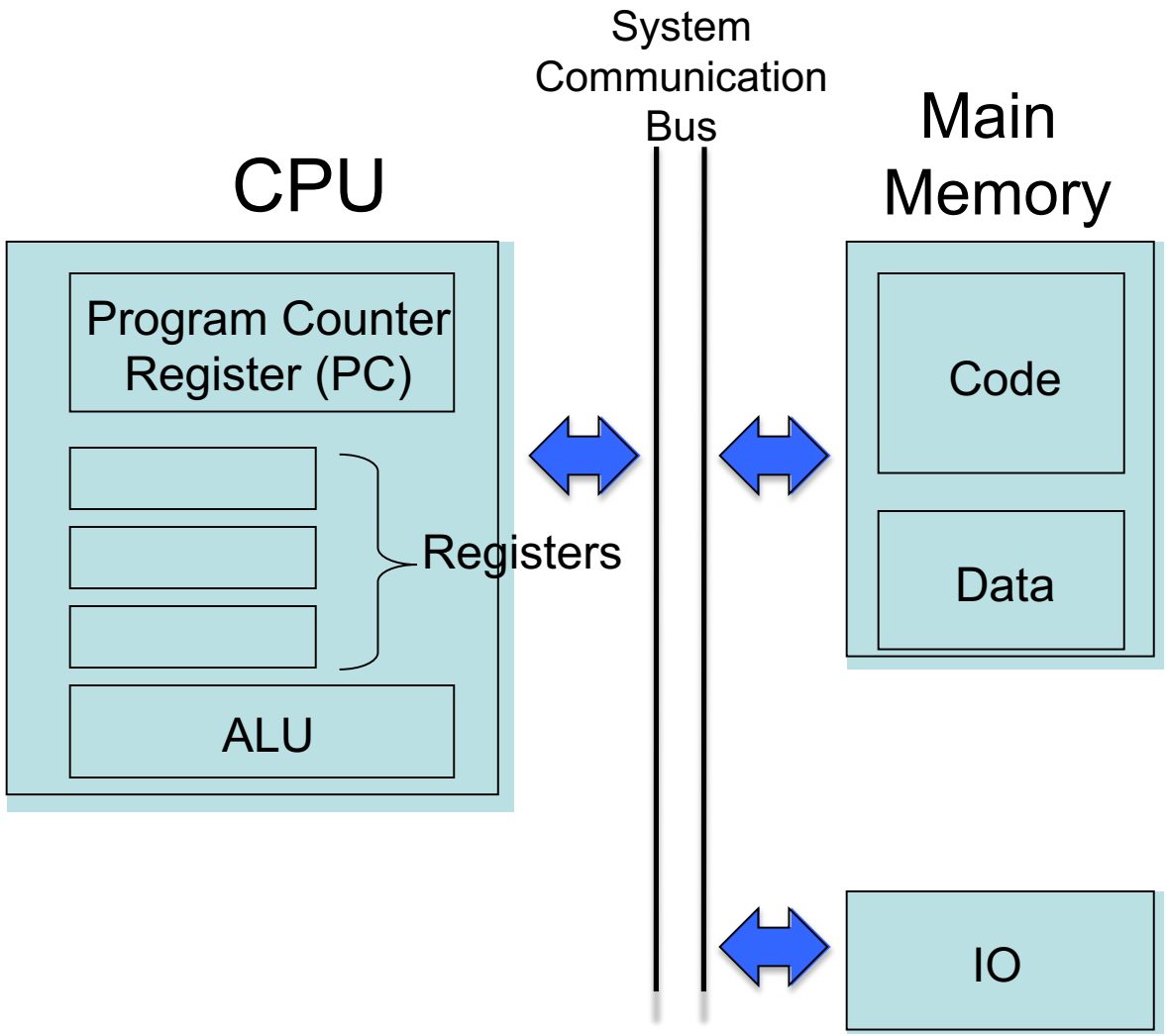
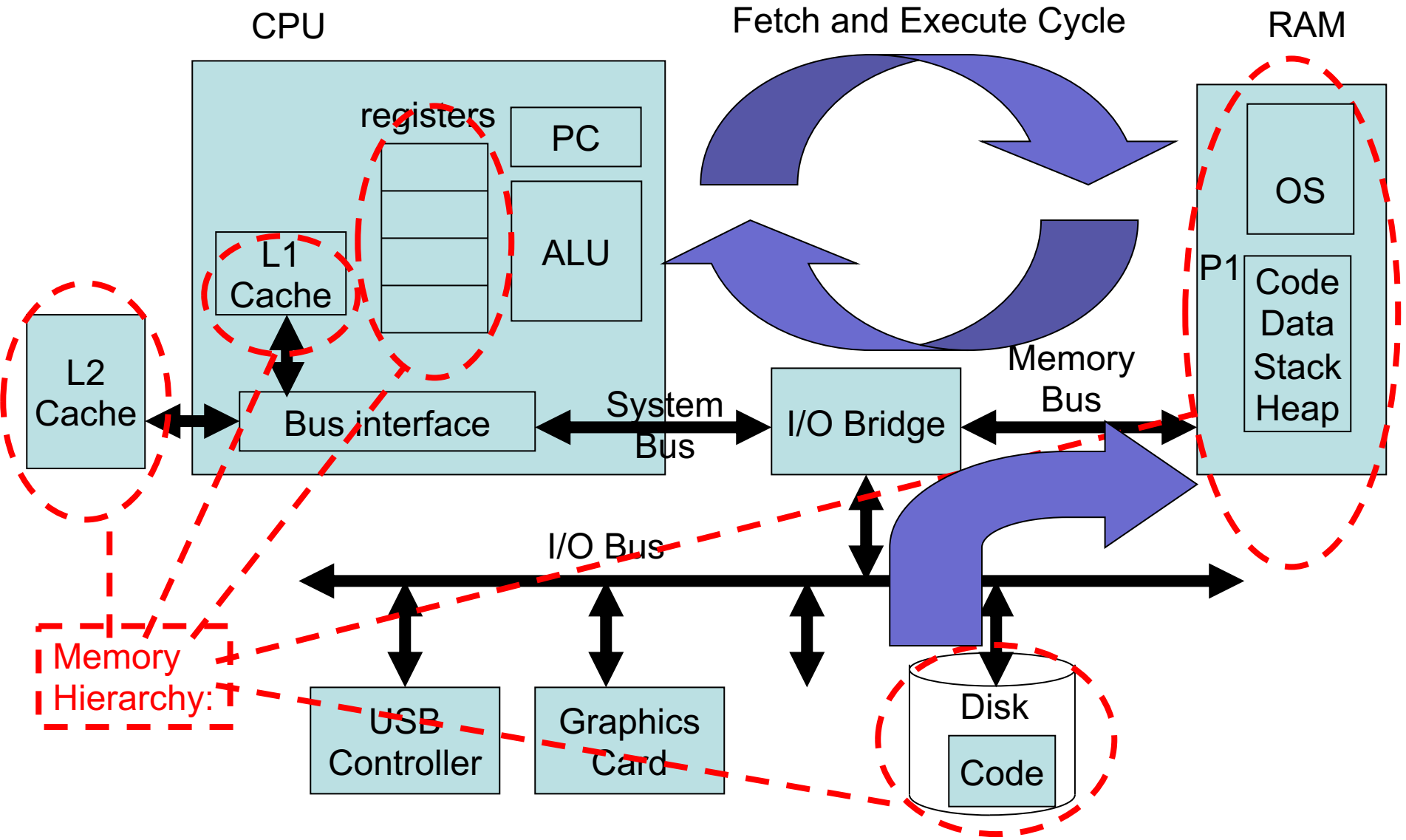
Lecture 16
Memory Management

