

MEMORY: SWAPPING

Andrea Arpaci-Dusseau
CS 537, Fall 2019

ADMINISTRIVIA

- Project 3 Due Thursday
 - Test cases now available
- Midterm I: Next Thursday
 - Review in Discussion tomorrow
 - Posted sample midterms
- Lecture Schedule
 - Today: Finish VM
 - Thursday: Start Concurrency (could be on midterm I)
 - Tuesday: Review for Midterm I
 - Thursday: Locks (NOT on midterm I)

AGENDA / LEARNING OUTCOMES

Memory virtualization

How we support virtual mem larger than physical mem?

What are mechanisms and policies for this?

PAGING TRANSLATION STEPS: TLBS

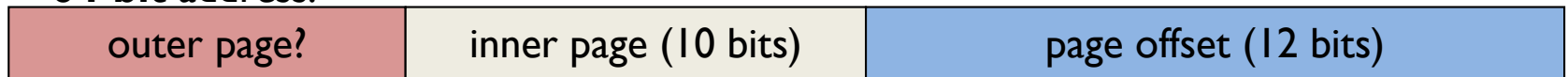
For each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. check TLB for **VPN**
 if TLB Miss:
 3. calculate addr of **PTE** (page table entry)
 4. read **PTE** from memory (repeat for each level of page table)
 5. replace some entry in TLB
6. extract **PFN** from TLB (page frame num)
7. build **PA** (phys addr)
8. read contents of **PA** from memory

PROBLEM WITH 2 LEVELS?

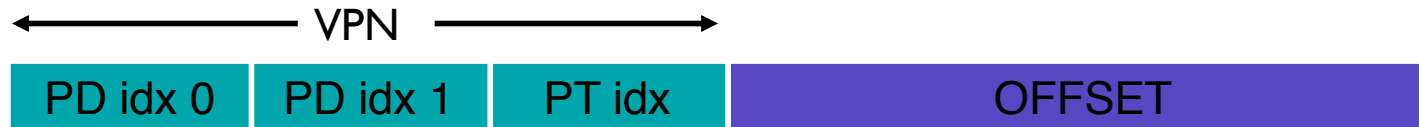
Problem: page directories (outer level) may not fit in a page

64-bit address:



Solution:

- Split page directories into pieces
- Use another page dir to refer to the page dir pieces.



Assume 4 KB pages, 4 byte PTEs, (each page table fits in page)

