ECE/CS 250
Computer Architecture

Fall 2022

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 <u>paging</u>)

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

How I'm going to cover this

We'll start with just **paging**, then add **swapping** later

II. MECHANICS OF <u>PAGING</u>

Paging is how you divvy up physical memory among processes

Demand Paging

Page

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

How to think about pages

- Hey, remember cache blocks? Remember how they were **fixed-size**, **contiguous**, **and aligned**?
- Pages work the same way
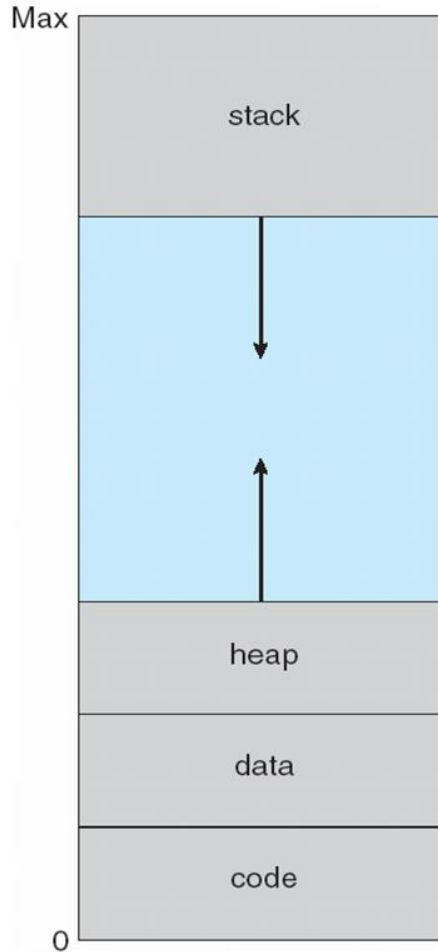
- Quick intuition reminder – suppose pages are 8 bytes:

Address	Address / 8 Address >> 3	Address % 8 Address & 7
0	0	0
1	0	1
2	0	2
3	0	3
4	0	4
5	0	5
6	0	6
7	0	7
8	1	0
9	1	1
10	1	2
11	1	3
12	1	4
13	1	5
14	1	6
15	1	7
16	2	0
17	2	1
18	2	2
19	2	3
20	2	4
21	2	5
22	2	6

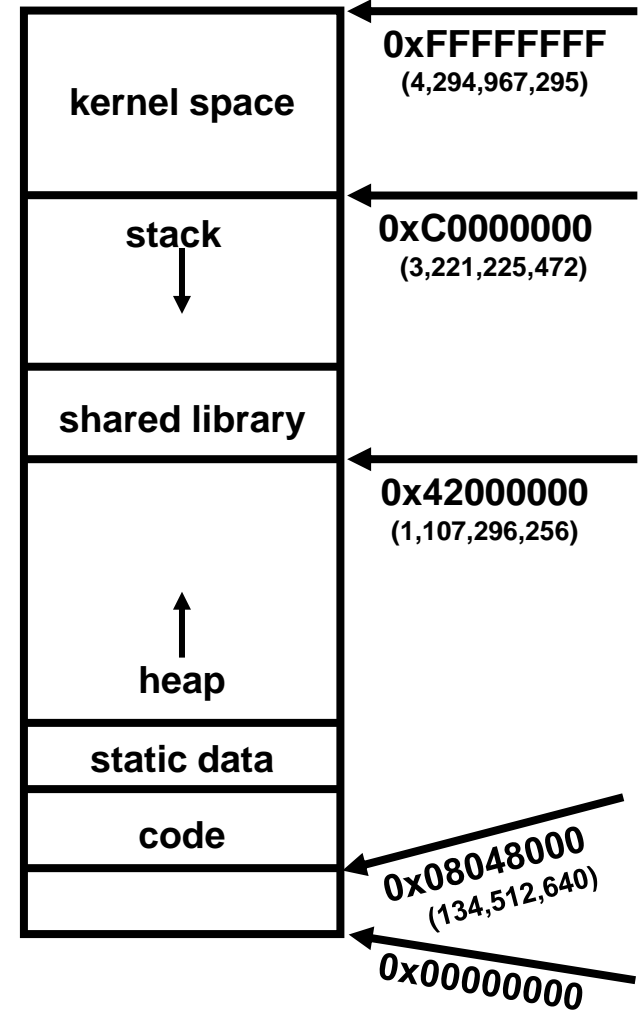- Dividing by the page size == taking the high bits in binary

  Therefore,

- Splitting an address into higher and lower bits can give you **page number** and **page offset** just like in caching

- Analogy: if given the number of minutes since midnight, you can divide & mod by 60 to get the time:
  - If it's 150 minutes since midnight, that's
    150/60 = **2** hours and
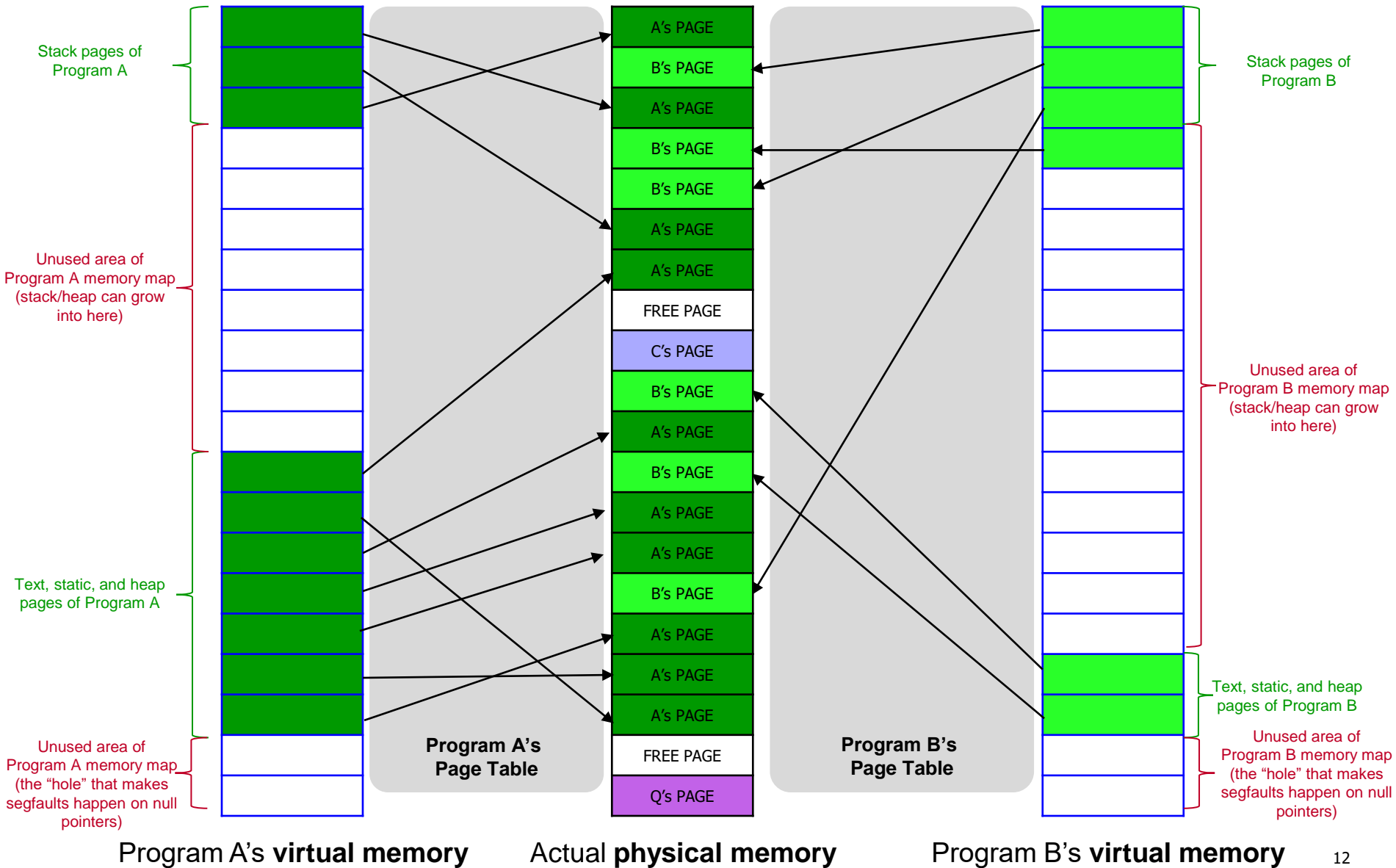   150%60 = **30** minutes
  - **2**:**30**

Virtual Address Space

(Fluffy academic version)

(Real 32-bit x86 view)

Every program has its own page mapping

Stack pages of Program A

Unused area of Program A memory map (stack/heap can grow into here)

Text, static, and heap pages of Program A

Unused area of Program A memory map (the "hole" that makes segfaults happen on null pointers)

Program A's Page Table

A's PAGE
B's PAGE
A's PAGE
B's PAGE
B's PAGE
A's PAGE
A's PAGE
FREE PAGE
C's PAGE
B's PAGE
A's PAGE
B's PAGE
A's PAGE
A's PAGE
B's PAGE
A's PAGE
A's PAGE
A's PAGE
FREE PAGE
Q's PAGE

Program B's Page Table

Stack pages of Program B

Unused area of Program B memory map (stack/heap can grow into here)

Text, static, and heap pages of Program B

Unused area of Program B memory map (the "hole" that makes segfaults happen on null pointers)

Program A's **virtual memory**   Actual **physical memory**   Program B's **virtual memory**
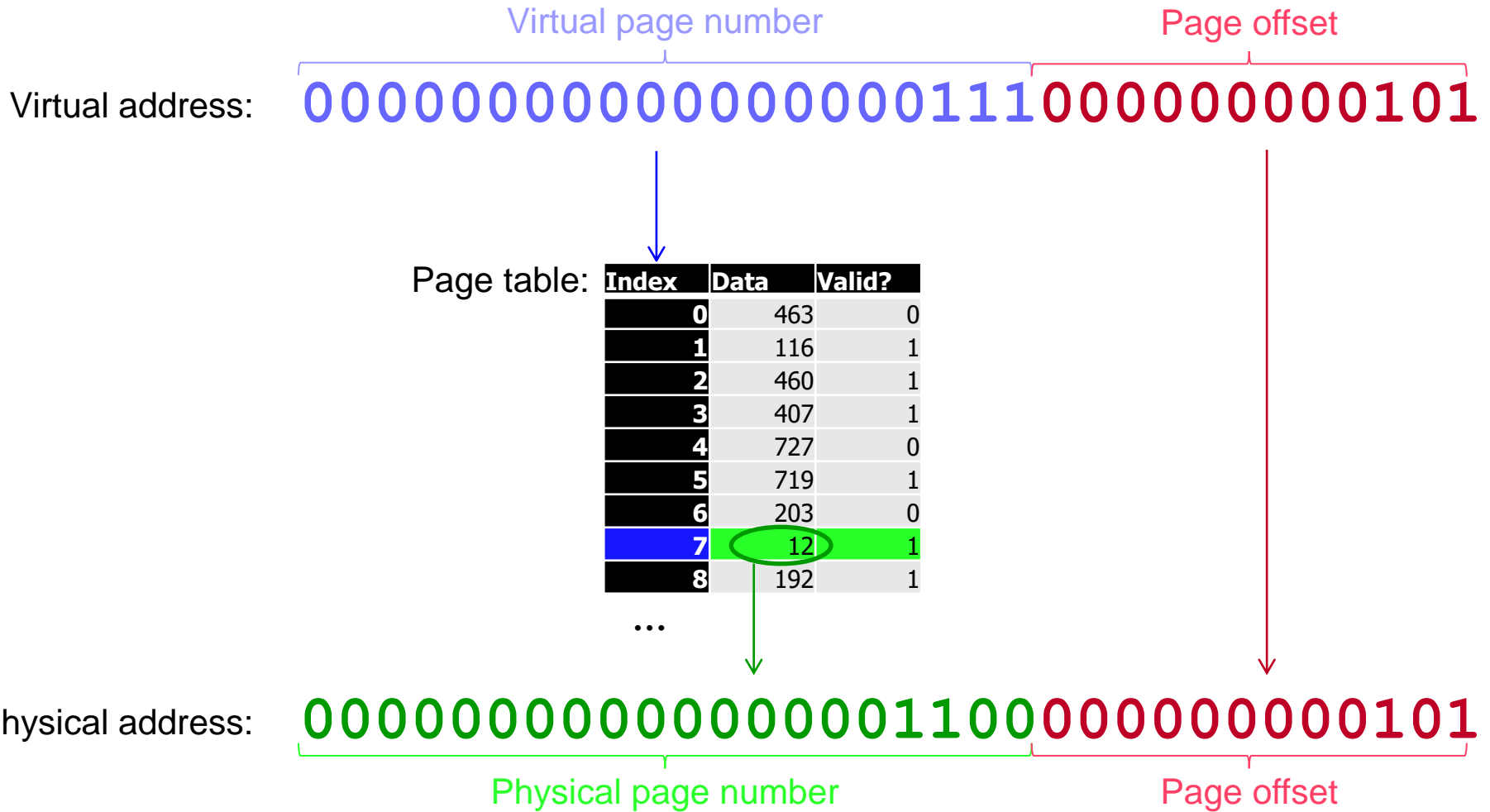
12

