# Memory: Overview

CS439: Principles of Computer Systems

March 2, 2015

# Where We Are In the Course

- Just finished:
  - Processes & Threads
  - CPU Scheduling
  - Synchronization
- Next:
  - Memory Management
  - Virtual Memory
  - Heap Memory Management
  - File Systems
- Finally:
  - Networks
  - Other Topics
    - Distributed and Parallel Systems, Security

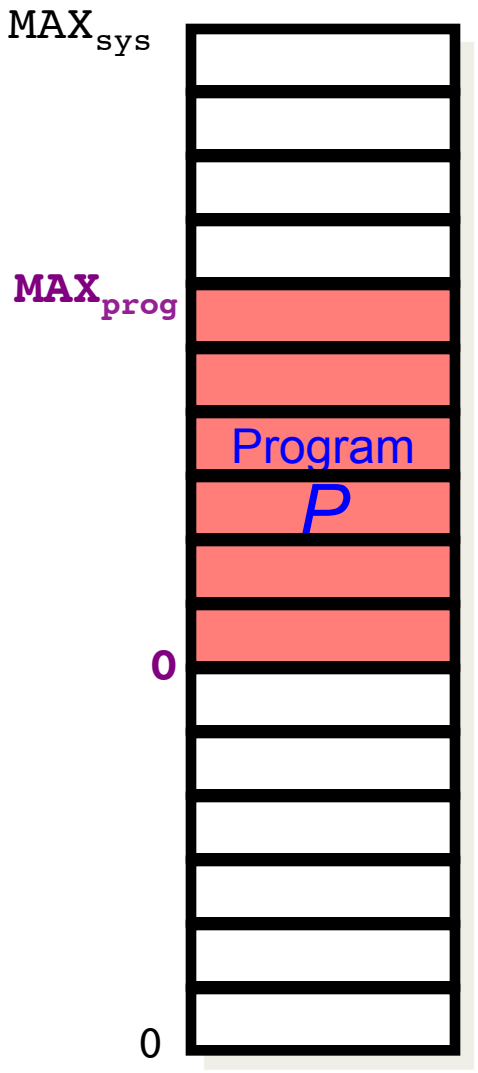
# Today's Agenda

Memory: Overview

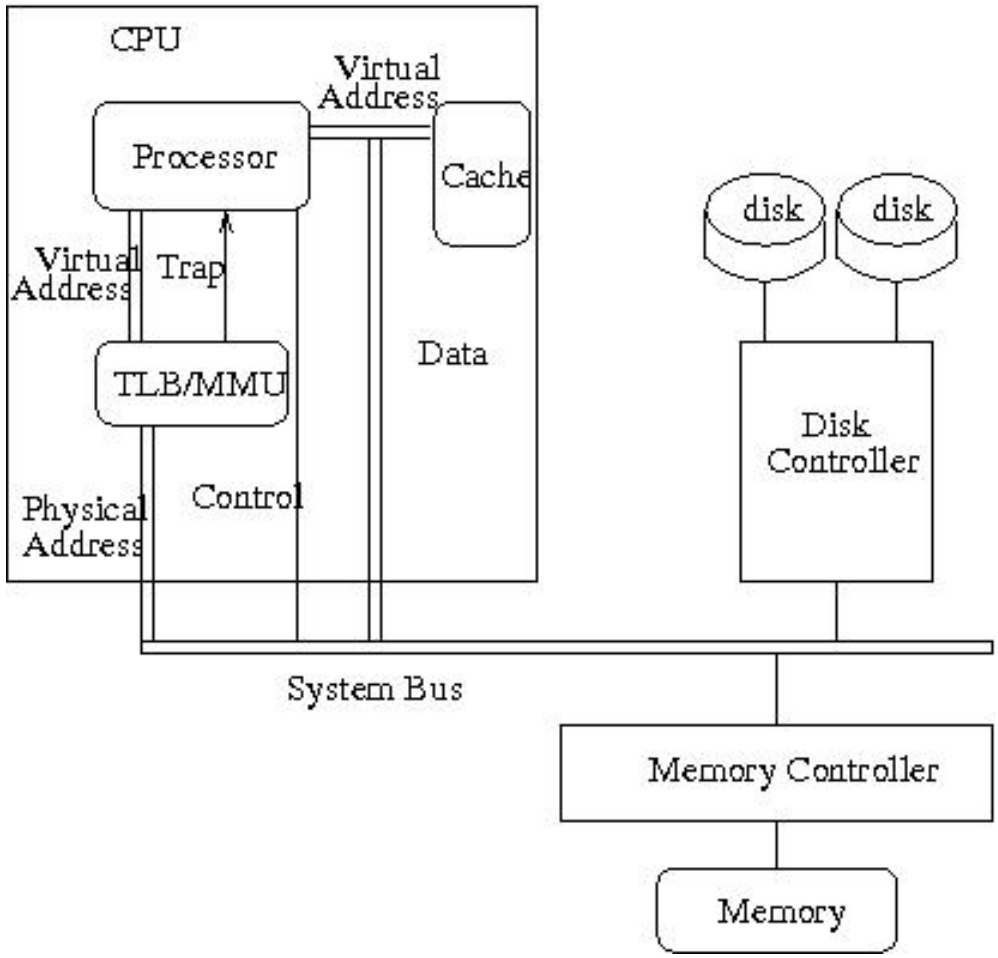
- Where is the executing process?
- How do we allow multiple processes to use main memory simultaneously?
- What is an address and how is one interpreted?

