Operating Systems and Networks

Summary of the course in spring 2010 by Gustavo Alonso and Timothy Roscoe

Stefan Heule

2010-06-06

Table of Contents

1 Network introduction 6
1.1 Layering 6
1.2 Performance 6
1.3 Units 7
1.4 Application requirements 7
2 Application Layer Protocols 7
2.1 Basics of application layer protocols 7
2.1.1 Services needed by applications 7
2.2 Client/server paradigm 7
2.3 Domain name system 8
2.3.1 Types of name servers 8
2.3.2 Basic DNS query 8
2.3.3 Caching 8
3 Remote procedure calls 9
3.1 Introduction and problem formulation 9
3.2 Individual steps in RPC 9
3.3 Call semantics 10
3.4 Summary 11
3.5 Distributed environments 11
3.6 Transactional RPC 11
4 Socket programming 13
4.1 Introduction 13
4.2 Comparison of Java and C API 13
4.3 Java socket programming with TCP 13
4.4 Java socket programming with UDP 14
4.5 C socket programming with TCP 15
5 Reliable data transfer 17
5.1 Incremental development of reliable data transfer (RDT) 17
5.1.1 RDT 1.0: reliable transfer over reliable channel 17
5.1.2 RDT 2.0: channel with bit errors 17
5.1.3 RDT 2.1 18
5.1.4 RDT 2.2: NAK-free 18
5.1.5 RDT 3.0: channels with errors and loss 18
5.2 Pipelining 19
5.2.1 Go-Back-N 19
5.2.2 Selective repeat 20
6 Queuing theory 22
6.1 Notation 22
6.1.1 Arrival rate distribution 22
6.1.2 Service time per job 22
6.1.3 Service discipline 22
6.1.4 System capacity 22
6.1.5 Number of servers 22
6.1.6 Population 23
6.2 Results 23
6.2.1 Birth-death-process 23
6.2.2 Steady state probability 24
6.2.3 M/M/1 24
6.3 Operational laws 24
6.3.1 Job flow balance 24
6.3.2 Derivations 25

7 Transport layer 26
7.1 User datagram protocol UDP 26
7.2 TCP 27
7.2.1 TCP segment layout 27
7.2.2 Sequence numbers 27
7.2.3 TCP connection management 27
7.2.4 TCP reliability and flow control 28
7.2.5 TCP round trip time and timeout 29
7.2.6 Fast retransmit 29
7.2.7 Congestion and congestion control 29
8 Network layer 31
8.1 Network layer services 31
8.2 Internet protocol IP 32
8.2.1 IP addresses 32
8.3 Additional protocols dealing with network layer information 33
8.3.1 Internet control message protocol ICMP 33
8.3.2 Dynamic host configuration protocol DHCP 34
8.3.3 Network address translation NAT 34
8.4 Routing 35
8.4.1 Distance vector protocols 35
8.4.2 Link-state routing protocols 37
8.4.3 Comparing routing algorithms 39
8.4.4 Interdomain routing 39
8.4.5 Routers 41
8.5 IPv6 42
9 Link layer 43
9.1 End-to-end argument 43
9.2 Encoding 43
9.2.1 Non-return to zero NRZ 43
9.2.2 Non-return to zero inverted 43
9.2.3 Manchester encoding 44
9.2.4 4B/5B encoding 44
9.3 Framing 44
9.3.1 Point to point protocol PPP 44
9.3.2 High-level data link control HDLC 45
9.3.3 Synchronous optical network 45
9.4 Error detection 45
9.4.1 Parity checking 45
9.4.2 Cyclic redundancy check CRC 46
9.5 Media access control 46
9.5.1 Turn-taking protocols (e.g. round robin) 46
9.5.2 Random access protocols 46
9.5.3 Slotted Aloha 46
9.5.4 Pure (unslotted) aloha 47
9.5.5 Demand assigned multiple access DAMA 47
9.5.6 Carrier sense multiple access CSMA 47
9.5.7 CSMA/CD (collision detect) 47
9.5.8 Ethernet 48
10 Packet switching 50
10.1 Virtual circuit switching 50
10.1.1 Asynchronous transfer mode ATM 50
10.2 Datagram switching 50

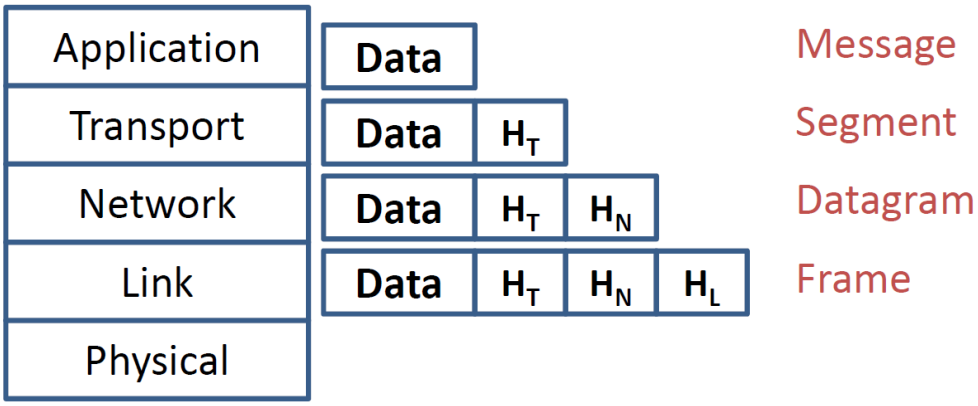
10.2.1 Address resolution protocol ARP 51
10.2.2 Bridges and switches 51
10.2.3 Virtual LANs 52
10.2.4 Switches vs. Routers 52
11 Naming 54
11.1 Introduction 54
11.2 Naming operations 54
11.3 Naming policy alternatives 54
11.4 Types of lookups 55
11.5 Default and explicit contexts, qualified names 55
11.6 Path names, naming networks, recursive resolution 56
11.7 Multiple lookup 56
11.8 Naming discovery 56
12 Virtualization 57
12.1 Examples 57
12.2 The operation system as resource manager 59
12.3 General operation system structure 60
13 Processes and threads 61
13.1 System calls 61
13.2 Processes 61
13.2.1 Process creation 62
13.3 Kernel threads 63
13.3.1 Kernel threads 63
13.3.2 User-space threads 64
14 Scheduling 67
14.1 Introduction 67
14.1.1 Batch workloads 67
14.1.2 Interactive workloads 67
14.1.3 Soft real-time workloads 67
14.1.4 Hard real-time workloads 67
14.2 Assumptions and definitions 68
14.3 Batch-oriented scheduling 68
14.4 Scheduling interactive loads 69
14.4.1 Linux O(1) scheduler 70
14.4.2 Linux “completely fair scheduler” 70
14.5 Real-time scheduling 70
14.5.1 Rate-monotonic scheduling RMS 71
14.5.2 Earliest deadline first EDF 71
14.6 Scheduling on multiprocessors 71
15 Inter-process communication 72
15.1 Hardware support for synchronization 72
15.1.1 Spinning 72
15.2 IPC with shared memory and interaction with scheduling 73
15.2.1 Priority inversion 73
15.2.2 Priority inheritance 73
15.3 IPC without shared memory 73
15.3.1 Unix pipes 74
15.3.2 Local remote procedure calls 74
15.3.3 Unix signals 74
16 Memory management 76
16.1 Memory management schemes 76
16.1.1 Partitioned memory 76

16.1.2 Segmentation 77
16.1.3 Paging 78
17 Demand paging 82
17.1 Copy-on-write COW 82
17.2 Demand-paging 82
17.2.1 Page replacement 83
17.2.2 Frame allocation 84
18 I/O systems 85
18.1 Interrupts 86
18.2 Direct memory access 86
18.3 Device drivers 87
18.3.1 Example: network receive 87
18.4 The I/O subsystem 88
19 File system abstractions 89
19.1 Introduction 89
19.2 Filing system interface 89
19.2.1 File naming 89
19.2.2 File types 90
19.2.3 Access control 90
19.3 Concurrency 91
20 File system implementation 92
20.1 On-disk data structures 92
20.2 Representing a file on disk 92
20.2.1 Contiguous allocation 92
20.2.2 Extent-based system 93
20.2.3 Linked allocation 93
20.2.4 Indexed allocation 93
20.3 Directory implementation 94
20.4 Free space management 94
20.5 In-memory data structures 94
20.6 Disks, partitions and logical volumes 95
20.6.1 Partitions 95
20.6.2 Logical volumes 95
21 Networking stacks 96
21.1 Networking stack 96

1 Network introduction

1.1 Layering

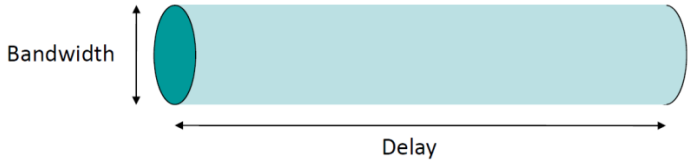
- TCP/IP reference model
  - Application (HTTP, Bittorrent)
  - Transport (TCP, UDP)
  - Network (IP)
  - Link (PPP, WiFi, Ethernet)
  - Physical (Fiber, Wireless)
- ISO/OSI model
  - Application
  - Presentation (syntax, format and semantics of information transmitted)
  - Session (Long-term transport issues, e.g. check pointing)
  - Transport
  - Network
  - Link
  - Physical
- Data encapsulation and naming



1.2 Performance

- Two basic measures
  - **Bandwidth**
    - Also known as **throughput**
    - approx.: bits transferred per unit of time
    - e.g. 25 Mbit/s
  - **Latency**
    - Time for a message to traverse the network
    - Also known as **delay**
    - Half the round-trip-time (RTT)
- Bandwidth-delay product

- What the sender can send before the receiver sees anything.



Throughput = (Transfer size) / (Transfer time)

Transfer time = RTT + (Transfer size) / Bandwidth
= (Request + First bit delay) + (Transfer time)

1.3 Units

- Memory often measured in bytes
  - $1\text{KB} = 2^{10}\text{Bytes}$
- Bandwidth often measured in bits
  - $1\text{kb/s} = 10^{3}\text{bits per seconds}$

1.4 Application requirements

- Jitter: Variation in the end-to-end latency (i.e. RTT)

2 Application Layer Protocols

2.1 Basics of application layer protocols

- Communication between distributed values.
- Application layer protocol is part of application
- Defines messages exchanged
- Uses communication facilities provided by transport layer (e.g. UDP, TCP)

2.1.1 Services needed by applications

- Data loss: some apps can accept some loss, others can't.
- Timing: some apps need low delay to be effective.
- Bandwidth: some apps need a minimum bandwidth to be useful, others use whatever bandwidth is available.

2.2 Client/server paradigm

- Client
  - Initiate contact ("speaks first")
  - request service
- Server
  - Provide service

2.3 Domain name system

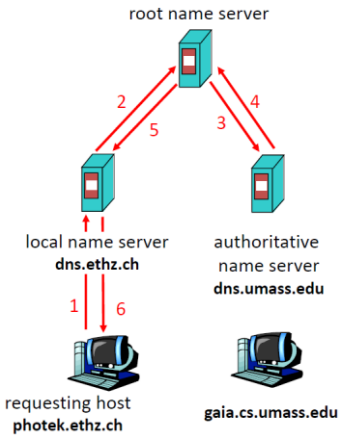
- Internally, the internet uses IP addresses, but humans rather have domain names.
- DNS serves as a distributed database to look up IP addresses for a given domain name.
- Implemented as hierarchy of many **name servers**.
    - It is not centralized for various reasons: Single point-of-failure, traffic volume, maintenance, distant database for some parts of the world. Also, it does not scale.
    - No server has all the mappings!

2.3.1 Types of name servers

- Root name servers
    - Known to all, 13 logical servers worldwide
    - Fixed configuration, updated manually
- Authoritative name server
    - Stores IP addresses of all nodes in that zone
- Local name server
    - Companies, ISPs

2.3.2 Basic DNS query

- Recursive!
- Resolve de.wikipedia.org
    - Ask root server (where is .org?), get IP of .org name server
    - Ask .org name server (where is .wikipedia.org), get IP of .wikipedia.org name server
    - …



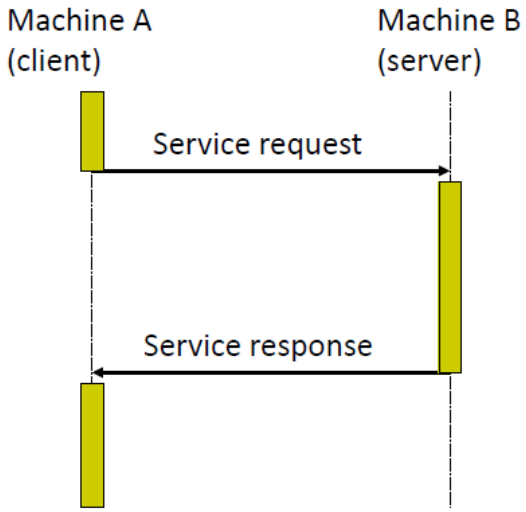
2.3.3 Caching

- DNS system makes *extensive* use of caching for efficiency/scalability

3 Remote procedure calls

3.1 Introduction and problem formulation

- Context
  - The most accepted standard for network communication is IP, where IP is designed to be hidden behind other layers, e.g. TCP and UDP.
  - TCP/IP and UDP/IP are visible to applications through sockets. However, these sockets are still quite low-level.
  - RPC appeared as a way to hide communication details behind a procedure call and bridge heterogeneous environments
  - RPC is the standard for distributed (client-server) computing



- Problems to solve
  - How do we make **service invocation part of the language** in a transparent manner?
  - How do we exchange data between machines that use **different data representations**?
    - Data type formats (e.g. byte orders)
    - Data structures (need to be flattened and reconstructed)
  - How do we **find** the service? The client should not need to know where the server resides or even which server provides the request
  - How do we deal with **errors**?
- Solutions
  - To exchange data, an intermediate representation format is used. The concept of transforming data being sent to an intermediate representation format and back is referred to by different (equivalent) names:
    - Marshalling/un-marshalling
    - Serializing/de-serializing

3.2 Individual steps in RPC

- Inteface definition language IDL

    - All RPC systems come with a language that allows to describe services in an abstract manner (independently of the programming language used)
    - The IDL allows to define each service in terms of their names, and input and output parameters
    - Given an IDL specification, the interface compiler performs a variety of tasks to generate the stubs in a target programming language
        - Generate the client stub procedure for each procedure signature in the interface. The stub will be then compiled and linked with the client code
        - Generate a server stub. A server *main* can also be created with the stub and the dispatcher compiled and linked into it. The code can then be extended by the developer by writing the implementations of the procedures.
        - It might generate a *.h file for importing the interface and all the necessary constants and types.
- Binding
    - A service is provided by a server located at a particular IP address and listening to a given port.
    - Binding is the process of mapping a service name to an address and port that can be used for communication purposes
    - Binding can be done:
        - Locally: the client must know the name (address) of the host of the server
        - Distributed: there is a separate service (service location, name and directory services, etc) in charge of mapping names and addresses.
    - With a distributed binder, several general options are possible
        - REGISTER (exporting an interface): a server can register service names and the corresponding port
        - WITHDRAW: a server can withdraw a service
        - LOOKUP (importing an interface): a client can ask the binder for the address and port of a given service
    - There must be a way to find the binder (e.g. predefined location, configuration file)
    - Bindings are usually cached

3.3 Call semantics

- A client sends an RPC to a service at a given server. After a time-out expires, the client may decide to resend the request, if after several tries there is no success, what may have happened depends on the call semantics
    - **Maybe**: no guarantees. The procedure may have been executed (the response message was lost) or may not have been executed (the request message was lost)
    - **At least-once**: the procedure will be executed if the server does not fail, but it is possible that it is executed more than once. This may happen, for instance, if the client resends the request after a timeout. If the server is designed so that service calls are idempotent (produce the same outcome for the same input), this might be acceptable.

- **At most-once**: the procedure will be executed either once or not at all. Re-sending the request will not result in the procedure being executed several times. The server must perform some kind of duplicate detection.

3.4 Summary

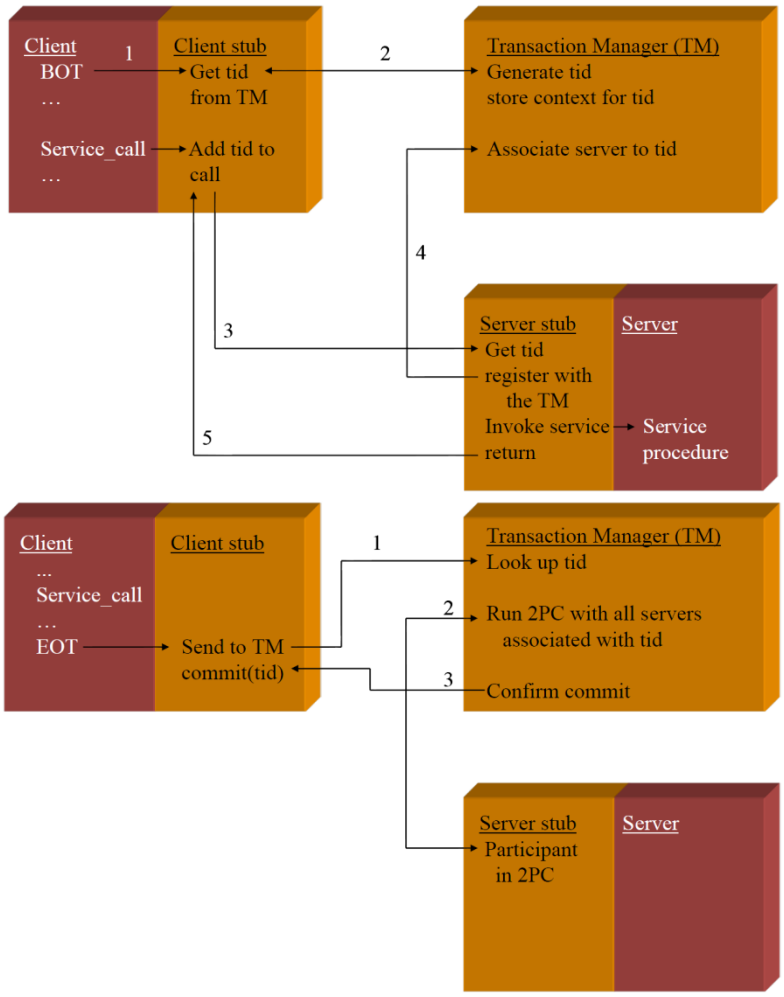
- Advantages
  - Implement distributed applications in a simple and efficient manner
  - RPC follows the programming techniques of the time (procedural languages)
  - RPC allows modular and hierarchical design of large distributed systems
    - Client and server are separate
    - The server encapsulates and hides the details of the back end systems (such as databases)
- Disadvantages
  - RPC is not a standard, it is an idea that has been implemented in many different ways
  - RPC allows building distributed systems, but does not solve many problems distribution creates. In that regard, it is only a low-level construct
  - Not very flexible, only one type of interaction: client/server

3.5 Distributed environments

- Context
  - When designing distributed applications, there are a lot of crucial aspects common to all of them. RPC does not address any of these issues
  - To support the design and deployment of distributed systems, programming and run time environments started to be created. These environments provide, on top of RPC, much of the functionality needed to build and run distributed applications
- Distributed computing environment (DCE)
  - standard implementation by the Open Source Foundation (OSF)
  - provides
    - RPC
    - Cell directory (sophisticated name and directory service)
    - Time for clock synchronization
    - Security (secure and authenticated communication)

3.6 Transactional RPC

- RPC was designed for one at a time interactions. This is not enough.
- This limitation can be solved by making RPC calls transactional. In practice this means that they are controlled by a 2 phase commit (2PC) protocol:
  - An intermediate entity runs the 2PC protocol, often called transaction manage (TM)


Client
BOT
…
Service_call
…
1
Client stub
Get tid
from TM
Add tid to
call
2
Transaction Manager (TM)
Generate tid
store context for tid
Associate server to tid
4
3
Server stub
Get tid
register with
the TM
Invoke service
return
Server
Service
procedure
5
Client
…
Service_call
…
EOT
Client stub
Send to TM
commit(tid)
1
Transaction Manager (TM)
Look up tid
2
Run 2PC with all servers
associated with tid
3
Confirm commit
Server stub
Participant
in 2PC
Server

4 Socket programming

4.1 Introduction

- A socket is a **host-local, application-created, OS-controlled interface** into which the application process can both send and receive messages to/from another (remote or local) application process.
- Socket programming can be used with both TCP and UDP
    - TCP
        - Client must contact server, i.e. the server process must first be running, and must have already created a socket that welcomes client's contact
        - TCP provides reliable, in-order transfer of bytes ("pipe") between client and server
    - UDP
        - No "connection" between client and server, i.e. no handshaking
        - UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

4.2 Comparison of Java and C API

- Java API
    - High-level, easy to use for common situations
    - Buffered I/O
    - Failure abstracted as exceptions
    - Less code to write
    - Focus: threads
- C API
    - Low-level, more code, more flexibility
    - Original interface
    - Maximum control
    - Basis for all other APIs
    - Focus: events

4.3 Java socket programming with TCP

```
// Java TCP client
Socket clientSocket = new Socket("hostname",6789);
DataOutputStream outToServer =
    new DataOutputStream(clientSocket.getOutputStream());

BufferedReader inFromServer =
    new BufferedReader  (new InputStreamReader(clientSocket.getInputStream()));

outToServer.writeBytes("message")
result = inFromServer.readLine();

clientSocket.close();
```

```
// Java TCP server
ServerSocket welcomeSocket = new ServerSocket(6789);

while (true) {
    Socket connectionSocket = welcomeSocket.accept();

    BufferedReader inFromClient =
        new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
    DataOutputStream outToClient =
        new DataOutputStream(connectionSocket.getOutputStream());

    String input = inFromClient.readLine();
    outToClient.writeBytes(f(input));
}
```

- Using this simple approach, one client can delay or even block other clients
- One solution: threads

```
// Java TCP server
ServerSocket welcomeSocket = new ServerSocket(6789);

while (true) {
    Socket connectionSocket = welcomeSocket.accept();

    ServerThread thread = new ServerThread(connectionSocket);
    thread.start(); // thread does the same steps as above
}
```

4.4 Java socket programming with UDP

```
// Java UDP client
DatagramSocket clientSocket = new DatagramSocket();
InetAddress IPAddress = InetAddress.getByName("hostname");

DatagramPacket sendPacket = new DatagramPacket(bytes,length,IPAddress,6789);
clientSocket.send(sendPacket);

DatagramPacket receivePacket = new DatagramPacket(bytes,length);
clientSocket.receive(receivePacket);

result = receivePacket.getData();

clientSocket.close();
```

```
// Java UDP server
DatagramSocket serverSocket = new DatagramSocket(6789);

while (true) {
    DatagramPacket receivePacket = new DatagramPacket(bytes,length);
    serverSocket.receive(receivePacket);
    InetAddress IPAddress = receivePacket.getAddress();

    int port = receivePacket.getPort();

    DatagramPacket sendPacket = new DatagramPacket(bytes,length,IPAddress,port);
    serverSocket.send(sendPacket);
}
```

4.5 C socket programming with TCP

- There are several steps to use TCP in C
    - Create a socket
    - Bind the socket
    - Resolve the host name
    - Connect the socket
    - Write some data
    - Read some data
    - Close and exit
- Servers use *listen(s,backlog)* on a bound, but not connected socket. After that, a call to *accept(s,addr,addr_len)* accepts connection requests

```
// C TCP server

#include <sys/socket.h>

// set up normal socket (incl binding), see client

// put the socket into listening mode
if (listen(ssock, MAX_PENDING) < 0) {
    printf("Putting socket into listen mode failed ...\n");
    return EXIT_FAILURE;
}

// run forever => accept one connection after each other.
while (1) {
    int csock = accept(ssock, 0, 0);
    do_something(csock);
    close(csock);
}
```

```
// C TCP client

#include <sys/socket.h>

// s is the socket descriptor
// AF_NET is the address familiy
// service type, e.g. SOCK_STREAM or SOCK_DGRAM
// protocola, 0 = let OS choose
int s = socket(AF_INET, SOCK_STREAM, 0)

// bind
struct sockaddr_in sa;
memset(&sa, 0, sizeof(sa));
sa.sin_family = PF_INET;
sa.sin_port = htons(0);
sa.sin_addr = htonl(INADDR_ANY);
if (bind (s, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
    perror("binding to local address");
    close(s);
    return -1;
}

// resolve hostnames
struct hostent* h;
h = gethostbyname(host)
if (!h || h->h_length != sizeof(struct in_addr)) {
    fprintf(stderr, "%s: no such host\n", host);
    return -1;
}

// connecting
struct sockaddr_in sa;
sa.sin_port = htons(port)
sa.sin_addr = *(struct sockaddr*)h->h_addr;
if (connect (s, (struct sockaddr*)&sa, sizeof(sa)) < 0 {
    perror(host);
    close(s);
    return -1;
}
```

- Sending and receiving data works through the commands *send(s,buf,len,flags)* and *recv(s,buf,len,flags)*

5 Reliable data transfer

- Reliable data transfer is implemented on the transport layer. It provides the application layer a reliable channel, and uses internally an unreliable channel from the layer below.

