Memory Management: Swapping to Disk

Kai Mast

CS 537

Fall 2022

ANNOUCEMENTS

- Project 2b will be released on Monday
  - Should be less work than P2a
  - Due on 10/24

- Midterm review session in class on 10/18

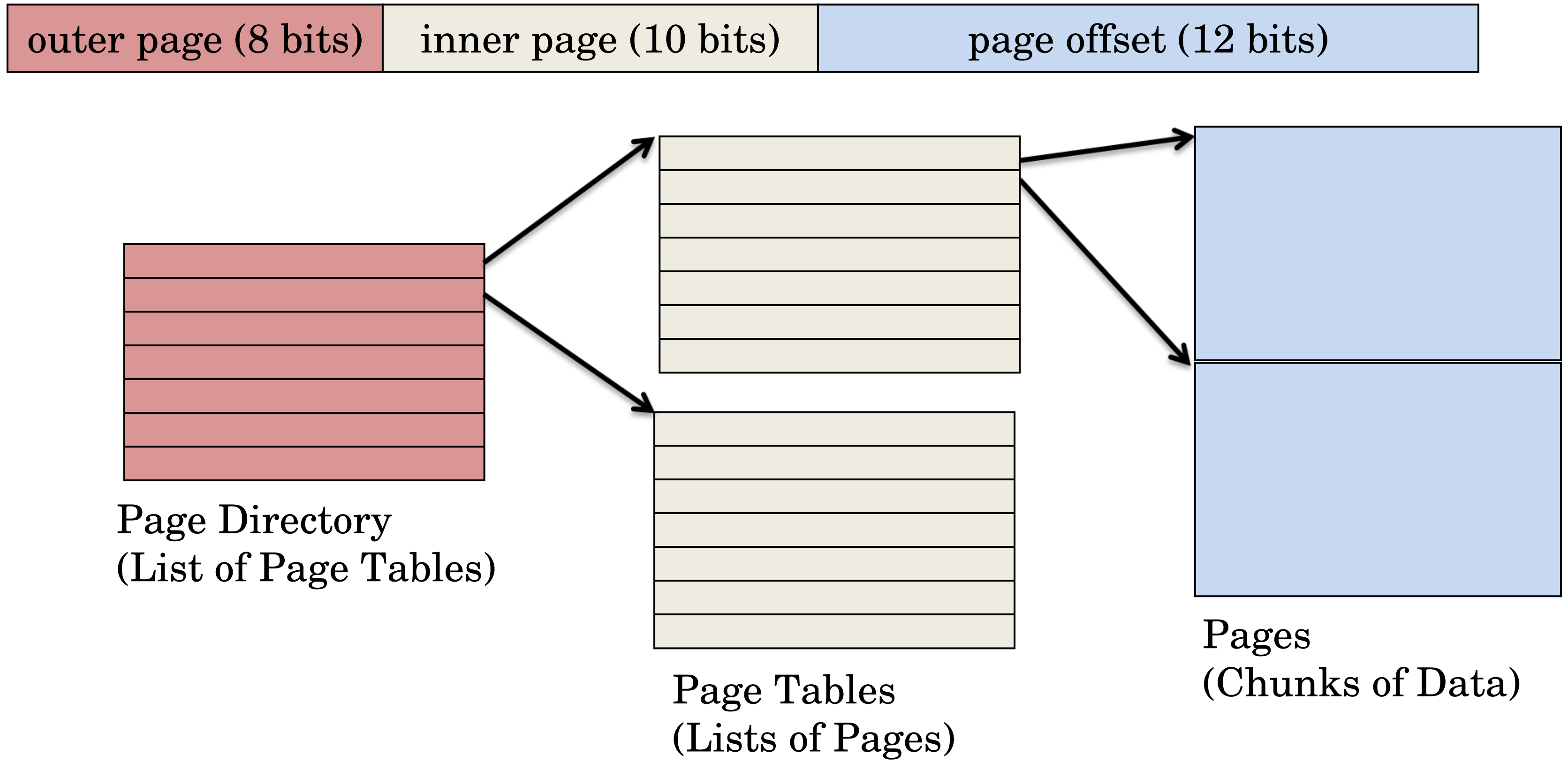
Recap: Smaller Pagetables

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

(Book Chapter 20)

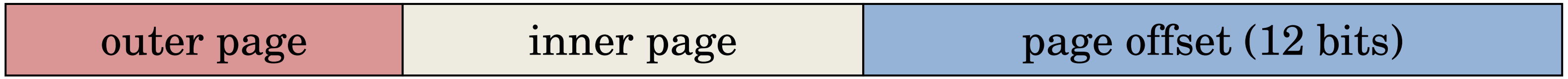
MULTI-LEVEL PAGE TABLES

30-bit virtual address:



ADDRESS FORMAT FOR MULTI-LEVEL PAGING

30-bit address



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

Goal? Each page table fits within a page (assume page size = 4KB, 2^12 bytes)

- number PTE * PTE size (assume 4bytes) = page size

How many page table entries can we fit on page?

