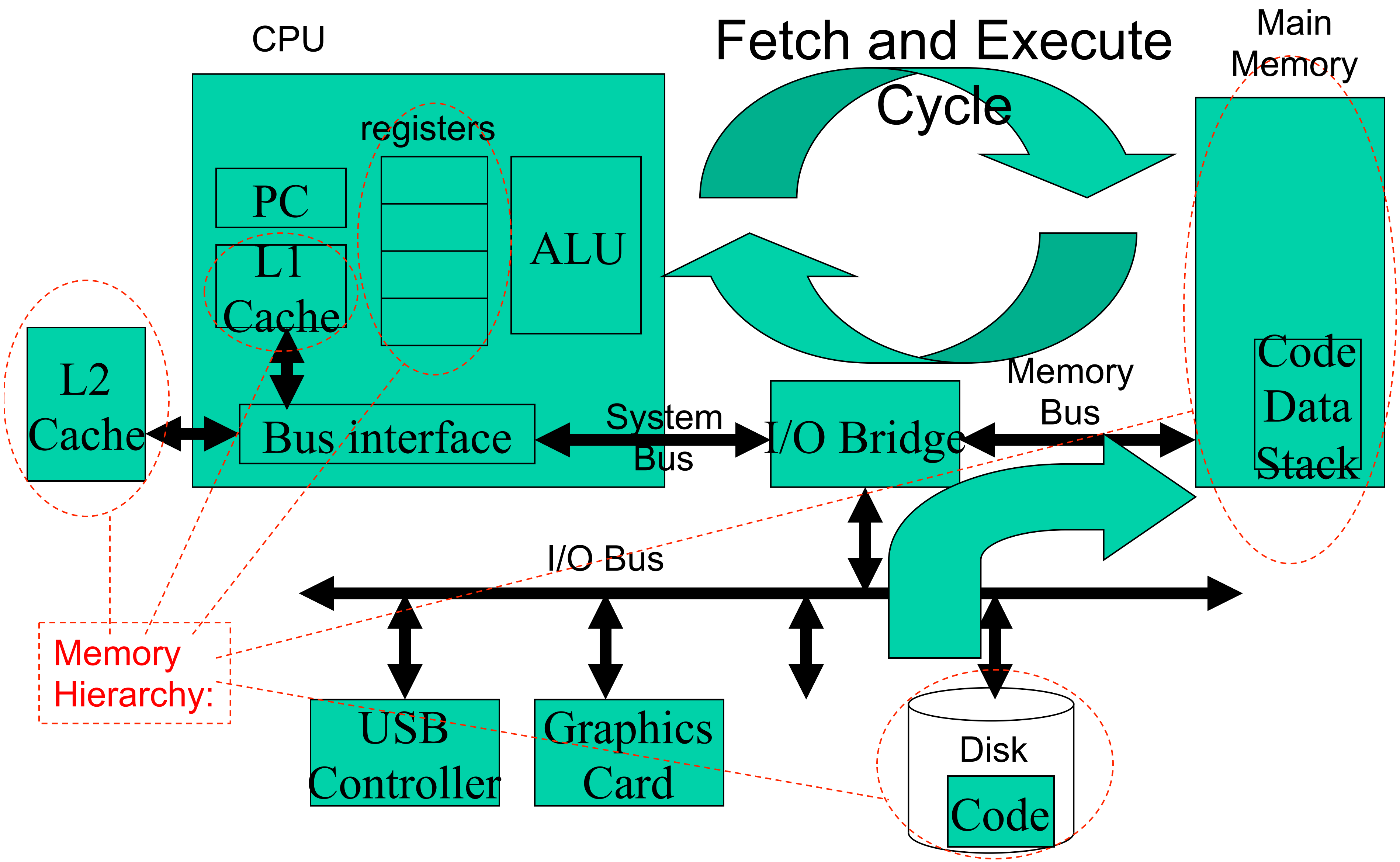
CSCI 3753
Operating Systems

Memory Management

Chapters 8 and 9

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 03/11/14

Memory Management



Memory Hierarchy

- CPU registers: fast and expensive
- L1, L2 and L3 cache between main memory and CPU
  - L1 = 1 clock cycle (~16 KB)
  - L2 = 4-5 clock cycles (~1 ns) (~1 MB)
  - L3 caches often shared between cores ~40 clock cycles (~10 ns) (~8-256 MB)
- RAM = ~100 cycles/~10-50 ns
- Permanent storage:
  - Flash = 10-100 μs (depends on read/write, flash type, etc.)
  - Disk = 10 ms