5.1 Incremental development of reliable data transfer (RDT)

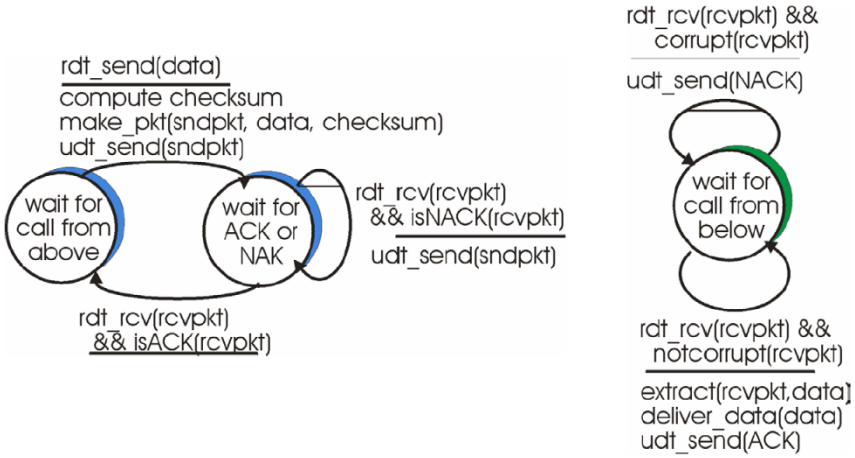
- Careful, the API used in the state diagrams is very counter-intuitive:
  - Sender side
    - *rdt_send*: called from above by the application.
    - *udt_send*: called by rdt, to transfer packet over unreliable channel
  - Receiver side
    - *deliver_data*: called by rdt to deliver data to the upper layer (i.e. application)
    - *rdt_rcv*: called when a packet arrives on the receiving side of a channel

5.1.1 RDT 1.0: reliable transfer over reliable channel

- Underlying channel is perfectly reliable => not much to do

5.1.2 RDT 2.0: channel with bit errors

- There is no packet loss, but channel may flip bits in packets
- How do we recover from errors?
  - Acknowledgements (ACKs): receiver explicitly tells sender that packet he received was ok
  - Negative acknowledgements (NAKs): receiver explicitly tells sender that packet he received had errors. The sender will then retransmit the packet
- New mechanisms
  - Error detection
  - Receiver feedback with control messages
  - Retransmission



5.1.3 RDT 2.1

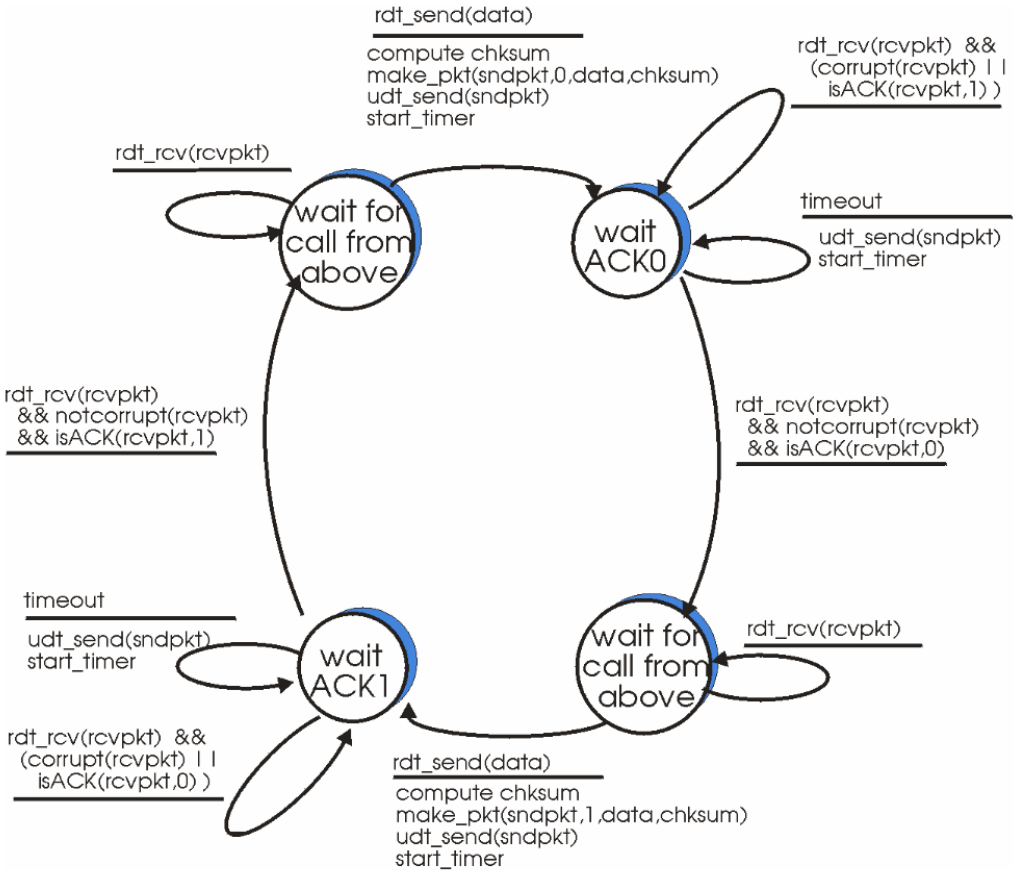
- RDT 2.0 has a fatal flaw: What happens if ACK/NAK is corrupted?
  - Sender does not know what happens at receiver
  - Retransmit might lead to duplicates
- Handling duplicates
  - Sender adds sequence number to each packet
  - Sender retransmits current packet if ACK/NAK garbled, and the receiver discards duplicate packets
- Two sequence numbers actually are enough. However, this leads to a duplication of states.

5.1.4 RDT 2.2: NAK-free

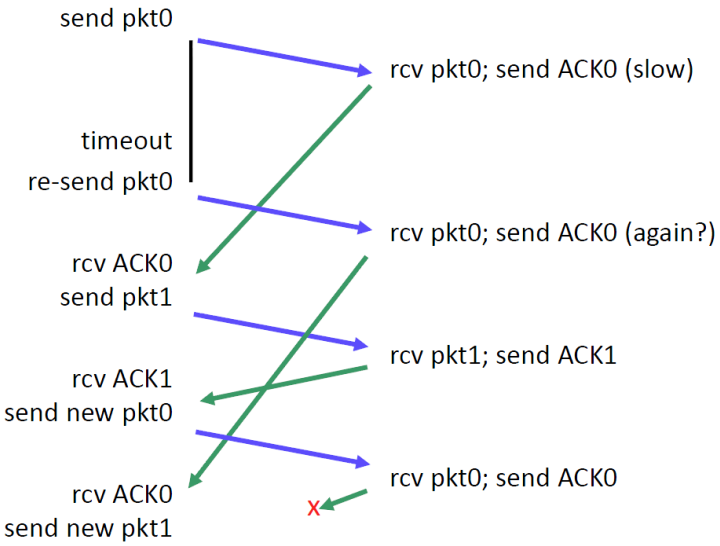
- Same functionality, but only using ACKs.
- Instead of NAK, receiver sends ACK for last packet received packet that was OK, i.e. the sequence number of the acknowledged package is included in the ACK message.
- If the sender sees duplicate ACKs, he needs to retransmit

5.1.5 RDT 3.0: channels with errors and loss

- Underlying channel can also lose packets (both data and ACKs)
- Sender waits “reasonable” amount of time for ACK, and retransmits if timeout occurs.



- Losing ACKs or premature timeouts are now handled fine. However, if the delay varies heavily, there might be problems, because we don't have a FIFO channel:

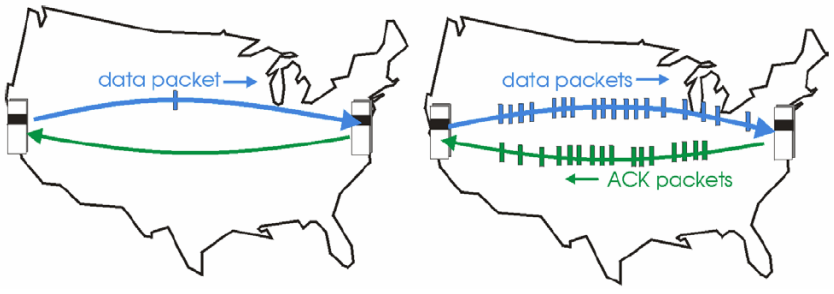


- Performance
  - Most of the time, the sender/receiver are just waiting. The utilization is very low, because we acknowledge every single message. If L denotes the size of a message and R the bandwidth, we get a utilization U of

$$U = \frac{\frac{L}{R}}{RTT + \frac{L}{R}}$$

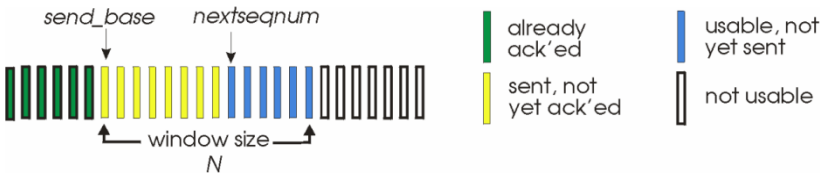
5.2 Pipelining

- The performance problem can be solved with pipelining, i.e. allowing multiple packets to be "in-flight"
  - Range of sequence numbers must be extended
  - Buffering at sender and/or receiver
  - Two variants: go-back-N and selective repeat



5.2.1 Go-Back-N

- Sender
  - Multiple-bit sequence number in packet header
  - "Window" of up to N consecutive unacknowledged packets allowed

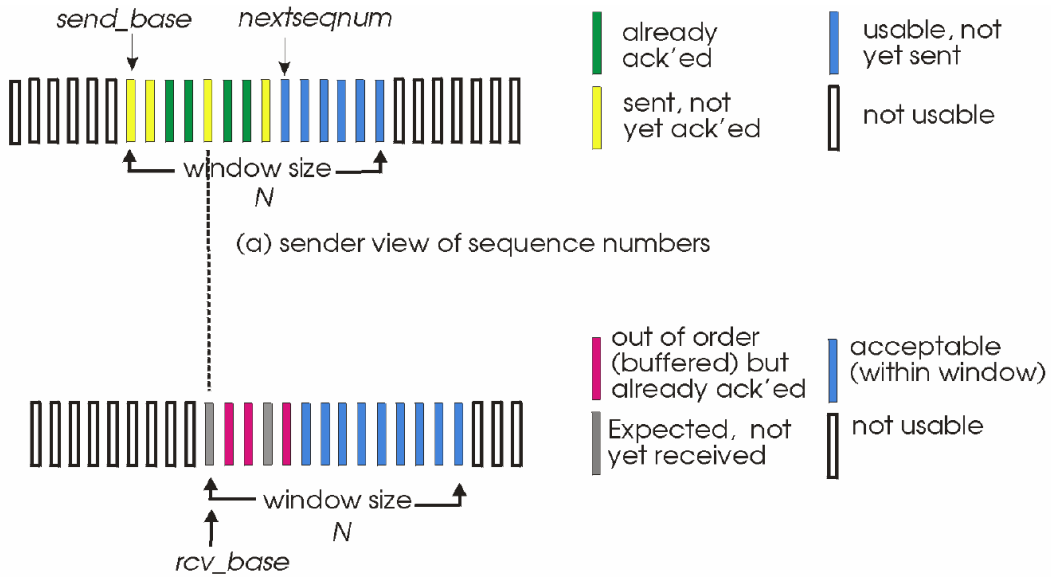


- ACK(n) acknowledges all packets up to and including sequence number n (i.e. cumulative ACK)
- There is a timer for each in-flight package
- timeout(n): retransmit packet n and all higher packets in window

- Receiver
  - ACK-only: always send ACK for correctly-received packet with highest in-order sequence number
    - May generate duplicate ACKs
  - Out-of-order packets are discarded (no receiver buffering), and highest package is re-acknowledged

5.2.2 Selective repeat

- Receiver individually acknowledges all correctly received packets, and also buffers packets for eventual in-order delivery to upper layer
- Sender only resends packets for which ACK has not been received (timer for each unacknowledged packet)



- Sender
  - *rdt_send*: If the next sequence number lies in the sender window, the data gets sent
  - *timeout(n)*: resend packet n, restart timer
  - *ACK(n)*
    - n must be in [sendbase,sendbase+N-1]
    - mark packet n as received, and if n is smallest unacknowledged packet, advance window base to next unacknowledged sequence number

- Receiver
    - *rdt_rcv(n)* with n in [rcvbase,rcvbase+N-1]
        - send *ACK(n)*
        - If the packet is in-order, deliver the packets and any packets that might be in the buffer, and advance the window to the next not-yet-received packet. Otherwise, if the packet arrives out-of-order: buffer.
    - *rdt_rcv(n)* with n in [rcvbase-N,rcvbase-1]
        - send *ACK(n)*
        - This is needed, because ACKs can be lost
    - otherwise ignore

6 Queuing theory

6.1 Notation

- To characterize a queuing system, the following notation is used: A/S/m/C/P/SD
  - A – arrival distribution
  - S – service distribution
  - m – number of servers
  - C – buffer capacity
  - P – population size (input)
  - SD – service discipline

6.1.1 Arrival rate distribution

- The interarrival times (time between two successive arrivals) are assumed to form a sequence of independent and identically distributed random variables, where the Poisson distribution is a common assumption.
- Mean interarrival time $E(\tau)$
- Mean arrival rate $\lambda = 1/E(\tau)$
- The arrival rate is assumed to be both state independent and stationary.

6.1.2 Service time per job

- The time it takes to process a job (not including the time it has been waiting) is denoted by s.
- Mean service rate $\mu = 1/E(s)$
- If there are m servers, the mean service rate is $m\mu$
- μ is sometime called throughput. However, this is only true in some cases
  - There are always jobs ready when a job finishes
  - No overhead in switching to new job
  - All jobs complete correctly
  - Service rate is both state independent and stationary

6.1.3 Service discipline

- FCFS: first come, first served (ordered queue)
- LCFS: last come, first served (stack)
- RR: round robin (CPU allocation to processes)
- RSS: random
- Priority based

6.1.4 System capacity

- The system (or buffer) capacity is the maximum number of jobs that can be waiting for service
- The system capacity includes both jobs waiting and jobs receiving service
- Often assumed to be infinite

6.1.5 Number of servers

- The service can be provided by one or more servers

- Servers assumed to work in parallel and independent, i.e. no interference
- Total service rate is aggregation of each individual service rate

6.1.6 Population

- The total number of potential jobs than can be submitted to the system, often assumed to be infinite
- In practice
  - Very large (e.g. number of clicks on a page)
  - Finite (e.g. number of homework submissions)
  - Closed system (output determines input)

6.2 Results

- Random variables
  - n number of jobs in the system
  - n_q number of jobs in the queue
  - n_s number of jobs receiving service
  - τ interarrival time
  - w time spent in the system
  - w_q time spent in the queue
  - s service time of a request

  - $n = n_q + n_s$
  - $w = w_q + s$
- Expected values
  - Mean arrival rate $\lambda = 1/E(\tau)$
  - Mean service rate $\mu = 1/E(s)$
  - Load $\rho = \lambda/(m\mu)$
  - $\rho = \lambda/m \cdot E(s)$
  - Utilization $U = 1 - p_0$
- Effective arrival rate for M/M/m/B
  - If the system buffer capacity is not infinite, at some point, packets are going to be dropped. In this case, the effective arrival rate λ' is used, where
$$\lambda' = \lambda \cdot (1 - p_B)$$
- Little's law
  - $E(n) = \lambda' \cdot E(w)$
  - $E(n_q) = \lambda' \cdot E(w_q)$
  - $E(n_s) = \lambda' \cdot E(s) = \lambda'/\mu$

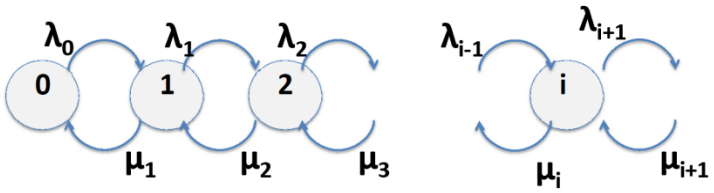
6.2.1 Birth-death-process

- Stochastic processes

    - Many of the values in a queuing system are random variables function of time. Such random functions are called stochastic process. If the values a process can take are finite or countable, it is a discrete process, or a stochastic chain
- Markov processes
    - If the future states of a process depend only on the current state and not on past states, the process is a Markov process. If the process is discrete, it is called Markov chain.
- Birth-death process
    - A Markov-chain in which the transition between states is limited to neighboring states is called a birth-death process

6.2.2 Steady state probability

- Consider the following state diagram



- Probability of being in state n is

$$p_n = \frac{\lambda_0 \lambda_1 \cdots \lambda_{n-1}}{\mu_1 \mu_2 \cdots \mu_n} p_0$$

6.2.3 M/M/1

- Memoryless distribution for arrival and service, single server with infinite buffers and FCFS
    - $p_n = \rho^n \cdot p_0$
    - $p_0 = 1 - \rho$
    - $p_n = (1-\rho)\rho^n$
    - $U = 1 - p_0 = \rho$
    - $E(n) = \rho/(1-\rho)$
    - $E(w) = \frac{E(n)}{\lambda} = \frac{1}{\mu \cdot (1-\rho)}$

6.3 Operational laws

- Operational laws are relationships that apply to certain measureable parameters in a computer system. They are independent of the distribution of arrival times or service rates.
- The parameters are observed for a finite time T_i, yielding operational quantities:
    - Number of arrivals A_i
    - Number of completions C_i
    - Busy time B_i

6.3.1 Job flow balance

- The job flow balance assumption is that the number of arrivals and completions is the same:

$$A_i = C_i$$

- The correctness of this assumption heavily depends on the semantics of “jobs competed”

6.3.2 Derivations

- Arrival rate $\lambda_i = A_i/T_i$
- Throughput $X_i = C_i/T_i$
- Utilization $U_i = B_i/T_i$
- Mean service time $S_i = B_i/C_i$
- Utilization law

$$U_i = \frac{B_i}{T_i} = X_i \cdot S_i$$

- Forced flow law
  - Assume a closed system, where several devices are connected as a queuing network. The number of completed jobs is C_0, and job flow balance is assumed.
  - If each job makes V_i (visit ratio) request of the ith device, then $C_i = C_0 V_i$

$$X_i = \frac{C_i}{T_i} = X \cdot V_i$$

  - Utilization of a device: $U_i = X_i S_i = X V_i S_i = X D_i$
  - Demand D_i of a device. The device with the highest demand is the bottleneck device.
- Little's law
  - R_i is the response time of the device, Q_i is the number of jobs in the device and $\lambda_i = X_i$ the arrival rate and throughput

$$Q_i = \lambda_i R_i = X_i R_i$$

- Interactive response time law
  - Users submit a request, when they get a response, they think for a time Z, and submit the next request.
  - Response time R, total cycle time is $R + Z$
  - Each user generates $T/(R+Z)$ requests in time T, and there are N users

$$X = \frac{Jobs}{Time} = \frac{N \frac{T}{R+Z}}{T} = \frac{N}{R+Z}$$

$$R = \frac{N}{X} - Z$$

7 Transport layer

- The transport layer provides logical communication between application processes running on different hosts
- Transport services
    - Reliable, in-order unicast delivery (TCP)
        - Congestion control, flow control, connection setup
    - Unreliable ("best-effort"), unordered unicast or multicast delivery (UDP)
        - Unicast: send to a specific destination
        - Multicast: send to multiple specific destinations
        - Broadcast: send to all destinations
- Services that are not available
    - Real-time/latency guarantees
    - Bandwidth guarantees
    - Reliable multicast
- Multiplexing
    - Gathering data from multiple application processes, enveloping data with header
    - Two types
        - Downward: several application can use the same network connection
        - Upward: Traffic from one application can be sent through different network connections
- Demultiplexing
    - Delivering received segments to correct application layer process
- Addressing based on source port, destination port and (source and destination) IP address

7.1 User datagram protocol UDP

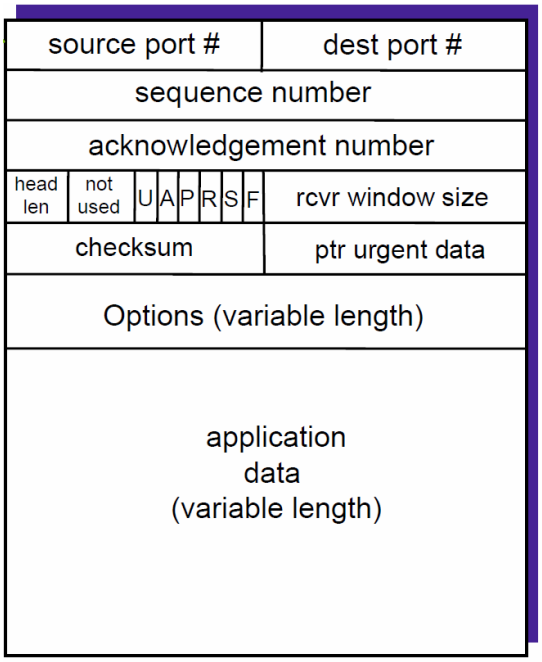
- "bare bones" internet transport protocol
- "best effort" service, segments may be lost or delivered out-of-order
- UDP is **connectionless**; no handshaking (can add delay). Each segment is handled independently of others
- Simple, no connection state at sender or receiver
- No congestion control, UDP can blast away as fast as desired
- Use cases
    - Multimedia streaming (loss tolerant, rate sensitive)
    - Reliable transfer over UDP, where reliability is implemented at the application layer
- UDP checksum
    - The sender treats the segment contents as a sequence of 16-bit integer and writes the 1's complement sum into the UDP checksum field
    - The receiver adds all 16-bit integers (including checksum) and if result is 11..1, no error has been detected

7.2 TCP

- Overview
  - Reliable, connection oriented byte-stream
    - Byte stream: no message boundaries
    - Connection oriented: point-to-point, 1 sender 1 receiver
    - Full-duplex: bidirectional data on one connection
  - Functionality
    - Connection management
      - Setup, teardown, protocol error handling
      - Multiplexing of connections over IP
      - End-to-end management over many hops
    - Flow control
      - Sender should not be able to overwhelm receiver
    - Reliability
      - Data is always delivered eventually, nothing is lost
      - Data is delivered in-order
    - Congestion control
      - Sender should not cause the internet to collapse

7.2.1 TCP segment layout

- Port numbers for multiplexing
- Sequence numbers for sliding windows (bytes, not segments)
- Flags
  - ACK: ack # valid
  - SYN: setup connection
  - FIN: teardown
  - RST: error
  - URG: urgent data
  - PSH push data
- Checksum (as in UDP)
- Receiving windows size for flow control



7.2.2 Sequence numbers

- Sequence numbers are the number of the first byte in segment's data
- ACKs contain the sequence number of the next byte expected from the other side. Also, the ACKs are cumulative.

