PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

File Systems I - Introduction

Baskin Engineering
UC SANTA CRUZ

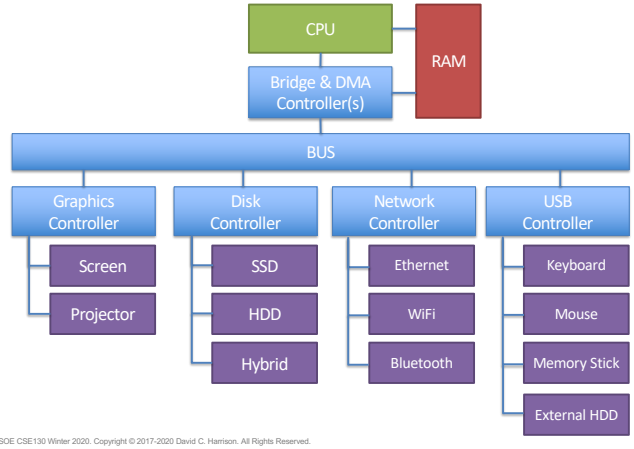
Notices

- **Lab 3** due **Sunday March 1**
- **Assignment 5** will be avaiable **Monday March 2**

Today's Lecture

- I/O Principles
- More Lab 3 Secret Sauce 

Generic Computer Schematic (simplified)



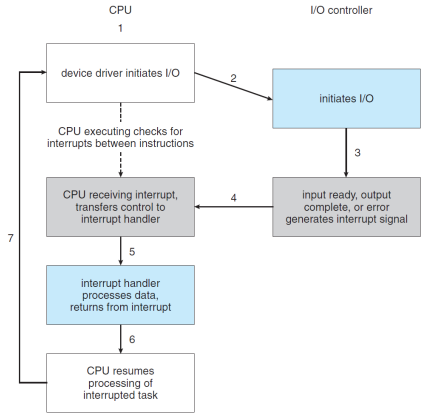
I/O Interface Basics

- **Interfaces**
  - I/O Instructions
  - Memory Mapped I/O
- **Interactions**
  - Polling
  - **Interrupts**
  - **Direct Memory Access (DMA)**
- **Device Drivers**
  - Device specific executable code
  - Executed by the kernel
  - Ensure correctly formatted instructions are placed on the bus
  - **Potential problems with device drivers?**

I/O Interactions : Polling

- **Status and command registers**
  - busy bit in status register
  - command-ready bit in command register
- **Handshake protocol**
  - Read busy bit from status register until 0
  - OS sets read or write bit - if write, copies data into data-out register
  - OS sets command-ready bit
  - Controller sets busy bit, executes transfer
  - Controller clears busy bit, error bit, command-ready bit when transfer done
- **Downsides?**
- **Does it matter if device is fast or slow?**

I/O Interactions : Interrupts



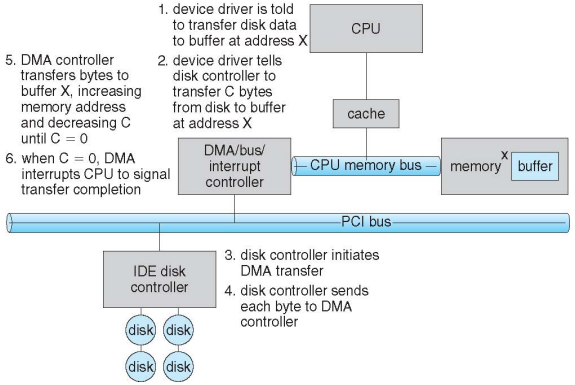
I/O Interactions : Interrupts

- Interrupt mechanism lets OS respond to **asynchronous events**
- **After executing each instruction, CPU checks to see if an interrupt has been raised**
  - If one has been raised, CPU executes the OS interrupt hander
- **Maskable** to ignore or delay certain interrupts
- Can be **prioritized**
- OS requires an **interrupt vector** to dispatch interrupt to correct interrupt handler
  - **You are extending the system call interrupt handler in Lab 3**

