CS 410/510

Languages & Low-Level Programming

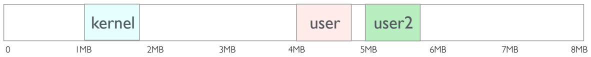
Mark P Jones
Portland State University

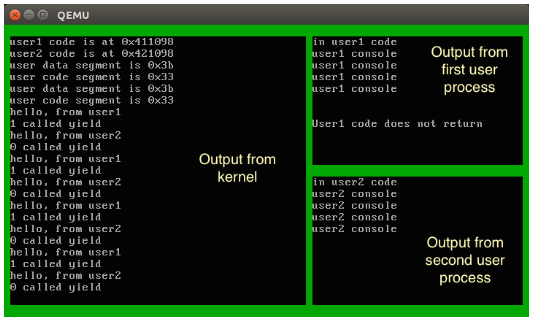
Spring 2016

Week 4: Memory Management

Loose Ends

The Week 3 Lab: Context Switching





Output from kernel

```
user1 code is at 0x411098
user2 code is at 0x421098
user data segment is 0x3b
user code segment is 0x33
user data segment is 0x3b
user code segment is 0x33
hello, from user1
1 called yield
hello, from user2
0 called yield
hello, from user1
1 called yield
hello, from user2
0 called yield
hello, from user1
1 called yield
hello, from user2
0 called yield
hello, from user1
1 called yield
hello, from user2
0 called yield
```

Output from first user process

```
in user1 code
user1 console
user1 console
user1 console
user1 console

User1 code does not return
```

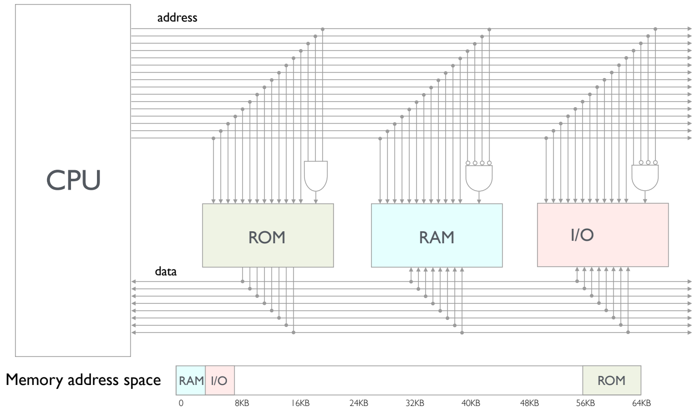
Output from second user process

```
in user2 code
user2 console
user2 console
user2 console
user2 console
```

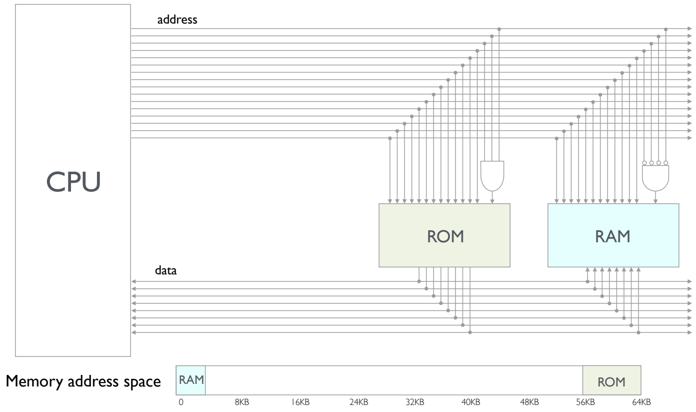
timer interrupt

Port I/O

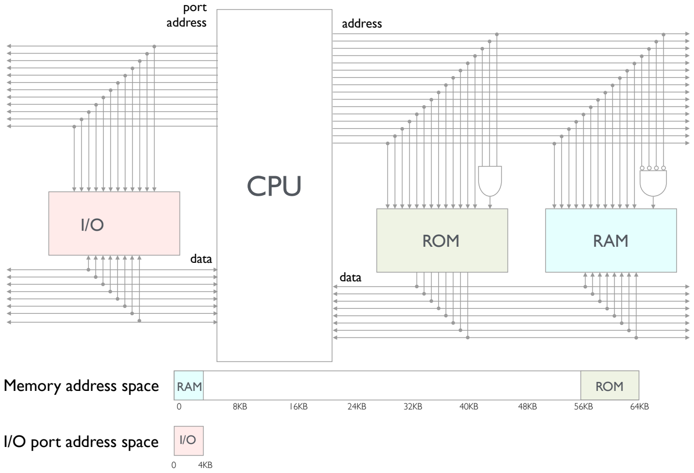
Memory mapped I/O



Memory mapped I/O



Port I/O



Port I/O in the IA32 instruction set

- The IA32 has a 16 bit I/O Port address space
- The hardware actually uses the same address bus and data bus with a signal to indicate whether memory or port access is being used
- You can write a byte/short/word to an I/O port using:

  `out[b|w|l] [%al,%ax,%eax], [imm8|%dx]`

  (use `imm8` for 8 bit port numbers, otherwise use `%dx`)
- You can read a byte/short/word from an I/O port using:

  `in[b|w|l] [imm8|%dx], [%al,%ax,%eax]`

Port I/O using gcc inline assembly

```
static inline void outb(short port, byte b) {
  asm volatile("outb  %1, %0\n" : : "dN"(port), "a"(b));
}


static inline byte inb(short port) {
  unsigned char b;
  asm volatile("inb %1, %0\n" : "=a"(b) : "dN"(port));
  return b;
}
```

- Arcane syntax, general form:

  `asm ( template : output operands : input operands : clobbered registers );`
- Operand constraints include:
  - "`d`" (use `%edx`), "`a`" (use `%eax`), "`N`" (imm8 constant), "`=`" (write only), "`r`" (register), …

The role of inline assembly

- We can already call assembly code from C and vice versa by following calling conventions like the System V ABI
- Inline assembly allows for even tighter integration between C and assembly code: code can be inlined, can have an impact on register allocation, etc…
- But there is essentially no checking of the arguments: it's up to the programmer to specify the correct list of clobbered registers to ensure correct semantics
- Programmers might want to check the generated code …
- How can a general language provide access to essential machine specific instructions and registers?

Standard port numbers on the PC platform

Port Range	Device
0x00-0x1f	First DMA controller (8237)
0x20-0x3f	Master Programmable Interrupt Controller (PIC) (8259A)
0x40-0x5f	Programmable Interval Timer (PIT) (8253/8254)
0x60-0x6f	Keyboard (8042)
0x70-0x7f	Real Time Clock (RTC)
0x80-0x9f	DMA ports, Refresh
0xa0-0xbf	Second Programmable Interrupt Controller (PIC) (8259A)
0xc0-0xdf	Second DMA controller (8237)
…	…
0x3f0-0x3f7	Primary floppy disk drive controller
0x3f8-0x3ff	Serial Port 1
…	…

