K. J. Somaiya College of Engineering, Mumbai

(A constituent College of Somaiya Vidyavihar University)

Operating System

Module 5. Memory Management

Dr. Prasanna Shete
Dept. of Computer Engineering
prasannashete@somaiya.edu
Mobile/WhatsApp 9960452937

Memory Management

- **Objectives**:
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques

Memory Basics

- Memory is central to operation of computing systems

- Memory = a large array of words/bytes
  - Each byte or word has own address

- Memory contains both; the program to be executed and data
  - Program is executed line by line with Instruction Fetch, Instruction Decode, Operand Fetch, Execute cycles
- Program Counter (PC) contains address of memory location to be executed next

Memory Basics

- The CPU fetches instructions from memory according to the value of the PC
  - These instructions may cause additional loading from and storing to specific memory addresses

- A typical instruction-execution cycle-

1. First fetches an instruction from memory
2. Instruction is then decoded and may cause operands to be fetched from memory
3. After the instruction has been executed on the operands, results may be stored back in memory

- Memory unit only sees a stream of addresses + read requests, or address + data and write requests

Memory Basics

- CPU can directly access only main memory and processor registers

- The instructions take main memory addresses as program arguments and not disk addresses
- Hence- Data and instructions executing them, have to be in Main memory

- Issues and concerns?
  - Speed of access: solution □ cache memory
  - Protect program data: solution □ base and limit registers
  - Size of memory: solution □ swapping, virtual memory
  - Many more ones….

Memory Management: Introduction

- 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 is done in one CPU clock (or less)
- 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: Basic Hardware

- Register access in one CPU/clock cycle (or less)
- Main memory access can take many cycles, causing a stall
- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

  - Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.

Memory Management: Basic Hardware

- Register access in one CPU clock (or less)
- Main memory access can take many cycles, causing a stall
- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

- Main memory is accessed via a transaction on the memory bus
- Completing a memory access may take many cycles of the CPU clock
- In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing

Speed of access

- One of the Speed judgement parameters: number of CPU clock cycles needed to perform memory operation
- CPU is a faster than memory
- CPU registers are accessible in one CPU clock cycle
- But MM is accessed via a transaction on memory bus
- This access takes many CPU clocks to complete
- **Result?** : **CPU has program but not the data to complete instruction execution** i.e. **CPU stalling**

- **Remedy: add fast memory between the CPU and main memory called <u>cache</u>**

Speed of access

- As the data are needed always for instruction execution, frequent memory stalls are bottleneck
- Solution : add a faster memory between CPU and main memory i.e. **cache memory**
- Cache memory: a buffer to accommodate the speed difference

Basic Hardware

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

- Apart from relative speed of accessing physical memory, we also must ensure correct operation.
- To protect the operating system from access by user processes
- To protect user processes from one another. This protection must be provided by the hardware.

Memory Access Protection

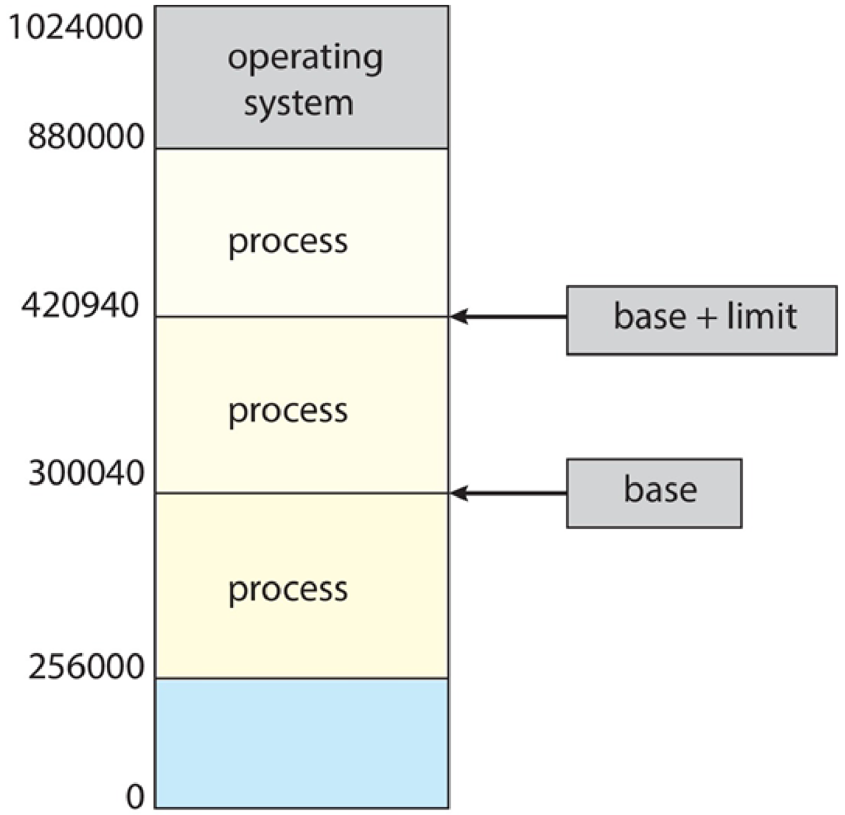
- OS must be protected from unauthorized access by user process
- User processes must be protected from each other

- Each process has range of legal addresses to access
- Need to ensure that a process can access only those addresses in its address space
- →This protection can be provided by using two registers; base and limit
- Base register and limit register ensure process can access only addresses within the legal range

  - We first need to make sure that each process has a separate memory space
  - Determine the range of legal addresses that the process may access
  - ensure that the process can access only these legal addresses

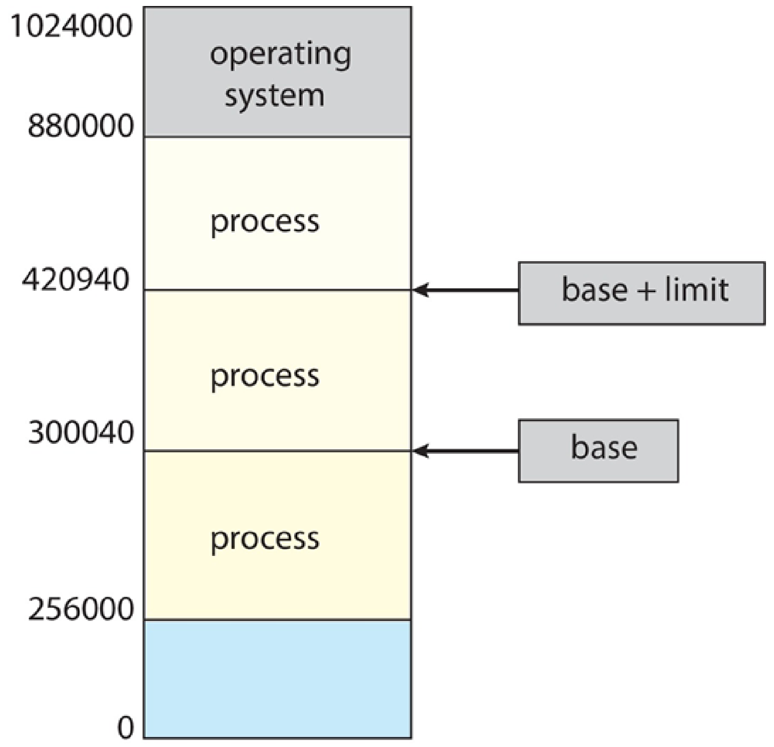
Memory Protection: Base and Limit Registers

- Base Register- Holds the smallest legal physical memory address
- Limit Register- specifies the size of the range



Memory Protection: Base and Limit Registers

- Example:
- if the base register holds 300040
- and the limit register is 120900,
- then the program can legally access all addresses from 300040 through 420939 (inclusive).

