Course Outline

- Processes
- CPU Scheduling
- Synchronization & Deadlock
- **Memory Management**
- File Systems & I/O
- Distributed Systems

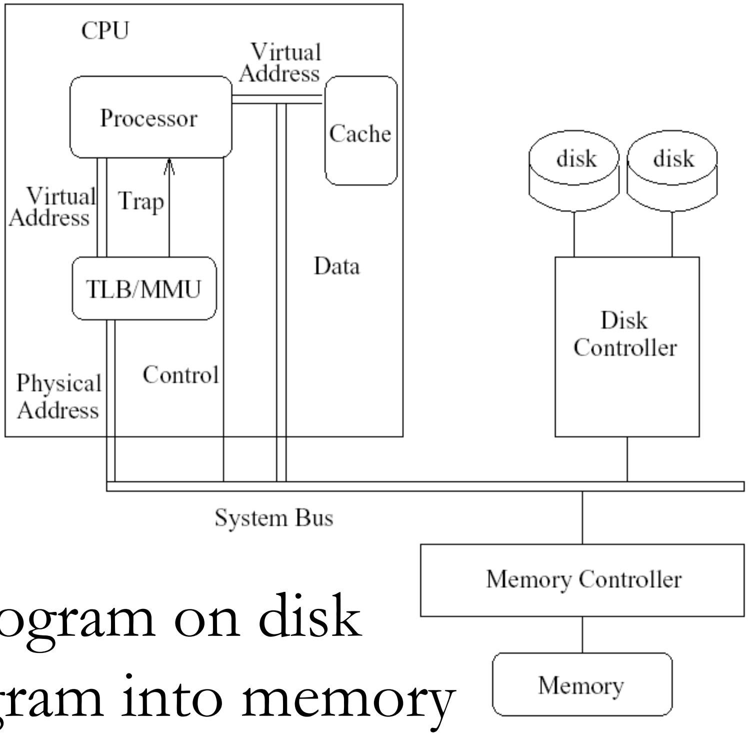
Today: Memory Management

- Terminology
- Uniprogramming
- Multiprogramming
  - Contiguous memory allocation
  - Fragmentation, compaction, swapping

Memory Management

- Where in memory is executing process?
- How do we allow multiple processes to share main memory?
- What's an *address* and how is one interpreted?

Background:
Computer Architecture

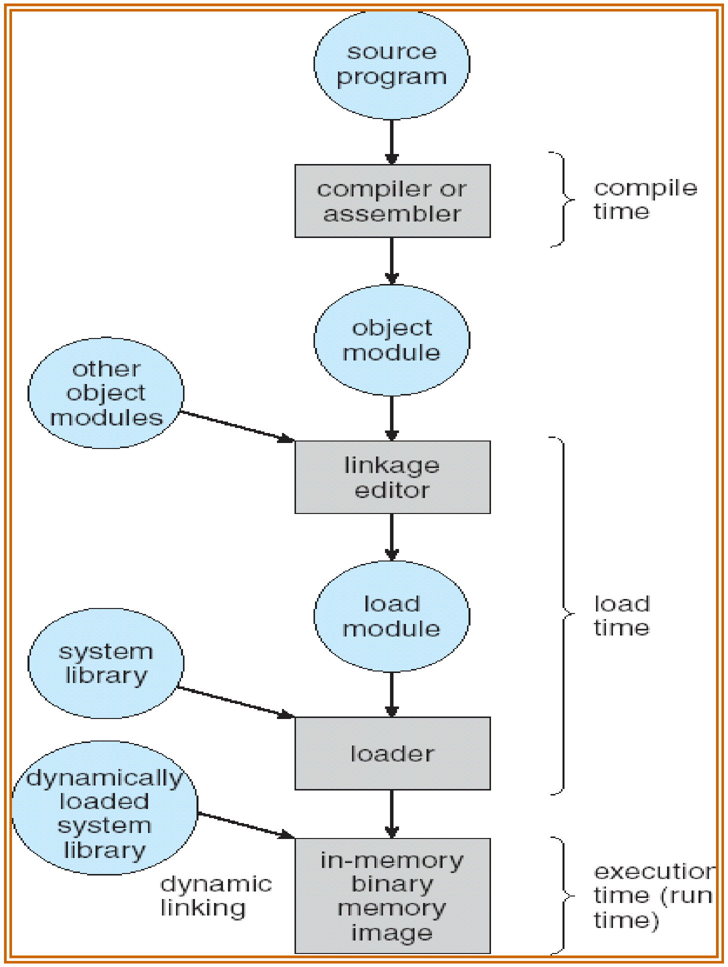


- Executable program on disk
- OS loads program into memory
- CPU fetches instructions & data from memory while executing program

Where Do Addresses Come From?

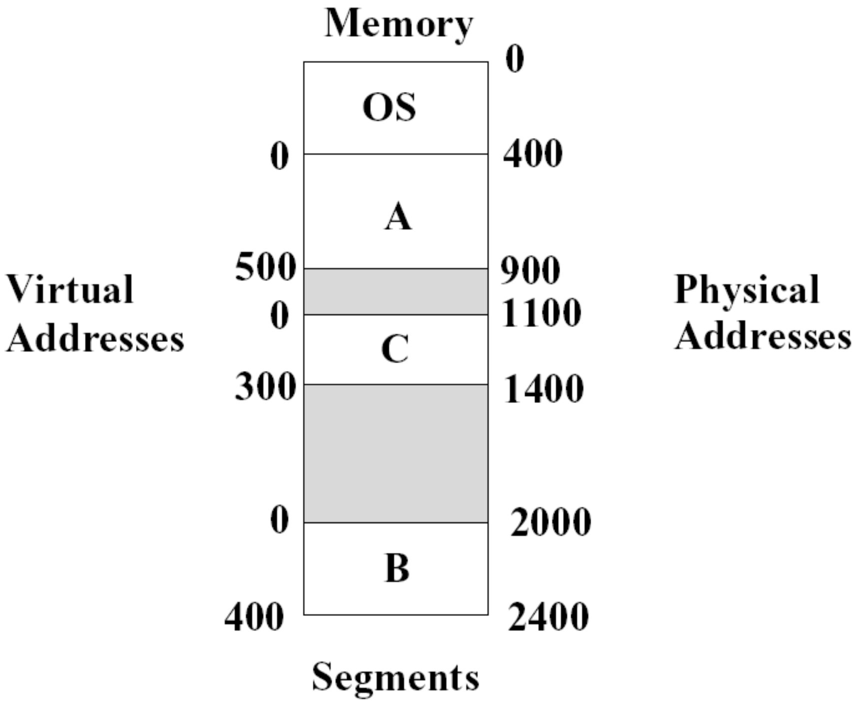
Instruction & data addresses

- **Compile-time:**
  - Exact physical location in memory starting from fixed position *k*
- **Load-time:**
  - OS determines process's starting position, fixes up addresses
- **Execution time:**
  - OS can place address anywhere in physical memory
  - Used by most general-purpose OS



