

---

# CSC 369

Week 6: *Memory Management*

Reading: Text, 4.1-4.3.1, 4.8

# Reminders

- Assignment 1 due Friday
  - Make src/asst1 dir for design document
  - Make sure all files are committed to your repo before the deadline
  - A grace day is any period of time greater than 1 minute and less than 24 hours
    - Fill in form on assignments page if you are using a grace day
- Midterm after Reading Week in tutorial time
  - Check midterm page for locations
  - Includes this week's material

# Memory Management

- Every active process needs memory
- CPU scheduling allows processes to share (multiplex) the processor
- Must figure out how to share main memory as well
- What should our goals be?
  - Support enough active processes to keep CPU busy
  - Use memory efficiently (minimize wasted memory)
  - Keep memory management overhead small
  - ... while satisfying five basic requirements

# Requirements

- **Relocation**

- Programmers don't know what physical memory will be available when their programs run
- Medium-term scheduler may swap processes in/out of memory, need to be able to bring it back in to a different region of memory
- This implies we will need some type of address translation

- **Protection**

- A process's memory should be protected from unwanted access by other processes, both intentional and accidental
- Requires hardware support

# More requirements

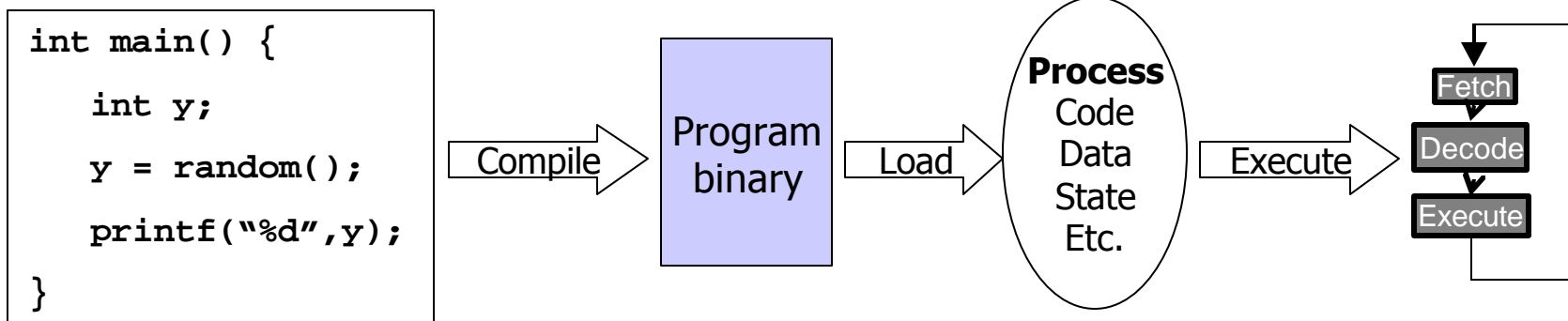
- **Sharing**
  - In some instances, processes need to be able to access the same memory
  - Need ways to specify and control what sharing is allowed
- **Logical Organization**
  - Machine accesses/addresses memory as a one-dimensional array of bytes
  - Programmers organize code in modules
  - Need to map between these views
- **Physical Organization**
  - Memory and Disk form a two-level hierarchy, flow of information between levels must be managed
  - CPU can only access data in registers or memory, not disk

# Meeting the requirements

- Modern systems use **virtual memory**
  - Complicated technique requiring hardware and software support
  - Based on simpler techniques of **segmentation** and/or **paging**
- We'll build up to virtual memory by looking at some simpler schemes first
  - Fixed partitioning
  - Dynamic partitioning
  - Paging
  - Segmentation
- We'll begin with loading and address translation

# Address Binding

- Programs must be in memory to execute
  - Program binary is **loaded** into a process
    - Needs memory for code (instructions) & data
  - Addresses in program must be **translated** (mapped, bound) to real addresses
    - Programmers use **symbolic** addresses (i.e., variable names) to refer to memory locations
    - CPU fetches from, and stores to, real memory addresses

