest complete tour that has been found so far. We can use this fact to "prune" infeasible paths from the tree. The larger the number of cities, the more dramatic the effect

pruning can have. For example, on sample data for ten cities, we have seen pruning reduce execution time by a factor of ten.

Below we present a sequential JR program to solve the traveling salesman problem. Our solution consists of three classes: a main class, a compute class, and a results class.

The results class is the simplest. It is simply a container for the results of the computation; it provides a `print` method to print out the results. (Note that cities in the output are numbered 1, 2, ... `n`, but within the programs in this chapter cities are numbered 0, 1, ..., `n-1` to simplify array subscripting.)

```
public class TSPResults {
  // return info from computing shortest path
  private int shortest; // length of shortest path
  private int [] shortest_path; // actual shortest path
  public TSPResults(int shortest, int [] shortest_path) {
    this.shortest = shortest;
    this.shortest_path = shortest_path;
  }
  public void print() {
    System.out.println("the shortest path has length "+shortest);
    System.out.println("the cities on the shortest path are:");
    for (int i = 0; i < shortest_path.length; i++) {
      // +1 is to adjust city in [0,n-1] to [1,n]
      System.out.print(" " + (shortest_path[i]+1));
    }
   System.out.println();
 }
}
```

The main class reads two command-line arguments: the number of cities and the name of a file that contains the distance matrix. It reads in the distance matrix, initiates computation, and outputs the results.

```
import java.io.*;
import java.util.*;
public class TSPSeq {

  public static void main(String [] args) {
    int n;
    int [][] dist;
    n = Integer.parseInt(args[0]);
    dist = read_dist(n, args[1]);
    TSPCompute tsp = new TSPCompute();
    TSPResults results = tsp.compute(n, dist);
    results.print();
  }
  private static int [][] read_dist(int n, String fname) {
    int [][] dist = new int [n][n];
```

```
    try {
      BufferedReader br;
      FileReader fr = new FileReader(fname);
      br = new BufferedReader(fr);
      for (int i = 0; i < n; i++) {
        StringTokenizer stok = new StringTokenizer(br.readLine());
        for (int j = 0; j < n; j++) {
         dist[i][j] = Integer.parseInt(stok.nextToken());
        }
      }
      fr.close();
    } catch (FileNotFoundException fe) {
      System.err.println("can't open " + fname);
    } catch (IOException ioe) {
      System.err.println("IO Exception for " + fname);
    }
    return dist;
  }
}
```

The compute class contains four methods. The computation is carried out by method `tsp`. The `compute` method invokes `tsp` once for each partial path of length 2; `tsp` in turn recursively examines all other feasible paths. The `visited` method is used to determine whether a city has already been visited on the given path. (A `visited` boolean array could be used instead; see Exercise 17.2.) The `update` method is used to update the shortest path.

```
public class TSPCompute {
  // best length so far and corresponding path.
  private int shortest = Integer.MAX_VALUE;
  private int [] shortest_path;

  private int n;
  private int [][] dist;

  public TSPResults compute(int n, int [][] dist) {
    this.n = n;
    this.dist = dist;
    int [] path = {0, -1}; // path of length 2
    for (int i = 1; i < n; i++) {
      path[1] = i;
      tsp(path, dist[0][i]);
    }
    return new TSPResults(shortest, shortest_path);
  }
  // using recursion and backtracking, examine all
  // paths that could be the shortest
  private void tsp(int [] path, int length) {
    int hops = path.length;
```

```
    if (hops < n) { // have incomplete tour, extend it
      for (int city = 1; city < n; city++) {
        if (!visited(path,city)) {
          // consider new path as path extended with city
          int newlength = length + dist[path[hops-1]][city];
          // recurse if new path possibly the best
          if (newlength < shortest) {
            int [] newpath = new int [hops+1];
            System.arraycopy(path, 0, newpath, 0, path.length);
            newpath[hops] = city;
            tsp(newpath, newlength);
          }
        }
      }
    }
    else if (hops == n) { // have a complete tour, see if best
      length += dist[path[n-1]][0]; // back to start
      update(path, length);
    }
  }
  // return true iff city already on this path
  private static boolean visited(int [] path, int city) {
    for (int i = 1; i < path.length; i++) {
      if (path[i] == city) {
        return true;
      }
    }
    return false;
  }
  private void update(int [] path, int length) {
    if (length < shortest) {
      shortest = length;
      shortest_path = path;
    }
  }
}
```

## 17.2 Replicated Workers and a Bag of Tasks

In the traveling salesman problem, paths are independent, so we could evaluate all of them in parallel. However, this would lead to far too much concurrency for most problem sizes and machines. An alternative is to employ a fixed number of worker processes that share a bag of tasks.

Each task contains a partial path, the number of cities (hops) on the path, and the path's length. We will initially put `n-1` tasks in the bag, representing the `n-1` tours starting at city 1. Each worker repeatedly takes a task from the bag and extends the path with every city that has not yet been visited. If a new path is too long, it is discarded. If a path does not include all cities and it is not

yet too long, the worker puts the new path and its length back into the bag of tasks. If a path includes all cities and it might be shorter than the shortest path found so far, the worker updates the shortest path.

Our program for this algorithm again has three classes: a main class, a compute class, and a results class. The results class is identical to that in the previous solution. The main class is nearly identical to that in the previous solution. The difference is that it now handles an additional command-line argument specifying the number of worker processes to employ and passes that to the compute class:

```
n = Integer.parseInt(args[0]);
w = Integer.parseInt(args[1]);
dist = read_dist(n, args[2]);
TSPCompute tsp = new TSPCompute(w);
TSPResults results = tsp.compute(n, dist);
results.print();
```

The compute class now contains a bag of tasks, code that initializes the bag with `n-1` tasks, and a family of `w worker` processes. The constructor code simply initializes the object's copy of `w`. The compute method initializes the bag by sending to it all partial paths of length 2 that start in the first city. (Note how the code uses a new, local array for each invocation for the reason given in Section 7.8.) It returns the results when the computation completes. Notice how the code creates and registers a quiescence operation, `done`, and waits for `done` to be invoked, which occurs when the computation has completed, i.e. the bag is empty and no workers are active.

Tours are computed by the `worker` processes. On each iteration, the worker process receives a new task from the shared bag. The task is a partial tour, which the worker extends by each city that has not yet been visited. If a complete tour is found that might be the best, the worker calls the shared procedure `update`. Because update alters shared variables and might be called by more than one worker at a time, its body now executes as a critical section. Again, we use the length of the shortest tour found so far to avoid searching infeasible paths. Note, though, that we do *access* the shared variable `shortest` without protection in two conditions on if statements. If `shortest` is changed immediately after it is accessed, then the program might perform some unnecessary computation, but the overall result will still be correct.

```
public class TSPCompute {
  // best length so far and corresponding path.
  private int shortest = Integer.MAX_VALUE;
  private int [] shortest_path;

  private int n;
  private int [][] dist;
```

```
private int w;
private op void bag(int [], int, int); // tasks
private op void update(int [], int);
private sem mutex = 1;

public TSPCompute(int w) {
  this.w = w;
}
public TSPResults compute(int n, int [][] dist) {
  this.n = n;
  this.dist = dist;
  for (int i = 1; i < n; i++) {
    // path must be local to each send
    int [] path = {0, i};
    send bag(path, 2, dist[0][i]);
  }
  // wait for workers to complete computation
  op void done ();
  try {
    JR.registerQuiescenceAction(done);
  } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
    e.printStackTrace();
  }
  receive done();
  return new TSPResults(shortest, shortest_path);
}
private process worker( (int id = 1; id <= w; id++) ) {
  while(true) {
    int [] path; int hops, length;
    receive bag(path, hops, length);
    if (hops < n) { // have incomplete tour, extend it
      for (int city = 1; city < n; city++) {
        if (!visited(path,city)) {
          // consider new path as path extended with city
          int newlength = length + dist[path[hops-1]][city];
          // if new path is viable, put in bag
          if (newlength < shortest) {
            int [] newpath = new int [hops+1];
            System.arraycopy(path, 0, newpath, 0, path.length);
            newpath[hops] = city;
            send bag(newpath, hops+1, newlength);
          }
        }
      }
    }
    else if (hops == n) { // have a complete tour, see if best
      length += dist[path[n-1]][0]; // back to start
      if (length < shortest) { // possibly best
        update(path, length);
      }
```

```
      }
    }
  }
  // return true iff city already on this path
  private static boolean visited(int [] path, int city) {
   // body same as in previous section
  }
  private void update(int [] path, int length) {
    P(mutex); // get lock for shared variables
    if (length < shortest) {
      shortest = length;
      shortest_path = path;
    }
    V(mutex); // release lock
  }
}
```

If there are more than a small number of cities (e.g., more than ten), this program generates a huge number of partial tours. In fact the size of the bag could become so large that the program will run out of memory. A better approach is to put some fixed number of tasks in the bag to start—say partial tours of length three. Then on each iteration a worker process extracts one partial tour and uses the sequential algorithm of the previous section to examine all paths starting with that partial tour. In addition to decreasing the amount of storage required for the bag, this approach also increases the amount of computation a worker does every time it accesses the bag.

## 17.3 Manager and Workers

The program in the previous section employs shared variables. However, variables cannot be shared across virtual machines. Instead each gets its own copy.

Here we present a distributed program that does not use shared variables. To do so, we now represent each worker within a separate class, `TSPWorker`. However, the bag of tasks and shortest path are now maintained within the compute class, which contains a manager process. The workers and manager use asynchronous message passing, RPC, and rendezvous to communicate with each other.

The main class and the results class are identical to those in the previous solution. See the previous sections for their code.

As in the previous section, the worker process repeatedly gets a partial tour from the bag and extends it with all cities that have not yet been visited. A worker process simply receives a new task from the bag, even though bag is declared in a different class (which could even be located on a different virtual machine).

One difference between the `worker` process below and the one in Section 17.2 is that the length of the shortest path is not directly accessible in a shared variable. Instead the manager keeps track of the shortest path. Any time it changes, the manager sends the new value to the `updatemin` operation exported by each instance of `Worker`.

```
public class TSPWorker {
  private int n;
  private int [][] dist;
  private int w;
  private remote TSPCompute c;
  public op void updatemin(int);

  public TSPWorker(int n, int [][] dist, remote TSPCompute c) {
    this.n = n;
    this.dist = dist;
    this.c = c;
  }

  private process worker {
    int shortest = Integer.MAX_VALUE;
    while(true) {
      int [] path; int hops, length;
      // see if there is a better shortest tour
      while (updatemin.length() > 0) {
        receive updatemin(length);
        shortest = Math.min(length, shortest);
      }
      // get a task and then process it
      receive c.bag(path, hops, length);
      if (hops < n) { // have incomplete tour, extend it
        for (int city = 1; city < n; city++) {
          if (!visited(path,city)) {
            // consider new path as path extended with city
            int newlength = length + dist[path[hops-1]][city];
            // if new path is viable, put in bag
            if (newlength < shortest) {
              int [] newpath = new int [hops+1];
              System.arraycopy(path, 0, newpath, 0, path.length);
              newpath[hops] = city;
              send c.bag(newpath, hops+1, newlength);
            }
          }
        }
      }
      else if (hops == n) { // have a complete tour, see if best
        length += dist[path[n-1]][0]; // back to start
        if (length < shortest) {
          // tell manager about this possibly best tour
```

```
          send c.newmin(path, length);
          shortest = length;
        }
      }
    }
  }

  // return true iff city already on this path
  private static boolean visited(int [] path, int city) {
    // body same as in previous section
}
```

At the start of each iteration, the `worker` process checks to see if there is a pending invocation of `updatemin`, which indicates that there is a new shortest path.

The compute class provides two public operations used by workers: `bag`, which contains the bag of tasks; and `newmin`, which is called by a worker when it thinks it has found a new shortest path. Its constructor simply saves `w`, the number of worker processes to be used.

The `compute` method acts as the manager. It first creates the `w` `TSPWorker` objects and passes each a remote object reference for itself (via `this.remote`); the worker accesses the `bag` and `newmin` operations through this reference. It also passes each instance the values of `n` and `dist` since these are no longer shared. Note that the manager needs references for the workers because it needs to invoke their `updatemin` operations.

The manager uses an input statement to service operation `newmin`. When the manager receives a new shortest path, it broadcasts the length of that path to the workers.

Two (or more) workers could, at about the same time, find what they believe to be new shortest paths. The input statement in the manager uses a scheduling expression to service the invocation of `newmin` that has the smallest value of parameter `length`. This can decrease the number of times that the manager needs to broadcast a new value of `shortest` to the workers.

```
public class TSPCompute {
  private int w;
  public op void bag(int [], int, int); // tasks
  public op void newmin(int [], int);

  public TSPCompute(int w) {
      this.w = w;
  }
  public TSPResults compute(int n, int [][] dist) { // ''manager''
    // best length so far and corresponding path.
    int shortest = Integer.MAX_VALUE;
    int [] shortest_path = null;
    remote TSPWorker [] ws;
```

```
    ws = new remote TSPWorker [w];
    // create w workers
    for (int id = 0; id < w; id++) {
      ws[id] = new remote TSPWorker(n, dist, this.remote);
    }
    for (int i = 1; i < n; i++) {
      int [] path = {0, i};
      send bag(path, 2, dist[0][i]);
    }
    // done is invoked when computation quiesces
    op void done ();
    try {
      JR.registerQuiescenceAction(done);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
    while (true) {
      // wait for candidate for shortest path
      inni void newmin(int path [], int length) by length {
        if (length < shortest) {
          shortest = length;
          shortest_path = path;
          // broadcast new minimum to all workers
          for (int id = 0; id < w; id++) {
            send ws[id].updatemin(shortest);
          }
        }
      }
      [] void done() {
        break;
      }
    }
  return new TSPResults(shortest, shortest_path);
  }
}
```

The manager uses a quiescence operation to detect when the workers have completed the computations. Its use is similar to that seen in the previous section, but here the `done` operation appears as an arm of an `inni`. Specifically, it appears as an alternative to `newmin`. The code for done just exits the loop, which causes the results to be returned from the manager.

Using the techniques shown in Section 16.4, we can readily extend the above program to execute on multiple virtual machines. For example, we could have `TSPCompute` and each instance of `TSPWorker` execute on a different virtual machine, which in turn could be on a different physical machine.

## Exercises

17.1 Modify the sequential program in Section 17.1 so that it does not prune infeasible paths. Compare the execution time of your program and the one given in the text.

17.2 Modify the sequential program in Section 17.1 so that it maintains the "visited" status of all cities in a `visited` boolean array instead of using the `visited` method to search the path. (The same technique will also work for the other programs in this chapter.)

17.3 Run the program in Section 17.2. Generate test data for various numbers of cities. (If you have access to an airline guide, you might want to use actual air distances between various cities.)

(a) Analyze the performance of the program for various sets of input data and various numbers of worker processes. Determine how large a number of cities you can handle without running out of storage for the bag of tasks.

(b) Modify the program as suggested at the end of Section 17.2. In particular, initialize the bag with some fixed number of tasks and have each worker use the sequential algorithm to extend a partial tour with all feasible tours. Analyze the performance of this program for various sets of input data and various numbers of worker processes. Compare the performance of this program to that of the program in Section 17.2.

17.4 Run the program in Section 17.3. Generate test data for various numbers of cities.

(a) Analyze the performance of the program for various sets of input data and various numbers of worker processes.

(b) Compare the performance of this program to the performance of the program in Section 17.2.

(c) Modify the program to have the manager and each worker execute on its own virtual machine, and place these on different physical machines. Analyze the performance of this program for various sets of input data and various numbers of worker processes.

17.5 The program in Section 17.2 terminates when the bag of tasks is empty and all worker processes are blocked. Suppose JR did not support automatic distributed termination detection. Modify the program to detect termination explicitly; invoke `JR.exit` when the bag is empty and all

workers are blocked. (Exercise 7.14 explores a similar problem in a different context.)

17.6 Repeat the previous exercise for the program in Section 17.3.

17.7 Consider the inner while loop in process `worker` in class `TSPWorker`; it receives all pending `updatemin` messages and updates `shortest`.

(a) The assignment to `shortest` can be replaced by

```
shortest = length;
```

but that might result in the worker performing extra computation. Explain both why doing so is correct and how extra computation might result.

(b) Consider replacing the loop by

```
// see if there is a better shortest tour
int tmp = shortest;
while (updatemin.length() > 0) {
  receive updatemin(length);
  tmp = length;
}
shortest = Math.min(tmp, shortest);
```

Comment on the correctness of this code. Explain how it works or give a counterexample of where it does not work. If it is correct, does it cause the worker to perform more work than the original?

17.8 Related to the previous problem, consider the general problem of servicing the last pending invocation for an operation, say `f (int x)`, and discarding all other pending invocations.

(a) Suppose the pending invocations appear in decreasing order of `x`.

(b) Suppose the pending invocations appear in arbitrary order. Solve this part in two ways: first using mechanisms only from Chapter 9 and then using mechanisms from Chapter 14.

17.9 Rewrite the program in Section 17.3 so that it uses no remote object reference for `TSPCompute`.

17.10 (a) Solve the traveling salesman problem by assigning one process to each city. City 1 generates partial tours of length 2 that are sent to each other city. When a city gets a partial tour, it extends it and sends it on to other cities. When it gets a complete tour, it sends it back to city 1.

(b) Compare the performance of your program to the performance of the program in Section 17.2. Explain any differences.

17.11 One heuristic algorithm for the traveling salesman problem is called the *nearest neighbor* algorithm. Starting with city 1, first visit the city, say `c`, nearest to city 1. Now extend the partial tour by visiting the city nearest to `c`. Continue in this fashion until all cities have been visited, then return to city 1.

Write a program to implement this algorithm. Compare its performance to that of the programs in the text. What is the execution time? How good or bad an approximate solution is generated? Experiment with several tours of various sizes.

17.12 Another heuristic is called the nearest insertion algorithm. First find the pair of cities that are closest to each other. Next find the unvisited city nearest to either of these two cities and insert it between them. Continue to find the unvisited city with minimum distance to some city in the partial tour, and insert that city between a pair of cities already in the tour so that the insertion causes the minimum increase in total length of the partial tour.

(a) Write a program to implement this algorithm. Compare its performance to that of the programs in the text. What is the execution time? How good or bad is the approximate solution that is generated? Experiment with several tours of various sizes.

(b) Compare the performance of this program to one that implements the nearest neighbor heuristic (Exercise 17.11).

17.13 A third traveling salesman heuristic is to partition the plane into strips, each of which contains some bounded number `B` of cities. Worker processes in parallel find minimal cost tours from one end of the strip to the others. In odd-numbered strips the tours should go from the top to the bottom; in even-numbered strips they should go from the bottom to the top. Once tours have been found for all strips, they are connected together.

(a) Write a program to implement this algorithm. Compare its performance to that of the programs in the text. What is the execution time? How good or bad is the approximate solution that is generated? Experiment with several tours of various sizes.

(b) Compare the performance of this program to one that implements the nearest neighbor heuristic (Exercise 17.11). Which is faster? Which gives a better solution?

17.14 Research heuristic algorithms and local optimization techniques for solving the traveling salesman problem. Start by consulting References [34] and [27]. Pick one or more of the better algorithms, write a program to implement it, and conduct a series of experiments to see how well it performs (both in terms of execution time and how good an approximate solution it generates).

17.15 The eight-queens problem is concerned with placing eight queens on a chess board in such a way that none can attack another. One queen can attack another if they are in the same row or column or are on the same diagonal.

Develop a parallel program to generate all 92 solutions to the eight-queens problem. Use a shared bag of tasks. Justify your choice of what constitutes a task. Experiment with different numbers of workers. Explain your results.

*This page intentionally left blank*

Chapter 18

# A DISTRIBUTED FILE SYSTEM

The three previous chapters presented examples of parallel programs. There the purpose of each program was to compute a result for a given set of input. In this chapter we present an example of a distributed program in which one or more users repeatedly interact with the program. This kind of program is sometimes called a *reactive* program since it continuously reacts to external events. At least conceptually, the program never terminates.

Our specific example is a program, which we call DFS, that consists of a distributed file system and a user interface. DFS executes on one or more host computers. Each host provides a simple file system. Users interact with DFS through a command interpreter, which is modeled on UNIX and supports commands to create, examine, and copy files. Users identify files located on remote hosts by using names that include host identifiers; these have the form `hostid:filename`. Thus DFS is similar to what is called a *network* file system. A user can log in to the system from any host and manipulate files on all hosts. A user's files on different hosts can differ; DFS does not provide a replicated file system.

In this chapter we first give an overview of the structure of DFS. Then we present the implementations of the file system and user interface. The program employs the client/server process interaction pattern that is prevalent in distributed systems. It also illustrates several aspects of JR: multiple virtual machines, operation types, dynamic object creation, UNIX file and terminal access, and the forward and reply statements. Our main purpose is to illustrate how to program this kind of distributed system. Consequently, our implementation of DFS does some error checking, but it is by no means all that one would desire. Our DFS implementation relies on some UNIX-specific file I/O features, so it will not work on non-UNIX platforms. Unlike the previous chapters

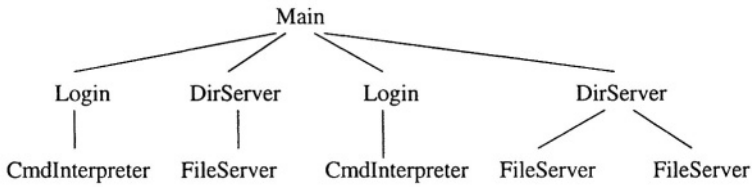


*Figure 18.1.* Snapshot of the structure of DFS

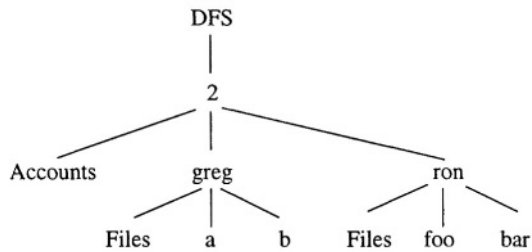


*Figure 18.2.* Underlying UNIX file structure for DFS logical host number 2

in this part, each of which presents several solutions to a given problem, this chapter outlines only one way to program a distributed file system.

