Operating Systems

Operating Systems

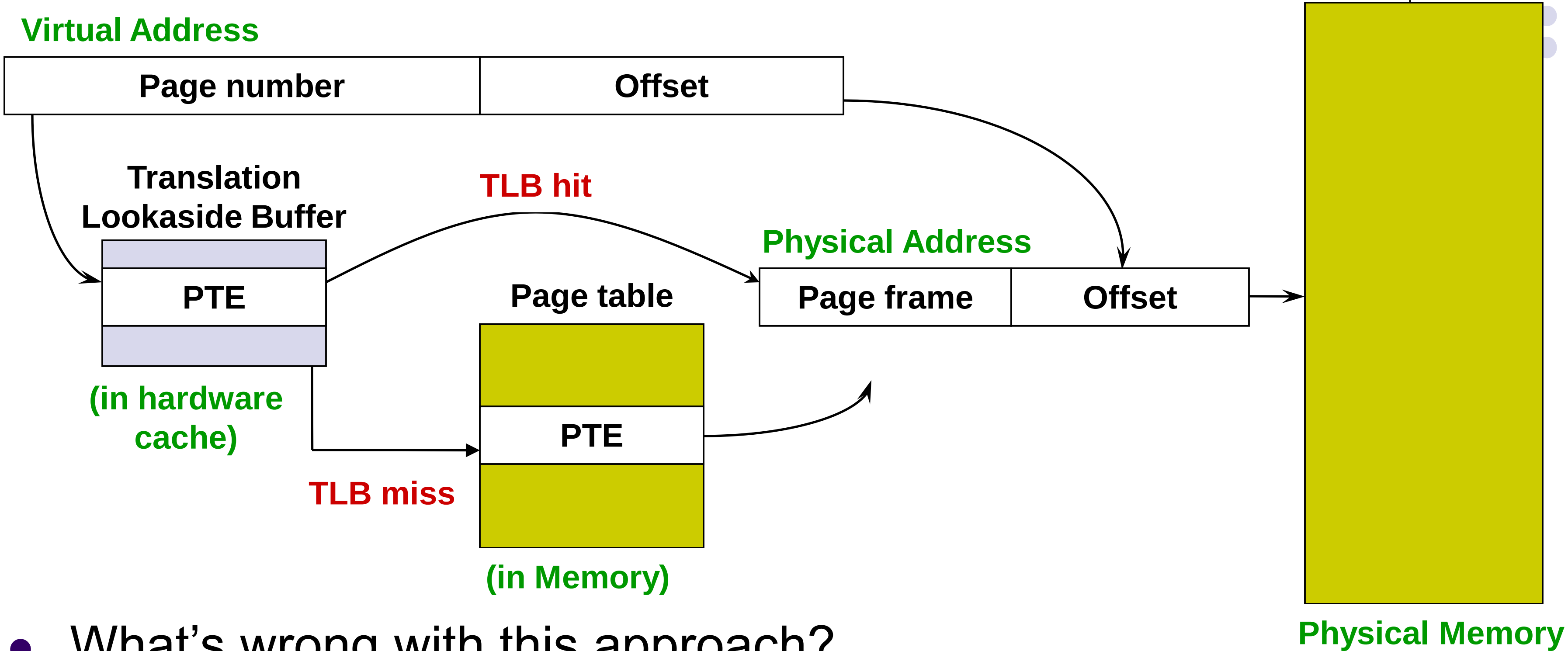
Sina Meraji

U of T

Recap

- Last time we looked at memory management techniques
  - Fixed partitioning
  - Dynamic partitioning
  - Paging

Page Lookups Overview

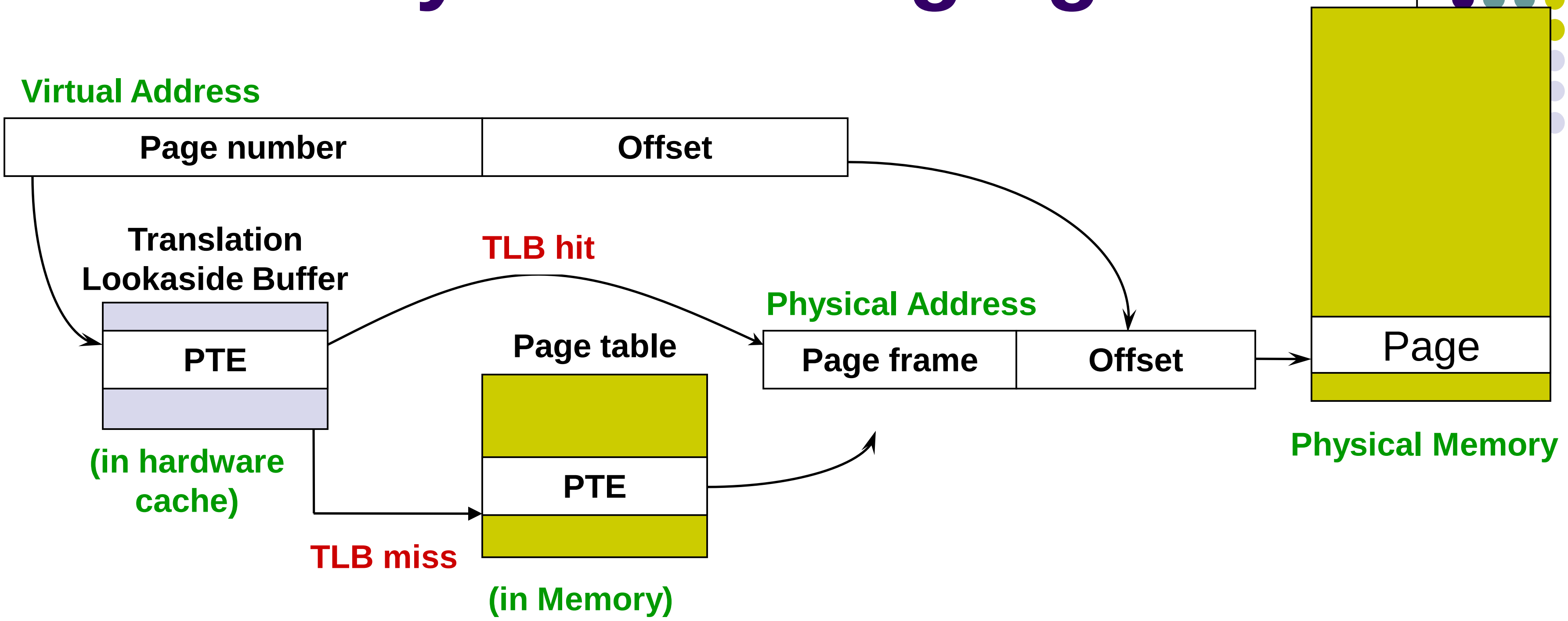


- What's wrong with this approach?
  - Need 2 references for address lookup (first page table, then actual memory)
- Idea: Use hardware cache of page table entries
  - Translation Lookaside Buffer (TLB)
  - Small, fully-associative hardware cache of recently used translations