7.2.3 TCP connection management

- Both client and server need to agree:

- Is there a connection at all? What state is it in?
- What sequence numbers shall we start with?
- When is the connection torn down?

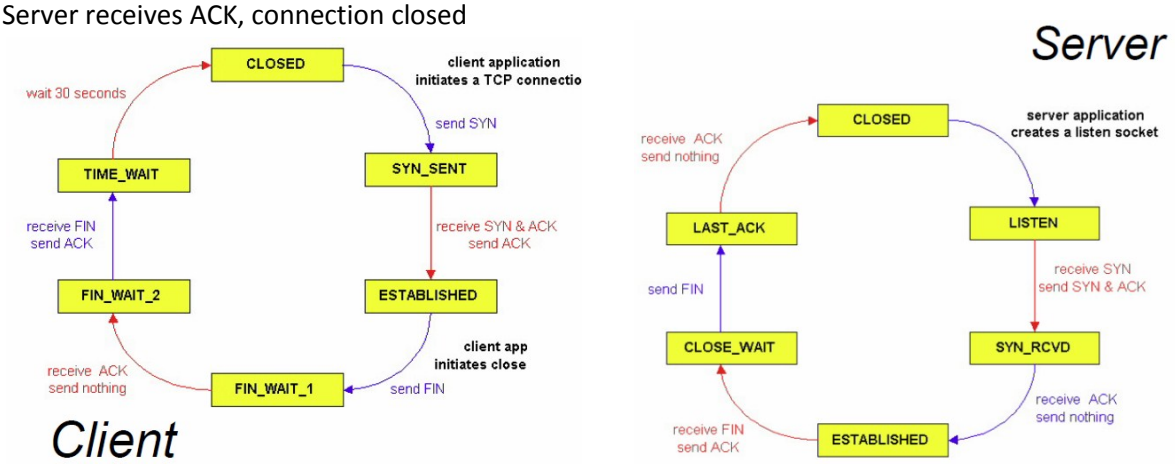
- What if either side misbehaves? How are lost packets handled?

7.2.3.1 Handshake

1. Client sends SYN with its sequence number
2. Server sends SYN+ACK of clients packet with another sequence number
3. Clients responds with ACK for servers sequence number
4. ACKs are always previous sequence number + 1 during the handshake, even if message does not contain any data

7.2.3.2 Connection teardown

1. Client sends FIN segment
2. Server replies with ACK. Closes connection, sends FIN
3. Client replies with ACK and enters “timed wait”, i.e. waits 30 seconds
4. Server receives ACK, connection closed



- The timed-wait is used to make sure, that any new and unrelated application that uses the same connection setting (i.e. port number) does not receive delayed packets of the already closed connection.

7.2.4 TCP reliability and flow control

- TCP ACK generation

Event	TCP receiver action
In-order segment arrival, no gaps, everything else already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK.
In-order segment arrival, no gaps, one delayed ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Out-of-order segment arrival, higher than expected sequence number, gap detected	Send duplicate ACK, indicating sequence number of next expected byte
Arrival of segment that partially or completely fills gap	Immediately ACK if segment starts at lower end of gap

- TCP flow control
    - Purpose: Sender won't overrun receiver's buffers by transmitting too much, too fast.
    - *RcvBuffer*: size of the TCP receive buffer
    - *RcvWindow*: amount of spare room in buffer
    - The receiver explicitly informs the sender of the (dynamically changing) amount of free buffer space via the *RcvWindow* field in the TCP segment
    - The sender keeps the amount of transmitted, but unacknowledged data less than the most recently received *RcvWindow*.

7.2.5 TCP round trip time and timeout

- The TCP timeouts should neither be too short, nor too long, but definitely longer than the RTT. However, the RTT will vary.
    - Too short
        - Premature timeout, resulting in unnecessary retransmissions
    - Too long
        - Slow reaction to segment loss
- Estimating the RTT
    - The sender measures the RTT in `SampleRTT` by taking the time between segment transmission and ACK receipt.
    - `EstimatedRTT = (1-α)·EstimatedRTT + α·SampleRTT`
        - Exponential weighted moving average
        - Influence of given sample decreases exponentially fast
        - Typical value for α is 0.125
- Timeouts
    - `Timeout = EstimatedRTT + 4·Deviation`
    - `Deviation = (1-β)·Deviation + β·|SampleRTT-EstimatedRTT|`
        - `EstimatedRTT` plus "safety margin", where the safety margin increases if the `EstimatedRTT` varies a lot.

7.2.6 Fast retransmit

- Time-out period is often fairly long, resulting in a long delay before lost packets are resent
- Lost segments can be detected via duplicate ACKs. In fact, if a segment is lost, it is likely that there will be many ACKs, as many segments are sent back-to-back.
- Solution/hack
    - If the sender receives 3 duplicate ACKs (i.e. 4 ACKs with the same number), the segment is assumed to be lost. A fast retransmit takes place, i.e. the sender resends the segment even if the timer has not yet expired.

7.2.7 Congestion and congestion control

- Purpose: too many sources sending too much data too fast for the *network* to handle.
    - Note that this concerns the network. Flow control on the other hand deals with problems of the receiver.
- Manifestations

    - Long delays due to queuing in router buffers
    - Lost packets due to buffer overflows at routers
- Approaches generally used for congestion control
    - Network assisted congestion control
        - Routers provide information to end systems, e.g.
            - Single bit indicating congestion
            - Explicit rate sender should send at
    - End-end congestion control
        - No explicit feedback about congestion from network
        - Congestion inferred from end-system observed loss, delay
        - Approach taken by TCP

7.2.7.1 End-to-end congestion control

- Detecting congestion in TCP
    - Long delays due to queues in routers that fill up. TCP sees estimated RTT going up
    - Packet losses due to routers dropping packets. TCP sees timeouts and duplicate ACKs
- "Probe" for usable bandwidth
    - Ideally, we would like to transmit as fast as possible without loss.
    - Increase rate until loss (congestion), decrease and start probing again
- Congestion window
    - In bytes, to keep track of current sending rate
    - Not the same as receiver window. The actual window used is the minimum of the two
- "Additive increase, multiplicative decrease"
    - Increase: linearly, when last congestions window's worth successfully sent
    - Decrease: Halve the congestion windows when loss is detected
- Congestion window details
    - TCP segments have a maximum segment size (MSS), determined by lower layer protocols
    - Increase window by MSS bytes, and never decrease to less than MSS bytes
    - W segments, each with MSS bytes sent in one RTT
        - $\text{throughput} = \frac{w \cdot \text{MSS}}{\text{RTT}}\ \text{Bytes/sec}$

7.2.7.2 TCP slow start

- Background
    - The start of TCP is fairly aggressive, but the algorithm used is called *slow start*. This has historical reasons, as in earlier version, the start was even faster.
- Start in *exponential growth phase* with a congestion windows size (CWND) of 1 MSS, and increase by 1 MSS for each ACK received. This behavior effectively **doubles** the windows size each round trip of the network.
- This continues until the first loss event occurs or CWND > threshold.
- After that, the *congestion avoidance* phase begins: Every CWND bytes that are acknowledged, CWND is increased by MSS. This results in linear growth per RTT.

8 Network layer

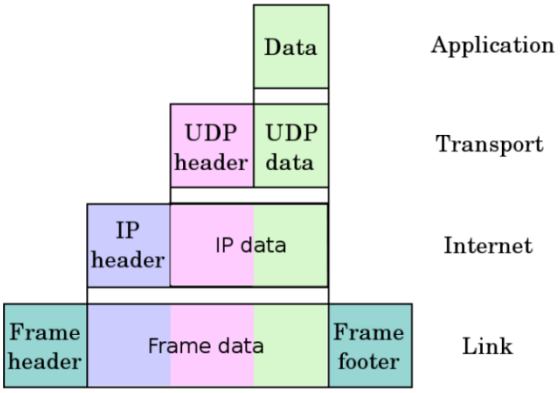
8.1 Network layer services

- Three important functions
  - **Path determination**: route taken by packets from source to destination, established by *routing algorithm*.
  - **Switching**: move packets from router's input to appropriate router output
  - **Call setup**: some network architectures require router call setup along path flows
- Service model
  - The service model defines the "channel" transporting packets from sender to receiver
    - Guaranteed bandwidth
    - Preservation of inter-packet timing (no jitter)
    - Loss-free/in-order delivery
    - Congestion feedback to sender
  - The network layer can work under two different models
    - Virtual circuit
    - Datagrams
- Virtual circuits
  - The source-to-destination path tries to behave like a physical circuit
  - The network layer maintains the illusion of a physical circuit:
    - Call setup for each call before data can flow, teardown afterwards
    - Each packet carries VC identifier (instead of the destination host ID)
    - Every router on source-destination path maintains state for each passing connection
    - Link, router resources (like bandwidth, buffers) may be allocated to VC
  - Signaling protocols
    - Used to setup, maintain and teardown VC
    - Used in ATM (asynchronous transfer mode), but not used in today's internet
- Datagram models (the internet model)
  - No call setup at network layer, routers maintain no state about end-to-end connections
  - Packets typically routed using destination host ID (and packets between same source/destination might take different paths)
  - Routing in a datagram model
    - Moving packets to their destination is done as a series of local routing decisions at each switch
    - Each switch maintains a forwarding or routing table that says which way packets have to go to reach a particular destination. This information in the routing table is gathered using a routing protocol
- Datagram versus VC network
  - Internet (IP)
    - Data exchange among computers, "elastic" service, no strict timing required

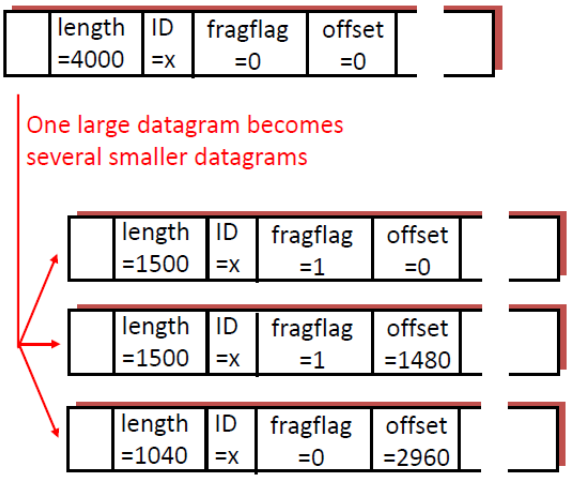
        - "Smart" end systems (computers) that can adapt, perform control and error recovery. Only simple inside the network, *complexity at the edge*.
        - Many link types with different characteristics
    - ATM
        - Evolved from telephony
        - Good for human conversation, strict timing, reliability requirements => need for guaranteed service
        - "Dumb" end systems (telephones), *complexity inside the network*.
        - Not really used today

8.2 Internet protocol IP

- Overview
    - Datagram based
        - Best effort, unreliable
        - Simple routers
        - Packet fragmentation and reassembly
    - Addressing scheme: IP Addresses
    - Routing protocols

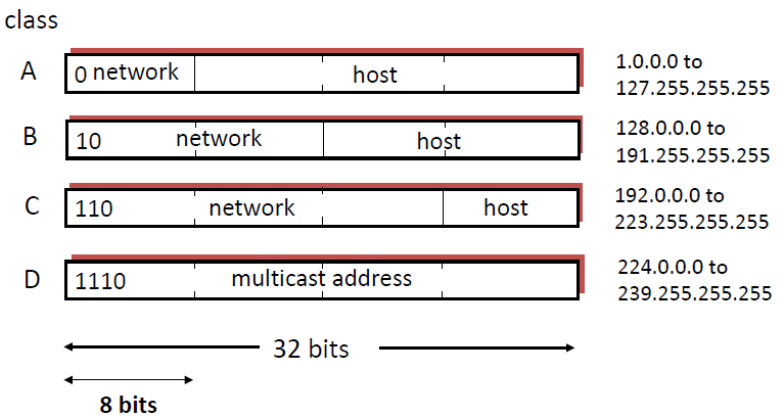


- Fragmentation and reassembly
    - IP needs to work over many different physical networks with different maximum packet size
    - Every network has **maximum transmission unit**, the largest IP datagram it can carry in the payload of a frame
    - Fragment when needed, reassembly only at destination
    - The fields "identifier", "flag" and "offset" are used to mark the fragments and reassemble them as needed



8.2.1 IP addresses

- "Class-full" addressing with three types of networks plus some reserved addresses



- Class A: Up to 126 networks (wide area)
- Class B: campus area
- Class C: local area

8.2.1.1 Classless interdomain routing CIDR

- Class-full addressing makes inefficient use of address space
- CIDR is an improvement over basic IP addressing for more efficient use of addresses by having network portions of arbitrary length
- Address format: `a.b.c.d/x`, where `x` is the number of bits defining the network portion of address

8.2.1.2 Address distribution

- How do hosts get an IP address?
  - Hardcoded by system admin
  - DHCP: dynamic host configuration protocol (seen later)
- How do networks get an address?
  - They get an allocated portion of the ISP's address space
- How does an ISP get a block of addresses?
  - From other (bigger) ISP
  - With ICANN: Internet corporation for assigned names and numbers
- Hierarchical addressing allows efficient advertisement of routing information: "Send me anything with addresses beginning 200.2.2.0/20"
  - What if an organization wants the switch to another ISP? The new ISP just adds a more specific advertisement, and the old discards traffic for the old addresses.

8.3 Additional protocols dealing with network layer information

8.3.1 Internet control message protocol ICMP

- Used by hosts, routers and gateways to communicate network-level information
  - Error reporting: unreachable host, network, port, protocol
  - Echo request/reply (used by ping)
- Network-layer "above" IP: ICMP messages carried in IP datagrams
- ICMP message: type, code and first 8 bytes of IP datagram causing error, examples include:

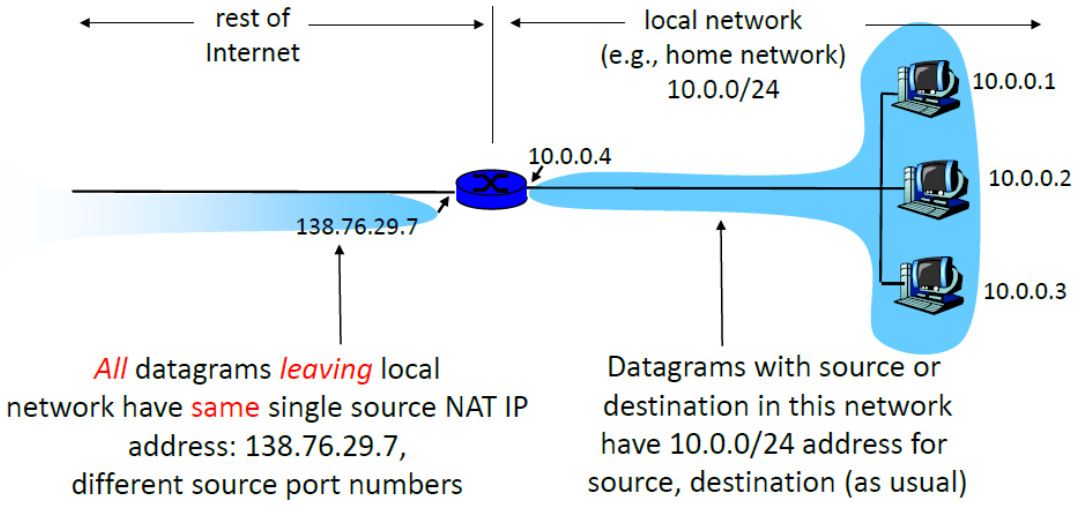
- Echo reply (ping)
- Destination network/protocol/port unreachable
- Destination network/host unknown
- Route advertisement, route discovery
- TTL expired

8.3.2 Dynamic host configuration protocol DHCP

- Goals
  - Allow hosts to *dynamically* obtain its IP address from network server when it joins the network
  - Can renew its lease on address in use
  - Allow reuse of addresses (only hold address while connected and “on”
  - Support for mobile users
- Overview
  - Dynamically get address (“plug-and-play”)
  - Host broadcasts “DHCP discover” message
  - DHCP server responds with “DHCP offer” message
  - Host request IP address: “DHCP request” message
  - DHCP server send address as “DHCP ack”

8.3.3 Network address translation NAT

- Datagrams with source or destintion in the LAN have 10.0.0/24 adderss for source and destination (as usual)
- All datagrams leaving the local network have the **same** single source NAT IP, only different port



- Motivation
  - Local network only uses one IP as far as the outside world is concerned
  - No need to allocate a range of addresses from the ISP
  - Change addresses of devices in local network without notifying the outside world
  - Devices inside local network are not directly addressable / visible by the outside world

        - Security plus
        - Machines cannot be servers
- NAT implementation: NAT router must
    - Outgoing datagrams: replace (source IP, port) of every outgoing datagram to (NAT IP, new port)
    - Remember (in network address translation table) every (source IP, port) to (NAT IP, new port) translation pair
    - Incoming datagram: replace (NAT IP, new port) in destination fields with corresponding (source IP, port) stored in NAT table
    - 16-bit port-number field allows 60000 simultaneous connections with a single LAN-side address
- NAT is controversial
    - Routers should only process up to layer 3
    - Violates end-to-end argument
        - NAT possibility must be taken into account by application designers, e.g. P2P applications
    - Address shortage of IPv4 should be solved by IPv6; NAT delays deployment of IPv6

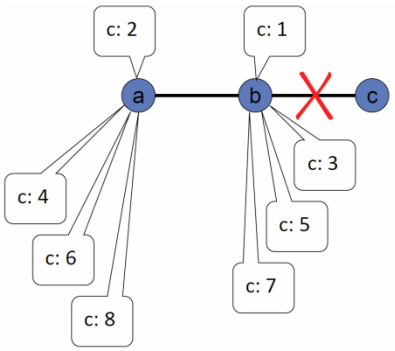
8.4 Routing

- Goal: Determine "good" path (sequence of routers) through network from source to destination
    - "good" typically means minimum cost path
- The network is abstracted as graph where the routers are nodes, and physical links are represented as edges
- Routing protocol classes
    - Distance vector protocol
        - Nodes know only distance (cost) to neighbors and exchange distance to all nodes with neighbors
        - Update local information base on received information
    - Link state protocols
        - All nodes know network topology and cost of each link (propagated through network by flooding)
        - Run protocol to find shortest path to each destination
- Important properties of routing protocols
    - Information needed (messages, storage necessary to keep information)
    - Convergence (how fast until it stabilizes, how fast it reacts to changes)

8.4.1 Distance vector protocols

- Every router maintains a *distance table* and a *routing table*
    - A node x has for each neighbor z an entry for each destination y: $D^x(y,z)$ is the distance from x to y through z. (*distance table*, $D^x(y,z)$)
    - The best route for a given destination y is marked (*routing table*, $D^x(y)$)
    - Formulas

        - $D^x(y) = \min_z D^x(y,z)$
        - $D^x(y,z) = c(x,z) + D^z(y)$
- Algorithm is
    - Iterative: continues until no nodes exchange information
    - Self-termination: no “signal” to stop
    - Asynchronous: nodes need not to iterate in lock-steps
    - Distributed: each node communicates only with direct neighbors
- Algorithm overview
    - Wait for a change in the local link cost or for a message from the neighbors (that says, that at least one of their least cost paths has changed
    - Recomputed distance table
    - If least cost path to any destination has changed, notify all neighbors
- Count to infinity problem
    - When a line breaks, it is possible that two nodes continuously change their costs, even if the destination is not reachable anymore at all



- Observations
    - “Good news travel fast”: if the local link cost decrease, this change is rapidly noticed in the whole network
    - “Bad news travel slow”: however, if the local link cost increase, a similar problem to count to infinity occurs. The network is rather slow until it stabilizes again.