- The page table is a data structure that:
  - Takes a **virtual page number (VPN)**
  - Looks up the corresponding **physical page number (PPN)** (if any)
    - We note missing mappings with a **valid bit**

- Simplest model: array with a spot for each virtual page
  - If word size is 32 and page size is 2^{12} (4kB),
    then you have $2^{32}/2^{12} = 2^{20} = 1048576$ pages

VPN (index)	PPN	Valid?
0	~~5675684~~	0
1	1501	1
2	12	1
...		
1048574	815	1
1048575	~~4574365~~	0

- Other data structures are possible
  (you'll think about that on HW5)

Every process gets its <u>own</u> page table!!

Address translation
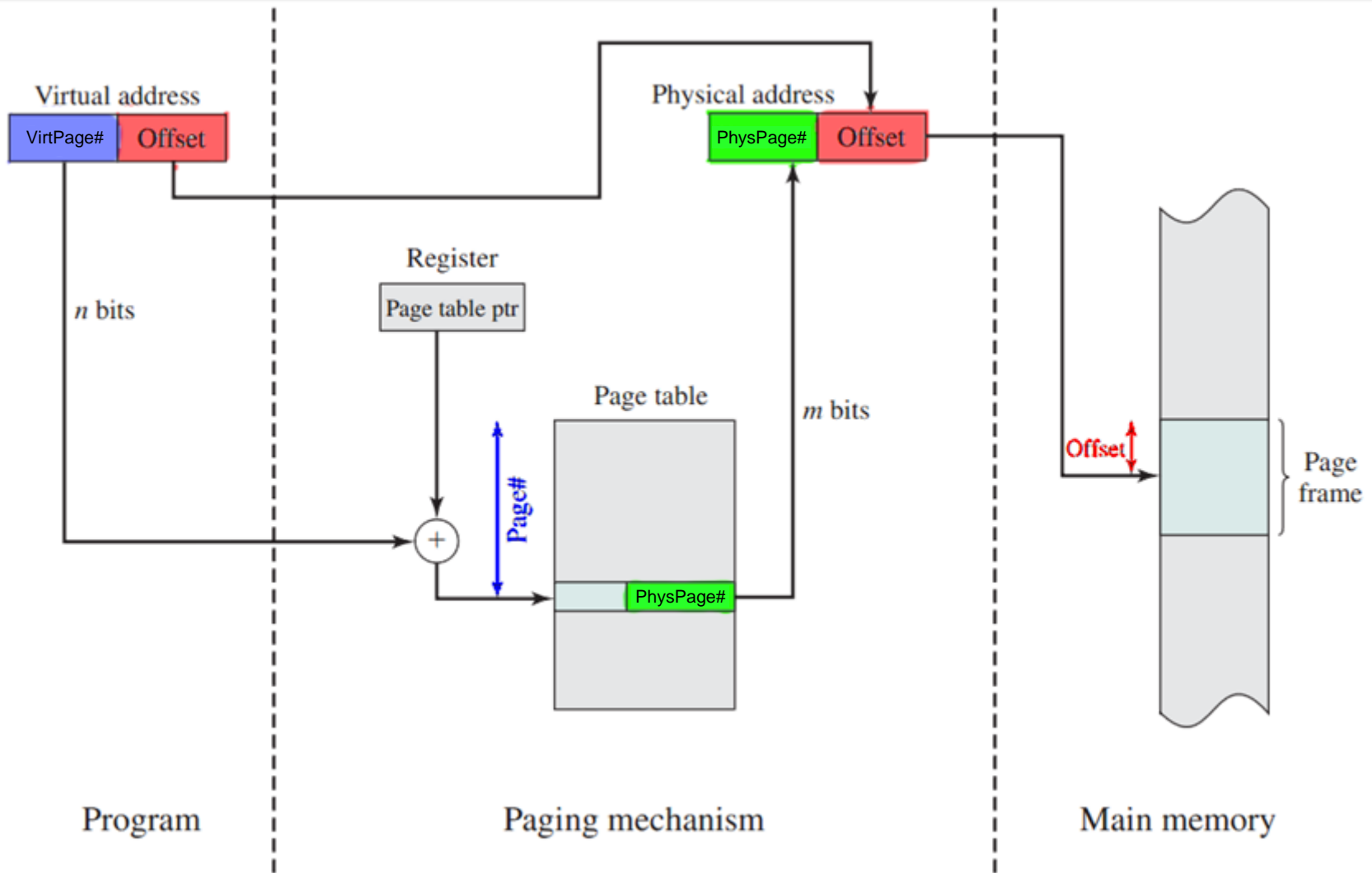
Virtual page number   Page offset

Virtual address: 00000000000000000111000000000101

Page table:

Index	Data	Valid?
0	463	0
1	116	1
2	460	1
3	407	1
4	727	0
5	719	1
6	203	0
7	12	1
8	192	1

...

Physical address: 00000000000000001100000000000101

Physical page number   Page offset

Address translation

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne

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

```c
struct page_table_entry {
  char valid; // one bit
  int phys_page;
}
struct page_table_entry page_table[1048576];

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

    if (!page_table[vpn].valid) DO_SEGFAULT_EXCEPTION();
    int ppn = page_table[vpn].phys_page; // table lookup

    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

- Simplified, and ignoring the valid check:

```
int P[1048576]; // converts vpn to ppn

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

Does this now make sense?

Stack pages of Program A

Unused area of Program A memory map (stack/heap can grow into here)

Text, static, and heap pages of Program A

Unused area of Program A memory map (the "hole" that makes segfaults happen on null pointers)

| A's PAGE |
| B's PAGE |
| A's PAGE |
| B's PAGE |
| B's PAGE |
| A's PAGE |
| A's PAGE |
| FREE PAGE |
| C's PAGE |
| B's PAGE |
| A's PAGE |
| B's PAGE |
| A's PAGE |
| A's PAGE |
| B's PAGE |
| A's PAGE |
| A's PAGE |
| A's PAGE |
| FREE PAGE |
| Q's PAGE |

Program A's Page Table

Program B's Page Table

Stack pages of Program B

Unused area of Program B memory map (stack/heap can grow into here)

Text, static, and heap pages of Program B

Unused area of Program B memory map (the "hole" that makes segfaults happen on null pointers)

Program A's **virtual memory**   Actual **physical memory**   Program B's **virtual memory**

18

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
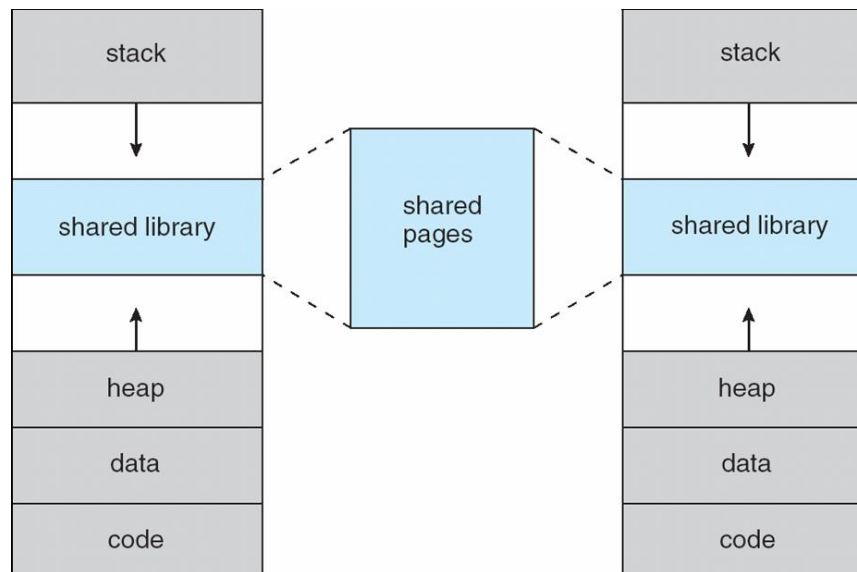
Address Translation Mechanics (1)

- 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 lowest cache miss or always? How would program know?
  - This sucks. Let's never do it.

Address Translation Mechanics (2)

- Option II: **operating system (OS)** translates for process
  - Page table resides in OS virtual address space
  - OS is called to do translation as needed
  - + User-level processes cannot view/modify their own tables
  - + User-level processes need not know about physical addresses
  - – Have to ask OS for *every* memory load??????? NOOOOOOO!
    - ^ Kills performance

- But what about the concept of **caching**?
  - We can improve things by only asking OS if the lowest cache misses
  - That helps, but can we do better?
  - If having a **data cache** helped speed up **data access**,
    what if we have a **page table cache** to speed **page table access**?

"Translation Buffer"
It's a cache for your page table!™

Translation Buffer
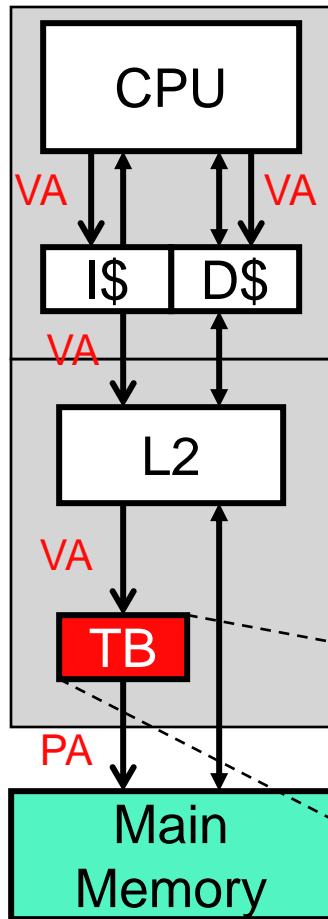
- Functionality problem? Add indirection!
- Performance problem? Add cache!

