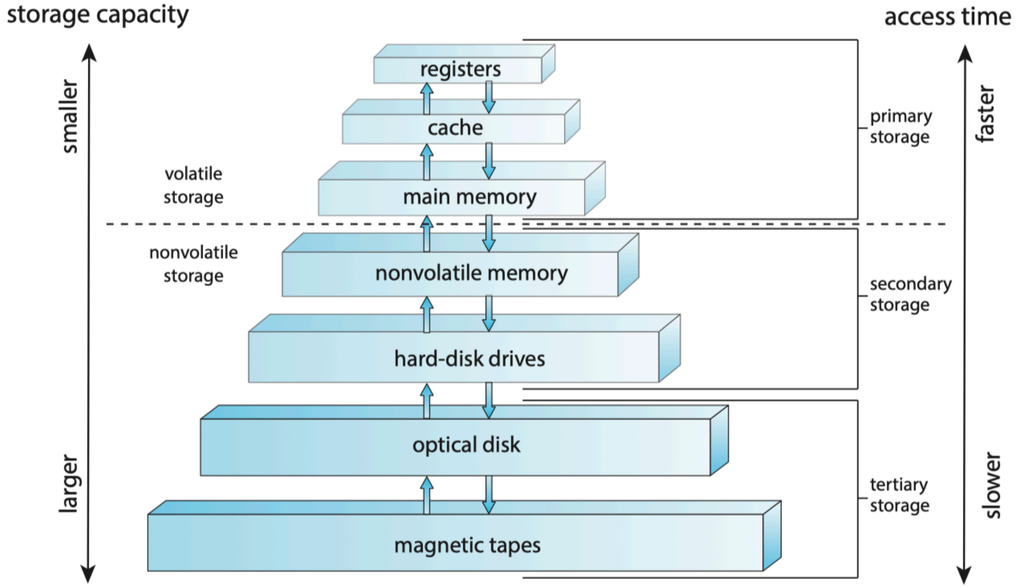
Memory Management

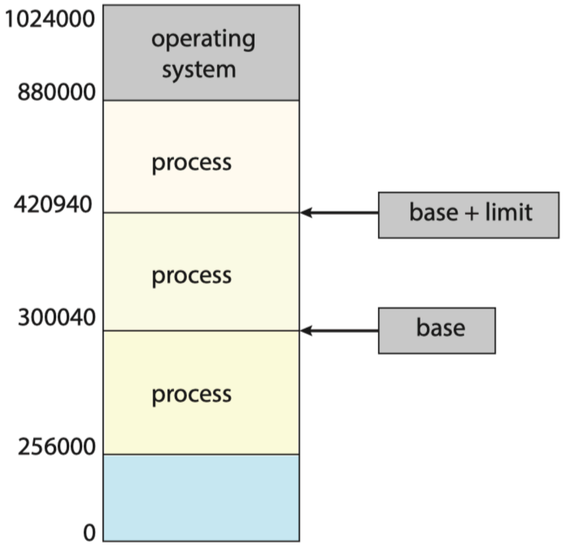
Memory Hierarchy

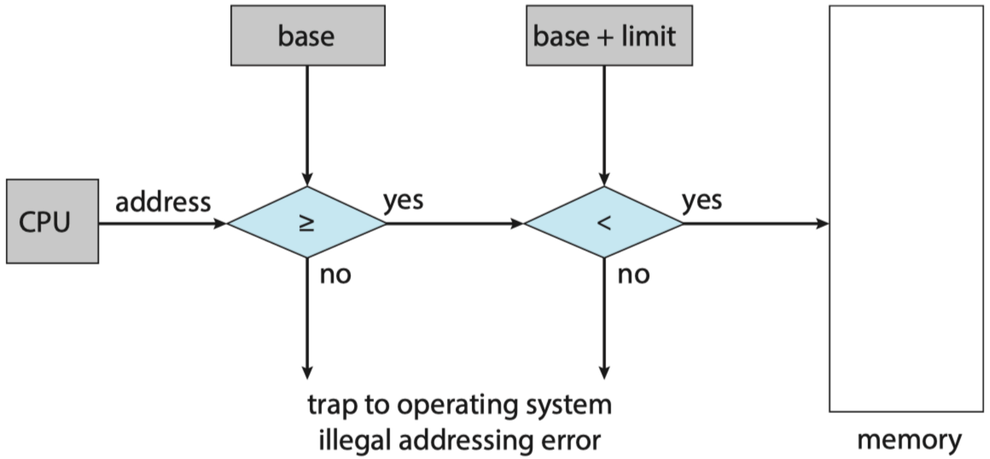
- Program must be in the directly addressable part of the memory before it can be executed
- Main memory + registers are the only memory units that can be directly addressed
- Memory units only see:
  - addresses + read requests (memory read)
  - addresses + data + write requests (memory write)
- Register access in one clock cycle
- Memory access in multiple clock cycles - stall the CPU operation



