

# ECE/CS 250 Computer Architecture

## Summer 2018

### Virtual Memory

Tyler Bletsch  
Duke University

Slides from work by  
Daniel J. Sorin (Duke), Amir Roth (Penn), and Alvin Lebeck (Duke)

Includes material adapted from Operating System Concepts by  
Silberschatz, Galvin, and Gagne

# I. CONCEPT

# Motivation (1)

- I want to run more than one program at once
- Problem:
  - Program A puts its variable X at address 0x1000
  - Program B puts its variable Q at address 0x1000
  - Conflict!
- Unlikely solution:
  - Get all programmers on the planet to use different memory addresses
- Better solution:
  - Allow each running program to have its own “view” of memory (e.g. “my address 0x1000 is different from your address 0x1000”)
- How? Add a layer of **indirection** to memory addressing:  
**Virtual memory paging**

## Motivation (2)

- Hey, *while you're messing with memory addressing...* can we improve efficiency, too?
- Code/data must be in memory to execute
- Most code/data not needed at a given instant
- Wasteful use of DRAM to hold stuff we don't need
- Solution:
  - Don't bother to load code/data from disk that we don't need immediately
  - When memory gets tight, shove loaded stuff we don't need anymore back to disk
- **Virtual memory swapping** (an add-on to paging)

# Benefits

## Paging benefits:

- **Simpler programs:**
  - We “virtualize” memory addresses, so every program believes it has a full  $2^{32}$  or  $2^{64}$  byte address space
- **Easier sharing:**
  - Processes can “share” memory, i.e. have virtual addresses that map to the same physical addresses

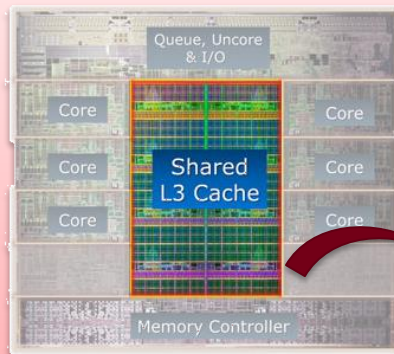
## Swapping benefits:

- **Bigger programs:**
  - Not just bigger than free memory, bigger than AVAILABLE memory!
- **More programs:**
  - Only part of each program loaded, so more can be loaded at once
- **Faster multitasking:**
  - Less effort to load or swap processes

# Figure: caching vs. virtual memory

**CACHING**

Copy if **popular**



**Cache**

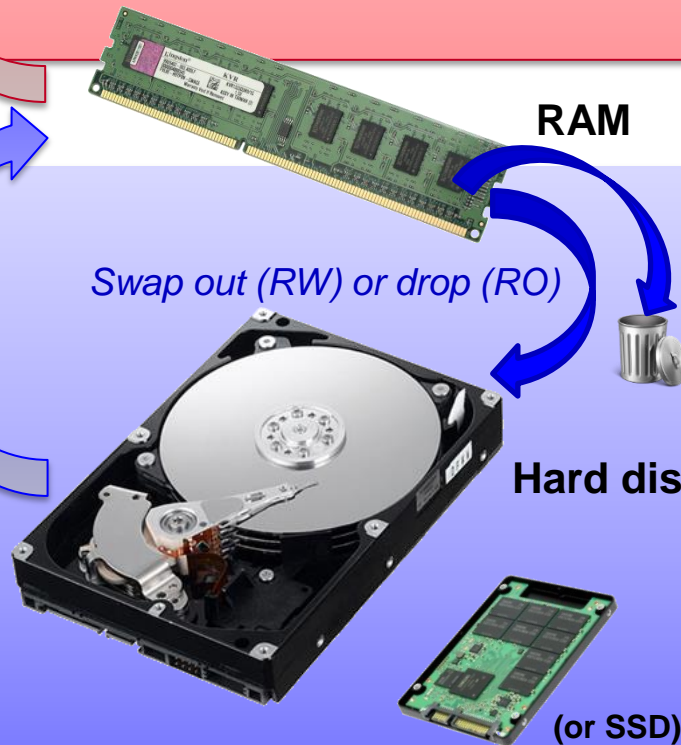
*Drop*

- Faster
- More expensive
- Lower capacity

**SWAPPING**

Load if **needed**

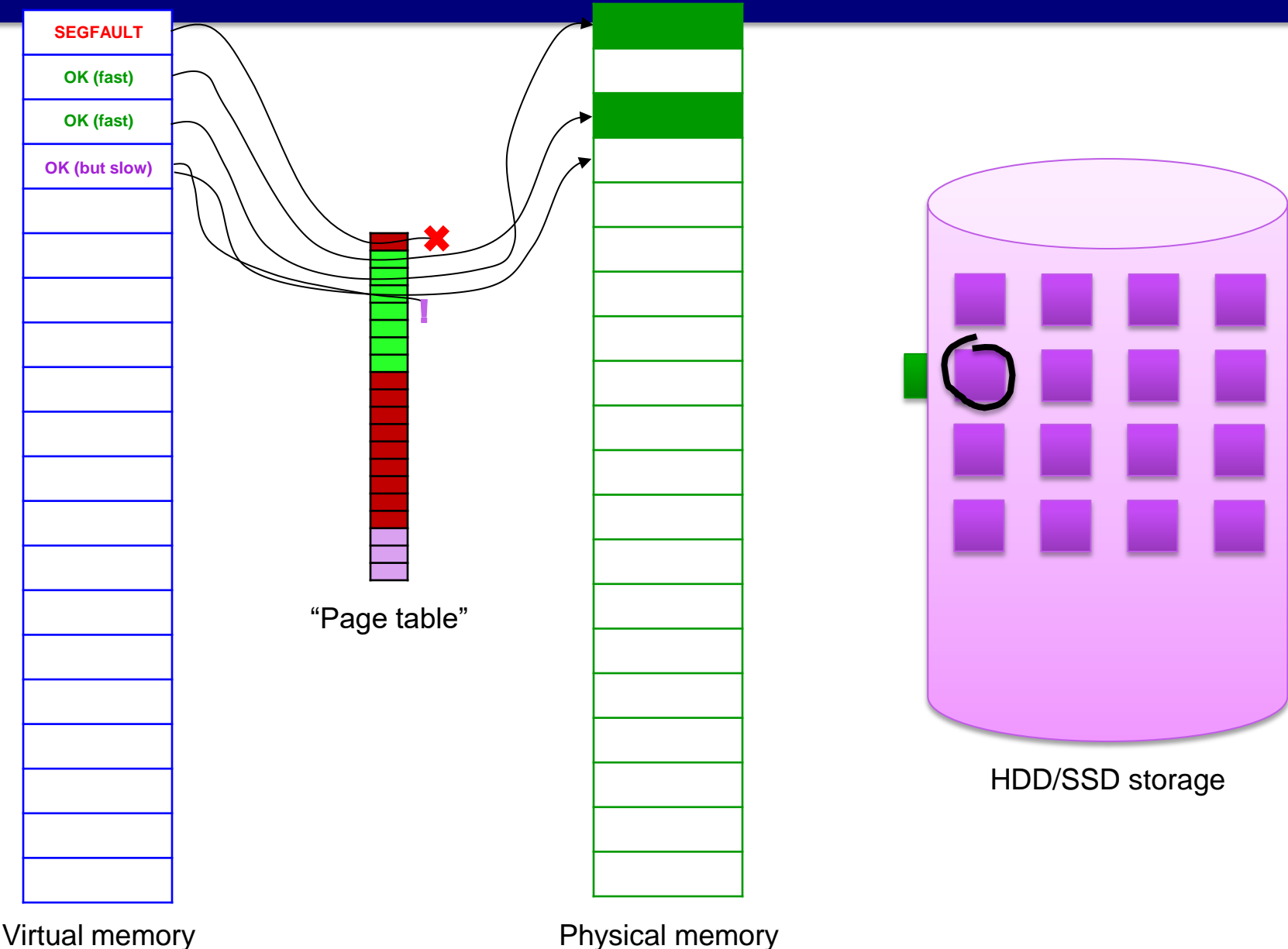
*Swap out (RW) or drop (RO)*



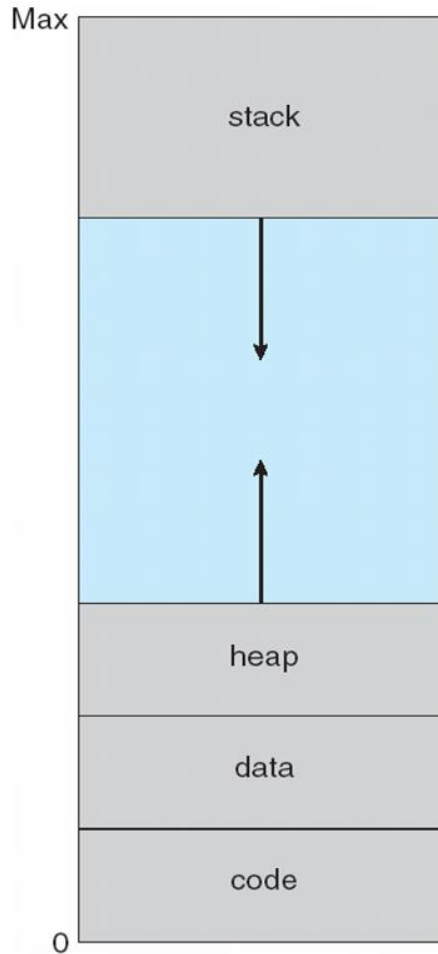
**Hard disk**

- Slower
- Cheaper
- Higher capacity

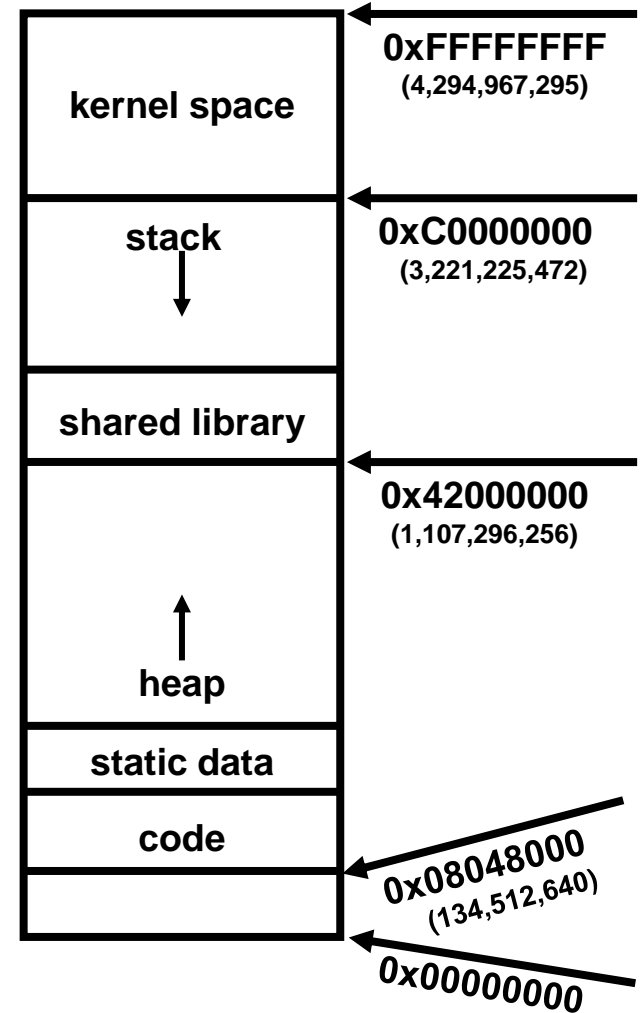
# High level operation



# Virtual Address Space



(Fluffy academic version)



