Chapter 8: Main Memory

Chapter 8: Memory Management

- **Background**
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

Objectives

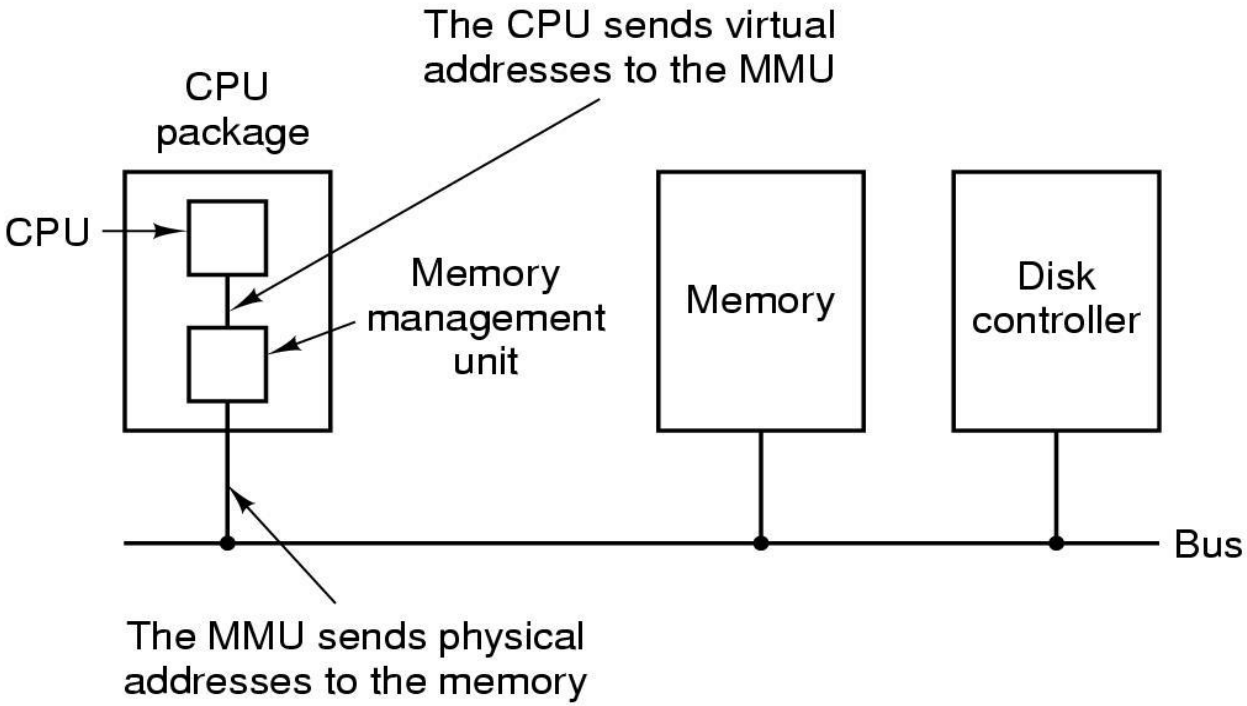
- To increase CPU utilization and response speed:
  - Several processes must be kept in main memory
  - We must share memory
  - How to manage main memory resource…?
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including ***paging*** and ***segmentation***

Background

- Program must be brought (from disk) into memory and be placed within a process for it to be run
- Main memory and registers are the only storage CPU can access directly
- Memory unit only sees a stream of
  - addresses + read requests, or address + data and write requests

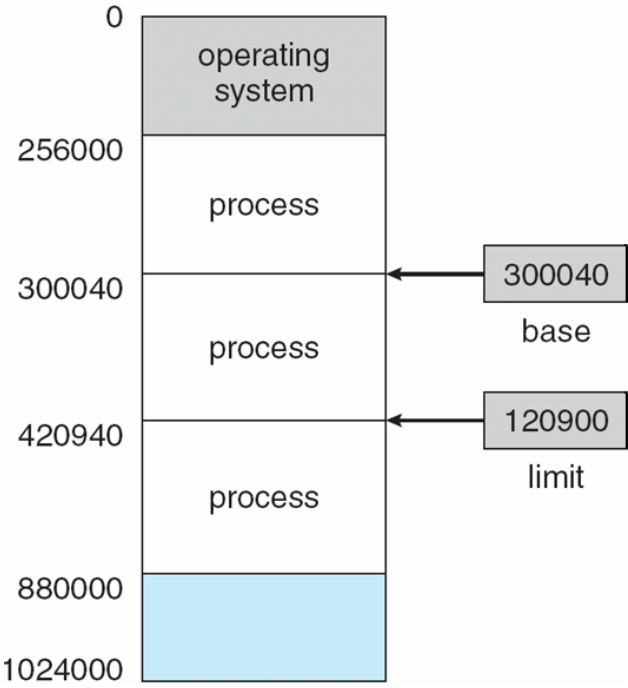
CPU, MMU and Memory

- Register access in one CPU clock (or less); very fast memories
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers; solution to stall issue
- Protection of memory required to ensure correct operation; hardware-level



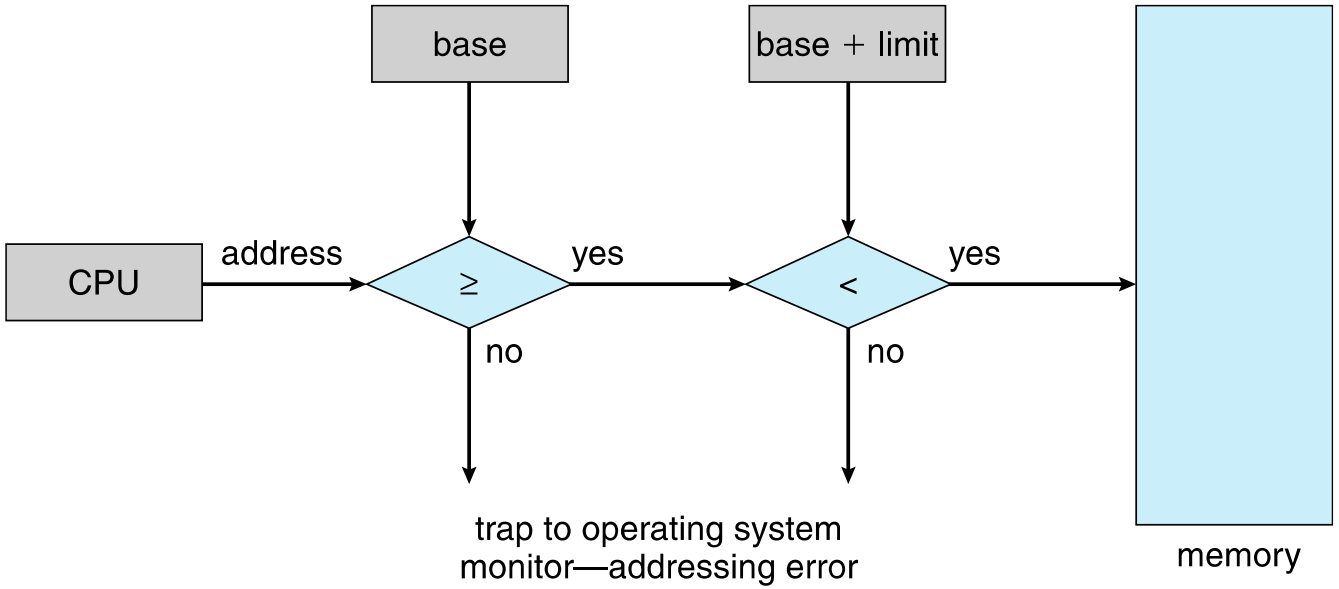
Protection of Memory: Base and Limit Registers

- Processes should not be able to ***reference*** memory locations in another process without permission.
- A pair of **base** and **limit registers** define the logical address space
  - To protect processes from each other ; each process has its own memory space
  - **Base register** holds the smallest legal physical memory address
  - **Limit register** specifies the size of the range of accessible addresses



Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user; to protect a process's memory space



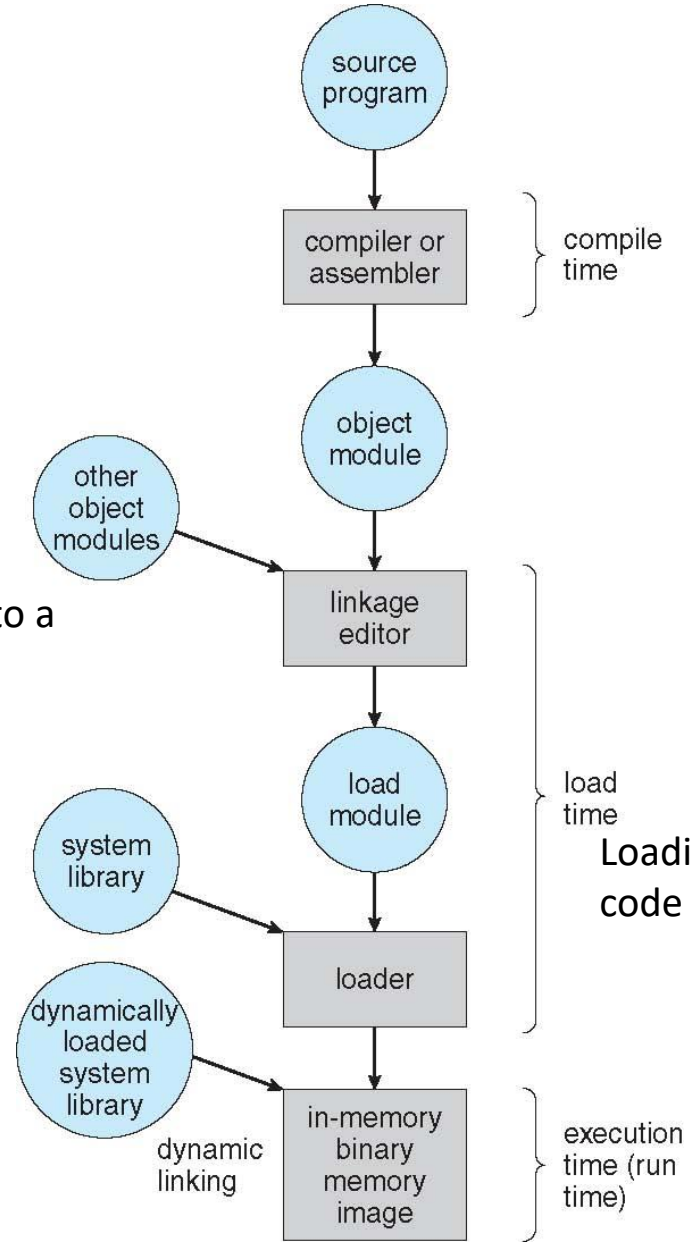
- Base and limit registers loaded only by the OS through a privileged instruction
  - To prevent users from changing these registers' contents
- OS has unrestricted access to OS memory and user memory
  - Load users' programs into users' memory, … etc
  - Access and modify system-calls' parameter, … etc

Multistep Processing of a User Program

Compiler→ generates object code

Linker → Combines the Object code into a single self sufficient executable code

Execution → dynamic memory allocation



Loading → Copies executable code into memory

Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
- Further, addresses represented in different ways at different stages of a program's life
- Source code addresses usually symbolic. They are ***names*** of variable, e.g., *count*
  - Compiled code addresses **bind** to ***relocatable addresses***
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind ***relocatable addresses*** to ***absolute addresses***
    - i.e. 74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

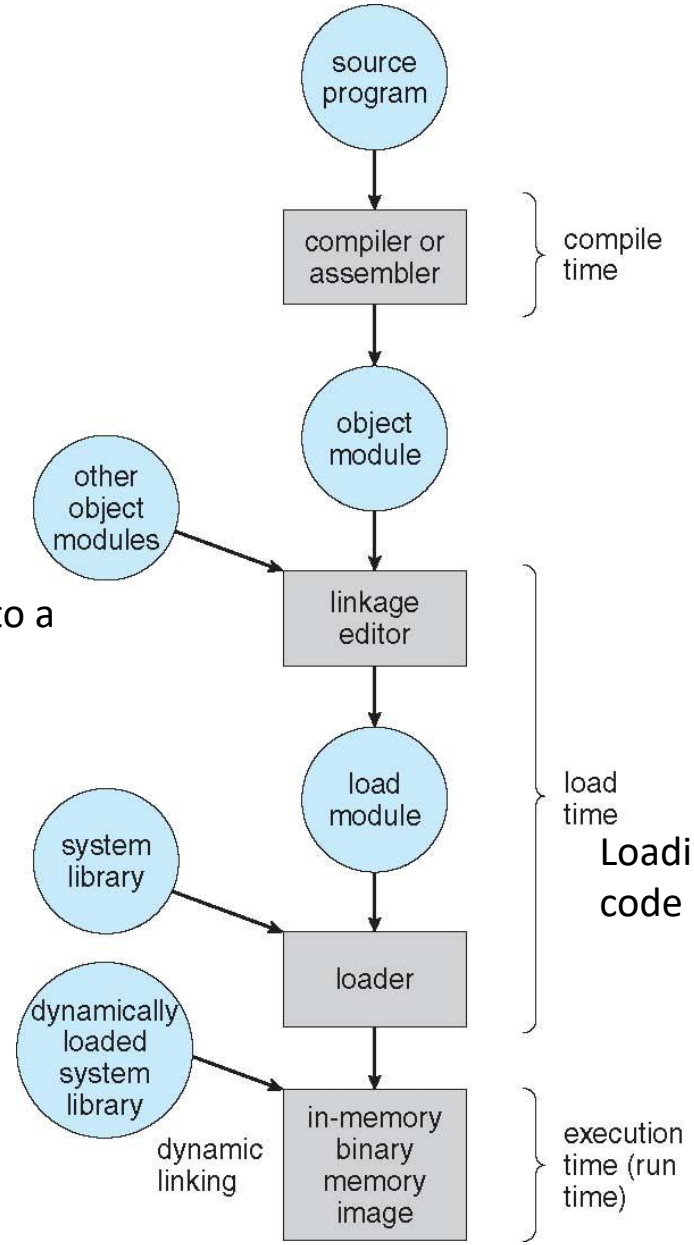
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If it is known at compile time where a program will reside in physical memory, then ***absolute code*** can be generated by the compiler, containing actual physical addresses.
  - **Load time**: If the location at which a program will be loaded is not known at compile time, then the compiler must generate **relocatable code**, which references addresses *relative* to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
  - **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

Compiler→ generates object code

Linker → Combines the Object code into a single self sufficient executable code

Execution → dynamic memory allocation



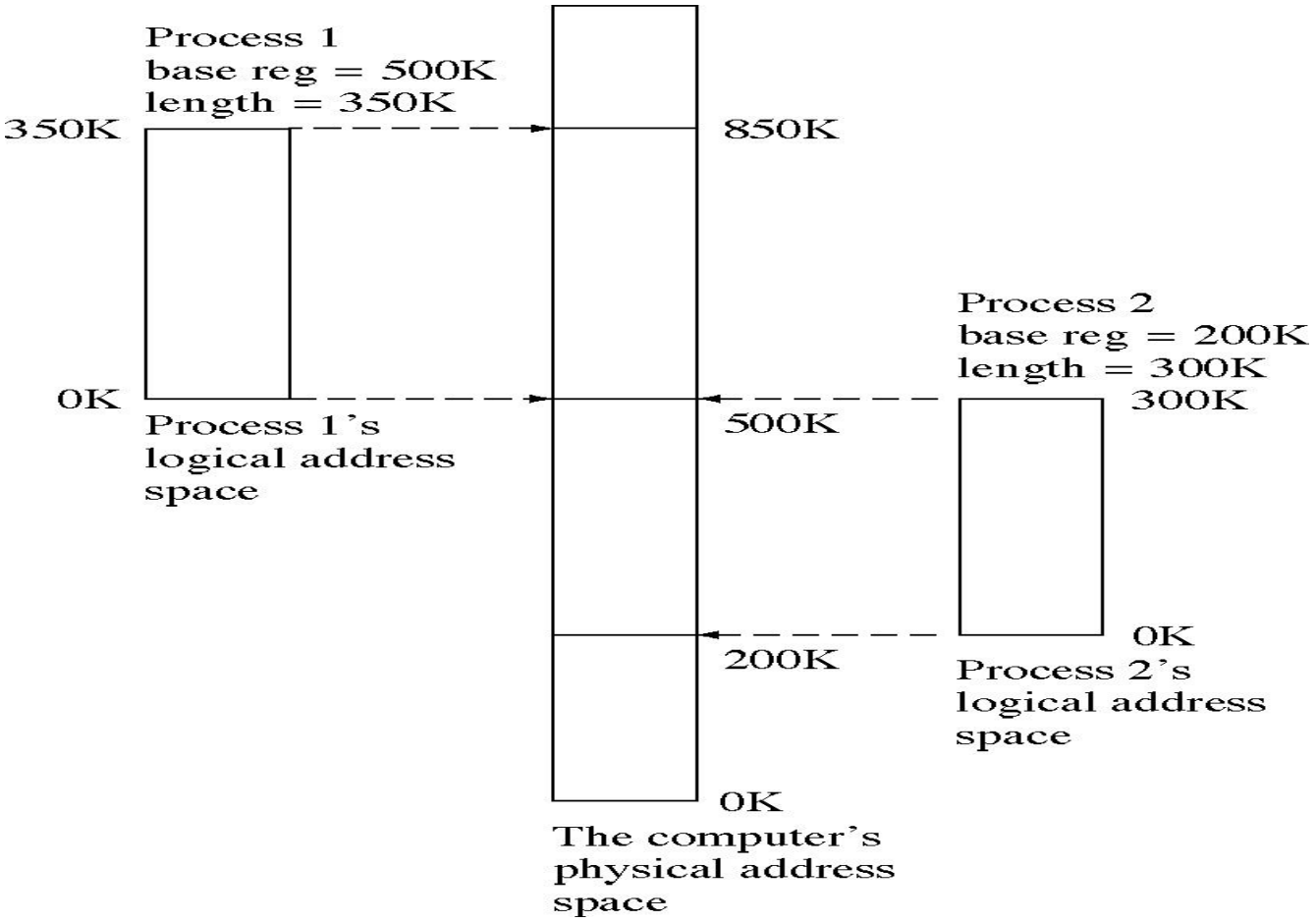
Loading → Copies executable code into memory

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-management unit** in its **memory-address register**
- The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**.

