Info

14 October 2020 13:13

Help

Operating System Concepts
Computer Networking: A Top-Down Approach

50% OS & 50% networks

- What is an OS, what are the main functions, how is it designed – how to analyse, write and design programs at an OS level
- In networks, learn communication between computers via networks, learn how networks are built, and how to use networks

Coursework is 20%

- Develop a multi-threaded intrusion detection system in C
- Nov 5th release, Dec 9th hand in

Exam is 80%

- April 2021
- online open book

Moodle
Module Page

Exam Revision
https://www.os-book.com/OS10/review-dir/index.html

Operating Systems

18 October 2020 15:17

Objectives:

- What is an OS?
- Main functions
- How is an OS designed?

An OS is a software that acts as an intermediary between a user and the hardware
The design varies on the purpose – e.g. PC OS are designed to be user friendly
The main job of an OS is to allocate hardware resources to user processes, and control their execution.

Goal: avoid failures and errors, e.g. two programs attempting to write at the same memory location simultaneously

The most basic function of an operating system is to load a program into memory, executing the program, and then stopping the program

While executing, the program may have to do I/O operations – for efficiency and protection, users cannot control devices directly

Programs may require read/write access to directories. OS provides system calls, which can be used to achieve this.

OS also allows communications between various processes. Inter-process communication happens through shared memory and another method that we'll discuss in Processes.

The OS also handles errors. E.g. power failure.

The OS handles accounting, by checking how many processes are running and what resources they're using. This info is gathered by process control blocks.

Kernel

- Kernel is the core of an OS
- It is loaded into the main memory at system startup
- It is the process running at all time on the computers
- There are certain function which only a kernel can perform – e.g. memory management, process scheduling, file handling.

The part of memory where the kernel runs is called the kernel space. The divide between kernel and user space is strict to protect the kernel from user operations.

System Calls – this is how user programs request the kernel to make certain privileged calls. System calls are low level operations.

To distinguish between user and kernel ops we use the mode bit, 0 or 1 depending on whether we are running In user or kernel mode.

A Layered Approach

Modular Design

- Can be achieved through separation of different layers.
- The bottom layer is the hardware and the top layer is the user.
- Layer K uses the service of layer K+1 and provides to layer K+1

The advantage of this approach:

- can modify the structure of one layer without modifying any other layers.
- It is simpler to construct a layered structure than a monolithic one
- Easier to debug
- Clear interfaces between layers

The disadvantages of this approach:

- Defining layers is difficult
- System calls access multiple layers to execute, which adds overhead. So system calls can be slower on layered structures
- Layers below can never require the services of a layer above

Microkernels

The Mac operating system modularised the kernel. Removed all non-essential components from the kernel and implemented them as either system or user-level programs.

- This results in a smaller kernel
- Extending the OS is easy, services are running in the user space so more secure and reliable.
- Performances hampered due to increased system call overhead.

Most modern OS's use Loadable kernel modules – similar to the layered approach but differs because any module can call any other module – similar to microkernels because the kernel only provides core functionality, while other services are implemented dynamically as the kernel is running.

Introduction to C

18 October 2020 15:12

Objectives:

1. Functions
2. Pointers
3. Arrays
4. Structures
5. Dynamic memory allocation

Pre-processing directories start with #:

- #include <filename> or #include "filename" includes the content of a filename
- #define name replacement
- Will replace occurrences of name with replacement

```
Int main() {
    Printf("hello");
}
```

%d for integers, %f for floats, %s for chars

To execute – **gcc filename.c -o executablename**

./executablename

Declare variables prior to use like

```
int I;
```

To take user input:

- `Scanf() t`
- `Scanf("%d",&I);`

For loop syntax is standard

```
for (I=0, I<10, I++) {}
```

Anything non zero is true in C

% is remainder

Always define the function prior to calling it, or define a prototype before and call it after.

Pointers, Addresses, and Dereferencing

```
Int x=30;

&x # would print the hex value of the location
X # would print 30
*(&x) # would dereference the location of the variable and print 30

Int *ptr; # declares a pointer
```

I.e. it is going to store the address of an integer

If we want to store the address we can do

```
ptr=&x;
```

So instead of de-referencing the address, we can print *ptr to print 30.

Function(&x) # is a call by reference – it passes the location of x

In function we can have float function(*x) to show that we're receiving a pointer

Then we can do *x=*x+2 to change the value in the location referenced by the pointer. This will now be changed in the calling function.

Arrays

Int x[10] # declares an array
X[I] = 4 # stores 4 at the I'th location

Array elements are stored in contiguous memory locations

E.g.

```
&x[0] will be next to &x[1] will only be different by the number of bytes
used to store an integer – sizeof(int)

&x[0] or x will give the base address

(x+I) # is the same as (&x[I])
*(x+I)=2;  # same as x[I] = 2
```

Structures

Group together variables of different types in a single block

```
Struct employee {
     Char *name;
     Int age;
}

Struct employee emp1
Emp1.name = "john";
Emp1.age = 50;
```

If you're using a pointer to a struct, then use

```
ptr->age # to get variables, or alternatively, dereference and call
normally e.g.
struct employee emp1 = *ptr;

Int age = emp1.age;
```

Dynamic Memory Allocation

If memory needs to be assigned at runtime, dynamic memory allocation is to be used.

```
Int *ptr;
Ptr = (int *) malloc(20*sizeof(int));
```

Memory can be allocated dynamically using mallo, calloc, realloc functions

Malloc returns a void pointer pointing to the base address of the block of memory

Calloc has 2 args – number of units and size of each unit
Ptr = (int *) caloc(20, sizeofint(int));

If we want to grow an existing array, then we can use realloc to keep the existing data in the memory block

Check for null pointer to see if we have successfully allocated a block of memory

Use free to free up memory

Processes

18 October 2020 16:48

Objectives:

- Process definition
- Process in memory
- Program control block
- States of a process
- Process scheduling

A process is a program in execution
Programs sit on the disk as an executable, and are passive – they just contain data and instructions

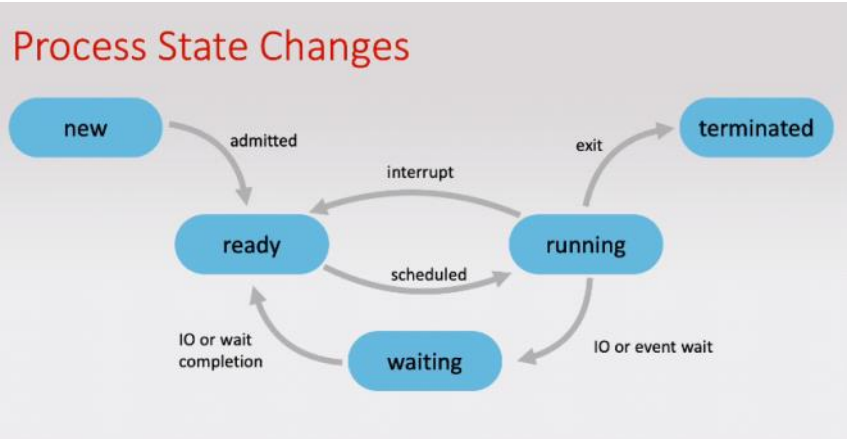
Processes are active. They are loaded into memory.

One program can create multiple processes.

A Process in Memory

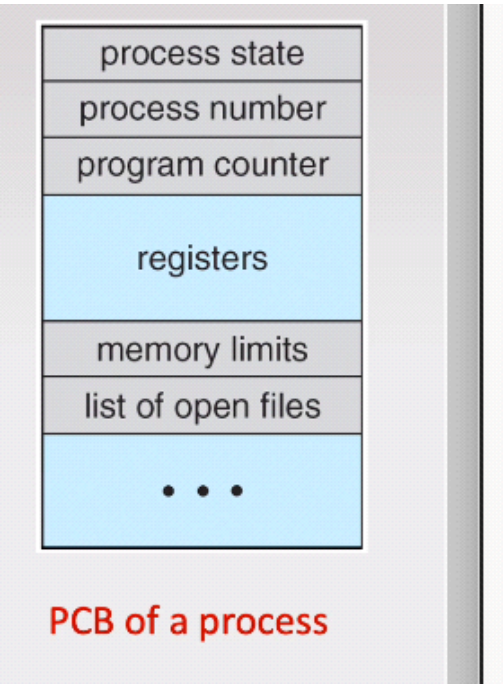
There is a virtual address space of a process that is allocated, in which the following reside:

- Text – stores the instructions
- Data – stores the global variables
- Heap – dynamically allocated memory
- Stack – stores local variables and function parameters such as return address of a function
- The space between the stack and heap allow them to grow and shrink during the program run-time.



Process Control Block

- Process state –running waiting ready
- Program counter – location of the next instruction
- CPU registers – contents of the CPU registers
- CPU scheduling information – prioriies, scheduling, queue pointers
- Memory management information – memory allocated to the process
- Accounting information – CPU used, time since state
- I/O status – list of open file / I/O devices



Process Scheduling

- To maximise CPU use, quickly switch process onto CPU for time sharing
- Process Scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes – job queue, ready queue, device queue

Short term scheduler

takes jobs from the ready queue, invoked once every 100ms. Schedules jobs from that queue

Long term scheduler

- When a process is finished executing and leaves the system. Maintains CPU bound and I/O jobs in main memory. Occurs much less frequently, controls the degree of multiprogramming, I.e. number of processes in memory. Tries to maintain a balance of jobs so that the system is stable and the queues don't grow out of hand. We have to strike a balance between the 2 types of jobs.
- Linux and windows don't have a long-term scheduler as they just dump all processes on the short-term scheduler.
- Linux/Windows assumes that the user themselves will notice a degradation in performance and kill jobs. Therefore, you are acting as the long term scheduler.

Processes Creation & Termination

25 October 2020 17:10

Objectives:

- Process creation
- Process termination

Process Creation

- A process may be created by another process
- Children processes can in turn create other processes forming a process tree
- Process are usually identified and managed via a PID

All user processes are rooted at PID 1 – the init process

The parent can choose various option

resource sharing options

- Files are shared between parents and children
- A subset of files are shared
- No resources are shared

execution options

- The parent may wait for the child to finish, or can continue concurrently

address space options

- The child's address space is a duplicate of that of its parents (has the same code, data and stack as parent)
- The child loads a new program into its address space

In unix, the fork() system call creates a new process with duplicate address space of the program

Process Termination

- Terminates automatically with exit or when last line is called
- Returns a status value to parent
- All resources of the process are released by OS

After a child terminates, and before the parent receives the code, the child is a zombie process. During the phase, all resources are released

What happens if a parent exists without invoking wait? The child is now an orphan process

In UNIX systems, the init process is assigned the parent

The init process periodically issues the wait() to collect the exit status of all orphan processes. This allows the exit status to be collected and released the orphans PID and process table entry

A parent process may terminate the execution of child processes using the abort() system call. This may happen if:

- If the child has exceed its allocated resources

- Task assigned to child is no longer required
- The parent is exiting and the OS does not allow a child to continue if its parent terminates – cascading termination

Process Communication

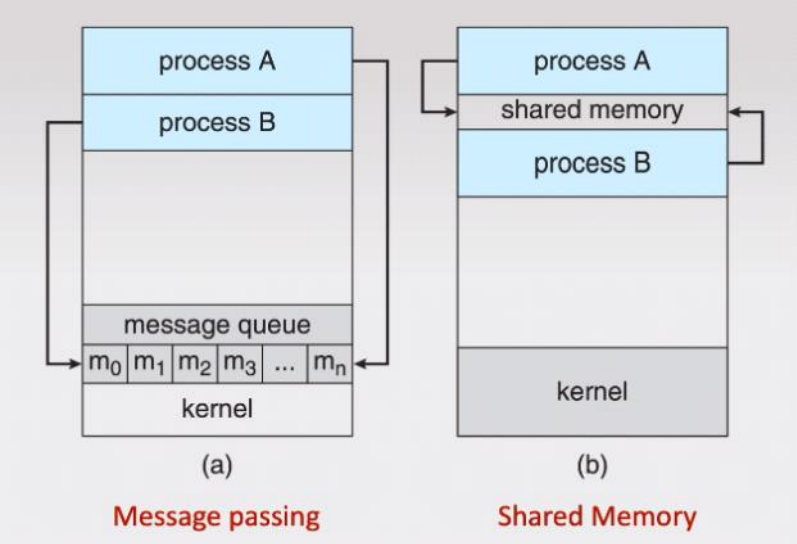
25 October 2020 17:37

Inter-Process Communication

Provide mechanisms for processes to communicate with each other, e.g. they're sharing data to complete a task

Two ways of inter-process communications

1. Shared memory
2. Message passing



Repeated system calls cause overhead in message passing.

Less overhead in the shared memory system because the only system calls necessary are the initial calls to allocate the shared memory

Both methods are commonly used

Message passing is useful for passing smaller amounts of data as no conflicts need to be avoided. It is also more easily implemented in distributed systems.

Share memory can be faster due to the lack of need for kernel intervention in system calls.

Shared Memory

One of the processes has to create a shared space, with the help of the kernel.