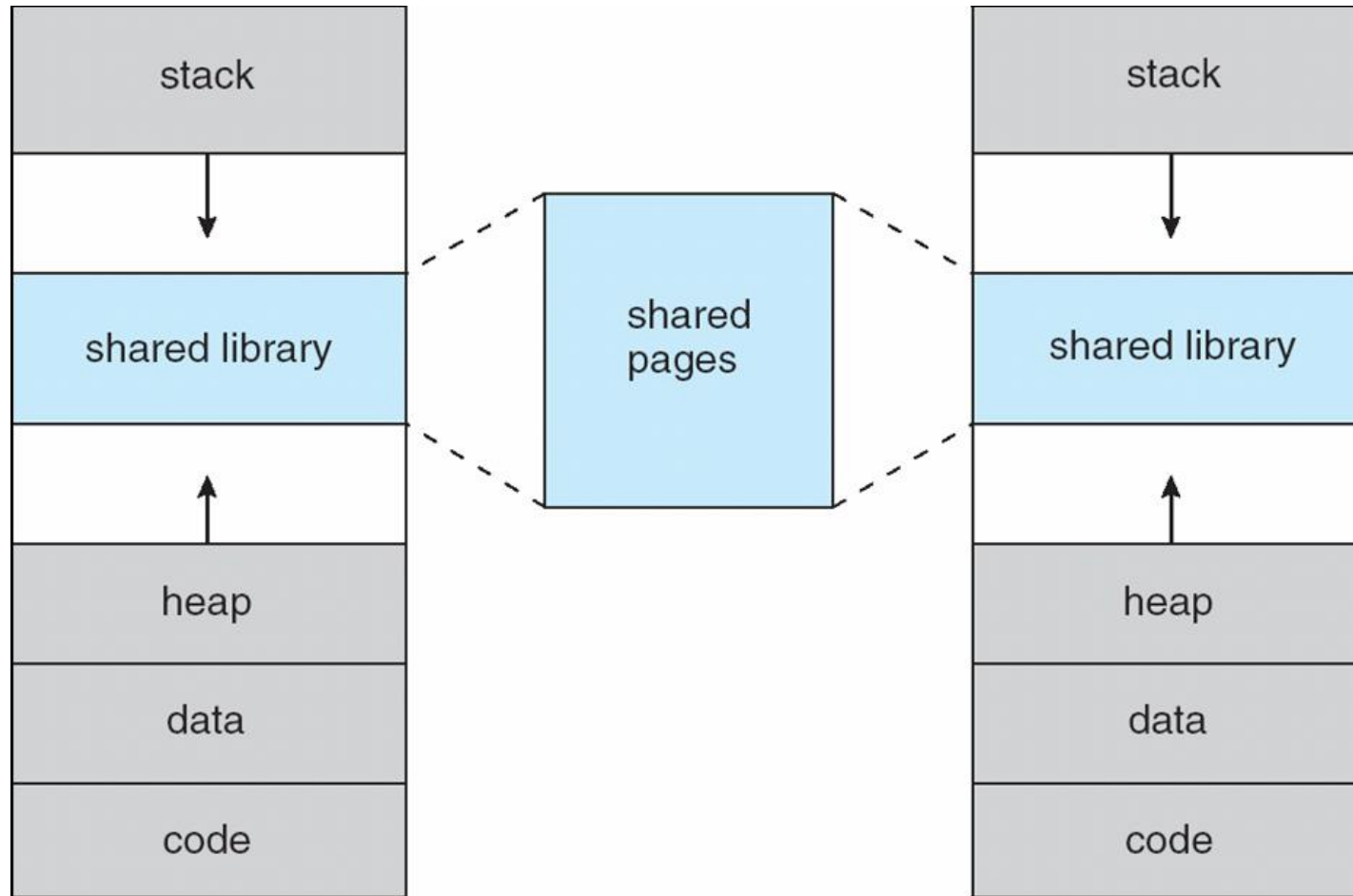
(Real 32-bit x86 view)



# Virtual Address Space

- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- **System libraries** shared via mapping into virtual address space
- **Shared memory** by mapping pages read-write into virtual address space
  - Pages can be shared during `fork()`, speeding process creation

# Shared Library Using Virtual Memory



# II. MECHANICS

# Mechanisms and Policies

- Mechanisms:
  - **Demand paging**: fixed-size regions
  - **Demand segmentation**: variable-size regions

^ Not covered in this lecture

- Policies:
  - When and how much to load?
  - How to select data to evict?

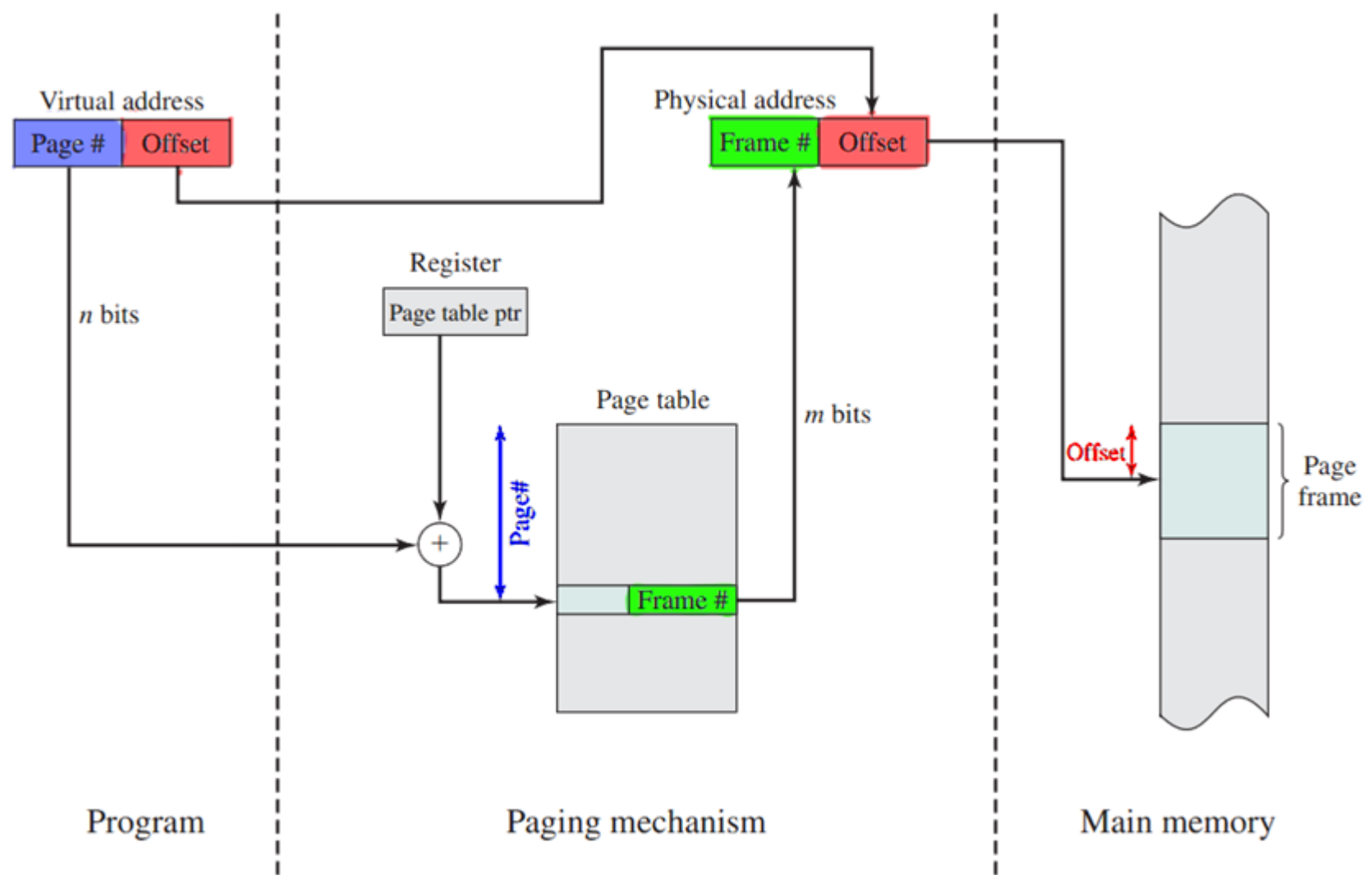
# Demand Paging

## Page

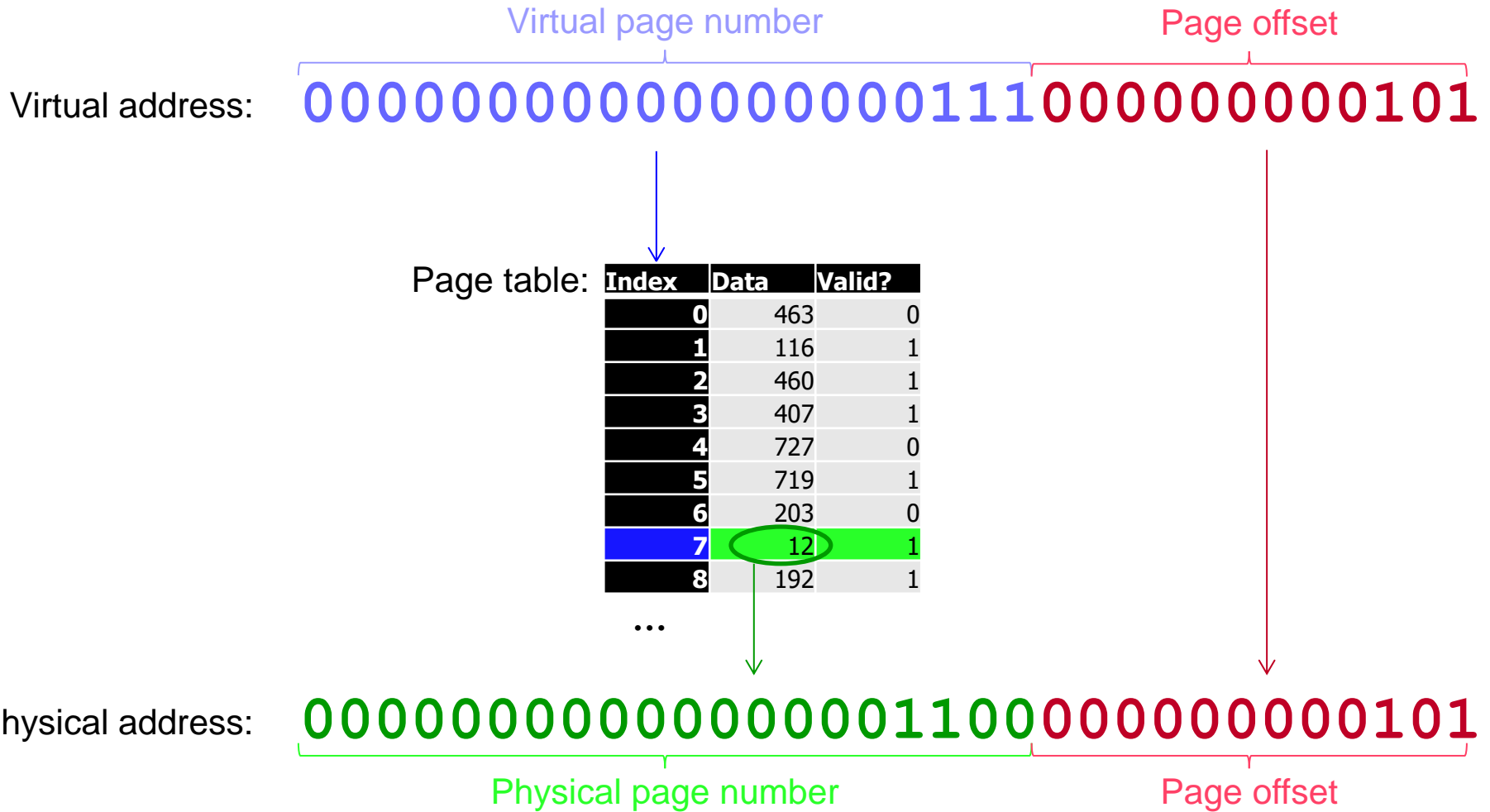
A chunk of memory with its own record in the memory management hardware. Often 4kB.



# Address translation



# Address translation



# Address translation

- Equivalent code (except this is done in hardware, not code!):  
Assume pages are 4096 bytes, so 12 bits for offset, 20 for page number

```
int page_table[1<<VIRT_PAGE_NUMBER_BITS];

int virt2phys(int virt_addr) {
    int offset = virt_addr & 0xFFF; // lower 12 bits
    int vpn = virt_addr >> 12; // upper 20 bits

    int ppn = page_table[vpn]; // table lookup

    if (ppn == INVALID) DO_SEGFAULT_EXCEPTION();

    int phys_addr = (ppn<<12) | offset; // combine fields
    return phys_addr;
}
```



# Address translation

- Equivalent code (except this is done in hardware, not code!):  
Assume pages are 4096 bytes, so 12 bits for offset, 20 for page number

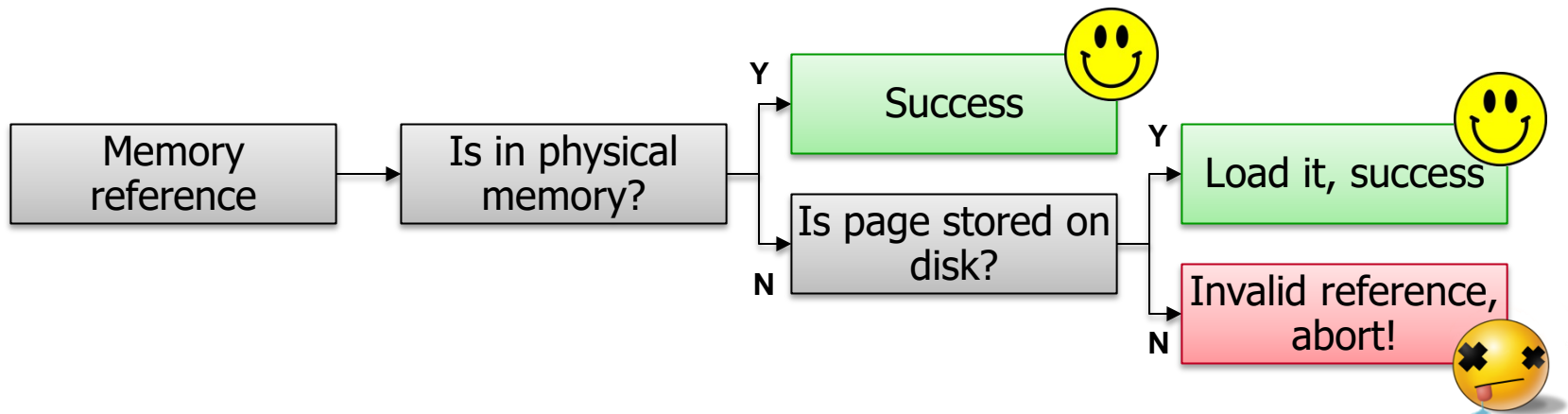
```
int P[VIRT_PAGE_NUMBER_BITS];

int virt2phys(int virt_addr) {
    return P[v_addr>>12]<<12 | v_addr&0xFFF;
    //      ^^--HIGH BITS--^^      ^-LOW BITS-^
}

