CS 423
Operating System Design: Smaller Page Tables and Swapping
02/19

Ram Alagappan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

AGENDA / LEARNING OUTCOMES

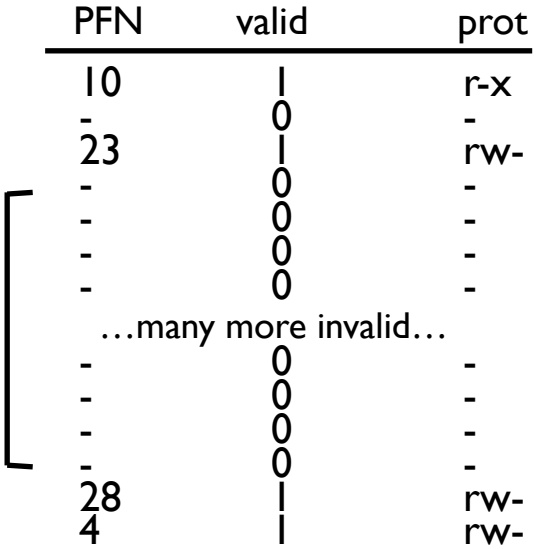
Finish discussion on smaller page tables

Support larger virtual mem than physical mem

What are the mechanisms and policies for this?

RECAP

MANY INVALID PTES



how to avoid storing these?

Problem: linear PT must still allocate PTE for each page (even unallocated ones)

APPROACHES

1. Segmented Pagetables
2. Multi-level Pagetables
    - Page the page tables
    - Page the pagetables of page tables…
3. Inverted Pagetables

Paging and Segmentation - Chat for 2 mins

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1

0x002070 read:
0x202016 read:
0x104c84 read:
0x010424 write:
0x210014 write:
0x203568 read:

...
0x01f
0x011
0x003
0x02a
0x013
...
0x00c
0x007
0x004
0x00b
0x006
...

0x001000

0x002000

Bounds: # PTE entries

Disadvantages of Paging and Segmentation

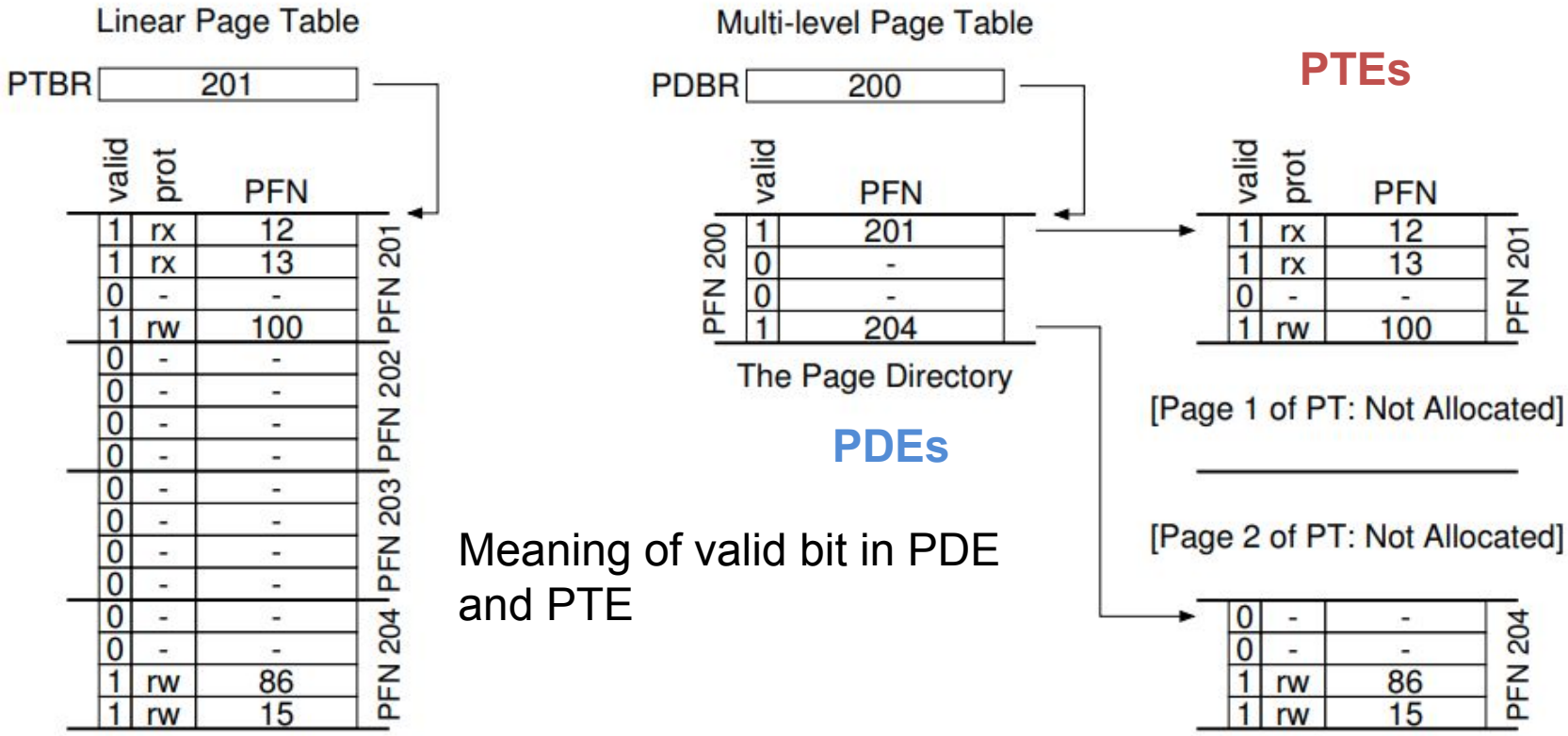
Potentially large page tables (for each segment)