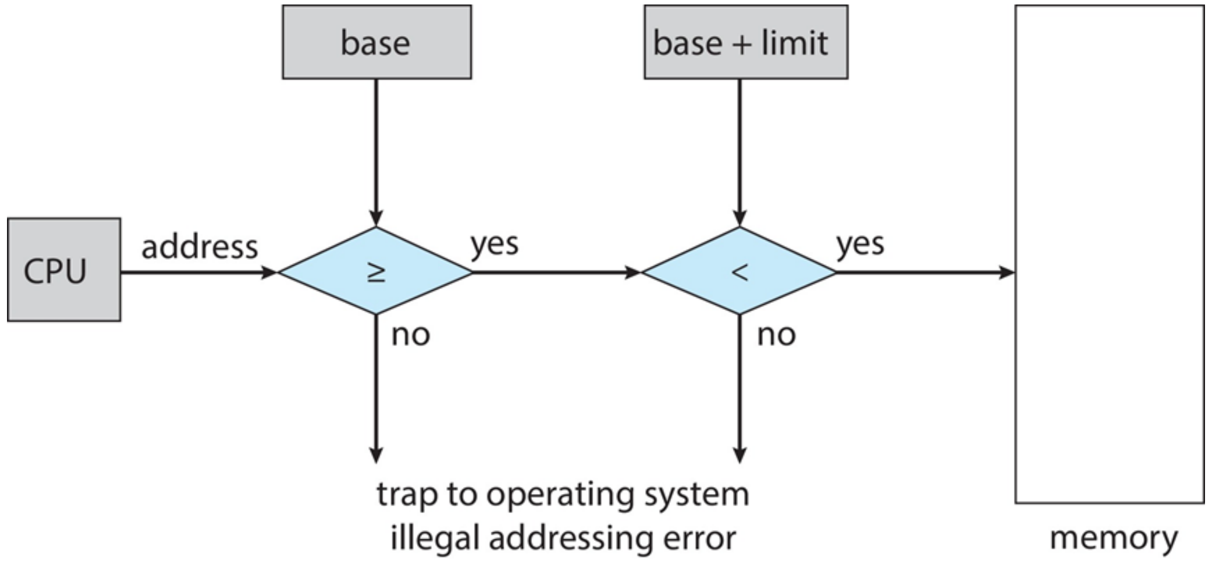


Memory Protection

- Protection of memory space is accomplished by the CPU hardware →compare every address generated in user mode with the registers
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory→ Results in a trap to the operating system, which treats the attempt as a fatal error
- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users

Memory Protection: H/W Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- the instructions to loading the base and limit registers are privileged

Memory Protection: H/W Address Protection

- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction
  - Since privileged instructions can be executed only in kernel mode, only the operating system can load the base and limit registers

- This allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

Address Binding

- Usually, a program resides on a disk as a binary executable file →To execute, the program must be brought into memory and placed within a process.
- Programs on disk, ready to be brought into memory to execute form an input queue
- The normal procedure is to select one of the processes in the input queue and to load that process into memory.
- As the process is executed, it accesses instructions and data from memory
- Eventually, the process terminates, and its memory space is declared available

Address Binding

- Computer address space starts at 00000
- But user process need not be loaded at 00000
  - The program is processed by various system softwares before being executed such as macro processor, compiler, linker, loader etc..
- Further, addresses represented in different ways at different stages of a program's life
- Source code addresses are usually symbolic (Eg-count)

Address Binding: Binding of Instructions and Data to Memory

- A compiler will typically bind these symbolic addresses 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 from one address space to another
- The binding of instructions and data to memory addresses i.e.
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**
  - **Load time**
  - **Execution time**

Address Binding: Binding of Instructions and Data to Memory

- Can be done at any step during execution based on available info
- If known at Compile time:
  - generates **absolute code** that resides at given memory.
  - Re-compile if the starting location changes
- If known at Load time but not at compile time:
  - Generates relocatable code
  - Final binding is delayed until load time
  - If starting address changes, only reload the user code to incorporate the changed value
- If known at Execution time
  - If process can be moved from one memory segment to another during execution
  - Binding is delayed until run time
  - Needs special hardware

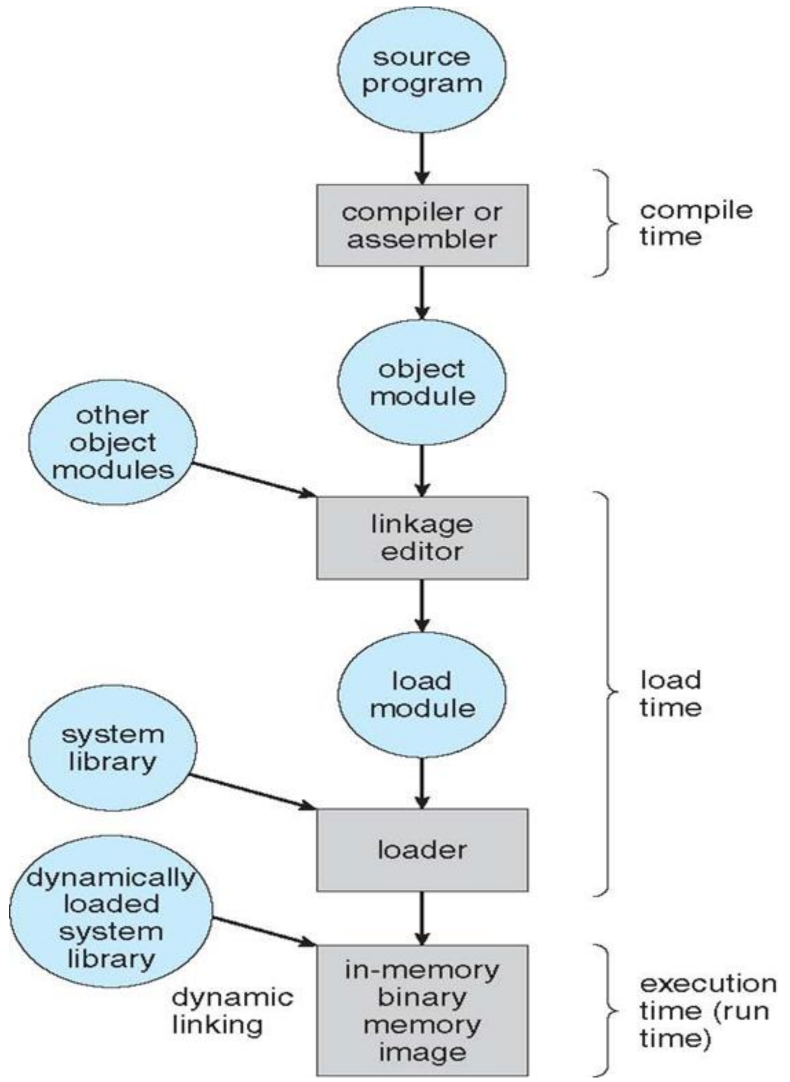
Address Binding: Binding of Instructions and Data to Memory

- **Compile time**:
  - **If you know at compile time where the process will reside in memory, then absolute code can be generated.**
  - **If memory location known a priori,**
  - For example, if you know that a user process will reside starting at location *R,* then the generated compiler code will start at that location and extend up from there.
- Must recompile code if starting location changes-
  - **If, at some later time, the starting location changes, then it will be necessary to recompile this code.**

Address Binding: Binding of Instructions and Data to Memory

- **Execution time**:
- If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.
- Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Logical Vs physical address space

- Logical address: An address generated by CPU
- Physical address: an address loaded into memory address register of memory
- The compile time and load time address-binding methods generate identical logical and physical address
- But, execution time binding results in different logical and physical address space
  - Logical address space: set of all addresses generated by a program
  - Physical address space: set of all physical addresses corresponding to all logical addresses