8.4.1.1 Routing information protocol RIP

- Overview
    - Distance vector algorithm, included in BSD-Unix distribution in 1982
    - Distance metric: number of hops, maximum 15 hops
    - Distance vectors: exchanged every 30 seconds via response message (also called advertisement)
    - Each advertisement: route to up to 25 destination networks
- Link failure and recovery
    - If no advertisement is heard after 180 seconds, then the neighbor/link is declared as dead
        - Routes via neighbor invalidated
        - New advertisements sent to neighbors

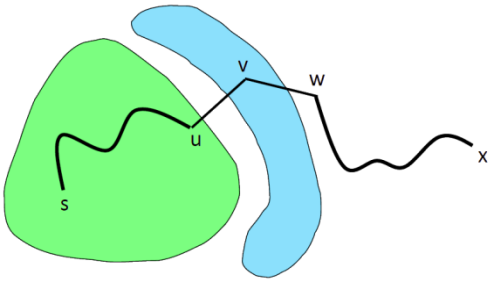
        - Neighbors in turn send out ne advertisements if their table changed
    - Link failure information quickly propagates to entire net, *poison reverse* used to prevent ping-pong loops
- Poison reverse
    - If z routes through y to get to x, z tells y, its (z's) distance to x is infinite (so y won't route to x via z)
    - Avoids the loop between two nodes
- RIP table processing
    - RIP routing tables managed by application-level process called route-d (daemon)
    - Advertisements sent in UDP packets, periodically repeated

8.4.1.2 (Enhanced) interior gateway routing protocol [E]IGRP

- CISCO proprietary successor of RIP
- Distance vector protocol
- Several cost metrics, such as delay, bandwidth, reliability, load, etc)
- Uses TCP to exchange routing updates
- Loop-free routing via distributed updating algorithm (DUAL) based on diffused computation

8.4.2 Link-state routing protocols

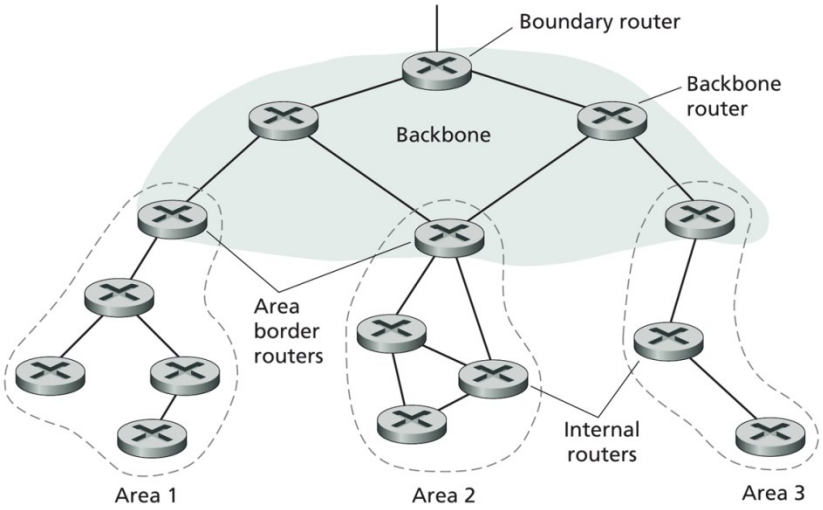
- Every node knows the topology and cost of every link. This is achieved through flooding:
    - Nodes send the information on their links and neighbors to all neighbors
    - Nodes forward information about other nodes to their neighbors
    - ACKs used to prevent message loss
    - Sequence numbers to compare versions
- With the information on topology and cost, the shortest path to every possible destination is calculated using Dijkstra's algorithm
- Dijkstra's algorithm
    - There are three groups of nodes in the network
        - Green nodes: we know the shortest path
        - Blue nodes: directly (with one edge) reachable from the green nodes
        - Black nodes: everything else



```
function Dijkstra(Graph, source):
    for each vertex v in Graph:           // Initializations
        dist[v] := infinity               // Unknown distance
        previous[v] := undefined          // Unknown previous node
    dist[source] := 0                     // Distance from source to source
    Q := the set of all nodes in Graph
    // All nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                 // The main loop
        u := vertex in Q with smallest dist[]
        if dist[u] = infinity:
            break           // all remaining vertices are inaccessible from source
        remove u from Q
        for each neighbor v of u:         // where v has not yet been removed from Q.
            alt := dist[u] + dist_between(u, v)
            if alt < dist[v]:             // Relax (u,v,a)
                dist[v] := alt
                previous[v] := u
    return dist[]
```

8.4.2.1 *Open shortest path first OSPF*

- Uses link state algorithm
    - Topology map at each node, route computation using Dijkstra's algorithm
- Advance features, not in RIP
    - Security: All messages are authenticated, therefore no malicious intrusion
        - TCP connections used
    - Multiple same-cost paths allowed (only one in RIP)
    - For each link, multiple cost metrics for different TOS (type of service)
        - E.g. a satellite link has low cost for best effort, but high cost for real time
    - Integrated uni- and multicast support
        - Multicast OSPF (MOSPF) uses same topology data base as OSPF
    - Hierarchical OSPF in large domains



        - Two-level hierarchy: local area or backbone
            - Link-state advertisement only in area

                - Each node has detailed area topology bt only knows direction (shortest path) to networks in other areas
            - Area border routers
                - "summarize" distance to networks in own area
                - Advertise to other area border routers
            - Backbone routers
                - Run OSPF routing limited to backbone
            - Boundary routers
                - Connect to other autonomous systems

8.4.3 Comparing routing algorithms

- Overview of distance vector (DV) versus link-state (LS)
    - DV: each node *talks only to its directly connected neighbors* but tells them *all it has learned* (distance to all nodes)
    - LS: each node *talks to all other nodes* but tells them only about the *state of its directly connected* links.
- Message complexity
    - LS: with n nodes and m links, the network is flooded with $O(nm)$ messages
    - DV: exchange between neighbors only, convergence time varies
- Speed of convergence
    - LS: $O(m + nlogn)$, but might have oscillations (e.g. if link costs depend on the amount of carried traffic)
    - DV: varies, count-to-infinity problem
- Robustness (what happens if a router malfunctions)
    - LS: Node can advertise incorrect **link** cost, each node computes only its own table
    - DV: Node can advertise incorrect **path** cost, each node's table is used by others, i.e. errors propagate through network.

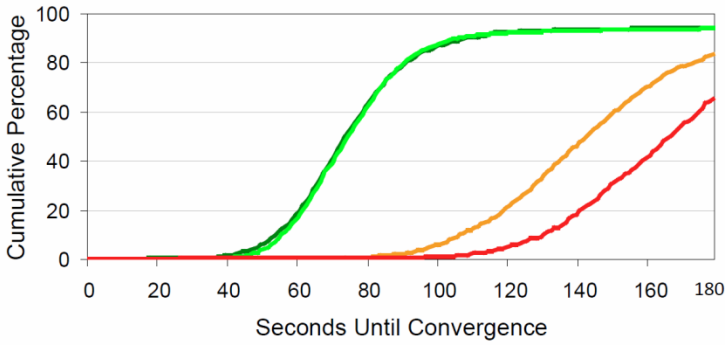
8.4.4 Interdomain routing

- In reality, the internet is a network of networks
    - Each network admin may want to control routing in own network
    - It is not possible to store 200 million destinations in one routing table; routing table exchange gets impossible
- Idea
    - Aggregate routers in groups, "autonomous systems" (AS)
        - Stub AS: small corporation
        - Multihomed AS: large corporation
        - Transit AS: provider
    - Routers in same AS run same routing protocol
        - "intra-AS" routing protocol
        - Routers in a different AS can run a different intra-AS routing protocol
    - Special gateway routers in AS's

        - Run intra-AS routing protocol with all other routers in AS
        - Run inter-AS routing protocol with other gateway routers

8.4.4.1 Border gateway protocol BGP

- BGP is *the* internet de-facto standard
- Path vector protocol
- Overview
    - Receive BGP update (announce or withdrawal) from a neighbor
    - Update routing table
    - Does the update affect active route? If yes, send update to all neighbors that are allowed by policy
- Details
    - BGP messages exchanged using TCP
    - BGP messages
        - OPEN: opens a TCP connection to peer and authenticates sender
        - UPDATE: advertises new path (or withdraws old)
        - KEEPALIVE: keeps connection alive in absence of updates. Also acknowledges OPEN request
        - NOTIFICATION: reports errors in previous messages, also used to close connection
    - Policy
        - Even if two BGP routers are connected, they may not announce all their routes or use all the routes of the other; e.g., if AS A does not want to route traffic of AS B, it can simple not announce anything to AS B.
    - Count-to-infinity
        - Not a problem, because the whole paths are announced, instead of only the directions and distances
- BGP convergence



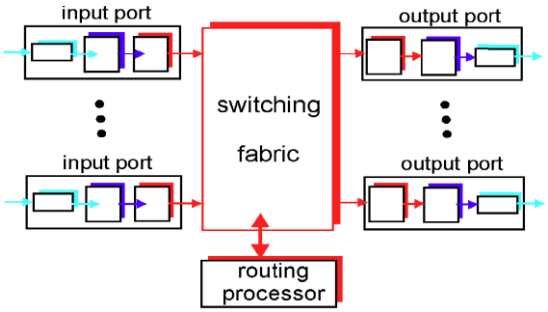
    - If a link comes up, the convergence time is in the order of time to forward a message on the shortest path
    - If a link goes down, the convergence time is in the order of time to forward a message on the longest path

- Possible solution: Include “cause tag” to the withdrawal message identifying the failed link/node. This solves the problem, but:
  - BGP is widely deployed, hard to be changed
  - ISP’s/AS’s don’t like the world to know that it is their link that is not stable
  - Race conditions

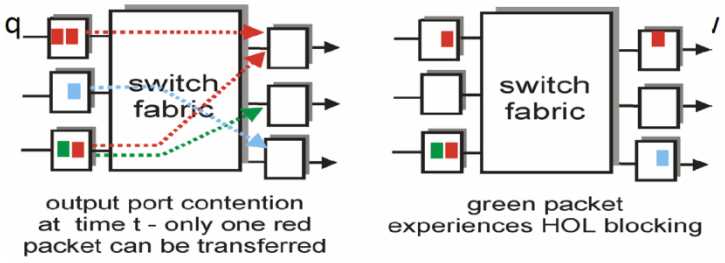
8.4.4.2 Why are intra- and inter-AS routing different

- Policy
  - Inter-AS: admin wants control over how traffic is routed and who routes though its network
  - Intra-AS: single admin, so no policy decisions needed
- Scale: hierarchical routing saves table size, reduces update traffic
- Performance
  - Intra-AS: can focus on performance
  - Inter-AS: policy may dominate over performance

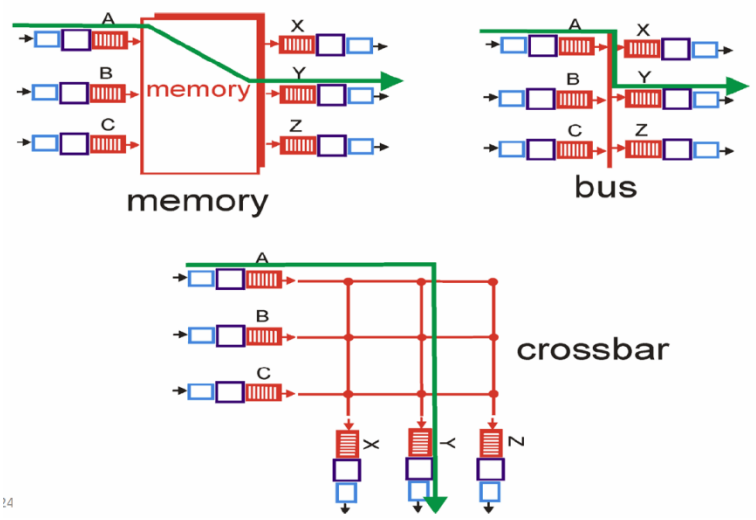
8.4.5 Routers



- Two key router functions
  - Run routing algorithms/protocols (RIP, OSPF, BGP)
  - Switch datagrams from incoming to outgoing link
- Input port function
  - Goal: complete input port processing at “line speed”; queuing if datagrams arrive faster than forwarding rate into switch fabric
- Input port queuing
  - Fabric slower than input ports combined, queuing may occur at input queues
  - Head-of-the-line (HOL) blocking: queued datagram at front of queue prevents others in queue from moving forward



  - Three types of switching fabrics

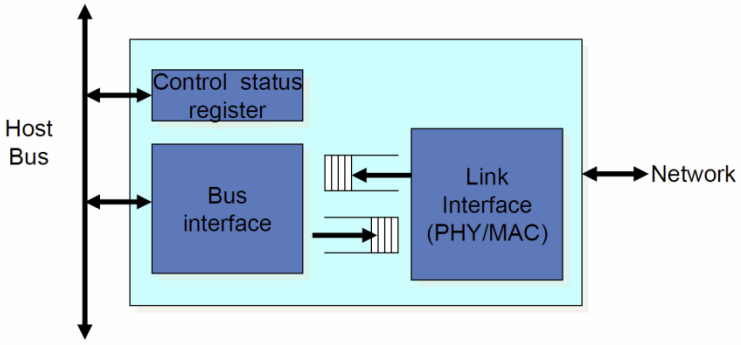


8.5 IPv6

- Motivation
  - 32-bit address space almost completely allocated
  - Header format helps speed processing/forwarding
  - Header changes to facilitate QoS (quality of service)
  - New “anycast” address: route to “best” of several replicated servers
- IPv6 datagram format
  - Fixed-length 40 byte header
  - No fragmentation allowed
- Transition from IPv4 to IPv6
  - Not all routers can be upgraded simultaneously
  - Two proposed approaches
    - Dual stack
      - Some routers with dual stack (both v4 and v6) can “translate” between formats
    - Tunneling
      - IPv6 carried as payload in IPv4 datagram among IPv4 routers

9 Link layer

- Overview
  - Two physically connected devices (host-router, router-router, host-host)
  - Unit of data is called a **frame**
- Network interface controller



- Link layer service: you can't just send IP datagrams over the link
  - Encoding: how to represent the bits (optically, electrically)?
  - Framing: where do the datagrams start and end?
  - Error detection: How to know if the datagram is corrupt?
  - Flow control: can we adjust the rate properly?
  - Link access: What if 3 or more machines share a link?

9.1 End-to-end argument

The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system.

Therefore, providing that questioned function as a feature of the communication system itself is not possible.

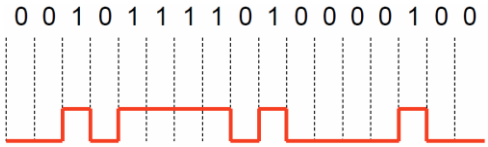
(Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

9.2 Encoding

- The problem
  - Suppose you can send high and low discrete signals: How do you send (represent) bits?

9.2.1 Non-return to zero NRZ

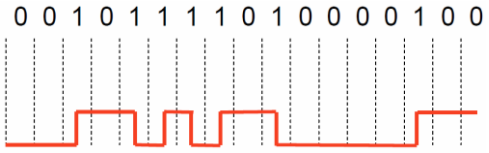
- Send 1 as high, and 0 as low.
- Problems
  - Baseline wander
  - No clock recovery



9.2.2 Non-return to zero inverted

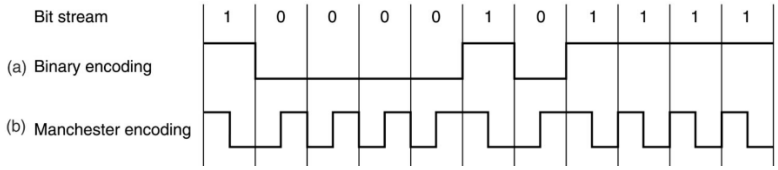
- Signal transition for a 1, stays for a 0

- Solves half the problem, long series of 1's are ok now
- Long runs of 0's still an issue



9.2.3 Manchester encoding

- Encode 1 as a transition from high to low, and 0 as a transition from low to high
  - No loss of sync, as each bit has a transition
  - Requires double the frequency (baud rate)



9.2.4 4B/5B encoding

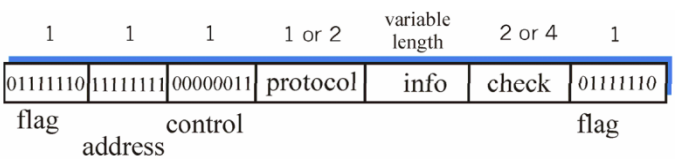
- Key idea: break up long sequences by inserting bits
- Encode every 4 symbols as 5 bits
  - 0 or 1 leading zero
  - ≤2 trailing zeros
- Send using NRZI
  - 80% efficiency vs. 50% for Manchester encoding
- Can use other symbols for control information

9.3 Framing

- Where do frames start and end?

9.3.1 Point to point protocol PPP

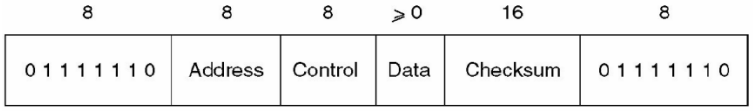
- Example of *byte-level framing*
- Byte-oriented protocol: a frame is a collection of bytes
- Bidirectional unicast link (i.e. only two machines)
- No sharing (no media access control or explicit MAC addressing)
- Design requirements
  - Packet framing
  - Bit transparency: must carry any bit pattern in the data field
  - Error detection (no correction)
  - Connection liveness: detect, signal link failure to network layer
  - Network layer address negotiation: endpoints can learn/configure each other's network address
- PPP data frame
  - Flag as delimiter
  - Address and control do nothing
  - Protocol: upper layer protocol to which frame is delivered



    - Info: data being carried
    - Check: CRC for error detection
- Byte stuffing in PPP
    - Problem: data transparency requirement says that any data must be allowed, including `01111110`, which would be interpreted as flag
    - Solution
        - Sender adds (stuffs) extra `01111110` byte after each `01111110` data byte
        - Receiver
            - Two `01111110` bytes in a row: discard one, continue data reception
            - Single `01111110`: that's the flag byte

9.3.2 High-level data link control HDLC

- Example of *bit-level framing*
- Somewhat similar to PPP, especially frame layout



- Bit-oriented protocol
- Bitstuffing for framing
    - Sender inserts a 0 after 5 consecutive 1's (except in flag)
    - Receiver interprets 0111111 as either flag or error

9.3.3 Synchronous optical network

- Example of *clock-based framing*
- *The* dominant standard for long-distance data transmission over optical networks
- No bit- or byte-stuffing
    - Frames are all the same size: 125 µs (actual data length depends on bandwidth)
- Framing is clock-based
    - Flag word: first 16 bits of each frame
    - Receiver looks for regular flag every 125 µs
- Encoding: NRZ but scrambled (XOR-ed with 127 bit pattern)

9.4 Error detection

- EDC = error detection and correction bits (redundancy)
- D = data protected by error checking, might include header fields
- Error detection not 100% reliable

9.4.1 Parity checking

- Single bit parity
    - E.g. 1-bit odd parity: parity bit is 1, if the number of 1's is even
- Two-dimensional bit parity
    - Detect and correct single bit errors

9.4.2 Cyclic redundancy check CRC

- Polynomials with binary coefficients
- Let whole frame (i.e. D+EDC) be the polynomial $T(x)$
- Idea: fill EDC (i.e. CRC) field such that $T(x) \bmod G(x) = 0$
    - This can be done by filling EDC with the remainder when dividing $T(x)$ with EDC = 00..0 by $G(x)$.

9.5 Media access control

- Three types of links
    - Point-to-point (single wire, e.g. PPP)
    - Broadcast (shared wire or medium, e.g. Ethernet or WLAN)
    - Switched (e.g. switched Ethernet, ATM)
- Multiple access protocols
    - Ideal features with a channel of rate R
        - When only one node has data to send: throughput of R
        - When M nodes have data to send: throughput of R/M
        - Decentralized protocol

9.5.1 Turn-taking protocols (e.g. round robin)

- No master node
- Token-passing protocol
    - Station k sends after station k-1
    - If a station needs to transmit, when it receives the token, it sends up to a maximum number of frames and then forwards the token
    - If a station does not need to transmit, it directly forwards the token

9.5.2 Random access protocols

- When node has a packet to send, it is transmitted at full channel rate; no a priori coordination among nodes
- When two or more nodes send at the same time, a *collision* occurs
- Random access MAC protocol specifies
    - How to detect collisions
    - How to recover from collisions

9.5.3 Slotted Aloha

- Time is divided into equal size slots (a slot is equal to the packet transmission time)
- Node with new arriving packets transmit at the beginning of the next slot
- If a collision occurs, the packet is retransmitted in future slot with probability p, until success
- Efficiency: $\frac{1}{e}R = 0.37R$ bps
- Comparison with round robin
    - Slotted aloha does not use every slot of the channel; the round robin protocol is better in this respect

    - Slotted aloha on the other hand is much more flexible, e.g. when new nodes join
- Variation: Adaptive slotted aloha
    - Idea: change the access probability with the number of stations
    - The number of stations in the system get estimated
        - If you see nobody sending, increase p
        - If you see more than one sending, decrease p

9.5.4 Pure (unslotted) aloha

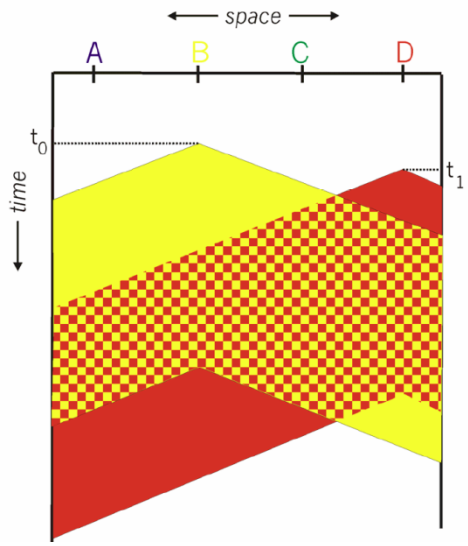
- No time slots: simpler, no synchronization
- Packet needs transmission: send directly (no waiting)
- Collision probability increases, because packets can partially overlap
- Efficiency is half the rate of slotted aloha

9.5.5 Demand assigned multiple access DAMA

- Channel efficiency only 37% for slotted aloha
- Practical systems therefore use reservation whenever possible
- But: every scalable system needs an aloha-style component
- Reservation
    - A sender reserves a future time-slot
    - Sending within this reserved time-slot is possible without collision
    - Reservation also causes higher delays

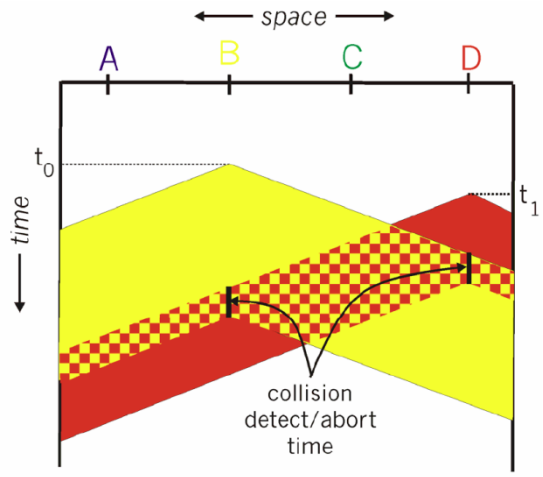
9.5.6 Carrier sense multiple access CSMA

- Idea: listen before transmitting
    - If channel sensed idle: transmit entire packet
    - If channel sensed busy: defer transmission. Two variants
        - Persistent CSMA: retry immediately with probability p when channel becomes idle (may cause instability)
        - Non-persistent CSMA: retry after random interval
    - Analogy: don't interrupt anybody already speaking
- Collision still can occur: two nodes may not yet hear each others transmission
    - Entire packet transmission time wasted



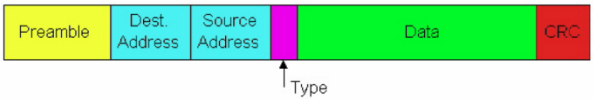
9.5.7 CSMA/CD (collision detect)

- Carrier sensing as in CSMA
    - Collision detected within short time
    - Colliding transmission get aborted, reducing channel wastage



9.5.8 Ethernet

- The predominant LAN technology
  - Cheap
  - First widely used LAN technology
  - Keeps up with the speed race
- Ethernet frame structure
  - Sending adapter encapsulates IP datagram (or other network layer protocol packet) in Ethernet frame
  - Preamble
    - 7 bytes with pattern 10101010, followed by 1 byte with 10101011
    - Used to synchronize receiver/sender clock rates
  - Addresses
    - 6 bytes, frame is received by all adapters on a LAN and dropped if address does not match
  - Type: indicates the higher layer protocol, mostly IP
  - CRC: checked at receiver, if error is detected, the frame is simply dropped



- Ethernet CSMA/CD algorithm
  1. Adapter gets datagram from network layer and creates frame
  2. If adapter senses channel idle, it starts to transmit frame. If it senses channel is busy, waits until channel is idle and then transmits.
  3. If adapter transmits entire frame without detecting another transmission, the adapter is done with the frame
  4. If adapter detects another transmission while transmitting, aborts and sends jam signal
  5. After aborting, adapter enters exponential backoff: after the nth collision, adapter chooses a K at random from $\{0,1,2,\ldots,2^m-1\}$ where $m = min(n,10)$. Adapter waits $K \cdot 512$ bit times and returns to step 2.

