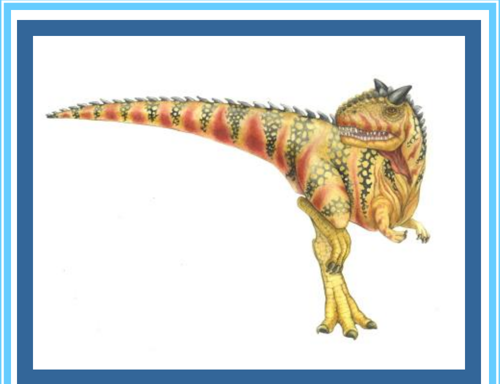
Memory Management



- **Slides mostly borrowed from Silberschatz & Galvin**
  - With occasional modifications from us

Memory Management: Topics

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

Memory Management: Topics

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

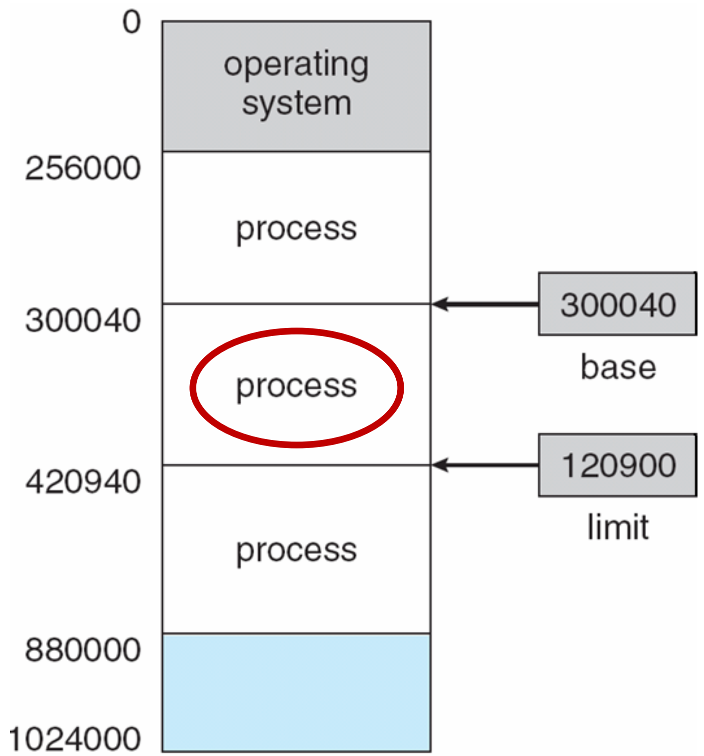
Background

- Main memory and registers are the only storage that the CPU can access directly
  - Register access much faster than main memory access
  - **Cache** sits between main memory and CPU registers

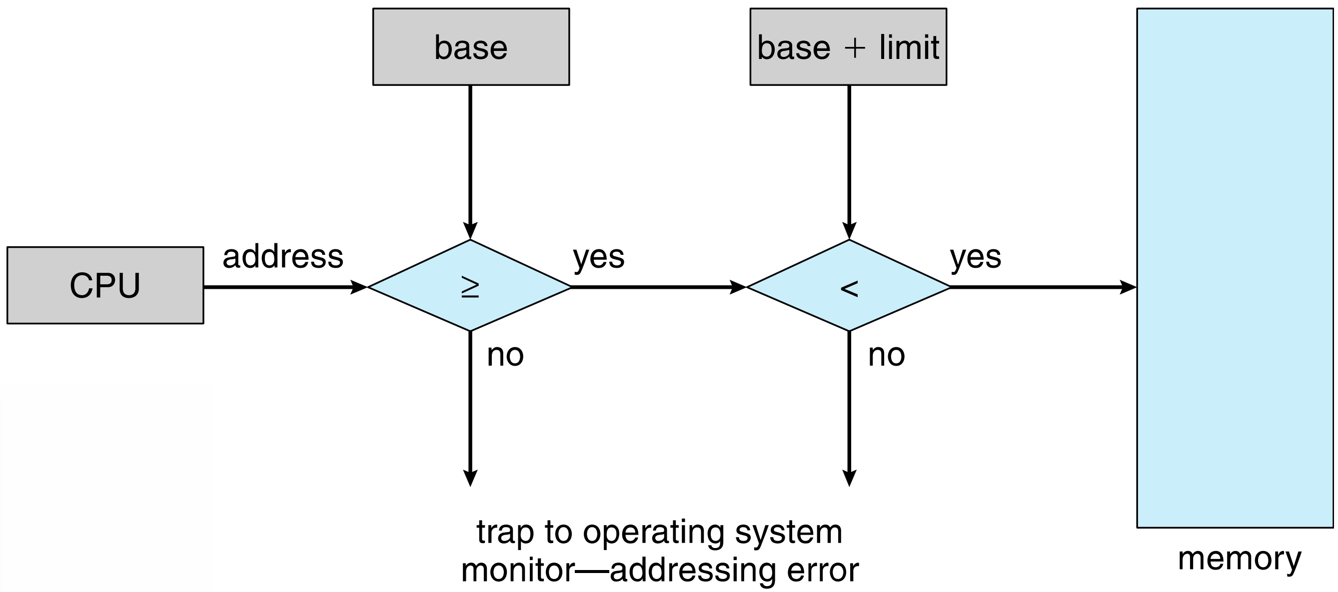
- By "Memory", we refer to "Main memory"
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Protection of memory required to ensure correct operation of various processes that share the memory (see next slide)

