Hello!

MEMORY: SWAPPING

Shivaram Venkataraman

CS 537, Spring 2020

ADMINISTRIVIA

Project 2b is out. Due Feb 24th, 10pm

Project 2a is done ?!?

Shivaram upcoming travel

- No class on Feb 27. Guest lecture March 3
- Discussion
  - No discussion Feb 20, Feb 27
  - **Discussion on Tue Feb 25 at 5.30pm**
- Video about how to use GDB

OFFICE HOURS

1. One question per student at a time → Have a version of code
2. Please be prepared before asking questions
3. The TAs might not be able to fix your problem
4. Limited time per student

Search Piazza?

AGENDA / LEARNING OUTCOMES

Memory virtualization

- How we support virtual mem larger than physical mem?
- What are mechanisms and policies for this?

RECAP

COMBINE PAGING AND SEGMENTATION

Divide address space into segments (code, heap, stack)

- Segments can be variable length

Divide each segment into fixed-sized pages

Logical address divided into three portions

Page table per segment

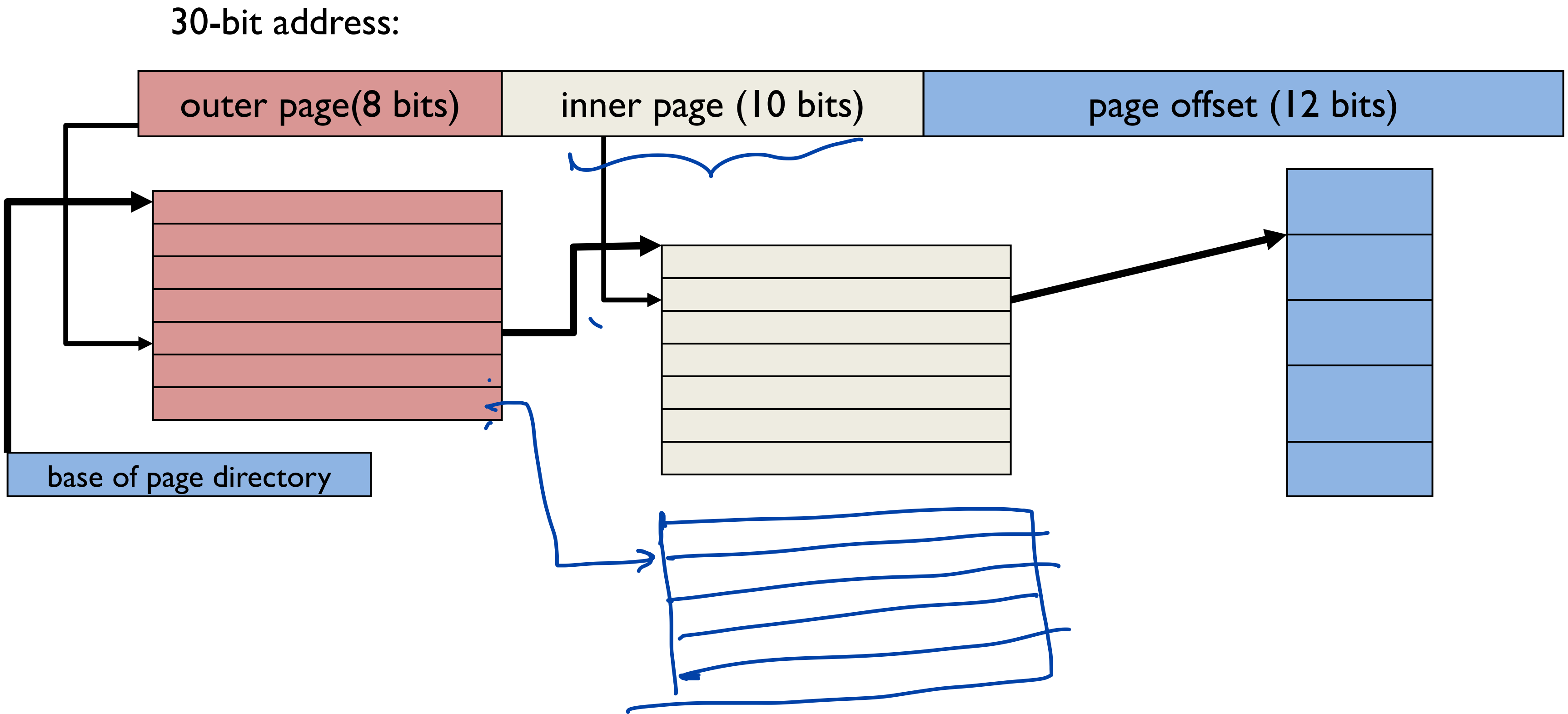
↳ space reduction

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

Implementation

- Each segment has a page table
- Each segment track base (physical address) and bounds of the **page table**

