Jackson Dam (2619114D)

# NOSE 2 Cheat Sheet

## Part A - Network Essentials

### Topic 1 – Introduction to networked systems

#### Week 1 Lecture 1

##### Networked systems

- Networked systems are autonomous computing devices that exchange data to achieve an application goal
- The system is explicitly aware of the network and any data exchange in it
- Examples of networked systems:
  Digital TV (Freeview), VOIP (Voice over IP), controller area networks inside vehicles/aircraft, sensor networks
- Key aspects of networked systems are
  Communication – how info is exchanged over individual links
  Networking – how the links are interconnected to build a wide area network
  Networked system – how systems communicate across the network

##### Communication

- We transfer messages from source to destination with a communication channel
- Size of messages may be bounded
- Can be
  simplex (one device can transfer at a time),
  half-duplex (two-way communication, but one device can transfer at a time),
  full duplex (two-way, both devices can transfer at the same time)

##### Information

- The amount of information in a message mathematically is known as information entropy, given by:
  - $H(X) = E[-\ln(P(X))] = -\sum_{i=1}^{n} P(x_i) \log_b(P(x_i))$
    - $b = 2 \rightarrow$ *bits* or *"shannons"*
- You can model capacity of channels to convey information
- Amount, speed, power used
- Physical limits

Jackson Dam (2619114D)

## Signals

- Physical form of messages are signals
- Could be a material object (carrier pigeon, CD)
- Or a wave (sound, electrical signal, light, radio wave)
- Can be analogue (smooth continuum of values), or digital (series of discreet symbols)
- Mapping information to symbols is known as coding

## Analogue signals

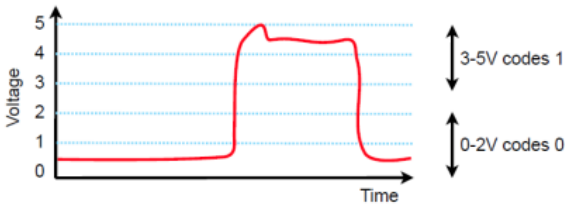
- For the simplest analogue signals, amplitude directly codes value of interest (e.g. AM radio, analogue telephone)
- Can be arbitrarily accurate (inherently accurate if signal is used directly)
- Susceptible to noise and interference on channel
- Difficult to process with digital electronics

## Sampling analogue signals digitally

- Any analogue signal can be digitally sampled
  - sample the signal at a suitable rate
  - quantise to nearest allowable discrete value (rounding to an approximate value with limited precision rather than the raw value)
  - convert this to a digital representation
- The sampling theorem determines the necessary sampling rate for a signal to be accurately reconstructed

## Digital signals

- Comprise of a sequence of discrete symbols – fixed alphabet, rather than arbitrary values
- But underlying channel is almost always analogue
- Modulation is used to map a digital signal onto the analogue channel
- e.g. NRZ modulation method (non-return to zero)

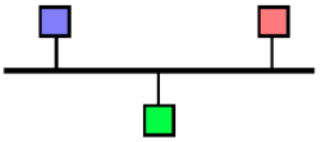


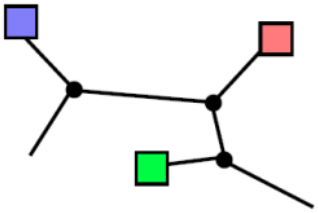

## Encoding of digital signals

- Computing systems often use binary encoding (two symbols, 0 and 1)
- Networked systems often use non-binary encoding
- Example: wireless links frequently use complex modulation schemes that allow for 16, 64, or 256 possible symbols
- Number of symbols transmitted per second is the baud rate

Jackson Dam (2619114D)

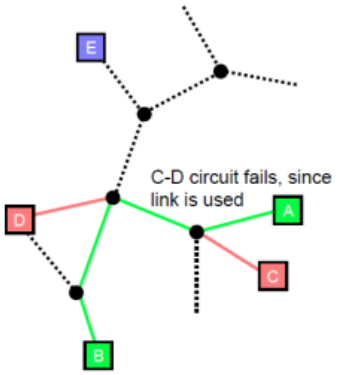
## Networks and network links

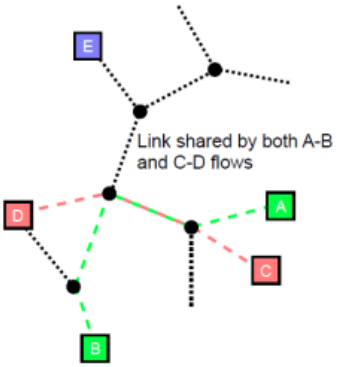




- A signal can be directly conveyed by a channel (e.g. electrical signals in a cable), or modulated onto an underlying carrier (e.g. radio)
- The signal and channel together form a link
- A link connects one or more hosts
- A network is several links connected together
- The devices connecting the links are called switches or routers depending on the network type

## Network switching




- Circuit switched networks
  - A dedicated circuit can be set up for A and B to communicate
  - A and B exchange arbitrary length messages
  - Guaranteed capacity once circuit is created
  - But the dedicated circuit can block other communications (e.g. the C to D path) – the capacity of the network gives the blocking probability
  - Example – traditional telephone network
- Packet switched networks
  - Messages are split into small packets before transmission
  - Allows A-B and C-D to communicate at the same time, sharing the bottleneck link
  - Packets are small and have a size constraint
  - A message can consist of many packets
  - Example – the internet

## Networked systems

- All networked systems are built using these basic components:
  - Hosts (act as sources and destinations)
  - Links (physical realisation of the channel, conveying messages)
  - Switches/routers (to connect multiple links)
- Layered on top of network protocols which give meaning to exchanged messages

Jackson Dam (2619114D)

Topic 2 – protocols and layers

Week 1 Lecture 2

Network protocols

- Communication occurs when two (or more) hosts exchange messages across a network
- To be meaningful, the messages need to follow a well known syntax (format), and have agreed semantics (meaning)
- A network protocol is an agreed language for encoding messages, along with the rules defining what messages mean and when they can be sent
- Numerous network protocols exist – some operate between hosts, some between routers, some between hosts and routers
- These protocols define the behaviour of the network

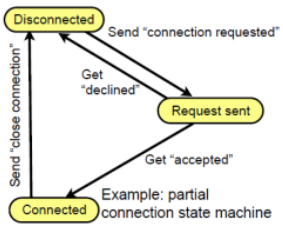
More on network protocols

- A protocol will comprise different types of message (protocol data units or PDUs)
- Each type of PDU has a particular syntax
  - describes what information is included in the PDU, how it is formatted
  - may be formatted as textual information or as binary data

  - textual PDUs have a syntax and grammar that describes their format (e.g. HTTP/1.1, SMTP, SIP, Jabber)

  - Binary PDUs have rules describing their format
    e.g. big/little endian data (most significant data stored at the end/start memory address), 32 or 64-bit, fixed/variable length, alignment requirements
  - examples: TCP/IP, RTP

- PDUs define what messages are legal to send
- Protocol semantics define when PDUs can be sent and the required response plus:
  - who can send PDUs and when
  - roles for the hosts (e.g. client and server, peer-to-peer)
  - what the entities that communicate are, how they are named
  - how errors are handled
- Commonly described with state-transition diagrams
  - states indicate stages of protocol operation
  - transitions occur in response to PDUs and may result in other PDUs being sent



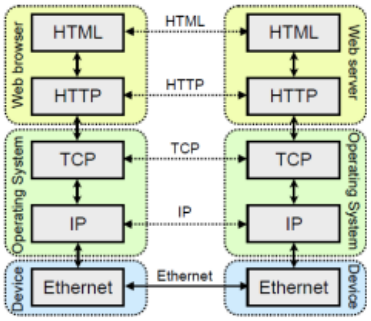
Jackson Dam (2619114D)

## Example protocol: morse code and a telegraph

- Channel = signals on electrical cable
- Syntax = pattern of dots and dashes to signal letters
- Semantics = different gap lengths to signal end of word, end of letter, use of STOP to end messages

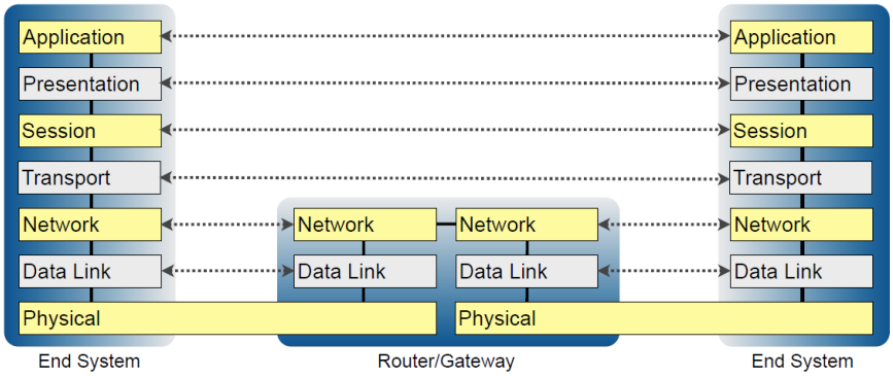
## Protocol layering

- Communications systems are typically organised as a series of protocol layers
  - structured design to reduce complexity
  - each layer offers services to the next higher layer, which it implements using the services of the lower layer (well defined interfaces)
  - the highest layer is the communicating application
  - the lowest layer is the physical channel
  - peers at some layer i, communicate via a layer i protocol, using lower layer services
- Example – web browser talking to a web server:
  - simplified view with five protocol layers, HTML, HTTP, TCP, IP, Ethernet



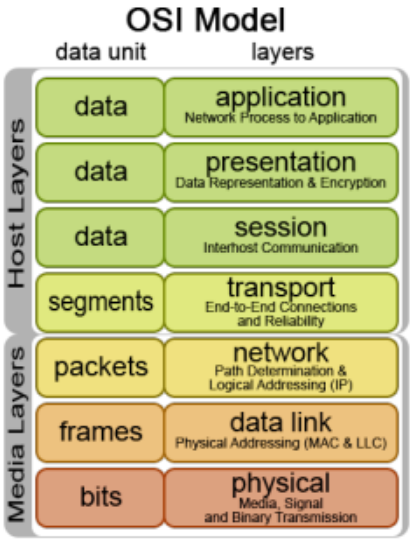

## OSI reference model

- A standard way of thinking about layered protocol design
- A design tool; real implementations are more complex



Jackson Dam (2619114D)

OSI model layers

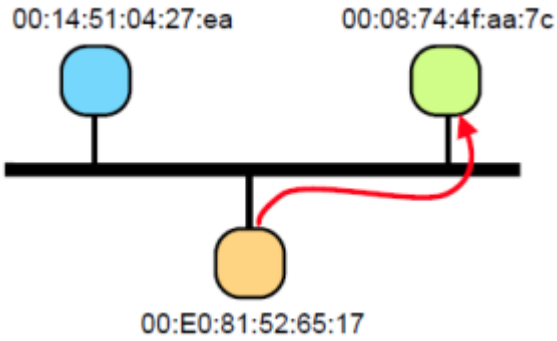



Physical layer

- Defines characteristics of the cable or optical fibre used
  - plug size/shape, cable/fibre length, type of cable, voltage, current, modulation
  - for fibres, single/multi-node, optical clarity, colour, power output, modulation of laser
  - for wireless links, radio frequency, transmission power, modulation scheme, antenna type, etc

Data link layer

- Structures and frames physical layer's bit stream
- Splits the bit stream into messages
- Detects and corrects errors in messages
- Reads parity and error correcting codes (e.g. a parity bit can describe the number of 1s being odd/even in a message, in order to detect the location of an error and automatically correct it by inverting an unexpected bit)
- Provide (negative) acknowledgements to hosts, request retransmission
- Perform media access control by:
  - assigning address to hosts on the link →
  - arbitrating access to link, and determining when hosts can send messages
  - ensuring fair access to the link and providing flow control to avoid overwhelming hosts
- Examples of data link layers – ethernet, 802.11



Jackson Dam (2619114D)

Network layer

- Interconnects multiple links to form a wide area network from source host to destination host
- Handles data delivery, naming and addressing, routing, admission/flow control
- Examples – IP, ICMP

Transport layer

- End-to-end transfer of data from source(s) to destinations(s)
- Transfers data between session level service at source to corresponding service at destination
- May provide reliability, ordering, framing, congestion control, depending on guarantees given by network layer
- Examples – TCP, UDP

Session layer

- Manages (multiple) transport layer connections
- Example session layer functions:
  - open several TCP/IP connections to download a web page using HTTP
  - use SMTP to transfer several emails over a single TCP/IP connection
  - coordinate control, audio, and video flows in a video conference
- Examples – NetBIOS, RPC

Presentation layer

- Manages the presentation, representation, and conversion of data
  - mapping data to a character set / language
  - translation of data to data markup languages (e.g. XML, HTML, JSON)
  - data format conversion (e.g. big/little endian)
  - content negotiation (means serving different versions of content to be chosen for presentation) (example protocols are MIME, SDP)
- Examples – SSL, Telnet, HTTP/HTML(agent)

Application layer

- User application protocols – e.g. WhatsApp uses XMPP, Facebook mobile app uses MQTT
- Examples: HTTP/HTTPS, DNS, FTP

Jackson Dam (2619114D)

## Protocol standards

- A formal description of a network protocol
- Ensures interoperability of diverse implementations
- Variety of standards setting procedures:
  - open or closed standards development process
  - free or restricted standards availability
  - rules around disclosure of intellectual property rights, use of encumbered (proprietary) technologies
  - individual/corporate/national registration of standard
  - completely new standard, or describing existing practices?
  - collaborative vs combative process
- Not all parties in the standards process have the same goals:
  - some may want to delay a standard to allow a proprietary solution to gain market share
  - some may want to implement intellectual property/patented technologies etc
  - some may want to enhance/subvert the security of a protocol

## Key standards organisations

- Internet Engineering Task Force
- International Telecommunications Union
- 3rd Generation Partnership Project
- World Wide Web Consortium

Jackson Dam (2619114D)

Topic 3 - Physical and data link layers

Week 2 Lecture 1

The physical layer (L1)

- Concerned with the transmission of raw data bits
- Need to consider type of cable/wireless link
- Need to consider how to encode bits onto the channel
- Need to consider channel capacity

Physical characteristics of cable or optical fibre

- Plug size/shape, maximum cable/fibre length
- Type of cable, influencing electrical voltage, current, modulation
- Type of fibre (single or multi-node, optical clarity, colour, power output, laser modulation method)

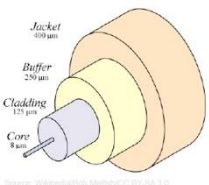
Main focus

- How to transmit a sequence of binary bits (0/1) over an analogue channel, considering noise, clock skew (the difference of arrival time of a clock signal between two different receiving components), hardware limitations etc
- We interface from the physical layer the sequence of bits to L2 (the data link layer), this hides away the complexity of encoding/decoding

Example of wired media

- Unshielded Twisted Pair (UTP)
  - Two wires twisted together (signal, and ground)
  - Each wire in the pair can send signal in one directions (unidirectional)
  - Twists reduce interference and noise pickup (more twists = less noise)
  - Can have cable lengths of several miles (at low data rates) high data rates can be maintained ~100 metres
  - Longer length = higher noise susceptibility
  - Examples – ethernet cables and telephone lines

- Optical fibre
  - Glass core and cladding, contained in a plastic protection jacket
  - Somewhat fragile, glass can crack if bent sharply
  - Unidirectional (transmission laser at one end, photodetector at the other)
  - Very low noise (electromagnetic interference doesn't affect light)
  - Very high capacity (10s of Gbps over 100s of miles)
  - Very cheap to manufacture
  - Relatively expensive lasers to operate



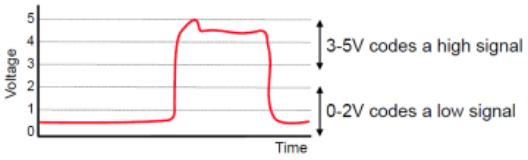
Jackson Dam (2619114D)

## Wired data transmission

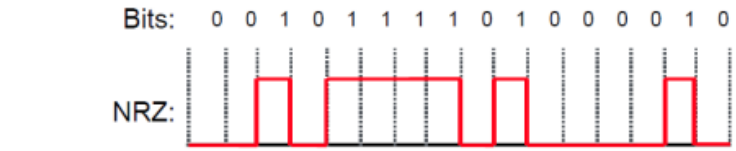
- Signal directly encoded onto channel (usually)
- Encoding is done by varying voltage on an electrical cable or the intensity of light in an optical fibre
- Many encoding schemes (NRZ, NRZI, Manchester, 4B/5B etc)

## Baseband data encoding

- Baseband data is data that has not yet been intermixed with other data and has not yet been modulated
- Encode the signal as change in voltage applied to cable or change in brightness of a laser in optical fibre

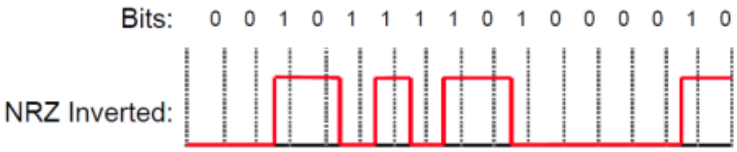



- Encoding scheme examples:



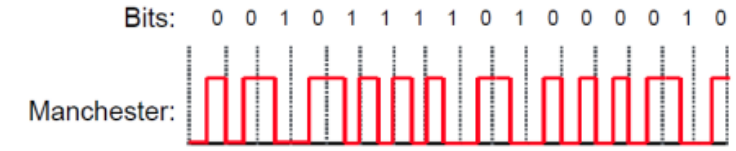

Encodes a 1 as a high signal, a 0 as a low signal

Runs of the same value → clock skew and baseline wander




Encode a 1 as a change in signal value, a 0 as a constant signal

Solves problem with consecutive 1s, not runs of consecutive 0s

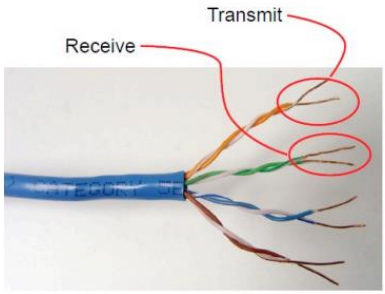



Encode a 1 as high-low transition, a 0 as a low-high transition

Doubles the bandwidth needed, but avoids problems with NRZ encoding

## Ethernet example

- Baseband data with Manchester encoding at 10Mbps, or 4B/5B encoding at 100Mbps

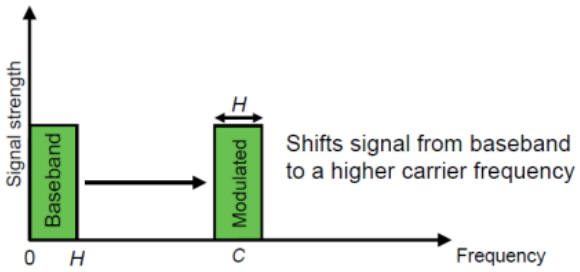



4 twisted pairs per cable
3 twists per inch
24 gauge (~0.5mm) copper
100m maximum cable length

Jackson Dam (2619114D)

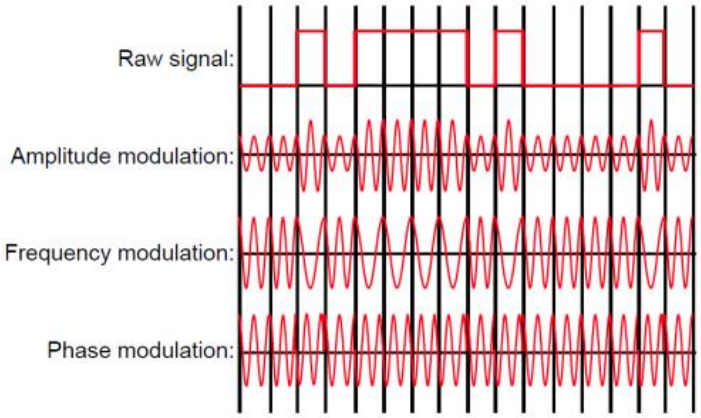
## Carrier modulation

- Carrier wave applied to channel at frequency, C
- The signal is modulated onto the carrier




- This allows you to pack multiple signals onto the same channel, as long as the carriers are spaced greater than bandwidth H of a single modulated signal
- Usually applied to wireless links but can be used on wired links (this is how ADSL and voice telephones share a single line)

## More on carrier modulation




- More complex modulation schemes allow more than one bit to be sent per baud (baud rate is symbols per second)
  - use multiple levels of the modulated component to represent more values than binary signalling (i.e. more voltage levels = more values)
  - example: gigabit Ethernet uses amplitude modulation with 5 levels instead of binary signalling
- You can combine modulation schemes
  - vary both phase and amplitude = quadrature amplitude modulation
  - example: 9600 bps modems use 12 phase shift values at two different amplitudes
- Extremely complex modulation schemes are regularly used

Jackson Dam (2619114D)

Spread spectrum communication

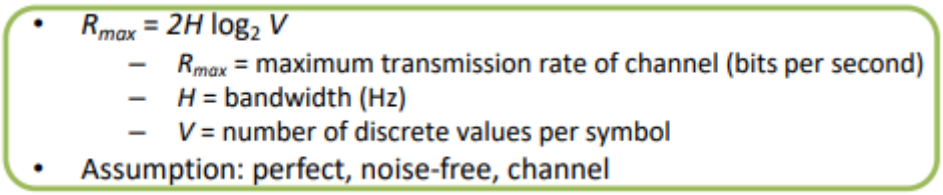
- Single frequency channels are prone to interference
  - can be mitigated by repeatedly changing carrier frequency (noise is unlikely to affect all the frequencies)
  - use a pseudo-random sequence to choose which carrier frequency is used for each time-slot
  - the seed of the pseudo-random number generator is shared in secret between the sender and the receiver to ensure security
  - example: 802.11b Wi-Fi uses spread spectrum among several frequencies centred around 2.4GHz with phase modulation

Bandwidth

- Bandwidth of a channel determines frequency range that it can transport
- Bandwidth fundamentally limited by physical properties of channel, design of end points, etc

Digital signals

- Nyquist's sampling theorem states that:
  to accurately digitise an analogue signal, you need a sample rate of at least 2H samples a second, where H is the bandwidth in Hz
- Maximum transmission rate of a digital signal depends on the channel bandwidth




- Real world channels are subject to noise that corrupts the signal, e.g. electrical interference, cosmic radiation, thermal noise
- You can measure a channel's signal power S and noise floor N, and calculate its signal-to-noise ratio (S/N or SNR):
  - typically quoted in dB, as $10log_{10}S/N$ decibels
  - example: ADSL modems report an S/N of ~30 dB for good quality phone lines (signal power ~1000x greater than noise)
- Maximum transmission rate of a channel grows logarithmically with SNR:

$$R_{max} = Hlog_2(1 + S/N)$$

- Bandwidth and SNR are considered fundamental limits, that might be reached with careful engineering but cannot be exceeded

Jackson Dam (2619114D)

