

What is the Internet?

- A massive network of networks.
- A computer network that **interconnects** billions of computing devices throughout the world.
- Traditional devices – PCs, Workstations, Servers – web pages, emails, etc.
- Internet “things” – laptops, PDAs, TVs, gaming consoles, home security systems, home appliances, watches, cars, traffic control systems, etc.,

COMPUTER NETWORKS

The Internet: A “Nuts and Bolts” View



Billions of connected computing **devices**:

- **hosts** = *end systems*
- running **network apps** at Internet's "edge"



Packet switches: forward packets (chunks of data)

- *routers, switches*



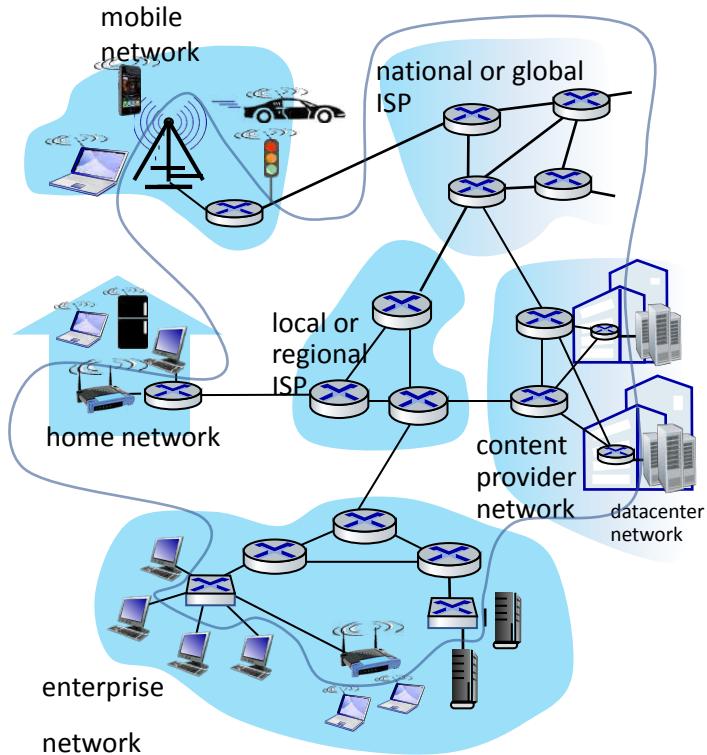
Communication links

- fiber, copper, radio, satellite
- transmission rate: *bandwidth*



Networks

- collection of devices, routers, links: managed by an organization



COMPUTER NETWORKS

The Internet: A “Nuts and Bolts” View

- **Internet:** “network of networks”

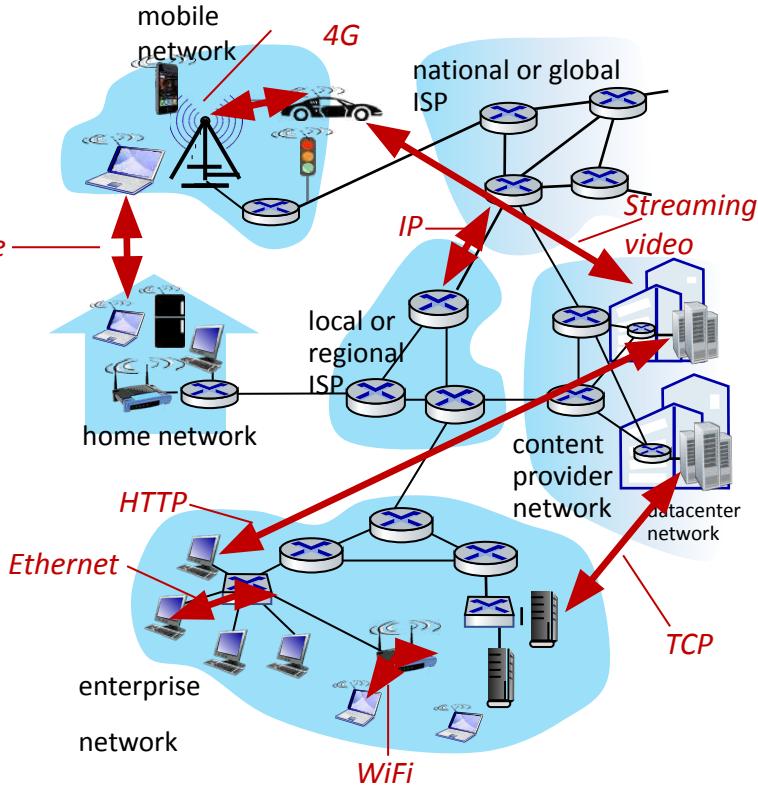
- Interconnected ISPs

- **Protocols** are everywhere

- control sending, receiving of messages
- e.g., HTTP (Web), streaming video, Skype, TCP, IP, WiFi, 4G, Ethernet

- **Internet standards**

- RFC: Request for Comments
- IETF: Internet Engineering Task Force



COMPUTER NETWORKS

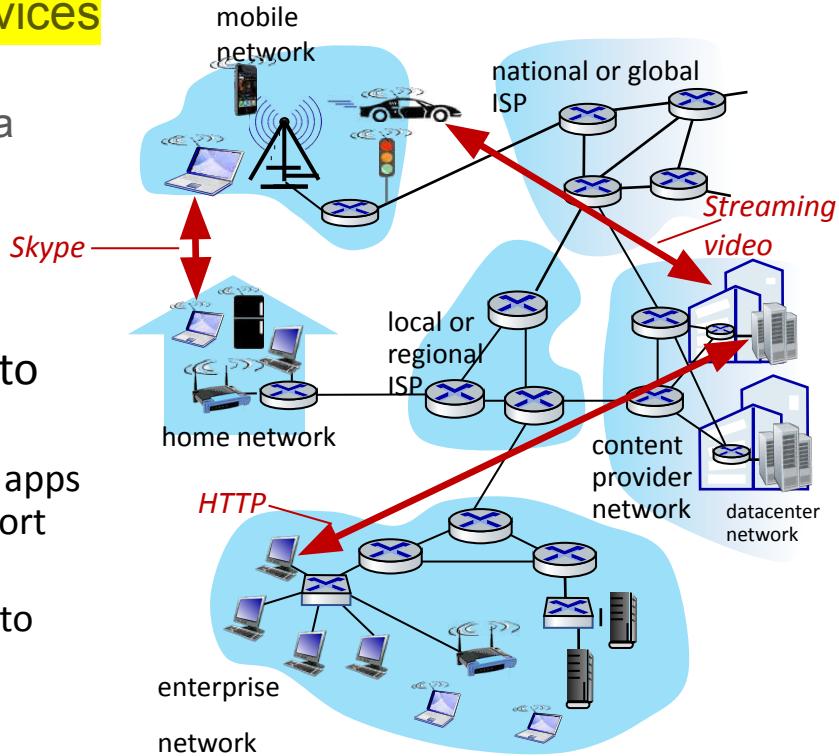
The Internet: A “Service” View

- *Infrastructure* that provides services to applications:

- Web, streaming video, multimedia teleconferencing, email, games, e-commerce, social media, inter-connected appliances, ...

- provides *programming interface* to distributed applications:

- “hooks” allowing sending/receiving apps to “connect” to, use Internet transport service
- provides service options, analogous to postal service



What is a Protocol?

Human protocols:

- “what’s the time?”
- “I have a question”
- introductions

... specific messages sent

... specific actions taken when message received, or other events

Network protocols:

- computers (devices) rather than humans
- all communication activity in Internet governed by protocols

*Protocols define the **format, order** of messages sent and received among network entities, and **actions taken** on msg transmission, receipt.*

Network Edge: A closer look at network structure

Network edge:

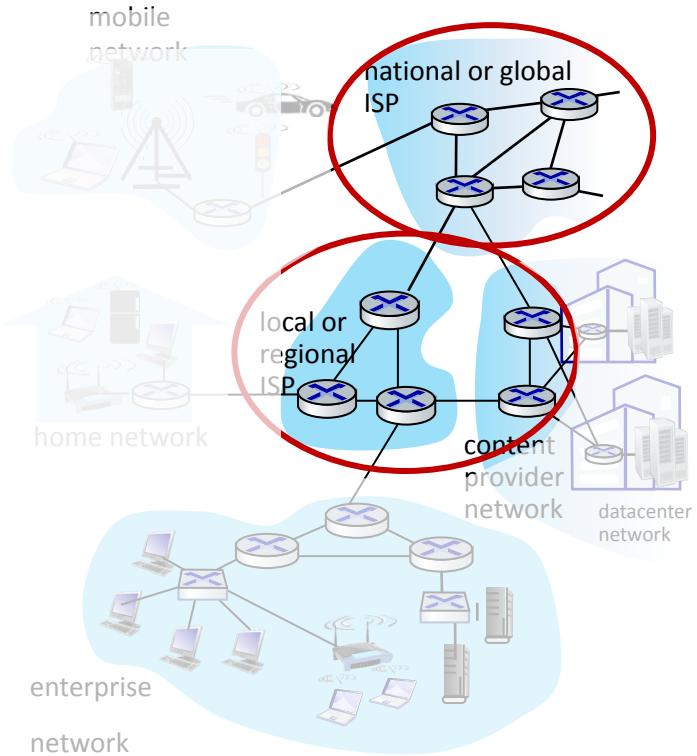
- Hosts: clients and servers
- Servers in data centers

Access networks, physical media:

- wired, wireless communication links

Network core:

- interconnected routers
- network of networks



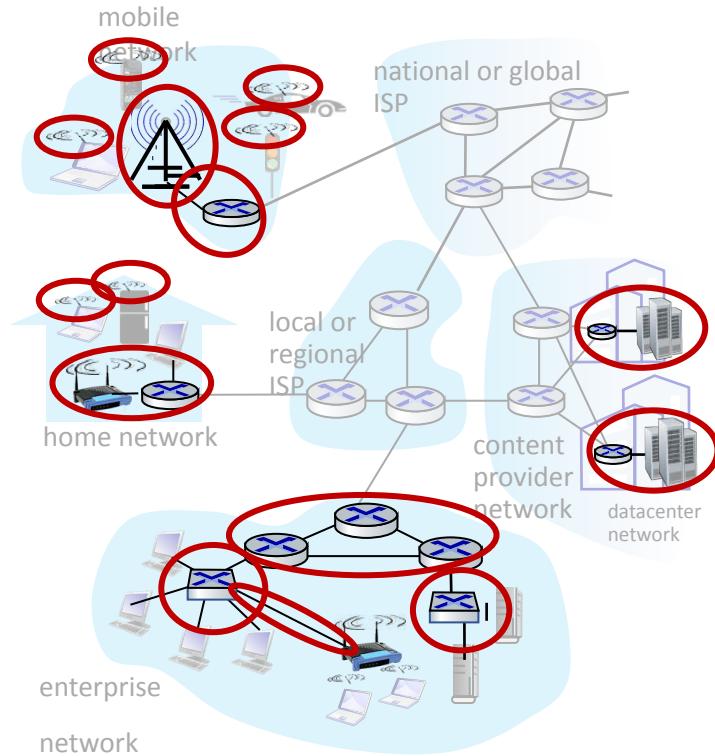
Network Edge: Access networks and Physical media

Q: How to connect end systems to edge router?

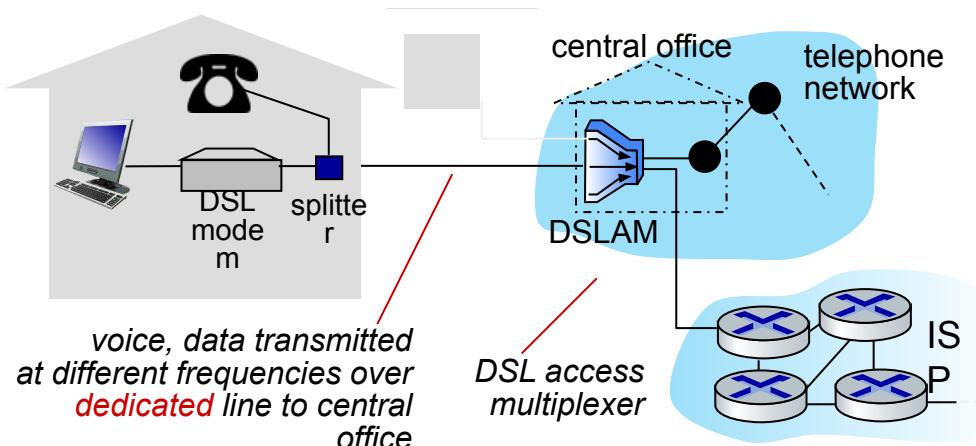
- Residential access networks
- Institutional access networks (school, company)
- Mobile access networks (WiFi, 4G/5G)

What to look for:

- Transmission rate (bits per second) of access network?
- Shared or dedicated access among users?



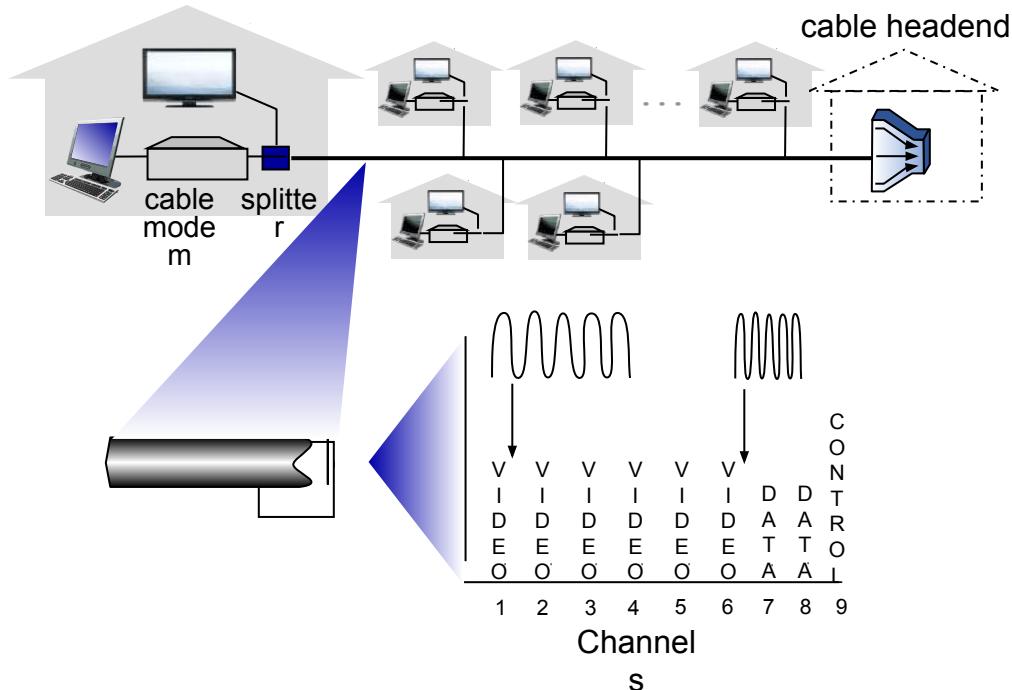
Network Edge: Access Networks - Digital Subscriber Line (DSL)



- 24-52 Mbps – downstream transmission rate
- 3.5-16 Mbps – upstream transmission rate
- Asymmetric access

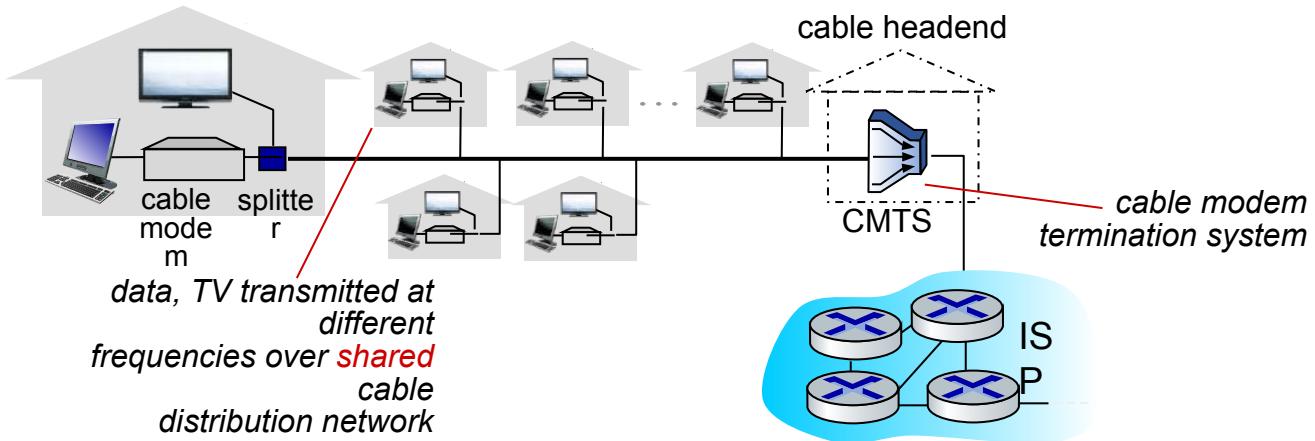
- use *existing* telephone line to central office DSLAM
 - **data** over DSL phone line goes to Internet
 - **voice** over DSL phone line goes to telephone net
- A high-speed downstream channel, in the 50 kHz to 1 MHz band
- A medium-speed upstream channel, in the 4 kHz to 50 kHz band
- An ordinary two-way telephone channel, in the 0 to 4 kHz band

Network Edge: Access Networks: Cable-based access



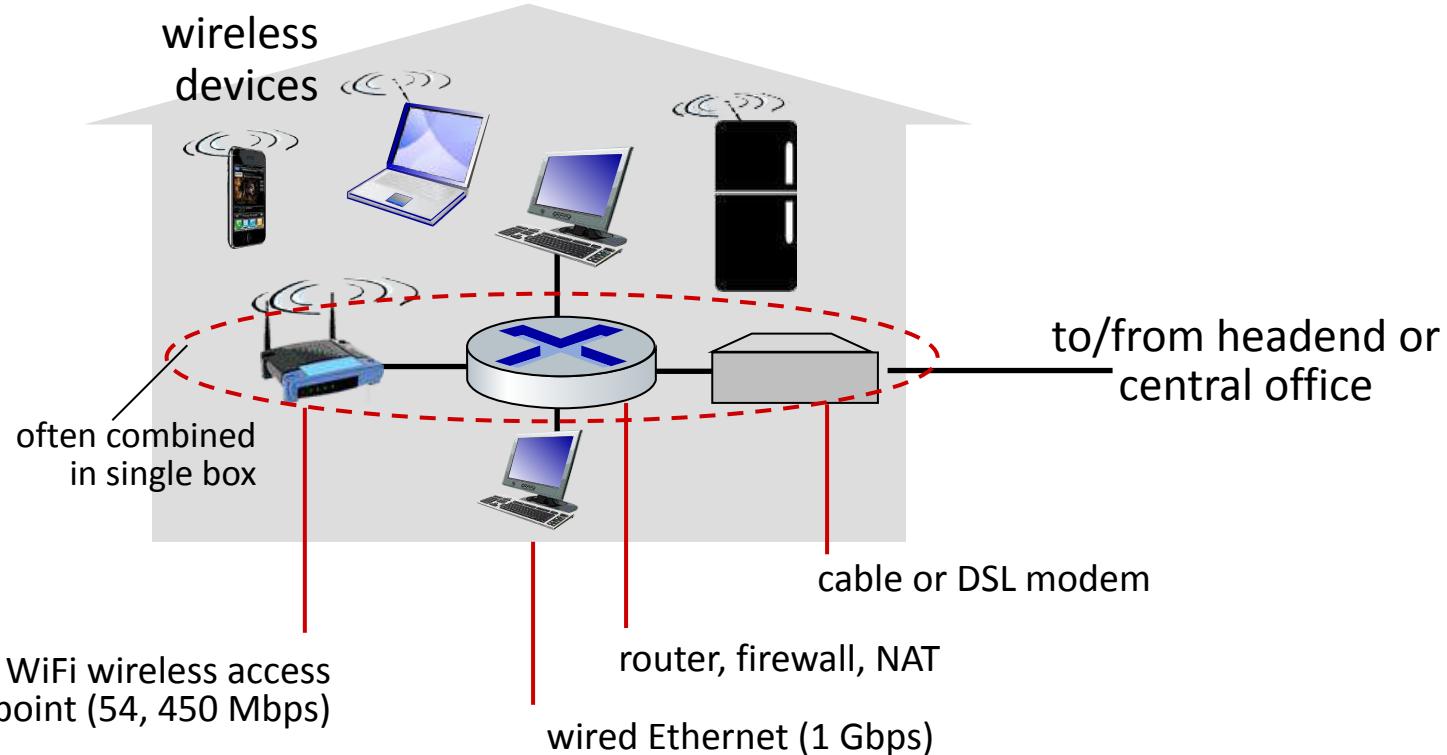
Frequency division multiplexing (FDM): different channels transmitted in different frequency bands

Network Edge: Access Networks: Cable-based access

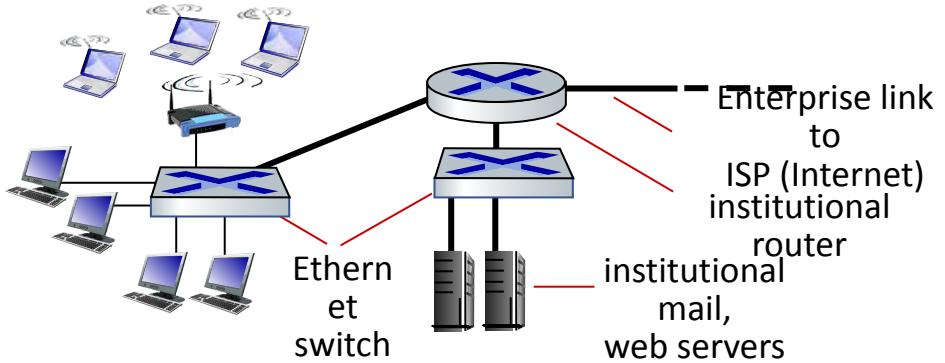


- HFC: hybrid fiber coax
 - Asymmetric:
 - up to 40 Mbps – 1.2 Gbs downstream transmission rate,
 - 30-100 Mbps upstream transmission rate
- Network of cable, fiber attaches homes to ISP router
 - homes **share access network** to cable headend

Network Edge: Access Networks – Home access



Network Edge: Access Networks – Enterprise networks



- companies, universities, etc.
- mix of wired, wireless link technologies, connecting a mix of switches and routers (we'll cover differences shortly)
 - Ethernet: wired access at 100Mbps, 1Gbps, 10Gbps
 - WiFi: wireless access points at 11, 54, 450 Mbps

Network Edge: Wireless Access Networks

Shared *wireless* access network connects end system to router

- via base station aka “access point”

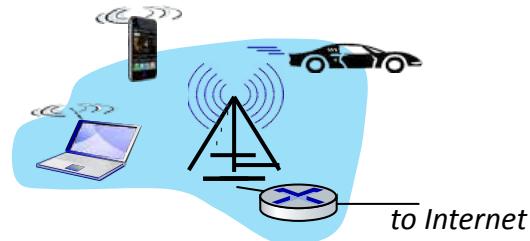
Wireless local area networks (WLANs)

- typically within or around building (~100 ft)
- 802.11b/g/n (WiFi): 11, 54, 450 Mbps transmission rate



Wide-area cellular access networks

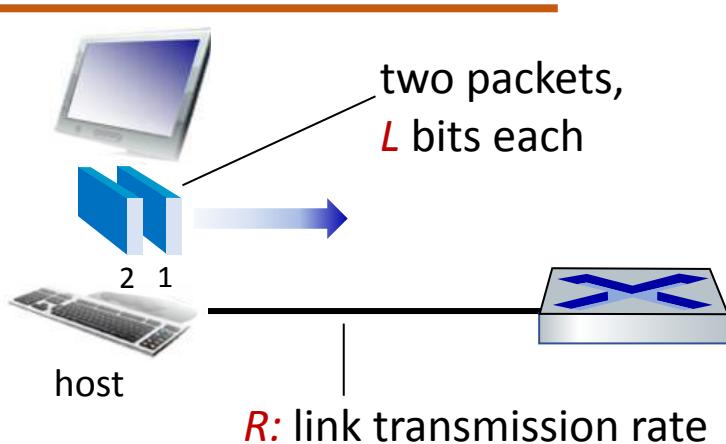
- provided by mobile, cellular network operator (10's km)
- 10's Mbps
- 4G cellular networks (5G coming)



Hosts: Send packets of data

Host sending function:

- takes application message
- breaks into smaller chunks, known as *packets*, of length L bits
- transmits packet into access network at *transmission rate R*
 - link transmission rate, aka link *capacity, aka link bandwidth*



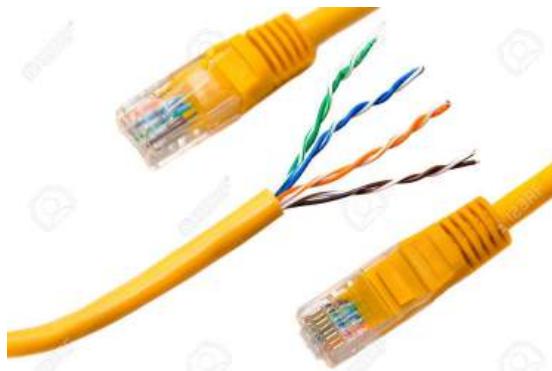
$$\text{packet transmission delay} = \frac{\text{time needed to transmit } L\text{-bit packet into link}}{R \text{ (bits/sec)}} = \frac{L \text{ (bits)}}{R \text{ (bits/sec)}}$$

Network Edge: Physical media

- **bit:** propagates between transmitter/receiver pairs
- **physical link:** what lies between transmitter & receiver
- **guided media:**
 - signals propagate in solid media: copper, fiber, coax
- **unguided media:**
 - signals propagate freely, e.g., radio

Twisted pair (TP)

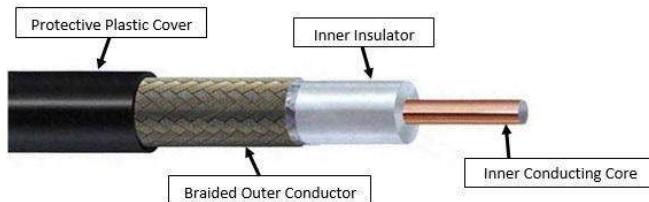
- two insulated copper wires (STP & UTP)
 - Category 5: 100 Mbps, 1 Gbps Ethernet
 - Category 6: 10Gbps Ethernet



Network Edge: Physical media

Coaxial cable:

- two concentric copper conductors
- concentric rather than parallel
- bidirectional
- broadband:
 - multiple frequency channels on cable
 - 100's Mbps per channel



Fiber optic cable:

- glass fiber carrying light pulses, each pulse a bit
- high-speed operation:
 - high-speed point-to-point transmission (10's-100's Gbps)
- low error rate:
 - repeaters spaced far apart
 - immune to electromagnetic noise



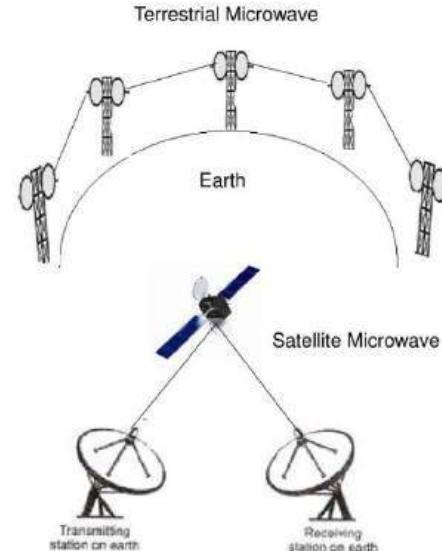
Network Edge: Physical media

Wireless radio

- signal carried in electromagnetic spectrum
- no physical “wire”
- broadcast and “half-duplex” (sender to receiver)
- propagation environment effects:
 - reflection
 - obstruction by objects
 - interference

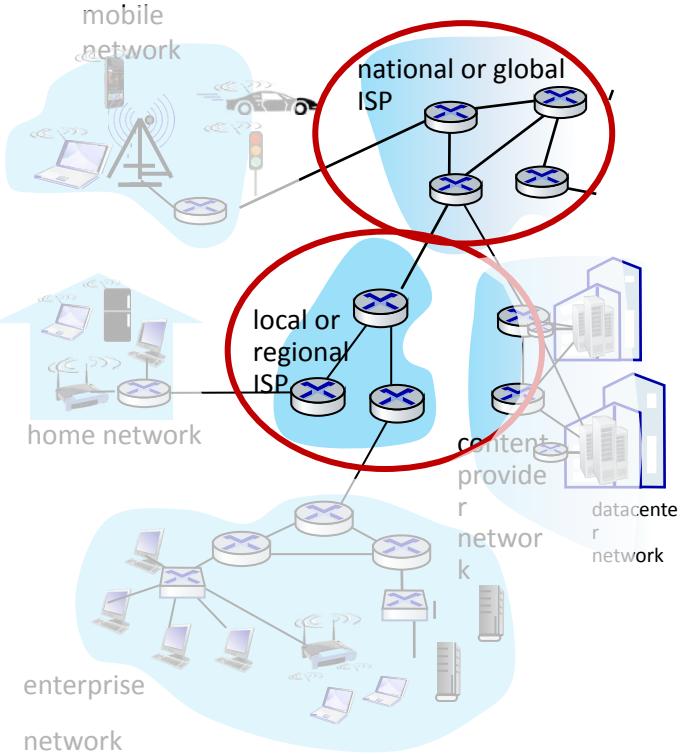
Radio link types:

- terrestrial microwave
 - up to 45 Mbps channels
- Wireless LAN (WiFi)
 - Up to 100's Mbps
- wide-area (e.g., cellular)
 - 4G cellular: ~ 10's Mbps
- satellite
 - up to 45 Mbps per channel
 - 280 msec end-end delay
 - geosynchronous vs. low-earth-orbit

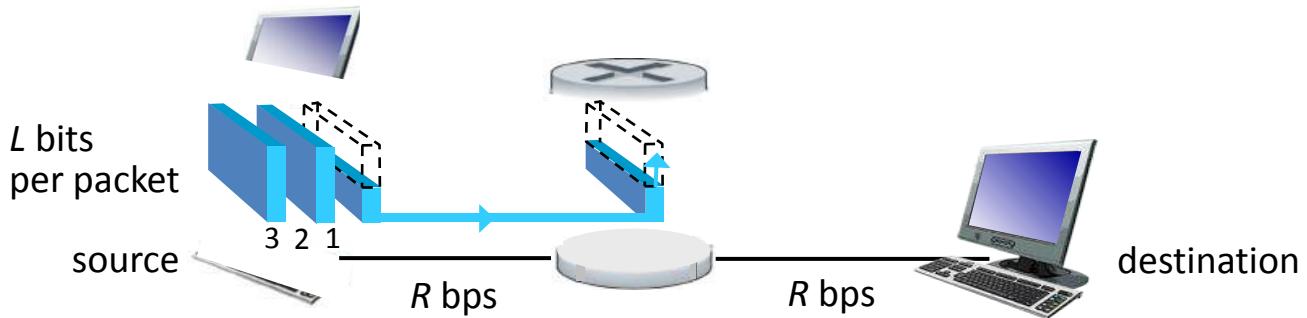


Network Core

- mesh of interconnected routers
- **packet-switching:** hosts break application-layer messages into *packets*
 - forward packets from one router to the next, across links on path from source to destination
 - each packet transmitted at full link capacity



Network Core: Packet Switching: store-and-forward

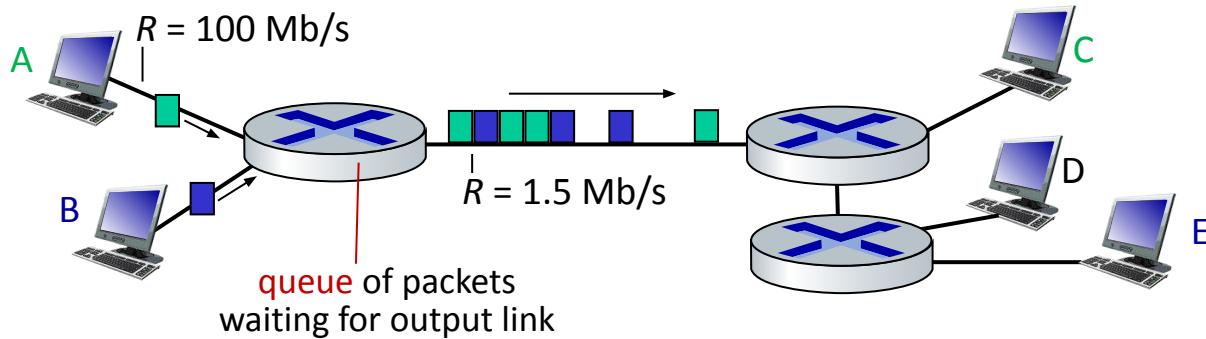


- **Transmission delay:** takes L/R seconds to transmit (push out) L -bit packet into link at R bps
- **Store and forward:** entire packet must arrive at router before it can be transmitted on next link
- **End-end delay:** $2L/R$ (above), assuming zero propagation delay
(more on delay shortly)

One-hop numerical example:

- $L = 10$ Kbits
- $R = 100$ Mbps
- one-hop transmission delay = 0.1 msec

Network Core: Packet Switching: queuing delay, loss



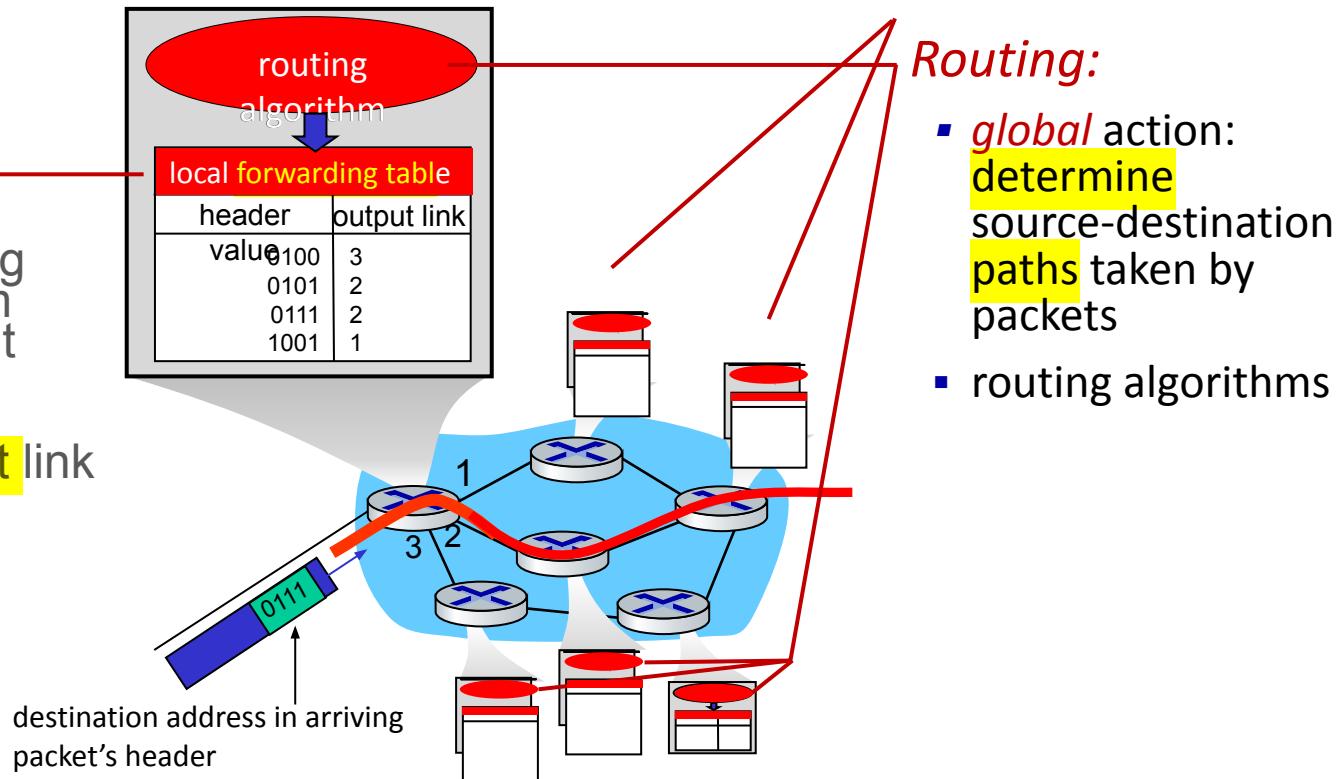
Packet queuing and loss: if arrival rate (in bps) to link exceeds transmission rate (bps) of link for a period of time:

- packets will queue, waiting to be transmitted on output link
- packets can be dropped (lost) if memory (buffer) in router fills up

Network Core: Two Key Network Core Functions

Forwarding:

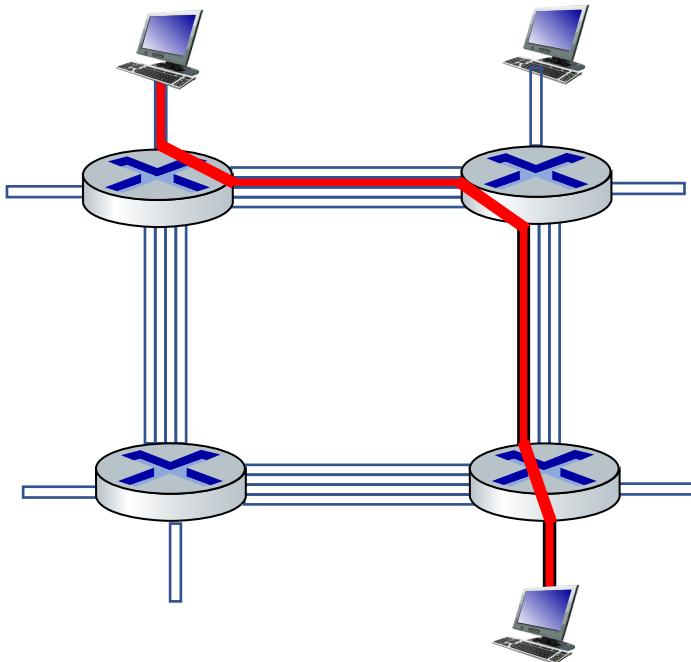
- **local** action: move arriving packets from router's input link to appropriate router output link



Network Core: Circuit Switching

end-end resources allocated to, reserved for “call” between source and destination (eg: telephone)

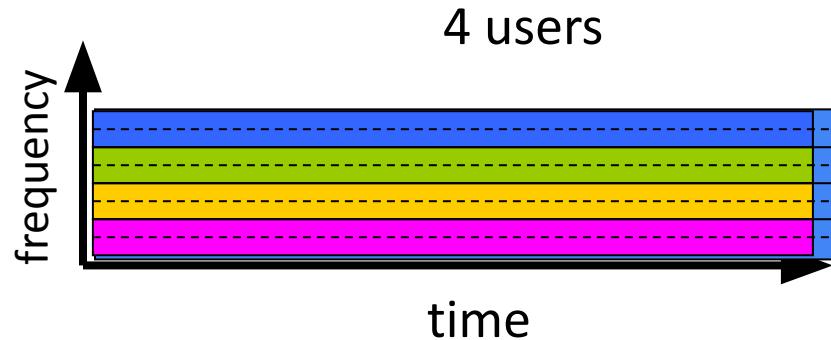
- in diagram, each link has four circuits.
 - call gets 2nd circuit in top link and 1st circuit in right link.
- dedicated resources: no sharing
 - circuit-like (guaranteed) performance
- circuit segment idle if not used by call (**no sharing**)
- commonly used in traditional telephone networks



Multiplexing in Circuit Switched Networks: FDM & TDM

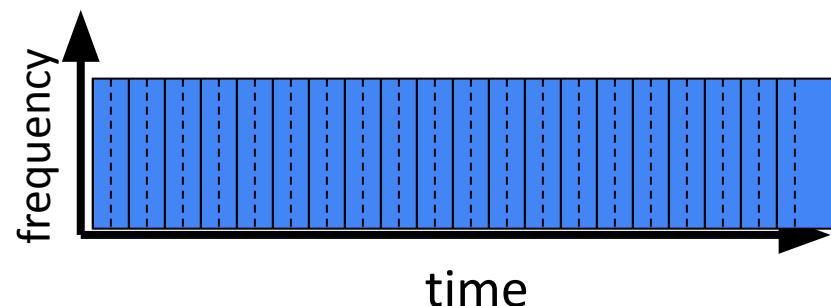
Frequency Division Multiplexing (FDM)

- optical, electromagnetic frequencies divided into (narrow) frequency bands
- each call allocated its own band, can transmit at max rate of that narrow band



Time Division Multiplexing (TDM)

- time divided into frames -> slots
- each call allocated periodic slot(s), can transmit at maximum rate of (wider) frequency band, but only during its time slot(s)

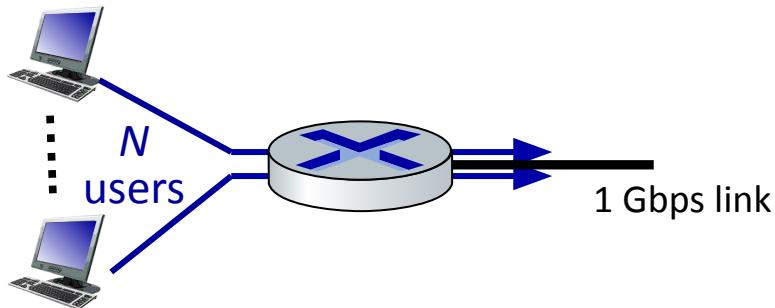


Network Core: Packet Switching vs Circuit Switching

packet switching allows more users to use network!

Example:

- 1 Gb/s link
- each user:
 - 100 Mb/s when “active”
 - active 10% of time



▪ **circuit-switching:** 10 users

- **packet switching:** with 35 users,
- probability > 10 active users at same time is less than .0004 *
- 10 or few active users, probability 0.9996

Q: how did we get value 0.0004?

Q: what happens if > 35 users ?

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive

Network Core: Packet Switching vs Circuit Switching

Is packet switching a “slam dunk winner”?

- great for “bursty” data – sometimes has data to send, but at other times not
 - resource sharing
 - simpler, no call setup
- **excessive congestion possible:** packet delay and loss due to buffer overflow
 - protocols needed for reliable data transfer, congestion control
- **Q: How to provide circuit-like behavior?**
 - bandwidth guarantees traditionally used for audio/video applications

Q: human analogies of reserved resources (circuit switching) versus on-demand allocation (packet switching)?

Packet Switching vs Circuit Switching – Numerical Example

- How long does it take to send a file of 640,000 bits (1 byte = 8 bits) from host A to host B over a circuit-switched network?
 - All links are 1.536 Mbps
 - Each link uses TDM with 24 slots/sec
 - 500 msec to establish end-to-end circuit

Let's work it out!

Solution:

- Each circuit has a transmission rate of $(1.536 \text{ Mbps})/24 = 64 \text{ kbps}$
- It takes $(640,000 \text{ bits})/(64 \text{ kbps}) = 10 \text{ seconds}$ to transmit the file
- To this 10 seconds we add the circuit establishment time, giving 10.5 seconds to send the file



Network Core: Packet Switching vs Circuit Switching

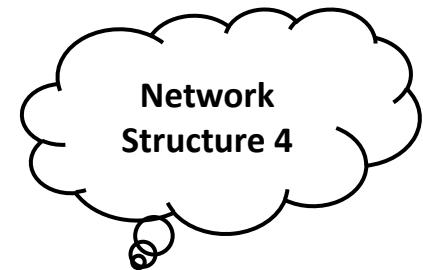
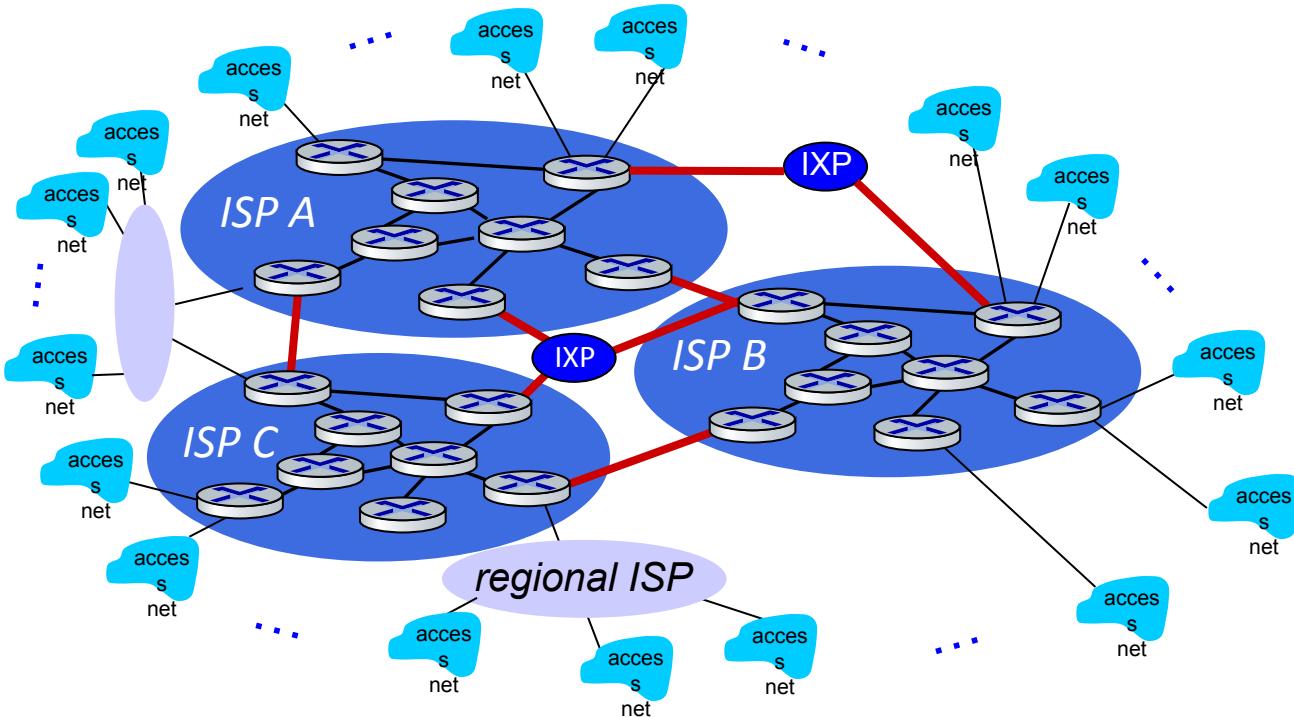
- Connectionless
- Designed for data
- Flexible
- Out of order, assembled at the dest
- Forward, Store & Fwd
- Network layer
- Bandwidth is saved (dynamic)
- Transmission of data – Source, routers
- Transmission delay
- Connection oriented
- Designed for voice
- Inflexible
- Message received in same order
- **FDM & TDM**
- Physical layer
- Bandwidth is wasted (fixed)
- Transmission of data – source
- Call setup delay

Internet Structure: a “network of networks”

- ❖ End systems connect to Internet via **access ISPs** (Internet Service Providers)
 - Residential, company and university ISPs
- ❖ Access ISPs in turn must be interconnected.
 - ❖ So that any two hosts can send packets to each other
- ❖ Resulting network of networks is very complex
 - ❖ Evolution was driven by **economics** and **national policies**
- ❖ Let's take a stepwise approach to describe current Internet structure

Internet Structure: a “network of networks”

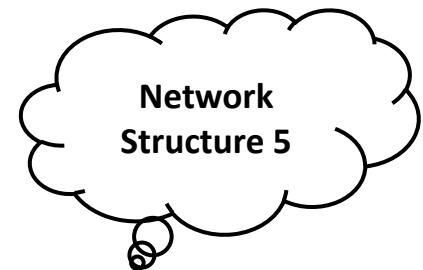
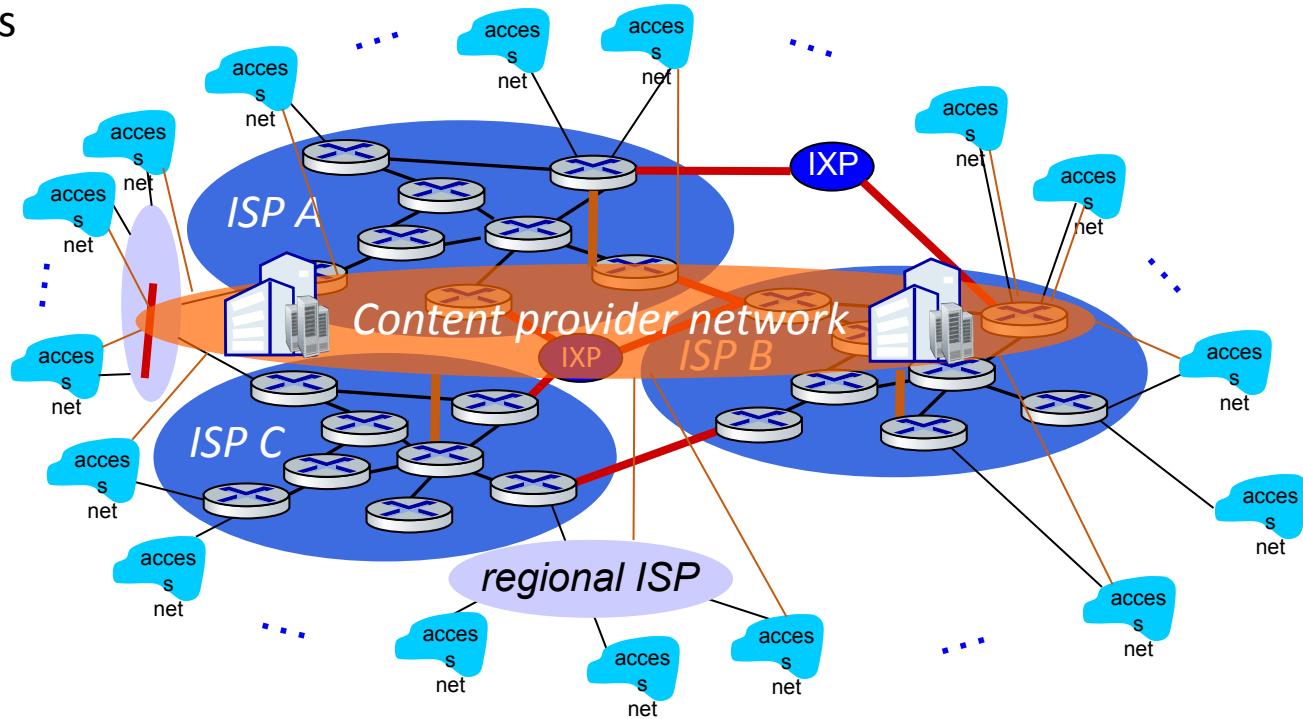
... and regional networks may arise to connect access nets to ISPs



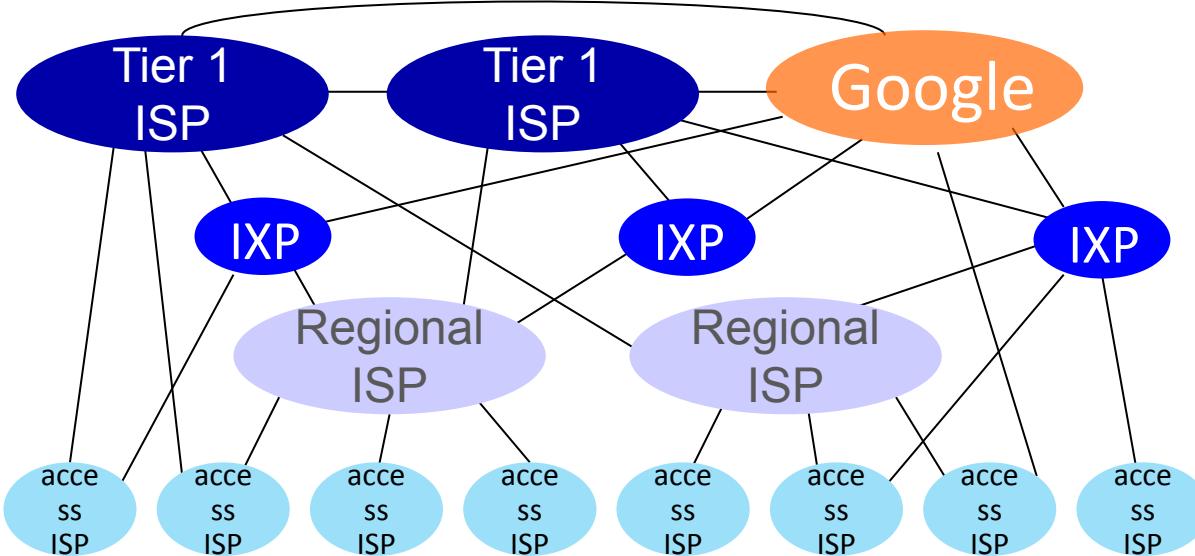
COMPUTER NETWORKS

Internet Structure: a “network of networks”

... and content provider networks (e.g., Google, Microsoft, Akamai) may run their own network, to bring services, content close to end users



Internet Structure: a “network of networks”



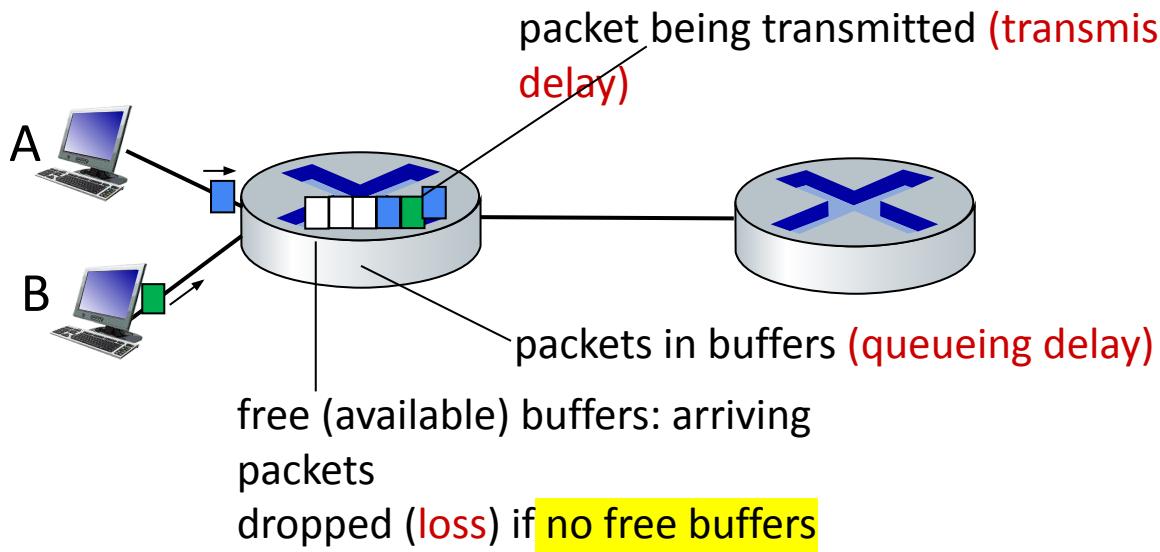
At “center”: small # of well-connected large networks

- **“tier-1” commercial ISPs** (e.g., Level 3, Sprint, AT&T, NTT), national & international coverage
- **content provider networks** (e.g., Google, Facebook): private network that connects its data centers to Internet, often bypassing tier-1, regional ISPs

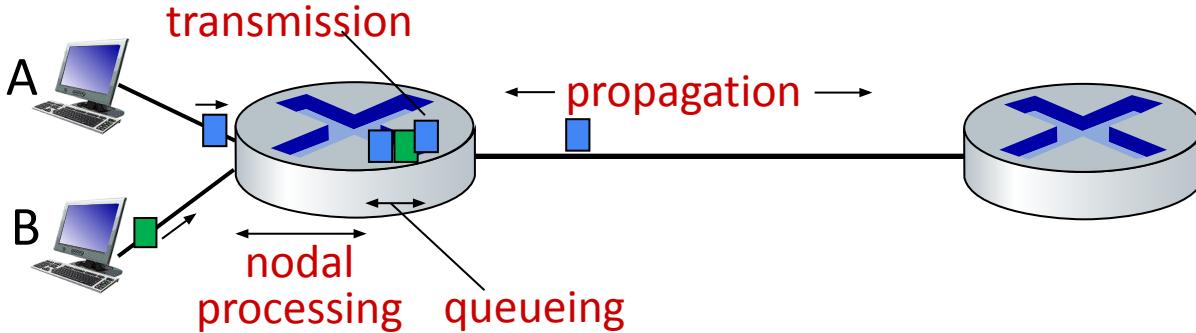
How do packet loss and delay occurs?

packets queue in router buffers

- packets queue, wait for turn
- arrival rate to link (temporarily) exceeds output link capacity: packet loss



Performance: Packet Delay – 4 Sources



$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

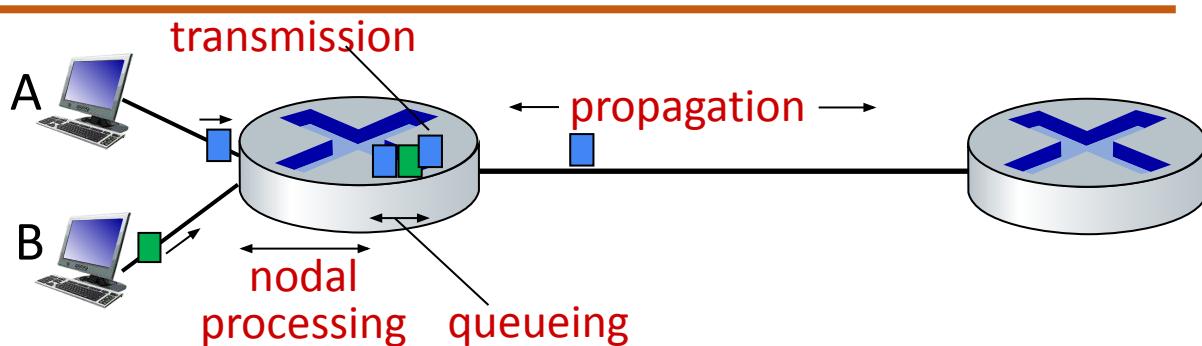
d_{proc} : nodal processing

- check bit errors
- determine output link
- typically < msec

d_{queue} : queueing delay

- time waiting at output link for transmission
- depends on congestion level of router
- microseconds to milliseconds

Performance: Packet Delay – 4 Sources



* Check out the online interactive exercises:
http://gaia.cs.umass.edu/kurose_ross

$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

d_{trans} : transmission delay:

- L : packet length (bits)
- R : link transmission rate (bps)
- $d_{\text{trans}} = L/R$

d_{trans} and d_{prop}
very different

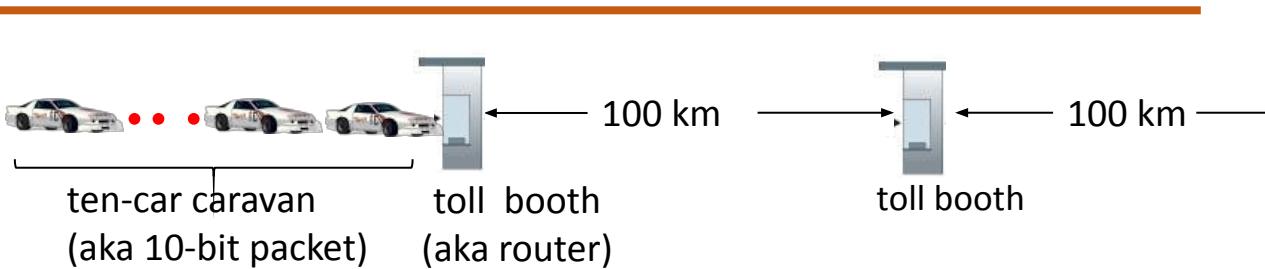
d_{prop} : propagation delay:

- d : length of physical link
- s : propagation speed ($\sim 2 \times 10^8$ m/sec)
- $d_{\text{prop}} = d/s$

Transmission Delay vs Propagation Delay

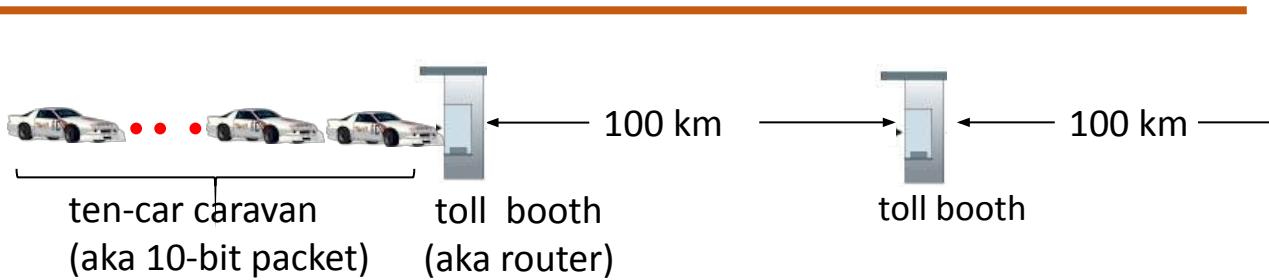
Transmission Delay	Propagation Delay
Time required for the router to push out the packet.	Time it takes a bit to propagate from one router to the next.
A function of the packet's length and the transmission rate of the link.	A function of the distance between the two routers.
$d_{trans} = L/R$	$d_{prop} = d/s$
Nothing to do with the distance between the two routers.	Nothing to do with the packet's length or the transmission rate of the link.

