CS 475
Operating Systems

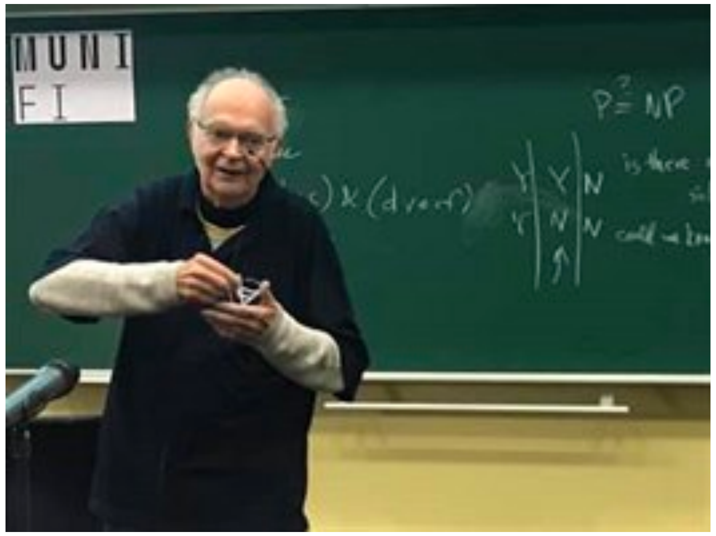
Department of Mathematics
and Computer Science

Lecture 9
Virtual Memory (Part I)

Observation: Knuth's *"90/10 Principle"*

- Modern programs require a lot of memory
  (And demand is growing!)
  - *But "90% of execution time is spent on 10% of the code"*
  - This principle is a rule of thumb about code performance and where to focus optimization efforts.
  - Can you think of some examples of Knuth's principle in the programs you write or use?

Professor Donald Knuth



Observation: Knuth's *"90/10 Principle"*

- Modern programs require a lot of memory (And demand is growing!)
  - *But "90% of execution time is spent on 10% of the code"*

- Examples:
  - **Sorting a large data set from a file:** 90% time spent on open_file() and quicksort() -- maybe just 50 lines combined.
  - **Web app:** 90% time spent on a few SQL queries. All others queries are fast and/or not commonly used.

- Takeaway for memory management: Can processes run on just a few bits of memory (10%) being loaded?

Goals for This Lecture...

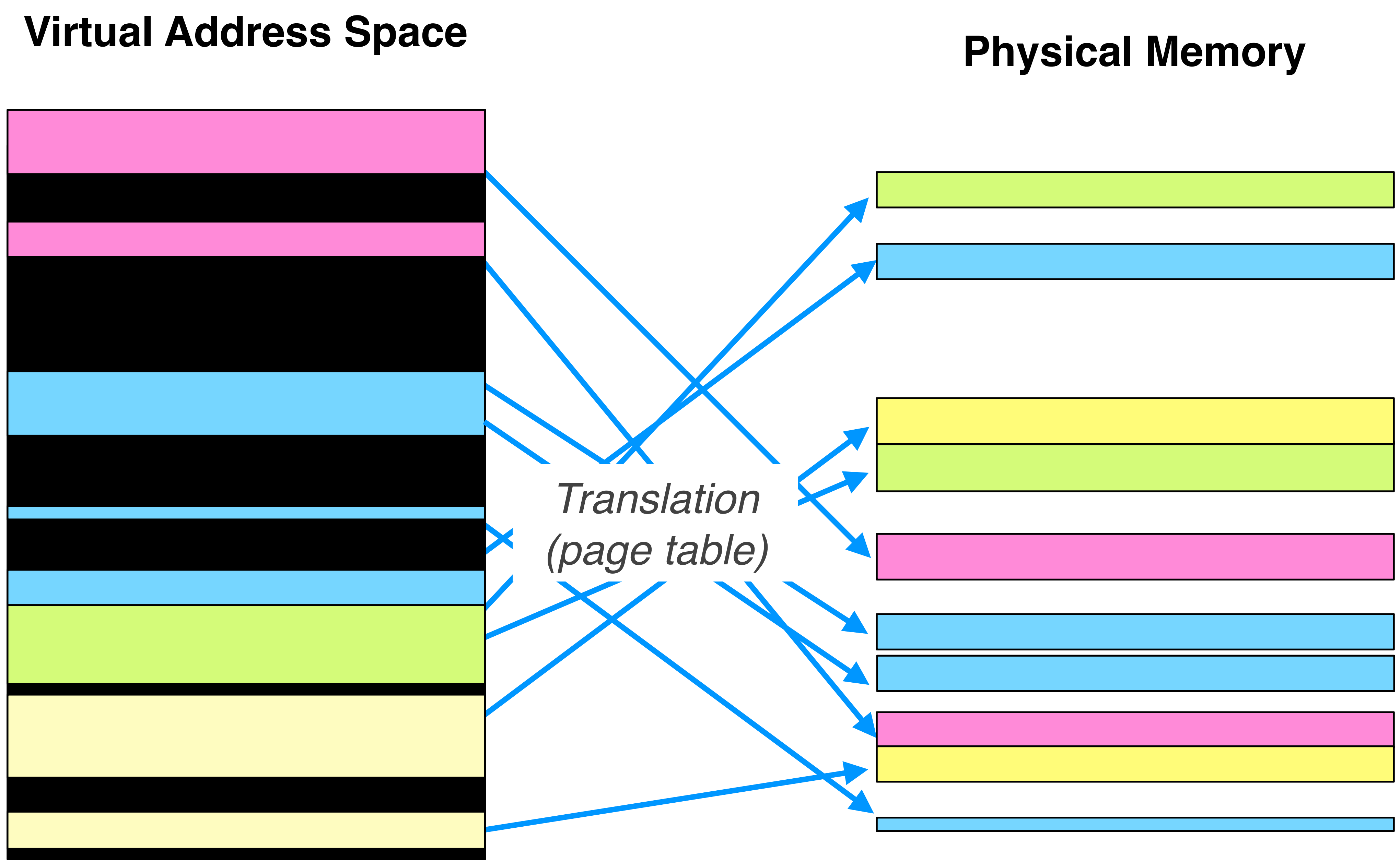
- Demand Paging
  - Motivation
  - Implementation
- Page Replacement Policies
  - FIFO
  - MIN (Optimal)
  - LRU
  - CLOCK
- Memory Allocation

Demand Paging

We don't need to load all pages of a process for it to run most of the time.

- We just need a subset of the process' address space loaded in RAM.
  - We call this the *"Working Set"* (or "Memory Footprint")
  - Leave the remaining pages on disk, and only load pages on access
  - This idea is called *Demand Paging*

Demand Paging Overview



Demand Paging:

A process loads a <u>subset of pages</u> into physical memory to run, leaving behind "holes"

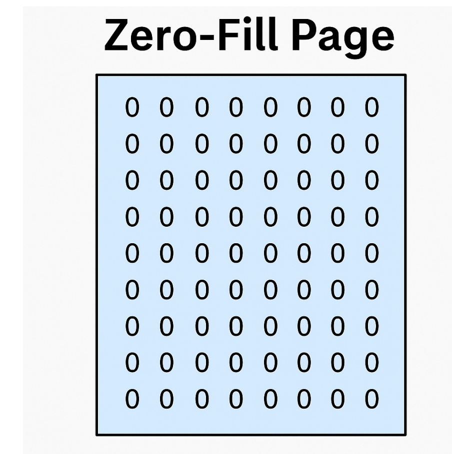
Holes in Process' Virtual Address Space?

- *Holes can also be pages that are on disk*
  - Pages from the code's executable file that haven't been accessed
    - Such as seldom used features
  - Pages that were swapped out to disk during execution
  - These pages are brought in after a page fault

- *Holes can also be pages that have not yet been allocated to process*
  - *e.g.* from growing the program stack
  - *e.g.* `malloc(), calloc()`
  - These pages are initially mapped to the "zero-fill page"

Zero-Fill Page Optimization

- *"Zero-Fill Page"*
  - OS pins a single frame in main memory pre-filled with zeroes (that's 4KB worth of 0s!)
  - Read-only & can be mapped by multiple processes



- Say our process calloc() 3 pages worth of data.
  - OS adds 3 new entries to the page table, marking them valid.
  - Each page table entry points to the frame containing *"zero-fill page"*
    - *This is an optimization for speed! Don't find 3 empty frames and zero them out!*
  - Map pages to "real" frames once a change has been made to the newly allocated pages.

Goals for This Lecture...

- Demand Paging
  - Motivation
  - Implementation
- Page Replacement Policies
  - FIFO
  - MIN (Optimal)
  - LRU
  - CLOCK
- Memory Allocation

Implementing Demand Paging

- Recall the page table structure: Array of page table entries (PTEs)

Page #	V	P	Frame #
[0]			
[1]			
[2]			
[...]			

- A PTE is just an integer (32-bit sequence) that packs information about a particular page:

	Status Bits			
	V	Prot		Frame #
Bits:	*1*	*3*	*unused bits (if any)*	$\log_2$ *num_frames*

Implementing Demand Paging (2)

Status Bits

V	Prot		Frame #
Bits: *1*	*3*	*unused bits (if any)*	$\log_2$ *num_frames*

- Importance of the Valid Bit (V)
  - `V = 1`: page is in memory. The given page number maps to a frame!
  - `V = 0`: page is out on disk's swap space.

- Suppose a process references an invalid page:
  - MMU issues a page-fault, sends a trap to the OS, which invokes the `page_fault_handler()` routine...

Page-Fault Handler Routine

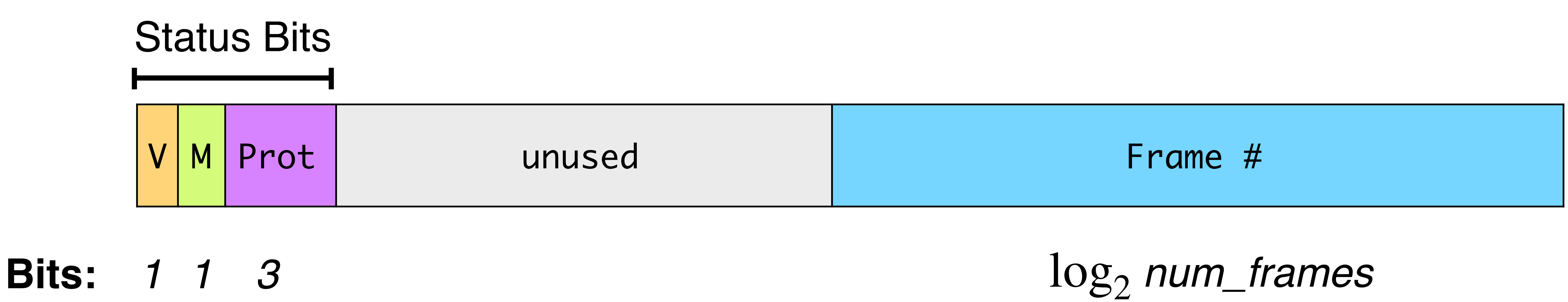
- The OS allocates N frames for a process to use. Process references a page, `new`, which is on disk. So the MMU traps a **page fault**...

