CS240c – Advanced OS Implementation

Instructor: David Mazières

CAs: Nickolai Zeldovich, Jinyuan Li, Antonio Nicolosi

Stanford University

Administrivia

- **Class web page:** `http://cs240c.scs.stanford.edu/`
  - All assignments, handouts, lecture notes
- **Part of each class will be spent discussing papers**
  - Read the papers before class
- **Grading based on four factors**
  - Participation in discussion (so read the papers before class!)
  - Two open-book quizes (bring all papers)
  - Lab assignments
  - Final project report & presentation due during exam period
- **Grade = max (project, all four factors)**

More administrivia

- **Class email list:** `cs240c@scs.stanford.edu`
  - Should reach all students
  - Let us know if you don't get mail test mail tonight (or if you want to subscribe under a different address)
  - Feel free to discuss labs, projects, papers, etc. on the list (but please don't post code or outright divulge solutions)
- **Staff email list:** `cs240c-staff@scs.stanford.edu`
  - Please mail staff list rather than individual staff members
  - Note: doesn't follow usual list naming conventions (I won't get mail to `cs240c-aut0506-staff@stanford.edu`)
- **Other than those two addresses, you *must* explicitly acknowledge all sources of help for the assignments.**

Prerequisites & Goal

- **We assume Programming experience in C**
- **Some familiarity with Unix and system calls**
- **An undergraduate "textbook" OS class**
  - Familiar with concepts like Virtual Memory, processes, etc.
  - But maybe never seen a real implementation, or written code to do things like manipulate page tables

Goal of class:

Deeper understanding of OS implementation issues, and introduction to OS research.

Course topics

- **User/kernel APIs**
- **Virtual memory**
- **Scheduling**
- **Threads**
- **I/O implementation**
- **Kernel architectures**
- **IPC & Synchronization**
- **OS security**

Programming assignments

- **Build minimal OS for PC hardware**
  - Bootstrap & Virtual Memory
  - User code & scheduler
  - System calls & copy-on-write fork
  - File system and/or shell
- **Final project of your choice, built on labs**
  - Look for ideas when reading research papers
  - Work in teams of ~ 2 people
  - Demo your OS & turn in paper during exam period

OS Platform

- **Your OS will run on a standard PC**
  - x86 architecture (Pentium, Athlon, etc.)
  - IDE disk, standard console, etc.
- **Developed mostly in C, some assembly language**
  - Use GCC `asm` extension for inline assembly
- **Class web page contains many references for PC hardware**
- **Will test and run code using bochs**
  - Faithful PC hardware simulator
  - Much easier to debug on than real hardware
  - But what runs on Bochs will run on real hardware

First lab

- **Booting and virtual memory**
  - Out by tomorrow (we will send email)
- **Due before class next Thursday**
  - But free extension to midnight if you come to class (so don't skip class to finish the assignment!)
- **Part A easy, but do it soon**
  - Read "class machines" link to set up your environment
  - Make sure you can run bochs, get software, etc.