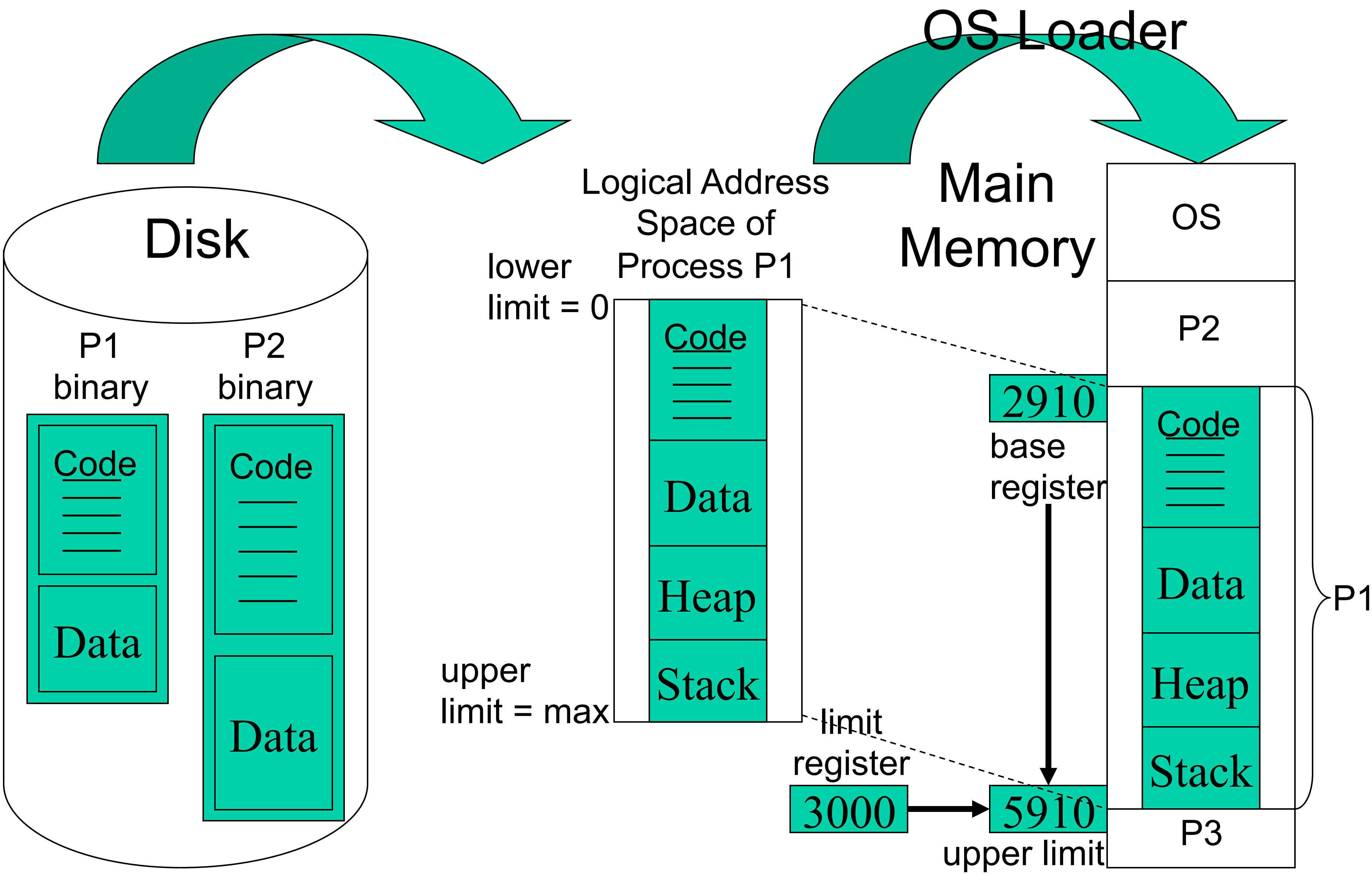
Memory Hierarchy

- Most modern CPUs have multiple types of caches
  - Instruction cache
  - Data cache
  - TLB for caching page table entries
  - e.g. AMD Athlon K8 has 64 KB L1 instruction cache, 64 KB L1 data cache, and 4 KB L1 TLB, and 1 MB L2 cache
- Different types of caches:
  - Fully Associative (cached item can go anywhere in cache) vs. Direct-mapped (cached item can only go in a slot/bin corresponding to its memory address) vs. N-way Set Associative (in between, N=2,4 usually, there are N items per bin in a cache)

Memory Hierarchy

- Different types of caches: (cont.)
  - Write-through (changes to caches written immediately) vs write-behind (lazy writes at some later time)
  - Strictly inclusive (all data in L1 must be in L2) vs. exclusive (data can only appear in either L1 or L2, not both) vs. mainly inclusive (some data can appear in both)
- Cache replacement policies:
  - When the cache gets full, need to find an entry to evict
  - A common approach is Least Recently Used (LRU)
