CS 423
Operating System Design: Swapping
Feb 19

Ram Kesavan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

AGENDA / LEARNING OUTCOMES

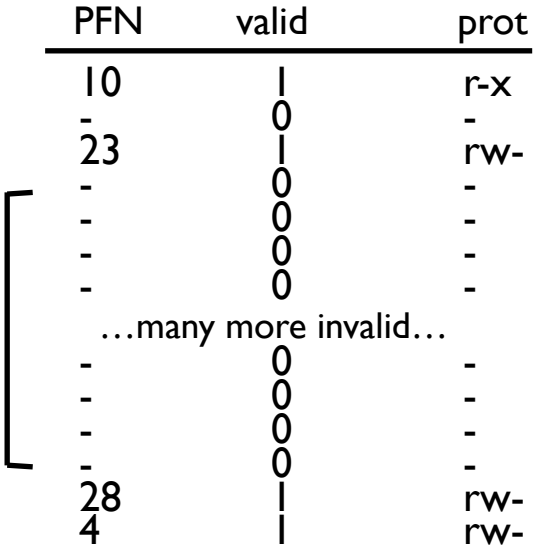
Finish discussion on better page tables

How do we make everything fit in memory?

What are the mechanisms and policies for this?

RECAP

MANY INVALID PTES



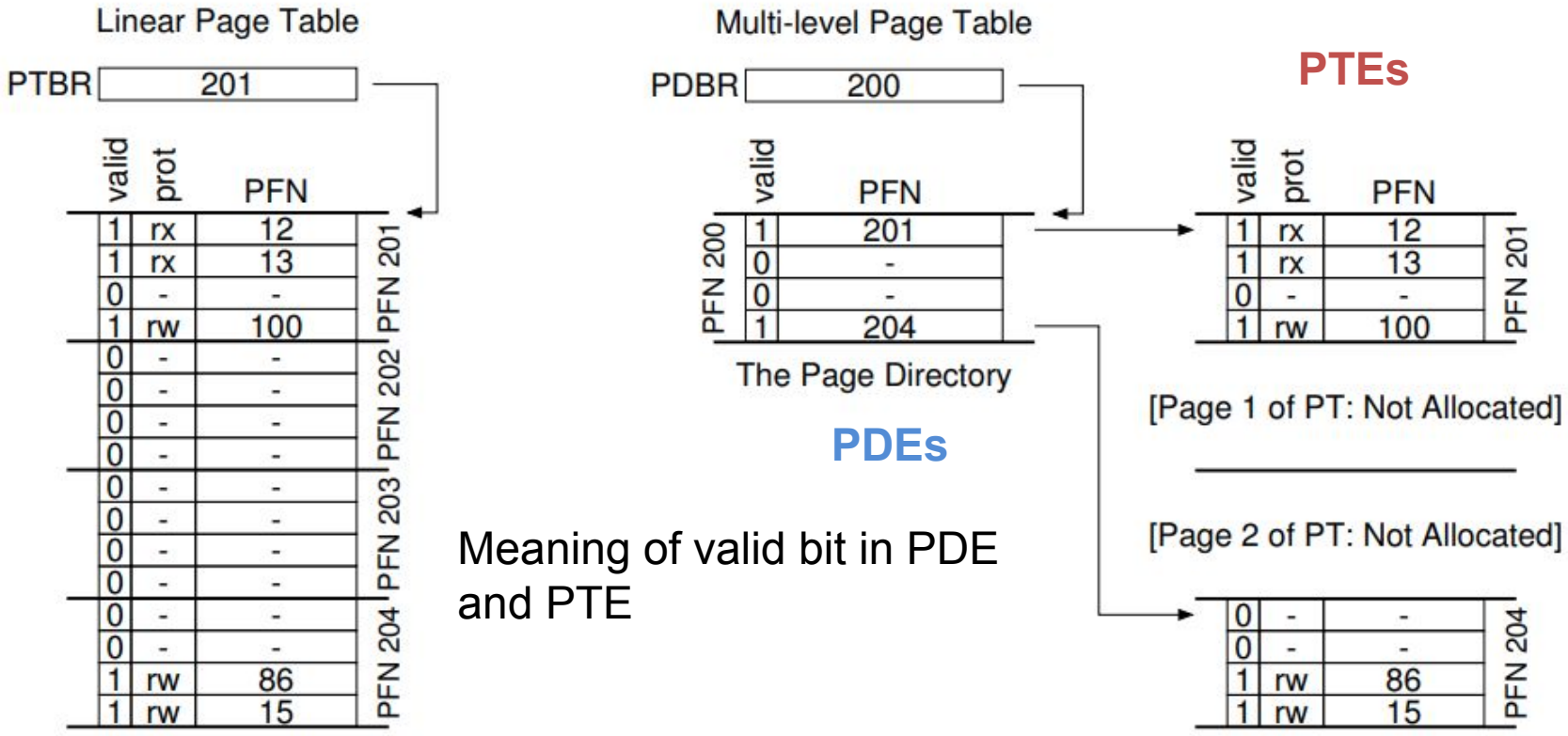
how to avoid storing these?