Performance: Delay – Caravan Analogy



- cars “propagate” at 100 km/hr
- toll booth takes 12 sec to service car (bit transmission time)
- car ~ bit; caravan ~ packet
- **Q:** How long until caravan is lined up before 2nd toll booth?
 - time to “push” entire caravan through toll booth onto highway = $12 * 10 = 120$ sec
 - time for last car to propagate from 1st to 2nd toll both:
 $100\text{km}/(100\text{km/hr}) = 1 \text{ hr}$
 - **A:** 62 minutes

Performance: Delay – Caravan Analogy (more)



- suppose cars now “propagate” at 1000 km/hr
- and suppose toll booth now takes one min to service a car
- **Q: Will cars arrive to 2nd booth before all cars serviced at first booth?**

A: Yes! after 7 min, first car arrives at second booth; three cars still at first booth

Performance: Packet Queueing Delay revisited

Unlike other delays (**dproc**, **dtrans**, **dprop**), **dqueue** is interesting.

- Can vary from packet to packet.
- Characterize d_{queue} -> average, variance, probability that it exceeds some specified value.

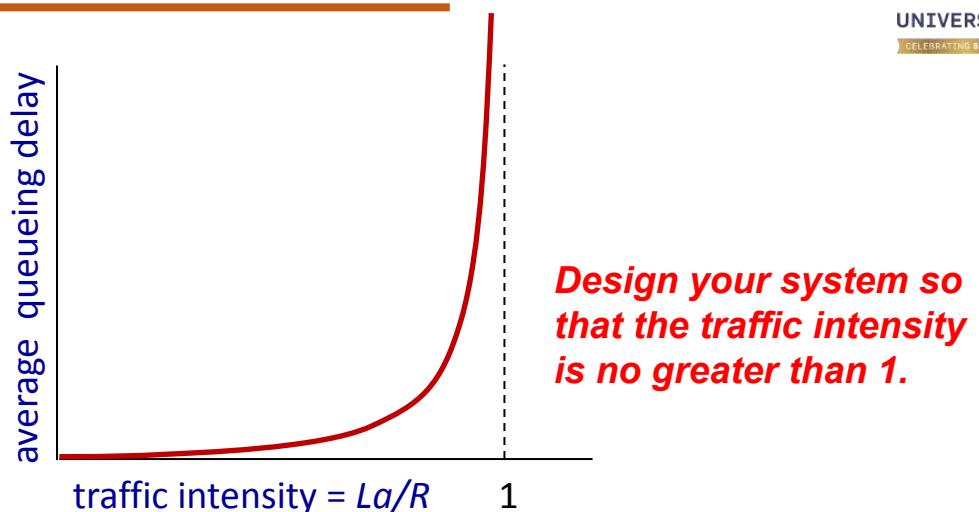
When is the queuing delay large and when is it insignificant?

- Rate at which traffic arrives at the queue,
- Transmission rate of the link,
- Nature of the arriving traffic – periodically or in bursts

Performance: Packet Queueing Delay revisited

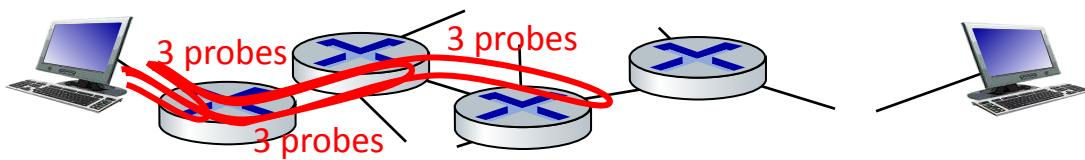
- R : link bandwidth (bps)
- L : packet length (bits)
- a : average packet arrival rate (pps)
- La : avg. rate at which bits arrive at the queue
- $La/R > 1$: more “work” arriving is more than can be serviced - average delay infinite!
- $La/R \leq 1$: nature of arriving traffic
- $La/R \sim 0$: avg. queueing delay small

$La/R > 1$: Average rate at which bits arrive at the queue exceeds the rate at which the bits can be transmitted from the queue.



“Real” Internet Delays and Routes

- what do “real” Internet delay & loss look like?
- **traceroute** program: provides delay measurement from source to router along end-end Internet path towards destination. For all i :
 - sends **three** packets that will reach router i on path towards destination (with time-to-live field value of i)
 - router i will return packets to sender
 - sender measures time interval between transmission and reply



“Real” Internet Delays and Routes

traceroute: gaia.cs.umass.edu to www.eurecom.fr

3 delay measurements from
gaia.cs.umass.edu to cs-gw.cs.umass.edu

3 delay measurements
to border1-rt-fa5-1-0.gw.umass.edu

trans-oceanic link

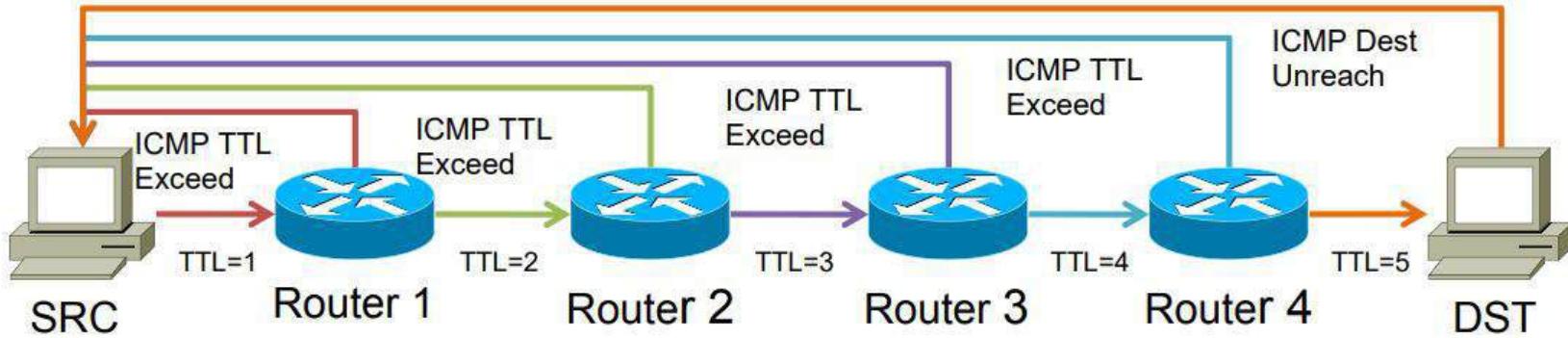
looks like delays
decrease! Why?

* means no response (probe lost, router not replying)

1	cs-gw (128.119.240.254)	1 ms	1 ms	2 ms	
2	border1-rt-fa5-1-0.gw.umass.edu (128.119.3.145)	1 ms	1 ms	2 ms	
3	cht-vbns.gw.umass.edu (128.119.3.130)	6 ms	5 ms	5 ms	
4	jn1-at1-0-0-19.wor.vbns.net (204.147.132.129)	16 ms	11 ms	13 ms	
5	jn1-so7-0-0-0.wae.vbns.net (204.147.136.136)	21 ms	18 ms	18 ms	
6	abilene-vbns.abilene.ucaid.edu (198.32.11.9)	22 ms	18 ms	22 ms	
7	nycm-wash.abilene.ucaid.edu (198.32.8.46)	22 ms	22 ms	22 ms	
8	62.40.103.253 (62.40.103.253)	104 ms	109 ms	106 ms	
9	de2-1.de1.de.geant.net (62.40.96.129)	109 ms	102 ms	104 ms	
10	de.fr1.fr.geant.net (62.40.96.50)	113 ms	121 ms	114 ms	
11	renater-gw.fr1.fr.geant.net (62.40.103.54)	112 ms	114 ms	112 ms	
12	nio-n2.cssi.renater.fr (193.51.206.13)	111 ms	114 ms	116 ms	
13	nice.cssi.renater.fr (195.220.98.102)	123 ms	125 ms	124 ms	
14	r3t2-nice.cssi.renater.fr (195.220.98.110)	126 ms	126 ms	124 ms	
15	eurecom-valbonne.r3t2.ft.net (193.48.50.54)	135 ms	128 ms	133 ms	
16	194.214.211.25 (194.214.211.25)	126 ms	128 ms	126 ms	
17	***				
18	***				
19	fantasia.eurecom.fr (193.55.113.142)	132 ms	128 ms	136 ms	

* Do some traceroutes from exotic countries at

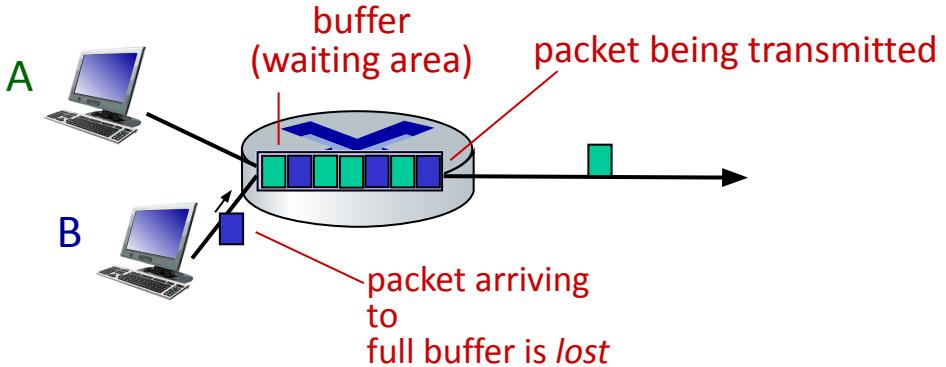
How Traceroute works?



Refer RFC 1393, **Traceroute Using an IP Option**
Don't Trust Traceroute (Completely)

Performance: Packet loss

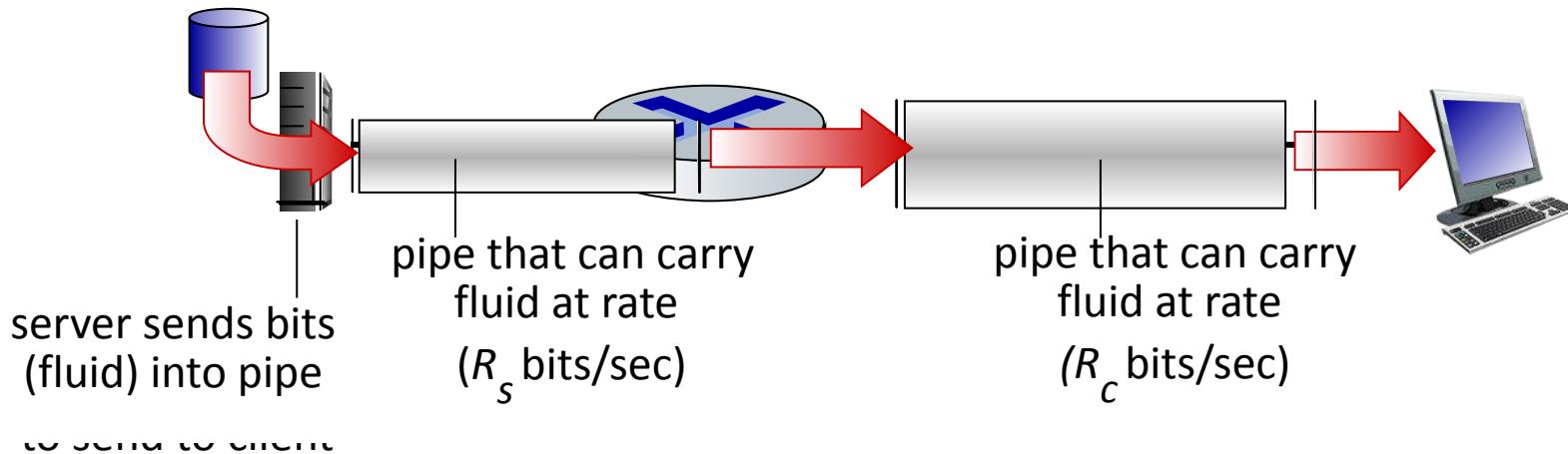
- queue (aka buffer) preceding link in buffer has finite capacity
- packet arriving to full queue dropped (aka lost)
- lost packet may be retransmitted by previous node, by source end system, or not at all



* Check out the Java applet for an interactive animation on queuing and loss

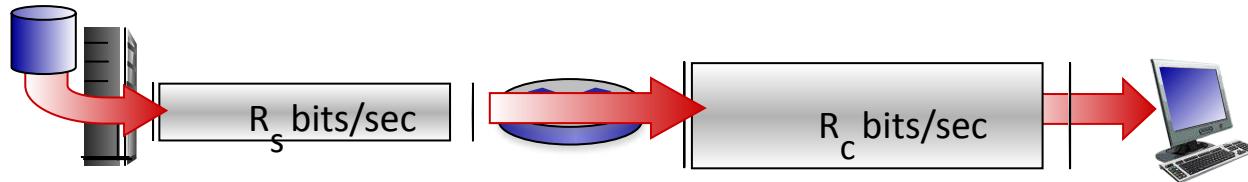
Performance: Throughput

- **throughput:** rate (bits/time unit) at which bits are being sent from sender to receiver
 - *instantaneous:* rate at given point in time
 - *average:* rate over longer period of time



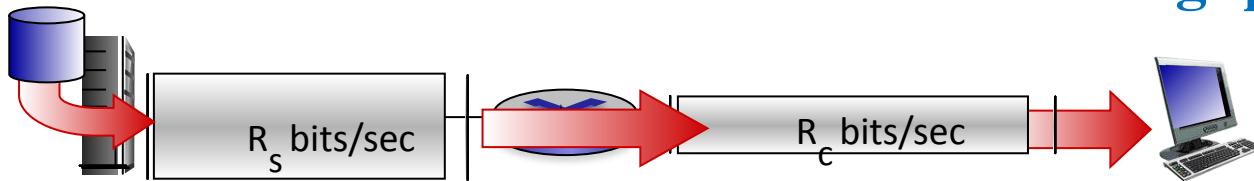
Performance: Throughput (more)

$R_s < R_c$ What is average end-end throughput?



$R_s > R_c$ What is average end-end throughput?

$$\text{Throughput} = \min\{R_s, R_c\}$$



bottleneck link

link on end-end path that constrains end-end throughput.

Throughput – Numerical Example

- Suppose you are downloading an MP3 file of $F = 32$ million bits.
- The server has a transmission rate of $R_s = 2$ Mbps and you have an access link of $R_c = 1$ Mbps.
- What is the time needed to transfer the file?

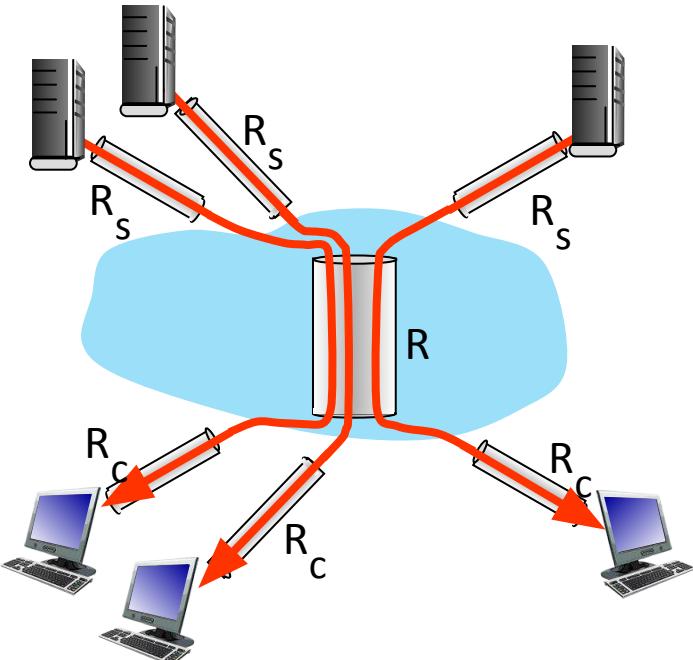
Let's work it out!

Solution:

- 32 seconds!



Performance: Throughput – Network Scenario



10 connections (fairly) share backbone bottleneck link R bits/sec

- per-connection end-end throughput: $\min(R_c, R_s, R/10)$
- in practice: R_c or R_s is often bottleneck

* Check out the online interactive exercises for more examples:
http://gaia.cs.umass.edu/kurose_ross/

- Suppose $R_s = 2$ Mbps, $R_c = 1$ Mbps, $R = 5$ Mbps
- 10 clients from 10 servers = 10 downloads

End-to-end throughput for each download is now reduced to 500 kbps.



Layering of Airline functionality



layers: each layer implements a service

- via its own internal-layer actions
- relying on services provided by layer below

*Q: describe in words
the service provided
in each layer above*

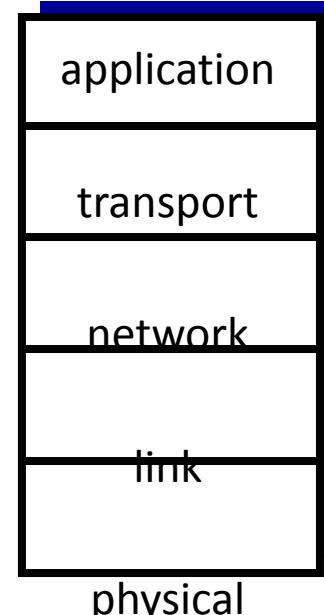
Why layering?

dealing with complex systems:

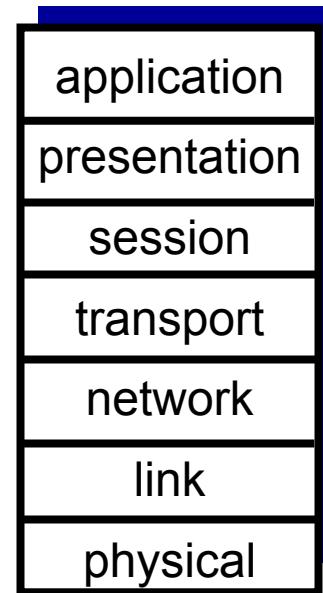
- explicit structure allows identification, relationship of complex system's pieces
 - layered *reference model* for discussion
- modularization eases maintenance, updating of system
 - change in layer's service *implementation*: transparent to rest of system
 - e.g., change in gate procedure doesn't affect rest of system
- layering considered harmful?
- layering in other complex systems?

Internet Protocol Stack

- ***application***: supporting network applications (access to network resources)
 - IMAP, SMTP, HTTP
- ***transport***: process-process data transfer (segmentation & reassembly, sockets, connection, flow and error control)
 - TCP, UDP
- ***network***: routing of datagrams from source to destination (addressing, routing)
 - IP, routing protocols
- ***link***: data transfer between neighboring network elements (framing, addressing, flow & error control)
 - Ethernet, 802.11 (WiFi), PPP
- ***physical***: bits “on the wire”



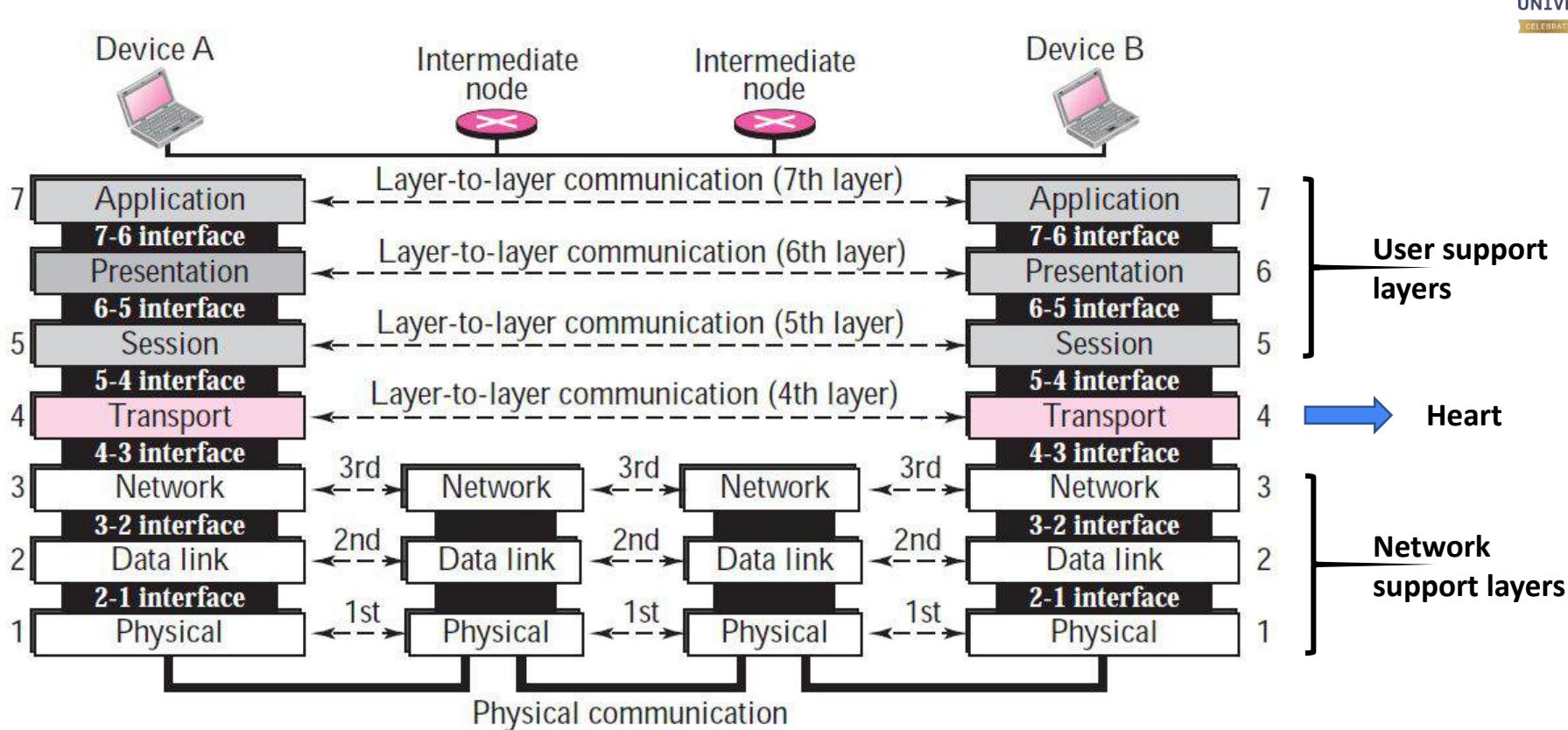
- *presentation*: allow applications to interpret meaning of data, (e.g., encryption, compression, machine-specific conventions)
- *session*: synchronization, checkpointing, recovery of data exchange
- Internet stack “missing” these layers!
 - these services, *if needed*, must be implemented in application
 - needed?



Open Systems Interconnection (OSI) model – introduced in late 1970s by ISO.

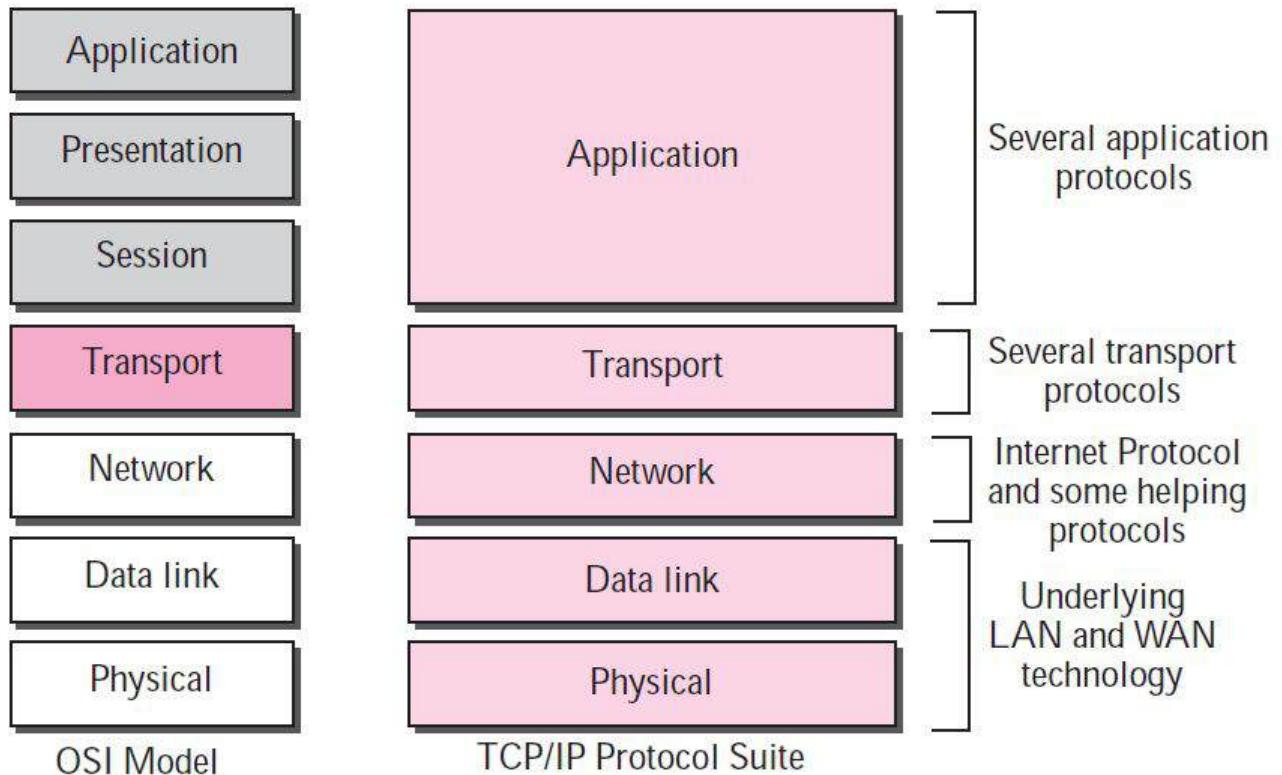
COMPUTER NETWORKS

OSI reference model (more)

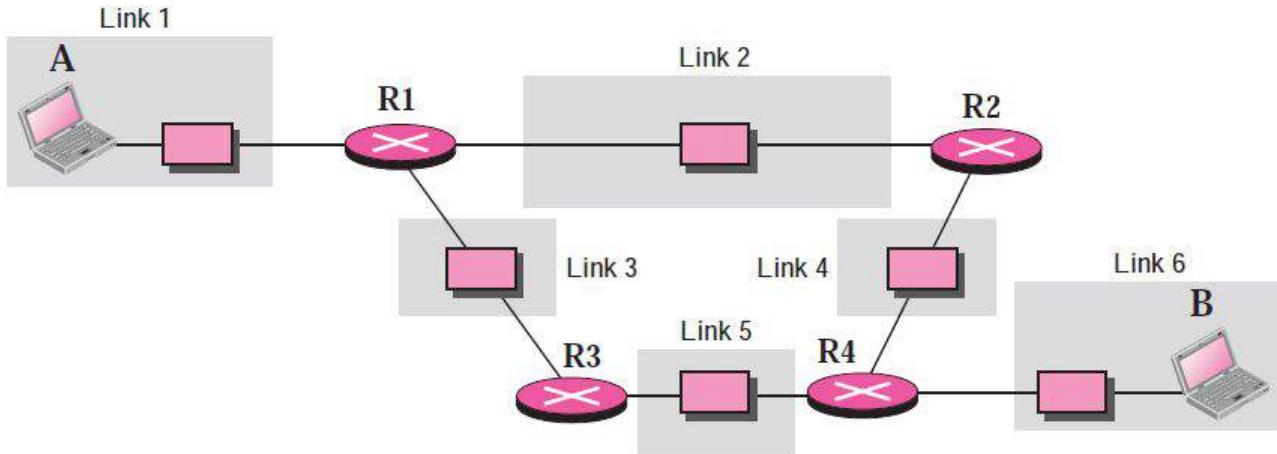


COMPUTER NETWORKS

TCP/IP vs OSI reference model

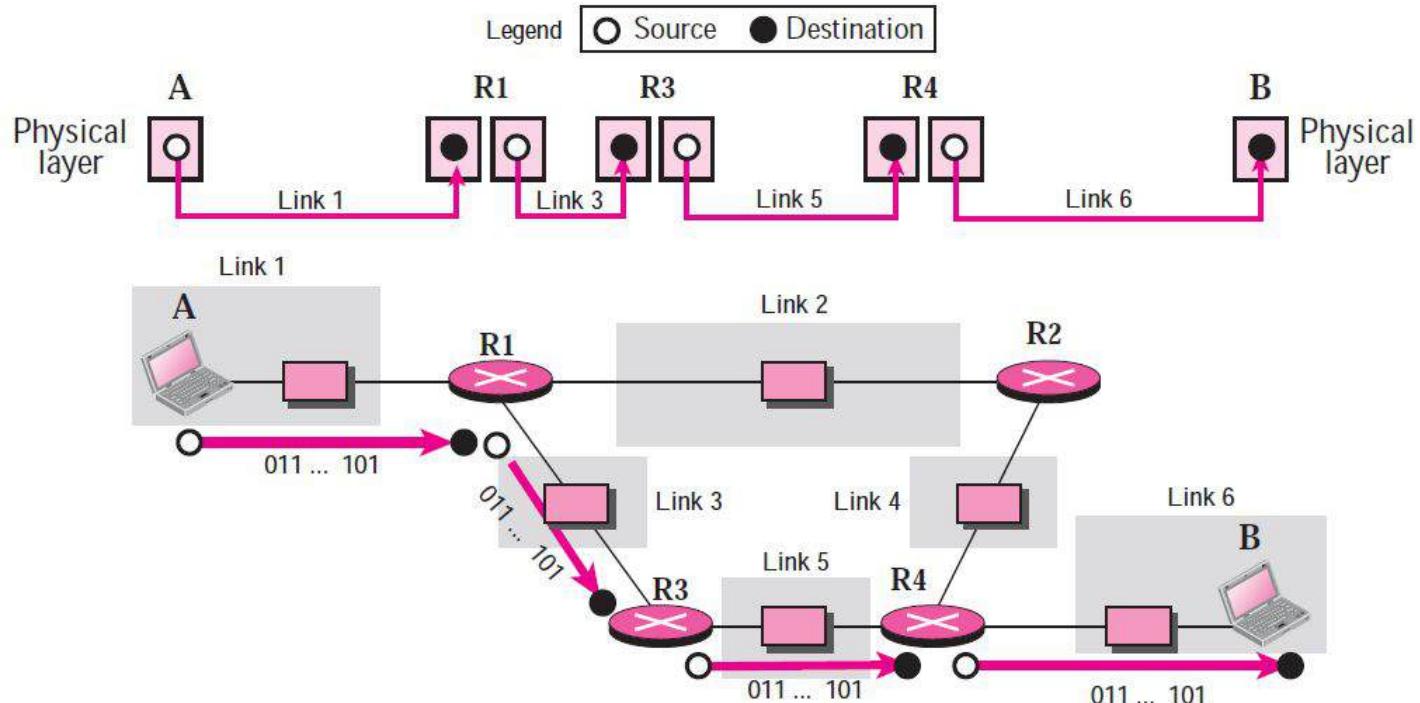


Layers in the TCP/IP Protocol Suite (more)



A private internet

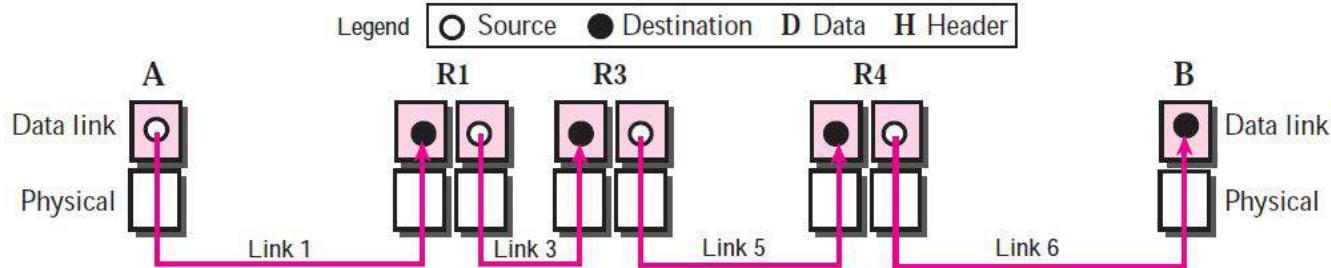
Layers in the TCP/IP Protocol Suite (more)



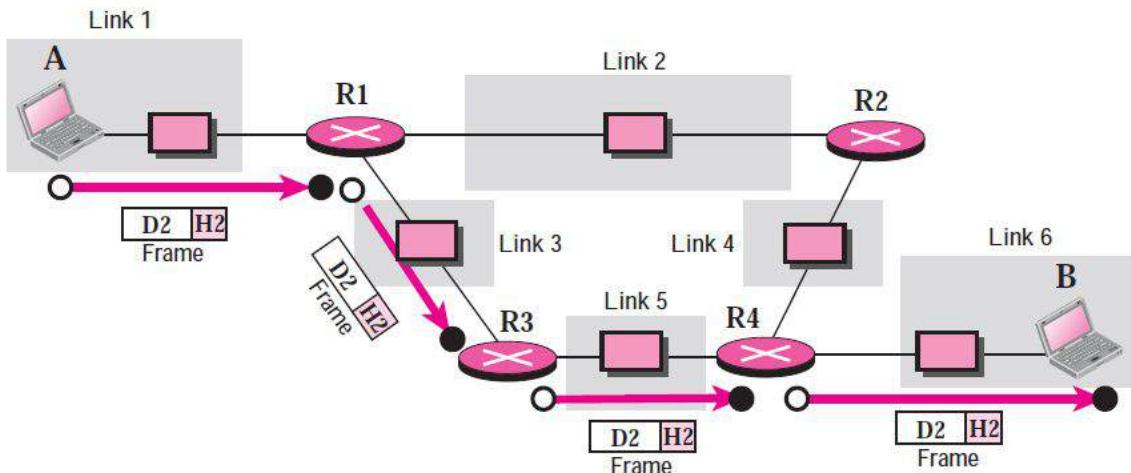
Communication at
the physical layer

Unit of Communication
– bit

Layers in the TCP/IP Protocol Suite (more)

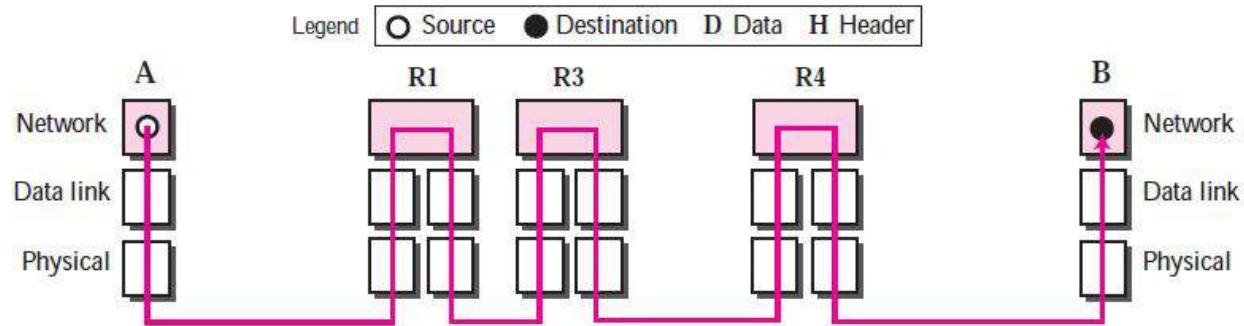


Communication at the
data link layer

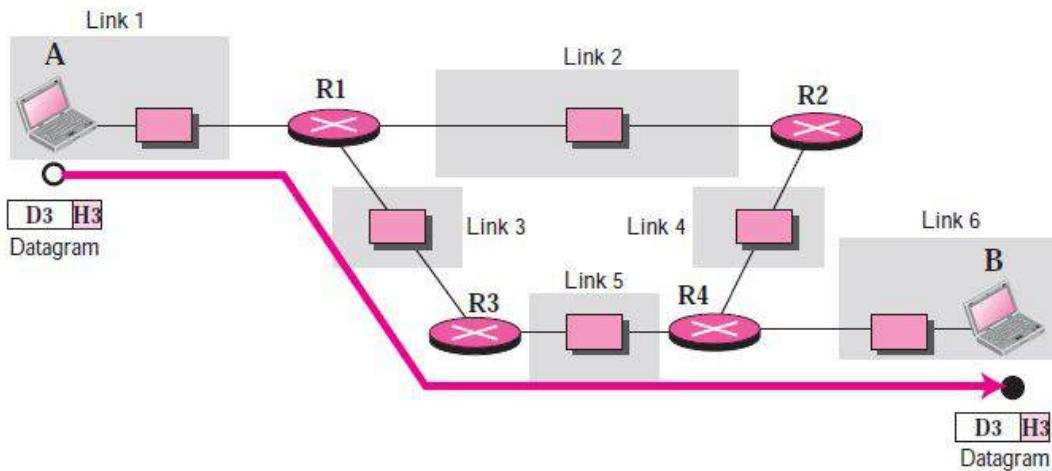


Unit of Communication –
frame

Layers in the TCP/IP Protocol Suite (more)

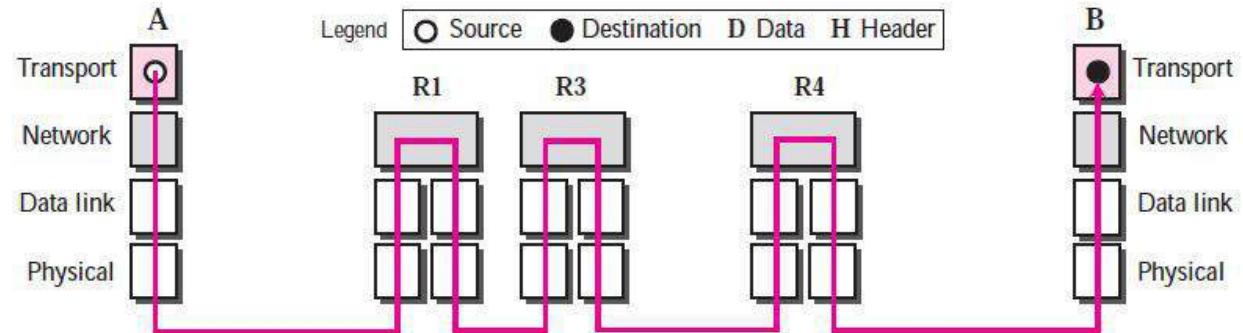


Communication at the network layer

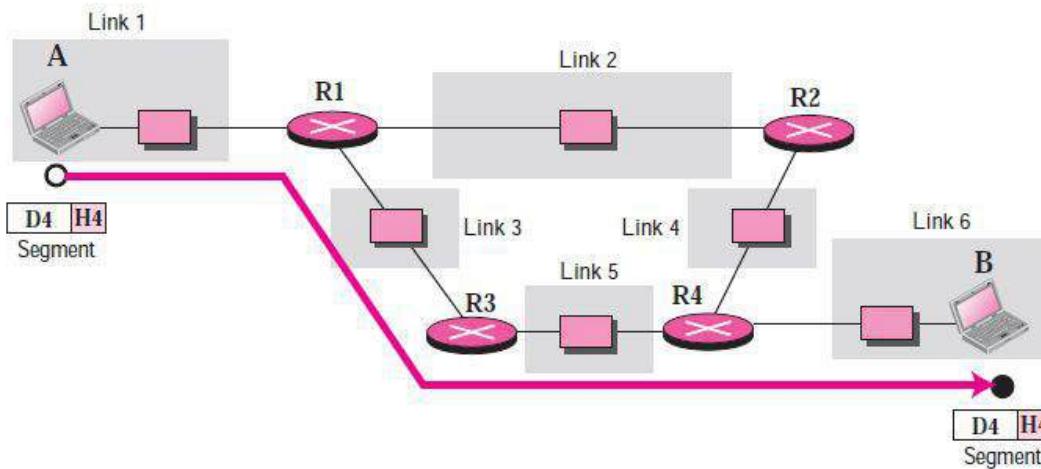


Unit of Communication –
datagram

Layers in the TCP/IP Protocol Suite (more)



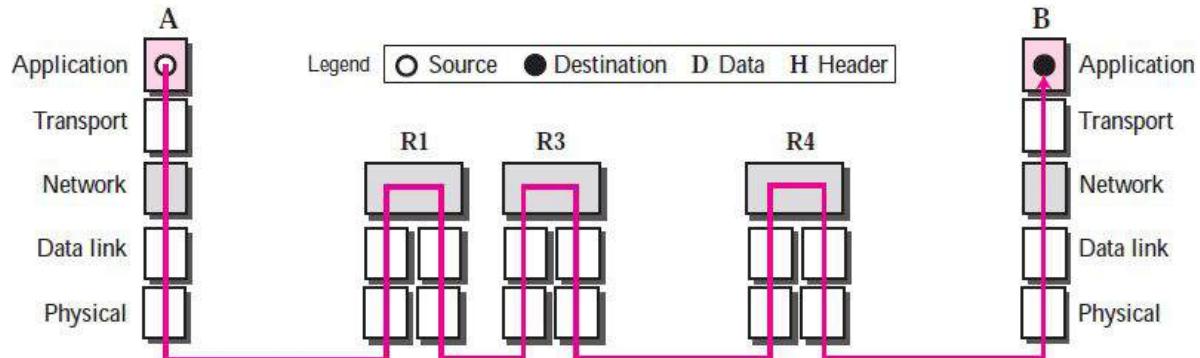
Communication at the transport layer



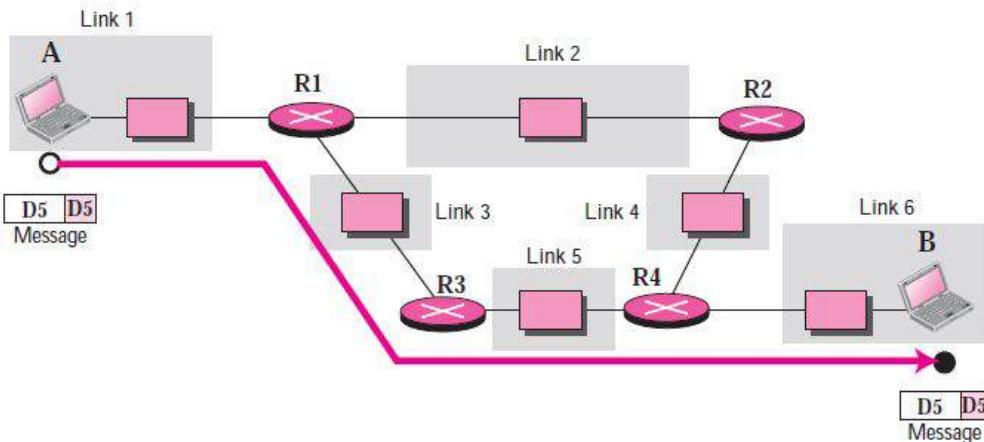
Unit of Communication
– segment/packet

COMPUTER NETWORKS

Layers in the TCP/IP Protocol Suite (more)



Communication at the application layer



Unit of Communication
— message

The OSI Model

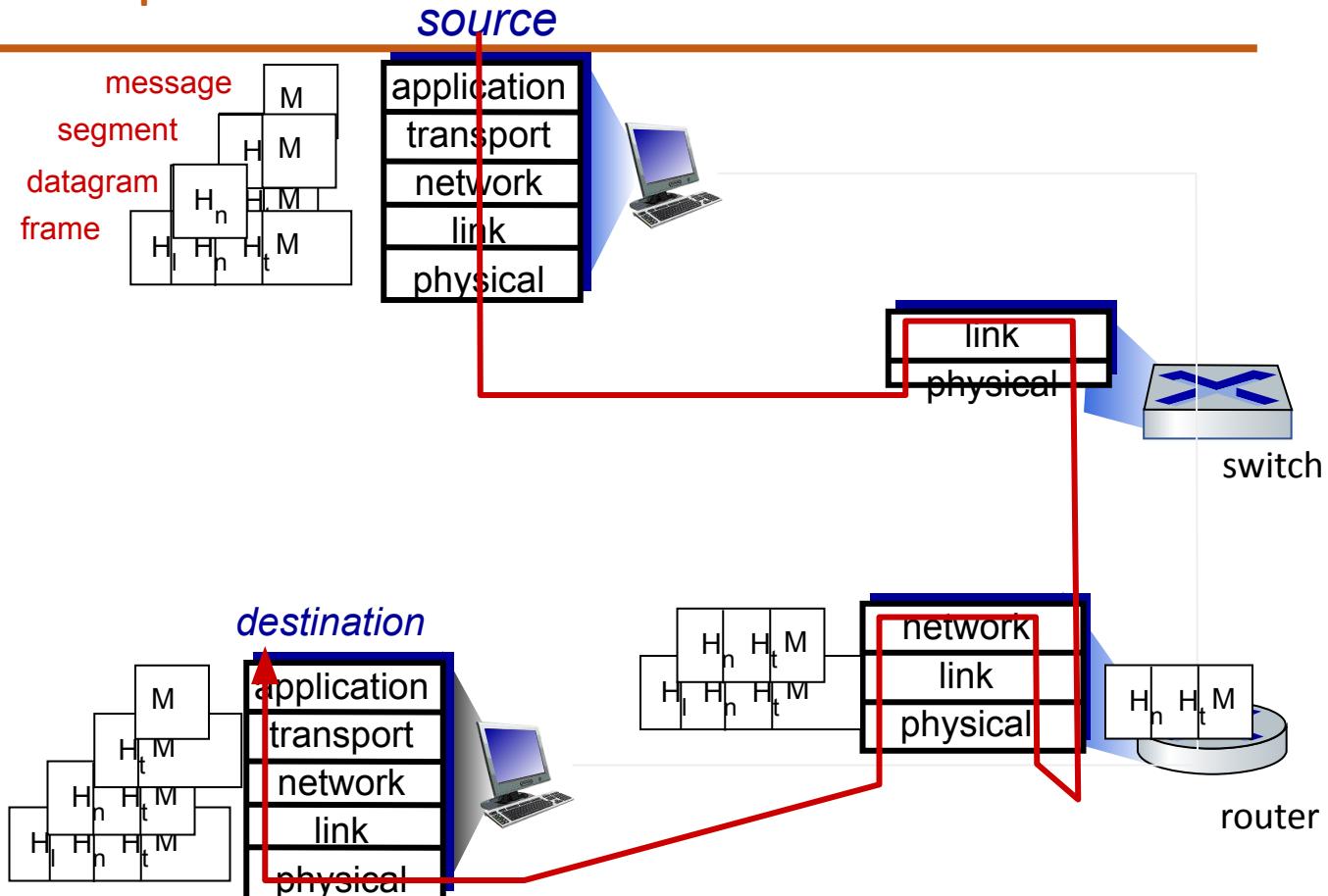


The TCP/IP Model

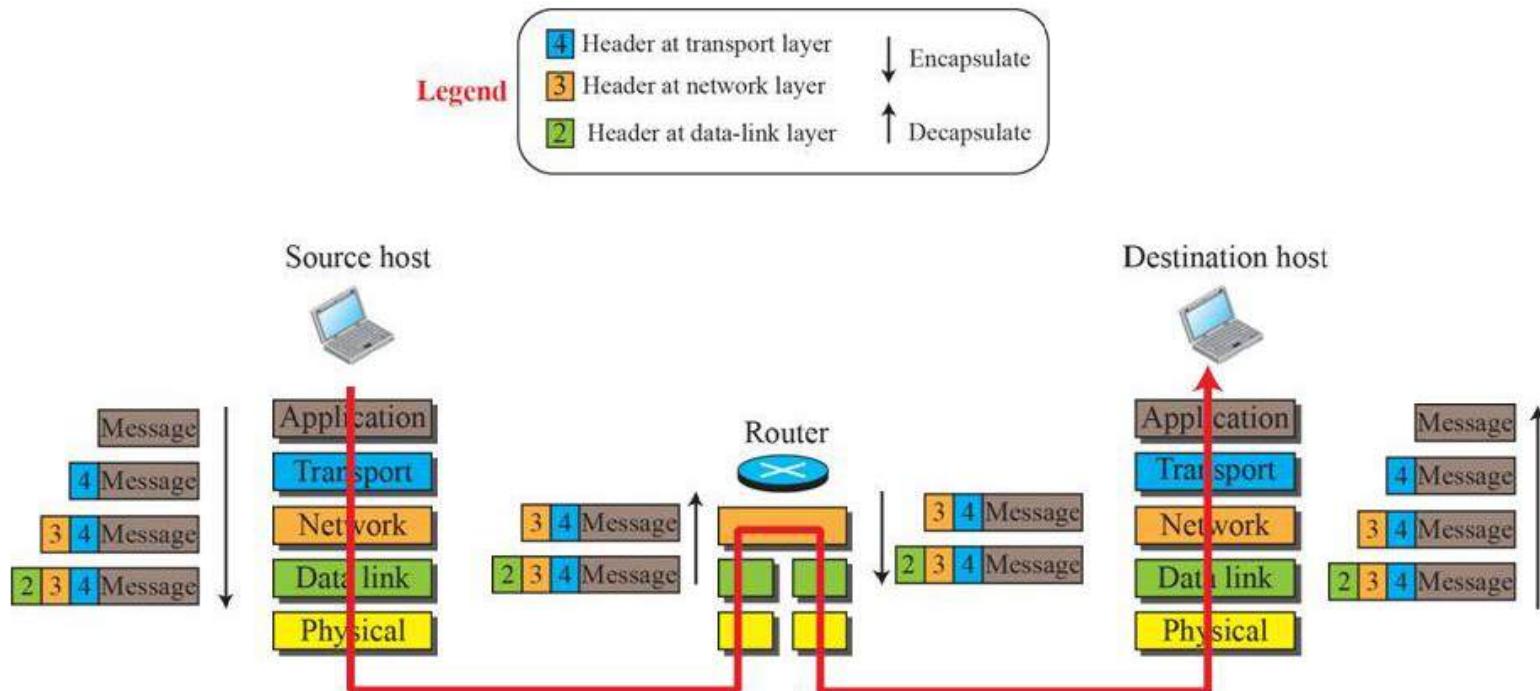


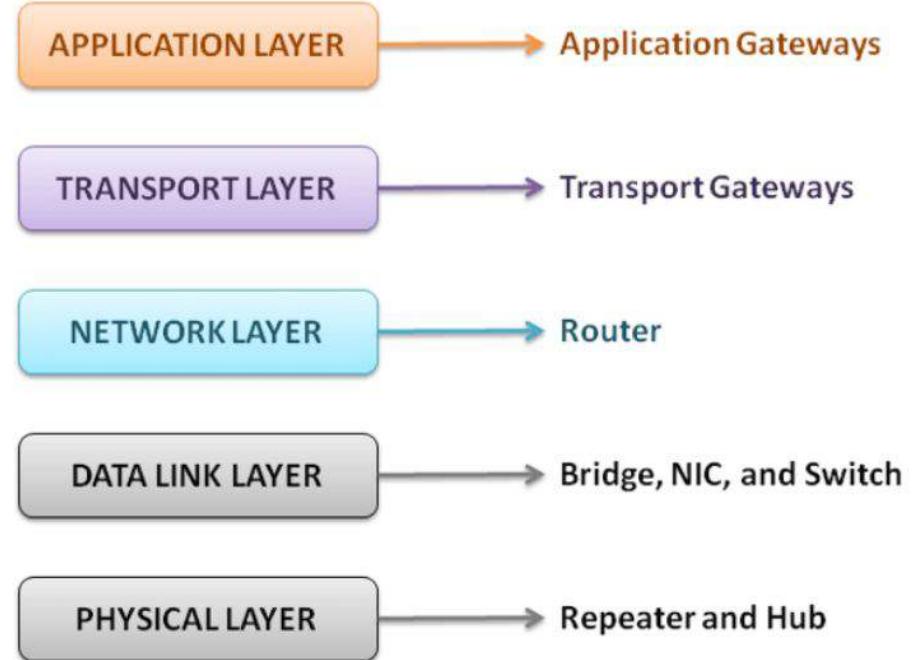
COMPUTER NETWORKS

Encapsulation – Data Communication in Protocol Stack



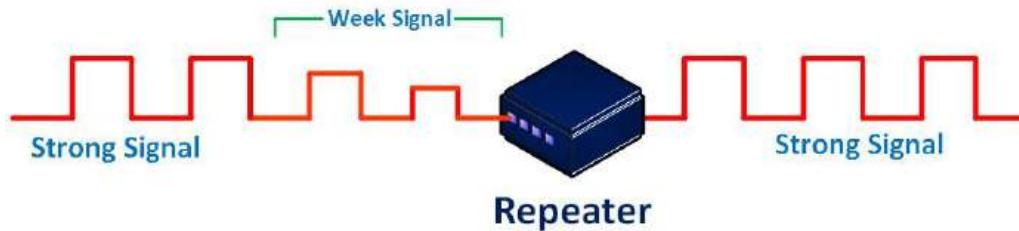
Encapsulation & Decapsulation





Repeater

- Regenerate the signal over the same network before the signal becomes too weak or corrupted to **extend** the length to which the signal can be transmitted over the same network
- Works at the Physical Layer



Working of a Repeater:-

1. Signal Reception: Captures incoming data signal (electrical voltages or electromagnetic waves).
2. Signal Amplification: **Strengthens** weakened signals due to attenuation and interference.
3. Signal Regeneration: Reconstructs the signal to ensure data **integrity** and network compliance.
4. Signal Transmission: Sends the amplified and regenerated signal to its target, minimizing further deterioration.

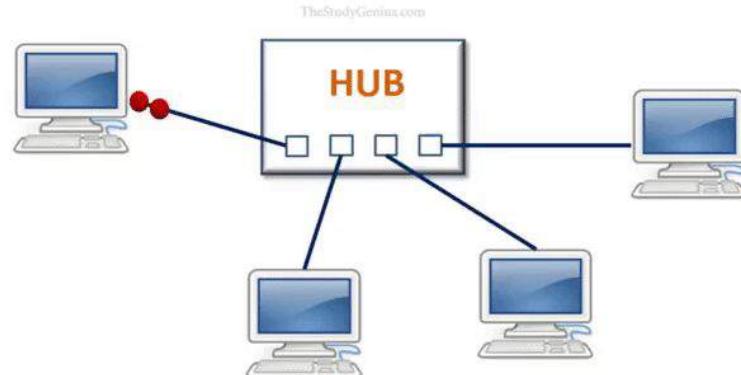
Introduction to Network Devices : Hub

Hub

- Basic networking device that is used to connect multiple devices together on a local network.
- It shares bandwidth among all connected devices and broadcasts data sent by one device to all others, giving each device equal access to the network
- Operates at Physical Layer

Features of a Hub:-

1. Always Broadcast: Hubs broadcast data to all connected devices, regardless of the intended recipient.
2. Half-duplex Communication: Only one device can transmit at once.
3. Cannot Store MAC Addresses: Hubs lack intelligence and cannot store MAC addresses or maintain tables.
4. No Data Filtering: Hubs transmit all data to all connected devices without filtering.



Modem

- Device that connects a computer to the internet by converting digital signals to analog and vice versa for data transmission.
- Operates at the **Physical Layer**



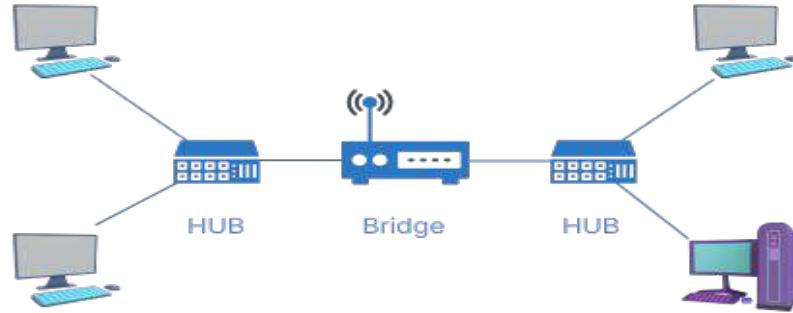
Modem Data Transmission Process:-

1. Modulation: The modem converts digital data from the computer into an analog signal for transmission.
2. Transmission: The modulated signal is sent over the communication line to the receiving modem.
3. Demodulation: The receiving modem converts the analog signal back into digital data.
4. Decoding: The digital data is then sent to the computer for use.