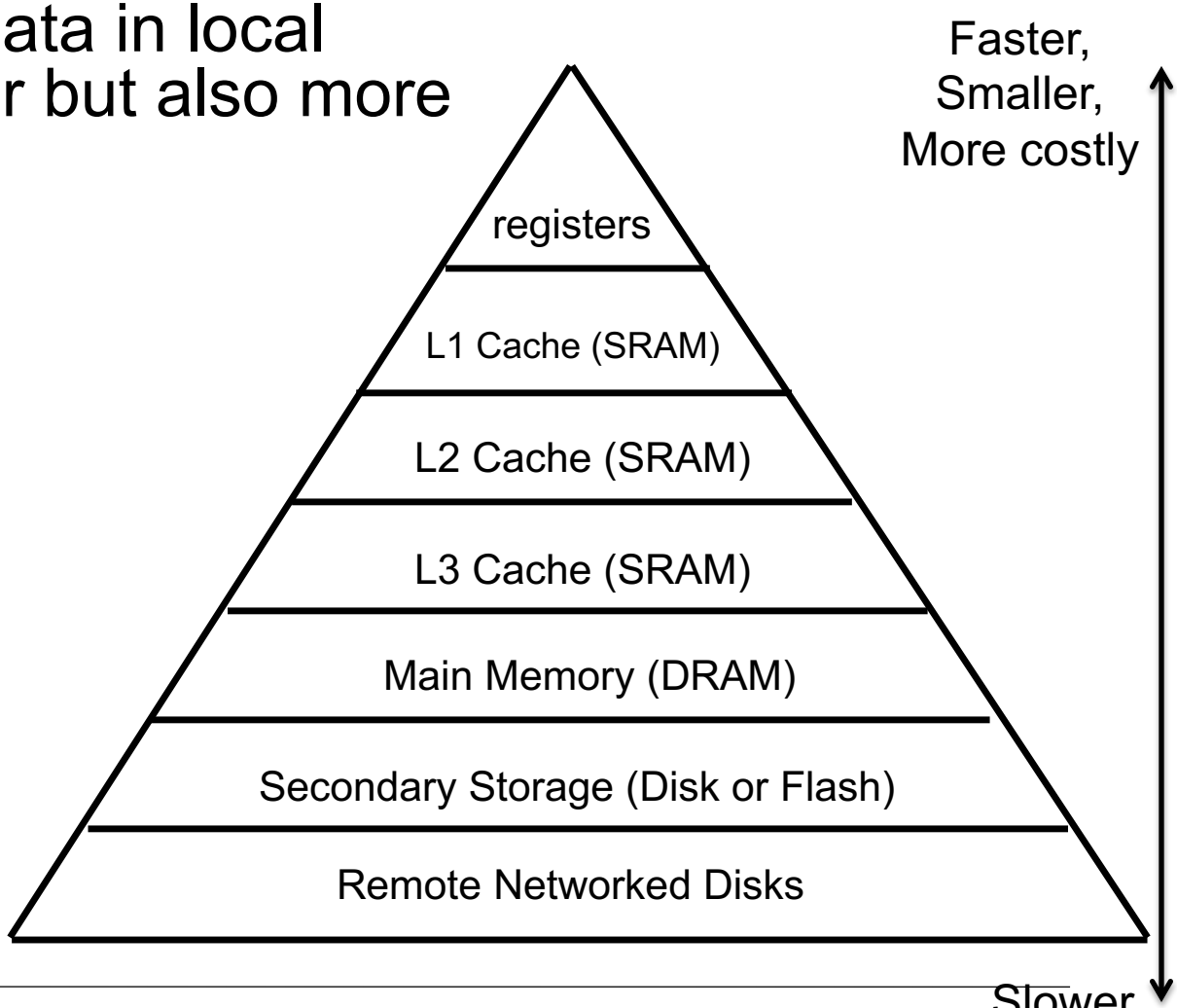

System
Communication
Bus
CPU
Main
Memory
Program Counter
Register (PC)
Registers
ALU
Code
Data
IO

Memory Management



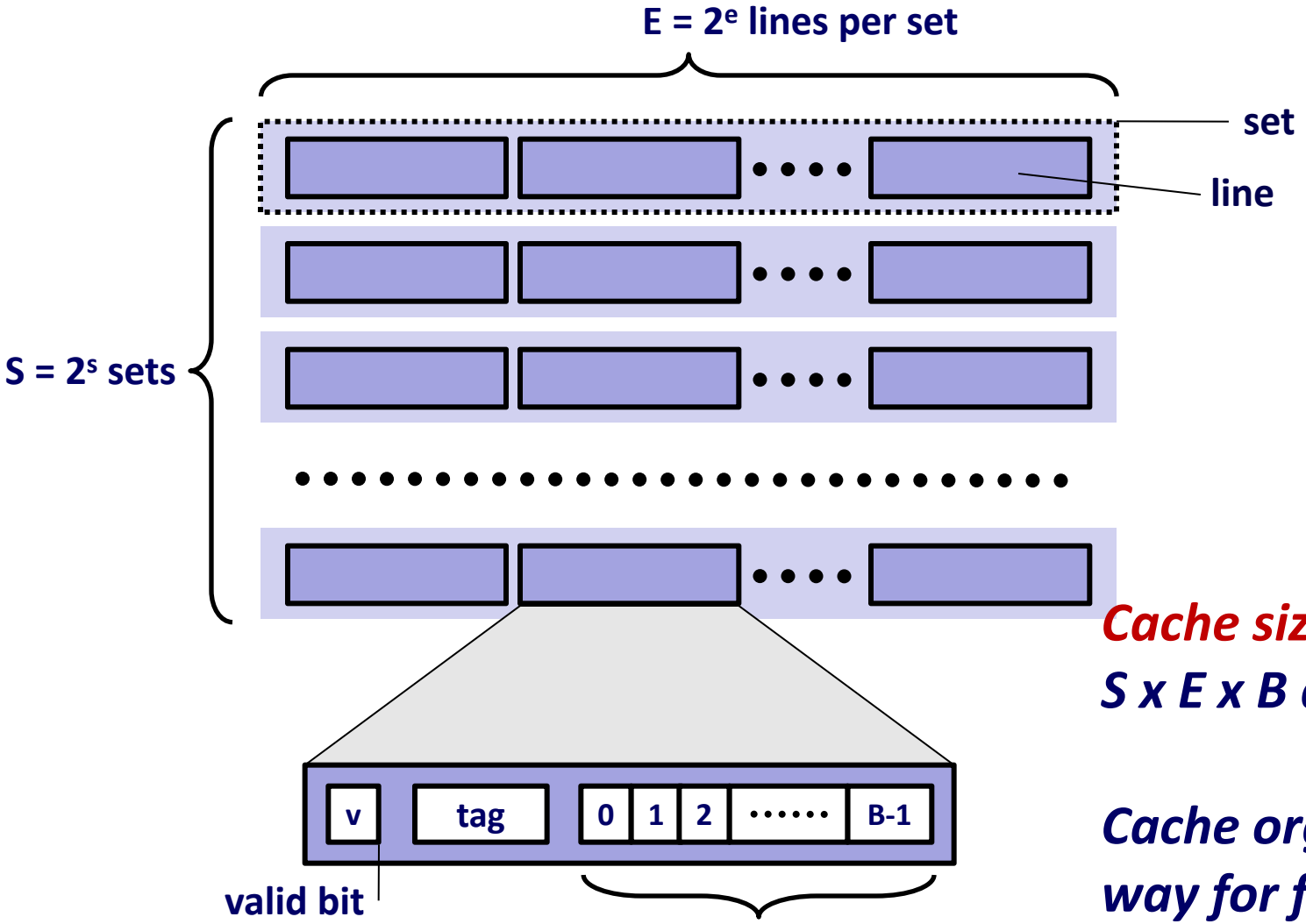
Memory Hierarchy

- cache frequently accessed instructions and/or data in local memory that is faster but also more expensive
  - Register < 1 cycle (16 B)
  - L1 = 1 cycle (~32 KB)
  - L2 = 3 cycles (~512 kB)
  - L3 = 10 cycles (2MB)
  - RAM = 200 cycles (GB)
  - Permanent storage:
    - Flash = 100k cycles (GB)
    - Disk =1M cycles (TB)
    - Network = 1B cycles (PB)



General Cache Organization (S, E, B)

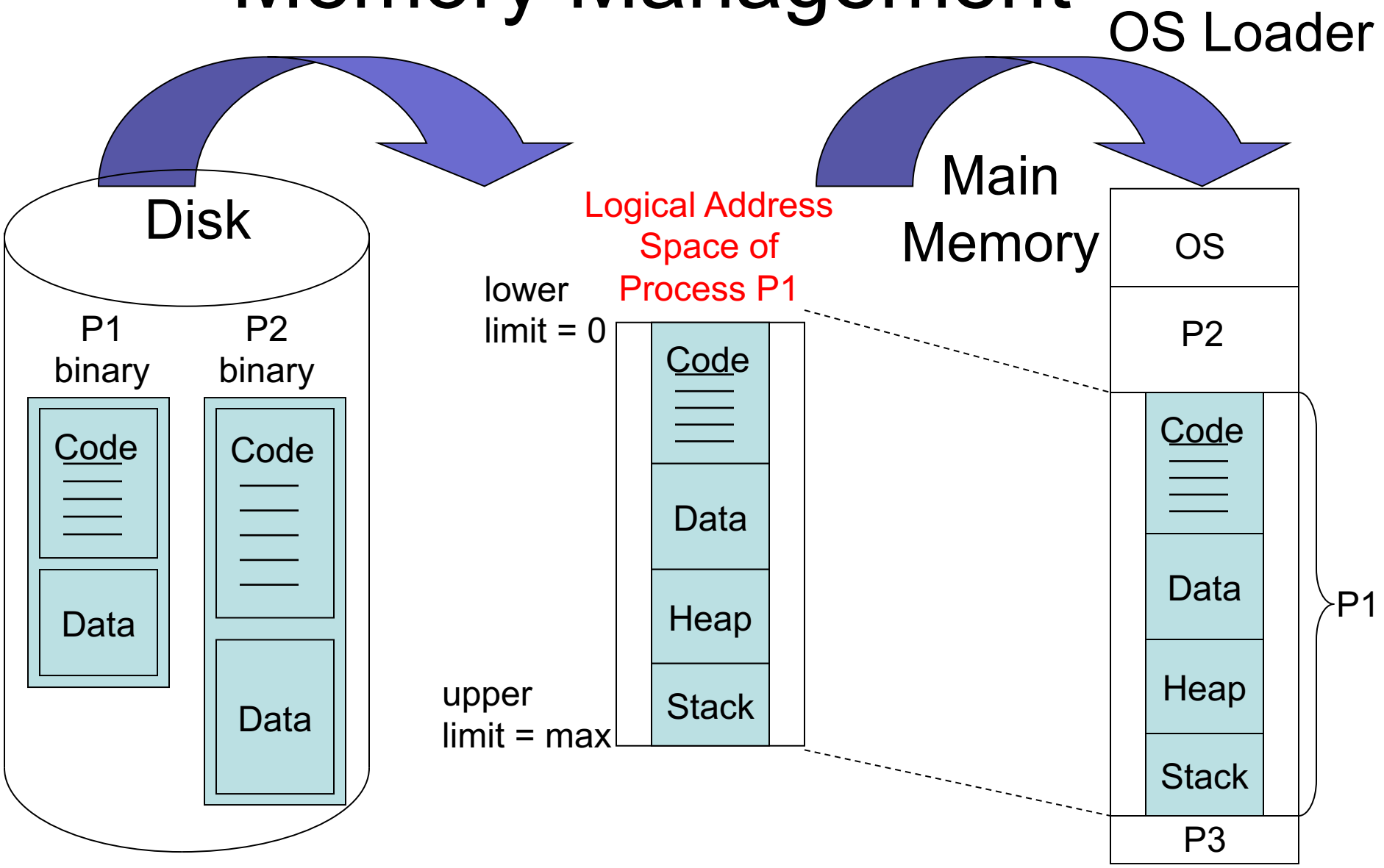
For L1 hardware caches, access must be fast, so organize as follows:



Cache size:
S x E x B data bytes

Cache organized this way for fast access

Memory 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.

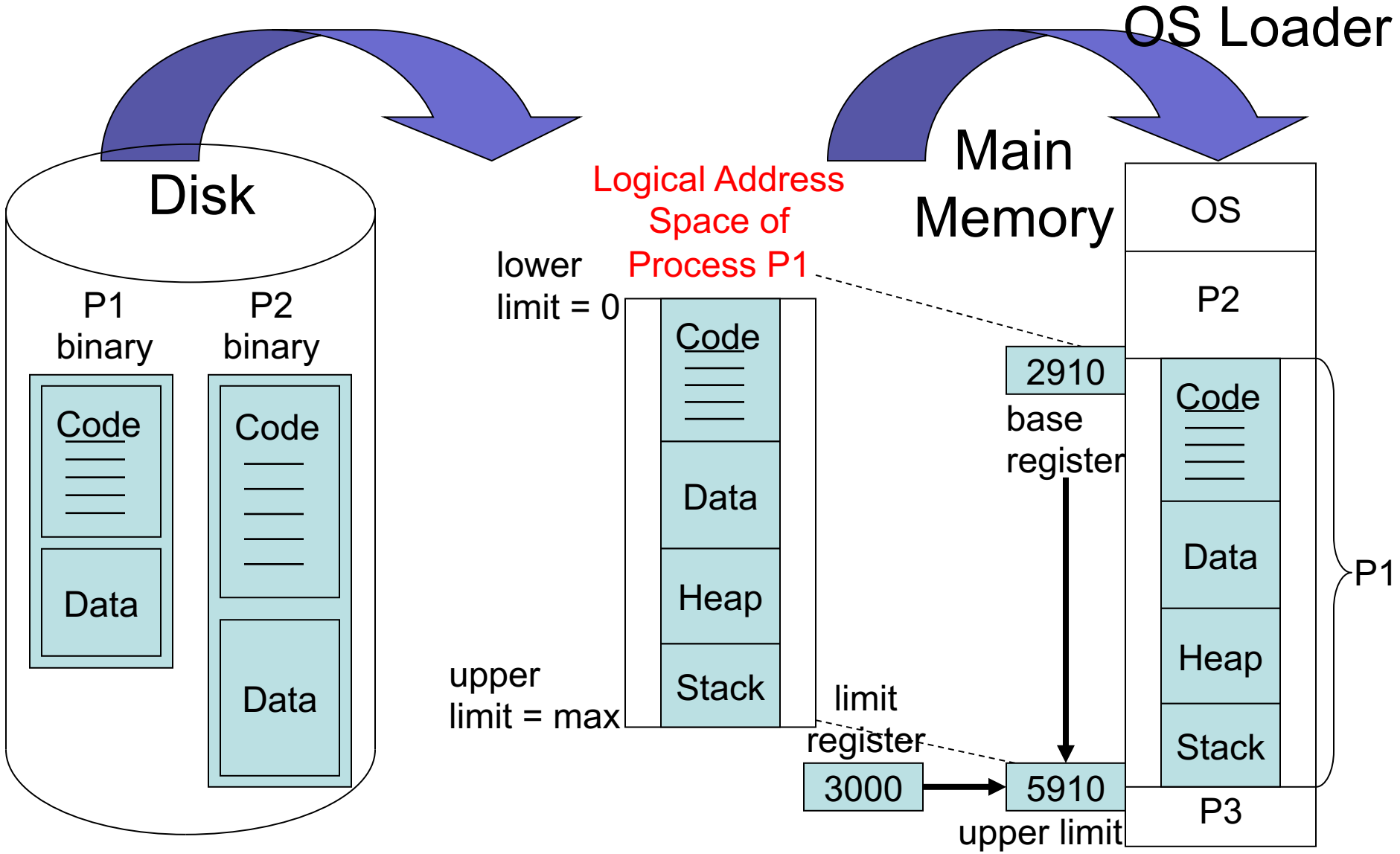
Memory Management

- 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

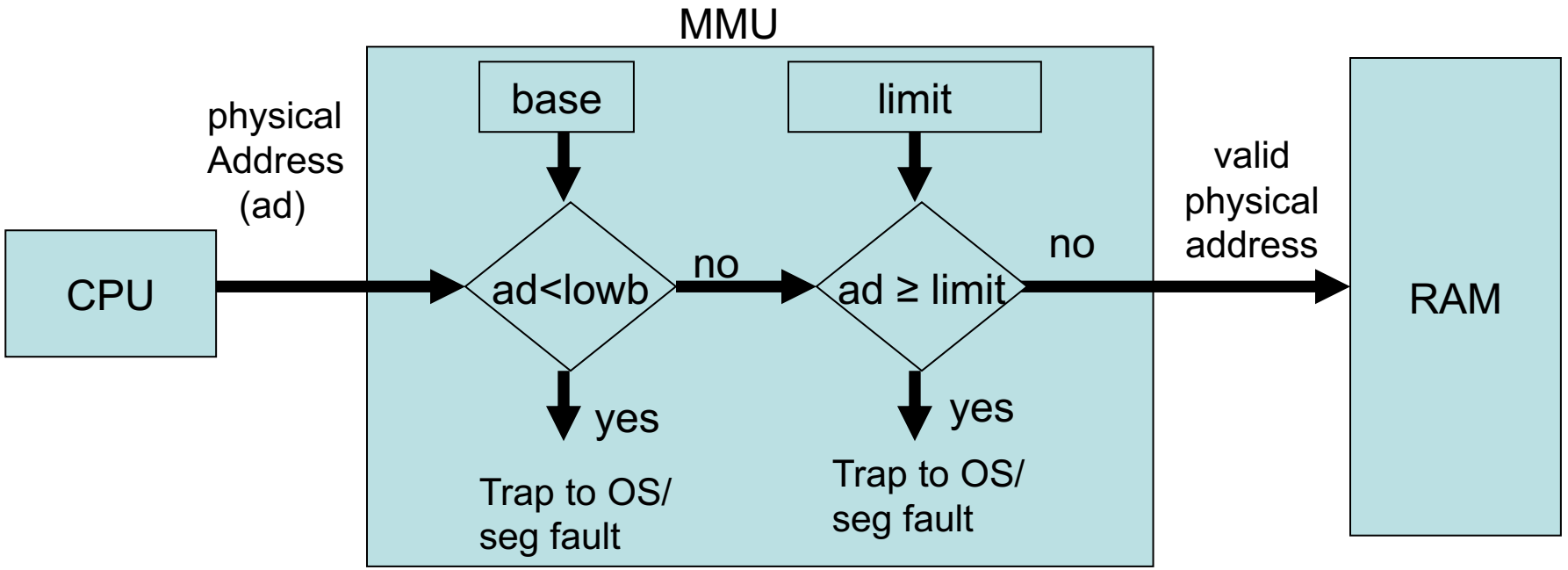


Memory Management

- base and limit registers provide hardware support for a simple MMU (Memory Management Unit)
  - 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