1. Place process on the **wait queue**
2. If process is already using all of its N frames:
   - Choose a page `old` to evict from its "working set"
   - Write contents of `old` to disk *(Slow)* ← *Do we have to?*
   - Invalidate `old` (PTE[v] = 0), and flush its TLB entry

   Step 2 is called "page replacement"
3. Load a page `new` from disk to a free frame *(Slow)*
4. Update `new's` PTE to valid (PTE[v] = 1)
5. Put process back on the **ready queue**
6. Re-execute the faulting instruction

Modify Bit "M": Optimizing Page-Fault Handler

- **Observe:** If contents of page wasn't changed, why write it back to disk during page replacement?
  - Borrow an unused bit in the PTE to indicate whether a page had been modified (called the *"Modify bit" - denoted as "M" in a PTE*)



Page-Fault Handler (Optimized)

- The OS allocates N frames for a process to use. Process references a page, `new`, which is on disk. So the MMU traps a page fault...

1. Place process on the **wait queue**
2. If process is already using all of its N frames:

   Step 2 is called "page replacement"

   - Choose a page `old` to evict from its "working set"
   - Write contents of `old` to disk if **PTE[m] == 1** *(Potentially) a Disk Access*
   - Invalidate `old` (PTE[v] = 0; **PTE[m] = 0**), and flush its TLB entry
3. Load new page `new` from disk to a free frame *Disk Access! (Slow)*
4. Update `new's` PTE to valid (PTE[v] = 1)
5. Put process back on the **ready queue**
6. Re-execute the faulting instruction

Example: Page Fault Handling Routine

- Assume:
  - 8-bit virtual and real addresses
  - 4 pages, 6 frames
  - 2-entry TLB
- *Show the contents after Process **B** references its memory address **74.***
- *Assume **Process A's page 0** hasn't been accessed in awhile... so it's a candidate for eviction*

TLB

PID	Page#	Frame #
B	2 (10)	4
A	0 (00)	1

PageTable: Process A

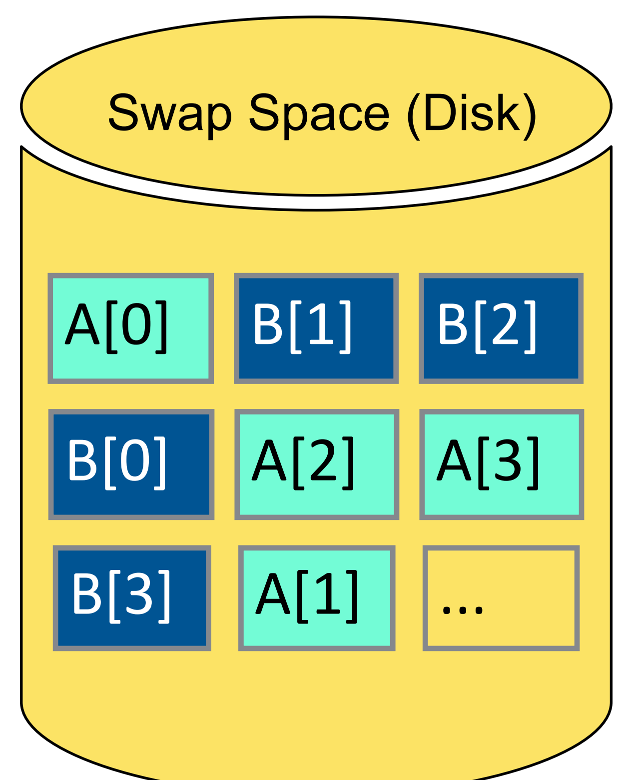
Page #	V	M	Prot	Frame #
0 (00)	1	1		1
1 (01)	1	1		2
2 (10)	0	0		-
3 (11)	1	1		5

PageTable: Process B

Page #	V	M	Prot	Frame #
0 (00)	1	0		3
1 (01)	0	0		-
2 (10)	1	1		4
3 (11)	1	1		0

Physical Memory

Frame#	Content
0	B[3]
1	A[0]
2	A[1]
3	B[0]
4	B[2]
5	A[3]



Solution (Red = Change)

- Assume:
  - v_addr (6 bits), real_addr (8 bits)
  - 4 pages, 6 frames
  - 2-entry TLB
- *Show the contents after Process **B** references its memory address **74.***
- *Assume **Process A's page 0** hasn't been accessed in awhile... so it's a candidate for eviction*

TLB

PID	Page#	Frame #
B	2 (10)	4
B	1 (01)	2

PageTable: Process A

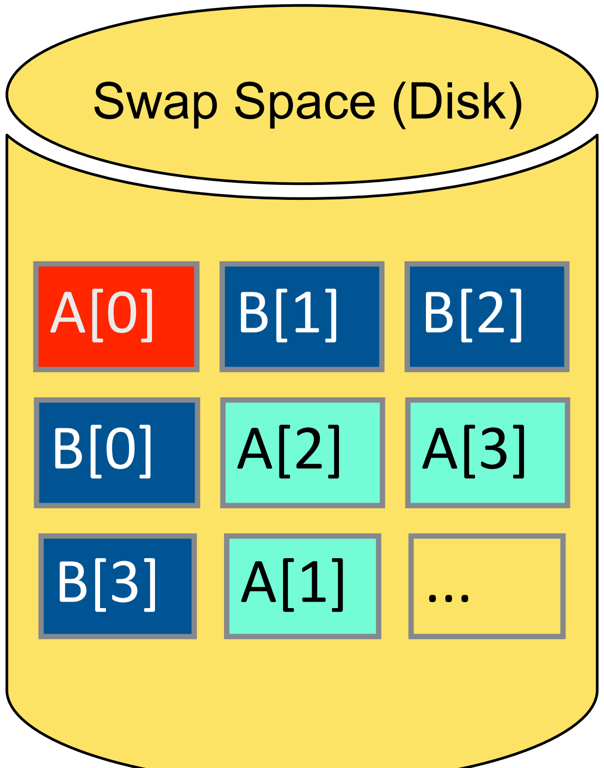
Page #	V	M	Prot	Frame #
0 (00)	0	0		-
1 (01)	1	1		2
2 (10)	0	0		-
3 (11)	1	1		5

PageTable: Process B

Page #	V	M	Prot	Frame #
0 (00)	1	0		3
1 (01)	1	0		1
2 (10)	1	1		4
3 (11)	1	1		0

Physical Memory

Frame#	Content
0	B[3]
1	B[1]
2	A[1]
3	B[0]
4	B[2]
5	A[3]



Goals for This Lecture...

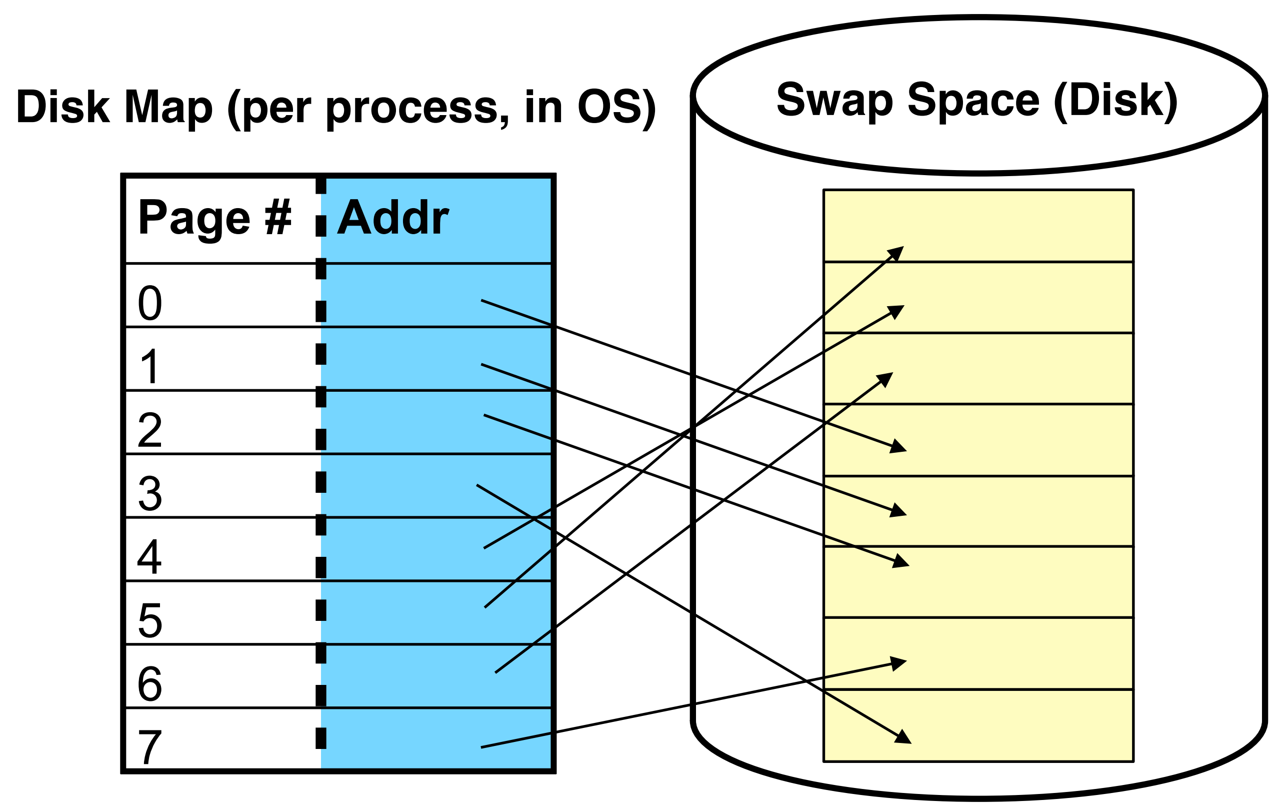
- Demand Paging
  - Motivation
  - Implementation
    - Swap space
- Page Replacement Policies
  - FIFO
  - MIN (Optimal)
  - LRU
  - CLOCK
- Memory Allocation

Mapping Pages to the Swap Space

- If a page is on disk, how does the OS know *__where__* to go fetch in the first place?
  - Keep a *Disk Map (or Swap Map)* for each process too
  - Maps a page to a "block" on the disk's swap space
    - (Generally, we *combine* the disk map with the page table by re-purposing the bits in an invalid PTE)

Page Table (per process, in OS)

Page #	V	M	P	Frame #
0	1			
1	0			
2	1			
3	0			
4	1			
5	1			
6	0			
7	1			



Pages (called "blocks" on disk)

Loading a Process 1: On `fork()`

- Step 1: Create a **PCB**, **Page Table**, and **Disk Map** for the process
  - Focus on memory management structures

PCB

```
pid
parent_pid
userid
time
priority
regs
...
```

Page Table

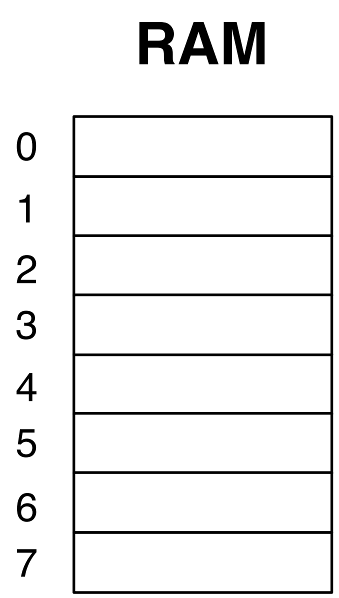
Page #	V	M	P	Frame #
0	0	0	0	0
...	0	0	0	0
n-2	0	0	0	0
n-1	0	0	0	0

