CS 423
Operating System Design: Midterm Review
Mar 24

Ram Kesavan

Logistics

Midterm Survey (~60% of class has responded): closes Wed 3/25 11:59pm CT

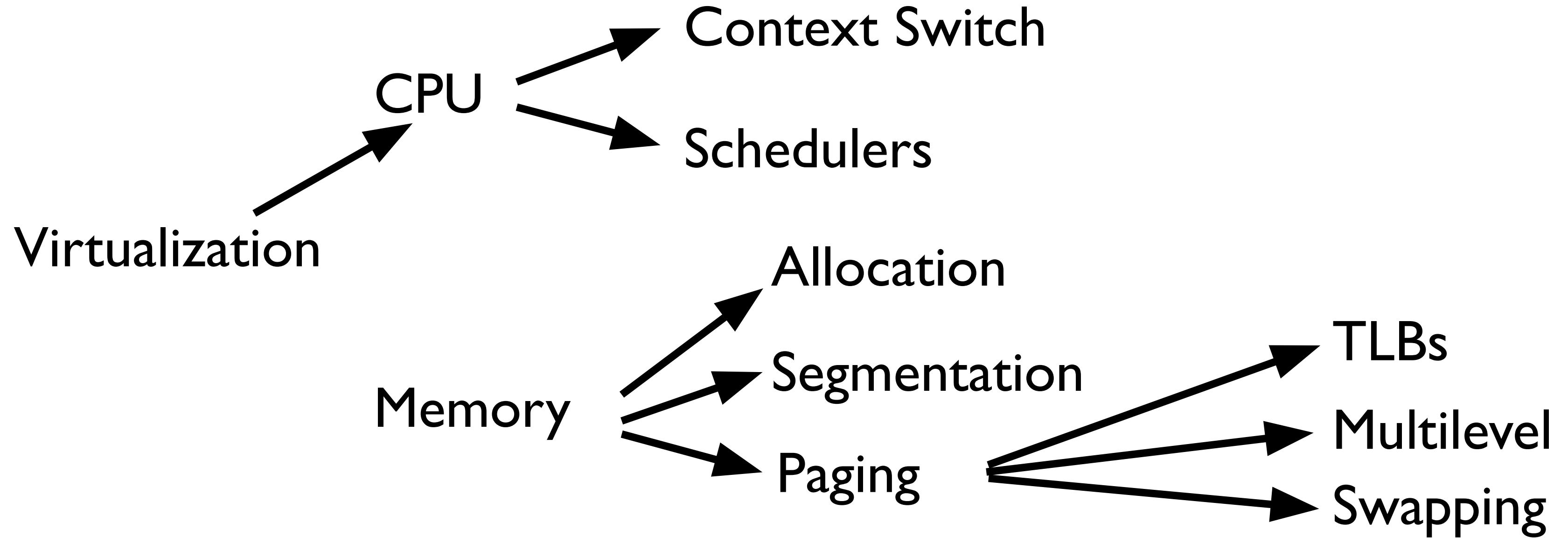
MP2 due tonight 3/24 by 11:59pm CT (1 day extension)

