Today

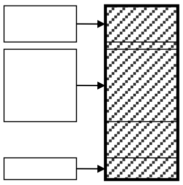
- Memory Management: Page Replacement

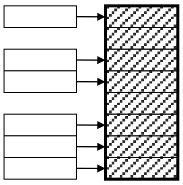
Review

- What is paging? Segmentation?
  - What are they used for?
  - How does the OS translate from the virtual address to the physical address?
  - Compare and contrast them
- What hardware support is provided for VM?
- How can we improve the efficiency/performance of address translations?

Review: Defining Regions

- Segmentation:
  - Partition address space and memory into logical segments
  - Segments have *varying* sizes
- Paging:
  - Partition address space and memory into pages
  - Pages are a constant, *fixed* size





Review: Pros and Cons of Segmentation

Pros

- Each segment can be
  - located independently
  - separately protected
  - grown/shrunk independently
- Small segment table size
  - ~256 Bytes → 1GB memory

Cons

- Variable-size allocation
  - Difficult to find holes in physical memory
  - External fragmentation

Review: Pros and Cons of Paging

What we'll assume is being used

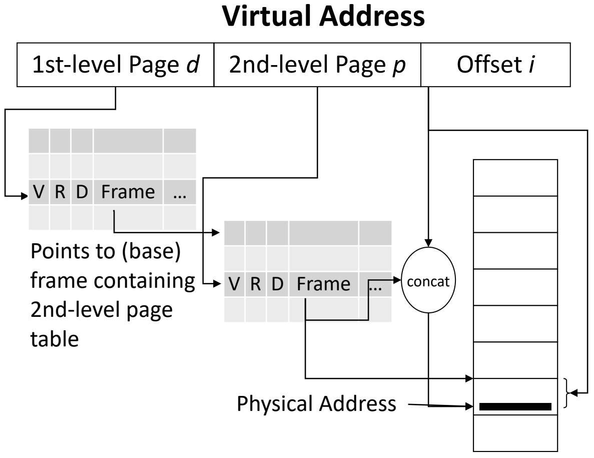
Pros

- Each page can be
  - located independently
  - separately protected
- Fixed-size pages and frames
  - No external fragmentation
  - No difficult placement decisions

Cons

- Large table size
  - ~4MB for 1GB of memory
    - That's for each process!
- *maybe* internal fragmentation

Review: Multi-Level Page Tables



Insight: VAS is typically sparsely populated

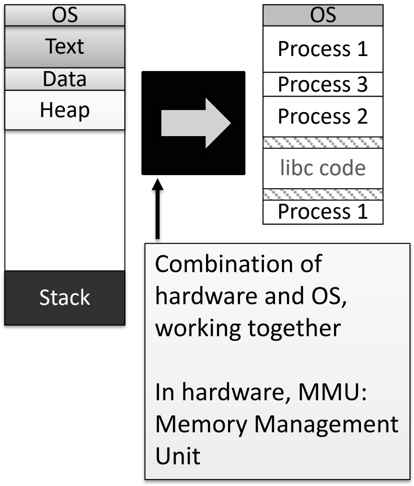
Idea: every process gets a page directory

- 1st-level table

Only allocate 2nd-level tables when the process is using that VAS region!

Reduce required size of page table!

Review:
Memory Management Unit (MMU)



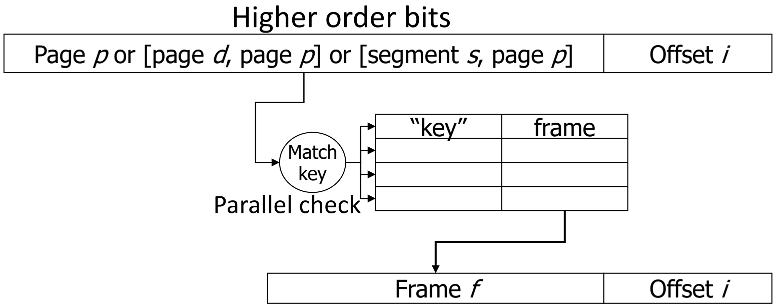
Combination of hardware and OS, working together

