Device Input/Output
Device Terminology

Device (mechanical hardware)
Device controller (electrical hardware)
Device driver (software)
Devices & Controllers

Components of a simple personal computer
Device Controllers

The Device vs. its Controller

Some duties of a device controller:
- Interface between CPU and the Device
- Start/Stop device activity
- Convert serial bit stream to a block of bytes
- Deal with error detection/correction
- Move data to/from main memory

Some controllers may handle several (similar) devices
Communication With Devices

Hardware supports I/O ports or memory mapped I/O for accessing device controller registers and buffers.
I/O Ports

Each port has a separate number.

CPU has special I/O instructions

\[
\begin{align*}
\text{in} & \quad \text{r4,3} \\
\text{out} & \quad 3,\text{r4}
\end{align*}
\]

Port numbers form an “address space”... separate from main memory

Contrast with

\[
\begin{align*}
\text{load} & \quad \text{r4,3} \\
\text{store} & \quad 3,\text{r4}
\end{align*}
\]
Memory-Mapped I/O

One address space for
main memory
I/O devices

CPU has no special instructions
load    r4,addr
store   addr,r4

I/O devices are “mapped” into
very high addresses
# I/O Device Speed

<table>
<thead>
<tr>
<th>Device</th>
<th>Data rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>10 bytes/sec</td>
</tr>
<tr>
<td>Mouse</td>
<td>100 bytes/sec</td>
</tr>
<tr>
<td>56K modem</td>
<td>7 KB/sec</td>
</tr>
<tr>
<td>Telephone channel</td>
<td>8 KB/sec</td>
</tr>
<tr>
<td>Dual ISDN lines</td>
<td>16 KB/sec</td>
</tr>
<tr>
<td>Laser printer</td>
<td>100 KB/sec</td>
</tr>
<tr>
<td>Scanner</td>
<td>400 KB/sec</td>
</tr>
<tr>
<td>Classic Ethernet</td>
<td>1.25 MB/sec</td>
</tr>
<tr>
<td>USB (Universal Serial Bus)</td>
<td>1.5 MB/sec</td>
</tr>
<tr>
<td>Digital camcorder</td>
<td>4 MB/sec</td>
</tr>
<tr>
<td>IDE disk</td>
<td>5 MB/sec</td>
</tr>
<tr>
<td>40x CD-ROM</td>
<td>6 MB/sec</td>
</tr>
<tr>
<td>Fast Ethernet</td>
<td>12.5 MB/sec</td>
</tr>
<tr>
<td>ISA bus</td>
<td>16.7 MB/sec</td>
</tr>
<tr>
<td>EIDE (ATA-2) disk</td>
<td>16.7 MB/sec</td>
</tr>
<tr>
<td>FireWire (IEEE 1394)</td>
<td>50 MB/sec</td>
</tr>
<tr>
<td>XGA Monitor</td>
<td>60 MB/sec</td>
</tr>
<tr>
<td>SONET OC-12 network</td>
<td>78 MB/sec</td>
</tr>
<tr>
<td>SCSI Ultra 2 disk</td>
<td>80 MB/sec</td>
</tr>
<tr>
<td>Gigabit Ethernet</td>
<td>125 MB/sec</td>
</tr>
<tr>
<td>Ultrium tape</td>
<td>320 MB/sec</td>
</tr>
<tr>
<td>PCI bus</td>
<td>528 MB/sec</td>
</tr>
<tr>
<td>Sun Gigaplane XB backplane</td>
<td>20 GB/sec</td>
</tr>
</tbody>
</table>
Hardware Performance Challenges

How to prevent slow devices from slowing down memory due to bus contention
  - What is bus contention?

How to access I/O addresses without interfering with memory performance
Dual Bus Architecture

CPU reads and writes of memory go over this high-bandwidth bus

CPU
Memory
I/O

All addresses (memory and I/O) go here

Bus

This memory port is to allow I/O devices access to memory
Pentium Bus Architecture
Software Performance Challenges

How to prevent CPU throughput from being limited by I/O device speed (for slow devices)
  - Why would slow devices affect the CPU?

How to prevent I/O throughput from being limited by CPU speed (for fast devices)
  - Why would device throughput be limited by the CPU?

