Announcements/Reminders

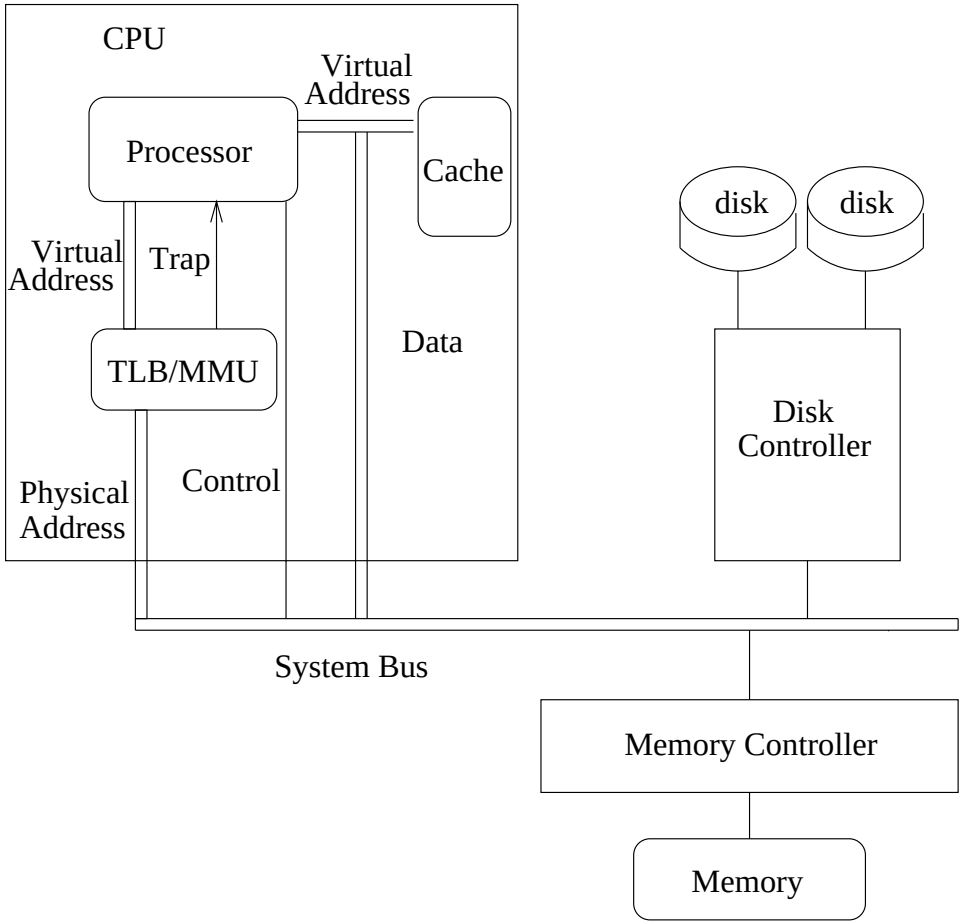
- HW 4 due Friday at 19:30
- Lab 3 due Wednesday, April 16

Memory Management

- 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?

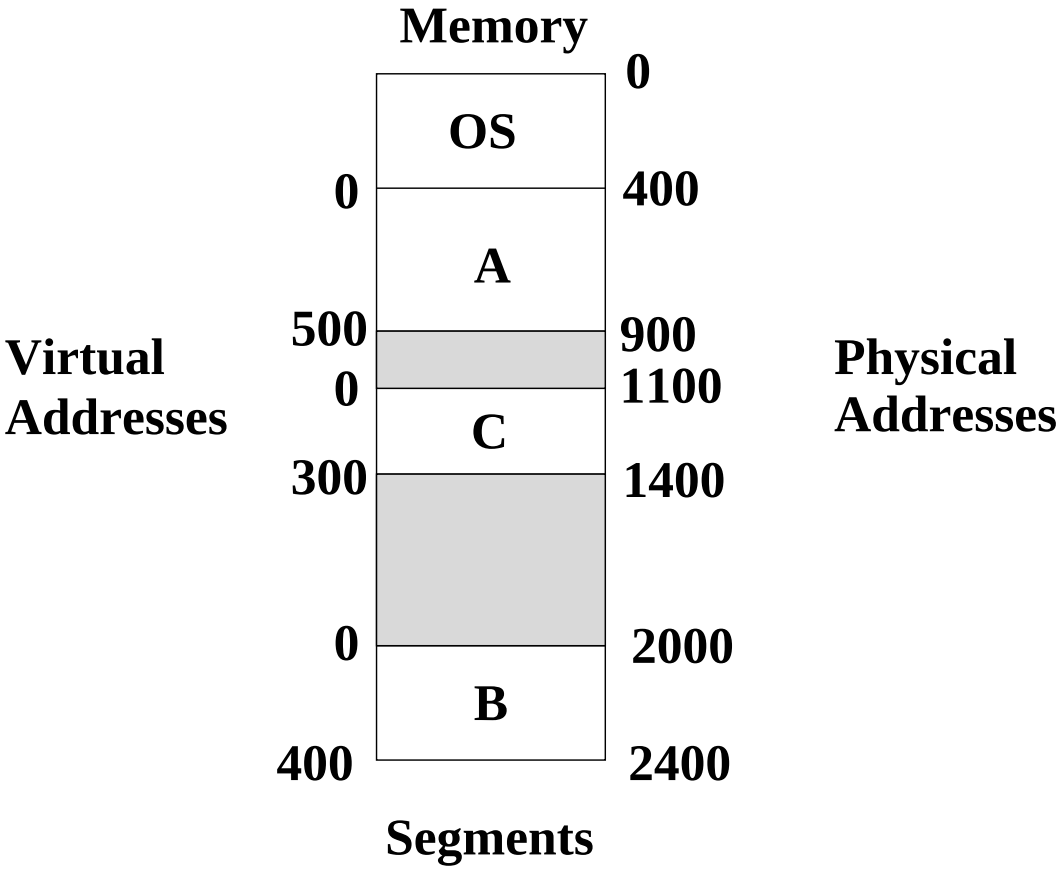
Background: Computer Architecture

- Program executable starts out on disk
- The OS loads the program into memory
- CPU fetches instructions and data from memory while executing the program



Memory Management: Terminology

- **Segment:** A chunk of memory assigned to a process.
- **Physical Address:** a real address in memory
- **Virtual Address:** an address relative to the start of a process's address space.



Where do addresses come from?

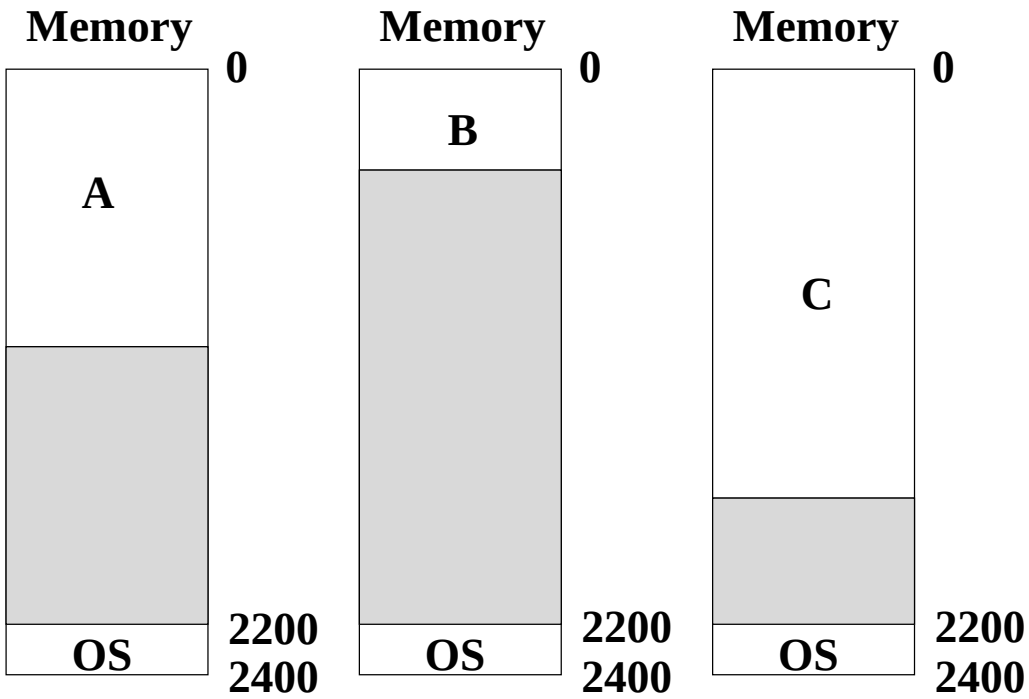
How do programs generate instruction and data addresses?