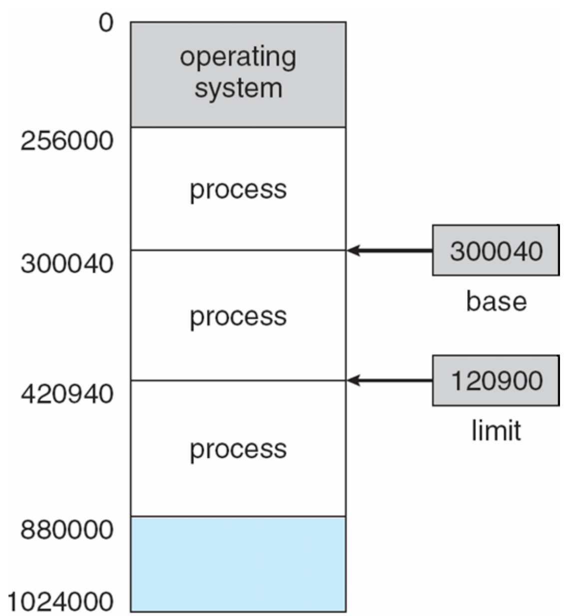
Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space of a user process
- CPU must check every memory access generated by a process in user mode to be sure it is between base and limit for that process



Hardware Address Protection



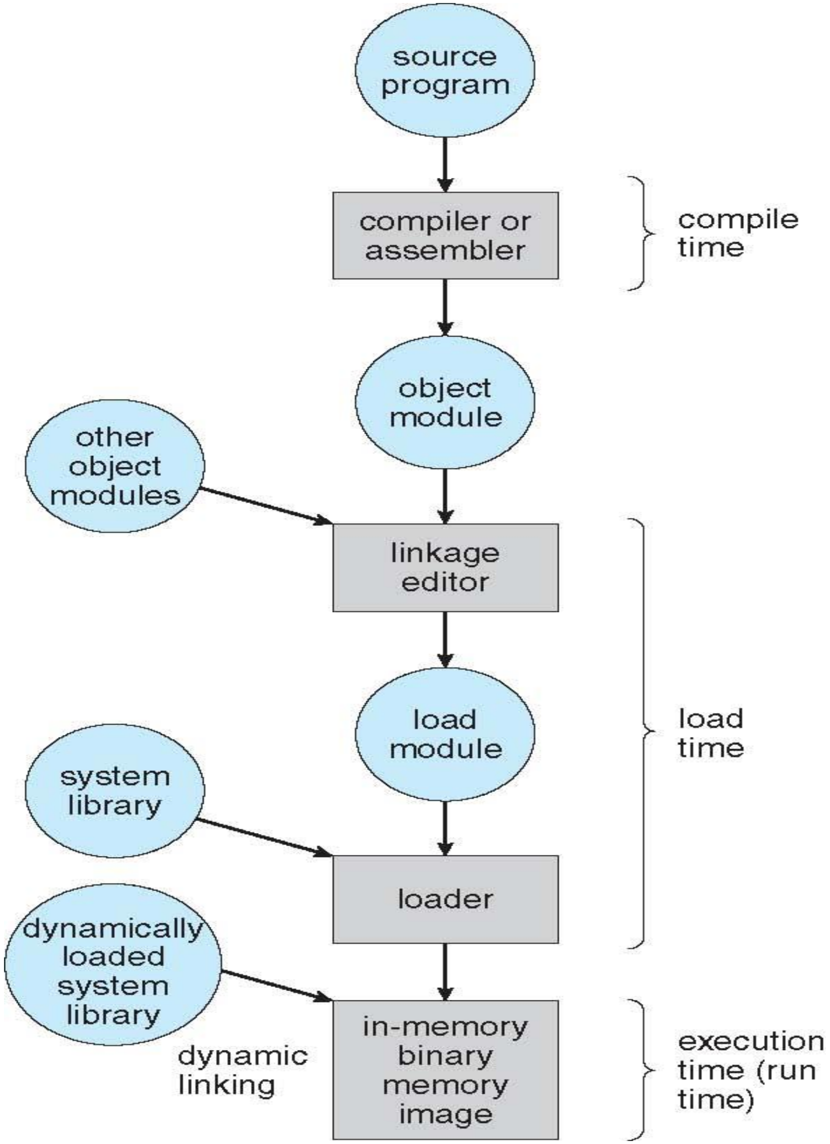


Address Binding

- Address Binding: mapping addresses from one address space to another
- Why is binding needed?

- Programs on disk, ready to be brought into memory to execute
  - Without support, must be loaded into address 0000
  - But kernel usually resides starting from the address 0000
  - Alternative: need a mechanism by which user processes can be stored in different areas of the memory
- We want to write programs without worrying about where in memory our program will be loaded (during execution)
  - Need mechanisms to map addresses generated by a user program/process to actual physical memory addresses

Multistep Processing of a User Program



- Addresses are represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - E.g., "14 bytes from beginning of this module"