# Basic Concepts: Address Spaces

- ***Physical address space***: Collection of memory addresses supported by the hardware
  - From address 0 to address $\mathbf{MAX_{sys}}$
- ***Logical/virtual address space:*** Collection of addresses that the process can access (this is the process's view)
  - From address 0 to address $\mathbf{MAX_{prog}}$
- ***Segment***: A chunk of memory assigned to a process



# Back to Architecture



# Back to Architecture: Text Description

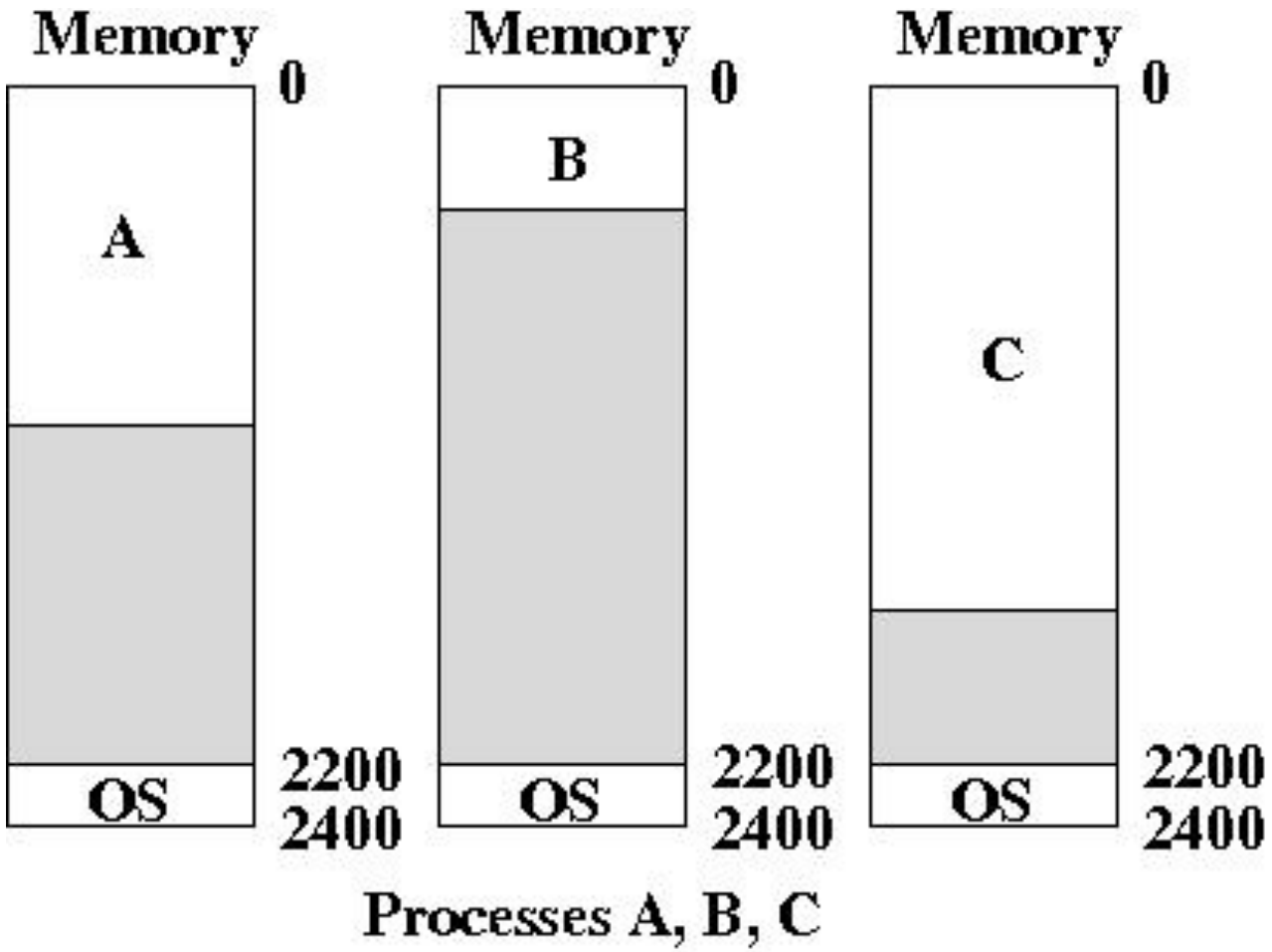
- This diagram shows a very basic layout of the relevant architecture.
- The CPU contains a processor, caches, and the TLB/MMU.
  - The CPU generates a virtual address
  - Using that address, it checks the cache for the relevant data (if the cache is virtually addresses, some are, some aren't)
  - That address must be translated to a physical address before memory can be accesses. That translation happens in the TLB/MMU, which will be discussed and explained in the next lecture. For now, it is a black box.
- Once the physical address is obtained, the CPU uses the system bus to access that memory location.
- Memory is attached to the system bus through a memory controller, which is a piece of hardware that controls memory.
- Any disks are also located off of the system bus, and they are controlled by a disk controller(s).

# Basic Concepts: Address Generation

**Uniprogramming:**

- One process executes at a time
- Process is always loaded starting at address 0
- Process executes in a contiguous section of memory
- OS gets fixed part of memory
- Compiler can generate physical addresses
- Maximum address = Memory Size – OS Size
- OS is protected from processes by address checking

# Uniprogramming Picture



# Uniprogramming Picture: Text Description

- This picture shows three versions of memory. Each has addresses from 0 to 2400.
- In the first, process A is loaded at address 0, and the OS is loaded at address 2200.
- In the second, process B is loaded at address 0, and the OS is loaded at address 2200.
- In the third, process C is loaded at address 0, and the OS is loaded at address 2200.

# Multiple Programs Share Memory

**Transparency**:

- We want multiple processes to coexist in memory
- No process should be aware that memory is shared
- Processes should not care what physical portion of memory they get

**Safety**:

- Processes must not be able to corrupt each other
- Processes must not be able to corrupt the OS