TLBs

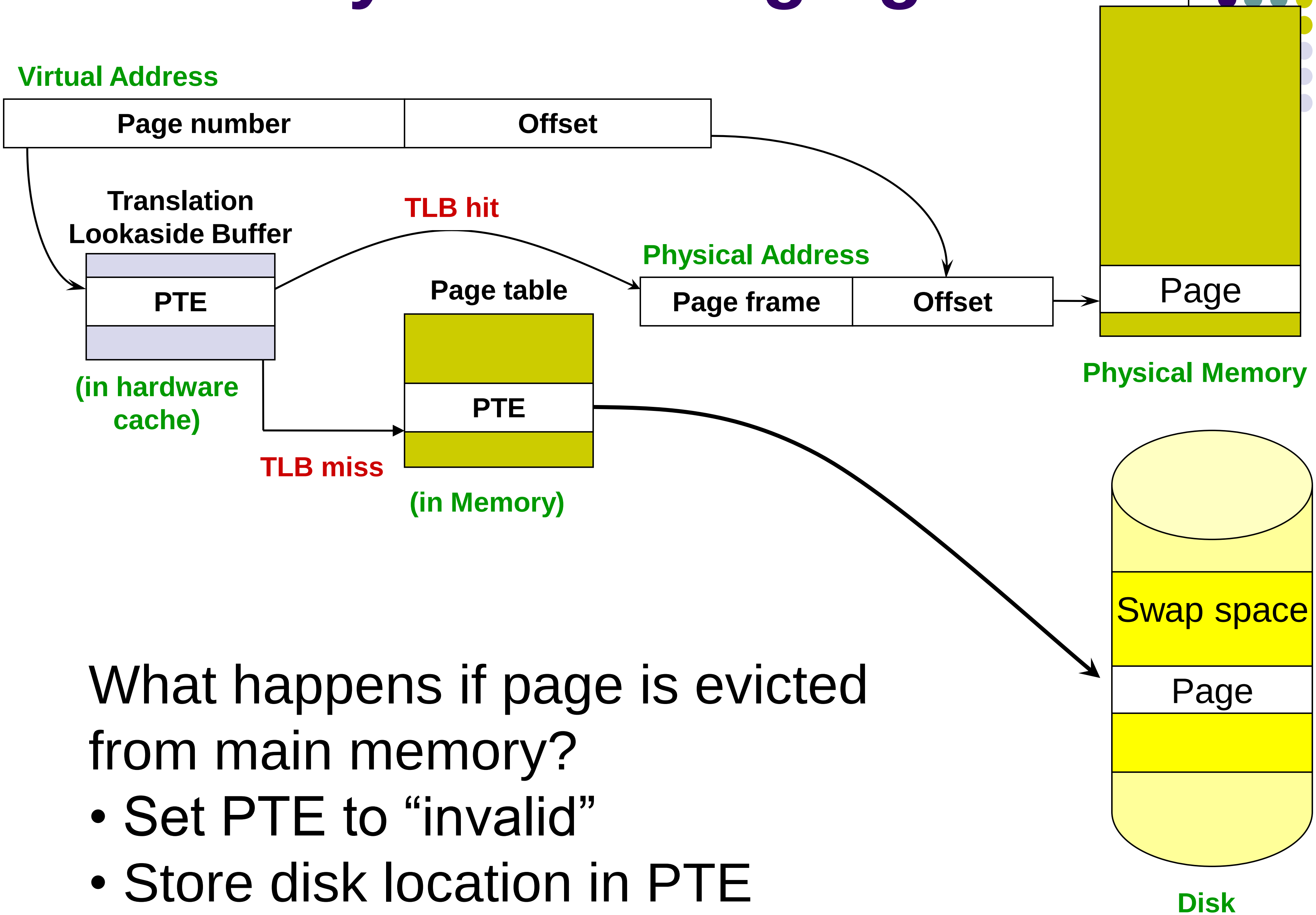
- TLBs are small (64 – 1024 entries)
- Still, address translations for most instructions are handled using the TLB
  - **>99% of translations**, but there are misses (TLB miss)…
- TLBs exploit **locality**
  - Processes only use a handful of pages at a time
    - 16-48 entries/pages (64-192K)
    - Only need those pages to be “mapped”
  - Hit rates are therefore very important

Summary so far: Paging



What happens if not all pages of all processes fit into physical memory?

Summary so far: Paging



What happens if page is evicted from main memory?

- Set PTE to “invalid”
- Store disk location in PTE

How much space does a page table take up?

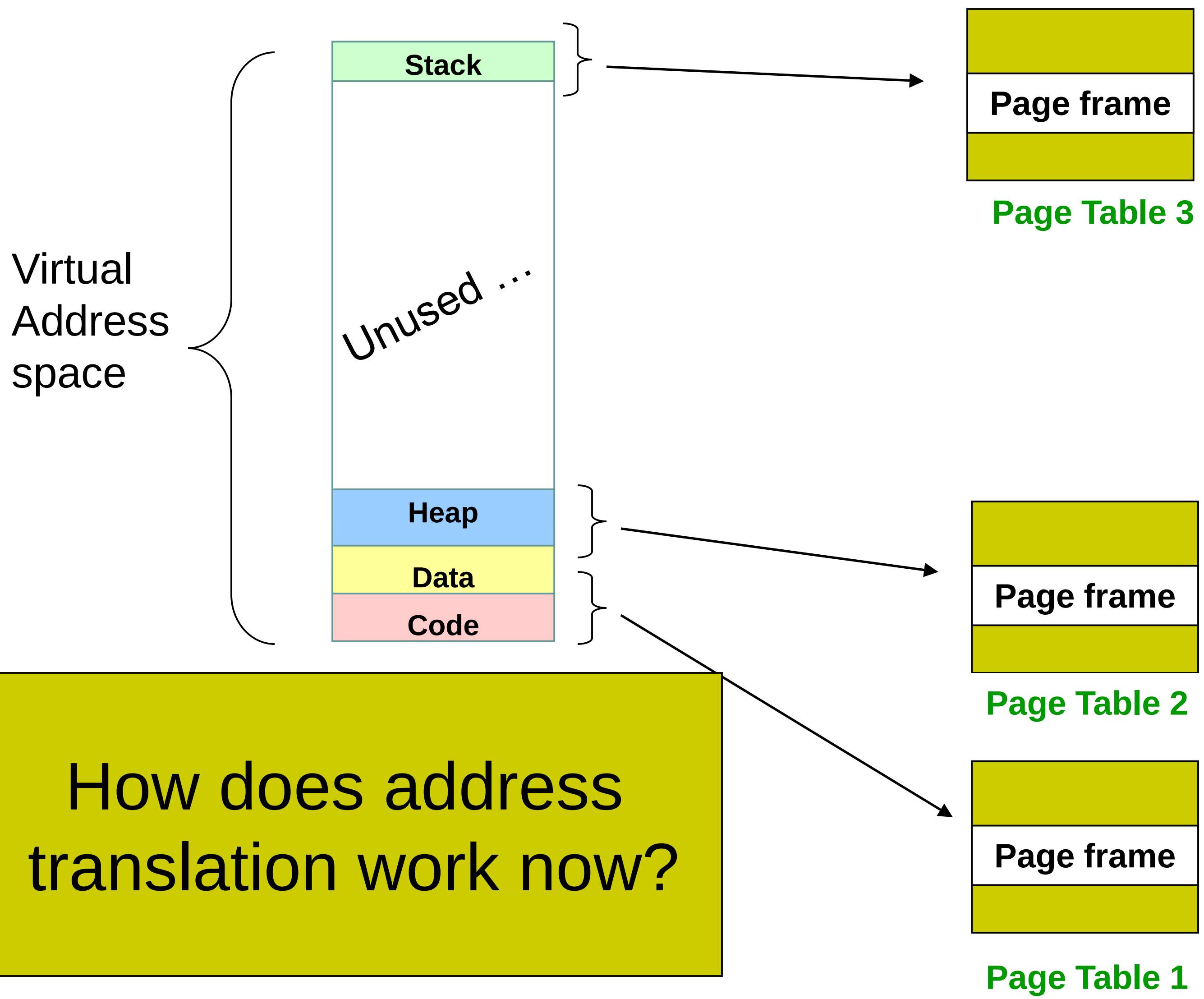
- Need one PTE per page
- 32 bit virtual address space w/ 4K pages
  - = 2^{20} PTEs
- 4 bytes/PTE = 4MB/page table
- 25 processes = 100MB just for page tables!
  - And modern processors have 64-bit address spaces -> 16 petabytes for page table!

