Operating Systems

Memory Management

Lecture 9
Michael O'Boyle

Chapter 8: Memory Management

- Background
- Logical/Virtual Address Space vs Physical Address Space
- Swapping
- Contiguous Memory Allocation
- Segmentation

Goals and Tools of memory management

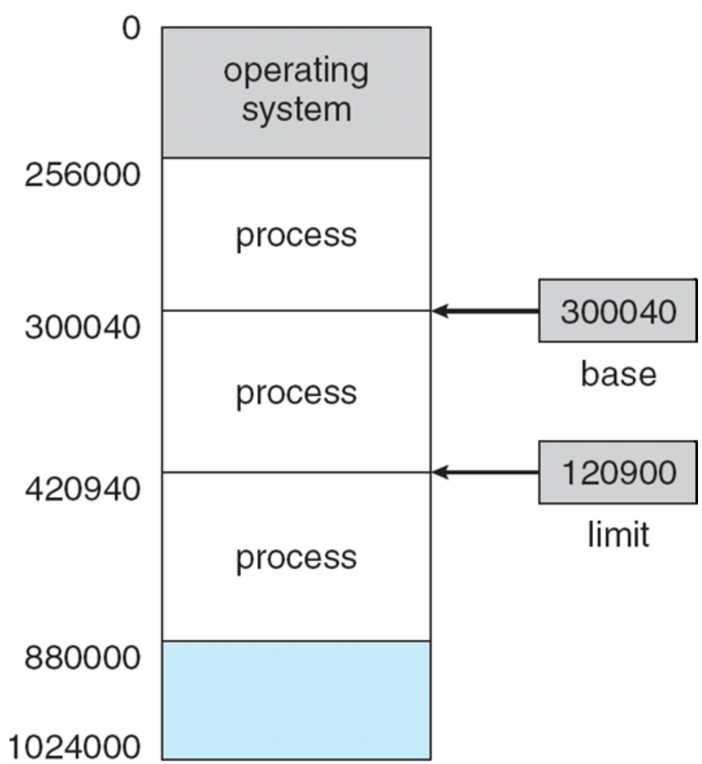
- Allocate memory resources among competing processes,
  - maximizing memory utilization and system throughput
- Provide isolation between processes
  - Addressability and protection: orthogonal
- Convenient abstraction for programming
  - and compilers, etc.
- Tools
  - Base and limit registers
  - Swapping
  - Segmentation
  - Paging, page tables and TLB (Next time)
  - Virtual memory: (Next next time)

Background

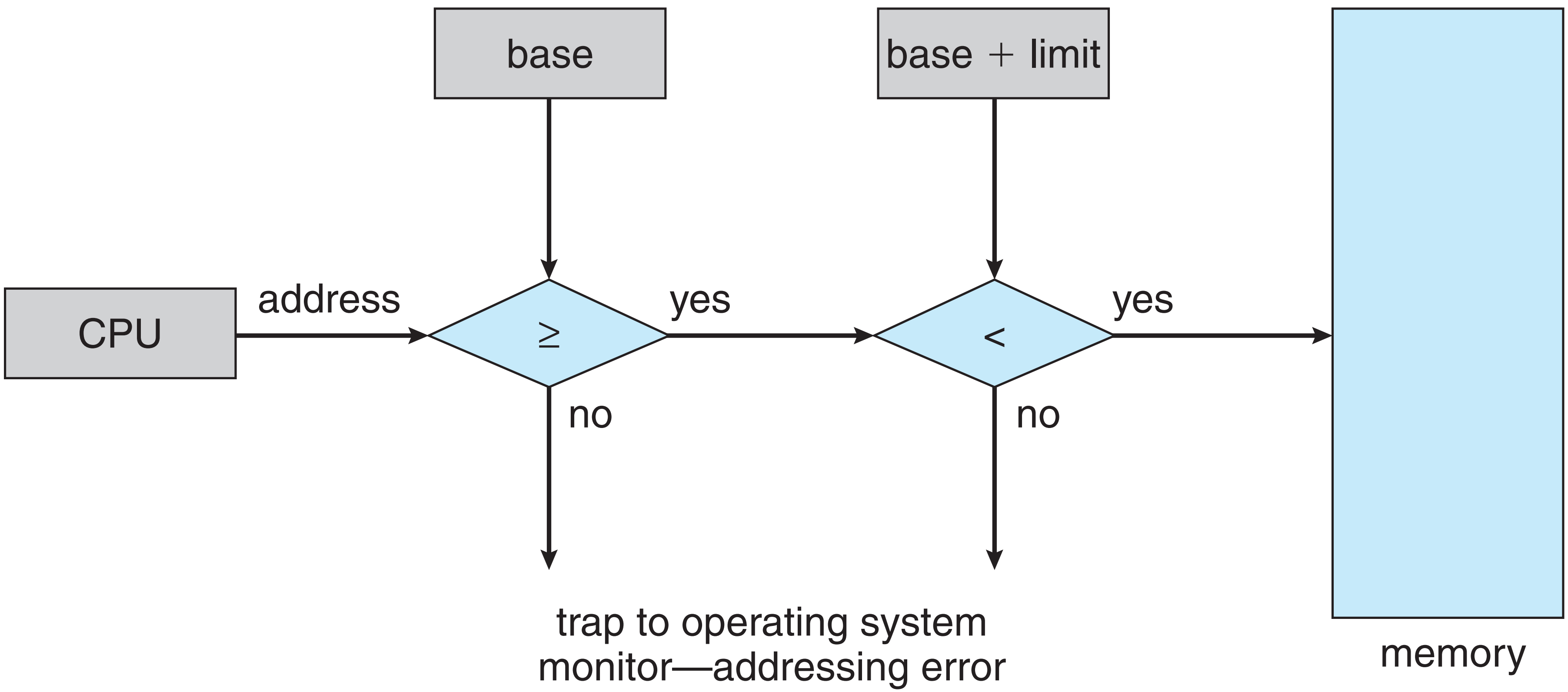
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or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

