Memory Management

CSCI 3753 Operating Systems

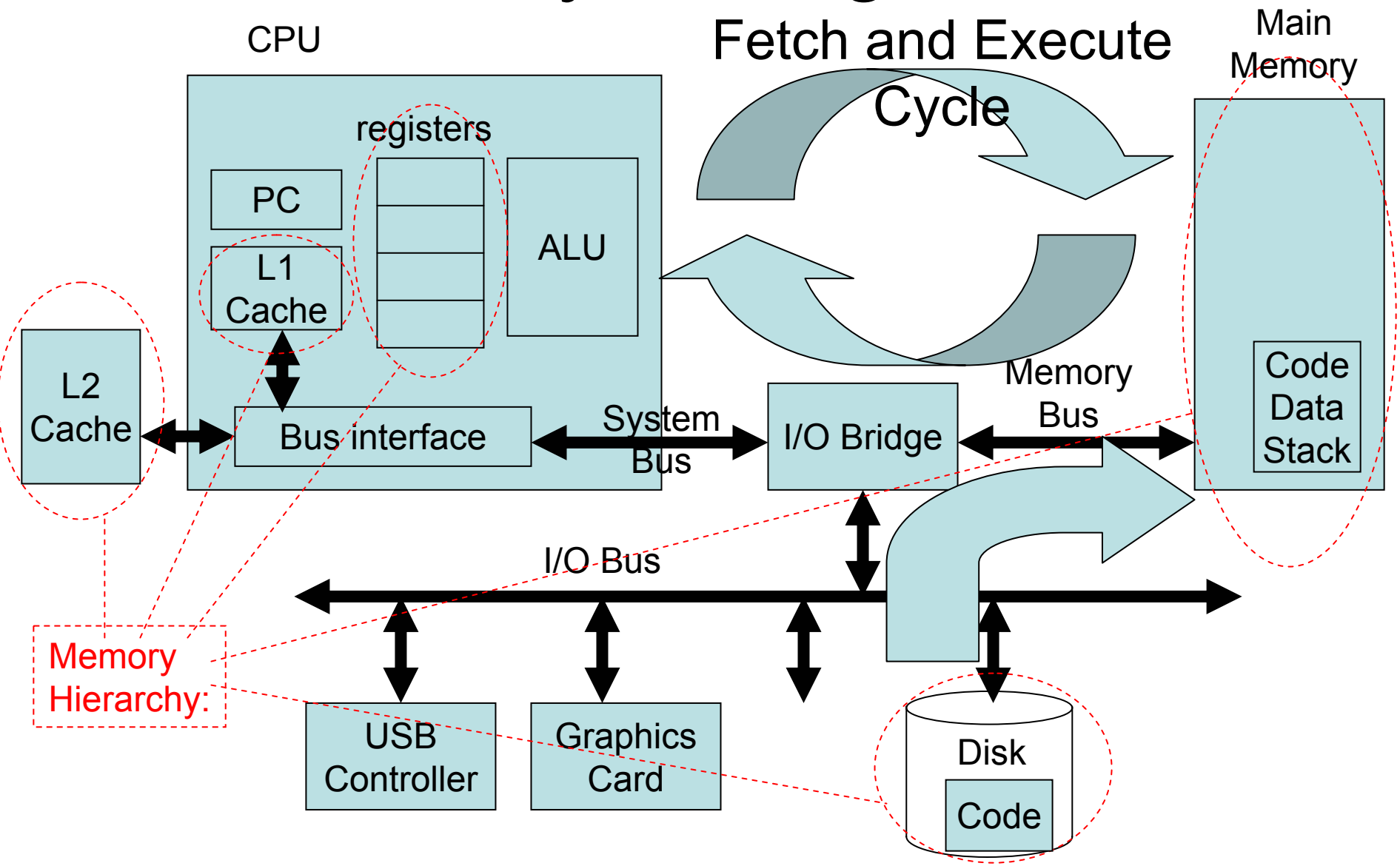
Spring 2005

Prof. Rick Han

Announcements