Serial port output in assembly

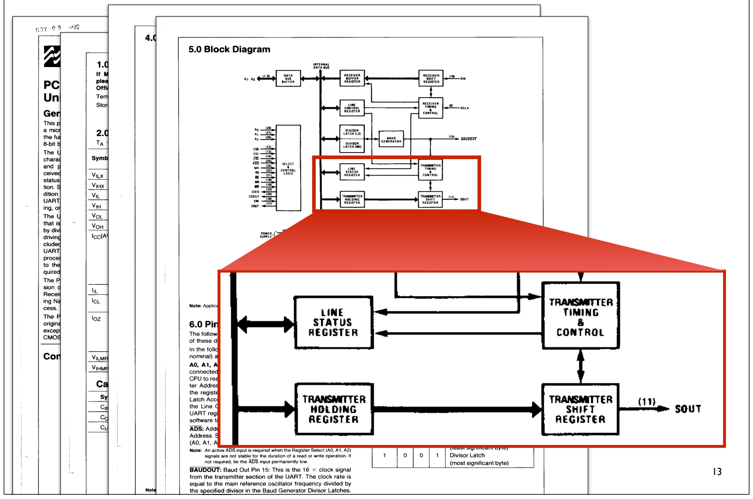
```
        .set    PORTCOM1, 0x3f8
serial_putc:
        pushl   %eax
        pushl   %edx

        movw    $(PORTCOM1+5), %dx
1:      inb     %dx, %al        # Wait for port to be ready
        andb    $0x60, %al
        jz      1b
        movw    $PORTCOM1, %dx  # Output the character
        movb    12(%esp), %al
        outb    %al, %dx

        cmpb    $0xa, %al       # Was it a newline?
        jnz     2f

        movw    $(PORTCOM1+5), %dx
1:      inb     %dx, %al        # Wait again for port to be ready
        andb    $0x60, %al
        jz      1b
        movw    $PORTCOM1, %dx  # Send a carriage return
        movb    $0xd, %al
        outb    %al, %dx

2:      popl    %edx
        popl    %eax
        ret
```

PC platform

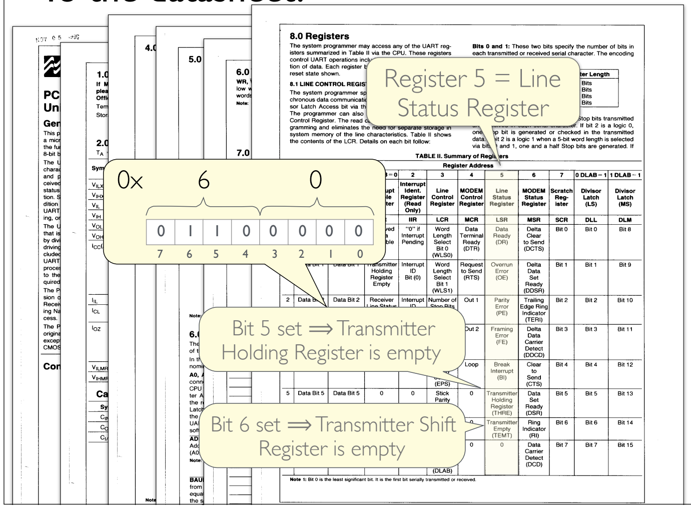
why +5?

why 0x60?

To the datasheet!


5.0 Block Diagram
LINE STATUS REGISTER
TRANSMITTER TIMING & CONTROL
TRANSMITTER HOLDING REGISTER
TRANSMITTER SHIFT REGISTER
(11)
SOUT

To the datasheet!



Register 5 = Line Status Register

Bit 5 set ⟹ Transmitter Holding Register is empty

Bit 6 set ⟹ Transmitter Shift Register is empty

Serial port output in assembly

```
        .set    PORTCOM1, 0x3f8
serial_putc:
        pushl   %eax
        pushl   %edx

        movw    $(PORTCOM1+5), %dx
1:      inb     %dx, %al         # Wait for port to be ready
        andb    $0x60, %al
        jz      1b
        movw    $PORTCOM1, %dx   # Output the character
        movb    12(%esp), %al
        outb    %al, %dx

        cmpb    $0xa, %al        # Was it a newline?
        jnz     2f

        movw    $(PORTCOM1+5), %dx
1:      inb     %dx, %al         # Wait again for port to be ready
        andb    $0x60, %al
        jz      1b
        movw    $PORTCOM1, %dx   # Send a carriage return
        movb    $0xd, %al
        outb    %al, %dx

2:      popl    %edx
        popl    %eax
        ret
```

Read the line status register

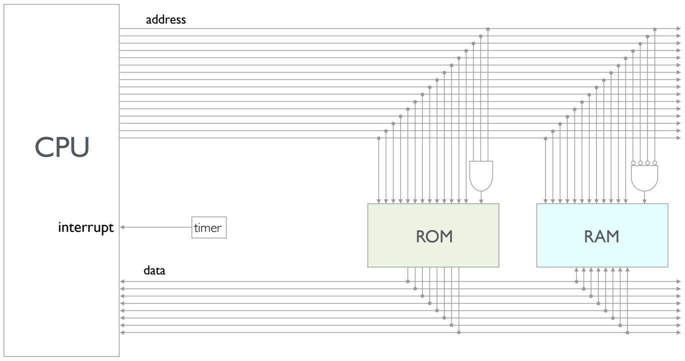
check for available transmitter register

Reading datasheets

- Datasheets present detailed technical information in a very terse format
- Unless you are already familiar with the details, and just looking for a reference, it can be hard to find the information you need
- But persevere, and practice; this can be a useful skill
- One thing you'll often see is that computer systems typically only use a fraction of the available functionality(/transistors)
- Sample code, from the manufacturers, or on the web, can also be very useful!

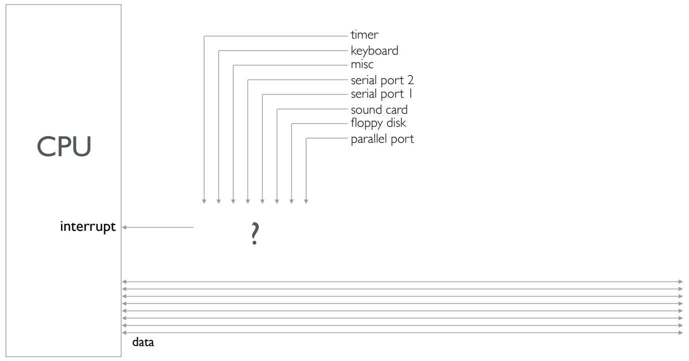
Interrupts

Hardware interrupts



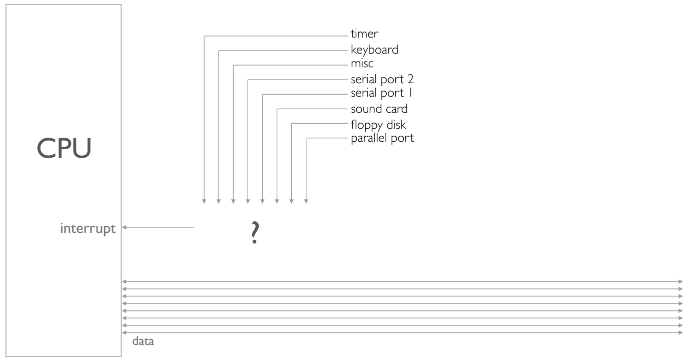
- The CPU has an interrupt pin
- Connect it to a timer to generate regular timer interrupts!

How to handle multiple interrupt sources?



