CS420: Operating Systems

Main Memory

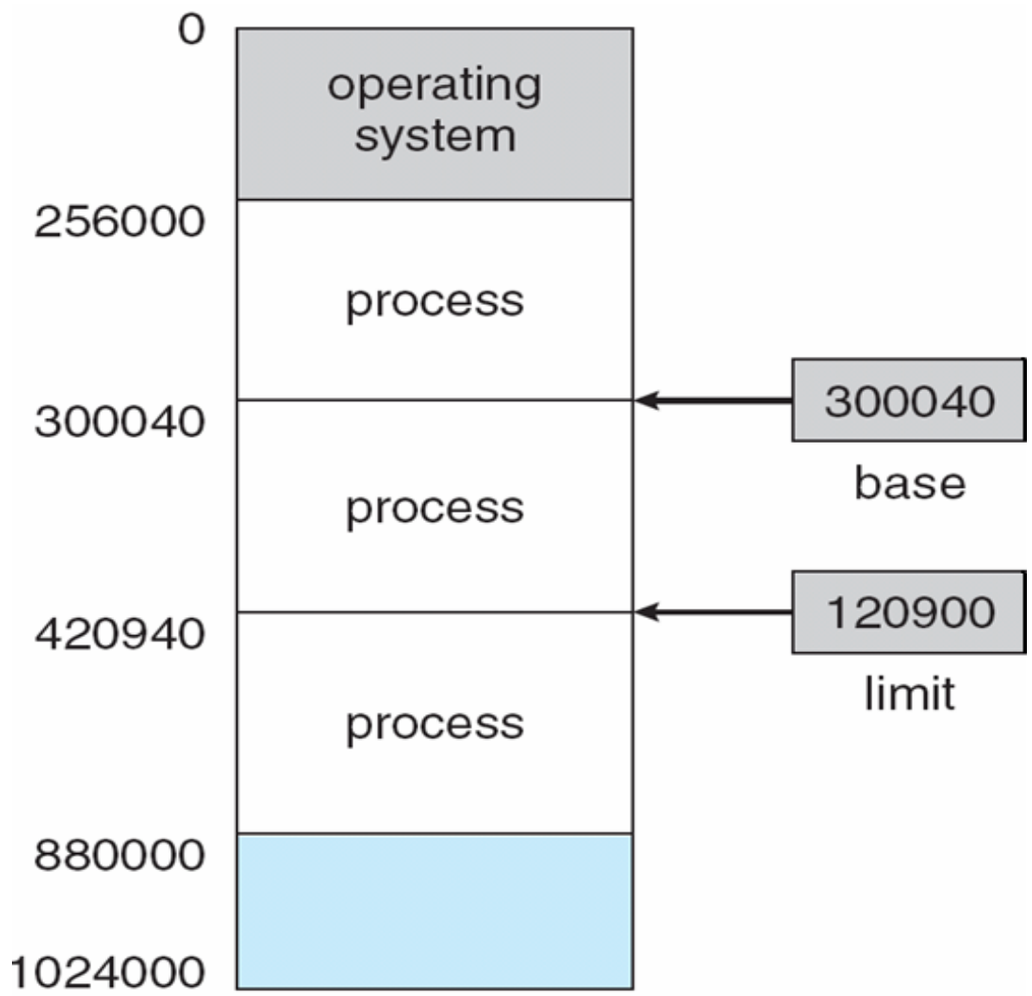
James Moscola
Department of Engineering & Computer Science
York College of Pennsylvania

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**
  - Register access takes one CPU clock cycle
  - Main memory access can take many CPU clock cycles
  - Cache sits between main memory and CPU registers
- **Memory unit only sees a stream of addresses, doesn't know if addresses contain instructions or data**
- **Protection of memory required to ensure correct operation**

