UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

EXAM 1: REVIEW

Questions answered in this lecture:

What are some useful things to remember about virtualization?

ANNOUNCEMENTS

P2: Graded in handin and Canvas; major problems contact TAs

P3: Handin lessons: Problems with instructional machines at last minute

- Future assignments may be due at 5pm instead of midnight
- Turn in a reasonable version before the last minute
  - Small changes less problematic vs. no code at all
- Use "cp –p" (or "scp –p"): Keep modification time, access time

P4: Available later today: xv6 multi-level scheduling

- Choose your own project partner
- Will post form on Canvas if you want us to assign (binding)

MIDTERM 1

Two hours – 7:30 – 9:30 pm, Thursday 10/10

- Room Ingraham 19:
  If you are enrolled in Discussion Section 301 (Wed 11-11:50am)
- Room Ingraham B10:
  All other discussion sections

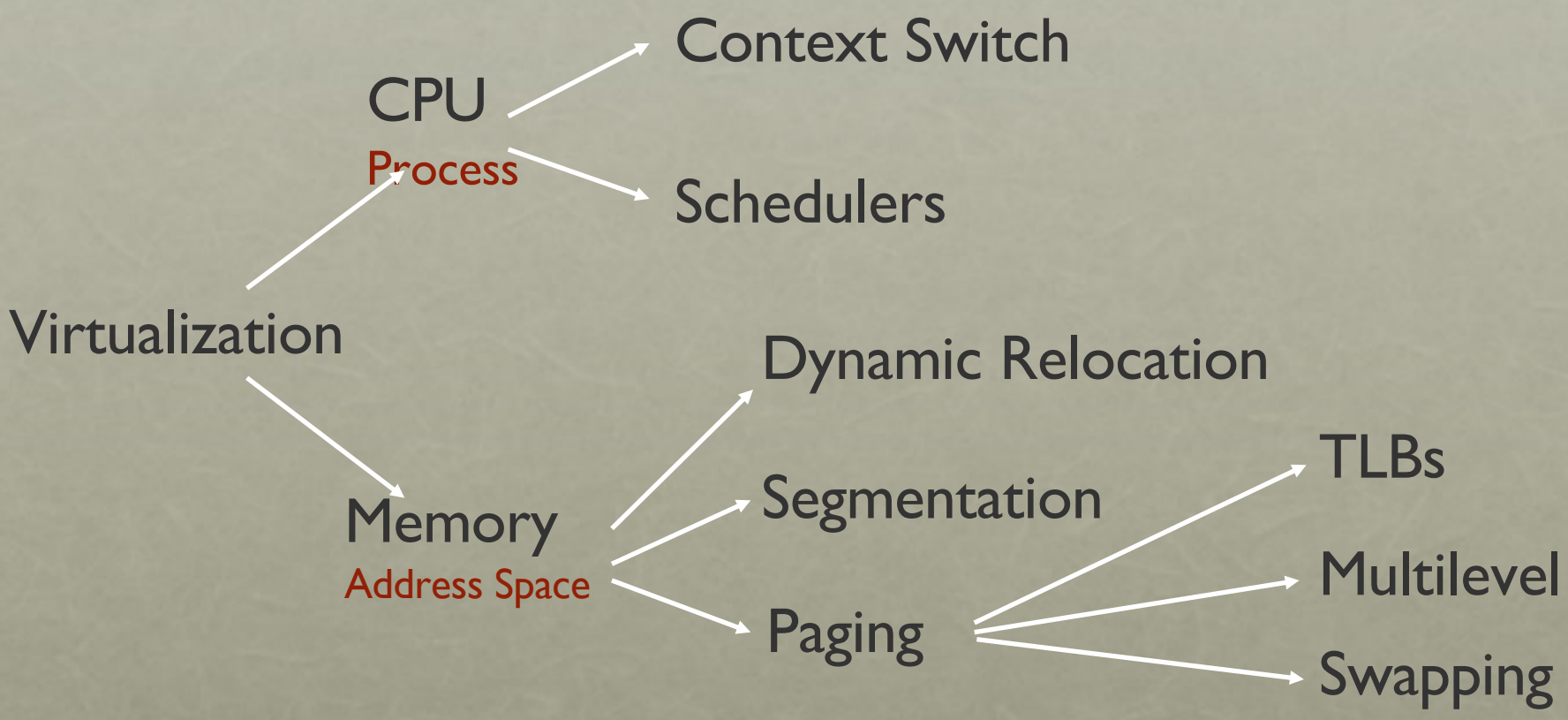
Bring #2 pencils and student id (need id number)

All multiple choice, fill in scantron bubbles

Covers everything so far in course:

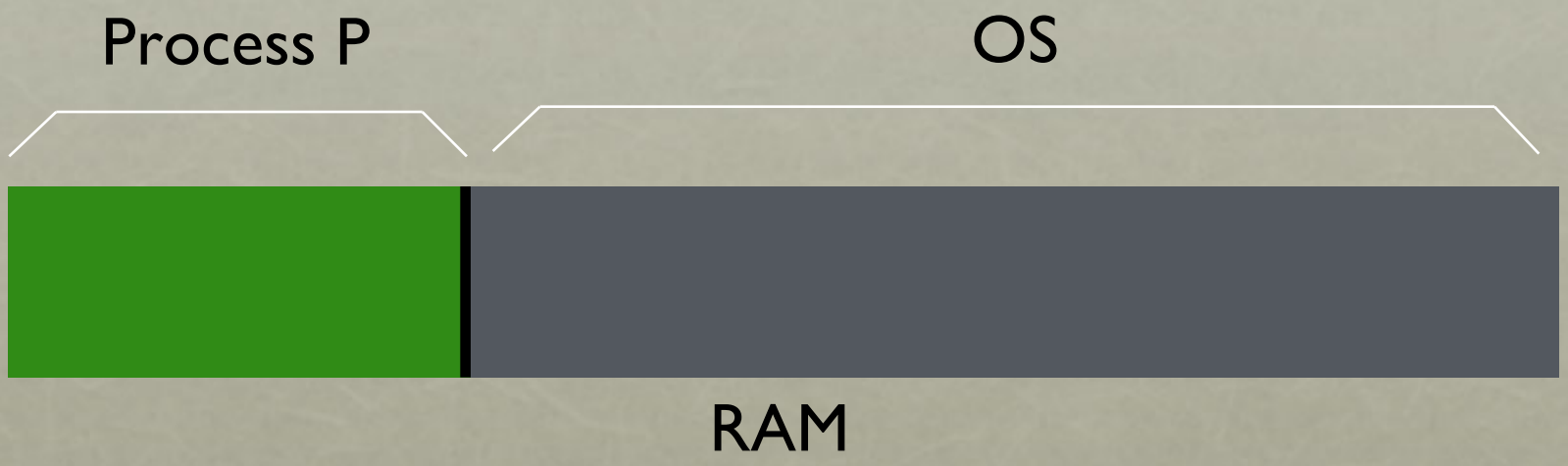
- Lectures including last Thursday (threads) + Reading + Quizzes + P1-3
- Chapters 1 - 26, excluding 10 (Multiprocessor Scheduling), 17 (Free-Space Management), and 23 (VAX/VMS Virtual Memory System)
- Look over sample exams