    - Can be implemented by incrementing counters for each cache item on each cache hit and zeroing the most recently fetched/hit item, so that the least recently used item has the highest counter while more recently used items will have been zeroed more recently and thus have lower counter values – we'll see this more later

Memory Management



Memory Management

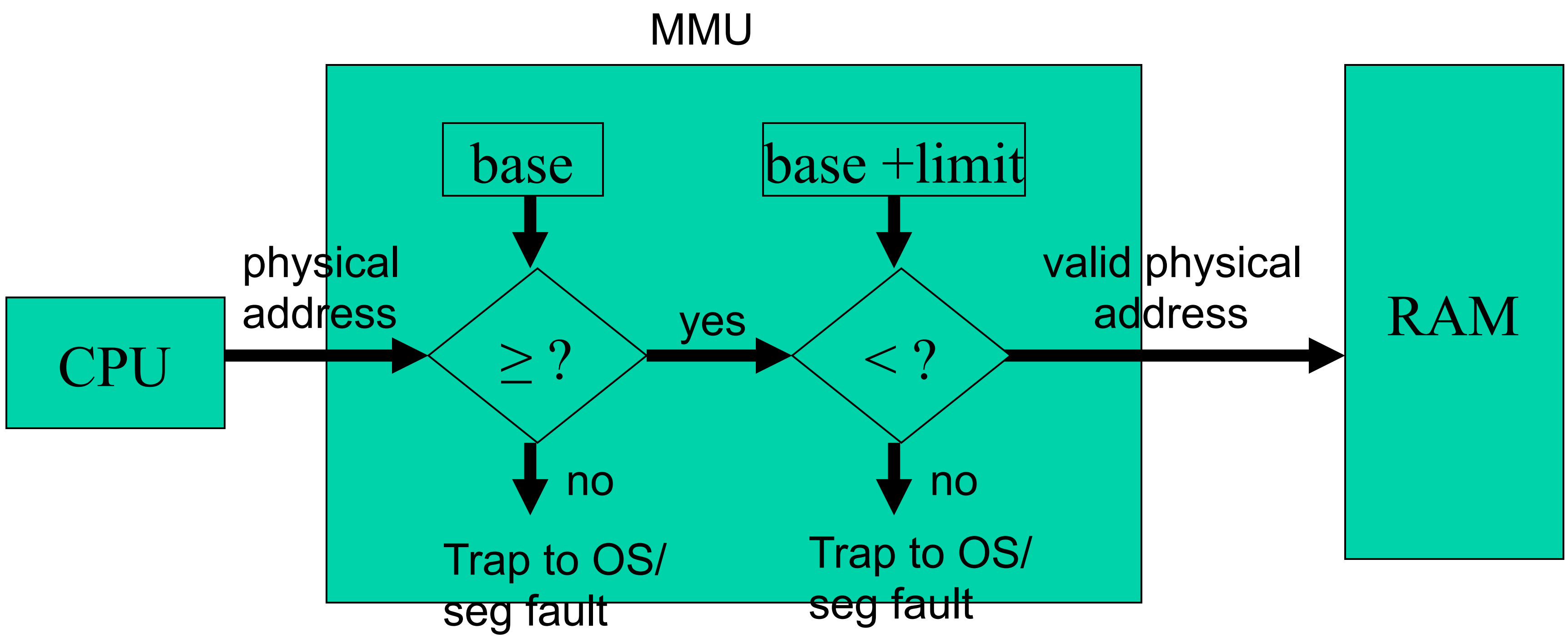
- When program P1 becomes an active process P1, then the process P1 can be viewed as conceptually executing in a *logical address space* ranging from *0* to *max*
  - In reality, the code and data are stored in physical memory
  - There needs to be a mapping from logical addresses to physical addresses - *memory management unit* (MMU) takes care of this.
- MMU
  - Address translation: translate logical addresses into physical addresses, i.e. map the logical address space into a *physical address space*
  - Bounds checking: check if the requested memory address is within the upper and lower limits of the address space
    - *base register* and *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