Disk Map

Page #	Block Addr (on disk)
0	NULL
...	NULL
n-2	NULL
n-1	NULL

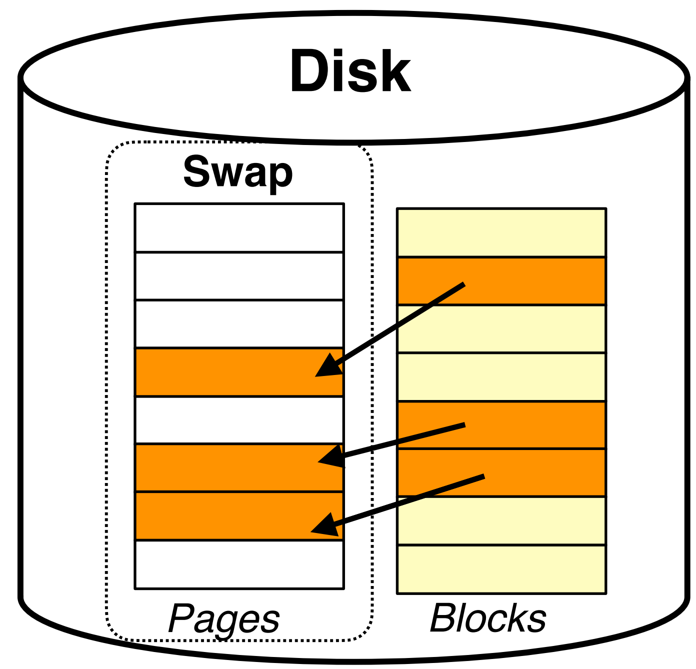
Loading a Process 2a: On exec()

- Step 2: Copy the program executable (code and data segment) into swap space.
  - Page size = frame size = disk-block size



Disk Map

Page #	Addr
0	NULL
...	...
n-2	NULL
n-1	NULL

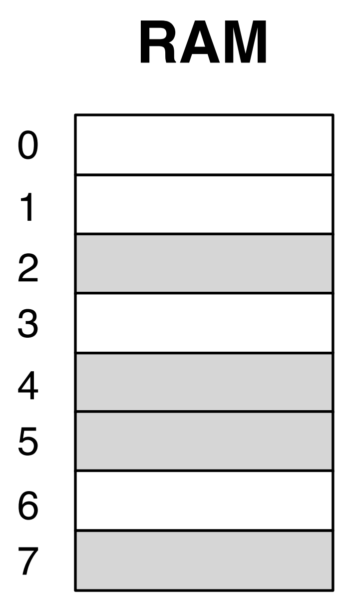


Page Table

Page #	V	M	P	Frame #
0	0	0	0	0
...	0	0	0	0
n-2	0	0	0	0
n-1	0	0	0	0

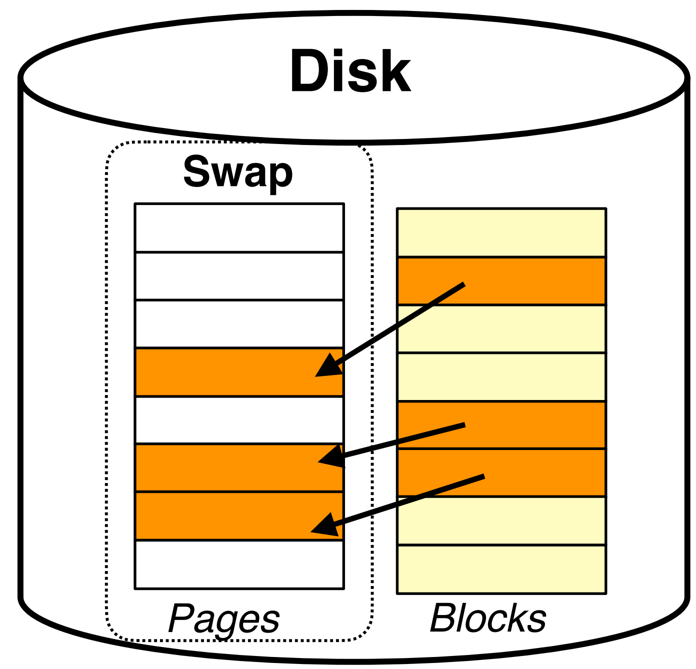
Loading a Process 2b: On `exec()`

- Step 3: OS reserves an initial number of frames for each process.
  - Suppose the OS gives the new process **4 frames** in RAM (in grey)



Disk Map

Page #	Addr
0	NULL
...	
n-2	NULL
n-1	NULL

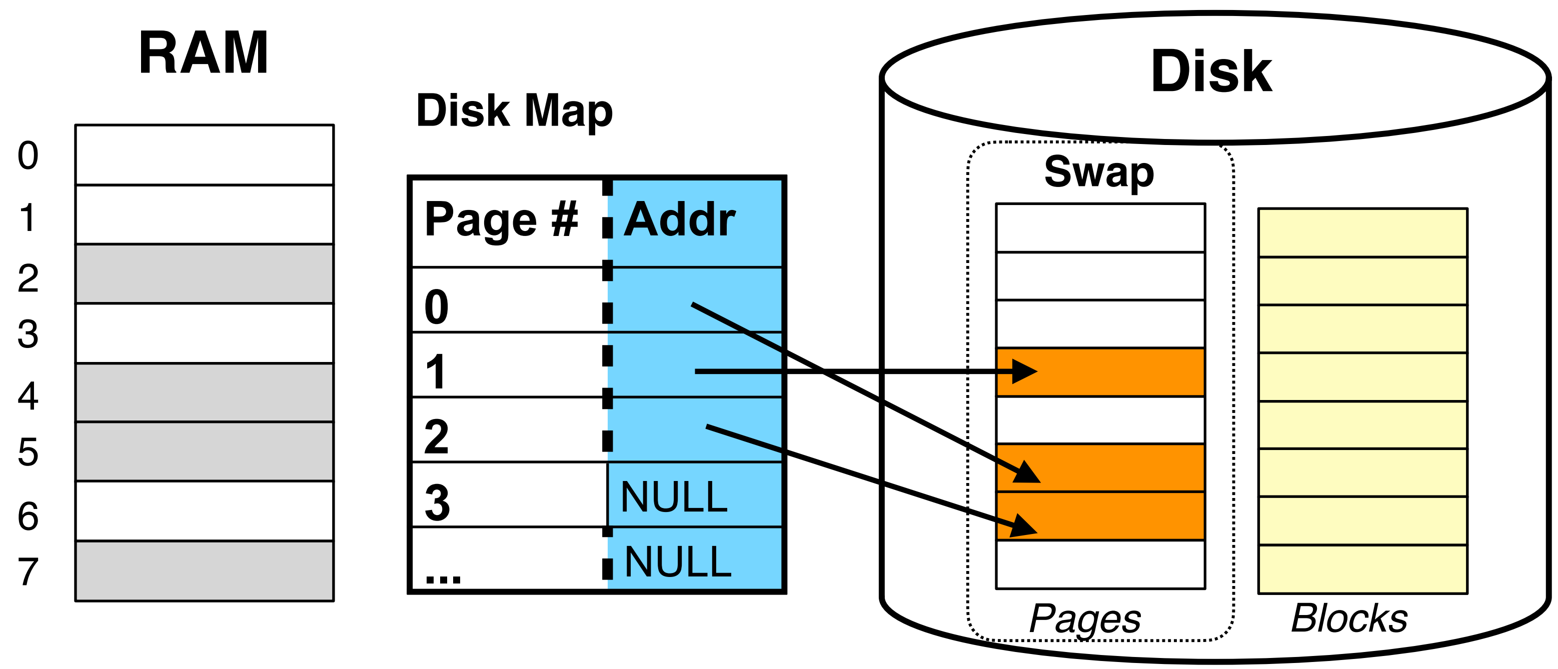


Page Table

Page #	V	M	P	Frame #
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
...	0	0	0	0

Loading a Process 2c: On exec( )

- Step 4: Partially populate the process' *Disk Map*
  - Map to executable in swap space. All other entries still point to NULL
  - Assume code and data take up 3 pages (in **orange**)

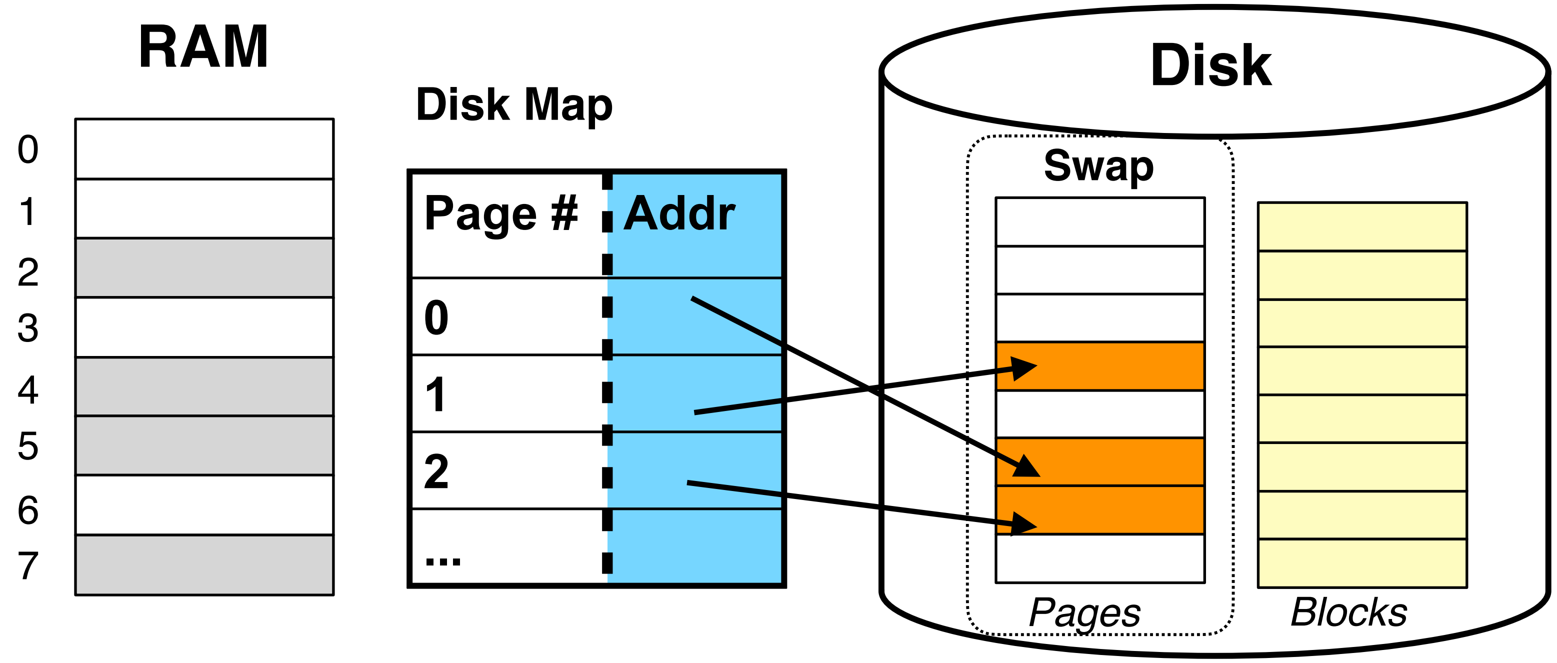


Page Table

Page #	V	M	P	Frame #
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
...	0	0	0	0