- 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

CPU

VA   VA

I$ D$

VA

L2

VA

TB

PA

Main Memory

"tag"   "data"

VPN	PPN
VPN	PPN
VPN	PPN
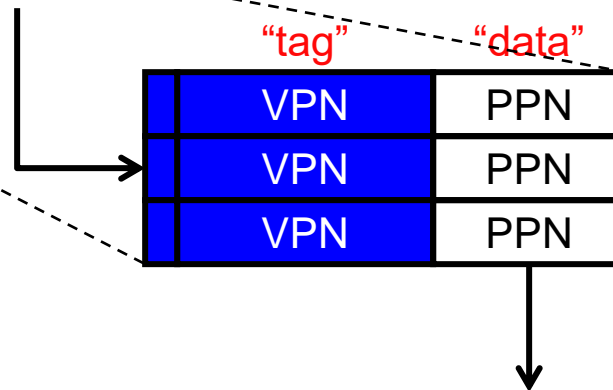
TB Misses

- **TB miss:** requested page table entry 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

Virtual Caches
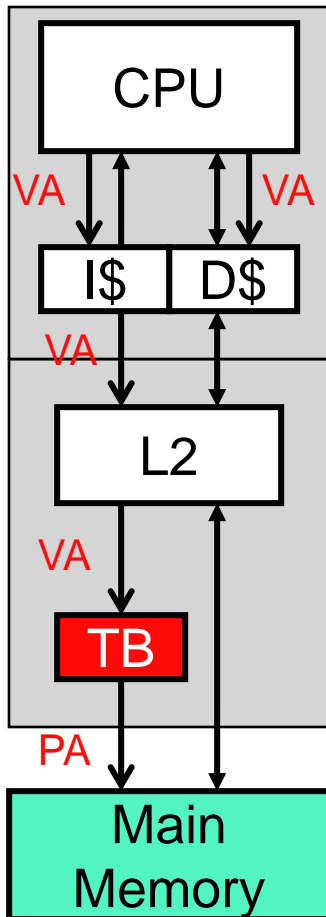
- Memory hierarchy so far: **virtual caches**
  - Indexed and tagged by VAs
  - Translate to PAs only to access memory
  - + Fast: avoids translation latency in common case

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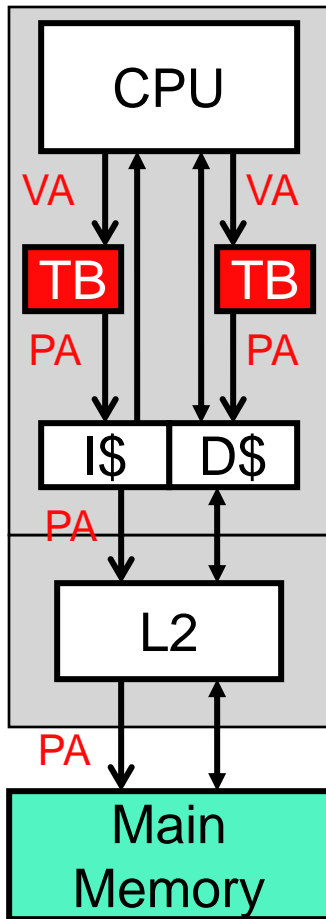
- Alternatively: **physical caches**
  - Indexed and tagged by PAs
  - Translate to PA at the outset
  - + No need to flush caches on process switches
    - Processes do not share PAs
  - + Cached inter-process communication works
    - Single copy indexed by PA
  - – Slow: adds 1 cycle to t_{hit}
 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

Virtual Physical Caches
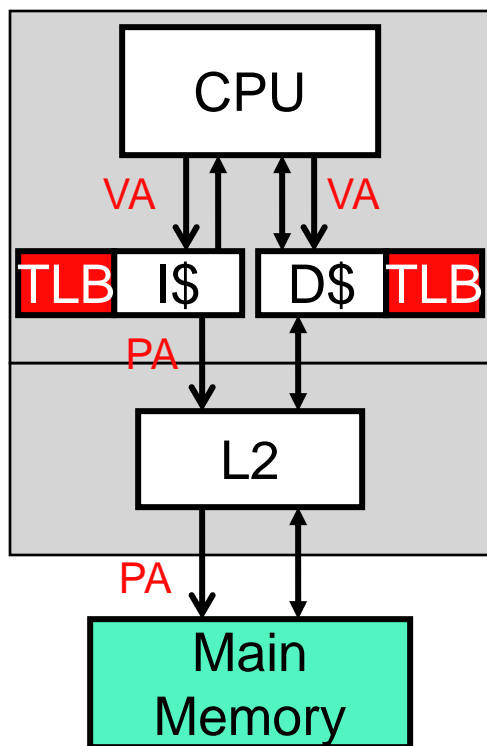
Compromise: **virtual-physical caches**

- 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

TLB

cache

TLB hit/miss

cache hit/miss

| 0 |
| 1 |
| 2 |
| 1022 |
| 1023 |

[31:12]   [11:2]   1:0

VPN [31:16]   POFS[15:0]

<<

address   data

Cache Size And Page Size

[31:12]	IDX[11:2]	1:0

VPN [31:16]	[15:0]
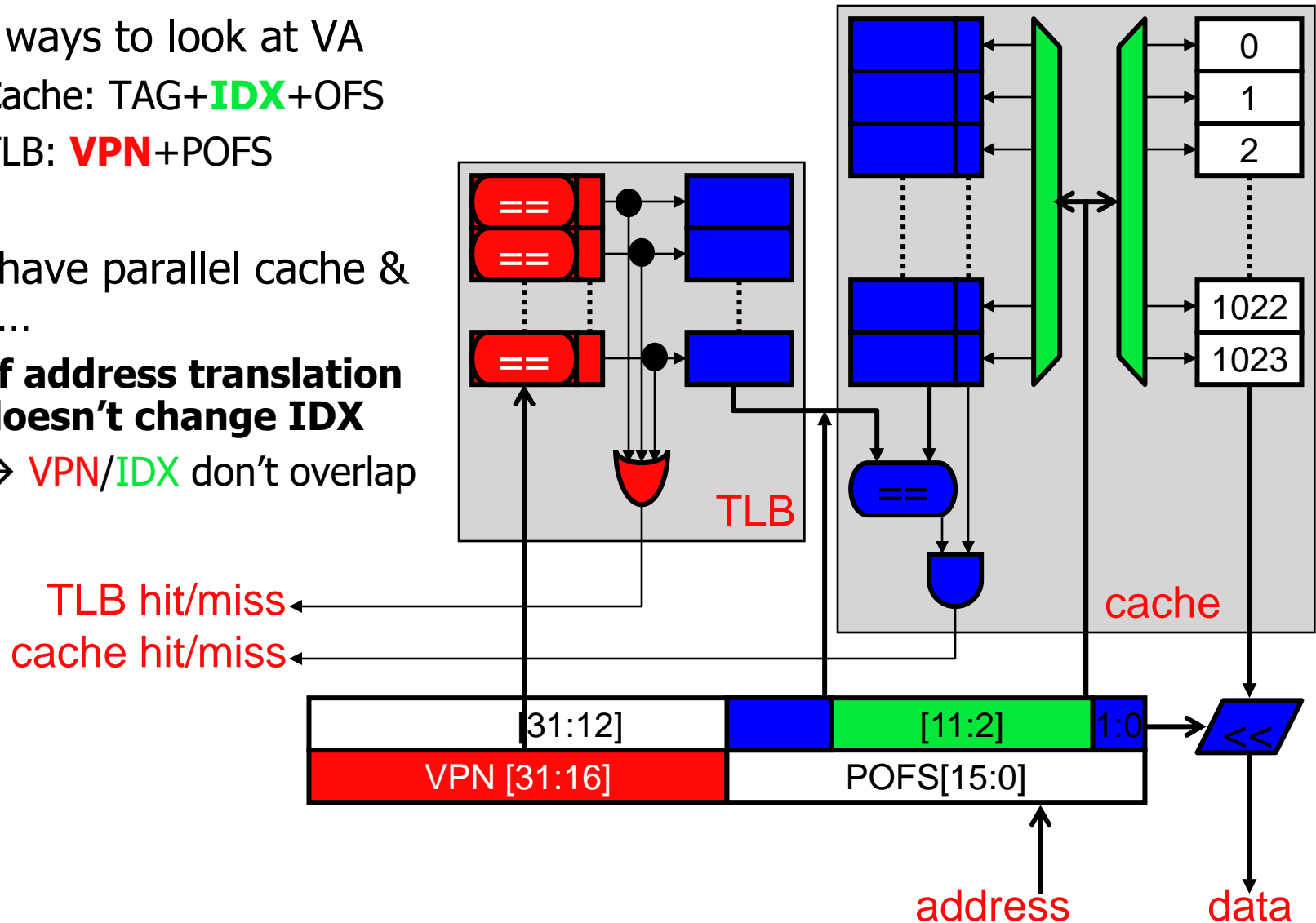
- Relationship between page size and L1 I$(D$) size
  - Forced by non-overlap between VPN and IDX portions of VA
    - Which is required for TLB access
  - I$(D$) size / **associativity** ≤ page size
  - Big caches must be set associative
    - Big cache → more index bits (fewer tag bits)
    - More set associative → fewer index bits (more tag bits)
  - Systems are moving towards bigger (64KB) pages
    - To amortize disk latency
    - To accommodate bigger caches

