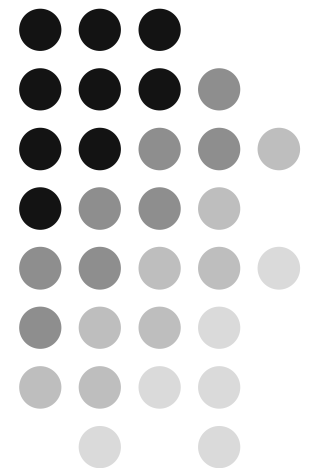
What You Need to Know for Project Three

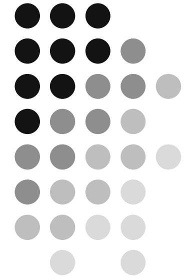
Steve Muckle

Wednesday, February 19th 2003

15-412 Spring 2003

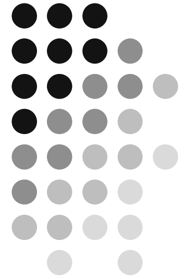


Overview



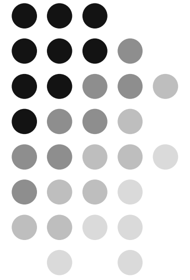
- Introduction to the Kernel Project
- Mundane Details in x86
  registers, paging, the life of a memory access, context switching, system calls, kernel stacks
- Loading Executables
- A Quick Debug Story
- Style Recommendations (or pleas)
- Attack Strategy

Introduction to the Kernel Project



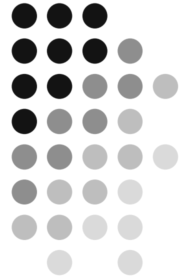
- P3 is the most conceptually challenging
- You will need to adjust how you think about program execution
- P1 introduced you to programming without making commonly made assumptions
- In P3 you need to provide assumptions to users

Introduction to the Kernel Project: Kernel Features



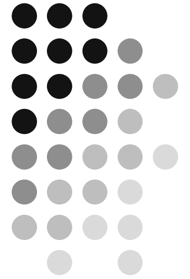
- Your kernels will feature:
  - preemptive multitasking
  - multiple virtual address spaces
  - a “small” selection of useful system calls
  - robustness (hopefully)

Introduction to the Kernel Project: Preemptive Multitasking



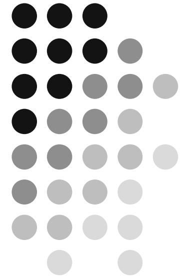
- Preemptive multitasking is forcing multiple user processes to share the CPU
- You will use the timer interrupt to do this
- Reuse your timer code from P1 if possible

Introduction to the Kernel Project: Preemptive Multitasking



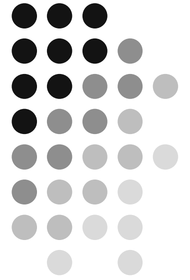
- Simple round robin scheduling will suffice
- Context switching is tricky but cool

Introduction to the Kernel Project: Multiple Virtual Address Spaces



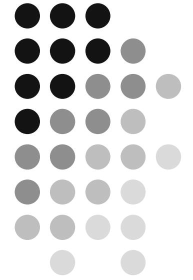
- The x86 architecture supports paging
- You will use this to provide a virtual address space for each user process
- Each user process will be isolated from other user processes
- We will also use paging to provide protection for the kernel

Introduction to the Kernel Project: System Calls



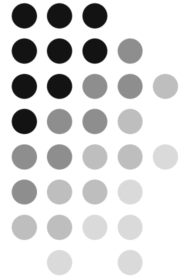
- You used them in P2
- Now you get to implement them
- Examples include fork, exec, and of course, minclone
- There are easier ones like getpid

Mundane Details in x86



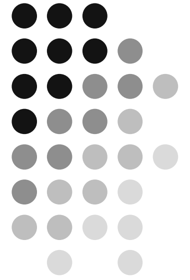
- We looked at some of these for P1
- Now it is time to get the rest of the story
- How do we control processor features?
- What does an x86 page table look like?
- What route does a memory access take?
- How do you switch from one process to another?