## 18.1 System Structure

Our program for DFS consists of five key classes.

| | |
|---|---|
| `Main` | creates the directory and login servers |
| `Login` | handles login protocol and creates command interpreters |
| `CmdInterpreter` | implements commands to operate on files |
| `DirServer` | manages the files stored on one host |
| `FileServer` | provides access to an open file |

The main class creates one virtual machine on each host machine. On each virtual machine, `Main` then creates one instance of `DirServer` and one instance of `Login` for each terminal that can be used to talk to that host.

When a user successfully logs in, the corresponding instance of `Login` creates an instance of `CmdInterpreter`. Thus at any point in time there are as many instances of `CmdInterpreter` as there are active users.

To manipulate a file, a command interpreter first interacts with the instance of `DirServer` on the target machine (i.e., the one on which the file resides). Access to a file is provided by an instance of `FileServer`. A directory server creates a new instance of `FileServer` every time it opens a file for a command interpreter. The command interpreter then interacts directly with the file server to read and/or write data. When the file is closed, the file server terminates.

Figure 18.1 gives a snapshot of the structure of one possible execution of the DFS program. It assumes there are two host machines and that each has

one terminal for user interaction. In the illustration there are two instances of `CmdInterpreter`, which means there are two active users. There are also three instances of `FileServer`, which means three files are being accessed. For example, the user on the leftmost machine might be copying a file from that machine to the other one (which uses two file servers), and the user on the rightmost machine might be creating or reading a local file (which uses one file server).

To store system data and user files in DFS, we make use of the underlying UNIX file system. Before running DFS, we first need to create a directory named `DFS` in the user's home directory on each host machine. (The DFS program creates virtual machines that will execute in that directory; see Section 10.8.) The `DFS` directory should contain one file, `Accounts`, that contains a list of the names of authorized users of DFS. (The names of these users are arbitrary; they need not correspond to real users in the underlying UNIX file system.) The `DFS` directory also contains a subdirectory named for each possible logical host in DFS. Hosts in DFS are named 0,1,... up to whatever maximum number the user chooses. Each host subdirectory contains one subdirectory for each user. Each such subdirectory will be used by DFS to store a user's files. In addition, DFS employs one additional file, `Files`, in each user subdirectory; this contains a list of the names of the user's DFS files. For simplicity DFS assumes that the `DFS` directories and their files and subdirectories described above exist already; it does not create them as needed.

Figure 18.2 gives an example of the UNIX directory structure used by DFS on one host machine, the host numbered 2. There are two users and each has two files. The structure on other hosts will be similar: each host subdirectory contains a `Accounts` file and a subdirectory for each user, which in turn contains a `Files` file. However, the particular user files stored on different hosts will, in general, differ.

The design of DFS uses logical host numbers, rather than physical host names, to be more flexible. DFS is likely to be run in an instructional setting where the underlying file system is NFS (Network File System, which provides the same file system on each system). Using logical host numbers provides a simple way to effect separate directories for each DFS host in such a system. This structure even allows DFS to be run on a single system (with different windows representing terminals on different systems). The code in this chapter does just that, but can be easily extended to run on multiple systems (see Exercise 18.3). In fact, the DFS structure can even use just one virtual machine with very minor changes to the code. It also allows DFS to run on a collection of systems in a non-NFS environment. Each physical machine is given a portion of the entire DFS structure, namely the `Accounts` file and the subdirectory corresponding to one logical host. The main class then sets up the logical

to physical host mapping by placing each virtual machine on the appropriate physical machine.

## 18.2 Directory and File Servers

Each instance of the directory server class manages the DFS files on one host. The class provides public operations to create a new file, open an existing file, remove a file, determine whether a user has an account, obtain the names of a user's files, and update the user's directory at the end of program execution.

`DirServer` implements these operations. First, however, its constructor code reads in the `Accounts` file stored in the local machine's `DFS` directory and reads in the list of each user's files. The `fopen` and `check` operations are serviced by methods since they do not need to execute with mutual exclusion. Multiple invocations of these operations can be serviced concurrently. However, the other three operations update shared variables and need to execute with mutual exclusion, so they are serviced by an input statement in a process. A forward statement is used at the end of the body of the `fcreate` operation. By using `forward` we avoid having the `ds` process delay until the file server's open operation has completed; this enables `ds` to service other requests while the file is being opened.

```
import java.io.*;
import java.util.*;
public class DirServer {
  public op FileDesc fcreate(String, String, String);
  public op boolean fremove(String, String);
  public op String [] list(String);

  private int myhost;
  private String hostpath; // path of files for this host
  private HashMap accounts = new HashMap();
  private final String FakeFileAttr = "x";
  // store each file as (name,FakeFileAttr)  pair.
  // (later, might want to store real file attributes.)

  public DirServer() {
    myhost = ((Myvm)(vm.thisvm)).GetHost();
    readAccounts();
    readFiles();
    hostpath = "DFS/"+myhost+"/";
  }

  // read in the names of users
  private void readAccounts() {
    BufferedReader br = null;
    FileReader fr = null;
    try {
```

```
      fr = new FileReader("DFS/Accounts");
      br = new BufferedReader(fr);
    } catch(java.io.FileNotFoundException e) {
      System.err.println("can't open DFS/Accounts for "+myhost);
      JR.exit(1);
    }
    try {
      String ln;
      while( (ln = br.readLine()) != null ) {
        if (accounts.put(ln, new HashMap()) != null) {
          System.err.println(
            "ignoring duplicate user in Accounts "+ln);
        }
      }
      fr.close();
    } catch(java.io.IOException e) {e.printStackTrace(); }
  }
  // read in the names of users' files
  private void readFiles() {
    Set a = accounts.keySet();
    Iterator i = a.iterator();
    while (i.hasNext()) {
      String user = (String) i.next();
      BufferedReader br = null;
      FileReader fr = null;
      String dir = "DFS/"+myhost+"/"+user+"/Files";
      try {
        fr = new FileReader(dir);
        br = new BufferedReader(fr);
        try {
          HashMap u = (HashMap) accounts.get(user);
          String ln;
          while( (ln = br.readLine()) != null ) {
            // store file if not duplicate
            if (u.put(ln, FakeFileAttr) != null) {
              System.err.println(
                "ignoring duplicate file in Files "+ln+" "+user);
            }
          }
          fr.close();
        } catch(java.io.IOException e) {e.printStackTrace(); }
      } catch(java.io.FileNotFoundException e) {
        // user has no directory.
        System.err.println("user has no directory "+dir);
      }
    }
  }

  public op FileDesc fopen(String user, String fname, String m) {
    remote FileServer fs = new remote FileServer(hostpath);
```

```
  forward fs.fopen(user, fname, m);
}
public op boolean check(String user) {
  return accounts.containsKey(user);
}
process ds {
  while (true) {
    inni FileDesc fcreate(String user, String fname,
                          String m) {
      // look up user in accounts
      HashMap u = (HashMap) accounts.get(user);
      // if necessary, add file to database
      if (u.get(fname) == null) {
        u.put(fname,FakeFileAttr);
      }
      // create file server and forward open to it
      remote FileServer fs =
        new remote FileServer(hostpath);
      forward fs.fopen(user, fname, "WRITE");
    }
    [] String [] list(String user) {
      Set files = ((HashMap) accounts.get(user)).keySet();
      String [] res = new String [files.size()];
      Iterator i = files.iterator();
      for (int k = 0; k < files.size(); k++) {
        res[k] = (String) i.next();
      }
      return res;
    }
    [] boolean fremove(String user, String fname) {
      // look up user in accounts
      HashMap u = (HashMap) accounts.get(user);
      // if necessary, remove file from database
      boolean res;
      if ((res = u.containsKey(fname))) {
        u.remove(fname);
      }
      return res;
    }
  }
}

// write the directories back to Files
public op void writeFiles() {
  Set a = accounts.keySet();
  Iterator i = a.iterator();
  while (i.hasNext()) {
    String user = (String) i.next();
    String dir = "DFS/"+myhost+"/"+user+"/Files";
    PrintWriter pw = null;
```

```
      try {
        pw = new PrintWriter(
               new BufferedWriter(
                 new FileWriter(dir)));
      } catch (Exception e) {e.printStackTrace();}
      Set files = ((HashMap) accounts.get(user)).keySet();
      Iterator fi = files.iterator();
      while (fi.hasNext()) {
        String file = (String) fi.next();
        pw.println(file);
      }
      pw.close();
    }
  }
}
```

The `writeFiles` method in `DirServer` writes out each user's list of files; it is invoked when execution of DFS terminates. This way the same set of files is accessible the next time DFS is executed on the same hosts.

One instance of `FileServer` is created each time a file is opened. It provides just one public operation, `fopen`, which is serviced by a method. The body of the method declares three local operations, which are invoked to read, write, and close the file. The code contains separate, but similar cases for reading and writing a file. When `fopen` is called, it constructs a record containing capabilities for its local operations (assuming the file can be opened successfully). It assigns these to return variable `fd` and then executes a reply statement. At this point the client process that invoked `fopen` can proceed, and the remainder of the body of `fopen` continues executing as an independent server process. These two processes then engage in a conversation in which the client reads and writes the file. Eventually, the client invokes the `cl` (close) operation, at which point the file server closes the file and then terminates.

```
import java.io.*;
public class FileServer {
  String hostpath;
  public FileServer(String hostpath) {
    this.hostpath = hostpath;
  }

  public op FileDesc fopen(String user, String fname, String m) {
    String filepath = hostpath+user+"/"+fname;
    if (m.equals("READ")) {
      // local operations
      op  FReadInfo rd(); op void cl();
      BufferedReader br = null;
      FileReader fr = null;
      try {
        fr = new FileReader(filepath);
```

```
      br = new BufferedReader(fr);
    } catch(java.io.FileNotFoundException ie) {
      System.err.println("can't open file");
      try {
        return new FileDesc(null, null, null);
      } catch (Exception e) {e.printStackTrace();}
    }
    reply new FileDesc(rd, null, cl);
    while (true) {
      inni  FReadInfo rd() {
        try {
          String ln = null;
          try {
            ln = br.readLine();
          } catch(java.io.IOException e) {e.printStackTrace(); }
          return new  FReadInfo(ln, ln==null);
        } catch (Exception e) {e.printStackTrace();}
        return null;
      }
      [] void cl() {
        try {
          fr.close();
        } catch(java.io.IOException e) {e.printStackTrace(); }
        break;
      }
    }
  }
  else if (m.equals("WRITE")) {
    // local operations
    op void wr(String); op void cl();
    PrintWriter pw = null;
    try {
      pw = new PrintWriter(
               new BufferedWriter(
                   new FileWriter(filepath)));
    } catch(java.io.FileNotFoundException ie) {
      System.err.println("can't open file");
      try {
        return new FileDesc(null, null, null);
      } catch (Exception e) {e.printStackTrace();}
    } catch(java.io.IOException e) {e.printStackTrace(); }
    reply new FileDesc(null, wr, cl);
    while (true) {
      inni void wr(String ln) {
        pw.println(ln);
      }
      [] void cl() {
        pw.close();
        break;
      }
```

```
      }
    }
    else {
      System.err.println("fopen given unknown accessmode "+m);
      JR.exit(1);
      return null; // needed even though JR.exit doesn't return.
    }
  }
}
```

The `FileServer` makes use of two other simple classes for encapsulating multiple return values. The `FileDesc` (file descriptor) class represents the operations that can be performed on a file.

```
public class FileDesc implements java.io.Serializable {
  // capabilities for read, write, close
  // (for simplicity, can open only for R or W
  // so one cap isn't used)
  public cap FReadInfo () r;
  public cap void (String) w;
  public cap void () c;
  public FileDesc(cap FReadInfo () r,
                  cap void (String) w,
                  cap void () c ) {
    this.r = r;
    this.w = w;
    this.c = c;
  }
}
```

It must be serializable for reasons described in Section 10.7. `FileServer` declares file operations locally and returns capabilities for them as the result of opening a file. In turn the `FileDesc` (file descriptor) class uses the `FReadInfo` class for representing the information that a read operation returns.

```
public class FReadInfo {
  // return info from a read
  public String ln; // the line read
  public boolean gotEOF; // true iff EOF
  public FReadInfo(String ln, boolean gotEOF) {
    this.ln = ln;
    this.gotEOF = gotEOF;
  }
}
```

In our implementation of DFS, we employ one instance of `FileServer` for each open file, which is an appropriate abstraction. However, since the code does not contain any class variables, we could employ fewer instances. For example, we could create just one instance of `FileServer` on each host

machine. Then we could have each process that is accessing a file interact with a separate instance of a process executing the `fopen` method.

## 18.3 User Interface

The remaining key components of DFS are the user interface and the main class. When a user of DFS first sits down at a terminal, that user is interacting with an instance of the `Login` class. Each instance of `Login` reads from and writes to one terminal device. This is likely to be a window on a workstation (e.g., an `xterm` window). The `Login` class first opens the associated keyboard and display. (These are like two files that happen to have the same name.) Then `Login` waits for a user to attempt to log in to DFS. If the user is successful, `Login` creates an instance of the command interpreter and then waits for the command interpreter to terminate.

```
import java.io.*;
public class Login {
  private int myhost, nhosts;
  private String device;
  private String [] host;
  private remote DirServer [] dsref;

  private BufferedReader ttyin;
  private PrintWriter ttyout;

  public Login(String device,
               int nhosts, remote DirServer [] dsref ) {
    myhost = ((Myvm)(vm.thisvm)).GetHost();
    this.nhosts = nhosts;
    this.dsref = dsref;
    try {
      String tty = "/dev/" + device;
      FileReader fr;
      fr = new FileReader(tty);
      ttyin  = new BufferedReader(fr);
      ttyout = new PrintWriter(
                          new BufferedWriter(
                           new FileWriter(tty)),true);

    } catch(java.io.FileNotFoundException e) {
      System.err.println("can't open file");
      return;
    } catch (java.io.IOException e) {e.printStackTrace();}
  }
  private process prompt {
    op void done();
    ttyout.println("Welcome to DFS");
    ttyout.println();
    while (true) {
```

```
      ttyout.print("login: ");
      ttyout.flush();
      String user = "";
      try {
        user = ttyin.readLine();
      } catch (java.io.IOException e) {e.printStackTrace();}
      if (!dsref[myhost].check(user)) {
        ttyout.println("unknown user");
        continue;
      }
      remote CmdInterpreter ciref =
        new remote CmdInterpreter(user, ttyin, ttyout,
           nhosts, dsref, done);
      // wait for ciref to finish
      receive done();
    }
  }
}
```

The terminal devices used by the DFS program are typically running UNIX shell processes. The name of each of these terminal devices needs to be given to DFS when it begins execution (see the `Main` class later in this section). The name of each terminal device can be obtained by using the `tty` UNIX shell command. Furthermore, to prevent the shells from intercepting input intended for DFS, each shell process should be put to sleep before DFS executes by using the `sleep` UNIX shell command.

The command interpreter is the largest component of DFS. It implements two kinds of user commands: ones that deal with files and ones that deal with the current working directory. It also implements a logout command. These commands are modeled on UNIX commands. They are summarized in the following table:

| | |
|---|---|
| `cr filename` | creates a new file by entering text |
| `cat filename` | prints the contents of a file |
| `cp filename1 filename2` | copies the first file into the second |
| `rm filename` | removes a file |
| `ls [machine]` | lists the files in a user's directory |
| `cd [machine]` | changes current working directory |
| `pwd` | prints host number of current working directory |
| `exit` | logs out of a session |

The `machine` argument is optional in the `ls` and `cd` commands. The default for `ls` is the current working directory, and the default for `cd` is to change to the original home directory.

A file name has the general form `machine:filename`, where `machine` is a logical host number and `filename` is a file on that host. If the machine name (and colon) are omitted, a file name is interpreted relative to the current working directory.

The `CmdInterpreter` class is the client of the file and directory server classes. Its body implements the above user commands. Many of the commands have similar implementations, so we present below only part of the body of `CmdInterpreter`. The source for the complete implementation is included with the JR distribution.

The local methods implement details of the file access commands. For example, the `cmd_cr` method given below implements the file-creation command by first creating a new file on the designated server machine and then reading terminal input and writing it to that file. The end of the input is indicated by a line with a single dot.

The command-interpreter process, `CI`, repeatedly writes a prompt and then reads and interprets a command. It uses an if statement, with one part per known command, to search through the known commands. The else part catches all unrecognized commands. Our implementation of the command interpreter does a reasonable amount of error checking, but by no means all that one would, in general, desire to have.

```
import java.io.*;
import java.util.*;
public class CmdInterpreter {
  private int myhost, nhosts;
  private int curhost;
  private String device;
  private String [] host;
  private remote DirServer [] dsref;

  private String user;
  private BufferedReader ttyin;
  private PrintWriter ttyout;
  private cap void () done;

  public CmdInterpreter(String user,
         BufferedReader ttyin, PrintWriter ttyout,
         int nhosts, remote DirServer [] dsref,
         cap void () done ) {
    curhost = myhost = ((Myvm)(vm.thisvm)).GetHost();
    this.user = user;
    this.ttyin = ttyin;
    this.ttyout = ttyout;
    this.nhosts = nhosts;
    this.dsref = dsref;
    this.done = done;
  }

  // command "cr filename"
  private void cmd_cr(String fname, int server) {
    FileDesc fd = dsref[server].fcreate(user, fname, "WRITE");
```

```
    if (fd.w == null) {
      ttyout.println("cr: cannot create file " + fname);
      return;
    }
    // read from ttyin and write to fd
    ttyout.println(
      "Enter contents; Last line should be a single dot");
    String ln = "";
    try {
      ln = ttyin.readLine();
    } catch (java.io.IOException e) {e.printStackTrace();}
    while (!ln.equals(".")) {
      fd.w(ln);
      try {
        ln = ttyin.readLine();
      } catch (java.io.IOException e) {e.printStackTrace();}
    }
    fd.c();
  }

  // command "cat filename"
  private void cmd_cat(String fname, int server) {
    ...
  }
  // command "cp filename1 filename2"
  private void cmd_cp(String f1, int s1, String f2, int s2) {
    ...
  }
  // command "rm filename"
  private void cmd_rm(String fname, int server) {
    if (!dsref[server].fremove(user, fname)) {
      ttyout.println("rm: cannot remove file " + fname);
    }
  }

  private process CI {
    String cmdline = null;
    while (true) {
      ttyout.print("<"+myhost+">% ");
      ttyout.flush();
      try {
        cmdline = ttyin.readLine();
      } catch (java.io.IOException e) {e.printStackTrace();}
      if (cmdline == null) break; // got EOF
      StringTokenizer stok = new StringTokenizer(cmdline);
      String cmd;
      if (!stok.hasMoreTokens()) continue; // empty line
      cmd = stok.nextToken();
      String [] args;
      if (cmd.equals("cr")) { // create a new file
```

```
    args = getargs(cmd, stok, 1, 1);
    if (args != null) {
      CArgs c = crackarg(cmd, args[0]);
      if (c != null) {
        cmd_cr(c.fname,c.host);
      }
    }
  }
  else if (cmd.equals("rm")) { // remove a file
    args = getargs(cmd, stok, 1, 1);
    if (args != null) {
      CArgs c = crackarg(cmd, args[0]);
      if (c != null) {
        cmd_rm(c.fname,c.host);
      }
    }
  }
  else if (cmd.equals("cat")) { // print an existing file
    ...
  }
  else if (cmd.equals("cp")) { // copy one file to another
    ...
  }
  // list contents of current directory
  else if (cmd.equals("ls")) {
    args = getargs(cmd, stok, 0, 1);
    if (args != null) {
      int h;
      if (args.length == 0) {
        h = curhost;
      }
      else {
        h = checkhost(cmd, args[0]);
        if (h == -1) continue;
      }
      String [] files = dsref[h].list(user);
      for (int i = 0; i < files.length; i++) {
        ttyout.println(files[i]);
      }
    }
  }
  else if (cmd.equals("cd")) {
    ...
  }
  else if (cmd.equals("pwd")) {
    args = getargs(cmd, stok, 0, 0);
    if (args != null) {
      ttyout.println(curhost);
    }
  }
```

```
      else if (cmd.equals("exit")) {
        args = getargs(cmd, stok, 0, 0);
        if (args != null) {
          break;
        }
      }
      else {
        ttyout.println("invalid command " + cmd);
      }
    }
    ttyout.println();
    // tell Login that this session is done
    send done();
  }
  // break args on command line into array of Strings;
  // check whether enough args for given command, cmd.
  private String [] getargs(String cmd, StringTokenizer stok,
                            int mina, int maxa) {
    int remain = stok.countTokens();
    if (remain < mina || remain > maxa) {
      ttyout.println(cmd + ": too few or too many arguments");
      return null;
    }
    String [] res = new String [remain];
    for (int a = 0; a < remain; a++) {
      res[a] = stok.nextToken();
    }
    return res;
  }
  // convert s to int and check that it's a valid host number
  // return -1 if not
  private int checkhost(String cmd, String s) {
    int h = Integer.parseInt(s);
    if (h < 0 || h >= nhosts) {
      ttyout.println(cmd + ": invalid host "+h);
      h = -1;
    }
    return h;
  }
  // "crack" arg of form [hname:]fname
  private CArgs crackarg(String cmd, String s) {
    CArgs res = new CArgs();
    StringTokenizer stok = new StringTokenizer(s,":");
     if (stok.countTokens() == 1) {
      res.host = curhost;
      res.hname = "";
      res.fname = s;
    }
    else {
      res.hname = stok.nextToken();
```

```
      res.host = checkhost(cmd, res.hname);
      if (res.host == -1) {
        res = null;
      }
      else {
        res.fname = stok.nextToken(); // assumes only 1 :
      }
    }
  return res;
  }
}
```

The next component of DFS is the main class, which gets everything started. It first reads the command-line arguments, which specify the number of host machines to use. Next `Main` creates one virtual machine for each logical host and one directory server on each virtual machine. `Main` then prompts for the names of each terminal that is to be used. After `Main` creates one instance of `Login` for each terminal, DFS is operational.

