Input/Output Devices

Chapter 5 of Tanenbaum.

Have both hardware and software. Want to hide the details from the programmer (user).

Ideally have the same interface to all devices (device independence). Think of UNIX. Process doesn't need to distinguish between input coming from terminal, the network, a file or another process.

I/O Devices

Two principal types:

1. block devices: stores information in fixed size blocks (128 to 1024 bytes). Example: disks.
   - can read or write blocks independently
   - related is ability to seek

   Tape drive: can seek and read, but probably cannot write at arbitrary place.
2. character devices: stream of characters (independent of blocks). Examples: terminals, line printers, network drivers
   - not addressable
   - no seek operation

Clocks fall outside of this categorization.

Device Controllers

Operating systems do not deal with devices directly. Rather there is a mechanical and electronic portion. Electronic portion is a *device controller*. A printed circuit card.

Look at Silbershatz Fig 13.1. Also a network interface card.

Example: CRT controller takes care of details of rescanning the CRT beam

How do controllers and the CPU communicate?

Use memory-mapped I/O

- CPU puts command in registers (I/O address space) (Silbershatz Fig. 13.2) example commands: read, write, seek. Also parameters.
- CPU goes off and does other things.
- When device done, the controller causes an interrupt. CPU reads any results from the controller's registers.

See Fig 5-5.

Polled I/O

No interrupts used. CPU puts request in controller's registers and then polls waiting for the request to finish.

Also called programmed I/O.

Might be used for debugging in operating system interrupt handler because it doesn't block.

Direct Memory Access (DMA)

What about large amounts of data to transfer? For example, disk read. Controller gets the data and buffers it.

CPU could only get data in small chunks (byte or word at a time).

Look at Fig. 5.4. Idea is to free CPU from being involved with transfer of data.

CPU also gives:

- memory address (physical, so the page must be tied down)
- count

Principles of I/O Software

Issues of the software:

- efficiency—I/O operations are very slow compared to main memory and the CPU. Want to avoid them being a bottleneck on the system.
- device independence—programs do not need to know about input or output device.
- uniform naming—same naming convention regardless of the device. Large systems typically have multiple disk drives, but user sees a common file space on top. Extend to a distributed file system.
- error handling—as close to device as possible, if controller detects an error it ideally corrects it or rereads (use checksum to detect)
- synchronous vs. asynchronous—physical I/O is asynchronous (interrupt-driven). However want to make it appear synchronous (blocking) to the user. Can also use *polled I/O* where the device driver continually polls the device (may be used by kernel to not wait in debugging a device driver).
- buffering—may have to buffer data read from a device (such as a packet from network). However, may end up and copy buffer multiple times.
- sharable vs. dedicated—file is shared, printer cannot be shared. Operating System must handle both kinds of devices.

I/O Software Layers

Correspond to layers in Fig 5-16.

Interrupt Handlers

Bowels of the system. Occurs when a device controller wants to tell CPU something (clock tick, write done, ready to read more, etc)

What happens on interrupt:

- Interrupt handler gets invoked.
- Could send a message to blocked device driver process.
- On other systems a semaphore is signalled

Device Drivers

Contains device-dependent code. May have a device driver for a class of related devices (terminal driver for example).

It knows details about device.

```
device independent request -> device driver -> device dependent request
```

It may queue up the request if device is already busy. Will send that request when the current request is complete. Puts data in registers and retrieves results as needed.

Reports results to device-independent software.

See Fig 5-11.

Device-Independent I/O Software

Two major functions:

- perform I/O functions common to all devices
- provide a uniform interface to the user-level software

- naming—read/write to terminal using */dev/tty*
- protection—look at file protection (user, group, world)
- buffering—read one byte from disk. Will actually read a block of data and pass one byte to program. Also a downside as shown in Fig 5-15.

User-Space I/O Software

For example, *printf()* is a library routine that calls *write()*.

Also user-level software to support other operations. *Spooling* for example where a *daemon* process controls access to a spooling directory. When you print, the file is put in the directory and when your turn comes it is printed.

Also used for file transfer. UUCP networks use this approach.