Mundane Details in x86: Registers



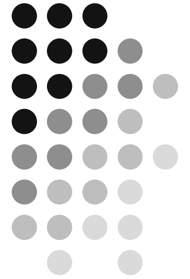
- General purpose regs (not interesting)
  - %eax, %ebx, %ecx, etc…
- Segment Selectors (somewhat interesting)
  - %cs, %ss, %ds, %es, %fs, %gs
- %eip (interesting)
- EFLAGS (interesting)
- Control Registers (very interesting)
  - %cr0, %cr1, %cr2, %cr3, %cr4

Mundane Details in x86: General Purpose Registers



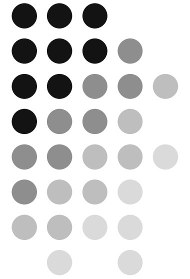
- The most boring kind of register
- %eax, %ebx, %ecx, %edx, %edi, %esi, %ebp, %esp
- %eax, %ebp, and %esp are exceptions, they are slightly interesting
  - %eax is used for return values
  - %esp is the stack pointer
  - %ebp is the base pointer

Mundane Details in x86: Segment Selector Registers



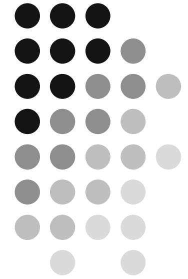
- Slightly more interesting
- %cs specifies the segment used to access code (also specifies privilege level)
- %ss specifies the segment used for stack related operations (pushl, popl, etc)
- %ds, %es, %fs, %gs specify segments used to access regular data
- Mind these during context switches…

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



- It's interesting
- Cannot be read from or written to
- Controls what instructions get executed
- ‘nuf said.

Mundane Details in x86: The EFLAGS Register



- It's interesting

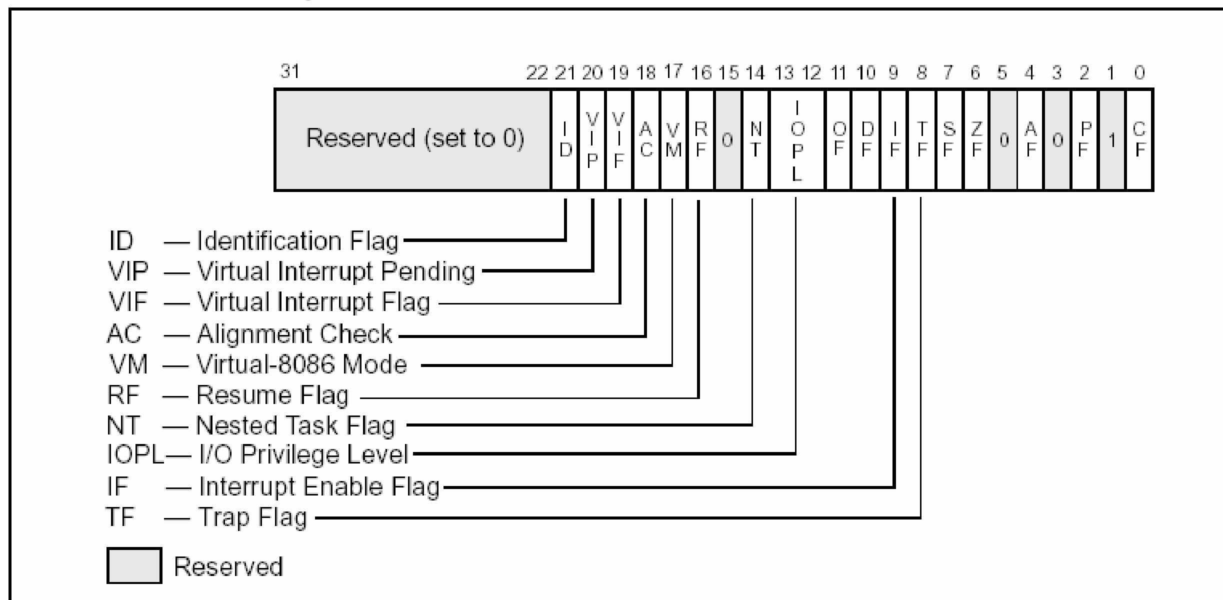
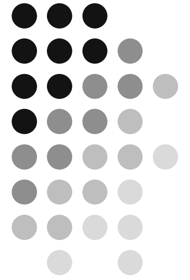


Figure 2-3. System Flags in the EFLAGS Register

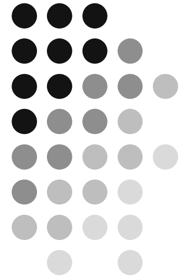
- Contains a bunch of flags, including interrupt-enable, arithmetic flags