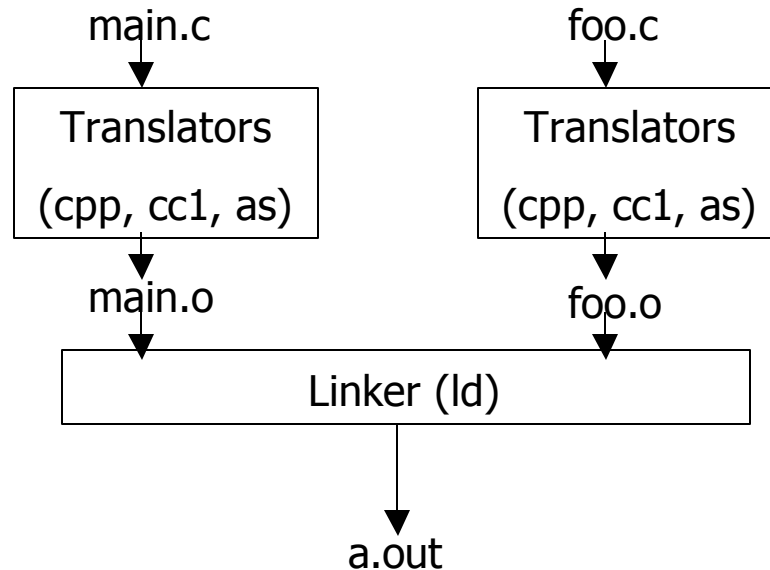


# When are addresses bound?

- Compile time
  - Must know what memory process will use during compilation
  - Called absolute code since binary contains real addresses
  - No relocation is possible
  - MS-Dos .COM programs worked like this
- Load time
  - Compiler translates (binds) symbolic addresses to **logical, relocatable** addresses within compilation unit (source file)
  - Linker takes collection of object files and translates addresses to logical, absolute addresses within executable
    - Resolves references to symbols defined in other files/modules
  - Loader translates logical absolute addresses to **physical** addresses when program is loaded into memory
  - Programs can be loaded to different address when they start, but cannot be relocated later

