Operating System
CS 2006
Lecture 10

Ms. Mahzaib Younas

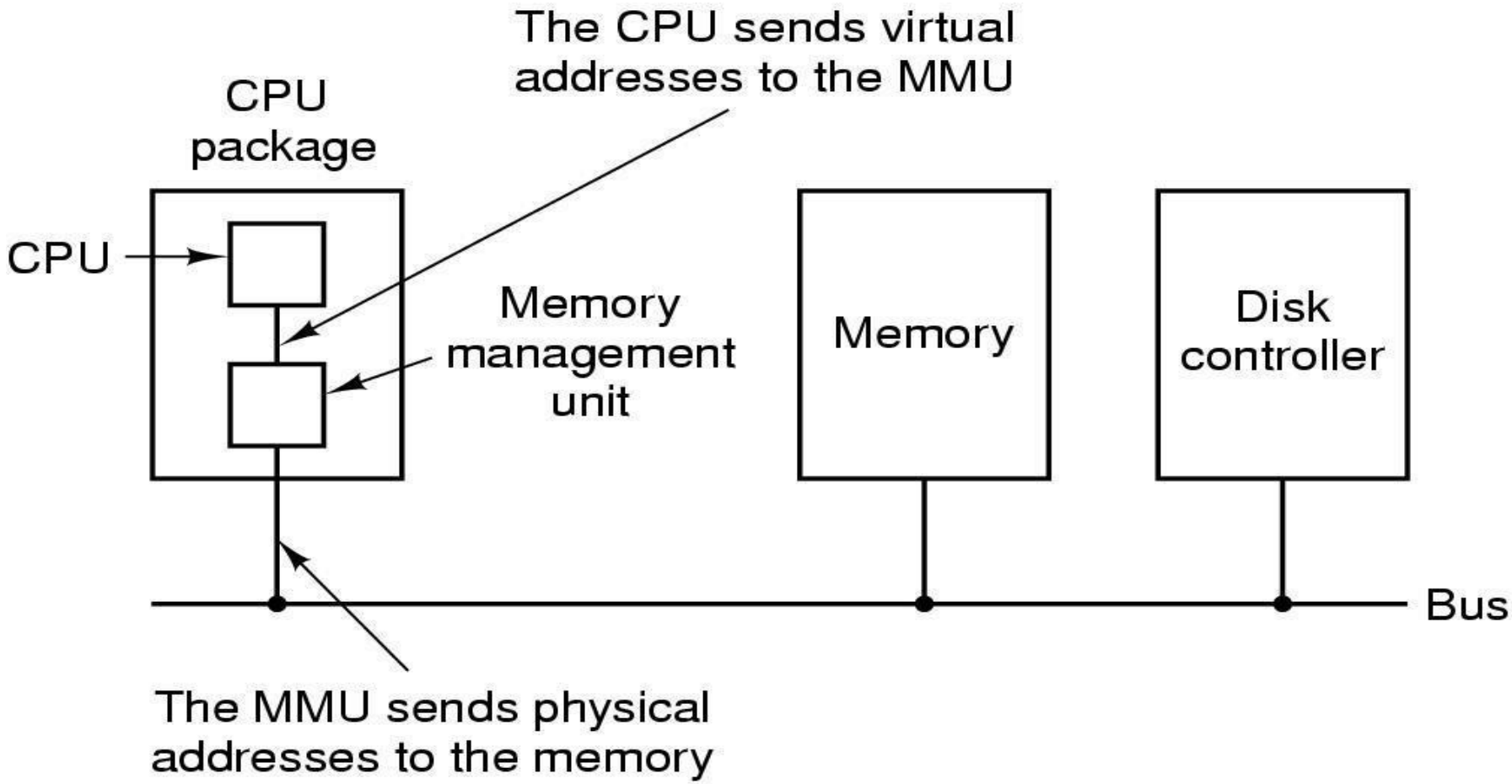
Lecturer Department of Computer Science

FAST NUCES CFD

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Memory management is all about managing process in the main memory such that
- Efficiently utilize the memory

CPU, MMU and Memory

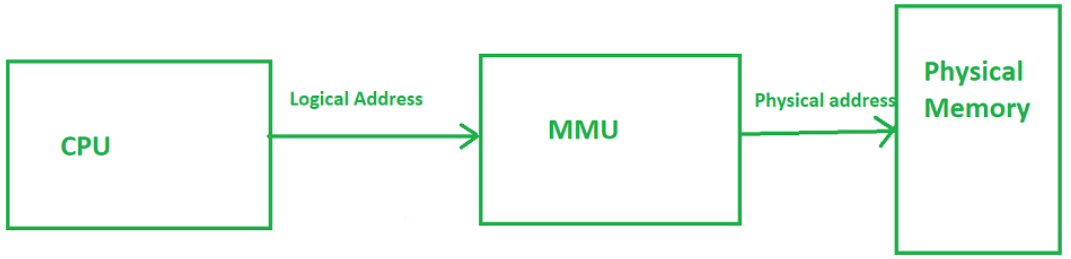


CPU MMU and Memory

- 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
- Protection of memory required to ensure correct operation

Memory Management Unit (MMU)

- MMU stands for Memory management unit also known as PMMU (paged memory management unit),
- Every computer system has a memory management unit , it is a hardware component whose main purpose is to convert virtual addresses/logic Address created by the CPU into physical addresses in the computer's memory.



Types of Address in CPU

- Virtual Address/Logic Address
  - A logical address is an address that is generated by the CPU during program execution. The logical address is a virtual address as it does not exist physically, and therefore, it is also known as a Virtual Address

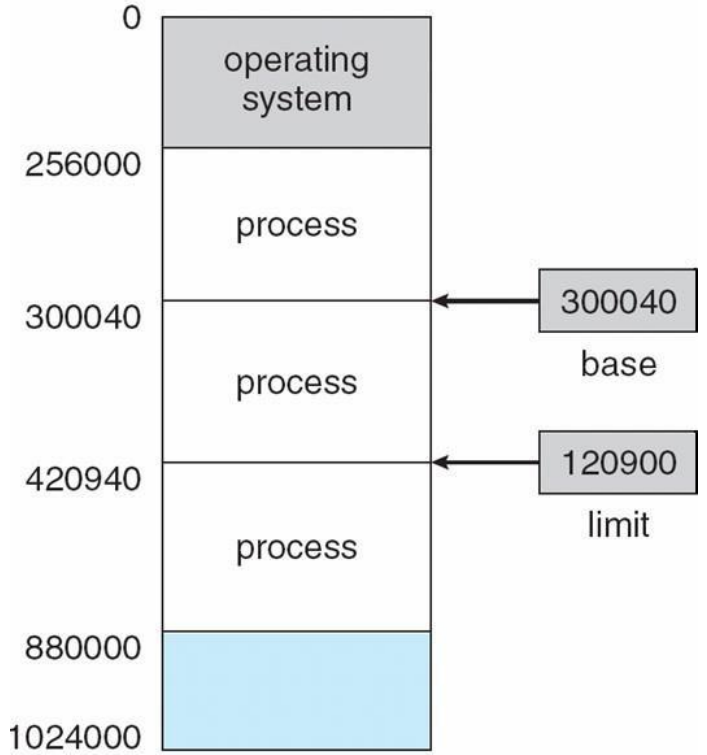
- Physical Address
  - The physical address identifies the physical location of required data in memory. The user never directly deals with the physical address but can access it by its corresponding logical address.

Components of MMU

- Processes should not be able to reference memory locations in another process without permission
- Base Register:
  - It contains the starting physical address of the process.
- Limit Register:
  - It mentions the limit relative to the base address on the region occupied by the process.

Base and Limit Register

- A pair of base and limit registers define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
- We need to Find the base and limit for each process
  - As shown in the diagram middle process
    - Base value = 300040
    - Limit = base(next) – Base(previous)
    - Limit= 420940 – 300040
    - Limit = 120900



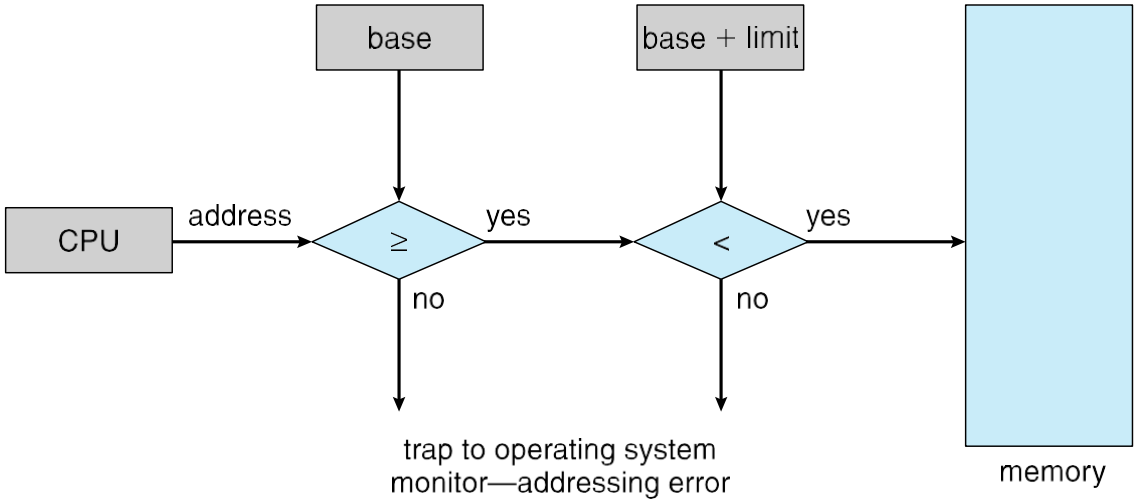
Next Location = Limit+ Base Address
Next Address = 300040 + 120900
Next Address = 420940

Base and Limit Register

- Address space
  - To protect process from each other: each process has its own memory space
- Two type of registers used
  - Base Register: Hold 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



Hardware Address Protection

- 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

Address Binding

- The Address Binding refers to the mapping of computer instructions and data to physical memory locations.
- logical and physical addresses are used in computer memory.
- It assigns a physical memory region to a logical pointer by mapping a physical address to a logical address known as a virtual address.
- It is also a component of computer memory management that the OS performs on behalf of applications that require memory access.