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

APPROACHES

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

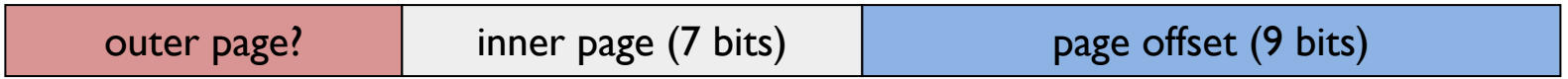
Multilevel Page Table – Key Idea



PROBLEM WITH 2 LEVELS?

Solution: page the page directory!

Add another level of page directory that points to PD pages



Can keep going recursively! Let page = 4KB (offset is 12 bits); 1K PTEs/page

2 level tree: 1K * 1K pages = 4GB (10 + 10 + 12 = 32 bit address)

3 level tree: 1K * 1K * 1K pages = 4TB (10 + 10 + 10 + 12 = 42 bit address)

SUMMARY: BETTER PAGE TABLES

Problem: Linear page table requires too much contiguous memory

Many options for efficiently organizing page tables

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

- Eg. inverted page tables (hashing)

If Hardware handles TLB miss, page tables decided beforehand

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

END RECAP

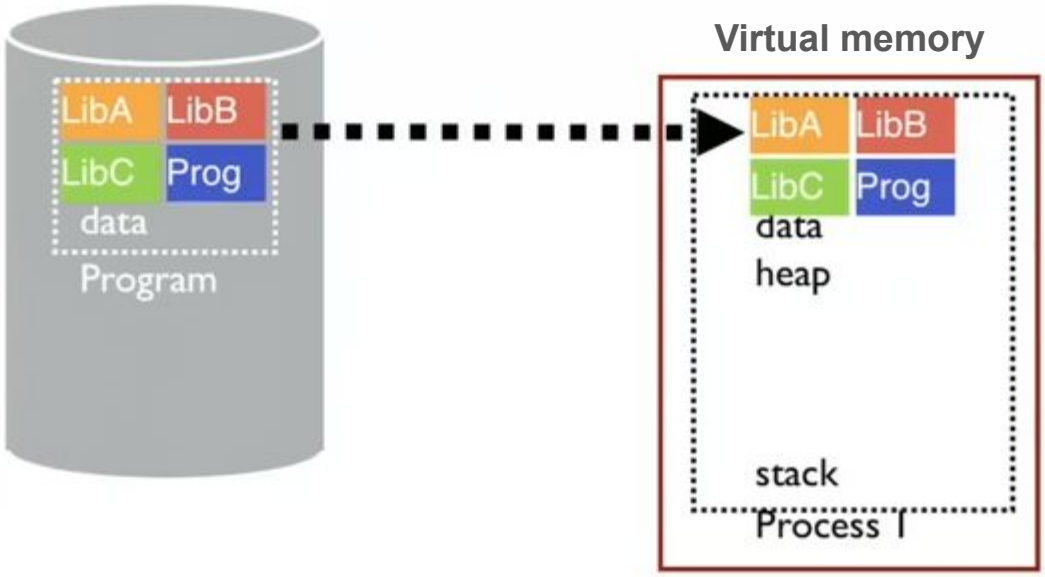
SWAPPING

ONE LAST PROBLEM

Memory virtualization thus far:

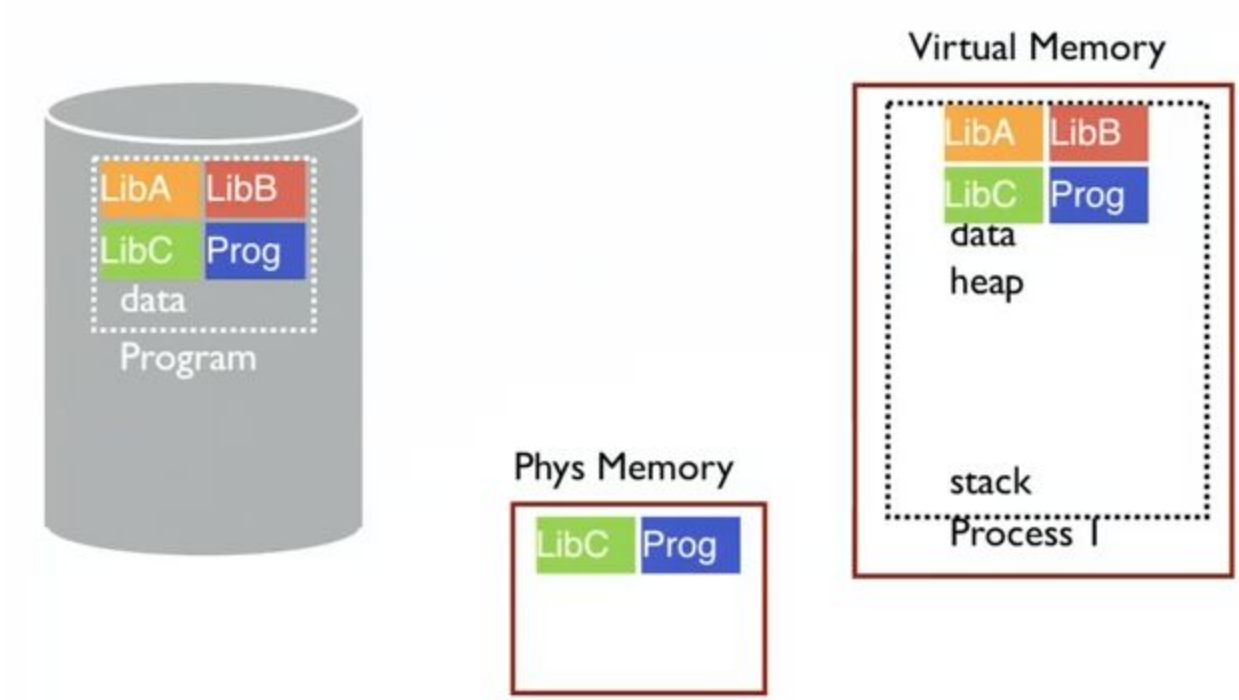
- Support multiple processes, each with its virtual memory
- The virtual memory for a process is contiguous
- …but it's physical memory doesn't need to be
- Solved external fragmentation using paging
- The page table doesn't need to be contiguous - page it!
- Use hardware support: TLB