Logical Vs physical address space

- Logical address – An address generated by the CPU
  - also referred to as virtual address
- Physical address – Address seen by the memory unit
  - that is, the one loaded into memory address register of the memory
- Logical address space- the set of all logical addresses generated by a program
- Physical address space- the set of all physical addresses corresponding to these logical addresses

- **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 Management Unit (MMU)

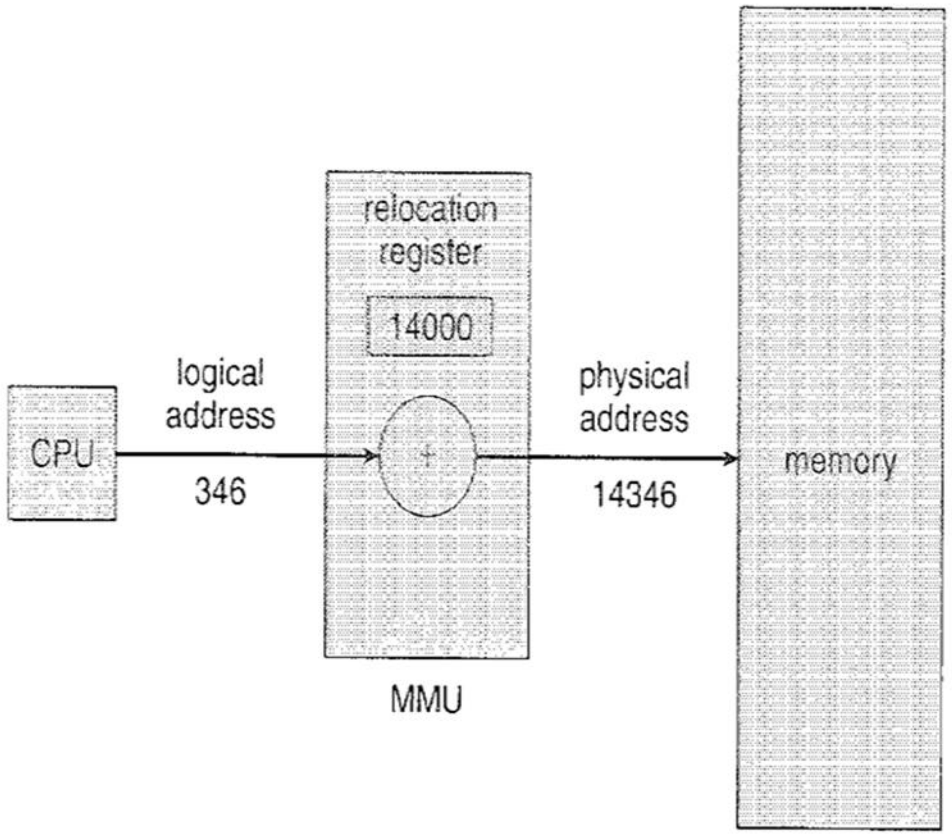


Figure 8.4 Dynamic relocation using a relocation register.

Memory Management Unit (MMU)

- The run-time mapping from virtual to physical addresses is done by a hardware device called the MMU
  - Many methods possible for this mapping
- MMU (memory management unit) maps virtual addresses to physical addresses
- Simple MMU scheme: generalization of base-register scheme
- Base register is called relocation register
- While sending to memory, relocation register value is added to every address generated by user process

Memory Management Unit (MMU)

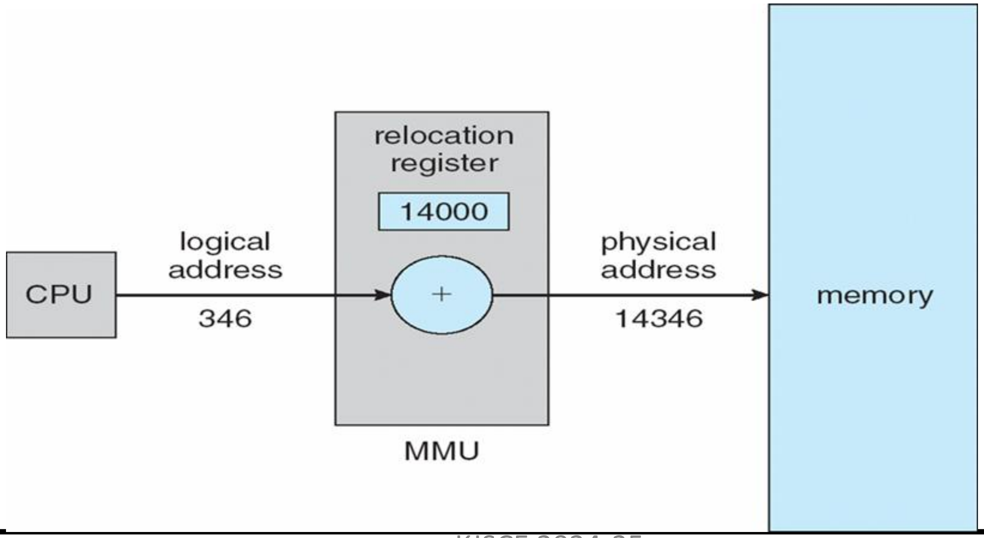
- Consider simple scheme –

■ Base register now called **relocation register**

  - The value in the relocation register is added to every address generated by a user process at the time it is sent to memory

  For example,

    - if the relocation is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000;
    - an access to location 346 is mapped to location 14346.



Memory Management Unit (MMU)

- The user program never sees the ‘real’ physical address.
- User program deals with logical addresses
- The memory mapping hardware converts logical addresses to physical addresses
- Logical addresses: Range is 0 to max
- Physical addresses: R+0 to R+max where R is base value

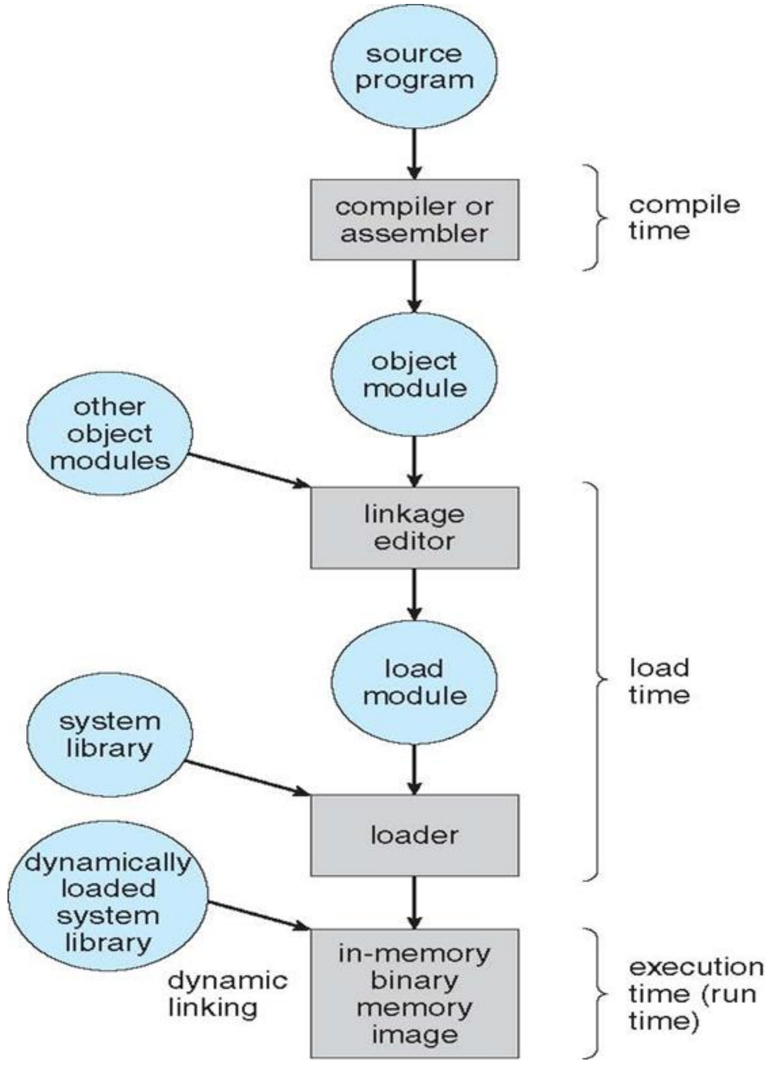
Dynamic Loading

- The entire program and all data of a process to be in physical memory for the process to execute.
- The size of a process has thus been limited to the size of physical memory
- To obtain better memory-space utilization, we can use Dynamic Loading