REVIEW: EASY PIECE 1



WHAT QUESTIONS DID YOU ASK?

HOW ARE SYSTEM CALLS PERFORMED?

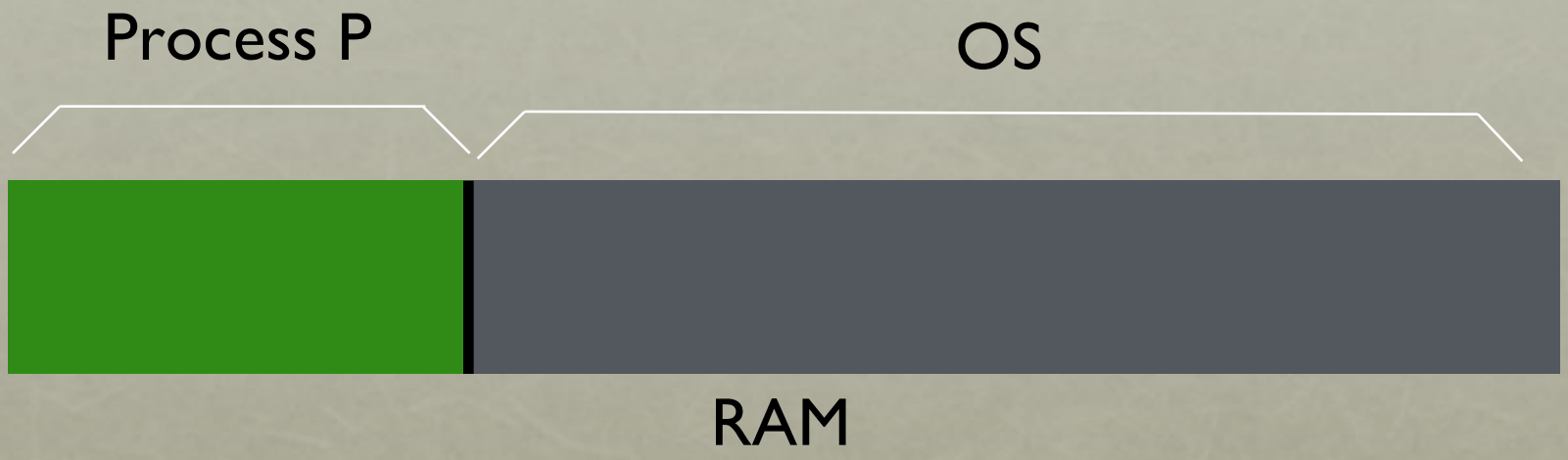


P can only see its own memory because of **user mode**
(other areas, including kernel, are hidden)