Week 2 Lecture 2

# The data link layer (L2)

## Purpose of data link layer

- Arbitrate access to physical layer
  - identifying devices with addressing
  - structure and frame the raw bit stream
  - detect and correct bit errors
  - control access to the channel (media access control)
- Interface with L1 (physical layer) is the raw bit stream
- Interface with L3 (network layer) is structured communication (addressing, packets)

## Basic services

- Addressing
  - physical links can be point-to-point or multi-access
  - wireless links are a common multi-access link, but several hosts to a single cable for a multi-access wired link
  - multi-access links require host addresses to identify senders/receivers
- Host addresses can be link-local or global scope
  - sufficient to be unique only amongst devices connected to a link
  - but need coordination between devices to assign addresses
  - many data-link-layer protocols (Ethernet, IEEE 802.11 Wi-Fi) use globally unique addresses
  - simpler to implement if devices can move, since there's no need to change address when connected to a different link (privacy)
- Framing and synchronisation
  - physical layer raw bit stream is unreliable, could have corrupted bits or disrupted timing
  - data link layer can fix these problems by breaking the raw bit stream into frames, transmitting, and repairing individual frames, and limiting scope of any transmission errors

## Error detection and correction

- Noise and interference (more common in wireless links) can cause bit errors
- Add error detecting code to each packet
- Example: parity codes
  - add a parity bit onto the end of a packet before sending (0 for even number of 1s in data, 1 for an odd number)
  - recalculate the parity bit on the received data, if the parity bit doesn't match, an error is detected, and the packet can be rejected

- Example: internet checksum
  - add all 16-bit values of a packet (except the checksum field) and take the one's complement of the sum to form the checksum
  - send the packet with the checksum field in it
  - recipient can verify checksum by adding all data 16-bit values (including the checksum) and taking one's complement of this sum, if no errors are detected the sum will be 0000
- More powerful error detecting codes exist, such as CRC (cyclic redundancy code)
- Error detecting codes can be extended to correct errors (by sending additional data in each frame)
  - this can allow correction of some errors without recontacting sender
  - e.g. Hamming Code allows recipient correction of all single-bit and some multi-bit errors
- More complex = fewer undetected errors, but there is a trade-off between amount of added data and ability to correct multi-bit errors
- Can also request retransmission upon error detection as a means of repair

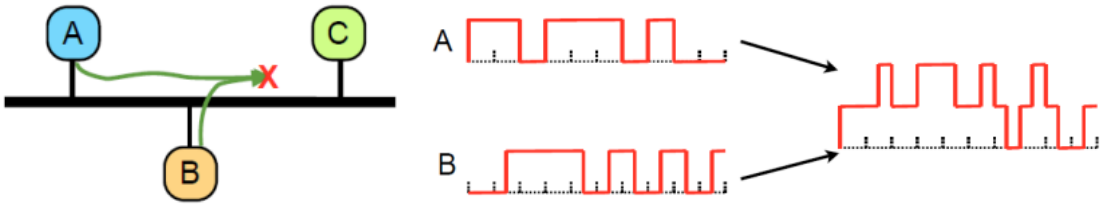
Synchronisation

- Hard to detect start of messages in a continuous raw bit stream
- Could leave time gaps between frames (but physical layer doesn't guarantee timing due to clock skew etc)
- Could have a length field preceding each frame (but the length may be corrupted, hard to find next frame)
- Could add a special start code to beginning of each frame (preamble allocated block)
  - a unique bit pattern that only occurs at the start of each frame
  - enables synchronisation after error (wait for next start code and begin reading frame headers)
- If the start code is in the actual data:
  - use bit stuffing to give a transparent channel
  - insert a 0 bit whenever sending any five consecutive 1 bits (unless sending start code)
  - if receiver finds five consecutive 1 bits, look at sixth bit, and remove if 0
  - otherwise if 1, look at seventh bit, if it is 0, it is the start code
  - if the seventh bit is 1, the frame is corrupt

Jackson Dam (2619114D)

## Media access control

- Enables arbitrating access to the link
- Links can be point-to-point or multi-access
  - point-to-point links are typically two unidirectional links with separate cables for each direction (still requires framing, but no contention)
  - multi-access links typically share a bidirectional link (single physical cable or single radio frequency, nodes have to contend for link access)
- Link contention is when two hosts transmit simultaneously and cause a collision, and as signals overlap, only garbage is received




## Contention-based MAC

- If multiple hosts share a channel where collisions can occur, system is contention-based
- Two-stage access to channel:
  - detect when collision is occurring / will occur (by listening to channel before/while sending)
- Send if no collision, otherwise use an algorithm to back off and retransmit data to avoid/resolve collision
  - retransmit with a random back-off delay that is randomised and increasing to avoid repeated collisions
  - can be arranged to give priority to specific hosts/users/types of traffic
- Access to the channel is probabilistic and variable latency

## More on contention-based MAC

- ALOHA network
  - transmit whenever data is available
  - if a collision occurs, wait a random time, and try again, then repeat until successful
  - simple but poor performance
  - low channel utilisation and long delays
- Carrier Sense Multiple Access (CSMA)
  - when propagation delay (time to get a signal from sender to recipient) is low, listen for an active transmission
  - if any other transmissions are active, back-off as if a collision occurred
  - if link is idle, send data immediately
  - improves utilisation as active transmissions are never interrupted by collisions
  - only new senders back-off if channel is active

Jackson Dam (2619114D)

More on collisions

- High propagation delay → increased collision rate, this is not addressed by plain CSMA
- Updated CSMA with collision detection (CSMA/CD)
  - listen to channel before and while transmitting
  - if a collision occurs, stop sending, back-off, and retransmit
  - collisions still corrupt both packets but channel blockage time due to collisions is reduced, resulting in better performance than plain CSMA
  - is embedded in Ethernet and 802.11 Wi-Fi
- Back-off interval durations should be random to avoid repeat collisions
- Should increase with the number of collisions
  - repeated collisions cause signal congestion, so reduce the transmission rate to allow network to recover
- Good strategy: an initial back-off interval of x seconds +/- 50% (random)
  - for each repeated collision before success, x = 2x

Jackson Dam (2619114D)

Topic 4 – Network Layer

Week 3 Lecture 1

The Network Layer (L3)

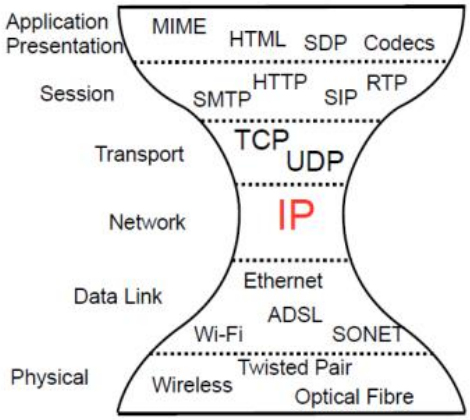
- First end-to-end layer in the OSI model
  - responsible for end-to-end delivery of data, across multiple link-layer hops and technologies
  - across multiple autonomous systems
- An internet comprises a set of interconnected networks
- Each network is administered separately (autonomous system with independent policy and technology choices)

Components of an internet

- A common end-to-end network protocol
  - to provide a single seamless service to the transport layer, manage delivery of data packets/provisioning of circuits and addressing of end systems
- A set of gateway devices (a.k.a. routers), that:
  - implement the common network protocol
  - hide differences in data link layer technologies by translating away: framing, addressing, flow control, error detection and correction
  - desire to perform the least translation necessary

The Internet Protocol

- Provides an abstraction layer that is below the transport layer/protocols and applications, but above the data link layer and physical layer
- It is a simple, best-effort, connectionless packet delivery service
- Handles addressing, routing, fragmentation, and reassembly
- Hour-glass protocol stack shows IP facilitating a uniform end-to-end connectivity across different transport, application layer, and link-layer technologies (subject to firewall policy)



Jackson Dam (2619114D)

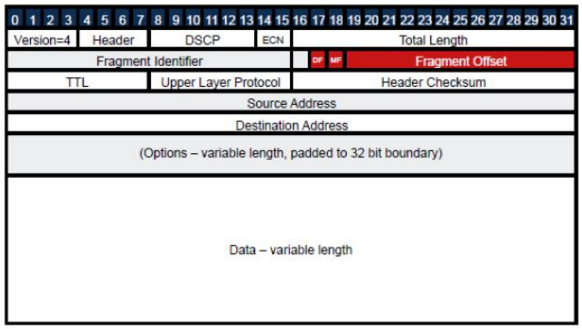
## More on the internet protocol

- IP service model: best-effort connectionless packet delivery
  - just send without setting up a connection beforehand
  - network makes best effort to deliver packets, but with no guarantees
  - time taken to transit the network can vary
  - packets may be lost, delayed, modified, reordered, duplicated, or corrupted
  - network discards undeliverable packets
  - easy to run over any type of link layer
- IPv4 is used in current generation internet
- IPv6 is next-gen (simpler header format, larger addresses, no fragmentation support, adds flow label)

## Addressing

- Unique address for every network interface on every host
- Host can have a dynamic address (for illusion of privacy)
- Addressable != reachable (firewalls in both IPv4 and IPv6)
- IPv4 addresses are 32-bit:
  123.456.789.987 (four 8-bit fields with a decimal value of 0-255)
  - we are running out of IPv4 addresses (significant problems)
- IPv6 addresses are 128-bit:
  2001:0DB8:AC10::FE01 (between the double colons are omittable zeros)
  - four 16-bit hexadecimal fields (as one hex digit is 4 bits)

## Fragmentation



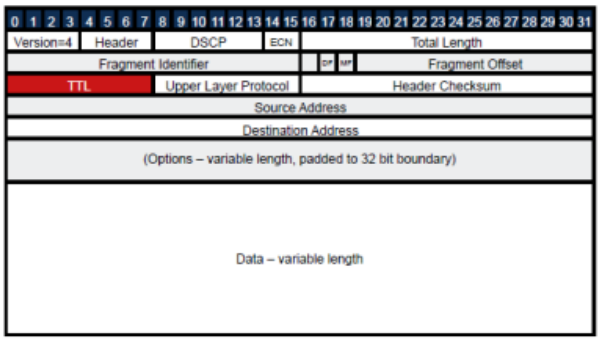

- IPv4 packets have DF (don't fragment) bit, MF (more fragment) bit and fragment offset
- Example: if sending 100 bytes of data with a maximum packet size (MTU) of 40 bytes:
  - Send first 40 byte packet with fragment offset 0 (no preceding data yet), and MF = 1, DF = 0
  - Send second 40 byte packet with fragment offset 40 (40 bytes preceding data), and MF = 1, DF = 0
  - Send third 20 byte packet with fragment offset 80 (80 bytes preceding data), and MF = 0, DF = 1 (no further fragmentation)

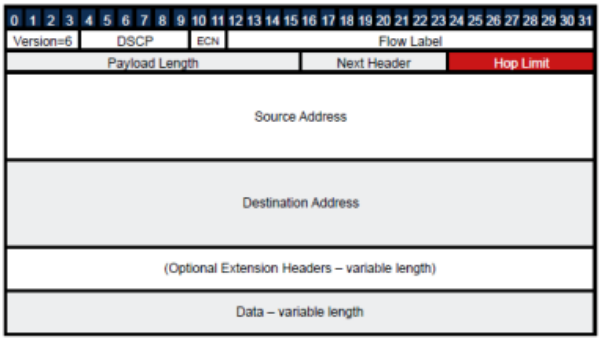
Jackson Dam (2619114D)

## More on fragmentation

- IPv4 routers reconstruct packets using the fragment offset
- If DF bit is set, router will discard large packets rather than fragmenting
- IPv6 doesn't support fragmentation
    - hard to implement for high-rate links, so this is left for the data-link layer

## Loop protection




- Packets have a forwarding limit, set to a non-zero value (64 or 128) when the packet is sent
- Each router that forwards the packet reduces the value by 1
- The packet is discarded if 0 is reached
- This stops the packet circling forever due to a loop caused by a network problem
- Assumption – network diameter is smaller than initial value of forwarding limit

## Header checksum

- IPv4 header contains a checksum to detect transmission errors
- Protects the IP header only, not the data
- Payload data must be protected by upper-layer protocols (transport layer) instead
- IPv6 has no checksum, and assumes data is protected by link layer checksum

## Transport Layer Protocol Identifier

- Identifies transport layer protocol used to pass the data to the correct upper-layer protocol
- Stored as "upper layer protocol" in IPv4, "next header" in IPv6
- TCP = 6, UDP = 17, DCCP = 33, ICMP = 1
- Protocols managed by IANA

Jackson Dam (2619114D)

IPv4 or IPv6?

- We've ran out of IPv4 addresses
- IPv6 is intended as a long term replacement
- Primary goal is to increase address space size to allow more hosts on the network
- Secondary goal is to simplify the protocol, making high-speed implementations easier
- Not yet clear if IPv6 will be widely deployed
    - Easy to build applications that support both IPv4 and IPv6
    - DNS query will return IPv6 address if it exists, else IPv4 address; all other communications calls use their returned values
    - New code needs to be written to support both IPv4 and IPv6

Identity and location

- Addresses can denote identity
    - give hosts a consistent address regardless of where or when they connect
    - simple upper layer protocols (no need to account in transport layer or applications for multi-homing or mobility)
    - adds complexity to the network layer (must determine location of host prior to data routing, and often requires an in-network database to map host identity to routable address)
- Alternatively, an address can indicate location at which a host attaches to the network
    - address structure matches the network structure (network can directly route data given an address)
    - simplifies network layer by pushing complexity to higher layers (multi-homing and mobility must be handled by transport layer or applications, since transport layer connections break when host moves)

More on IP addresses

- IPv4 and IPv6 addresses encode location
- Addresses are split into a network and a host part
- A netmask describes number of bits in the network part
- The network itself has the address with the host part equal to zero
- The broadcast address for a network has all bits of the host part equal to one
    - allows messages to be sent to all hosts on the network
- A host with several network interfaces will have one IP address per interface (e.g. Ethernet and Wi-Fi on a laptop = 2 IP addresses)

    -

Jackson Dam (2619114D)

IP address management

- IPv4 has 2^32 addresses
    - IANA administers the pool of unallocated addresses
    - Historically ISPs and large enterprises etc were directly assigned addresses
    - Now, addresses are assigned to RIRs (regional internet registries) as needed (assigned in "/8 blocks", containing 2^24 addresses each)
    - RIRs assign to ISPs and large enterprises within their region
    - ISPs allocate to customers
- IANA has allocated all addresses to RIRs already (last allocation on 3/2/11)
- IPv6 can give 2^128 addresses, solving the problem for a long time (665,570,793,348,866,943,898,599 addresses per m^2 of the Earth's surface)

IPv6 deployment issues

- Need to make changes to every host, router, firewall, application
- Changes done to macOS, Windows, Linux, FreeBSD, iOS, Android, etc
- Backbone routers generally support IPv6, home routers and firewalls starting t be updated
- Many applications updated already

Jackson Dam (2619114D)

Topic 5 – Routing

Week 3 Lecture 2

## Routing

- The network layer is responsible for routing data from source to destination via multiple hops
- Nodes (network devices, routers, hosts) need to learn a subset of the network topology and run a routing algorithm to decide where to forward packets destined for other hosts
- End hosts usually have a simple view of the topology (local network and everything else), and a simple routing algorithm (send to default gateway if it's not on local network)
- Gateways (routers) exchange topology information, decide best route to destination based on knowledge of entire network topology

## Unicast routing

- Where routing algorithms deliver packets from a single source to a single destination
- Choice of algorithm affected by usage scenario
    - intra-domain routing
    - inter-domain routing
    - politics (country doesn't want specific packets) or economics (cheaper to route a certain way)
- Inter-domain unicast routing (routing between domains)
    - each network administered separately (autonomous system – AS)
    - different technologies and policies in each AS, mutual distrust between AS and its peers
- Intra-domain unicast routing (routing within domain/autonomous system)
    - enables routing between members of an AS
    - single trust domain, so no policy restrictions on who can determine network topology or which links can be used
    - desire for shortest path for most efficient routing, making best use of available network
    - two approaches – distance vector (Routing Information Protocol (RIP)), and link state (Open Shortest Path First Routing (OSPF))

Jackson Dam (2619114D)

Inter-domain routing

- Find best route to destination network, treating each network as a single node, and route without reference to internal network topology
- Each network comprises multiple ASes (autonomous systems)
- Routing problem is finding best AS-level path from source AS to destination AS
- Treat each AS as a node on the routing graph ("AS topology graph")
- Treat connections between ASes as edges in the graph
- AS level topology:
    - well connected core networks
    - sparsely connected edges, getting service from the core networks
- Edge networks can use a default route to the core
- Core networks need full routing table
    - can be accessed from the default free zone (DFZ) (Regional Internet Registries (RIRs) and Internet Exchange Points (IXPs))

Routing at the DFZ

- Core networks are well-connected, must know about every other network
    - the default free zone is where there is no default route
    - route is based on policy, not necessarily shortest path
    - e.g. use AS x in preference to AS y, use AS x only to reach addresses in this range, use the path that crosses the fewest number of ASes, or avoid ASes located in a specific country
- Requires complete AS-level topology information

Intra-domain routing

- Operates within a network
    - any network operated as single entity is an autonomous system
    - operates a single routing protocol (typically OSPF link-state protocol is used for exchanging routes to IP address prefixes)
    - runs on IP routers within an AS (typically with fibre connections for wide area and ethernet connections for local area)
    - exchanges routes to IP prefixes, representing regions in network topology

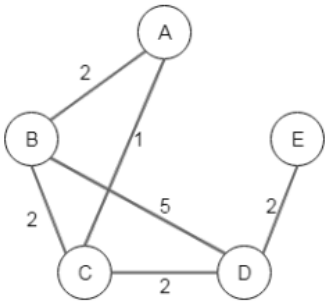
Jackson Dam (2619114D)

## Distance vector protocols

- Each node maintains a vector containing the distance to every other node in the network
  - periodically exchanged with neighbours so eventually each node knows the distance to all other nodes
  - routing table "converges" on a steady states
  - links which are down or unknown have distance = ∞
  - forward packets along the route with the least distance to destination

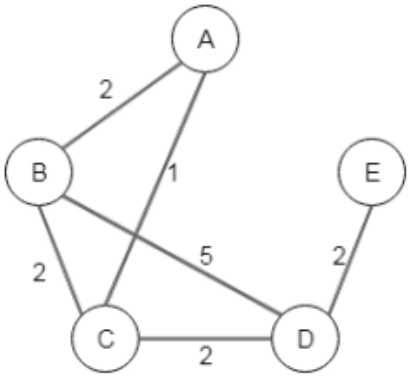
## Distance vector protocols example

- Information stored at every node, to allow routing to other nodes:
  - cost (length of path to X)
  - next hop (next node on the path to X -- dash (-) signifies unknown next hop, dot (•) signifies no hop required)
  - tables at every node encoded as a row in the table below
- This example uses names (A, B, C, ...) to keep the diagram readable
  - real implementations identify nodes by their IP address, or by IP prefixes if routing to networks
- Initially table is empty – know of no other nodes




| Routing table at node | Cost / Next hop to node | | | | |
|---|---|---|---|---|---|
| | A | B | C | D | E |
| A | 0/• | ∞/- | ∞/- | ∞/- | ∞/- |
| B | ∞/- | 0/• | ∞/- | ∞/- | ∞/- |
| C | ∞/- | ∞/- | 0/• | ∞/- | ∞/- |
| D | ∞/- | ∞/- | ∞/- | 0/• | ∞/- |
| E | ∞/- | ∞/- | ∞/- | ∞/- | 0/• |

Jackson Dam (2619114D)




- Time = 0
  - nodes initialised, only know about their immediate neighbours
  - example: routing table at A has:
  - (cost 0 to A, no next hop), (cost 2 to B, B is the next hop), (cost 1 to C, and C next hop), no known paths to D or E, hence infinite cost and unknown next hop

| Routing table at node | Cost / Next hop to node | | | | |
|---|---|---|---|---|---|
| | A | B | C | D | E |
| A | 0/• | **2/B** | **1/C** | ∞/- | ∞/- |
| B | **2/A** | 0/• | **2/C** | **5/D** | ∞/- |
| C | **1/A** | **2/B** | 0/• | **2/D** | ∞/- |
| D | ∞/- | **5/B** | **2/C** | 0/• | **2/E** |
| E | ∞/- | ∞/- | ∞/- | **2/D** | 0/• |

- Time = 1
  - routing data spreads one more hop (nodes know neighbours of neighbours)
  - A is receiving routing state from B and C:

| | | | | | |
|---|---|---|---|---|---|
| B: | 2 | 0 | 2 | 5 | ∞ |
| C: | 1 | 2 | 0 | 2 | ∞ |

  - A then sees that it can reach D via both B (cost: 2 + 5) and C (1 + 2); C is thus selected as the next hop to D
  - B sees that it can reach D via C with a lower cost (2 + 2) than the previous one (5); also E via D (through C)

| Routing table at node | Cost / Next hop to node | | | | |
|---|---|---|---|---|---|
| | A | B | C | D | E |
| A | 0/• | 2/B | 1/C | **3/C** | ∞/- |
| B | 2/A | 0/• | 2/C | **4/C** | **6/C** |
| C | 1/A | 2/B | 0/• | 2/D | **4/D** |
| D | **3/C** | **4/C** | 2/C | 0/• | 2/E |
| E | ∞/- | **7/D** | **4/D** | 2/D | 0/• |

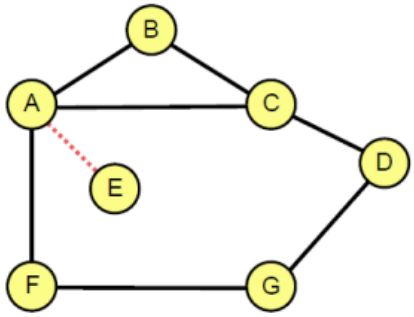
- Time = 2
  - routing data has spread 2 hops (routing table complete)
- Time = 3 onwards
  - nodes continue to exchange distance metrics in case topology changes

| Routing table at node | Cost / Next hop to node | | | | |
|---|---|---|---|---|---|
| | A | B | C | D | E |
| A | 0/• | 2/B | 1/C | 3/C | **5/C** |
| B | 2/A | 0/• | 2/C | 4/C | 6/C |
| C | 1/A | 2/B | 0/• | 2/D | 4/D |
| D | 3/C | 4/C | 2/C | 0/• | 2/E |
| E | **5/D** | **6/D** | 4/D | 2/D | 0/• |

See slides 20-22 of Topic 5 for failed link example (too lengthy)

Jackson Dam (2619114D)

Count to infinity problem example




- What if A-E link fails?
  - A advertises distance ∞ to E at the same time as C advertises a distance 2 to E (the old route via A)
  - B receives both, concludes that E can be reached in 3 hops via C, and advertises this to A
  - C sets its distance to E to ∞ and advertises this
  - A receives the advertisement from B, decides it can reach E in 4 hops via B, and advertises this to C
  - C receives the advertisement from A, decides it can reach E in 5 hops via A...
  - Loops, eventually counting up to infinity...