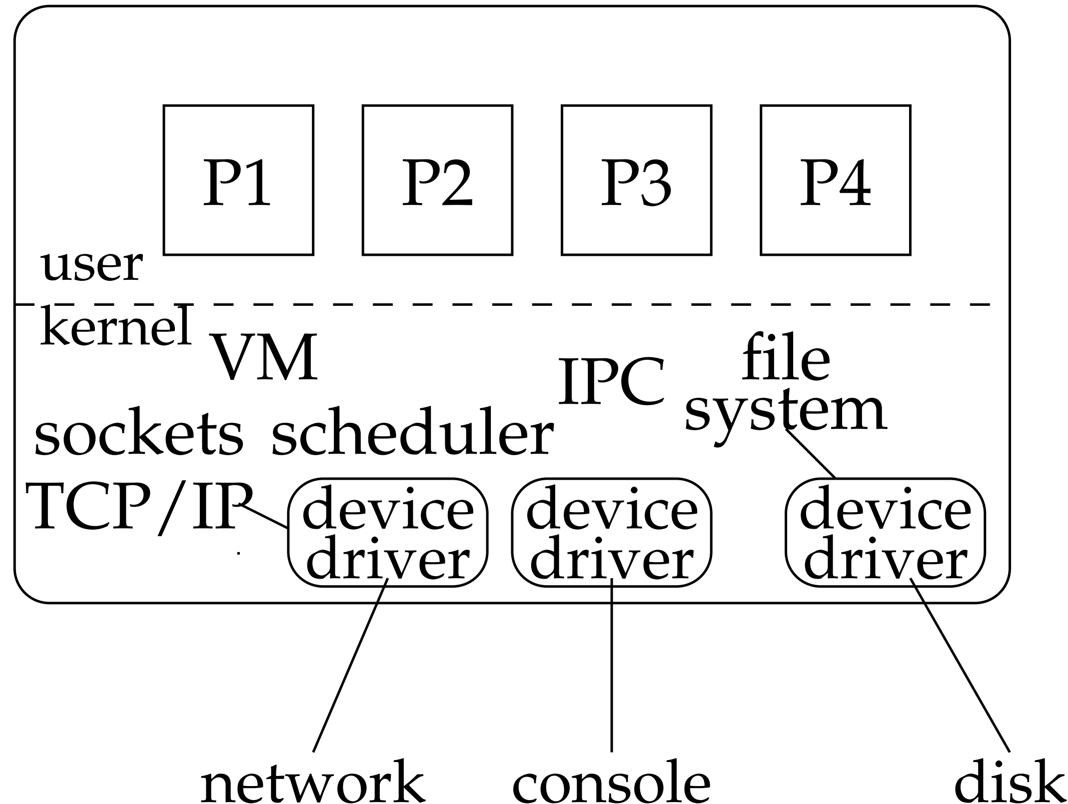
What is an operating system?

- **Makes hardware useful to the programmer**
- **Provides abstractions for applications**
  - Manages and hides details of hardware
  - Accesses hardware through low/level interfaces unavailable to applications
- **Provides protection**
  - Prevents one process/user from clobbering another

Why study operating systems?

- **Operating systems are a maturing field**
  - Most people use a handful of mature OSes
  - Hard to get people to switch operating systems
  - Hard to have impact with a new OS
- **High-performance servers are an OS issue**
  - Face many of the same issues as OSes
- **Resource consumption is an OS issue**
  - Battery life, radio spectrum, etc.
- **Security is an OS issue**
  - Hard to achieve security without a solid foundation
- **New "smart" devices need new OSes**

Typical OS structure



- **Most software runs as user-level processes**
- **OS kernel handles "privileged" operations**
  - Creating/deleting processes
  - Access to hardware

The different Unix contexts

- **User-level**
- **Kernel “top half”**
  - System call, page fault handler, kernel-only process, etc.
- **Software interrupt**
- **Device interrupt**
- **Timer interrupt (hardclock)**
- **Context switch code**

Transitions between contexts

- **User $\rightarrow$ top half:**
- **User/top half $\rightarrow$ device/timer interrupt:**
- **Top half $\rightarrow$ user/context switch:**
- **Top half $\rightarrow$ context switch:**
- **Context switch $\rightarrow$ user/top half**

Transitions between contexts

- **User $\rightarrow$ top half: syscall, page fault**
- **User/top half $\rightarrow$ device/timer interrupt: hardware**
- **Top half $\rightarrow$ user/context switch: return**
- **Top half $\rightarrow$ context switch: sleep**
- **Context switch $\rightarrow$ user/top half**

Top/bottom half synchronization

- **Top half kernel procedures can mask interrupts**

```
int x = splhigh ();
/* ... */
splx (x);
```

- **splhigh disables all interrupts, but also splnet, splbio, splsoftnet, . . .**
- **Masking interrupts in hardware can be expensive**
  - Optimistic implementation – set mask flag on splhigh, check interrupted flag on splx
- **Interface designed before multiprocessors common**

Kernel Synchronization

- **Need to relinquish CPU when waiting for events**
  - Disk read, network packet arrival, pipe write, signal, etc.
- `int tsleep(void *ident, int priority, ...);`
  - Switches to another process
  - `ident` is arbitrary pointer—e.g., buffer address
  - `priority` is priority at which to run when woken up
  - PCATCH, if ORed into `priority`, means wake up on signal
  - Returns 0 if awakened, or ERESTART/EINTR on signal
- `int wakeup(void *ident);`
  - Awakens all processes sleeping on `ident`
  - Restores SPL to value when they went to sleep (so fine to sleep at splhigh)