Logical and Physical Address Spaces

- Assume there are two processes, process 1 and process 2:
- Process 1's length is 350K and Process's 2 length is 300K

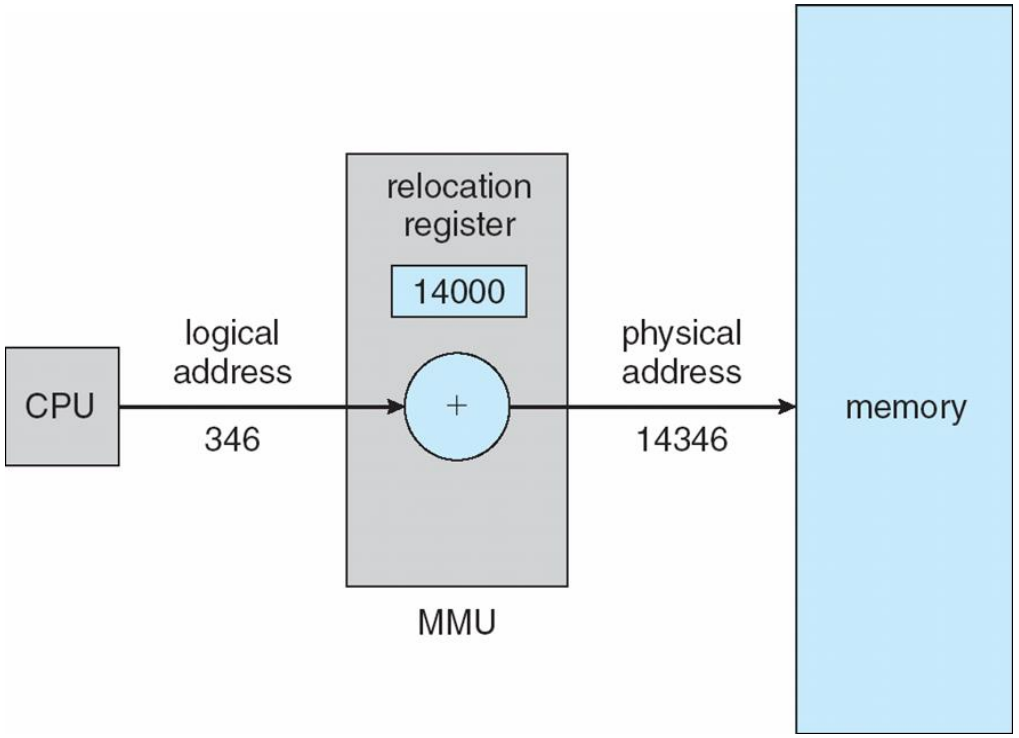


Memory-Management Unit (MMU)

- MMU is the hardware that does virtual-to-physical addr mappings at run-time
- Hardware device that at run-time maps virtual address to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the ***relocation register*** is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register;**
- The user program deals with ***logical* addresses**; it never sees the **real physical addresses**
  - Logical address mapped to physical addresses before use
    - Logical addresses: in range **0 to *max***
    - Physical addresses: in range ***R* + 0 to *R* + *max***
      - For a base value *R* in the relocation register

Memory Management Unit

- The runtime mapping from virtual → physical address



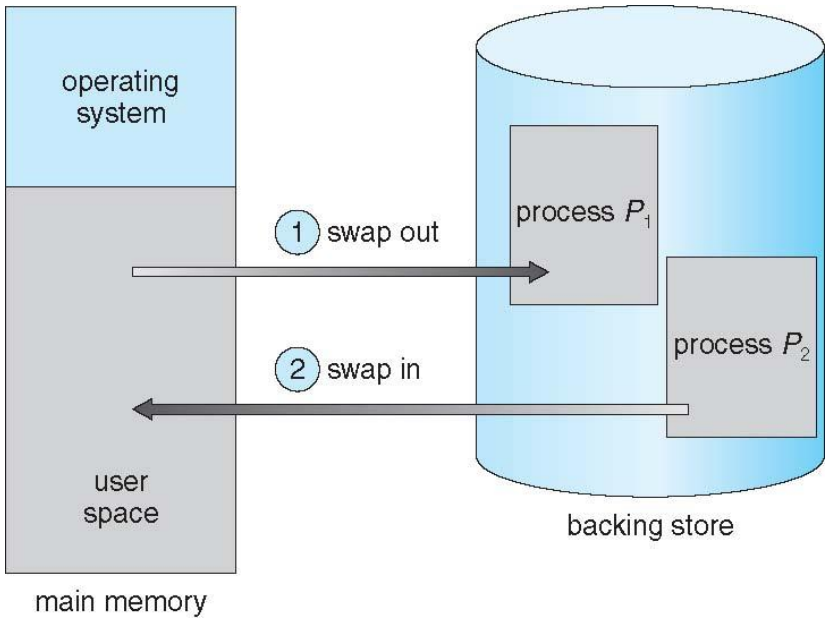
- Relocation register is added to every address generated by user process

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

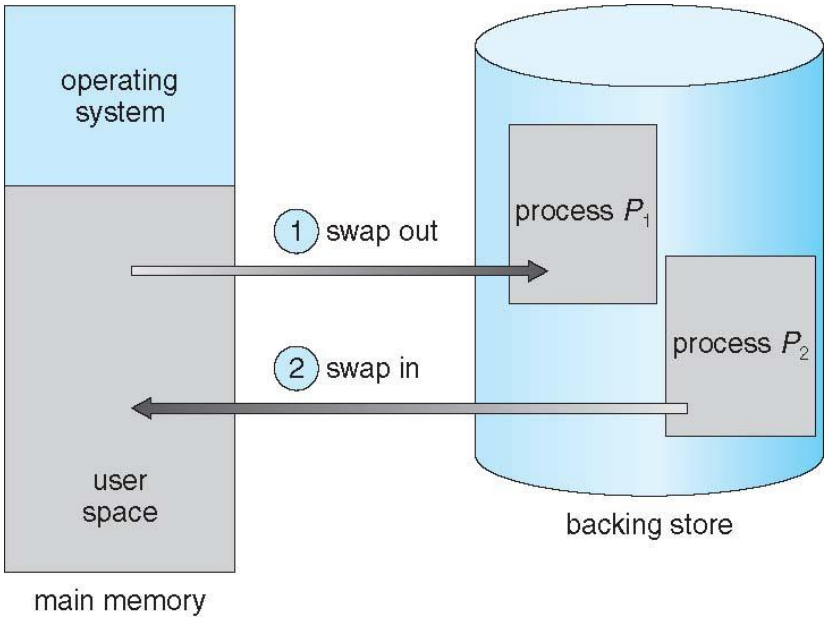
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
    - Hence, swapping increases the degree of multiprogramming
- Backing store – fast disk large enough to accommodate copies of all memory images for all processes; must provide direct access to these memory images



Schematic View of Swapping

- System maintains a ready queue of ready-to-run processes which have memory images on the backing store or in memory. **Standard swapping method:**
  - Dispatcher called when the CPU scheduler selects a process P_2 from queue
    - If not enough free space in memory for P_1
      - Swap in P_2 from backing store and swap out some P_1 from memory
      - Reload registers and transfer control to P_2; i.e. P_2 is now in running state



Context Switch Time including Swapping

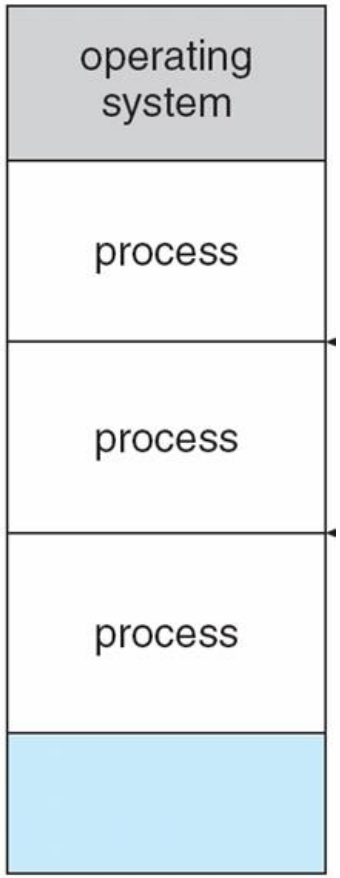
- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms = 2seconds
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- We have ignored other components of context-switch time and ignored other disk performance aspect. **But, the major part of swap time is transfer time**
  - Can reduce if reduce size of memory swapped – by knowing how much memory really being used

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

Contiguous Memory Allocation

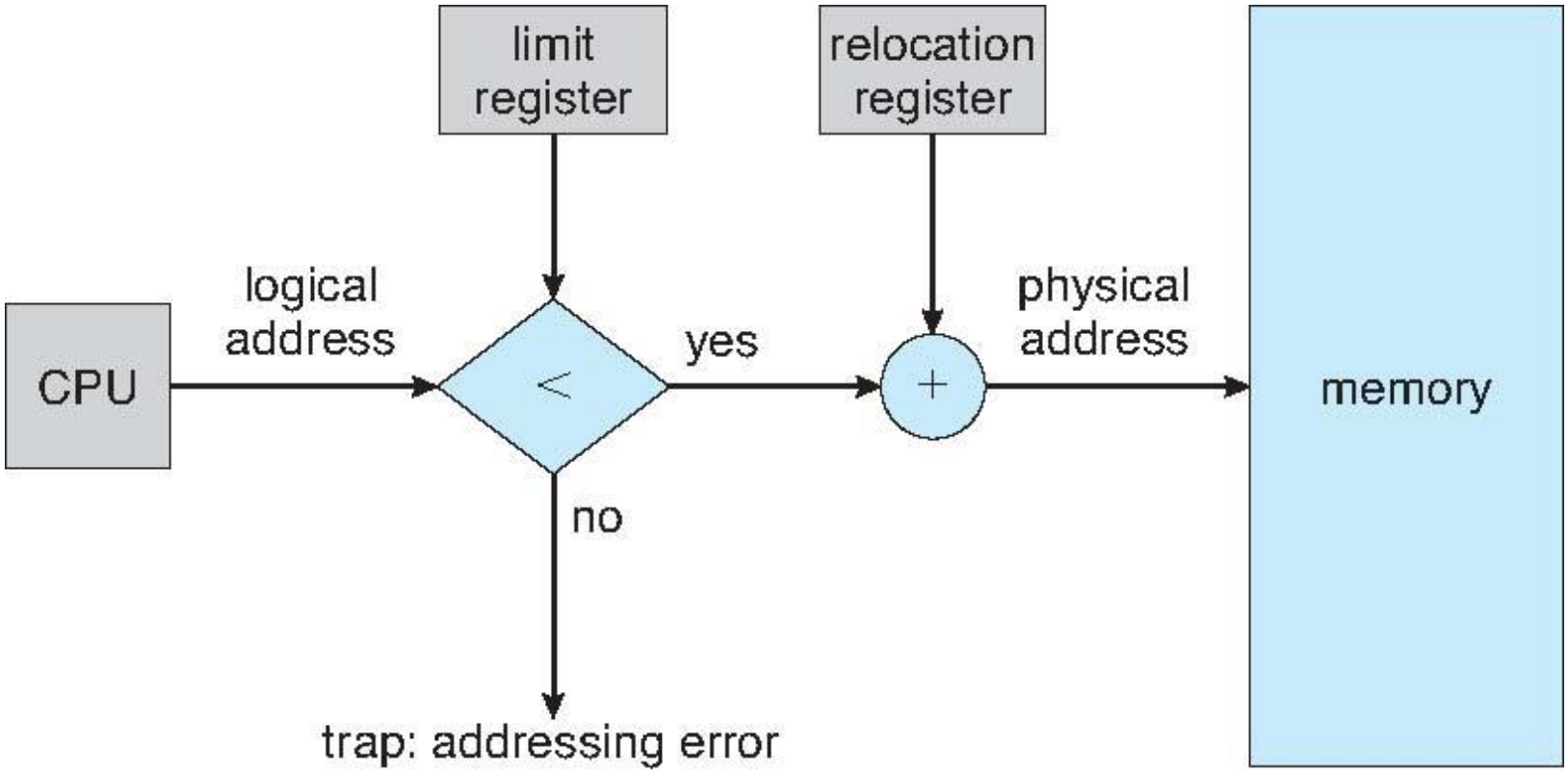
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually divided into two **partitions**:
  - Resident operating system usually held in low memory
  - User processes then held in high memory
  - Each process contained **in single contiguous** section of memory



Contiguous Allocation-memory protection

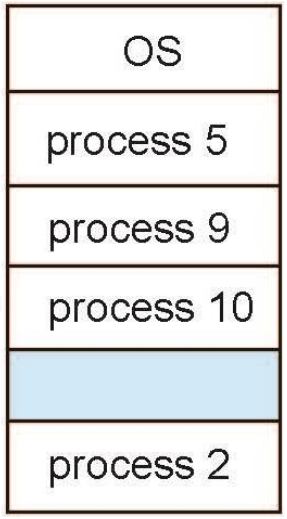
- We can protect a process's memory by combining ideas from previous slides
- **Relocation registers** used to protect user processes from each other, and from changing operating-system code and data
  - For each process:
    - **Relocation register** contains value of smallest physical address;
    - **Limit register** contains range of logical addresses
      - each logical address must be less than the limit register ;
  - MMU maps logical address *dynamically*
    - By adding logical address to relocation value;
  - Dispatcher always loads these two registers with their correct values
    - All logical addresses are checked against Limit register
    - Thus, protecting both each OS and each user program and data

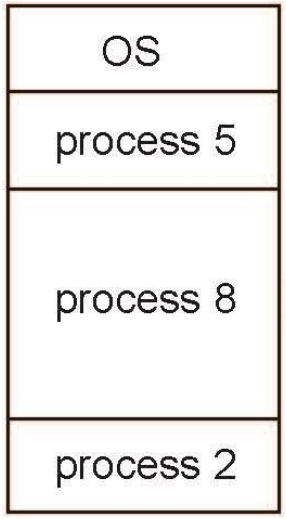
Hardware Support for Relocation and Limit Registers



Multiple-partition allocation

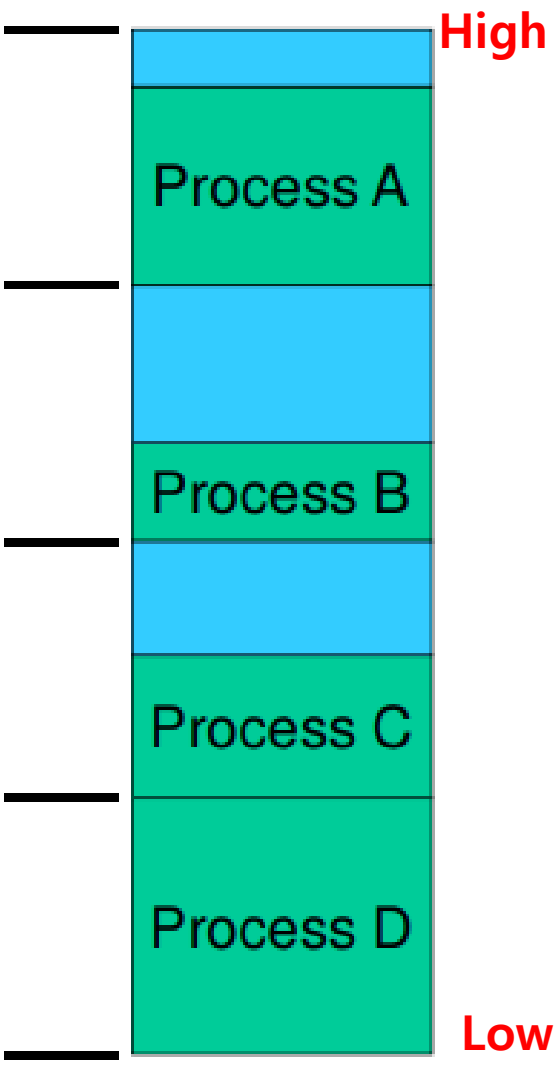
- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - Example: 2 approaches
    - Fixed Partition
      - equal
      - unqual size partitions
    - Variable Partition (or Dynamic Partitioning)





Old Technique#1: Fixed, Equal Partitions

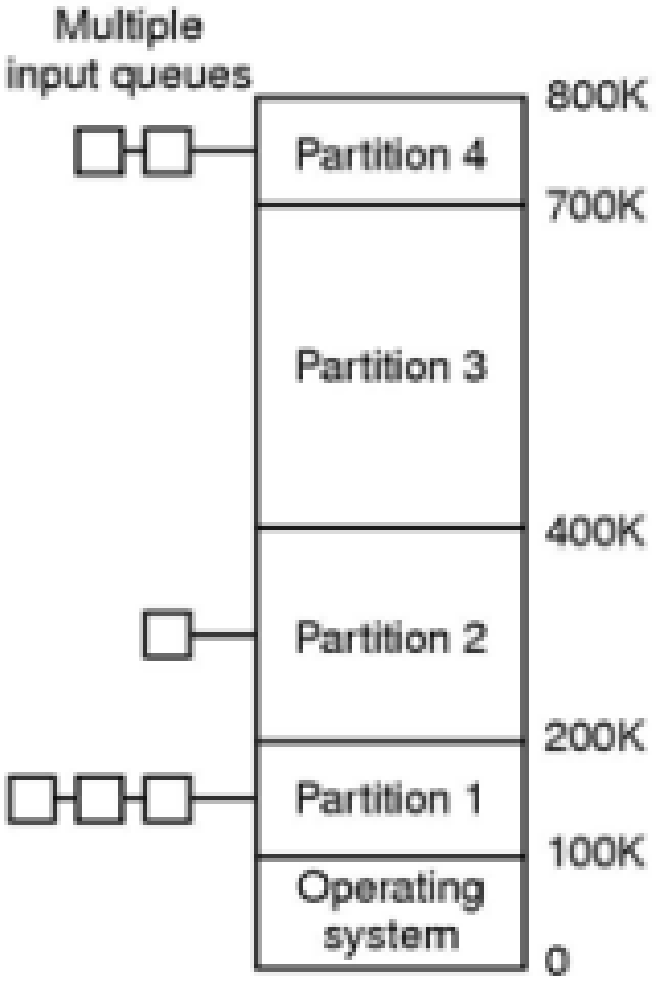
- Physical memory is broken up into fixed, equal partitions
  - hardware requirement: base/relocation register, limit register
    - physical address = logical address + base register
    - base register loaded by OS when it switches to a process
- All partitions are of equal size, small processes waste space, Large processes don't fit in (Solution: Fixed Unequal Partitions)