- 4KB / 4bytes = 1K (1024 = 2^10) entries
- bits for selecting inner page = 10

Remaining bits for outer page: 30 bits - 12 bits - 10 bits = 8 bits

MULTI-LEVEL 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
-	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`

`0x23ABC`

translate `0x04000`

`0x80000`

translate `0xFEED0`

`0x55ED0`

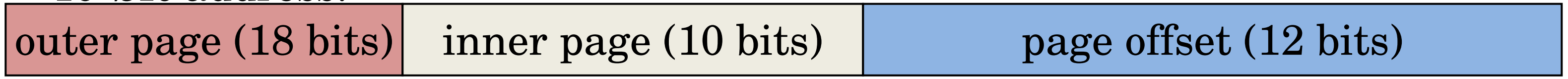
20-bit address

outer page(4 bits)	inner page(4 bits)	page offset (12 bits)

PROBLEMS WITH TWO LEVELS?

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

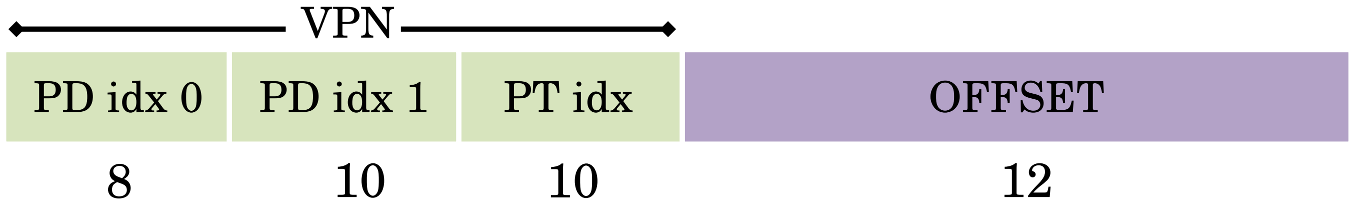
40-bit address:



PageSize / PTE Size = 4KB / 4 bytes => max 1K entries per level

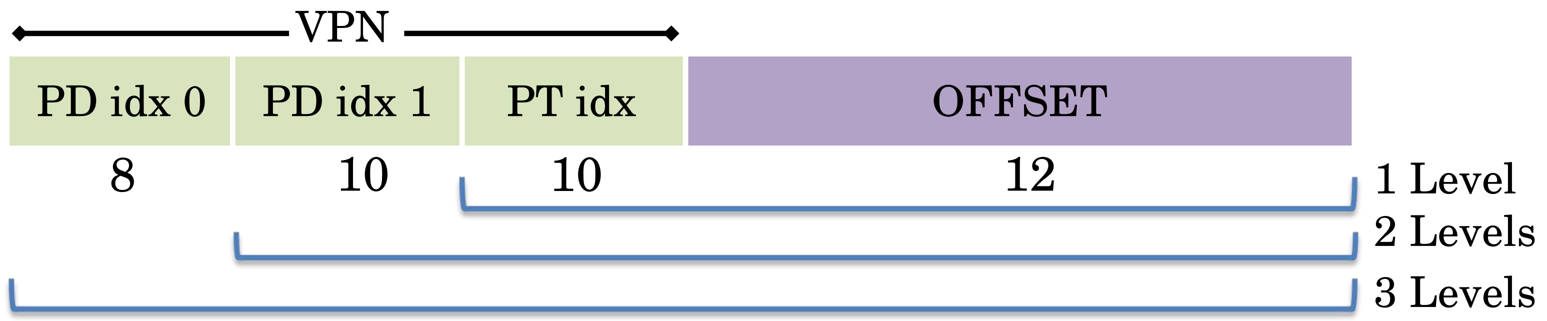
Solution:

- Split page directories into pieces
- Use another page directory to refer to the page directory pieces



MORE THAN TWO LEVELS OF PAGING

- Split page directories into pieces
- Use another page directory to refer to the page directory pieces



How large is virtual address space available with each level?