MULTILEVEL PAGE TABLES



SUMMARY: BETTER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

↳ Per process

Many options for efficiently organizing page tables

If OS traps on TLB miss, OS can use any data structure

- Inverted page tables (hashing)

If Hardware handles TLB miss, page tables must follow specific format

- Multi-level page tables used in x86 architecture
- Each page table fits within a page → Page size

Page Table entry size

QUIZ 13

| 6 bits | offset

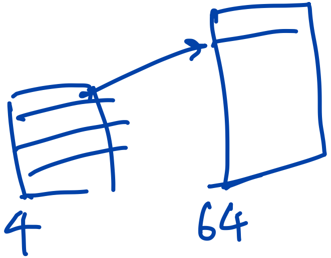
2 bits

https://tinyurl.com/cs537-sp20-quiz13

16-bit

- 256 byte page size
- offset: 8 bits

= 2 Page Table + 1 mem



Every mem access

1 Page Table

1 mem location

VPN	PPN
0x3F	< ?
0x53	< >
0x76	< >
0x40	

Virtual Addresses

0x3FF8: load 0x5320, %eax

0x3FFC: load 0x7640, %ebx

0x4000: mul %ecx, %eax, %ebx

0x4004: store %ebx, 0x5324

0x4008: load 0x5328, %ebx

Linear PT(no TLB)	Linear PT,5-entry TLB	2-level page table, 5-entry TLB
2 + 2	2 + 2	3 + 3
2 + 2	1 +2	1 + 3
2	2	3
4	1 + 1	1 + 1
4	1 + 1	1 + 1
18	13	17

SWAPPING

MOTIVATION

OS goal: Support processes when not enough physical memory

- Single process with very large address space
- Multiple processes with combined address spaces