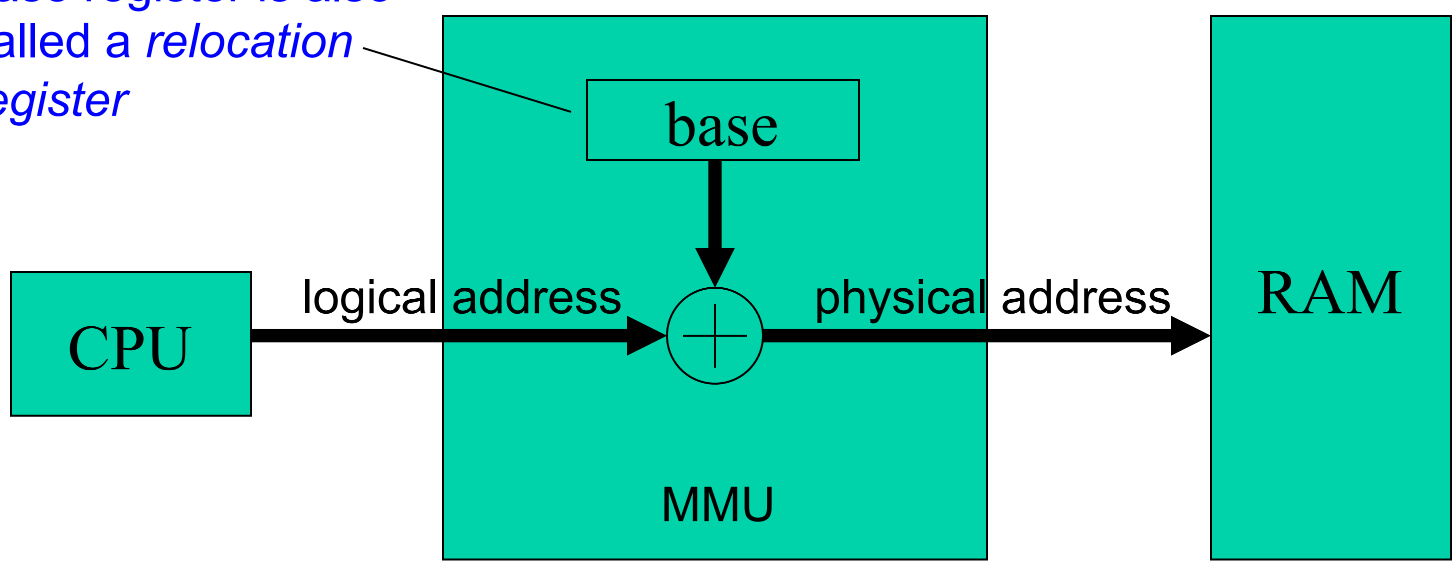
Address Binding

- 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 → *load module*
- Run Time/Execution Time: (most modern OS's do this)
  - Code is first compiled in relocatable format as if executing in its own logical/virtual address space. As each instruction is executed, i.e. at run time, the MMU relocates the logical address to a physical address using hardware support such as base/relocation registers.

