

# MEMORY: SWAPPING

Shivaram Venkataraman

CS 537, Spring 2023

# ADMINISTRIVIA

Project 3 was due **Monday**.

Project 4: Scheduling new dates: Feb 22<sup>nd</sup> to March 6<sup>th</sup>

Midterm I: In class midterm, Multiple choice.

No notes / calculators. (We will give a table of powers of 2)

Old exams on Canvas

Discussion: Practice problems

# OFFICE HOURS

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

1. Increase number of TAs close to deadline
2. Study groups

# AGENDA / LEARNING OUTCOMES

## Memory virtualization

How we support virtual mem larger than physical mem?

What are mechanisms and policies for this?

**RECAP**

# MULTILEVEL, INVERTED PAGE TABLES

PPN VPN Prot

PPN	VPN	Prot

Multi-level Page Table

PDBR

	valid	PFN
PFN 200	1	201
	0	-
	0	-
	1	204

The Page Directory

	valid	prot	PFN
PFN 201	1	rx	12
	1	rx	13
	0	-	-
	1	rw	100

[Page 1 of PT: Not Allocated]

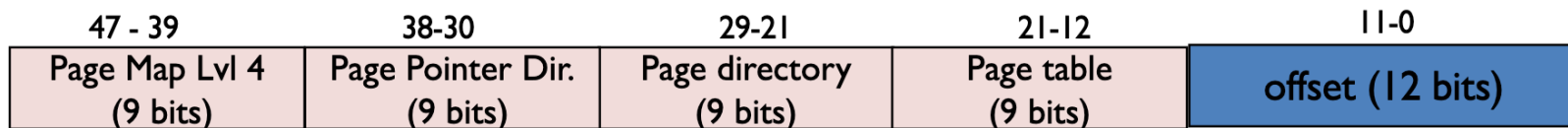
[Page 2 of PT: Not Allocated]

	0	-	-
	0	-	-
PFN 204	1	rw	86
	1	rw	15

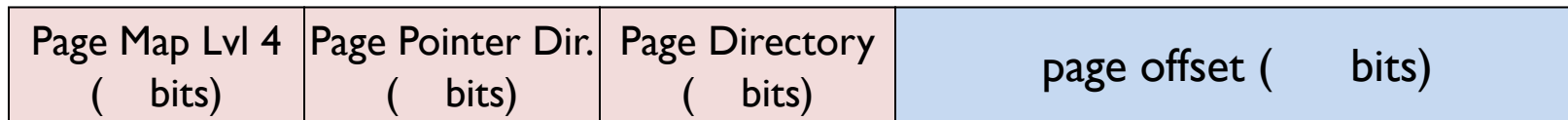
# TRANSLATING LARGE PAGES

HugePages saves TLB entries. But how does it affect page translation?

4KB pages: 4 levels → 4 memory accesses



2MB pages:



# SUMMARY: BETTER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

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 inner page table fits within a page

Large pages can reduce TLB use and number of accesses for translation



**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)

# WORKLOAD PROPERTIES

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

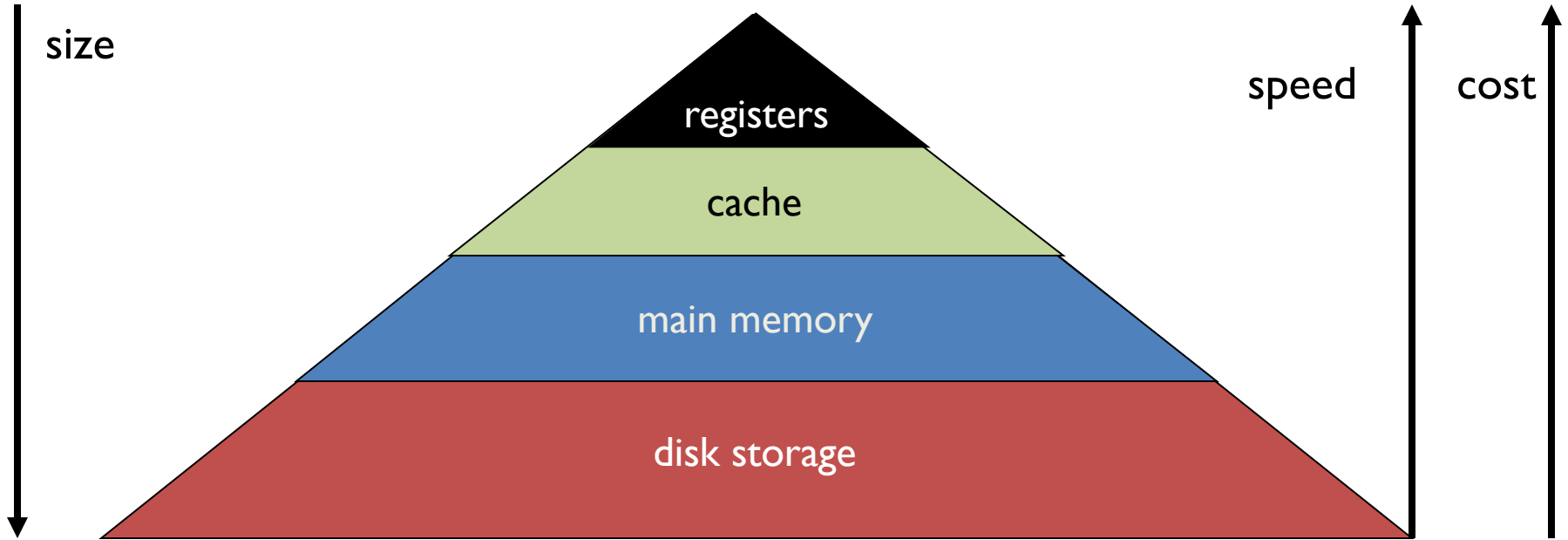
Implication:

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

# HARDWARE: 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**

Disk



Phys Memory



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

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?

# 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

**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

# PAGE REPLACEMENT

Which page in main memory should be selected as victim?

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

**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

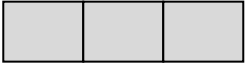
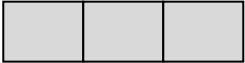
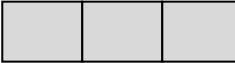















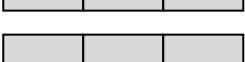
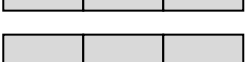
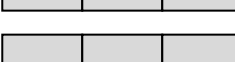



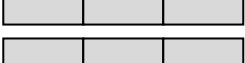
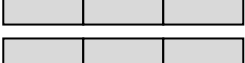
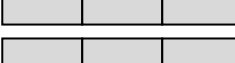






- Intuition: First referenced long time ago, done with it now
- Advantages: Fair: All pages receive equal residency; Easy to implement
- Disadvantage: Some pages may always be needed

**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

Three pages  
of physical  
memory

# PAGE REPLACEMENT

	OPT	FIFO	LRU
			
D			
D			
B			
B			
A			
A			
C			
C			
B			
B			
D			
D			

Page reference string:  
DDBBACBDBD

Metric:  
Miss count

# QUIZ 13

<https://tinyurl.com/cs537-sp23-quiz13>

Page reference string: ABCABDADBCB

		OPT	FIFO	LRU
Metric:	ABC	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>
Miss count	A	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>
	B	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>
Three pages	D	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>
of physical	A	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>
memory	D	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>
	B	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>
	C	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>
	B	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>	<input type="text"/> <input type="text"/> <input type="text"/>

# PAGE REPLACEMENT COMPARISON

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?

# 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

LRU is an approximation anyway, so approximate more?

# CLOCK ALGORITHM

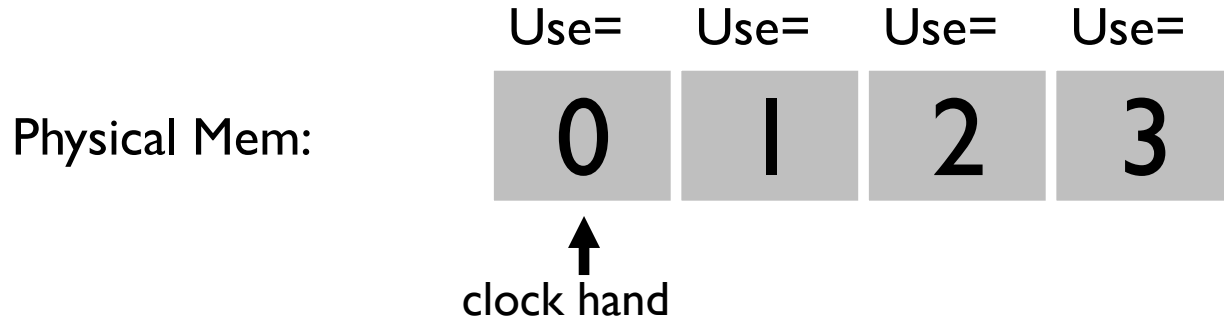
## Hardware

- 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



Use = 1,1,0,1 to begin

# 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

Next class: New module on Concurrency!