User code should be independent of amount of physical memory

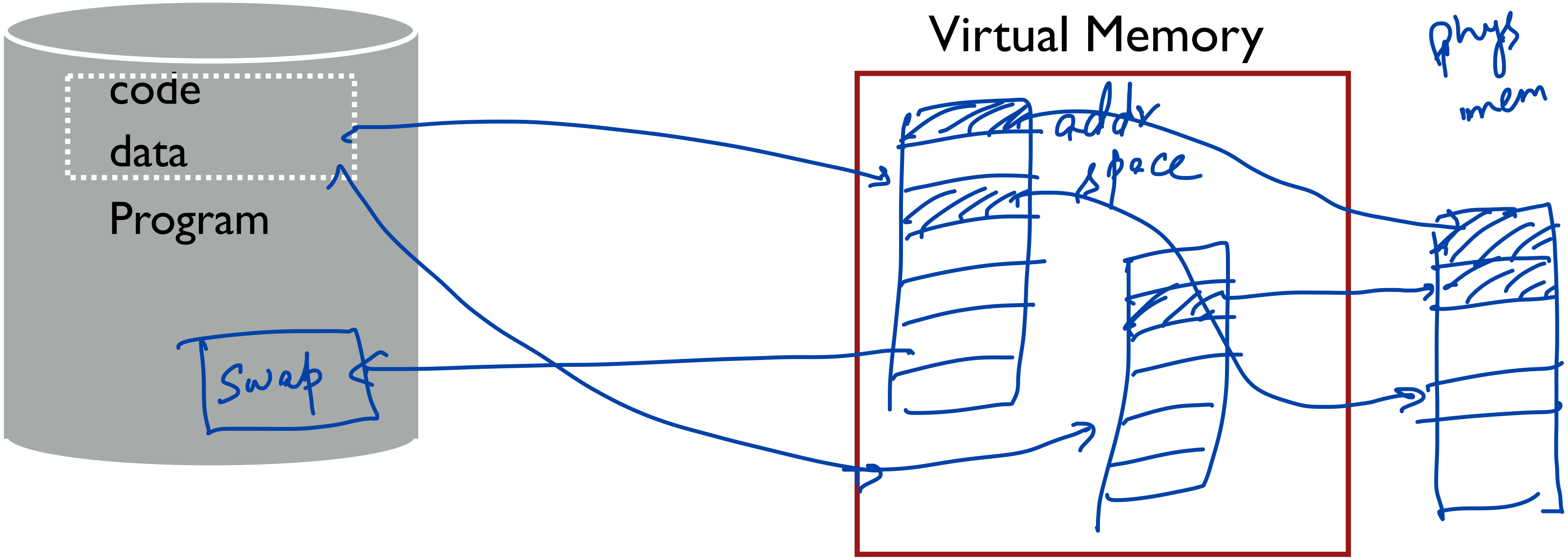
- Correctness, if not performance

Virtual memory: OS provides illusion of more physical memory

Why does this work?

- Relies on key properties of user processes (workload) →
  and machine architecture (hardware)

→ larger capacity



code

data

Program

Virtual Memory

LOCALITY OF REFERENCE → Workload

TLB hit rate

Leverage locality of reference within processes

- Spatial: reference memory addresses **near** previously referenced addresses
- Temporal: reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
  - Estimate: 90% of time in 10% of code

Implication:

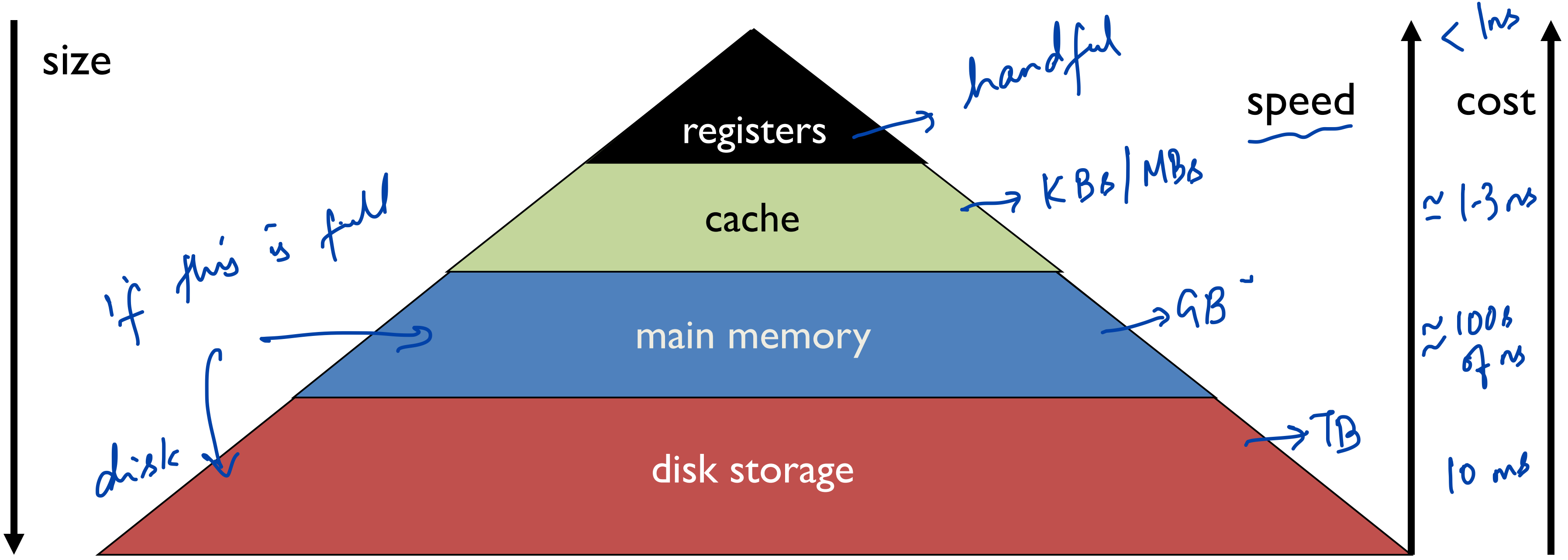
- Process only uses small amount of address space at any moment
- Only small amount of address space must be resident in physical memory

