Lecture 13: Swapping + RAID

CS343 – Operating Systems

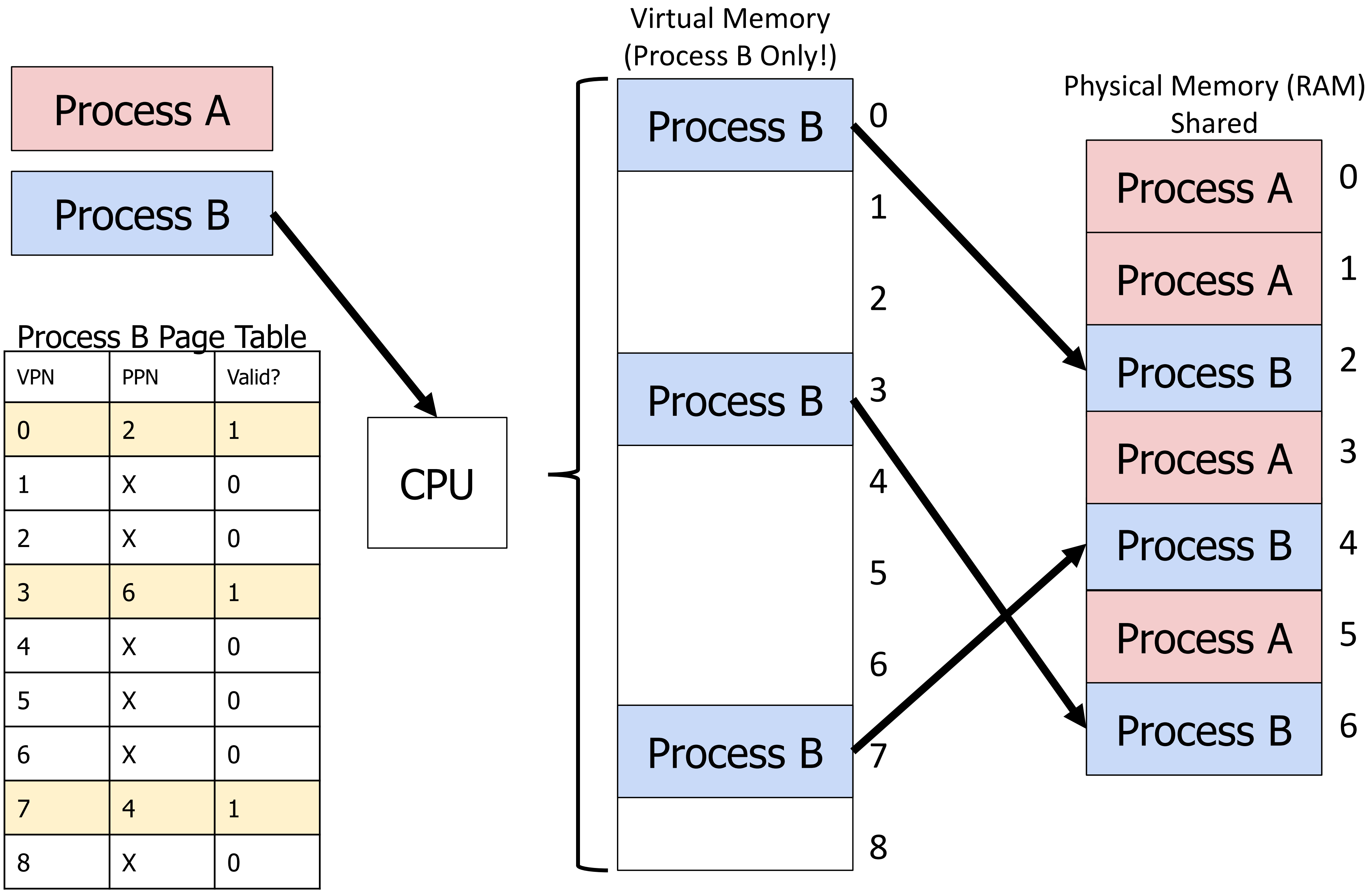
Branden Ghena – Spring 2022

Some slides borrowed from:
Stephen Tarzia (Northwestern), Shivaram Venkataraman (Wisconsin), and UC Berkeley CS162

Northwestern

Today’s Goals

- Introduce swapping as a mechanism for enabling more virtual memory than physical memory.
- Understand several page replacement policies that control swapping.
- Explore topic of RAID – redundancy in disks.



Process B Page Table

VPN	PPN	Valid?
0	2	1
1	X	0
2	X	0
3	6	1
4	X	0
5	X	0
6	X	0
7	4	1
8	X	0

The OS view on memory

- Managed through virtual memory translation
  - Paging (or Segmentation) that we talked about last time
- OS chooses which portions of processes go in RAM
  - Lazy loading: don’t put stuff in RAM until it’s needed
  - If memory overfills, some portions of memory get “swapped” to disk
    - Writeable memory (stack, heap, global data) must be preserved
    - Read-only memory (code) can be reloaded from original location

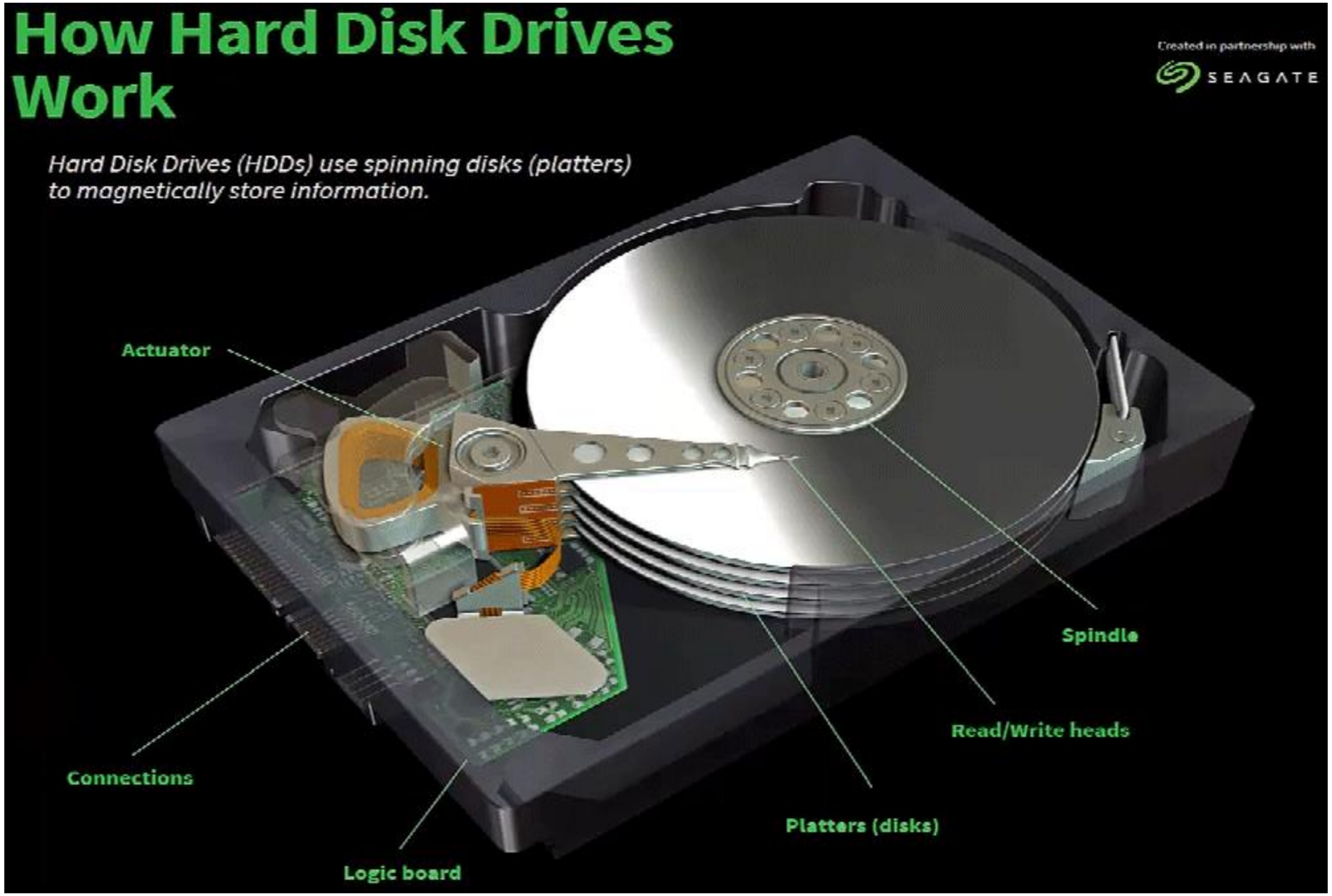
The OS view on disk

- Non-volatile memory store
  - Everything else on the system disappears when power is removed (and cannot be trusted across reboots)

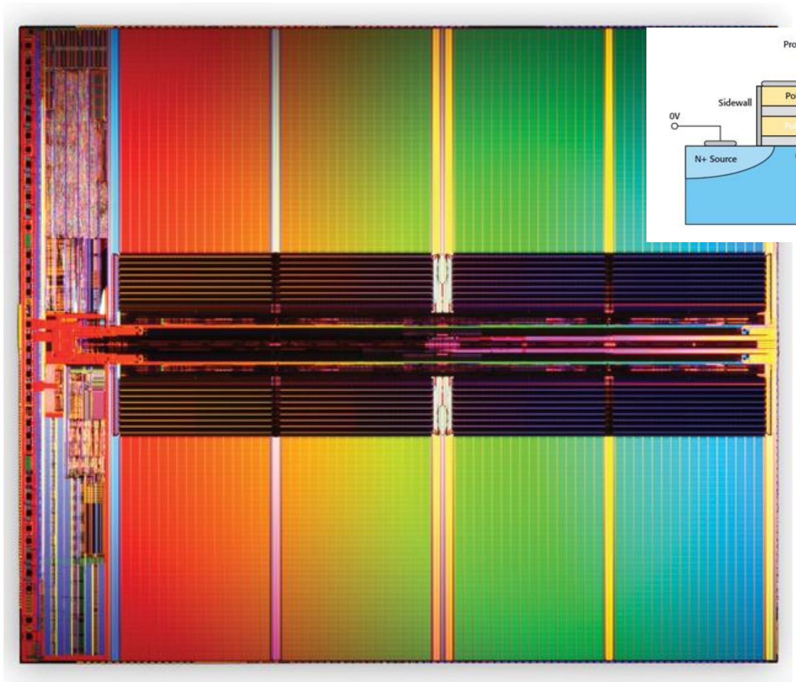
- Backing store for lots of information
  - Boot information: via “Master Boot Record” on disk
  - Filesystem, which the OS manages access to through system calls
  - Swap space, which the OS moves extra pages in and out of
    - Disk is significantly bigger than RAM, so this will work

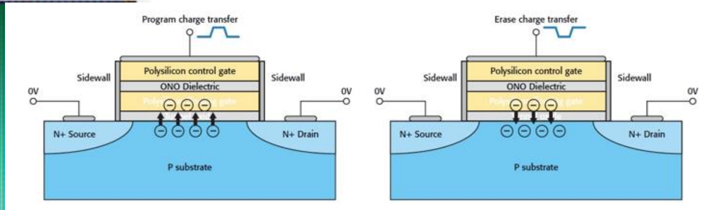
- Disk is a device that the OS manages and reads in “blocks”
  - Compare to memory, which is directly addressed by processes

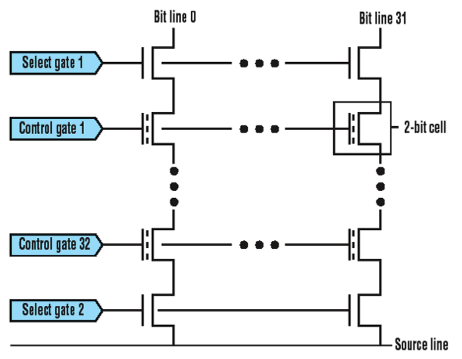
Traditional hard disk drives (HDDs) use magnetic regions


How Hard Disk Drives Work
Created in partnership with
SEAGATE
Hard Disk Drives (HDDs) use spinning disks (platters) to magnetically store information.
Actuator
Spindle
Read/Write heads
Connections
Platters (disks)
Logic board

Solid state drives (SSDs) use flash memory


2. Micron's triple-level cell (TLC) flash memory stores 3 bits of data in each transistor.





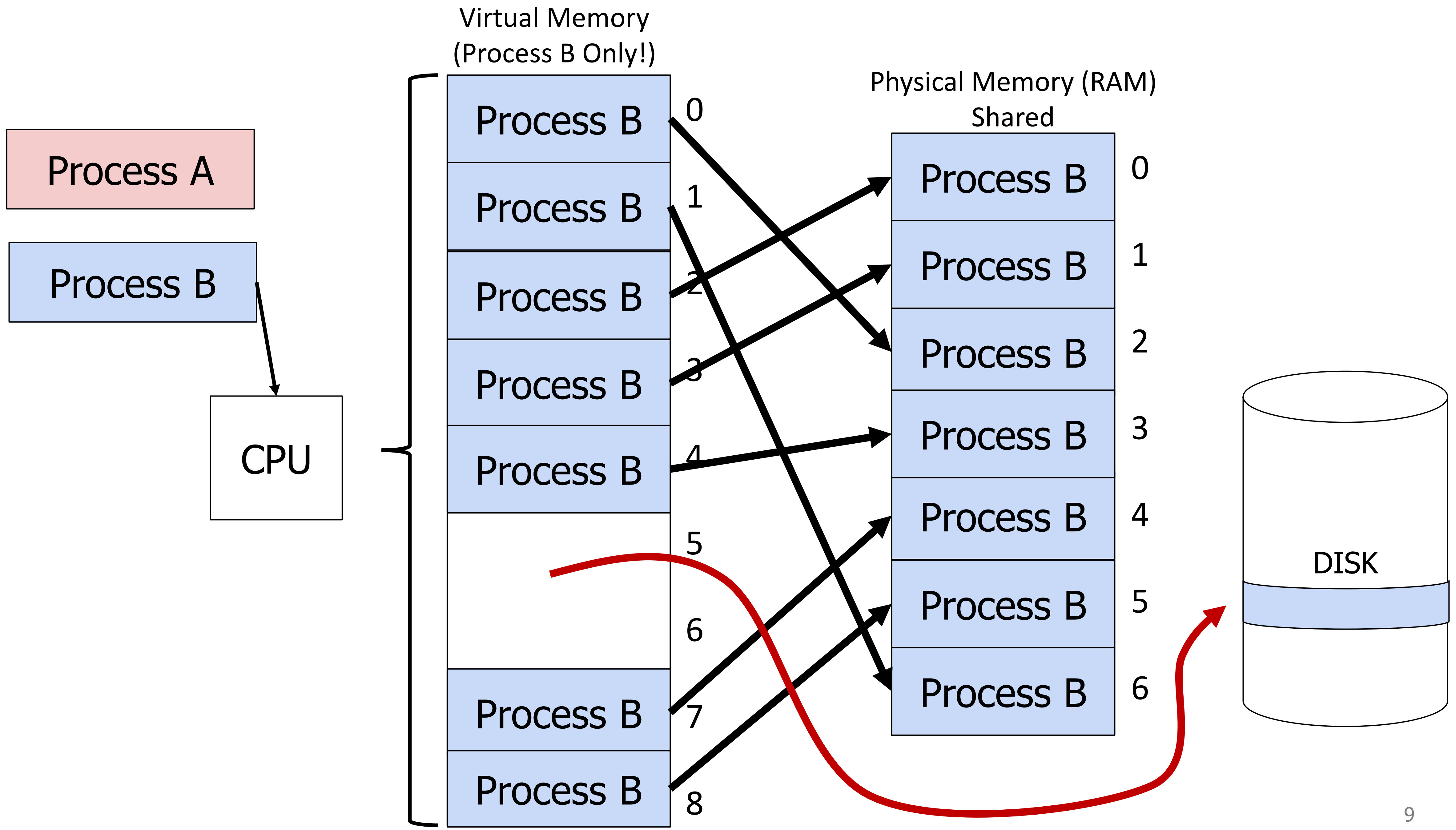
In the basic functional block used in multilevel NAND flash memories, 32 rows of bit lines and 32 control-gate lines form a building block that's repeated many times to form the memory array. The select gate lines are used with the control gate lines to control access to the array.