Thursday: midterm exam: 2pm–3:15pm

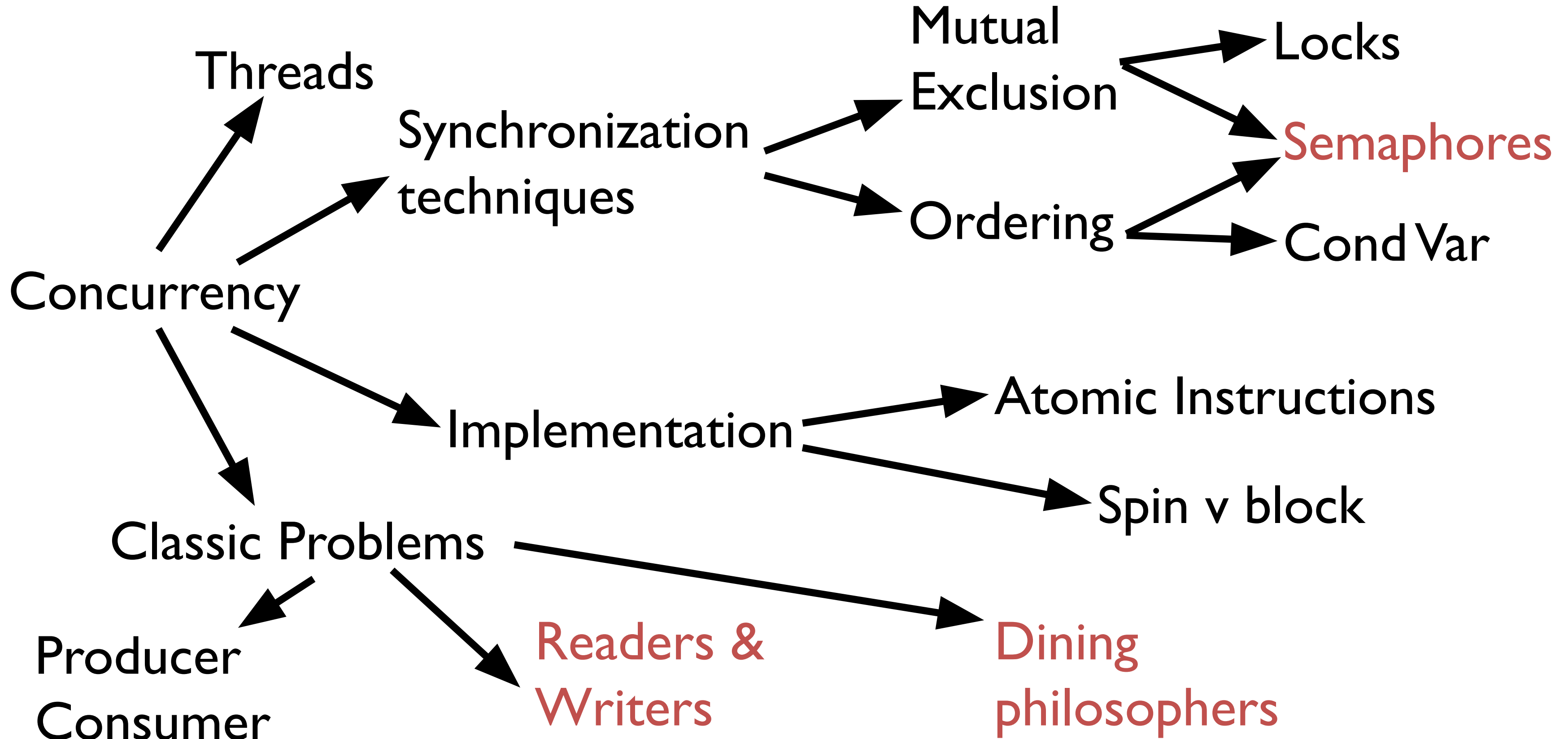
- Last name starting with A–L: **DCL 1310**
- Last name starts with M-Z: **Everitt Lab 2310**
- No books, notes, cheat-sheet, or devices.
- One standard calculator allowed (cannot be a networked device)
- MCQs (no partial credit) and 4 long-form questions

Finals: 5/13 7–10pm; DCL 1320. If you have a conflict let the staff know ASAP.

VIRTUALIZATION: EASY PIECE 1



Concurrency: Easy Piece 2



Miscellaneous

Notation: KiB, MiB, GiB vs KB, MB, GB

Page table is all in physical memory;

- i.e., MMU doesn't consult TLB when walking the page table

We learned how to implement a mutex BUT not a cv

- We only learned how to use cv
- PracticeQs has an implementation of a cv

Non Blocking System Call

Process A

Run main() ...
Call system call
trap into OS

move to kernel mode
regs(A) $\longrightarrow$ k-stack(A)
jump to trap handler

Handle the trap
Do work of syscall
return-from-trap

regs(A) $\longleftarrow$ kstack(A)
move to user mode
jump to PC past trap instruction