Base and Limit Registers

- 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



Hardware Address Protection



Virtual addresses for multiprogramming

- To make it easier to manage memory of multiple processes, make processes use logical or virtual addresses
  - Logical/virtual addresses are independent of location in physical memory data lives
    - OS determines location in physical memory
  - instructions issued by CPU reference logical/virtual addresses
    - e.g., pointers, arguments to load/store instructions, PC …
  - Logical/virtual addresses are translated by hardware into physical addresses (with some setup from OS)

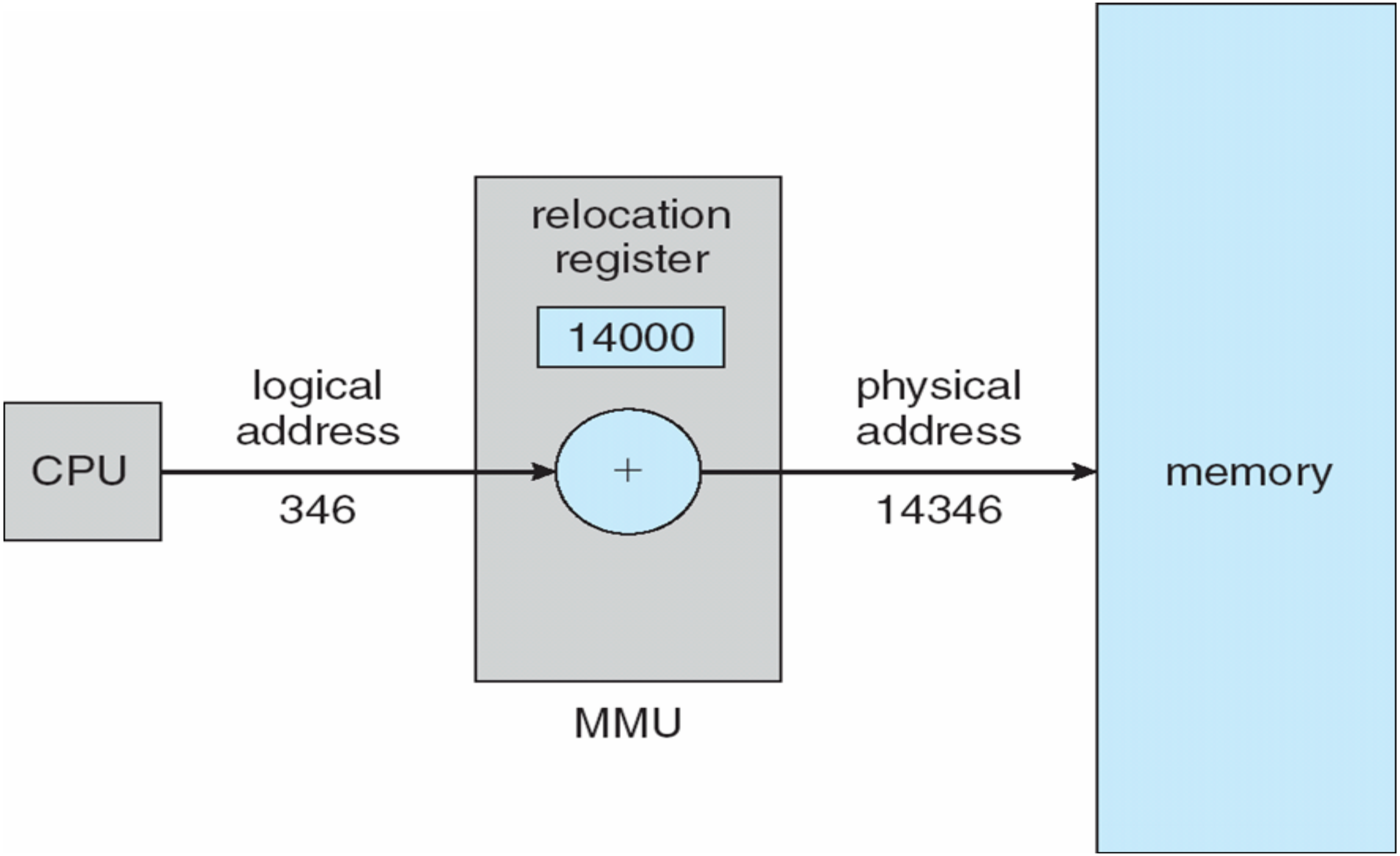
Logical/Virtual Address Space

- The set of logical/virtual addresses a process can reference is its address space
  - many different possible mechanisms for translating logical/virtual addresses to physical addresses
- Program issues addresses in a logical/virtual address space
  - must be translated to physical address space
  - Think of the program as having a contiguous logical/virtual address space that starts at 0,
  - and a contiguous physical address space that starts somewhere else
- **Logical/virtual 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
  - at run time maps virtual to physical address
- Many methods possible
- 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 as a relocation register



