What You Need to Know for Project Three

Dave Eckhardt
Babu Pillai

Steve Muckle

Synchronization

Project 3 Checkpoint 1 demo

- Friday, October 10th
  - Meet in Wean 5207 cluster during class time
- Attendance is *mandatory* (*nobody* has a conflict!)
  - We expect *you* even if your code isn't quite done
  - Regardless of the reason

Synchronization

Near-term items

- Please respond ***quickly*** to any mail from me
- Please begin reading the P3 handout ***today***
- Thanks!

Synchronization

- Reminder: Book report
  - If end-of-semester won't be the best deadline for you, it's ok for you to submit it early!
- Thinking about the future
  - Spring/Fall: 15-418/618, 15-445/645
  - Spring: 15-411/611; 15-721: Database Systems (by audition);
  - Google “Summer of Code”?
  - Fall: 15-412/612(?); 15/18-746
  - Fall: 15-712 (by audition)

Overview

- Introduction to the Kernel Project
- Mundane Details in x86
  - registers, paging, the life of a memory access, context switching, system calls, kernel stacks
- Loading Executables
- Style Recommendations (or pleas)
- Attack Strategy
- A Quick Debug Story

Introduction to the Kernel Project

P3:P2 :: P2:P1!

P2

- Stack, registers, stack, race conditions, stack

P3

- Stack, registers, page tables, scheduling, races...

You will “become one with” program execution

P1: living without common assumptions

P3: providing those assumptions to users

The P3 Experience

- Goals/challenges
  - More understanding
    - Of OS
    - Practice with synthesizing design requirements
  - More code
    - More planning
    - More organization
  - More quality!
    - Robust
  - ***More debugging!***

Introduction to the Kernel Project: Kernel Features

Your kernels will feature:
- preemptive multitasking
- multiple virtual address spaces
- a “small” selection of useful system calls
- robustness (hopefully)

Introduction to the Kernel Project: Preemptive Multitasking

- "Preemptive multitasking" is forcing multiple threads to share the CPU
- You will use the timer interrupt to do this
- Reuse your timer code from P1 if possible

Introduction to the Kernel Project: Preemptive Multitasking

- Simple round-robin scheduling will suffice
  - Some system calls will modify the sequence
  - Think about them <u>before</u> committing to a design
- Context switching is tricky but cool
- As in P2, creating a new task/thread is hard
  - Especially given memory sharing
- As in P2, exiting is tricky too
  - At least one “How can I do <u>that</u>???” question

Introduction to the Kernel Project: Multiple Virtual Address Spaces

- The x86 architecture supports paging
- You will use this to provide a virtual address space for each user task
- Each user task will be isolated from others
- Paging will also protect the kernel from users
- Segmentation will not be used for protection

Introduction to the Kernel Project: System Calls

You used them in P2

Now you get to implement them

Examples include fork(), exec(), thread_fork

There are easier ones like gettid()

- The core cluster – <u>must</u> work solidly
  - fork(), exec()
  - vanish(), wait()

Mundane Details in x86

We looked at some of these for P1

Now it is time to get the rest of the story

How do we control processor features?

What does an x86 page table look like?

What route does a memory access take?

How do you switch from one process to another?

Mundane Details in x86: Registers

General purpose regs (not interesting)

Segment registers (somewhat interesting)

- %cs, %ss, %ds, %es, %fs, %gs

%eip (a little interesting)

EFLAGS (interesting)

Control Registers (very interesting)

- %cr0, %cr1, %cr2, %cr3, %cr4
- esp0 field in the hardware “task segment”

Mundane Details in x86: General Purpose Registers

The most boring kind of register

%eax, %ebx, %ecx, %edx, %edi, %esi, %ebp, %esp

%eax, %ebp, and %esp are exceptions, they are slightly interesting

- %eax is used for return values
- %esp is the stack pointer
- %ebp is the base pointer

Mundane Details in x86: Segment Selector Registers

Slightly more interesting

%cs specifies the segment used to access code (<u>also specifies privilege level</u>)

%ss specifies the segment used for stack related operations (pushl, popl, etc)

%ds, %es, %fs, %gs specify segments used to access regular data

Mind these during context switches!!!

If something <u>specific</u> breaks, check these

Mundane Details in x86: The Instruction Pointer (%eip)

It's interesting

Cannot be read from or written to directly

- (branch, call, return)

Controls which instructions get executed

'nuff said.