Process A

Operating System | Hardware | Program

Blocking System Call

Process A

Run main()…

System call

Trap into OS

move to kernel mode

regs(A) $\rightarrow$ k-stack(A)

jump to trap handler

handle the trap

do work of syscall

if blocking:

call switch()

kernel regs(A) $\rightarrow$ proc-struct(A)

kernel regs(B) $\leftarrow$ proc-struct(B)

switch to k-stack(B)

return-from-trap (into B)

regs(B) $\leftarrow$ k-stack(B)

move to user mode

jump to B's IP

Process B

Operating System

Hardware

Program

Timer Interrupt + Switch

Process A
Running…

timer interrupt

move to kernel mode

regs(A) → k-stack(A)

jump to trap handler

Handle the trap

Call switch() routine

kernel regs(A) → proc-struct(A)

kernel regs(B) ← proc-struct(B)

switch to k-stack(B)

return-from-trap (into B)

regs(B) ← k-stack(B)

move to user mode

jump to B's IP

Process B

Operating System | Hardware | Program

TLB & Page Fault

Process A

Running…

TLB miss

Page-fault Trap to OS

move to kernel mode

regs(A) → k-stack(A)

jump to trap handler

handle the trap

work of page fault handler

If (hard), kick off read IO

call switch()

kernel regs(A) → proc-struct(A)

kernel regs(B) ← proc-struct(B)

switch to k-stack(B)

return-from-trap (into B)

regs(B) ← k-stack(B)

move to user mode

jump to B's IP

Process B

Program

Operating System | Hardware

Scheduling

Process stops running because

- Timer interrupt: OS decides its run long enough
- Blocking system call

Scheduling policies: metrics like turnaround time, response time

FIFO, SJF, STCF (preemptive version of SJF); the last 2 must know job time

- In reality don't know, so RR is a good scheme

Real-time systems: EDF, RMS

MLFQ: GENERAL PURPOSE SCHEDULER

Must support two job types with distinct goals

- “interactive” programs care about response time
- “batch” programs care about turnaround time

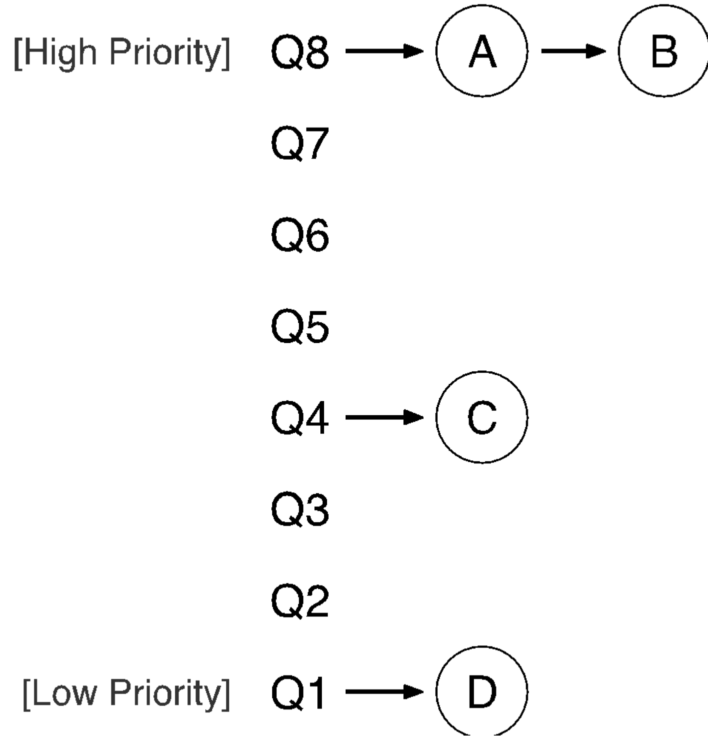
Approach:

Multiple levels of round-robin

Each level has higher priority than lower level

Jobs can be preempted

MLFQ EXAMPLE



"Multi-level" – Each level is a queue

Rules for MLFQ

1: If pri(A) > pri(B), A runs
2: If pri(A) == pri(B), A & B run in RR
3: Processes start at the highest pri
4: Processes get demoted after time-slice
5: After some time, move all jobs to highest pri