CS240c Kernel

- **Asynchronous interface, not like UNIX**
  - Only one kernel stack
  - Interrupts always disabled in kernel (except in idle loop)
  - Kernel never sleeps (except in idle loop)
- **Why do away with threads in kernel?**
  - Vastly complicates programming (more error-prone)
  - Ill-suited to certain user-level applications
  - Conversely, can simulate traditional synchronous kernel interface at user-level in terms of asynchronous interface

System calls

- **Goal: invoke kernel from user-level code**
    - Like a library call, but into more privileged OS code
- **Applications request operations from kernel**
- **Kernel supplies well-defined *system call* interface**
    - Applications set up syscall arguments and *trap* to kernel
    - Kernel performs operation and returns result
- **Higher-level functions built on syscall interface**
    - `printf`, `scanf`, `gets`, etc. all user-level code
- **Example: POSIX/UNIX interface**
    - Your kernel system call interface will be lower-level
    - But can build POSIX-like functions in libraries

Assembly language

- **Program "text" contains binary instructions**
  - CPU executes one instruction at a time
  - Usually executes next sequential instruction in memory
  - Branch/jump/call inst. specifies different next instruction
- **Instructions typically manipulate**
  - Registers – small number of values kept by processor
  - Memory
  - "Special" registers whose bits have particular significance
  - The instruction pointer (IP) – which inst. to execute next
  - I/O devices

x86 assembly language crash course

- **Mostly two operand instructions**
- **Unfortunately *two* prevalent syntaxes**
  - "Intel syntax": `op dst, src`
  - "AT&T (gcc/gas) syntax": `op src, dst`
  - We will always use AT&T syntax
  - But a lot of documentation uses Intel syntax
- **Examples:**

Assembly	**C pseudo-code**
`movl %eax,%edx`	`edx = eax;`
`movl $0x123, %edx`	`edx = 0x123;`
`movl 0x124, %edx`	`edx = *((int32_t*) 0x124);`
`movl (%ebx), %edx`	`edx = *((int32_t*) ebx);`
`movl 4(%ebx), %edx`	`edx = *((int32_t*) (ebx+4));`

Real vs. protected mode

- **Real mode – 16-bit registers, 1 MB virtual mem**
  - Segment registers provide top 4 bits of physical address: `movw (%ax),%dx` means `dx` = `*(int_32_t*)`$(16 \times$ `ds` $+$ `ax`$)$
- **Protected mode – segment registers virtualized**
  - Load segment registers from table of *segment descriptors*
  - Depending on `%cs` descriptor, default ops can be 32 bits
  - 32-bit virtual address space, can optionally be paged
  - 32- or 36-bit physical address space, depending on mode
- **We will mostly use 32-bit protected mode**
  - All remaining examples will be 32-bit code
  - 32-bit AT&T Instructions have **l** suffix, for long

More 32-bit instructions

- **ALU ops: addl, subl, andl, orl, xorl, notl, ...**
  - incl, decl – add or subtract 1
  - cmpl – like subl, but discards subtraction result
- **Stack instructions:**

Stack op	**equivalent**
`pushl %eax`	`subl $4,%esp` `movl %eax,(%esp)`
`popl %eax`	`movl (%esp),%eax` `addl $4,%esp`

- **Other stack instructions: pushfl, pushal**
  - `leave` means: `movl %ebp,%esp; popl %ebp`

Conditional branches

- **Conditional branches based EFLAGS reg. bits**
  - CF (carry flag) set if op carried/borrowed → `jc, jnc`
  - ZF (zero flag) set if result zero → `jz/je, jnz/jne`
  - SF (sign flag) set to high bit of result → `jn, jp`
  - OF (overflow flag) set if result too large → `jo, jno`
  - `jge` → "Jump if greater or equal", i.e., SF=OF
  - `jg` → "Jump if greater", i.e., SF=OF and ZF=0
- `jmp` **unconditional jump, call/ret uses stack:**

Stack op	**psedo-asm equiv**
`call $0x12345`	`pushl %eip` `movl $0x12345,%eip`
`ret`	`popl %eip`