- Solutions
  - Hierarchical (multi-level) page tables
  - Hashed page tables
  - Inverted page tables

Managing Page Tables

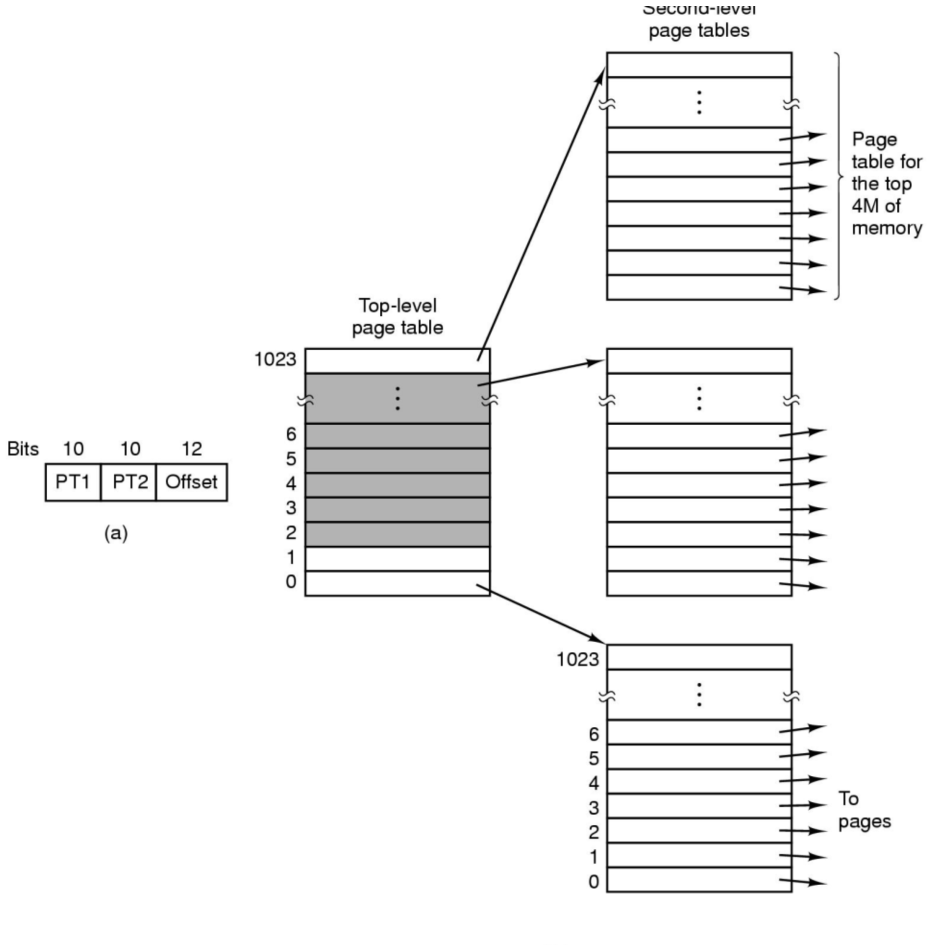
- How can we reduce space overhead?
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)
- How do we only map what is being used?
  - Can dynamically extend page table…
  - Does not work if addr space is sparse (internal fragmentation)
- Use another level of indirection: two-level page tables (or multi-level page tables)

Motivation: two-level page tables



How does address translation work now?

Multilevel Page Tables

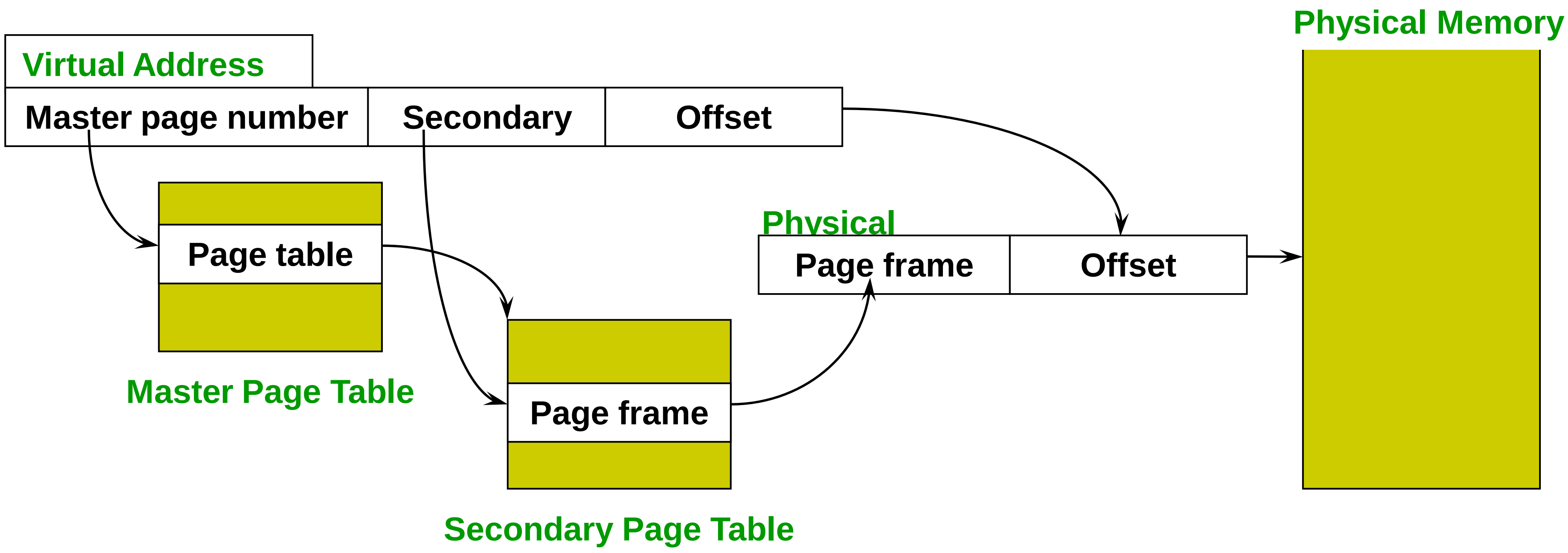


(a) A 32-bit address with two page table fields.
(b) Two-level page tables.