P wants to call read() but no way to call it directly

OS is not part of P's address space

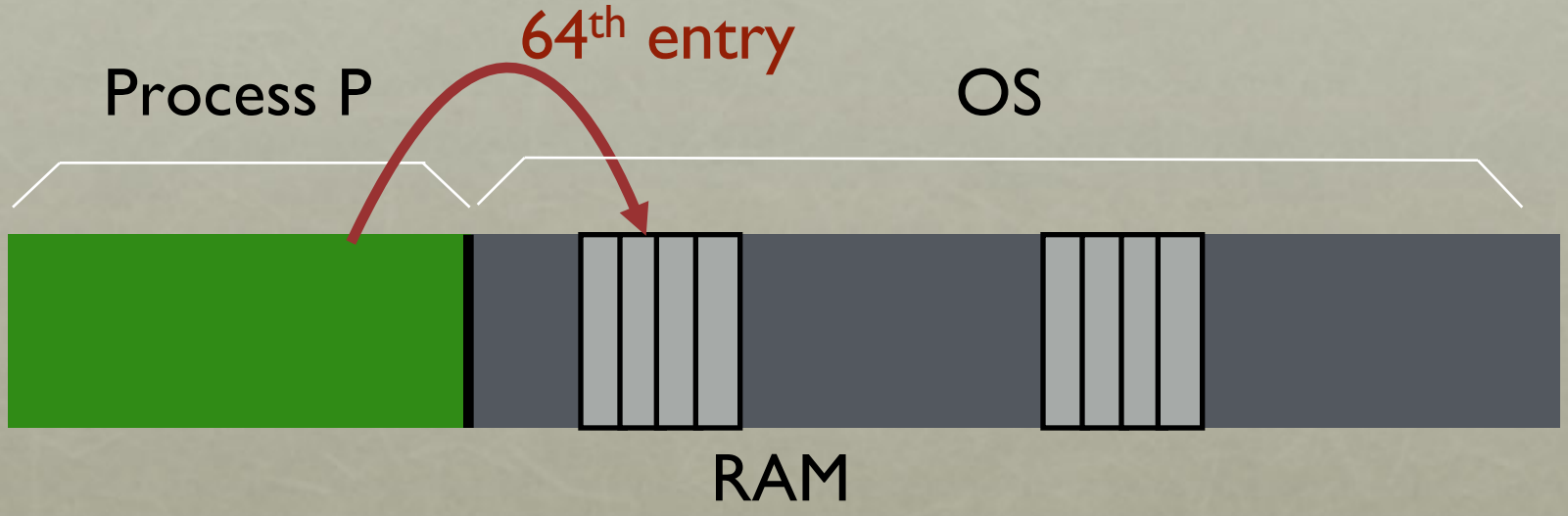
SYSTEM CALL



read():

```
movl $6, %eax;    int $64
```

SYSTEM CALL

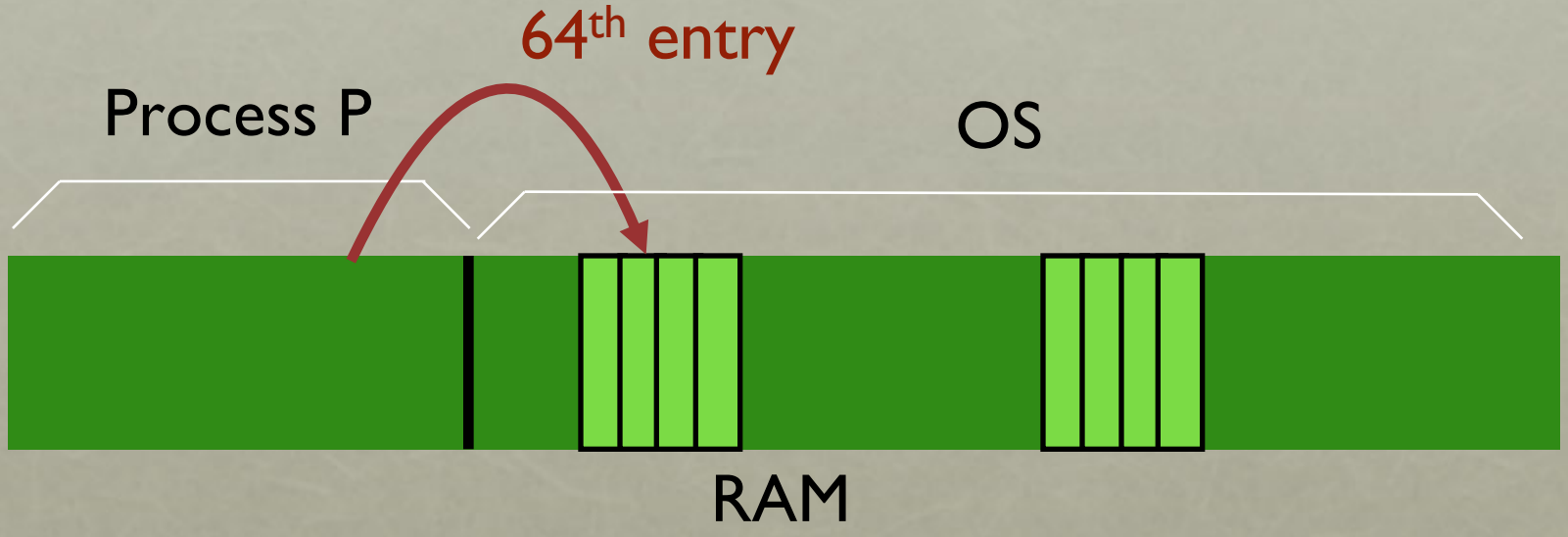



```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

SYSTEM CALL



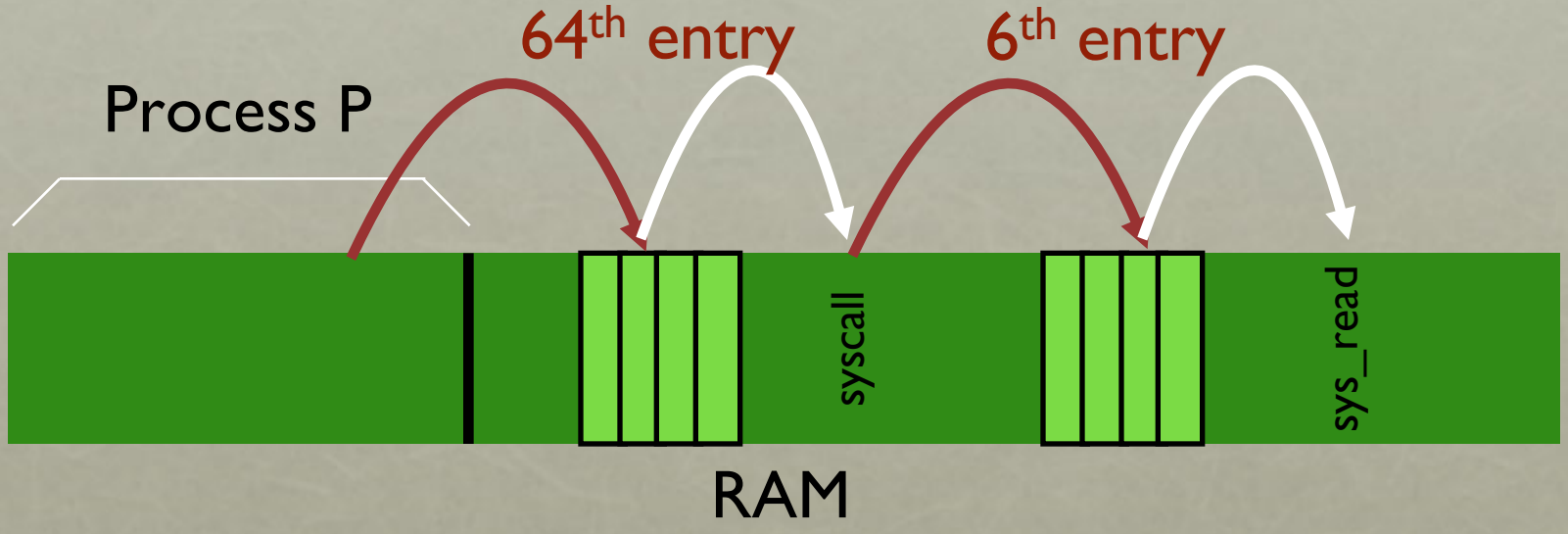

```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

Kernel mode: we can do anything!