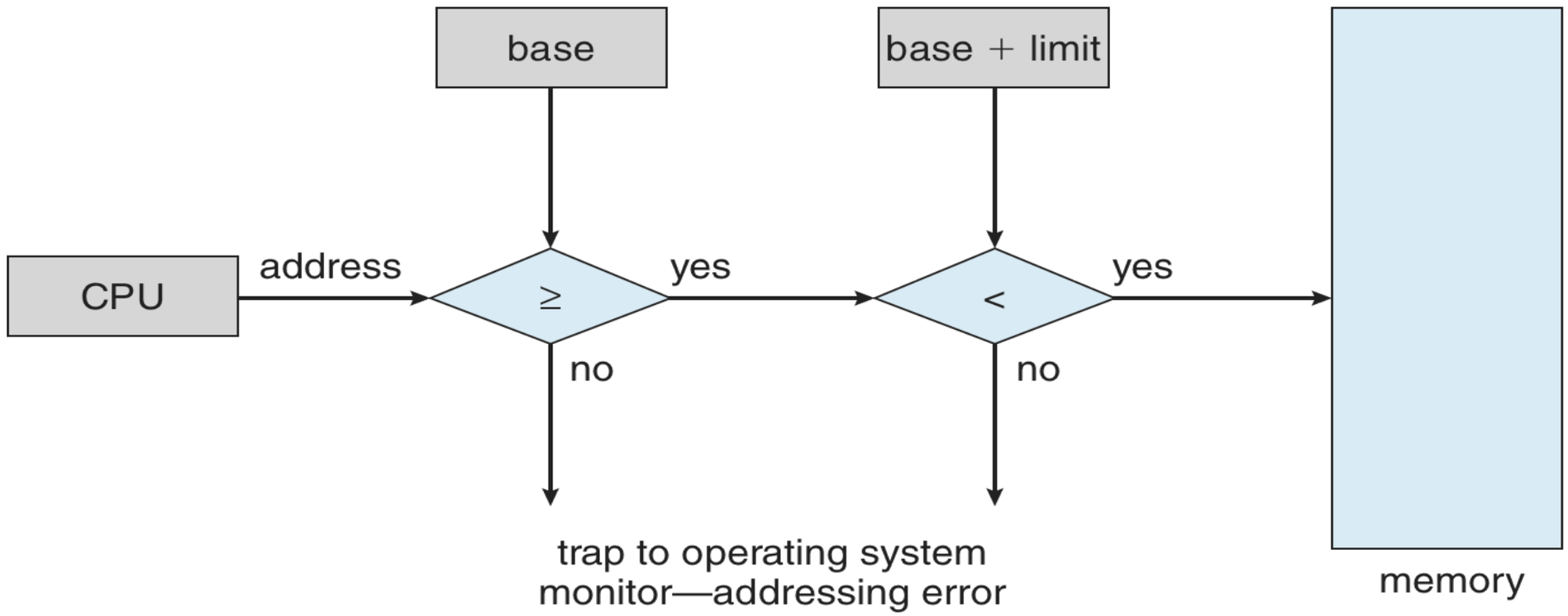
Base and Limit Registers

- **A pair of base and limit registers define the logical address space for each process**



Hardware Address Protection

- **Base and limit registers provide memory protection**
- **Prevent a process from accessing memory outside of its own memory space**
- **Only the operating system can change the contents of the base and limit registers (must be done in kernel mode)**