Mundane Details in x86: The EFLAGS Register

It’s interesting

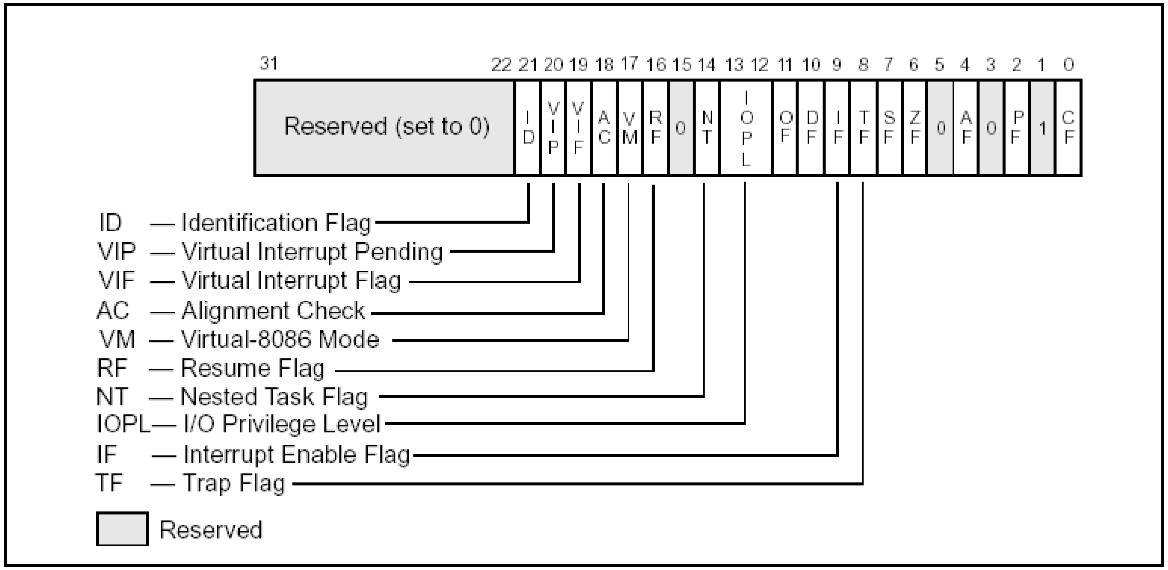


Figure 2-3. System Flags in the EFLAGS Register

Flag city, including interrupt-enable, arithmetic flags

- You want “alignment check” off

Mundane Details in x86: Control Registers

Very interesting!

An assortment of important flags and values

%cr0 contains powerful system flags that control things like paging, protected mode

%cr1 is reserved (now that’s really interesting)

%cr2 contains the address that caused the last page fault

Mundane Details in x86: Control Registers, cont.

%cr3 contains the address of the current page directory, as well as a couple paging related flags

%cr4 contains… more flags (not as interesting though)
- Protected mode virtual interrupts?
- Virtual-8086 mode extensions?
- Most of these are not usefully modified...

  ...but you should make an inventory.

Mundane Details in x86: Registers

How do you write to a special register?

Most of them: **`movl`** instruction

Some (like %cr's) you need PL0 to access

We provide assembly wrappers for some

- Maybe we should skip some!
- Think about each before using.

EFLAGS is a little different, but you may not be writing directly to it anyway

Mundane Details in x86: The Life of a Memory Access

Logical Address (consists of 16 bit segment selector, 32 bit offset)

Segmentation

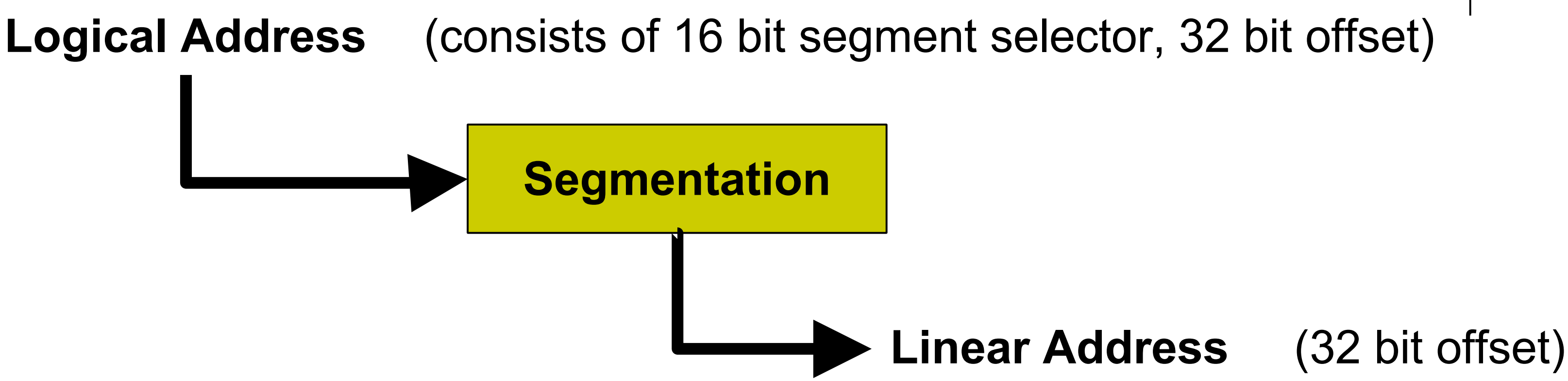
Linear Address (32 bit offset)

Paging

Physical Address

(32 bit offset)