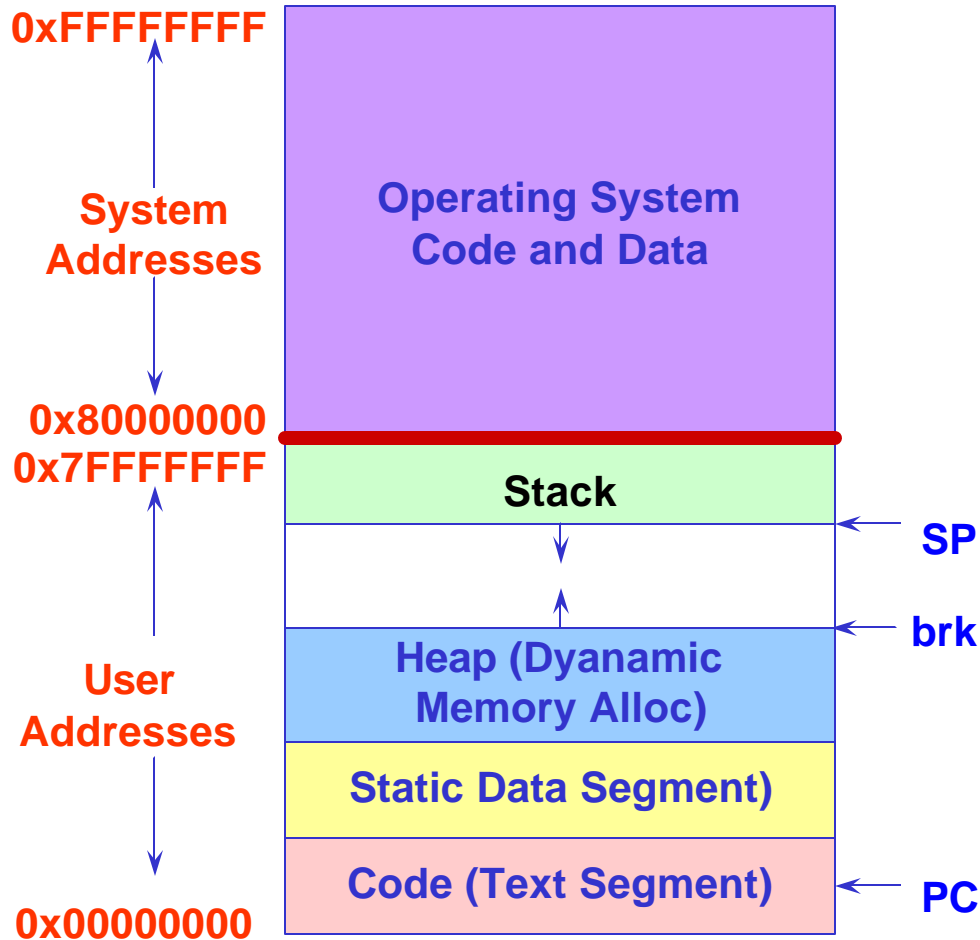
# A better plan

- Bind addresses at execution time



- Executable object file, `a.out`, contains logical addresses for entire program
  - translated to a real, physical address during execution
  - Flexible, but requires special hardware (as we will see)

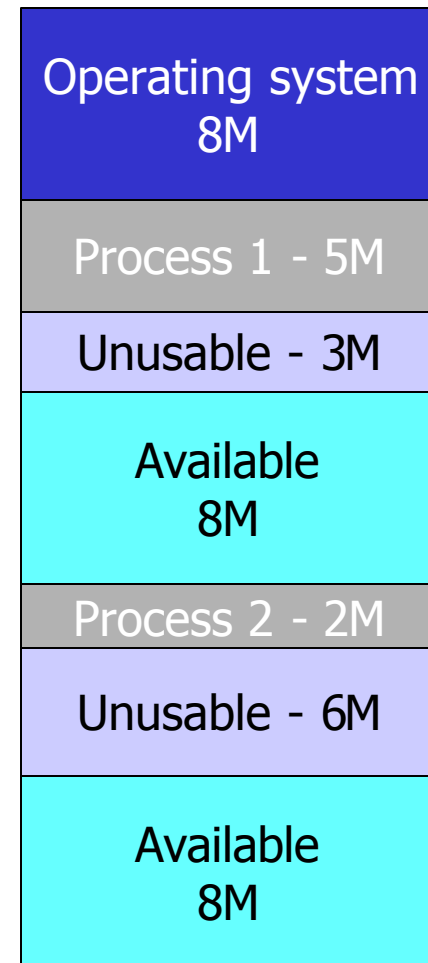
# Logical vs. Physical Address Space



- Recall process address space (A.S.) layout
  - logical or **virtual A.S.**
- CPU generates logical addresses in this space as program executes
  - Called **virtual addresses**
- Memory system must see physical (real) addresses
  - Translation is done by **memory management unit (MMU)**
  - Physical memory must be allocated for each virtual location used by the program

# Fixed Partitioning of Physical Memory

- Divide memory into regions with fixed boundaries
  - Can be equal-size or unequal-size
- Operating system occupies one partition
- A single process can be loaded into each remaining partition
  - Memory is wasted if process is smaller than partition (**internal fragmentation**)
  - Programmer must deal with programs that are larger than partition (**overlays**)