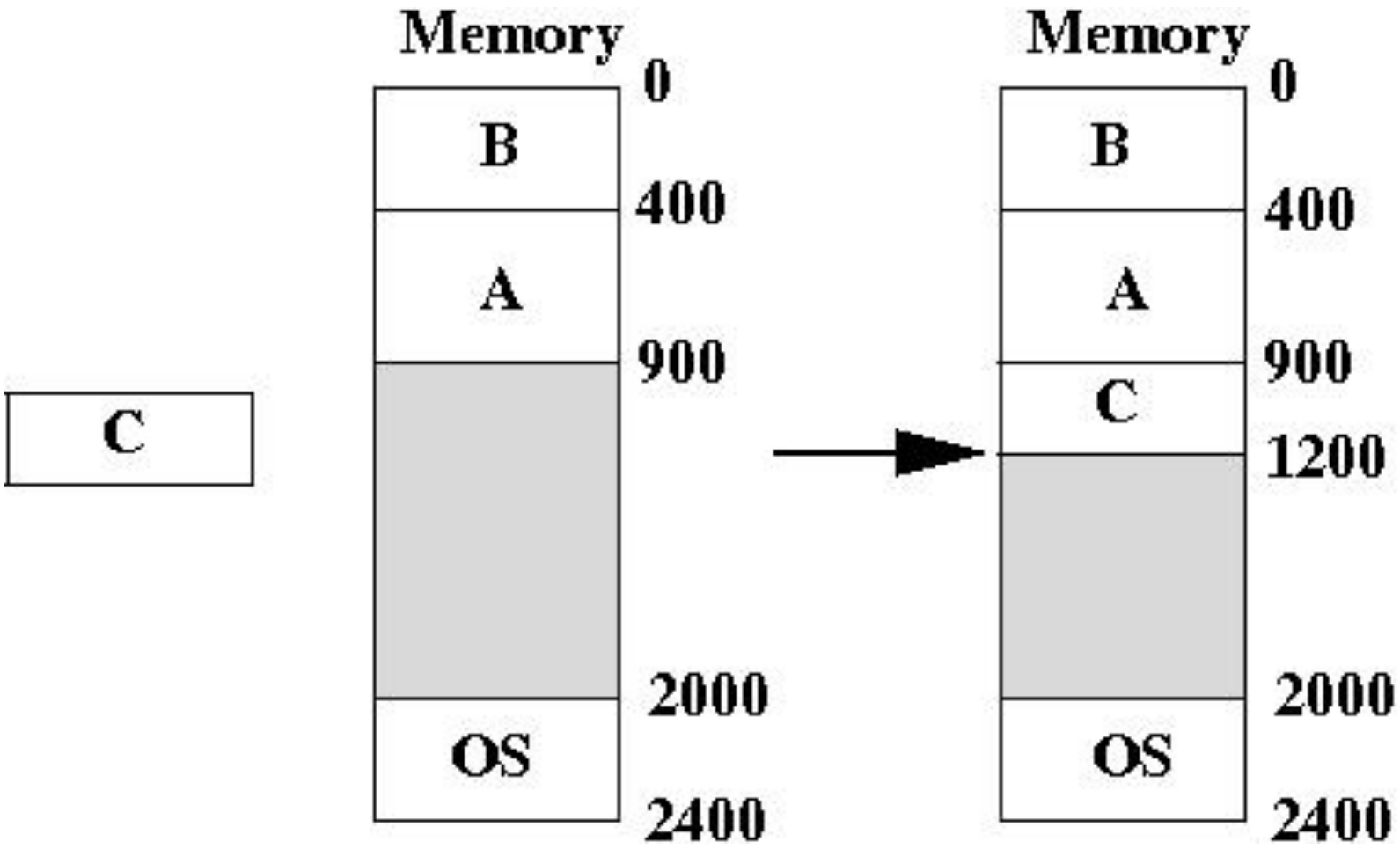
**Efficiency**:

- Performance of the CPU and memory should not be degraded badly due to sharing

# Relocation

- Put the OS in the highest memory
- Assume at compile/link time that the process starts at 0 with a Maximum address = Memory Size – OS Size
- When the OS loads the process, it allocates a contiguous segment of memory in which the process fits. If it does not fit, the OS waits for a process to terminate
- The first (smallest) physical address of the process is the *base* address and the largest physical address the process can access is the *limit* address
- The base address is also known as the *relocation address*

