06 - Main Memory

CEG 4350/5350 Operating Systems Internals and Design

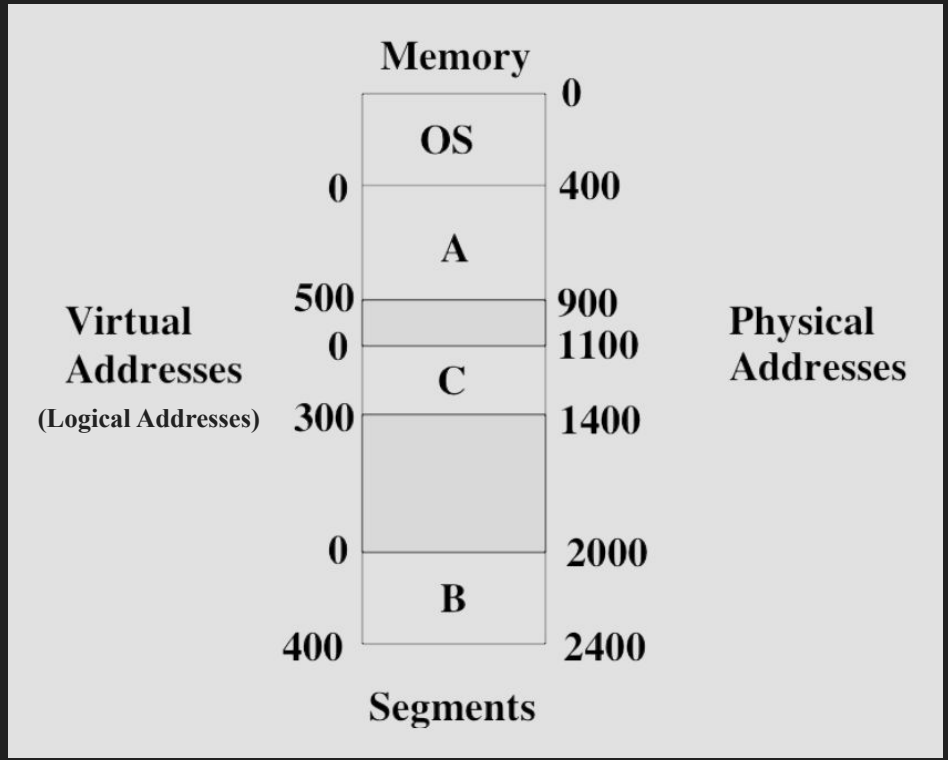
Max Gilson

Background

- All of our programs exist on our disk
- Before we can execute them they have to be brought into main memory (RAM) by the OS
- The CPU will fetch instructions from RAM to execute our program
- Your OS project does this already for your kernel
    - Before your kernel can be executed, it must be loaded off of the floppy disk

Memory Terminology

- Segment - a chunk of memory assigned to a process
- Physical address - a real address in memory that corresponds to an actual physical memory location
  - e.g. x000B8000 or xFFFF1234
- Virtual address - an address relative to the start of the process's address space
  - e.g. if a process exists within the range x1234 to x1FFF it will have virtual addresses from x000 to xDCB
  - This gives our programs more flexibility when accessing memory
  - Also called "logical address"
- Contiguous memory - memory that is contained within one region, one address after the other

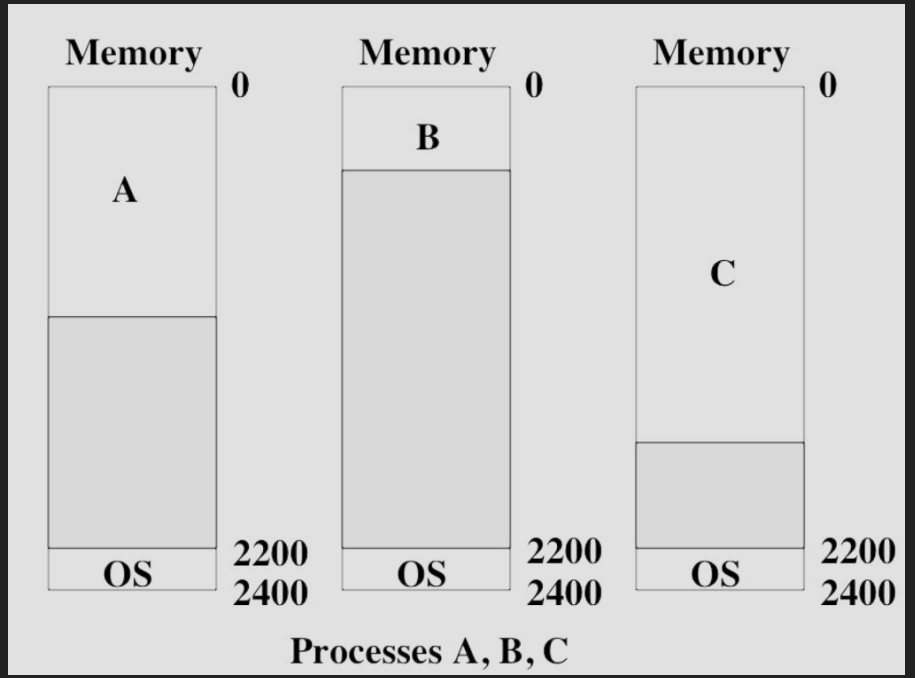


Where Do Addresses Come From?

- How do programs generate addresses for instructions and data?
- 3 different methods:
  - Compile time
    - The compiler decides where the program starts in memory at a fixed physical memory address
    - The OS does nothing
  - Load time
    - The compiler decides a starting address
    - The OS determines where this starting address gets placed in physical memory
    - Once loaded, the process does not move in memory
  - Execution time
    - The compiler decides a starting address
    - The OS determines where this starting address gets placed in physical memory
    - When the process is running, the OS will translate the virtual addresses to physical addresses

Uniprogramming

- Uniprogramming - running only 1 process + OS
- Assume the OS gets a fixed part of memory up to the highest address (DOS-like)
- Process is loaded at physical address x00000000
  - Process executes in a contiguous section of memory
- Maximum address = Memory Size - OS Size
- Simple but does not allow for overlap of I/O and CPU usage

