Chapter 8: Memory Management

CSCI 3753 Operating Systems

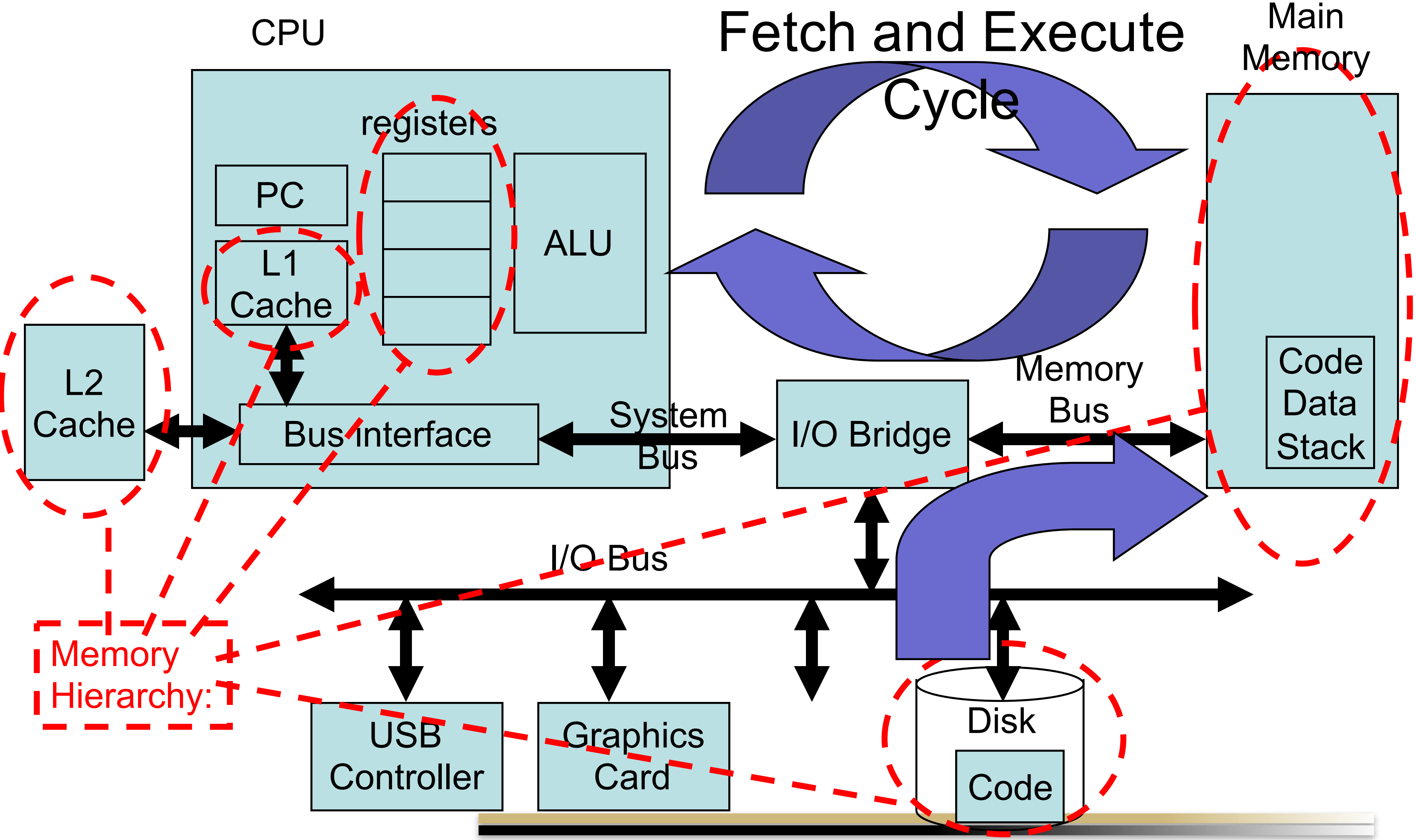
William Mortl & Dr. Rick Han

University of Colorado **Boulder**

Pre/Post-midterm Topics

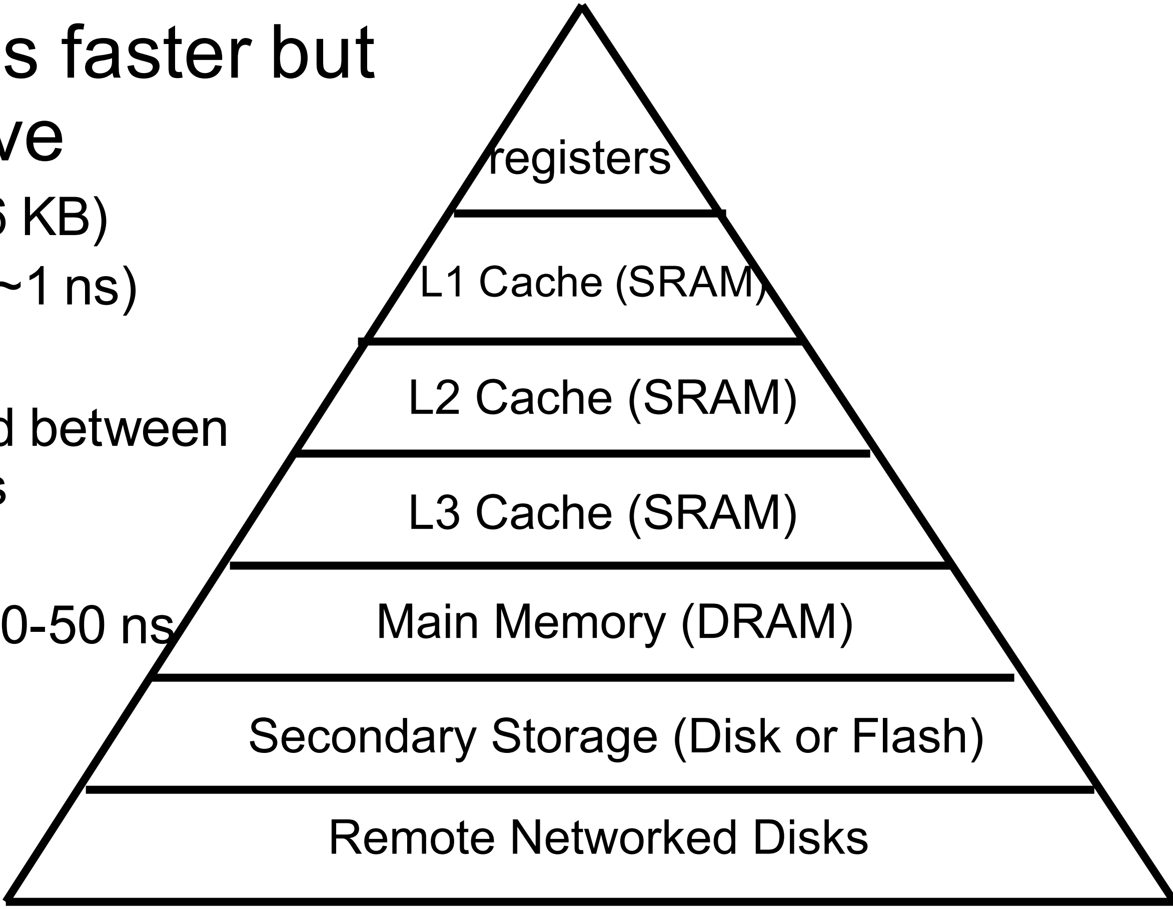
- Pre-midterm:
  - – Introduction to OS, Chapters 1-2
  - – Managing I/O devices, Chapters 13
  - – Managing processes and threads, Chapters 3-7
    - Scheduling
    - Synchronization
- Post-midterm:
  - – *Managing memory – Chapters 8-9*