# Relocation Picture



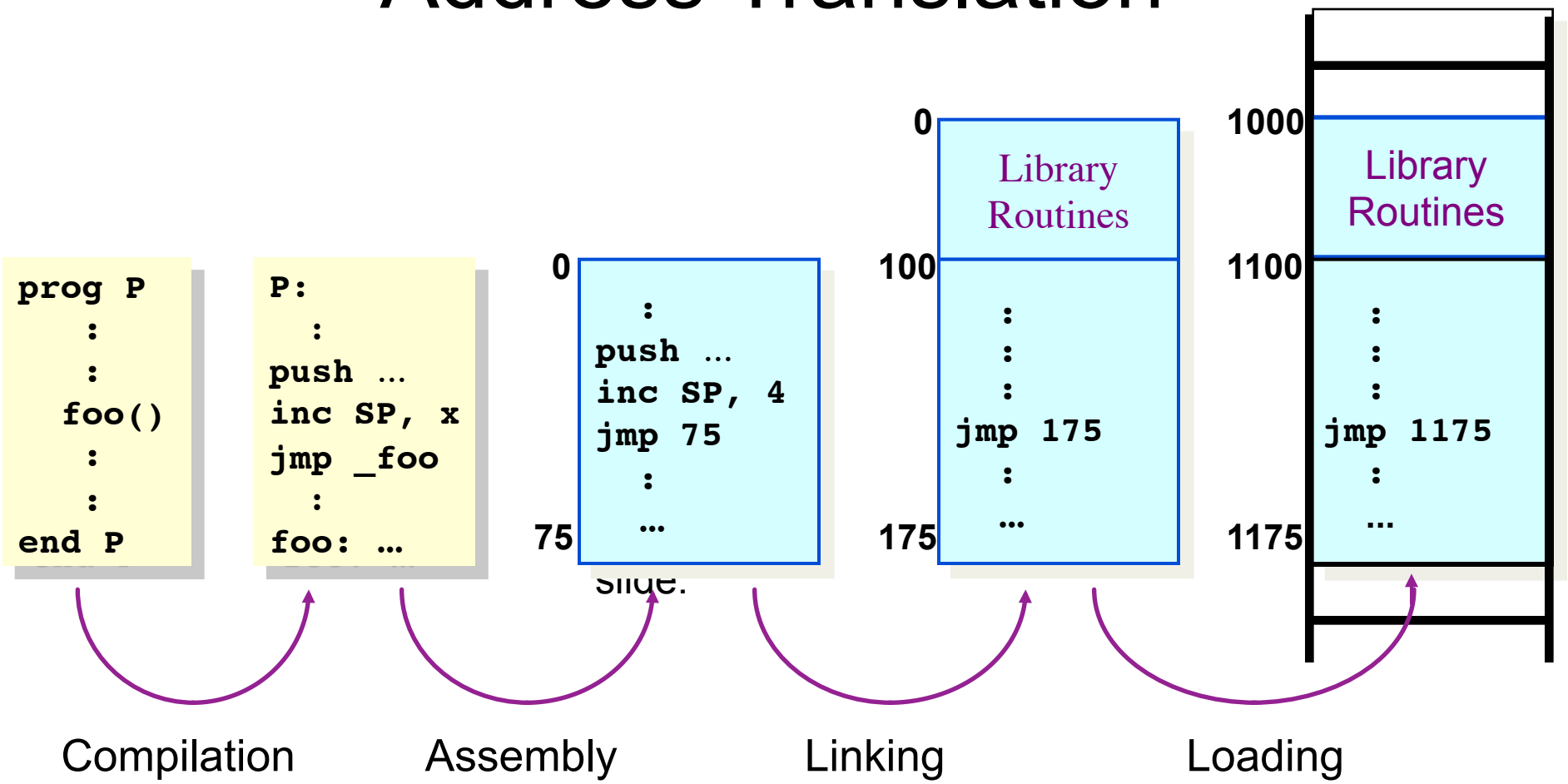
# Relocation Picture: Text Description

- This picture shows two versions of memory, a Before version and an After version. Both versions have addresses from 0 to 2400.
- The Before version shows
  - Process B loaded into memory from address 0 to 400.
  - Process A loaded into memory from address 400 to 900.
  - The OS loaded into memory from address 2000 to 2400.
- Process C enters the system and the OS must assign it physical memory.
- The After version shows
  - Processes A, B, and the OS at the same locations as the Before version
  - Process C loaded into memory from address 900 to 1200.

# Relocation: Two Types

- Static:
  - OS adjusts the addresses in a process (at load time) to reflect its location in memory
  - Once process is assigned a place in memory and starts executing, OS cannot move it
- Dynamic: In parallel
  1. Hardware adds relocation register (base) to virtual address to get physical address
  2. Hardware compares address with limit register
     - address must be less than limit
     - if test fails, the processor raises an exception

# Static Relocation: Address Translation

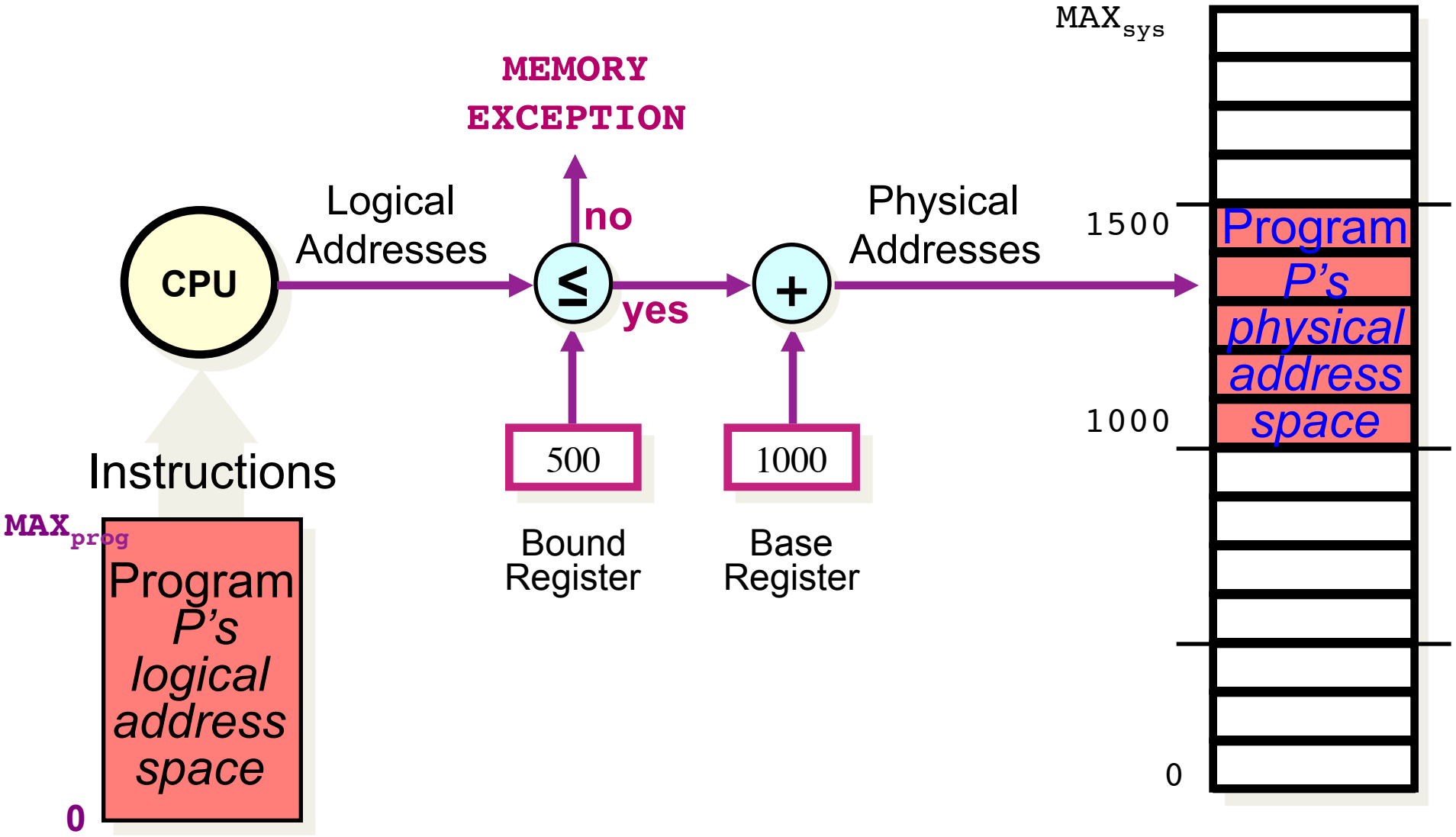



The Compilation Pipeline

# Static Relocation: Text Description

- Order of 4 Stages in Compilation Pipeline: Compilation, Assembly, Linking and Loading
- During the Compilation stage addresses go from lexical locations in the user's text to labels. These labels are still lexical locations in an assembly language program. Instructions are converted to assembly.
  - Example: a call to foo() becomes a jump to _foo and the function definition is now labeled with foo:
- During the Assembly stage the labels become addresses in a logical address space.
  - Example: A space where valid addresses range from 0 to 75. The jmp _foo instruction is now a jmp 75 instruction (where 75 is the location of the _foo definition) and the _foo label is removed.
- During the Linking stage addresses are adjusted in a new logical address space that contains library routines.
  - Example: Now the Library Routines live in addresses 0 to 100 and the addresses from the previous step now live in addresses 100 to 175. Essentially 100 was added to every address that existed before the Linking happened. The call to foo is now a jmp 175 instruction.
- During Loading the addresses are translated into actual physical addresses in the system, and that address is determined by where the system has room for the process.
  - Example: The logical address space will actually start at 1000 not 0, so we have to add 1000 to all the addresses from the last step. Now Library Routines live in addresses 1000 - 1100 and program addresses live in addresses 1100 to 1175. The call to foo is now a jump 1175 instruction.