II. MECHANICS OF <u>SWAPPING</u>

Swapping is when you shove unpopular data to disk and fetch it back as needed

Memory as a finite resource

- We now see that physical memory pages are divvied up among processes

Limited resource, managed by OS
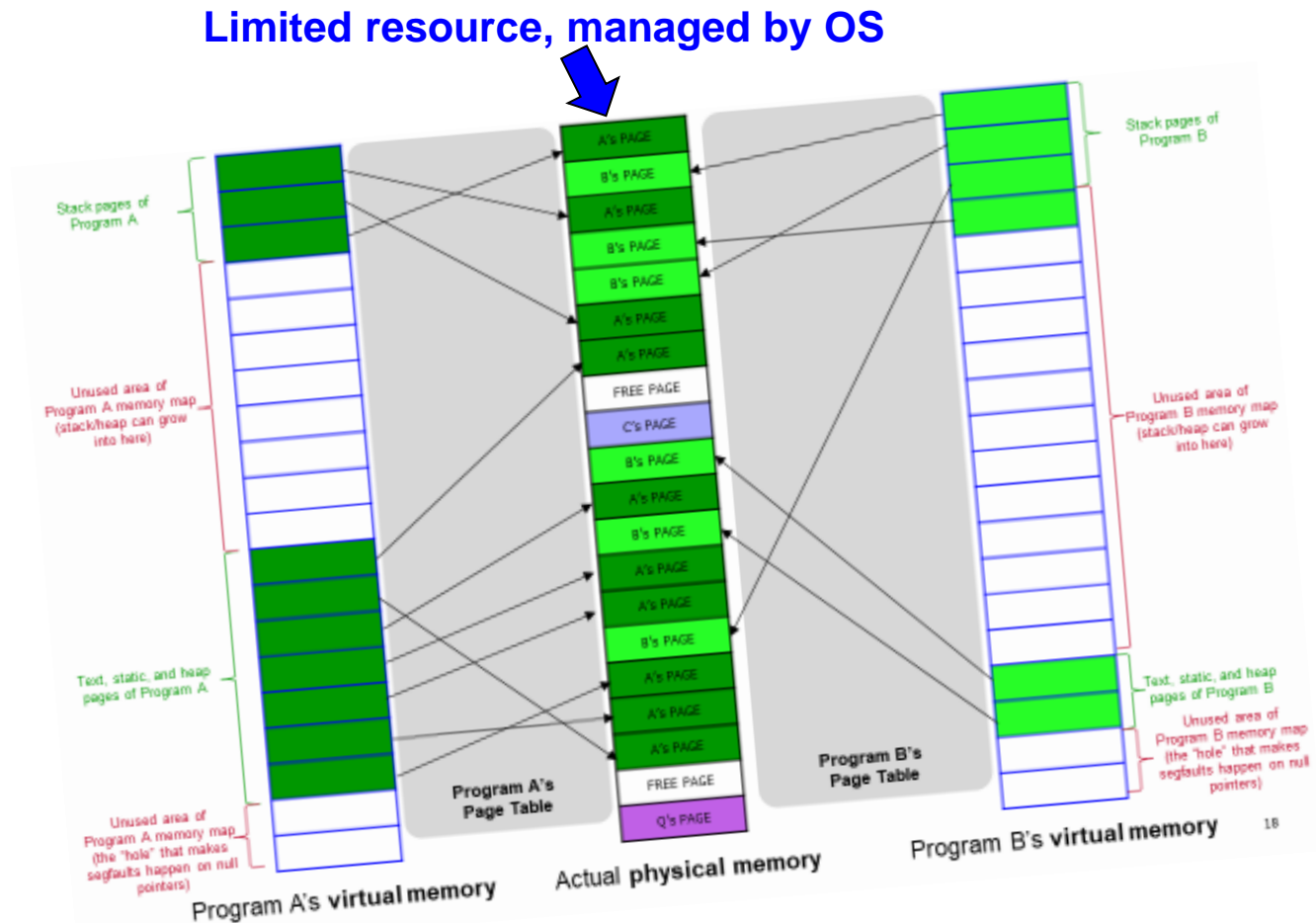
Thought experiment

- Let's consider Windows Calculator. 99.99% of the time, you use the main window of it, but it also has an about screen (Help | About):
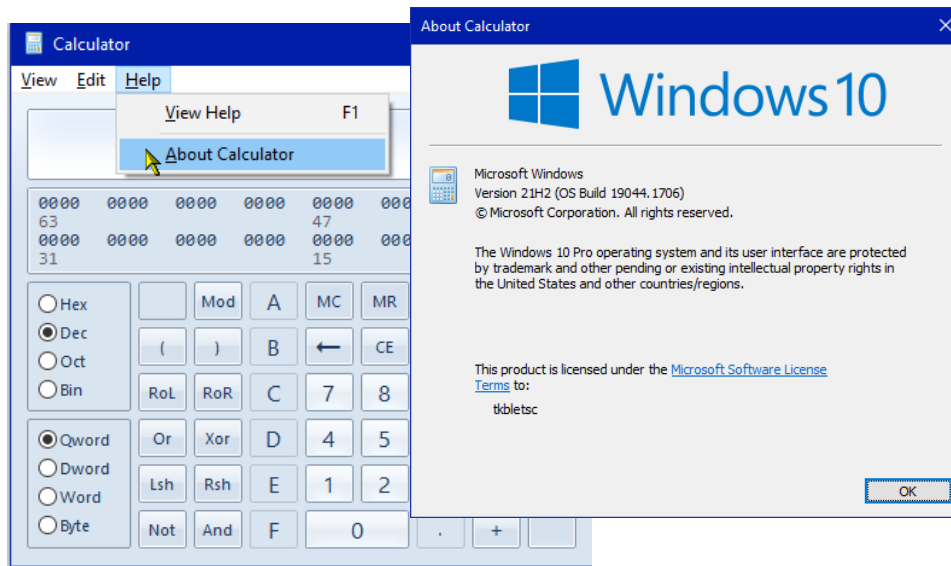
Should we spend precious physical memory to load this thing, which is almost never seen?

Wouldn't it be okay if it was a little slow to load?

- What about some of these browser tabs?
  There's some you haven't looked at since 2019…
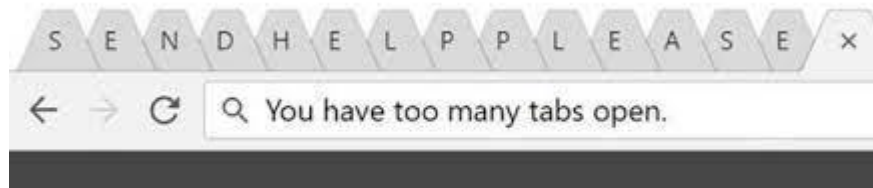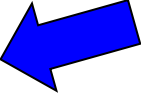
How does memory get allocated?

- When a program is **launched** or does a **malloc**, the OS identifies free physical pages and adds them to the page table for that process
  - This consumes physical memory

- Decision: what to do if there are NO free physical pages?
  - Deny the malloc? (Usually breaks the requesting program)
  - Kill some other program? (Dang, that's cold)
  - Can we find a better choice?
- Alternative:
  - Identify unimportant pieces of memory
  - Shove them somewhere (where?)
  - Only bring them back if/when they're needed
- This is **swapping**

Is there a place on your computer that's bigger than RAM, but slower?

Yes!
Permanent storage (disk or SSD)

Two parts to swapping

- **Swapping**: Pushing unneeded pieces of memory to disk, bringing them back as needed

- Two questions:
  1. How/when to swap things **in?** ⬅
  2. How/when to swap things **out?**

Page Faults

- What happens if I look for a page table entry, but the result has valid=0?

- **Page fault**: Page table entry not in TLB or in PT
  - Page is simply not in memory at all!
  - Starts out as a TLB miss, detected by OS handler/hardware FSM
  - OS page fault routine is triggered to respond

- What does it mean to have a **page fault**?
  - The virtual address requested doesn't correspond to anything in physical memory
  - One possibility: programmer tried to access an invalid pointer
  - Result? The OS kills your process and prints out "Segmentation fault"
    - THIS IS WHERE SEGFAULTS COME FROM!

  - *Another possibility: time to **swap** something back in from disk!*

Swapping: a neat magic trick

SEGFAULT

OK (fast)

OK (fast)

OK (but slow)

Page table

Virtual memory

Physical memory

HDD/SSD storage

35
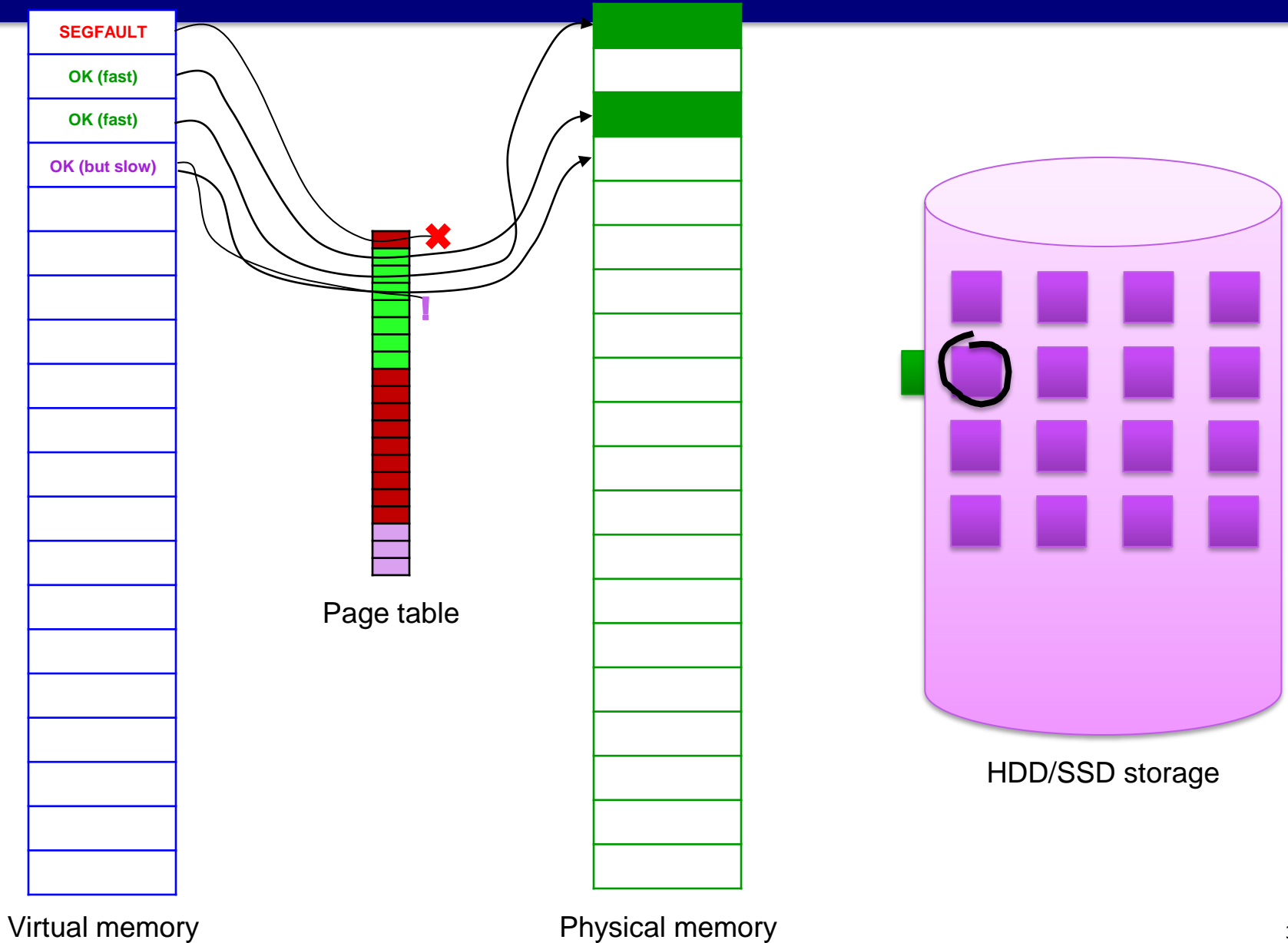
How to tell a segfault from a swap

- When there's a **page fault**, the OS handler kicks in
  - Looks at OS data structures to figure out:
    - Is this something I swapped out earlier? Swap it back in… or
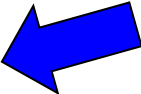   - Did the programmer just screw up? Kill the process…

Done in hardware

Memory reference → Is in physical memory? → Y → Success

PAGE FAULT

N → Is page stored on disk? → Y → Load it, success

N → Invalid reference, abort!