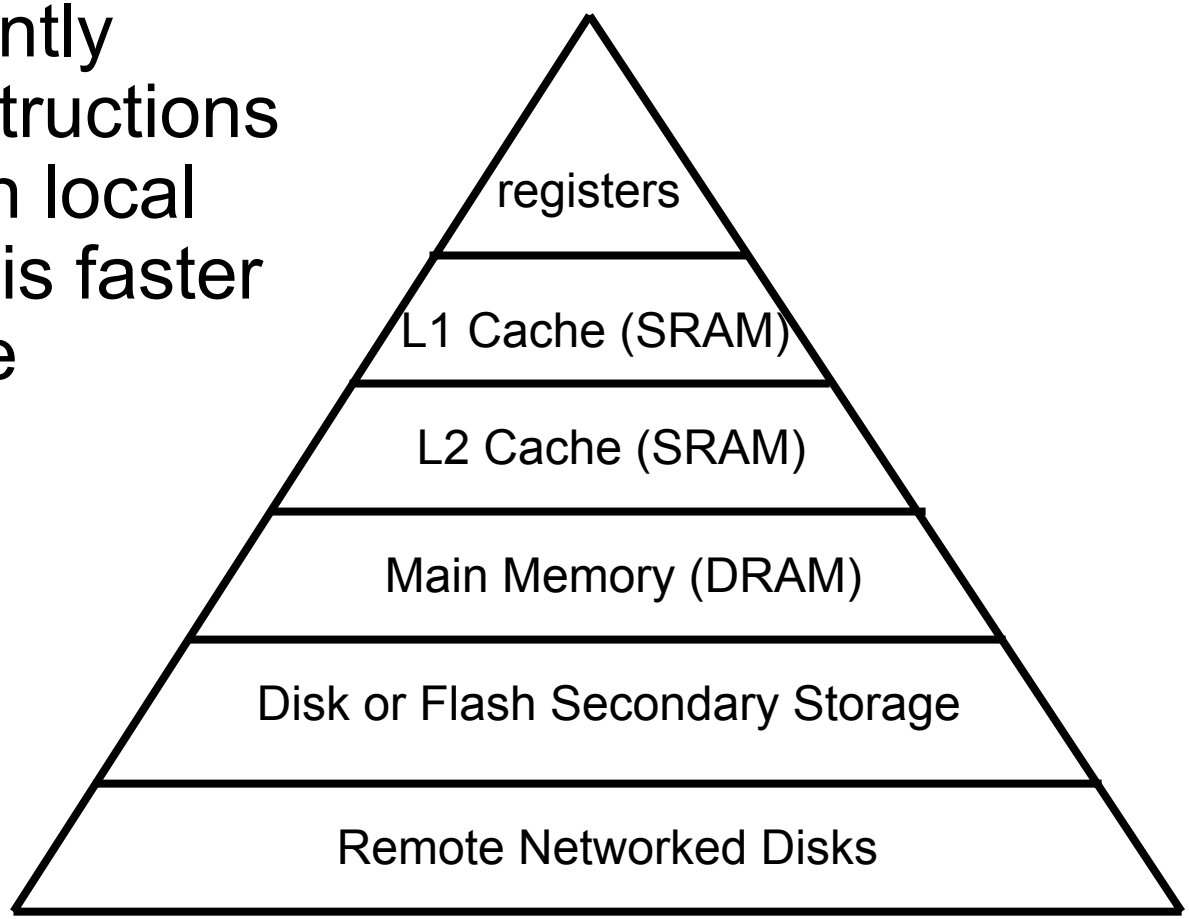
- PA #2 due Friday March 18 11:55 pm - note extension of a day
- Midterms returned after spring break
- Read chapters 11 and 12