# Dynamic Relocation: Address Translation



# Dynamic Relocation: Text Description

- How we get from a program's logical address space to physical memory.
- The bound register is used to make sure that the program isn't trying to access memory outside of its space. For example, if the programs logical address space ranges from 0 to 500 then the bound register would hold a value of 500.
- The base register holds the beginning of that program's addresses in physical memory. So if the programs memory started at address 1000 then the base register would hold 1000.
- Steps to Address Translation:
  - The program gives a logical address, or instruction, to the CPU.
  - The MMU now does 2 things at once:
    - It checks the address against the bound register, if the address is greater than the bound register a memory exception is thrown. This exception is a hardware interrupt that will be handled by the OS. This exception indicates the program was trying to access something that doesn't belong to it.
    - It adds the base register to the logical address to get the physical address. For example logical address 8 would become 1008 if the base register was 1000.
- NOTE: Even when the address is successfully translated, you are not protected from memory errors: you could still accidentally try to access memory in your address space that hasn't be initialized yet.

# Base and Bound: Implementation

- Hardware:
  - Add base and bound registers to the CPU
- Software
  - Add base and bound registers to PCB
  - On context switch, change base and bound registers (privileged)

# iClicker Question

Address Translation in Dynamic Relocation does the following steps:

A. Converts address by adding base, then checks against limit

B. Checks address against limit, then adds base

C. Check address against limit and adds base in parallel

# Dynamic Relocation

**Advantages**

- OS can easily move a process during execution
- OS can allow a process to grow over time
- Protection is easy
- Simple, fast hardware: Two special registers (add and compare)

**Disadvantages**

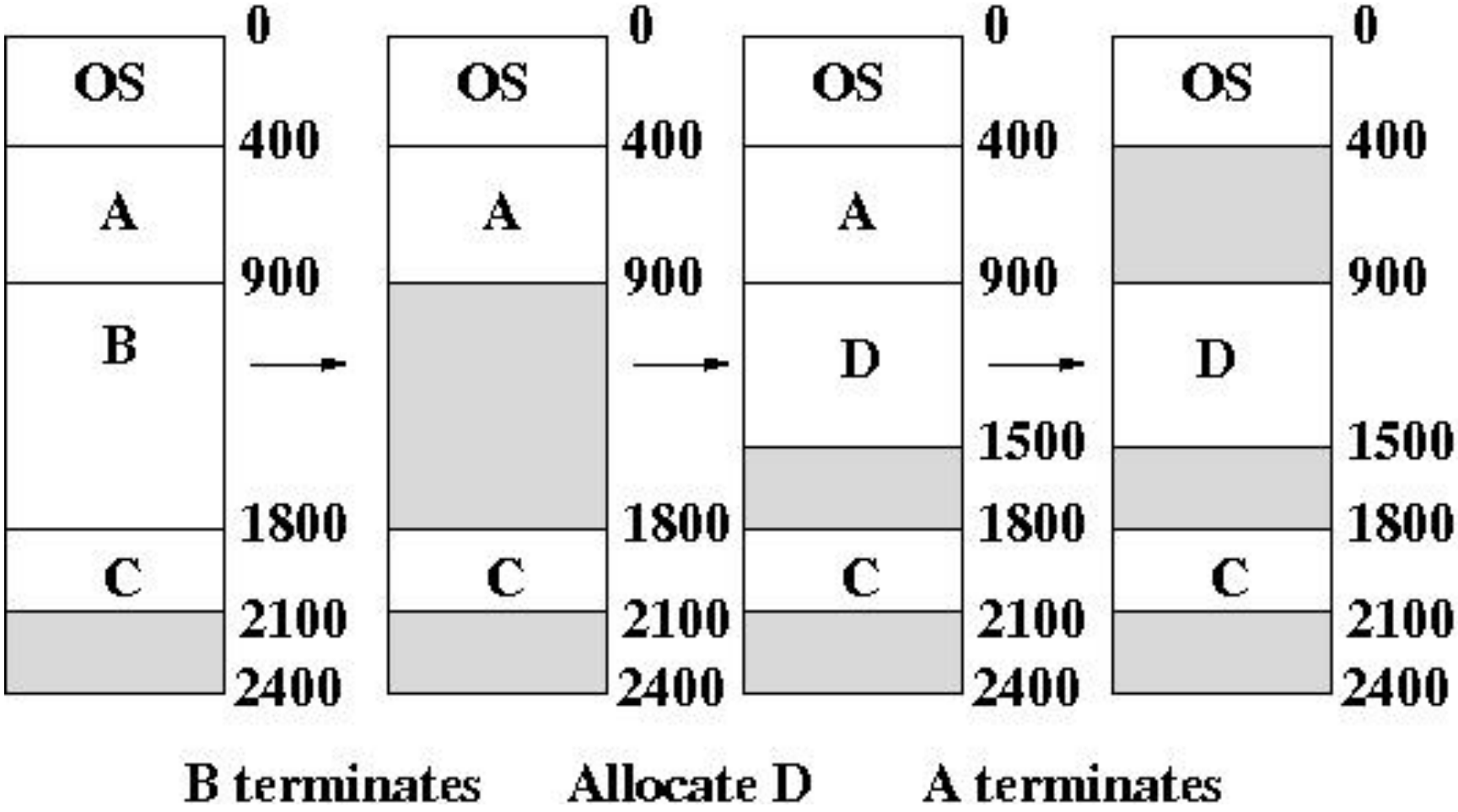
- Requires contiguous allocation
- Sharing is hard
- Degree of multiprogramming is limited (all active processes must fit in memory!)
  - And each process is limited to physical memory size
- Slows down hardware due to the add on every memory reference
- Complicates *memory management*

# Memory Management

As processes enter the system, grow, and terminate, the OS must track which memory is available (the holes) and which is utilized

Given a memory request from a starting process, the OS must decide which hole to use for the process