Distance vector protocols – limitations

- Count to infinity problem
- Solution 1 – how big is infinity?
  - define infinity as 16, to bound time it takes to count to infinity, and hence duration of the disruption
  - network must never be more than 16 hops across in this case
- Solution 2 – split horizon
  - when sending a routing update, don't send route learned from a neighbour back to that neighbour
  - prevents loops involving two nodes, doesn't prevent three node loops
- No general solution exists
  - distance vector routing always suffers from slow convergence due to the count to infinity problem
  - implies distance vector routing is only suitable for small networks
- Distance vector routing tries to minimise state at nodes, resulting in slow convergence
- Link state routing is an alternative

Jackson Dam (2619114D)

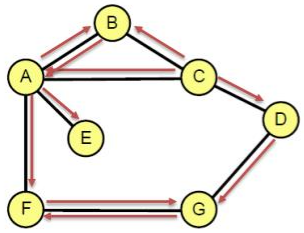
## Link state routing

- All nodes know links to their neighbours + the cost of using these links (the link state information)
- Reliably flood this information, giving all nodes a complete map of the network
- Each node then directly calculates shortest path to every other node locally, and uses this as a routing table
- Link state information is flooded on start-up, and when topology changes
- Each update contains:
  - the address of the node that sent the update
  - list of directly connected neighbours of that node (with the cost of link to each neighbour and the range of addresses assigned to each link, if it connects more than one host)
  - a sequence number

## Link state routing – forwarding and route updates

- Forward packets based on calculated shortest path
  - static forwarding decision based on weights distributed by the routing protocol
  - does not account for network congestion
- Recalculate shortest paths on every routing update
- Updates occur when a link fails, or a new link is added

## Link state routing example




- Node C sends an update to each of its neighbours
- Each receiver compares the sequence number with that of the last update from C
  - If greater it forwards the update on all links except the link on which it was received
- Each receiver compares the sequence number with that of the last update from C
  - If greater it forwards the update on all links except the link on which it was received
- Eventually, the entire network has received the update

## Distance vector routing vs. link state routing

### Distance Vector Routing

- Simple to implement
- Routers only store the distance to each other node
  - O(n)
- Suffers from slow convergence
- Example: **Routing Information Protocol (RIP)**

### Link State Routing

- More complex
- Requires each router to store complete network map
  - $O(n^2)$
- Much faster convergence
- Example: **Open Shortest Path First (OSPF)**

Jackson Dam (2619114D)

# Topic 6 – Transport Layer

## Week 4 Lecture 1

### Transport Layer (L4)

- Role is to isolate upper layers from network layer
- Hide network complexity
- Make unreliable network appear reliable
- Enhance QoS (quality of service) of network layer
- Provide a useful, convenient, easy to use service
- An easy to understand service model
- An easy to use API (e.g. the Berkeley sockets API, widely used compared to network layer API that is usually hidden in OS internals)
- Functions:
    - Addressing
    - Multiplexing
    - Reliability
    - Framing
    - Congestion control
- Operates process-to-process, not host-to-host

### More on the transport layer

- Addressing and multiplexing (putting data from multiple processes into one unified data flow, demultiplexing is separating it back to the correct processes)
    - the network layer identifies a host
    - the transport layer identifies a user process – a service – running on a host
- Reliability:
    - network layer is unreliable
    - best effort packet switching on the Internet, but even nominally reliable circuits may fail
- Transport layer enhances the quality of service provided by the network, to match application needs with appropriate end-to-end reliability
- Different applications need different reliability:
    - email and file transfer → all data must arrive in order but no strict timeliness requirement
    - voice or streaming video → can tolerate a small amount of data loss, but requires timely delivery

Jackson Dam (2619114D)

## Even more on the transport layer

- Framing
  - applications may wish to send structured data
  - transport layer responsible for maintaining the boundaries
  - transport layer must frame the original data if this is part of the service model
- Congestion and flow control
  - transport layer controls the application sending rate, to match rate of network layer data delivery (congestion control)
  - to match rate of receiver data processing (flow control)
  - must be performed end-to-end since only endpoints know characteristics of entire path
  - different applications have different needs for congestion control
  - email and file transfer → elastic applications (faster is better, but actual sending rate not vital)
  - voice and streaming video → inelastic applications (have minimum and maximum sending rates, care about the actual sending rate)

## The end-to-end principle

- Have as little functionality as necessary on the network
- Reserve everything else for end systems / endpoints
- e.g. put reliability in transport layer rather than the network
- If the network is not guaranteed 100% reliable, the application will have to check the data anyway
- Don't check in the network, leave to the end-to-end transport protocol, where the check is visible to the application
- Defining principle of the Internet

## Internet transfer protocols

- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
- Datagram Congestion Control Protocol (DCCP)
- Stream Control Transmission Protocol (SCTP)
- QUIC (pronounced "quick")
- Each makes different design choices

## Internet Transfer Protocols: UDP

- Simplest transfer protocol
- Exposes raw IP service to applications
  - connectionless, best effort packet delivery
  - framed but unreliable
  - no congestion control, 16-bit port number as service identifier

Jackson Dam (2619114D)

## Internet Transfer Protocols: TCP

- Reliable byte stream protocol over IP
- Adds reliability
- Adds congestion control
- Not framed, only delivers an ordered byte stream (application must impose structure)
- 16-bit port number used as service identifier (0 – 65535)

## Berkeley Sockets API

- Widely used low-level C networking API
    - first introduced in 4.3BSD Unix
    - now available on most platforms (Linux, Windows, macOS, BSD, etc)
    - largely compatible cross-platform

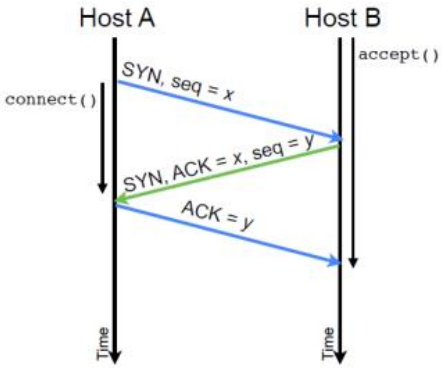
- Sockets provide a standard interface between network and application
- Two types of socket:
    - stream (provides a virtual circuit service)
    - datagram (delivers individual packets)
- Independent of network type
    - commonly used with IP sockets but not specific to the Internet protocols
    - stream sockets map onto TCP/IP connections
    - datagram sockets map onto UDP/IP

## More on TCP

- Reliability is added when initiating/terminating a connection and during communication through:
    - sequence numbers
    - acknowledgements
    - multi-round processes

Jackson Dam (2619114D)

TCP connection setup: the 3-way handshake

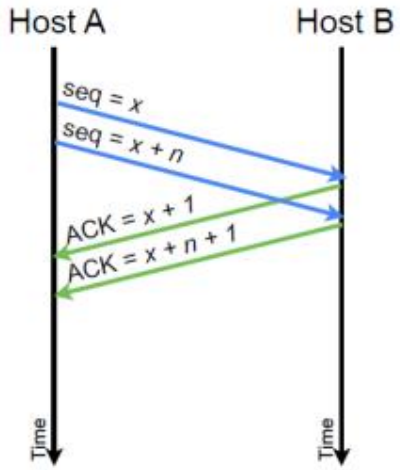



- The SYN and ACK flags in the TCP header signal connection progress
- Initial packet sent from A to B has the SYN bit set, and includes a randomly chosen sequence number
- Reply also has SYN bit set, and ACKnowledges the initial packet, and also has a randomly chosen sequence number
- Handshake completed by ACKnowledgement of the second packet
- Random sequence numbers give robustness to delayed packets or restarted hosts
- Acknowledgements ensure reliability
- Similar handshake with the FIN bit in the header signifies disconnection

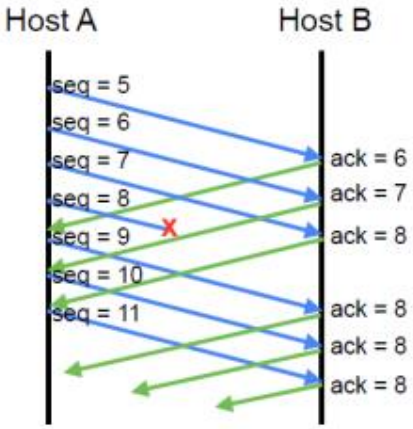
Jackson Dam (2619114D)

Week 4 Lecture 2

TCP reliability

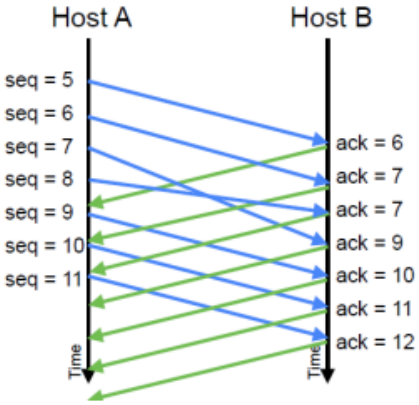



- TCP connections are reliable
  - each TCP packet has a sequence number and acknowledgement number
  - sequence number shows how many bytes are sent (unrealistic example shows 1 byte sent per packet)
  - acknowledgement number specifies the next byte expected to be received
  - this is cumulative positive acknowledgement
  - only acknowledge contiguous (in order) data packets (sliding window protocol, so several data packets are in flight)
  - if a packet is lost, future packets will receive duplicate acknowledgements
  - TCP layer retransmits lost packets invisibly to the application




- Loss is detected by:
  - triple duplicate ACK → some packets lost, but later packets arriving
  - triple duplicate = four identical ACKs in a row
  - timeout → is when you send data but acknowledgements stop returning (receiver or network path has failed)

More on TCP reliability




- Packet reordering:
  - duplicate ACKs can be caused by packet delay, leading to reordering
  - gives appearance of loss, even though data was merely delayed
  - TCP uses triple duplicate ACKs to indicate loss, to prevent reordered packets causing retransmissions
  - this assumes packets will only be delayed a little; if delay is enough to cause a triple duplicate ACK, it will be treated as loss and a retransmission will occur
- End result: packets are delivered in order, even after loss occurs
  - a recv() for missing data will block until the data arrives
  - TCP always delivers data in an in-order contiguous sequence

Jackson Dam (2619114D)

## Congestion control

- Definition: adapting speed of transmission to match available end-to-end network capacity, to prevent congestion collapse of a network
- Can implement congestion control on the network or the transport layer
- Network layer:
  - \+ Safe
  - \+ Ensures all transport protocols are congestion controlled
  - – Requires all applications to use the same congestion control scheme
- Transport layer:
  - \+ Flexible
  - \+ Transports protocols can optimise congestion control for applications
  - – A misbehaving transport can congest the network
- Key principles of congestion control are conservation of packets, and additive increase and multiplicative decrease (AIMD) of the sending rate, together ensuring network stability

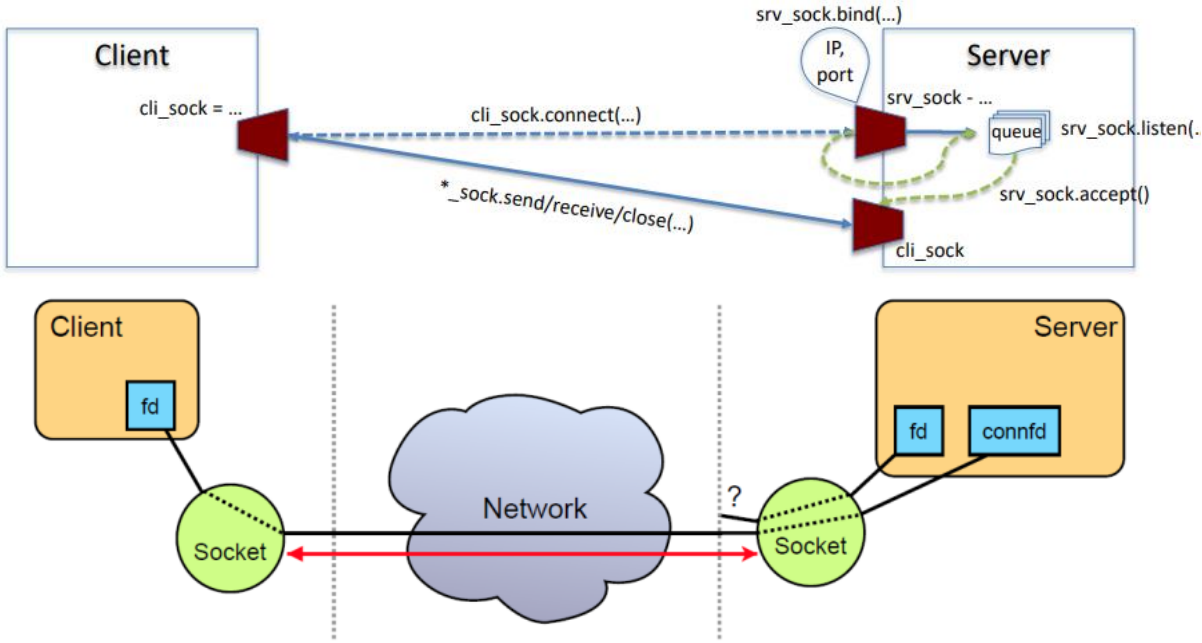
## More on congestion control

- Conservation of Packets:
  - the network has a certain capacity
  - the (bandwidth * delay time) product of the path (note that volume/time * time = volume)
- When in equilibrium at that capacity, send one packet for each acknowledgement received
- Total number of packets in transit is constant
- Automatically reduces sending rate as network gets congested and delivers packets more slowly

- Adjust sending rate according to an Additive Increase Multiplicative Decrease (AIMD) algorithm:
  - start slowly, increase gradually to find equilibrium
  - add a small amount to the sending speed each time interval without loss
  - respond to congestion rapidly
  - multiply sending window by some factor $\beta < 1$ each interval where loss is seen
  - faster reduction than increase $\rightarrow$ stability

Jackson Dam (2619114D)

## TCP sockets code example




```
cli_sock = socket.socket(socket.AF_INET,
     socket.SOCK_STREAM)



cli_sock.connect(('somehost', port num))
request = …
cli_sock.send(request)
response = cli_sock.recv(num bytes)
cli_sock.close()
```

```
srv_sock = socket.socket(socket.AF_INET,
     socket.SOCK_STREAM)
srv_sock.bind(('somehost', port num))
srv_sock.listen(size of queue)

cli_sock, cli_addr = srv_sock.accept()
request = cli_sock.recv(num bytes)
response = …
cli_sock.send(response)
cli_sock.close()
```

## TCP sockets

- Sockets initially unbound, and can either accept or make a connection
- Most commonly used in a client-server fashion:
  - one host makes the socket bind(), listen(), and accept() connections on a known 16-bit port number, making it into a server
  - the other host makes the socket connect() to that port on the server
  - once connection is established, either side can send() data into the connection, where it becomes available for the other side to recv()

Jackson Dam (2619114D)

TCP port numbers

| Port Range | | Name | Intended use |
|---|---|---|---|
| 0 | 1023 | Well-known (system) ports | Trusted operating system services |
| 1024 | 49151 | Registered (user) ports | User applications and services |
| 49152 | 65535 | Dynamic (ephemeral) ports | Private use, peer-to-peer applications, source ports for TCP client connections |

- Servers must listen on a known port
- IANA maintains a registry
- Distinction between system and user ports ill-advised
  - security problems resulted
- Insufficient port space available
  - >75% of ports are registered
- TCP clients traditionally connect from a randomly chosen port in the ephemeral range

Communication over TCP

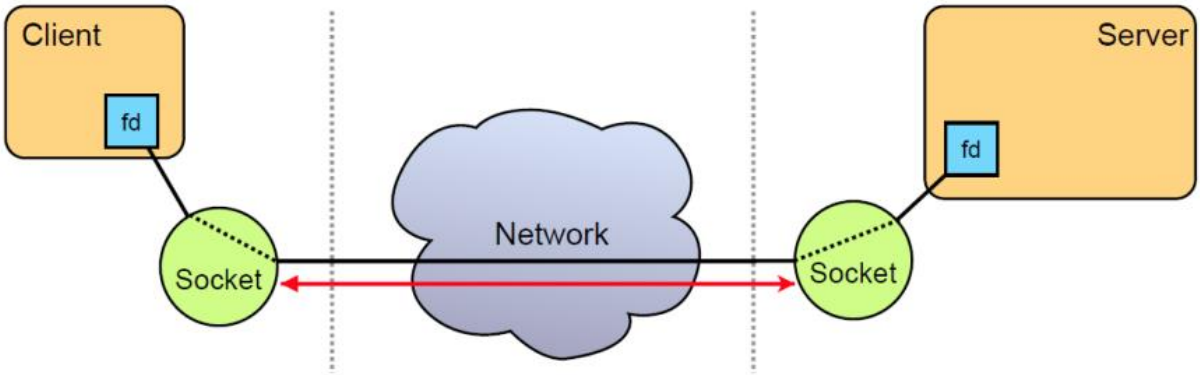
- 3-way handshake happens during the connect()/accept() calls
- Call send() to transmit data
  - will block until the data can be written, and returns amount of data sent
  - might not be able to send all the data, if the connection is congested
  - returns actual number of bytes sent
- Call recv(X) to read up to X bytes of data from a connection
  - will block until some data is available or the connection is closed
  - returns a bytes object with the data that was received from the socket
  - received data is not null terminated – potential security risk?
- All errors handled through exceptions

More on communication over TCP

- The send() call enqueues data for transmission
- This data is split into segments
- Each segment is placed in a TCP packet
- That packet is sent when allowed by the congestion control algorithm
- If the data in a send() call is too large to fit into one segment, the TCP implementation will split it into several segments
  - similarly, several send() requests might be aggregated into a single TCP segment
  - both are done transparently by the TCP implementation and are invisible to the application
- Implication: the data returned by recv() doesn't necessarily correspond to a single send() call
- The recv() call can return data in unpredictably sized chunks – applications must be written to cope with this

More on UDP sockets

Jackson Dam (2619114D)




```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((..., ...))
sock.sendto(..., (..., ...))
sock.recvfrom(...)
sock.close()
```

Communication over UDP

- The sendto() call sends a single datagram
  - each call to sendto() can send to a different address, even though they use the same socket
  - alternatively, connect() to an address, then use send() to send the data
  - connect() on a UDP socket merely sets a default destination address for future send() calls
- The recvfrom() call may be used to read a single datagram (also returns the sender's address)
  - recv() also usable but doesn't provide the sender's address
- Unlike TCP, each UDP datagram is sent as exactly one IP packet (which may however be fragmented in IPv4)
  - each recvfrom() corresponds to a single sendto()

More on communication over UDP

- Transmission is unreliable: packets may be lost, delayed, reordered, or duplicated in transit
  - the application is responsible for correcting the order, detecting duplicates, and repairing loss, if necessary
  - generally requires the sender to include some form of sequence number in each packet sent
- Application must organise the data so it's useful if some packets lost
  - e.g. streaming video with intermediate and predicted frames

Jackson Dam (2619114D)

## Even more on communication over UDP

- Need to provide sequencing, reliability, and timing in applications
  - sequence numbers and acknowledgements
  - retransmission and/or forward error correction
  - timing recovery
- Need to implement congestion control in applications
  - to avoid congestion collapse of the network
  - should be approximately fair to TCP
  - difficult to do well – TCP congestion control covers many corner cases that are easy to miss
- RFC 3448 provides a detailed specification for a well-tested algorithm
- IETF RMCAT working group developing standard congestion control algorithms for
  - interactive video applications running over UDP
- Google's QUIC protocol builds on UDP to give more sophisticated service

## Applications of internet transfer protocols

- UDP Applications
  - useful for applications that prefer timeliness to reliability
  - voice-over-IP
  - streaming video
  - must be able to tolerate some loss of data
  - must be able to adapt to congestion in the application layer
- TCP Applications
  - useful for applications that require reliable data delivery, and can tolerate some timing variation
  - file transfer and web downloads
  - email
  - instant messaging
  - TCP is the default choice for most applications

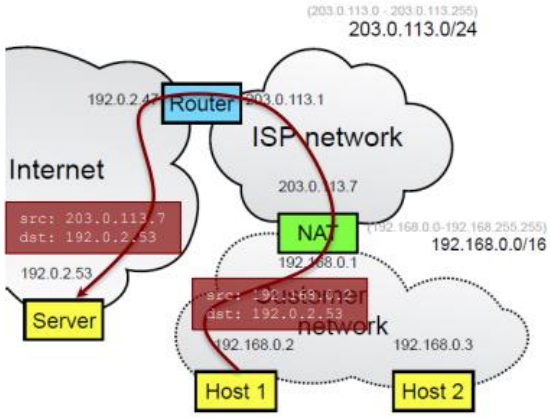
Jackson Dam (2619114D)

Topic 7 – NAT and higher layers

Week 5 Lecture 1

Network Address Translation – ISP scenario

- Customer acquires a NAT box, which gets a customer's IP address
    - customer gives each host a private address
- NAT performs address translation
    - rewrites packet headers to match its external IP address
    - likely also rewrites the TCP/UDP port number
- The NAT hides a private network behind a single public IP address
    - private IP network addresses:
    - 10.0.0.0/8 (Class A - Host addresses in the range: 10.0.0.1 - 10.255.255.254)
    - 172.16.0.0/12 (Class B - Host addresses in the range: 172.16.0.1 - 172.31.255.254)
    - 192.168.0.0/16 (Class C - Host addresses in the range: 192.168.0.1 - 192.168.255.254)
    - gives the illusion of more address space




Note on IP blocks

- 192.168.0.0/16 has 2^(32-16) = 2^16 = 65536 addresses in the range, as IPv4 addresses consist of 32 bits.