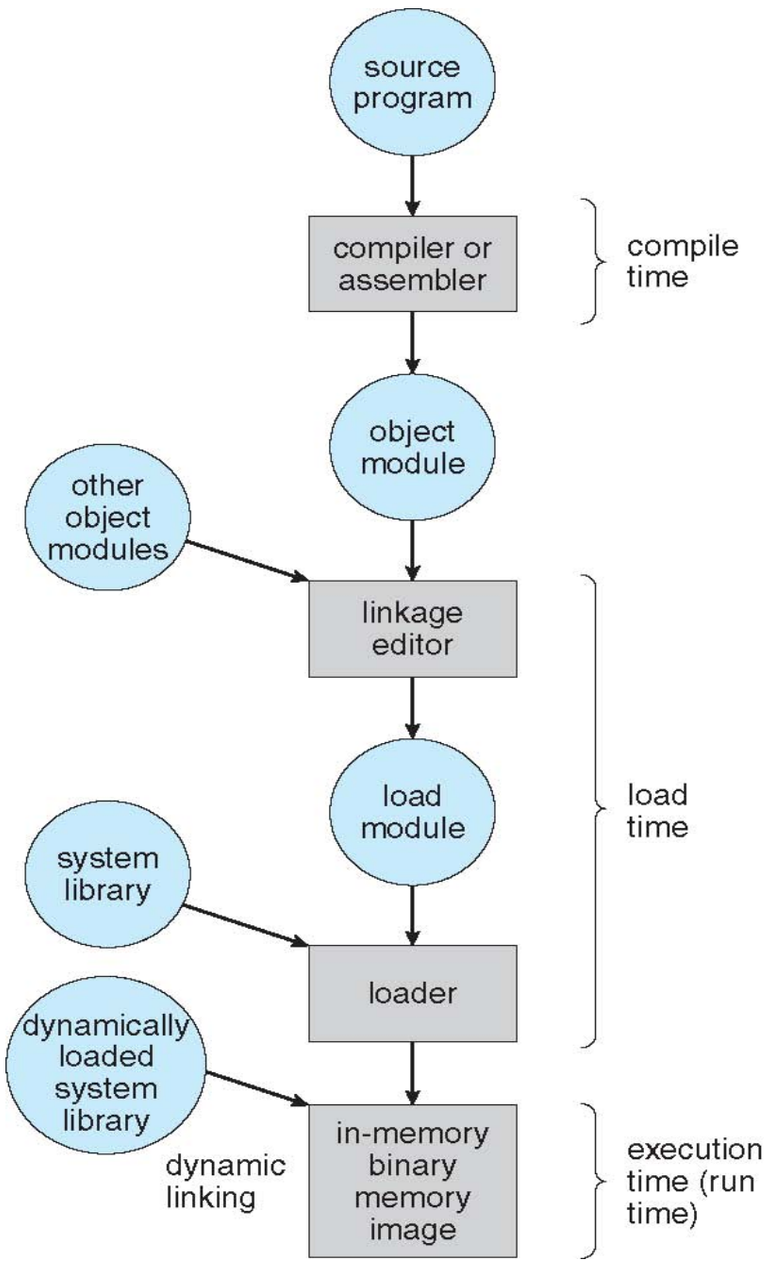


Address Binding

- **Not very convenient to have first user process physical address always at memory address 0x0000 (and subsequent processes following that)**
- **Addresses are represented differently at different stages of a program's life**
  - In source code addresses are usually symbolic (e.g. variables, function names)
  - In compiled code addresses bind to relocatable addresses (a relative address)
    - i.e. "8 bytes from beginning of this program module"
  - Linker or loader will bind relocatable addresses to absolute addresses when program is executed
    - i.e. address = 0x74000 + 0x00008 = 0x74008
      - program module starts at 0x74000
      - the 8th byte of the program is at 0x74008

Binding of Program Instructions and Data to Memory

- **Address binding of program 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)



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 (refers to physical memory in the machine)