Example

```
        for (i = 0; i < a; i++)    /* i = %edx, a = %ecx */
          sum += i;
------------------------------------------------------------
        xorl %edx,%edx     # i = 0 (more compact than movl)
        cmpl %ecx,%edx     # test (i - a)
        jge .L4            # >= 0 ? jump to end
        movl sum,%eax      # cache value of sum in register
.L6:
        addl %edx,%eax     # sum += i
        incl %edx          # i++
        cmpl %ecx,%edx     # test (i - a)
        jl .L6             # < 0 ? go to top of loop
        movl %eax,sum      # store value of sum back in memory
.L4:
```

Assembler local labels

- **Often want to define macros in assembly language**
  - Typically .S files are C-preprocessor source
- **Problem: how to choose unique labels**
  - If there's a loop in macro, and used multiple times
  - You would have a duplicate label
- **Solution: Numeric labels are local**
  - f suffix means forwards
  - b suffix means backwards

Example w. local labels

```
        for (i = 0; i < a; i++)
          sum += i;
------------------------------------------------------------
        xorl %edx,%edx      # i = 0 (more compact than movl)
        cmpl %ecx,%edx      # test (i - a)
        jge 2f              # >= 0 ? jump to end
        movl sum,%eax       # cache value of sum in register
1:
        addl %edx,%eax      # sum += i
        incl %edx           # i++
        cmpl %ecx,%edx      # test (i - a)
        jl 1b               # < 0 ? go to top of loop
        movl %eax,sum       # store value of sum back in memory
2:
```

32-bit protected-mode registers

Caller-saved:

`%eax`

`%edx`

`%ecx`

Callee-saved:

`%ebx` `%ebp` ⟵ frame pointer

`%esi` `%esp` ⟵ stack pointer

`%edi`

Special-purpose: eflags, `%cr3`, GDTR, IDTR, LDTR, TSS

Segment Registers: `%cs %ss %ds %es` [`%fs %gs`]

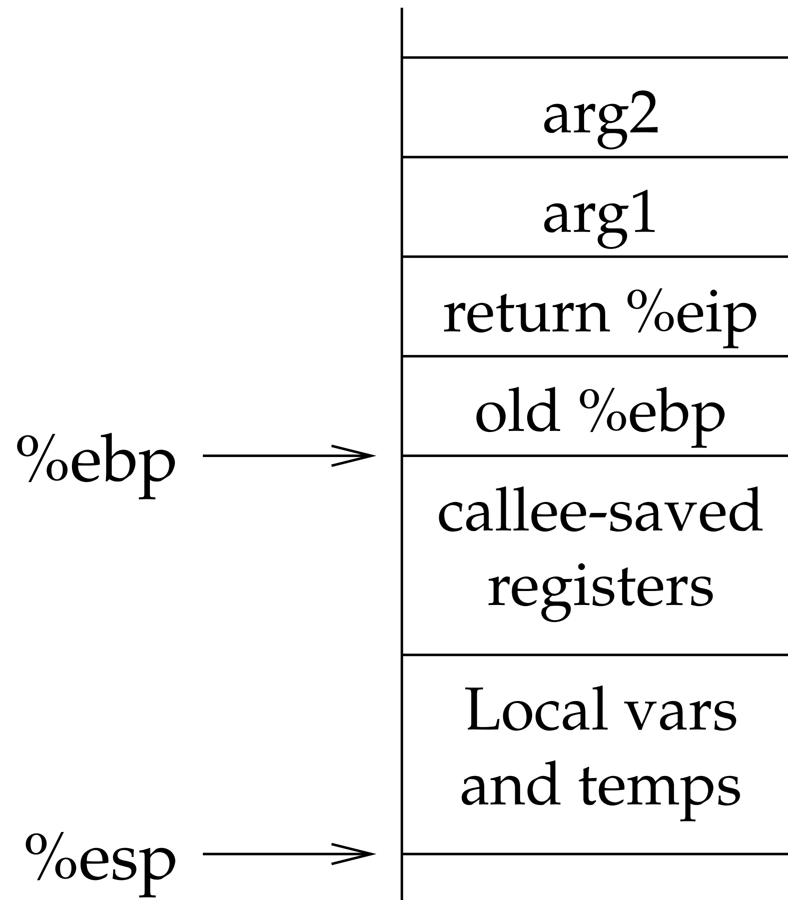
TI : 0 = global/1 = local table

RPL : Requestor privilege level (0–3)