SYSTEM CALL



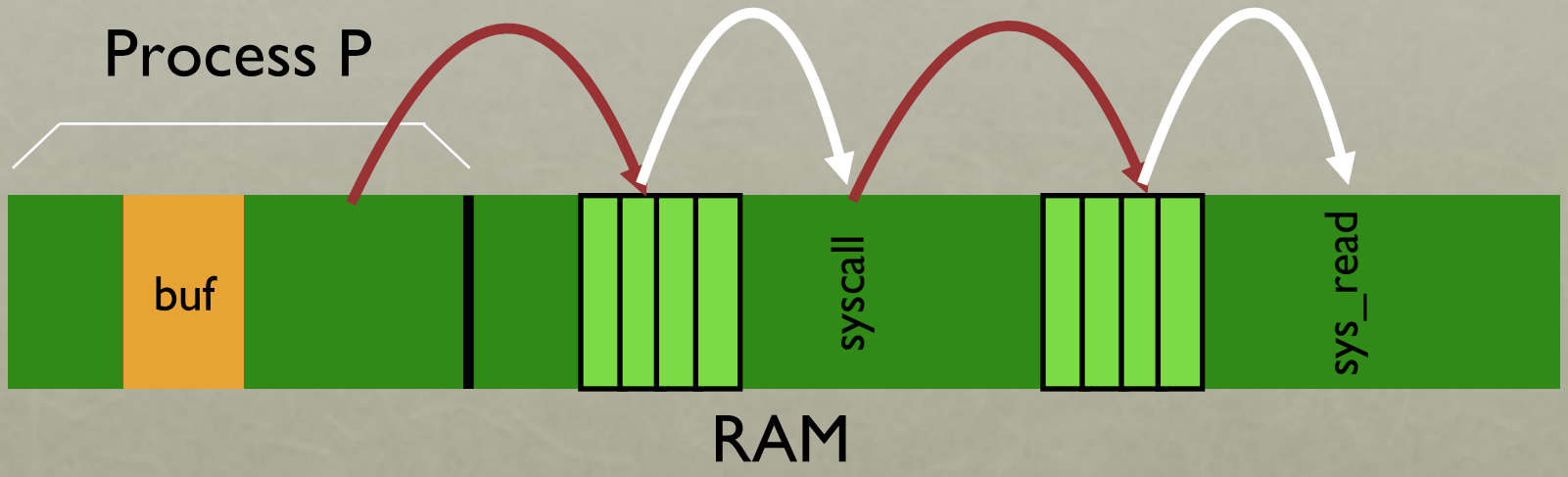

```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

Follow entries to correct system call code

SYSTEM CALL




```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

Kernel can access user memory to fill in user buffer

`return-from-trap` at end to return to Process P

HW, OS, OR USER PROCESS?

Create entry for process list	OS	HW	USER
Allocate memory for program	OS	HW	USER
Load program into memory	OS	HW	USER
Setup user stack with argv	OS	HW	USER
Fill kernel stack with reg/PC	OS	HW	USER
execute return-from-trap instruction	OS	HW	USER
restore regs from kernel stack	OS	HW	USER
switch to user mode	OS	HW	USER
set PC to main()	OS	HW	USER
Start running in main()	OS	HW	USER
Call a system call	OS	HW	USER
execute trap instruction	OS	HW	USER
save regs to kernel stack	OS	HW	USER
switch to kernel mode	OS	HW	USER
set PC to OS trap handler	OS	HW	USER
Handle trap	OS	HW	USER
Do work of syscall	OS	HW	USER
execute return-from-trap instruction	OS	HW	USER
restore regs from kernel stack	OS	HW	USER
switch to user mode	OS	HW	USER
set PC to instruction after earlier trap	OS	HW	USER
Call exit() system call	OS	HW	USER

PROCESS API: HW IN BOOK

Write a program using fork(). The child process should print "hello"; the parent process should print "goodbye". You should try to ensure that the child process always prints first; can you do this without calling wait() in the parent?

- Waitpid, sleep, other synchronization primitives such as condition variables and semaphores (next topic!)

Is it possible for child process to wait for a parent or does it always have to be the other way around?

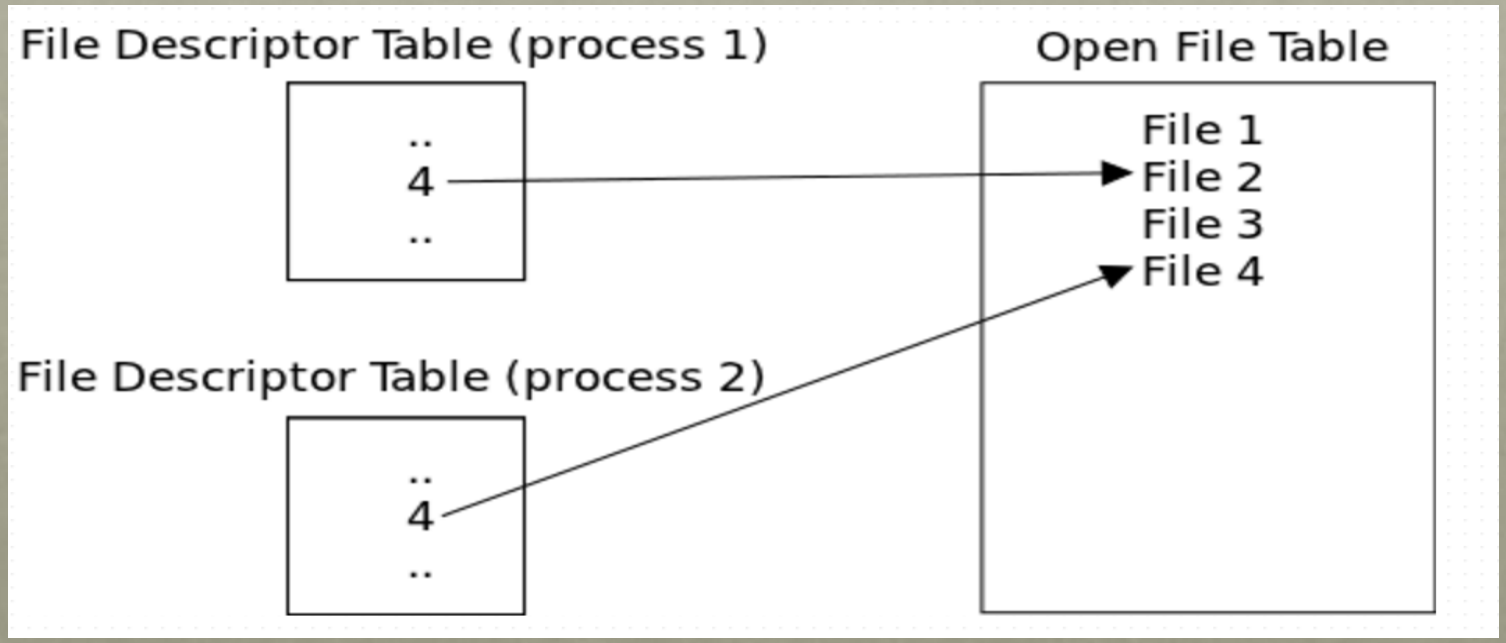
- Wait() and waitpid() apply to children processes