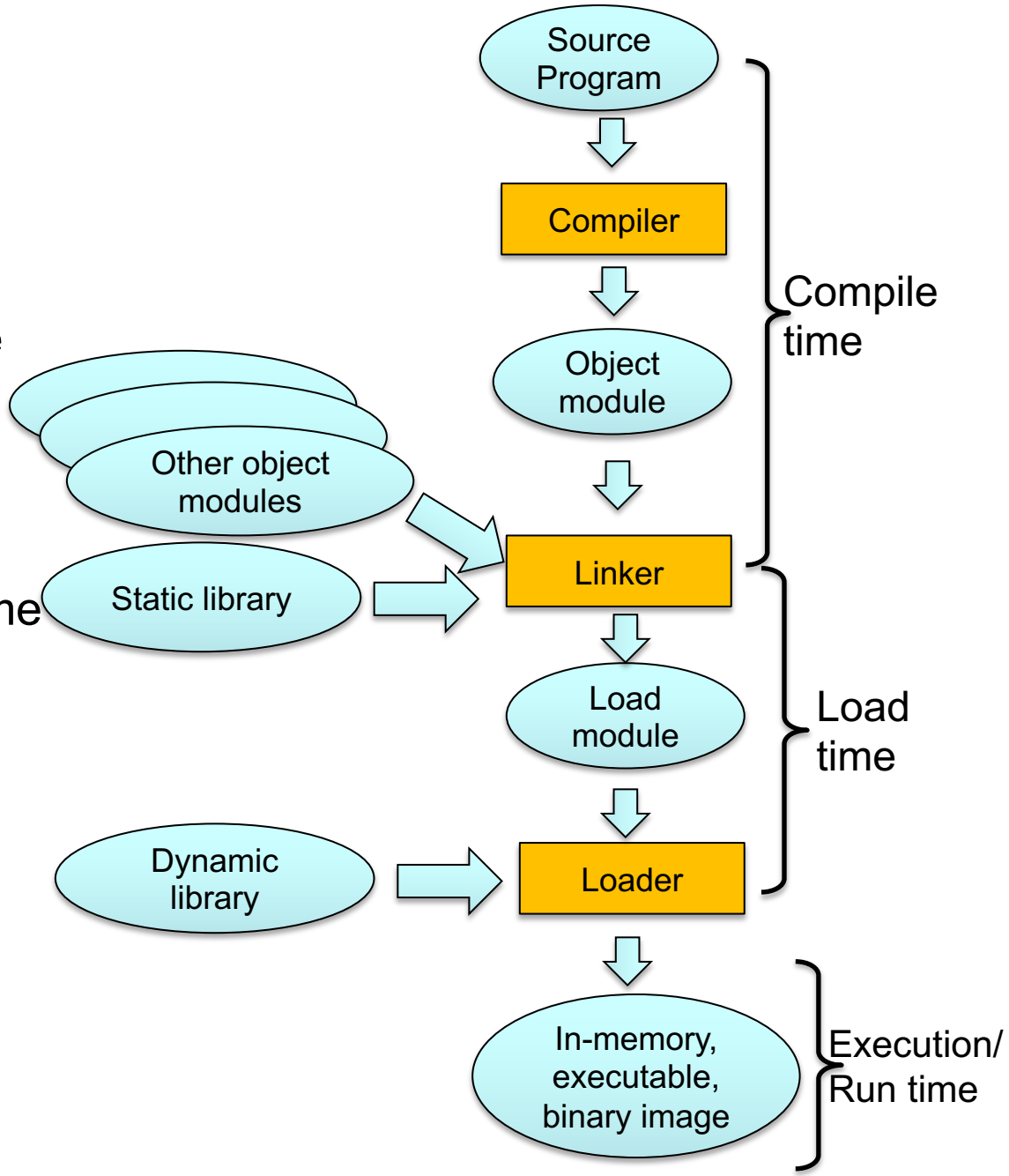
- MMU needs to check if physical memory access is out of bounds



Loading Tasks into Memory

- Using contiguous memory to hold task
- Task is stored as Code and Data on disk
- Task contains Code, Data, Stack, and Heap in memory
- Multiple tasks can be loaded in memory at the same time
- Kernel is responsible for protecting task memory from other tasks
  - Memory Management Unit(MMU) is used to validate all addresses.
  - Each task needs a base and limit register
  - Context switch must save/restore these address registers

- How does it know the addresses we want to access?
  - Binding at compile time
  - Binding at load time
  - Binding at execution time



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.
  - **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).

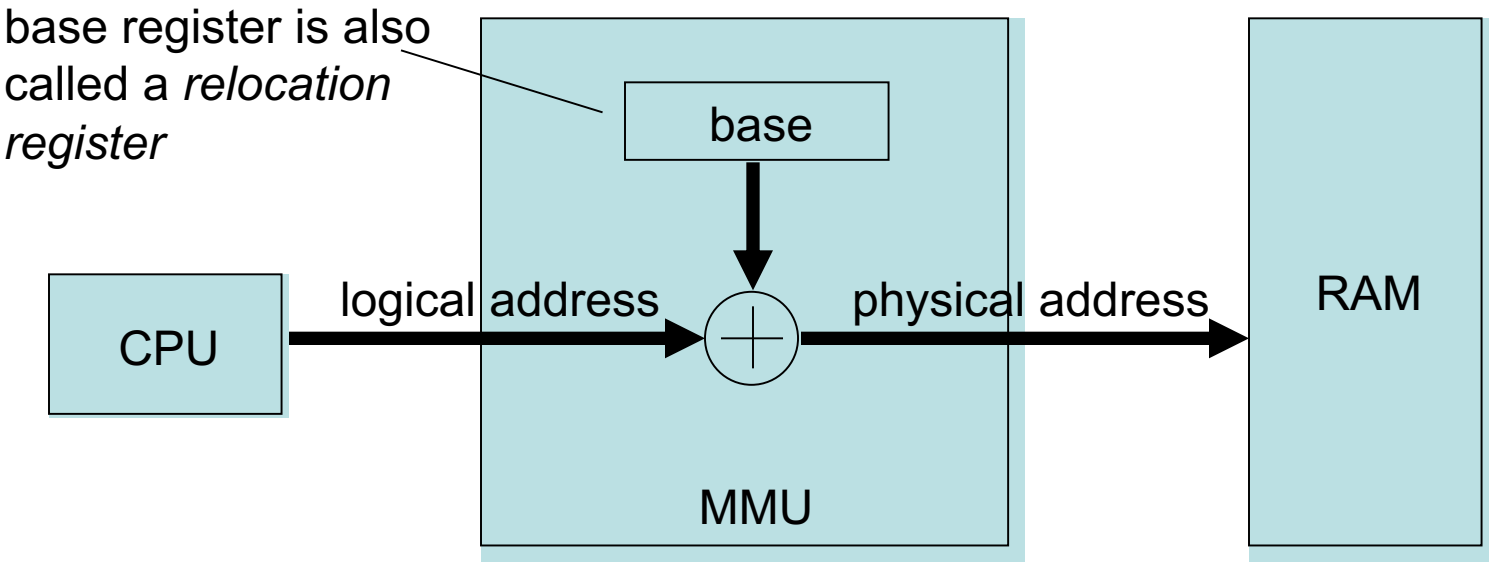
  - Once the binary has been thus changed, it is not really portable to any other area of memory, hence load time bound processes are not suitable for swapping (see later slides)

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

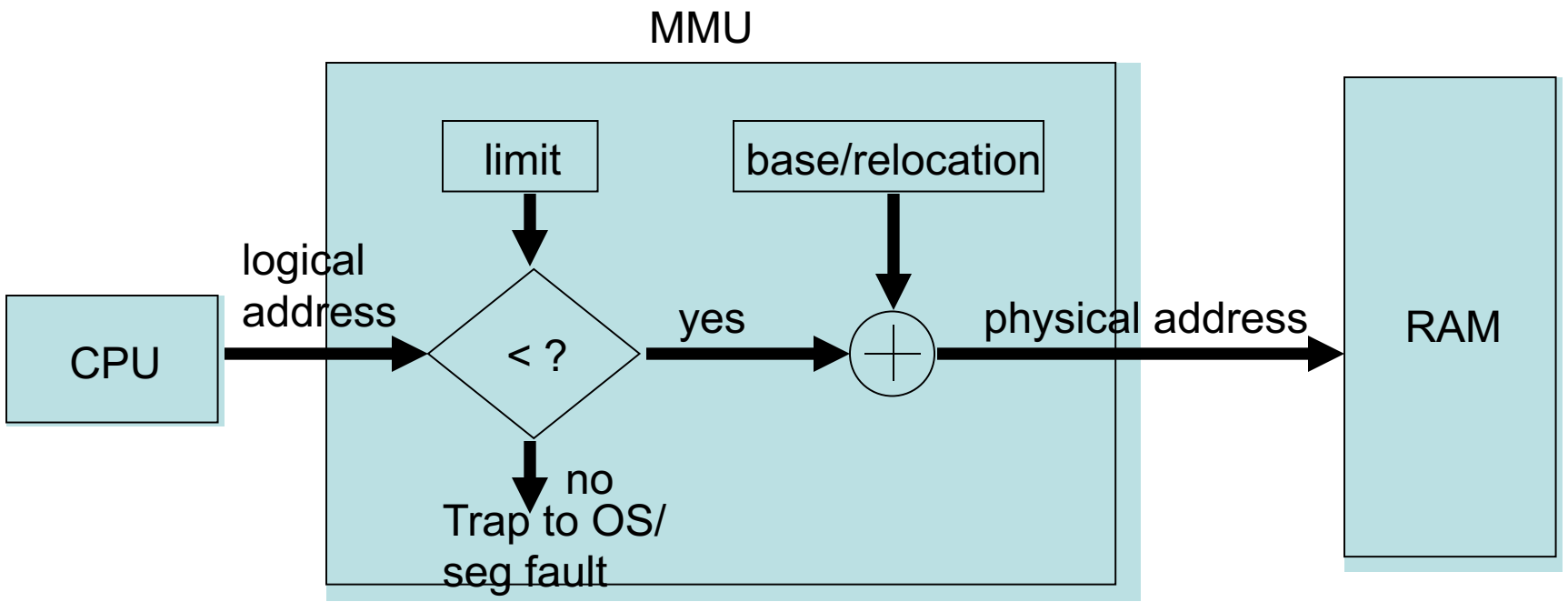
    MEM_ADDR_X is compiled as a logical address, and implicitly the **hardware MMU** will translate it to base+MEM_ADDR_X when the instruction executes

