Chapter - Four

Memory Management

Outline

- Overview of processors and memory
- Memory management definition
- Binding of Instructions and Data to Memory
- Executable code
- Simple memory management schemes
- Swapping
- Logical vs. Physical Address Space
- Memory protection with dynamic relocation (MMU)
- Contiguous memory allocation
- Segmentation
- Paging
- Segmentation with paging
- Memory caching

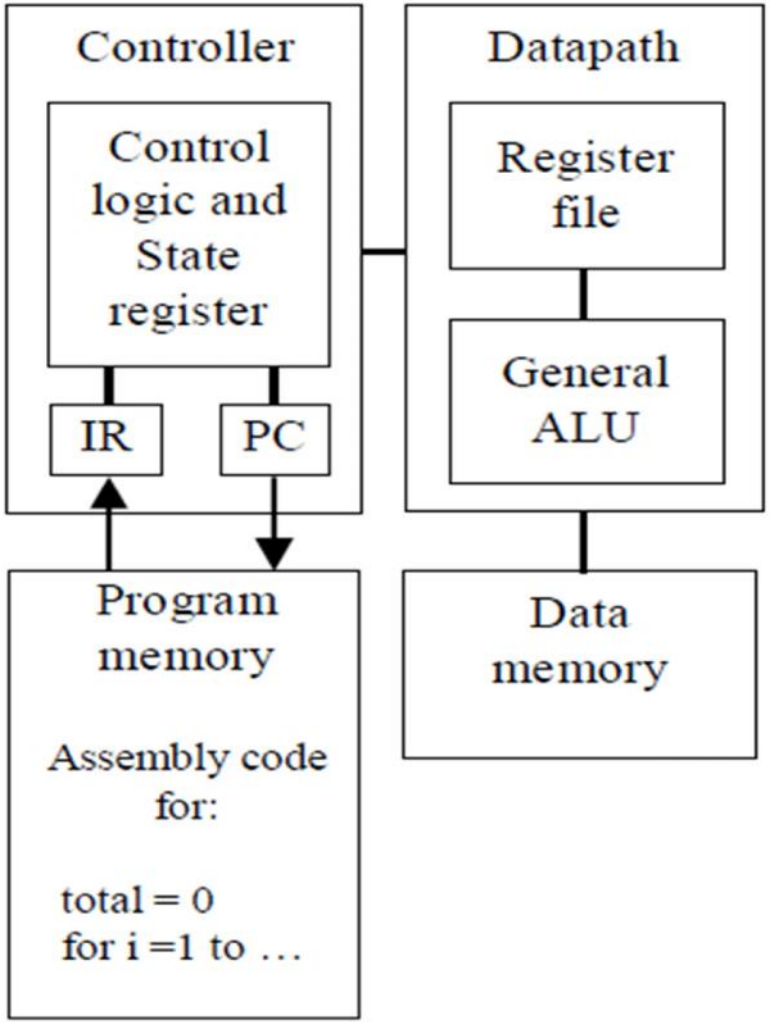
Memory Management

- The CPU fetches instructions and data of a program from memory;
- Therefore, both the program and its data must reside in the main memory (RAM and ROM).
- Modern multiprogramming systems are capable of storing more than one program, together with the data they access, in the main memory.

- A fundamental task of the memory management component of an operating system is to ensure safe execution of programs by providing:
  - Sharing of memory
  - Memory protection

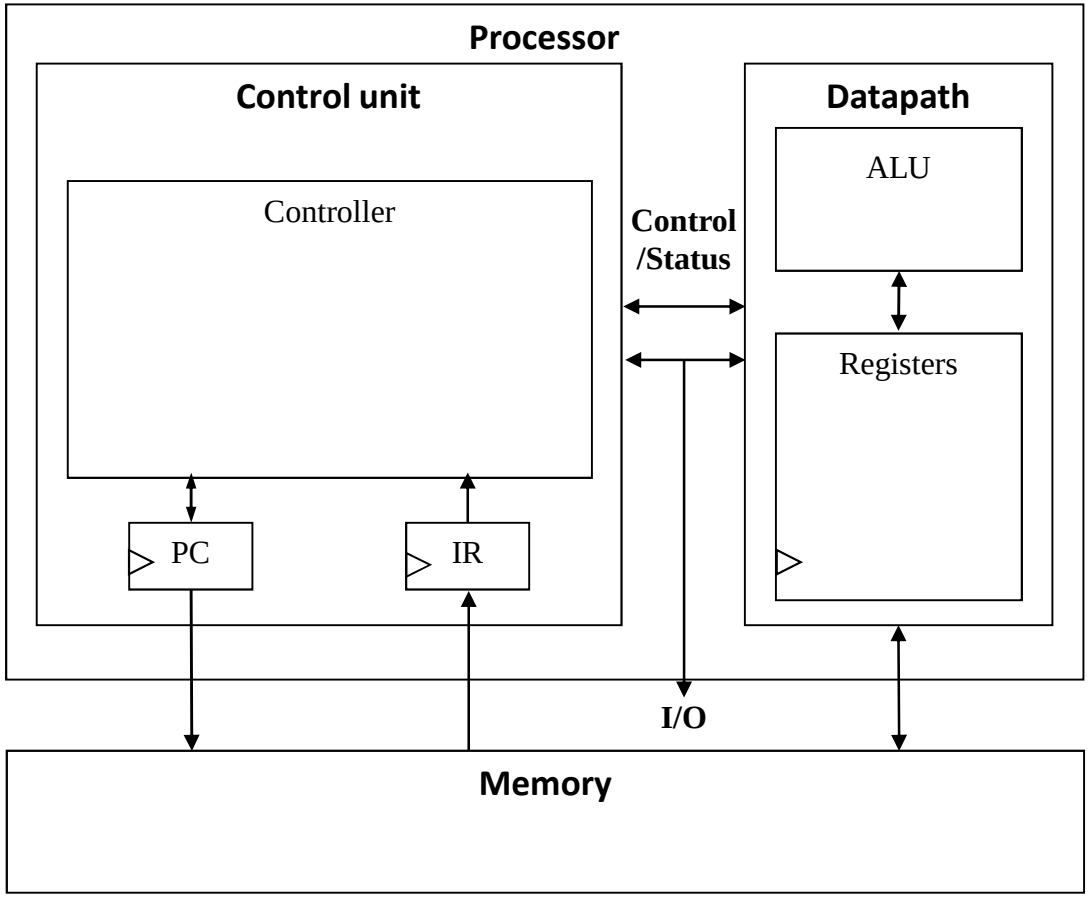
General-purpose processors

- Programmable device used in a variety of applications
  - Also known as “microprocessor”
- Features
  - Program memory
  - General data path with large register file and general ALU
- User benefits
  - Low time-to-market and costs
  - High flexibility
- “Pentium” the most well-known, but there are hundreds of others



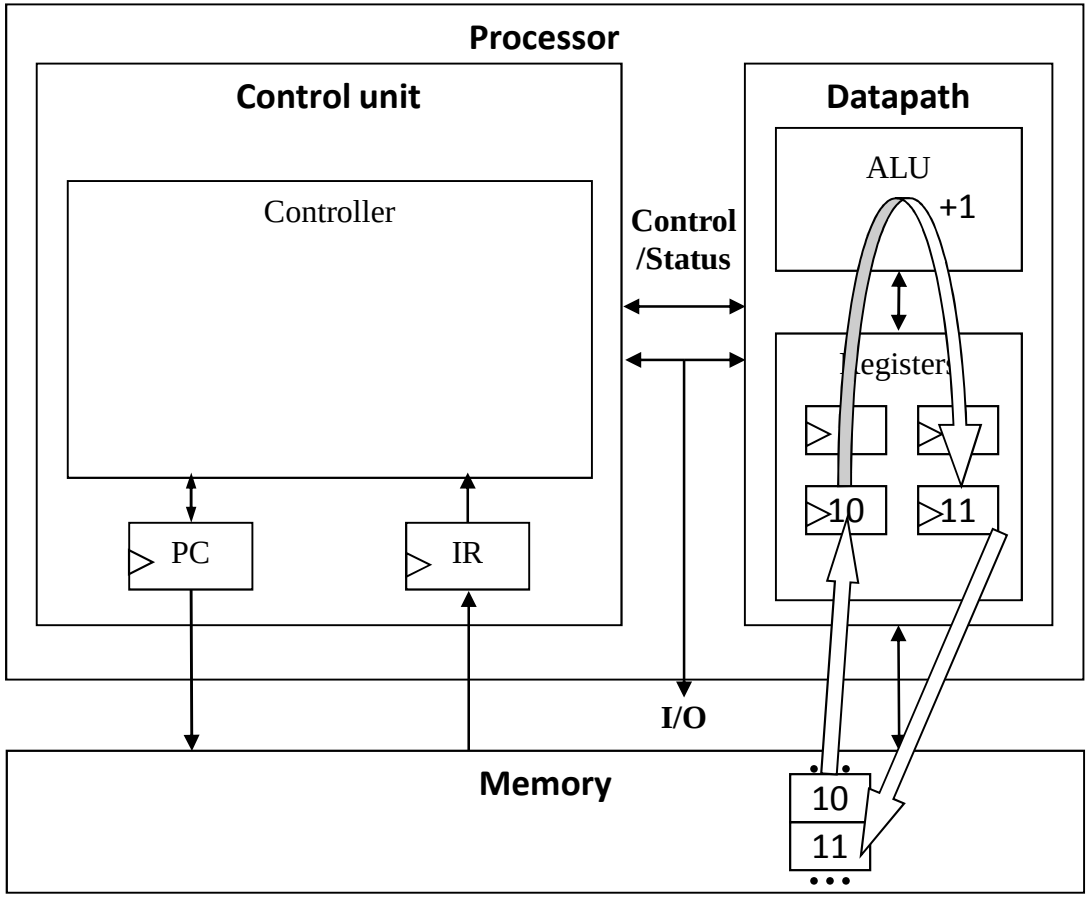
Basic Architecture

- It has Control unit and datapath
  - Note similarity to single-purpose processor
- Key differences
  - Datapath is general
  - Control unit doesn't store the algorithm – the algorithm is "programmed" into the memory



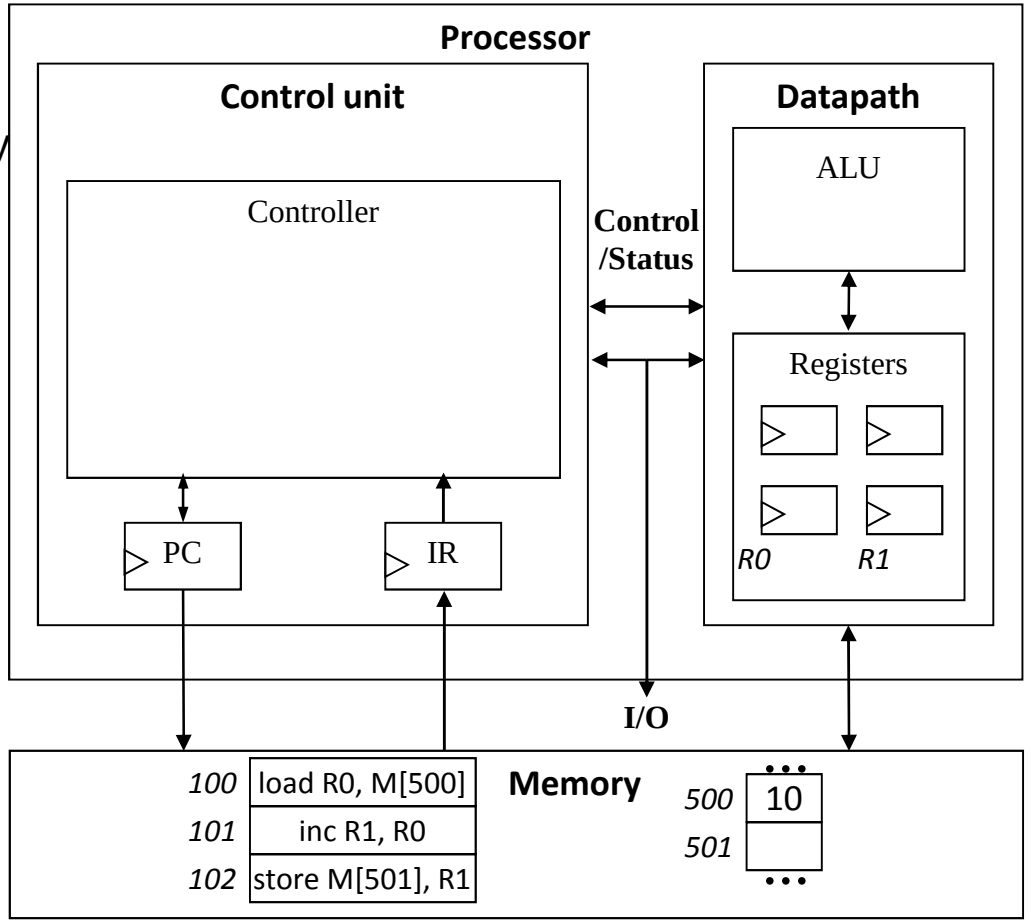
Datapath Operations

- Load
  - Read memory location into register
- ALU operation
  - Input certain registers through ALU, store back in register
- Store
  - Write register to memory location



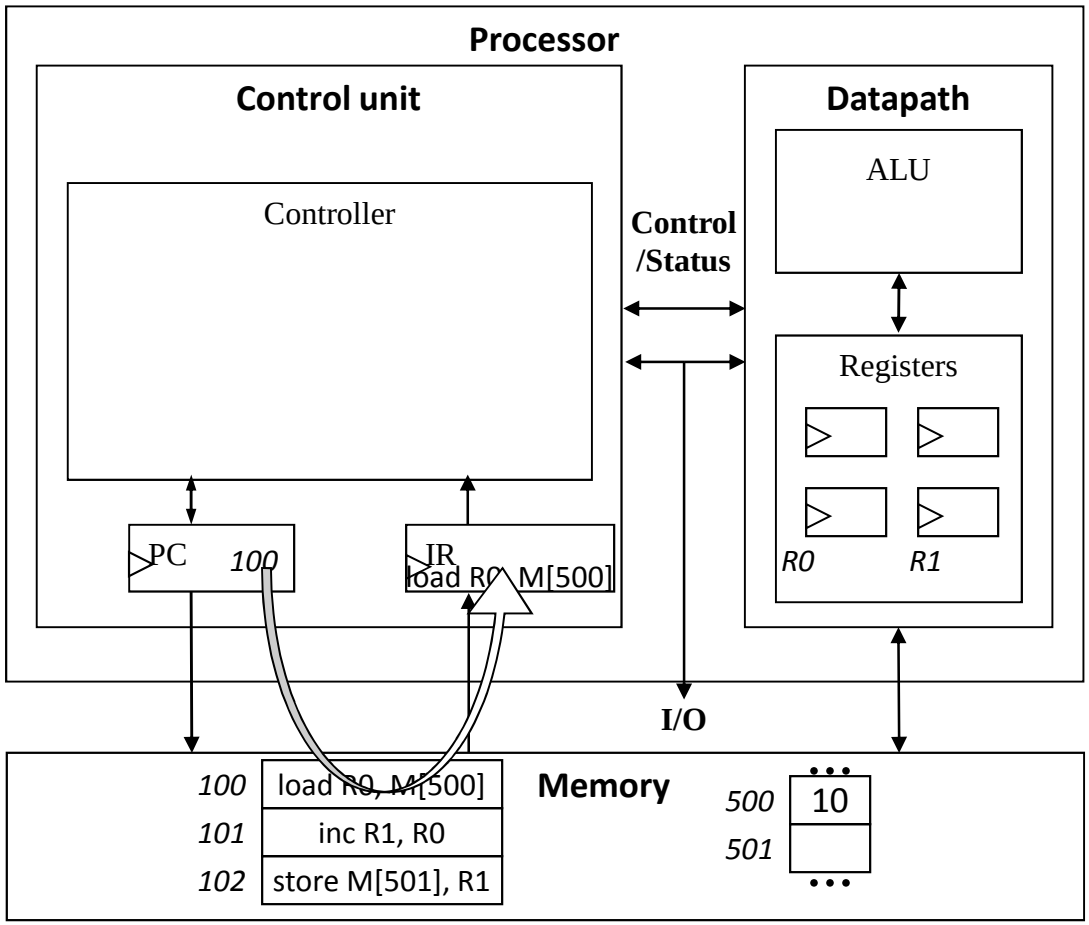
Control Unit

- Control unit: configures the datapath operations
  - Sequence of desired operations ("instructions") stored in memory – "program"
- Instruction cycle – broken into several sub-operations, each one clock cycle, e.g.:
  - Fetch: Get next instruction into IR
  - Decode: Determine what the instruction means
  - Fetch operands: Move data from memory to datapath register
  - Execute: Move data through the ALU
  - Store results: Write data from register to memory



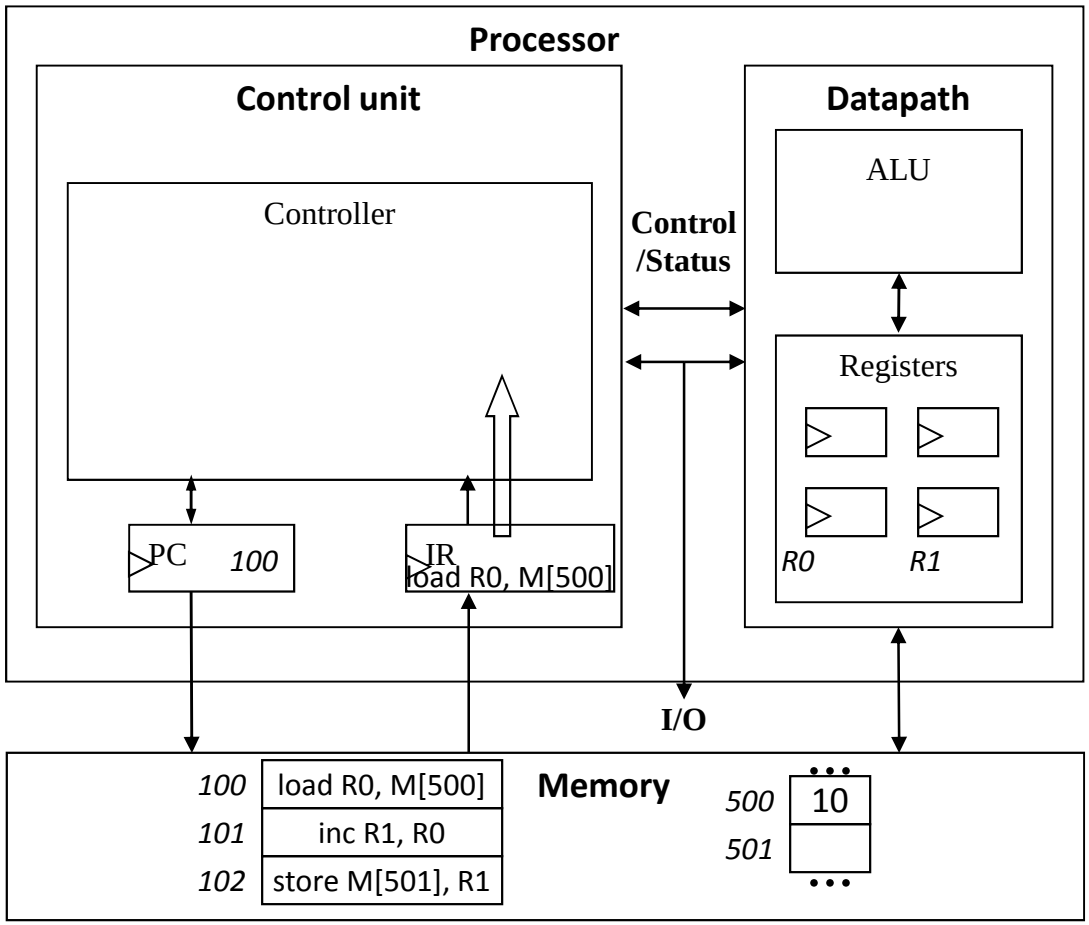
Control Unit Sub-Operations

- Fetch
  - Get next instruction into IR
  - PC: program counter, always points to next instruction
  - IR: holds the fetched instruction