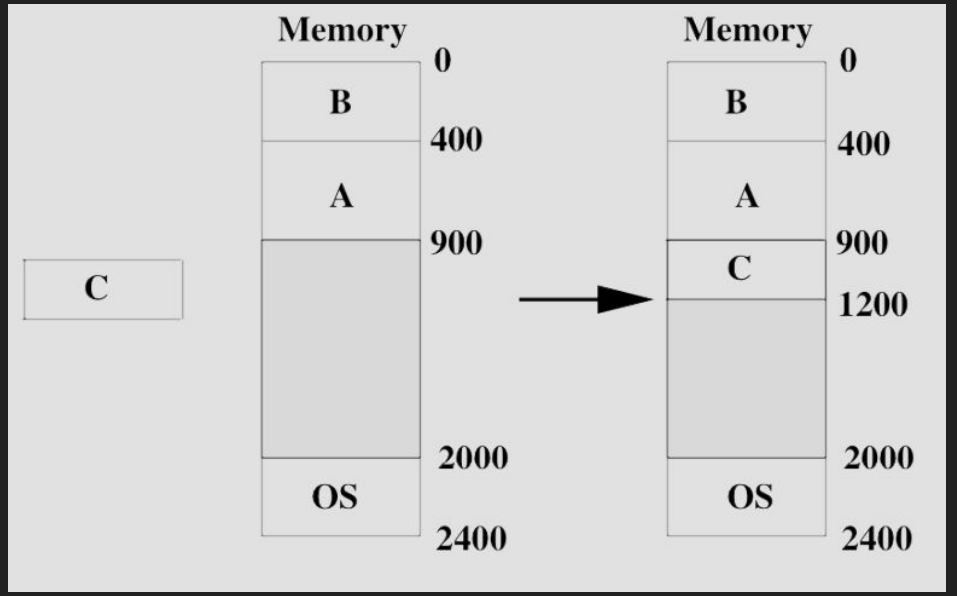


Requirements of Multiprogramming

- Transparency
  - We want multiple processes to coexist in physical memory
  - No process should be aware memory is shared
  - Processes should not care where they are in memory
- Safety
  - Processes must not corrupt each other or the OS
- Efficiency
  - Performance of CPU and memory should not be degraded

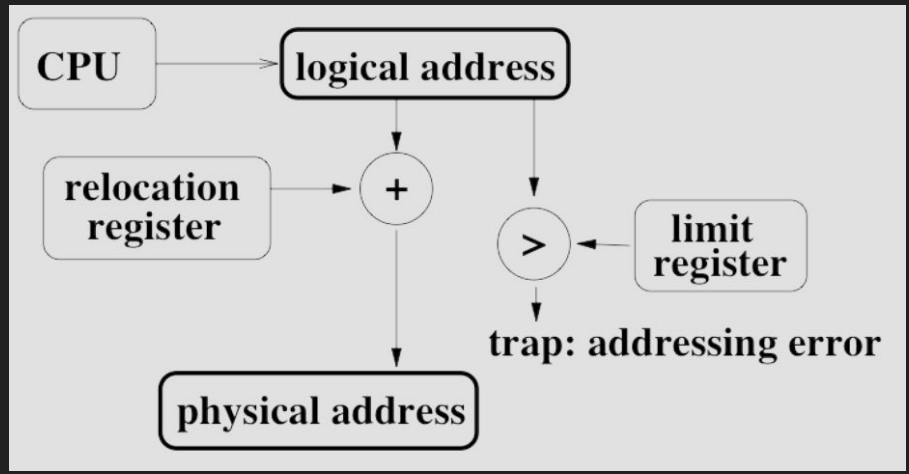
Relocation

- Relocation is moving a process to any location in memory
- Assume the OS gets a fixed part of memory up to the highest address (DOS-like)
- Assume at compile time the process starts at address 0 with: Maximum address = Memory Size - OS Size
- Base address
  - The first physical address of the process
- Limit address
  - The last physical address of the process



Types of Relocation

- Static Relocation
  - When the process is loaded, the OS offsets all addresses to reflect the process's new location in memory
  - Once the process is assigned to memory it cannot be moved
    - Moving would require re-offsetting all the addresses for every instruction in the program!
- Dynamic Relocation
  - Hardware has base register that gets added to virtual address, the result is the physical address
  - Hardware compares address with limit address
    - If the address exceeds the limit address, execute a trap service routine to handle addressing error and ignore physical address
  - Assume all logical addresses are positive



Benefits of Dynamic Relocation

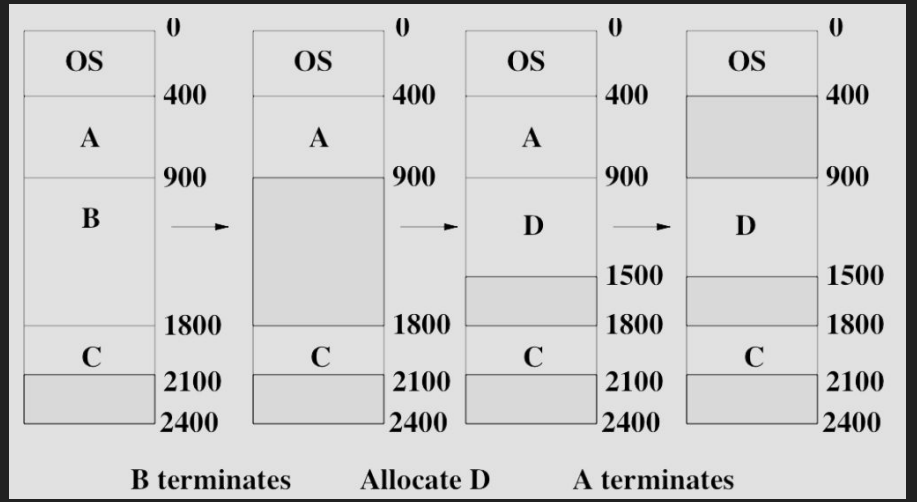
- Advantage:
  - OS can move a process in memory much easier
  - OS can allow a process to grow over time
  - Hardware requirements are simple
    - 2 extra registers, addition, and comparison
- Disadvantage:
  - Extra hardware may increase time to access memory
  - Sharing memory between processes is impossible
    - Each process is restricted to its segment
  - All running processes must coexist in physical memory
  - We are still using contiguous memory

Benefits of Dynamic Relocation (cont.)