Mundane Details in x86: The Life of a Memory Access

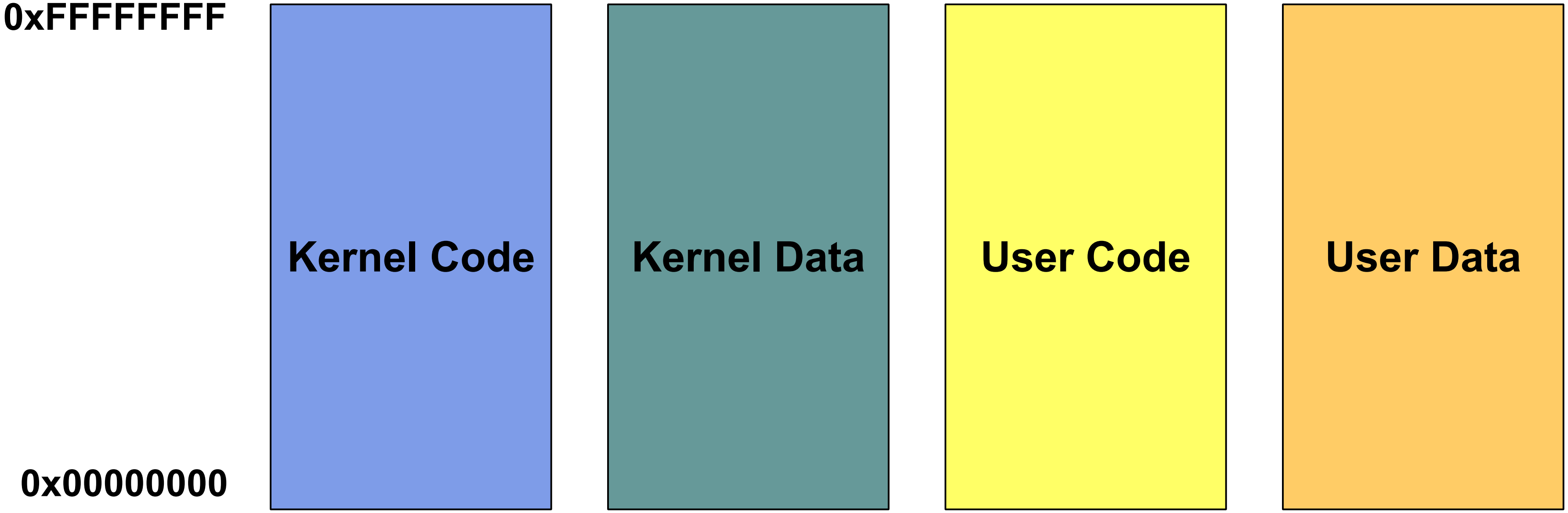


- The 16 bit segment selector comes from a segment register (%CS & %SS implied)
- The 32 bit offset is added to the base address of the segment
- That gives us a 32 bit offset into the virtual address space

Mundane Details in x86: Segmentation

Segments need not be backed by physical memory and can overlap

Segments defined for these projects:

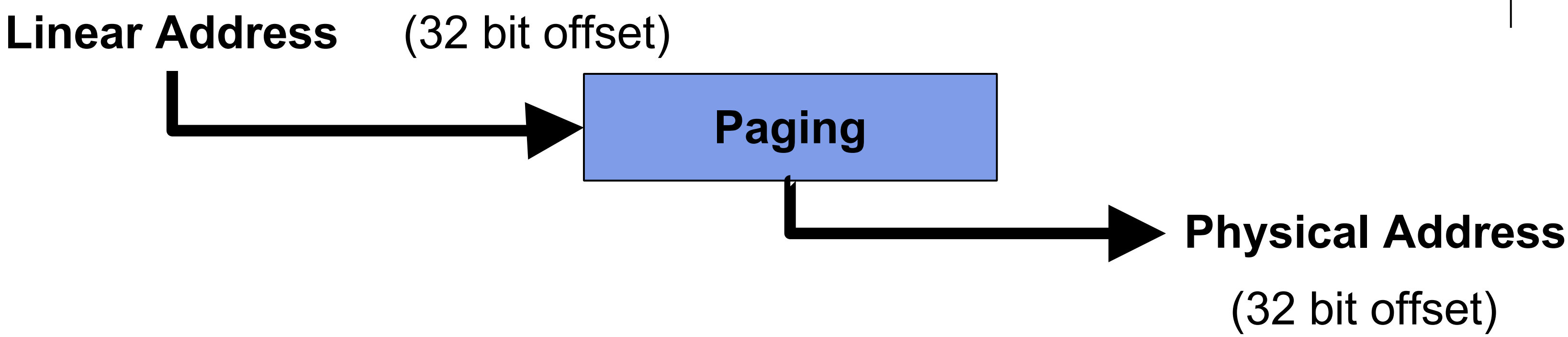


Mundane Details in x86: Segmentation

For Project 3 we are abusing segmentation

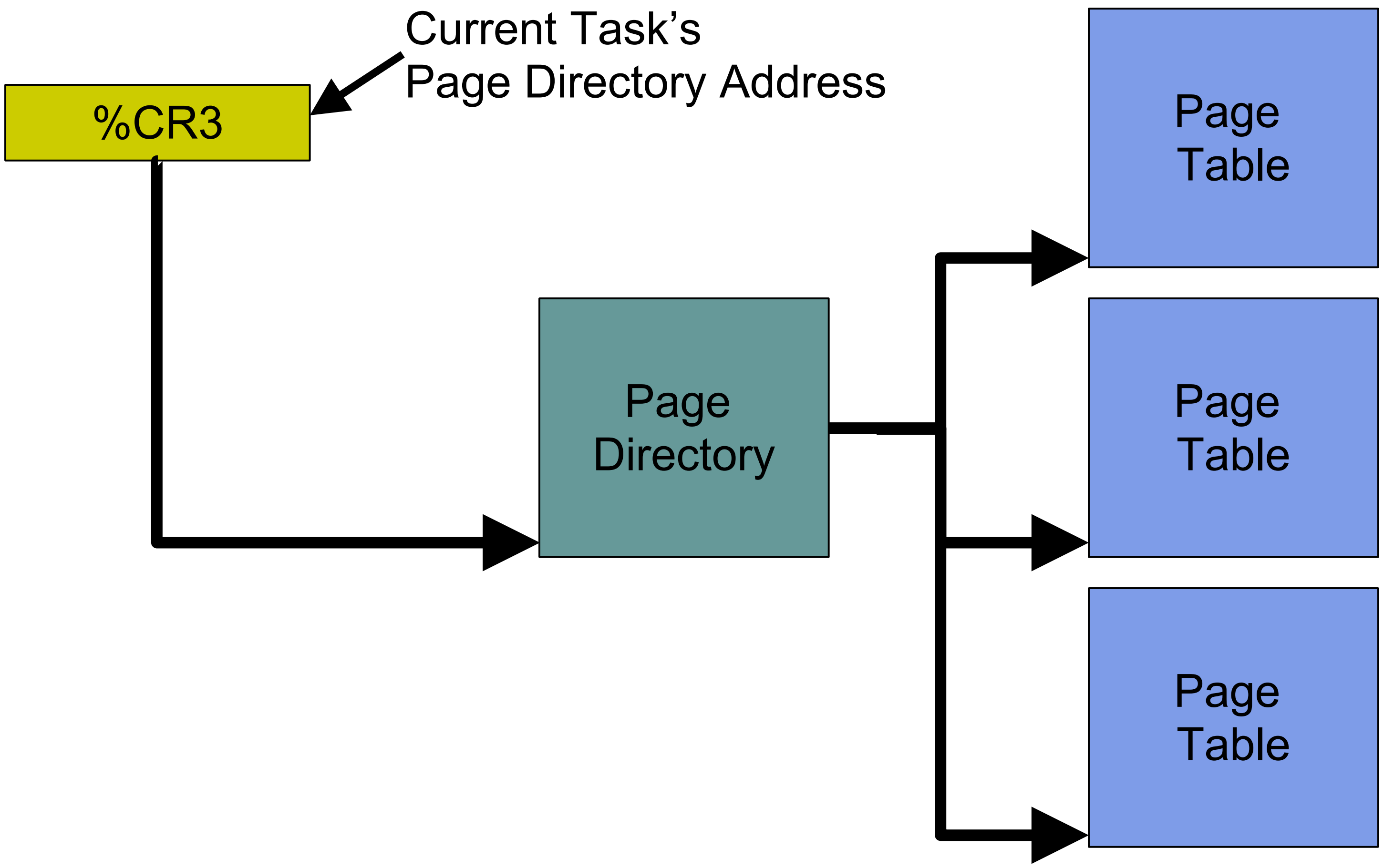
- All segments “look the same”
- Each linear address is just the “low-order 32 bits” of the logical address
- Confusing, but simplifies life for you
- See 15-410 segmentation guide on web site