4KB / 4 bytes → 1K entries per level
→ 10 bits in virtual address for each level

SWAPPING

MOTIVATION

OS goal: Support processes when not enough physical memory

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

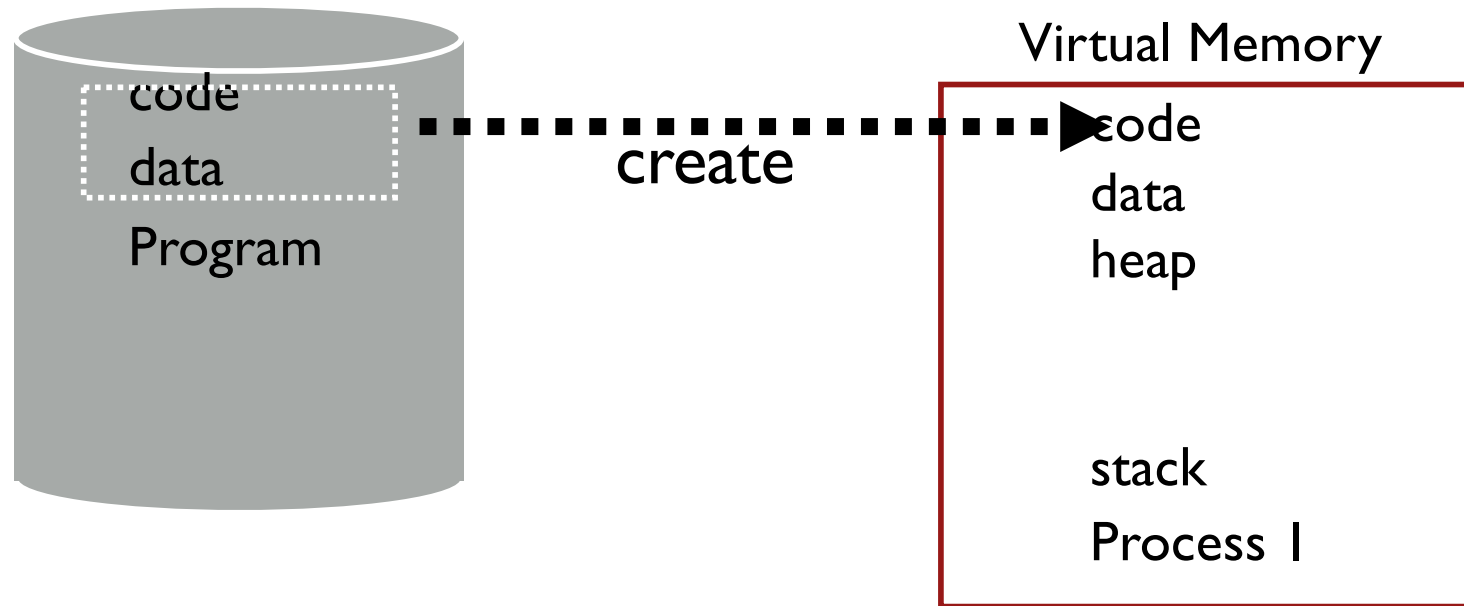
User code should be independent of amount of physical memory

- Correctness, if not performance

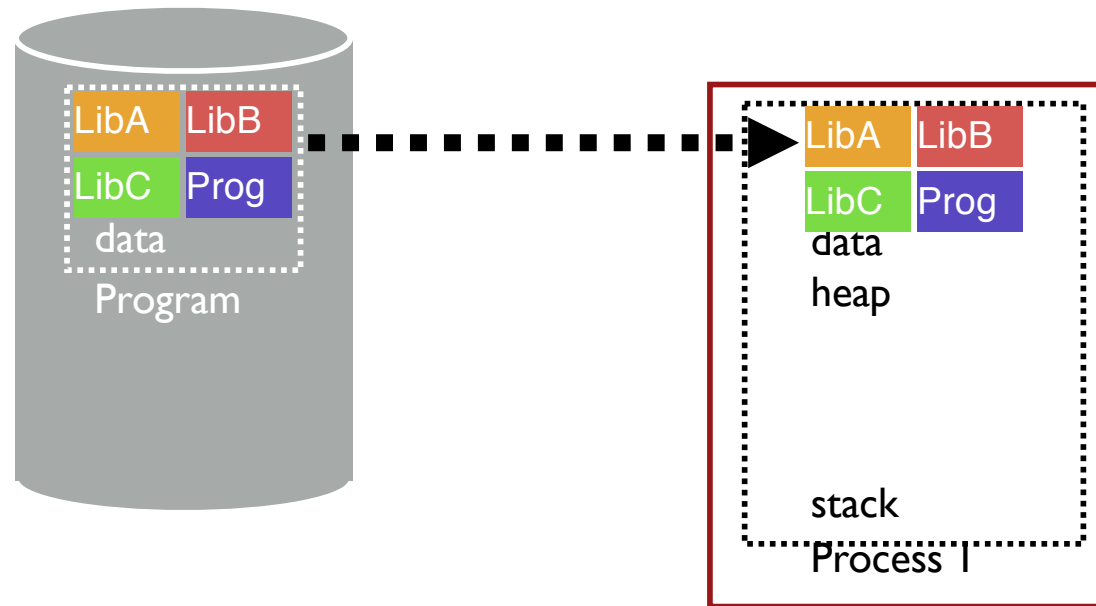
Virtual memory: OS provides illusion of more physical memory

Why does this work?