Introduction to Network Devices : Bridge

Bridge

- A 2-port device that **joins two or more network segments** and routes data between them according to MAC addresses
- Operates at the Data Link Layer



Functions :-

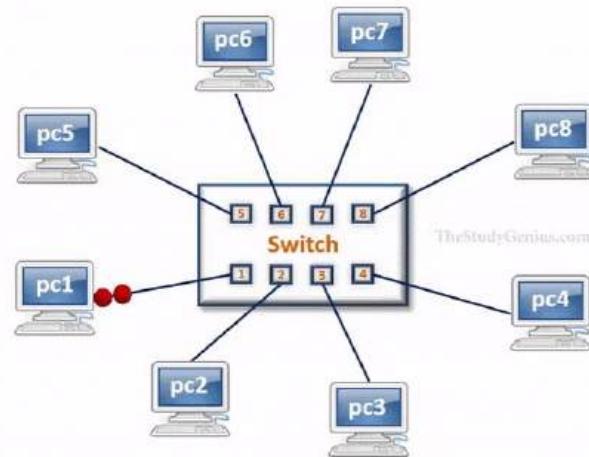
1. Segmentation: Divides a large network into smaller segments to reduce congestion and improve performance.
2. Filtering: Uses MAC addresses to **selectively forward** packets, enhancing security and bandwidth usage.
3. Learning: Tracks MAC addresses and port mappings to optimize packet forwarding.
4. Forwarding: Directs packets between segments, reducing unnecessary traffic and ensuring proper delivery.

Switch

- An intelligent network device, functioning as a multiport bridge, uses MAC addresses to forward data packets to the appropriate destination ports.
- Operates at Data Link Layer

Features of a Switch:-

1. Employs packet switching to receive and route data from source to destination.
2. Can perform some error checking before forwarding data to the destined port.
3. Can operate in full-duplex mode, enabling simultaneous data transmission & reception in network
4. Store MAC addresses of connected devices, routing communication only to the intended destination, reducing collisions and eliminating broadcast domains.



Introduction to Network Devices : NIC

NIC - Network Interface Card

- It is a physical card or chip, which contains MAC addresses, helps to identify the device on the network.
- Used at the Data Link Layer

Functions of a NIC:-

1. Connectivity: Links the device to the network (wired or wireless).
2. Data Transmission: Sends and receives data between devices.
3. Data Conversion: Converts data for transmission and reception.
4. Error Handling: Detects and corrects transmission errors.
5. Traffic Management: Prioritizes or limits data to avoid congestion.
6. Security: Provides encryption and authentication for protection.

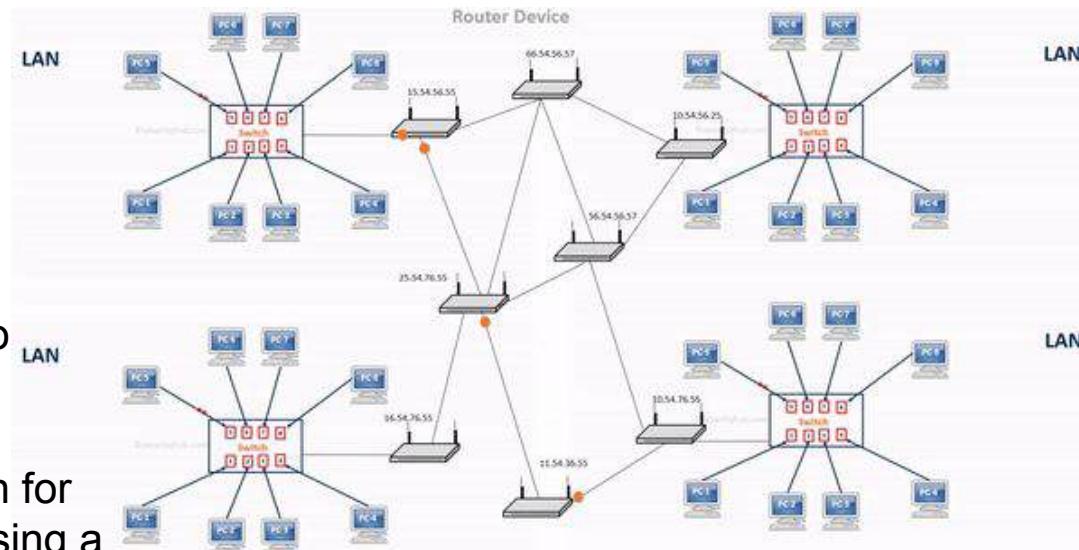


Router

- Device that routes/forwards data packets based on their IP addresses and have a dynamically updating routing table based on which they make decisions on routing the incoming packets.
- Operates at the Network Layer

Functions of a Router:-

1. Forwarding: Receives packets, checks headers, performs basic functions, and forwards packets to the appropriate output port.
2. Routing: Determines the best path for a packet to reach its destination using a routing table built with algorithms.



Router

Functions of a Router:-

3. Network Address Translation (NAT): Translates between private and public IP addresses, enabling devices on a private network to access the internet.



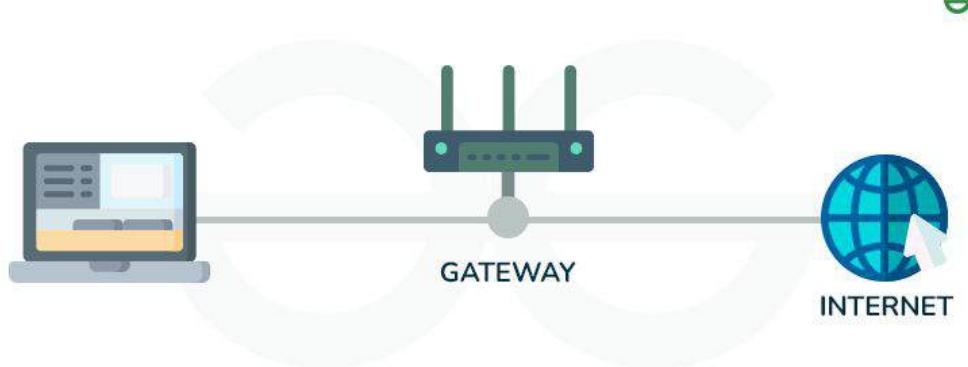
Introduction to Network Devices : Gateway

Gateway

- Network connectivity device that connects two different configuration networks
- The gateway acts as a portal between two applications via protocol communications, allowing them to share data on the same or other systems. Hence, a gateway is also a protocol converter that can operate at any OSI model layer.

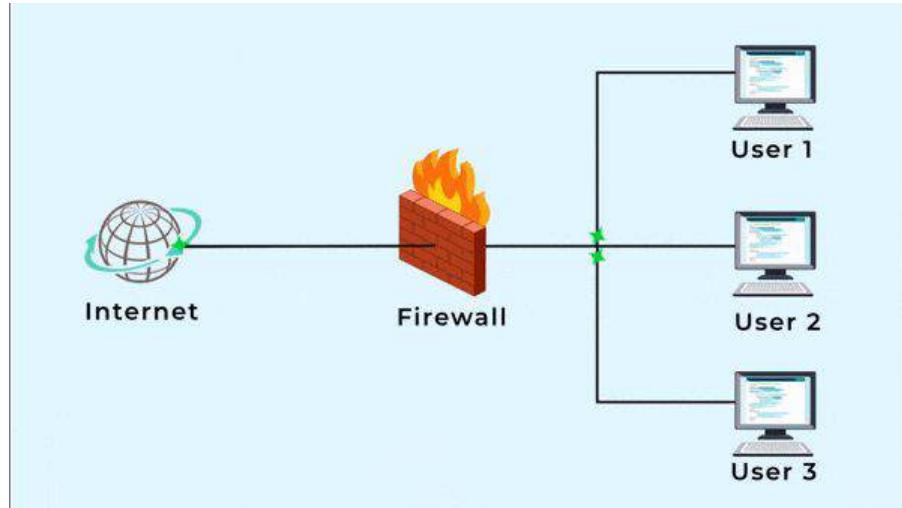
Types of Gateway:-

1. Unidirectional Gateway: Data flows in one direction, with changes replicated from source to destination only.
2. Bidirectional Gateway: Data flows in both directions, used for synchronization.



Firewall

- Network security device that monitors incoming and outgoing network traffic and decides whether to allow or block specific traffic based on a defined set of security rules
- It can be used in the transport layer or the application layer



Introduction to Network Devices

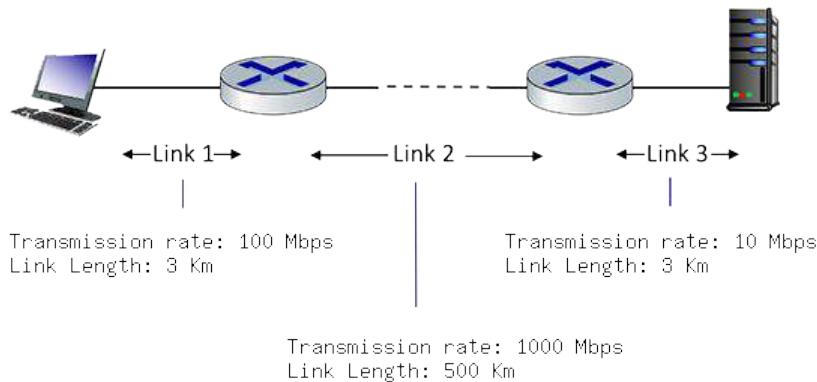
Feature	Hub	Switch
Function	Broadcasts data to all devices	Forwards data to the specific destination device based on MAC address
Layer	Operates at Physical Layer	Operates at Data Link Layer
Data Handling	Sends data to all connected devices	Sends data only to the intended device
Collision Domain	Single collision domain	Multiple collision domains (one per port)
Bandwidth	Shared bandwidth among all devices	Dedicated bandwidth for each device
Intelligence	Basic, no intelligence	More intelligent, stores MAC addresses and makes forwarding decisions
Data Filtering	No filtering, all data is broadcasted	Filters data based on MAC addresses
Usage	Suitable for small networks	Suitable for larger, more complex networks
Performance	Lower performance due to collisions	Higher performance with reduced collisions

Introduction to Network Devices

Feature	Switch	Router
Definition	A network device that connects multiple devices within the same network.	A network device that connects multiple networks and routes data between them.
Data Transmission	Uses MAC addresses to forward data within the network.	Uses IP addresses to route data between networks.
Function	Operates at Data Link Layer	Operates at Network Layer
Broadcast Domain	Typically operates within a single broadcast domain.	Breaks up broadcast domains and creates separate ones.
Usage	Used in LANs to expand network capacity.	Used to connect LANs to the internet or other LANs.
Speed	High-speed data transfer within the network.	Typically slower due to complex routing processes.

Computing end-end delay (transmission and propagation delay)

Consider the figure below, with three links, each with the specified transmission rate and link length.



Find the end-to-end delay (including the transmission delays and propagation delays on each of the three links, but ignoring queueing delays and processing delays) from when the left host begins transmitting the first bit of a packet to the time when the last bit of that packet is received at the server at the right.

The speed of light propagation delay on each link is 3×10^8 m/sec.

Note that the transmission rates are in Mbps and the link distances are in Km.

Assume a packet length of 8000 bits. Give your answer in milliseconds.

Solution:

Link 1 transmission delay = $L/R = 8000 \text{ bits} / 100 \text{ Mbps} = 80 \text{ microsec}$

Link 1 propagation delay = $d/s = 3 \text{ Km} / 3*10^{**8} \text{ m/sec} = 10 \text{ microsec}$

Link 1 total delay = $80+10 = 90 \text{ microsec}$

Link 2 transmission delay = $L/R = 8000 \text{ bits} / 1000 \text{ Mbps} = 8 \text{ microsec}$

Link 2 propagation delay = $d/s = 500 \text{ Km} / 3*10^{**8} \text{ m/sec} = 1670 \text{ microsec.}$

Link 2 total delay = $8 + 1670 = 1678 \text{ microsec}$

Link 3 transmission delay = $L/R = 8000 \text{ bits} / 10 \text{ Mbps} = 800 \text{ microsec}$

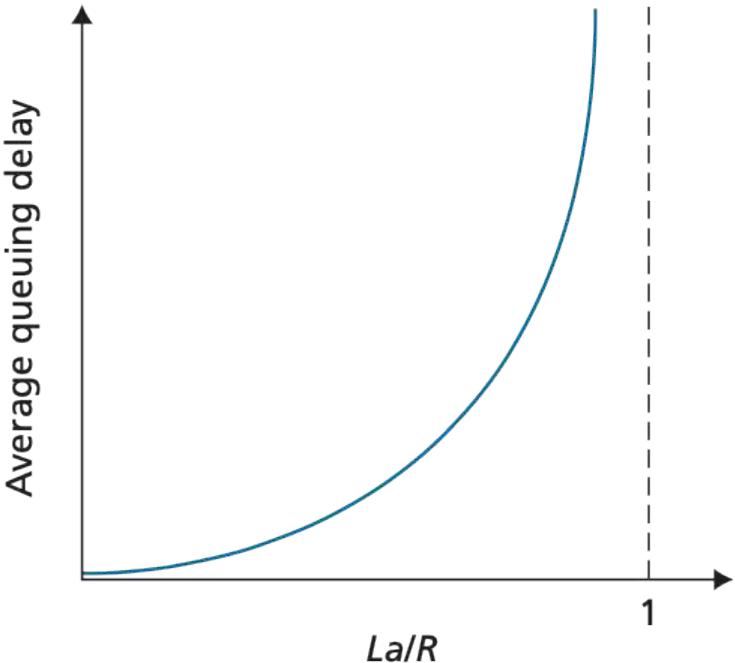
Link 3 propagation delay = $d/s = 3 \text{ Km} / 3*10^{**8} \text{ m/sec} = 10 \text{ microsec}$

Link 3 total delay = $800 + 10 = 810 \text{ microsec}$

Total end-to-end delay is the sum of these six delays: $90 + 1678 + 810 = 2.578 \text{ ms}$

Computing Queuing Delay

Consider the queuing delay in a router buffer.



Assume:

a constant transmission rate of

$$R = 1800000 \text{ bps},$$

a constant packet-length $L = 7300$ bits, and
a is the average rate of packets/second.

Traffic intensity $I = La/R$, and
Queuing delay is calculated as:

$$IL/(R(1 - I)) \text{ for } I < 1.$$

1. In practice, does the queuing delay tend to vary a lot? Answer with Yes or No

2. Assuming that $a = 30$, what is the queuing delay?

3. Assuming that $a = 76$, what is the queuing delay?

4. Assuming the router's buffer is infinite, the queuing delay is 0.8647 ms, and 1762 packets arrive. How many packets will be in the buffer 1 second later?

5. If the buffer has a maximum size of 956 packets, how many of the packets would be dropped upon arrival from the previous question?

1. Yes, in practice, queuing delay can vary significantly. We use the above formulas as a way to give a rough estimate, but in a real-life scenario it is much more complicated.

2. Queuing Delay = $\frac{IL}{R(1 - I)} * 1000 = \frac{0.1217 * 7300}{(1800000 * (1 - 0.1217))} * 1000 = 0.561$ ms.

3. Queuing Delay = $\frac{IL}{R(1 - I)} * 1000 = \frac{0.3082 * 7300}{(1800000 * (1 - 0.3082))} * 1000 = 1.806$ ms.

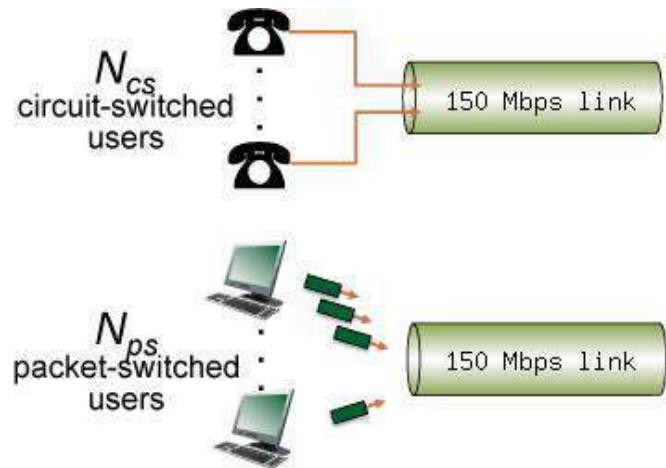
4. Packets left in buffer = $a - \text{floor}(1000/\text{delay}) = 1762 - \text{floor}(1000/0.8647) = 606$ packets.

5. Packets dropped = $\text{packets} - \text{buffer size} = 1762 - 956 = 806$ dropped packets.

Quantitative Comparison of Packet Switching and Circuit Switching

Consider the two scenarios below:

- A circuit-switching scenario in which N_{cs} users, each requiring a bandwidth of 10 Mbps, must share a link of capacity 150 Mbps.
- A packet-switching scenario with N_{ps} users sharing a 150 Mbps link, where each user again requires 10 Mbps when transmitting, but only needs to transmit 30 percent of the time.



Round your answer to two decimals after leading zeros

Answer the following questions:

1. When circuit switching is used, what is the maximum number of circuit-switched users that can be supported? Explain your answer.
2. For rest of the questions, suppose packet switching is used. Suppose there are 29 packet-switching users (i.e., $N_{ps} = 29$). Can this many users be supported under circuit-switching? Explain.
3. When one user is transmitting, what fraction of the link capacity will be used by this user? Write your answer as a decimal.

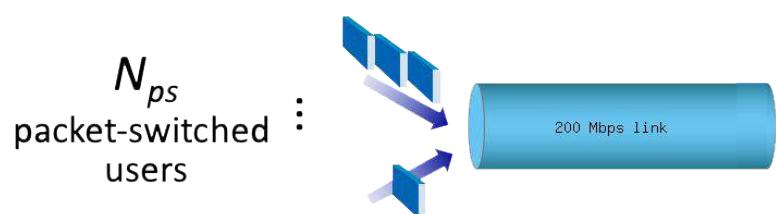
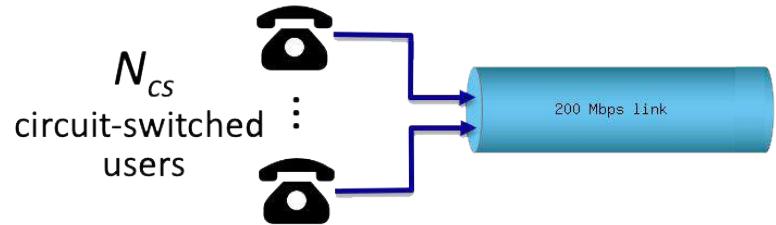
Answers:

1. 15
2. No.
3. This user will be transmitting at a rate of 10 Mbps over the 150 Mbps link, using a fraction $10/100 = \mathbf{0.06}$ of the link's capacity when busy.

Quantitative Comparison of Packet Switching and Circuit Switching

Consider the two scenarios below:

- A circuit-switching scenario in which N_{cs} users, each requiring a bandwidth of 25 Mbps, must share a link of capacity 200 Mbps.
- A packet-switching scenario with N_{ps} users sharing a 200 Mbps link, where each user again requires 25 Mbps when transmitting, but only needs to transmit 20 percent of the time.



Answer the following questions:

1. When circuit switching is used, what is the maximum number of users that can be supported?
2. Suppose packet switching is used. If there are 15 packet-switching users, can this many users be supported under circuit-switching? Yes or No.
3. When one user is transmitting, what fraction of the link capacity will be used by this user? Write your answer as a decimal.

Answers:

1. 8
2. No.
3. This user will be transmitting at a rate of 25 Mbps over the 200 Mbps link, using a fraction $25/200 = 0.125$ of the link's capacity when busy.

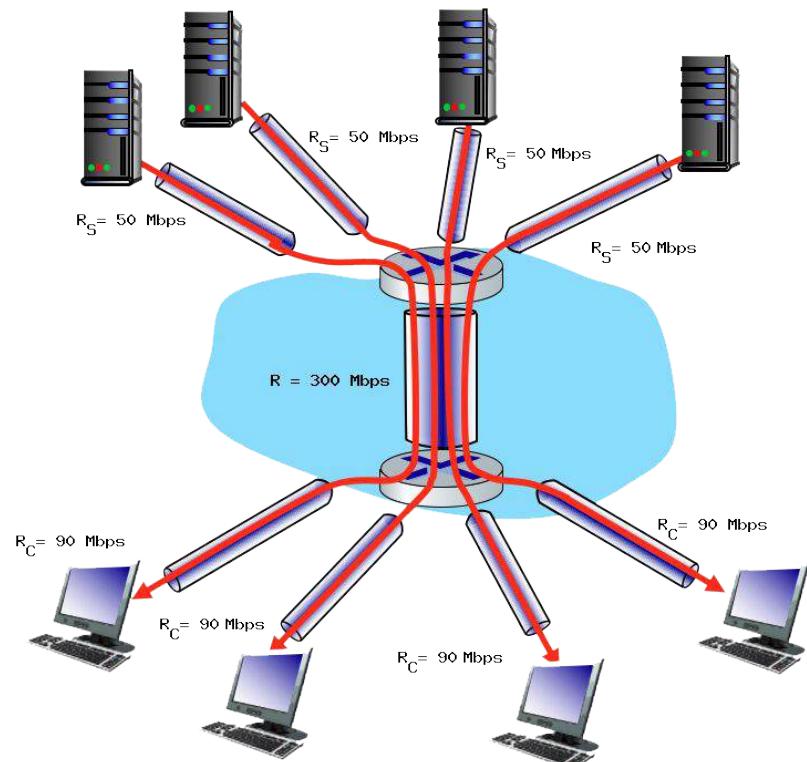
END TO END THROUGHPUT AND BOTTLENECK LINKS

Consider the scenario shown below, with four different servers connected to four different clients over four three-hop paths.

The four pairs share a common middle hop with a transmission capacity of $R = 300$ Mbps.

The four links from the servers to the shared link have a transmission capacity of $R_S = 50$ Mbps.

Each of the four links from the shared middle link to a client has a transmission capacity of $R_C = 90$ Mbps.



Answer the following questions:

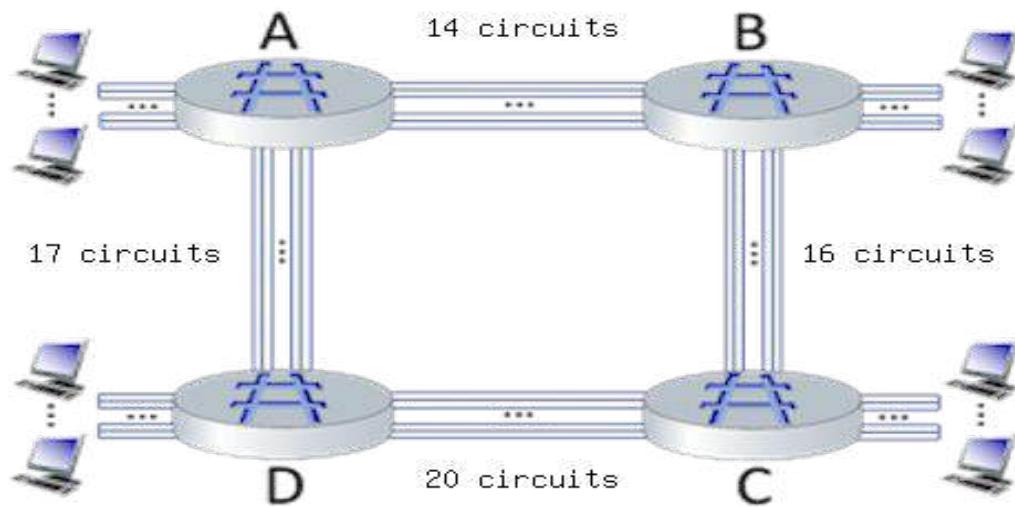
1. What is the maximum achievable end-end throughput (in Mbps) for each of four client-to-server pairs, assuming that the middle link is fairly shared (divides its transmission rate equally)?
2. Which link is the bottleneck link? Format as R_c , R_s , or R
3. Assuming that the servers are sending at the maximum rate possible, what are the link utilizations for the server links (R_s)? Answer as a decimal
4. Assuming that the servers are sending at the maximum rate possible, what are the link utilizations for the client links (R_c)? Answer as a decimal
5. Assuming that the servers are sending at the maximum rate possible, what is the link utilizations for the shared link (R)? Answer as a decimal

Solutions:

1. The maximum achievable end-end throughput is the capacity of the link with the minimum capacity, which is 50 Mbps
2. The bottleneck link is the link with the smallest capacity between R_s , R_c , and $R/4$. The bottleneck link is R_s .
3. The server's utilization = $R_{\text{bottleneck}} / R_s = 50 / 50 = 1$
4. The client's utilization = $R_{\text{bottleneck}} / R_c = 50 / 90 = 0.56$
5. The shared link's utilization = $R_{\text{bottleneck}} / (R / 4) = 50 / (300 / 4) = 0.67$

Packet Switching vs Circuit Switching – Numerical Example

Consider the circuit-switched network shown in the figure below, with circuit switches A, B, C, and D. Suppose there are 14 circuits between A and B, 16 circuits between B and C, 20 circuits between C and D, and 17 circuits between D and A



Packet Switching vs Circuit Switching – Numerical Example

Question List

1. What is the maximum number of connections that can be ongoing in the network at any one time?

2. Suppose that these maximum number of connections are all ongoing. What happens when another call connection request arrives to the network, will it be accepted? Answer Yes or No

3. Suppose that every connection requires 2 consecutive hops, and calls are connected clockwise. For example, a connection can go from A to C, from B to D, from C to A, and from D to B. With these constraints, what is the maximum number of connections that can be ongoing in the network at any one time?

4. Suppose that 12 connections are needed from A to C, and 15 connections are needed from B to D. Can we route these calls through the four links to accommodate all 27 connections? Answer Yes or No



Packet Switching vs Circuit Switching – Numerical Example

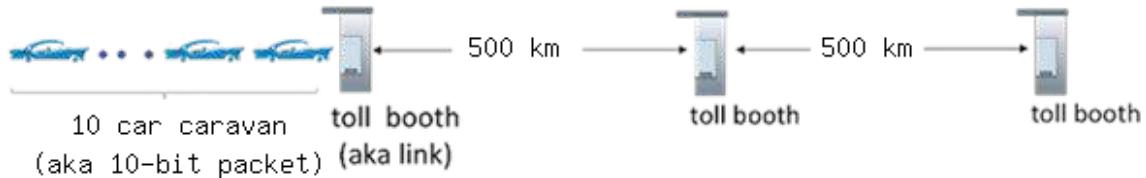
Solution

1. The maximum number of connections that can be ongoing at any one time is the sum of all circuits, which happens when 14 connections go from A to B, 16 connections go from B to C, 20 connections go from C to D, and 17 connections go from D to A. This sum is 67.
2. No, it will be blocked because there are no free circuits.
3. There can be a maximum of 33 connections. Consider routes A->C and C->A, sum the bottleneck links, consider any leftover capacity that would allow for B->D and D->B connections, and compare that value to the equivalent of B->D and D->B.
4. Using our answer from question 4, the sum of our needed connections is 27, and we have 33 available connections, so it is possible.



Car - Caravan Analogy

Consider the figure below, adapted from Figure 1.17 in the text, which draws the analogy between store-and-forward link transmission and propagation of bits in packet along a link, and cars in a caravan being serviced at a toll booth and then driving along a road to the next tollbooth.



Suppose the caravan has 10 cars, and that the tollbooth services (that is, transmits) a car at a rate of one car per 2 seconds. Once receiving serving a car proceeds to the next tool both, which is 500 kilometers away at a rate of 10 kilometers per second. Also assume that whenever the first car of the caravan arrives at a tollbooth, it must wait at the entrance to the tollbooth until all of the other cars in its caravan have arrived, and lined up behind it before being serviced at the toll booth. (That is, the entire caravan must be stored at the tollbooth before the first car in the caravan can pay its toll and begin driving towards the next tollbooth).



Car - Caravan Analogy

Question List

1. Once a car enters service at the tollbooth, how long does it take until it leaves service?

2. How long does it take for the entire caravan to receive service at the tollbooth (that is the time from when the first car enters service until the last car leaves the tollbooth)?

3. Once the first car leaves the tollbooth, how long does it take until it arrives at the next tollbooth?

4. Once the last car leaves the tollbooth, how long does it take until it arrives at the next tollbooth?

5. Once the first car leaves the tollbooth, how long does it take until it enters service at the next tollbooth?

6. Are there ever two cars in service at the same time, one at the first toll booth and one at the second toll booth?
Answer Yes or No

7. Are there ever zero cars in service at the same time, i.e., the caravan of cars has finished at the first toll both but not yet arrived at the second tollbooth? Answer Yes or No



Car - Caravan Analogy

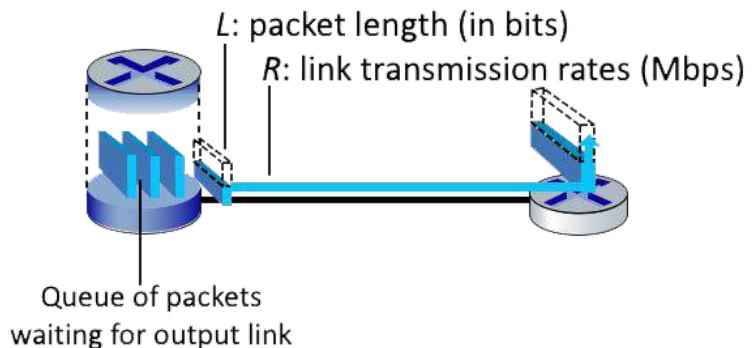
Solution

1. Service time is 2 seconds
2. It takes 20 seconds to service every car, ($10 \text{ cars} * 2 \text{ seconds per car}$)
3. It takes 50 seconds to travel to the next toll booth ($500 \text{ km} / 10 \text{ km/s}$)
4. Just like in the previous question, it takes 50 seconds, regardless of the car
5. It takes 68 seconds until the first car gets serviced at the next toll booth ($10-1 \text{ cars} * 2 \text{ seconds per car} + 500 \text{ km} / 10 \text{ km/s}$)
6. No, because cars can't get service at the next tollbooth until all cars have arrived
7. Yes, one notable example is when the last car in the caravan is serviced but is still travelling to the next toll booth; all other cars have to wait until it arrives, thus no cars are being serviced



Computing the one-hop transmission delay

Consider the figure below, in which a single router is transmitting packets, each of length L bits, over a single link with transmission rate R Mbps to another router at the other end of the link.



Suppose that the packet length is $L = 8000$ bits, and that the link transmission rate along the link to router on the right is $R = 1$ Mbps.

Round your answer to two decimals after leading zeros



Computing the one-hop transmission delay

Question List

1. What is the transmission delay?
2. What is the maximum number of packets per second that can be transmitted by this link?

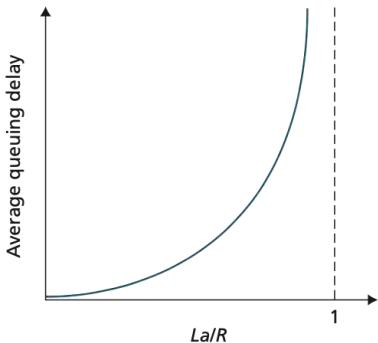
Solution

1. The transmission delay = $L/R = 8000 \text{ bits} / 1000000 \text{ bps} = 0.008 \text{ seconds}$
2. The number of packets that can be transmitted in a second into the link = $R / L = 1000000 \text{ bps} / 8000 \text{ bits} = 125 \text{ packets}$



Queuing Delay

Consider the queuing delay in a router buffer, where the packet experiences a delay as it waits to be transmitted onto the link. The length of the queuing delay of a specific packet will depend on the number of earlier-arriving packets that are queued and waiting for transmission onto the link. If the queue is empty and no other packet is currently being transmitted, then our packet's queuing delay will be zero. On the other hand, if the traffic is heavy and many other packets are also waiting to be transmitted, the queuing delay will be long.



Assume a constant transmission rate of $R = 1900000$ bps, a constant packet-length $L = 4700$ bits, and a is the average rate of packets/second. Traffic intensity $I = La/R$, and the queuing delay is calculated as $\frac{IL}{R(1 - I)}$ for $I < 1$.



Queuing Delay

Question List

1. In practice, does the queuing delay tend to vary a lot? Answer with Yes or No
2. Assuming that $a = 35$, what is the queuing delay? Give your answer in milliseconds (ms)
3. Assuming that $a = 77$, what is the queuing delay? Give your answer in milliseconds (ms)
4. Assuming the router's buffer is infinite, the queuing delay is 0.3815 ms, and 613 packets arrive. How many packets will be in the buffer 1 second later?
5. If the buffer has a maximum size of 627 packets, how many of the 613 packets would be dropped upon arrival from the previous question?



Queuing Delay

Solution

1. Yes, in practice, queuing delay can vary significantly. We use the above formulas as a way to give a rough estimate, but in a real-life scenario it is much more complicated.

$$I = La/R = 4700*35/1900000 = 0.865$$

$$2. \text{ Queuing Delay} = \frac{IL}{R(1 - I)} * 1000 = 0.0865*4700/(1900000*(1-0.0865)) * 1000 = 0.23 \text{ ms.}$$

$$I = La/R = 4700*77/1900000 = 0.19$$

$$3. \text{ Queuing Delay} = \frac{IL}{R(1 - I)} * 1000 = 0.19*4700/(1900000*(1-0.19)) * 1000 = 0.58 \text{ ms.}$$

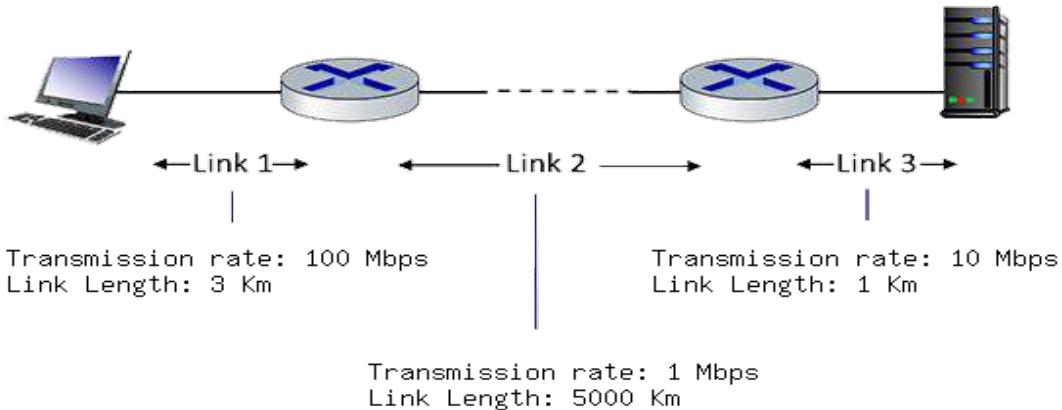
$$4. \text{ Packets left in buffer} = a - \text{floor}(1000/delay) = 613 - \text{floor}(1000/0.3815) = 0 \text{ packets.}$$

$$5. \text{ Packets dropped} = \text{packets} - \text{buffer size} = 613 - 627 \text{ hence } 0 \text{ dropped packets.}$$



Computing end-end delay (transmission and propagation delay)

Consider the figure below, with three links, each with the specified transmission rate and link length.



Assume the length of a packet is 12000 bits. The speed of light propagation delay on each link is 3×10^8 m/sec

Round your answer to two decimals after leading zeros



Computing end-end delay (transmission and propagation delay)

Question List

1. What is the transmission delay of link 1?
2. What is the propagation delay of link 1?
3. What is the total delay of link 1?
4. What is the transmission delay of link 2?
5. What is the propagation delay of link 2?
6. What is the total delay of link 2?
7. What is the transmission delay of link 3?
8. What is the propagation delay of link 3?
9. What is the total delay of link 3?
10. What is the total delay?



Computing end-end delay (transmission and propagation delay)

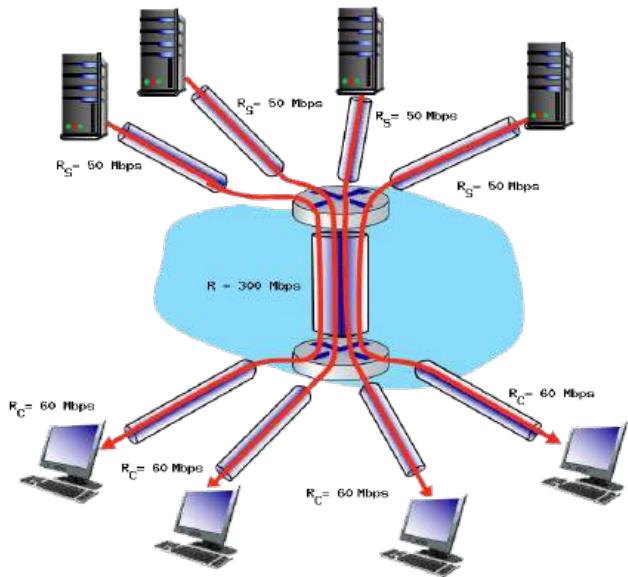
Solution

1. Link 1 transmission delay = $L/R = 12000 \text{ bits} / 100 \text{ Mbps} = 0.00012 \text{ seconds}$
2. Link 1 propagation delay = $d/s = (3 \text{ Km}) * 1000 / 3*10^8 \text{ m/sec} = 1.00E-5 \text{ seconds}$
3. Link 1 total delay = $d_t + d_p = 0.00012 \text{ seconds} + 1.00E-5 \text{ seconds} = 0.00013 \text{ seconds}$
4. Link 2 transmission delay = $L/R = 12000 \text{ bits} / 1 \text{ Mbps} = 0.012 \text{ seconds}$
5. Link 2 propagation delay = $d/s = (5000 \text{ Km}) * 1000 / 3*10^8 \text{ m/sec} = 0.017 \text{ seconds}$
6. Link 2 total delay = $d_t + d_p = 0.012 \text{ seconds} + 0.017 \text{ seconds} = 0.029 \text{ seconds}$
7. Link 3 transmission delay = $L/R = 12000 \text{ bits} / 10 \text{ Mbps} = 0.0012 \text{ seconds}$
8. Link 3 propagation delay = $d/s = (1 \text{ Km}) * 1000 / 3*10^8 \text{ m/sec} = 3.33E-6 \text{ seconds}$
9. Link 3 total delay = $d_t + d_p = 0.0012 \text{ seconds} + 3.33E-6 \text{ seconds} = 0.0012 \text{ seconds}$
10. The total delay = $d_{L1} + d_{L2} + d_{L3} = 0.00013 \text{ seconds} + 0.029 \text{ seconds} + 0.0012 \text{ seconds} = 0.03 \text{ seconds}$



End to End Throughput and Bottleneck Links

Consider the scenario shown below, with four different servers connected to four different clients over four three-hop paths. The four pairs share a common middle hop with a transmission capacity of $R = 300$ Mbps. The four links from the servers to the shared link have a transmission capacity of $R_s = 50$ Mbps. Each of the four links from the shared middle link to a client has a transmission capacity of $R_c = 60$ Mbps.



End to End Throughput and Bottleneck Links

Question List

1. What is the maximum achievable end-end throughput (in Mbps) for each of four client-to-server pairs, assuming that the middle link is fairly shared (divides its transmission rate equally)?
2. Which link is the bottleneck link? Format as R_c , R_s , or R
3. Assuming that the servers are sending at the maximum rate possible, what are the link utilizations for the server links (R_s)? Answer as a decimal
4. Assuming that the servers are sending at the maximum rate possible, what are the link utilizations for the client links (R_c)? Answer as a decimal
5. Assuming that the servers are sending at the maximum rate possible, what is the link utilizations for the shared link (R)? Answer as a decimal



End to End Throughput and Bottleneck Links

Solution

1. The maximum achievable end-end throughput is the capacity of the link with the minimum capacity, which is 50 Mbps
2. The bottleneck link is the link with the smallest capacity between R_s , R_c , and $R/4$. The bottleneck link is R_s .
3. The server's utilization = $R_{\text{bottleneck}} / R_s = 50 / 50 = 1$
4. The client's utilization = $R_{\text{bottleneck}} / R_c = 50 / 60 = 0.83$
5. The shared link's utilization = $R_{\text{bottleneck}} / (R / 4) = 50 / (300 / 4) = 0.67$



Application Layer: Overview

Our goals:

- Conceptual *and* implementation aspects of application-layer protocols
 - transport-layer service models
 - client-server paradigm
 - peer-to-peer paradigm
- Learn about protocols by examining popular application-layer protocols
 - HTTP

Some Network Apps

- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video
(YouTube, Hulu, Netflix)
- P2P file sharing
- voice over IP
- real-time video conferencing
(e.g., Skype, Hangouts)
- Internet search
- remote login
- ...

COMPUTER NETWORKS

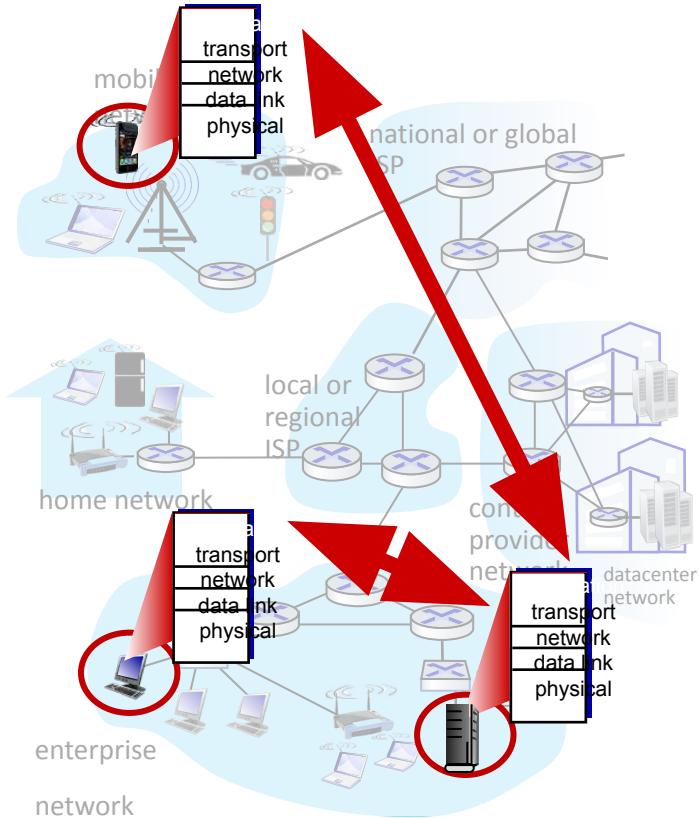
Creating a Network App

write programs that:

- run on (different) end systems
- communicate over network
- e.g., web server software communicates with browser software

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation

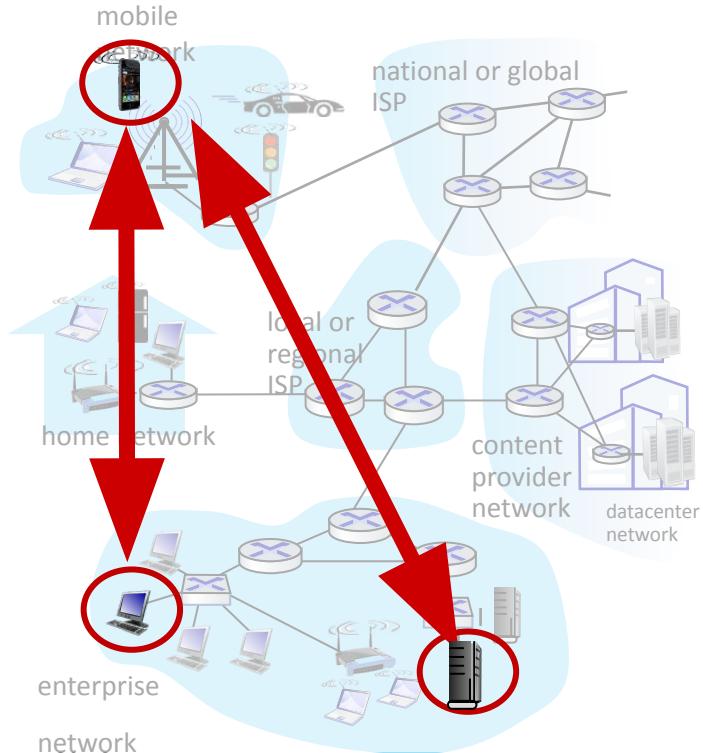


server:

- always-on host
- permanent IP address
- often in data centers, for scaling

clients:

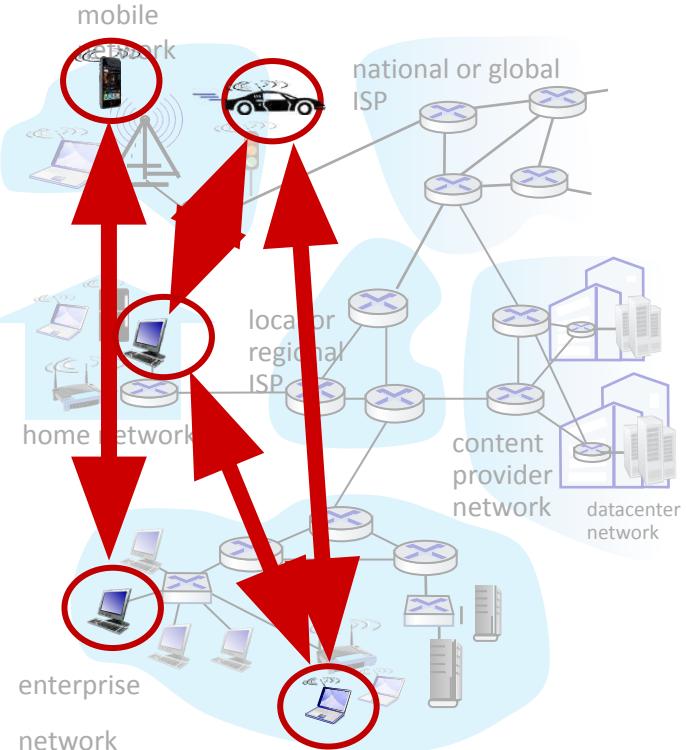
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other
- examples: HTTP, IMAP, FTP



COMPUTER NETWORKS

Peer-to-Peer Architecture

- no always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
 - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
 - complex management
- example: P2P file sharing



Processes Communicating

process: program running within a host

- within same host, two processes communicate using **inter-process communication** (defined by OS)
- processes in different hosts communicate by exchanging **messages**

clients, servers

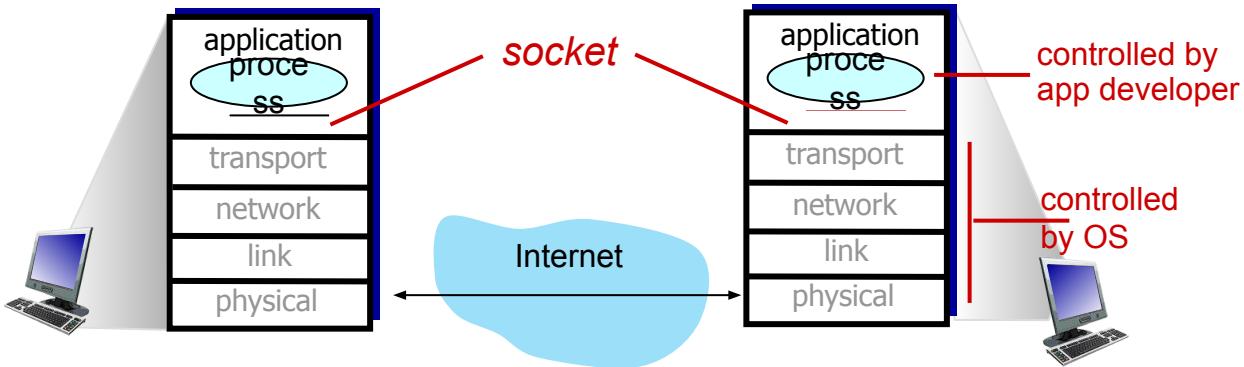
client process: process that initiates communication

server process: process that waits to be contacted

- note: applications with P2P architectures have client processes & server processes

Sockets

- process sends/receives messages to/from its **socket**
- socket analogous to door
 - sending process shoves message out door
 - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
 - two sockets involved: one on each side



Addressing Processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
 - A: no, *many* processes can be running on same host
- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
 - HTTP server: 80
 - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
 - IP address: 128.119.245.12
 - port number: 80
- more shortly...

An Application-layer Protocol defines:

- types of messages exchanged,
 - e.g., request, response
- message syntax:
 - what fields in messages & how fields are delineated
- message semantics
 - meaning of information in fields
- rules for when and how processes send & respond to messages

open protocols:

- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype

What transport service does an App needs?

data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be “effective”
- other apps (“elastic apps”) make use of whatever throughput they get

security

- encryption, data integrity, ...

Transport service requirements: common apps

application	data loss	throughput	time sensitive?
file transfer/download	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kbps-1Mbps video:10Kbps-5Mbps	yes, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	yes and no

TCP service:

- ***reliable transport*** between sending and receiving process
- ***flow control***: sender won't overwhelm receiver
- ***congestion control***: throttle sender when network overloaded
- ***does not provide***: timing, minimum throughput guarantee, security
- ***connection-oriented***: setup required between client and server processes

UDP service:

- ***unreliable data transfer*** between sending and receiving process
- ***does not provide***: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Internet Transport Protocol Services

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP 1.1 [RFC 7320]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary (Skype)	TCP or UDP
streaming audio/video	DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

First, a quick review...

- web page consists of *objects*, each of which can be stored on different Web servers
- object can be HTML file, JPEG image, Java applet, audio file,...
- web page consists of *base HTML-file* which includes *several referenced objects, each* addressable by a *URL*, e.g.,
- If a Web page contains HTML text and 5 JPEG images, then the Web page has 6 objects: the base HTML file plus the 5 images.

www.someschool.edu/someDept/pic.gif

host name

path name

HTTP Overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model:
 - *client*: browser that requests, receives, (using HTTP protocol) and "displays" Web objects
 - *server*: Web server sends (using HTTP protocol) objects in response to requests



Defined in RFC 1945; RFC 2616

HTTP uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is “stateless”

- server maintains *no* information about past client requests

aside

protocols that maintain “state” are complex!