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(fixed and independent to run time) to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Types of address Binding in OS

- There are mainly three types of an address binding in the OS. These are as follows:

1. Compile Time Address Binding
2. Load Time Address Binding
3. Execution Time or Dynamic Address Binding

Types of address Binding in OS

1. **Compile Time Address Binding**

   If memory location known a priori, absolute code can be generated; must recompile code if starting location changes

2. **Load Time Address Binding**

   Must generate relocatable code if memory location is not known at compile time

3. **Execution Time or Dynamic Address Binding**

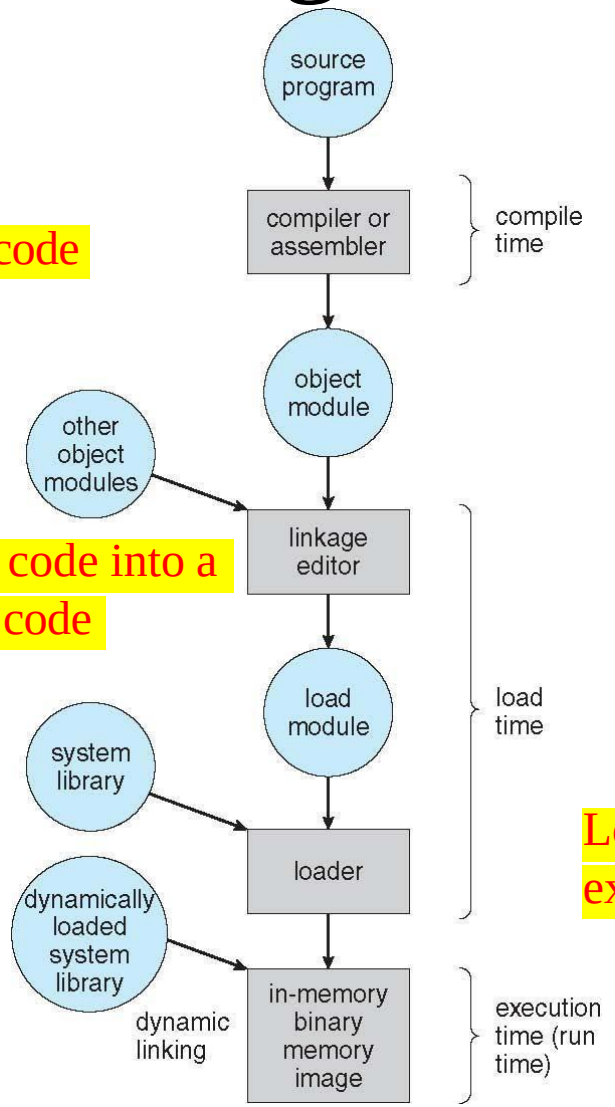
   Binding delayed until run time if the process can be moved during its execution from one memory segment to another [it may need hardware support]

Multi step programming for User Variable

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

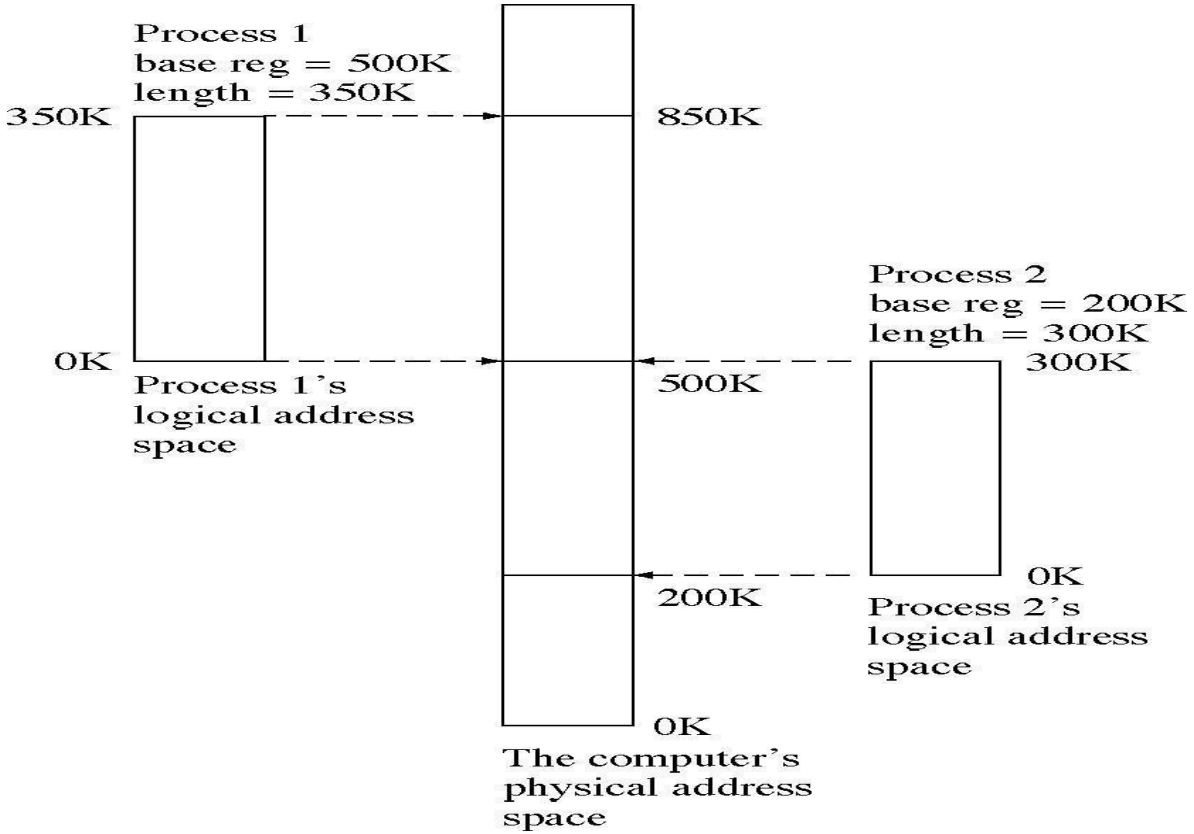
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit

Logical vs Physical Address Space

- 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
- Logical address space
  - the set of all logical addresses generated by a program
- Physical address space
  - the set of all physical addresses generated by a program

Example of Logical and Physical Address Space

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



MMU

- 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
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with logical addresses; it never sees the real physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

MMU

- Logical address mapped to physical addresses before use

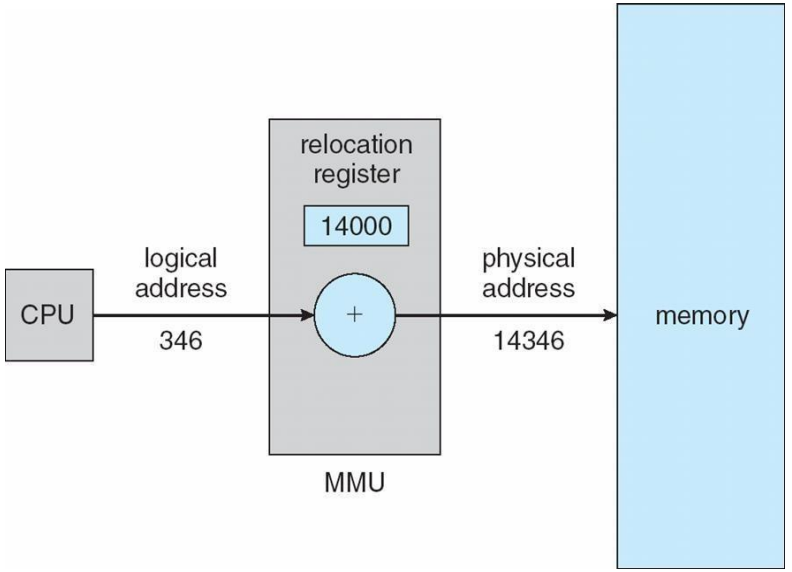
  Logical addresses: in range 0 to max

  Physical addresses: in range R + 0 to R + max

- For a base value R in the relocation register

Dynamic Relocation using relocation register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



Dynamic Linking

- A user routine is not loaded until it is called.
- Loading is delayed until run-time
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- Loaded program portion is much smaller than total program.

Static vs Dynamic Linking

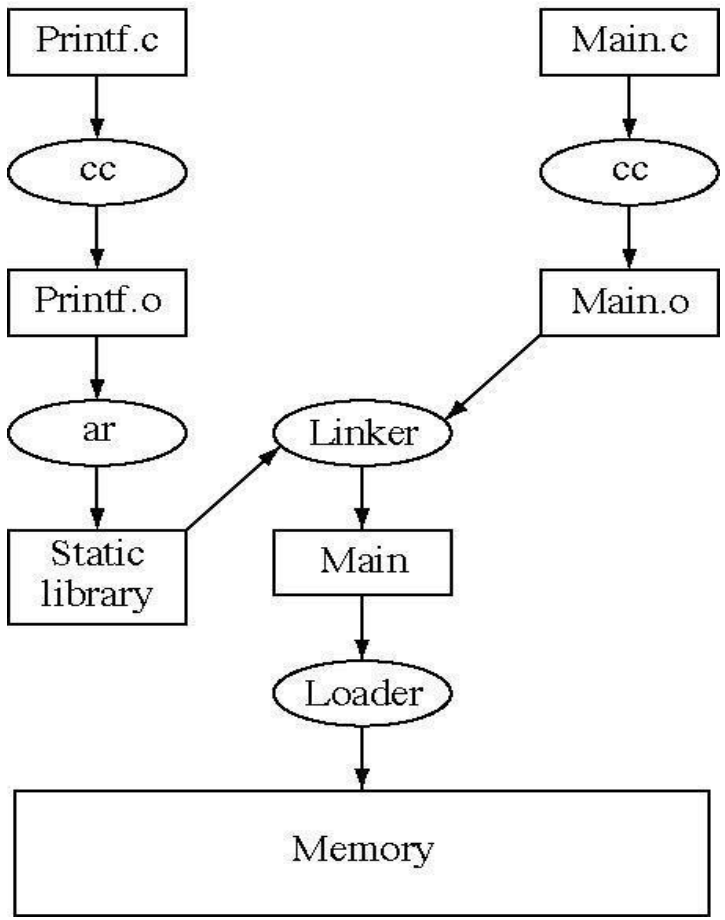
Static Linking

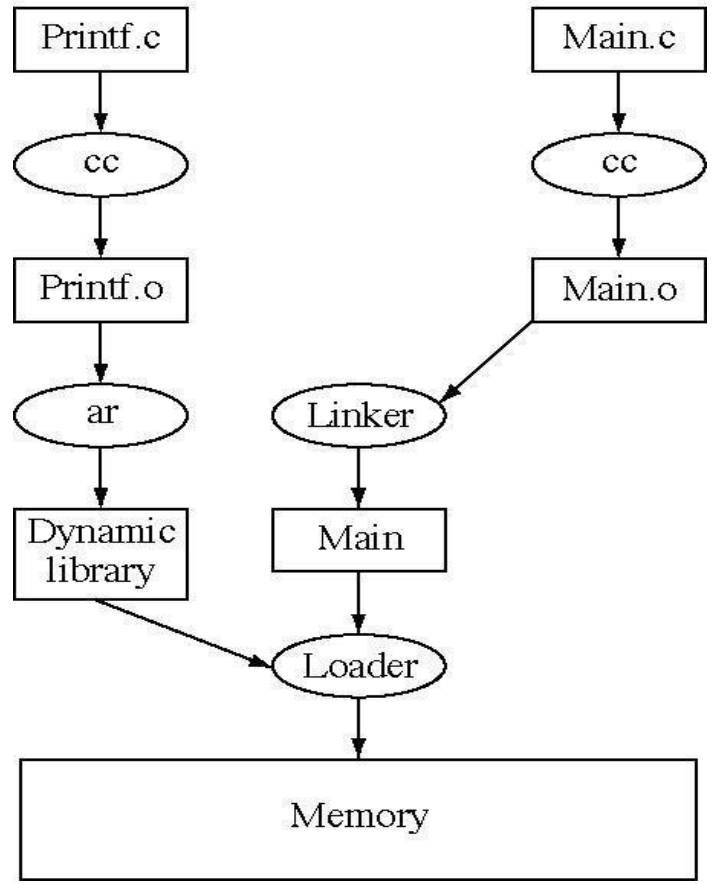
- system libraries and program code combined by the loader into the binary program image (the final executable file) running .exe files