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

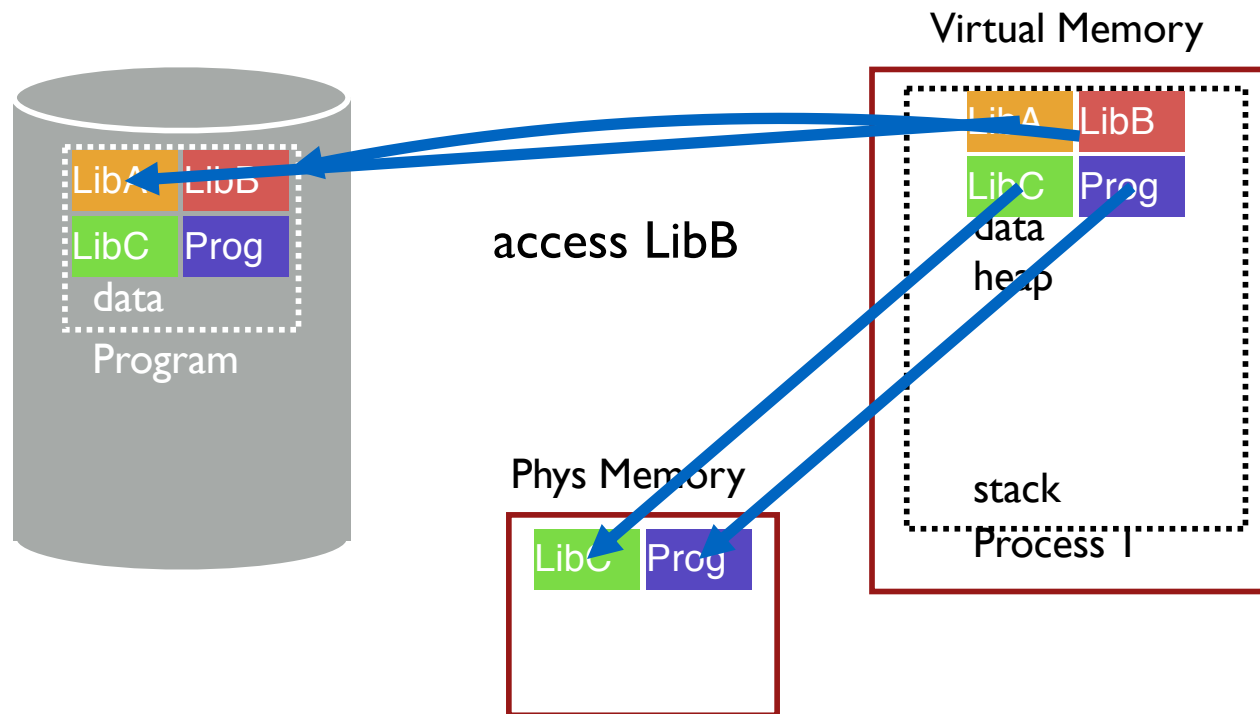


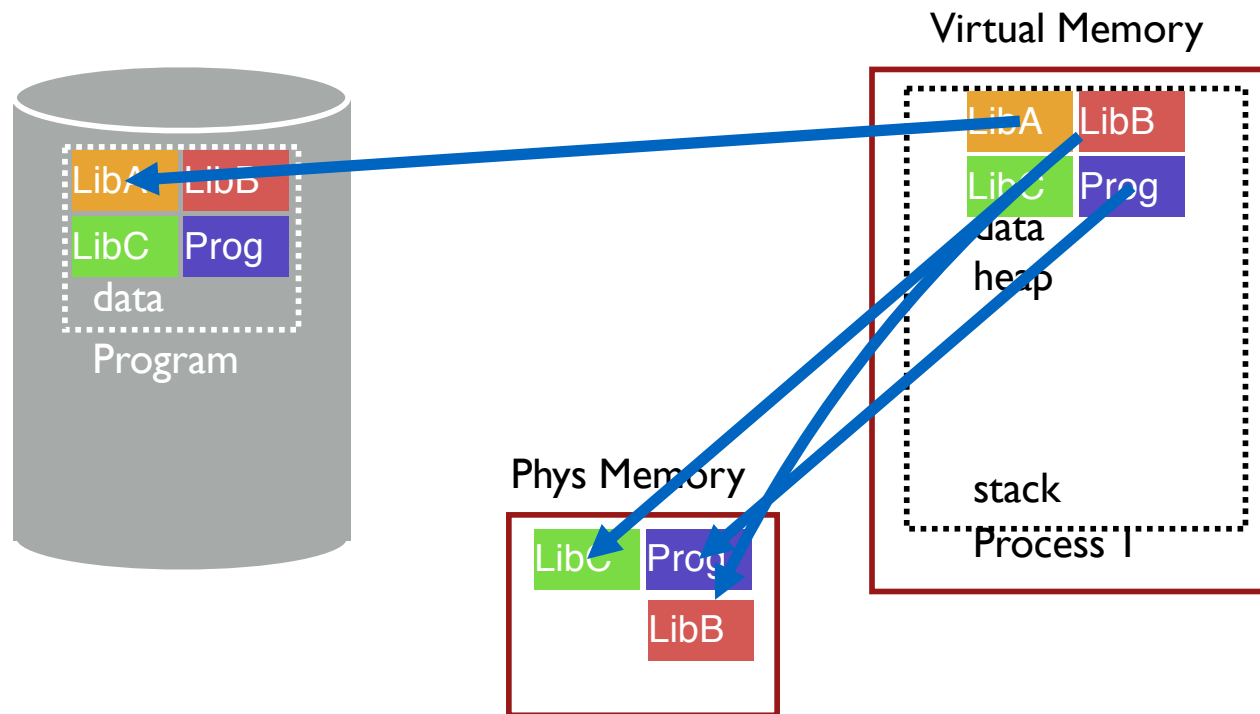
what's in code?



Code: many large libraries, some of which are rarely/never used