  - – Managing files – Chapters 10-12
  - – Other topics: security, networking, VMs, …

Memory Management



Memory Hierarchy

- cache frequently accessed instructions and/or data in local memory that is faster but also more expensive
  - 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
      (10^7 ns)

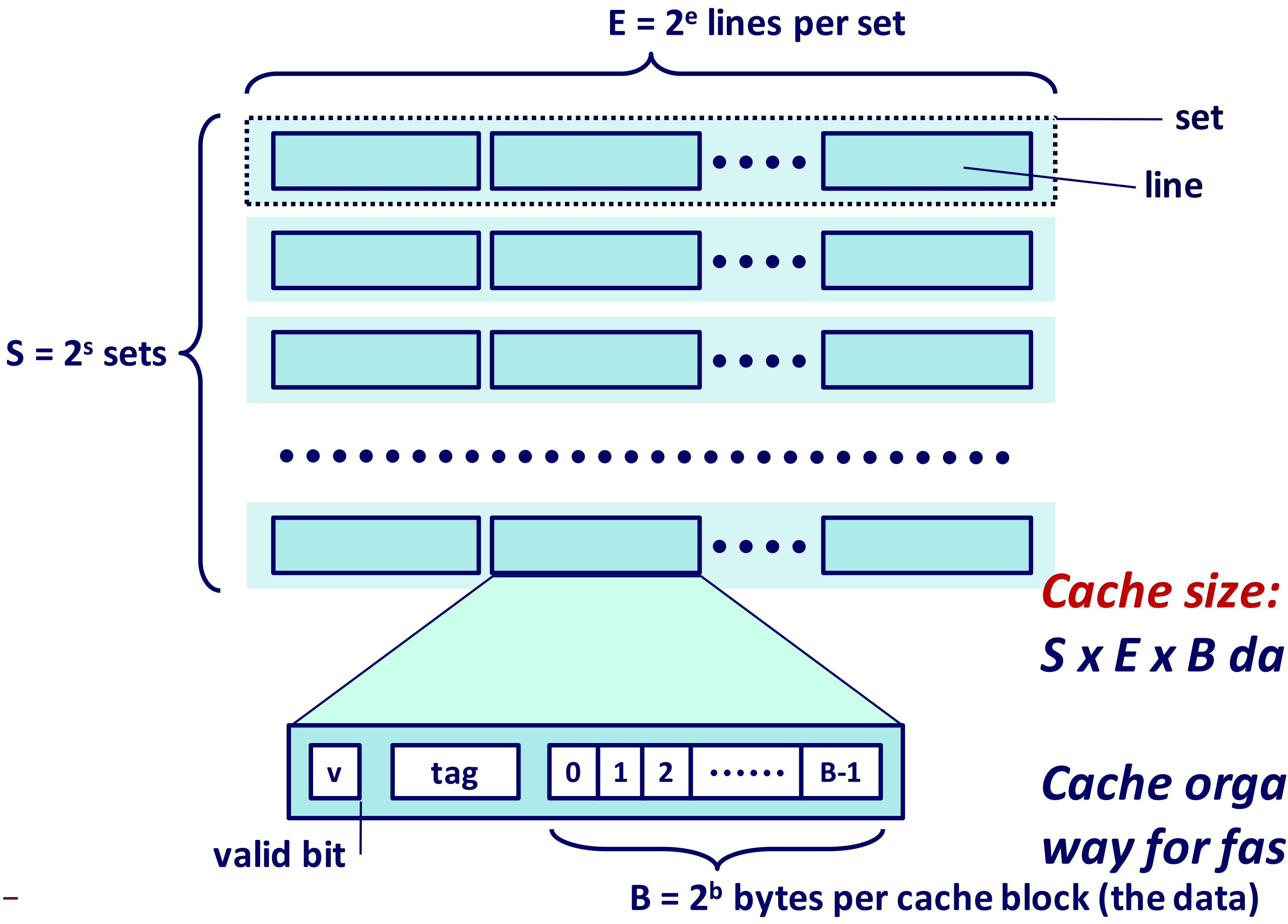


Memory Hierarchy

- 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

General Cache Organization (S, E, B)