More Scheduling

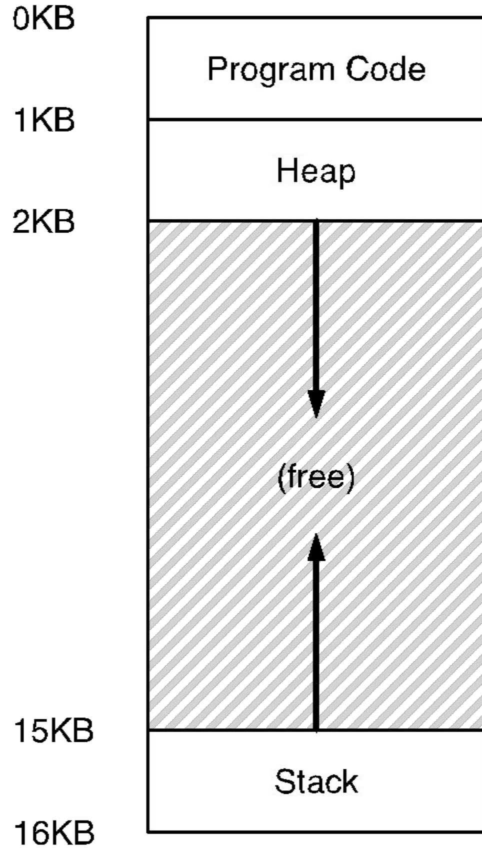
Proportional share in scheduling

- Lottery scheduling

Linux CFS

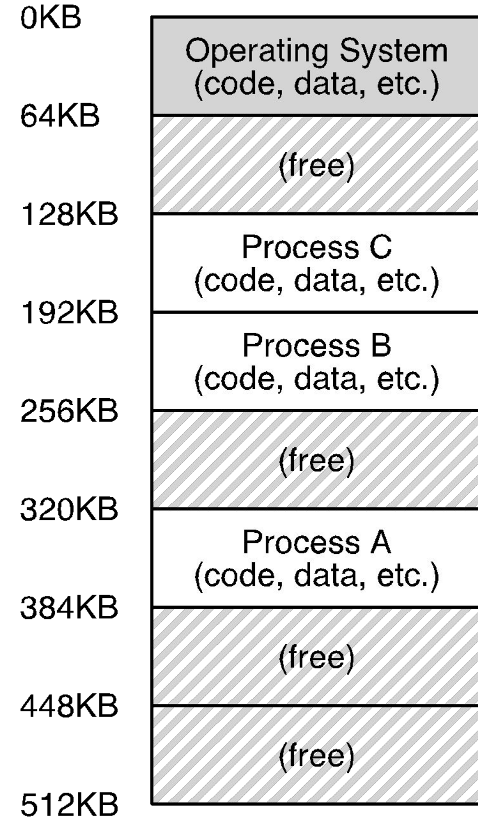
- Picks job with lowest *vruntime*
- Uses *sched_latency* to decide time slice per run
- Unix priority (nice) => weight; biases *vruntime* & *sched_latency*
  - time_slice = max(*min_granularity*, (*sched_latency*)/(#runnable-jobs))

ABSTRACTION: ADDRESS SPACE



Address Space:
Each process has its own set of addresses

OS aims to provide Illusion of private memory



HOW TO VIRTUALIZE MEMORY

Problem: How to run multiple processes simultaneously?
Addresses are “hardcoded” into process binaries
How to avoid collisions?

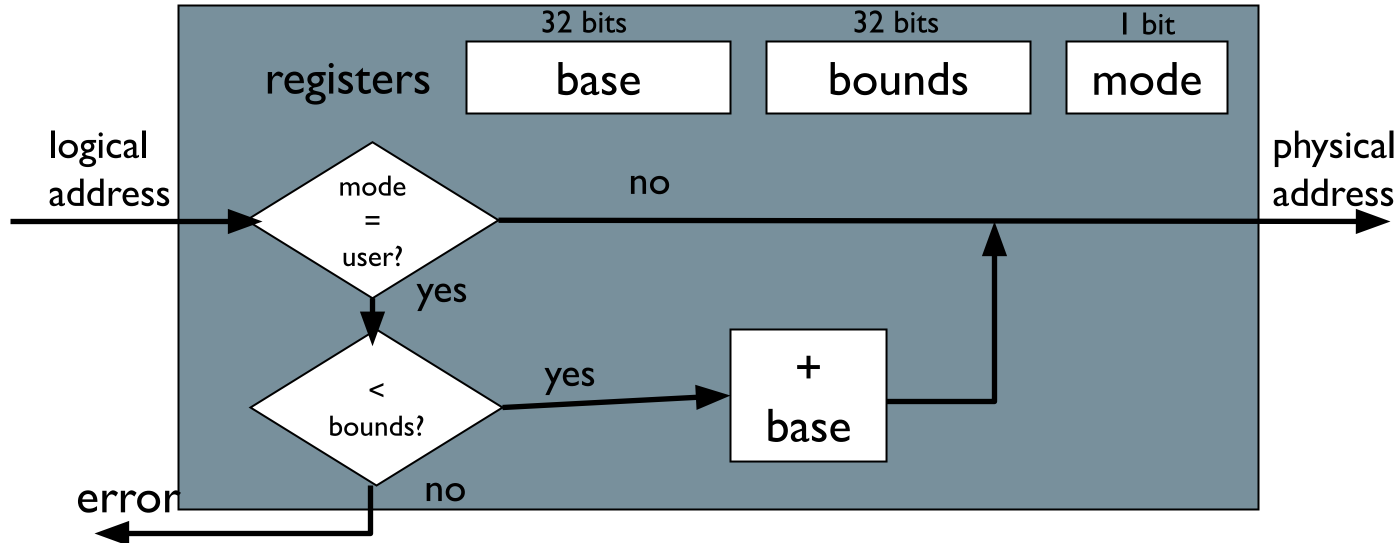
We studied many mechanisms; the most interesting/relevant are