- Routine is not loaded until it is called; All routines kept on disk in relocatable load format

1. The main program is loaded into memory and is executed.
2. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
3. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change.
4. Then control is passed to the newly loaded routine

Dynamic Loading



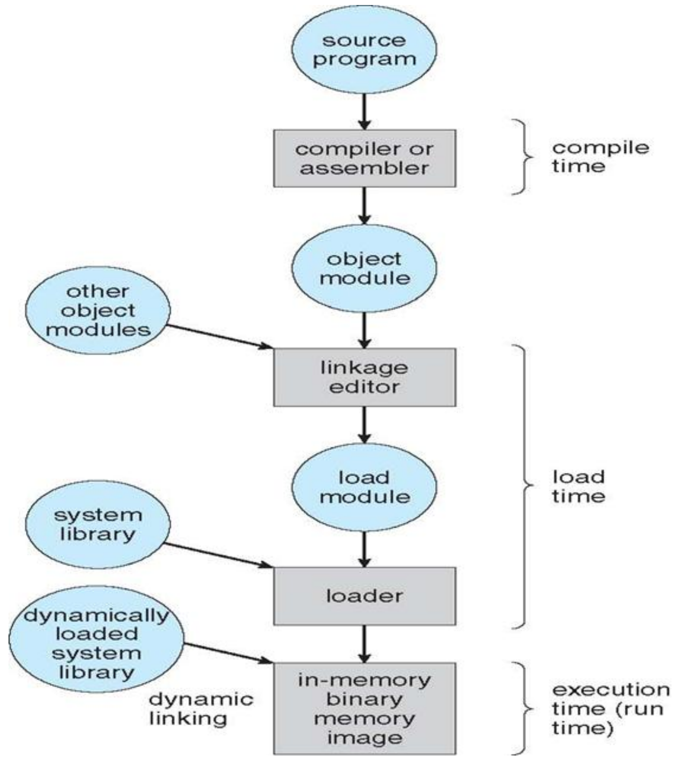
Dynamic Loading

- Advantage
- Unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases. Eg- Error Routines
- No special support from the operating system is required

- Implemented through program design-
  - It is the responsibility of the users to design their programs to take advantage of such a method.
- Operating systems may help the programmer,
  - by providing library routines to implement dynamic loading.

Dynamic Linking

- Static linking –
- System libraries and program code combined by the linker/loader into the binary program image
- Dynamic linking –linking postponed until execution time



Dynamic Linking

- This feature is usually used with system libraries, such as language subroutine libraries

- Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image
- This requirement wastes both disk space and main memory

- A stub is included in the image for each library routine reference
- The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present

Dynamic Linking

- When the stub is executed,

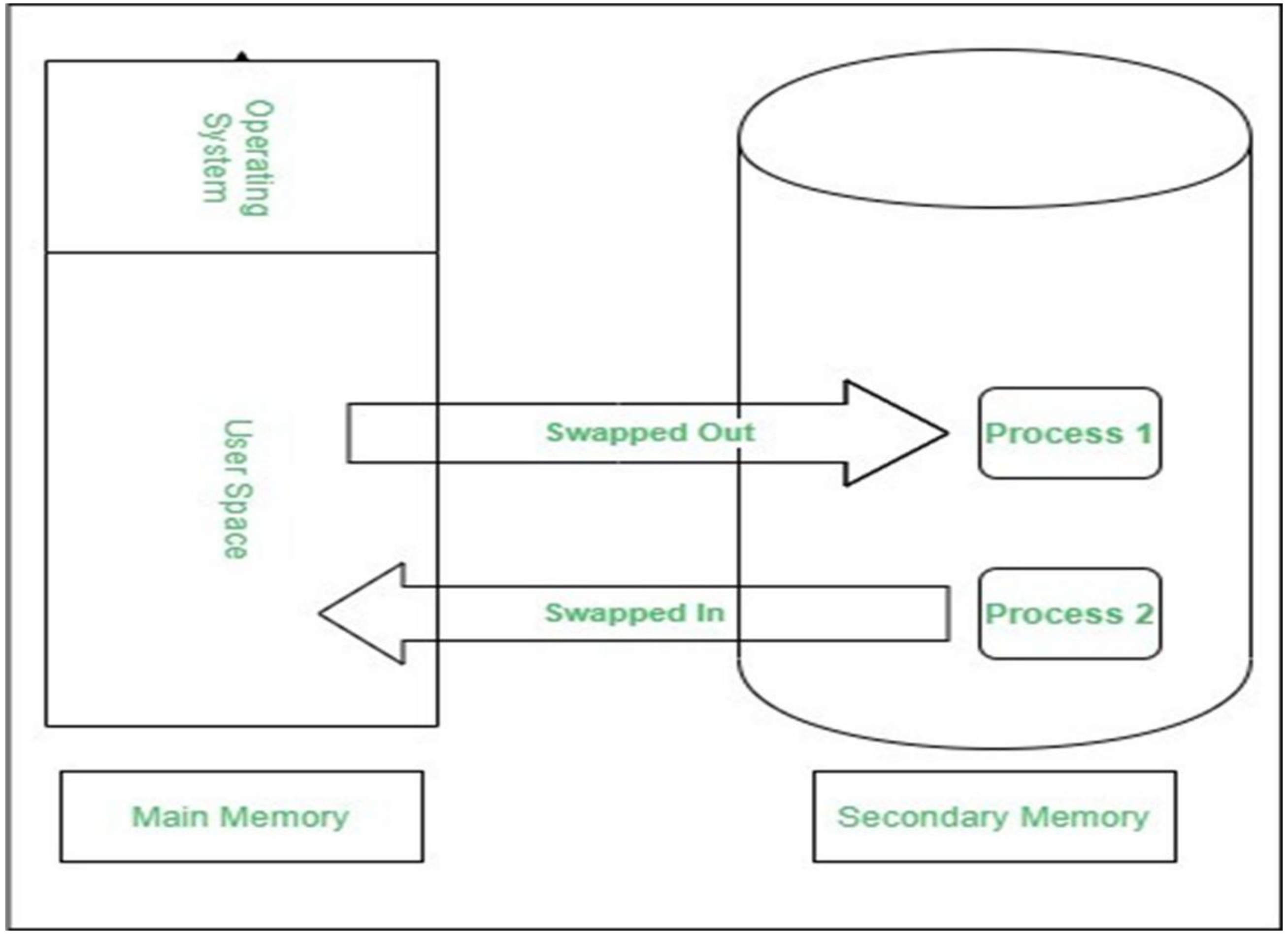
1. It checks to see whether the needed routine is already in memory
2. If it is not, the program loads the routine into memory
3. Either way, the stub replaces itself with the address of the routine and executes the routine
4. Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking

Dynamic Linking

- This feature can be extended to library updates (such as bug fixes).
- A library may be replaced by a new version,
- All programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library.