- Details of Ethernet's CSMA/CD
    - Jam signal: make sure all other transmitters are aware of collision
    - Bit time: depends on some things
    - Exponential backoff
        - Goal: adapt retransmission attempts to estimated current load
            - Heavy load: random wait will be longer
        - First collision chose K from $\{0,1\}$
        - After second collision, choose K from $\{0,1,2,3\}$
        - After ten (or more) collisions, chose K from $\{0,1,2,3,4,\ldots,1023\}$

10 Packet switching

10.1 Virtual circuit switching

- Every frame has a VCI (virtual circuit identifier)
    - VCI has link-local scope, not global
- Every switch has a forwarding table that maps *incoming(VCI,port)* to *outgoing(VCI,port)*
- Properties
    - One RTT of delay before first data can be sent
    - Connection request has full (global) address for destination, but data packets don't (less overhead)
    - If a link or switch fails, connection must be set up again
    - Routing is also needed to plan the path in the first place
    - Congestion can be prevented in advance by
        - Allocating buffers to each circuit (since state)
        - Admission control: don't allow connection setup
        - Billing: charging per connection

10.1.1 Asynchronous transfer mode ATM

- Connection oriented (signaled), packet-switched
- Unusual feature: fixed size, small data units (cells, 53 bytes)
    - Easier to build a switch to handle fixed-size frames
        - Particularly in parallel
    - Cell queues are shorter
        - Total switching time reduces (can switch with only 53 bytes) => end-to-end latency and jitter reduced (telephony)
    - Traffic delay is reduced
- ATM vs. IP
    - Historically, huge religious war
    - End-to-end: ATM lost to IP
    - In the LAN: ATM lost to Ethernet
    - Major uses today
        - Carrying IP traffic for virtual private networks
            - Guaranteed service, isolation
        - Link layer for DSL

10.2 Datagram switching

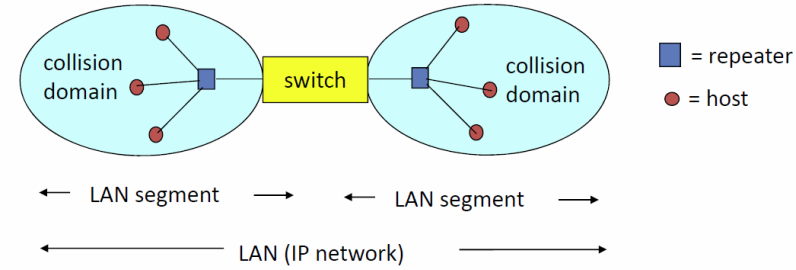
- Switch forwards based on MAC address
    - Fixed for all time in ROM with global scope (unique)
    - MAC addresses are flat (instead of hierarchical) => portability
    - 6-octet MAC address allocation administered by IEEE
- Forwarding table maps destination addresses to outgoing ports

10.2.1 Address resolution protocol ARP

- Goal: determine MAC address of B given B's IP address
- Each IP node (host or router) has an ARP table that maps IP addresses to (MAC; TTL)
- TTL (time to live): time after which address mapping will be forgotten (typically few minutes)

10.2.2 Bridges and switches

- Switches
    - Store and forward frames (e.g. Ethernet)
    - Examines frame header and selectively forward frame based on MAC destination address
    - Transparent
        - Hosts are unaware of presence of switches
    - Plug-and-play, self-learning
        - Switches do not need to be configured
- Traffic isolation
    - Switch installation breaks LAN into LAN segments
    - Switches filter packets
        - Same-LAN-segment not usually forwarded onto other LAN segments
        - For CSMA/CD networks: separate collision domains

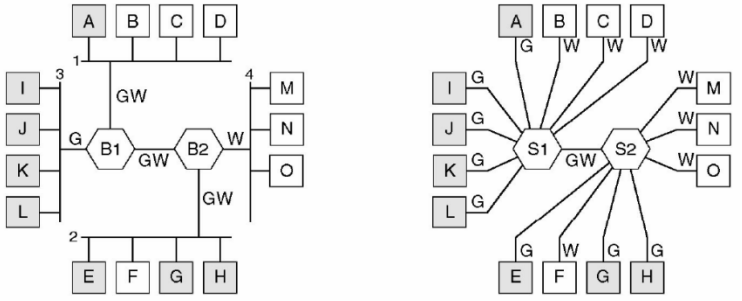


- Forwarding
    - Problem: how to determine to which LAN segment to forward frames.
    - Self-learning
        - A switch has a MAC address table
            - Entry (Node MAC address, switch port, time stamp)
            - Stale entries in table dropped (TTL can be 60 minutes)
        - Switches learn which host can be reached through which interface
            - When frame received, switch learns location of sender and records sender/location pairs in switch table
    - Filtering/Forwarding
        - When switch receives a frame, it indexes the switch table using the MAC destination address
        - If an entry is found: If the destination port is the same as the incoming port of the frame, the frame is dropped. Otherwise the frame is sent out to the destination port
        - If no entry is found: Forward on all but the port on which frame arrived

    - Problem: Loops (solved using spanning trees)
- Spanning tree algorithm from bridges
    - Spanning tree of the graph with networks as nodes and switches as edges
    - Bridges elect leader (root)
        - Broadcast serial number, pick lowest
    - Construct tree based at the root
        - Follow links out from root
    - Packets forwarded along resulting spanning tree
    - Continuous tree maintenance
    - Allows redundant paths for fault tolerance

10.2.3 Virtual LANs

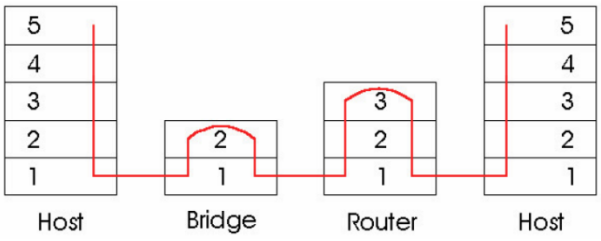
- Key idea: Make a set of switches look like a larger set of switches



- Route IP traffic between virtual LANs
- Why?
    - Security: isolate networks without separate wiring
    - Limit broadcasting (even switches forward all broadcasts)
    - Rapidly reconfigure network connections in software
- How?
    - Switch assigns each port to different VLAN
    - Switch assigns each MAC address to a VLAN
    - VLAN based on packet classification (e.g. protocol)
    - Explicit tag in each packet
- How are VLANs connected? IP routing.

10.2.4 Switches vs. Routers

- Both store-and-forward devices
    - Routers: network layer device (examine network layer headers)
    - Switches are link layer devices
- Routers maintain routing tables, implement routing algorithms
- Switches maintain MAC tables, implement filtering, learning and spanning tree algorithms



- Switches
  - + Switches operation is simpler, requiring less packet processing
  - + Switch tables are self learning
  - – All traffic confined to spanning tree, even when alternative bandwidth is available
  - – Switches do not offer protection from broadcast storms
- Routers
  - + arbitrary topologies can be supported, cycling is limited by TTL counters (and good routing protocols)
  - + provide protection against broadcast storms
  - – require IP address configuration (not plug-and-play)
  - – require higher packet processing
- Summary
  - Switches do well in small topologies, while routers are used in large networks (complex/wide topologies)

11 Naming

11.1 Introduction

- Binding
  - The association between a name and a value is called a **binding**
  - In most cases, the binding isn't immediately visible
    - Often conflated with creating the value itself
  - Sometimes bindings are explicit, and are objects themselves
- A general model of naming
  - Designer creates a **naming scheme**
    - Name space: What names are valid?
    - Universe of values: What values are valid?
    - Name mapping algorithm: What is the association of names to values?
  - Mapping algorithm also known as a resolver
  - Requires a context
- Context
  - Any naming scheme must have at least one context, though it might not be stated

11.2 Naming operations

- Resolution
  - Value = context.RESOLVE(name)
- Managing bindings
  - Status = BIND(name, value, context)
  - Status = UNBIND(name, context)
  - May fail according to naming scheme rules
  - Unbind may need a value, too
- Enumeration
  - List = ENUMERATE(context)
  - Not always available, but returns all bindings (or names) in a given context
- Comparing names
  - Result = COMPARE(name1, name2)
  - Requires definition of equality
    - Are the names themselves the same?
    - Are they bound to the same object?
    - Do they refer to identical copies of one thing?

11.3 Naming policy alternatives

- How many values for a name?
  - If one, mapping is injective or "one-to-one"
    - Car number plates, virtual memory address
  - Multiple values for the same name
    - Phone book (multiple phone numbers per person)

- How many names for a value?
    - Only one name for each value
        - Names of models of car
        - IP protocol identifiers
    - Multiple names for the same value
        - Phonebook (people sharing phone)
- Unique identifier spaces and stable bindings
    - At most one value bound to a name
    - Once created, bindings can never be changed
    - Useful: can always determine identity of two objects
        - Social security numbers

11.4 Types of lookups

- Table lookup
    - Simplest scheme; like a phone book
    - Examples
        - Processor registers are named by small integers
        - Memory cells named by number
        - IP addresses named by DNS names
        - Unix sockets are named by small integers
- Recursive lookup (pathnames)
- Multiple lookups (search paths)

11.5 Default and explicit contexts, qualified names

- Where is the context?
    - Default (implicit): supplied by the resolver
        - Constant: built in to the resolver
            - Universal name space, e.g. DNS (context is the DNS root server)
        - Variable: from current environment (state)
            - E.g. current working directory
    - Explicit: supplied by the object
        - Per object
            - Context is a name, sometimes called base name
        - Per name (qualified name)
            - Each name comes with its context, or rather the name of the context
                - (context,name) = qualified name
            - Recursive resolution process
                - Resolve context to a context object
                - Resolve name relative to resulting context
            - Examples
                - troscoe@inf.ethz.ch or /var/log/syslog

11.6 Path names, naming networks, recursive resolution

- Path names
    - Recursive resolution => path names
    - Recursion must terminate
        - Ether at a fixed, known context reference (the root)
        - Or at another name, naming the default context (e.g. relative pathnames)
- Soft links
    - So far, names resolve to values
        - Values may be names in a different naming scheme (usually are)
    - Names can also resolve to other names in the *same* scheme
        - Unix symbolic links, windows shortcuts
        - Forwarding addresses (WWW, Email)

11.7 Multiple lookup

- Sometimes, one context is not enough: Multiple lookup, or “search path” (try several context in order)
- Union mounts: overlay two or more contexts
- Examples
    - Binary directories in Unix
    - Resolving symbols in link libraries
- Somewhat controversial

11.8 Naming discovery

- How to find a name in the first place?
    - Many options
        - Well known
        - Broadcast the name
        - Query
        - Broadcast the query
        - Resolve some other name to a name space
        - Physical rendezvous
    - Often reduces to another name lookup

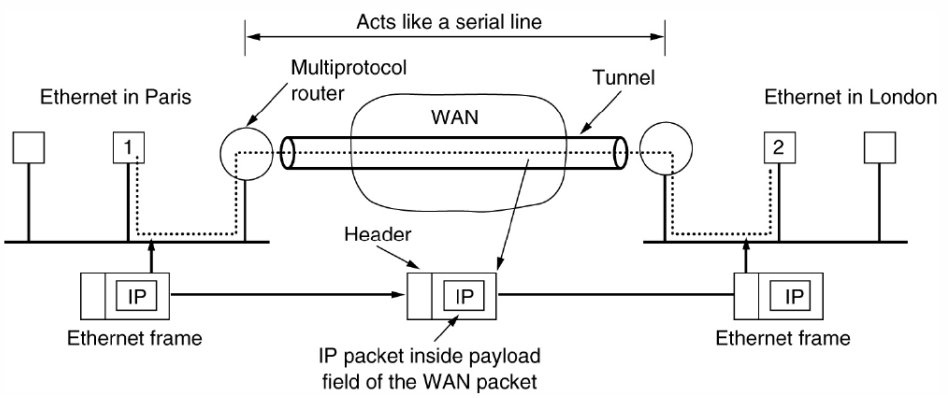
12 Virtualization

- Goal: create the illusion of a “real” resource (processor, storage, network, links)
- How?
    - Multiplexing: divide resources up among clients
    - Emulation: create the illusion of a resource using software
    - Aggregation: join multiple resources together to create a new one
    - Combination of the above
- Why?
    - Sharing: enable multiple clients to use a single resource
    - Sandboxing: prevent a client from accessing other resources
    - Decoupling: avoid tying a client to a particular instance of a resource
    - Abstraction: make a resource easier to use

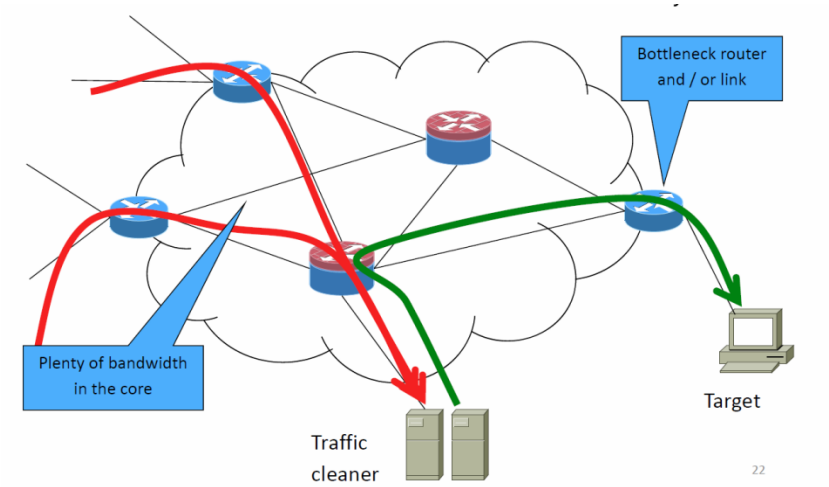
12.1 Examples

- Virtual machines
- Protocol layers
    - Virtual resource: protocol service
    - Physical resource: protocol below
    - Mechanism: headers, etc
- Virtual web servers
    - Virtual resource: web site
    - Physical resource: web server
    - Method: multiplexing
    - Mechanism: DNS alias and HTTP headers
- Virtual circuits
    - Physical resource: network link
    - Virtualization method: multiplexing
    - Mechanism: VC identifiers, VC switching
- Threads
    - Virtual resource: threads
    - Physical resource: CPU
    - Method: Multiplexing
    - Mechanism: pre-emption, time slicing, context switching, scheduling
- VLANs
    - Methods: Multiplexing
    - Mechanisms: port assignment, tags
- RAM disks
    - Physical resource: RAM
    - Method: emulation
    - Mechanism: interposition
- Tunnels (virtualizing networks)

- E.g. Ethernet over HDLC



- Why tunneling?
  - Personal routing / NAT avoidance
  - Mitigating DDoS attacks



  - Traffic aggregation and management
  - Security, anonymity
  - Mobility
    - When coputer moves, address has to change
    - Solution: computer has two addresses
      - Locally acquired one (e.g. WiFi in coffee shop)
      - Semi-permanent (acquired from “home” or provider)
    - IP traffic on semi-permanent address tunneled to provider over local interface (using PPTP)
    - MobileIP doesn’t use tunneling (much); but this method is becoming more popular

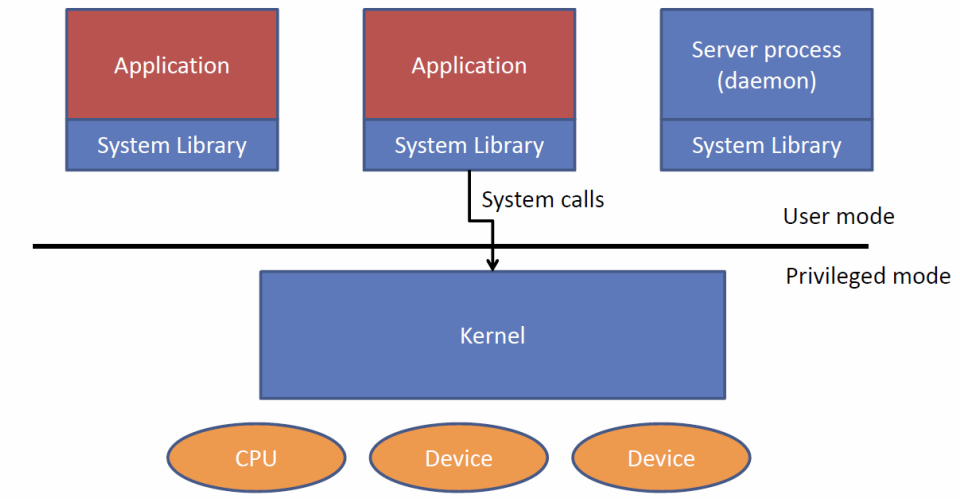
- Virtual memory
  - Virtual resource: virtual memory

    - Physical resource: RAM
    - Method: Multiplexing
    - Mechanism: virtual address translation
    - Benefit: easier memory to manage
- Paged virtual memory
    - Virtual resource: *paged* virtual memory
    - Physical resource: RAM *and disk*
    - Method: Multiplexing *and emulation*
    - Mechanism: *virtual memory + paging from/to disk*
    - Benefit: *more memory than you really have*
- Files
    - Physical resource: disk
    - Method: multiplexing, emulation
    - Mechanism: block allocation, metadata
- Windows
    - Physical resource: frame buffer and/or GPU
    - Method multiplexing and emulation
    - Mechanism: windows as separate bitmaps/textures

12.2 The operation system as resource manager

- Allocate resource to applications
    - Sharing
        - Multiplex hardware among applications (CPU, memory, devices)
        - Applications shouldn't need to be aware of each other
    - Protection
        - Ensure one application can't read/write another's data (in memory, on disk, over network)
        - Ensure one application can't see another's resources (CPU, storage space, bandwidth)
- Goals
    - Fairness: No starvation, every application makes progress
    - Efficiency: Best use of complete machine resources (e.g. minimize power consumption)
    - Predictability: guarantee real-time performance
    - All in mutual contraction

12.3 General operation system structure

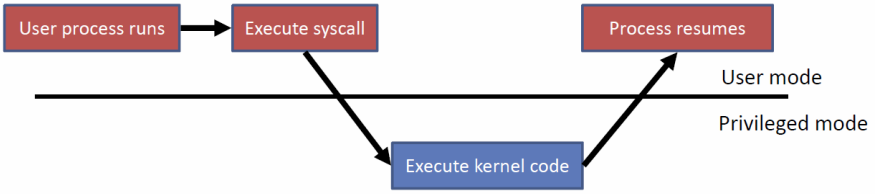


- Kernel
  - The part of the OS which runs in privileged mode
    - Most of Unix/Windows (except libraries)
  - Also known as nucleus, nub or supervisor
  - Kernel is just a (special) computer program
    - Typically an event-driven server
  - Responds to multiple entry points
    - System calls
    - Hardware interrupts
    - Program traps
- System libraries
  - Convenience functions
    - Common functionality like “strcmp”
  - System call wrappers
    - Create and execute system calls from high-level languages
- Services provided by the OS
  - Program execution: Load program, execute on one or more processors
  - Access to I/O devices (disk, network, keyboard, screen)
  - Protection and access control (for files, connections, etc)
  - Error detection and reporting (trap handling, etc)
  - Accounting and auditing (statistics, billing, forensics, etc)

13 Processes and threads

13.1 System calls

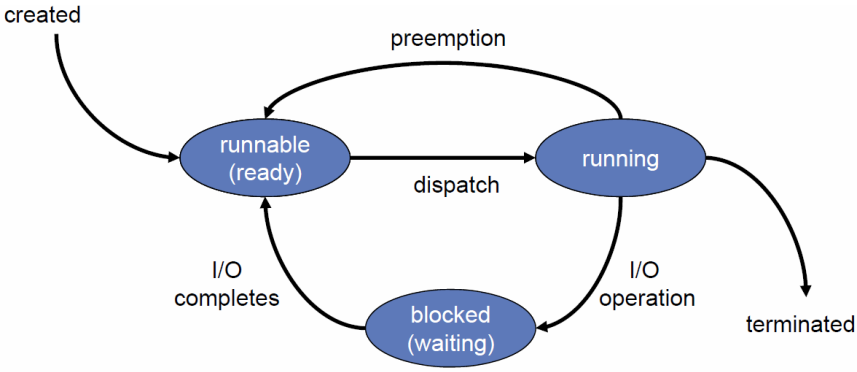
- RPC to the kernel
- Kernel is a series of system call event handlers
- Mechanism is hardware dependent



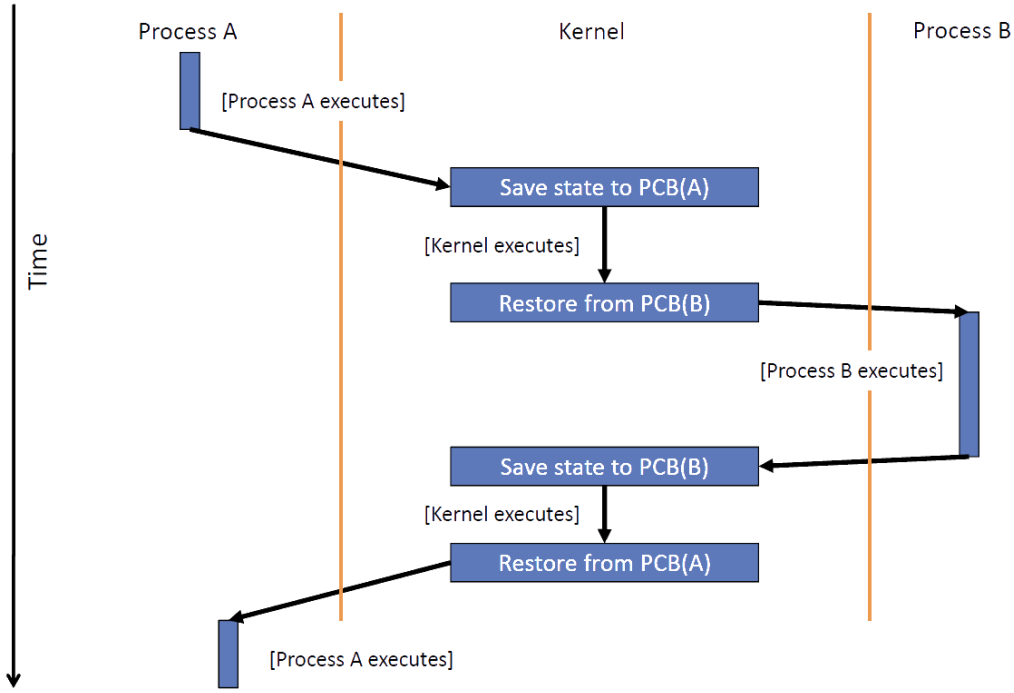
- System calls are *the* way a program request a service from the OS kernel
- System call arguments: implementation varies
    - Passed in processor registers
    - Stored in memory (address in registers)
    - Pushed on the stack
    - System library (libc) wraps as a C function
    - Kernel code wraps handler as C call
- Traps
    - System calls are a special case of traps
        - Transfer control to the kernel, which saves resume information for user process
    - Others include
        - Hardware interrupts (including timer)
        - Error (e.g. divide by zero, protection violation)
        - Page faults

13.2 Processes

- Ingredients
    - Virtual processor
        - Address space
        - Registers
        - Instruction pointer / program counter
    - Program text (object code)
    - Program data (static, heap, stack)
    - OS “stuff”
        - Open files, sockets, CPU share
        - Security rights
- Process lifecycle



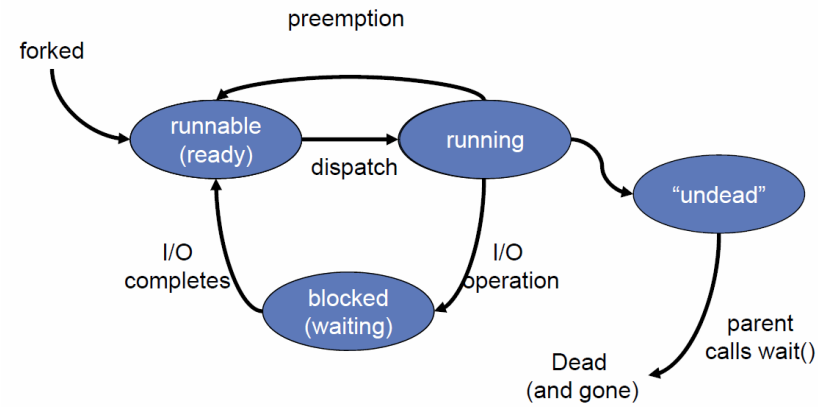
- Multiplexing
  - OS time-division multiplexes processes
  - Each process has a process control block
    - In-kernel data structure
    - Holds all virtual processor state
- Process switching



