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

Port I/O

Memory mapped 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:
  ```
  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

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

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

- Arcane syntax, general form:
  ```c
  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>0x20-0x2f</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>0x80-0x9f</td>
<td>DMA ports, Refresh</td>
</tr>
<tr>
<td>0x00-0xff</td>
<td>Programmable Interrupt Controller (PIC2) (8259A)</td>
</tr>
<tr>
<td>0x00-0xff</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

```
.set PORTCOM1, 0x3f8
serialputc:
push %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
```

Serial port output in assembly

```
.set PORTCOM1, 0x3f8
serialputc:
pushl %esx
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!

- 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!
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?

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 chained together
- Any interrupt on PIC2 triggers interrupt 2 on PIC1
### Initializing the PICs

- **.equ** `IRQ_BASE`, 0x20    # lowest hw irq number
- **.equ** `PIC_1`, 0x20
- **.equ** `PIC_2`, 0x30

# Send ICWs (initialization control words) to initialize PIC.
```assembly
.macro initpic port, base, info, init
  movb $0x11, %al
  outb %al, $\langle$port$\rangle$ # ICWI: Initialize + will be sending ICW4
  movb $\langle$base$\rangle$, %al      # ICW1: Initialize
  outb %al, $\langle$(port$+1)\rangle$
  movb $\langle$info$\rangle$, %al    # ICW3: configure for two PICs
  outb %al, $\langle$(port$+1)\rangle$
  movb $(\langle$init$\rangle$, %al    # ICW4: 8086 mode
  outb %al, $\langle$(port$+1)\rangle$
  movb $0xa1, %al  # ICW1: set initial mask
  outb %al, $\langle$(port$+1)\rangle$

.initPIC: initpic PIC_1, IRQ_BASE, 0x04, 0xfb # all but IRQ2 masked out
    initpic PIC_2, IRQ_BASE+$\langle$, 0x02, 0xff
    ret
.endm
```

### Enabling and disabling individual IRQs

- **.static inline** void `enableIRQ(byte irq)` {
  if (irq) {
    outb(0x62, (irq&0x60)) & inb(0x01);
  } else {
    outb(0xa1, -(irq&0x21)) & inb(0x01);
  }
}

- **.static inline** void `disableIRQ(byte irq)` {
  if (irq) {
    outb(0x62, (irq&0x60)) & inb(0x01);
  } else {
    outb(0xa1, -(irq&0x21)) & inb(0x01);
  }
}

- **Individual IRQs are enabled by clearing the mask bit in the corresponding PIC:**
  ```assembly
  static inline void enableIRQ(byte irq) {
    if (irq) {
      outb(0x62, (irq&0x60)) & inb(0x01);
    } else {
      outb(0xa1, -(irq&0x21)) & inb(0x01);
    }
  }
  
  static inline void disableIRQ(byte irq) {
    if (irq) {
      outb(0x62, (irq&0x60)) & inb(0x01);
    } else {
      outb(0xa1, -(irq&0x21)) & inb(0x01);
    }
  }
  ```

- **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:
    ```assembly
    static inline void maskAndIRQ(byte irq) {
      if (irq) {
        outb(0x62, (irq&0x60)) & inb(0x01);
        outb(0x20, 0x60); // EOI to PIC2
        outb(0x20, 0x62); // EOI for IRQ2 on PIC1
      } else {
        outb(0x20, 0xff); // EOI to PIC1
      }
    }
    ```
    - When the interrupt has been handled, reenable the IRQ

```assembly
initPIC: initpic PIC_1, IRQ_BASE, 0x04, 0xfb # all but IRQ2 masked out
    initpic PIC_2, IRQ_BASE+$\langle$, 0x02, 0xff
    ret
```
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

- 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, …)

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

The first user program

```
unsigned flag = 0;
for (i=0; i<600; i++) {
    ...
    printf("My flag is at 0x4025b0\n", &flag);
    while (flag==0) {
        /* do nothing */
    }
    printf("Somebody set my flag to %d\n", flag);
    ...
}
```

- 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

```
unsigned flag = 0;
for (i=0; i<600; i++) {
    ...
    printf("My flag is at 0x4025b0\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\n", flagAddr);
    *flagAddr = 1234;
    printf("\n\nUser2 code does not return\n");
    for (;;) { /* Don't return */
    }
}
```