How to achieve good utilization of CPU and I/O devices

How to meet the real-time requirements of devices
Programmed I/O

Steps in printing a string
Programmed I/O

Example: Writing a string to a serial output or printing a string

CopyFromUser(virtAddr, kernelBuffer, byteCount)
for i = 0 to byteCount-1
    while *serialStatusReg != READY
        endwhile
    *serialDataReg = kernelBuffer[i]
endFor
return

Called “Busy Waiting” or “Polling”
Problem: CPU is continually busy working on I/O!
Interrupt-Driven I/O

Getting the I/O started:

CopyFromUser(virtAddr, kernelBuffer, byteCount)
EnableInterrupts()
while *serialStatusReg != READY
endWhile
*serialDataReg = kernelBuffer[0]
Sleep()

The Interrupt Handler:

if i == byteCount
    Wake up the user process
else
    *serialDataReg = kernelBuffer[i]
    i = i + 1
endIf
Return from interrupt
Hardware Support For Interrupts

How interrupts happen. Connections between devices and interrupt controller actually use interrupt lines on the bus rather than dedicated wires.
Interrupt Driven I/O Problem

Problem:
- CPU is still involved in every data transfer
- Interrupt handling overhead is high
- Overhead cost is not amortized over much data
- Overhead is too high for fast devices
  - Gbps networks
  - Disk drives
Direct Memory Access (DMA)

Data transferred from device straight to/from memory

CPU not involved

**The DMA controller:**
- Does the work of moving the data
- CPU sets up the DMA controller ("programs it")
- CPU continues
- The DMA controller moves the bytes
Sending Data Using DMA

Getting the I/O started:

CopyFromUser(virtAddr, kernelBuffer, byteCount)
Set up DMA controller
Sleep()

The Interrupt Handler:

Acknowledge interrupt
Wake up the user process
Return from interrupt
Direct Memory Access (DMA)

1. CPU programs the DMA controller
2. DMA requests transfer to memory
3. Data transferred
4. Ack

CPU

DMA controller

Address
Count
Control

Disk controller

Buffer

Drive

Main memory

Interrupt when done

Main memory

Bus
Direct Memory Access (DMA)

Cycle Stealing
- DMA Controller acquires control of bus
- Transfers a single byte (or word)
- Releases the bus
- The CPU is slowed down due to bus contention

Burst Mode
- DMA Controller acquires control of bus
- Transfers all the data
- Releases the bus
- The CPU operation is temporarily suspended
Principles of I/O Software

Device Independence
- Programs can access any I/O device
  Hard Drive, CD-ROM, Floppy,...
  ... without specifying the device in advance

Uniform Naming
- Devices / Files are named with simple strings
- Names should not depend on the device

Error Handling
- Should be as close to the hardware as possible because its often device-specific
Principles of I/O Software

Synchronous vs. Asynchronous Transfers
- Process is blocked vs. interrupt-driven or polling approaches

Buffering
- Data comes off a device
- May not know the final destination of the data
e.g., a network packet... Where to put it???

Sharable vs. Dedicated Devices
- Disk should be sharable
- Keyboard, Screen dedicated to one process
Software Engineering Challenges

How to remove the complexities of I/O handling from application programs
  - Standard I/O APIs (libraries and system calls)

How to support a wide range of device types on a wide range of operating systems
  - Standard interfaces for device drivers (DDI)
  - Standard/published interfaces for access to kernel facilities (DKI)
I/O Software Layers

- User-level I/O software
- Device-independent operating system software
- Device drivers
- Interrupt handlers
- Hardware
I/O Software Layers

User-level I/O software

Device-independent operating system software

Device drivers

Interrupt handlers

Hardware
Interrupt Handling

I/O Device Driver starts the operation
- Then blocks until an interrupt occurs
- Then it wakes up, finishes, & returns

The Interrupt Handler
- Does whatever is immediately necessary
- Then unblocks the driver

Example: The BLITZ “DiskDriver”
- Start I/O and block (waits on semaphore)
- Interrupt routine signals the semaphore & returns
Interrupt handlers are divided into *scheduled* and *non scheduled* tasks