For L1 hardware caches, access must be fast, so organize as follows:



Cache size:

S x E x B data bytes

Cache organized this way for fast access

Memory Hierarchy

- Different types of caches: hardware layout
  - *Fully Associative* = one row
    - cached item can go anywhere in cache row
  - *Direct-mapped* = one column
    - cached item can only go in a row slot/bin corresponding to its memory address
  - vs. *N-way Set Associative* = N columns and M rows
    - in between, 2,4 usually, there are N items per bin in a cache

Memory Hierarchy

- Different types of caches: write behavior
  - *Write-through* on a cache hit: changes to caches written immediately to memory (and in-between caches)
    - Good for consistency
    - Bad: may take some time to complete
  - *Write-back* on a hit (also called write-behind): lazy writes to memory at some later time, e.g. replacement of a cache line
    - Good: fast response in writing
    - Bad: inconsistency between caches and possible loss of cached data

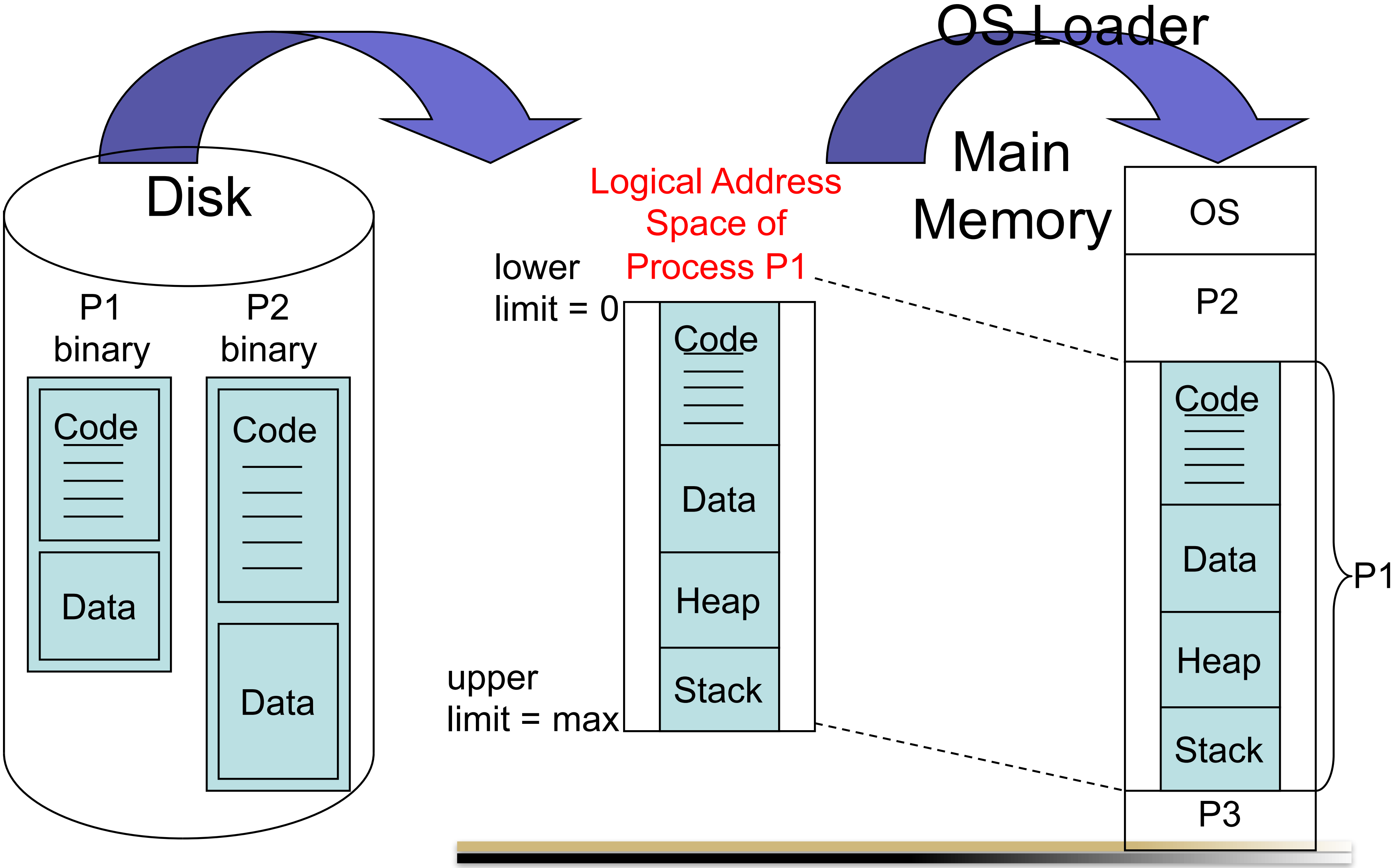
Memory Hierarchy

- Different types of caches: write behavior (cont.)
  - *Write-allocate* on a miss: load into cache, update line in cache
    - Good if temporal/spatial locality
  - *No-write-allocate* on a miss: write immediately to memory
  - Pairing of write-through with no-write-allocate
    - Writes go to main memory
  - Pairing of write-back with write-allocate
    - Writes go to cache