```

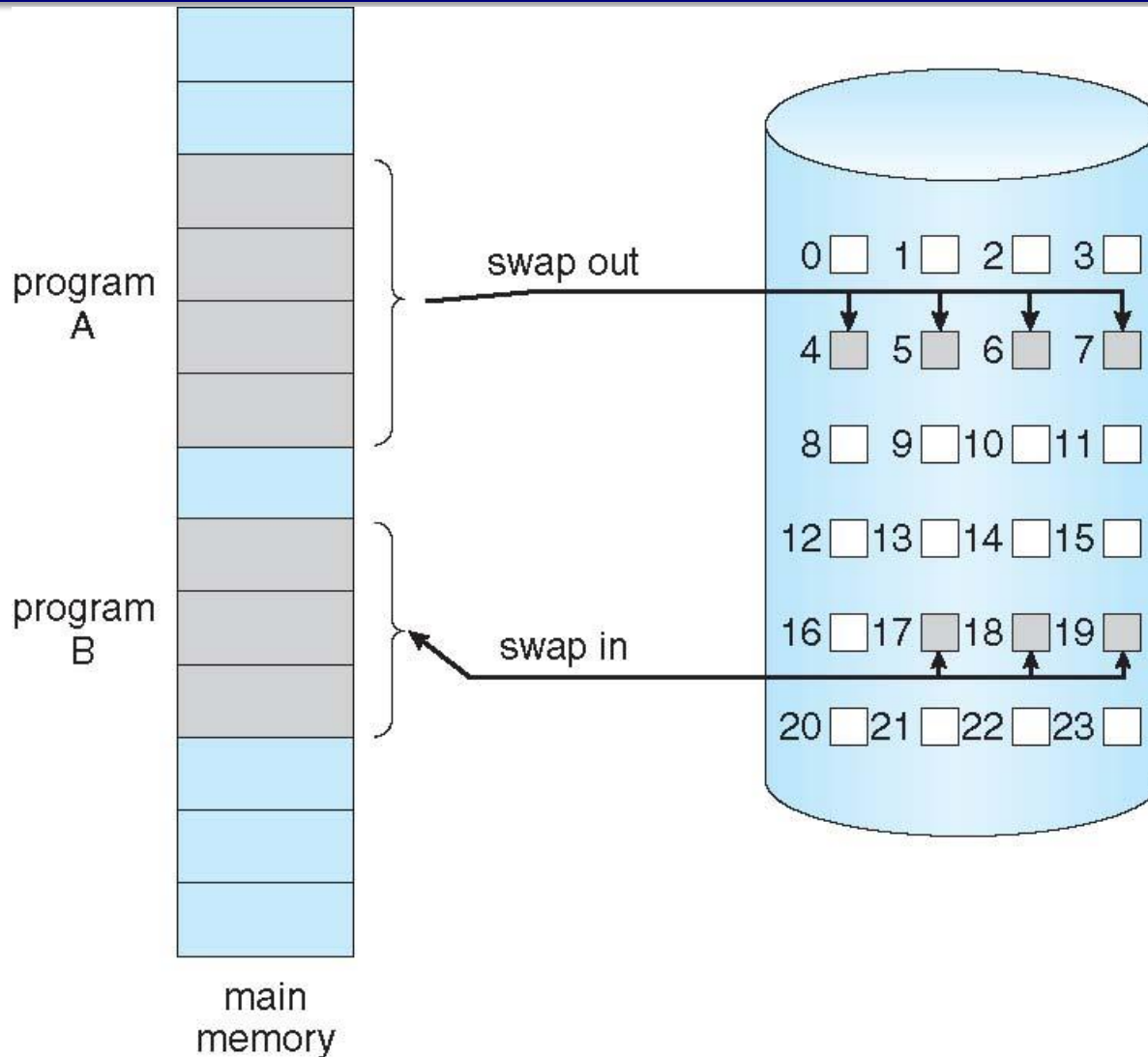
# Demand Paging

- On process load:
  - Rather than bring entire process into memory, bring it in page-by-page as needed
  - Less I/O needed, no unnecessary I/O



- The swapper process is called the **pager**

# Transfer of a Paged Memory to Contiguous Disk Space



# Valid-Invalid Bit

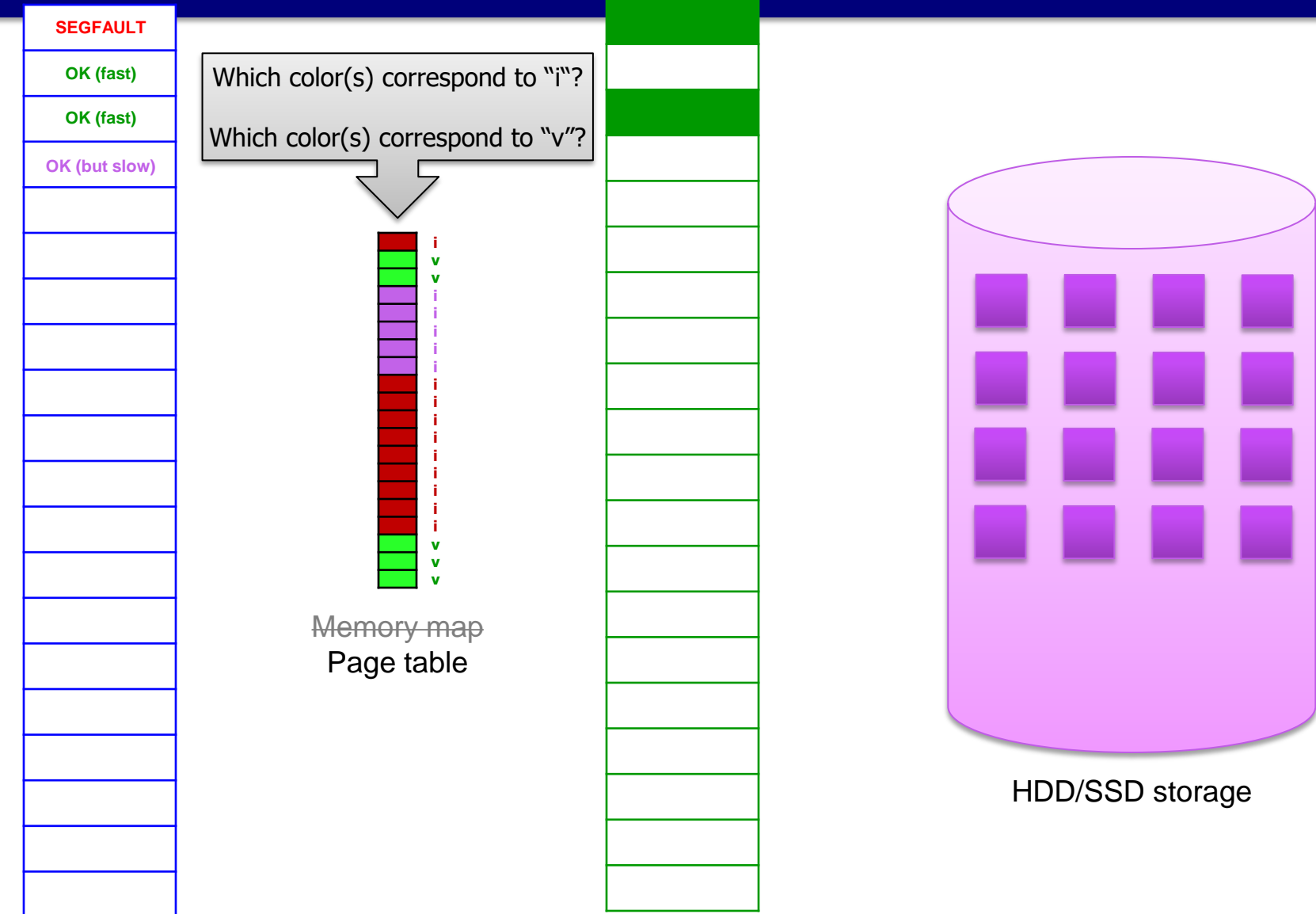
- With each page table entry a valid–invalid bit is associated ( $v \Rightarrow$  in-memory – memory resident,  $i \Rightarrow$  not-in-memory)
- Initially valid–invalid bit is set to  $i$  on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	<b>v</b>
	<b>v</b>
	<b>v</b>
	<b>v</b>
	<b>i</b>
....	
	<b>i</b>
	<b>i</b>

page table

- During address translation, if valid–invalid bit in page table entry is **i**  $\Rightarrow$  page fault

# High level operation



Virtual memory

Physical memory

HDD/SSD storage

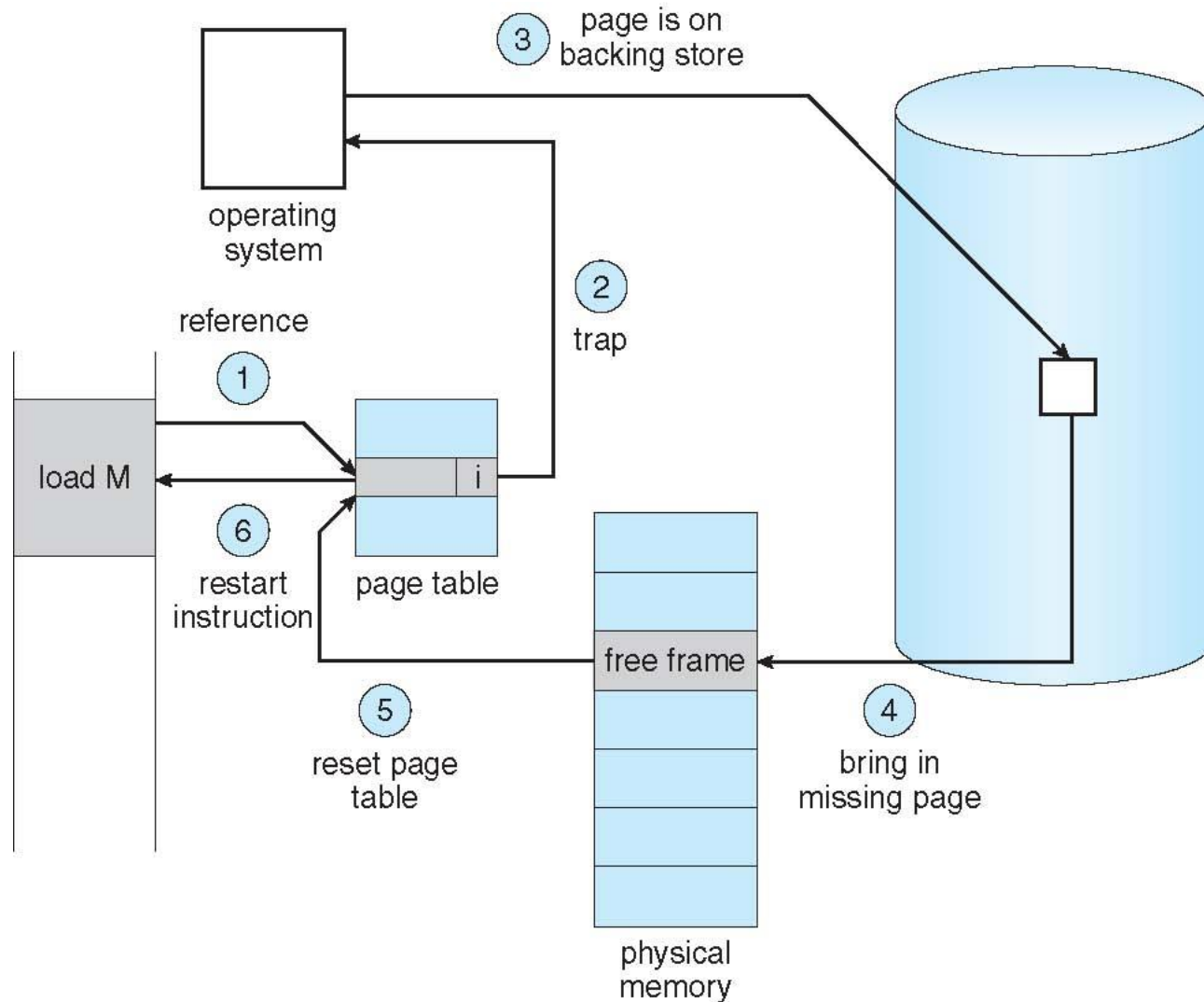
# Page Fault

- Reference to a page with bit set to “i”:  
**Page fault**
- OS looks at another table to decide:
  - Invalid reference? Abort.
- Just not in memory? Load it:
  - Get empty memory frame
  - Swap page into frame via scheduled disk operation
  - Reset tables to indicate page now in memory  
Set validation bit = “v”
  - Restart the instruction that caused the page fault

# Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

# Steps in Handling a Page Fault





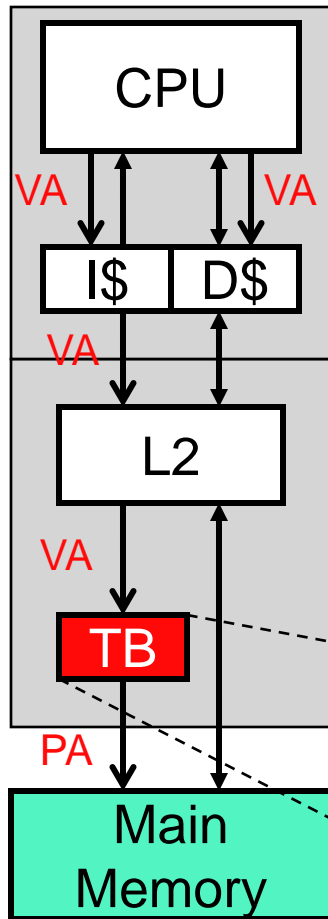
# Address Translation Mechanics

- The six questions
  - What? address translation
  - Why? compatibility, multi-programming, protection
  - How? page table
  - **Who performs it?**
  - **When?**
  - **Where does page table reside?**
- Option I: process (program) translates its own addresses
  - Page table resides in process visible virtual address space
    - Bad idea: implies that program (and programmer)...
      - ...must know about physical addresses
        - Isn't that what virtual memory is designed to avoid?
      - ...can forge physical addresses and mess with other programs
  - Translation on L2 miss or always? How would program know?

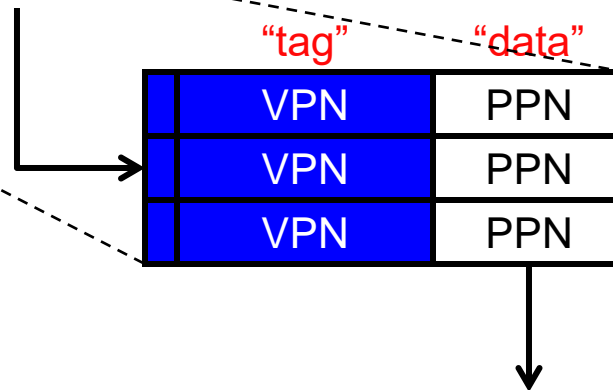
# Who? Where? When? Take II

- Option II: **operating system (OS)** translates for process
  - Page table resides in OS virtual address space
  - + User-level processes cannot view/modify their own tables
  - + User-level processes need not know about physical addresses
  - Translation on L2 miss
    - Otherwise, OS SYSCALL before any fetch, load, or store
- L2 miss: interrupt transfers control to OS handler
  - Handler translates VA by accessing process's page table
  - Accesses memory using PA
  - Returns to user process when L2 fill completes
  - Still slow: added interrupt handler and PT lookup to memory access
  - What if PT lookup itself requires memory access? Head spinning...

# Translation Buffer



- Functionality problem? Add indirection!
- Performance problem? Add cache!
- Address translation too slow?
  - Cache translations in **translation buffer (TB)**
    - Small cache: 16–64 entries, often fully assoc
    - + Exploits temporal locality in PT accesses
    - + OS handler only on TB miss



# TB Misses

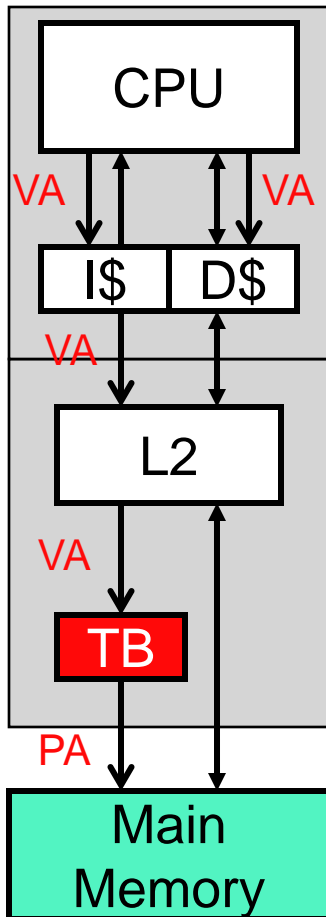
- **TB miss:** requested PTE not in TB, but in PT
  - Two ways of handling
- **1) OS routine:** reads PT, loads entry into TB (e.g., Alpha)
  - Privileged instructions in ISA for accessing TB directly
  - Latency: one or two memory accesses + OS call
- **2) Hardware FSM:** does same thing (e.g., IA-32)
  - Store PT root pointer in hardware register
  - Make PT root and 1st-level table pointers physical addresses
    - So FSM doesn't have to translate them

+ Latency: saves cost of OS call

# Page Faults

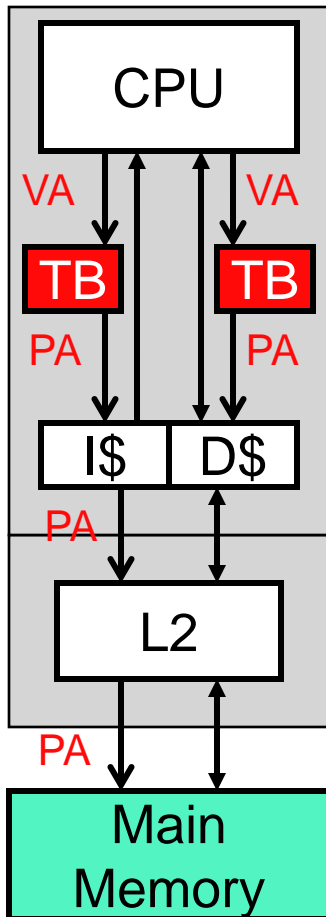
- **Page fault:** PTE not in TB or in PT
  - Page is simply not in memory
  - Starts out as a TB miss, detected by OS handler/hardware FSM
- **OS routine**
  - OS software chooses a physical page to replace
    - **“Working set”:** more refined software version of LRU
      - Tries to see which pages are actively being used
      - Balances needs of all current running applications
    - If dirty, write to disk (like dirty cache block with writeback \$)
  - Read missing page from disk (done by OS)
    - Takes so long (10ms), OS schedules another task
  - Treat like a normal TB miss from here

# Virtual Caches



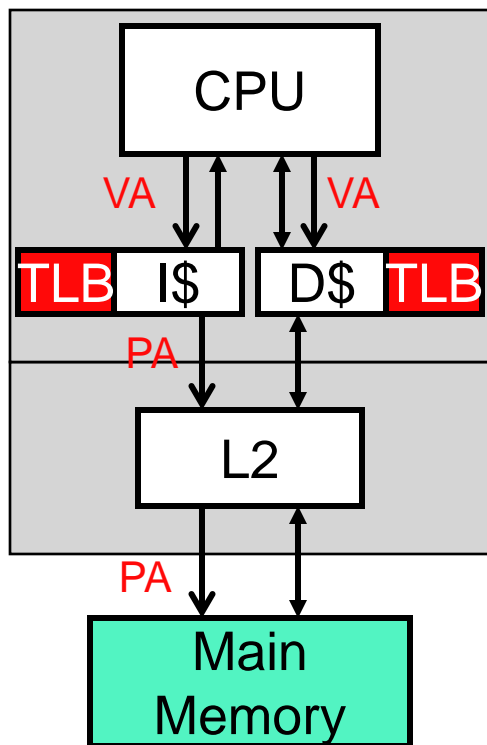
- Memory hierarchy so far: **virtual caches**
  - Indexed and tagged by VAs
  - Translate to PAs only to access memory
  - + Fast: avoids translation latency in common case
- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags
- Does inter-process communication work?
  - **Aliasing**: multiple VAs map to same PA
    - How are multiple cache copies kept in sync?
    - Also a problem for I/O (later in course)
  - Disallow caching of shared memory? Slow

# Physical Caches



- Alternatively: **physical caches**
  - Indexed and tagged by PAs
  - Translate to PA at the outset
- + No need to flush caches on process switches
  - Processes do not share PAs
- + Cached inter-process communication works
  - Single copy indexed by PA
- Slow: adds 1 cycle to  $t_{hit}$

# Virtual Physical Caches



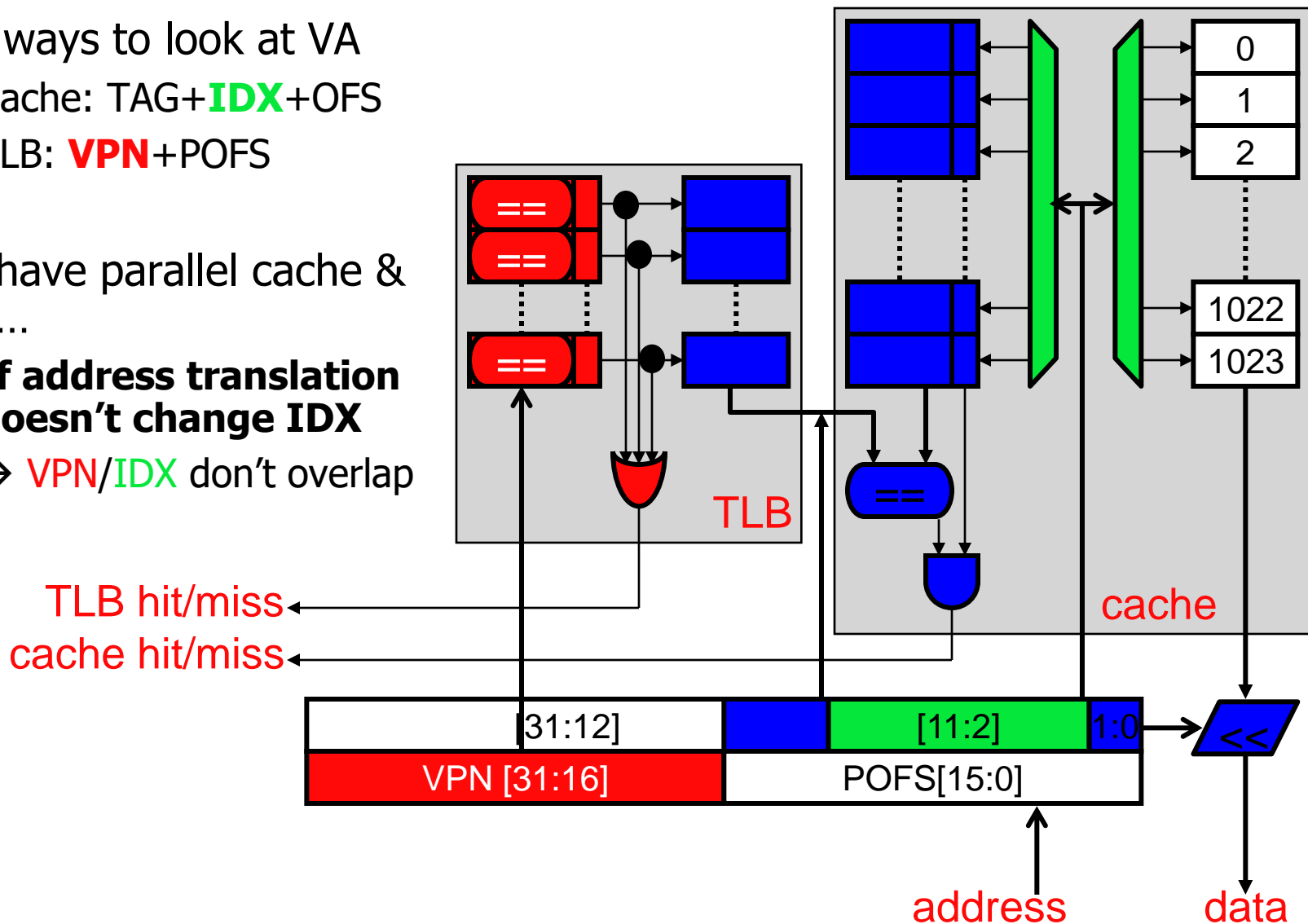
Compromise: **virtual-physical caches**

- Indexed by VAs
- Tagged by PAs
- Cache access and address translation in parallel
  - + No context-switching/aliasing problems
  - + Fast: no additional  $t_{hit}$  cycles
- A TB that acts in parallel with a cache is a **TLB**
  - **Translation Lookaside Buffer**
- Common organization in processors today



# Cache/TLB Access

- Two ways to look at VA
  - Cache: TAG+**IDX**+OFS
  - TLB: **VPN**+POFS
- Can have parallel cache & TLB ...
  - **If address translation doesn't change IDX**
  - → **VPN/IDX** don't overlap