1 level:	
2 levels:	
3 levels:	

How many extra memory accesses?

FULL SYSTEM WITH TLBS

On TLB miss: look-ups across multiple levels are more expensive

Assume 3-level page table and 256-byte pages
Assume 16-bit addresses
Assume ASID of current process is 211

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

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

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

`0xaa`: (TLB miss -> 3 for addr trans) + 1 instr fetch

Total: 8

`0x11`: (TLB miss -> 3 for addr trans) + 1 movl

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

`0xbb`: (TLB hit -> 0 for addr trans) + 1 instr fetch from `0x9113`

Total: 1

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

`0x05`: (TLB miss -> 3 for addr trans) + 1 instr fetch

Total: 5

`0xff`: (TLB hit -> 0 for addr trans) + 1 movl into `0x2310`

OTHER APPROACHES

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

INVERTED PAGE TABLES

Observation

- We only need entries for virtual pages w/ valid physical mappings (assuming there is no swapping; see later…)
- Have entries based on physical pages (inverted)

Naïve approach

- Search through data structure <PFN, VPN+ASID> to find match
- Too much time to search entire structure

Faster

- Find possible matches entries by hashing VPN+ASID
- Smaller number of entries to search for exact match

TLB still manages most cases

- Managing inverted page table requires software-controlled TLB

SUMMARY: SMALLER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

Many options exist 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 page table fits within a page

Next Topic: What if allocated virtual memory does *not* fit into physical memory?

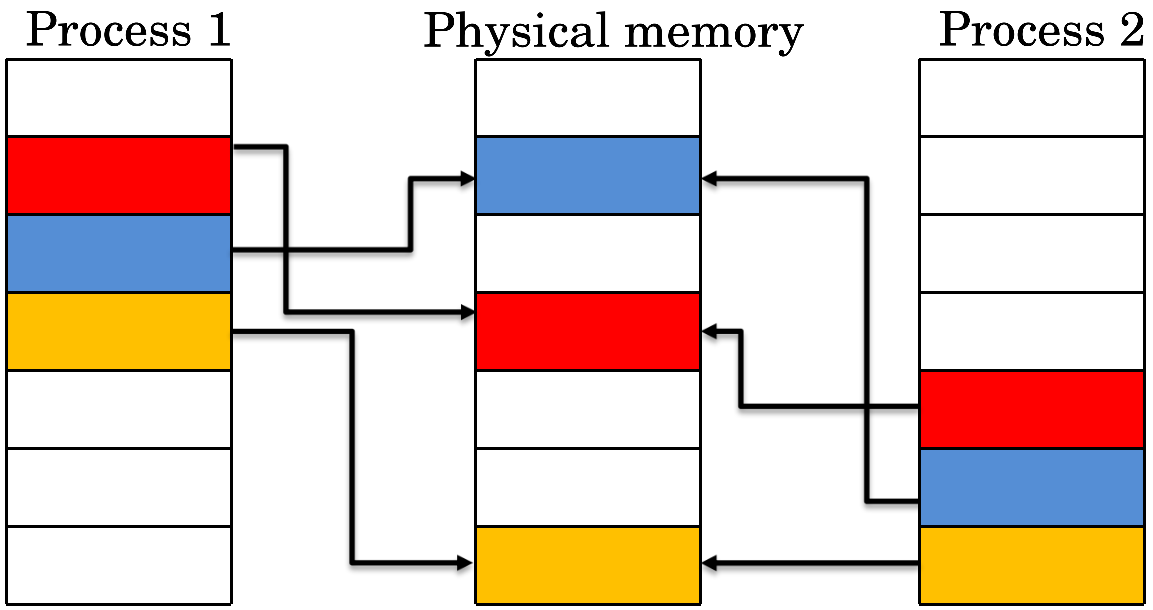
SWAPPING MEMORY

(Book Chapter 21)

PAGING USES: SHARED MEMORY

- Exploit level of indirection between VA and PA
- Regions of two separate processes' address spaces map to the same physical pages
  - read/write: access to share data
  - execute: shared libraries!
- Separate PTEs per process
  - Easy to provide different access privileges