Dynamic Linking

- Linking postponed until execution time
- The executing program (the running process)
- Dynamic linking is particularly useful for large system libraries
- System also known as shared libraries with different version numbers

Static vs load time Dynamic Linking





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

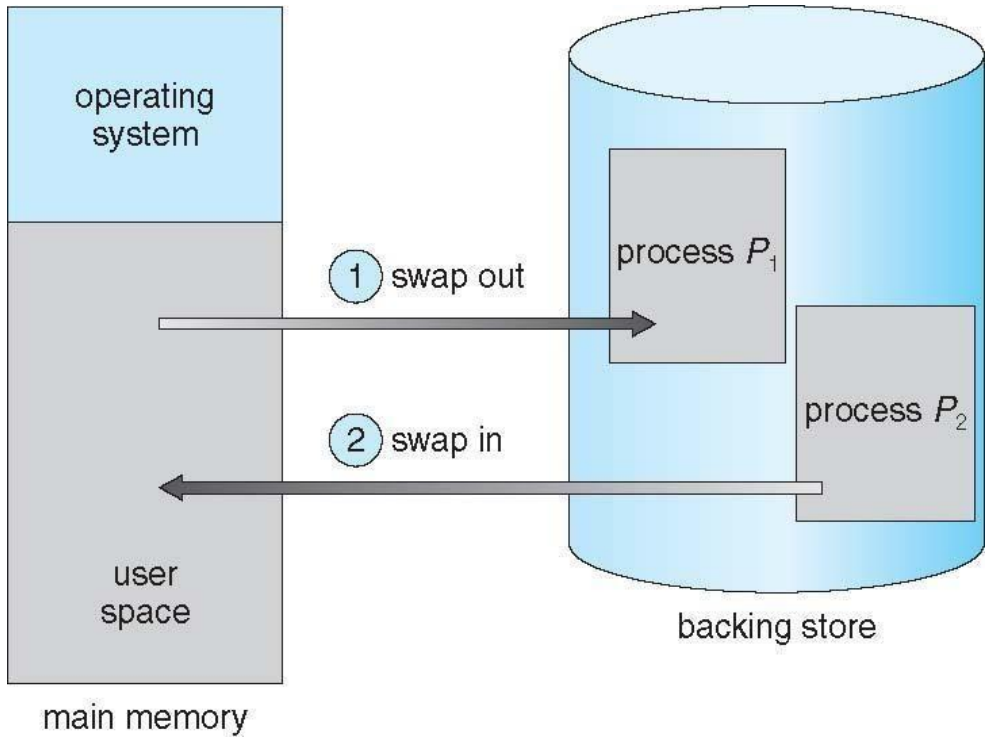
Hence, swapping increases the degree of multiprogramming

Swapping

- **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
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping

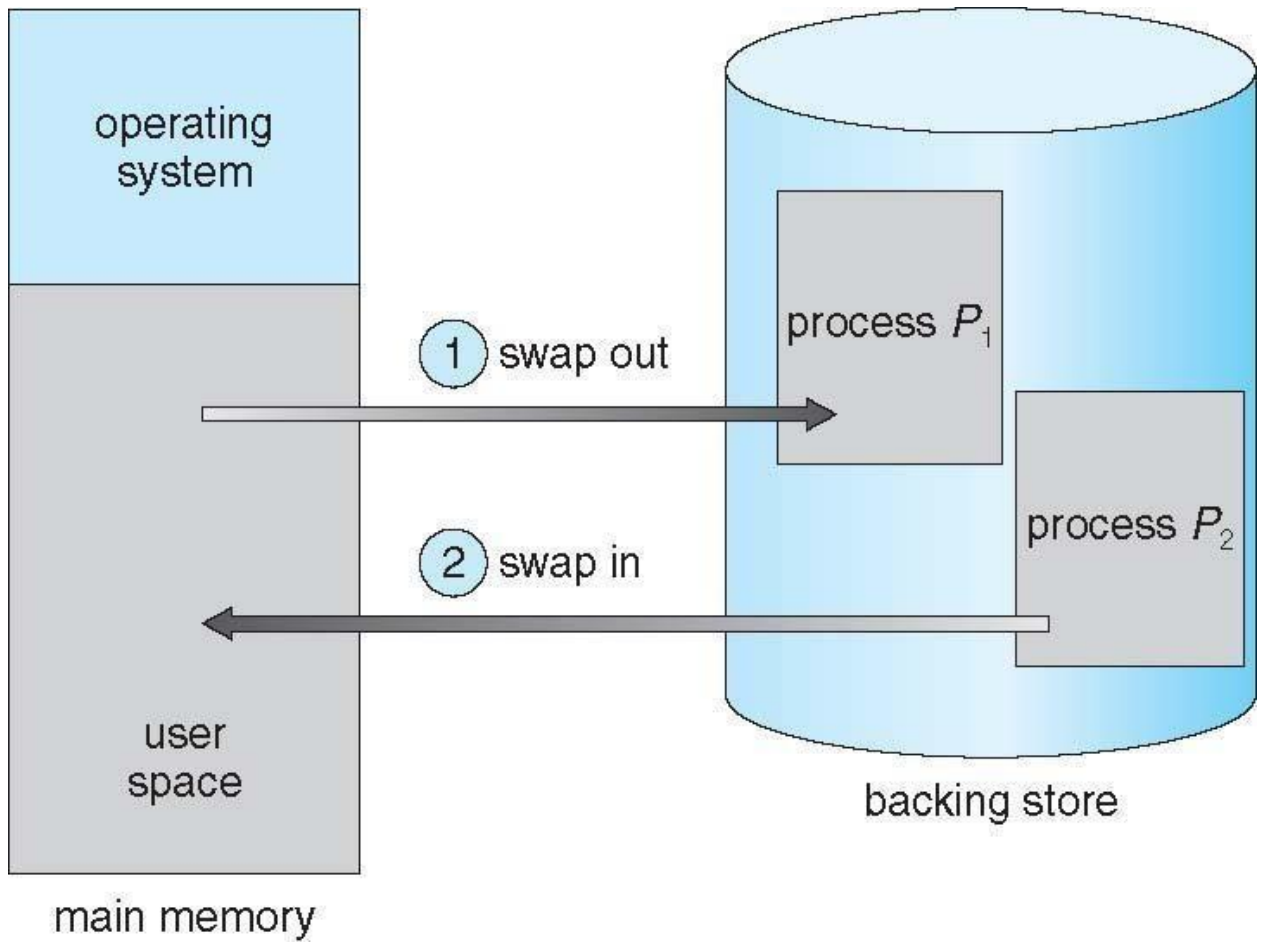
- Dispatcher called when the CPU scheduler selects a process P2 from queue
- If not enough free space in memory for P1
- Swap in P2 from backing store and swap out some P1 from memory
- Reload registers and transfer control to P2; i.e. P2 is now in running state



Swapping

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



Context Switch Time Include in 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
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via request_memory() and release_memory()

Swapping

- Other constraints as well on swapping
  - Pending I/O – can’t swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

Contiguous Allocation

- Contiguous memory allocation is a memory management technique used by operating systems to allocate a block of contiguous memory to a process.
- The allocation of contiguous memory to a process involves dividing the available memory into fixed-sized partitions or segments.
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

Contiguous Allocation

- 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

Multiple-partition allocation

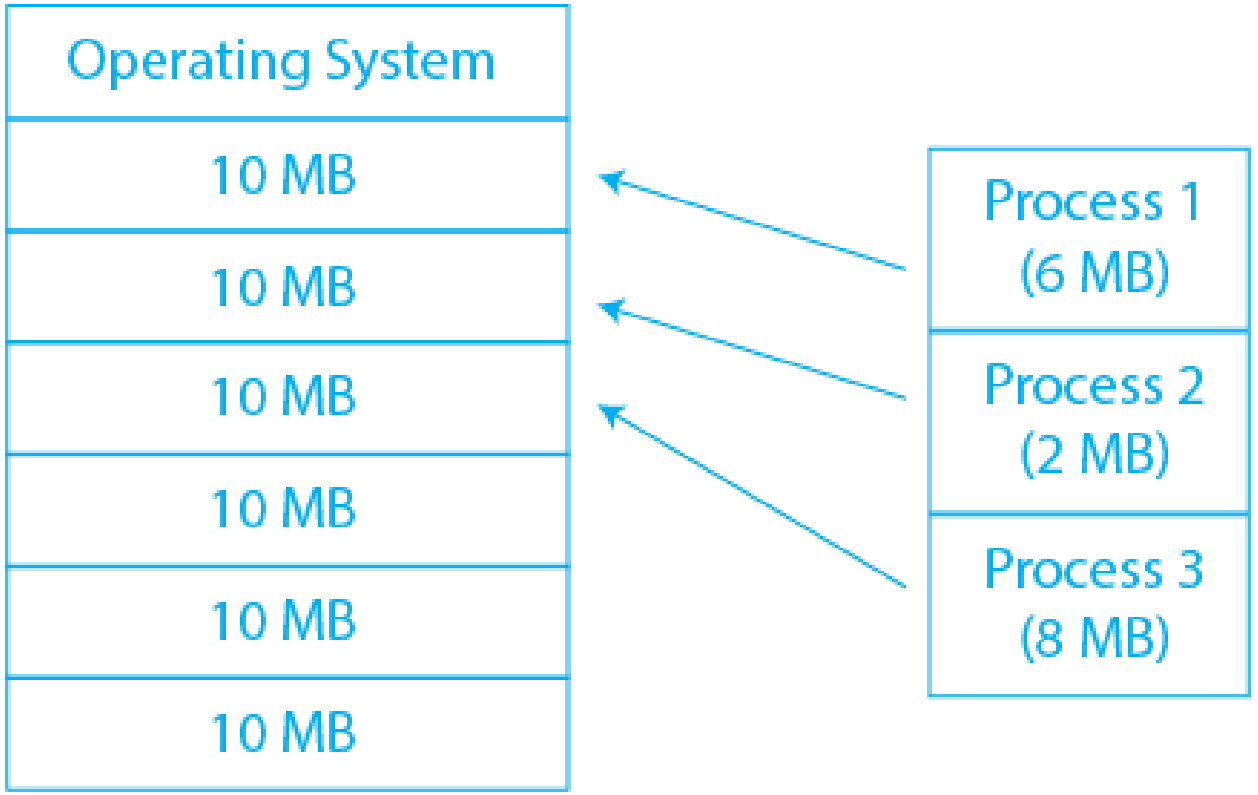
- There are mainly two contiguous memory allocation techniques, which can be used to allocate the contiguous memory.

