

I/O Systems

- In some ways, I/O may be viewed as the lowest-level OS function, just barely above the hardware
- In a way, we have followed a top-down sequence in studying the main functions of an OS: process management depends on memory management; memory management (particularly virtual memory) relies on secondary storage and/or the file system; finally, the file system depends on the I/O subsystem
- But, while it is very close to the hardware, an I/O subsystem does still have abstractions of its own

I/O Layers

- I/O delineates the boundary between a computer system and the physical world; thus, it makes sense that I/O starts with any *device* that takes some physical phenomenon and converts it into bits and back
- Many devices include a *controller*, which sends and receive commands and information from a *bus*
- A *device driver* abstracts the specifics for a device's controller from the operating system
- The OS then presents this abstraction to applications in the form of an *application programming interface (API)*

I/O Hardware

A number of I/O hardware concepts are common across computer and operating systems:

- Devices connect to a computer system at a specific, named point, called a *port*; as mentioned, data travels to/from this port along a shared *bus*
- Devices may connect to each other instead of directly to a computer; this is called a *daisy chain*
- Devices may have a *controller* or *host adapter* that coordinates activities between a port, a bus, and the computer system

- Software (primarily the *device driver*) may interact with the controller in two primary ways:
 - ◆ Using CPU-specific *I/O instructions* to read/write designated *I/O registers* that serve as bridges to the device controllers
 - ◆ Special areas of a system's address space may be designated as send/receive points for device data; this approach is called *memory-mapped I/O*
- The concept of an I/O port has four components:
 - ◆ *Data-in* and *data-out registers* transfer information from/to the device on a port
 - ◆ A *control register* receives commands for the device to perform, while a *status register* holds state information

Interacting with I/O Hardware

A number of established models exist for software interaction with I/O devices:

- *Polling* is relatively simple and efficient — software runs in a loop that repeatedly checks a device's status, then sends a command when the device becomes available
 - ◆ The exact values, registers, or sequence of events involved constitute a simple *protocol*, which in the I/O realm is specifically called *handshaking*
 - ◆ Polling is also called *busy-waiting*, since the CPU doesn't do anything else while waiting for a device to become ready
 - ◆ A single polling iteration takes around 3 CPU cycles (*read, compare, branch*) — not bad, but it becomes inappropriate when I/O waits can be long or unpredictable

- *Interrupts* provide an alternative to polling that frees up the CPU while a device is busy or working
 - ◆ The CPU can request an I/O activity then move on
 - ◆ The device *raises* an interrupt when I/O completes
 - ◆ An OS's *interrupt vector* therefore points to I/O-specific handlers
 - ◆ *Priorities* and *maskability* serve to stratify interrupts by their urgency or level of criticality

- Communication via designated I/O registers (commands, status, data) is called *programmed I/O*; a final model, *direct memory access (DMA)*, uses main memory instead of special registers
 - ◆ DMA works by having both the CPU and a device read from or write to main memory
 - ◆ A *DMA controller* implements a simple *request-acknowledge* handshaking protocol — the CPU sets up and designates the main memory address to use prior to sending an I/O request to the controller; when I/O is complete, an acknowledge signal is raised and the main memory address should now contain the result of the operation

I/O API

- At first glance, we may be quick to say “of course we should abstract I/O behind a uniform API” — and it’s true, this is precisely what device drivers do for the operating system and for application code
- However, it isn’t *quite* that easy, due to the huge diversity of available I/O devices
- Two general approaches to this: (1) define different APIs based on I/O device types or categories, and (2) define a *back door* API that facilitates direct communication with a device driver (e.g., Unix’s *ioctl()*)

Block vs. Character

- A significant variation among I/O devices is the granularity of data transfer — *block* devices read/write data in terms of fixed-size sets of bytes; *character* or *byte* devices transfer one character or byte at a time
- As seen, file system implementations are built upon anything that can be viewed as a block device
- For efficiency and convenience, *memory mapping* may be layered on top of a block device; layers can also go over character devices, depending on the specific byte stream (e.g., printers, human interface devices [HID])

Remote (Network) Devices

- Remote or network connections are sufficiently different from direct ones (e.g., performance, addressing, errors) to merit a distinct API
- The notion of a network device's hardware port (e.g., Ethernet, WiFi, Bluetooth) gets subsumed by a *socket* abstraction frequently consisting of an *address* and, confusingly, a *network port* number
- Many frameworks allow us to *stack* network APIs — enabling mix-and-match combinations such as IP over USB or Bluetooth via USB

Calling Models

- I/O APIs also have varying *calling models* or *styles*
- A *blocking* I/O call does not return until the I/O operation completes; *non-blocking* calls return right away with whatever data is available, with additional repeat calls providing more data as they arrive
- Blocking and non-blocking calls are *synchronous* — they do their work during the call; *asynchronous* calls return right away as with non-blocking calls, but they need only be called once, using a notification mechanism when the I/O operation is complete

Kernel I/O Issues

- All I/O activities, ranging from the installation and management of device drivers to the handling of application I/O requests through the public I/O API(s), require some involvement from the OS kernel
- One major issue is *I/O scheduling* — after all, all I/O calls go through the kernel, and at that level, the kernel can decide which call gets serviced first, depending on device speeds, latencies, etc.
- Proper scheduling also involves data structure management: wait queues, status tables, priorities

I/O Buffering

- Another major area handled by the kernel is *buffering* — instead of direct I/O communication, information is deposited in a kernel-managed memory area first
- This accommodates the vast diversity of I/O devices in terms of: (a) speed (e.g., transfer from a modem to a disk); and (b) transfer size (e.g., network packets vs. disk blocks or terminal character streams)
- Buffering also ensures *copy semantics* — after an I/O request is made, an application can reuse its data without munging the data involved in the I/O request

Caching and Spooling

- Like a buffer, a *cache* is a block of memory used for holding I/O data; unlike a buffer, caches only hold *copies* of the data involved — their purpose is speed, by substituting I/O activities for faster memory accesses
- A *spool* is another structure that resembles a buffer but is distinct from it: spools are used for I/O devices that can service only one request at a time, such as a printer — a spool is essentially a queue that holds pending I/O requests; these requests can typically be viewed or even removed before they reach the device

Error Handling and I/O Protection

- I/O is subject to errors, both transient and permanent — disconnects, hardware failures — so error handling and reporting is a key kernel issue
 - ◆ Transient errors, when possible, should be masked from the application, since they are transient after all: retries, error correction code, etc.
 - ◆ Permanent errors require a reporting mechanism; options include a return value, a global status variable (since I/O operations have their own return values), or traps
- *I/O protection* is also a source of errors — illegal direct hardware access, use of privileged instructions — at the same time, *some* “unshielded” access remains necessary, for performance reasons

Performance Considerations

- More than other OS functions, I/O involves precise coordination across a number of diverse components; this coordination potentially results in overhead that may unnecessarily drag down performance
- Assorted “tweaks” can be made to reduce context switches, data transfers, and busy waiting, or to increase concurrency and hardware primitives
- Generally, I/O starts in application code, then moves to the kernel, device drivers, and finally hardware as algorithms standardize and mature