- Must allocate page table for each segment *contiguously*
- Page table size?
  - Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Each page table is:
= Number of entries * size of each entry
= Number of pages * 4 bytes
= 2^18 * 4 bytes = 2^20 bytes = 1 MB!

END RECAP

Multilevel Page Table – Key Idea



Meaning of valid bit in PDE and PTE

Address format for multilevel Paging

30-bit address:

How should logical address be structured? How many bits for each paging level?

- Goal: each inner page table fits within a page
- PTE size * number PTE = page size

  Assume PTE size = 4 bytes

  Each inner page can have 1024 PTE

  # bits for inner page = 10

Remaining bits for outer page:

- 30 – 12 – 10 = 8 bits

Multilevel Translation EXAMPLE

page directory

PPN	valid
0x3	1
-	0
-	0
	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
	0
-	0
-	0
-	0
0x92	1

page of PT (@PPN:0x3)

PPN	valid
0x10	1
0x23	1
-	0
-	0
0x80	1
0x59	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

page of PT (@PPN:0x92)

PPN	valid
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x55	1
0x45	1

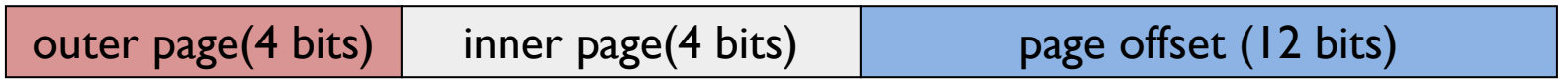
translate 0x01ABC

Physical address?

Look PD[0]

Look PT[1] → arrive at page 0x23

Concat ABC to arrive at 0x23ABC



20-bit address

Multilevel Translation EXAMPLE

page directory

PPN	valid
0x3	1
-	0
-	0
	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
	0
-	0
-	0
-	0
0x92	1

page of PT (@PPN:0x3)

PPN	valid
0x10	1
0x23	1
-	0
-	0
0x80	1
0x59	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

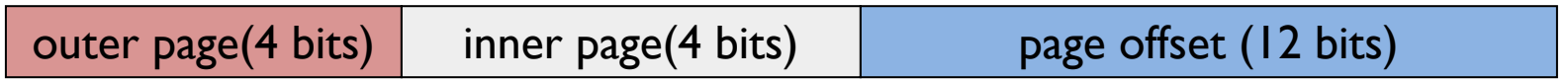
page of PT (@PPN:0x92)

PPN	valid
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x55	1
0x45	1

Translate

VA: 0x04000

VA: 0xFEED0



PROBLEM WITH 2 LEVELS?