1. Fixed-size partition schema
2. Variable-size/fixed and unequal partition schema

Fixed Size Partition

- Fixed-size partition schema is a technique used in contiguous memory allocation to divide the available memory into fixed-sized partitions or segments.
- Each partition is of a fixed size and can be allocated to a single process.
- The size of the partition is typically determined by the operating system and may be based on factors such as the amount of physical memory available and the expected size of the processes.
- In this technique, each process gets the same amount of memory size thus this technique is also called static partition schema

Fixed Size Partition Layout

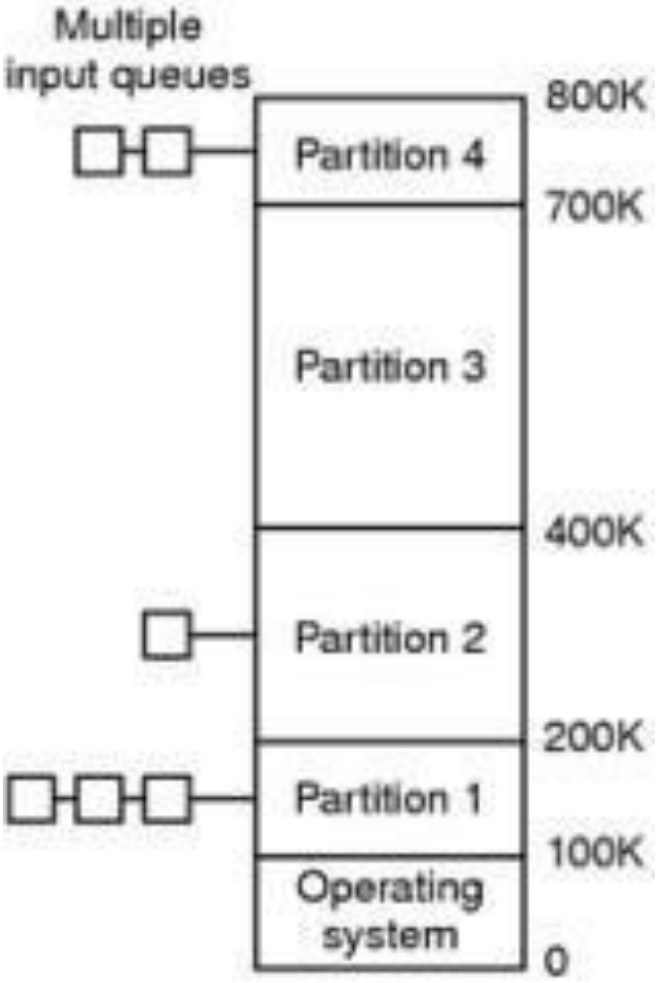


All partitions are of equal size, small processes waste space, Large processes don't fit in (Solution: Fixed Unequal Partitions)

Fixed and unequal Partition

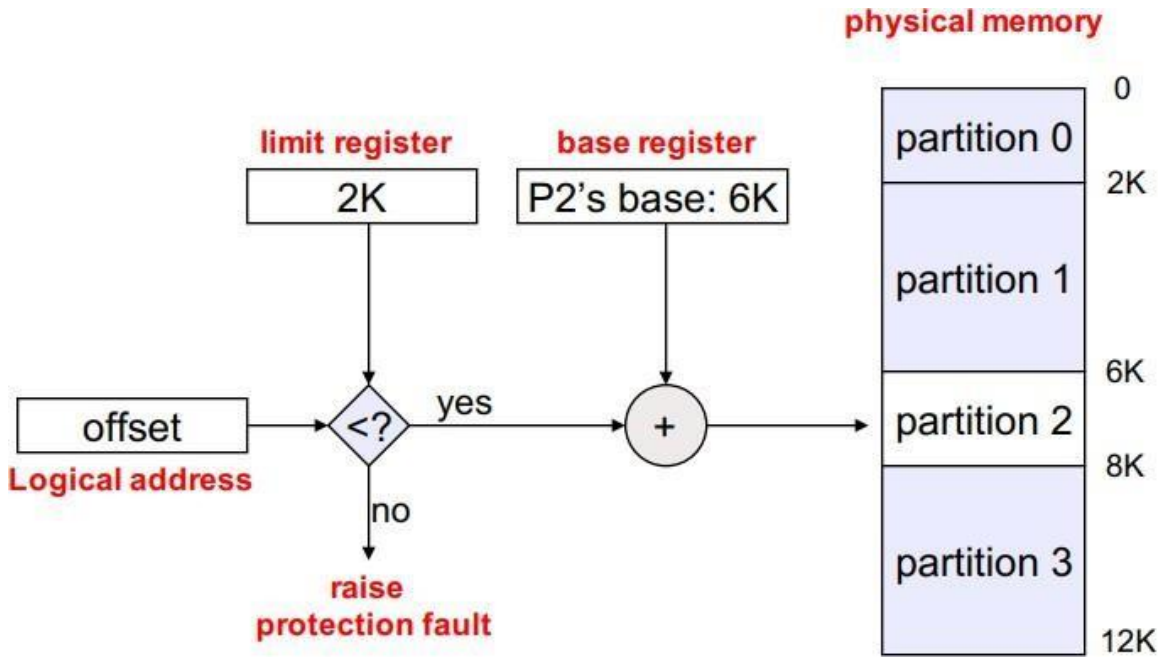
- Physical memory is broken up into fixed but unequal partitions

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



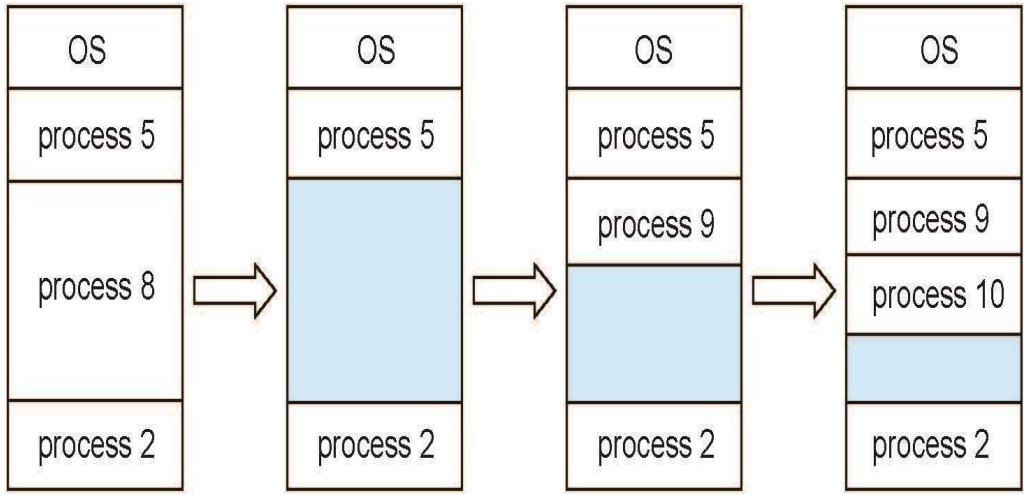
Fixed Partition Mechanism

- 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.