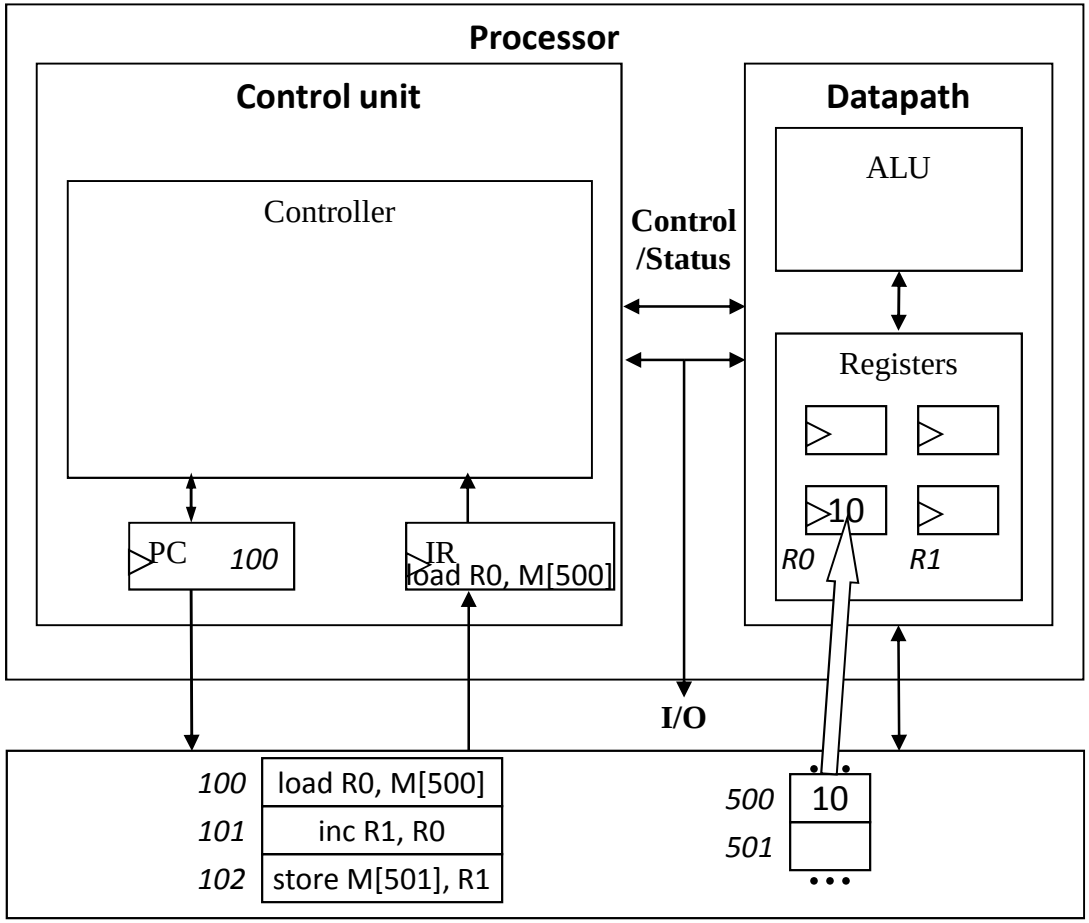
Control Unit Sub-Operations

- Decode
  - Determine what the instruction means



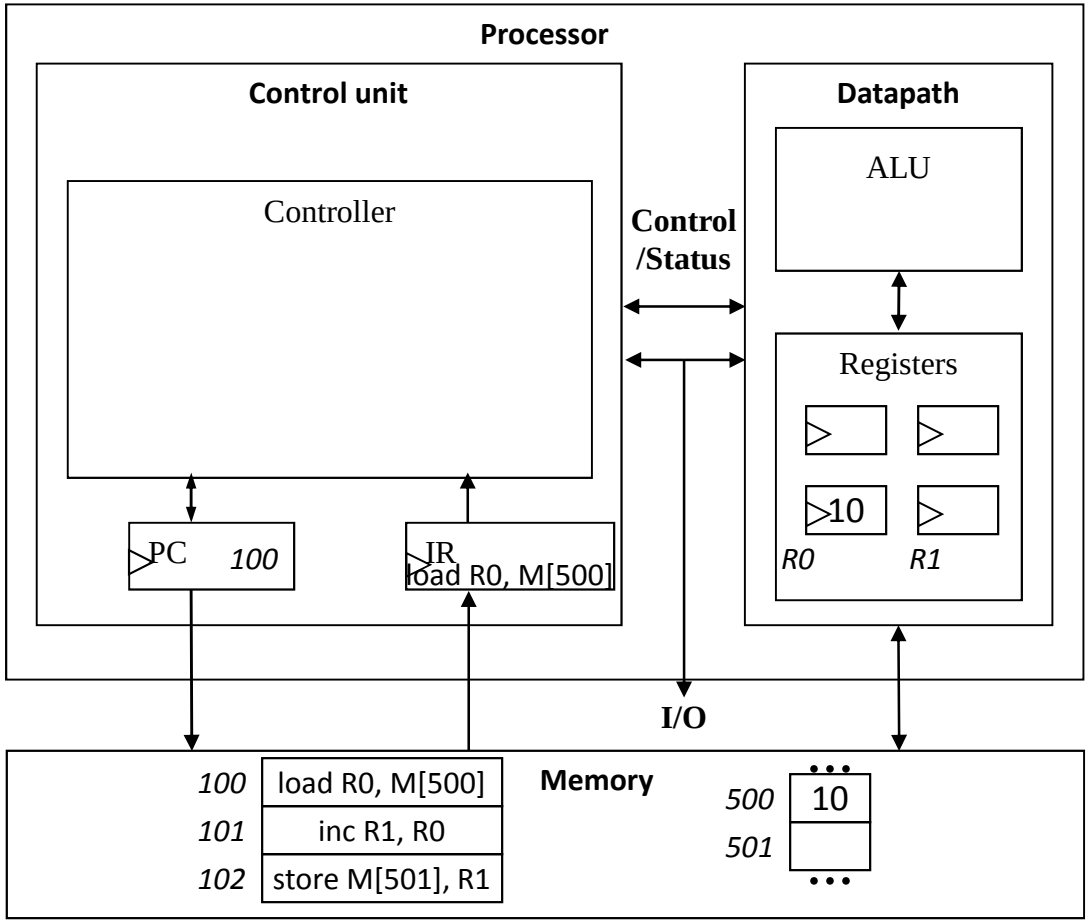
Control Unit Sub-Operations

- Fetch operands
  - Move data from memory to datapath register



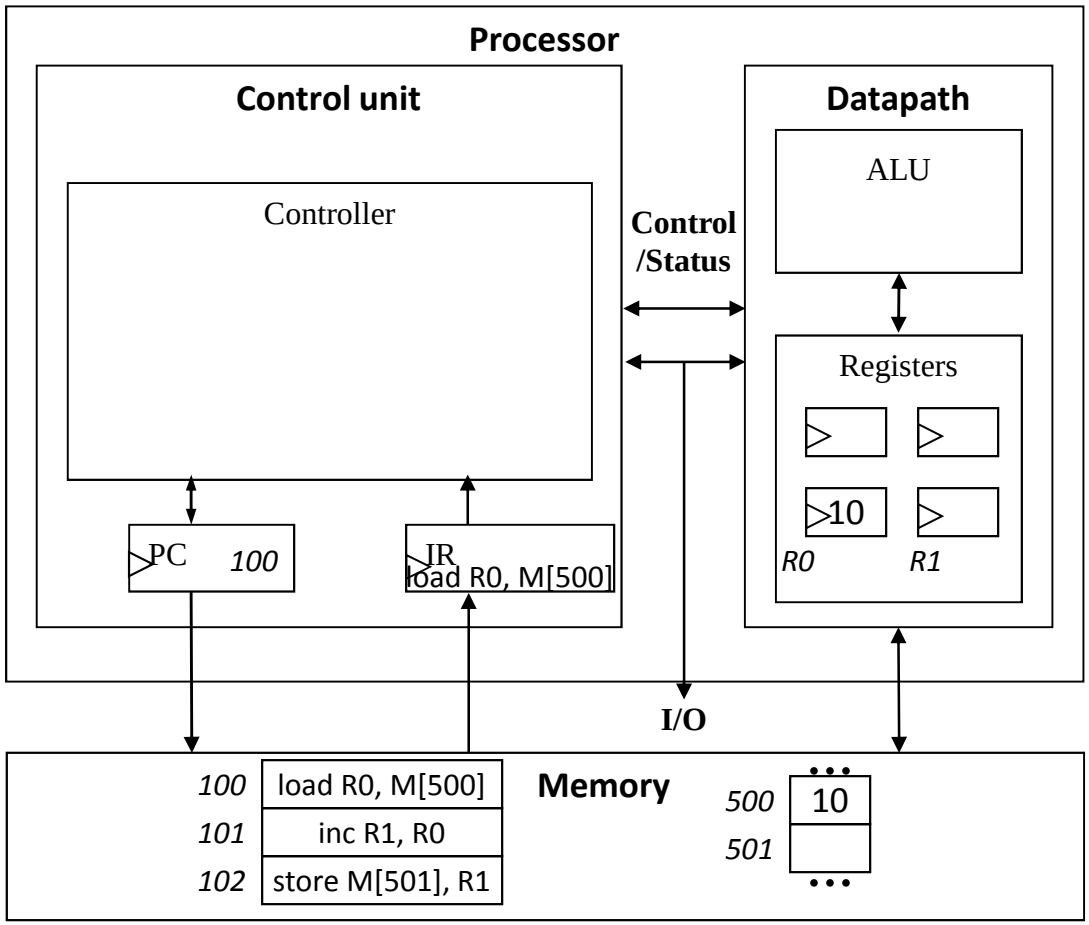
Control Unit Sub-Operations

- Execute
  - Move data through the ALU
  - This particular instruction does nothing during this sub-operation

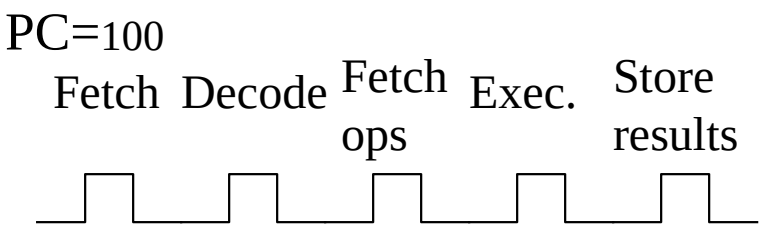


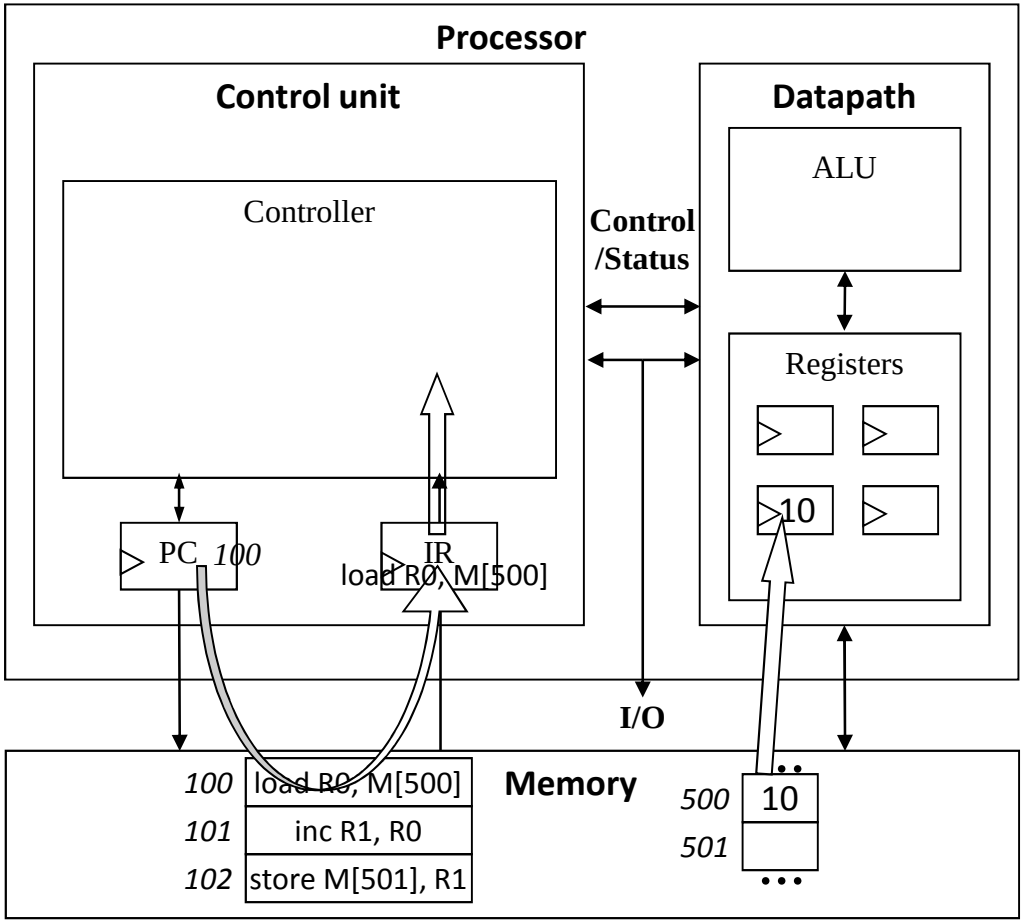
Control Unit Sub-Operations

- Store results
  - Write data from register to memory
  - This particular instruction does nothing during this sub-operation



Instruction Cycles

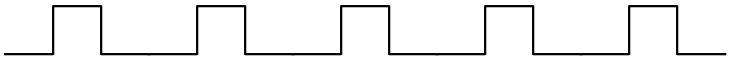




Instruction Cycles

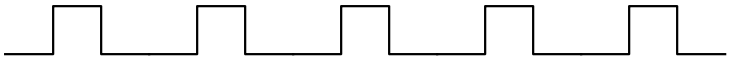
PC=100

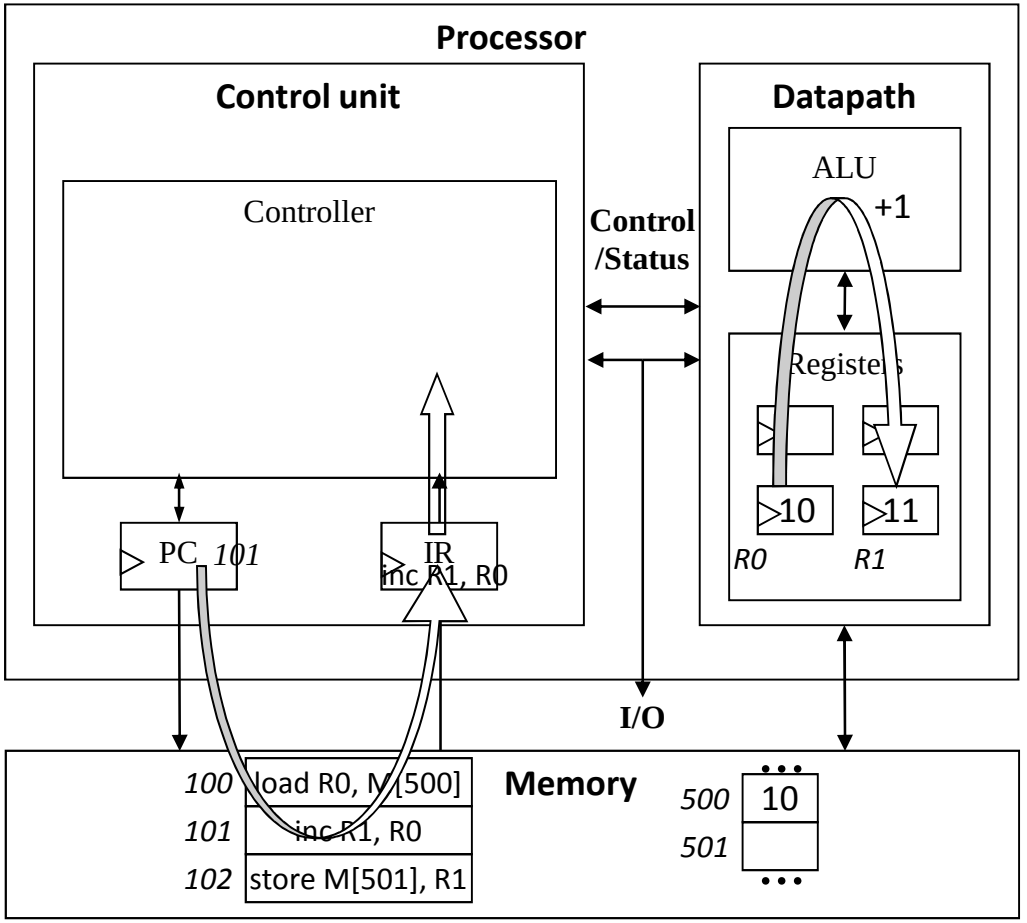
Fetch Decode Fetch ops Exec. Store results



PC=101

Fetch Decode Fetch ops Exec. Store results





Instruction Cycles

PC=100

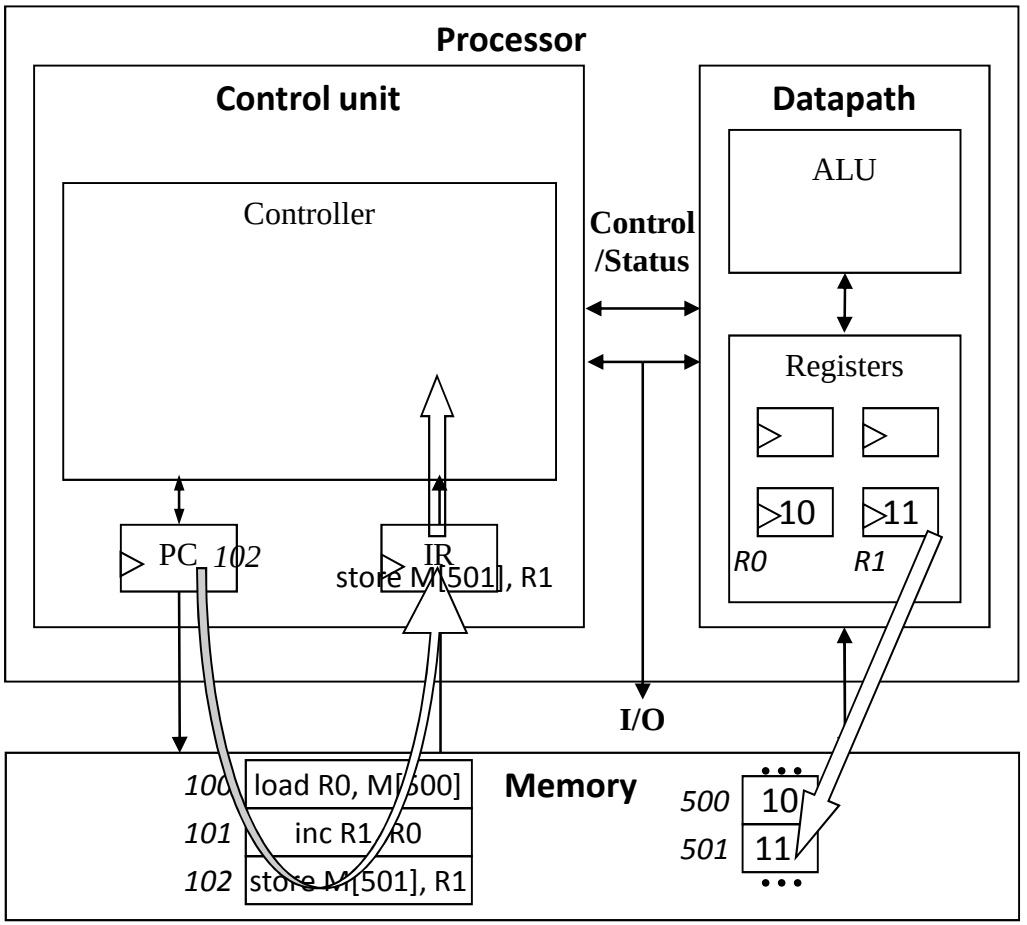
Fetch Decode Fetch ops Exec. Store results

PC=101

Fetch Decode Fetch ops Exec. Store results

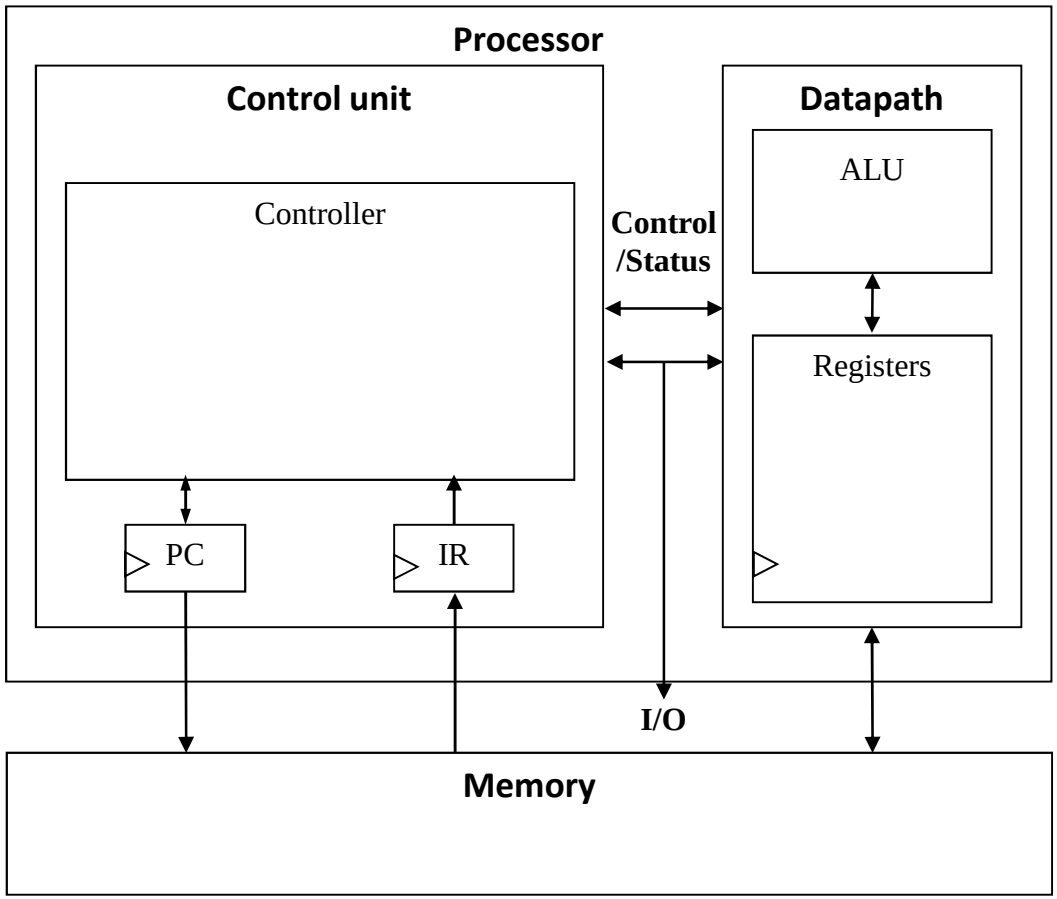
PC=102

Fetch Decode Fetch ops Exec. Store results



Architectural Considerations

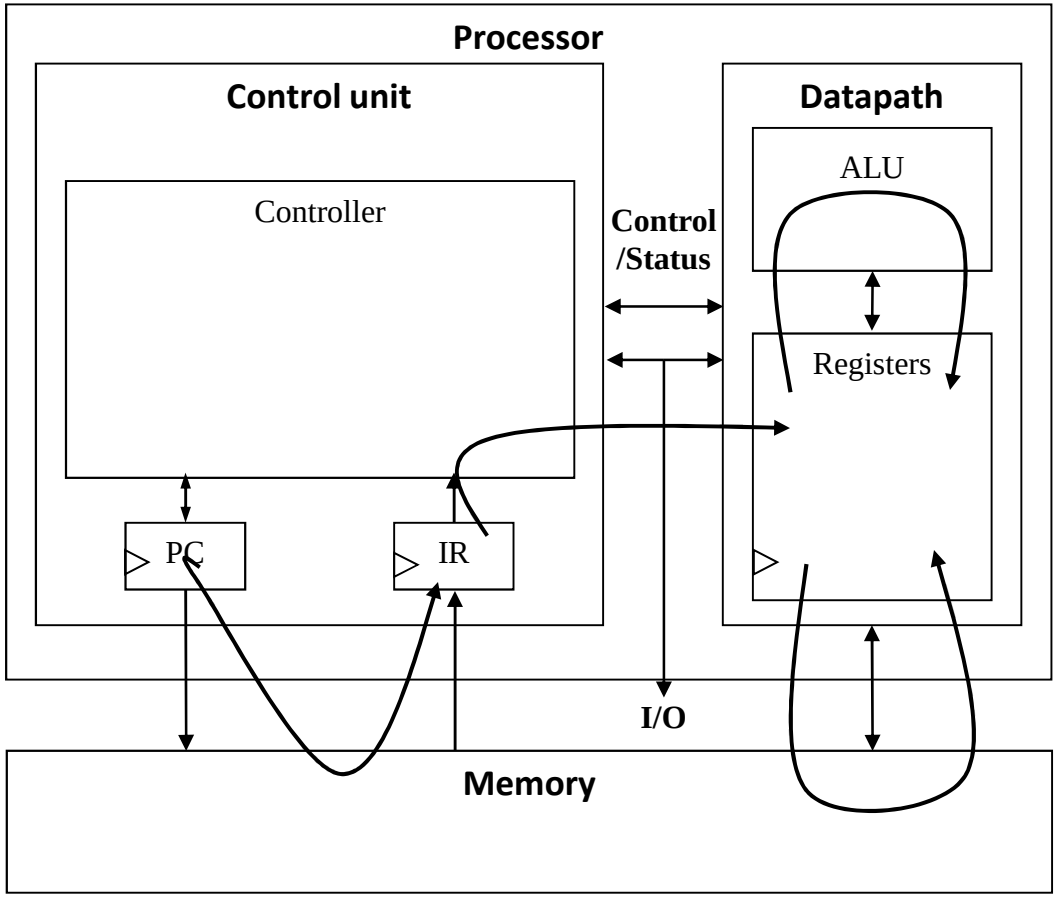
- N-bit processor
  - N-bit ALU, registers, buses, memory data interface
  - Embedded: 8-bit, 16-bit, 32-bit common
  - Desktop/servers: 32-bit, even 64
- PC size determines address space



Architectural Considerations

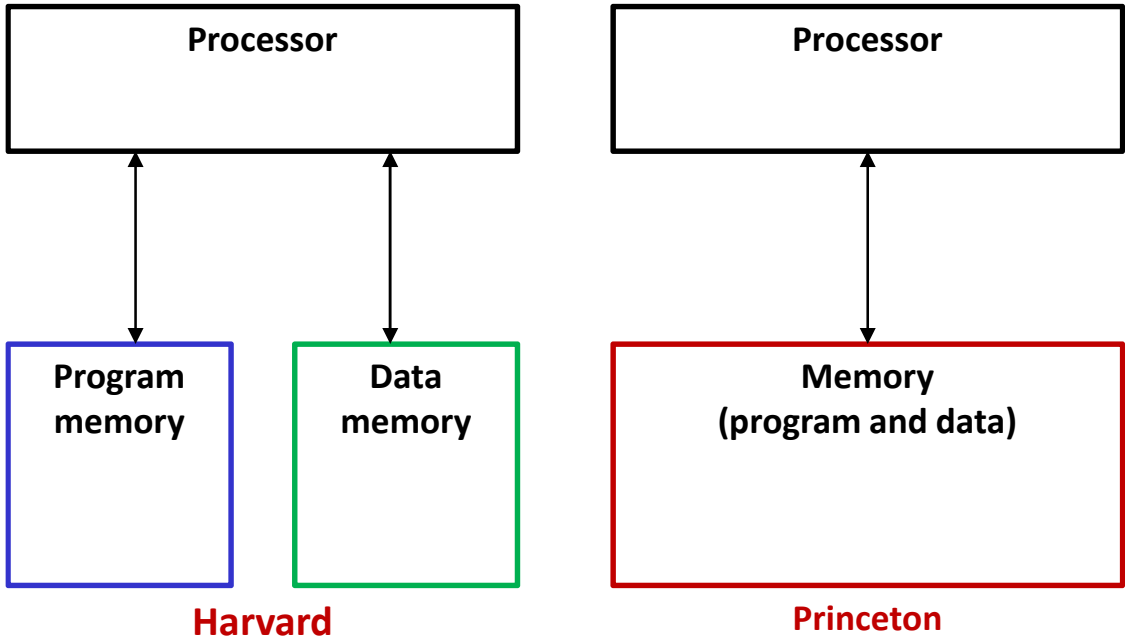
- Clock frequency
- Inverse of clock period
- Must be longer than longest register to register delay in entire processor
- Memory access is often the longest
- E.g clock cycle 10 nanoseconds =

$1/10x10^{-9}$ Hz =100 MHz



Two Memory Architectures

- Princeton
  - Fewer memory wires
- Harvard
  - Simultaneous program and data memory access



Memory Management...