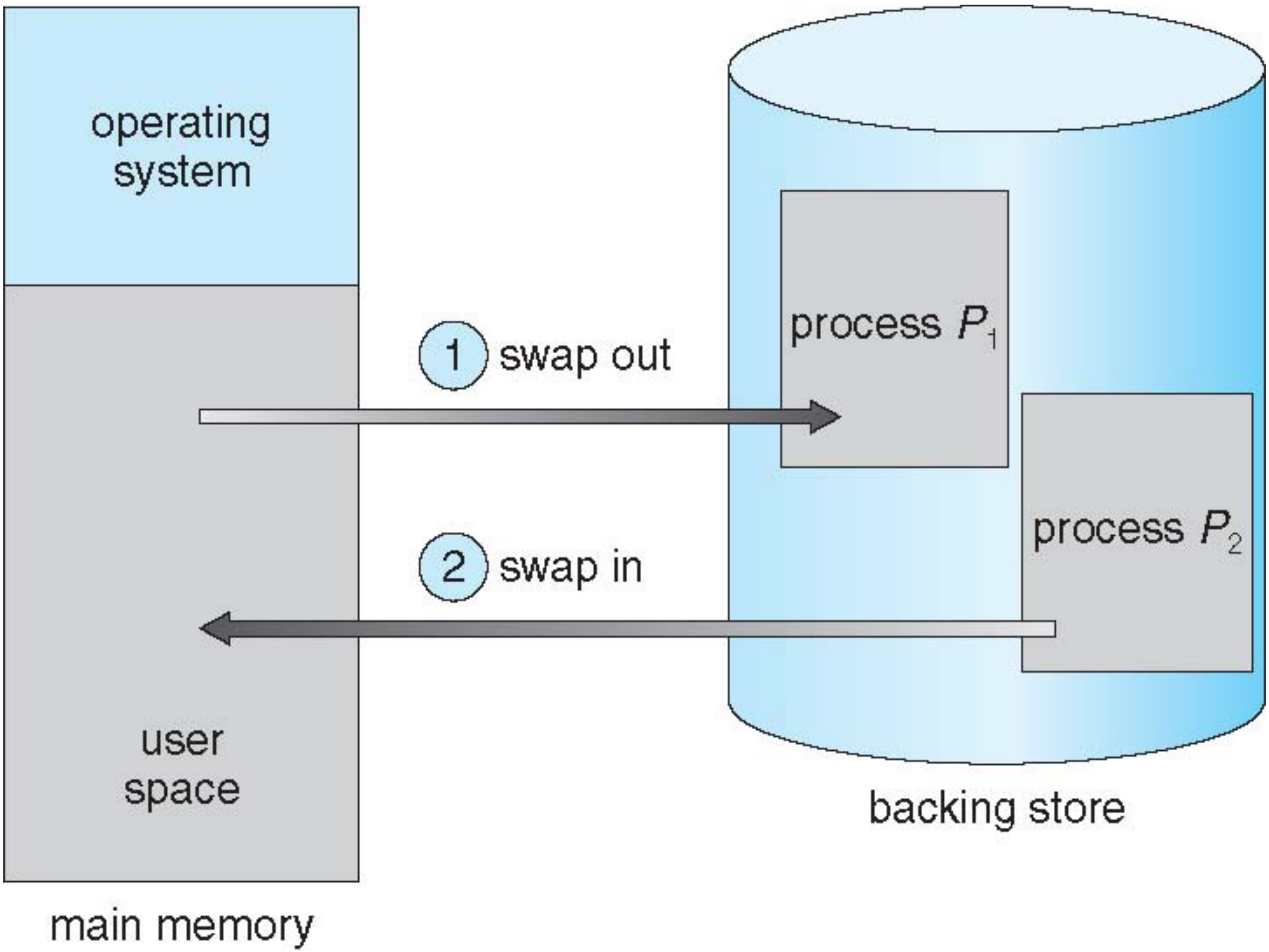
Swapping

- What if not enough memory to hold all processes?
- A process can be **swapped** temporarily
  - out of memory to a backing store,
  - brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **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**
  - ready-to-run processes which have memory images on disk

Swapping

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - MMU prevents the ned for this
  - But 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 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 swapping component time of 4000ms (4 seconds)
- Can reduce cost
  - 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()`

Context Switch Time and Swapping

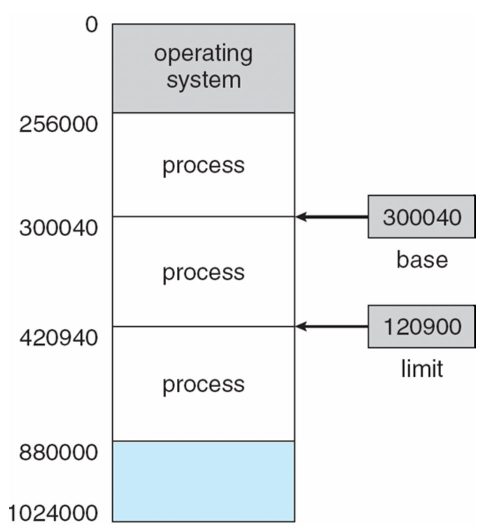
- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
- Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version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

Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging discussed in next lecture