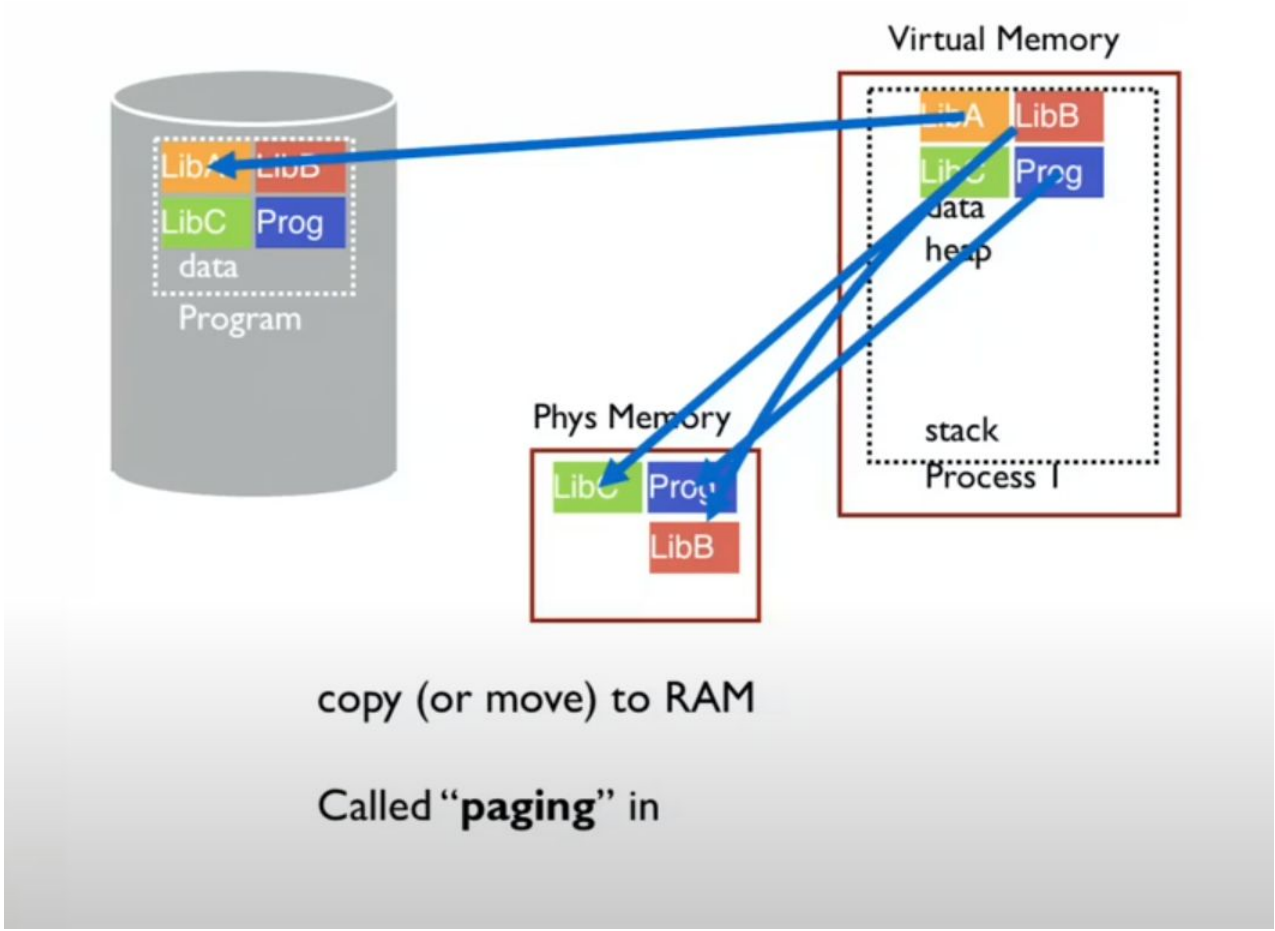
One remaining problem: can't fit everything in physical memory



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


Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
Phys Memory
LibC
Prog
LibB
LibA
LibB
LibC
Prog
data
Program
copy (or move) to RAM
Called "paging" in

