CS 134: Operating Systems

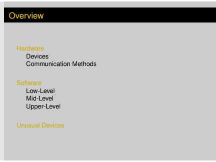
I/O Hardware

2013-05-17

CS34

CS 134:
Operating Systems
I/O Hardware

Overview



Overview

Hardware
Devices
Communication Methods

Software
Low-Level
Mid-Level
Upper-Level

Unusual Devices

Classifying Devices

What is an I/O device?

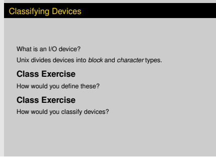
Unix divides devices into *block* and *character* types.

Class Exercise

How would you define these?

Class Exercise

How would you classify devices?


Classifying Devices

What is an I/O device?

Unix divides devices into *block* and *character* types.

Class Exercise

How would you define these?

Class Exercise

How would you classify devices?

A block device can be defined as one that can only be accessed in block-sized units, or as one that has a fixed size. A character device is anything else. Are these sensible definitions?

Besides the obvious devices, what about clocks and memory-mapped screens? What about GPUs? Are they even devices? You can give them a list of polygons to display. . . but you can also give them password-cracking code to execute. Are there any other “weird” example?

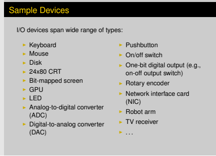
Sample Devices

I/O devices span wide range of types:

- Keyboard
- Mouse
- Disk
- 24x80 CRT
- Bit-mapped screen
- GPU
- LED
- Analog-to-digital converter (ADC)
- Digital-to-analog converter (DAC)
- Pushbutton
- On/off switch
- One-bit digital output (e.g., on-off output switch)
- Rotary encoder
- Network interface card (NIC)
- Robot arm
- TV receiver
- . . .

2013-05-17

CS34
└Hardware
 └Devices
  └Sample Devices


Sample Devices

I/O devices span wide range of types:

- Keyboard
- Mouse
- Disk
- 24x80 CRT
- Bit-mapped screen
- GPU
- LED
- Analog-to-digital converter (ADC)
- Digital-to-analog converter (DAC)
- Pushbutton
- On/off switch
- One-bit digital output (e.g., on-off output switch)
- Rotary encoder
- Network interface card (NIC)
- Robot arm
- TV receiver
- . . .

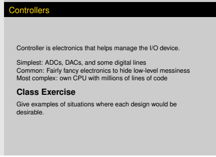
Controllers

Controller is electronics that helps manage the I/O device.

Simplest: ADCs, DACs, and some digital lines
Common: Fairly fancy electronics to hide low-level messiness
Most complex: own CPU with millions of lines of code

Class Exercise

Give examples of situations where each design would be desirable.



Controllers

Controller is electronics that helps manage the I/O device.

Simplest: ADCs, DACs, and some digital lines
Common: Fairly fancy electronics to hide low-level messiness
Most complex: own CPU with millions of lines of code

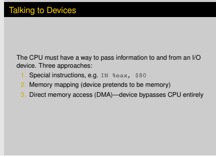
Class Exercise

Give examples of situations where each design would be desirable.

Talking to Devices

The CPU must have a way to pass information to and from an I/O device. Three approaches:

1. Special instructions, e.g. `IN %eax, $80`
2. Memory mapping (device pretends to be memory)
3. Direct memory access (DMA)—device bypasses CPU entirely


Talking to Devices

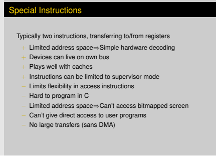
The CPU must have a way to pass information to and from an I/O device. Three approaches:

1. Special instructions, e.g. `IN %eax, $80`
2. Memory mapping (device pretends to be memory)
3. Direct memory access (DMA)—device bypasses CPU entirely

Special Instructions

Typically two instructions, transferring to/from registers

+ Limited address space⇒Simple hardware decoding
+ Devices can live on own bus
+ Plays well with caches
+ Instructions can be limited to supervisor mode
– Limits flexibility in access instructions
– Hard to program in C
– Limited address space⇒Can't access bitmapped screen
– Can't give direct access to user programs
– No large transfers (sans DMA)