Calling conventions

- **GCC dictates how stack is used**
- **After call instruction:**
  - %esp points at return address
  - %esp+4 points at first argument
- **After ret:**
  - %esp points at arguments pushed by caller
  - called function may have trashed arguments
  - %eax contains return value (or trash if function is void)
  - %ecx, %edx may be trashed
  - %ebp, %ebx, %esi, %edi must have previous contents

Picture of stack



- **Code may push temp vars on stack at any time**
  - So refer to args and locals using %ebp

Typical function code

```
  int main(void) { return f(8)+1; }
  int f(int x) { return g(x); }
  int g(int x) { return x+3; }
------------------------------------------------------------
main:
        pushl %ebp
        movl %esp,%ebp
...
        pushl $8
        call f
        incl %eax
        leave
        ret
```

code for f

```
  int f(int x) { return g(x); }
------------------------------------------------------------
f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $20, %esp
        pushl   8(%ebp)
        call    g
        leave
        ret
```

code for g

```
  int g(int x) { return x+3; }
------------------------------------------------------------
g:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        addl    $3, %eax
        leave
        ret
```

Inline assembly language

- **Large assembly language files are a pain**
  - Often want to write C, but need a particular asm instruction
  - Thus, gcc provides asm extension
- **Straw man, just inject assembly language:**
  - E.g., `asm ("movl %esp,%eax");`
  - But what if compiler needed value in %eax?
  - And what if you need some value the compiler has? (remember how gcc cached value of sum in %eax)

GCC inline assembly language

- **Specify values needed, output, and clobbered**

```
asm ("statements" : output_values
     : input_values : clobbered);
```

- **Example:**

```
u_int32_t stkp;
asm ("movl %esp,%0" : "=r" (stkp) ::);
printf ("The stack pointer is 0x%x\n", stkp);
```

- **Notes:**
  - "r" means any register, or can specify w. a/b/c/d/S/D
  - "m" means memory, "g" general, "I" small constant
  - If in/out value same, specify, e.g., "0" for in value
  - clobbered may need "memory" and/or "flags"

I/O instructions

- **How to interact with devices?**
- **PC design – use special I/O space**
  - special instructions inb/inw, outb/outw (for 8/16 bits)
  - Load and store bytes & words, like normal memory
  - But special processor I/O pin says "this is for I/O space"
- **To access from C code:**

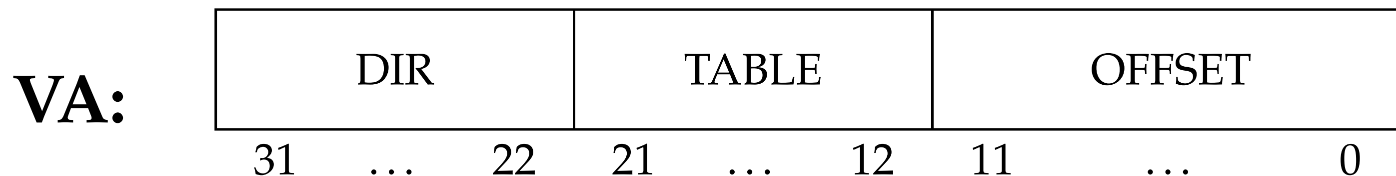
```
static inline u_char inb (int port) {
  u_int8_t data;
  asm volatile("inb %w1,%0" : "=a" (data) : "d" (port));
  return data;
}
static inline void outb(int port, u_int8_t data) {
  asm volatile("outb %0,%w1" :: "a" (data), "d" (port));
}
```

x86 hardware tables

LDT/GDT. Descriptor tables, indexed by segment registers.
IDT. Vectors for 256 exceptions, interrupts, and user traps.
TSS. Task state segment.

- Stack pointers for privilege increases.
- I/O-space permissions with byte granularity (allows `cli`).

Page Directory/Tables. Two-level page tables in hardware.



Special register `%cr3` points to page directory.

x86 segments

32 types of segments: 16/32-bit, expand-up/down, read/write, code/data, conforming/non-conforming, call/trap/interrupt/task gate, available/busy TSS, LDT.

- **user segments.** 32-bit base, 16-bit limit (granularity byte/4K). RPL bits of `%cs` and `%ss` determine current privilege level.
- **trap gates.** 16-bit segment selector, 32-bit offset.
- **interrupt gates.** Trap gates that disable interrupts.