Memory Access Control

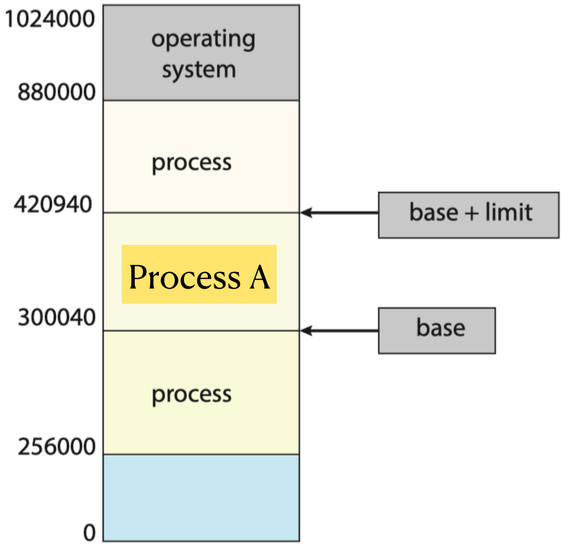
- We can have different programs loaded in memory - we want to ensure one program does not have a way of accessing other program or other programs data
- We need to partition the data and enforce partitioning without the program cooperating
- Idea: use base and limit registers, base registers tell the start of the memory segment and limit register specifies the size of the valid memory range
- CPU hardware compares every memory access is within the base and limits

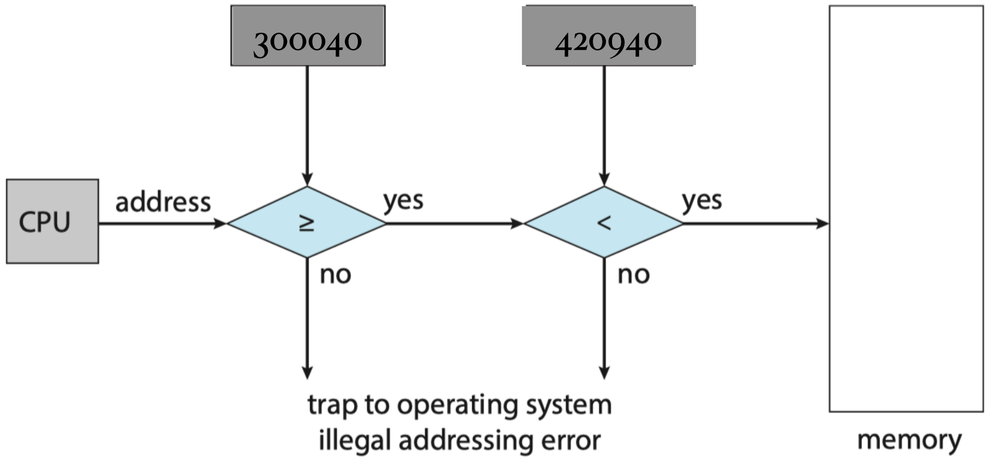




Example Memory Access

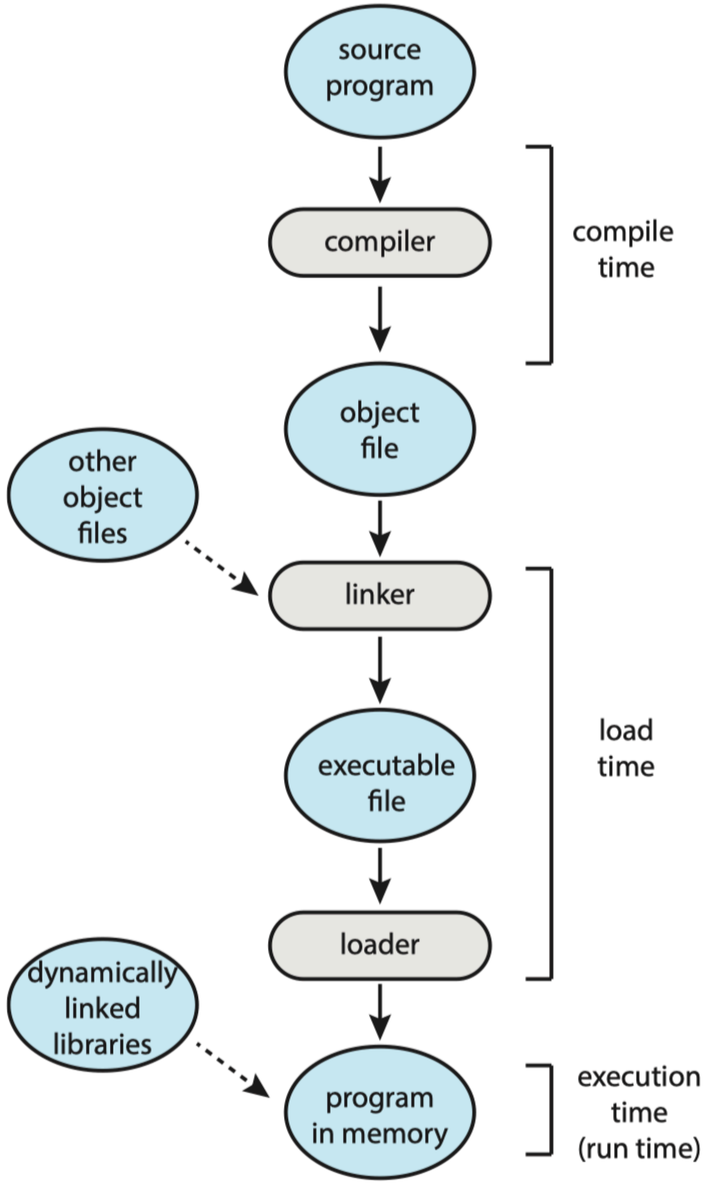
- Process A is generating
- **load #305000 rA**,
- The address is valid (Process A can access), it ends up loading the value at that address into register A
- **store rA, #300030**, the address is below the segment allocated for Process A, trapped into OS
- Memory access happens at the address generated by the program - check is done to validate the address
- Each **process has its own base and limit register** values!