- **Compile time:** The compiler generates the exact physical location in memory starting from some fixed starting position k. The OS does nothing.
- **Load time:** Compiler generates an address, but at load time the OS determines the process' starting position. Once the process loads, it does not move in memory.
- **Execution time:** Compiler generates an address, and OS can place it any where it wants in memory.

Uniprogramming

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

Uniprogramming



⇒Simple, but does not allow for overlap of I/O and computation.

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 to which they are assigned.

Safety:

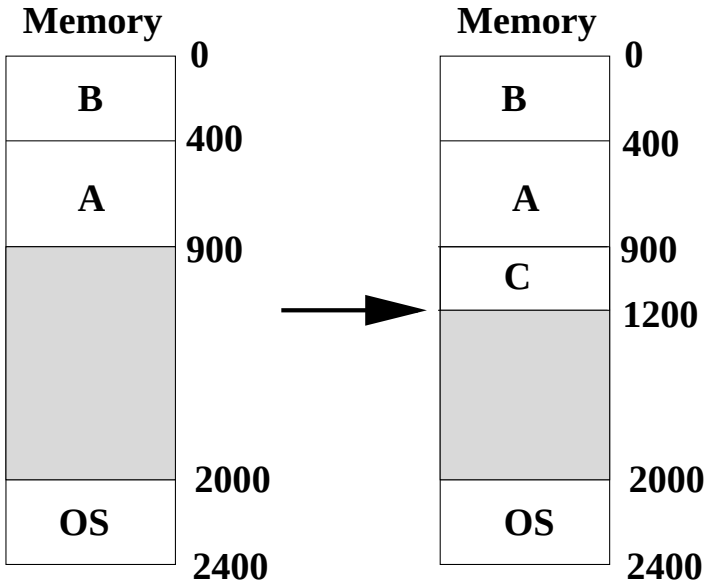
- Processes must not be able to corrupt each other.
- Processes must not be able to corrupt the OS.

Efficiency:

- Performance of 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.



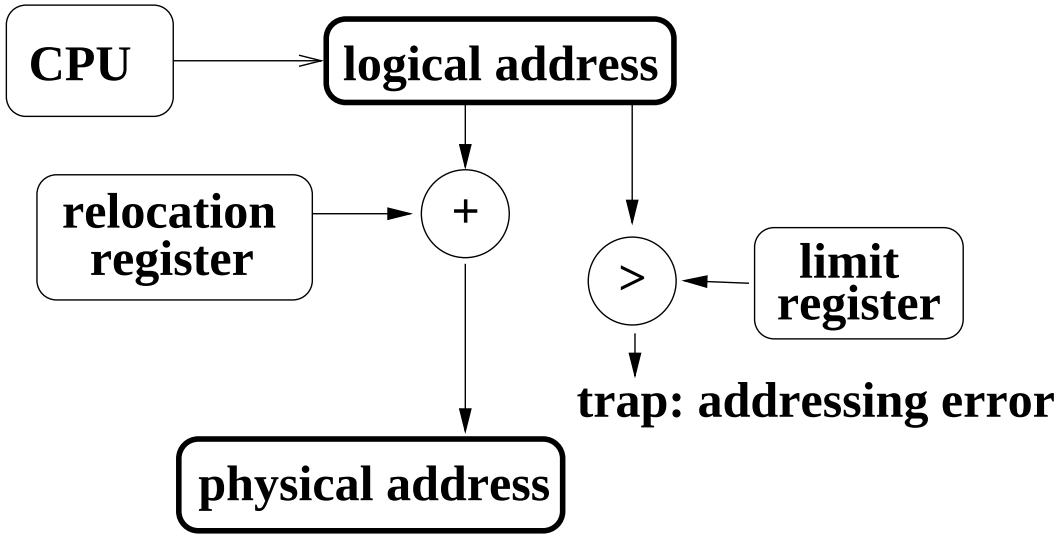
- Load a process by allocating a contiguous segment of memory in which the process fits.
- The first (smallest) physical address of the process is the *base* address and the largest physical address the process can access is the *limit* address.

Relocation

- **Static Relocation:**
  - At load time, the OS adjusts the addresses in a process to reflect its position in memory.
  - Once a process is assigned a place in memory and starts executing it, the OS cannot move it. (Why?)

Relocation

- **Dynamic Relocation:**
  - Hardware adds relocation register (base) to virtual address to get a physical address;
  - Hardware compares address with limit register (address must be less than limit).
  - If test fails, the MMU generates an address trap and ignores the physical address.



Relocation

- **Advantages:**
  - OS can easily move a process during execution.
  - OS can allow a process to grow over time.
  - Simple, fast hardware: two special registers, an add, and a compare.
- **Disadvantages:**
  -
  -
  -
  -
  -

Relocation

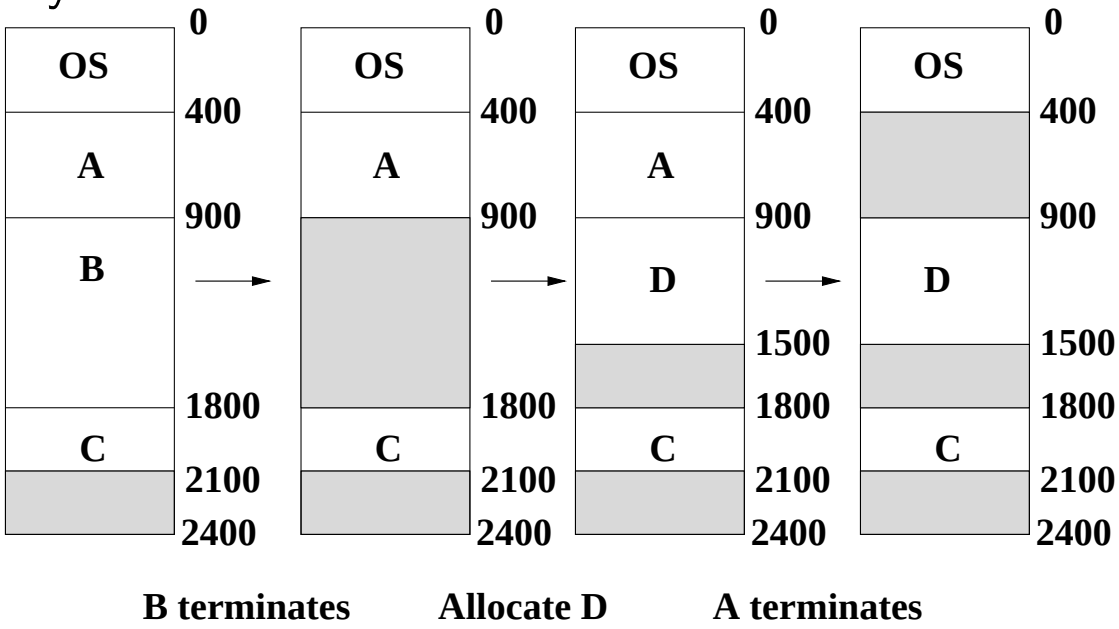
- **Advantages:**
  - OS can easily move a process during execution.
  - OS can allow a process to grow over time.
  - Simple, fast hardware: two special registers, an add, and a compare.
- **Disadvantages:**
  - Slows down hardware due to the add on every memory reference.
  - Can't share memory (such as program text) between processes.
  - Process is still limited to physical memory size.
  - Degree of multiprogramming is very limited since all memory of all active processes must fit in memory.
  - Complicates *memory management*.