The terminal used to initiate execution of DFS is not one of the terminals within the system itself. Rather it serves as an "operator's console." The operator stops execution of DFS by entering a string on the console. Upon reading the string, `Main` determines whether the user intends to save the results of the current session; if so, it invokes methods in the directory servers to save DFS's record of user files.

```
import java.io.*;
public class Main {
  public static void main(String [] args) {
    System.out.println("Welcome to DFS! (system console)");
    int nhosts = Integer.parseInt(args[0]);
    vm vmref [] = new vm [nhosts];
    remote DirServer [] dsref;

    // create vms, one per host; each will execute in home directory.
    for (int h = 0; h < nhosts; h++) {
      vmref[h] = new Myvm(h);
    }

    // create directory server for each host
    dsref = new remote DirServer[nhosts];
    for (int h = 0; h < nhosts; h++) {
      dsref[h] = new remote DirServer() on vmref[h];
    }

    // prompt for names of terminal devices and
    // create a login server for each terminal
    BufferedReader br;
    InputStreamReader isr = new InputStreamReader(System.in);
```

```
    br = new BufferedReader(isr);
    for (int h = 0; h < nhosts; h++) {
      System.out.print("number of ttys for host "+h+":");
      System.out.flush();
      String s = "";
      try {
        s = br.readLine();
      } catch (java.io.IOException e) {e.printStackTrace();}
      int nttys = Integer.parseInt(s);
      System.out.println("enter tty names");
      for (int t = 1; t <= nttys; t++) {
        try {
          s = br.readLine();
        } catch (java.io.IOException e) {e.printStackTrace();}
        new remote Login(s, nhosts, dsref) on vmref[h];
      }
    }
    System.out.println("DFS now executing; enter string to stop");

    // see whether user wants to save DFS Files.
    boolean save = true;
    String leave = "";
    try {
      leave = br.readLine();
    } catch (java.io.IOException e) {e.printStackTrace();}
    if (!leave.equals("save")) {
      System.out.println("save before exiting? (y/n)");
      System.out.flush();
      leave = "n";
      try {
        leave = br.readLine();
      } catch (java.io.IOException e) {e.printStackTrace();}
      save = !leave.equals("n");
    }

    // if don't save, files that were created during session
    // will not appear in DFS Files (but will appear
    // in underlying UNIX directory).
    if (save) {
      System.out.println("saving...");
      System.out.flush();
      for (int h = 0; h < nhosts; h++) {
        dsref[h].writeFiles();
      }
    }
  }
}
```

The final class is the parameterized virtual machine class, `Myvm`. It just provides a `GetHost` method, which returns the DFS host number.

```
public class Myvm extends vm {
  private int myhost;
  public Myvm(int myhost) {
    this.myhost = myhost;
 }
 public op int GetHost() {
   return myhost;
 }
}
```

## Exercises

18.1 The text gave one set of activities that results in the structure shown in Figure 18.1. Give another.

18.2 Run the DFS program. Use at least two logical hosts and at least one terminal window per machine. Experiment interacting with DFS and report on your experience. What features do you like, and why? What features do you not like, and why? What features do you miss having, and why?

18.3 Modify the DFS program to execute on multiple physical hosts. For example, read the host names from a file or from the system console. Then experiment and report with your modified DFS program as described in the previous question.

18.4 Suppose the same user is logged in more than once on DFS. What happens if the user tries simultaneously to access the same file from more than one terminal? Consider each combination of file-access commands (`cr`, `cat`, `cp`, and `rm`).

18.5 In the `DirServer` class, the `list` operation is serviced by an input statement in process `ds`. Suppose `list` were serviced by a method instead. Carefully explain what could go wrong.

18.6 In the `DirServer` class, modify the `list` operation so that it returns the set of file names, instead of an array of strings. Make necessary changes to where `list` is used in `CmdInterpreter`. To do so, the particular set representation you choose (from the `java.util` package) needs to be serializable. Explain why.

18.7 The DFS implementation of `rm` does not actually remove files from the underlying file system. Modify DFS to do so. (Be careful not to make programming mistakes that unintentionally remove non-DFS files!)

18.8 If the user on the system console decides not to save a session, the DFS implementation does not restore the DFS structure to exactly what

it was before the session. With respect to the underlying file system, newly created files are not deleted and files deleted during the session are not restored. Modify DFS to do so. (Be careful of programming mistakes that unintentionally remove non-DFS files!)

18.9 Extend DFS so users can access each other's files. Define some protection scheme so that a user can control the way in which other users access his files.

18.10 Add commands and functionality to DFS, such as who and wc commands, a cat command with multiple arguments, tree-structured directories, pipes, I/O redirection, etc.

18.11 DFS is static with respect to the users and hosts that participate in a session. Add "operator"commands and functionality to DFS, such as to add or delete a user, to add or delete a host, etc. during DFS execution.

18.12 Add file caching to DFS.

18.13 Add automatic replication to DFS, so a user's files are the same on all hosts. (This exercise is interesting only if DFS is running in a non-NFS environment!)

18.14 Add file locking to DFS to prevent several users from concurrently updating the same file. Permit concurrent reading, however.

18.15 Rewrite the DFS classes so they do not use parameterized virtual machines.

*This page intentionally left blank*

Chapter 19

# DISCRETE EVENT SIMULATION

A discrete event system is one in which state changes, or events, occur at discrete instants of time. The arrivals and departures of buses and passengers at a bus stop, for example, can be represented as such a system. The movements of planes on runways and between airports is another example of something that can be modeled as a discrete event system. In contrast, the flow of air over the wing of an aircraft cannot be modeled as a discrete system since the system state changes continuously. (Continuous systems can often be modeled by partial differential equations and hence simulated as shown in Chapter 16.)

A discrete event simulation is a program that models a discrete event system. The main components in a discrete event simulation are *simulation processes,* which represent active objects such as people and buses; *resources,* which represent passive objects such as a bus stop; and an *event scheduler,* which controls the order in which simulation activities occur.

Concurrent programming languages are well suited for programming discrete event simulations because processes in concurrent programs correspond closely to simulation processes. JR is especially well suited because its rich collection of synchronization mechanisms makes the interactions between the simulation components easy to program.

This chapter presents a simple discrete event simulation problem and describes a JR solution to it. The problem and a solution programmed in Ada originally appeared in Reference [13]. The presentation in this chapter is based on Reference [39].

## 19.1 A Simulation Problem

The specific problem we consider here is simulating one aspect of a simple multiprocessor architecture. Several processors compete to access a common memory bus. Each processor cyclically seizes the bus, transfers data on the

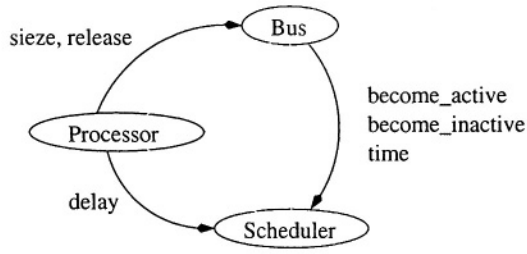


*Figure 19.1.* Simulation component interaction pattern

bus, releases the bus, and then performs some other activity. Each processor is a simulation process, and the bus is a simulation resource. The purpose of the simulation is to gather statistics on bus utilization and on delays encountered by the processors.

We use one class to implement each simulation component. The main class sets the simulation parameters, starts up the other parts of the simulation, and shuts down the simulation when it has run long enough. The `Processor` class contains one process for each processor in the system. The `Bus` class implements the data bus. It provides public `seize` and `release` operations, which the (simulated) processors call to seize and release the bus; it also makes available a `print` operation, which is used to print the statistics `Bus` maintains.

The `Scheduler` class implements the event scheduler. In particular, it maintains the simulation clock and event list. When no (simulated) processors are active, the scheduler picks the next event from the event list and updates the simulation clock to that event's time.

The `Scheduler` provides four public operations. The processors call `delay` to simulate the passage of time during data transfers and other activity; the end of each such time period defines an event. The bus controller calls `become_inactive` to inform the scheduler that a processor has been blocked in its attempt to seize the bus. The bus controller calls `become_active` when the processor subsequently obtains access. These operations return the value of the simulation clock so that statistics can be gathered. This interaction between the scheduler and bus controller allows the scheduler to maintain a count of active processors. Bus controllers call the final `Scheduler` operation, `time`, to get the value of the simulation clock.

Figure 19.1 illustrates the interaction between the three simulation components. The details of the `Processor` and `Bus` classes are specific to this example. However, the `Scheduler` class provides functionality that is commonly required in any discrete event simulation. It is an abstract data type that could be reused; it could also be extended with additional functionality.

## 19.2 A Solution

This section presents and discusses these four classes. For pedagogic reasons, we describe the classes in the following order: `Main`, `Processor`, `Bus`, and `Scheduler`.

### 19.2.1 Main Class

The following is the simulation program's main class:

```
public class Main {
  public static void main(String [] args) {
  // default values
    int NUM_PROCESSORS = 3;
    double TIME = 1000.0;

    remote Scheduler sched;
    remote Bus       bus;
    // create instances of Scheduler, Bus, and Processor
    sched = new remote Scheduler(NUM_PROCESSORS+1);
    bus   = new remote Bus(sched);
    new remote Processor(NUM_PROCESSORS, sched, bus);
    // run the simulation for TIME clock ticks
    sched.delay(TIME);
    // print usages from bus, then stop the simulation
    bus.print();
  }
}
```

Program execution begins in the main method. The code first sets the default number of processors and the default length of time to run the simulation. (In practice, it should allow optional command-line arguments to override those values.) Then it creates one `Scheduler` object, one `Bus` object, and one `Processor` object, assigning references for the first two to `sched` and `bus`, respectively. `sched` is passed to `bus` when it is created so `bus` can invoke `sched`'s operations. For similar reasons, `sched` and `bus` are passed to the `Processor` object.

The main method code lets the simulation run for `TIME` simulation clock ticks by delaying itself for that long. After that delay it requests that statistics on bus utilization be output and then stops the simulation.

### 19.2.2 Processor Class

The `Processor` class implicitly creates `NUM_PROCESSORS` instances of `processor`, as shown in the following code.

```
public class Processor {
  int n; // number of processors
  remote Scheduler sched;
```

```
  remote Bus       bus;
  public Processor(int n, remote Scheduler sched, remote Bus bus) {
    this.n = n;
    this.sched = sched;
    this.bus = bus;
  }
  process processor( ( int i = 1; i <= n; i++ ) ) {
    while(true) {
      bus.seize();                      // grab the bus
      sched.delay(10.0*Math.random()); // use the bus
      bus.release();                    // release the bus
      sched.delay(20.0*Math.random()); // do something else
    }
  }
}
```

Each `processor` process interacts with `bus` and `sched` to simulate seizing the bus, using the bus for a random amount of time, releasing the bus, and then performing some other activity for a different random amount of time. The upper bounds on the delay intervals are fixed in the above program; in practice they too should be command-line arguments.

### 19.2.3 Bus Controller Class

The `Bus` class provides public `seize`, `release`, and `print` operations. It is programmed as follows:

```
public class Bus {
  public op void seize();
  public op void release();
  public op void print();
  private op void try_seize(cap void () go_ahead);

  private remote Scheduler sched;

  public Bus(remote Scheduler sched) {
    this.sched = sched;
  }
  private process bus_manager() {
    // nreq is number of bus requests from processors
    // bus_time is total time bus is in use
    // wait_time is total waiting time of processors
    // free indicates whether the bus is free
    int nreq = 0;
    double bus_time = 0.0;
    double wait_time = 0.0;
    boolean free = true;
    // queue of processors blocked on seizing
    op void block_list(cap void ());
    while( true ) {
```

```
      inni void try_seize(cap void() go_ahead) {
        nreq++;
        if (free) {
          free = false;
          bus_time -= sched.time();
          send go_ahead();
        }
        else {
          wait_time -= sched.become_inactive();
          send block_list(go_ahead);
        }
      }
      [] void release() {
        if (block_list.length() > 0) {
          // awaken a blocked processor
          wait_time += sched.become_active();
          cap void () go_ahead;
          receive block_list(go_ahead);
          send go_ahead();
        }
        else { // no processor waiting
          free = true;
          bus_time += sched.time();
        }
      }
      [] void print() { // compute and output statistics
        double now = sched.time();
        // average wait time and percent bus use
        double avg_wait, pct_bus;
        avg_wait = (wait_time+now*block_list.length())/nreq;
        if (free) {
          pct_bus = 100*bus_time/now;
        }
        else {
          pct_bus = 100*(bus_time+now)/now;
        }
        System.out.println( "avg_wait="+avg_wait
                            +" pct_bus="+pct_bus);

      }
    }
  }
  // provide a simple call interface for processors
  public void seize() {
    op void go_ahead();
    send try_seize(go_ahead);
    receive go_ahead();
  }
}
```

The `seize` operation is serviced by a method, which hides the fact that seizing the bus requires sending a `try_seize` message and receiving a `go_ahead` message. The `go_ahead` operation is declared local to `seize`, so that a new instance of `go_ahead` is created for each instance of `seize`. A capability for `go_ahead` is sent to `try_seize`. That `go_ahead` capability will be invoked when the associated seize request can be satisfied, thus allowing `seize` to complete, and its caller to continue.

The `try_seize`, `release`, and `print` operations are repeatedly serviced by an input statement in the background process `bus_manager`. The `try_seize` operation determines if the bus is free. If so it allocates the bus and allows the invoker to proceed by invoking `go_ahead`. If not it saves the capability for the invoking process's `go_ahead` operation. The `release` operation awakens a waiting process, if one is present, by sending to that process's `go_ahead` operation; `release` marks the bus as free if no process is waiting. The `try_seize` and `release` operations also gather information on bus use and processor waiting time. The `print` operation outputs statistics based on that information.

The `bus_manager` process maintains two local variables `bus_time` and `wait_time` that record the total time the bus is in use and the total time that processes have waited to obtain access to the bus, respectively. One way to view `bus_time` is as the sum of the lengths of the intervals during which the bus is in use. The endpoints of such intervals are the simulation clock values of when the bus became busy and when it became free again; the lengths are then the differences in these two clock values. Thus process `bus_manager` *subtracts* the simulation clock from `bus_time` when the bus becomes busy, and it *adds* the simulation clock to `bus_time` when the bus later becomes free. The effect of this subtraction and addition is that `bus_time` is incremented by the length of the interval during which the bus is in use. Process `bus_manager` similarly maintains the variable `wait_time`.

The `bus_manager` maintains the queue of `go_ahead` capabilities for waiting processes by using the local `block_list` operation. An element is added to the end of the queue by sending to `block_list`. An element is removed from the front of the queue by receiving from `block_list`. The number of processes currently blocked in their attempts to seize the bus is therefore the number of pending invocations on `block_list`, i.e., the value of `block_list.length()`.

### 19.2.4 Scheduler Class

The final class, `Scheduler`, provides four public operations. Its code is as follows:

```
public class Scheduler {
  public op double become_active();
  public op double become_inactive();
```

```
public op double time();
public op void   delay(double);

private op void event_list(cap void () go_ahead, double t);
private double clock = 0.0;
private int num_tasks;

public Scheduler(int num_tasks) {
   this.num_tasks = num_tasks;
}
private process event_manager() {
  // number of active simulation processes
  int active = num_tasks;
  while (true) {
    inni double become_active() {
      active++;
      return clock;
    }
    [] double become_inactive() {
      active--;
      return clock;
    }
    [] double time() {
      return clock;
    }
    [] void event_list(cap void () go_ahead, double t)
                st active == 0 by t {
      clock = t;
      active++;
      send go_ahead();
      // awaken any other processes scheduled
      // for this same time
      while (true) {
        inni void event_list(cap void () go_ahead2,
                                         double t2)
                        st t2 == clock {
          active++;
          send go_ahead2();
        }
        [] else { // no more -- exit the while loop
          break;
        }
      }
    }
  }
}
// provide a simple call interface for processors
public void delay(double t) {
  op void go_ahead();
  send event_list(go_ahead,t+clock);
```

```
    send become_inactive();
    receive go_ahead();
  }
}
```

The `delay` operation is serviced by a method for much the same reason that `seize` in the `Bus` class is serviced by a method. Here the method hides the fact that delaying requires two sends and a receive.

The other four operations—`become_active`, `become_inactive`, `time`, and `event_list`—are repeatedly serviced by an input statement in the background process `event_manager`. The code for both `become_active` and `become_inactive` simply updates the number of active simulation processes (`active`) and returns the value of the simulation `clock(clock)`. The code for `time` returns the value of `clock`.

Process `event_manager` services an invocation of `event_list` when no simulation processes are active. It advances the simulation to the next event on the event list and activates the associated simulation process. The guard on `event_list` uses a synchronization expression to ensure that no simulation processes are active, and it uses a scheduling expression to select from the event list the invocation with the smallest time `t`. The code for `event_list` then sets the simulation clock to the time in that invocation, increments `active`, and sends a `go_ahead` signal to the process associated with the invocation so that it may continue. Process `event_manager` then checks the event list for other events scheduled for the same time. Each iteration of `event_list`'s loop removes an invocation whose recorded time is identical to the current time, if one is present. The inner input statement employs a synchronization expression to select appropriate invocations. For each such invocation, the code increments `active` and sends the `go_ahead`, as above. The loop exits when the event list has no invocations with the current time.

## 19.3 Observations

The code in the `Bus` and `Scheduler` classes employs a technique of using the implicit queues of pending invocations associated with message passing instead of using explicit, programmer-defined queues. Thus operations are similar to data-containing semaphores (see Section 7.6). Using this technique can lead to more concise solutions to problems involving lists.

The `block_list` operation in `Bus`'s event manager process is used to maintain the list of processes that are blocked trying to seize the bus. Similarly, the `event_list` operation in `Scheduler` is used to maintain the list of events scheduled to occur in the future. However, these two operations are used differently. The `block_list` operation is local to, and is invoked only within, the `event_manager` process in `Bus`. On the other hand, `event_list` is global to `Scheduler` and multiple instances of `delay` append (send) elements to

`event_list`. Processes accessing `event_list` are automatically synchronized. If the event list were coded instead to use an explicit, programmer-defined queue, then the processes accessing the queue would need to be synchronized explicitly.

Programs written in a language, such as JR, that provides a variety of synchronization mechanisms are in many cases simpler than those written in languages that provide only one form of synchronization. The simulation program presented in this chapter employs both rendezvous and asynchronous message passing. For example, consider how a process is delayed when it attempts a `seize` that cannot be satisfied immediately. It sends a message and waits for a `go_ahead`. The `bus_manager` sends the `go_ahead` when the bus becomes free; it does not need to respond immediately, as it would if it instead used a rendezvous. Of course, this kind of interaction can be programmed using rendezvous alone, but it is cumbersome (see Exercise 19.4).

The `Scheduler` class illustrates the use of two other interesting JR language features not found in many concurrent programming languages. First, invocation parameters can be used in synchronization expressions. For example, the second input statement that services `event_list` in the `Scheduler` class (see Section 19.2) selects invocations whose time parameter matches the current simulation clock. Second, invocations can be selected in an order dictated by their parameters. For example, the first input statement that services `event_list` in `Scheduler` uses a scheduling expression to select the invocation with the smallest time. These features of JR contribute to more concise solutions to many programming problems, of which discrete event simulation is just one example.

## Exercises

19.1 Why does the main method (see Section 19.2) pass `NUM_PROCESSORS+1` instead of just `NUM_PROCESSORS` when it creates `Scheduler`.

19.2 The program in this chapter uses `JR.exit` to terminate. At that time, processes in `Processor`, `Bus`, and `Scheduler` still exist. Modify the code so that the main method first explicitly terminates those processes before using `JR.exit` to terminate the entire program.

19.3 Suppose that the simulation program did not need to gather statistics. Show how that would simplify the code.

19.4 Program the bus controller class `Bus` (see Section 19.2) using only rendezvous for synchronization. (Have fun!)

19.5 Program the inner input statement in class `Scheduler` (see Section 19.2) without using a synchronization expression.

19.6 Program the outer input statement in `Scheduler` without using a scheduling expression.

19.7 The `delay` operation in `Scheduler` is serviced as a separate method. Can it be serviced as an arm of the input statement in process `event_manager`? Explain your answer.

19.8 The `Scheduler` variable `clock` is shared between `event_manager` and instances of `delay`. Explain why it is safe for `delay` to add `clock` to t.

19.9 Consider the `delay` method in `Scheduler`. Describe the effects of the following:

(a) Changing the first send (only) to a call.

(b) Changing the second send (only) to a call.

(c) Changing both sends to calls.