Issues in sharing memory

Transparency

- Several processes may co-exist, unaware of each other, in the main memory and run regardless of the number and location of processes.

Safety (or protection)

- Processes must not corrupt each other (nor the OS!)

Efficiency

- CPU utilization must be preserved and memory must be fairly allocated.

Relocation

- Ability of a program to run in different memory locations.

Storage allocation

- Information stored in main memory can be classified in a variety of ways:
  - Program (code) and data (variables, constants)
  - Read-only (code, constants) and read-write (variables)
  - Address (e.g., pointers) or data (other variables);

- The OS, compiler, linker, loader and run-time libraries all cooperate to manage this information.

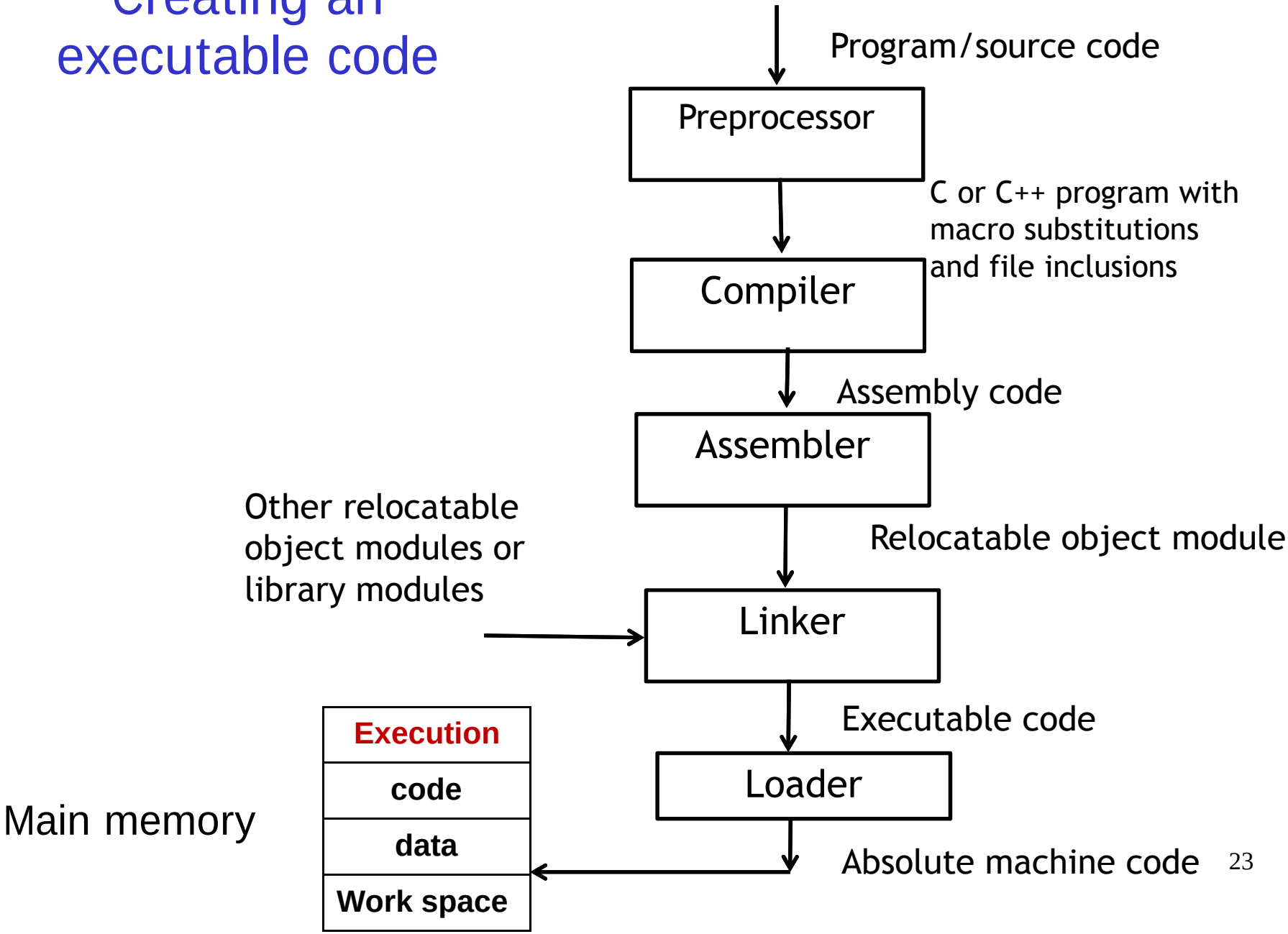
Memory Management: Definitions

- Relocatable - Means that the program image can reside anywhere in physical memory.
- Binding - Programs need real memory in which to reside. When is the location of that real memory determined?
  - This is called mapping logical to physical addresses.
  - This binding can be done at compile or run time.
- Compiler - If it's known where the program will reside, then absolute code is generated. Otherwise compiler produces relocatable code.
- Load - Binds relocatable to physical address.
- Execution - The code can be moved around during execution.

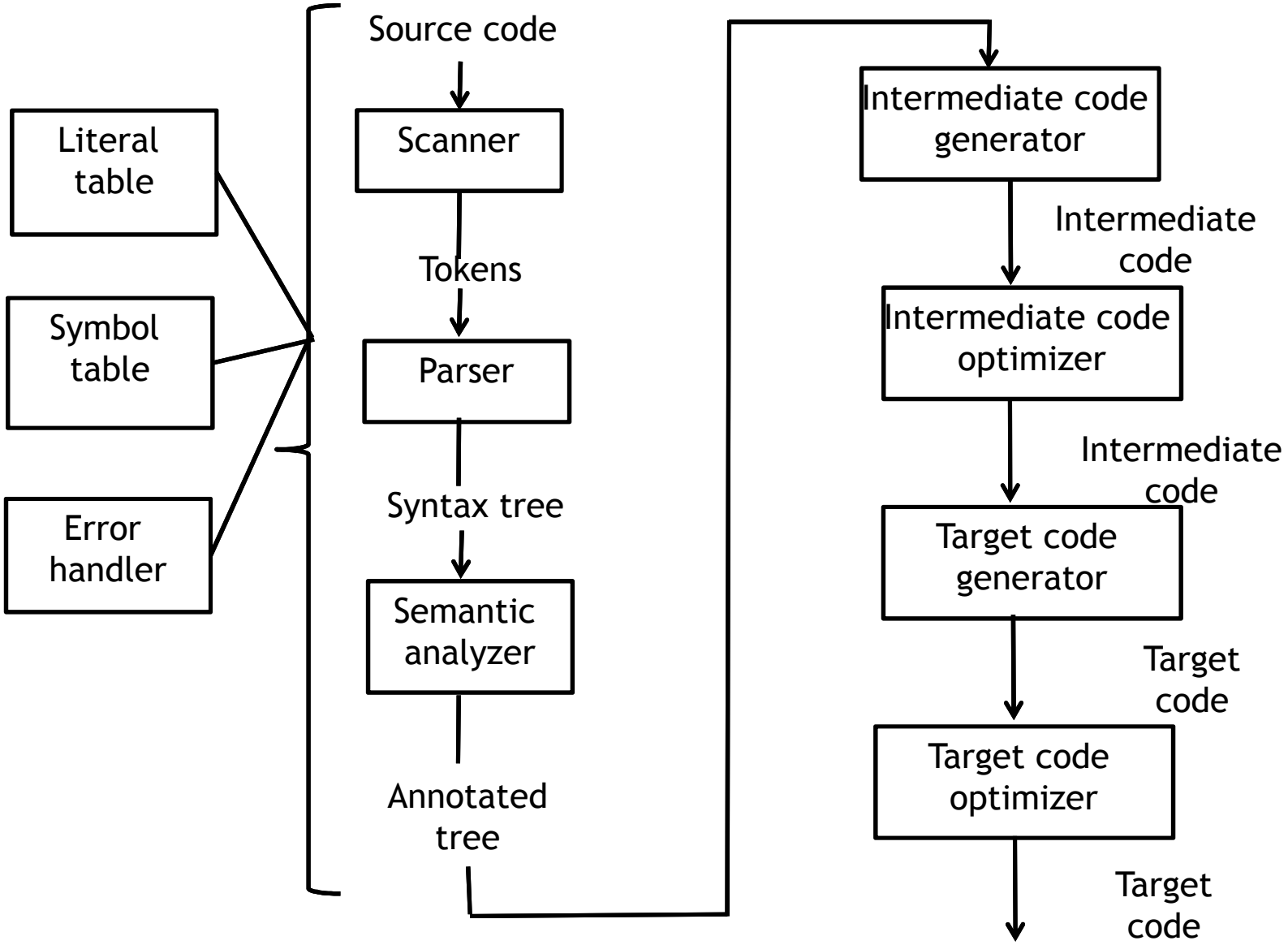
Creating an executable code

- Before a program can be executed by the CPU, it must go through several steps:
  - Compiling (translating) - generates re-locatable object code.
  - Linking - combines the object code into a single self-sufficient executable code.
  - Loading - copies the executable code into memory.
  - Execution - dynamic memory allocation.

Creating an executable code



Compiler


Source code
Scanner
Tokens
Parser
Syntax tree
Semantic analyzer
Annotated tree
Literal table
Symbol table
Error handler
Intermediate code generator
Intermediate code
Intermediate code optimizer
Intermediate code
Target code generator
Target code
Target code optimizer
Target code

Functions of a linker

- Linker collects and puts together all the required pieces to form the executable code.

- Issues:
  - Relocation
    - ✓Where to put pieces.
  - Cross-reference
    - ✓where to find pieces.
  - Re-organization
    - ✓new memory layout.

Functions of a loader

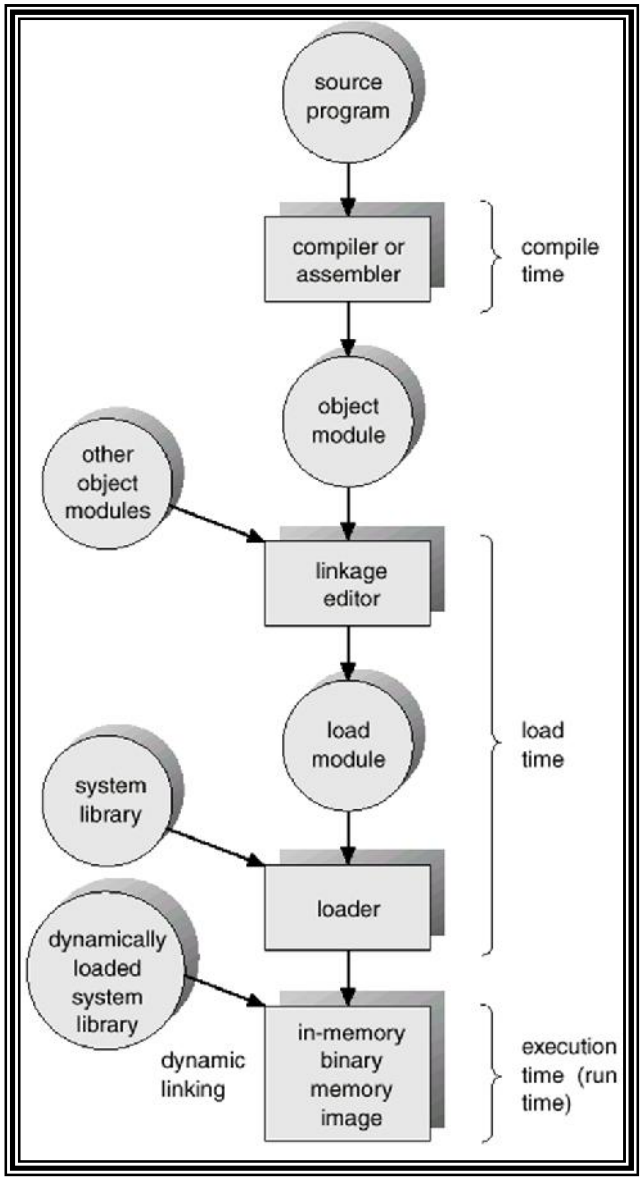
- A loader places the executable code in main memory starting at a pre-determined location (base or start address).

- This can be done in several ways, depending on hardware architecture:
  - Absolute loading: always loads programs into a designated memory location.
  - Relocatable loading: allows loading programs in different memory locations.
  - Dynamic (run-time) loading: loads functions when first called (if ever).

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages.
  - Compile time: If memory location known a priori, absolute code can be generated, other wise relocatable
  - Load time: Compiler must generate relocatable code if memory location is not known at compile time.
  - Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another.
  - Need hardware support for address maps (e.g., base and limit registers).

Multistep Processing of a User Program



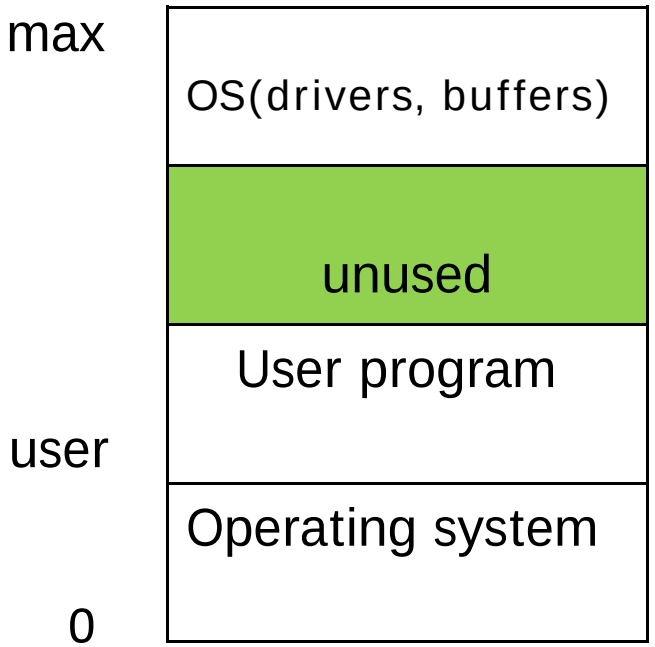
Simple management schemes

- An important task of a memory management system is to bring (load) programs into main memory for execution.

- The following contiguous memory allocation techniques were commonly employed by earlier operating systems :
    - Direct placement
    - Overlays
    - Partitioning

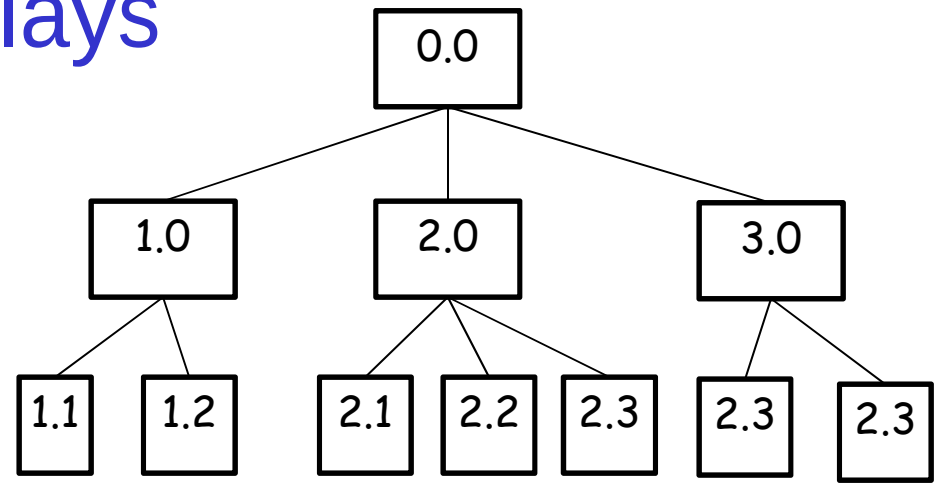
Direct placement

- Memory allocation is trivial.No special relocation is needed,
- because the user programs are always loaded (one at a time) into the same memory location (absolute loading).
- The linker produces the same loading address for every user program.
- Examples: Early batch monitors, MS-DOS



Overlays

- A program was organized (by the user) into a tree-like structure of object modules, called overlays.
- The root overlay was always loaded into the memory,
- whereas the sub trees were (re-loaded as needed by simply overlaying existing code.)

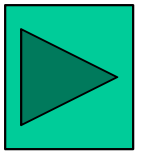


Overlay tree

Memory snapshot

Partitioning

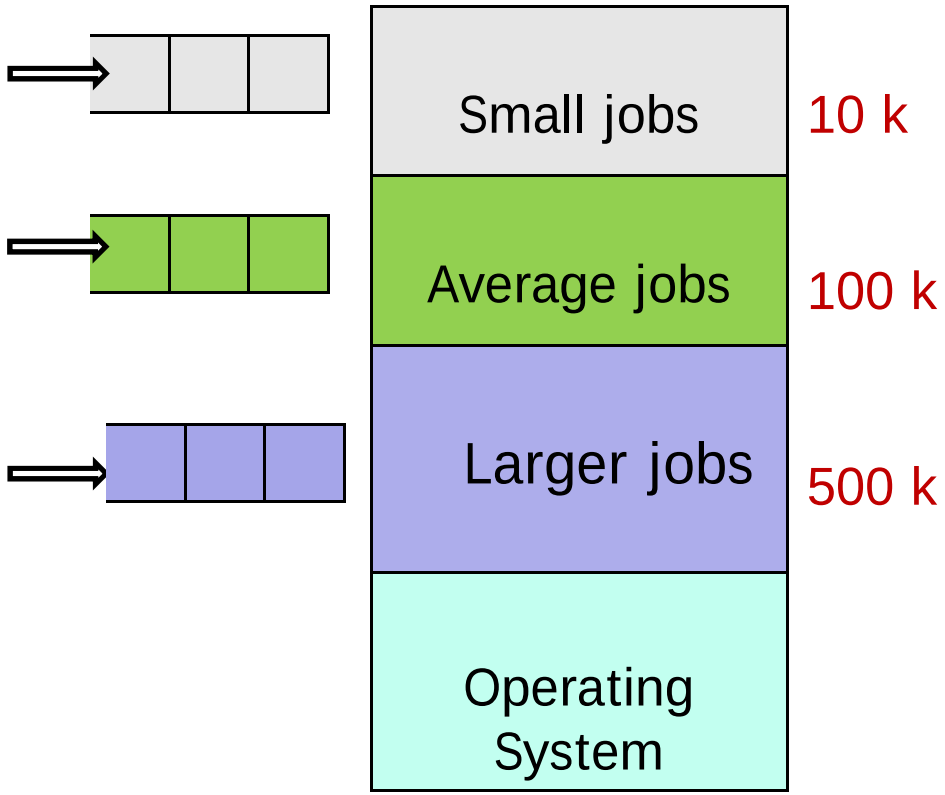
- A simple method to accommodate several programs in memory at the same time (to support multiprogramming) is partitioning.
- In this scheme, the memory is divided into a number of contiguous regions, called partitions.
- Two forms of memory partitioning, depending on when and how partitions are created (and modified), are possible:
  - Static partitioning
  - Dynamic partitioning
- These techniques were used by the IBM OS/360 operating system:
  - MFT (Multiprogramming with Fixed Number of Tasks) and
  - MVT (Multiprogramming with Variable Number of Tasks.)



Partitioning...

Static partitioning

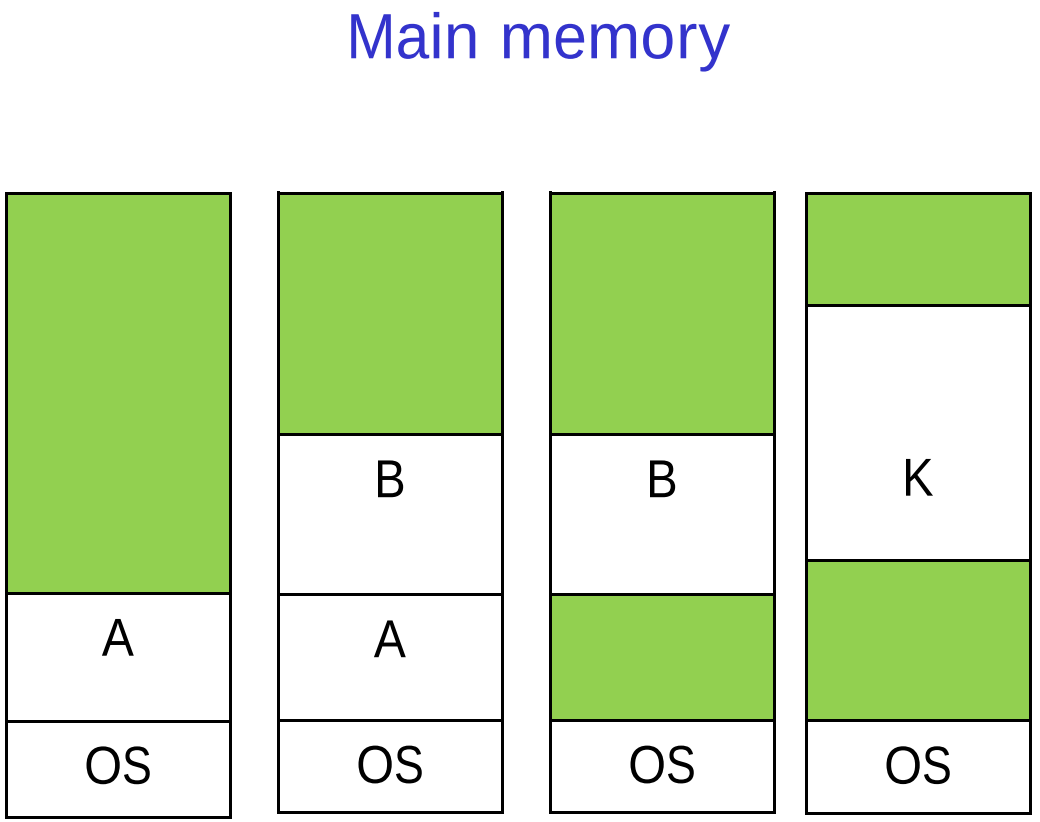
- Main memory is divided into fixed number of (fixed size) partitions during system generation or start-up.
- Programs are queued to run in the smallest available partition.
- An executable prepared to run in one partition may not be able to run in another, without being re-linked.
- This technique uses absolute loading.



Partitioning...

Dynamic partitioning

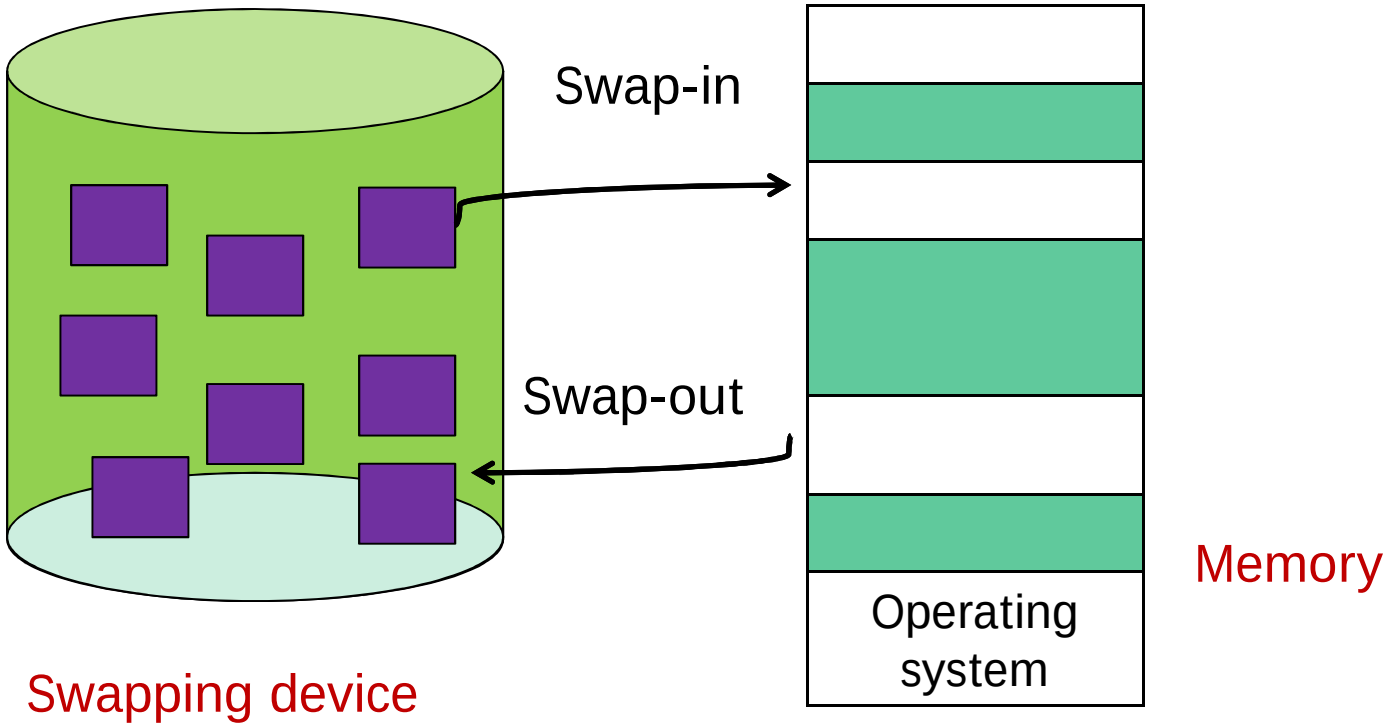
- Any number of programs can be loaded to memory as long as there is room for each.
- When a program is loaded (relocatable loading), it is allocated memory in exact amount as it needs.
- Also, the addresses in the program are fixed after loaded, so it cannot move.
- The operating system keeps track of each partition (their size and locations in the memory.)