Done by OS (software)
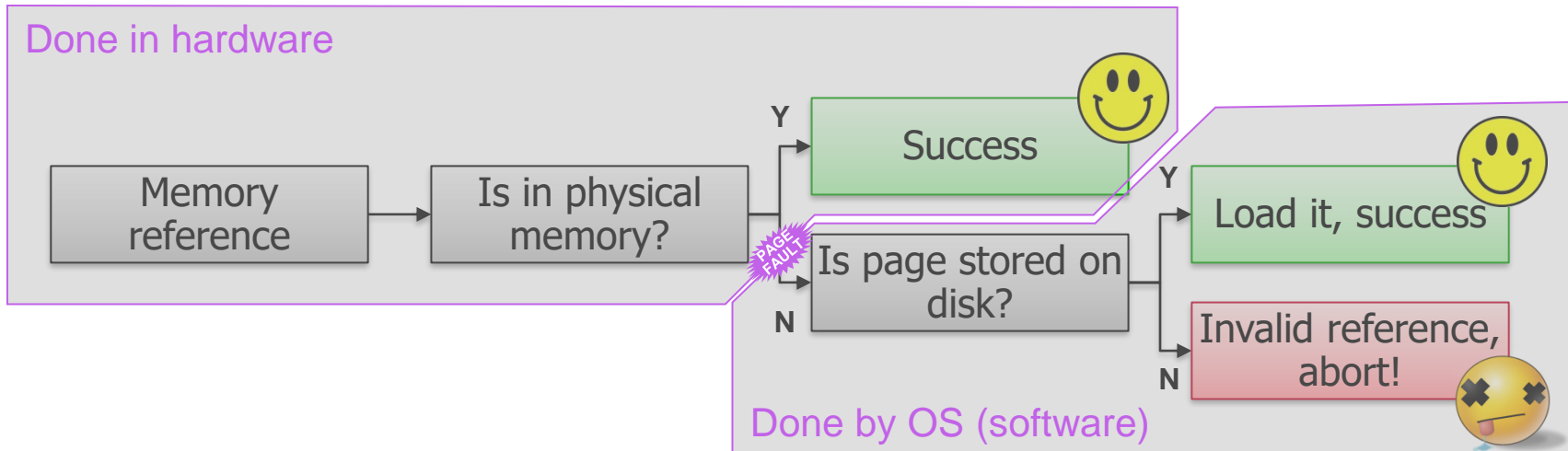
Page Fault

- Steps to swap in:
  - Get empty physical page
  - Schedule a disk read to load the data into the new physical page
  - Reset page table entry to indicate the new page and valid=1
  - Restart the instruction that caused the page fault: now it works!
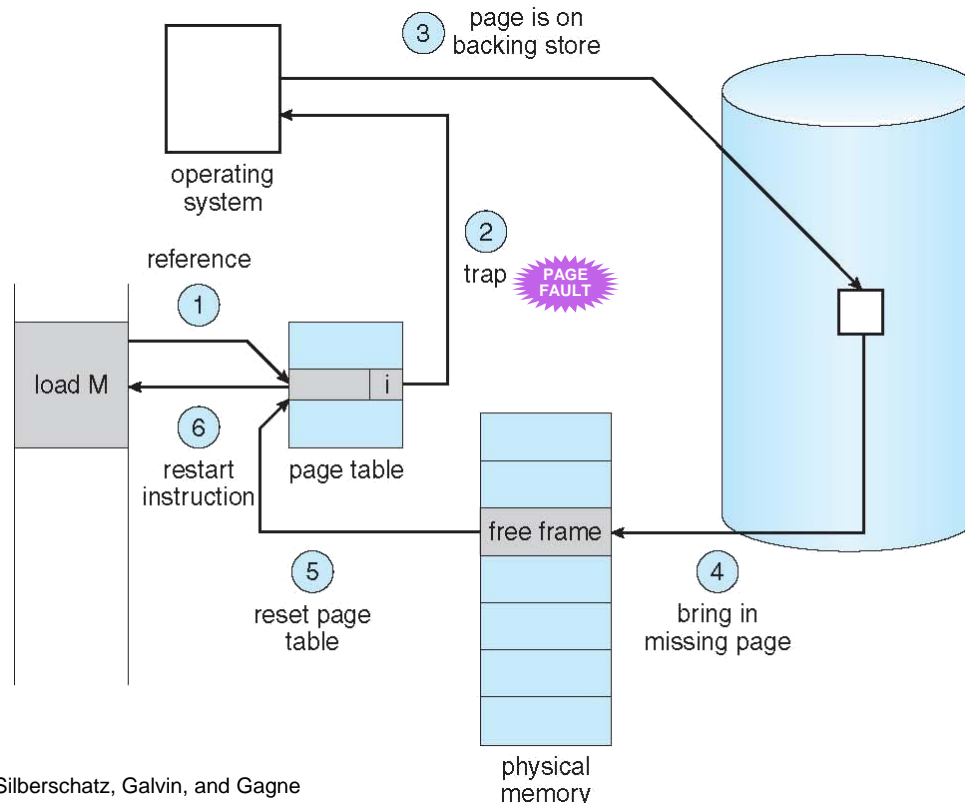
Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne

Two parts to swapping

- **Swapping**: Pushing unneeded pieces of memory to disk, bringing them back as needed

- Two questions:
  1. How/when to swap things **in?**
  2. How/when to swap things **out?** ⬅

What happens if there is no free physical page?

- **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
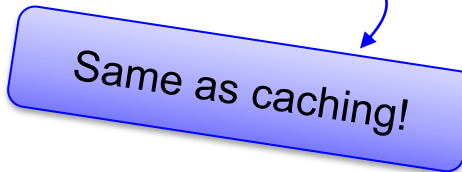
Page Replacement Algorithms

- **Physical page allocation algorithm**
    - How many physical pages to give each process
    - We'll talk about this later in the context of *working set analysis...*

- **Page-replacement algorithm**
    - Picks which page to swap out to disk
    - 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

- 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 2 1 2 0 1 7 0 1

7	7	7	2		2	2	4	4	4	0		0	0		7	7	7
	0	0	0		3	3	3	2	2	2		1	1		1	0	0
		1	1		1	0	0	0	3	3		3	2		2	2	1
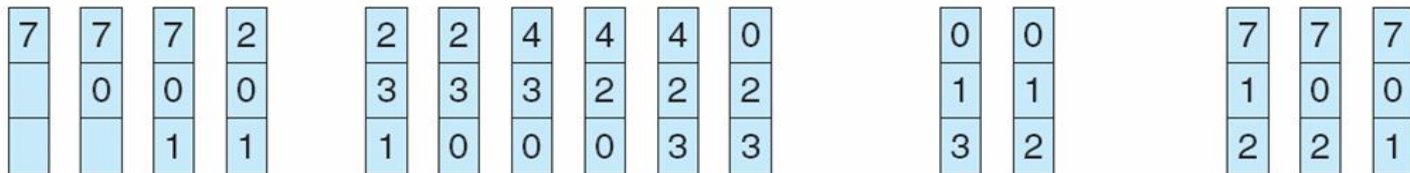
page frames

FIFO Illustrating Belady's Anomaly

- 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)
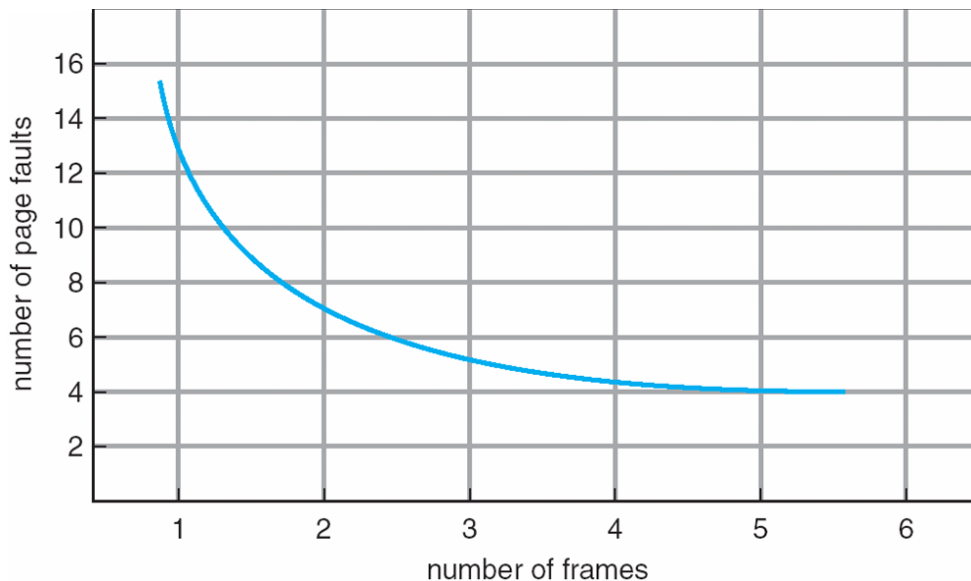
Reality
(behavior of FIFO algorithm)

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne

42

Optimal Algorithm

- 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

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne
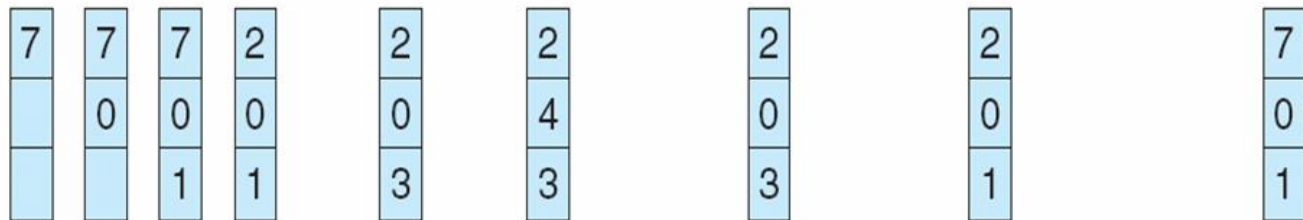
Least Recently Used (LRU) Algorithm

- 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 |
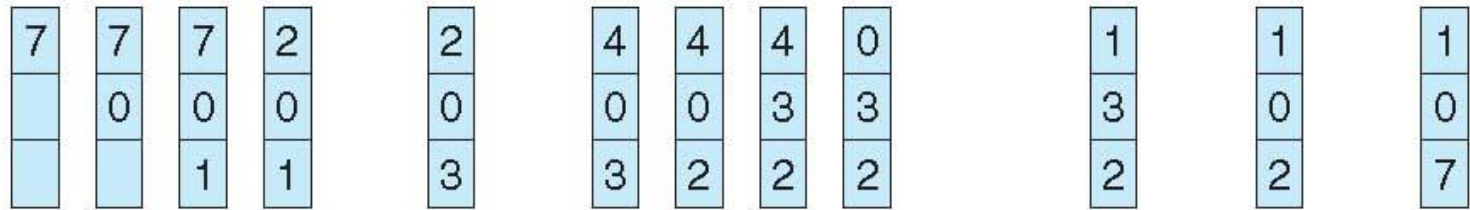
page frames

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

44

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne

LRU Algorithm (Cont.)

- 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, <u>find</u> 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

- 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

- 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

What happens if there is no free physical page?