SHARED MEMORY VISUALIZATION

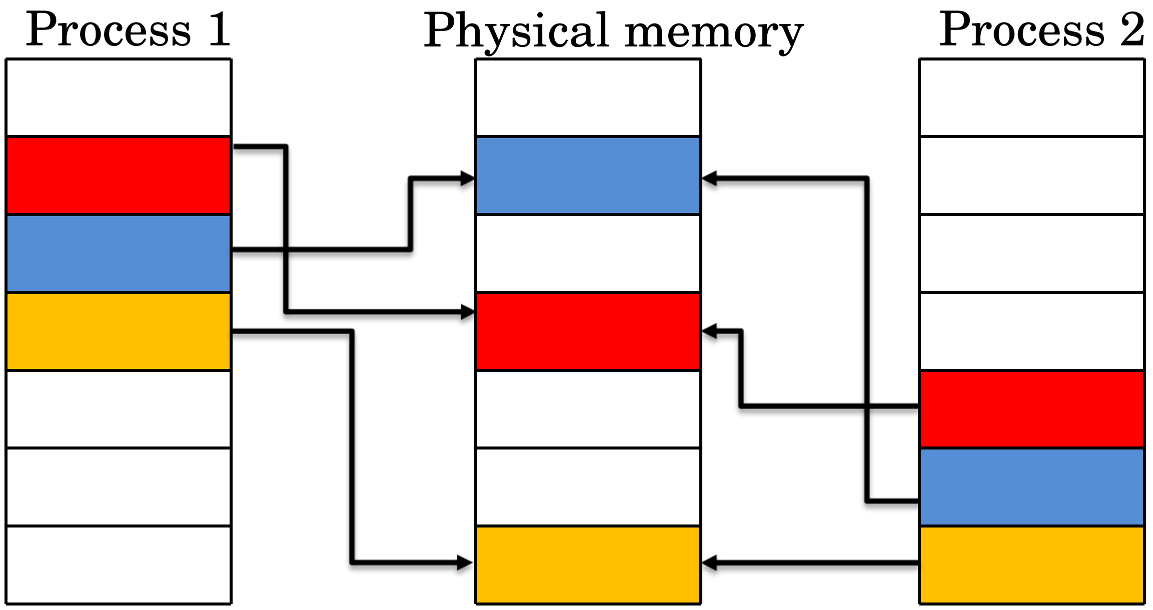


Paging Uses: Copy-on-Write

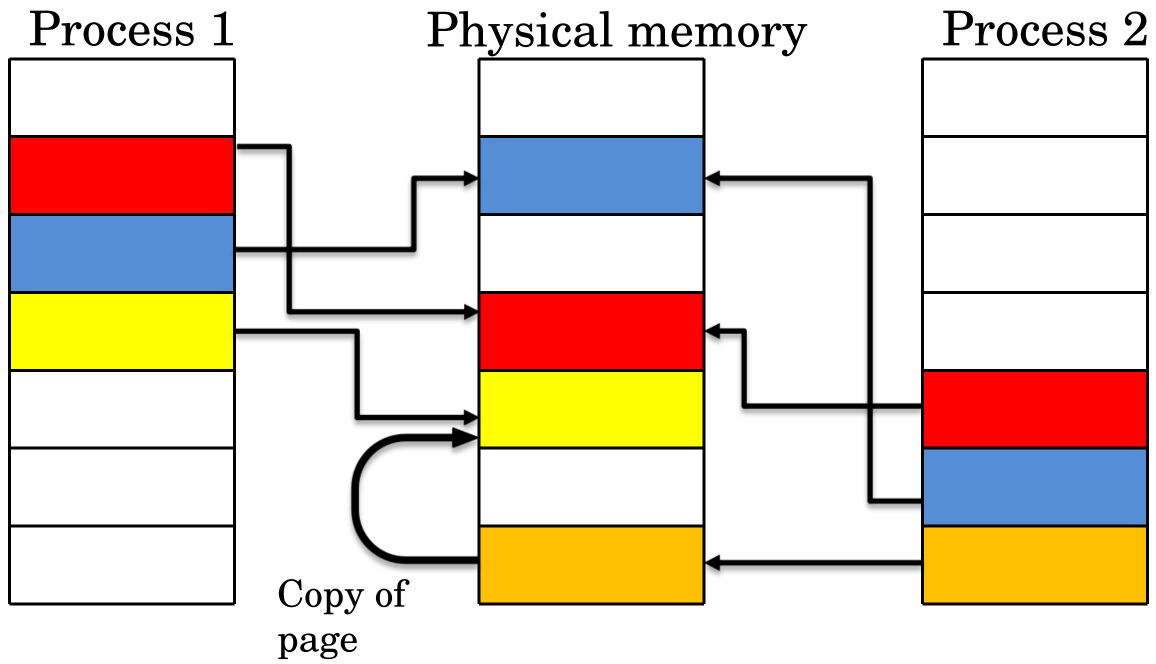
Copy-on-write (COW)