Partition allocation at different times

Swapping

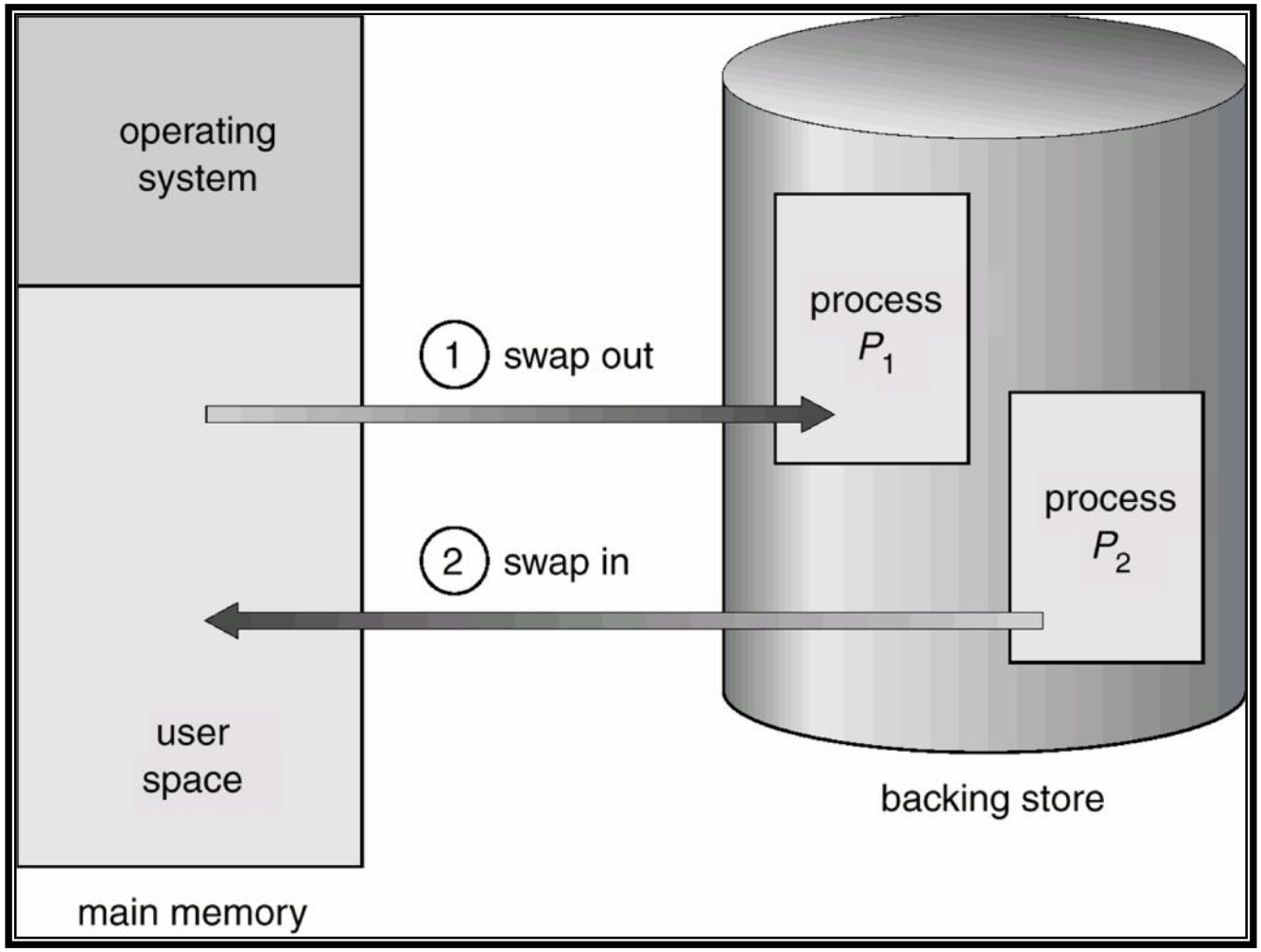
- The basic idea of swapping is to treat main memory as a "pre-emptable" resource.
- A high-speed swapping device is used as the backing storage of the pre-empted processes.



Swapping...

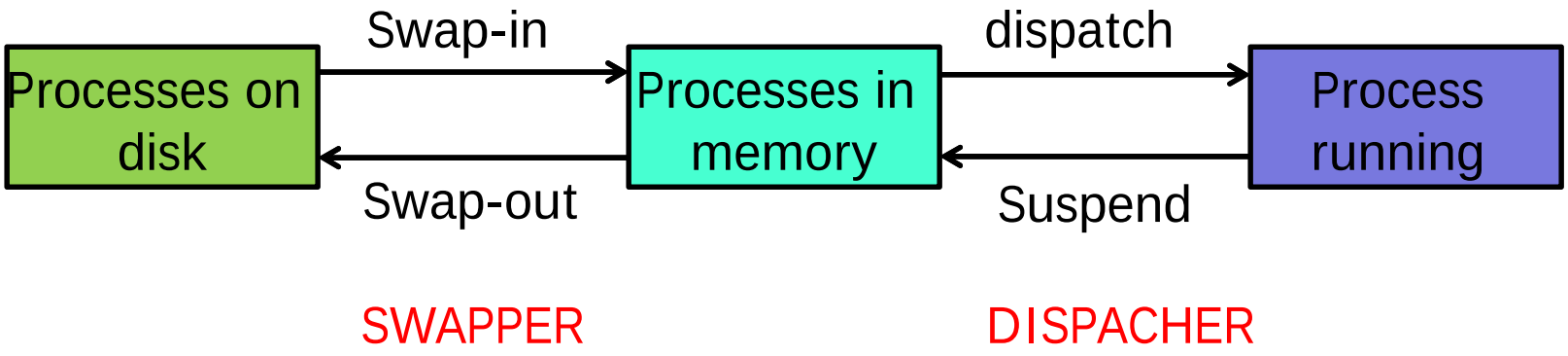
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- Roll out, roll in – swapping variant used for priority-based scheduling algorithms.
  - lower-priority process is swapped out so higher-priority process can be loaded and executed.
- UNIX, Linux, and Windows.

Schematic View of Swapping



Swapping...

- Swapping is a medium term scheduling discipline



- Swapping brings flexibility even to systems with fixed partitions, because:
  - " if needed, the operating system can always make room for high-priority jobs, no matter what!'"

Swapping...

- Swapping is a medium-term scheduling method.
- The responsibilities of a swapper include:
  - Selection of processes to swap out
    - criteria: suspended/blocked state, low priority, time spent in memory
  - Selection of processes to swap in
    - criteria: time spent on swapping device, priority
  - Allocation and management of swap space on a swapping device.
- Swap space can be:
  - system wide
  - dedicated

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.
  - Logical address – generated by the CPU; also referred to as virtual address.
  - Physical address – address seen by the memory unit.

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- logical (virtual) and physical addresses differ in execution-time address-binding scheme.

Memory Protection

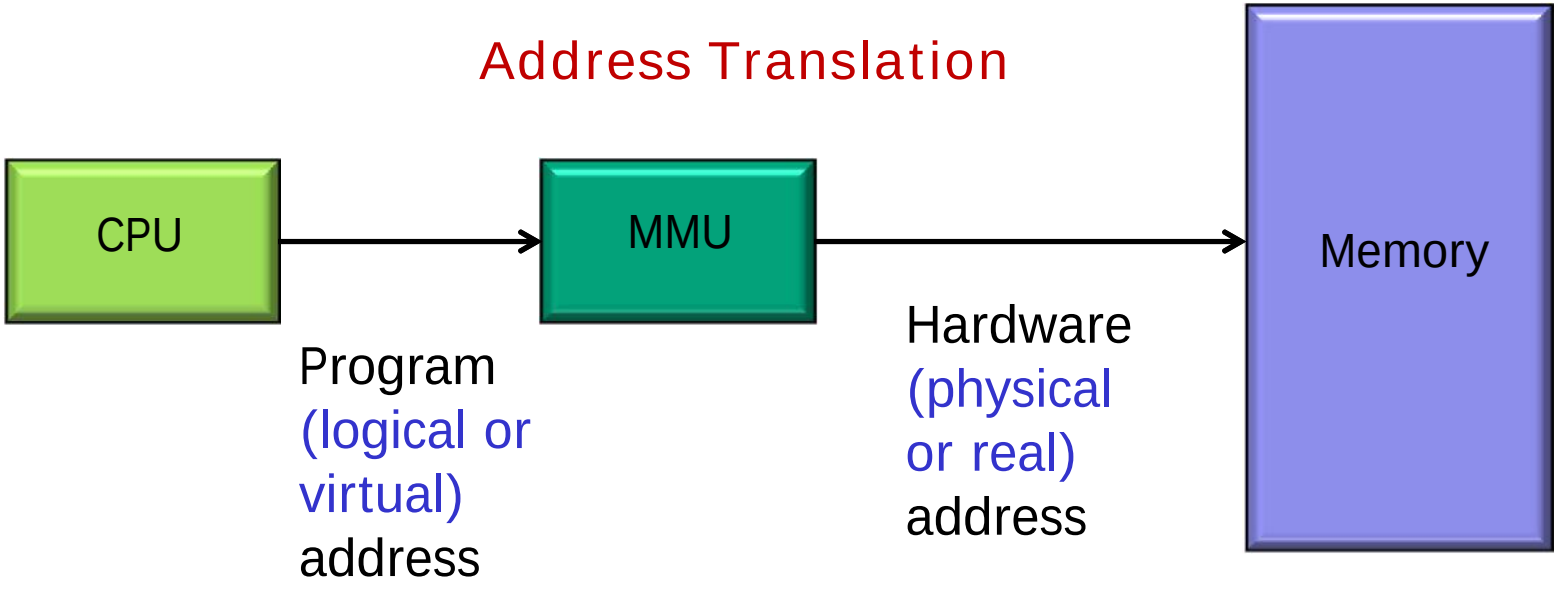
- The other fundamental task of a memory management system is to protect programs sharing the memory from each other.
- This protection also covers the operating system itself.

- Memory protection can be provided at either of the two levels:

- Hardware:
  - ✓ address translation
- Software:
  - ✓ language dependent: strong typing
  - ✓ language independent: software fault isolation

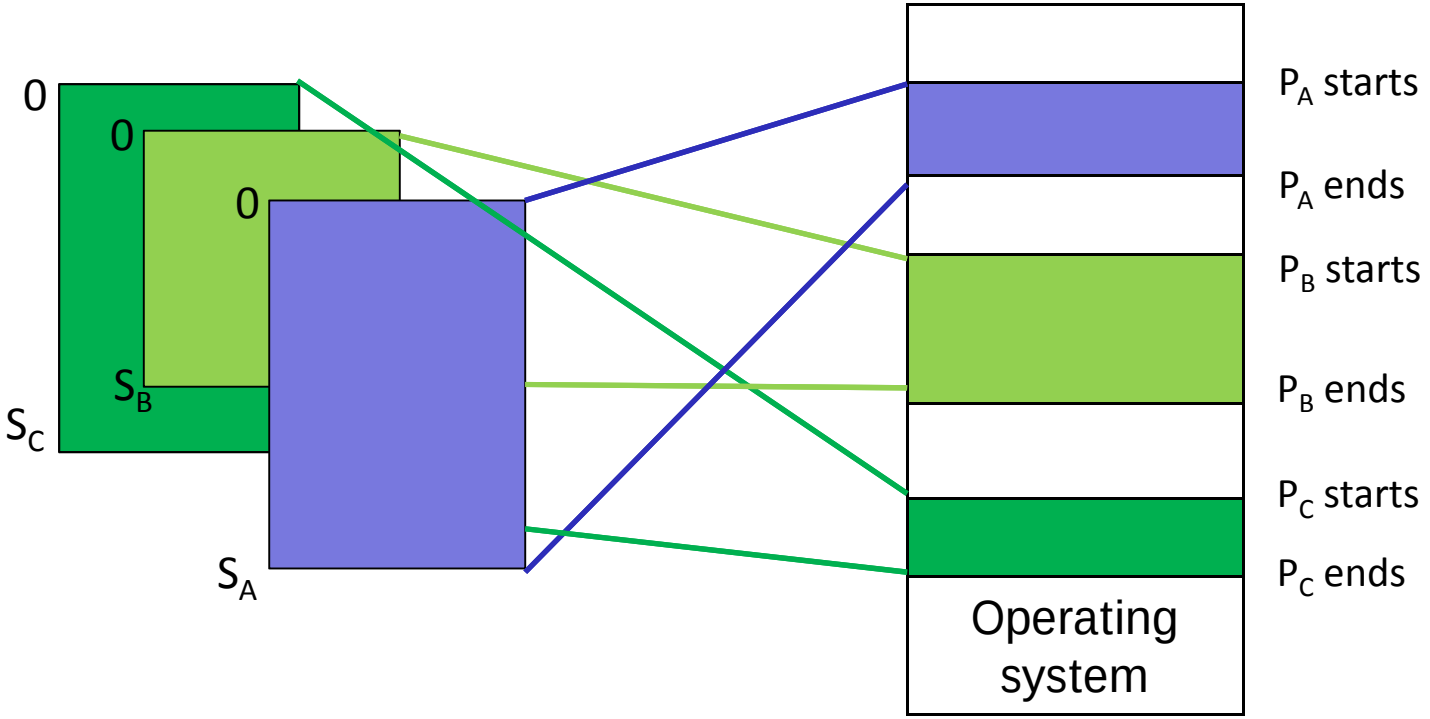
Dynamic relocation

- With dynamic relocation, each program-generated address (logical address) is translated to hardware address (physical address) at runtime for every reference, by a hardware device known as the memory management unit (MMU).



Two views of memory

- Dynamic relocation leads to two different views of main memory, called address spaces.



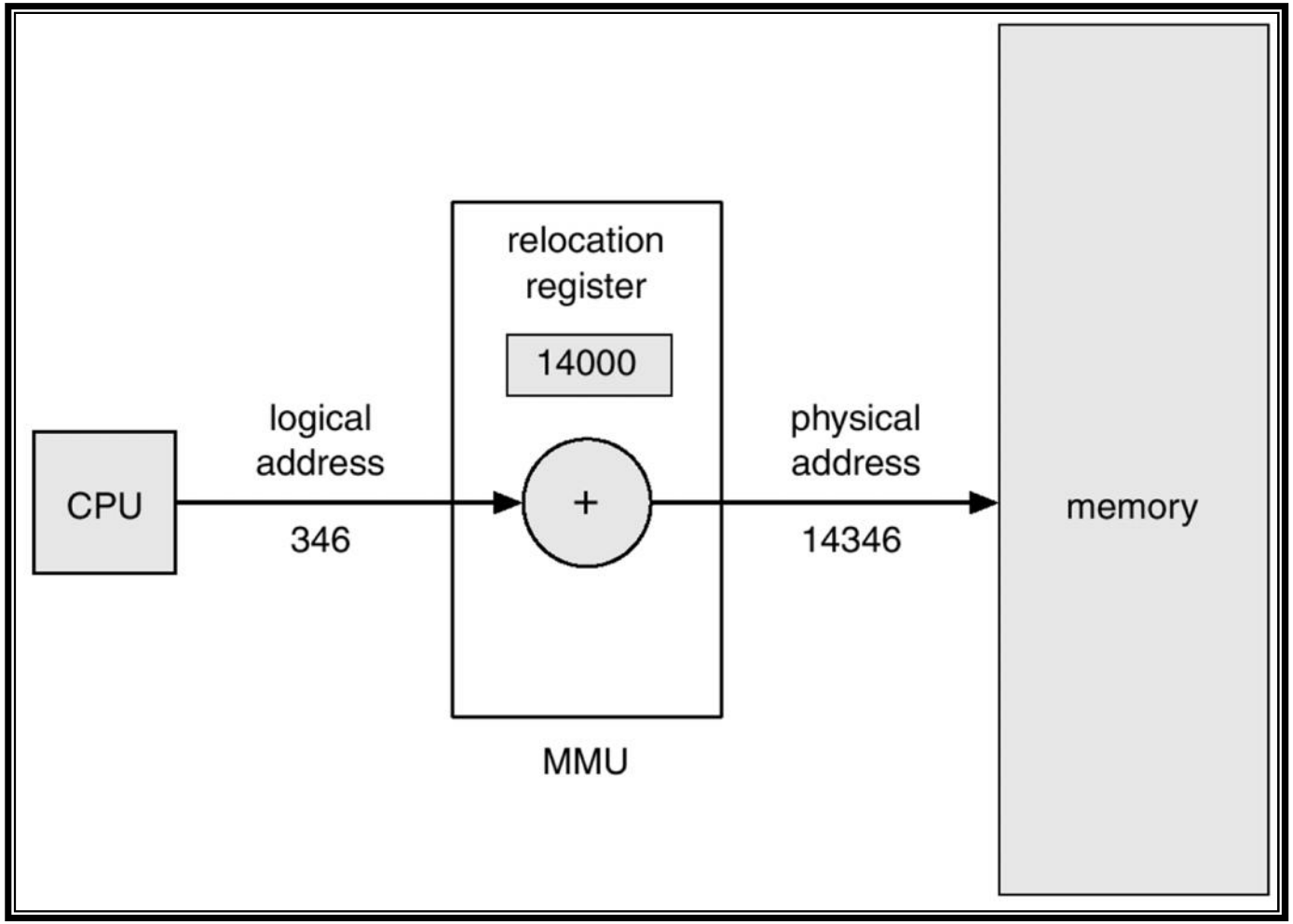
Logical view
(logical address space)

Physical view
(physical address space)

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with logical addresses; it never sees the real physical addresses.

Dynamic relocation using a relocation/base register

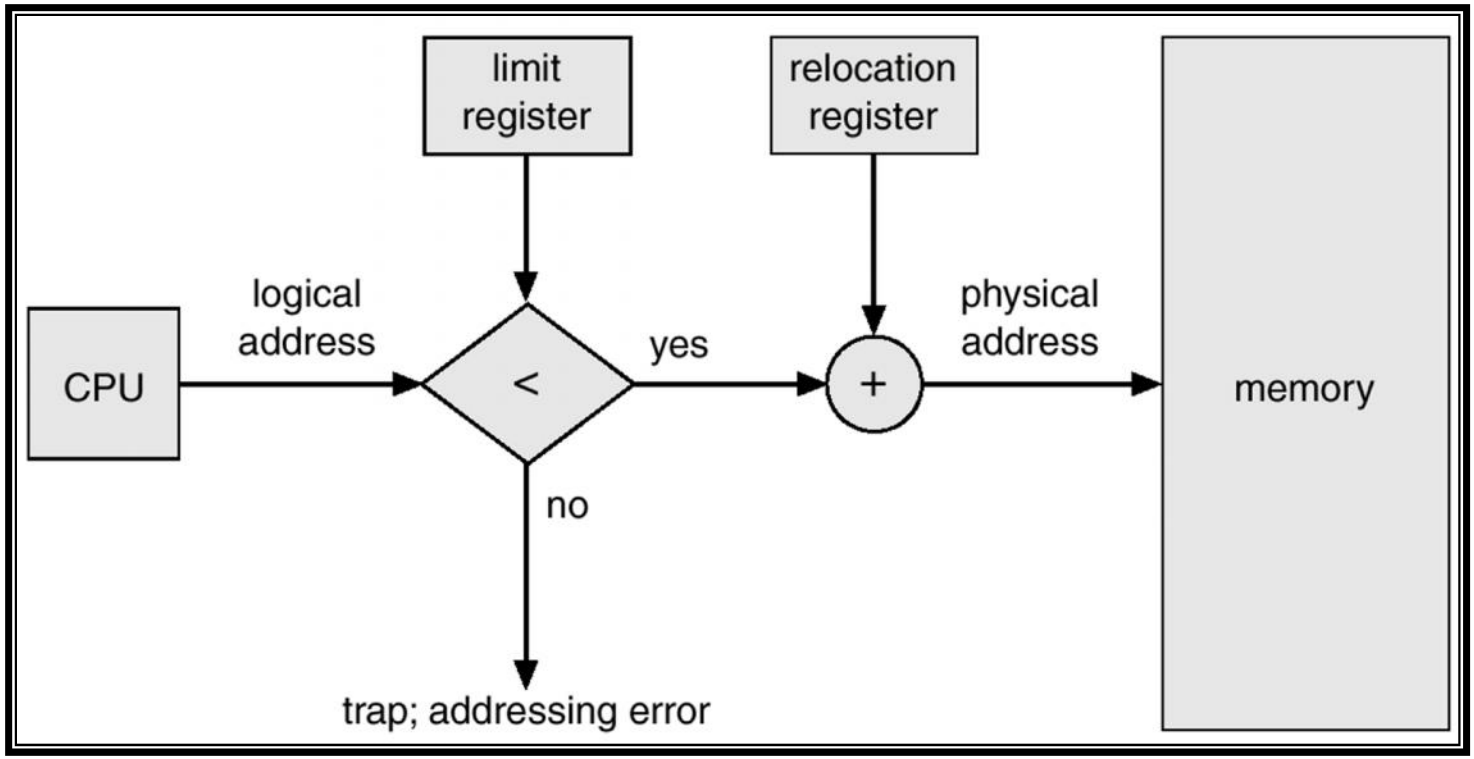


Contiguous Allocation

- Main memory is divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.

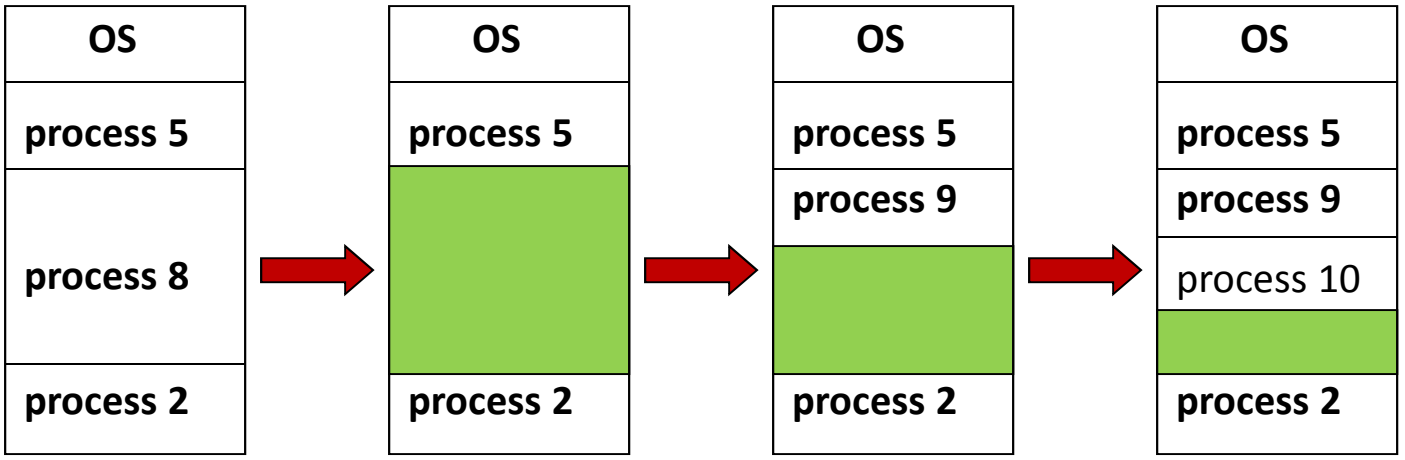
- Single-partition allocation
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
  - Relocation register contains value of smallest physical address;
  - Limit register contains range of logical addresses – each logical address must be less than the limit register.