1. Base + Bounds
2. Segmentation (base + bounds per segment)
3. Paging

Memory Management Unit

Translate every virtual memory access of a user process

- One MMU per CPU/core
- MMU contains registers/memory/TLB & computation logic for this



Segmentation

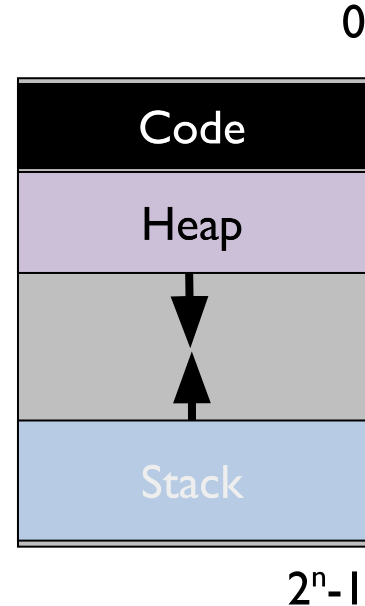
Divide address space into logical segments

- Each segment corresponds to logical entity in address space (code, stack, heap)

Each segment has separate base + bounds register

Each segment can independently:

1. Be placed in physical memory
2. Grow and shrink
3. Be protected (read/write/exec)

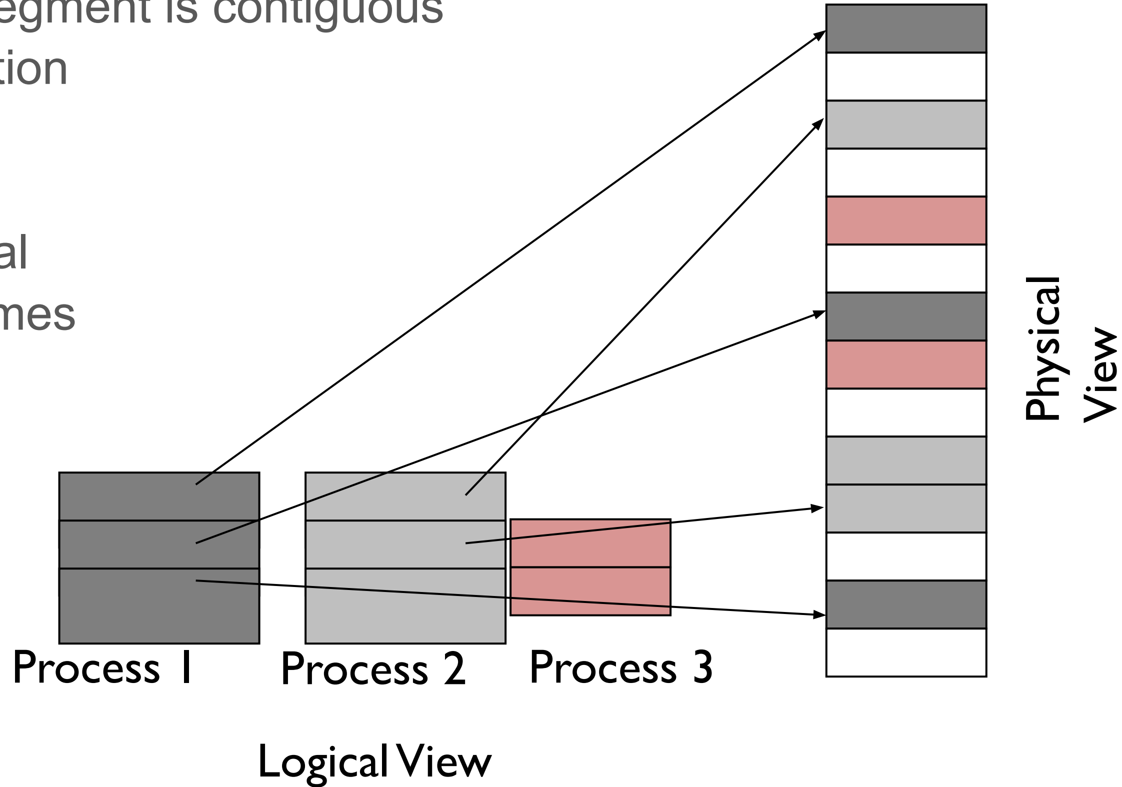


Paging

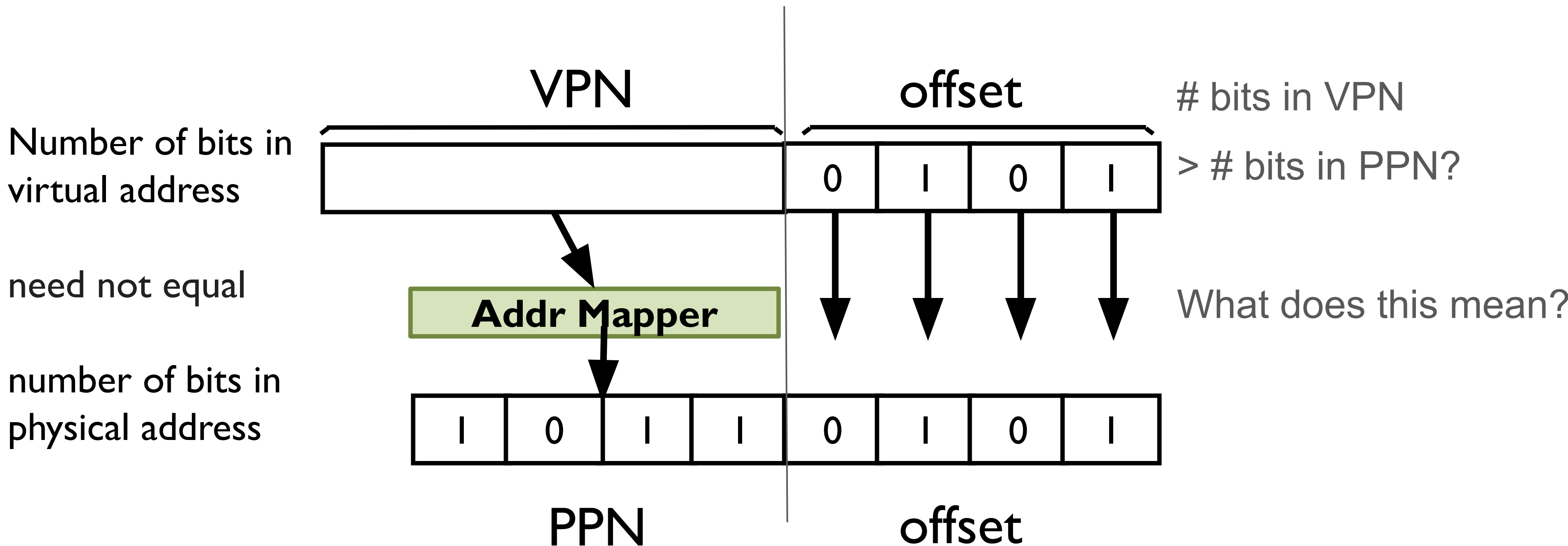
Goal: Eliminate requirement that segment is contiguous
Eliminate external fragmentation

Idea:
Divide address spaces and physical memory into fixed-sized pages/frames

Size: 2^n, Example: 4KB



VIRTUAL -> PHYSICAL PAGE MAPPING

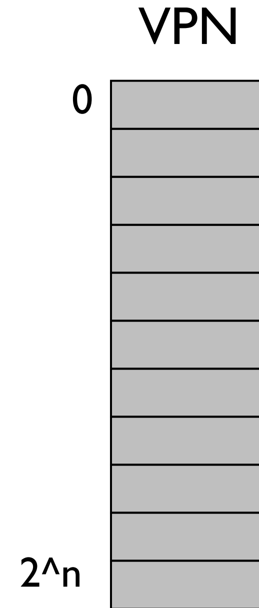


How should OS translate VPN to PPN/PFN?