Two-Level Page Tables

Virtual addresses (VAs) have three parts:

- Master page number, secondary page number, and offset
- Master page table maps VAs to secondary page table
- Secondary page table maps page number to physical frame
- Offset selects address within physical frame



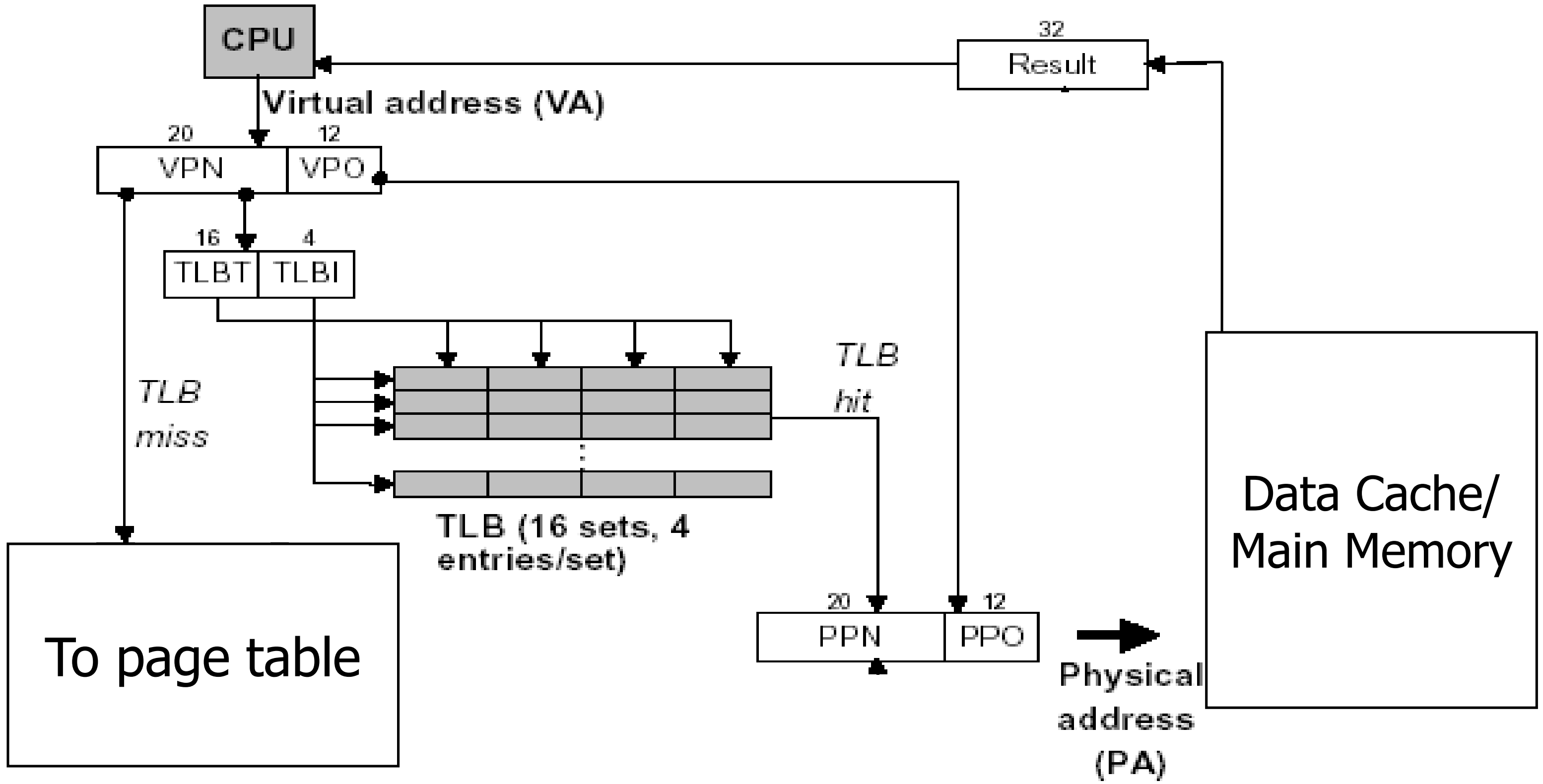
2-Level Paging Example

Virtual Address

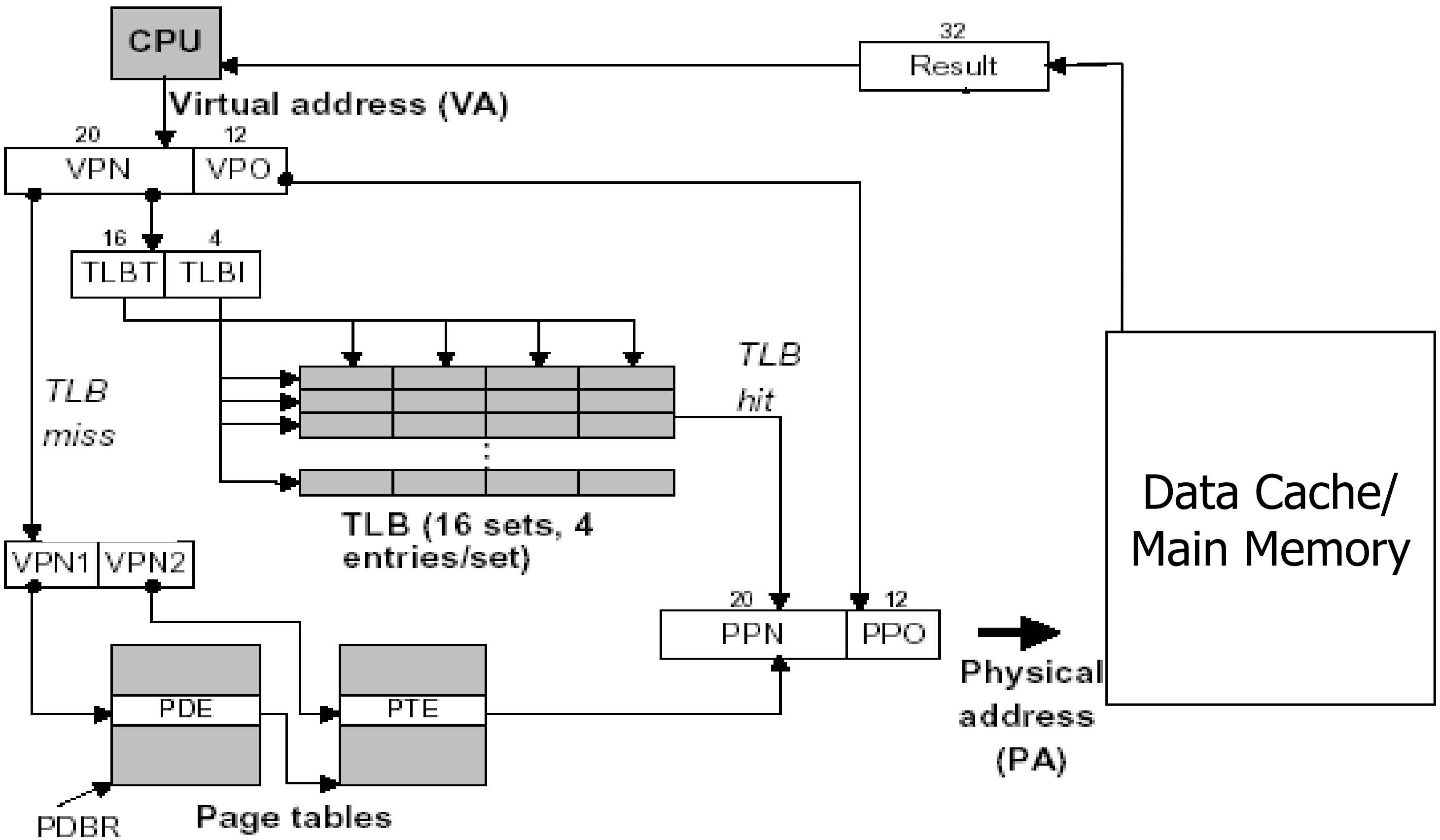
Master page number	Secondary	Offset
10 bits	10 bits	12 bits

- 32-bit virtual address space
  - 4K pages, 4 bytes/PTE
  - How many bits in offset?
    - 4K = 12 bits, leaves 20 bits
  - Want master/secondary page tables in 1 page each:
    - 4K/4 bytes = 1K entries.
      - How many bits to address 1K entries?
      - 10 bits
    - master = 10 bits
    - offset = 12 bits
    - secondary = 32 – 10 – 12 = 10 bits
  - This is why 4K is common page size!