Old Technique#1: Fixed, UnEqual Partitions

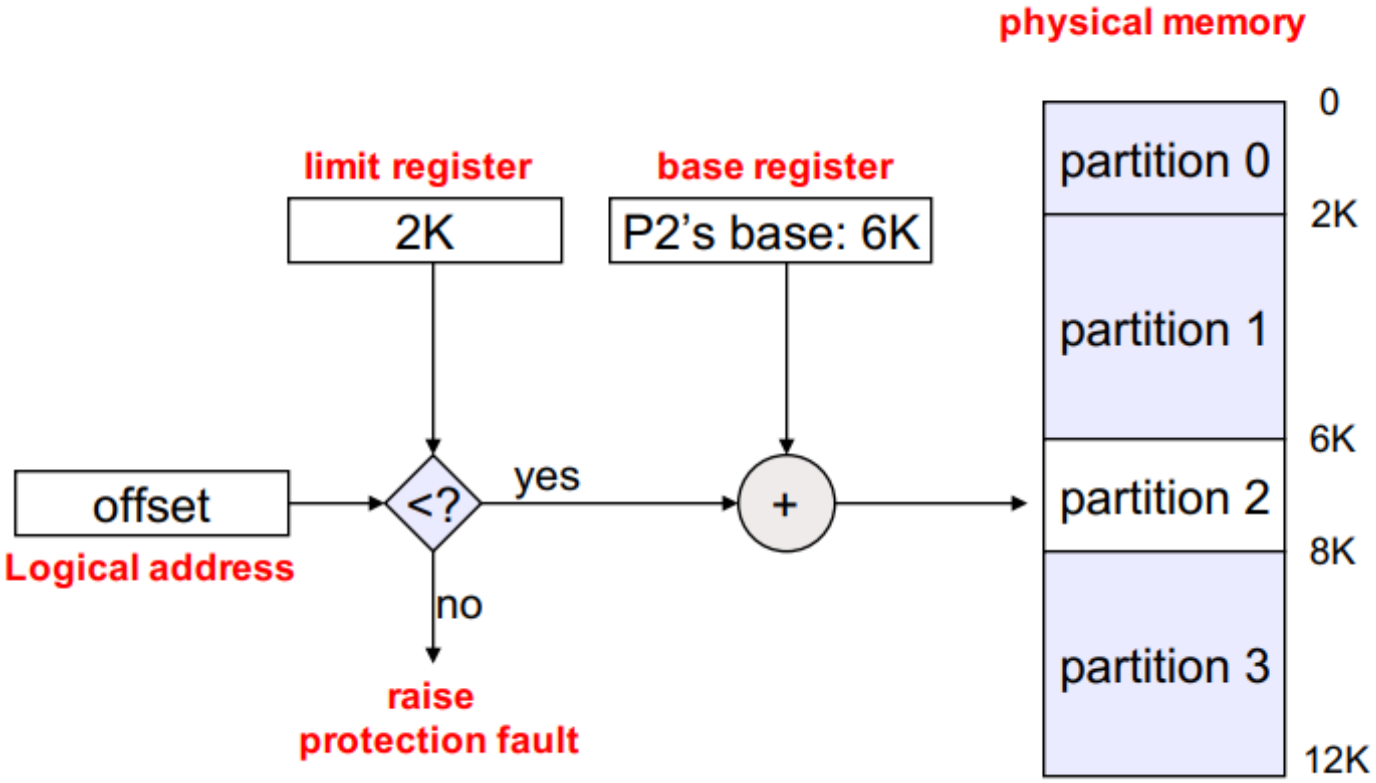
- Physical memory is broken up into fixed but unequal partitions

- Multiple Queues:
  - Place process in queue for smallest partition that it fits in.



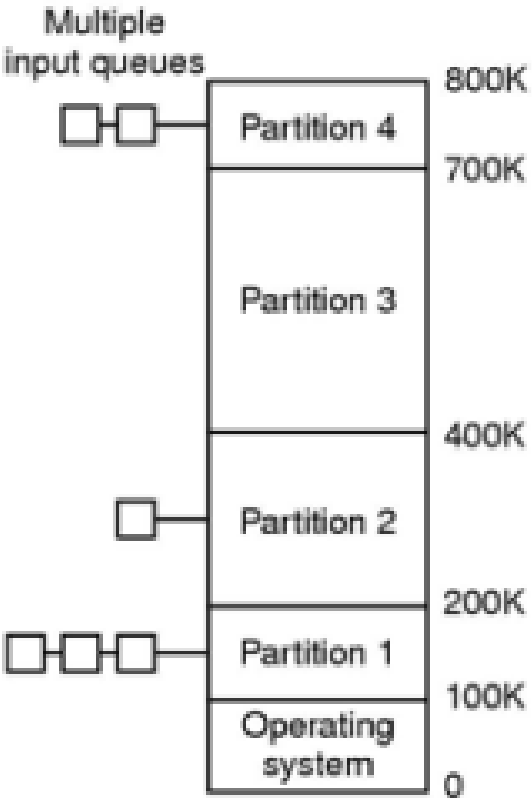
Mechanisms of Fixed Partitions (Equal, Unequal)

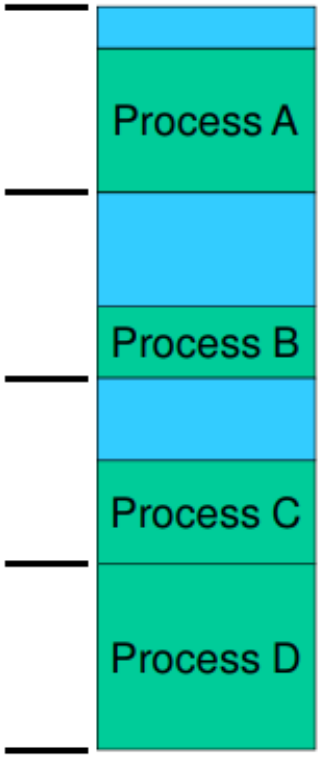
- Logical address is generated by the CPU.
- Suppose Process 2 is resides in partition 2.
- Limit register for process 2 is 2k and Base = 6k..
- Offset is displacement inside the user process. E.g., 1,2,3,4,..upto max 2K for the Process 2 in the below example.



Internal Fragmentation in Old Technique#1: Fixed Partitions

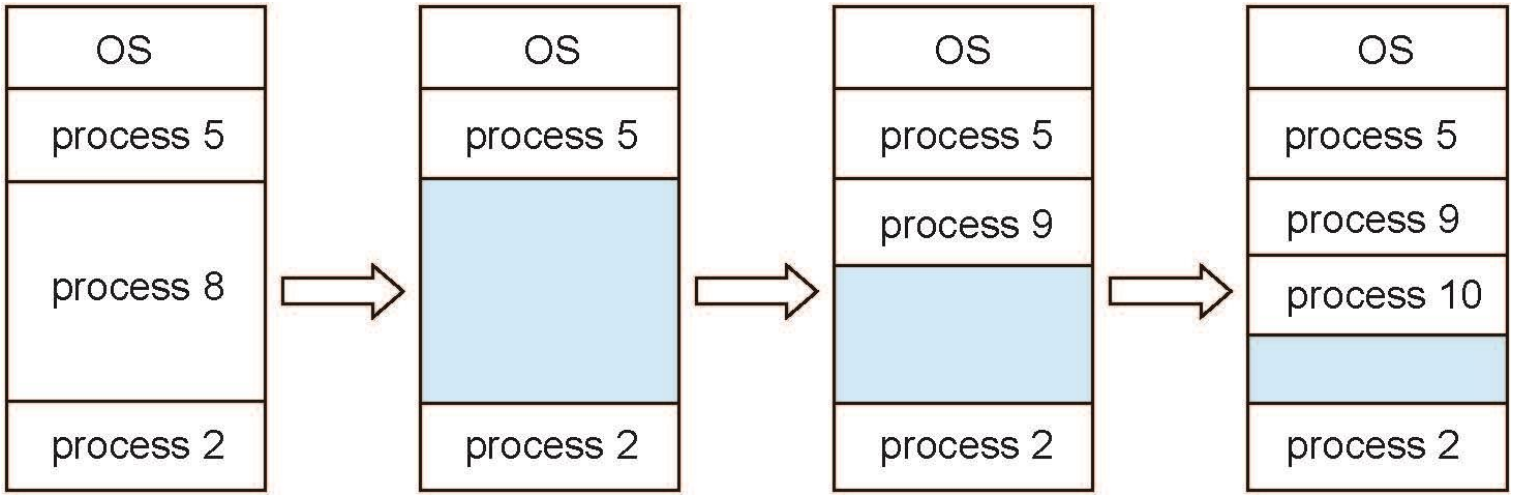
- Advantages of Fixed Partitions (Equal or Unequal)
  - Simple
- Problems
  - **internal fragmentation**: the available partition is larger than what was requested. ; this size difference is memory internal to a partition, but not being used



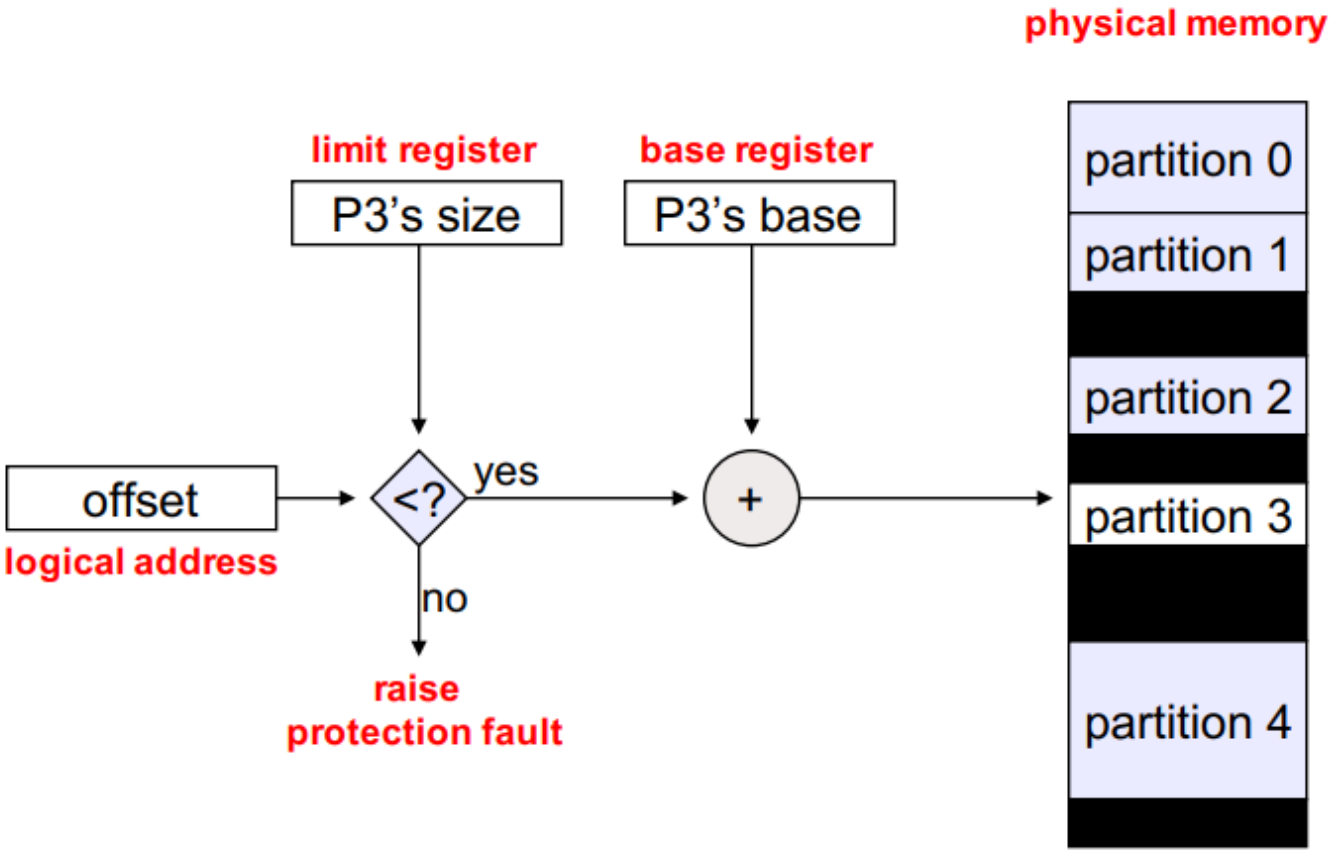


Old Technique#2: Variable Partitions (Dynamic Partitioning)

- Obvious next step: physical memory is broken up into partitions **dynamically** – partitions are tailored to processes
  - hardware requirements: **base register, limit register**
  - physical address = logical address + base register
- Advantages: no ***internal fragmentation***
  - simply allocate partition size to be just big enough for process (assuming we know what that is!)
- Problems: **external fragmentation**
  - as we load and unload jobs, holes are left scattered throughout physical memory



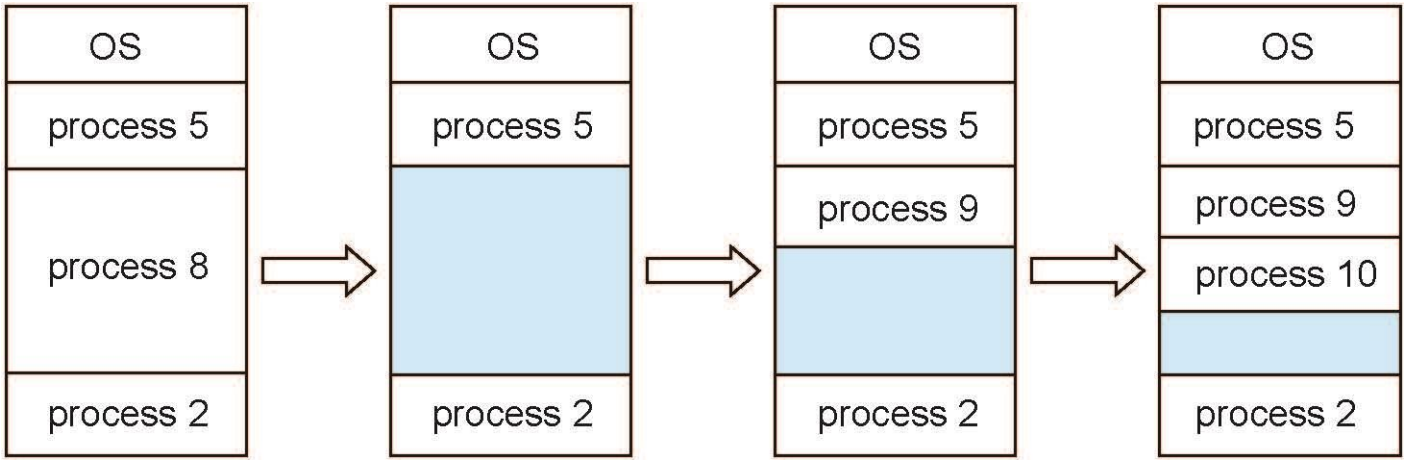
Mechanisms of variable Partitions



Multiple-partition allocation

- Multiple-partition allocation
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
    - **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
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:

    a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

1. **First-fit**: Allocate the ***first*** hole that is big enough

2. **Next-fit:** Allocate the ***next*** available hole that is big enough

3. **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size

   - Produces the smallest leftover hole

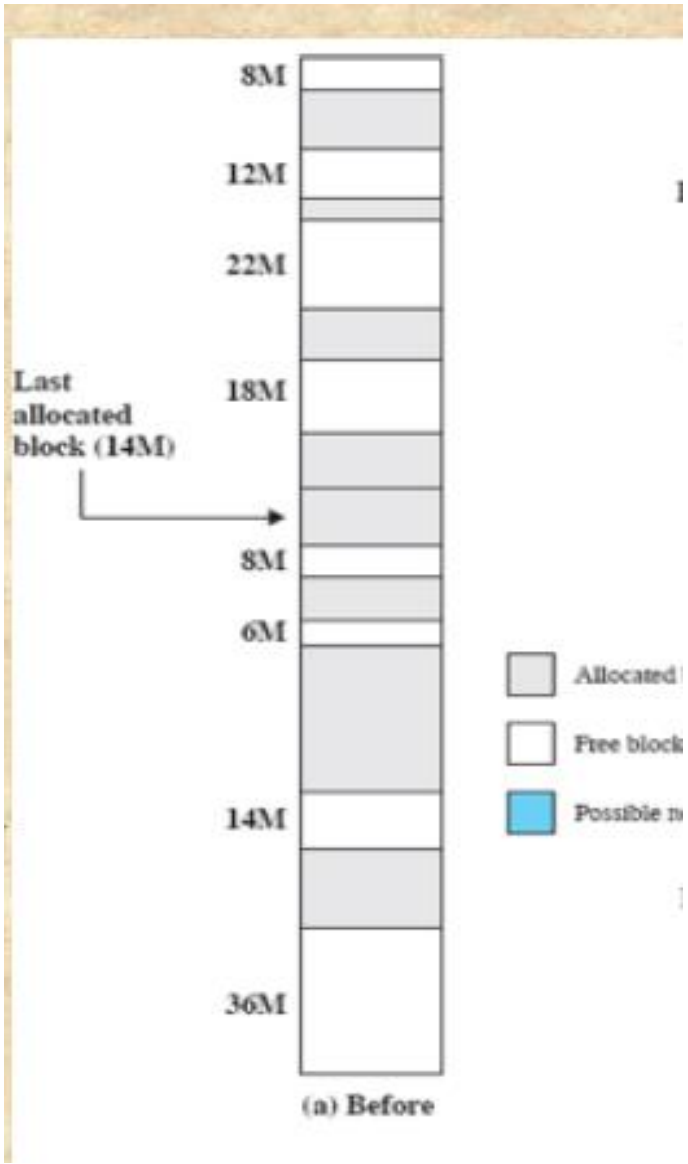
4. **Worst-fit**: Allocate the ***largest*** hole; must also search entire list

   - Produces the largest leftover hole
     - Which may be more useful than the smaller leftover hole from a ***best-fit*** approach;

- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
  - First-fit is generally the fastest

