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

PFN	valid	prot
10	1	r-x
-	0	-
23	1	rw-
-	0	-
-	0	-
-	0	-
-	0	-
…many more invalid…		
-	0	-
-	0	-
-	0	-
-	0	-
28	1	rw-
4	1	rw-

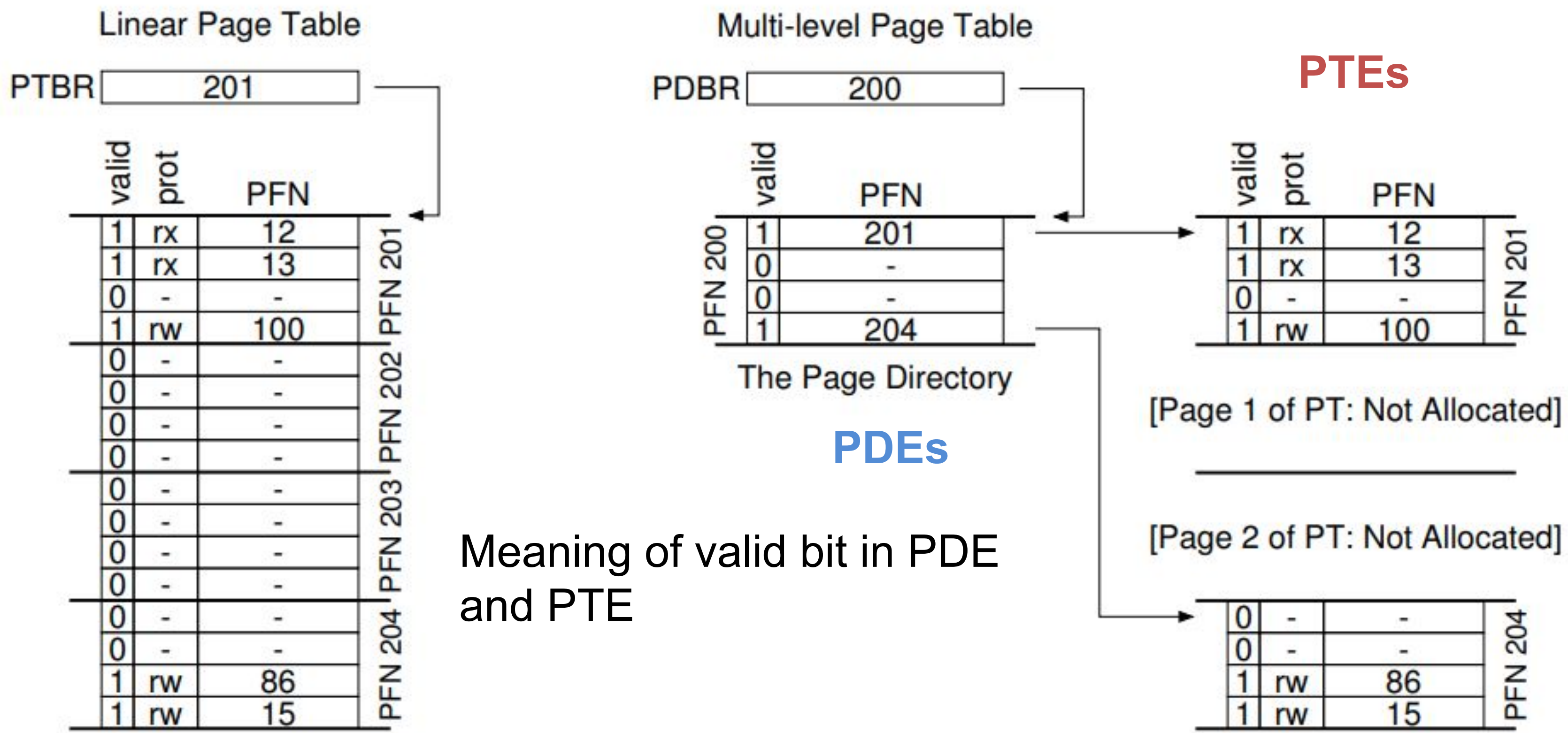
how to avoid storing these?