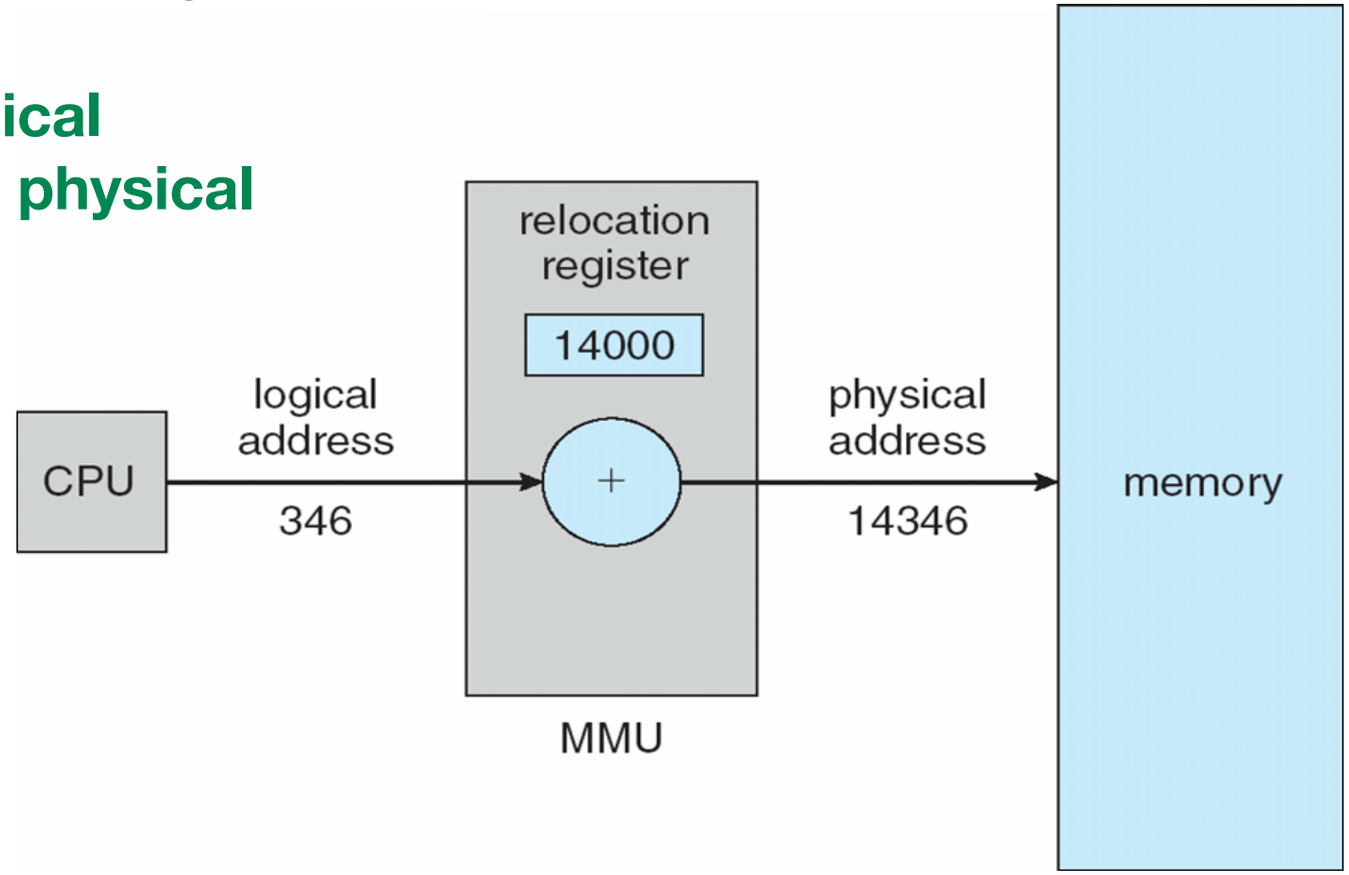
- **Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme**

- **Logical address space is the set of all logical addresses generated by a program**

- **Physical address space is the set of all physical addresses generated by a program**