- Transparency
  - Processes are unaware of other processes in memory
- Safety
  - Each memory access is checked by hardware
- Efficiency
  - Slightly slower but still very fast
  - Moving a process to a new location in memory is very slow
    - This may be necessary if a process grows

Memory Allocation

- As processes start, grow, and terminate, the OS tracks all used and unused memory
- When a new process starts, the OS must decide where the process goes into memory
  - A *hole* is a location in memory that is not filled
  - The OS tries to fill these holes with new processes
- What if we tried to allocate B again in the last diagram?
  - Hint: it won't fit

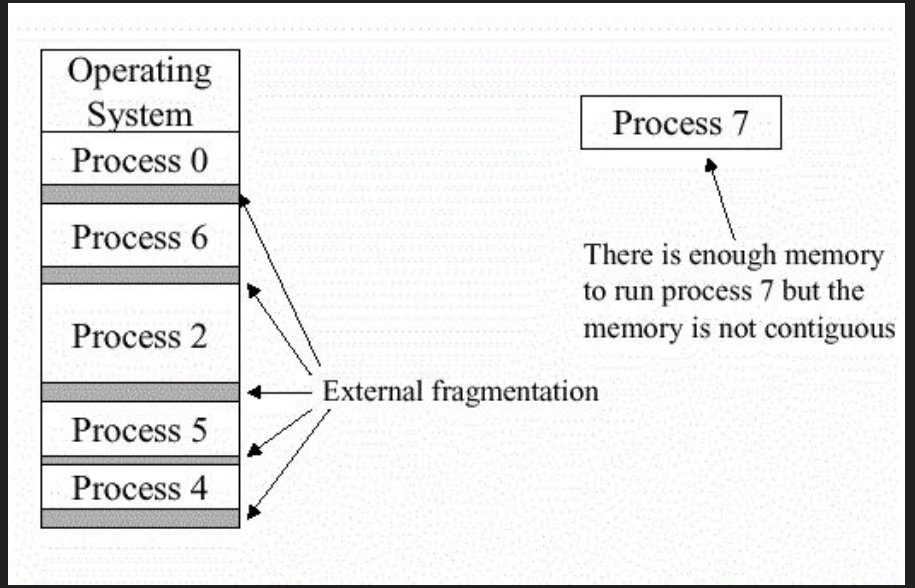


Memory Allocation Policies

- First Fit
  - Allocate the process to the first hole it fits in
  - Generally faster than Best Fit
- Best Fit
  - Allocate the process to the smallest hole that the process still fits in
  - Generally better storage utilization than Worst Fit
- Worst Fit
  - Allocate the process to the largest hole in memory
  - The remaining memory after the process might be large enough to fit another process or allow the process itself to grow
- For Best Fit and Worst Fit, the OS must search the entire list of holes to find the desired location to put the process
  - This can be slow if there are 1,000’s of holes!

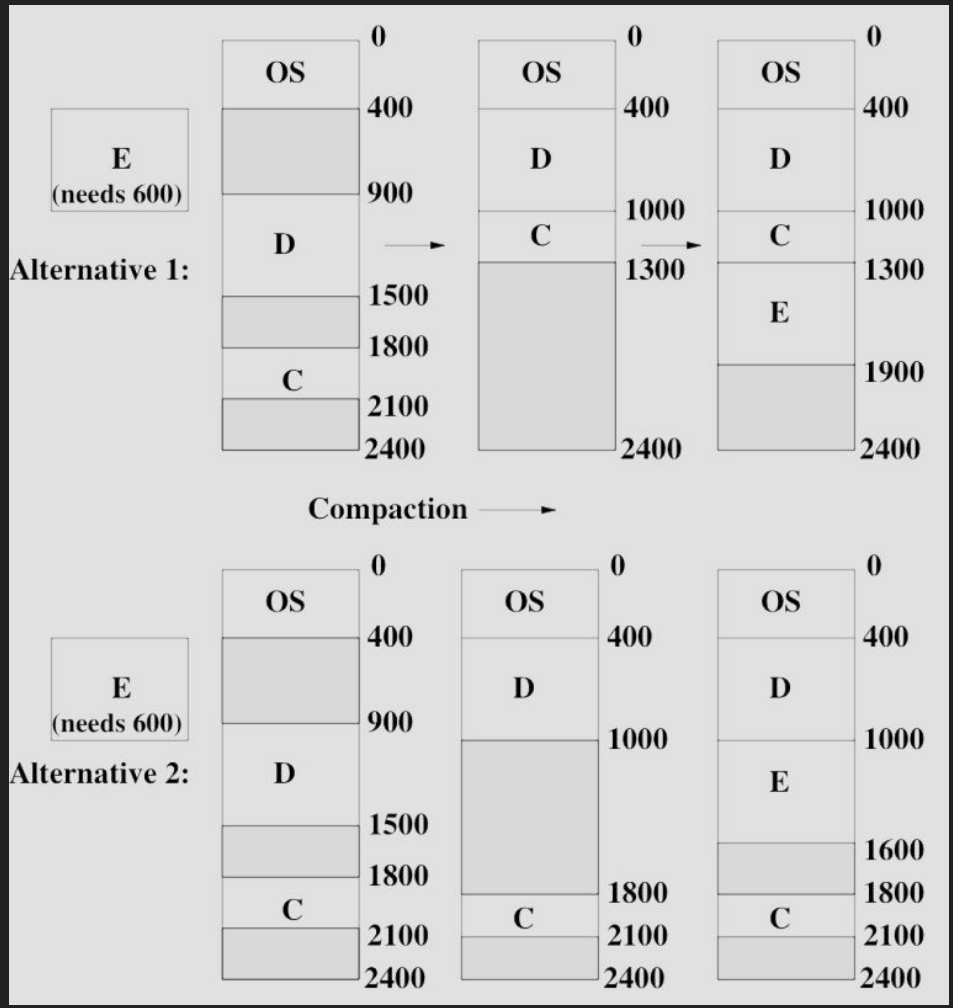
Fragmentation

- External Fragmentation
  - The memory between processes that is too small to be used
- Internal Fragmentation
  - Occurs if memory split into fixed sized chunks
  - If a process occupies one fixed sized chunk, but doesn't use it all