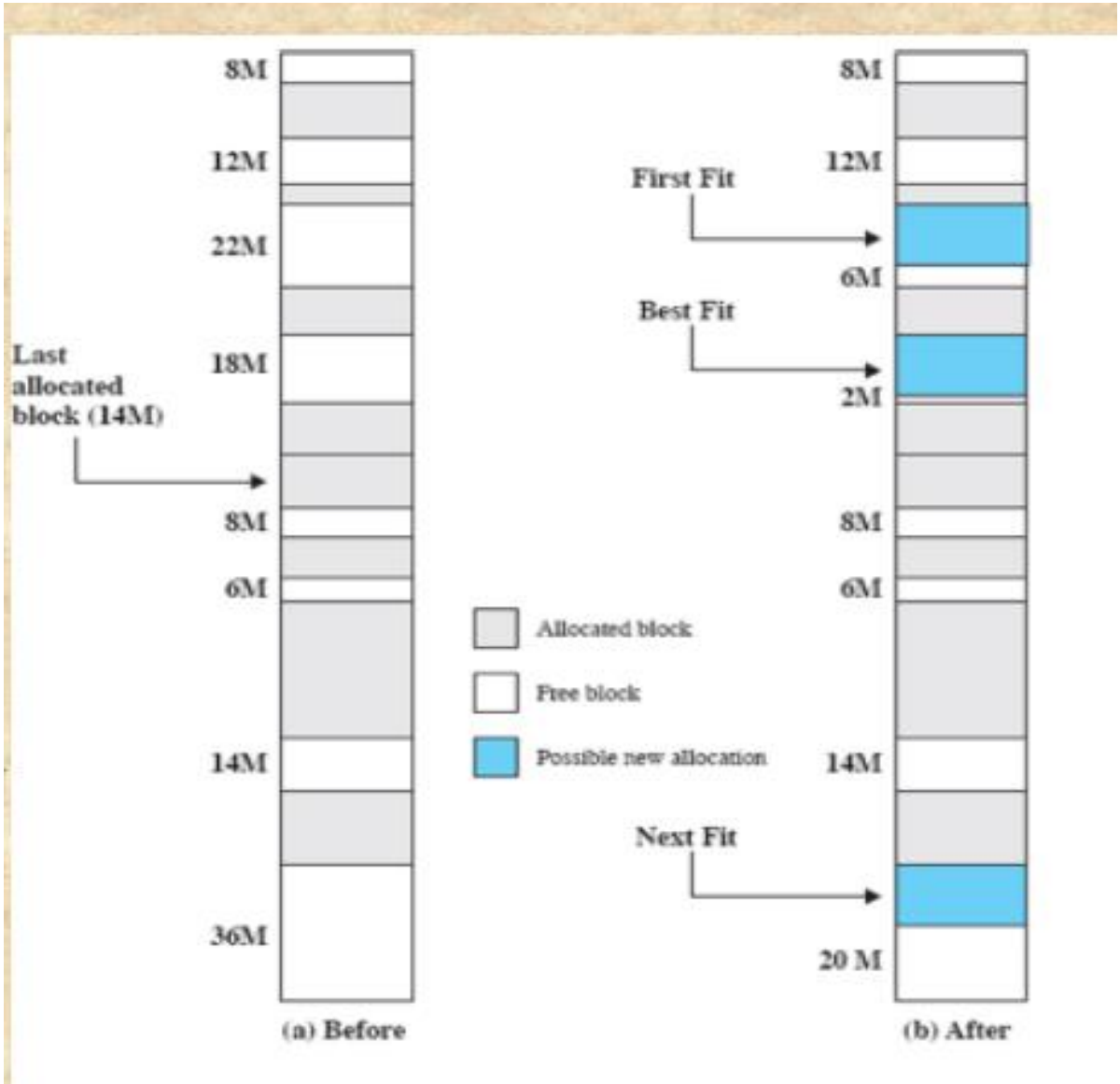
Dynamic Storage Allocation

- **Suppose a new process requests 16 MB then which partition each would *best fit*, *next-fit* and *first-fit* algorithm pick?**



Dynamic Storage Allocation

- **16 MB: Which partition each would best fit, next-fit and first-fit algorithm pick?**



Fragmentation

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
  - Example: a process requests 18,462 bytes and is allocated memory hole of 18,464 bytes; we are then left with a hole of 2 bytes
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - Free memory space is broken into pieces as processes are loaded and removed from memory
  - Statistical analysis of first-fit reveals that given N allocated blocks, 0.5N blocks will be lost to fragmentation

External Fragmentation

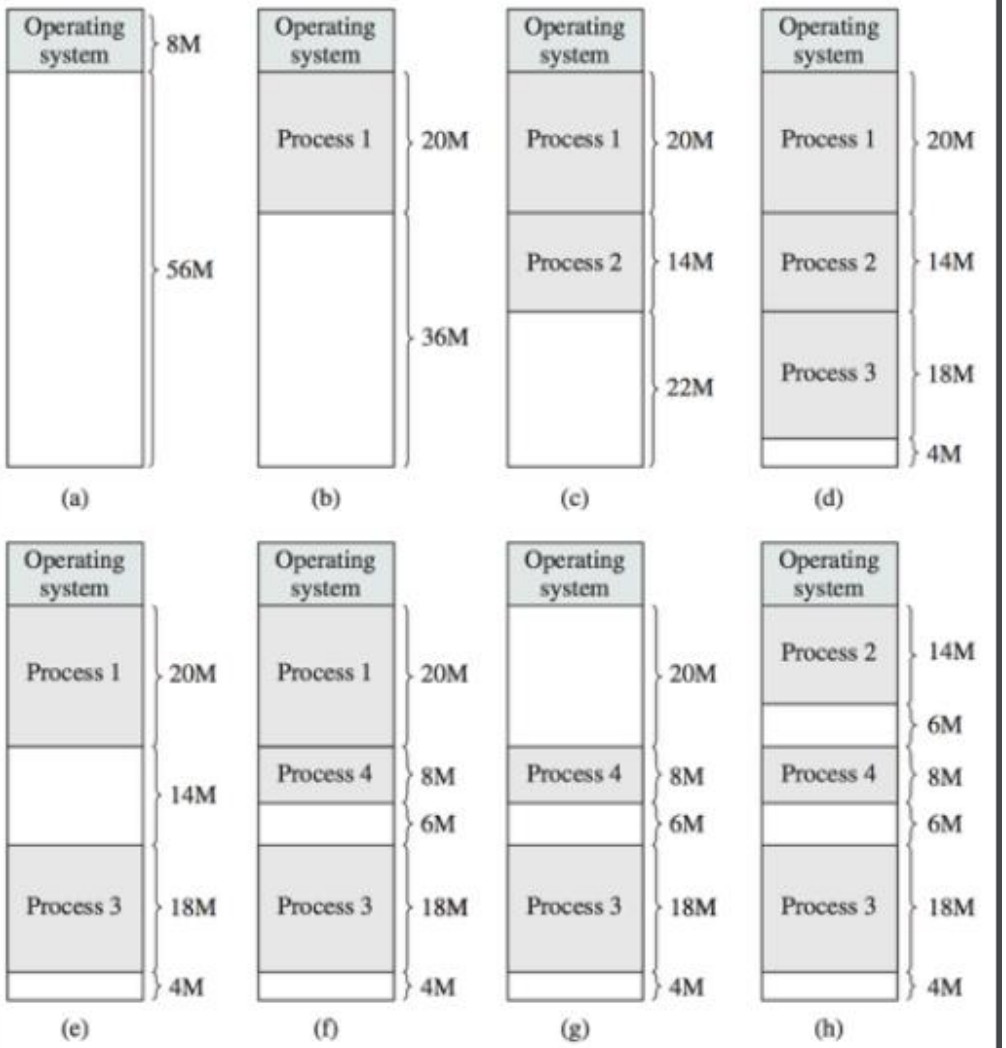
- In the end, a new process larger than 6MB and less than 16 cannot be allocated memory

Effect of Dynamic Partitioning

Process 1 needs 20M of memory, 2 needs 14, 3 needs 18, and 4 needs 8M.

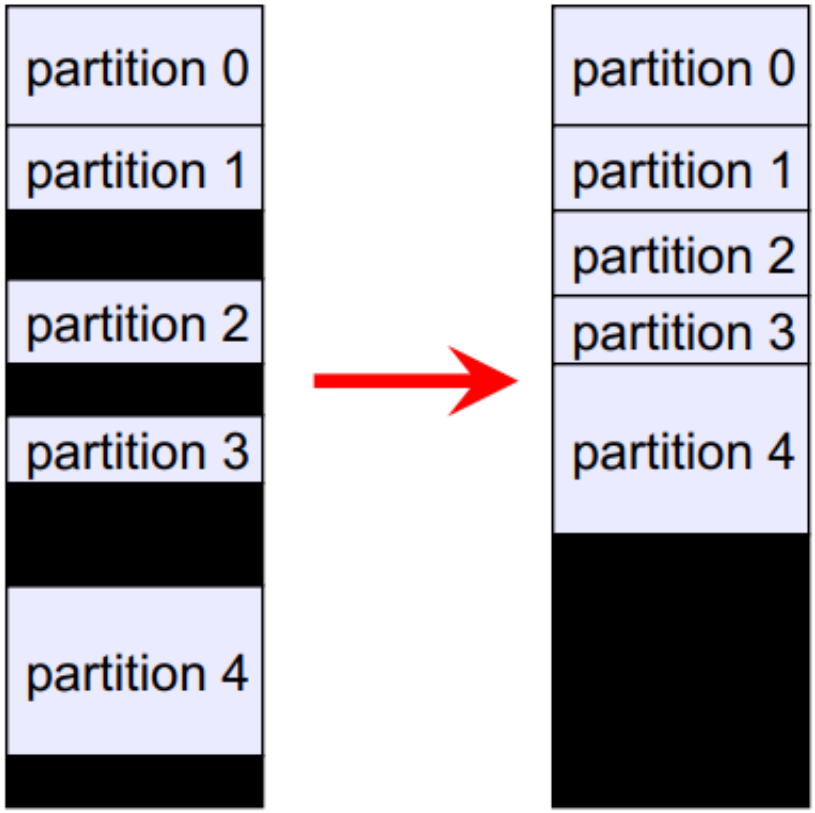
External Fragmentation:

A lot of small hols at the end!



Dealing with Fragmentation

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time



Dealing with Fragmentation

- Other solutions to external fragmentation problems: **Segmentation and Paging**
  - Permit the logical address space of each process to be non-contiguous
    - Process can be allocated physical memory wherever it is available

Chapter 8: Memory Management

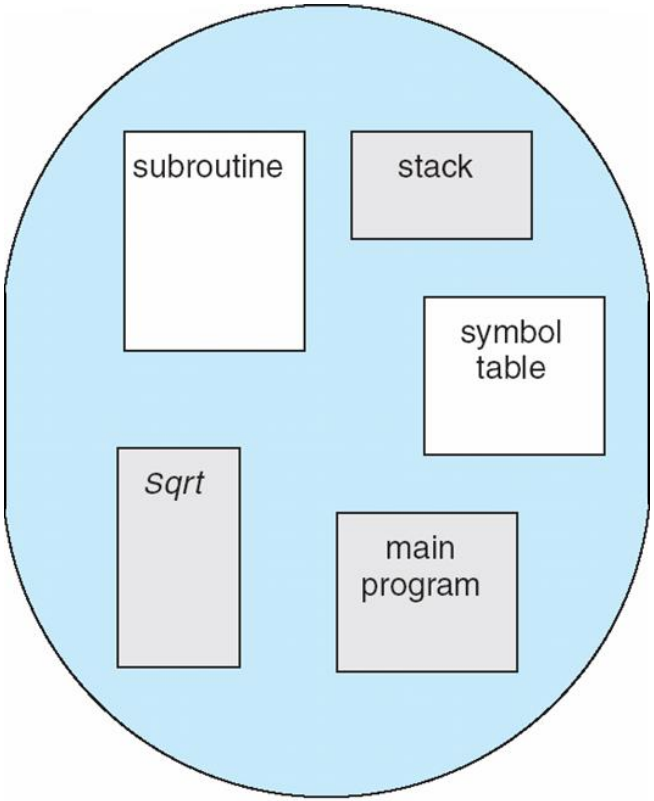
- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

Segmentation

- Segmentation **eliminates** the need for continuous addresses

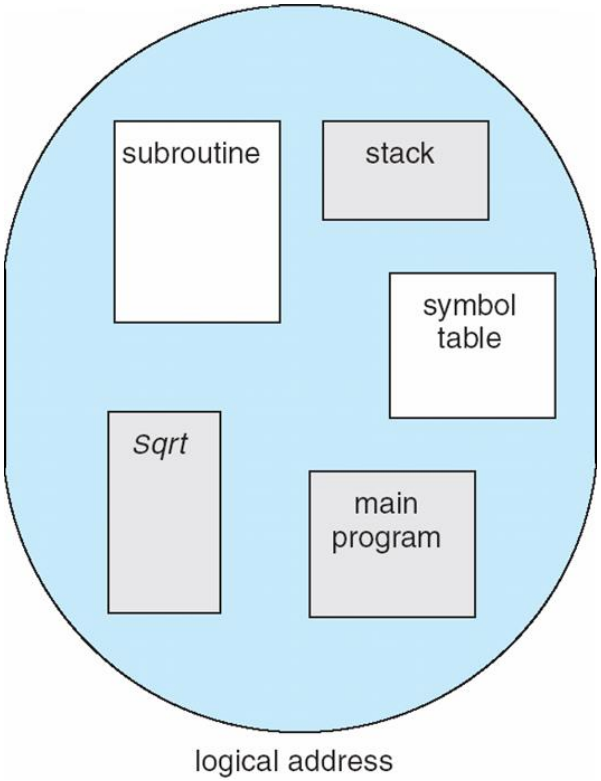
- A program is a collection of segments
  - A segment is a logical unit such as:

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



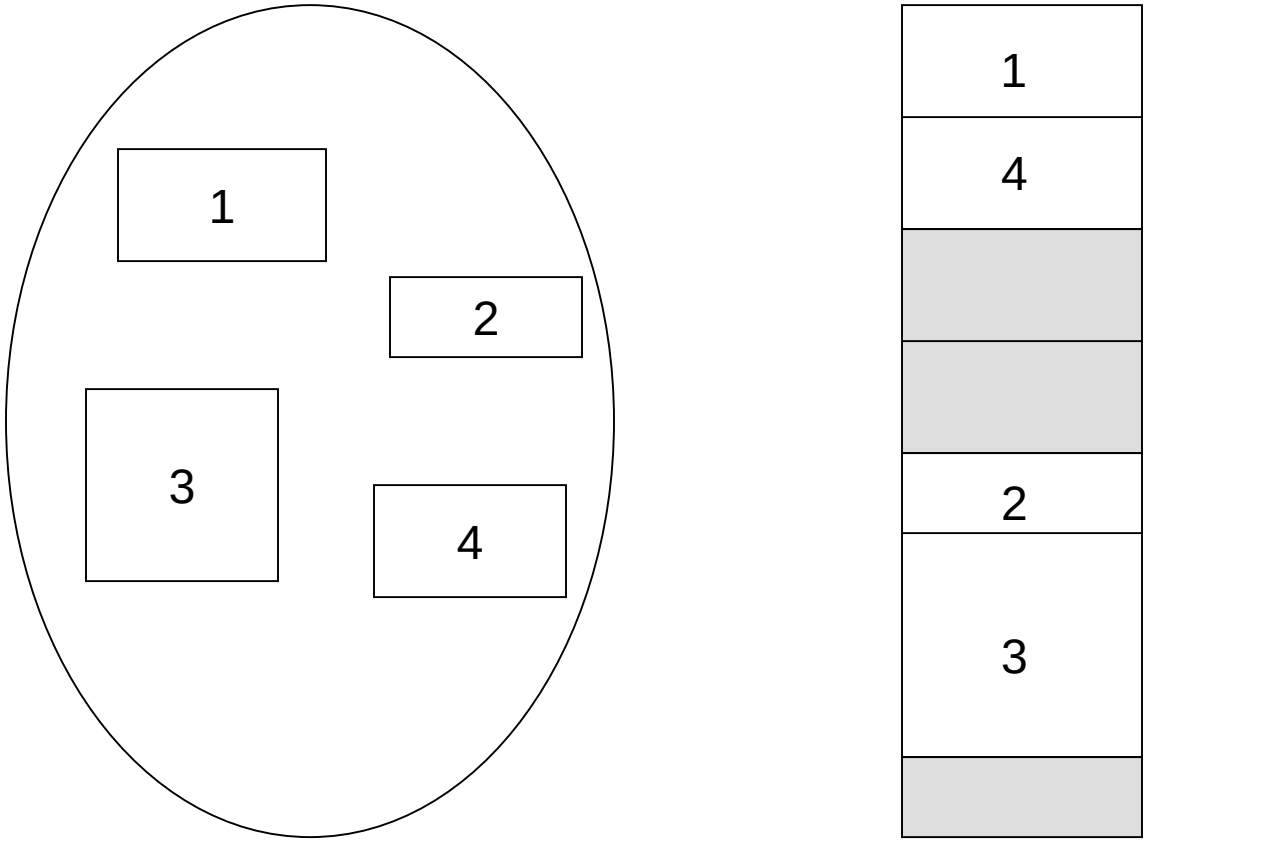
Segmentation: Addressing Mechanism

- Addressing consists of two parts:
  - segment number
  - an offset
- Because of the use of unequal-size segments, segmentation is similar to *dynamic partitioning*
- A natural extension of variable-sized partitions
  - variable-sized partition = 1 segment/process
  - segmentation = many segments/process



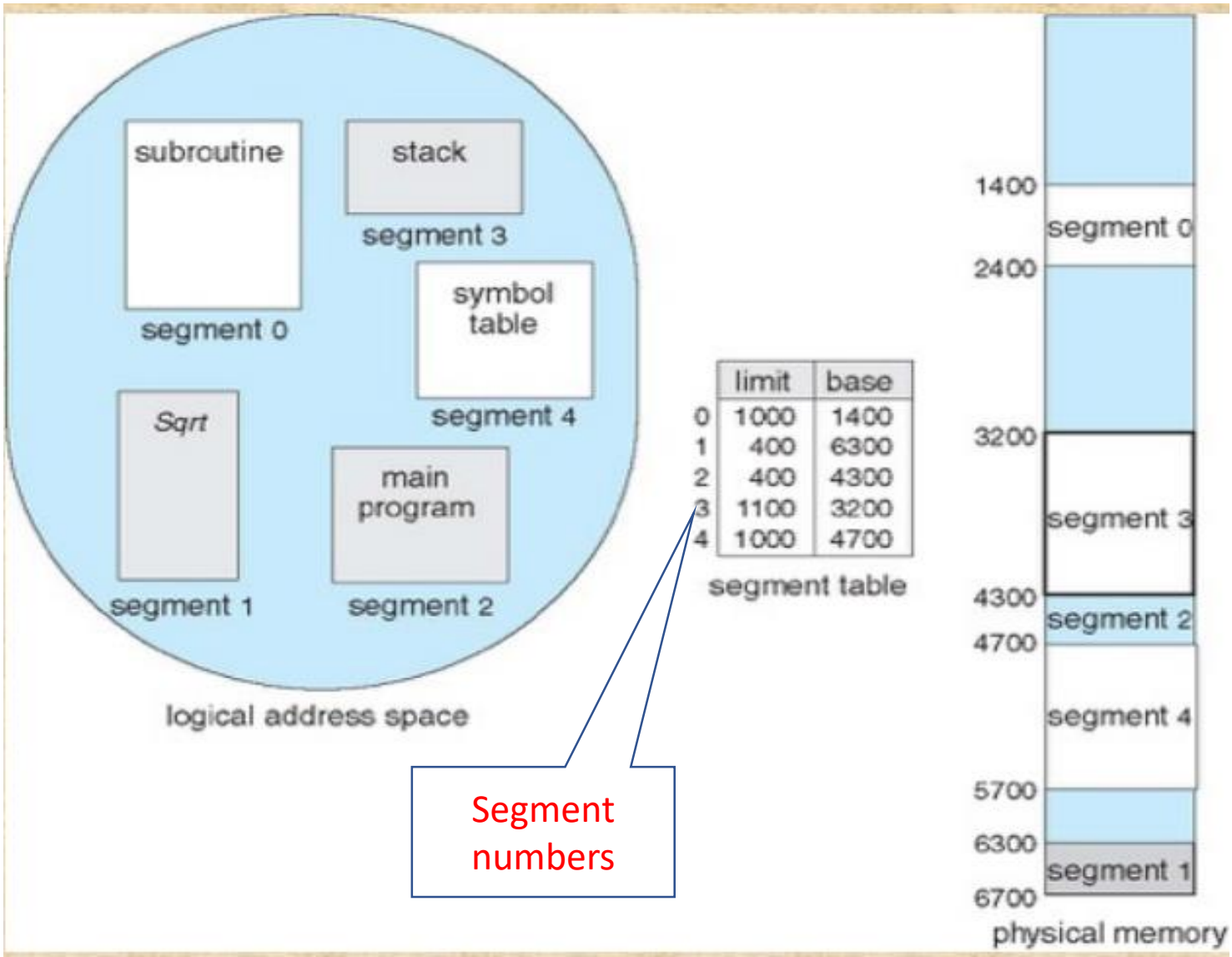
Logical View of Segmentation

- Segment 1, Segment 2… etc. are basically main program, stack, subroutine etc.