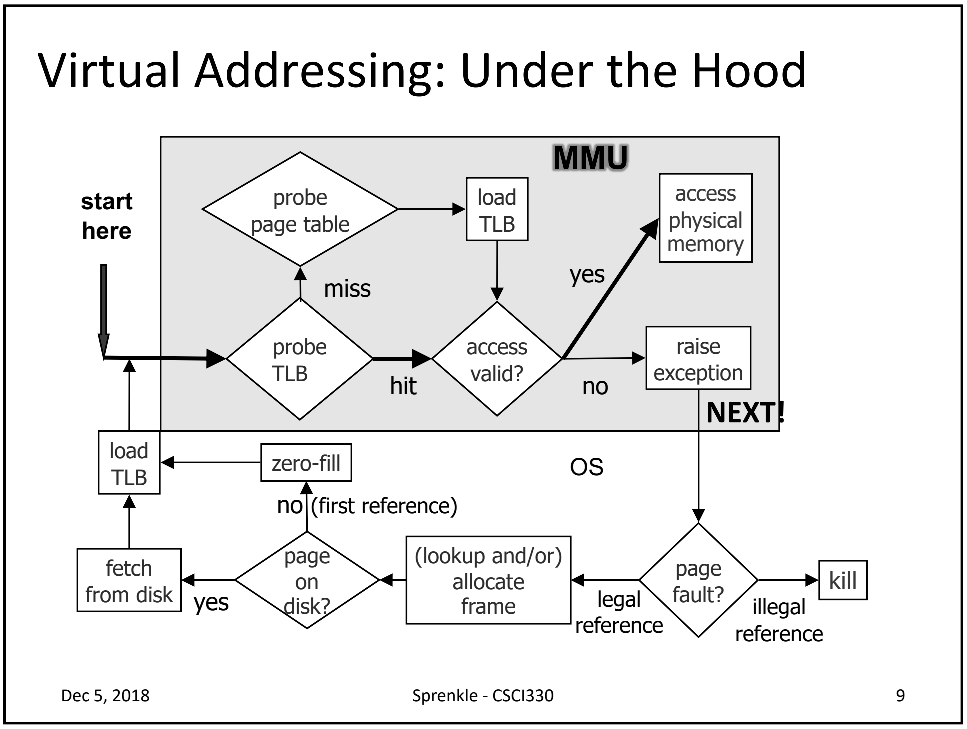
In hardware, MMU: Memory Management Unit

- When a process tries to use memory, send the address to MMU
- MMU will do as much work as it can
    - ➢ If it knows the answer, great!
- If it doesn't
    - ➢ trigger exception (OS gets control)
    - ➢ consult software table

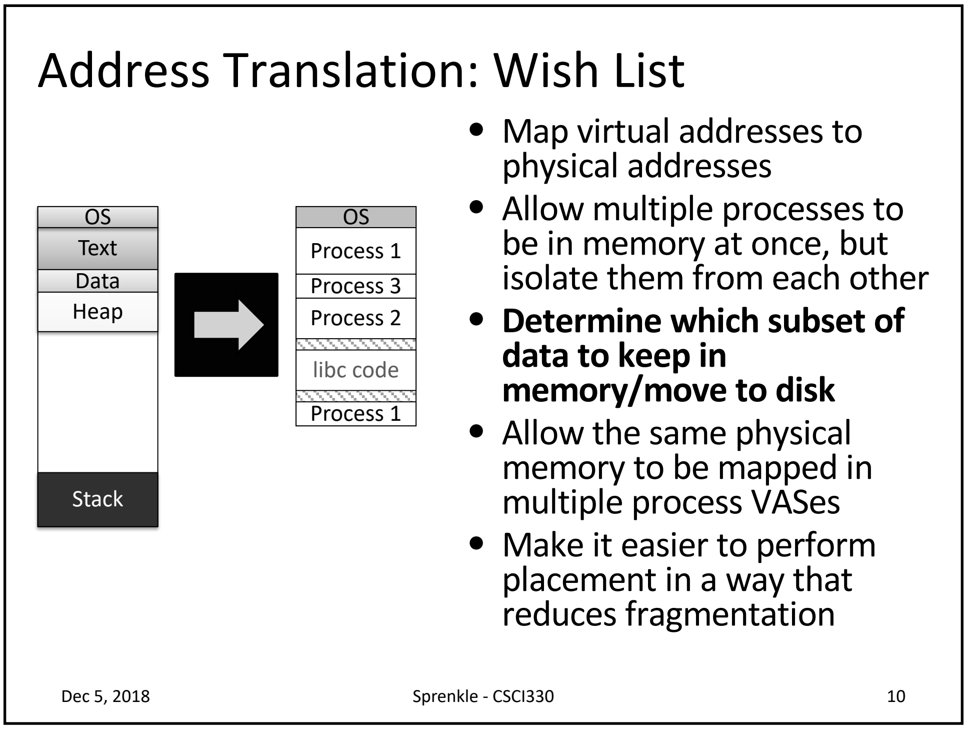
Review:
Translation Look-aside Buffer (TLB)

- Fast memory mapping cache inside MMU keeps most recent translations
    - ➢ If key matches, get frame number quickly
    - ➢ Otherwise, wait for normal translation
        - Add to TLB