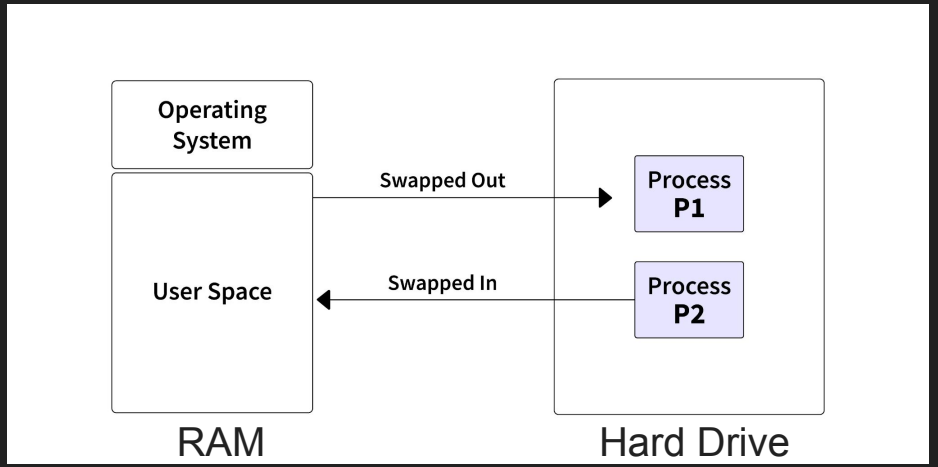
Compaction

- Compaction involves moving processes inside of memory to remove external fragmentation
- Advantage:
  - We can now load another process into memory
- Disadvantage:
  - Expensive operation, takes a lot of time
- Question:
  - We only need processes in memory if we're executing them. So, why do all 3 processes have to be in memory?



Swapping

- When we aren't executing a process, save it to the disk so we can resume it later
  - This frees up memory for other processes to use
- If the process becomes active again, reload it into memory
  - If using static relocation, process must go into same memory location
  - If using dynamic relocation, process can go anywhere in memory and OS must update relocation/limit registers
    - Compaction can be performed at this time
- Swapping processes takes a long time
  - If our scheduler wants to execute a process that is on disk, it should load that process while scheduling other processes first

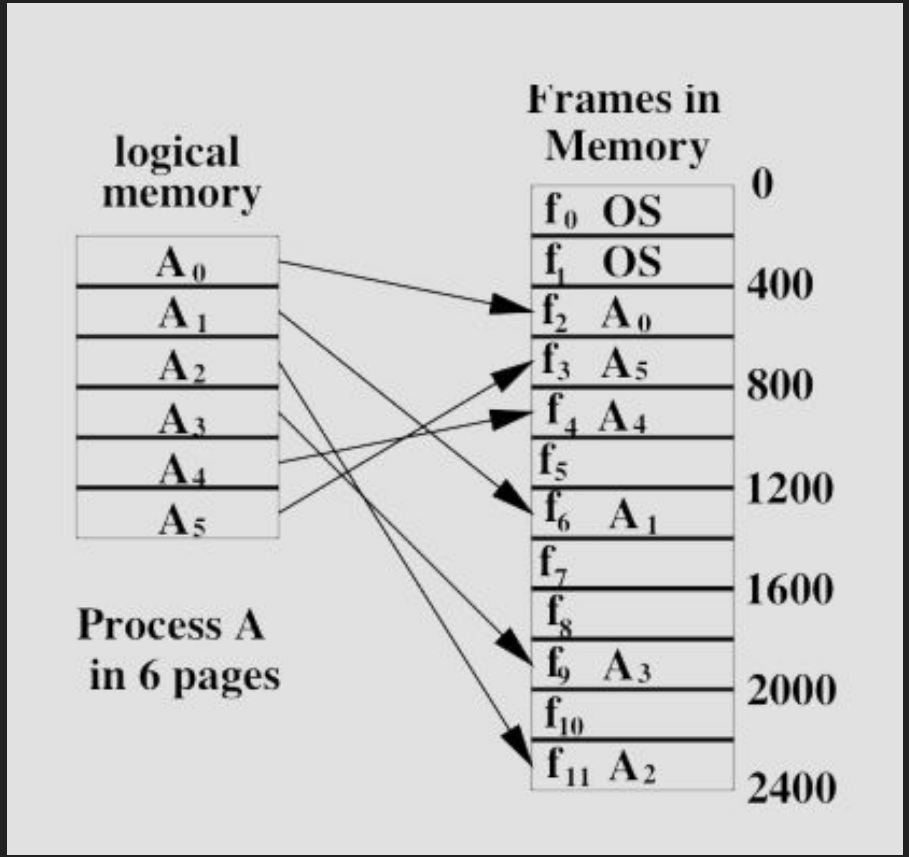


Paging

- Let’s assume that processes spend 90% of their time accessing 10% of their memory
  - Let’s only keep that 10% of their memory *in* memory unless the process needs more
- The logical memory of a process is contiguous
- The physical memory of a process is NOT contiguous
- Memory is divided into fixed size pages
  - This eliminates external fragmentation
  - Internal fragmentation still exists, about ½ a page is wasted per process
- Each process has its own table that translates pages to frames in memory
  - This table is managed by the OS

Paging (cont.)