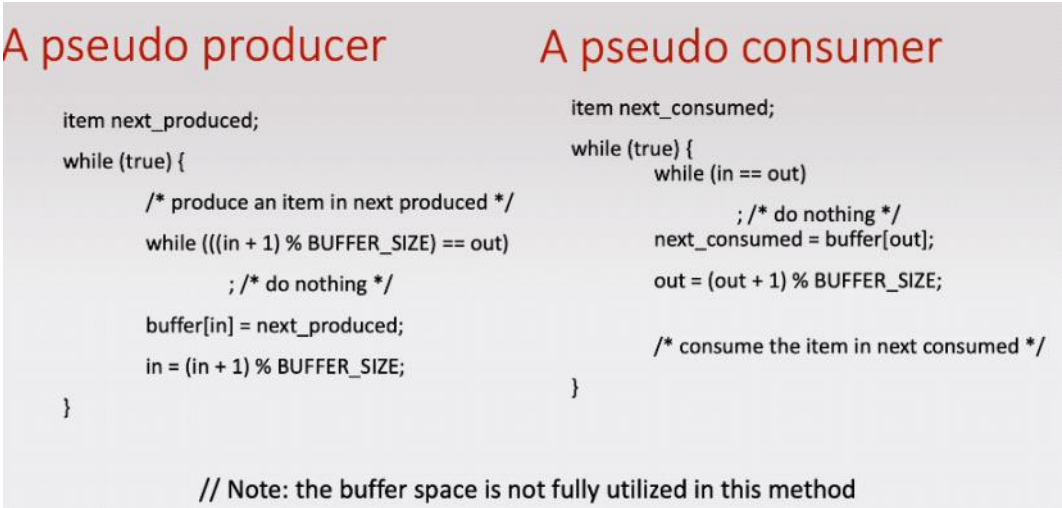
The programmer needs to make sure that both processes don't try to write to the shared memory at the same time.

A process is either a producer or a consumer at any given moment.

1. Consumer should wait when buffer empty or it will read garbage.
2. Producer must wait when the buffer is full

We can use a circular queue to solve the above two problems.

Why do we use a circular queue to solve this problem?



Message Passing

1. Needs a send and a receive operation
2. These are implemented using system calls
3. The kernel has to make sure the messages are sent and received correctly

Link implementation – direct or indirect
Synchronisation between send and receive
Buffer size representing the communication link

Direct Communication

1. Processes must name each other explicitly
2. Properties of communication link
3. Links are established automatically
4. A link is associated with exactly one pair of communicating processes
5. Can't hardcode the PID because every time the process runs its id can change

Indirect Communication – preferred
Sent and received through a mailbox, using a mailbox id. The receiver then reads from the box

Similar to shared memory system but the mailbox is managed by the kernel

Synchronous or Asynchronous

Blocking is considered synchronous

- Blocking send – the sender is blocked until the message is received
- Blocking receive – the receiver is blocked until a message is available

Non-blocking is considered asynchronous

- Non-blocking send – the sender sends the message and continues
- Non-blocking receive – the receiver receivers a valid message or a null message

Communication link is a buffer. The implementation of send and receive depends on the capacity of this buffer.

- Zero capacity – the queues a maximum length of zero, so sender must block until the recipient receives the message
- Bounded capacity – the queue has a finite length, when full the sender must block
- Unbounded capacity – the queues length is potentially infinite – sender never blocks

Communicating back to the parent with shared memory - in lecture at 20 mins

Ordinary Pipes

- Allow simple one way communication
- Connects the output of one process to the input of another

Named Pipes

- A problem with ordinary pipes is that they die when the process terminates
- Named pipes are more powerful – no parent-child relationship is required
- And once established, they persist

Introduction to Computer Networks

06 November 2020 09:48

- Networks and their functions
- Network delays and loss
- Packet switching vs Circuit switching
- Layered structure of the internet

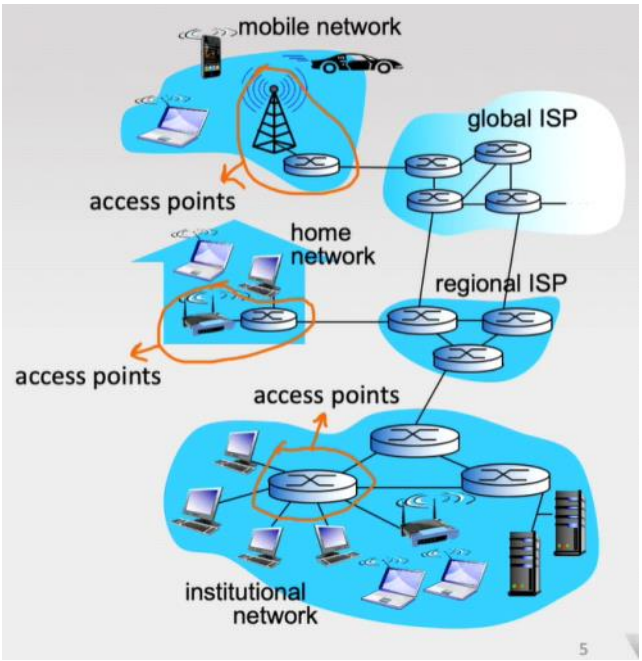
What is a Computer Network?

A network of inter-connected computing devices which enable processes running on different devices to communicate

Components

- Network edge consists of end hosts running apps
- Packet switches help forward data packets – switches and routers
- Communication links carry data packets between network devices e.g. fibre, copper, radio, satellite

End hosts connect to the internet through access points
AP's are provided by internet service providers
ISP's are organized in hierarchical structures
Internet is the network of ISP's



End Hosts

Run app processes which generate messages
Breaks down application messages into smaller chunks called packets
Adds additional information (headers) like port number and IP dest. We need port because end hosts run many processes so need to know which process the packet is for
Sends bits of a physical medium
If needed, provides reliable and orderly delivery of packets
Controls rate of transmission

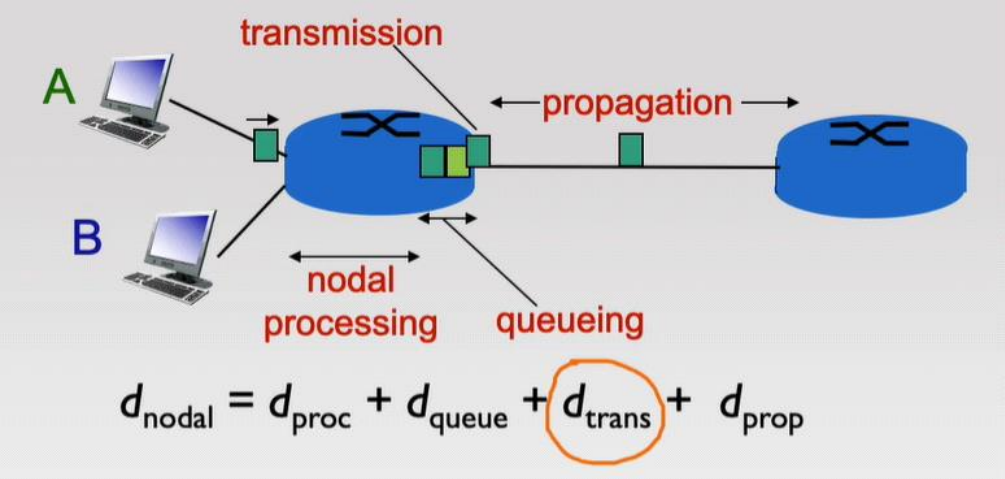
Access Points

End points connect to access points
Typically ethernet and WIFI to connect to routers

Network Core

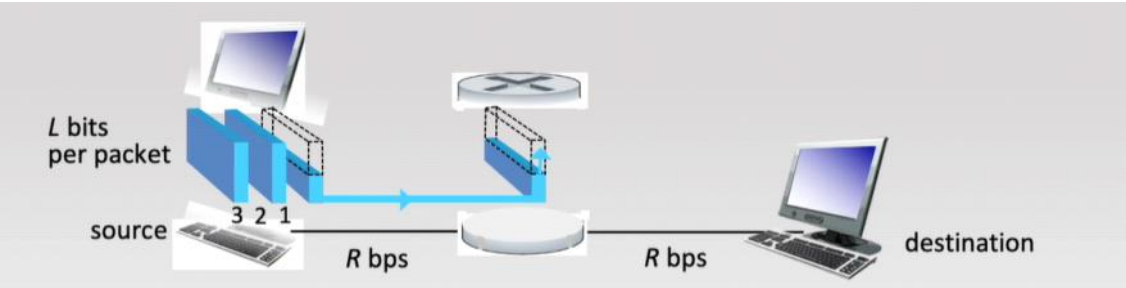
Packet switches run routing algorithms which compute the next hop router for the next hop router
Forwarding: once a packet arrives it is forwarded to the appropriate link

Packet Delay



Transmission Delay - Store-and-Forward Principle

Entire packets must arrive at router before it can be transmitted on next link



Thus it takes L/R seconds to transmit L-bit packets into a link at R bps

Queuing Delay

- If arrival rate to links exceeds transmission rate of links for a period of time then packets will queue, waiting to be transmitted on link
- Packets can be dropped if memory fills up

Propagation Delay

Length of physical link
Propagation speed in medium

Nodal Processing

Checking bit errors
Determining output link
Typically < msec

Throughput

We can talk about data transmission speed in terms of throughput
Specific to a flow or communicating pair
Rate at which bits are transferred from src to dest in a given time window

Protocol

Communicating nodes must agree on certain rules
A protocol defines rules for communication – what to do if a message is received and sequence and format of messages

Protocols define the format and order fo messages sent and received among network entities, and actions taken on message transmission and receipt

Network protocols can be implemented as software or as hardware

E.g. routers run protocols to forward message e.g. IP implemented as software
Network interface cards implement hardware protocols to send bits through physical mediums

Packet Switching

The internet uses packet switching
Different flows can share resources along their routes

If one flow is not using any of the shared links then the other flow can use it
Flow can change routes

However, due to the way resources are shared, packets may have to wait in a queue and can even be lost in buffer overflows

Hence we can use circuit switching in some cases as there is a guaranteed rate and no losses.

Circuit Switching

Was used in telephone networks
A circuit is reserved for each flow for the entire call duration

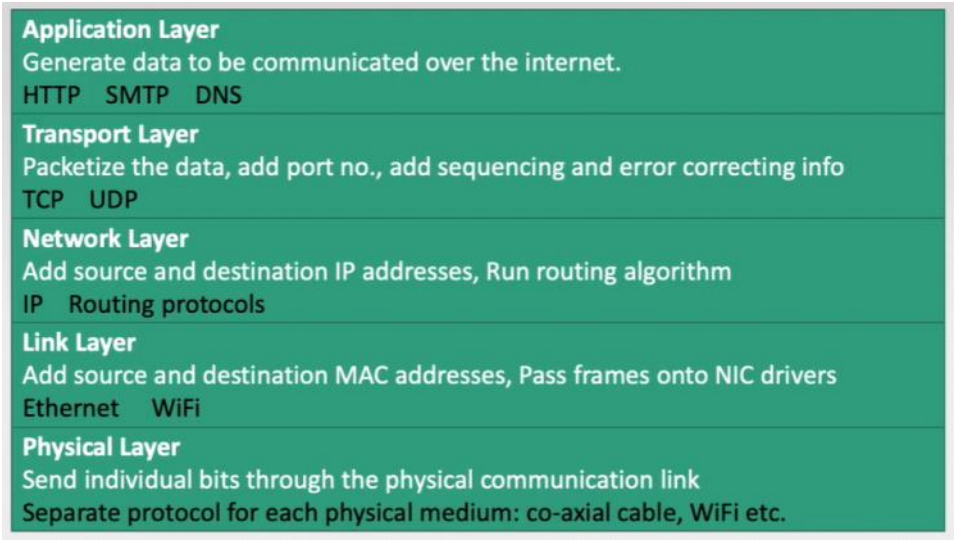
- Flows do not share resources
- If one flow is not using its assigned circuit during the call, it cannot be sued by another flow
- Not ideal for internet traffic which is bursty in nature

Layering

- Network devices perform complex functions
- It is better to divide these functions into layers
- Each layer performs a subset of functions
- Layer N uses the services on the layer below, N-1
- Similarly, the layer above N can use the services of N

The layering approach lets system designers change layers, adding or changing services without affecting other layers

The internet protocol stack has 5 layers containing all possible functions – all layers may not be present in a network device


Application Layer
Generate data to be communicated over the internet.
HTTP SMTP DNS
Transport Layer
Packetize the data, add port no., add sequencing and error correcting info
TCP UDP
Network Layer
Add source and destination IP addresses, Run routing algorithm
IP Routing protocols
Link Layer
Add source and destination MAC addresses, Pass frames onto NIC drivers
Ethernet WiFi
Physical Layer
Send individual bits through the physical communication link
Separate protocol for each physical medium: co-axial cable, WiFi etc.

Threads

06 November 2020 12:17

What are Threads?

