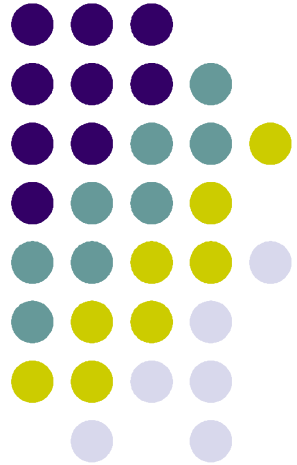
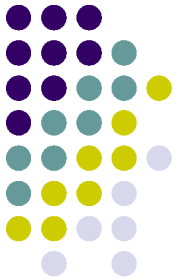
What You Need to Know for Project Three

Dave Eckhardt
Steve Muckle



Overview



Introduction to the Kernel Project

Mundane Details in x86

- registers, paging, the life of a memory access, context switching, system calls, kernel stacks

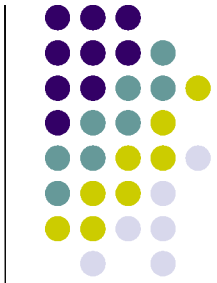
Loading Executables

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

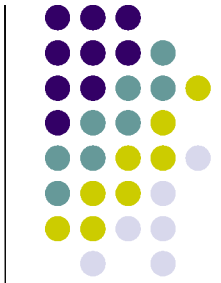
- Stack, registers, page tables, scheduling, races...

You will become one with program execution

P1: living without common assumptions

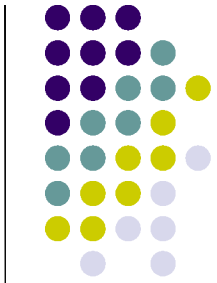
P3: providing assumptions to users

The P3 Experience



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

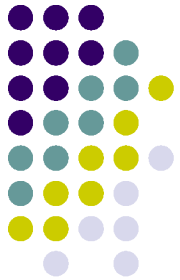
Introduction to the Kernel Project: Kernel Features



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

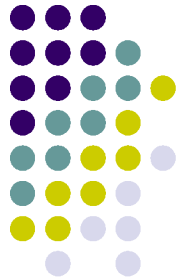


Preemptive multitasking is forcing multiple user processes to share the CPU

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 before committing to a design

Context switching is tricky but cool

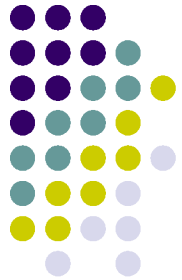
As in P2, creating a new process/thread is hard

- Especially given memory sharing

As in P2, exit is tricky too

- At least one “How can I do that???” question

Introduction to the Kernel Project: Multiple Virtual Address Spaces



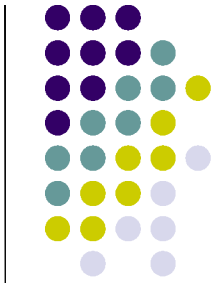
The x86 architecture supports paging

You will use this to provide a virtual address space for each user process

Each user process will be isolated from others

Paging will also protect the kernel from users

Introduction to the Kernel Project: System Calls



You used them in P2

Now you get to implement them

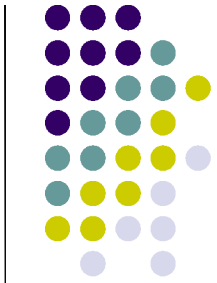
Examples include fork, exec, and thread_fork

There are easier ones like getpid

The core cluster – ***must*** work solidly

- fork(), exec()
- exit(), wait()

Mundane Details in x86



We looked at some of these for P1

Now it is time to get the rest of the story

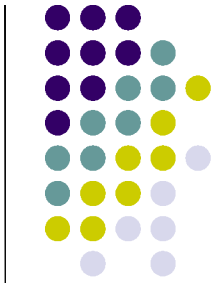
How do we control processor features?

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)

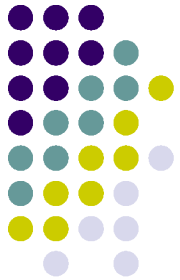
Segment registers (somewhat interesting)
- %cs, %ss, %ds, %es, %fs, %gs

%eip (a little interesting)

EFLAGS (interesting)

Control Registers (very interesting)
- %cr0, %cr1, %cr2, %cr3, %cr4
- esp0 field in hardware task

Mundane Details in x86: General Purpose Registers



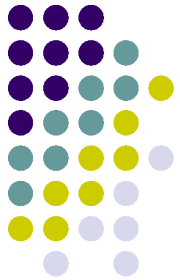
The most boring kind of register

%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

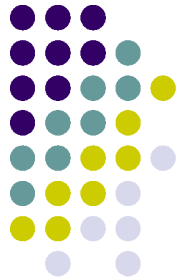
%cs specifies the segment used to access code (also specifies privilege level)

%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!!!

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



It's interesting

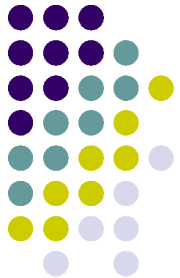
Cannot be read from or written to directly

- (branch, call, return)

Controls which instructions get executed

'nuff said.

