Last Class: Deadlocks

- Necessary conditions for deadlock:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Ways of handling deadlock
  - Deadlock detection and recovery
  - Deadlock prevention
  - Deadlock avoidance - Banker's algorithm

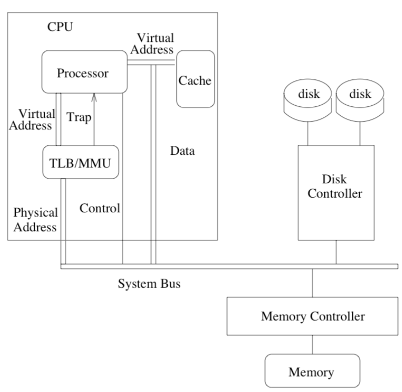
Where we are in the course

- Discussed:
  - Processes & Threads
  - CPU Scheduling
  - Synchronization & Deadlock
- Next up:
  - Memory Management
- Yet to come:
  - File Systems and I/O Storage
  - Distributed Systems

Memory Management

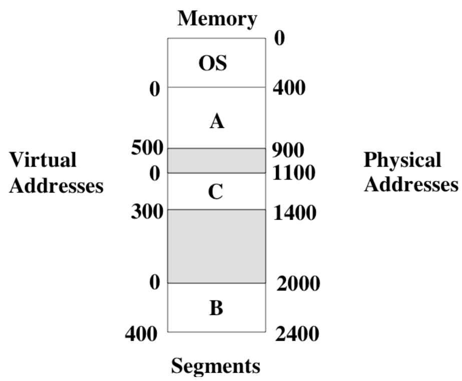
- Where is the executing process?
- How do we allow multiple processes to use main memory simultaneously?
- What is an address and how is one interpreted?

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?

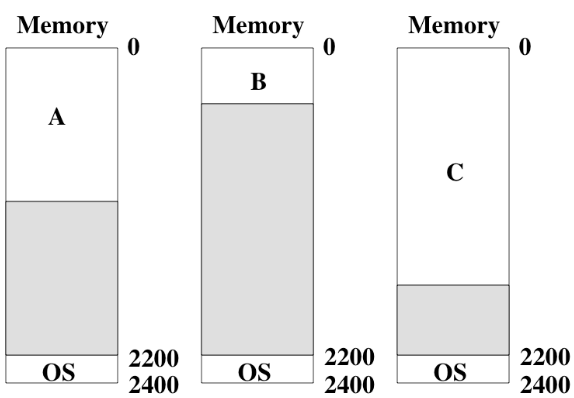
How do programs generate instruction and data addresses?

- **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



Processes A, B, C

⇒ 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 they are assigned to.

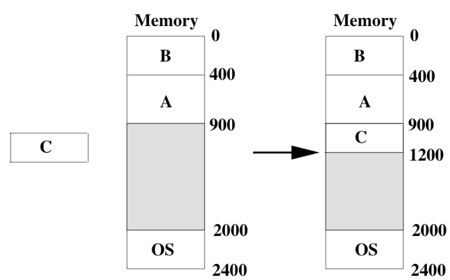
Safety:

- 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.

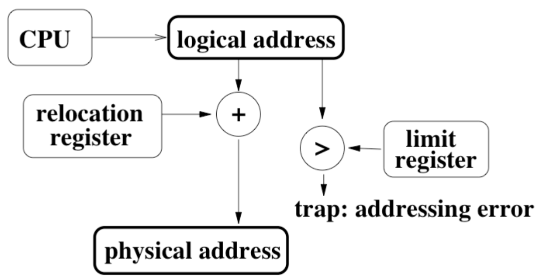
Relocation



- 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?)
- **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 processor takes an address trap and ignores the physical address.



Dynamic Relocation

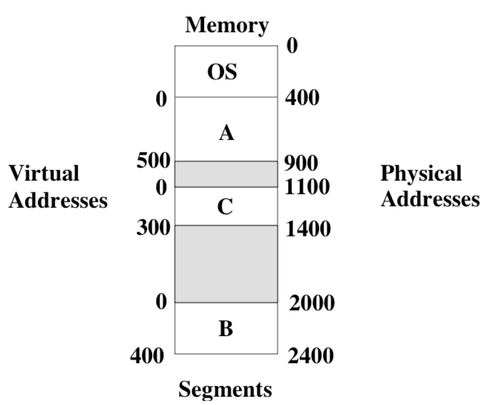
- **Advantages:**
  - OS can easily move a process during execution.
  - OS can allow a process to grow over time.
  - Simple, fast hardware: two special registers, an add, and a compare.
- **Disadvantages:**
  - Slows down hardware due to the add on every memory reference.
  - Can't share memory (such 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.