How to avoid wasting physical pages to store rarely used virtual pages?





copy (or move) to RAM

Called “**paging**” in

LOCALITY OF REFERENCE

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

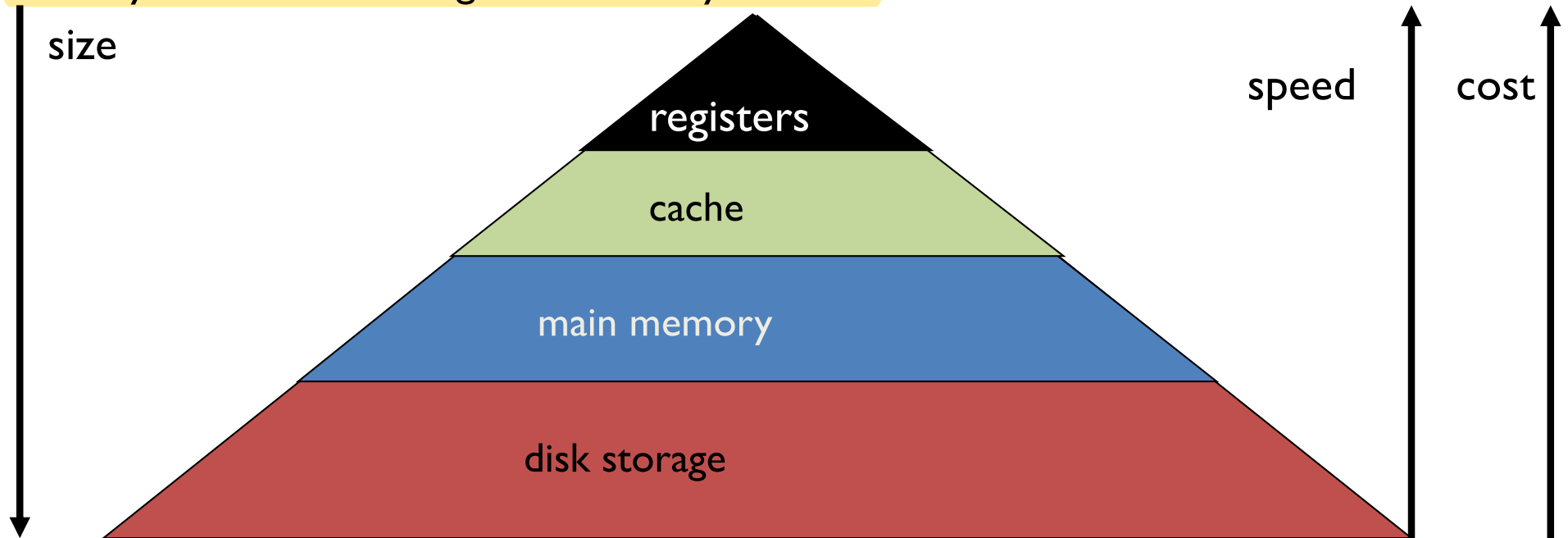
Implication:

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

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** for determining which pages live in memory and which on disk



SWAPPING MECHANISMS

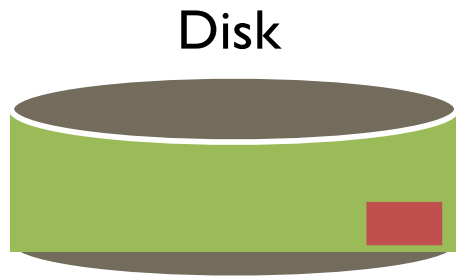
Each page in virtual address space maps to one of three locations:

- 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 (if hardware-managed TLB)

PRESENT BIT



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

Where does each page reside?
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 // TLB miss...

- Hardware or OS walk page tables
- If PTE designates page is present, then page in physical memory
load TLB with vpn -> ppn mapping

Else // Page fault

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

INTERACTION WITH OS SCHEDULER?

Page faults are very expensive; read page from disk (ms)

When a process has a page fault, what state is process moved to?

- BLOCKED

When OS finishes reading page into physical memory and sets present bit?

- READY

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?

1) 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)
- **Problems?**

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

2) 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

CHAT: PAGE REPLACEMENT EXAMPLE

Page reference string: ABCABDADBCB

		OPT	FIFO	LRU
Metric:	ABC	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
Miss count	A	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
Three pages of physical memory	B	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
	D	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
	A	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
	D	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
	B	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
	C	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
	B	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>

Three pages
of physical
memory

3 misses

	OPT			
ABC	<table><tr><td>A</td><td>B</td><td>C</td></tr></table>	A	B	C
A	B	C		
A	<table><tr><td>A</td><td>B</td><td>C</td></tr></table>	A	B	C
A	B	C		
B	<table><tr><td>A</td><td>B</td><td>C</td></tr></table>	A	B	C
A	B	C		
D	<table><tr><td>A</td><td>B</td><td>D</td></tr></table>	A	B	D
A	B	D		
A	<table><tr><td>A</td><td>B</td><td>D</td></tr></table>	A	B	D
A	B	D		
D	<table><tr><td>A</td><td>B</td><td>D</td></tr></table>	A	B	D
A	B	D		
B	<table><tr><td>A</td><td>B</td><td>D</td></tr></table>	A	B	D
A	B	D		
C	<table><tr><td>C</td><td>B</td><td>D</td></tr></table>	C	B	D
C	B	D		
B	<table><tr><td>C</td><td>B</td><td>D</td></tr></table>	C	B	D
C	B	D		

1 miss

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!

CHAT: FIFO PERFORMANCE MAY DECREASE!