  - Linker or loader will **bind** relocatable addresses to absolute addresses
    - E.g., 74014

Binding of Instructions and Data to Memory

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

Code can be moved around in memory without any problem.

Requires base register, relative addressing, etc.

Logical vs. Physical Address Space

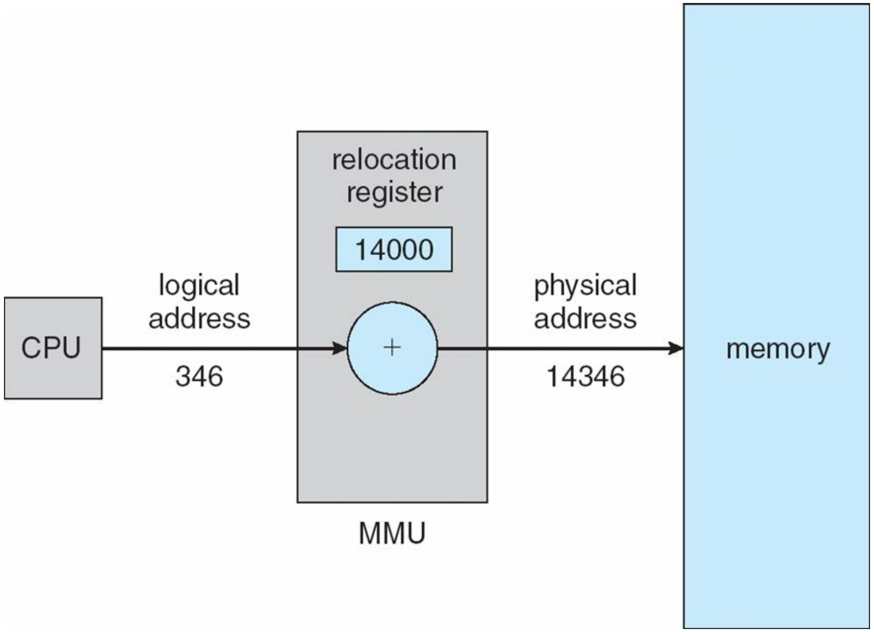
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical (virtual) and physical addresses
  - Are the same in compile-time and load-time address-binding schemes
  - Differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program (**i.e., the CPU**)
- **Physical address space** is the set of all physical addresses generated by a program (**i.e., accessed in main memory**)

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- To start, consider a 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
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Dynamic relocation using a relocation register