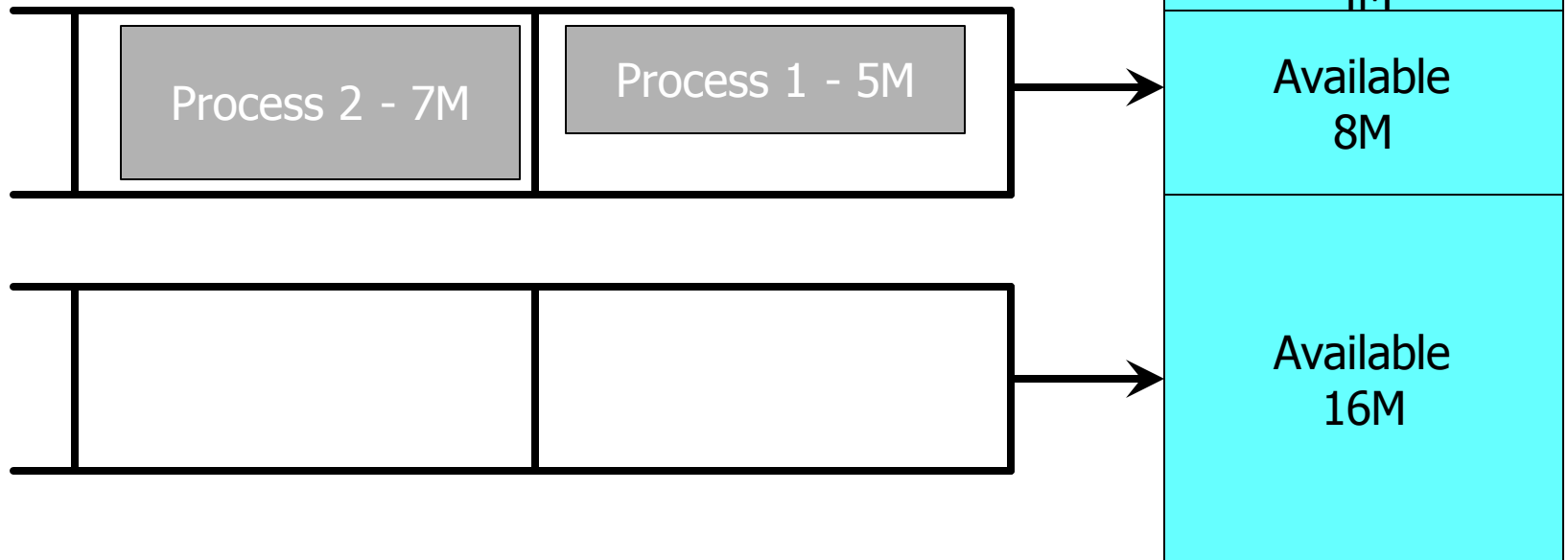


# Placement w/ Fixed Partitions

- Number of partitions determines number of active processes
- If all partitions are occupied by waiting processes, swap some out, bring others in
  - See text for details on swapping
- Equal-sized partitions:
  - Process can be loaded into any available partition
- Unequal-sized partitions:
  - Queue-per-partition, assign process to smallest partition in which it will fit
    - A process always runs in the same size of partition
  - Single queue, assign process to smallest available partition