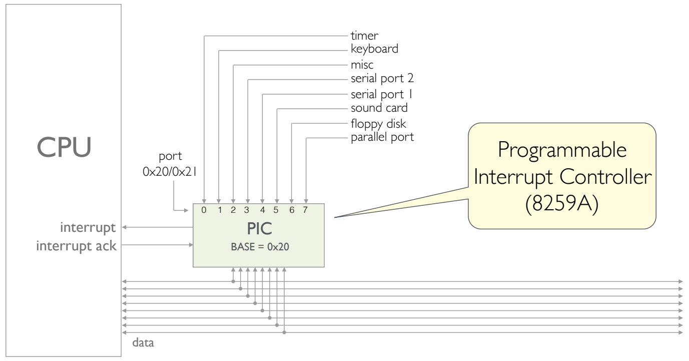
- How do we combine multiple interrupt signals?
- How do we identify and prioritize interrupt sources?

How to handle multiple interrupt sources?



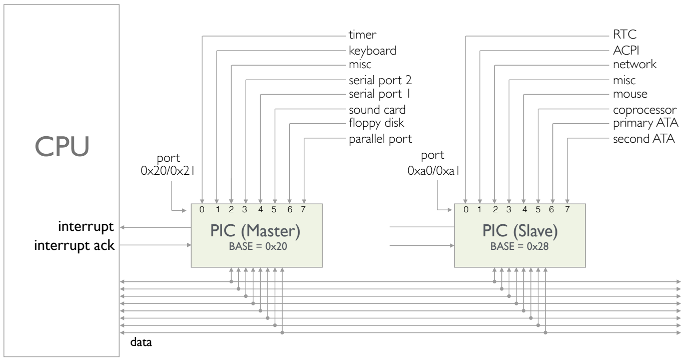
- One option: use an "or" to combine the interrupt signals
- Use the CPU to "poll" to determine which interrupt fired …

Adding an interrupt controller



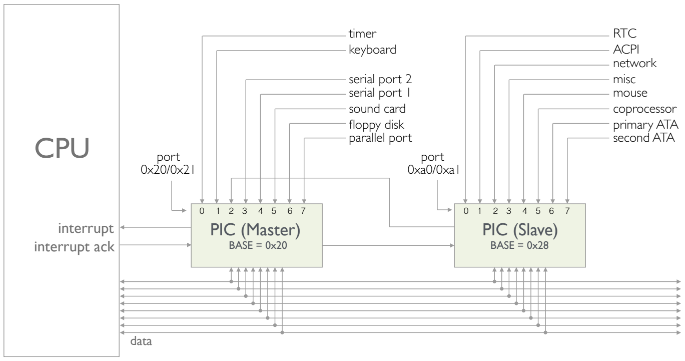
- The PIC allows individual interrupts to be masked/unmasked
- Responds to ack with programmed BASE + IRQ (interrupt request number) on data bus

Adding multiple interrupt controllers



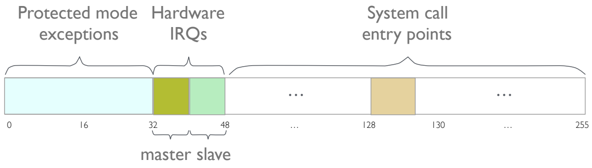
- Two PICs in master/slave configuration

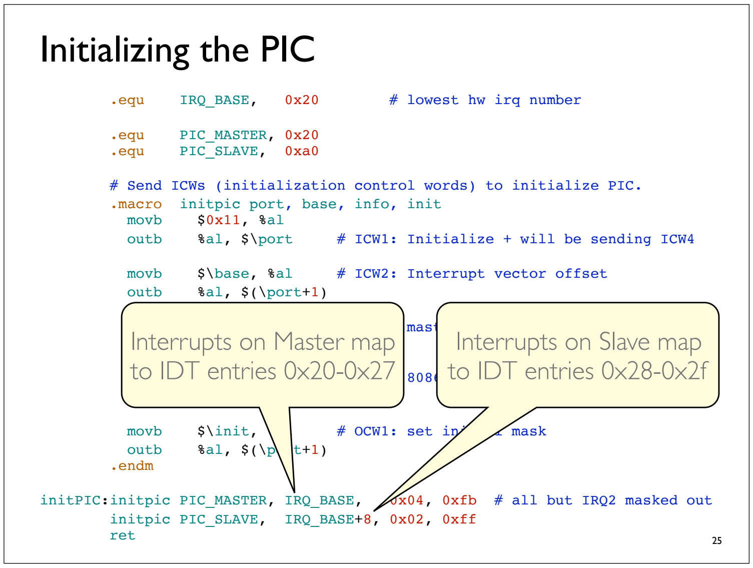
Adding multiple interrupt controllers



- Two PICs in master/slave configuration
- Any interrupt on the Slave triggers interrupt 2 on the Master

IDT structure



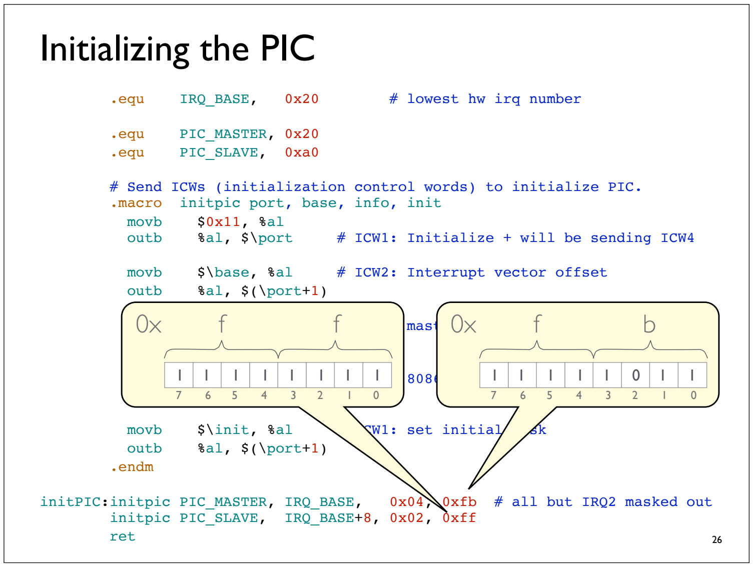


Initializing the PIC

```
        .equ    IRQ_BASE,   0x20        # lowest hw irq number

        .equ    PIC_MASTER, 0x20
        .equ    PIC_SLAVE,  0xa0

        # Send ICWs (initialization control words) to initialize PIC.
        .macro  initpic port, base, info, init
          movb    $0x11, %al
          outb    %al, $\port      # ICW1: Initialize + will be sending ICW4

          movb    $\base, %al      # ICW2: Interrupt vector offset
          outb    %al, $(\port+1)

                                         mast

                                         808

          movb    $\init,          # OCW1: set in       mask
          outb    %al, $(\p   t+1)
        .endm

initPIC:initpic PIC_MASTER, IRQ_BASE,    0x04, 0xfb  # all but IRQ2 masked out
        initpic PIC_SLAVE,  IRQ_BASE+8, 0x02, 0xff
        ret
```