Mundane Details in x86: The EFLAGS Register



It's interesting

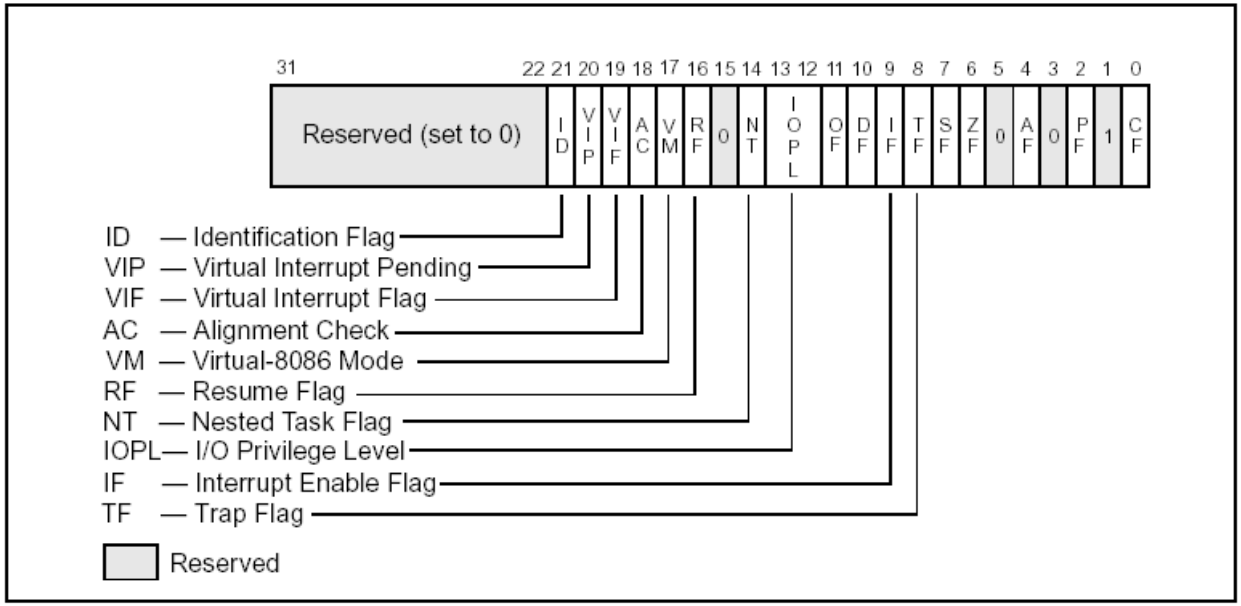
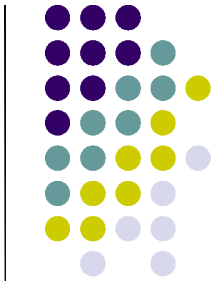


Figure 2-3. System Flags in the EFLAGS Register

Flag city, including interrupt-enable, arithmetic flags

- "Alignment check" off

Mundane Details in x86: Control Registers



Very interesting!

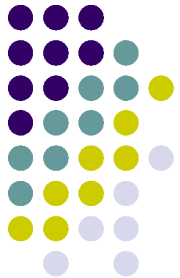
An assortment of important flags and values

%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.

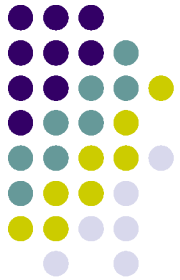


%cr3 contains the address of the current page directory, as well as a couple paging related flags

%cr4 contains… more flags (not as interesting though)

- protected mode virtual interrupts?
- virtual-8086 mode extensions?
- No thanks

Mundane Details in x86: Registers



How do you write to a special register?