Memory Management

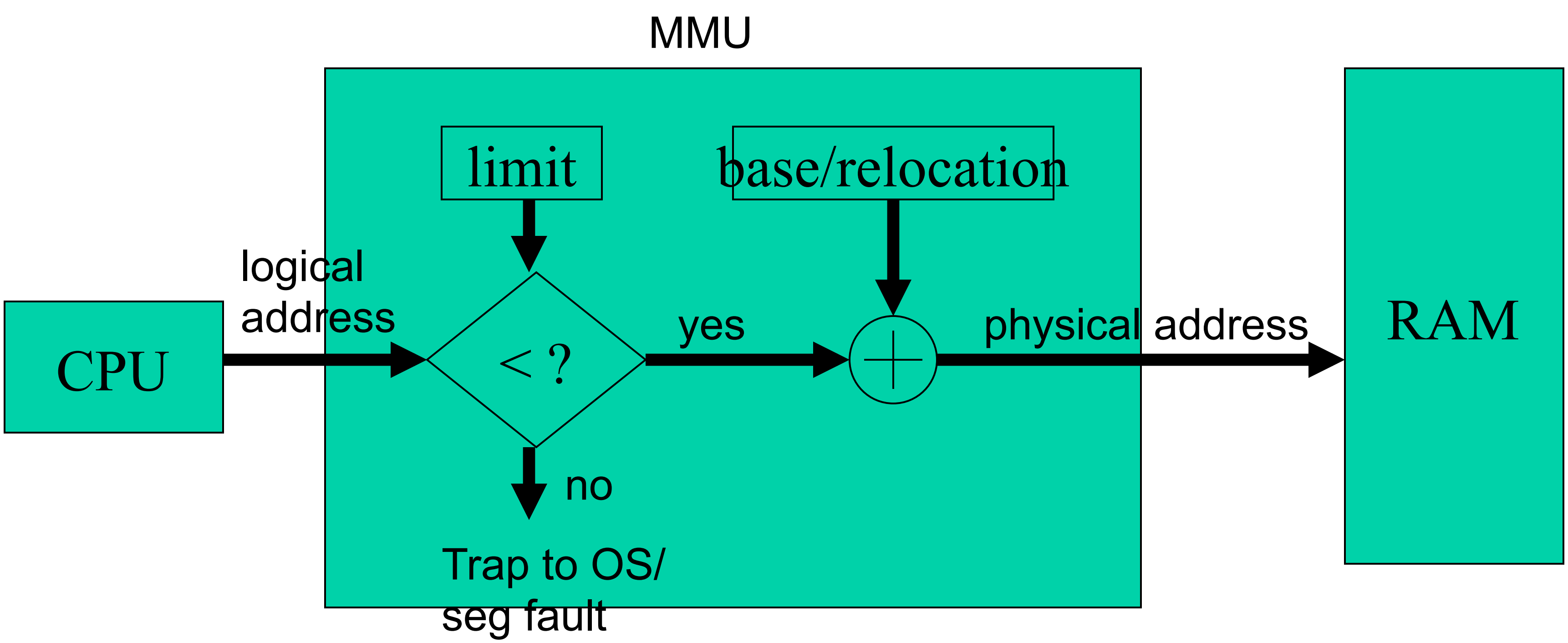
- For run-time address binding, MMU performs run-time *mapping* of logical addresses to physical addresses
  - each logical address is *relocated or translated* to a physical address that is used to access main memory
  - The application program never sees the actual physical memory - it just presents a logical address to MMU

base register is also called a *relocation register*



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
  - Also, by checking the limit first, no need to do relocation if out of bounds