Memory Hierarchy

- Cache replacement policies:
  - When the cache gets full, need to find an entry to evict
  - A common approach is to use *Least Recently Used (LRU)*
    - Evict the data item that has not been read/written for the longest time
    - Can be implemented by incrementing counters for each cache item on each cache hit and zeroing the most recently fetched/hit item,
    - least recently used item has the highest counter
    - 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

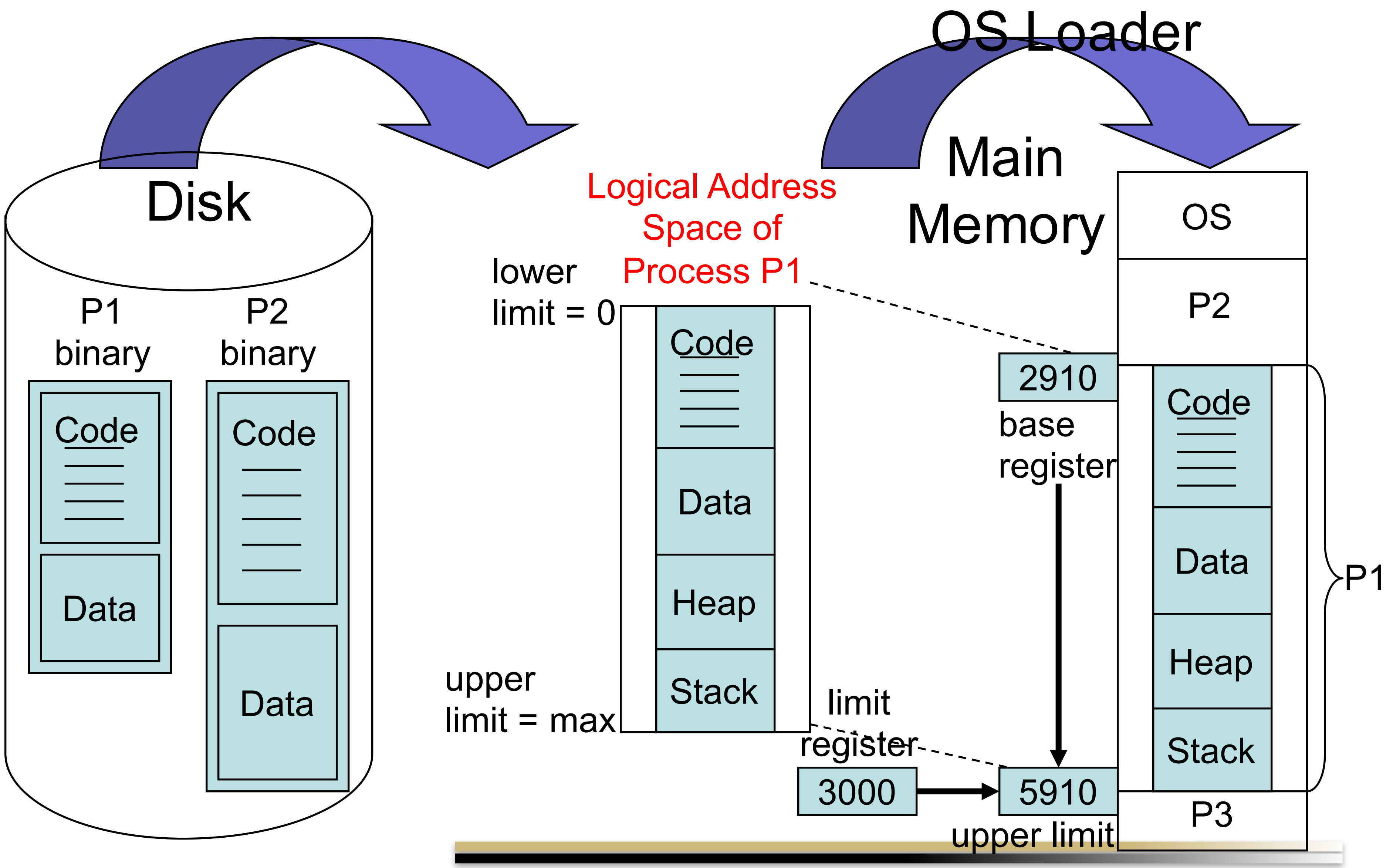
- In the previous figure, want newly active process P1 to execute in its own *logical address space* ranging from 0 to max
  - It shouldn't have to know exactly where in physical memory its code and data are located
  - This decouples the compiler from run-time execution
  - There needs to be a mapping from logical addresses to physical addresses at run time - memory management unit (MMU) takes care of this.

Memory Management

- MMU must do:
  1. Address translation: translate logical addresses into physical addresses, i.e. map the logical address space into a *physical address space*
  2. Bounds checking: check if the requested memory address is within the upper and lower limits of the address space

  One approach is:
  - *base register* in hardware 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