Logical View of Segmentation

- The value in the base register represent the real address of the corresponding segment in the physical memory.



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 is kept in memory:

- **Segment-table base register (STBR)** points to the segment table's location in memory

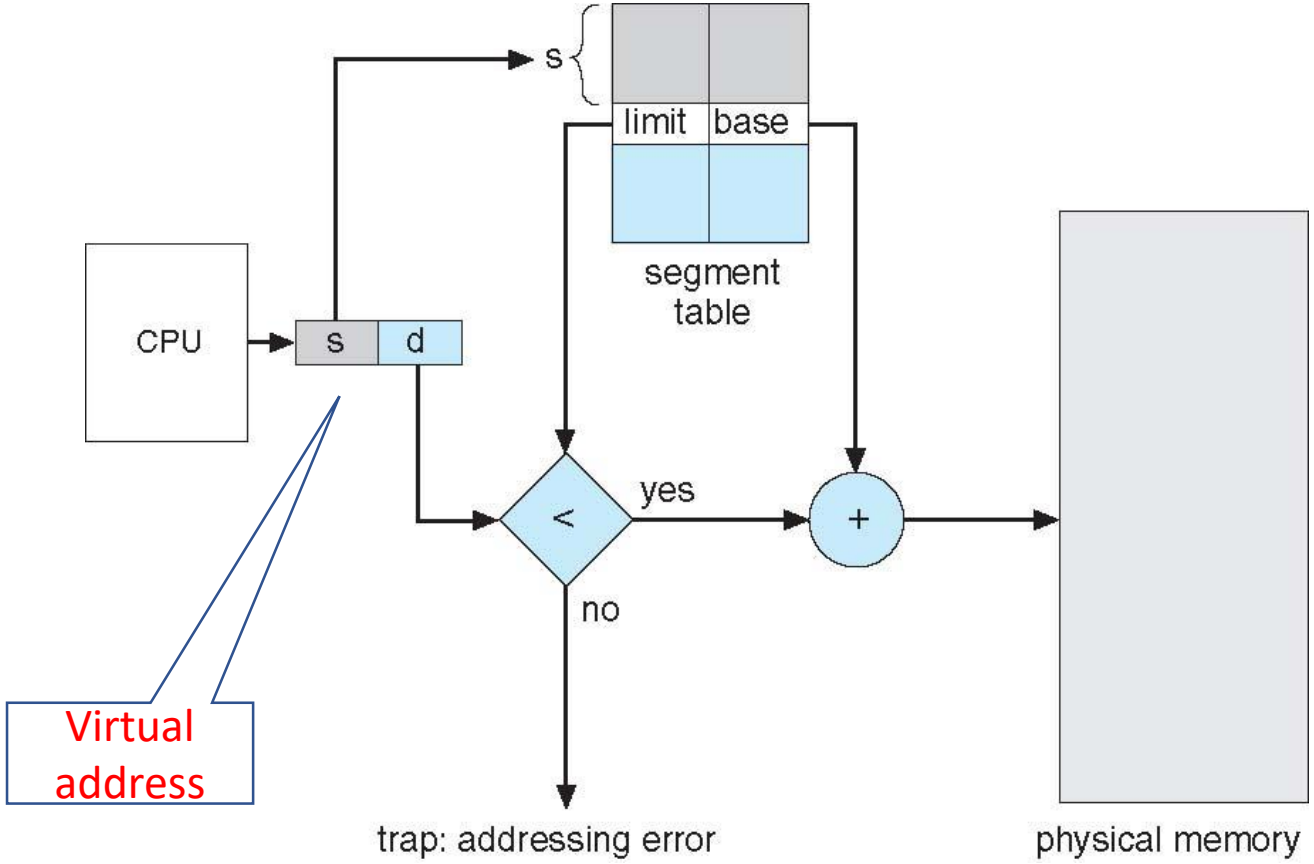
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Hardware

- Logical address consists of a two tuple:
  <**segment-number, offset**>,

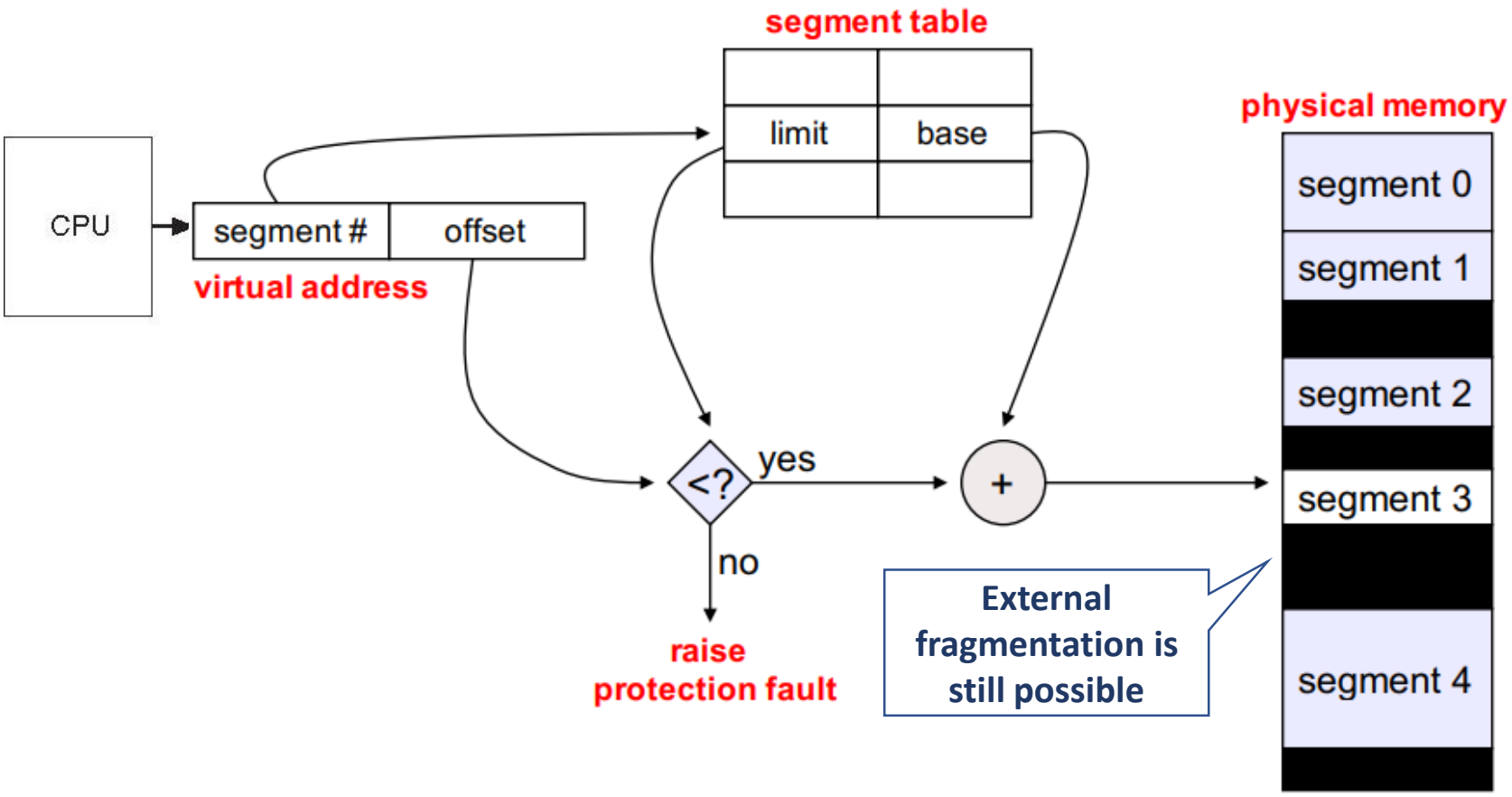
- Each logical address <**s, d**> is compared with the segment table



Segmentation Hardware: Segment Lookups

- Logical address consists of a two tuple:
  <**segment-number, offset**>,

- Each logical address <**s, d**> is compared with the segment table



Chapter 8: Memory Management

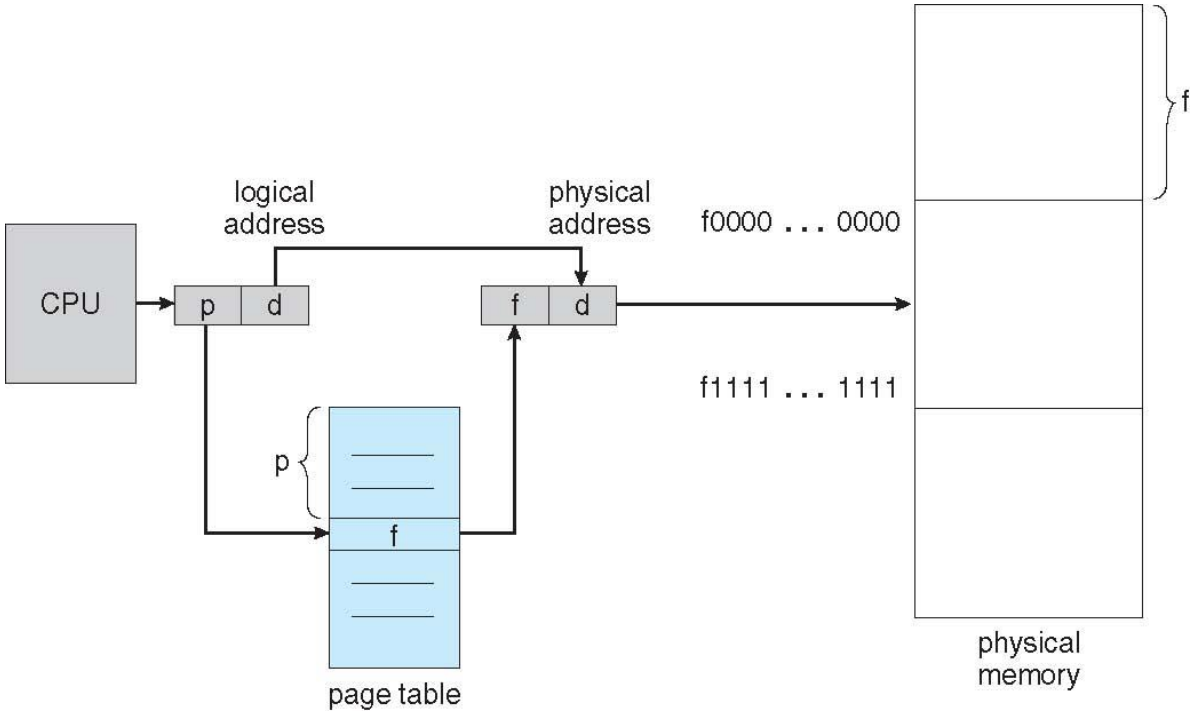
- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- **Paging**
- Structure of the Page Table

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids **external fragmentation** and the need for compaction
  - Avoids problem of varying sized memory chunks
- **Paging**
- Divide physical memory into fixed-sized blocks called **frames**
    - Size is power of 2, between 512 bytes and 16 Mbytes
  - Divide logical address space into blocks of same size as the frames, called **pages**
  - The page size (like the frame size) is defined by the hardware; see Slide-29
  - Divide backing store into [clusters of] blocks of same size as the frames
- 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
- Limitation: Still have Internal fragmentation

Paging Hardware

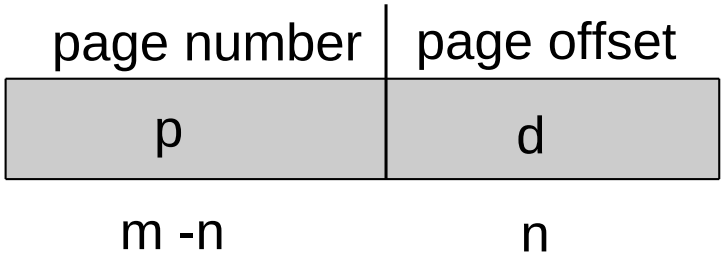
- Logical address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table**
    - **Page Table** contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is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



Address Translation Scheme

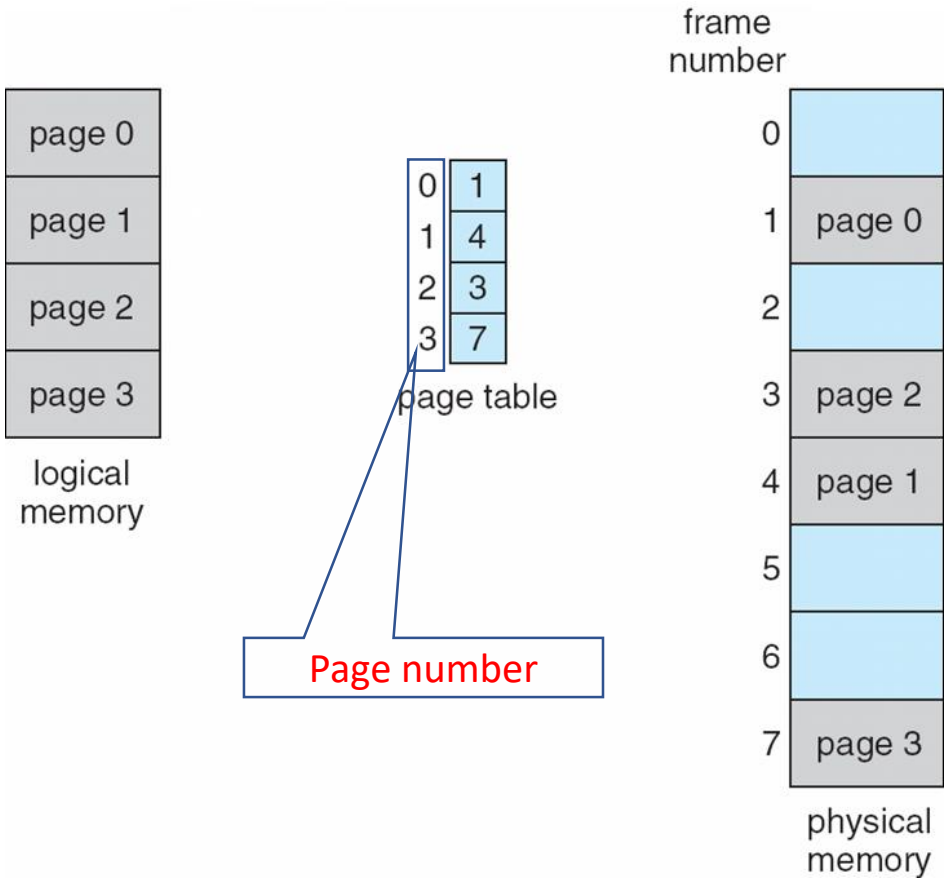
- Logical address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table**
    - **Page Table** contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit
  - If the logical address space is 2^m and the page size is 2^n
    - The binary representation of the logical address has **m bits**, such that
      - The **m – n** leftmost bits designate the page number **p**
        - **p** is index into the page table
      - The rightmost **n** bits designate the page offset **d**
        - **d** is displacement within the page

  - For given logical address space 2^m and page size 2^n



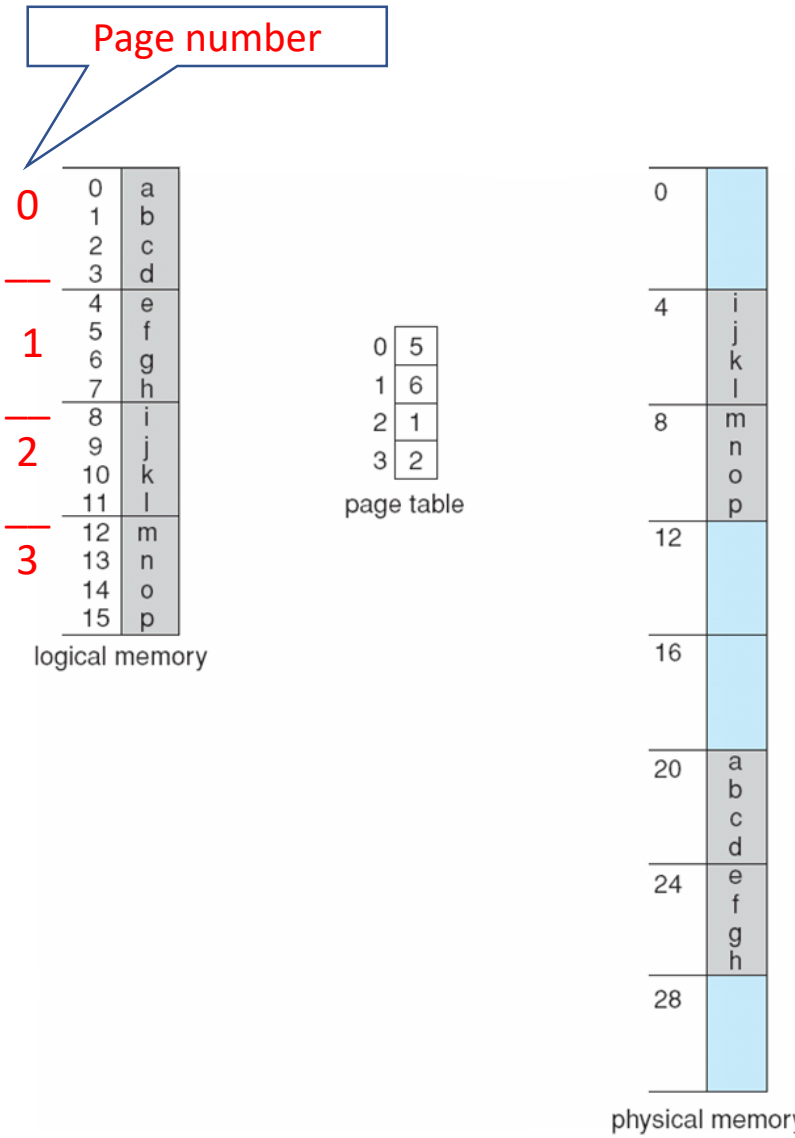
Basic Paging Model of Logical and Physical Memory

- Page table maps virtual page number to physical frame number.
- A process's logical address space consists of 4 pages, i.e., m = 4.