Execution/Run Time Binding With Static Linking

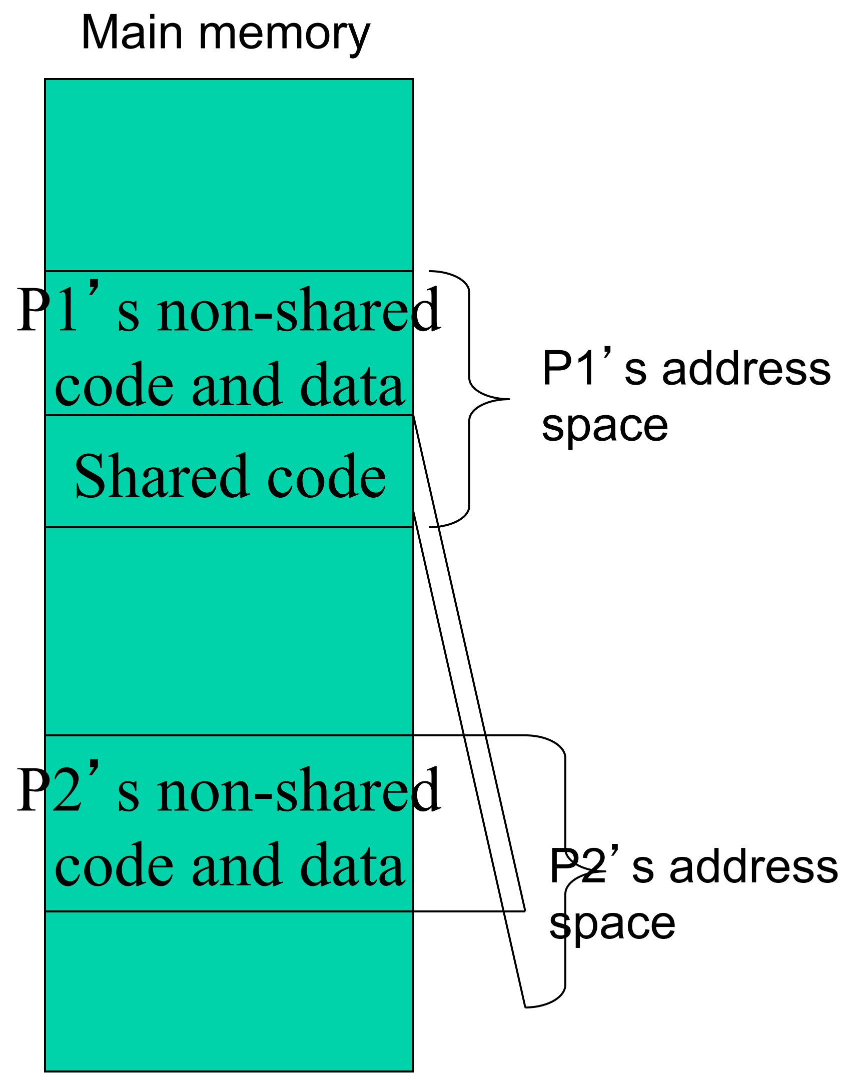
- Logical addresses are translated instruction by instruction into physical addresses at run time, and the entire executable has all the code it needs at compile time through static linking
- Static linking can cause each application to have a copy of the same code
  - e.g. C library – not very efficient

Run Time Binding with Dynamic Linking

- Executable code does not have all the code it needs at compile time
  - At compile time, include only a *stub* that contains info on how to find the dynamically linked library function
  - At run time, translate logical to physical addresses instruction by instruction, but when hitting a stub, the OS looks for the dll function, loads it if it's not already loaded, and replaces itself with a reference to the actual function
    - Dll is written as position-independent and reentrant code, so it can be put anywhere in memory and executed by multiple processes
    - In UNIX, dynamically linked libraries are typically named with a .so suffix, i.e. libfoo.so

Run Time Binding with Dynamic Linking

- One copy of the code shared among all programs
  - We'll see later how code is shared between address spaces
- Programs have access to the most recently patched dll's
  - Compare to a statically compiled executable that may have a much older version of code when it was originally compiled years ago
- Smaller size – stubs stay stubs unless activated