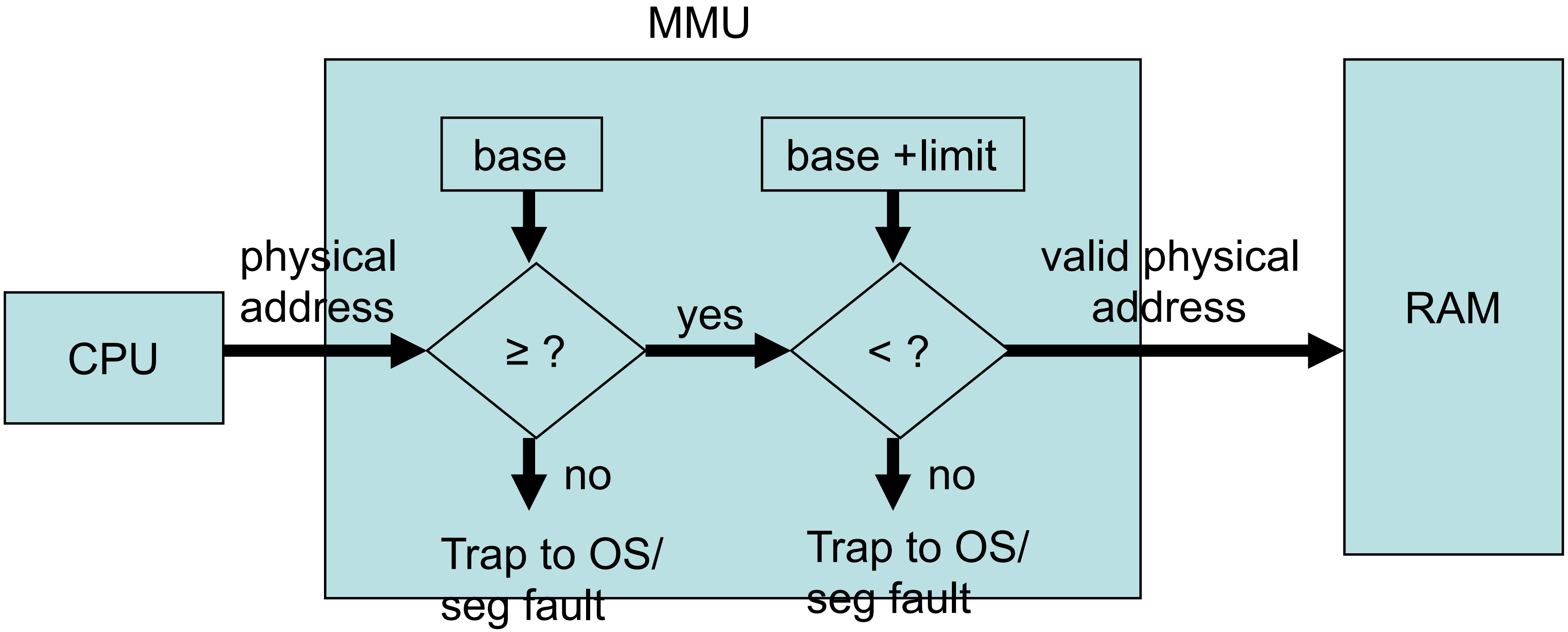


Memory Management

- 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

- Address Binding at Compile Time:
  - If you know in advance where in physical memory a process will be placed, then compile your code with absolute physical addresses
  - Example: LOAD MEM_ADDR_X, reg1

    STORE MEM_ADDR_Y, reg2

  MEM_ADDR_X and MEM_ADDR_Y are hardwired by the compiler as absolute physical addresses

Memory Management

- Address Binding at Load Time
  - Code is first compiled in relocatable format. Then replace logical addresses in code with physical addresses during loading
  - Example: LOAD MEM_ADDR_X, reg1

    STORE MEM_ADDR_Y, reg2

    At load time, the loader replaces all occurrences in the code of MEM_ADDR_X and MEM_ADDR_Y with (base+MEM_ADDR_X) and (base+MEM_ADDR_Y).
  - Once the binary has been thus changed, it is not really portable to any other area of memory, hence load time bound processes are not suitable for swapping (see later slides)

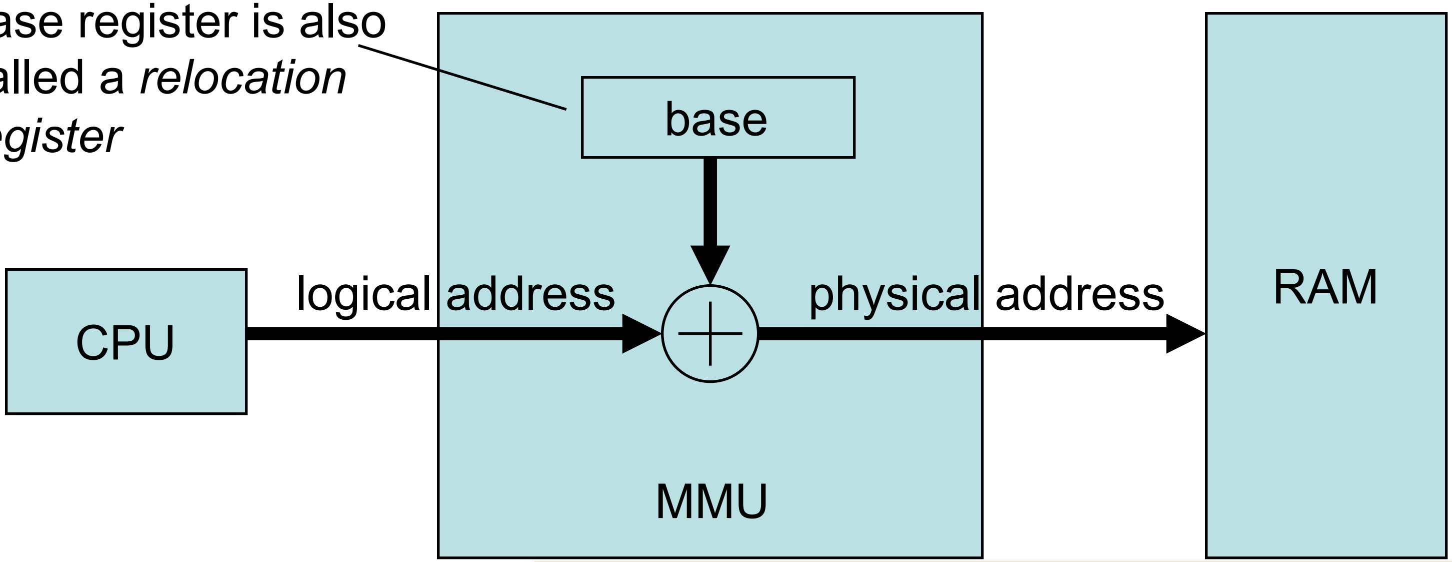
Memory Management

- Address Binding at Run 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.
  - Example: LOAD MEM_ADDR_X, reg1