19.10 Explain why it would be easier to replace the `block_list` operation (in `Bus`) by an explicit, programmer-defined queue than it would be to do the same for `event_list` (in `Scheduler`).

19.11 Devise and execute various timing tests to determine the relative costs of maintaining a queue shared by several processes as an operation and as a programmer-defined linked list with explicit synchronization.

19.12 Show how to use an operation to maintain a stack.

19.13 Program a discrete event simulation that represents a simple cafeteria. Customers obtain food from a single food server and then pay for it at a single cashier. They then eat before repeating their activities. Reuse as much of the code presented in this chapter as you can.

19.14 Program a discrete event simulation that models the distributed solution to the Dining Philosophers Problem (see Section 11.2). Use it to gather statistics on fork utilization and waiting times.

19.15 Program a simulation of passengers and buses arriving and departing from a bus stop.

19.16 Program a simulation of traffic on a grid of city streets. Assume there is a stop light at each intersection.

19.17 Program a simulation of the movements of planes on runways and between airports. Include gates from which planes depart and at which they arrive.

Chapter 20

# INTERFACING JR AND GUIs

Graphical user interfaces (GUIs) are becoming more prevalent, powerful, flexible, and easy to use. For example, the Swing and AWT toolkits enable Java programs to easily include GUIs. GUIs are also useful to visualize the executions of concurrent programs.

This chapter uses an example to illustrate how JR can be used with Swing. It introduces a primitive game called "Balls and Boxes", or BnB, for short. Although BnB as a game is neither particularly interesting nor polished, it does illustrate many of the useful pieces and a general structure that can be used in building more interesting JR programs that use GUIs. For details about Swing, Reference [48] and its online versions (`http://java.sun.com/docs/books/tutorial/uiswing/index.html` and `http://java.sun.com/docs/books/tutorial/uiswing/mini/index.html`) are extremely helpful.

## 20.1 BnB Game Overview

BnB is a multiplayer game intended to be played across several systems, with one player per system. Each player sees a single game window, which shows the game's status on only that system, and interacts with the game via the system's keyboard and mouse. However, BnB is designed so that it can be played by a single player on a single system too. On a single system, the game can display just a single window or multiple windows. In the latter case, user interaction is still via the keyboard and mouse according to which window has focus.

Figure 20.1 gives a screen snapshot of the game.[1] It shows two windows on a single system display. As seen, each window consists of a menu, a button, and a board area. Windows are assigned unique identifiers (0,1, 2,...) and are considered to be ordered left-to-right according to those identifiers. Each board area holds two kinds of "toys": balls and boxes. The menu and button are used to create new balls. Each menu click creates a single green or orange ball. Each button click creates a blue ball and a red ball. The balls and boxes move within the board area. Each board area is initially assigned a box. Only one box for each window exists during each game. The balls and boxes are labeled with the number of the window from which they originated. The board area also contains a text message, which simply moves randomly within the board area.

Balls move automatically left to right across the window, zig-zagging between the top and bottom. They "bounce" off the top or bottom, effecting a change in their upward or downward direction. Each ball's initial placement within its originating window is randomly chosen. When a ball reaches the right border, it moves to the next window on the right (according to the ordering on window identifiers) or wraps around to the first window. Each ball expires after it has moved a certain number of times.

Boxes move under user keyboard control. When a box reaches the left or right border, it moves to the previous or next window, respectively. Boxes cannot move beyond the top or bottom border. Each box lives for the duration of the game.

As noted in the introduction to this chapter, the game is fairly simple so as to illustrate the basic concepts in using Swing. Exercises 20.4 and 20.5 suggest how to make the game more interesting.

## 20.2 BnB Code Overview

The BnB program consists of the following classes:

`Main`: the main program. It creates all the windows in the game.

`Window`: represents a window (one for each player). It creates the graphical components it uses: a menu, a button, and a board.

`SwingApplication`: the button.

`Board`: the board. It creates and controls the toys on the board.

`Toy`: a base class (superclass) for all toys. Its extended classes (subclasses) are:

[1] Because this book is printed in black and white, the colors mentioned in this section do not appear in the figure. A color version of this figure is available on the JR webpage

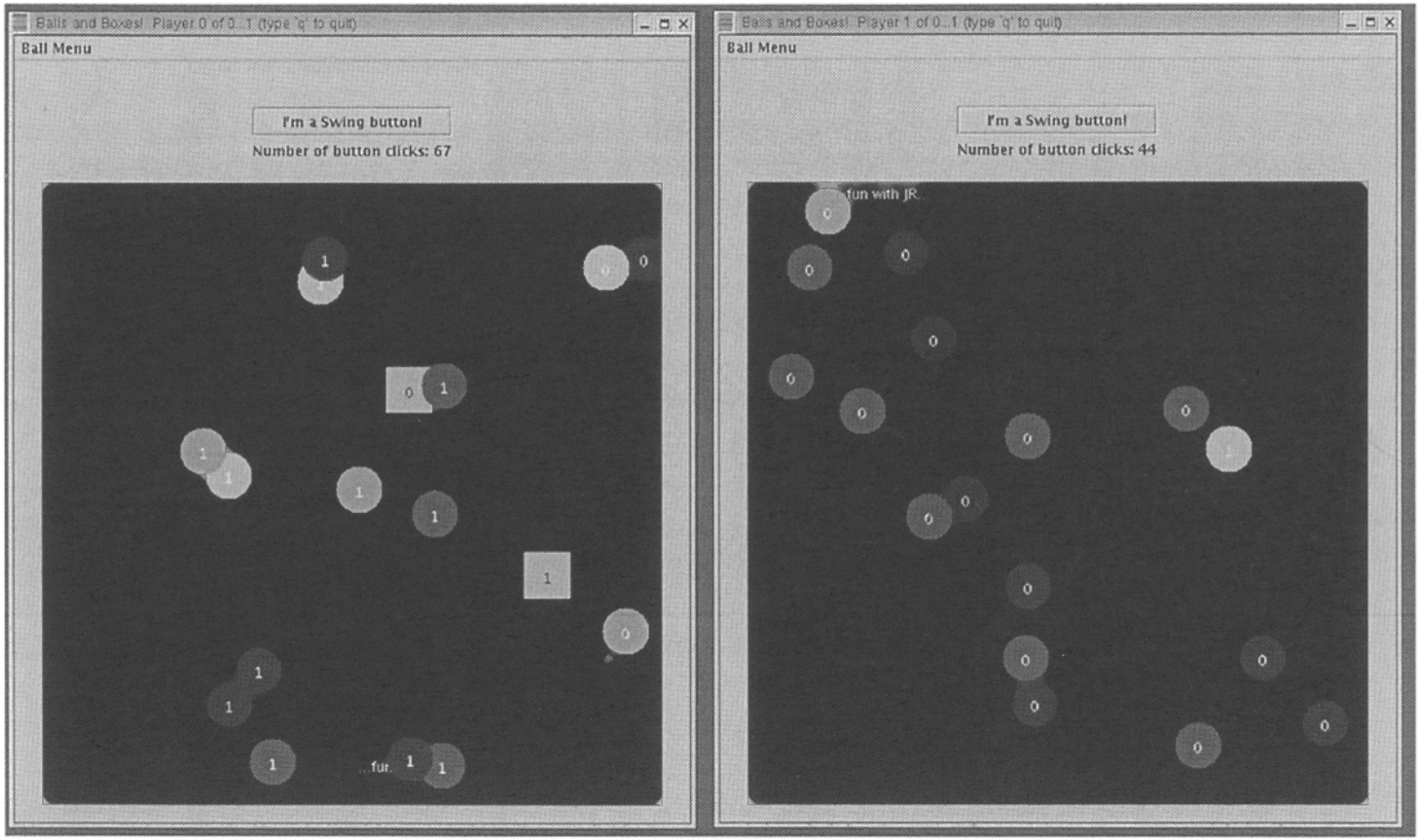


*Figure 20.1.* BnB game in action

- `Ball`
- `Box`
- `Mtext` (moving text)

`KeyInput`: keyboard input.

`MouseInput`: mouse input.

The subsections below present the code for these classes in the above order.

In a program that uses Swing, each graphical component provides a `paintComponent` method. This method is invoked whenever the frame repaints itself. Thus, in the BnB program, `Board` provides a `paintComponent` method; this method is responsible for drawing all the toys on the board. The program's other two graphical components — button and menu — have predefined `paintComponent` methods. The frame repaints itself whenever, for example, it detect that its contents have changed, e.g., the button has been clicked. It also repaints itself in response to explicit requests via the `repaint` method, such as those made within the code in `Board`. However, due to the event-driven nature of Swing, such requests do not necessarily happen immediately. (A repaint request can be viewed as initiating an asynchronous activity, much like JR's send statement.) Moreover, several repaint requests might be combined by Swing into a single one. Swing also provides the `paintImmediately` method, an alternative to `repaint` that can be used when painting without delay is desired.

### 20.2.1 `Main` Class

`Main` creates one `Window` for each player. It places each on its own virtual machine. The virtual machines are created on the physical machines specified by the command-line arguments or on the local host if none are specified (using code similar to that in Section 10.2). It then gives each board remote references for all the boards so that the boards can pass balls and boxes between them. This structure is similar in purpose to that used to establish connections ("links") between the servants in Dining Philosophers in the code in Section 11.3 and between the "points" in Matrix Multiplication in the code in Section 15.3, but the code here differs in two ways. First, the code here also uses an additional `goahead` operation to ensure that all boards have received their links before any `ballManager` is started. (See Exercise 20.1.) Second, the code here also sends each board references for all boards. Although this game uses only the references for the two adjacent boards, all are sent to make the code easier to change when adding new features to the game.

```
import java.awt.*;  import java.lang.*;
```

```
public class Main {
  private static final int winSize = 540, ToySize = 40;

  public static void main(String [] args) {
    String [] machs;
    // create players (Windows)
    if (args.length > 0) {
      machs = args;
    }
    else {
      // default: one machine, the local host.
      // do this work here to make code below smoother.
      machs = new String [1];
      machs[0] = "localhost";
    }
    remote Window [] rWindow = new remote Window [machs.length];
    for (int i = 0; i < machs.length; i++) {
       vm avm = new vm() on machs[i];
       rWindow[i] = new remote Window(winSize, ToySize,
                                      i, machs.length) on avm;
    }
    remote Board [] rBoard = new remote Board [machs.length];
    for (int i = 0; i < machs.length; i++) {
        rBoard[i] = rWindow[i].getrBoard();
    }
    for (int i = 0; i < machs.length; i++) {
        send rBoard[i].links(rBoard);
    }
    for (int i = 0; i < machs.length; i++) {
        send rBoard[i].goahead();
    }
  }
}
```

### 20.2.2 `Window` Class

`Window` creates the graphical components needed by the game. It creates a board, a button, and a menu. The code is somewhat detailed, but that is necessary to specify all the desired graphical features. The code first creates the board, the button, and the menu with its items. It then specifies how these components should be placed together. Specifically, the button and board are placed together in a `JPanel` named `mainPanel`. The window has listeners that are used to quit the game and to give focus to the board. The window code enables the button to start up new balls by passing to it a capability for the board's `startBall` operation. Similarly, it contains code for the menu items that invoke the board's `startBall` operation.

```
// menu building and handling based on
// http://java.sun.com/docs/books/tutorial/uiswing/components/
//        example-swing/MenuDemo.java
import java.awt.*;          import javax.swing.*;
import java.awt.event.*;    import java.lang.*;
import java.util.*;         import edu.ucdavis.jr.JR;

public class Window extends JFrame implements ActionListener {

  private final Board board;
  JMenuItem menuItem1, menuItem2;

  public Window(int winSize, int ToySize,
                int playerid, int players) {
    board = new Board(winSize, winSize, ToySize, playerid, players);

    Container contentPane = getContentPane();
    SwingApplication app = new SwingApplication();
    Component button = app.createComponents(board.startBall);
    contentPane.add(button);

    // build menu and its items
    JMenuBar menuBar = new JMenuBar();
    setJMenuBar(menuBar);
    JMenu menu = new JMenu("Ball Menu");
    menuBar.add(menu);
    menuItem1 = new JMenuItem("Green");
    menuItem1.addActionListener(this);
    menuItem2 = new JMenuItem("Orange");
    menuItem2.addActionListener(this);
    menu.add(menuItem1);
    menu.addSeparator();
    menu.add(menuItem2);
    pack();

    contentPane.add(board);

    JPanel mainPanel = new JPanel();
    mainPanel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
    setContentPane(mainPanel);
    mainPanel.add(button);
    mainPanel.add(board);
    // make window high enough for Board, button, and menu.
    // Swing doesn't seem to honor these requests, so ask for more
    final int sizeFudge = 60;
    setSize(winSize+sizeFudge,
            winSize+sizeFudge+button.getHeight()+menu.getHeight());
    setTitle("Balls and Boxes!  Player "+playerid+
             " of 0..."+(players-1)+" (type 'q' to quit)");
    getContentPane().setBackground(Color.pink);
```

```
    // set up window listeners
    final Board boardfinal = board;
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        JR.exit(0);
      }
      // when move into window, give focus to board.
      public void windowActivated(WindowEvent e) {
         boardfinal.requestFocus();
      }
    });
    setVisible(true);
  }

  public void actionPerformed(ActionEvent e) {
    JMenuItem source = (JMenuItem)(e.getSource());
    if (source == menuItem1) {
      board.startBall(Color.green, Color.white);
    }
    else if (source == menuItem2) {
      board.startBall(Color.orange, Color.white);
    }
  }

  public op remote Board getrBoard() {
    return board.remote;
  }
}
```

### 20.2.3 Button Class

The `SwingApplication` class creates the button. This code is modified slightly from that given in Reference [48]. The button counts the number of times it is clicked. On each click, it also creates two balls, one blue and one red. It uses a capability for the board's `startBall` operation to create each ball.

```
// based on
// http://java.sun.com/docs/books/tutorial/uiswing/
//        mini/secondexample.html
import javax.swing.*;  import java.awt.*;  import java.awt.event.*;

public class SwingApplication {
  private static String labelPrefix = "Number of button clicks: ";
  private int numClicks = 0;

  public Component createComponents(
                     cap void (Color, Color) startBall) {
    final JLabel label = new JLabel(labelPrefix + "0    ");
    JButton button = new JButton("I'm a Swing button!");
```

```
    // need to make it final since used within inner class
    final cap void (Color, Color) sf = startBall;
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        numClicks++;
        sf(Color.blue, Color.white);
        sf(Color.red, Color.white);
        label.setText(labelPrefix + numClicks);
      }
      });
    label.setLabelFor(button);

    // put space between a top-level container and its contents:
    // put the contents in a JPanel that has an "empty" border.
    JPanel pane = new JPanel();      // (top,left,bottom,right)
    pane.setBorder(BorderFactory.createEmptyBorder(30,30,10,30));
    pane.setLayout(new GridLayout(0, 1));
    pane.add(button);
    pane.add(label);
    return pane;
  }
}
```

### 20.2.4 `Board` Class

The `Board` class creates the board and controls the movements of the toys. It keeps a list of its toys in `myToys`. When a toy moves from one board to another, it is removed from the current board's `myToys` and is passed to its new board, where it is added to that board's `myToys`. The board code also contains a `startup` process to handle board start up (receiving remote references for all boards, as described in Section 20.2.1) and to create the balls and box that initially belong on the board. For each kind of toy, the board provides a manager to control the movement of the individual toy.

```
import java.awt.*;  import javax.swing.*;  import java.lang.*;
import java.util.*;

public class Board extends JPanel {

  private final int W, H, ToySize, playerid, players;
  private final java.util.List myToys;

  private final int next, prev; // next, previous boards indices

  public Board(int W, int H, int ToySize,
               int playerid, int players) {
    this.W = W;  this.H = H;  this.ToySize = ToySize;
    this.playerid = playerid; this.players  = players;
```

```
    next = (playerid+1)%players;
    prev = (playerid-1+players)%players;
    myToys = Collections.synchronizedList(new ArrayList());
    setBackground(Color.black);

    addKeyListener(new KeyInput(playerid,mymove));
    addMouseListener(new MouseInput(playerid,noop));
  }

  Random r = new Random();

  process MtextManager {
    int x = 100, y = 100;
    final Mtext myMtext = new Mtext("...fun with JR...",
                                    x, y, Color.yellow,
                                    ToySize, playerid);
    myToys.add(myMtext);
    while(true) {
      x = r.nextInt(W);
      y = r.nextInt(H);
      myMtext.set(x,y);
      repaint();
      try {
        Thread.sleep(1000+r.nextInt(1000));
      } catch (Exception e) {e.printStackTrace();}
    }
  }

  public op void links(remote Board [] rBoard);
  public op void goahead();
  remote Board [] rBoard;

  process startup {
    // need to receive rBoard before any ballManager starts
    receive links(rBoard);
    receive goahead();
    startBall(Color.blue, Color.white);
    startBall(Color.red, Color.white);
    startBox(Color.cyan, Color.black);
  }

  public op void startBall(Color myco, Color mycot) {
    // initial placement of ball.
    // W/2 to ensure stay around for a bit (L->R)
    final int Iters = 400; // number of iterations
    int x = r.nextInt(W/2);
    int y = r.nextInt(H);
    final int Xinc = 2;
    final int Yinc = 3;
    Ball nb = new Ball(x, y, myco, mycot, ToySize, playerid);
```

```
  restartBall(nb, x, y, Xinc, Yinc, Iters);
}

private op void restartBall(Ball b, int x, int y,
                            int xinc, int yinc, int iters) {
  myToys.add(b);
  send ballManager(b, x, y, xinc, yinc, iters);
}

// base amount for toys to sleep between moving
private static final int sleep = 100;

// move left->right (although code handles right->left too)
private op void ballManager( Ball b, int x, int y,
                             int xinc, int yinc, int iters) {
  repaint();
  while (iters-- > 0) {
    x += xinc;
    y += yinc;
    // compute new y before new x in case send off this board
    if (y < 0) { // top
      y = 0;
      yinc = -yinc;
    }
    else if (y >= H-ToySize) { // bottom
      y = H-ToySize;
      yinc = -yinc;
    }
    if (x > W) { // right
      x = 0;
      b.set(x,y);
      send rBoard[next].restartBall(b, x, y,
                                    xinc, yinc, iters);
      break;
    }
    else if (x < 0) { // left
      x = W;
      b.set(x,y);
      send rBoard[prev].restartBall(b, x, y,
                                    xinc, yinc, iters);
      break;
    }
    b.set(x,y);
    repaint();
    try {
      Thread.sleep(sleep/2+r.nextInt(sleep/2));
    } catch (Exception e) {e.printStackTrace();}
  }
  myToys.remove(b);
  repaint();
```

```
}

private op void startBox(Color myco, Color mycot) {
  final int x = 20;
  final int y = 20;
  final int Xinc = ToySize/2;
  final int Yinc = ToySize/2;
  Box b = new Box(x, y, myco, mycot, ToySize, playerid);
  restartBox(b, mymove, x, y, Xinc, Yinc);
}

private op void restartBox(Box b,
                           cap void (char) move,
                           int x, int y,
                           int xinc, int yinc) {

  myToys.add(b);
  send boxManager(b, move, x, y, xinc, yinc);
}

private op void mymove(char); // keyboard chars sent here

private op void boxManager( Box b,
                            cap void (char) move,
                            int x, int y,
                            int xinc, int yinc) {
  repaint();
  while (true) {
    char dir;
    receive move (dir);
    if (dir == 'j') { // left
      x -= xinc;
      if (x < 0) {
        x = W-ToySize;
        b.set(x,y);
        send rBoard[prev].restartBox(b, move, x, y, xinc, yinc);
        break;
      }
    }
    else if (dir == 'l') { // right
      x += xinc;
      if (x > W-ToySize) {
        x = 0;
        b.set(x,y);
        send rBoard[next].restartBox(b, move, x, y, xinc, yinc);
        break;
      }
    }
    else if (dir == 'i') { // up
      y = (y-yinc >= 0)?y-yinc:0;
```

```
      }
      else if (dir == 'k') { // down
        y = (y+yinc <= H-ToySize)?y+yinc:H-ToySize;
      }
      else {
        System.err.println("oops -- bad direction "+dir);
      }
      b.set(x,y);
      repaint();
      try {
        Thread.sleep(sleep/2+r.nextInt(sleep/2));
      } catch (Exception e) {e.printStackTrace();}
    }
    myToys.remove(b);
    repaint();
  }

  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    // synchronized access is necessary
    synchronized(myToys) {
      Iterator ti = myToys.iterator();
      while (ti.hasNext()) {
        Toy t = (Toy) ti.next();
        t.mydraw(g2);
      }
    }
  }
  public Dimension getMaximumSize(){
    return new Dimension(W,H);
  }
  public Dimension getMinimumSize(){
    return getMaximumSize();
  }
  public Dimension getPreferredSize(){
    return getMaximumSize();
  }
  // need this to make "tab" switch between button and board.
  public boolean isFocusTraversable(){
    return true;
  }
}
```

The `MtextManager` is the simplest because only one `Mtext` object exists on each board and the object never moves to another board. The manager first creates an `Mtext` object. It then enters an infinite loop in which it randomly picks the new position for the text, informs the object of its new position,

requests that the board be repainted, and then sleeps for a random amount of time.