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

Consider 30 bit address with 512-byte pages

We need 9 bits for offset, leaving 21 bits for VPN

Remember our goal: each inner page should fit within a page

So how many PTE per page? With 512-byte pages and 4-byte PTE, we can have 128 entries → this means 7 bits for inner page, leaving 14 bits for outer page (or directory) → 2^{14} PDEs

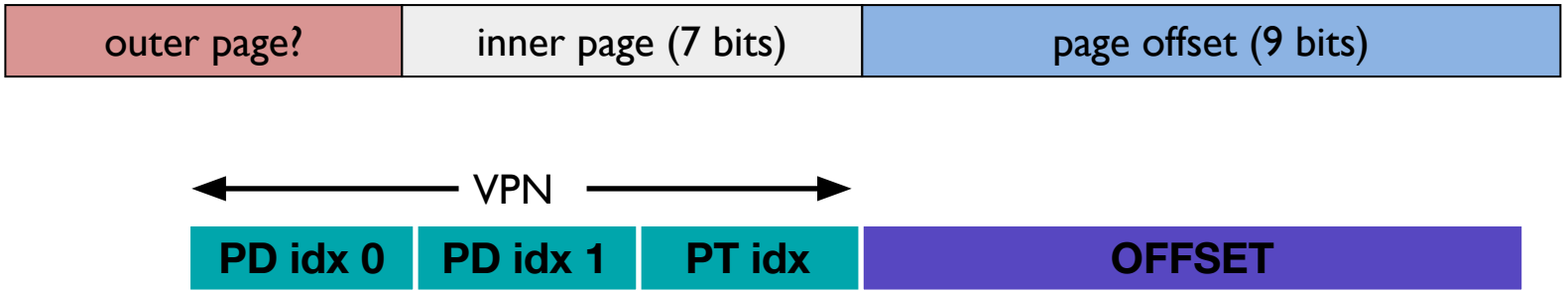
Assume 4-byte PDE, then PD itself will span _____ pages?

PD cannot be contained in one page now!

PROBLEM WITH 2 LEVELS?

Solution: page the page directory!

Add another level of page directory that points to PD pages



Can keep going recursively…

FULL SYSTEM WITH TLBS - Chat for 2 mins

On TLB miss: lookups with more levels more expensive

Assume 3-level page table

Assume 256-byte pages

Assume 16-bit addresses

Assume ASID of current process is 211

ASID	VPN	PFN	Valid
211	0xbb	0x91	1
211	0xff	0x23	1
122	0x05	0x91	1
211	0x05	0x12	0

How many physical accesses for each instruction? (Ignore ops changing TLB)

(a) 0xAA10: movl 0x1111, %edi

(b) 0xBB13: addl $0x3, %edi

(c) 0x0519: movl %edi, 0xFF10

INVERTED PAGE TABLE

Single table for the system - keeps track of which process and which virtual page within the process is using the physical page (size of PT: number of physical pages)

Naïve approach:

Search through data structure <ppn, vpn+asid> to find match

Better:

Find possible match entries by hashing vpn+asid

Smaller number of entries to search for exact match

Used in IBM PowerPC

Managing inverted page table requires software-controlled TLB (although doing some these in HW is possible)