    MEM_ADDR_X is compiled as a logical address, and implicitly the hardware MMU will translate it to base+MEM_ADDR_X when the instruction executes

Memory Management

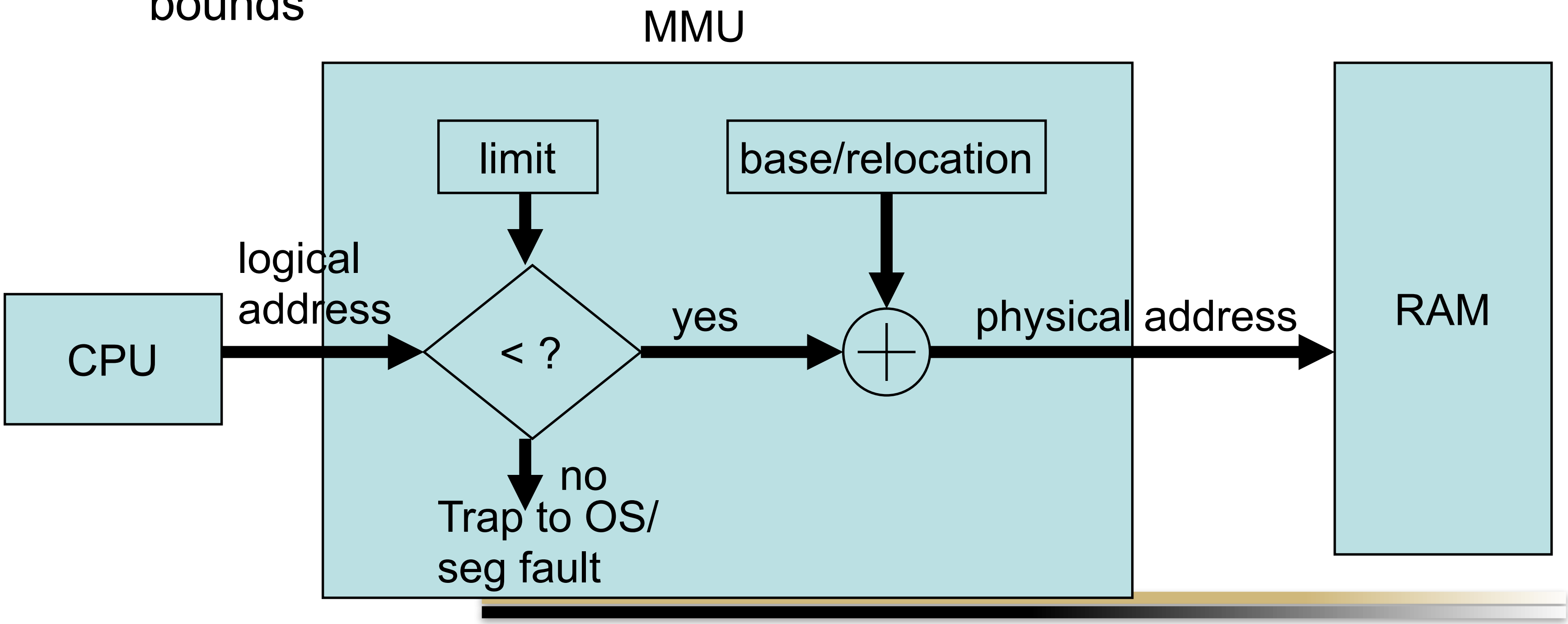
- For run-time address binding, MMU needs to perform run-time *mapping* of logical/virtual addresses to physical addresses
  - each logical address is *relocated or translated* by MMU to a physical address that is used to access main memory/RAM
  - thus the application program never sees the actual physical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



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

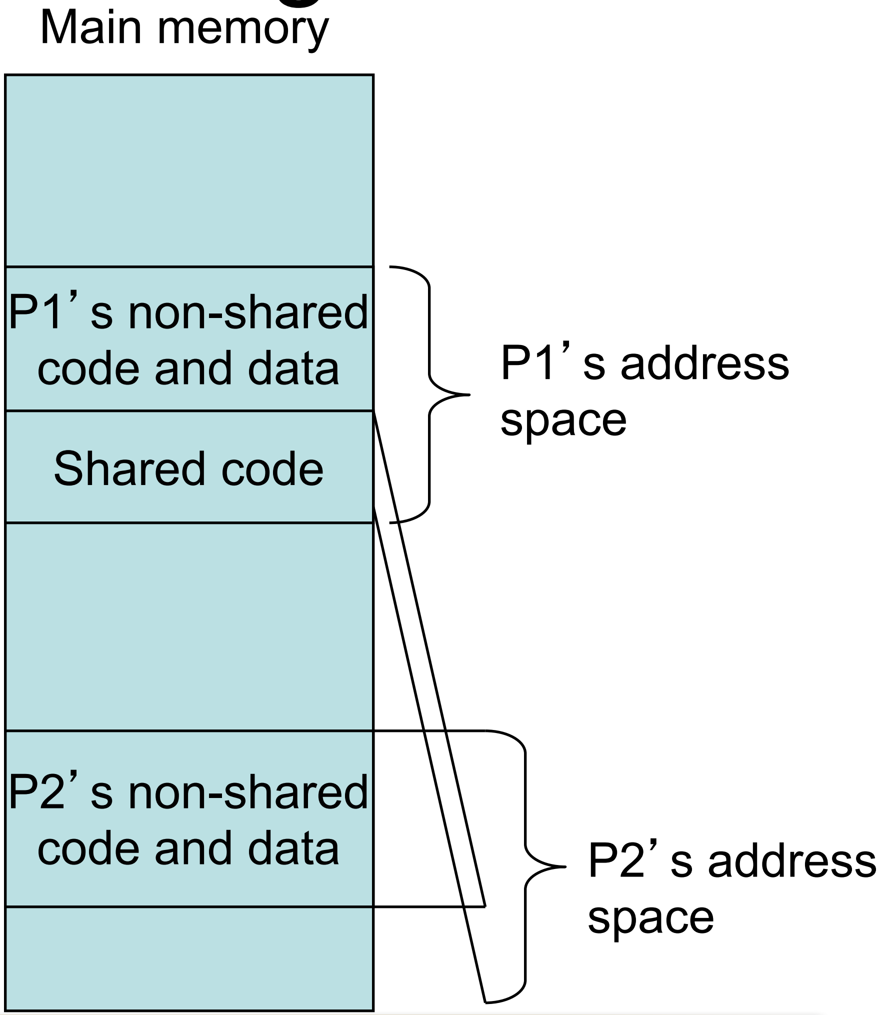
- Statically linked executable
  - Logical addresses are translated instruction by instruction into physical addresses at run time, *and the entire executable has all the code it needs at compile time through static linking*
  - Once a function is statically linked, it is embedded in the code and can't be changed except through recompilation
    - Your code can contain outdated functions that don't have the latest bug fixes, performance optimizations, and/or feature enhancements