Memory Management: Terminology

- **Segment**: chunk of memory assigned to process
- **Physical address**: real address in memory
- **Virtual address**: address relative to start of process's address space

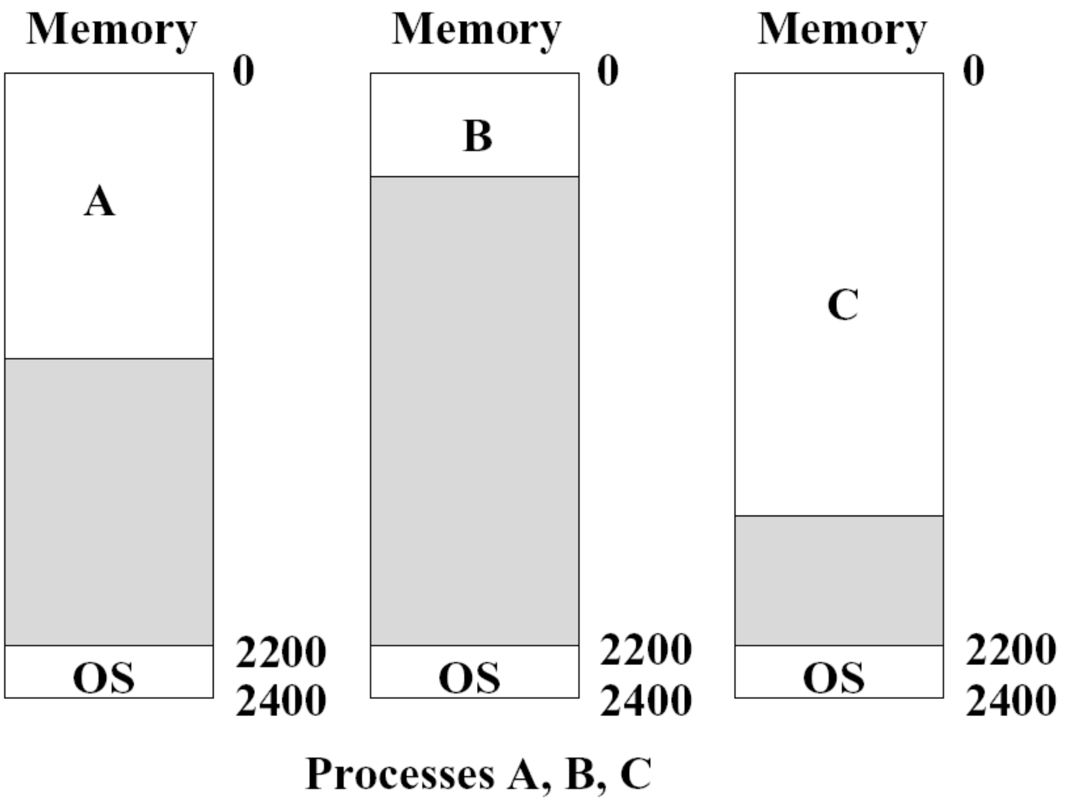


Uniprogramming

Only one program at a time: memory management is easy

- OS gets fixed region of memory (e.g., highest)
- One process at a time
  - Load at address 0
  - Executes in contiguous memory
- Compiler generates physical addresses
  - Max address = memory size – OS size
  - OS protected from process by checking addresses

Example: Uniprogramming



Processes A, B, C

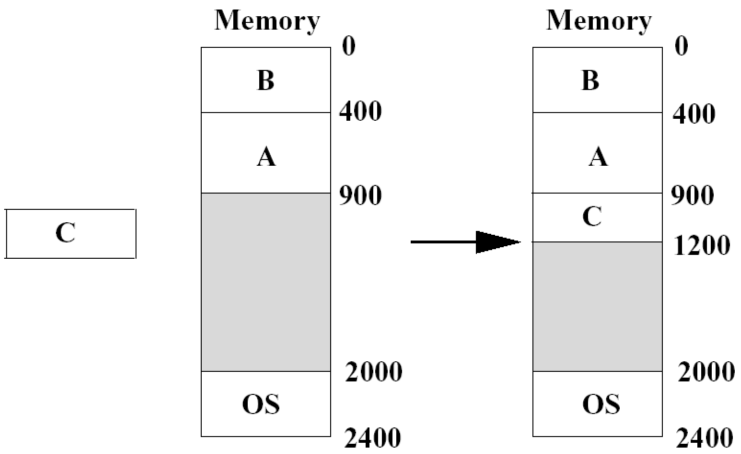
- Simple – but no overlap of I/O, computation

Multiprogramming Requirements

- **Transparency**
  - No process aware memory is shared
  - Process has no constraints on physical memory
- **Safety**
  - Processes cannot corrupt each other or OS
- **Efficiency**
  - Performance not degraded due to sharing

