Class Announcements

1. Programming Assignment 3 is due by 9 AM day after tomorrow (11/04 – Wednesday). As usual, you can submit it upto 48 hours after the deadline with penalties.
Lecture Overview

1. Interrupts and Exception
2. Intro to Processes
To control access to individual devices, we place code that communicates with the devices into the OS. (This portion of the OS is also called the kernel) and in special routines called device drivers.

```
x86

1. call getchar

int getchar()
```

```
stdio library

1. int getchar()
```

```
app code

1. ch = getchar();
```

```
keyboard device driver
```

1. ret

2. int syscall

1. user types 'X'

2. 'X' → Keyboard_Data
1. 'Q' appears in memory

2. character appears on display
The keyboard driver code:

```
movl Keyboard.Data, %eax
int
ret from syscall
```

The display driver code:

```
movl %eax, Display.Data
int
ret from syscall
```

What happens if user has not typed a char on the keyboard?
We want **blocking input**.
(no ret from syscall until there is a char)

We need a **status bit**

\[ \begin{array}{c}
1 & \text{ready} \\
0 & \text{not ready (busy)} \\
\end{array} \]

Place this bit into its own memory mapped word & make it the **msb**, so code can test for \( \geq 0 \) or \( \leq 0 \).

Diagram:

```
Keyboard Data
Keyboard Status
```

```
Display Data
Display Status
```
1. user types 'X'

2a. 'X' into Keyboard_Data

2b. 1 into Keyboard_Status msb
Now, driver code uses a spin wait loop
(to implement blocking I/O)

```
kb_spin:
  testl  Keyboard_Status, Keyboard_Status
  jz     kb_spin
  movl   Keyboard_Data, %eax
  ret from syscall

disp-spin:
  testl  Display_Status, Display_Status
  jz     disp-spin
  movl   %eax, Display_Data
  ret from syscall
```
memory mapped

addresses 0xfffff0008
0xfffff000c

Data
Status

Keyboard
Display

addresses 0xfffff0010
0xfffff0014

Data
Status

P

M
Byte transfers are OK, 
But, what about faster devices that like to transfer more than a byte?

the solution: **DMA**

**Direct Memory Access**
Issue for spin wait loop implementations:

One byte only in _Data has the potential for an incorrect result.

For example, if the user types 2 characters on the keyboard before `getchar()` is called.

The needed fix introduces a kernel-maintained queue for each device.

Then, the kernel polls to check status bits and handle any ready devices.
here is an analogy...

Teacher  is  OS

each student  is  I/O device

Consider the *inefficiency* of OS polling.
Because polling is so inefficient,

instead of

OS

\[ \text{ask} \]

device 1 ready?  device 2 ready?  \ldots  device n ready?
Turn the situation upside down

OS

device 1  device 2  ...  device n

©These slides may be freely used, distributed, and incorporated into other works.
An anatomy of an exception

An exceptions is an abrupt change in control flow.

Examples: div by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C
Classes of Exceptions

1. **Interrupts (Asynchronous):** Always return to next instruction.

2. **Traps & System Calls (Synchronous):** Always return to next instruction.

3. **Faults (Synchronous):** Might return to next instruction.

4. **Aborts (Synchronous):** Never returns
Interrupts

Examples of Interrupts:

- Timer interrupt
- Arrival of a packet from a network
- When a key is pressed on the keyboard
- When the mouse is moved
Traps are intentionally issued by executing an instruction.

Example: System calls

1. Application makes a system call
2. Control passes to handler
3. Trap handler runs
4. Handler returns to instruction following the syscall
Faults

Faults result from error conditions that might be correctable.

Examples: Page fault, Divide error

(1) Current instruction causes a fault

(2) Control passes to handler

(3) Fault handler runs

(4) Handler either reexecutes current instruction or aborts.
Aborts result from unrecoverable fatal errors.

Example: parity errors due to DRAM bit corruption

1. Fatal hardware error occurs
2. Control passes to handler
3. Abort handler runs
4. Handler returns to abort routine

abort
Exception Table

Exception table

Exception table

Code for exception handler 0

Code for exception handler 1

Code for exception handler 2

Code for exception handler n-1
Exception Table lookup

Exception is similar to procedure calls except for some important differences:

- Return address is not the next instruction always
- Push EFLAGS register also onto kernel stack
- Run exception handler in kernel mode
<table>
<thead>
<tr>
<th>Exception Number</th>
<th>Description</th>
<th>Exception Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
<td>Fault</td>
</tr>
<tr>
<td>13</td>
<td>General protection fault</td>
<td>Fault</td>
</tr>
<tr>
<td>14</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>18</td>
<td>Machine check</td>
<td>Abort</td>
</tr>
<tr>
<td>32-127</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
<tr>
<td>128 (0x80)</td>
<td>System call</td>
<td>Trap</td>
</tr>
<tr>
<td>129-255</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
</tbody>
</table>
Rather important, but not covered in textbook:

If running a handler, and a new interrupt request arrives, what should happen?

- Continue on, complete handling of current interrupt, then, when done, deal with new request? (probably) non-reentrant

- Interrupt the handling of this interrupt? reentrant
Every architecture has a control bit which identifies whether the F+E cycle is paying attention to IRQs. Called interrupt enable

```
on x86:
    EFLAGS
        IF
            1 enabled
            0 disabled
```

fetch+execute cycle:

- if IF=1 and interrupt requested, go handle it

1. fetch instr
2. PC update
3. decode
...
Consider the x86 instruction:
cli    clear IF

What happens if an application includes this cli instruction?

Irrelevant (to this discussion) x86 instruction:
sti   set IF
OS relies on clock interrupts to allocate processing time. . .

As the clock interrupts, the kernel runs, and it decides which program runs next.
Clarified instruction:

cli clear IF if CPL is high enough, otherwise trap

Does sti also need to be a privileged instruction?
Keep IF = 1 while CPL = 00.

(So, applications can always be interrupted)

HW must disable interrupts while saving state & at least until first instruction within handler is fetched.

Better Definitions:

nonreentrant  IF = 0 the entire time a handler runs

reentrant  interrupts may be reenabled while handler runs (usually only for higher priority requests)
Non reentrant

timeline

IRQ 1

IRQ 2

IRQ 2
Reentrant timeline

dev 2 interrupts are higher priority