Pentium Address Translation



Pentium Address Translation



Inverted Page Tables(Read the book)

- Keep one table with an entry for each physical page frame
- Entries record which virtual page # is stored in that frame
  - Need to record process id as well
- Less space, but lookups are slower
  - References use virtual addresses, table is indexed by physical addresses
  - Use hashing (again!) to reduce the search time

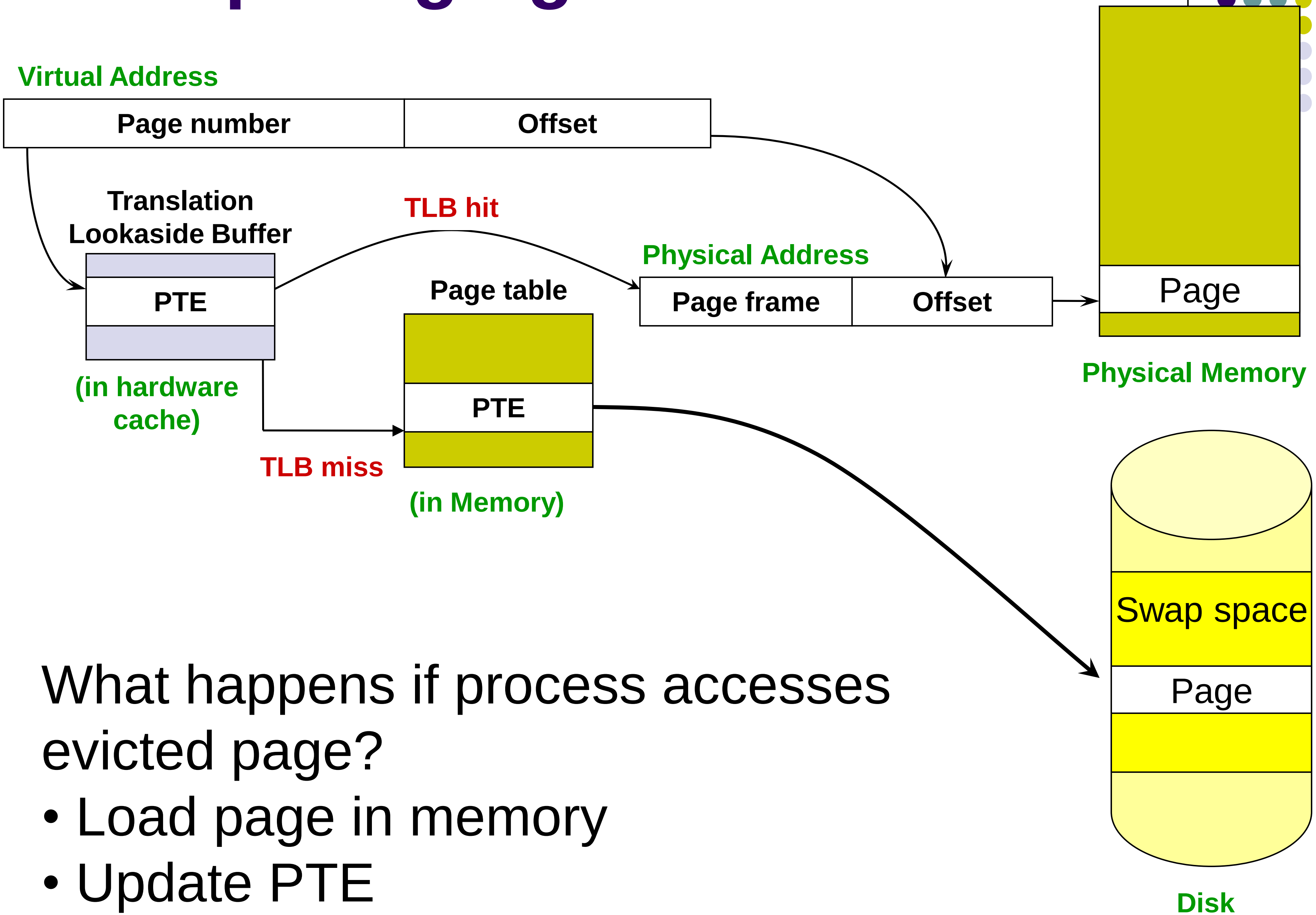
Efficient Translations

- Our original page table scheme already doubled the cost of doing memory lookups
  - One lookup into the page table, another to fetch the data
- Two-level page tables triple the cost!
  - Two lookups into the page tables, a third to fetch the data
  - And this assumes the page table is in memory
- TLB's hide the cost for frequently-used pages

Page allocation & eviction

- What happens when new page is allocated?
  - Initially, pages are allocated from memory
  - When memory fills up:
    - Some other page needs to be evicted from memory
    - This is why physical memory pages are called "frames"
  - Evicted pages go to disk (the swap file)
  - When it evicts a page, the OS sets the **PTE as invalid** and stores the location of the page in the swap file in the PTE

Recap: Paging



What happens if process accesses evicted page?