A thread is unit of CPU execution

We can make a process multi-threaded. Each thread can handle a separate task
Threads share globals and open files, data, heap, signals

Each thread needs a *thread id, a program counter, a register set and a stack*

- Thread creation has less overhead
- Process creation has more overhead

Modern web-servers use multi-threading due to high traffic

Benefits of Threads

- Economy - cheaper than process creation because of lower overhead. Thread switching is faster than context switching
- Scalability – large number of concurrent tasks
- Responsiveness – allows threads to continue executing when others are blocked
- Resource sharing – resources are shared so easier on memory usage

Concurrency vs Parallelism

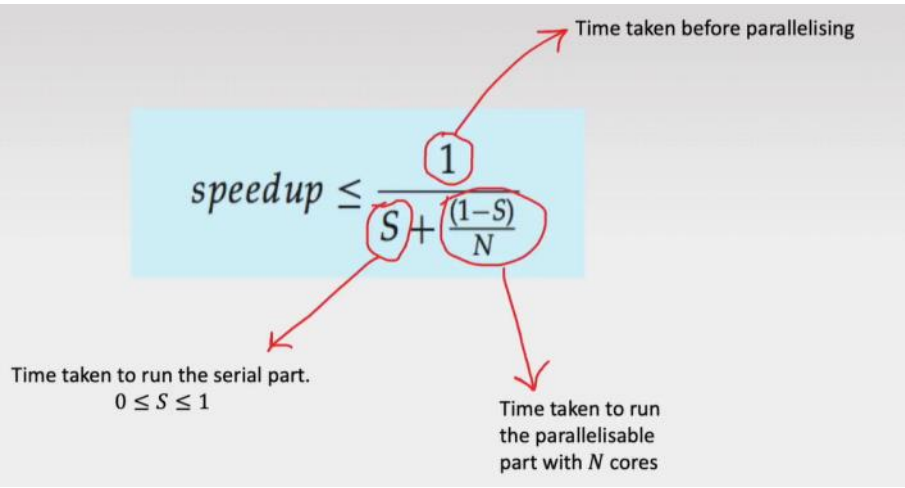
Concurrency

Supports more than one task making progress – a single CPU may appear to be running tasks concurrently by interleaving their execution

Parallelism

Different processes running simultaneously on multiple cores

1. Data parallelism distributes subset of the same data across multiple cores performing the same operation on each core – summing the contents of an array
2. Task parallelism splits threads performing different tasks across multiple cores



Thread Synchronisation

When multiple threads write to the same location, we must synchronise the threads
Synchronisation ensures that one thread does not overwrite the contents written by the other thread

Otherwise we end up with a race condition

To synchronise we can use mutex locks – mutual exclusion locks

Each thread must first acquire a lock to do an update

Thread Types

User level threads

- Implemented by user in user space
- Kernel not aware of them
- There is no kernel involvement, so less overhead
- Cannot run in parallel on different CPU's

Kernel Level Threads

- Implemented by kernel in the kernel space
- Kernel can schedule them on different CPU's
- Kernel managed so more overhead

Many-to-One Model

Many user level threads mapped to a single kernel thread

- Advantage – less overhead
- Disadvantage – multiple threads may not run in parallel because only one may be in kernel at a time

One blocking thread causes all to block

Multiple user levels threads are mapped to a smaller number of kernel level threads
User threads take turns to use the kernel

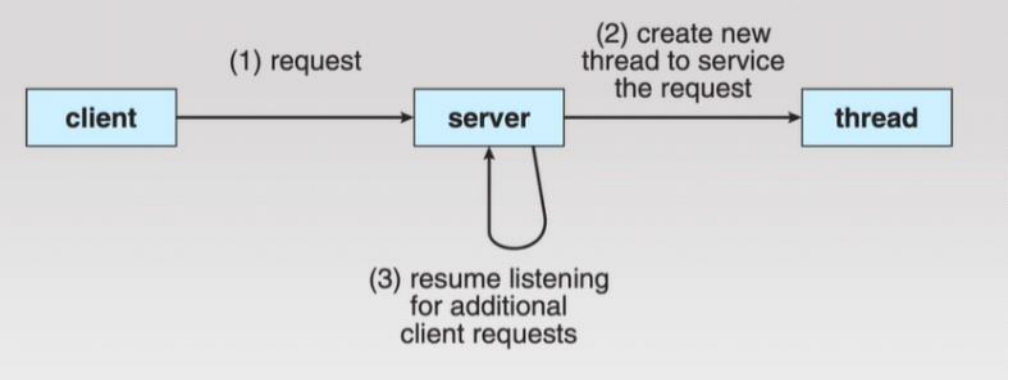
Programmers can decide how many kernel threads to use and then can run in parallel

One-to-One Model

Each user level thread maps to a kernel model

- Advantage – other threads can run even when one thread is blocking
- Different threads can be run on different cores in parallel
- Disadvantage – a user process can create a large number of kernel threads slowing the system down

One Thread Per Request



Server creating a separate thread to handle each client request

Problems

Making a new thread per client request can take a long time if kernel level threads
Large number of concurrent threads, slowing down the system

Thread Pool

- Main thread creates worker threads
- Each incoming request is added to a queue
- The workers pull requests from the pool queue

Advantages of Thread Pooling

- Usually slightly faster to service a request with an existing thread rather than creating a new thread
- Allows the number of threads in the applications to be bound to the size of the pool. This size is decided by the programmer taking into account the number of processing cores, memory in use, expected number of concurrent client requests

When there are no jobs for the thread, it continues to try processing jobs. This is inefficient.
One way of solving this problem is to use condition variables...

Condition Variables

- We can solve this using condition variables
- A thread can use wait() to wait on a condition variable
- Until some other thread signals the variable using signal() or broadcast()

Signal Handling

Signals are used in UNIX systems to notify a process about the occurrence of certain events

1. Synchronous – internally generated by a process e.g. division by 0 or illegal memory access
2. Asynchronous – externally generated e.g. terminating a process using ctrl+c sends the signal SIGINT

- All signals follow the same pattern – a signal is generated by an event, the signal is delivered to the process to which it applies and once delivered the signal is handled
- The signal can be handle in two ways, the kernel runs a default kernel handling routine
- By user-defined signal handling we generally override a general function used by the kernel

Application Layer

06 November 2020 10:59

Objectives

1. What is a network app
2. How network applications can communicate with other layers
3. Addressing app process running end hosts
4. Transport layer services available to an application process
5. Building network applications: Socket programming

What is a Network Application?

Processes running on different host machines can communicate by sending message over a network
The client process initiates the request for information, and is served by the server process

Developing a Network Application

Dev must develop both client side and the serve side of the program
Developing one side of the program is enough for applications specified by standard protocols

Sockets

User processes constitute the application process
Layers below the application layer are controlled by the kernel

The socket lies between the application and the kernel. **They are software doors between the transport layer and the application layer**

The application layer writes messages into the socket – it's like an API between the application and network
Creating, reading and writing to sockets is done with system call

Messages need to be addressed to the correct process running within the correct end host

Any internet device can be identified by IP addresses.

- Processes can be identified by port numbers
- Port numbers are 16 bit numbers
- Port numbers 0-1023 are reserved for well-known network applications
- Port numbers above 1023 can be used for application programs

Transport Layer Services

Application processes use transport layer services
All transport layer protocols offer some basic services - packetization, addressing, sequencing, error correcting bits

Additional services are sometimes needed – you have to choose the transport layer depending on what you need

TCP and UDP are the main protocols that can be chosen

TCP

Reliable in order data transfer - packets can be lost or arrive out of order but TCP ensures that all packets sent are received in-order

Connection oriented service – setup required between client and server processes before they start transferring data – TCP handshake

- Syn bit sent to server, carries no data but is used as a handshake
- When received, server allocates resources like a port
- The server sends back a syn-ack packet to acknowledge sync
- The client then sends the data back to the server

UDP

UDP provides no guarantees on data transfer
Best effort service – packetize data and sends it to the network
No effort is made to recover losses

UDP has less overhead so is very fast – no connection setup needed
Headers are smaller

Building Network Applications

Goal = learn how to build client/server applications that communicate using sockets

Socket programming creates sockets, reads from and writes to sockets through system calls
Two socket types – connectionless socket and connection oriented sockets

HTTP

Web browsers communicate with web servers using HTTP as the application layer protocol – HTTP is specified by IETF
HTTP uses TCP on port 80

An HTTP request returns a web page from the server – this consists of objects. An object is nothing but a fiel, e.g. a HTML file, JPEG etc, that is stored on the web server

Each object is addressable by a URL with a host name and a path name

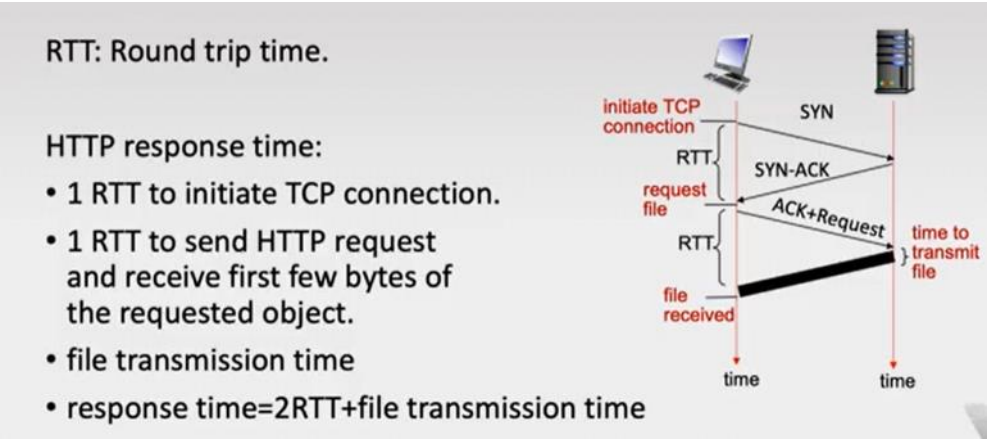
How HTTP Works

HTTP uses TCP – the server runs the server process on port 80
The client initiates a TCP connection to the server, on port 80
TCP connection is established after a TCP handshake which is a 3 way process starting with a SYN bit from the client, followed by a SYN ACK packet from the server, followed by an ACK bit from the server
Once this exchange is complete, the connection is closed

Non-Persistent HTTP Connections (v1.0)

Each object is obtained over a separate TCP connection – downloading multiple objects will require multiple TCP connections
1) SYN bit, server responds with SYN-ACK
2) client sends ACK and adds request for the base HTML file
3) The server establishes the connection and responds with the base HTML file
4) HTTP server closes TCP connection

5) Client receives the hTML file and examines it to find the 10 other referenced objects



Problems with Non-Persistent HTTP

Each object requires at least 2RTT to be downloaded
Each TCP connection the OS of the server has to allocate some resources. If there are many objects, then this can be a problem

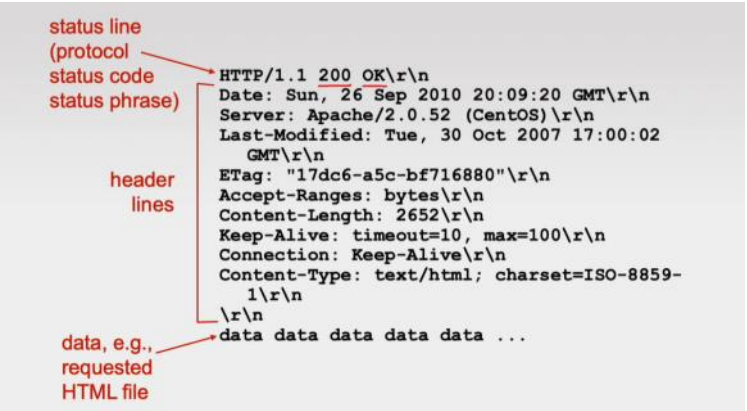
Persistent HTTP Connections (v1.1)

Multiple objects can be downloaded over a single TCP connection

- Server leaves the connection open after sending response
- Subsequent HTTP messages are exchanged over the open conection
- Client sends requests back to back as soon as it encounters a referenced object
- All objects can be downloaded in 2RTT + total data transfer time

HTTP Request Messages

HTTP Response Message



We can use web caches to speed up response time – this is because the cache is closer to the client

and so the full journey to the main server doesn't have to be made

Transport Layer

02 January 2021 11:00

Objectives

1. What is the transport layer of the internet?
2. Basic services provided by any transport protocol
3. Details of TCP services:
    a. Reliable data transfer
    b. Flow control
    c. Congestion control

What is the Transport Layer?

Transport layer services and protocols provide *logical communication between application processes* running on different hosts.

They abstract the details of the lower layers away. The communicating process on the sending side transmits the data to the transport layer, and the transport layer then handles the communication from there.

On the sending side, the transport layer *breaks the message into segments, adds headers, and passes the packets to the network layer*.