# Cache Size And Page Size



- Relationship between page size and L1 I\$(D\$) size
  - Forced by non-overlap between VPN and IDX portions of VA
    - Which is required for TLB access
  - I\$(D\$) size / **associativity**  $\leq$  page size
  - Big caches must be set associative
    - Big cache  $\rightarrow$  more index bits (fewer tag bits)
    - More set associative  $\rightarrow$  fewer index bits (more tag bits)
  - Systems are moving towards bigger (64KB) pages
    - To amortize disk latency
    - To accommodate bigger caches

# III. DESIGN CHOICES AND PERFORMANCE

# The Table of Time

Event	Picoseconds	≈	Hardware/target	Source
Average instruction time*	30	30 ps	Intel Core i7 4770k (Haswell), 3.9GHz	<a href="https://en.wikipedia.org/wiki/Instructions_per_second">https://en.wikipedia.org/wiki/Instructions_per_second</a>
Time for light to traverse CPU core (~13mm)	44	40 ps	Intel Core i7 4770k (Haswell), 3.9GHz	<a href="http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested/5">http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested/5</a>
Clock cycle (3.9GHz)	256	300 ps	Intel Core i7 4770k (Haswell), 3.9GHz	Math
Memory read: L1 hit	1,212	1 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	<a href="http://www.7-cpu.com/cpu/SandyBridge.html">http://www.7-cpu.com/cpu/SandyBridge.html</a>
Memory read: L2 hit	3,636	4 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	<a href="http://www.7-cpu.com/cpu/SandyBridge.html">http://www.7-cpu.com/cpu/SandyBridge.html</a>
Memory read: L3 hit	8,439	8 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	<a href="http://www.7-cpu.com/cpu/SandyBridge.html">http://www.7-cpu.com/cpu/SandyBridge.html</a>
Memory read: DRAM	64,485	60 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	<a href="http://www.7-cpu.com/cpu/SandyBridge.html">http://www.7-cpu.com/cpu/SandyBridge.html</a>
Process context switch or system call	3,000,000	3 us	Intel E5-2620 (Sandy Bridge), 2GHz	<a href="http://blog.tsuninet.net/2010/11/how-long-does-it-take-to-make-context.html">http://blog.tsuninet.net/2010/11/how-long-does-it-take-to-make-context.html</a>
Storage sequential read**, 4kB (SSD)	7,233,796	7 us	SSD: Samsung 840 500GB	<a href="http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html">http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html</a>
Storage sequential read**, 4kB (HDD)	65,104,167	70 us	HDD: 2.5" 500GB 7200RPM	<a href="http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html">http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html</a>
Storage random read, 4kB (SSD)	100,000,000	100 us	SSD: Samsung 840 500GB	<a href="http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html">http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html</a>
Storage random read, 4kB (HDD)	10,000,000,000	10 ms	HDD: 2.5" 500GB 7200RPM	<a href="http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html">http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html</a>
Internet latency, Raleigh home to NCSU (3 mi)	21,000,000,000	20 ms	courses.ncsu.edu	Ping
Internet latency, Raleigh home to Chicago ISP (639 mi)	48,000,000,000	50 ms	dls.net	Ping
Internet latency, Raleigh home to Luxembourg ISP (4182 mi)	108,000,000,000	100 ms	euodns.com	Ping
Time for light to travel to the moon (average)	1,348,333,333,333	1 s	The moon	<a href="http://www.wolframalpha.com/input/?i=distance+to+the+moon">http://www.wolframalpha.com/input/?i=distance+to+the+moon</a>

\* Based on Dhrystone, single core only, average time per instruction

\*\* Based on sequential throughput, average time per block

# Performance of Demand Paging

## Stages in Demand Paging:

- **Trap** to the operating system (us)
- Save the user registers and process state (ns)
- Check that the page reference was legal and determine the location of the page on the disk (ns)
- Issue a read from the disk to a free frame:
  - Wait in a queue for this device until the read request is serviced (us ms)
  - Wait for the device seek and/or latency time
  - Begin the transfer of the page to a free frame
- While waiting, allocate the CPU to some other process
- Receive an interrupt from the disk I/O subsystem (I/O completed)
- Save the registers and process state for the other process (ns)
- Correct the page table and other tables to show page is now in memory (ns)
- Wait for the CPU to be allocated to this process again (?)
- Restore the user registers, process state, and new page table, and then resume the interrupted instruction (us)

# Performance of Demand Paging (Cont.)

- Page Fault Rate  $0 \leq p \leq 1$ 
  - if  $p = 0$  no page faults
  - if  $p = 1$ , every reference is a fault

- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & \quad + \text{swap page out} \\ & \quad + \text{swap page in} \\ & \quad + \text{restart overhead} \\ & \quad ) \end{aligned}$$

# Demand Paging Example

- Memory access time = 60 nanoseconds
- Average page-fault service time = 10 milliseconds
- $$\begin{aligned} \text{EAT} &= 60(1 - p) + (10 \text{ milliseconds})(p) \\ &= 60(1 - p) + 10,000,000p \\ &= 60 + 9,999,940p \end{aligned}$$
- If one access out of 1,000 causes a page fault, then  $\text{EAT} \approx 10$  microseconds!
  - This is a slowdown factor of 166!!
- If want performance degradation < 10%:
$$\begin{aligned} (110\%)(60) &> 60 + 9,999,940p \\ 6 &> 9,999,940p \\ p &< .0000006 \end{aligned}$$
  - Less than one page fault in every 1.6 million memory accesses!
  - Luckily, we can do this: just don't have programs that do regular memory access patterns larger than the size of memory.
  - Alternately, if you \*do\* try to use more RAM than you have, it works, it's just very, very slow.

# What Happens if There is no Free Frame?

- **Page replacement** – find some page in memory, but not really in use, page it out
  - Algorithm?
  - Want an algorithm which will result in minimum number of page faults
- Optimization: use **“dirty” bit** in page table to track pages modified since loading; only modified pages are written to disk



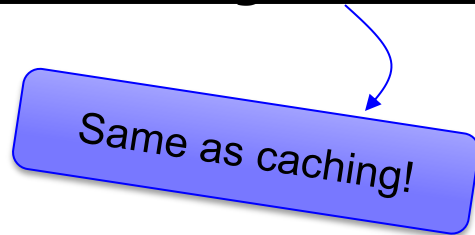
# Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a **victim frame**
  - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

# Page Replacement Algorithms

- **Frame-allocation algorithm determines**
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
  - ***This decision is just like choosing the caching replacement algorithm!***



Same as caching!

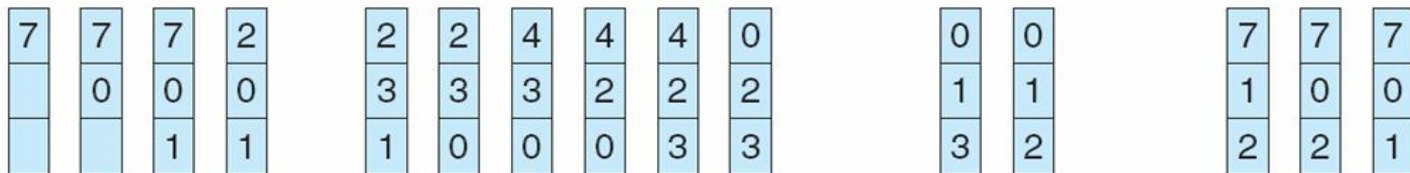
# First-In-First-Out (FIFO) Algorithm

Same as caching!

- Reference string:  
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames

reference string

7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1



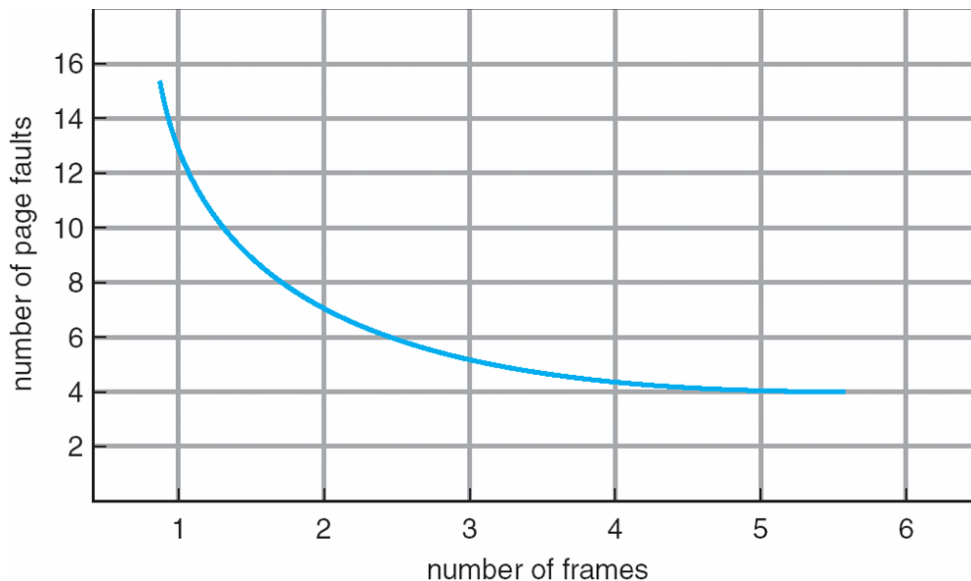
page frames

# FIFO Illustrating Belady's Anomaly

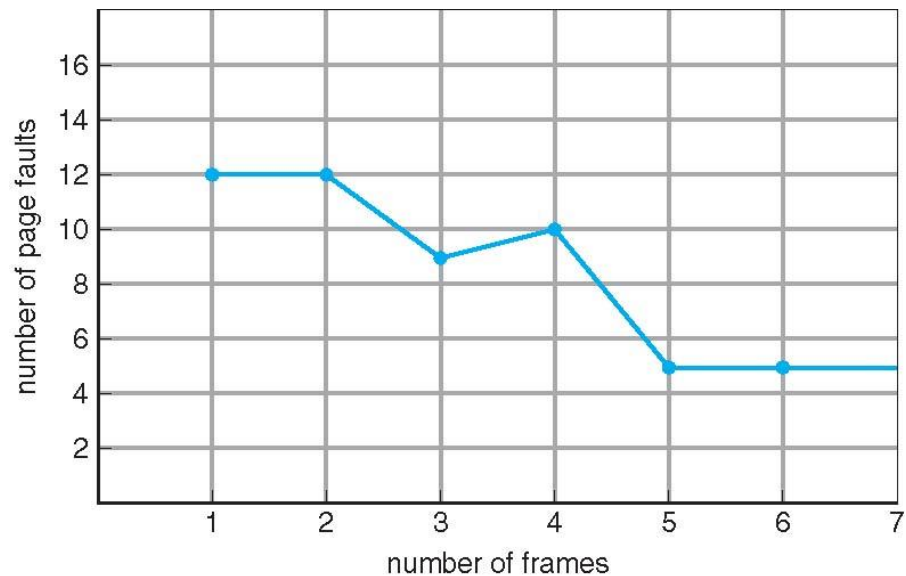
Same as caching!

- What if we add more frames?
- Adding more frames can cause more page faults! This is **Belady's Anomaly**.
  - Solution: use a better algorithm...

Expectation  
(behavior of optimal algorithm)



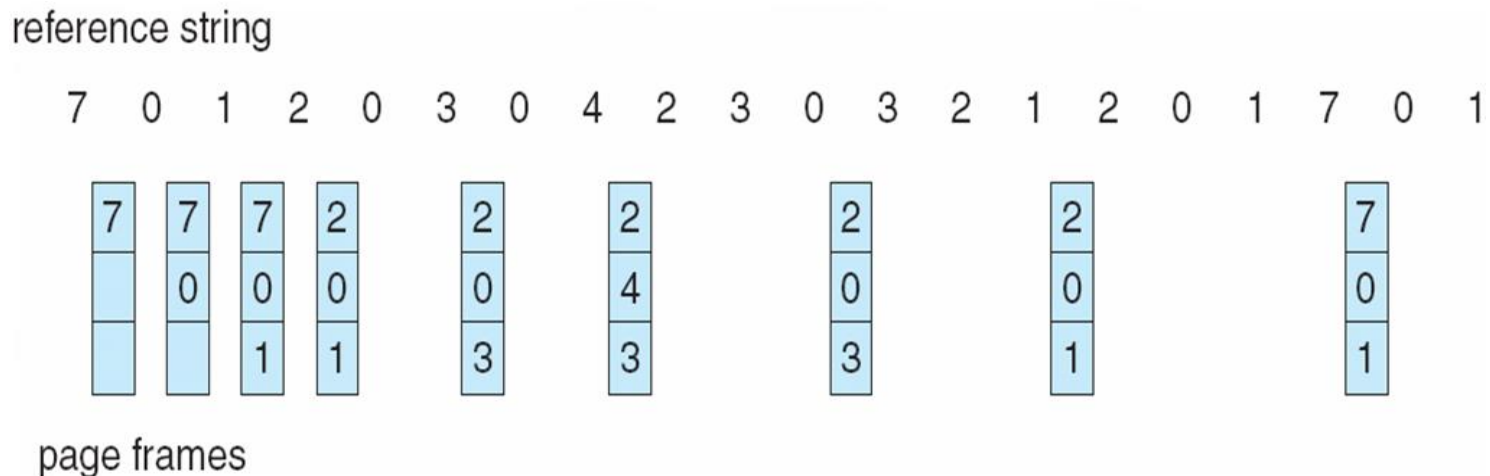
Reality  
(behavior of FIFO algorithm)



# Optimal Algorithm

Same as caching!