Typical workflow of creating a new process is to call exec in child after forking. Would there ever be a reason to create a child and call exec in the parent instead?

- No good reason I can think of

PROCESS API

If a parent and a child can access the same file descriptor, why does closing a file descriptor in a child not effect the parent process? Is it just because the file descriptor table is unique for each, but each entry references the same file?



MULTI-LEVEL FEEDBACK QUEUE (MLFQ) RULES

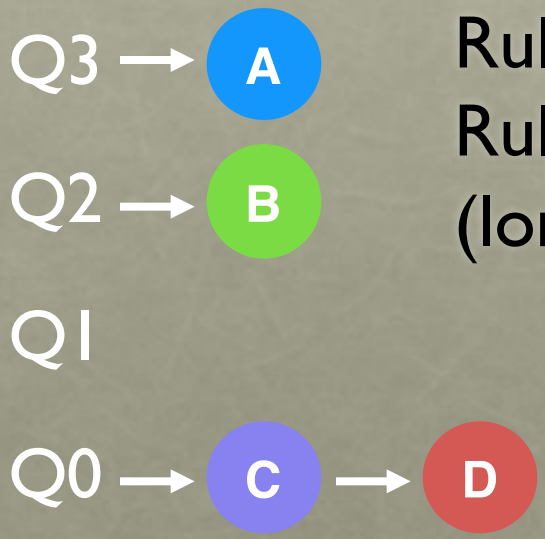
Rule 1: If priority(A) > Priority(B),
A runs

Rule 2: If priority(A) == Priority(B),
A & B run in RR

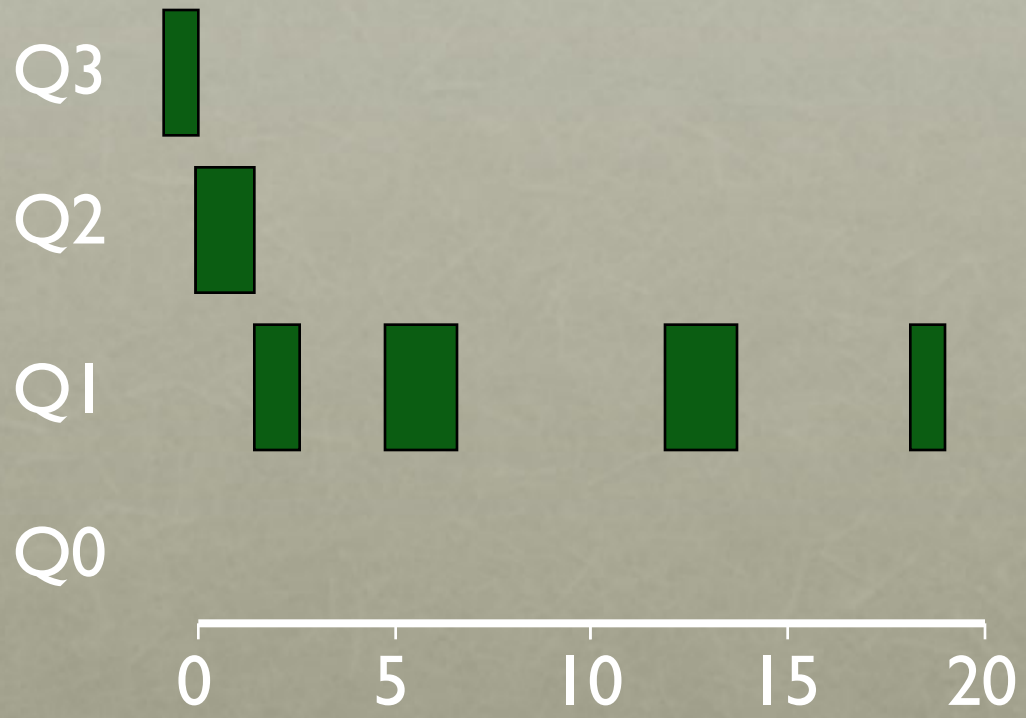
More rules:

Rule 3: Processes start at top priority

Rule 4: If job uses whole slice, demote process (longer time slices at lower priorities)



