I/O Systems and Networks

DM510 Operating Systems

Lars Rohwedder

Disclaimer

These slides contain (modified) content and media from the official Operating System Concepts slides:
https://www.os-book.com/OS10/slide-dir/index.html

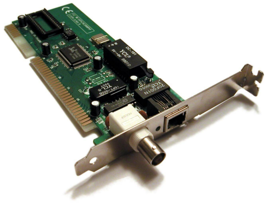
Today’s lecture

- Introduction to second project
- Chapter 12 of course book
- Chapter 19 of course book

I/O Devices

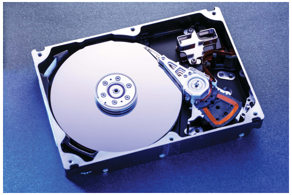
Types of devices

Computer systems contain a large variety of devices, for example:



Network card

Graphics processor



Hard disk drive

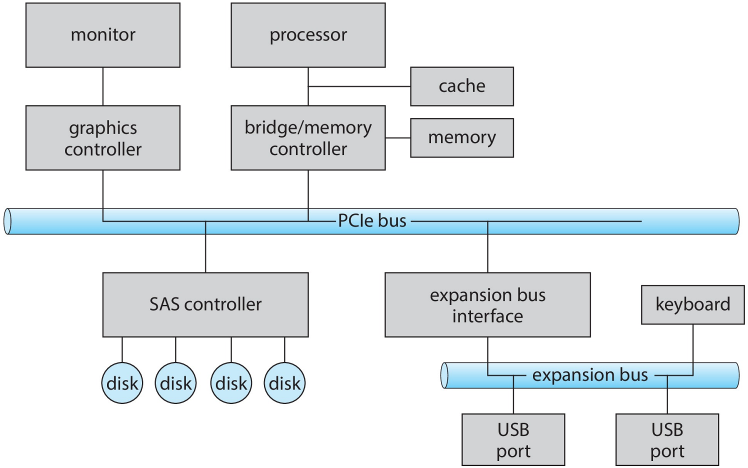
These devices have built-in **controllers**, small processors that operate asynchronously from the CPU and have their own local registers

Examples of device registers

- **Data-in register:** CPU writes data to it, device reads
- **Data-out register:** Device writes data to it, CPU reads
- **Status register:** Device indicates its status, for example, task completed, data available, error
- **Control register:** CPU writes command to device in it

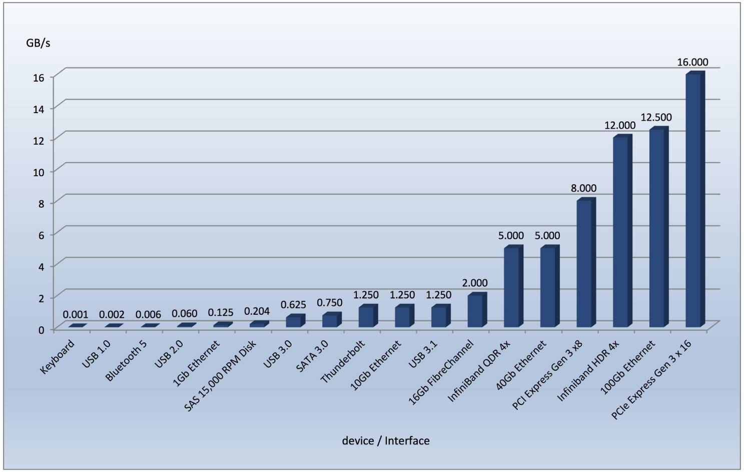
Physical connection to devices

Devices are typically connected via **busses**, wires that often connect endpoints and have their own controllers to multiplex data transfers between various endpoints



Bus speeds

With faster and more CPU cores, speed of busses needs to scale accordingly to avoid becoming the bottleneck



Software access to devices

Port-mapped I/O. Special instructions to read and write register values to **port** addresses, which are associated with device controllers. Has become less common in recent time.

Memory-mapped I/O. Specific memory addresses are associated with device controllers. When the CPU writes to such an address (as it would to normal main memory), the data is instead sent to a register of the device.

Software access to devices

Port-mapped I/O. Special instructions to read and write register values to **port** addresses, which are associated with device controllers. Has become less common in recent time.

Memory-mapped I/O. Specific memory addresses are associated with device controllers. When the CPU writes to such an address (as it would to normal main memory), the data is instead sent to a register of the device.

- Simplest variant of I/O is by **polling**, where CPU actively checks if data can be written, read, etc. Polling is inefficient if device controller is slower than CPU

Polling example for CPU

```
while( status.busy || control.ready )
    ;
data_out = produce();
control.ready = 1;
```

Polling example for device controller

```
while( control.ready == 0 )
    ;
status.busy = 1;
control.ready = 0;
consume(data_out);
status.error = 0;
status.busy = 0;
```

Software access to devices

Port-mapped I/O. Special instructions to read and write register values to **port** addresses, which are associated with device controllers. Has become less common in recent time.