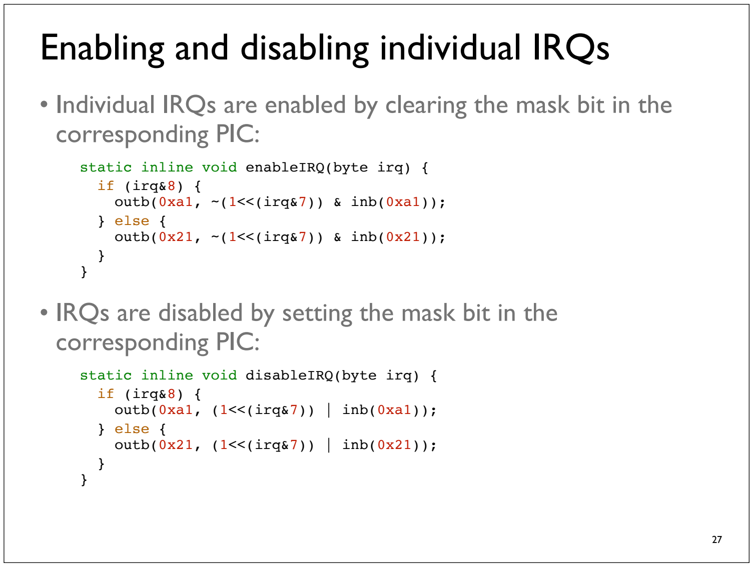
Interrupts on Master map to IDT entries 0x20-0x27

Interrupts on Slave map to IDT entries 0x28-0x2f



Initializing the PIC

```
        .equ    IRQ_BASE,   0x20        # lowest hw irq number

        .equ    PIC_MASTER, 0x20
        .equ    PIC_SLAVE,  0xa0

        # Send ICWs (initialization control words) to initialize PIC.
        .macro  initpic port, base, info, init
          movb    $0x11, %al
          outb    %al, $\port      # ICW1: Initialize + will be sending ICW4

          movb    $\base, %al      # ICW2: Interrupt vector offset
          outb    %al, $(\port+1)

                                         mast

                                         808

          movb    $\init, %al       CW1: set initial    sk
          outb    %al, $(\port+1)
        .endm

initPIC:initpic PIC_MASTER, IRQ_BASE,    0x04, 0xfb  # all but IRQ2 masked out
        initpic PIC_SLAVE,  IRQ_BASE+8, 0x02, 0xff
        ret
```



Enabling and disabling individual IRQs

- Individual IRQs are enabled by clearing the mask bit in the corresponding PIC:

```
static inline void enableIRQ(byte irq) {
  if (irq&8) {
    outb(0xa1, ~(1<<(irq&7)) & inb(0xa1));
  } else {
    outb(0x21, ~(1<<(irq&7)) & inb(0x21));
  }
}
```

- IRQs are disabled by setting the mask bit in the corresponding PIC:

```
static inline void disableIRQ(byte irq) {
  if (irq&8) {
    outb(0xa1, (1<<(irq&7)) | inb(0xa1));
  } else {
    outb(0x21, (1<<(irq&7)) | inb(0x21));
  }
}
```

IRQ handling lifecycle

- Install handler for IRQ in IDT
- Use the PIC to enable that specific IRQ (the CPU will still ignore the interrupt if the IF flag is clear)
- If the interrupt is triggered, disable the IRQ and send an EOI (end of interrupt) to reenable the PIC for other IRQs:

```
static inline void maskAckIRQ(byte irq) {
  if (irq&8) {
    outb(0xa1, (1<<(irq&7)) | inb(0xa1));
    outb(0xa0, 0x60|(irq&7)); // EOI to slave
    outb(0x20, 0x62);         // EOI for IRQ2 on master
  } else {
    outb(0x21, (1<<(irq&7)) | inb(0x21));
    outb(0x20, 0x60|(irq&7)); // EOI to master
  }
}
```

- When the interrupt has been handled, reenable the IRQ

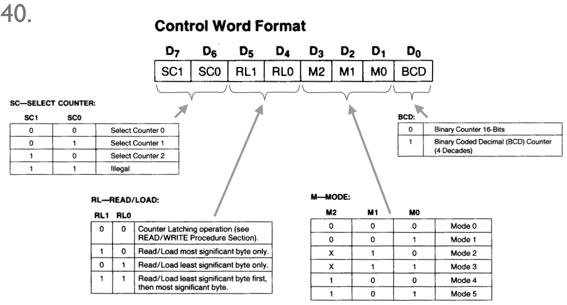
Timers

The programmable interval timer (PIT)

- The IBM PC included an Intel 8253/54 programmable interval timer (PIT) chip
- The PIT was clocked at 1,193,181.8181Hz, for compatibility with the NTSC TV standard
- The PIT provides three counter/timers. On the PC, these were used to handle:
  - Counter 0: Timer interrupts
  - Counter 1: DRAM refresh
  - Counter 2: Playing tones via the PC's speaker

… continued

- The PIT is programmed by sending a control word to port 0x43 followed by a two byte counter value (lsb first) to port 0x40.



Control Word Format

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	RL1	RL0	M2	M1	M0	BCD

SC—SELECT COUNTER:

SC1	SC0	
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Illegal

BCD:

0	Binary Counter 16-Bits
1	Binary Coded Decimal (BCD) Counter (4 Decades)

RL—READ/LOAD:

RL1	RL0	
0	0	Counter Latching operation (see READ/WRITE Procedure Section).
1	0	Read/Load most significant byte only.
0	1	Read/Load least significant byte only.
1	1	Read/Load least significant byte first, then most significant byte.

M—MODE:

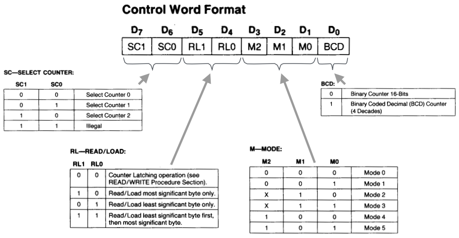
M2	M1	M0	
0	0	0	Mode 0
0	0	1	Mode 1
X	1	0	Mode 2
X	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

- Each timer/counter runs in one of six modes.

Example: Programming the PIT

To configure for timer interrupts:

```
#define HZ           100           // Frequency of timer interrupts
#define PIT_INTERVAL  ((1193182 + (HZ/2)) / HZ)
#define TIMERIRQ      0

static inline void startTimer() {
  outb(0x43, 0x34);  // PIT control (0x43), counter 0, 2 bytes, mode 2, binary
  outb(0x40, PIT_INTERVAL        & 0xff);  // counter 0, lsb
  outb(0x40, (PIT_INTERVAL >> 8) & 0xff);  // counter 0, msb
  enableIRQ(TIMERIRQ);
}
```



Control Word Format

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	RL1	RL0	M2	M1	M0	BCD

SC—SELECT COUNTER:

SC1	SC0	
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Illegal

BCD:

0	Binary Counter 16-Bits
1	Binary Coded Decimal (BCD) Counter (4 Decades)

RL—READ/LOAD:

RL1	RL0	
0	0	Counter Latching operation (see READ/WRITE Procedure Section).
1	0	Read/Load most significant byte only.
0	1	Read/Load least significant byte only.
1	1	Read/Load least significant byte first, then most significant byte.