Mundane Details in x86: Control Registers



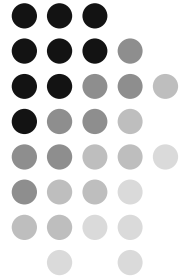
- Very interesting!
- An assortment of important flags and values
- %cr0 contains powerful system flags that control things like paging, protected mode
- %cr1 is reserved (now that’s really interesting)
- %cr2 contains the address that caused the last page fault

Mundane Details in x86: Control Registers, cont'



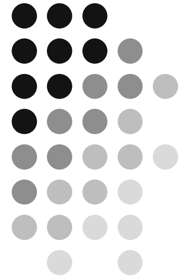
- %cr3 contains the address of the current page directory, as well as a couple paging related flags
- %cr4 contains… more flags (not as interesting though)
  - protected mode virtual interrupts?
  - virtual-8086 mode extensions?
  - No thanks

Mundane Details in x86: Registers



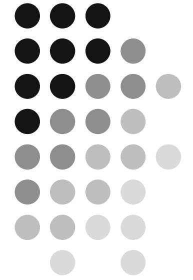
- How do you write to a special register?
- Most of them can simply be written to using the movl instruction
- Some (like CRs) you need PL0 to access
- We will provide inline assembly wrappers
- EFLAGS is a little different, but you will not be writing to it anyway

Mundane Details in x86: Paging



- The x86 offers several page sizes
- We will use 4k pages
- The x86 uses a two level paging scheme
- The top of the paging structure is called a page directory
- The second level structures are called page tables

Mundane Details in x86: Page Directories and Tables



Current Process' Page Directory Address

CR3

Page Directory

Page Table

Page Table

Page Table

Mundane Details in x86: Page Directory

- The page directory is 4k in size
- Contains pointers to page tables
- Not all entries have to be valid

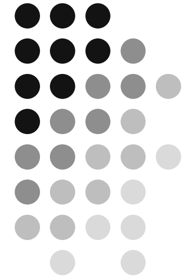
Page-Directory Entry (4-KByte Page Table)

31–12	11–9	8	7	6	5	4	3	2	1	0
Page-Table Base Address	Avail	G	PS	0	A	PCD	PWT	U/S	R/W	P

Available for system programmer's use
Global page (Ignored)
Page size (0 indicates 4 KBytes)
Reserved (set to 0)
Accessed
Cache disabled
Write-through
User/Supervisor
Read/Write
Present

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
- Not all entries have to be valid

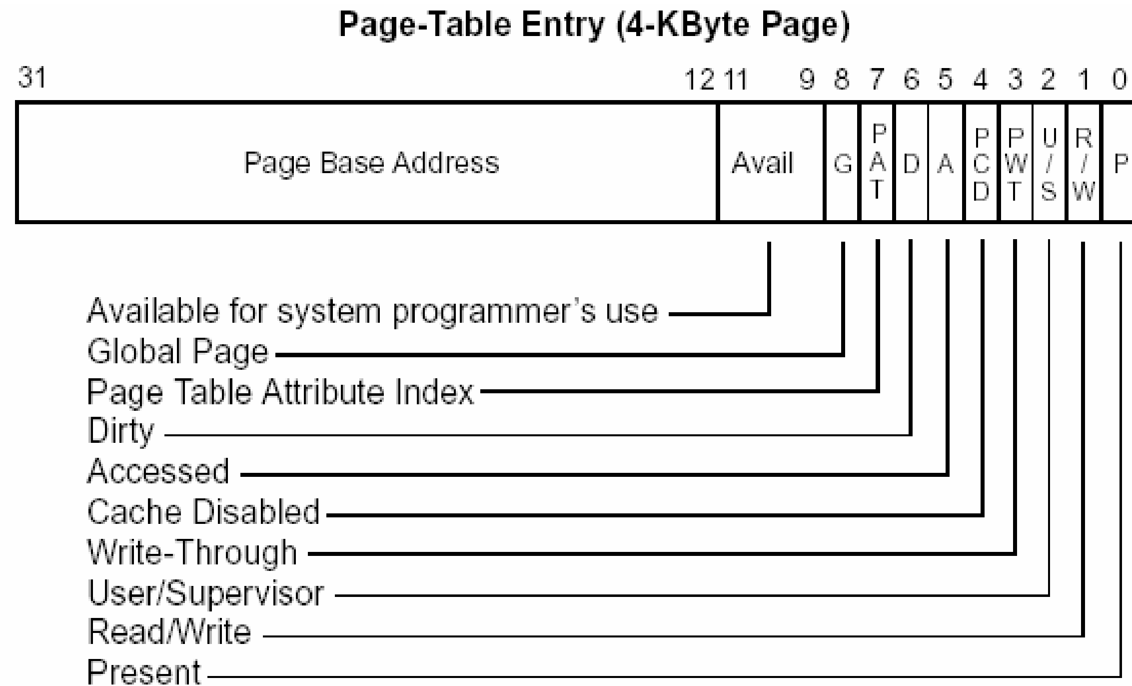
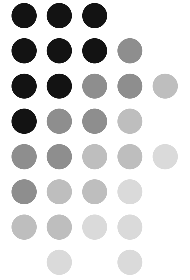
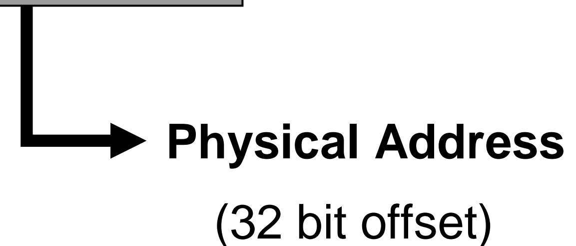