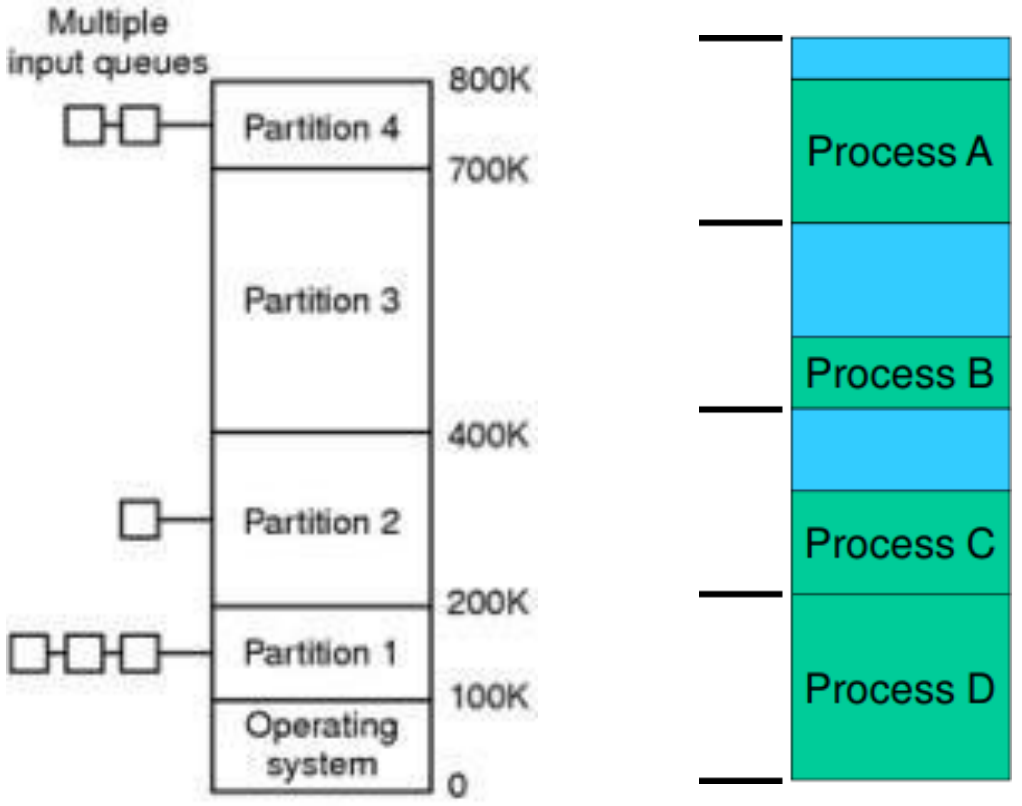
Dynamic Partition

- 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

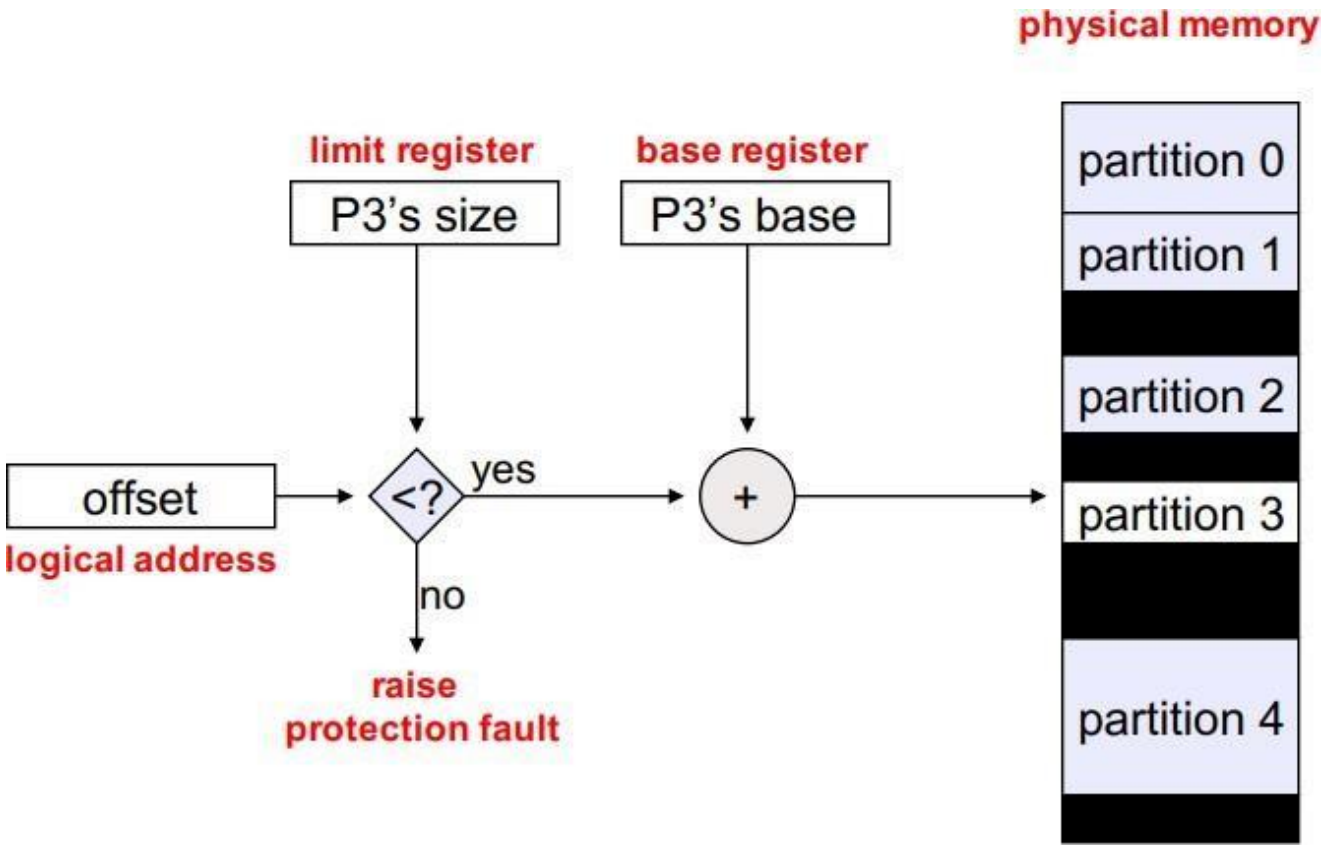


Internal Fragmentation in old Techniques

- 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)

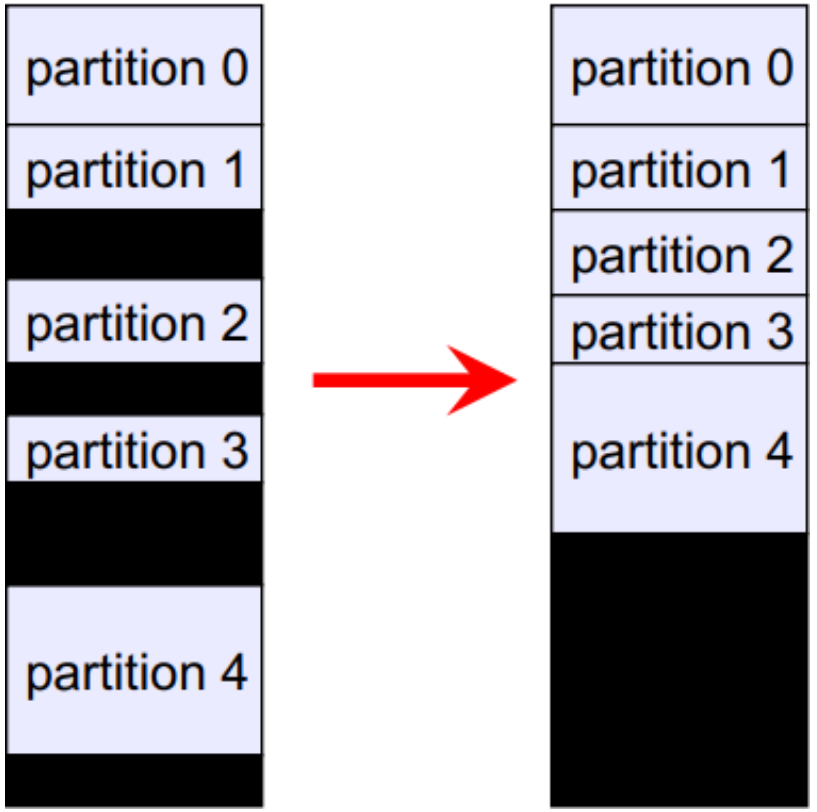


Fragmentation

- 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
- 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**
- First fit analysis reveals that **given N blocks allocated, 0.5 N blocks** lost to fragmentation
  - **1/3 may be unusable -> 50-percent rule**

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
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems



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

Dynamic Storage Allocation

- First-fit: Allocate the first hole that is big enough
- Next-fit: Allocate the next available 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

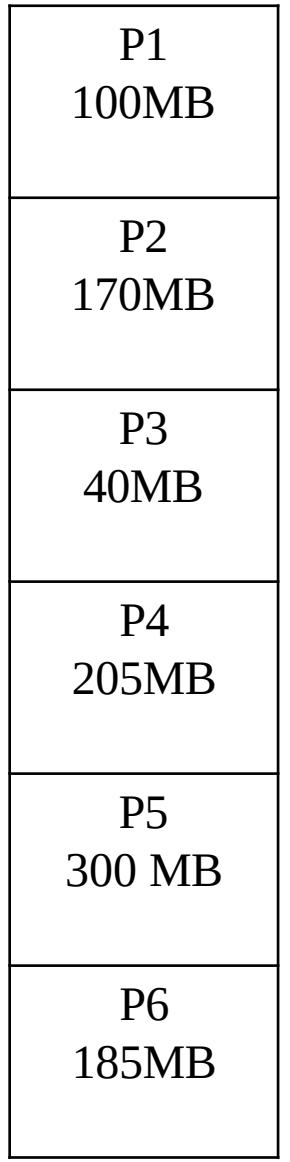
- Worst-fit: Allocate the largest hole; must also search entire list
  - Produces the largest leftover hole

Dynamic Storage Allocation

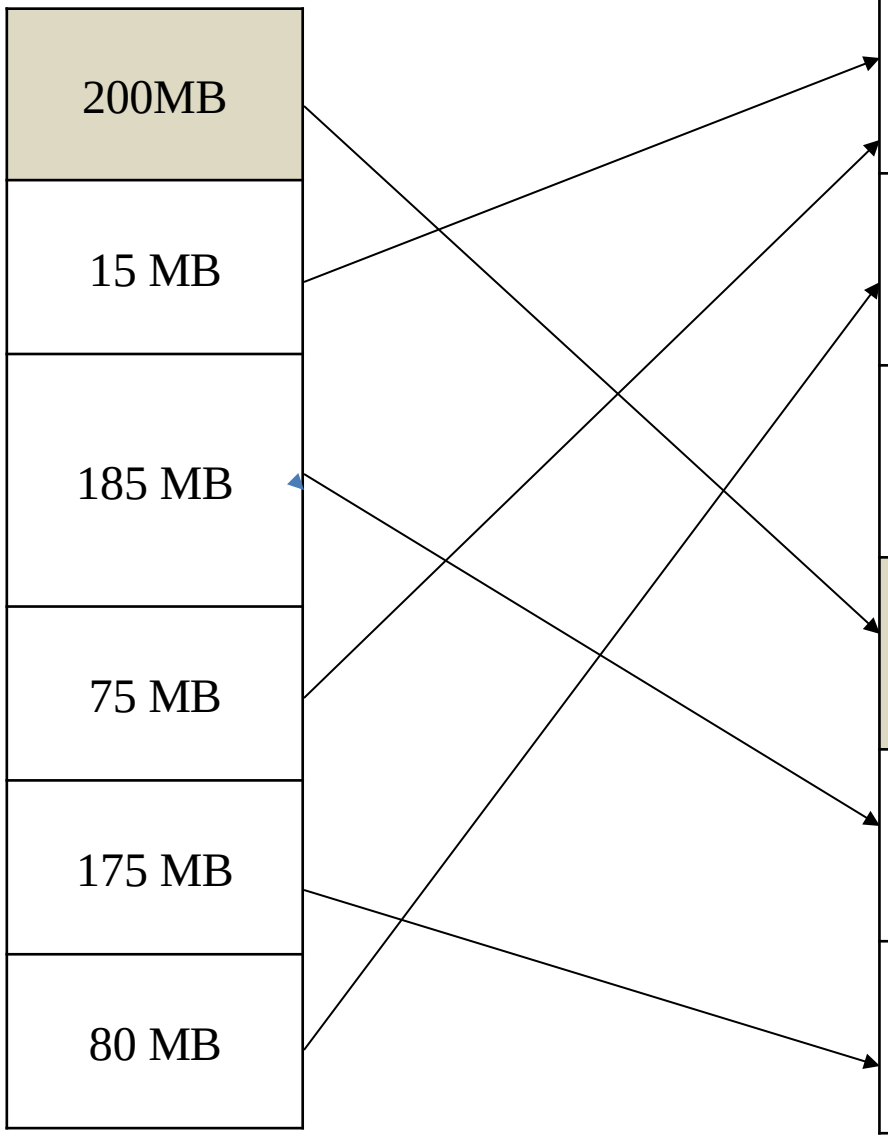
Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the **first-fit, best-fit, and worst-fit** algorithms place processes of size 200 MB, 15 MB, 185 MB, 75 MB, 175 MB, and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. **Comment on how efficiently each of the algorithms manages memory**.

Dynamic Storage Allocation

Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the **first-fit, best-fit, and worst-fit** algorithms place processes of size 200 MB, 15 MB, 185 MB, 75 MB, 175 MB, and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. **Comment on how efficiently each of the algorithms manages memory**.