- Load page in memory
- Update PTE

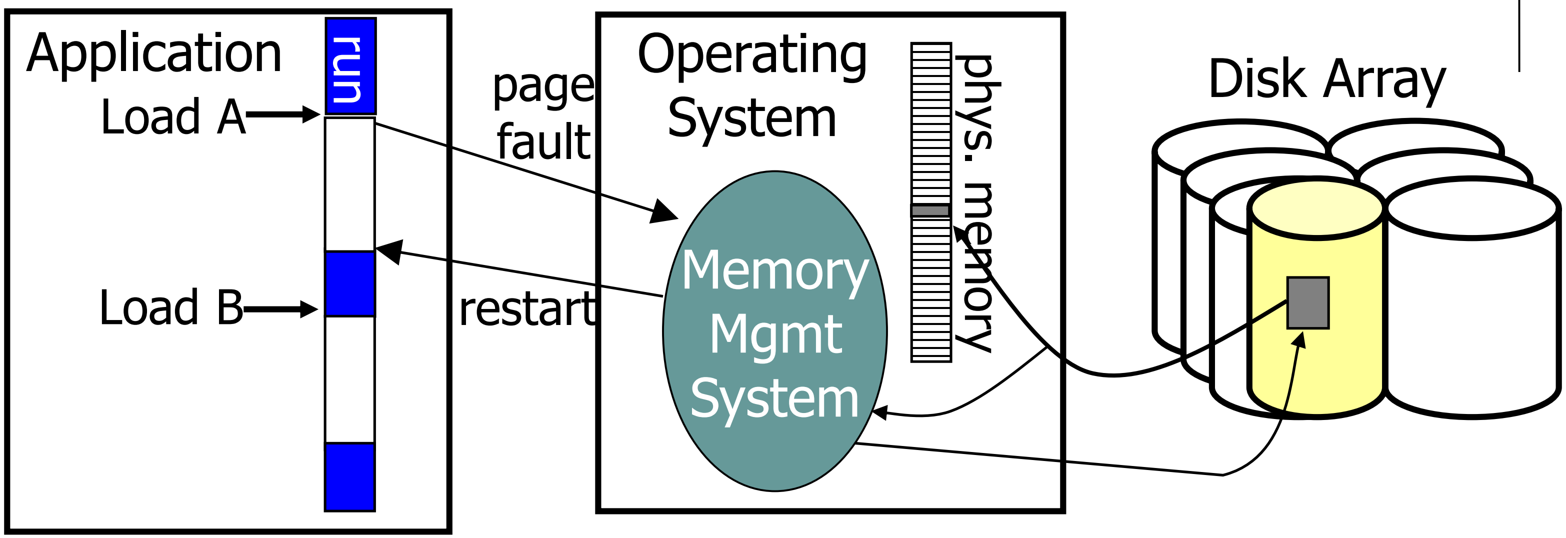
Page Faults

- What happens when a process accesses a page that has been evicted?
  1. When a process accesses the page, the invalid PTE will cause a trap (**page fault**)
  3. The trap will run the OS page fault handler
  4. Handler uses the invalid PTE to locate page in swap file
  5. Reads page into a physical frame, updates PTE to point to it
  6. Restarts process

Policy Decisions

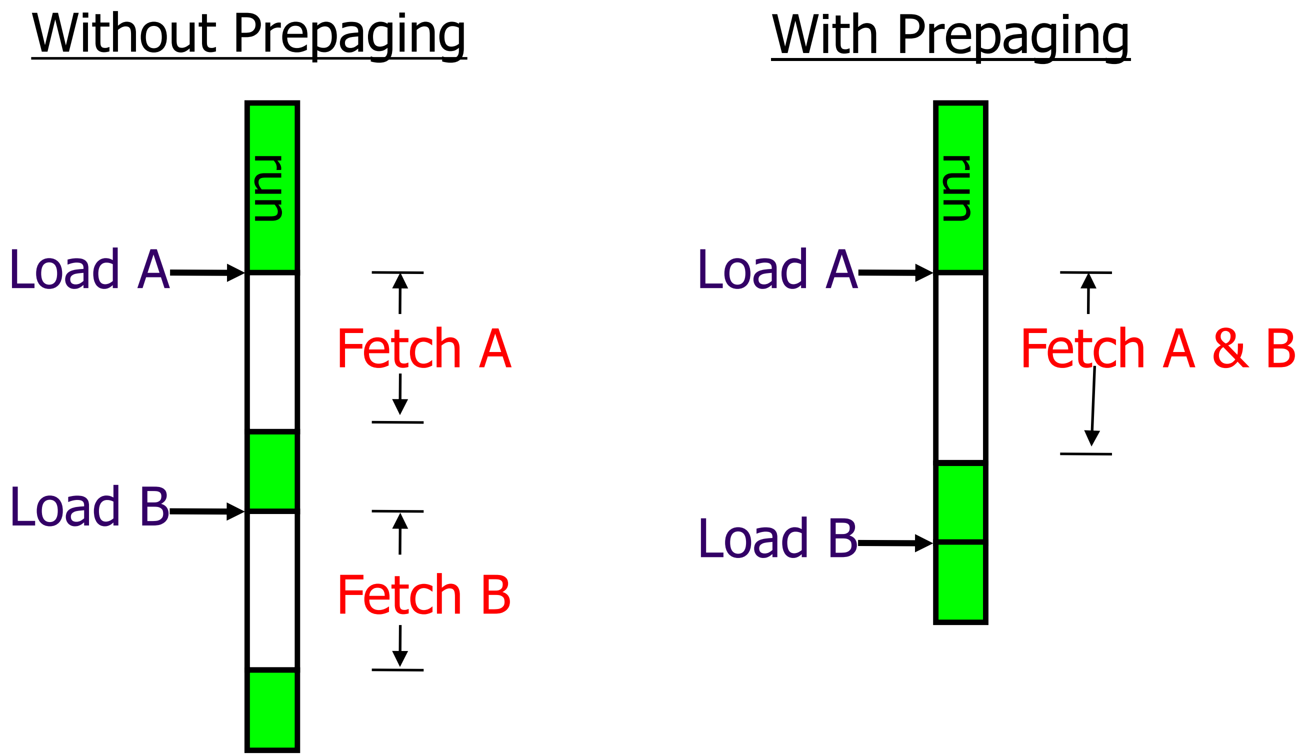
- Page tables, MMU, TLB, etc. are *mechanisms* that make virtual memory possible
- Next, we'll look at *policies* for virtual memory management:
  - Fetch Policy – *when* to fetch a page
  - Placement Policy – *where* to put the page
  - Replacement Policy – *what* page to evict to make room?

Demand Paging



- Timing: Disk read is initiated *when the process needs the page*
- Request size: Process can only page fault on one page at a time, disk sees single page-sized read
- What alternative do we have?

Prepaging (aka Prefetching)



- Predict future page use at time of current fault
  - On what should we base the prediction? What if it's wrong?

Policy Decisions

- Page tables, MMU, TLB, etc. are *mechanisms* that make virtual memory possible
- Next, we'll look at *policies* for virtual memory management:
  - Fetch Policy – *when* to fetch a page
    - Demand paging vs. Prepaging
  - Placement Policy – *where* to put the page
    - Are some physical pages preferable to others?
  - Replacement Policy – *what* page to evict to make room?
    - Lots and lots of possible algorithms!

Placement Policy

- In paging systems, memory management hardware can translate any virtual-to-physical mapping equally well
- Why would we prefer some mappings over others?
  - NUMA (non-uniform memory access) multiprocessors
    - Any processor can access entire memory, but local memory is faster
  - Cache performance
    - Choose physical pages to minimize cache conflicts
- These are active research areas!

Policy Decisions

- Page tables, MMU, TLB, etc. are *mechanisms* that make virtual memory possible
- Next, we'll look at *policies* for virtual memory management:
  - Fetch Policy – *when* to fetch a page
    - Demand paging vs. Prepaging
  - Placement Policy – *where* to put the page
    - Are some physical pages preferable to others?
  - ➡ Replacement Policy – *what* page to evict to make room?
    - Lots and lots of possible algorithms!