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

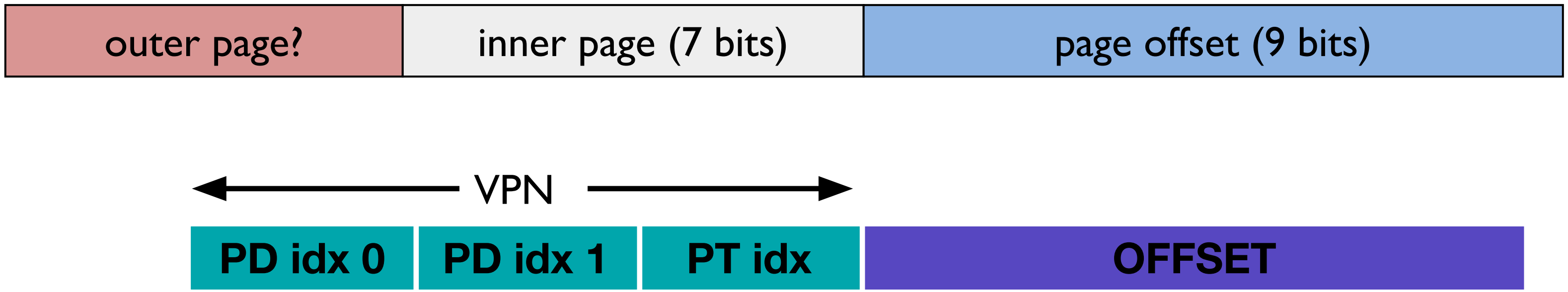


Meaning of valid bit in PDE and PTE

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

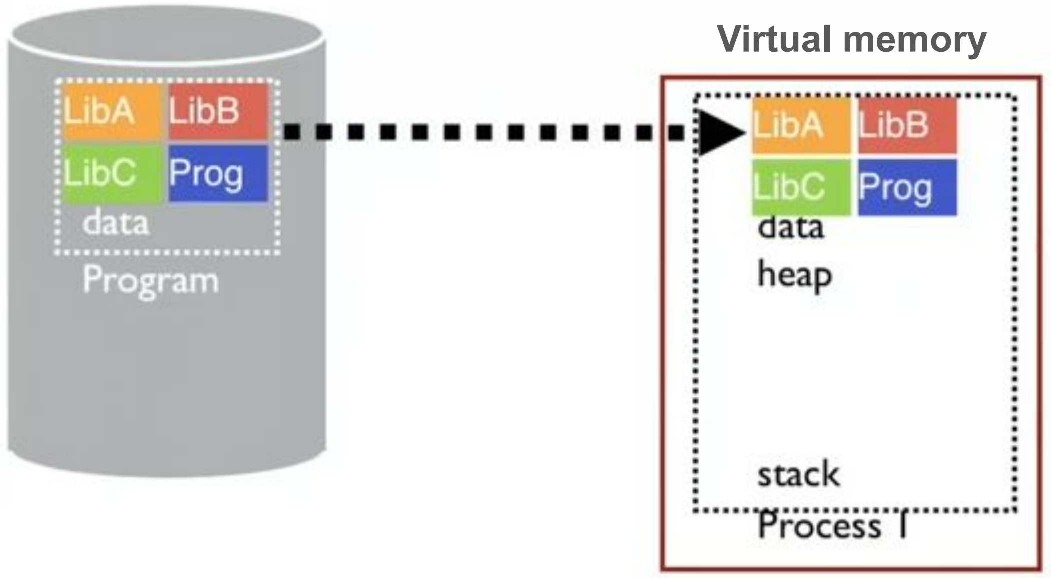
SWAPPING

ONE LAST PROBLEM

Memory virtualization thus far:

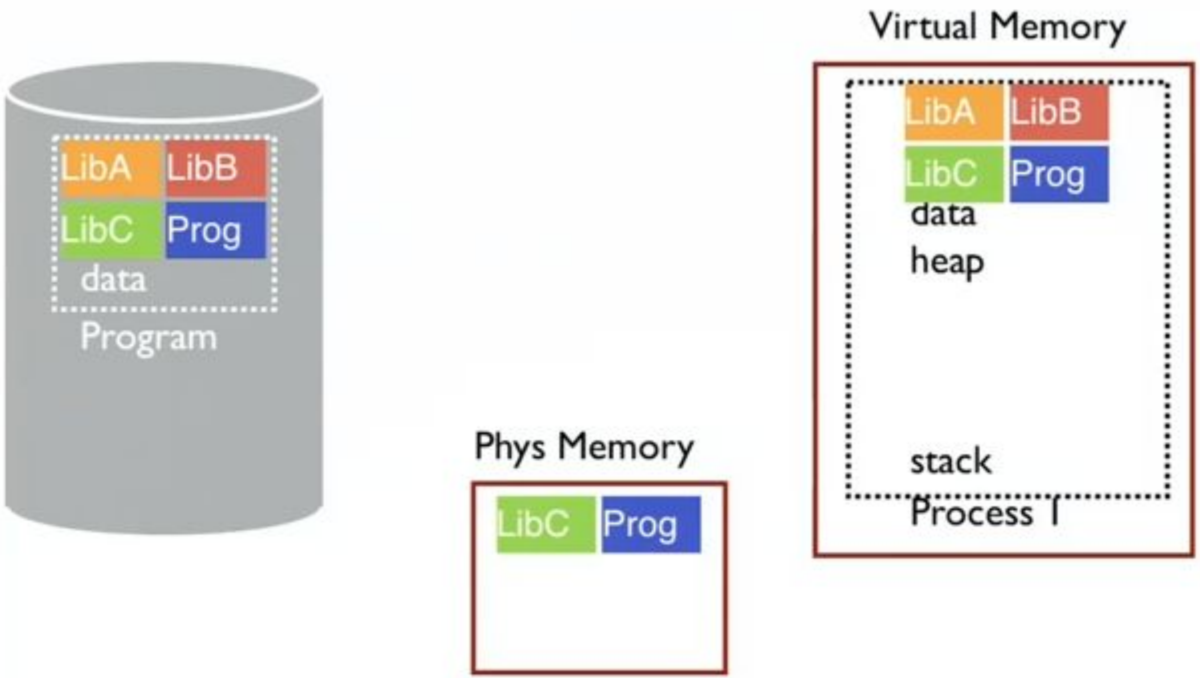
- Support multiple processes, each with its virtual memory
- The virtual memory for a process is contiguous
- …but it’s physical memory doesn’t need to be
- Solved external fragmentation using paging
- The page table doesn’t need to be contiguous - page it!
- Use hardware support: TLB

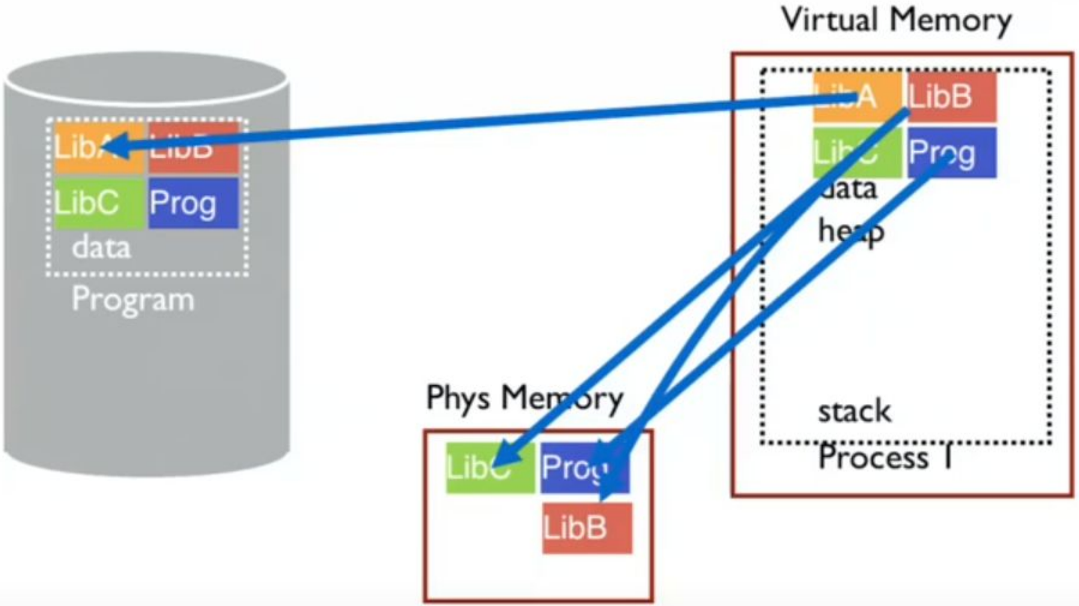
One remaining problem: can’t fit everything in physical memory