- past history (state) must be maintained
- if server/client crashes, their views of “state” may be inconsistent, must be reconciled

HTTP Connections: two types

Non-persistent HTTP

1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

downloading multiple objects required multiple connections

Persistent HTTP

- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed

Non-persistent HTTP: example

User enters URL: **www.someSchool.edu/someDepartment/home.index**
(base HTML file containing text, references to 10 jpeg images)



1a. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on port 80



1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80 “accepts” connection, notifying client

time
↓

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket.
Message indicates that client wants object someDepartment/home.index



3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

Non-persistent HTTP: example (more)

User enters URL: `www.someSchool.edu/someDepartment/home.index`
(containing text, references to 10 jpeg images)



5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

4. HTTP server closes TCP connection.



6. Steps 1-5 repeated for each of 10 jpeg objects

time

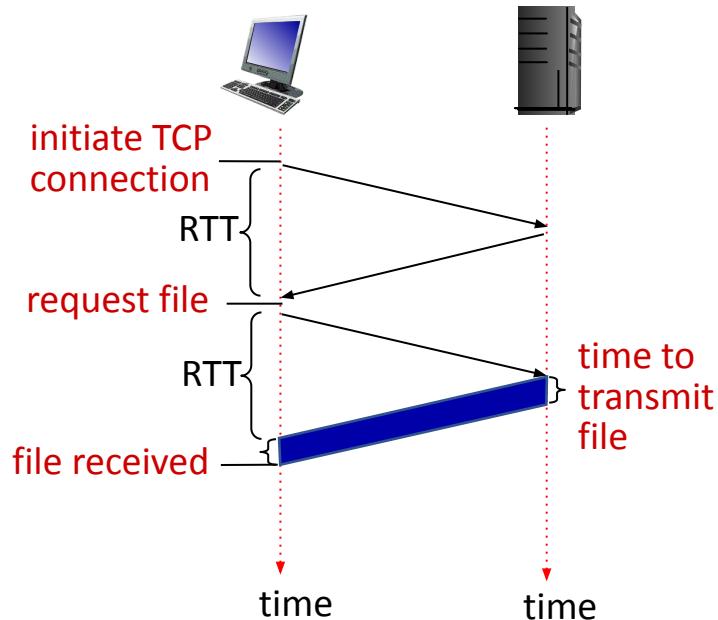


Non-persistent HTTP: response time

RTT (definition): time for a small packet to travel from client to server and back

HTTP response time (per object):

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- object/file transmission time



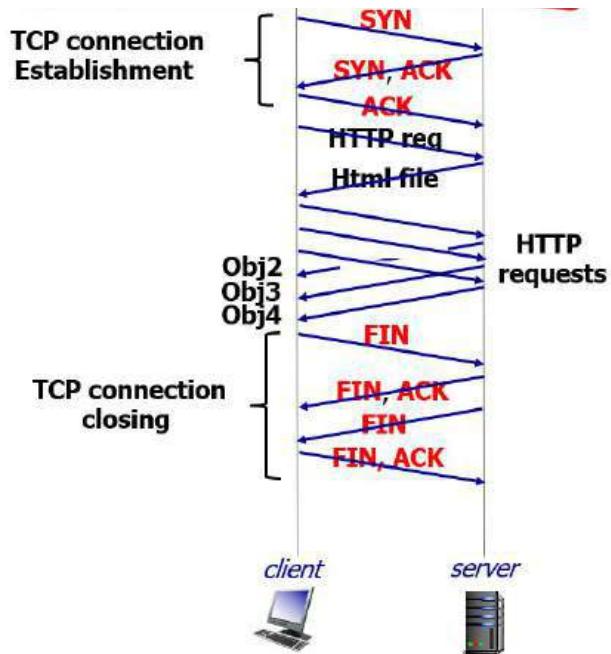
$$\text{Non-persistent HTTP response time} = 2\text{RTT} + \text{file transmission time}$$

Non-persistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection (TCP buffer and variables)
- browsers often open multiple parallel TCP connections to fetch referenced objects in parallel

Persistent HTTP (HTTP1.1):

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects (cutting response time in half)



Connection Management in HTTP/1.x

Client

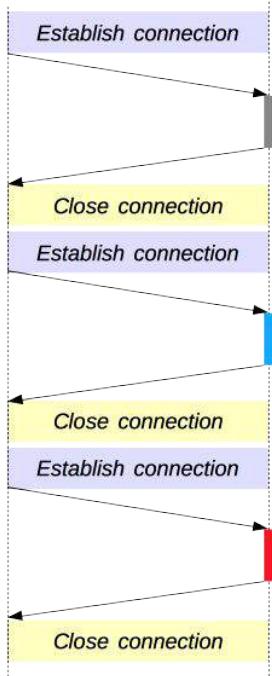
Server

Client

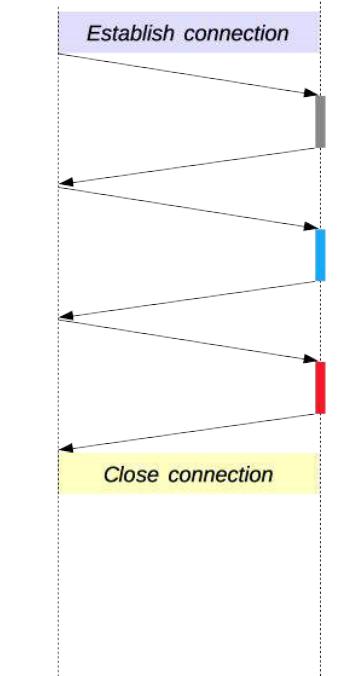
Server

Client

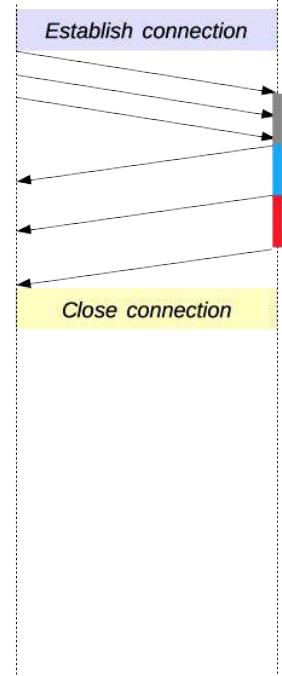
Server



Short-lived connections



Persistent connection



HTTP Pipelining

HTTP Request Message

- two types of HTTP messages: *request, response*
- **HTTP request message:**
 - ASCII (human-readable format)

request line (GET, POST,
HEAD commands)

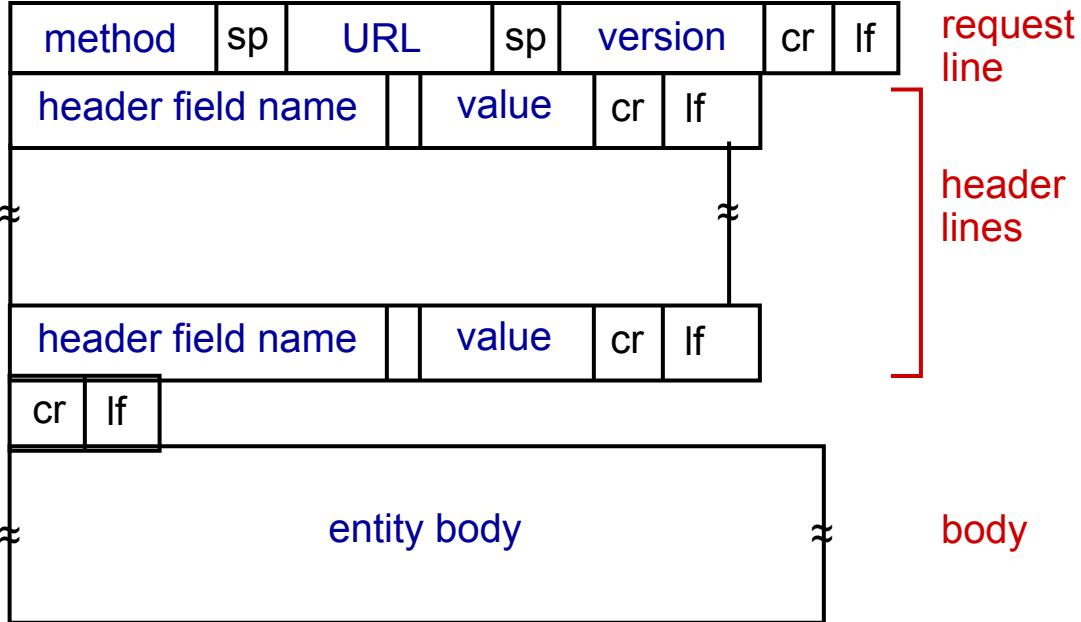
header
lines

carriage return, line feed
at start of line indicates
end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept:
    text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset:
    ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\nCheck out the online interactive exercises for more
examples: http://gaia.cs.umass.edu/kurose_ross/interactive/
```

carriage return character
line-feed character

HTTP Request Message: General Format



HTTP specifications [RFC 1945; RFC 2616; RFC 7540]

POST method:

- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

HEAD method:

- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

GET method (for sending data to server):

- include user data in URL field of HTTP GET request message (following a '?'):

PUT method:

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

www.somesite.com/animalsearch?monkeys&banana

status line (protocol
status code status phrase)

HTTP/1.1 200 OK\r\nDate: Sun, 26 Sep 2010 20:09:20 GMT\r\nServer: Apache/2.0.52 (CentOS) \r\nLast-Modified: Tue, 30 Oct 2007 17:00:02
GMT\r\n

header
lines

ETag: "17dc6-a5c-bf716880"\r\nAccept-Ranges: bytes\r\nContent-Length: 2652\r\nKeep-Alive: timeout=10, max=100\r\nConnection: Keep-Alive\r\nContent-Type: text/html;
charset=ISO-8859-1\r\n\r\ndata data data data data ...

data, e.g., requested
HTML file

HTTP Response Status Codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

200 OK

- request succeeded, requested object later in this message

301 Moved Permanently

- requested object moved, new location specified later in this message (in Location: field)

400 Bad Request

- request msg not understood by server

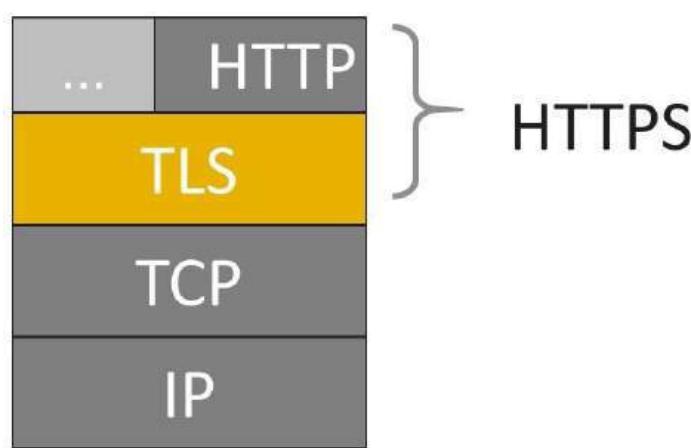
404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

HTTP vs HTTPS (more)

- HTTP + TLS -> Encrypted
- Uses port no. 443 for data communication.
- HTTPS is based on public/private-key cryptography.
 - The public key is used for encryption
 - The secret private key is required for decryption.
- SSL certificate is a web server's digital certificate issued by a third party CA.
 - Create an encrypted connection and establish trust.
- Is my certificate SSL or TLS?



Any message encrypted with Bob's public key can be only decrypted with Bob's private key.

How does SSL works?

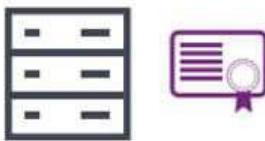
- Step 1: Browser requests secure pages (HTTPS) from a server.
- Step 2: Server sends its public key with its SSL certificate (digitally signed by a third party – CA).
- Step 3: On receipt of certificate, browser verifies issuer's digital signature. (green padlock key)
- Step 4: Browser creates a symmetric key (shared key), keeps one and gives a copy to server. Encrypts it using server's public key.
- Step 5: On receipt of encrypted secret key, decrypts it using its private key and gets browser's secret key.
- Asymmetric and Symmetric key algorithms work together.
- Asymmetric key algorithm – verify identity of the owner & its public key -> Establish trust.
- Once connection is established, Symmetric key algorithm is used to encrypt and decrypt the traffic.

COMPUTER NETWORKS

How does SSL works?



Client messages server to initiate SSL/TLS communication



Server sends back an encrypted public key/certificate.



Client checks the certificate, creates and sends an encrypted key back to the server
(If the certificate is not ok, the communication fails)

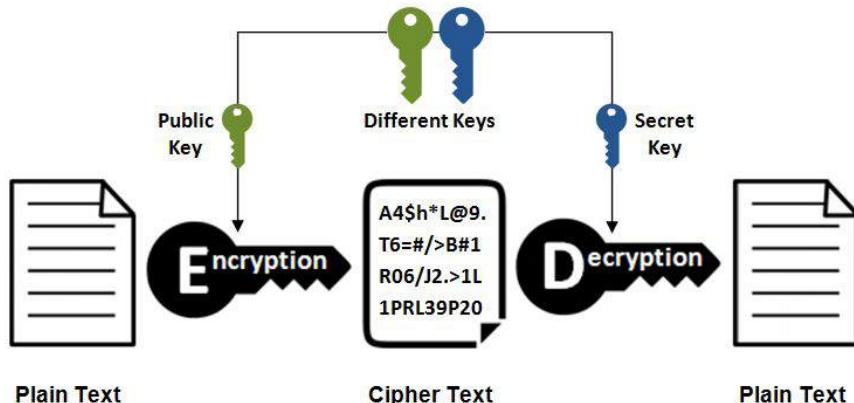


Server decrypts the key and delivers encrypted content with key to the client

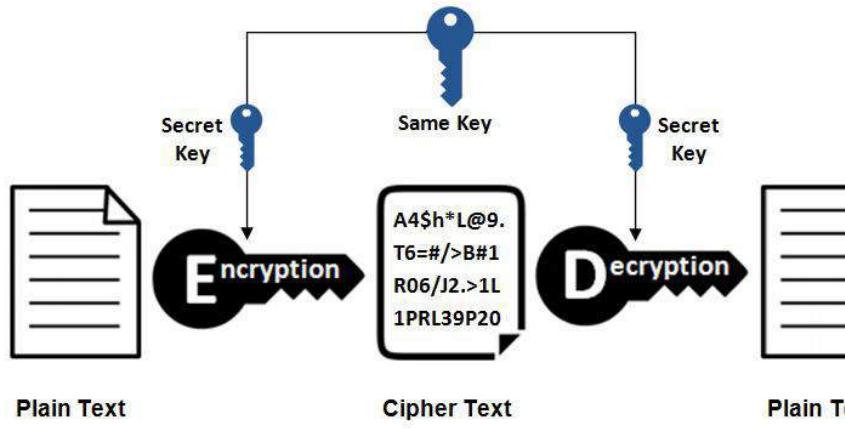


Client decrypts the content completing the SSL/TLS handshake

Asymmetric Encryption



Symmetric Encryption

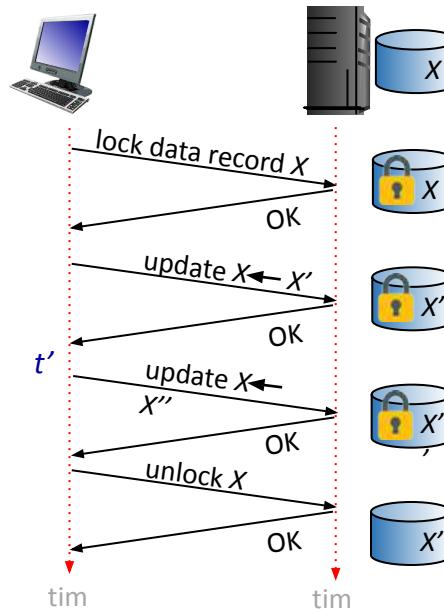


Maintaining user/server state: cookies

Recall: HTTP GET/response interaction is *stateless*

- no notion of multi-step exchanges of HTTP messages to complete a Web “transaction”
 - no need for client/server to track “state” of multi-step exchange
 - all HTTP requests are independent of each other
 - no need for client/server to “recover” from a partially-completed-but-never-completely-completed transaction

a **stateful protocol**: client makes two changes to X, or none at all



Q: what happens if network connection or client crashes at t' ?

Maintaining user/server state: cookies (more)

Web sites and client browser use *cookies* to maintain some state between transactions

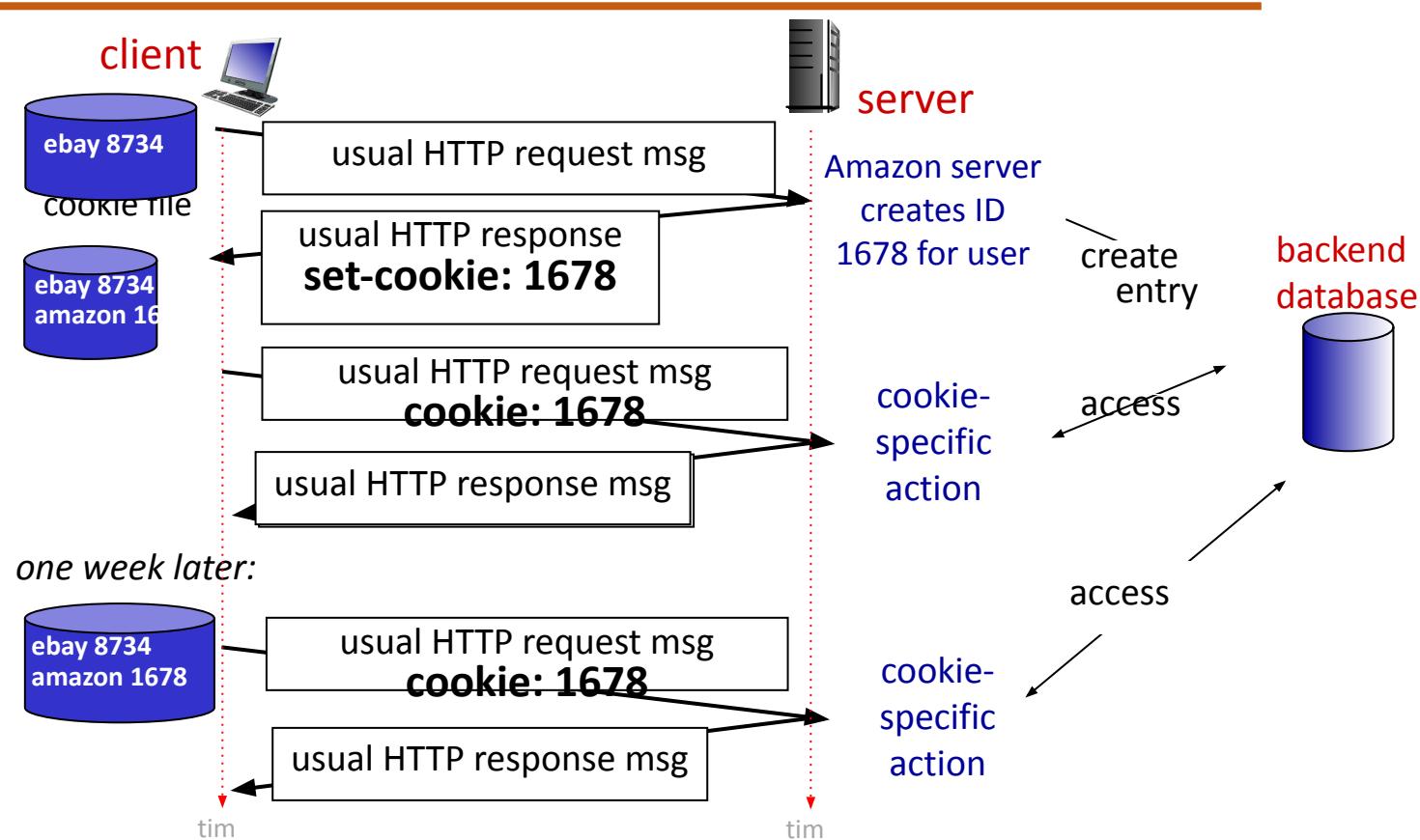
four components:

- 1) cookie header line of HTTP *response* message
- 2) cookie header line in next HTTP *request* message
- 3) cookie file kept on user's host, managed by user's browser
- 4) back-end database at Web site

Example:

- Susan uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
 - unique ID (aka “cookie”)
 - entry in backend database for ID
- subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to “identify” Susan

Maintaining user/server state: cookies (more)



What cookies can be used for:

- track user's browsing history
- remembering login details
- track visitor count
- shopping carts
- recommendations
- save coupon codes for you

Challenge: How to keep state:

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: HTTP messages carry state

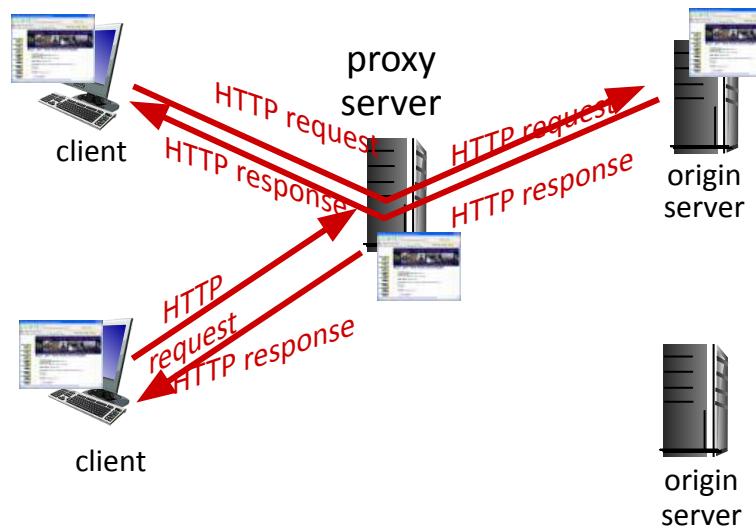
aside

cookies and privacy:

- cookies permit sites to *learn* a lot about you on their site.
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites

Goal: satisfy client request without involving origin server

- user configures browser to point to a *Web cache*
- browser sends all HTTP requests to cache
 - *if* object in cache: cache returns object to client
 - *else* cache requests object from origin server, caches received object, then returns object to client



Web Caches (Proxy Servers)

- Web cache acts as both client and server
 - server for original requesting client
 - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

Why Web caching?

- reduce response time for client request (speed)
 - cache is closer to client
- reduce traffic on an institution's access link (saves bandwidth)
- internet is dense with caches
 - enables "poor" content providers to more effectively deliver content
- privacy – surf the internet anonymously
- activity logging

Caching example

$$(15 \text{ req/sec}) * (100 \text{ Kbits/req}) / (1.54 \text{ Mbps}) = 0.974$$

$$(15 \text{ req/sec}) * (100 \text{ Kbits/req}) / (1 \text{ Gbps}) = 0.0015$$

Scenario:

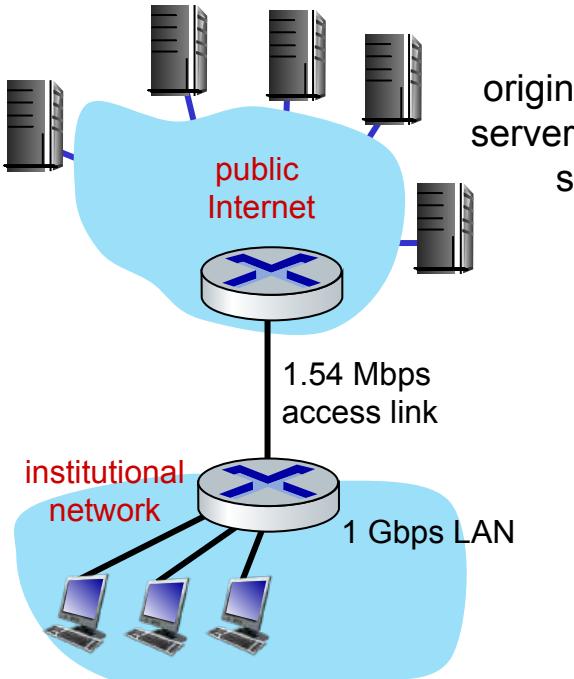
- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Average request rate from browsers to origin servers: 15/sec
 - average data rate to browsers: 1.50 Mbps

problem: large delays at high utilization!

Performance:

- LAN utilization: .0015
- access link utilization = .97
- end-end delay = Internet delay + access link delay + LAN delay

$$= 2 \text{ sec} + \text{minutes} + \text{usecs}$$



COMPUTER NETWORKS

Caching example: buy a faster access link

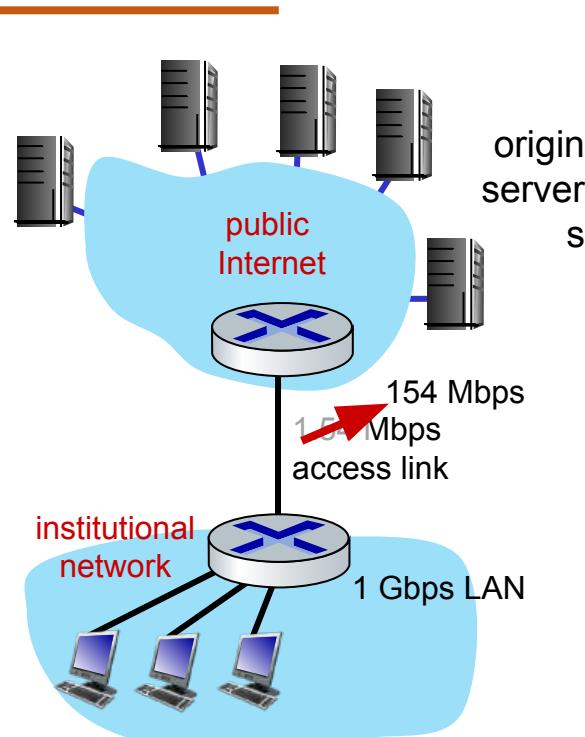
Scenario:

- access link rate: 154 Mbps 
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15/sec
 - avg data rate to browsers: 1.50 Mbps

Performance:

- LAN utilization: .0015
- access link utilization = ~~.07~~  .0097
- end-end delay = Internet delay +
access link delay + LAN delay
= 2 sec + minutes + usecs

Cost: faster access link (expensive!)  msecs



Caching example: install a web cache

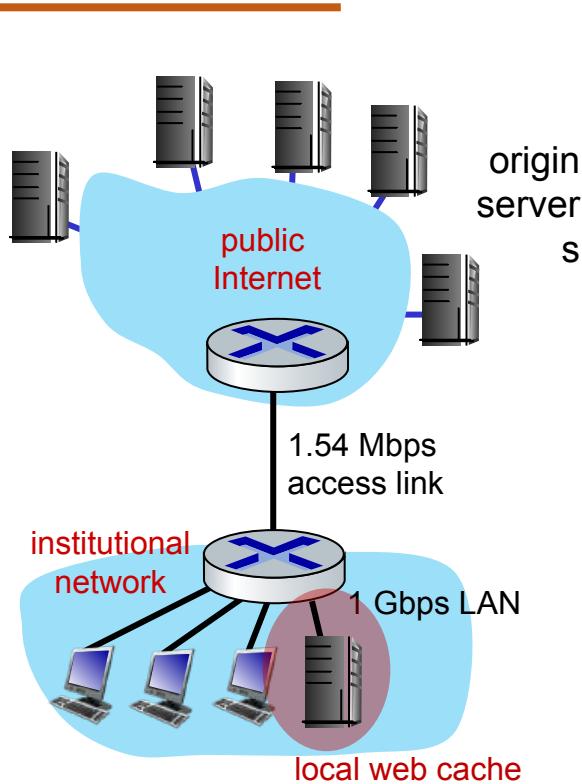
Scenario:

- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15/sec
 - avg data rate to browsers: 1.50 Mbps

Performance: *How to compute link utilization, delay?*

- LAN utilization: .? *utilization, delay?*
- access link utilization = ?
- average end-end delay = ?

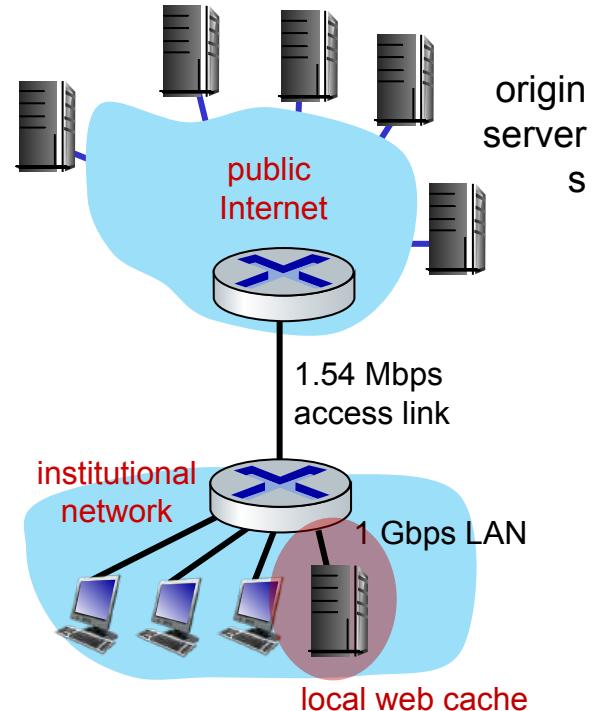
Cost: web cache (cheap!)



Caching example: install a web cache

Calculating access link utilization,
end-end delay with cache:

- suppose cache hit rate is 0.4: 40% requests satisfied at cache, 60% requests satisfied at origin
- access link: 60% of requests use access link
- data rate to browsers over access link
 - = $0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$
- utilization = $0.9/1.54 = .58$
- average end-end delay
 - = $0.6 * (\text{delay from origin servers})$
 - + $0.4 * (\text{delay when satisfied at cache})$
- = $0.6 (2.01) + 0.4 (\sim \text{msecs}) = \sim 1.2 \text{ secs}$

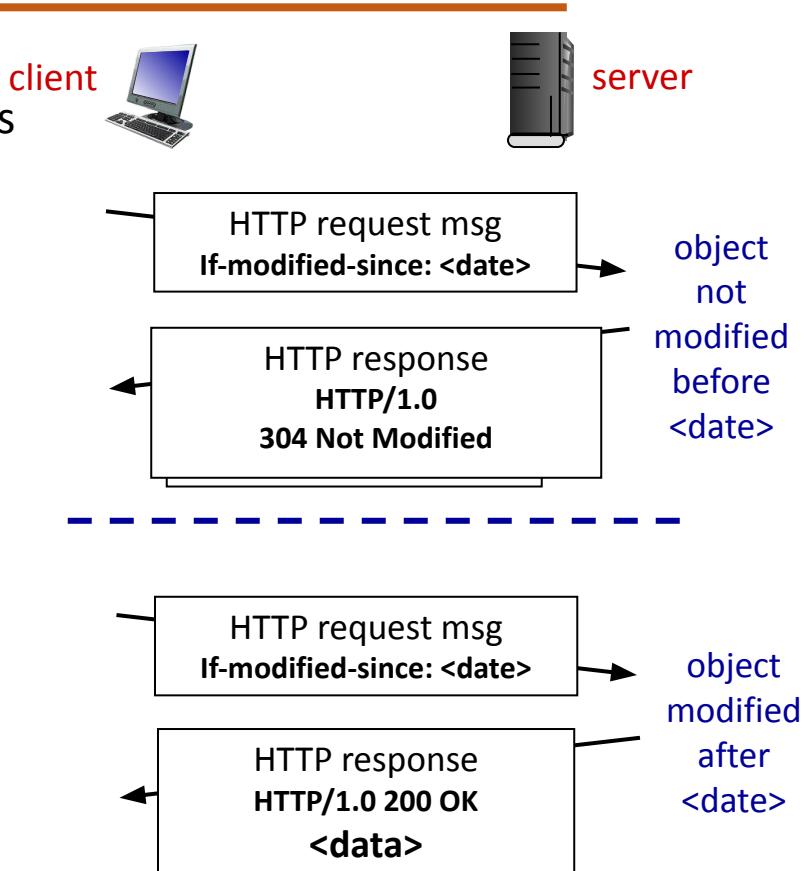


lower average end-end delay than with 154 Mbps link (and cheaper too!)

Conditional Get

Goal: don't send object if cache has up-to-date cached version

- no object transmission delay
 - lower link utilization
- **cache:** specify date of cached copy in HTTP request
If-modified-since: <date>
- **server:** response contains no object if cached copy is up-to-date:
HTTP/1.0 304 Not Modified



COMPUTER NETWORKS

Conditional Get (more)

Microsoft:\Device\NPF_{483C83F4-DCBA-4863-B523-3C4E1B03D06F} | Wireshark 1.8.5 (SVN Rev 47350 from /trunk-1.8)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

Filter: http Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	Info
4	20:33:03.198438000	10.36.40.181	149.152.32.102	HTTP	715	GET /ssp_director/p.php?a=UUFRXiQyPSEqYHt1Pz04JzU6I5s7PT4uN1o4MTI%2BNjkmky0gP5cjKdOnNz8xX
321	20:33:03.427289000	149.152.32.102	10.36.40.181	HTTP	1514	[TCP out-of-order] HTTP/1.1 200 OK (JPEG/JFIF image)
340	20:33:09.383079000	10.36.40.181	128.119.245.12	HTTP	473	GET /wireshark-labs/HTTP-wireshark-file2.html HTTP/1.1
341	20:33:09.407557000	128.119.245.12	10.36.40.181	HTTP	701	HTTP/1.1 200 OK (text/html)
343	20:33:09.677244000	10.36.40.181	128.119.245.12	HTTP	384	GET /favicon.ico HTTP/1.1
344	20:33:09.689986000	128.119.245.12	10.36.40.181	HTTP	532	HTTP/1.1 404 Not Found (text/html)
347	20:33:14.318343000	10.36.40.181	128.119.245.12	HTTP	586	GET /wireshark-labs/HTTP-wireshark-file2.html HTTP/1.1
348	20:33:14.331881000	128.119.245.12	10.36.40.181	HTTP	319	HTTP/1.1 304 Not Modified
349	20:33:14.410192000	10.36.40.181	128.119.245.12	HTTP	384	GET /favicon.ico HTTP/1.1
350	20:33:14.426443000	128.119.245.12	10.36.40.181	HTTP	532	HTTP/1.1 404 Not Found (text/html)

Ethernet II, Src: HonHaiPr_0a:de:6b (cc:af:78:0a:de:6b), Dst: Cisco_4c:61:3f (00:1e:f7:4c:61:3f)
Internet Protocol Version 4, Src: 10.36.40.181 (10.36.40.181), Dst: 128.119.245.12 (128.119.245.12)
Transmission Control Protocol, Src Port: 55404 (55404), Dst Port: http (80), Seq: 750, Ack: 1126, Len: 532
Hypertext Transfer Protocol
GET /wireshark-labs/HTTP-wireshark-file2.html HTTP/1.1\r\nHost: gaia.cs.umass.edu\r\nConnection: keep-alive\r\nCache-Control: max-age=0\r\nAccept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\nUser-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.22 (KHTML, like Gecko) Chrome/25.0.1364.97 Safari/537.22\r\nAccept-Encoding: gzip,deflate,sdch\r\nAccept-Language: en-US,en;q=0.8\r\nAccept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3\r\nIf-Modified-Since: wed, 27 feb 2013 01:33:01 gmt\r\n\r\n[Full request URI: http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html]

01d0	20	49	53	4f	2d	38	38	35	39	2d	31	2c	75	74	66	2d	ISO-8859-1,utf-
01e0	38	3b	71	3d	30	2e	37	2c	2a	3b	71	3d	30	2e	33	0d	8;q=0.7,*;q=0.3;
01f0	04	49	66	2d	4e	6f	66	65	2d	4d	61	74	63	68	3a	20	If-None-Match:
0200	22	64	38	63	39	36	2d	31	37	33	2d	63	31	33	33	36	d6c96-1 73-c1336
0210	64	34	30	22	0d	0a	49	66	2d	4d	6f	64	69	66	69	65	d40". If-Modified-
0220	64	2d	53	69	6e	63	65	3a	20	57	65	64	2c	20	32	37	d-Since: wed, 27
0230	20	46	65	62	20	32	30	31	33	20	30	31	3a	33	33	3a	Feb 2013 01:33:
0240	30	31	20	47	4d	54	0d	0a	0d	0a	01	GMT...	..				

Text item (text), 50 bytes

Packets: 367 Displayed: 10 Marked: 0 Dropped: 0

Profile: Default

people: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- “name”, e.g., cs.umass.edu - used by humans

Q: how to map between IP address and name, and vice versa ?

Domain Name System:

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol*: hosts, name servers communicate to *resolve* names (address/name translation)
 - note: core Internet function, *implemented as application-layer protocol*
 - complexity at network’s “edge”

DNS services

- hostname to IP address translation
- host aliasing
 - canonical, alias names
- mail server aliasing
- load distribution
 - replicated Web servers: many IP addresses correspond to one name

www.abc.example.com -> Canonical Host Name

www.example.com -> Alias Name

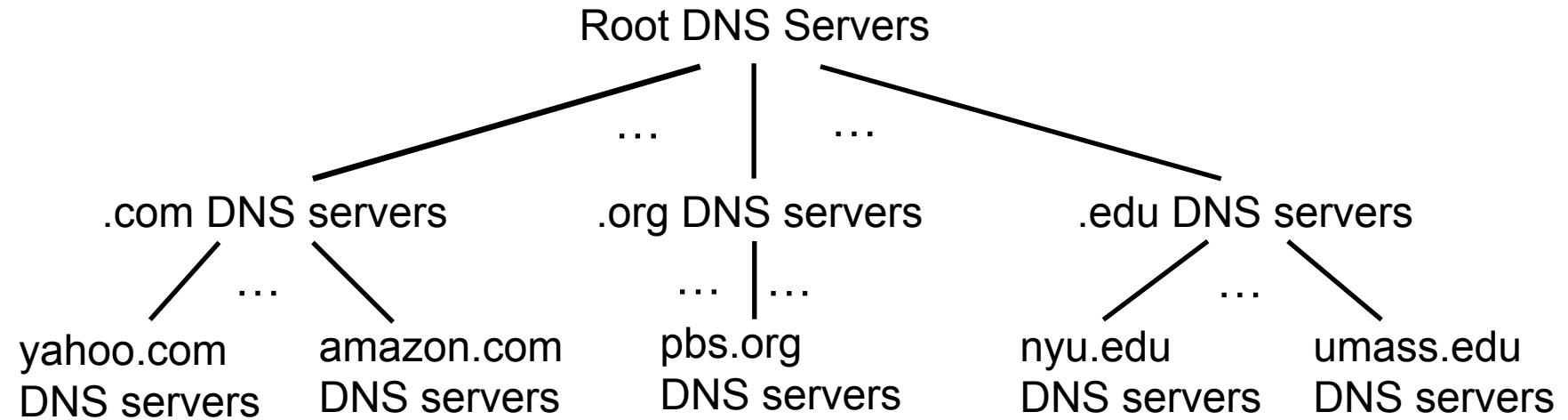
Q: Why not centralize DNS?

- single point of failure
- traffic volume
- distant centralized database
- maintenance

www.abc.example.com ->
Canonical Host Name
bob@example.com ->
Alias Name

A: doesn't scale!

- Comcast DNS servers alone:
600B DNS queries per day



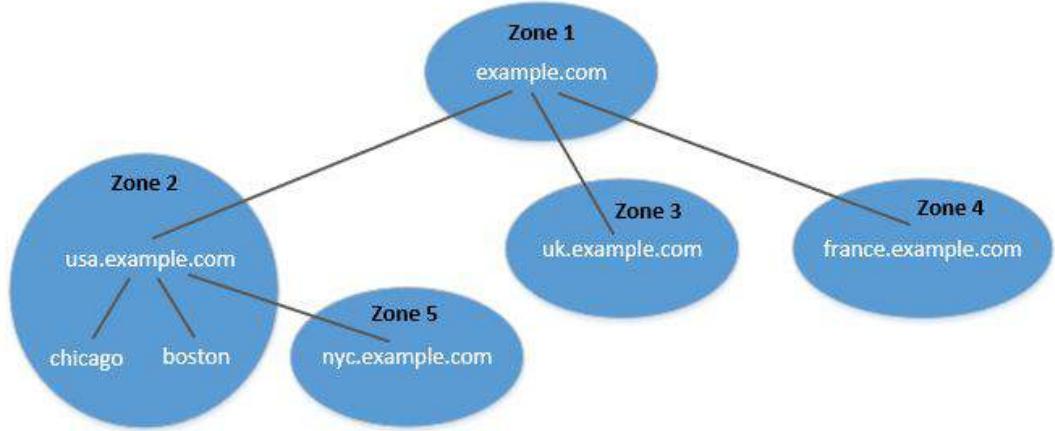
Root

Top Level Domain

Authoritative

Client wants IP address for www.amazon.com; 1st approximation:

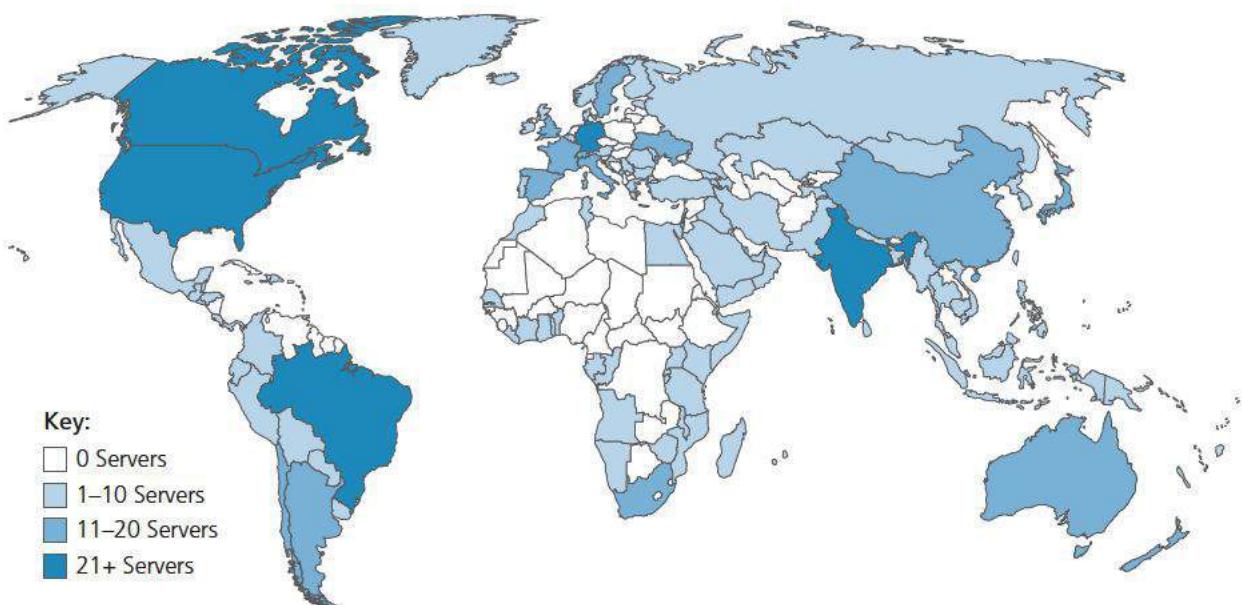
- client queries root server to find .com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com



- DNS is organized according to zones.
- A zone groups contiguous domains and subdomains on the domain tree.
- Assign management authority to an entity.
- The tree structure depicts subdomains within example.com domain.
- Multiple DNS zones one for each country. The zone keeps records of who the authority is for each of its subdomains.
- The zone for example.com contains only the DNS records for the hostnames that do not belong to any subdomain like mail.example.com

- official, contact-of-last-resort by name servers that can not resolve name
- *incredibly important* Internet function
 - Internet couldn't function without it!
 - DNSSEC – provides security (authentication and message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

13 logical root name “servers” worldwide each “server” replicated many times (~200 servers in US)



Top-Level Domain (TLD) servers:

- responsible for .com, .org, .net, .edu, .aero, .jobs, .museums, and all top-level country domains, e.g.: .cn, .uk, .fr, .ca, .jp
- Network Solutions: authoritative registry for .com, .net TLD
- Educause: .edu TLD

Authoritative DNS servers:

- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

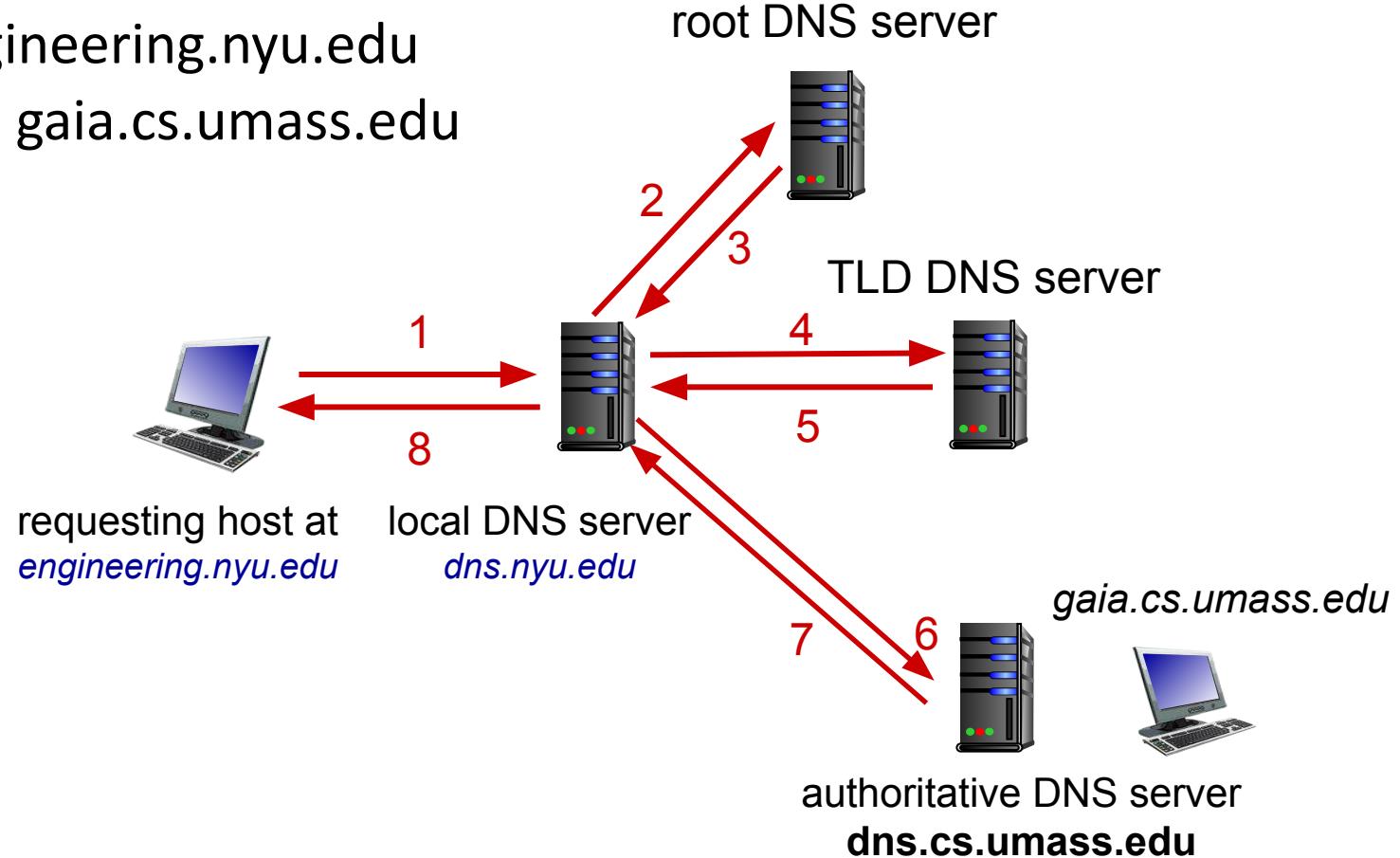
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
 - also called “default name server”
- when host makes DNS query, query is sent to its local DNS server
 - has local cache of recent name-to-address translation pairs (but may be out of date!)
 - acts as proxy, forwards query into hierarchy

DNS name resolution: iterated query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

Iterated query:

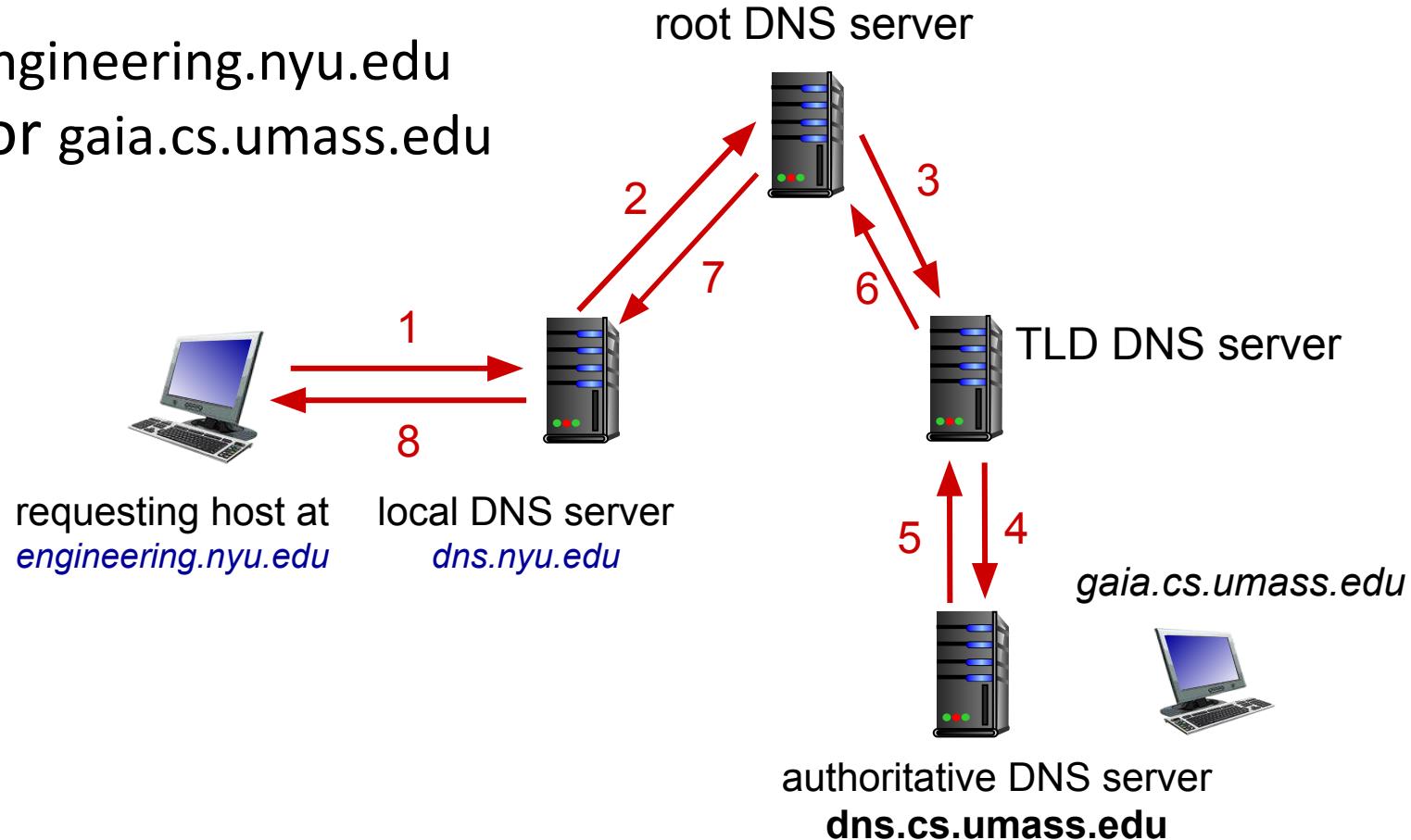
- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”



Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

Recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



Caching and Updating DNS Records

- Suppose that a host **apricot.nyu.edu** queries **dns.nyu.edu** for the IP address for the hostname **cnn.com**. After an hour later, another NYU host, say, **kiwi.nyu.edu**, also queries **dns.nyu.edu**.
- once (any) name server learns mapping, it *caches* mapping
 - cache entries timeout (disappear) after some time (TTL)
 - TLD servers typically cached in local name servers
 - thus root name servers not often visited
- cached entries may be *out-of-date* (**best-effort name-to-address translation!**)
 - if name host changes IP address, may not be known Internet-wide until all TTLs expire!
- update/notify mechanisms proposed IETF standard
 - RFC 2136

DNS: distributed database storing resource records (**RR**)

RR format: (name, value, type, ttl)

type=A

- name is hostname
- value is IP address

relay1.bar.foo.com, 145.37.93.126, A

type=NS

- name is domain (e.g., foo.com)
- value is hostname of authoritative name server for this domain

foo.com, dns.foo.com, NS

type=CNAME

- name is alias name for some “canonical” (the real) name
- www.ibm.com is really severeast.backup2.ibm.com
- value is canonical name

ibm.com, severeast.backup2.ibm.com, CNAME

type=MX

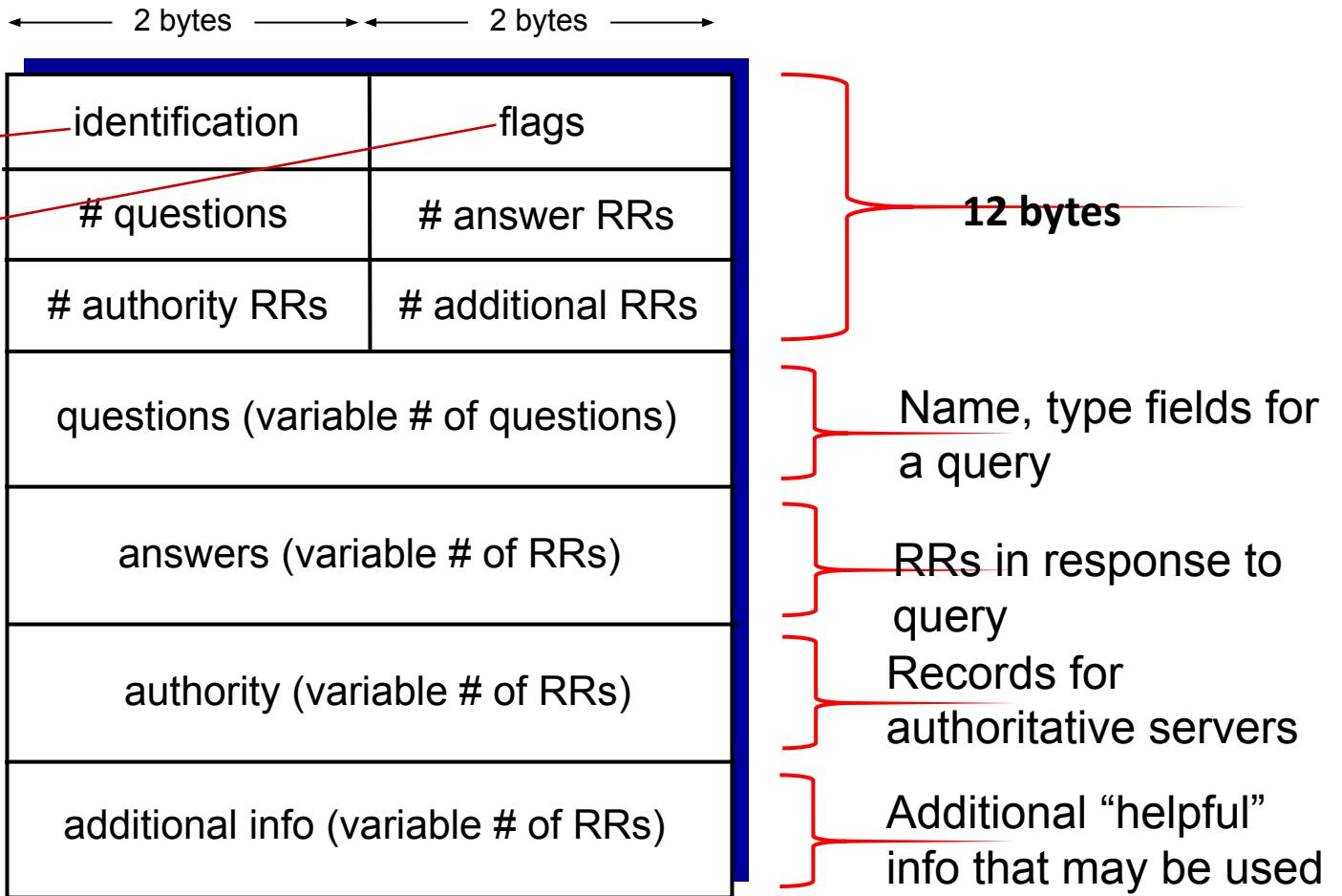
- value is canonical name of a mailserver associated with alias hostname name

example.com, mail.example.com, MX

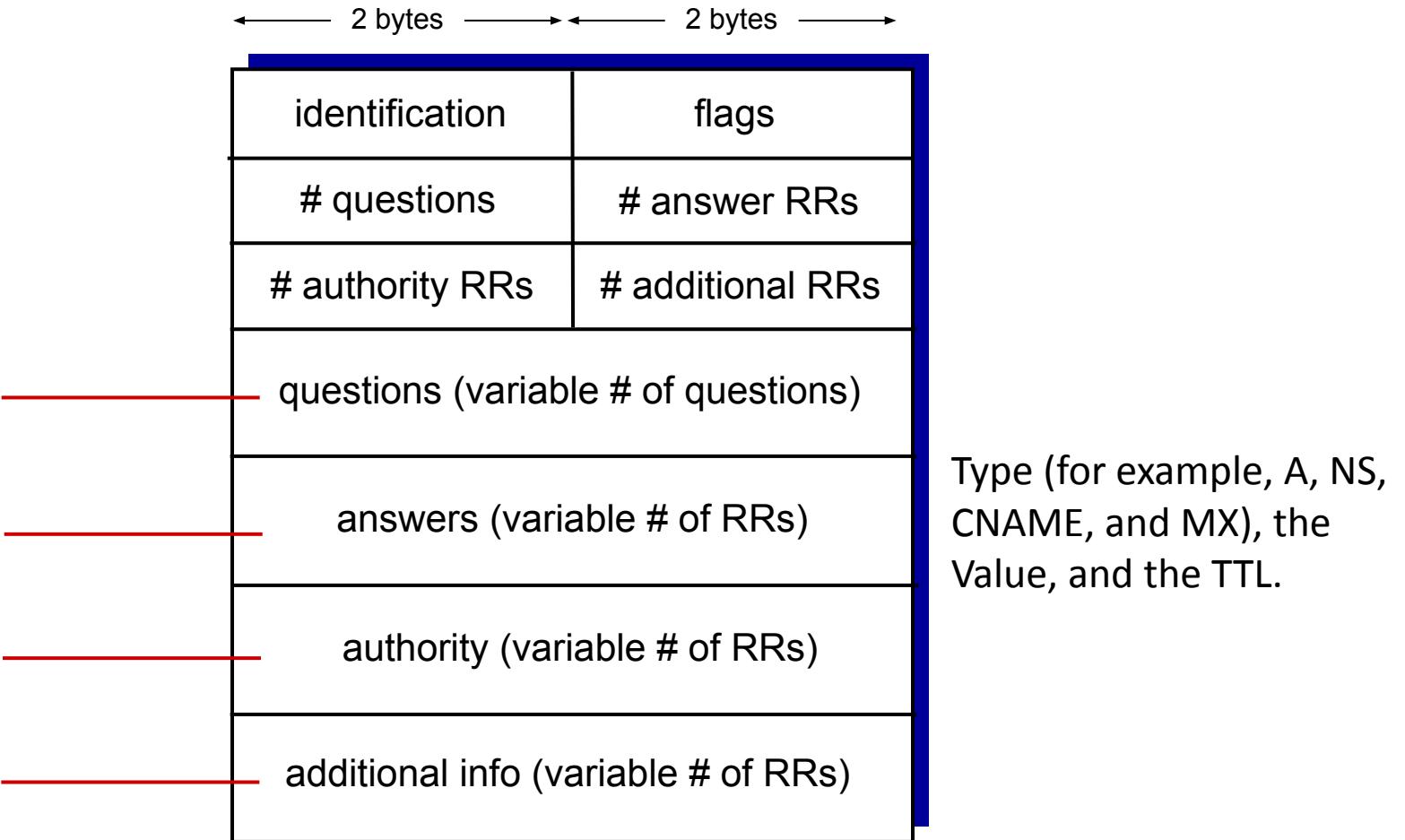
DNS *query* and *reply* messages, both have same *format*:

message header:

- **identification:** 16 bit # for query,
reply to query uses same #
- **flags:**
 - query or reply (1-bit)
 - recursion desired
 - recursion available
 - reply is authoritative



DNS *query* and *reply* messages, both have same *format*:



Emulating Local DNS Server (Step 1: Ask Root)

Directly send the query to this server.

```
seed@ubuntu:~$ dig @a.root-servers.net www.example.net
```

(Only a portion of the reply is shown here)

```
;; QUESTION SECTION:
;www.example.net.          IN      A

;; AUTHORITY SECTION:
net.           172800  IN      NS      m.gtld-servers.net.
net.           172800  IN      NS      l.gtld-servers.net.
net.           172800  IN      NS      k.gtld-servers.net.

;; ADDITIONAL SECTION:
m.gtld-servers.net.    172800  IN      A      192.55.83.30
l.gtld-servers.net.    172800  IN      A      192.41.162.30
k.gtld-servers.net.    172800  IN      A      192.52.178.30
```

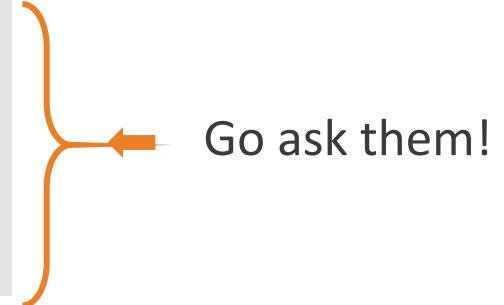
No answer (the root does not know the answer)

Go ask them!

Steps 2-3: Ask .net & example.net servers

```
seed@ubuntu:~$ dig @m.gtld-servers.net www.example.net
```

```
;; QUESTION SECTION:  
;www.example.net.          IN      A  
  
;; AUTHORITY SECTION:  
example.net.        172800  IN      NS      a.iana-servers.net.  
example.net.        172800  IN      NS      b.iana-servers.net.  
  
;; ADDITIONAL SECTION:  
a.iana-servers.net.  172800  IN      A       199.43.132.53  
b.iana-servers.net.  172800  IN      A       199.43.133.53
```



Go ask them!