Consider access stream: ABCDABEABCDE

Consider 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, do not implement Perfect LRU

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

Software Perfect LRU

- OS maintains ordered list of physical pages by reference time
- When page is referenced:
- When need victim:
- Trade-off:

Hardware Perfect LRU

- Associate timestamp register with each page
- When page is referenced:
- When need victim:
- Trade-off:
(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

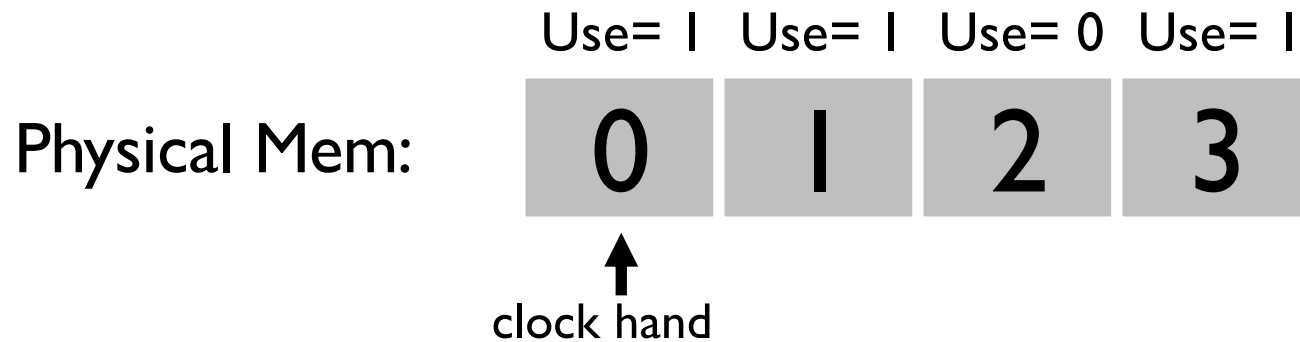
Hardware

- 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**

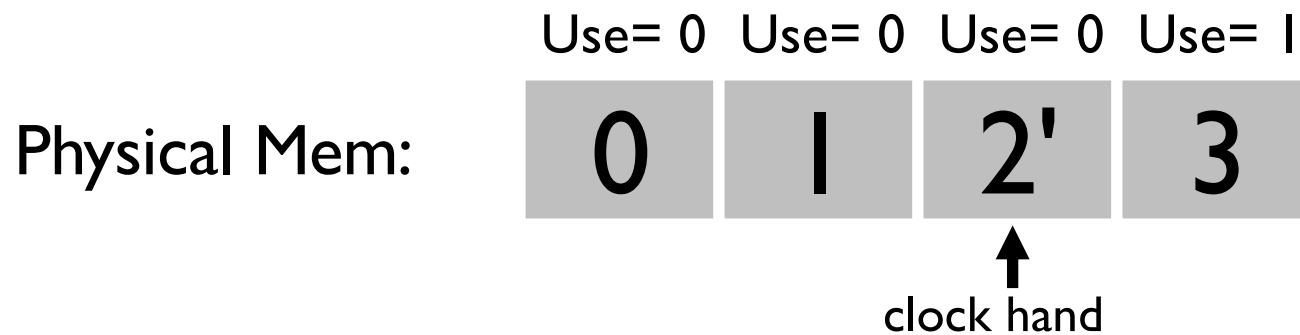
CLOCK: LOOK FOR A PAGE



Need to find page to replace; which will OS pick?