NAT with internet transport protocols

- NAT with TCP:
    - outgoing connection creates state in the NAT box's state table
    - the NAT box expects periodic data from each host, or the NAT state will time out (TCP session considered terminated)
    - recommended timeout interval is 2 hours, but many NAT boxes use shorter
    - no state is made for incoming connections
    - NAT boxes do not know where to forward incoming connections without manual configuration (port forwarding)
    - affects servers behind a NAT, or peer-to-peer applications

Jackson Dam (2619114D)

- NAT with UDP:
  - NATs tend to have short time outs for UDP
  - not connection-oriented, so the NAT box can't detect end of flows
  - recommended time out interval is 2 minutes, many use shorter
  - VoIP NAT traversal standards suggest sending a keep alive message every 15 seconds
  - P2P connections are easier than with TCP
  - UDP NATs are often more permissive about allowing incoming packets than TCP NATs
  - many allow replies from anywhere to an open port

More on network address translation

- Many applications fail with NAT:
  - client-server applications, where the client is behind NAT, work without changes (web, email)
  - client-server applications with the server behind NAT fail (need port forwarding)
  - P2P applications fail – complex algorithm (ICE – RFC 5245) needed to connect
- NAT provides no security benefit on its own (although most NATs include a firewall)

Even more on network address translation

- NAT breaks many applications – why use it?
  - many ISPs have insufficient addresses to give customers their own prefix
  - many customers don't want to pay their ISP for more addresses
  - both problems are caused by IPv4 limited address space
  - IPv6 may help remove need for NAT (cheap, plentiful addresses)
  - to avoid re-numbering a network when moving ISP, hardcoding IP addresses in config files / apps is a bad idea compared to DNS names
  - many people do it anyway - makes changing IP addresses difficult

- IPv6 tries to make renumbering networks easier, by providing better auto-configuration (insufficient experience to know how well this works in practice)

Jackson Dam (2619114D)

Higher layer protocols

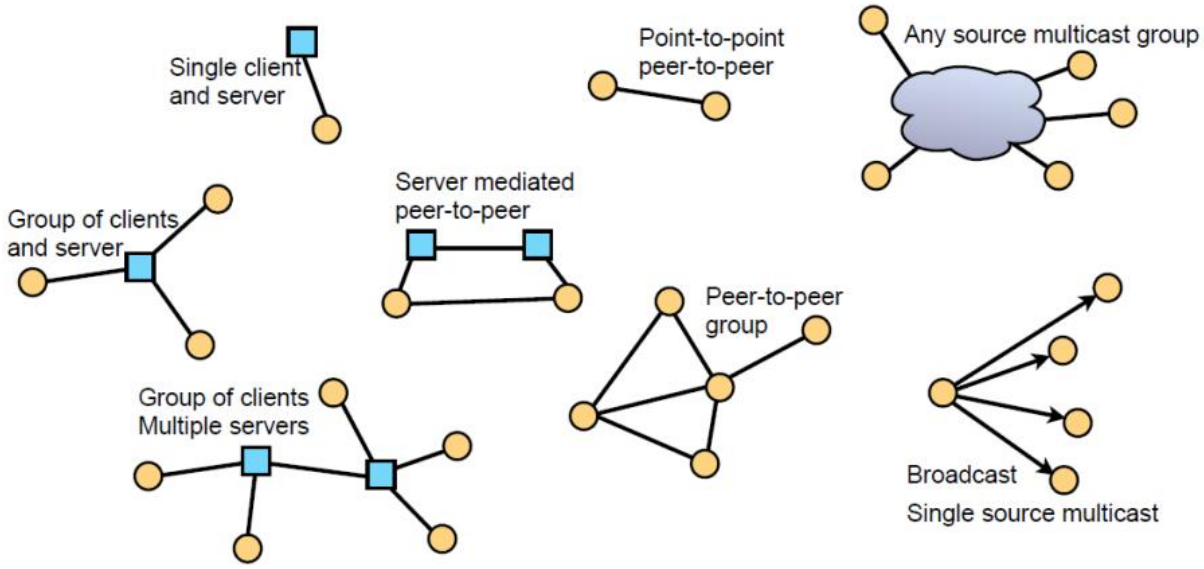
- They are the three layers session layer, presentation layer, application layer above the transport layer
- Goal of higher layer protocols are to support application needs e.g.:
  - setup/manage transport layer connections
  - name and locate application-level resources
  - negotiate data formats and perform format conversion if needed
  - present data in an appropriate manner

- Relatively ill-defined boundaries between layers
- Typically implemented within an application or library, rather than within the kernel

Session Layer (L5)

Required connection types

- Session layer must be aware of the required connection types at any given time




More on the session layer

- How to find participants?
  - look-up name in a directory (e.g. DNS, web search engine)
  - or a server mediated connection (e.g. instant messenger, VoIP call)
- How to setup connections?
  - direct connection to named host (→ NAT issues)
  - or mediated service discovery, followed by peer-to-peer connection
- How does session membership change?
  - does the group size vary greatly?
  - how rapidly do participants join and leave?
  - are all participants aware of other group members?

Jackson Dam (2619114D)

Session layer role in maintaining connections

- IP addresses encode location → mobility breaks transport layer connections
  - session layer must find new location, establish new connections
  - might be redirected by the old location – e.g., HTTP redirect
  - mobile devices may update a directory with new location
  - complexity is pushed up from the network to the higher layers (end-to-end principle)
- A single session may span multiple transport connections
  - session layer responsible for co-ordinating the connections
  - e.g., retrieving a web page containing images – one connection for the page, then one per image
  - e.g., a peer-to-peer file sharing application, building a distributed hash table

More on the session layer

- Some protocols rely on middleboxes or caches
  - web cache – optimise performance, moving popular content closer to hosts
  - email server – supports disconnected operation by holding mail until user connects
  - SIP proxy servers and instant messaging servers – locate users, respond for offline users
- The end-to-end argument applies, once again
  - only add middleboxes when absolutely necessary

Jackson Dam (2619114D)

Week 5 Lecture 2

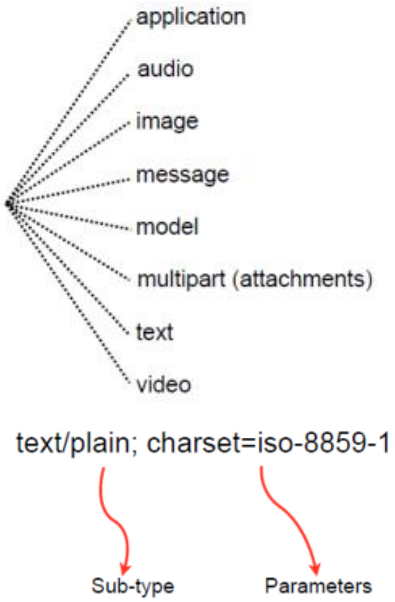
Presentation layer (L6)

Presentation layer

- Manages the presentation, representation, and conversion of data:
  - media types and content negotiation
  - channel encoding and format conversion
  - internationalisation, languages, and character sets
- Contains common services used by many applications

Media types

- Data formats often not self-describing (e.g. message includes data but not metadata describing format and meaning)
  - media types identify the format of the data
  - categorise formats into eight top-level types
  - each has many sub-types
  - each sub-type may have parameters
- Media types included in protocol headers to describe format of included data




Content negotiation

- Many protocols negotiate the media formats used
  - ensure sender and receiver have common format both understand
- Typically some version of an offer-answer exchange
  - the offer lists supported formats in order of preference
  - receiver picks highest preference format it understands, includes this in its answer
  - negotiates common format in one round-trip time

Channel encoding

- Does the protocol exchange text or binary data?
  - text – flexible and extensible
- High-level application layer protocols (e.g., email, web, instant messaging, ...)
  - binary – highly optimised and efficient
  - audio and video data (e.g., JPEG, MPEG, Vorbis, ...)
  - low-level or multimedia transport protocols (e.g., TCP/IP, RTP, ...)
- Recommendation: prefer extensibility, rather than performance, unless profiling shows performance is a concern
- Text-based protocols can't directly send binary data
  - so we need encoding...

Jackson Dam (2619114D)

## More on channel encoding

- Data must be encoded to fit the character set in use and the encoding must be signalled
  - may require negotiation of an appropriate transfer encoding, if data passing through several systems
- Issues when designing a binary coding scheme:
  - must be backwards compatible with text-only systems
  - some systems only support 7-bit ASCII, enforce a maximum line length, etc.
  - must survive translation between character sets (legacy systems using ASCII, national extended ASCII variants, EBCDIC, etc.)
  - must not use non-printing characters
  - must avoid escape characters that might be interpreted by the channel (e.g., $ \ # ; & “)

## Channel encoding considerations

- Many protocols send binary directly, not encoded in textual format
  - e.g. TCP/IP headers, RTP, audio-visual data
- Two issues to consider:
  - byte ordering
    - the Internet is big endian (most significant bit at lowest storage address)
    - must convert from little-endian PC format (least significant bit at lowest storage address)
  - word size
    - how big is an integer (e.g., 16, 32, or 64 bit)?
    - how is a floating point value represented?

## Application layer (L7)

- Protocol functions specific to the application logic
  - deliver email, retrieve a web page, stream video, ...
- Issues to consider:
  - what types of message are needed?
- Highly application dependent – difficult to give general guidelines
  - how do interactions occur?
  - does the server announce its presence? or does it wait for the client to start?
  - is there an explicit request for every response? can the server send unsolicited data?
  - is there a lot of chatter, or does the communication complete within a single round?
- How are errors reported?
  - many applications settled on a three digit numeric code

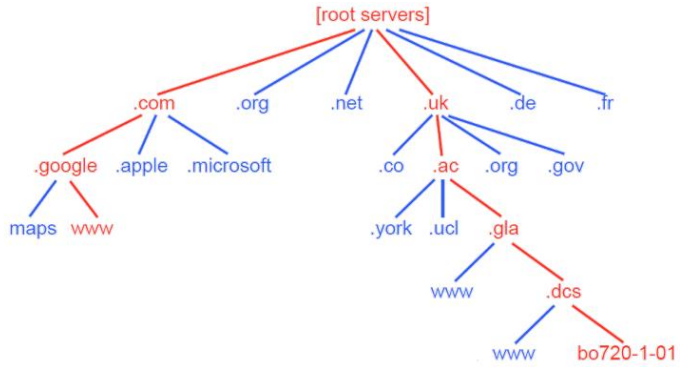
| Error Code | Meaning |
|---|---|
| 1xx | In progress |
| 2xx | Ok |
| 3xx | Redirect |
| 4xx | Client error |
| 5xx | Server error |

  - first digit indicates response type, last two digits give specific error/response
  - allows signalling new error types, but lets older clients give meaningful response

Case study – Domain Name Service

- The most widely used UDP-based application is the domain name service/system (DNS)
  - the network operates entirely on IP addresses, and has no concept of names for hosts
  - the DNS is an application that translates from user-visible names to IP address
  - www.dcs.gla.ac.uk → 130.209.240.1
  - DNS is an application layer protocol, running over the network
  - not necessary for the correct operation of the transport or network layers, or lower
- Why run over UDP?
  - request-response protocol, where it was thought TCP connection setup and congestion control was too much overhead
  - unclear if this design is appropriate (e.g. UDP has very little embedded security)

DNS operation




- Hierarchy of DNS zones
- One logical server per zone
- Delegation follows hierarchy
- Hop-by-hop name look-up, follows hierarchy via root
- Results have TTL, cached at intermediate servers

Jackson Dam (2619114D)

Case study - HyperText Transfer Protocol (HTTP)

- A generic, stateless, protocol which can be used for many tasks through extension of its request methods, error codes and headers
- Basic characteristics:
  - an application layer protocol (SSL at the presentation layer protocol)
  - request-response client-server (pull) protocol
  - presumes running on a reliable transport layer protocol (usually TCP/IP)
  - stateless -- each request is agnostic of previous requests
- Old default (HTTP/1.0): a single request/response per connection
- New default ( HTTP/1.1): multiple request/response pairs per connection
- Typing and negotiation of data representation
  - allows systems to be built independently of the data being transferred
- For text-based requests, binary data should be encoded, e.g., using base64 encoding

HTTP Uniform Resource Identifiers (URIs)

- Endpoints defined by a Uniform Resource Identifier consisting of 4 parts:
  - protocol (e.g., http, https)
  - hostname (IP address or DNS domain name)
  - port number (if omitted, the protocol's default port is used instead)
  - path and filename (location of requested resource), which can refer to items other than files/directories, mapped through the server's configuration
  - can also include request parameters (after a ?), named anchors (#anchor), etc.
  - cannot contain special characters, such as blank or ~
  - special characters are encoded in the form %XX, XX being the character's ASCII hex code

Jackson Dam (2619114D)

## HTTP requests

- Client parses URI
  - extracts protocol, hostname, port number
  - extracts path and filename
- Client constructs request message
  - format:
    Method Request-URI | HTTP-version | \r\n | (Misc headers \r\n)* | \r\n

```
GET / HTTP/1.0\r\n\r\n
```

```
GET /index.html HTTP/1.1\r\n
Host: www.gla.ac.uk\r\n
Accept: image/gif, image/jpeg, */*\r\n
Accept-Language: en-us\r\n
Accept-Encoding: gzip, deflate\r\n
User-Agent: Mozilla/4.0\r\n
\r\n
```

- Client connects to server using a TCP/IP socket
  - uses hostname and port number extracted earlier
  - if hostname is not an IP, must first resolve it to one
  - if no port number was defined, uses default for protocol
- Client sends request message through socket

- Server parses client request
  - maps path and filename to local resource
- Server constructs response message
  - format:
    HTTP-version | Status-code | Reason-phrase | \r\n | (Misc headers \r\n)* | \r\n [Message body]

```
HTTP/1.1 200 OK\r\n
Date: Tue, 16 Oct 2018 12:00:00 GMT\r\n
Server: Apache/2.2.14\r\n
Last-Modified: Fri, 14 Sep 2018 19:46:29 GMT\r\n
Accept-Ranges: bytes\r\n
Content-Length: 44\r\n
Connection: close\r\n
Content-Type: text/html\r\n
<html><body>Hello World!</body></html>
```

- Client reads response from socket until completion (i.e., Content-Length bytes received after last \r\n) or until server closes connection (if keep-alive is off)
- Client parses HTML and extracts URIs
- Subsequent requests start from the beginning (or reuse this socket, if keep-alive is on)

Jackson Dam (2619114D)

Part B – Security/Privacy Basics

Topic 8 – Security and Privacy

Week 6 Lecture 1

Network Monitoring...not that private

- Possible to intercept traffic on a network
- Many countries monitor traffic for legal reasons
  - much is desirable – good reasons for law enforcement to intercept some traffic – but Edward Snowden showed pervasive monitoring widespread
  - IETF consensus: "we cannot defend against the most nefarious actors while allowing monitoring by other actors no matter how benevolent some might consider them to be, since the actions required of the attacker are indistinguishable from other attacks"
- Organisations may monitor traffic for business reasons
  - "Your call may be monitored for quality and training purposes" – regulatory requirements to be able to monitor some traffic
  - to support network operations and trouble-shooting
- Malicious users may monitor traffic on a link
  - for example, many Wi-Fi links have poor security allowing anyone on the same Wi-Fi network to observe all traffic on that network
  - hacked routers may allow monitoring of backbone links
  - steal data and user credentials; identity theft; active attacks

CIA triad

- Confidentiality, Integrity, Availability
  - confidentiality : who is allowed to access what
  - integrity: data to be protected and not tampered with/modified/deleted by unauthorized party(ies)
  - availability: data to be protected but also available when needed

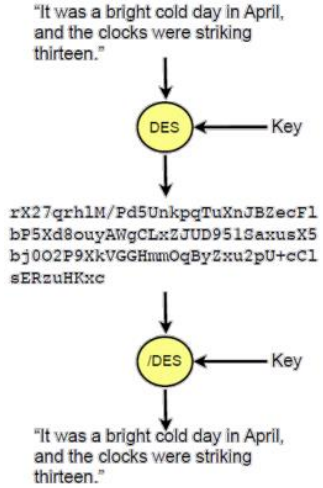
Towards CIA

- Must encrypt data in a computationally efficient manner to serve the needs for CIA.
- Two basic approaches
  - symmetric cryptography (Advanced Encryption Standard (AES))
  - asymmetric (public key cryptography) (the Diffie-Hellman algorithm, the Rivest-Shamir-Adleman (RSA) algorithm, elliptic curve-based algorithms
- Could also use complex mathematics – course will not attempt to describe

Jackson Dam (2619114D)

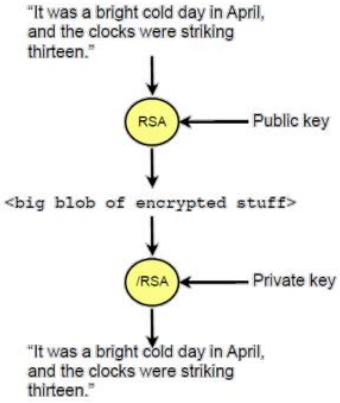
## Symmetric cryptography

- Function converts plain text into cipher-text
  - fast – suitable for bulk encryption
  - cipher-text is binary data, and may need base64 encoding
- Conversation is protected by a secret key
  - the same key is used to encrypt as is used to decrypt
- Key must be kept secret, else security lost
  - problem: how to distribute the key?
- Suitable for on-device encryption (where the key is stored securely on device hardware, used for fast decryption of data for apps etc).




## Asymmetric (public key) cryptography

- Key split into two parts
  - public key – is widely distributed
  - private key – must be kept secret
- Encrypt using public key → need private key to decrypt
- Public keys are published in a well-known directory (e.g. Certificate Authority)
  - solves the key distribution problem
  - problem: very slow to encrypt and decrypt




## Hybrid cryptography

- Use combination of public-key and symmetric cryptography for security and performance
  - generate a random, ephemeral (limited time validity) session key that can be used with symmetric cryptography
- Use a public-key system to securely distribute this session key that was encrypted by the public key – relatively fast, since session key is small
- Encrypt the data using symmetric cryptography, keyed by the session key
- The receiver of the data can later decrypt it, by first decrypting the session key with its own private key, then decrypting the data with this session key
- Example: Transport Layer Security (TLS) protocol used with HTTP (i.e. HTTPS)

Jackson Dam (2619114D)

## Authentication

- Encryption can ensure confidentiality – but how to tell if a message has been tampered with (i.e. keeps its integrity)?
    - use combination of a cryptographic hash and public key cryptography to produce a digital signature
    - gives some confidence that there is no man-in-the-middle attack in progress
    - can also be used to prove origin of data

## More on authentication

- Cryptographic hash functions generate a fixed length (e.g., 256 bit) hash code of an arbitrary length input value
- Should not be feasible to derive input value from hash
- Should not be feasible to generate a message with the same hash as another
- Examples: MD5, SHA-1 (both broken, don't use), SHA-2 (a.k.a SHA-256)

## Even more on authentication

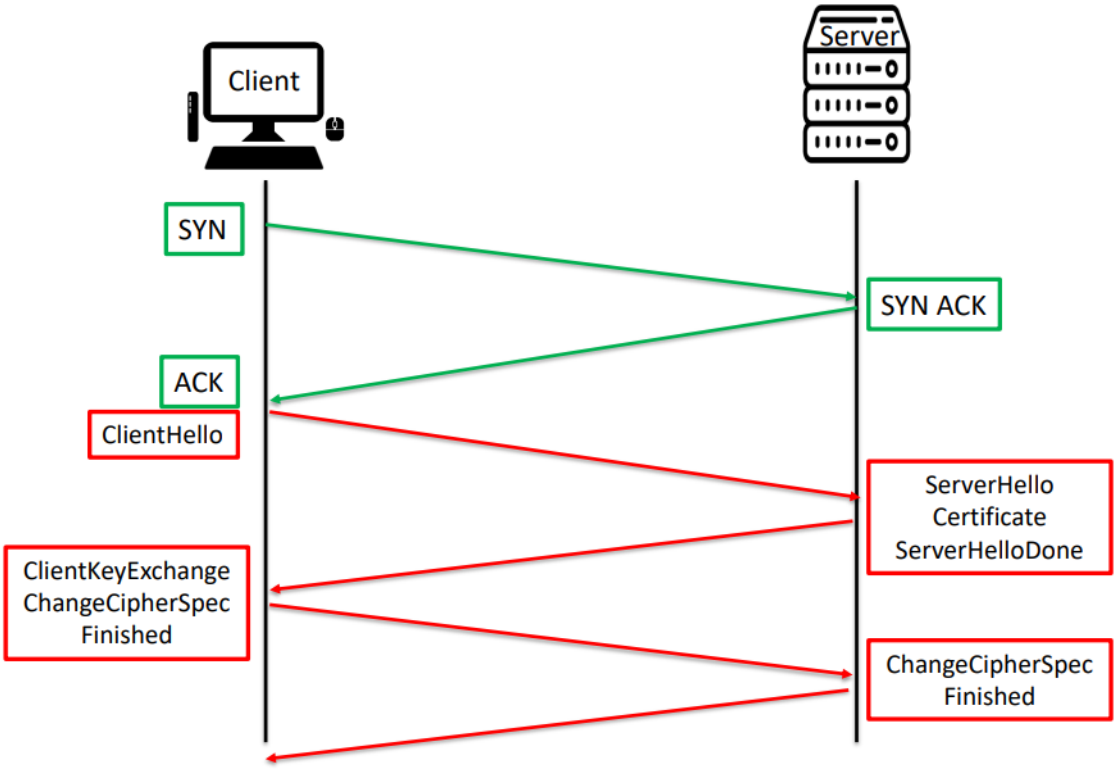
- Digital signature algorithms generating a digital signature by:
    - generating a cryptographic hash of the data
    - encrypt the hash with your private key to give a digital signature
- To verify a digital signature:
    - re-calculate the cryptographic hash of the data
    - decrypt the signature using the public key, compare with the calculated hash value → should match

## Transport Layer Security (TLS)

- De facto standard/protocol for Internet security
- Enables privacy and data integrity between two communicating applications
- Based on Secure Socket Layer protocol (SSL v.3)
- Used in every browser
- Now we have versions 1.2 and 1.3
- Reliable in terms of CIA

Jackson Dam (2619114D)

TLS handshake (HTTPS foundation)




IPSec

- A framework of open standards that incorporates/employs cryptographic secure services in IP networks
- Initially developed by IETF in 1998 for IPv6, then adapted for IPv4
- Resides on the network layer
- Implemented in the OS-level (mainly, and often implemented in kernel-space)
- Quite complex

IPSec modes

- Tunnel mode (used for gateway-to-gateway, end host-to-gateway, and gateway-to-server, or whenever gateway acts as a proxy for the hosts behind it)
- Transport mode (end-to-end security between hosts, used for host-to-host)
- Also used for end host-to-gateway, where gateway is treated as a host (vulnerable to eavesdroppers)

Jackson Dam (2619114D)

## IPSec & ISAKMP

- Security Association (SA): shared security attributed between two network components.
- SAs in IPSec enabled by the Internet Security Association & Key Management Protocols (ISAKMP).
- The ISAKMP protocol (RFC 2408) defines procedures for:
  - authentication
  - creation/management of SAs
  - key generation /key transport techniques
  - mitigation of threats (e.g. DoS attacks, replay attacks)

## IKE, ESP & AH

- Internet Key Exchange (IKE): used to establish SAs – it builds upon ISAKMP and the Oakley protocol.
- Encapsulating Security Payload (ESP): payload encryption ensuring authenticity, integrity and confidentiality – it does not provide integrity and authentication for the ENTIRE packet
- Authentication Header (AH): authentication only for payload and header

## IPSec negotiation phases

- IKE Negotiation Phase 1
  - establishment of SAs between two entities
  - two modes: Main or Aggressive
  - 4 techniques for authentication (public key signatures, symmetric key, public key encryption, revised public key encryption)
  - uses ephemeral Diffie-Helman (EDH) to establish session key
  - provides perfect forward secrecy
- IKE Negotiation Phase 2
  - establishment of SAs for ESP/AH protocols
  - one mode : Quick Mode

Jackson Dam (2619114D)

Week 6 Lecture 2

## Privacy

- Privacy can have different definitions
    - legal privacy (e.g., someone's right to be left alone)
- Our focus: data privacy (a.k.a information privacy)
- Data privacy focus: use and governance of data

## Personal data

- Personal data is defined as any information about a living individual which is capable of identifying that individual
- Sensitive personal data is defined as any information relating to an individual's racial or ethnic origin, political opinions, religious beliefs, trade union membership, physical or mental health or condition, sexual life, alleged or actual criminal activity and criminal record.
- Under GDPR sensitive personal data is referred to as "special categories of personal data"

## General Data Protection Regulation (GDPR)

- EU Regulation (a set of laws) applied in 2018
- UK: Data Protection Act 2018 (inherits from GDPR)
- Data Protection Act: defines your data privacy rights and "how your personal information is used by organisations, businesses or the government."