- MMU needs to perform run-time *mapping* of logical/virtual addresses to physical addresses
  - For run-time address binding,
    - 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



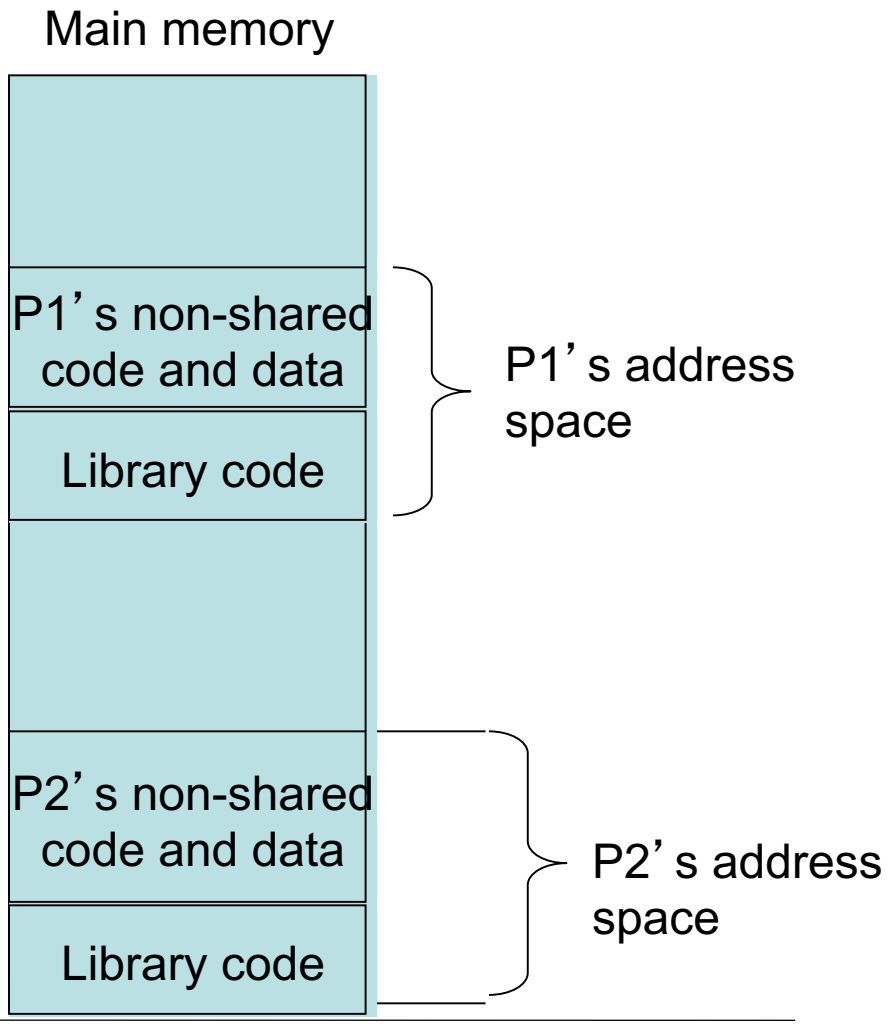
Memory Management

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



Run Time Binding with Static Linking

- Advantages of static linking:
  - Applications have access to the same library code even if it has changed recently
  - Can move to machines without the library
  - Self contained processes