# Memory Management Picture



# Memory Management Picture: Text Description

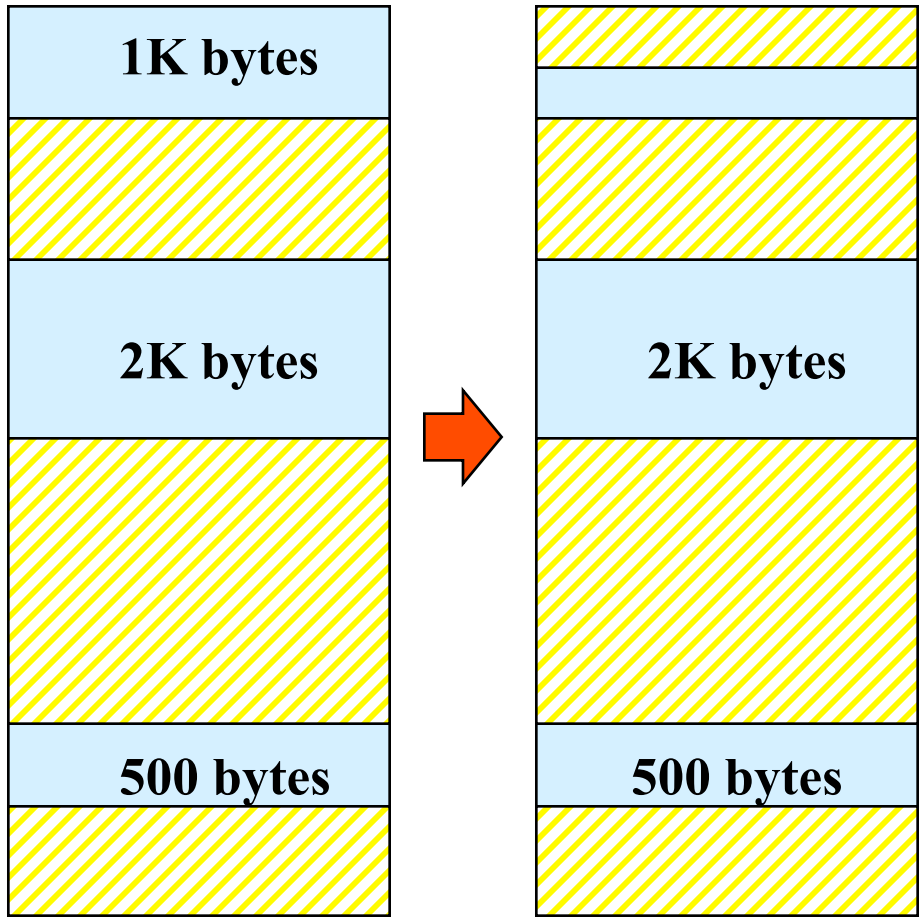
- We have a system with 4 processes
  - Process A has size 500
  - Process B has size 900
  - Process C has size 300
  - Process D has size 600
- The OS has a size 400 and will always live at the top of memory. So addresses 0 to 400 will always hold the OS.
- The full memory covers addresses 0 to 2400
- Originally processes A, B and C are running on the system. So the memory has the following setup: The OS is in addresses 0 to 400, process A is in addresses 400 to 900, process B is in addresses 900 to 1800 and process C is in addresses 1800 to 2100. There are also unused addresses from 2100 to 2400.
- Now process B terminates, this means there is a 900 address gap in the system. Now addresses 900 to 1800 are free space.
- Process D is allocated with size 600. The OS decides to stick in the gap just left by process B when it terminated. Now process D lives from 900 to 1500 and there is a gap from address 1500 to 1800.
- Finally process A terminates, leaving a gap in memory from 400 to 900. All the other processes stay where they were originally placed. The final setup is as follows: The OS is in address 0 to 400, there is a gap from addresses 400 to 900, process D lives from 900 to 1500, there is a gap from 1500 to 1800, process C has addresses 1800 to 2100 and the last 300 addresses, from 2100 to 2400 are free.
- Next, we look at the policies the OS uses to choose where to place the processes in memory.

# Memory Allocation Policies: Evaluation

- Minimize wasted space
- Two types of wasted space:
  - **External Fragmentation**
    - Unused memory between units of allocation
    - OR holes between processes
    - For example, two fixed tables for two but a party of four
  - **Internal Fragmentation**
    - Unused memory within a unit of allocation
    - For example, a party of three at a table for four

# Memory Allocation Policies: First-Fit Example

To allocate *n* bytes, use the *first* available free block such that the block size is larger than *n.*




To allocate 400 bytes, we use the 1st free block available

# First-Fit Example: Text Description

- First-fit: To allocate *n* bytes, use the *first* available free block such that the block size is larger than *n.*
- The example shows a memory region:
  - 1000 bytes are free
  - Some space is allocated
  - 2000 bytes are free
  - Some space is allocated
  - 500 bytes are free
  - Some space is allocated
- The OS needs to allocate 400 bytes, so it uses the first free space that is at least 400 bytes
- Overtime the holes will get smaller and smaller as new processes added and squeezed in the first place they fit.

# Memory Allocation Policies: First-Fit

**Goal**: Simplicity of Implementation

**Requirements:**

- Free block list sorted by address
- Allocation requires a search for a suitable position
- De-allocation requires a check to see if the freed partition could be merged with any adjacent free partitions

