

Hello!

# MEMORY: SMALLER PAGE TABLES AND SWAPPING

Shivaram Venkataraman

CS 537, Spring 2023

# ADMINISTRIVIA

Project 3 is due **Monday**. → *start now if you haven't*

Project 4: Scheduling out next.

Midterm I: In class midterm, Multiple choice → *March 2 . Virtualization*

Soon: Practice exams, Review session details

*CPU / Mem*

# AGENDA / LEARNING OUTCOMES

## Memory virtualization

What are the challenges with paging ?

How we go about addressing them?

How we support virtual mem larger than physical mem?

What are mechanisms and policies for this?

**RECAP**

# PROS/CONS OF PAGING

## Pros

No external fragmentation

- Any page can be placed in any frame in physical memory

Fast to allocate and free

- Alloc: No searching for suitable free space
- Free: Doesn't have to coalesce with adjacent free space

## Cons

Additional memory reference

- MMU stores only base address of page table

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
- Entry needed even if page not allocated ?

for every process

Page Table

# TLB SUMMARY

Pages are great, but accessing page tables for every memory access is slow

Cache recent page translations → TLB

- MMU performs TLB lookup on every memory access

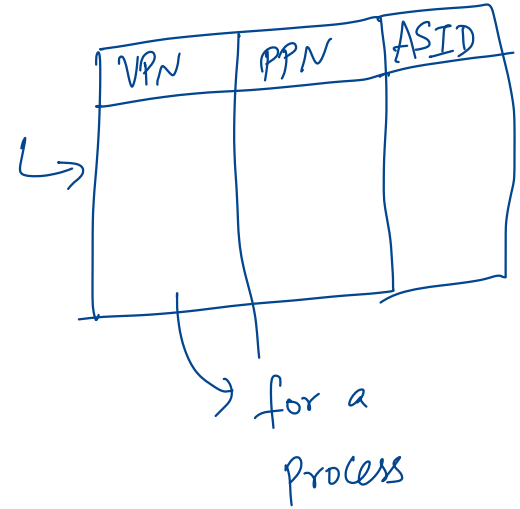
TLB performance depends strongly on workload

↳ *Sequential, locality*

TLBs increase cost of context switches

- Flush TLB on every context switch
- Add ASID to every TLB entry

In different systems, hardware or OS handles TLB misses



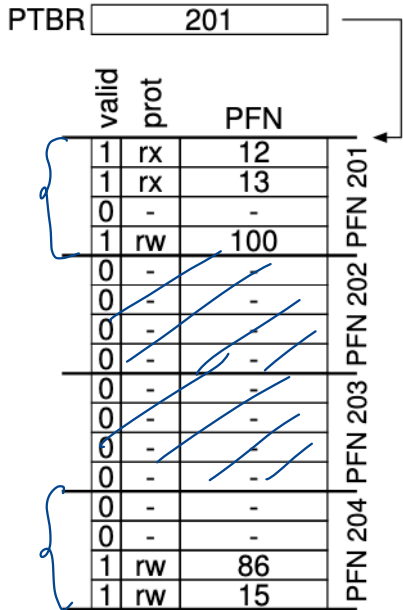


# MULTILEVEL PAGE TABLES

Large array

Tree-like  
(B-Tree)