Contiguous Allocation

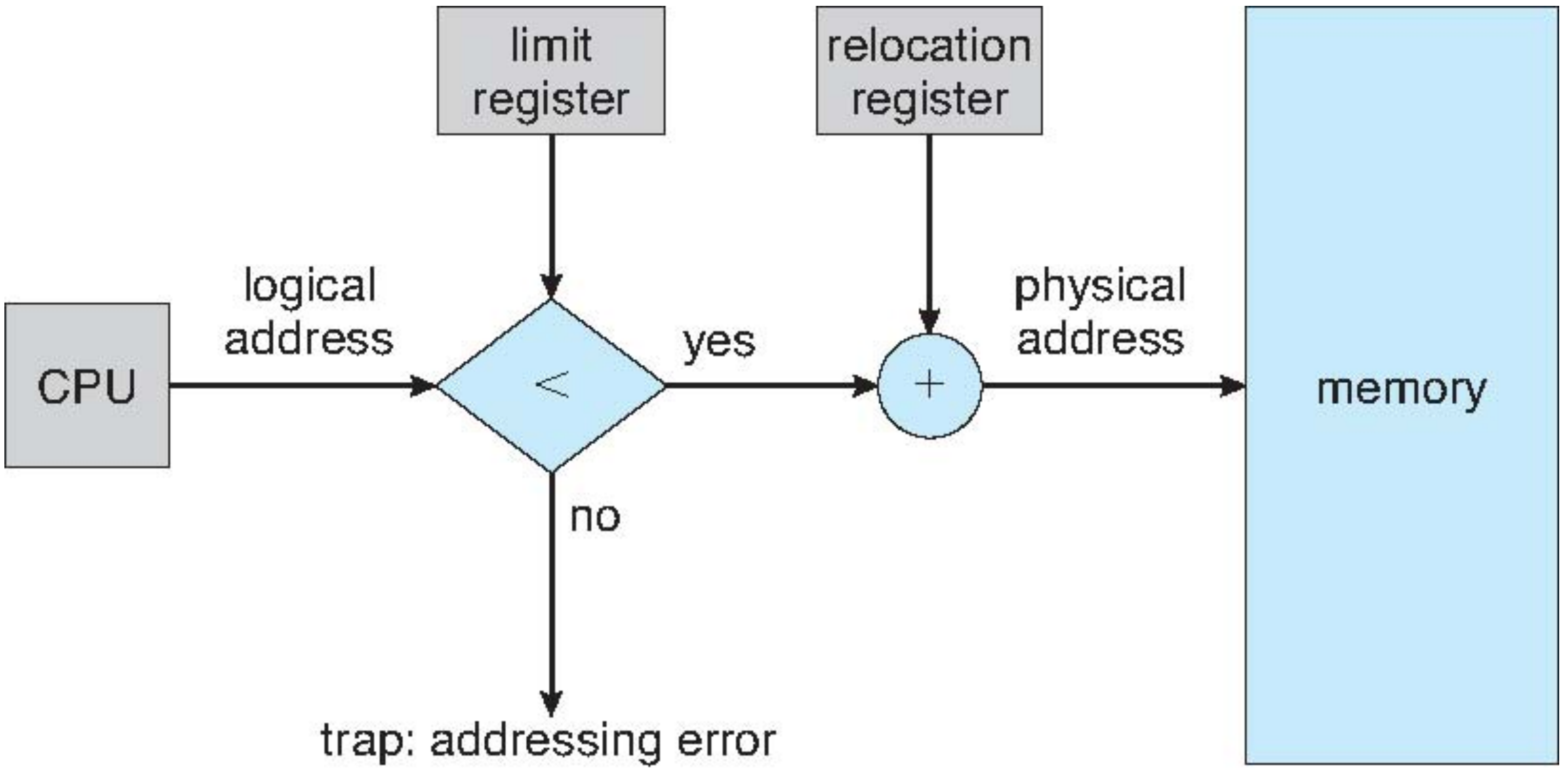
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in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



Contiguous Allocation

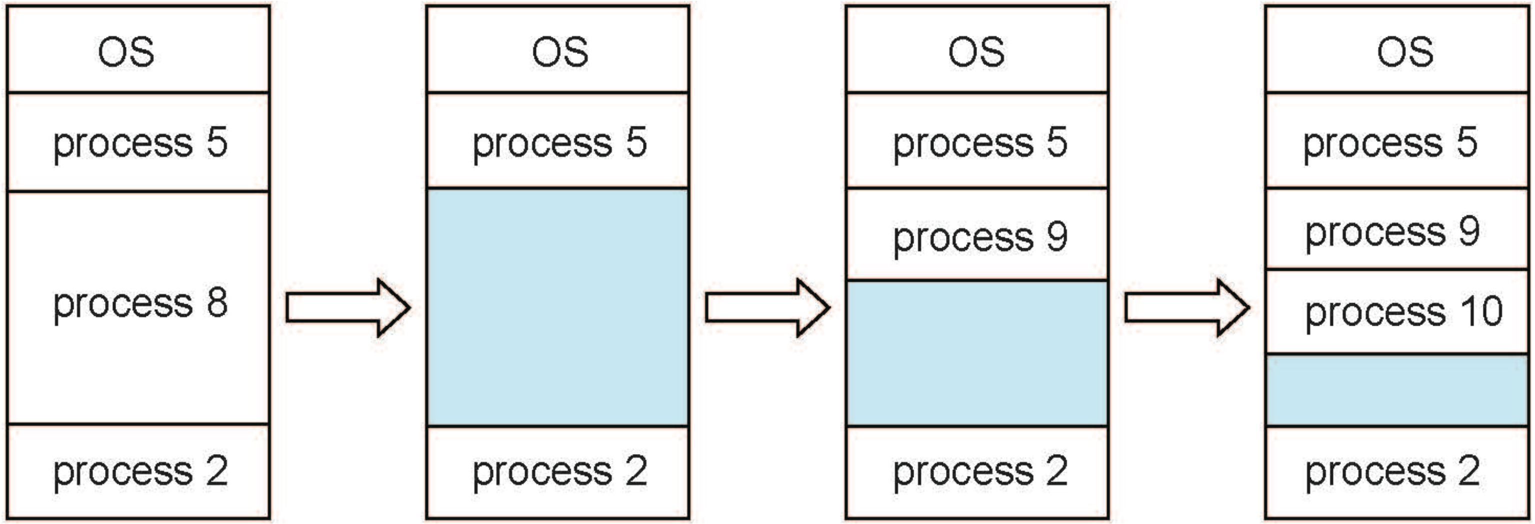
- Relocation registers
  - used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
- MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