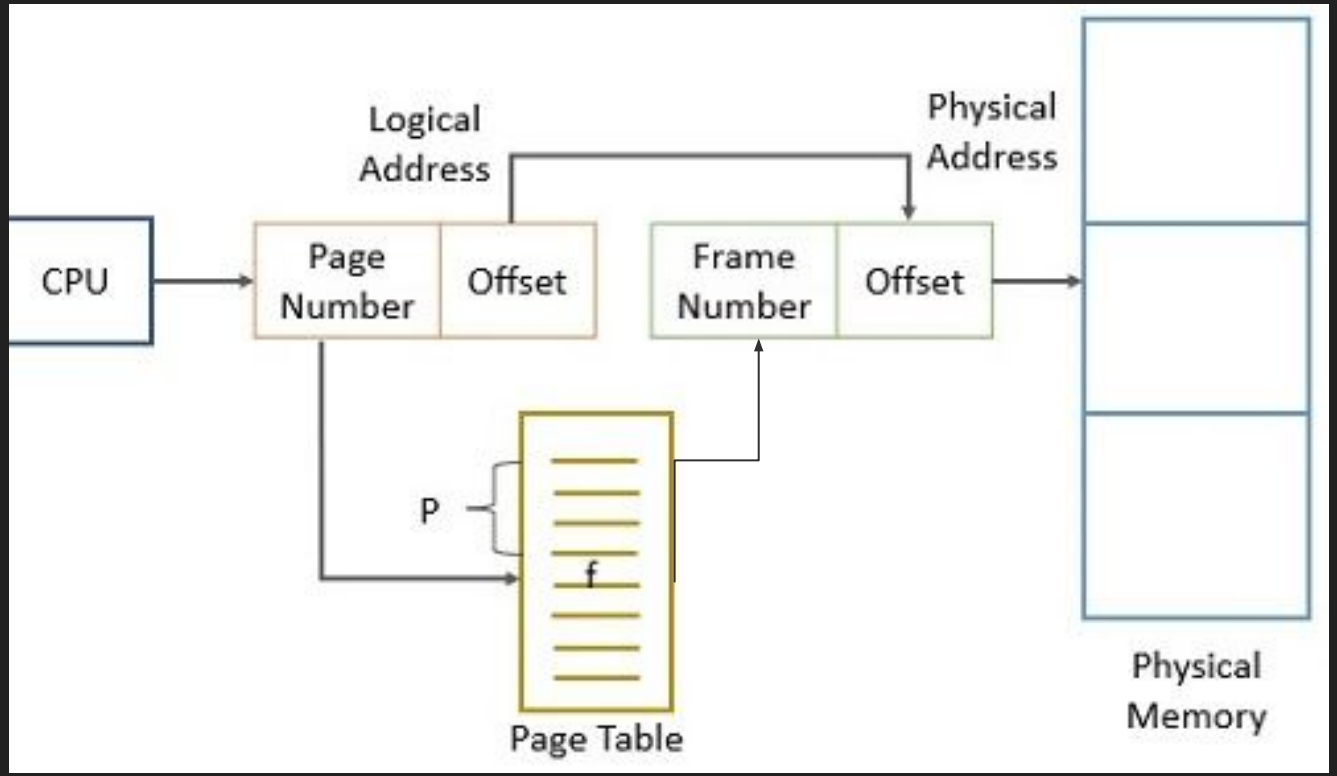
- The entirety of our process is divided up into fixed size pages
- These pages get mapped to frames in physical memory
- Pages can be moved, removed, added to frames in memory
- Our process should not care where its mapped to in physical memory
    - Our OS must do extra work keeping track of and maintaining processes page mapping
    - Our OS must translate logical addresses to physical addresses at runtime



Creating and Mapping Pages

- Processes refer to their memory locations with a virtual address
- Process generates contiguous virtual addresses from 0 to the size of the process
- The OS separates the process into pages and keeps track of their mapping to frames in the page table
- The paging hardware translates a virtual address (page number and page offset) into a physical address (frame number and frame offset)

Paging Hardware



Finding a Physical Address

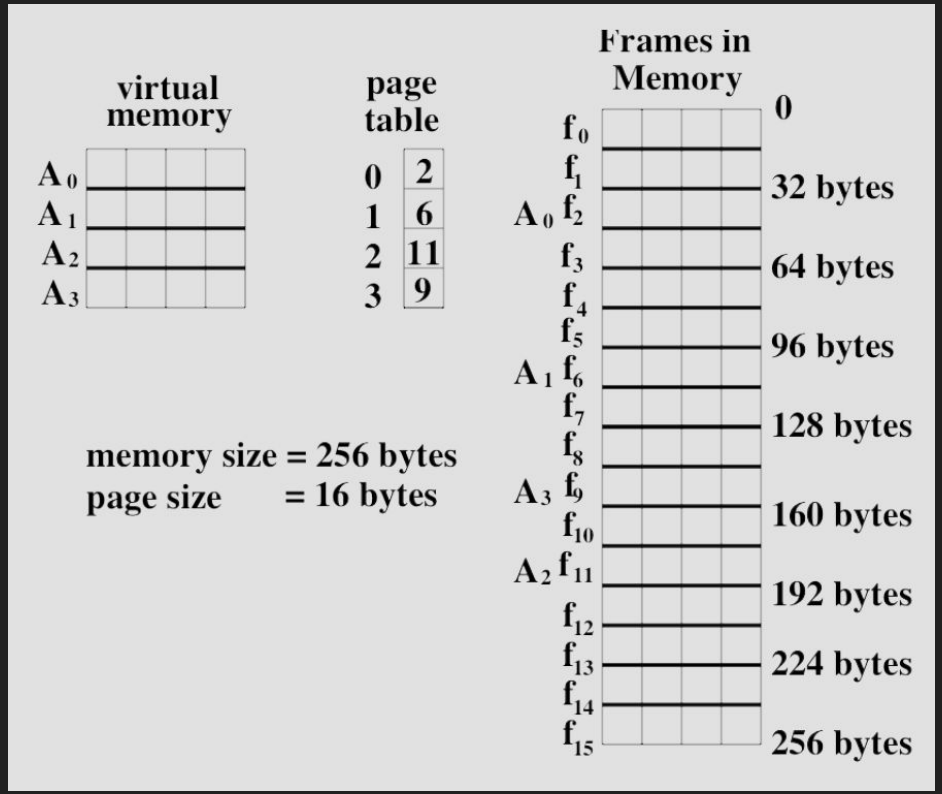
- Pages are typically 4096 bytes
- Assume 32 bit addresses with byte addressable memory
- Assume that the logical address requested by the CPU is:
  - x01A2C003
- Assume that there is an entry in the page table:
  - x01A2C : x20002
- What is the physical address that will be accessed?
  - x20002003

Paging Hardware

- Paging is a form of dynamic relocation, each virtual address is bound by the paging hardware to a physical address
- The page table acts as a set of relocation registers, one for each frame
- Mapping is invisible to the process, the OS does the mapping and the hardware does the translation
- Protections are provided with the same mechanisms used in dynamic relocation