Linear Page Table



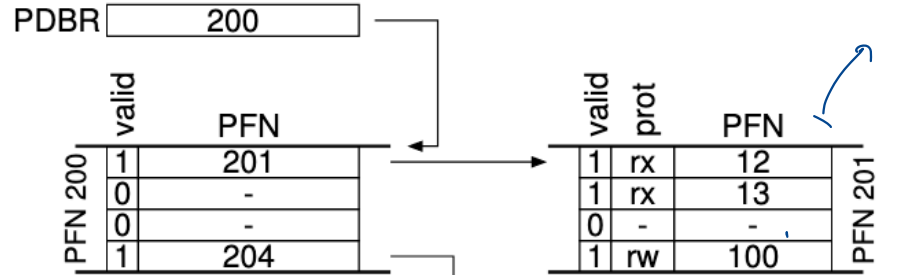
Creates multiple levels of page tables

Only allocate page tables for pages in use

Allow page table to be allocated non-contiguously

2 Pages useful, 2 empty

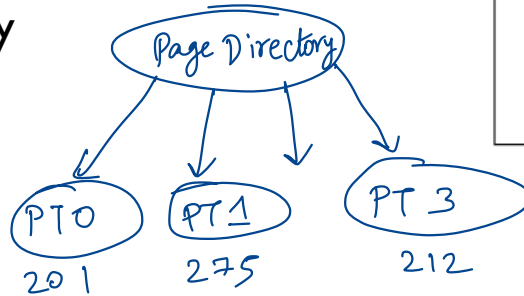
Multi-level Page Table



The Page Directory

[Page 1 of PT: Not Allocated]

[Page 2 of PT: Not Allocated]



0	-	-	PFN 204
0	-	-	
1	rw	86	
1	rw	15	



# MULTILEVEL TRANSLATION EXAMPLE

page directory

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

page of PT (@PPN:0x92)

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

translate 0x01ABC

outer page = 0  
 PT @ 0x3 is inner PT

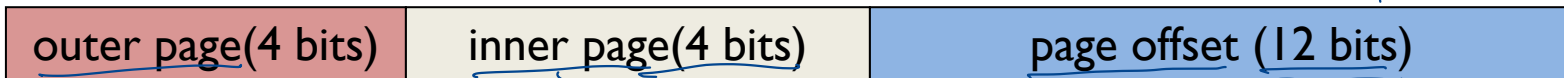
inner page = 1

0x23 < ABC >

↪ final PA

↪ 4KB

20-bit address:

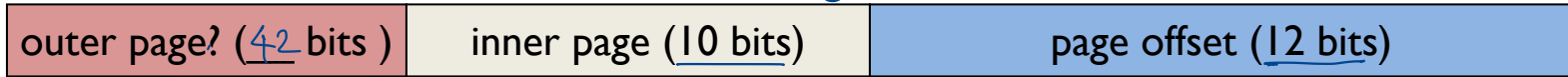


# MORE THAN 2 LEVELS?

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

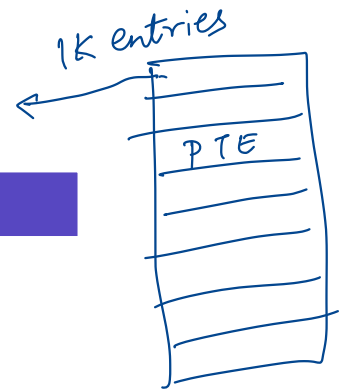
*Inner PT fits in one page*

**64-bit address:**



Solution:

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



How large is virtual address space with 4 KB pages, 4 byte PTEs, (each page table fits in page)

1 level:  $1K \times 4KB = 4MB$

2 levels:  $1K \times 1K \times 4KB = 4GB$

$4KB / 4 \text{ bytes} \rightarrow 1K \text{ entries per level}$

# EXAMPLE: X86-64

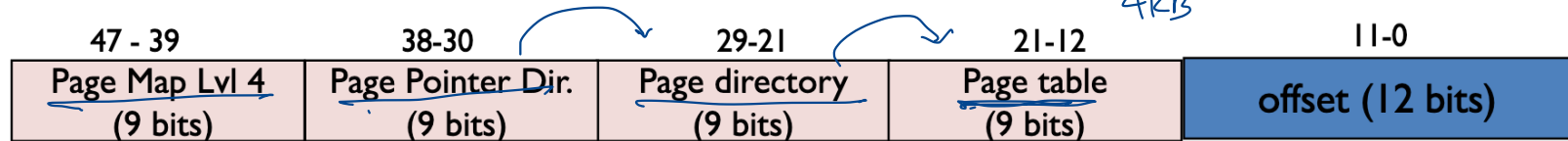
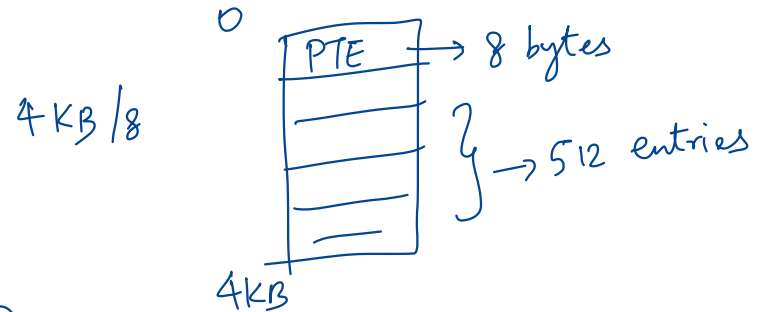
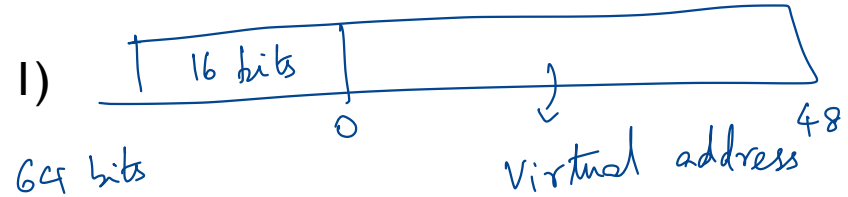
Virtual address: 48 bits