Special Instructions

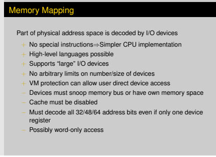
Typically two instructions, transferring to/from registers

+ Limited address space⇒Simple hardware decoding
+ Devices can live on own bus
+ Plays well with caches
+ Instructions can be limited to supervisor mode
– Limits flexibility in access instructions
– Hard to program in C
– Limited address space⇒Can't access bitmapped screen
– Can't give direct access to user programs
– No large transfers (sans DMA)

Memory Mapping

Part of physical address space is decoded by I/O devices

- + No special instructions⇒Simpler CPU implementation
- + High-level languages possible
- + Supports “large” I/O devices
- + No arbitrary limits on number/size of devices
- + VM protection can allow user direct device access
- – Devices must snoop memory bus or have own memory space
- – Cache must be disabled
- – Must decode all 32/48/64 address bits even if only one device register
- – Possibly word-only access



Memory Mapping

Part of physical address space is decoded by I/O devices

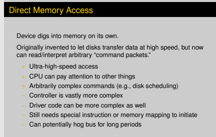
- + No special instructions⇒Simpler CPU implementation
- + High-level languages possible
- + Supports “large” I/O devices
- + No arbitrary limits on number/size of devices
- + VM protection can allow user direct device access
- – Devices must snoop memory bus or have own memory space
- – Cache must be disabled
- – Must decode all 32/48/64 address bits even if only one device register
- – Possibly word-only access

Direct Memory Access

Device digs into memory on its own.

Originally invented to let disks transfer data at high speed, but now can read/interpret arbitrary “command packets.”

+ Ultra-high-speed access
+ CPU can pay attention to other things
+ Arbitrarily complex commands (e.g., disk scheduling)
– Controller is vastly more complex
– Driver code can be more complex as well
– Still needs special instruction or memory mapping to initiate
– Can potentially hog bus for long periods


Direct Memory Access

Device digs into memory on its own.

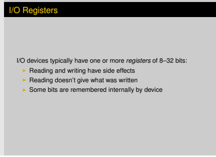
Originally invented to let disks transfer data at high speed, but now can read/interpret arbitrary “command packets.”

+ Ultra-high-speed access
+ CPU can pay attention to other things
+ Arbitrarily complex commands (e.g., disk scheduling)
– Controller is vastly more complex
– Driver code can be more complex as well
– Still needs special instruction or memory mapping to initiate
– Can potentially hog bus for long periods

I/O Registers

I/O devices typically have one or more *registers* of 8–32 bits:

- Reading and writing have side effects
- Reading doesn't give what was written
- Some bits are remembered internally by device



I/O Registers

I/O devices typically have one or more *registers* of 8–32 bits:

- Reading and writing have side effects
- Reading doesn't give what was written
- Some bits are remembered internally by device

Example of I/O Registers—8251 UART

Status Register—Read:

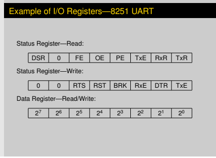
DSR	0	FE	OE	PE	TxE	RxR	TxR

Status Register—Write:

0	0	RTS	RST	BRK	RxE	DTR	TxE

Data Register—Read/Write:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0



Example of I/O Registers—8251 UART

Status Register—Read:

DSR	0	FE	OE	PE	TxE	RxR	TxR

Status Register—Write:

0	0	RTS	RST	BRK	RxE	DTR	TxE

Data Register—Read/Write:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

FE: Framing error; OE: Overflow; PE: Parity; TxE: Transmitter Empty; TxR: Ready (can be ready w/o being empty). BRK is transient; RxE/TxE are enable bits. Data register reads only on RxR and resets RxR. Data register write transmits and resets TxR.

I/O Completion

Always need way to detect I/O completion:

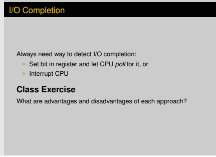
- Set bit in register and let CPU *poll* for it, or
- Interrupt CPU

Class Exercise

What are advantages and disadvantages of each approach?

2013-05-17

CS34
└Hardware
  └Communication Methods
    └I/O Completion


I/O Completion

Always need way to detect I/O completion:

- Set bit in register and let CPU *poll* for it, or
- Interrupt CPU

Class Exercise

What are advantages and disadvantages of each approach?