Memory-mapped I/O. Specific memory addresses are associated with device controllers. When the CPU writes to such an address (as it would to normal main memory), the data is instead sent to a register of the device.

- Simplest variant of I/O is by **polling**, where CPU actively checks if data can be written, read, etc. Polling is inefficient if device controller is slower than CPU

Polling example for CPU

```
while( status.busy || control.ready )
    ;
data_out = produce();
control.ready = 1;
```

Polling example for device controller

```
while( control.ready == 0 )
    ;
status.busy = 1;
control.ready = 0;
consume(data_out);
status.error = 0;
status.busy = 0;
```

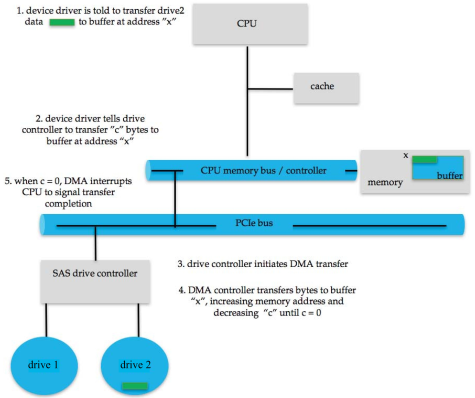
- Alternative is using interrupts to notify CPU (**interrupt-driven I/O**)
- Some devices have **FIFO buffers** that store several commands to be executed by device

Direct-memory access

Programmed I/O where CPU transfers data byte by byte from memory to device can be inefficient and limit throughput.

In **direct-memory access (DMA)**, devices can read and write to main memory without CPU

- CPU sends command and address of memory section
- Device controller writes or reads independently to section
- Device controller sends interrupt to CPU when done
- May still impact CPU performance, since device controller can occupy memory bus, but much lower impact than programmed I/O



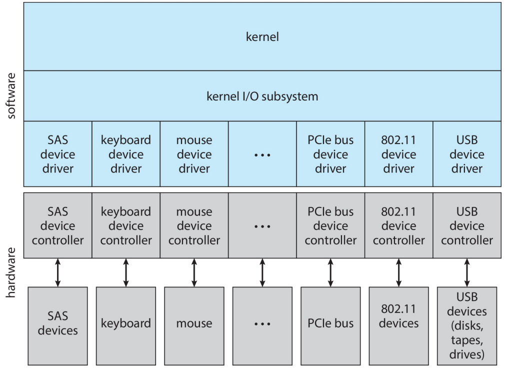
1. device driver is told to transfer drive2 data to buffer at address "x"
CPU
cache
2. device driver tells drive controller to transfer "c" bytes to buffer at address "x"
CPU memory bus / controller
x
memory
buffer
5. when c = 0, DMA interrupts CPU to signal transfer completion
PCIe bus
SAS drive controller
3. drive controller initiates DMA transfer
4. DMA controller transfers bytes to buffer "x", increasing memory address and decreasing "c" until c = 0
drive 1
drive 2

Device Drivers

Drivers

Drivers hide most of the low level details of device communication and provide standardized interfaces. They are often provided in modules that can be loaded into kernel without having to update the kernel

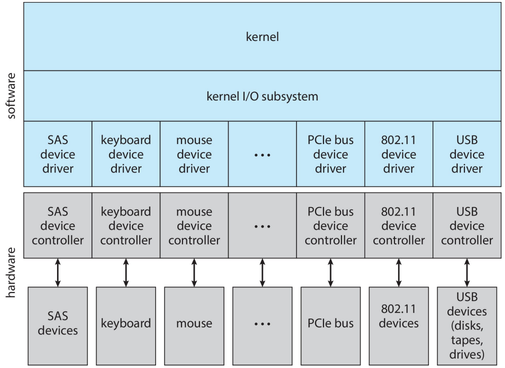
- For example, application programmer should not need to adjust code for different network cards that may use slighly different device registers

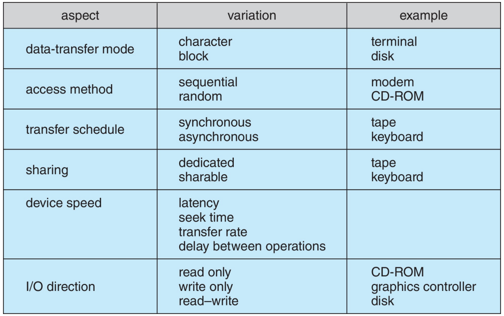


Drivers

Drivers hide most of the low level details of device communication and provide standardized interfaces. They are often provided in modules that can be loaded into kernel without having to update the kernel