Locality of Reference

Leverage locality of reference within processes

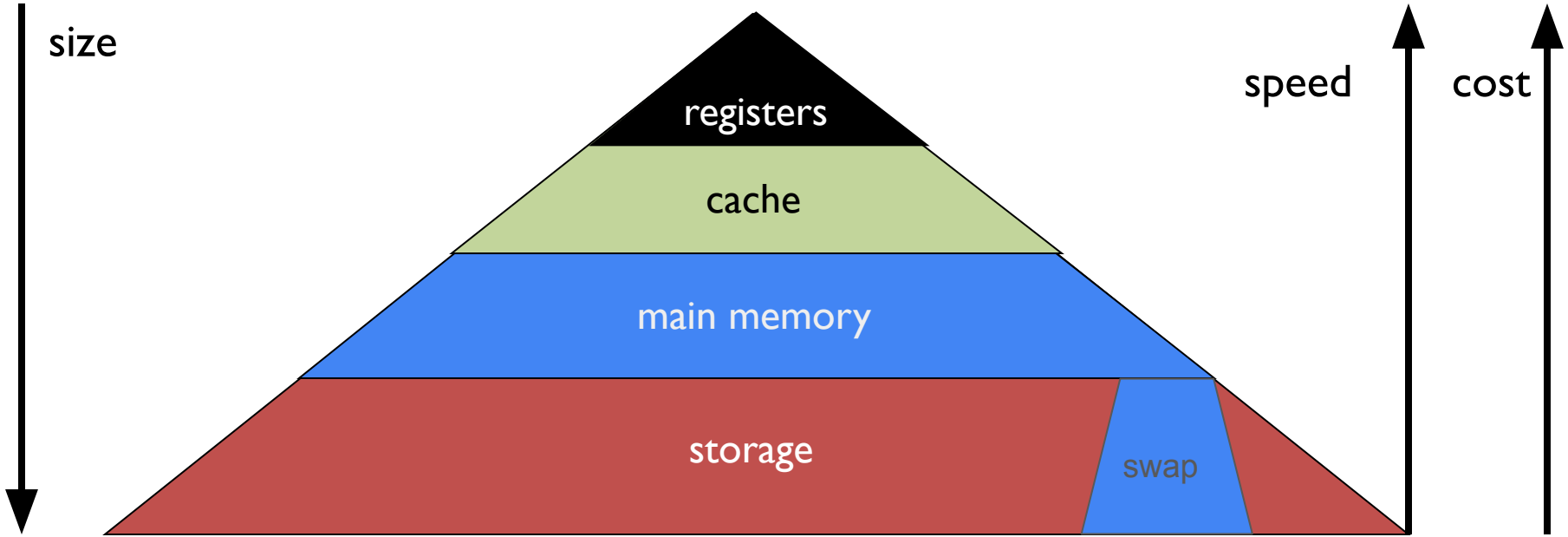
- Spatial: reference memory addresses **near** previously referenced addresses
- Temporal: reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
  - Estimate: 90% of time in 10% of code

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



Swap Space

Designated & reserved fraction of persistent storage

- Sizing is configurable; usually multiple of the physical memory

Persistent storage is accessed in units of “blocks”

- Typically, block size is equal to or a multiple of page size.
- “Paging out”: write a page out to swap space
- “Paging in”: read a page in from swap space

OS tracks free space in the swap space

- Simple block addressing: linearly from 0 to n

SWAPPING Intuition

Idea: OS keeps unreferenced/unneeded pages in swap space

- Slower, cheaper storage backing the memory

Process can run even when all its pages are not in main memory
OS and h/w cooperate to make storage seem like memory

- Illusion: process address space is entirely in main memory

Requirements:

- **Mechanism:** locate (move) pages in (between) memory and storage
- **Policy:** determine which pages to move, and when

Note: Books/Internet refers to swap space as disk (can be SSD)

SWAPPING

Question 1: Can a USB-stick serve as swap space?