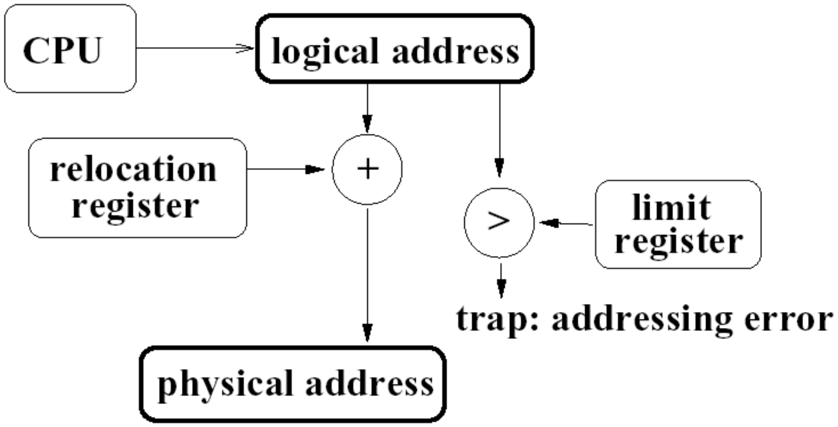
Contiguous memory allocation

- Put OS in high memory
- Process starts at 0
  - Max addr = memory size – OS size
- Load process by allocating contiguous segment for process
- Smallest addr = *base*, largest = *limit*



Address Translation

- Hardware adds relocation register (base) to virtual address to get physical address
- Hardware compares address with limit register
  - Test fails → trap



Properties

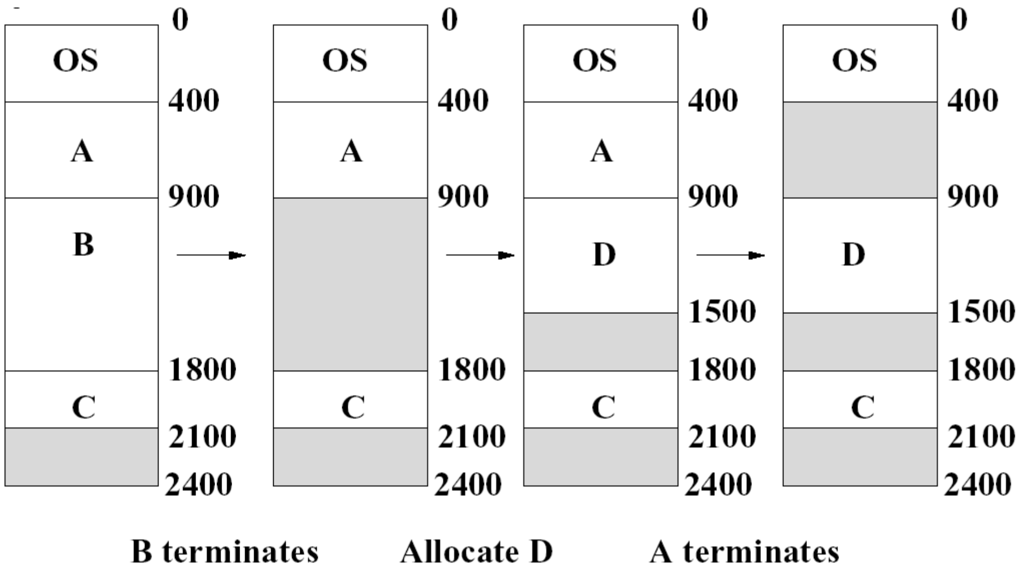
- Transparency
  - Processes largely unaware of sharing
- Safety
  - Each memory reference checked
- Efficiency
  - Memory checks fast if done in hardware
  - But: if process grows, may have to be moved (SLOW)

Pros & Cons

- Advantages
  - Simple, fast hardware
    - Two special registers, add & compare
- Disadvantages
  - Process limited to physical memory size
  - Degree of multiprogramming limited
    - All memory of active processes must fit in memory

Allocating "holes"

- As processes enter system, grow & terminate, OS must track available and in-use memory



- Can leave *holes*
  - OS must decide where to put new processes

Memory Allocation Policies

- **First-fit:**
  - Use first hole in which process fits
- **Best-fit:**
  - Use smallest hole that's large enough
- **Worst-fit:**
  - Use *largest* hole

- What's best? First-fit and best-fit comparable, better than worst-fit in speed and memory utilization

Fragmentation

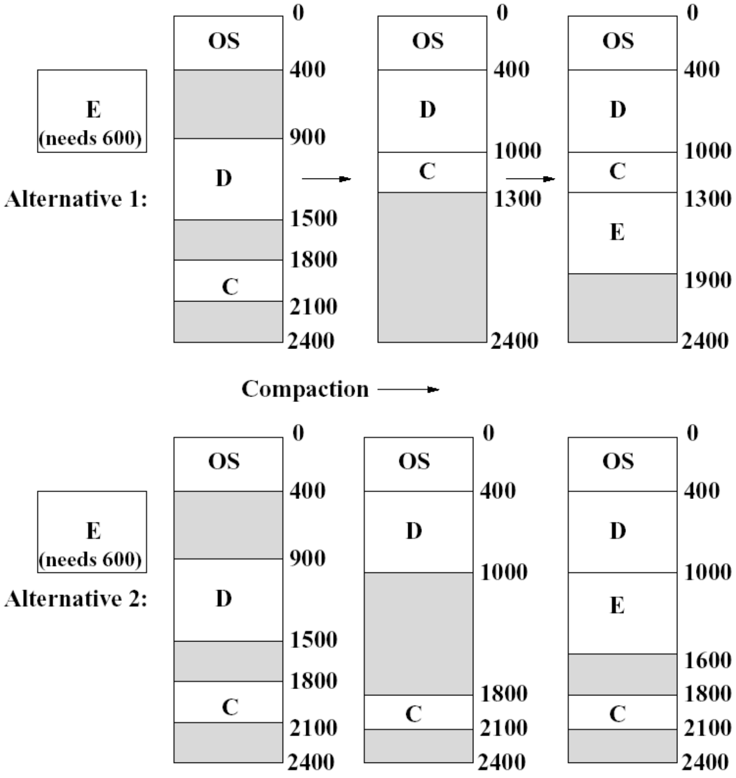
- Fragmentation = % memory unavailable for allocation, but not in use
- **External fragmentation:**
    - Large # of small holes s.t. even the total size satisfies a request; no contiguous chunk can be found
    - Caused by repeated unloading & loading
- **Internal fragmentation:**
    - Space inside process allocations
        - Unavailable to other processes

Compaction

- Can make space available by shuffling process space
  - Eliminate holes
  - Place free memory together
  - Cannot move a process if addresses are determined at compile or load time

Compaction Example

- Issues
  - Amount of memory moved
  - Size of created block
  - Other choices?