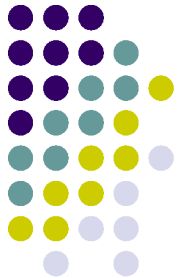
Most of them: `movl` instruction

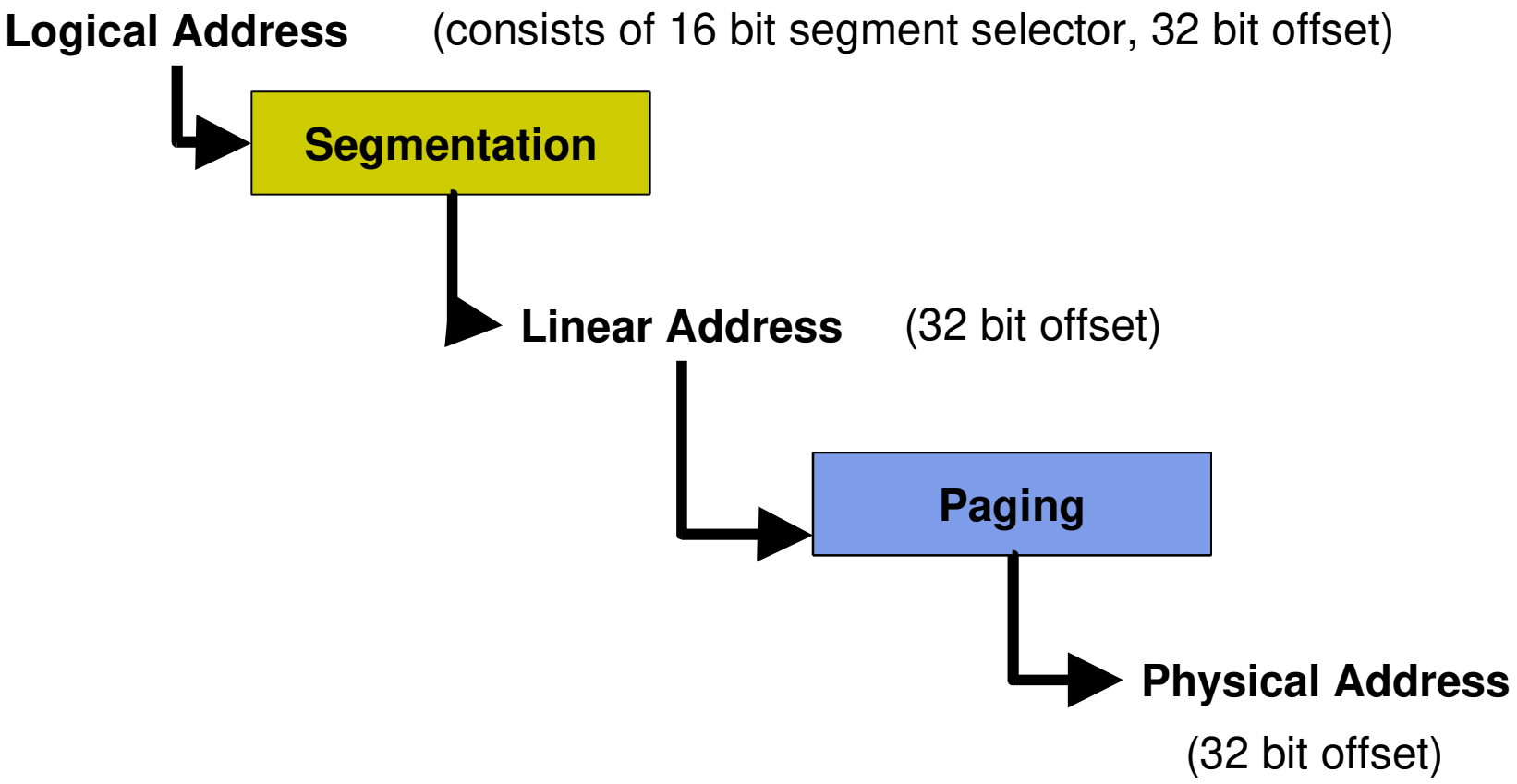
Some (like CRs) you need PL0 to access

We will provide inline assembly wrappers

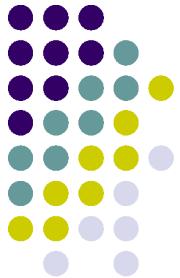
EFLAGS is a little different, but you will not be writing directly to it anyway

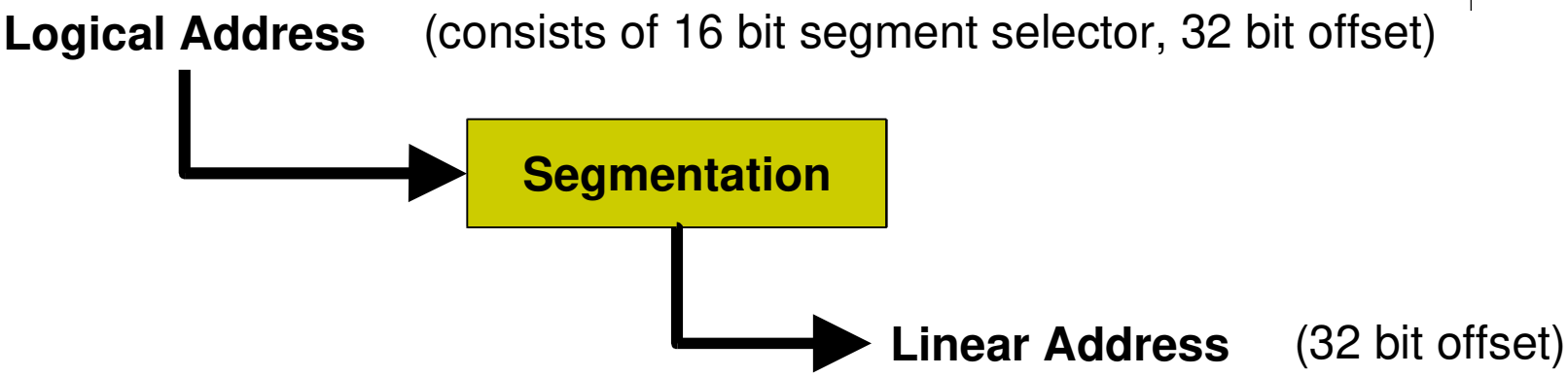
Mundane Details in x86: The Life of a Memory Access





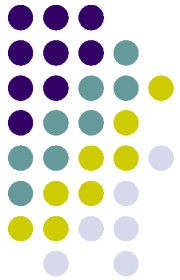
Mundane Details in x86: The Life of a Memory Access