Address Binding

- Program resides on disk as a binary image
- We need to load the program into memory and resolve all address references so they point to valid memory locations - that is, memory locations occupied by the program and not some other program
- So, we need to bind instructions to addresses: can be done in three ways
  - **Compile time** - we know where the program would reside in memory and it does not change - generate **absolute code**. Code for Read-Only Memory (ROM) is generated this way

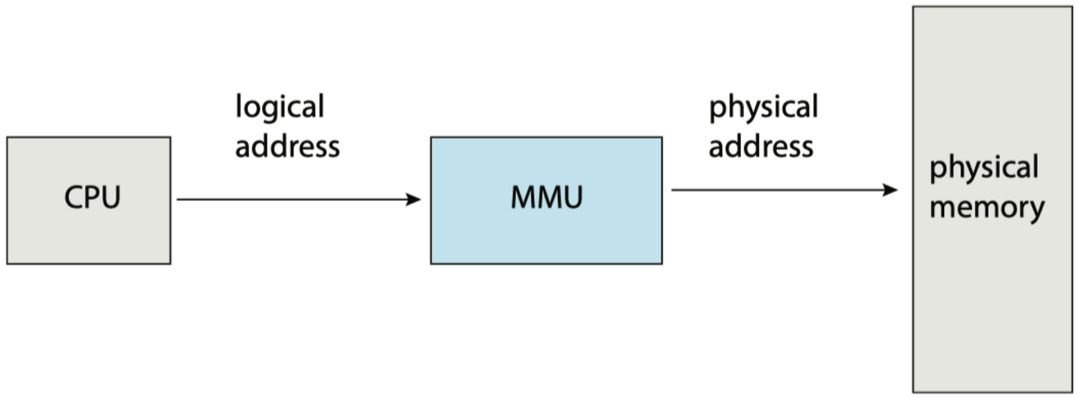


Address Binding

- **Load time** - At compile time, we don't know where the program would reside. Addressing needs to be relative so we can load the code segment to different memory locations - stays there after loading.
- **Execution time** - Want to change the memory locations of the code segment at run time. Need hardware support. Why? The code segment could be loaded and unloaded several times during execution, this is the reason for execution time binding.

Logical vs. Physical Address Space

- Address generated by the CPU is a **logical address**, the address seen by the memory is **physical address.**
- Logical address space — set of all logical addresses generated by a program
- Physical address space - set of all physical addresses corresponding to the addresses in the logical address space

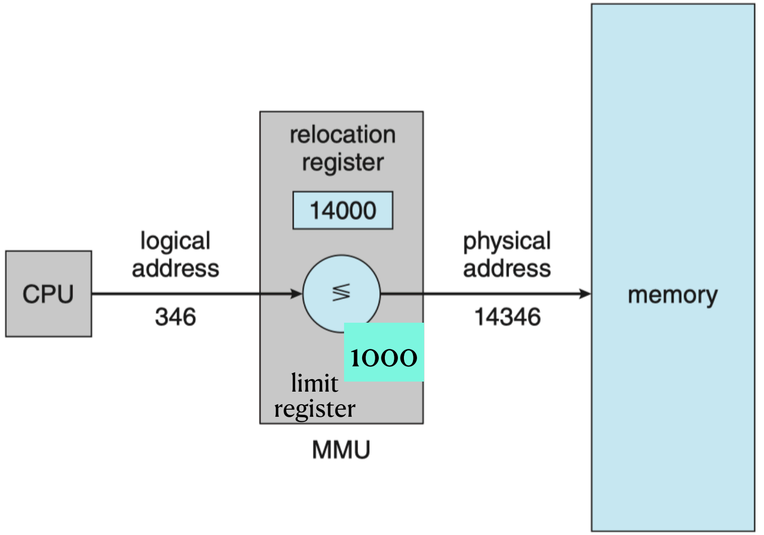


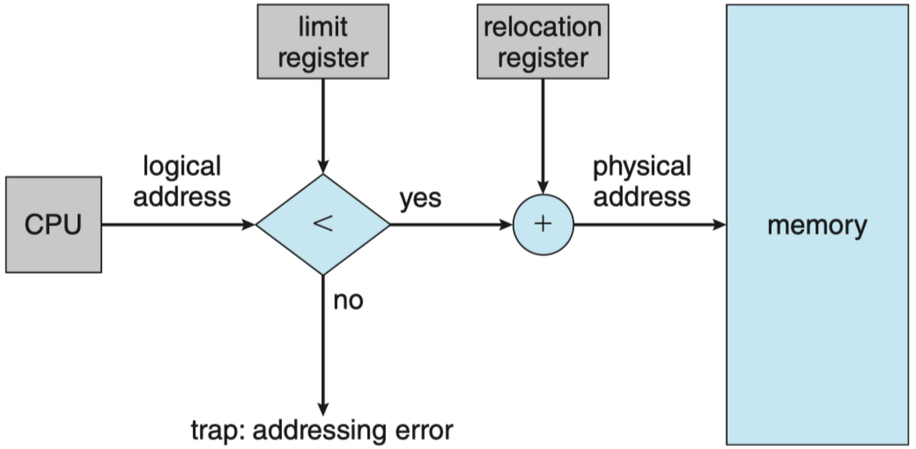
- Memory management unit (MMU) does the translation between the two address spaces
- Logical addresses and virtual addresses are used interchangeably
- Simple idea for MMU is based base register scheme, here it is called **relocation register**
- Relocation register is added to the value generated by the CPU
- Before adding to relocation register we validate the logical address