Running a Process

- Step 5a: Begin code execution (notice, frames in RAM are still empty!)
  - Remember, CPU *thinks* code starts at some low address in page 0
  - Page is invalid, trap page-fault, run page_fault_handler()

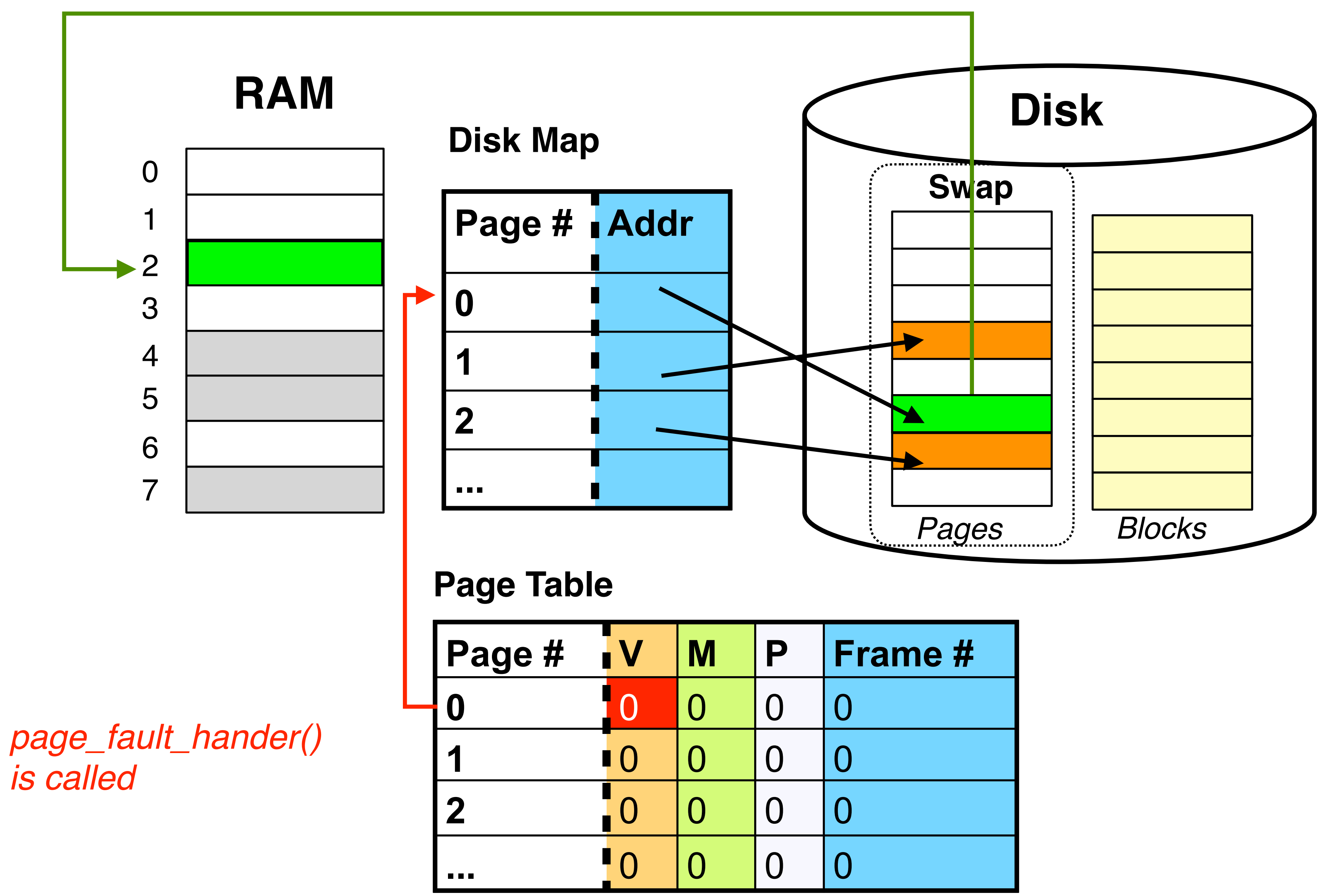


Page Table

Invalid!
Trap a page fault

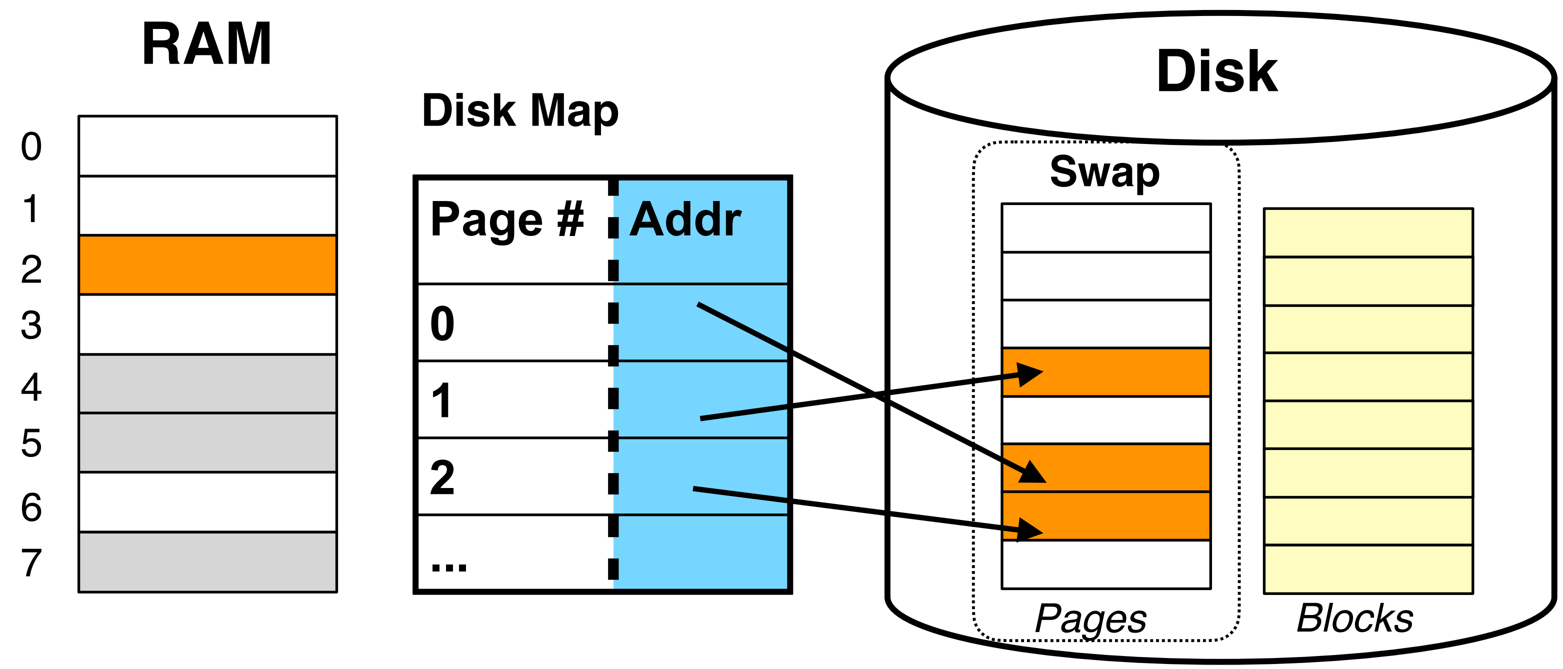
Page #	V	M	P	Frame #
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
...	0	0	0	0

- Step 5b: Page Fault!
  - OS `page_fault_handler()` looks up page in Disk Map, grabs a free frame, and loads the page in the free frame!



Running a Process

- Step 5c: Re-issue the faulting instruction.
  - Update the page table entry. Re-execute instruction.
  - Repeat if necessary for all other memory accesses



Page Table

Update here!

Page #	V	M	P	Frame #
0	1	0	...	2
1	0	0	0	0
2	0	0	0	0
...	0	0	0	0

Demand Paging Summary

- Be lazy! Load pages from disk only as needed
  - May also need to kick pages out to make room in RAM

- Repurpose physical memory to be a *cache* for pages on disk
  - When process starts: many page faults! (Slow)
    - "Compulsory misses" -- nothing yet loaded in main memory.
  - Fewer page faults after process runs for some time
    - Principle of locality kicks in

Demand Paging Summary

- Pros: "Virtual Memory"
  - Processes think they have as much memory as are addressable
    - *e.g.*, 32-bit addresses $\Longrightarrow 2^{32}$ bytes can be addressed, or 4 GB
    - *e.g.*, 64-bit addresses $\Longrightarrow 2^{64}$ bytes can be addressed, or 16 EB (ExaBytes)
    - *Reality check: Most systems today have around 16 to 32 GB of RAM that has to be shared among all processes + OS!*
  - *"Virtual?" Try to malloc() more memory than is physically available -- it'll work!*

Demand Paging Summary

- Pros: "Virtual Memory"
  - Processes think they have as much memory as are addressable
  - Faster loading and switching
    - Only load the pages process actually uses, instead of all pages right from the start
    - Fewer unused "junk" pages cluttering up main memory. Easier to find free frames.
  - More processes can be loaded and run concurrently if just a subset of their pages need to be loaded to run!

Demand Paging Summary

- Treat the memory like a cache for the disk
- Process runtime performance is directly tied to page-fault rate
  - Page faults occur when a desired page is invalid (on disk)
- How to reduce page faults?
  - Page fault = "miss" in memory
  - What causes cache misses (the 3 C's):
    - ***Compulsory Miss*** *- cache starts cold 🥶 The first few accesses will be misses.*
    - ***Capacity Miss*** *- no more room in cache. Evicted a page that's used in the future.*
    - ***Conflict Miss*** *- there's enough room, but multiple pages map to the same frame.*
      - *(This case doesn't happen in paging, as there's no forced mapping to a frame)*