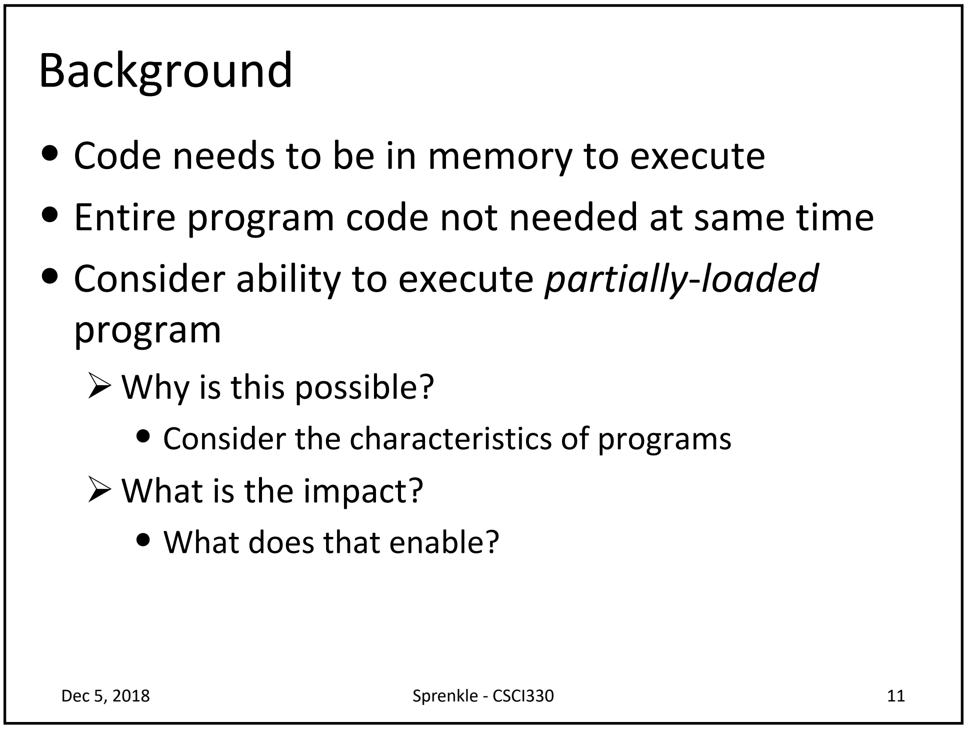


Virtual Addressing: Under the Hood



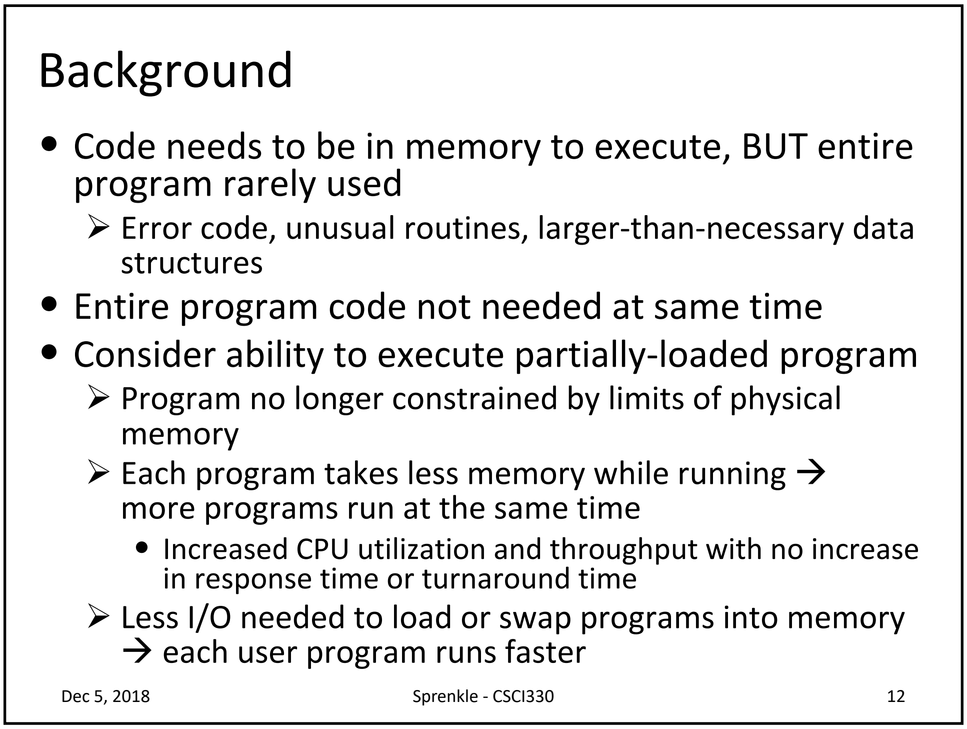
Address Translation: Wish List

- Map virtual addresses to physical addresses
- Allow multiple processes to be in memory at once, but isolate them from each other
- **Determine which subset of data to keep in memory/move to disk**
- Allow the same physical memory to be mapped in multiple process VASes
- Make it easier to perform placement in a way that reduces fragmentation



Background

- Code needs to be in memory to execute
- Entire program code not needed at same time
- Consider ability to execute *partially-loaded* program
  - ➢ Why is this possible?
    - Consider the characteristics of programs
  - ➢ What is the impact?
    - What does that enable?