Relocation and Limit Registers

A barebones memory management unit

- Process A has 1000 bytes starting from 14000 allocated to it
- It can access 14000 to 14999 - anything outside is a memory access violation
- Program (running in the CPU) is generating logical addresses - in the range - 0 to 999
- MMU is translating to the allocated range
- First step: check against the limit (999) and validate that the access is within the range
- Second step: translate to the target range



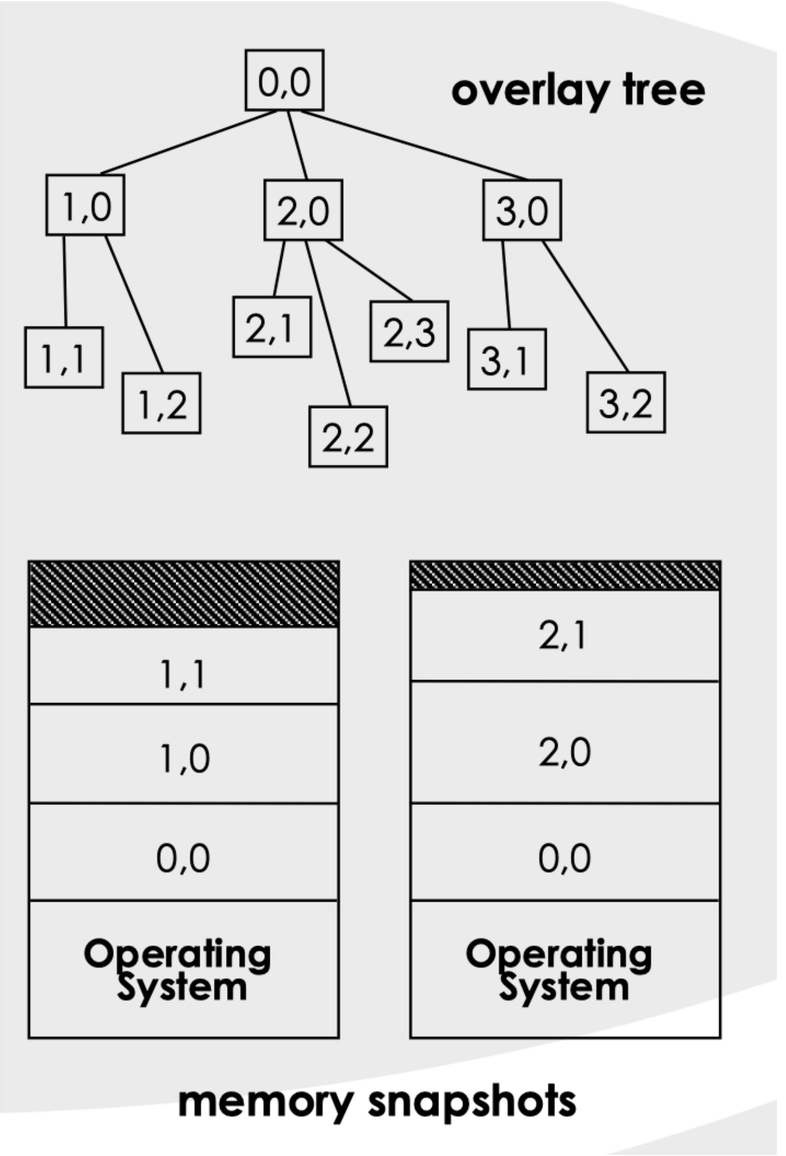


Dynamic Loading

- What is dynamic loading? It is is **opposite of static loading**, which is loading the whole program at the start. Problems: (1) wait for the whole program to load into memory, (2) waste memory by loading unused portions of the program for a given run
- We can defer loading as much as we can and load when it is needed
- Disadvantage: **loading can happen at runtime** and run times can be affected by loading, loaded segments can be **cached so future references** can be quick - only first reference suffers loading