Loading segments:

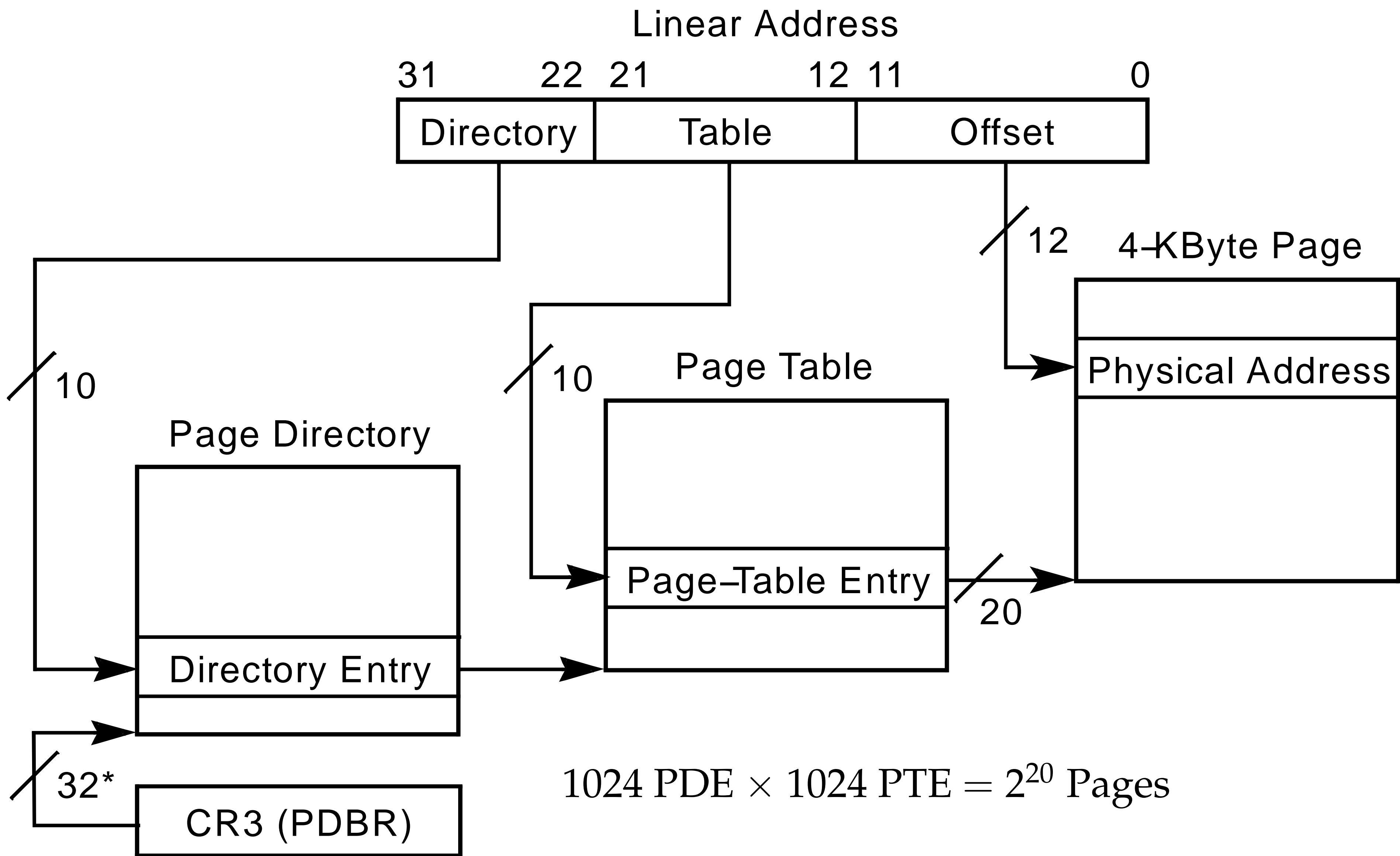
- direct load, far jump, `int`: $MIN(CPL, RPL) \leq DPL$
- exception, interrupt: DPL not checked
- all gates: adjust CPL to DPL of designated segment.