Background

- Code needs to be in memory to execute, BUT entire program rarely used
  - ➢ Error code, unusual routines, larger-than-necessary data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - ➢ Program no longer constrained by limits of physical memory
  - ➢ Each program takes less memory while running → more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - ➢ Less I/O needed to load or swap programs into memory → each user program runs faster

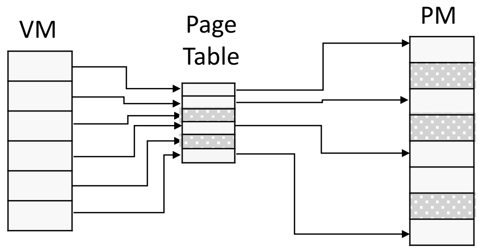
Virtual Memory

- **Idea**: use physical memory to hold only the portions of each executing process that are currently being used
  - ➢ Only part of the program needs to be in memory for execution
  - ➢ Parts of executing process that are not currently being used are held on secondary storage until needed.
- **Impact**:
  - ➢ Logical address space can be much larger than physical address space
  - ➢ Allows address spaces to be shared by several processes
  - ➢ Less I/O needed to load or swap processes

“Swapping” Pages to Disk

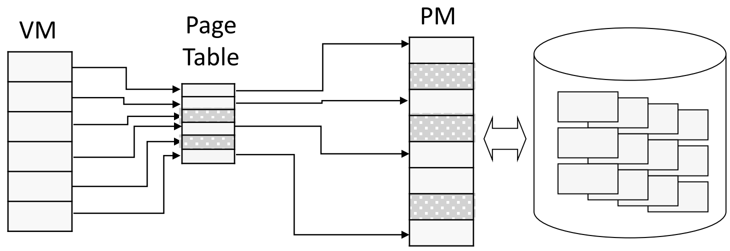
- Intuition: If a process isn’t using a page, why keep it in physical memory? Instead, send it to disk and reclaim that space
- Illusion: memory size is physical memory + disk (with non-uniform access times)
- Supporting this idea requires:
  - ➢ Identifying where a chunk of memory is (physical memory or disk?)
  - ➢ Moving data between physical memory and disk (mechanism)
  - ➢ Algorithm for governing what gets moved to disk and what stays (policy)

Virtual Memory based on Paging



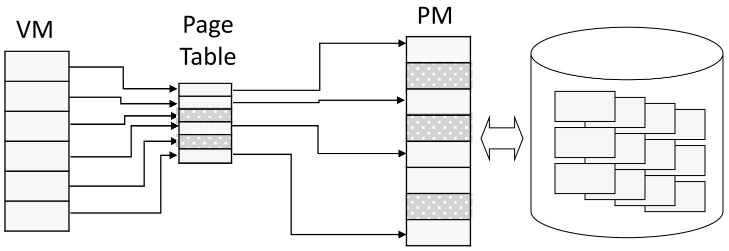
- Before
  - All virtual pages were in physical memory.

Virtual Memory based on Paging



- Now
  - Pages, if they exist, reside in physical memory or on disk (or both)
  - Which pages are on disk? In memory?

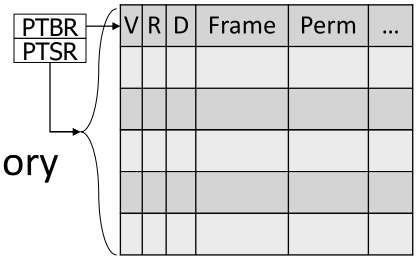
Virtual Memory based on Paging



- Now
  - Pages, if they exist, res disk (or both)
  - Which pages are on disk? In memory?

For disk, assume simple lookup structure:
- Key: Process ID, Page Number
- Value: Location of page on disk (or error if not there)

Page Table: Revisited



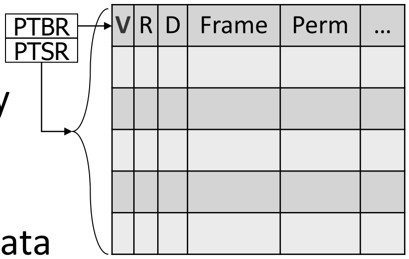
PTBR
PTSR

V	R	D	Frame	Perm	…

- One table per process
- Table parameters in memory
  - Page table base register
  - Page table size register
- By loading these registers, the **hardware** (MMU) knows where the page table is for the current process!

OS maintains the table,
but **hardware can access** it to help **improve performance**!