Working set

MEMORY HIERARCHY

Leverage memory hierarchy of machine architecture

Each layer acts as "backing store" for layer above



SWAPPING INTUITION

Idea: OS keeps unreferenced pages on disk

- Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory

OS and hardware cooperate to make large disk seem like memory

- Same behavior as if all of address space in main memory

Requirements:

- OS must have **mechanism** to identify location of each page in address space → in memory or on disk
- OS must have **policy** to determine which pages live in memory and which on disk

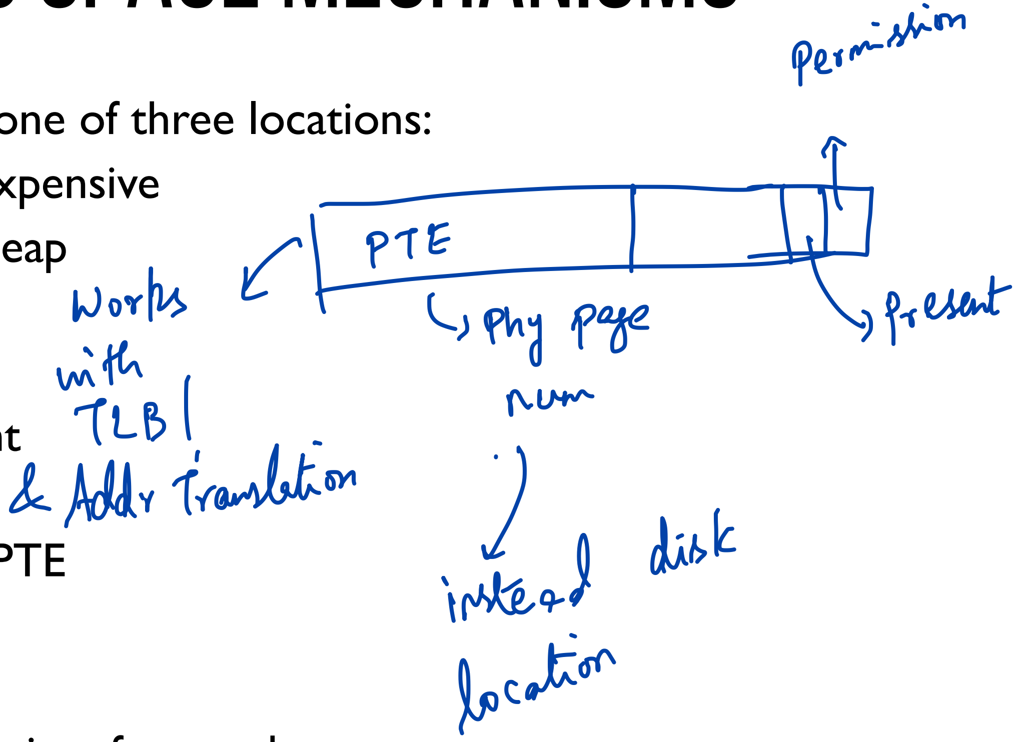
VIRTUAL ADDRESS SPACE MECHANISMS

Each page in virtual address space maps to one of three locations:

- Physical main memory: Small, fast, expensive
- Disk (backing store): Large, slow, cheap
- Nothing (error): Free

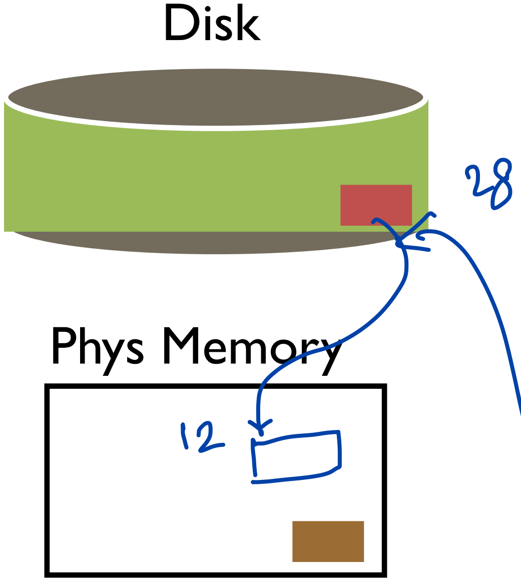
Extend page tables with an extra bit: present

- permissions (r/w), valid, present
- Page in memory: present bit set in PTE
- Page on disk: present bit cleared
  - PTE points to block on disk
  - Causes trap into OS when page is referenced
  - Trap: page fault



reading a page
~10ms

Linear Page Table



0x0

PFN	valid	prot	present
10	1	r-x	1
-	0	-	-
23	1	rw-	0
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
~~28~~ 12	1	rw-	~~0~~ 1
4	1	rw-	1

prot → Permission

present → Is this page in phy mem?

Page fault
- ↳ fetch page
- ↳ set present bit
- ↳ retry addr translation

What if access vpn 0xb?

VIRTUAL MEMORY MECHANISMS

First, hardware checks TLB for virtual address

- if TLB hit, address translation is done; page in physical memory

Else ...

- Hardware or OS walk page tables
- If PTE designates page is present, then page in physical memory
  (i.e., present bit is cleared)

Else

- Trap into OS (not handled by hardware)
- OS selects victim page in memory to replace
  - Write victim page out to disk if modified (use dirty bit in PTE)
- OS reads referenced page from disk into memory
- Page table is updated, present bit is set
- Process continues execution

SWAPPING POLICIES

SWAPPING POLICIES

Goal: Minimize number of page faults

- Page faults require milliseconds to handle (reading from disk)
- Implication: Plenty of time for OS to make good decision

OS has two decisions

- Page selection

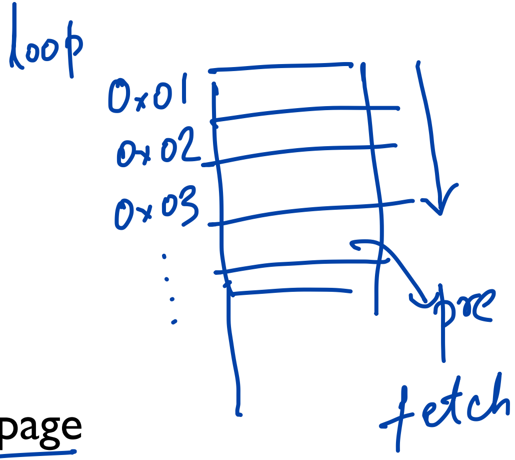
  When should a page (or pages) on disk be **brought into** memory?

- Page replacement

  Which resident page (or pages) in memory should be **thrown out** to disk?

When

PAGE SELECTION



Demand paging: Load page only when page fault occurs

- Intuition: Wait until page must absolutely be in memory
- When process starts: No pages are loaded in memory
- Problems: Pay cost of page fault for every newly accessed page

~ 10ms

Prepaging (anticipatory, prefetching): Load page before referenced

- OS predicts future accesses (oracle) and brings pages into memory early
- Works well for some access patterns (e.g., sequential)

Hints: Combine above with user-supplied hints about page references

- User specifies: may need page in future, don't need this page anymore, or sequential access pattern, ...
- Example: madvise() in Unix → Hint I may need this page!