Relocation: Properties

- **Transparency:** processes are largely unaware of sharing.
- **Safety:** each memory reference is checked.
- **Efficiency:** memory checks and virtual to physical address translation are fast as they are done in hardware, BUT if a process grows, it may have to be moved which is very slow.

Memory Management: Memory Allocation

As processes enter the system, grow, and terminate, the OS must keep track of which memory is available and utilized.



- **Holes:** pieces of free memory (shaded above in figure)
- Given a new process, the OS must decide which hole to use for the process

Memory Allocation Policies

- **First-Fit:** allocate the first one in the list in which the process fits. The search can start with the first hole, or where the previous first-fit search ended.
- **Best-Fit:** Allocate the smallest hole that is big enough to hold the process. The OS must search the entire list or store the list sorted by size hole list.
- **Worst-Fit:** Allocate the largest hole to the process (so as to leave as large a hole as possible). Again – the OS must search the entire list or keep the list sorted.
- Simulations show first-fit and best-fit usually yield better storage utilization than worst-fit; first-fit is generally faster than best-fit.

Fragmentation

- **External Fragmentation**
  - Frequent loading and unloading programs causes free space to be broken into little pieces
  - External fragmentation exists when there is enough memory to fit a process in memory, but the space is not contiguous
  - *50-percent rule:* Simulations show that for every $2N$ allocated blocks, N blocks are lost due to fragmentation (i.e., $1/3$ of memory space is wasted)
  - We want an allocation policy that minimizes wasted space.

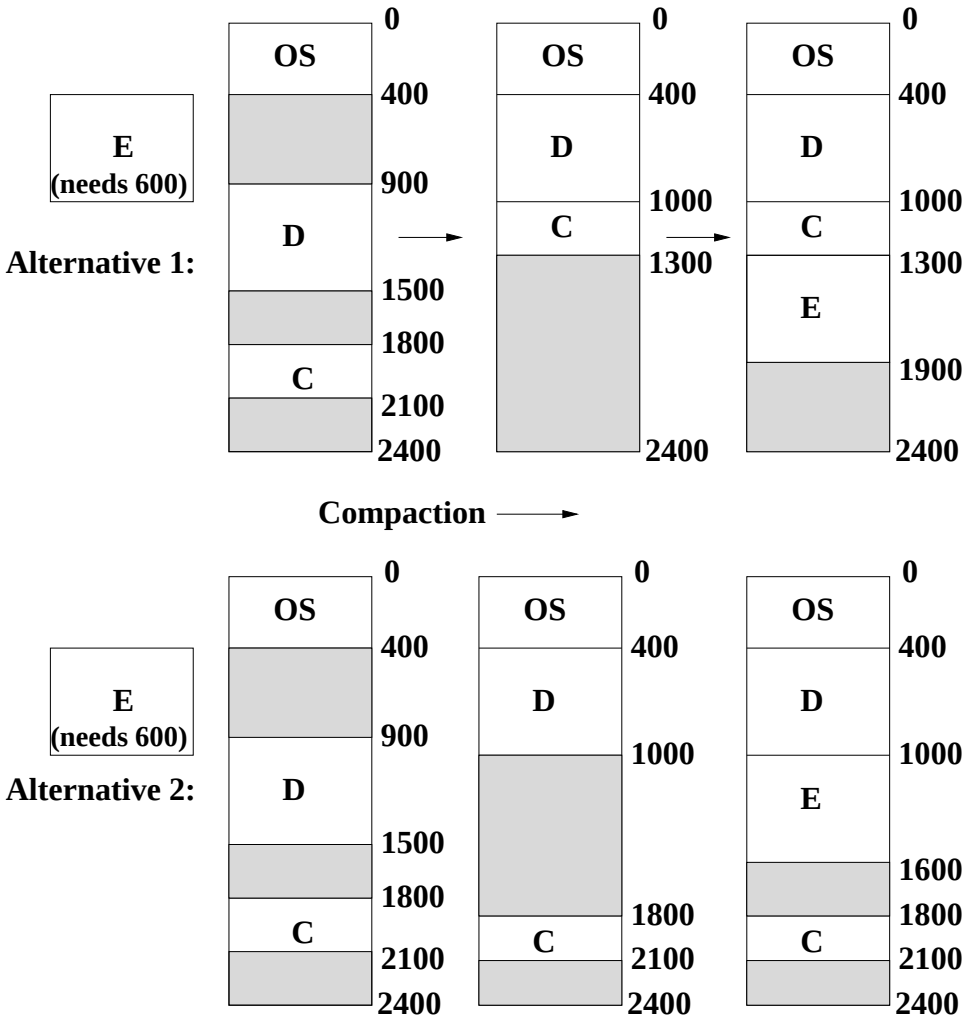
Fragmentation (cont)

- **Internal Fragmentation:**
  - Consider a process of size 8846 bytes and a block of size 8848 bytes

  $\Rightarrow$ it is more efficient to allocate the process the entire 8848 block than it is to keep track of 2 free bytes

  - Internal fragmentation exists when memory internal to a partition goes unused

Compaction



- How much memory is moved?
- How big a block is created?
- Any other choices?

Swapping

- Roll out a process to disk, releasing all the memory it holds.
- When process becomes active again, the OS must reload it in memory.
  - With static relocation, the process must be put in the same position.
  - With dynamic relocation, the OS finds a new position in memory for the process and updates the relocation and limit registers.
- If swapping is part of the system, compaction is easy to add.
- How could or should swapping interact with CPU scheduling?

Summary

- Processes must reside in memory in order to execute.
- Processes generally use virtual addresses which are translated into physical addresses just before accessing memory.
- Segmentation 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, but increases context switch time.

Next Time

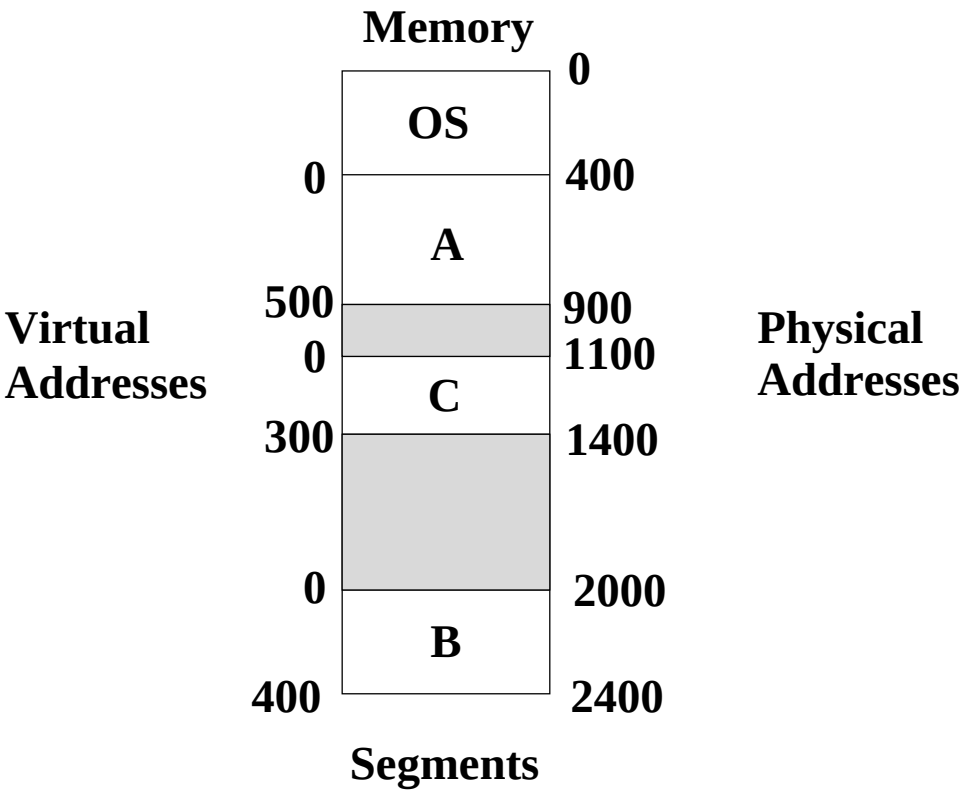
- Monday: Paging
- Wednesday: Paging

Announcements/Reminders

- Lab 3 due Wednesday
- Lab 3 demos are due no later than 1 week from Wednesday
- I am out of town next week (so this is the week to ask questions in preparation for the exam)