Figure from page 87 of intel-sys.pdf

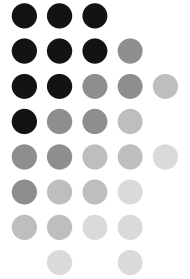
Mundane Details in x86: The Life of a Memory Access



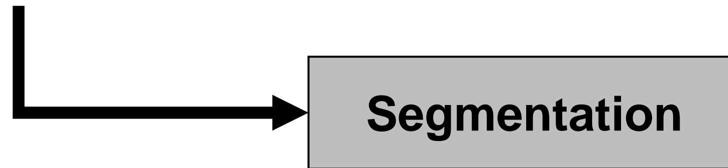
Logical Address (consists of 16 bit segment selector, 32 bit offset)



Mundane Details in x86: The Life of a Memory Access



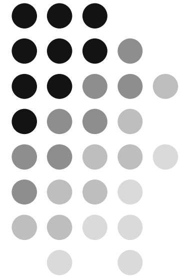
Logical Address (consists of 16 bit segment selector, 32 bit offset)



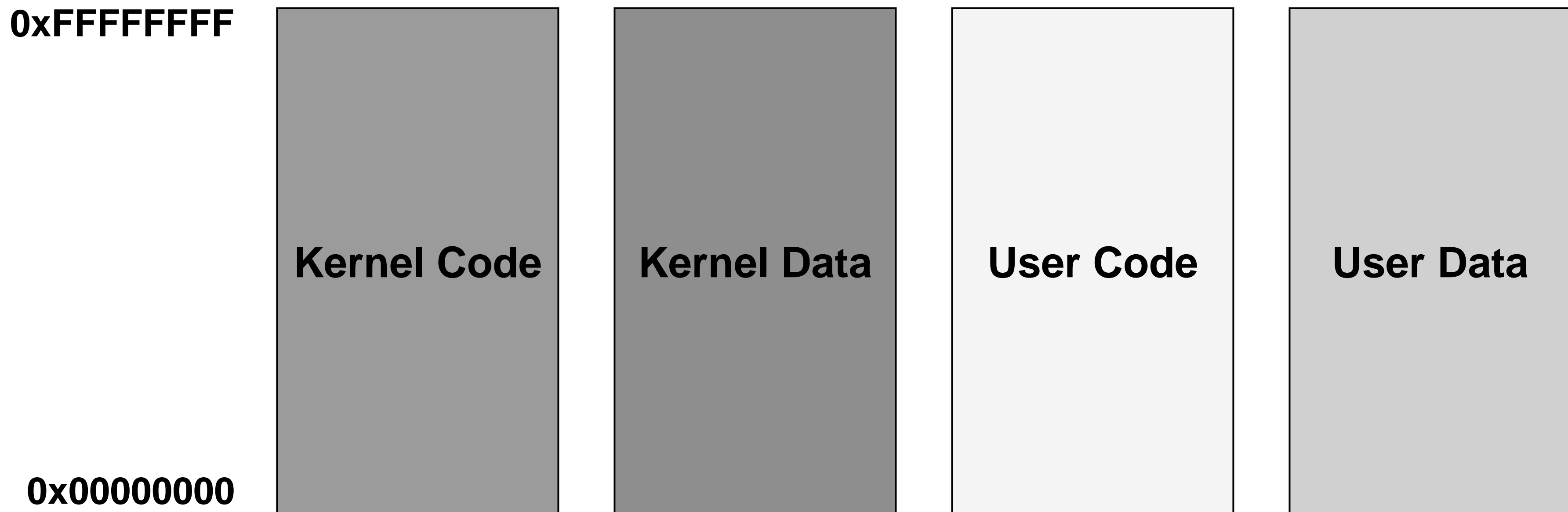
Linear Address (32 bit offset)