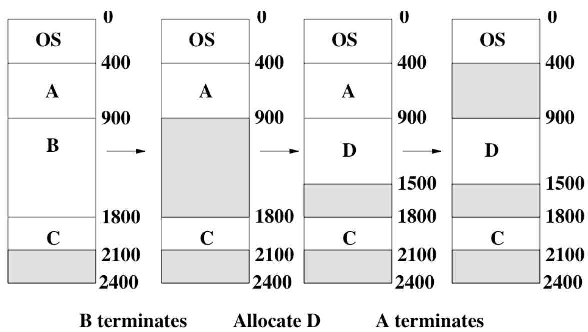
Recap

- Uniprogramming
- Static Relocation
- Dynamic Relocation
- Contiguous allocation



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

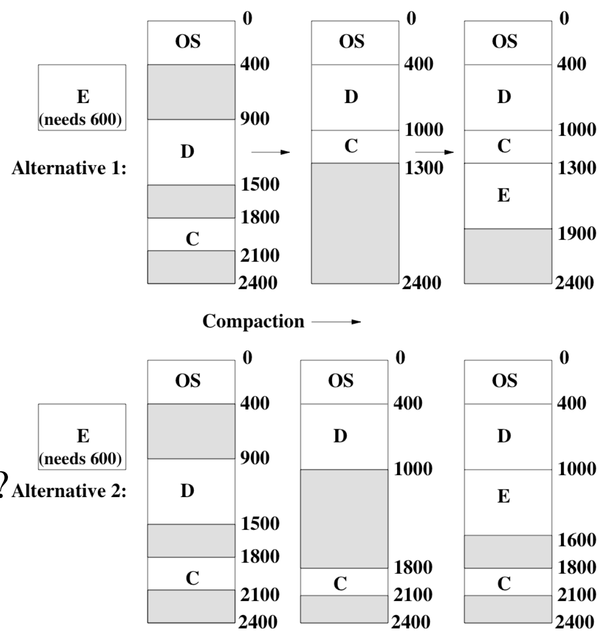
Memory Allocation Policies

- **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. 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 2*N* 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.
- **Internal Fragmentation:**
  - Consider a process of size 8846 bytes and a block of size 8848 bytes
  - ⇒ 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 that is wasted

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?

Problems

- Fragmentation
  - Frequent compaction needed
- Contiguous allocation
  - Difficult to grow or shrink process memory
- Requirement that process resides entirely in memory
  - Swapping helps but not perfect

Paging: Motivation & Features

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

=> Keep only those parts of a process in memory that are actually being used

- Pages greatly simplify the hole fitting problem
- 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 (1/2 page per process)

Paging

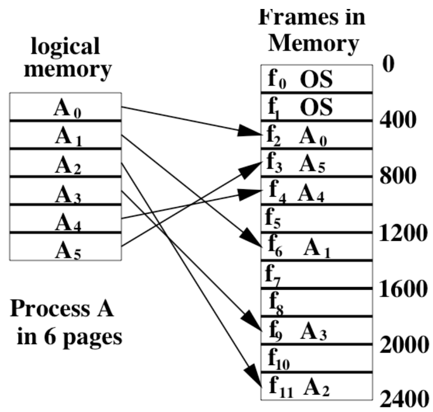
Processes typically do not use their entire space in memory all the time.

Paging

1. divides and assigns processes to fixed sized ***pages***,
2. then selectively allocates pages to ***frames*** in memory, and
3. manages (moves, removes, reallocates) pages in memory.

Paging: Example

Mapping pages in logical memory to frames in physical memory

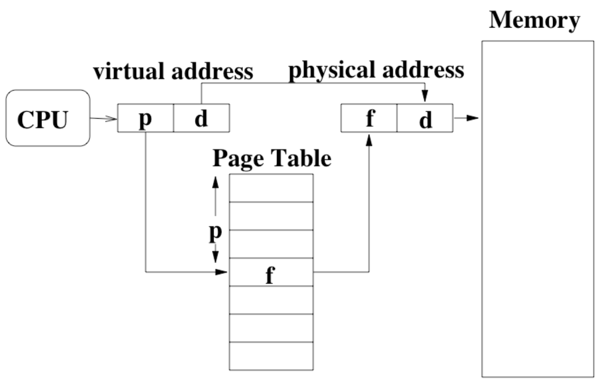


Paging Hardware

- **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.
  - *page table* keeps track of the page frame in memory in which the page is located.

Paging Hardware

Translating a virtual address to physical address



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.