On the receiving side, *it reassembles segments into messages, and passes to the application layer*

TCP & UDP Evaluation

UDP

UDP provides bare minimum serves

- Packetization
- Adds headers
- Sends to network layer

If packets are lost or sent out of order, UDP makes no efforts to fix this.
UDP is connection-less – each UDP segment is treated independently.
UDP senders can send at any rate they wish – there is no congestion control

UDP is used because it has less overhead and it is fast

- no connection establishment
- Smaller header size because doesn't need to deliver packets in correct order
- No congestion control – UDP can blast as fast as it wants
- For video streaming, it is not important to be lossless, so UDP is useful

If UDP users want reliable data transfer, they have to add it themselves into the app later

TCP

- Reliable data transfer – recovers losses, re-orders out of order packets
- Flow control – matches the sending speed of the sender to the reading speed of the receiver
- Congestion control – controls the sending rate according to perceived network congestion

Within the internet packets can be lost, corrupted or arrive out of order

- TCP ensures none of this is perceives by the app process
- Hence, TCP has to enhance the unreliable network service

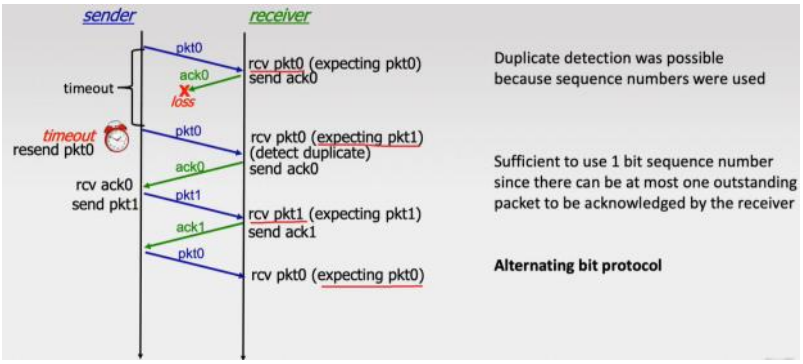
Reliable Data Transfer

- Bit errors can happen because of electrical noise
- Data can get lost in buffers at routers

Checksums – include the sum of all 16-bit words in the header
ACKS – indicates if a packet is correctly received at the receiver
Timeout mechanism – sender times out if ack is not received within a timeout interval
Retransmissions – to retransmit lost or corrupted packets – automatic repeat request (ARQ)
Sequence number – to detect duplicates at the receiver – packets can be duplicated if ACK is lost or corrupted

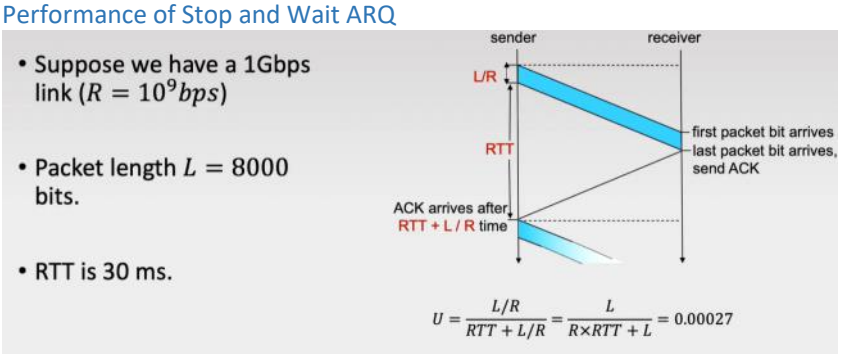
Stop and Wait ARQ

1. Sender sends packet
2. Waits until it receives an ACK
3. If ACK arrives, send next packet
4. Else, time out and retransmit the same packet



Provides reliable data transfer.

Performance of Stop and Wait ARQ



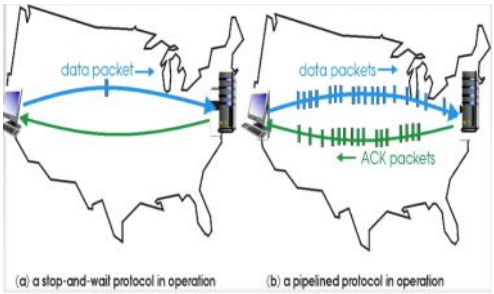
- Sender has to wait until an ACK is received for the previously transmitted packet to transmit the next
- Poor utilisation of the link capacity – poor performance

Bits in the Pipeline

- The receiving process typically has a finite buffer of B bits
- The receiving process may not be reading from its buffer all the time
- Hence, to avoid buffer overflow, the sender should not send more than B bits at a time
- Length of pipeline = L + R x RTT bits
- Buffer size = B bits
- Max no of bits without waiting for ACK = min (B, L + R * RTT)

Pipeline Protocols

- Allow multiple unacknowledged packets in the pipeline
- ACK is send individually or cumulatively
- Range of sequence numbers must be increased



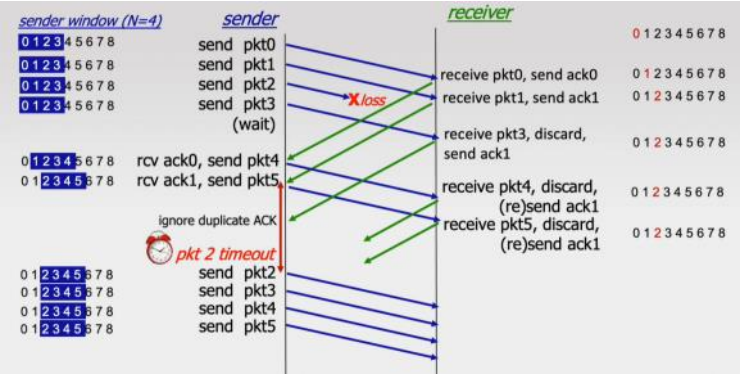
Generic Pipeline Protocols

There are 2 generic pipeline protocols that are common. TCP uses a combination of both Go-Back-N, and Selective Repeat.

Go-Back-N

- Sender can send up to N packets without ACK
- N = window size, depends on the delay-bandwidth product, receive buffer size and other factors
- The receiver maintains a variable *expectedseqnum* which keeps track of the next expected sequence number to be received
- If the receiver correctly receives packet n and n = *expectedseqnum*, then it sends ACK(n) which acknowledges all packets up to and including packet n – cumulative ack – increments

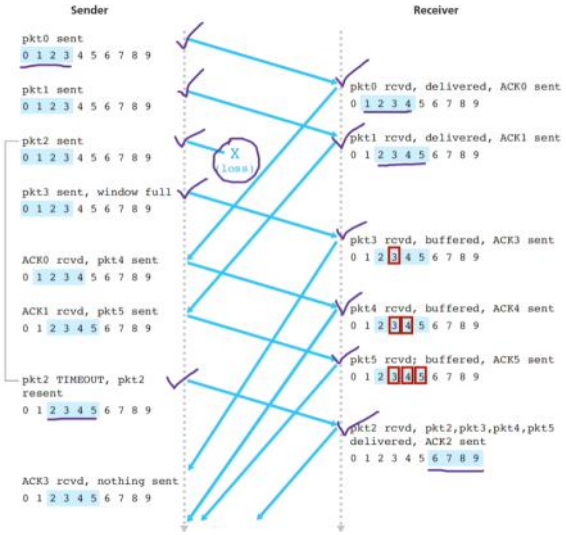
expectedseqnum by 1
- In all other cases, n != *expectedseqnum*, the receiver discards the incoming packet and sends ACK(expected – 1)



Selective Repeat

- Out of order packets are not discarded if they are within the receivers window
- ACKs are individual, not cumulative
- This allows sender to retransmit the unacknowledged packets

The SR sender does not have to retransmit out-of-order packets



TCP implements a combination or GBN and SR protocols

- Like GBN, TCP uses cumulative ACKs
- Like SR, the TCP sender only retransmits the segment causing the timeout

Sequence Numbers

To understand TCP, we need to understand how sequence numbers are given to TCP segments

In TCP, each byte of data is numbered
Sequence number of a transmitted TCP segment is the byte number of the first byte of the segment

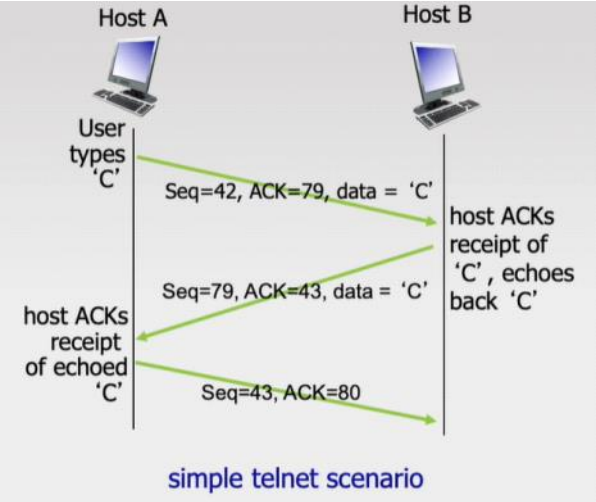
In TCP, the ACK number is the number of the next expected byte form the other side

TCP uses cumulative ACKs – when it receives packet 100 it will confirm 1-99

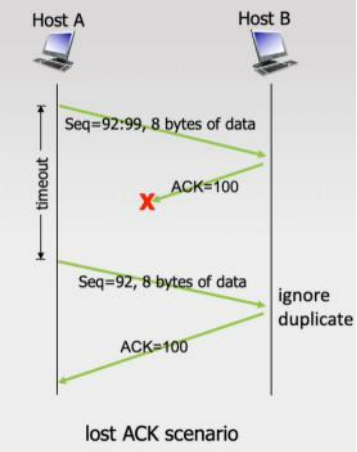
In TCP, data can flow in both directions

To reduce the number of transmissions, TCP piggybacks ACKs on data segments

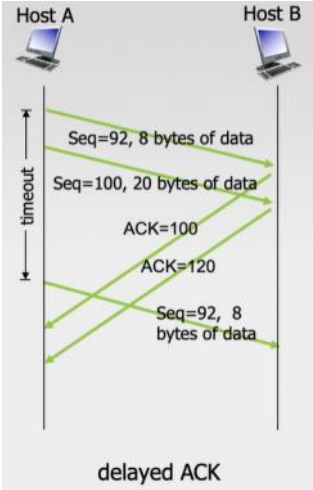
A segment can carry data and serve as an ACK



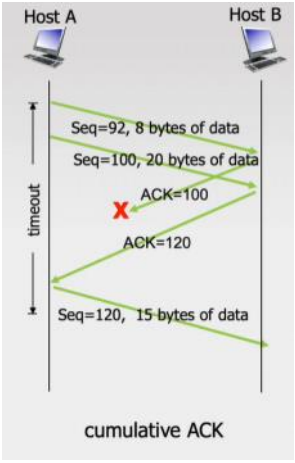
TCP Retransmission Scenarios



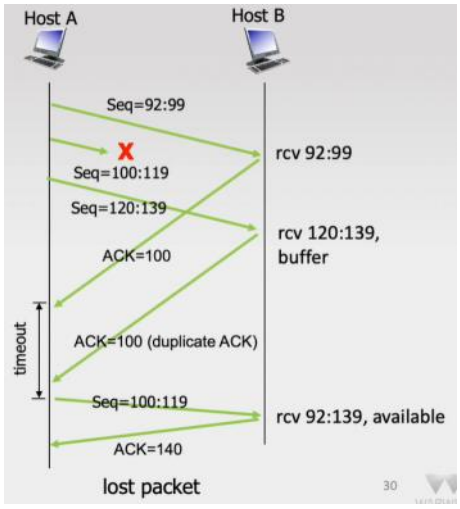
ACK is lost on its way – after a while, the sender will timeout and resend the bit. The duplicate will be ignored by the receiver, but the ACK will be send



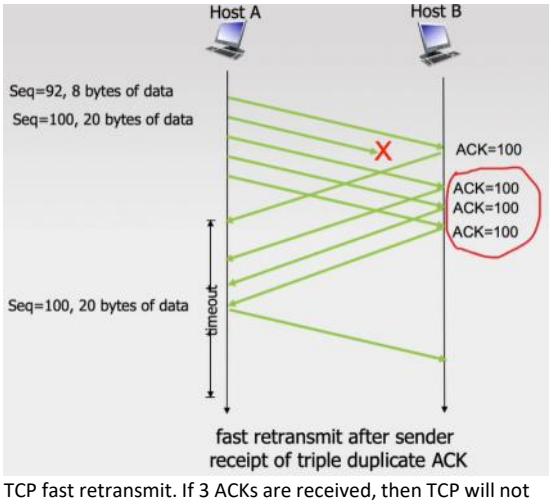
When an ACK is delayed – the sender sends 2 segments (92 and 100). The receiver sends ACK 100 and ACK 120. By the time the receiver sends these acks however, the sender has timed out. So, the sender retransmits the oldest segment in its window (92)



Even though the ACK 100 is lost, the next ACK does the job



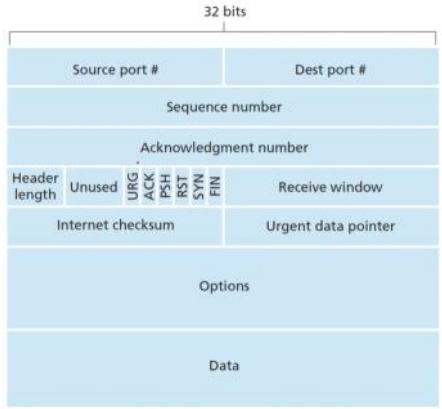
Sender times out waiting for confirmation of 100:119, then retransmits
TImout period is long, so there is a long delay before resending lost packet
TCP fast retransmit – duplicate ACKs are good indicators of packet loss – they are sent when there is a gap in the received stream of bytes
Upon receiving 3 duplicate ACKs for a segment, TCP



TCP fast retransmit. If 3 ACKs are received, then TCP will not wait for the timeout, but rather immediately send the unacknowledged packet. Note that 3 ACKs is much less severe than a timeout, as discussed later

sender retransmits that segment without waiting for timeout to occur

TCP Segment Structure



1. Sequence Number – 32 bit sequence number of the segment indicating the number of the first byte
2. Acknowledgement Number – number of the next byte expected
3. Receive window – used for flow control

Flow Control

- The data in the pipeline should not exceed the receive buffer size
- Otherwise, the receive buffer will overflow and data will be lost

How does the sender know the space in the receivers buffer? We have to use flow control

Receiver advertises free buffer space in the **receive window** field
Sender limits amount of un-ACK'd "in flight" data to the receive window size

Congestion Control

- Reliable data transfer and flow control are directly used by applications
- Congestion control is beneficial for the network as a whole
- TCP Provides congestion control
- Controls the rate of transmission according to the level of perceived congestion
- Occurs at a router when the input rate > output rate

Queues in router grow, and the buffer overflows, resulting in lost packets and queue delays. This happens when senders send data too fast

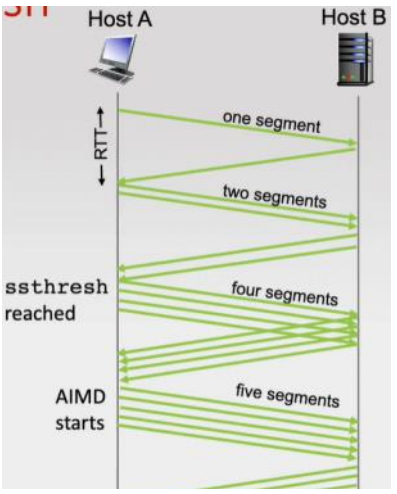
Detecting Congestion

- Detects network congestion through losses and delays
- A TCP sender assumes the network to be congested when a timeout occurs
- Three duplicate ACKS are received

TCP treats these events differently – **A timeout event is taken more seriously** than the reception of 3 duplicate ACKs

Controlling Congestion

- TCP is a window based protocol
- The rate of the sender can be controlled by controlling the size of the send window
- By controlling the window size, we can control the transmission rate



MSS = maximum segment size
Determined by the maximum frame size specified by the link layer protocol

How is the congestion window size varied?