- The 16 bit segment selector comes from a segment register
- 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

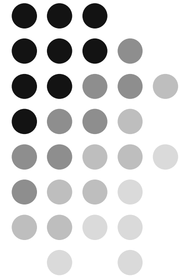
Mundane Details in x86: Segmentation

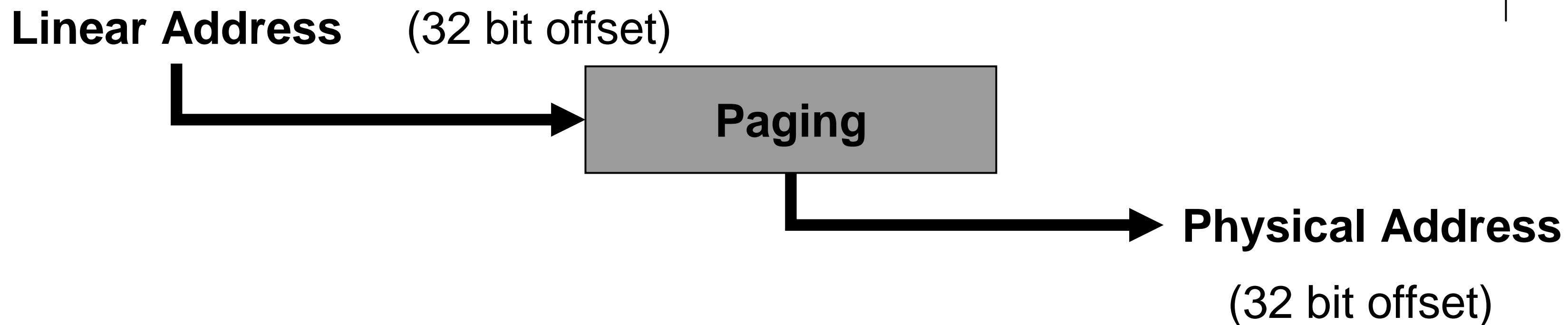


- Segments need not be backed by physical memory and can overlap
- Segments defined for these projects:



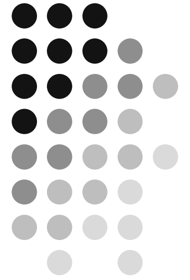
Mundane Details in x86: The Life of a Memory Access





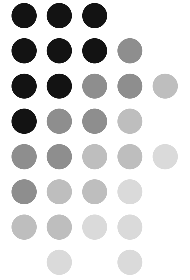
- Top 10 bits index into page directory, point us to a page table
- The next 10 bits index into page table, point us to a page
- The last 12 bits are an offset into that page

Mundane Details in x86: The Life of a Memory Access

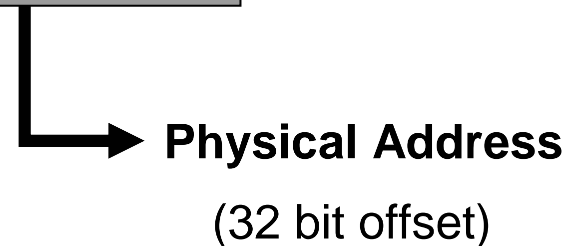


- Whoa there slick… what if the page directory entry isn’t there?
- What happens if the page table entry isn’t there?
- It’s called a page fault, it’s an exception, and it lives in IDT entry 13
- You will have to write a handler for this exception and do something intelligent