## More on GDPR

- In order to comply with the new law:
    - one must have a legitimate reason for processing data – this covers what and how much processing of the data can be undertaken
    - consent must be freely and unambiguously given and can be easily withdrawn
    - data processing activities must start with "privacy by design and default"

## GDPR principles

- Lawfulness, fairness, and transparency – as with Data Protection Act
- Purpose limitation – only collect and use for specific purposes
- Data minimisation – only collect the data needed for a specified purpose
- Accuracy – data needs to be correct and kept up-to-date
- Storage limitation – data should not be kept for longer than need to fulfil specific purpose
- Integrity and confidentiality – data needs to be protected appropriately,
    - e.g. secured through appropriate access control, encryption, etc.
- Accountability – a data processor must be able to prove that they are complying with the GDPR regulations

Jackson Dam (2619114D)

Data protection design principles

- Data Protection requires avoiding harm to individuals by possible misuse or mismanagement of personal data
- Any data collected, used, or stored about a person therefore is under the auspice of relevant legislation (e.g. GDPR)
- Data protection principles include: data should only be collected for a specific purpose, collected information can only be used for specific and agreed purposes
- Records and stored data have to be kept accurate, up to date, safe and secure
- Information should only be processed lawfully
- Data subjects should be granted access in line with relevant legislation

Developing secure network applications – the robustness principle

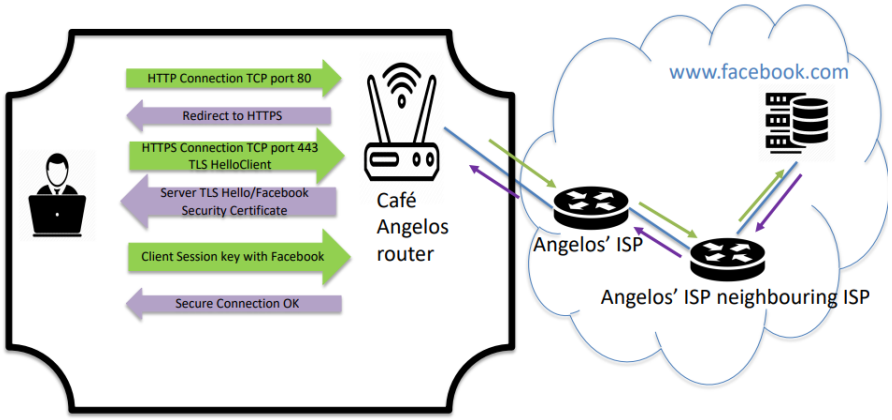
- Also known as Postel's Law
- "Be liberal in what you accept, and conservative in what you send"
- "Be strict when sending and tolerant when receiving"

Some types of Man-in-the-Middle (MitM) attack

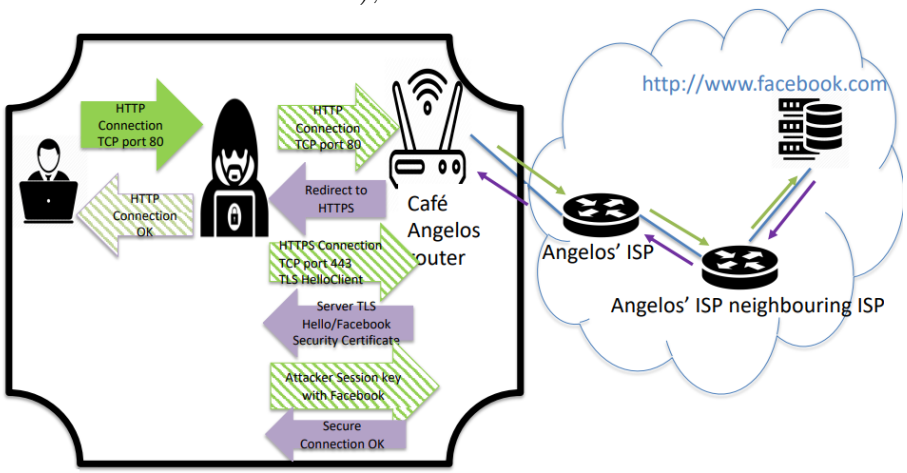
- ARP poisoning
- Port stealing
- IP address spoofing
- ICMP redirection
- SSL hijacking
- HTTPS Spoofing
- SSL Stripping

Jackson Dam (2619114D)

SSL stripping




- Note in the image below:
  - attacker intercepts the client's initial HTTP connection request to the server
  - attacker impersonates the client and sends the request to the server on their behalf
  - they can then establish a HTTPS connection with the server, and send the relevant TLS handshake messages to successfully establish a session with the server
  - the attacker then impersonates the server to the client (e.g. pretends to be Facebook), and serves them with an insecure connection



Jackson Dam (2619114D)

Validating input data

- Networked applications fundamentally deal with data supplied by untrusted third parties
  - data read from the network may not conform to the protocol specification
  - due to ignorance and/or bugs
  - due to malice, and a desire to disrupt services
- Must carefully validate all data before use
- The network is hostile: any networked application is security critical
  - must carefully specify behaviour with both correct and incorrect inputs
  - must carefully validate inputs and handle errors
  - must take additional care if using type- and memory-unsafe languages, such as C and C++, since these have additional failure modes

Buffer overflow attacks

- Memory-safe programming languages check array bounds
  - fail cleanly with exception on out-of-bound access
  - behaviour is clearly defined at all times
- Unsafe languages, such as C and C++, don't check
  - responsibility of the programmer to ensure bounds are not violated
  - easy to get wrong – typically results in a "core dump" – or undefined behaviour
- Buffer overflows in network code are one of the main sources of security problems
  - if you write network code in C/C++, be very careful to check array bounds
  - if your code can be crashed by received network traffic, it probably has an exploitable buffer overflow

Tips for secure applications

- Do not trust client-side data
  - check for unescaped special characters
  - null characters should all be removed
- Make sure that you go through re-authentication procedures (issuing new sessions keys) for your client

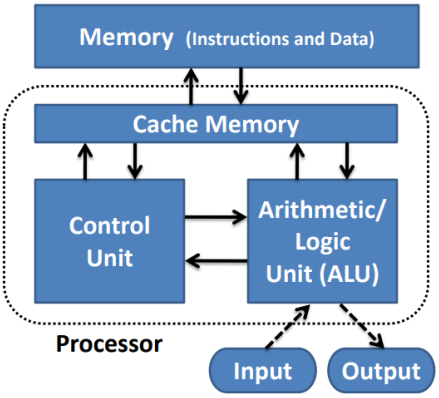
Jackson Dam (2619114D)

Part C – Operating System Essentials

Topic 9 – OS Intro

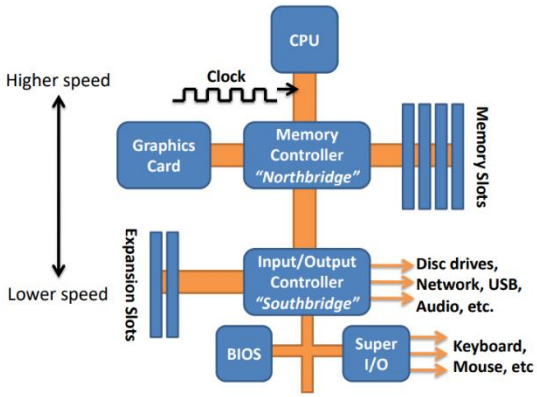
Week 7 Lecture 1

von Neumann architecture and the processor




- Computer architecture is largely standardised at a high level of abstraction by the von Neumann architecture
- The processor (CPU) has an ALU + control unit
- CPU often contains some internal, high-speed cache memory

A (fairly) modern PC architecture




- Architecture is divided into layers/regions according to speed of operation

Instruction set architecture (ISA)

- The interface of the processor for programs running on it to give instructions
- Encompasses available instructions, available registers, number of operands for each instruction, size/types of each operand
- How to access the operands, how many operands can be in register vs. in memory, how many clock ticks to execute an instruction

Registers

- Registers are storage areas for data being worked on inside the CPU
- Arithmetic and logic instructions are often register-only because register memory is much faster
- Data-transfer instructions are used for register to main memory transfers and vice versa
- Registers used for the arithmetic and logic instructions are general-purpose registers
- Processors also have special registers like the program counter (PC) and the stack pointer (SP)

Jackson Dam (2619114D)

## Goals of an OS

- Act as intermediary program between the user and the computer hardware (through the ISA)
- Make the computer system usable
- Allow the user to execute programs as per needs
- Use the computer hardware efficiently

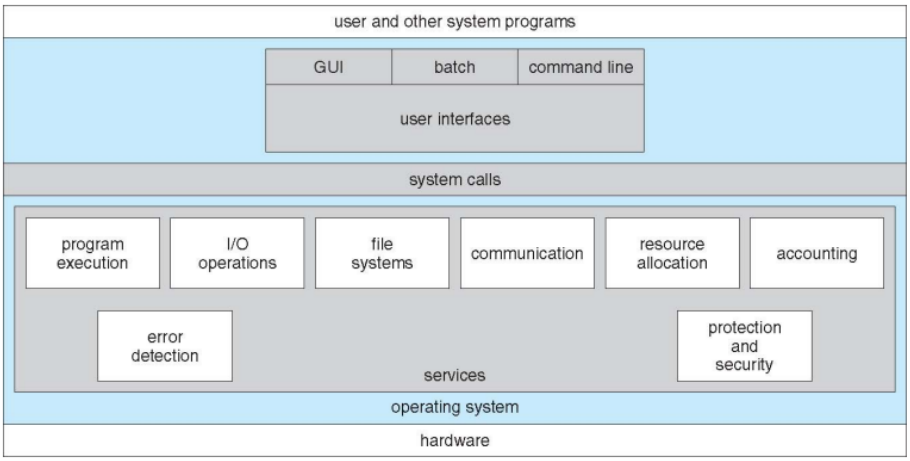
## Roles of an OS

- OS acts as a resource allocator:
  - manages all resources
  - in case of conflicting requests ensures efficient and fair resource use
- OS acts as a control program:
  - try to prevent errors and improper use of the computer system

## Core components

- Kernel:
  - first thing that's started, highest privilege, manages all other programs
  - the program running at all times on the computer; core part of the OS
  - everything else is either a system program or an application program
- Bootstrap program:
  - also known as firmware, BIOS, etc.
  - loaded at power-up or reboot
  - typically stored in ROM (Read Only Memory), flash ROM to update
  - initializes all aspects of system
  - loads operating system and starts execution

## OS services (in context of higher to lower levels)



Jackson Dam (2619114D)

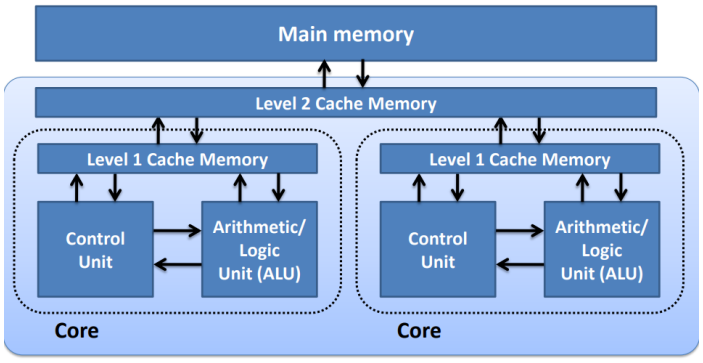
## Volatility and cache

- Volatile = short-term memory, can change at any time
- Non-volatile = long-term memory
- Cache = copying data from main memory to faster storage, for later quick access

## Parallel architectures

- Recent renewed interest in parallel architectures
- Faster, but they complicate system software
- Not always able to hide the complexity from applications (especially to take advantage of performance)

## Multi-core architecture diagram




*A dual-core* processor

## Coarser-grained parallelism: clusters

- Can link computers with a high-speed network for more performance
- Blade servers = computers in a cluster without screens
- Applications run across the cluster (ideally), but not all applications can have their workloads easily decomposed in this way

## Computer system organisation

- One or more CPUs, device controllers → access to shared memory
- CPUs and devices competing for memory cycles
- Each device controller is in charge of a particular device type and has a local buffer
    - CPU moves data between main memory and local buffers
    - I/O communication from the device to local buffer of controller
    – device controller informs CPU that it has finished its operation by causing an interrupt

## Direct memory access

- Instead of waiting for an I/O request to be processed, a single-purpose DMAC (DMA controller) can be used to directly copy from device controller buffer storage to main memory, without CPU intervention

Jackson Dam (2619114D)

DMA hardware requirements

- DMAC and CPU accessing memory simultaneously require either dual-port memory or arbitration circuitry (both complex/expensive)

Processes

- Process = program in execution that needs resources
- Program is a passive entity (executable file in storage/disk), process is an active entity (running/ongoing task on CPU)
- A program is started by loading its executable file into memory
- One program can have several processes
- Single-threaded processes have one program counter specifying memory location of next instruction to execute
- Multi-threaded processes have one program counter per thread

Process management

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronisation
- Providing mechanisms for process communication
- Providing deadlock handling

Memory management

- All instructions in memory
- Memory management determines what is in memory
- Involves:
    - keeping track of which parts of memory are in use and by what
    - deciding which processes (or parts thereof) and data to move into and out of memory
    - allocating and deallocating memory space as needed

Storage management

- Filesystem management:
    - files organised into directories
    - access control on most systems to determine who can access what
- OS:
    - handles creating, deleting, and editing files/directories
    - mapping files onto secondary storage
    - backing up files onto stable (non-volatile) storage media

Jackson Dam (2619114D)

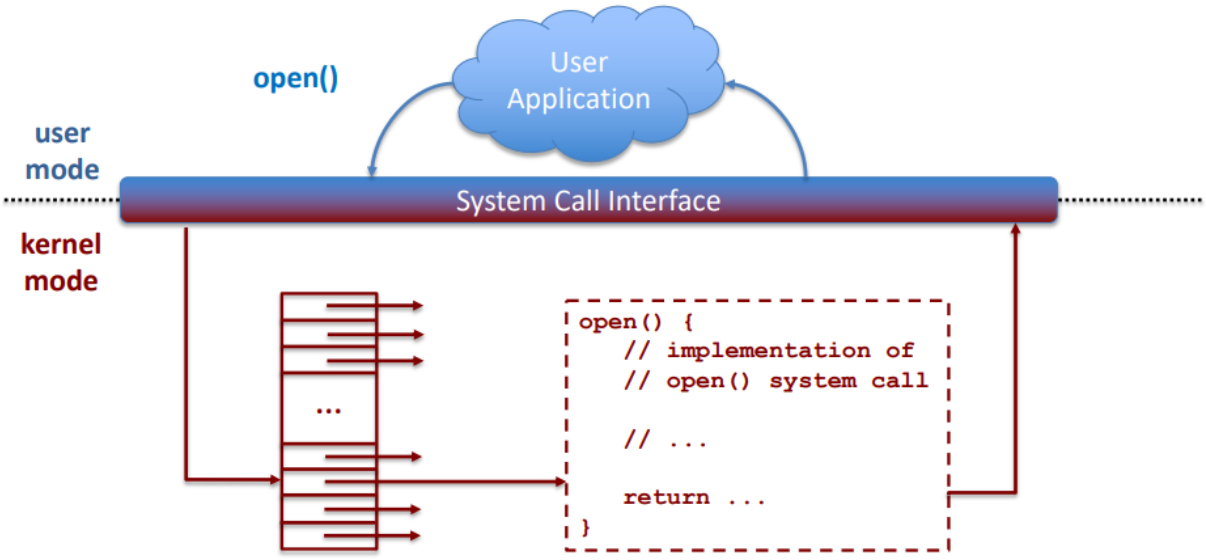
Topic 10 – System Calls and IPC

Week 7 Lecture 2

System Calls

- Provide a programming interface to the services provided by the OS
- Usually, system functions are accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
- Operating Systems have been implemented in high-level systems programming languages (C or C++) and there are wrapper functions for system calls available in these

Example of system call




- Application calls open(), and control is transferred from userland to the kernel
- The kernel uses its elevated privilege and runs its implementation of open(), then transfers control back to the user application
- Some instructions are known as privilege instruction (only the OS can issue these)
- If a user program tries to issue these, the trap mechanism gives control to the kernel so that it can intervene / prevent the operation

System call parameter passing

- Methods for passing parameters to the OS kernel:
  - pass the parameters in registers
  - parameters stored in memory, and address passed as a parameter in a register
  - placed/pushed onto the stack by the program and popped off the stack by the operating system

Jackson Dam (2619114D)

Examples of Windows and Unix system calls

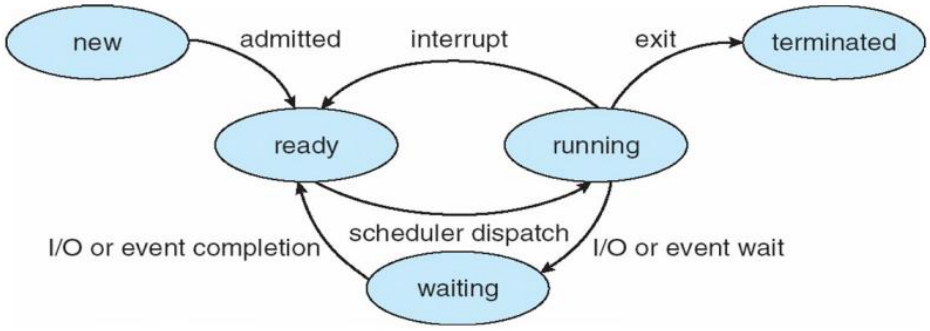
| | Windows | Unix |
|---|---|---|
| Process Control | CreateProcess() | fork() |
| | ExitProcess() | exit() |
| | WaitForSingleObject() | wait() |
| File Manipulation | CreateFile() | open() |
| | ReadFile() | read() |
| | WriteFile() | write() |
| | CloseHandle() | close() |
| Device Manipulation | SetConsoleMode() | ioctl() |
| | ReadConsole() | read() |
| | WriteConsole() | write() |
| Information Maintenance | GetCurrentProcessID() | getpid() |
| | SetTimer() | alarm() |
| | Sleep() | sleep() |
| Communication | CreatePipe() | pipe() |
| | CreateFileMapping() | shmget() |
| | MapViewOfFile() | mmap() |
| Protection | SetFileSecurity() | chmod() |
| | InitializeSecurityDescriptor() | umask() |
| | SetSecurityDescriptorGroup() | chown() |

OS services – program execution

Multi-process program example

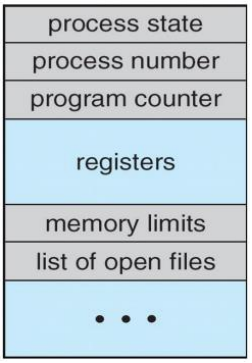
- Many web browsers run as a single process (some still do)
- But Google Chrome is multi-process:
  - browser process manages UI, disk/network I/O
  - renderer process
  - extension (plugin) process for each type of extension
  - processes for web apps ("web workers")

Diagram of process states




Process control blocks

- Information associated with each process, also known as Process Table Entry

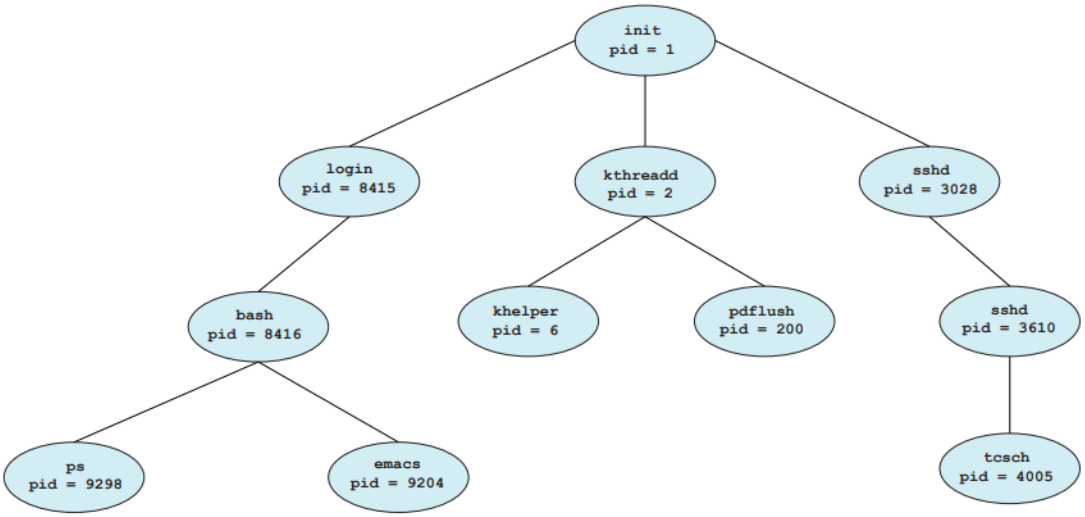


Jackson Dam (2619114D)

## Process creation

- Processes are managed as a process tree (parent processes can have child processes)
- System calls – fork(pid) to create child process and exec(pid)
- Execution options:
  - parent and children execute concurrently
  - parent waits for children to terminate
- Resource sharing options:
  - parent and children share all resources
  - children share subset of parent resources
  - parent and child share no resources

## Linux process tree example



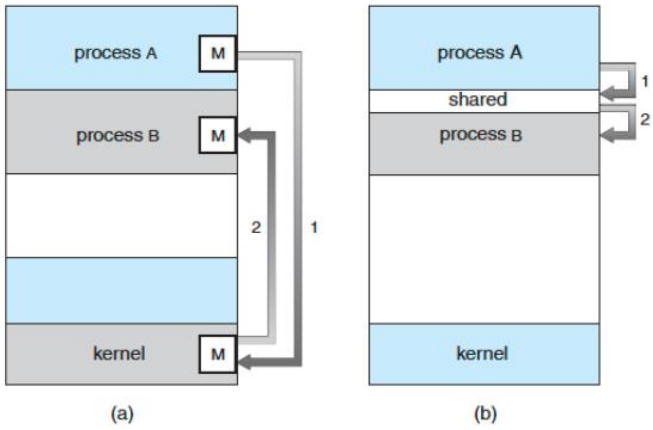

## OS services – communication

## Inter-process Communication (IPC)

- Processes may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Advantages of cooperating processes:
  - information sharing
  - computation speedup
  - modularity
  - convenience

## Communications models

- Two models of IPC: message passing through the kernel, shared memory



Jackson Dam (2619114D)

## Message passing

- Requires a "communication link" between communicating processes
- Provides at least two operations: send(message) and receive(message)
- Considerations:
  - direct vs indirect communication
  - synchronous vs asynchronous communication
  - automatic vs explicit buffering