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



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

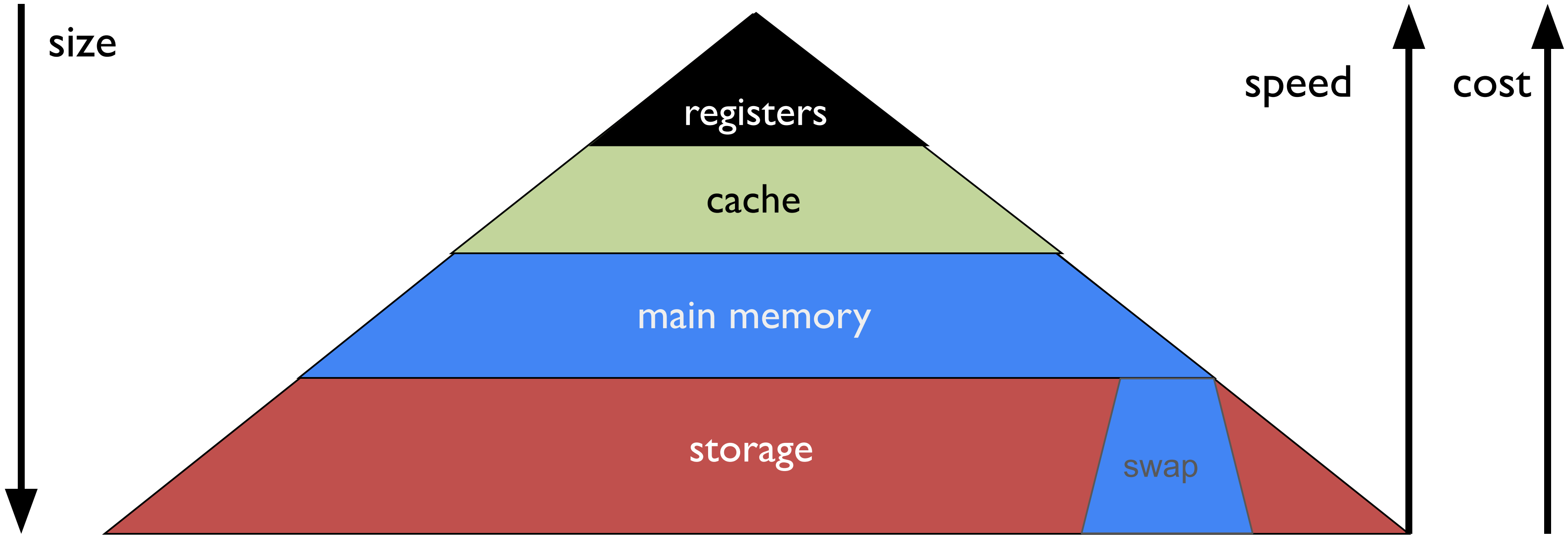
Implication:

- 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

*not necessarily

SWAPPING Intuition

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 in 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:

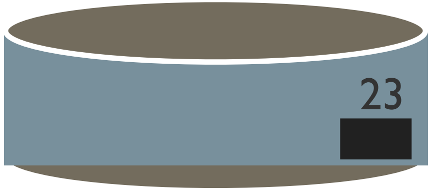
- Physical main memory: fast
- Swap Space (backing store): slow
- Nothing: Free

Extend page tables with an extra bit: present

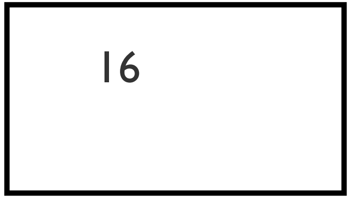
- permissions (r/w), valid, dirty, present
- Page in memory: present bit set in PTE
- Page on disk: present bit cleared
  - PTE points to block on disk (use the same bits for PFN or block#)
  - Causes trap into OS when page is not in memory: Page Fault
  - OS reads page from swap space and puts it into memory

When vpn 0x2 is accessed

Disk



Phys Memory



PFN	valid	prot	present
10	1	r-x	1
-	0	-	-
23 / 16	1	rw-	0 / 1
-	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 transitions to BLOCKED, OS does a context switch

Virtual Memory: Full Mechanism 2

The read I/O was 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

Unless qualified, page fault is always considered hard

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.)