Last Class: Memory Management

- Uniprogramming
- Static Relocation
- Dynamic Relocation
- Monolithic memory segments



Next: Paging

Observation: Processes typically do not use their entire space in memory all the time.

Paging:

1. Divides and assigns processes to fixed sized *pages* (logical blocks of memory),

2. then selectively allocates pages to *frames* in the physical memory, and

3. manages (moves, removes, reallocates) pages in memory.

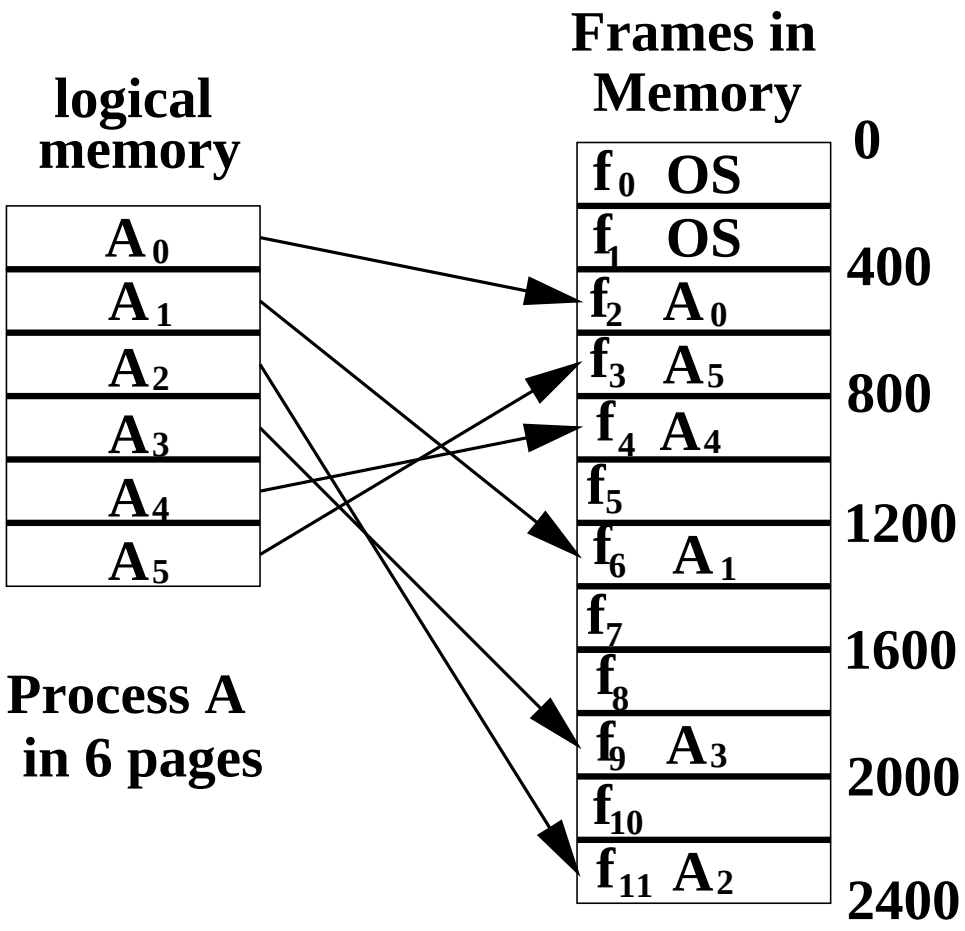
Paging: Motivation & Features

90/10 rule: Processes spend 90% of their time accessing 10% of their space in memory.

$\Rightarrow$ Keep only those parts of a process in memory that are actually being used

- Pages greatly simplify the hole fitting problem: all pages are interchangeable
- The logical memory of the process is contiguous, but pages need not be allocated contiguously in memory.
- By dividing memory into fixed size pages, we can eliminate external fragmentation.
- Paging does not eliminate internal fragmentation ($\sim 1/2$ page per process)