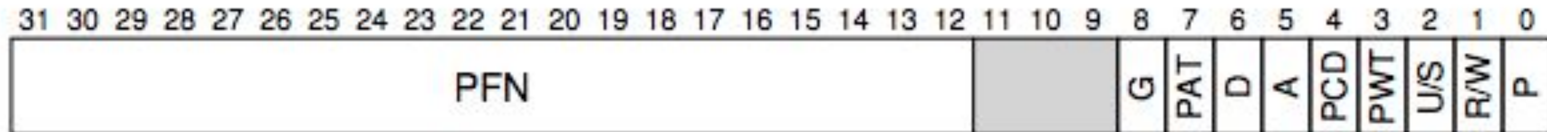
Linear Page Table

What is a good data structure ?

Simple solution: Linear page table aka *array*



A Single PTE:



Problems & Solutions

Additional memory reference to page table entry

- Inefficient, even if page table is stored in memory
- Extra memory accesses needed for each memory access!

  Solution: TLB

Space needed for page tables is too large

- Simple page table: requires PTE for each virtual page number
- Page tables must be contiguously allocated

  Solution: paging the page tables!

But how do we make it all fit in physical memory?

Solution: swap to storage

HW AND OS ROLES

If H/W handles TLB Miss:

CPU must know where page tables are

- CR3 register on x86
- Page table structure fixed and agreed upon between HW and OS
- HW “walks” the page table and fills TLB

If OS handles TLB Miss:

“Software-managed TLB”

- CPU traps into OS upon TLB miss.
- OS interprets page tables as it chooses
- Modify TLB entries with privileged instruction

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)