– Upper 16 bits are sign extended (all 0, all 1)

Physical address: 48 bits

PTE size = 48 bits + metadata = 8 bytes

PTEs/4kb page = 512 = 9bits



↓  
7 bits if  
46

4 level paging scheme

→ 4 mem access

# FULL SYSTEM WITH TLBS

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	<u>0xbb</u>	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: addl \$0x5, %edx

*3 mem access for translation = 4 physical accesses*  
*1 for instruction*

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

*1 mem access for instruction fetch*

# INVERTED PAGE TABLE

Pros: space savings

Cons:

Only store entries for virtual pages w/ valid physical mappings

↓  
valid

Naïve approach:

Search through data structure  $\langle \text{ppn}, \text{vpn} + \text{asid} \rangle$  to find match

Too much time to search entire table

VPN	PPN
1	
2	
33	
34	

# INVERTED PAGE TABLE

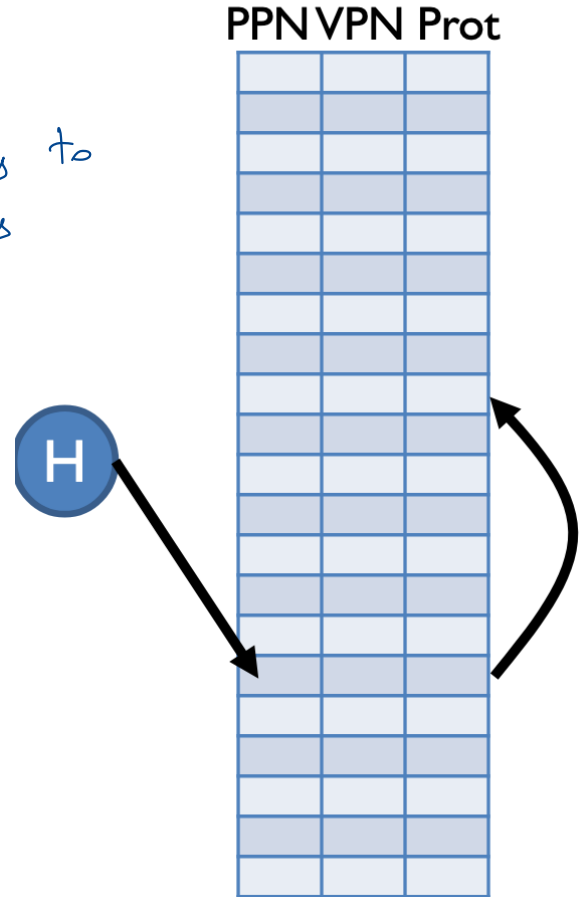
Hash table: VPN + Asid

Better:

- Find possible matches entries by hashing vpn+asid
  - Use chaining to handle collisions
  - Smaller number of entries to search for exact match
- } OS needs to do this

Managing inverted page table requires software-controlled TLB

Used in IBM POWER and Intel Itanium, but complicated



# QUIZ 12

<https://tinyurl.com/cs537-sp23-quiz12>



Consider a virtual address space of 16KB with 64-byte pages.