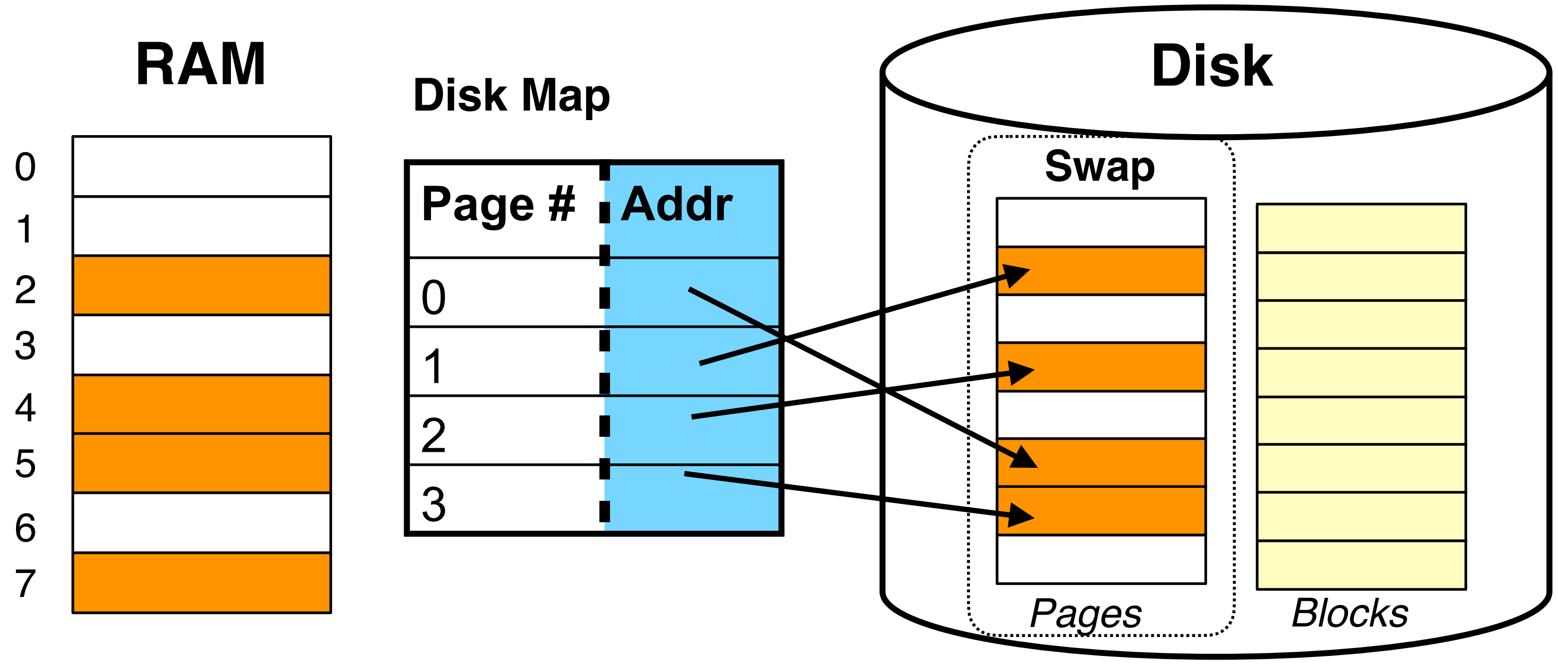
Goals for This Lecture...

- Demand Paging
  - Motivation
  - Implementation
- Page Replacement (Eviction) Policies
  - FIFO
  - MIN (Optimal)
  - LRU
  - CLOCK
- Memory Allocation

What if a Process' Frames Fill up?

- Need another page, but the 4 frames allocated to this process are filled!
  - *We need to evict or more from RAM! But which one(s)?*

4 frames are all used up!



Page Table

Page #	V	M	P	Frame #
0	1	0	..	2
1	1	1	..	7
2	1	1	..	4
3	1	1	..	5
...	...	...	...	...

Caching Analogy

- We'll use closets as an analogy to caches.
  - Item of clothing = Page
  - **Primary closet** are the clothes you can carry.
    - Smaller capacity, but access is fast **(RAM)**
  - **Secondary closet** is in your dorm room.
    - Larger capacity. It can store every item of clothing you have.
    - Access is slow **(Swap Disk)**

- *Replacement Problem:* Your hands are full, but you need to grab a new item of clothing. What do you replace and leave in the dorm?
  - **Goal:** Minimize future trips back to your dorm.

Random Replacement Policy

Random Replacement Policy

1. For each process,
   a. Allocate N frames in physical memory
   b. Maintain a list of allocated pages
2. On page fault <u>and</u> no more free frames,
   a. Choose a random frame and throw out its page.

- Full closet analogy: Kick a random piece of clothing out to the dorm. *Hope* we won't need to wear it soon.
- Evaluation
  - But *very* fast and easy to implement (often used for TLB)

FIFO Replacement Policy

First-In-First-Out (FIFO)

1. For each process,
   a. Allocate N frames in physical memory
   b. Maintain a FIFO queue of frames
2. On page fault and no more free frames allocated to the process:
   a. Evict the contents of the frame sitting at the head of the queue
   b. Enqueue the new frame

- Kick out the piece of clothing that you've been holding onto the longest.

FIFO Example

- Suppose the process is allocated N=3 frames
  - 5 virtual pages: A, B, C, D, E
  - Access (Reference) Pattern: A B C D A B E A B C D E

Time →

Access Pattern	**A**	**B**	**C**	**D**	**A**	**B**	**E**	**A**	**B**	**C**	**D**	**E**
Frame 1	A	A	A	D	D	D	E	E	E	E	E	E
Frame 2		B	B	B	A	A	A	A	A	C	C	C
Frame 3			C	C	C	B	B	B	B	B	D	D

Number of Page Faults = ??

FIFO Example

- Suppose the process is allocated N=3 frames
  - 5 virtual pages: A, B, C, D, E
  - Access (Reference) Pattern: A B C D A B E A B C D E

Time →

Access Pattern	**A**	**B**	**C**	**D**	**A**	**B**	**E**	**A**	**B**	**C**	**D**	**E**
Frame 1	A	A	A	D	D	D	E	E	E	E	E	E
Frame 2		B	B	B	A	A	A	A	A	C	C	C
Frame 3			C	C	C	B	B	B	B	B	D	D

Number of Page Faults = 9 *Hits = 3* *Hit-Rate = 3/12 = 0.25*

FIFO Example 2

- Same setup, but process is **now allocated 4 frames**
  - Same Access Pattern: A B C D A B E A B C D E

Time →

Access Pattern	**A**	**B**	**C**	**D**	**A**	**B**	**E**	**A**	**B**	**C**	**D**	**E**
Frame 1	A	A	A	A	A	A	E	E	E	E	D	D
Frame 2		B	B	B	B	B	B	A	A	A	A	E
Frame 3			C	C	C	C	C	C	B	B	B	B
Frame 4				D	D	D	D	D	D	C	C	C

Number of Page Faults = 10!!! *Hits = 2* *Hit-Rate = 2/12 = 0.167*

Evaluation of FIFO

- Pros:
  - Fair: Every page gets the same amount of time to remain in memory
  - Fast, easy to implement
- Cons:
  - Old pages don't mean they're less useful!
    - Once a page is referenced, it doesn’t go to back of FIFO queue!
  - Suffers from Belady's Anomaly (1969):
    - Performance might worsen as memory allocation increases!
    - Title: *"An anomaly in space-time characteristics of certain programs running in a paging machine*." *(pdf)*

Lazlo Belady (1928-2021)

MIN (Provably Optimal - Minimal Page Faults)

MIN Replacement Policy

1. For each process,
   a. Allocate N frames in physical memory for it
   b. Maintain a *magical queue* of allocated frames
      i. It orders frames by seeing into the future
      ii. Frame to be used farthest in the future is placed in the head
2. On page fault and no free frames,
3. Throw out page that won't be used for the longest period of time

- Full closet: Kick out the clothing that you'll use farthest in the future

MIN Example

- Suppose process is allocated 3 frames
  - Access Pattern: A B C A B D A D B C B

Time →

Access Pattern	A	B	C	A	B	D	A	D	B	C	B
Frame 1	A	A	A	A	A	A	A	A	A	C	C
Frame 2		B	B	B	B	B	B	B	B	B	B
Frame 3			C	C	C	D	D	D	D	D	D

Number of Page Faults = 5 *Hits = 6* *Hit-Rate = 6/11 = 0.55*

Least Recently Used (LRU)

- Can't look into future? Approximate MIN by examining the past.

Least Recently Used (LRU)

1. For each process,
   a. Allocate N frames in physical memory to it
   b. Maintain a list of allocated frames
2. Associate *Timestamp* with each allocated frame
3. Every time a page is referenced, update its frame's *Timestamp*
4. On page fault and no free frames, throw out frame with "**coldest**" timestamp

- Full closet: Kick out the clothing I wore the longest time ago.

LRU Example

- So LRU is supposed to approximate MIN
  - 3 frames allocated to this process, 4 pages used by process
  - Access Pattern: A B C A B D A D B C B

Time →

Access Pattern	A	B	C	A	B	D	A	D	B	C	B
Frame 1	A	A	A	A	A	A	A	A	A	C	C
Frame 2		B	B	B	B	B	B	B	B	B	B
Frame 3			C	C	C	D	D	D	D	D	D

Only 5 page faults! This was MIN's result!

LRU Example (Oh no!)

- When would LRU fail?
  - 3 frames allocated to this process, 4 pages used by process
  - Access Pattern: A B C D A B C D A B C D

Time →

Access Pattern	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**
Frame 1	A	A	A	D	D	D	C	C	C	B	B
Frame 2		B	B	B	A	A	A	D	D	D	C
Frame 3			C	C	C	B	B	B	A	A	A

Number of Page Faults = 12 (This was the worst case)

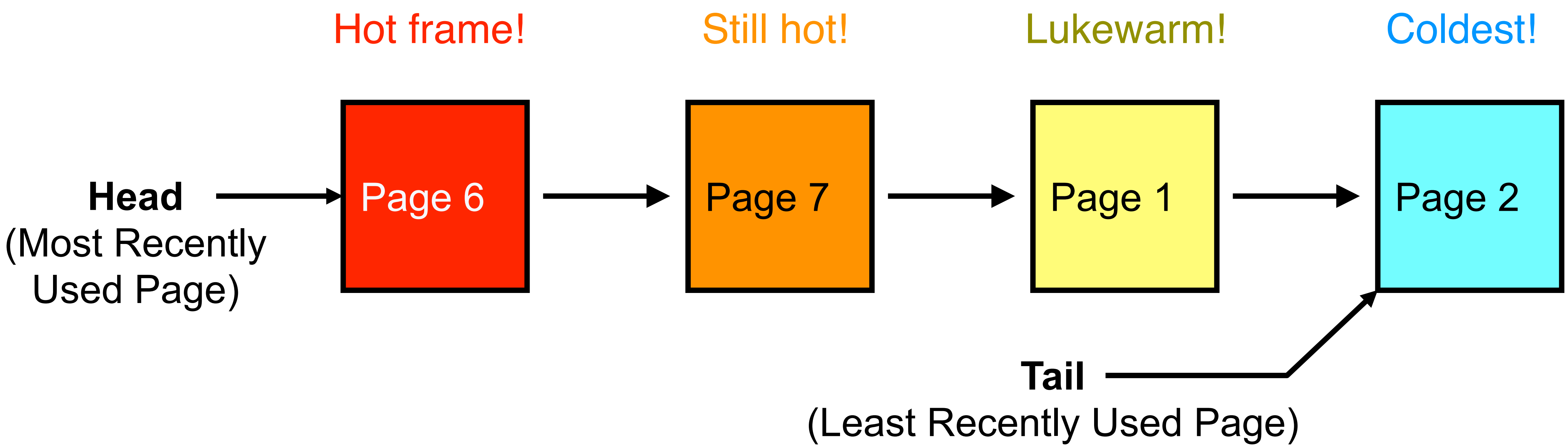
LRU Implementation 1: Page Table Timestamp

- First try...
  - Add a "timestamp" to the page table
  - Whenever a page is referenced, update its timestamp in its entry.