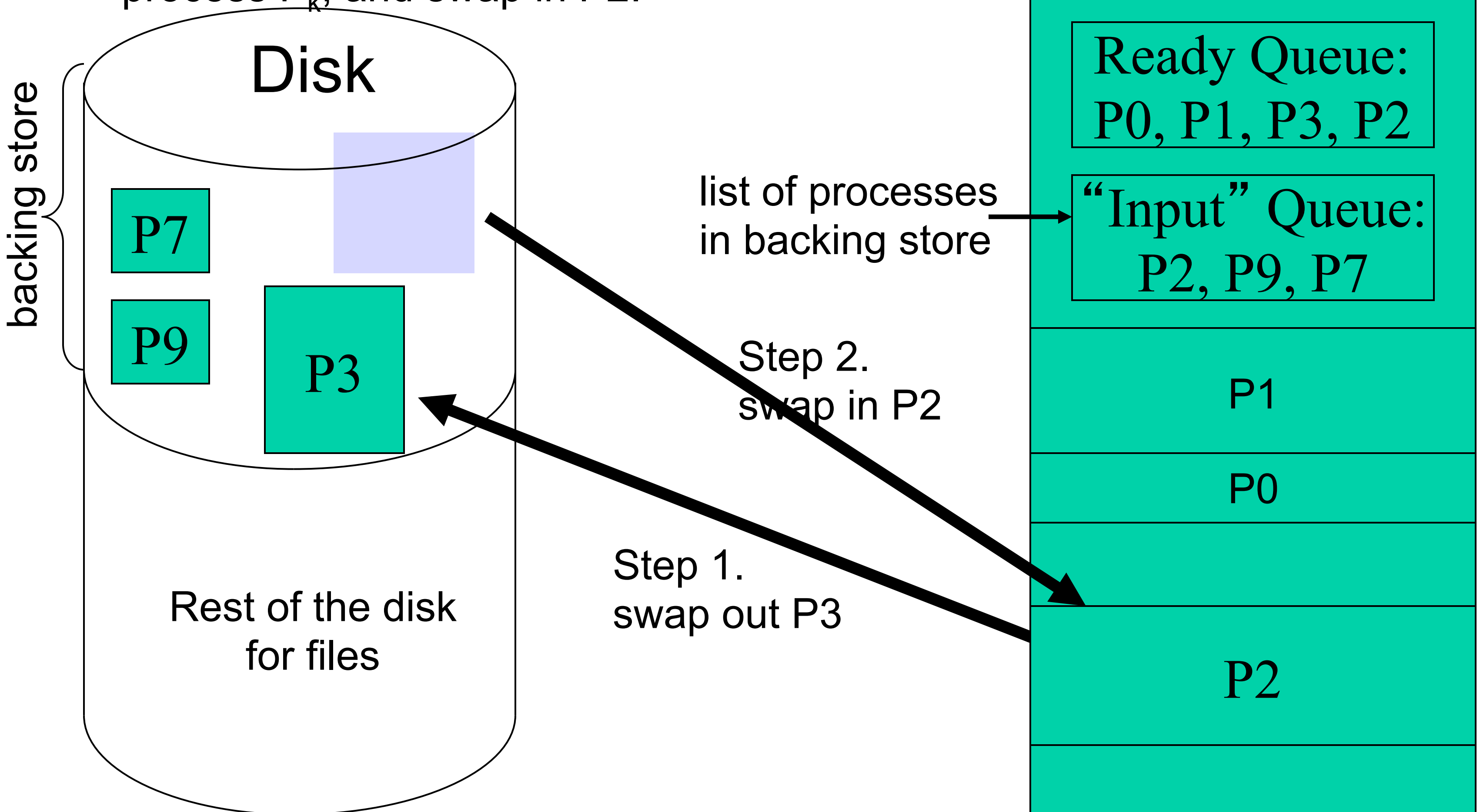


Swapping

- 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 allocated for this is called *backing store* or *swap space*. This is faster to access – don't go through the normal file system.
- 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

Swapping

- Swapping: when the OS scheduler wants to run a process 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.