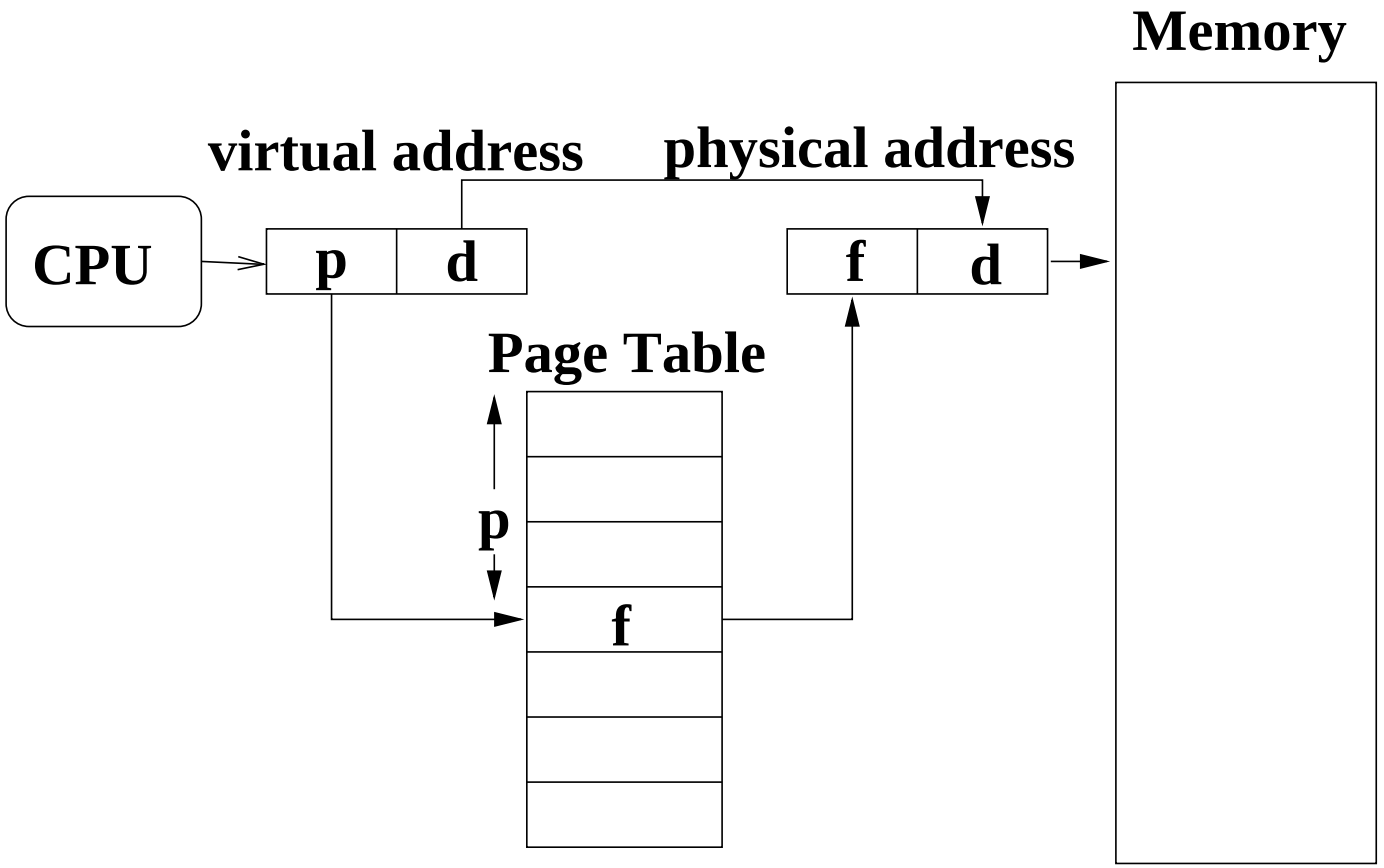
Paging: Example



Paging Hardware

- **Problem:** How do we find addresses when pages are not allocated contiguously in memory?
- **Virtual Address:**
  - Processes use a virtual (logical) address to name memory locations.
  - Process generates contiguous, virtual addresses from 0 to size of the process.
  - The OS lays the process down on pages and the paging hardware translates virtual addresses to actual physical addresses in memory.
  - In paging, the virtual address identifies the page and the page offset.
  - A *page table* keeps track of the frame in memory in which the page is located.

Paging Hardware



Paging Hardware

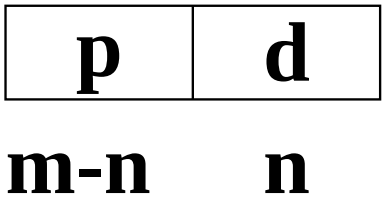
- Paging is a form of dynamic relocation, where each virtual address is bound by the paging hardware to a physical address.
- Think of the page table as a set of relocation registers, one for each frame.
- Mapping is invisible to the process; the OS maintains the mapping and the hardware does the translation.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

Paging Hardware: Practical Details

- Page size (frame sizes) are typically a power of 2 between 512 bytes and 8192 bytes per page.
- Powers of 2 make the translation of virtual addresses into physical addresses easier. Why?

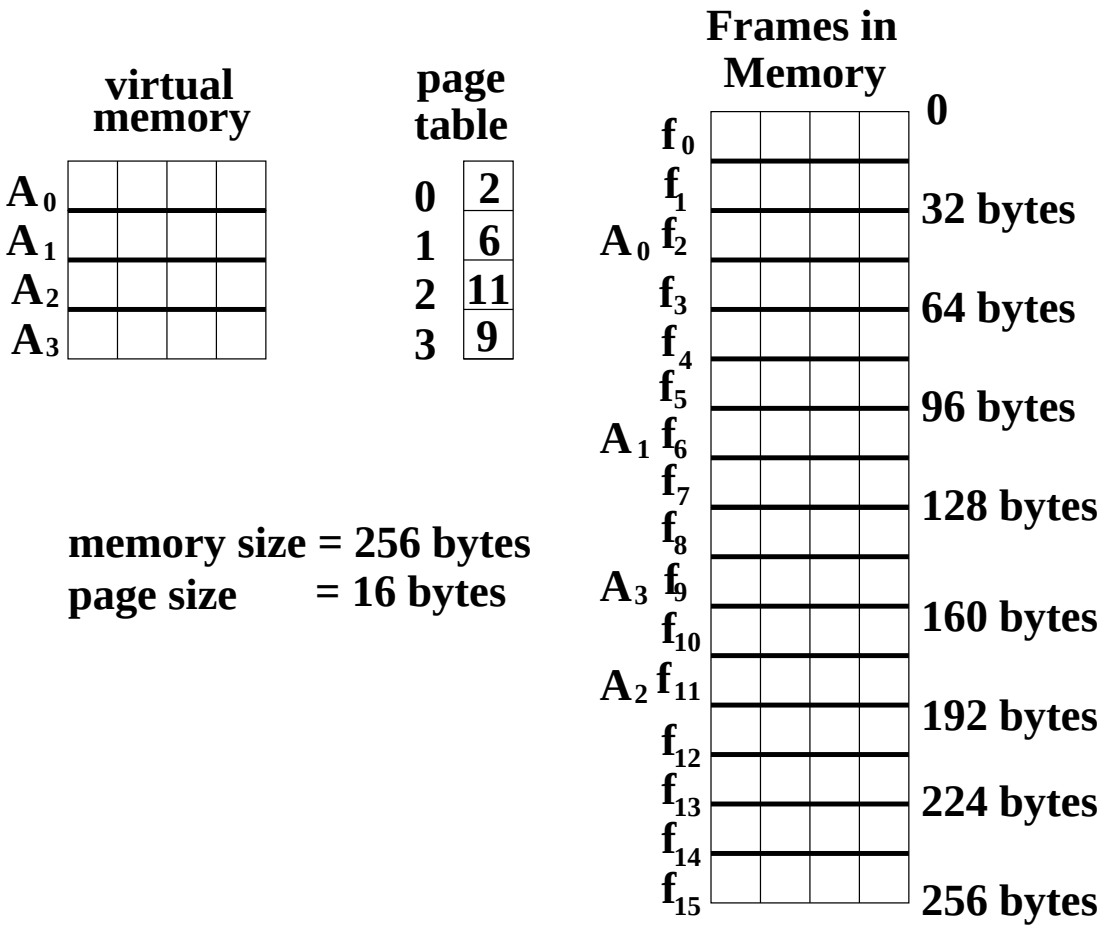
Paging Hardware: Practical Details

- Powers of 2 make the translation of virtual addresses into physical addresses easier. For example, given:
  1. virtual address space of size 2^m bytes and a page of size 2^n, then
  2. the high order $m - n$ bits of a virtual address select the page, and
  3. the low order n bits select the offset in the page



p: page number
d: page offset

Address Translation Example



Address Translation Example

- How big is the page table?
- How many bits for a physical address? Assume we can address in 1-byte increments.
- What part is p, and d?
- Given virtual address 24 ($0x18$), what is the virtual to physical translation?

Address Translation Example

- How big is the page table?

  4 entries

- How many bits for a physical address? Assume we can address in 1-byte increments.

  8 bits: 4 for page, 4 for offset

- What part is p, and d?

  p: most significant bits; d: least significant

- Given virtual address 24 ($0x18$), what is the virtual to physical translation?

  p = 1, d = 8 (virtual)

  f = 6, d = 8 (physical)

Address Translation Example

- How many bits for an address? Assume we can only address in 1-word (4 byte) increments?
- What part is p, and d?
- Given virtual address 13 ($0xD$), what is the virtual to physical translation?
- What needs to happen on a process context switch?

Address Translation Example

- How many bits for an address? Assume we can only address in 1-word (4 byte) increments?

  6 bits: 4 for page, 2 for offset

- What part is p, and d?

  (again): p is most significant, d is least.

- Given virtual address 13 ($0xD$), what is the virtual to physical translation?

  p = 3, d = 1 (virtual)

  f = 9, offset = 1 (physical)

Address Translation Example (cont)

- What needs to happen on a process context switch?

  Need to save page table in PCB, and then restore page table of the new process.

Making Paging Efficient

How should we store the page table?

- **Registers:**

  Advantages?

  Disadvantages?

- **Memory:**

  Advantages?

  Disadvantages?

Making Paging Efficient

How should we store the page table?

- **Registers:**

  Advantages? Fast.

  Disadvantages? If lots of pages, need many registers. Context switch would require saving/restoring registers which would be slow.

- **Memory:**

  Advantages? Lots of memory. Could just save/restore a pointer to the page table on context switch.