Non-scheduled tasks execute immediately on interrupt and run in the context of the interrupted thread
- Ie. There is no VM context switch
- They should do a minimum amount of work so as not to disrupt progress of interrupted thread
- They should minimize time during which interrupts are disabled

Scheduled tasks are queued for processing by a thread
- This thread will be scheduled to run later
- May be scheduled preemptively or nonpreemptively
Interrupt Handler’s Jobs

Set up stack for interrupt service procedure
Ack interrupt controller, reenable interrupts
Copy registers from where saved
Run service procedure
I/O Software Layers

- User-level I/O software
- Device-independent operating system software
- Device drivers
- Interrupt handlers
- Hardware
Device drivers in kernel space
Device Drivers

Device drivers are device-specific software that connects devices with the operating system

- Typically an assembly-level job
  - Must deal with hardware-specific details
  - Must deal with O.S. specific details
- Goal: hide as many device-specific details as possible from higher level software

Device drivers are typically given kernel privileges for efficiency

- Bugs can bring down the O.S.!
- Open challenge: how to provide efficiency and safety?
I/O Software Layers

- User-level I/O software
  - Device-independent operating system software
  - Device drivers
  - Interrupt handlers
- Hardware
Device-Independent I/O Software

Functions and responsibilities
- Uniform interfacing for device drivers
- Buffering
- Error reporting
- Allocating and releasing dedicated devices
- Providing a device-independent block size
Device-Independent I/O Buffering

(a) Unbuffered input
(b) Buffering in user space
(c) Buffering in the kernel followed by copying to user space
(d) Double buffering in the kernel
Copying Overhead in Network I/O

Networking may involve many copies
Before mounting,
- files on floppy are inaccessible

After mounting floppy on b,
- files on floppy are part of file hierarchy
I/O Software Layers

- User-level I/O software
- Device-independent operating system software
- Device drivers
- Interrupt handlers
- Hardware
User-Space I/O Software

In user’s (C) program

```c
  count = write (fd, buffer, nbytes);
  printf ("The value of %s is %d\n", str, i);
```

Linked with library routines.

The library routines contain:
- Lots of code
- Buffering
- The syscall to trap into the kernel
Communicating Across I/O Layers

- **User processes**: Make I/O call; format I/O; spooling
- **Device-independent software**: Naming, protection, blocking, buffering, allocation
- **Device drivers**: Set up device registers; check status
- **Interrupt handlers**: Wake up driver when I/O completed
- **Hardware**: Perform I/O operation
Example I/O Devices

Timers
Monitors
Key boards
Graphical user interfaces
Network interfaces
Disk
...

Programmable Timer

Crystal oscillator

Counter is decremented at each pulse

Holding register is used to load the counter

One-shot mode:
Counter initialized then decremented until zero
At zero a single interrupt occurs

Square wave mode:
At zero the counter is reinitialized with the same value
Periodic interrupts (called “clock ticks”) occur
Time

500 MHz Crystal (oscillates every 2 nanoseconds)
32 bit register overflows in 8.6 seconds
  - So how can we remember what the time is?

Backup clock
  - Similar to digital watch
  - Low-power circuitry, battery-powered
  - Periodically reset from the internet
  - UTC: Universal Coordinated Time
  - Unix: Seconds since Jan. 1, 1970
  - Windows: Seconds since Jan. 1, 1980
Goals of Timer Software

Maintain time of day
- Must update the time-of-day every tick

Prevent processes from running too long

Account for CPU usage
- Separate timer for every process
- Charge each tick to the current process

Handling the “Alarm” syscall
- User programs ask to be sent a signal at a given time

Providing watchdog timers for the OS itself
- When to stop the disk, switch to low power mode, etc

Doing profiling, monitoring, and statistics gathering
Software Timers

A process can ask for notification (alarm) at time T
At time T, the OS will signal the process

Processes can “go to sleep until time T”

Several processes can have active timers

The CPU has only one clock
Must service the “alarms” in the right order

Keep a sorted list of all timers
Each entry tells when the alarm goes off and what to do
then
Software Timers

Alarms set for 4203, 4207, 4213, 4215 and 4216. Each entry tells how many ticks past the previous entry. On each tick, decrement the “NextSignal”. When it gets to 0, then signal the process.