M—MODE:

M2	M1	M0	
0	0	0	Mode 0
0	0	1	Mode 1
X	1	0	Mode 2
X	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

Time stamp counter

- Modern Intel CPUs include a 64 bit time stamp counter that tracks the number of cycles since reset
- The current TSC value can be read in `edx:eax` using the `rdtsc` instruction
- `rdtsc` is privileged, but the CPU can be configured to allow access to `rdtsc` in user level code
- Can use differences in TSC value before and after an event to measure elapsed time
- But beware of complications related to multiprocessor systems; power management (e.g., variable clock speed); …
- … and virtualization …. (e.g., QEMU, VirtualBox, …)

Volatile Memory

The first user program

user

```
unsigned flag = 0;

for (i=0; i<600; i++) {
  ...
}
printf("My flag is at 0x%x\n", &flag);
while (flag==0) {
   /* do nothing */
}
printf("Somebody set my flag to %d!\n", flag);
...
```

"My flag is at 0x4025b0"

- According to the semantics of C, there is no way for the value of the variable flag to change during the while loop …
- … so there is no way that the "Somebody set my flag …" message could appear
- … the compiler could delete the code after the while loop …

The second user program

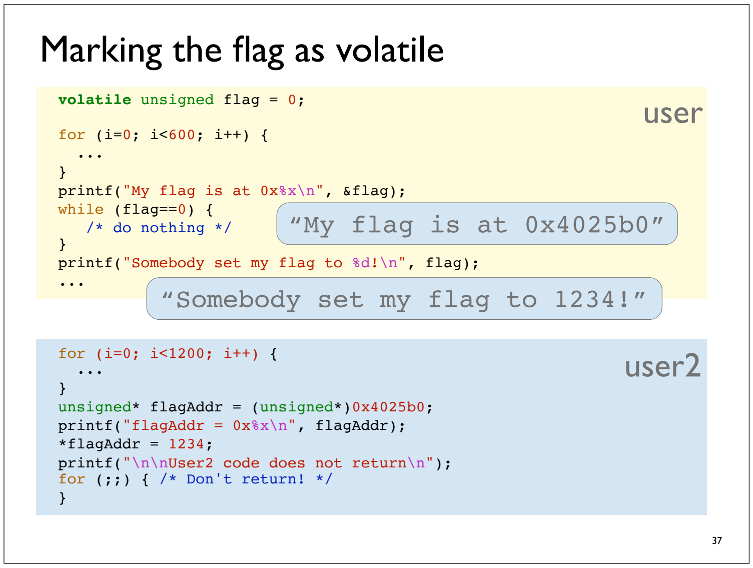
user

```
unsigned flag = 0;

for (i=0; i<600; i++) {
  ...
}
printf("My flag is at 0x%x\n", &flag);
while (flag==0) {
   /* do nothing */
}
printf("Somebody set my flag to %d!\n", flag);
...
```

"My flag is at 0x4025b0"

user2

```
for (i=0; i<1200; i++) {
  ...
}
unsigned* flagAddr = (unsigned*)0x4025b0;
printf("flagAddr = 0x%x\n", flagAddr);
*flagAddr = 1234;
printf("\n\nUser2 code does not return\n");
for (;;) { /* Don't return! */
}
```



Marking the flag as volatile

user

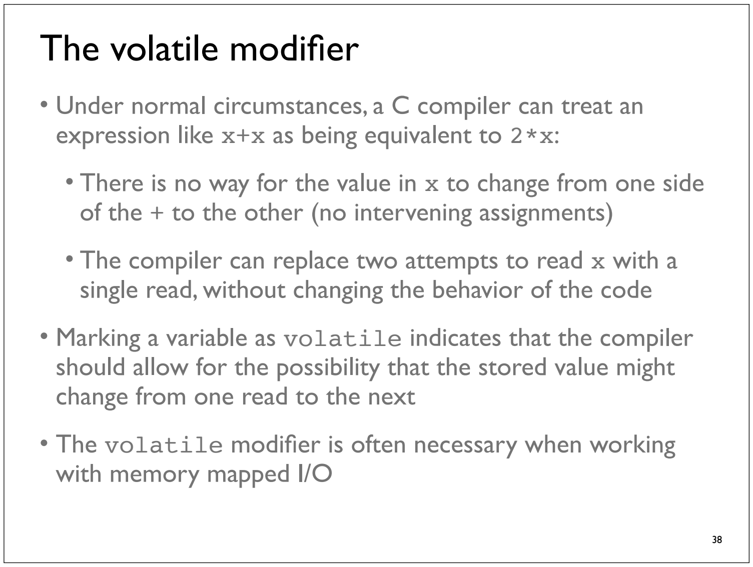
```
volatile unsigned flag = 0;

for (i=0; i<600; i++) {
  ...
}
printf("My flag is at 0x%x\n", &flag);
while (flag==0) {
   /* do nothing */
}
printf("Somebody set my flag to %d!\n", flag);
...
```

“My flag is at 0x4025b0”

“Somebody set my flag to 1234!”

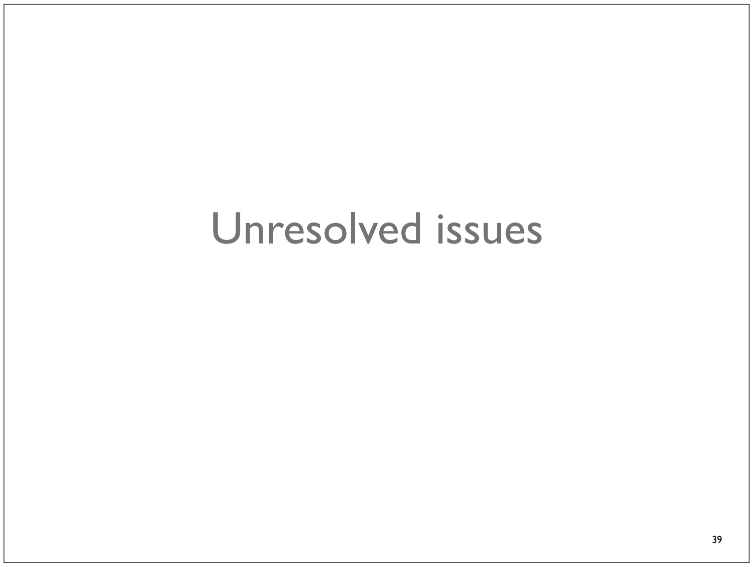
user2

```
for (i=0; i<1200; i++) {
  ...
}
unsigned* flagAddr = (unsigned*)0x4025b0;
printf("flagAddr = 0x%x\n", flagAddr);
*flagAddr = 1234;
printf("\n\nUser2 code does not return\n");
for (;;) { /* Don't return! */
}
```



The volatile modifier

- Under normal circumstances, a C compiler can treat an expression like `x+x` as being equivalent to `2*x`:
  - There is no way for the value in `x` to change from one side of the + to the other (no intervening assignments)
  - The compiler can replace two attempts to read `x` with a single read, without changing the behavior of the code
- Marking a variable as `volatile` indicates that the compiler should allow for the possibility that the stored value might change from one read to the next
- The `volatile` modifier is often necessary when working with memory mapped I/O



Unresolved issues

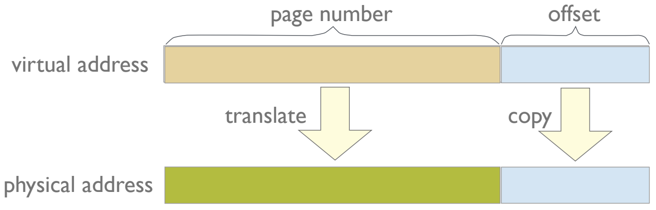
Issues with the Week 3 lab example

- Although we are running in protected mode, we are using segments that span the full address space, so there is **no true protection** between the different programs
- Address space layout is ad hoc: different programs load and run at different addresses; there is no consistency
- We had to choose different (but essentially arbitrary) start addresses for user and user2, even when they were just two copies of the same program
- Why should worries about low level memory layout & size propagate in to the design of higher-level applications?
- Our user programs included duplicate code (e.g., each one has its own implementation of printf). How can we support sharing of common code or data between multiple programs?