13.2.1 Process creation

- Bootstrapping problem, need:
  - Code to run
  - Memory to run it in
  - Basic I/O set up (so you can talk to it)
  - Way to refer to the process
- Typically, “spawn” system call takes enough arguments to constructs, from scratch, a new process
- Unix `fork` and `exec`

- Dramatically simplifies creating processes
  - `fork` creates “child” copy of calling process
  - `exec` replaces text of calling process with new program
  - There is no “spawn”
- Unix is entirely constructed as a family tree of such processes
- Process state diagram on Unix

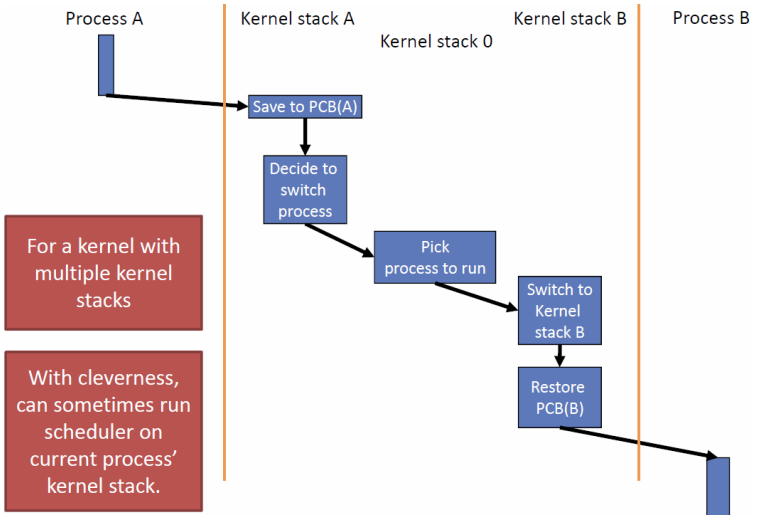


13.3 Kernel threads

- Types of threads
  - Kernel threads
  - One-to-one user-space threads
  - Many-to-one, many-to-many

13.3.1 Kernel threads

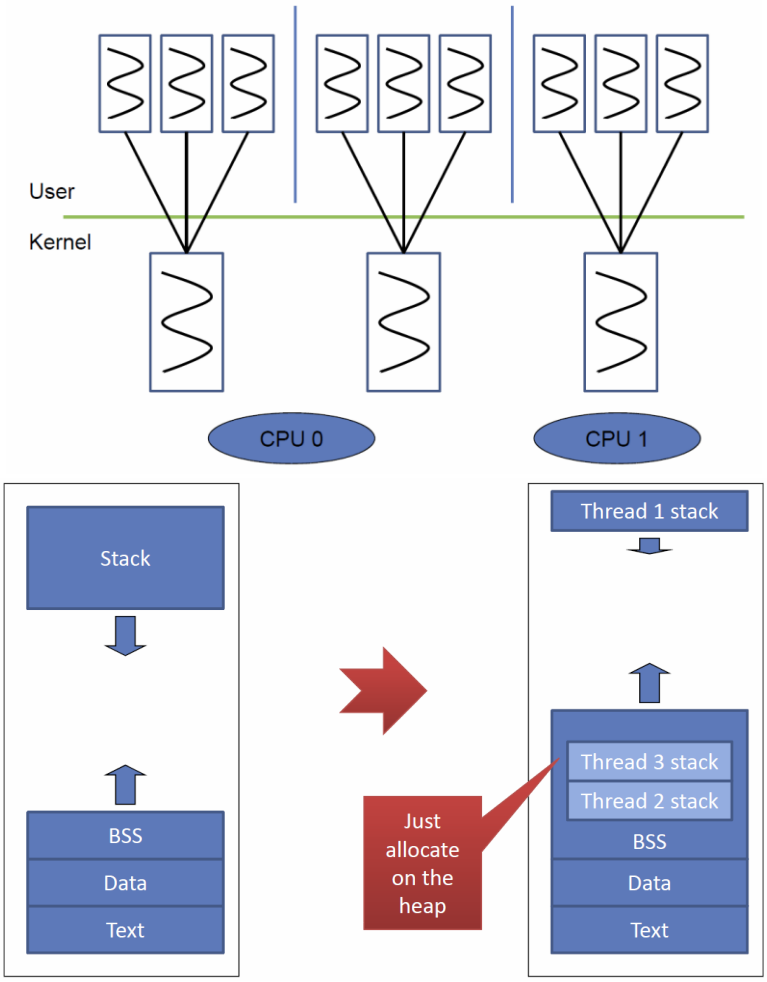
- Kernels can (and some do) implement threads
- Multiple execution context inside the kernel
  - Much as in a JVM
- Says nothing about user space
  - Context switch still required to/from user process
- First, how many stacks are there in the kernel?
  - Unix has a kernel stack per process. The thread scheduler runs on the first thread (every context switch is actually two)
  - Others only have one kernel stack per CPU



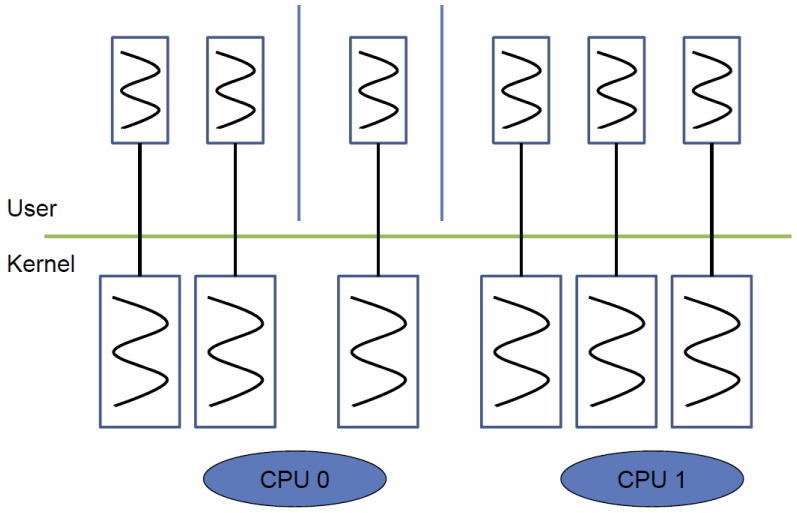
- System calls in more detail
    - In user space
        1. Marshall the arguments somewhere safe
        2. Saves registers
        3. Loads system call number
        4. Executes syscall instruction
    - In kernel space
        - Kernel entered at fixed address (privileged mode set)
        - Need to call the right function and return:
            1. Save user stack pointer and return address (in the process control block)
            2. Load stack pointer for this process' *kernel* stack
            3. Create a C stack frame on the kernel stack
            4. Look up the system call number in a jump table
            5. Call the function
    - Returning in the kernel
        1. Load the user stack pointer
        2. Adjust the return address to point to
            - Return path in user space back from the call, or
            - Loop to retry system call if necessary
        3. Execute "syscall return" instruction

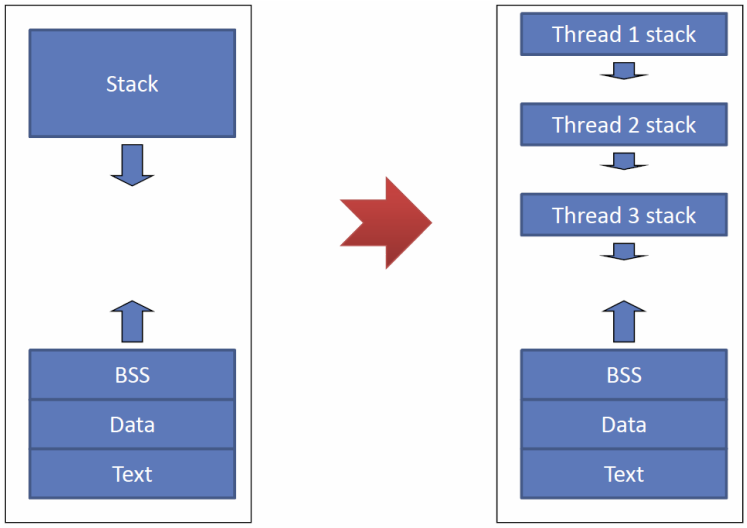
13.3.2 User-space threads

- Possible options
    - Implement threads within a process
    - Multiple kernel thread in a process
    - Some combination of the above
- Many-to-one threads
    - Early thread libraries
    - No kernel support required

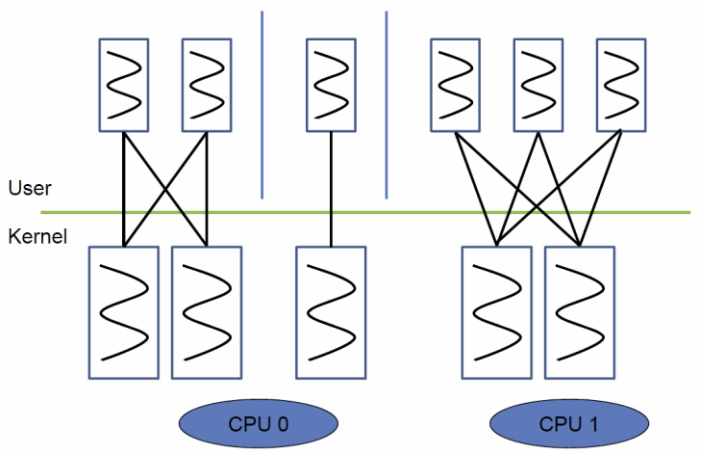


- One-to-one user threads
  - Every user thread is/has a kernel thread
  - Equivalent to multiple processes sharing an address space, except that “process” now refers to a group of threads
  - Most modern OS thread packages (Linux, Windows, MacOSX)





- Comparison
  - User-level threads
    - Cheap to create and destroy
    - Fast context switch
    - Can block entire process (not just on system calls)
  - One-to-one threads
    - Memory usage (kernel stack)
    - Slow to switch
    - Easier to schedule
    - Nicely handles blocking
- Many-to-many threads
  - Multiplex user-level threads over several kernel-level threads
  - Only way to go for a multiprocessor
  - Can "pin" user thread to kernel thread for performance/predictability
  - Thread migration costs are "interesting"



14 Scheduling

14.1 Introduction

- Objectives
    - Fairness
    - Enforcement of policy
    - Balance
    - Power consumption
    - Others depend on workload
        - Batch jobs
        - Interactive
        - Real-time and multimedia

14.1.1 Batch workloads

- "Run this job to completion, and tell me when you're done"
    - Typical mainframe use case
    - Making a comeback with large clusters
- Goals
    - Throughput (jobs per hour)
    - Wait time (time to execution)
    - Turnaround time (submission to termination)
    - CPU utilization (don't waste resources)

14.1.2 Interactive workloads

- "Wait for external events, and react before the user gets annoyed"
    - Word processing, browsing, etc. Common for PCs, phones
- Goals
    - Response time: How quickly does something happen?
    - Proportionality: something should be quicker

14.1.3 Soft real-time workloads

- "This task must complete in less than 50ms", or "This program must get 10ms CPU time ever 50ms"
    - Data acquisition, I/O processing
    - Multimedia application (audio and video)
- Goals
    - Deadlines, guarantees
    - Predictability (real time ≠ fast)

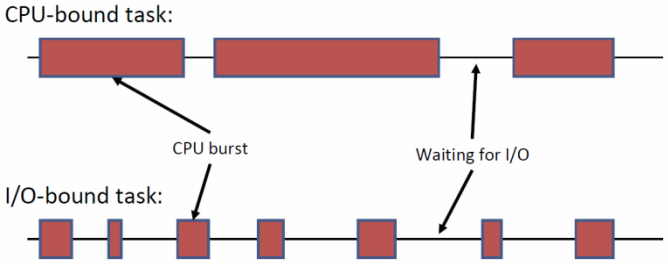
14.1.4 Hard real-time workloads

- "Ensure the plane's control surfaces move correctly in response to pilots actions" or "Fire the spark plugs in the car's engine at the right time"
    - Mission-critical, extremely time-sensitive control applications

- Not covered here, very different techniques required

14.2 Assumptions and definitions

- CPU and I/O-bound tasks



- Simplifying assumptions
  - Only one processor
  - Processor runs at a fixed speed
- When to schedule?
  1. A running process blocks (e.g. initiates blocking I/O, or waits on a child)
  2. A blocked process unblocks (e.g. I/O completes)
  3. A running or waiting process terminates
  4. An interrupt occurs (e.g. I/O or timer)
  - 2 or 4 can involve preemption
- Preemption
  - Non-preemptive scheduling
    - Require each process to explicitly give up the scheduler (e.g. by starting I/O, calling "yield", etc. Windows 3.1 or older MacOS, some embedded systems)
  - Preemptive scheduling
    - Process dispatched and descheduled without warning (often on a timer interrupt, page fault, etc)
    - The most common case in most OSes
    - Soft real-time systems are usually preemptive, hard real-time systems often not
- Overhead
  - Dispatch latency: time taken to dispatch a runnable process
  - Scheduling cost: $2 \cdot (\text{half context switch}) + (\text{scheduling time})$
  - Time slice allocated to a process should be significantly more than scheduling overhead
  - Tradeoff: response time vs. scheduling overhead

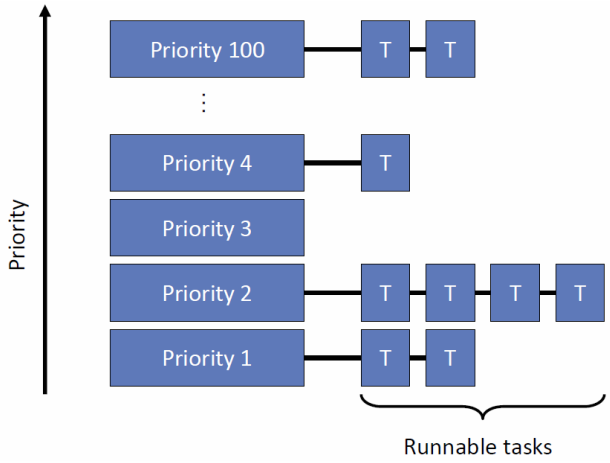
14.3 Batch-oriented scheduling

- First-come first-served
  - Simplest algorithm
  - Unpredictable
  - Convoy phenomenon: short processes back up behind long-running processes
- Shortest job first (SJF)
  - Always run process with shortest execution time

- Optimal: minimizes waiting time (and hence turnaround time)
- Execution time estimation
  - Mainframes: ask users
  - Non-batch workloads: use CPU burst time and keep exponential average of prior bursts
- Problem: jobs arrive all the time
  - "Shortest remaining time next": New, short jobs may preempt longer jobs already running
  - Still not an ideal match for dynamic, unpredictable workloads

14.4 Scheduling interactive loads

- Round-robin
  - Simplest interactive algorithm
  - Run all runnable tasks for a fixed quantum in turn
  - Advantages
    - Easy to implement, analyze and understand
    - Higher turnaround time than SJF, but better response
  - Disadvantages
    - It's rarely what you want, treats all tasks the same
- Priority
  - Very general class of algorithms
  - Assign every task a priority
  - Dispatch highest priority runnable task
  - Priorities can be dynamically changed
  - Schedule processes with same priority using round-robin, FCFC, etc
  - Priority queue



  - Multi-level queues: can schedule different priority levels differently (e.g. round-robin for interactive, high-priority, but FCFS for batch, background, low priority jobs)
    - Ideally generalizes to hierarchical scheduling
- Starvation

    - Strict priority schemes do not guarantee progress for all tasks
    - Solution: Ageing
        - Tasks which have waited a long time are gradually increased in priority
        - Eventually, any starving task ends up with the highest priority
        - Reset priority when quantum is used up
- Multilevel feedback queues
    - Idea: penalize CPU-bound tasks to benefit I/O bound tasks
        - Reduce priority for processes which consume entire quantum
        - Eventually, re-promote process
        - I/O bound task tend to block before using their quantum => remain higher priority
    - Very general: any scheduling algorithm can reduce to this (the problem is implementation)

14.4.1 Linux O(1) scheduler

- 140 level multilevel feedback queue
    - 1-100 (high priority)
        - Static, fixed, “real-time”
        - FCFS or RR
    - 100-140: user tasks, dynamic
        - RR within a priority level
        - Priority ageing for interactive (I/O intensive) tasks
- Complexity of scheduling is independent of the number of tasks
    - Two arrays of queues: “runnable” and “waiting”
    - When no more tasks in “runnable” array, swap array

14.4.2 Linux “completely fair scheduler”

- Task’s priority = how little progress it has made
    - Adjusted by fudge factors over time
- Implementation uses red-black tree
    - Sorted list of tasks
    - Operations now O(n), but this is fast
- Essentially, this is the old idea of “fair queuing” from packet networks

14.5 Real-time scheduling

- Problem: giving real-time-based guarantees to tasks
    - Tasks can appear at any time
    - Tasks can have deadlines
    - Execution time is generally known
    - Tasks can be periodic or aperiodic
- Must be possible to reject tasks which are not schedulable, or which would result in no feasible schedule

14.5.1 Rate-monotonic scheduling RMS

- Schedule periodic tasks by always running task with shortest period first
  - Static (offline) scheduling algorithm
- Suppose
  - m tasks
  - C_i is the execution time of ith task
  - P_i is the period of ith task
- Then RMS will find a feasible schedule if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m \cdot \left(2^{\frac{1}{m}} - 1\right)$$

14.5.2 Earliest deadline first EDF

- Schedule task with earliest deadline first
  - Dynamic, online
  - Tasks don't actually have to be periodic
  - More complex for scheduling decisions: O(n)
- EDF will find a feasible schedule if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- Assumes zero context switch time

14.6 Scheduling on multiprocessors

- Multiprocessor scheduling is two-dimensional
  - When to schedule a task?
  - Where (which core) to schedule it on?
- General problem is NP complete
  - Locks make it even worse
  - Caches/NUMA also a problem
- Wide open research topic

15 Inter-process communication

15.1 Hardware support for synchronization

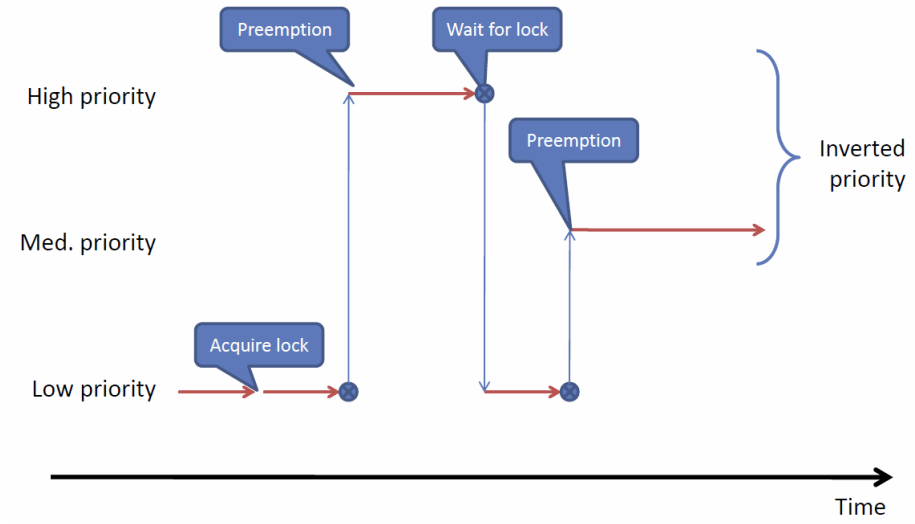
- Disabling interrupts
  - Disable interrupts, critical section, enable interrupts
  - Nice and simple
  - Can't be rescheduled inside critical section (data can't be altered by anything else)
  - .. except another processor
  - Very efficient if kernel is on a uni-processor
- Test-and-set instruction (TAS)
  - Atomically read the value of a memory location and set the location to 1
  - Available on some hardware
- Compare-and-swap (CAS)
  - Atomically compare flag to some "old" value and if equal, set flag to some "new" value
  - Theoretically, slightly more powerful than TAS
- Load-locked, store-conditional
  - Factors CAS into two instructions
    - LL: load from a location and mark "owned"
    - SC: atomically
      - Store only if already marked by this processor
      - Clear any marks set by other processors
      - Return whether it worked

15.1.1 Spinning

- On a uni-processor: not much point in spinning at all (what's going to happen?)
- On a multiprocessor
  - Can't spin forever
  - Another spin is always cheap
  - Blocking thread and rescheduling is expensive
  - Spinning only works if lock holder is running on another core
- Competitive spinning
  - Question: How long to spin for?
  - Within a factor of 2 of optimal, offline (i.e. impossible) algorithm
  - Good approach: spin for the context switch time
    - Best case: avoid context switch entirely
    - Worst case: twice as bad as simply rescheduling

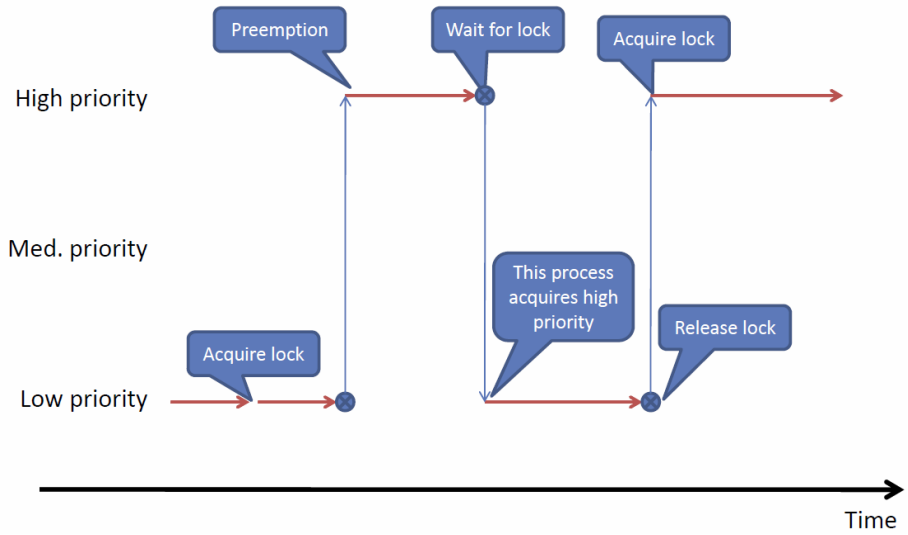
15.2 IPC with shared memory and interaction with scheduling

15.2.1 Priority inversion



15.2.2 Priority inheritance

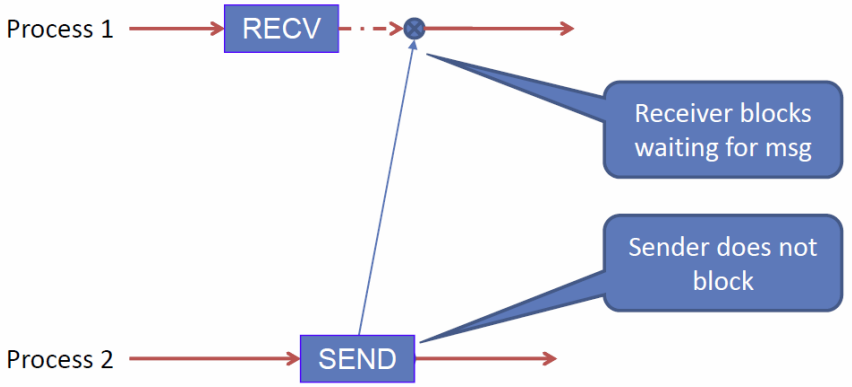
- Process holding the lock inherits the priority of highest priority process that is waiting for the lock.
    - Releasing the lock returns the priority to the previous value
    - Ensures forward progress



