Logistics

- Keep working hard on Project 2
  - Help desk: Monday 12-2pm, Tuesday 12-2pm
  - Help desk: Wednesday 1-3pm
- Office hours on Wednesday from 12-1pm
- Project 2 discussion at end of lecture
Outline

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- STREAMS
- Performance
Objectives

- Explore the structure of an OS I/O subsystem

- Discuss the principles of I/O hardware and its complexity

- Provide details of the performance aspects of I/O hardware and software
Overview

- I/O management is a major component of operating system design and operation
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent

- Ports, busses, device controllers connect to various devices

- Device drivers encapsulate device details
  - Present uniform device-access interface to I/O subsystem
I/O Hardware (1)

- Incredible variety of I/O devices
  - Storage
  - Transmission
  - Human-interface

- Common concepts
  - Signals from I/O devices interface with computer
  - Port – connection point for device
  - Bus – daisy chain or shared direct access
    - PCI bus common in PCs and servers, PCI Express (PCIe)
    - expansion bus connects relatively slow devices
  - Controller – electronics to operate port, bus, device
    - sometimes integrated
    - sometimes separate circuit board (host adapter)
    - contains processor, microcode, private memory, bus controller, etc
A Typical PC Bus Structure

- Monitor
- Graphics controller
- Processor
- Bridge/memory controller
- Cache
- Memory
- SCSI controller
- Disks
- IDE disk controller
- Expansion bus interface
- Expansion bus
- Keyboard
- Parallel port
- Serial port
- PCI bus
I/O Hardware (2)

- I/O instructions control devices
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - Data-in / data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- Devices have addresses, used by
  - Direct I/O instructions
  - Memory-mapped I/O
    - device data and command registers mapped to processor address space
    - especially for large address spaces (graphics)
### Device I/O Port Locations on PCs (partial)

<table>
<thead>
<tr>
<th>I/O address range (hexadecimal)</th>
<th>device</th>
</tr>
</thead>
<tbody>
<tr>
<td>000–00F</td>
<td>DMA controller</td>
</tr>
<tr>
<td>020–021</td>
<td>interrupt controller</td>
</tr>
<tr>
<td>040–043</td>
<td>timer</td>
</tr>
<tr>
<td>200–20F</td>
<td>game controller</td>
</tr>
<tr>
<td>2F8–2FF</td>
<td>serial port (secondary)</td>
</tr>
<tr>
<td>320–32F</td>
<td>hard-disk controller</td>
</tr>
<tr>
<td>378–37F</td>
<td>parallel port</td>
</tr>
<tr>
<td>3D0–3DF</td>
<td>graphics controller</td>
</tr>
<tr>
<td>3F0–3F7</td>
<td>diskette-drive controller</td>
</tr>
<tr>
<td>3F8–3FF</td>
<td>serial port (primary)</td>
</tr>
</tbody>
</table>
Polling

- For each byte of I/O
  - Read busy bit from status register until 0
  - Host sets read or write bit
    - if write copies data into data-out register
  - Host sets command-ready bit
  - Controller sets busy bit, executes transfer
  - Controller clears busy bit, error bit, command-ready bit when transfer done

- Step 1 is *busy-wait* cycle to wait for I/O from device
  - Reasonable if device is fast
  - But inefficient if device slow
  - CPU switches to other tasks?
    - but if miss a cycle data overwritten / lost
Interrupts (1)

- Polling can happen in 3 instruction cycles
  - Read status, logical-and to extract status bit, branch if not zero
  - How to be more efficient if non-zero infrequently?
  - How to avoid the overhead of polling?

- CPU *interrupt-request line* triggered by I/O device
  - Checked by processor after each instruction

- *Interrupt handler* receives interrupts
  - *Maskable* to ignore or delay some interrupts

- *Interrupt vector* to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some *nonmaskable*
  - Interrupt chaining if more than one device at same interrupt number
Interrupt-Driven I/O Cycle

1. CPU
   - device driver initiates I/O

2. I/O controller
   - initiates I/O

3. CPU executes checks for interrupts between instructions

4. CPU receiving interrupt, transfers control to interrupt handler
   - interrupt handler processes data, returns from interrupt

5. CPU resumes processing of interrupted task

6. input ready, output complete, or error generates interrupt signal
## Intel Pentium Processor Event-Vector Table