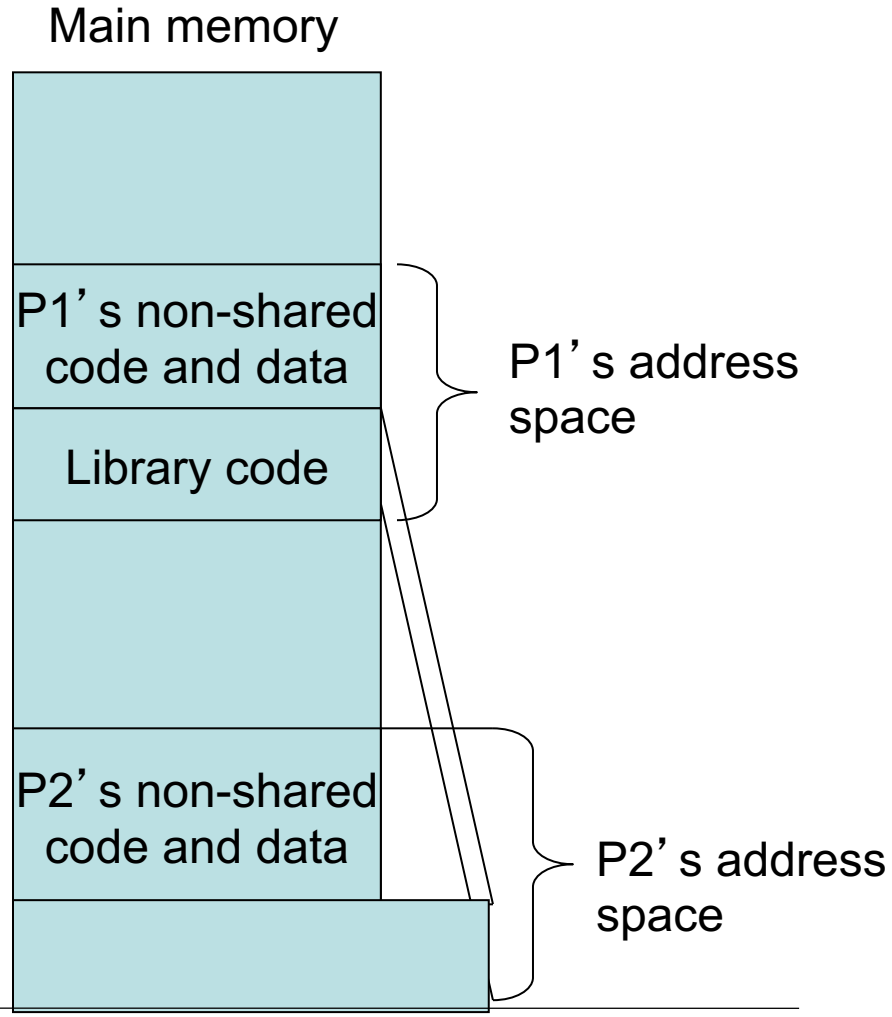


Run Time Binding

- 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

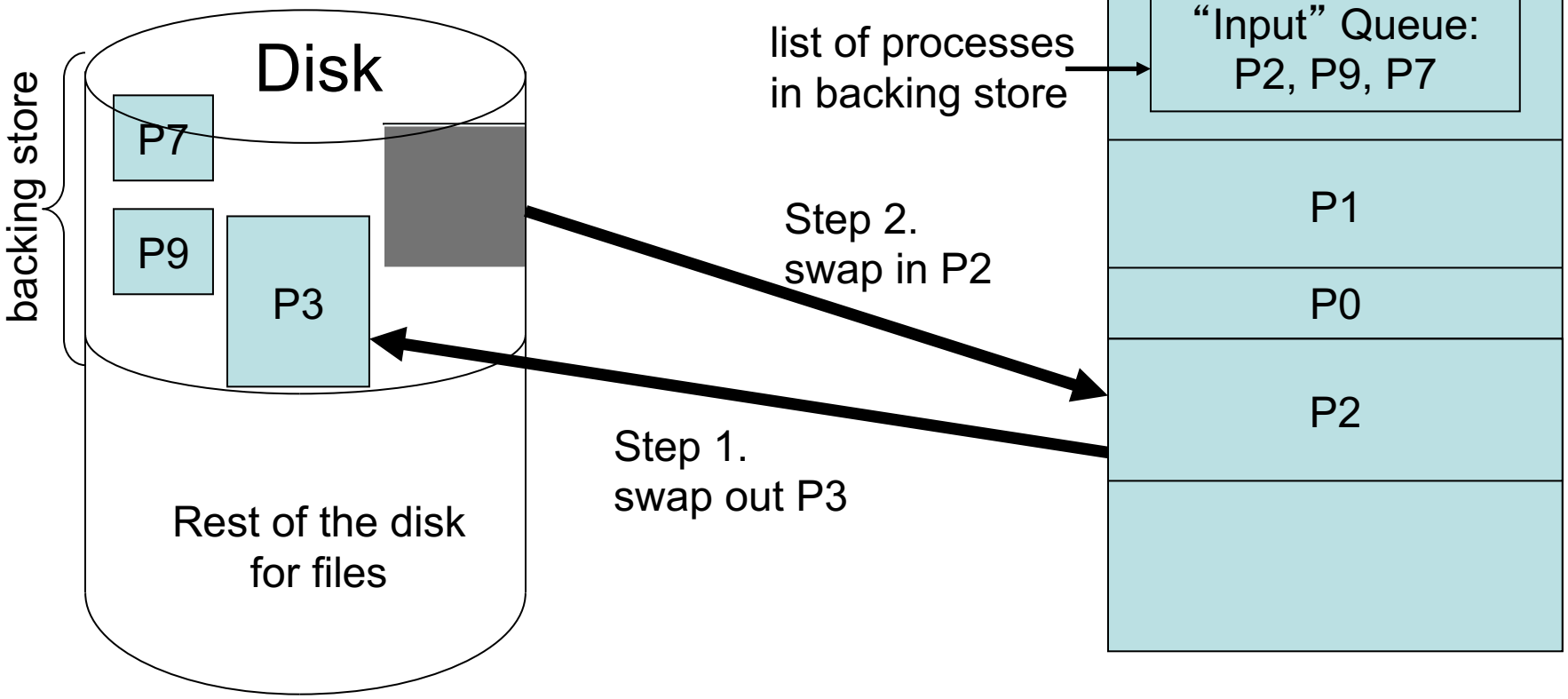
- Advantages of dynamic linking:
  - Applications have access to the latest code at run-time, e.g. most recent patched dlls,
  - Smaller size – stubs stay stubs unless activated
  - Can have only one copy of the code that is shared among all applications
    - We'll see later how code is shared between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



Swapping, Fragmentation, and Segmentation

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, to make room



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 per access
  - 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
  - avoids swapping unless the memory usage exceeds a threshold

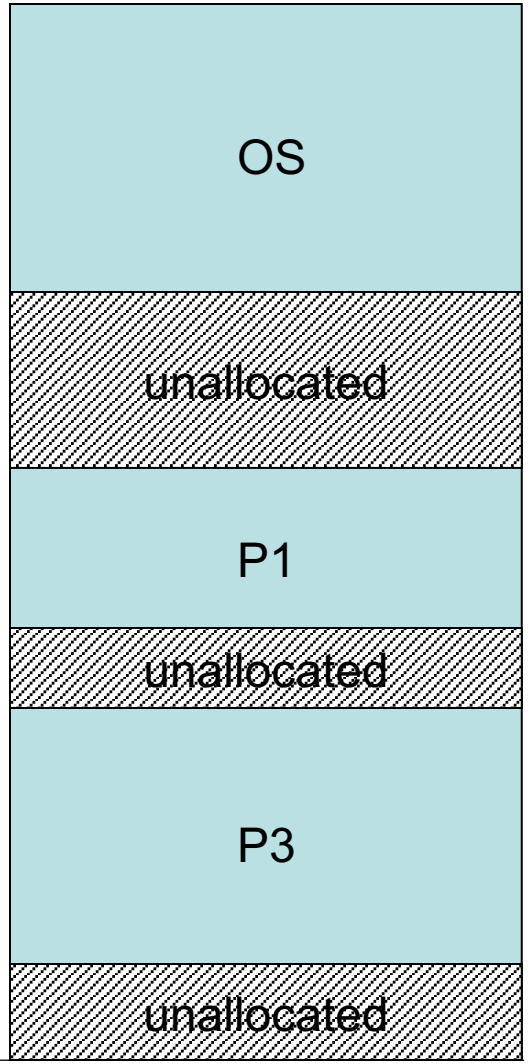
Swapping Difficulties

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

Memory Allocation

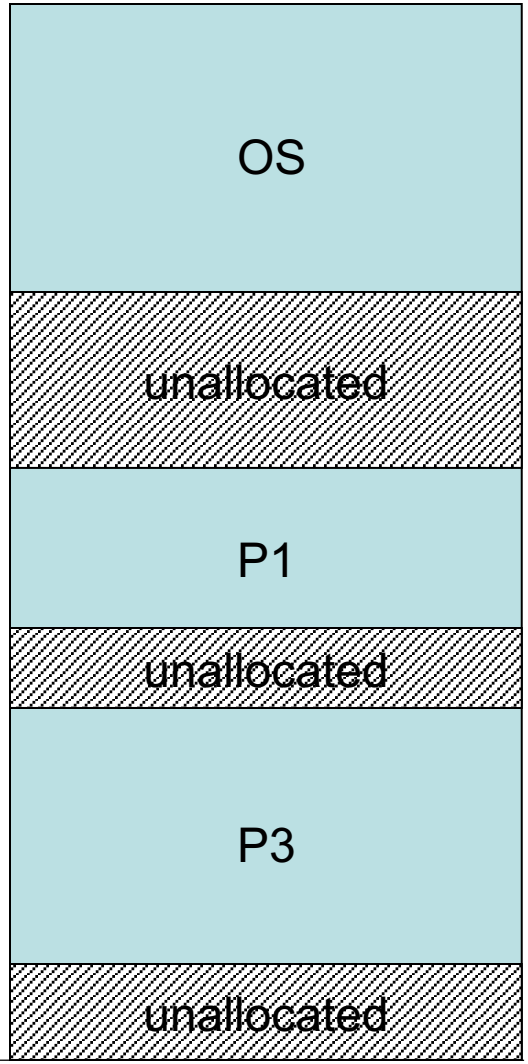
as processes arrive, they're allocated a space in main memory

- 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



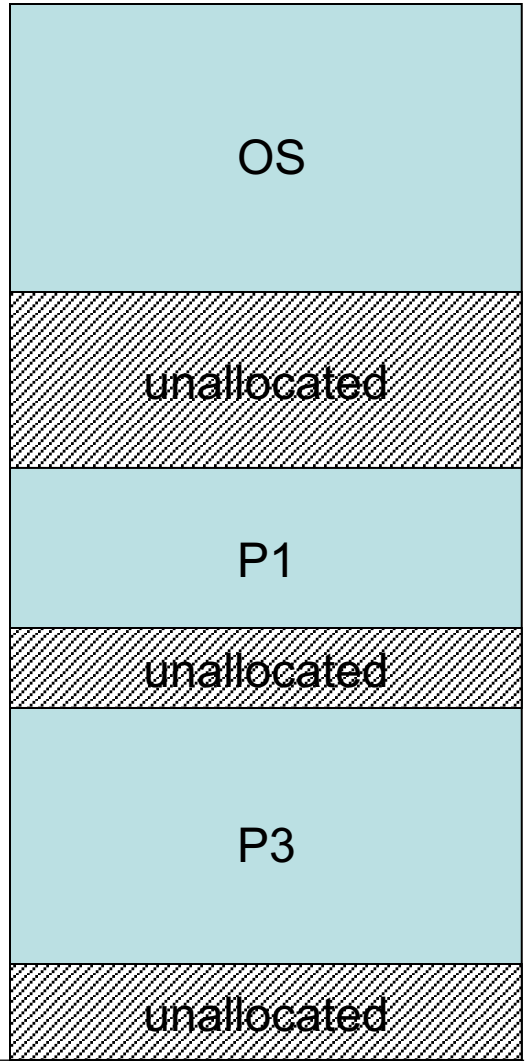
Memory Allocation

- over time, processes leave, and memory is deallocated
- results in *fragmentation* of main memory
  - There are many small chunks of non-contiguous unallocated memory between allocated processes in memory
- for the next process,
  - OS must find a large enough unallocated chunk in fragmented memory
  - May have enough memory, but not contiguous to allow a process to load