Hardware Support for Relocation and Limit Registers



Multiple-partition allocation

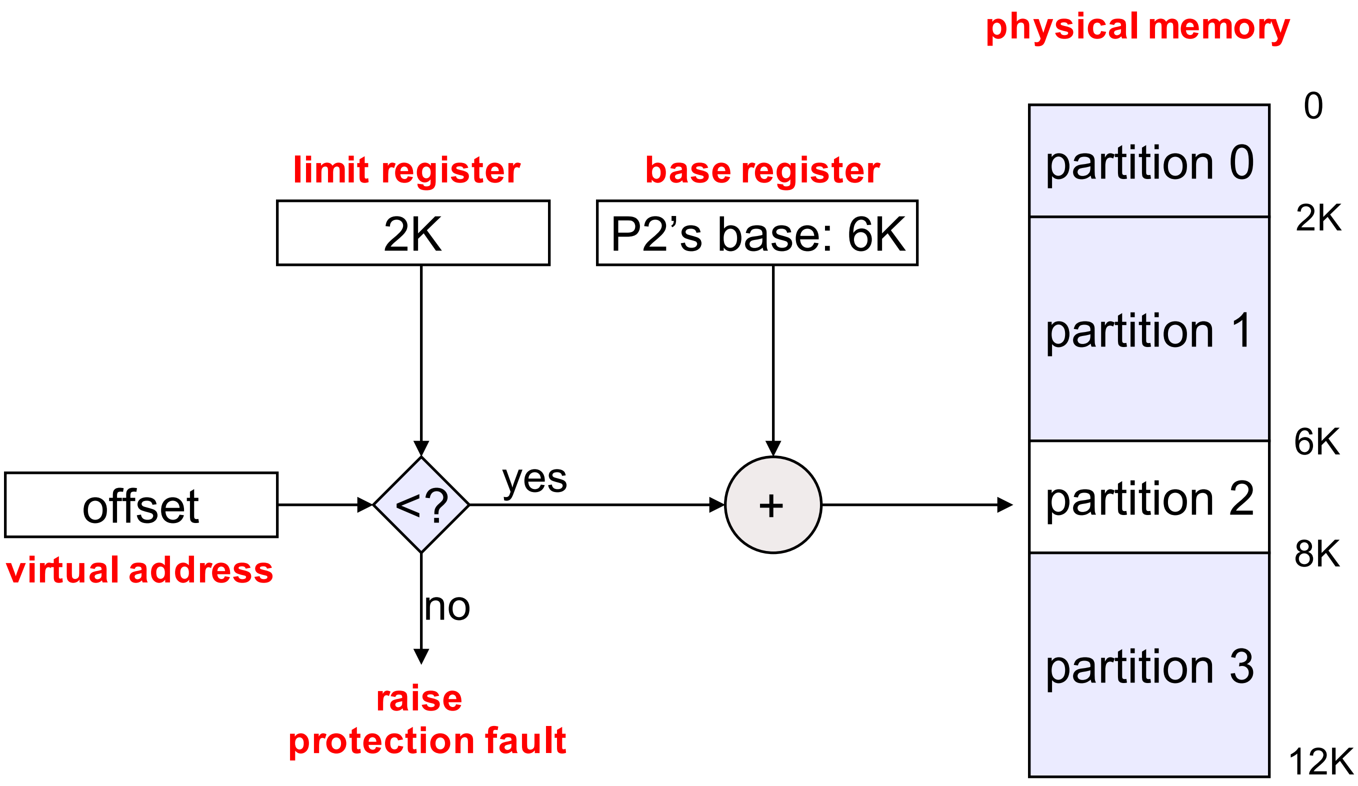
- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - Exam 2 approaches
    - Fixed partition
    - Variable partition



Old technique #1: Fixed partitions

- Physical memory is broken up into fixed partitions
  - partitions may have different sizes, but partitioning never changes
  - hardware requirement: base register, limit register
    - physical address = virtual address + base register
    - base register loaded by OS when it switches to a process
- Advantages
  - Simple
- Problems
  - internal fragmentation: the available partition is larger than what was requested