I/O Interactions : Direct Memory Access

- Used to **avoid programmed I/O** for large data movement
- Requires specialist hardware – the **DMA controller**
- **Bypasses CPU to transfer data directly between I/O device and memory**
- Once data is successfully stored in memory, raises an **interrupt**

The Six Stages of DMA

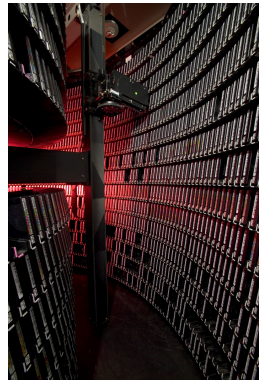


I/O Device Types

- Devices vary in many dimensions
  - Character-stream or block
  - Sequential or random-access
  - Synchronous or asynchronous (or both)
  - Sharable or dedicated
  - Speed of operation
  - Read-write, read only, or write only

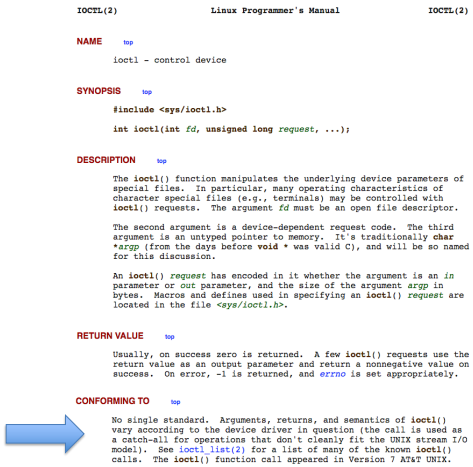
I/O Device Types : Examples

- **Disk:**
  - Character-stream or block?
- **Tape:**
  - Sequential or random-access?
  - Synchronous or asynchronous?
  - Sharable or dedicated?
- **CD-Rom or DVD-Rom:**
  - Read-write, read only, or write only?
- **Screen:**
  - Read-write, read only, or write only?



I/O Device Characteristics

- I/O devices typically grouped by OS into:
  - Block I/O
  - Character stream I/O
  - Memory-mapped file access
  - Network sockets
- For direct manipulation of I/O device specific characteristics, there is usually a *when-all-else-fails* access mechanism
  - Unix and Unix-like `ioctl()` enables an application to access any functionality that can be implemented by any device driver
  - **Potential Problems?**



IOCTL(2) Linux Programmer's Manual IOCTL(2)

NAME top

ioctl - control device

SYNOPSIS top

#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);

DESCRIPTION top

The ioctl() function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests. The argument fd must be an open file descriptor.

The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's traditionally char *argp (from the days before void * was valid C), and will be so named for this discussion.

An ioctl() request has encoded in it whether the argument is an in parameter or out parameter, and the size of the argument argp in bytes. Macros and defines used in specifying an ioctl() request are located in the file <sys/ioctl.h>.

RETURN VALUE top

Usually, on success zero is returned. A few ioctl() requests use the return value as an output parameter and return a nonnegative value on success. On error, -1 is returned, and errno is set appropriately.

CONFORMING TO top

No single standard. Arguments, returns, and semantics of ioctl() vary according to the device driver in question (the call is used as a catch-all for operations that don't cleanly fit the UNIX stream I/O model). See ioctl_list(2) for a list of many of the known ioctl() calls. The ioctl() function call appeared in Version 7 AT&T UNIX.

I/O Device Types : Block & Character

- **Block devices** include disk drives
  - Commands include read, write, seek
  - Raw I/O or file-system access
    Question: **What might we want to store on a "raw" disk partition?**
  - Memory-mapped file access possible
- **Character devices** include keyboards, mice, serial (USB) ports
  - Commands include `get`, `put`
  - Libraries layered on top allow line editing

I/O Device Types : Network

- Vary enough from block and character to have own interface
- Unix and Windows include **socket interface**
- Separates network protocol from network operation
- Approaches vary
  - Pipes
  - FIFOs
  - Streams
  - Queues
  - Mailboxes
  - etc.