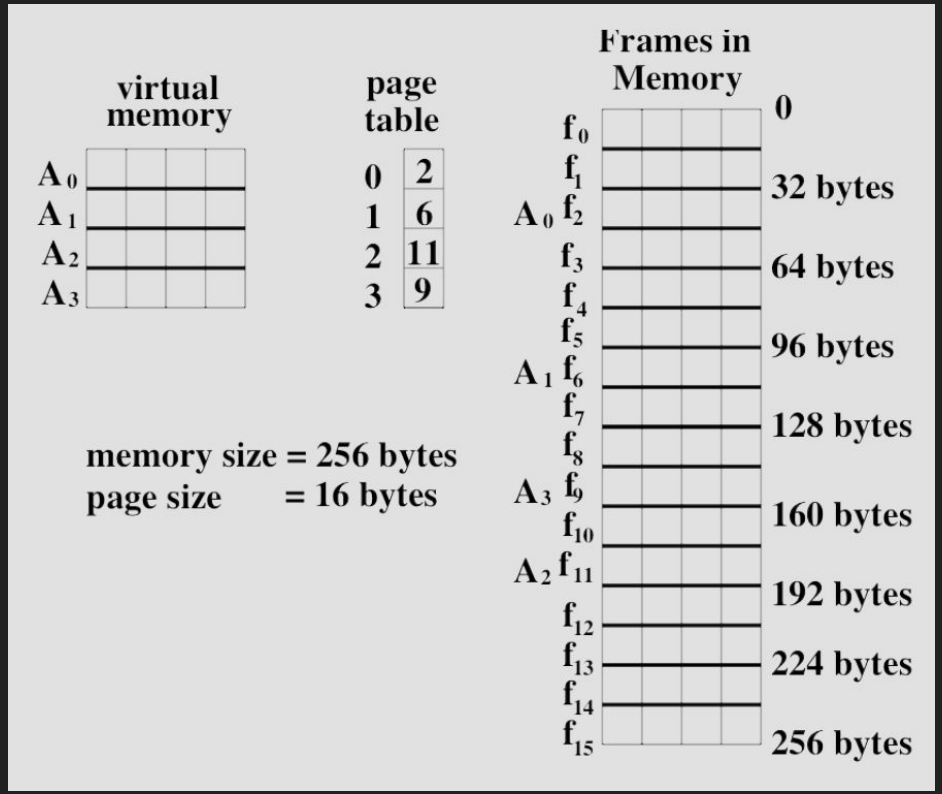
Address Translation Example 1

- Assume 256 bytes of total memory
- Assume byte addressable
- Assume page size of 16 bytes
- How big is the page table?
  - 16 entries = 256 total bytes / 16 bytes per page
- How many bits for a physical address?
  - 4 bits for frame (16 frames)
  - 4 bits for offset (16 bytes per frame)
  - 8 bits = 4 + 4 (256 addresses)
- How many bits for a virtual address?
  - 4 bits for 16 entries (page)
  - 4 bits for 16 bytes per page (offset)
  - 8 bits = 4 + 4
- Given a virtual address 0001 1000 in process A, what is the frame and offset of the physical address?
  - Page = 1, offset = 8
  - Frame = 6 (from page table), offset = 8
  - Physical address = 0110 1000



Address Translation Example 2

- Assume 256 bytes of total memory
- Assume word addressable with 4 byte word size
- Assume page size of 16 bytes
- How big is the page table?
  - 16 entries = 256 total bytes / 16 bytes per page
- How many bits for a physical address?
  - 4 bits for frame (16 frames)
  - 2 bits for offset (2 words per frame)
  - 6 bits = 4 + 2 (64 addresses) (64 = 256 / 4)
- How many bits for a virtual address?
  - 4 bits for 16 entries (page)
  - 2 bits for 4 words per page (offset)
  - 6 bits = 4 + 2
- Given a virtual address 0011 01 in process A, what is the frame and offset of the physical address?
  - Page = 3, offset = 1
  - Frame = 9 (from page table), offset = 1
  - Physical address = 1001 01



Where is The Page Table?

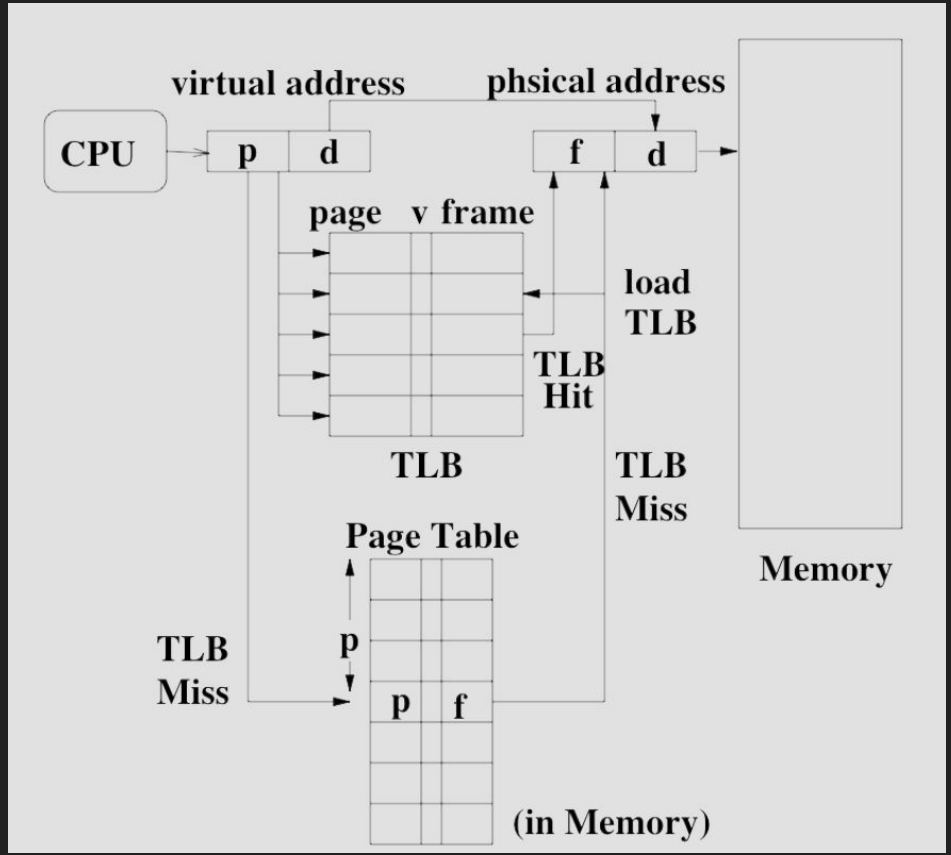
- Registers?
  - Very fast but we only have a few fixed number to work with
- Memory?
  - Slow but we probably have plenty of space
- Translation Look-Aside Buffer (TLB)
  - Works like a cache
  - Much larger than registers
  - Much faster than memory
  - Holds only a portion of our page table, the rest can exist in memory

Where is The Page Table?