- Still non-volatile
- Significantly faster
  - 0.1 ms to access (10 ms for disk)
- More limited lifetime than disk
  - Limited writes

NMOS transistor with an additional conductor between gate and source/drain which "traps" electrons. The presence/absence is a 1 or 0

Outline

- **Swapping**
  - **Overview**
  - When To Swap
  - Page Replacement Policies
  - Implementing LRU
- RAID


Virtual Memory
(Process B Only!)
Physical Memory (RAM)
Shared
Process A
Process B
CPU
Process B
0
1
2
3
4
5
6
7
8
DISK

Motivation for swapping

- Processes should be independent of the amount of physical memory
  - Should be correct, even if not performant

- OS goal: support processes when not enough physical memory
  - Multiple processes combining to more than physical memory
  - Single process with very large address space
    - Video games: Red Dead Redemption 2 – 150 GB
    - Large-scale data processing: Compiling Android – 16 GB

- OS provides illusion of more physical memory by using disk

Locality of reference

- If disk is involved with memory, won’t this be ridiculously slow?

- Leverage *locality of reference* within process
  - **Spatial**: memory addresses near referenced address likely to be next
  - **Temporal**: referenced addresses likely to be referenced again
  - Processes spend majority of time in a small portion of code
    - Estimate: 90% of time spent in 10% of code (loops)

- Implication
  - Process only uses small amount of address space at any moment
  - Only small amount of address space needs to be in physical memory
  - RAM acts as a sort of cache for program memory

How swapping works

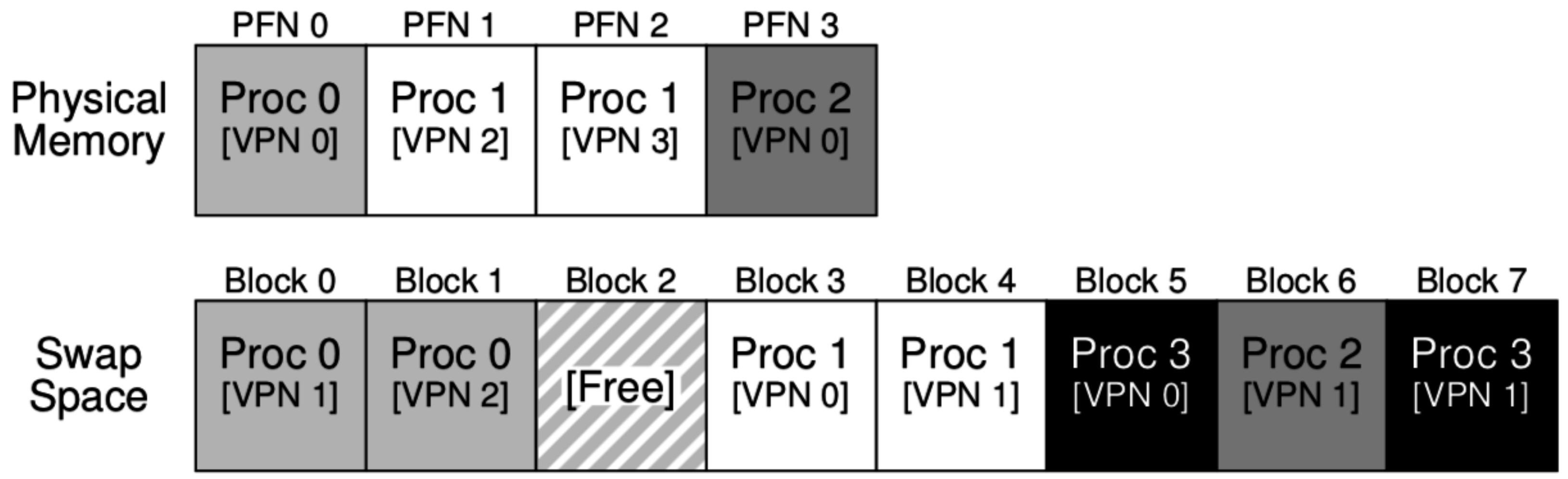
- OS moves some pages from RAM to disk

- Processes can still run when not all pages are in physical memory
- OS and hardware cooperate to make memory available when needed
    - Same behavior as if all of address space always was in memory
    - Except in terms of time, but processes don’t know about time…

- Requirements
    - OS needs **mechanism** to identify location of address space pages on disk and move them into RAM when necessary
    - OS needs **policy** to determine which pages go in RAM or disk

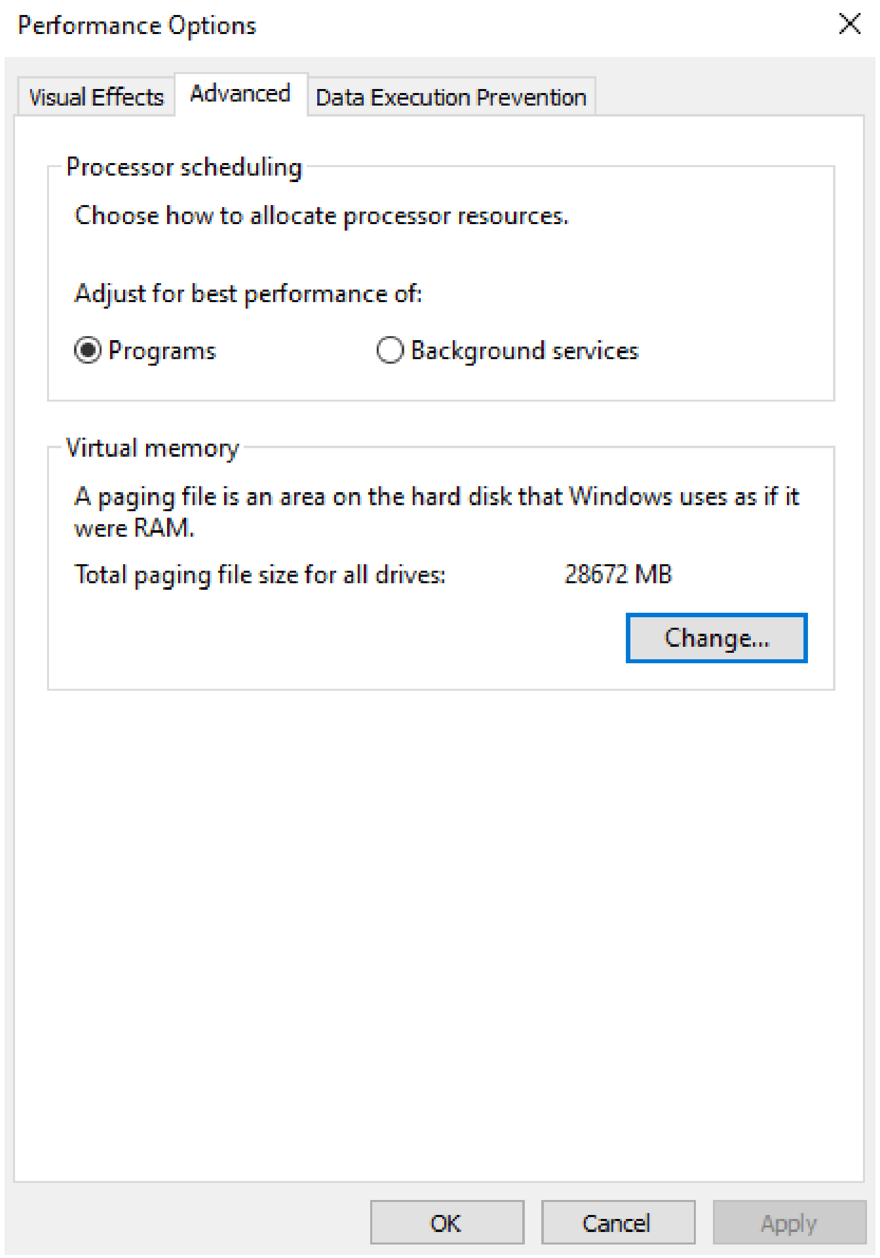
Combination of swapping and paging

- Processes have memory pages, which are distributed among RAM and Disk
- Example:
  - Processes 0, 1, and 2 are partially in RAM
  - Process 3 is entirely in “swap space” on disk



Paging on Windows

- Windows lets you see and even set the size of swap space on disk
  - This is only space for temporary storage of physical memory pages (from RAM)
  - After it runs out, other processes can't be loaded!
  - This is separate from leaving some parts of a process on disk on unloaded



Processor scheduling

Choose how to allocate processor resources.

Adjust for best performance of:

[✓ Programs] Background services

Virtual memory

A paging file is an area on the hard disk that Windows uses as if it were RAM.

Total paging file size for all drives: 28672 MB

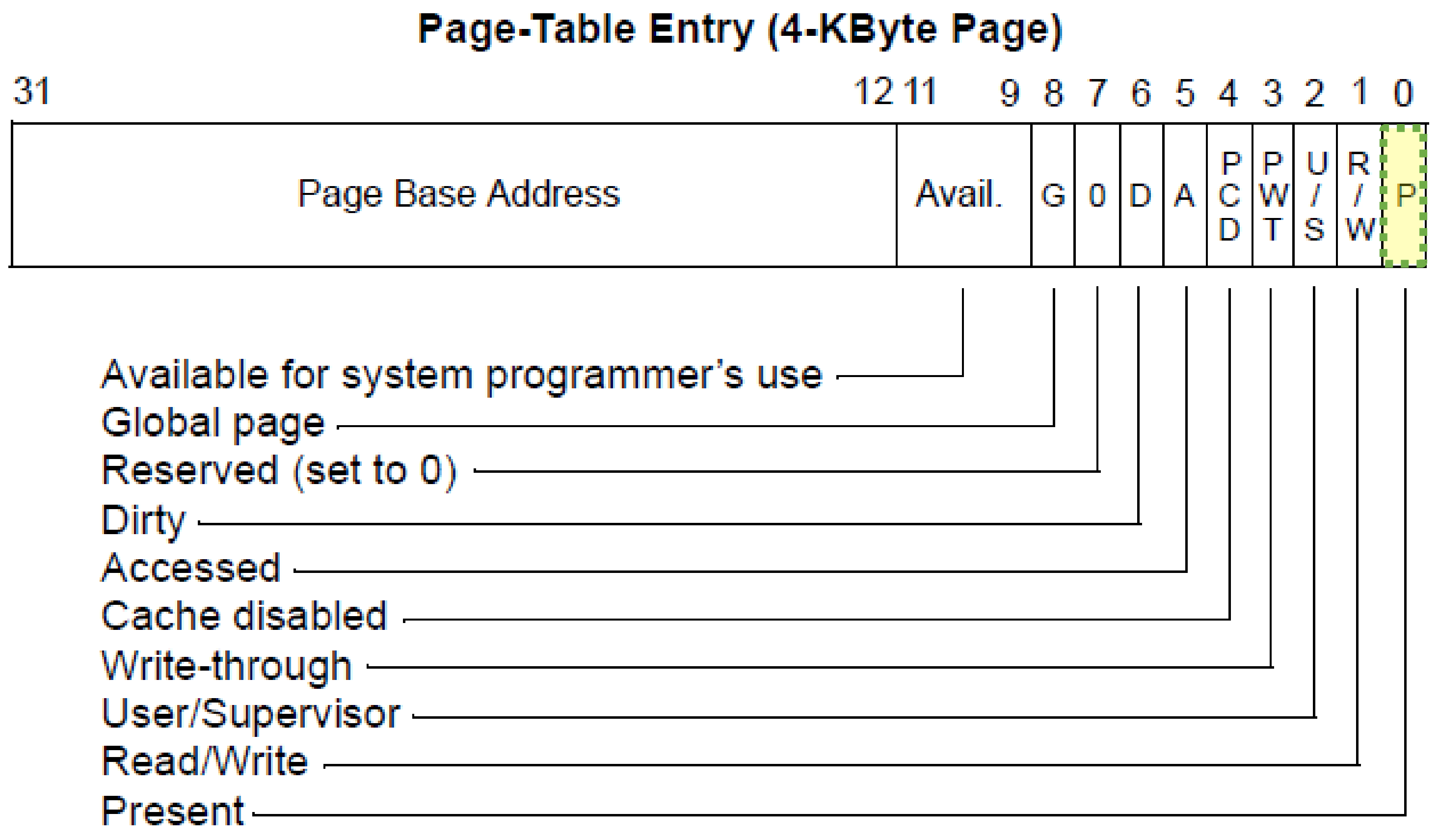
Mechanisms to support swapping

- Each page in virtual address space lives in a location
    1. Physical memory
    2. Disk
    3. Nowhere

- Extend page tables with an extra bit – present
    - Physical Page Number, Permissions, Valid, Present
        - Page in memory, valid and present
        - Page on disk, valid but not present
            - Page Table Entry points to block on disk instead!
            - Trap to OS on reference
            - Maybe never loaded before, or maybe moved to swap space (works the same either way)
        - Invalid page, not valid and not present OR bad permissions
            - Trap to OS on reference

Other bits in a page table entry

- Page Base Address can be reused to hold disk block
- Dirty bit
  - Whether page has been modified
  - If page needs to be swapped out, only preserve if modified



Short Break + Question

- What are reasons for a Page Fault? (when the MMU calls the OS)

Short Break + Question

- What are reasons for a Page Fault? (when the MMU calls the OS)

1. Invalid permissions

2. Page actually on disk
   - Because the OS never actually loaded it! (lazy loading)
   - Because the OS swapped it to disk (when low on RAM)

3. Page isn’t valid at all

Types of page faults

- **Minor/soft**: Page is loaded in memory, but PTE is not configured:
  - Memory could be a shared library already in memory from another process.
  - OS could be tracking accesses to this page. (hardware without a dirty bit)

  Response: update the PTE.

- **Major/hard**: A disk access will be needed:
  - Anonymous page (process data) may have been swapped out.
  - Lazy-loading program executable.

  Response: load the page from disk

- **Invalid**: User program misbehaved:
  - Dereference null or invalid pointer.
  - Write to page that is read-only.
  - Execute code on a page that is not executable (for security).

  Response: terminate the process.