Paging

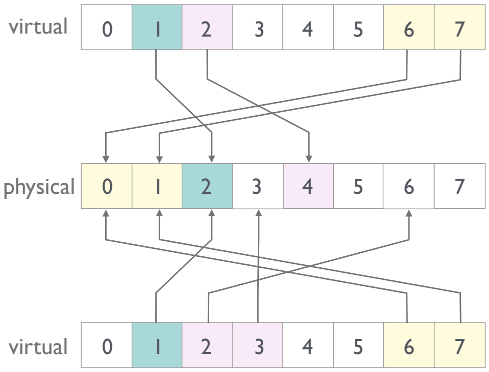
Paging

- "All problems in computer science can be solved by another level of indirection" (David Wheeler)
- Partition the address space in to a collection of "pages"
- Translate between addresses in some idealized "virtual address space" and "physical addresses" to memory.



Example

- Suppose that we partition our memory into 8 pages:

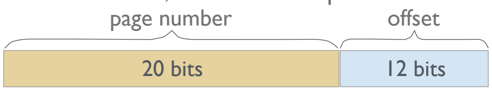


Virt	Phys
0	
1	2
2	4
3	
4	
5	
6	0
7	1

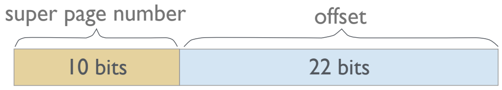
Virt	Phys
0	
1	2
2	6
3	3
4	
5	
6	0
7	1

Practical reality

- IA32 partitions the 32-bit, 4GB address space in to 4KB pages



- It also allows the address space to be viewed as 4MB "super pages"



- We need a table with 2^{10} entries to translate virtual super page numbers in to physical page numbers
- With 4 bytes/entry, this table, called a **page directory**, takes 2^{12} bytes - one 4K page!

Paging with 4MB super pages

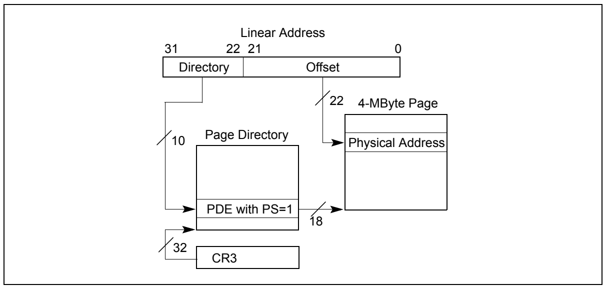


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

- The cr3 register points to the "current" page directory
- Individual page directory entries (PDEs) specify a 10 bit physical super page address plus some additional control bits

Page tables

- A table describing translations for all 4KB pages would require 2^{20} entries
- With four bytes per entry, a full page table would take 4MB
- Most programs are small, at least in comparison to the full address space

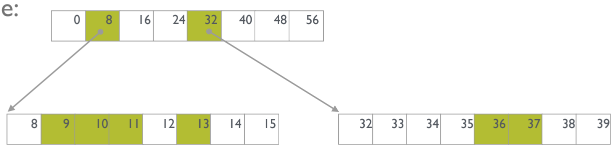
  ⟹ most address spaces are fairly sparse
- is there a more compact way to represent their page tables?

Example

- Suppose that our memory is partitioned in to 64 pages
- But we are only use a small number of those pages…
- … in fact, only a small number of the rows

0	1	2	3	4	5	6	7
8	**9**	**10**	**11**	12	**13**	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	**36**	**37**	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

- Then we can represent the full table more compactly as a tree:



0 **8** 16 24 **32** 40 48 56

8 **9** **10** **11** 12 **13** 14 15

32 33 34 35 **36** **37** 38 39

Paging with 4KB pages

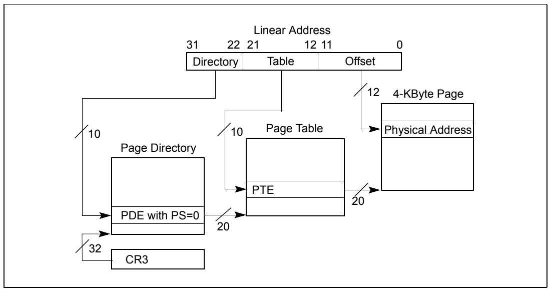
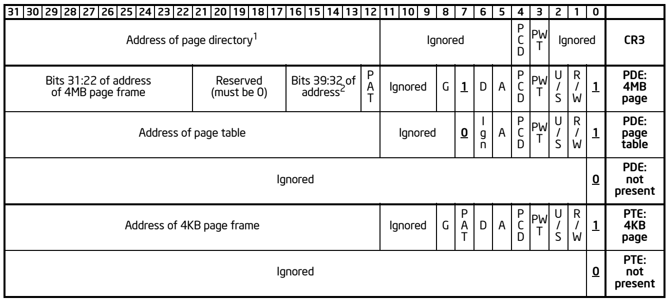


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

- A typical address space can now be described by a page directory plus one or two page tables (i.e., 4-12KB)
- Can mix pages and super pages for more flexibility

CR3, PDEs, PTEs



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			**CR3**
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	**1**	D	A	PCD	PWT	U/S	R/W	**1**	**PDE: 4MB page**
Address of page table																				Ignored				**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**PDE: page table**
Ignored																															**0**	**PDE: not present**
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	**1**	**PTE: 4KB page**
Ignored																															**0**	**PTE: not present**

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Details

- Paging structures use physical addresses
- P(resent) bit 0 is used to mark valid entries (an OS can use the remaining "ignored" fields to store extra information)
- Hardware updates D(irty) and A(ccessed) bits to track usage
- R/W bits allow regions of memory to be marked "read only"
- S/U bits allow regions of memory to be restricted to "supervisor" access only (rather than general "user")
- G(lobal) bit allows pages to be marked as appearing in every address space
- PCD and PWD bits control caching behavior

The translation lookaside buffer (TLB)

- Recall that the IA32 tracks current segment base and limit values in hidden registers to allow for faster access
- A more sophisticated form of cache, called the translation lookaside buffer (TLB), is used to keep track of active mappings within the CPU's memory management unit
- Programmers typically ignore the TLB: "it just works"
- But not so in programs that modify page directories and page tables: extra steps are required to ensure that the TLB is updated to reflect changes in the page table
  - Loading a value in to CR3 will flush the TLB
  - the "`invlpg addr`" instruction removes TLB entries for a specific address

Segmentation and paging

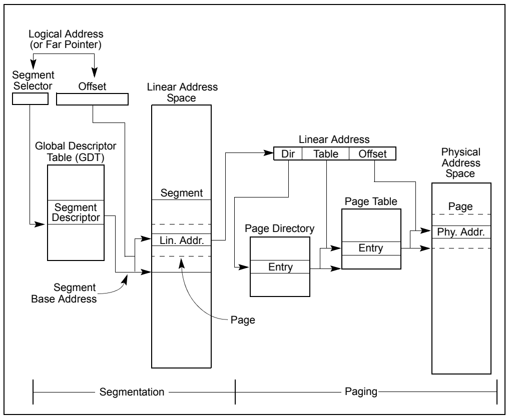
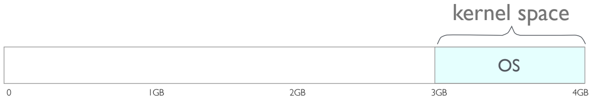