Alternative: Swapping

- Swapping = copy process to disk, release all memory
  - When process active, must reload
  - Static relocation: same position(!)
  - Dynamic relocation: ok
- Drawback?

Summary

- Processes must reside in memory to execute
- Generally use virtual addresses
    - Translated to physical addresses before accessing memory
- Contiguous memory allocation:
    - Allows processes to share memory
    - Pros and cons

Paging

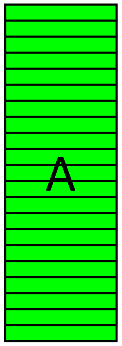
- Motivation
- Page Tables
- Hardware Support
- Benefits

Problems with Continuous memory allocation

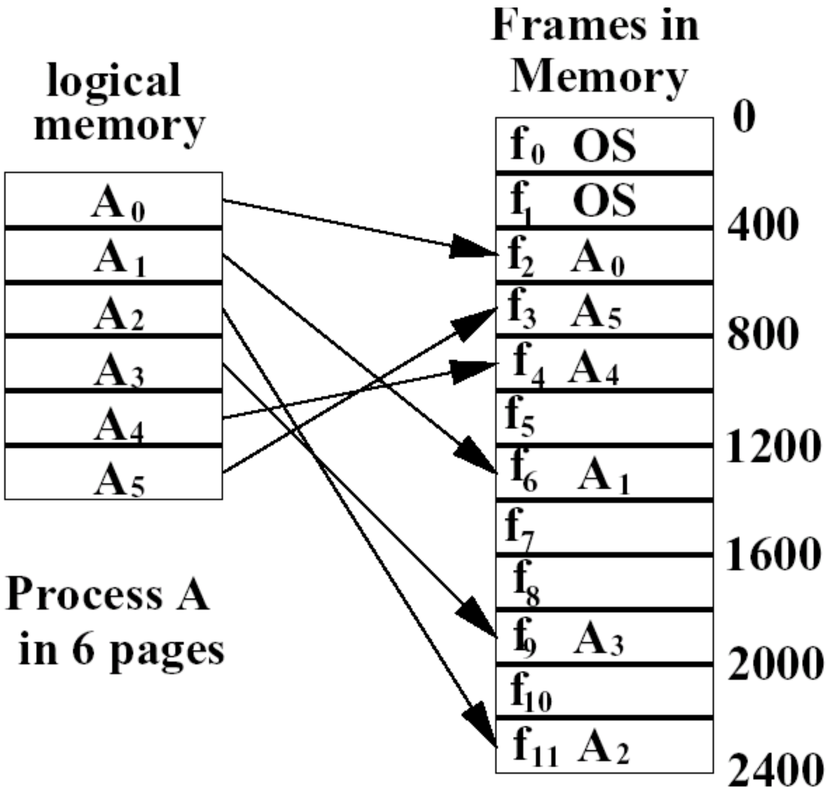
- Processes don’t (usually) use its entire space in memory all the time
- Fragmentation problematic
- Compaction expensive

Alternative: Paging

- Divide logical memory into fixed-sized *pages* (4K, 8K)
- Divide physical memory into fixed-sized *frames*
  - Pages & frames same size
  - OS manages pages
    - Moves, removes, reallocates
- Disk space: blocks same size as frames
  - Pages copied to and from disk to frames



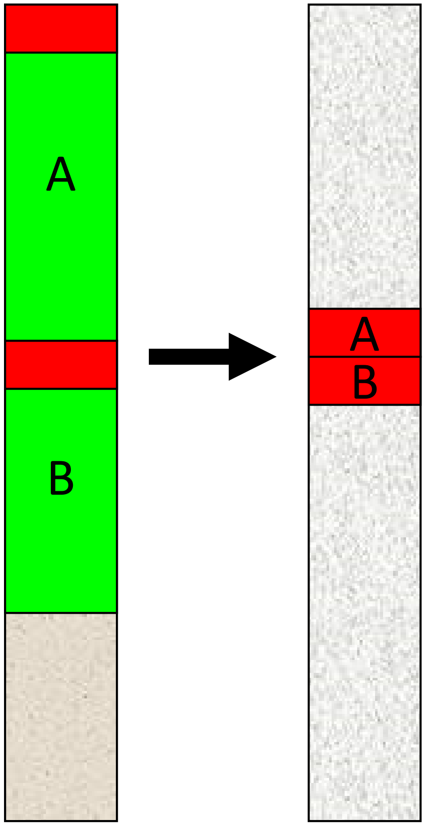
Example: Page Layout



- How does this help?

Paging Advantages

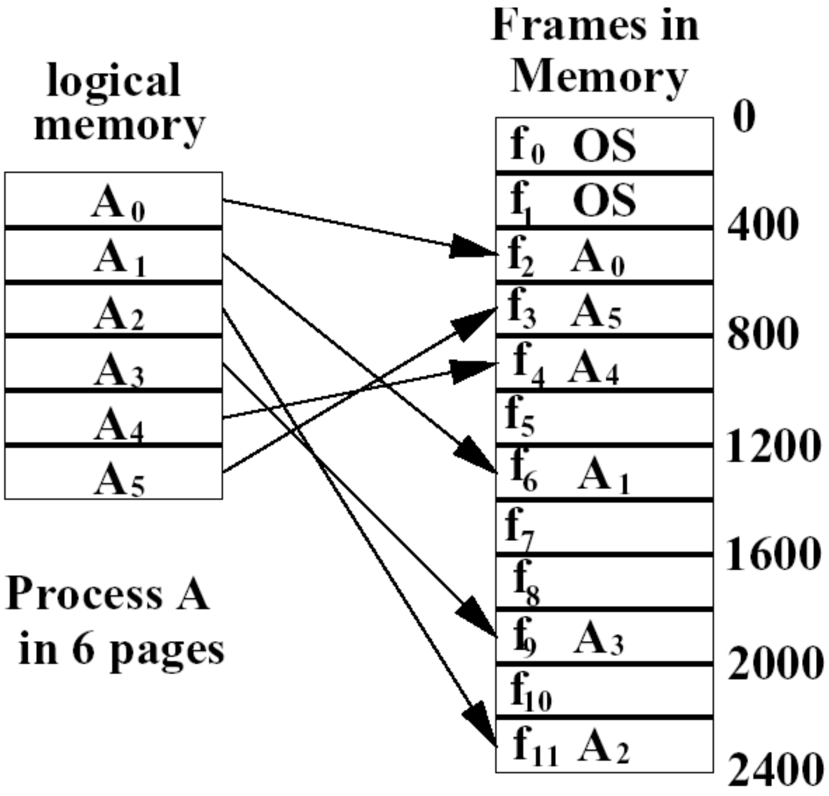
- Most programs obey 90/10 "rule"
  - 90% of time spent accessing 10% of memory
- Exploiting this rule:
  - Only keep "live" parts of process in memory



Paging Advantages

- “Hole-fitting problem” vanishes!
  - Logical memory contiguous
  - Physical memory not required to be
- Eliminates external fragmentation
  - But not internal (why not?)
- But: Complicates address lookup...