  Disadvantages? Each memory address requires 2 memory accesses: one to translate from virtual to physical memory, one to actually access memory.

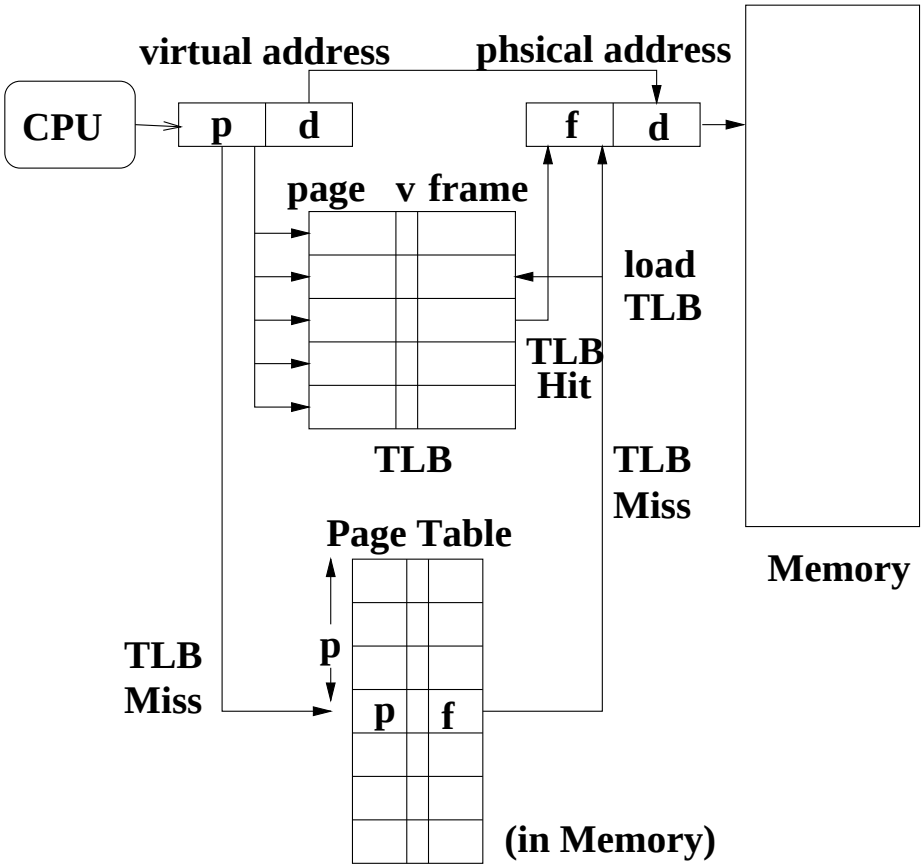
Translation Look-aside Buffers (TLB)

A fast, fully associative memory that stores page numbers (the key) and the frame (the value) in which they are stored.

- If memory accesses have locality, address translation has locality too.
- Typical TLB sizes range from 8 to 2048 entries.

TLB Implementation

v: valid bit that says the entry is up-to-date



Costs of Using The TLB

1. What is the effective memory access cost if the page table is in memory?

2. What is the effective memory access cost with a TLB?

Costs of Using The TLB

1. What is the effective memory access cost if the page table is in memory?

$$ema = 2 * ma$$

2. What is the effective memory access cost with a TLB?

$$ema = (ma + TLB) * p + (1 - p) * (2ma + TLB)$$

A large TLB improves hit ratio, and thus decreases average memory cost.

Initializing Memory when Starting a Process

1. Process needing k pages arrives.
2. If k page frames are free, then allocate these frames to pages. Else free frames that are no longer needed.
3. The OS puts each page in a frame and then puts the frame number in the corresponding entry in the page table.
4. OS marks all TLB entries as invalid (flushes the TLB).
5. OS starts process.
6. As process executes, OS loads TLB entries as each page is accessed, replacing an existing entry if the TLB is full.

Saving/Restoring Memory on a Context Switch

- The Process Control Block (PCB) must be extended to contain:
  - The page table
  - Possibly a copy of the TLB
- On a context switch:
  1. Copy the page table base register value to the PCB.
  2. Copy the TLB to the PCB (optional).
  3. Flush the TLB.
  4. Restore the page table base register.
  5. Restore the TLB if it was saved.
- **Multilevel Paging**: If the virtual address space is huge, page tables get too big. Many systems use a multilevel paging scheme...

Sharing

Paging allows sharing of memory across processes, since memory used by a process no longer needs to be contiguous.

- Shared code must be reentrant, that means the processes that are using it cannot change it (e.g., no data in reentrant code).
- Sharing of pages is similar to the way threads share text and memory with each other.
- A shared page may exist in different parts of the virtual address space of each process, but the virtual addresses map to the same physical address.
- The user program (e.g., emacs) marks text segment of a program as reentrant with a system call.

Sharing (cont)

- The OS keeps track of available reentrant code in memory and reuses them if a new process requests the same program.
- Can greatly reduce overall memory requirements for commonly used applications.

Summary

- Paging is a big improvement over segmentation:
  - Eliminates the problem of external fragmentation and therefore the need for compaction.
  - Allows sharing of code pages among processes, reducing overall memory requirements.
  - Enables processes to run when they are only partially loaded in main memory.
- However, paging has its costs:
  - Translating from a virtual address to a physical address is more time-consuming.
  - Paging requires hardware support in the form of a TLB to be efficient enough.
  - Paging requires more complex OS to maintain the page table.

Next Time

- Segmented Paging
- Next Week:
  - Monday: Holiday
  - Wednesday: Discussion and exam review
  - Friday: Exam 2 (same place, same time)

Announcements/Reminders

- Lab 3 due today
- Exam 2 is 1 week from Friday
- Lab 4, HW 4 solutions and HW 5 are available from the web page

Quiz

Who Said: "Operating Systems are Destined to be Free"?

What did he mean?

What was he arguing for?

Quiz

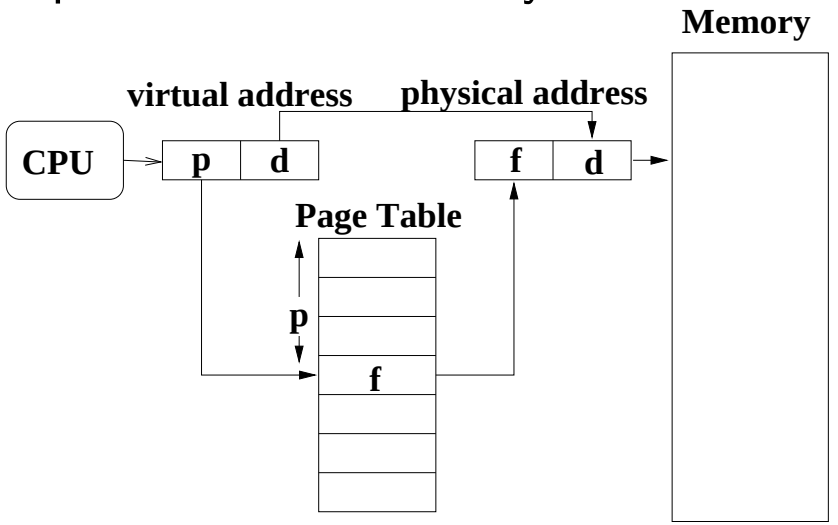
Who Said: "Operating Systems are Destined to be Free"?

Neal Stephenson in: "In the Beginning was the Command Line."

also wrote: Snow Crash and Cryptonomicon

So Far: Paging

- Process generates virtual addresses from 0 to Max.
- OS divides the process onto pages; manages a page table for every process; and manages the pages in memory.
- Hardware maps from virtual addresses to physical addresses.
- Sharing of pages is possible, but not very useful as implemented. Why?



Today: Segmentation

Segments take the user's view of the program.