- **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

LRU or something like it

Two parts to swapping – summarized

- **Swapping**: Pushing unneeded pieces of memory to disk, bringing them back as needed

- Two questions:
  1. How/when to swap things **in?**
     - When we have a page fault (page table has no valid entry), OS checks if it's because that thing was swapped out before
     - If so, it reads it into a free physical page, updates page table, restarts instruction that triggered fault
  2. How/when to swap things **out?**
     - When we reach for a free physical page and find none, we identify a victim page to write to disk
     - We update the page table for the removed victim page (valid=0) and make note of the swap so we can bring it back in if needed

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	https://en.wikipedia.org/wiki/Instructions_per_second
Time for light to traverse CPU core (~13mm)	44	40 ps	Intel Core i7 4770k (Haswell), 3.9GHz	http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested/5
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	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: L2 hit	3,636	4 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: L3 hit	8,439	8 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: DRAM	64,485	60 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Process context switch or system call	3,000,000	3 us	Intel E5-2620 (Sandy Bridge), 2GHz	http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html
Storage sequential read**, 4kB (SSD)	7,233,796	7 us	SSD: Samsung 840 500GB	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage sequential read**, 4kB (HDD)	65,104,167	70 us	HDD: 2.5" 500GB 7200RPM	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage random read, 4kB (SSD)	100,000,000	100 us	SSD: Samsung 840 500GB	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage random read, 4kB (HDD)	10,000,000,000	10 ms	HDD: 2.5" 500GB 7200RPM	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
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	eurodns.com	Ping
Time for light to travel to the moon (average)	1,348,333,333,333	1 s	The moon	http://www.wolframalpha.com/input/?i=distance+to+the+moon

* 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
- Save the user registers and process state
- Check that the page reference was legal and determine the location of the page on the disk
- Issue a read from the disk to a free frame:
  - Wait in a queue for this device until the read request is serviced
  - 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
- Correct the page table and other tables to show page is now in memory
- 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

The pain of swapping

- A single swap event is orders of magnitude slower than RAM
- **Result: need a very low swap rate**

- Here are some optimizations that help achieve that

Keep track of stuff already on disk: "Dirty bit"

- Some memory data will *already* be on disk

  - Example: parts of program code,
    like Calculator's About screen

  - Also: something swapped out previously,
    brought back in, and not modified since

- Optimization: use **"dirty" bit** in page table to track pages
  modified since loading; only modified pages are written to disk

- To "swap out" a page that's "clean", you just drop it,
  knowing that you can find a copy of it on disk
    (avoids the write-to-disk step)

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

We want to avoid <u>thrashing</u>

- 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**: a process is busy swapping pages in and out

Thrashing (Cont.)

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne
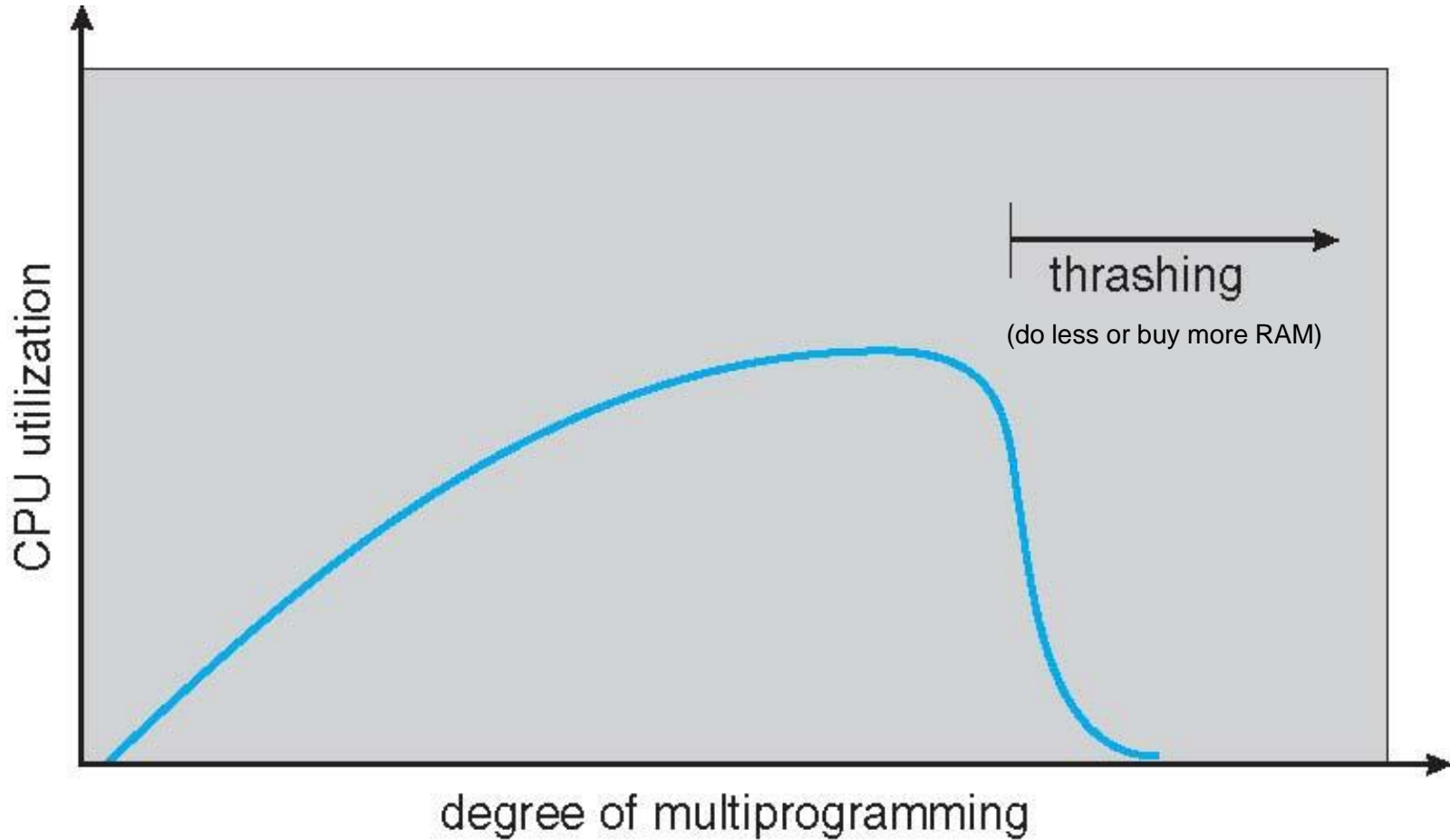
Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne
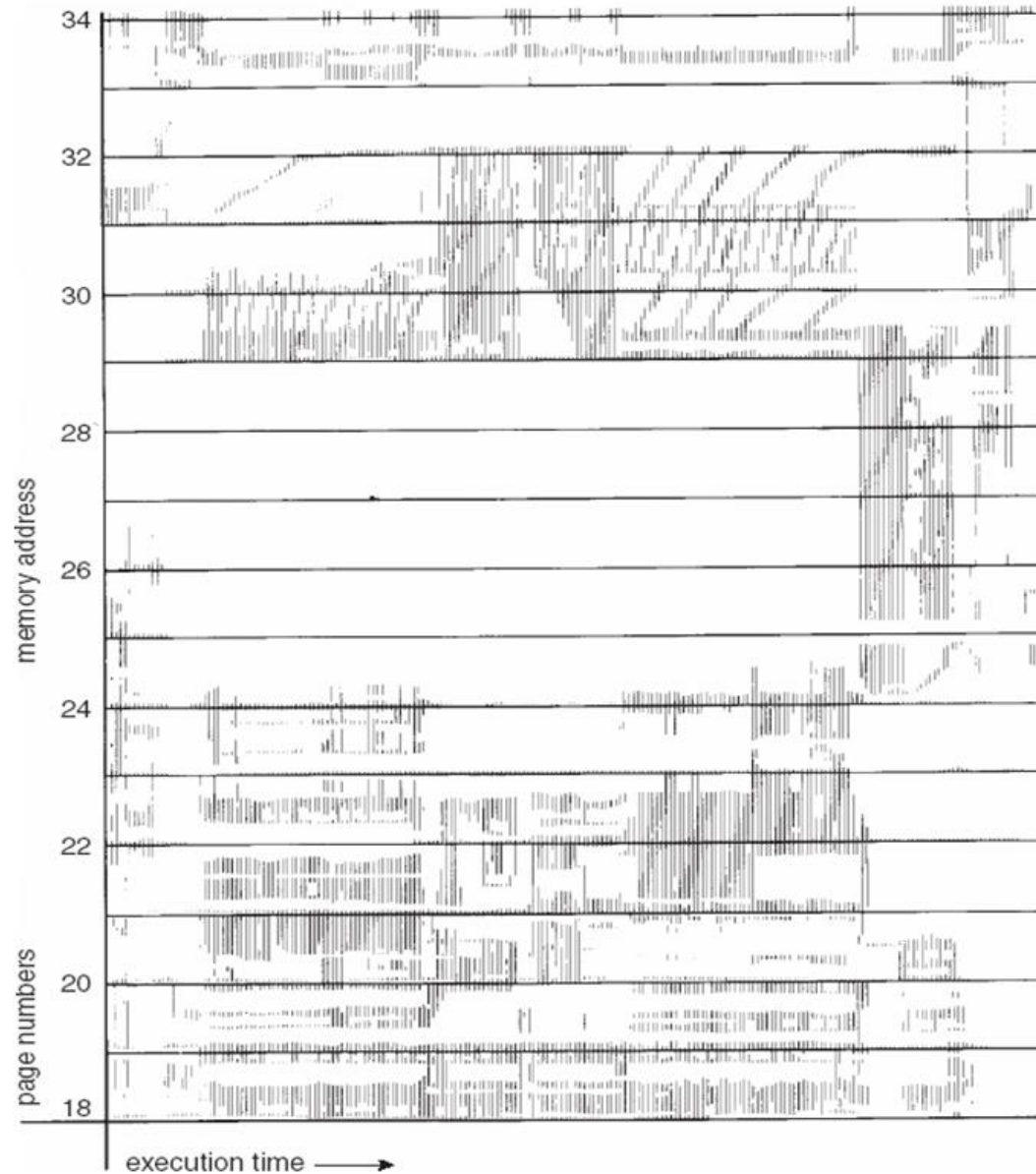
Working-set model

- $\Delta \equiv$ 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 Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

- $D = \Sigma \, WSS_i \equiv$ 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 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .

Δ t_1 Δ t_2
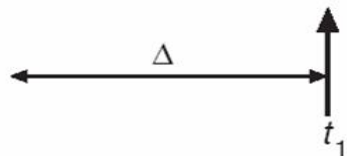
$WS(t_1) = \{1,2,5,6,7\}$ $WS(t_2) = \{3,4\}$
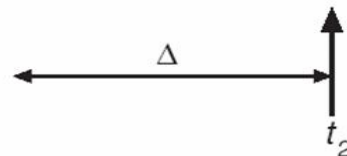
- Address translation via **page table**
  - Page table turns VPN to PPN (noting the valid bit)
- Page marked as valid=0? **Page fault**.
  - If OS has stored page on disk, load and resume
  - If not, this is invalid access, kill app (seg fault)
- Governing policies:
  - Keep a certain **number of frames 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.
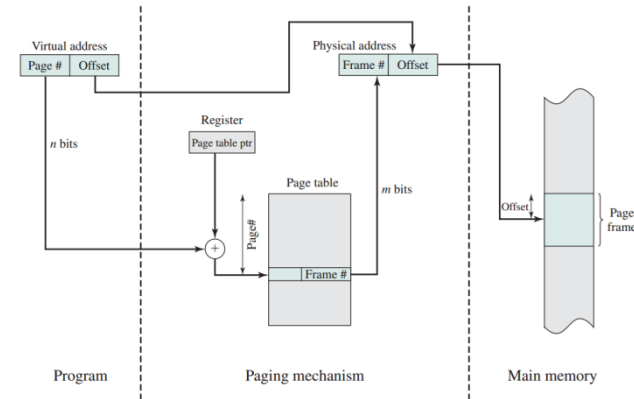
OTHER CONSIDERATIONS
(TIME PERMITTING)

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 scanred. 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)?