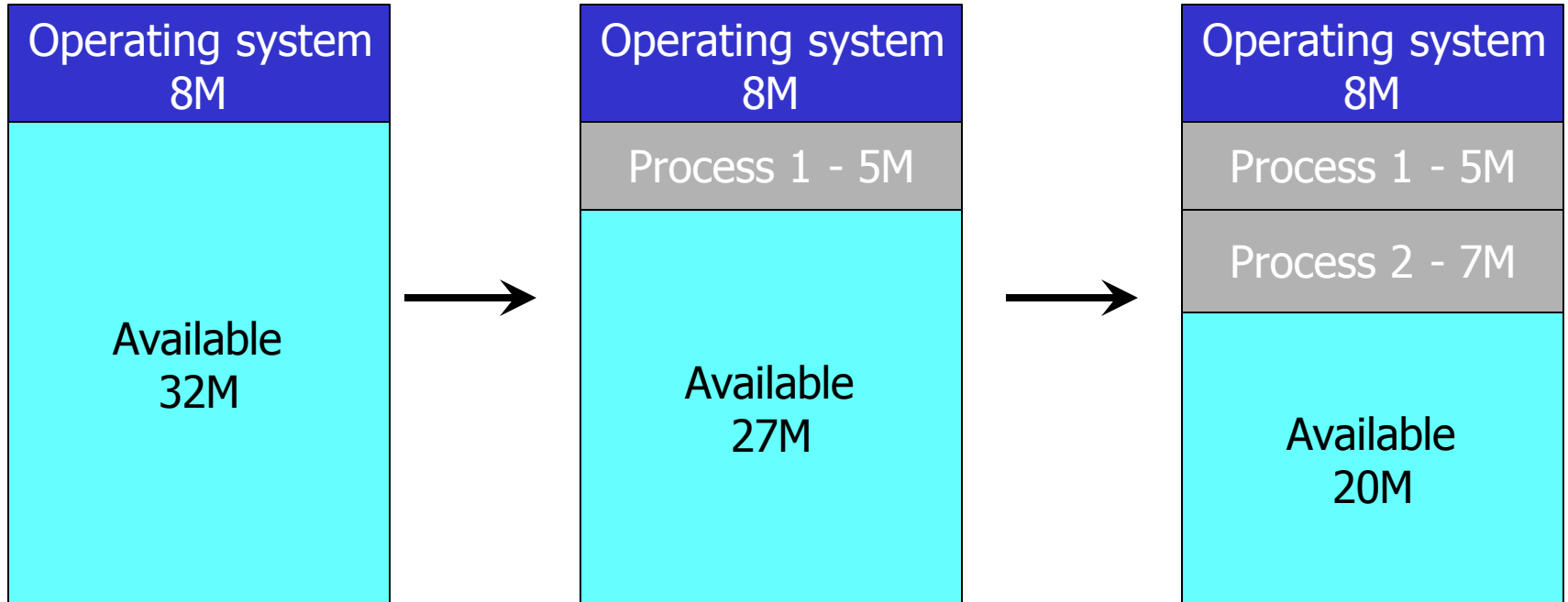
# Placement Example

Process 1 and Process 2 fit in same partition. With smallest-partition policy, both must share 8M partition while 16M partition goes unused.



# Dynamic Partitioning

- Partitions vary in length and number over time
- When a process is brought in to memory, a partition of exactly the right size is created to hold it



# More Dynamic Partitioning

- As processes come and go, "holes" are created
  - Some blocks may be too small for any process
  - This is called **external fragmentation**
- OS may move processes around to create larger chunks of free space
  - E.g. if Process 3 were allocated immediately following Process 1, we would have a 25M free partition
  - This is called **compaction**
  - Requires processes to be relocatable



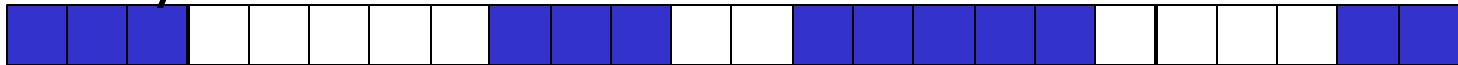
# Heap Management

- How are `malloc()` / `free()` implemented in C library?
  - `malloc(size)` returns a pointer to a block of memory of at least "size" bytes, or `NULL`
  - `free(ptr)` releases the previously-allocated block pointed to by "ptr"
  - Internally, `malloc/free` manage a contiguous range of logical addresses
    - Starts just after uninitialized data segment
    - Can be extended with `brk()` system call
  - Dynamic partitioning system, without relocation

# Tracking Memory Allocation

- Bitmaps
  - 1 bit per allocation unit
  - "0" == free, "1" == allocated
  - See kern/arch/mips/mips/dumbvm.c
    - Allocation unit is a page of physical memory
  - Allocating a N-unit chunk requires scanning bitmap for sequence of N zero's
    - Slow

Memory:

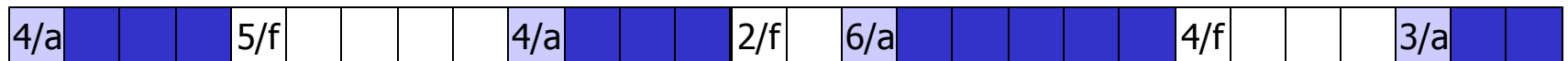


Bitmap:

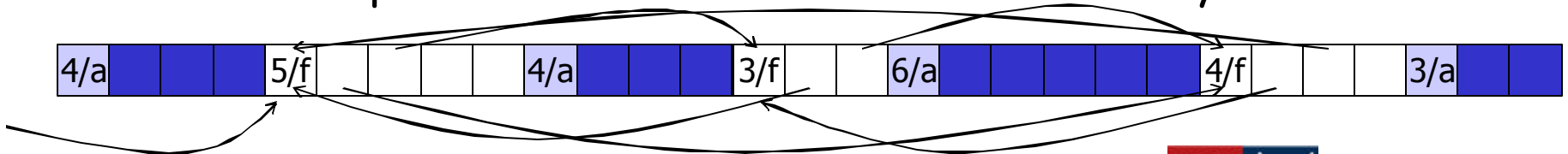
111000001110011111000011

# Tracking Allocation (2)

- Free lists
  - Maintain linked list of allocated and free segments
  - List needs memory too. Where do we store it?
- Implicit list
  - Each block has header that records size and status (allocated or free)
  - Searching for free block is linear in total number of blocks