- Do not copy all pages; rather share the mappings
- Allows sharing of pages before changes are made
- Child process
  - Read-only mapping
  - On a write, protection fault occurs; copy page and resume
- Allows doing `fork()` very efficiently

COW – BEFORE MODIFICATION



COW – AFTER MODIFICATION

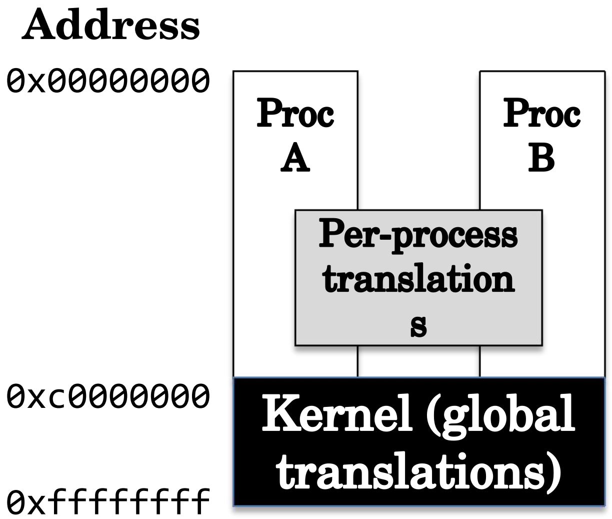


Side-note: User vs Kernel Addresses

- Low region of address space if private per-process memory
- High region reserved for kernel use and has same translations for all processes

Challenge: How to avoid accessing kernel memory

- Privileged bit in PTE marks high region as only accessible when in privileged mode



SWAPPING MOTIVATION

OS Goal: Allow execution even when not enough physical memory available

- Single process with very large address space
- Multiple processes whose total address spaces are large

Process execution should be independent of the amount of physical memory

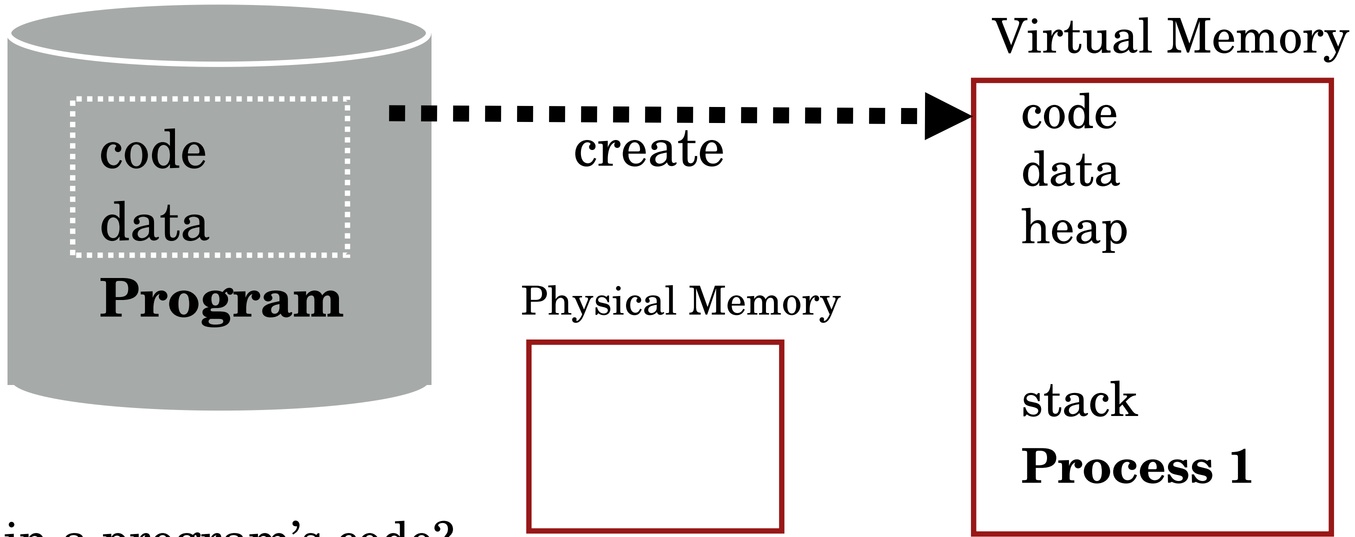
- Correctness, if not performance

Virtual memory: OS provides illusion of more physical memory

Why does swapping work?