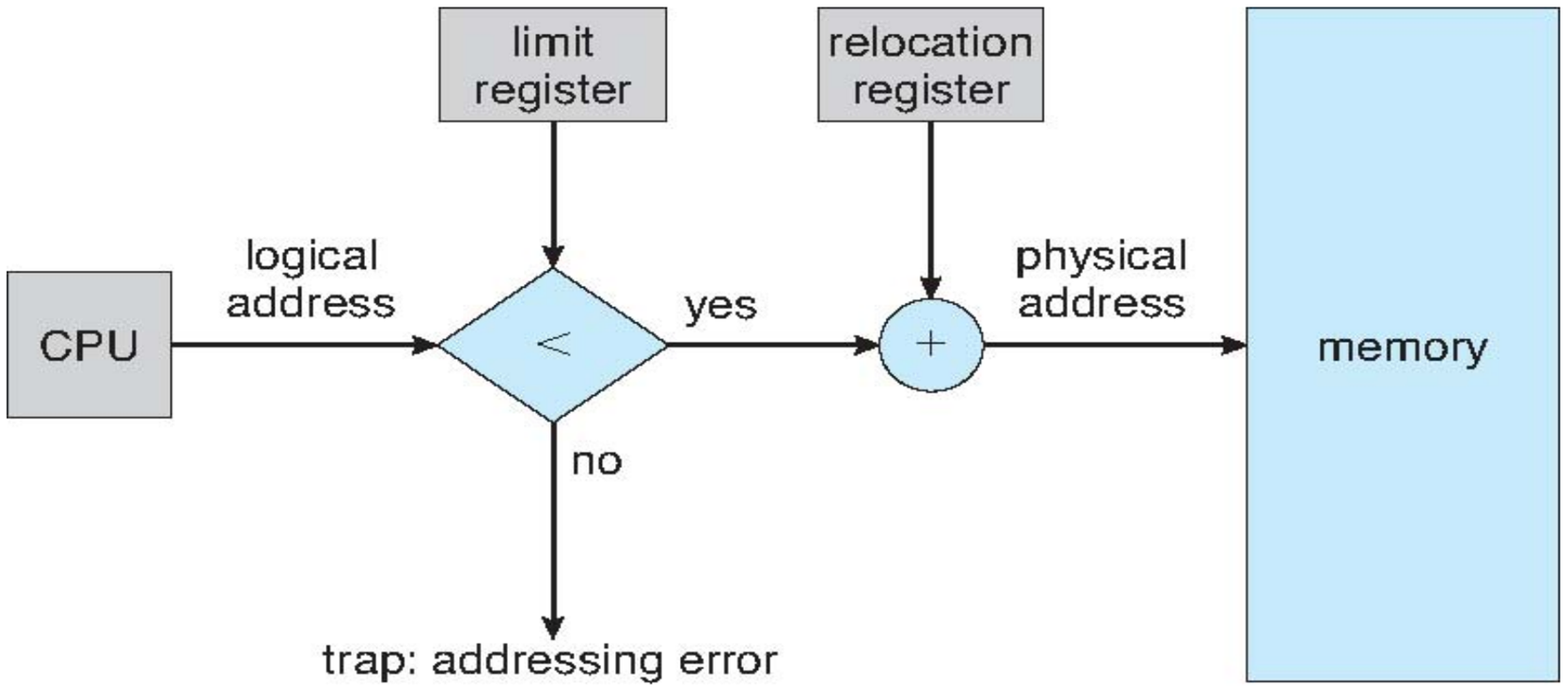
Memory-Management Unit (MMU)

- **Hardware device that, at run time, maps logical address to physical address (for execution-time binding)**
- **Consider simple scheme where the value in a 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 by MMU



Hardware Support for Relocation and Limit Registers

- **Relocation registers used to protect user processes from each other, and from changing operating-system code and data**
  - Relocation register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address dynamically



Dynamic Loading

- **To make better utilization of available memory dynamic loading can be used**
  - A software routine is not loaded into memory until it is called
  - Unused routines are never loaded
- **All routines kept on disk in relocatable load format**
- **Useful when large amounts of code are needed to handle infrequently occurring cases (e.g. error routines)**
- **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

- **To reduce the size of programs, and thus the disk space required for storage and the memory required for execution, use dynamic linking**
  - Static linking – system libraries and program code combined by the loader into the binary program image (i.e. large executable files)
  - Dynamic linking – linking postponed until execution time; smaller executable files that make use of dynamically linked libraries (e.g. .dll files)