Example: Page Layout



- So how do we resolve addresses?

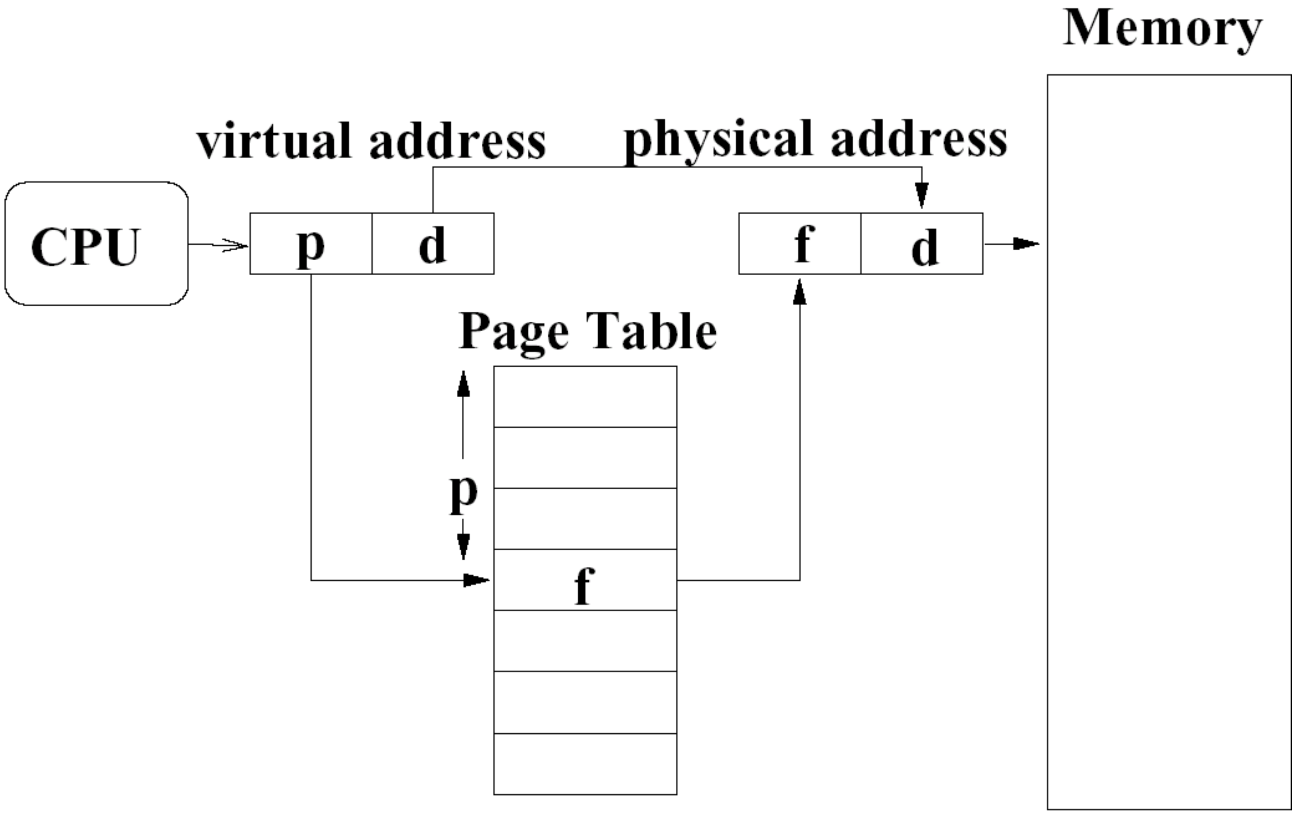
Paging

- Motivation
- Page Tables
- **Hardware Support**
- Benefits

Paging Hardware

- Processes use virtual addresses
  - Addresses start at 0 or other known address
  - OS lays process down on pages
- MMU (memory-management unit):
  - Hardware support for paging
  - Translates virtual to physical addresses
  - Uses *page table* to keep track of frame assigned to memory page

Paging Hardware: Diagram



Paging Hardware: Intuition

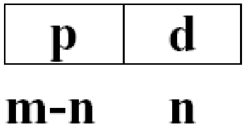
- Paging: form of dynamic relocation
  - Virtual address bound by paging hardware to physical address
- Page table: similar to a set of relocation registers
- Mapping – invisible to process
  - OS maintains mapping
  - H/W does translation
- Protection – provided by same mechanisms as in dynamic relocation

Paging Hardware: Nitty-Gritty

- Page size (= frame size):
  - Typically power of 2 between 512 & 8192 bytes
  - Linux, Windows: 4K; Solaris: 8K
  - Support for larger page sizes varies (e.g., 128K)
- Use of powers of 2 simplifies translation of virtual to physical addresses