Figure 3-1. Segmentation and Paging

Protection and address space layout

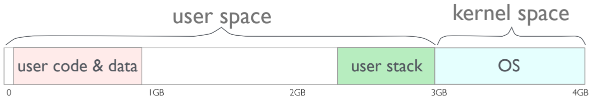
- A typical operating system adopts a virtual memory layout something like the following for all address spaces:



- The operating system is in every address space; it’s pages are protected from user programs by limiting those parts of the page directory to “supervisor” access
- The OS portion of the page directory can take advantage of G(lobal) bits so that TLB entries for kernel space are retained when we switch between address spaces

Protection and address space layout

- A typical operating system adopts a virtual memory layout something like the following for all address spaces:



- User code and data mappings differ from one address space to the next
  - there is no way for one user program to access memory regions for another program …
  - … unless the OS provides the necessary mappings
  - user programs do not have a **capability** to access unauthorized regions of memory

Control registers to enable paging

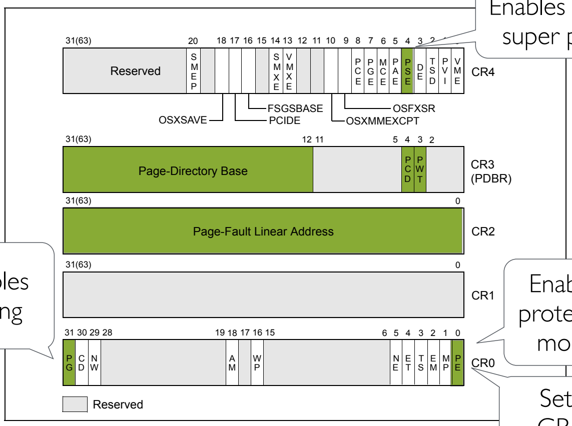


Figure 2-7. Control Registers

Enables use of super pages

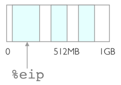
Enables paging

Enables protected mode

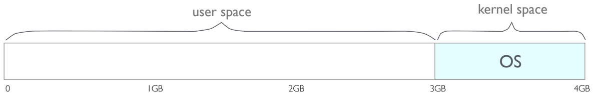
Set by GRUB

Initialization

- How do we get from physical memory, after booting:



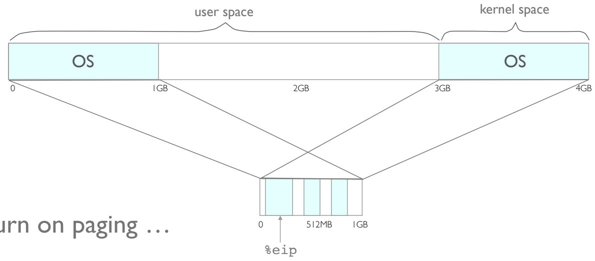
- to virtual address spaces with paging enabled?



- Two key steps
  - Create an initial page directory
  - Enable the CPU paging mechanisms

Creating a 1:1 mapping

- While running at lower addresses, create an initial page directory that maps the lower 1GB of memory in two different regions of the virtual address space



- Turn on paging …
- jump to an address in the upper 1GB of virtual memory …
- and then proceed without the lower mapping …

Working with physical & virtual addresses

- It is convenient to work with page directories and page tables as regular data structures (virtual addresses):

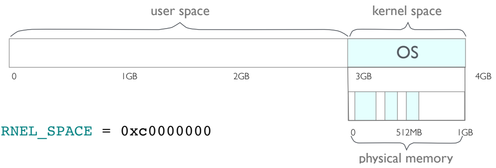
```
struct Pdir { unsigned pde[1024]; };
struct Ptab { unsigned pte[1024]; };

/*-------------------------------------------------------------------------
 * Return a pointer to the page table for the ith entry of the specified
 * pdir, or NULL if it is not present (0x1) or is a super page (0x80).
 */
static inline struct Ptab* getPagetab(struct Pdir* pdir, unsigned i) {
  return ((pdir->pde[i]&0x81)==0x1)
          ? fromPhys(struct Ptab*, align(pdir->pde[i], PAGESIZE)) : 0;
}
```

- But sometimes we have to work with physical addresses:

```
/*-------------------------------------------------------------------------
 * Set the page directory control register to a specific value.
 */
static inline void setPdir(unsigned pdir) {
  asm("  movl  %0, %%cr3\n" : : "r"(pdir));
}
```

From physical to virtual, and back again

- Because we map the top 1GB of virtual memory to the bottom 1GB of physical memory, it is easy to convert between virtual and physical addresses:



```
KERNEL_SPACE = 0xc0000000

#define fromPhys(t, addr) ((t)(addr+KERNEL_SPACE))
#define toPhys(ptr)       ((unsigned)(ptr) - KERNEL_SPACE)
```

- (But how can we do this in a type safe language … ?)

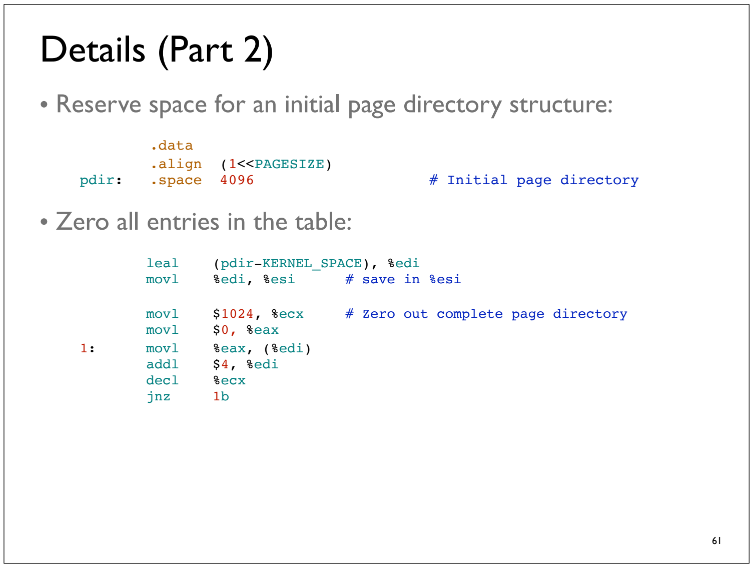
Details (Part 1)

- Constants to describe the virtual address space

```
KERNEL_SPACE    = 0xc0000000      # Kernel space starts at 3GB
KERNEL_LOAD     = 0x00100000      # Kernel loads at 1MB
```

- The kernel is configured to load at a low physical address but run at a high virtual address:

```
OUTPUT_FORMAT(elf32-i386)
ENTRY(physentry)

SECTIONS {

  physentry = entry - KERNEL_SPACE;
  . = KERNEL_LOAD + KERNEL_SPACE;

  .text ALIGN(0x1000) : AT(ADDR(.text) - KERNEL_SPACE) {
    _text_start = .; *(.text) *(.handlers) _text_end = .;
    *(.rodata*)
    *(.data)
    _start_bss = .; *(COMMON) *(.bss) _end_bss = .;
  }
}
```



