CS 410/510
Languages & Low-Level Programming

Mark P Jones
Portland State University

Fall 2018

Week 4: Memory Management

Copyright Notice

• These slides are distributed under the Creative Commons Attribution 3.0 License

• You are free:
  • to share—to copy, distribute and transmit the work
  • to remix—to adapt the work

• under the following conditions:
  • Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows: “Courtesy of Mark P. Jones, Portland State University”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
Loose Ends

The Week 3 Lab: Context Switching

<table>
<thead>
<tr>
<th>kernel</th>
<th>user</th>
<th>user2</th>
</tr>
</thead>
</table>

Output from kernel:
- User1 code is at 0x411058
- User1 code is at 0x421058
- User data segment is 0x38
- 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
- output: World!

Output from first user process:
- User1 code does not return

Output from second user process:
- User2 code
- User2 console
- User2 console
- User2 console
- User2 console
- User2 console
- User2 console
- User2 console
- User2 console
- User2 console
- output: World!
Port I/O

Memory mapped I/O

CPU

Memory address space

RAM | I/O
---|---
0  | 8KB | 16KB | 24KB | 32KB | 40KB | 48KB | 56KB | 64KB
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 can use the same address bus and data bus
  with a signal to distinguish between memory and port access
• You can write a byte/short/word to an I/O port using:
  \[
  \text{out[b|w|l] [%al,%ax,%eax], [imm8|%dx]}
  \]
  (use \text{imm8} for 8 bit port numbers, otherwise use \%dx)
• You can read a byte/short/word from an I/O port using:
  \[
  \text{in[b|w|l] [imm8|%dx], [%al,%ax,%eax]}
  \]

Port I/O using gcc inline assembly

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

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

• Arcane syntax, general form:
  \[
  \text{asm ( template : output operands : input operands : clobbered registers });}
  \]

• 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

<table>
<thead>
<tr>
<th>Port Range</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00-0x1f</td>
<td>First DMA controller (8237)</td>
</tr>
<tr>
<td>0x20-0x3f</td>
<td>Programmable Interrupt Controller (PIC1) (8259A)</td>
</tr>
<tr>
<td>0x40-0x5f</td>
<td>Programmable Interval Timer (PIT) (8253/8254)</td>
</tr>
<tr>
<td>0x60-0x6f</td>
<td>Keyboard (8042)</td>
</tr>
<tr>
<td>0x70-0x7f</td>
<td>Real Time Clock (RTC)</td>
</tr>
<tr>
<td>0x80-0x8f</td>
<td>DMA ports, Refresh</td>
</tr>
<tr>
<td>0xa0-0xbf</td>
<td>Programmable Interrupt Controller (PIC2) (8259A)</td>
</tr>
<tr>
<td>0xc0-0xdff</td>
<td>Second DMA controller (8237)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x3f0-0x3f7</td>
<td>Primary floppy disk drive controller</td>
</tr>
<tr>
<td>0x3f8-0x3ff</td>
<td>Serial Port 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Serial port output in assembly

```assembly
.set PORTCOM1, 0x3f8

.serial_putchar:
    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
```

To the datasheet!
To the datasheet!

Serial port output in assembly

```
.set PORTCOM1, 0x3f8

serial_putchar:
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 $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
```
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
• The CPU has an interrupt pin
• Connect it to a timer to generate regular timer interrupts!

• 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 ... twice as many input pins ...

- Two PICs chained together
- Any interrupt on PIC2 triggers interrupt 2 on PIC1
IDT structure

Protected mode exceptions

System call entry points

Hardware IRQs

PIC1  PIC2

0  16  32  48  …  128
130  132  …  255

Initializing the PICs

.equ IRQ_BASE, 0x20  # lowest hw irq number
.equ PIC_1, 0x20
.equ PIC_2, 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, $$\port+1
  movb $\info, %al  # ICW3: configure for two PICs
  outb %al, $$\port+1
  movb $0x01, %al  # ICW4: 8086 mode
  outb %al, $$\port+1
  movb $\init, %al  # OCW1: set initial mask
  outb %al, $$\port+1
.endm

initPIC: initpic PIC_1, IRQ_BASE, 0x04, 0xfb  # all but IRQ2 masked out
        initpic PIC_2, IRQ_BASE+8, 0x02, 0xff
ret
Initializing the PICs

.equ IRQ_BASE, 0x20    # lowest hw irq number
.equ PIC_1, 0x20
.equ PIC_2, 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)
    movb $\info, %al    # ICW3: configure for two PICs
    outb %al, $(port+1)
    movb $0x01, %al    # ICW4: 8086 mode
    outb %al, $(port+1)
    movb $init, %al    # OCW1: set initial mask
    outb %al, $(port+1)
.endm

initPIC: initpic PIC_1, IRQ_BASE, 0x04, 0xfb  # all but IRQ2 masked out
    initpic PIC_2, IRQ_BASE+8, 0x02, 0xff
    ret

Initializing the PICs

.equ IRQ_BASE, 0x20    # lowest hw irq number
.equ PIC_1, 0x20
.equ PIC_2, 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)
    movb $\info, %al    # ICW3: configure for two PICs
    outb %al, $(port+1)
    movb $0x01, %al    # ICW4: 8086 mode
    outb %al, $(port+1)
    movb $init, %al    # OCW1: set initial mask
    outb %al, $(port+1)
.endm

initPIC: initpic PIC_1, IRQ_BASE, 0x04, 0xfb  # all but IRQ2 masked out
    initpic PIC_2, IRQ_BASE+8, 0x02, 0xff
    ret
Enabling and disabling individual IRQs