Address Translation

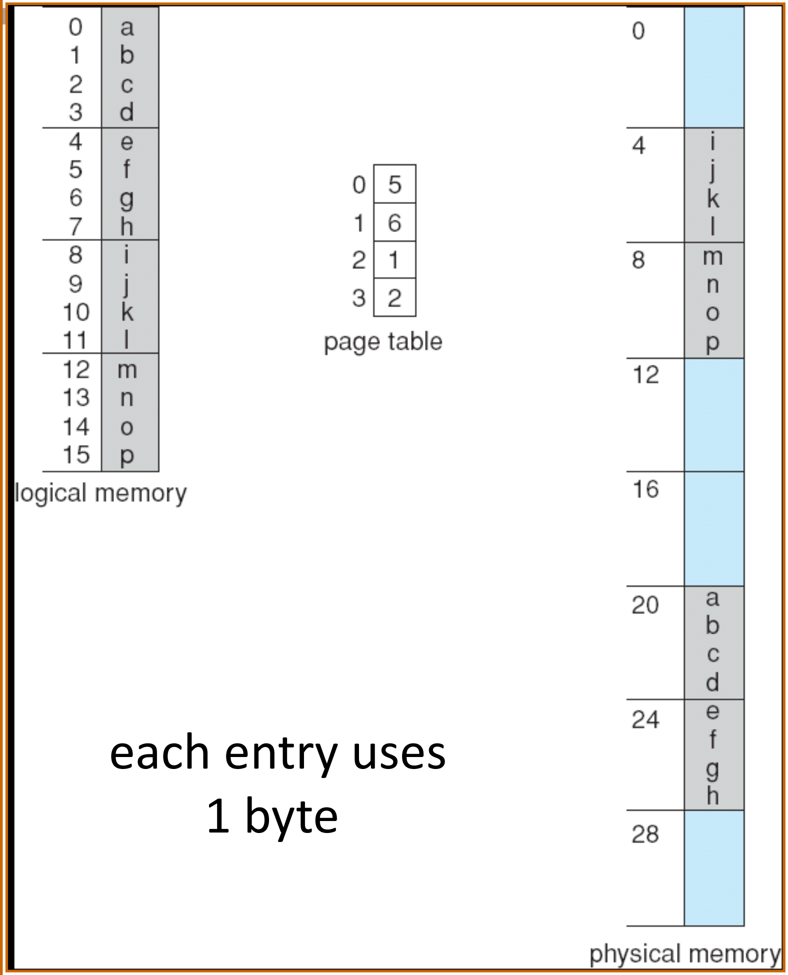
- Powers of 2:
  - Virtual address space: size 2^m
  - Page size 2^n
- High-order m-n bits of virtual address select page
- Low order n bits select offset in page



p: page number
d: page offset

Address Translation: Example

- Assume 1 byte addressing, each page contains 4 bytes:
  - Length of p, d?
  - Given virtual address 0, 4, 10, 13, do virtual to physical translation



each entry uses 1 byte

p	d
m-n	n

p: page number
d: page offset

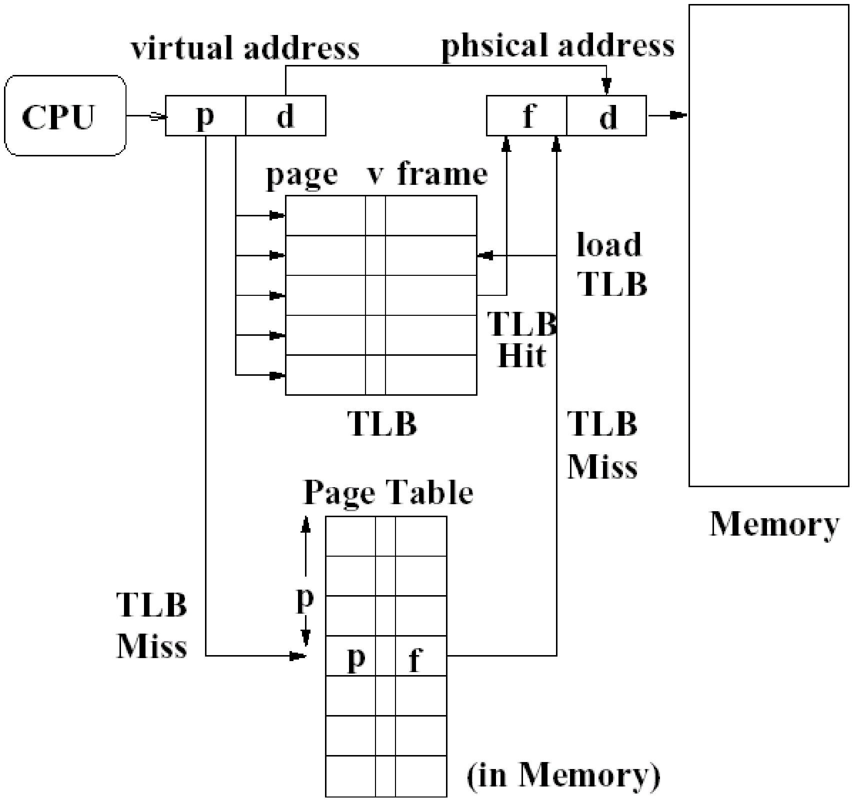
Making Paging Efficient

- Where should the page table go?
  - Registers:
    - Pros? Cons?
  - Memory:
    - Pros? Cons?

Translation Lookaside Buffer (TLB)

- Small, fast-lookup hardware cache
- TLB sizes: 8 to 2048 entries

TLB: Diagram



- v = valid bit: entry is up-to-date

Effectiveness of TLB

- Processes exhibit locality of reference
  - **Temporal locality**: processes tend to reference same items repeatedly
  - **Spatial locality:** processes tend to reference items near each other (e.g., on same page)
- Locality in memory accesses → locality in address translation

Benefits from TLB

- Example:
  - Hit ratio: 0.8;
  - on average: search TLB: 20 nanosec; search memory: 100 nanosec
  - What is the average cost to access/read an item in memory?
  - What if no TLB is used?