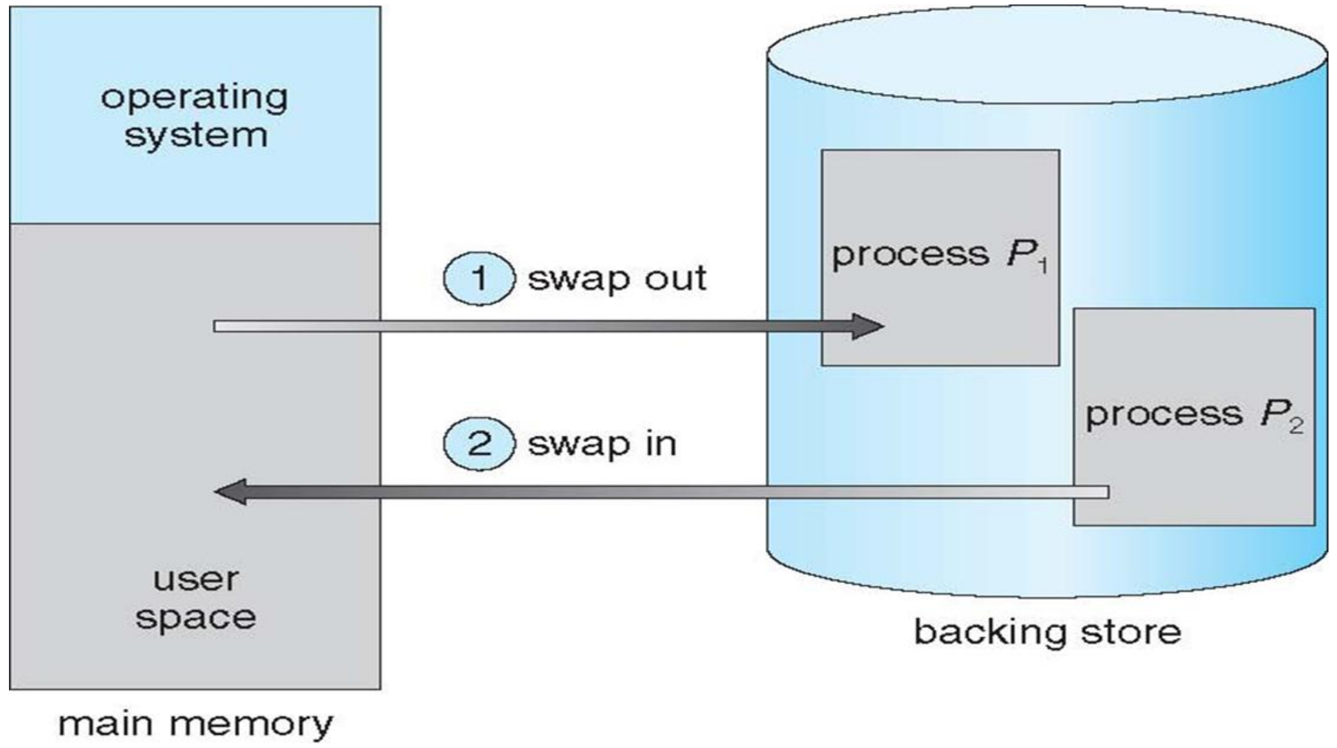
Swapping

- Swapping is a process of swapping a process temporarily to a secondary memory from the main memory



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 use
- must provide direct access to these memory images

Swapping

- Example:
- Assume a multiprogramming environment with a round-robin CPU-scheduling algorithm.

1. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed
2. In the meantime, the CPU scheduler will allocate a time slice to some other process in memory.
3. When each process finishes its quantum, it will be swapped with another process.
4. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU.

Swapping

- Roll out, roll in –
- A variant of this swapping policy is used for priority-based scheduling algorithms.

1. If a higher-priority process arrives and wants service,
2. The memory manager can swap out the lower-priority process and then load and execute the higher-priority process.
3. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued.
4. This variant of swapping is sometimes called roll out, roll in.

Swapping

- 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

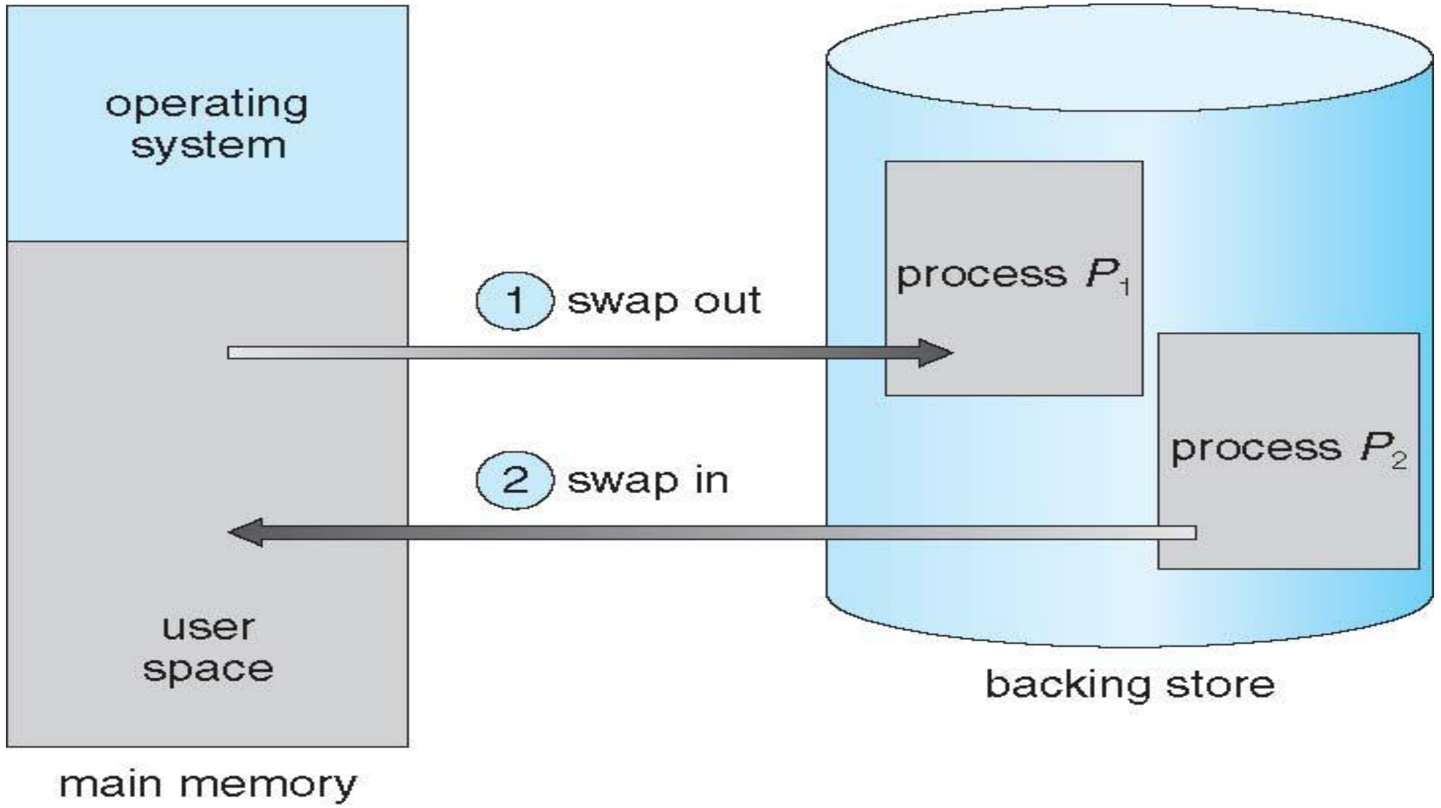
- Does the swapped out process need to swap back in to same physical addresses?
- Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously
- The restriction is dictated by the method of address binding
- If binding is done at assembly/Compile or load time, then the process cannot be easily moved to a different location
- if execution time binding is being used, then a process can be swapped into a different memory space as the physical addresses are computed at execution time