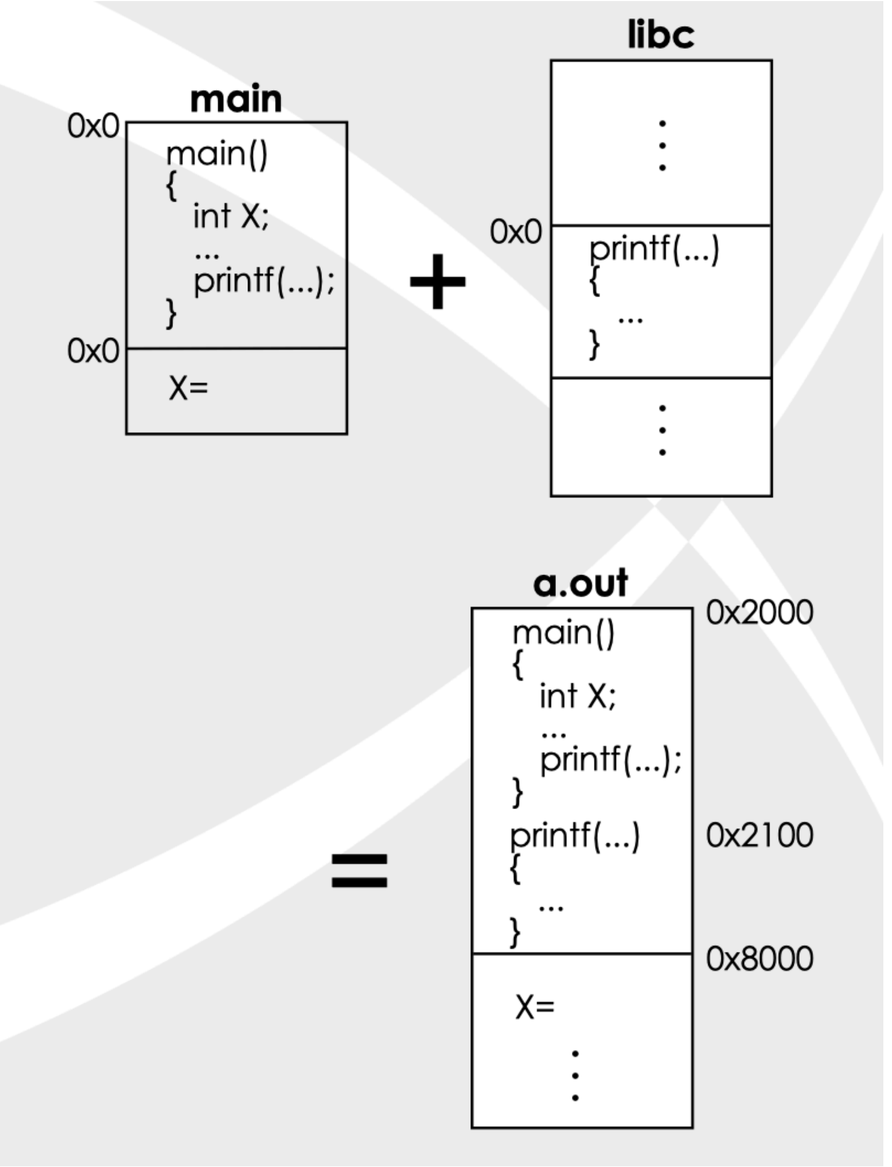
Example of Dynamic Loading

- One example of dynamic loading is overlays
- Program is split by user or compiler into overlays
- Different overlays are brought in as needed
- There will be an overlay manager loaded to facilitate the loading and unloading of overlays



Linking

- Programs need additional libraries to run
- Linking is a way stitch a valid executable image using user programs and external libraries
- Linker collects all pieces and forms the executable image
- Where to put the different pieces in the final image - **relocation**
- Where to find the pieces needed in the linking process - **resolving references**

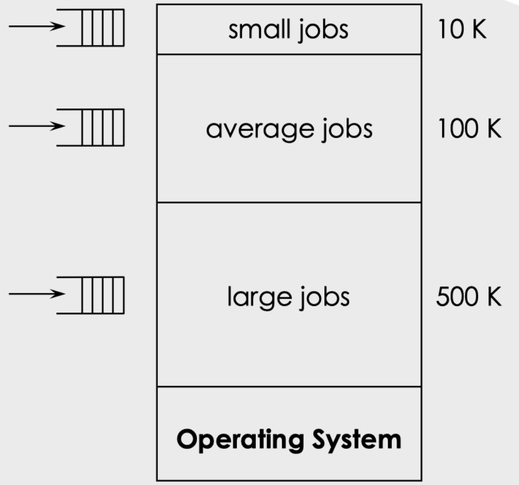


Memory Allocation

- Major challenge with memory management is memory allocation - **allocate memory for different programs that are co-residing in memory**
- Challenges
  - Need to allocate memory without **wasting memory space** - allocating too much or splitting the memory into chunks that cannot be allocated to any process
  - Prevent starvation of processes - processes waiting very long for their memory allocations
  - Protect memory access - prevent one process from accessing memory that is allocated for another process
  - Reacting to changing memory usage requirements of the processes - memory needs of a program changes depending on input data - hard to foresee