Managing TLB: Process Initialization & Execution

- Process arrives, needs *k* pages
- If *k* page frames free, allocate; else free frames that are no longer needed
- OS:
  - puts pages in frames
  - puts frame numbers into page table
  - marks all TLB entries as invalid (*flush*)
  - starts process
  - loads TLB entries as pages are accessed, replaces entries when full

Managing TLB: Context Switches

- Extend Process Control Block (PCB) with:
  - Page table
  - Copy of TLB (optional)
- Context switch:
  - Copy page table to PCB
  - Copy TLB to PCB, Flush TLB (optional)
  - Restore page table
  - Restore TLB (optional)
- Use *multilevel paging* if tables too big (see text)

Paging

- Motivation
- Page Tables
- Hardware Support
- **Benefits**

Benefits: Compared to Contiguous-Memory Allocation

- Eliminates external fragmentation (thus avoiding need for compaction)
- Enables processes to run when only partially loaded in main memory

Benefits: Allow Sharing

- Paging allows sharing of memory across processes
  - Shared pages –different virtual addresses, point to same physical address
- Compiler marks "text" segment (i.e., code) of applications (e.g., emacs) - read-only
- OS: keeps track of such segments
  - Reuses if another instance of app arrives
- Can *greatly* reduce memory requirements

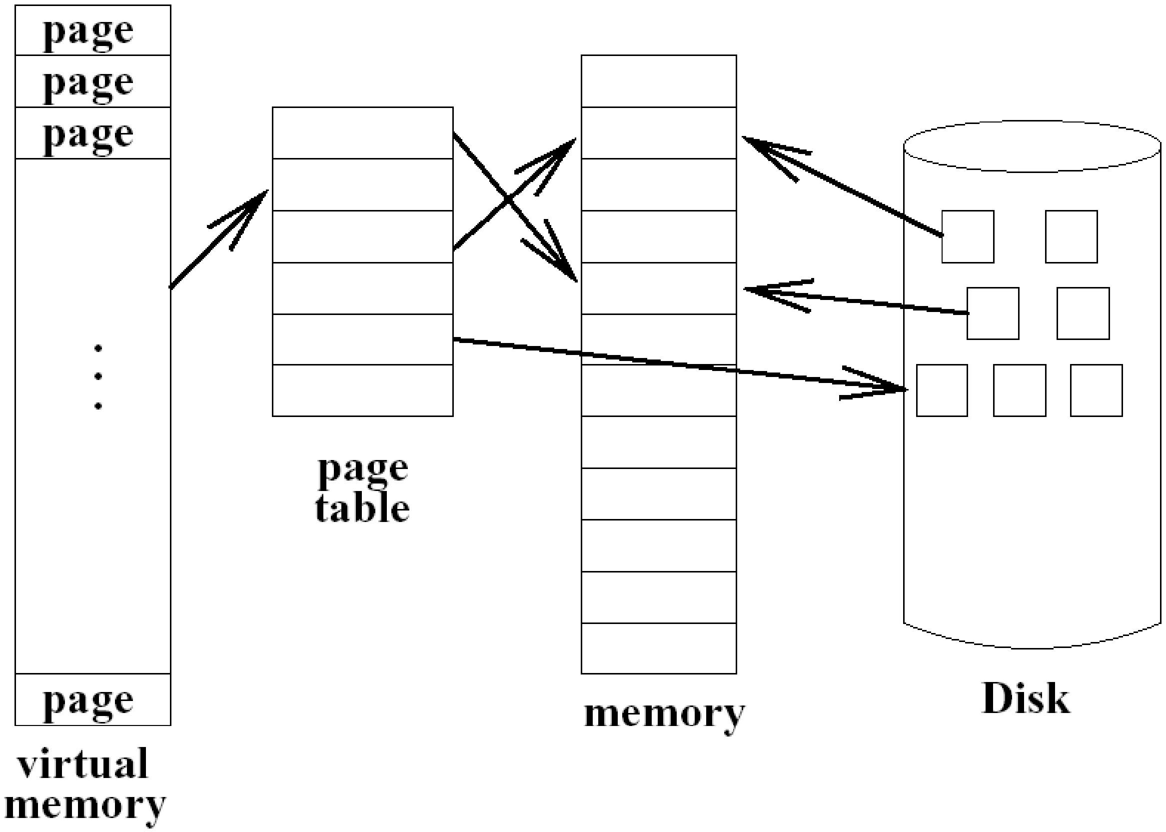
Paging Disadvantages

- Paging: some costs
  - Translating from virtual addresses to physical addresses efficiently requires hardware support
    - Larger TLB → more efficient, but more expensive
  - More complex operating system required to maintain page table
  - More expensive context switches (why?)

Demand-Paged VM

- Reading pages
- Writing pages
  - Swap space
- Page eviction
- Cost of paging
- Page replacement algorithms
  - Evaluation

Demand-Paging Diagram



Key Policy Decisions

- Two key questions:
  - When do we read page *from* disk?
  - When do we write page *to* disk?

Reading Pages

- Read **on-demand**:
  - OS loads page on its first reference
  - May force an **eviction** of page in RAM
  - Pause while loading page = **page fault**
- Can also perform **pre-paging**:
  - OS *guesses* which page will next be needed, and begins loading it
  - Advantages? Disadvantages?
- Most systems just do demand paging

Demand Paging

- On every reference, check if page is in memory (valid bit in page table)
- If not: trap to OS
- OS checks address validity, and
  - Selects **victim page** to be replaced
  - Begins loading new page from disk
  - Switches to other process (demand paging = implicit I/O)
- Note: must restart instruction later

Demand Paging, Continued

- Interrupt signals page arrival, then:
  - OS updates page table entry
  - Continues *faulting* process
    - Stops current process
- We could continue currently executing process – but why not?

- And where does the victim page go?

Demand Paging, Continued

- Interrupt signals page arrival, then:
  - OS updates page table entry
  - Continues *faulting* process
    - Stops current process
- We could continue currently executing process – but why not?
  - Page just brought in could get paged out…

Virtual Memory Locations

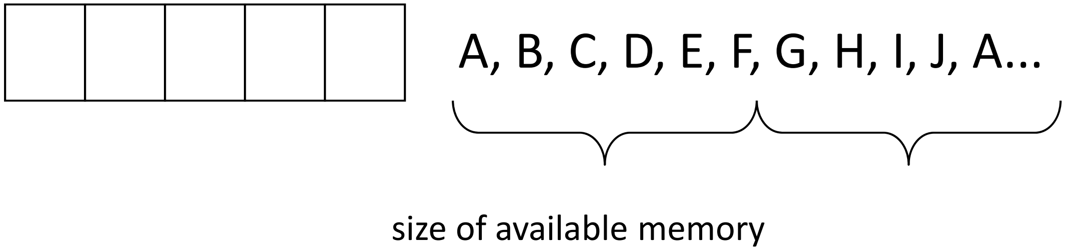
- VM pages can now exist in one or more of following places:
  - Physical memory (in RAM)
  - Swap space (victim page)
  - Filesystem (why?)