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

```c
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:

```c
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:

```c
static inline void maskAckIRQ(byte irq) {
    if (irq&8) {
        outb(0xa1, (1<<(irq&7)) | inb(0xa1));
        outb(0xa0, 0x60|(irq&7)); // EOI to PIC2
        outb(0x20, 0x62);         // EOI for IRQ2 on PIC1
    } else {
        outb(0x21, (1<<(irq&7)) | inb(0x21));
        outb(0x20, 0x60|(irq&7)); // EOI to PIC1
    }
}
```

• When the interrupt has been handled, reenable the IRQ
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](image)

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

Example: Programming the PIT

To configure for timer interrupts:

```c
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);
}
```
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

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

• 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

```c
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);
...

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

```c
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);
...
```

```c
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:

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>
Practical reality

- IA32 partitions the 32-bit, 4GB address space into 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 into 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

- 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 into 64 pages
- But we are only use a small number of those pages…
- ... in fact, only a small number of the rows
- Then we can represent the full table more compactly as a tree:
Paging with 4KB pages

- 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

![Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging](image)

![Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging](image)

![Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging](image)
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(cessed) 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 “inv1pg 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 space

<table>
<thead>
<tr>
<th>user code &amp; data</th>
<th>user stack</th>
<th>OS</th>
</tr>
</thead>
</table>
```

- 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

- CR0: Selects 32-bit or 64-bit mode
- CR1: Enables protected mode
- CR2: Enables paging
- CR3: Page-Fault Linear Address
- CR4: Enables use of super pages

*Figure 2-7. Control Registers*
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):

```c
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:

```c
/* 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:

```c
#define fromPhys(t, addr) (((t)(((unsigned)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

\[
\begin{align*}
    \text{KERNEL\_SPACE} &= 0xc0000000 \quad \# \text{Kernel space starts at 3GB} \\
    \text{KERNEL\_LOAD} &= 0x00100000 \quad \# \text{Kernel loads at 1MB}
\end{align*}
\]

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

\[
\begin{align*}
    \text{OUTPUT\_FORMAT(elf32-i386)} \\
    \text{ENTRY(physentry)} \\
    \text{SECTIONS} \\
    \text{physentry} &= \text{entry} - \text{KERNEL\_SPACE}; \\
    \cdot &= \text{KERNEL\_LOAD} + \text{KERNEL\_SPACE}; \\
    \text{.text \text{ALIGN}(0x1000)} : \text{AT(ADDR(.text)) - KERNEL\_SPACE)} \{ \\
    \text{.text\_start} &= \cdot; *(.text) *(.handlers) \text{.text\_end} = \cdot; \\
    *(.rodata) \\
    \text{.data} \\
    \text{.start\_bss} &= \cdot; *(\text{COMMON}) *(.bss) \text{.end\_bss} = \cdot;
\}
\]

Details (Part 2)

• Reserve space for an initial page directory structure:

\[
\begin{align*}
    \text{.data} \\
    \text{.align} &= (1<<(\text{PAGESIZE}) \\
    \text{initdir} : \text{.space} &= 4096 \quad \# \text{Initial page directory}
\end{align*}
\]

• Zero all entries in the table:

\[
\begin{align*}
    \text{leal} &= (\text{initdir}\text{-KERNEL\_SPACE}), \%edi \\
    \text{movl} &= \%edi, \%esi \quad \# \text{save in} \%esi \\
    \text{movl} &= \$1024, \%ecx \quad \# \text{Zero out complete page directory} \\
    \text{movl} &= \$0, \%eax  \\
    \text{l:} \\
    \text{movl} &= \%eax, (%edi) \\
    \text{addl} &= \$4, \%edi \\
    \text{deql} &= \%ecx \\
    \text{jnz} &= \text{l}
\end{align*}
\]
Details (Part 3)

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

  ```assembly
  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:

  ```assembly
  movl  %esi, %cr3       # Set page directory
  mov  %cr4, %eax        # Enable super pages (CR4 bit 4)
  orl  $(1<<4), %eax     # and protection (1<<0)
  movl  %eax, %cr4
  ```

Details (Part 4)

- Turn on paging:

  ```assembly
  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:
  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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>The fault was caused by a non-present page.</td>
</tr>
<tr>
<td></td>
<td>0 The fault was caused by a page-level protection violation.</td>
</tr>
<tr>
<td>W/R</td>
<td>The access causing the fault was a read.</td>
</tr>
<tr>
<td></td>
<td>1 The access causing the fault was a write.</td>
</tr>
<tr>
<td>U/S</td>
<td>A user-mode access caused the fault.</td>
</tr>
<tr>
<td></td>
<td>1 A user-mode access caused the fault.</td>
</tr>
<tr>
<td>RSVD</td>
<td>The fault was not caused by reserved bit violation.</td>
</tr>
<tr>
<td></td>
<td>0 The fault was not caused by reserved bit violation.</td>
</tr>
<tr>
<td></td>
<td>1 The fault was caused by a reserved bit set to 1 in some paging-structure entry.</td>
</tr>
<tr>
<td>I/D</td>
<td>The fault was not caused by an instruction fetch.</td>
</tr>
<tr>
<td></td>
<td>0 The fault was not caused by an instruction fetch.</td>
</tr>
<tr>
<td></td>
<td>1 The fault was caused by an instruction fetch.</td>
</tr>
</tbody>
</table>

Figure 4-12. Page-Fault Error Code

Ok, kernel, over to you ...