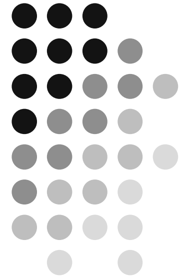
Mundane Details in x86: The Life of a Memory Access



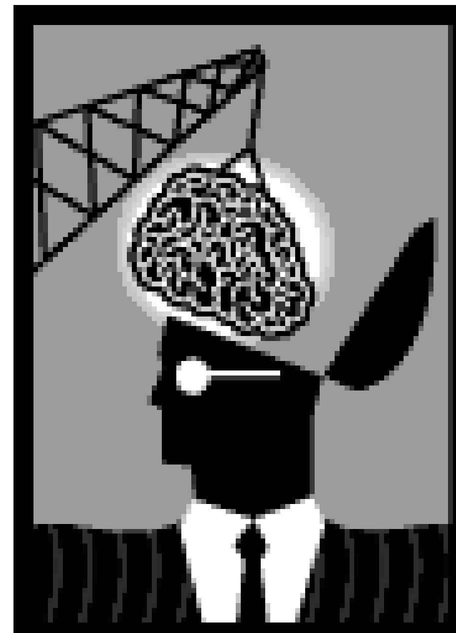
Logical Address (consists of 16 bit segment selector, 32 bit offset)



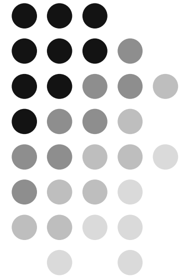
Mundane Details in x86: Context Switching



- We all know that processes take turns running on the CPU
- This means they have to be stopped and started over and over
- How does this occur?

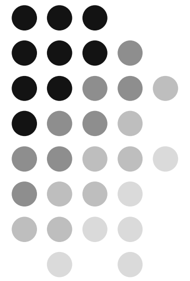


Mundane Details in x86: Context Switching



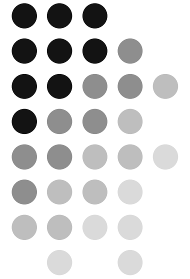
- The x86 architecture provides a hardware mechanism for “tasks”
- This makes context switching easy
- It is actually faster to manage processes in software
- We can also tailor our process abstraction to our particular needs
- You must have at least one hardware task defined, OSKit takes care of this for you

Mundane Details in x86: Context Switching



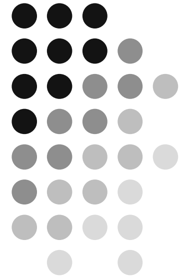
- Context switching is a very delicate procedure
- Great care must be taken so that when the process is started, it does not know it ever stopped
- Registers must be exactly the same (%cr3 is the only control register you have to update)
- It's stack must be exactly the same
- It's page directory must be in place

Mundane Details in x86: Context Switching



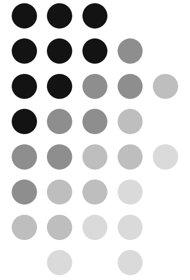
- Hints on context switching:
  - use the stack, it is a convenient place to store things
  - if you do all your switching in one location, you have eliminated one thing you have to save (%eip)
  - new processes will require some special care