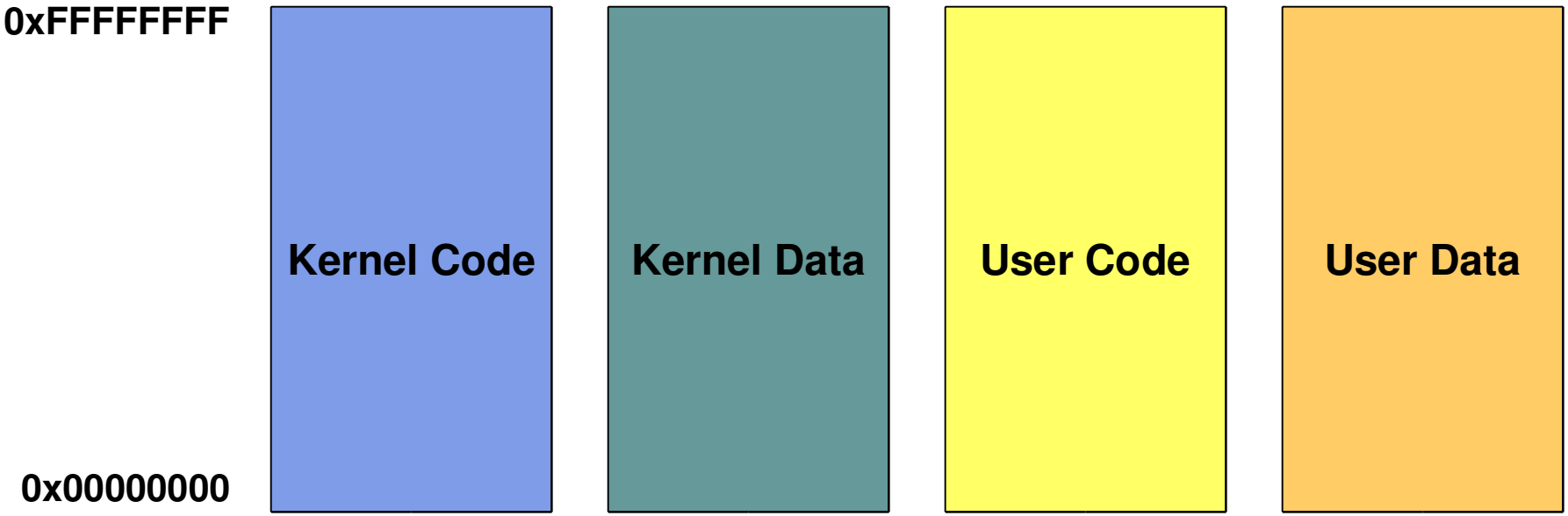
- The 16 bit segment selector comes from a segment register (%CS, %SS implied)
- The 32 bit offset is added to the base address of the segment
- That gives us a 32 bit offset into the virtual address space

Mundane Details in x86: Segmentation

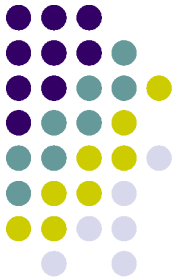


Segments need not be backed by physical memory and can overlap

Segments defined for these projects:



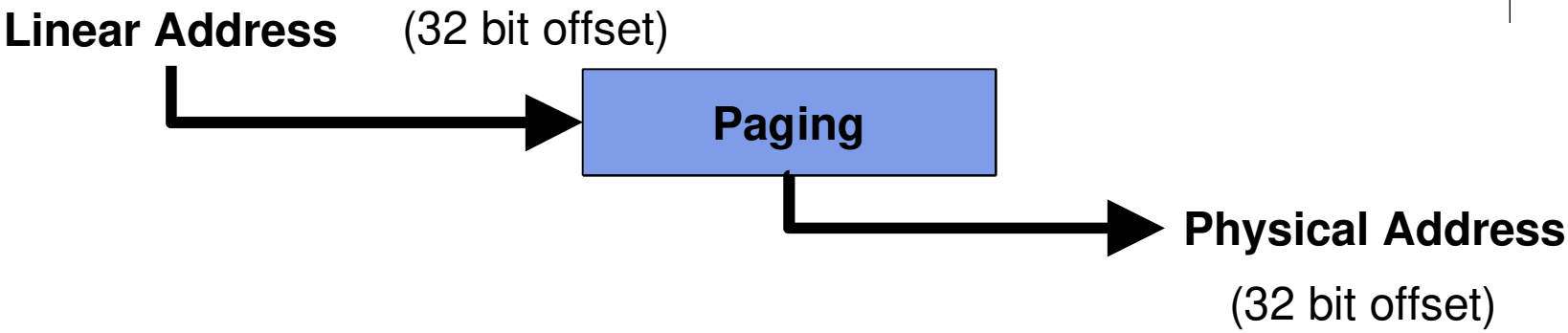
Mundane Details in x86: Segmentation



For Project 3 we are abusing segmentation

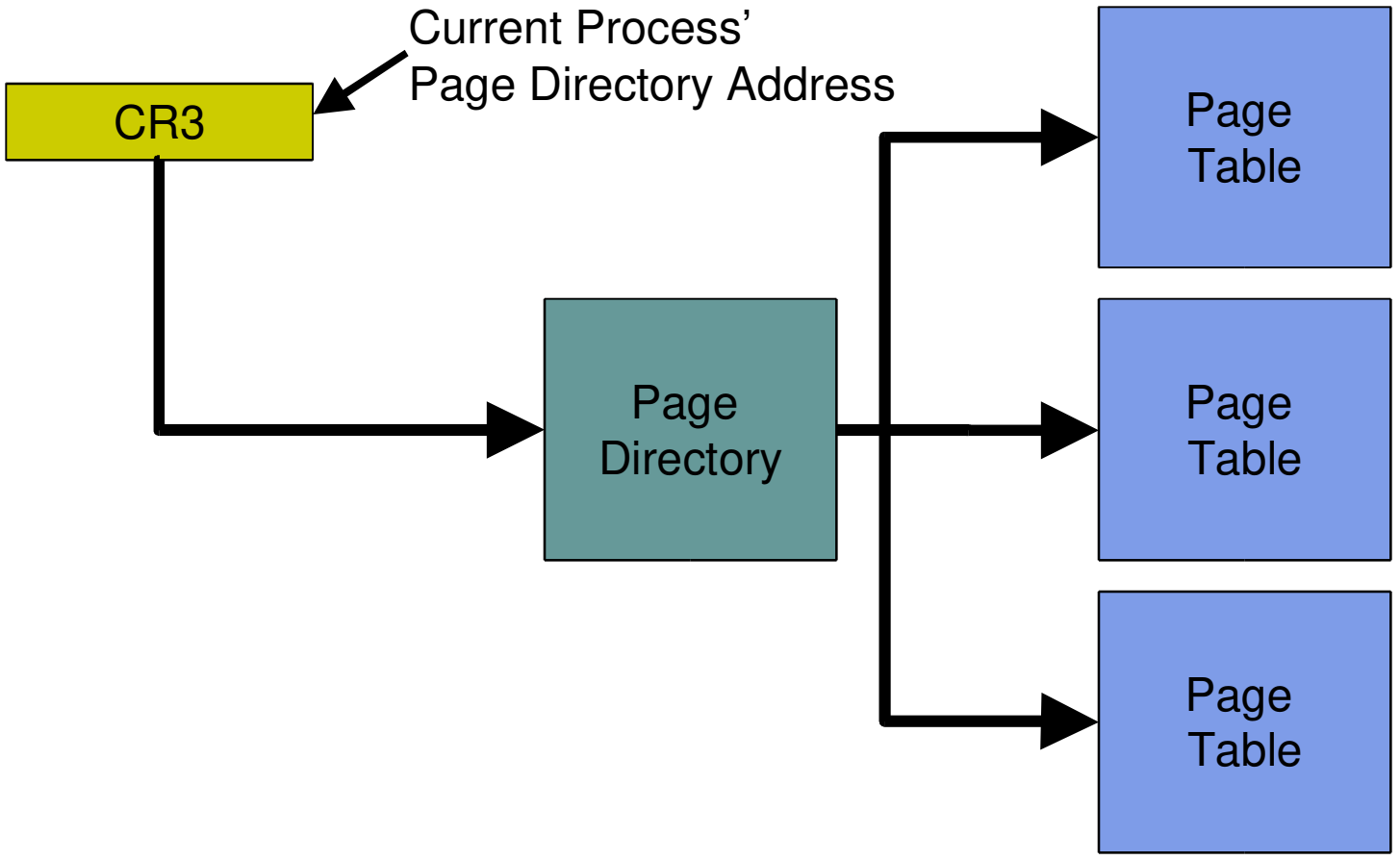
- All segments “look the same”
- Confusing, but simplifies life for you
- See 15-410 segmentation guide on web site

Mundane Details in x86: The Life of a Memory Access

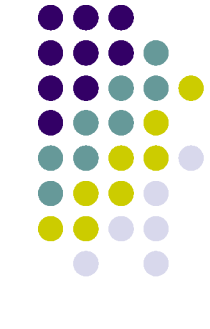


- Top 10 bits index into page directory, point us to a page table
- The next 10 bits index into page table, point us to a page
- The last 12 bits are an offset into that page

Mundane Details in x86: Page Directories and Tables



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)

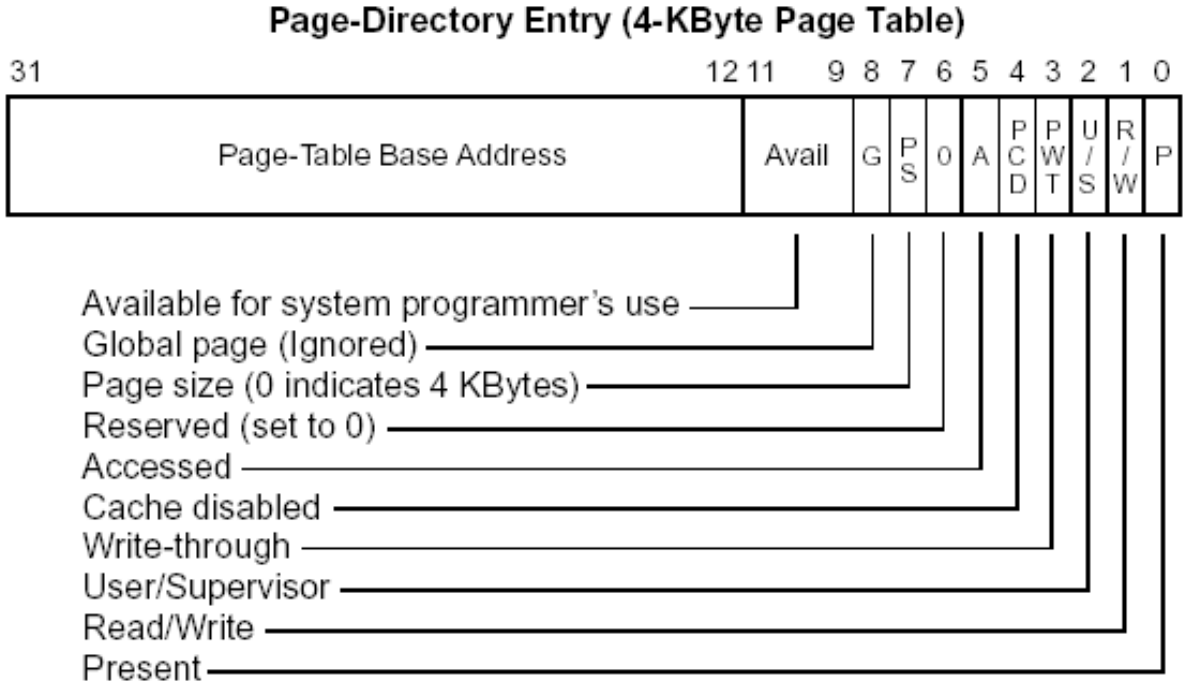
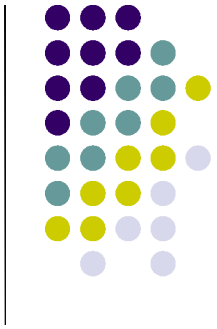