Steps to a memory access with swapping

1. Hardware checks TLB for virtual address
    - If Hit, address translation complete AND page in physical memory
2. Hardware (or OS) walks page tables
    - If valid and present, then page in physical memory
3. Trap into OS (a.k.a. Page Fault!)
    - If invalid or bad permissions, fault process (segmentation fault)
    - If valid but not present
        - If memory is full, select a victim page in memory to replace
            - If modified (dirty), write to disk
            - Invalidate TLB entry for that page
        - OS reads referenced page from disk into memory
        - Page table is updated, present bit is set
        - Resume process execution (could be complicated on CISC machines)

Outline

- **Swapping**
    - Overview
    - **When To Swap**
    - Page Replacement Policies
    - Implementing LRU
- RAID

Policies to determine swapping evictions

- Goal: minimize the number of page faults
  - Page faults need to read/write from disk and are very slow
  - So the OS can take plenty of time to make a *good* decision

- OS has two decisions
  1. Page Selection
     - **When** should a page be brought into memory?

  2. Page Replacement
     - **When** should a page be swapped into disk?
     - **Which** page should be swapped out of physical memory?

When do we load in pages? (page selection)

- Demand paging: Load page only when page fault occurs (lazy)
  - Intuition: Wait until page must absolutely be in memory
  - When process starts: No pages are loaded in memory
  - Problems: Pay cost of page fault for every newly accessed page

- Pre-paging (prefetching): Load page before referenced (eager)
  - OS predicts future accesses and brings pages into memory early
  - Works well for some access patterns (e.g., sequential)

- Hints: Combine above with user-supplied hints about page references
  - User specifies: may need page in future, don't need this page anymore, or sequential access pattern, ...
  - Example: `madvise`() in POSIX – "give advice about use of memory"

When do we swap out pages? (page replacement)

- Demand swapping: whenever the page fault actually occurs
  - Simplest method
  - Swap actually occurs asynchronously
    - Start the disk I/O and block the process that faulted

- Background swapping: preemptively when RAM is getting full
  - Background service in kernel periodically runs (kswapd)
  - If number of free physical pages < “low water mark”, evict a bunch
    - Writing many pages to disk in one operation is way more efficient

Thrashing

- Thrashing: when swapping happens frequently
  - Policy could be bad (working set keeps getting swapped to disk)
  - More likely RAM is too small
- Frequent swapping slows down the whole computer to a crawl
  - Constantly waiting on disk I/O!
- Solution for thrashing
  - Kill processes until it stops (relieves memory pressure)
  - Install more RAM in the computer

Outline

- **Swapping**
  - Overview
  - When To Swap
  - **Page Replacement Policies**
  - Implementing LRU

- RAID

Which page should be evicted?

- **Page replacement policy** determines page to evict

- Very similar process as cache eviction or TLB eviction
    - Misses are expensive, so make sure you evict the right page
    - Page faults are extremely long, so a sophisticated policy is possible

Optimal page replacement policy

- Optimal page replacement
    - Evict page that will be accessed furthest in the future
- Advantages
    - Guaranteed to minimize the number of page faults
- Disadvantages
    - Requires the OS to predict the future 🧙
    - Doesn’t actually exist
- This is a performance “upper bound”
    - This is the best anything can do, so it is useful to compare against
    - Still has misses due to cold-start and capacity

First-In-First-Out replacement policy

- FIFO replacement
  - Evict page that has been in memory the longest

- Advantages
  - Fair as all pages have equal residency
  - Easy to implement

- Disadvantages
  - Some pages of memory are always needed (stack)
    - Memory doesn’t really need “fairness” like processes did

Least Recently Used replacement policy

- LRU replacement
  - Replace page not accessed for longest time
  - Using the past to predict the future (temporal locality)
- Advantages
  - With locality, LRU approximates Optimal
- Disadvantages
  - Harder to implement as we need to track when pages are accessed
  - Cyclical patterns can make LRU fail (bigger concern for cache than RAM)

Check your understanding – simple replacement policies

Page Requested

time

	Optimal		
D			
D			
B			
B			
A			
C			
B			
D			
B			
D			

	FIFO		
D			
D			
B			
B			
A			
C			
B			
D			
B			
D			

	LRU		
D			
D			
B			
B			
A			
C			
B			
D			
B			
D			

Check your understanding – simple replacement policies

Page Requested

time

Optimal

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C				
B				
D				
B				
D				

FIFO

D			
D			
B			
B			
A			
C			
B			
D			
B			
D			

LRU

D			
D			
B			
B			
A			
C			
B			
D			
B			
D			

Check your understanding – simple replacement policies

Page Requested

time

Optimal

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C				
B				
D				
B				
D				

FIFO

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C				
B				
D				
B				
D				

LRU

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C				
B				
D				
B				
D				

These are only **replacement** policies.
So they don't matter until RAM is full!

Check your understanding – simple replacement policies

Page Requested

time

Optimal

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	D	B	C	Miss
B	D	B	C	Hit
D	D	B	C	Hit
B	D	B	C	Hit
D	D	B	C	Hit

FIFO

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C				
B				
D				
B				
D				

LRU

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C				
B				
D				
B				
D				

For Optimal, which page do we replace?

Check your understanding – simple replacement policies

Page Requested

time

Optimal

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	D	B	C	Miss
B	D	B	C	Hit
D	D	B	C	Hit
B	D	B	C	Hit
D	D	B	C	Hit

FIFO

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	C	B	A	Miss
B	C	B	A	Hit
D	C	D	A	Miss
B	C	D	B	Miss
D	C	D	B	Hit

LRU

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C				
B				
D				
B				
D				

For FIFO, which pages do we replace?

Check your understanding – simple replacement policies

Page Requested

time

Optimal

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	D	B	C	Miss
B	D	B	C	Hit
D	D	B	C	Hit
B	D	B	C	Hit
D	D	B	C	Hit

FIFO

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	C	B	A	Miss
B	C	B	A	Hit
D	C	D	A	Miss
B	C	D	B	Miss
D	C	D	B	Hit

LRU

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	C	B	A	Miss
B	C	B	A	Hit
D	C	B	D	Miss
B	C	B	D	Hit
D	C	B	D	Hit

For LRU, which pages do we replace?

Check your understanding – simple replacement policies

Page Requested

time

Miss rate = 40%

Optimal

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	D	B	C	Miss
B	D	B	C	Hit
D	D	B	C	Hit
B	D	B	C	Hit
D	D	B	C	Hit

Miss rate = 60%

FIFO

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	C	B	A	Miss
B	C	B	A	Hit
D	C	D	A	Miss
B	C	D	B	Miss
D	C	D	B	Hit

Miss rate = 50%

LRU

D	D			Miss
D	D			Hit
B	D	B		Miss
B	D	B		Hit
A	D	B	A	Miss
C	C	B	A	Miss
B	C	B	A	Hit
D	C	B	D	Miss
B	C	B	D	Hit
D	C	B	D	Hit

For each, what are the final miss rates?

Break + Thinking

- How would you track the least recently used page?
    - Consider hardware and software

Outline

- **Swapping**
  - Overview
  - When To Swap
  - Page Replacement Policies
  - **Implementing LRU**

- RAID

Implementing LRU

- Implementing *perfect* LRU is difficult in practice

- Software perfect LRU
  - OS maintains an ordered list of physical pages by reference time
    - When page is referenced: move to end of list
    - When swap is needed: evict front of list
  - Tradeoff: slow on memory reference, fast on replacement (unacceptable)

- Hardware perfect LRU
  - Associate a timestamp with each physical page
    - When page is referenced: hardware updates timestamp for page
    - When swap is needed: OS searches through all pages for oldest
  - Tradeoff: fast on memory reference, extremely slow on replacement and needs special hardware

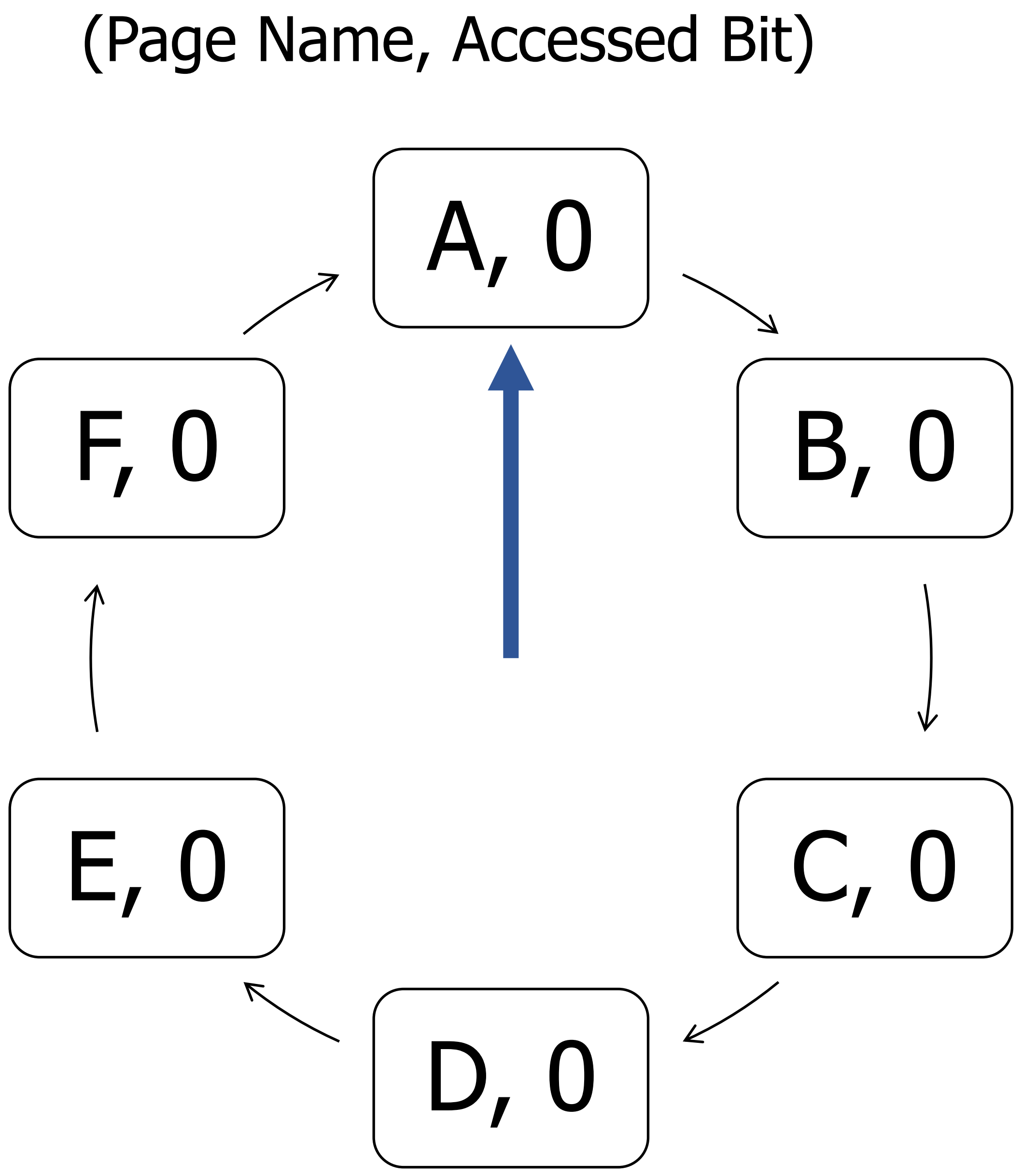
Clock algorithm

- LRU approximates Optimal anyways, so approximate a little more
    - Goal: find *an* old page, not necessarily the *oldest* page

- Clock algorithm
    - One “accessed” bit added to each page
        - When page is referenced: accessed bit is set to one (hardware)
        - When swap is needed:
            - Cycle through pages looking for one with accessed bit zero
            - Update accessed bit to zero after checking a page
            - Continue from where you left off when next swap is needed
    - Essentially looks for page that hasn’t been referenced this “cycle”

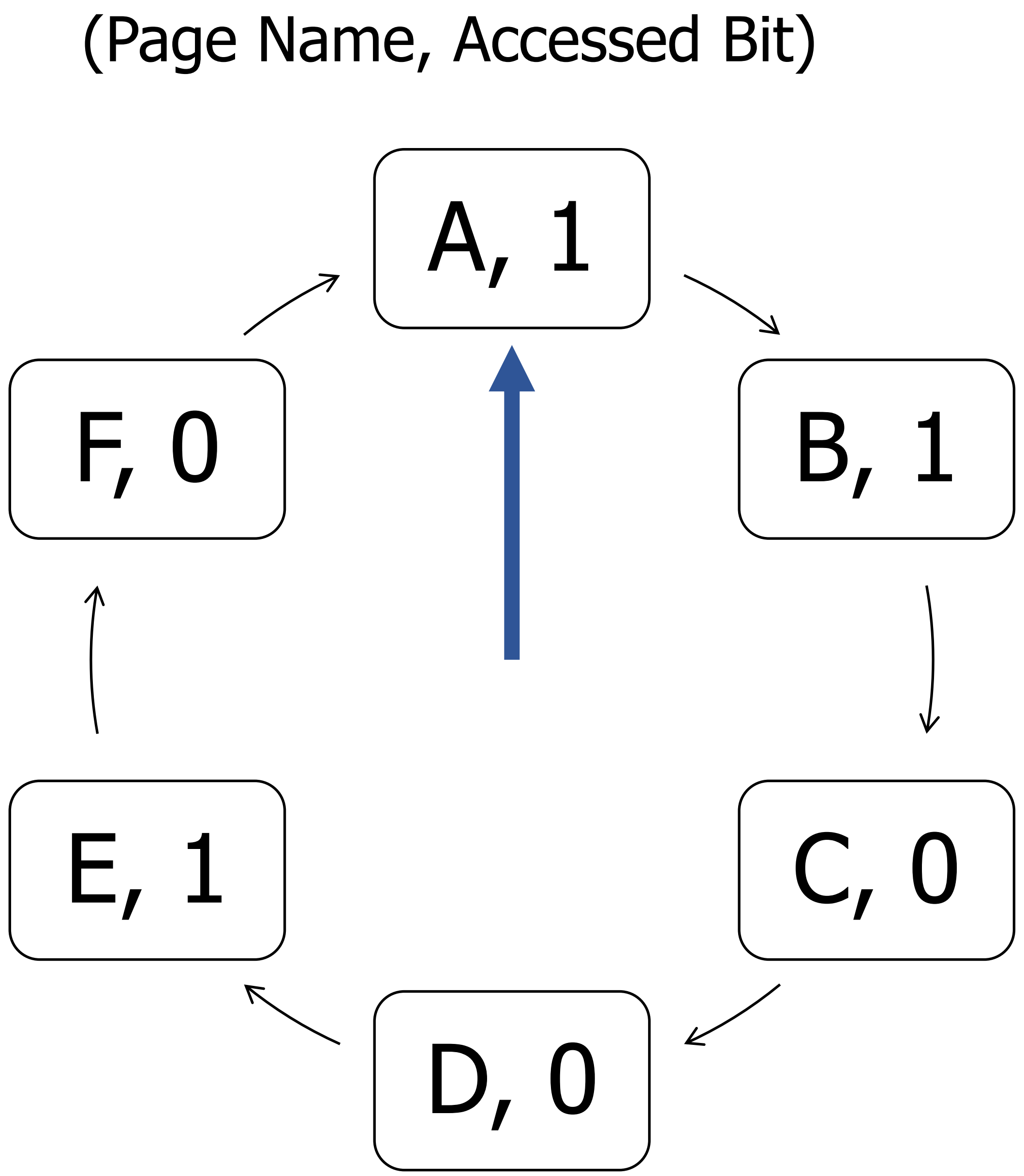
Clock algorithm example

- Initial setup
  - 6 pages total fit in memory
  - Accessed starts as zero
  - “clock hand” points at first page