## Message passing – direct communication

- Processes must name each other explicitly (symmetric):
  - send(P, message) – send a message to Process P
  - receive(Q, message) – receive a message from Process Q
- Alternatively, only the recipient is named (asymmetric)
  - send(P, message) – send a message to Process P
  - receive(id, message) – receive a message from any process (id set to the name of the sender when receiving a message)
- Communication links are established by the OS
- Exactly one link between a pair of communicating processes
- Links may be unidirectional, but usually are bi-directional

## Message passing – indirect communication

- Messages are directed and received from mailboxes
- Mailboxes are dedicated memory areas managed by the OS for the communication
- Links between processes are established only if processes share a common mailbox
- A mailbox may be associated with many processes
- Each pair of processes may share several communication links that may be unidirectional or bi-directional

## More on mailboxes

- OS must provide mechanisms to:
  - create a new mailbox
  - send and receive messages through the mailbox
  - destroy the mailbox
- Initially, the creator of a mailbox is the only recipient
  - ownership/receiving privilege can be passed to other processes to share access
- Interface:
  - send(A, message) – send a message to mailbox A
  - receive(A, message) – receive a message from mailbox A

Jackson Dam (2619114D)

Message passing – synchronisation

- Blocking (synchronous)
  - blocking send → sender blocked until the message is received by the recipient
  - blocking receive → receiver blocked until the message is available

- Non-blocking (asynchronous)
  - non-blocking send → sender enqueues the message and continues operation
  - non-blocking receive → receiver receives either a valid message or a null

Message passing – buffering

- Exchanged messages go through a temporary queue (buffer)
- How large is this buffer?
- Zero capacity – 0 messages:
  - sender must wait for receiver
- Bounded capacity – finite length of n messages
  - sender must wait when link full
- Unbounded capacity – infinite length
  - sender never waits

Jackson Dam (2619114D)

# Topic 11 – Process Scheduling

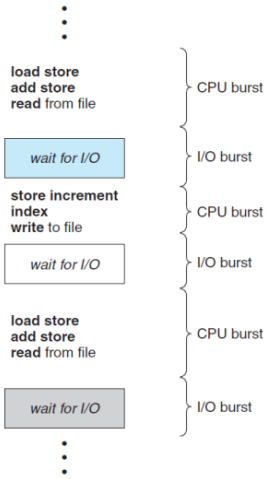
## Week 8 Lecture 1

### Key concepts

- Multiprogramming: run multiple programs/processes at a time
- Time-sharing: switch among (concurrently running) processes quickly enough to give the impression of parallel execution
- Cycle of a process: CPU burst → IO burst (alternates)

### Typical process lifecycle

- If a process mainly does CPU operations, it will be CPU-bound
- If a process mainly does I/O operations, it will be I/O bound




### Schedulers

- Long-term scheduler (or job scheduler)
  - loads programs into memory (i.e., turns a program into a process)
  - strives for good mix of IO-bound and CPU-bound processes
- Medium-term scheduler
  - freezes and unloads (swaps out) a process, to be reintroduced (swapped in) at a later stage
  - e.g., when there is not enough RAM for all processes
- Short-term scheduler (or CPU scheduler)
  - selects which process to execute next

### Process scheduling

- (CPU) scheduling's main priority is to maximise CPU utilisation
- Several queues of processes need to be maintained:
  - job queue (all processes in the system)
  - ready queue (processes in RAM and ready to execute)
  - device queues (processes waiting on some device; one such queue per device)
  - processes can migrate along the various queues

### CPU scheduler

- CPU scheduling decisions may take place when a process:
  1.) Switches from running to waiting state (e.g., IO request, sleep, etc.)
  2.) Switches from running to ready state (e.g., interrupt occurs)
  3.) Switches from waiting to ready (e.g., completion of IO)
  4.) Terminates

Jackson Dam (2619114D)

## More on CPU scheduling

- Non pre-emptive scheduling: Let the process give up (yield) the CPU (e.g., 1 and 4) (some definitions will only consider 4)
- Pre-emptive: the scheduler decides when a process yields the CPU (e.g., all of 1-4)

## Dispatcher

- Gives control of the CPU to the process selected by the short-term scheduler
- Needs to:
  - switch context (load registers, PCB, etc.)
  - switch to user mode
  - jump to the proper instruction in the user program to continue execution
- Dispatcher needs to do work to put a new process onto the CPU
  - this is known as dispatch latency / context switch time
  - context switch is pure overhead on top of required burst time

## Criteria for comparing CPU schedulers

- CPU utilization (more is better)
  - keep the CPU as busy as possible (40-90% is a good range)
- Throughput (more is better)
  - number of processes completed per time unit
- Waiting time (less is better)
  - total time spent in the READY queue (i.e., loaded but not having the CPU)
- Turnaround time (less is better)
  - time from submission of a process to its completion
- Response time (less is better)
  - time from submission of a process until first response produced

## CPU schedulers – FCFS/FIFO algorithm (first come first served / first in first out)

- Non pre-emptive
- Processes (or their PCBs) are added to a queue in order of arrival
- The algorithm always picks the first process in the queue and executes it to completion
- Very simple to implement
- Average waiting time can be quite high

Jackson Dam (2619114D)

CPU schedulers – SJF algorithm (shortest job first)

- Non pre-emptive
- Processes (or their PCBs) are added to a queue; queue is kept ordered by the CPU burst time
- The algorithm picks the first process in the queue and executes it to completion
- If multiple processes have the same CPU burst time, use FCFS among them to break the tie
- Fairly simple to implement
- Provable minimum average waiting time
- Unfortunately unrealistic to use in actual systems (as we don't normally know all the CPU burst times beforehand)

CPU schedulers – SRTF algorithm (shortest remaining time first)

- Pre-emptive version of SJF
- Processes (or their PCBs) are added to a queue; queue is kept ordered by the remaining CPU burst time
- Picks the first process in the queue and executes it
- Scheduler is also called when new processes arrive
- When a new process arrives, scheduler could pre-empt (temporarily stop) the current process and transfer control to a new one with a shorter remaining time
- If multiple processes have the same CPU burst time, use FCFS among them to break the tie

CPU schedulers – non pre-emptive priority algorithm

- Processes (or their PCBs) are added to a queue ordered by their priorities (usually: higher priority value = lower priority)
- If multiple processes have the same priority, use FCFS among them to break the tie
- Algorithm picks the first process in the queue and executes it to completion
- Fairly simple to implement
- SJF/SRTF are a special case of priority scheduling (priority value = (remaining) CPU burst time)
- Can lead to starvation of low-priority processes (can solve this with "aging")

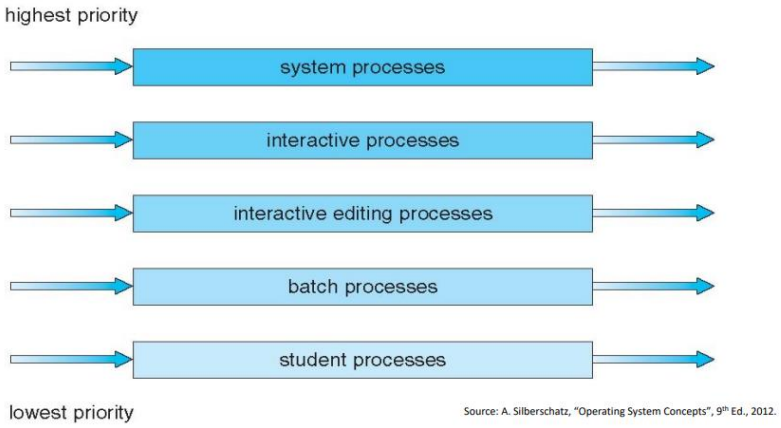
Jackson Dam (2619114D)

CPU schedulers – round-robin (RR) algorithm

- Pre-emptive version of FCFS
- Processes (or their PCBs) are added to a queue ordered by their time of arrival
- The algorithm picks the first process in the queue and executes it for up to a time interval (time quantum/time slice)
- If the process needs more time to complete, it is placed at the end of the queue (i.e., last)
- If the process ends before its time slice elapses, the algorithm moves to the next process in the queue

Multi-level queue scheduling

- OS may prioritise between multiple queues



Jackson Dam (2619114D)

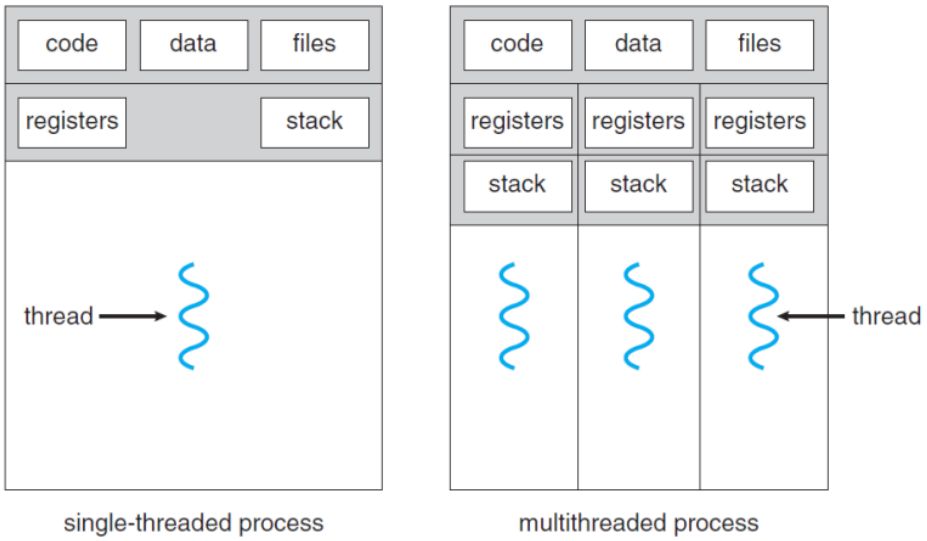
# Topic 12 – Process Synchronisation

## Week 8 Lecture 1

### Concurrency vs. parallelism

- Concurrency: multiple tasks make progress over time
  - time-sharing system, requiring scheduling and synchronization
- Parallelism: multiple tasks (threads/processes) execute at the same time
  - requires multiple execution units (CPUs, CPU cores, CPU vCores, etc.)

### Threads within processes



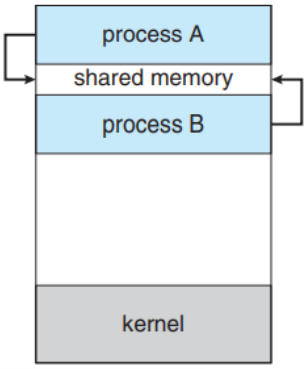

### Threads vs. processes

- Threads share memory and resources by default
  - processes require IPC mechanisms (shared memory, message passing) to be configured explicitly
- Threads are faster/more economical to create
  - each process has its own copy of the in-memory data, hence process creation means page table duplication (possibly also data duplication, if copy-on-write not used)
- In older operating systems, it was much faster to context switch between threads than processes
  - no longer the case, as the kernel keeps the same information for both, hence context switching has almost identical cost
- Can have different schedulers for processes and threads
  - depends on how threads are implemented (user threads vs kernel threads)
- If a single thread in a process crashes, the whole process crashes as well
  - if a process dies, other processes are unaffected

Jackson Dam (2619114D)

Two parallel processes – critical section, race condition (uncertain outcome) example

- Consider two processes $P_A$ and $P_B$, communicating via shared memory
- Assume shared memory consists of a single integer X = 20




$P_A$

```
...
my_X = shared_mem.X
if (my_X >= 10):
   my_X = my_X - 10
shared_mem.X = my_X
...
```

$P_B$

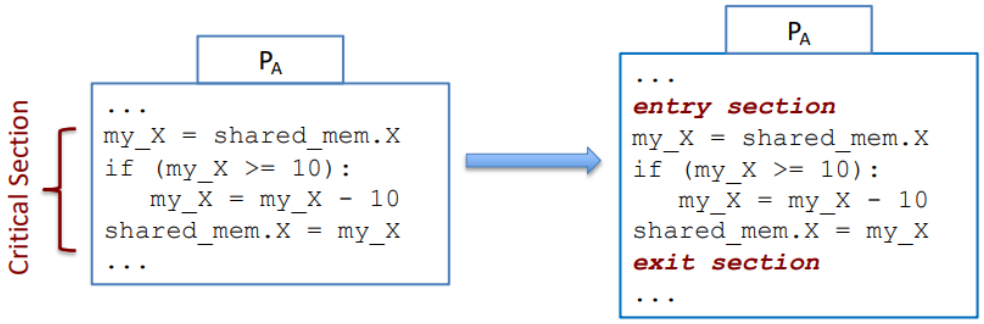
```
...
my_X = shared_mem.X
if (my_X >= 15)
   my_X = my_X - 15
shared_mem.X = my_X
...
```

A Race Condition!

- What is the final value of X?
- $P_A$ executes fully before $P_B$:
  - $P_A$'s check for X >= 10 succeeds ➔ X = 10; $P_B$'s: check for X >= 15 fails ➔ X = 10
- $P_B$ executes fully before $P_A$:
  - $P_B$'s check for X >= 15 succeeds ➔ X = 5; $P_A$'s check for X >= 10 fails ➔ X = 5
- But enter time-sharing/preemptive scheduling/parallel execution/...

- Also consider there could be partial execution (if X was a bank balance, -10 and -15 may both occur, so X would end up as -5 (overdrawn!))

- How would you alleviate this problem?
- Critical Section: A section of code where a process touches shared resources
  - Shared memory, common variables, shared database, shared file, ...
- Goal: No two processes in their critical section at the same time




The critical section problem

- All acceptable solutions need these properties:
  - mutual exclusion: no two processes in their critical section at the same time
  - progress: entering one's critical section should only be decided by its contenders in due time (assuming no process already in its critical section); a process cannot immediately re-enter its critical section if other processes are waiting for their turn
  - bounded waiting: it should be impossible for a process to wait indefinitely to enter its critical section, if other processes are allowed to

Jackson Dam (2619114D)

Peterson's solution

$P_A$

```
...
flag_A = True
turn = B
while flag_B == True and turn == B:
   pass
my_X = shared_mem.X
if (my_X >= 10):
   my_X = my_X - 10
shared_mem.X = my_X
flag_A = False
...
```

$P_B$

```
...
flag_B = True
turn = A
while flag_A == True and turn == A:
   pass
my_X = shared_mem.X
if (my_X >= 10):
   my_X = my_X - 10
shared_mem.X = my_X
flag_B = False
...
```

- Mutual Exclusion: If $P_A$ is in its critical section, then either `flag_B == False` (== $P_B$ is out of its critical section) or `turn == A` (== $P_B$ is waiting in the while loop)
- Progress: $P_A$ cannot immediately reenter its critical section, as `turn = B` means $P_B$ will be given the go next
- Bounded waiting: $P_A$ will wait at most one turn before it can enter its critical section again

- Is that good enough? Two problems with this solution:
  - busy waiting: the while loop eats up CPU cycles unnecessarily (a.k.a., spinlock)
  - doesn't account for instruction-level parallelism and memory access reordering (CPUs tend to reorder execution of memory accesses to avoid pipeline stalls)

- Atomic instructions at the hardware level are a better solution
  - "test and set"
  - "compare and swap"

*value* passed by reference

```
test_and_set(value):
   my_value = value
   value = True
   return my_value
```

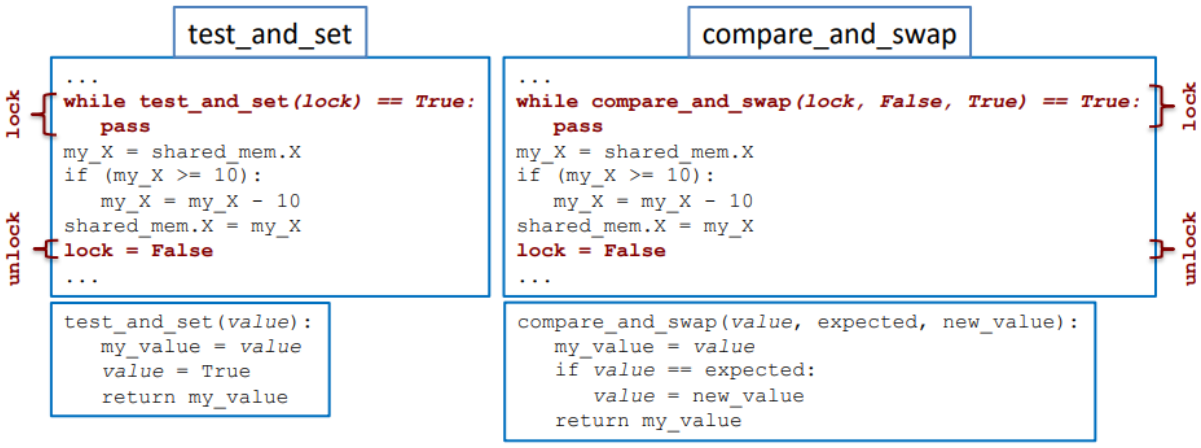
```
compare_and_swap(value, expected, new_value):
   my_value = value
   if value == expected:
      value = new_value
   return my_value
```

- Test and set assigns the value of the reference variable (value of value) to a new variable (my_value), sets the value of the actual reference variable to True, and returns the original value

- compare_and_swap assigns the value of the reference variable to a new variable, then checks if the value was as expected; if it was, then set it to the new value and return the old one

Jackson Dam (2619114D)

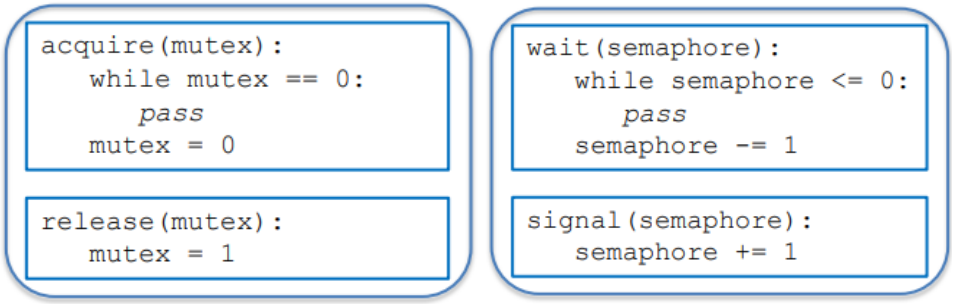
## Revised Peterson's solution




- Mutual Exclusion: Only one process will execute `test_and_set`/`compare_and_swap` with the lock value originally being `False`
- Progress/Bounded waiting: There's nothing stopping a process from immediately re-entering its critical section!

## Beyond test_and_set/compare_and_swap

- Test_and_set/compare_and_swap work, but are a bit clunky and a bit too low-level
- Enter mutex locks and semaphores:
  - internal state: a single integer value (mutexes can be 0 or 1, semaphores any value >= 0)
  - API offers two atomic functions:




- A mutex is a locking mechanism (only one process can acquire the mutex at once to lock access to a shared resource, and only the owner process that imposed the lock can unlock it)
- A semaphore is a signalling mechanism (when a process is done, it will subtract from the semaphore value)
- Can augment semaphores with a list of blocked processes each:
  - wait(semaphore) would instead add processes to said list if value <= 0
  - signal(semaphore) would instead remove one process from said list

Jackson Dam (2619114D)

Week 8 Lecture 2

## Atomic operations and spinlocking

- Atomic operations provide a good solution to inter-process/thread synchronization
- You do an atomic operation on a single processor by disabling interrupts while atomic function is running
- But in a multi-processing system, it is difficult to allow one processor to peacefully run an atomic function with no other touching the state
- So a bit of targeted spinlocking is okay!

## How to use semaphores

- Set semaphore value to number/size of shared resources -- i.e., number of acceptable concurrent users of the resource
  - 1 if only 1 user is allowed, N for an N-sized queue, etc.
- Decrease (wait) the semaphore every time a resource is used
- Increase (signal) the semaphore every time a resource is released

```
wait(semaphore):
    while semaphore <= 0:
        pass
    semaphore -= 1

signal(semaphore):
    semaphore += 1
```

## Bounded buffer problem – case study

- Also known as the producer-consumer problem
- Assume a list where items are placed by producers, and removed by consumers
- Assume we want our list to never contain more than N items
- Goal: Allow producers of items to add them to the list, but have them wait first if the list is full
- Goal: Allow consumers of items to remove an item from the list, but have them wait first if the list is empty

```
semaphore mutex = 1
semaphore empty = N
semaphore full  = 0
```

```
producer():
    while True:
        # Produce an item
        wait(empty)
        wait(mutex)
        # Add item to list
        signal(mutex)
        signal(full)
```

```
consumer():
    while True:
        wait(full)
        wait(mutex)
        # Remove an item from list
        signal(mutex)
        signal(empty)
        # Do stuff with item
```

Reading hint:

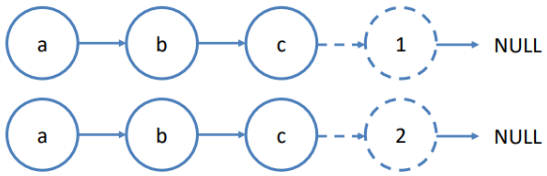
- `empty`: list is still a bit empty
- `full`: list is filled at least a bit
- Starts *fully* empty, can become *fully* full (`N` transferred between `empty` to `full`)

Jackson Dam (2619114D)

## Protecting lists

- Consider a singly linked list


- What happens if two processes simultaneously add an element?




## Readers-writers problem – case study

- Assume a variable shared among many processes, some of which only read its value (readers) while others also need to update it (writers)
- Goal: When a writer accesses the variable, no other process should be able to either read or update it
- Goal: When a reader accesses the variable, more readers can also access it, but writers should wait until no reader accesses it

```
semaphore rw_mutex = 1
semaphore mutex    = 1
int read_count     = 0
```

```
writer():
    while True:
        wait(rw_mutex)
        # Update the value
        signal(rw_mutex)
```

```
reader():
    while True:
        wait(mutex)
        read_count += 1
        if (read_count == 1)
            wait(rw_mutex)
        signal(mutex)
        # Read value
        wait(mutex)
        read_count -= 1
        if (read_count == 0)
            signal(rw_mutex)
        signal(mutex)
```

## Matters of life and death

- What would happen with writers if readers keep on arriving at the system?
  - writers would "starve"
- Starvation: Processes/threads unable to enter their critical section because of "greedy" contenders
  - think: trying to get on an extremely busy motorway with no one giving you some space
- Livelock: special case of starvation where competing parties both try to "avoid" each other at the same time
  - think: bumping into a person in a corridor, then both going left/right at the same time only to bump into each other again
- Priority inversion: special case where lower priority processes can keep higher priority processes waiting
  - think: having to sleep but being kept awake by social media notifications...

