

CSE380 - Operating Systems

Notes for Lecture 15 - 11/1/07

© Matt Blaze

(some examples by Insup Lee)

Communicating with Devices

- Modern architectures support convenient communication with devices
 - memory mapped I/O
 - ports via special I/O instructions
- Hardware can hide details and support efficient use of devices
 - interrupts
 - device controllers
 - DMA
- Main problems for OS:
 - protocol for communicating with device controllers
 - linking processes with the devices they are using

Kinds of devices

- Some devices are naturally associated with the OS itself
 - clock interrupt, etc,
 - OS problem: protocol for communicating w/ device
 - this is what we've looked at so far
- Others are naturally associated with some user process
 - serial ports, terminal, lab instruments, etc.
 - additional OS problems: associating process with device; relaying data between the device and the process
- Still others shared with many processes simultaneously
 - disks, network interfaces, etc.
 - additional OS problems: demultiplexing (e.g., file system abstraction); relaying data between device and the appropriate process

Devices and user processes

- *Naming* problem: how does a user process reference a given device?
- *Communication* problem: how does a user process send & received data to & from a given device? Also, blocking and unblocking process when I/O not ready
- *Control* problem: how can user process issue privileged commands to a device
- *Contention and multiplexing* problems: how does OS manage multiple processes access to a given device

Naming Devices

- Even with memory mapped I/O, it isn't usually reasonable to require processes to know (or hard-code) the addresses used for communicating with the various devices installed on a system
- Instead, many operating systems have a notion of a *device namespace* that processes use to refer to devices
 - System calls operate on devices in this namespace, similar to system calls operating on files
 - e.g., open them, etc.
- On Unix systems, the device namespace is integrated with the *file system* namespace
 - e.g., the first printer might be named `/dev/printer1`

User Process Communication with Devices

- Once a process tells the OS the name of a device it wants to use, how does it communicate with it?
- If the device uses memory-mapped I/O, the OS could map the device's addresses to a page in the process' virtual address space
 - e.g., a system call that takes the name of the device and returns a pointer to the mapped memory
 - advantage: no direct O/S involvement on each I/O
 - disadvantage: no chance for O/S involvement in I/O
- Or the process could treat the device like a file
 - syscall returns file descriptor, process does read, write, etc
 - assumes device can use file-style I/O
- Or the OS could have special system calls for communicating with a particular device

User Process Control of Devices

- Not all operations on devices can/should be available to user processes
 - e.g., installing interrupt handlers
- But some functions *need* to be made available
 - e.g., setting the bit rate on a serial interface
 - device interface to such operations is often esoteric
- Usual solution is to perform these functions through standardized system calls
 - device driver provides interface between control system call and device
 - Unix systems use the `ioctl` system call for this

Multiplexing, contention, scheduling devices

- Some devices might be used by more than one process at a time
 - e.g., disks (via file accesses)
- Some sequences of operations must be done atomically, others must be translated first
 - e.g., move disk head and write sector
- Solutions: Have all access go through a subsystem that translates requests, schedules I/O and enforces order
 - might be in OS kernel or in a user process
 - e.g., file system manages access to disks

Device drivers

- Device drivers are the interface between the OS and a device
 - OS usually provides a standard set of “hooks” for installing new device drivers
 - driver includes device-specific interrupt handlers
 - ... and device-specific I/O routines
 - ... and device-specific control routines (ioctl)
- Device drivers may be hard-coded part of OS kernel, dynamically loaded, or even part of a user process
 - depends on OS and kind of device

Example device: serial interface (“tty”)

- Asynchronous serial port (RS-232) is a standard communication interface to many kinds of peripheral equipment
 - modems, terminals, printers, lab equipment, etc
- Can send and receive bits at low-moderate speed (e.g., 28Kbps)
 - typical transfer unit is one byte (8 bits)
 - devices usually issue an interrupt when input comes in or output queue ready
- Many options and configurations
 - speed, parity bit, start/stop bits, flow control, etc.

Serial interface device drivers

- Read and write syscalls would be a natural interface for sending and receiving data
- Need input and output *buffers*:
 - interface is much slower than CPU
 - might still be sending last byte when process tries to send next one
 - data might arrive at any time
 - even though process hasn't done read yet
 - buffering avoids need for busy waiting and helps prevent lost data
- Also need some way to set options
 - e.g., speed, parity, etc

Serial interface in Unix

- Namespace in file system
 - e.g., `/dev/tty04`
- **open** of a tty returns a file descriptor
 - can do read & write with it, just like a file
- Device driver includes interrupt handler
 - device sends interrupt when new data arrives and when output queue available
- Options set by an **ioctl**
 - arguments include file descriptor and options
 - ioctl options determined by driver
 - man 4 tty, man 2 ioctl for details

Pseudo-terminals in Unix

- The serial interface is so useful that it's even used when there's no physical serial device
 - two process can be hooked together via a pseudo terminal; a write to one is available as a read to the other
 - as far as the processes are concerned they're talking to a serial port
- A “virtual” device driver relays bytes between pseudo terminals
- No actual hardware involved, but still uses device drivers

Next example: Disk Drives

- We've already talked about how a disk works
- Soon we'll talk about how file systems use disks
 - file system acts as the interface between most user processes and the disk devices
 - processes don't communicate directly with the disk driver
 - file system does disk scheduling and implements the system's file structure
- But right now, we'll talk about how the OS communicates with disks

Disk Access

- Disk interface: move head to location, read/write sector
 - addresses are complex
 - (cylinder number, head number, sector number)
 - sectors can be given “logical” numbers that get decoded by controller
- Driver provides simpler interface to processes
 - open, seek to byte #, read, write
 - provides view of disk as one big linear array of bytes
 - standard open, seek, read and write system calls work
- Design considerations for driver
 - what is the interface provided by the controller?
 - how to optimize performance