Hardware Support for Relocation and Limit Registers for memory protection discussed before



Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory.
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Main Memory

Dynamic Storage-Allocation Problem

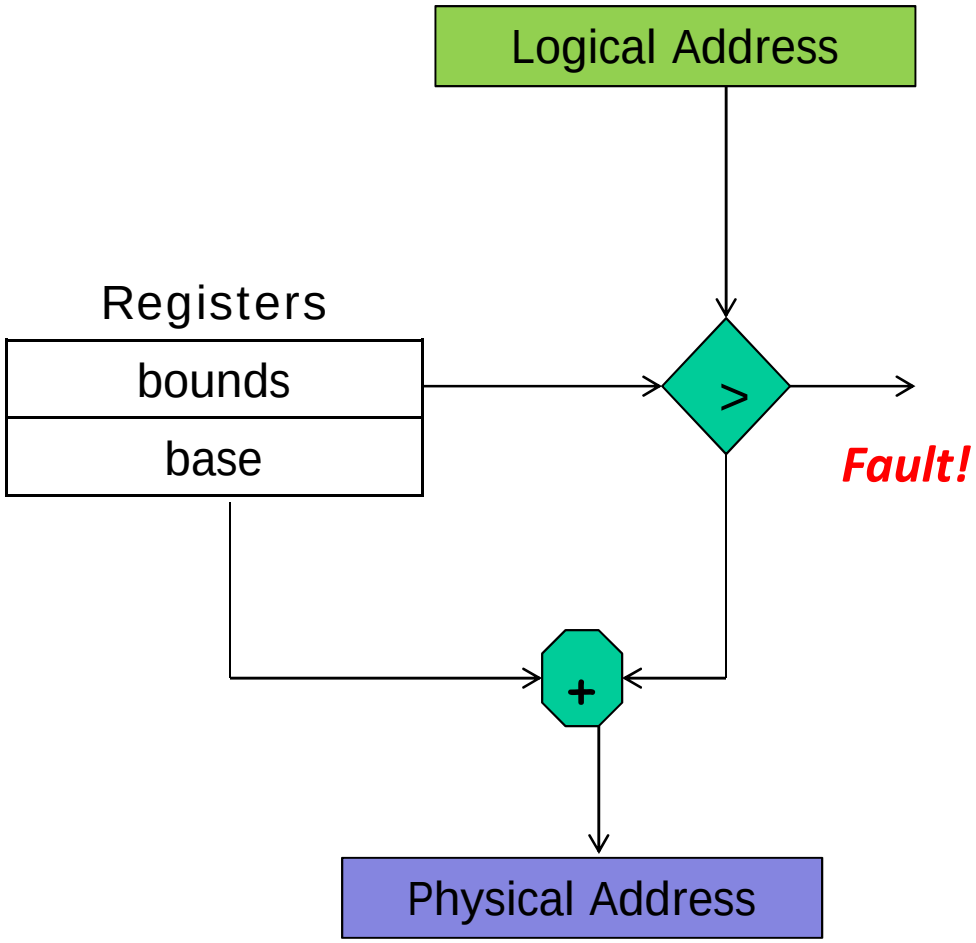
- How to satisfy a request of size n from a list of free holes.
  - First-fit: Allocate the first hole that is big enough.
  - Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
  - Worst-fit: Allocate the largest hole; must also search entire list. Produces the largest leftover hole.
  - First-fit and best-fit better than worst-fit in terms of speed and storage utilization.

Fragmentation

- Fragmentation refers to the unused memory that the management system cannot allocate.
- Internal fragmentation
  - Waste of memory within a partition, caused by the difference between the size of a partition and the process loaded.
  - Severe in static (fixed) partitioning schemes.
- External fragmentation
  - Waste of memory between partitions, caused by scattered non-contiguous free space.
  - Severe in dynamic (variable size) partitioning schemes.
  - Compaction is a technique that is used to overcome external fragmentation.

Gentle reminder: Base and bounds relocation

- Each program is loaded into a contiguous region of memory.
- This region appears to be ‘ ‘private’ ’ and the bounds register limits the range of the logical address of each program.
- Hardware implementation is cheap and efficient:
  - 2 registers plus an adder and a comparator.



Segmentation

- The most important problem with base-and-bounds relocation is that there is only one segment for each process.
- Segmentation is a memory management scheme that supports user's view of memory.
- A segment is a region of contiguous memory.
- Segmentation generalizes the base-and-bounds technique by allowing each process to be split over several segments.
  - A segment table holds the base and bounds of each segment.
  - Although the segments may be scattered in memory, each segment is mapped to a contiguous region.
  - Additional fields (Read/Write and Shared) in the segment table adds protection and sharing capabilities to segments.

Segmentation...

- A program is a collection of segments.
- A segment is a logical unit such as:

  main program,
  procedure,
  function,
  method,
  object,
  local variables, global variables,
  common block,
  stack,
  symbol table, arrays

Segmentation Architecture

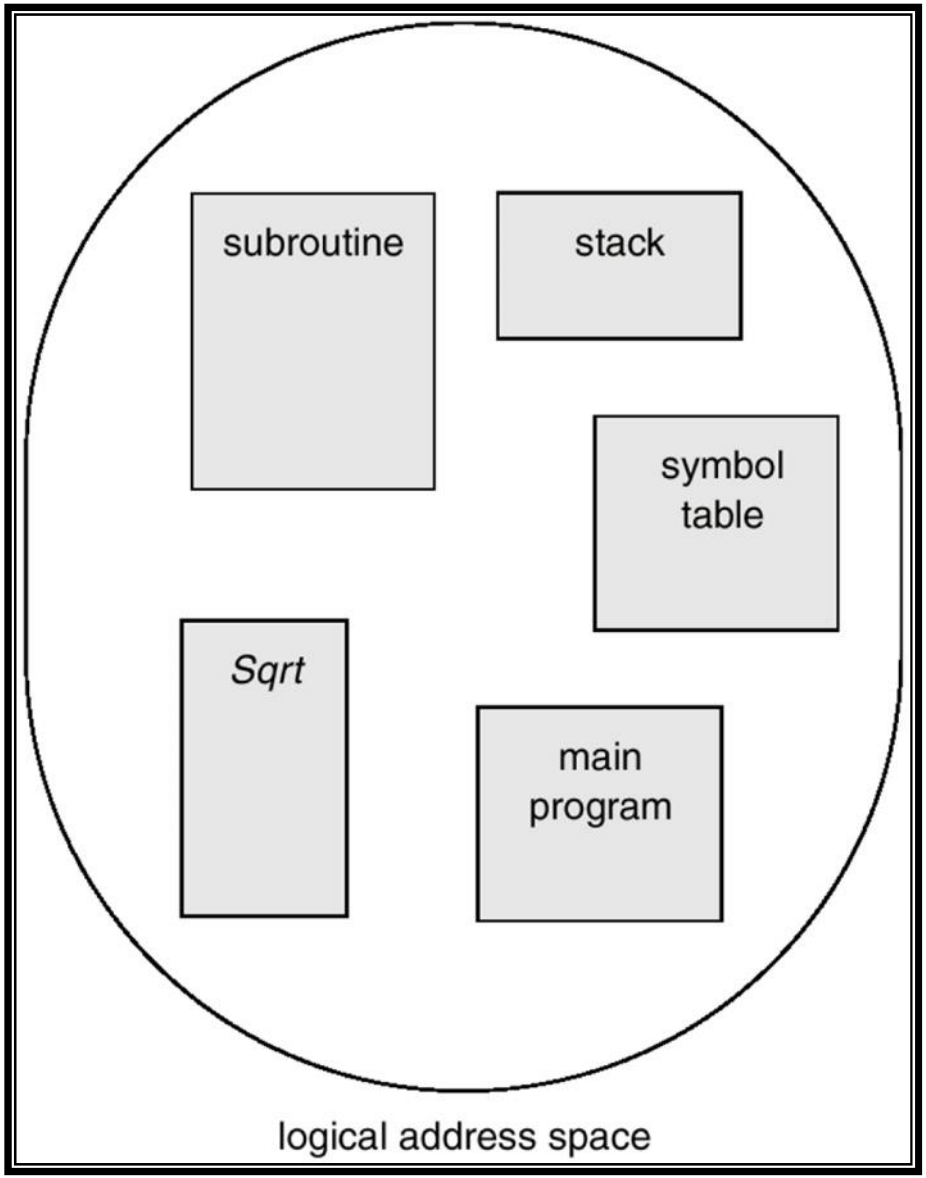
- Logical address consists of a two tuple:

  <segment-number, offset>,

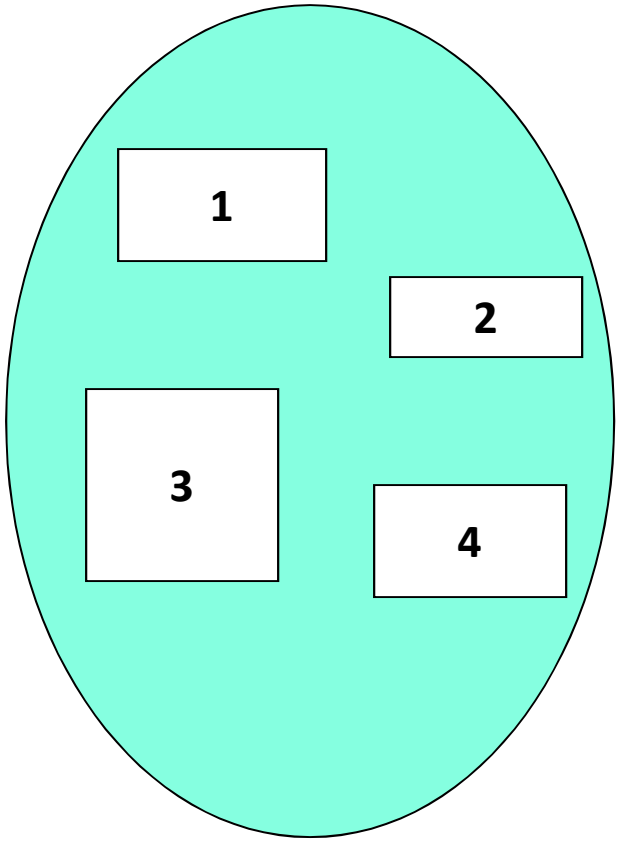
- Segment table – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory.
  - limit – specifies the length of the segment.
- Segment-table base register (STBR) points to the segment table's location in memory.
- Segment-table length register (STLR) indicates number of segments used by a program;

  segment number s is legal if s < STLR.

User's View of a Program



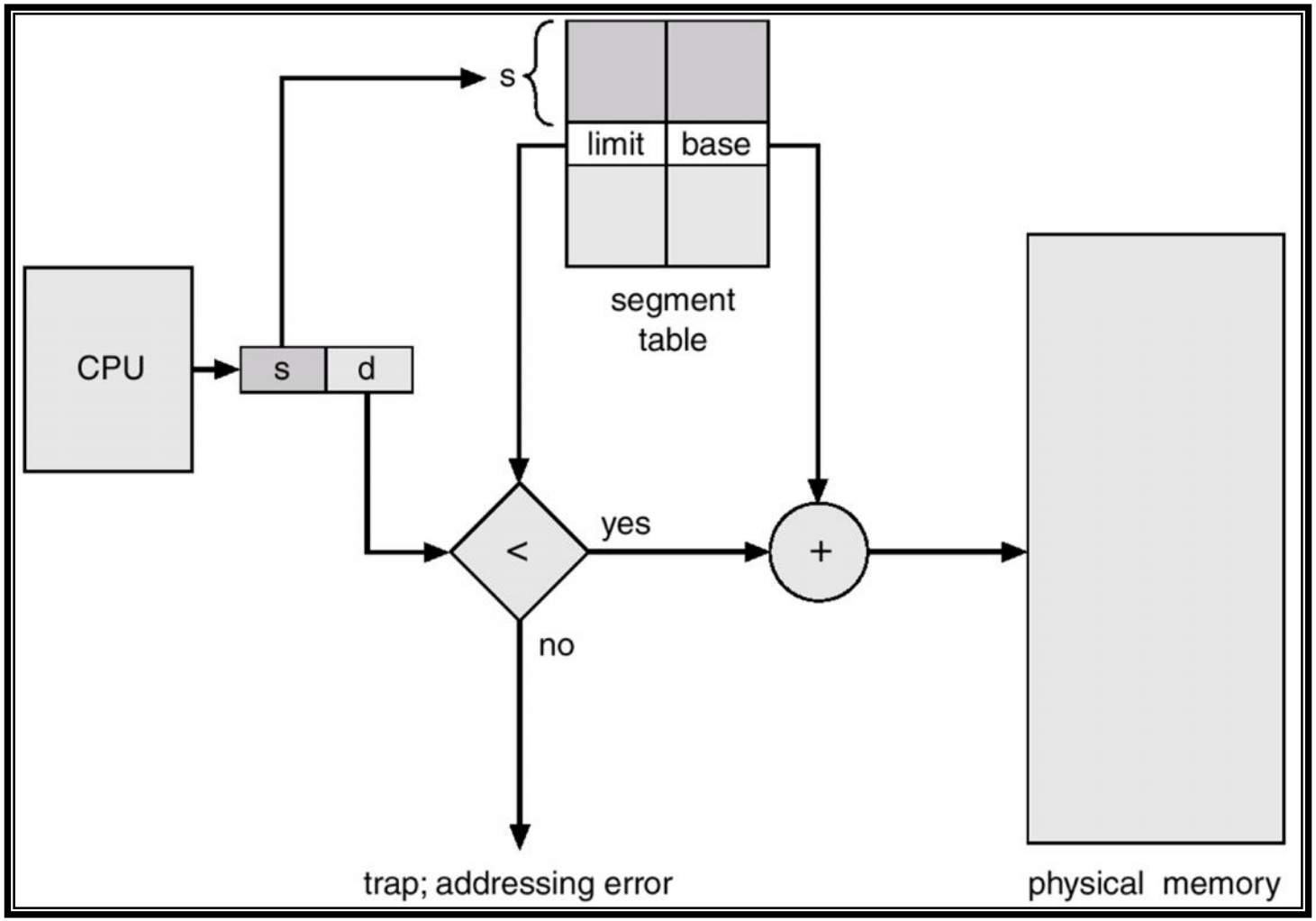
Logical View of Segmentation



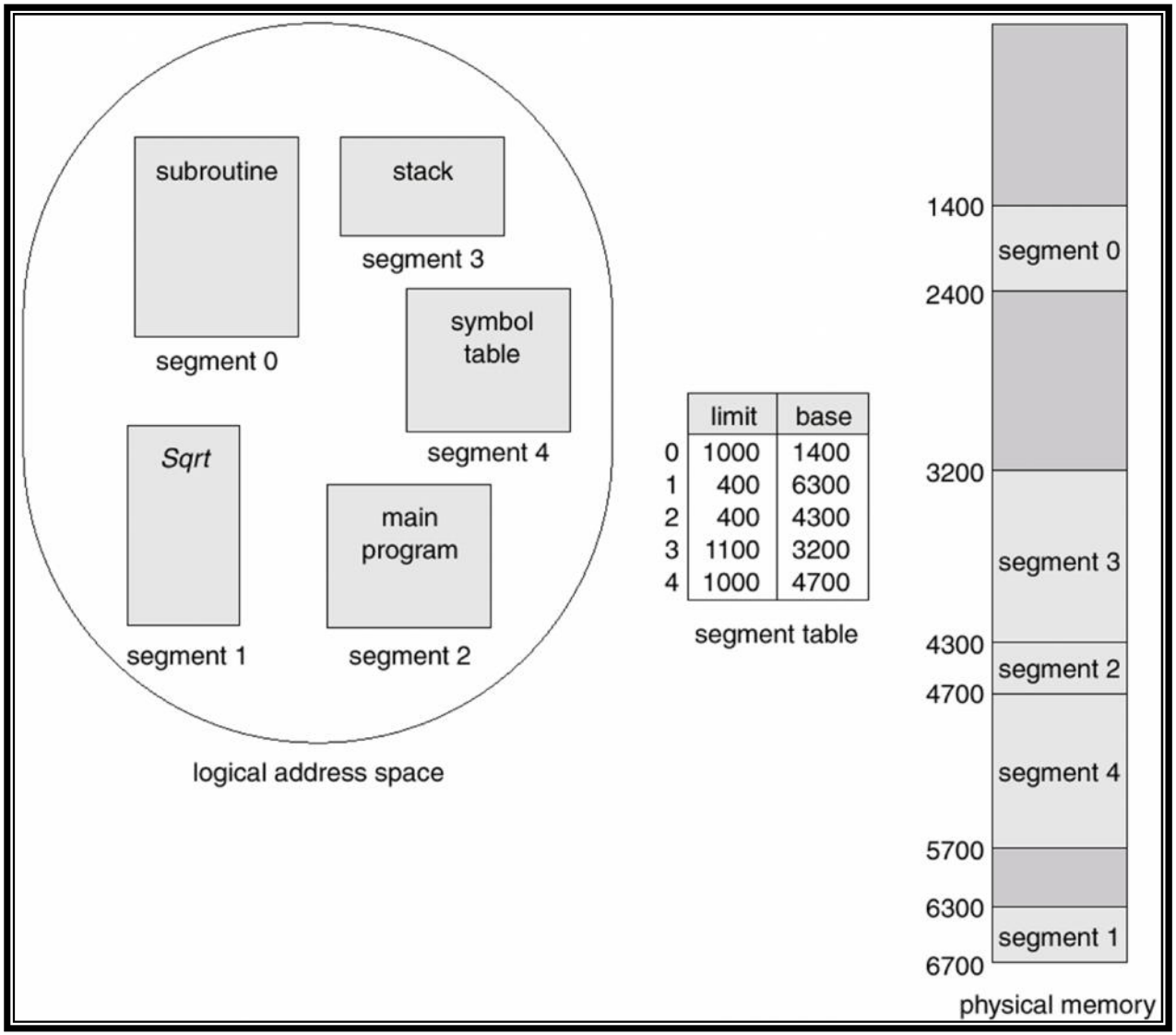
user space

physical memory space

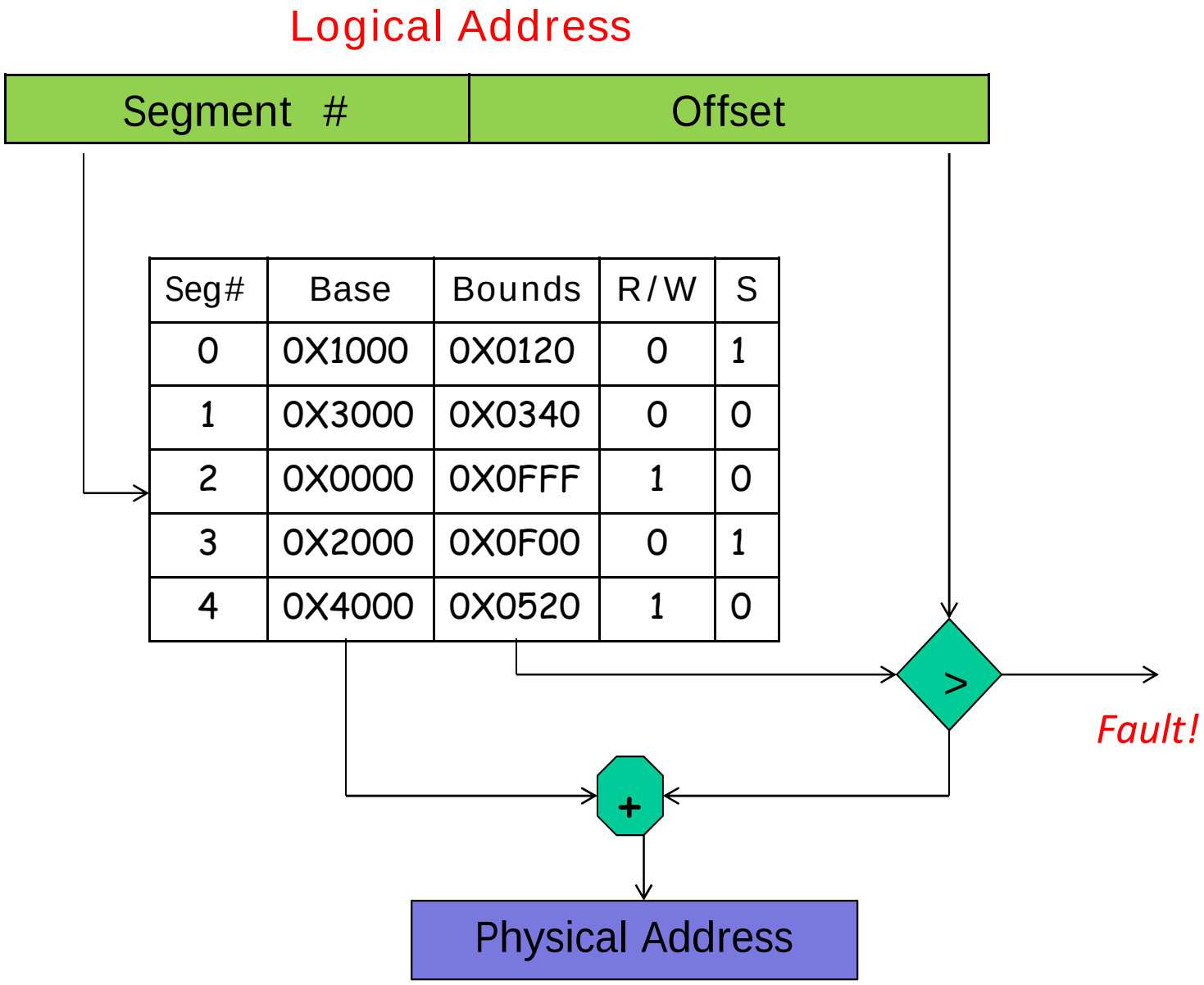
Segmentation Hardware



Example of Segmentation



Relocation with segmentation



Segmentation...

- When a process is created, a pointer to an empty segment table is inserted into the process control block.
- Table entries are filled as new segments are allocated for the process.
- The segments are returned to the free segment pool when the process terminates.
- Segmentation, as well as the base and bounds approach, causes external fragmentation and requires memory compaction.
- An advantage of the approach is that only a segment, instead of a whole process, may be swapped to make room for the (new) process.

Paging

- Paging is a memory-management scheme that permits the physical address space of a process to be noncontigious.
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 16 mbytes per page).
- Divide logical memory into blocks of same size called pages.
- Keep track of all free frames.
- To run a program of size n pages, need to find n free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Problem: Internal fragmentation.

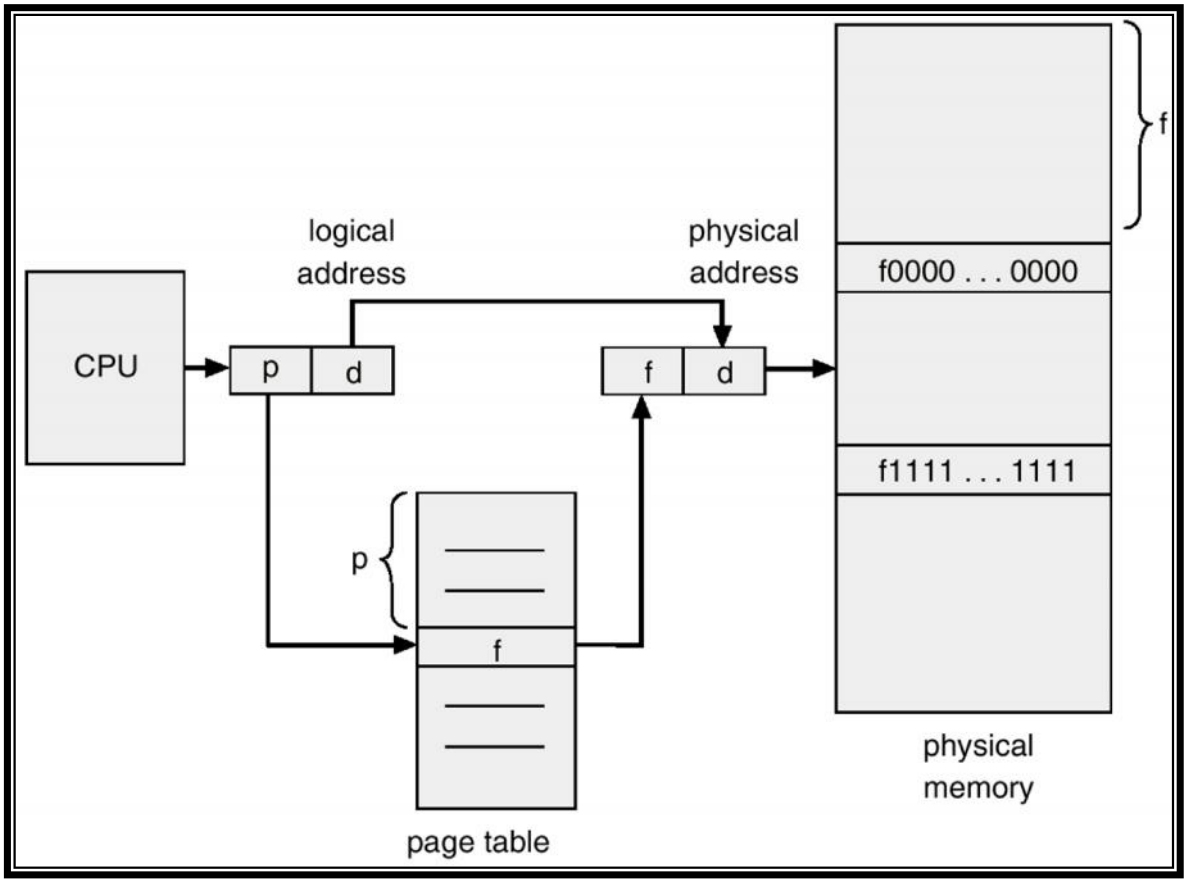
Paging...