Jackson Dam (2619114D)

## Deadlock example

- What would happen in the producer/consumer problem if order of locking/unlocking mutex and empty/full was reversed?

```
semaphore mutex = 1
semaphore empty = N
semaphore full  = 0
```

```
producer():
    while True:
        # Produce an item
        wait(mutex)
        wait(empty)
        # Add item to list
        signal(full)
        signal(mutex)
```

```
consumer():
    while True:
        wait(mutex)
        wait(full)
        # Remove an item from list
        signal(empty)
        signal(mutex)
        # Do stuff with item
```

- Assume a consumer goes first…
  - `mutex` ➔ locked; consumer waiting on `full`
  - producer waiting on `mutex`
- Deadlock: all parties of a group are waiting indefinitely for another party (incl. themselves) to take action (but no one can)

## Beyond semaphores

- Semaphores/mutexes allow for mutual exclusion, but once a process is blocked that's it
- Enter monitors:
  - combination of semaphores and condition variables
  - each condition variable "associated" with a semaphore
  - allows for processes to have both mutual exclusion, and wait (block) on a condition
  - cond_wait(condvar, mutex): unlock the mutex to wait for a condition, then atomically
  - reacquire the mutex when condition is met
  - cond_signal(condvar): unblock one of the processes waiting on condition

```
semaphore mutex = 1
condvar   empty = N
condvar   full  = 0
list      items = 0
```

```
producer():
    while True:
        # Produce an item
        wait(mutex)
        while len(items) == N:
            cond_wait(empty, mutex)
        # Add item to list
        cond_signal(full)
        signal(mutex)
```

```
consumer():
    while True:
        wait(mutex)
        while len(items) == 0:
            cond_wait(full, mutex)
        # Remove an item from list
        cond_signal(empty)
        signal(mutex)
        # Do stuff with item
```

Jackson Dam (2619114D)

# Topic 13 – Memory Management

## Week 9 Lecture 1

### Memory management considerations

- Need to make memory accesses fast
- Need to safeguard memory allocated to different processes

### Caching

- Programs exhibit both spatial (data in relatively close memory locations due to loops in code, sequential access to data), and temporal (frequent reuse of data/instructions) locality
- Introduce two types of memory – primary (large, slow), and cache (fast to match CPU speed, but small)
- Keep recently accessed data/instructions in cache
- Can't fit all of RAM in cache - must make sure cache residency check is fast
- There is a high probability memory access will refer to data already in cache, so it will be fast

### More on caching

- Implementation issues:
  - amount of data in a cache location: cache line size
  - given an address, determine whether the data is in the cache and where
  - if not, determine where to put the data in the cache, after memory fetch
  - if cache is full, determine which lines to overwrite (what to evict from the cache): cache replacement policy

### Even more on caching

- Each location in the cache contains three parts:
  - a word of data
  - a valid bit (set if data is not empty, unset otherwise)
  - a tag: the actual address of this data in the primary memory
- On every memory access, the hardware checks the address against the tags in the cache in parallel
  - if the address refers to data in the cache, it is a cache hit, and the data is retrieved quickly, at processor speed
- Otherwise it's a cache miss
  - an access to primary memory is performed, and the result is placed in the cache for future reuse
  - the processor waits for the slow memory access to complete

Jackson Dam (2619114D)

Caching and memory stores

- Assume a store instruction changes only the cache entry, but not the respective memory location
  - then the cache and memory become inconsistent
  - a cache miss will require the cache entry to be written out later
- In write-through caches, a store changes both the cache entry and the memory
  - to reduce the cost of the memory operation, the data may be buffered in registers
- In write-back caches, a store changes only the cache entry
  - the cache is allowed to become inconsistent with the memory
  - a cache miss requires the replaced cache entry to be stored in RAM (if it has been modified)
  - write-back may reduce the number of memory stores, and improve performance

Cache lines

- In practice, we don't just keep individual memory locations (bytes or words) in the cache
- The cache is organised in "large words" called cache lines
  - 2^c words, at an address which is a multiple of 2^c
  - e.g. 16 bytes at an address which is a multiple of 16
  - If a is a multiple of 2c, a memory access to an address in [a, a + 2^c - 1] will actually refer to the same cache line
  - a cache miss for an address b will result in fetching all memory locations in [2c * floor(b, 2c), 2c * floor(b, 2c) + 2c - 1] and storing them in one cache line
- Different machines use different cache line sizes
  - bigger cache lines increase the probability of cache hits
  - but also increase the penalty of a cache miss

Searching the cache

- Given an address in the main memory address space, to locate where that address would be in the cache, the cache has to be searched on each memory access (needs to be fast!)
- Three main approaches:
  - direct-mapped caches: each address can be mapped to only one cache line
  - fully-associative caches: each address can be mapped to any cache line
  - n-way set-associative caches: each address can be mapped to any of a set of n cache lines
- A set-associative cache can subsume the other two
  - if only one line per set (n=1) → Direct mapped
  - if all cache lines in a single set → Fully associative

Jackson Dam (2619114D)

## Cache searching example – how to find offset, set ID, tag of a requested address

- Assume $m$-bit addresses, and a cache with $2^{\lambda}$ lines, each $2^{c}$ words wide
  - e.g., 8-bit addresses ($m = 8$), a cache with 8 lines ($\lambda = 3$) each being 4 words wide ($c = 2$)
- Each $m$-bit memory address is broken down into three parts:
  - Offset: the least significant $c$ bits, if the cache lines are $2^{c}$ words wide
    - Accesses to addresses with the same most significant $m - c$ bits, refer to words in the same line
    - $c = 2$
  - Set id: for a $2^{s}$-way set associative cache, the next most significant $\lambda - s$ bits (as the cache then contains $2^{\lambda}/2^{s} = 2^{\lambda - s}$ sets )
    - 1-way set associative cache (direct-mapped): $s = 0$ ➔ $\lambda - s = 3 - 0 = 3$
    - 2-way set associative cache: $s = 1$ ➔ $\lambda - s = 3 - 1 = 2$
    - 8-way set associative cache (fully associative): $s = 3 (= \lambda)$ ➔ $\lambda - s = 3 - 3 = 0$
  - Tag: the (remaining) most significant $m - c - (\lambda - s)$ bits
    - 1-way set associative cache: $m - c - (\lambda - s) = 8 - 2 - (3 - 0) = 3$
    - 2-way set associative cache: $m - c - (\lambda - s) = 8 - 2 - (3 - 1) = 4$
    - 8-way set associative cache: $m - c - (\lambda - s) = 8 - 2 - (3 - 3) = 6$
- On every memory access, the hardware:
  - extracts the tag and set id from the memory address
  - locates the set of cache lines for this address
  - checks the address tag against the tags in the cache line set's entries in parallel
  - if the tag matches that of an entry AND its valid bit is set, it is a cache hit; otherwise, it is a cache miss

## Accessing set associative cache example

- Assume memory addresses are 8 bits wide ([b7,b6,...,b0], b7: MSB, b0: LSB)
- Assume we have a cache with 8 lines, each 4 words wide
  - 4 words per line ➔ 2-bit offset
- Assume we have a 2-way associative cache (i.e., 2 cache lines per set)
  - 8/2 = 4 sets ➔ 2-bit set id
  - 8-bit addresses ➔ 8 - 2 - 2 = 4-bit tag
- [b7,b6,b5,b4,b3,b2,b1,b0]
- Access memory position 138
  - 138 = 10001010
- Access memory position 41
  - 41 = 00101001
- Access memory position 43
  - 43 = 00101011
- Access memory position 136
  - 136 = 10001000

| line | set | tag | valid | data |
|---|---|---|---|---|
| (0) 000 | 00 | | 0 | |
| (1) 001 | 00 | | 0 | |
| (2) 010 | 01 | | 0 | |
| (3) 011 | 01 | | 0 | |
| (4) 100 | 10 | 1000 | 1 | {11,10,01,00} |
| (5) 101 | 10 | 0010 | 1 | {11,10,01,00} |
| (6) 110 | 11 | | 0 | |
| (7) 111 | 11 | | 0 | |

Jackson Dam (2619114D)

Explanation of example above

- When accessing 138, the first line with that set ID (10) is EMPTY initially

- As there are no lines with the set ID that are both valid and have the same 4-bit tag...

- This is a CACHE MISS (red lightning bolt in example) - so we store the tag, set the valid bit to 1, and then load in all 4 words (8 bits) into the line

- Then, when accessing 41, there is also a CACHE MISS (no valid lines in set ID 10 with 4-bit tag 0010), so we store the tag, set valid bit to 1, and store the data

- When accessing 43, there is a valid line in the set ID 10 with tag 0010 (CACHE HIT, represented by sunshine in example), so we can simply access the correct word using the offset

- When accessing 136, there is a valid line in the set ID 10 with tag 1000 (CACHE HIT), so we can simply access the correct word using the offset

Memory address space

- Each process is assumed to have its own address space
- A region of memory locations usually split into two parts:
    - program segment (read-only, contains code that may be shared across multiple processes)
    - data segment (contains variables/data, usually read-write, should only be accessible by its owner process)
- Lowest address in an address space is known as the origin

More on memory address space

- When multiple processes are running, their code is loaded at an arbitrary location in memory, so what happens to all the addresses in the code?
- Could have loader re-relocate all of the addresses after program loaded in RAM
    - would take more time to load the program
    - would need to be repeated if process is swapped out/in

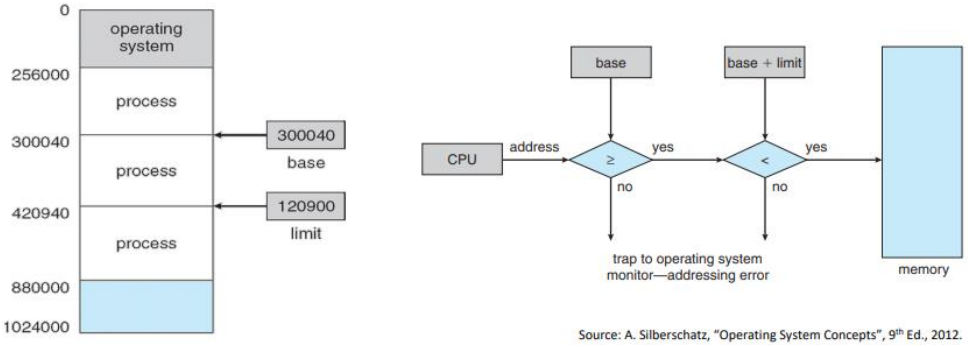
Jackson Dam (2619114D)

Memory protection

- We still have not solved the memory protection problem (program could access memory locations beyond its own)

Idea #1 – base register and limit register

- Use special registers to store the first and last address in a process's address space (base register (BR) and limit register (LR))
- Add BR's value to the memory addresses of all memory accessing instructions
- Check that the resulting memory address is not beyond LR
- Only the OS can set/update the values in BR and LR
- This is a virtual address space!




Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.

Idea #2 – RAM partitioning

- Partition RAM into fixed-size partitions and allocate one to each running process
  - works, but is inflexible and clunky
  - creates internal fragmentation (unused space within partitions)

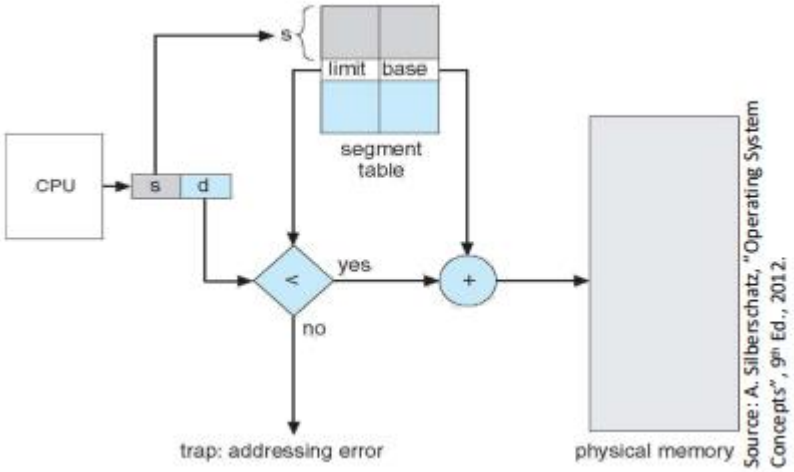
Idea #3 – variable sized partitions

- The OS keeps lists of allocated and unallocated ranges in RAM
- When a new process arrives, it blocks until a large enough unallocated range is found
- Several strategies available for this:
  - first fit: use the first unallocated range that is large enough for the process
  - best fit: use the smallest unallocated range
  - worse fit: use the largest unallocated range
- If the unallocated range is too large, it's split into two parts:
  - one allocated to the new process
  - add the remainder to the list of unallocated ranges
- Two or more adjacent unallocated ranges may be merged
- The OS can then check whether the merged range is large enough for any waiting processes
- Can create external fragmentation (space in between partitions too small to be used)

Jackson Dam (2619114D)

Idea #4 – segments

- Ideas 1-3 assume a process's memory address space must occupy a big contiguous space in RAM
- This is inflexible – so we can use segments to allow non-contiguous allocation of memory locations
    - maintain several "specialised" segments
    - maintain a table with info for each segment (starting address of segment (base), and size of the segment (limit))
- An extension of BR/LR but to multiple mini address spaces (segments)
- But how do we map these segments to unallocated ranges in RAM?
- See idea #5 in next topic (paging)

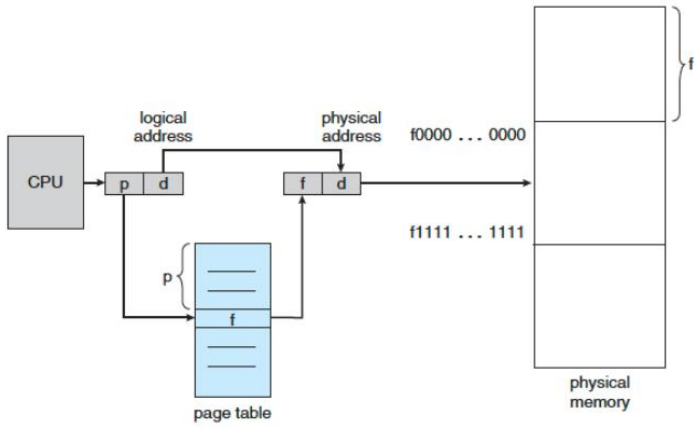


Jackson Dam (2619114D)

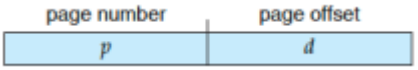
Topic 14 – Paging

Week 9 Lecture 2

Idea #5 – paging




- Partition address space in equally sized, fixed-sized partitions (pages)
  - size always a power of 2
  - typical page sizes are 1KB – 4KB (could be larger on modern OS)
  - each page is kept on disk (so there can be a lot of them)
- A location in the address space can be given as either an address, or a page number + an offset within set page
- Given an address with n bits:
  - the least significant d bits are the offset
  - the most significant p (= n – d) bits define the page number

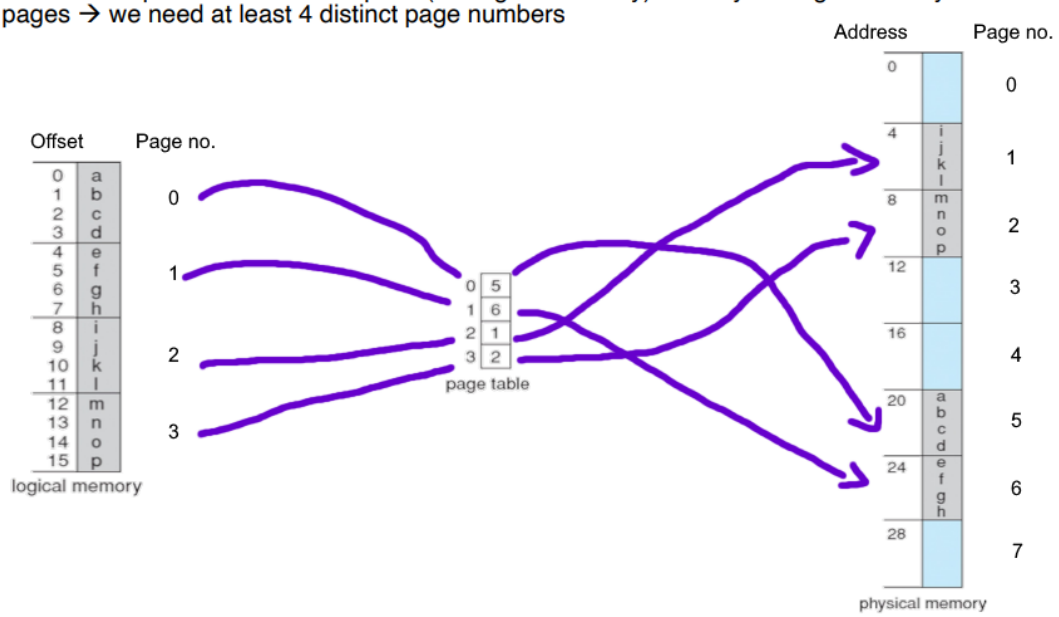



- Partition a large portion of RAM into page frames, each equal size of a page
- Maintain a page table for each address space; each entry contains:
  - a resident/valid bit (1 if page is loaded into a page frame)
  - a frame address (contains the physical address of the first location in the frame, if the resident bit for that entry is set to 1)
  - can be per process or contain additional data (e.g. process ID) for protection
- Paging means moving a page/frame of data from disk to memory or vice-versa

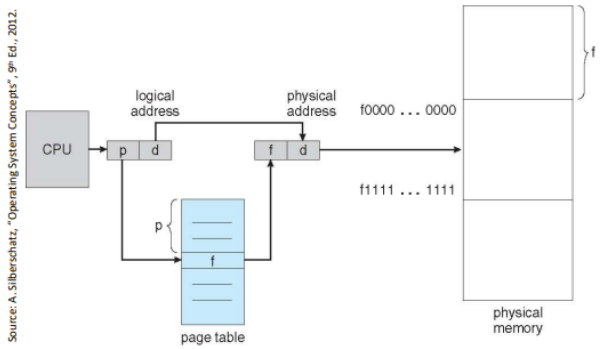
Jackson Dam (2619114D)

## Paging example

- Assume we have a main memory of 32 bytes and 4-byte frames → we need 8 frames and an offset with 2 bits (d = 2)
- Assume our process's address space (i.e. logical memory) is 16 bytes large and 4-byte pages → we need at least 4 distinct page numbers




## Memory address translation




Wait... What if **p** and **f** were a different size?

Should then **p** be larger or smaller than **f**? ⟹ p<=f

- You must have at least as many frames as pages, since each page table translation must resolve to a unique page

## Allocation of frames

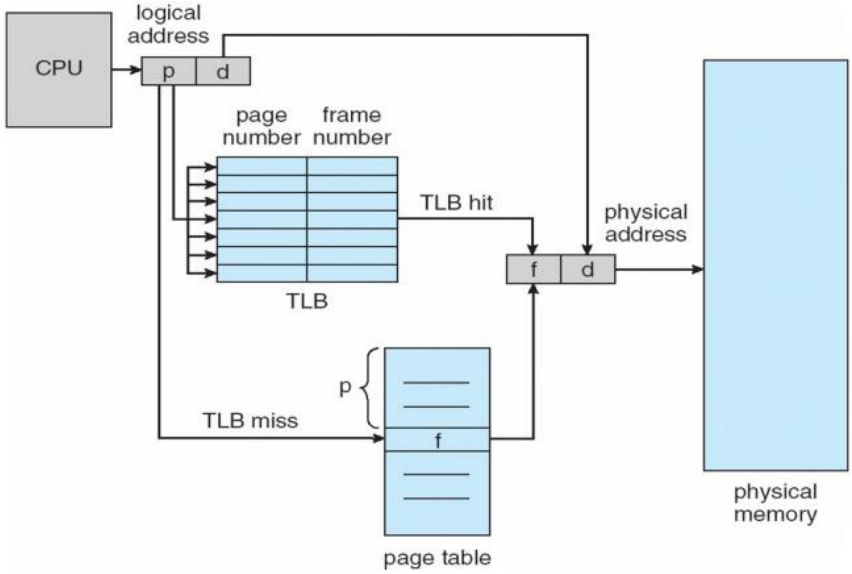
- Each process needs a minimum number of frames
- The maximum for a process is total system frames minus OS-allocated frames
- Three major allocation schemes:
  - fixed allocation (divide available frames equally among processes)
  - proportional allocation (give each process a % of frames equal to its size divided by the size of all processes
  - priority allocation (like proportional allocation, but considering process priority, possibly with size as well)
- Memory access speed can vary by CPU (e.g. multi-processor NUMA (non-uniform memory access) architectures will show faster access speeds if you allocate memory "closer" to the CPU where the process that caused the page fault is running)

Jackson Dam (2619114D)

Implementation considerations

- The page table is an OS construct with hardware assistance
- The page table is kept in main memory
    - page table base register (PTBR) points to beginning of page table location
    - page table length register (PTLR) indicates size of page table
- But then every data/instruction access requires two memory accesses
    - one for the page table entry and one for the actual data/instruction
- So to get acceptable performance, we must use caching
- Use an on-CPU hardware cache for the page table → translation lookaside buffer (TLB)

Translation lookaside buffer (TLB) diagram




Shared pages

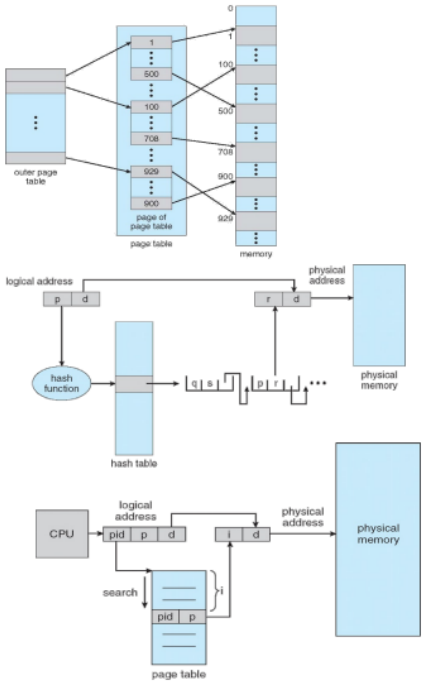
- Pages can be shared across processes
    - often done for pages containing code
    - one copy of read-only code shared among processes
- When forking a process, it can be expensive to create a full copy of the parent process's address space
- So we can make it faster with copy on write (CoW)
    - both processes share the same pages in memory
    - only when attempting to write to (modify) a page do we make a private copy

Jackson Dam (2619114D)

## Partitioning the page table

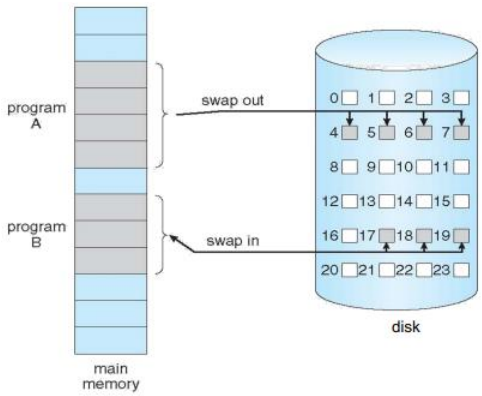
- If the page table doesn't fit in RAM, we need to partition it