Page Table: Revisited



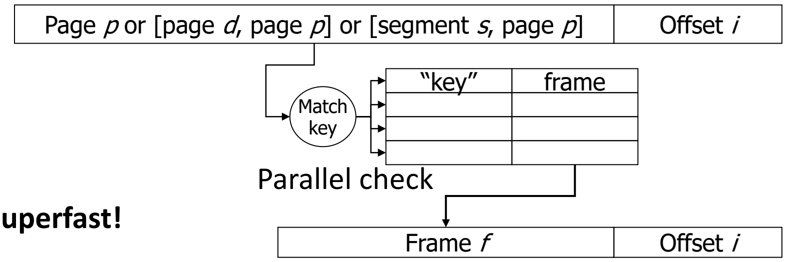
- One table per process
- Table parameters in memory
  - Page table base register
  - Page table size register
- Table elements: Page metadata
  - **V: valid bit**
  - R: referenced bit
  - D: dirty bit
    - If page has been moc
  - Frame: location in phy
  - Perm: access permissions

Valid bit, checkable by hardware, says if the page is in physical memory:
- 1: in memory, use frame field to find where
- 0: not in memory

Memory Access Case 1

- TLB Hit
  - MMU Hardware resolves address
  - lookup in TLB only

1. User accesses a virtual address
2. The upper bits / key find a match in TLB hardware cache
3. The resolution is complete, use TLB value



Superfast!

Memory Access Case 2

- TLB miss: Page table contains valid entry
  - ➢ MMU Hardware resolves address, lookup in TLB and page table

1. User accesses a virtual address
2. The upper bits/key *do* ***not*** find a match in TLB hardware cache
3. The MMU hardware knows where the page table is!
   - ➢ MMU indexes into table, finds frame number

   Fast!
4. MMU loads the TLB and completes address resolution

OS doesn't have to do anything. Its work was done in setting up the table in advance. NO context switch!

Valid vs Invalid Pages

- So far: Valid pages
  - ➢ Much better performance-wise
  - ➢ No OS intervention required

- What if a page is *invalid*, i.e., it's not in memory?
  - ➢ Causes a ***page fault***

Memory Access Case 3

- TLB miss: page table contains *invalid* entry, disk has the page
  - ➢ OS resolves address, lookup in TLB, page table, and disk

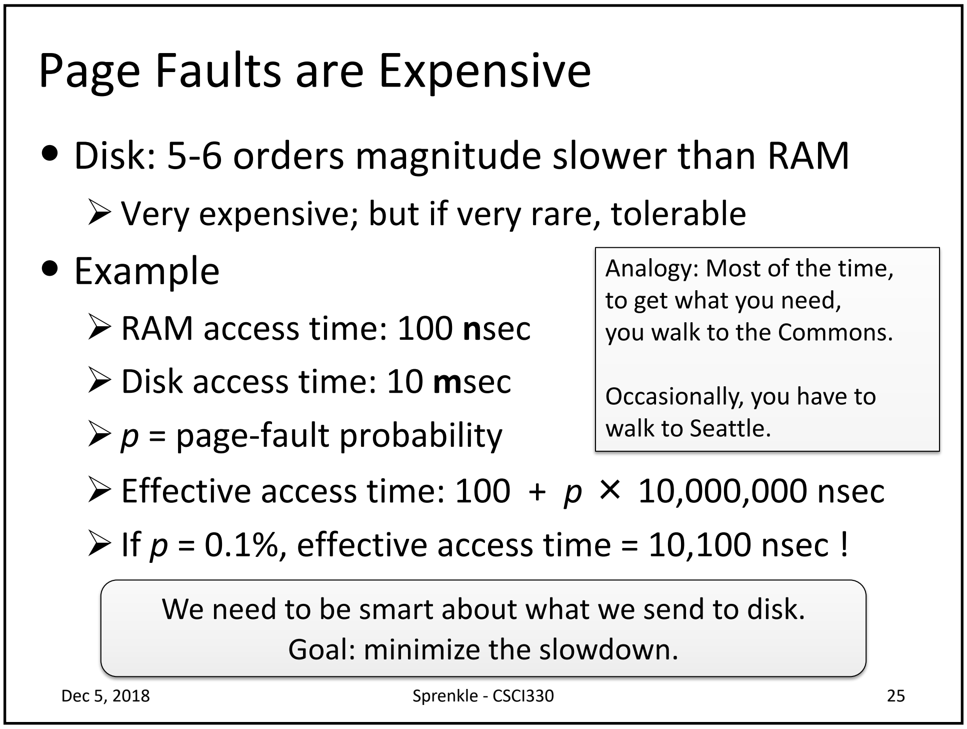
1. User accesses a virtual address
2. The upper bits/key don't find a match in TLB hardware cache
3. The MMU hardware knows where the page table is!
   - ➢ MMU indexes into table, but page table entry is *invalid*…
4. MMU raises exception—**OS gets control of the CPU**
5. OS finds faulting page on disk, brings it into memory, and restarts process from the instruction that faulted

Next time page is accessed*, should be faster!