<table>
<thead>
<tr>
<th>vector number</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>divide error</td>
</tr>
<tr>
<td>1</td>
<td>debug exception</td>
</tr>
<tr>
<td>2</td>
<td>null interrupt</td>
</tr>
<tr>
<td>3</td>
<td>breakpoint</td>
</tr>
<tr>
<td>4</td>
<td>INTO-detected overflow</td>
</tr>
<tr>
<td>5</td>
<td>bound range exception</td>
</tr>
<tr>
<td>6</td>
<td>invalid opcode</td>
</tr>
<tr>
<td>7</td>
<td>device not available</td>
</tr>
<tr>
<td>8</td>
<td>double fault</td>
</tr>
<tr>
<td>9</td>
<td>coprocessor segment overrun (reserved)</td>
</tr>
<tr>
<td>10</td>
<td>invalid task state segment</td>
</tr>
<tr>
<td>11</td>
<td>segment not present</td>
</tr>
<tr>
<td>12</td>
<td>stack fault</td>
</tr>
<tr>
<td>13</td>
<td>general protection</td>
</tr>
<tr>
<td>14</td>
<td>page fault</td>
</tr>
<tr>
<td>15</td>
<td>(Intel reserved, do not use)</td>
</tr>
<tr>
<td>16</td>
<td>floating-point error</td>
</tr>
<tr>
<td>17</td>
<td>alignment check</td>
</tr>
<tr>
<td>18</td>
<td>machine check</td>
</tr>
<tr>
<td>19–31</td>
<td>(Intel reserved, do not use)</td>
</tr>
<tr>
<td>32–255</td>
<td>maskable interrupts</td>
</tr>
</tbody>
</table>
Interrupts (2)

- Interrupt mechanism also used for exceptions
  - Terminate process, crash system due to hardware error
- Page fault executes when memory access error
- System call executes via trap to trigger kernel to execute request
- Multi-CPU systems can process interrupts concurrently
  - If operating system designed to handle it
- Used for time-sensitive processing, frequent, must be fast
Direct Memory Access (DMA)

- Used to avoid *programmed I/O* (one byte at a time) for large data movement
- Requires DMA controller
- Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location of command block to DMA controller
  - Bus mastering of DMA controller – grabs bus from CPU
    - *cycle stealing* from CPU but still much more efficient
  - When done, interrupts to signal completion
- Version that is aware of virtual addresses can be even more efficient
  - Direct Virtual Memory Access (DVMA)
Six Step Process to Perform DMA Transfer

1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- 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
# Characteristics of I/O Devices (1)

<table>
<thead>
<tr>
<th>aspect</th>
<th>variation</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-transfer mode</td>
<td>character block</td>
<td>terminal disk</td>
</tr>
<tr>
<td>access method</td>
<td>sequential random</td>
<td>modem CD-ROM</td>
</tr>
<tr>
<td>transfer schedule</td>
<td>synchronous</td>
<td>tape</td>
</tr>
<tr>
<td></td>
<td>asynchronous</td>
<td>keyboard</td>
</tr>
<tr>
<td>sharing</td>
<td>dedicated</td>
<td>tape</td>
</tr>
<tr>
<td></td>
<td>sharable</td>
<td>keyboard</td>
</tr>
<tr>
<td>device speed</td>
<td>latency</td>
<td></td>
</tr>
<tr>
<td></td>
<td>seek time</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transfer rate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delay between</td>
<td></td>
</tr>
<tr>
<td></td>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>I/O direction</td>
<td>read only</td>
<td>CD-ROM graphics controller</td>
</tr>
<tr>
<td></td>
<td>write only</td>
<td>disk</td>
</tr>
<tr>
<td></td>
<td>read–write</td>
<td></td>
</tr>
</tbody>
</table>
Characteristics of I/O Devices (2)

- Subtleties of devices handled by device drivers
- Broadly I/O devices can be grouped by the OS into
  - Block I/O
  - Character I/O (Stream)
  - Memory-mapped file access
  - Network sockets
- For direct manipulation of I/O device specific characteristics, usually an escape / back door
  - Unix ioctl() call to send arbitrary bits to a device control register and data to device data register
Block and Character Devices

- Block devices include disk drives
  - Commands include `read()`, `write()`, `seek()`
  - Raw I/O, direct I/O, or file-system access
  - Memory-mapped file access possible
    - File mapped to virtual memory and clusters brought via demand paging
  - DMA

- Character devices include keyboards, mice, serial ports
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
Network Devices

- Varying enough from block and character to have own interface
  - Many different varieties of network devices

- Linux, Unix, Windows and many others include socket interface
  - Separates network protocol from network operation
  - Includes `select()` functionality to multiplex requests among multiple sockets

- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)
Clocks and Timers

- Provide current time, elapsed time, timer
- Normal resolution about 1/60 second
- Some systems provide higher-resolution timers
- Programmable interval timer used for timings, periodic interrupts
- `ioctl()` (on UNIX) covers odd aspects of I/O such as clocks and timers
Nonblocking and Asynchronous I/O

- **Blocking** – process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs

- **Nonblocking** – I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
  - `select()` to find if data ready, `read()` or `write()` to transfer

- **Asynchronous** – process runs while I/O executes
  - More tricky to use
  - I/O subsystem signals process when I/O completed
Two I/O Methods

Synchronous

Asynchronous
Vectored I/O

- Vectored I/O allows one system call to perform multiple I/O operations
- For example, Unix `readve()` accepts a vector of multiple buffers to read into or write from
- This scatter-gather method better than multiple individual I/O calls
  - Decreases context switching and system call overhead
  - Some versions provide atomicity
    - avoid for example worry about multiple threads changing data as reads / writes occurring
Kernel I/O Subsystem

- Scheduling
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
  - Some implement Quality Of Service (i.e. IPQOS)

- Buffering – store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain “copy semantics”
  - Double buffering – two copies of the data
    - kernel and user
    - varying sizes
    - full / being processed and not-full / being used
    - copy-on-write can be used for efficiency in some cases
### Device-status Table

<table>
<thead>
<tr>
<th>Device</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyboard</td>
<td>idle</td>
</tr>
<tr>
<td>laser printer</td>
<td>busy</td>
</tr>
<tr>
<td>mouse</td>
<td>idle</td>
</tr>
<tr>
<td>disk unit 1</td>
<td>idle</td>
</tr>
<tr>
<td>disk unit 2</td>
<td>busy</td>
</tr>
</tbody>
</table>

**Request for Laser Printer**
- Address: 38546
- Length: 1372

**Request for Disk Unit 2**
- File: `xxx`
- Operation: read
- Address: 43046
- Length: 20000

**Request for Disk Unit 2**
- File: `yyy`
- Operation: write
- Address: 03458
- Length: 500
Sun Enterprise 6000 Device-Transfer Rates

- System bus
- HyperTransport (32-pair)
- PCI Express 2.0 (×32)
- Infiniband (QDR 12X)
- Serial ATA (SATA-300)
- Gigabit Ethernet
- SCSI bus
- FireWire
- Hard disk
- Modem
- Mouse
- Keyboard
Kernel I/O Subsystem

- **Caching** – faster device holding 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
  - Print spooling is a good example

- **Device reservation** – provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock
Error Handling

- OS can recover from disk read, device unavailable, transient write failures
  - Retry a read or write, for example
  - Some systems more advanced
    - Track error frequencies, stop using device with increasing frequency of retry-able errors

- Most return an error number or code when I/O request fails

- System error logs hold problem reports
I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
    - memory-mapped I/O must be protected
    - I/O port memory locations must be protected too
Use of a System Call to Perform I/O

1. trap to monitor
2. perform I/O
3. return to user
Kernel Data Structures

- Kernel keeps state info for I/O components
  - Includes open file tables, network connections, character device state

- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks, …

- Some use object-oriented methods and message passing to implement I/O
  - Windows uses message passing
    - message with I/O information passed from user mode into kernel
    - message modified as it flows through to device driver and back to process
UNIX I/O Kernel Structure

- **System-wide open-file table**
  - File-system record
    - Inode pointer
    - Pointer to read and write functions
    - Pointer to select function
    - Pointer to ioctl function
    - Pointer to close function
  - Networking (socket) record
    - Pointer to network info
    - Pointer to read and write functions
    - Pointer to select function
    - Pointer to ioctl function
    - Pointer to close function

- **Active-inode table**
- **Network-information table**
Power Management

- Not strictly domain of I/O, but much is I/O related
- Computers and devices use electricity, generate heat, frequently require cooling
- OSes can help manage and improve use
  - Cloud computing environments move virtual machines between servers
    - Can end up evacuating whole systems and shutting them down
- Mobile computing has power management as first class OS aspect
Android Power Management

- Component-level power management
  - Understands relationship between components
  - Build device tree representing physical device topology
  - System bus $\rightarrow$ I/O subsystem $\rightarrow$ \{flash, USB storage\}
  - Device driver tracks state of device, whether in use
  - Unused component – turn it off
  - All devices in tree branch unused – turn off branch