- Physical memory is divided into a number of fixed size blocks, called frames.
- The logical memory is also divided into chunks of the same size, called pages.
- The size of frame/page is determined by the hardware and typically is some value between 512 bytes (VAX) and 16 megabytes (MIPS 10000)!
- A page table maps the base address of pages for each frame in the main memory.
- The major goals of paging are:
  - to make memory allocation and swapping easier and
  - to reduce fragmentation.
  - Paging also allows allocation of non-contiguous memory (i.e., pages need not be adjacent.)

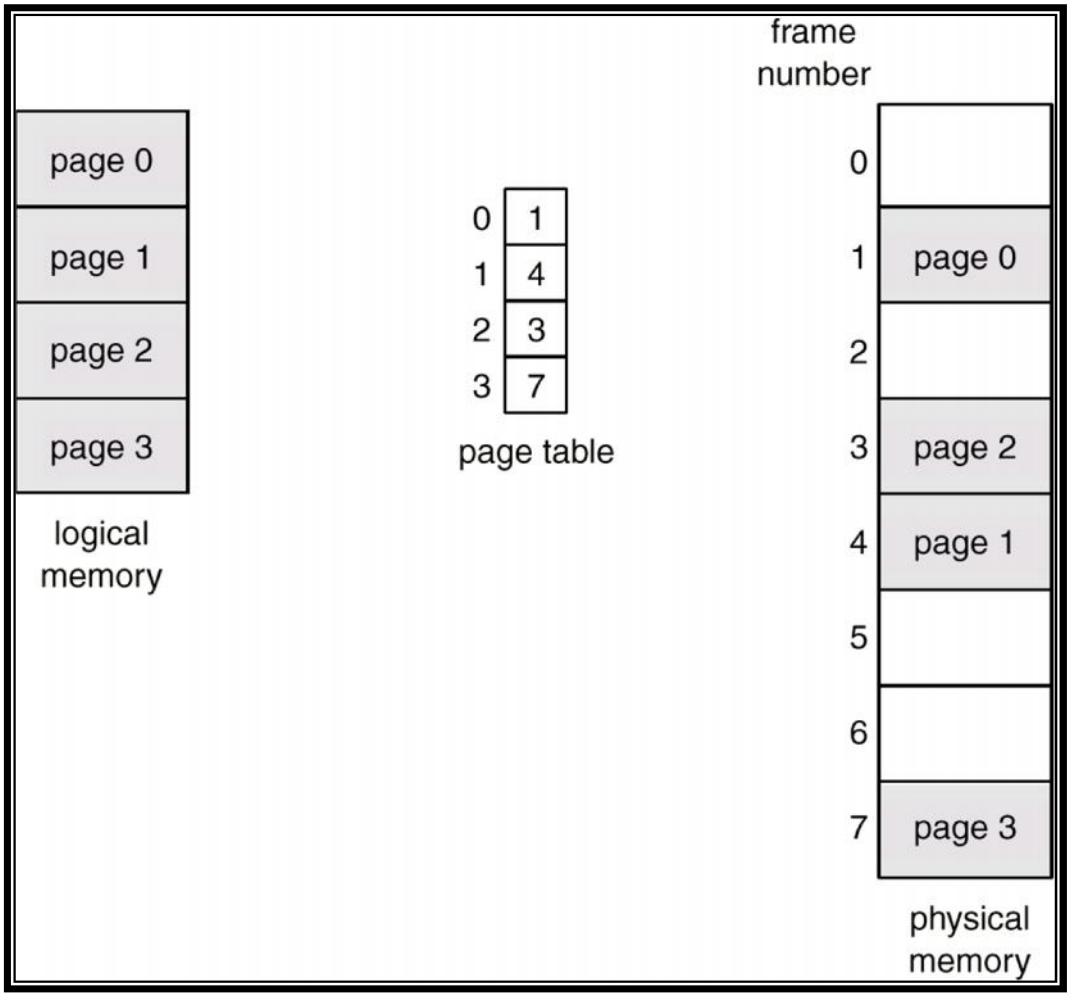
Address Translation Scheme

- Address generated by CPU is divided into:
  - Page number (p) – used as an index into a page table which contains base address of each page in physical memory.
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit.

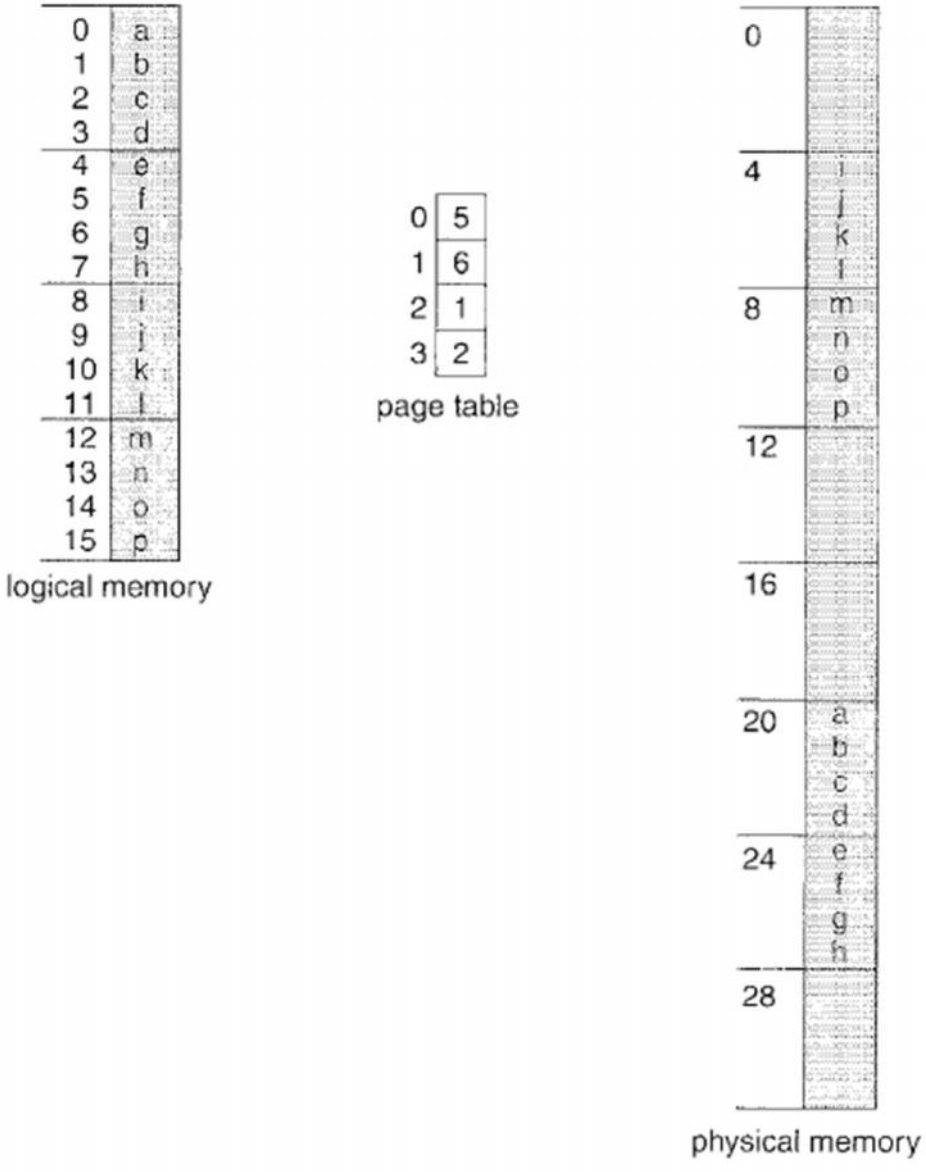
Address Translation Architecture (HW) using paging



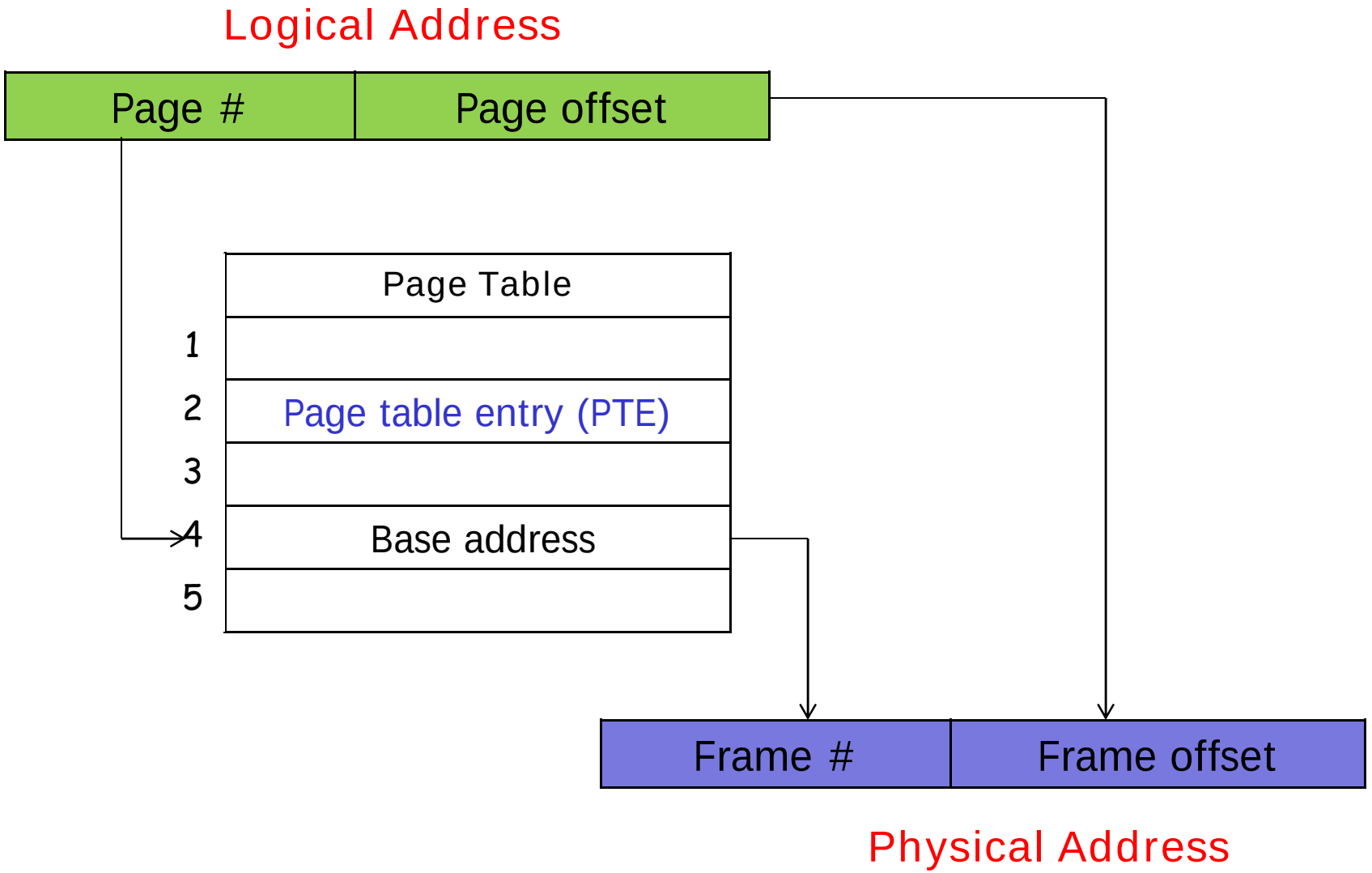
Paging Example



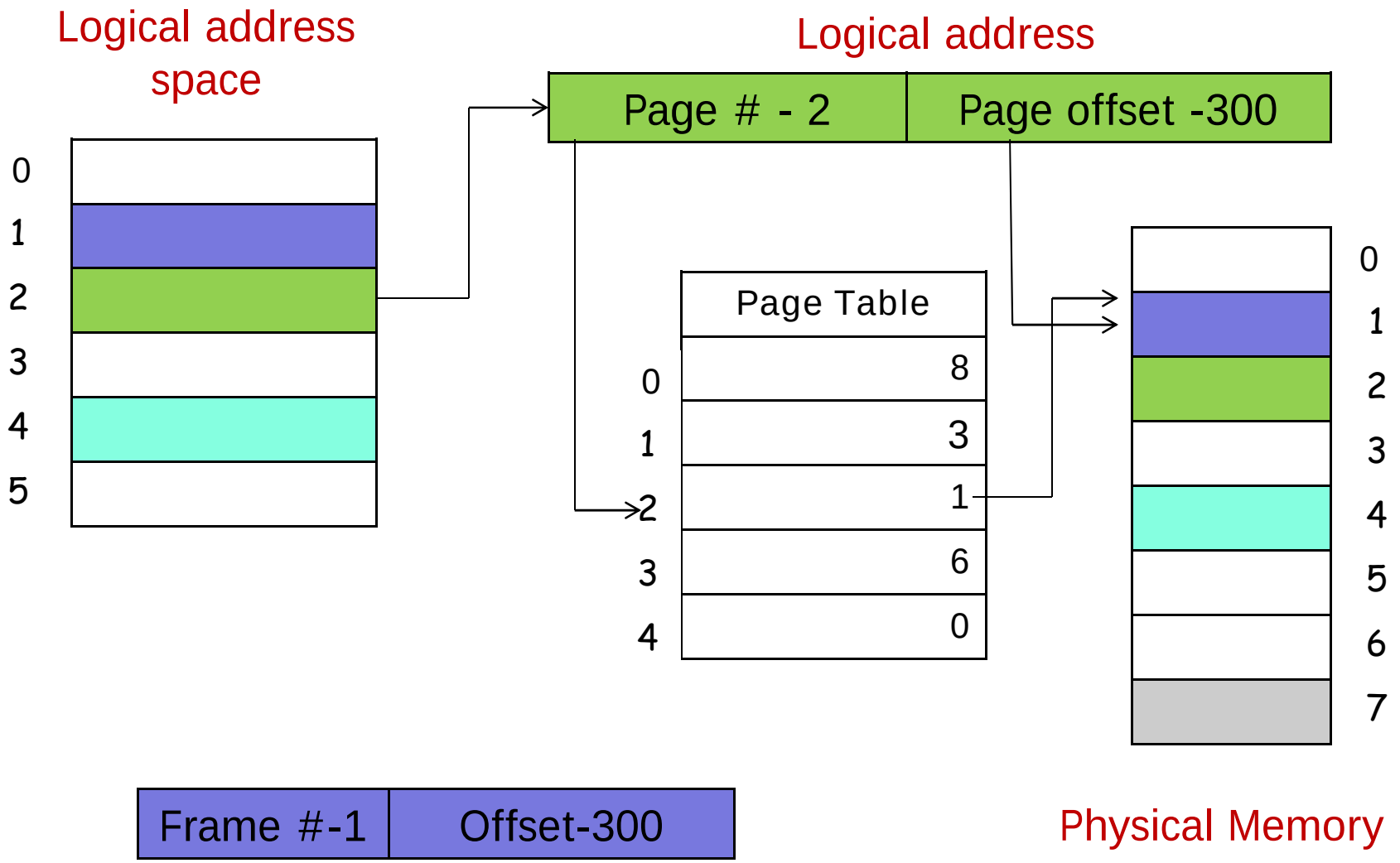
Paging example for a 32-byte memory with four bytes pages



Relocation with paging



Paging - an example

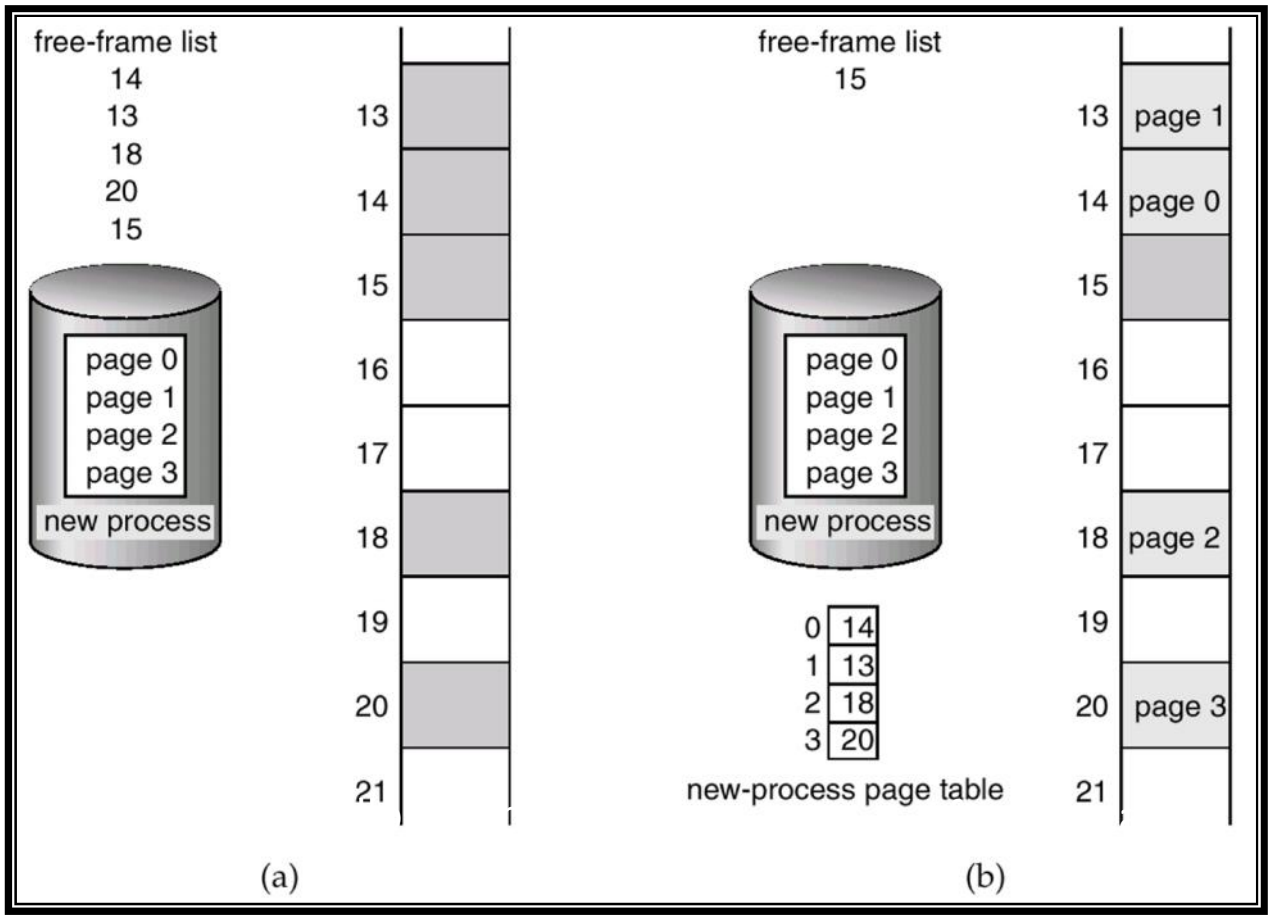


Paging...

- Paging itself is a form of dynamic relocation.
- Every logical address is bound by the paging HW to some physical address.
- Using paging is similar to using a table of base (relocation) registers, one for each frame of memory.

- When using paging scheme, there is no external fragmentation.
- Any free frame can be allocated to a process that needs it.
- However, we may have some internal fragmentation.

Free Frames


free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

Implementation of Page Table

- Page table is kept in main memory.
- Page-table base register (PTBR) points to the page table.
- Page-table length register (PRLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses.
  - One for the page table and
  - one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)

Associative memory

- Problem: Both paging and segmentation schemes introduce extra memory references to access translation tables.
- Solution? Translation buffers.
- Based on the notion of locality (at a given time a Process is only using a few pages or segments), a very fast but small associative (content addressable) memory is used to store a few of the translation table entries.
- This memory is known as a translation look-aside buffer or TLB

Associative Memory...

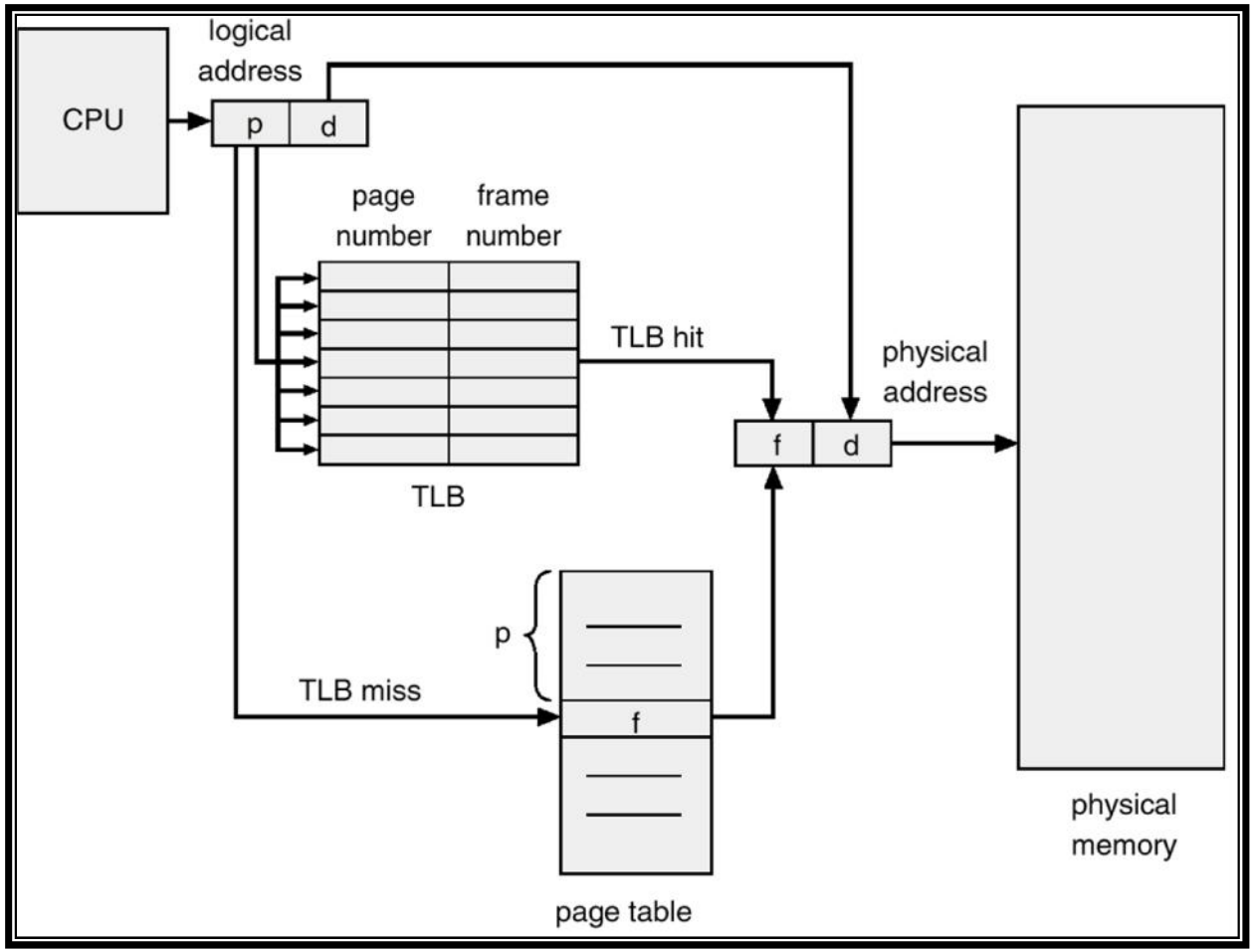
- Associative memory – parallel search

Page #	Frame #

Address translation (A´, A´´)

- If A´ is in associative register, get frame # out.
- Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers.
- Hit ratio = α
- Effective Access Time (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

Effective Access Time...