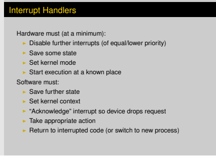
Interrupt Handlers

Hardware must (at a minimum):

- Disable further interrupts (of equal/lower priority)
- Save some state
- Set kernel mode
- Start execution at a known place

Software must:

- Save further state
- Set kernel context
- “Acknowledge” interrupt so device drops request
- Take appropriate action
- Return to interrupted code (or switch to new process)



Interrupt Handlers

Hardware must (at a minimum):

- Disable further interrupts (of equal/lower priority)
- Save some state
- Set kernel mode
- Start execution at a known place

Software must:

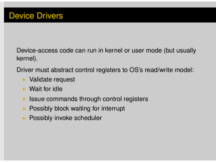
- Save further state
- Set kernel context
- “Acknowledge” interrupt so device drops request
- Take appropriate action
- Return to interrupted code (or switch to new process)

Device Drivers

Device-access code can run in kernel or user mode (but usually kernel).

Driver must abstract control registers to OS's read/write model:

- Validate request
- Wait for idle
- Issue commands through control registers
- Possibly block waiting for interrupt
- Possibly invoke scheduler


Device Drivers

Device-access code can run in kernel or user mode (but usually kernel).

Driver must abstract control registers to OS's read/write model:

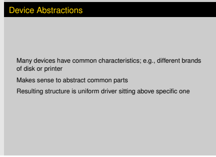
- Validate request
- Wait for idle
- Issue commands through control registers
- Possibly block waiting for interrupt
- Possibly invoke scheduler

Device Abstractions

Many devices have common characteristics; e.g., different brands of disk or printer

Makes sense to abstract common parts

Resulting structure is uniform driver sitting above specific one


Device Abstractions

Many devices have common characteristics; e.g., different brands of disk or printer

Makes sense to abstract common parts

Resulting structure is uniform driver sitting above specific one

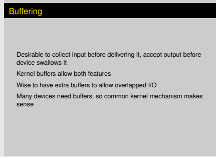
Buffering

Desirable to collect input before delivering it, accept output before device swallows it

Kernel buffers allow both features

Wise to have extra buffers to allow overlapped I/O

Many devices need buffers, so common kernel mechanism makes sense



Buffering

Desirable to collect input before delivering it, accept output before device swallows it

Kernel buffers allow both features

Wise to have extra buffers to allow overlapped I/O

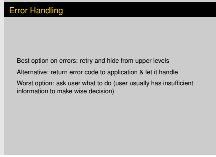
Many devices need buffers, so common kernel mechanism makes sense

Error Handling

Best option on errors: retry and hide from upper levels

Alternative: return error code to application & let it handle

Worst option: ask user what to do (user usually has insufficient information to make wise decision)


Error Handling

Best option on errors: retry and hide from upper levels

Alternative: return error code to application & let it handle

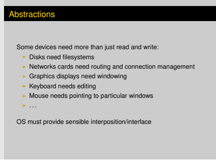
Worst option: ask user what to do (user usually has insufficient information to make wise decision)

Abstractions

Some devices need more than just read and write:

- Disks need filesystems
- Networks cards need routing and connection management
- Graphics displays need windowing
- Keyboard needs editing
- Mouse needs pointing to particular windows
- . . .

OS must provide sensible interposition/interface



Abstractions

Some devices need more than just read and write:

- Disks need filesystems
- Networks cards need routing and connection management
- Graphics displays need windowing
- Keyboard needs editing
- Mouse needs pointing to particular windows
- . . .

OS must provide sensible interposition/interface

API

User-space applications need standardized interface

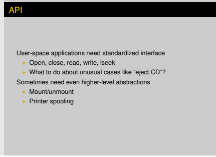
- Open, close, read, write, lseek
- What to do about unusual cases like “eject CD”?

Sometimes need even higher-level abstractions

- Mount/unmount
- Printer spooling

2013-05-17

CS34
└Software
  └Upper-Level
    └API


API

User-space applications need standardized interface
- Open, close, read, write, lseek
- What to do about unusual cases like “eject CD”?