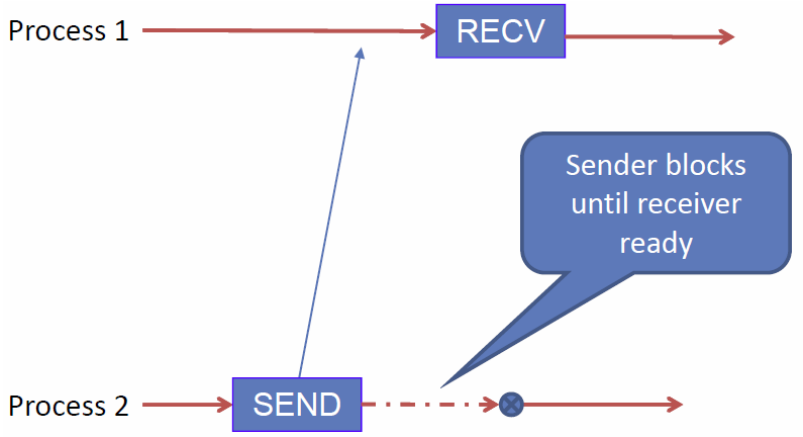
- Alternative: Priority ceiling
    - Process holding the lock acquires the priority of highest-priority process than *can ever* hold the lock
    - Requires static analysis, used in embedded real-time systems

15.3 IPC without shared memory

- Asynchronous buffered IPC



- Synchronous (unbuffered) IPC



15.3.1 Unix pipes

- Basic (first) Unix IPC mechanism
- Unidirectional, buffered communication channel between two processes
- `int pipe(int pipefd[2])`
- Pipes are created in one process, and then fork is used
- Naming pipes
  - Pipes can be put in the global namespace by making them a “named pipe”
  - Special file of type “pipe” (also known as a FIFO) => `mkfifo`

15.3.2 Local remote procedure calls

- RPC can be used locally, but naïve implementations are slow
  - Lots of things (like copying) don’t matter with a network, but do matter between local processes
  - Can be made very fast

15.3.3 Unix signals

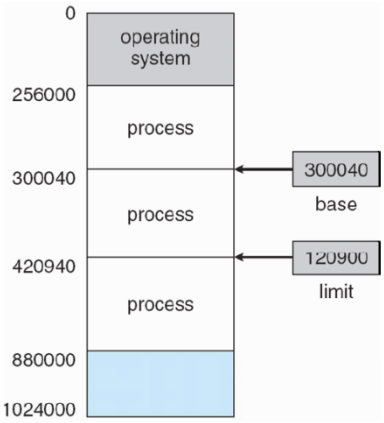
- Asynchronously received notification from the kernel
  - Receiver doesn’t wait: signal just happens
  - Interrupt process, and
    - Kill it

        - Stop (freeze) it
        - Jump to a handler function
- Unix signal handlers
    - Function executed on current user stack
        - Leads to “interesting” issues, since user process is in undefined state when signal is delivered
        - If user process is in the kernel, may need to retry current system call
        - Handler can be set to run on a different (alternative) stack
    - Some signal types pre-defined
        - Memory protection fault, CTRL-C, timer, etc
    - Otherwise, delivered by another user process, using “kill”

16 Memory management

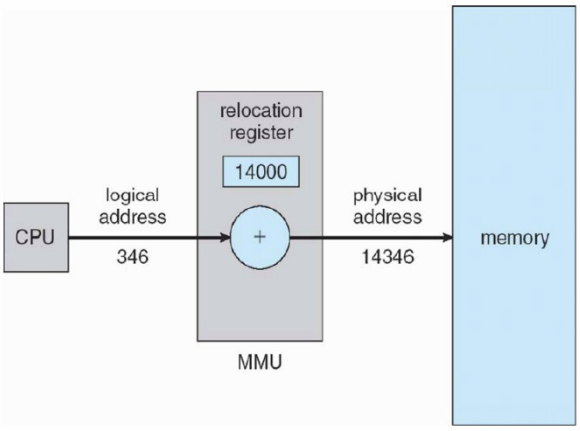
- Goals of memory management
  - Allocate physical memory to applications
  - Protect an application's memory from others
  - Allow applications to share areas of memory (data, code, etc)
  - Create illusion of a whole address space
    - Virtualization of the physical address space
  - Create illusion of more memory than you have
    - Demand paging
- Functions
  - Allocating physical addresses to applications
  - Managing the name translation of virtual addresses to physical addresses
  - Performing access control on memory access

16.1 Memory management schemes



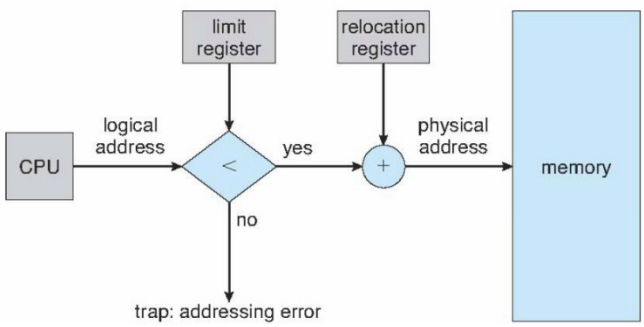
16.1.1 Partitioned memory

- A pair of *base* and *limit* register define the logical address space
- Issue: base address isn't known until load time
- Options
  - Compiled code must be completely position-independent
  - Relocation register maps compiled addresses dynamically to physical addresses



- Contiguous allocation
  - Main memory is usually split into two partitions
    - Resident operation system, normally held in low memory with interrupt vector
    - User process then held in high memory
  - Relocation registers used to protect user processes from each other, and from changing operating system code and data
    - Base register contains value of smallest physical address

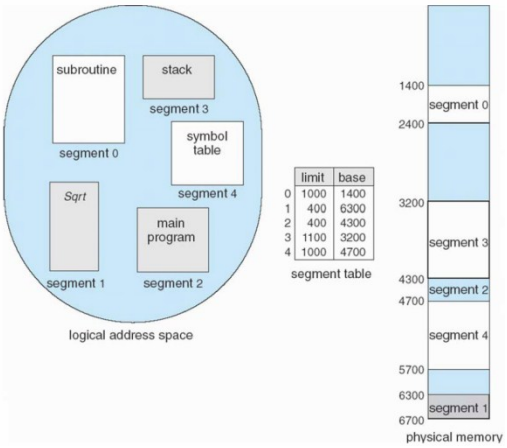
        - Limit register contains range of logical addresses, each logical address must be less than the limit register
        - MMU maps logical address dynamically
    - Hardware support for relocation and limit register

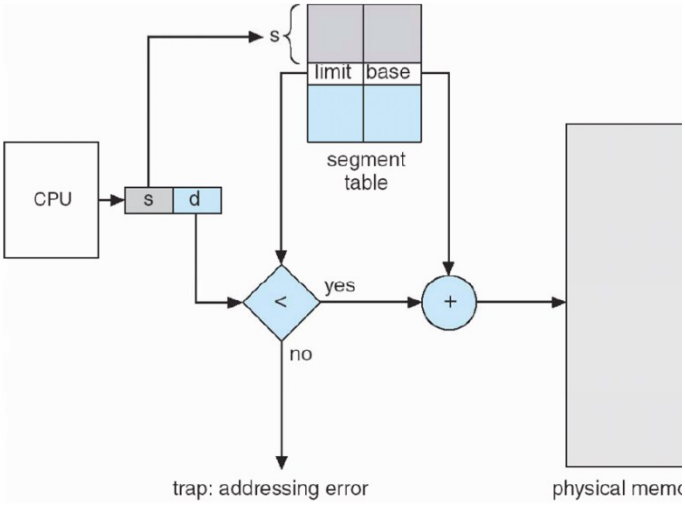


- Summary
    - Simple to implement
    - Physical memory fragmentation
    - Only a single contiguous address range
        - How to share data or code between applications?
        - How to load code dynamically
        - Total logical address space ≤ physical memory

16.1.2 Segmentation

- Generalize base + limit
    - Physical memory is divided into segments
    - Logical address = (segment id, offset)
- Segment identifier supplied by
    - Explicit instruction reference
    - Explicit processor segment register
    - Implicit process state
- Segmentation hardware

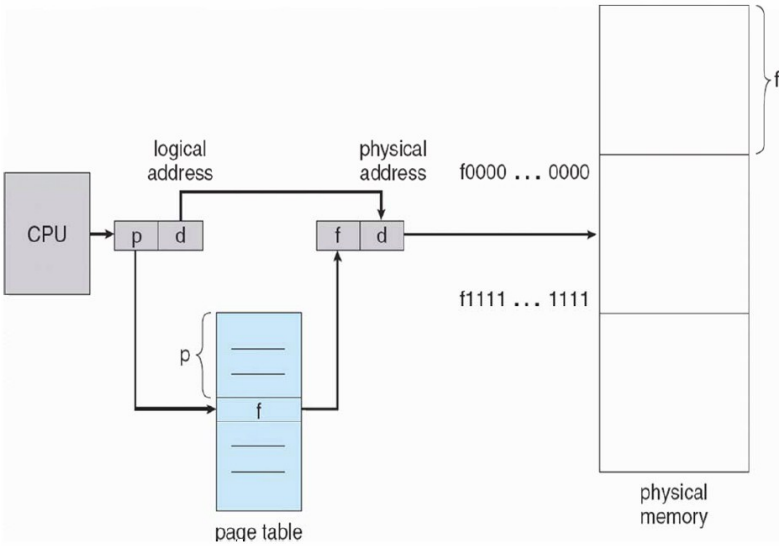




- Segmentation architecture
    - Segment table – each entry:
        - Base: starting physical address of the segment
        - Limit: length of the segment
    - Segment-table base register (STBR)
        - Current segment table location in memory
    - Segment-table length register (STLR)
        - Current size of the segment table
    - Segment number s is legal if s < STLR
- Summary
    - Fast context switch: simply reload STBR/STLR
    - Fast translation
        - 2 loads, 2 compares
        - Segment table can be cached
    - Segments can easily be shared: segments can appear in multiple segment tables
    - Physical layout must still be contiguous, fragmentation still a problem

16.1.3 Paging

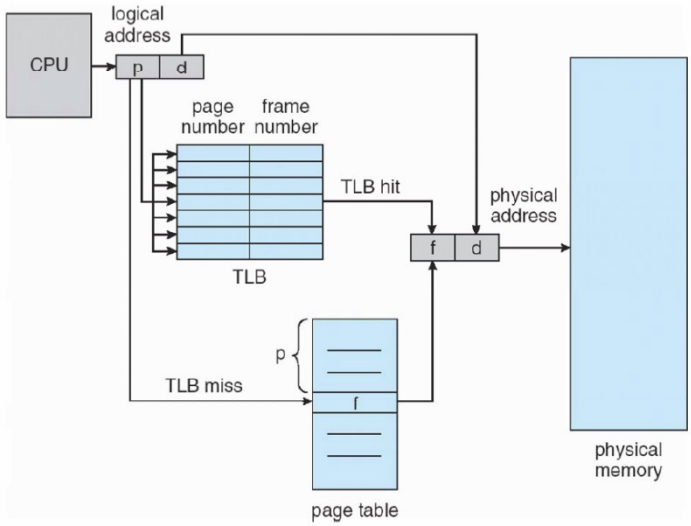
- Solves contiguous physical memory problem: process can always fit if there is available free memory
- Divide physical memory into **frames** (size is power of two, e.g. 4096 bytes)
- Divide logical memory into **pages** of the same size
- For a program of n pages in size
    - Find and allocate n frames
    - Load program
    - Set up **page table** to translate logical pages to physical frames
- Paging hardware



- Address translation scheme

- Page table base register and length register (specify current size and location of page table)
- Address generated by CPU is divided into
  - Page number – index into the page table
    - Table entry give the base address of frame in physical memory
  - Page offset – offset into frame

- Page table definitions/names
  - Page table maps VPNs to PFNs
  - Page table entry PTE
  - VPN = virtual page number (upper bits of virtual or logical address)
  - PFN = page frame number (upper bits of physical address)
- Problem: performance
  - Every logical memory access needs two physical memory accesses
    - Load page table entry to get the PFN
    - Load the desired location
  - Half as fast as with no translation
    - Solution: cache page table entries
- Paging hardware with TLB



- Effective access time
  - Associative lookup = ε time unit
  - Assume memory cycle time is 1 microsecond
  - Hit ratio – percentage of time that a page number is found in the associative registers; ratio related to number of associative registers
  - Hit ratio = α
  - Effective access time EAT
    - EAT = (1+ε) α + (2+ε)(1-α)
    - = 2 + ε – α

16.1.3.1 Page protection

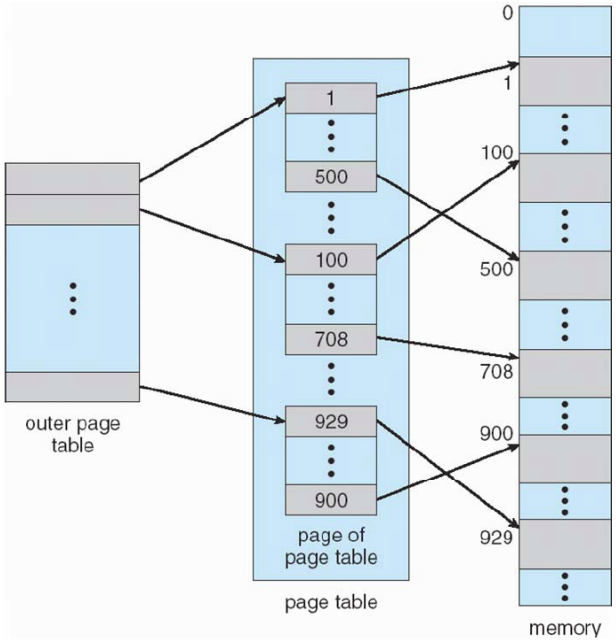
- Associate protection info with each frame
  - Actually associated with the PTE
- Valid-invalid bit
  - "valid" – page mapping is "legal"
  - "invalid" page is not part of the address space (i.e. entry does not exist)
  - Requesting an "invalid" address results in a "fault" or "page fault", …
- Protection information typically includes
  - Readable/writable
  - Executable (can fetch to instruction-cache)
  - Reference bits used for demand paging

16.1.3.2 Page sharing

- Shared code
  - One copy of read-only code shared among processes
  - Shared code appear in same location in the logical address space of all processes
  - Data segment is not shared, different for each process
  - But still mapped at the same address (so the code can find it)
- Private code and data
  - Allows code to be relocated anywhere in address space
- Protection
  - Protection bits are stored in page table => independent of frames themselves
  - Each page can have different protection to different processes

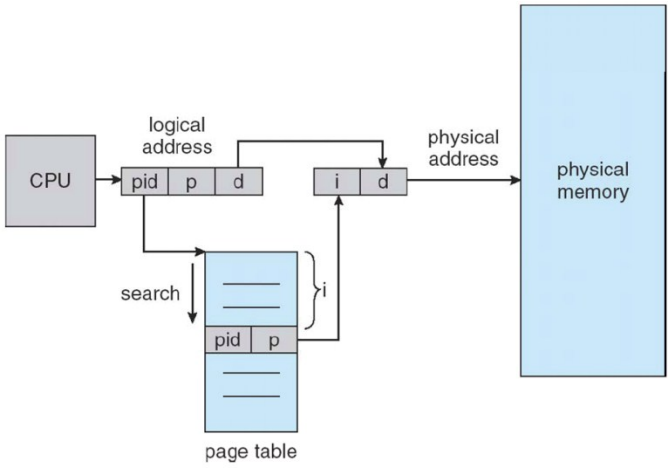
16.1.3.3 Page table structure

- Page tables can be very large (if one contiguous block)
- Observation: most PTEs are not actually used
- Solution 1: Put the page table into logical memory
  - Each table lockup must be translated a second time to obtain the PTE for an address
  - Unused ranges in the page table are represented as invalid pages
  - TLB hides most of the double lookups
- Solution 2: Hierarchical page tables
  - Break up the logical address space into multiple page tables; simplest technique is a two-level page table
- Solution 3: Hashed page tables
  - VPN is hashed into table



- Hash bucket has a chain of logical to physical page mappings
- Hash chain is traversed to find a match
- Can be fast, but unpredictable

- Solution 4: Inverted page table
  - One system-wide table now maps PFN to VPN
  - One entry for each real page of memory, contains VPN and which process owns the page
  - Bounds total size of all page information on machine
    - Hashing used to locate an entry efficiently



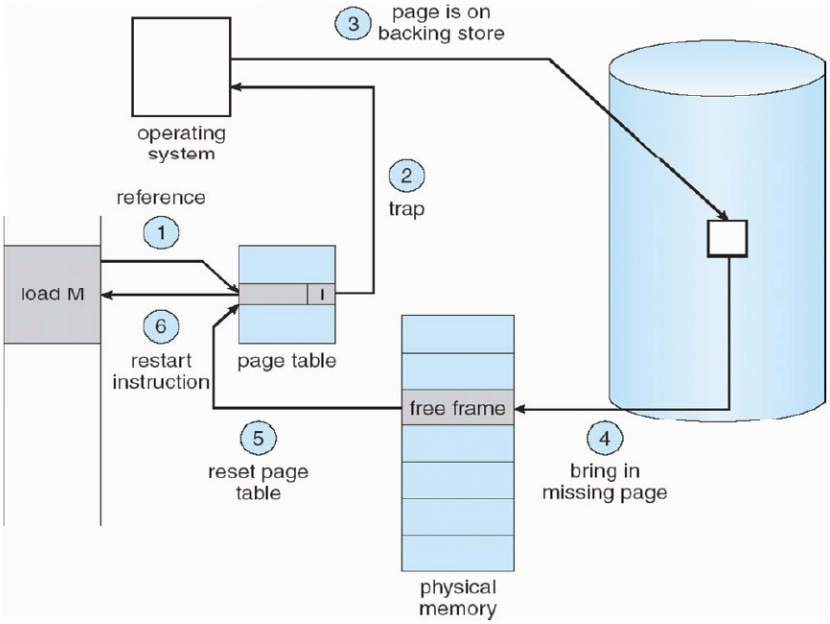
17 Demand paging

17.1 Copy-on-write COW

- Problem
  - `fork` can be extremely expensive, because the entire process' address space has to be copied. This is especially true when exec is called afterwards => a lot of memory gets copied, but will be directly overwritten.
  - `vfork` shares address space, doesn't copy. This is fast, but dangerous
  - It would be better to copy only when you know something is going to get written, and share the page otherwise
- Copy-on-write (COW) allows both the parent and child process to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
  - COW allows more efficient process creation as only modified pages are copied
  - Free pages are allocated from a pool of zeroed-out pages
- How does it work?
  - Initially mark all pages as read-only
  - If either process writes, a page fault occurs
  - Fault handler allocates a new frame and makes a copy
  - Both frames are now marked as writable
- Only modified pages are copied
  - Less memory usage, more sharing
  - Cost is page fault for each mutated page

17.2 Demand-paging

- Idea: Virtual memory can be larger than the physical memory
  - Frames can be "paged out" to disk
- Bring a page into memory only when it is needed, resulting in less I/O, less memory, faster response and support for more users
  - Turns RAM into a cache for the disk
- *Lazy swapper* – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager
- *Strict demand paging*: only page in when referenced
- Page fault
  - Operating system looks at another table to decide where this is an invalid reference, or the page is just not in memory
  - Get empty frame
  - Swap page into frame
  - Reset tables
  - Set validation bit to "valid"
  - Restart the instruction that caused a page fault



17.2.1 Page replacement

- When a page is swapped in, but there is not enough space, another page needs to be replaced. Page replacement is to try to pick a victim page which won't be referenced in the future

17.2.1.1 First-in-first-out FIFO

- Replace the page that has been first swapped in
- Belady's anomaly: Increasing the total number of frames can increase the number of page faults

17.2.1.2 Optimal algorithm

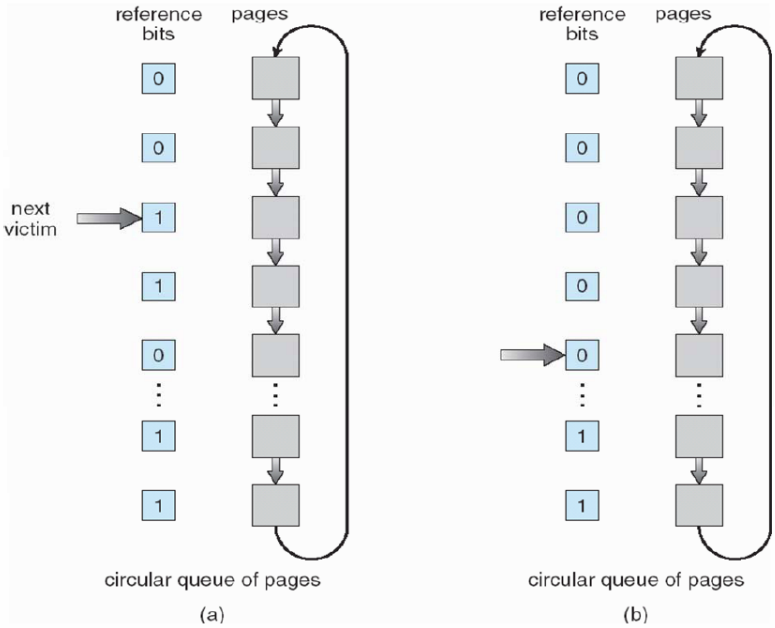
- Replace the page that will not be used for the longest period of time
- Offline algorithm, not possible in practice
- Used to compare how well other algorithms perform

17.2.1.3 Least recently used LRU

- Replace the page that has been used least recently
- Implementation (counter)
    - Every page entry has a counter; every time the page is referenced through this entry, copy the clock into the counter
    - When a page needs to be changed, loot at the counters to determine which are to change
- Implementation (stack)
    - Keep a stack of page numbers in a double link form
    - Page referenced: move it to the top (requires 6 pointers to change)
    - No search for replacement
    - Stack algorithms have the property that adding frames always reduces the number of page faults (no Belady's anomaly)

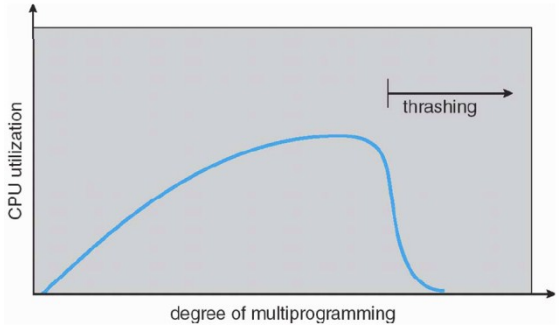
17.2.1.4 LRU approximation

- Reference bit
  - With each page associate a bit that is initially 0
  - When page is referenced, set the bit to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however
- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1, then
    - Set reference bit to 0
    - Leave page in memory
    - Replace next page (in clock order), subject to same rules