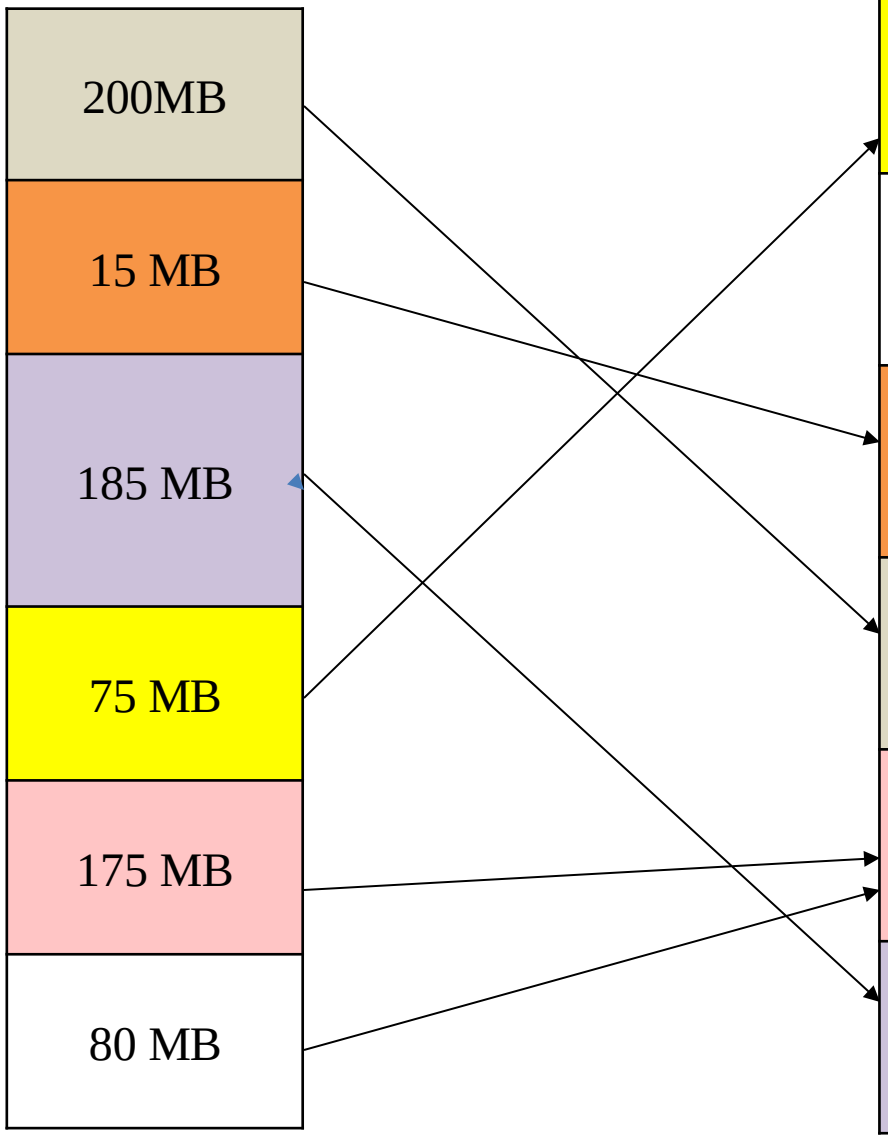


First Fit Algorithm



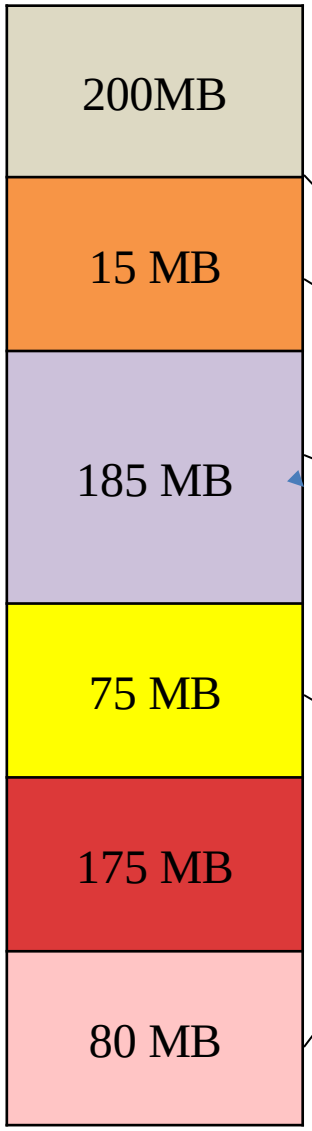
Memory	Left Memory	
P1 100MB	85 MB	10 Mb
P2 170MB	90 MB	
P3 40MB	40 MB	
P4 205MB	5 MB	
P5 300 MB	115 MB	
P6 185MB	10 MB	

Best Fit Algorithm



Memory	Left Memory	
P1 100MB	25 MB	
P2 170MB	170 MB	
P3 40MB	35 MB	
P4 205MB	5 MB	
P5 300 MB	125MB	45 Mb
P6 185MB	0 MB	

Worst Fit Algorithm



Memory	Left Memory	
P1 100MB	100 MB	
P2 170MB	90 MB	
P3 40MB	40 MB	
P4 205MB	190 MB	5 MB
P5 300 MB	100 Mb	
P6 185MB	110 MB	

Worst Fit Algorithm

- 200 MB is placed in the 300 MB partition, leaving 100 MB, 170 MB, 40 MB, 5 MB, 100 MB, and 185 MB. 2. 15 MB is placed in the 205 MB partition, leaving 100 MB, 170 MB, 40 MB, 5 MB, 100 MB, and 185 MB. 3. 185 MB is placed in the 190 MB partition, leaving 100 MB, 170 MB, 40 MB, 5 MB, 5 MB, and 185 MB. 4. 75 MB is placed in the 185 MB partition, leaving 100 MB, 170 MB, 40 MB, 5 MB, 5 MB, and 110 MB. 5. 175 MB cannot be satisfied as there is no partition large enough to hold the request. 6. 80 MB is placed in the 170 MB partition, leaving 100 MB, 90 MB, 40 MB, 5 MB, 5 MB, and 110 MB.

Calculate the internal Fragmentation

Calculate the internal fragmentation if page size is 2048 bytes and process size is 72,776 bytes.

Solution:

Page Size = 2048 bytes
Process Size = 72,776 bytes.

Step 1: Calculate the no of pages

$$= \frac{Process\ Size}{Page\ Size}$$

$$= \frac{72776}{2048} = 35.35 \cong 35$$

Calculate the remaning bytes in last page

Calculate the Remaining Bytes in the Last Page:

Remaining bytes = Process size % Page size

Remaining bytes = 72,766 % 2,048 = 1,086 bytes

Final Internal fragmentation per page .

Internal fragmentation = Page size - Remaining bytes

Internal fragmentation = 2,048 - 1,086 = 962 bytes

Paging

Paging is a storage mechanism that allows OS to retrieve processes from the secondary storage into the main memory in the form of pages .

- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**

Paging

- 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
- Backing store likewise split into pages
- Still have Internal fragmentation

Page Size

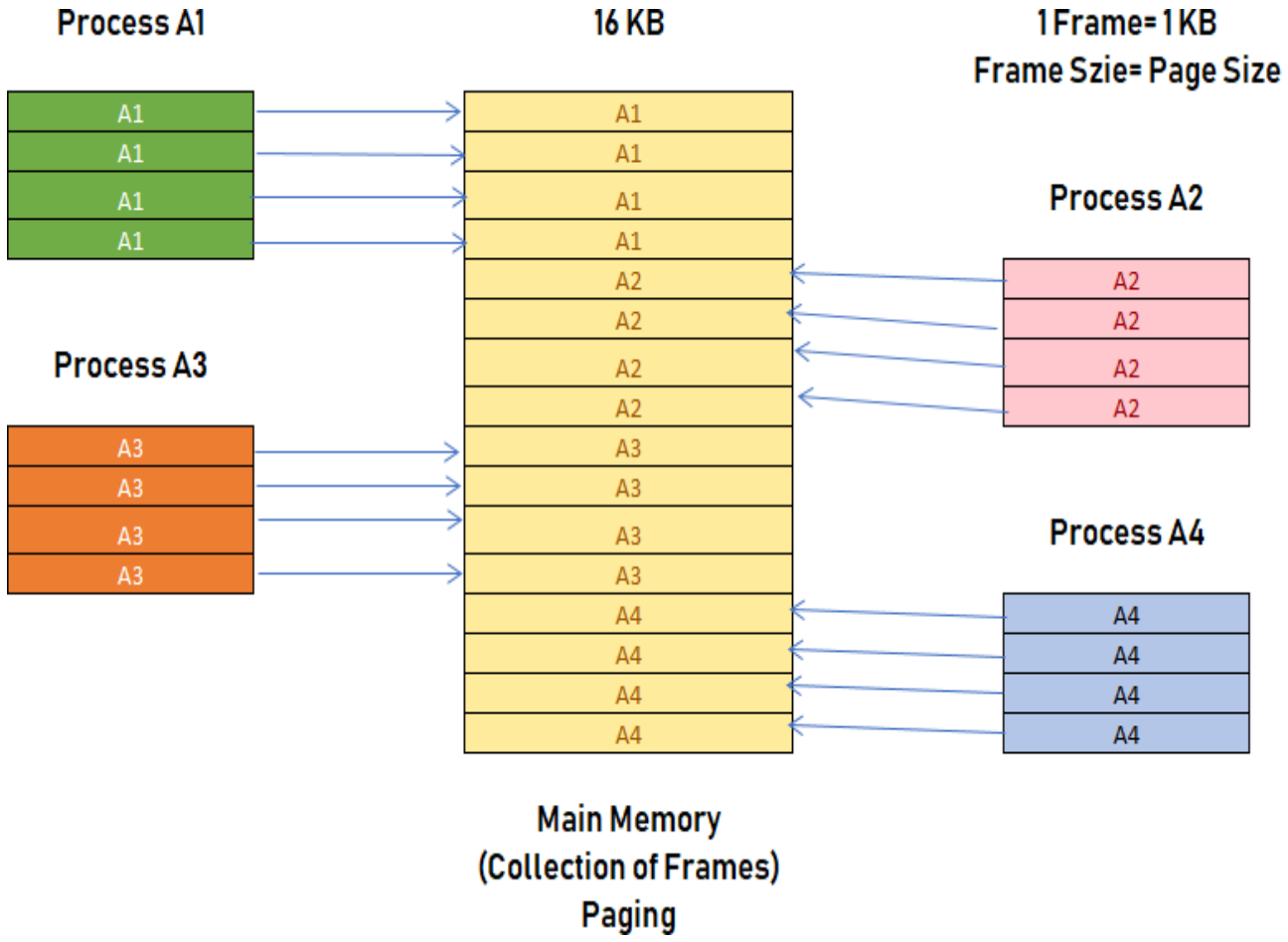
- The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, typically varying between 4 KB and 1 GB per page, depending on the computer architecture.