Evicting the best page

- The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove
- Replacement algorithms are evaluated on a *reference string* by counting the number of page faults

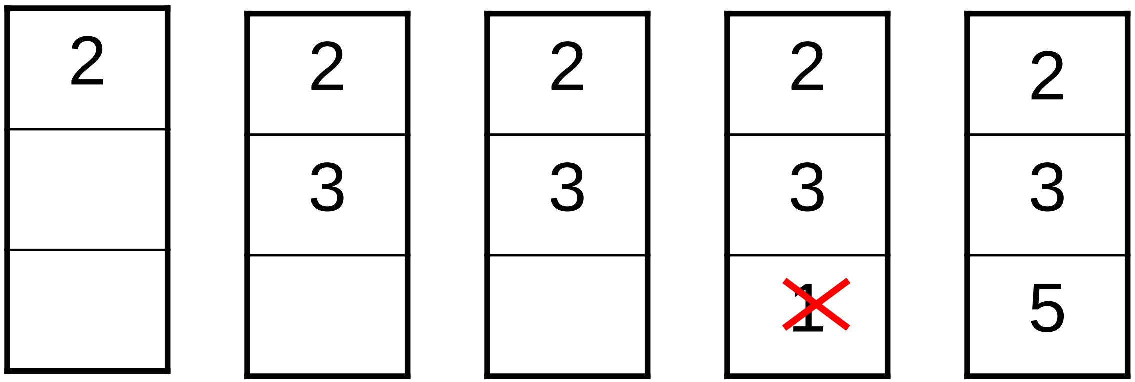
- Let's start by cheating a little bit …
  - Assume we know the reference string – what is the best replacement policy in this case?

Evicting the best page

- Page address list: 2,3,2,1,5,4,5,3,5,3,2

Cold misses: first access to a page (unavoidable)

Capacity misses: caused by replacement due to limited size of memory



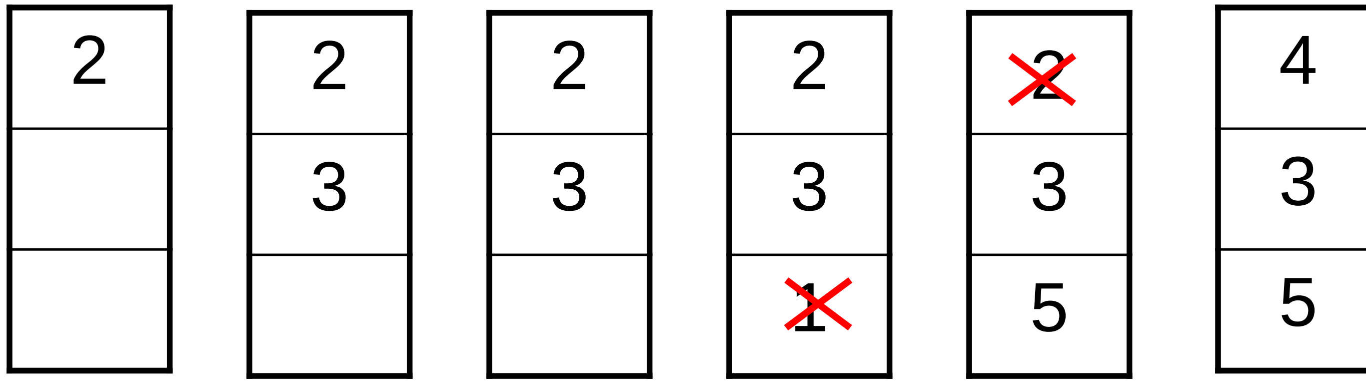
- Lesson 1:
  - The best page to evict is the one never used again
  - Will never fault on it

Evicting the best page

- Page address list: 2,3,2,1,5,4,5,3,5,3,2

Cold misses: first access to a page (unavoidable)

Capacity misses: caused by replacement due to limited size of memory



- Lesson 2:
  - Never is a long time, so picking the page closest to "never" is the next best thing
  - Evicting the page that won't be used for the longest period of time minimizes the number of page faults
  - Proved by Belady, 1966

Belady's Algorithm

- Belady's algorithm is known as the *optimal* page replacement algorithm because it has the lowest fault rate for any page reference stream (aka OPT or MIN)
  - Idea: Replace the page that will not be used for the longest period of time
  - Problem: Have to know the future perfectly
- Why is Belady's useful then? Use it as a yardstick
  - Compare implementations of page replacement algorithms with the optimal to gauge room for improvement
  - If optimal is not much better, then algorithm is pretty good
  - If optimal is much better, then algorithm could use some work
    - Random replacement is often the lower bound

What are possible replacement algorithms?

- First-in-first-out (FIFO)
- Least-recently-used (LRU)
- Least-frequently-used
- Most-frequently-used

•Many of these require book-keeping …
•Let’s start with algorithms that require only information contained in PTE

Page Table Entries (PTE)

1	1	1	3	26
M	R	V	Prot	Page Frame Number

- *Modify* (M)
  - says whether or not page has been written
- *Reference* (R)
  - says whether page has been accessed
  - is cleared periodically (e.g. at clock interrupt)
- *Valid* (V)
  - says whether PTE can be used
- Protection bits:
  - what operations are allowed on page

Not-Recently-Used (NRU)

Divide pages into 4 classes:

- *Class1:* Not referenced, not modified
- *Class 2:* Not referenced, modified
- *Class 3:* Referenced, not modified
- *Class 4:* Referenced, modified

- Remove page at random from lowest-numbered class that’s not empty

First-In First-Out (FIFO)

- FIFO is an obvious algorithm and simple to implement
  - Maintain a list of pages in order in which they were paged in
  - On replacement, evict the one brought in longest time ago
- Why might this be good?
  - Maybe the one brought in the longest ago is not being used
- Why might this be bad?
  - Then again, maybe it's not
  - We don't have any info to say one way or the other
- FIFO suffers from "Belady's Anomaly"
  - The fault rate might actually increase when the algorithm is given more memory (**very bad**)

Example of Belady's anomaly

- Page Address List: 0,1,2,3,0,1,4,0,1,2,3,4

	0	1	2	3	0	1	4	0	1	2	3	4
Youngest	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Oldest			0	1	2	3	0	0	0	1	4	4

3 frames, 9 faults

Anomaly	0	1	2	3	0	1	4	0	1	2	3	4
Youngest	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Oldest				0	0	0	1	2	3	4	0	1

4 frames, 10 faults

Second-Chance

- Idea:
  - FIFO (First-in-first-out) considers only age
  - NRU (Not recently used) considers only usage
  - Maybe we should combine the two!

- Second chance algorithm:
  - Don’t evict the oldest page if it has been used.
  - Evict the oldest page that has not been used.

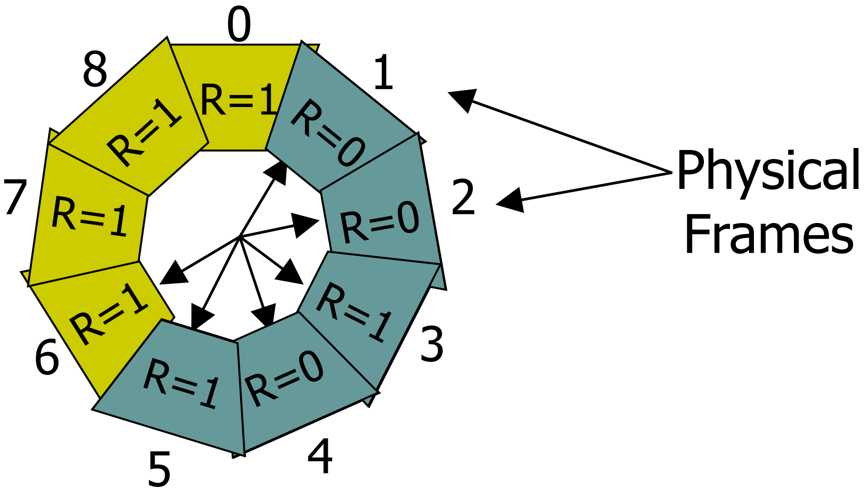
- Pages that are used often enough to keep reference bits set will not be replaced

Implementing Second Chance (clock)

Replace page that is "old enough"

- Arrange all of physical page frames in a big circle (clock)
- A clock hand is used to select a good LRU candidate
  - Sweep through the pages in circular order like a clock
  - If the ref bit (aka use bit) is off, it hasn't been used recently
    - Evict the page
  - If the ref bit is on
    - Turn it off and go to next page
- Arm moves quickly when pages are needed
- Low overhead when plenty of memory

Modelling Clock



- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

Least Recently Used (LRU)

- LRU uses reference information to make a more informed replacement decision
  - Idea: We can't predict the future, but we can make a guess based upon past experience
  - On replacement, evict the page that has not been used for the longest time in the past (Belady's: future)
  - When does LRU do well? When does LRU do poorly?
- On average performs very well (close to Belady)
  - But ….