address

data
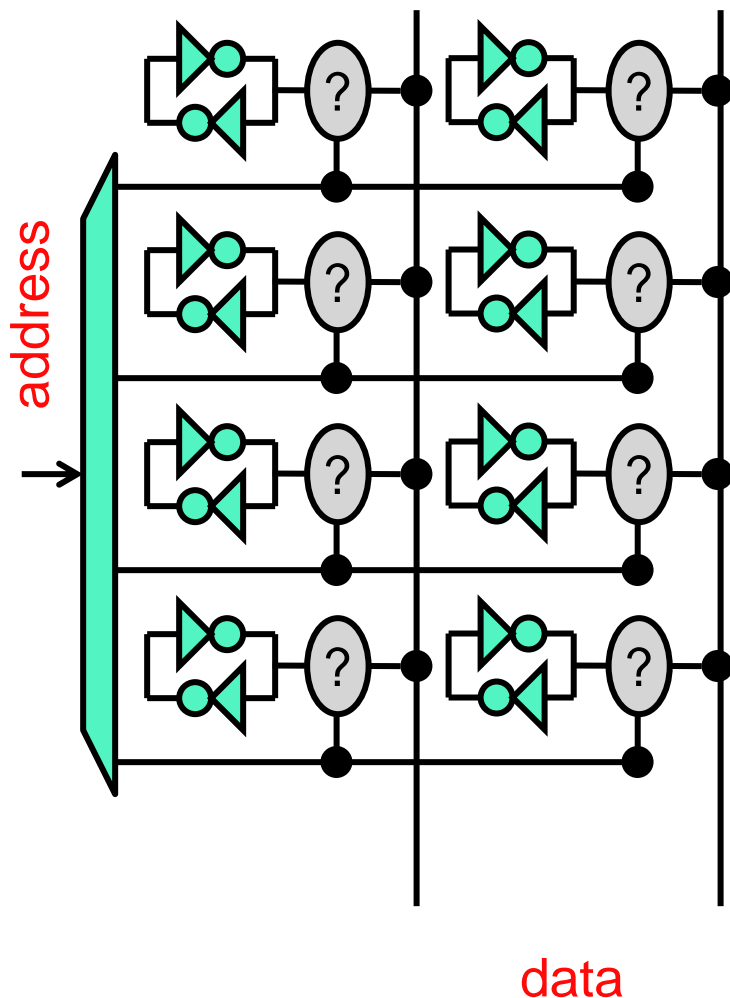
- **SRAM**: static RAM
  - Bits as cross-coupled inverters
  - Four transistors per bit
  - More transistors for ports

- **"Static"** means
  - Inverters connected to power/ground
  - Bits naturally/continuously "refreshed"
  - Bit values never decay

- Designed for speed

© Daniel J. Sorin from Roth

Dynamic RAM (DRAM)

address

data
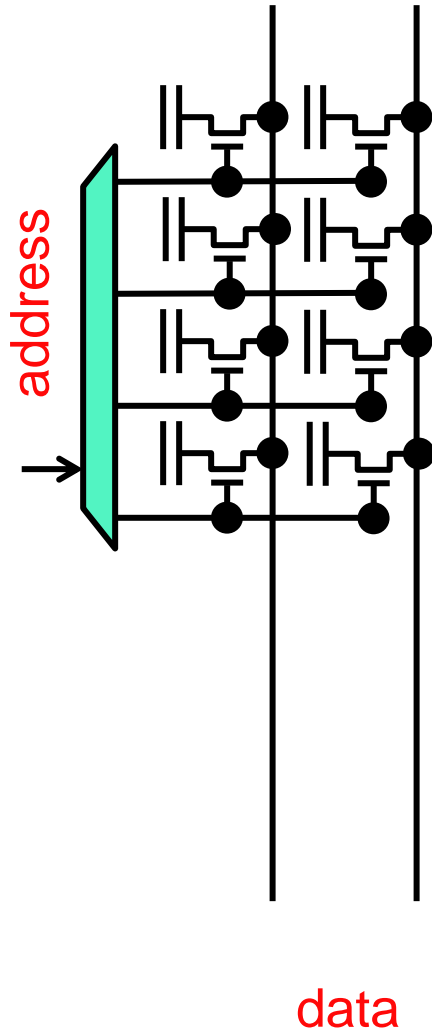
- **DRAM**: dynamic RAM
  - Bits as capacitors (if charge, bit=1)
  - "Pass transistors" as ports
  - One transistor per bit/port

- **"Dynamic"** means
  - Capacitors not connected to power/gnd
  - Stored charge decays over time
  - Must be explicitly refreshed

- Designed for density
  - Moore's Law …

© Daniel J. Sorin from Roth

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
  - DDR4 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 ~1K pages
- We could, but won't, spend rest of semester on DRAM

YET MORE STUFF

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

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
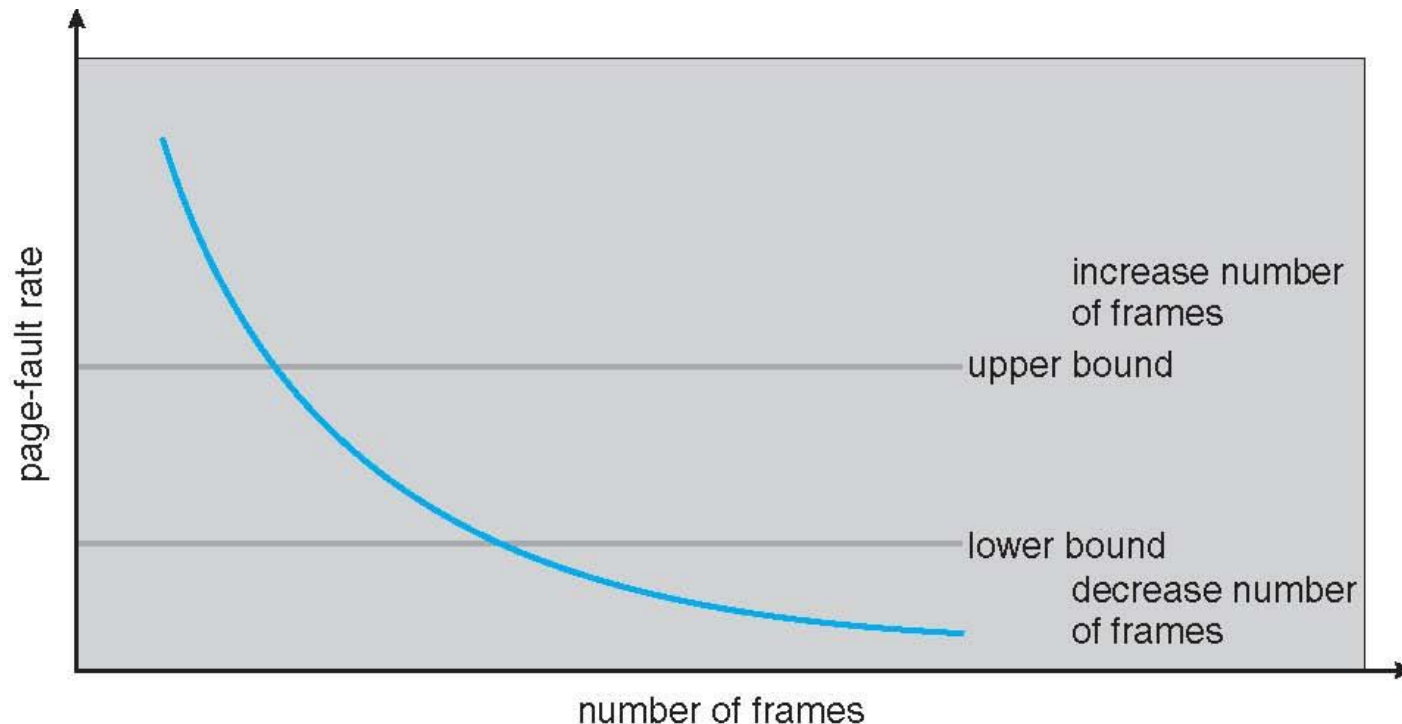
Copy-on-Write

- 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?
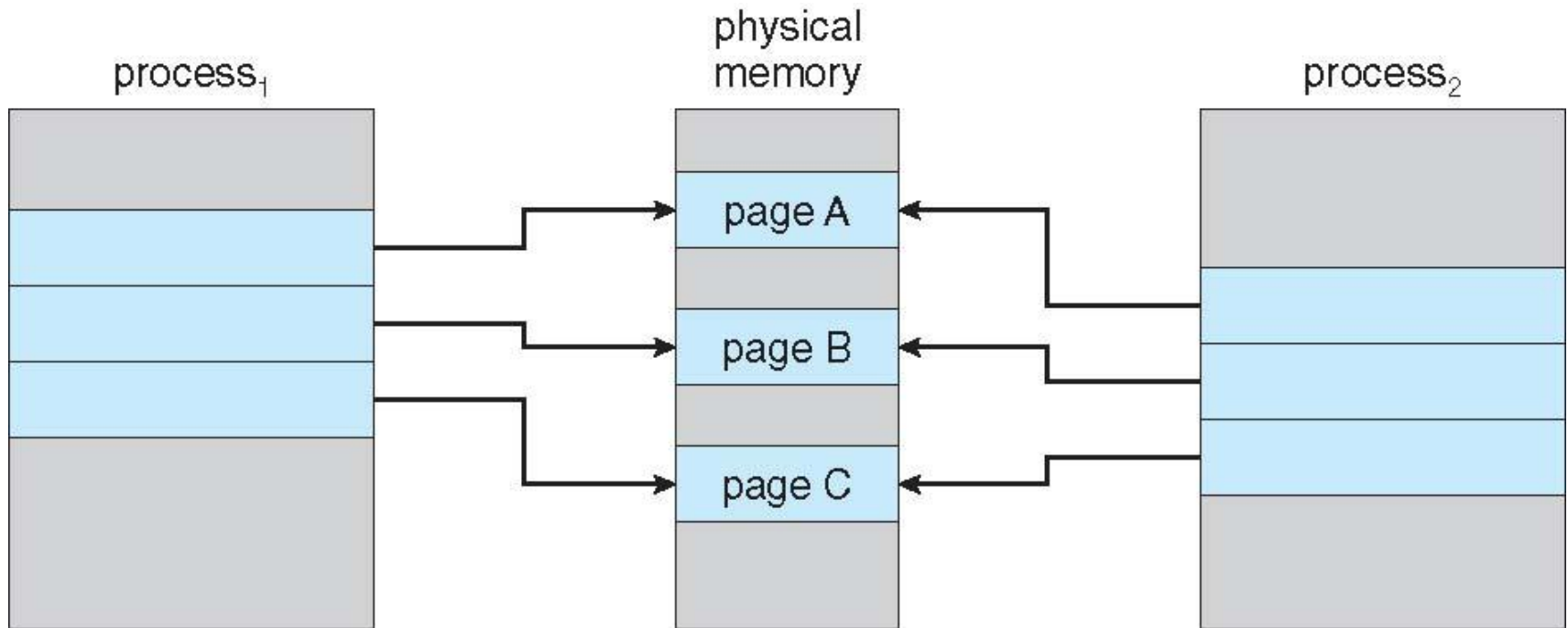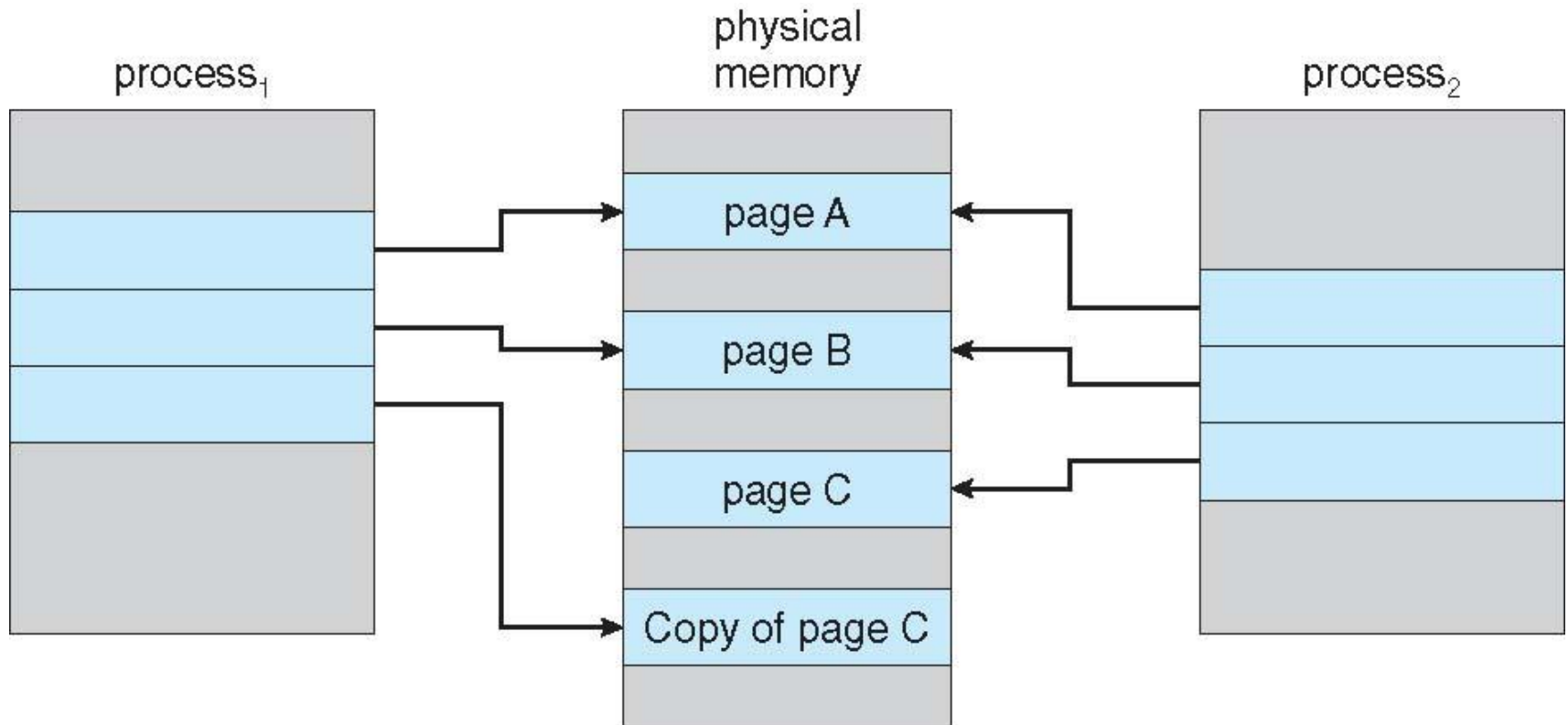
Non-Uniform Memory Access