Question 2: What happens to swap space when the OS is restarted, i.e., machine is rebooted?

Question 3: What happens to swap space when a laptop hibernates?

Question 3: What if (total memory to all processes) > (RAM + swap space)?

Virtual Address Space Mechanisms

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

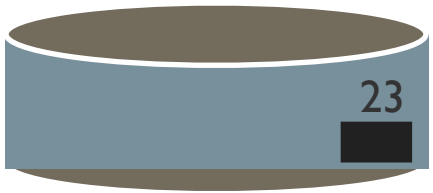
- Physical main memory: fast
- Swap Space (backing store): slow
- Nothing: Free

Extend page tables with an extra bit: present

- permissions (r/w), valid, dirty, 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 or block#)
  - Causes trap into OS when page is not in memory: Page Fault
  - OS reads page from swap space and puts it into memory

When vpn 0x2 is accessed

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

BITS: valid, dirty, present

Dirty:

- Page has been written to by the process
  - So, heap or stack
- If dirty bit for PFN 16 never gets set, we can avoid writing it back to 23 in swap space
  - So, preserve 23 in swap space until process dirties PFN 16
- For code & static data: page it in from the compiled binary each time
  - Don’t need swap space for these pages

Valid:

- Virtual memory address has been allocated
- So was mapped to physical memory at some point in the past

Present:

- The page is currently mapped to physical memory

Virtual Memory: Full Mechanism 1

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 valid + present bits set, then page in physical memory
  - Insert PTE into TLB, retry instruction

  Else (valid and/or present is 0) //Page fault
  - Trap into OS (not handled by hardware)
  - Find free PFN. If necessary,
    - Select victim page in memory to kick out
    - If modified (dirty bit set), page out victim page to swap
  - Kick off I/O to read the page from storage (swap or otherwise) to PFN
  - Process is marked as BLOCKED, OS does a context switch

Virtual Memory: Full Mechanism 2

The read I/O is tagged with PID, PTE, destination PFN, etc.
When read I/O completes, interrupt handler runs

- Updates PTE with new PFN
- Sets the present bit
- Makes the original process (PID) runnable

When process runs:

- Wakes up in kernel mode in the page fault handler
- Cleans itself up
- Returns to user mode to retry instruction
- Results in TLB miss
  - Will find PTE with present bit

// previous page

REPLACEMENT

Does not really occur 1 page at a time

- Inefficient to wait until memory is entirely full!

Swap/Page daemon: background process

- Low & High watermarks (LW & HW)
- When #free-pages < LW
  - Evict sufficient pages until #free-page > HW
  - Go to sleep

Other Advantages?

Virtual Memory: Full Mechanism 2 (modification)

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 present bit set, then page in physical memory
  - Insert PTE into TLB, retry instruction

  Else //Page fault
  - Trap into OS (not handled by hardware)
  - Find free PFN. ~~If necessary,~~ ← if #free-pages < LW
    - ~~Select victim page in memory to kick out~~ ← wake up daemon
    - ~~If modified (dirty bit set), page out victim page to swap~~

      ←if no free-pages, sleep for daemon
  - Kick off I/O to read the page from swap space to PFN
  - Process is marked as BLOCKED, OS does a context switch

Soft vs. Hard Page Faults

Hard: Expensive (process transitions to BLOCKED)

Requires reading the page from disk

Soft: Cheap (no context switch required)

Page already in memory, but OS need to do some work,

E.g.,

(1) COW when a fork(ed) child dirties a page

(2) PTE can point to a shared PFN (shared code, library, etc.)

Interaction With OS Scheduler

During TLB miss, process is still in RUNNING state

But (hard) 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?