Memory Management

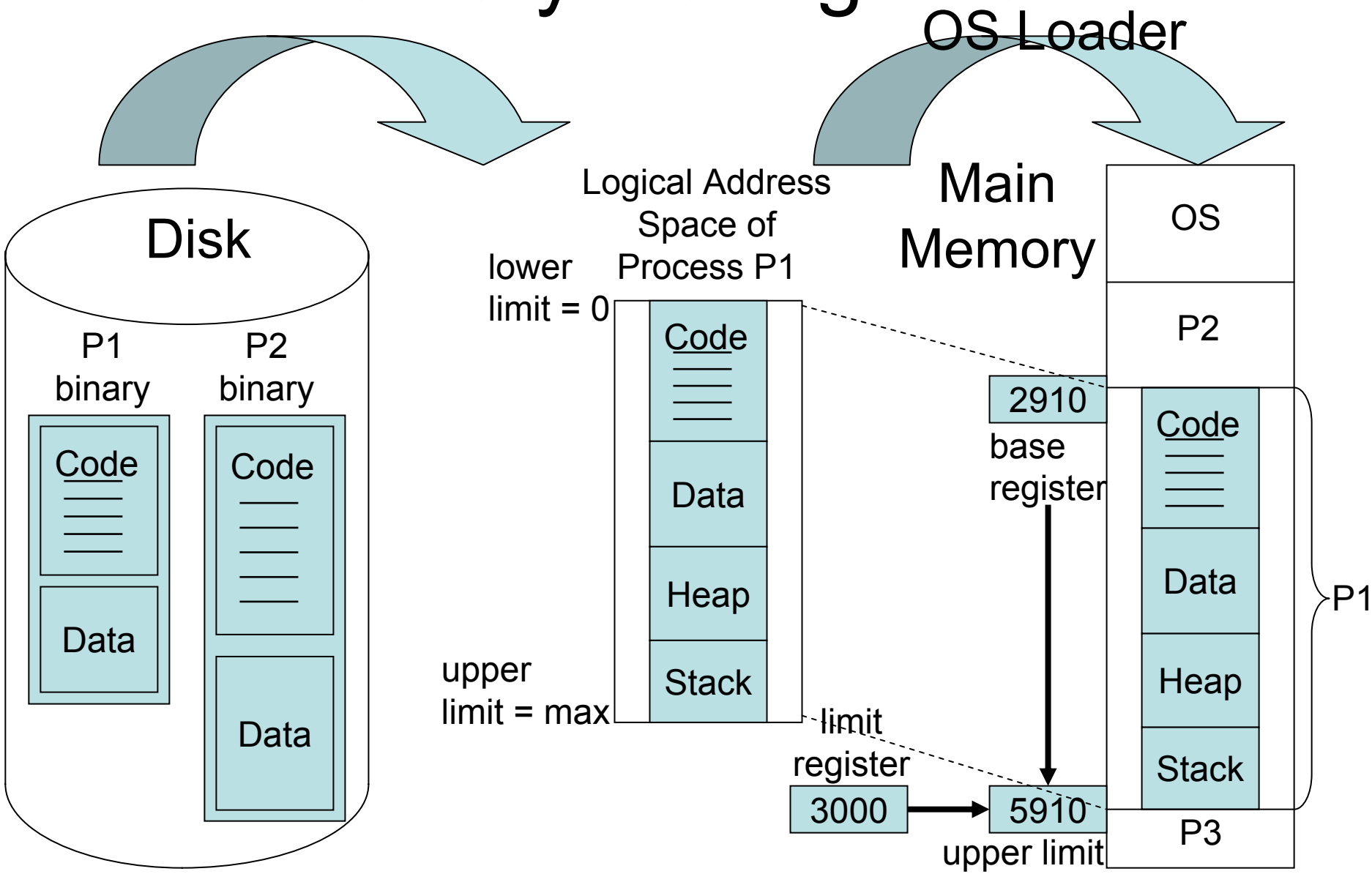


Memory Management

- Memory Hierarchy
  - cache frequently accessed instructions and/or data in local memory that is faster but also more expensive