JOB THAT PERFORMS I/O PERIODICALLY



Stays in Q1 queue as long as doesn't use entire Q1 timeslice

PREVENT GAMING

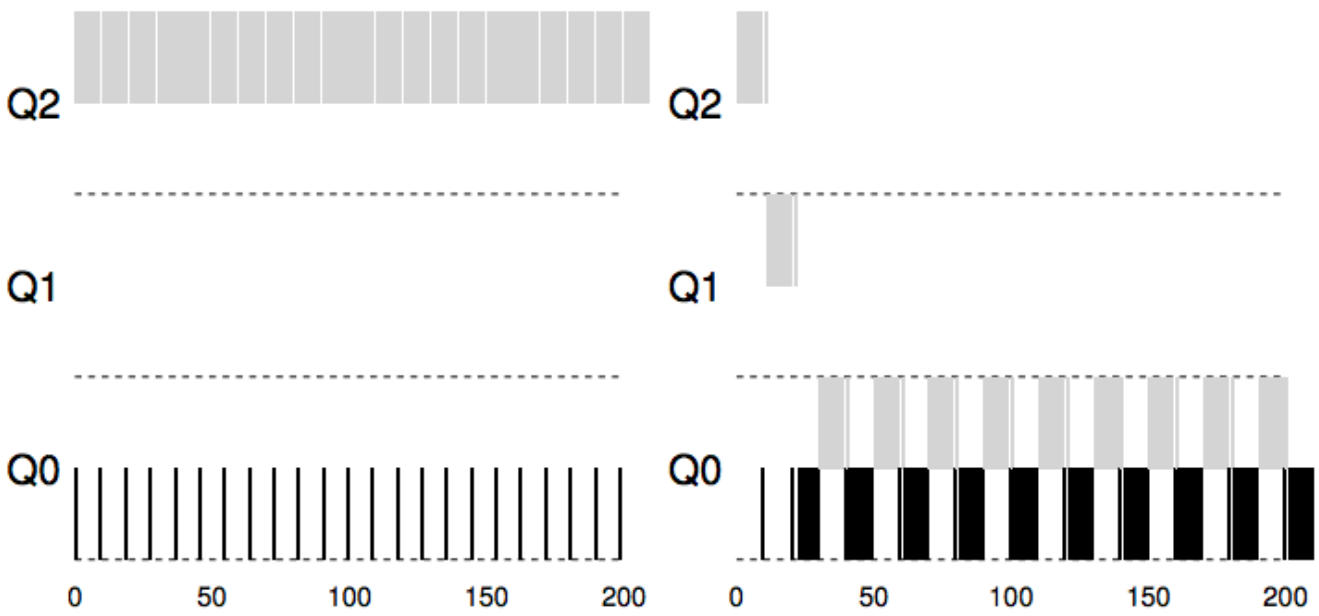


Figure 8.6: **Without (Left) and With (Right) Gaming Tolerance**

Problem: High priority job could trick scheduler and get more CPU by performing I/O right before time-slice ends

Fix: Account for process's total run time at priority level downgrade when exceed threshold

HOW ARE VIRTUAL ADDRESSES GENERATED?

- What do addresses look like from the program's perspective? (from the user process's perspective)
- Generated by compiler and contents of registers

VIRTUAL MEMORY ACCESSES?

Initial %rip = 0x10
%rbp = 0x200

```
0x10:  movl 0x8(%rbp), %edi
0x13:  addl $0x3, %edi
0x19:  movl %edi, 0x8(%rbp)
```

%**rbp** is the base pointer:
points to base of current stack frame

%rip is instruction pointer (or program counter, PC)

Memory Accesses to what **virtual addresses?**

Include instruction fetches!

1) Fetch instruction at addr 0x10
Exec:

2) load from addr 0x208

3) Fetch instruction at addr 0x13
Exec:

no memory access

4) Fetch instruction at addr 0x19
Exec:

5) store to addr 0x208

HIGH-LEVEL APPROACHES FOR VIRTUALIZING MEMORY?

Description	Name of approach
1. one process uses RAM at a time	
2. rewrite code and addresses before running	
3. add per-process starting location to virt addr to obtain phys addr	
4. dynamic approach that verifies address is in valid range	
5. several base+bound pairs per process	

Candidates: Segmentation, Static Relocation, Base, Base+Bounds, Time Sharing
Which need hardware support?

REVIEW: 4) BASE AND BOUNDS ADVANTAGES

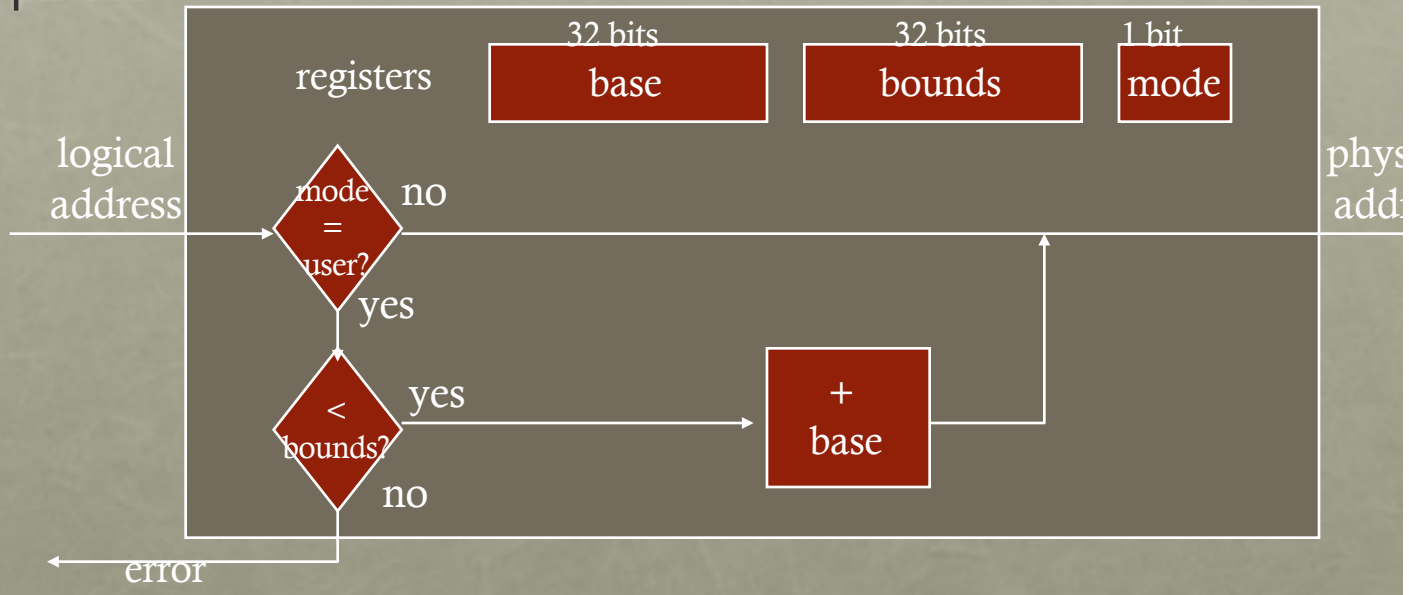
Provides protection (both read and write) across address spaces

Supports dynamic relocation

Can place process at different locations initially and also move address spaces

Simple, inexpensive implementation: Few registers, little logic in MMU

Fast: Add and compare in parallel



Subtraction for stack segment (not on exam)

PAGING AND SEGMENTATION

73) One advantage of adding segmentation to paging is that it potentially reduces the size of the page table.