SWAP SPACE LOCATION

Hard disk or SSD

- Slow/cheap vs fast/expensive

Storage device (hard disk or SSD) can be remote

- Remote: not on the same physical system
- RDMA-based swapping

SWAPPING POLICIES

SWAPPING Policies

Goal: Minimize number of page faults

- Page faults require millisecs to handle (reading from swap)
- Implication: Lots of time for OS to make good decisions

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 **kicked out?**

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
- Problem: Pay cost of page fault for every newly accessed page

Prepaging (anticipatory, prefetching): Load page beforehand

- OS predicts future access (oracle) and prefetches pages into memory
- 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 to give hints to OS

```
MADV_RANDOM          // kernel may disable prefetching
MADV_SEQUENTIAL      // kernel may aggressively prefetch…
                     // …and kick out after access
MADV_WILLNEED
MADV_DONTNEED
```

and more…

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

Page Replacement

How to select victim page in main memory to kick out?

- Write out victim page to disk if modified (dirty bit set)
- If victim page is not modified (clean), just discard
- When urgent, pick clean pages as victims

OPTIMAL: Replace the page accessed furthest in the future

- Pro: Guaranteed to minimize #page faults
- Con: Requires knowing the future! Impractical, but good for comparisons

Page Replacement

FIFO: Replace page that has been in memory the longest

- Intuition: Referenced long time ago, get rid of it
- Advantage: Fair as each page gets equal residency; Easy to implement
- Disadvantage: Some pages may remain popular

LRU: Least-recently-used

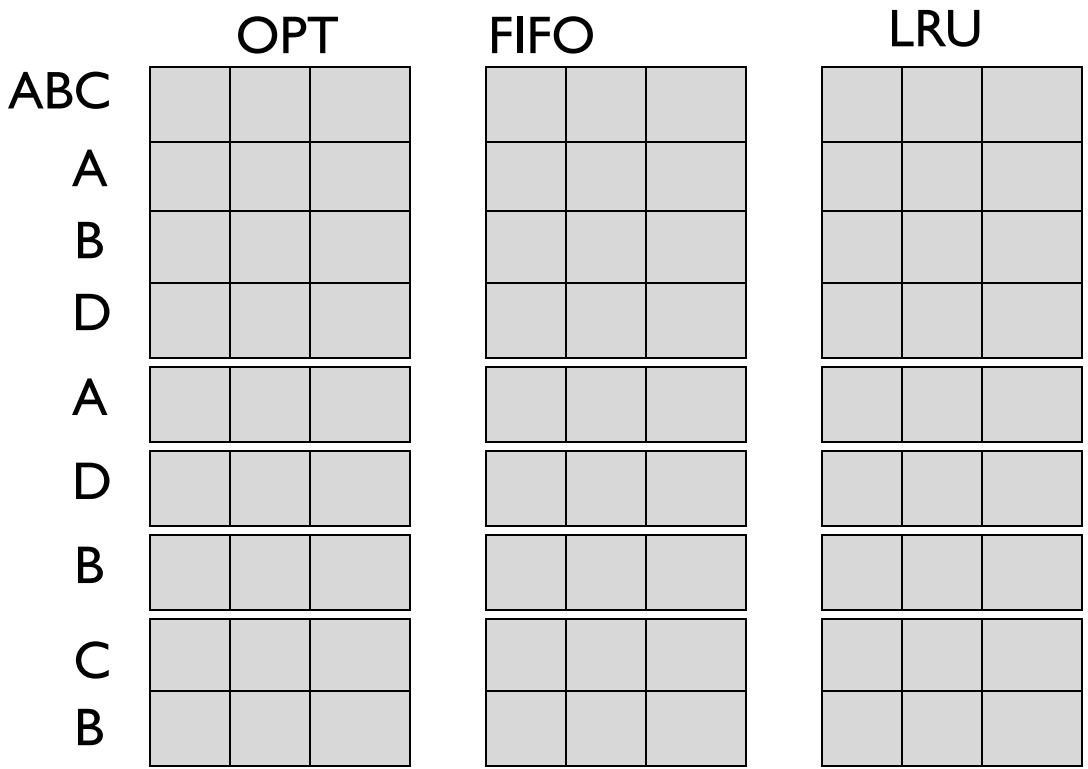
- Intuition: Recent past predicts the near future
- Advantages: With locality, LRU approximates OPT
- Disadvantages:
  - Harder to implement, must track which pages have been accessed