Dispatcher and Swapper

- Swapping requires a backing store.
- The backing store is commonly a fast disk, large enough to accommodate copies of all memory images for all users
  - Provides direct access to these memory images
- The system maintains a ready queue of all processes whose memory images are in memory and are ready to run
- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher
- The dispatcher checks to see whether the next process in the queue is in memory.
- If it is not, and if there is no free memory region, the swapper swaps out a process currently in memory and swaps in the desired process
- It then reloads registers and transfers control to the selected process

- The context-switch time in such a swapping system is fairly high.
- Assume a user process of 10 MB size and the backing store is a standard hard disk with a transfer rate of 40 MB per second. The actual transfer of the 10-MB process to or from main memory takes:
  - 10000 KB/ 40000 KBps =1/4 sec = 250 ms
- Assuming that no head seeks are necessary, and assuming an average latency of 8 milliseconds, the swap time is 258 milliseconds.
- Considering both swap out and swap in, the total swap time = 516 milliseconds.
- For efficient CPU utilization, the execution time for each process should be long relative to the swap time.
- Thus, in a round-robin CPU-scheduling algorithm, for example, the time quantum should be substantially larger than 0.516 seconds.

Schematic View of Swapping



Memory Management Techniques: Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- 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

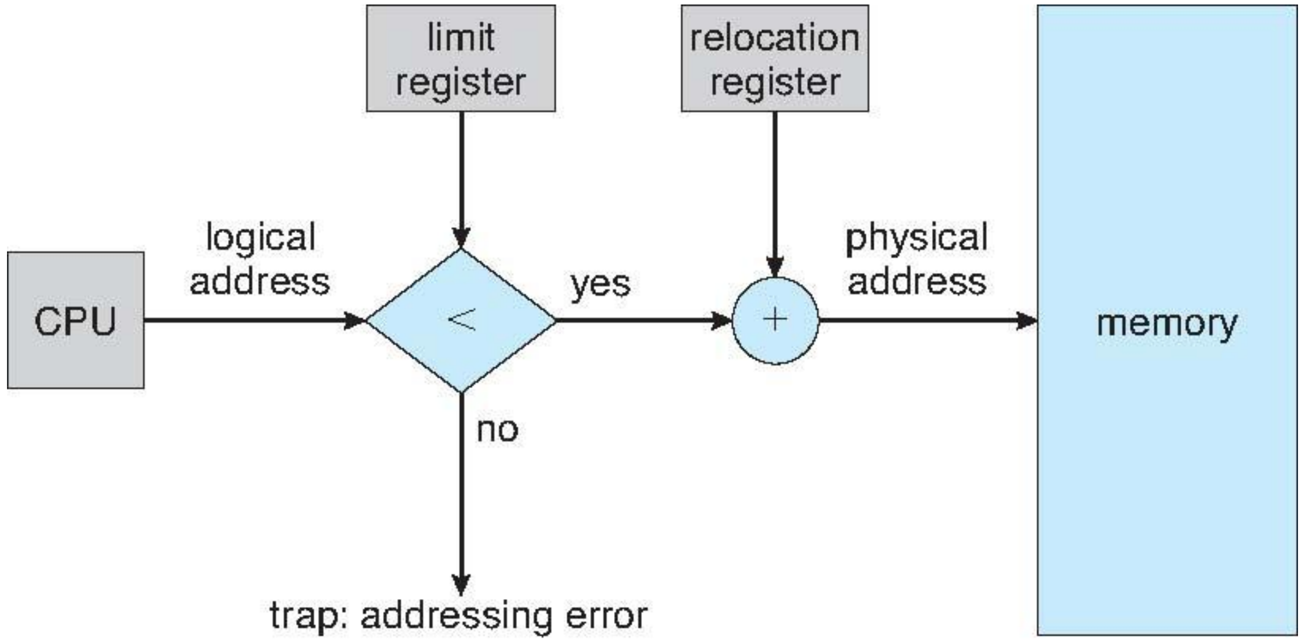
Memory Management Techniques: Contiguous Allocation

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
- Base 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

Contiguous Allocation: Hardware Support for Relocation and Limit Registers

1) **Each logical address must be less than the limit register;**
2) **The MMU maps the logical address *dynamically* by adding the value in the relocation register.**
3) **This mapped address is sent *to* memory**

(for example, relocation= 100040 and limit= 74600).



Memory Management Techniques

- **Contiguous**: In contiguous memory allocation, each process is contained in a single contiguous section of memory

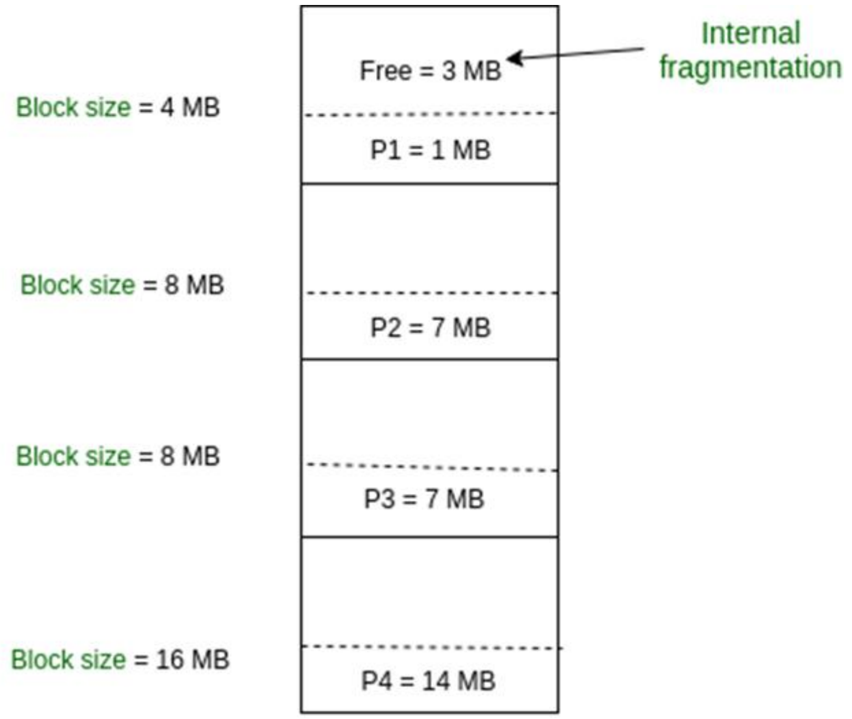
- **Non-Contiguous:** process allocated non-contiguous section of memory

- In Contiguous Technique, executing process must be loaded entirely in the main memory
- The contiguous Technique can be divided into:
  - Fixed (or static) partitioning
  - Variable (or dynamic) partitioning

Memory Management: Fixed Partitioning

- Also known as static partitioning
- memory allocation technique used to divide the physical memory into fixed-size partitions or regions, each assigned to a specific process or user
- No of Partitions are fixed
  - the number of partitions (non-overlapping) in memory is fixed but the size of each partition may or may not be the same
- Partitions are made before execution or during system configure
- Each partition remains dedicated to a specific process until it terminates or releases the partition

Memory Management: Fixed Partitioning



Fixed size partition

- **Disadvantages**:
- Can lead to internal fragmentation, where memory in a partition remains unused
- This can happen when the process’s memory requirements are smaller than the partition size, leaving some memory unused