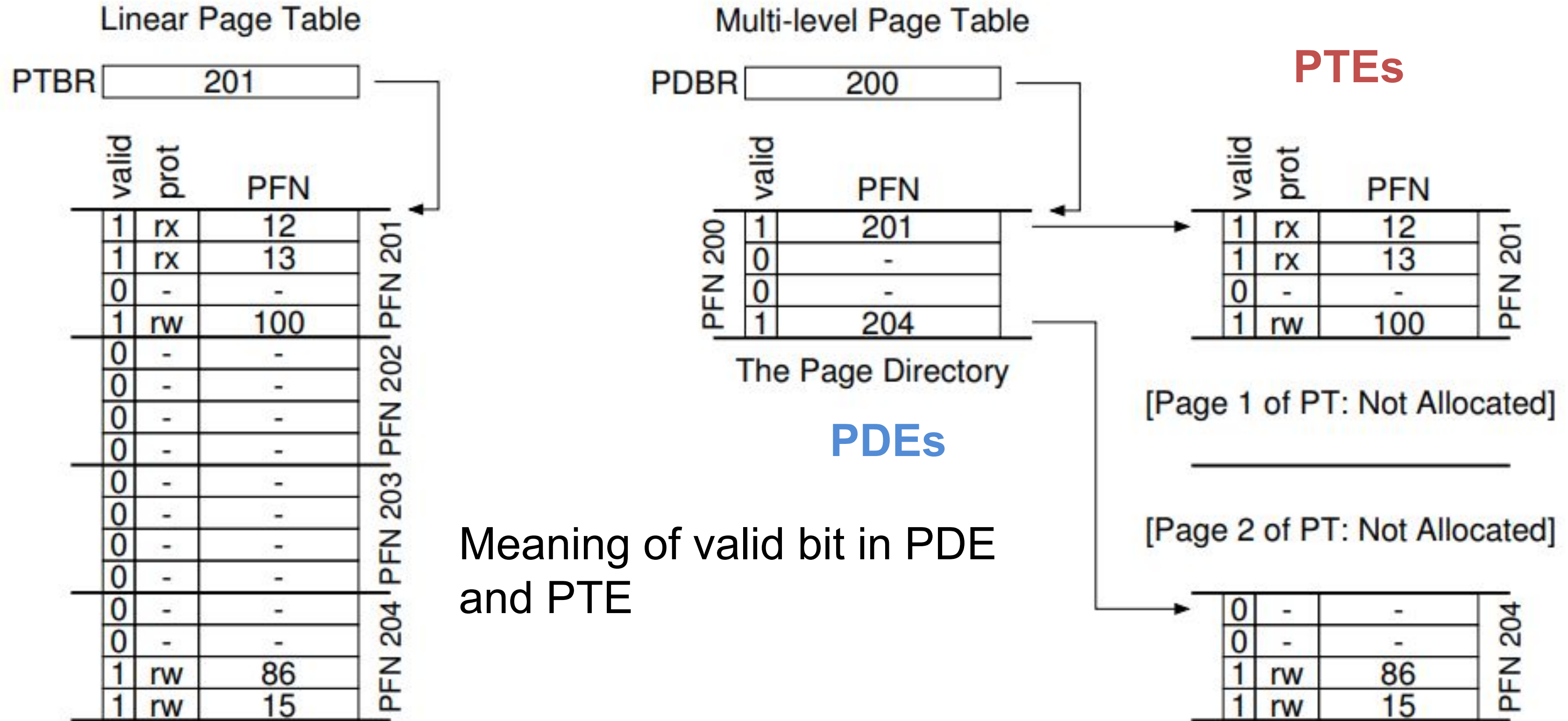
Multilevel Page Tables

Goal: Allow page table to be allocated non-contiguously

Idea: Page the page tables!

- Creates multiple levels of page tables; outer level “page directory”
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)

Multilevel Page Table – Key Idea



Meaning of valid bit in PDE and PTE

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 too

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

Soft vs. Hard Page Faults

Hard:

requires reading the page from storage (swap or not)

expensive

Soft: PTE could be valid & present!

page already in memory, but OS need to do some work, cheaper

Q: Give me an example

REVIEW: PROCESSES VS THREADS

```
int a = 0;
int main() {
  fork();
  a++;
  fork();
  a++;
  if (fork() == 0) {
    printf("Hello!\n");
  } else {
    printf("Goodbye!\n");
  }
  a++;
  printf("a is %d\n", a);
}
```

How many times will "Hello!\n" be displayed?

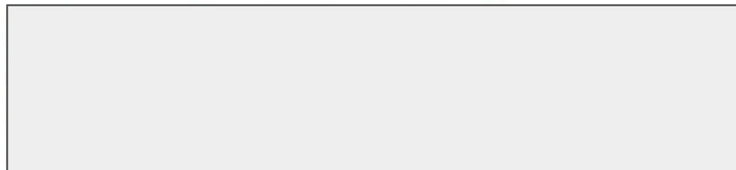
What will be the **final** value of "a" as displayed by the final line of the program?

REVIEW: PROCESSES VS THREADS

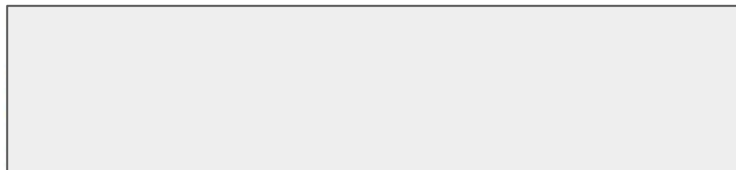
```
volatile int balance = 0;
void *mythread(void *arg) {
    int result = 0;
    result = result + 200;
    balance = balance + 200;
    printf("Result is %d\n", result);
    printf("Balance is %d\n", balance);
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final Balance is %d\n", balance);
}
```

How many total threads are part of this process?

When thread p1 prints "Result is %d\n", what value of `result` will be printed?



When thread p1 prints "Balance is %d\n", what value of `balance` will be printed?



Practice Q: Virtualization

PDF posted on course webpage

Q1.

Practice Q: Virtualization

Q2.

Practice Q: Virtualization

Q3.

Practice Q: Virtualization

Q4.

Practice Q: Concurrency

Note: we never covered CV implementation in class!

Q3.

GOOD LUCK!