SUMMARY: BETTER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

Many options for efficiently organizing page tables

If software-managed TLB, 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

SWAPPING

Motivation

OS goal: Support processes when not enough physical memory

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

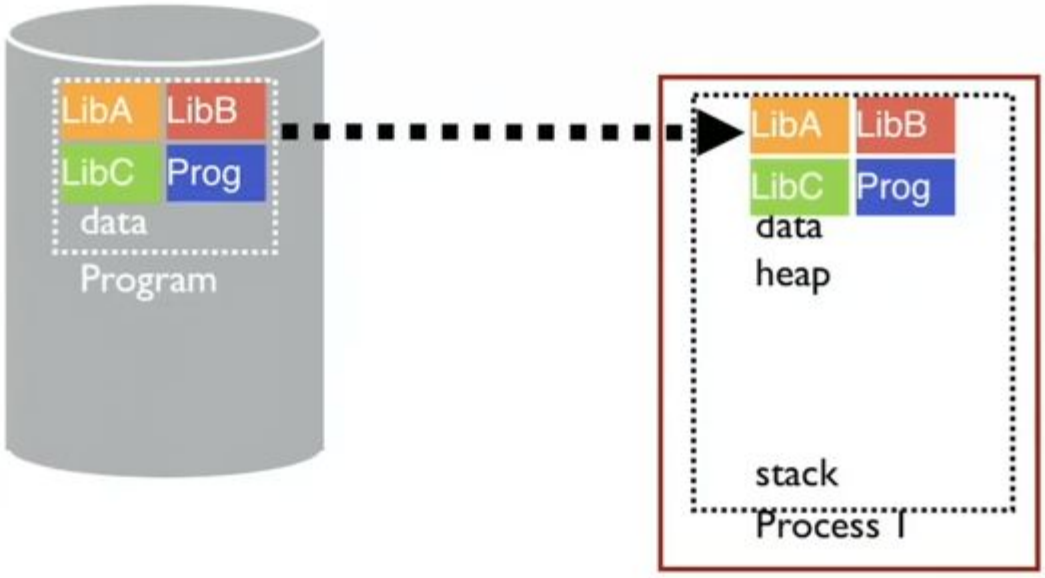
User code should be independent of amount of physical memory

- Correctness, performance

Virtual memory: OS provides illusion of more physical memory

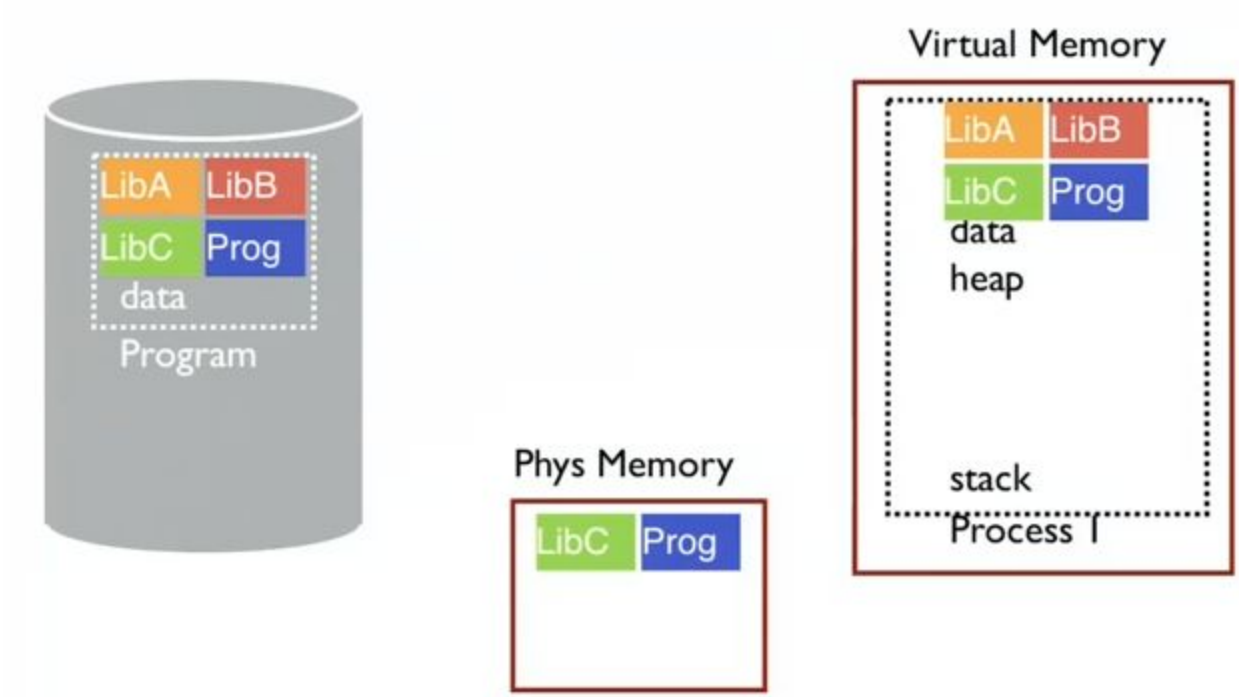
Why does this work?