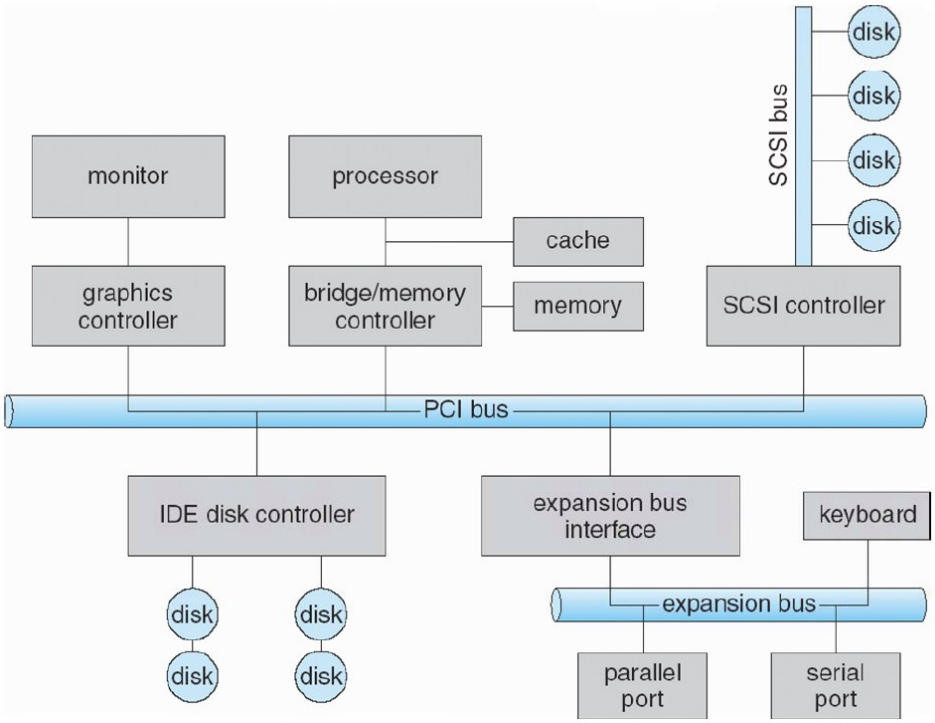
17.2.2 Frame allocation

- Global replacement
  - Process selects a replacement frame from the set of all frames: one process can take a frame from another
- Local replacement
  - Each process selects only its own set of allocated frames
- Thrashing
  - A process is busy swapping pages in and out



18 I/O systems

- What is a device, from an OS programmer's point of view?
  - Piece of hardware visible from software
  - Occupies some location on a bus
  - Set of registers (memory mapped or I/O space)
  - Source of interrupts
  - May initiate *direct memory access DMA* transfer
- I/O hardware, incredible variety of devices
  - Disks, other mass storage devices
  - Networking and communication interfaces
  - Graphics processor
  - Keyboards, mice, other input devices
  - Adaptors
- Bus structure



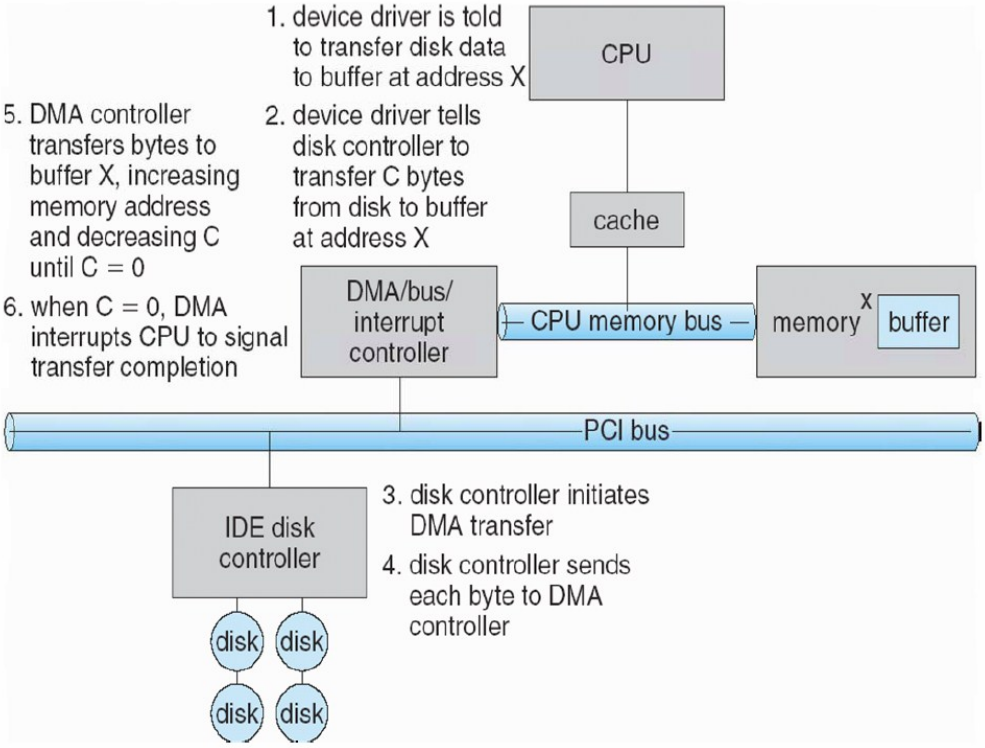
- Registers
  - Memory locations, or similar
  - CPU can load from, to obtain status information, or read input data
  - CPU can store to, to set device state/configuration, write output data or reset states
- Polling
  - Determines state of a device (e.g. command ready, busy, error)
  - Busy-wait cycle to wait for I/O from device
    - Wasteful of CPU
    - Doesn't scale to lots of devices

18.1 Interrupts

- The OS can't poll every device to see if it's ready.
- Solution: device can *interrupt* the processor
    - Acts as a trap: saves state and jumps to kernel address
    - Information about source encoded in the vector
- Interrupts
    - Interrupt handler receives interrupts
    - Some are maskable to ignore or delay

18.2 Direct memory access

- Avoid *programmed I/O* for lots of data
    - E.g. fast network or disk interface
- Requires DMA controller (generally built-in these days)
- Bypasses the CPU to transfer data directly between I/O device and main memory
    - Doesn't take up CPU time
    - Can save memory bandwidth
    - Only one interrupt per transfer

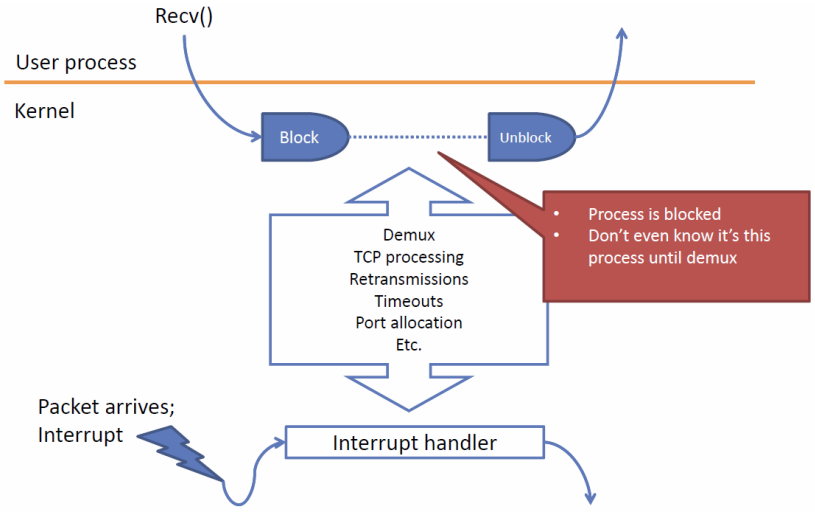


- I/O protection
    - I/O operations can be dangerous
        - Dedicated I/O instructions usually privileged
        - I/O performed via system call
        - DMA transfer must be carefully checked (bypasses memory protection)

18.3 Device drivers

- Software object (module, object, process, hunk of code) which abstracts a device
  - Sits between hardware and the rest of the OS
  - Understand device registers, DMA, interrupts
  - Presents uniform interface to the rest of the OS
- Device abstractions ("driver models") vary
  - Unix starts with "block" and "character" devices
    - Block: deal with large blocks of data at a time, in Unix often look like files
    - Character: single character or short strings via get/put, like keyboard, mice, serial
- Basic problem
  - Hardware is interrupt driven. System must respond to unpredictable I/O event
  - Applications are (often) blocking
  - There may be considerable processing in between (e.g. TCP/IP processing, file system processing)

18.3.1 Example: network receive



- Solution 1: driver threads
  - Interrupt handler
    - Masks interrupt
    - Does minimal processing
    - Unblocks driver thread
  - Thread
    - Performs all necessary packet processing
    - Unblocks user processes
    - Unmasks interrupt
  - User process
    - Per-process handling
    - Copies packet to user space

        - Returns from kernel
- Solution 2: deferred procedure calls (DPC)
    - When a packet arrives, enqueue as DPC (closure)
    - Later in user process run all pending DPCs (executed in the *next* process to be dispatched, before it leaves the kernel)
    - Solution in most versions of unix

18.4 The I/O subsystem

- Caching – fast memory holding copy of data (key to performance)
- Spooling – hold output for a device if it can serve only one request at a time (e.g. printing)
- Scheduling – some I/O request ordering via per-device queue (some OS try fairness)
- Buffering – store data in memory while transferring between devices or memory (to cope with device speed mismatch and device transfer size mismatch)

19 File system abstractions

19.1 Introduction

- What is the filing system?
  - Virtualizes the disk
  - Between disk (blocks) and programmer abstractions (files)
  - Combination of multiplexing and emulation
  - Generally part of the OS
- What the file system builds on
  - Device I/O subsystem
  - Block-oriented storage device
    - Random access
    - Read/write blocks at a particular location
- Goals
  - Virtualized the disk into smaller, abstract units
  - Implement a naming scheme for those units
  - Provide a standard interface for accessing data
  - Ensure the data that is stored remains valid
  - Deliver performance for reading and writing
  - Provide durability in the event of crashes or failures
  - Provide access control between different users
  - Provide controlled sharing between processes

19.2 Filing system interface

- What is a file to the filing system
  - Some data (including its size)
  - One or more names
  - Other metadata and attributes
    - Time of creation, last change
    - Ownership
  - The type of the file
  - Where on disk the data is stored

19.2.1 File naming

- Single-level directory
  - A single directory for all users
  - Advantage: very fast lookup
  - Disadvantage: needs more organization
- Tree-structured directories
- Acyclic graph directories
  - Allow shared subdirectories and files
  - Two different names (aliasing)

    - New directory entry type: link (another name (pointer) to an existing file
- General graph directory
    - How do we guarantee no cycles?
        - Allow only links to files and not directories
        - Garbage collection with cycle collector
        - Check for cycles when new link is added
        - Restrict directory links to parents (all cycles are trivial)

19.2.2 File types

- Some file types are treated specially by the OS
- Simple, common cases
    - Executable files
    - Directories, symbolic links, other file system data
- More types possible
- Unix also uses the file namespace for
    - Naming I/O devices
    - Named pipes
    - Unix domain sockets
    - Process control

19.2.3 Access control

- File owner should be able to control what can be done and by whom
    - Read, write, append, execute, delete, list
- Access control matrix

Rights \ Principals	A	B	C	D	E	F	G	H	J	...
Read	☑	☑	☑			☑	☑			
Write	☑	☑		☑			☑			
Append	☑				☑					
Execute	☑	☑	☑	☑						
Delete	☑									
List	☑				☑					
...										

- Access control list (ACL, row-wise)
    - For each right, list the principals
    - Store with the file
    - Advantage: easy to change rights, scales to a large number of files
    - Disadvantage: does not scale to a large number of principals
- Capabilities (column-wise)
    - Each principal with a right on a file holds a *capability* for that right
        - Stored with principal, not file

        - Cannot be forged
    - Advantage: Very flexible, highly scalable in principals
    - Disadvantage: Revocation is hard
- POSIX access control
    - Simplifies ACLs: each file identifies 3 principals
        - Owner (a single user)
        - Group (a collection of users)
        - The world (everyone)
    - For each principal, the file defines three rights
        - Read (or traverse, if directory)
        - Write (or create a file, if directory)
        - Execute (or list, if a directory)
- Windows has a very powerful ACL support

19.3 Concurrency

- Goals
    - Must ensure that, regardless of concurrent access, file system integrity is ensured
        - Careful design of file system structures
        - Internal locking in the file system
        - Ordering of writes to disk to provide transactions
    - Provide mechanism for users to avoid conflicts themselves
        - Advisory and mandatory locks

20 File system implementation

20.1 On-disk data structures

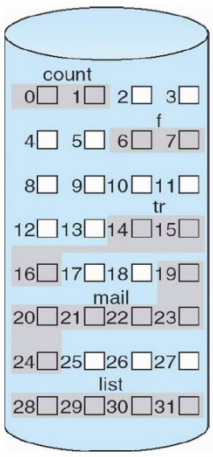
- Design
  - A **structure** or **layout** of data on disk (block array)
  - A **procedure** updating it
- Such that
  - **Fast** for file lookup, read write
  - **Consistent** if machine crashes at any point
  - Tolerates lost blocks on disk with **minimal data loss**
  - **Scales** well with large number of files
  - Makes **efficient** use of disk space

20.2 Representing a file on disk

- Disk addressing
  - Disks have tracks, sectors, spindles, bad sector maps, etc
  - More convenient to use *logical block addresses*
    - Treat disk as compact linear array of usable blocks
    - Block size typically 512 bytes
    - Ignore geometry except for performance
  - Also abstracts other block storage devices
    - Flash drives, storage-area networks, virtual disks
- File layout: how to allocate and layout the blocks used for a file on disk?
  - Contiguous allocation
  - Extent-based allocation
  - Indexed allocation
  - Multi-level indexed allocation

20.2.1 Contiguous allocation

- Keep start/length for each file
- Properties
  - Fast random access (easy address computation)
  - Fast sequential access (blocks are close on the disk)
  - Crash resilience
    - Can write all file allocation information in a single write
  - Space fragmentation
  - May need to copy entire file to grow it (this is bad!)



directory

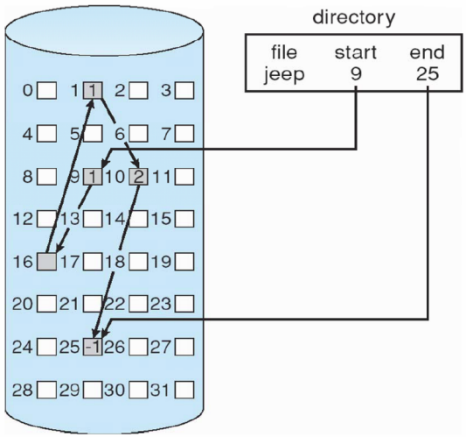
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

20.2.2 Extent-based system

- Allocate block in large chunks, so-called *extents*
  - File consist of one or more extents
  - Most of the advantages of contiguous allocation
  - File only needs to be split when growing
- Fragmentation
  - Only happens as file system ages
  - May require periodic defragmentation
- Crash resilience
  - Now need to ensure integrity of extent list

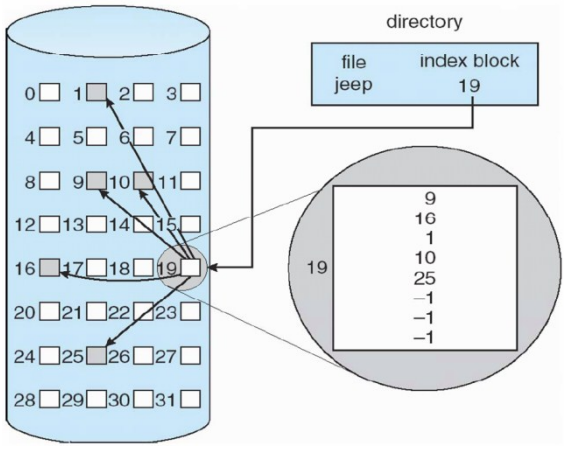
20.2.3 Linked allocation

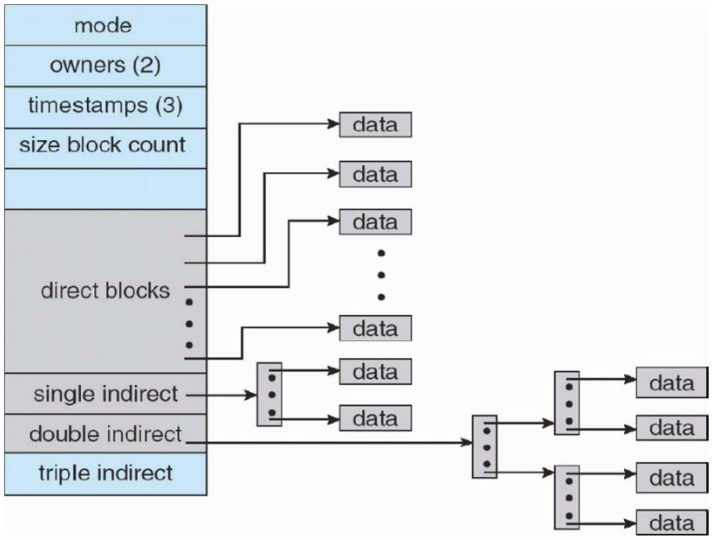
- File is internally-linked list of blocks
  - Blocks can now be anywhere on disk, and blocks contain a "next" pointer
- File allocation table (FAT)
  - List of head pointers
- Simple, but random access is extremely expensive (requires traversal of list => linear in file size)



20.2.4 Indexed allocation

- File is represented using an index block, a so-called *inode*
  - File metadata
  - List of blocks for each part of the file
  - Directory contains pointers to inodes
- Advantages
  - Fast random access
  - Freedom to allocate blocks anywhere
- Issues
  - How big is an inode
  - How can the inode grow?
- Solutions
  - Linked list of inodes
    - Arbitrary file size, access time scales with file size
  - Tree-representation: multi-level index
    - Limited file size, bounded file access time
  - Hybrid approach
    - Progressively deeper indexes as file grows
    - Fast performance case for small files
- Unix file system inode format





20.3 Directory implementation

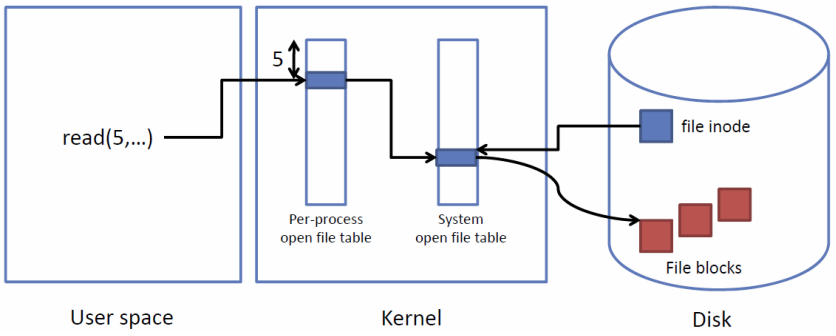
- Linear list of (file name, inode pointer) pairs
  - Simple to program
  - Lookup is slow for lots of files
- Hash table – linear list with closed hashing
  - Fast name lookup
  - Collisions
  - Fixed size
- B-Tree – indexed by name, leaves are inode pointers
  - Complex to maintain, but scales well

20.4 Free space management

- Linked list of free blocks
  - Simple
  - May be slow to allocate lots of blocks
  - Slow recovery if the head pointer is lost
  - No waste of space
  - Cannot get contiguous space easily
- Bitmap
  - Keep one bit for each block, indication whether the block is free or not
  - Easy to get contiguous blocks

20.5 In-memory data structures

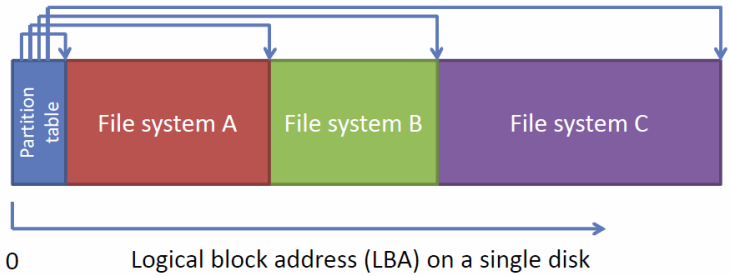
- Reading and writing files
  - Per-process open file table (index into)
  - System open file table (cache of inodes)



20.6 Disks, partitions and logical volumes

20.6.1 Partitions

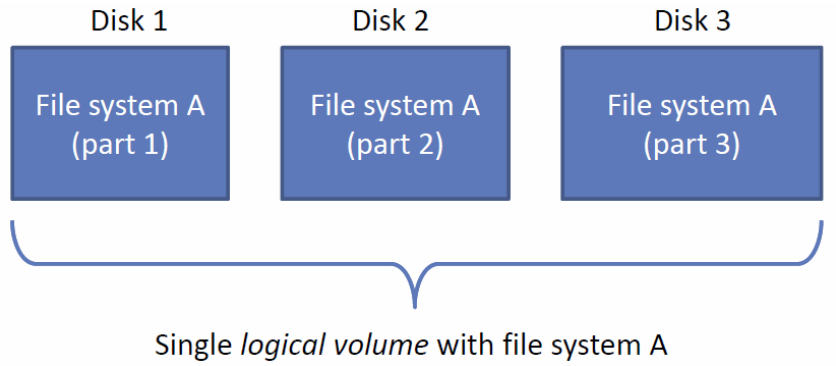
- Multiplexing a single disk among more than one file system



- Contiguous block ranges per file system

20.6.2 Logical volumes

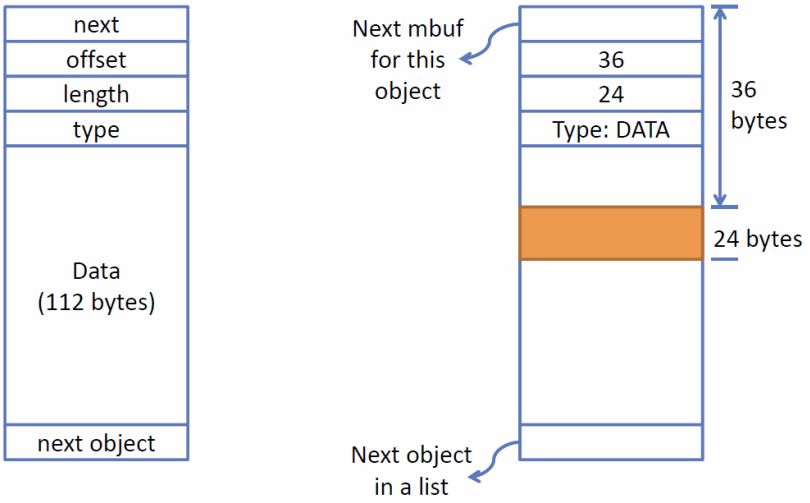
- Emulate 1 virtual disk from more than one physical disks
- Single file system spanning over multiple disks



21 Networking stacks

21.1 Networking stack

- Memory management
  - Problem: how to ship packet data around
  - Need a data structure that can
    - Easily add and remove headers
    - Avoid copying lots of payload
    - Uniformly refer to half-defined packets
    - Fragment large datasets into smaller units
  - Solution: data is held in a linked list of “buffer structures”
- BSD Unix mbufs (Linux equivalent: sk_buffs



- Protocol graphs
  - Graph nodes can be
    - Per protocol (handle all flows)
      - Packets are “tagged” with demux tags
    - Per connection (instantiated dynamically)
      - Multiple interfaces as well as connections