Memory Allocation: Fixed Partitioning

- Each partition may contain exactly one process
  - Thus, the degree of multiprogramming is bound by the number of partitions
- In this Multiple partition method, when a partition is free, a process is selected from the *input queue* and is loaded into the free partition
- When the process terminates, the partition becomes available for another process.
- This method was originally used by the IBM OS/360 operating system (called MFT); it is no longer in use

Memory Allocation: Fixed Partitioning

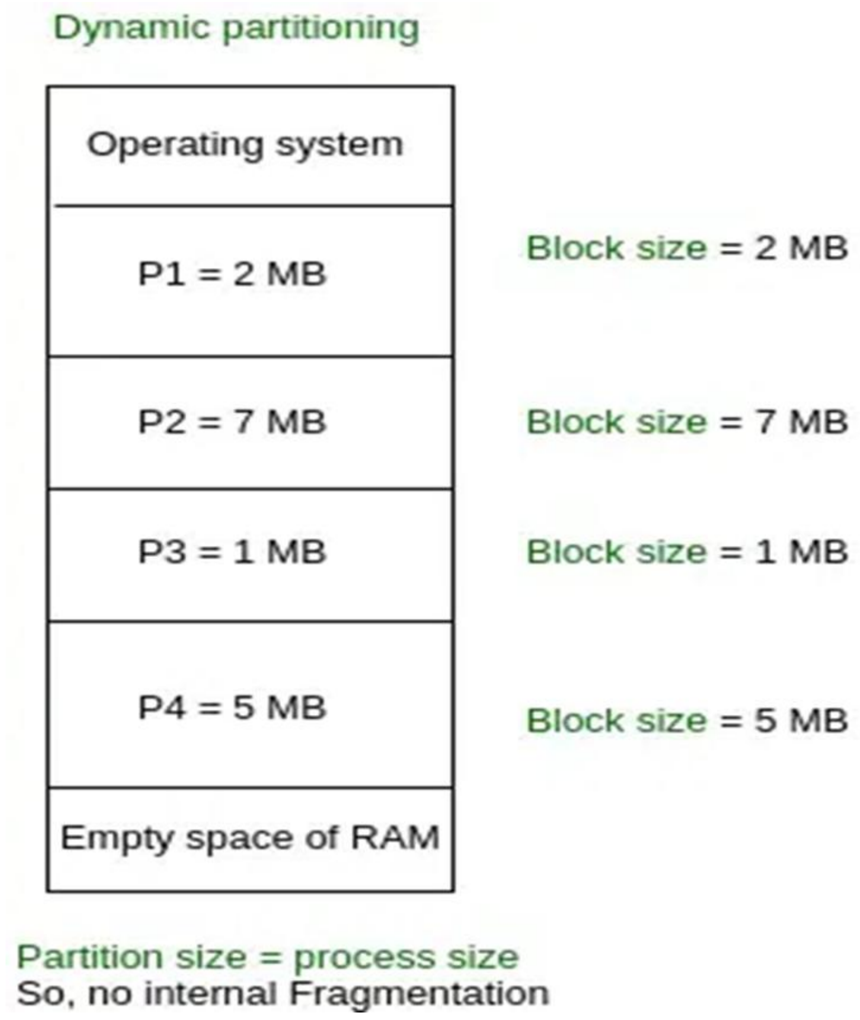
- As processes enter the system, they are put into an input queue
- At any given time, we have a list of available block sizes and an input queue
- The operating system can order the input queue according to a scheduling algorithm.
    - Process selected from input queue is allocated memory from a hole large enough to accommodate it
    - The operating system can wait until a large enough block is available,
    - or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met

    - **Hole** – block of available memory

Memory Allocation: Variable (or Dynamic) Partitioning

- In contrast with fixed partitioning, partitions are not made before the execution or during system configuration
- Initially, memory is empty and partitions are made during the run-time according to the process's need instead of partitioning during system configuration
- Size of the partition will be equal to the incoming process
  - Variable-partition sizes for efficiency (sized to a given process' needs)
- The partition size varies according to the need of the process so that internal fragmentation can be avoided to ensure efficient utilization of RAM
- No of partitions are not fixed; depends on the number of incoming processes and the Main Memory's size

Memory Allocation: Variable (or Dynamic) Partitioning



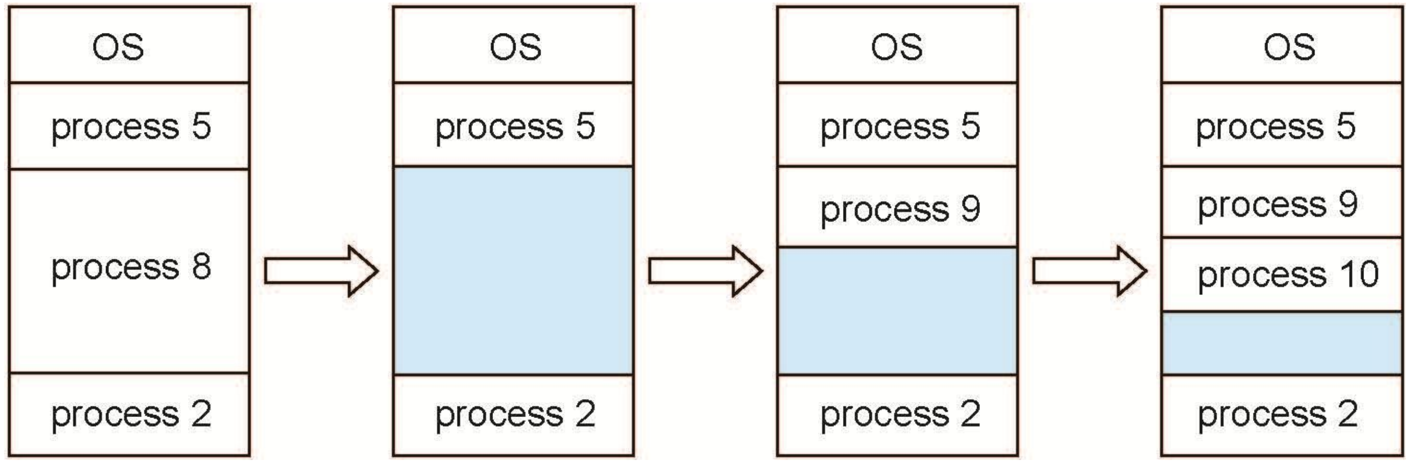
- The operating system keeps a table indicating which parts of memory are available and which are occupied
- Initially, all memory is available for user processes and is considered one large block of available memory-**a Hole.**

  "Hole – block of available memory; holes of various size are scattered throughout memory"

- **Eventually, memory contains a set of holes of various sizes.**

Memory Allocation: Variable (or Dynamic) Partitioning – Multiple partition allocation

- 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)



Memory Allocation: Dynamic Storage-Allocation Problem

- How to satisfy a request of size n from a list of free holes?
  - First-fit
  - Best-fit
  - Worst-fit

  - **First-fit:**
  - Allocate the first hole that is big enough
  - Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended
  - Stop searching as soon as we find a free hole that is large enough

Memory Allocation: Dynamic Storage-Allocation Problem

- Best-fit:
  - Allocate the smallest hole that is big enough;
  - must search entire list, unless ordered by size
  - This strategy produces the smallest leftover hole

- Worst-fit:
  - Allocate the largest hole;
  - must also search entire list unless it is sorted by size
  - This strategy produces the largest leftover hole which may be more useful than the smaller leftover hole from a best-fit approach