Memory Management



Memory Management

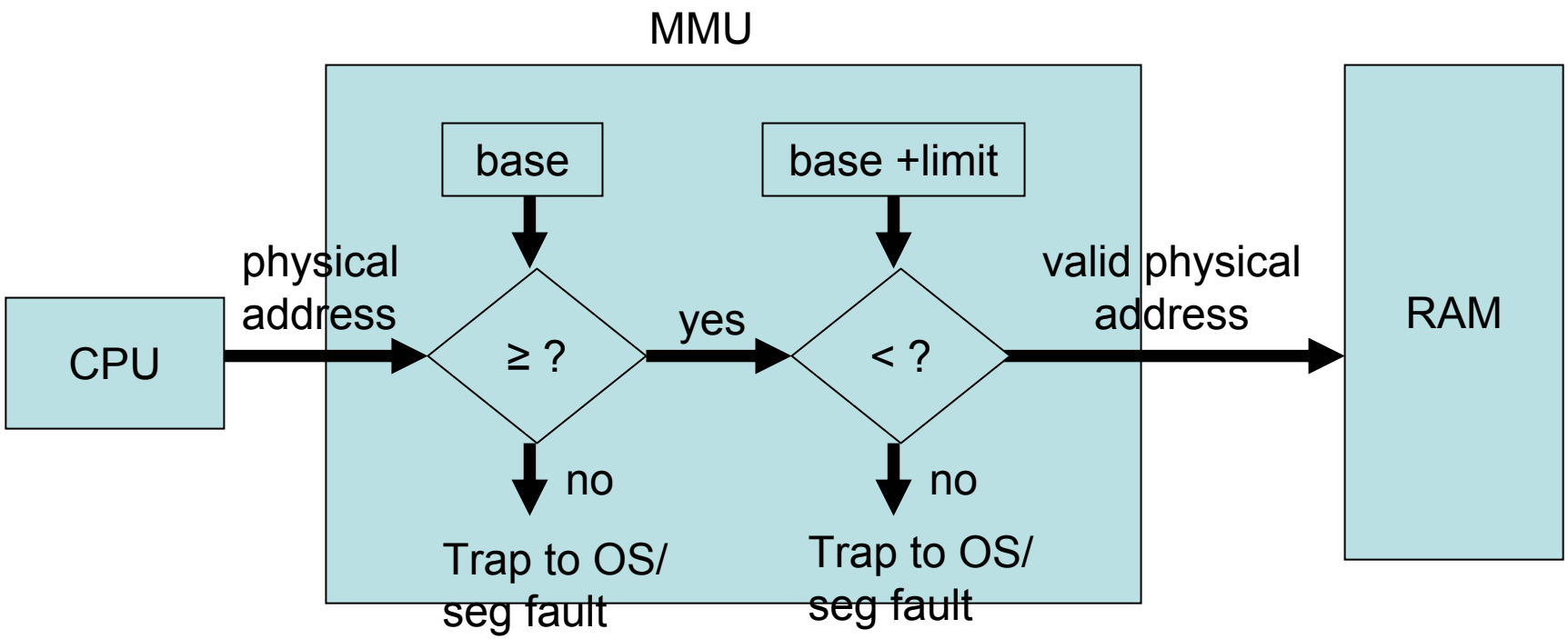
- In the previous figure, when the program P1 becomes an active process P1, then the process P1 can be viewed as conceptually executing in a *logical address space* ranging from *0* to *max*
  - In reality, the code and data are stored in physical memory
  - There needs to be a mapping from logical addresses to physical addresses - *memory management unit* (MMU) takes care of this
- Usually, these logical addresses are mapped into physical addresses when the process is loaded into memory
  - the logical address space becomes mapped into a *physical address space*
  - *base register* keeps track of lower limit of the physical address space
  - *limit register* keeps track of size of logical address space
  - upper limit of physical address space = base register + limit register