**Advantages**

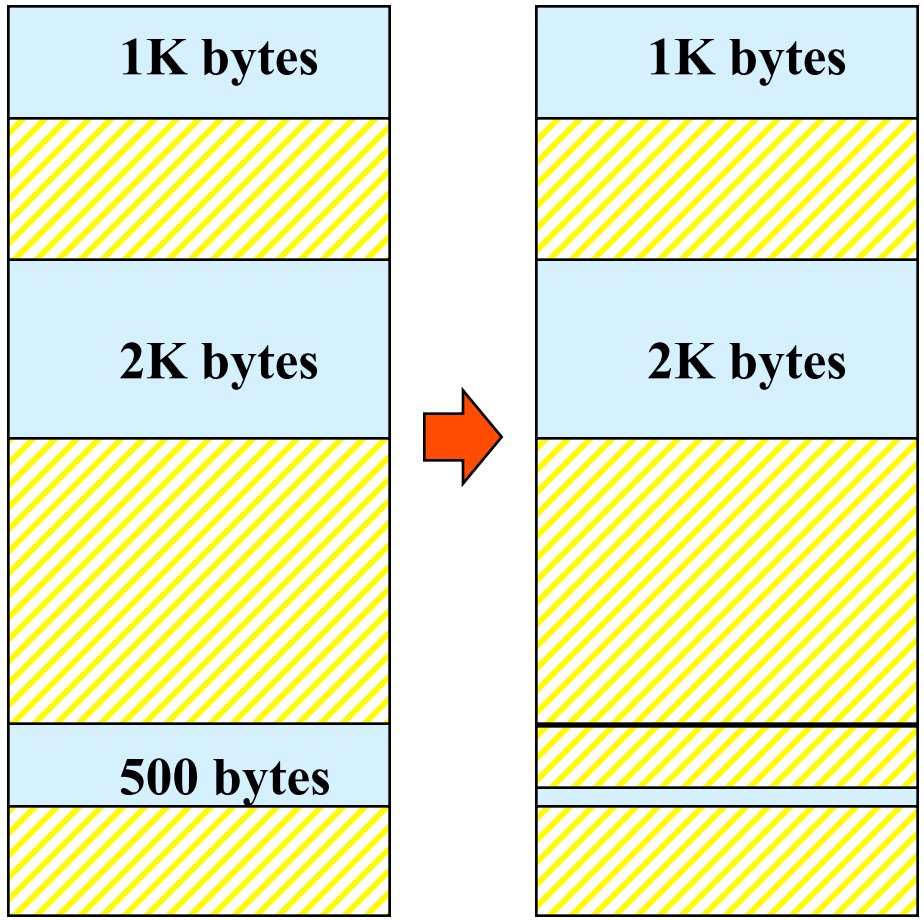
- Simple to implement
- Tends to produce larger free blocks towards the end of the address space

**Disadvantages**

- Slow allocation
- External fragmentation

# Memory Allocation Policies: Best-Fit

To allocate *n* bytes, use the *smallest* available free block such that the block size is larger than *n.*




To allocate 400 bytes, we use the 3rd free block available (smallest)

# Best-Fit: Text Description

- To allocate *n* bytes, use the *smallest* available free block such that the block size is larger than *n*
- The example shows a memory region:
  - 1000 bytes are free
  - Some space is allocated
  - 2000 bytes are free
  - Some space is allocated
  - 500 bytes are free
  - Some space is allocated
- The OS needs to allocate 400 bytes, so it uses the 500-byte space because it is the smallest available block that can hold them.
- Over time the free spaces will become very small as small pieces of memory are left out during best fit.

# Memory Allocation Policies: Best-Fit

**Goals**:

- To avoid fragmenting big free blocks
- To minimize the size of resulting external fragments

**Requirements:**

- Free block list sorted by size
- Allocation requires a search for a suitable position
- De-allocation requires a check to see if the freed partition could be merged with any adjacent free partitions

**Advantages**

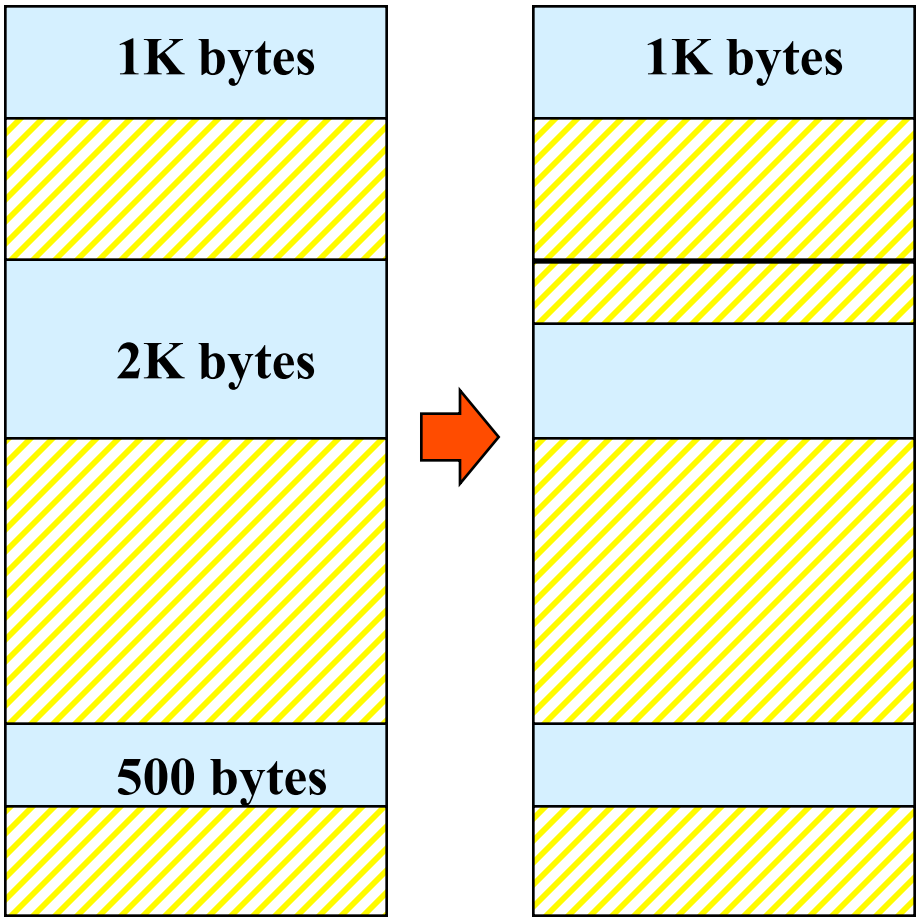
- Works well when most allocations are of small size
- Relatively simple

**Disadvantages**

- External fragmentation
- Slow de-allocation (why?)

# Memory Allocation Policies: Worst-Fit

To allocate *n* bytes, use the *largest* available free block such that the block size is larger than *n.*




To allocate 400 bytes, we use the 2nd free block available (largest)

# Worst-Fit: Text Description

- To allocate n bytes, use the largest available free block such that block size is larger than n.
- The example shows a memory region:
  - 1000 bytes are free
  - Some space is allocated
  - 2000 bytes are free
  - Some space is allocated
  - 500 bytes are free
  - Some space is allocated
- The OS needs to allocate 400 bytes, so it uses the 2000-byte space because it is the largest.
- Over time the gaps would become smaller and bigger pieces would be broken up, making it harder to allocate memory for a bigger process later.