- User views the program in logical *segments*, e.g., code, global variables, stack, heap (dynamic data structures), not a single linear array of bytes.
- New idea: the compiler generates references that identify the segment and the offset in the segment, e.g., a code segment with offset = 399
- Thus processes use virtual addresses that include both the segment and segment offset.

$\Rightarrow$ Segments make it easier for the call stack and heap to grow dynamically. Why?

$\Rightarrow$ Segments make both sharing and protection easier. Why?

Today: Segmentation

⇒ Segments make it easier for the call stack and heap to grow dynamically. Why?

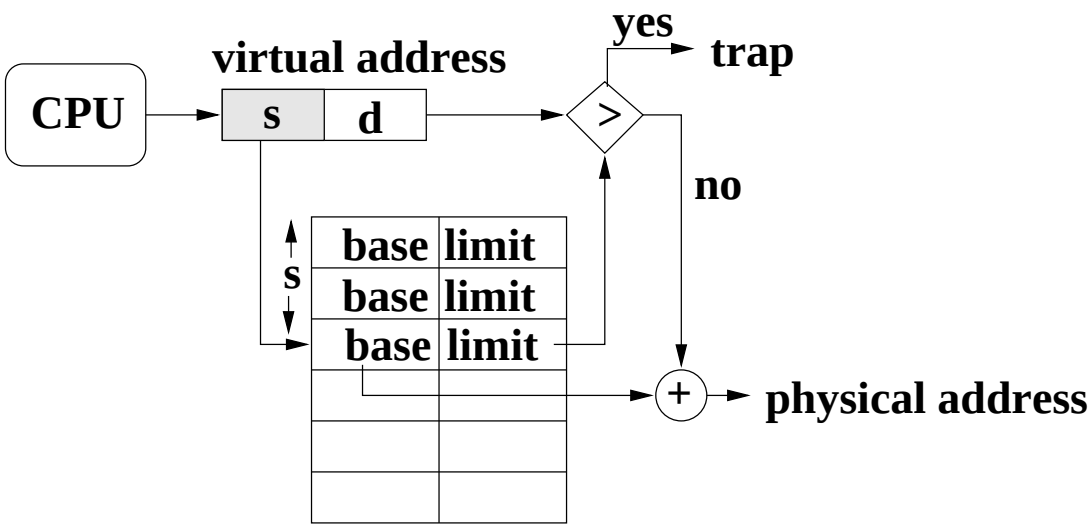
- Without segmentation, the stack and the heap grow toward each other.
- With segmentation, each segment is managed separately, so if the stack or the heap grows, the OS can increase the segment size.

⇒ Segments make both sharing and protection easier. Why?

- The OS manages these logical units separately (instead of managing a fixed-size page).

Implementing Segmentation

- Segment table: each entry contains a base address in memory, length of segment, and protection information (can this segment be shared, read, modified, etc.).
- Hardware support: multiple base/limit registers.
- How is this different than a TLB?



Implementing Segmentation

- Compiler needs to generate virtual addresses whose higher-order bits are a segment number.
- Segmentation can be combined with a dynamic or static relocation system,
  - Each segment is allocated a contiguous piece of physical memory.
  - External fragmentation can be a problem again
- Similar memory mapping algorithm as paging. We need something like the TLB if programs can have lots of segments

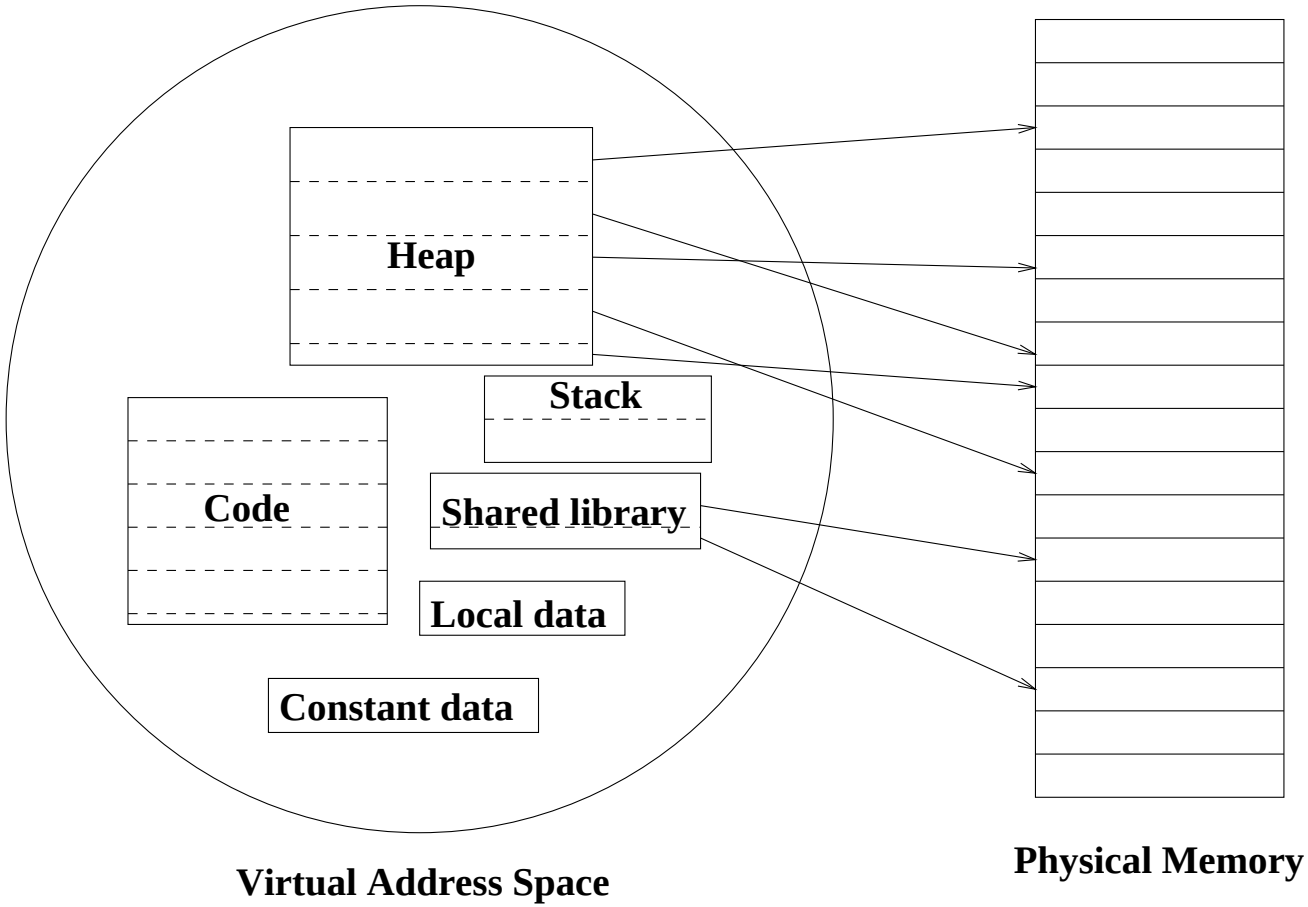
Let's combine the ease of sharing we get from segments with efficient memory utilization we get from pages.