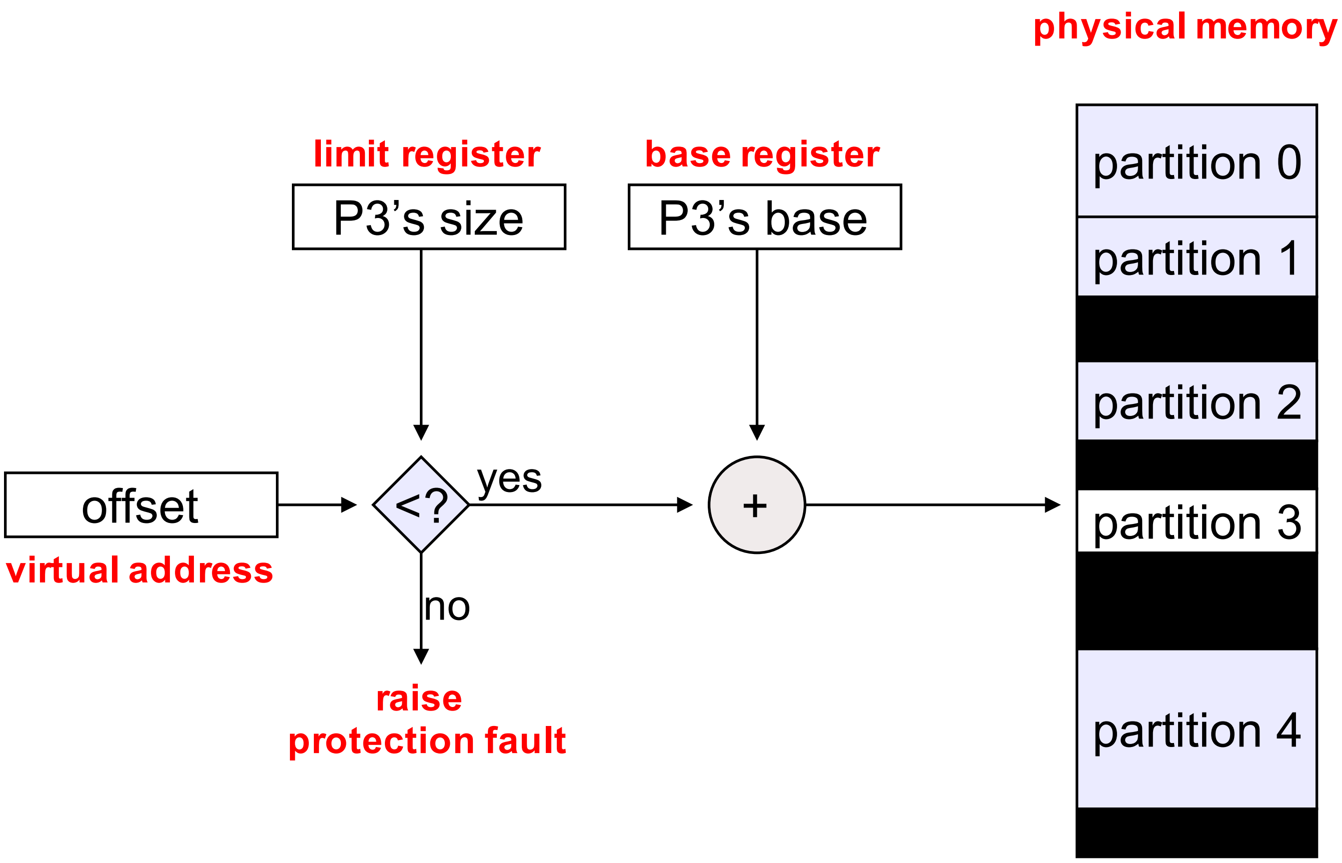
Mechanics of fixed partitions



Old technique #2: Variable partitions

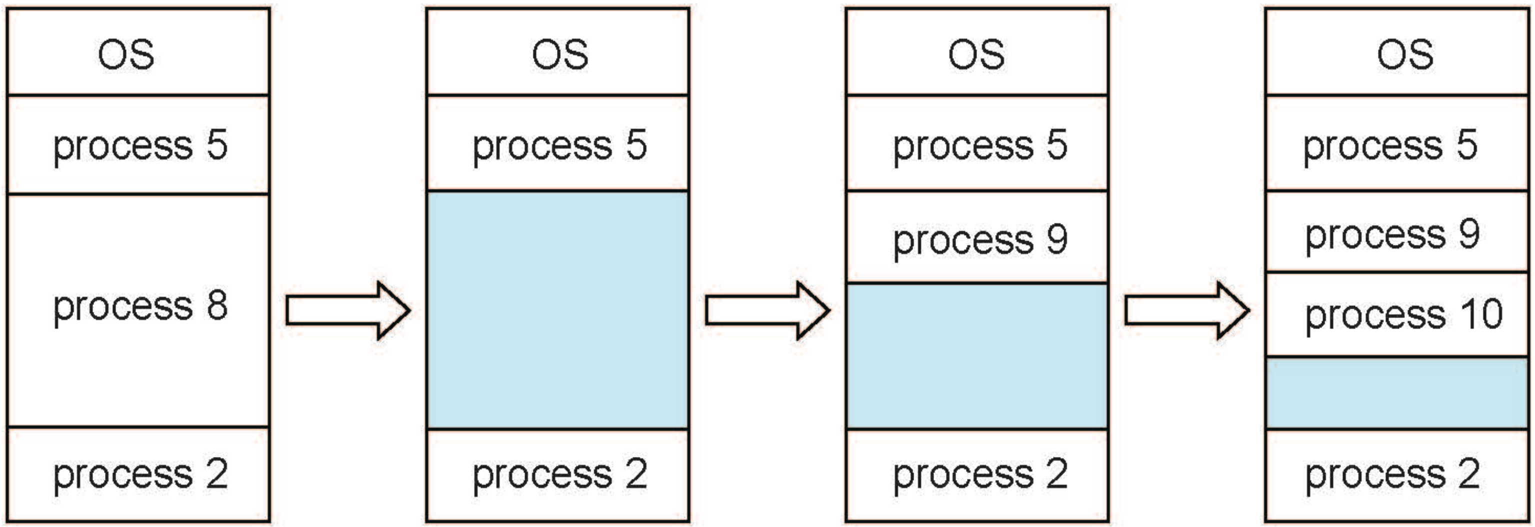
- Obvious next step: physical memory is broken up into partitions dynamically – partitions are tailored to programs
  - hardware requirements: base register, limit register
  - physical address = virtual address + base register
- Advantages
  - no internal fragmentation
    - simply allocate partition 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

Mechanics 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, 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; 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

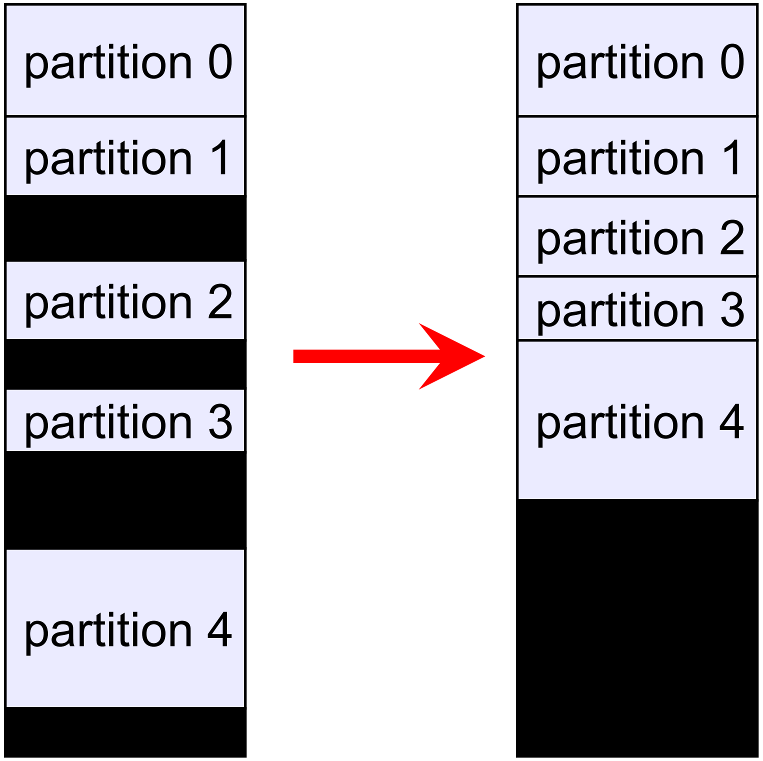
First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory;
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Dealing with fragmentation