- Replace page that will not be used for longest period of time
- How do you know this?
  - Read the future using magic/witchcraft (cheat)
  - Used for measuring how well your algorithm performs



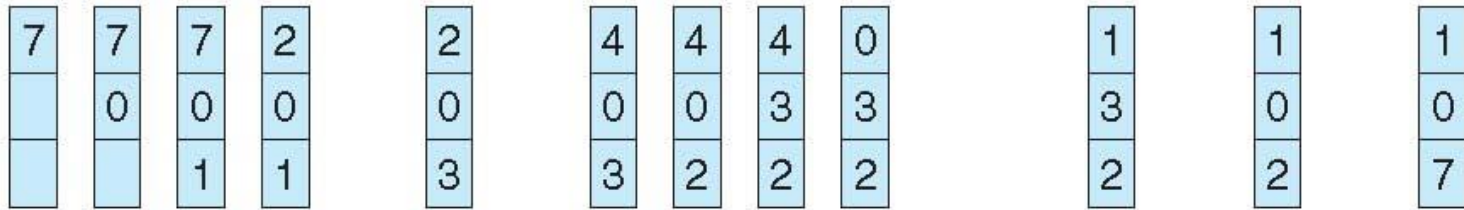
# Least Recently Used (LRU) Algorithm

Same as caching!

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

# LRU Algorithm (Cont.)

Same as caching!

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock\* into the counter
  - When a page needs to be changed, find the smallest counter value
- Stack implementation
  - Keep a stack of page numbers in a double link form
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of stack algorithms that don't have Belady's Anomaly

\* "Clock" can just be number of cycles since boot, etc.

# LRU Approximation Algorithms

Same as caching!

- LRU needs special hardware and still slow
- **Reference bit** for each page
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules






# Counting Algorithms

Same as caching!

- Keep a counter of the number of references that have been made to each page
  - Not common
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

# Page Replacement Algorithms Summary

Same as caching!

- **FIFO**: Too stupid 
- **OPT**: Too impossible 
- **LRU**: Great, but can be expensive 
- **Reference bit, second-chance**:  
Tradeoff between LRU and FIFO
- **LFU/MFU**:  
Seldom-used counter-based algorithms

# Page-Buffering Algorithms

- Keep a **pool of free frames**, always
  - Then frame available when needed, not found at fault time
  - Instead of fault→evict→load, do fault→load→queue for eviction
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store is idle, write pages there and set to non-dirty
  - Then the “evict” becomes “drop” instead of “store”
- Possibly, keep free frame contents intact
  - If referenced again before evicted, no need to reload it from disk
  - Reduces penalty if wrong victim frame selected

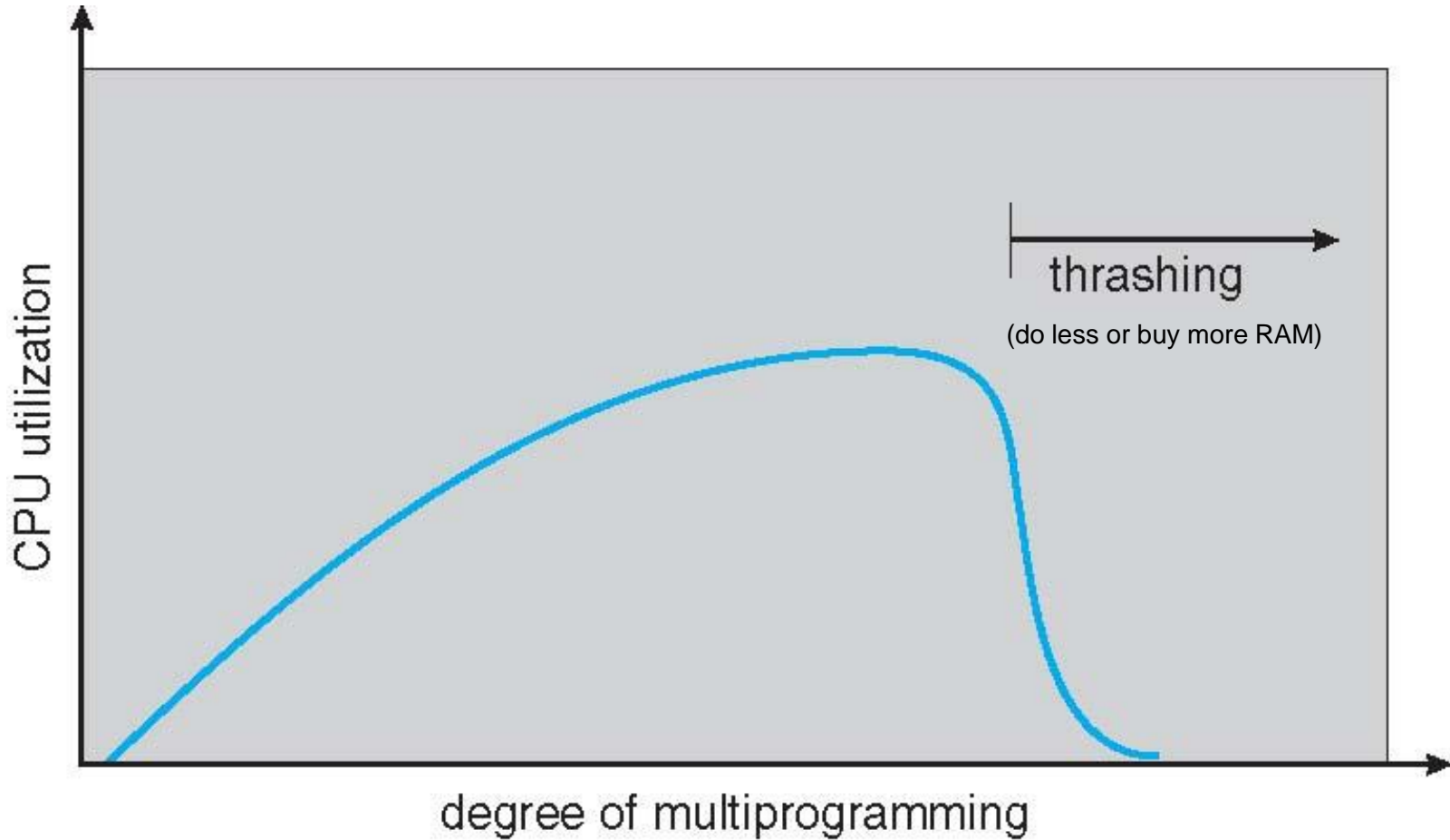
# Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

# Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system
- Thrashing  $\equiv$  a process is busy swapping pages in and out

# Thrashing (Cont.)



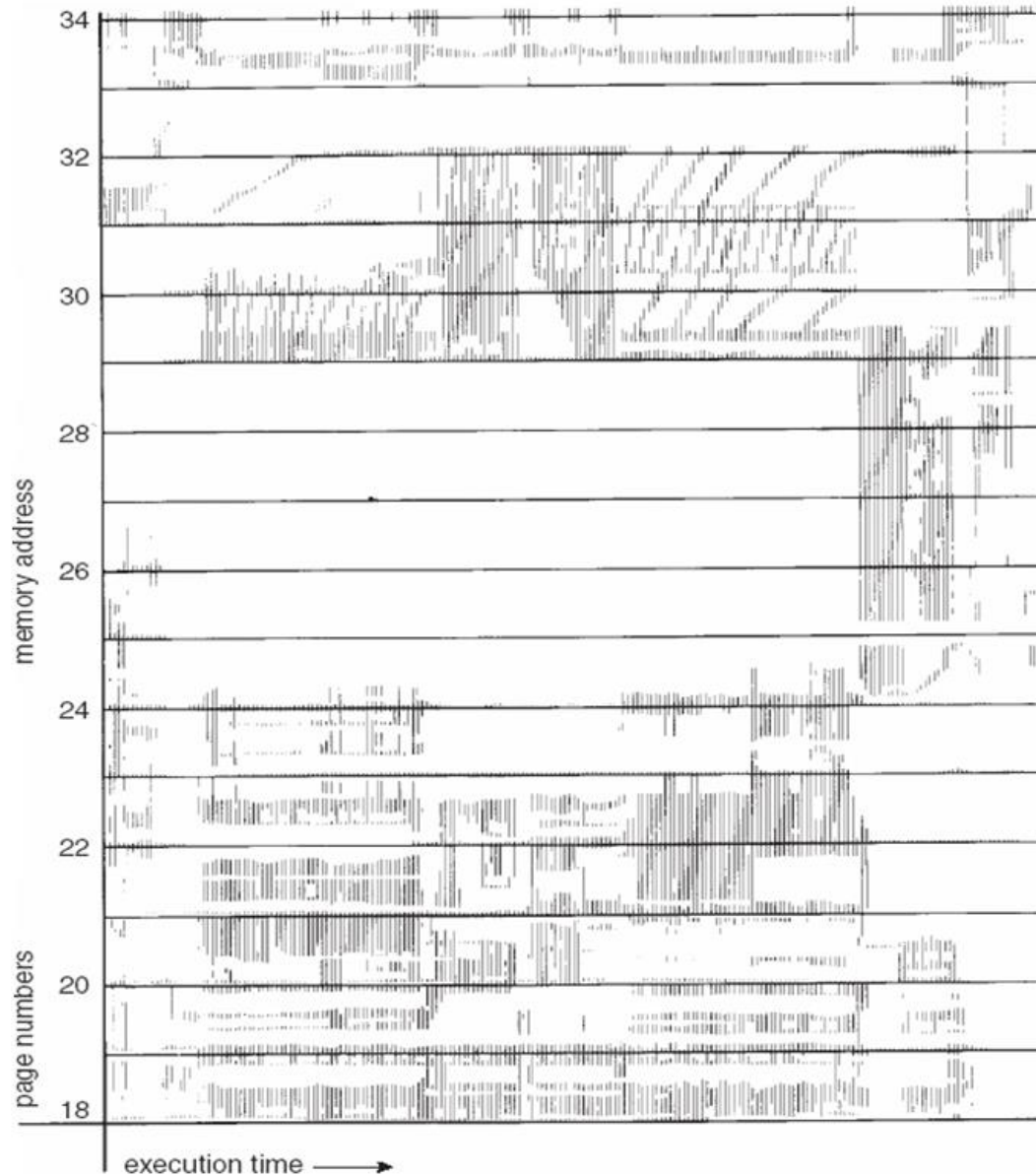
# Demand Paging and Thrashing

- Why does demand paging work?

## **Locality model**

- Process migrates from one locality to another
  - Localities may overlap
- 
- Why does thrashing occur?  
 $\Sigma$  size of locality > total memory size
    - Limit effects by using local or priority page replacement