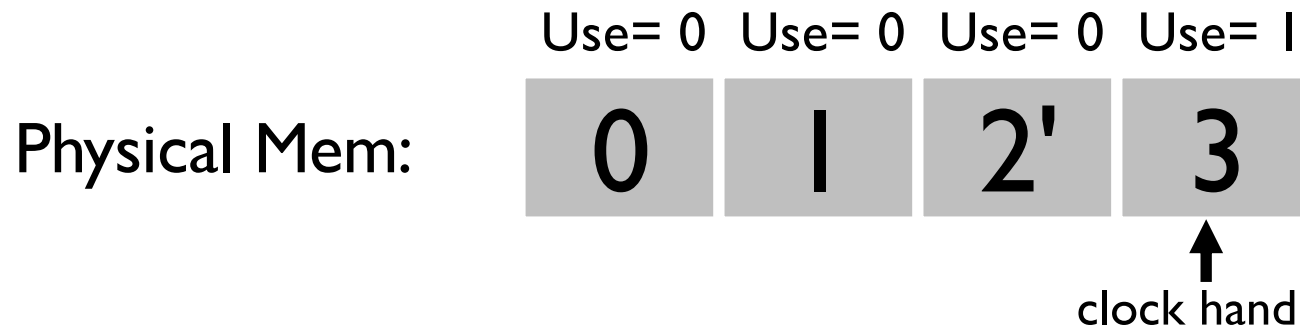
Clear use bit for page 1 and advance clock hand...

CLOCK: LOOK FOR A PAGE



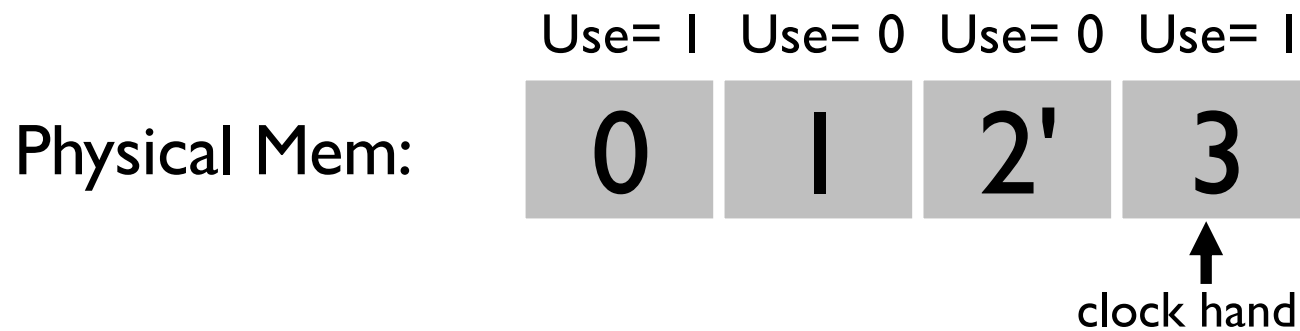
Evict **page 2** because it has not been recently used
Load physical page 2 with new contents from disk

CLOCK: LOOK FOR A PAGE



Continue the running process
Imagine page 0 is accessed

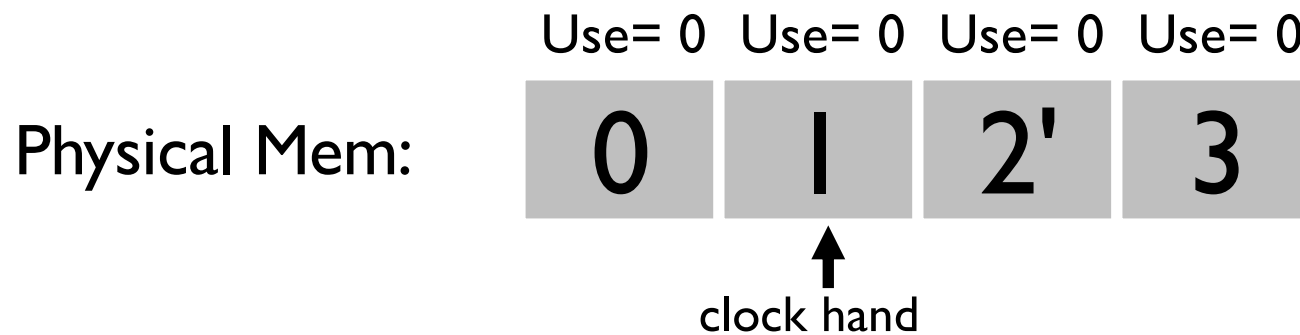
CLOCK: LOOK FOR A PAGE



Set use bit for page 0

When need to find next victim, which?