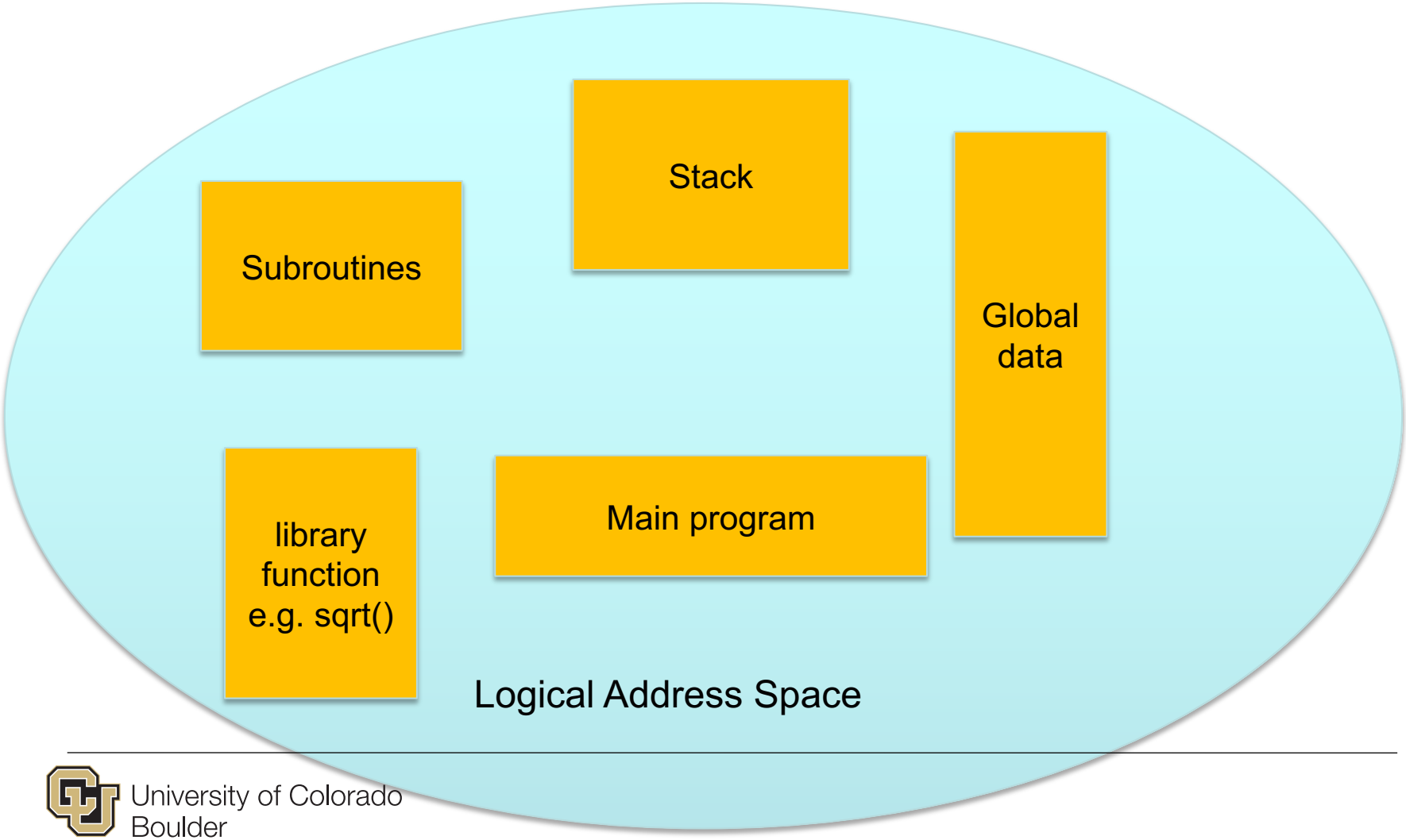


Fragmentation Problem

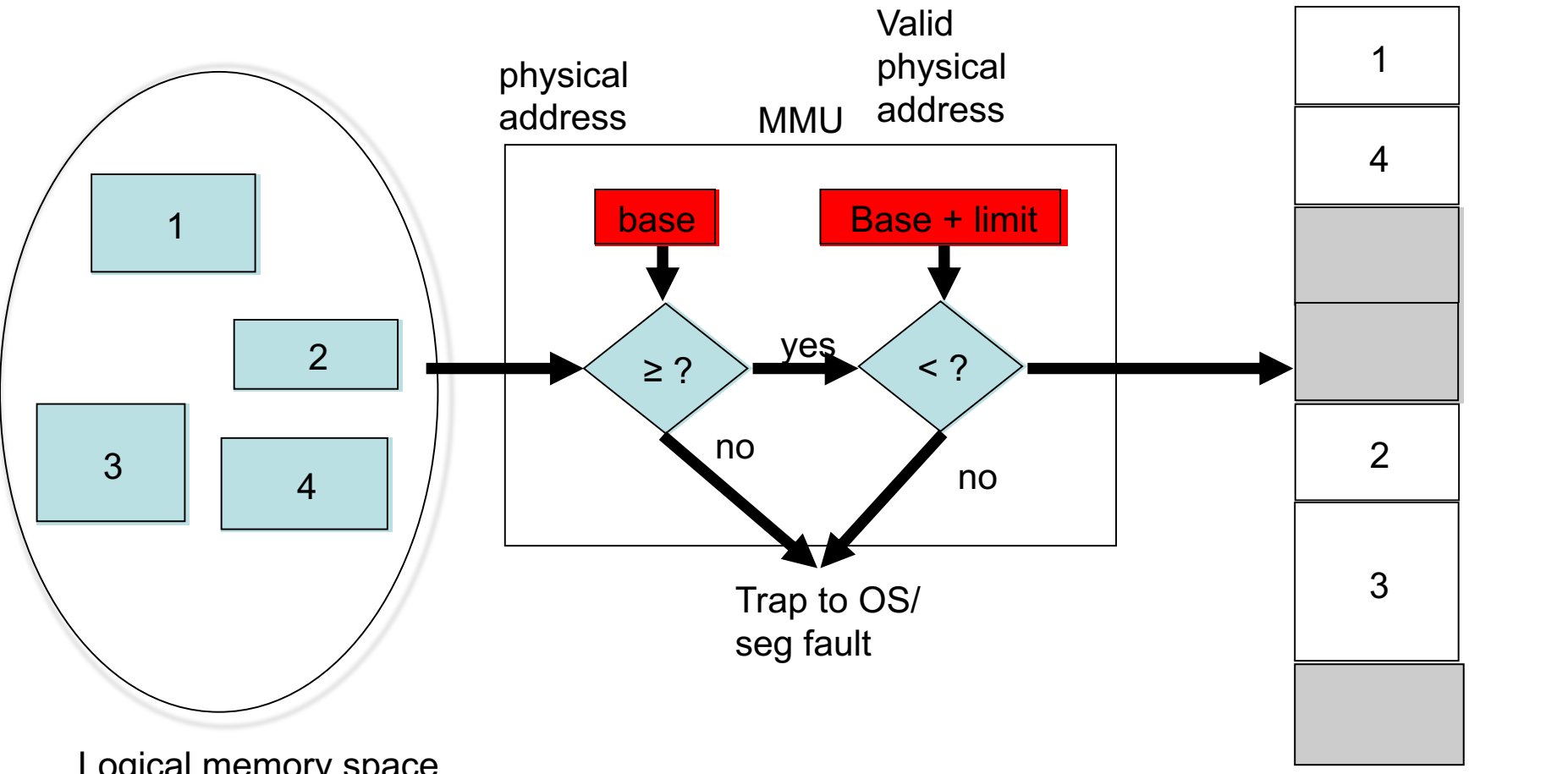
- This results in ***external fragmentation*** of main memory
  - There are many small chunks of non-contiguous unallocated memory between allocated processes in memory
- OS must find a large enough unallocated chunk in fragmented memory that a process will fit into
- De-fragmentation/Compaction
  - Algorithms for recombining contiguous segments is used, but memory still gets fragmented
  - Moving memory is expensive



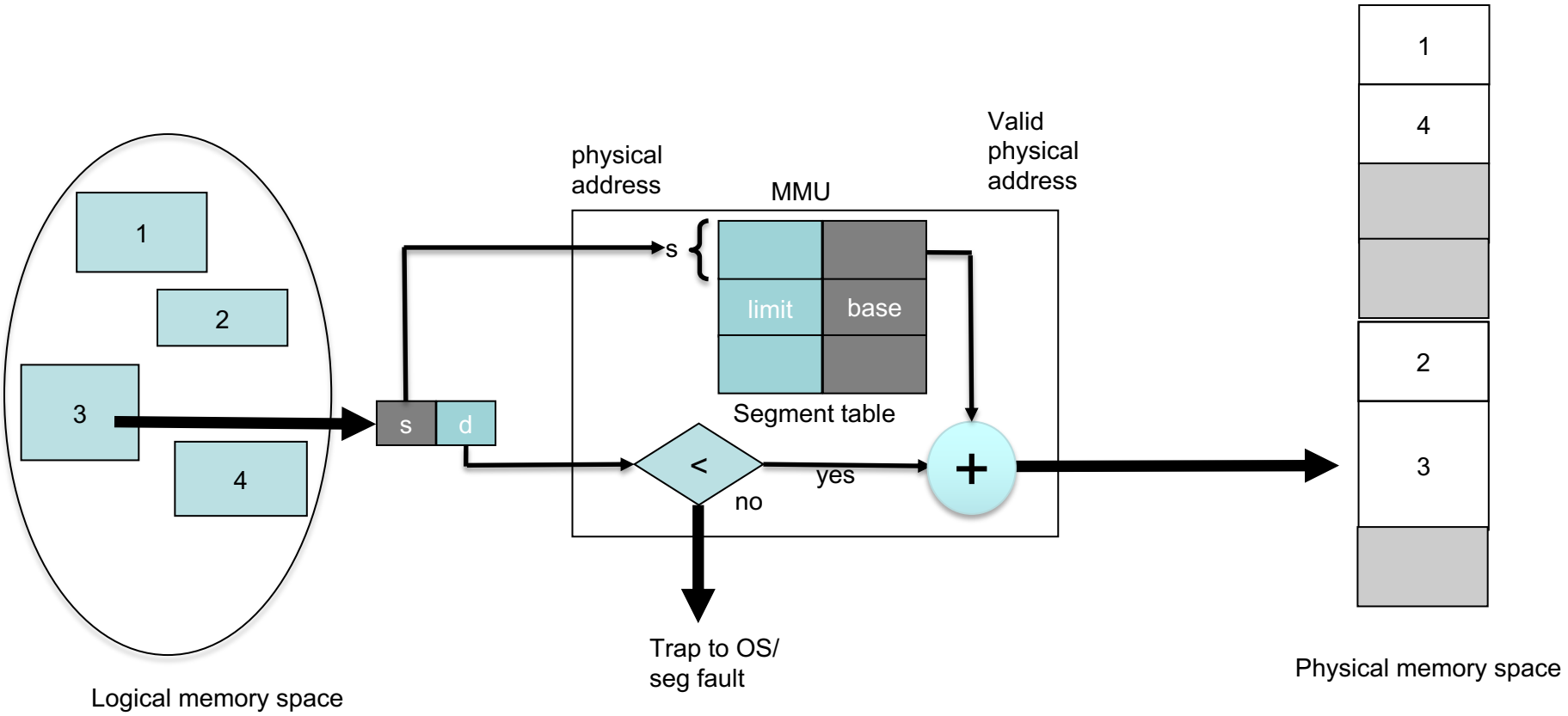
Instead of one big address space break it up into smaller segments