Page #	Last Ref	V	M	Frame #
0 (00)	now - 4	1	1	1
1 (01)	now	1	1	2
2 (10)	now - 9	1	0	7
3 (11)	...	1	1	5

- The good: Updating the timestamp is $O(1)$
  - (By doing translation, you've *already* found the entry in the page table!)
- The bad:
  - A timestamp would eat up some bits in the page table entry.
  - Every page fault needs an $O(n)$ scan of the table to find **coldest page**

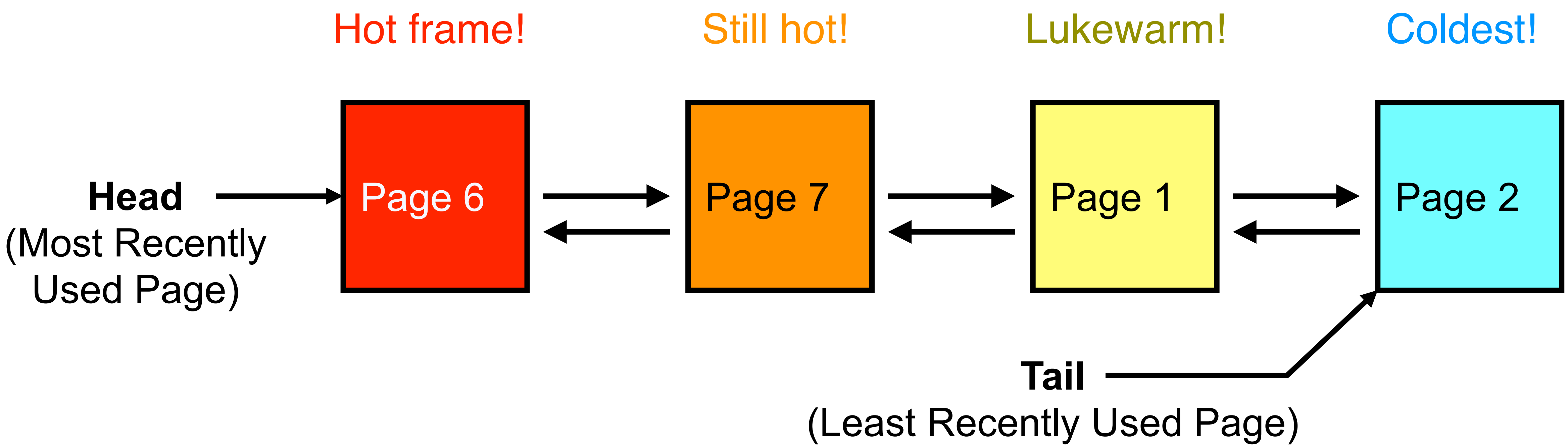
LRU Implementation 2: Singly Linked List



- A LinkedList can track the order of usage!
- Good: Fast eviction. **Coldest frame** located in $O(1)$ time
  - But you also need to update the new tail: $O(n)$ memory accesses (yikes!)
- Bad: For every page reference $\rightarrow$ Move it to the head (It's hot now!)
  - Find the referenced frame in list: $O(n)$ memory accesses

LRU Implementation 2: Doubly Linked List?

UNIVERSITY of PUGET SOUND



- Good: Fast eviction. **Coldest frame** located in $O(1)$ time
  - But you also need to update the new tail: $O(1)$ - ask the victim node for its **prev**!
- Bad: For every page reference $\rightarrow$ Move it to the head (It's hot now!)
  - Find the referenced frame in list: $O(n)$ memory accesses still

Evaluation of LRU

- Pros:
  - On average, LRU approximates MIN
  - Exploits temporal locality of memory accesses, and is fair:
    - Kicks out the page that hasn't been used the longest
      - Don't conflate "least recently used" with "oldest"
  - Does not suffer from Belady's Anomaly
- **Cons:**
  - LRU is just too demanding to be used for real-time caching/paging systems
  - Eviction is just too slow! We can't absorb an $O(n)$ time overhead.
  - We *can* make everything $O(1)$ but at the memory overhead is prohibitive!

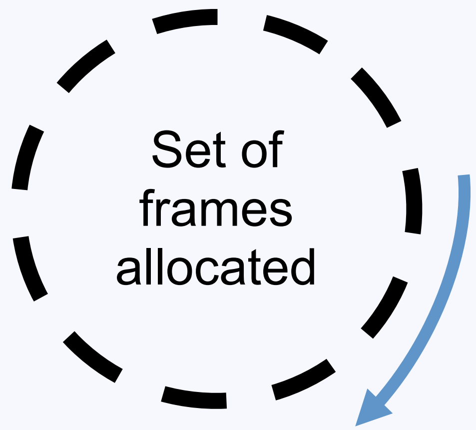
Clock Replacement Policy

- What makes LRU bad? Either because:
  - The search for the least recently used page, or
  - The search for the page that was just accessed, or
  - Memory overhead to make those searches fast!

- Must we find the ***coldest* page?**
  - What if we relaxed that constraint?
  - Evict ***any cold page*** (*i.e.*, Clock ~ LRU ~ MIN)
  - Most modern OS implement some variant of Clock
  - Clock first appeared in Multics
    - Link to paper: http://www.multicians.org/paging-experiment.pdf

Clock (or "2nd Chance") Policy

Organize frames in a circular, clockwise fashion



a. The clock hand points to a frame.

On page fault <u>and</u> process has no free frames

b. If the frame holds a hot page, then set the page to cold

(Gives the page 2nd chance to survive in the frame)

c. Otherwise, evict this page from the frame and replace with new page

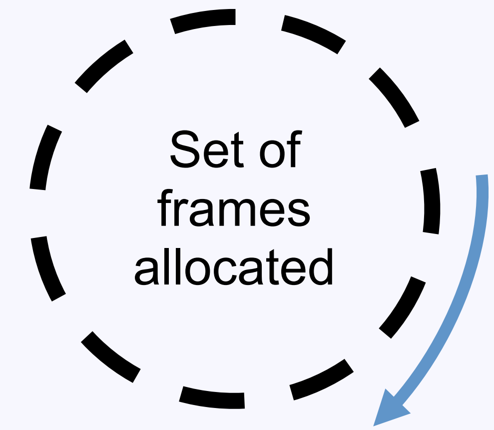
d. Advance the clock hand to the next frame

- How to know if a page is hot or cold?

Clock (or "2nd Chance") Policy (2)

Organize frames in a circular, clockwise fashion

a. The clock hand points to a frame.



On page fault and process has no free frames

b. If the frame holds a hot page, then set the page to cold

(Gives the page 2nd chance to survive in the frame)

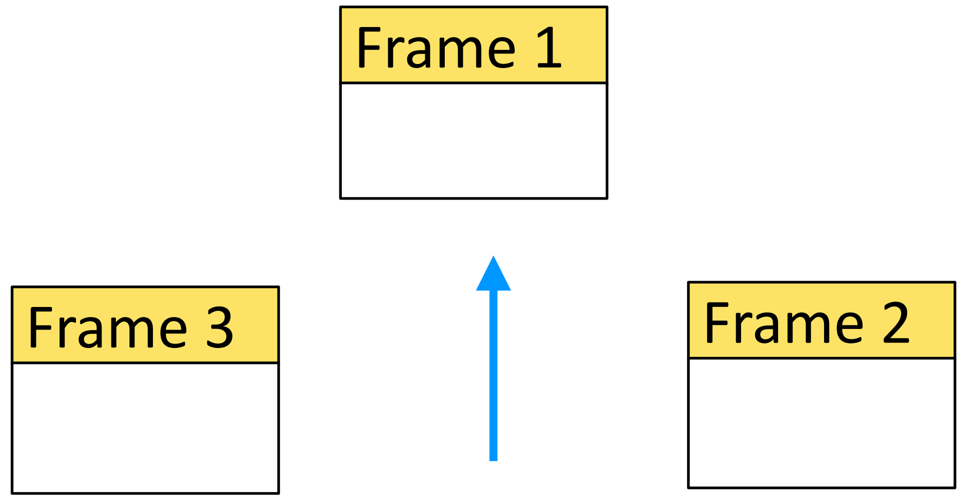
c. Otherwise, evict this page from the frame and replace with new page

d. Advance the clock hand to the next frame

- How to remember if a page is hot or cold?
  - Add a "reference bit" (R) to the page table entry (R=1 hot , R=0 cold)
  - Set R=1 every time the page is referenced!

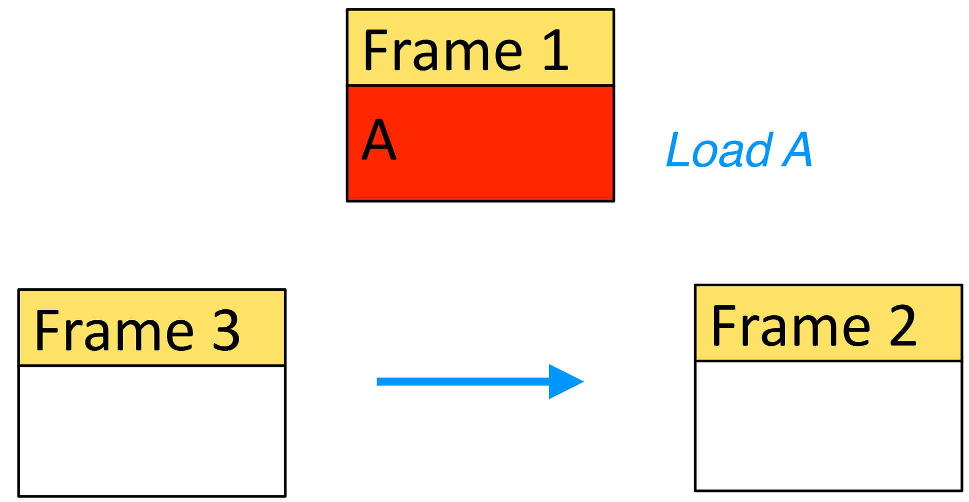
V	M	R	P	unused	Frame #

Clock Example: 3 Frames Allocated to Process



Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1									
Frame 2									
Frame 3									
Reference Bit (R)									
For A									
For B									
For C									
For D									

Clock Example: 3 Frames Allocated to Process

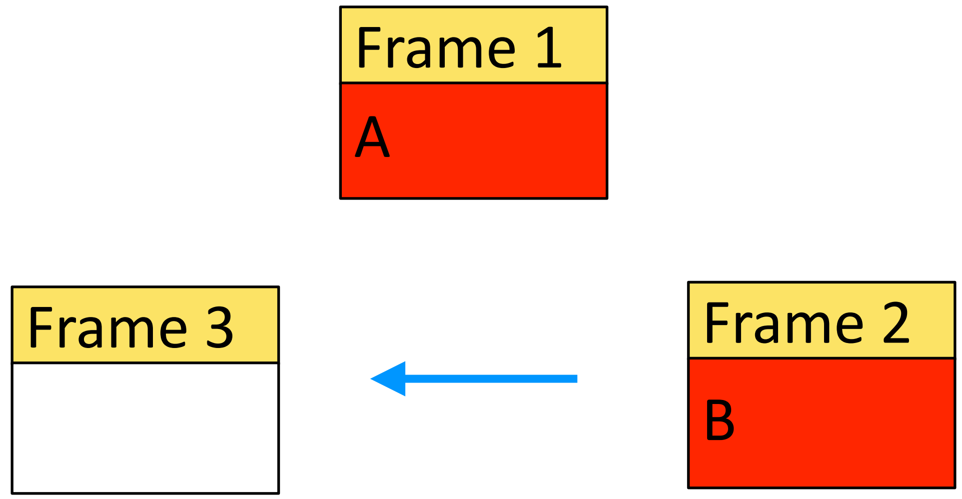


Compulsory miss.
Advance hand.