True; only need page table entries for valid pages in each separate segment.

74) One advantage of adding paging to segmentation is that it reduces the amount of internal fragmentation.

False; Paging has internal fragmentation.

QUIZ: ADDRESS FORMAT

Given known page size, how many bits are needed in address to specify offset in page?

Page Size	Low Bits (offset)
16 bytes	4
1 KB	10
1 MB	20
512 bytes	9
4 KB	12

Assuming byte addressable architecture

QUIZ: ADDRESS FORMAT

Given number of bits in virtual address and bits for offset, how many bits for virtual page number?

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (vpn)
16 bytes	4	10	6
1 KB	10	20	10
1 MB	20	32	12
512 bytes	9	16	5
4 KB	12	32	20

Correct?

QUIZ: ADDRESS FORMAT

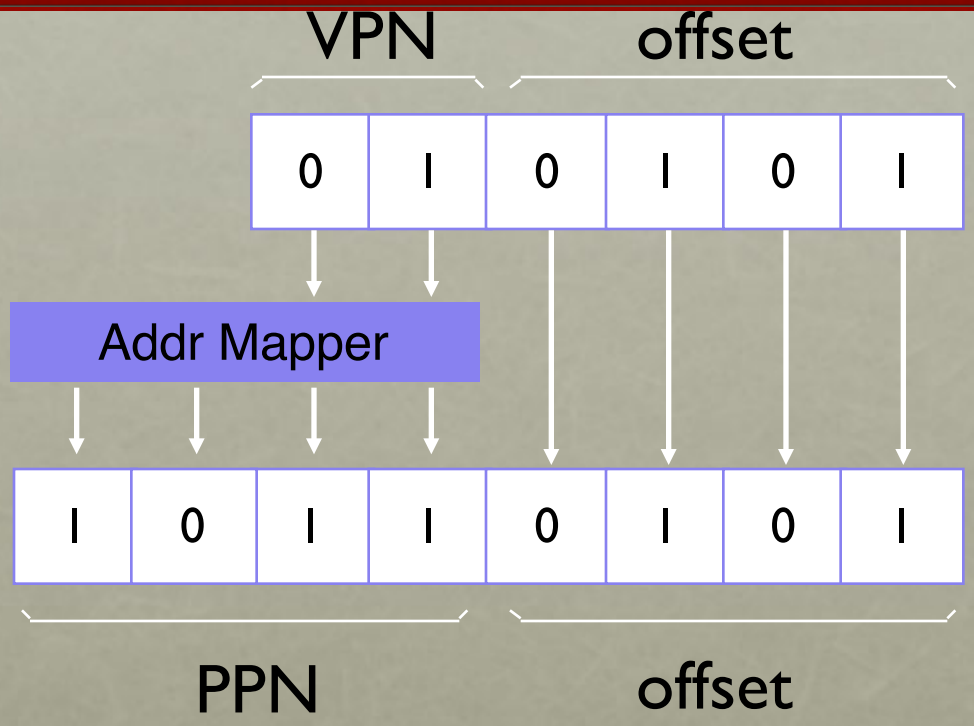
Given number of bits for vpn,
how many virtual pages can there be in an address space?

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (vpn)	Virt Pages
16 bytes	4	10	6	64
1 KB	10	20	10	1 K
1 MB	20	32	12	4 K
512 bytes	9	16	7	128
4 KB	12	32	20	1 M

Tells you how many entries are needed in page tables!

VIRTUAL => PHYSICAL PAGE MAPPING

Number of bits in virtual address format does not need to equal number of bits in physical address format



How should OS translate VPN to PPN?

For segmentation, OS used a formula (e.g., phys addr = virt_offset + base_reg)

For paging, OS needs more general mapping mechanism

What data structure is good? Big array: pagetable

WHERE ARE PAGETABLES STORED?

How big is a typical page table?

- assume **32-bit** address space
- assume 4 KB pages
- assume 4 byte page table entries (PTEs)

Final answer: 2 ^ (32 - log(4KB)) * 4 = **4 MB**

- Page table size = Num entries * size of each entry
- Num entries = num virtual pages = 2^(bits for vpn)
- Bits for vpn = 32– number of bits for page offset
  = 32 – lg(4KB) = 32 – 12 = 20
- Num entries = 2^20 = 1 MB
- Page table size = Num entries * 4 bytes = 4 MB

Implication: Store each page table in memory

- Hardware finds page table base with register (e.g., CR3 on x86)

What happens on a context-switch?

- Change contents of page table base register to newly scheduled process
- Save old page table base register in PCB of descheduled process

HOW BIG ARE PAGE TABLES?

How big is each page table?

1. PTE's are **2 bytes**, and **32** possible virtual page numbers

   32 * 2 bytes = 64 bytes

2. PTE's are **2 bytes**, virtual addrs are **24 bits**, pages are **16 bytes**

   2 bytes * 2^(24 – lg 16) = 2^21 bytes (2 MB)

3. PTE's are **4 bytes**, virtual addrs are **32 bits**, and pages are **4 KB**

   4 bytes * 2^(32 – lg 4K) = 2^22 bytes (4 MB)