I/O Device Types : Clocks

- Provide current time, elapsed time, timer
- If **programmable** - interval timer used for periodic interrupts **(Lab 1)**
- `ioctl()` covers odd aspects of I/O including clocks and timers

Blocking & Non-blocking I/O

- **Blocking** - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- **Non-blocking** - I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
    - Slave thread makes a blocking call
  - Returns quickly with count of bytes read or written
- **Asynchronous** - process runs while I/O executes
  - Difficult to use, but powerful
  - I/O subsystem signals process when I/O completed

The Kernel I/O Subsystem

- **Scheduling**
  - Some I/O request ordering via per-device queues
  - Any algorithm can be used, including round-robin, priority based, shortest-job-first
- **Buffering** (store data in memory while transferring between devices)
  - Deals with device speed and transfer size mismatched
  - Maintains "copy semantics" (prevent changes while copying the buffer)
  - Double buffering (two copies of the data, switch between the two)
  - Circular buffering
- **Caching** (faster device holds a copy of data)
  - Always just a copy
  - Key to performance
  - Sometimes combined with buffering
- **Spooling** (hold output for a device)
  - If device can serve only one request at a time
  - E.g. Printers
- **Device Reservation** (provides exclusive access to a device)
  - System calls for allocation and deallocation
  - Watch out for deadlock ☹

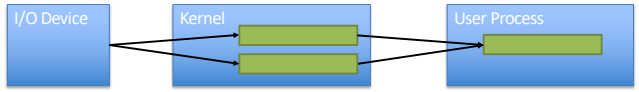
No Buffering

- Process must read/write from/to I/O device one byte at a time
- Each system call adds significant overhead
- Process must what until each I/O is complete
  - Blocking/waking adds to overhead
  - Many short runs of a process are inefficient
- **What happens if buffer is paged out to disk?**
  - Could lose data while buffer is paged in
  - Could lock buffer in memory (needed for DMA) but many processes doing I/O reduce RAM available for paging
  - Can cause deadlock as RAM is limited ☹

Single Buffering

- Simplest type of buffering
  - Operating system assigns a buffer in Kernel memory for an I/O request
- **Stream Oriented**
  - Used a line at time
  - User input from a terminal with carriage return signaling the end of the line
  - Output to the terminal is one line at a time
- or **Block-oriented**
  - Input transfers made to buffer in blocks

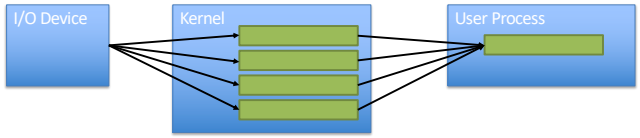
Double Buffering



- Use two system buffers instead of one
  - A process can transfer data to or from one buffer while the operating system empties or fills the other
  - Computation and Memory Copy can be done "simultaneously" with transfer, especially if DMA is available
- May be insufficient for "bursty" traffic
  - Intensive burst of writes between long periods of computation
  - Problem can often be alleviated by using more than two buffers

See class video for animation

Circular Buffering



- **Multiple buffers** used
- Each individual buffer is a component of a **circular buffer**
- Used when I/O operation must keep up with process
- Most useful type of buffering
  - User process blocks if kernel circular buffer is empty
  - As we've seen, trivial to implement with concurrency primitives ☺

See class video for animation

I/O Error Handling

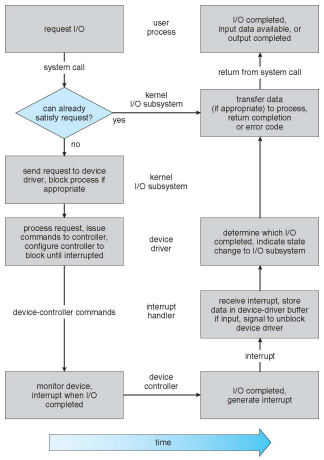
- OS must **recover from failures** including disk read, device unavailable, transient write failures
  - OS can retry a failed read or write
  - Also track error frequencies
    - Stop using (i.e. black-list) devices with increasing frequency of retry-able errors
- Most devices return an **error code** when I/O request fails
- System error logs hold problem reports

Kernel I/O Data Structures

- Kernel keeps state info for I/O components, including ***open file tables***, network connections, character device state
  - **You need to store a per-thread open file table in Lab 3**
- Many complex data structures to track buffers, memory allocation, "dirty" blocks, etc. etc.