Clock algorithm example

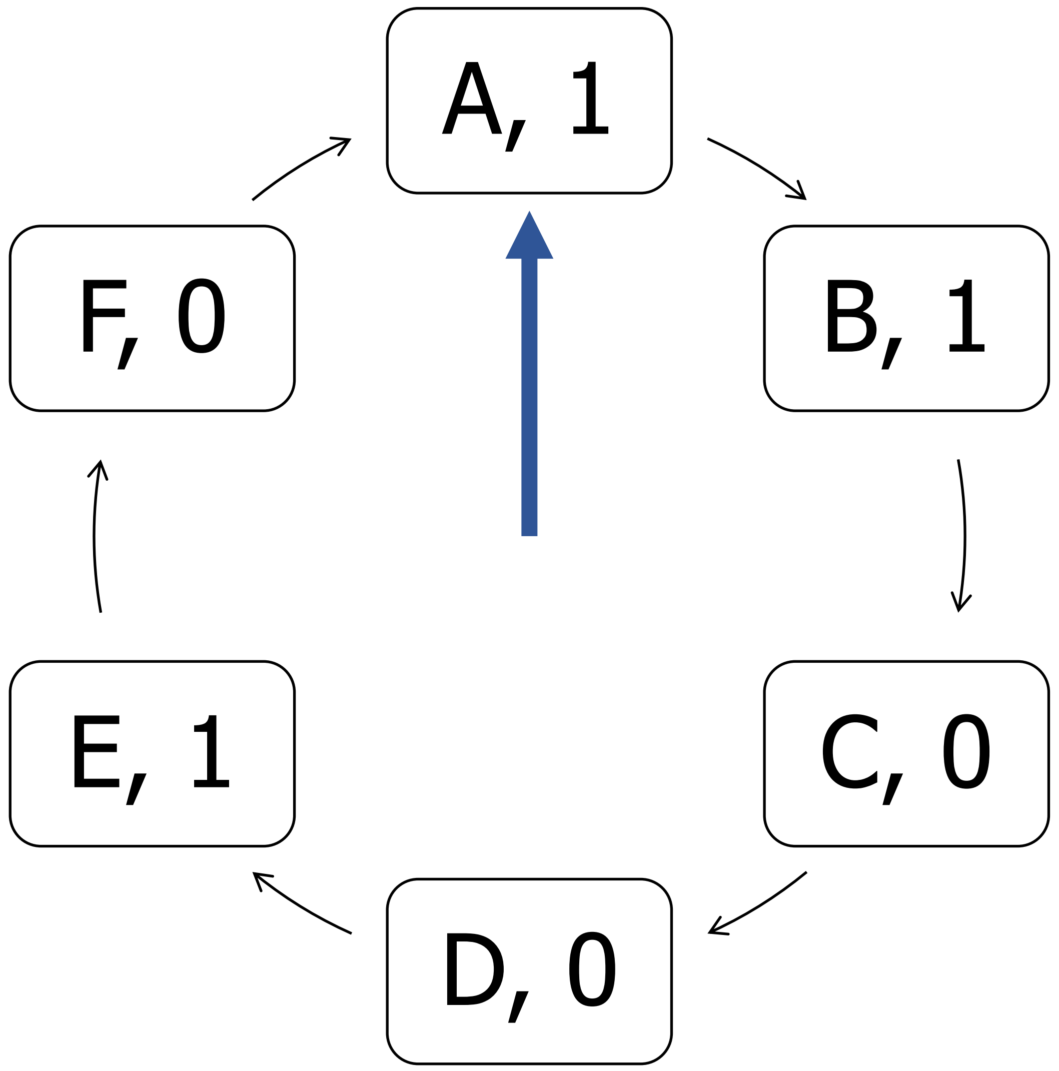
- After running a little while
  - Pages A, B, E are accessed



Clock algorithm example

- OS needs to swap pages
  - Algorithm starts
- A is recently accessed

(Page Name, Accessed Bit)

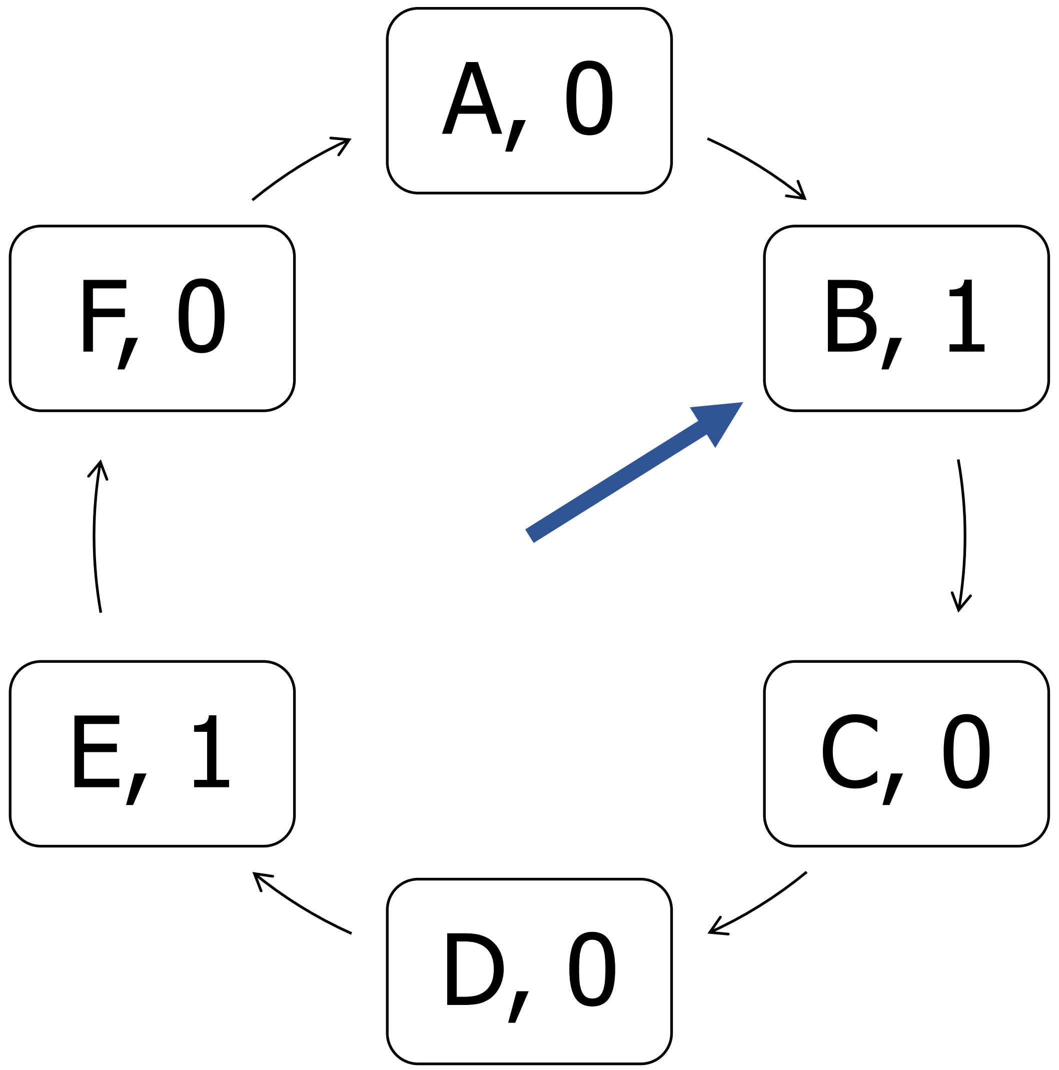


Clock algorithm example

- OS needs to swap pages
  - Algorithm starts

- A is recently accessed
- B is recently accessed

(Page Name, Accessed Bit)

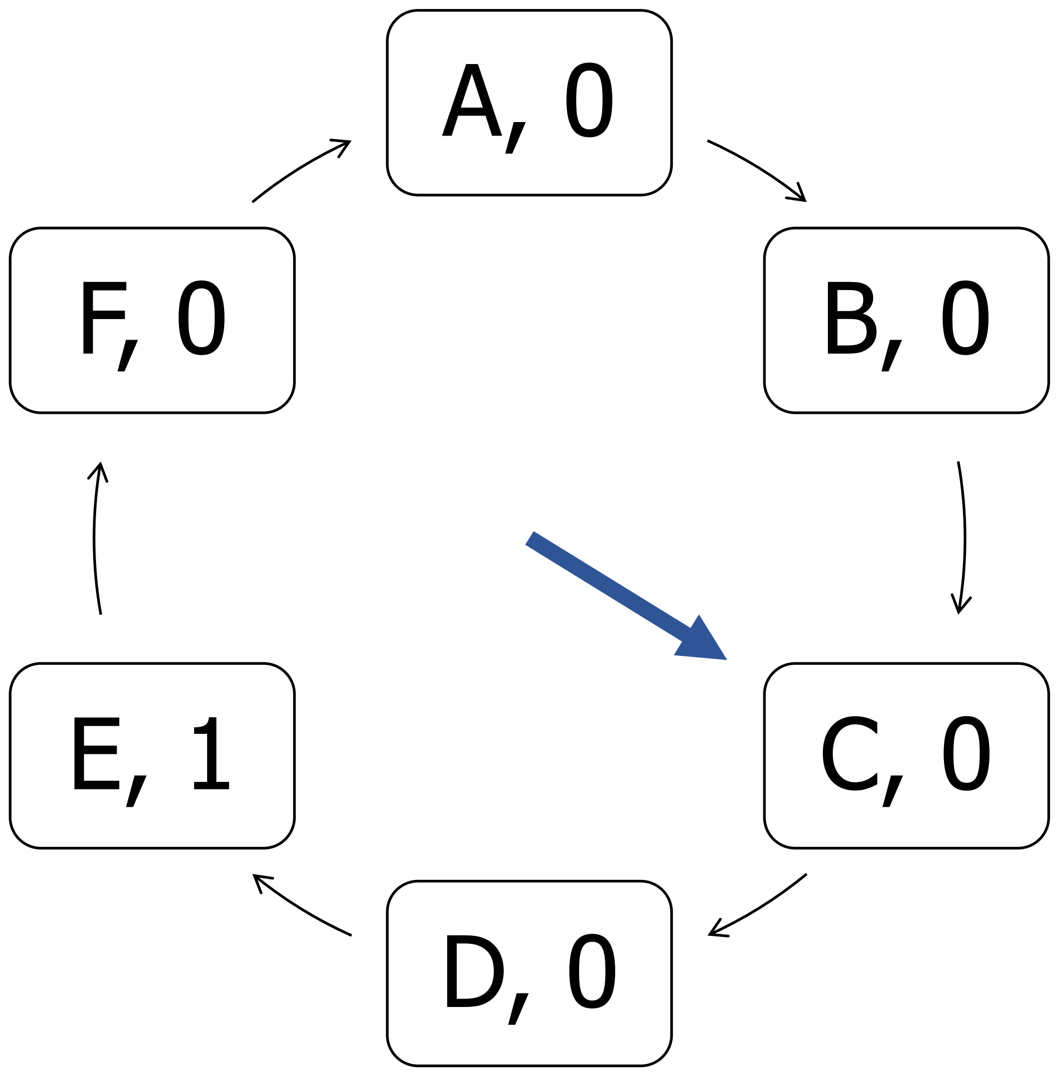


Clock algorithm example

- OS needs to swap pages
  - Algorithm starts

- A is recently accessed
- B is recently accessed
- C has not been recently accessed

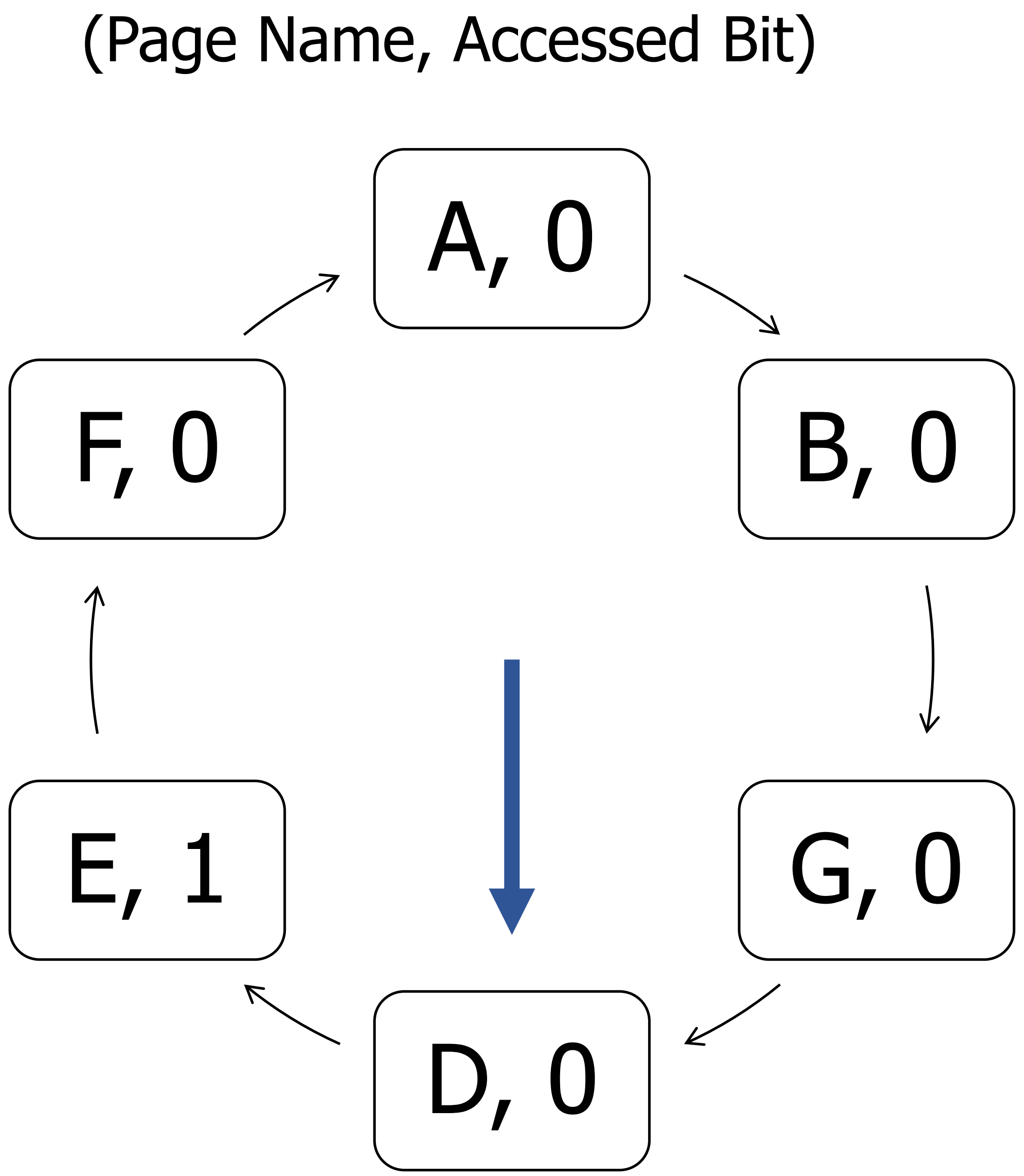
(Page Name, Accessed Bit)