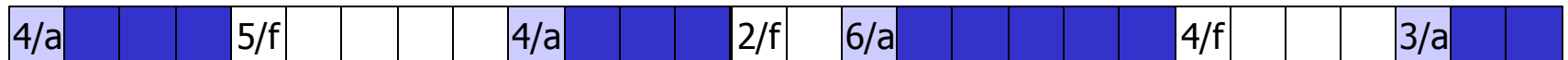


- Explicit list
  - Store pointers in free blocks to create doubly-linked list



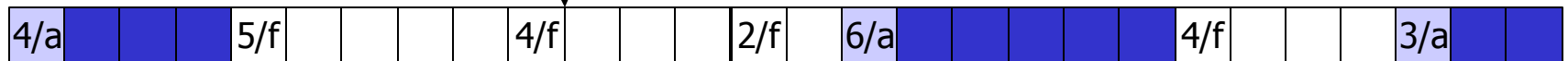
# Freeing Blocks

- Adjacent free blocks can be coalesced



↑  
p = malloc(3);

...  
free(p);



- Easier if all blocks end with a footer with size/status info (called boundary tag)

# Placement Algorithms

- Compaction is time-consuming and not always possible
- We can reduce the need for it by being careful about how memory is allocated to processes over time
- Given multiple blocks of free memory of sufficient size, how should we choose which one to use?
  - First-fit - choose first block that is large enough; search can start at beginning, or where previous search ended (called next-fit)
  - Best-fit - choose the block that is closest in size to the request
  - Worst-fit - choose the largest block
  - Quick-fit - keep multiple free lists for common block sizes

# Comparing Placement Algs.

- **Best-fit**
  - left-over fragments tend to be small (unusable)
  - In practice, similar storage utilization to first-fit
- **First-fit**
  - Simplest, and often fastest and most efficient
  - May leave many small fragments near start of memory that must be searched repeatedly
  - Next-fit variant tends to allocate from end of memory
    - Free space becomes fragmented more rapidly
- **Worst-fit**
  - Not as good as best-fit or first-fit in practice
- **Quick-fit**
  - Great for fast allocation, generally harder to coalesce

# Relocation

- Swapping and compaction require a way to change the physical memory addresses a process refers to
  - can we repeat address translation as done at initial load?
- Really, need dynamic relocation (aka execution-time binding of addresses)
  - process refers to **relative** addresses, hardware translates to physical address as instruction is executed
- Let's begin with minimum requirements to relocate fixed or dynamic partitions...
  - All memory used by process is **contiguous** in these methods

# Hardware for Relocation

- Basic idea: add relative address to process starting (base) address to form real, or physical, address
  - check that address generated is within process's space
- 2 registers, "base" and "limit"
  - When process is assigned to CPU (i.e., set to "Running" state), load base register with starting address of process
  - Load limit register with last address of process
  - On memory reference instruction (load, store) add base to address and compare with limit
  - If compare fails, trap to operating system
    - if ( $\text{addr} < \text{base} \parallel \text{addr} \geq (\text{base} + \text{limit})$ ) then trap
    - Illegal address exception