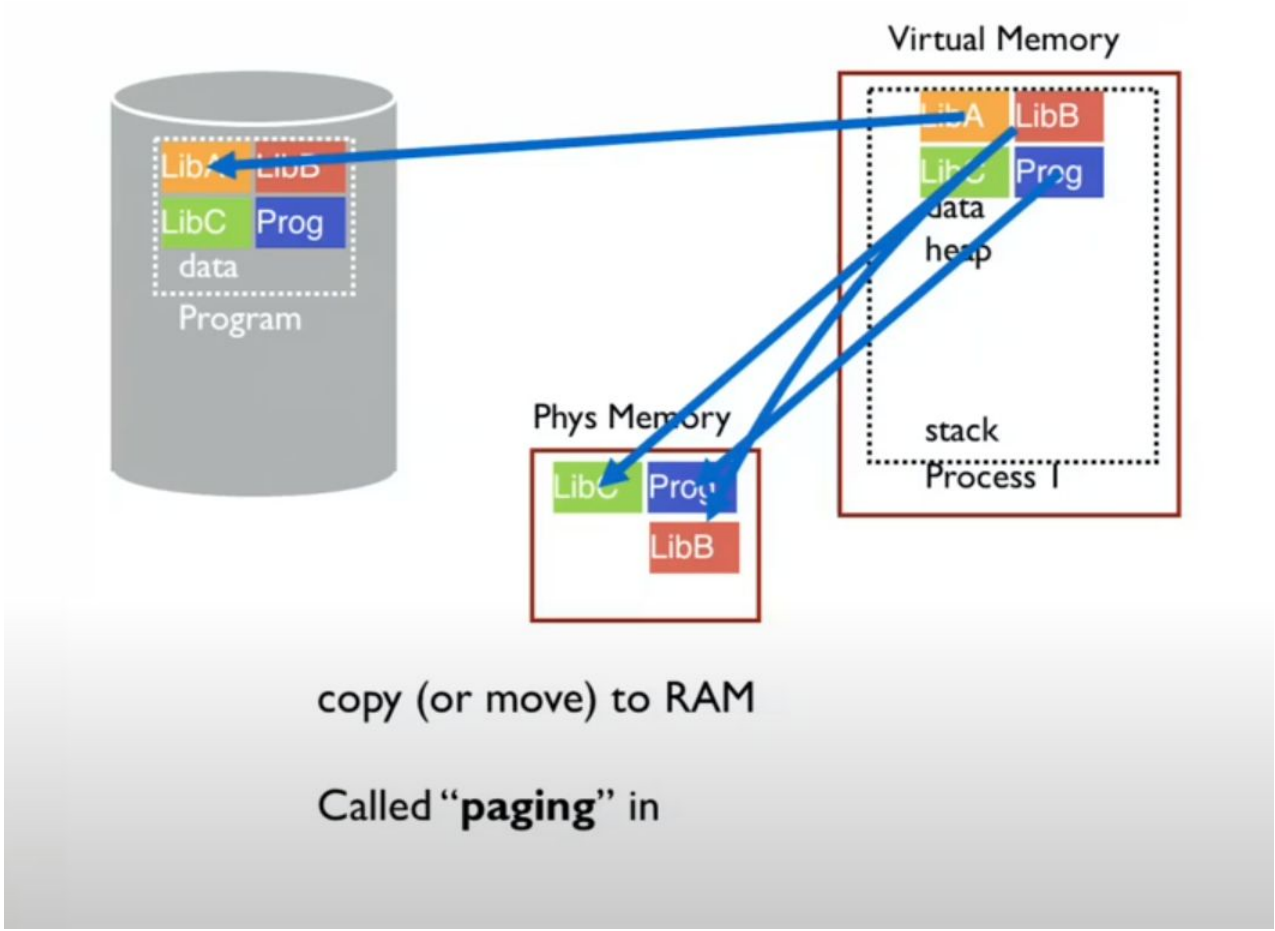
- Relies on key properties of user processes (workload) and machine architecture (hardware)