4. PTE's are **4 bytes**, virtual addrs are **64 bits**, and pages are **4 KB**

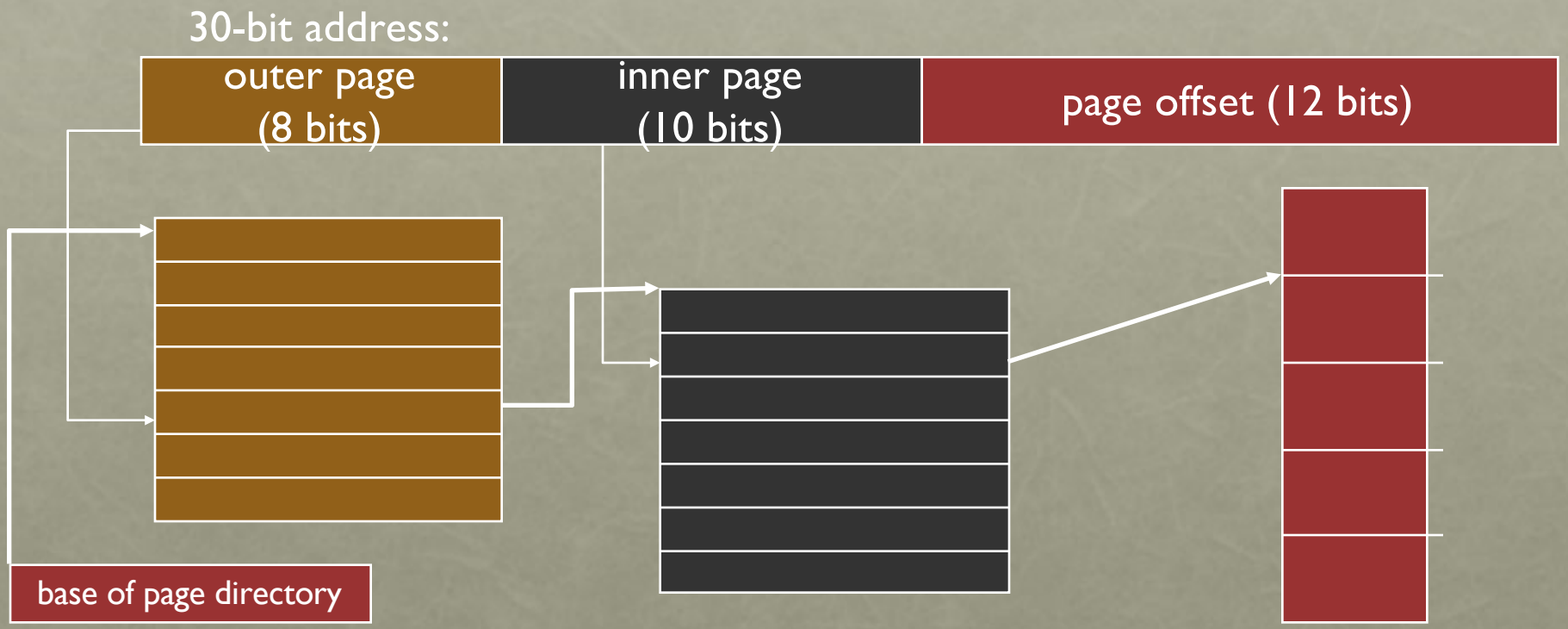
   4 bytes * 2^(64 – lg 4K) = 2^54 bytes

3) MULTILEVEL PAGE TABLES

Goal: Allow each 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)



SIMULATIONS: MULTI-LEVEL PAGETABLES

Each page is 32 bytes

The virtual address space for process in question (assume only one) is 1024 pages
Physical memory consists of 128 pages

Thus, a virtual address needs 15 bits (5 for the offset, 10 for the VPN).

A physical address requires 12 bits (5 offset, 7 for the PFN).

1 byte PTE

The system assumes a multi-level page table. Thus, the upper five bits of a virtual address are used to index into a page directory; the page directory entry (PDE), if valid, points to a page of the page table. Each page table page holds 32 page-table entries (PTEs). Each PTE, if valid, holds the desired translation (physical frame number, or PFN) of the virtual page in question.

The format of a PTE is thus:

VALID | PFN6 ... PFN0

and is thus 8 bits or 1 byte.

The format of a PDE is essentially identical:

VALID | PT6 ... PT0

SIMULATIONS: MULTI-LEVEL PAGETABLES

Virtual address 0x611c contents →

PDBR: 108 (decimal) [page directory is held in this page]

```
page 108: 83 fe e0 da 7f d4 7f eb be 9e d5 ad e4 ac 90 d6 92 d8 c1 f8 9f e1 ed e9 a1 e8
c7 c2 a9 d1 db ff
```

Which entry of PageDir?

PDE: 16+8=24

→ 0xa1

→ 1010 0001

→ Valid and 33

0110 0001 0001 1100

```
page  33: 7f 7f 7f 7f 7f 7f 7f 7f b5 7f 9d 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f
f6 b1 7f 7f 7f 7f
```

Which entry of Page Table?

PTE: 8

→ b5

→ 1011 0101

→ valid and 32+16+4+1 = Page 53

```
page  53: 0f 0c 18 09 0e 12 1c 0f 08 17 13 07 1c 1e 19 1b 09 16 1b 15 0e 03 0d 12 1c 1d
0e 1a 08 18 11 00
```

Which offset on page?

→ offset 16+8+4 = 28

Final data value: 08!

TLBS: HW AND OS ROLES

Who Handles TLB Hit?

Who Handles TLB Miss? HW or OS

H/W

H/W must know where pagetables are stored in memory

- CR3 register on x86
- Pagetable structure fixed and agreed upon between HW and OS
- HW "walks" known pagetable structure and fills TLB

HW AND OS ROLES

Who Handles TLB MISS? **H/W** or **OS**?

OS:

CPU traps into OS upon TLB miss
"Software-managed TLB"

OS interprets pagetables as it chooses; any data structure possible
Modifying TLB entries is privileged instruction

PRESENT VS VALID BIT

- Virtual memory when page is not allocated in physical memory (RAM); instead on disk
- Why is a present bit needed? Why not just use valid bit?

VIRTUAL ADDRESS SPACE MECHANISMS

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

- Nothing (error): Free
- Physical main memory: Small, fast, expensive
- Disk (persistent storage): Large, slow, cheap

Extend page tables with an extra bit: `present`

- `permissions (r/w), valid, present`
- Page is not allocated or mapped (not `valid`)
  - Segmentation fault
- Page in memory: `present` bit set in PTE, hold PPN
- Page on disk: `present` bit cleared
  - PTE points to **block address on disk**
  - Causes trap into OS when page is referenced
  - **Trap: page fault**

BELADY'S ANOMALY FOR FIFO

Page reference string: ABCDABEABCDE

Three or Four pages of physical memory

Metric: Miss count

GOOD LUCK!

Remember ID and pencils

Two hours – 7:30 – 9:30 pm, Thursday 10/10

- Room Ingraham 19:
  If you are enrolled in Discussion Section 301 (Wed 11-11:50am)
- Room Ingraham B10:
  All other discussion sections