Additive Increase Multiplicative Decrease (**AIMD**)

1. Sender linearly increases the transmission rate until a loss occurs
2. When a loss occurs, decrease the rate by a factor of 2
3. Additive increase – increase cwnd by 1 MSS every RTT until loss
4. Multiplicative decrease – cut cwnd in half every loss

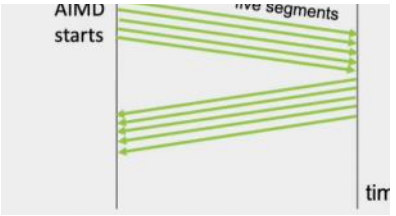
Why AIMD in TCP?

Achieves fair rate allocating among competing flows. Eventually will converge around a fair size for each flow

- Convergence speed of AIMD is low

Achieves fair rate allocating among competing flows. Eventually will converge around a fair size for each flow

- Convergence speed of AIMD is low
- To avoid waiting long, TCP uses an initial slow start phase where the window size is increased exponentially fast starting at a small value (slow start) until a threshold is met or a loss is detected
- This ensures that the sender reaches the right operating speed quickly



An example of slow start in action – MSS grows exponentially until it reaches ssthres (4 bytes here) then grows linearly

Reacting to Losses – Timeout and Duplicate ACKs

- Losses are detected through timeouts and 3 duplicate ACKs
- Duplicate ACKs – some segments are lost but some are received – be lenient
- Timeout – no segment is receives – take drastic measure

1. **Timeout – drastic action**: ssthresh = 0.5 cwnd, cwnd = 1MSS. After that the sender enters into a slow start phase
2. **3 Duplicate ACKs – take les drastic action** – ssthresh = 0.5 cwnd, cwnd = ½ cwnd and window then grows linearly

Deadlocks

02 January 2021 13:23

Objectives

- Definition
- Conditions
- Deadlock detection
- Deadlock prevention and avoidance algorithms

What is a Deadlock?

A set of processes is in a deadlock when each process in the set is waiting for an event that can be caused only by another process in the set

The events we are interested in are acquisition or release of some type of resource

- Mutex locks
- CPU
- File
- I/O devices

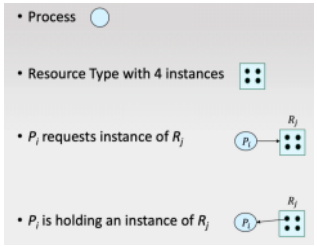
Necessary Conditions for Deadlock

- Mutual exclusion – only one process at a time can use a resource
- Hold and Wait – there must be a process holding some resources while waiting to acquire additional resources held by other processes
- No preemption – a resource can be released only voluntarily, and it cannot be snatched
- Circular wait – there must exist a subset of processes waiting for each other in a circular manner

Resource allocation graph

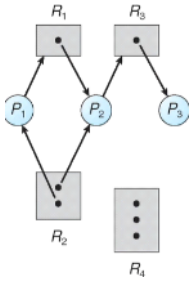
A directed graph G=(V,E) where

- V is partitioned into two types:
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$



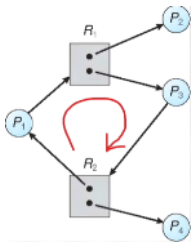
Resource Allocation Graph Example 1

- P1 has acquired an instance of R2 and is waiting for R1
- P2 has acquired an instance of R2 and is waiting for R3
- R3 has already been acquired by P3



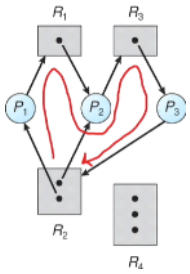
There is no deadlock in this graph. Once P3 finishes, it will release R3 and then finish, giving R3 to P2, which will then finish

Resource Allocation Graph Example 2



Resource Allocation Graph Example 3

There is a cycle in the graph, but there is no deadlock.



There is a cycle, and insufficient resources for any of the cyclic processes to finish, so they can't free up the resources needed by the other processes.

There is a deadlock

- If there is no cycle, there cannot be a deadlock
- If there is a cycle, there might be a deadlock

Algorithms for Deadlock Detection

	Allocation			Currently Available			Request		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P1	0	1	0	0	0	0	1	0	0
P2	1	1	0				0	0	1
P3	0	0	1				0	0	0

- Allocation represents current allocation of resource to Process
- Currently available represents how much is available in a system e.g. at the moment there are no resources available
- Requests represents pending requests - e.g. p2 is requesting 0 r1, 0 r2 and 1 r3 – this is pending

We also need a Boolean flag corresponding to whether a process has finished

Handling Deadlocks

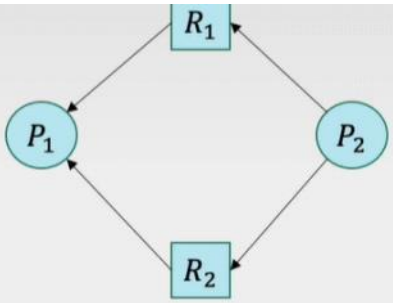
Deadlock Prevention

If we can ensure one of the necessary conditions doesn't hold, we can ensure the system is deadlock free

- Mutual exclusion – it is difficult to avoid this
- Hold and Wait – avoiding this means we have to ensure that when a process requests a resource, it does not hold any other resources
- No preemption - if we allow snatching of resources, we can avoid deadlocks
- Circular wait - we could give each resource a number and require that processes request resources in order of number

This is restrictive – harmless requests could be blocked e.g.

- Consider the following system:
  - P_1 and P_2 each requires resources R_1 and R_2
  - Resources are numbered: $n(R_2) > n(R_1)$ Each process must request resources in the increasing order
  - If R_2 is already held by P_1, then the request of R_1 by P_1 will be blocked (not granted)
  - But it is safe to grant this request since there will be no cycle in the resulting system



Deadlock Avoidance

- Better than deadlock prevention
- Determine whether a request should be granted based on whether it will leave the system in a safe state
- Safe state is a state when a deadlock can't occur

How do we know if a state is safe?

We need to know what states can arrive in the future

Deadlock Avoidance Algorithm

- Needs advanced information on resource requirements for each process
- Each process declares the max number of instances of each resource that It may need
- Upon receiving a resource request, the deadlock avoidance algorithm checks if the granting of the resource leaves the system in a safe state
- If so, grant the request, otherwise wait until the state of the system changes to a state where the request can be granted safely

How to determine whether a state is safe

- State is cycle free
- Even if there is a cycle, it can still be safe
- So, we use the Banker's Safety Algorithm

Banker's Safety Algorithm

- 5 processes, p0, p1, p2, p3 p4
- 3 resources, a=10, b=5, c=7

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

1. P1 needs 322, which we can service
2. So, we execute it and claim the processes resources back.
3. Then we have 532 available

Resource Request Algorithm

Pretend the request is granted and determine if the resulting state is safe using Banker's safety algorithm

- If so, grant request immediately
- Otherwise keep the request pending until state change

Main Memory

02 January 2021 16:03

Objectives:

1. How does the OS manage main memory?
    a. Memory protection
    b. Address binding
    c. Logical and physical addresses
2. Explore various ways of allocating memory to processes
    a. Contiguous memory allocation
    b. Segmentation
    c. Paging
    - Page table implementation
    - Page table structure
      -Hierarchical page table
      -Hashed page table
      -Inverted page table

CPU and Memory

- Memory is an array of bytes each having an address
- A program must be taken from disk into memory to be executed
- CPU can read instructions and data from memory and store data into the memory
- CPU fetches the next instruction from address stored in the Program Counter
- Execution of the instruction may involve reading data for memory or writing data into memory
- In either case, memory accessed through an address

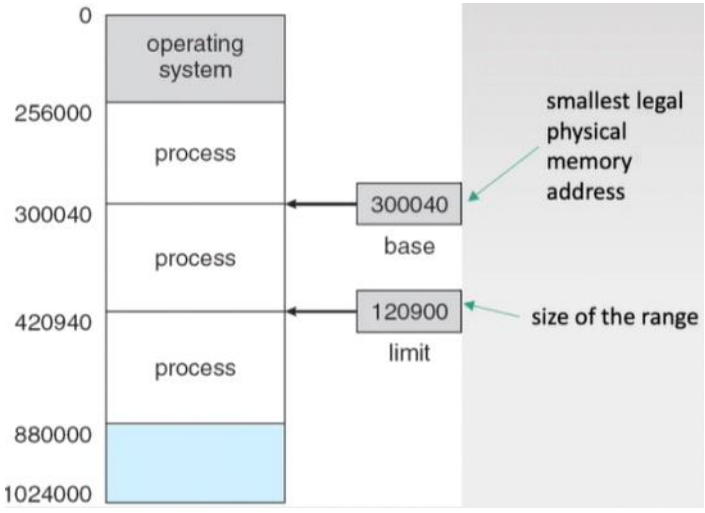
Memory Protection

Addresses accessible by a process must be unique to that process – no other process should be able ot access the same set of addresses

If not, then processes can write into each other's address space – big security concern

Base and Limit Registers

A pair of base and limit registers define the range of legal addresses
The OS loads these registers when a process is scheduled – allows only the OS to change the base and limit as required



- CPU checks if each address falls within the legal range
- If not, it lets the OS know which takes necessary actions to prevent the memory access

Method assumes contiguous memory allocation, but other methods exist.

Address Binding

A program usually resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process.

Depending on memory management, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brough into memory for execution form the input queue.

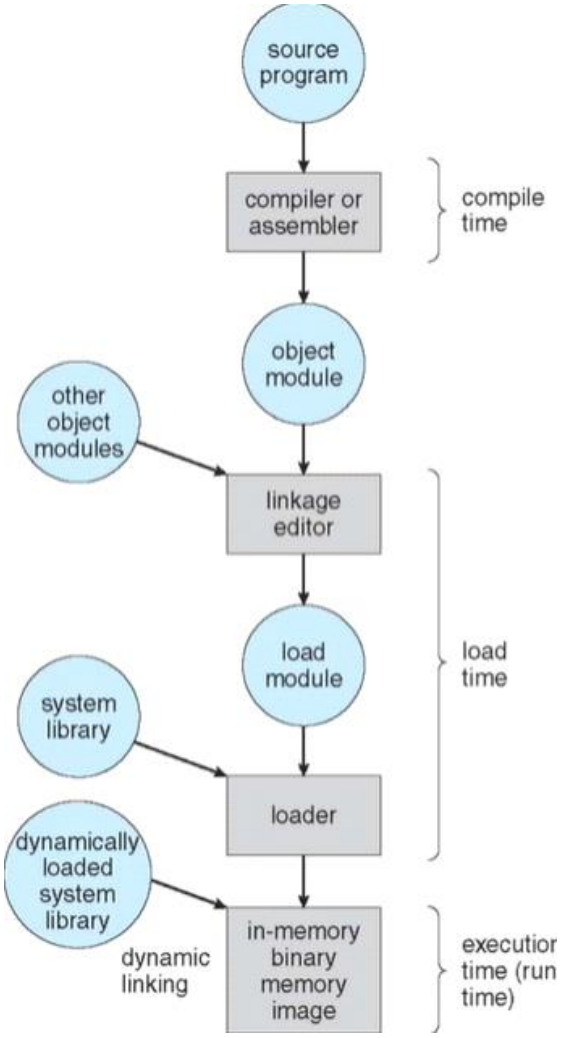
Generally, the user program may go through many steps before being executed. Addresses are represented in different ways at different stages of a programs life
e.g. **variable, count, function** refer to memory addresses that depend on step we're in in the program execution.

- When we compile the code, we **bind** the symbolic addresses to relocatable addresses e.g. "14 bytes from the beginning of this module, you will find X"

The linkage editor or loader in turn binds the relocatable addresses to absolute addresses such as 74014. Each binding is a mapping from one address space to another

We can bind instructions and data to memory addresses at any step along the way:

1. **Compile time** – if you know at compile time where the process will reside in memory, then absolute code can be generated. For example if you know that a user process will reside starting at location R, then the generated compiler code will start from that location and extend from there. However, if the location of the process changes, then the starting location will change and it will be necessary to recompile the program.
2. **Load time** – if it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. If the starting address changes between compilation and execution, we need only reload the user code to incorporate the changed value. Final binding is delayed until load time.
3. **Execution time** – If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Most general purpose OS's use this method, but it does require special hardware that we will discuss.