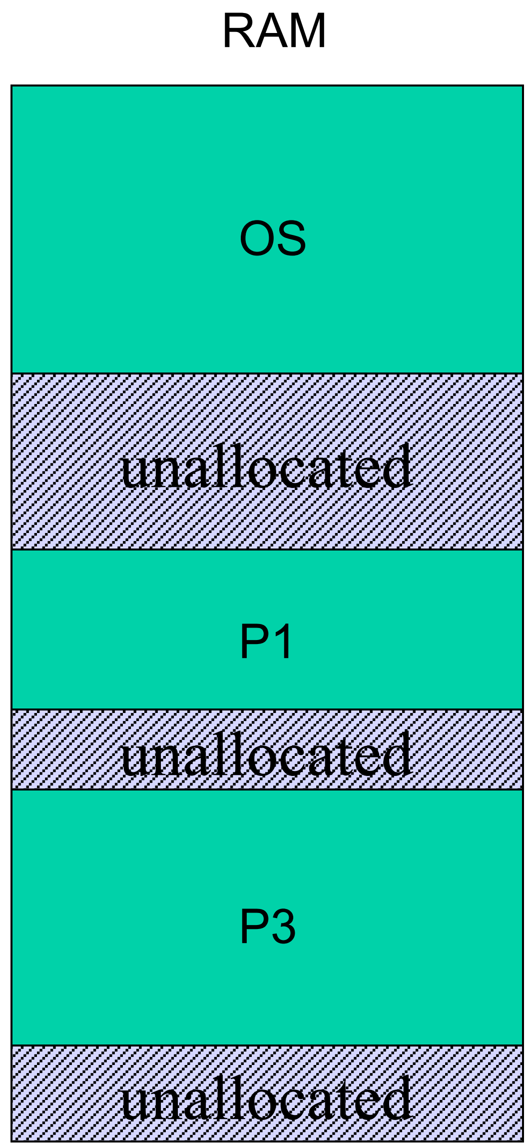


Problem with Swapping

- Context-switch time of swapping is very slow
  - On the order of tens to hundreds of milliseconds
  - Hide this latency by having other processes to run while swap is taking place, e.g. in RR, swap out the just-run process, and have enough processes in round robin to run before swap-in completes
  - 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
- 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
- *fragmentation* of main memory becomes a big issue
  - can also get fragmentation of backing store disk

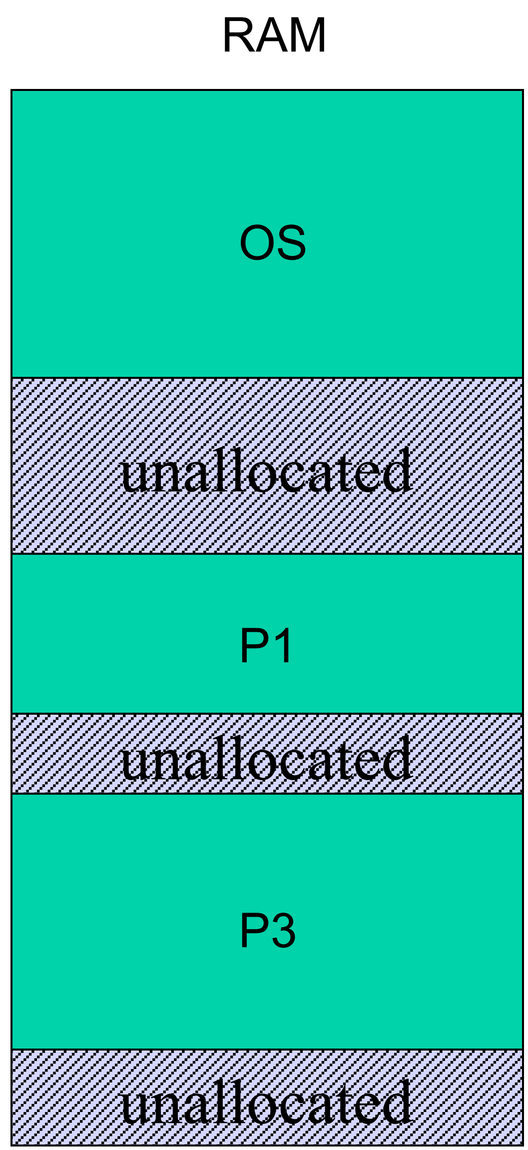
Memory Allocation

- 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.



Memory Allocation Strategies

- *best fit* - find the smallest chunk that is big enough
  - This results in more and more fragmentation
- *worst fit* - find the largest chunk that is big enough
  - This leaves the largest contiguous unallocated chunk for the next process
- *first fit* - find the 1st chunk that is big enough
  - This tends to fragment memory near the beginning of the list
- *next fit* – view fragments as forming a circular buffer, find the 1st chunk that is big enough after the most recently chosen fragment



Memory Allocation

- 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