Memory Allocation Ideas

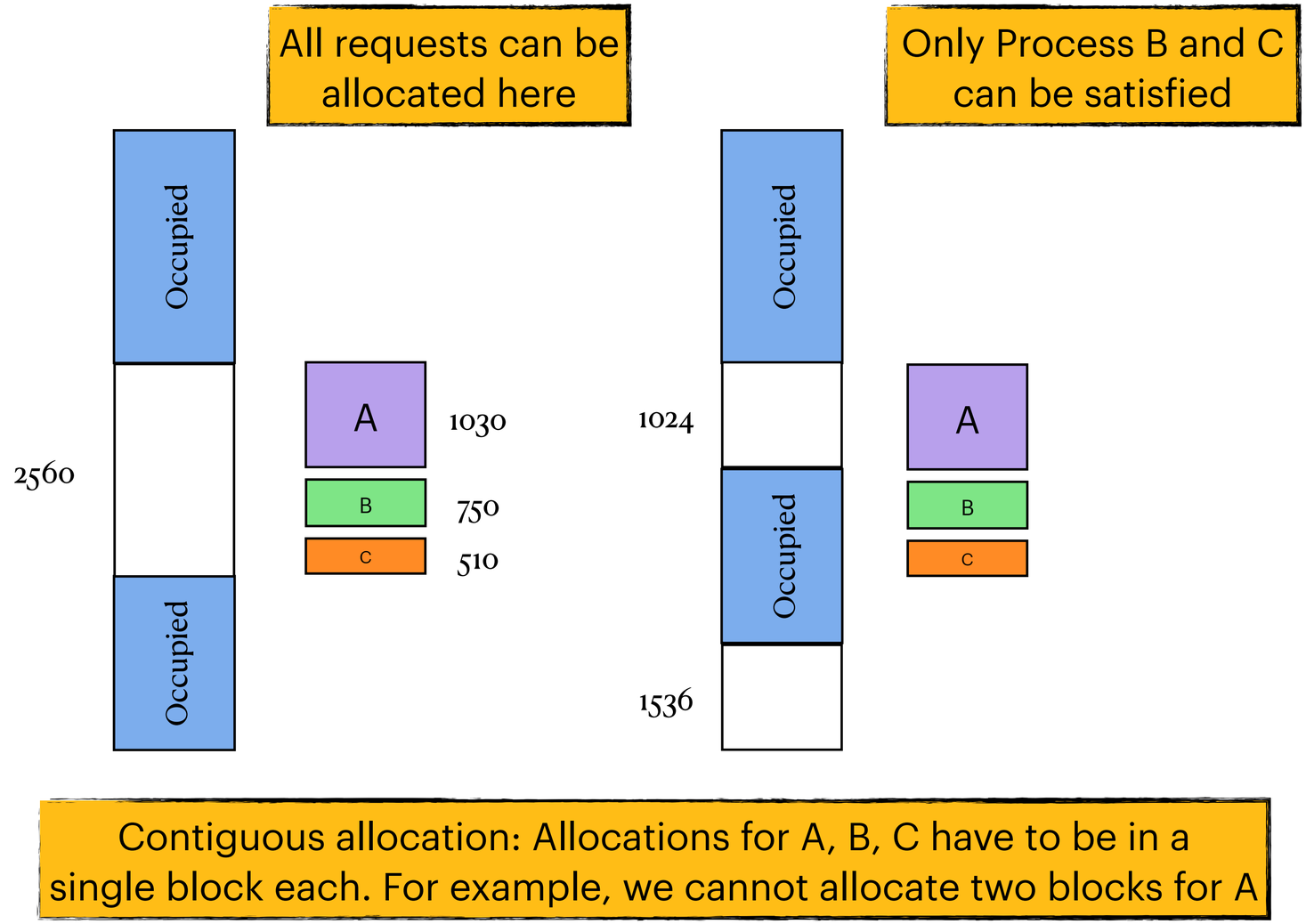
- Partitioning: Splitting the available memory into partitions - kernel partition and user partition
- Kernel partition used for kernel code segments, buffers, etc
- User partition used for all user programs
- We can also partition memory in other ways: depending on memory requirements - small, medium, large jobs



- Memory is divided into *fixed* number of (fixed size) partitions
- Programs are queued to run in the smallest available partition
- Program prepared to run in one partition may not be able to run in another, without being re-linked -- *absolute loading*