# Problems with Partitioning

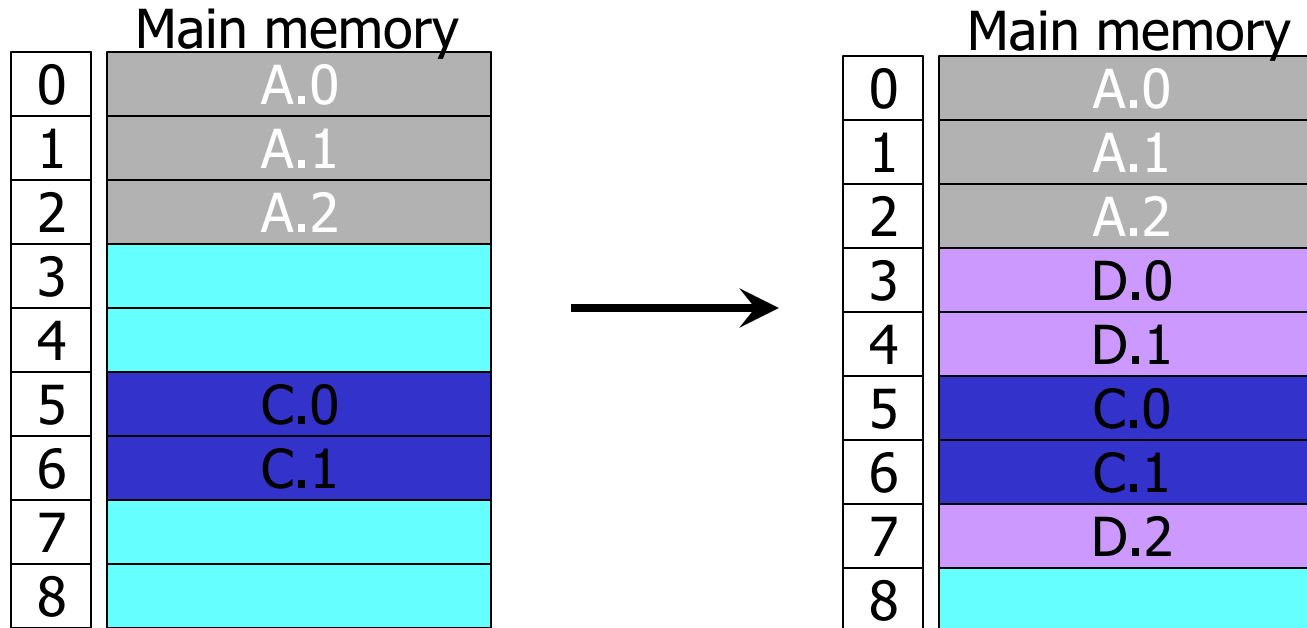
- With fixed partitioning, internal fragmentation and need for overlays are big problems
  - Scheme is too inflexible
- With dynamic partitioning, external fragmentation and managing the available space are major problems
- Basic problem is that processes must be allocated to contiguous blocks of physical memory
  - Hard to figure out how to size these blocks given that processes are not all the same
- We'll look now at **paging** as a solution

# Paging

- Partition memory into equal, fixed-size chunks
  - These are called **page frames** or simply **frames**
- Divide processes' memory into chunks of the same size
  - These are called **pages**
- Any page can be assigned to any free page frame
  - External fragmentation is eliminated
  - Internal fragmentation is at most a part of one page per process
- Possible page frame sizes are restricted to powers of 2 to simplify translation

# Example of Paging

Suppose a new process, D, arrives needing 3 frames of memory



- We can fit Process D into memory, even though we don't have 3 contiguous frames available!

# Support for Paging

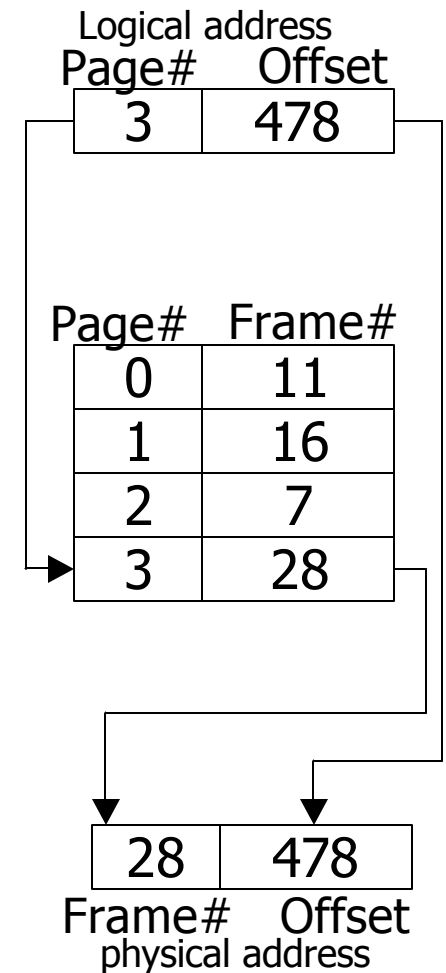
- Need more than base & limit registers now
- Operating system maintains a **page table** for each process
  - Page table records which physical frame holds each page
  - virtual addresses are now **page number + page offset**
    - **page number =  $vaddr / page\_size$**
    - **page offset =  $vaddr \% page\_size$**
    - Simple to calculate if page size is power-of-2
  - On each memory reference, processor translates page number to frame number and adds offset to generate a physical address
  - Keep a "page table base register" to quickly locate the page table for the running process