- Registers?
  - Very fast but we only have a few fixed number to work with
- Memory?
  - Slow but we probably have plenty of space
- Translation Lookaside Buffer (TLB)
  - Works like a cache
  - Much larger than registers
  - Much faster than memory
  - Holds only a portion of our page table, the rest can exist in memory

TLB

- Have a small cache (TLB) that's going to store most of our page table
- TLB Hit:
  - If the page we requested exists in the TLB, translate the virtual address to a physical address
- TLB Miss:
  - If the page we requested does not exist in the TLB, go to memory to get it and put the missing page into the TLB
- Valid bit indicates if the TLB entry matches the page table entry
- A replacement algorithm must be implemented to replace entries in the TLB
  - Least recently used page can be replaced



Costs of Using the TLB

- How much time does it take to access memory if the page table is inside of memory?
  - T = total time, ma = memory access time
  - T = 2 * ma
  - We have to access the page table, then the physical memory, so 2 memory accesses are required
- How much additional time does the TLB add?
  - tlb = TLB access time, p = hit rate
  - T = (ma + tlb) * p + (2 * ma + tlb) * (1 - p)
- A larger TLB size decreases average memory access time

Initializing Memory With a New Process

1. Process arrives and needs k number of pages
2. If k frames are free, allocate the frames to pages, otherwise, free up frames that are no longer needed
3. OS puts each page inside each frame and adds the frame number to the page table
4. OS marks all TLB entries as invalid (flushes TLB)
5. OS starts executing process
6. OS loads TLB entries as each page is accessed, replacing existing entries if needed

The Process Control Block

- The PCB must be extended to contain
  - The entire process's page table
  - A copy of the TLB maybe
- During a context switch
  - Copy the page table base register value into the PCB for the process we are switching out
    - Page table base register (PTBR) points to the page table in memory for this process
  - Copy the TLB into the PCB (optional)
  - Flush the TLB
  - Restore the page table base register for process we're switching to
  - Restore the TLB if it was saved in the new process's PCB

Sharing Code Between Processes

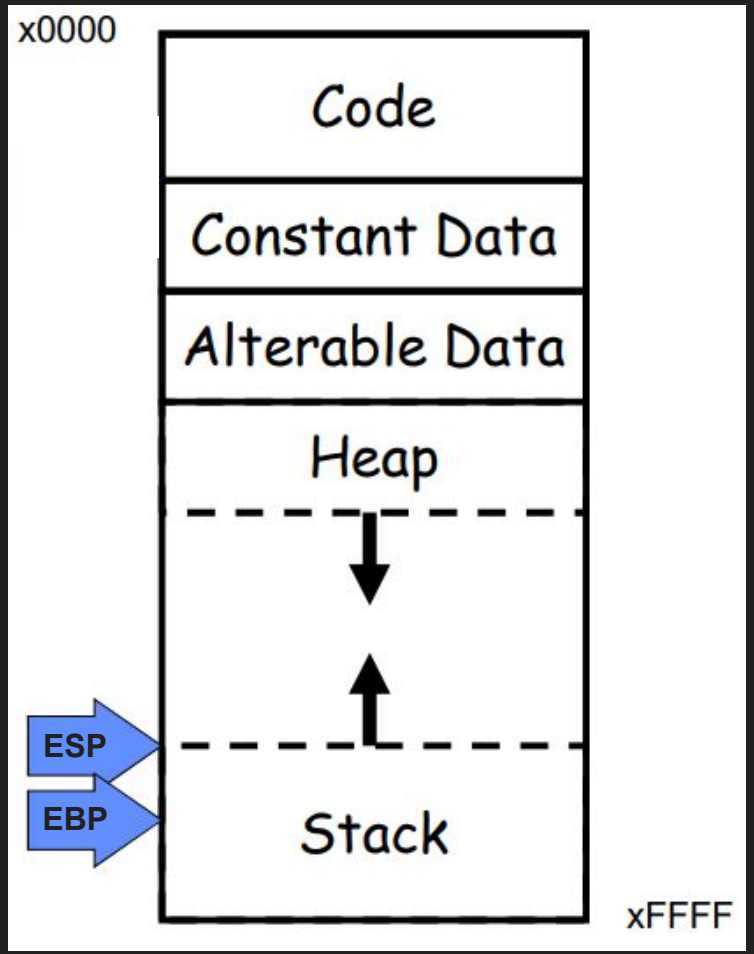
- Since a process's physical memory does not need to be contiguous, multiple processes could be allowed to access the same frame
- Any shared code must be reentrant, meaning processes should not be allowed to change it
  - Shared libraries (like standard C libraries) can easily be shared amongst processes and only one instance of the library is needed to be in memory

Paging Summary

- Advantages to paging:
  - Eliminates external fragmentation and compaction
  - Allows for sharing of code amongst processes
  - Processes can be partially loaded into memory
- Disadvantages to paging:
  - Translating a virtual address to a physical address takes more time
  - Requires hardware support with TLB
  - Requires more complex OS to maintain page table

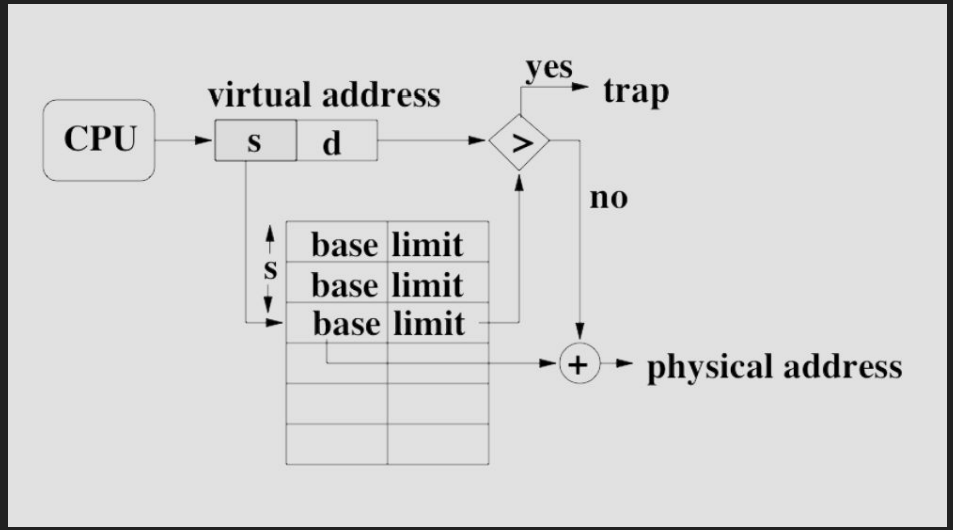
Segmentation

- Segmentation is another form of memory management
  - OSs may use segmentation instead of paging or they may use a combination of both
  - Windows and Linux use a combination of paging and segmentation
- Processes are made of different segments:
  - Code (instructions), variables (data), stack, heap
- Segmentation refers to these segments directly, instead of treating a process's memory like a linear array of bytes
  - The virtual address is a combination of a base register that identifies the segment and an offset inside that segment