Mundane Details in x86: System Calls

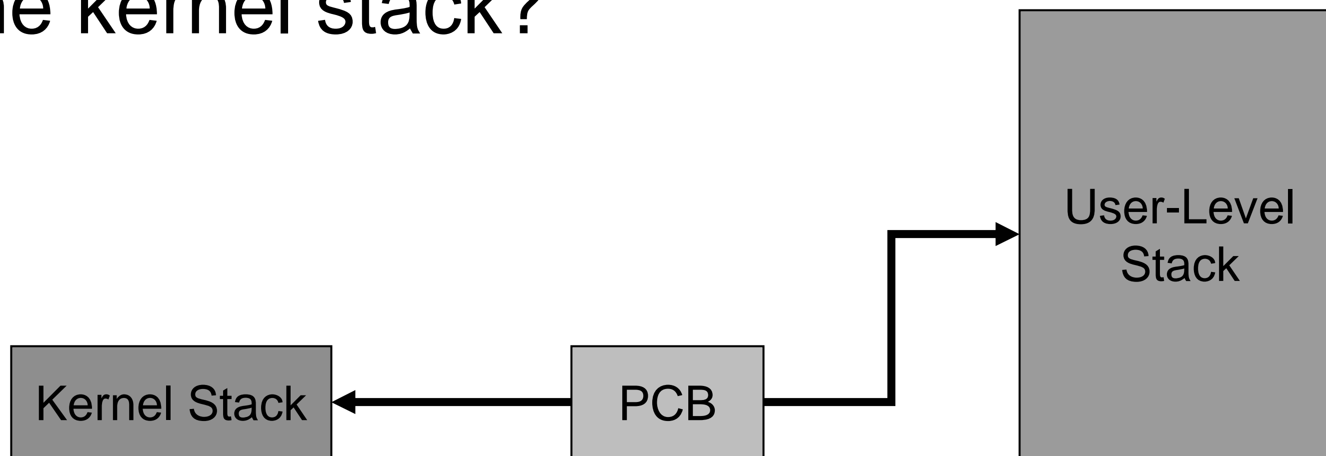


- System calls use software interrupts
- Install a handler just as you did for the timer, keyboard
- Use one software interrupt to implement all of your system calls
- If you are rusty on the IDT refer back to P1

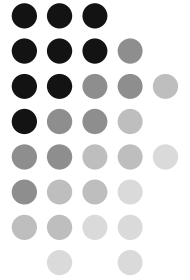
Mundane Details in x86: Kernel Stacks



- User processes should have a separate stack for their kernel activities
- It should be located in kernel space
- How does the stack pointer get switched to the kernel stack?

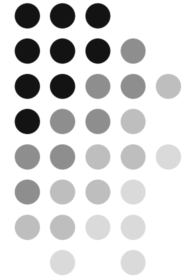


Mundane Details in x86: Kernel Stacks

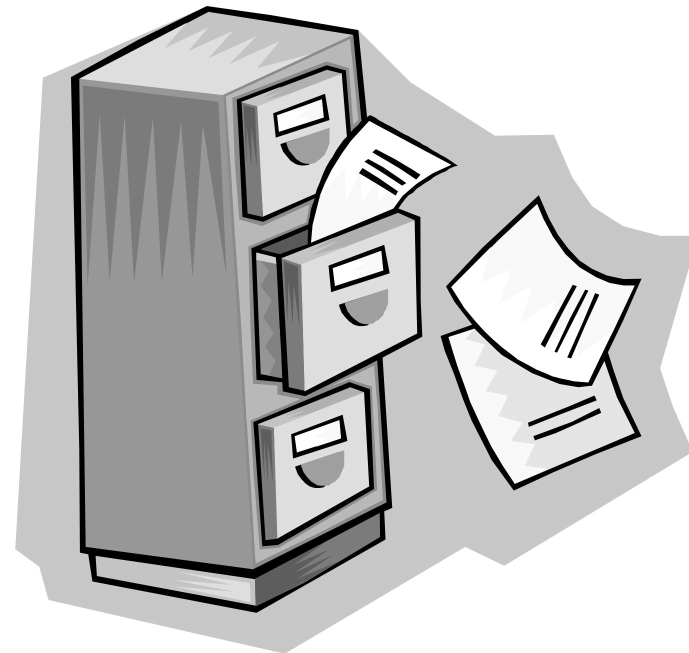


- When the CPU switches from user mode to kernel mode the stack pointer is changed
- The new stack pointer is stored in the configuration of the CPU hardware task
- We provide a function to change this value set_esp0(void* ptr)

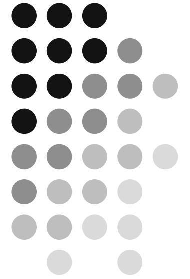
Loading Executables



- You are probably expecting a file system
- But… you have not written one yet
- We have cooked up a small utility to help you

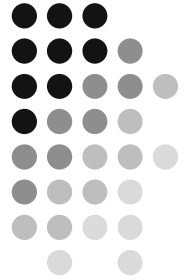


Loading Executables: exec2obj



- Takes a file as input
- Spits out a .c file containing a char array initialized to the contents of the input file
- You can compile this into your kernel

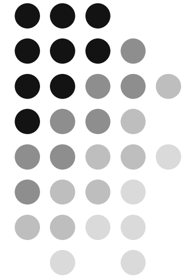
Loading Executables: The Loader



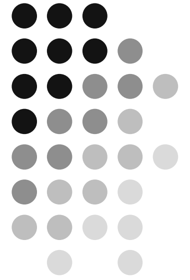
- You have access to the bytes
- You need to load them into the process' address space
- Guess what… you get to write a loader!