```
seed@ubuntu:$ dig @a.iana-servers.net www.example.net
```

```
;; QUESTION SECTION:  
;www.example.net.          IN      A  
  
;; ANSWER SECTION:  
www.example.net.        86400   IN      A       93.184.216.34
```

- Ask an example.net nameservers.

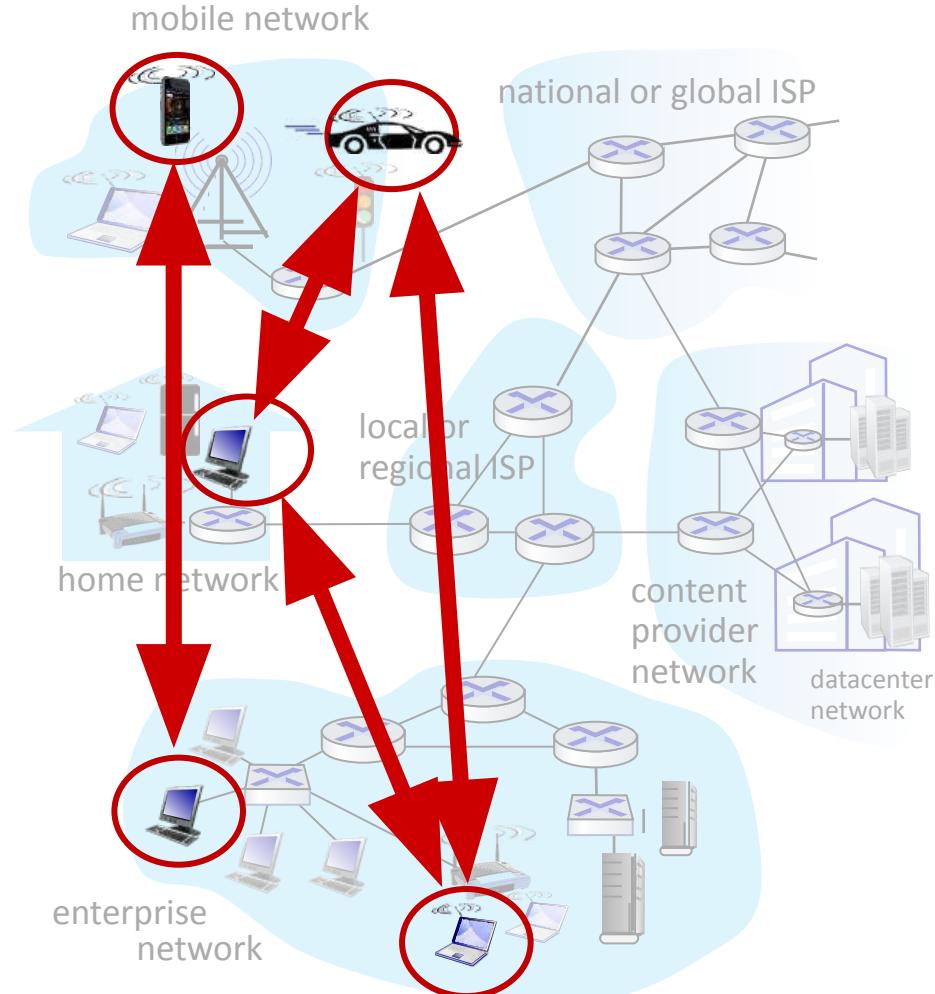
Finally got the answer

Example: new startup “Network Utopia”

- register name **networkutopia.com** at *DNS registrar* (e.g., Network Solutions)
 - provide names, IP addresses of authoritative name server (primary and secondary)
 - registrar inserts NS, A RRs into .com TLD server:
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)
- create authoritative server locally with IP address 212.212.212.1
 - type A record for www.networkutopia.com
 - type MX record for networkutopia.com

Peer-to-peer (P2P) architecture

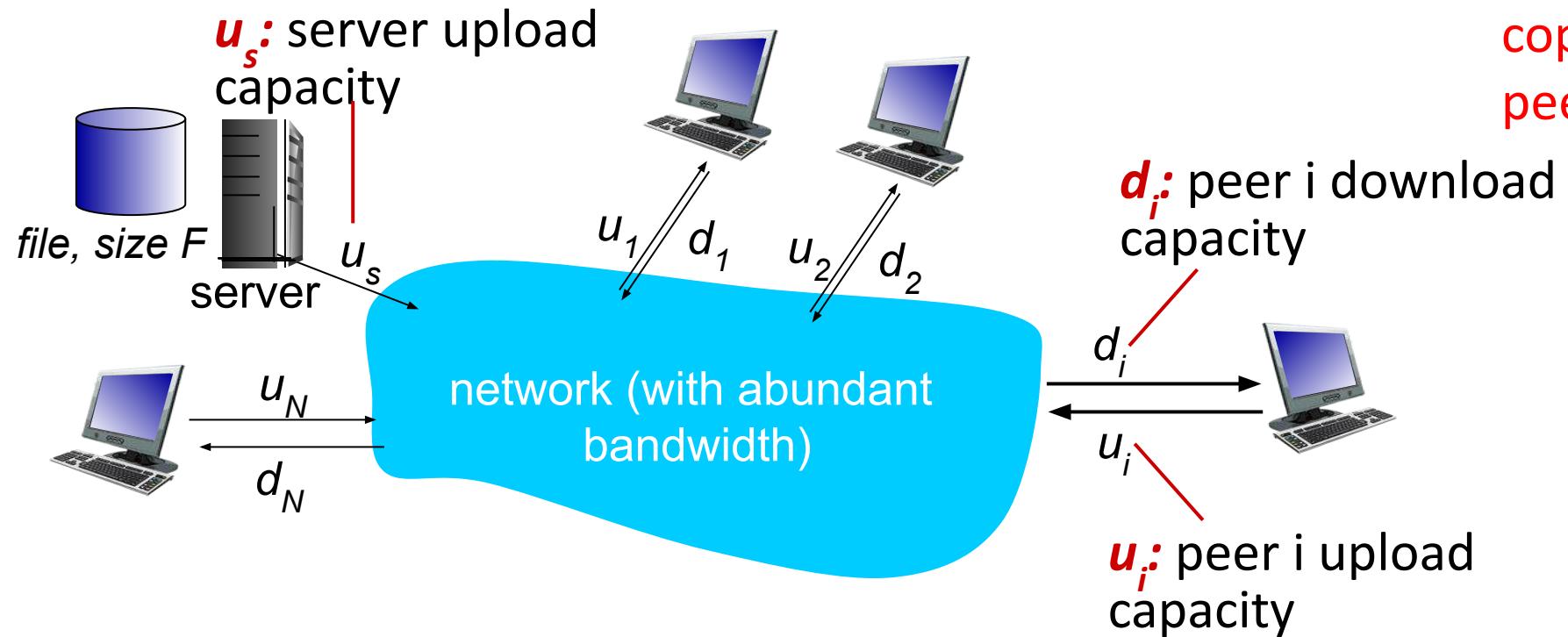
- no always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
 - *self scalability* – new peers bring new service capacity, and new service demands
- peers are intermittently connected and change IP addresses
 - complex management
- examples: P2P file sharing (BitTorrent), media streaming (Spotify), VoIP (Skype)



File distribution: client-server vs P2P

Q: how much time to distribute file (size F) from one server to N peers?

- peer upload/download capacity is limited resource



The **distribution time** is the time it takes to get a copy of the file to all N peers.

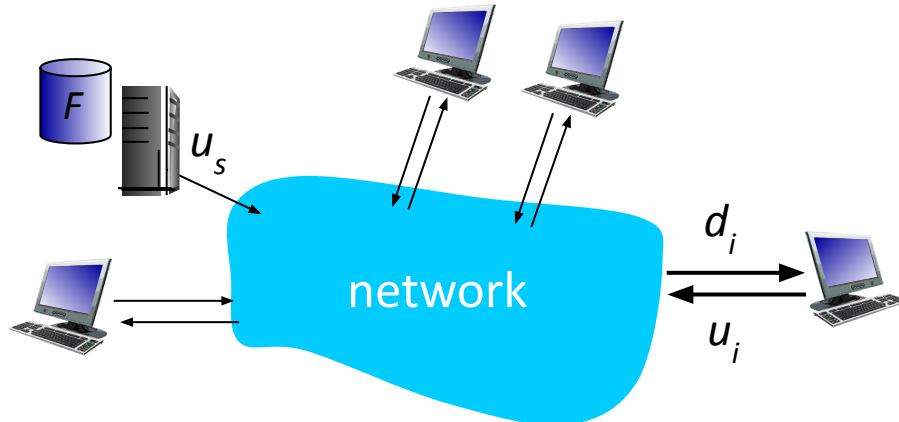
File distribution time: client-server

- **server transmission:** must sequentially send (upload) N file copies:

- time to send one copy: F/u_s
- time to send N copies: NF/u_s

- **client:** each client must download file copy

- d_{min} = min client download rate =
i.e., $d_{min} = \min \{d_1, d_2, \dots, d_N\}$
- min client download time: F/d_{min}



*time to distribute F
to N clients using
client-server approach* $D_{c-s} > \max\{NF/u_s, F/d_{min}\}$

increases linearly in N

File distribution time: P2P

- **server transmission:** must upload at least one copy:

- time to send one copy: F/u_s

- **client:** each client must download file copy

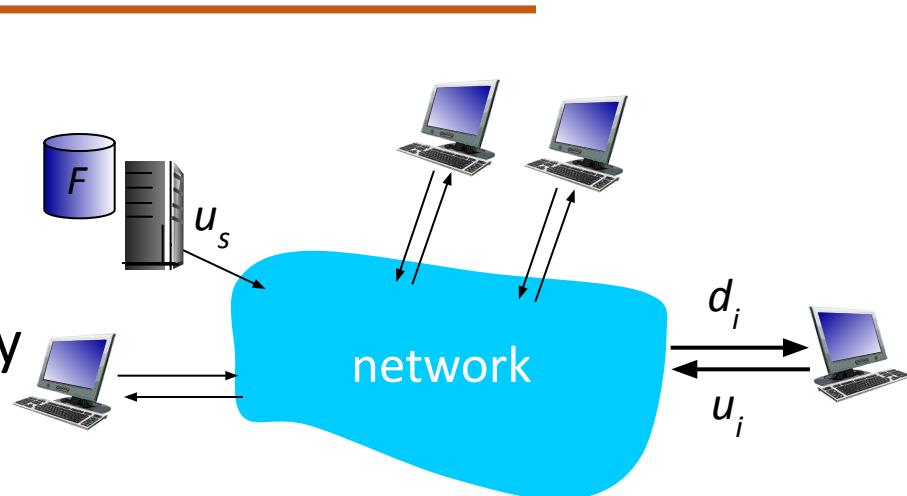
- min client download time: F/d_{min}

- **clients:** as aggregate must download NF bits

- max upload rate (limiting max download rate) is $u_s + \sum u_i$

time to distribute F
to N clients using
P2P approach

$$D_{P2P} > \max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

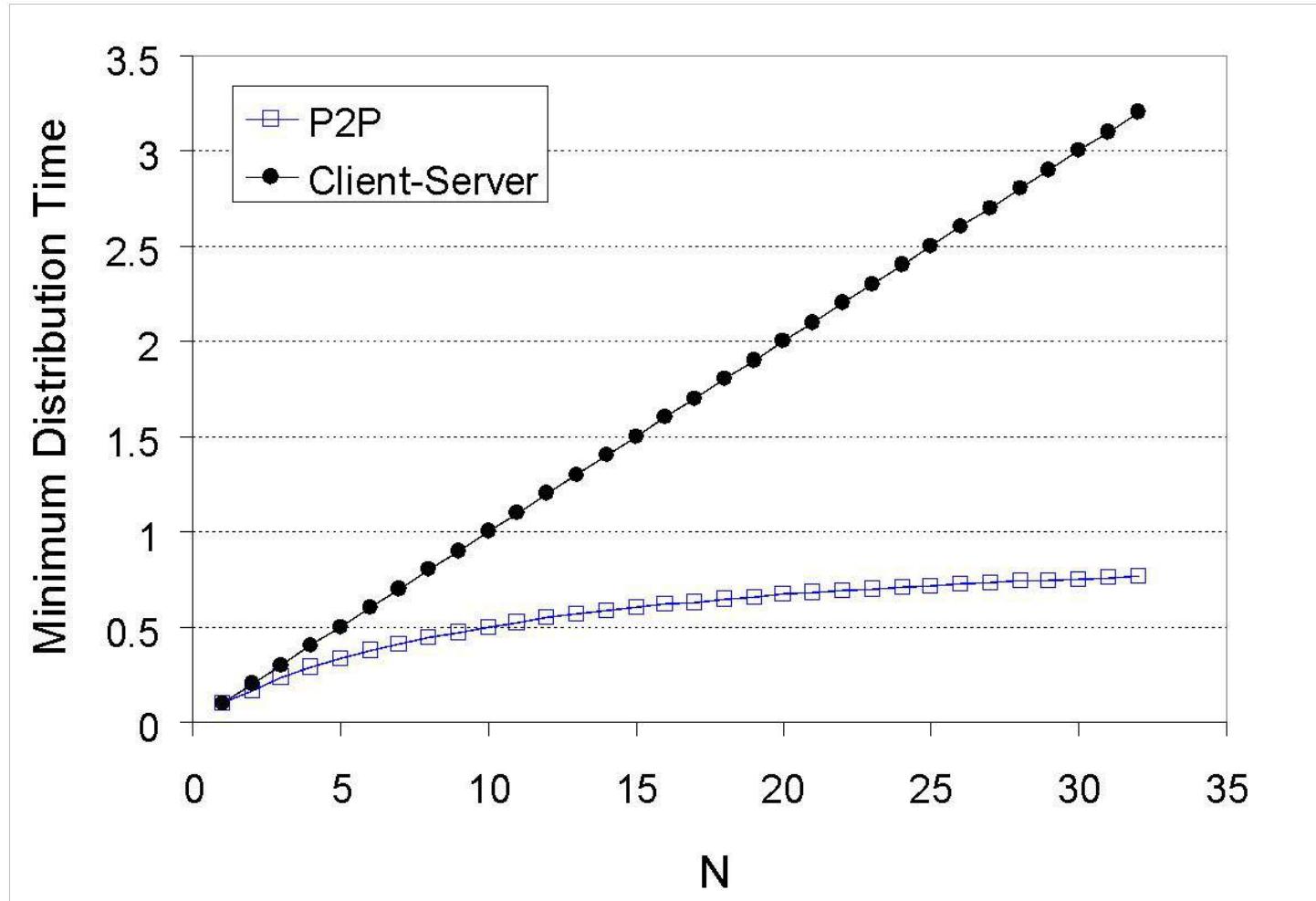


Total upload capacity of
the system as a whole

increases linearly in N
... but so does this, as each peer brings service
capacity

Eqtn - provides a lower bound for the minimum distribution time for the P2P architecture.

Client (all peers) upload rate = u , $F/u = 1$ hour, $u_s = 10u$, $d_{min} \geq u_s$

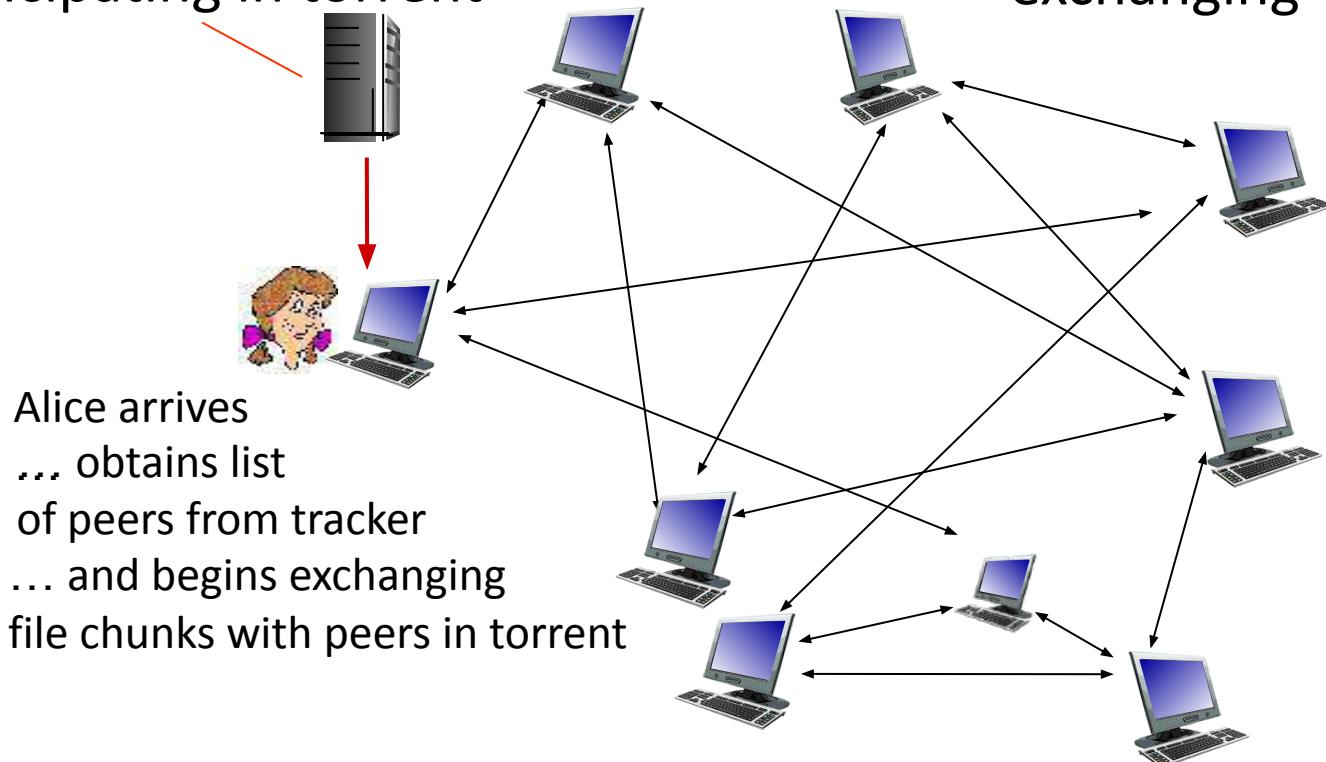


- A peer can transmit the entire file in one hour.
- The server transmission rate is 10 times the peer upload rate.
- Peer download rates are set large enough so as not to have an effect.

P2P file distribution: BitTorrent

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

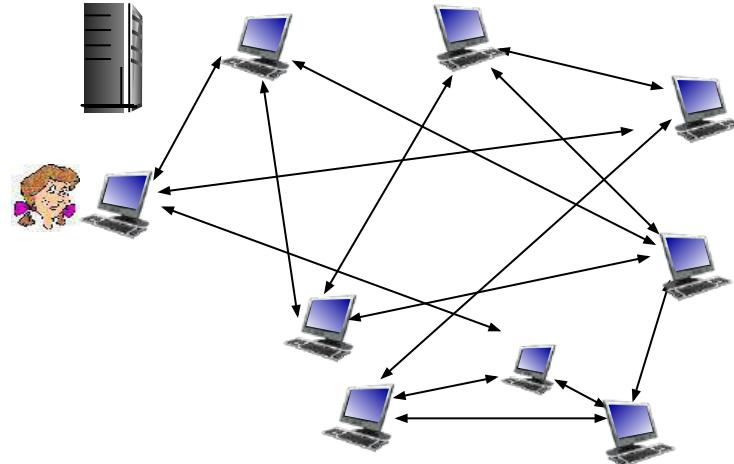
tracker: tracks peers participating in torrent



Alice arrives
... obtains list
of peers from tracker
... and begins exchanging
file chunks with peers in torrent

torrent: group of peers exchanging chunks of a file

- peer joining torrent:
 - has no chunks, but will accumulate them over time from other peers
 - registers with tracker to get list of peers, connects to subset of peers (“neighbors”)
- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn*: peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent



Requesting chunks:

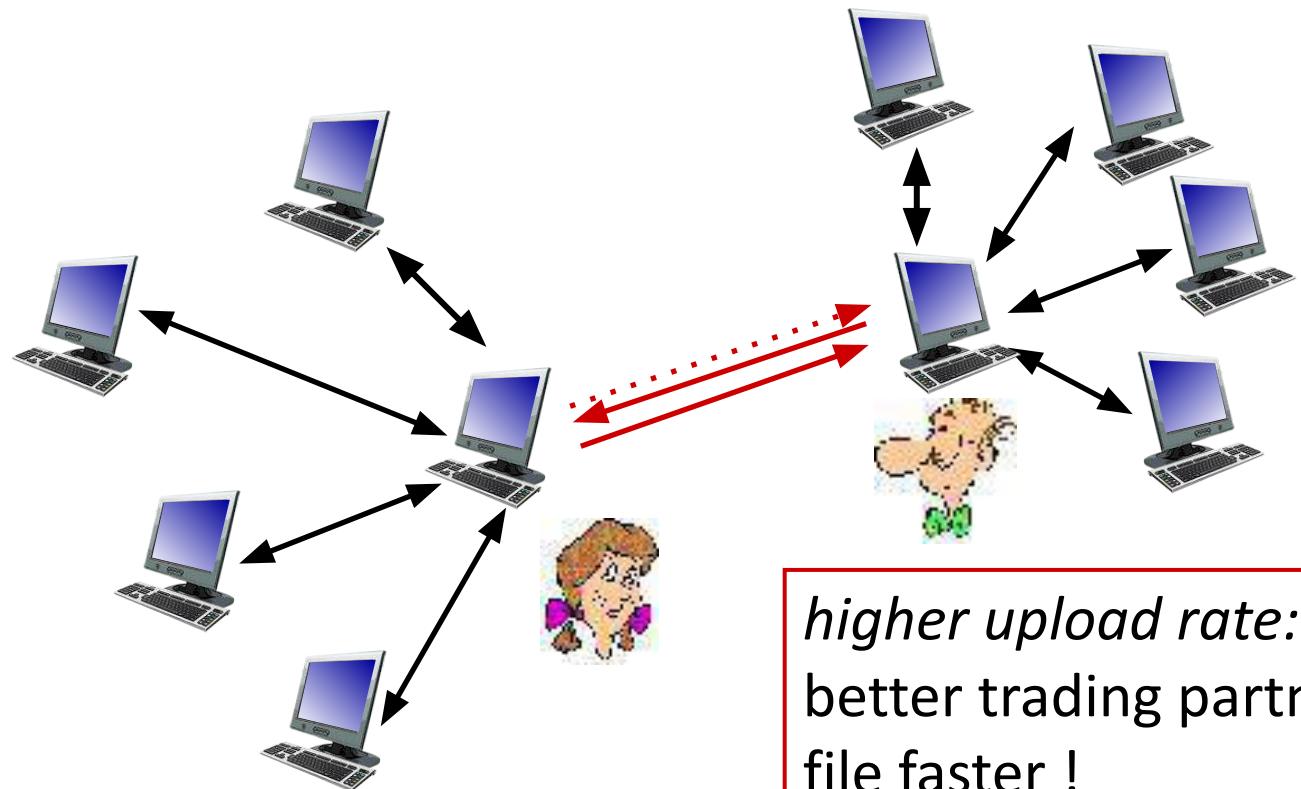
- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, **rarest first**

Sending chunks: tit-for-tat

- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
 - other peers are choked by Alice (do not receive chunks from her)
 - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
 - “**optimistically unchoke**” this peer
 - newly chosen peer may join top 4

BitTorrent: tit-for-tat

- (1) Alice “optimistically unchoke” Bob
- (2) Alice becomes one of Bob’s top-four providers; Bob reciprocates
- (3) Bob becomes one of Alice’s top-four providers

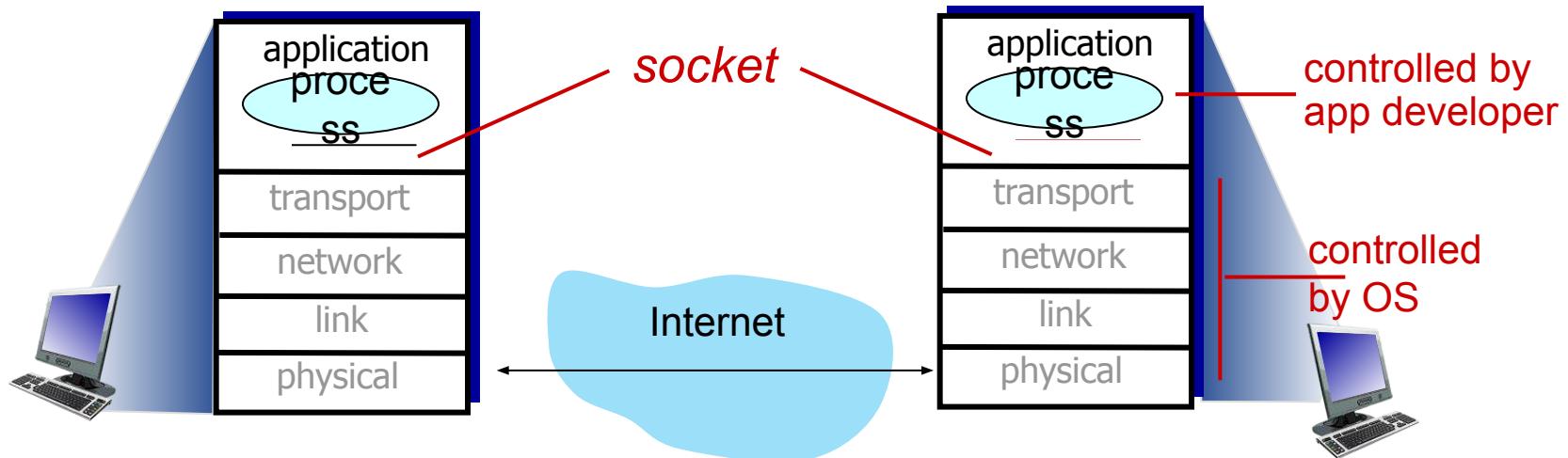


**Pieces (mini-chunks),
pipelining, random first
selection, endgame mode,
and anti-snubbing**

*higher upload rate: find
better trading partners, get
file faster !*

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Two socket types for two transport services:

- *UDP*: unreliable datagram
- *TCP*: reliable, byte stream-oriented

Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

UDP: no “connection” between client & server

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes (“datagrams”) between client and server



server (running on serverIP)

```
create socket, port= x:  
serverSocket =  
socket(AF_INET,SOCK_DGRAM)
```

```
read datagram from  
serverSocket
```

```
write reply to  
serverSocket  
specifying  
client address,  
port number
```

client 

```
create  
clientSocket =  
socket(AF_INET,SOCK_DGRAM)
```

```
Create datagram with server IP and  
port=x; send datagram via  
clientSocket
```

```
read datagram from  
clientSocket  
close  
clientSocket
```

Python UDPCClient

```
include Python's socket library → from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                     (serverName, serverPort))
modifiedMessage, serverAddress =
clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

create UDP socket for server →

get user keyboard input →

attach server name, port to message; send into →

socket

read reply characters from socket into string →

print out received string and close socket →

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(("", serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

create UDP socket →
bind socket to local port number 12000 →
loop forever →
Read from UDP socket into message, getting →
client's address (client IP and port) →
send upper case string back to this client →

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

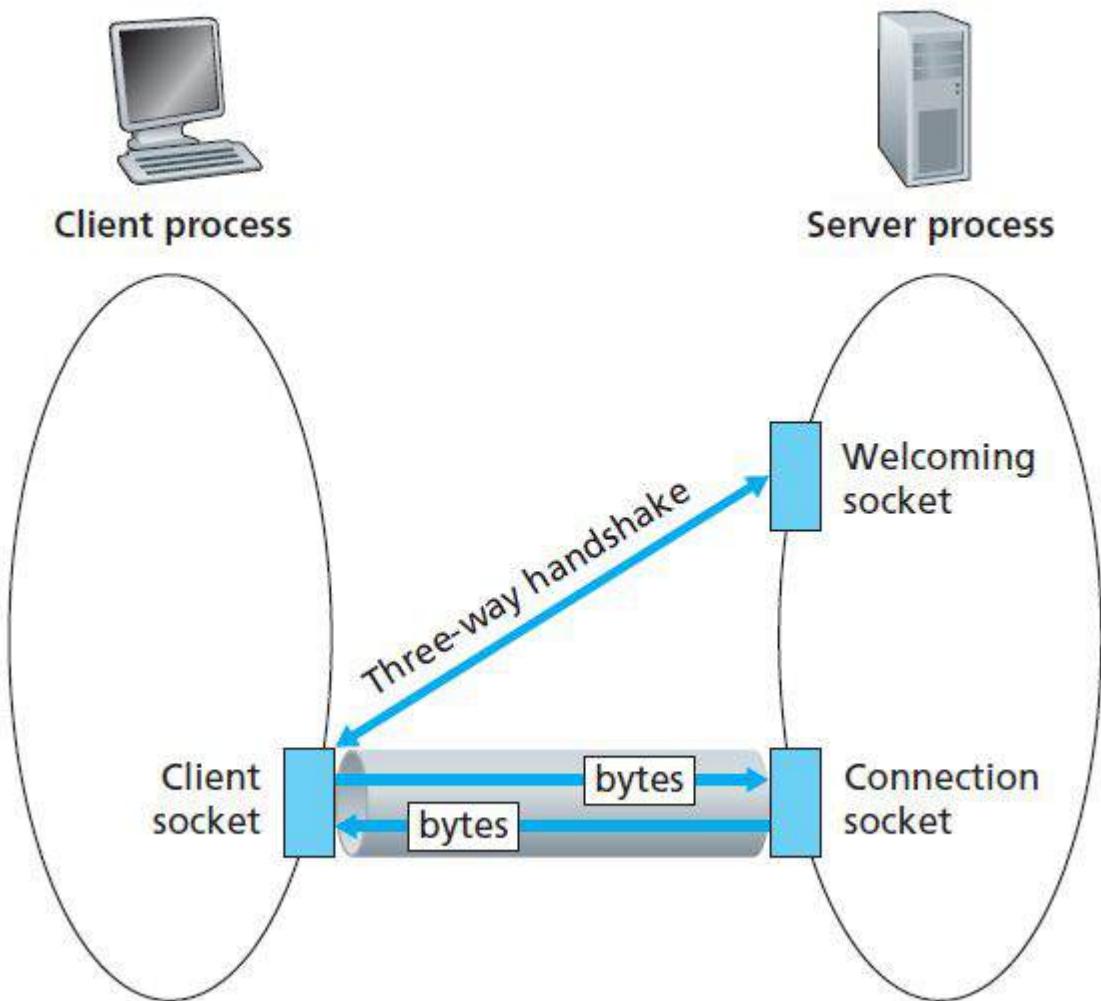
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket*: client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
 - allows server to talk with multiple clients
 - source port numbers used to distinguish clients (more in Chap 3)

Application viewpoint

TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server

The TCP Server Process has Two Sockets





server (running on hostid)



client

create socket,
port=x, for incoming
request:
serverSocket = socket()

wait for incoming
connection request
connectionSocket =
serverSocket.accept()

read request
from
connectionSock

et
write reply to
connectionSock

et

close
connectionSock
et

TCP

create socket,
connect to **hostid**, port=x
clientSocket = socket()

send request using
clientSocket

read reply from
clientSocket
close
**clientSoc
ket**

Python TCPClient

create TCP socket for
server, remote port
12000



```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

No need to attach
server name, port



SOCK_STREAM

Python TCP Server

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(("",serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                           encode())
connectionSocket.close()
```

create TCP welcoming socket → from socket import *

server begins listening for incoming TCP requests → serverPort = 12000

loop forever → serverSocket = socket(AF_INET,SOCK_STREAM)

server waits on accept() for incoming requests, new socket created on return → serverSocket.bind(("",serverPort))

read bytes from socket (but not address as in UDP) → serverSocket.listen(1)

close connection to this client (but *not* welcoming socket) → print 'The server is ready to receive'

→ while True:

→ connectionSocket, addr = serverSocket.accept()

→ sentence = connectionSocket.recv(1024).decode()

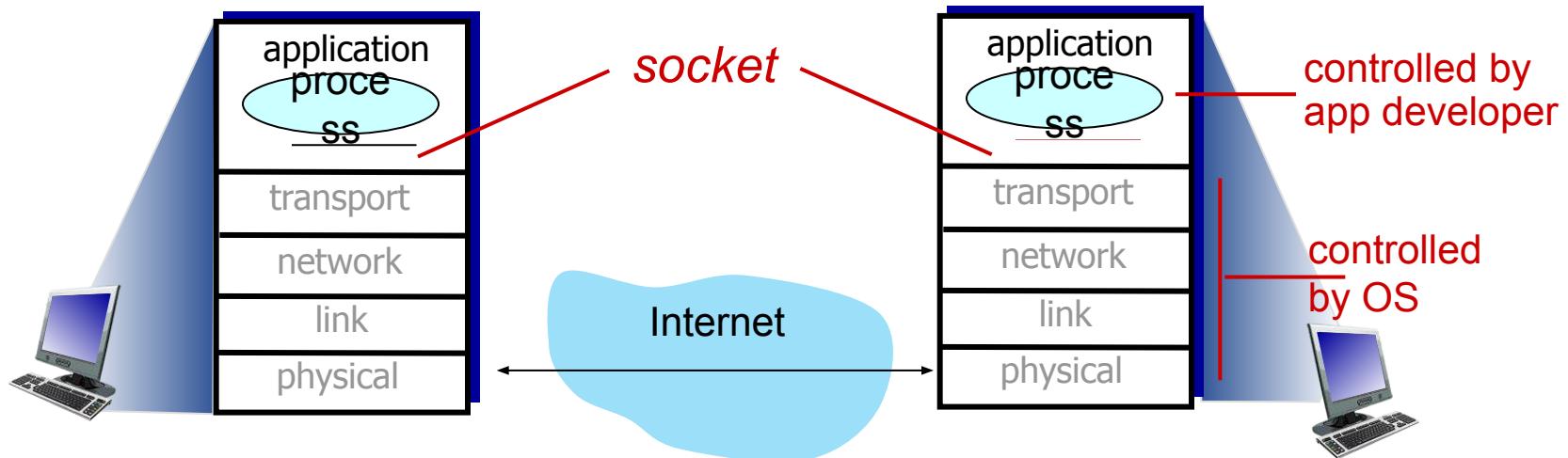
→ capitalizedSentence = sentence.upper()

→ connectionSocket.send(capitalizedSentence.

→ encode())

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Two socket types for two transport services:

- *UDP*: unreliable datagram
- *TCP*: reliable, byte stream-oriented

Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

UDP: no “connection” between client & server

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes (“datagrams”) between client and server



server (running on serverIP)

```
create socket, port= x:  
serverSocket =  
socket(AF_INET,SOCK_DGRAM)
```

```
read datagram from  
serverSocket
```

```
write reply to  
serverSocket  
specifying  
client address,  
port number
```

client 

```
create  
clientSocket =  
socket(AF_INET,SOCK_DGRAM)
```

```
Create datagram with server IP and  
port=x; send datagram via  
clientSocket
```

```
read datagram from  
clientSocket  
close  
clientSocket
```

Python UDPCClient

```
include Python's socket library → from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                     (serverName, serverPort))
modifiedMessage, serverAddress =
clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

create UDP socket for server →

get user keyboard input →

attach server name, port to message; send into →

socket

read reply characters from socket into string →

print out received string and close socket →

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(("", serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

create UDP socket →
bind socket to local port number 12000 →
loop forever →
Read from UDP socket into message, getting →
client's address (client IP and port) →
send upper case string back to this client →

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

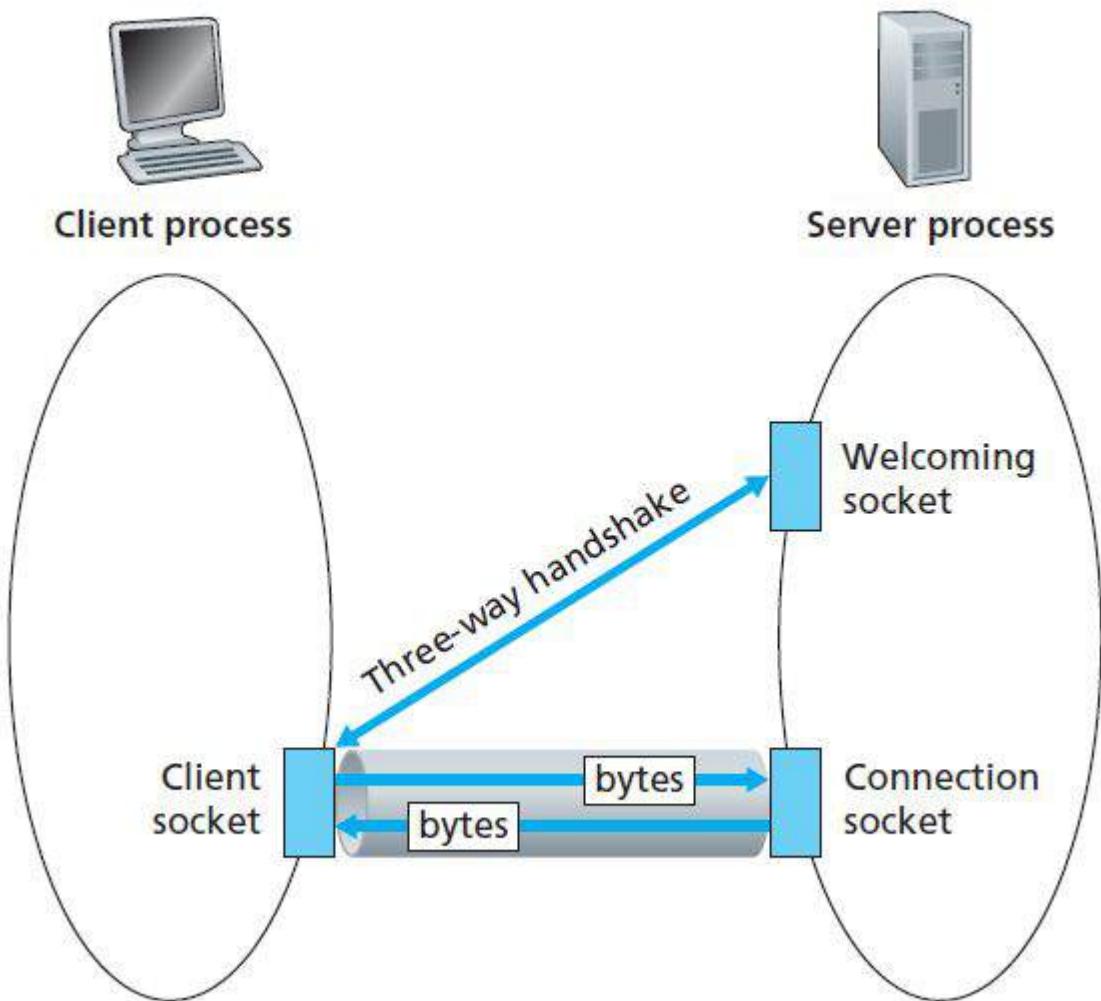
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket*: client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
 - allows server to talk with multiple clients
 - source port numbers used to distinguish clients (more in Chap 3)

Application viewpoint

TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server

The TCP Server Process has Two Sockets





server (running on hostid)



client

create socket,
port=x, for incoming
request:
serverSocket = socket()

wait for incoming
connection request
connectionSocket =
serverSocket.accept()

read request
from
connectionSock

et
write reply to
connectionSock

et
close
connectionSock

et

TCP

connection setup

create socket,
connect to **hostid**, port=x
clientSocket = socket()

send request using
clientSocket

read reply from
clientSocket
close
**clientSoc
ket**

Python TCPClient

create TCP socket for
server, remote port
12000



```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

No need to attach
server name, port



clientSocket = socket(AF_INET, **SOCK_STREAM**)

Python TCP Server

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

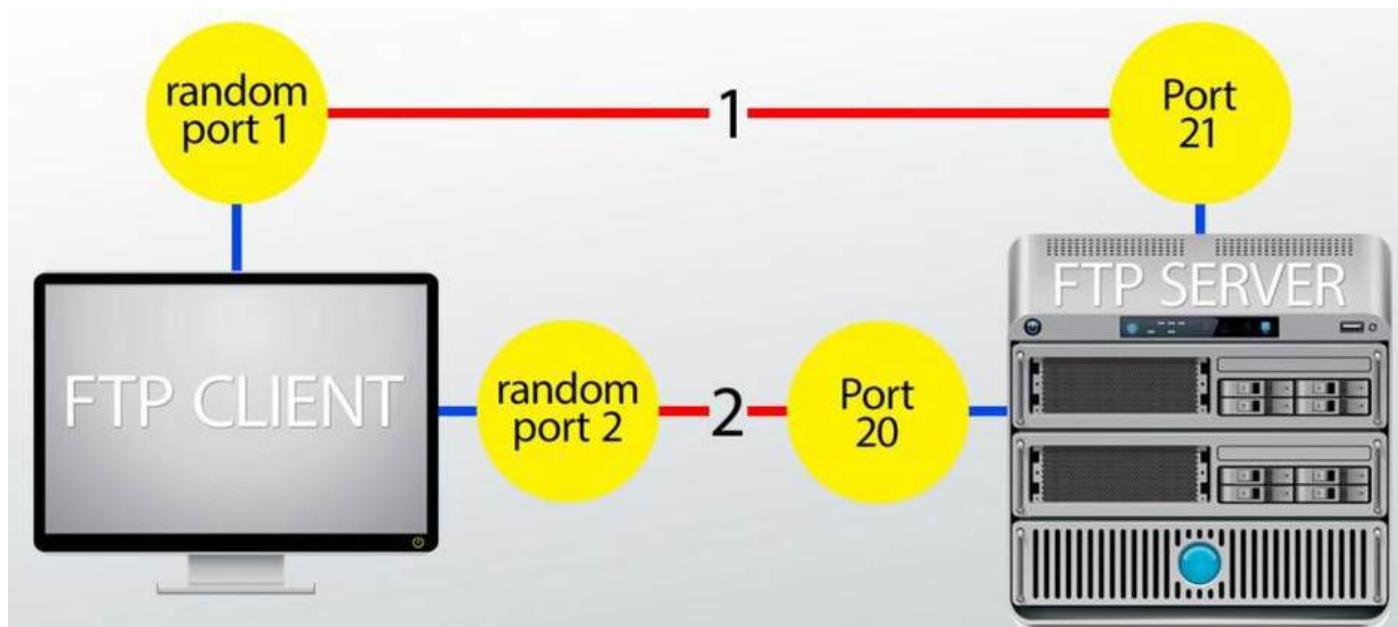
server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

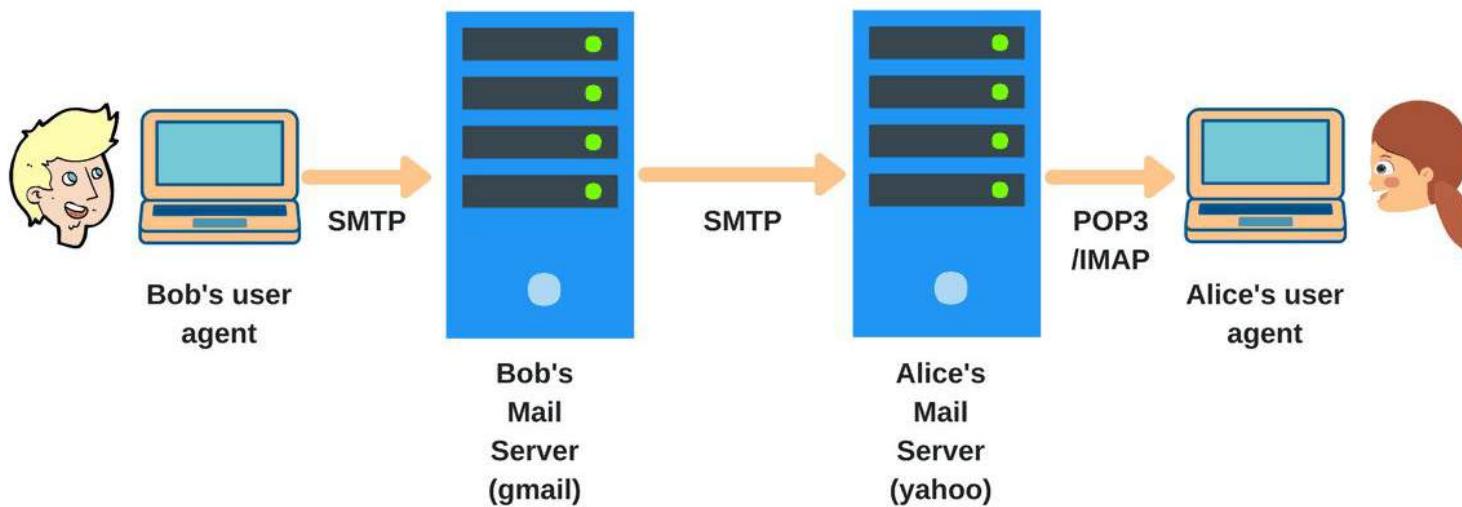
close connection to this client
(but *not* welcoming socket)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(("",serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
connectionSocket.close()
```

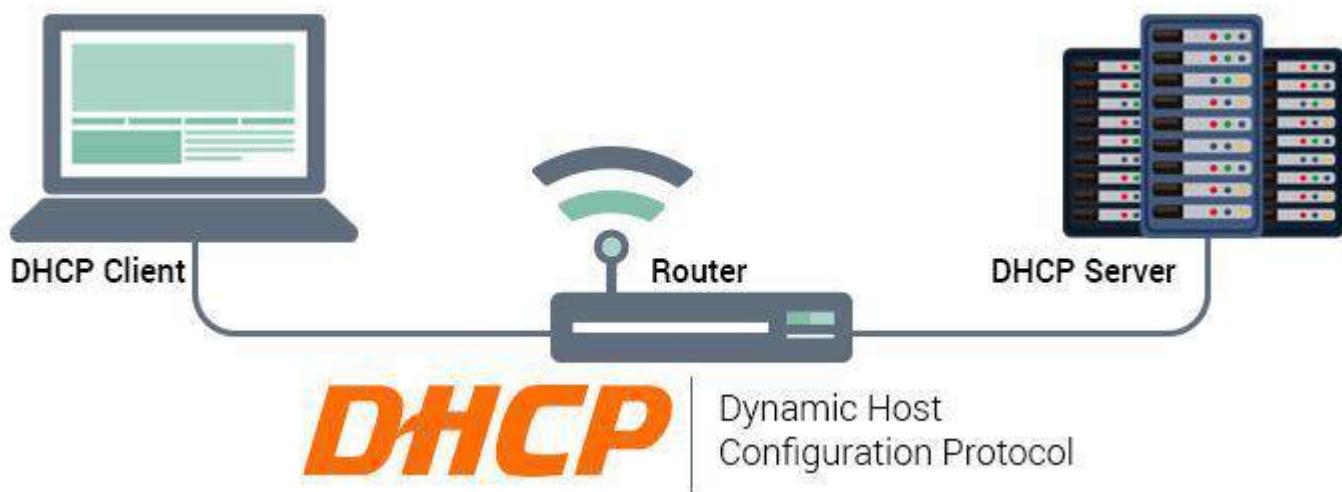
- File Transfer Protocol (FTP) - used to exchange large files on the internet TCP
- Invoked from the command prompt or some GUI.
- Allows to update (delete, rename, move, and copy) files at a server.
- Data connection (Port No. 20) & Control connection (Port No. 21)



- **Simple Mail Transfer Protocol** - an internet standard for e-mail Transmission.
- Connections are secured with SSL (Secure Socket Layer).
- Messages are stored and then forwarded to the destination (relay).
- SMTP uses a port number 25 of TCP.

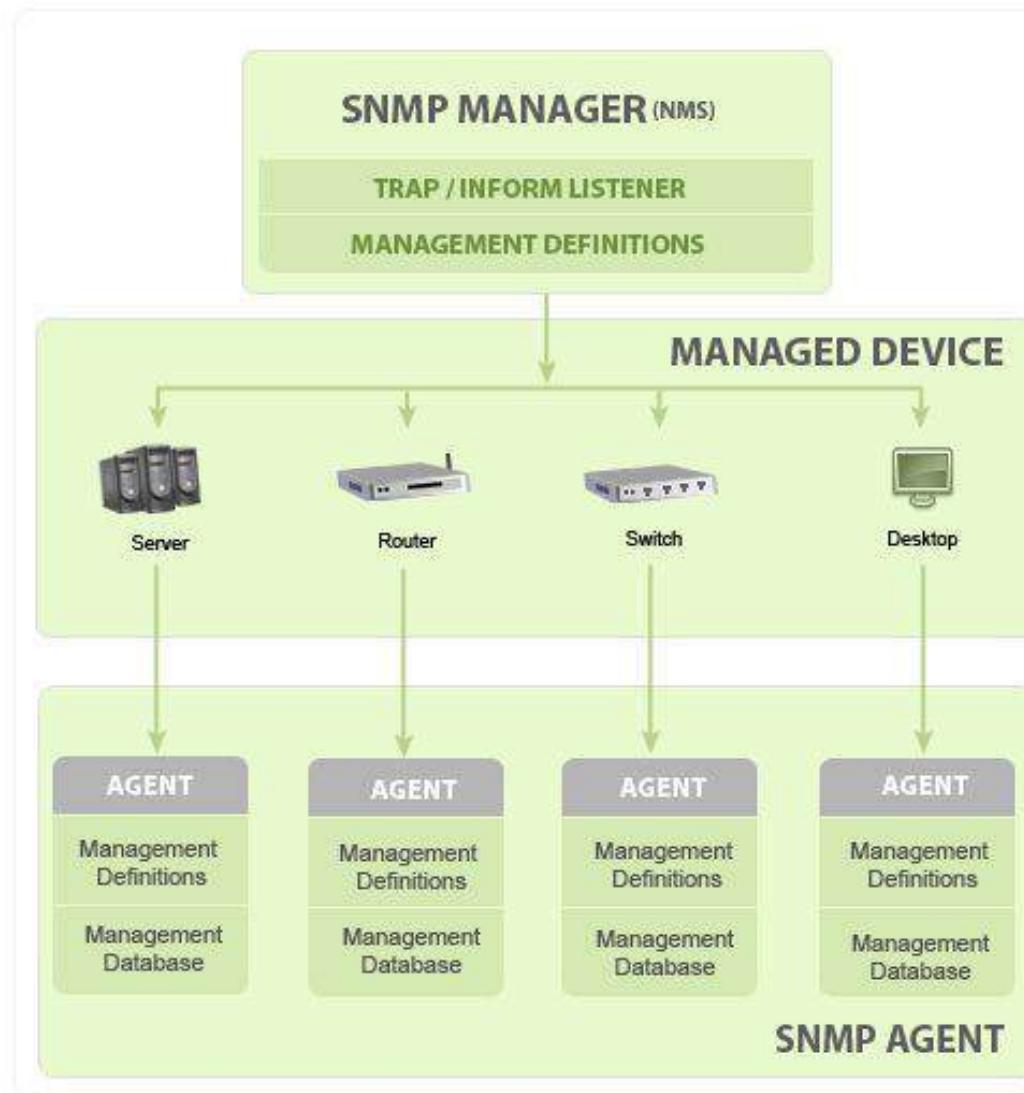


- **Dynamic Host Configuration Protocol** - assign IP addresses to computers in a network dynamically.
- IP addresses may change even when computer is in network (DHCP leases).
- DHCP port number for server is 67 and for the client is 68.
- A client-server model & based on **discovery, offer, request, and ACK**.
- Includes subnet mask, DNS server address, default gateway



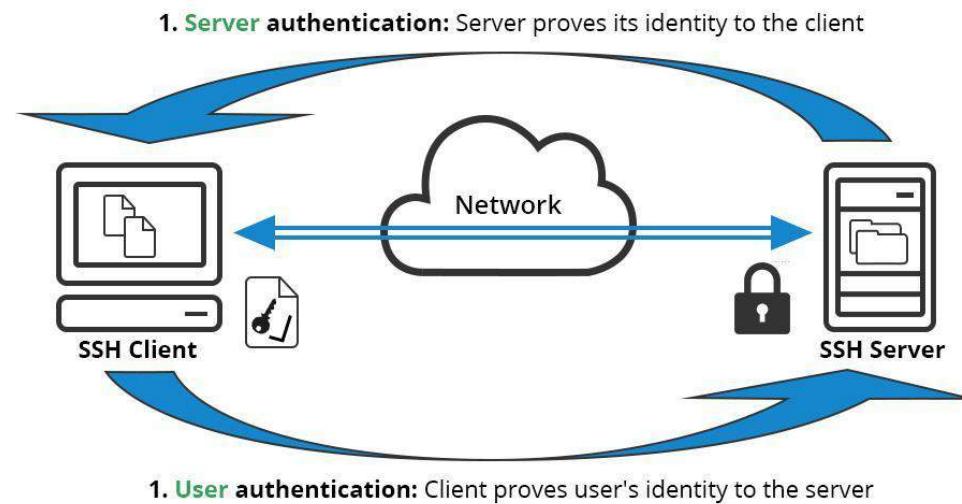
Other Application Layer Protocols - SNMP

- **Simple Network Management Protocol** - exchange management information between network devices.
- Basic components & functionalities
 - SNMP Manager
 - Managed Devices
 - SNMP Agents
 - MIB (Management Information Base)



Other Application Layer Protocols – Telnet & SSH

- Allows a user to communicate with a remote device.
- Used mostly by network admin to remotely access and manage devices.
- Telnet client and server installed – uses TCP port no. 23
- SSH – uses public key **encryption** & TCP port 22 by default.



Summary of Application Layer Protocols

Port #	Application Layer Protocol	Type	Description
20	FTP	TCP	File Transfer Protocol - data
21	FTP	TCP	File Transfer Protocol - control
22	SSH	TCP/UDP	Secure Shell for secure login
23	Telnet	TCP	Unencrypted login
25	SMTP	TCP	Simple Mail Transfer Protocol
53	DNS	TCP/UDP	Domain Name Server
67/68	DHCP	UDP	Dynamic Host
80	HTTP	TCP	HyperText Transfer Protocol
123	NTP	UDP	Network Time Protocol
161,162	SNMP	TCP/UDP	Simple Network Management Protocol
389	LDAP	TCP/UDP	Lightweight Directory Authentication Protocol
443	HTTPS	TCP/UDP	HTTP with Secure Socket Layer

our study of network application layer is now complete!

- application architectures
 - client-server
 - P2P
- application service requirements:
 - reliability, bandwidth, delay
- Internet transport service model
 - connection-oriented, reliable: TCP
 - unreliable, datagrams: UDP
- specific protocols:
 - HTTP
 - DNS
 - P2P: BitTorrent
- socket programming:
TCP, UDP sockets

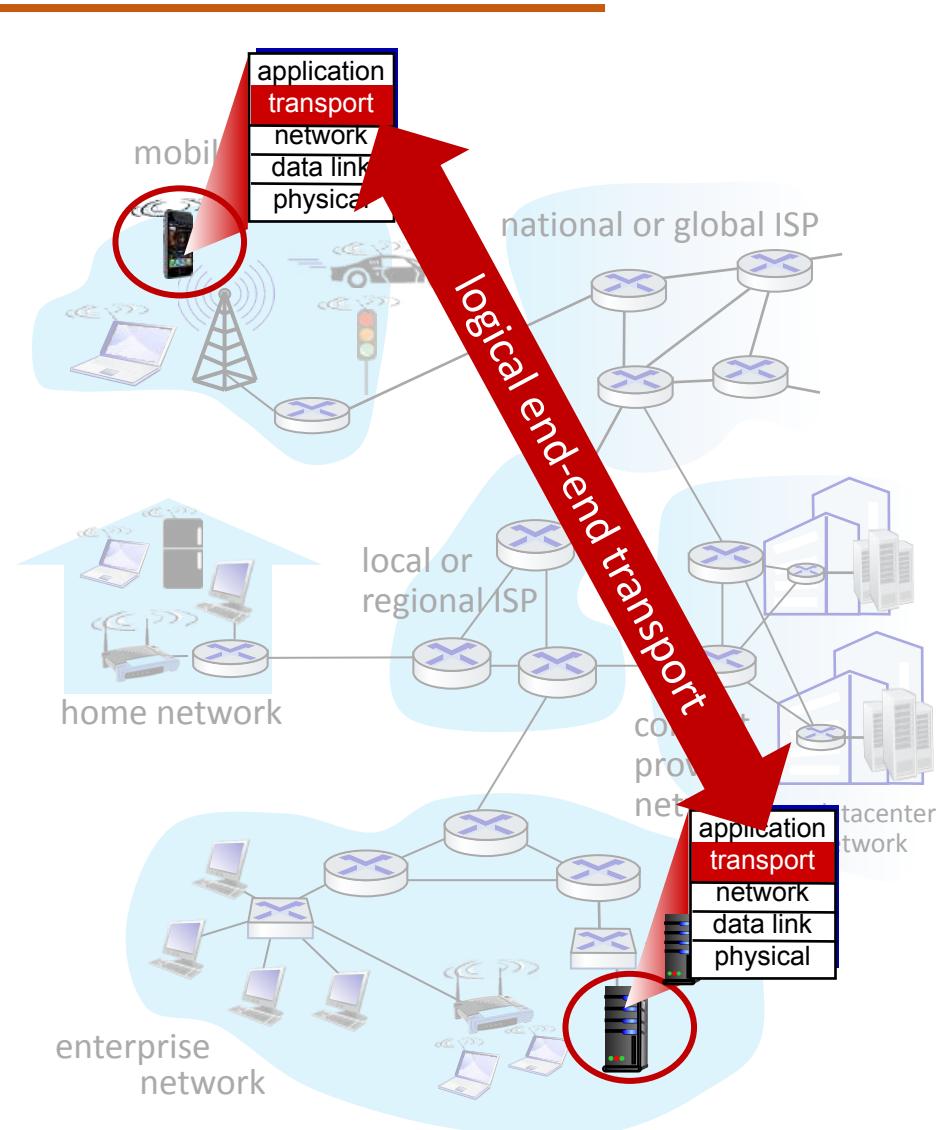
Most importantly: learned about *protocols*!

- typical request/reply message exchange:
 - client requests info or service
 - server responds with data, status code
- message formats:
 - *headers*: fields giving info about data
 - *data*: info(payload) being communicated

important themes:

- centralized vs. decentralized
- stateless vs. stateful
- scalability
- reliable vs. unreliable message transfer
- “complexity at network edge”

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
 - sender: breaks application messages into *segments*, passes to network layer
 - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
 - TCP, UDP



Transport vs. Network Layer Services and Protocols

- network layer: logical communication between *hosts*
- transport layer: logical communication between *processes*
 - relies on, enhances, network layer services

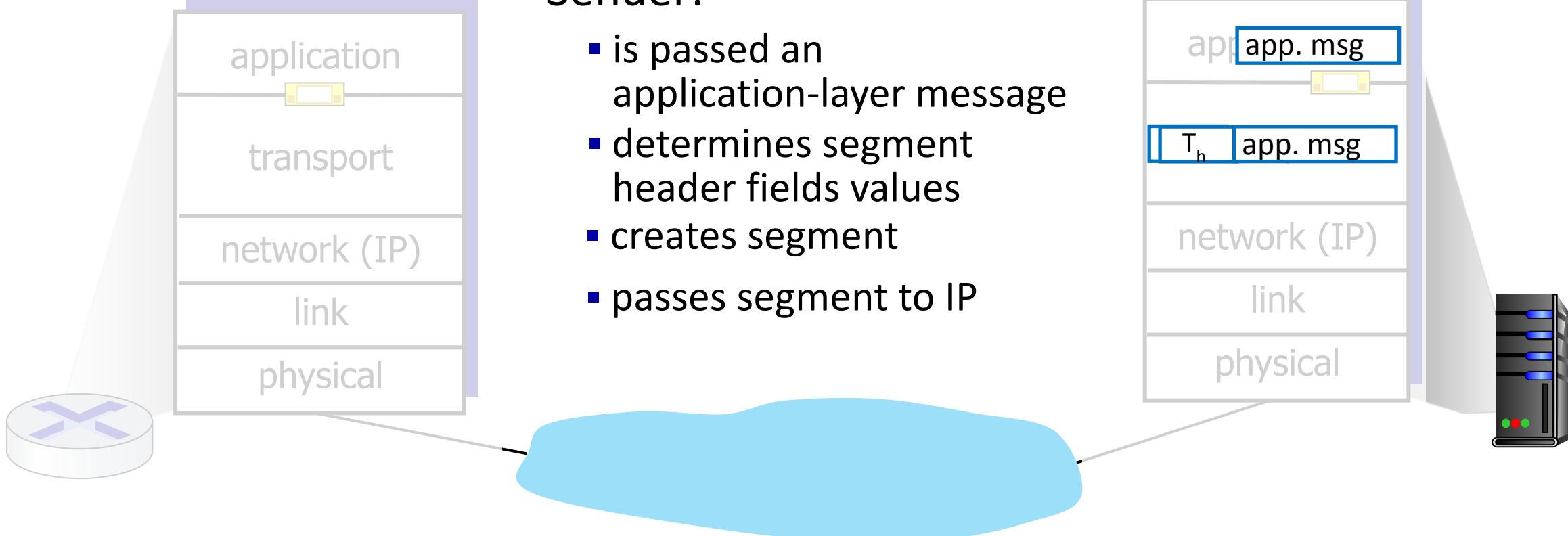
household analogy:

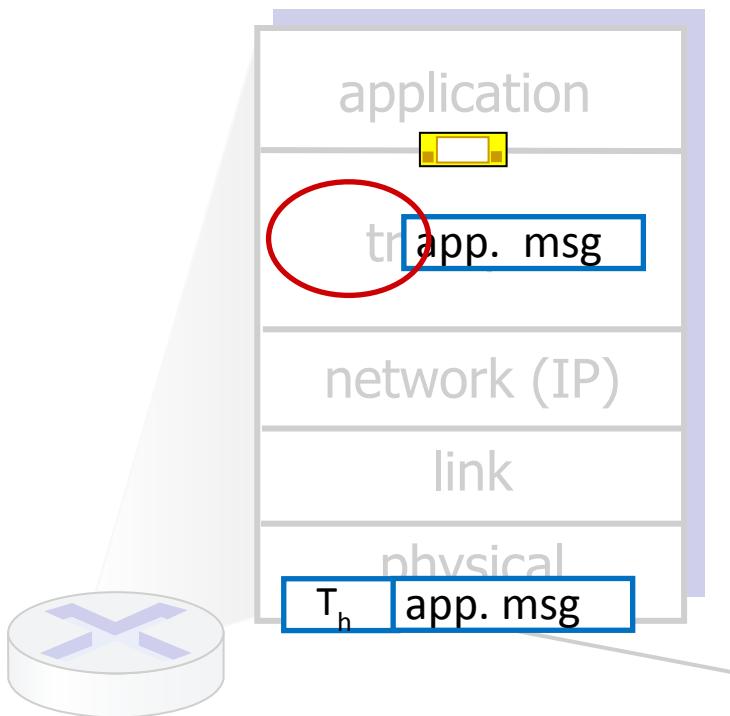
*12 kids in Ann's house
sending letters to 12 kids
in Bill's house:*

- **hosts** = houses
- **processes** = kids
- **app messages** = letters in envelopes

Sender:

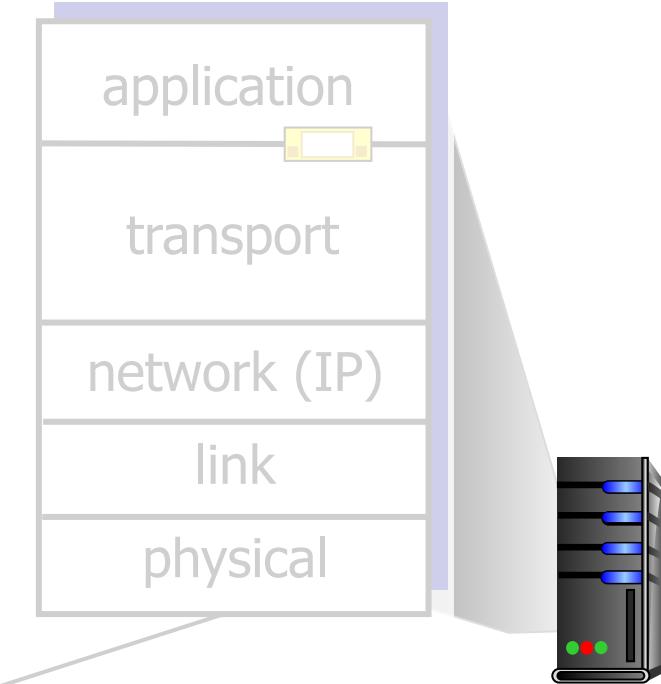
- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP





Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket



Principal Internet Transport Layer protocols

- **TCP:** Transmission Control Protocol

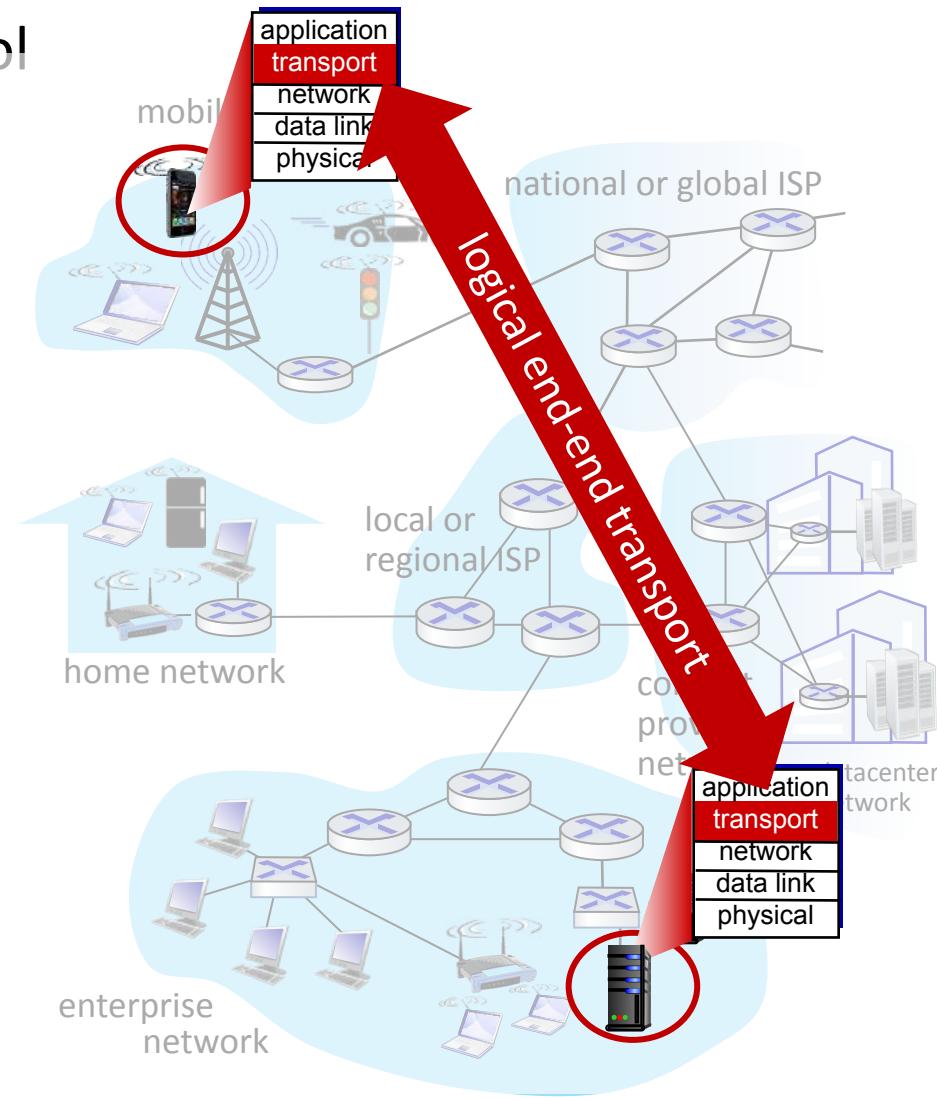
- reliable, connection oriented
- in-order delivery
- congestion control
- flow control
- connection setup

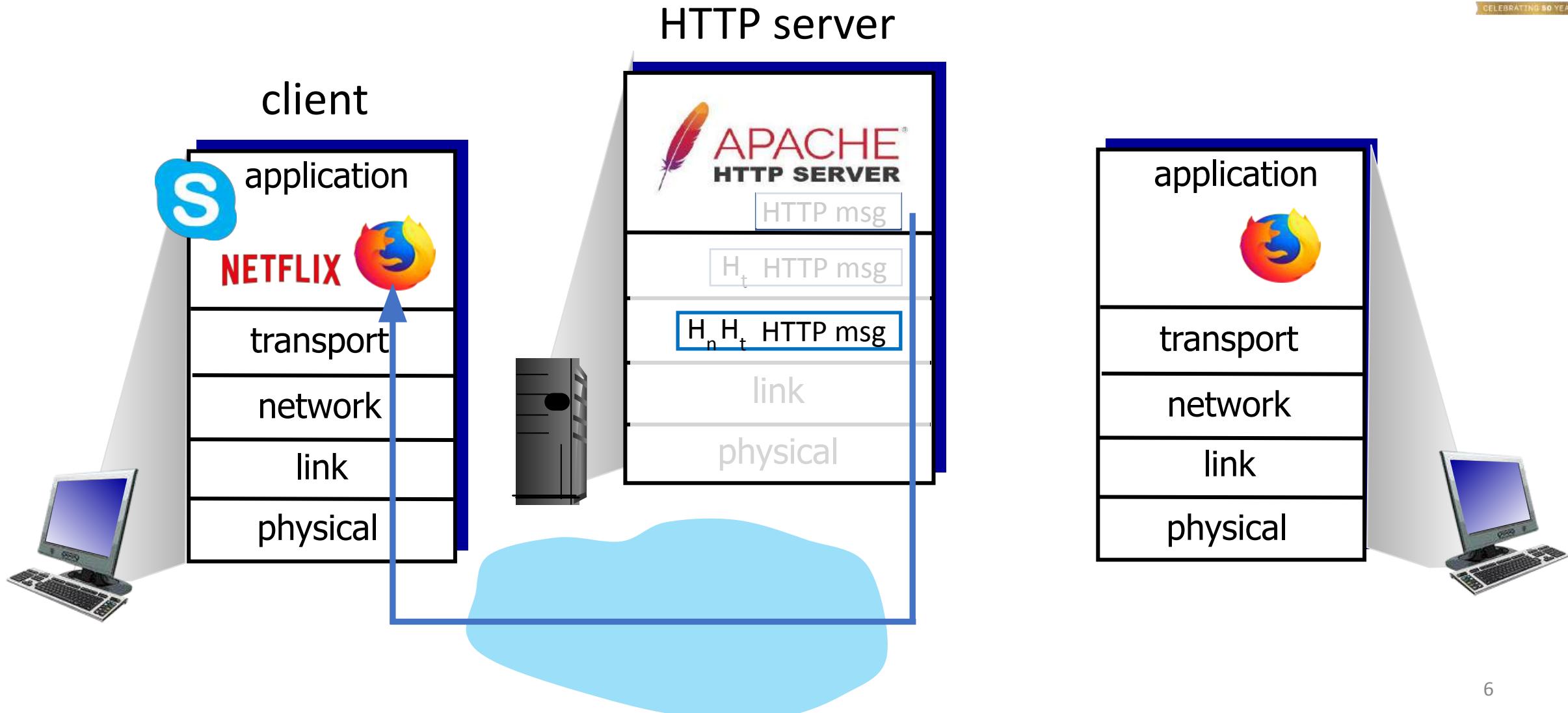
- **UDP:** User Datagram Protocol

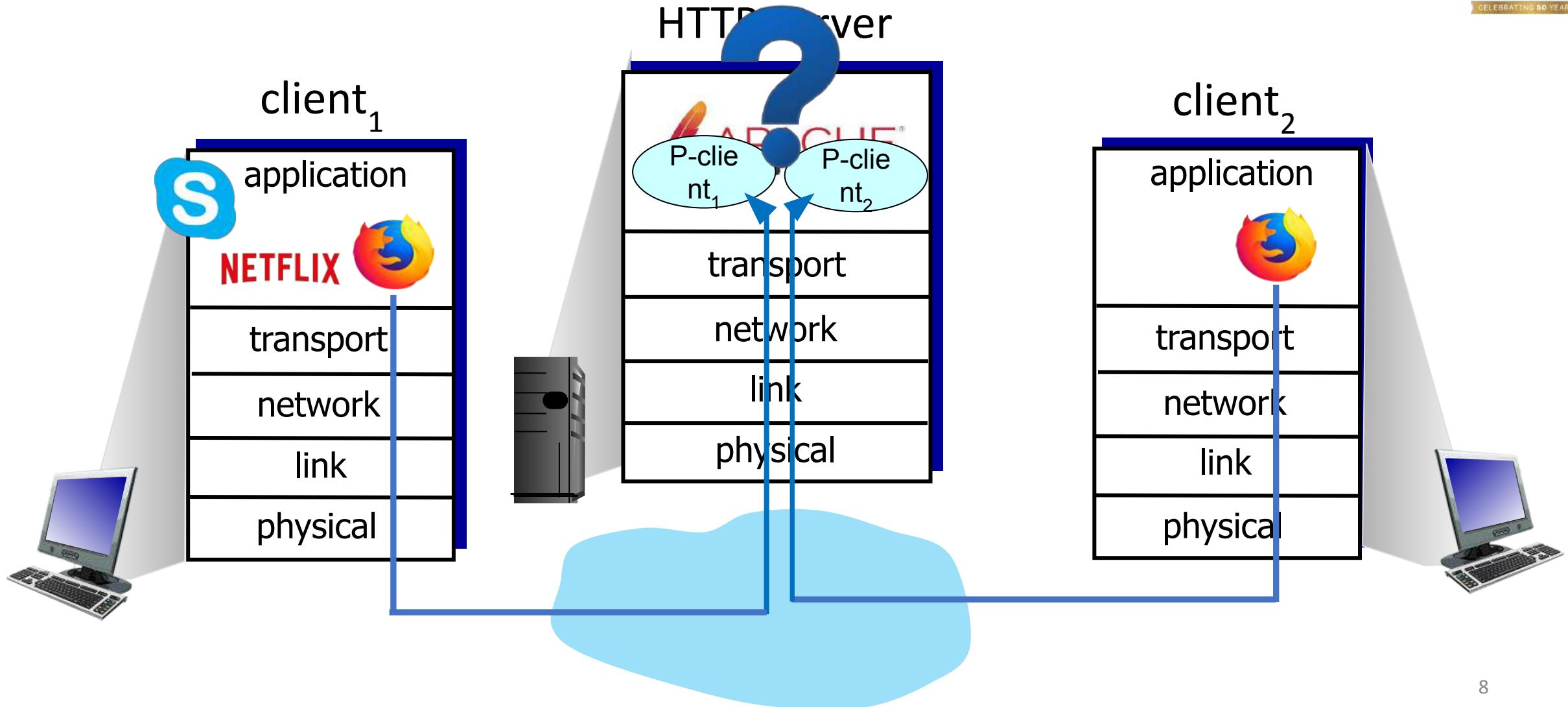
- unreliable, connectionless
- unordered delivery
- no-frills extension of “best-effort” IP

- services not available:

- delay guarantees
- bandwidth guarantees





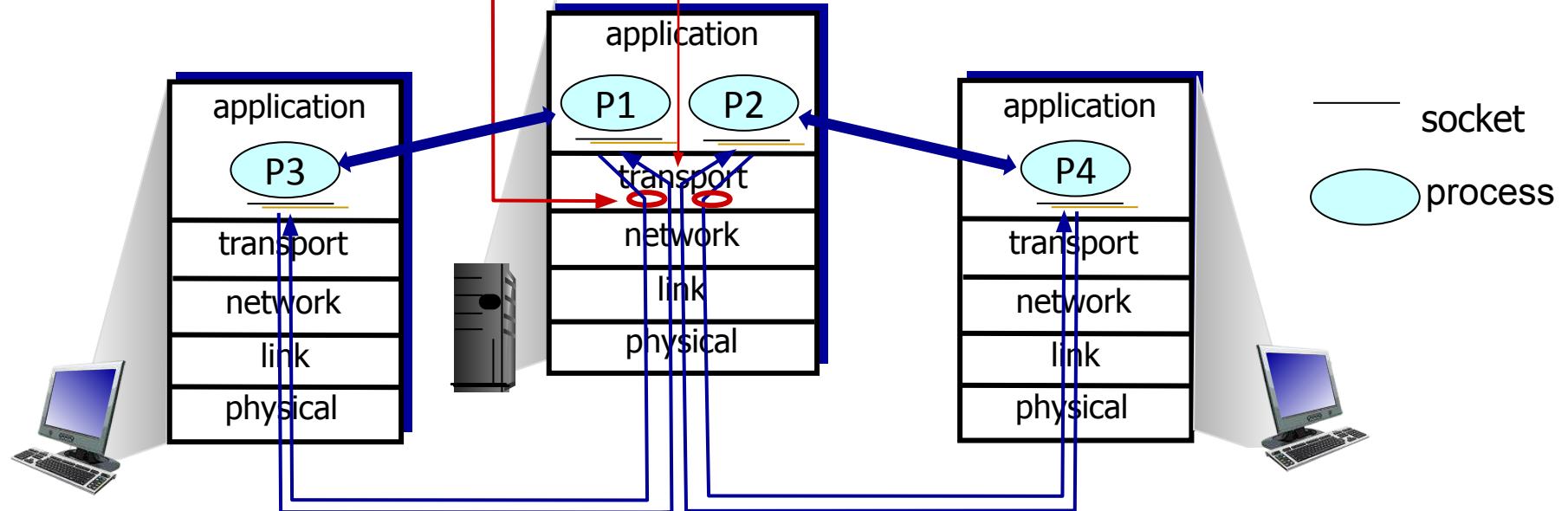


multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

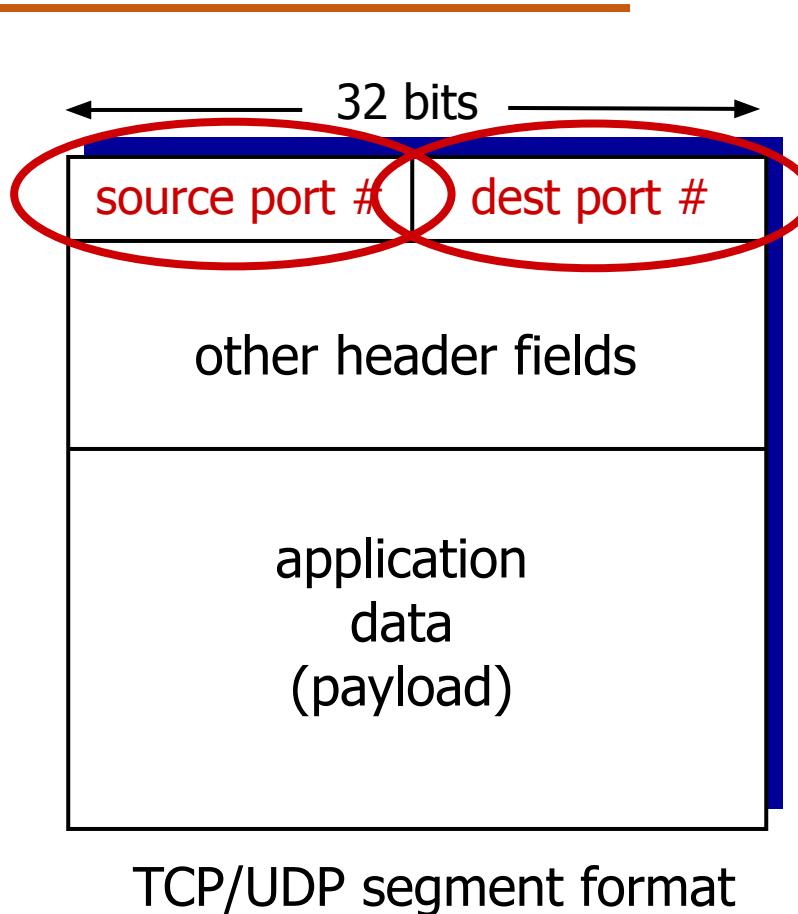
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

- host receives IP datagrams
 - each datagram has source IP address, destination IP address
 - each datagram carries one transport-layer segment
 - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



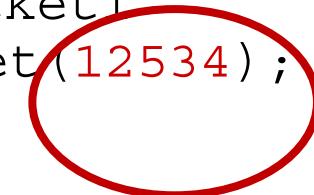
- Each port number - ranges from 0 to 65535.
- Port numbers ranging from 0 to 1023 are called **well-known port numbers** (restricted/reserved)

Connectionless demultiplexing

Recall:

- when creating socket, must specify *host-local* port #:

```
DatagramSocket mySocket1  
= new DatagramSocket(12534);
```



- when creating datagram to send into UDP socket, must specify
 - destination IP address
 - destination port #

when receiving host receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #



IP/UDP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

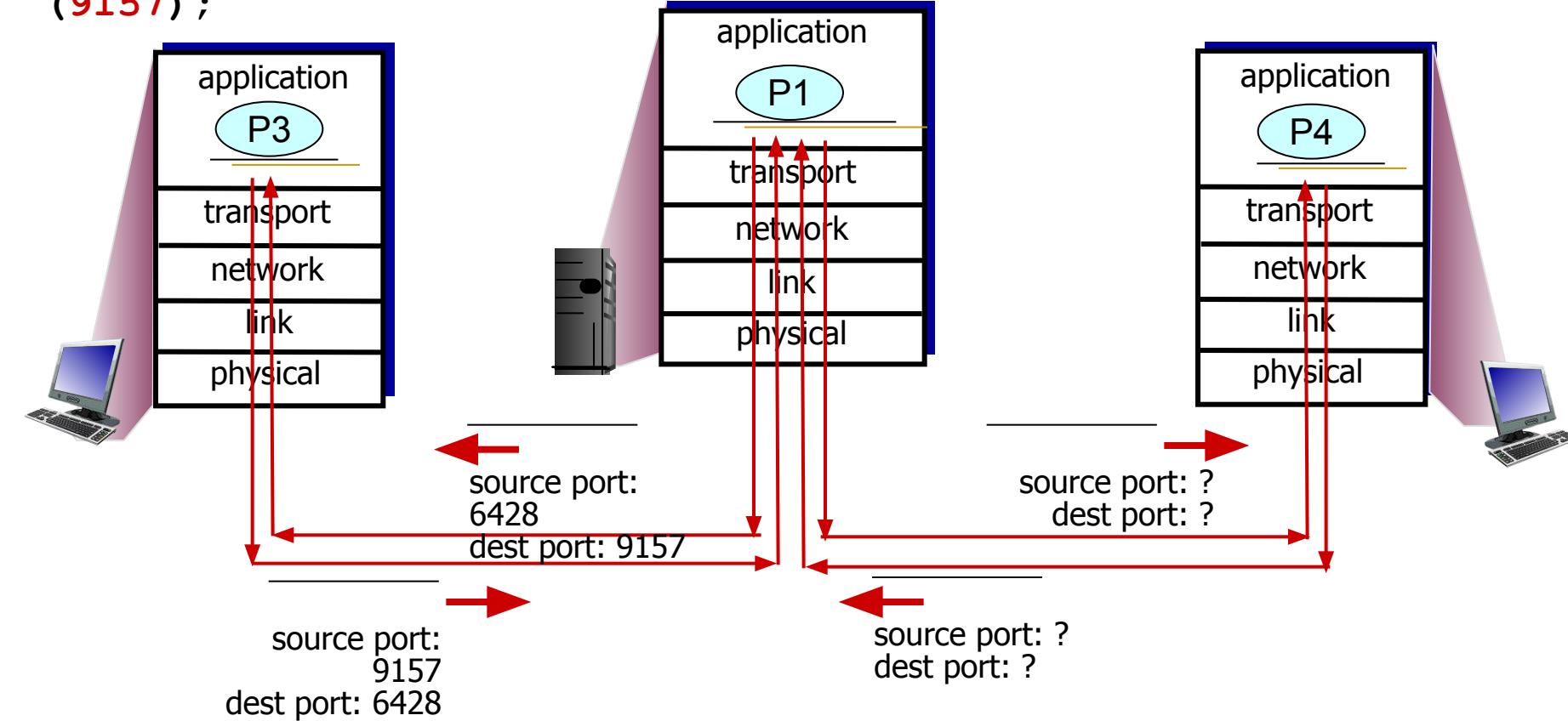
Connectionless demultiplexing: example

```
DatagramSocket mySocket2 = new DatagramSocket
(9157);
```

DatagramSocket

```
serverSocket = new
DatagramSocket
(6428);
```

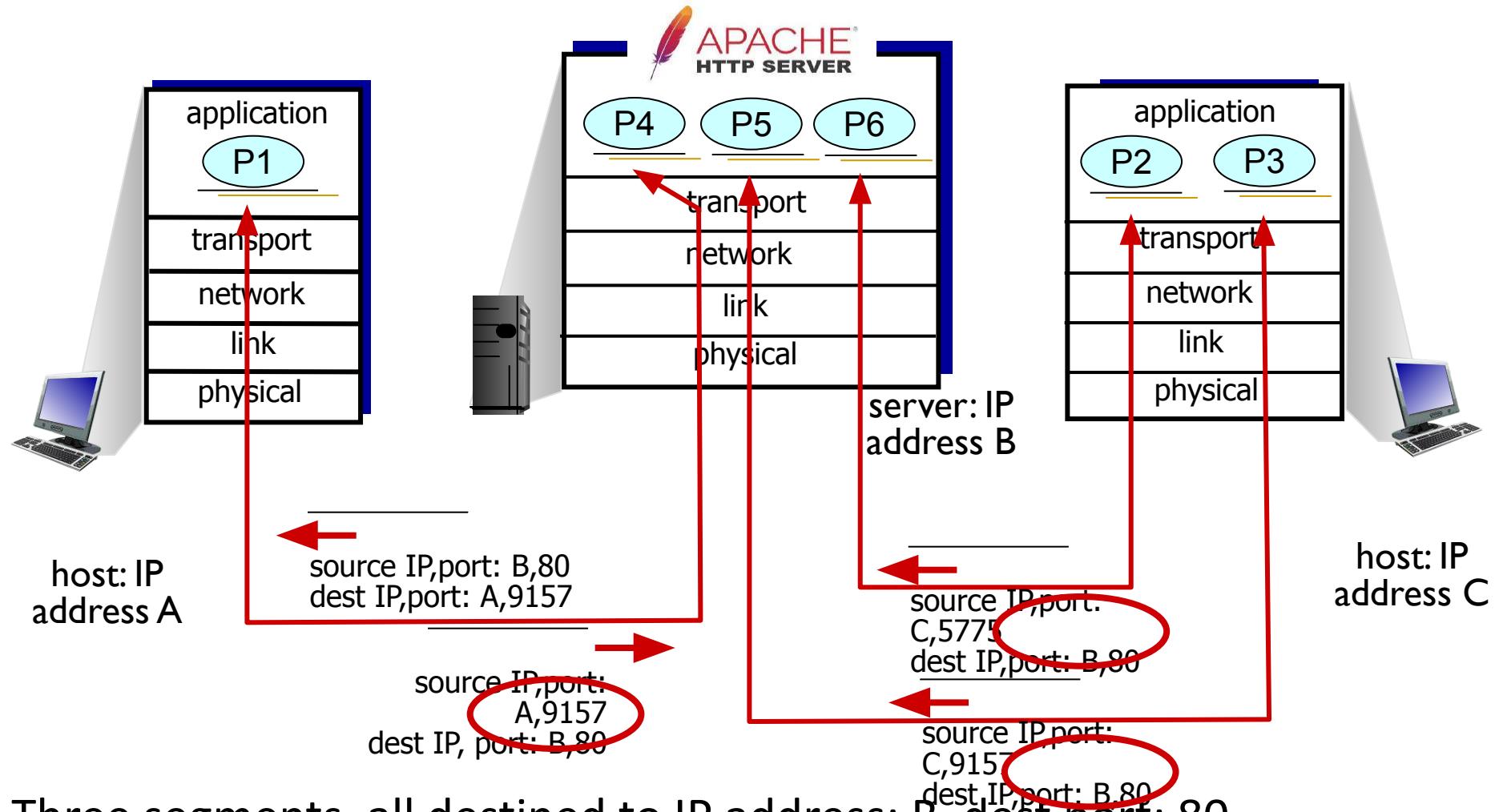
```
DatagramSocket mySocket1
= new DatagramSocket
(5775);
```



Connection-oriented demultiplexing

- TCP socket identified by 4-tuple:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- demux: receiver uses **all four values (4-tuple)** to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
 - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
 - non-persistent HTTP will have different socket for each request

Connection-oriented demultiplexing: example



Three segments, all destined to IP address: B, dest port: 80
are demultiplexed to *different* sockets

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
- Multiplexing/demultiplexing happen at all layers

- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
 - lost
 - delivered out-of-order to app
- *connectionless*:
 - no handshaking between UDP sender, receiver
 - each UDP segment handled independently of others

Why is there a UDP?

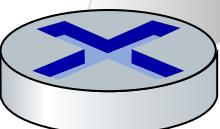
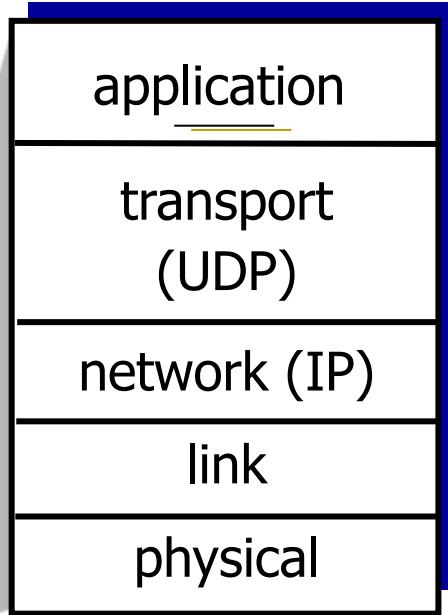
- **no connection** establishment (which can add RTT delay)
- **no connection state** at sender, receiver (buffer, seq, ack, c-c parameters)
- **small header size** (8 vs 20 bytes)
- **no congestion control**
 - UDP can blast away as fast as desired!
 - can function in the face of congestion

- UDP use:
 - streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
 - SNMP
 - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
 - add needed reliability at application layer
 - add congestion control at application layer

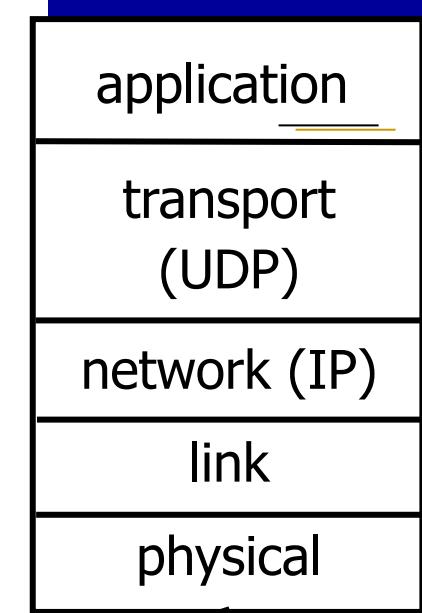
Popular Internet Applications using TCP/UDP

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Name translation	DNS	Typically UDP

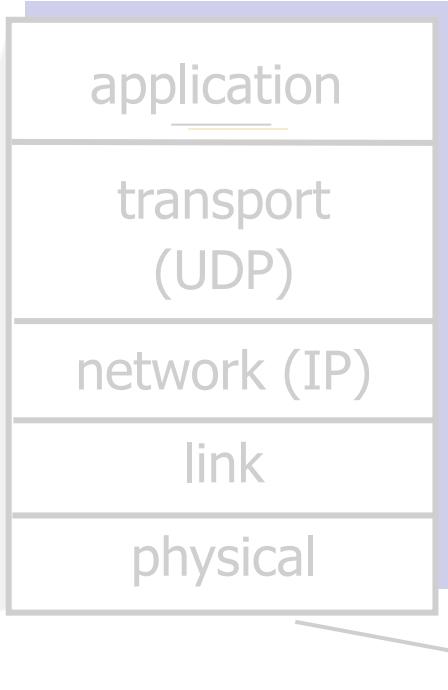
SNMP client



SNMP server



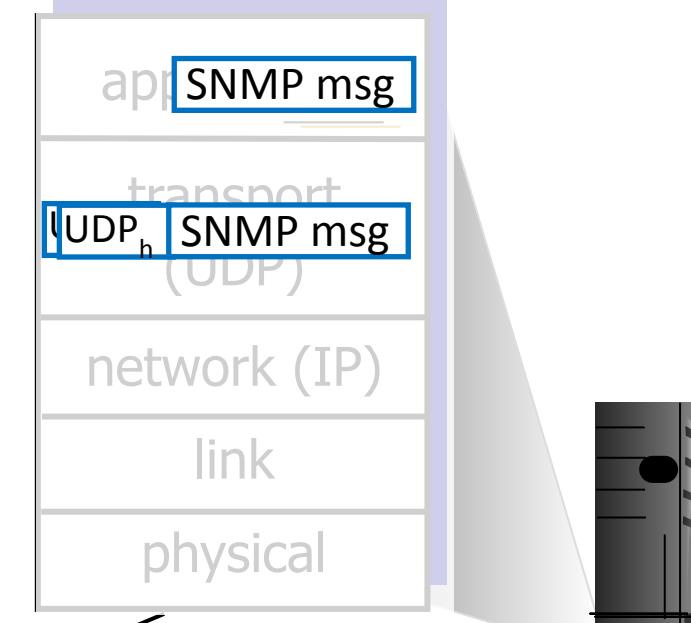
SNMP client



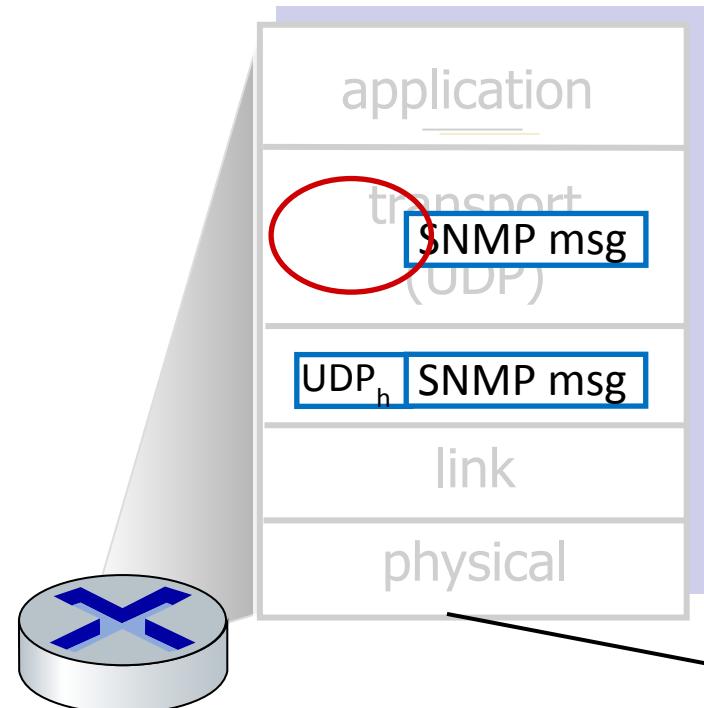
UDP sender actions:

- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP

SNMP server



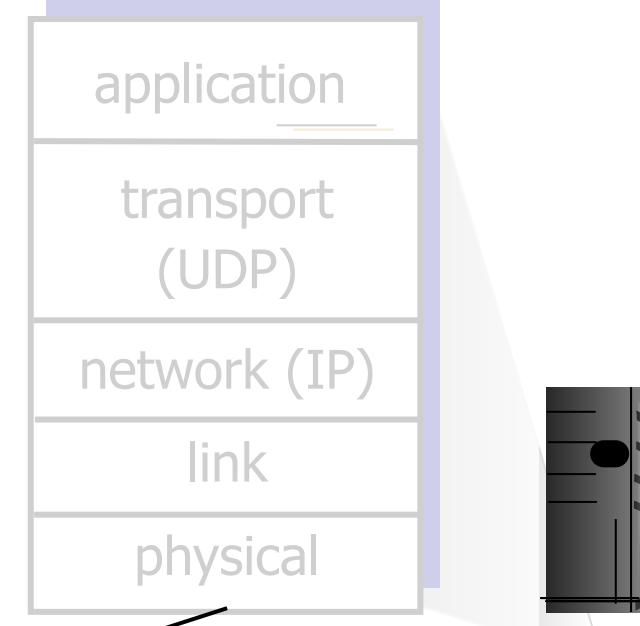
SNMP client



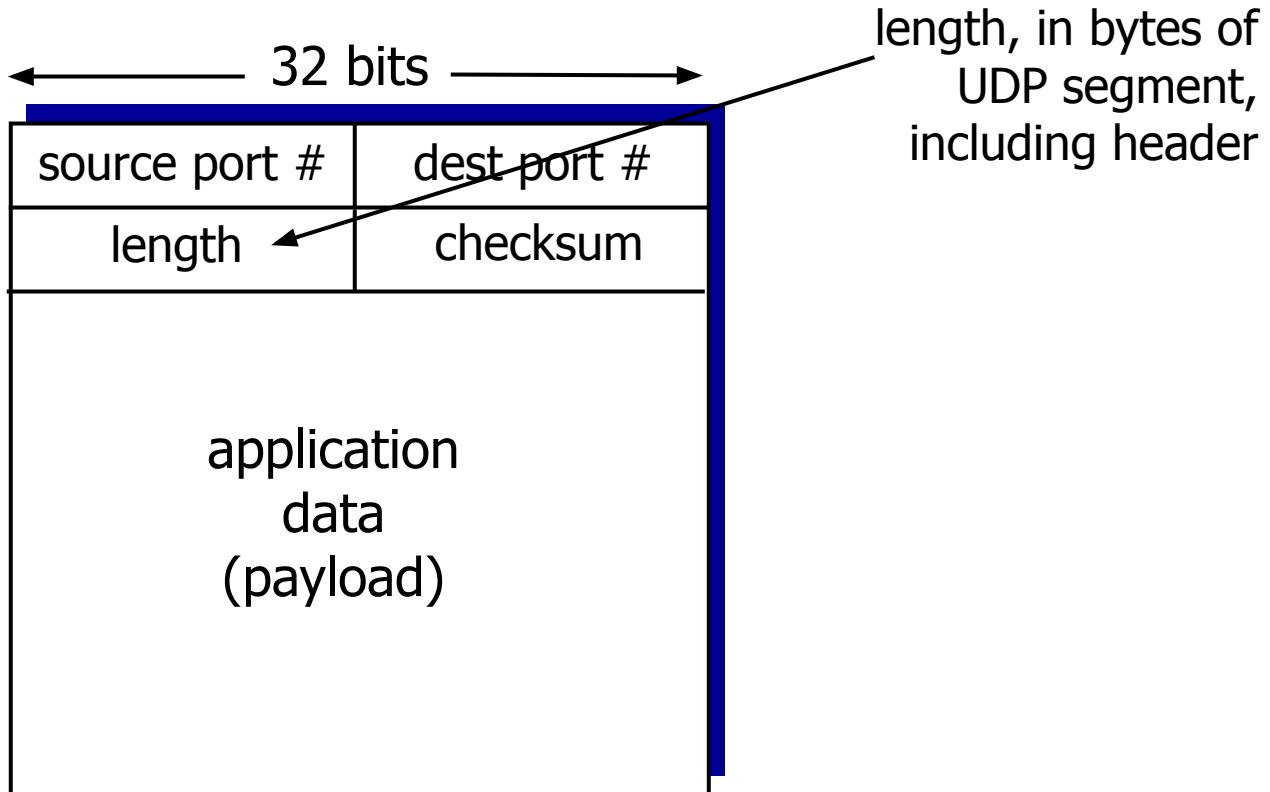
UDP receiver actions:

- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket

SNMP server



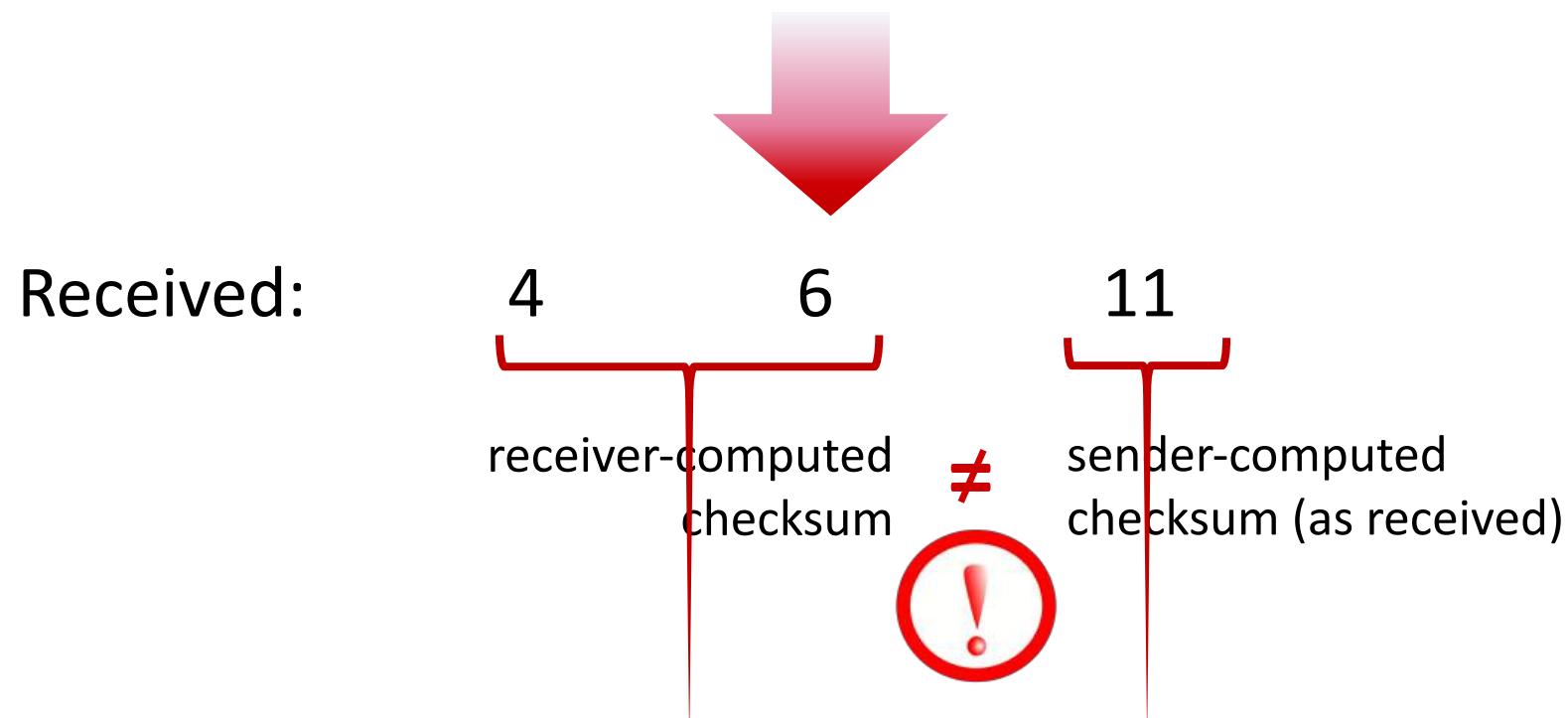
UDP: segment header



UDP segment format

Goal: detect “errors” (e.g., flipped bits) in transmitted segment

	1 st number	2 nd number	sum
Transmitted:	5	6	11



Goal: detect errors (*i.e.*, flipped bits) in transmitted segment

sender:

- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- **checksum:** addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
 - Not equal - error detected
 - Equal - no error detected. *But maybe errors nonetheless?* More later

example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	0	1	1
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Even though numbers have changed (bit flips), **no** change in checksum!

example: add three 16-bit integers

	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0	0
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	1	0	0	0	1	1	1	1	0	0	0	0	1	1	0	0
<hr/>																
sum	0	1	0	0	1	0	1	0	1	1	0	0	0	0	1	0
<hr/>																
checksum	1	0	1	1	0	1	0	1	0	0	1	1	1	1	0	1



Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Input: Zero or more quantities are externally supplied

Definiteness: Each instruction is clear and unambiguous

Finiteness: The algorithm terminates in a finite number of steps.

Effectiveness: Each instruction must be primitive and feasible

Output: At least one quantity is produced

Why do we need Algorithms?

- It is a tool for solving well-specified Computational Problem.
- Problem statement specifies in general terms relation between input and output
- Algorithm describes computational procedure for achieving input/output relationship
This Procedure is irrespective of implementation details

Why do we need to study algorithms?

Exposure to different algorithms for solving various problems helps develop skills to design algorithms for the problems for which there are no published algorithms to solve it

What do you mean by Algorithm Design Techniques?

General Approach to solving problems algorithmically .

Applicable to a variety of problems from different areas of computing

Various Algorithm Design Techniques

- Brute Force
- Divide and Conquer
- Decrease and Conquer
- Transform and Conquer
- Dynamic Programming
- Greedy Technique
- Branch and Bound
- Backtracking

Importance Framework for designing and analyzing algorithms
for new problems

- **Natural language**
 - Ambiguous
- **Pseudocode**
 - A mixture of a natural language and programming language-like structures
 - Precise and succinct.
 - Pseudocode in this course
 - omits declarations of variables
 - use indentation to show the scope of such statements as for, if, and while.
 - use \leftarrow for assignment
- **Flowchart**
 - Method of expressing algorithm by collection of connected geometric shapes

Design and Analysis of Algorithms

Methods of Specifying an Algorithm



➤ Euclid's Algorithm

Problem: Find $\text{gcd}(m, n)$, the greatest common divisor of two nonnegative, not both zero integers m and n

Examples: $\text{gcd}(60, 24) = 12$, $\text{gcd}(60, 0) = 60$, $\text{gcd}(0, 0) = ?$

Euclid's algorithm is based on repeated application of equality

$$\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$$

until the second number becomes 0, which makes the problem trivial.

Example: $\text{gcd}(60, 24) = \text{gcd}(24, 12) = \text{gcd}(12, 0) = 12$

Two descriptions of Euclid's algorithm

Euclid's algorithm for computing $\text{gcd}(m,n)$

Step 1 If $n = 0$, return m and stop; otherwise go to Step 2

Step 2 Divide m by n and assign the value of the remainder to r

Step 3 Assign the value of n to m and the value of r to n . Go to step 1.

ALGORITHM Euclid(m,n)

//computes $\text{gcd}(m,n)$ by Euclid's method

//Input: Two nonnegative,not both zero integers

//Output:Greatest common divisor of m and n

while $n \neq 0$ do

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

ALGORITHM *SequentialSearch*($A[0..n - 1]$, K)

//Searches for a given value in a given array by sequential search

//Input: An array $A[0..n - 1]$ and a search key K

//Output: The index of the first element of A that matches K

// or -1 if there are no matching elements

$i \leftarrow 0$

while $i < n$ **and** $A[i] \neq K$ **do**

$i \leftarrow i + 1$

if $i < n$ **return** i

else return -1

- To determine resource consumption
 - CPU time
 - Memory space
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm for solving the problem

- A measure of the performance of an algorithm
- An algorithm's performance is characterized by
 - Time complexity
 - How fast an algorithm maps input to output as a function of input
 - Space complexity
 - amount of memory units required by the algorithm in addition to the memory needed for its input and output

How to determine complexity of an algorithm?

- Experimental study(Performance Measurement)
- Theoretical Analysis (Performance Analysis)

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

Design and Analysis of Algorithms

Performance Analysis



- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n .
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Design and Analysis of Algorithms

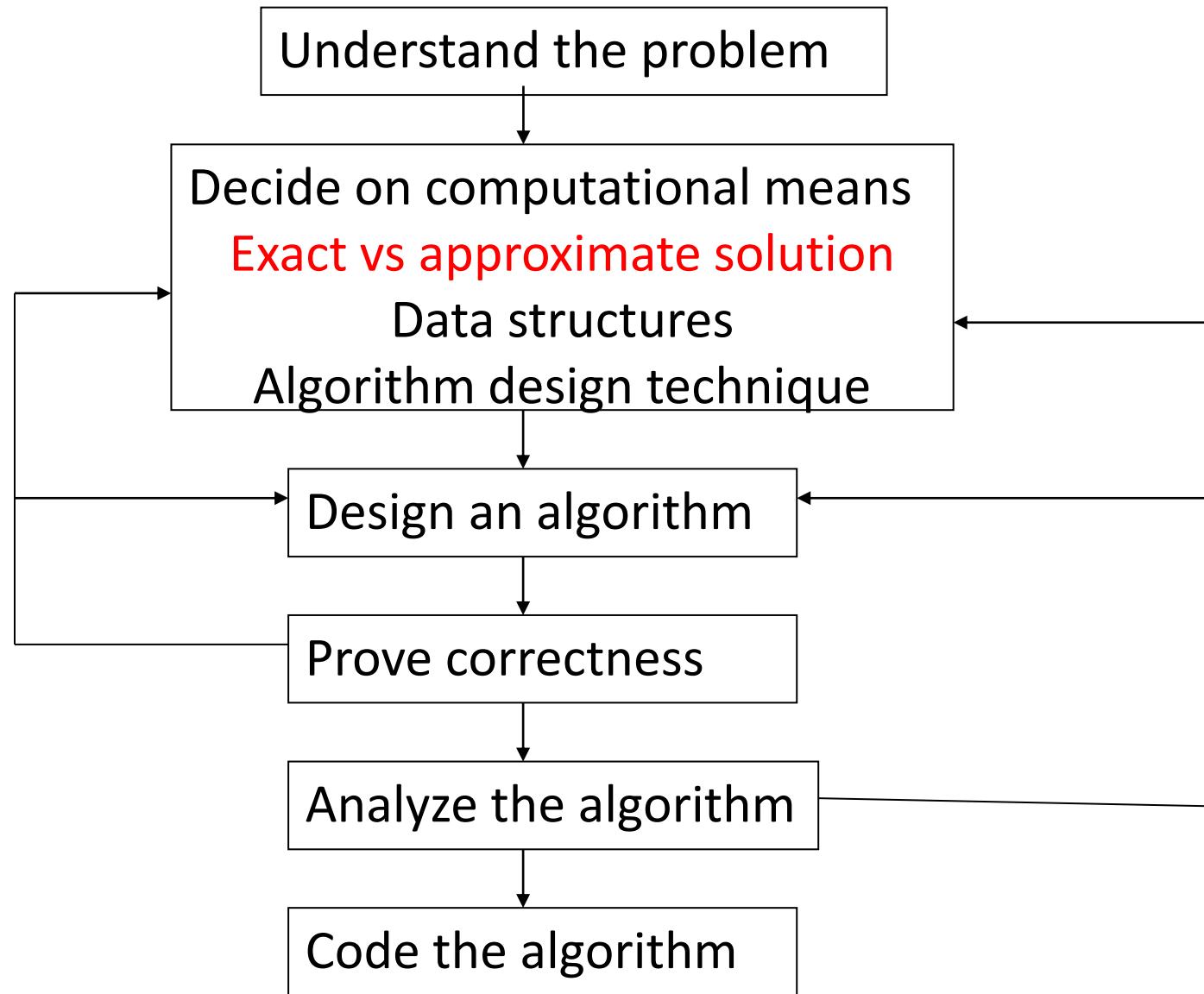
Computational Means



Computational Device the algorithm is intended for

RAM Sequential Algorithms

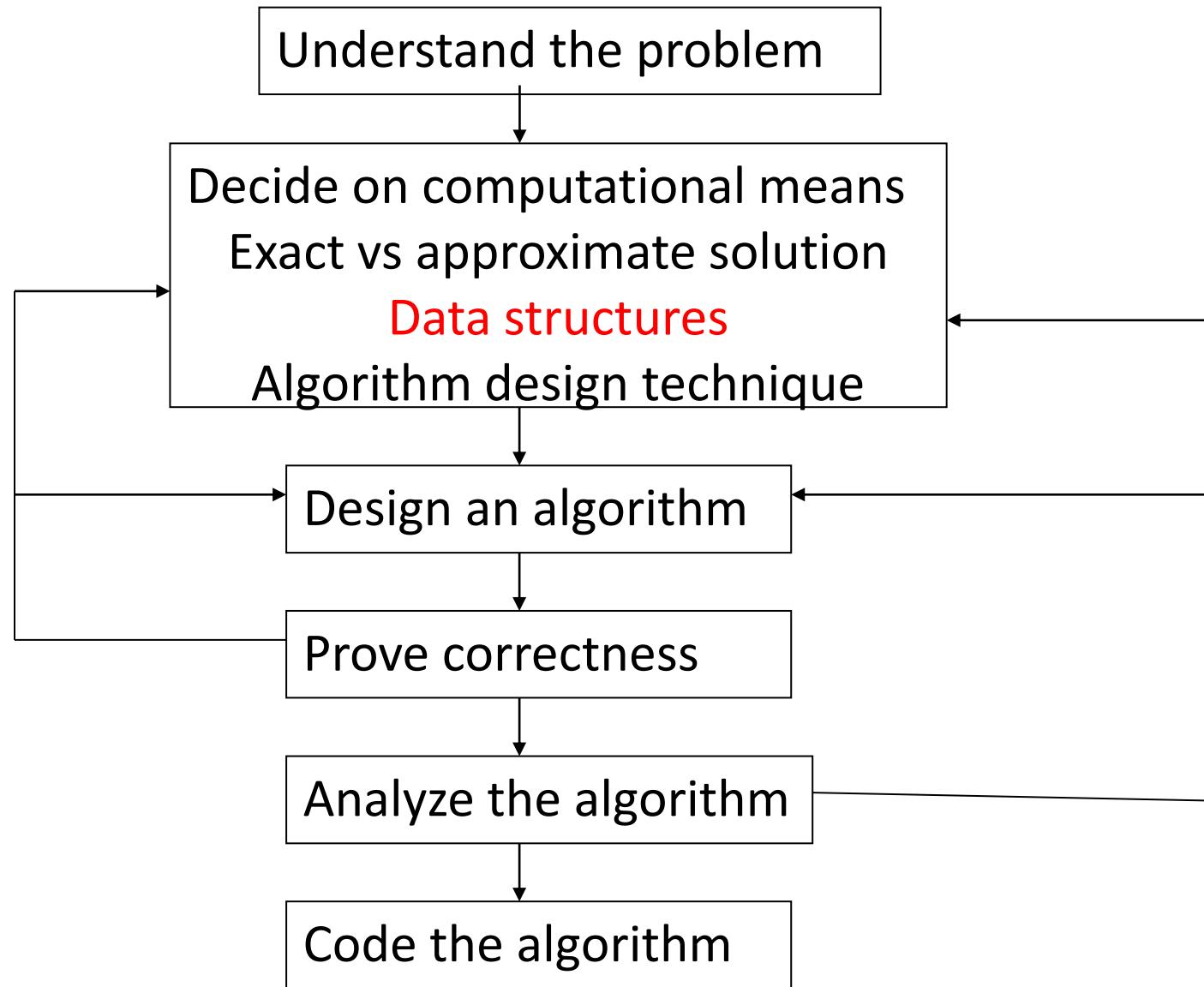
PRAM Parallel Algorithms



Travelling Salesman Problem

NP complete!!!

Approximate algorithm can be used to solve it



Design and Analysis of Algorithms

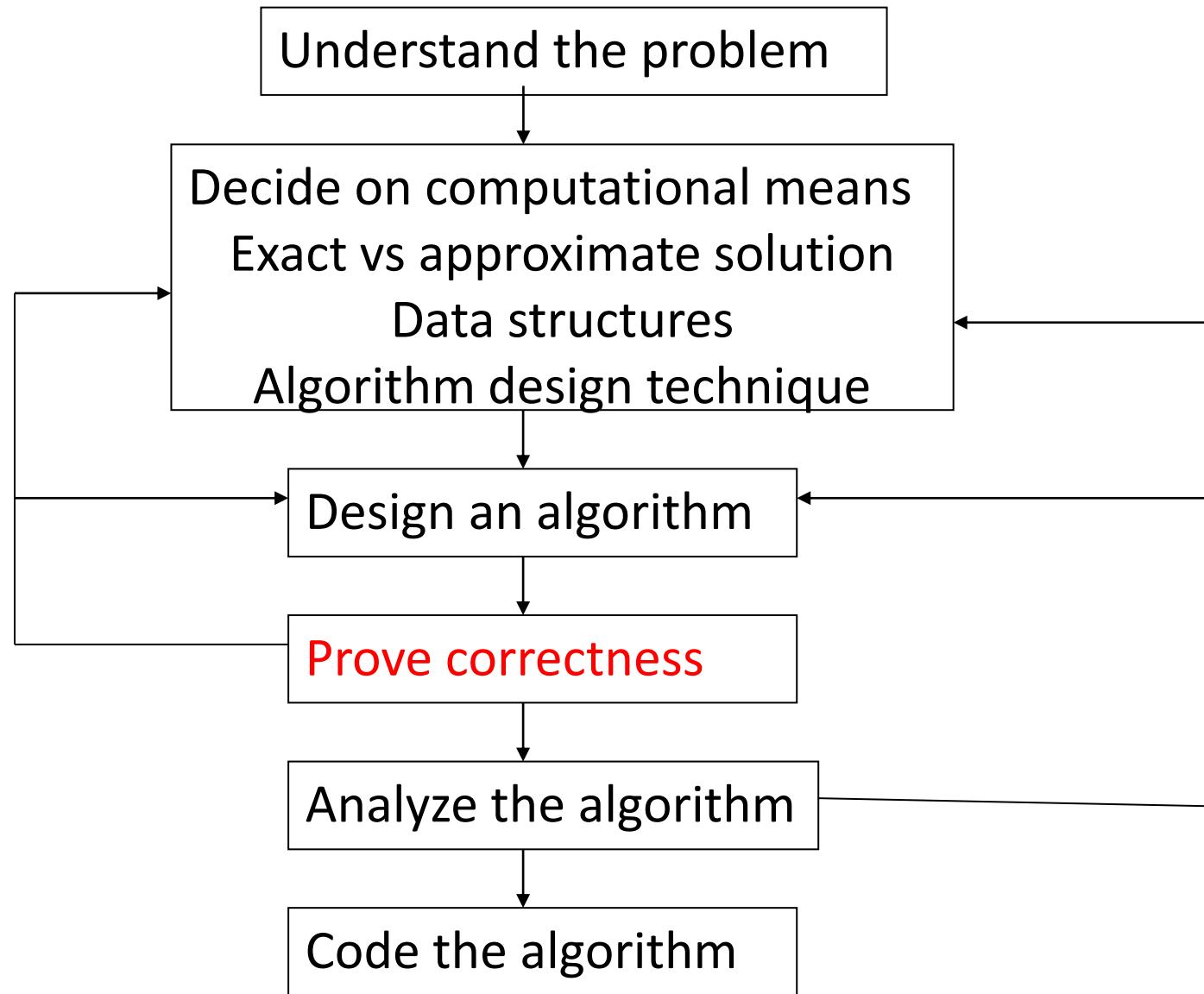
Deciding on Data Structure



- Linear
 - Linear list, Stack, Queues
- Non Linear
 - Trees, Graphs

Choice of Data structure for solving a problem using an algorithm may dramatically impact its time complexity

Dijkstra Algorithm
 $O(V \log V + E)$ with Fibonacci heap



Exact algorithms

Proving that algorithm yields a correct result for legitimate input in finite amount of time

Approximation algorithms

Error produced by algorithm does not exceed a predefined limit

- Efficiency
 - Time efficiency
 - Space efficiency
- Simplicity
- Generality
 - Design an algorithm for the problem posed in more general terms
 - Design an algorithm that can handle a range of inputs that is natural for the problem at hand

- Efficient implementation
- Correctness of program
 - Mathematical Approach: Formal verification for small programs
 - Practical Methods: Testing and Debugging
- Code optimization

- Rearrange the items of a given list in ascending order.
 - Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$
 - Output: A reordering $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.
- Why sorting?
 - Help searching
 - Algorithms often use sorting as a key subroutine.
- Sorting key

A specially chosen piece of information used to guide sorting.
Example: sort student records by SRN.

- Rearrange the items of a given list in ascending order.
- Examples of sorting algorithms
 - Selection sort
 - Bubble sort
 - Insertion sort
 - Merge sort
 - Heap sort ...
- Evaluate sorting algorithm complexity: the number of key comparisons.
- Two properties
 - Stability: A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
 - In place : A sorting algorithm is in place if it does not require extra memory, except, possibly for a few memory units.

Design and Analysis of Algorithms

Important Problem Types: Searching



Find a given value, called a **search key**, in a given set.

Examples of searching algorithms

- Sequential searching
- Binary searching...

A string is a sequence of characters from an alphabet.

Text strings: letters, numbers, and special characters.

String matching: searching for a given word/pattern in a text.

Text: I am a **computer** science graduate

Pattern: computer

Definition

Graph G is represented as a pair $G = (V, E)$,
where V is a finite set of vertices and E is a finite set of edges

Modeling real-life problems

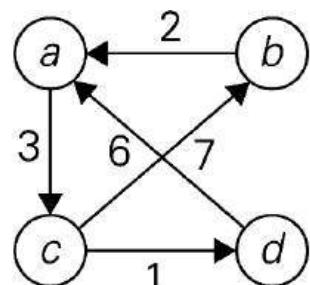
- Modeling WWW
- communication networks
- Project scheduling ...

Examples of graph algorithms

- Graph traversal algorithms
- Shortest-path algorithms
- Topological sorting

Shortest paths in a graph

To find the distances from each vertex to all other vertices.



(a)

$$W = \begin{bmatrix} a & b & c & d \\ a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{bmatrix}$$

(b)

$$D = \begin{bmatrix} a & b & c & d \\ a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{bmatrix}$$

(c)

FIGURE 8.5 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

Minimum cost spanning tree

- A spanning tree of a connected graph is its connected acyclic sub graph (i.e. a tree).

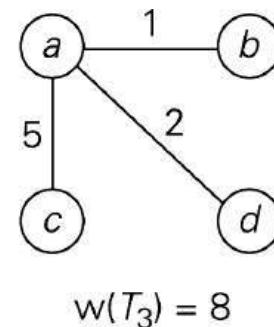
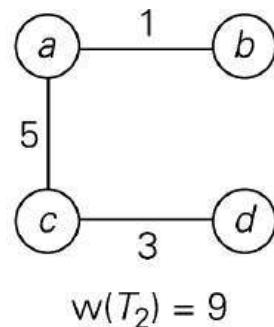
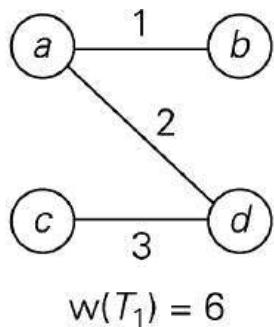
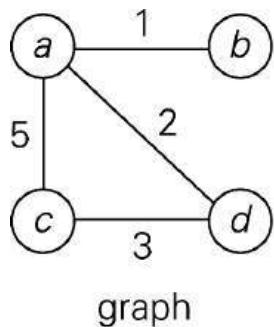
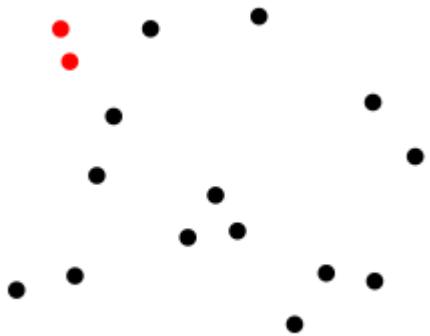
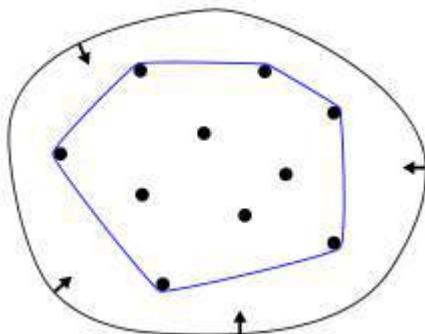


FIGURE 9.1 Graph and its spanning trees; T_1 is the minimum spanning tree

Closest Pair problem



Convex Hull Problem



- Solving Equations
- Computing definite integrals
- Evaluating functions

Design and Analysis of Algorithms

Analysis Framework



What do you mean by analysing an algorithm?

Investigation of Algorithm's efficiency with respect to two resources

- Time
- Space

What is the need for Analysing an algorithm?

- To determine resource consumption
 - CPU time
 - Memory space
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm

- A measure of the performance of an algorithm
- An algorithm's performance depends on
 - *internal factors*
 - Time required to run
 - Space (memory storage) required to run
 - *external factors*
 - Speed of the computer on which it is run
 - Quality of the compiler
 - Size of the input to the algorithm

Design and Analysis of Algorithms

Performance of Algorithm



important Criteria for performance:

- Space efficiency - the memory required, also called, space complexity
- Time efficiency - the time required, also called time complexity

$$S(P) = C + SP(I)$$

- Fixed Space Requirements (C)
Independent of the characteristics of the inputs and outputs
 - instruction space
 - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements (SP(I))
dependent on the instance characteristic I
 - number, size, values of inputs and outputs associated with I
 - recursive stack space, formal parameters, local variables, return address

$$S(P)=C+S_P(I)$$