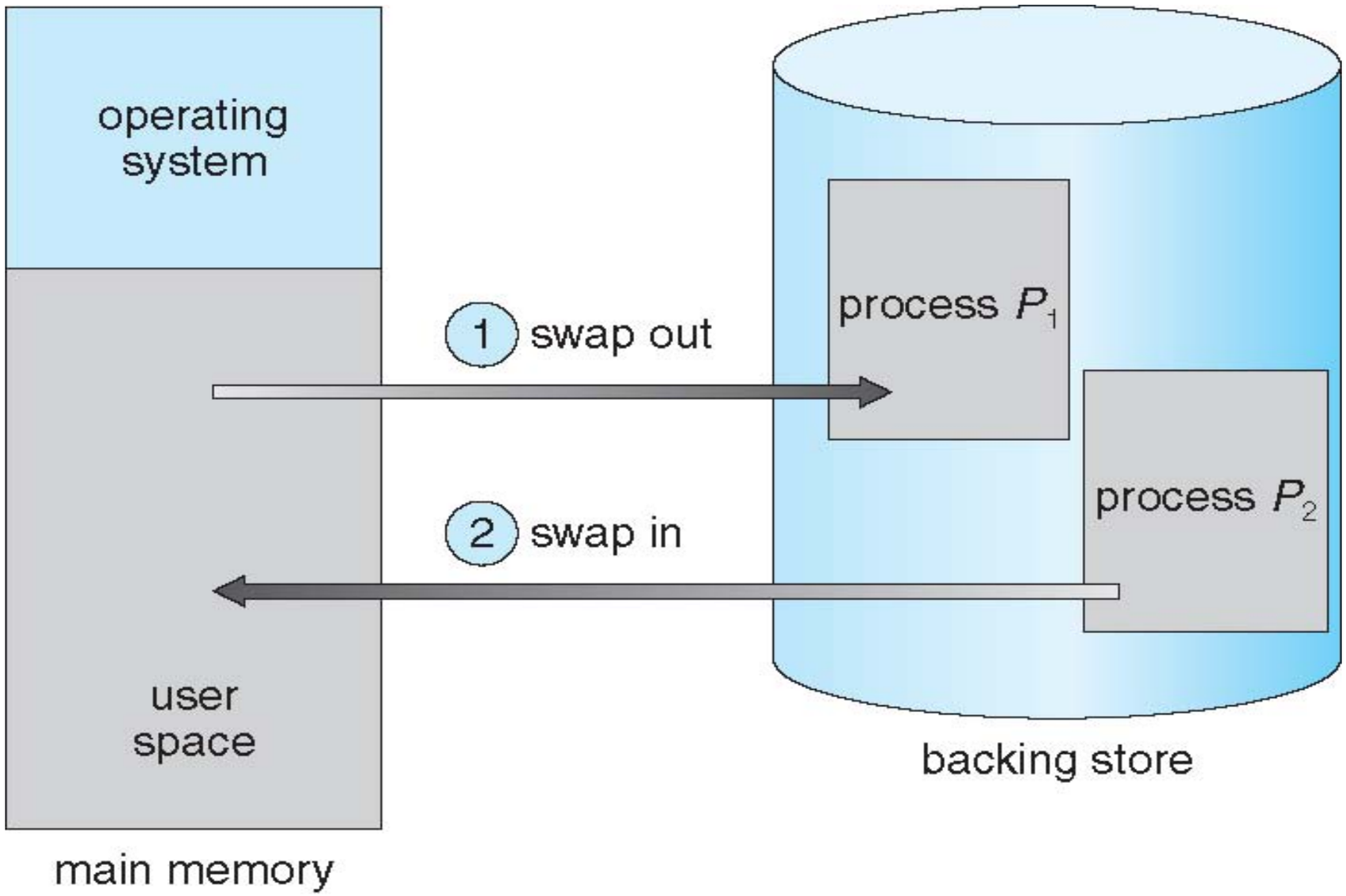
- **A stub is included in the binary image of a program for each reference to a library routine**
  - Used to locate the appropriate library routine and load it into memory (if it hasn't already been loaded)
  - Stub replaces itself with the address of the routine, and executes the routine
  - Subsequent calls to the linked library do not incur a loading penalty

- **Dynamic linking is particularly useful for libraries that are shared amongst many processes**

Swapping

- **A process can be swapped temporarily out of memory to a backing store (a disk), and then brought back into memory to continue execution**
  - Total physical memory space of processes can exceed physical memory
- **Major part of swap time is transfer time to and from backing store; total transfer time is directly proportional to the amount of memory swapped**
- **System maintains a queue of ready-to-run processes which have memory images on disk**
- **If using execution-time binding, the process need not be swapped back into the same physical address from whence it came**
- **Versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)**
  - Swapping is normally disabled
  - Started if memory allocation exceeds some threshold
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



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 be very high when swapping since process needs to be copied from disk storage into system memory**