- Function 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



Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- **Dynamic linking** – linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
    - Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
    - If not in address space, add to address space

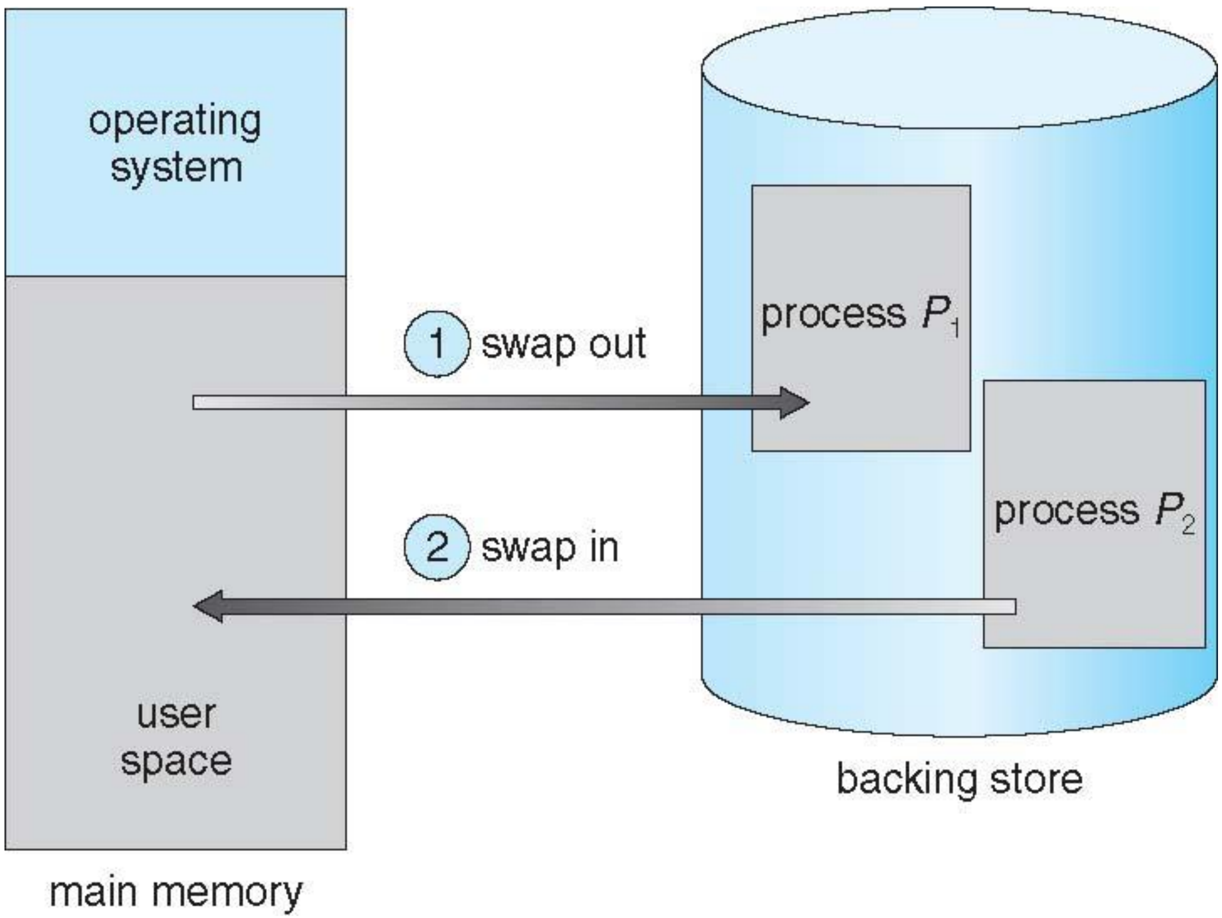
Memory Management: Topics

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

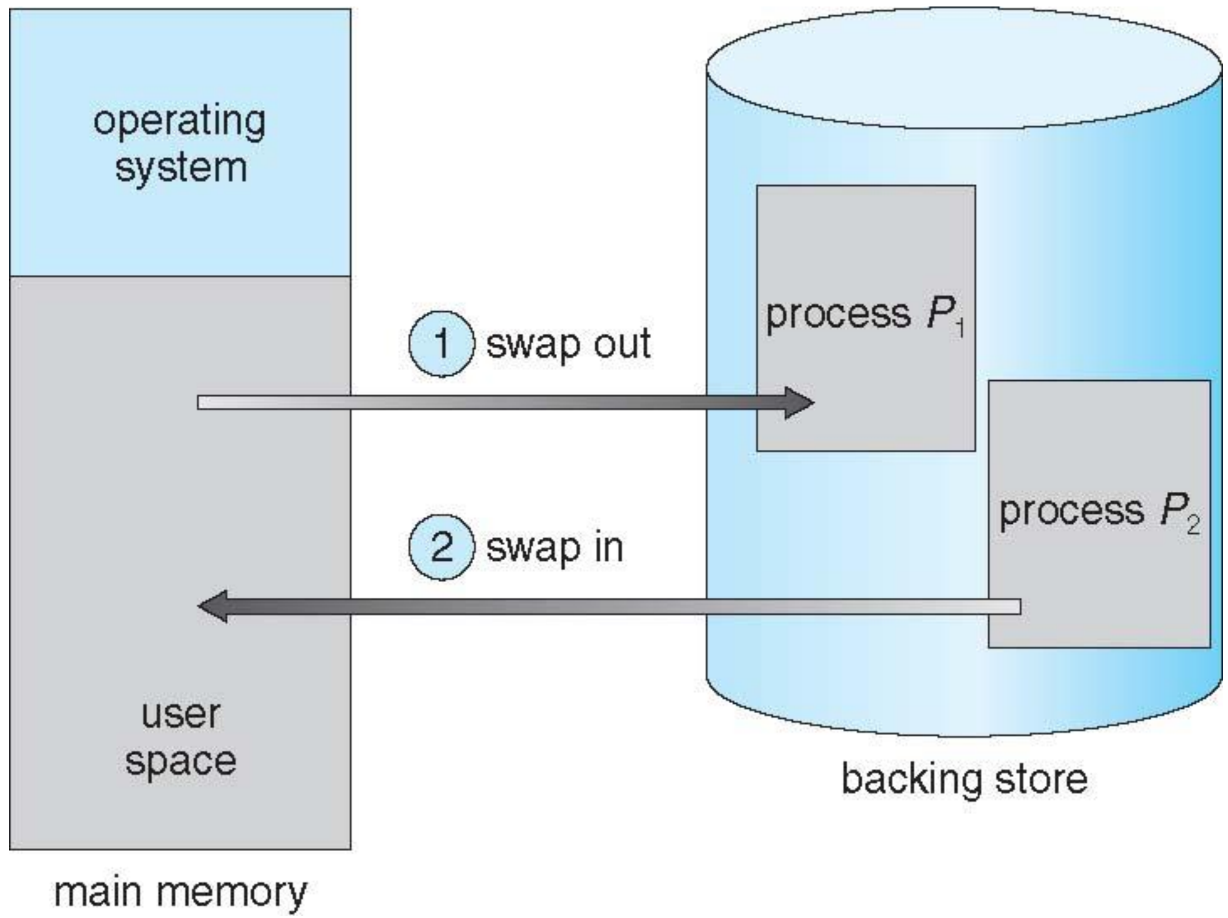
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all user processes
- 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

Schematic View of Swapping



Schematic View of Swapping



Note: Total physical memory space of processes can exceed the size of physical memory

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
  - Plus swap in of same sized process
  - Total context switch time of 4000ms (4 seconds)

- Can reduce the time if reduce size of memory swapped – by knowing how much memory really being used

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - If a process has 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