Mundane Details in x86: The Life of a Memory Access

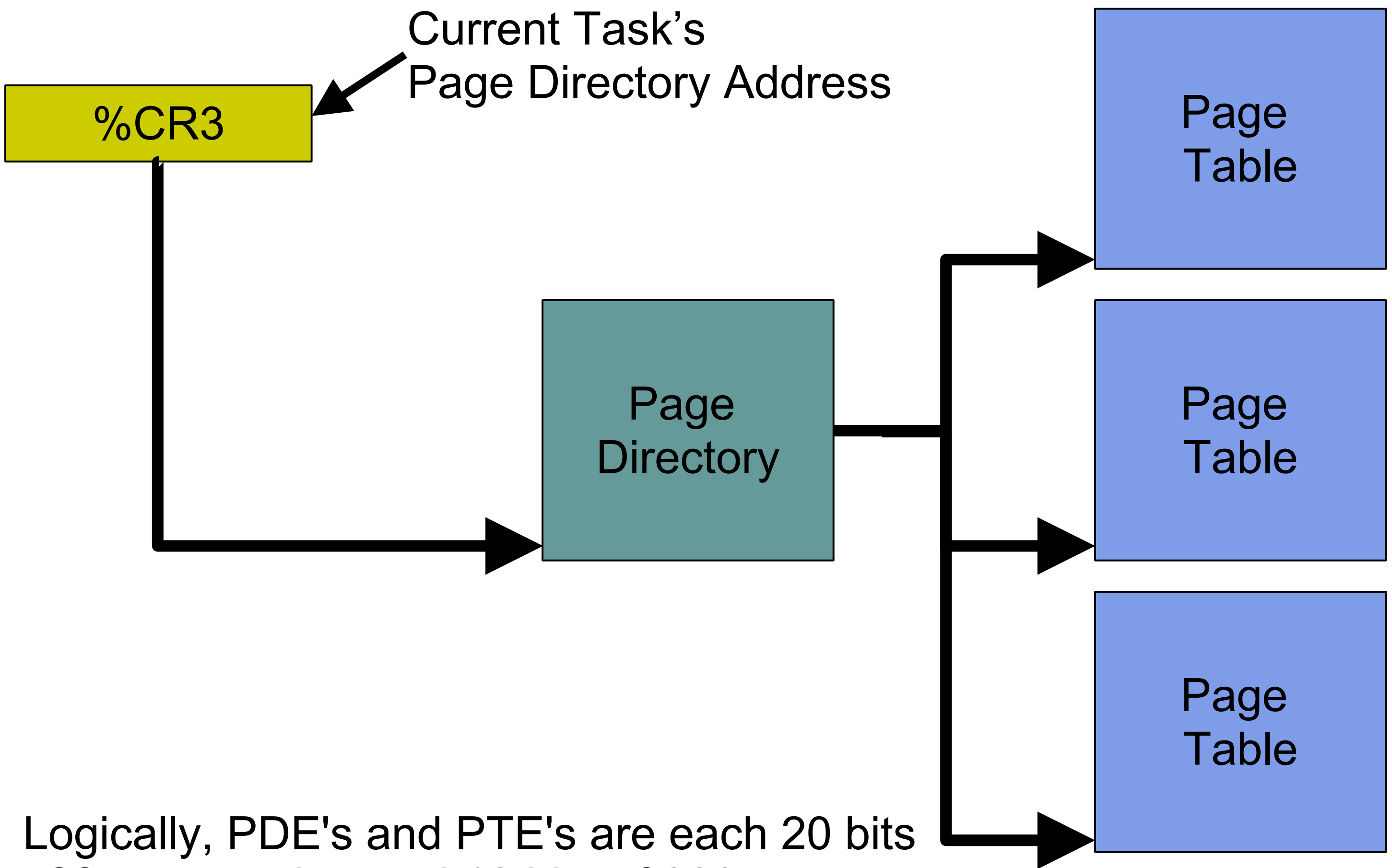


- Top 10 bits index into page directory, point us to a page table
- The next 10 bits index into page table, point us to a frame
- The last 12 bits are an offset into that page (and frame)

Mundane Details in x86: Page Directories and Tables



Mundane Details in x86: Page Directories and Tables



Logically, PDE's and PTE's are each 20 bits of frame number and 12 bits of 000.

Mundane Details in x86: Page Directory

- The page directory is 4k in size
- Contains pointers to page tables
- Entries may be invalid (see “P” bit)

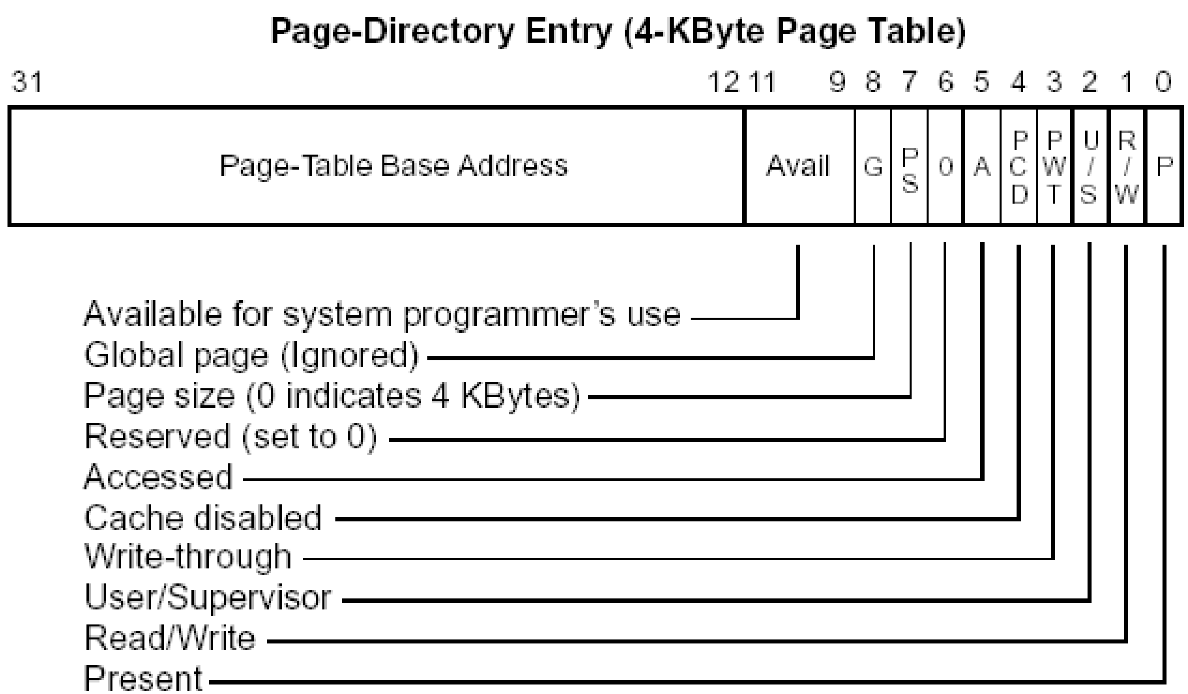


Figure from page 87 of intel-sys.pdf
This a jumping-off point!