The balls require more complicated code to create and control. The `Board` class provides a start and a restart operation for balls. The start operation is used to create a new ball. The restart operation is used by the start operation and also when a ball moves from one board to another. The restart code starts up a `ballManager` process to manage the ball. The `ballManager` process loops for at most `iters` iterations. On each iteration of the loop, it computes the new location of the ball. If the new location is at the top or the bottom of the board, it in effect makes the ball "bounce off" the border. If the new location of the ball is off the board to the left or the right, the `ballManager` process sends a message to the adjacent board's `restartBall` operation, exits the loop, removes the ball from this board's `myToys`, and terminates. The restart operation is given the ball's position and how many iterations remain for the ball. Otherwise, it informs the ball object of its new position, requests that the board be repainted, and then sleeps for a random amount of time.

Boxes are created and controlled similarly to how balls are. As for balls, the `Board` class provides a start and a restart operation for boxes and a `boxManager` process to manage each box. However, `boxManager` takes character input to determine how to move the box. Note how `boxManager` receives this input from the `move` capability. When a box moves off the current board, its manager on the current board terminates and the box is passed to an adjacent board, where a new box manager for it executes. The box still receives input when the focus is on its home board, though, because a capability for the `mymove` operation on the home board is passed with it. (See Exercise 20.3.)

Note that the `myToys` collection is declared as a `synchronizedList` and that the `paintComponent` method uses a Java `synchronized` statement to provide mutually exclusive access to the `myToys` collection. It ensures that other processes do not change `myToys` while it is being redrawn, i.e., while the iterator is going through the collection. Generally, we advise against intermixing Java's and JR's synchronization, but the above is simple and safe because those threads blocked by a `synchronized` statement will not block indefinitely and interfere with JR's quiescence detection.

### 20.2.5 Toy Classes

The `Toy` class is an abstract base class for the toys. It simply provides fields and methods that the toys have in common. It is serializable so that toys can be passed between boards, which might be located on different virtual machines (see Section 10.7).

```
import javax.swing.*;  import java.awt.*;  import java.lang.*;

abstract public class Toy implements java.io.Serializable {
```

```
  protected int x, y;
  protected final Color myco, mycot;
  protected final int size, playerid;

  public Toy(int x, int y, Color myco, Color mycot,
             int size, int playerid) {
    this.x = x;  this.y = y;
    this.myco = myco;  this.mycot = mycot;
    this.size = size;  this.playerid = playerid;
  }

  public void set(int sx, int sy) {
    x = sx;  y = sy;
  }

  abstract public void mydraw (Graphics2D g2);
}
```

The `Ball` and `Box` classes extend the `Toy` class. Each has a `mydraw` method that is used to draw the ball or the box at the toy's current location.

```
import javax.swing.*;  import java.awt.*;  import java.lang.*;

public class Ball extends Toy {

  public Ball(int x, int y, Color myco, Color mycot,
              int size, int playerid) {
    super(x, y, myco, mycot, size, playerid);
  }

  private boolean firstmydraw = true;
  private int deltax, deltay;
  private String mystring;
  public void mydraw (Graphics2D g2) {
    // since size and font size don't change,
    // compute only on first time.
    // takes into account size of text to center text.
    if (firstmydraw) {
      firstmydraw = false;
      mystring = playerid+"";
      FontMetrics metrics = g2.getFontMetrics();
      int width = metrics.stringWidth( mystring );
      int height = metrics.getHeight();
      deltax = (size-width)/2;
      deltay = (size+height)/2;
    }
    g2.setColor( myco );
    g2.fillOval(x,y,size,size);
    g2.setColor( mycot );
    g2.drawString(mystring,x+deltax,y+deltay);
```

```
  }
}

import javax.swing.*;  import java.awt.*;  import java.lang.*;

public class Box extends Toy {

  public Box(int x, int y, Color myco, Color mycot,
             int size, int playerid) {
    super(x, y, myco, mycot, size, playerid);
  }

  private boolean firstmydraw = true;
  private int deltax, deltay;
  private String mystring;
  public void mydraw (Graphics2D g2) {
    // since size and font size don't change,
    // compute only on first time.
    // takes into account size of text to center text.
    if (firstmydraw) {
      firstmydraw = false;
      mystring = playerid+"";
      FontMetrics metrics = g2.getFontMetrics();
      int width = metrics.stringWidth( mystring );
      int height = metrics.getHeight();
      deltax = (size-width)/2;
      deltay = (size+height)/2;
    }
    g2.setColor( myco );
    g2.fillRect(x,y,size,size);
    g2.setColor( mycot );
    g2.drawString(mystring,x+deltax,y+deltay);
  }
}
```

### 20.2.6 Input Classes

The `KeyInput` and `MouseInput` classes are used to provide input to the board. The `KeyInput` class takes a capability (`charKey`) for an operation in its board to which characters are to be sent. It sends only meaningful characters and ignores others. It also handles exiting the game.

```
import javax.swing.*;  import java.awt.event.*;
import edu.ucdavis.jr.JR;

public class KeyInput extends KeyAdapter {

  private final int me;
  private final cap void (char) charKey;
```

```
  public KeyInput(int me, cap void (char) charKey) {
    this.me = me;
    this.charKey = charKey;
  }

  public void keyPressed(KeyEvent e) {
    char c = e.getKeyChar();
    switch(c) {
      case 'q':    case 'Q':
      case '\177': case '\003':
        {JR.exit(0);}
      case 'i': // up
      case 'k': // down
      case 'j': // left
      case 'l': // right
        { send charKey(c); break; }
      default: break; // just ignore
    }
  }
}
```

The `MouseInput` class has a similar structure. However, for this game, mouse clicks are not used by the board. (`MouseInput` does print out the cursor's position, though.) So, the board passes a `noop` capability as the `mouseClick` parameter to its instance of `MouseInput`.

```
import javax.swing.*;  import java.awt.event.*;

public class MouseInput extends MouseAdapter {

  private final int me;
  private final cap void (int, int) mouseClick;

  public MouseInput(int me, cap void(int, int) mouseClick) {
    this.me = me;
    this.mouseClick = mouseClick;
  }

  public void mouseClicked(MouseEvent e) {
    int x = e.getX(), y = e.getY();
    System.out.println(me + " got click "+ x + " " + y);
    send mouseClick( x, y );
  }
}
```

## 20.3 Miscellany

This section contains miscellaneous comments about the BnB program, using Swing with JR programs, and GUI packages.

The BnB code has many constants in it, for example, to specify the sizes of windows and toys. These constants could easily be specified as command-line arguments (but they were not to keep the code a bit simpler to present).

Since BnB's `Board` consists of a `mainPanel` that contains another panel (`board`) and a `button`, the keyboard focus will be on one or the other of these components. (Even though the button in BnB responds to only mouse clicks, buttons in general can be triggered by keys too.) The focus can be changed between components, as in many GUI applications, by hitting the tab key. The code also sets the focus to the board area whenever the window is activated, i.e., the mouse is moved into the window. Whether a particular program needs to set focus, as here in BnB, depends on the exact structure of the components it uses.

As discussed in Section 4.5, JR's normal behavior is to terminate a program once the program becomes quiescent. If a program is waiting to read input from a terminal, the program will not be terminated. However, the current JR implementation is not aware of Swing (or AWT) events, which occur asynchronously with respect to the JR program. Suppose, for example, that a JR program using Swing is blocked waiting to receive a character from the keyboard (as in the `boxManager`) or a mouse click, and that the program has no other processes in it. Then, the JR implementation will consider the program to be quiescent and will terminate it. (The BnB program does not terminate because, among other processes, its `MtextManager` executes an infinite loop with no blocking except for sleeping, of which the JR implementation is aware.) The simplest way to prevent such termination is to use a command-line option when running the program (see Appendix C for details). Another way is to use an additional "sleeper" process that simply sleeps for longer than the program should execute or sleeps for any period of time within an infinite loop.

Swing contains many other useful features. Here is a partial list of some features that might be useful in writing JR programs with Swing.

- many kinds of buttons, menus, etc.
- menu selection via keyboard, possibly using "accelerators".
- other kinds of frames. For example, the BnB program could employ a text frame to allow users on different systems to chat with one another. Or it could display a small status window within each window to show what all boards look like.
- images. For example, the BnB program could, instead of drawing a rectangle for each box, display a picture of a cat.
- animation timers.

The BnB program uses Swing. It could also be written using AWT. Swing is a very flexible, but complicated package due to its many options. AWT is lower-level, but probably simpler to learn, than Swing. However, Swing seems to be the recommended choice. One key reason is that Swing programs will present the same graphical displays on any system, whereas AWT programs are not guaranteed to do so because AWT depends on the host's native windowing system. Such portability is very desirable in many applications.

## Exercises

20.1 Suppose the BnB program did not use the `goahead` operation. Give a step-by-step execution sequence that could yield an error.

20.2 Rewrite the startup code so that it uses capabilities for operations rather than remote references.

20.3 Suppose the receive statement in `boxManager` receives from the `mymove` operation instead of from the `move` capability. First, speculate as to the differences in program behavior. Then, run the modified program to see what actually happens.

20.4 Modify the BnB program so that:

(a) balls can also move from right to left. The direction of a particular ball is chosen randomly when it is created.

(b) modify the previous part so that balls created from the menu move in the direction (left-to-right or right-to-left) specified in the menu, which will need to be expanded to include direction choices.

(c) boxes are allowed to wrap around from the top to the bottom of the board and vice versa.

(d) boxes keep moving in the last direction specified if no key is hit.

(e) the board includes barriers, which are displayed on the screen. Balls can pass through the barriers, but boxes cannot.

(f) the game includes a new, triangle toy. In effect, there is one triangle toy in the whole game; it is displayed in the same position on each board. (Or one could view the game as having one triangle toy per board; the triangles are displayed in the same relative position on each board.) It moves randomly and periodically (say once per second).

(g) hitting the 'c' key will cause the box to move directly to the board on which it was initially created.

(h) hitting the 'p' key will cause the game to pause (in all windows) until another key is hit (in the same window as original 'p').

(i) clicking the mouse within the board will move the box to the mouse's location. (If the box is on a different board, the box should move to the specified position located on the other board.)

(j) clicking the mouse within the board will set the focus (for the keyboard) to the board.

20.5 Modify the BnB program (or one of its variants developed in the previous exercise) so as to make the game more interesting. Keep scores for each player and display the current scores on each player's display. Award points when, for example,

- a box collides with a ball on the screen.
- a box collides with another box but only if the first box is not on its home board.
- a box's bullet hits a ball. This change will require that boxes shoot bullets, which will necessitate some other changes. Examples: the shape of the box might change to indicate in which direction the box's shooter is pointing; the box should be able to rotate (e.g., say via key 'a' to rotate counterclockwise and key 's' to rotate clockwise) to line up its shots.
- other such rules of your own creation.

20.6 Develop a GUI that visualizes one of the programs developed in earlier chapters, such as

- one of the versions of Dining Philosophers (Chapter 11)
- grid computation (Chapter 16)
- the DFS program (Chapter 18)

20.7 Develop a distributed JR program that includes a GUI to perform some animation or visualization. Possible examples include various arcade games, card games, and other games:

| | | | |
|---|---|---|---|
| Poker | Hearts | Tron | Combat |
| Pong | PacMan | Frogger | Alien Invaders |
| Battleship | Othello | Simon | Space Invaders |
| Risk | Monopoly | Maze games | Tetris |

and other, more serious applications:

Shortest Path Finding
Distributed Make
Cache Simulation
Multiprocessor Visualization
Distributed Graphics Renderer
Wave Simulation
Simplex Method Tableau Implementation
Highway (with Traffic Light) Visualization
Elevator Visualization
Railroad or Subway Visualization

Aim first for a simple prototype that demonstrates the essential, concurrent aspects of your chosen application. Then, refine it and add more features.

Chapter 21

# PREPROCESSORS FOR OTHER CONCURRENCY NOTATIONS

In earlier chapters, we have seen several synchronization mechanisms and how they are represented in JR. In particular, we have seen semaphores (Chapter 6), asynchronous message passing (Chapter 7), remote procedure call (Chapter 8), and rendezvous (Chapter 9). Besides these mechanisms, several other notations for concurrency are noteworthy for historical, conceptual, and practical reasons. This chapter will explore Conditional Critical Regions (CCRs), monitors, and Communicating Sequential Processes (CSP).

JR itself does not provide CCR, monitor, or CSP notation. However, the JR implementation comes with preprocessors for each of these notations. Such a preprocessor converts a program written in a particular notation into an equivalent JR program, which can then be translated and executed in the normal fashion. The concurrency notations use additional constructs; the keywords used in these constructs begin with an underscore (e.g., `_region`).

The rest of this chapter briefly describes the three notations and how synchronization is expressed in each. It also gives sample programs in each of the notations. For more information about the preprocessors and specifics of the notations (how to use them, summary of syntax, etc.), see the documentation and sample programs that come with the JR implementation. For a thorough general discussion of the three concurrency notations, see Reference [7]. (The bounded buffer examples in this chapter are based on those in Reference [7] and the overall presentation in this chapter is influenced by that work too.)

## 21.1 Conditional Critical Regions (CCRs)

Conditional Critical Regions (CCRs) [24] were developed to provide a synchronization mechanism that is higher-level than semaphores. In programs written with semaphores, it is not always apparent which variables are being shared and whether `P` and `V` semaphore operations are being used where and as

they should be. For example, it is simple to accidentally omit a `P` or a `V`, use a `V` where a `P` is needed, or `P` or `V` the wrong semaphore.

CCRs provide an abstraction mechanism, called a *resource,* to specify collections of shared variables. They also provide a new statement, the *region* statement, to access such collections. Variables within a resource can be accessed only within region statements that name that resource. Mutual exclusion to variables in a given resource is implicit: only one process at a time is allowed to be executing a region statement for a given resource.

As an example, consider the following problem involving three processes. Each of the first two processes generates a number and adds its number to a running sum. The third process waits until both of the other two processes have added in their numbers before retrieving the sum.

Here is the CCR solution to this problem. First, we define a `Data` resource, which contains two variables.

```
_resource Data {
  public int count = 0;
  public int sum = 0;
}
```

Note that the two variables are defined to be public, even though they can only be accessed within region statements. Then, we define the code that instantiates a `Data` resource and uses it.

```
public class Sum {
  // the CCR
  private static Data data = new Data();
  public static void main(String [] args) {
  }

  private static process p( (int i = 1; i <= 2; i++) ) {
    // generate a number, x
    int x;
    ...
    _region Data _with thisd = data {
      thisd.count++;
      thisd.sum += x;
    }
  }
  private static process q {
    // get the sum of numbers generated by p1 and p2
    int x;
    _region Data _with thisd = data _when thisd.count == 2 {
      x = thisd.sum;
    }
  }
}
```

Notice that each _region specifies the name of a resource and a _with part, and may specify a _when part. The _with part names a (new) variable to be used within the region statement to reference variables within the named resource. The optional _when part specifies under what condition (i.e., when) the region statement is allowed to execute. Evaluation of the _when part and, if that evaluates to true, subsequent execution of the region statement's body is atomic.

Thus, in the above program, the two p processes simply gain exclusive access to the variables in the resource (a missing _when part is considered to be true) and update the variables. The q process waits until count is 2, i.e., until both p processes have incremented sum.

For the above CCR program, the BB resource and the BBMain class are placed in separate files. Because each contains CCR constructs, each must be translated by the CCR preprocessor to generate JR code.

As a more interesting example, consider the CCR solution to the bounded buffer problem (described in Section 9.3). The BB resource declares the variables associated with the bounded buffer.

```
_resource BB {
  public static final int N = 10;
  public int count = 0, front = 0, rear = 0;
  public int [] buf = new int [N];
}
```

The BBMain class instantiates a bounded buffer and families of depositor and fetcher processes to use it. The _region in the depositor ensures that there is room in the buffer to deposit a new item; the _region in the fetcher ensures that there is an item in the buffer to fetch.

```
public class BBMain {
  private static int N = 10;
  // the CCR
  private static BB mybb = new BB();
  public static void main(String [] args) {
  }
  private static process depositer( (int i = 1; i <= N; i++) ) {
    _region BB _with thisbb = mybb _when thisbb.count < BB.N {
      thisbb.buf[thisbb.rear] = i*100;
      thisbb.rear = (thisbb.rear+1) % BB.N;
      thisbb.count++;
    }
  }
  private static process fetcher( (int i = 1; i <= N; i++) ) {
    _region BB _with thisbb = mybb _when thisbb.count>0 {
      System.out.println("fetch " + thisbb.buf[thisbb.front]);
      thisbb.front = (thisbb.front+1) % BB.N;
```

```
      thisbb.count--;
    }
  }
}
```

A CCR program will terminate itself automatically if all of its processes have terminated or are blocked waiting on their guards in `_region` statements.

## 21.2 Monitors

Monitors [25] have appeared in several languages and have influenced others, including Java (see the end of this section). Like CCRs, monitors also provide a higher-level abstraction than do semaphores. With CCRs, the declarations of shared variables (i.e., resources) and the code that access shared variables (i.e., region statement) are scattered throughout the program. With monitors, these appear together in a single construct; this construct is called a monitor and resembles a class, but with some difference, which will be described below. Processes outside the monitor access monitor variables indirectly by calling monitor procedures (methods).

Like CCRs, mutual exclusion for monitors is implicit: only one process at a time may be executing within a given monitor. Unlike CCRs, whose region statement allows a boolean expression for synchronization, code within a monitor uses explicit signaling, more like what is used with semaphores. This signaling is accomplished via *condition variables.* A condition variable defines a queue of processes, each of which is waiting to be signaled by some other process.

Two statements manipulate condition variables: `_wait` and `_signal`. The wait statement places the currently executing process at the rear of the queue for the specified condition variable; it also releases the exclusive access this process had for the monitor, thus allowing another process to gain access. The signal statement awakens the first process on the queue for the specified condition variable; the signaling process continues its execution in the monitor. The signaled process will compete for access to the monitor with other processes trying to enter the monitor. (Other possible definitions of signal's behavior are described later in this section.)

The wait and signal statements have some similarity to `P` and `V` semaphore operations. However, a signal statement has no effect if the condition variable's queue is empty; unlike a `V` (which would increment the semaphore's value in such a case), a signal is not remembered. Also, a wait statement always delays the currently executing process; recall that a `P` operation only delays the process if the semaphore's value is zero.

As a simple example, here's a monitor solution to the `Sum` problem from the previous section. The code that uses the monitor is straightforward. It

instantiates a Data monitor and the processes simply invoke the appropriate monitor procedures.

```
public class Sum {
  // the monitor
  private static Data data = new Data("my data monitor");
  public static void main(String [] args) {
  }

  private static process p( (int i = 1; i <= 2; i++) ) {
    // generate a number, x
    int x;
    ...
    data.addToSum(x);
  }
  private static process q {
    // get the sum of numbers generated by p1 and p2
    int x;
    x = data.getSum();
    System.out.println(x);
  }
}
```

The monitor Data contains the shared variables and the code that accesses them, and code that enforces the desired synchronization. It provides two procedures: addToSum and getSum. The first is used by the two p processes and the second is used by the q process.

```
_monitor Data {
  _var int count = 0;
  _var int sum = 0;
  _condvar both;
  _proc void addToSum(int x) {
    sum += x;
    count++;
    _signal(both);
  }
  _proc int getSum() {
    while (count < 2 ) { _wait(both); }
    _return sum;
  }
}
```

Data also declares a condition variable, both. Each execution of addToSum signals both. The code in getSum uses a loop to delay the invoking process if necessary. For example, if only one p process has added its number to sum, then the invoking q process needs to wait until the other p process has added its number to sum (and incremented count to 2) and signaled. This monitor can be written without using a loop (Exercise 21.3). However, wait statements

will generally be contained in loops to avoid bad effects that can occur due to "signal stealers", which will be described as part of the next example.

For the above monitor program, the `Data` monitor and the Sum class are placed in separate files. The Data monitor contains monitor constructs, so it must be translated by the monitor preprocessor to generate JR code. The `Sum` class contains no monitor constructs so it is translated as a regular JR program. (The invocations in `Sum` of the monitor's methods are regular JR invocations.)

The next example gives a monitor solution to the bounded buffer problem. Here again, the code that uses the monitor is straightforward. It instantiates a `BB` monitor and its processes invoke the monitor procedures to deposit or fetch an item.

```
public class BBMain {
  // number of fetchers and depositors (each) processes
  private static final int N = 6;
  // the monitor
  private static BB bb = new BB("bb");
  public static void main(String [] args) {
  }
  private static process prod( (int i = 1; i <= N; i++) ) {
    bb.deposit(i);
  }
  private static process cons( (int i = 1; i <= N; i++) ) {
    int got = bb.fetch();
    System.out.println(got);
  }
}
```

The monitor `BB` declares the shared variables and two condition variables. A process attempting to deposit an item will proceed only if there is room in the buffer; it will delay on the `not_full` condition variable if the buffer is full, until a consumer process later signals. When a process has completed a deposit, it signals `not_empty` in case a consumer process is waiting. The actions of a consumer process attempting to fetch an item are symmetric.

```
_monitor BB {
  private static final int N = 4; // buffer size
  _var int [] buf = new int [N];
  _var int front = 0;  _var int rear = 0;
  _var int count = 0;
  _condvar not_full;  _condvar not_empty;
  _proc void deposit(int data) {
    while (count == N) {
      _wait(not_full);
    }
    buf[rear] = data;
    rear = (rear+1) % N;
```

```
    count++;
    _signal(not_empty);
  }
  _proc int fetch() {
    while (count == 0) {
      _wait(not_empty);
    }
    int result = buf[front];
    front = (front+1) % N;
    count--;
    _signal(not_full);
    _return result;
  }
}
```

Here again the waiting is done within a loop. On the contrary, suppose that the code used an if statement instead of a while statement. Then, the following sequence of activities could occur.