Logical Versus Physical Address Space

An address generated by the CPU is typically called a **logical address**, whereas an address seen by the memory unit, that is one loaded into the MAR, is commonly referred to as a **physical address**.

The *compile-time and load-time address-bindings methods generate identical logical and physical addresses*. However, the execution-time address binding scheme results in different logical and physical addresses.

In this case we usually refer to the logical address as a **virtual address.**

- The set of all logical addresses generated by the program make up the logical address space.
- The set of physical addresses corresponding to these logical addresses ais a physical address space

Thus in execution-time binding, we have a different physical and virtual address space.

During run time, the mapping of the physical address space to the logical address space is done by the MMU (Memory Management Unit). We can choose many different methods to achieve this mapping.

A simple example of an MMU scheme would be to have a value of 14000 in the relocation register. Thus, when we try to access logical address 345, the MMU maps that to 14345.

The user program deals with these logical addresses, and so can manipulate 345 as much as it wants, without knowing that it's actually being remapped to 14345. The Memory Mapping hardware converts these logical addresses into physical addresses outside the program.

Contiguous Memory Allocation

Main memory must accommodate the OS and the user processes. We therefore need to allocate main memory in the most efficient way possible. One method is CMA.

The memory is usually divided into two parts – one for the OS and one for the user processes.

In CMA each process is contained in a single section of memory that is contiguous to the section containing the next process.

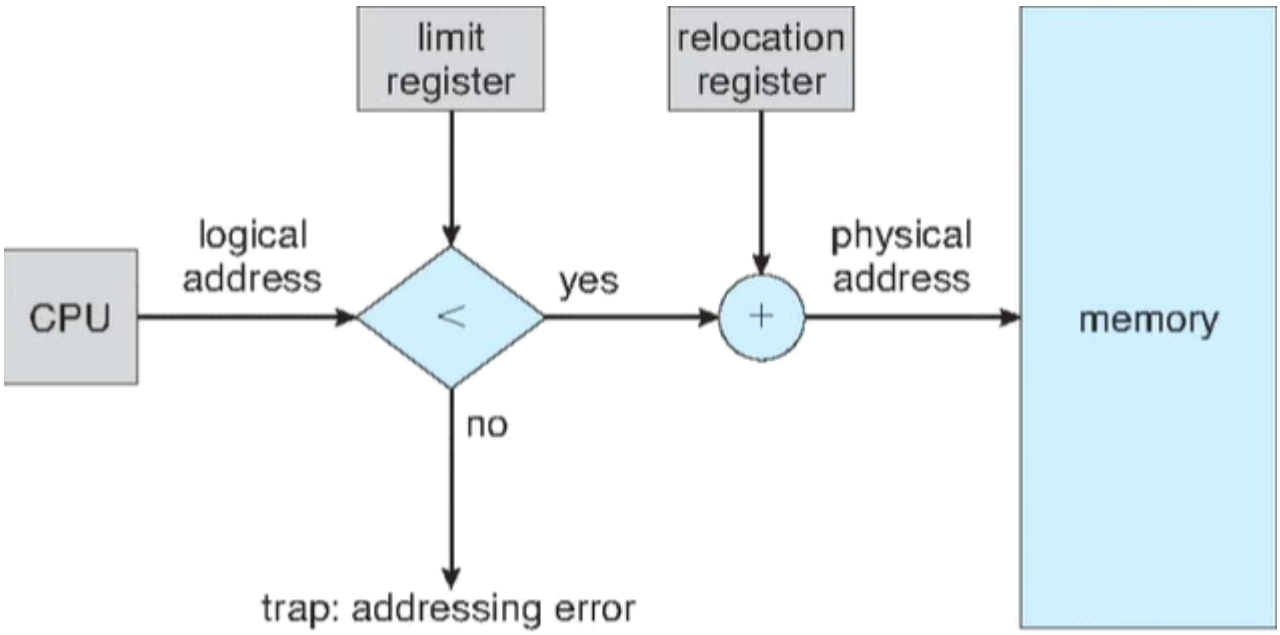
Memory Protection

We can prevent a process from accessing memory that it does not own by combining two ideas – the relocation register and the limit register.

The relocation register contains the value of the smallest address physical address
The limit register contains the range of logical addresses
e.g. relocation = 100040 and limit = 74600. Each logical address must fall within the range specified by the limit register.

The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory.

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked against these registers ,we can protect both the operating system and the other users programs and data from being modified by this running process.



Memory Allocation

One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process.

In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

In the variable partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for the user processes and is considered one large block of available memory, a hole.

At any given time we have an input queue, and a list of available hole sizes. The OS can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied.

The OS then waits until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

1. **First fit** – this allocation method finds the first hole that is big enough to fit the process. We stop searching when we find a hole that is large enough
2. **Best fit** – Allocate the smallest hole that is big enough .We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
3. **Worst fit** – allocate the largest hole – we must search the entire list unless it is sorted. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Best-fit and first-fit are better than worst-fit in terms of storage utilisation. First-fit is generally fastest.

Fragmentation

Both first-fit and best-fit strategies for memory allocation suffer from external fragmentation. As processes are loaded and removed, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous – storage is fragmented into a large number of small holes.

50% Rule – given N allocated blocks, another 0.5N will be unusable.

If we have a block of 18,723 bytes and we have a process that requests 18,720 bytes, then there will be 3 bytes left over. Keeping track of this small hole costs substantially more than the 3 bytes we're keeping track of.

To avoid this, we break down the memory into fixed size blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation – unused memory internal to a partition.

Solution to External Fragmentation

- **Compaction** – shuffle memory contents to place all free memory together. Not possible sometimes, e.g. when relocation is static and done at assembly or load time. Possible only when relocation is dynamic and done at execution time.
- **Permit logical address space to be non-contiguous** – allows a process to be allocated physical memory wherever such memory is available. Two techniques achieve this solution – segmentation and paging . These techniques can be combined.

Segmentation

The user's view of memory is not the same as the actual physical memory. Segmentation provides a way of mapping the programmer's view to the actual physical memory, so that the system has more freedom to manage memory.

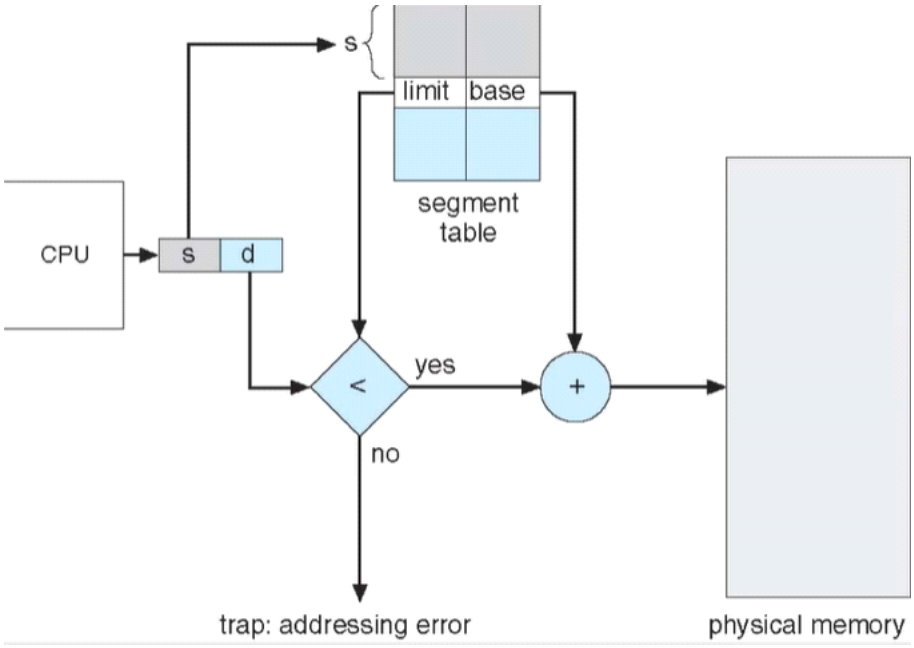
Segmentation is a memory management method that supports the programmers view that each

section of code is stored in its own segment of memory. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities – a segment and an offset.

<**segment-number**, **offset**>

A C compiler might create separate segments for the following:

- The code
- Global variables
- The heap, from which memory is allocated
- The stacks used by each thread
- The standard C library



Paging

Summary of method:

- Divide program into fixed-sized blocks called pages
- Divide physical memory into fixed sized-blocks called frames
- Page size = frame size = e.g. 4kb

Segmentation permits the physical address space of a process to be non-contiguous. Paging is another memory management scheme that offers this advantage. However, paging avoids external fragmentation and the need for compaction, whereas segmentation does not.

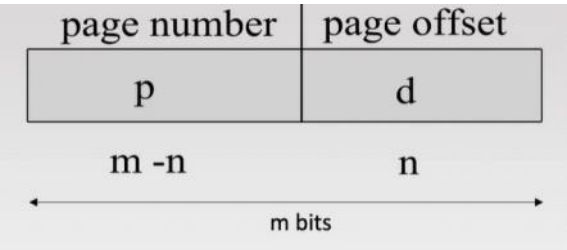
It also solves the problem of fitting memory chunks of varying sizes onto the backing store.

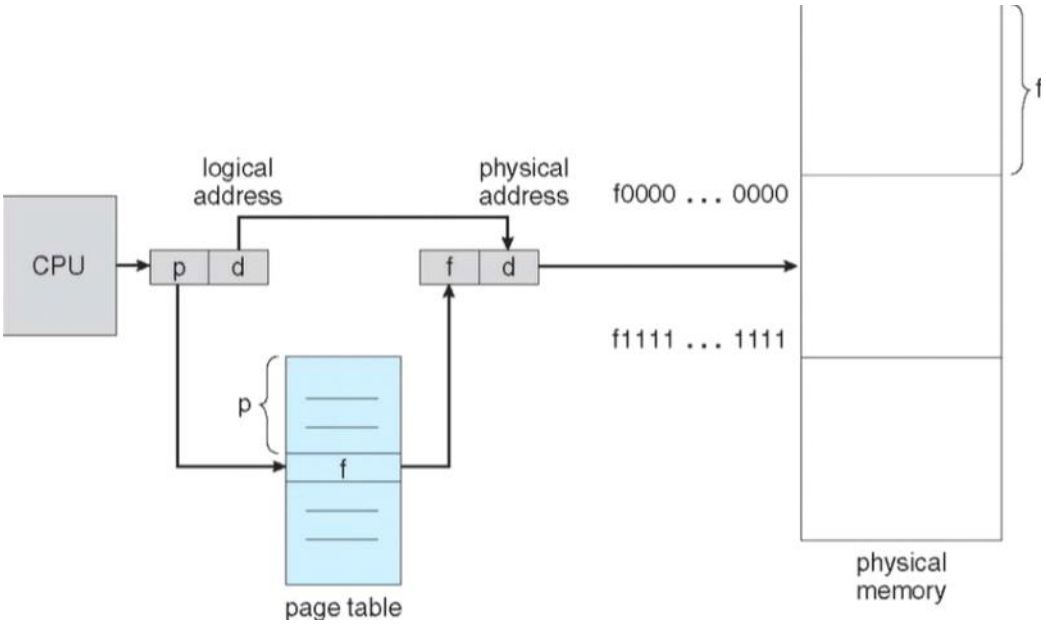
The method involves breaking down the memory into fixed-size blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source.

Each address generated by the CPU is divided into two parts – the **page number** and the **page table**. The page number is used as an index in the page table. The page table contains the base address of each apge in phyiscal memory. *This base address is combined with the page offset to define the physical memory adderess that is sent to the memory unit.*

Page number – used as an index into a page table which contains base address of each page in physical memory
Page offset - combined with base address to define the physical memory address that is sent to the

memory unit





We end up with some internal fragmentation - if the page size is 2048 bytes and the process is 72,766 bytes, the we end up with 35 frames plus 1086 bytes. Thus, we need to allocate 36 bytes and end up with 962 wasted bytes.

Are smaller frames desirable?