- Wake locks – like other locks but prevent sleep of device when lock is held

- Power collapse – put a device into very deep sleep
  - Marginal power use
  - Only awake enough to respond to external stimuli (button press, incoming call)
I/O Requests to Hardware Operations

- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process
Life Cycle of An I/O Request

1. User process requests I/O.
2. System call is made.
3. Kernel I/O subsystem checks if request can already be satisfied.
   - If yes, data is transferred, return completion or error code.
   - If no, proceed to the next step.
4. Send request to device driver, block process if appropriate.
5. Process request, issue commands to controller, configure controller to block until interrupted.
6. Device controller commands are sent.
7. Monitor device, interrupt when I/O completed.
8. Receive interrupt, store data in device-driver buffer if input, signal to unblock device driver.
9. Determine which I/O completed, indicate state change to I/O subsystem.
10. I/O completed, generate interrupt.
STREAMS

- *STREAM* – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond

- A STREAM consists of:
  - STREAM head interfaces with the user process
  - driver end interfaces with the device
  - zero or more STREAM modules between them

- Each module contains a *read queue* and a *write queue*

- Message passing is used to communicate between queues
  - Flow control option to indicate available or busy

- Asynchronous internally, synchronous where user process communicates with stream head
The STREAMS Structure

user process

stream head

read queue | write queue

modules

read queue | write queue

read queue | write queue

read queue | write queue

driver end

device
I/O a major factor in system performance:

- Demands CPU to execute device driver, kernel I/O code
- Context switches due to interrupts
- Data copying
- Network traffic especially stressful
Intercomputer Communications

Lecture 14 – I/O Systems

Character typed

Sending system

Kernel

Device driver

User process

Context switch

Interrupt generated

State save

Interrupt handled

System call completes

Interrupt generated

Device driver

Kernel

User process

Context switch

Receiving system

Network subdaemon

Network daemon

Kernel

Context switch

Network adapter

Device driver

Kernel

Context switch

Network adapter

Device driver

Kernel

Network adapter

Interrupt generated

State save

Interrupt handled

System call completes

Character typed

Sending system

Kernel

Device driver

User process

Context switch

Receiving system

Network subdaemon

Network daemon

Kernel

Context switch

Network adapter

Device driver

Kernel

Network adapter

Interrupt generated

State save

Interrupt handled

System call completes

Character typed

Sending system

Kernel

Device driver

User process

Context switch

Receiving system

Network subdaemon

Network daemon

Kernel

Context switch

Network adapter

Device driver

Kernel

Network adapter

Interrupt generated

State save

Interrupt handled

System call completes

Character typed

Sending system

Kernel

Device driver

User process

Context switch

Receiving system

Network subdaemon

Network daemon

Kernel

Context switch

Network adapter

Device driver

Kernel

Network adapter

Interrupt generated

State save

Interrupt handled

System call completes

Character typed

Sending system

Kernel

Device driver

User process

Context switch

Receiving system

Network subdaemon

Network daemon

Kernel

Context switch

Network adapter

Device driver

Kernel

Network adapter

Interrupt generated

State save

Interrupt handled

System call completes

Character typed

Sending system

Kernel

Device driver

User process

Context switch

Receiving system

Network subdaemon

Network daemon

Kernel

Context switch

Network adapter

Device driver

Kernel

Network adapter

Interrupt generated

State save

Interrupt handled

System call completes

Character typed

Sending system

Kernel

Device driver

User process

Context switch

Receiving system

Network subdaemon

Network daemon

Kernel

Context switch

Network adapter

Device driver

Kernel

Network adapter

Interrupt generated

State save

Interrupt handled

System call completes

Character typed

Sending system

Kernel

Device driver

User process

Context switch
Improving Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput
- Move user-mode processes / daemons to kernel threads
Device-Functionality Progression

- New algorithm
  - Application code
    - Kernel code
      - Device-driver code
        - Device-controller code (hardware)
          - Device code (hardware)

- Increased efficiency
- Increased development cost
- Increased abstraction
- Increased time (generations)
- Increased flexibility
Project 2

- Testing
  - Need to validate that your code is working
  - Suggest publishing articles that look like:
    “publisher: pubid, article: a, topic: topicid”
    where $a$ is a monotonically increasing number starting at 0
  - Subscriber then prints out the entire article string
- Please provide a makefile and a README file
- You can use C++, but you are not allowed to use C++ libraries that already include multi-threaded synchronization for shared data structures
- Grading
  - Part 1  15%
  - Part 2  20%
  - Part 3  20%
  - Part 4  20%
  - Part 5  15%
  - Part 6  10%