Mundane Details in x86: Page Table

- Each page table is also 4k in size
- Contains pointers to pages
- "P" bit again

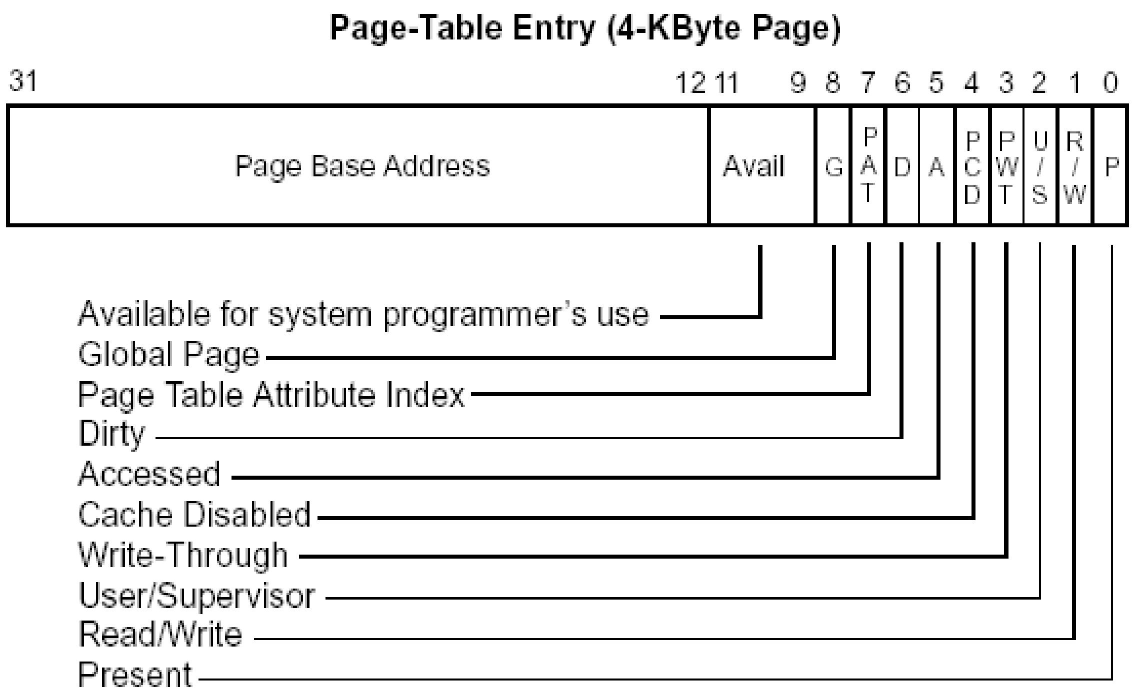


Figure from page 87 of intel-sys.pdf
This a jumping-off point!

Mundane Details in x86: The Life of a Memory Access

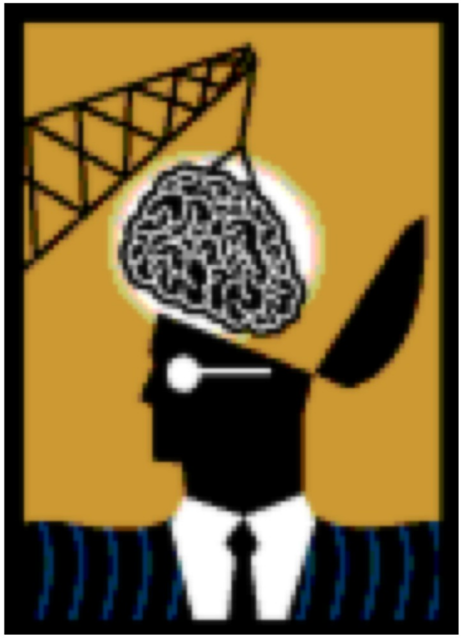
- Whoa there, Slick… What if the page directory entry isn't there?
- What happens if the page table entry isn't there?
- It's called a page fault, it's an exception, and it lives in IDT entry 14
- You will have to write a handler for this exception and do something intelligent

Mundane Details in x86: Context Switching

We all know that threads take turns running on the CPU

This means they have to be stopped and started over and over

How?



Mundane Details in x86: Context Switching

The x86 provides a hardware “task” abstraction

- This makes context switching “easy”

But...

- Often faster to manage processes in software
- We can also tailor our process abstraction to our particular needs
- Our OS is more portable if it doesn't rely on one processor's notion of “task”

Protected mode <u>requires</u> one hardware task

- Already set up by 410 boot code

Mundane Details in x86: Context Switching

- Context switching is a very delicate procedure
- Great care must be taken so that when the thread is restarted, it does not know it ever stopped
- "User" registers must be exactly the same (%cr3 is the key non-user register)
- Its stack must be exactly the same
- Its page directory must be in place
- Please <u>carefully heed</u> the handout warnings!

Mundane Details in x86: Context Switching

Hints on context switching:

- Use the stack, it is a convenient place to store things
- If you do all your switching in one routine, you have eliminated one thing you have to save (%eip)
- New threads will require some special care
  - Try to confine new-thread code; don't infect your beautiful pure context-switcher

Mundane Details in x86: System Calls

System calls use "software interrupts"

- Which are <u>not</u> actually interrupts!
  - They are immune to disable_interrupts()
    - Which *defers*, not disables, anyway!

Mundane Details in x86: System Calls

System calls use “software interrupts”

- Which are not actually interrupts!
  - They are immune to disable_interrupts()
    - Which *defers*, not disables, anyway!