Memory Access Case 4

- TLB miss: page table contains invalid entry, disk does **not** have page
  - ➢ OS can't resolve address, lookup in TLB, page table, and disk

1. User accesses a virtual address
2. The upper bits / key don't find a match in TLB hardware cache
3. The MMU hardware knows where the page table is!
   - ➢ MMU indexes into table, but page table entry is invalid…
4. MMU raises exception – **OS gets control of the CPU**
5. OS looks for page on disk but *not there!*
   - ➢ It was never allocated!
6. OS terminates the offending process



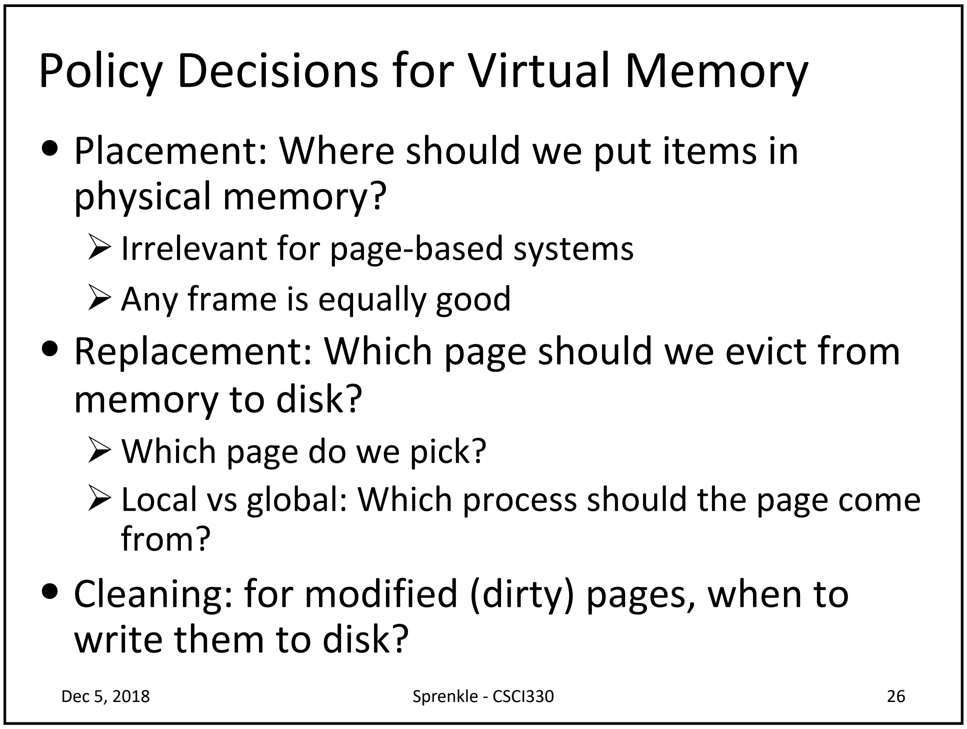
Page Faults are Expensive

- Disk: 5-6 orders magnitude slower than RAM
  - Very expensive; but if very rare, tolerable
- Example
  - RAM access time: 100 **n**sec
  - Disk access time: 10 **m**sec
  - *p* = page-fault probability
  - Effective access time: 100 + *p* × 10,000,000 nsec
  - If *p* = 0.1%, effective access time = 10,100 nsec !

Analogy: Most of the time, to get what you need, you walk to the Commons.

Occasionally, you have to walk to Seattle.

We need to be smart about what we send to disk.
Goal: minimize the slowdown.

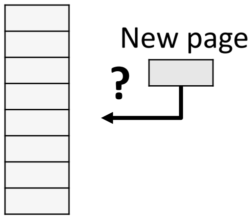


Policy Decisions for Virtual Memory

- Placement: Where should we put items in physical memory?
  - Irrelevant for page-based systems
  - Any frame is equally good
- Replacement: Which page should we evict from memory to disk?
  - Which page do we pick?
  - Local vs global: Which process should the page come from?
- Cleaning: for modified (dirty) pages, when to write them to disk?

Page Replacement

- For now, assume one process and that it has a fixed number of frames
- Problem specification:
  - A *page fault* has just occurred
  - All of the process's frames are full
  - To complete the faulting instruction, one of the existing pages must be evicted to free up a frame
- Eviction: remove the page from a frame
  - put on disk if it isn't already
- Victim: the page that was chosen for eviction



Page Replacement Goals

1. Minimize page faults
   - Achieve good **temporal locality**: reuse of pages within a short period of time
   - (Spatial locality: use of close data elements)
2. Easy to implement and low overhead to manage
   - Don't need a lot of state
   - Better if HW can handle most of requests

Candidate Algorithms

- László Bélády – Hungarian computer scientist who studied this problem for IBM
- Bélády's Optimal Algorithm (a.k.a. Clairvoyant algorithm):
  - ➢ Look ahead into the future and evict the page that won't be used for the longest time
- Why is this worth considering when we clearly can't build it?
  - ➢ Gives us a benchmark
  - ➢ Can't do any better than this