```
float rsum(float list[ ], int n)
{
    if (n)
        return rsum(list, n-1) + list[n-1]
    return 0
}
```

$$S_{\text{sum}}(I)=S_{\text{sum}}(n)=6n$$

Type	Name	Number of bytes
parameter: float	list []	2
parameter: integer	n	2
return address:(used internally)		2
TOTAL per recursive call		6

Time Complexity

$$T(P) = C + T_P(I)$$

- Compile time (C)
independent of instance characteristics

- run (execution) time T_P

Experimental study

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Get an accurate measure of the actual running time
- Use a method like `System.currentTimeMillis()`
- Plot the results

Design and Analysis of Algorithms

Limitations of Experimental study



- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n .
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Two approaches:

1. Order of magnitude/asymptotic categorization –

This uses coarse categories and gives a general idea of performance.

If algorithms fall into the same category, if data size is small, or if performance is critical, use method 2

2. Estimation of running time -

1. *operation counts* - select operation(s) that are executed most frequently and determine how many times each is done.

2. *step counts* - determine the total number of steps, possibly lines of code, executed by the program.

Design and Analysis of Algorithms

Analysis Framework



- Measuring an input's size
- Measuring running time
- Orders of growth (of the algorithm's efficiency function)
- Worst-base, best-case and average efficiency

Efficiency is defined as a function of input size.

Input size depends on the problem.

Example 1, what is the input size of the problem of sorting n numbers?

Example 2, what is the input size of adding two n by n matrices?

Design and Analysis of Algorithms

Units for Measuring Running Time

- Measure the running time using standard unit of time measurements, such as seconds, minutes?
Depends on the speed of the computer.
- count the number of times each of an algorithm's operations is executed.
(step count method)
Difficult and unnecessary
- count the number of times an algorithm's basic operation is executed.

Basic operation: the most important operation of the algorithm, the operation contributing the most to the total running time.

For example, the basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

Analysis in the RAM Model

SmartFibonacci(n)	<i>cost</i>	<i>times</i> ($n > 1$)
1 if $n = 0$	c_1	1
2 then return 0	c_2	0
3 elseif $n = 1$	c_3	1
4 then return 1	c_4	0
5 else $p\text{prev} \leftarrow 0$	c_5	1
6 $\text{prev} \leftarrow 1$	c_6	1
7 for $i \leftarrow 2$ to n	c_7	n
8 do $f \leftarrow \text{prev} + p\text{prev}$	c_8	$n - 1$
9 $p\text{prev} \leftarrow \text{prev}$	c_9	$n - 1$
10 $\text{prev} \leftarrow f$	c_{10}	$n - 1$
11 return f	c_{11}	1

$$T(n) = c_1 + c_3 + c_5 + c_6 + c_{11} + nc_7 + (n - 1)(c_8 + c_9 + c_{10})$$

$T(n) = nC_1 + C_2 \Rightarrow T(n)$ is a *linear function* of n

Input Size and Basic Operation Examples

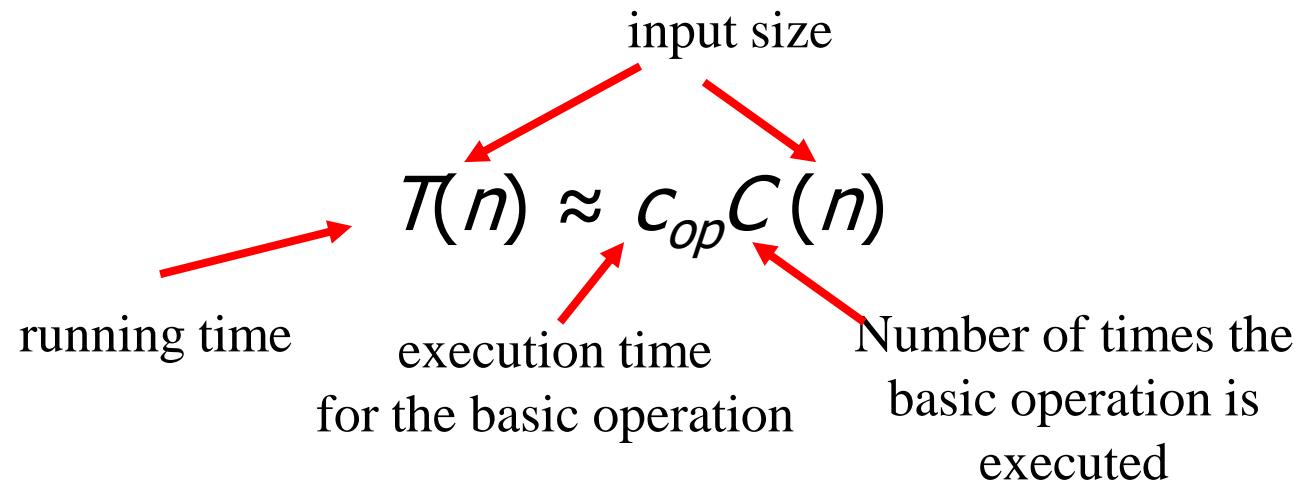
<i>Problem</i>	<i>Input size measure</i>	<i>Basic operation</i>
Search for a key in a list of n items	Number of items in list, n	Key comparison
Add two n by n matrices	Dimensions of matrices, n	addition
multiply two matrices	Dimensions of matrices, n	multiplication

Theoretical Analysis of Time Efficiency: Basic operation count

Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size.

Assuming $C(n) = (1/2)n(n-1)$,

how much longer will the algorithm run if we double the input size?



C(n) Basic Operation Count

- The efficiency analysis framework ignores the multiplicative constants of C(n) and focuses on the orders of growth of the C(n).

- Simple characterization of the algorithm's efficiency by identifying relatively significant term in the C(n).

Why do we care about the order of growth of an algorithm's efficiency function, i.e., the total number of basic operations?

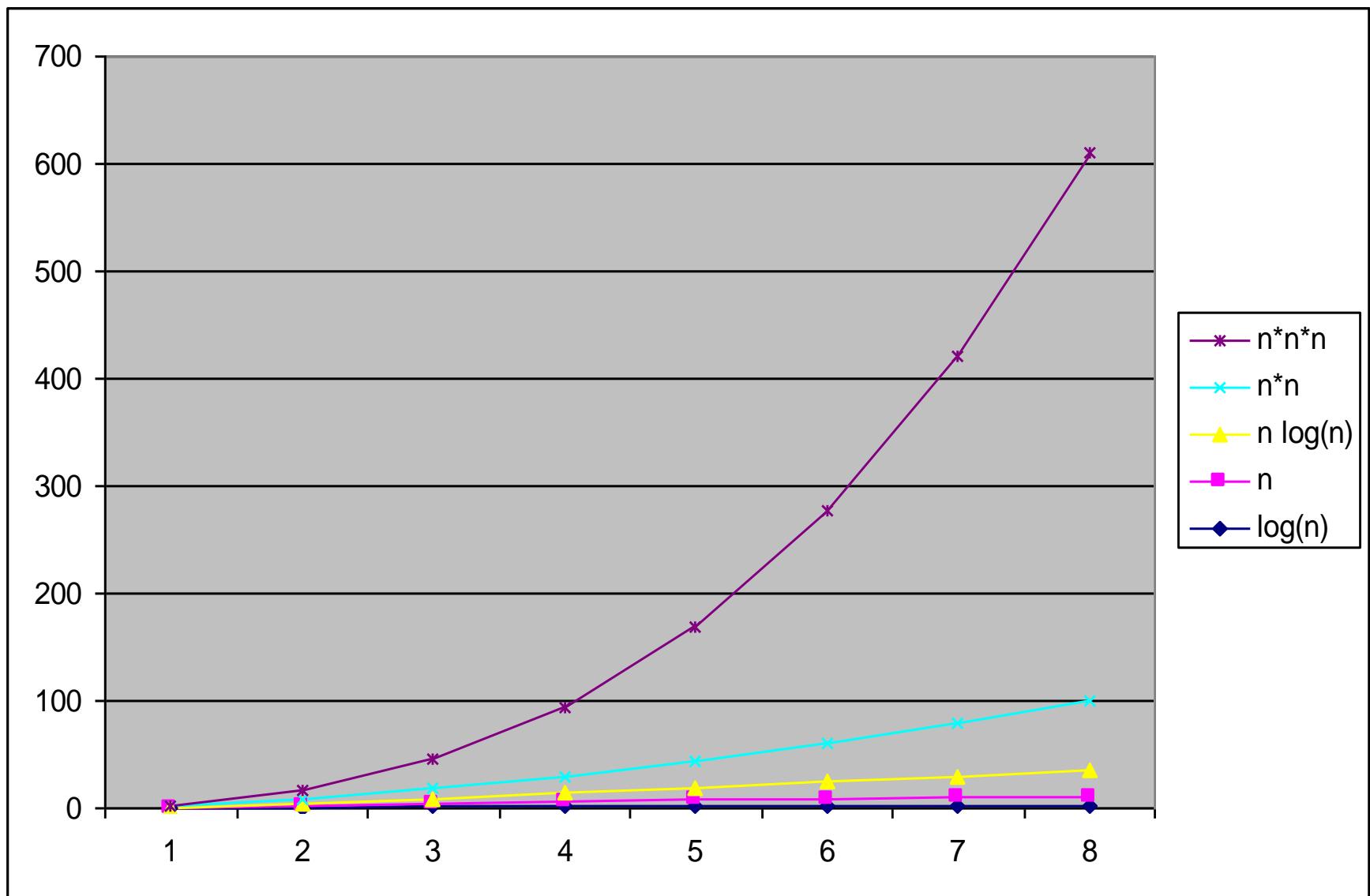
- Because, for smaller inputs, it is difficult to distinguish inefficient algorithms vs. efficient ones.
- For example, if the number of basic operations of two algorithms to solve a particular problem are n and n^2 respectively, then
 - if $n = 2$, Basic operation will be executed 2 and 4 times respectively for algorithm1 and 2.
Not much difference!!!
 - On the other hand, if $n = 10000$, then it does makes a difference whether the number of times the basic operation is executed is n or n^2 .

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$	Exponential-growth functions
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$	
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$	
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9			
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}			
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}			
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}			

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Orders of growth:

- consider only the leading term of a formula
- ignore the constant coefficient.



1	constant
$\log n$	logarithmic
n	linear
$n \log n$	$n\text{-log-}n$
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

- Algorithm efficiency depends on the input size n
- For some algorithms efficiency depends on type of input.

Example: Sequential Search

Problem: Given a list of n elements and a search key K , find an element equal to K , if any.

Algorithm: Scan the list and compare its successive elements with K until either a matching element is found (successful search) or the list is exhausted (unsuccessful search)

Given a sequential search problem of an input size of n ,
what kind of input would make the running time the longest?
How many key comparisons?

➤ Worst case Efficiency

- Efficiency (# of times the basic operation will be executed) for the worst case input of size n.
- The algorithm runs the longest among all possible inputs of size n.

➤ Best case

- Efficiency (# of times the basic operation will be executed) for the best case input of size n.
- The algorithm runs the fastest among all possible inputs of size n.

➤ Average case:

- Efficiency (#of times the basic operation will be executed) for a typical/random input of size n.
- NOT the average of worst and best case

➤ How to find the average case efficiency?

ALGORITHM SequentialSearch(A[0..n-1], K)

//Searches for a given value in a given array by sequential search

//Input: An array A[0..n-1] and a search key K

//Output: Returns the index of the first element of A that matches K or -1 if there are no matching elements

i \leftarrow 0

while i < n and A[i] \neq K do

 i \leftarrow i + 1

if i < n //A[i] = K

 return i

else

 return -1

- Worst-Case: $C_{worst}(n) = n$
- Best-Case: $C_{best}(n) = 1$
- Average-Case
 - from $(n+1)/2$ to $(n+1)$

Let 'p' be the probability that key is found in the list

Assumption: All positions are equally probable

Case1: key is found in the list

$$C_{\text{avg,case1}}(n) = p * (1 + 2 + \dots + n) / n = p * (n + 1) / 2$$

Case2: key is not found in the list

$$C_{\text{avg, case2}}(n) = (1-p) * (n)$$

$$C_{\text{avg}}(n) = p(n + 1) / 2 + (1 - p)(n)$$

Orders of growth of an algorithm's basic operation count is important

How do we compare order of growth??

Using Asymptotic Notations

A way of comparing functions that ignores constant factors and small input sizes

$O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$

$\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$

$\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$

$o(g(n))$: class of functions $f(n)$ that grow at slower rate than $g(n)$

$\omega(g(n))$: class of functions $f(n)$ that grow at faster rate than $g(n)$

O-notation

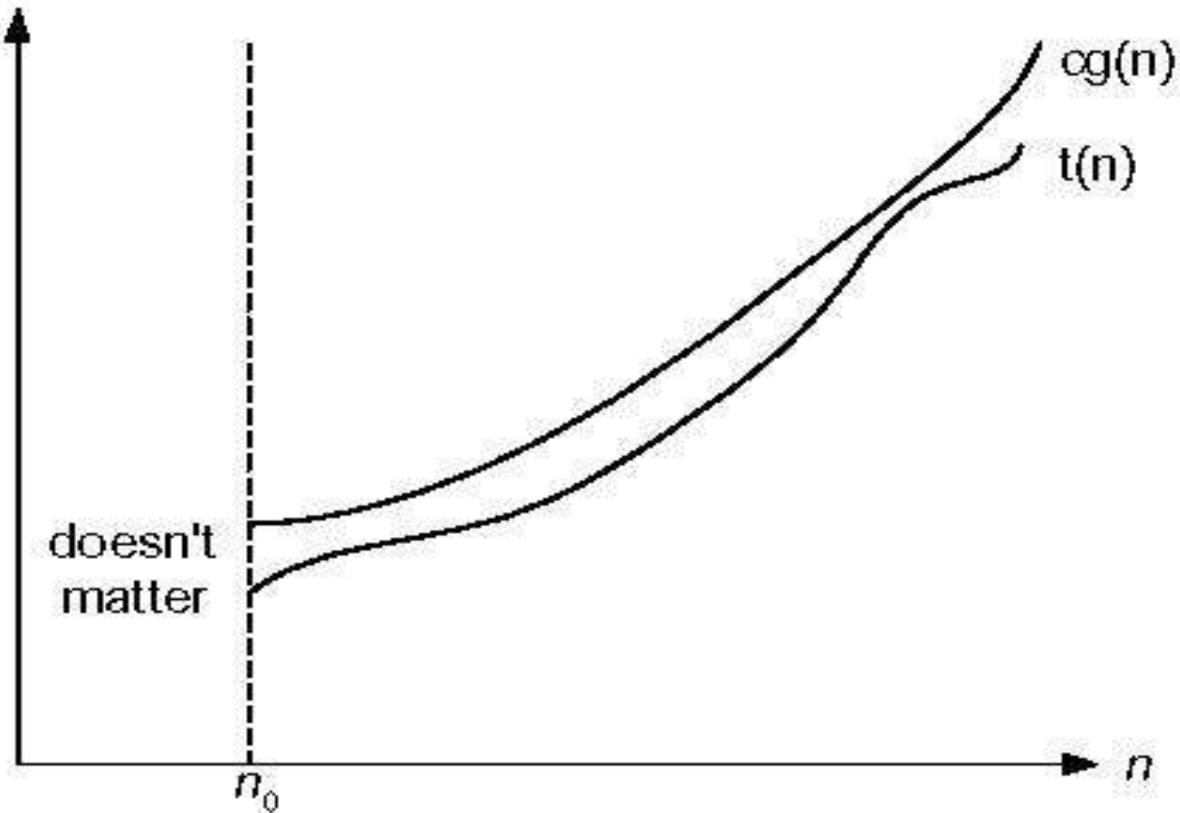


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

O-notation

Formal definition

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n ,
i.e., if there exist **some** positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$

Exercises: prove the following using the above definition

$$10n^2 \in O(n^2)$$

$$10n^2 + 2n \in O(n^2)$$

$$100n + 5 \in O(n^2)$$

$$5n+20 \in O(n)$$

Design and Analysis of Algorithms

Ω -notation

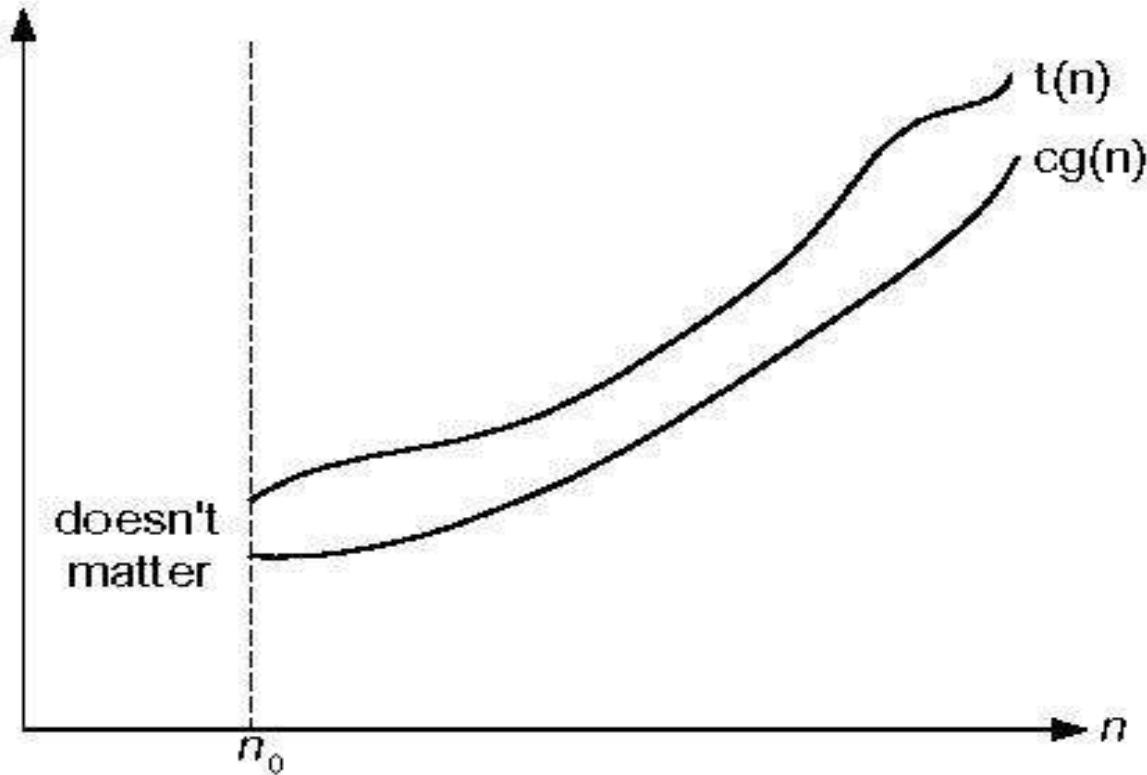


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Formal definition

A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n ,

i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \text{ for all } n \geq n_0$$

Exercises: prove the following using the above definition

$$10n^2 \in \Omega(n^2)$$

$$10n^2 + 2n \in \Omega(n^2)$$

$$10n^3 \in \Omega(n^2)$$

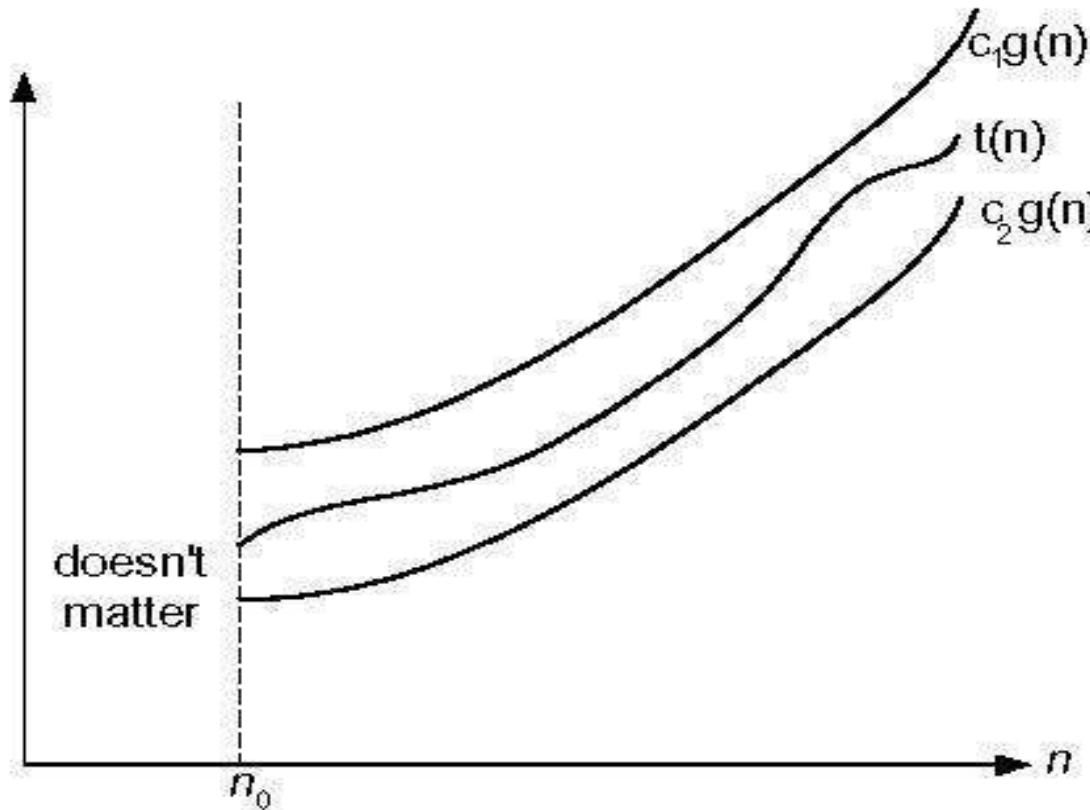


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Θ -notation

Formal definition

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n ,

i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

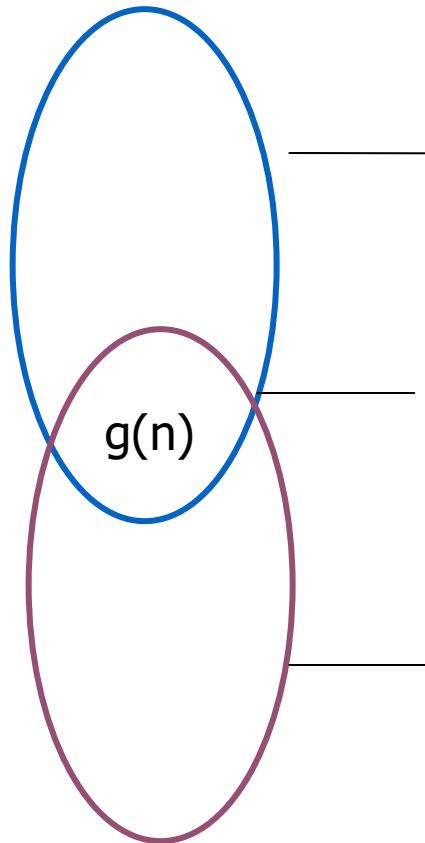
$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$

Exercises: prove the following using the above definition

$$10n^2 \in \Theta(n^2)$$

$$10n^2 + 2n \in \Theta(n^2)$$

$$(1/2)n(n-1) \in \Theta(n^2)$$



\geq

$\Omega(g(n))$, functions that grow at least as fast as $g(n)$

$=$

$\Theta(g(n))$, functions that grow at the same rate as $g(n)$

\leq

$O(g(n))$, functions that grow no faster than $g(n)$

Little-o Notation

Formal Definition:

A function $t(n)$ is said to be in Little-o($g(n)$), denoted $t(n) \in o(g(n))$,
if for any positive constant c and some nonnegative integer n_0

$$0 \leq t(n) < cg(n) \text{ for all } n \geq n_0$$

Little-o Notation

Example: $f(n) = 2n^2$ and $g(n) = n^2$ and $c = 2$

$f(n) = O(g(n))$ - Big-O

$f(n) \neq o(g(n))$ - little-o

If $f(n) = 2n$ & $g(n) = n^2$,
then for any value of $c > 0$,

∴ $f(n) < c(n^2)$
 $f(n) \neq o(g(n))$

Note : For non-negative functions, $f(n)$ and $g(n)$,
 $f(n)$ is little o of $g(n)$, if and only if,

$f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$
[strict upper bound, no lower bound]

Little Omega Notation

Formal Definition:

A function $t(n)$ is said to be in Little- $\omega(g(n))$, denoted $t(n) \in \omega(g(n))$,
if for any positive constant c and some nonnegative integer n_0

$$t(n) > cg(n) \geq 0 \text{ for all } n \geq n_0$$

Little Omega Notation

Example : If $f(n) = 3n^2 + 2$, $g(n) = n$
then for any value of $c > 0$
 $f(n) > cg(n)$

∴ $f(n) = \omega(n)$

Note : For non-negative functions, $f(n)$ and $g(n)$,
 $f(n)$ is little ω of $g(n)$, if and only if

$f(n) = \Omega(g(n))$ and $f(n) \neq \Theta(g(n))$
[strict lower bound, no upper bound]

Theorems

➤ If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

For example,

$$5n^2 + 3n\log n \in O(n^2)$$

➤ If $t_1(n) \in \Theta(g_1(n))$ and $t_2(n) \in \Theta(g_2(n))$, then

$$t_1(n) + t_2(n) \in \Theta(\max\{g_1(n), g_2(n)\})$$

➤ $t_1(n) \in \Omega(g_1(n))$ and $t_2(n) \in \Omega(g_2(n))$, then

$$t_1(n) + t_2(n) \in \Omega(\max\{g_1(n), g_2(n)\})$$

Implication: The algorithm's overall efficiency will be determined by the part with a larger order of growth, i.e., its least efficient part.

Design and Analysis of Algorithms

Using Limits to Compare Order of Growth

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

Case1: $t(n) \in O(g(n))$

Case2: $t(n) \in \Theta(g(n))$

Case3: $g(n) \in O(t(n))$ $t'(n)$ and $g'(n)$ are first-order derivatives of $t(n)$ and $g(n)$

L'Hopital's Rule $\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{t'(n)}{g'(n)}$

Stirling's Formula $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ for large values of n

Design and Analysis of Algorithms

Using Limits to Compare Order of Growth: Example 1

Compare the order of growth of $f(n)$ and $g(n)$ using method of limits

$$t(n) = 5n^3 + 6n + 2, \quad g(n) = n^4$$

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{5n^3 + 6n + 2}{n^4} = \lim_{n \rightarrow \infty} \left(\frac{5}{n} + \frac{6}{n^3} + \frac{2}{n^4} \right) = 0$$

As per case1

$$t(n) = O(g(n))$$

$$5n^3 + 6n + 2 = O(n^4)$$

Design and Analysis of Algorithms

Using Limits to Compare Order of Growth: Example 2

$$t(n) = \sqrt{5n^2 + 4n + 2}$$

using the Limits approach determine $g(n)$ such that $f(n) = \Theta(g(n))$

Leading term in square root n^2

$$g(n) = \sqrt{n^2} = n$$

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{\sqrt{5n^2 + 4n + 2}}{\sqrt{n^2}}$$

$$= \lim_{n \rightarrow \infty} \sqrt{\frac{5n^2 + 4n + 2}{n^2}} = \lim_{n \rightarrow \infty} \sqrt{5 + \frac{4}{n} + \frac{2}{n^2}} = \sqrt{5}$$

non-zero constant

Hence, $t(n) = \Theta(g(n)) = \Theta(n)$

Design and Analysis of Algorithms

Using Limits to Compare Order of Growth

$$\lim_{n \rightarrow \infty} t(n)/g(n) \neq 0, \infty \Rightarrow t(n) \in \Theta(g(n))$$

$$\lim_{n \rightarrow \infty} t(n)/g(n) \neq \infty \Rightarrow t(n) \in O(g(n))$$

$$\lim_{n \rightarrow \infty} t(n)/g(n) \neq 0 \Rightarrow t(n) \in \Omega(g(n))$$

$$\lim_{n \rightarrow \infty} t(n)/g(n) = 0 \Rightarrow t(n) \in o(g(n))$$

$$\lim_{n \rightarrow \infty} t(n)/g(n) = \infty \Rightarrow t(n) \in \omega(g(n))$$

Design and Analysis of Algorithms

Using Limits to Compare Order of Growth: Example 3

Compare the order of growth of $t(n)$ and $g(n)$ using method of limits

$$t(n) = \log_2 n, g(n) = \sqrt{n}$$

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n \rightarrow \infty} \frac{(\log_2 e) \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2 \log_2 e \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0$$

$$\log_2 n \in o(\sqrt{n})$$

Design and Analysis of Algorithms

Orders of growth of some important functions

- All logarithmic functions $\log_a n$ belong to the same class

$\Theta(\log n)$ no matter what the logarithm's base $a > 1$ is

$$\log_{10} n \in \Theta(\log_2 n)$$

- All polynomials of the same degree k belong to the same class:

$$a_k n^k + a_{k-1} n^{k-1} + \dots + a_0 \in \Theta(n^k)$$

- Exponential functions can have different orders of growth for different a's

$$3^n \notin \Theta(2^n)$$

- order $\log n < \text{order } n^\alpha (\alpha > 0) < \text{order } a^n < \text{order } n! < \text{order } n^n$

Design and Analysis of Algorithms

How to Establish Orders of Growth of an Algorithm's Basic Operation Count

Summary

- Method 1: Using limits.
 - L' Hôpital's rule
- Method 2: Using the theorem.
- Method 3: Using the definitions of O-, Ω -, and Θ -notation.

Design and Analysis of Algorithms

Time Efficiency of Non-recursive Algorithms

Steps in mathematical analysis of non-recursive algorithms:

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- Check whether the number of times the basic operation is executed depends only on the input size n . If it also depends on the type of input, investigate worst, average, and best case efficiency separately.
- Set up summation for $C(n)$ reflecting the number of times the algorithm's basic operation is executed.
- Simplify summation using standard formulas

Design and Analysis of Algorithms

Useful Summation Formulas and Rules

$$\sum_{l \leq i \leq u} 1 = 1 + 1 + \dots + 1 = u - l + 1$$

$$\sum_{1 \leq i \leq n} i = 1 + 2 + \dots + n = n(n+1)/2$$

$$\sum_{1 \leq i \leq n} i^2 = 1^2 + 2^2 + \dots + n^2 = n(n+1)(2n+1)/6$$

$$\sum_{0 \leq i \leq n} a^i = 1 + a + \dots + a^n = (a^{n+1} - 1)/(a - 1) \text{ for any } a \neq 1$$

$$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \quad \Sigma c a_i = c \Sigma a_i$$

$$\sum_{1 \leq i \leq u} a_i = \sum_{1 \leq i \leq m} a_i + \sum_{m+1 \leq i \leq u} a_i$$

$$\sum_{i=l}^u 1 = (u - l + 1)$$

Design and Analysis of Algorithms

Example 1: Finding Max Element in a list

Algorithm *MaxElement (A[0..n-1])*

//Determines the value of the largest element
in a given array

//Input: An array A[0..n-1] of real numbers

//Output: The value of the largest element in A

maxval \leftarrow A[0]

for i \leftarrow 1 to n-1 do

 if A[i] > maxval

 maxval \leftarrow A[i]

return maxval

- The basic operation- comparison
- Number of comparisons is the same for all arrays of size n.
- Number of comparisons

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Design and Analysis of Algorithms

Example 2: Element Uniqueness Problem

```
Algorithm UniqueElements (A[0..n-1])
//Checks whether all the elements in a given
array are distinct
//Input: An array A[0..n-1]
//Output: Returns true if all the elements in A
are distinct and false otherwise
for i ← 0 to n - 2 do
    for j ← i + 1 to n – 1 do
        if A[i] = A[j] return false
return true
```

Best-case:

If the two first elements of the array are the same
No of comparisons in Best case = 1 comparison

Worst-case:

- Arrays with no equal elements
- Arrays in which only the last two elements are the pair of equal elements

Design and Analysis of Algorithms

Example 2: Element Uniqueness Problem

$$\begin{aligned}C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\&= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2\end{aligned}$$

Best-case: 1 comparison

Worst-case: $n^2/2$ comparisons

$T(n)_{\text{worst case}} = O(n^2)$

Design and Analysis of Algorithms

Example 3:Matrix Multiplication

```
Algorithm MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1] )  
//Multiplies two square matrices of order n by the definition-based algorithm  
//Input: two n-by-n matrices A and B  
//Output: Matrix C = AB  
for i  $\leftarrow$  0 to n - 1 do  
    for j  $\leftarrow$  0 to n - 1 do  
        C[i, j]  $\leftarrow$  0.0  
        for k  $\leftarrow$  0 to n - 1 do  
            C[i, j]  $\leftarrow$  C[i, j] + A[i, k] * B[k, j]  
return C
```

$$M(n) \in \Theta(n^3)$$

Design and Analysis of Algorithms

Steps in Mathematical Analysis of Recursive Algorithms



- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- If the number of times the basic operation is executed varies with different inputs of same sizes , investigate worst, average, and best case efficiency separately
- Set up a recurrence relation and initial condition(s) for $C(n)$ -the number of times the basic operation will be executed for an input of size n
- Solve the recurrence or estimate the order of magnitude of the solution

Design and Analysis of Algorithms

Important Recurrence Types



➤ Decrease-by-one recurrences

A decrease-by-one algorithm solves a problem by exploiting a relationship between a given instance of size n and a smaller size $n - 1$.

Example: $n!$

The recurrence equation has the form

$$T(n) = T(n-1) + f(n)$$

➤ Decrease-by-a-constant-factor recurrences

A decrease-by-a-constant algorithm solves a problem by dividing its given instance of size n into several smaller instances of size n/b , solving each of them recursively, and then, if necessary, combining the solutions to the smaller instances into a solution to the given instance.

Example: binary search.

The recurrence has the form

$$T(n) = aT(n/b) + f(n)$$

Design and Analysis of Algorithms

Decrease-by-one Recurrences

- One (constant) operation reduces problem size by one.

$$T(n) = T(n-1) + c \quad T(1) = d$$

Solution: $T(n) = (n-1)c + d$ linear

- A pass through input reduces problem size by one.

$$T(n) = T(n-1) + c \ n \quad T(1) = d$$

Solution: $T(n) = [n(n+1)/2 - 1] c + d$ quadratic

Design and Analysis of Algorithms

Methods to solve recurrences

- Substitution Method
 - Mathematical Induction
 - Backward substitution
- Recursion Tree Method
- Master Method (Decrease by constant factor recurrences)

Design and Analysis of Algorithms

Recursive Evaluation of $n!$

$$n! = 1 * 2 * \dots * (n-1) * n \quad \text{for } n \geq 1 \quad \text{and} \quad 0! = 1$$

Recursive definition of $n!$:

$$F(n) = F(n-1) * n \quad \text{for } n \geq 1 \quad \text{input size?}$$

$$F(0) = 1$$

ALGORITHM $F(n)$ basic operation?

//Computes $n!$ recursively

//Input: A nonnegative integer n

//Output: The value of $n!$

if $n = 0$ **return** 1

else return $F(n - 1) * n$

Best/Worst/Average Case?

$$M(n) = M(n - 1) + 1 \quad \text{for } n > 0,$$

$$M(0) = 0.$$

$$M(n-1) = M(n-2) + 1; \quad M(n-2) = M(n-3)+1$$

$$M(n) = n$$

Overall time Complexity: $\Theta(n)$

Design and Analysis of Algorithms

Counting number of binary digits in binary representation of a number

ALGORITHM *BinRec(n)*

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if n = 1 return 1
else return BinRec(⌊n/2⌋) + 1
```

input size?

basic operation?

Best/Worst/Average Case?

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$

$$A(2^0) = 0.$$

$$\begin{aligned} A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\ &= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\ &= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots \\ &\quad \dots && \\ &= A(2^{k-i}) + i && \\ &\quad \dots && \\ &= A(2^{k-k}) + k. && \end{aligned}$$

$$A(n) = \log_2 n \in \Theta(\log n).$$

Design and Analysis of Algorithms

Tower of Hanoi

Algorithm TowerOfHanoi(n , Src, Aux, Dst)

```
if ( $n = 0$ )
    return
TowerOfHanoi( $n-1$ , Src, Dst, Aux)
Move disk  $n$  from Src to Dst
TowerOfHanoi( $n-1$ , Aux, Src, Dst)
```

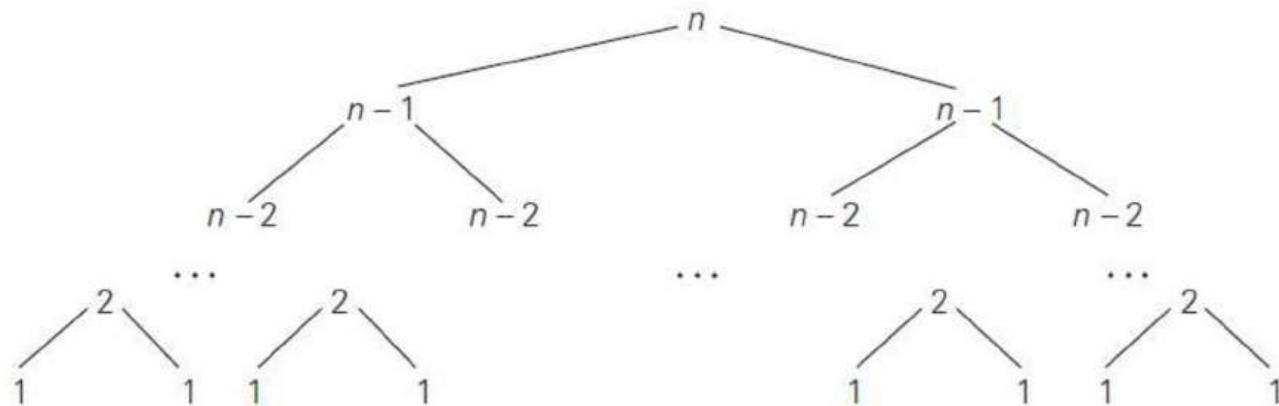
Input Size: n

Basic Operation : Move disk n from Src to Dst

$$\begin{aligned}C(n) &= 2C(n-1) + 1 \text{ for } n > 0 \text{ and } C(0)=0 \\&= 2^n - 1 \in \Theta(2^n)\end{aligned}$$

Design and Analysis of Algorithms

Tower of Hanoi : Tree of Recursive calls



$$C(n) = \sum_{l=0}^{n-1} 2^l = 2^n - 1$$

Design and Analysis of Algorithms

Solving Recurrences: Example1

$$T(n) = T(n-1) + 1 \quad n > 0 \quad T(0) = 1$$

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= T(n-2) + 1 + 1 = T(n-2) + 2 \end{aligned}$$

$$= T(n-3) + 1 + 2 = T(n-3) + 3$$

...

$$= T(n-i) + i$$

...

$$= T(n-n) + n = n = O(n)$$

Design and Analysis of Algorithms

Solving Recurrences: Example2

$$\begin{aligned} T(n) &= T(n-1) + 2n - 1 & T(0) = 0 \\ &= [T(n-2) + 2(n-1) - 1] + 2n - 1 \\ &= T(n-2) + 2(n-1) + 2n - 2 \\ &= [T(n-3) + 2(n-2) - 1] + 2(n-1) + 2n - 2 \\ &= T(n-3) + 2(n-2) + 2(n-1) + 2n - 3 \\ &\quad \dots \\ &= T(n-i) + 2(n-i+1) + \dots + 2n - i \\ &\quad \dots \\ &= T(n-n) + 2(n-n+1) + \dots + 2n - n \\ &= 0 + 2 + 4 + \dots + 2n - n \\ &= 2 + 4 + \dots + 2n - n \\ &= 2 * n * (n+1) / 2 - n \\ // \text{arithmetic progression formula } 1 + \dots + n = n(n+1)/2 // \\ &= O(n^2) \end{aligned}$$

Design and Analysis of Algorithms

Solving Recurrences: Example3

$$T(n) = T(n/2) + 1 \quad n > 1$$

$$T(1) = 1$$

$$T(n) = T(n/2) + 1$$

$$= T(n/2^2) + 1 + 1$$

$$= T(n/2^3) + 1 + 1 + 1$$

.....

$$= T(n/2^i) + i$$

.....

$$= T(n/2^k) + k \quad (k = \log n)$$

$$= 1 + \log n$$

$$= O(\log n)$$

Design and Analysis of Algorithms

Solving Recurrences: Example4

$$T(n) = 2T(n/2) + cn \quad n > 1 \quad T(1) = c$$

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/2^2) + c(n/2)) + cn = 2^2 T(n/2^2) + cn + cn \\ &= 2^2 (2T(n/2^3) + c(n/2^2)) + cn + cn = 2^3 T(n/2^3) + 3cn \\ &\dots\dots \\ &= 2^i T(n/2^i) + icn \\ &\dots\dots \\ &= 2^k T(n/2^k) + kc n \quad (k = \log n) \\ &= nT(1) + cn \log n = cn + cn \log n \\ &= O(n \log n) \end{aligned}$$

Design and Analysis of Algorithms

Performance Evaluation of Algorithm

➤ Performance Analysis

- Machine Independent
- Prior Evaluation

➤ Performance Measurement

- Machine Dependent
- Posterior Evaluation

Design and Analysis of Algorithms

Performance Analysis of Sequential search :Worst Case

ALGORITHM SequentialSearch(A[0..n-1], K)

//Searches for a given value in a given array by sequential search

//Input: An array A[0..n-1] and a search key K

//Output: Returns the index of the first element of A that matches K or -1 if there are no matching elements

i \leftarrow 0

while i < n and A[i] \neq K do

Basic operation: A[i] \neq K

 i \leftarrow i + 1

Basic operation count: n

if i < n //A[i] = K

Time Complexity: T(n) \in O(n)

 return i

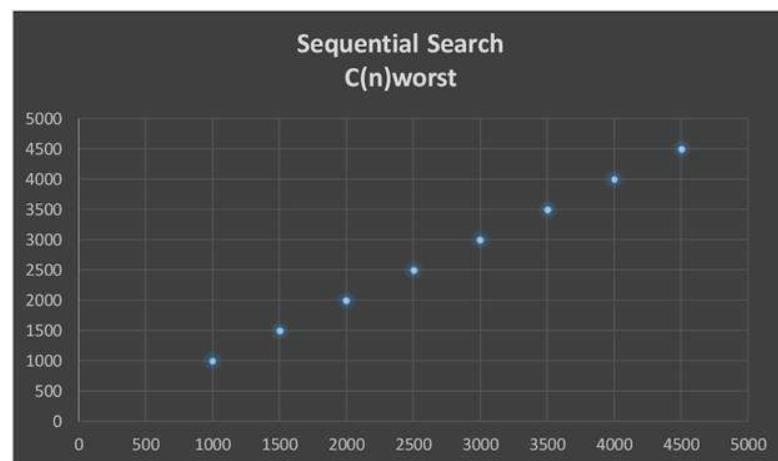
else

 return -1

Design and Analysis of Algorithms

Performance Analysis of Sequential Search

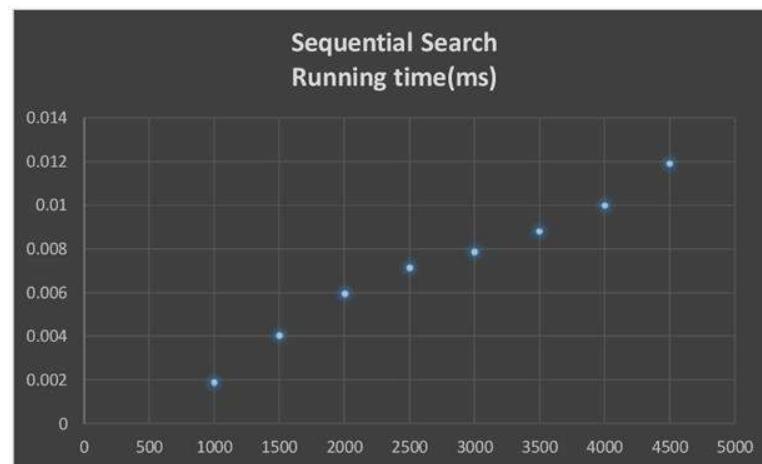
Input Size	Sequential Search $C(n)worst$
1000	1000
1500	1500
2000	2000
2500	2500
3000	3000
3500	3500
4000	4000
4500	4500



Design and Analysis of Algorithms

Performance Measurement of Sequential Search

Input Size	Sequential Search Actual Running Time(ms)
1000	0.001907
1500	0.004053
2000	0.00596
2500	0.007153
3000	0.007868
3500	0.008821
4000	0.010014
4500	0.011921



Brute Force

- In computer science, **brute-force search** or **exhaustive search**, also known as **generate and test**, is a very general problem-solving technique and algorithmic paradigm that consists of:
 - systematically enumerating all possible candidates for the solution
 - checking whether each candidate satisfies the problem's statement

Brute Force

- A brute-force algorithm to find the divisors of a natural number n would
 - enumerate all integers from 1 to n
 - check whether each of them divides n without remainder
- A brute-force approach for the eight queens puzzle would
 - examine all possible arrangements of 8 pieces on the 64-square chessboard
 - check whether each (queen) piece can attack any other, for each arrangement
- The brute-force method for finding an item in a table (linear search)
 - checks all entries of the table, sequentially, with the item

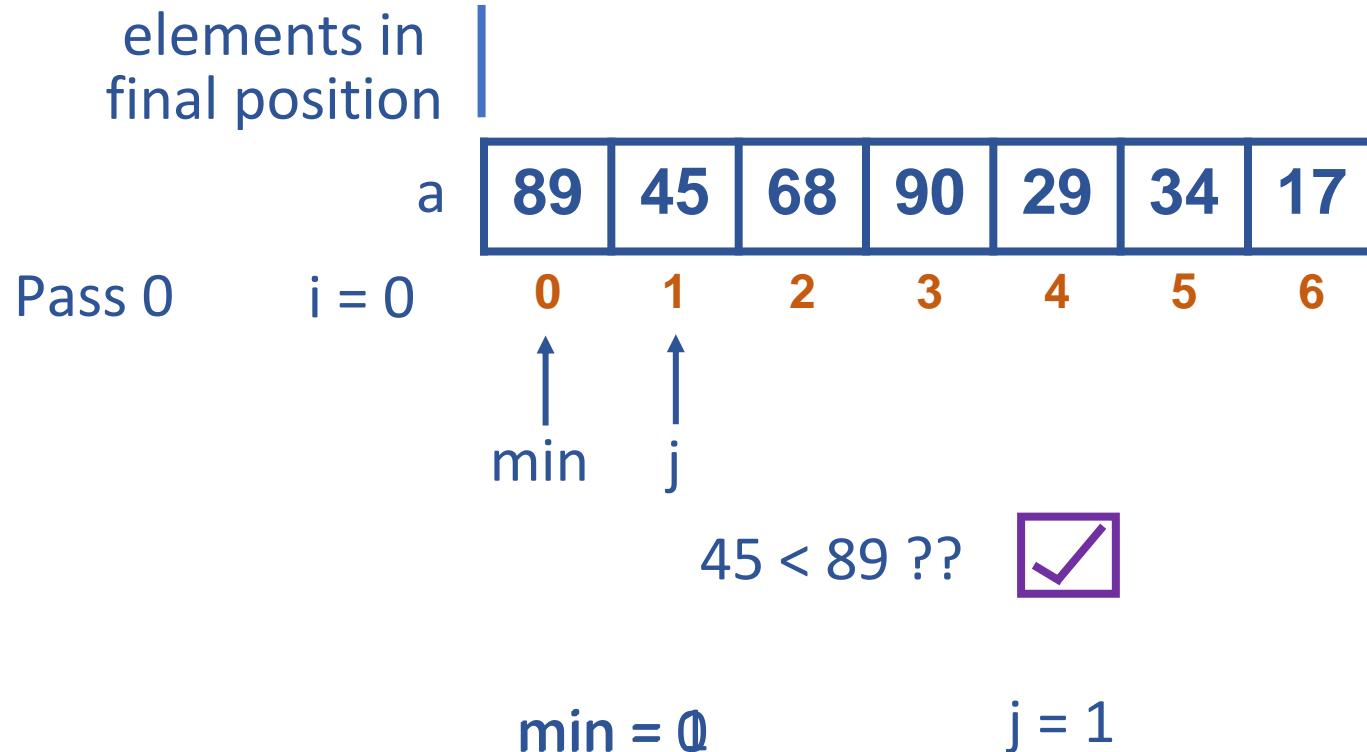
Brute Force

- A brute-force search is simple to implement, and will always find a solution if it exists
- But, its cost is proportional to the number of candidate solutions – which in many practical problems tends to grow very quickly as the size of the problem increases (Combinatorial explosion)
- Brute-force search is typically used
 - when the problem size is limited
 - when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size
 - when the simplicity of implementation is more important than speed

Selection Sort

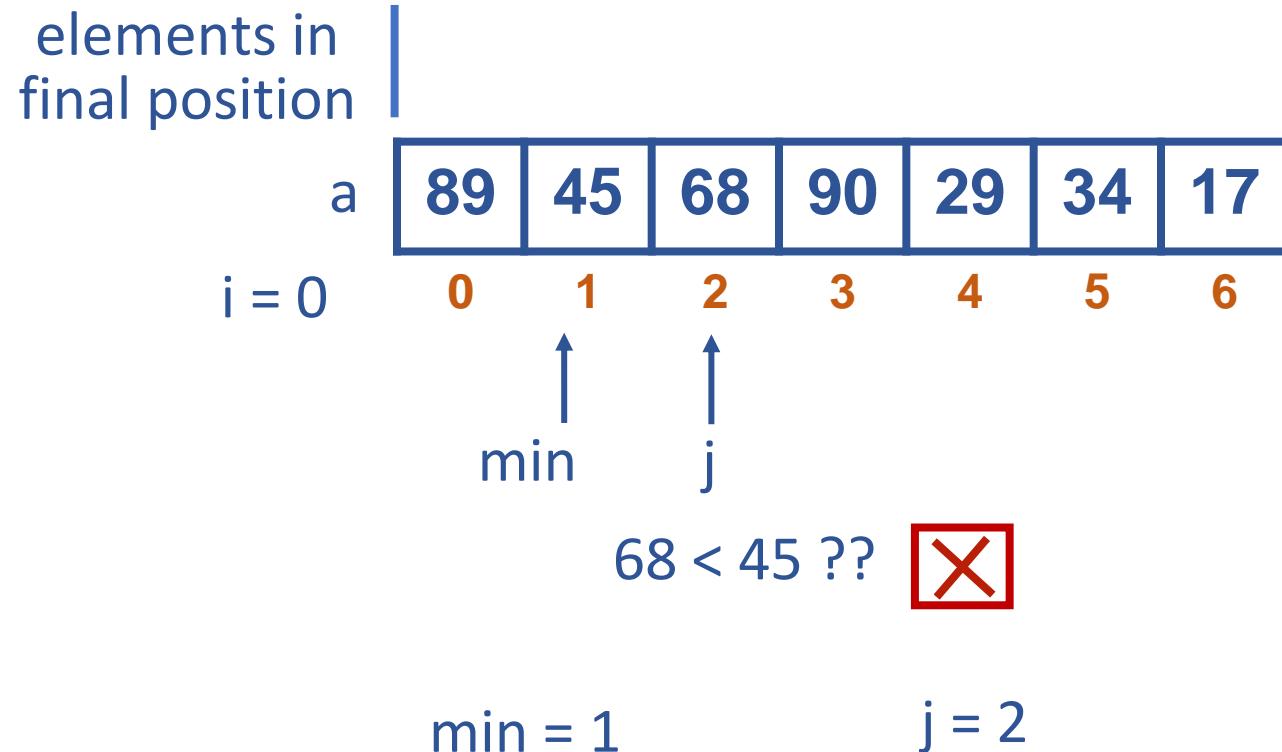
- Scan the array to find its smallest element and swap it with the first element, putting the smallest element in its final position in the sorted list
 - Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second element, putting the second smallest element in its final position
 - Generally, on pass i ($0 \leq i \leq n-2$), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$

