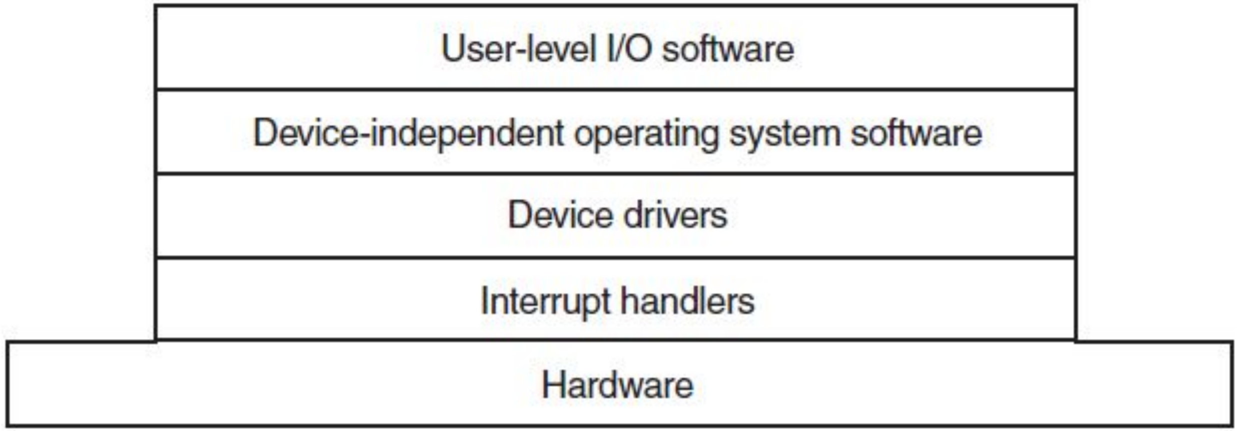
I/O Software Layers

Module 20

Layers - get more specific



Interrupt Handlers

- A device driver will send a control signal to a device's controller.
  - i.e. read from disk 10 bytes, put in memory at address 342AFE.
- Device driver may then **block** - perhaps calling **down** on a semaphore and then **waiting**.
- When interrupt is fired, interrupt handler can perform an **up** on the semaphore - waking it up.

Details of an Interrupt

When an interrupt fires:

1. Save registers (PSW, PC) so we can return to whatever process was running
2. Set up context/stack for interrupt procedure
3. Acknowledge interrupt controller
4. Run the interrupt procedure (extract data from device controller registers)
5. Signal device driver to wake back up
6. Run scheduler

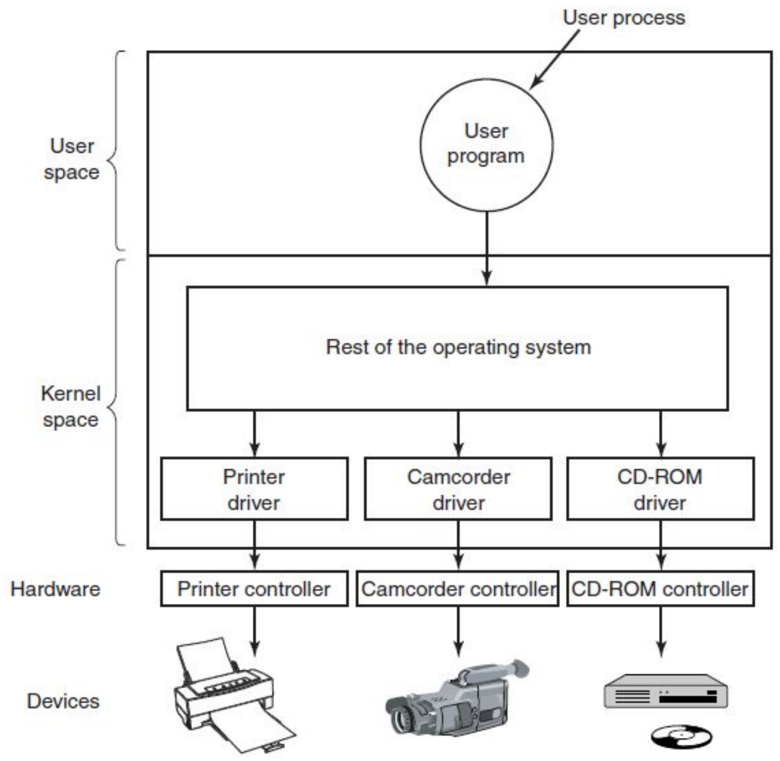
These steps are slightly different than the ones enumerated on page 357 in the text… but are similar in idea.

Device Drivers

Device drivers are the translators between OS parameters and registers / control signals on the device controller.

- Some types of devices use generic drivers:
  - SCSI disks, USB devices
  - The device controllers “speak the same language”.