- For example, application programmer should not need to adjust code for different network cards that may use slighly different device registers





aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Drivers in Linux

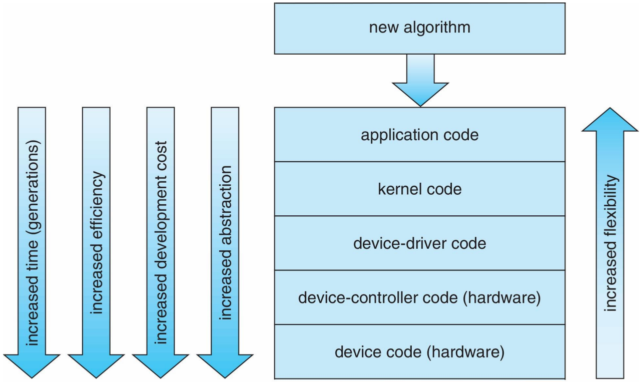
- Devices are exposed as file-like objects in file system, typically in directory `/dev/`
- Additionally, devices have **major and minor numbers** for identification. The major number identifies the device type and the minor number the instance
- Opening, closing, reading and writing to devices uses same system calls as files
- For other operations, special system call **ioctl** is used. Examples: ejecting a CD ROM disk, configuring a device, etc.
- Programming a driver means essentially implementing a number of functions (open, read, write, etc.). On the corresponding system call on the device, the kernel will call the driver module's functions

Project 2

In the second project you will write your own device driver for the Raspberry Pi

Tradeoffs for levels of abstraction

Functionality can be implemented at different levels of the hardware-software stack. This comes with tradeoffs regarding development cost, flexibility, efficiency, etc.



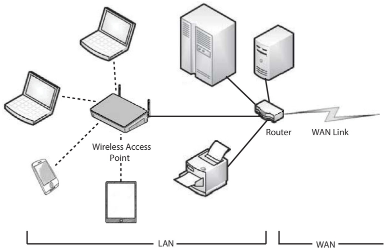
Networks

Setting

- We want to allow communication between different hosts
- Applications should be able to establish connection to other applications on (usually) different host and use it as reliable bi-directional stream of bytes (see TCP later)
- How does data reach correct LAN, host, application? How can we achieve reliability?

Network types

- **Local-area network (LAN):** hosts are all within small geographical area, for example, home, office, university
- **Wide-area network (WAN):** hosts span large geographical area, for example, internet, World Wide Web

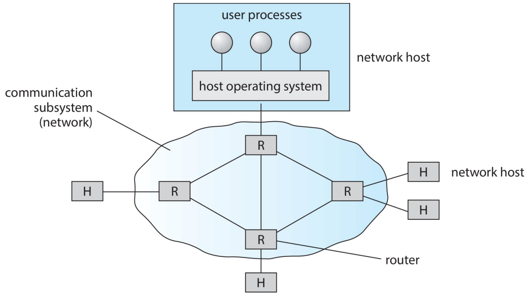


Setting

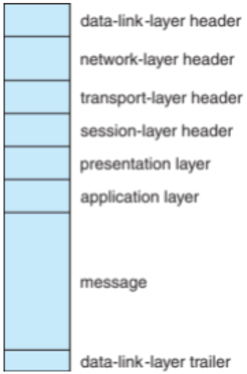
- We want to allow communication between different hosts
- Applications should be able to establish connection to other applications on (usually) different host and use it as reliable bi-directional stream of bytes (see TCP later)
- How does data reach correct LAN, host, application? How can we achieve reliability?

Network types

- **Local-area network (LAN):** hosts are all within small geographical area, for example, home, office, university
- **Wide-area network (WAN):** hosts span large geographical area, for example, internet, World Wide Web



Protocol stacks



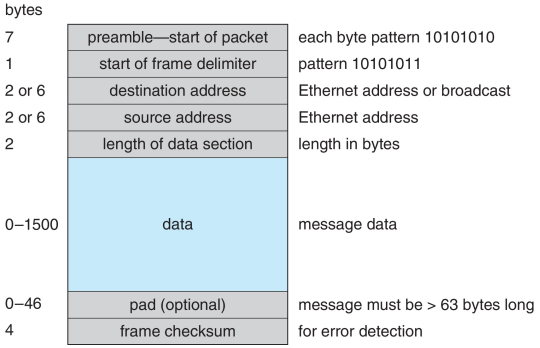
- Network communication is implemented as stack of protocol layers that add more and more abstraction, each protocol making use of the next lower protocol
- Within a data packet each protocol reserves a number of bytes, the **header**, for its information
- OSI model proposes various layers and their roles (see left). Note: session, presentation, application layers not used in modern technology

Focus of this lecture

We only look at TCP/IP related technology. We omit technical details and focus on concepts.