# Memory Allocation Policies: Worst-Fit

**Goals**:

- To avoid having too many tiny fragments

**Requirements:**

- Free block list sorted by size
- Allocation is fast (get the largest)
- De-allocation requires a check to see if the freed partition could be merged with any adjacent free partitions

**Advantages**

- Works best if allocations are of medium size

**Disadvantages**

- External fragmentation
- Slow de-allocation
- Tends to break large free blocks such that large partitions cannot be allocated

# iClicker Question

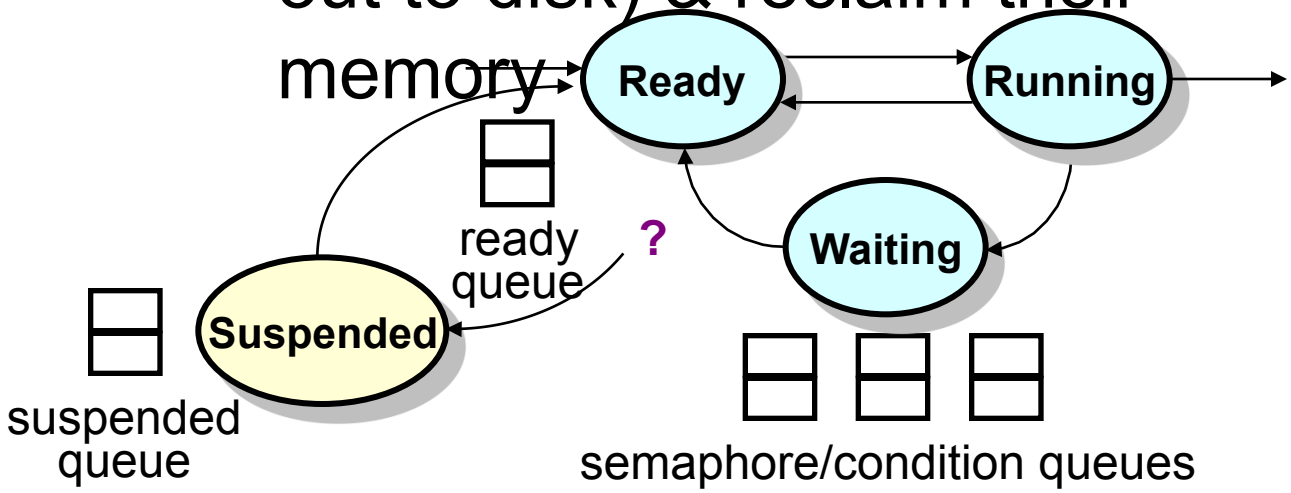
First-fit, Best-fit, and Worst-fit all suffer from external fragmentation:

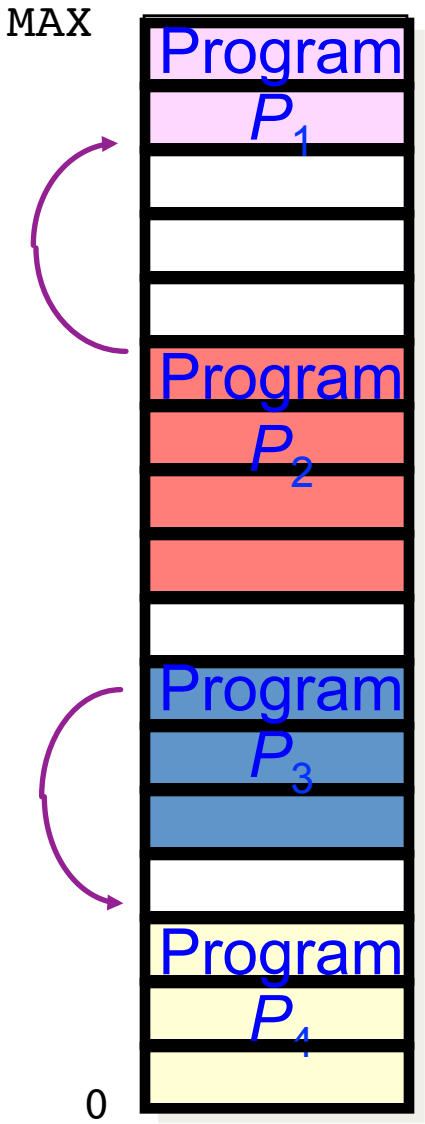
A. True

B. False

# Eliminating Fragmentation

- Compaction
  - Relocate programs to coalesce holes
- Swapping
  - Preempt processes (roll them out to disk) & reclaim their memory



# Eliminating Fragmentation: Text Description

- Compaction: relocate programs to coalesce holes
  - Example: Have program A at addresses 0 to 200, a gap from addresses 200 to 500, program B at addresses 500 to 1000 and another gap from 1000 to 500. To coalesce would move program B to be from addresses 200 to 700, eliminating the gap between the two programs and making a gap after them from addresses 700 to 1500. This would allow the gap after program B to go from size 500 to size 800.
- Swapping: preempt processes, roll them out to disk, and reclaim their memory
  - Allows total amount of memory being used to exceed the total amount of physical memory.
  - Example: Have program A at addresses 0 to 200 and a gap from addresses 200 to 300 and all other memory is taken. Need to allocate program D of size 300. Since program A hasn't been used in a while we can move it out to disk, leaving a 300 address gap in the memory. Now program D can move into that 300 address gap.
  - This adds another state to our process life cycle: suspended. Processes that are swapped out to memory are suspended. Processes can be suspended from either the blocked state or the ready queue, but they are usually not restored to memory until they can be added to the ready queue again.

# Summary

- Processes must reside in memory in order to execute
- Addresses need context to be interpreted, a logical/virtual address can be very different from a physical address.
- Processes generally use virtual addresses which are translated into physical addresses just before accessing memory
- Relocation allows multiple processes to share main memory, but makes it expensive for processes to grow over time
- Swapping allows the total memory being used by all processes to exceed the amount of physical memory available
- C and C++ memory managers use these same principles

# Announcements

- Project 2 posted Friday due Friday, 3/27
  - Stack check due Monday, 3/9
  - Difficult
  - Typing in the code is not the hard part---conceptualizing what you should do and how IS
  - You MUST get this one working---Projects 3 and 4 require it
- Groups of 2 to 4
  - You MUST work in a group
- Homework 5 due Friday