Bélády's Optimal Algorithm

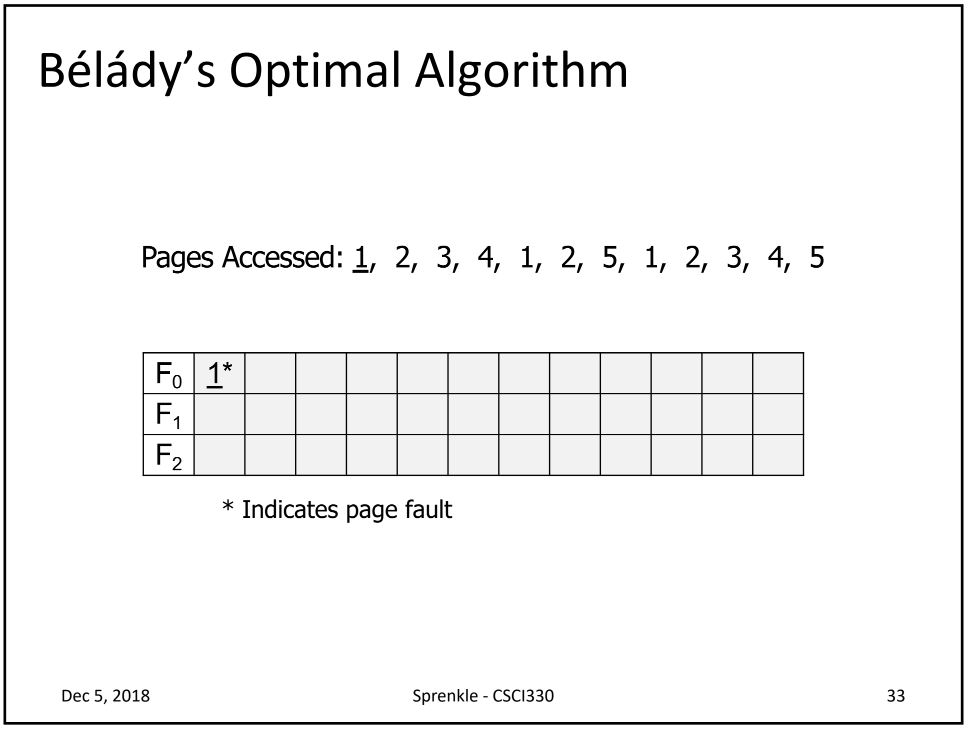
Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

time →

3 frames

F_0												
F_1												
F_2												

* Indicates page fault

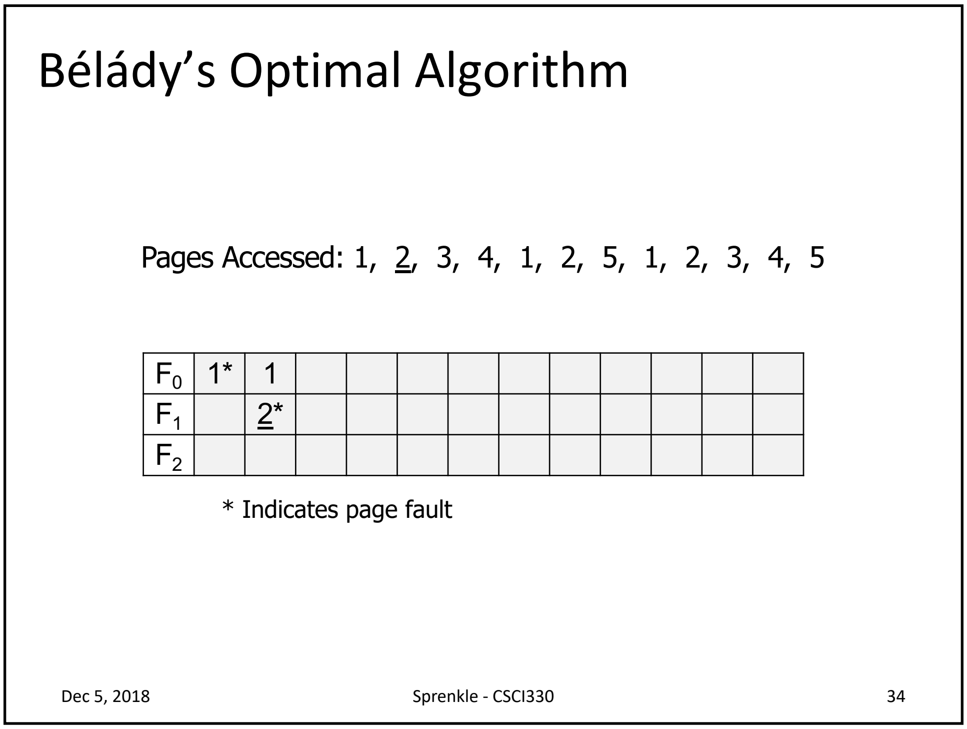


Bélády’s Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*											
F_1												
F_2												

* Indicates page fault



Bélády’s Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1										
F_1		2*										
F_2												

* Indicates page fault

Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1									
F_1		2*	2									
F_2			3*									

* Indicates page fault

Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	?								
F_1		2*	2	?								
F_2			3*	?								