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

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


LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog
Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1



copy (or move) to RAM

Called "**paging**" in

Locality of Reference

Leverage locality of reference within processes

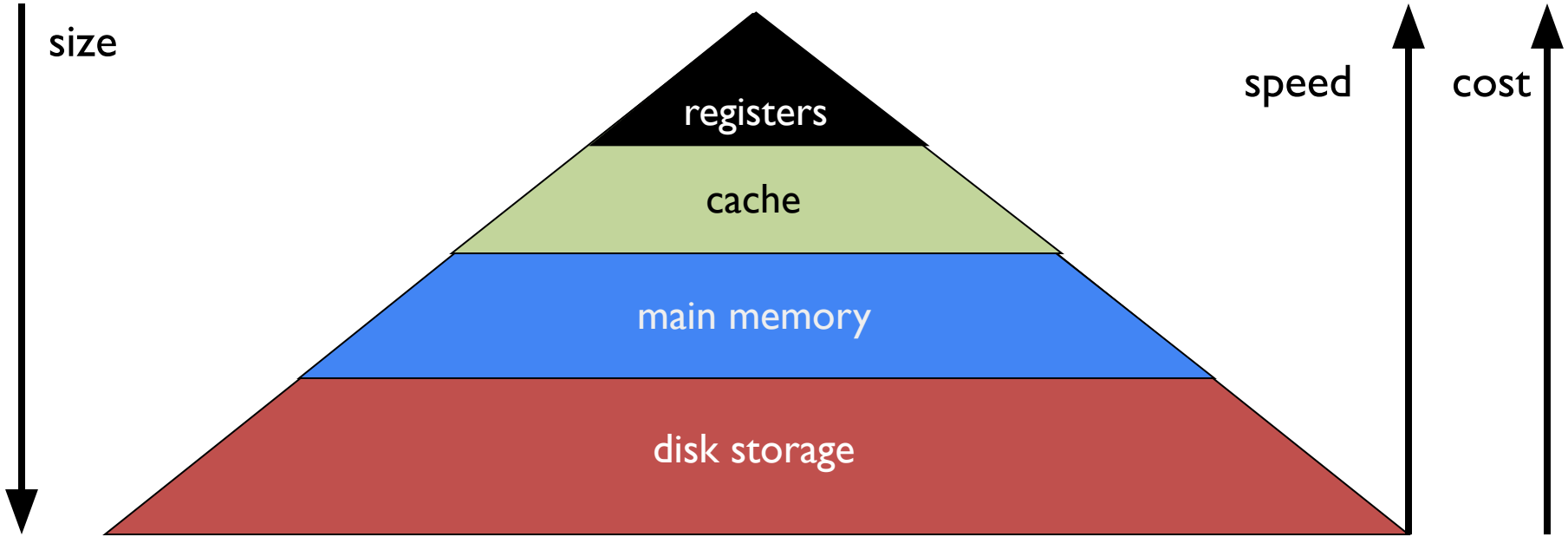
- Spatial: reference memory addresses **near** previously referenced addresses
- Temporal: reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
  - Estimate: 90% of time in 10% of code

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** to determine which pages live in memory and which on disk

Virtual Address Space Mechanisms

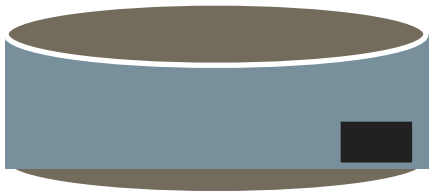
Each page in virtual address space maps to one of three locations:

- 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 (use the same bits for PFN and block#)
  - Causes trap into OS when page is referenced
  - OS handles by reading from disk and putting the page in memory

Disk



Phys Memory

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

What if access vpn 0x2?