Page Replacement - chat for 2 mins

Page reference string: ABCABDADBCB

Metric:
Miss count

Three pages of physical memory



LRU vs. OPT

QUESTION: Think of a workload & config where LRU will be way worse than OPT. Random replacement might do better that LRU!

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

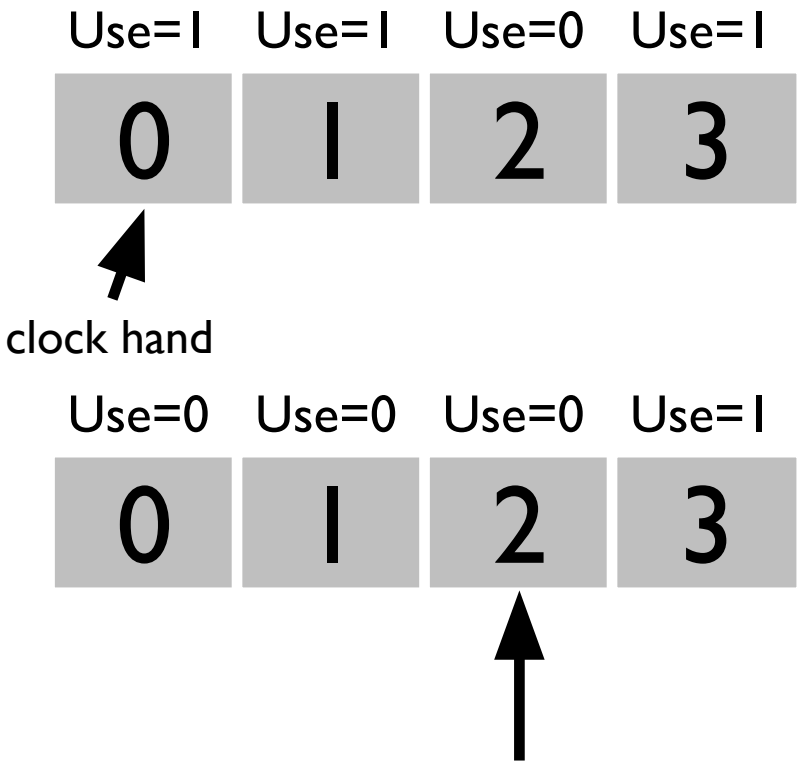
In practice, do not implement Perfect LRU

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

CLOCK ALGORITHM: APPROX LRU

- 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 fashion
  - 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 EXTENSIONS

Replace multiple pages at once

- Intuition: Efficient to write pages to swap space in bulk
- 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: Can tell how often pages are accessed
Increment software counter if use bit is 0
Replace when change exceeds some specified limit

LINUX 2Q

Active list and inactive list

On first reference, put in inactive list

- If accessed again, promoted to active list
- Pick victim pages from inactive list in FIFO order to swap out

Active list is very similar to a clock

In background: move pages from active to inactive list until active list is <= 2/3rd of physical memory size

LINUX CODE

include/linux/swap.h

- Data structure used to track swap area
  - struct swap_info_struct
- Operates in units of 256 pages “clusters”
  - struct swap_cluster_info
- Parallelism: each CPU is given a cluster

OUT OF MEMORY

What happens when total memory allocated to all processes > RAM + swap space?

1. OS constantly pages in/out from swap space–aka *thrashing*

   System will noticeably slow down

2. OOM Killer (kernel) gets invoked

   Kill processes that hog memory; typically younger processes

3. User processes: malloc() returns null

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)