* Indicates page fault

Which frame should we evict to make room for page 4? Why?

A: Frame 0 B: Frame 1 C: Frame 2

Bélády’s Optimal Algorithm

↓ ↓ ↓

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	1								
F_1		2*	2	2								
F_2			3*	4*								

* Indicates page fault

Bélády’s Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	1	1							
F_1		2*	2	2	2							
F_2			3*	4*	4							

* Indicates page fault

Bélády’s Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	1	1	1						
F_1		2*	2	2	2	2						
F_2			3*	4*	4	4						

* Indicates page fault

Bélády’s Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	1	1	1	1					
F_1		2*	2	2	2	2	2					
F_2			3*	4*	4	3	5*					

* Indicates page fault

Bélády's Optimal Algorithm

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	1	1	1	1	1	1	3*	3	3
F_1		2*	2	2	2	2	2	2	2	2	4*	4
F_2			3*	4*	4	3	5*	5	5	5	5	5

* Indicates page fault

# page faults: 7

Candidate Algorithms - Reality

- Can't know the future of page accesses…
- Straightforward algorithm: FIFO
  - Always replace the oldest page

FIFO Replacement

(can't see future…)

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

First three pages start the same way:
fill in free frames.

F_0	1*	1	1									
F_1		2*	2									
F_2			3*									

* Indicates page fault

FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	4*								
F_1		2*	2	2								
F_2			3*	3								

* Indicates page fault

FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	4*	4							
F_1		2*	2	2	1*							
F_2			3*	3	3							

* Indicates page fault

FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	4*	4	4						
F_1		2*	2	2	1*	1						
F_2			3*	3	3	2*						

* Indicates page fault

FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	4*	4	4	5*					
F_1		2*	2	2	1*	1	1					
F_2			3*	3	3	2*	2					

* Indicates page fault

FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	4*	4	4	5*	5	5			
F_1		2*	2	2	1*	1	1	1	1			
F_2			3*	3	3	2*	2	2	2			

* Indicates page fault

FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	4*	4	4	5*	5	5	5		
F_1		2*	2	2	1*	1	1	1	1	3*		
F_2			3*	3	3	2*	2	2	2	2		

* Indicates page fault

FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	4*	4	4	5*	5	5	5	5	
F_1		2*	2	2	1*	1	1	1	1	3*	3	
F_2			3*	3	3	2*	2	2	2	2	4*	

* Indicates page fault

FIFO Replacement

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	1*	1	1	4*	4	4	5*	5	5	5	5	5
F_1		2*	2	2	1*	1	1	1	1	3*	3	3
F_2			3*	3	3	2*	2	2	2	2	4*	4

* Indicates page fault

Total: 9 Page faults.

Analyze FIFO

- Recall our goals:
  - Minimize page faults
  - Easy to implement, low overhead to manage

✓ Easy to implement BUT
- It exhibits poor locality
- It doesn’t manage memory (frames) well

Bélády's Anomaly: FIFO

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	**1***	1	1	**4***	4	4	**5***	5	5	5	5	**5**
F_1		**2***	2	2	**1***	1	1	**1**	1	**3***	3	3
F_2			**3***	3	3	**2***	2	2	**2**	2	**4***	4

9 page faults

F_0												
F_1												
F_2												
F_3												

More memory! Should do better, right…?

Bélády's Anomaly: FIFO

Pages Accessed: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

F_0	**1***	1	1	**4***	4	4	**5***	5	5	5	5	**5**
F_1		**2***	2	2	**1***	1	1	**1**	1	**3***	3	3
F_2			**3***	3	3	**2***	2	2	**2**	2	**4***	4

9 page faults

F_0	**1***	1	1	1	**1**	1	**5***	5	5	5	**4***	4
F_1		**2***	2	2	2	**2**	2	**1***	1	1	1	**5***
F_2			**3***	3	3	3	3	3	**2***	2	2	2
F_3				**4***	4	4	4	4	4	**3***	3	3

10!!! page faults

Candidate Algorithms - Reality

- Can’t know the future of page accesses…

- Straightforward algorithm: FIFO
  - Always replace the oldest page

- Classic cache replacement algorithm: LRU
  - Replace the page that hasn’t been used for the longest time

LRU Replacement

New page sequence!

Pages Accessed: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

First four pages: fill in free frames.

F_0	**2***	2	**2**	2								
F_1		**3***	3	3								
F_2				**1***								

* Indicates page fault

LRU Replacement

Pages Accessed: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

F_0	**2***	2	**2**	2	2							
F_1		**3***	3	3	5*							
F_2				**1***	1							

* Indicates page fault

Looking Ahead

- Project 5 due Friday
- Course evaluations due Sunday
  - ➢ 60% completion → 1% extra credit to OS Project grade
  - ➢ 10% completion increase → additional 1% to OS Project grade