Memory Management

- base and limit registers provide hardware support for a simple MMU
  - memory access should not go out of bounds. If out of bounds, then this is a segmentation fault so trap to the OS.
    - MMU will detect out-of-bounds memory access and notify OS by throwing an exception
- Only the OS can load the base and limit registers while in kernel/supervisor mode
  - these registers would be loaded as part of a context switch

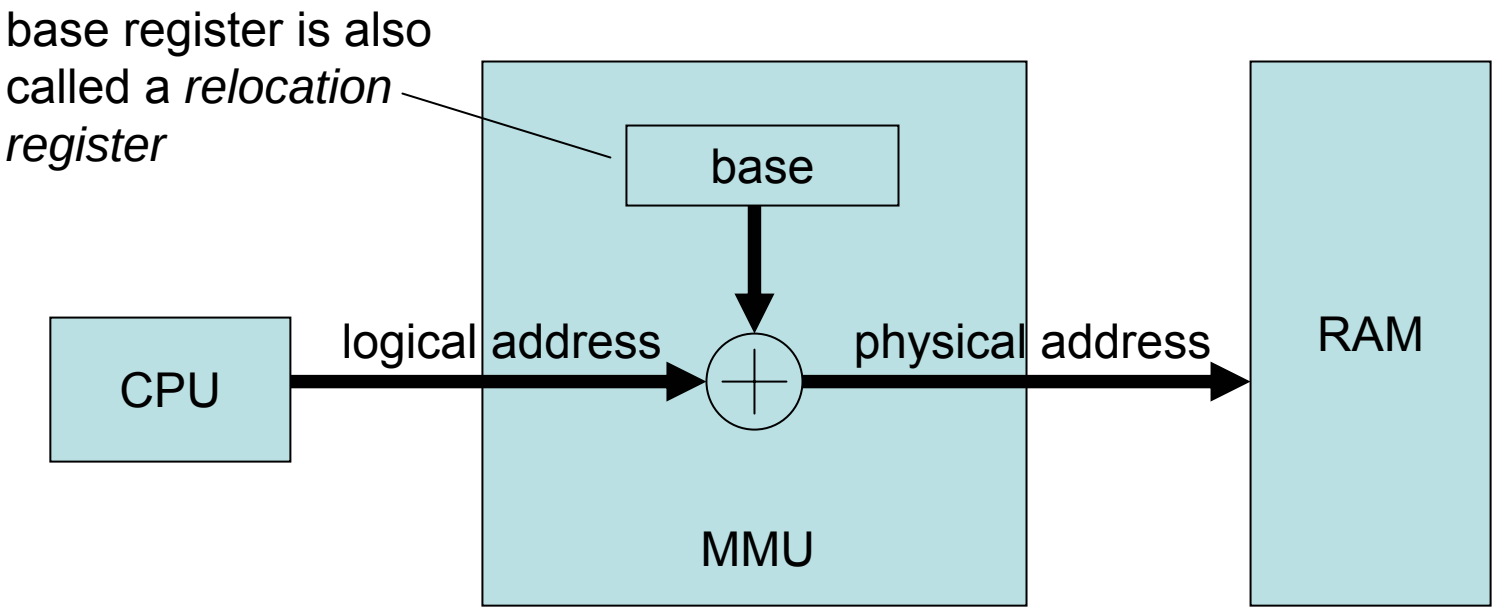
Memory Management

- MMU needs to check if physical memory access is out of bounds



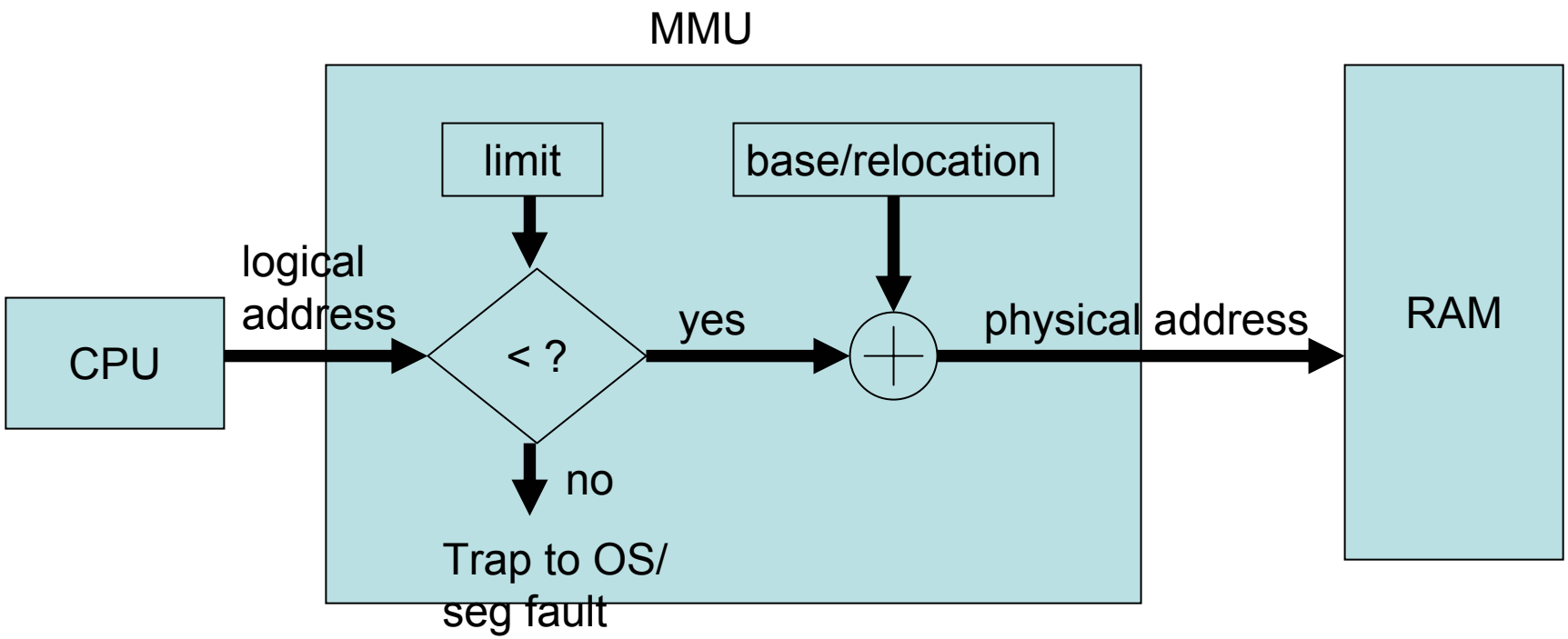
Memory Management

- MMU also needs to perform run-time *mapping* of logical/virtual addresses to physical addresses
  - each logical address is *relocated or translated* by MMU to a physical address that is used to access main memory/RAM
  - thus the application program never sees the actual physical memory - it just presents a logical address to MMU



Memory Management

- Let's combine the MMU's two tasks (bounds checking, and memory mapping) into one figure
  - since logical addresses can't be negative, then lower bound check is unnecessary - just check the upper bound by comparing to the limit register