- The percentage of times that a particular page number is found is called hit ratio.

Assumption:

- 80 % hit ratio
- If it takes 20 nsec to search TLB, 100 nsec to access MM
  - Then mapped-memory access takes 120 nsec when the page number is in TLB.
- If we fail to find the page number in TLB (20 nsec),
  - Then we must first access memory for the page table and frame number (100 nsec) then access the desired byte in memory (100 nsec), for a total of 220 nsec.
- To find the EAT:

$$EAT = 0.80 \times 120 + 0.20 \times 220$$

$$= 140\text{nsec}$$

Effective Access Time...

- In this example, we suffer a 40% slow down in memory-access time (from 100 - 140 nsec).

- For a 98% hit ratio, we have

EAT = 0.98 X 120 + 0.02 X 220

= 122 nsec

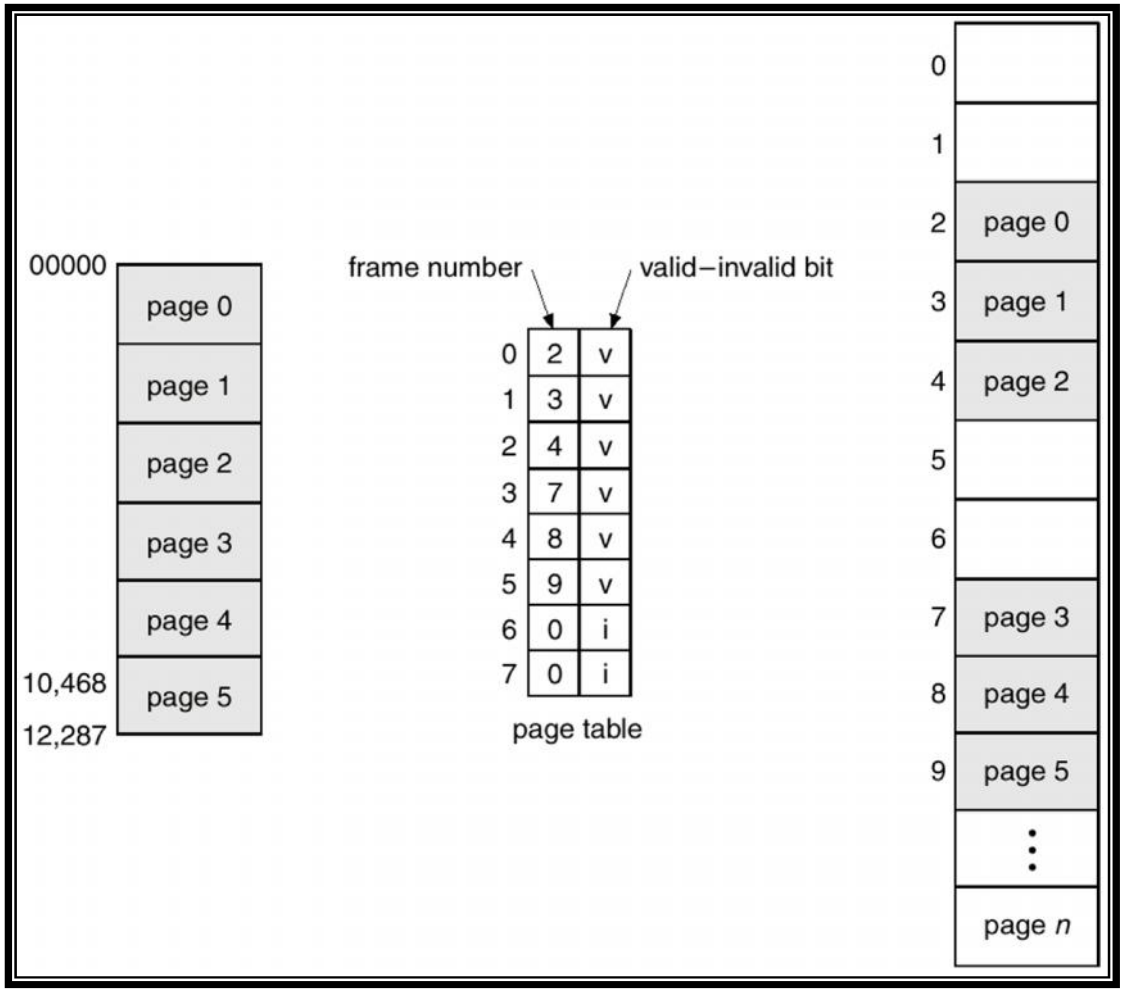
- This increased hit rate produces only 22 percent slow down in access time.

Memory Protection

- Memory protection in a paged environment implemented by associating protection bit with each frame.
- These bit are kept in page table.
- One bit for read-only, read-write, execute-only.

- Valid-invalid bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page.
  - "invalid" indicates that the page is not in the process' logical address space.

Valid (v) or Invalid (i) Bit In A Page Table



Hardware support for paging

- There are different hardware implementations of page tables to support paging.
  - A set of dedicated registers, holding base addresses of frames.
  - In memory page table with a page table base register (PTBR).
  - Same as above with multi-level page tables.

Page Table Structure

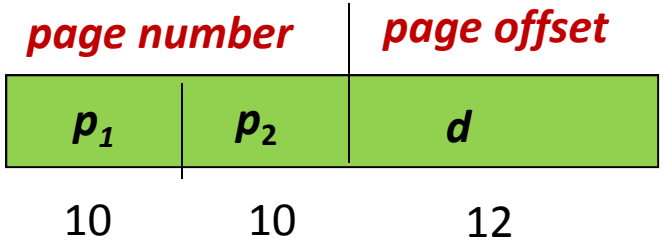
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Most modern computer systems support a large logical address space (2^{32} -2^{64}).
- The page table itself becomes excessive.
- For example: system with 32-bit logical address space.
- If page size is 4KB (2^{12}), then page table = 2^{32} / 2^{12} = up to 1 million entries.
- Assume that each entry = 4 bytes, each process may need up to 4 MB of physical address space for the page table alone.
- Solution
  - Break up the logical address space into multiple page tables.
  - A simple technique is a two-level page table.

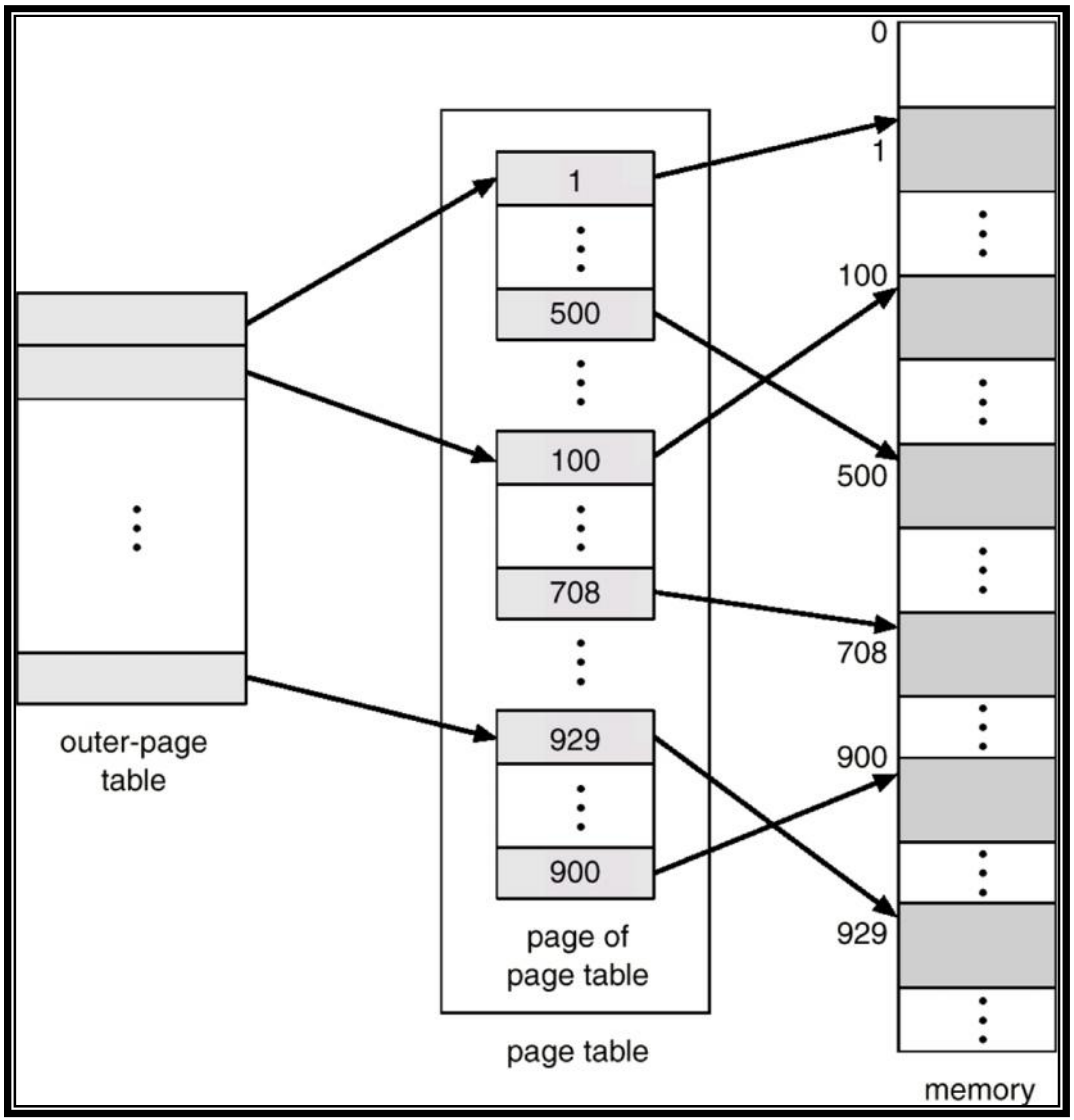
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- Thus, a logical address is as follows:



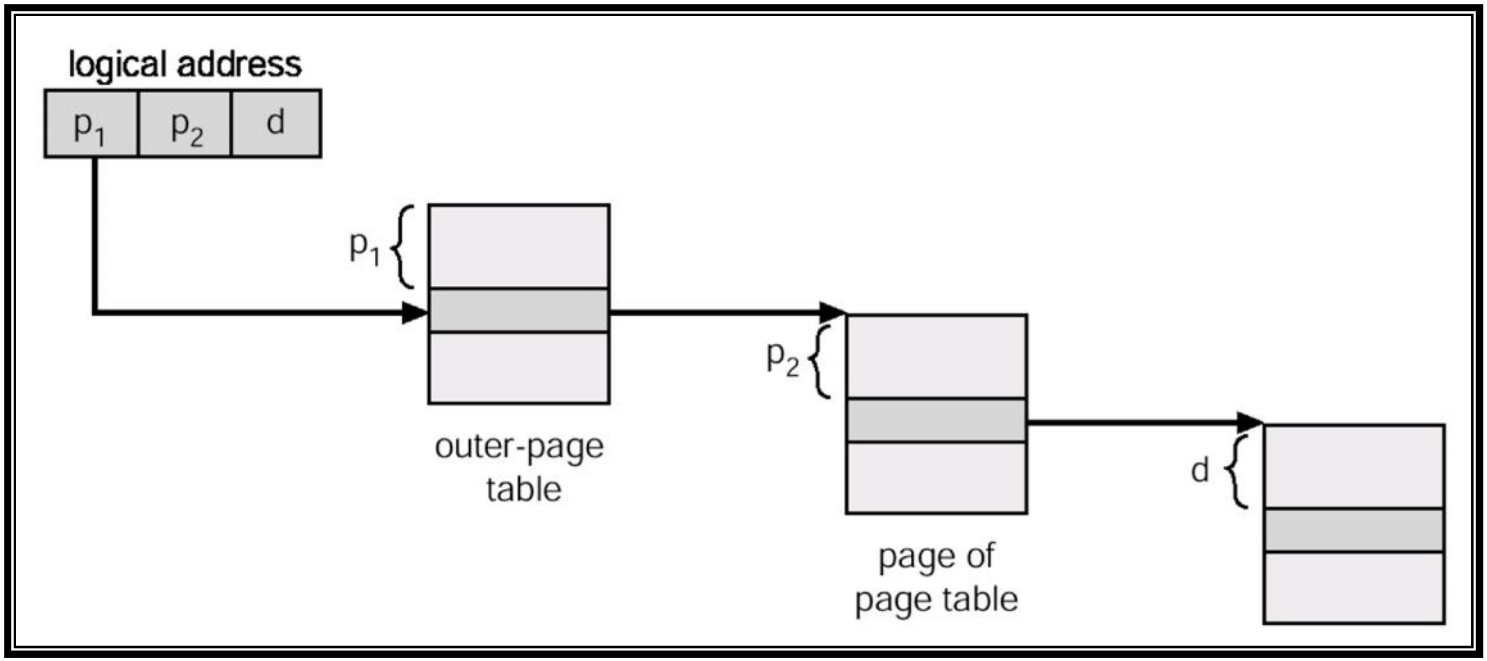
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

Two-Level Page-Table Scheme

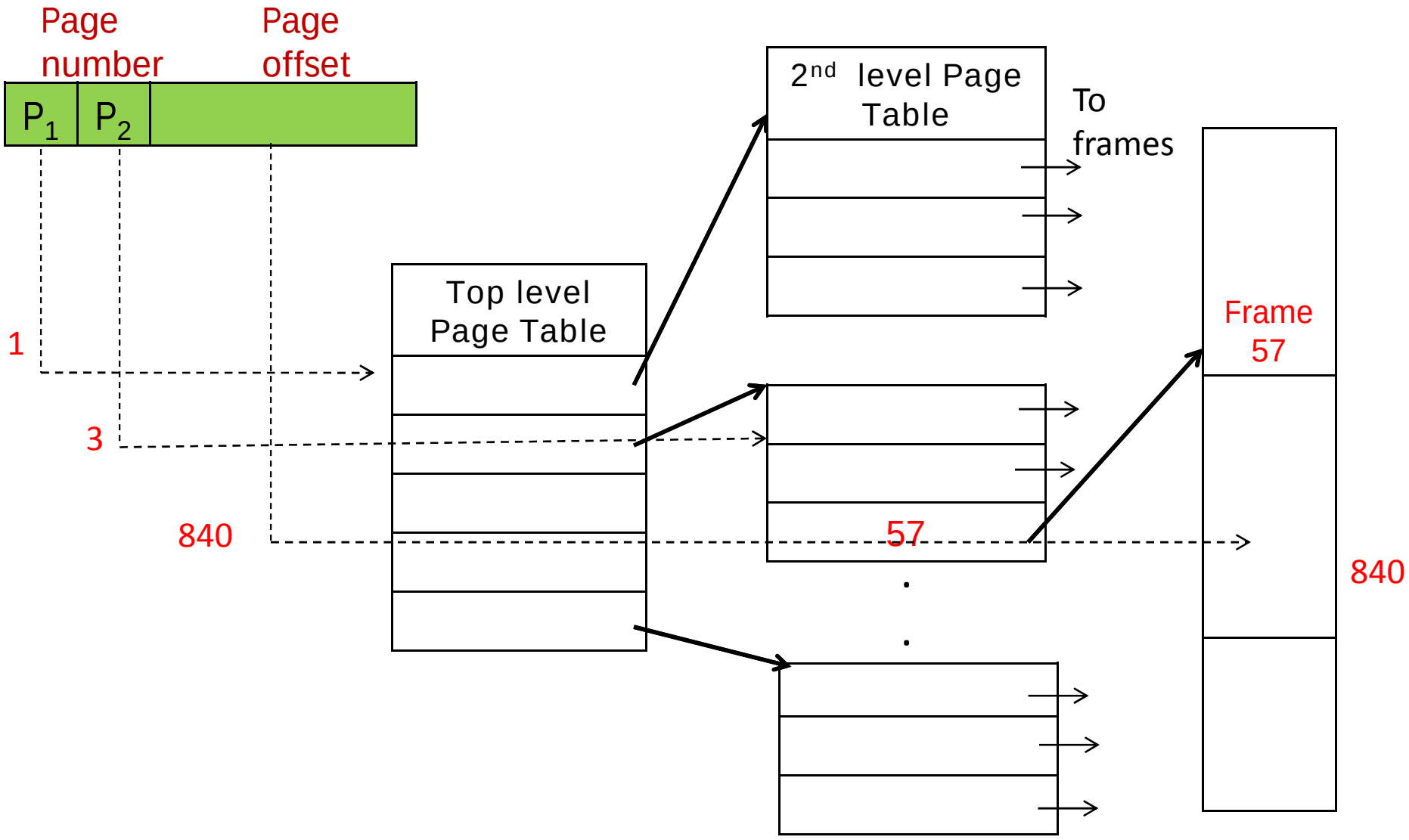


Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

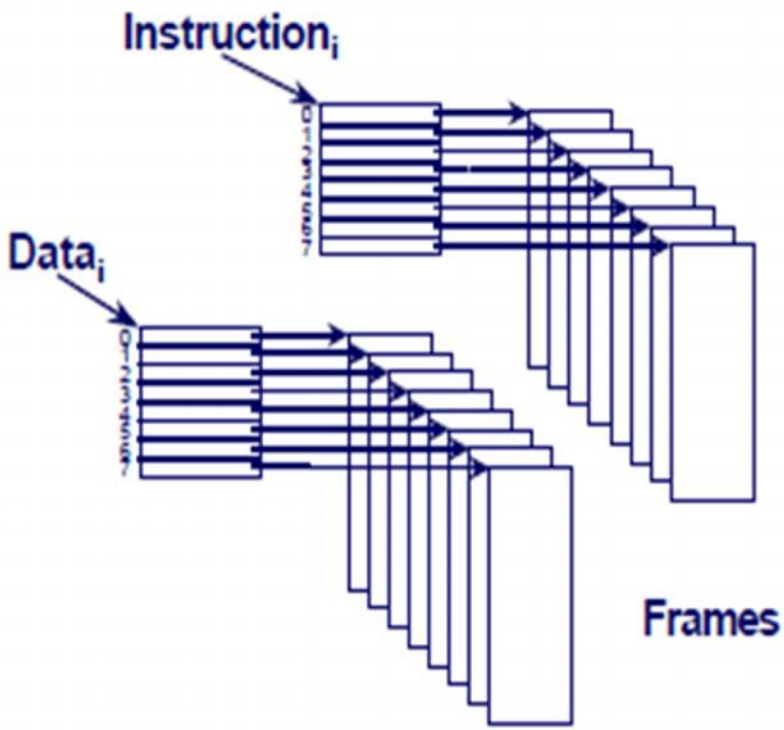


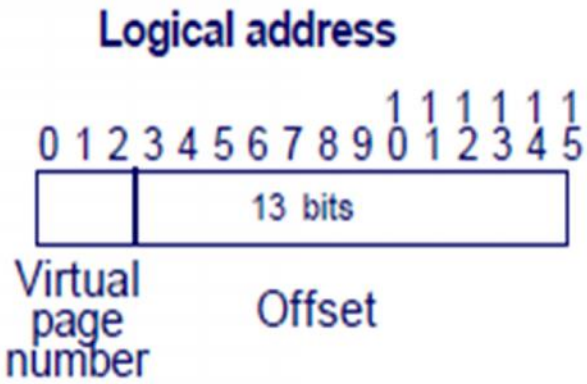
Multi-level paging – an example

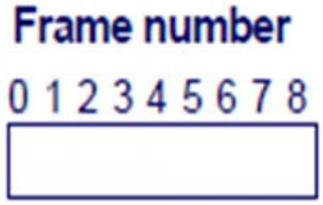


One level paging – The PDP 11

- The larger PDP-11 models have 16-bit logical addresses and up to 4MB of memory with page size of 8KB.
- There are two separate logical address spaces; one for instructions and one for data.
- The two page tables have eight entries, each
- Each controlling one of the eight frames per process

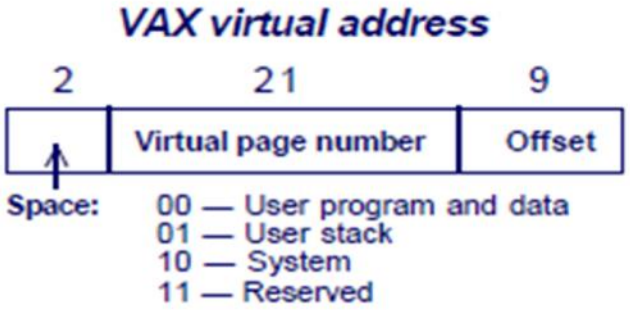






Two level paging – The VAX

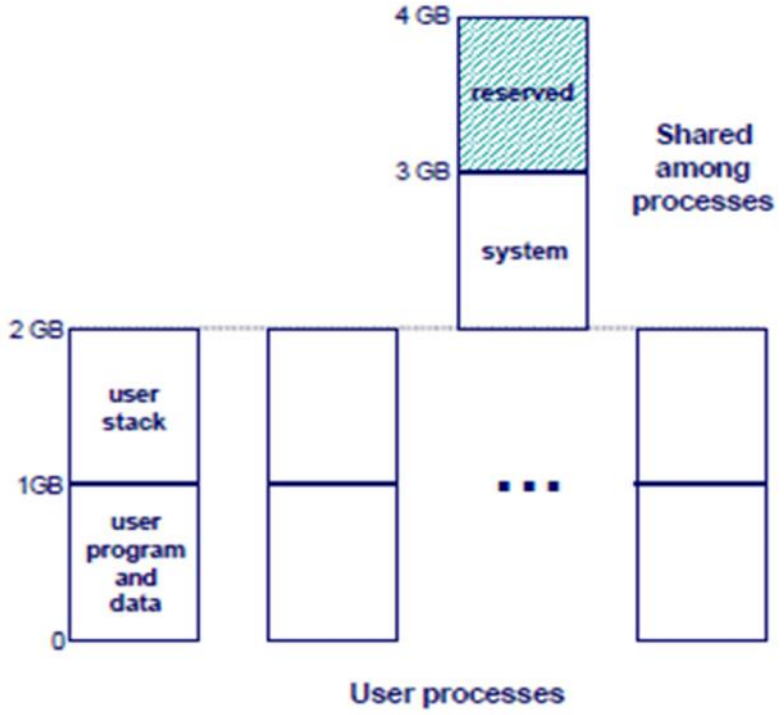
- The VAX is the successor of the PDP-11, with 32-bit logical addresses.
- The VAX has 512 byte pages.