- Consumer 1 attempts to fetch. It finds count to be 0, so it waits on `not_empty`.
- Producer 1 begins depositing an item.
- Consumer 2 attempts to enter the monitor. It delays because Producer 1 has access.
- Producer 1 finishes depositing an item. Producer 1 signals `not_empty`, which awakens Consumer 1.
- Producer 1 leaves the monitor.
- At this point, Consumer 1 and Consumer 2 are attempting to enter the monitor. However, because Consumer 2 arrived first, it is granted access.
- Consumer 2 fetches the item, sets `count` to 0, and leaves the monitor.
- Consumer 1 now continues in the monitor (after its wait statement, which we supposed is not within a loop). It fetches an undefined item, sets count to -1 (oops!), and leaves the monitor.

Thus, the monitor's data has been corrupted. The reason is that, in effect, Consumer 2 stole the signal intended for Consumer 1. Enclosing the wait statement in a loop, as in the original code, prevents the bad *effect* signal stealers can cause (but it does not prevent signal stealing).

The particular definition of signal given above is known as signal and continue. The four standard definitions are summarized in the following table. They differ in the effects of signal on the signaling and signaled processes.

| | | |
|---|---|---|
| SC | signal and continue | signaler continues executing in monitor.<br>signaled competes for re-access to monitor. |
| SW | signal and wait | signaler steps out of monitor; it competes for re-access to the monitor.<br>signaled executes in monitor next. |
| SU | signal and urgent wait | like SW, but signaling processes are given preference in re-accessing the monitor. |
| SX | signal and exit | signaler leaves monitor as though it executed a return (i.e., signal also terminates the invocation).<br>signaled executes in the monitor next. |

Thus, in the last three, the signaled process *preempts* the signaling process in executing in the monitor; due to preemption, placing wait statements within a loop is generally needed only for SC. The preprocessor for monitor notation supports all these signaling disciplines. The choice is made by a command-line option; the default is SC.

The monitor notation also provides a few additional features. It allows arrays of condition variables and testing whether a condition variable has any processes waiting on it (`_empty`). It also provides prioritized waiting on a condition variable (a variant of `_wait`) and a function to determine the priority of the first process on a condition variable (`_minrank`). Finally, the notation provides a way to awaken all processes waiting on a condition variable (`_signal_all;` only allowed for SC). These features are described in more detail in the documentation for the monitor preprocessor.

A monitor program will terminate itself automatically if all of its processes have terminated or are blocked due to `wait` statements.

Note that Java itself provides a form of monitor, but it does not provide the various signaling disciplines and its signal (`notify()`) chooses the process to awaken non-deterministically. The signals defined in this section, as in most traditional definitions of monitors, awaken processes in first-come, first-served order.

## 21.3 Communicating Sequential Processes (CSP)

CSP [26] is a separate language notation, but its key synchronization aspects have appeared in or influenced several languages, including Ada [1], occam [14, 46], SR [9], and JR. CSP provides a form of rendezvous as its way for processes to communicate. The exact form of rendezvous differs from what we have seen earlier (Chapter 9). CSP uses input/output commands for rendezvous. An input command is analogous to `receive` or `inni` in JR; an output command is analogous to a call invocation in JR. However, as we shall see below, there are significant differences. Input/output commands specify the name of the process with which to communicate and the kind of message they wish to receive or send. Input/output commands can appear as independent statements or as part of if and do statements.

As our first CSP program, we consider the Sum program seen in the previous sections.

```
_program Sum {
  _specs {
    p [2] {}
    q { add(int); }
  }
  _process p (i) {
    // generate a number, x
    int x;
    ...
    _o q!add(x);
  }
  _process q {
    int sum = 0;
    int x;
    _i p[0]?add(x);
    sum += x;
    _i p[1]?add(x);
    sum += x;
    System.out.println(sum);
  }
}
```

A CSP program requires the specification of the processes followed by the code for the processes. The `_spec` gives their names and the kinds of messages they can receive via input commands. In this program, `p` is declared being a one-dimensional family of processes with two members; the code for `p` uses `i` as the process's "id".

Process `p` uses an output statement to send its value of `x` to process `q`. Process `q` uses an input statement to receive its value of `x` from each process `q`. Both input and output statements are blocking. That is, processes delay until an input command in one process *corresponds* with an output command in another process. Correspondence means that an input command in one process names the other process, an output command in the second process names the first process, and the messages specified in the two commands match. When the commands correspond, the values are copied from the output command to the variables in the input command and the two processes continue their executions.

Note that in the above program, `q` rendezvouses with `p[0]` before it rendezvouses with `p[1]`. Suppose that `p[0]` must perform a lot of work to compute its value, whereas `p[1]` does not. Then, `q` and `p[1]` are waiting unnecessarily. To improve the program's performance, `q` can be modified to rendezvous with whichever `p` is ready first. The new code for process `q` is as follows:

```
  _process q {
    int sum = 0;
```

```
    int x;
    _i [i = 0 _to 1] p[i]?add(x);
    sum += x;
    _i [i = 0 _to 1] p[i]?add(x);
    sum += x;
    System.out.println(sum);
  }
```

Each of `q`'s input statements now uses a quantifier to indicate that it is willing to communicate with either of the two `p` processes. Note that this kind of quantified input statement in CSP inspired a similar quantified input statement in JR (see Section 9.8).

A CSP program, such as the above program, consists of just one `_program` construct. Thus, the entire program is placed in one file, which is then translated by the CSP preprocessor to generate JR code.

The next example is the bounded buffer problem, as seen in previous sections. It shows how input/output commands can also appear as guards of if and do statements. These statements are represented as `_if` and `_do`. They consist of multiple "arms", each of which contains a *guard* followed by a block of code. Each guard is an input command, an output command, or a plain boolean expression.

`_if` and `_do` execute differently from their counterparts in Java. A key difference is that all the guards for all arms are evaluated, conceptually, in parallel. Then, one of the guards is chosen nondeterministically and the associated code is executed. (Contrast that with evaluation of Java's if statement, which sequentially evaluates the expression in the "if" part, then the expression in the "else if" part, etc., stopping when it finds one true and executing the associated code.) As a simple example, consider the following CSP if statement to set max to the maximum of `x` and `y`.

```
_if
  _g (x >= y) { max = x; }
  _g (y >= x) { max = y; }
_fi
```

It contains two arms; the guard of each consists of just a boolean expression. Note that if `x` and `y` have the same value, both guards are true and either assignment statement may be chosen to execute.

A CSP do statement executes similarly to a CSP if statement. However, if any guard is true, then after the associated code is executed, the statement is executed again, by reevaluating the guards, etc. The do statement terminates when all of its guards evaluate to false.

As noted above, input/output commands can appear in guards in CSP if and do statements. In this role, an input/output command may be preceded by one or more quantifiers, as seen earlier, and by a boolean expression. If

the boolean expression evaluates to false, then the entire guard is considered to be false. Otherwise, the guard is considered to be true if the input/output command corresponds with one in another process. A guard's value is "not yet determined" if its boolean expression is true, but its input/output command does not (yet) correspond with one in another process. If the evaluation of a process's if or do statement yields no true guards but one or more not yet determined guards, then the process is delayed until the value of those guards can be determined. Such a guard becomes true when another process's input/output command corresponds with it.[1] Such a guard becomes false when the process named in the input/output command terminates.

Here is a CSP solution to the bounded buffer problem.

```
_program BB {
   public static final int S = 10; // number of processes
  _specs {
    Prod [S]  {}
    Cons [S]  { fetch(int); }
    B         { deposit(int); more(); }
  }

  _process Prod(i) {
    ...
    _o B!deposit(i*100);
    ...
  }
  _process Cons(i) {
    ...
    _o B!more();
    int x;
    _i B?fetch(x);
    ...
  }
  _process B { // buffer manager
   final int N = 6; // buffer size
   int count = 0, front = 0, rear = 0;
   int [] buf = new int [N];
   _do
     _i [i = 0 _to S-1] (count < N) Prod[i]?deposit(buf[rear]) {
       rear = (rear+1) % N;
       count++;
     }
     _i [i = 0 _to S-1] (count > 0) Cons[i]?more() {
       _o Cons[i]!fetch(buf[front]);
```

[1] To be more precise, for the guard to be true, the two processes must also commit to communicating with one another via this pair of commands, and not with other processes with whom they might also have input/output commands that correspond (or other commands with the same processes).

```
      front = (front+1) % N;
      count--;
    }
  _od
 }
}
```

It has a family of producer processes and a family of consumer processes. It uses a buffer manager process, which contains the actual buffer. As shown, a producer process uses an output statement to deposit an item; a consumer process uses an output statement followed by an input statement to fetch an item. The buffer manager uses a do statement with two arms. One arm is used to communicate with producers, but only when the buffer is not full. Similarly, the other arm is used to communicate with consumers, but only when the buffer is not empty. Both guards become false once all producers and consumers terminate, which causes the buffer manager's do statement to terminate too.

The above solution is asymmetric with respect to how the producers and consumers interact with the buffer manager. A more pleasing, symmetric solution uses an output command in a guard, so the code in the consumer is now just an input statement.

```
_program BB {
   public static final int S = 10; // number of processes
 _specs {
   Prod [S]  {}
   Cons [S]  { fetch(int); }
   B         { deposit(int); }
 }

 _process Prod(i) {
   ...
   _o B!deposit(i*100);
   ...
 _process Cons(i) {
   ...
   int x;
   _i B?fetch(x);
   ...
 }
 _process B { // buffer manager
  final int N = 6; // buffer size
  int count = 0, front = 0, rear = 0;
  int [] buf = new int [N];
  _do
    _i [i = 0 _to S-1] (count < N) Prod[i]?deposit(buf[rear]) {
      rear = (rear+1) % N;
      count++;
    }
```

```
    _o [i = 0 _to S-1] (count > 0) Cons[i]!fetch(buf[front]) {
      front = (front+1) % N;
      count--;
    }
  _od
 }
}
```

Some definitions of CSP or CSP-like rendezvous mechanisms (e.g., JR's `inni` statement, Occam's `alt` statement, and Ada's `select/accept` statements) do not allow output commands or their equivalents as guards.

The description above indicates that when a process terminates, any guards in which it is named become false. This semantics is known as *implicit termination.* An alternative semantics is known as *explicit termination.* In this semantics, such a guard's value would remain undetermined. Implicit termination is often simpler for the programmer, since explicit termination requires the programmer to write additional code. (See Exercise 21.7.) The CSP preprocessor supports both kinds of termination. The choice is made by a command-line option; the default is implicit termination.

## Exercises

21.1 The `Sum` CCR code is not reusable. That is, suppose we want each of the two `p` processes to produce two numbers and the `q` process to get the sum of the first pair of numbers and then the second pair. Modify the code to do so. Be sure your code pairs up one number from each `p` and prevents two numbers from one `p` from being considered a pair.

21.2 Repeat Exercise 21.1 but for the `Sum` monitor.

21.3 Rewrite monitor `Data` so it does not use a loop (but so it still provides the desired synchronization).

21.4 Explain why the original `Sum` CSP program is already reusable (in the sense of Exercises 21.1 and 21.2). Is the `Sum` CSP program that uses quantifiers reusable? Explain your answer.

21.5 Rewrite the `Sum` program using semaphores (Chapter 6).

21.6 Rewrite the monitor programs in this chapter using only Java's monitor-like mechanisms.

21.7 Rewrite the CSP bounded buffer program so that it uses explicit termination. When each producer and consumer process finishes, have it inform the buffer manager via a "done" message. Then, the buffer manager can terminate.

21.8 Program each of the following using the CCR, monitor, and CSP preprocessors.

(a) the `CSOrdered` program (Section 6.1).

(b) a barrier (Section 6.3).

(c) the Readers/Writers Problem (Section 9.3).

(d) the Atomic Broadcast Problem (Exercises 7.10 and 9.15).
For the monitor solution, use the SC discipline.

- Solve Exercise 7.10(a) in two ways: first without `signal_all` and then with `signal_all`. Do not use `signal_all` in the other parts.
- You may use arrays of condition variables only in your solutions to Exercise 7.10(b) and Exercise 7.10(c) but if you do, explain why in each case you need them.
- Use only the monitor procedures `void deposit(int msg)` and `int fetch()`, with no additional parameters (unless the parameters are used only to make the program's output more clear), except `fetch` can be passed a process's id in your solution to Exercise 7.10(b) and Exercise 7.10(c).

(e) the Savings Account Problem (Exercises 7.11 and 9.16).
For the monitor solution, use the SC discipline.

- First, develop a solution that is correct, but services any waiting withdrawals it can when a deposit is made. You may use `signal_all` in your solution, but if you do, explain why it was useful. Use only the monitor procedures `void deposit (int amount)` and `void withdrawal(int amount)`, with no additional parameters (unless the parameters are used only to make the program's output more clear).
- Then, modify your solution so that withdrawals are serviced FCFS. For example, suppose the current balance is $200 and one customer is waiting to withdraw $300. If another customer then requests to withdraw $10, it must be delayed until the earlier withdrawal request has been completed. (Hint: record the amount parameters of the processes waiting to complete their withdrawals.)

(f) the One-Lane Bridge Problem (Exercise 9.17).

(g) the Bus Problem (Exercises 9.18 and 9.19). The CSP solution can use a manager process, but the other solutions cannot. Do not use `_signal_all` in the monitor solution.

(h) the Dining Philosophers Problem (Chapter 11).

21.9 *Set partition.* Follow the directions for Exercise 9.32, but solve the problem using the CSP preprocessor.

Assume implicit termination. Do not use output commands in guards.

Source files containing parts of this program come with the JR distribution. Run your program on each of the supplied data files.

21.10 *Dutch National Flag.* Follow the directions for Exercise 9.33, but solve the problem using the CSP preprocessor.

Your program may use output commands in guards. Your solution can end in deadlock (but only after finishing sorting).

Source files containing parts of this program come with the JR distribution. Run your program on each of the supplied data files.

21.11 *Pairing Problem* [20]. Given are $N$ processes, each corresponding to a node in a connected graph. Each process has one or more neighbors to which it is connected. The goal is for each process to pair itself with one of its neighbors. When the processes finish "pairing", each process is paired or single, and no two single processes are neighbors.

Denote the processes `node[i]`, for `i` between `0` and `N-1`. The graph connectivity is stored in a global `N × N` boolean matrix connect, where `connect[i][j]` is true if and only if `node[i]` is a neighbor of `node[j]`—`connect` is therefore symmetric. For all `i`, `connect[i][i]` is false.

All node processes execute the same algorithm, which terminates, and does not share variables other than *connect,* which cannot be modified. A node only exchanges messages with nodes to which it is connected. When done pairing, each process stores its pairing into its local integer variable `p`. That is, if nodes `i` and `j` pair with each other, then `p` in `node[i]` should be set to `j` and `p` in `node[j]` to `i`; otherwise, `p` in `node[i]` should be set to `i`. Consider the following (implicit termination) solution:

```
_specs {
  node [N] { pairup(); }
}

_process node(i) {
  int p = i;
  _do
    _i [j = 0 _to N-1] (p==i && connect[i][j])
                       node[j]?pairup() { p = j;}
```

```
    _o [j = 0 _to N-1] (p==i && connect[i][j])
                       node[j]!pairup() { p = j;}
  _od
  System.out.println(i + " paired with " + p);
}
```

(a) Briefly explain how the above solution works. Also, discuss whether it would work if `_do` were replaced by `_if`. (Note: it is not optimal in the sense of minimizing the number of single processes; that would make this problem *very* hard.)

(b) Suppose the above program were run under explicit termination. Exactly which processes, if any, will not terminate? Explain.

(c) Modify the above solution so it uses explicit termination (and all processes terminate) and output commands in guards. Do not introduce additional processes or shared variables. Your solution must be symmetric.

(d) Modify the above solution so it uses explicit termination (and all processes terminate) and does not use output commands in guards. Do not introduce additional processes or shared variables. Your solution need not be symmetric, but it still must reasonably distribute the "work" among the processes; e.g., do not have one process compute the pairings and send the results to the others.

Source files containing parts of this program come with the JR distribution. Run your program on each of the supplied data files.

21.12 *Pairing Problem.*

(a) Repeat the previous question using JR. Because JR does not have the equivalent of implicit termination or output commands in guards, your solution will be closest to that for Exercise 21.11(d). Create all processes within the same object. Use arrays of operations. Make sure your solution terminates correctly.

Source files containing parts of this program come with the JR distribution. Run your program on each of the supplied data files.

(b) Compare the CSP solution to Exercise 21.11 with your JR solution. Which was easier to program? to understand? etc.

21.13 *Set minimum problem* [35] (based roughly on Exercise 8.5 in Reference [7]). A set of $N$ integers is distributed over $N$ processes, so that each process has one integer value. The goal is for the processes to determine the minimum of the set. The processes repeatedly interact, with

each process trying to give away to another process the minimum value it has seen so far. If a process gives away its minimum value, then it terminates. Otherwise, it tries to interact again. Eventually, one process will be left and it will know the minimum of set.

All processes execute the same algorithm, which terminates and which does not share any variables.

Define *MinCount* and *MaxCount* as, respectively, the minimum and maximum number of values that any process has seen during execution of the program. The process's initial value is included in these counts.

(a) Solve this problem using implicit termination and output guards in commands. Hint: your program will be similar to the solution given to Exercise 21.11. Give *MinCount* and *MaxCount* for your solution and explain your answer.

(b) Suppose your solution to the previous part were run under explicit termination. Exactly which processes, if any, will not terminate? Explain.

(c) Modify your solution so it uses explicit termination (and all processes terminate) and output commands in guards. Do not introduce additional processes or shared variables. Your solution must be symmetric. A process no longer needs to terminate immediately after it gives away its value. Describe how processes in your solution behave in this regard. Give *MinCount* and *MaxCount* for your solution and explain your answer.

(d) Modify the above solution so it uses explicit termination (and all processes terminate) and does not use output commands in guards. Do not introduce additional processes or shared variables. Your solution need not be symmetric, but it still must reasonably distribute the "work" among the processes. In particular, *MaxCount* must be approximately *N*/2. Explain how your solution satisfies that requirement. Hint: consider first the straightforward solutions when *MaxCount* is 2 and when *MaxCount* is *N*. For simplicity, you may assume that $N >= 2$.

Source files containing parts of this program come with the JR distribution. Run your program on each of the supplied data files.

21.14 *Set Minimum Problem.*

(a) Repeat the previous question using JR. Because JR does not have the equivalent of implicit termination or output commands in guards, your solution will be closest to that for Exercise 21.13(d). Create all

processes within the same object. Use arrays of operations. Make sure your solution terminates correctly.

Source files containing parts of this program come with the JR distribution. Run your program on each of the supplied data files.

(b) Compare the CSP solution to Exercise 21.13 with your JR solution. Which was easier to program? to understand? etc.

# Appendix A
# Synopsis of JR Extensions

## Additional Syntax

JR extends Java with additional statements, types, etc. The syntax below shows these extensions in terms of the syntax and the notation used in Chapter 18 of the Java Language Specification (JLS) [28].[1] Similar to the JLS BNF, the BNF below uses the following conventions:

- BNF symbols appear in italics. Terminal symbols (i.e., tokens) appear in Courier. For example, parentheses appear as BNF symbols as *(* and *)* versus as ( and ) as tokens.[2]
- *[x]* denotes zero or one occurrence of *x*.
- {*x*} denotes zero or more occurrences of *x*.
- *x*|*y* means one of either *x* or *y*.
- Multiple alternatives for each rule are placed on separate lines (i.e., an implicit | is assumed between such lines).
- The nonterminal being defined in each rule is given a suffix:
  - *(JLS)* indicates that the nonterminal is defined in the JLS BNF and that the rule given here is in addition to those in the JLS BNF.
  - *(new)* indicates that the nonterminal is not defined in the JLS BNF, but it is defined entirely here.

[1] Specifically, the grammar at the time of this writing is described in `http://java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html#44467`.

[2] Due to the typeface used in this book, { and } do not appear to be italicized. There should be no confusion, though, because the BNF in this section uses no { or } token.