Figure from page 87 of intel-sys.pdf

Mundane Details in x86: Page Table



The page table is also 4k in size

Contains pointers to pages

“P” bit again

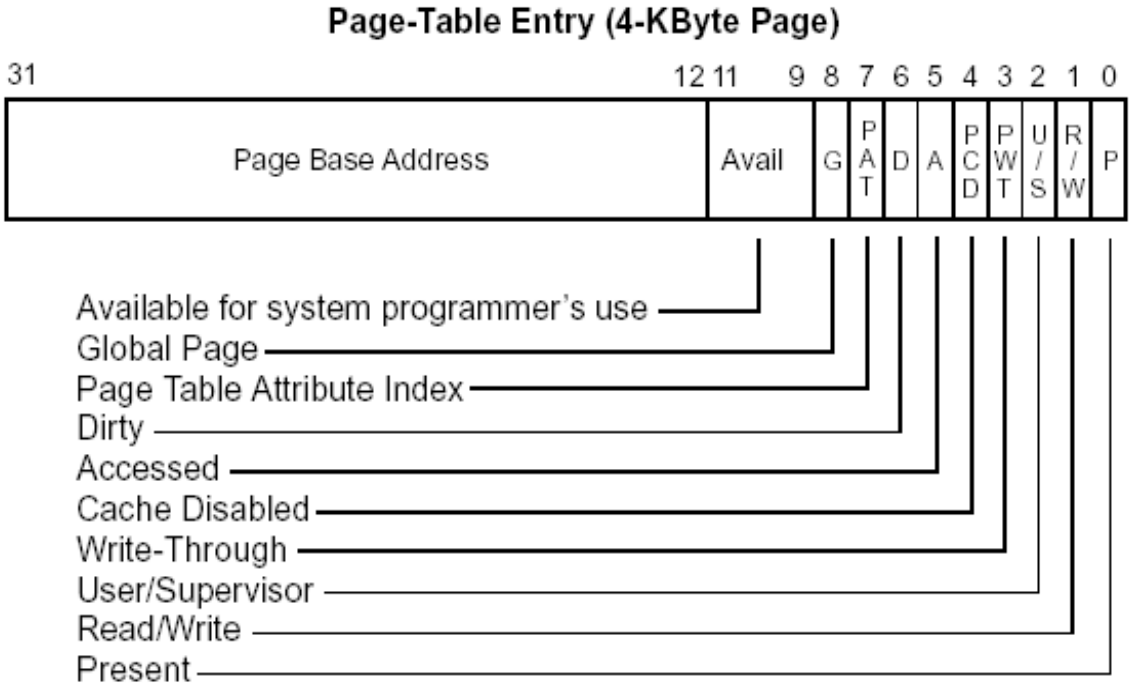
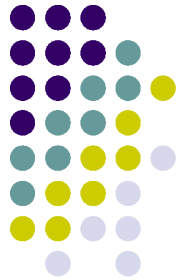


Figure from page 87 of intel-sys.pdf

Mundane Details in x86: The Life of a Memory Access



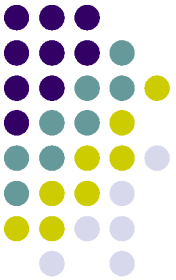
Whoa there slick… What if the page directory entry isn't there?

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 13

You will have to write a handler for this exception and do something intelligent

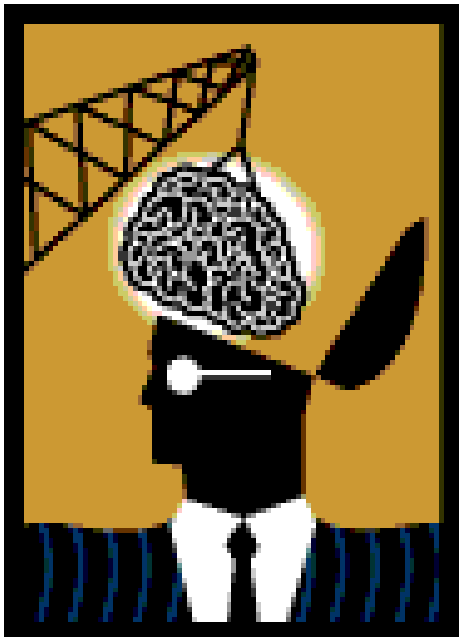
Mundane Details in x86: Context Switching



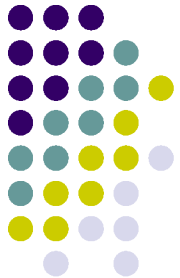
We all know that
    processes take turns running on the CPU

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

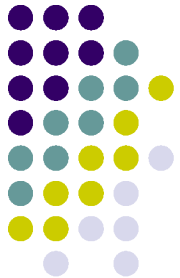
- This makes context switching easy

But...