# Example Address Translation

- Suppose addresses are 16 bits, pages are 1024 bytes



- Least significant 10 bits of address provide offset within a page ( $2^{10} = 1024$ )
- Most significant 6 bits provide page number
- Maximum number of pages =  $2^6 = 64$
- To translate virtual address: 0xDDE
  - Extract page number (high-order 6 bits)
    - $\rightarrow pg = vaddr \gg 10 \quad (== vaddr/1024) == 3$
  - Get frame number from page table
  - Combine frame number with page offset
    - $offset = vaddr \& 0x3FF \quad (== vaddr \% 1024)$
    - $paddr = (frame \ll 10) | offset$ 
      - Equivalent to  $paddr = frame * 1024 + offset$



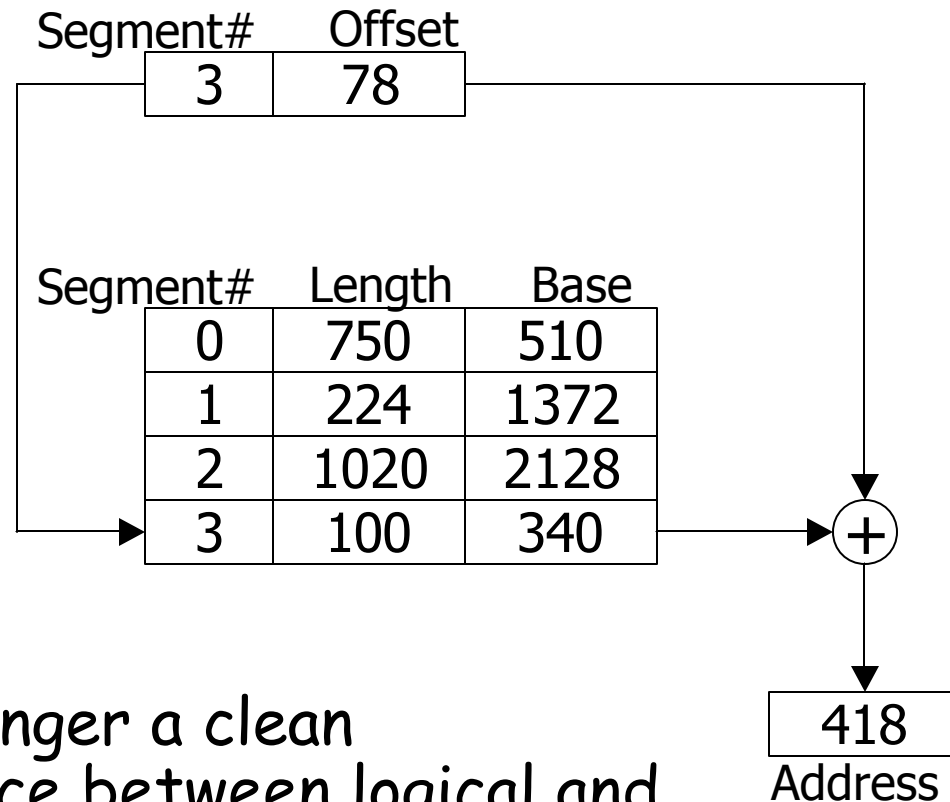
# Segmentation

- Alternate means of dividing user program
- Divisions reflect **logical** organization of the program
  - Text segment - read-only
  - Data segment - read/write, may be subdivided further
- Segments are variable-sized
  - A lot like dynamic partitioning, but a process may occupy multiple, non-contiguous segments
  - Suffers from external fragmentation
  - No simple mapping from logical to physical addresses

# Address Translation

- Operating system maintains a **segment table**
  - Like the page table, but records start address and length for each segment
  - Physical start of segment need not be power-of-2
- Logical addresses consist of a segment # and an offset within that segment
  - For translation, may reserve a fixed number of high-order bits for segment number
  - Maximum segment size is determined by the number of bits left for the offset.
  - E.g., 16 bit address, 4 bit segment number = 16 segments of max size 4096 bytes ( $2^{12} = 4096$ )

# Example



- There is no longer a clean correspondence between logical and physical addresses

# Next time...

- So far, we have managed a more efficient use of memory but still require that entire process be in physical memory when it executes
  - Supporting larger programs is the responsibility of the programmer via overlays
- We would like to remove this restriction and have the system manage partially loaded processes
  - This is the job of virtual memory, which is the topic of next week's lecture (after Reading Week)