1. How many bits will we have in our virtual address for this address space?

$$\log_2(16 \text{ KB}) = \log_2(2^4 \cdot 2^{10}) = 14 \text{ bits}$$

2. What is the total number of entries in the Linear Page Table for such an address space?

$$\frac{\text{addr space}}{\text{page size}} = \frac{2^4 \cdot 2^{104}}{2^6} = 2^8 = 256$$



3. Consider a two-level page table now with a page directory. How many bits will be used to select the inner page assuming PTE size = 4 bytes?

$$14 \text{ bits} \rightarrow 6 \text{ bits offset} = \log_2(64)$$

16 entries in my inner PT  $\rightarrow$  4 bits to select

# QUIZ12

page directory

PPN	valid
0x3	0
-	1
-	2
-	3
-	4
-	5
-	6
-	7
-	8
-	9
-	10
-	11
-	12
-	13
-	14
0x92	15

*Typo*

20-bit address:

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
-	0
-	0

page of PT (@PPN:0x92)

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

translate 0xFEED0

outer page = F  
bits

get 0x92

next inner page bits  
= E

gives us 0x55EED0

2 mem accesses

outer page(4 bits)

inner page(4 bits)

page offset (12 bits)



# EFFECT OF PAGE SIZE

Assume **4KB** pages

32 TLB entries

```
int sum = 0;
int a[1024*1024];
for (i=0; i<1024*1024; i++) {
    sum += a[rand() %
            (1024*1024)];
}
```

*1024 pages*  
*1M entries*

*524,288 ints in 1 page*  
*⇒ 2 pages*

Assume **2MB** pages

32 TLB entries

Miss rate of TLB: #TLB misses / #TLB lookups

#TLB lookups? number of accesses to a = *1M lookups*

Chance of a TLB miss?

$$= 1 - 32/1024$$

#TLB lookups? number of accesses to a = *1M lookups*

Chance of a TLB miss?

$$= 2/1M$$

# LARGE PAGES (HUGE PAGES)

TLB reach: how much memory can be accessed without a TLB miss?

1000 entries 4KB pages → 4MB

Large pages

1000 entries, 2MB pages → 2GB!

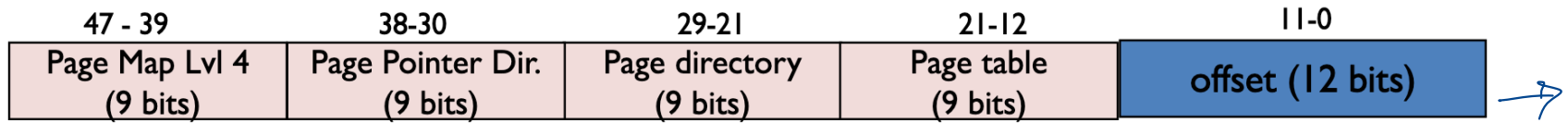
How to use? → *linux*

- Programmer requested: `mmap(MAP_HUGE)` returns huge pages
- Transparent Huge Pages (THP, in Linux)
  - OS uses huge pages when available for > 2MB allocations

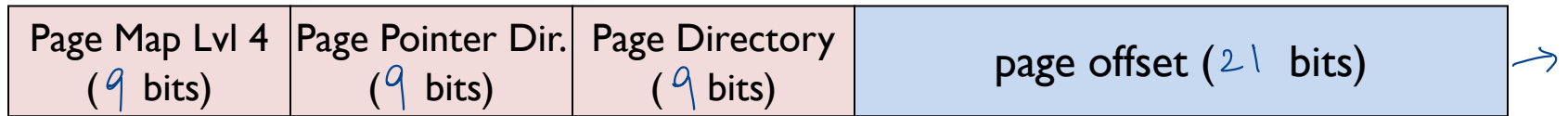
# TRANSLATING LARGE PAGES

HugePages saves TLB entries. But how does it affect page translation?