# Locality In A Memory-Reference Pattern



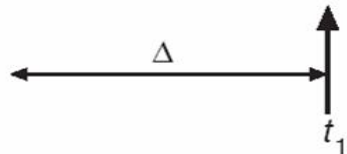


# Working-set model

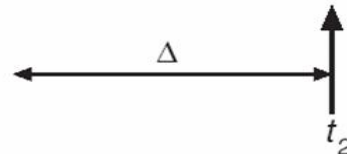
- $\Delta \equiv$  working-set window  $\equiv$  a fixed number of page references  
Example: 10,000 instructions
- $WSS_i$  (working set of Process  $P_i$ ) =  
total number of pages referenced in the most recent  $\Delta$  (varies in time)
  - if  $\Delta$  too small will not encompass entire locality
  - if  $\Delta$  too large will encompass several localities
  - if  $\Delta = \infty \Rightarrow$  will encompass entire program
- $D = \sum WSS_i \equiv$  total demand frames
  - Approximation of locality
- if  $D > m \Rightarrow$  Thrashing
- Policy if  $D > m$ , then suspend or swap out one of the processes

page reference table

... 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



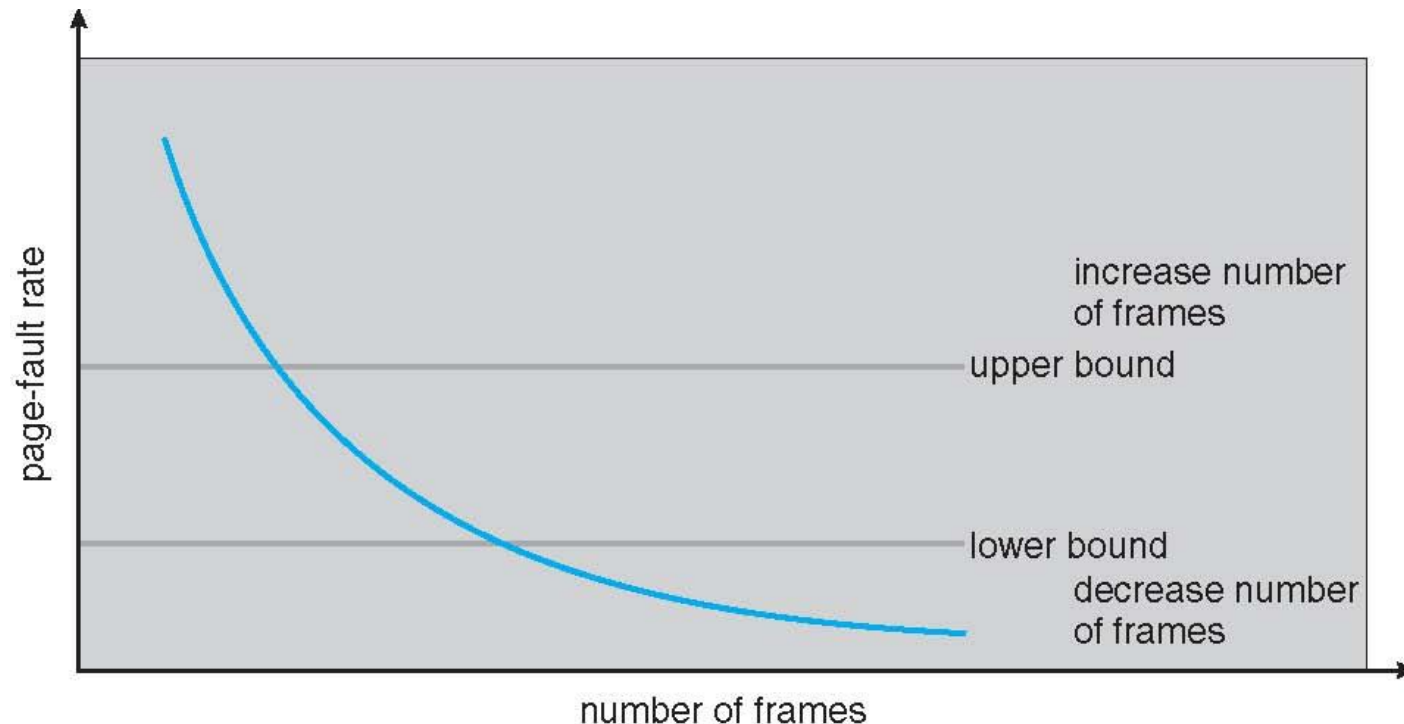
$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



$$WS(t_2) = \{3, 4\}$$

# Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency** rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



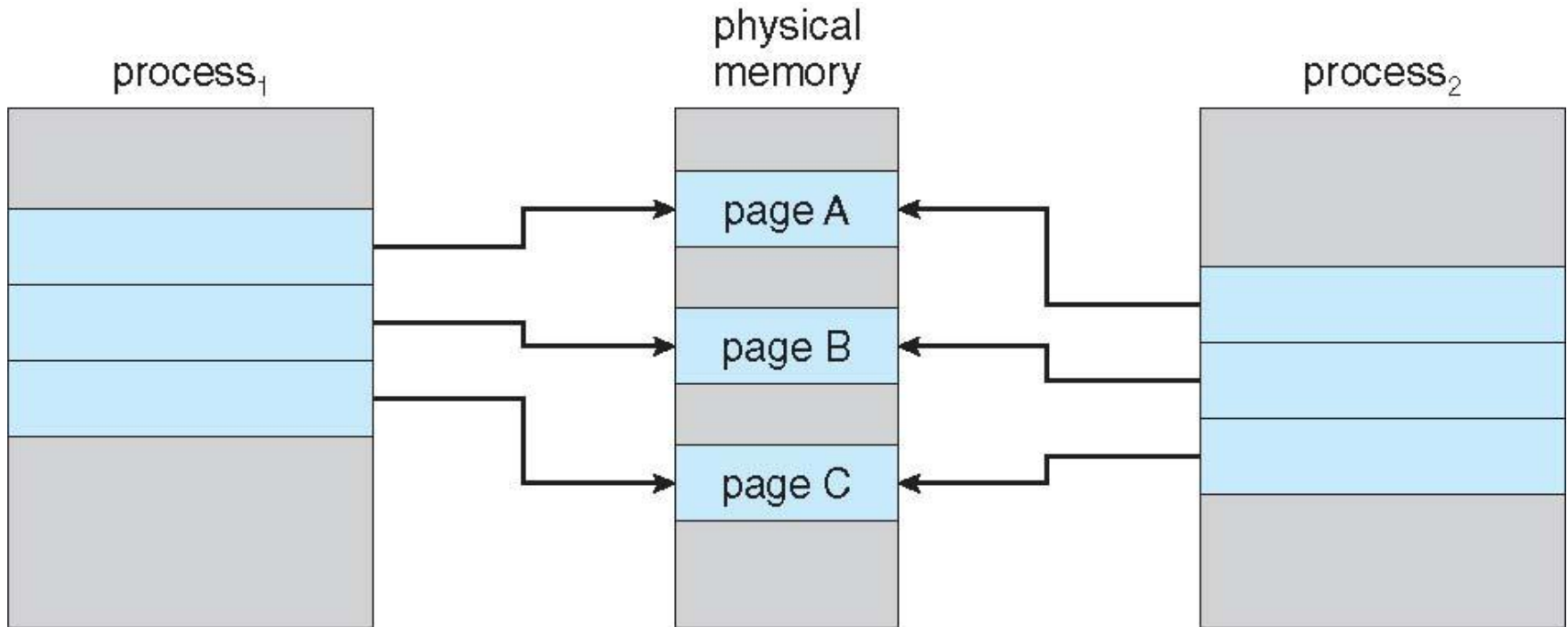
# Copy-on-Write

- Side-note: a useful trick made possible by paging:

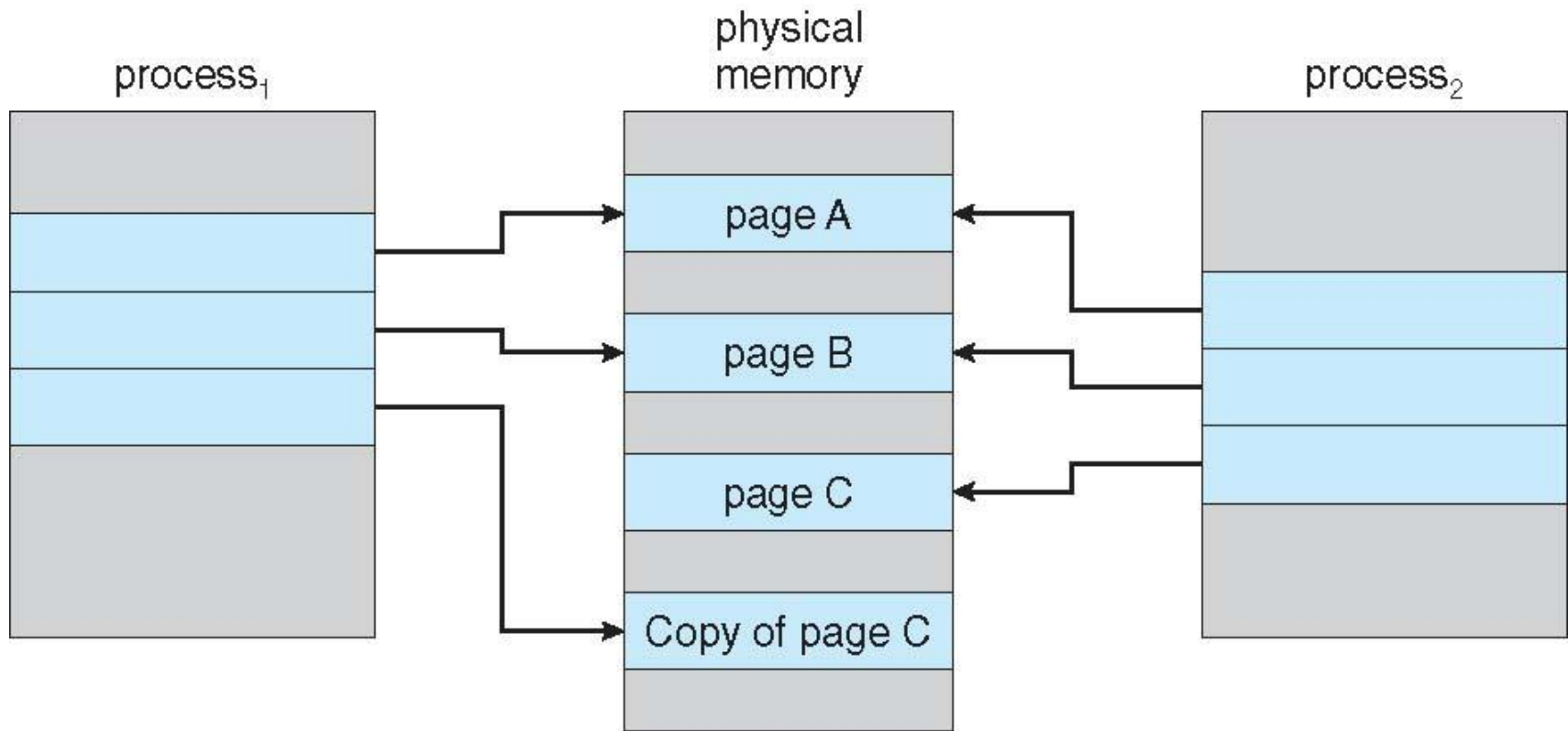
## Copy-on-Write (COW)

- Allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
  - Allows more efficient process creation
- In general, free pages are allocated from a pool of zero-fill-on-demand pages
  - Why zero-out a page before allocating it?

# Before Process 1 Modifies Page C



# After Process 1 Modifies Page C



# IV. OS EXAMPLES

# Windows XP

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **maximum**
- When free memory falls below a threshold, automatic working set trimming removes pages from processes that have pages in excess of their working set minimum, thus restoring the amount of free memory

# Solaris

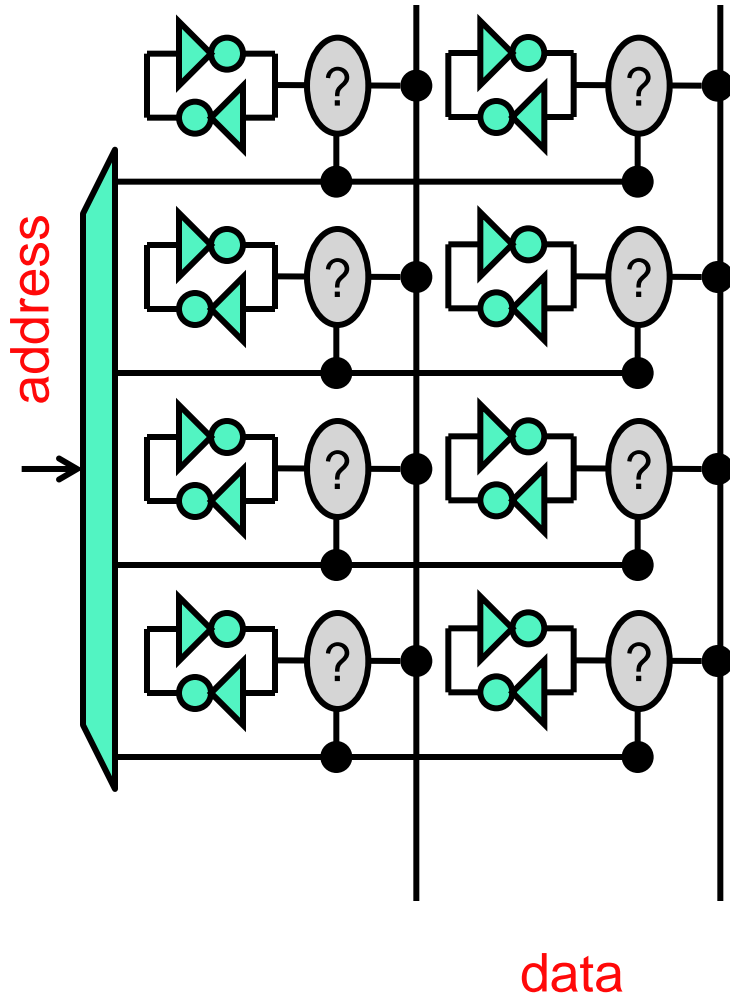
- Maintains a list of free pages to assign faulting processes
- Paging is performed by *pageout* process
  - Scans pages using modified clock algorithm
- Parameters:
  - *Lotsfree* – threshold parameter (amount of free memory) to begin paging
  - *Desfree* – threshold parameter to increasing paging
  - *Minfree* – threshold parameter to being swapping
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
  - Pageout is called more frequently depending upon the amount of free memory available
- Priority paging gives priority to process code pages





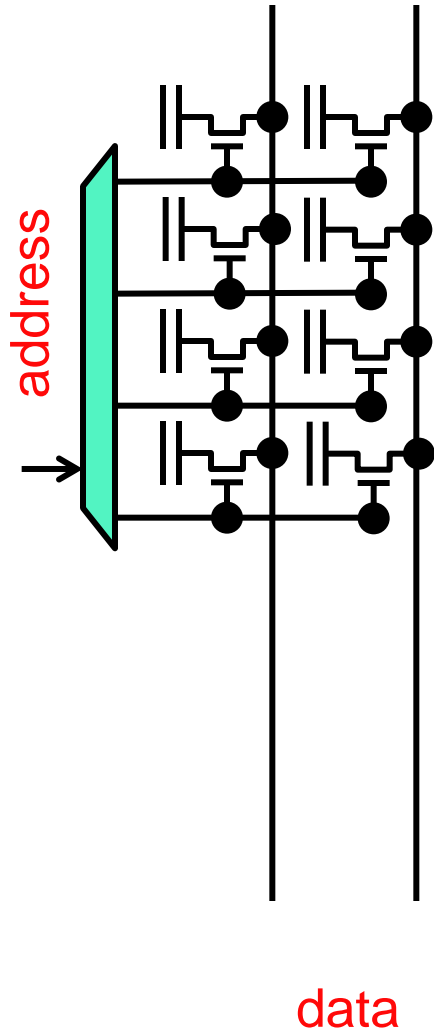
# V. BUT WHAT'S RAM MADE OF?

