I/O and Operating Systems

- As previously discussed, all I/O must go through the operating system
- A typical system has (or could have) many different types of devices
- What kind of software talks to these devices?
- What is the interface between the OS and that software?
- What is the interface to application programs?
- Are there any hardware constraints?

1 / 36

I/O Models

- All operating systems impose tight controls on disk access, because the OS itself needs to access the disk and control file storage
- Operating systems frequently provide high-level access to common types of devices: serial ports, printers, tapes, etc.
- Some operating systems permit user-specified complex I/O to some devices (tapes, displays)
- The more "normal" an I/O device is, the more likely it is to be accessed by a standard model
- Try for an I/O model that lets most applications be device-independent

2 / 36

The Unix Model

- Two categories, *block devices* and *character devices*
- File systems live on top of block devices
- Block devices are designed for “give me N blocks starting at block number M”
- Almost never used directly by the application programmer
- Character devices — everything else — return a series of unstructured bytes
- Provides a high degree of device independence — for *most* applications. . .

3 / 36

The Windows Model

- More oriented towards bit maps
- Screens are bitmaps
- Printers are bitmaps
- Disks? Disks use the same block model as in Unix

4 / 36

Networking

- Network devices are completely different
- They're neither block nor character-oriented
- They're used very differently by the OS

5 / 36

I/O Model

- Character devices
- Block devices
- Bitmaps
- Networks

6 / 36

Back to Unix

- All devices have filenames:

```
$ ls --time-style "+" -Ggl \
    /dev/sda1 /dev/pts/6
crw--w----  1 136, 6  /dev/pts/6
brw-rw----  1   8, 1  /dev/sda1
```

- The “c” indicates a character I/O devices; the “b” indicates a block device
- The “136” and the “8” are *major device* numbers
- The “6” and the “1” are *minor device numbers*
- Device files are created with `mknod`

7 / 36

Major Device Numbers

- Major device numbers indicate the device type. More specifically, they indicate the *device driver* to be used
- Device driver numbering is very system-specific
- Block and character device numbers are separate namespaces

8 / 36

Minor Device Numbers

- Minor device numbers indicate the unit: which disk drive, which serial port, etc.
- In some cases, the minor device number has other flags encoded in the number
- Example: on Linux, the minor device for a SCSI disk is $16 \times \text{drivenum} + \text{partition}$

9 / 36

Device-Specific Semantics

- Not all devices are alike
- Some provision has to be made for device-specific semantics
- Example: skip a “file” on a tape
- Example: handle special characters from terminals: backspace, ^C, etc.
- Certain operations are handled by `ioctl` calls
- Tty processing is handled by a “line discipline” that is invoked by all tty-like devices

10 / 36

Device Driver Interface

- All devices have certain standard entry points: open, close, read, write, ioctl, etc.
- Devices implement Unix semantics as best they can
- Device-specific restrictions may, of course, be honored

11 / 36

Block Devices

- Also have a standardized interface, but it's quite different
- Again, primary function is read/write blocks on disk
- Integrated with buffer cache and memory subsystem
- Accessed primarily through the file system
- (Exception: some high-end database systems)
- Still need device-specific commands (see `hdparm` command on Linux)

12 / 36

Disk Scheduling

- Disk seeks are *slow*
- Rotational delay takes time, too
- Disk I/O driver must optimize throughput

13 / 36

Conflicting Demands

- Fairness: do I/O operations in order of arrival
- Efficiency: don't waste a lot of time seeking
- Promptness: don't delay some requests indefinitely

14 / 36

Two Not So Good Answers

- First come, first served — *very* inefficient
- Shortest seek first — unfair to requests at the ends of the disk; vulnerable to indefinite overtaking

15 / 36

Elevator Algorithm

- Move the arm in one direction at a time
- Implication: must sort I/O requests by sector
- First write in ascending sector order, then write in descending sector order
- Still has fairness issues — what if you get a lot of requests close to the current point? (Remember that file systems try to allocate blocks locally)

16 / 36

Order of Operations

- Recall that file system safey semantics demand that some blocks be written in a particular order
- This causes a discontinuity in the queue
- Besides, do you really know the disk geometry?
- The controller may implement its own version of this algorithm; the driver has to send it enough simultaneous requests that it can optimize

17 / 36

Rotational Delay

- Rotations take a while, too
- Scatter blocks around the disk to optimize for reading consecutive sectors
- Can be done at several layers: file system, device driver, controller
- Comments about disk geometry still apply

18 / 36

A Simple Model

- The screen is just an array of pixels
- Map each pixel to some set of bits or bytes; have software calculate the value of each one
- Tell the screen the address of the array

Too Slow!