4KB pages: 4 levels  $\rightarrow$  4 memory accesses

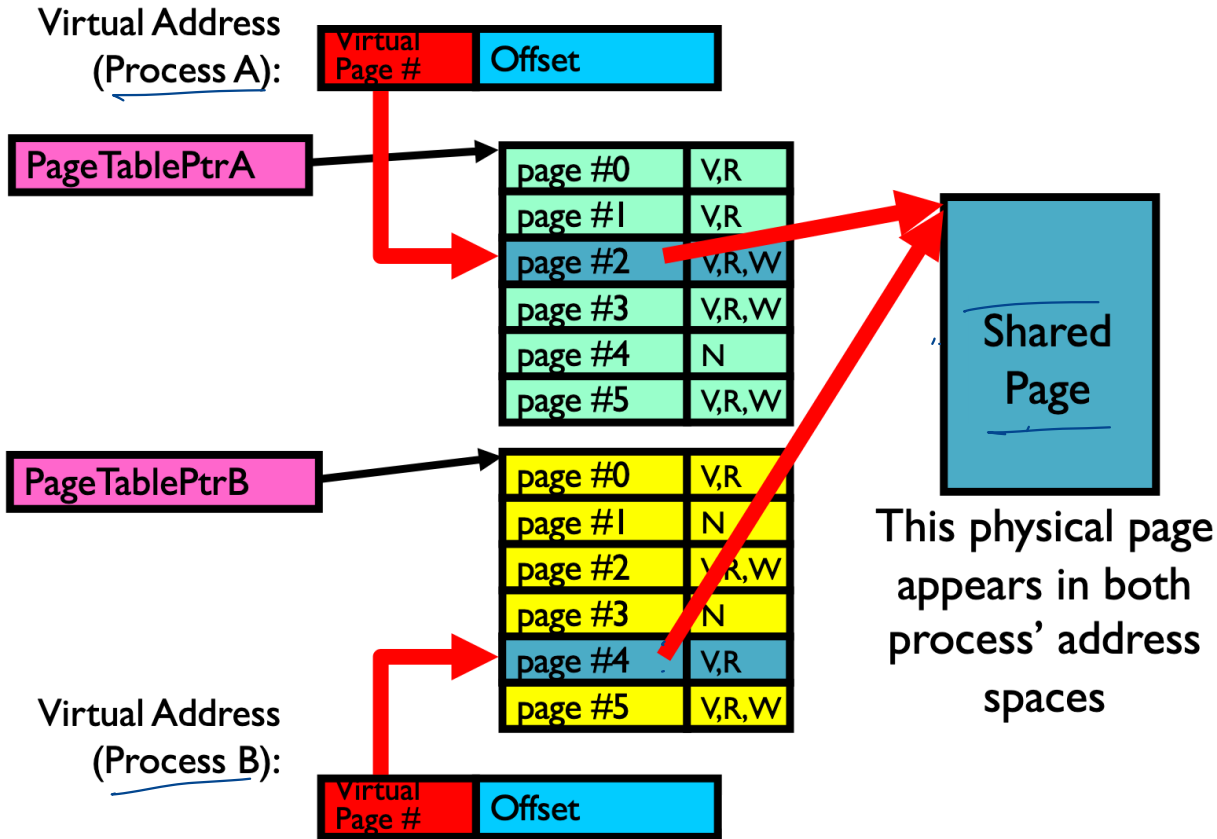


2MB pages:  $\log(2^{21}) =$



Downside: Fragmentation within page

# SHARING WITH PAGE TABLES



## Where is page sharing used?

- Kernel data mapped into each process
- Different processes running the same binary
- User-level system libraries
- Shared pages as IPC