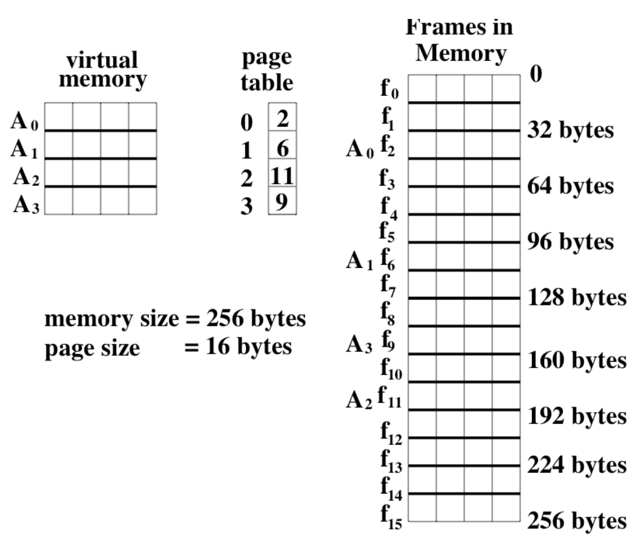
Paging Hardware: Practical Details

- 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. For example, given
- virtual address space of size 2^m bytes and a page of size 2^n, then
- the high order *m-n* bits of a virtual address select the page,
- the low order *n* bits select the offset in the page

p	d
m-n	n

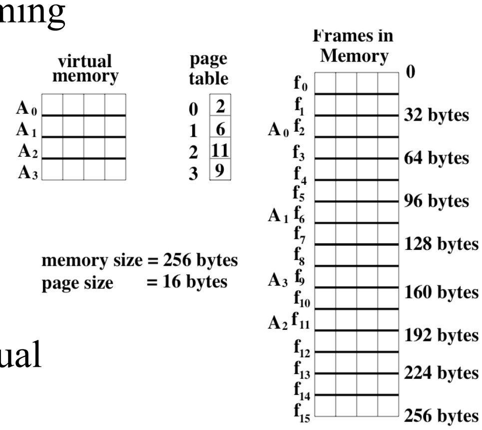
p: page number
d: page offset

Address Translation Example



Address Translation Example

- How big is the page table?
  - 16 entries (256 memory bytes / 16 byte pages)
- How many bits for an address, assuming we can address 1 byte increments?
  - 8 bits (to address 256 bytes)
- What part is p, and d?
  - 4 bits for page and 4 for offset
- Given virtual address 24, do the virtual to physical translation.
  - page p=1, offset d=8
  - frame f=6, offset d=8



Address Translation Example

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

Address Translation Example

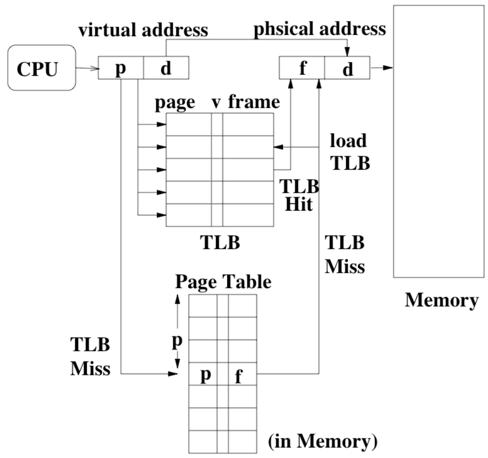
- How many bits for an address? Assume we can address only 1 word (4 byte) increments?
  - 6 bits (16 addresses of 4-byte words in 256 byte memory space)
- What part is p, and d?
  - 4 bits for for page (still 16 pages), 2 bits for offset
- Given virtual address 13, do the virtual to physical translation.
  - p=3, d=1 (virtual)
  - F=9, offset=1 (physical)

Making Paging Efficient

How should we store the page table?

- **Registers:** Advantages? Disadvantages?
- **Memory:** Advantages? Disadvantages?
- **TLB:** a fast fully associative memory that stores page numbers (key) and the frame (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.

The Translation Look-aside Buffer (TLB)



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

Costs of Using The TLB

- What is the effective memory access cost if the page table is in memory?
- What is the effective memory access cost with a TLB?

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

Costs of Using The TLB

- What is the effective memory access cost if the page table is in memory?
  - ema = 2 * ma
- What is the effective memory access cost with a TLB?
  - ema = (ma + TLB) * p + (2ma + TLB) * (1-p)

A large TLB improves hit ratio, 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 (optionally).
  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, and many systems use a multilevel paging scheme (refer OSC for details)

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.
- 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:
  - They eliminate the problem of external fragmentation and therefore the need for compaction.
  - They allow sharing of code pages among processes, reducing overall memory requirements.
  - They enable 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.