Sometimes need even higher-level abstractions
- Mount/unmount
- Printer spooling

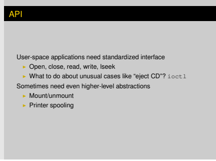
API

User-space applications need standardized interface

- Open, close, read, write, lseek
- What to do about unusual cases like “eject CD”? `ioctl`

Sometimes need even higher-level abstractions

- Mount/unmount
- Printer spooling


API

User-space applications need standardized interface
- Open, close, read, write, lseek
- What to do about unusual cases like “eject CD”? ioctl

Sometimes need even higher-level abstractions
- Mount/unmount
- Printer spooling

Windowing Systems

Modern GUIs need window management:

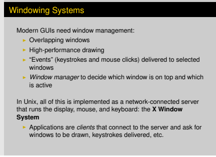
- Overlapping windows
- High-performance drawing
- “Events” (keystrokes and mouse clicks) delivered to selected windows
- *Window manager* to decide which window is on top and which is active

In Unix, all of this is implemented as a network-connected server that runs the display, mouse, and keyboard: the **X Window System**

- Applications are *clients* that connect to the server and ask for windows to be drawn, keystrokes delivered, etc.

2013-05-17

CS34
└Software
 └Upper-Level
  └Windowing Systems


Windowing Systems

Modern GUIs need window management:

- Overlapping windows
- High-performance drawing
- “Events” (keystrokes and mouse clicks) delivered to selected windows
- *Window manager* to decide which window is on top and which is active

In Unix, all of this is implemented as a network-connected server that runs the display, mouse, and keyboard: the **X Window System**

- Applications are *clients* that connect to the server and ask for windows to be drawn, keystrokes delivered, etc.

Clocks

Clocks come in two flavors:

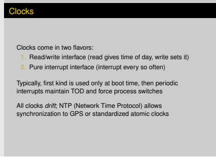
1. Read/write interface (read gives time of day, write sets it)
2. Pure interrupt interface (interrupt every so often)

Typically, first kind is used only at boot time, then periodic interrupts maintain TOD and force process switches

All clocks *drift*; NTP (Network Time Protocol) allows synchronization to GPS or standardized atomic clocks

2013-05-17

CS34
└ Unusual Devices
  └ Clocks


Clocks

Clocks come in two flavors:

1. Read/write interface (read gives time of day, write sets it)
2. Pure interrupt interface (interrupt every so often)

Typically, first kind is used only at boot time, then periodic interrupts maintain TOD and force process switches

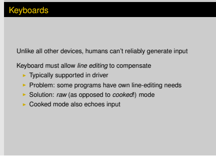
All clocks *drift*; NTP (Network Time Protocol) allows synchronization to GPS or standardized atomic clocks

Keyboards

Unlike all other devices, humans can’t reliably generate input

Keyboard must allow *line editing* to compensate

- Typically supported in driver
- Problem: some programs have own line-editing needs
- Solution: *raw* (as opposed to *cooked*!) mode
- Cooked mode also echoes input



Keyboards

Unlike all other devices, humans can’t reliably generate input

Keyboard must allow *line editing* to compensate

- Typically supported in driver
- Problem: some programs have own line-editing needs
- Solution: *raw* (as opposed to *cooked*!) mode
- Cooked mode also echoes input

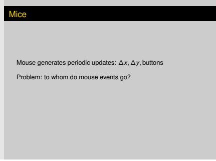
Mice

Mouse generates periodic updates: $\Delta x, \Delta y$, buttons

Problem: to whom do mouse events go?

2013-05-17

CS34
└─Unusual Devices
  └─Mice


Mice

Mouse generates periodic updates: $\Delta x, \Delta y$, buttons

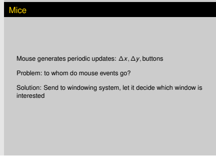
Problem: to whom do mouse events go?

Mice

Mouse generates periodic updates: $\Delta x, \Delta y$, buttons

Problem: to whom do mouse events go?

Solution: Send to windowing system, let it decide which window is interested


Mice

Mouse generates periodic updates: $\Delta x, \Delta y$, buttons

Problem: to whom do mouse events go?

Solution: Send to windowing system, let it decide which window is interested