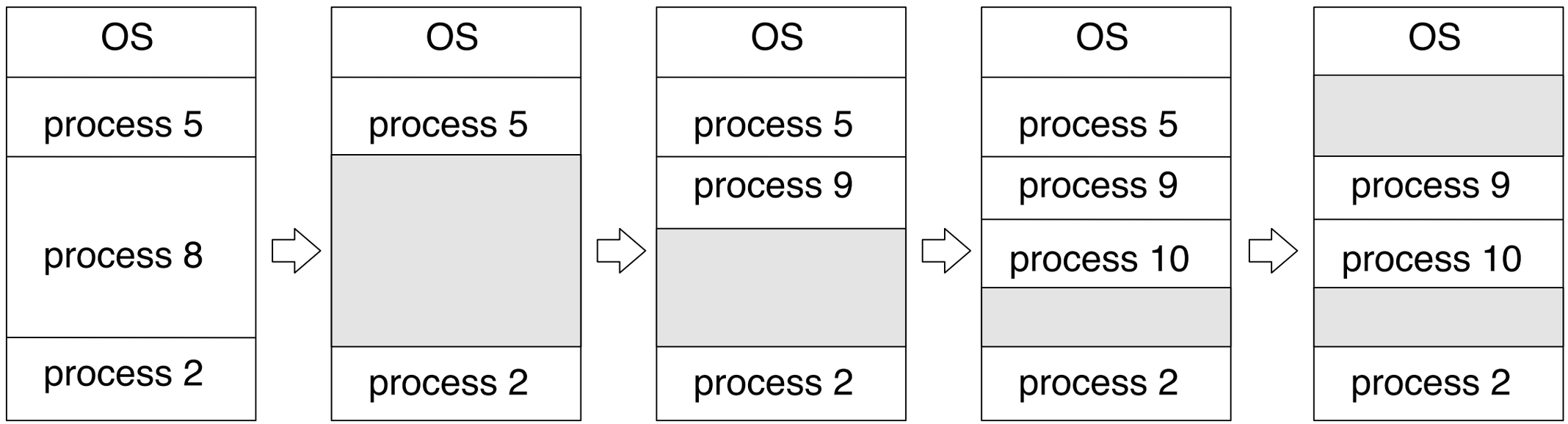
- **500 MB process swapping to hard disk with transfer rate of 500 MB/sec**
  - Plus disk latency of 8 ms
  - Swap out time of 1008 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 2016ms (> 2 seconds)

Memory Allocation: Contiguous Allocation

- **Main memory usually divided into two partitions:**
  - Resident operating system, usually held in low memory
  - User processes held in high memory
- **Need to allocate memory efficiently, a common approach is contiguous allocation**
  - The logical address space for each process is contained in single contiguous section of physical memory
- **It is desirable to allow these contiguous memory segments to move around in physical memory**
  - Requires a single base/relocation register and a single limit register per process

Memory Allocation: Partitioning

- **When allocating memory, the physical memory is divided into multiple, variable-sized, partitions**
  - Degree of multiprogramming is limited by number of partitions (how many process in memory)
  - A hole is a 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
  - When a process exits, it frees its partition, adjacent free partitions are combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (holes)



Dynamic Storage-Allocation Problem

- **How should a memory request of size *n* be satisfied from a list of free holes?**
- **Several approaches:**
  - **First-fit**: Scan list and allocate the first hole that is big enough
  - **Best-fit**: Scan list and allocate the smallest hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
  - **Worst-fit**: Scan list and 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**
- **First-fit is generally the fastest**

Fragmentation

- **Repeatedly allocating and deallocating memory can lead to fragmentation of available memory (fragmentation of holes)**
  - External Fragmentation – memory space exists to satisfy a request, but it is not contiguous; small holes are scattered throughout the physical address space
- **Analysis of first fit allocation reveals that given *N* blocks allocated, another 0.5 *N* blocks will be lost to fragmentation**
  - 1/3 of memory may be unusable -> 50-percent rule
- **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

Fragmentation (Cont.)

- **In some instances, may allocate a larger memory block than required by process**
  - Extra space in memory block is unused by process
  - Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used