- It is actually faster to manage processes in software
- We can also tailor our process abstraction to our particular needs

You must have at least one hardware task defined, we take care of this for you

Mundane Details in x86: Context Switching



Context switching is a very delicate procedure

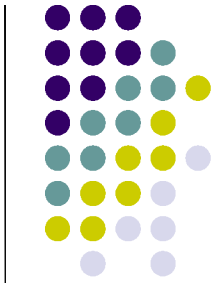
Great care must be taken so that when the process is started, it does not know it ever stopped

Registers must be exactly the same (%cr3 is the only control register you have to update)

Its stack must be exactly the same

Its page directory must be in place

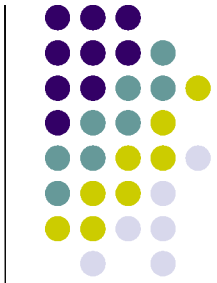
Mundane Details in x86: Context Switching



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 processes will require some special care

Mundane Details in x86: System Calls



System calls use software interrupts

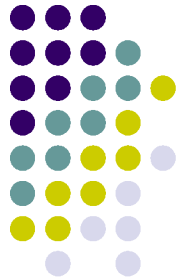
Install handlers just as you did for the timer, keyboard

Calling convention specified in handout

- Matches P2

If you are rusty on the IDT refer back to P1

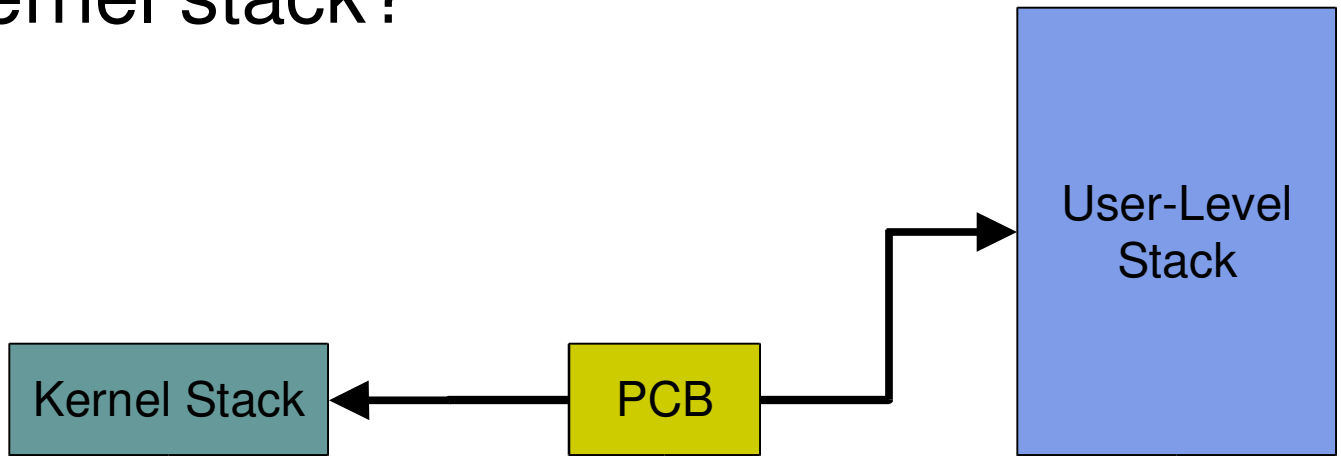
Mundane Details in x86: Kernel Stacks



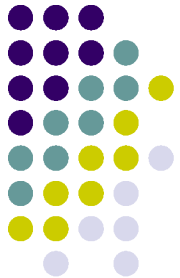
User processes have a separate stack for their kernel activities