Segmentation Hardware

- A segment table contains the base and limit registers of the segment
  - Additional information may be included to specify if a segment can be shared, read, written, etc.
- Typically, each process only has a few segments
  - x86 architecture only allows for 6 segments with the following registers: CS, SS, DS, ES, FS, GS
  - If a system uses many segments a TLB-like system may need to be implemented
- Segmentation does not eliminate external fragmentation
  - Segments must be contiguous and are of arbitrary size



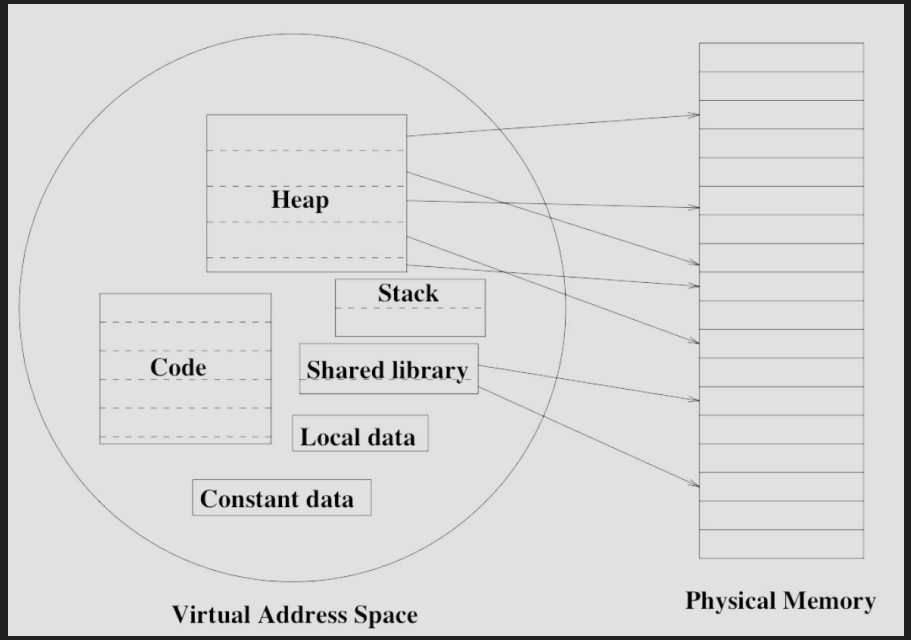
CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment
FS	General Purpose Segments
GS	

Segmentation Implementation

- At compile time, virtual addresses are generated where the most significant bit(s) are the segment number and the least significant bits are the offset into that segment
- Segmentation could be combined with dynamic or static relocation
  - Each segment is contiguous and could be assigned to any region of physical memory

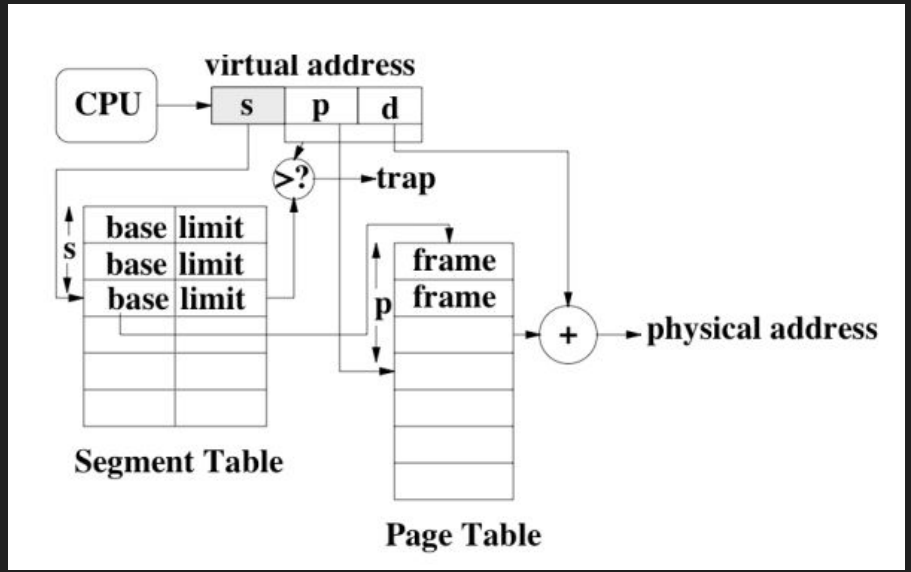
Segments and Pages

- Treat virtual address space as a collection of segments
- Treat physical memory as a sequence of fixed size frames
- Split the virtual segments into pages and map onto multiple frames



Translating Addresses in Segmented Paging

- The CPU uses a virtual address that contains a segment number, page number, and offset
- Both a segment table and page table are required
- Segment table can exist inside of registers or memory
    - Limits the maximum number of segments
- Page table can exist inside of memory and utilize a TLB
- Each base segment register points to a different page table that corresponds to the given segment



Segmented Paging Example 1

- Assume 256 bytes of total memory
- Assume byte addressable memory
- Assume page size of 32 bytes
- Assume maximum segments of 8
- How big is the page table?
  - 8 entries = 256 total bytes / 32 bytes per page
- How many bits for a physical address?
  - 3 bits for frame (8 frames)
  - 5 bits for offset (32 bytes per frame)
  - 8 bits (256 bytes)
- How many bits for a virtual address?
  - 3 bits (8 segments)
  - 3 bits (8 entries) (page)
  - 5 bits (32 bytes per page) (offset)
  - 11 bits = 3 + 3 + 5
- Given a virtual address 011 001 10101 in process A, what is the segment, frame, and offset of the physical address? Assume the segment points to the page table listed in the diagram.
  - Segment = 3
  - Page = 1, offset = 21
  - Frame = 2 (from page table), offset = 21
  - Physical address = 010 10101