Combining Segments and Paging

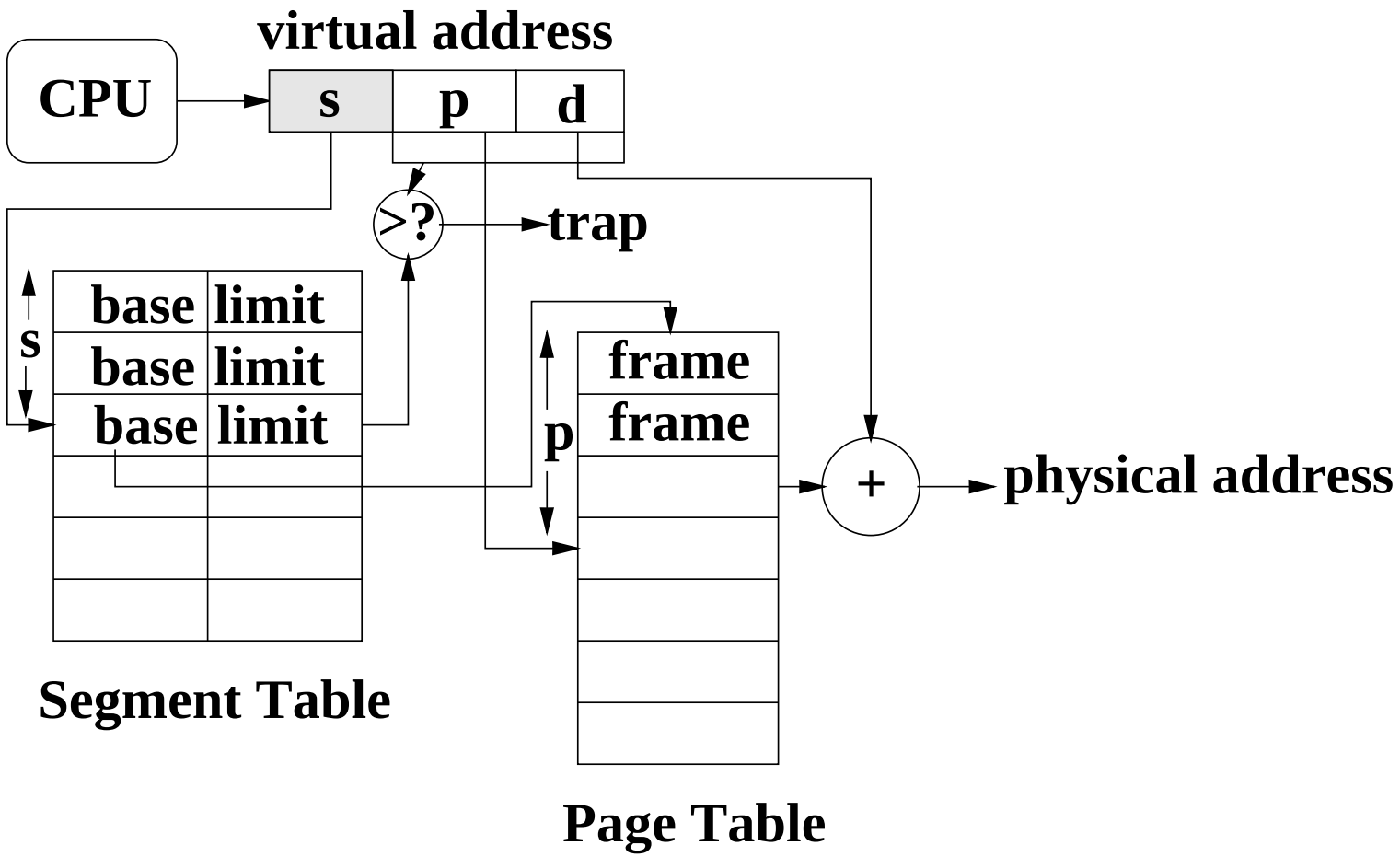
- Treat virtual address space as a collection of segments (logical units) of arbitrary sizes.
- Treat physical memory as a sequence of fixed size page frames.
- Segments are typically larger than page frames,

  $\Rightarrow$ Map a logical segment onto multiple page frames by paging the segments

Combining Segments and Paging



Addresses in a Segmented Paging System



Addresses in a Segmented Paging System

- A virtual address becomes a segment number, a page within that segment, and an offset within the page.
- The segment number indexes into the segment table which yields the base address of the page table for that segment.
- Check the remainder of the address (page number and offset) against the limit of the segment.
- Use the page number to index the page table. The entry is the frame. (The rest of this is just like paging.)
- Add the frame and the offset to get the physical address.

Addresses in a Segmented Paging System: Example

- Given a memory size of 256 addressable words,
- a page table indexing 8 pages,
- a page size of 32 words, and
- 8 logical segments

1. How many bits is a physical address?
2. How many bits for the segment number, page table, and offset?
3. How many bits is a virtual address?
4. How many segment table entries do we need?

Addresses in a Segmented Paging System: Example

- Given a memory size of 256 addressable words,
- a page table indexing 8 pages,
- a page size of 32 words, and
- 8 logical segments

1. How many bits is a physical address? 8 bits
2. How many bits for the segment number, page table, and offset? 3 3 5
3. How many bits is a virtual address? 11 bits
4. How many segment table entries do we need? 8

Sharing Pages and Segments

- Share individual pages by copying page table entries.
- Share whole segments by sharing segment table entries, which is the same as sharing the page table for that segment.
- Need protection bits to specify and enforce read/write permission.
  - When would segments containing code be shared?
  - When would segments containing data be shared?

Sharing Pages and Segments

- When would segments containing code be shared?
  - Read-only sharing of common applications or common libraries.
- When would segments containing data be shared?
  - Read-only sharing of constant data used by an application.
  - Read-write sharing of heap data used by multiple threads within a process.

Sharing Pages and Segments: Implementation Issues

- Where are the segment table and page tables stored?
  1. Store segment tables in a small number of associative registers; page tables are in main memory with a TLB
     (faster but limits the number of segments a program can have)
  2. Both the segment tables and page tables can be in main memory with the segment index and page index combined & used in the TLB lookup
     (slower but no restrictions on the number of segments per program)

- Protection and valid bits can go either on the segment or the page table entries

- **Note:** Just like recursion, we can do multiple levels of paging and segmentation when the tables get too big.

Segmented Paging: Costs and Benefits

- **Benefits:** faster process start times, faster process growth, memory sharing between processes.
- **Costs:** somewhat slower context switches, slower address translation.
- Pure paging system $\Rightarrow$(virtual address space)/(page size) entries in page table. How many entries in a segmented paging system?
- What is the performance of address translation of segmented paging compared to contiguous allocation with relocation? Compared to pure paging?
- How does fragmentation of segmented paging compare with contiguous allocation? With pure paging?

Segmented Paging: Costs and Benefits

- How many entries in a segmented paging system?

  There are as many page tables as there are segments. The total number of page table entries for a process is <= the number required in a pure paging scheme.

- What is the performance of address translation of segmented paging compared to contiguous allocation with relocation? Compared to pure paging?
  - Slower than contiguous allocation since need to index through segment table AND page table. If TLB miss, much slower.
  - Slower than pure paging because index through segment table. But: normally done in hardware, so not a big problem

Segmented Paging: Costs and Benefits

- How does fragmentation of segmented paging compare with contiguous allocation? With pure paging?
  - No external fragmentation, so no need for compaction.
  - Internal fragmentation: 1/2 page per segment on average, so worse than pure paging.

Putting it All Together: A Historical View

- **Relocation** using Base and Limit registers
  - simple, but inflexible
- **Segmentation:**
  - compiler's view presented to OS
  - segment tables tend to be small
  - memory allocation is expensive and complicated (first fit, worst fit, best fit).
  - compaction is needed to resolve external fragmentation.

Putting it all together

- **Paging:**
  - simplifies memory allocation since any page can be allocated to any frame
  - page tables can be very large (especially when virtual address space is large and pages are small)
- **Segmentation & Paging**
  - only need to allocate as many page table entries as we need (large virtual address spaces are not a problem).
  - easy memory allocation, any frame can be used
  - sharing at either the page or segment level
  - increased internal fragmentation over paging
  - two lookups per memory reference

Next Time

- Friday: Exam Review and start Virtual Memory
- Wednesday: Discussion (hw 3/4 and Memory Management)
- Next Friday: Exam 2
- Following Monday: Virtual Memory and Demand Paging (Chapter 10)