- **Idea #1: Hierarchical page tables**
  - 2-level PTs quite common
- **Idea #2: Hashed page tables**
  - Hash table of linked lists of mappings
  - Can contain multiple mappings per entry (clustered PTs)
- **Idea #3: Inverted page table**
  - Maintain mappings of each physical frame to virtual addresses
  - Mapping must also contain owner process id




## More on paging

- Paging is moving a page/frame of data from disk to RAM or vice-versa
- Pages all start out on disk
- We move a page from disk to memory when it is needed/accessed
- This is demand paging (i.e. using a "lazy" pager)




## Demand paging

- If a process tries to access a page that isn't loaded in RAM, a page fault occurs
- The OS's pager (or page fault handler) does the following:
  1. Trap to OS (original process is blocked à context switch)
  2. Kernel computes page location on disk
  3. Kernel issues a disk read to load contents of the page to a free frame (in RAM)
  4. Jump to the process scheduler (other processes execute)
  5. Interrupt from the disk (I/O completed)
  6. Context switch & control passed to the OS / pager again
  7. Pager updates the page table (frame address, resident bit)
  8. Original process moved to the READY queue
  9. Pager returns to the process scheduler (which selects some process)

Jackson Dam (2619114D)

More on demand paging

- Virtual memory can be almost as fast as real memory if the proportion of instructions causing a page fault is low enough
- Programs tend to perform almost all memory accesses close to where they recently accessed data
- Nearly all accesses are to a working set which is kept in page frames
- Page faults occur when a program changes its working set (e.g. a method finishes and calls another)
- Occasionally, a program may cause too many page faults (this is called thrashing, and can cause execution speed to slow down by several '000 times)
- Effective Access Time (EAT) = (1 – p) * hit memory access time + p * (page fault overhead + swap page out + swap page in + restart overhead), where p is the page fault rate between 0 and 1

Page replacement

- Normally all or most of the page frames in RAM are in use
- In this case, when a page fault occurs, you cannot simply read the page into an empty frame
- The system must:
    - select a full frame (via page replacement strategy/algorithm)
    - write its contents to disk (page-out)
    - read the required page into this frame (page-in)
- In practice, it is better to keep several empty frames
- On a page fault, the read (to load the data) can start immediately
- A separate write (to clear another frame) can be done right after that (or lazily even later)
- Can further optimise page-outs by introducing "dirty/modified" bit
- Set to 1 if page has been updated à only write page to disk if true

Jackson Dam (2619114D)

Week 10 Lecture 1

## More on page replacement

- With page replacement, larger virtual memory can be provided on a smaller physical memory
- Page replacement requires hardware...
- Circuitry in the CPU must:
  - extract the page number
  - look up the page table entry
  - check for residency
  - if resident, find the real memory address
  - if not resident, generate an interrupt (page fault) (has to be done in hardware, for efficiency (as it happens often))
- also requires software...
- In the event of a page fault, the pager must:
  - perform the disk I/O
  - maintain the page table
  - implement the page replacement policy
  - has to be done in software, because of complexity and flexibility

## Even more on page replacement

- If a page is written from a frame to disk, and data on this page is required again, it will have to be read back in
  - this can result in too much disk I/O (slow)
- Therefore the pager does not randomly choose a frame to clear
  - it uses a page replacement policy
  - tries to choose a page that will probably not be needed again soon
- We will use page replacement algorithms, but there are alternatives:
  - global replacement (consider all non-OS frames)
  - local replacement (only consider frames allocated to the process that caused the page fault)

Jackson Dam (2619114D)

Page replacement algorithms – optimal (OPT or MIN)

- Page out the page that will not be used for the longest period of time (farthest access into the future)
    - unrealistic in practise as we don't know about future accesses
- Access string: B, C, A, B, B, D, A, C, D, B
- '*' = page fault

| B | C | A | B | B | D | A | C | D | B |
|---|---|---|---|---|---|---|---|---|---|
| B | B | B | B | B | D | D | D | D | D |
|  | C | C | C | C | C | C | C | C | B |
|  |  | A | A | A | A | A | A | A | A |
| * | * | * |  |  | * |  |  |  | * |

Frames

- C is paged out for the final access (B) because it is the furthest *back* rather than the furthest ahead (we go back because we are at the end of the access string)

Page replacement algorithms – random

- Page out a random page
- Generally lower page faults than FIFO but worse than LRU in practice
- Access string: B, C, A, B, B, D, A, C, D, B
- '*' page fault

| B | C | A | B | B | D | A | C | D | B |
|---|---|---|---|---|---|---|---|---|---|
| B | B | B | B | B | B | B | C | C | C |
|  | C | C | C | C | D | D | D | D | B |
|  |  | A | A | A | A | A | A | A | A |
| * | * | * |  |  | * |  | * |  | * |

Frames

Jackson Dam (2619114D)

Page replacement algorithms – first in first out (FIFO)

- Store the page-in time of every page and page out the oldest one each time
- i.e. add loaded pages to the tail of a queue, and page out the head of it

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining page-in time for every page
- '*' = page fault

| B | C | A | B | B | D | A | C | D | B |
|---|---|---|---|---|---|---|---|---|---|
| B (0) | B (0) | B (0) | B (0) | B (0) | D (5) | D (5) | D (5) | D (5) | D (5) |
| | C (1) | C (1) | C (1) | C (1) | C (1) | C (1) | C (1) | C (1) | B (9) |
| | | A (2) | A (2) | A (2) | A (2) | A (2) | A (2) | A (2) | A (2) |
| * | * | * | | | * | | | | * |

Frames

Page replacement algorithms – least recently used (LRU)

- Store the most recent time of use for each page – choose the page that has not been used for the longest
- i.e. move accessed pages to the tail of queue, and page out the head of it

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining last access time for every page
- '*' = page fault

| B | C | A | B | B | D | A | C | D | B |
|---|---|---|---|---|---|---|---|---|---|
| B (0) | B (0) | B (0) | B (3) | B (4) | B (4) | B (4) | C (7) | C (7) | C (7) |
| | C (1) | C (1) | C (1) | C (1) | D (5) | D (5) | D (5) | D (8) | D (8) |
| | | A (2) | A (2) | A (2) | A (2) | A (6) | A (6) | A (6) | B (9) |
| * | * | * | | | * | | * | | * |

Frames

Jackson Dam (2619114D)

Page replacement algorithms – least frequently / non frequently used (LFU/NFU)

- Maintain counter of accesses to each page; choose the page with the lowest count to page-out
- To resolve ties, remove the page that was paged-in the longest ago

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining access counts for every page
- '*' = page fault

| B | C | A | B | B | D | A | C | D | B |
|---|---|---|---|---|---|---|---|---|---|
| B (1) | B (1) | B (1) | B (2) | B (3) | B (3) | B (3) | B (3) | B (3) | B (4) |
| | C (1) | C (1) | C (1) | C (1) | D (1) | D (1) | C (1) | D (1) | D (1) |
| | | A (1) | A (1) | A (1) | A (1) | A (2) | A (2) | A (2) | A (2) |
| * | * | * | | | * | | * | * | |

Frames

Page replacement algorithms – simple reference bit based

- Maintain an "accessed bit" per page (initially 0); resetting after two frame accesses (unless accessed again); choose oldest page with a 0 bit

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining reference bit for every page, resetting after two accesses
- '*' = page fault

| B | C | A | B | B | D | A | C | D | B |
|---|---|---|---|---|---|---|---|---|---|
| B (1) | B (1) | B (0) | B (1) | B (1) | B (1) | B (0) | C (1) | C (1) | C (0) |
| | C (1) | C (1) | C (0) | C (0) | D (1) | D (1) | D (0) | D (1) | D (1) |
| | | A (1) | A (1) | A (0) | A (0) | A (1) | A (1) | A (0) | B (1) |
| * | * | * | | | * | | * | | * |

Frames

Jackson Dam (2619114D)

Page replacement algorithms – aging / additional reference bits

- Maintain a multi-bit word per page; set MSB (most significant / left-most bit) to 1 on access;
- Shift all bits right with every access; choose page with lowest value to page-out

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining 3 bits for every page, shifting on every access
- '*' = page fault

| B | C | A | B | B | D | A | C | D | B |
|---|---|---|---|---|---|---|---|---|---|
| B (100) | B (010) | B (001) | B (100) | B (110) | B (011) | B (001) | C (100) | C (010) | C (001) |
| | C (100) | C (010) | C (001) | C (000) | D (100) | D (010) | D (001) | D (100) | D (010) |
| | | A (100) | A (010) | A (001) | A (000) | A (100) | A (010) | A (001) | B (100) |
| * | * | * | | | * | | * | | * |

Frames

Page replacement algorithms – second chance (clock)

- Select oldest page with FIFO (lowest page in time bit)
- If it has an access bit of 0, page it out
- Otherwise, if it has an access bit of 1:
  - still page it out if all other pages have an access bit of 1
  - otherwise, give it a new page in time (the current time), and page out the next oldest page with an access bit of 0 instead
  - set all surviving pages' access bits to 0 on each access that causes a paging out

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining page-in time and access bit for every page → (page_in, access)
- '*' = page fault

| B | C | A | B | B | D | A | C | D | B |
|---|---|---|---|---|---|---|---|---|---|
| B (0,1) | B (0,1) | B (0,1) | B (0,1) | B (0,1) | D (5,1) | D (5,1) | D (5,1) | D (5,1) | D (5,0) |
| | C (1,1) | C (1,1) | C (1,1) | C (1,1) | C (1,0) | C (1,0) | C (1,1) | C (1,1) | B (9,1) |
| | | A (2,1) | A (2,1) | A (2,1) | A (2,0) | A (2,1) | A (2,1) | A (2,1) | A (2,0) |
| * | * | * | | | * | | | | * |

Frames

Jackson Dam (2619114D)

## Page replacement algorithms – not recently used (NRU)

- Have page in, accessed, modified bits (3 bits per page)
- Use the last 2 bits (accessed and modified) to make a score:
  - (0,0) = 0: not accessed, not modified (best candidate) → if found, page out
  - (0,1) = 1: not recently used but modified (will need write out to disk) → write-out, clear modified bit, continue the search
  - (1,0) = 2: recently used but not modified (better keep it as might be used again soon) → clear accessed bit, continue the search
  - (1,1) = 3: accessed and modified (worst candidate) → clear accessed bit, continue the search
- Go through all pages to find the oldest page with the lowest score, performing the actions above if not paged out until a page out candidate is found
- May require multiple passes – by the 3rd pass, all pages will have (0, 0)

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining page-in time, and access and modify bit for every page → (page_in, access, modify)
- Assume underlined accesses are modifications
- '*' = page fault

| B | C | A | B | B | D | A | C | D | B |
|---|---|---|---|---|---|---|---|---|---|
| B (0,1,0) | B (0,1,0) | B (0,1,0) | B (0,1,1) | B (0,1,1) | B (0,0,1) | B (0,0,1) | C (7,1,0) | C (7,1,0) | C (7,0,0) |
| | C (1,1,0) | C (1,1,0) | C (1,1,0) | C (1,1,0) | D (5,1,0) | D (5,1,0) | D (5,0,0) | D (5,1,1) | D (5,0,1) |
| | | A (2,1,0) | A (2,1,0) | A (2,1,0) | A (2,0,0) | A (2,1,1) | A (2,0,1) | A (2,0,1) | B (9,1,0) |
| * | * | * | | | * | | * | | * |

In order of current score (and oldest, for same scores), until a frame is paged out:
(X,0,0) = 0 → page out!
(X,0,1) = 1 → (X,0,0) clear modified bit
(X,1,0) = 2 → (X,0,0) clear accessed bit
(X,1,1) = 3 → (X,0,1) clear accessed bit

Jackson Dam (2619114D)

Topic 15 – File Storage

Week 10 Lecture 2

Disk anatomy and I/O access

- Recipe for a mechanical hard disk drive:
  - take a (magnetic) disk
  - dig concentric tracks
  - split tracks into circular sectors (~512B)
  - logically group up consecutive sectors into fragments (~1KB)
  - logically group up consecutive fragments into blocks (~1-10KBs)
  - organise each file in blocks
  - store each file block in a disk block

- Read/Write Access (I/O)
  - position r/w head (seek delay: ~3-6ms)
  - spin until selected sector under the head (rotation delay: ~3-5ms)
  - transfer data from track to disk buffer (extremely fast)
  - transfer data from disk buffer to RAM via bus (using DMA - also fast)

Disk anatomy – formatting/partitioning

- Disk platters are originally empty
  - disk needs to be low-level formatted to create sectors
  - usually done at the factory
  - sectors usually also contain disk-related metadata (e.g., error correcting code, sector id/no, etc.) → allows for detection of bad blocks
- Disks can also be partitioned to appear as several different "disks"
  - usually on a cylinder boundary
  - i.e., cylinders 0 - X → partition 1, X+1 – Y → partition 2, etc.
  - nothing physically changes on the disk – this is an OS abstraction
- Last, each partition needs to be initialized
  - logical formatting or file system formatting
  - computes and stores file-system specific metadata in blocks
  - creates root directory
  - initialises list of free/allocated blocks in the partition

Jackson Dam (2619114D)

Disk anatomy – abstractions

- For simplicity, consider the disk as a long, flat array of disk blocks/clusters
  - all blocks have the same size (although some modern disks can be low-level formatted to have different block sizes)
  - note: each block is made up of multiple (fixed number) fragments
  - note: each fragment is made up of multiple (fixed number) sectors
  - note: fragments have addresses, from 0 to size of disk (in fragments); blocks are addressed with the address of their 1st fragment
- Comparison to RAM:
  - sector → memory bit; fragment → memory byte; block → memory word
- Not all sectors will have the same natural performance characteristics (e.g. sectors at the outer part of the disk vs. the centre)

Disk anatomy – scheduling

- Use scheduling to provide a (semi-)constant data rate
- Typical disk may have many I/O requests waiting to be served
- The average disk I/O operation takes several milliseconds
- Several processes may have executed in that time
- Many of them may have issued I/O requests, so we need schedulers to deal with this queue

Disk anatomy – schedulers

- FCFS
- SSTF (shortest seek time first) – choose request with least seek distance from the current head position
- SCAN/LOOK (aka the elevator algorithm) – choose next request in direction the head is currently travelling
- SCAN = until end of disk
- LOOK = if no more requests in current direction, reverse motion and serve requests in opposite direction
- C-SCAN/C-LOOK = same as SCAN/LOOK, but reset to centre of disk rather than reversing direction

SSDs just use FCFS (usually) – access latency is pretty much the same for all sectors

Jackson Dam (2619114D)

## Filesystems

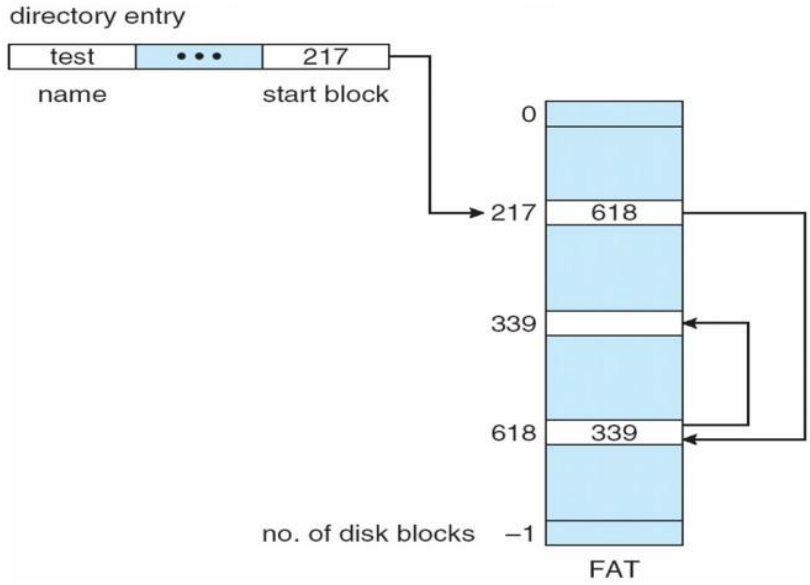
- Platters/tracks/sectors... too complicated for us to keep track of
- We use a filesystem to provide:
  - ability to organise data in files
  - impression of contiguous logical address space for each file
  - high-level API that builds on this abstraction
  - ability to add/delete/update data in a file
  - ability to organise/access files through an intuitive interface
  - access control/security
- Filesystem layers
  - logical file system: provides naming and logical organisation (files, directories, access rights)
  - presents user-facing API and keeps track of per-process open files
  - passes all I/O operations to the layer below
  - virtual file system (optional): presents a unified view of potentially multiple physical layer implementations below to the logical layer above
  - physical file system: handles disk blocks, reading/writing blocks, buffering and memory management
  - interfaces with the device driver/hardware

## Physical filesystem

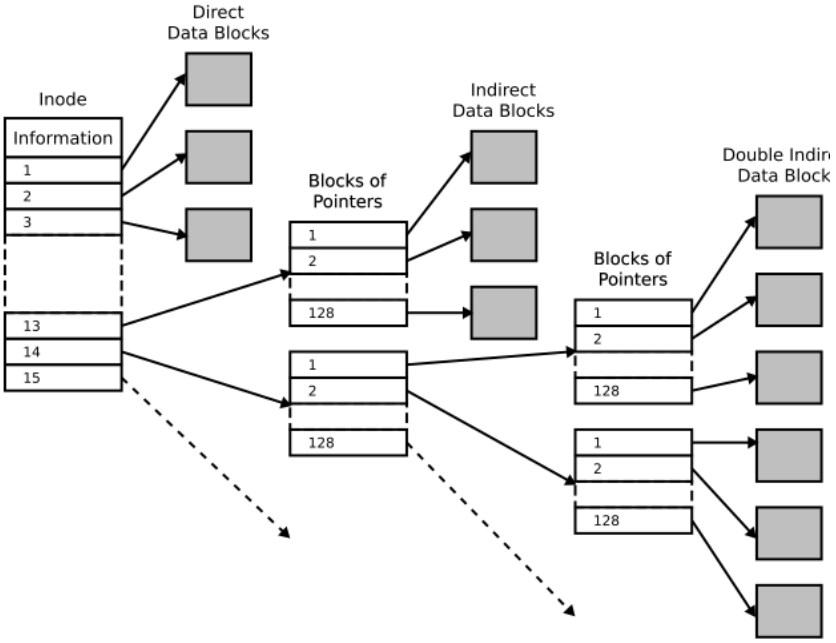
- A file is a set of blocks of data (blocks are data agnostic – contents doesn't matter)
- Allocation methods for files:
  - contiguous (neighbouring) allocation: a file contains spatially consecutive blocks (aka self-navigation file)
  - linked allocation: each block i has a pointer to the logically next block i+1 anywhere on the disk (i.e. a linked list of blocks, navigation information stored in each block of the file)
  - indexed allocation: there exists an index block (can be anywhere on the disk), that maintains a list of pointers pointing to the physical address of each block, navigation information is stored in the index block
- Contiguous = good for sequential and random I/O
- Linked = fairly good for sequential, bad for random
- Indexed = as good as linked for sequential, good as contiguous for random

Jackson Dam (2619114D)

FAT32 linked allocation diagram




Linux/Unix/*nix i-node system diagram (more flexible on where you can store files, no need to be contiguous)




Free space management

- File system maintains list of unallocated blocks
  - bit vector or bit map (1 bit per block, 0/1 = (un)allocated)
  - list of ranges of empty blocks
- List of pairs containing address of first empty block in a range, plus count of empty blocks after it
  - note: "list" not referring to the list data structure -- these "lists" are often loaded into balanced tree data structures at runtime
  - note: doesn't have to be across the full disk

Jackson Dam (2619114D)

Disk I/O problems

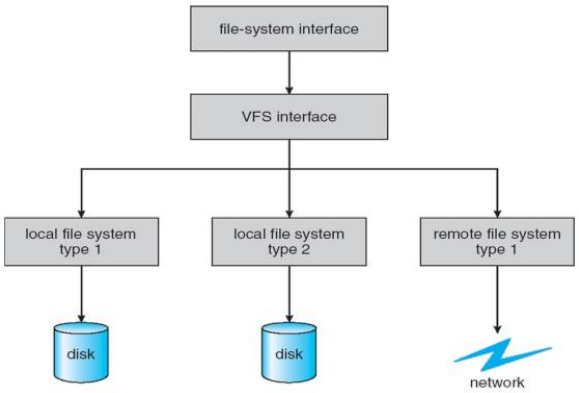
- Sequential I/O is several orders of magnitude slower than RAM
- Random I/O is even slower (often by another several orders of magnitude, compared to sequential I/O)
- To improve performance:
  - reduce preliminary seeks by keeping data and metadata "close" together
  - reduce data access seeks (e.g., journaling, log-structured merge trees (LSM), etc. → turn random I/O to sequential I/O plus indexing
  - allow space for files to grow → alleviate fragmentation
  - caching (buffering)
  - read-ahead (fetch and cache blocks before you need them)
  - free-behind (remove a block from cache as soon as process moved on to next block)

Access types

- Sequential vs random:
  - sequential: start from point A in file, read/write all data until point B
  - random: read/write from/to random positions in the file – i.e., seek to a position, read/write, seek to another position, read/write, etc.
  - could be a mix of both; seek to a position, then read several bytes sequentially, etc
- Buffered vs unbuffered:
  - unbuffered: read/write bytes as needed
  - buffered: use part of memory (buffer) to store data coming from/going to the disk
  - read from the disk more than needed and store in buffer – subsequent reads do not need to go to disk, until the buffer is exhausted
  - for writes, store data in the buffer and write it out to disk when the buffer is full or after some time
  - result: disk I/O is amortised (performance increases) across time and requests

Jackson Dam (2619114D)

Virtual filesystem




- VFS interface makes different types of file system accessible under the one actual interface

Logical filesystem

- File structure represented via File Control Blocks
- Each FCB (one per file) contains:
  - access control (owner/group/etc IDs, access rights descriptors), r = readable, w = writable, x = executable
    - *nix: 9 bits -- owner R/W/X, group R/W/X, others R/W/X
    - E.g., `rw-r--r--` (=0644 (octal)); `rwx------` (=0700)
  - dates (creation, last access, last modified, etc)
  - size of file (in bytes)
  - location and organisation of file's blocks (i-node for *nix, location of first block for FAT, etc)

More on logical filesystem

- Several pieces of data are needed to manage open files
- Open-file table (aka the file descriptor table – fdtable)
  - contains FCBs of open files, and tracks open files
  - one global OFT/fdtable and one per process
  - per-process fdtables contain pointers to global fdtable entries
- File pointer (pointer to last read/written location in file), one per file and one per process that has the file open (part of process fdtable)
- File-open count (counter of number of times a file is open), per-file (global fdtable), allows removal of global fdtable entry when last process closes it

Logical filesystem – directory structure

- A directory is just a file
- Contains pairs of type {file name, FCB block location}
- Entries can point to files or other directories etc
- Hard links: entries can be shared across directories
- Soft/symbolic links: special files that contain a relative pointer/path to another file