```
volatile unsigned flag = 0;
for (i=0; i<600; i++) {
    ...
    printf("My flag is at 0x4025b0\n", &flag);
    while (flag==0) {
        /* do nothing */
    }
    printf("Somebody set my flag to 1234!\n", flag);
    ...
}
```

Marking the flag as volatile

```
volatile unsigned flag = 0;
for (i=0; i<600; i++) {
    ...
    printf("My flag is at 0x4025b0\n", &flag);
    while (flag==0) {
        /* do nothing */
    }
    printf("Somebody set my flag to 1234!\n", flag);
    ...
}
```

```
for (i=0; i<1200; i++) {
    ...
    unsigned* flagAddr = (unsigned*)0x4025b0;
    printf("flagAddr = 0x\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 \texttt{x+x} as being equivalent to \texttt{2\times x}:
  - There is no way for the value in \texttt{x} to change from one side of the + to the other (no intervening assignments)
  - The compiler can replace two attempts to read \texttt{x} with a single read, without changing the behavior of the code
- Marking a variable as \texttt{volatile} indicates that the compiler should allow for the possibility that the stored value might change from one read to the next
- The \texttt{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

- “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>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

Practical reality

- IA32 partitions the 32-bit, 4GB address space in to 4KB pages
  - page number: 20 bits, offset: 12 bits
  - super page number: 10 bits, offset: 22 bits
- 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

- 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

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

```
<table>
<thead>
<tr>
<th>Bits</th>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:22</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>Bits</th>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:22</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>
```
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 using 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:

```
  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
```

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

```
Address of page directory1 Ignored CR3
Bits 39:32 of address of 4PTE page frame Reserved (must be 0)
Bits 31:22 of address2 Ignored C D A P
Ignored C D A P
Reserved (must be 0)
Ignored C D A P
Reserved (must be 0)
Ignored C D A P
```

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”)
- Global 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 into CR3 will flush the TLB
- The “invlpag addr” instruction removes TLB entries for a specific address
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:

  ![Virtual Memory Layout Diagram]

- 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

- Enables use of super pages.
- Enables paging.
- Enables protected mode.
- Set by GRUB.

Initialization

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

  ![Initial Memory Layout Diagram]

- To virtual address spaces with paging enabled?

  ![Virtual Memory Initial Layout Diagram]

- 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.

  ![Initial Page Directory Diagram]

- 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]; }
  ```

- But sometimes we have to work with physical addresses:

  ```c
  static inline void setPdir(unsigned pdir, {
    asm("movl %0, @cr3")
  }
  ```

-’tail’ movl b, @cr3: ‘’movl’ (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
  KERNEL_SPACE = 0x00000000
  #define toPhys(ptr) (($(unsigned)ptr)+KERNEL_SPACE))
  #define fromPhys(t, addr) ((t)(((unsigned)addr)+KERNEL_SPACE))
  ```

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

### Details (Part 1)

- Constants to describe the virtual address space

  ```c
  KERNEL_SPACE = 0x00000000  # Kernel space starts at 1MB
  KERNEL_LOAD = 0x00010000    # Kernel loads at 1MB
  ```

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

  ```c
  OUTPUT_FORMAT(elf32-i386)
  ENTRY(phyentry)
  SECTIONS {
    _start_bss = .space
    _start = .space
    _text_start = .text
    _text_end = .text
    _end_bss = .space
  }
  ```

### Details (Part 2)

- Reserve space for an initial page directory structure:

  ```c
  .data
  #align (<<PAGESIZE)
  initdir.space 4096  # Initial page directory
  ```

- Zero all entries in the table:

  ```c
  leal \{initdir-KERNEL_SPACE\}, %edi
  movl %edi, %esi      # save in %esi
  movl $00000001, %ecx # zero out complete page directory
  movl %esi, %ecx
  addl %4, %esi
  decr %ecx
  jnz 1b
  ```

### Details (Part 3)

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

  ```c
  movl %esi, %edi      # Set up 1:1 and kernelspace mappings
  movl $(\$KERNEL_SPACE|<<10), %ecx
  movl \$\(\$\$KERNELSPACE\), %eax
  ```

- Load the CR3 register:

  ```c
  movl %esi, %cr3     # Set page directory
  movl %cr4, %rax     # Enable super pages (CR4 bit 4)
  ```

- 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 into CR2.
- Details of the fault are in the error code in the context:

Figure 4-12. Page-Fault Error Code

Ok, kernel, over to you …