*Statement (JLS):*
`call` *Expression* ;
`send` *Expression* [`handler` *Expression*] ;
`forward` *Expression* [`handler` *Expression*] ;
`reply` [*Expression*] [`handler` *Expression*] ;
`P(` *Expression* `)` ;
`V(` *Expression* `)` ;
`receive` *Expression* ;
`inni` *InniArm* {`[]` *InniArm*} [`else` *Statement*]
`inni with` *Expression* `over` *InniSelectArm* {`[]` *InniSelectArm*} [`else` *Statement*]
`view` *Expression ViewArm* {*ViewArm*} [`else` *Statement*]

*InniArm (new):*
*InniArmOp* [`st` *Expression*] [`by` *Expression*] *Statement*
*InniArmOp* [`by` *Expression*] [`st` *Expression*] *Statement*

*InniArmOp (new):*
[ `(` *Quantifiers* `)` ] (`void` | *Type*) *Expression FormalParameters*
[`throws` *QualifiedIdentifierList*]

*InniSelectArm (new):*
*InniArmOp Statement*

*ViewArm (new):*
`as` *FormalParameters Statement*

*Literal (JLS):*
`noop`

*Type (JLS):*
`cap` *CapOrOpDecl*
`vm`
`remote` *Identifier* { `.` *Identifier* } *BracketsOpt*

*CapOrOpDecl (new):*
(`void` | *Type*) *CapOrOpDeclFormalParameters* [`throws` *QualifiedIdentifierList*]

*CapOrOpDeclFormalParameters (new):*
`(` *CapOrOpDeclFormalParameter* { `,` *CapOrOpDeclFormalParameter* } `)`

*CapOrOpDeclFormalParameter (new):*
[`final`] *Type* [*Identifier*] *BracketsOpt*

*MemberDecl (JLS):*
    *OpDecl*
    *OpMethodDecl*
    *SemDecl*
    *ProcessDecl*

*BlockStatement (JLS):*
    *OpDecl* ;
    *SemDecl* ;

*OpDecl (new):*
    `op` *CapOrOpDecl Identifier*

*OpMethodDecl (new):*
    `op` *CapOrOpDecl Identifier Block*

*SemDecl (new):*
    `sem` *Identifier VariableDeclaratorRest*

*ProcessDecl (new):*
    `process` *Identifier [ ( Quantifiers ) ] Block*

*Quantifiers (new):*
    *Quantifier* { , *Quantifier* }

*Quantifier (new):*
    ( *ForInitOpt* ; *[Expression]* ; *ForUpdateOpt* [ ; *[Expression]* ])

*Primary (JLS):*
    `new remote` *Creator*
    `new vm()`
    `new sem` *[ ( Expression ) ]*
    `new op` *CapOrOpDecl*

A few notes regarding the syntax:

- The above BNF is "covering" in the same sense that the JLS BNF is. For example, the JLS BNF allows conflicting modifiers, e.g., `private public int x;`. But, that is not allowed in the language definition and is checked for by the translator. Similarly, the JR BNF allows, for example, `receive 1;`.
- A call invocation occurs when an operation is invoked from a call statement (as given in the above BNF) or as part of an *Expression,* e.g., `f()` or `x = g(3)/10` where `f` and `g` are operations or capabilities.
- In many cases, the expression in *InniArmOp* (or the receive statement) will simply be an identifier directly naming the operation to service. However, it can also be an identifier naming a capability variable, whose value is the operation to service. Moreover, it can be any expression that evaluates to a capability for an operation.
- *CapOrOpDeclFormalParameters* is the same as *FormalParameters* except that identifiers are optional.

- *OpMethodDecl,* which is defined in terms of *CapOrOpDeclFormalParameters,* requires that identifiers are specified. (That requirement is not shown in the above BNF.)
- *Quantifier*'s first expression must specify a new variable (see Section 4.1).
- Interestingly, all Java implementations we have tried allow identifiers and brackets to be mixed in the specification of formal parameters. For example, each of the following is legal:

```
public static void ff(int [] args []){}
public static void ff(int [] [] args){}
public static void ff(int args [] []){}
```

  However, the grammar given in the JLS allows only the last of the above forms. The JR implementation follows the Java implementation and allows such mixing.

## Additional Keywords

The following table lists all additional (with respect to Java) JR keywords and a brief description of how each is used. For more details on a particular keyword, consult the pages referenced in its entry in the Index.

| keyword | how used |
|---|---|
| `as` | arm of invocation `view` statement |
| `by` | scheduling expression in input statement |
| `call` | synchronous invocation |
| `cap` | declaration of capability type |
| `forward` | forwards invocation |
| `handler` | exception operation handler specifier |
| `inni` | input statement (for rendezvous and receive) |
| `noop` | capability constant value (do nothing) |
| `op` | operation declaration |
| `over` | selection method |
| `P` | semaphore primitive |
| `process` | process abbreviation |
| `receive` | receive statement |
| `remote` | remote object specifier |
| `reply` | reply statement |
| `sem` | semaphore declaration |
| `send` | asynchronous invocation |
| `st` | synchronization expression in input statement |
| `V` | semaphore primitive |
| `view` | view invocations |
| `vm` | virtual machine |
| `with` | selection method |

## Additional Classes

Details for the below additional classes appear in Appendix B.

| class | how used |
|---|---|
| `ArmEnumeration` | selection methods |
| `Invocation` | selection methods |
| `InvocationEnumeration` | selection methods |
| `Timestamp` | selection methods |

## Additional Predefined Methods

| method | how used |
|---|---|
| *oper*.`length()` | returns number of pending invocations of operation *oper* |
| `JR.exit(int status)` | exits program with `status` as exit status |
| `JR.registerQuiescenceAction(cap void() q)` | registers `q` as quiescence operation |

## Additional Predefined Fields

| field | how used |
|---|---|
| *r*.`remote` | remote reference for object reference *r* (often used as `this.remote`) |
| `vm.thisvm` | reference for vm on which this object resides |

## Additional Predefined Exceptions

| exception | when raised |
|---|---|
| `edu.ucdavis.jr.QuiescenceRegistrationException` | failure of `JR.registerQuiescenceAction` |

*This page intentionally left blank*

# Appendix B
# Invocation and Enumeration Classes

This appendix provides the details of the classes and methods that are used in conjunction with the inter-operation invocation selection mechanism described in Chapter 14.

Selection methods make use of `ArmEnumeration`, `InvocationEnumeration`, and `Invocation` objects to implement invocation selection algorithms. Each of the following classes is a member of the `edu.ucdavis.jr` package. These classes provide methods to facilitate the implementation of different selection algorithms. Each of these methods is discussed in detail below.

## ArmEnumeration Methods

The `ArmEnumeration` class provides access to the invocations of an associated input statement through the enumeration of a number of `InvocationEnumeration` objects (one per arm of the input statement). The `ArmEnumeration` class supports the following methods.

### hasMoreElements

public boolean **hasMoreElements**()

Tests if this enumeration contains more elements.

**Returns:**

`true` if and only if this enumeration object contains at least one more element to provide; `false` otherwise.

---

### nextElement

public Object **nextElement**() throws NoSuchElementException

Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

**Returns:**

the next element of this enumeration.

**Throws:**

NoSuchElementException — if no more elements exist.

---

**reset**
public void **reset**()

Resets the enumeration to the beginning of the underlying group of elements.

---

**size**
public int **size**()

Returns the number of elements in the underlying container.

**Returns:**

the number of elements in the underlying container.

## InvocationEnumeration Methods

The `InvocationEnumeration` class provides access to the invocations of a specific arm of an input statement through the enumeration of its pending invocations (represented by `Invocation` objects). The pending invocations are ordered by logical timestamp (longest pending first). The logical timestamp is an implementation specific data structure that ensures causal ordering of messages. Logical timestamps need not (and do not) correspond to actual time. The `InvocationEnumeration` class supports the following methods.

### hasMoreElements

public boolean **hasMoreElements**() throws RemoteException

Tests if this enumeration contains more elements.

**Returns:**

`true` if and only if this enumeration object contains at least one more element to provide; `false` otherwise.

**Throws:**

RemoteException — if a RemoteException occurs.

---

**nextElement**
public Object **nextElement**() throws NoSuchElementException, RemoteException

Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

**Returns:**

the next element of this enumeration.

**Throws:**

NoSuchElementException — if no more elements exist.
RemoteException — if a RemoteException occurs.

---

**reset**
public void **reset**() throws RemoteException

Resets the enumeration to the beginning of the underlying group of elements.

**Throws:**

RemoteException — if a RemoteException occurs.

---

**size**
public int **size**() throws RemoteException

Returns the number of elements in the underlying container.

**Returns:**

the number of elements in the underlying container.

**Throws:**

RemoteException — if a RemoteException occurs.

## Invocation Methods

The `Invocation` class provides access to a single pending invocation of a specific arm of an input statement. The values of the actual arguments within an invocation are accessed using a `view` statement as discussed in Section 14.2. The `Invocation` class supports the following method.

### getTimestamp

public Timestamp **getTimestamp**()

Returns the logical timestamp of the invocation.

**Returns:**

the logical timestamp of the invocation.

## Timestamp Methods

The `Timestamp` class stores a logical timestamp for an invocation. The `Timestamp` class implements `java.lang.Comparable` and supports the following methods.

### equals

public boolean **equals**(Object obj)

Compares this object to the specified object. The result is true if and only if the argument is not null and is a `Timestamp` object that contains the same logical timestamp value as this object.

**Parameters:**

obj — the `Object` with which to compare.

**Returns:**

`true` if the objects have the same logical timestamp; `false` otherwise.

---

**compareTo**

public boolean **compareTo**(Timestamp another Timestamp)

Compares two `Timestamp` objects numerically.

**Parameters:**

anotherTimestamp — the `Timestamp` with which to compare.

**Returns:**

the value 0 if this `Timestamp` is equal to the argument `Timestamp`; a value less than 0 if this `Timestamp` precedes in time the argument `Timestamp`; and a value greater than 0 if this `Timestamp` succeeds the argument `Timestamp`

---

**compareTo**

public boolean **compareTo**(Object obj)

Compares this `Timestamp` object to another object. If the object is a `Timestamp`, this function behaves like `compareTo` (`Timestamp`). Otherwise, it throws a `ClassCastException` (as `Timestamp` objects are only comparable to other `Timestamp` objects).

**Parameters:**

obj — the `Object` with which to compare.

**Returns:**

the value 0 if this `Timestamp` is equal to the argument `Timestamp`; a value less than 0 if this `Timestamp` precedes in time the argument `Timestamp`; and a value greater than 0 if this `Timestamp` succeeds the argument `Timestamp`

**Throws:**

ClassCastException — if the argument is not a `Timestamp`.

# Appendix C
# Program Development and Execution

This appendix provides a brief overview of the components of the JR system and the way they are used to develop and execute JR programs. Detailed descriptions of commands and tools are contained in the manual pages that are part of the JR distribution and in the JR webpage.

## Basics of Translation and Execution

Section 1.6 outlines the basic steps and tools involved in translating and executing a JR program. As described there, JR program code appears in files with `.jr` suffixes. The `jrc` tool translates these files to Java code, which is then compiled using the Java translator. The `jr_rmic` tool then adapts the resultant Java bytecode to execute with RMI. The `jrrun` tool then executes the resultant code on the Java Virtual Machine (JVM).

Section 1.6 also mentions additional tools to simplify translation and execution: `jr`, which combines the basic translation and execution steps; and `jrgo` and `jrgox`, which are even simpler versions of `jr` and `jrrun`. The `jrgo` and `jrgox` tools will work with most programs.

See the JR webpage for specific details on how to use all of these tools. For example, it describes the necessary settings for environment variables, search paths, etc.

## Preprocessors

Chapter 21 describes the three preprocessors for use with JR. The first, `ccr2jr`, converts source code written using a form of conditional critical regions (CCRs) into JR code. The second, `m2jr`, converts source code written using a form of monitors into JR code; it also supports several different monitor signaling disciplines. The third preprocessor, `csp2jr`, converts source code written in a variant of Communicating Sequential Processes (CSP) into JR code. See the man pages for descriptions of the preprocessors, including command-line options.

## Multiprocessor Programs

When a JR program is executed on a single processor, JR processes are executed one at a time in an interleaved fashion. However, on shared-memory multiprocessors, the JR implementation supports true concurrency. In either case, the JR implementation simply uses the underlying Java implementation. (See Appendix D for an overview of the JR implementation.)

## Distributed Programs

A JR program is treated as a distributed program if it makes explicit use of virtual machines. In this case, it executes in cooperation with an execution-time manager, `jrx`. Execution of `jrx` starts automatically when a JR program first creates an instance of a `vm`.

A new virtual machine is created on physical machine `X` by executing `create vm()` on `X`. As described in Chapter 10, machine `X` can be specified in either of two ways. The simplest way is to use a string expression that gives the symbolic name of a machine; this is of course installation dependent. The second way is to specify an existing virtual machine; in which case the new virtual machine is co-located with the existing one.

A distributed JR program can use only those hosts on a network to which a user has access. A user's login name must be the same on all these hosts. JR programs can be distributed over machines that run the same version of the JR implementation.

When a JR program creates a virtual machine on a host, the `jrx` manager uses the `rsh` command to initiate execution of the JR program on the remote host. Code is loaded onto the remote host as it is needed from the host on which program execution was initiated.

## Automatic Termination

Section 4.5 discussed how JR programs can deal with program quiescence. The JR implementation's default behavior is to terminate a program when it becomes quiescent. This behavior can be controlled via command-line options to `jr` or `jrrun`: `-implicit` and `-explicit`. The first option specifies the default behavior; the second requires that the program terminate itself, e.g., via `JR.exit(0)`.

The JR implementation's default behavior is not to output anything when it terminates the program. However, that can be changed by specifying a positive "verbosity" level as a command-line option to `jrrun`, e.g., `-verbosity=1`.

These options cannot be used with `jrgo` or `jrgox`.

## Cautions and Pitfalls

We have tried to make the JR language easy to understand and consistent. However, like any programming language, there are a few things in JR that might cause surprises, which are summarized in the following list.

- undefined return value for invocation of `noop` (Section 3.3).
- static versus non-static processes (Section 4.3).
- mismatched variables in quantifiers (see Exercise 4.13).[1]
- passing object references (Section 7.8).
- passing object references across virtual machines and serializability (Section 10.7).
- the directory in which execution begins for new virtual machines (Section 10.8).
- side effects in synchronization and scheduling expressions (Section 9.5).
- premature GUI program automatic termination (Section 20.3).
- use of an operation after it has gone out of scope (Appendix E).

[1] A similar pitfall occurs for Java's for statement, but would not occur for a language with notation that specifies the index variable once, e.g., `for i := 0 to 9`.

# Appendix D
# Implementation and Performance

The JR implementation has two major components: the translator and the run-time system (RTS). The JR translator extends the Java[1] compiler available in SUN's JDK, Version 1.2.1 to support JR-specific features. This extension was, for the most part, straightforward, but did require modifications to allow variables to be used as methods (to support capabilities). The translator converts JR programs into standard Java programs that are supported by the JR run-time system. Appendex C briefly describes the translation and compilation process. This appendix gives an overview of the RTS, focusing on how it implements the concurrency features of JR.

The JR run-time system is implemented in standard Java. The RTS provides the environment in which the compiled program executes. In particular, it supports creating virtual machines and remote objects, invoking and servicing operations, and loading class files over a network.

When a user starts execution of a JR program, the RTS creates one JR virtual machine (JRVM) on the local physical machine. After initializing the JRVM, the RTS then executes the program's main method. Each JRVM executes in a single Java virtual machine. JRVMs exchange messages using Java's Remote Method Invocation (RMI).

The RTS hides the details of the network from the executable program; i.e., the number of machines and their topology is transparent. When a request for a service provided on another machine is initiated—e.g., creating a remote object or invoking an operation—the invocation of the appropriate method is remotely executed via RMI on the remote machine. Results from such requests are transmitted back through the standard RMI method return.

The remainder of this appendix describes how the RTS implements virtual machines, remote objects, operations, invocation statements, and input statements. The last section discusses the relative performance of the various thread-interaction mechanisms.

## D.1 JR Virtual Machines

Each JR virtual machine is a small Java program that provides a thin layer over the standard Java virtual machine to allow remote object creation. To create a new virtual machine, a thread must contact a centralized virtual machine manager, called JRX, which plays a role similar to that of SR's SRX [9]. JRX contacts (via `rsh` or a specified alternate program) the physical host on

[1] JR currently extends Java 1.4. We have plans to further extend JR to incorporate the changes in Java 1.5.

which the virtual machine is to be created and initiates the execution of the `jrvm` program. Once created, a virtual machine executes a remote method invocation to contact JRX and register itself as ready to receive requests. A reference to the virtual machine is then returned to the instantiating thread so that remote objects can subsequently be created on the virtual machine.

## D.2 Remote Objects

Remote objects are created, using a new expression, on either the local virtual machine or on a remote virtual machine. Upon execution of the `new` expression, the virtual machine on which the object is to be created is contacted (via RMI) and passed the type of the object to be instantiated, the types of each of the constructor's arguments, and each of the arguments. The virtual machine then uses Java's reflection support to create a new instance of the object. After the new instance is created, a remote "reference object" is returned to the instantiating thread.

These "reference objects" are implemented as `java. io. Serializable` proxy objects. Each "reference object" contains references to each of the operations defined within the remote object's class. These operation references are actually operation capabilities that also implement the `java,io.Serializable` interface. The subclassing of the remote proxy objects mimics the inheritance hierarchy of the user-level JR classes with which they are associated. As such, a reference to a remote `java.lang.Object` object can refer to a remote `java.lang.String` object.

### D.2.1 Remote Class Loading

The dynamic class loading described in the Java RMI specification [45] allows for class files to be loaded from either the local CLASSPATH or from a predefined URL. The JR run-time system requires only that necessary class files for the program be accessible through the CLASSPATH at the originating host (where the program is initially executed). When a remote object is created, the necessary class files are retrieved from the JRX object on the originating host through a custom network class loader. This reduces the amount of setup required by the user and eliminates the need for a separate server (e.g., an http or ftp server) to provide file access.

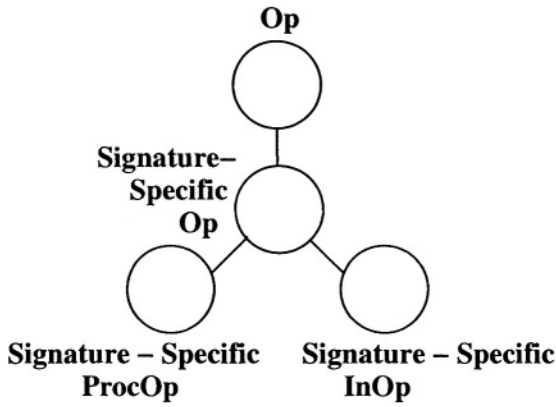


*Figure D.1.* Actual JR operation inheritance hierarchy

## D.3 Operations and Operation Capabilities

Figure D.1 shows the actual inheritance hierarchy of operation classes in the RTS. Each operation serviced by a method is implemented as a separate ProcOp object defined within the class that declares the operation. Thus, a ProcOp object may be serviced by a private method. Invocations of the operation are translated into invocations of the appropriate method (i.e., `call`, `send`, etc.) in the ProcOp object. This translation is very similar to the common technique of simulating a method reference (not directly representable in Java without using reflection) as an object with a well-defined interface. Figure D.2 shows an approximation of this translation.

```
public class A {
  public op
    void foo(int i) {
    ...
  }
  public void bar() {
    foo(3);
  }
}
```

(a) JR code

```
public class A {
  // implementation of ProcOp foo
  ...

  public void bar() {
    JRget_op_foo_intTovoid().call(3);
  }
}
```

(b) Generated Java code

*Figure D.2.* Translation of the invocation of a ProcOp

An operation serviced by `inni` statements is implemented as an InOp object that contains a message queue in which the arguments for each invocation are stored. An invocation of an InOp is translated into an appropriate method invocation on the corresponding object. All operations implement the `java.rmi.Remote` interface allowing methods to be invoked from remote hosts.

An operation capability is an object that implements the `java.io.Serializable` interface and that contains a reference to the appropriate signature specific operation. As such, operation capabilities can be passed over a network (at which point the Java RMI system will clone the capability object) while retaining a reference to original operation object. Invocations of the operation primitives (i.e., `call`, `send`, etc.) via a capability object are forwarded to the actual operation to which the capability refers.

## D.4 Invocation Statements

Synchronous invocations (i.e., `calls`) of an operation are translated directly into equivalent Java RMI statements that invoke the `call` method supported by the target operation object. If the operation is serviced by a method (i.e., it is a ProcOp), then the `call` method invokes the actual servicing method. If the operation is serviced by `inni` statements (i.e., it is an InOp), then the `call` method places a message containing the invocation's actual arguments into the operation's invocation queue and blocks until the message is serviced.

Since the JR run-time system is built using RMI, the support for asynchronous message passing is built upon a synchronous method invocation system. As such, the send statement is not truly asynchronous, but is actually semi-synchronous [11]. A send is implemented as an RMI invocation of the send method in the object that corresponds to the operation being invoked. A ProcOp's send method spawns a new thread to execute the method associated with the ProcOp and then returns, releasing the invoking thread. An InOp's send method places a message containing the actual arguments into the invocation queue and then returns. This strategy produces the desired effect, but does require the invoker to temporarily block.

### D.4.1 Inheritance

In Java, a method invocation causes a dynamic lookup to determine the actual method to invoke. However, this lookup is not done when accessing a data field. Since each operation in JR is implemented as a signature-specific Op object, the generated Java code must provide support for dynamic lookup of operations. This support is provided via access methods used to retrieve the appropriate operation object, an example of which can be seen in Figure D.2 (b).

## D.5 Input Statements

Input statements are the most complicated statements in the language and hence have the most complicated implementations. In its most general form, a single input statement can service one of several operations and can use synchronization and scheduling expressions or a `with/over` clause to select the invocation it wants. Moreover, an operation can be serviced by input statements in more than one thread, which then compete to service invocations.

Classes are fundamental to the implementation of input statements. They are used to identify and control conflicts between threads that are trying to service the same invocations. Classes are created dynamically as determined by the presence of operations in `inni` statements. A class of operations is an equivalence class of the transitive closure of the relation "serviced by the same input statement."

At run-time, when an `inni` statement is to be executed, its class of operations is determined by the operations that the `inni` statement is about to service. The RTS represents each class by a common lock. Those operations that exist in the same class share this common lock. This lock is used to synchronize the arrival of messages and to store a list of threads that are waiting to access the class. Further details can be found in [40].

At most one thread at a time is allowed to access the pending invocations of operations in a given class structure. That is, for a given class at most one thread at a time can be selecting an invocation to service or be appending a new invocation. Threads are given access to both pending and new invocations in a class in first-come/first-served order. Thus a thread waiting to access the invocations in a class will eventually obtain access as long as all methods in synchronization and scheduling expressions and in `with/over` clauses terminate eventually.