CLOCK: LOOK FOR A PAGE



Clear Use bits as sweep

Evict page 1 because not recently accessed



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 have 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

Add software counter (“chance”)

- Intuition: Better ability to differentiate across pages (how much they are being accessed)
- Increment `chance` software counter if use bit is 0
- Replace when `chance` exceeds some specified limit

PROBLEMS WITH LRU-BASED REPLACEMENT

Previous lecture (TLBS): LRU poor when working set $>$ available resources

LRU does not consider **frequency** of accesses

- Is a page accessed **once** in the past equal to one accessed **N** times?
- Common workload problem:
 - Scan (sequential read, never used again) one large data region flushes memory

Solution: Track frequency of accesses to page

Pure LFU (Least-frequently-used) replacement

- Problem: LFU can never forget pages from the far past

Examples of other more sophisticated algorithms:

- LRU-K and 2Q: Combines recency and frequency attributes
- Expensive to implement, LRU-2 used in databases

Similar policies used for replacing blocks in file buffer cache

WORKING SET

Working Set:

Set of memory locations that need to be cached for reasonable cache hit rate (for individual process)

As add more memory, hit rate increases (generally)

Thrashing:

When system has too small of memory cache

CHAT: PERFORMANCE IMPACT?

Assume workload of multiple processes

- Processes each alternate between CPU and I/O
- Each access some amount of memory

What happens to system performance (throughput = jobs/sec) as we increase the number of processes?

- If the sum of the working sets $>$ physical memory?

BONUS: WHAT IF NO HARDWARE SUPPORT?

What can the OS do if hardware does not have use bit (or dirty bit) (and hardware-filled TLB)?

- Can the OS “emulate” these bits?

Leading question:

- How can the OS get control (i.e., generate a trap) every time use bit should be set? (i.e., when a page is accessed?)

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)

SIMULATIONS: MULTI-LEVEL PAGETABLES

Each page is 32 bytes

The virtual address space for process in question (assume only one) is 1024 pages, or 32 KB

Physical memory consists of 128 pages

Thus, a virtual address needs 15 bits (5 for the offset, 10 for the VPN).

A physical address requires 12 bits (5 offset, 7 for the PFN).

The system assumes a multi-level page table. Thus, the upper five bits of a virtual address are used to index into a page directory; the page directory entry (PDE), if valid, points to a page of the page table. Each page table page holds 32 page-table entries (PTEs). Each PTE, if valid, holds the desired translation (physical frame number, or PFN) of the virtual page in question.

The format of a PTE is thus:

VALID | PFN6 ... PFN0

and is thus 8 bits or 1 byte.



The format of a PDE is essentially identical:

VALID | PT6 ... PT0

SIMULATIONS: MULTI-LEVEL PAGETABLES

Virtual Address **611c**: Contents? Or Fault?

0x611c →

PDBR: 108 (decimal) [page directory is held in this page]

0110 0001 0001 1100

page 108: 83 fe e0 da 7f d4 7f eb be 9e d5 ad e4 ac 90 d6 92 d8 c1 f8 9f e1 ed e9 a1 e8 c7 c2 a9 d1 db ff

Which entry of PageDir?

PDE: 16+8=24

→ 0xa1

→ 1010 0001

→ Valid and 33

page 33: 7f 7f 7f 7f 7f 7f 7f 7f b5 7f 9d 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f f6 b1 7f 7f 7f 7f

Which entry of Page Table?

PTE: 8

→ b5

→ 1011 0101

→ valid and 32+16+4+1 = Page 53

page 53: 0f 0c 18 09 0e 12 1c 0f 08 17 13 07 1c 1e 19 1b 09 16 1b 15 0e 03 0d 12 1c 1d 0e 1a 08 18 11 00

Which offset on page?

→ offset 16+8+4 = 28

Final data value: 08!