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
  (i.e., present bit is cleared)

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 (use dirty bit in PTE)
- OS reads referenced page from disk into memory
- Page table is updated, present bit is set
- Process continues execution

Soft vs. Hard Page Faults

Hard:

- requires reading the page from disk
- expensive

Soft:

- Page already in memory, but OS need to do some work, cheaper
- E.g., COW – forked child modifies a page

Interaction With OS Scheduler

During TLB miss, process is still in RUNNING state

But page faults are expensive (OS must read from disk)

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

What state when the OS brings the page into physical memory and sets present bit?

Always Swapping to Disk?

The backing store or swap partition can be SSD

It can be remote memory! (RDMA-based swapping) – more recent and experimental

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)
- Two costs of bad prefetching?

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, ...

Hints with madvise()

int madvise(void *addr, size_t length, int advice) – allows user program to give hints to OS about how page can be prefetched

MADV_RANDOM

MADV_SEQUENTIAL

MADV_WILLNEED

Many more on linux…

Generally, hard to tune and extract better performance than OS default

Page Replacement

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

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

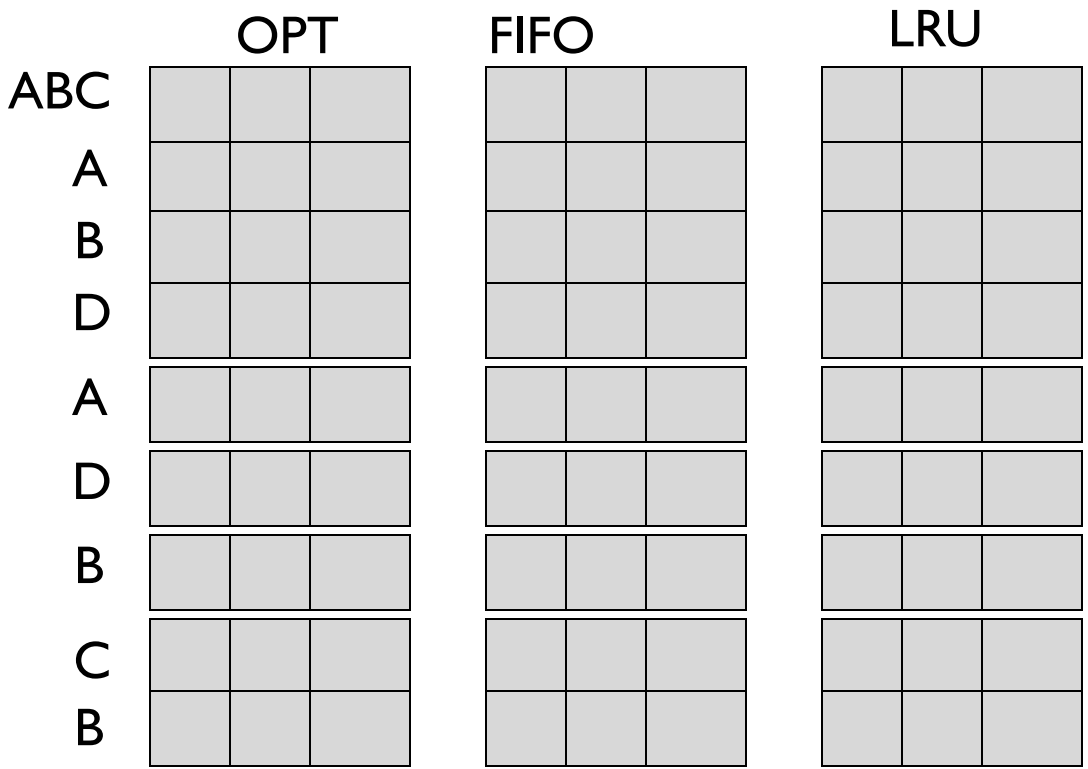
- Intuition: Use past to predict the future
- Advantages: With locality, LRU approximates OPT
- Disadvantages:
  - Harder to implement, must track which pages have been accessed