PAGE REPLACEMENT

binary

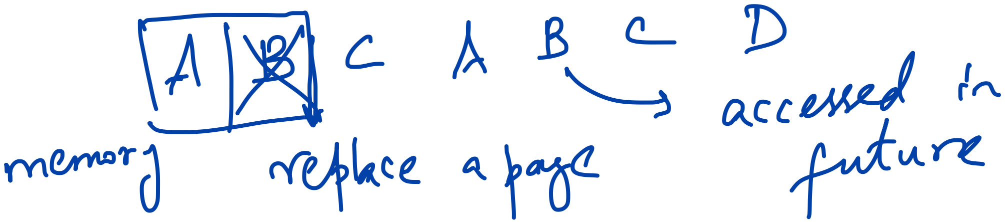
↓

Which page in main memory should selected as victim?

- Write out victim page to disk if modified (dirty bit set)
- If victim page is not modified (clean), just discard → Code page

OPT: Replace page not used for longest time in future

- Advantages: Guaranteed to minimize number of page faults
- Disadvantages: Requires that OS predict the future; Not practical, but good for comparison



PAGE REPLACEMENT

FIFO: Replace page that has been in memory the longest

- Intuition: First referenced long time ago, done with it now
- Advantages: Fair: All pages receive equal residency; Easy to implement
- Disadvantage: Some pages may always be needed

LRU: Least-recently-used: Replace page not used for longest time in past

- Intuition: Use past to predict the future
- Advantages: With locality, LRU approximates OPT
- Disadvantages:
  - Harder to implement, must track which pages have been accessed
  - Does not handle all workloads well

Temporal locality

PAGE REPLACEMENT

Three pages of physical memory

Page reference string:
DDBBACBDBD

time

Hit/ Miss		OPT : 4				FIFO = 6			LRU = 5			
M	D	D			M	D			D			M
H	D	D			H	D			D			H
M	B	D	B		M	D	B		D	B		M
H	B	D	B		H	D	B		D	B		H
M	A	D	B	A	M	D	B	A	D	B	A	M
M	C	D	B	C	M	C	B	A	C	B	A	M
H	B	D	B	C	H	C	B	A	C	B	A	H
H	D	D	B	C	M	C	D	A	C	B	D	M
H	B	D	B	C	M	C	D	B	C	B	D	H
H	D	D	B	C	H	C	D	B	C	B	D	H

Metric:

Miss count

↳ 4

Miss rate : 40%

QUIZ 14

https://tinyurl.com/cs537-sp20-quiz14

Page reference string: ABCABDADBCB

Metric: Miss count

Three pages of physical memory

			OPT				FIFO = 7			LRU : 5		
3		ABC	A	B	C	3	A	B	C	A	B	C
	H	A	A	B	C	H	A	B	C	A	B	C
	H	B	A	B	C	H	A	B	C	A	B	C
4		D	A	B	D	M	D	B	C	A	B	D
	H	A				M	D	A	C			
	H	D				H	D	A	C			
	H	B	A	B	D	M	D	A	B			
$\boxed{5}$	M	C	A	B	C	M	C	A	B			
	H	B				H						

PAGE REPLACEMENT COMPARISON

1GB → 4GB memory

Add more physical memory, what happens to performance?

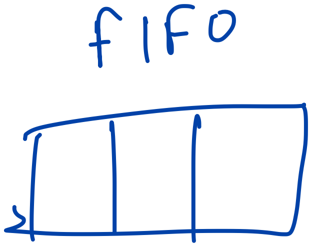
LRU, OPT:

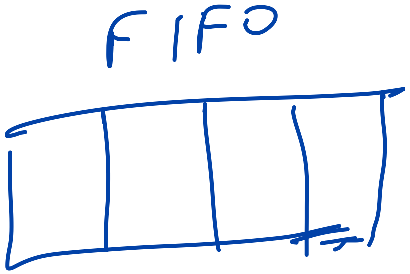
- Guaranteed to have fewer (or same number of) page faults
- Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
- Stack property: smaller cache always subset of bigger

FIFO:

- Usually have fewer page faults
- Belady's anomaly: May actually have more page faults!