Ethernet (data-link layer)

- Network hardware can send bit stream of low or high signals
- Ethernet protocol allows sending variable-size packets between hosts with direct physical connection. Sometimes more than two hosts can be directly connected via hub
- WiFi or mobile networks have similar role to ethernet



bytes		
7	preamble—start of packet	each byte pattern 10101010
1	start of frame delimiter	pattern 10101011
2 or 6	destination address	Ethernet address or broadcast
2 or 6	source address	Ethernet address
2	length of data section	length in bytes
0–1500	data	message data
0–46	pad (optional)	message must be > 63 bytes long
4	frame checksum	for error detection

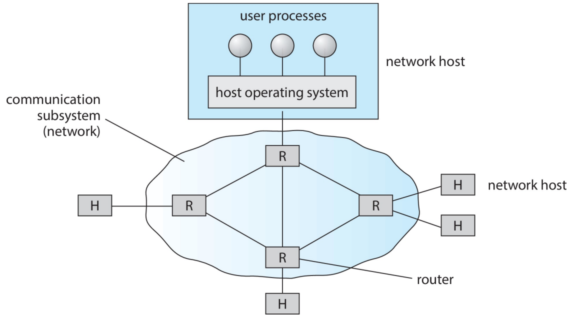
Ethernet header

- Each device capable of ethernet has a unique **ethernet (MAC) address** provided by the vendor
- Each package has source and destination MAC addresses
- Preamble, frame delimiter, and length used to identify start and end of package
- checksum to detect bit errors

Internet protocol (network layer)

In a large network, packages need to be **routable**: we need to be able to find the destination without sending it to every host

- In the internet protocol (IP), each host has unique **IP address** (with some exceptions that we omit here)
- The IP address is hierachical, which means that it can be seen whether an IP address belongs to a host outside of the LAN and needs to be send to the **gateway**
- Routers (devices dedicated to forwarding packets to the right networks) have **routing tables** that they use to identify where to forward packaets to



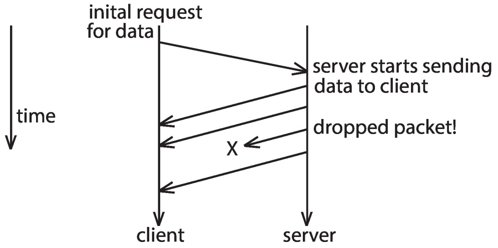
Domain name service

- Instead of IP addresses we are often given domain names, for example, sdu.dk, wikipedia.org
- Hosts typically know IP addresses of one or more **name servers** that can be asked to translate domain names into IP addresses
- There is a local variant of DNS, called mDNS, where hosts broadcase their names within the local network. You may know this from the Raspberry Pi

User datagram protocol (transport layer)

Transport layer allows several applications to use network. The dominant protocols here are UDP and TCP. The User datagram protocol (UDP) adds only minimal features on top of IP.

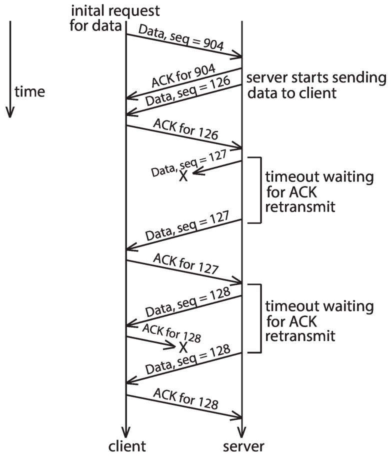
- Each UDP packet contains a **source** and a **destination port**
- Applications can listen to specific ports to receive packets sent to this port (at most one application per port)
- Connectionless protocol, but source port can be used for response
- No guarantee or acknowledgement that packets arrive, no guarantee that they arrive in order



Transmission control protocol (transport layer)

The transmission control protocol (TCP) creates connections of reliable byte stream between applications on (usually) different hosts. It is used for vast majority of applications (much more than UDP)

- TCP packets contain a **source** and a **destination port** (as in UDP)
- Applications is either **server** and listens to specific ports to accept connections or **client** and makes connection request to a server
- 3-way handshake to establish connection
- Packages have **sequence numbers** that are used to ensure correct order
- Each package is acknoledged by its sequences number and otherwise retransmitted



time
inital request for data
Data, seq = 904
ACK for 904
Data, seq = 126
server starts sending data to client
ACK for 126
Data, seq = 127
X
timeout waiting for ACK retransmit
Data, seq = 127
ACK for 127
Data, seq = 128
ACK for 128
X
timeout waiting for ACK retransmit
Data, seq = 128
ACK for 128
client
server