Install handlers much as you did for the timer, keyboard

Calling convention specified in handout

- Matches P2

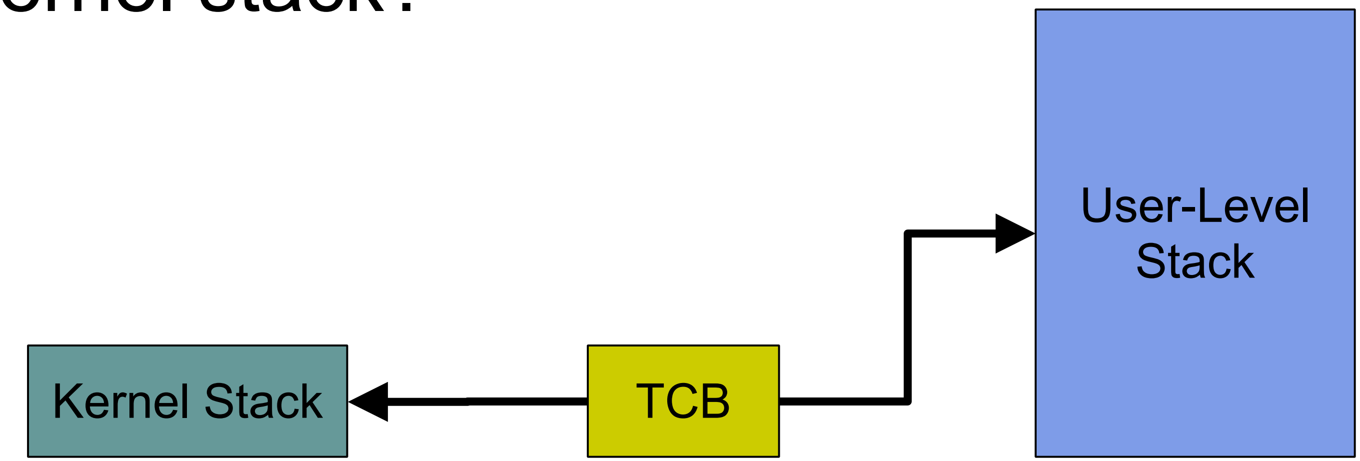
If you are rusty on the IDT refer back to P1

Mundane Details in x86: Kernel Stacks

User threads have a separate stack for their kernel activities

Located in kernel space

How does the stack pointer get switched to the kernel stack?



Mundane Details in x86: Kernel Stacks

When the CPU switches from user mode to kernel mode the stack pointer is changed

The new (kernel) stack pointer to use is stored in the configuration of the CPU hardware task

- Remember: we use only one "x86 task"

We provide a function to change this value set_esp0(void* ptr)

Used during <u>next</u> user ⇒ kernel transition

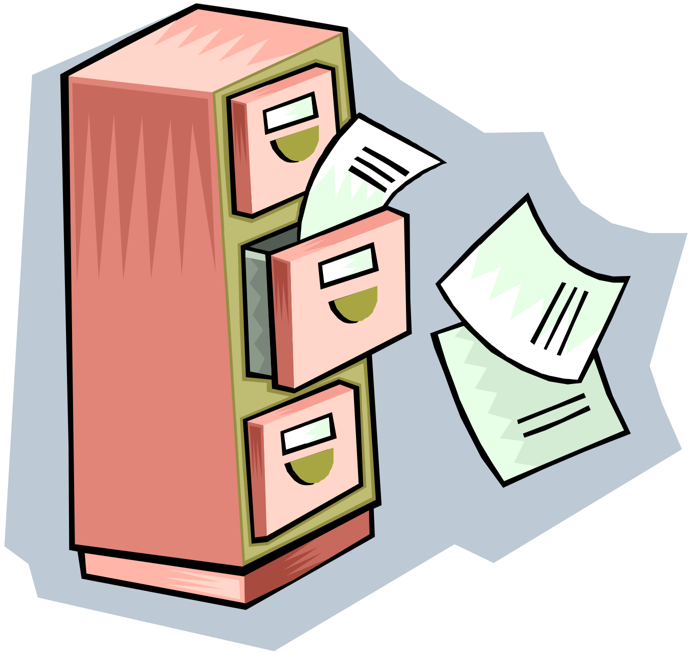
- So set_esp0() "does nothing" (until later)

Loading Executables

Same approach as P2

"RAM disk" file system

But you must write a loader



Loading Executables: The Loader

RAM-disk bytes are part of the kernel data area

You need to load them into the task's address space

Code, rodata, data, bss, stack – all up to you!

Executables will be in "simple ELF" format

References to resources are in the handout

Encapsulation!!!!!

You will re-implement chunks of your kernel

It will be painful if code is holographic

Don't "use a linked list of threads"

Do define a thread-list interface

- find(), append(), first(), ...

You may need to add a method...

- ...which changes the implementation entirely...
- But most existing interface uses (calls) will be ok

Machine State Summary

256 MB RAM, keyboard, console, timer

IDT

CPU state

- General-purpose registers
- Segment registers
- EFLAGS, cr0...cr4, esp0

We set up for you

- Hardware task
- GDT (global descriptor table) – 4 segments

Warning

Attack Strategy

- There is an attack strategy in the handout
- It represents where we think you should be in particular weeks
- You <u>WILL</u> have to turn in checkpoints
- <u>Excellent</u> data indicate...
  - Missing one checkpoint is dangerous...don't miss two!