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

WORKLOAD HELP



What is in a program's code?

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

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

Paging in and out

WORKLOAD: LOCALITY OF REFERENCE

Leverage locality of reference within processes

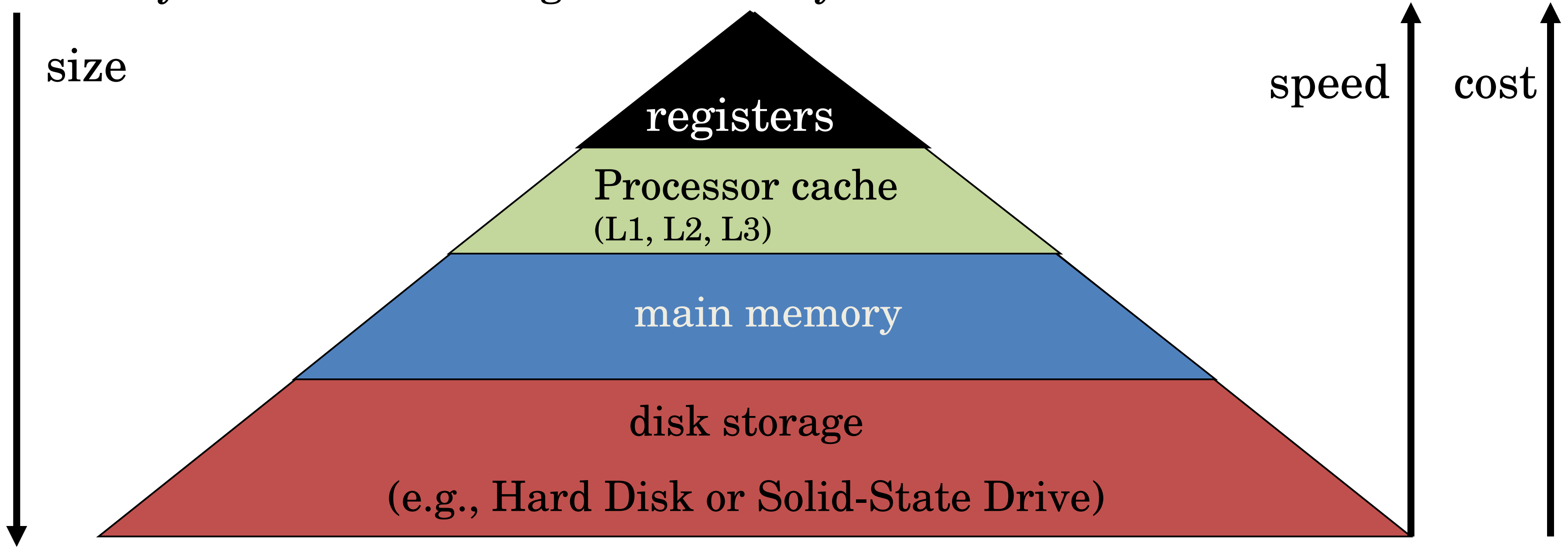
- Spatial: reference memory addresses **near** previously referenced addresses
- Temporal: reference memory addresses that have referenced in the **past**
- Processes spend majority of time in a small portion of their code
    - Estimate: 90% of time in 10% of code

Implication:

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

HARDWARE HELP: MEMORY HIERARCHY

Leverage memory hierarchy of machine architecture
Each layer acts as “backing store” for layer above



What entity controls the placement of data into each layer?

SWAPPING INTUITION

Idea: OS keeps unreferenced pages on disk (HDD or SSD)

- Slower, cheaper backing store than memory

Process can run even when not all its pages are loaded into main memory

- OS and hardware cooperate to make large disk seem like memory
- Same behavior (correctness) as if all of address space in main memory

Requirements:

- OS must have a **mechanism** to identify location of each page in address space in memory or on disk
- OS must have a **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 capacity, fast performance, expensive cost
- Disk (backing store): Large capacity, slow performance, inexpensive
- Nothing (error): Infinite size, infinitely fast, no cost

Extend page tables with extra bits: present and dirty

- Permissions (r/w), valid, present, dirty in every PTE
- Page in memory: present bit set
- Page on disk: present bit cleared
    - PTE points to block on disk
    - Causes trap into OS when page is referenced (page fault)
- Page in memory, but in-memory contents don’t match those on disk: dirty bit set