Clock algorithm example

- OS needs to swap pages
  - Algorithm starts

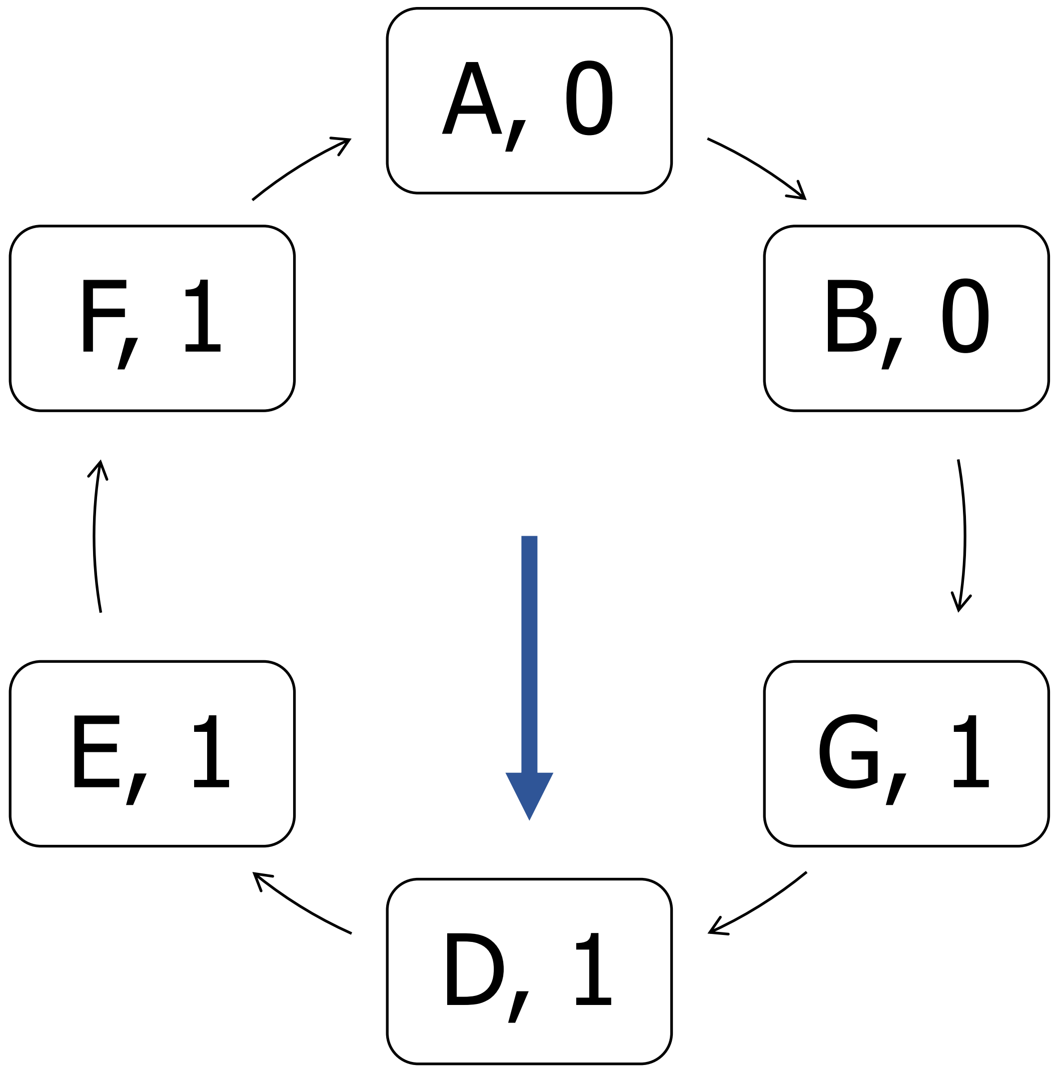
- A is recently accessed
- B is recently accessed
- C has not been recently accessed
  - So swap it
  - And advance hand once more



Clock algorithm example

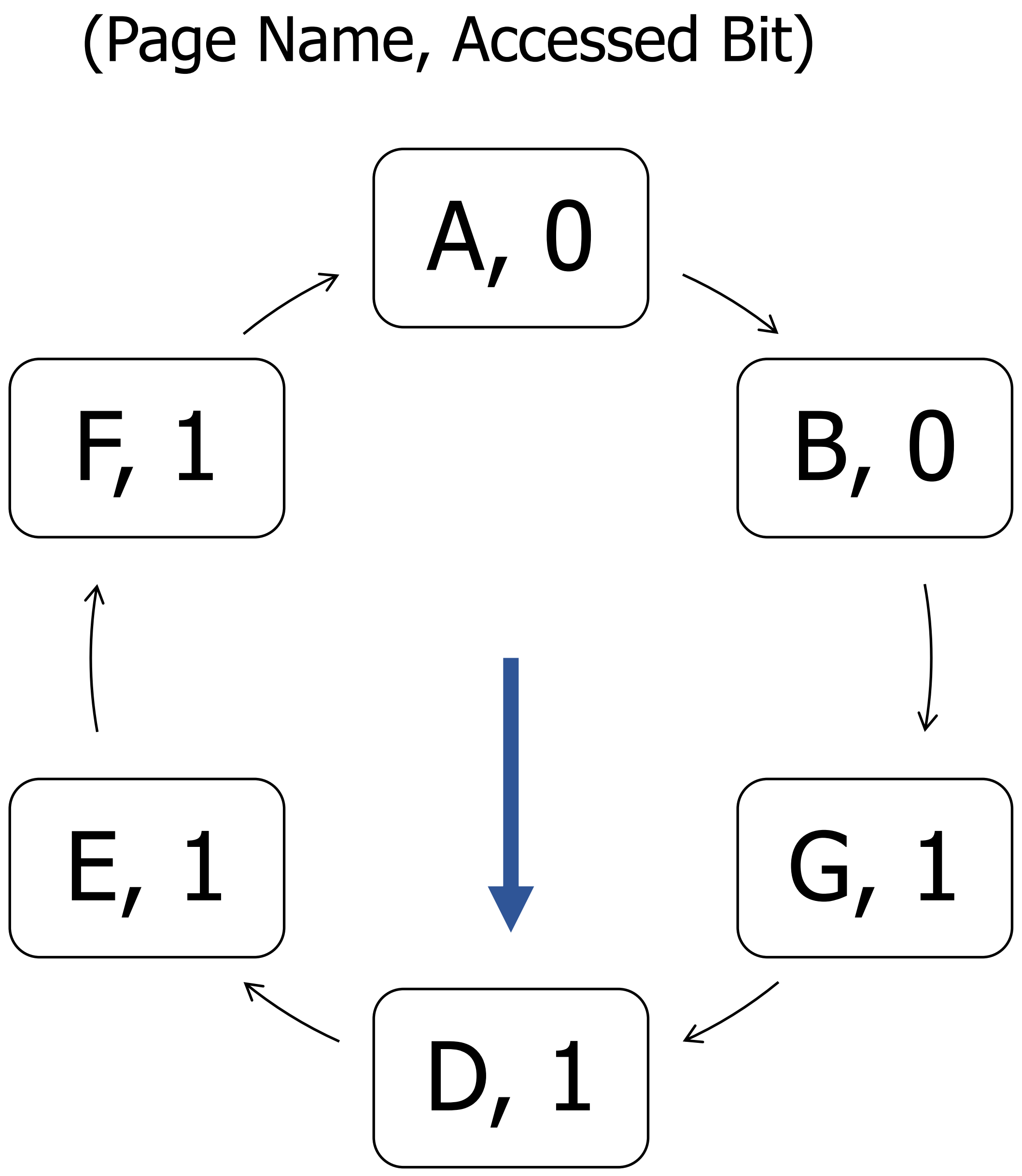
- Programs continue running for a while
    - Pages G, D, F are accessed

(Page Name, Accessed Bit)



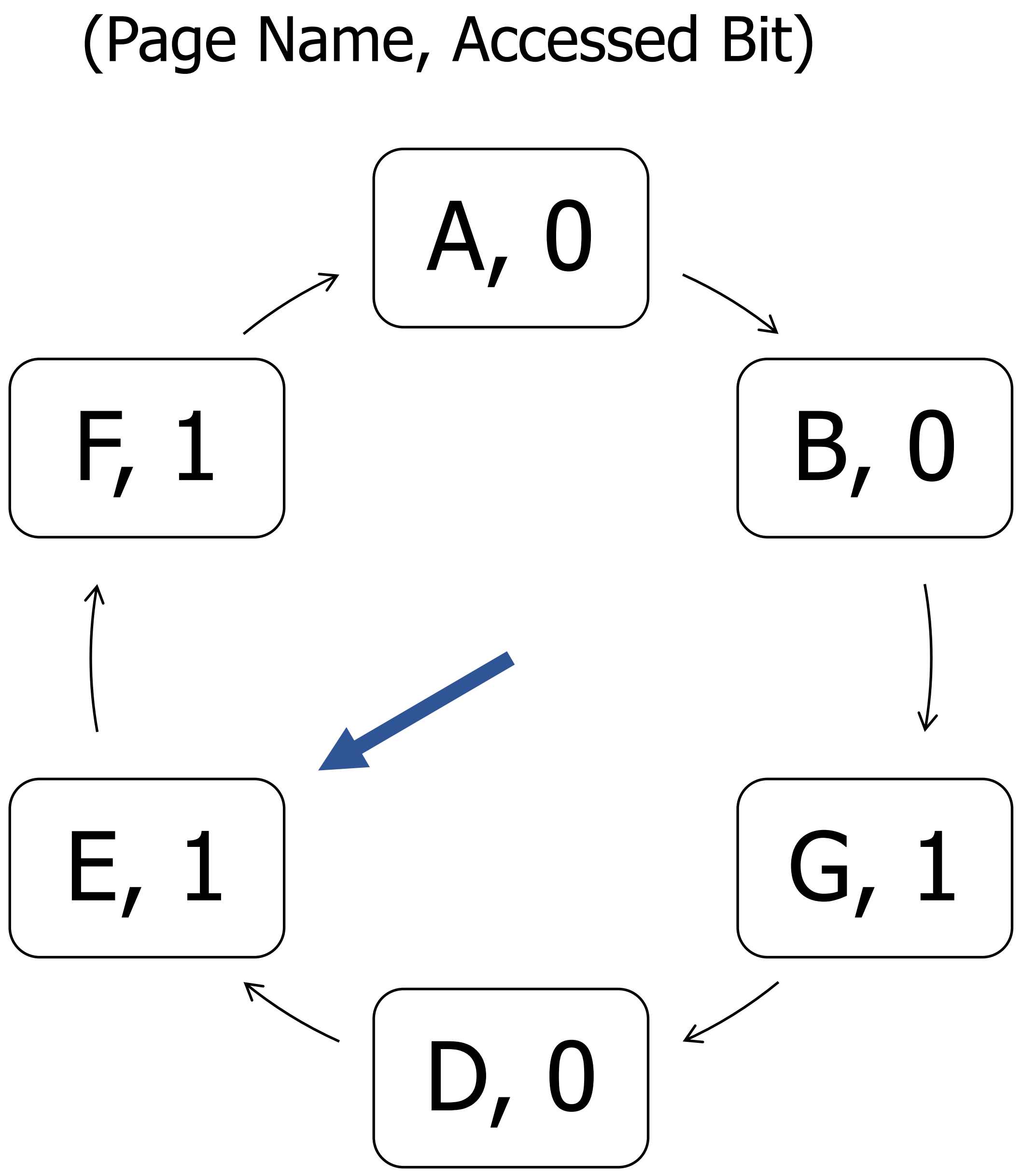
Clock algorithm example

- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new



Clock algorithm example

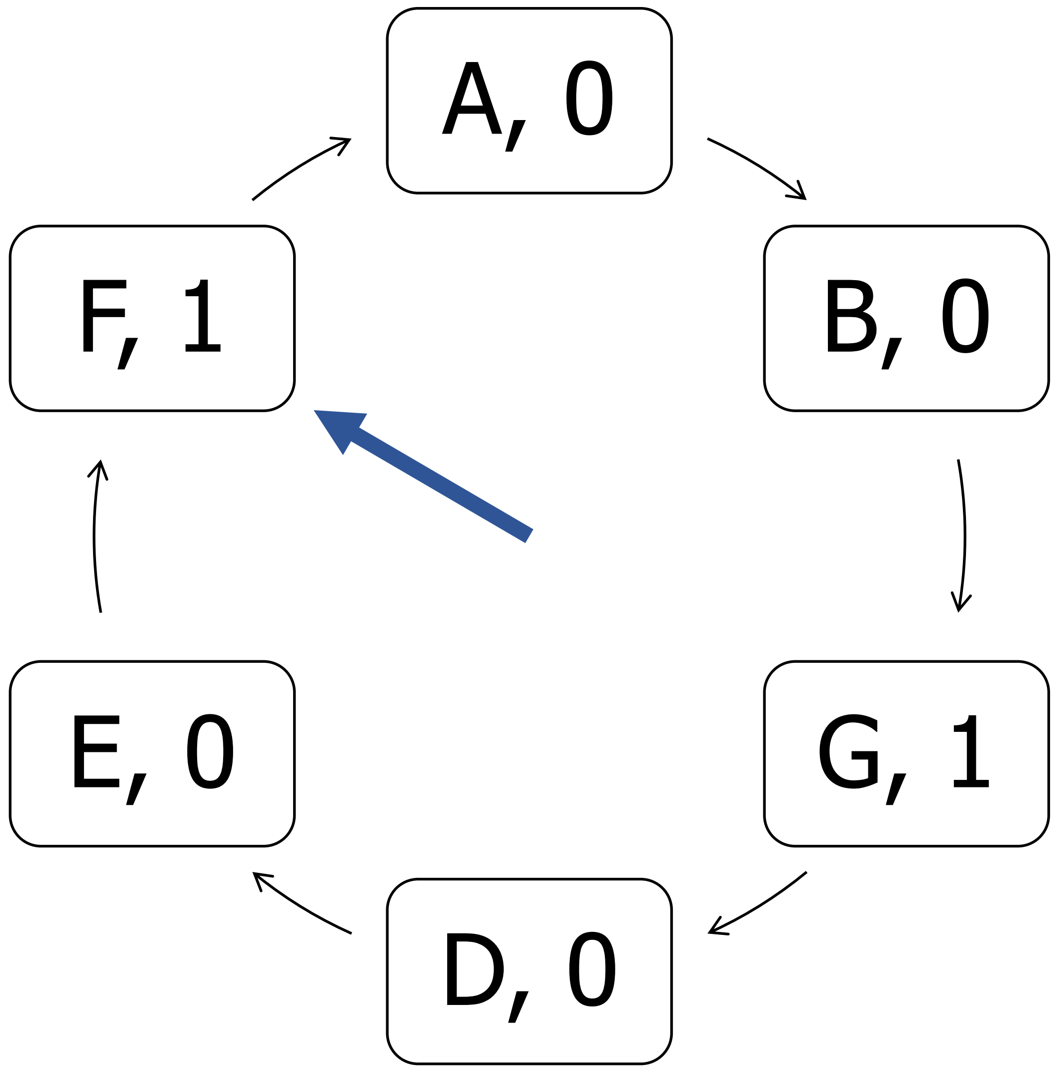
- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new
- D recently accessed