- The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

- If the size of the logical address space is 2m, and a page size is 2n bytes, then the high-order m−n bits of a logical address designate the page number, and the n low-order bits designate the page offset

Example of Paging

- Example:
  - If the main memory size is 16 KB and Frame size is 1 KB. Here, the main memory will be divided into the collection of 16 frames of 1 KB each.
  - There are 4 separate processes in the system that is A1, A2, A3, and A4 of 4 KB each. Here, all the processes are divided into pages of 1 KB each so that operating system can store one page in one frame.

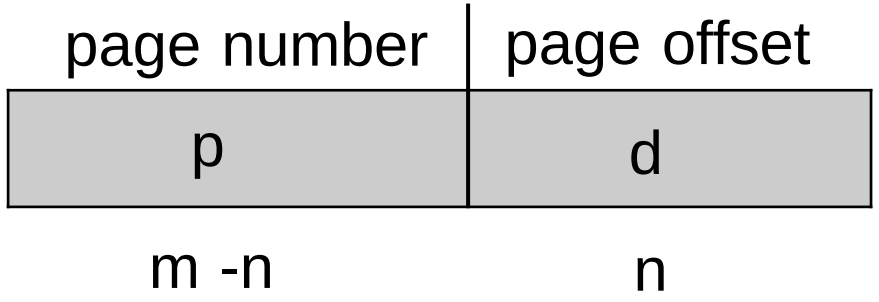


Address Space in Paging

- The logical address is the address generated by the CPU for every page while the physical address is the actual address of the frame where each page will be stored.
- When a page is to be accessed by the CPU by using the logical address, the operating system needs to obtain the physical address to access that page physically.

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

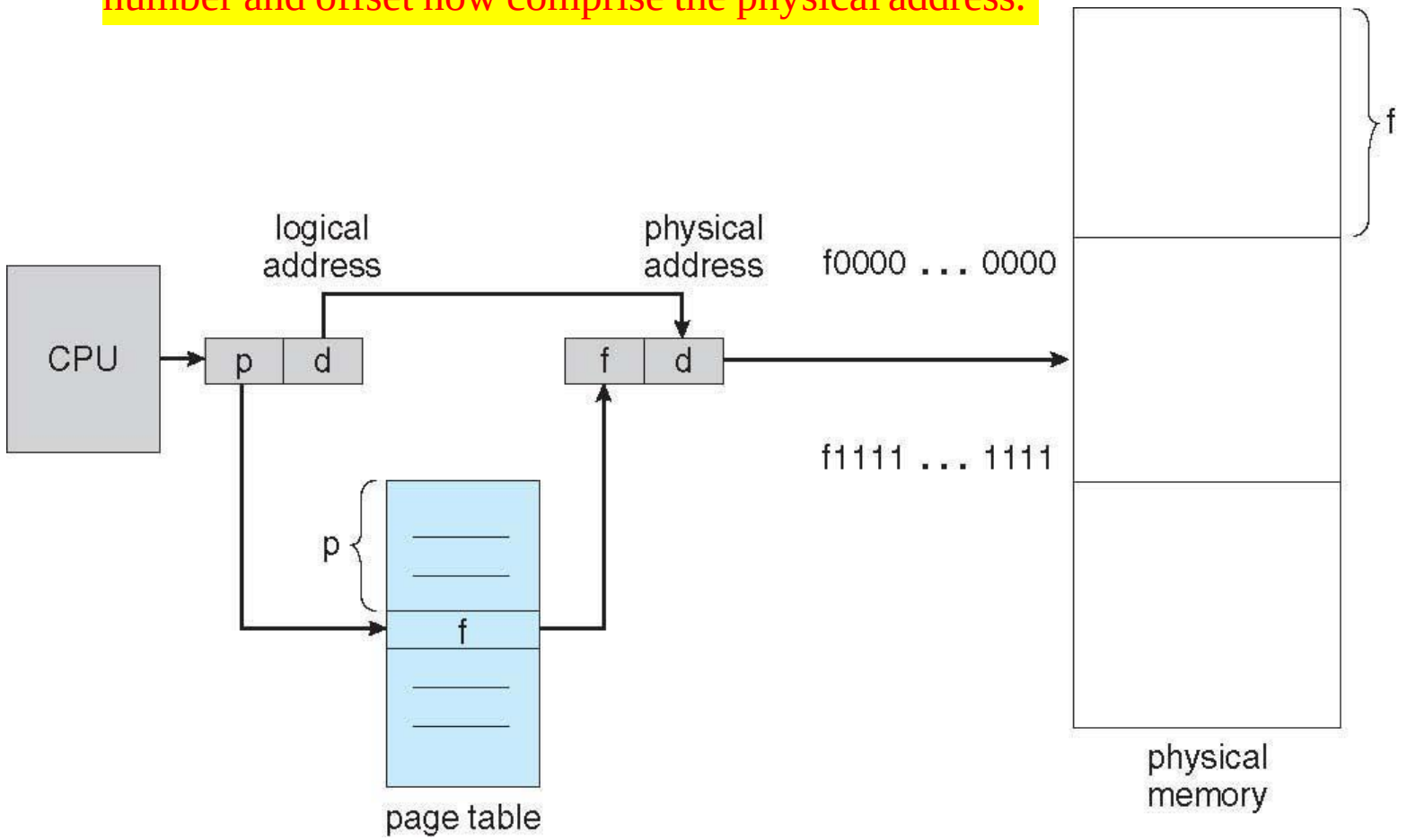


Memory management unit of OS needs to convert the page number to the frame number.

Paging Model

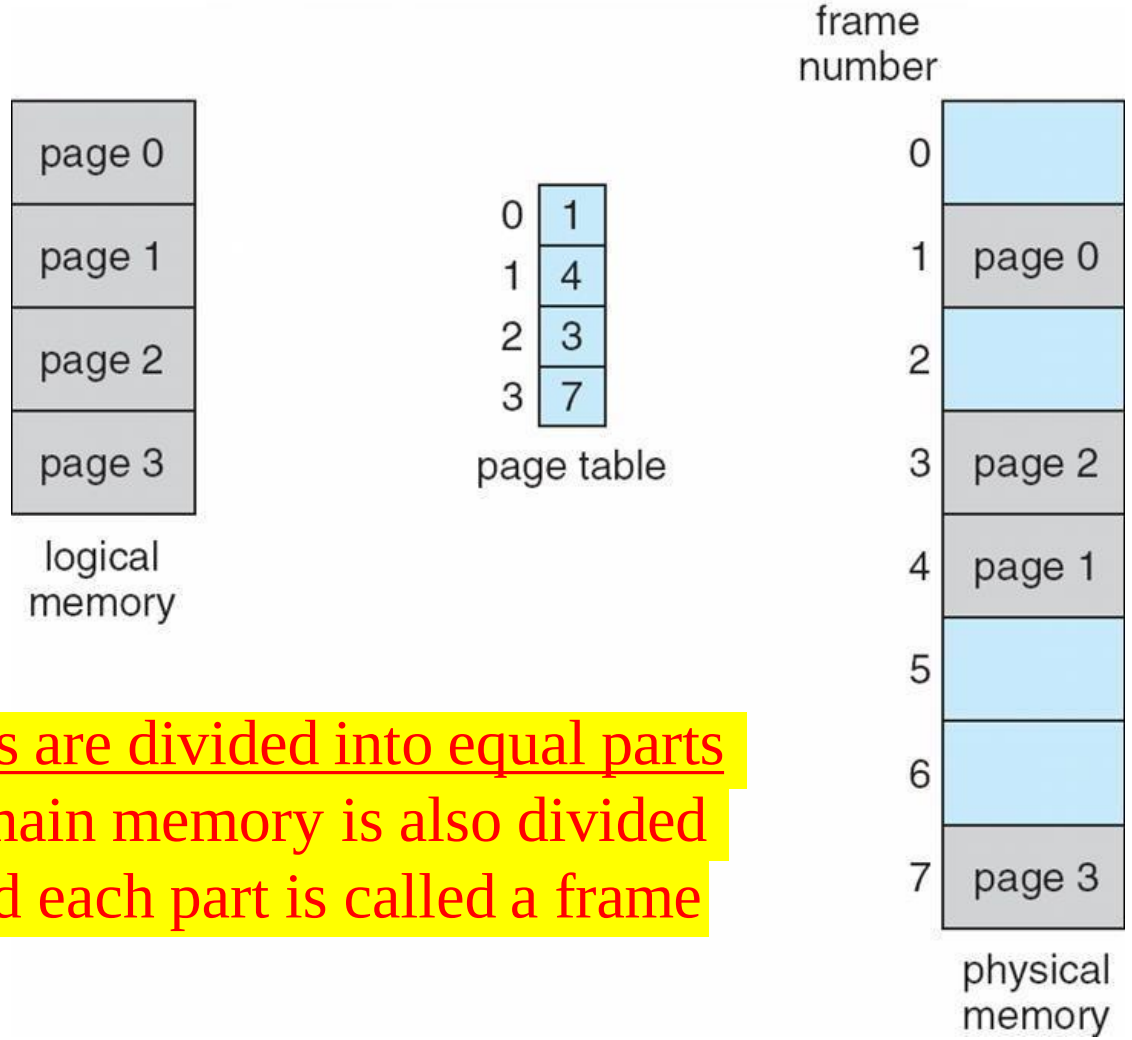
As offset d does not change, it is not replaced, and the frame number and offset now comprise the physical address.

1. Extract the page number p and use it as an index into the page table.
2. Extract the corresponding frame number f from the page table.
3. Replace the page number p in the logical address with the frame number f .