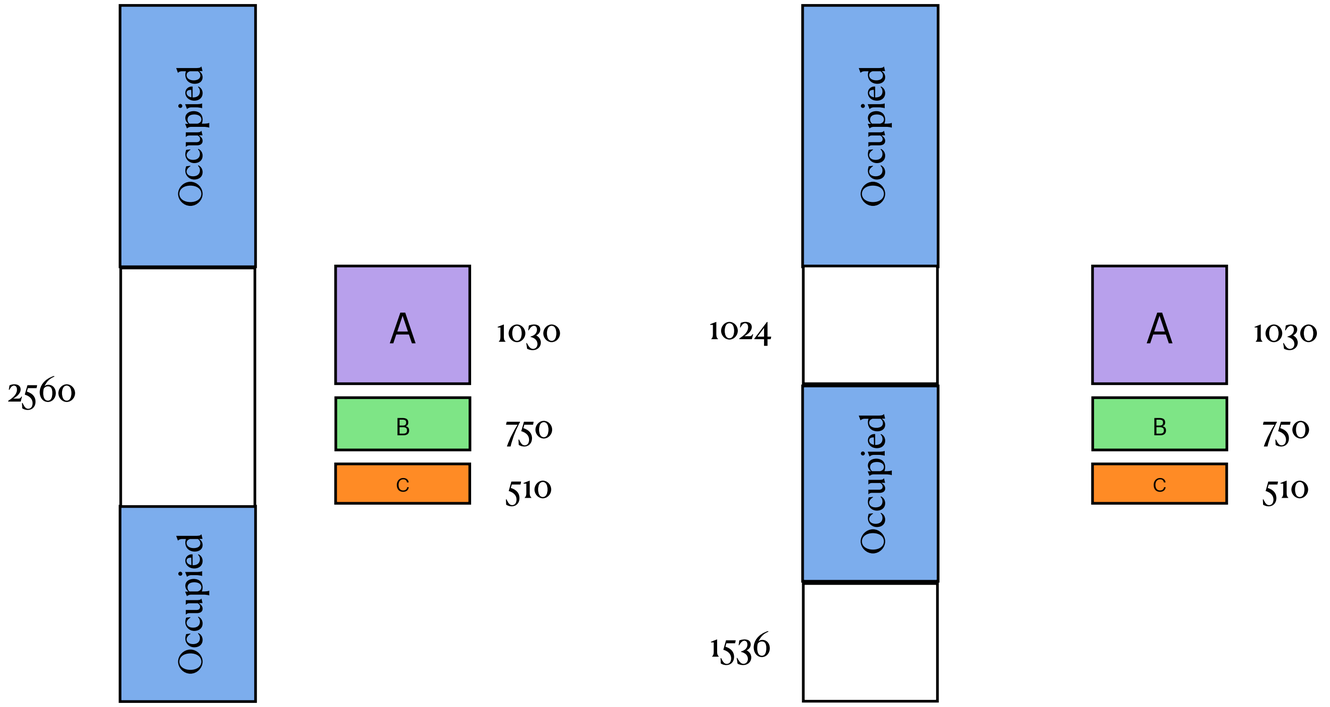
Contiguous Memory Allocation

- Process A wants 1030 bytes
- Process B wants 750 bytes
- Process C wants 510 bytes
- Lets say we have free space 2560 bytes in free space in total
- We can have the free space in different configurations: a single block of 2560 bytes or two different blocks 1024 and 1536 bytes that are not next to each other, or some other config



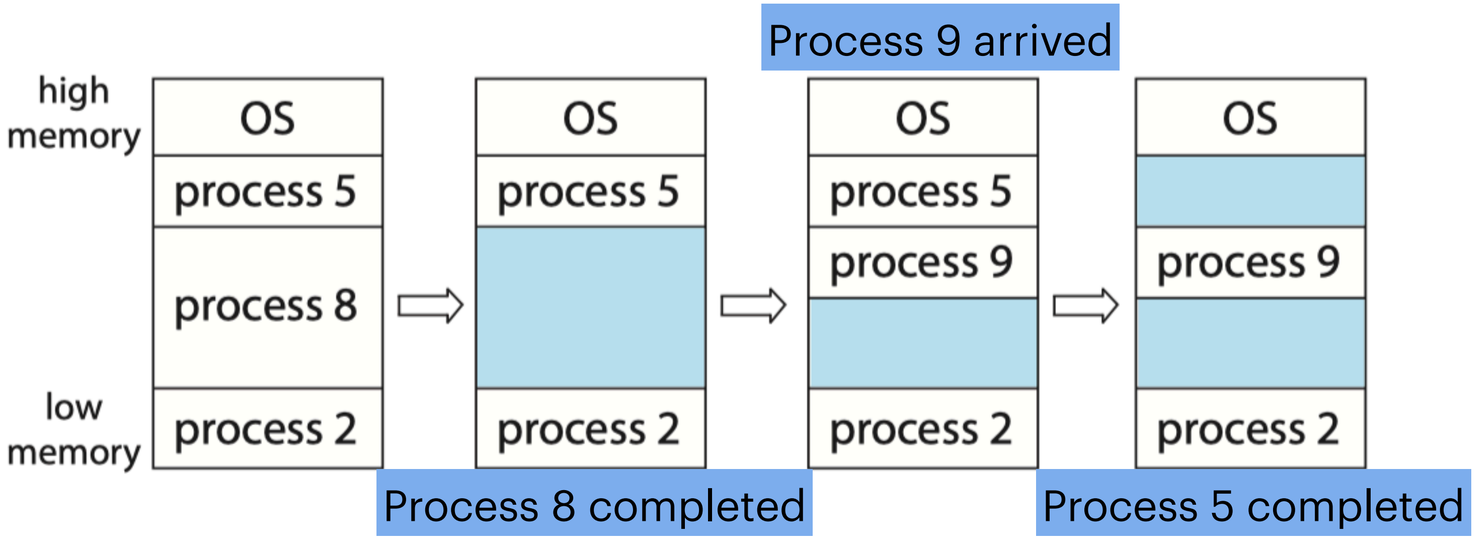
Contiguous Memory Allocation

- Process A wants 1030 bytes
- Process B wants 750 bytes
- Process C wants 510 bytes
- Lets say we have free space 2560 bytes in free space in total
- We can have the free space in different configurations: a single block of 2560 bytes or two different blocks 1024 and 1536 bytes that are not next to each other, or some other config

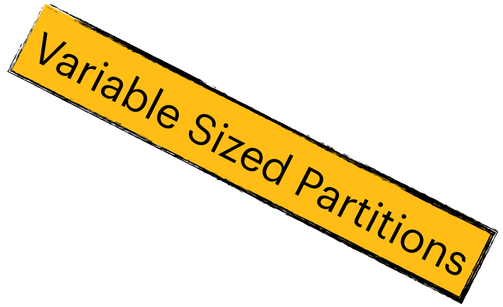


Contiguous Memory Allocations

- We start with a single block of free memory - whole available memory (this is everything that is not already used by OS)
- As processes are created, we allocate variable chunks of memory for each process
- Processes release their allocations as they complete
- We end up creating holes in memory — free spaces in memory that are surrounded by allocated spaces
- If the holes are small and cannot accommodate requests, then they are wasted (remain unallocated until reboot!)