Clock algorithm example

- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new
- D recently accessed
- E recently accessed

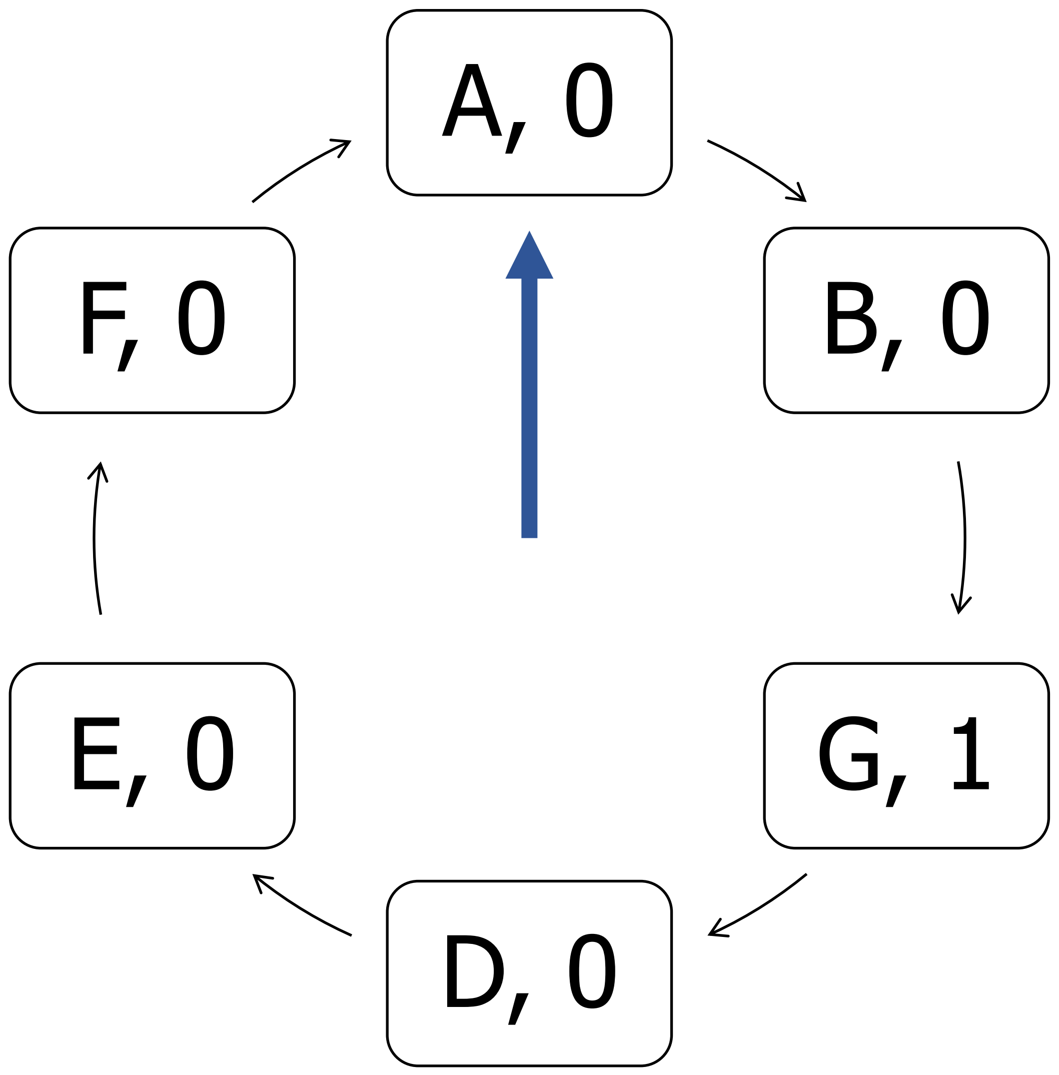
(Page Name, Accessed Bit)



Clock algorithm example

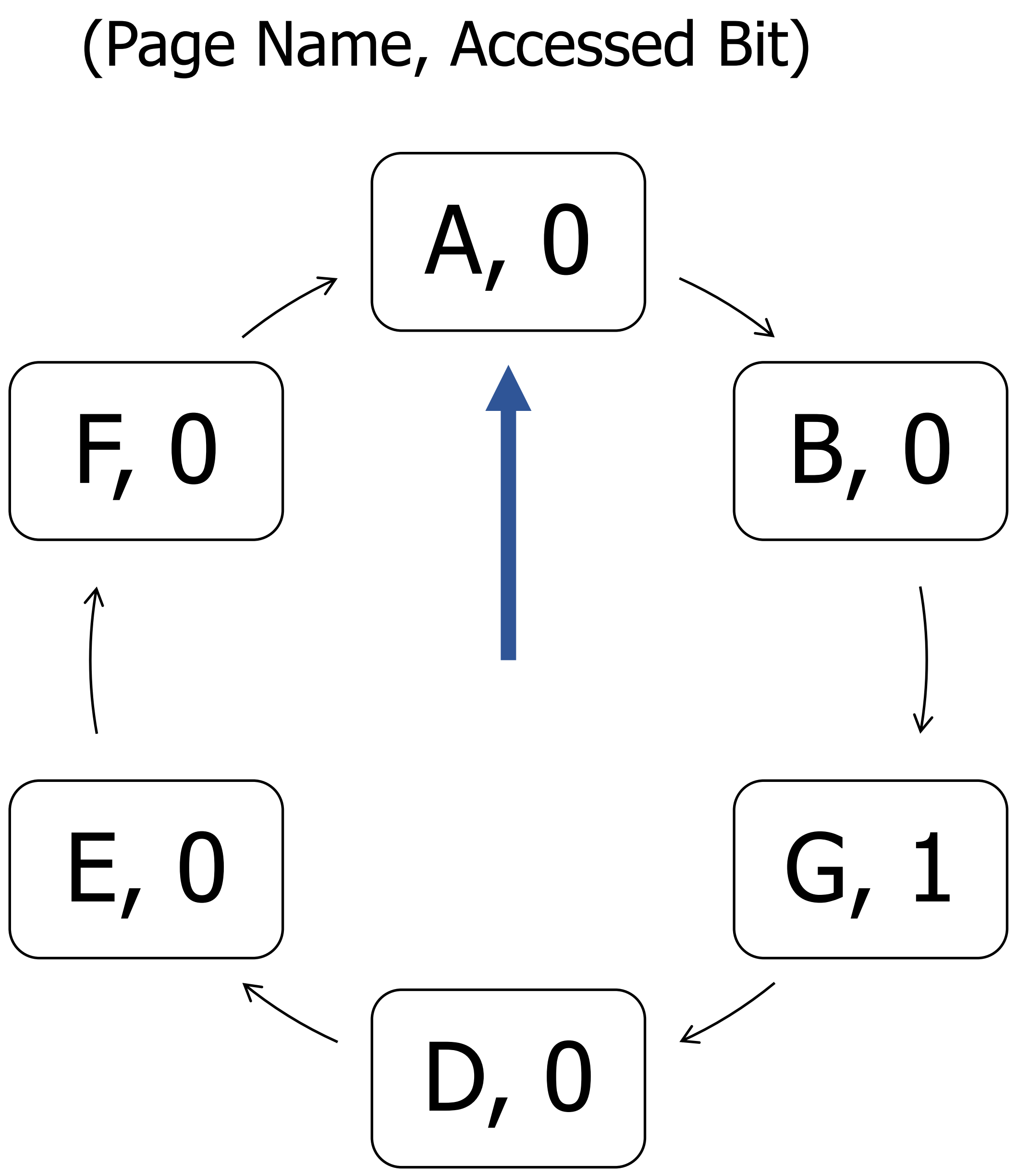
- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new
- D recently accessed
- E recently accessed
- F recently accessed

(Page Name, Accessed Bit)



Clock algorithm example

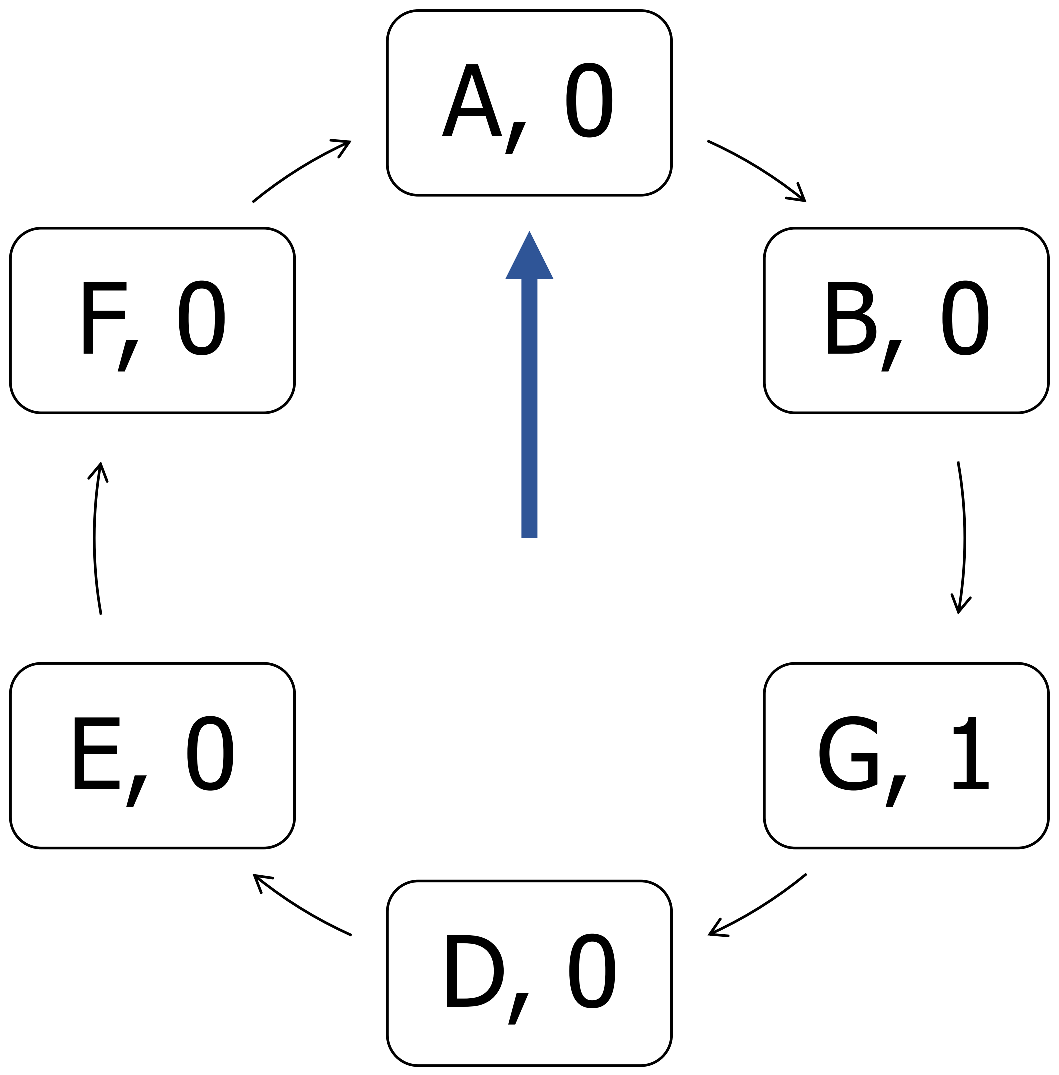
- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new
- D recently accessed
- E recently accessed
- F recently accessed
- A gets swapped!



Clock algorithm example

- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new
- D recently accessed
- E recently accessed
- F recently accessed
- A gets swapped
  - Was A or B the actual LRU?
  - Probably doesn't matter

(Page Name, Accessed Bit)



Clock algorithm is actually used in real computers

- Modern OSes often use some variation on Clock Algorithm

- x86 hardware supports an accessed bit in page table entries

- Clock algorithm could be built without hardware support too
  - Mark all pages as valid but not present initially (soft page fault)
  - On OS fault, update accessed bit for page, mark as present
    - Only fault on *first* access per clock-hand-cycle
  - Reset page to not present whenever accessed is reset to zero

Improving clock algorithm access notion

- Add multiple “accessed” bits to create accessed counter
  - Increment or decrement bits on use or clock-hand-pass respectively
  - Only remove pages with 0 accessed (or less than some minimum)

- Combine with timestamp notion to ensure page is “old” (WSClock)
  - Keep a timestamp in addition to accessed bit
  - Only remove pages with 0 accessed and older than some amount
    - Still not necessarily oldest, but definitely old

Improving clock algorithm evictions

- Keep track of number of times a page re-enters memory (Clock-PRO)
  - Give eviction preference to pages that haven’t been brought back a bunch
  - Bringing it back implies it was important, even if it was old

- Keep track of which pages are dirty
  - Give eviction preference to clean pages (also to read-only pages)
  - Means no write to disk is necessary!

- Evict several pages at once each time it is required
  - Find first N with accessed bit of zero
  - Takes advantage of disk I/O properties

Break + Administrivia

- Device Driver Lab due Thursday
  - Remember that you do have three slip days across the quarter

- Last lab is implementing Paging Virtual Memory for Nautilus
  - Should be released on Thursday

Outline

- Swapping
  - Overview
  - When To Swap
  - Page Replacement Policies
  - Implementing LRU

- **RAID**

Failure rates for disks are a serious problem

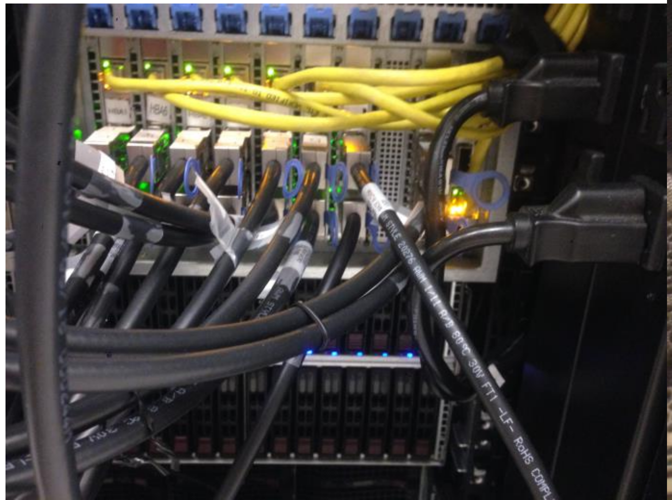
- Problem: disks fail
  - HDDs have physical actuators that wear out
  - SSDs have limited numbers of writes

- Big problem: servers have many disks
  - Assume rate of failure per year of disk is 1%
    - And failures aren’t correlated
  - And a server has 264 disks
  - What are the odds that a disk will fail this year?

  - $1 – (1 – 0.1)^{264}$ = 93% odds that at least one disk will fail

Database server at Northwestern

- 264 fast (10k RPM) magnetic disks (for production)
- 56 slow (7200 RPM) magnetic disks (for backup)
- ~150 TB storage capacity
- Comprised of 6 physical chassis (boxes) in one big cabinet, about the size of a coat closet.