Memory Management: Topics

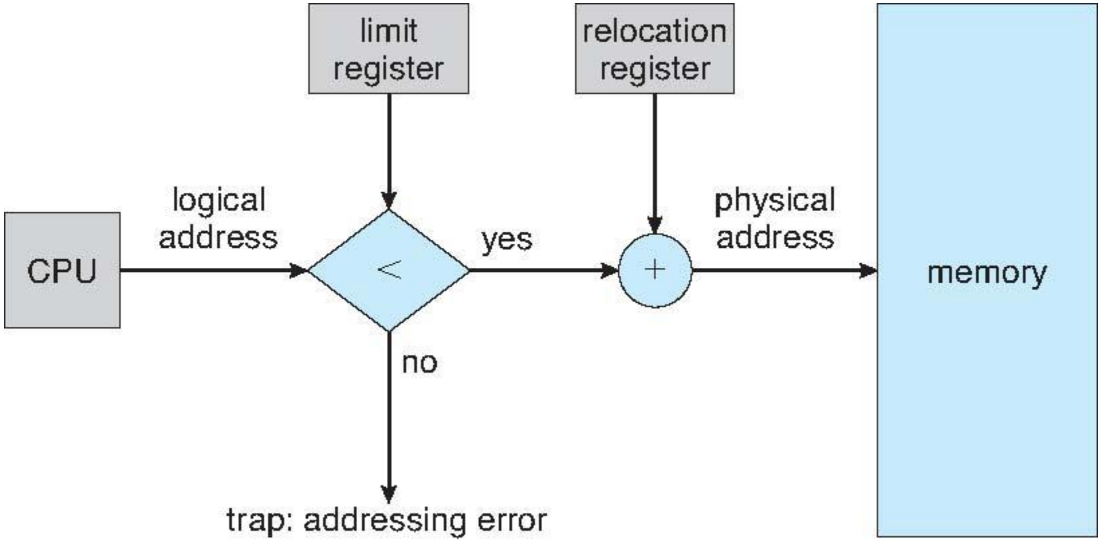
- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

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 is split into two **partitions**:
  a) Resident operating system, usually held in low memory with interrupt vector
  b) User processes then held in high memory
     - Each process contained in single contiguous section of memory

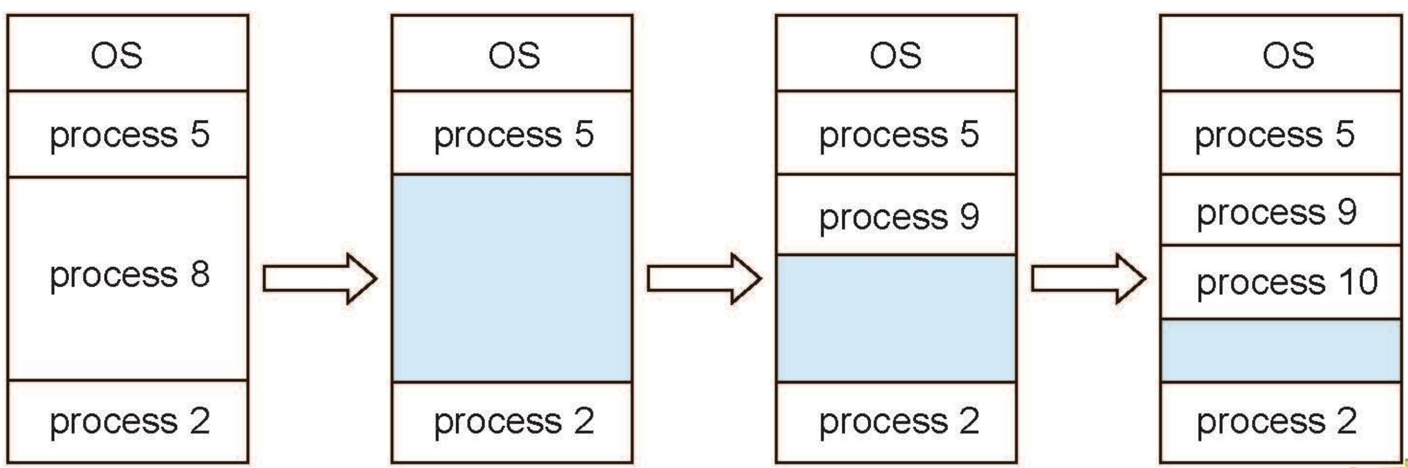
Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - **Base / relocation register** contains value of smallest physical address of a user process
  - **Limit register** contains range of logical addresses accessible by a user process – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Instructions to load these registers must be ***privileged***



Multiple-partition allocation

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **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?

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough
  - Benefit: Produces the smallest leftover hole
  - Cost: Must search entire list, unless ordered by size
- **Worst-fit**: Allocate the ***largest*** hole
  - Must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - Holes of sizes 10K, 15K and 25K exists
  - A process of size 35K arrives --- cannot be loaded

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
  - Allocated memory = 20K, but using only 18K
  - 2K wasted due to internal fragmentation

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory (**i.e. holes**) 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

Double buffering:

- Transfer I/O to kernel space
- Move from kernel space to user space

Memory Management: Topics till now

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