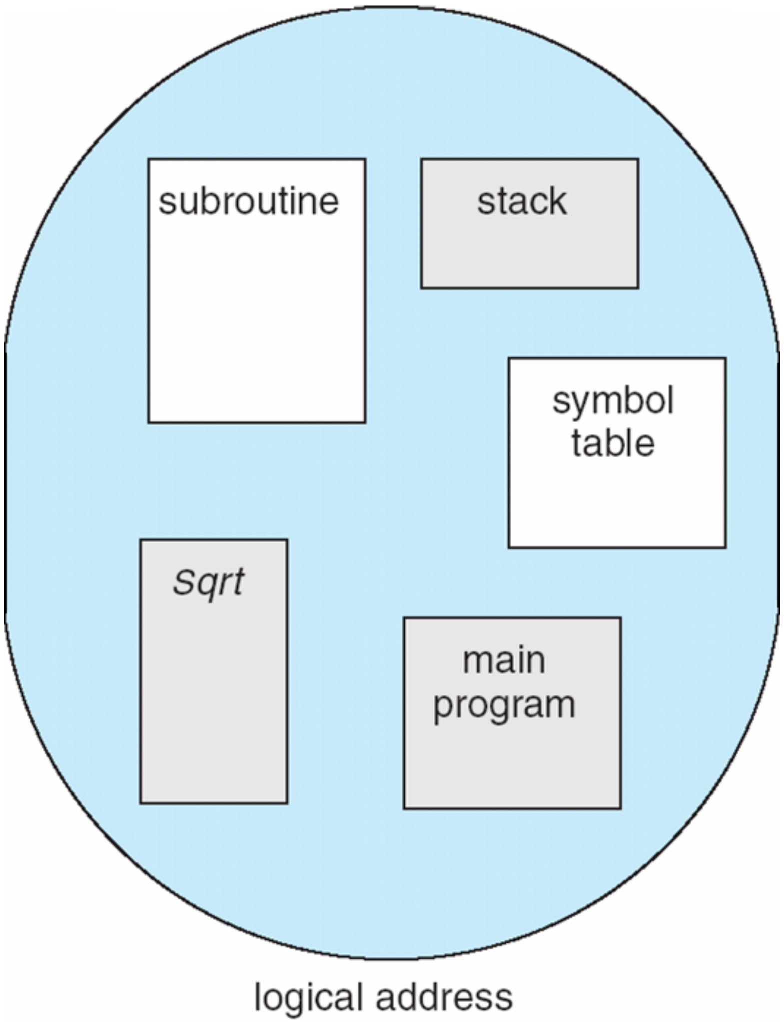
- Compact memory by copying
  - Swap a program out
  - Re-load it, adjacent to another
  - Adjust its base register
  - Compaction is possible *only* if relocation is dynamic
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers



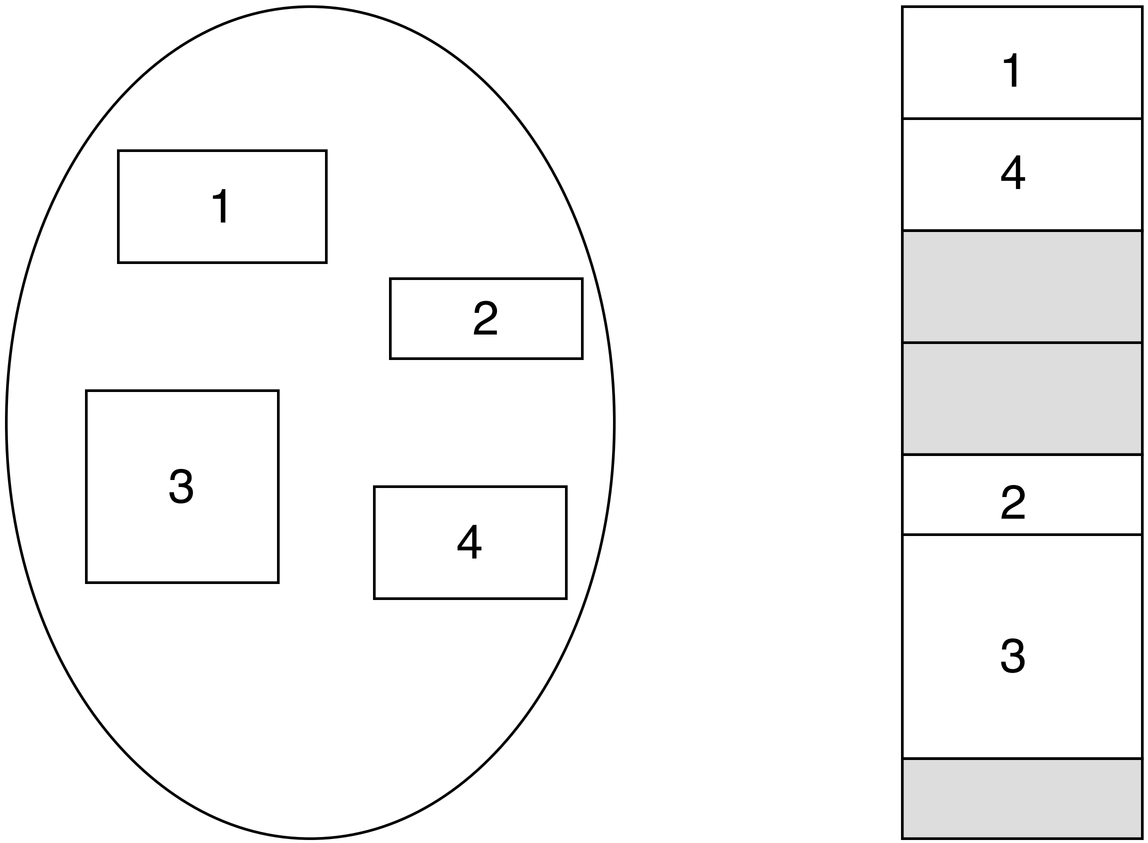
Segmentation

- Dealing with fragmentation
  - Why not remove need for continuous adresses?
- Segmentation
  - partition an address space into *logical* units
    - stack, code, heap, subroutines, …
  - a virtual address is <segment #, offset>
- Facilitates sharing and reuse
  - a segment is a natural unit of sharing – a subroutine or function
- A natural extension of variable-sized partitions
  - variable-sized partition = 1 segment/process
  - segmentation = many segments/process

User’s View of a Program



Logical View of Segmentation



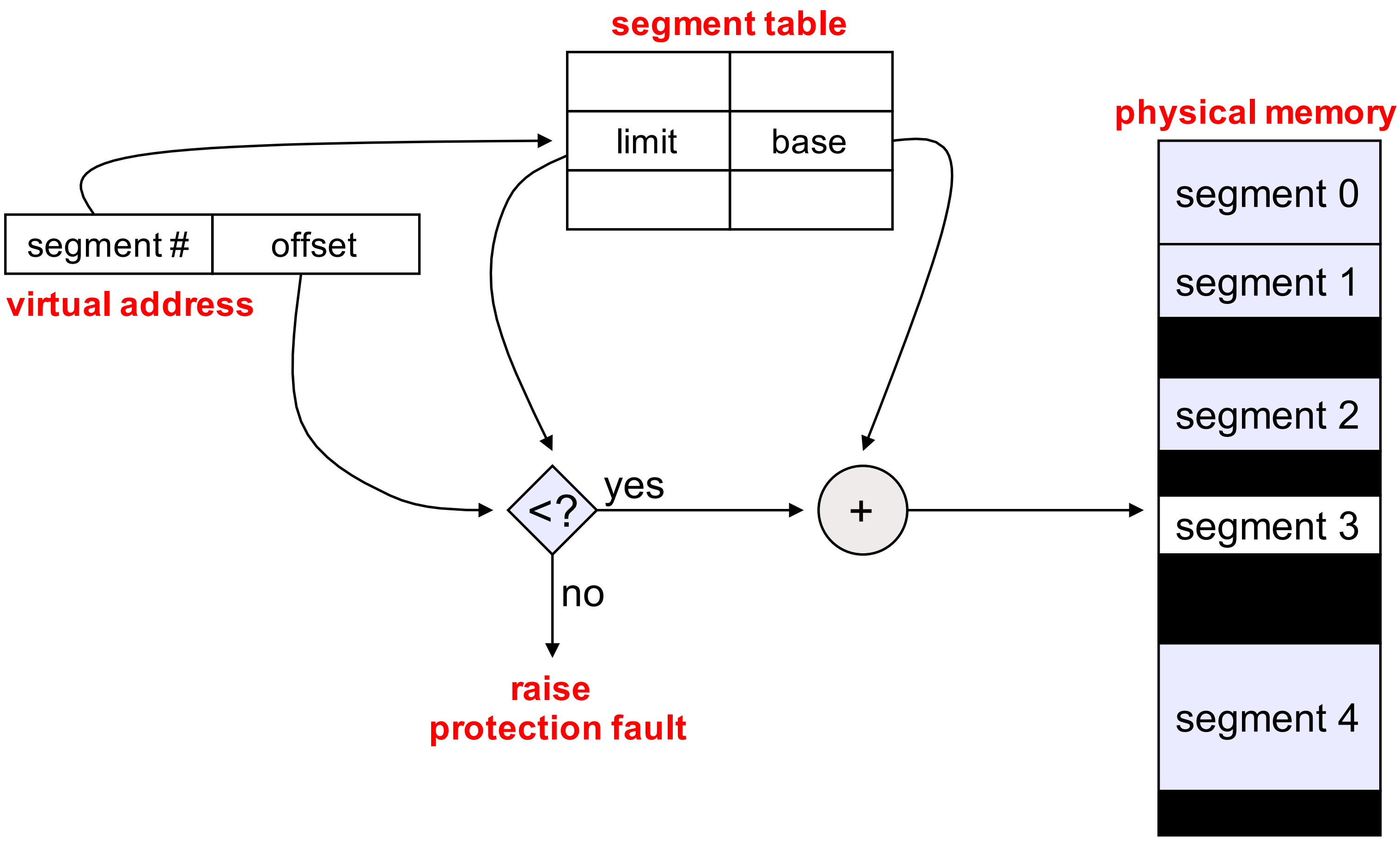
user space

physical memory space

Hardware support

- Segment table
  - multiple base/limit pairs, one per segment
  - segments named by segment #, used as index into table
    - a virtual address is <segment #, offset>
  - offset of virtual address added to base address of segment to yield physical address

Segment lookups



Pros and cons

- Logical and it facilitates sharing and reuse
- Allows non-contiguous physical addresses
  - Helps exploits varying sized holes
- But it has the complexity of a variable partition system
  - except that linking is simpler, and the "chunks" that must be allocated are smaller than a "typical" linear address space
- Segmentation rarely used alone
  - Paging is the basis for modern memory management
  - Covered in next lecture

Summary

- Logical/Virtual Address Space vs Physical Address Space
- Swapping
- Contiguous Memory Allocation
- Fragmentation
- Segmentation
- Paging
  - A better solution
  - Next lecture