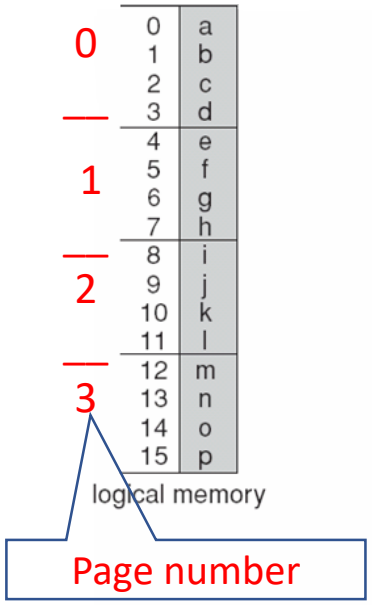


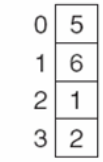
Paging Example (1)

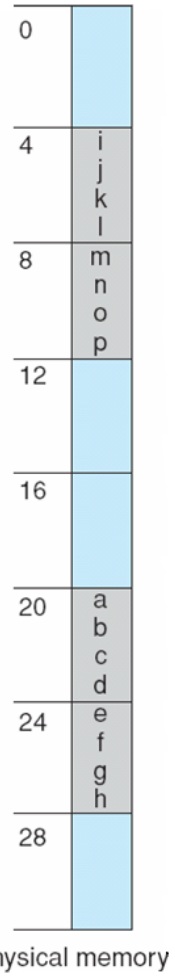
- Example: *m* = 4 and *n* = 2
- What is logical address space of the process?
- What is Page size?
- What is the total number of pages?
- Physical memory has 5 bits.
- Now if logical address is 5→ **0101** (f)
  - What is Physical address ?
- Similarly if logical address is 13→ 1101 (n)
  - What is Physical address?



Paging Example (1)



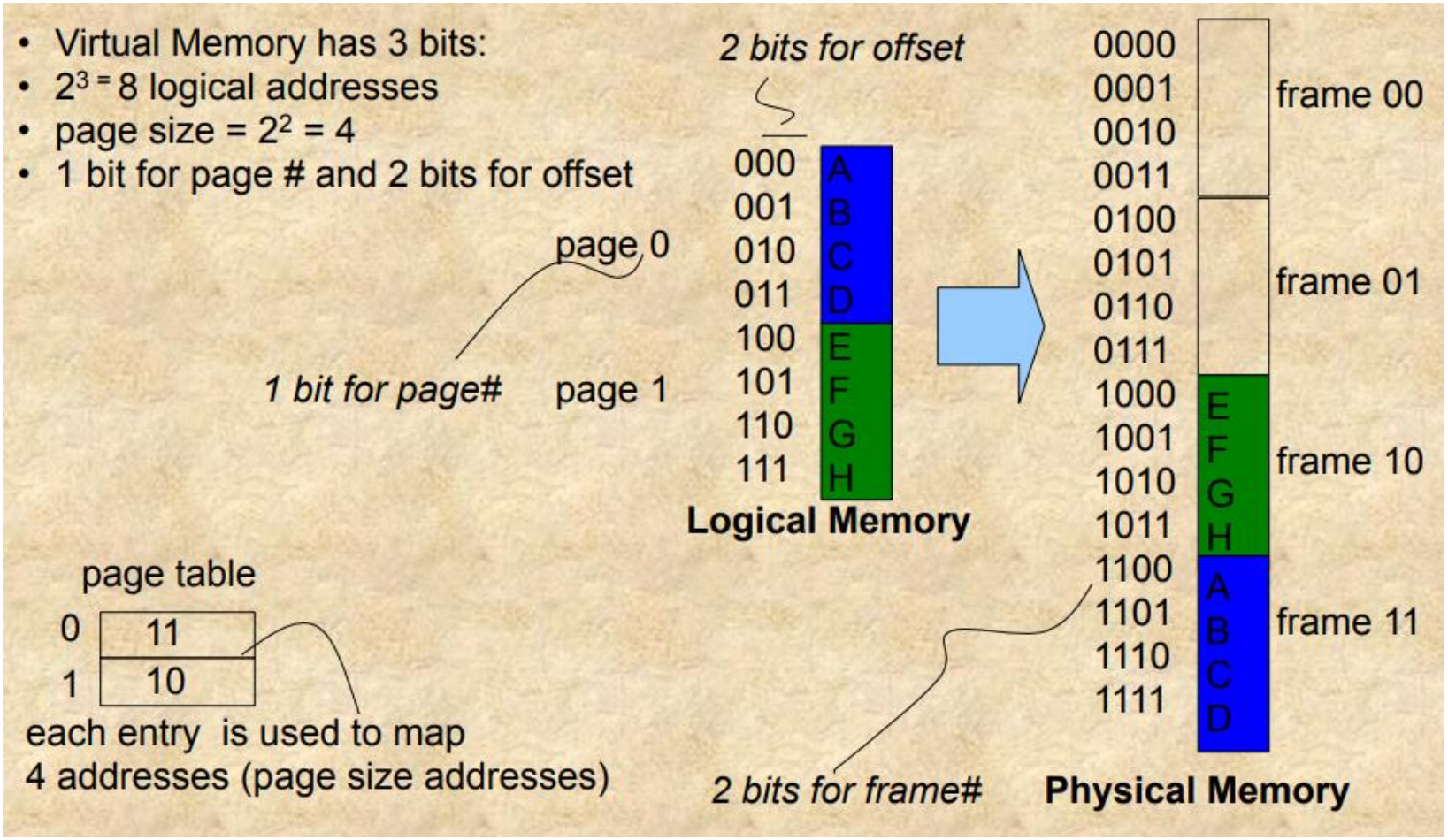




- Example: *n* = 2 and *m* = 4
- What is logical address space?
  - $2^m = 2^4 = 16$
- What is Logical Page size?
  - $2^n = 2^2 = 4$
- What is the total number of pages?
  - Logical address space/page size
  - Or m-n = 4-2 = 2 bits→2^2 pages

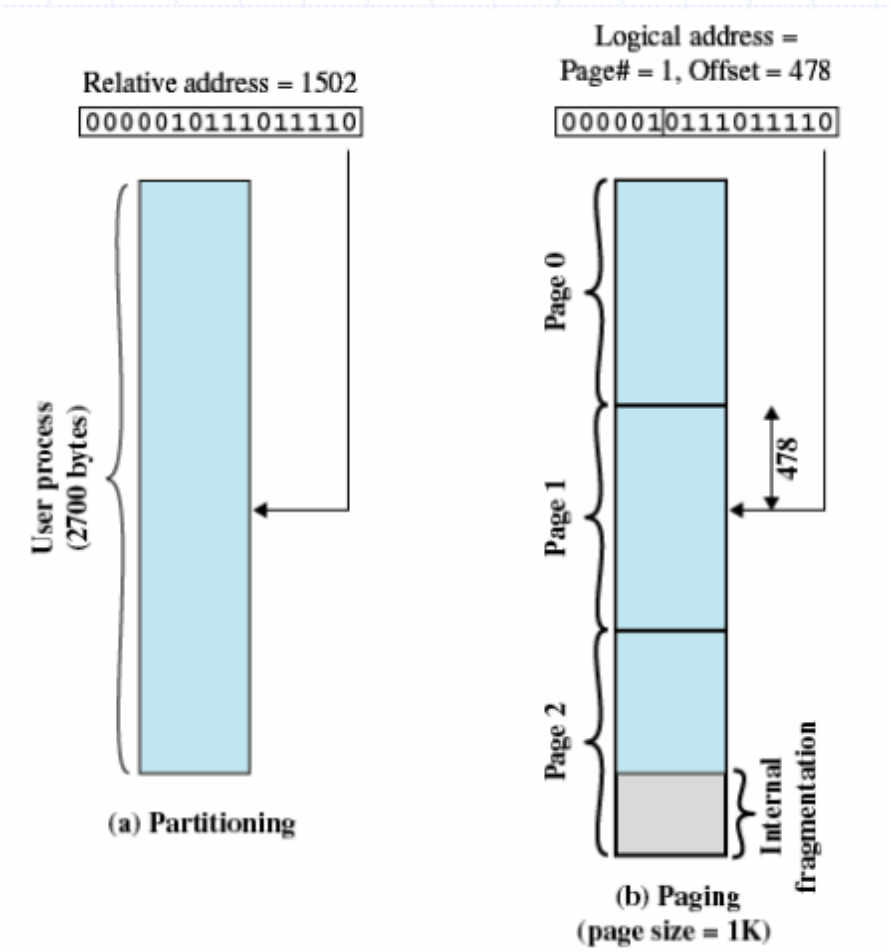
- Physical memory has 5 bits → so physical address space = 2^5
  - 3 bits for number of pages (total 8 pages) and remaining 2 bits for page size.
- Now if logical address is 5→ 0101 (f)
  - From page table page#1 corresponds to frame#6. So Physical address is 11001
- Similarly if logical address is 13→ 1101 (n)
  - From page table page#3 corresponds to framee#2. Physical address is 01001

Paging Example (2)

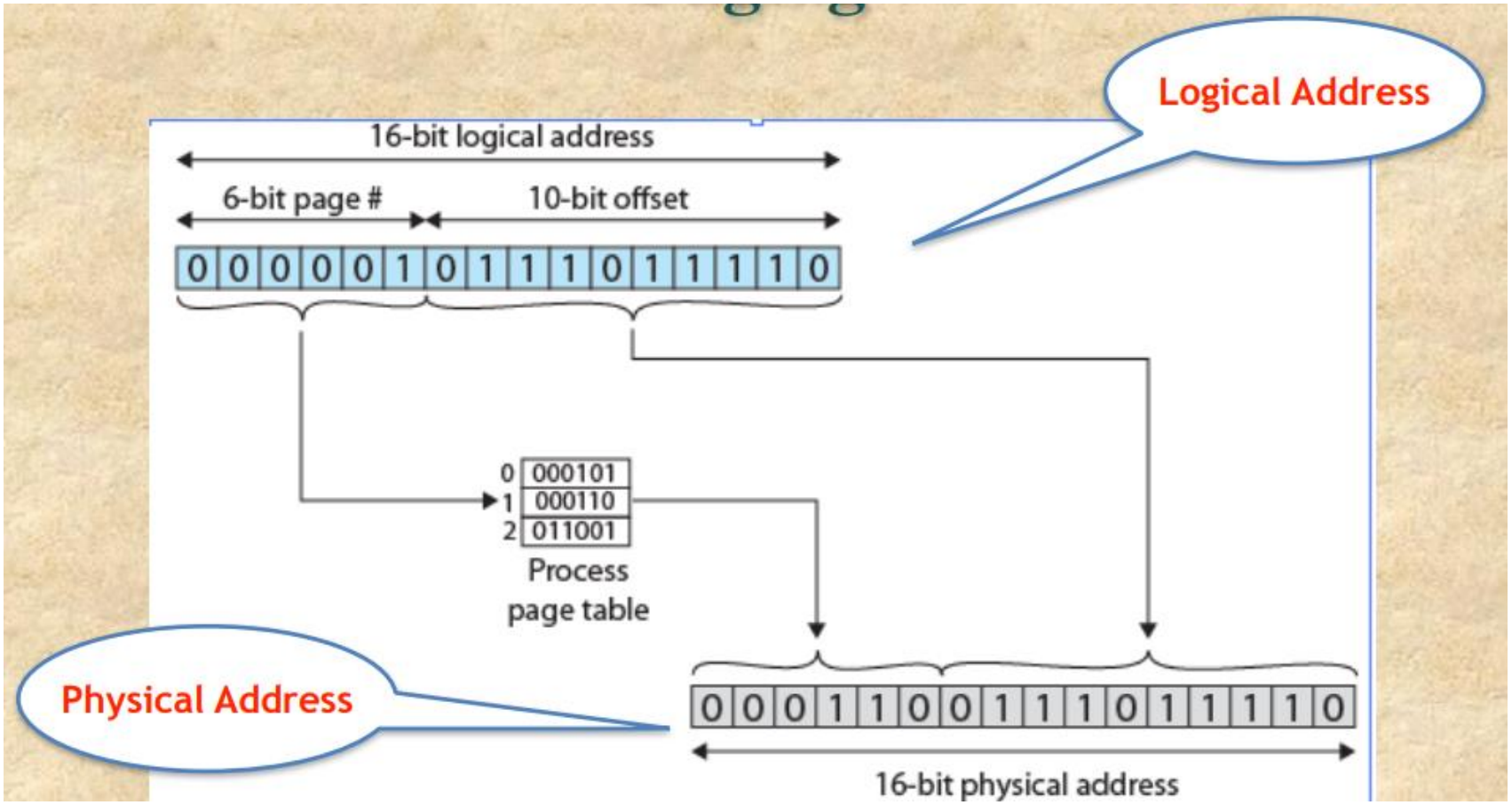


Logical to Physical address Translation (1/2)

- For a 16-bit (m=16) address the logical address space is 2^{16} bytes
- Page size (n=10) = 1KB = 2^{10} Bytes
- Maximum 64 (2^6) pages of 1K or 1024 bytes each
- What is a Logical address of relative 1502?



Logical to Physical address Translation (2/2)



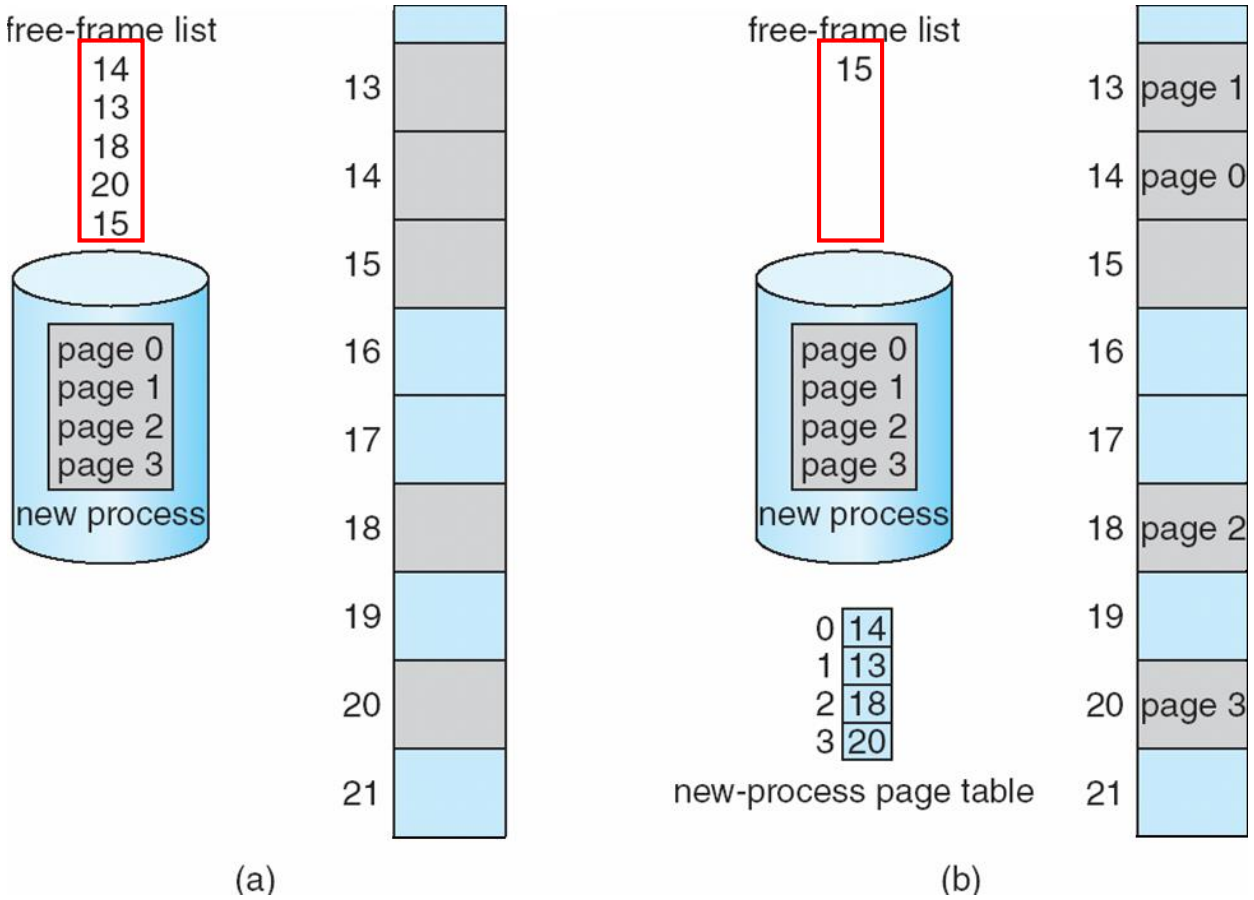
Paging (Cont.)

- There is no external fragmentation when using paging scheme
- **Internal fragmentation is possible**. Calculating internal fragmentation
  - If Page size = 2,048 bytes and
  - Process size = 72,766 bytes
    - Process has 35 pages + 1,086 bytes, thus 36 frames required
    - Internal fragmentation of 2,048 - 1,086 = 962 bytes unused memory
  - Worst case fragmentation is when 1 frame contains only 1 byte used memory
  - On average fragmentation = 1/2 page size per process size

Paging (Cont.)