Note that synchronization and scheduling expressions and `with/over` clauses are evaluated by the executable program, not the RTS. We do this for two reasons. First, these expressions can reference objects such as local variables for which the RTS would need to establish addressing if it were to execute the code that evaluates the expression. Second, these expressions can contain invocations; it would greatly complicate the RTS to handle such invocations in a way that does not cause the RTS to block itself. A consequence of this approach to evaluating synchronization and scheduling expressions and `with/over` clauses is that the overhead of evaluating such expressions is paid for only by threads that use them.

## D.6 Quiescence Detection

JR supports an automatic quiescence detection mechanism that will either cause a program to terminate or that will invoke an operation if all threads are blocked. Java, however, does not provide a simple reliable means to detect such a scenario. Since the current implementation of JR does not modify the Java virtual machine, quiescence detection is implemented at the user-level. In the generated code, each time that a thread is created via an asynchronous invocation a "thread birth" is logged. Likewise, whenever such a thread terminates a "thread death" is logged. If a JR virtual machine goes idle at any time, then it notifies JRX, the virtual machine manage. Once JRX determines that all JR virtual machines are idle, and that there are not messages in transit, it executes the quiescence action (program termination by default).

This approach, though functional, has a negative impact on the overall performance of JR programs. The performance impact can be lessened by disabling quiescence detection when executing JR programs. Even so, since quiescence detection is a run-time option, programs that disable the feature still incur some overhead since the code was generated to support both options.

## D.7 Performance Results

A number of microbenchmarks have been used to study the performance of JR programs against equivalent RMI programs. The performance results demonstrate that remote method invocations in JR incur little overhead compared with equivalent invocations in standard RMI (upon which JR is built). Each of the following experiments was conducted on a cluster of 2 GHz Intel Pentium 4 workstations connected via a 10 Mbps Ethernet network. All experiments were conducted using SUN's JDK, Version 1.4.2.01 on Linux kernel 2.4.20-8.

The first experiment demonstrates the time needed to invoke an empty method. Table D. 1 shows the results of repeatedly invoking an empty ProcOp in JR and an empty method in Java. The method takes as an argument a single object that contains an array of a specified number of integers.

*Table D.1.* Time in microseconds to invoke an empty JR ProcOp and an empty Java method in a local object

| | *Object Size (ints)* | | | |
|---|---|---|---|---|
| | *1* | *1k* | *10k* | *100k* |
| JR | 0.1567 | 0.1562 | 0.1567 | 0.1576 |
| Java | 0.0118 | 0.0108 | 0.0112 | 0.0112 |
| JR time / Java time | 13.28 | 14.5 | 13.99 | 14.07 |

As evidenced by this experiment, a JR method invocation takes about fourteen times longer than a standard method invocation.[2] This is because an invocation of a ProcOp (or an InOp) in the current implementation of JR requires additional method invocations to support dynamic dispatch on operations, causal ordering of messages, quiescence detection, and communication exception handling. The overhead to support dynamic dispatch on operations could be reduced through implementation techniques similar to those used for methods. Further optimization at the compiler level may reduce the overhead of local operation invocations by eliminating the invocations for causal ordering.

The next experiment extends the previous experiment to measure invocations of an empty method in a remote object. Table D.2 shows the results of repeatedly invoking an empty ProcOp in JR and an empty method using standard RMI. The remote method takes as an argument a single object that contains an array of a specified number of integers.

The performance differences demonstrated in Table D.2 are attributable to method invocation overhead inherent in the current implementation of JR[3]. A remote method invocation begins by

[2]For an earlier version of JR that did not support quiescence detection, the ratio was about five [30].

[3]The differences are so minor, however, that instances of JR outperforming RMI, due to extraneous network traffic, have been observed.

*Table D.2.* Time in milliseconds to invoke an empty JR ProcOp and an empty RMI method in a remote object

| | *Object Size (ints)* | | | |
|---|---|---|---|---|
| | *1* | *1k* | *10k* | *100k* |
| JR | 0.55 | 1.04 | 4.75 | 37.42 |
| JDK RMI | 0.55 | 0.95 | 4.33 | 36.58 |
| JR time / JDK RMI time | 1.00 | 1.09 | 1.10 | 1.02 |

invoking the call method of the operation capability. The operation capability call method invokes the call method of the ProcOp. This invocation transmits the parameters to the remote host using RMI. At the remote host, the ProcOp call method invokes the actual user-defined method.

Table D.3 shows the results of multiple executions of the Readers/Writers program using both JR and RMI. In the JR solution the Readers/Writers server uses `inni` statements to service invocations of the different InOps associated with requesting and releasing read/write access. The RMI solution in this experiment, however, uses a semaphore-like approach to solving the Readers/Writers problem. Therefore, Table D.3 includes performance results for a roughly equivalent JR semaphore-like solution.

*Table D.3.* Time in milliseconds to complete execution of all iterations for all readers and writers

| | *Readers/Writers/Reader-Iters./Writer-Iters.* | | |
|---|---|---|---|
| | *20/10/3/3* | *50/15/3/3* | *100/30/3/5* |
| JR (inni) | 3612.2 | 4891.8 | 11498.8 |
| JR (semaphore) | 2839.8 | 3883.8 | 10611.4 |
| JDK RMI | 2554.4 | 3562.6 | 10594.4 |
| JR (inni) / JDK RMI | 1.41 | 1.37 | 1.09 |
| JR (semaphore) / JDK RMI | 1.11 | 1.09 | 1.00 |

The JR semaphore-like solution uses JR's semaphore abbreviations. These abbreviations translate into sends and receives on an InOp. Further optimization of the InOp process fairness code should improve the performance of this solution. It should be noted, however, that the JR semaphore solution did outperform the RMI solution when the RMI server was restarted between each test. Restarting the server between tests reduced the effectiveness of Java's Just-In-Time (JIT) compiler for the RMI solution. The results shown in Table D.3 for the RMI solution are for a persistent server.

The JR solution that uses an `inni` statement did not perform as well as the other two solutions. The performance difference is attributable to the current implementation of the `inni` statement's fairness preserving semantics. Table D.4 shows the percentage of time spent executing code that pertains to the fairness semantics for the Reader/Writer experiment. As shown in the table, a

large percentage of time is spent selecting the invocation to service from the currently pending invocations (e.g., readRequest).

*Table D.4.* JR (inni) Solution: Percentage of total execution time spent executing synchronization code for the Readers/Writers experiment

| | *Readers/Writers/Reader-Iters./Writer-Iters.* | | |
|---|---|---|---|
| | *20/10/3/3* | *50/15/3/3* | *100/30/3/5* |
| Invocation Sort | 0.19 | 0.34 | 0.27 |
| Select Invocation | 12.90 | 14.20 | 9.70 |
| Remove Invocation | 0.20 | 0.40 | 0.24 |
| Lock Acquire (Wait) | 67.90 | 67.40 | 86.70 |

Invocation selection takes a large percentage of the total execution time, in this program, because all invocations in an operation are examined until one is found that satisfies the arm's `st` clause. In this program, however, the `st` clause does not reference invocation arguments. Therefore, if the `st` clause is false, all invocations in the operation are examined needlessly. As an optimization for such cases, the `st` clause can be lifted out of the selection loop in the generated code.

Table D.5 compares the performance results of the standard sequential version of the Java Grande Forum *Fourier* Benchmark [19] against distributed versions written in JR and RMI.

*Table D.5.* Time in seconds to calculate the first n coefficients of the function $(x+1)^x$ defined on the interval [0,2]

| | *Number of coefficients* | |
|---|---|---|
| | *10000* | *100000* |
| Sequential Java | 35.47 | 398.55 |
| JR (1 Server) | 35.35 | 336.82 |
| JR (2 Servers) | 18.57 | 171.28 |
| JDK RMI (1 Server) | 35.22 | 400.98 |
| JDK RMI (2 Servers) | 17.84 | 170.87 |
| JR / JDK RMI (1 Server) | 1.00 | 0.84 |
| JR / JDK RMI (2 Servers) | 1.04 | 1.00 |

The distributed versions of the program divide the computation equally among the available servers. The JR program uses asynchronous message-passing to initiate each computation and then collects the results using an `inni` statement. The RMI version uses threads to concurrently initiate invocations of the remote method and to collect the results.

*This page intentionally left blank*

# Appendix E
# History of JR

JR is an extension of the Java programming language with additional concurrency mechanisms based on those in the SR (Synchronizing Resources) programming language [6,9]. So, the history of JR begins with the history of SR. Below, we summarize the history of SR (based on Appendix F of Reference [9]), discuss the development of JR, and present differences between SR and JR.

## History of SR

The basic ideas in SR—resources, operations, input statements, and asynchronous (`send`) and synchronous (`call`) invocations—were conceived by Andrews in 1978 and written up in early 1979; that paper eventually appeared in late 1981 [2]. The initial version of a full SR language, now called **$SR_0$**, was defined in the early 1980s and implemented by Andrews, Olsson, and several graduate students [3]. Based on experience with **$SR_0$**, Andrews and Olsson designed a new version in the mid 1980s; it added RPC, semaphores, early reply, and several additional mechanisms. Andrews and Olsson [4] describe the evolution of SR, explaining what was changed and why, as well as what was not changed and why not (also see Olsson's Ph.D. dissertation [38]). After using and testing the new version locally, the authors of SR began distributing SR in March 1988. Andrews et al. [6] describe SR version 1.0, explain the implementation, and compare SR with other languages.

Feedback from users of SR 1.0—and contributions from many of them—led to version 1.1, which was released in May 1989 [5]. Further experience, plus the desire to provide better support for parallel programming using shared variables and operations, led to the design of version 2.0, which is the version described in the SR book [9]. Additional work on SR has added a few new features (e.g., dynamic operations and receive from operation capabilities) and improved the implementation, and has led to the most recent version [44].

SR has been used primarily at universities to teach concurrent programming and in numerous research projects. A variant of SR, called MPD, with more C-like syntax has also been used in Andrews's recent book [8]. SR has been included in two survey articles on languages and mechanisms for concurrent programming [10, 12]. SR is also included as one of the languages in the proposed knowledge units for programming languages in the ACM's Curriculum 2001 [41].

## Development of JR

The initial ideas of how to represent SR's concurrency mechanisms in an object-oriented fashion were developed in 1997. Work began shortly thereafter by Tingjian Ge, then a UC Davis graduate student, on a proof-of-concept implementation. That implementation, although only functional for a small subset of what is now JR, demonstrated the feasibility and highlighted some of the implementation problems that would need to be addressed for a more complete language. Work on a new implementation of JR, the basis for the current implementation, began in January 1999 by Aaron Keen, then a UC Davis graduate student. During the summer of 1999, Justin Maris, then a UC Davis undergraduate student, developed an SR to JR translator, which was especially helpful in developing a suite of JR test programs from SR's existing suite. The first prototype JR implementation was used in a class at UC Davis in Fall 2000. JR has been used since then in UC Davis classes in programming languages and concurrent programming. The initial JR design and implementation ideas appeared in 2001 [29]. JR's approach to exception handling appeared in Reference [31] and its approach to inter-operation invocation selection appeared in Reference [32]. The complete integration of SR's concurrency mechanisms with Java's object-oriented mechanisms into a coherent language was the topic of Keen's Ph.D. dissertation [33]. Details of JR's invocation selection mechanism appear in Reference [40]. The most recent description of the design and implementation of JR appears in Reference [30].

Several other extensions to Java have modified its concurrency model to include, for example, asynchronous communication, distributed shared memory, and active agents. A few of these extensions have included support for some, limited SR-like concurrency mechanisms. None of these extensions provide the flexibility of operations, capabilities, and `inni` statements. Moreover, many of the previous extensions still require the user to manually start remote programs. For further discussion, see References [33] and [30]. JR can be classified a concurrent object-oriented programming language. Many such languages have been proposed, e.g., as discussed in a recent survey [12].

## Differences between SR and JR

The major differences between SR and JR are:

- JR, being an extension of Java, is object-oriented. SR is object-based: it has dynamic modules (resources) accessed via pointers (capabilities), but it lacks inheritance and virtual methods.
- JR takes an object-oriented view of operations (see Chapter 13), which was not possible in SR.
- JR provides exception handling mechanisms defined for invocations of operations (see Chapter 12). SR does not provide an exception handling mechanism.
- JR provides additional support for invocation selection (see Chapter 14).
- JR's implementation runs on UNIX-based and Windows-based systems. SR's implementation runs on only UNIX-based systems.

In addition, some other differences between SR and JR are:

- JR does not provide `call` or `send` restrictions on how operations may be invoked. (This feature might be added in a future version.)
- JR, consistent with its Java heritage, does not provide variable or result parameters.
- In JR, an operation does not "disappear" until there are no references to it. In SR, an operation "disappears" when control exits the block in which the operation is declared. For example,

suppose operation `f` is declared within process `P1` and a capability for `f` is held by another process `P2`. Then, in JR, `P2` can invoke `f` even after `P1` has terminated. (In many programs, such invocations are never serviced, although they might be via the capability for `f`.) In SR, such an invocation would cause a run-time error.

- JR has no restrictions on what operations or capabilities can be used in input statements. SR has restrictions that ensure that an input statement requires at most a single exchange of messages with a different virtual machine. This language design decision represents a classic tradeoff between expressiveness and efficiency. In the JR implementation, only those input statement that use such remote operations incur additional overhead. For details, see Reference [40].
- The semantics of invocation servicing in JR and SR differ slightly in the exact order of selection. See Section 9.5 and Exercises 14.4 and 14.5 for details.
- JR does not provide "optypes", which SR provides as a convenience for specifying the types of operations and capabilities.
- JR does not directly provide recursive operation declarations (e.g., `op f (int) returns cap f` in SR); however, the effect can be easily simulated as shown in Exercise 9.35.
- JR allows an invocation to be forwarded to any operation that has the same return type. SR requires that the types of the operations' parameters match too.
- JR does not allow `reply` to appear in constructor code. SR allows `reply` to appear in resource initialization code (the rough equivalent of constructor code). In JR programs, a similar effect can be simulated by making the code that would appear after the reply statement a separate process. To illustrate, suppose that JR allowed `reply` to appear in constructor code. Then the code in the `Philosopher` class in Section 11.1 could be rewritten; its `phil` process can be eliminated by moving that code into the constructor after a `reply`.
- JR does not have a concurrent invocation statement, but we are presently implementing one.
- JR provides parameterized virtual machines (Section 10.6).
- JR provides no equivalent of SR locate function, which is used to associate an integer with a physical machine name. (In SR, the on clause used in creating a virtual machine can also be an integer expression.)

Some of the above will be significant in converting programs written in SR to JR or vice versa. The JR webpage includes a feature comparison chart.

*This page intentionally left blank*

# References


[1] American National Standards Institute. *Reference Manual for the Ada Programming Language,* 1983. ANSI/MIL-STD-1815A.

[2] G. R. Andrews. Synchronizing resources. *ACM Transactions on Programming Languages and Systems,* 3(4):405-430, October 1981.

[3] G. R. Andrews. The distributed programming language SR—mechanisms, design, and implementation. *SOFTWARE — Practice and Experience,* 12(8):719–754, August 1982.

[4] G. R. Andrews and R. A. Olsson. The evolution of the SR language. *Distributed Computing,* 1(3): 133–149, July 1986.

[5] G. R. Andrews and R. A. Olsson. Report on the SR programming language, version 1.1. Technical Report TR 89-6, The University of Arizona, Department of Computer Science, May 1989.

[6] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems,* 10(1):51–86, January 1988.

[7] G.R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.

[8] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley Longman, Inc., Reading, MA, 2000.

[9] G.R. Andrews and R.A. Olsson. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1993.

[10] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys,* 21(3):261–322, September 1989.

[11] A. J. Bernstein. Predicate transfer and timeout in message passing systems. *Information Processing Letters,* 24(1):43–52, January 1987.

[12] J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys,* 30(3):291–329, September 1998.

[13] G. Bruno. Using Ada for discrete event simulation. *SOFTWARE — Practice and Experience,* 14(7):685–695, July 1984.

[14] A. Burns. *Programming in occam* 2. Addison Wesley, Reading, MA, 1988.

[15] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems,* 6(4):632–646, October 1984.

[16] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM,* 14(10):667–668, October 1971.

[17] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages,* pages 43–112. Academic Press, New York, NY, 1968.

[18] E. W. Dijkstra. The structure of the "THE" multiprogramming system. *Communications of the ACM,* 11(5):341–346. May 1968.

[19] Edinburgh Parallel Computing Centre. Java Grande Forum Benchmark Suite. `http://www.epcc.ed.ac.uk/research/javagrande/benchmarking.html`.

[20] R. A. Finkel, M. Solomon, and M. L. Horowitz. Distributed algorithms for global structuring. In *AFIPS Conference Proceedings, National Computer Conference (AFIPS 1979),* pages 455–460, Montvale, NJ, May 1979.

[21] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming,* 17(1):1–17, January 1988.

[22] M. D. Hill and J. R. Larus. Cache considerations for multiprocessor programmers. *Communications of the ACM,* 33(8):97–102, August 1990.

[23] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Communications of the ACM,* 29(12):1170–1183, December 1986.

[24] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques.* Academic Press, New York, 1972.

[25] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM,* 17(10):549–557, October 1974.

[26] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM,* 21(8):666–677, 1978.

[27] D. S. Johnson. Local optimization and the traveling salesman problem. In *Proceedings of the 17th Colloquium on Automata, Languages, and Programming,* pages 446–461, Berlin, 1990. Springer-Verlag.

[28] Bill Joy, Guy Steele, Jr., James Gosling, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley, Reading, Massachusetts, second edition, 2000. `http://java.sun.com/docs/books/jls/`.

[29] A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS 2001),* pages 575–584, Phoenix, Arizona, April 2001.

[30] A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. *ACM Transactions on Programming Languages and Systems,* May 2004. to appear.

[31] A. W. Keen and R. A. Olsson. Exception handling during asynchronous method invocation. In B. Monien and R. Feldmann, editors, *Euro-Par 2002 Parallel Processing,* number 2400 in Lecture Notes in Computer Science, pages 656–660, Paderborn, Germany, August 2002. Springer–Verlag.

[32] A. W. Keen and R. A. Olsson. An inter-entry invocation selection mechanism for concurrent programming languages. In Harald Kosch, László Böszörmény, and Hermann Hellwagner, editors, *Euro-Par 2003 Parallel Processing,* number 2790 in Lecture Notes in Computer Science, pages 770–780, Klagenfurt, Austria, August 2003. Springer–Verlag.

[33] Aaron William Keen. *Integrating Concurrency Constructs with Object-Oriented Programming Languages: A Case Study.* PhD dissertation, University of California, Davis, Department of Computer Science, June 2002.

[34] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *The Traveling Salesman Problem.* John Wiley & Sons, Chichester, 1985.

[35] G. M. Levin and D. Gries. A proof technique for Communicating Sequential Processes. *Acta Informatica,* 15:281–302, 1981.

[36] S. Mahadevan and R. K. Shyamasundar. Correctness preserving transformations for distributed programs. In R. E. A. Mason, editor, *Proceedings of the IFIP 9th World Computer Congress (IFIP 1983),* pages 307–313, Paris, France, September 1983. North-Holland/IFIP.

[37] U. Manber. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, Reading, MA, 1989.

[38] R. A. Olsson. *Issues in distributed programming languages: the evolution of SR.* PhD dissertation, The University of Arizona, Department of Computer Science, August 1986.

[39] R. A. Olsson. Using SR for discrete event simulation: A study in concurrent programming. *SOFTWARE—Practice and Experience,* 20(12): 1187–1208, December 1990.

[40] R. A. Olsson, G. D. Benson, T. Ge, and A. W. Keen. Fairness in shared invocation servicing. *Computer Languages, Systems and Structures,* 28(4):327–351, December 2002.

[41] Programming Language Knowledge Unit Focus Group. Proposed knowledge units for programming languages for curriculum 2001. *ACM SIGPLAN Notices,* 35(4):29–43, April 2000.

[42] M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, Cambridge, MA, 1986.

[43] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts.* John Wiley & Sons, New York, sixth edition, 2002.

[44] The SR programming language, version 2.3.2, August 1999. `http://www.cs.arizona.edu/sr/.`

[45] Sun Microsystems. *Java Remote Method Invocation Specification.* Sun Microsystems, Palo Alto, CA, 1997.

[46] Kent retargetable occam compiler. http://www.cs.kent.ac.uk/projects/ofa/kroc/.

[47] C.A. Waldspurger and W.E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating System Design and Implementation,* pages 1–11, Monterey, California, November 1994. USENIX.

[48] K. Walrath and M. Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs.* Addison-Wesley, Reading, Massachusetts, 1999. http://java.sun.com/docs/books/tutorial/uiswing/index.html.

# Index

# About the Authors

Ronald A. Olsson received the B.A. degree in mathematics and in computer science, and the M.A. degree in mathematics, from the State University of New York, College at Potsdam. He received the M.S. degree in computer science from Cornell University and the Ph.D. degree in computer science from The University of Arizona. Dr. Olsson has been at the University of California, Davis since 1986, where he is currently a Professor of Computer Science. He received the UC Davis Academic Senate Distinguished Teaching Award in 1995. His research interests include concurrent programming, programming languages, verification, systems software, operating systems, and computer security. Dr. Olsson co-authored (with Greg Andrews) the book "The SR Programming Language: Concurrency in Practice". For more information, see `http://www.cs.ucdavis.edu/~olsson`.

Aaron W. Keen received a B.S. degree in computer science and engineering, an M.S. degree in computer science, and a Ph.D. in computer science from the University of California, Davis. He has been an assistant professor at the California Polytechnic State University, San Luis Obispo since 2002. Dr. Keen's research interests include concurrent programming, programming languages, verification, compilers, and operating systems. For more information, see `http://www.csc.calpoly.edu/~akeen`.