Run Time Binding with Dynamic Linking

- With dynamic linking, the executable 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
    - UNIX dynamically linked libraries use a .so suffix, eg libfoo.so

Run Time Binding with Dynamic Linking

- Advantages of dynamic linking:
  - Applications have access to the latest code at run-time, e.g. most recent patched dlls,
  - Smaller size – stubs stay stubs unless activated
  - Can have only one copy of the code that is shared among all applications
    - We'll see later how code is shared between address spaces using page tables

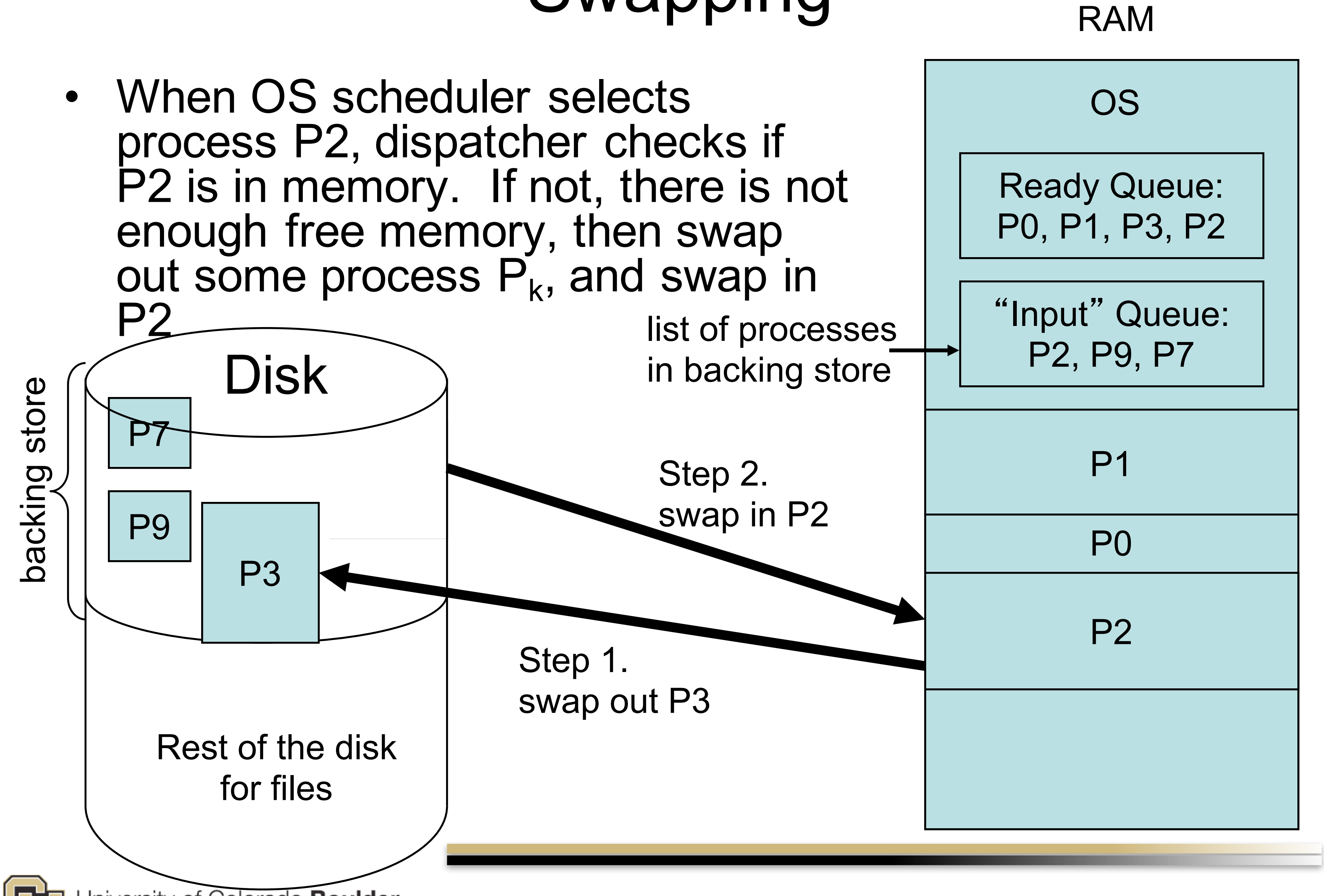


Swapping

- Main memory may not be able to store all processes that are in the ready queue
- Use disk/secondary storage to store some processes that are temporarily swapped out of memory
- Enables other (swapped in) processes to 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.