$2^9 = 512$

$2^{13} = 8192$

$2^{21} = 2097152$



Other MMU architectures

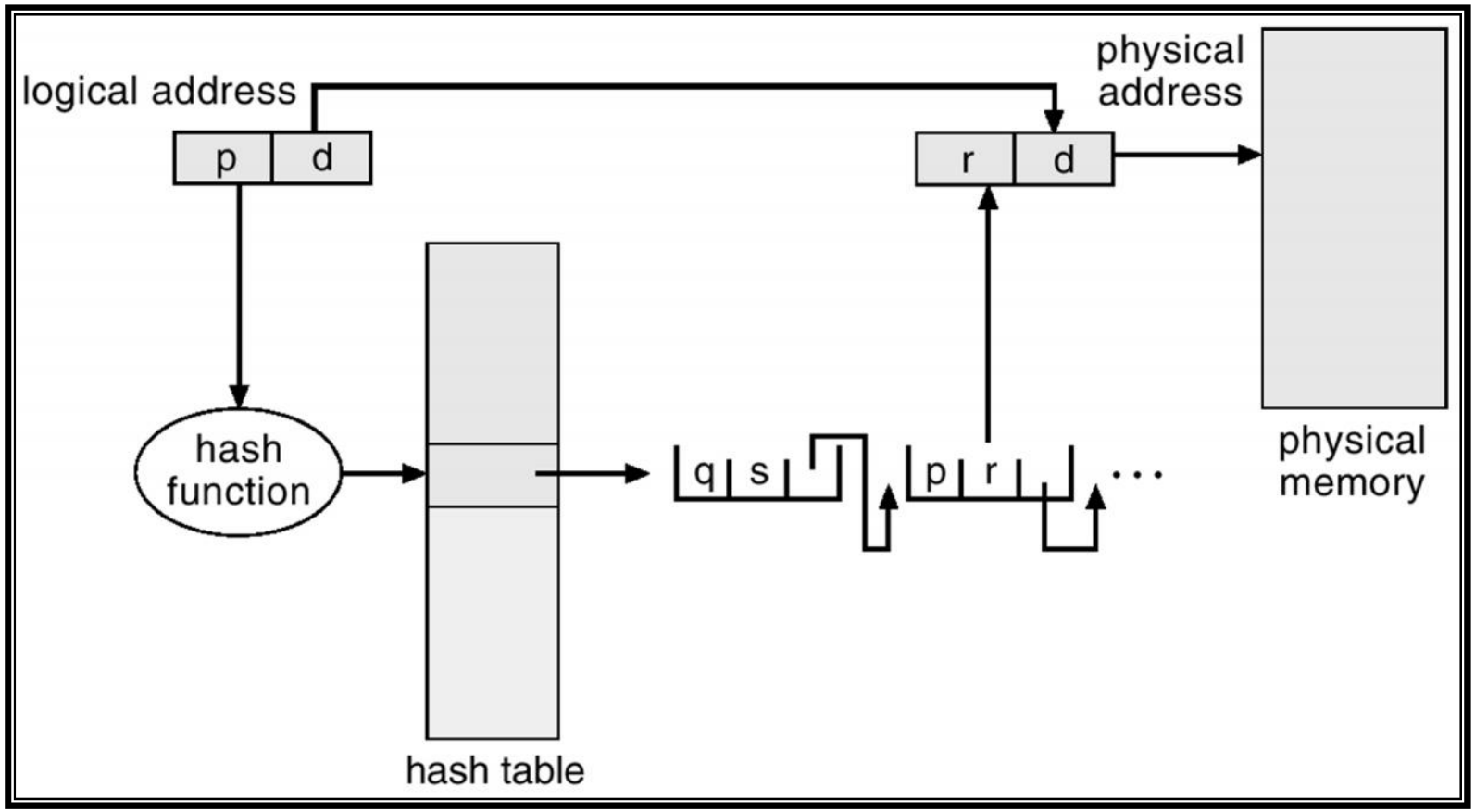
- Some SPARC processors used by the Sun workstations have a paging MMU with three-level page tables and 4KB pages.
- Motorola 68030 processor uses on-chip MMU with programmable multi-level (1-5) page tables and
- PowerPC processors support complex address translation mechanisms and, based on the implementation, provide 280 (64-bit) and 252 (32-bit) byte long logical address spaces.
- Intel Pentium processors support both segmented and paged memory with 4KB pages.

Hashed Page Table

- A common approach for handling address space larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collision).
- Each element consists of three fields:

1. The virtual page number
2. The value of the mapped page frame
3. A pointer to the next element in the linked list.

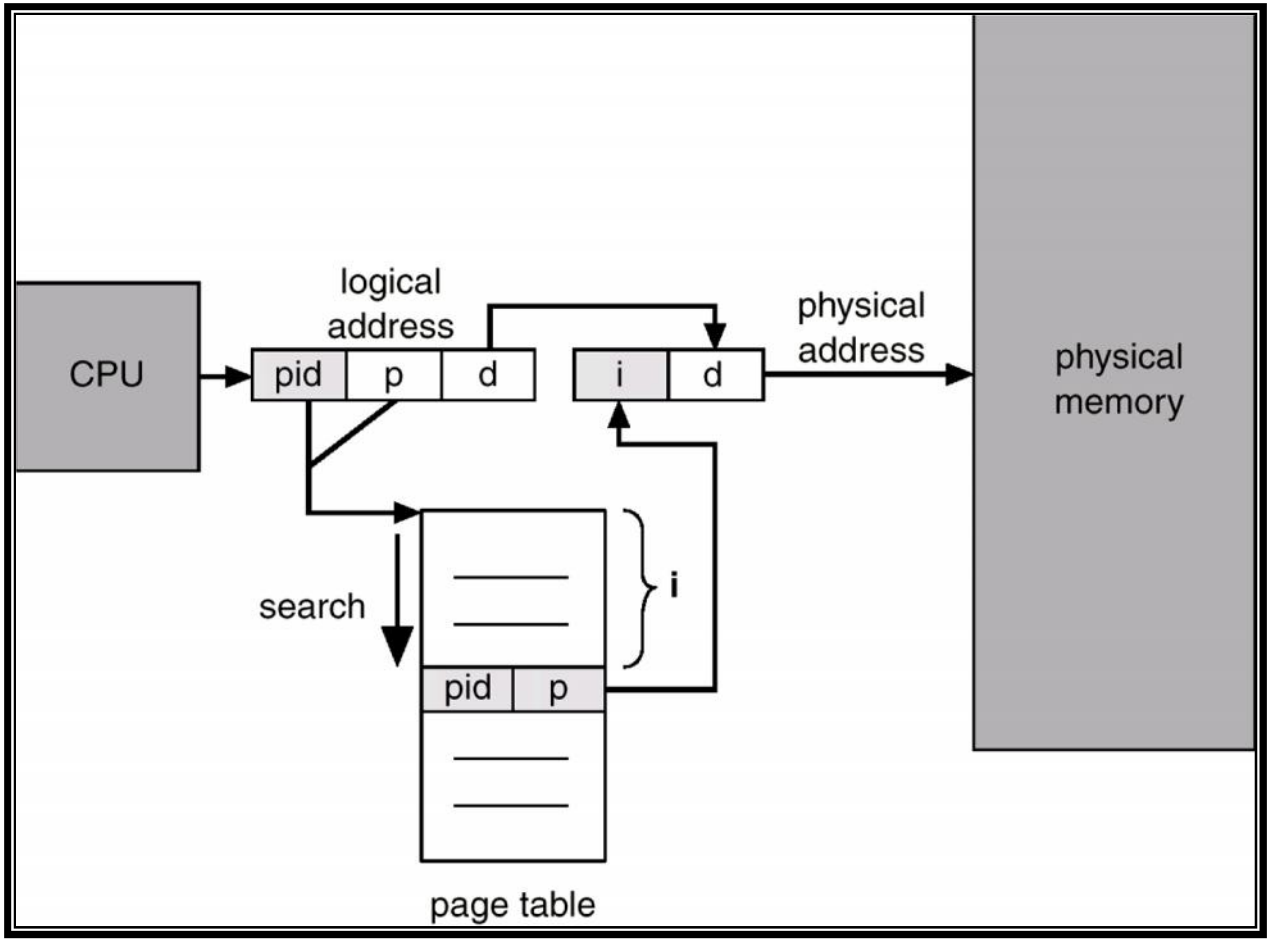
Hashed Page Table



Inverted Page Table

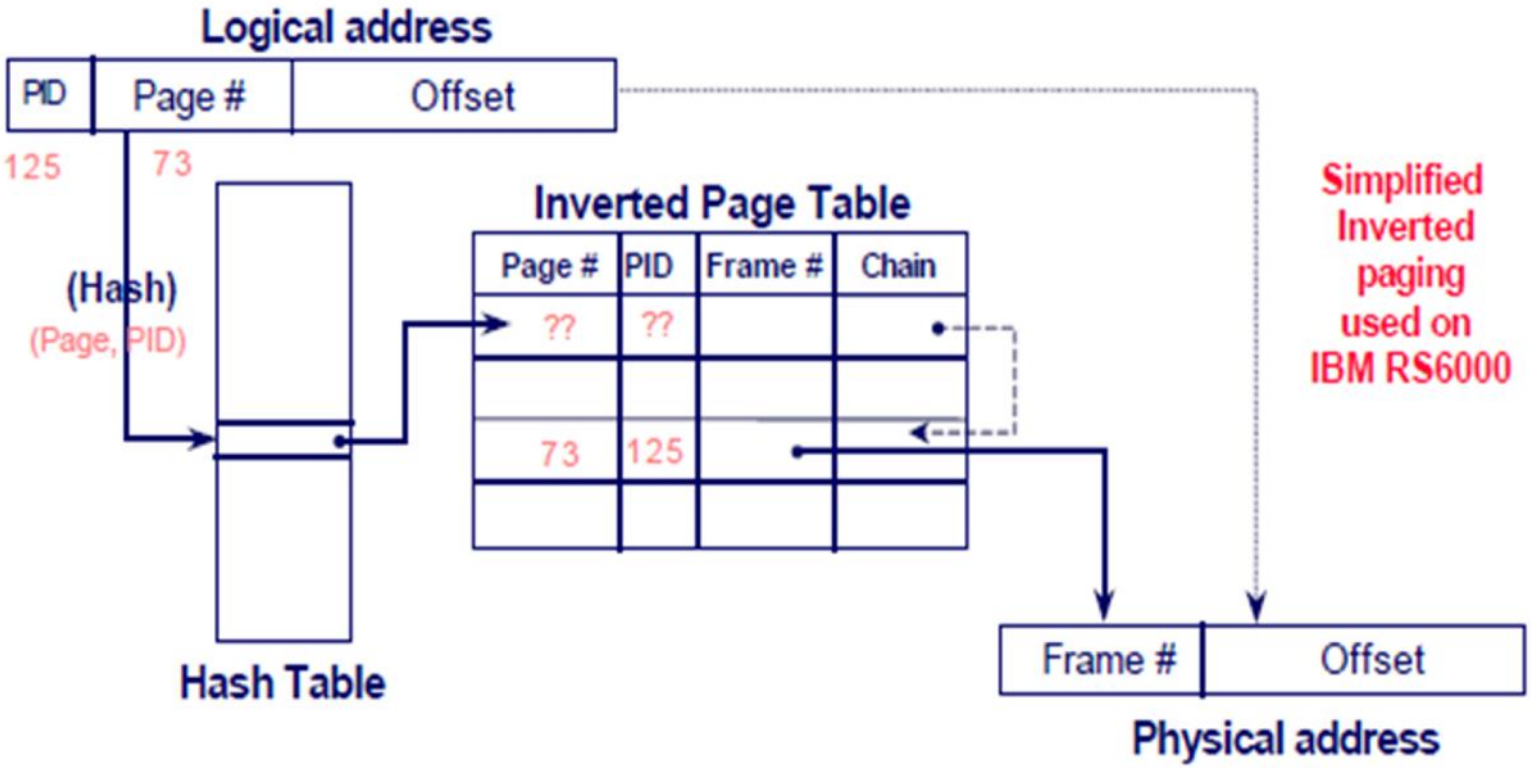
- Each process has an associated page table.
- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.

Inverted Page Table Architecture...



Inverted page table...

- The inverted page table has one entry for each memory frame.
- Adv: independent of size of address space; small table(s).
- Hashing is used to speedup table search.
- Here the inverted page table is system-wide, since the PID is shown.
- The Inverted Page Table can also be one per process.

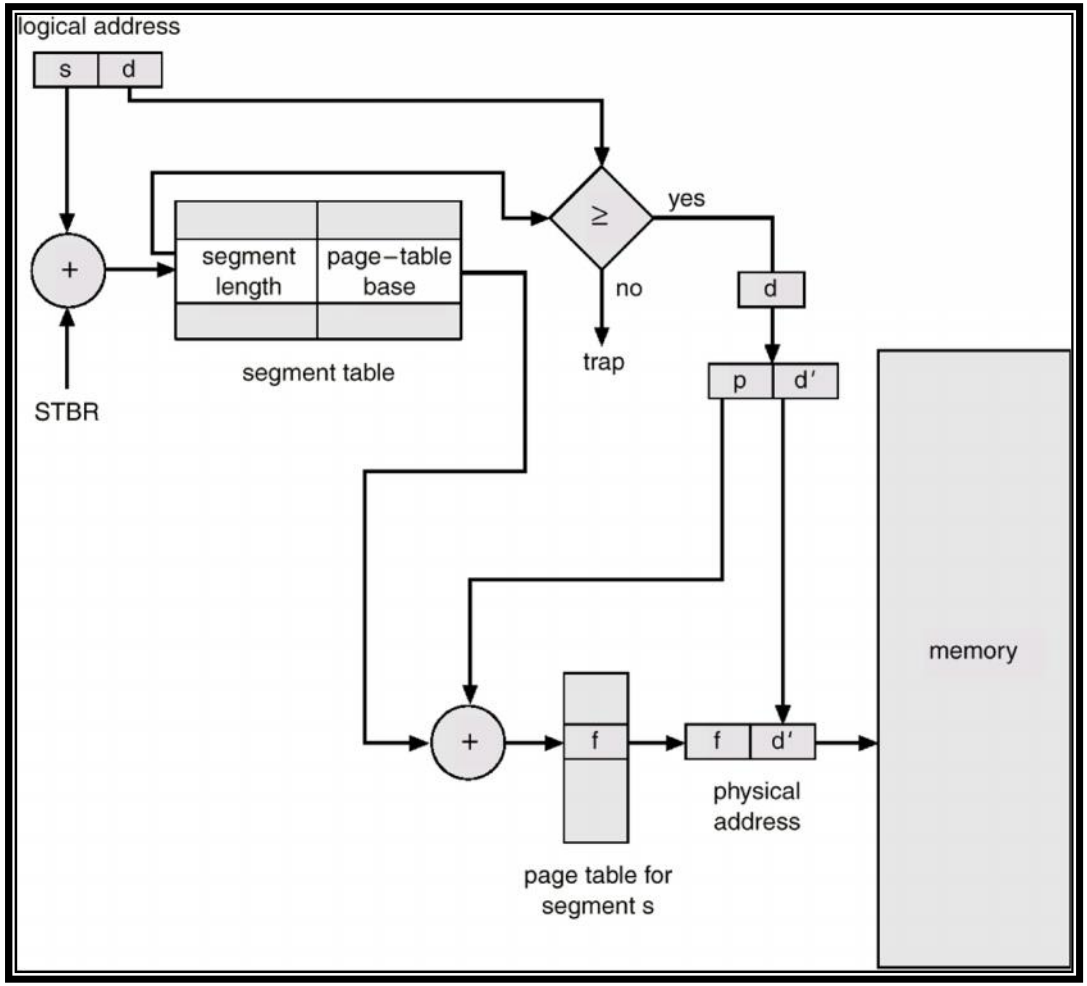


Segmentation with Paging – MULTICS

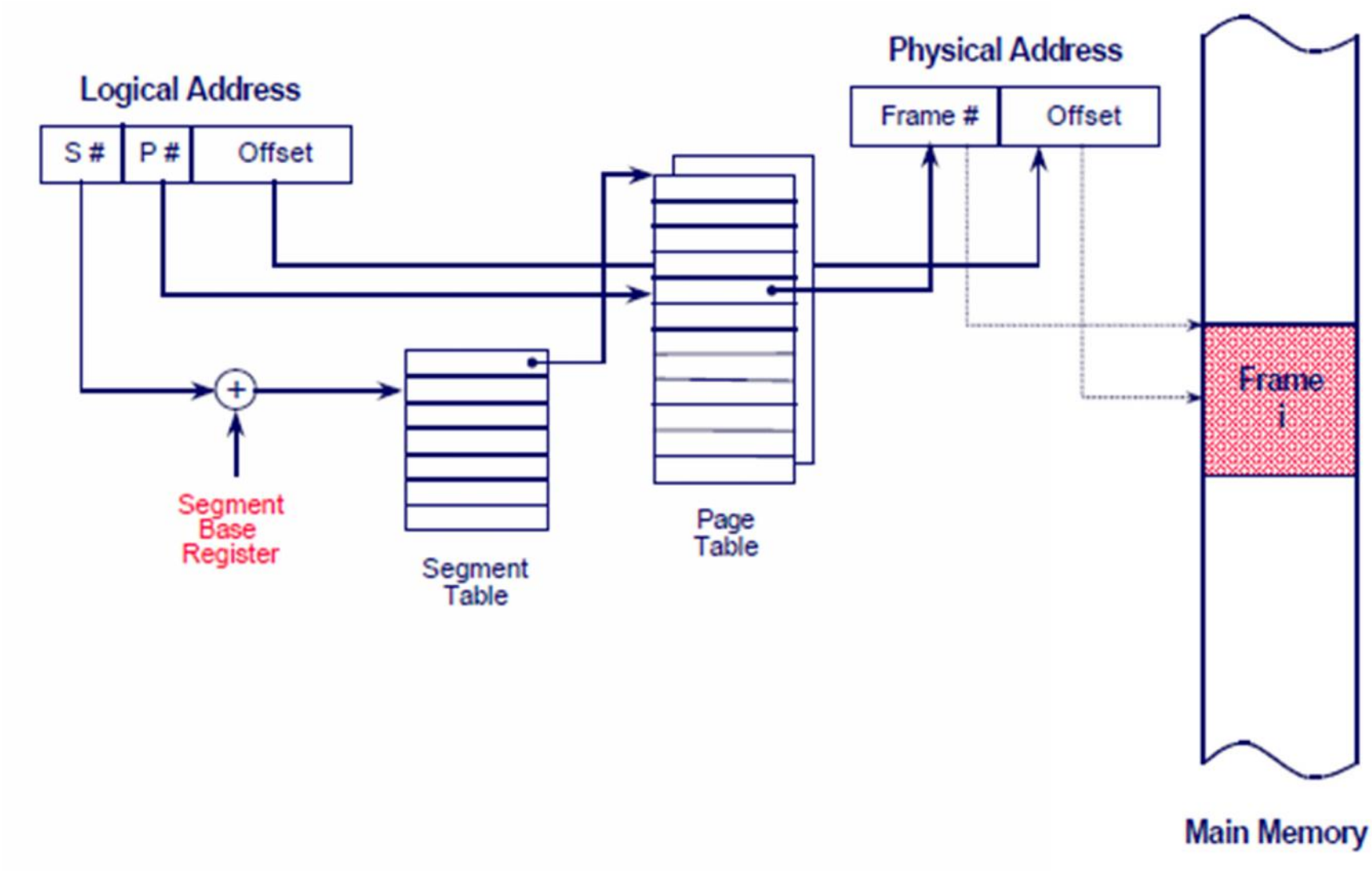
- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.

- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a page table for this segment.

MULTICS Address Translation Scheme



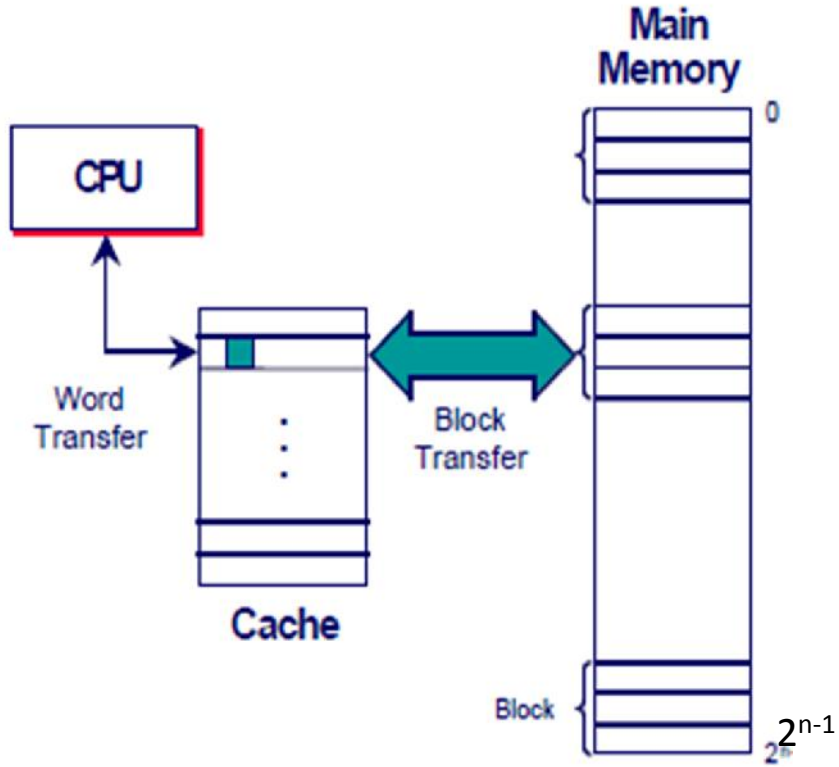
Segmentation with Paging



Memory catching

- Similar to storing memory addresses in TLBs,
- frequently used data in main memory can also be stored in fast buffers, called cache memory, or simply cache.
- memory access occurs as follows:

```
for each memory reference
if data is not in cache <miss>
  if cache is full
     remove some data (make space)
  if read access
     issue memory read
     place data in cache
     return data
  else <hit>
     if read access
        return data
    else
   update data in cache & memory
```



Cache terminology

- Cache hit: item is in the cache.
- Cache miss: item is not in the cache; must do a full operation.
- Categories of cache miss:
  - Compulsory : the first reference will always miss.
  - Capacity : non-compulsory misses because of limited cache size

- Effective access time:

  EAT= P(hit) * cost of hit + P(miss)* cost of miss

  P(miss) = 1- P(hit)

Issues in cache design

- Although there are many different cache designs, all share a few common design elements:
- Cache size — how big is the cache?
  - The cache only contains a copy of portions of main memory. The larger the cache the slower it is. Common sizes vary between 4KB and 4MB.
- Mapping function — how to map main memory blocks into cache lines?
  - Common schemes are: direct, fully associative, and set associative. (see later)
- Replacement algorithm — which line will be evicted if the cache lines are full and a new block of memory is needed.
    - A replacement algorithm, such as LRU, FIFO, LFU, or Random is needed only for associative mapping (Why?)

Issues in cache design...

- Write policy—What if CPU modifies a (cached) location?
  - This design issue deals with store operations to cached memory locations.
- Two basic approaches are:
  - write through - modify the original memory location as well as the cached data and
  - Write back (update the memory location only when the cache line is evicted.)

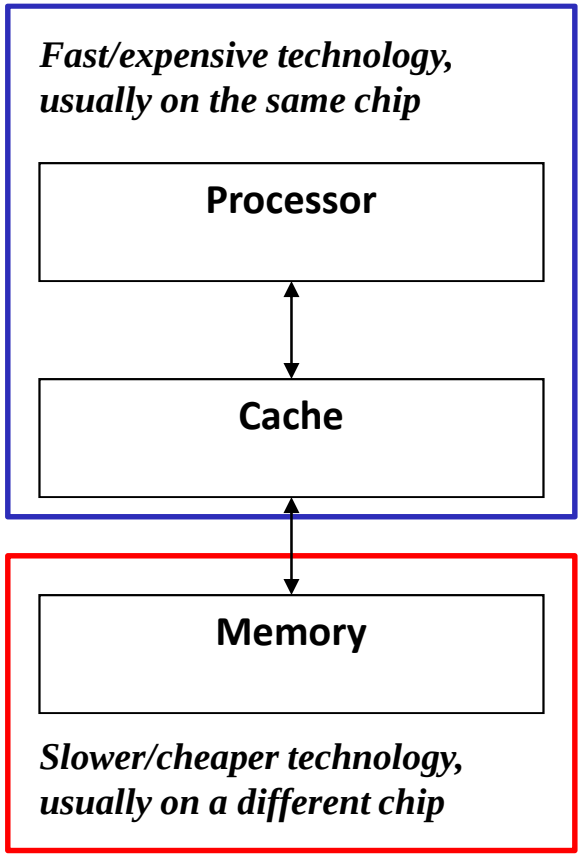
- Block (or line) size—how many words can each line hold?
  - Studies have shown that a cache line width of 4 to8 addressable
  - units (bytes or words) provide close to optimal number of hits.

Issues in cache design...

- Number of caches—how many levels? Unified or split cache for data and instructions?
- Studies have shown that a second level cache improves performance.
- Pentium and Power PC processors each have onchip level-1 (L1) split caches.
- Pentium Pro processors have onchip level-2 (L2) cache, as well.

Cache Memory...

- Memory access may be slow
- Cache is small but fast memory close to processor
  - Holds copy of part of memory
  - Hits and misses



Mapping function

- Since there are more main memory blocks (Blocki for i=0 to n) than cache lines (Linej for j=0 to m, and n >> m), an algorithm is needed for mapping main memory blocks to cache lines.
- Direct mapping—maps each block of memory into only one possible cache line.
  - $Block_i$ maps to $Line_j$, where i = j modulo m.
- Associative mapping—maps any memory block to any line of the cache.
- Set associative mapping—cache lines are grouped into sets and a memory block can be mapped to any line of a cache set.
  - $Block_i$ maps to Set_j where i=j modulo v and v is the number of sets with k lines each.

Set associative cache organization