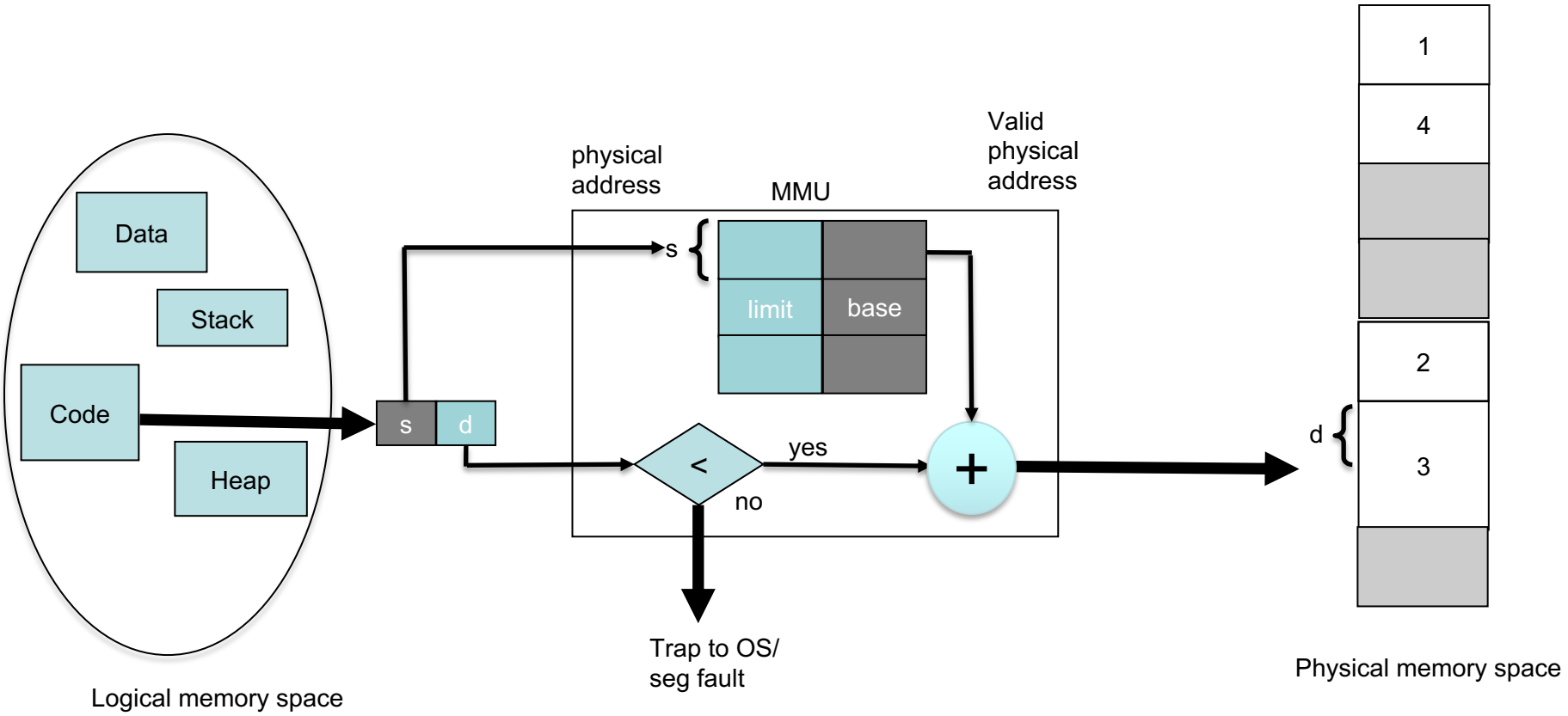
Segmentation



Segmentation of Memory

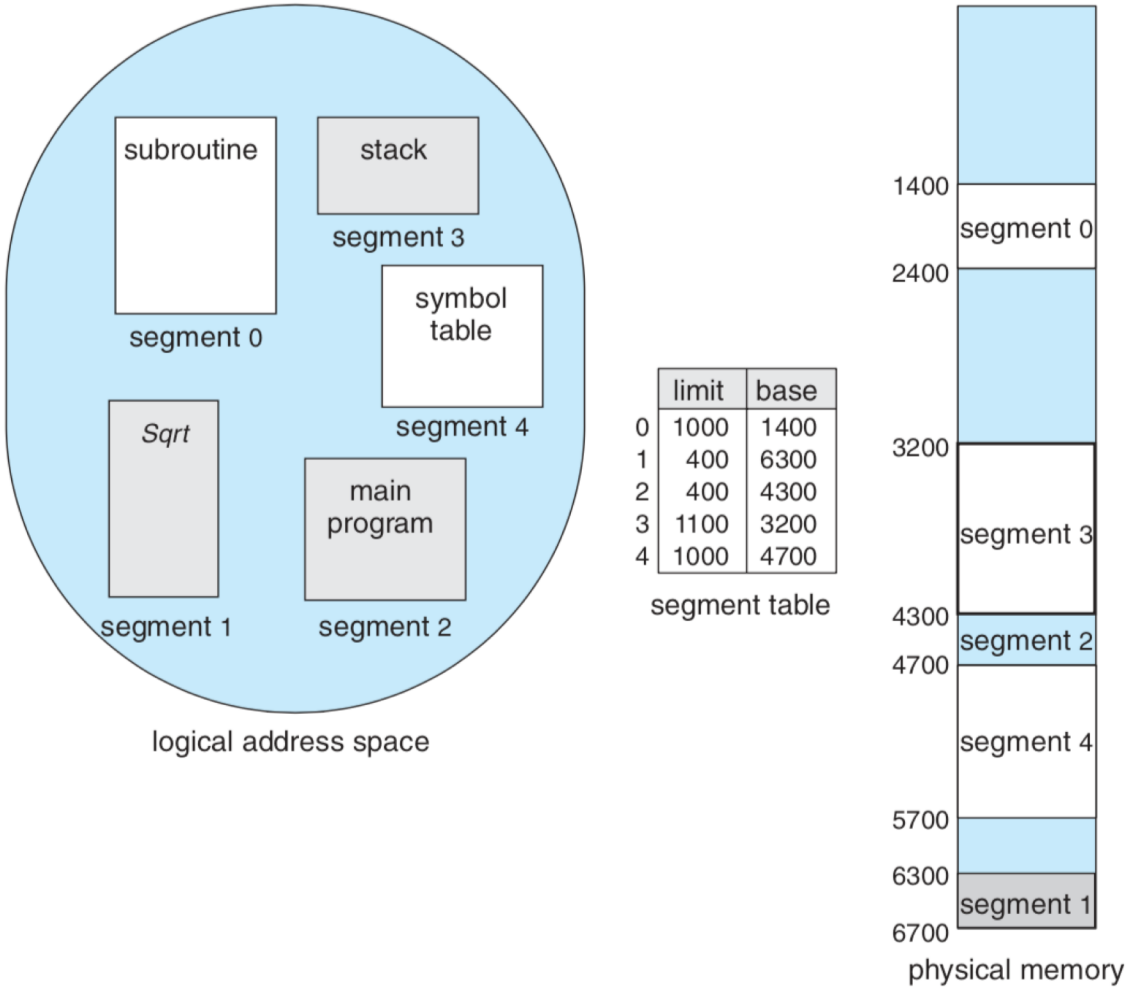


Segmentation of Memory



Segmentation allows physical address of a process to be non-contiguous

Example


subroutine
stack
segment 3
segment 0
symbol table
segment 4
Sqrt
main program
segment 1
segment 2
logical address space
limit
base
0 1000 1400
1 400 6300
2 400 4300
3 1100 3200
4 1000 4700
segment table
1400
segment 0
2400
3200
segment 3
4300
segment 2
4700
segment 4
5700
6300
segment 1
6700
physical memory