Memory Management

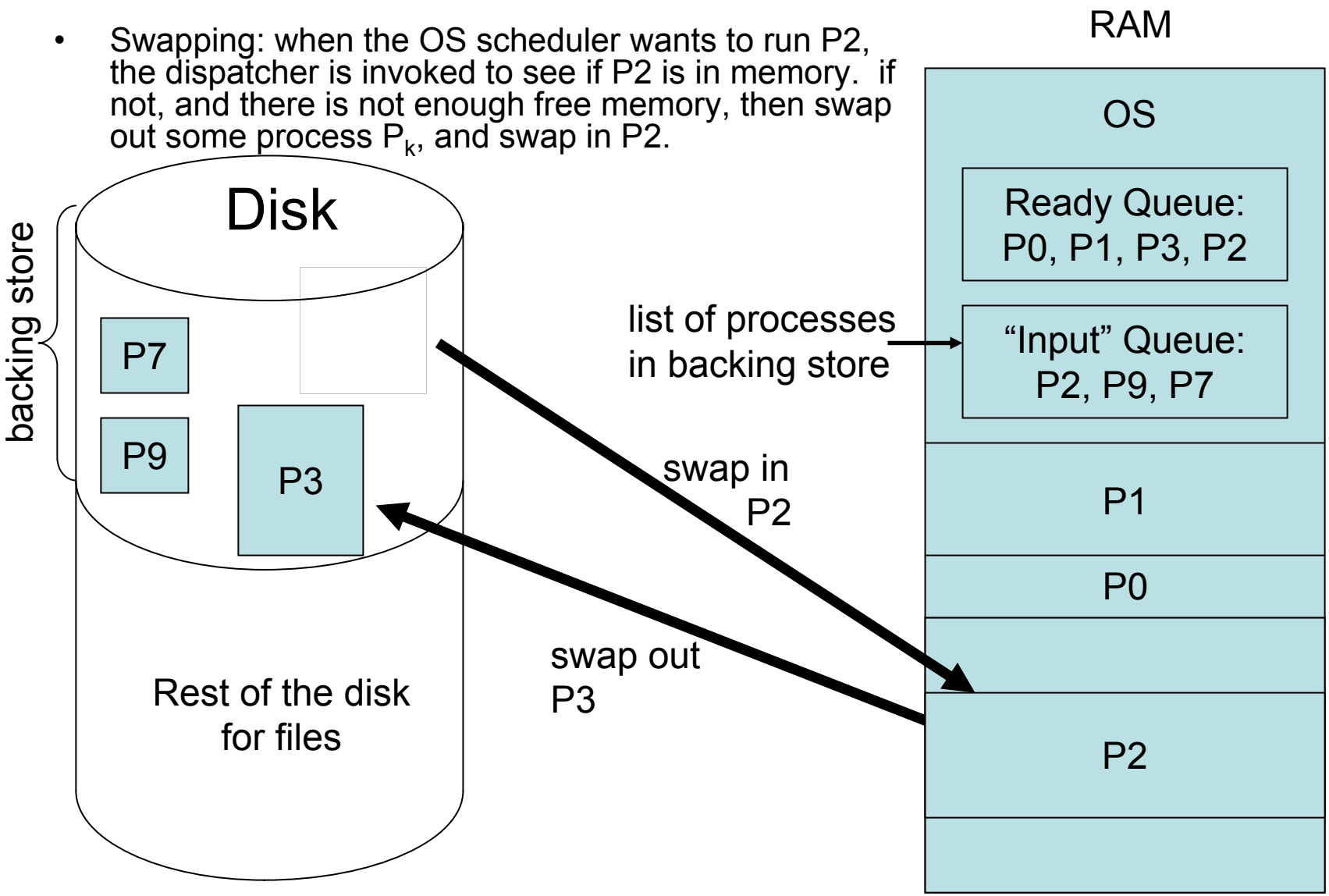
- Address Binding
  - Compile Time:
    - If you know in advance where in physical memory a process will be placed, then compile your code with absolute physical addresses
  - Load Time:
    - Code is first compiled in relocatable format. Then replace logical addresses in code with physical addresses during loading
      - e.g. relative address (14 bytes from beginning of this module) is replaced with a physical address (74002)
  - Run Time/Execution Time: (most modern OS's do this)
    - Code is first compiled in relocatable format as if executing in its own virtual address space. As each instruction is executed, i.e. at run time, the MMU relocates the virtual address to a physical address using hardware support such as base/relocation registers.