- 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).
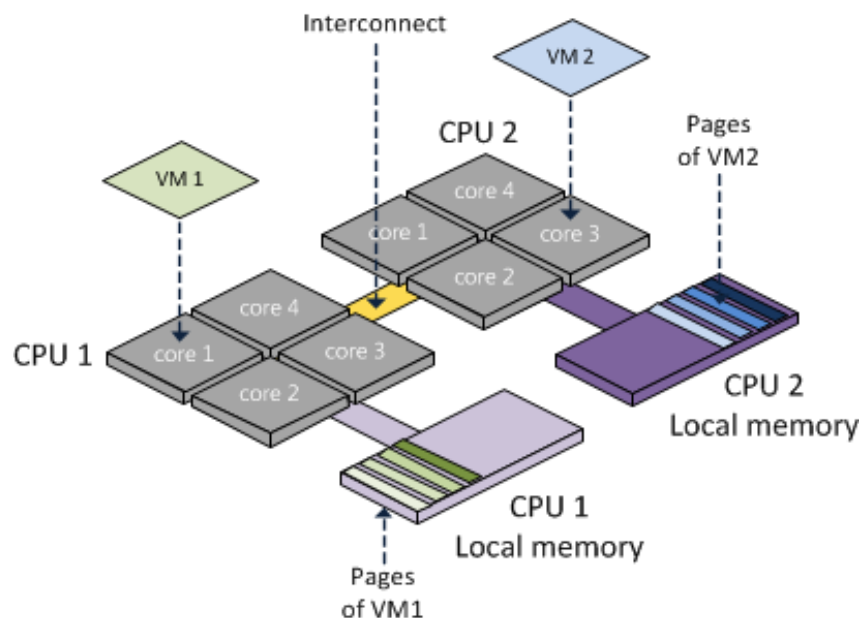
Memory-Mapped Files

- 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

process A virtual memory

physical memory

process B virtual memory

disk file
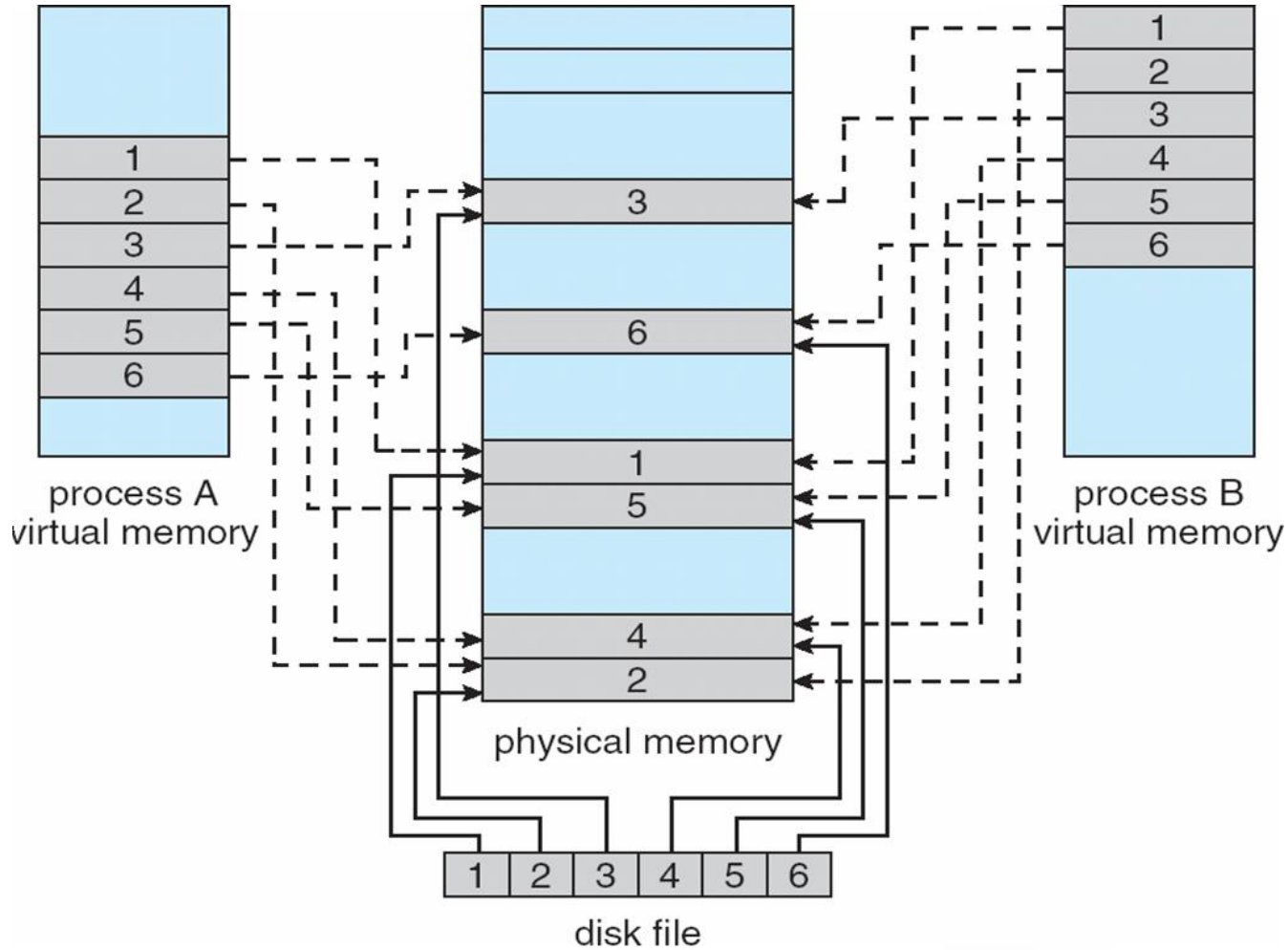
Applications and Page Replacement

- 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)
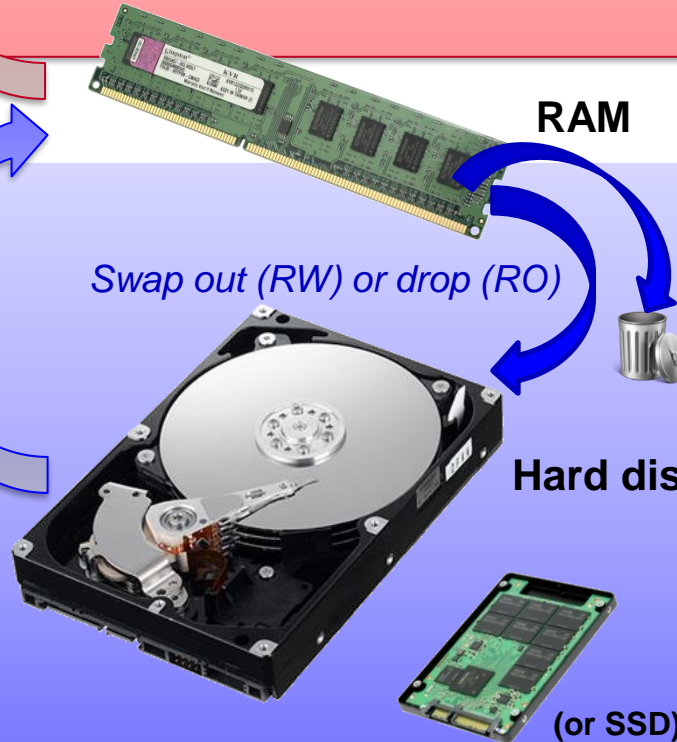
Caching and Swapping together

CACHING

Shared L3 Cache
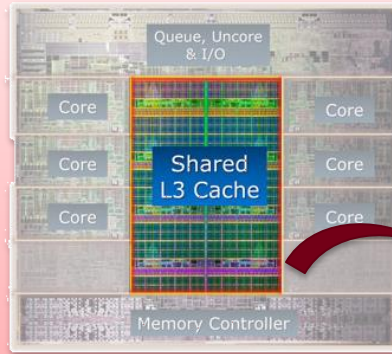
Cache

- Faster
- More expensive
- Lower capacity

Drop

Copy if **popular**

RAM

SWAPPING

Load if **needed**

Swap out (RW) or drop (RO)

Hard disk

- Slower
- Cheaper
- Higher capacity

(or SSD)