- Don't worry, it's not hard
- The executables will be in NMAGIC a.out format
- References to resources are in the handout

A Quick Debug Story

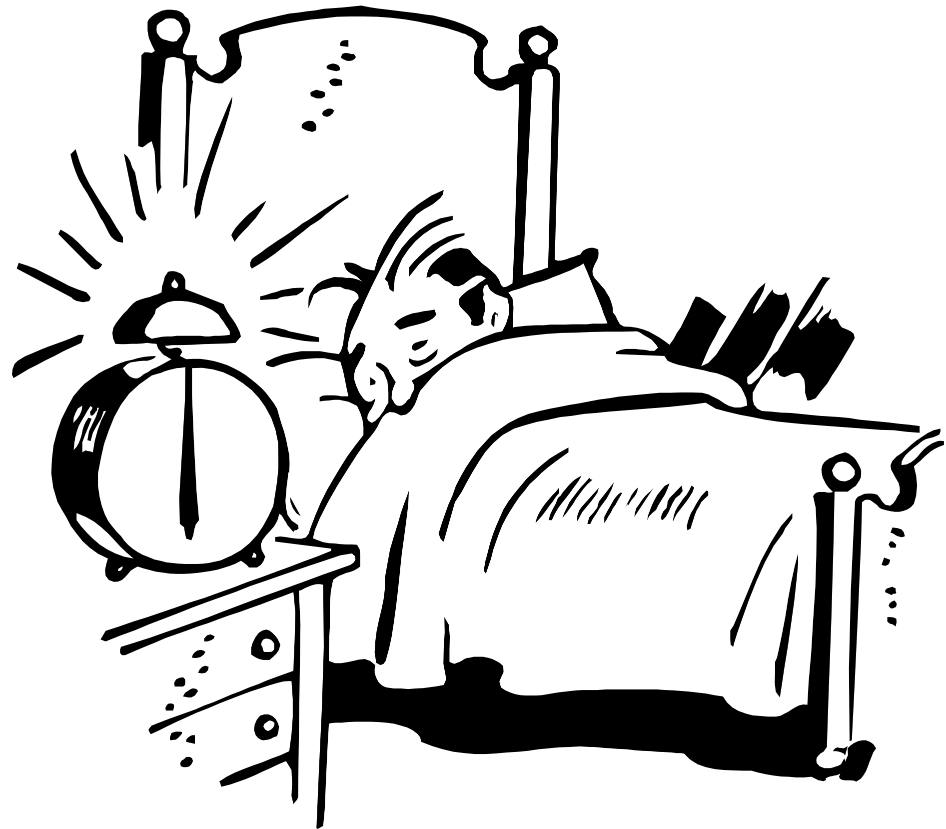


- Ha! You'll have to have been to lecture to hear this story.

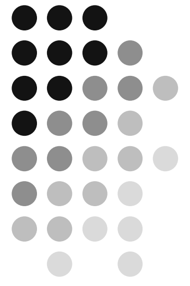


A Quick Debug Story

- The moral is, please start early.

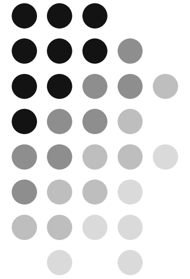


Style Recommendations



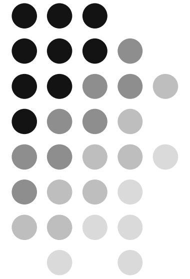
- Do not use a global when a local would do.
- Comment where comments are needed
  - not "comment everywhere"
  - not "do not comment"
- Do NOT hand in your project in one file called kernel.c
  - Do not hand in your project in one file called anything, actually
  - Dave might bite your kneecaps

Attack Strategy

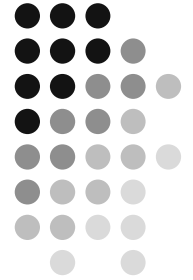


- There is an attack strategy in the handout
- It represents where we think you should be in each of the four weeks
- You WILL have to turn in checkpoint two

Attack Strategy



- Please read the handout a couple times over the next few days
- Then start writing pseudocode!



Good Luck on Project 2!