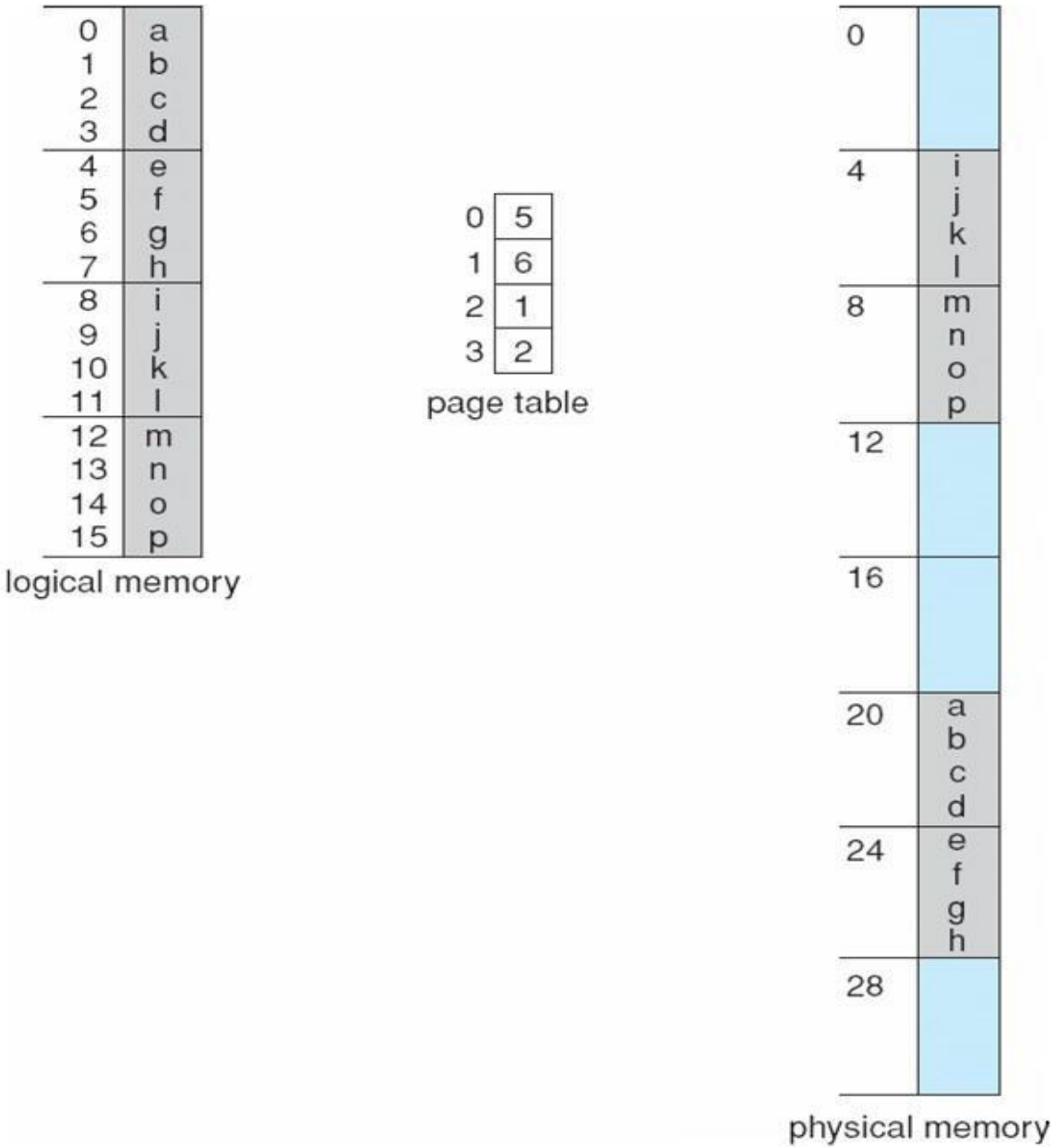
The page number is used as an index in to a per-process page table.

Paging Table of Logical and Physical Memory



In paging, processes are divided into equal parts called pages, and main memory is also divided into equal parts and each part is called a frame

Paging Example



Free Frames

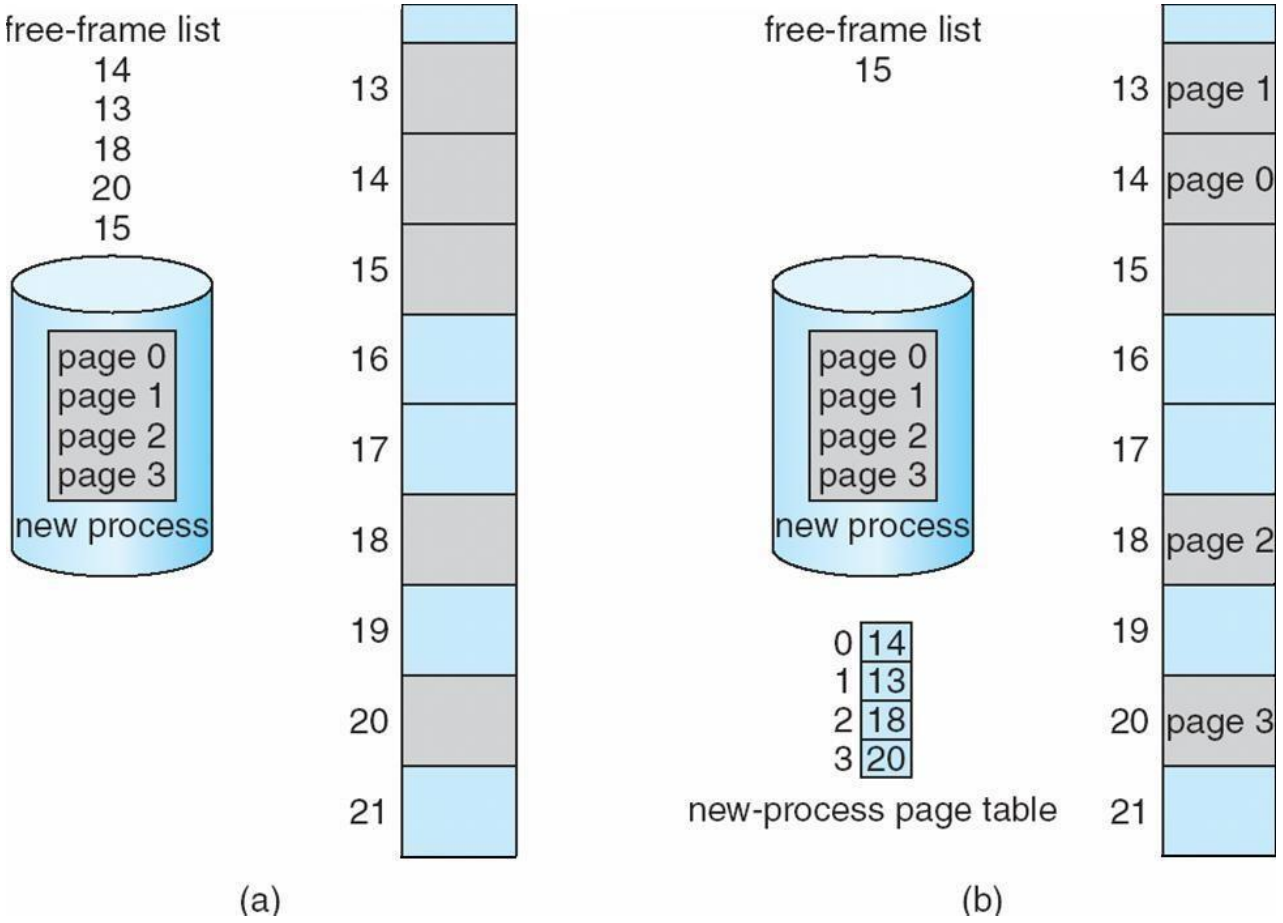


Figure A represent the free frame list , now in Figure B we get the new-process page table with page frame after loading these process in memory the free frame list contain only one frame that is 15 as shown in figure b.

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 (PTLR) 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)

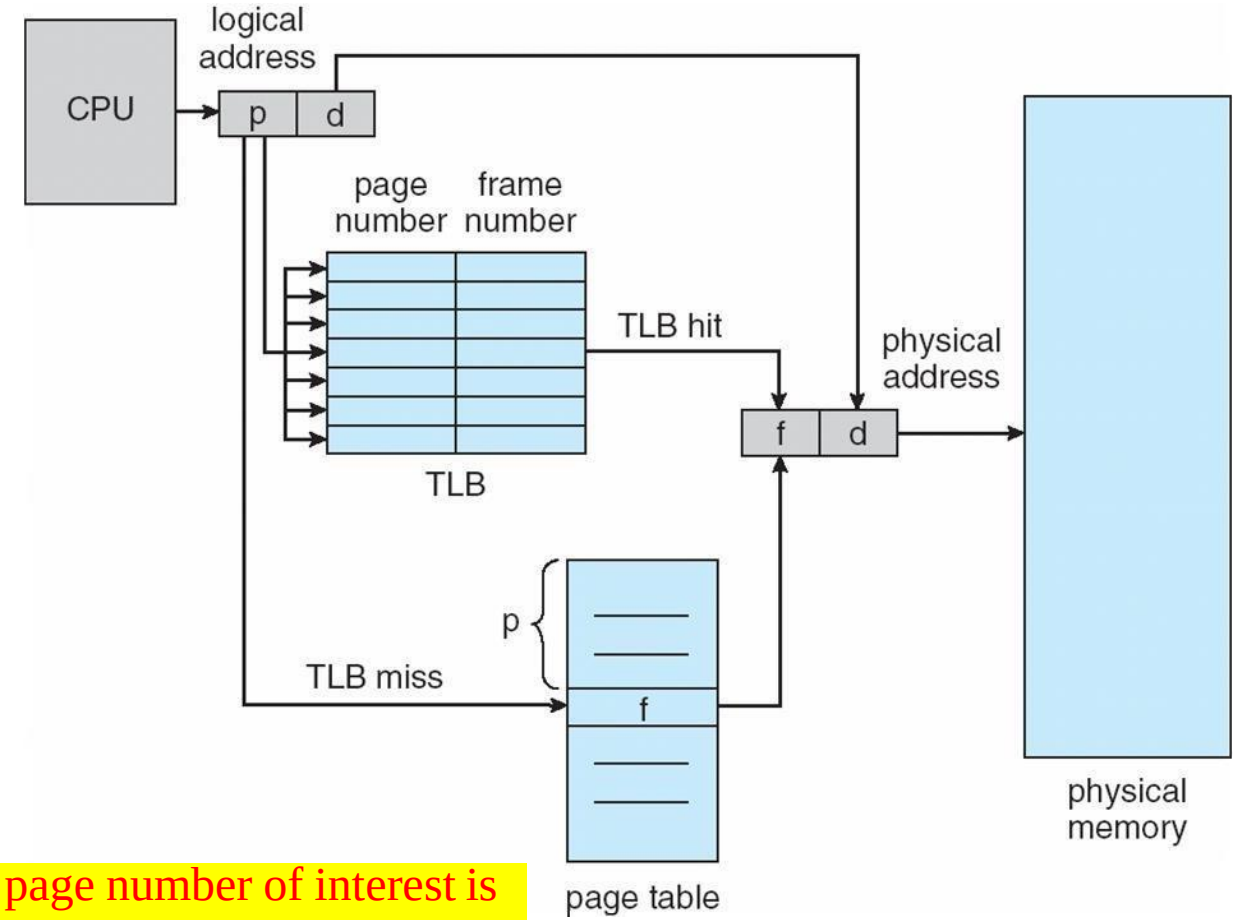
Associative Memory

- Associative memory – parallel search

Page #	Frame #

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

Paging Hardware with TLB



special, small, fast-lookup hardware cache called a translation look-aside buffer (TLB).

If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random

The percentage of times that the page number of interest is found in the TLB is called the hit ratio.

Effective Access Time

- If we fail to find the page number in the TLB then we must first access memory for the
- page table and frame number (10 nanoseconds) and then access the desired byte in memory (10 nanoseconds), for a total of 20 nanoseconds. (We are assuming that a page table lookup takes only one memory access, but it can take more, as we shall see.)
- To find the effective memory-access time, we weight the case by its probability with 80 percent hit ratio:

$$\text{effective access time} = 0.80 \times 10 + 0.20 \times 20$$

$$= 12 \text{ nanosecond}$$