Swapping

- When OS scheduler selects process P2, dispatcher checks if P2 is in memory. If not, there is not enough free memory, then swap out some process P_k, and swap in P2



Swapping

- If run 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 Difficulties

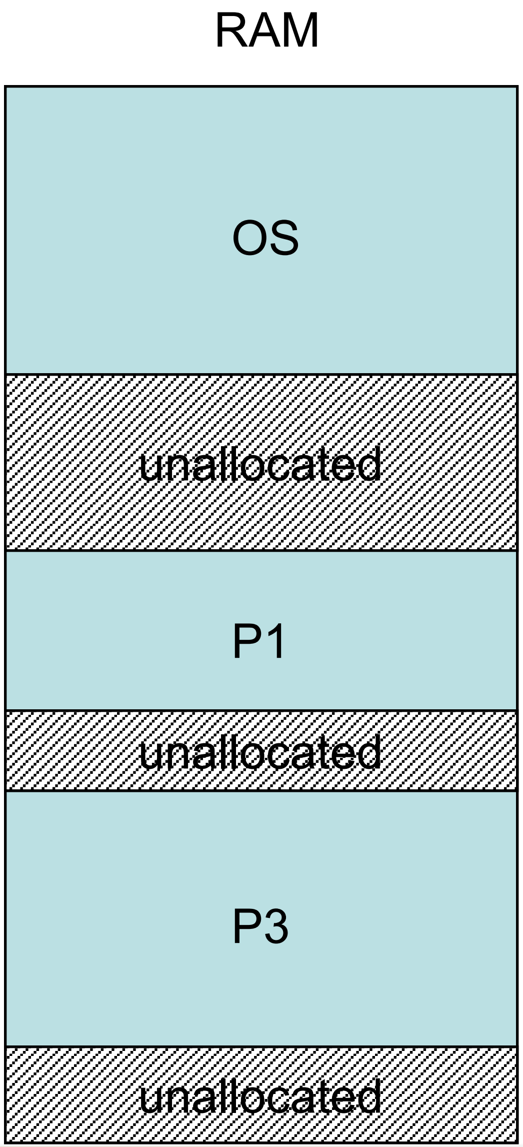
- context-switch time of swapping is very slow
  - Disks take on the order of 10s-100s of ms
  - When adding the size of the process to transfer, then transfer time can take seconds
  - Ideally hide this latency by having other processes to run while swap is taking place behind the scenes,
    - e.g. in RR, swap out the just-run process, and have enough processes in round robin to run before swap-in completes & newly swapped-in process is ready to run
  - 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 Difficulties

- 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 – see next slide
  - can also get fragmentation of backing store disk
- Modern OS's swap portions of processes in conjunction with virtual memory and demand paging

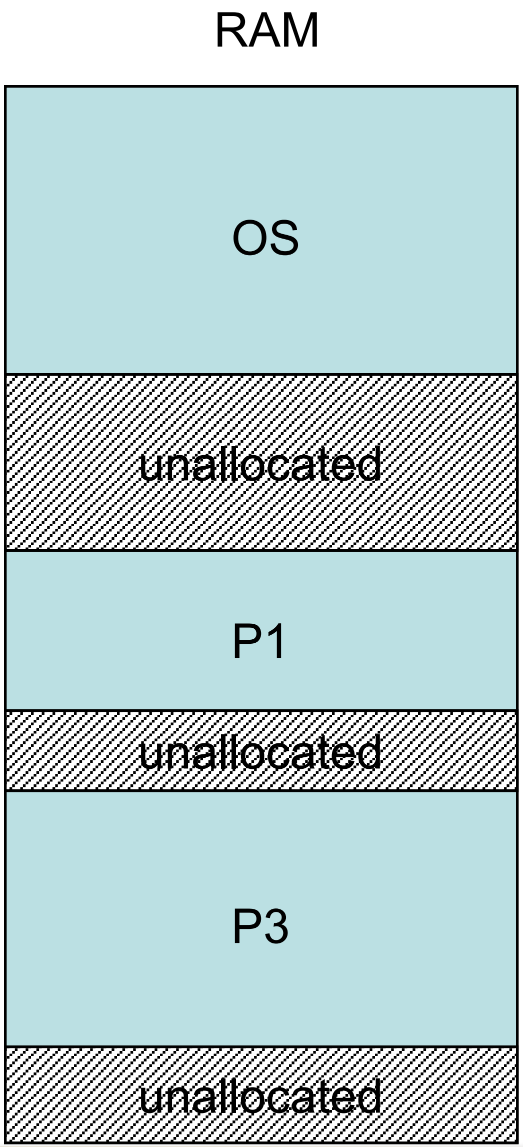
Allocation

- 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
  - There are many small chunks of non-contiguous unallocated memory between allocated processes in memory
- OS must find a large enough unallocated chunk in fragmented memory that a process will fit into



Allocation

- Multiple 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



Fragmentation

- though there is enough total free memory, there may not be one contiguous chunk of free memory large enough to fit a process
  - free memory is fragmented in small pieces in RAM
- Both first-fit & best-fit aggravate external fragmentation, less so with worst-fit
- Solution: periodically *compact/de-fragment* memory
  - only possible if addresses are bound at run-time
  - Expensive: translate the address spaces of most if not all processes, and CPU is unable to do other meaningful work during this compaction