Device Drivers



Most device drivers run in kernel mode

Device Driver Interface

Device Drivers are written by 3rd parties

They adhere to an interface, defined by the OS

2 main interfaces: Block and Stream

```
class BlockDevice {
    virtual int read(int block_number, int num_blocks, int dest_address) = 0;
    virtual int write(int block_number, int num_blocks, int src_address) = 0;
};
```

OS Software (Device Independent)

- The OS is, of course, device **independent**
  - It defines the interface (last slide)
  - It also provides:
    - Buffering
    - Error reporting
    - Allocation/Release of devices (to processes)
    - Defines block sizes

User-Space I/O

We know that mode switch from user to kernel the moment the **system call** is made

- Simple I/O routines perform very little in user space:

```
count = write(fd, buffer, nbytes);
```

- This call basically sets three registers and invokes the system call!

User-space I/O

- Other routines might very well do some work before making the system call:

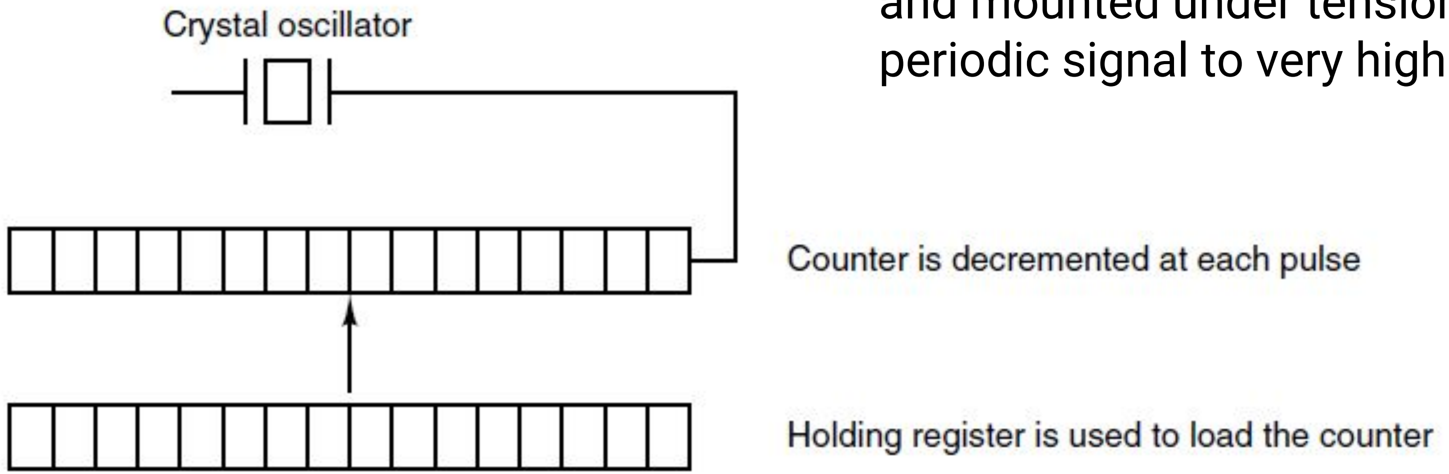
```
printf(“The square of %3d is %6d\n”, i, i*i);
```

  - Eventually, printf and write call the same system call - but printf does a whole lot more first!
- Entire processes might also be in user space!
  - Print spoolers

Clocks

Hardware Timer

Quartz, cut to specific specifications, and mounted under tension, emits a periodic signal to very high accuracy


Crystal oscillator
Counter is decremented at each pulse
Holding register is used to load the counter

Electronics can reduce/increase the frequency, and more electronics can cause a counter to decrement on each “pulse”.

Programmable Timer

- To make a timer, we have a holding register - set by a system call.
  - When the system call is invoked, the holding register is copied to the counter
  - An interrupt fires when the clock hits 0
  - Can be automatically restarted (with holding) register to provide “clock ticks”

Side note… *actual time*.

- A “timer” is not what keeps your computer knowing what time it is between restarts…
  - On your motherboard, there is a Quartz crystal oscillator and a dedicated register holding current time
  - You also have a small (but very long-lasting) battery that keeps powering the register and electronics to attenuate the Quartz oscillator
  - On an old computer, they batteries can die, and the computer will lack the ability to hold time.
  - Nowadays, we also synchronized with clocks on the internet...

Soft Timers

- Hardware interrupts are costly
  - Interrupt latency means these “timers” are never perfectly accurate - even if the Quartz is!
  - Alternatively, the OS can keep a list of “software timers”, and each time it is doing anything, it can review the list to see if they’ve expired
    - Quite practical in a ***preemptive operating system!***

User Interface Devices

Keyboard Drivers: More Layers

- A Keyboard has a microcontroller
- On key press, and key release, the controller sets a register value with an associated **scan code**.
- An interrupt is fired
- The Keyboard **device driver** take care of the rest

What is the rest?

- Typically, a driver will process key up/down events to create “press” events
  - With a delay, when holding the key down
- It also converts scan code to ASCII character codes
  - Ctrl+, Shift Keys, etc.

Mouse Software

Rubber ball is on rollers, the rollers signal the controller how much movement has occurred in x/y direction

Optical Mouse

Image processing chip (DSP) takes **frequent** pictures (low resolution) of surface, calculates distance moved

Device controller sends interrupts for delta x/y and button state, similar to keyboard

Output software

Section 5.6.2 in text is an interesting read - but it’s not normally considered part of the **operating system**.

Of course, the graphics device does indeed have a driver, one which is exceptionally important. See CMPS 342 :)