Segments are mostly a pain

- **Segment base + offset known as *linear address***
- **Usually don't want to worry about segments**
  - But can't disable segmentation hardware
- **Solution: Flat model – offset=linear address**
  - Give all segments a base address of 0
  - Now mostly don't have to worry about segments
- **However, still need segments for interrupts/traps**

x86 paging

- **Translation occurs on linear address output of segmentation.**
- **4K pages.**
- **PTEs have the following options:**
  - **writeable.** Disables user and kernel (486+) mode writes.
  - **user.** Access with $CPL = 3$ when set, otherwise just 0–2.
  - **c**ache disable bit, cache write-through bit
  - **d**irty bit, accessed bit, present bit.
- `%cr3` **designates address space by selecting page directory. Loading** `%cr3` **flushes the TLB.**



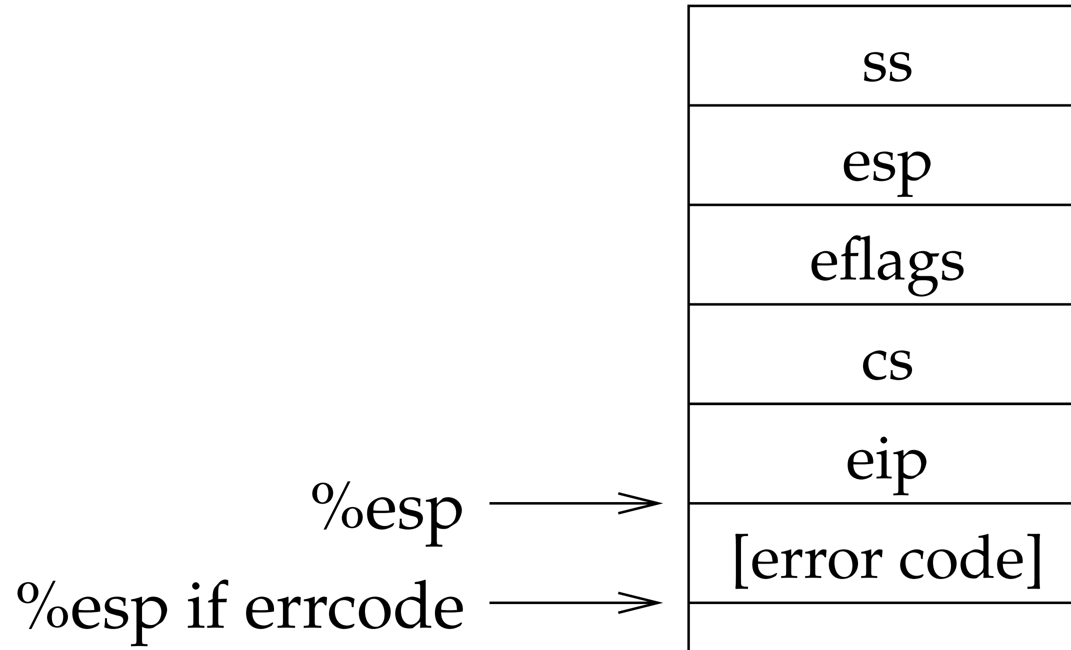
*32 bits aligned onto a 4–KByte boundary

Page–Table Entry (4–KByte Page)

Interrupts and traps

- **CPU supports 256 interrupts**
  - IDT contains segment descriptors for each int
  - Trap gate says what code segment / offset to use
  - Interrupt gate like trap gate, but disables interrupts
- **How does CPU vector to IDT entry?**
  - `int`, `int3`, `into` instructions
  - Built-in trap (e.g., page fault, trap numbers hard-coded 0–19)
  - Interrupt from external device (8-bit interrupt number supplied on CPU pins)

Trap frame



- **Only some traps have error codes**
- **Interrupts do not cause error code to be pushed**

Example: page fault – 14

- **Has error code, bits mean:**
  - bit 0 – 0=page not present, 1=protection violation
  - bit 1 – 0=access was read, 1=access was write
  - bit 2 – 0=fault in user mode, 1=supervisor mode
- **In addition, special register `%cr2` holds faulting virtual address**

Discussion

- **Why might page fault occur in supervisor mode?**
- **Where does stack pointer come from after trap?**
  - Why is this important?
- **What happens if user code calls `int 14`?**
- **W$\wedge$X**