Redundant Array of Independent Disks (RAID)

- Observation in 1988 (Patterson, Gibson, Katz)
  - Servers could use a high-quality mainframe disk drive

    OR for the same cost

  - Servers could use several redundant low-quality consumer disk drives

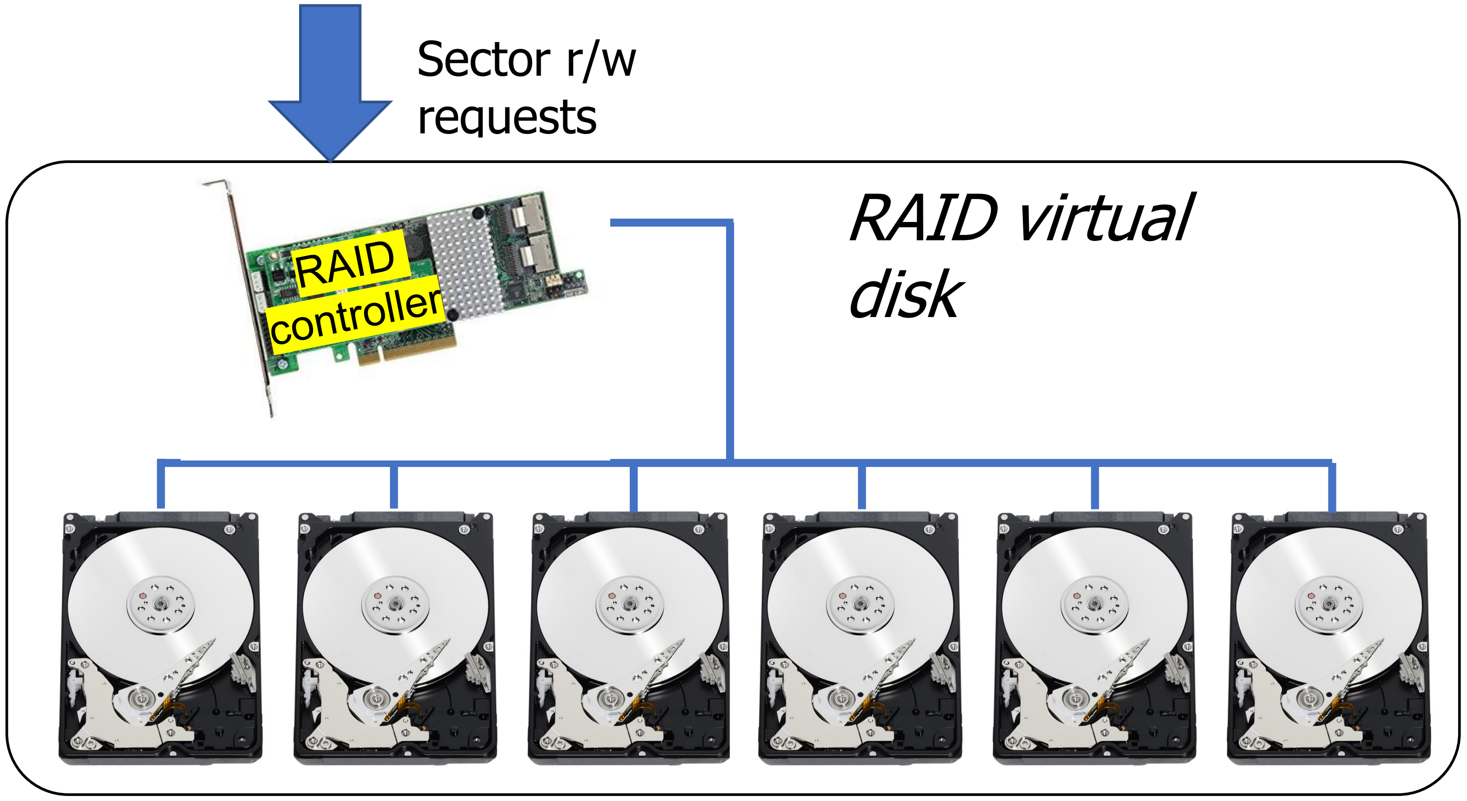
- Furthermore using an array of disks improves multiple things
  - Reduce impact of a ***failure*** by storing data redundantly on multiple disks.
  - Increase ***capacity*** by making multiple disks available to store data.
  - Increase ***throughput*** by accessing data in *parallel* on multiple disks.

Basic idea of RAID

- Combine many disks to create one *superior* virtual disk.
- The RAID array provides the same interface as a single disk.

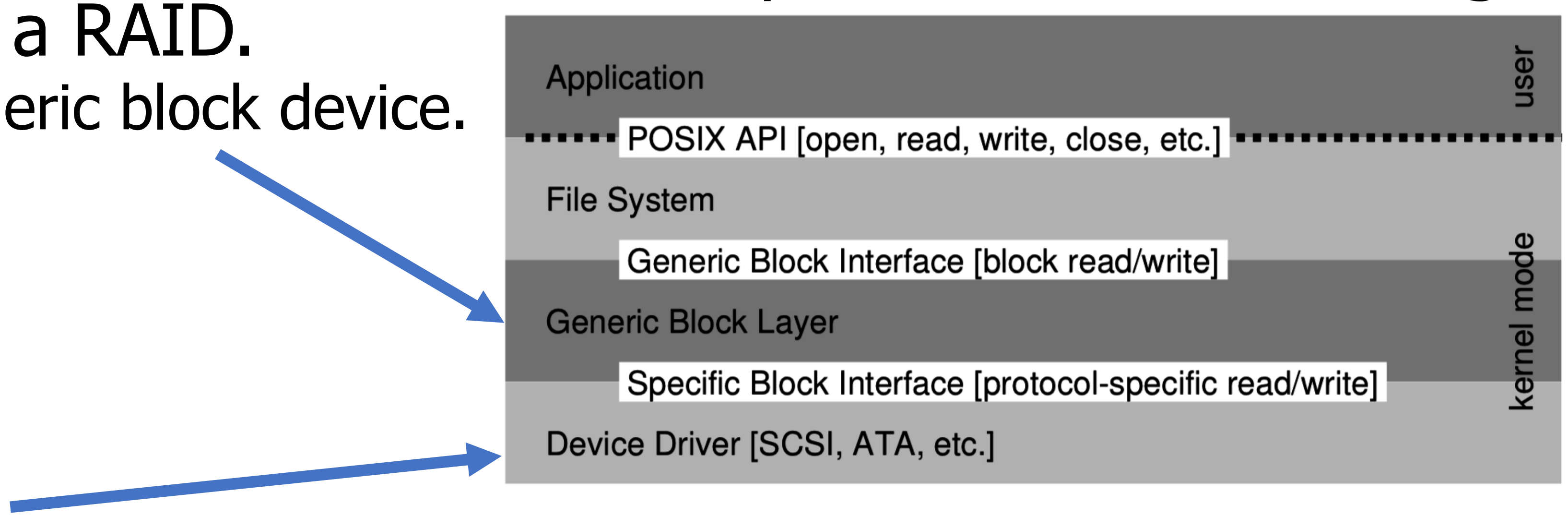
Computer thinks it's dealing with this:

But it's just an illusion. The reality is:



How does RAID fit into the OS?

- RAID can be implemented in software or hardware
- ***Software RAID*** means that the OS is responsible for assembling multiple disks into a RAID.
    - Implements a generic block device.



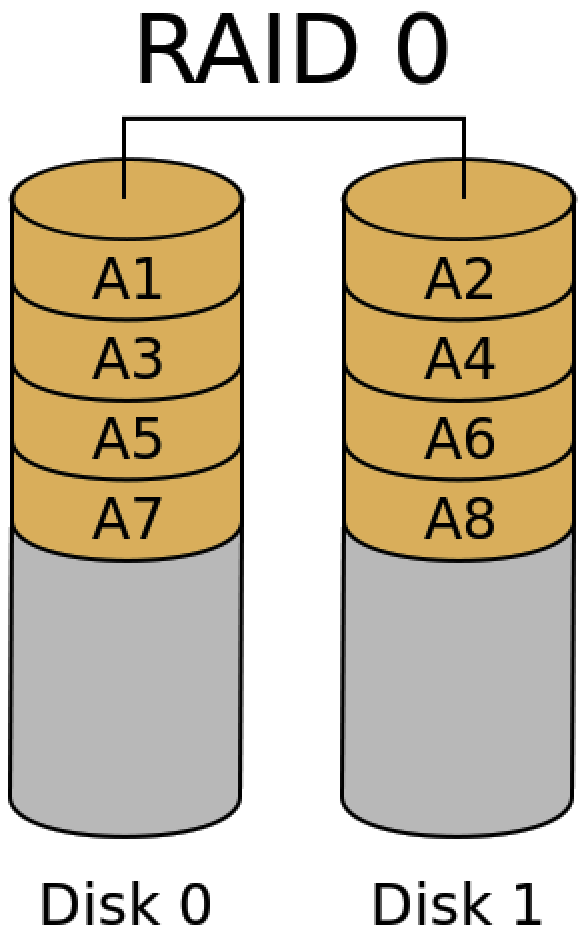
Application
user
POSIX API [open, read, write, close, etc.]
File System
Generic Block Interface [block read/write]
Generic Block Layer
kernel mode
Specific Block Interface [protocol-specific read/write]
Device Driver [SCSI, ATA, etc.]

- ***Hardware RAID*** requires a specialized controller card that coordinates the multiple disks, presenting interface of one disk.
    - OS just needs a driver for the RAID controller, like any other disk controller.

RAID levels

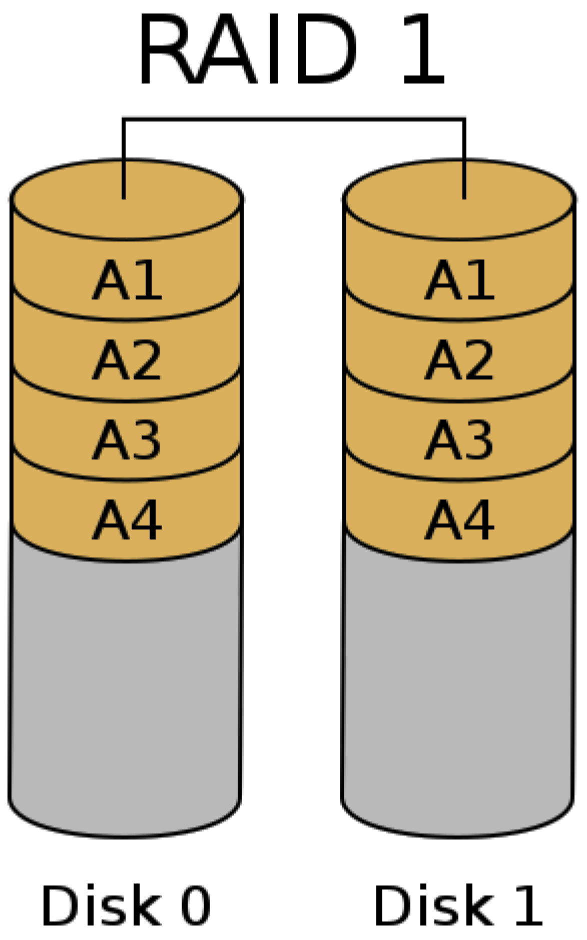
- RAID 0 – ***Striping***:
  - Distribute data across 2 disks for twice the peak throughput.
- RAID 1 – ***Mirroring***:
  - Copy data onto 2 disks to tolerate failure of one.
- RAID 4/5/6 – ***Parity:***
  - Keep parity bits around for each block to check for errors and rebuild.
  - Typically involves 3+ disks.

RAID 0 – Striping *(for throughput and capacity)*



- Divide the logical disk into chunks (A1, A2, A3 …) 1 or more blocks in size
- Distribute the chunks regularly over two or more (*N*) physical disks.
- (+) Throughput for both random and sequential access scales with N.
  $T_{RAID0} = N * T_{disk}$
- (+) Capacity also scales by N.
- (+) Cost per byte is identical
- (–) But Mean Time To Failure is worse because failure of a single disk is catastrophic:
  $MTTF_{RAID0} = MTTF_{disk}/N$

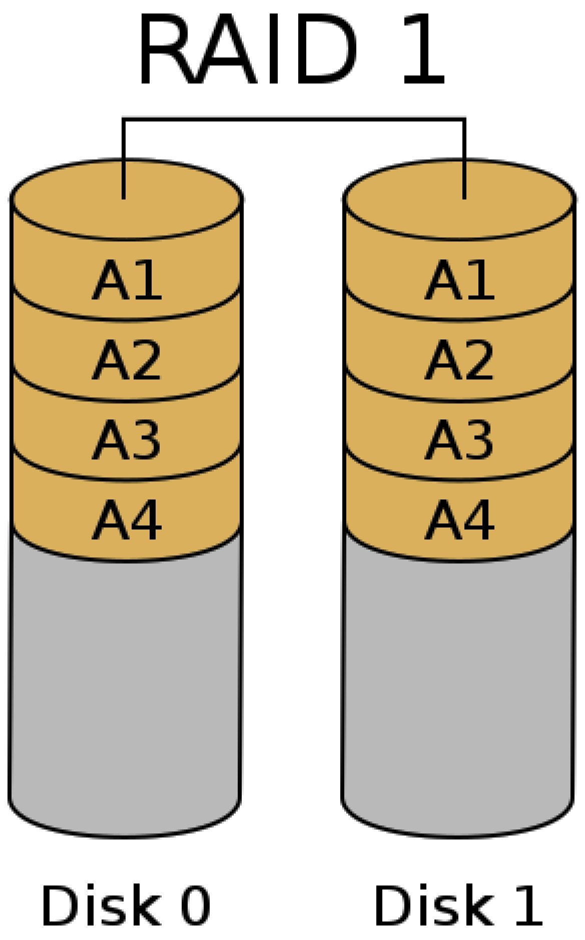
RAID 1 – Mirroring *(for fault tolerance)*



- Duplicate each chunk on each of N physical disks.
- (+) It is impossible to lose data unless all disks fail simultaneously.
  - i.e., failure window is reduced to the time it takes to replace a broken disk.
- (–) Write throughput is not improved
- (–) Capacity is the same as a single disk
- (–) Cost per byte is greater

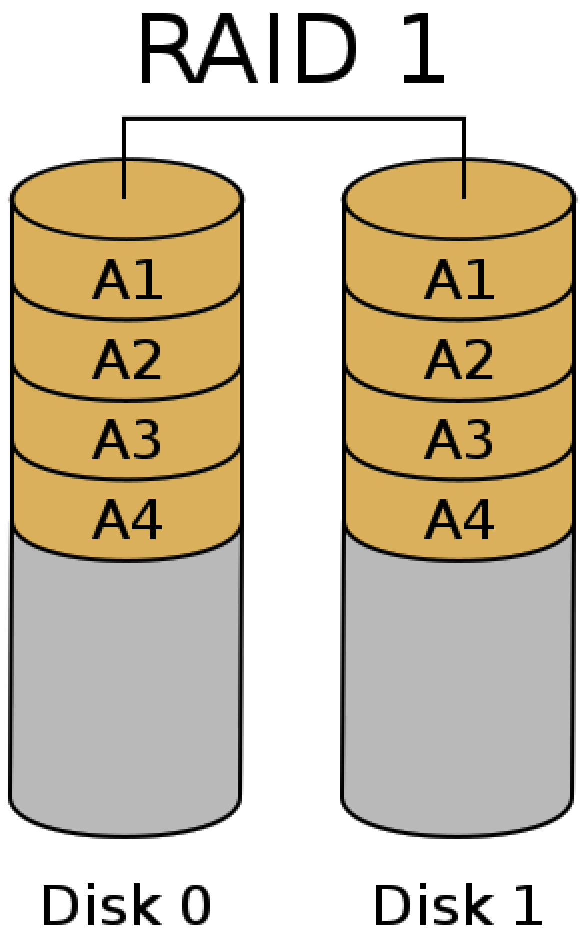
$$\$_{RAID1} = N * \$_{disk}$$

Check your understanding – RAID 1



- (–) Write throughput is not improved
- Is write throughput reduced in RAID 1? Or is it the same as a single disk?
- What about read throughput?