Attack Strategy

Please read the handout a couple times over the next few days

Create doxygen-only files

- scheduler.c, process.c, ...
- Document major functions
- Document key data structures
- A <u>very</u> iterative process

Suggestion: doxygen tentative responsibilities

- For a good time, estimate #lines, #days

Partnership

Make an explicit partnership plan

- How often you'll meet, for how long
  - Regular, fixed meetings are vital!
- Information flow
  - When will you read each other's code?

- Meeting agenda suggestions
  - Last time's open issues
  - New issues
  - Who will do what by next meeting?

Grading Approach

These numbers are not final!

Weight	*Section*
5	Kernel builds as directed
45	Shell loads, runs test programs
10	Concurrency
10	Style/structure
10	Basic tests
15	Non-basic tests
5	Thread tests (not using your P2)

“Hurdle” Model

We will release a test suite

- ~15 “basic” tests
- ~15 “solidity” tests
- ~2 “stability” tests

Successful completion of Project 3 requires

- ~80% of each section of test suite
- Acceptable preemptibility and robustness

You will self-test your P3 when you turn it in

“Hurdle” Model

Leap the P3 hurdle?

- Work on Project 4
  - ~2 weeks after P3
  - ~5% of course grade
- A modification/extension of your kernel
  - Goal: “interesting”, more than “hard”

Thwarted?

- Extra time for P3 (~1 week)
- 0% will be assigned for P4 grade

Warning!

To continue to P4, kernel must be complete

- We will publish criteria
- Seemingly “trivial” things on the checklist cost 20% of grade!

P3extra is *not optional* if kernel isn't complete

- We won't assign a P4 grade, so p3extra is the only option

This is serious

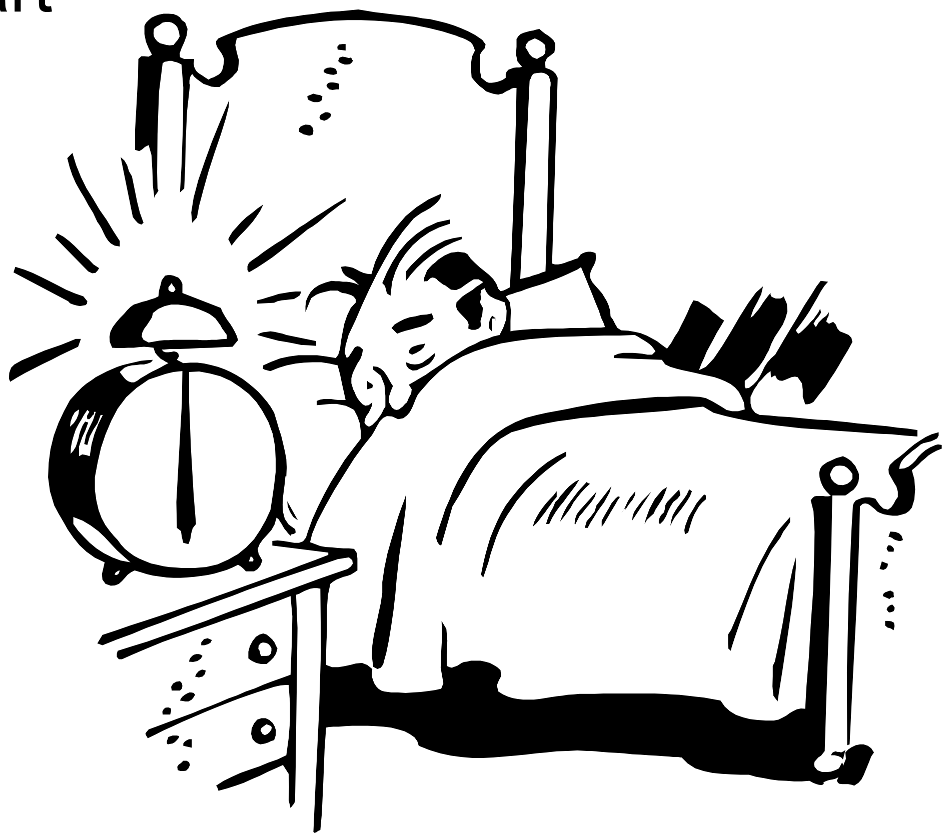
- Please be serious about it

A Quick Debug Story

Ha! You'll have to have been to lecture to hear this story.

A Quick Debug Story

The moral is, please start early.



Our Hopes for You

Project 3 can be a <u>transformative experience</u>

- You may become a different programmer
  - Techniques, attitudes

Employers care about this experience

Alumni care about this experience

#include <end_of_412_concern_stories>

Exhortation

Please read the project handout ASAP!

You need to <u>plan</u> how to get to Checkpoint 1

- Simple loader
- Dummy VM
    - *please* write (encapsulated) bad code!!
- Getting from kernel mode to user mode
- Getting from user mode to kernel mode
- Lots of faults
    - Solving them will require “story telling”

Encouragement

This can be done

Stay on track

- Make all checkpoints
- Don't ignore the plan of attack
- Don't postpone merges

It is possible for everybody to finish the project

- Perhaps if weird practices are engaged?
- Let's do it!

Good Luck on Project 3!