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 \( \rightarrow \) is \( \rightarrow \) OS

each student \( \leftarrow \) is \( \rightarrow \) I/O device

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

instead of

```
device 1 ready?  device 2 ready?  ...  device n ready?
```
Turn the situation upside down

OS

ready!

device 1  device 2  ...  device n
Exceptions

1. Traps

Examples:

- int instruction
- Intel's faults, esp. page faults
- Intel's aborts:
  - bad address for a mem. access
  - divide by zero

They are synchronous.

Rerunning the same program with the same inputs will again cause a trap at the same instruction.

They are caused by the running program.
2. **Interrupts**

Examples:
- device ready
- HW error

Intel calls this **Interrupt Request**

They are **asynchronous**. Interrupts occur at a point in time that is unrelated to the program's fetch & execute cycle.
The mechanism for handling exceptions is the same. Both HW & SW is needed.

Assume a program application is running....

(for interrupt, finish F+Ecyle)

save state (IP, EFLAGS, registers)

set new state (reason for exception, registers (%esp), privilege level)

find & run handler

if appropriate:

restore saved state
return to previously running code
For an interrupt, the application does not run for a period of time (while handler runs), but is given no indication that processing time was lost.
Some specifics of the x86 mechanism:

- **Reason for exception**: called a **vector**
- A small, unsigned integer values of 0 - 255

The **vector** becomes an index into an array of **descriptors** or (Intel) **gates**

```
  +-----------------+   +-----------------+
  |    0            |---|    1            |
  |    gate: vector |---|    gate: vector  |
  +-----------------+   +-----------------+
                      +-----------------+
                      |    2            |
                      +-----------------+
                          +-----------------+
                          |    255          |
                          +-----------------+
```

**Called**

- **Interrupt**
- **Descriptor Table**
- **IDT**
Each gate: 8 bytes

Used to find address of handler code.
Segment selector

offset into

\[
\text{address of handler code} = \text{base addr} + \text{offset MSbits} \parallel \text{offset LSbits}
\]
Part of a gate

<table>
<thead>
<tr>
<th>#bits</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td></td>
<td>DPL</td>
<td>011?</td>
<td>I/T</td>
<td>000</td>
<td>reserved</td>
</tr>
</tbody>
</table>

... 0 if interrupt
     1 if trap

Descriptor Privilege Level =

00 kernel
01
10
11 user (least privilege)

... Present = tells if handler's code segment is in main memory or out on disk
Current Privilege Level (CPL) kept in the code segment register, %CS.
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:

```
+---+---+---+---+---+---+---+---+
| q | ... | z | i | f |
```

EFLAGS

**fetch+execute cycle:**

1. if \( IF = 1 \) and interrupt requested, go handle it
2. fetch instr
3. PC update
4. decode

\( IF \):

- 1 enabled
- 0 disabled
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
Reentrant timeline

IRQ 1

IRQ 2

dev 2 interrupts are higher priority