Implementing Exact LRU

- Option 1:
  - Time stamp every reference
  - Evict page with oldest time stamp
  - Problems:
    - Need to make PTE large enough to hold meaningful time stamp (may double size of page tables, TLBs)
    - Need to examine every page on eviction to find one with oldest time stamp
- Option 2:
  - Keep pages in a stack. On reference, move the page to the top of the stack. On eviction, replace page at bottom.
  - Problems:
    - Need costly software operation to manipulate stack on EVERY memory reference!

Modelling Exact LRU

- Page Address List: 0,1,2,3,0,1,4,0,1,2,3,4

3 Frames	0	1	2	3	0	1	4	0	1	2	3	4
MRU page	0	1	2	3	0	1	4	0	1	2	3	4
		0	1	2	3	0	1	4	0	1	2	3
LRU page			0	1	2	3	0	1	4	0	1	2

10 faults

4 Frames	0	1	2	3	0	1	4	0	1	2	3	4
MRU page	0	1	2	3	0	1	4	0	1	2	3	4
		0	1	2	3	0	1	4	0	1	2	3
			0	1	2	3	0	1	4	0	1	2
LRU page				0	1	2	3	3	3	4	0	1

8 faults

Approximating LRU

- Exact LRU is too costly to implement
- LRU approximations use the PTE *reference* bit
- Basic Idea:
  - Initially, all R bits are zero; as processes execute, bits are set to 1 for pages that are used
  - Periodically examine the R bits – we do not know *order* of use, but we know pages that were (or were not) used
- Additional-Reference-Bits Algorithm
  - Keep a counter for each page
  - At regular intervals, for every page do:
    - Shift R bit into high bit of counter register
    - Shift other bits to the right
    - Pages with “larger” counters were used more recently

Counting-based Replacement

- Count number of uses of a page
- Least-Frequently-Used (LFU)
  - Replace the page used least often
  - Pages that are heavily used at one time tend to stick around even when not needed anymore
  - Newly allocated pages haven’t had a chance to be used much
- Most-Frequently-Used (MFU)
  - Favours new pages
- Neither is common, both are poor approximations of OPT

What are possible replacement algorithms?

	Ease of implementation	Performance
• Not-recently-used (NRU)	+	
• First-in-first-out (FIFO)	+	
• Least-recently-used (LRU)	-	
• Least-frequently-used	-	
• Most-frequently-used	-	

What are possible replacement algorithms?

	Ease of implementation	Performance
• Not-recently-used (NRU)	+	~
• First-in-first-out (FIFO)	+	~
• Least-recently-used (LRU)	-	++
• Least-frequently-used	-	-
• Most-frequently-used	-	-

What are possible replacement algorithms?

	Ease of implementation	Performance
• Not-recently-used (NRU)	+	~
• First-in-first-out (FIFO)	+	~
• Least-recently-used (LRU)	-	++
• Least-frequently-used	-	-
• Most-frequently-used	-	-
• Second chance	+	+

Fixed vs. Variable Space

- In a multiprogramming system, we need a way to allocate memory to competing processes
- Problem: How to determine how much memory to give to each process?
  - Fixed space algorithms
    - Each process is given a limit of pages it can use
    - When it reaches the limit, it replaces from its own pages
    - Local replacement
      - Some processes may do well while others suffer
  - Variable space algorithms
    - Process' set of pages grows and shrinks dynamically
    - Global replacement - one process can ruin it for the rest
    - Local replacement - replacement and set size are separate

Working Set Model

- How do you decide how large the fixed or variable space for a process should be?
- Depends on access pattern …

Process 1

6 1 5 2 1 6 2 7 5 1

5 pages
{1,2,5,6,7}

Process 2

4 4 3 3 4 1 3 4 4 4

2 or 3 pages
{3,4} or {1,3,4}

Working Set Model

- A working set of a process is used to model the dynamic locality of its memory usage
  - Defined by Peter Denning in 60's
- Definition
  - WS(t,Δ) = {pages P such that P was referenced in the time interval (t, t-Δ)}
  - t = time, Δ = working set window (measured in page refs)
- A page is in the working set (WS) only if it was referenced in the last Δ references

…2 **6 1 5 2 1 6 7 5 1** 6   1 2 3 **4 4 4 3 4 3 4 4 4** 1

Δ   Δ

WS(t1) = {1,2,5,6,7}   t1   WS(t2) = {3,4}   t2

Working Set Size

- The working set size is the number of pages in the working set
  - The number of pages referenced in the interval (t, t-Δ)
- The working set size changes with program locality
  - During periods of poor locality, you reference more pages
  - Within that period of time, the working set size is larger
- Intuitively, want the working set to be the set of pages a process needs in memory to prevent heavy faulting
  - Each process has a parameter Δ that determines a working set with few faults
  - Denning: Don't run a process unless working set is in memory

Working Set Problems

- Problems
  - How do we determine Δ?
  - How do we know when the working set changes?
- Too hard to answer
  - So, working set is not used in practice as a page replacement algorithm
- However, it is still used as an abstraction
  - The intuition is still valid
  - When people ask, “How much memory does Netscape need?”, they are in effect asking for the size of Netscape’s working set

Page Fault Frequency (PFF)

- Page Fault Frequency (PFF) is a variable space algorithm that uses a more ad-hoc approach
  - Monitor the fault rate for each process
  - If the fault rate is above a high threshold, give it more memory
    - So that it faults less
    - But not always (FIFO, Belady's Anomaly)
  - If the fault rate is below a low threshold, take away memory
    - Should fault more
    - But not always
- Hard to use PFF to distinguish between changes in locality and changes in size of working set

Thrashing

- Page replacement algorithms avoid thrashing
  - When more time is spent by the OS in paging data back and forth from disk than executing user programs
  - No time spent doing useful work (making progress)
  - In this situation, the system is overcommitted
    - No idea which pages should be in memory to reduce faults
    - Could just be that there isn't enough physical memory for all of the processes in the system
    - Ex: Running Windows Vista with 4 MB of memory…
  - Possible solutions
    - Swapping – write out all pages of a process and suspend it
    - Buy more memory

Windows XP Paging Policy

- Local page replacement
  - Per-process FIFO
  - Pages are stolen from processes using more than their minimum working set
  - Processes start with a default of 50 pages
  - XP monitors page fault rate and adjusts working-set size accordingly
  - On page fault, *cluster* of pages around the missing page are brought into memory

Linux Paging

- Global replacement, like most Unix
- Modified second-chance clock algorithm
  - Pages ***age*** with each pass of the clock hand
  - Pages that are not used for a long time will eventually have a value of zero
- Continually under development…