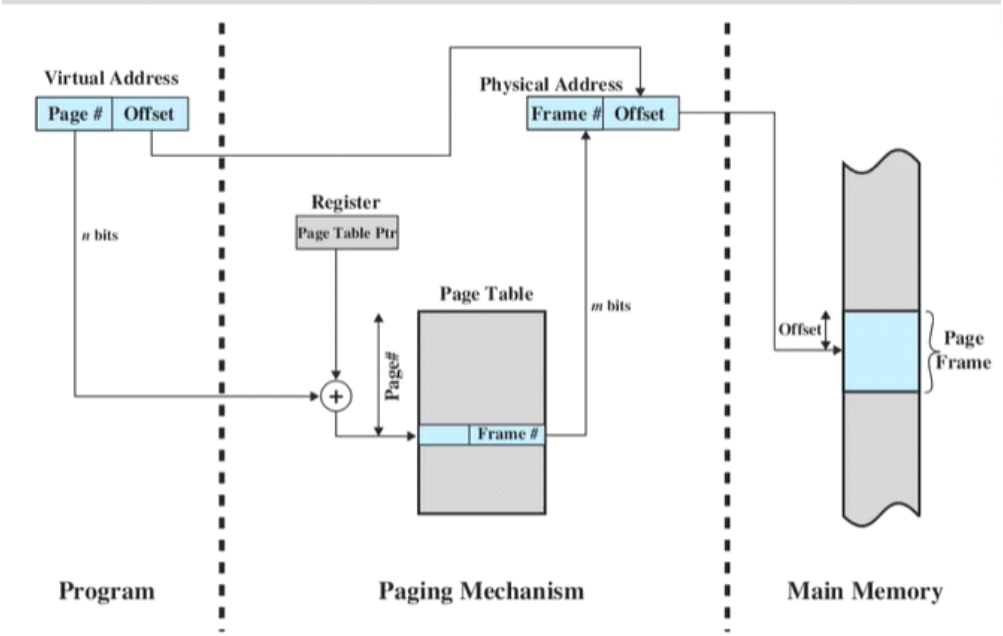
- Smaller frames mean larger page tables
- Page size have grown over time – with growing data sets and main memory

Page Tables

- Remember that the page table stores the mapping between page numbers and the frame numbers.
- OS keeps a page table in the memory for each process

MMU can consist of a set of fast registers to hold page table entries

- These registers are loaded by the OS when a process is scheduled
- Fast registers are of limited size – not suitable for systems with millions of page table entries
- e.g. 32 bit logical addresses, 4KB page size => 2^20 page table entries



Instead, we can access the page table directly from the memory. We need to store the base address of a table in a register called page table base register (PTBR)

The problem with this approach is that the overhead of memory access doubles – you have to access memory to get the page information before you access memory again

Instead, we use a translation lookaside buffer (TLB)
TLB stores < 256 frequently used page table entries – lookup is very fast
If the page is not found in the TLB, then it's a cache miss. The actual page table has to be found

When the table entry is brought from memory, it evicts an entry of the TLB.

Many cache replacement algorithms can be considered.

- One popular choice is least used (LRU)
- Each algorithm has a corresponding hit ratio.
- The effective access time depends on the hit ration
- EAT = hit ration x 1 x memory access + (1- hit ration) * 2 * memory access

E.g. with a 95% hit ration and a 100ns access time, what is the EAT?
0.95 x 100 + (0.05) * 200 = 105ns

The TLB can store page table entries for multiple processes, so each entry needs an identifier to uniquely identify the process requesting the TLB search.

This is called an **Address Space Identifier or ASID.** If ASID's match, then the frame number is returned, otherwise the request is considered to be a cache miss. This guarantees memory protection.

Structure of the Page Table

Most processes only use a small number of logical addresses out of the total 2^20 logical addresses.

A valid-invalid bit is used for each page table entry to indicate if there is a physical memory frame corresponding to a page number

- This bit is 1 when there is no physical frame corresponding to a page

Most of the 2^20 entries would be invalid, yet they would occupy memory space

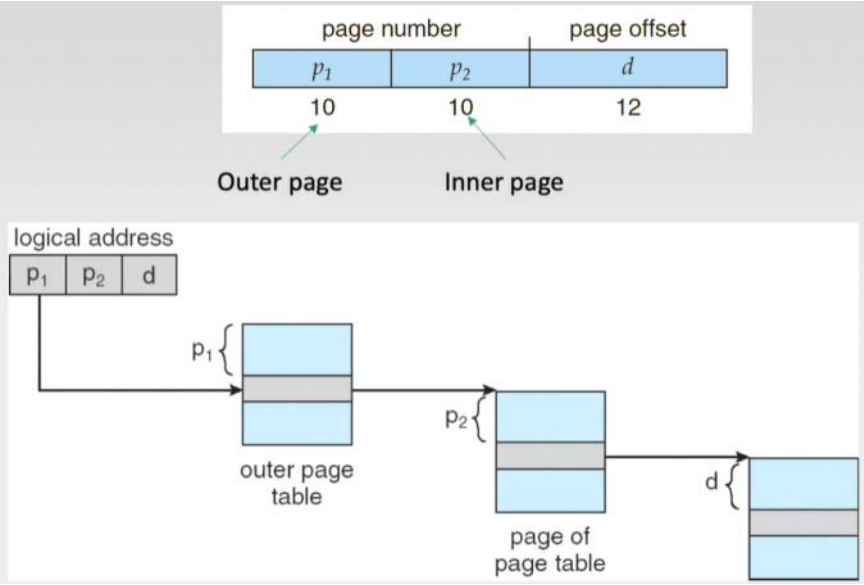
It would be better to group together invalid entries and mark them as invalid memory spaces.

Hierarchical Page Tables

In this method, we page the paging table. Pageception. We divide the page table into pages and store each page in a frame in the memory

The mapping between pages and frame numbers of the paged paging table can be stored in an outer page table. If the outer page table is still larger than one page, we can further divide it.

The advantage is, we have multiple smaller page tables. If a block of logical addresses are not used, we can set the invalid bit and ignore those bits.

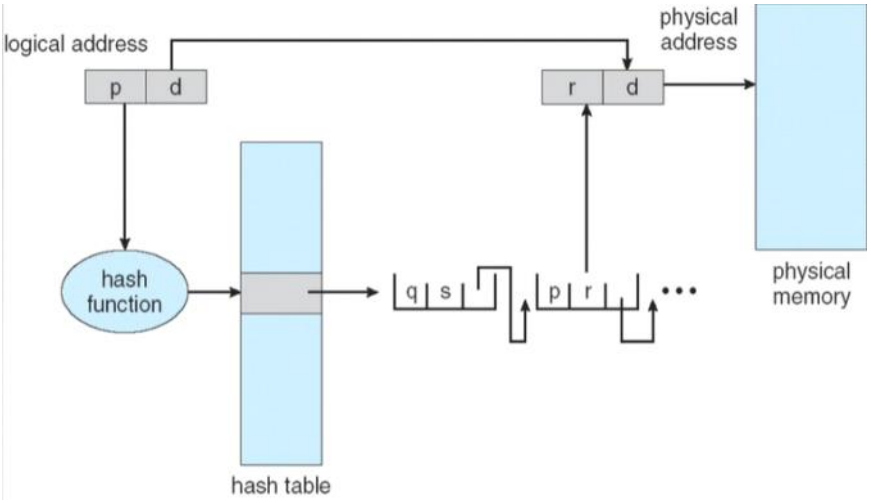


Hash Page Tables

A hashed page table has a hash value that is the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location. Each element consists of three fields – the virtual page number, the value of the mapped page frame, and a pointer to the next element in the linked list.

Algorithm

- Each virtual address is hashed into the hash table
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is match, the corresponding page frame in field 2 is used to form the desired physical address
- If there is no match, subsequent sentries in the linked l ist are searched for a ma tching virtual page number.



Inverted Page Tables

A page table has one entry for each page that the process is using, or one slot for each virtual address, regardless of the latter's validity.

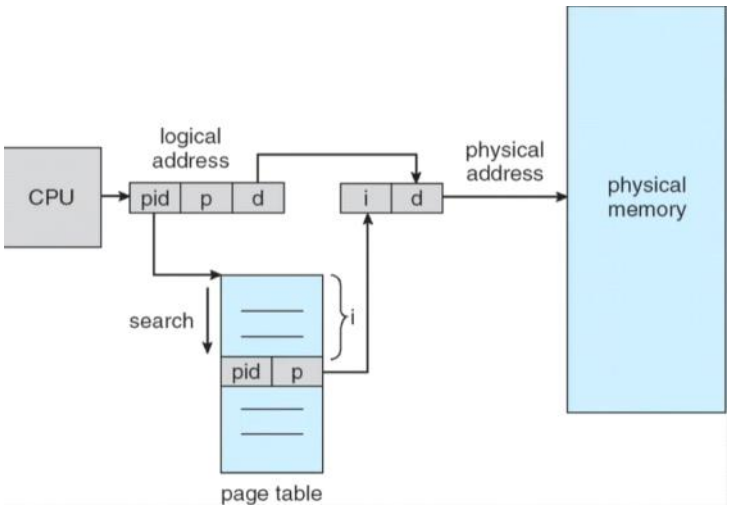
This table representation is a natural one, since processes reference pages through the pages' virtual addresses.

Since the table is sorted by virtual address, the OS is able to calculate where in the table the associated physical address entry is located and use that value.

One of the drawbacks is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

We can solve this with an inverted page table.

It has once entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus only one page table is in the system and it has only one entry for each page of physical memory.



Each virtual address consists of the following:

<**process-id**, **page-number**, **offset**>

Process-id = assumes the role of the address-space identifier

When a memory access is attempted, the process-id is searched in the inverted page table. If a match is found, then the physical address <I, offset> is generated. Otherwise, it is an illegal access.

This schemes decreases the amount of memory needed to store each page table, but increases the amount of time needed to search the table when a page reference occurs.

Because the inverted page table is sorted by physical addresses, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found. This search would take far too long.

Network Layer

03 January 2021 14:04

Objectives:

- Understand the main functions of the network layer
- IP addressing
- Routing

Main Functions

- Moves packets from a source to a destination through intermediate nodes
- Network layer runs in end hosts and routers

One of the main protocols running in the network layer is the IP

- **IP at source**: adds IP header, containing src and dest IP addresses
- **IP at routers:** checks destination Ip address of incoming packets to decide the next hop router
- **IP at destination**: receives Ip datagram, strips Ip header and delivers to transport header

There are other responsibilities of the layer – fragmenting packets and reassembling at destination

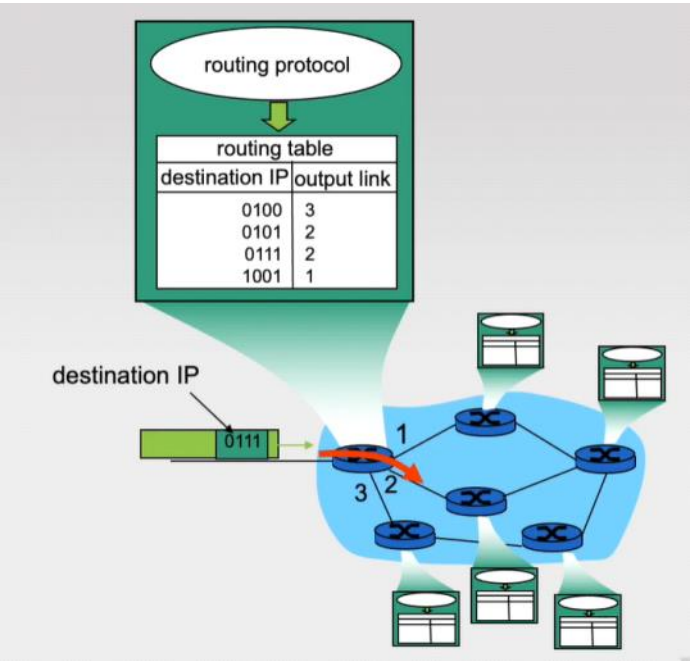
The Network Layer in the routers runs routing algorithms to compute routes

Two Key Functions of Routers

- Forwarding
- Move packets from router's input to appropriate router output

Routing

Constructing routing table – using routing protocol. These run in each routing router



Looks up destination IP in table and decides which output link to use. This table is made using routing protocols

Routing Table

- Impractical to keep a separate entry for each IP

- Increases the size of routing table and lookup time
- Many IPs map to the same outgoing link
- Better if each entry corresponds to a group of IPs and maps that to a single outgoing interface

e.g.

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111	0
11001000 00010111 00011000 00000000 through 11001000 00010111 00011000 11111111	1
11001000 00010111 00011001 00000000 through 11001000 00010111 00011111 11111111	2
otherwise	3

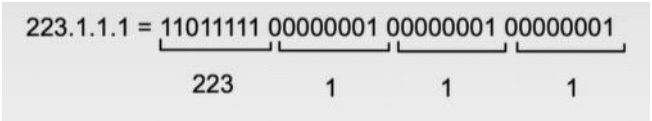
Longest Prefix Matching

When looking for forwarding table entry for given destination address, use the longest address prefix that matches destination address.

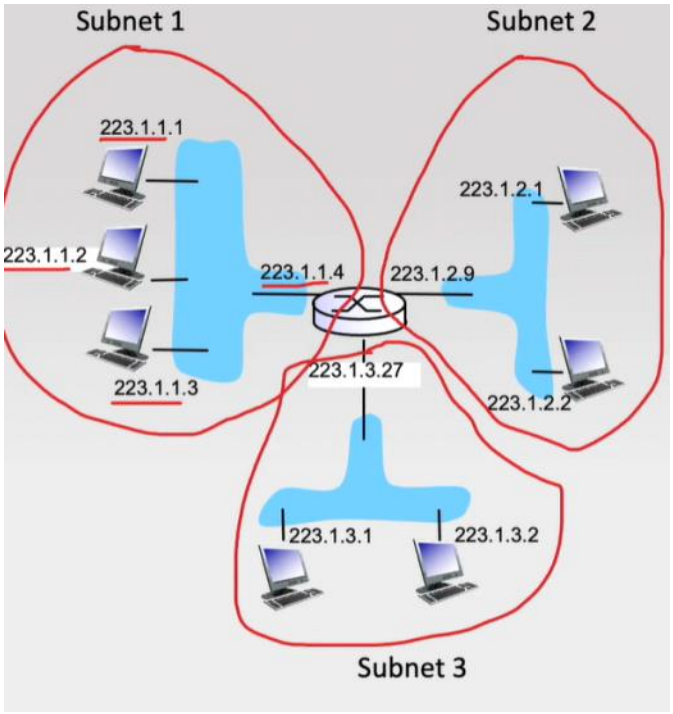
e.g. if 10101100 routes to 1 and 10101011 routes to 2, and we're looking up a destination IP 10101111 then we will choose route 1, since it matches more of the IP than the IP assigned to route 2.