Handling I/O Requests

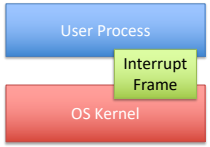
- Consider a process that wants to read a file from disk:
- Kernel has to:
  - Determine device holding file
  - Translate file name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process
  - Requires many expensive context switches ☹
- **I/O a major factor in system performance**, or lack thereof:
  - Demands CPU execute device driver and kernel I/O code
  - Multiple context switches due to interrupts
  - Data copy latency
  - Network traffic especially stressful ☹



Secret Sauce

The Interrupt Frame : "IF"

- In part 1 you (the operating system) populated a specific memory location (the stack) so a user program can get it's arguments when it executes
- In part 2 you (again, the operating system) take from a specific memory location the arguments to a system call, and return to the same location the result of the system call
- This new memory location is called **the interrupt frame** because it gets populated whenever a system call is made and, as we've seen, for the CPU to switch from a user process back to the operating system, an interrupt must be raised
- This is a special case of a general technique known as **shared memory inter-process communication** (SMIPC) similar to Assignment 1



Communication via the Interrupt Frame

User Level Code

```
                // defined in a header file somewhere
(4) (3) (2)     bool create(char *fname, int isize);

                int main(int argc, char *argv[])
                {
(1)                 bool ok = create("fred", 1024);
                }
                                                  (11)
```

(1) User code invokes `create()` system call
(2) `create()` puts its unique identifier `SYS_CREATE` and the two arguments `fname` and `isize` into the IF (in that order) and makes space in the IF for the return value
(3) `create()` then sets the CPU interrupt register that indicates a system call has been made
(4) CPU sees the interrupt register has been set and a context switch occurs - the user process is kicked off the CPU and the OS gets back on
(5) The OS checks what kind of interrupt was raised, sees it was the system call interrupt and calls the system call interrupt handler
(6) The system call interrupt handler checks which system call it was by checking the first 4 bytes in the IF and calls the appropriate system call specific handler
(7) The create system call handler extracts the two arguments from the IF (in the order in which they were placed there) and passes them to an internal delegate to actually create the file
(8) The internal delegate attempts to create the file and returns `true` or `false` depending on whether it was successful
(9) The create system call handler puts the internal delegate's return value in IF
(10) System call handler exits which causes the CPU to context switch the OS out and the user process is put back on the CPU
(11) `create()` gets the return value from the IF and returns it to the user code

Kernel Level Code

```
                // signature is exact duplicate of user visible
                // create() system call
                static bool sys_create(char *fname, int isize)
                {
(8)               // whatever
                  return true or false;
                }

                static void create_handler(struct intr_frame *f)
                {
                  const char* fname;
                  int isize;

(7)               umem_read(f->esp + 4, &fname, sizeof(fname));
                  umem_read(f->esp + 8, &isize, sizeof(isize));
(9)
                  f->eax = sys_create(fname, isize);
(10)            }
(5)             static void syscall_handler(struct intr_frame *f)
                {
                  int syscall;
                  umem_read(f->esp, &syscall, sizeof(syscall));
                  thread_current()->current_esp = f->esp;

                  switch (syscall) {
                  ...
(6)               case SYS_CREATE:
                    create_handler(f);
                    break;
                  ...
                  }
                }
```


The Per-Process Open File Table

- Pintos System Calls:

```
int open(const char *file); // returns a file descriptor
int read(int fd, void *buffer, unsigned length);
int write(int fd, const void *buffer, unsigned length);
```

- Pintos File System:

```
struct file *filesys_open(const char *name);
off_t file_read(struct file *, void *, off_t);
off_t file_write(struct file *, const void *, off_t);
```

- You need to:
  - Map numeric file descriptors to file pointers
  - Cannot use file descriptors 0, 1, or 2 (`stdin`, `stdout`, and `stderr`)
  - Where are you going to store the per process open file table?

Next Lecture

- File System Basics
- The Layered File System
- Virtual File Systems