# SUMMARY: BETTER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

Many options 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

Large pages can reduce TLB use and number of accesses for translation

**SWAPPING**

# MOTIVATION

OS goal: Support processes when not enough physical memory

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

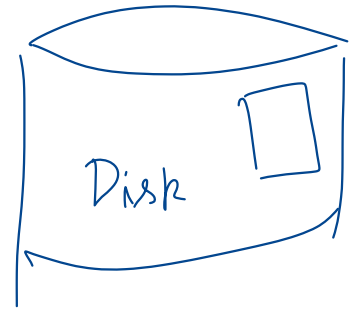
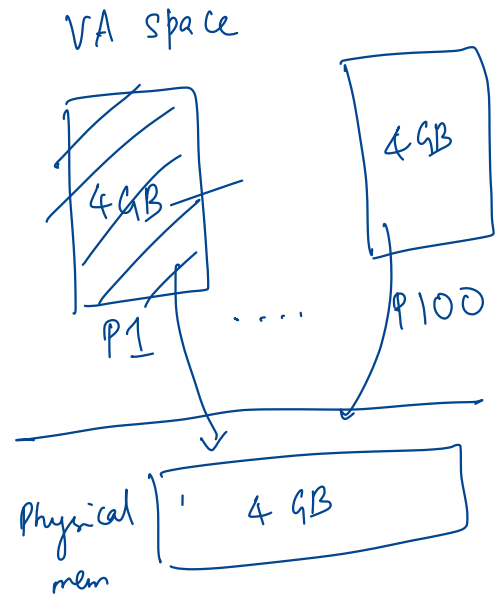
User code should be independent of amount of physical memory

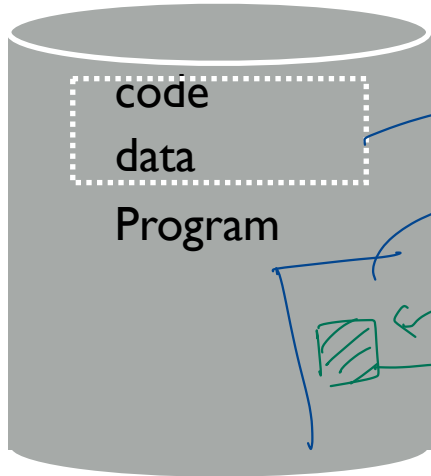
- Correctness, if not performance

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)



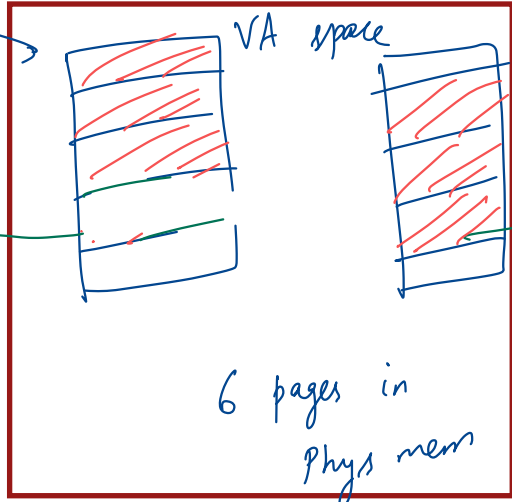


code  
data  
Program

Swap space 1G

temp.  
on disk

## Virtual Memory



VA space

want to  
allocate

6 pages in  
phys mem

Swapping



# 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

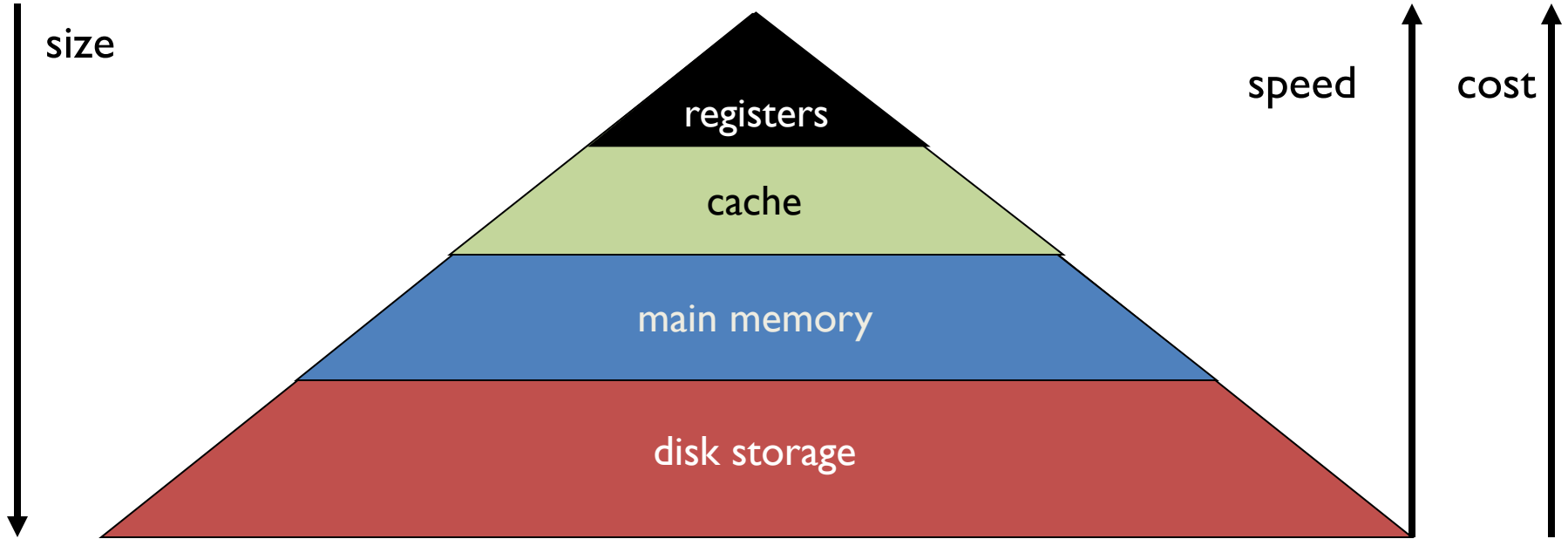
Implication:

- 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
  - Causes trap into OS when page is referenced
  - Trap: page fault

Disk



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

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 ...

- Hardware or OS walk page tables
- If PTE designates page is present, then page in physical memory (i.e., present bit is cleared)

Else

- 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

# 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)

**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

# 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
















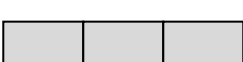
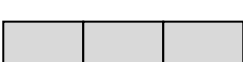



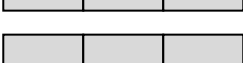
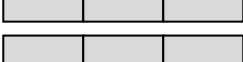
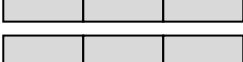
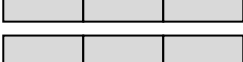


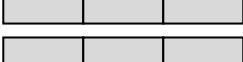
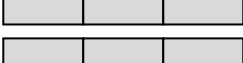
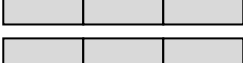

- Intuition: Use past to predict the future
- Advantages: With locality, LRU approximates OPT
- Disadvantages:
  - Harder to implement, must track which pages have been accessed
  - Does not handle all workloads well

# PAGE REPLACEMENT

Three pages  
of physical  
memory

Page reference string:  
DDBBACBDBD

Metric:  
Miss count

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

# 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?

# 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

# CLOCK ALGORITHM

## Hardware

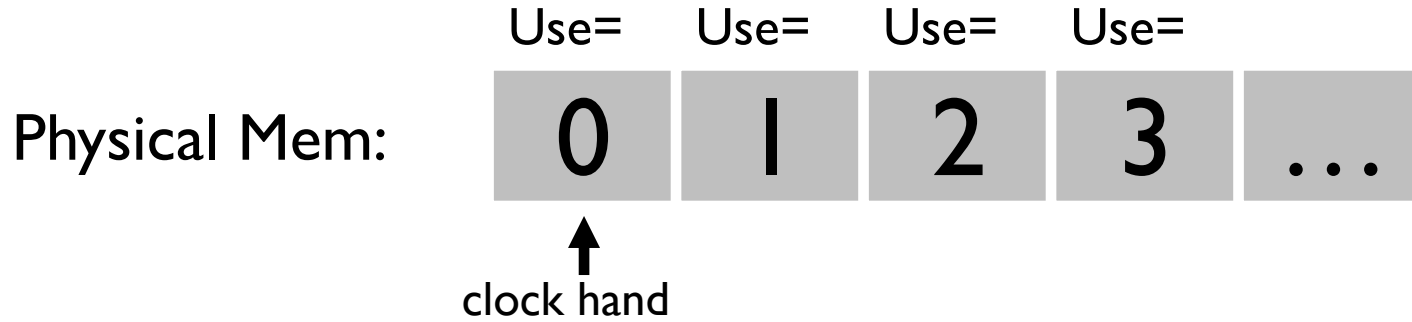
- Keep use (or reference) bit for each page frame
- When page is referenced: set use bit

## 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



# 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

# 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)

# NEXT STEPS

Project 3: Due soon!