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

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

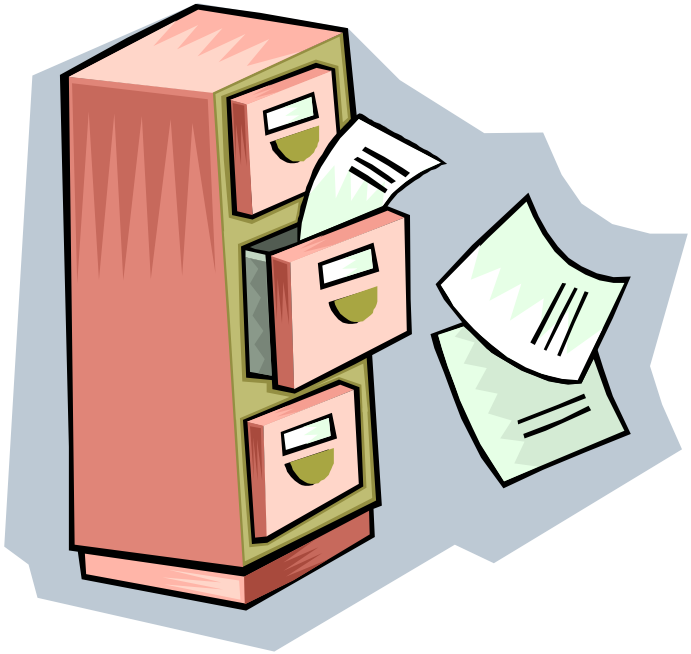
Used during next user⇒kernel transition

Loading Executables

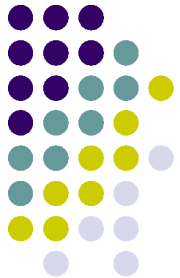
Same approach as P2

"RAM disk" file system

But you must write a loader



Loading Executables: The Loader



You have access to the bytes

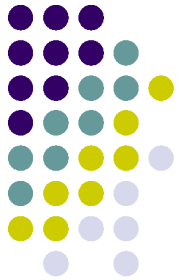
You need to load them into the process' address space

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 processes”

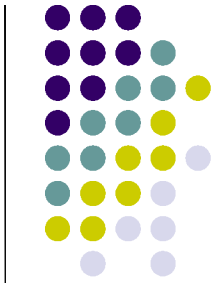
Do define a process-list interface

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

You may need to add a method...

- ...that 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

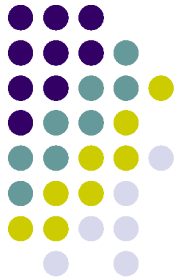
Attack Strategy

There is an attack strategy in the handout

It represents where we think you should be in particular weeks

You WILL have to turn in checkpoints

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 very iterative process

Suggestion: doxygen tentative responsibilities

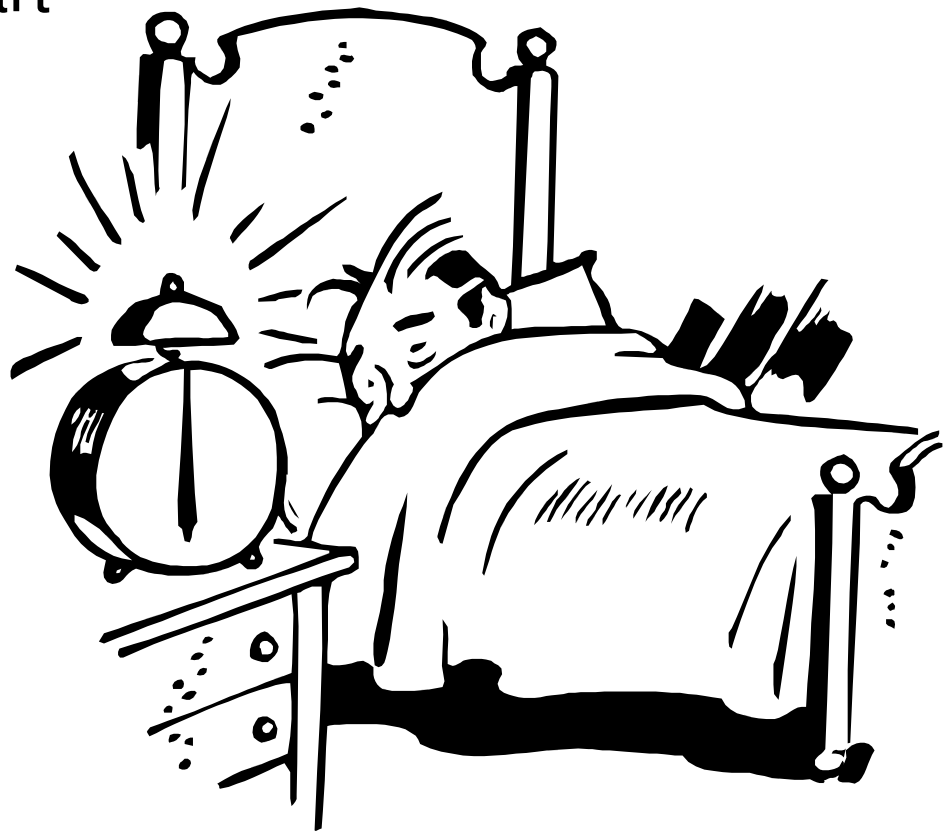
- For a good time, estimate #lines, #days

A Quick Debug Story

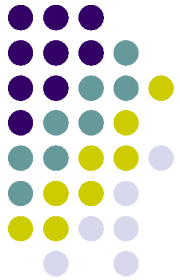
Ha! You'll have to have been to lecture to hear this story.

A Quick Debug Story

The moral is, please start early.



Our Hopes for You



Project 3 can be a transformative experience

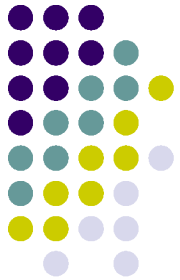
- You may become a different programmer
  - Techniques, attitudes

Employers care about this experience

Alumni care about this experience

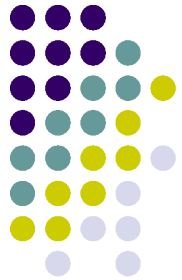
#include <end_of_412_concern_stories>

Exhortation



Please read the project handout soon!

You need to plan how to get to Checkpoint 1



Good Luck on Project 3!