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A								
Frame 2									
Frame 3									
Reference Bit (R)									
For A	1								
For B									
For C									
For D									

Clock Example: 3 Frames Allocated to Process

UNIVERSITY of PUGET SOUND Est. 1888

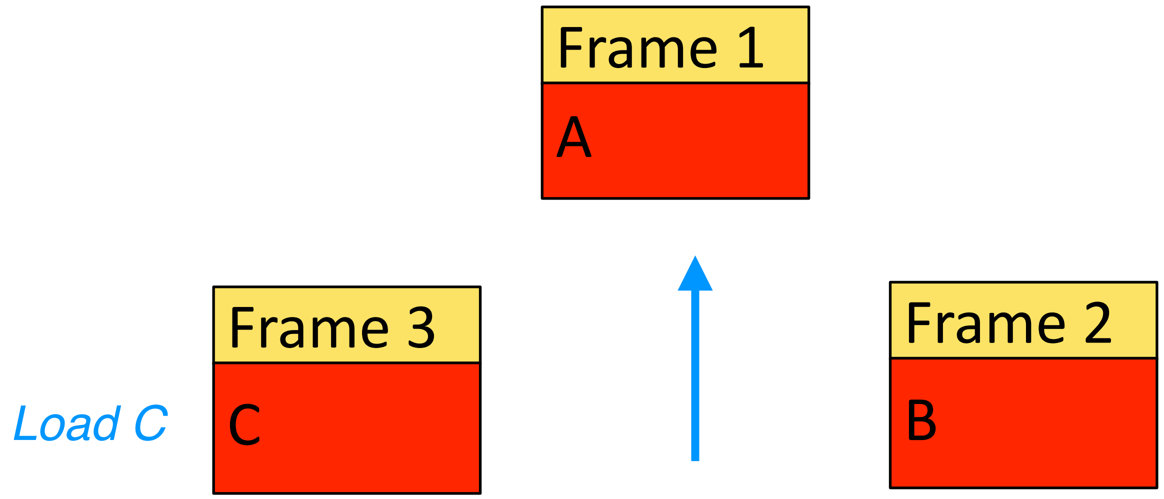


Compulsory miss.
Advance hand.

Load B

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A							
Frame 2		B							
Frame 3									
Reference Bit (R)									
For A	1	1							
For B		1							
For C									
For D									

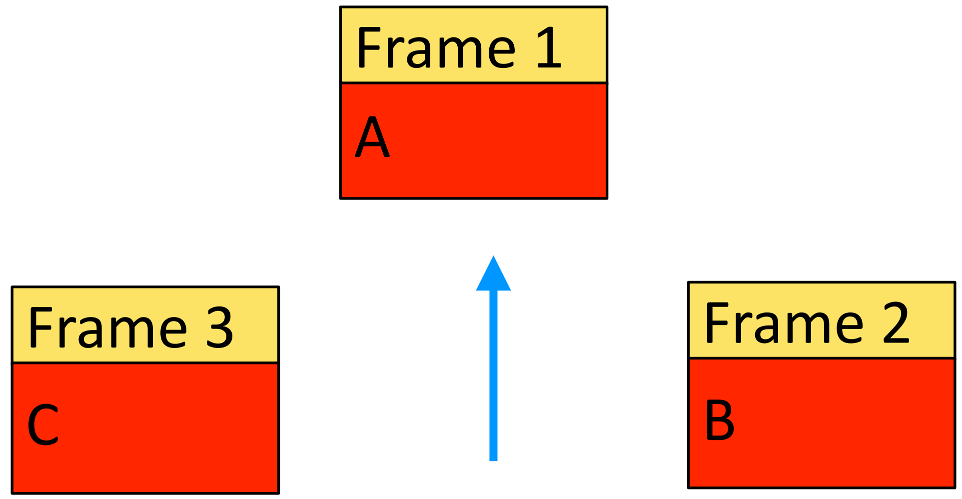
Clock Example: 3 Frames Allocated to Process



Compulsory miss.
Advance hand.

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A						
Frame 2		B	B						
Frame 3			C						
Reference Bit (R)									
For A	1	1	1						
For B		1	1						
For C			1						
For D									

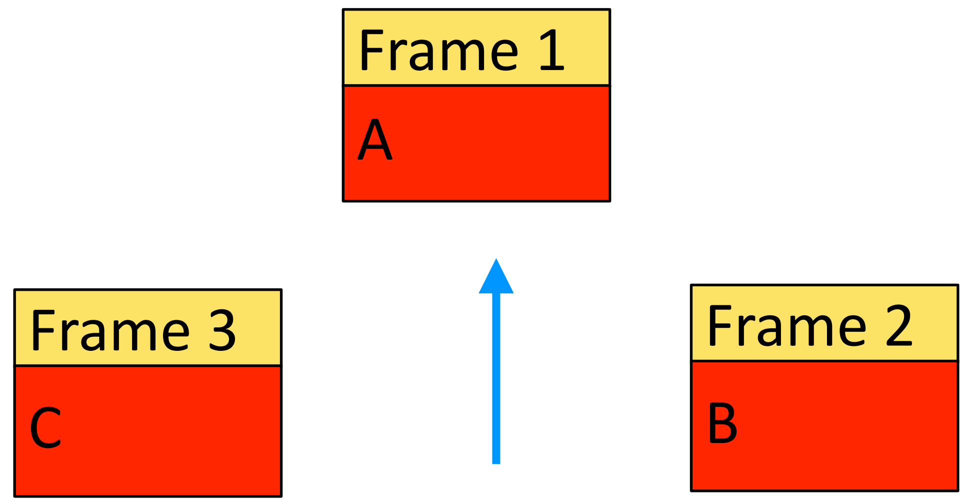
Clock Example: 3 Frames Allocated to Process



A is a Hit!
Don't advance.

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A					
Frame 2		B	B	B					
Frame 3			C	C					
Reference Bit (R)									
For A	1	1	1	1					
For B		1	1	1					
For C			1	1					
For D									

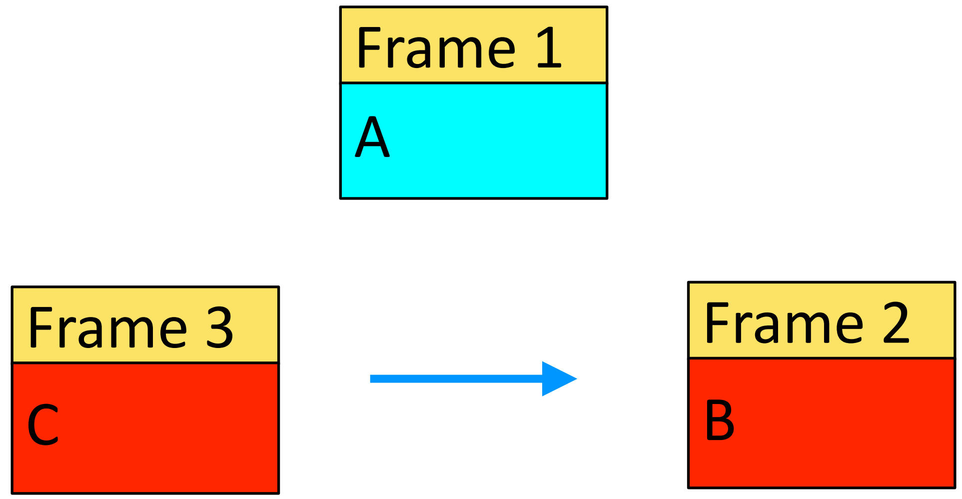
Clock Example: 3 Frames Allocated to Process



B is a Hit!
Don't advance.

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A	A				
Frame 2		B	B	B	B				
Frame 3			C	C	C				
Reference Bit (R)									
For A	1	1	1	1	1				
For B		1	1	1	1				
For C			1	1	1				
For D									

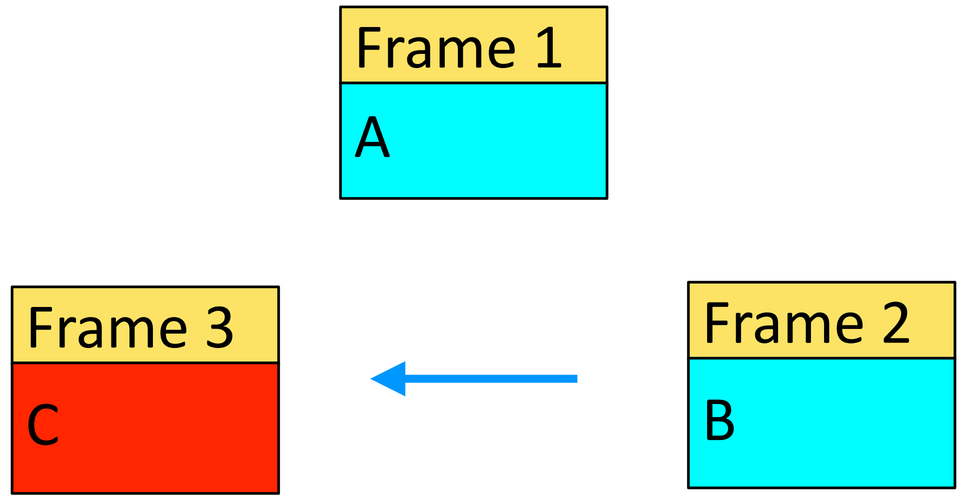
Clock Example: 3 Frames Allocated to Process



Set A to cold.
Advance hand.

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A	A				
Frame 2		B	B	B	B				
Frame 3			C	C	C				
Reference Bit (R)									
For A	1	1	1	1	1	0			
For B		1	1	1	1				
For C			1	1	1				
For D									

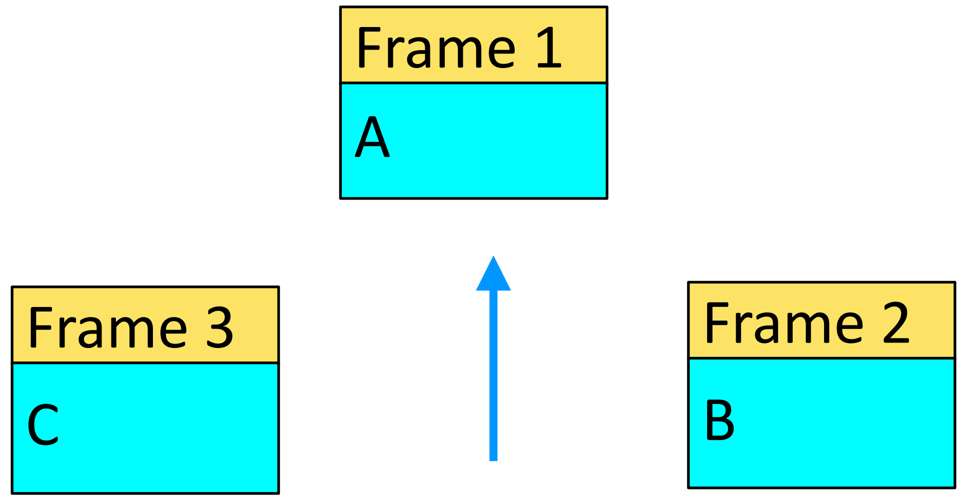
Clock Example: 3 Frames Allocated to Process



Set B to cold.
Advance hand.