Page Replacement

- Process is given a fixed memory space of *n* pages
- Question:
  - process requests a page
  - page is not in memory, all *n* pages are used
  - which page should be evicted from memory?

Page Replacement: Cost of Paging

- Worst-case analysis
  - Easy to construct *adversary* example: every page requires page fault
  - Not much you can do, paging useless



A, B, C, D, E, F, G, H, I, J, A…

size of available memory

Page Replacement: Cost of Paging, cont’d

- But: processes exhibit locality,
  so performance generally not bad
  - **Temporal locality**: processes tend to reference same items repeatedly
  - **Spatial locality:** processes tend to reference items near each other (e.g., on same page)

Metric: Effective Access Time

- Let p = probability of page fault ($0 \le p \le 1$)
  ma = memory access time
- Effective access time =
  $(1 - p) * ma + p *$ **page fault service time**
  - Memory access = 200ns, page fault = 25ms: effective access time = (1-p)*200 + p*25,000,000

Evaluating Page Replacement Algorithms

- Average-case:
  - Empirical studies – real application behavior
- Theory: **competitive analysis**
  - Can't do better than optimal
  - How far (in terms of faults) is algorithm from optimal in worst-case?
    - **Competitive ratio**
    - If algorithm can't do worse than 2x optimal, it's **2-competitive**

Page Replacement Algorithms

- MIN, OPT (optimal)
- RANDOM
  - evict random page
- FIFO (first-in, first-out)
  - give every page equal *residency*
- LRU (least-recently used)
- MRU (most-recently used)

MIN/OPT

- Invented by Belady (“MIN”), now known as “OPT”: optimal page replacement
  - Evict page to be accessed furthest in the future
- Provably optimal policy
  - Just one small problem...
- Requires predicting the future
  - Useful point of comparison

MIN/OPT example

	A	B	C	A	B	D	A	D	B	C	B
frame 1	A	A	A			A				C	
frame 2		B	B			B				B	
frame 3			C			D				D	

- Page faults: 5

RANDOM

- Evict *any* page
- Works surprisingly well
- Theoretically: very good
- Not used in practice:
  takes no advantage of locality

LRU

- Evict page that has not been used in longest time (least-recently used)
  - Approximation of MIN if recent past is good predictor of future
  - A *variant* of LRU used in all real operating systems
- Competitive ratio: n, (n: # of page frames)
  - Best possible for deterministic algs.

LRU example

	A	B	C	A	B	D	A	D	B	C	B
frame 1											
frame 2											
frame 3											

- Page faults: ?

LRU example

	A	B	C	A	B	D	A	D	B	C	B
frame 1	A	A	A			A				C	
frame 2		B	B			B				B	
frame 3			C			D				D	

- Page faults: 5

LRU, example II

	A	B	C	D	A	B	C	D	A	B	C	D
frame 1												
frame 2												
frame 3												

- Page faults: ?

LRU, example II

	A	B	C	D	A	B	C	D	A	B	C	D
frame 1	A	A	A	D	D	D	C	C	C	B	B	B
frame 2		B	B	B	A	A	A	D	D	D	C	C
frame 3			C	C	C	B	B	B	A	A	A	D

- Page faults: 12

FIFO

- First-in, first-out: evict *oldest* page
  - Also has competitive ratio *n*
- But: performs miserably in practice!
  - LRU takes advantage of locality
  - FIFO does not
- Suffers from **Belady's anomaly:**
  - More memory can mean more paging!

FIFO & Belady’s Anomaly

- Request sequence

A B C D A B E A B C D E

- Q1: # of page faults when n=3?
- Q2: # of page faults when n=4?
- Q3: what are the results under LRU?

FIFO & Belady’s Anomaly

	A	B	C	D	A	B	E	A	B	C	D	E
frame 1	A	A	A	D	D	D	E			E	E	
frame 2		B	B	B	A	A	A			C	C	
frame 3			C	C	C	B	B			B	D	
frame 1	A	A	A	A			E	E	E	E	D	D
frame 2		B	B	B			B	A	A	A	A	E
frame 3			C	C			C	C	B	B	B	B
frame 4				D			D	D	D	C	C	C

- When n=3, 9 page faults
- When n=4, 10 page faults

LRU: No Belady's Anomaly

	A	B	C	D	A	B	E	A	B	C	D	E
frame 1	A	A	A	D	D	D	E			C	C	C
frame 2		B	B	B	A	A	A			A	D	D
frame 3			C	C	C	B	B			B	B	E
frame 1	A	A	A	A			A			A	A	E
frame 2		B	B	B			B			B	B	B
frame 3			C	C			E			E	D	D
frame 4				D			D			C	C	C

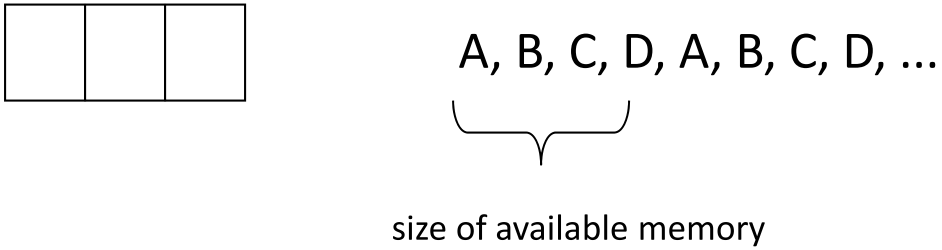
- When n=3, 10 page faults
- When n=4, 8 page faults

Why no anomaly for LRU?

- “Stack” property:
  - Pages in memory for memory size of *n* are also in memory for memory size of *n+1*

MRU

- Evict most-recently used page
- Shines for LRU’s worst-case: loop that exceeds RAM size



$\underbrace{\text{A, B, C,}}_{\text{size of available memory}}$ D, A, B, C, D, …

- What we really want: *adaptive algorithms* (e.g., EELRU – Kaplan & Smaragdakis)

Summary

- Reading pages
- Writing pages
    - Swap space
- Page eviction
- Cost of paging
- Page replacement algorithms
    - Evaluation