- That algorithm couldn't keep up with even modest graphics demands
- It wouldn't come close to being fast enough for games
- Must have hardware assist

Hardware Assist

- User programs work in higher-level terms: characters, polygons, filled or shaded areas, curors, scrolling, etc.
- Each graphics controller is programmed — some use a standardized language, some a proprietary one
- Graphics processors are quite sophisticated (though specialized); they've been used for sorting and encryption
- OS must permit graphics driver to send high-level commands to device

21 / 36

Programming Interface

- Complex topic; out of scope for this class
- Always mediated through OS; screen is a shared resource
- Basic concept: the window
- Windows can be nested; application writes to a window are clipped at the window boundary
- The OS uses some screen area for itself (title bars, icons, etc.)

22 / 36

Network Devices

- Like disk drives, always shared among many processes
- However, no equivalent to directories or FATs
- Every input and output packet has to carry some identification (that's an oversimplification)

23 / 36

Network Output

- Output packets are relatively easy
- Mark packet depending on data from user's file descriptor
- Device drivers still have to adapt to local reality
- Ethernet: map 32-bit IP address to 48-bit Ethernet address
- Dial-up (PPP): must provide packet framing

24 / 36

Network Input

- Arbitrary incoming packets can be received
- Must use packet marking to associate packet with user process
- Different than disk: input is unsolicited
- Rate can be arbitarily high, and you can't slow it down

25 / 36

The Network Stack

- Network processing involves many layers that provide different functions
- OS has to provide connections between the layers
- Unix System V had file system entries for each network device and for each protocol layer, plus commands to do the plumbing
- Most other systems use a differnet (non-filesystem) name space for network devices

26 / 36

Real I/O Issues

- Device drivers have to cope with physical world quirks
- More specifically, device drivers have to map from a simplified, abstract model of the world to whatever the processor and device present
- Many, many strange things out there...

The Linux Printer Driver

Here are some comments from `drivers/char/lp.c`:

> The so called ‘buggy’ handshake is really the well documented compatibility mode IEEE1284 handshake. They changed the well known Centronics handshake acking in the middle of busy expecting to not break drivers or legacy application, while they broken linux lp until I fixed it reverse engineering the protocol by hand some month ago...

Interrupt Loads

- Normally, you start an I/O operation and return; an interrupt occurs when it's complete
- But processing I/O interrupts is expensive
- For high input rates, adapt dynamically; configure device to interrupt every nth input event
- For output, it may be cheaper to poll in a spin loop, then sleep — via a timer — if the output *FIFO* is full

29 / 36

Programmed I/O and DMA

- Simple, low-speed devices operate by *programmed I/O*: the CPU copies a byte at a time to the device
- Higher-speed devices use *Direct Memory Access* (DMA): the CPU passes the device an address and a count, and it copies the data directly from memory
- Sometimes called *cycle-stealing*: the device steals bus cycles from the CPU

30 / 36

Memory Access Issues

- Cache consistency
- Virtual memory
- Direct I/O from user space
- Address space size

Virtual Memory

- The kernel is dealing with (kernel) virtual addresses
- These do not correspond to physical addresses
- Contiguous virtual addresses are probably not contiguous in physical memory
- Some systems have an I/O map — the I/O bus manager has a (version of) the virtual memory map
- More often, the kernel has to translate the virtual addresses itself

Scatter/Gather I/O

- Suppose we're reading a single packet or disk block into two or more non-contiguous pages
- The I/O transfer has to use more than one ⟨address,length⟩ pair for that transfer to *scatter* the data around memory
- The same applies on output, where it has to be *gathered* from different physical pages

33 / 36

Direct I/O

- For efficiency, we may want to avoid copying data to/from user space
- Sometimes possible to do *direct I/O*
- Must consult user virtual memory map for translation
- Must lock pages in physical memory during I/O

34 / 36

Address Space Size

- Older I/O devices and buses may not be able to address all of system memory
- Example: ISA bus has a 16-bit address register: 64K bytes
- Solution: *bounce buffers*
- Do I/O to permissible memory — typically, permissible *physical* memory — then copy the page
- Usually, the page is copied, not remapped – we need that precious low memory space

35 / 36

Disk Block Address Space

- Others made similar mistakes: they didn't anticipate the growth of disk sizes
- Some historical disk size limits:

504M	maximum value for CHS
1.97G	12-bit cylinder size limit in BIOS
2G	Largest FAT16 disk
3.94G	With 16 heads, can't have more than 8K cylinders
7.38G	To work around the 8K cylinder limit, can't have more than 240 heads
7.88G	BIOS can't address more than 7.88G with CHS addressing; must use LBA
128G	disk spec uses 28-bit number for sector
2.2T	Windows XP limit

36 / 36