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A	A				
Frame 2		B	B	B	B				
Frame 3			C	C	C				
Reference Bit (R)									
For A	1	1	1	1	1	0			
For B		1	1	1	1	0			
For C			1	1	1				
For D									

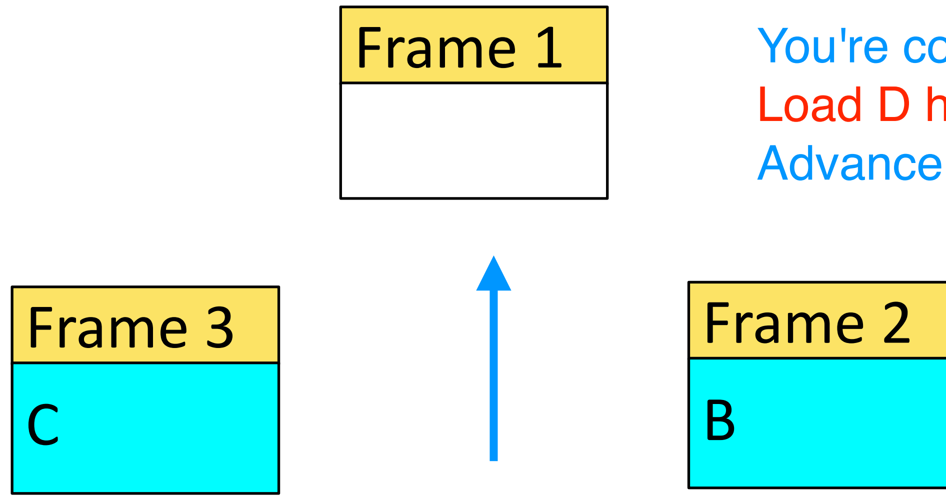
Clock Example: 3 Frames Allocated to Process



Set C to cold.
Advance hand.

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A	A				
Frame 2		B	B	B	B				
Frame 3			C	C	C				
Reference Bit (R)									
For A	1	1	1	1	1	0			
For B		1	1	1	1	0			
For C			1	1	1	0			
For D									

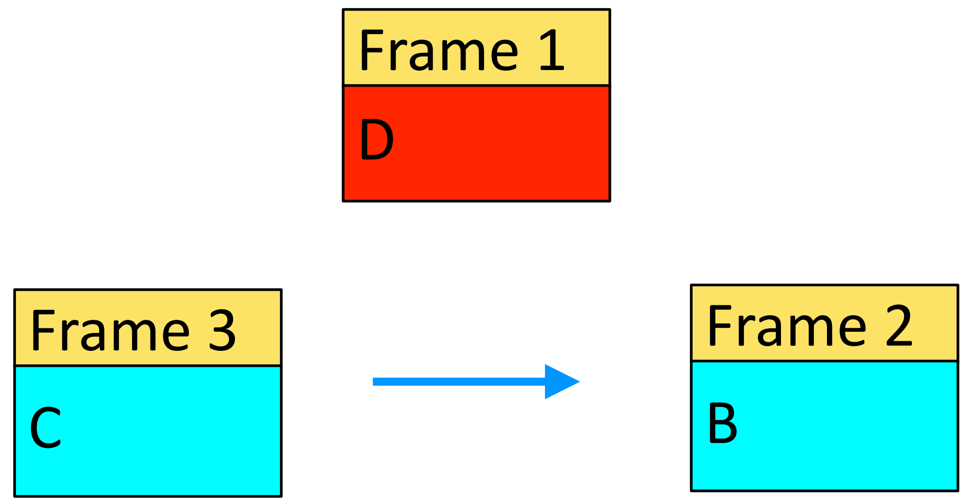
Clock Example: 3 Frames Allocated to Process



You're cold! Evict A.
Load D here. Set D to hot.
Advance hand.

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A	A				
Frame 2		B	B	B	B				
Frame 3			C	C	C				
Reference Bit (R)									
For A	1	1	1	1	1				
For B		1	1	1	1	0			
For C			1	1	1	0			
For D									

Clock Example: 3 Frames Allocated to Process



Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A	A	D			
Frame 2		B	B	B	B	B			
Frame 3			C	C	C	C			
Reference Bit (R)									
For A	1	1	1	1	1				
For B		1	1	1	1	0			
For C			1	1	1	0			
For D						1			

Clock Example: 3 Frames Allocated to Process

Frame 1
D

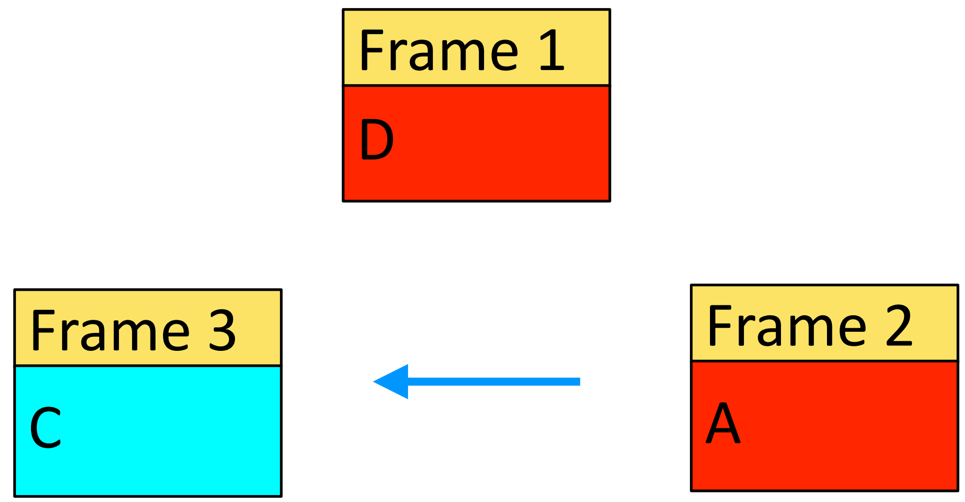
Frame 3
C

Frame 2

You're cold! Evict B.
Load A here. Set A to hot.
Advance hand.

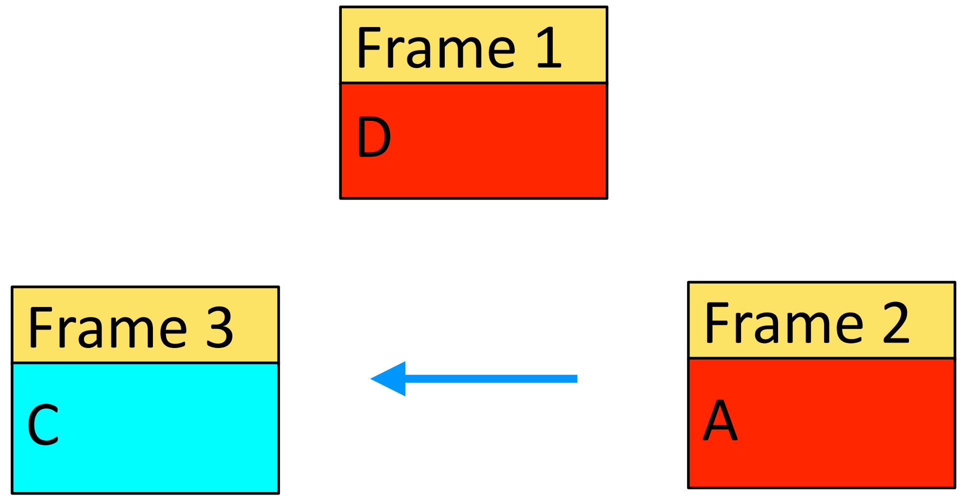
Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A	A	D			
Frame 2		B	B	B	B	B			
Frame 3			C	C	C	C			
Reference Bit (R)									
For A	1	1	1	1	1				
For B		1	1	1	1	0			
For C			1	1	1	0	0		
For D						1	1		

Clock Example: 3 Frames Allocated to Process



Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A	A	D	D		
Frame 2		B	B	B	B	B	A		
Frame 3			C	C	C	C	C		
Reference Bit (R)									
For A	1	1	1	1	1		1		
For B		1	1	1	1	0			
For C			1	1	1	0	0		
For D						1	1		

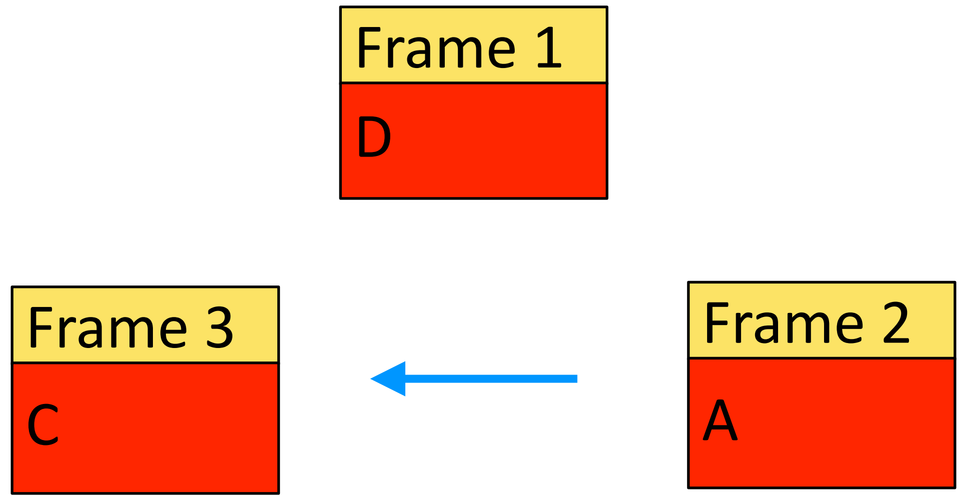
Clock Example: 3 Frames Allocated to Process



D is a Hit!
Don't advance.

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A	A	D	D	D	
Frame 2		B	B	B	B	B	A	A	
Frame 3			C	C	C	C	C	C	
Reference Bit (R)									
For A	1	1	1	1	1		1	1	
For B		1	1	1	1	0			
For C			1	1	1	0	0	0	
For D						1	1	1	

Clock Example: 3 Frames Allocated to Process



C is a Hit!
C *was* cold. Make it hot.
Don't advance.

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A	A	D	D	D	D
Frame 2		B	B	B	B	B	A	A	A
Frame 3			C	C	C	C	C	C	C
Reference Bit (R)									
For A	1	1	1	1	1		1	1	1
For B		1	1	1	1	0			
For C			1	1	1	0	0	0	1
For D						1	1	1	1

Clock Example (Solution)

Access Pattern	A	B	C	A	B	D	A	D	C
Frame 1	A	A	A	A	A	D	D	D	D
Frame 2		B	B	B	B	B	A	A	A
Frame 3			C	C	C	C	C	C	B

Number of Page Faults = 6

Evaluation of CLOCK

- Pros:
  - On average, much faster than LRU implementations
    - Average case: $O(1)$ eviction
    - (Clock-hand moves short distance, evict page)
  - Consumes only 1 additional bit in the PTE
- Cons:
  - If all pages are **hot (R == 1)**, clock hand must still scan all frames
    - So, worst case: it's still $O(n)$ eviction time
    - (But it doesn't happen often in practice)

Administrivia 4/25

- This Saturday
  - Math/CS Day in TH 395 @ 9am

- Next Wednesday
  - Department BBQ and Student Awards
  - ?? 5? in the Courtyard

Administrivia 4/25 (2)

- Last time...
  - Knuth's principle of optimization and a small memory footprint
  - Virtual memory: "Use the RAM as a cache for the disk"
  - Demand paging and page-fault handling

- Today:
  - How does an OS locate pages that are on disk?
  - When a process' allocation fills up, what next?
    - Page replacement

Administrivia 4/28

- Reminders:
  - Final exam on Monday 4-6p, is comprehensive
  - BBQ Wed @ 5p
- Last time...
  - Management of swap space
- Today:
  - When a process' allocation fills up, what next?
    - Page replacement
  - FIFO, LRU, CLOCK