# Remember Static RAM (SRAM)?



- **SRAM**: static RAM
  - Bits as cross-coupled inverters
  - Four transistors per bit
  - More transistors for ports
- **"Static"** means
  - Inverters connected to power/ground
  - Bits naturally/continuously "refreshed"
  - Bit values never decay
- Designed for speed

# Dynamic RAM (DRAM)



- **DRAM**: dynamic RAM
  - Bits as capacitors (if charge, bit=1)
  - “Pass transistors” as ports
  - One transistor per bit/port
- **“Dynamic”** means
  - Capacitors not connected to power/gnd
  - Stored charge decays over time
  - Must be explicitly refreshed
- Designed for density
  - Moore’s Law ...

# Moore's Law (DRAM chip capacity)

Year	Capacity	\$/MB	Access time
1980	64Kb	\$1500	250ns
1988	4Mb	\$50	120ns
1996	64Mb	\$10	60ns
2004	1Gb	\$0.5	35ns
2008	2Gb	~\$0.15	20ns
2013	8Gb	~\$0	<10ns

- Commodity DRAM parameters
  - 16X increase in capacity every 8 years = 2X every 2 years
    - Not quite 2X every 18 months (Moore's Law) but still close

# Access Time and Cycle Time

- DRAM access much slower than SRAM
  - More bits → longer wires
  - SRAM access latency: 2–3ns
  - DRAM access latency: 20-35ns
- DRAM cycle time also longer than access time
  - **Cycle time**: time between start of consecutive accesses
  - SRAM: cycle time = access time
    - Begin second access as soon as first access finishes
  - DRAM: cycle time = 2 \* access time
    - Why? Can't begin new access while DRAM is refreshing row

# Memory Access and Clock Frequency

- Computer's advertised **clock frequency** applies to CPU and caches
  - DRAM connects to processor chip via memory "bus"
  - Memory bus has its own clock, typically much slower
- Another reason why processor clock frequency isn't perfect performance metric
  - Clock frequency increases don't reduce memory or bus latency
  - May make misses come out faster
    - At some point memory bandwidth may become a **bottleneck**
    - Further increases in (core) clock speed won't help at all

# DRAM Packaging

- DIMM = dual inline memory module
  - E.g., 8 DRAM chips, each chip is 4 or 8 bits wide



# DRAM: A Vast Topic

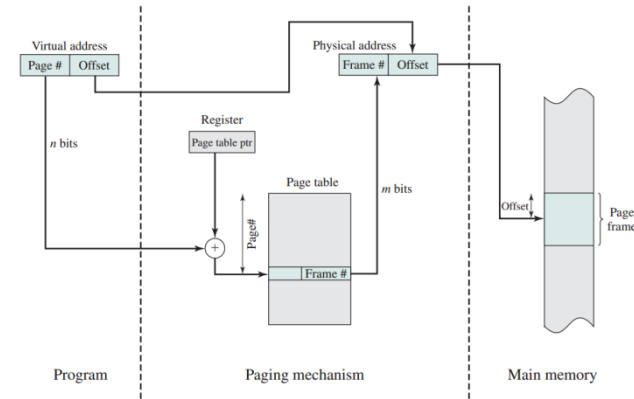
- Many flavors of DRAMs
  - DDR3 SDRAM, RDRAM, etc.
- Many ways to package them
  - SIMM, DIMM, FB-DIMM, etc.
- Many different parameters to characterize their timing
  - $t_{RC}$ ,  $t_{RAC}$ ,  $t_{RCD}$ ,  $t_{RAS}$ , etc.
- Many ways of using row buffer for “caching”
- Etc.
- There’s at least one whole textbook on this topic!
  - And it has  $\sim 1K$  pages
- We could, but won’t, spend rest of semester on DRAM



# Virtual memory summary



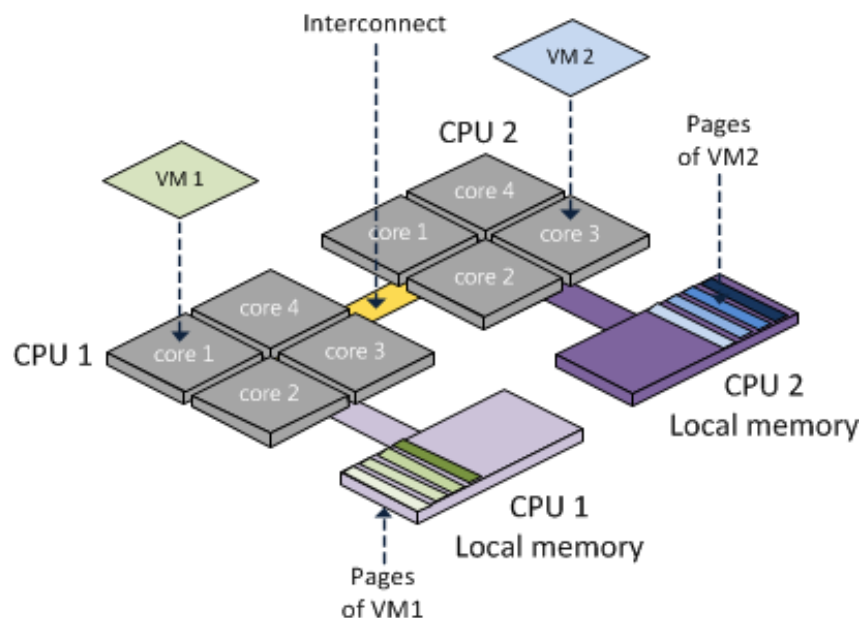
- Address translation via **page table**
  - Page table turns VPN to PPN (noting the valid bit)
- Page is marked 'i'? **Page fault.**
  - If OS has stored page on disk, load and resume
  - If not, this is invalid access, kill app (seg fault)
- Governing policies:
  - Keep a certain **number of frames loaded** per app
  - Kick out frames based on a **replacement algorithm** (like LRU, etc.)
- Looking up page table in memory too slow, so cache it:
  - The **Translation Buffer (TB)** is a hardware cache for the page table
  - When applied at the same time as caching (as is common), it's called a **Translation Lookaside Buffer (TLB)**.
- **Working set size** tells you how many pages you need over a time window.
- **DRAM** is slower than SRAM, but denser. Needs constant refreshing of data.



# **OTHER CONSIDERATIONS (TIME PERMITTING)**

# Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are NUMA – speed of access to memory varies
  - E.g. multi-socket systems, even some single-socket multi-core systems
- Want to allocate memory “close to” a process’s CPU
  - Must modifying the scheduler to schedule the thread on a core “near” its memory
  - If an app needs more memory than local memory can provide, must use some “remote” memory.
  - The problem of “local” allocation vs. “remote” is basically another layer of the memory hierarchy (and is solved the same way).



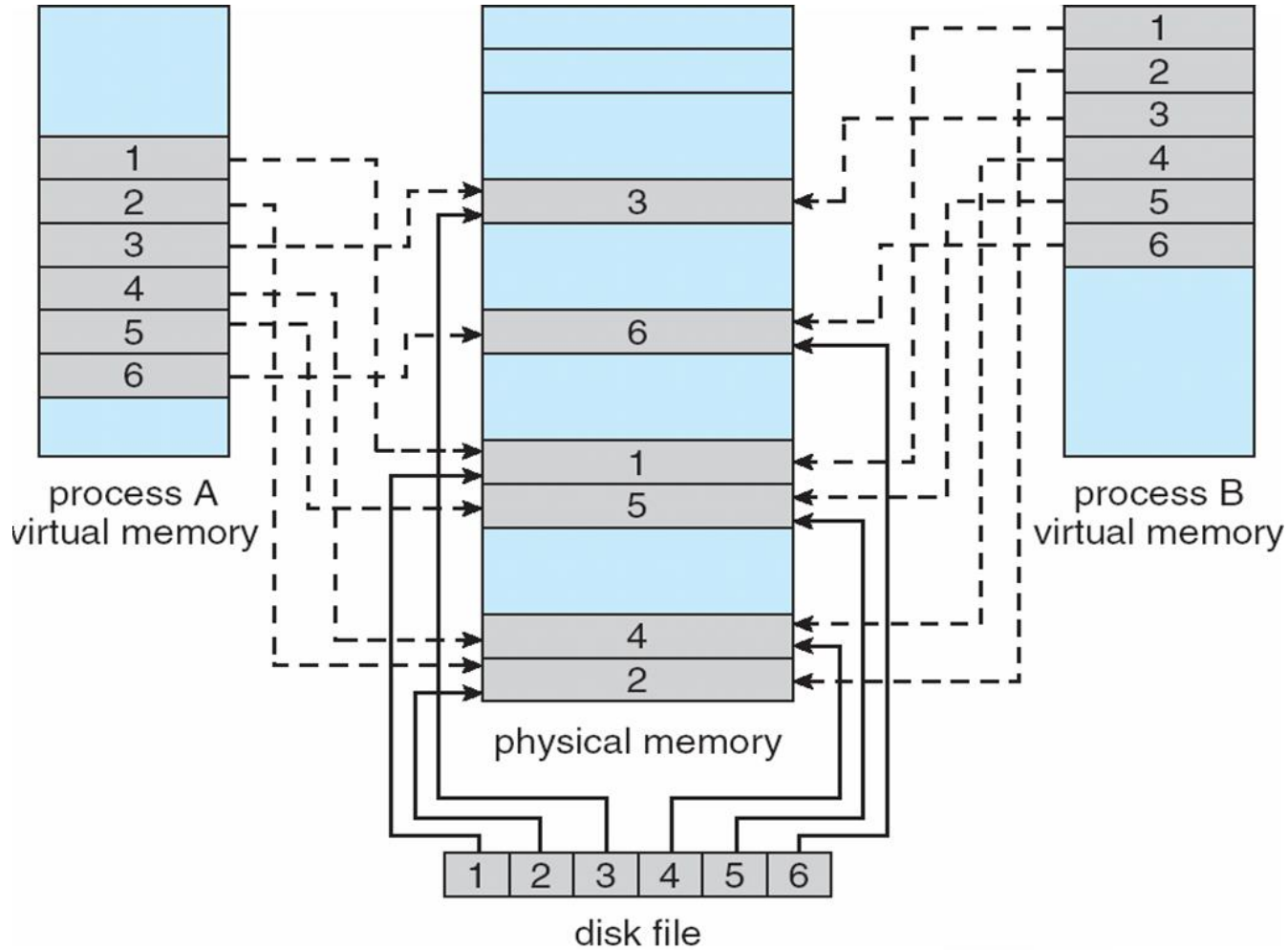
# Memory-Mapped Files

- Tell the OS: “treat this region of memory as if it’s data that should be swapped in from THIS swapfile”. `mmap()` on \*nix.
- File is accessed via demand paging (for initial I/O) or simple memory access (for subsequent I/O)
- Benefits:
  - Can be faster (depending on I/O pattern)
  - Can be simpler (depending on problem)
  - Allows several processes to map the same file, allowing the pages in memory to be shared
- When does written data make it to disk?
  - Periodically and / or at file `close()` time

# Memory-Mapped File Technique for all I/O

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
  - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway
  - But map file into kernel address space
  - Process still does `read()` and `write()`
    - Copies data to and from kernel space and user space
  - Uses efficient memory management subsystem
    - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages

# Memory Mapped Files



# Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge, e.g., databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- OS can give direct access to the disk, getting out of the way of the applications
  - **Raw disk mode**
  - Bypasses buffering, locking, etc

# Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - I/O overhead
  - Number of page faults
  - TLB size and effectiveness
- Always power of 2, usually 4kB~4MB
- Modern x86 supports 4kB “normal” pages and 2MB “huge” pages at the same time (the latter requiring special consideration to use)



# Any Questions?