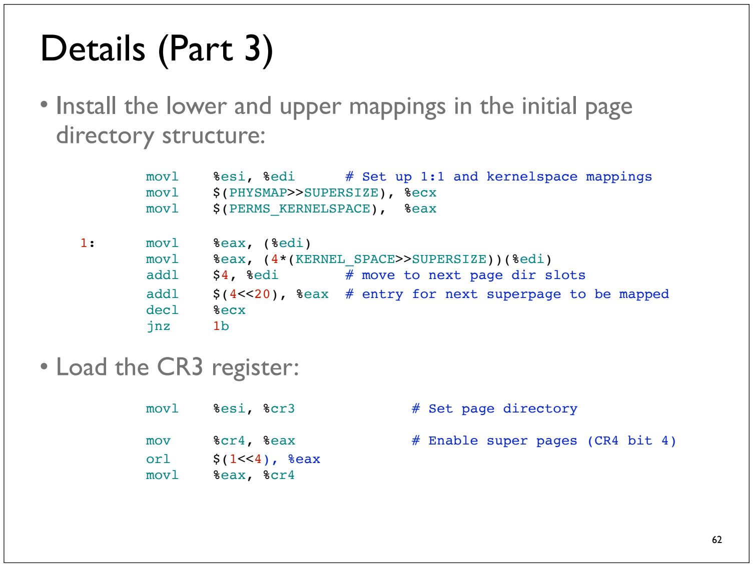
Details (Part 2)

- Reserve space for an initial page directory structure:

```
        .data
        .align  (1<<PAGESIZE)
pdir:   .space  4096                    # Initial page directory
```

- Zero all entries in the table:

```
        leal    (pdir-KERNEL_SPACE), %edi
        movl    %edi, %esi      # save in %esi

        movl    $1024, %ecx     # Zero out complete page directory
        movl    $0, %eax
1:      movl    %eax, (%edi)
        addl    $4, %edi
        decl    %ecx
        jnz     1b
```



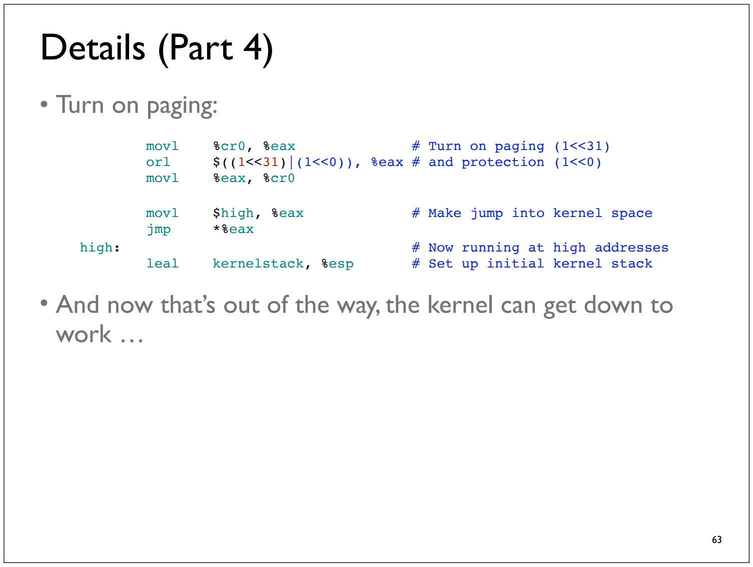
Details (Part 3)

- Install the lower and upper mappings in the initial page directory structure:

```
        movl    %esi, %edi      # Set up 1:1 and kernelspace mappings
        movl    $(PHYSMAP>>SUPERSIZE), %ecx
        movl    $(PERMS_KERNELSPACE),  %eax

1:      movl    %eax, (%edi)
        movl    %eax, (4*(KERNEL_SPACE>>SUPERSIZE))(%edi)
        addl    $4, %edi        # move to next page dir slots
        addl    $(4<<20), %eax  # entry for next superpage to be mapped
        decl    %ecx
        jnz     1b
```

- Load the CR3 register:

```
        movl    %esi, %cr3              # Set page directory

        mov     %cr4, %eax              # Enable super pages (CR4 bit 4)
        orl     $(1<<4), %eax
        movl    %eax, %cr4
```



Details (Part 4)

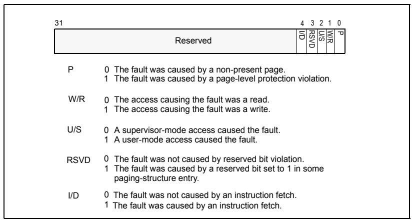
- Turn on paging:

```
        movl    %cr0, %eax              # Turn on paging (1<<31)
        orl     $((1<<31)|(1<<0)), %eax # and protection (1<<0)
        movl    %eax, %cr0

        movl    $high, %eax             # Make jump into kernel space
        jmp     *%eax
high:                                   # Now running at high addresses
        leal    kernelstack, %esp       # Set up initial kernel stack
```

- And now that's out of the way, the kernel can get down to work …

Page faults

- If program tries to access an address that is either not mapped, or that it is not permitted to use, then a page fault exception (14) occurs
- The address triggering the exception is loaded in to CR2
- Details of the fault are in the error code in the context:



P
0 The fault was caused by a non-present page.
1 The fault was caused by a page-level protection violation.

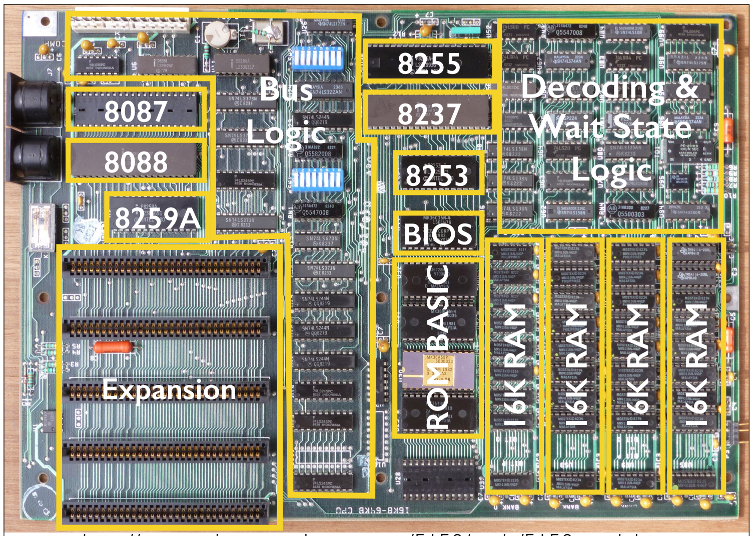
W/R
0 The access causing the fault was a read.
1 The access causing the fault was a write.

U/S
0 A supervisor-mode access caused the fault.
1 A user-mode access caused the fault.

RSVD
0 The fault was not caused by reserved bit violation.
1 The fault was caused by a reserved bit set to 1 in some paging-structure entry.

I/D
0 The fault was not caused by an instruction fetch.
1 The fault was caused by an instruction fetch.

Figure 4-12. Page-Fault Error Code



http://www.minuszerodegrees.net/5150/early/5150_early.htm

Ok, kernel, over to you …