Memory Management

- Swapping
  - memory may not be able to store all processes that are ready to run, i.e. that are in the ready queue
  - Use disk/secondary storage to store some processes that are temporarily swapped out of memory, so that other (swapped in) processes may execute in memory
    - special area on disk is allocated for this *backing store*. This is fast to access.
  - If execution time binding is used, then a process can be easily swapped back into a different area of memory.
  - If compile time or load time binding is used, then process swapping will become very complicated and slow - basically undesirable

Memory Management

- Swapping: when the OS scheduler wants to run P2, the dispatcher is invoked to see if P2 is in memory. if not, and there is not enough free memory, then swap out some process P_k, and swap in P2.

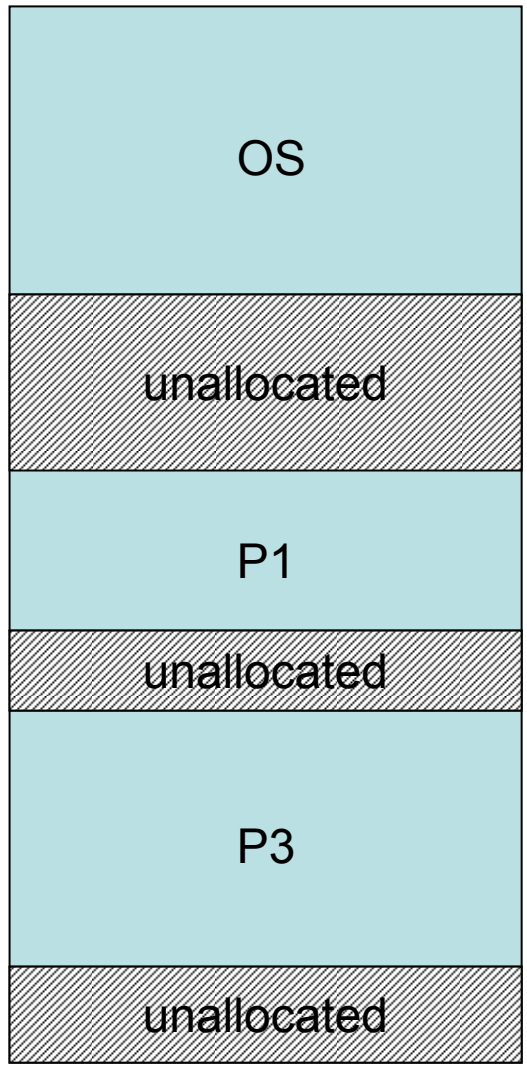


Memory Management

- Some difficulties with swapping:
  - context-switch time of swapping is very slow
    - on the order of tens to hundreds of milliseconds
    - can't always hide this latency if in-memory processes are blocked on I/O
    - UNIX avoids swapping unless the memory usage exceeds a threshold
  - *fragmentation* of main memory becomes a big issue
    - can also get fragmentation of backing store disk
  - swapping of processes that are blocked or waiting on I/O becomes complicated
    - one rule is to simply avoid swapping processes with pending I/O

Memory Management

- Allocation
  - as processes arrive, they're allocated a space in main memory
  - over time, processes leave, and memory is deallocated
  - This results in *external fragmentation* of main memory, with many small chunks of non-contiguous unallocated memory between allocated processes in memory
  - OS must find an unallocated chunk in fragmented memory that a process fits into. Multiple strategies:
    - *first fit* - find the 1st chunk that is big enough
    - *best fit* - find the smallest chunk that is big enough
    - *worst fit* - find the largest chunk that is big enough
      - this leaves the largest contiguous unallocated chunk for the next process



Memory Management

- Fragmentation can mean that, even though there is enough overall free memory to allocate to a process, there is not one contiguous chunk of free memory that is available to fit a process's needs
  - the free memory is scattered in small pieces throughout RAM
- Both first-fit and best-fit aggravate external fragmentation, while worst-fit is somewhat better
- Solution: execute an algorithm that *compacts* memory periodically to remove fragmentation
  - only possible if addresses are bound at run-time
  - expensive operation - takes time to translate the address spaces of most if not all processes, and CPU is unable to do other meaningful work during this compaction