Check your understanding – RAID 1

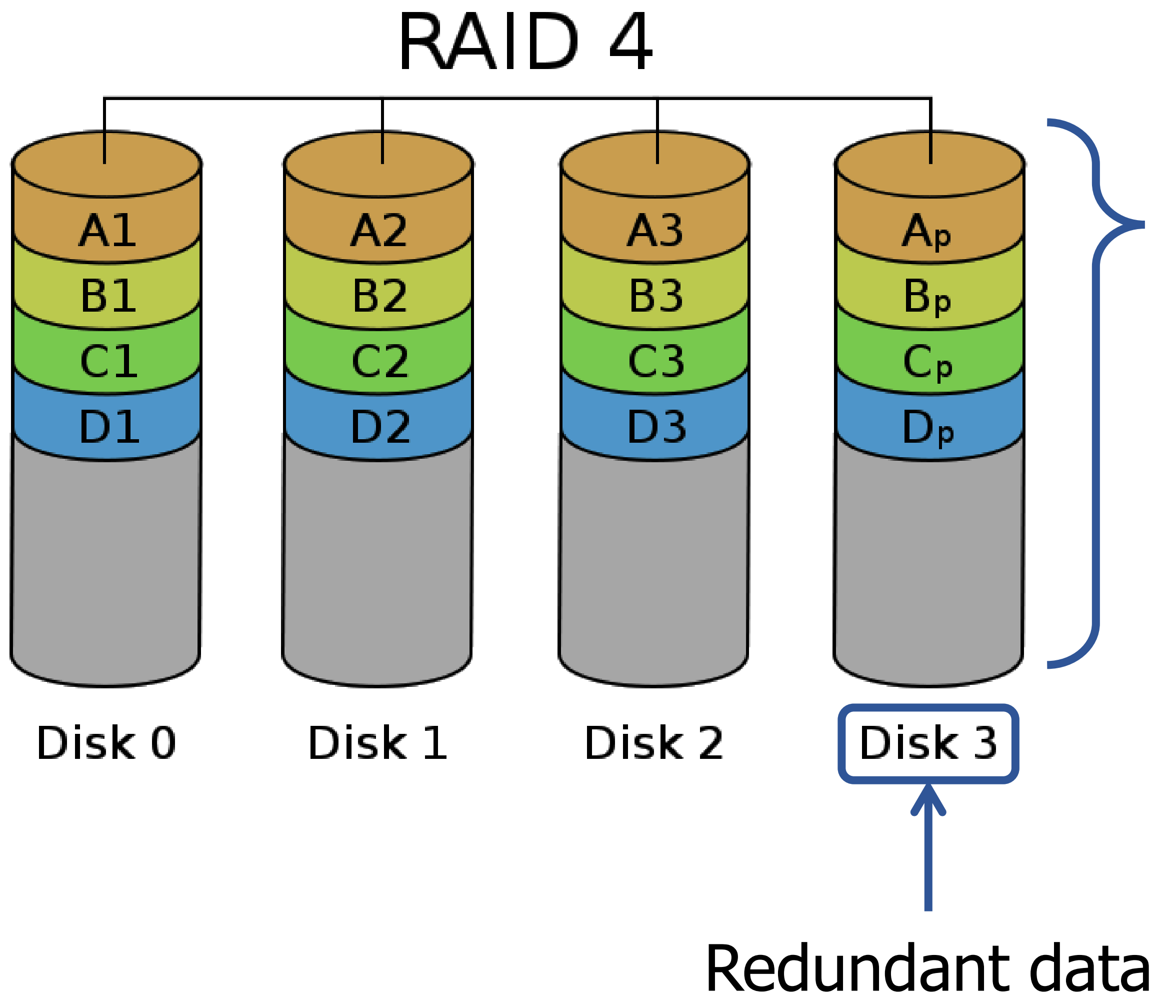


- (–) Write throughput is not improved

- Is write throughput reduced in RAID 1? Or is it the same as a single disk?
  - Same as a single disk
  - Write can go to both disks in parallel

- What about read throughput?
  - Better than a single disk
  - Can read two different blocks at once!

RAID 4 – Parity *(for fault tolerance, capacity & throughput)*



- Distribute the chunks across the first (N-1) disks.
- On the Nth disk, store a corresponding ***parity*** chunk.
  - Parity block is redundant data about a set of chunk (a ***stripe***)
- Can tolerate loss of any one disk
- Parity disk becomes bottleneck for writes limiting throughput

How does parity work?

- ***Even parity*** – add a 0 or 1 such that the total number of 1's is even.
  - There also exists odd parity which makes the total number of 1's odd
- Examples (Even Parity):
  - 0b0000_0000 – zero ones -> parity bit = 0
  - 0b1111_1111 – eight ones -> parity bit = 0
  - 0b0110_1101 – five ones -> parity bit = 1
- If a single bit is lost, the parity bit allows us to infer the value of the lost bit

Check your understanding – Parity Recovery

- What are the values of the missing bits?
  - Even Parity: make the total number of 1s even

- [0, 0, 1, 0, **?**, 0, 1, 1] – Even Parity with value: 1

- [0, **?**, 1, 1, 1, 0, 0, 0] – Even Parity with value: 0

Check your understanding – Parity Recovery

- What are the values of the missing bits?
    - Even Parity: make the total number of 1s even

- [0, 0, 1, 0, **?**, 0, 1, 1] – Even Parity with value: 1
    - Value must be a 0
    - Because parity plus ones is already even

- [0, **?**, 1, 1, 1, 0, 0, 0] – Even Parity with value: 0
    - Value must be a 1
    - Because parity plus ones is not currently even

Parity can only fix a single error

- What if two bits are missing?

- [**?**, 0, 1, 0, **?**, 0, 1, 1] – Even Parity: 1
  - Could both be zeros
  - Could both be ones
  - Impossible to tell which

- More advanced “error correcting codes” are possible to detect/fix two or more errors
  - Hamming Code (single error correcting, double error detecting)

Parity chunk in RAID

Disk 0	Disk 1	Disk 2	**Disk 3 (parity)**
0001 0010 1100 1100	0000 1111 0000 1111	1101 1111 0011 0001	**1100 0010 1111 0010**
1111 1111 1111 1111	0001 0001 0001 0001	1101 1001 0110 0110	**0011 0111 1000 1000**
0000 0000 0000 0000	1101 1011 0011 0011	1111 0011 0011 1000	**0010 1000 0000 1011**

Useful storage capacity

Redundancy overhead

- Parity is computed bit-wise across corresponding chunks.
- Chunks are one or more blocks (multiple of 4 kB) in size
- Writing a small file will involve one disk *plus the parity disk.*
    - (parity disk can become a bottleneck)
- Writing a large file will involve all the disks.

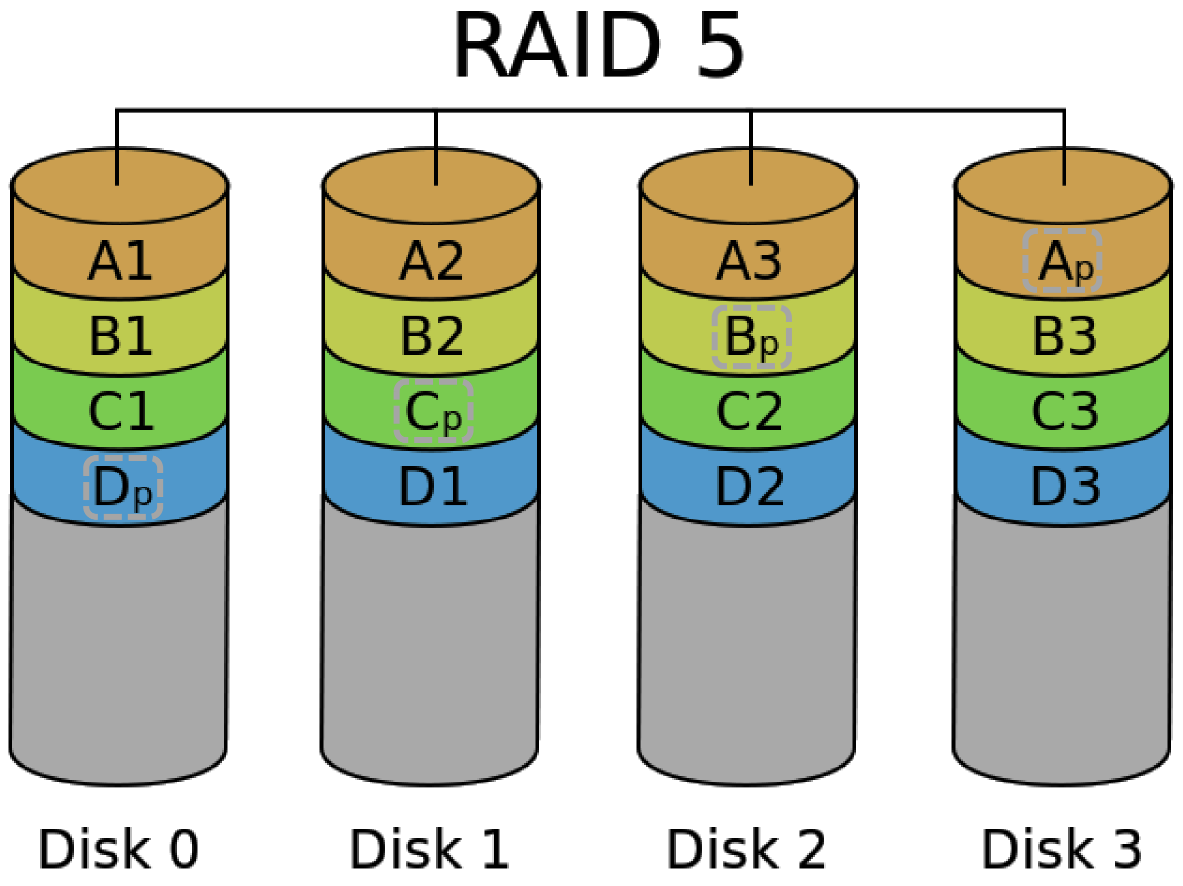
Rebuilding an array after failure

Disk 0	Disk 1	Disk 2	**Disk 3 (parity)**
0001 0010 1100 1100		1101 1111 0011 0001	**1100 0010 1111 0010**
1111 1111 1111 1111		1101 1001 0110 0110	**0011 0111 1000 1000**
0000 0000 0000 0000		1111 0011 0011 1000	**0010 1000 0000 1011**

Disk failed!

- If a disk fails, then we remove it and replace it with a working disk.
- Then scan through the entire array to compute and write missing data.
  - This is called "rebuilding" the array
  - We cannot tolerate another disk failure until rebuild completes.
  - Reads/writes can continue while array is rebuilding!

RAID 5 – Distributed Parity *(the winner in practice)*



- Distribute parity chunks across the disks, to avoid a small-write bottleneck
- (+) Failure of one disk is OK
- (+) Throughput is good

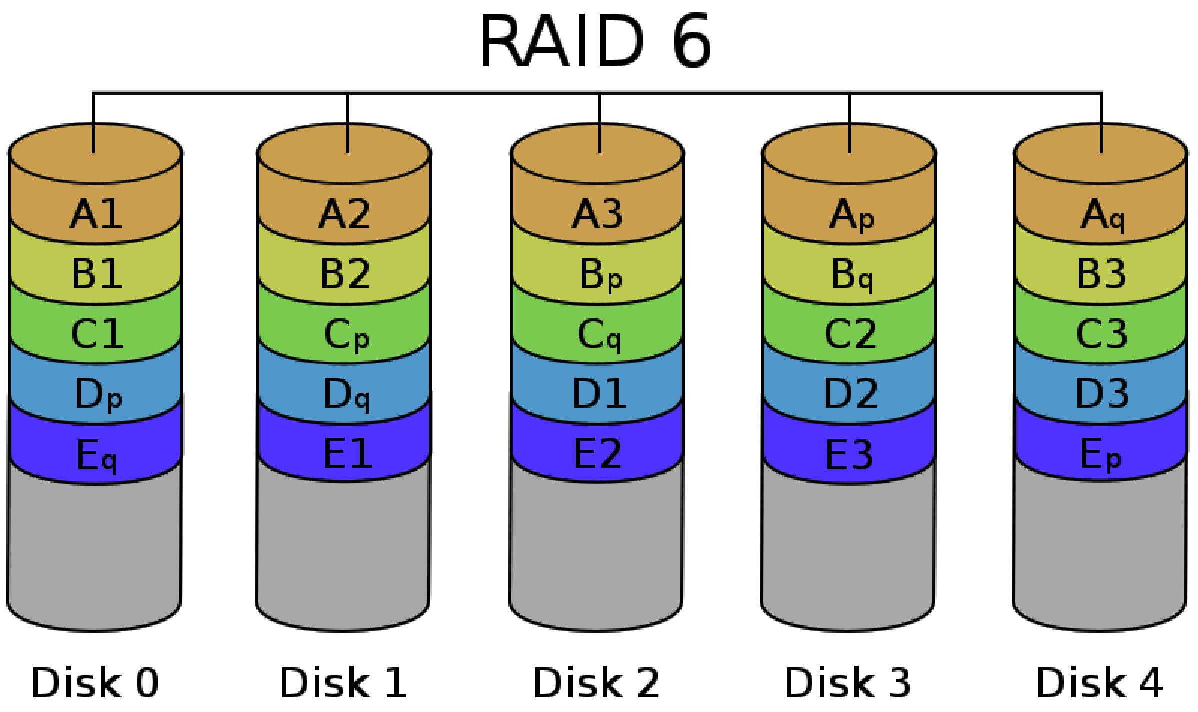
$$T_{RAID5} = (N-1) * T_{disk}$$

- (+) Cost per byte is good

$$\$_{RAID1} = N/(N-1) * \$_{disk}$$

- (–) High overhead for small N
- (–) Failure risk is high for large N
- N is typically 3 to 8

RAID 6 – Double Parity *(for large arrays)*



- Add another disk and keep two parity chunks per stripe
  - 2nd parity is computed differently
- (+) Failure of ***two*** disks is OK
- (~) Throughput is less:

$$T_{RAID5} = (N-2) * T_{disk}$$

- (~) Cost per byte is higher:

$$\$_{RAID1} = N/(N-2) * \$_{disk}$$

- Makes sense for larger N (>8)

Outline

- Swapping
  - Overview
  - When To Swap
  - Page Replacement Policies
  - Implementing LRU
- RAID