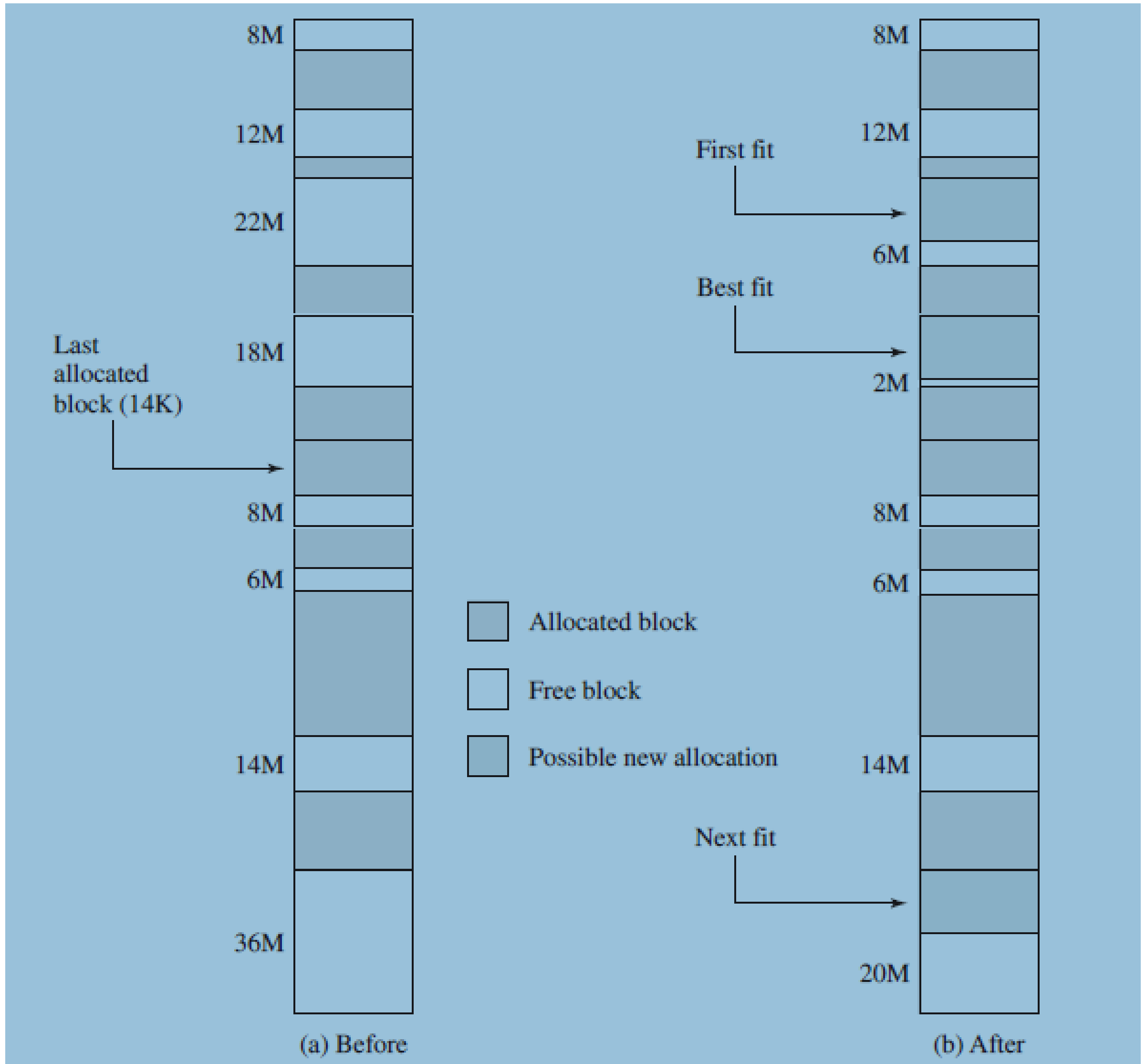


Figure 7.5 **Example Memory Configuration before and after Allocation of 16-Mbyte Block**

Memory Allocation: Dynamic Storage-Allocation Problem

- 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

- Simulations have shown that:
  - Both first fit and best fit are better than worst fit in terms of speed and storage utilization
  - Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

Memory Management

Consider four memory partitions of size 18KB, 16KB, 20KB, 8KB and 30KB. These partitions need to be allocated to the processes with their requirements as given in the table below

Perform memory allocation using the **First Fit, Best Fit, and Worst Fit** allocation methods

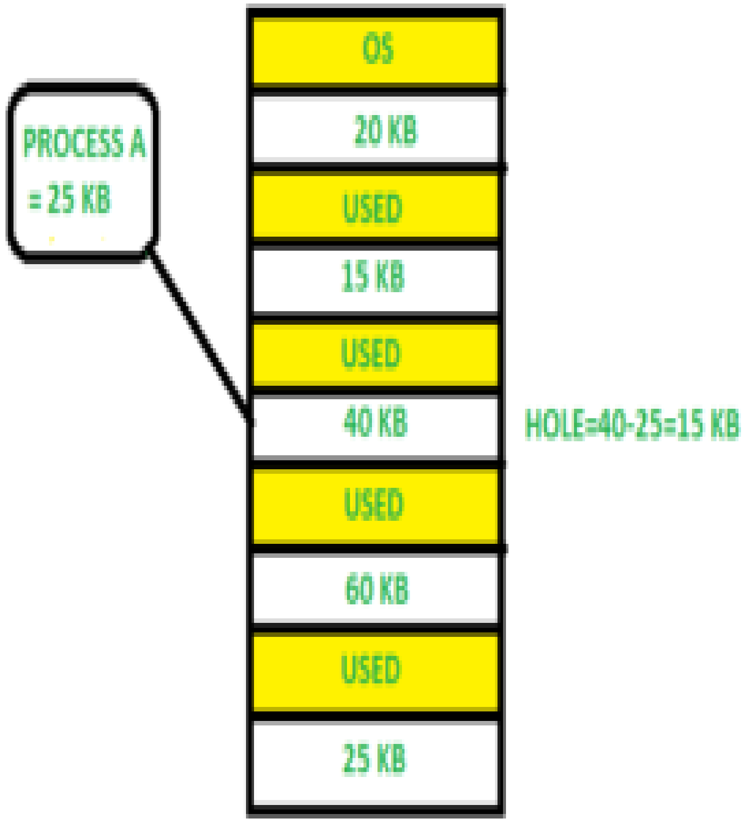
Process	Memory Required
P1	20 KB
P2	14 KB
P3	18 KB
P4	8 KB

Memory Allocation: Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces → memory fragmentation
- Memory fragmentation can be-
  - External Fragmentation
  - Internal Fragmentation
- External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference in memory is internal to a partition, but not being used- Unused Memory Internal to a partition
- First fit analysis reveals that given N blocks allocated, 0.5 N blocks lost to fragmentation
  - 1/3 may be unusable -> 50-percent rule

Memory Allocation: Fragmentation

- External Fragmentation –
  - Total memory space exists to satisfy a request, but it is not contiguous
  - Storage is fragmented into a large number of small holes.
  - fragmentation problem can be severe
  - In the worst case,
  - A block of free (or wasted) memory between every two processes
    - If all these small pieces of memory were in one big free block instead, we might be able to run several more processes



Memory Allocation: 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
- If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address
- I/O problem
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers

Memory Allocation: Fragmentation…

- The simplest compaction algorithm is
  - To move all processes toward one end of memory;
  - all holes move in the other direction,
  - producing one large hole of available memory
  - This scheme can be expensive

- Another possible solution-
  - To permit the logical address space of the processes to be non contiguous,
  - thus allowing a process to be allocated physical memory wherever such memory is available.

Non-Contiguous Memory Allocation

- In this technique, each process is allocated a series of non-contiguous blocks of memory that can be located anywhere in the physical memory

- Paging and Segmentation are the two ways that allow a process's physical address space to be non-contiguous
- It has the advantage of reducing memory wastage but it increases the overheads due to address translation
- Slows the execution of the memory because time is consumed in address translation