IP Addressing and Subnets

IPv4 addresses are 32 bit addresses which uniquely identify network interfaces



Below we can see 3 different LAN's or Subnets. They have a unique subnet mask which is the prefix. The first 24 bits shows the subnet mask/prefix.



Each subnet mask here has a 24 bit subnet mask.

CIDR (**classless inter-domain routing**)
Notation: a.b.c.d/x means the first x bits are used as the mask
e.g. 223.1.3.27/24 means 223.1.3 is the subnet

A sender first checks if the destination IP has the same prefix as its subnet mask. If so, then

- obtain the MAC address of the dest (through ARP)
- Create a link layer frame
- Forward it to the link layer switch

What if the destination doesn't belong to the same subnet?

Forward packet to default gateway
Default gateway will look up the destination in its routing table and forward it to the right outgoing interface

How does a node get an IP address?

- Manually configured
- DHCP – Dynamic Host Configuration Protocol – application layer protocol that dynamically assigns Ip addresses from a server (usually the gateway router) to clients (nodes trying to connect to the subnet)

In either method, the subnet and default gateway must be specified

How does a subnet know which IP addresses to assign?

- The network gets allocated a portion of its provider ISP's address space
- The global authority ICANN is ultimately responsible for allocating IP addresses to ISPs

Network Address Translation

IPv4 addresses are 32 bits, and in short-supply as there are only 4 billion
ICANN gave out its last IPv4 addresses in 2011

It is not possible for ISP's to provide a unique IP to each device connected to a subnet

Instead a globally unique public IP address is provided to only gateway routers. This is globally unique. These are called public IPs.

What about other devices?

For these, we allocate private IP addresses

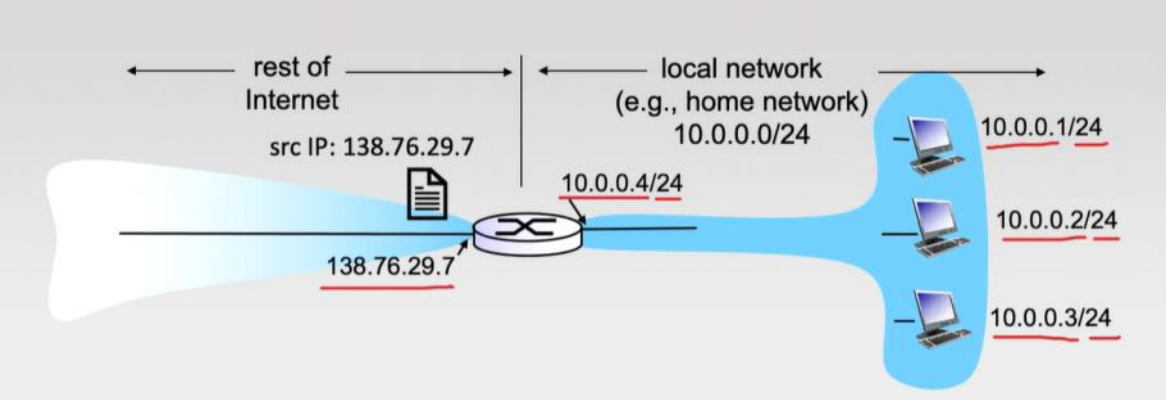
Public and Private IP

Home devices are not publicly visible – they use gateways that are publicly unique.

The following blocks are private IP addresses

- 10.0.0.0 to 10.255.255.255
- 172.16.0.0 to 172.31.255.255
- 192.16.0.0 to 192.168.255.255

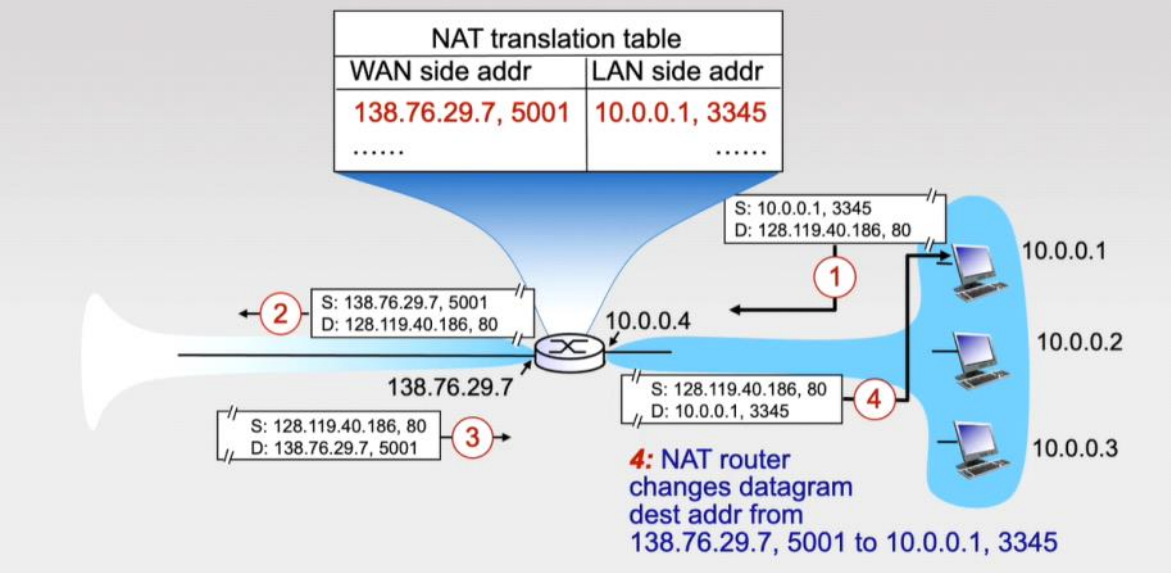
Packets with private IP's can't be sent in the public internet – they will be discarded



It is not sufficient to simply do this.

How will the replies that get sent to the default gateway be distributed to the correct private hosts on the network?

The answer is port numbers.



When the request is sent to the default gateway, the source IP is converted to the IP of the default gateway, with the appended port. This relationship is stored in the NAT, so that when a reply is received it can be looked up and sent to the correct private host

All modern devices now support IPv6

Routing

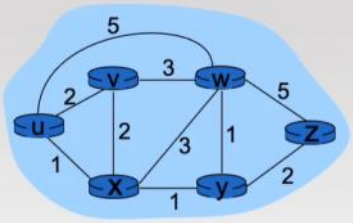
Each routing protocol implements a routing algorithm that they use to create a routing table

graph: G = (N,E)

N = set of routers = { u, v, w, x, y, z }

E = set of links ={ (u,v), (u,x), (v,x), (v,w), (x,w), (x,y), (w,y), (w,z), (y,z) }

Each edge (x, y) has a cost $c(x, y)$ associated with it,
e.g, $c(u, v) = 2$



$c(x, y) = \infty$ if x and y are not direct neighbours

cost of path $(x_1, x_2, x_3, \ldots, x_p) = c(x_1,x_2) + c(x_2,x_3) + \ldots + c(x_{p-1},x_p)$

key question: Given source x find and dest y, what is the least cost path from x to y.
At x we need to know the least cost path to every other node to populate the routing table.

1. Global Routing Algorithms – requires the knowledge of the complete topology at each router including costs – these are link state algorithms

2. Local Routing Algorithms – requires knowledge of only local neighbourhood at each router

Djikstra's Algorithm - a Link State Algorithm

Computes least cost path from one node to all other nodes
Implemented in OSPF protocol
Each node requires entire topology including the cost of each link
Obtained through broadcasting of link costs or link states

```
Initialization:
→ N' = {u}    // set of visited nodes, initially contains only the source
  for all nodes v
    if v adjacent to u:
       D(v) = c(u,v) // stores the current estimate of the shortest distance
       p(v)=u        // stores the predecessor node of v along the current shortest path from u to v
    else D(v) = ∞ , p(v)=NULL
Loop:
  find w not in N' such that D(w) is a minimum
  add w to N'                    // least cost path to w definitively known
  for all v adjacent to w and not in N' :
    if D(v) > D(w)+c(w,v)  // update distance to the unvisited neighbor v of w
       D(v) = D(w) + c(w,v)    //  if the new distance through w is smaller
       p(v)=w
until all nodes in N'
```

Distance Vector – a Local Information Algorithm

Based on Bellman Ford equation from dynamic programming

Bellman-Ford equation

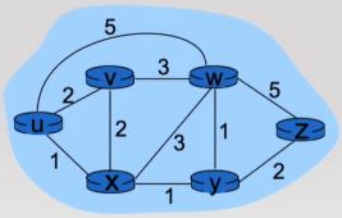
$d_x(y)$ = length of shortest path from x to y

BF equation relates $d_x(y)$ to $d_v(y)$ where $v \in N(x)$ (set of neighbours of x)

BF equation:

$$d_x(y) = \min_{v \in N(x)} \{c(x,v) + d_v(y)\}$$

If v^* minimises the above sum, then v^* is the next-hop node in the shortest path



$d_u(z) = \min \{ c(u,v) + d_v(z),$
$c(u,x) + d_x(z),$
$c(u,w) + d_w(z) \}$
$= \min \{2 + 5,$
$1 + 3,$
$5 + 3\} = 4$

Next hop node is x

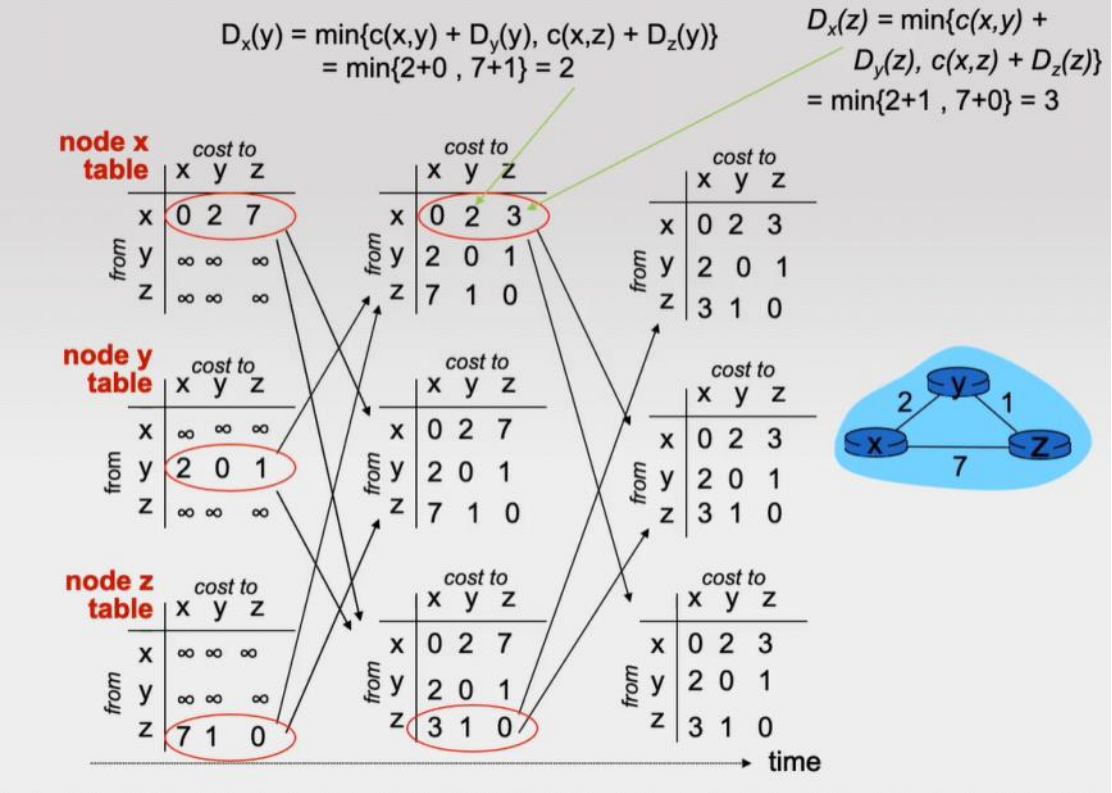
- $D_x(y)$ = current estimate of minimum distance from x to y (different from actual minimum distance $d_x(y)$)
- DV algorithm tries to converge estimates to their actual values
- Each node x maintains distance vector
  $\mathbf{D}_x = [D_x(y): y \in N]$ (vector of current estimates)
- node x performs update $D_x(y) = \min_v\{c(x,v)+D_v(y)\}$
- To perform the the update node x needs
  - cost to each neighbor v: c(x,v)
  - distance vector of each neighbour v: $\mathbf{D}_v = [D_v(y): y \in N]$ (obtained through message passing)
- Whenever any of these is updated, the node **recomputes its distance vector** and **sends the updated distance vector to all its neighbours**

each node:

wait for (change in local link cost or msg from neighbor)

recompute estimates using B-F eqn

if DV to any dest has changed, *notify* neighbors


$D_x(y) = \min\{c(x,y) + D_y(y), c(x,z) + D_z(y)\}$
$= \min\{2+0 , 7+1\} = 2$
$D_x(z) = \min\{c(x,y) +$
$D_y(z), c(x,z) + D_z(z)\}$
$= \min\{2+1 , 7+0\} = 3$
node x table
node y table
node z table
cost to
from
x y z
time
2
1
7