Page Replacement - chat for 2 mins

Page reference string: ABCABDADBCB

Metric:
Miss count

Three pages of physical memory



Page Replacement

Page reference string: ABCABDADBCB

Metric:
Miss count

Three pages of physical memory

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

LRU vs. OPT

Think of a workload/setting where LRU will be way worse than OPT…

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?

9 misses with 3 pages and 10 misses with 4 pages

3 Pages
A, B, C, D, A, B, E miss
A hit, B hit, C, D miss
E hit

4 pages
A, B, C, D miss
A hit, B hit, E miss –
A, B, C, D, E miss

Implementing LRU (conceptually)

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

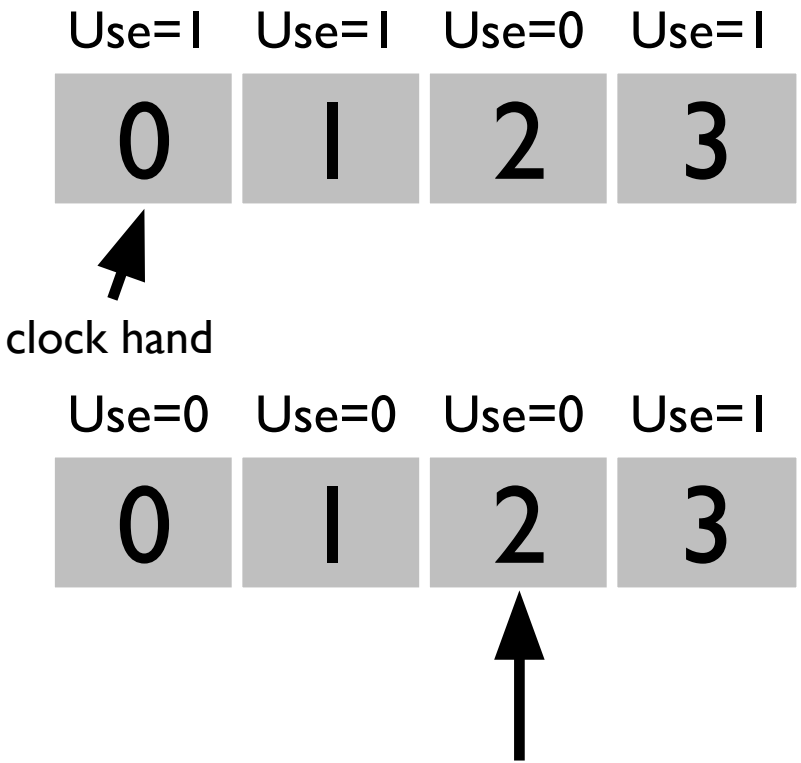
- LRU is an approximation anyway, so approximate more
- Goal: Find an old page, but not necessarily the very oldest

Clock Algorithm

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

- 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



Evict contents of frame 2, load in new page

Clock Algorithm - Who does what - HW or OS

- Keep use (or reference) bit for each page frame
- When page is referenced: set use bit
- Page replacement: Look for page with use bit cleared
  (has not been referenced for awhile)
  - 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 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

Add software counter ("change")

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

Linux 2Q

Active list and inactive list

On first reference, put in inactive list, if accessed again, move to active list

Otherwise, it will be swapped to disk (uses FIFO to make the decision)

Active list is very similar to a clock

Move pages that not referenced to the inactive list until active list is 2/3rd of physical memory size

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)