- So Are small frame sizes desirable?
  - Not necessarily; each page-table entry takes memory to track (overhead)
  - Large frame sizes better when transferring data to/from disk; efficient disk I/O

Free Frames-Before and After Allocation



Before allocation

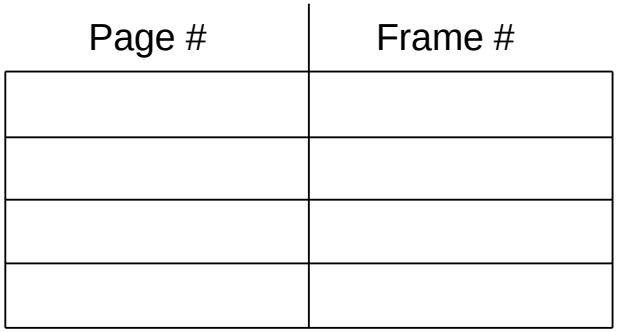
After allocation

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** (**PTLR**) indicates size of the page table
  - Both, PTBR and PTLR are also stored in the process's PCB
- In this scheme every data/instruction access requires two memory accesses
  - First: access the page table using PTBR value to retrieve its frame number
  - Second: access the actual memory location given the frame number
    - This is a serious time overhead that needs to reduced
  - 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**)

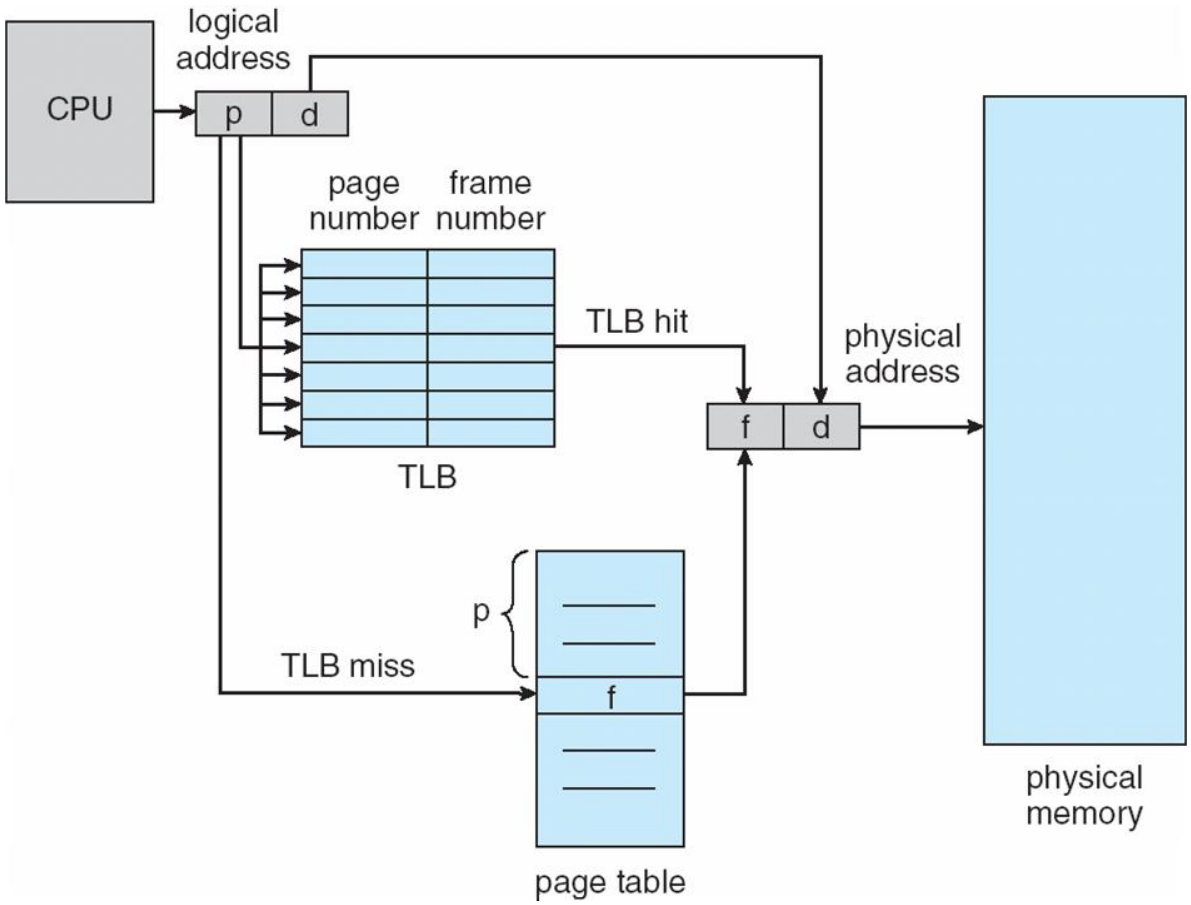
Associative Memory

- Associative memory



- Address translation (p, d)
  - If p is in associative register, get frame # out directly
  - Otherwise get frame # from page table in main memory

Paging Hardware With Translate Lookaside Buffer (TLB)



Paging hardware with TLB

❑ MMU in CPU first looks in TLB's to find a match for a given logical address

– **if match found**, then quickly call main memory with physical address frame f (plus offset d)

- this is called a **TLB hit**
- TLB as implemented in hardware does a fast parallel match of the input page to all stored values in the cache - about 10% overhead in speed

– **if no match found**, then

1. go through regular two-step lookup procedure: go to main memory to find page table and index into it to retrieve frame #f, then retrieve what's stored at address <f,d> in physical memory

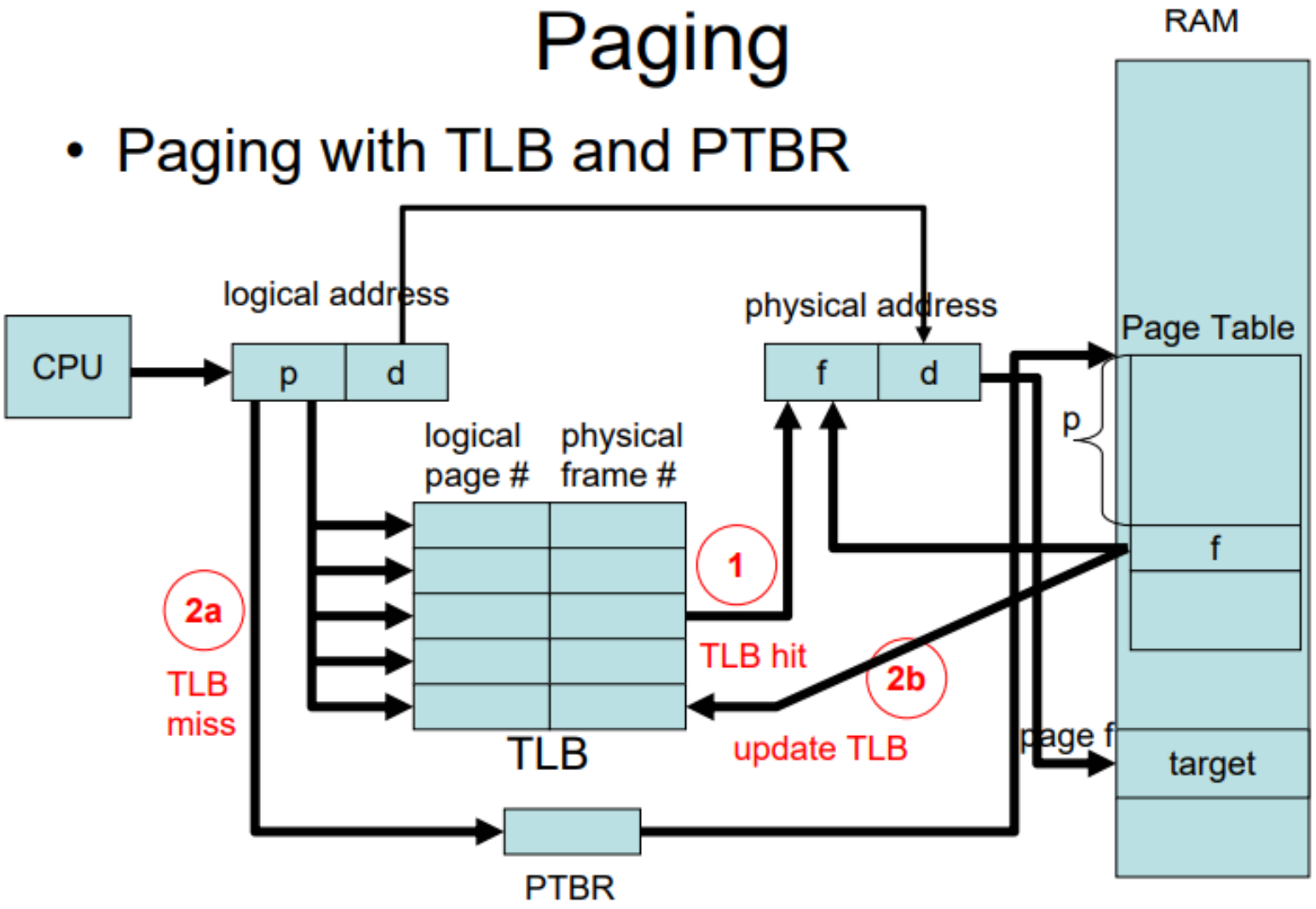
2. Update TLB cache with the new entry from the page table

– if cache full, then implement a cache replacement strategy, e.g. Least Recently Used (LRU) - we'll see this later

- This is called a **TLB miss**

• Goal is to maximize TLB hits and minimize TLB misses

Paging Hardware with TLB



Effective Access Time (EAT)

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

Consider α = 80%, and miss ratio = 1- α, ε = 20ns for TLB search, 100ns for memory access

- EAT = 0.80 x 20 + 0.20 x 100 = 36ns

Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access

- **EAT = ?**

Effective Access Time (EAT)

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

Consider α = 80%, and miss ratio = 1- α, ε = 20ns for TLB search, 100ns for memory access

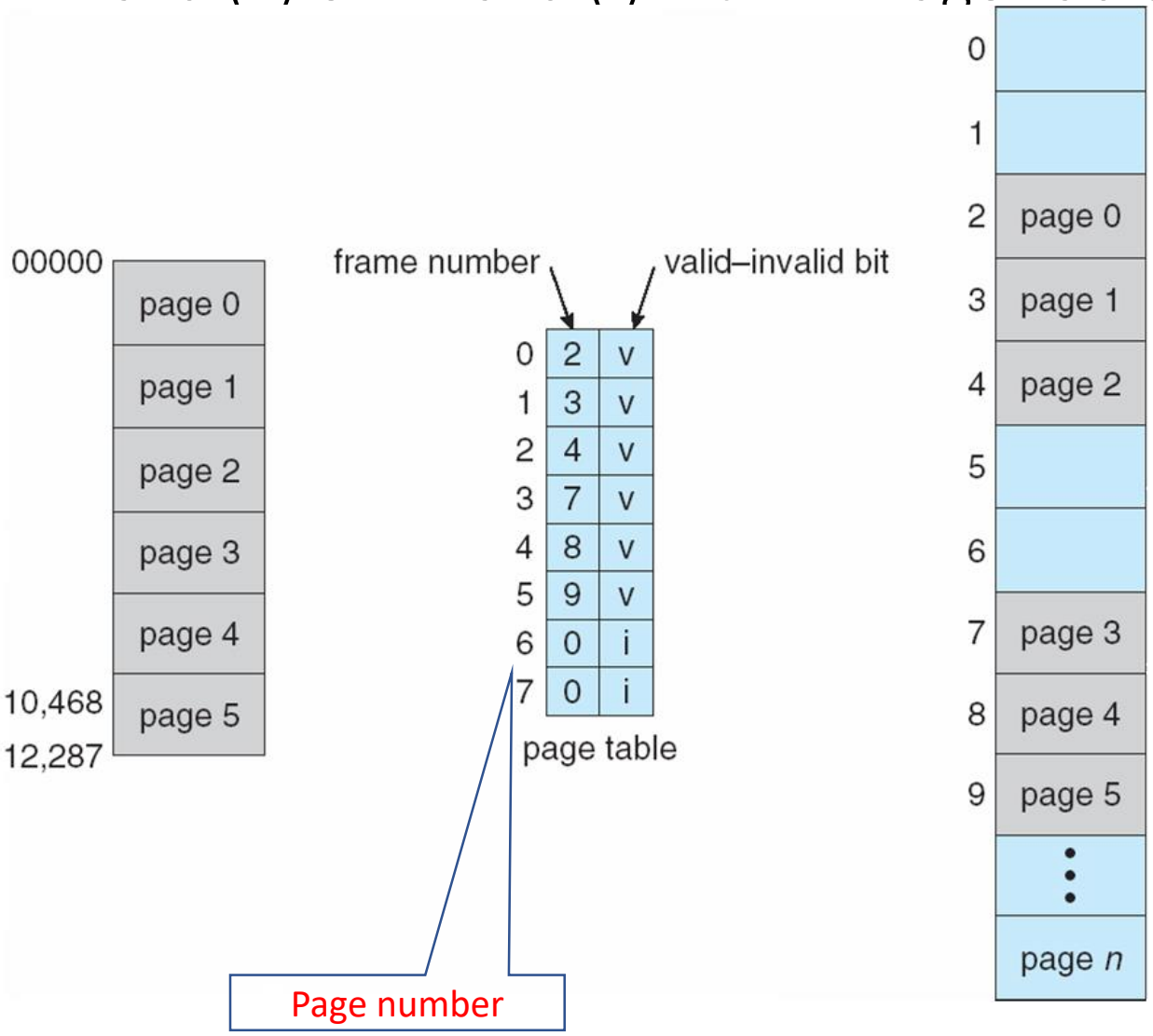
  - EAT = 0.80 x 20 + 0.20 x 100 = 36ns

- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - **EAT = 0.99 x 20 + 0.01 x 100 = 20.8ns**

Memory Protection

- Memory protection implemented
  - by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page ***execute-only***, and so on
- **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
  - Further details in chapter#9 (virtual memory)
- Any violations result in a trap to the kernel

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

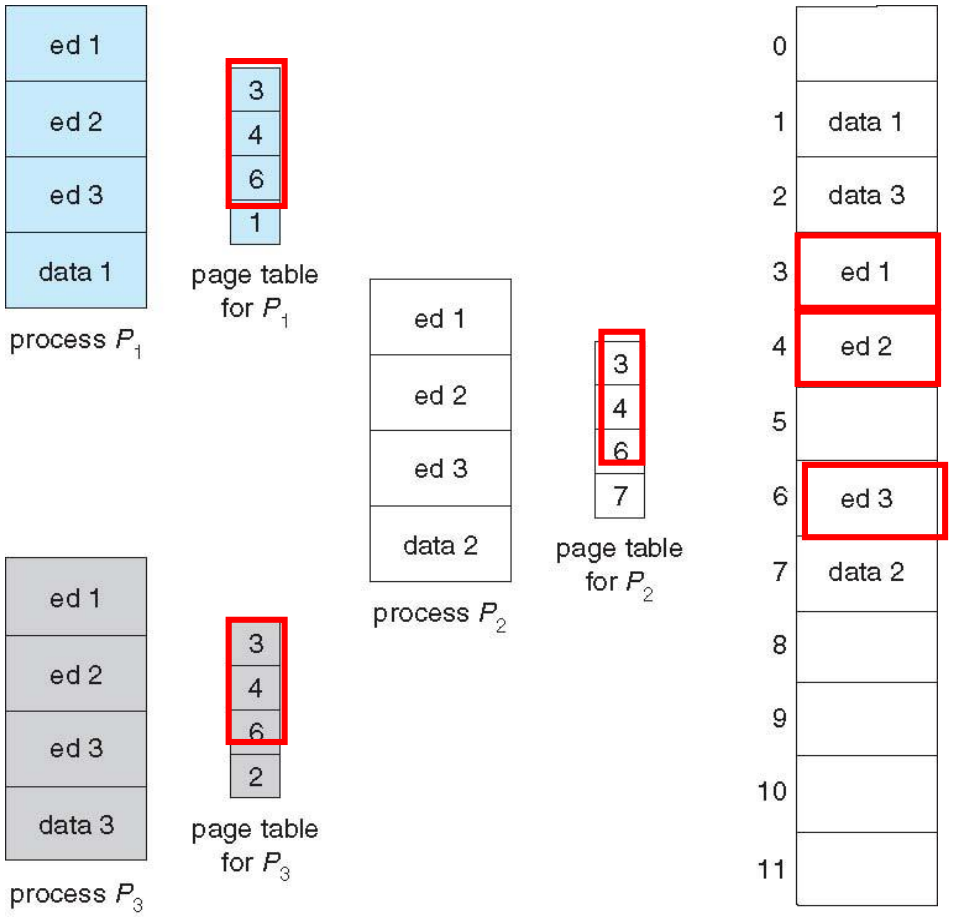


Shared Pages

- **Shared code**
  - One copy of read-only code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example

- There are three processes. Each process has its own private data.
- However, three processes share three read-only pages namely ed 1, ed 2 and ed 3, loaded into frame#3, 4, and 6 respectively.



Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- **Structure of the Page Table**

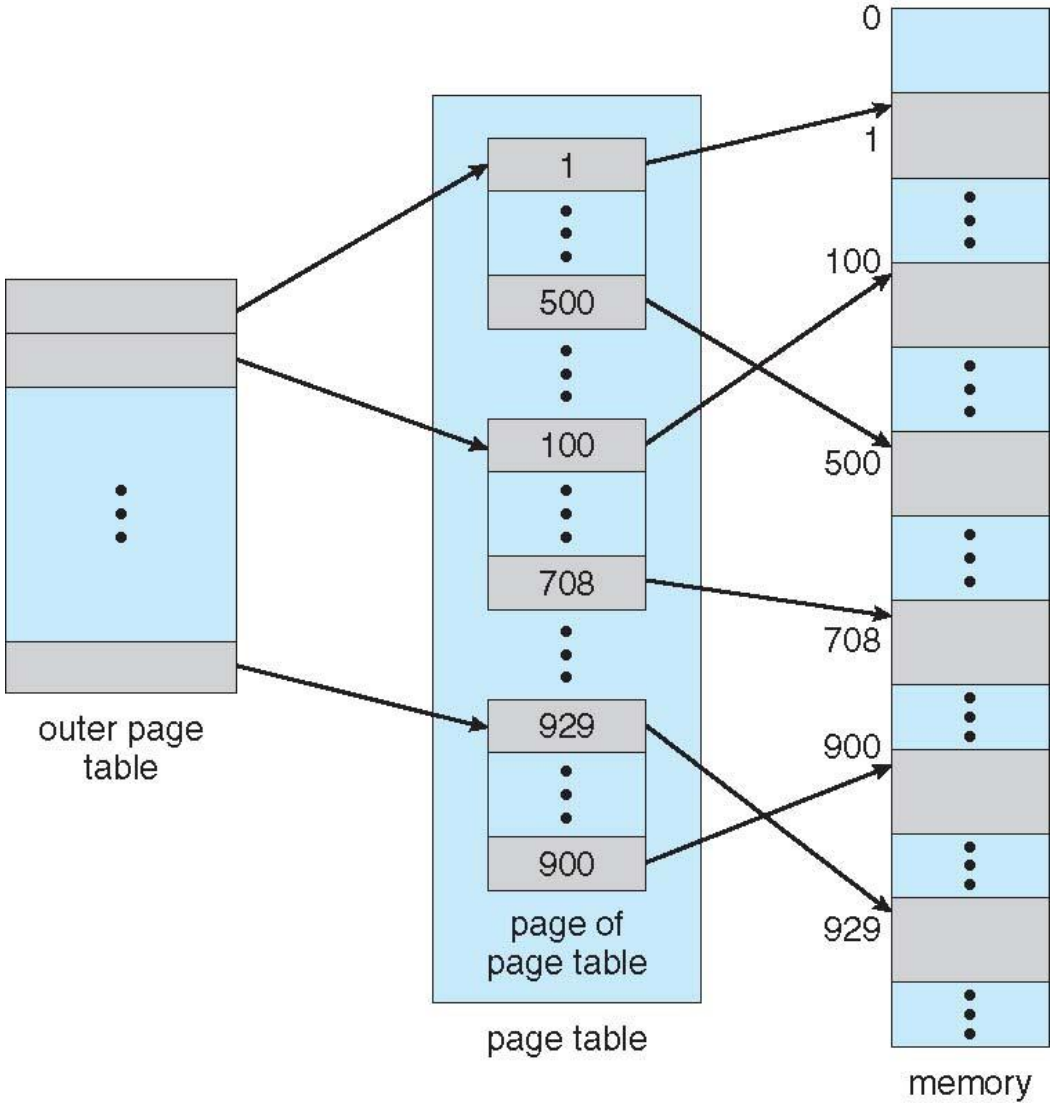
Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - Assume each entry is 4 bytes → 4 MB of physical memory for page table alone
    - That amount of memory used to cost a lot if there are a lot of processes
    - Don't want to allocate that contiguously in main memory
- **Method#1:** Hierarchical Paging
- **Method#2:** Hashed Page Tables
- **Method#3:** Inverted Page Tables

Method#1: Hierarchical Page Tables

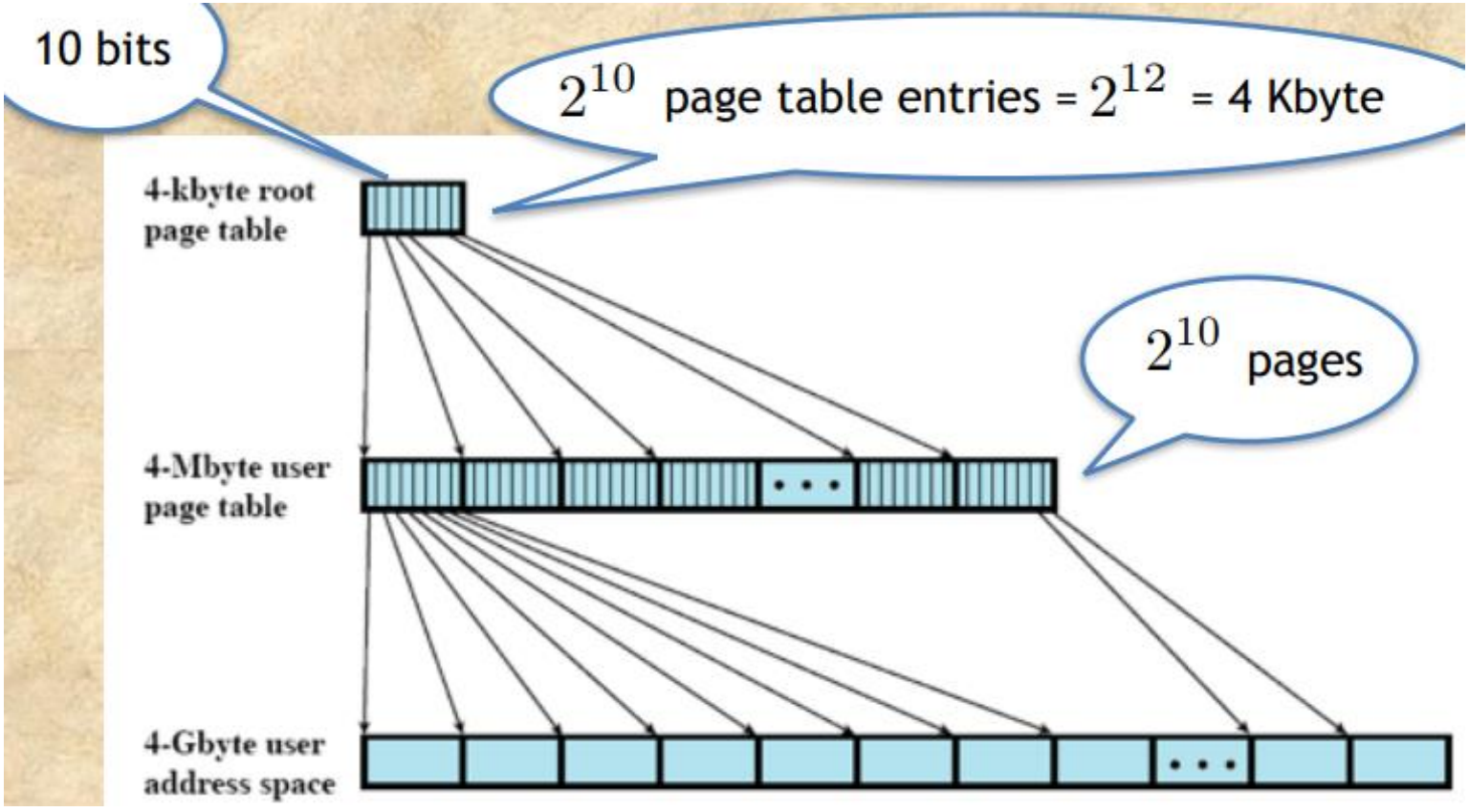
- **Paging Principle**:
  - Break up the logical address space into **multiple page tables**
  - Subdividing a process into pages helped to fit a process into memory by allowing it to be scattered in non-contiguous pieces of memory, thereby solving the external fragmentation problem
- so reapply that principle here to fit the **page table** into memory, allowing the page table to be scattered non-contiguously in memory
- A simple technique is a two-level page table
- We then **page the page table**

Method#2: Two-Level Page-Table Scheme



Two-Level Hierarchical Page Table

- Logical address space is 2^{32} and each page size is 2^{12}.
- Inner page table has 1 million entries therefore its size is 4Mbytes. Each page of the inner page table is also 4kbytes. 4MB/4KB→ **2^{10}** = 1024 pages.
- Now we need an outer page table, in which each entry points to a different page of the **inner page table.**
- In **outer page table** there are **2^{10}** entries each of them is 4 bytes, therefore, total outer page table is also 4kbytes.

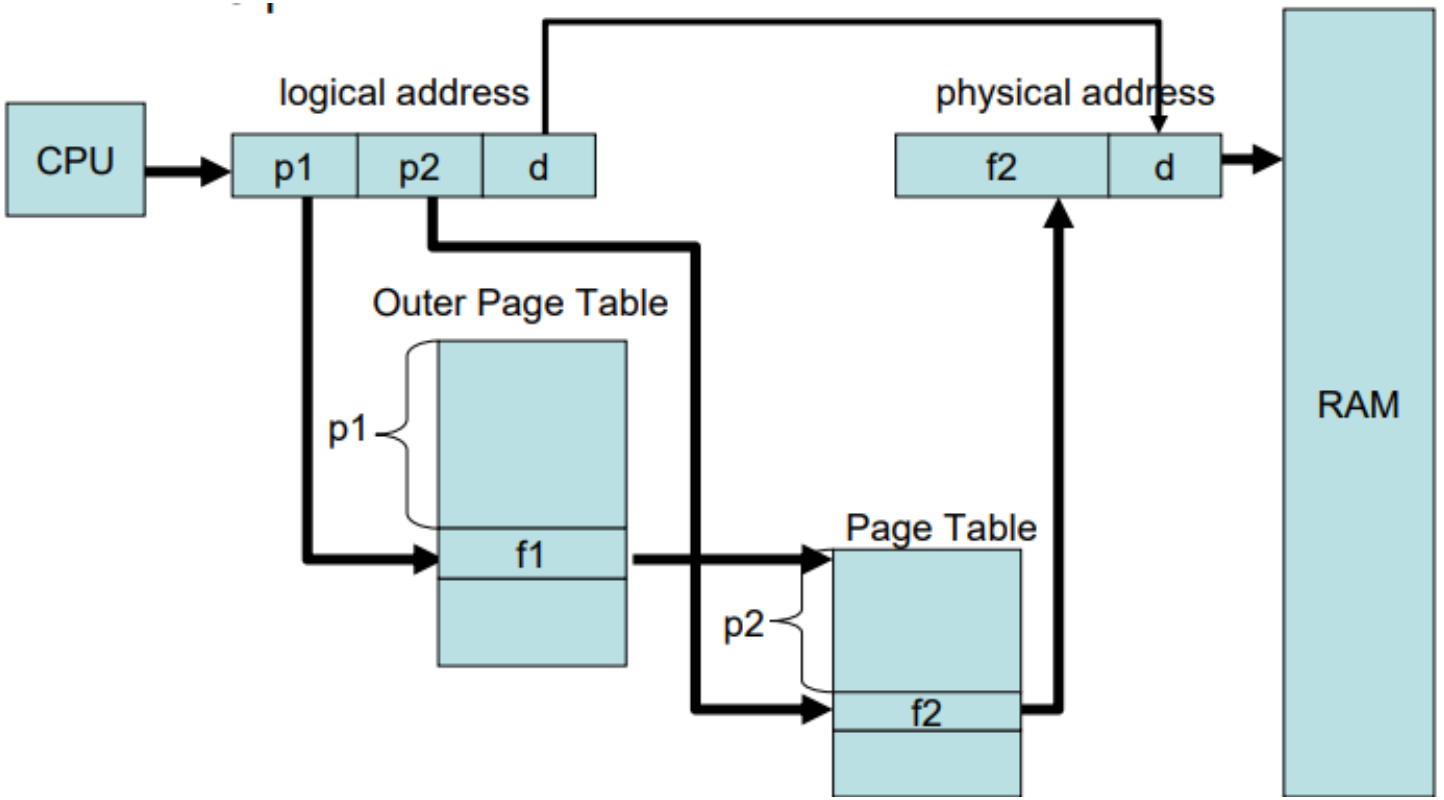


Two-Level Paging Example

- A logical address (on 32-bit machine with 4KB page size (or 2^{12})) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Assuming each page of the ***inner page table*** is also 4KB, then each page holds 2^12 bytes / 4 bytes per entry = 2^10 entries per POPT
- Since there are 2^20 total entries in the page table, there must be 2^10 pages in the page table. → requires 2^{10} entries in outer page table (POPT)
- This leaves us with 10 bits left over for the ***outer page table***
- Hence we have:
  - a 10-bit page number (P1)
  - a 10-bit page offset (P2)

Hierarchical Paging

- Hierarchical (2-level) paging divides the logical address into 3 parts

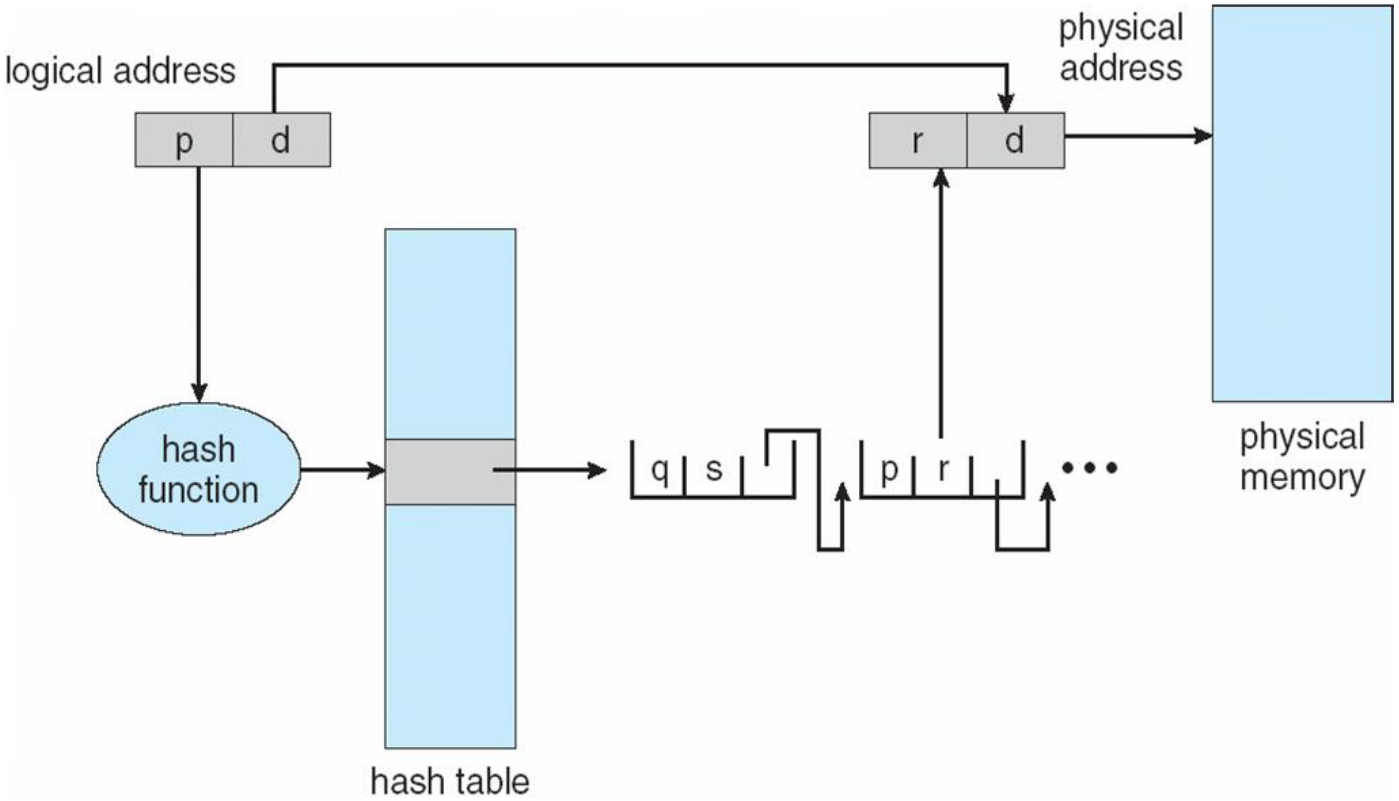


Method#2: Hashed Page Tables

- Hashed page tables are common when address spaces > 32 bits
- The virtual page number is hashed into a hashed page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains
  - (1) the virtual page number
  - (2) the value of the mapped page frame
  - (3) a pointer to the next element
- Since there might be multiple page table indexes that hash to the same value, the hashed page table provides a linked list for each hashed entry.
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

Method#2: Hashed Page Table

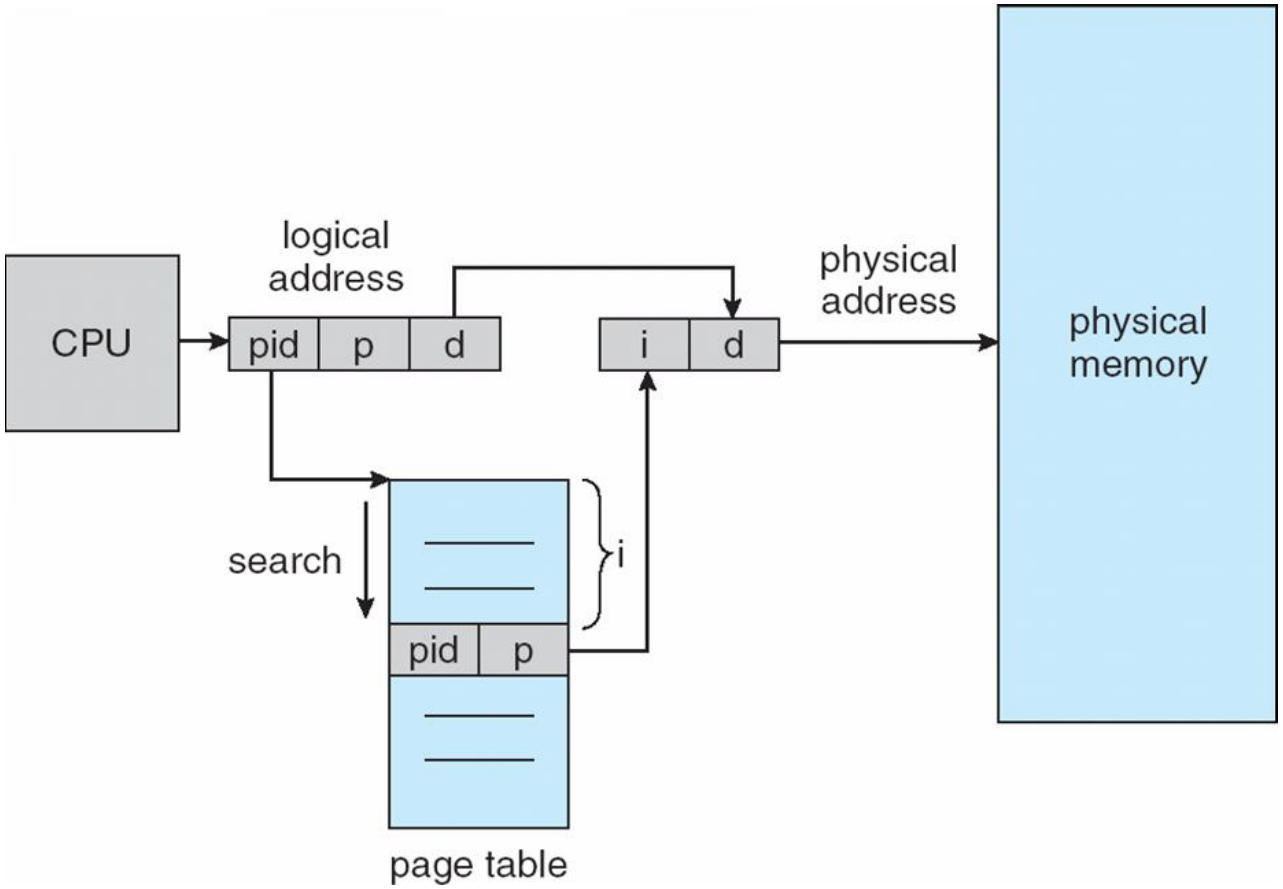
- The benefit of this approach is that in a hash table, the lookup time should be independent of the number of entries in the table.



Method#3: Inverted Page Table

- Normally, logical addresses are mapped to physical addresses in a page table for each process. This requires each process to have their own page table, each with potentially millions of entries.
- The ***inverted page table*** maps physical frames to logical pages.
- One entry for each real frame 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
- Structure is called inverted because it indexes page table entries by frame number rather than by virtual page number

Inverted Page Table Architecture



End of Chapter 8