FIFO PERFORMANCE MAY DECREASE!





Consider access stream: ABCDABEABCDE

Physical memory size: 3 pages vs. 4 pages

How many misses with FIFO?

Anamoly

IMPLEMENTING LRU

Software Perfect LRU

- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list
- When need victim: Pick page at back of list
- Trade-off: Slow on memory reference, fast on replacement

Hardware Perfect LRU

- Associate timestamp register with each page
- When page is referenced: Store system clock in register
- When need victim: Scan through registers to find oldest clock
- Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)

In practice, do not implement Perfect LRU

- LRU is an approximation anyway, so approximate more
- Goal: Find an old page, but not necessarily the very oldest

CLOCK ALGORITHM

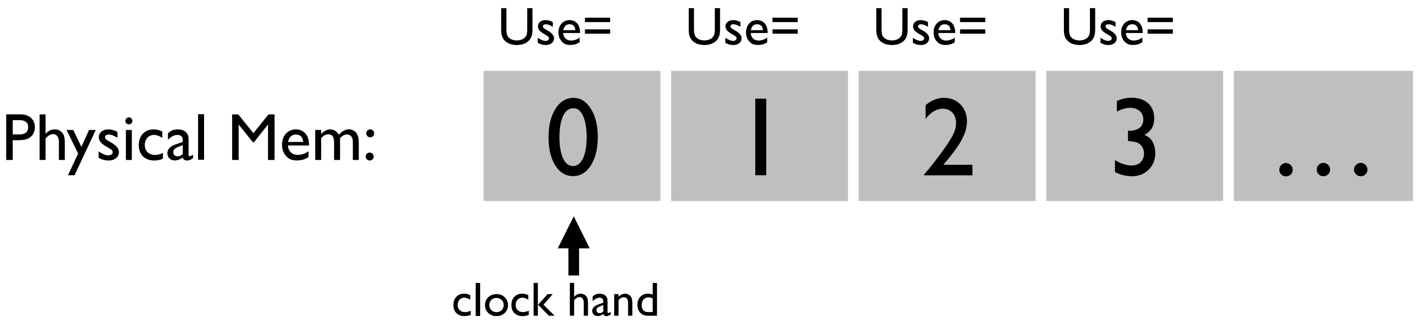
Hardware

- Keep use (or reference) bit for each page frame
- When page is referenced: set use bit

Operating System

- Page replacement: Look for page with use bit cleared (has not been referenced for awhile)
- Implementation:
  - Keep pointer to last examined page frame
  - Traverse pages in circular buffer
  - Clear use bits as search
  - Stop when find page with already cleared use bit, replace this page

CLOCK: LOOK FOR A PAGE



CLOCK EXTENSIONS

Replace multiple pages at once

- Intuition: Expensive to run replacement algorithm and to write single block to disk
- Find multiple victims each time and track free list

Use dirty bit to give preference to dirty pages

- Intuition: More expensive to replace dirty pages
  Dirty pages must be written to disk, clean pages do not
- Replace pages that have use bit and dirty bit cleared

SUMMARY: VIRTUAL MEMORY

Abstraction: Virtual address space with code, heap, stack

Address translation

- Contiguous memory: base, bounds, segmentation
- Using fixed sizes pages with page tables

Challenges with paging

- Extra memory references: avoid with TLB
- Page table size: avoid with multi-level paging, inverted page tables etc.

Larger address spaces: Swapping mechanisms, policies (LRU, Clock)

NEXT STEPS

Project 2b: Out now

Next class: New module on Concurrency