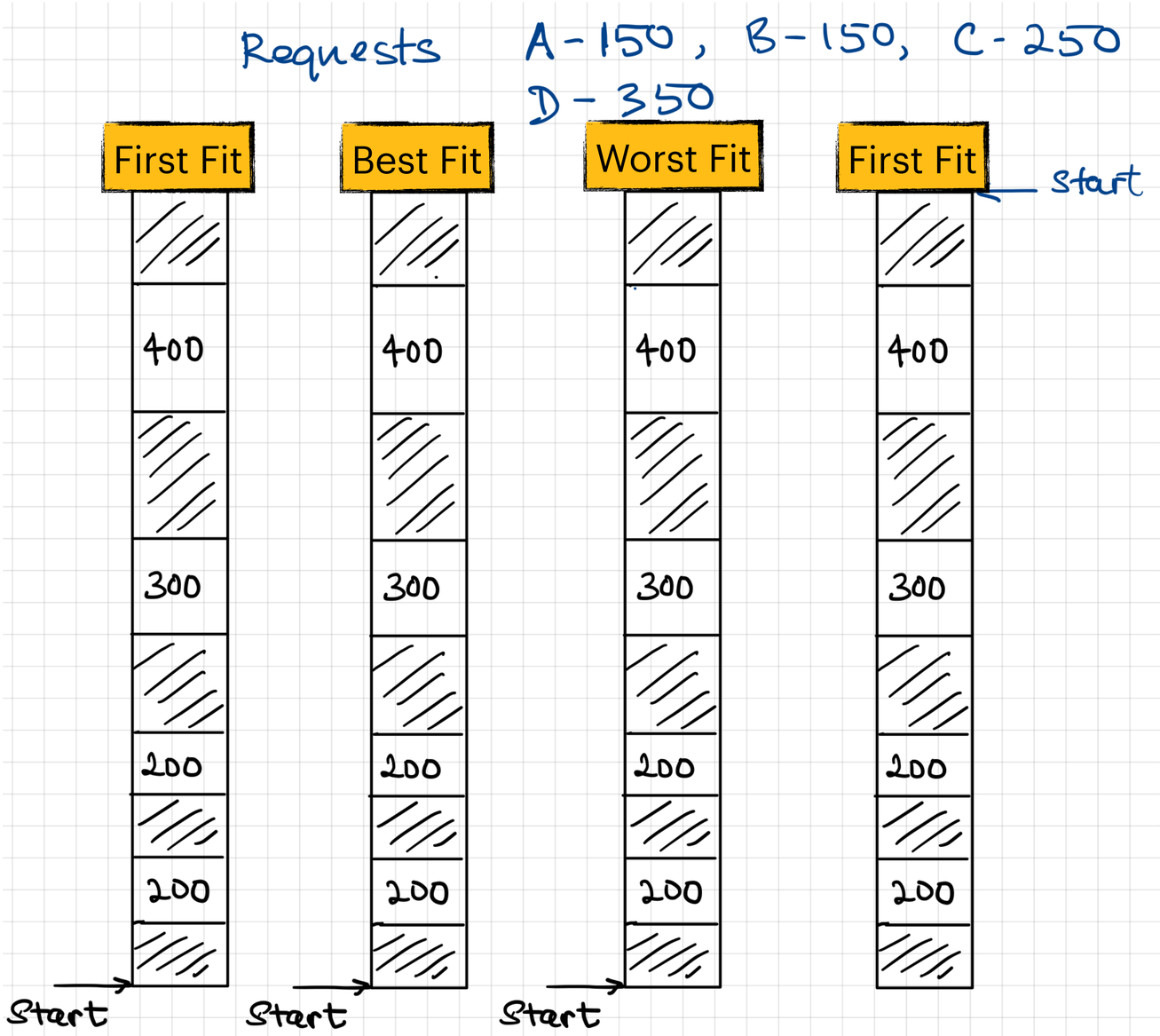


Dynamic Storage Allocation Problem



- How to satisfy a request of *n* from a list of free holes?
- Need to find a free hole that will satisfy the request
  - **First fit**: Allocate the hole that is big enough scanning the memory from the start location. We stop the search as soon as the hole to fit the request is found.
  - **Best fit**: Allocate the hole that is smallest and yet big enough. Must search the entire list of free holes.
  - **Worst fit**: Allocate the largest hole assuming it fits the request. Must search the entire list of free holes. Produces the largest leftover hole, which might be useful - sort of balance the hole sizes
  - **Next fit**: A variation of the first that starts the next search from where the previous one left.

Example Memory Allocations



Fragmentation

- Variable sized partitions suffer from **external fragmentation**
- Memory space is broken into small pieces as processes are loaded and unloaded
- Total free memory is large to fit the request, but no single block is large enough to fit the request ⇒ request denied allocation
- Statistical analysis of first fit shows, with N blocks $0.5N$ blocks will be lost to fragmentation ⇒ known as the 50-percent rule
- Sometimes leftover memory is not worth tracking (few bytes), in that case we can over allocate to a request ⇒ internal fragmentation (unused space within an allocation)
- Over allocation is also a way to accommodate growing memory regions