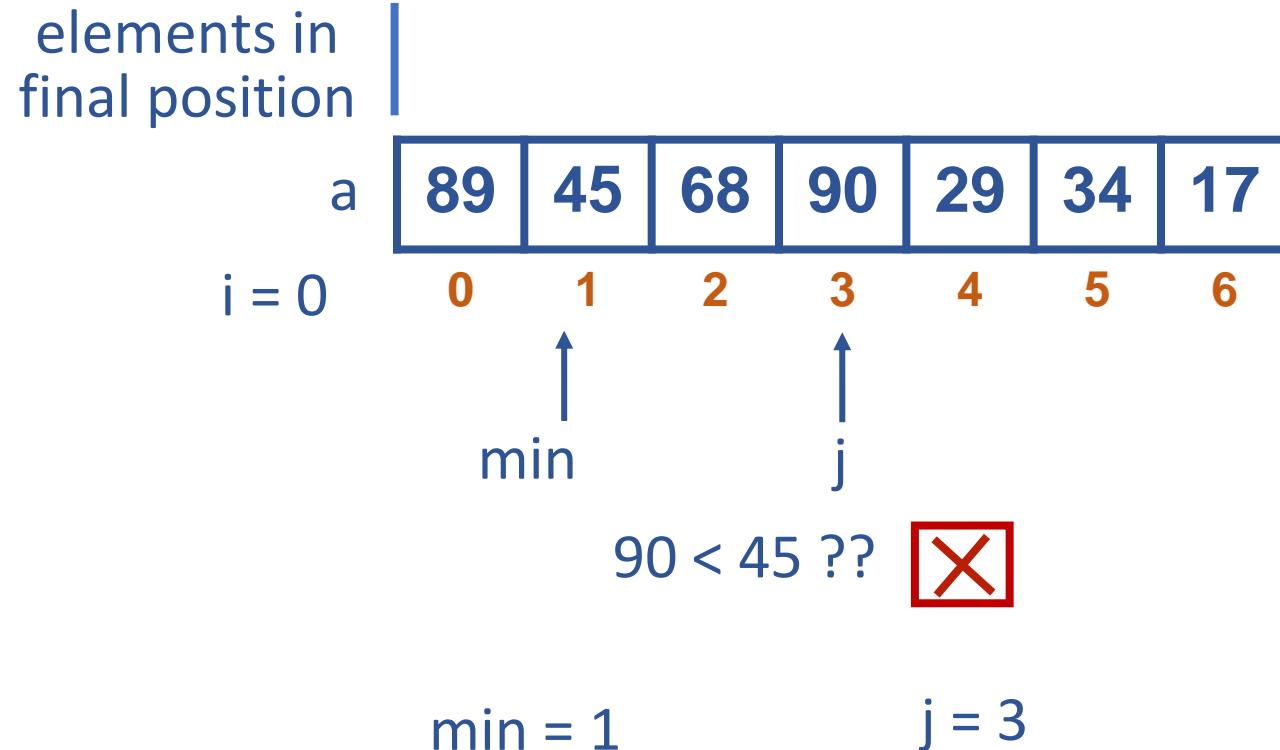


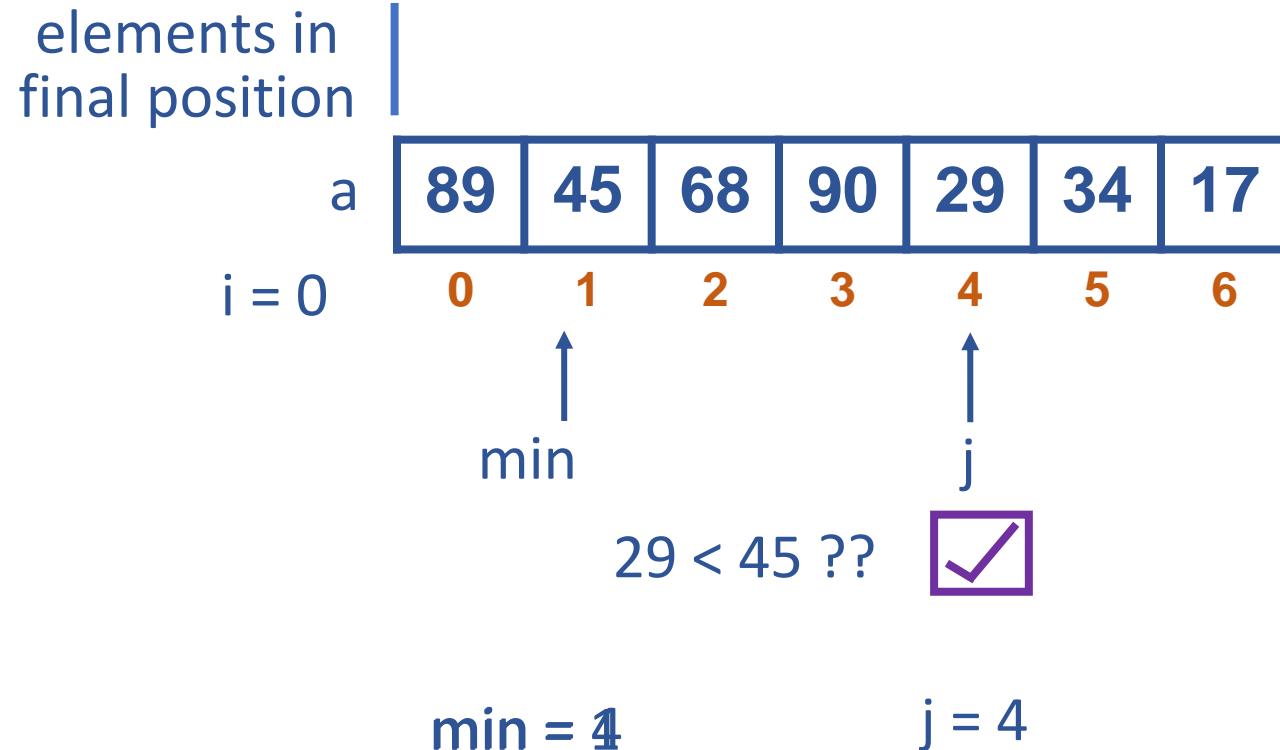


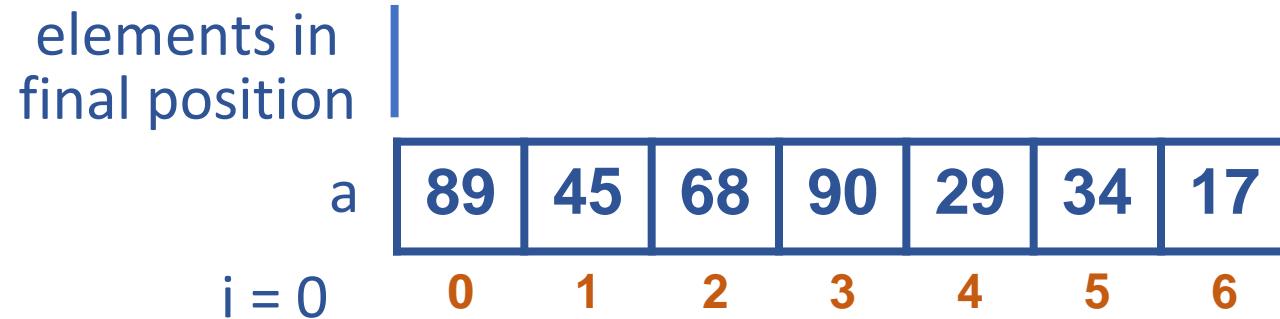
DESIGN AND ANALYSIS OF ALGORITHMS

Selection Sort









min j

$34 < 29 ??$ 

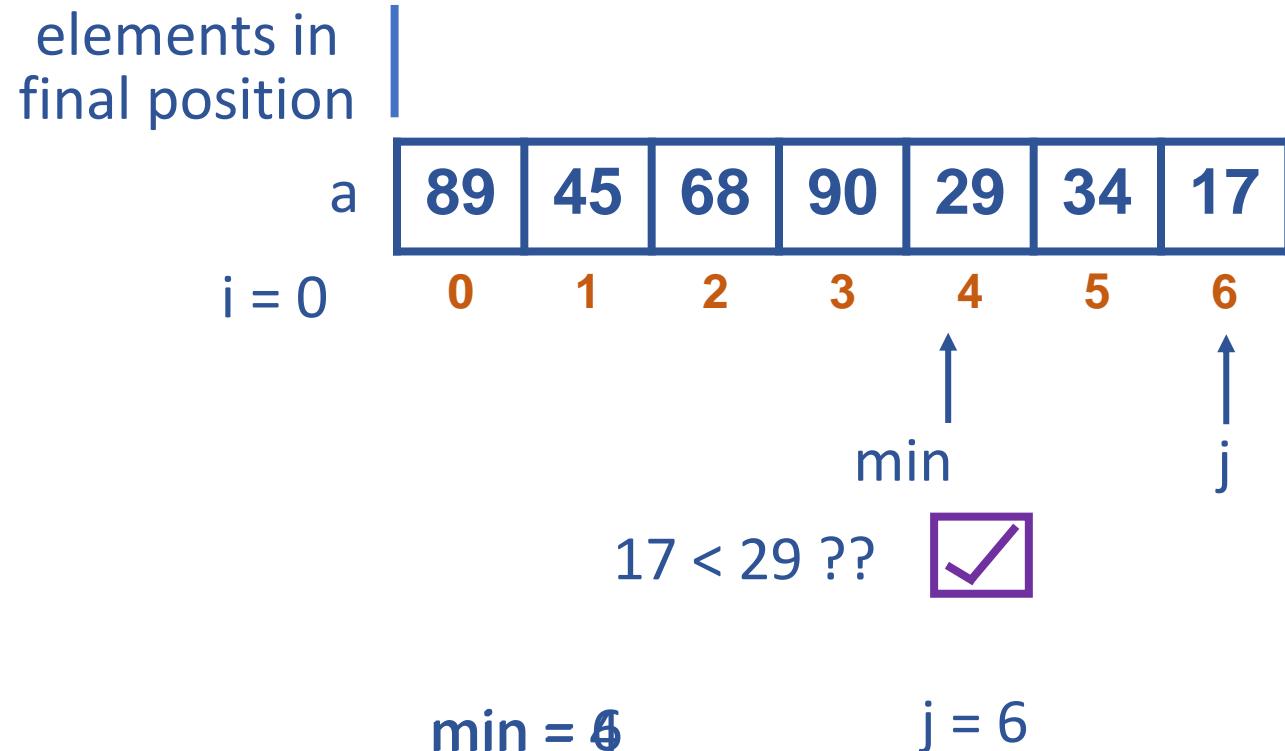
$\text{min} = 4$

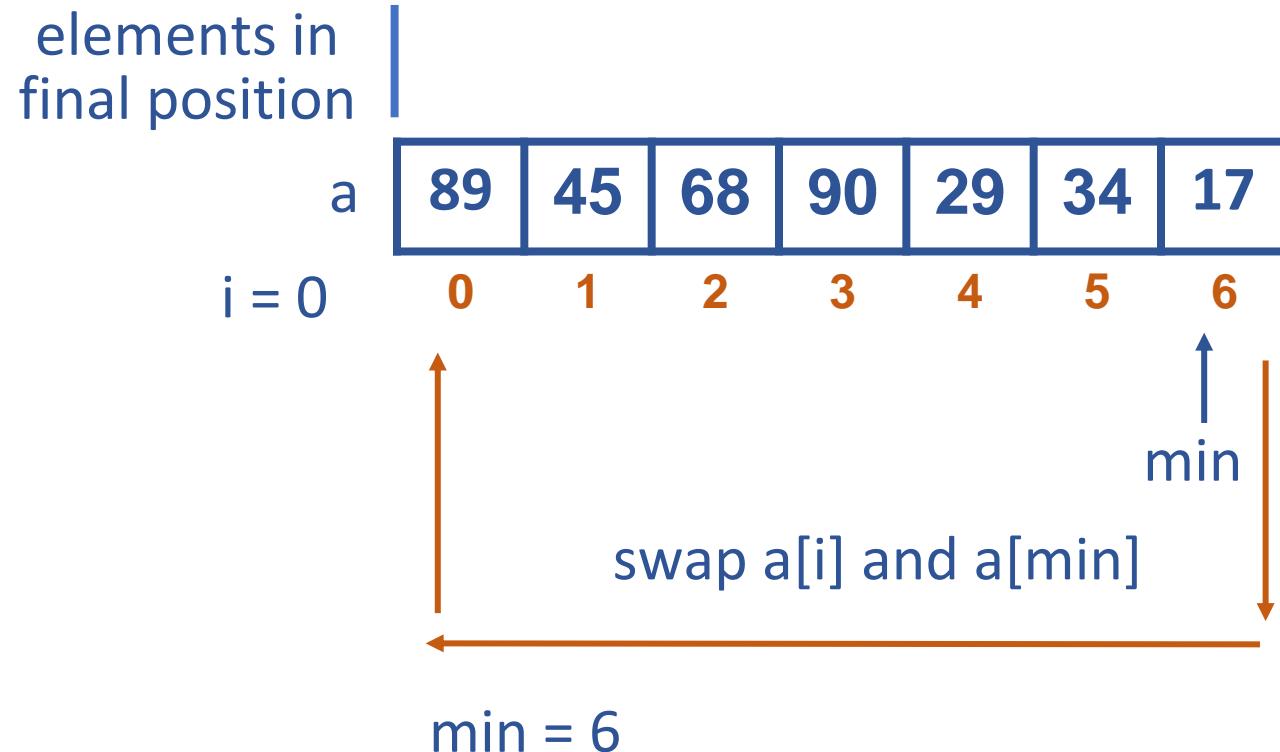
$j = 5$

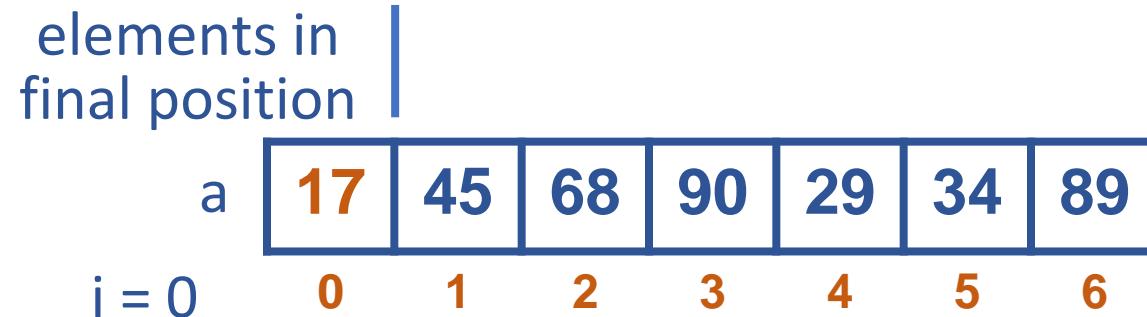
DESIGN AND ANALYSIS OF ALGORITHMS

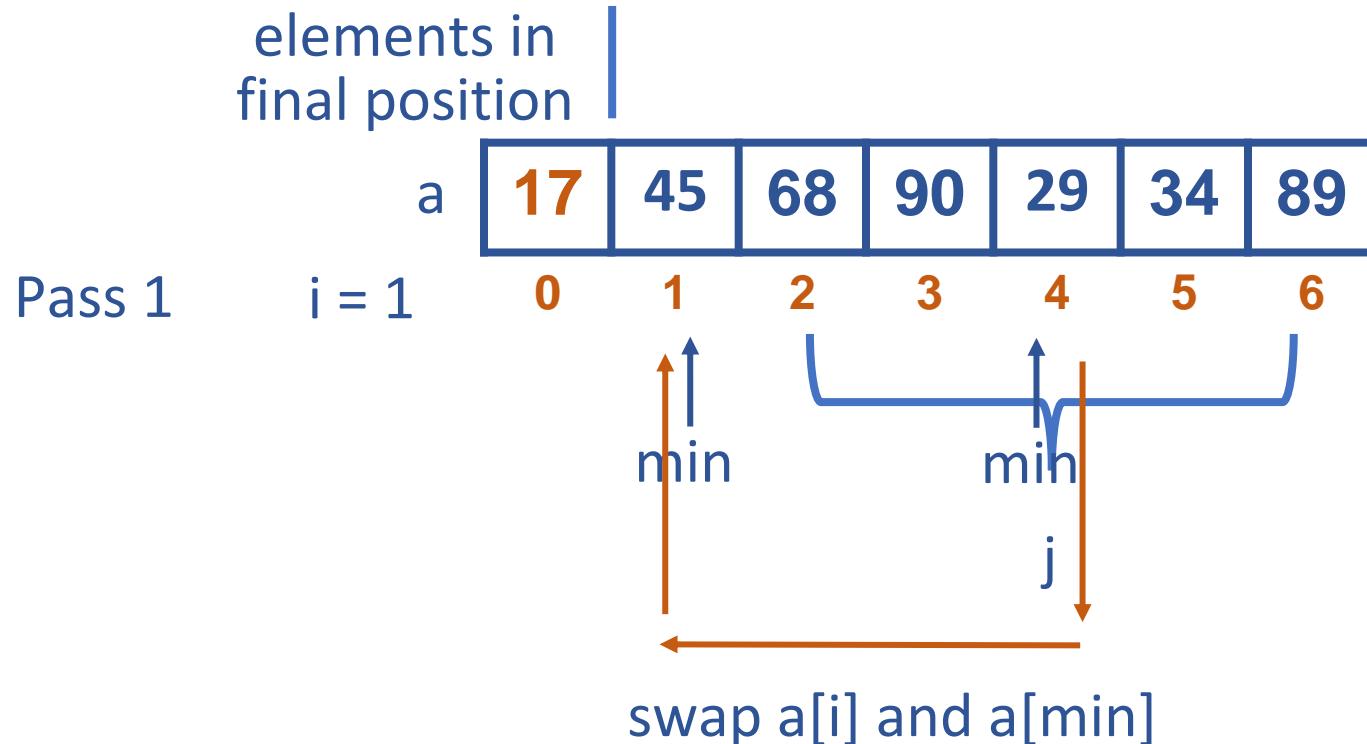


Selection Sort

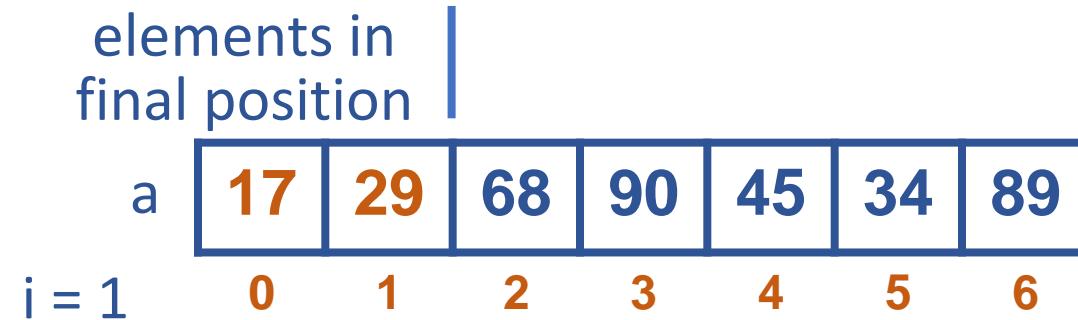


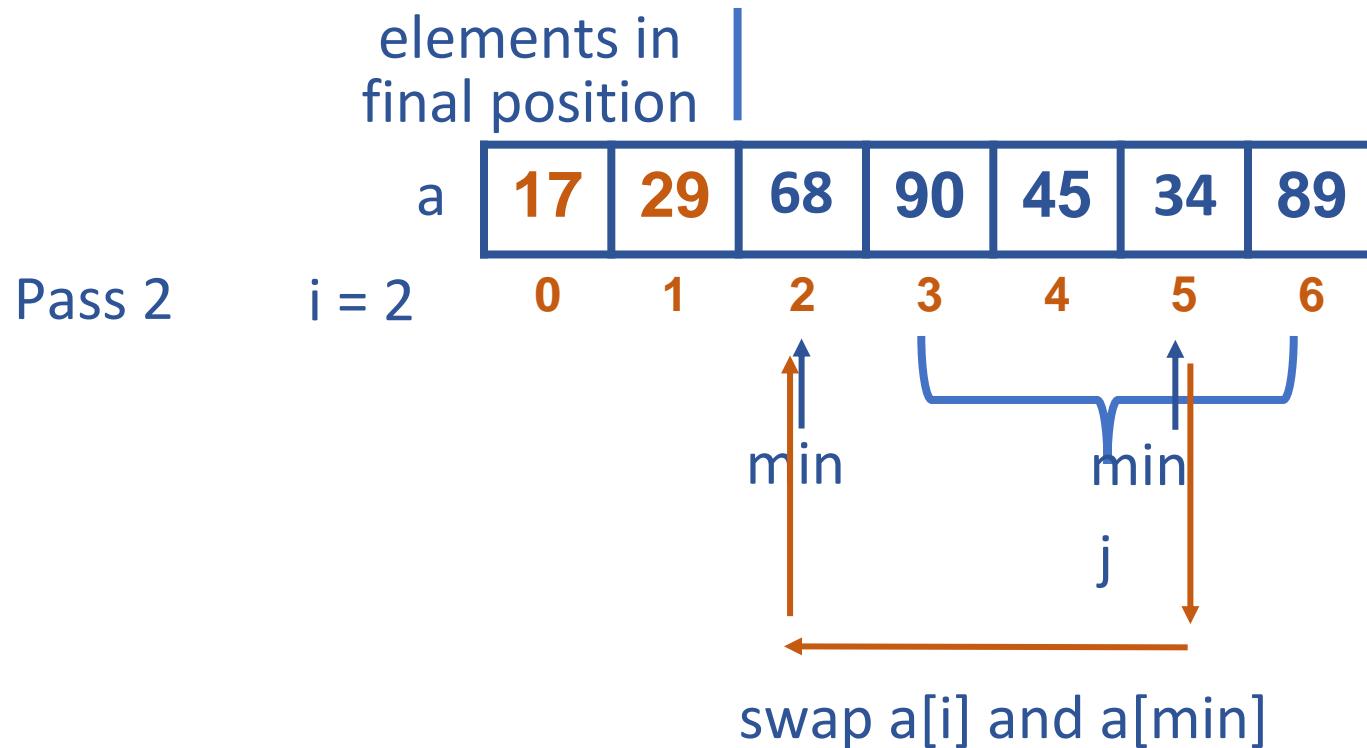




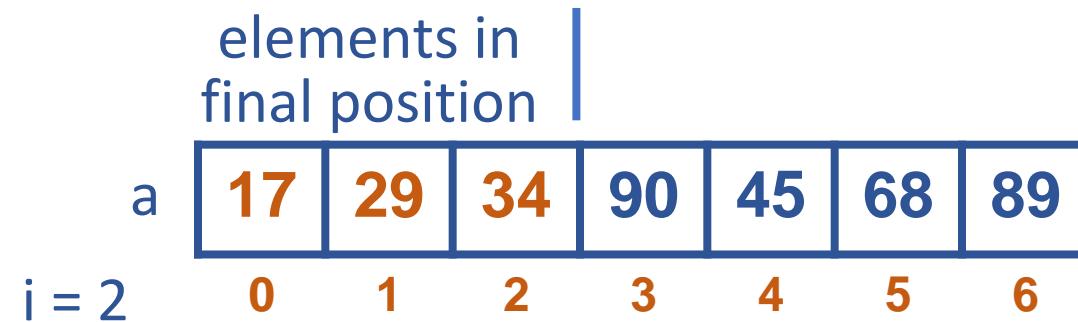


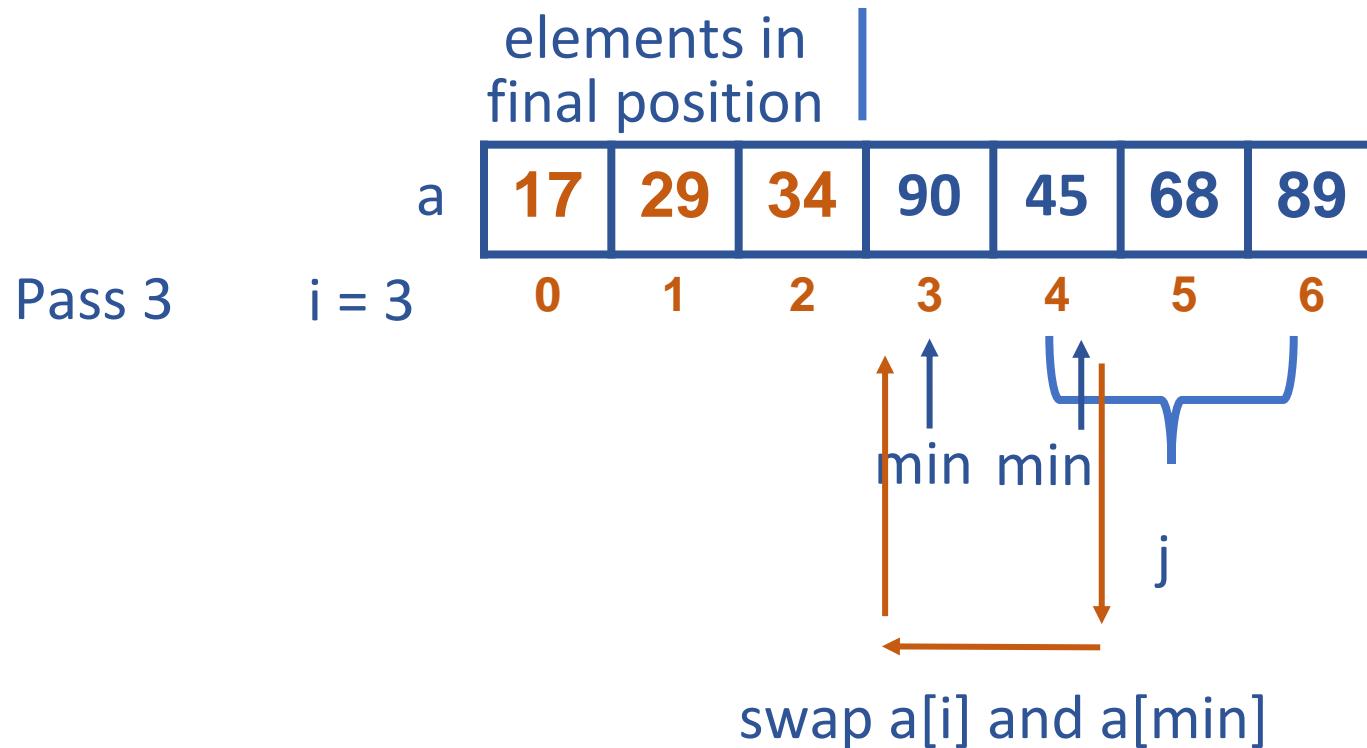
$min = 4$



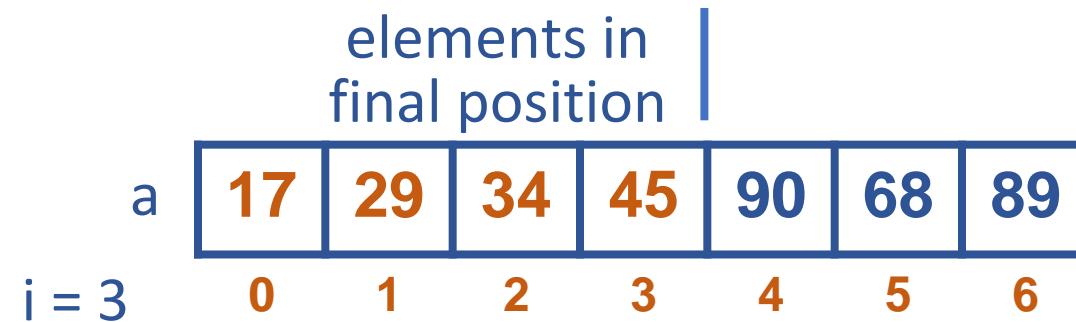


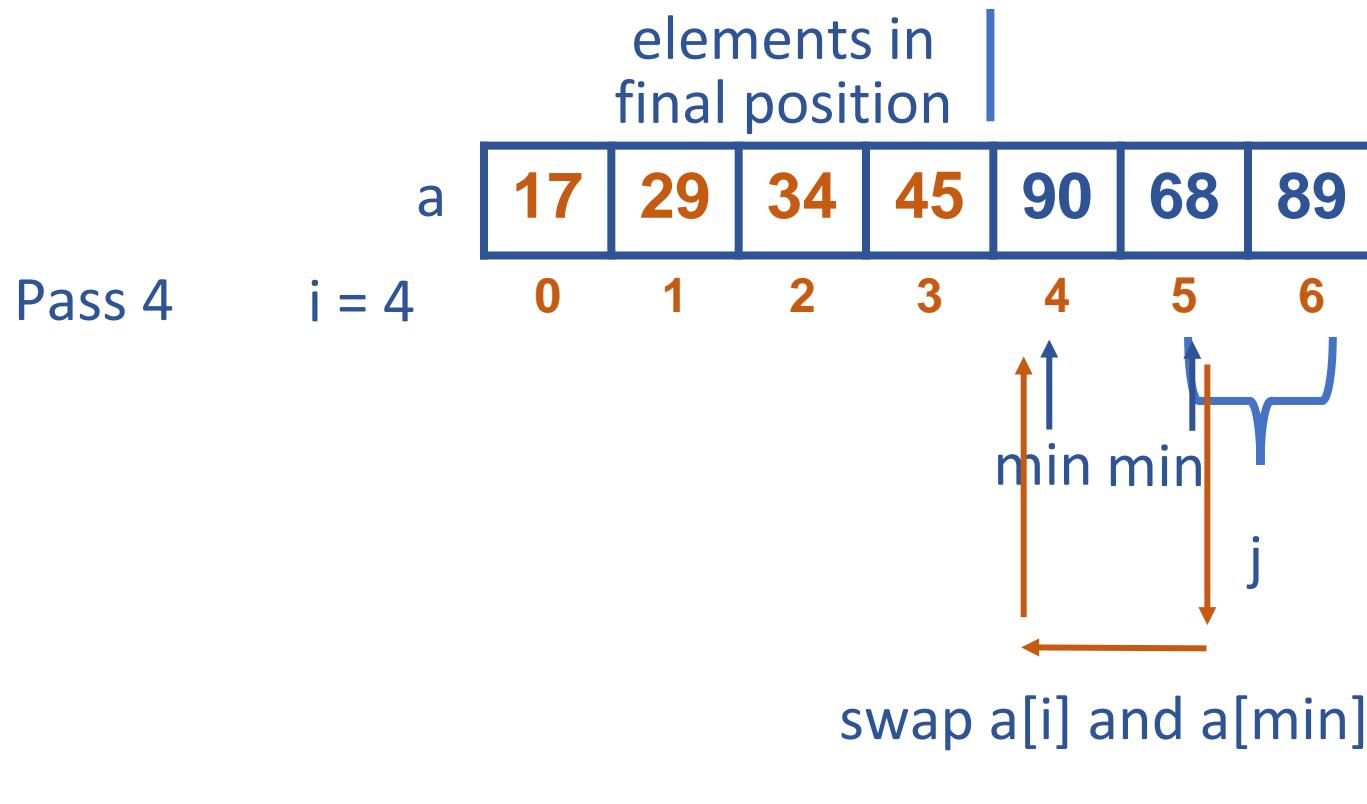
$\min = 5$

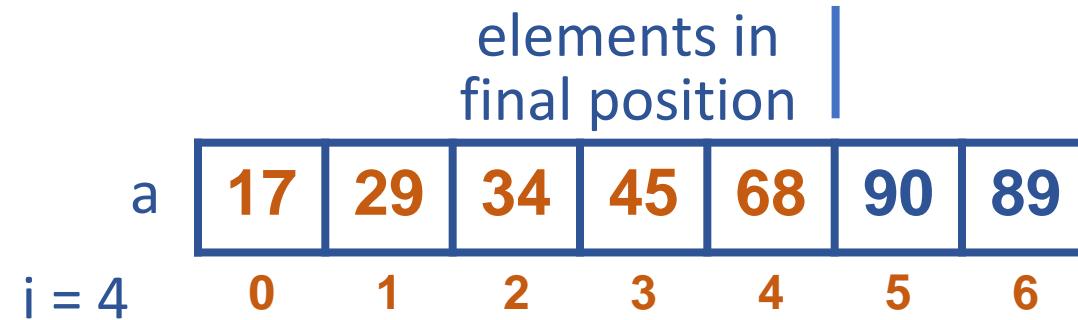


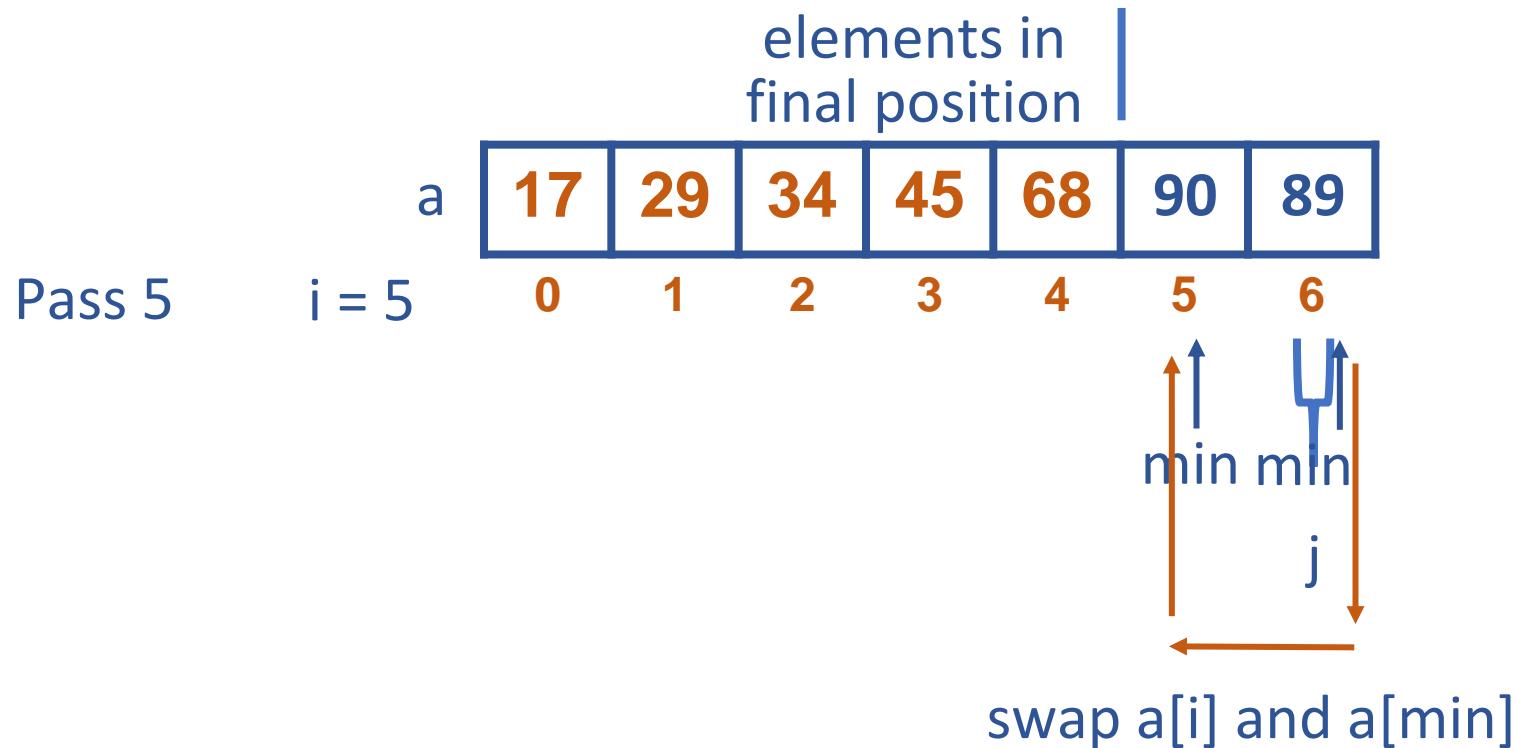


$\min = 4$

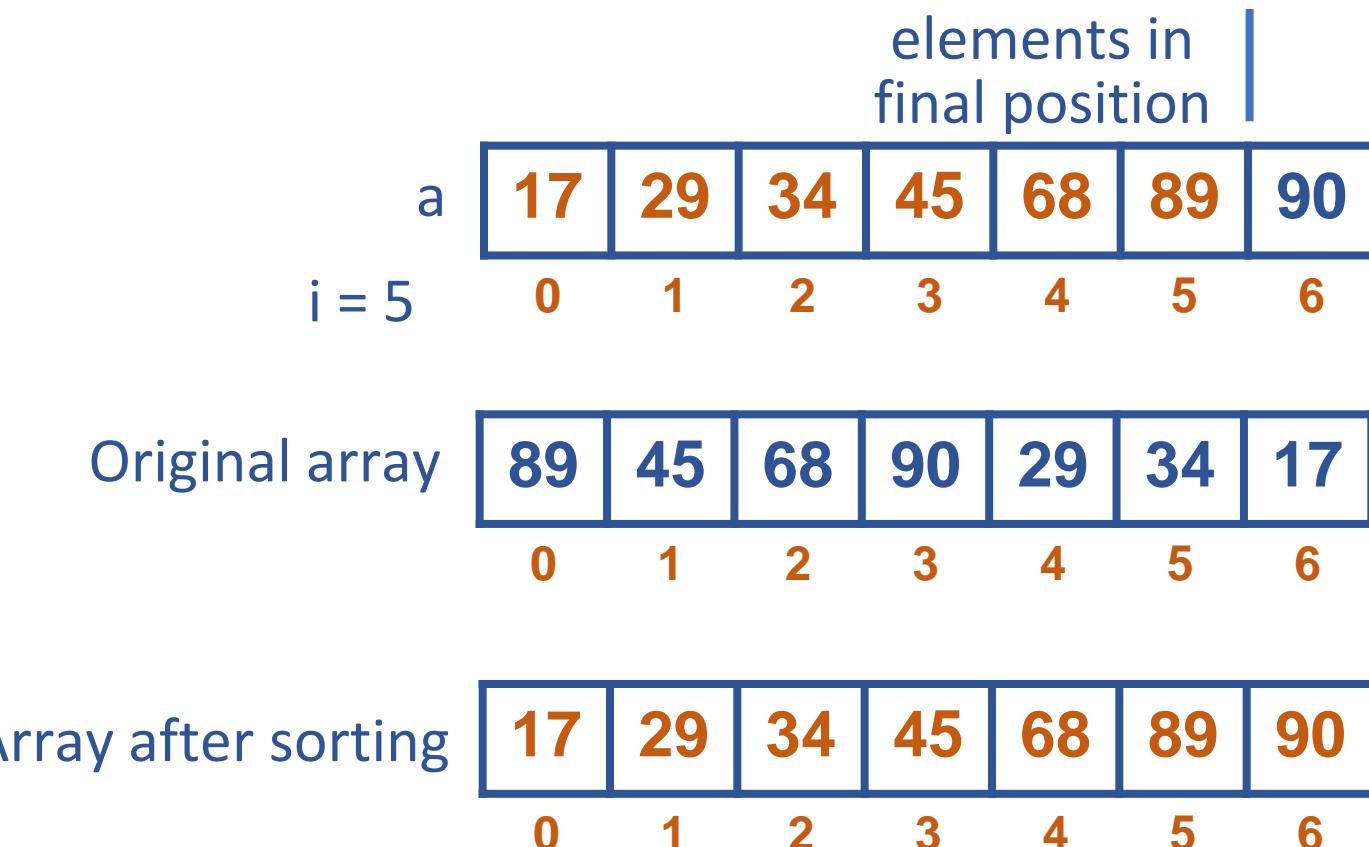








$\min = 6$



```
ALGORITHM SelectionSort(A[0 .. n -1])
//Sorts a given array by selection sort
//Input: An array A[0 .. n - 1] of orderable elements
//Output: Array A[0 .. n - 1] sorted in ascending order
for i <- 0 to n - 2 do
    min <- i
    for j <- i+1 to n-1 do
        if A[j] < A[min] min <- j
    swap A[i] and A[min]
```

Selection Sort Analysis

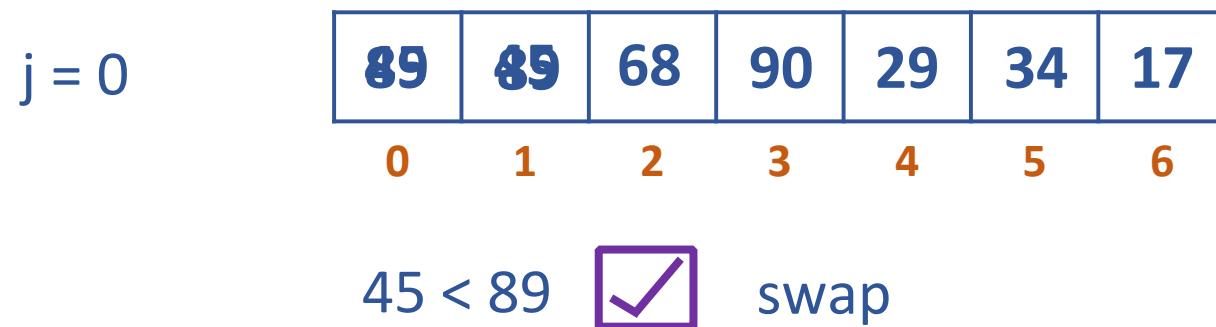
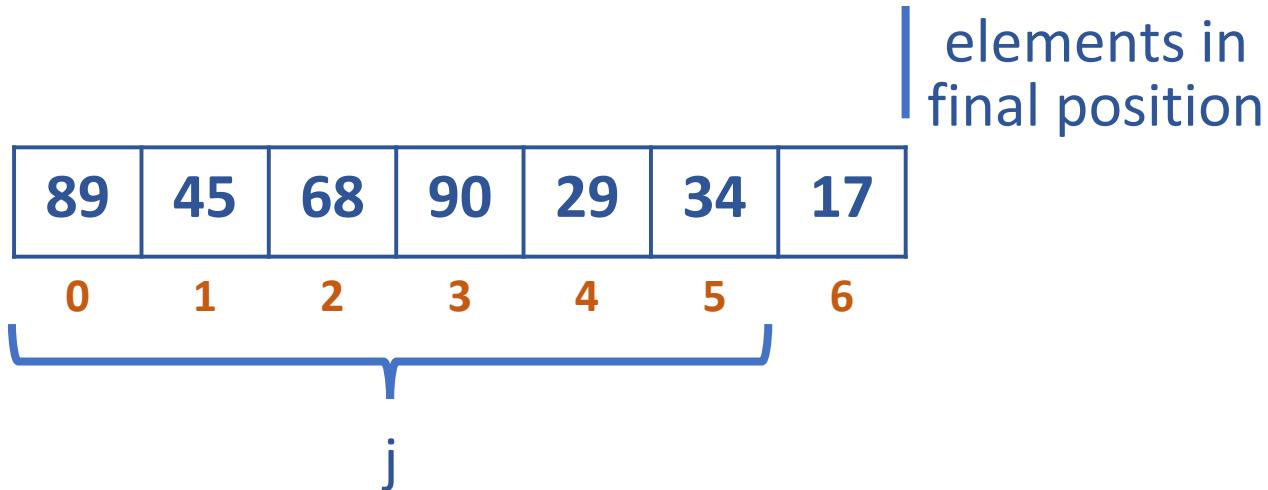
$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n - 1 - i) = \frac{(n - 1)n}{2}$$

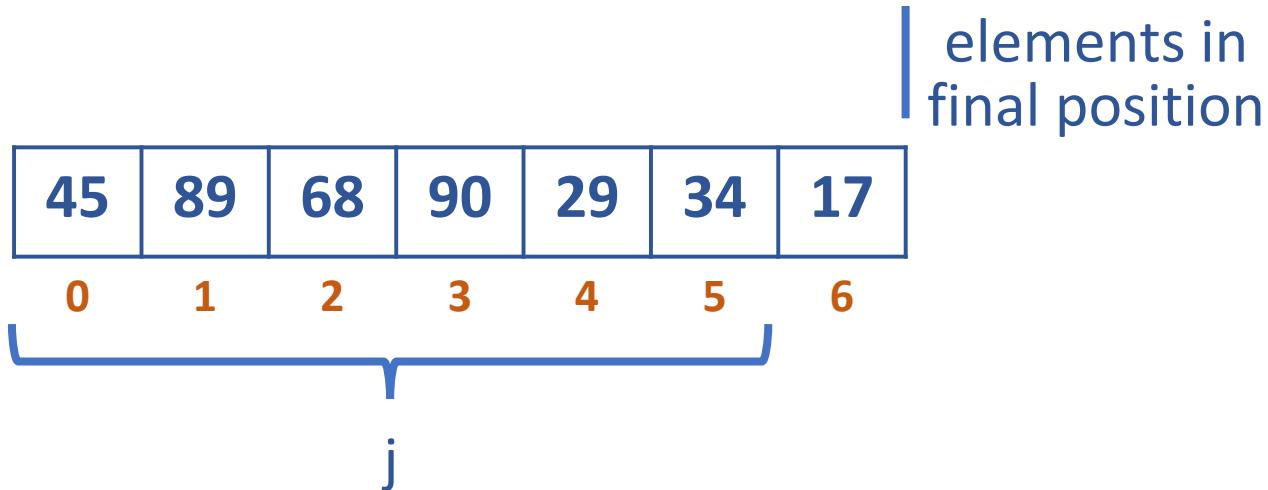
Selection Sort is a $\Theta(n^2)$ algorithm

- Compare adjacent elements of the list and exchange them if they are out of order
- By doing it repeatedly, we end up bubbling the largest element to the last position on the list
- The next pass bubbles up the second largest element and so on and after $n - 1$ passes, the list is sorted
- Pass i ($0 \leq i \leq n - 2$) can be represented as follows:

$A[0], A[1], A[2], \dots, A[j] \leftrightarrow ? A[j+1], \dots, A[n-i-1] \mid A[n-i] \leq \dots \leq A[n-1]$

in their final positions

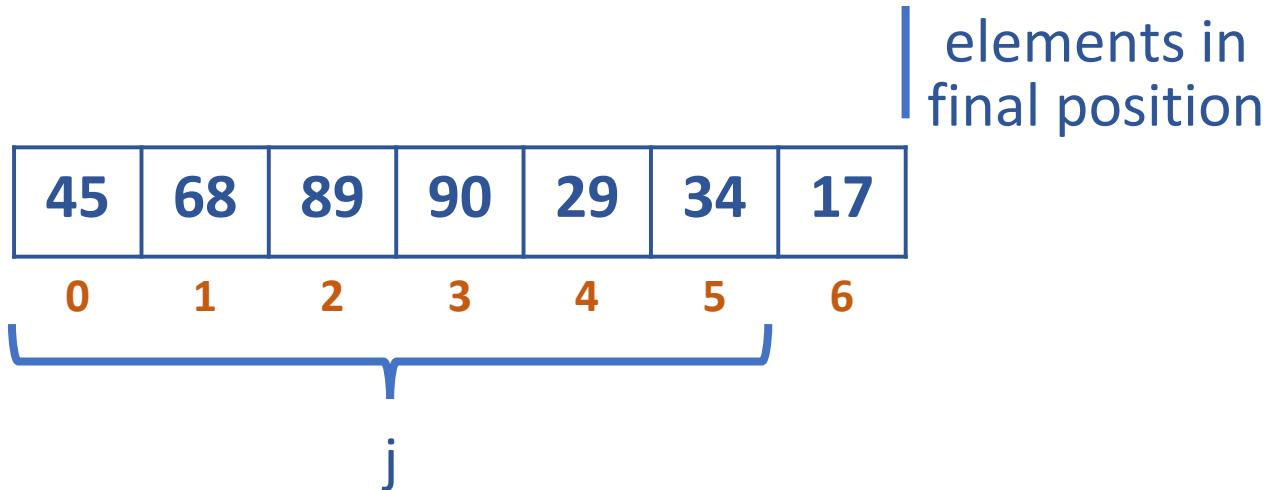




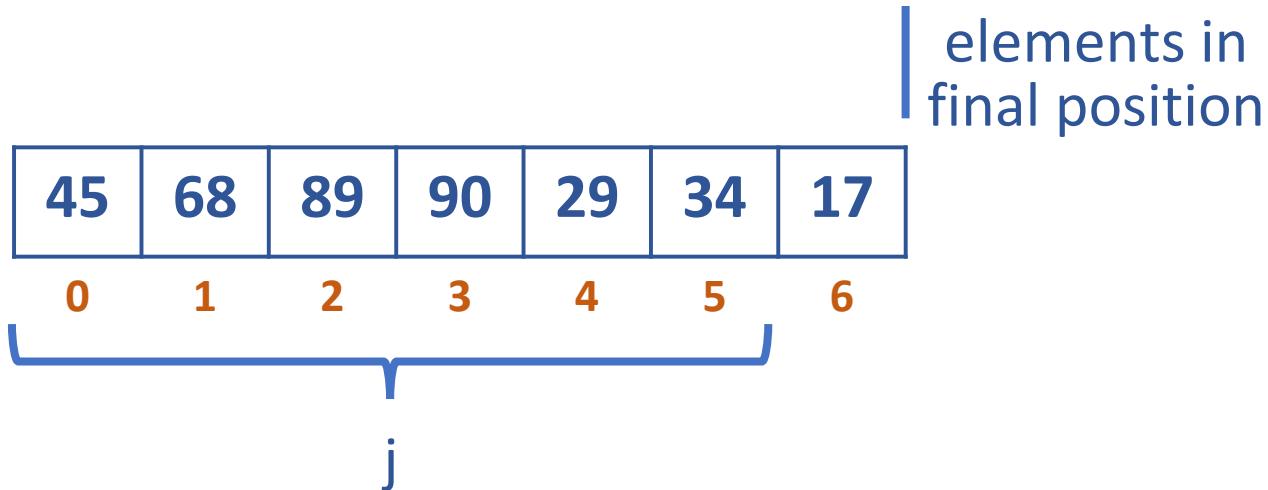
j = 1

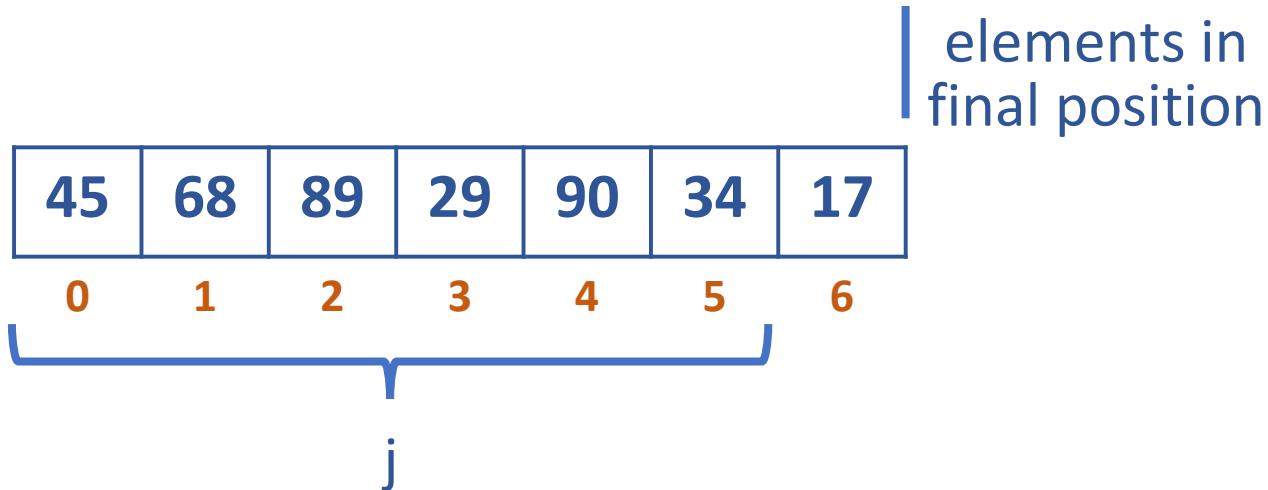
45	89	68	90	29	34	17
0	1	2	3	4	5	6

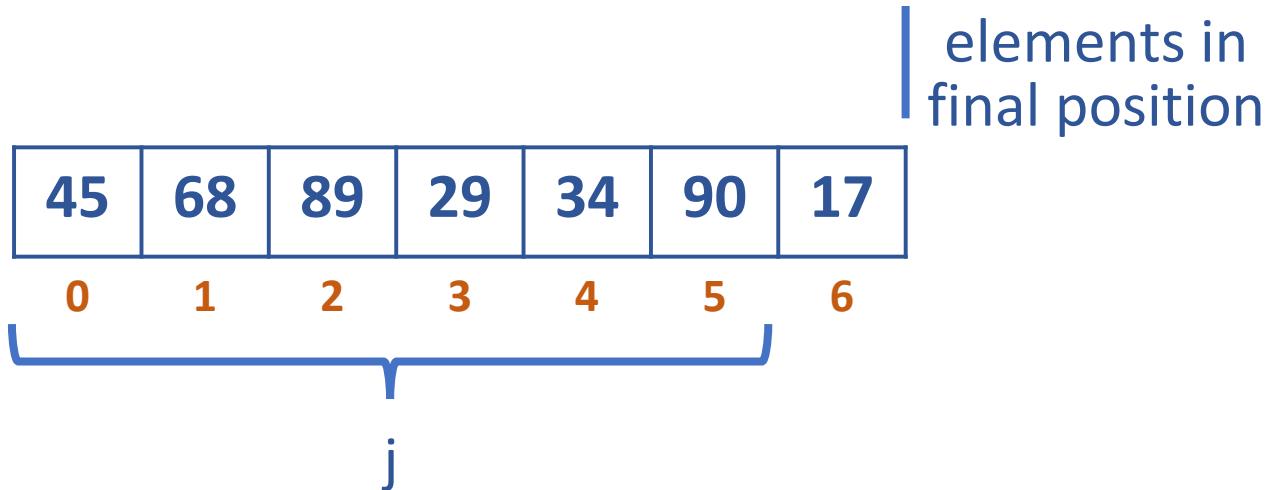
$68 < 89$  swap



90 < 89 X no swap





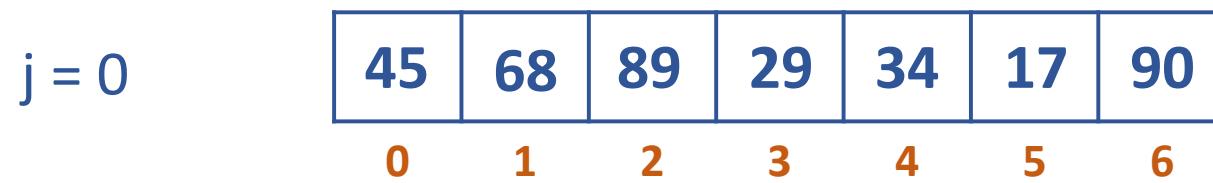
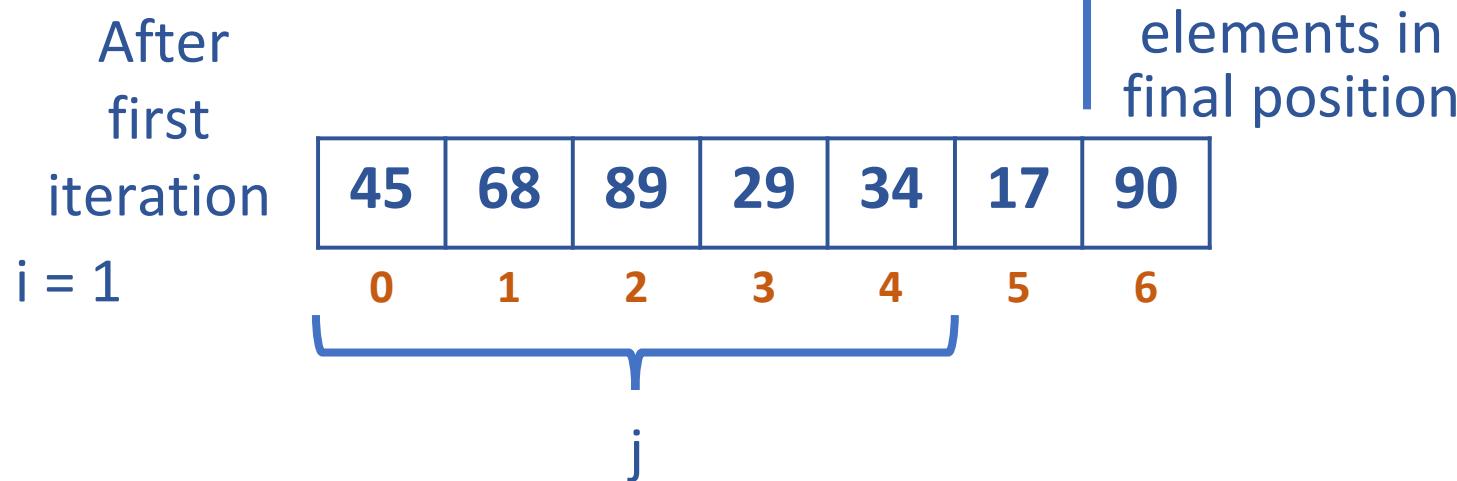


j = 5

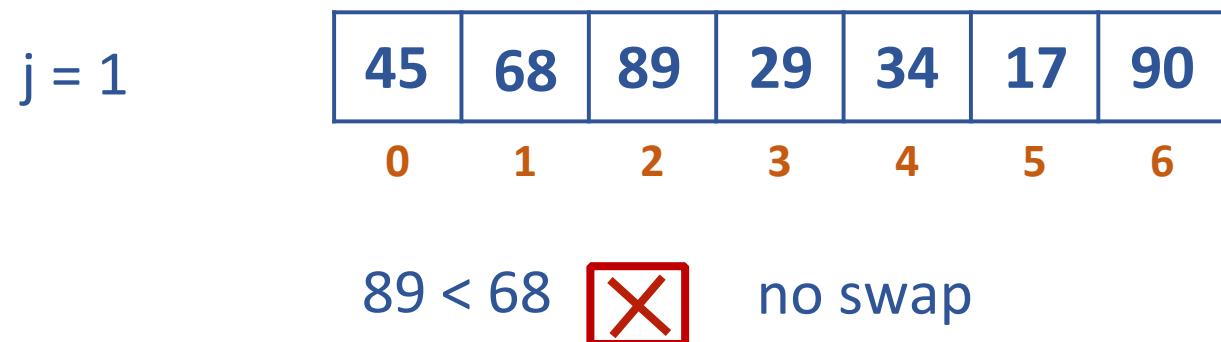
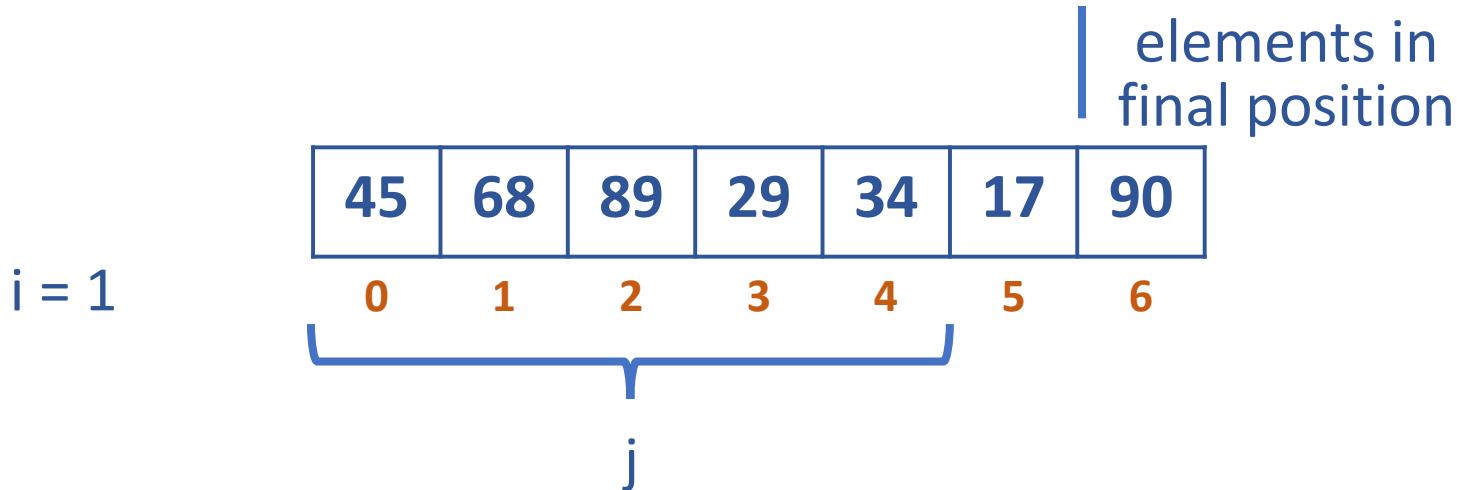
45	68	89	29	34	90	90
0	1	2	3	4	5	6

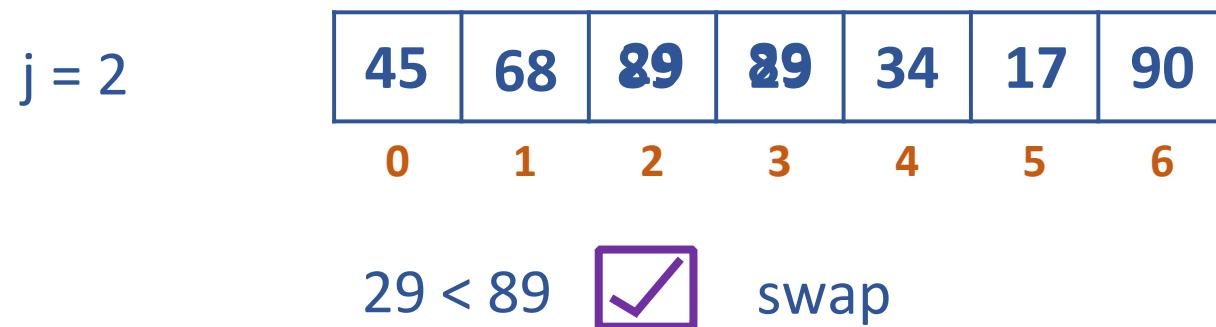
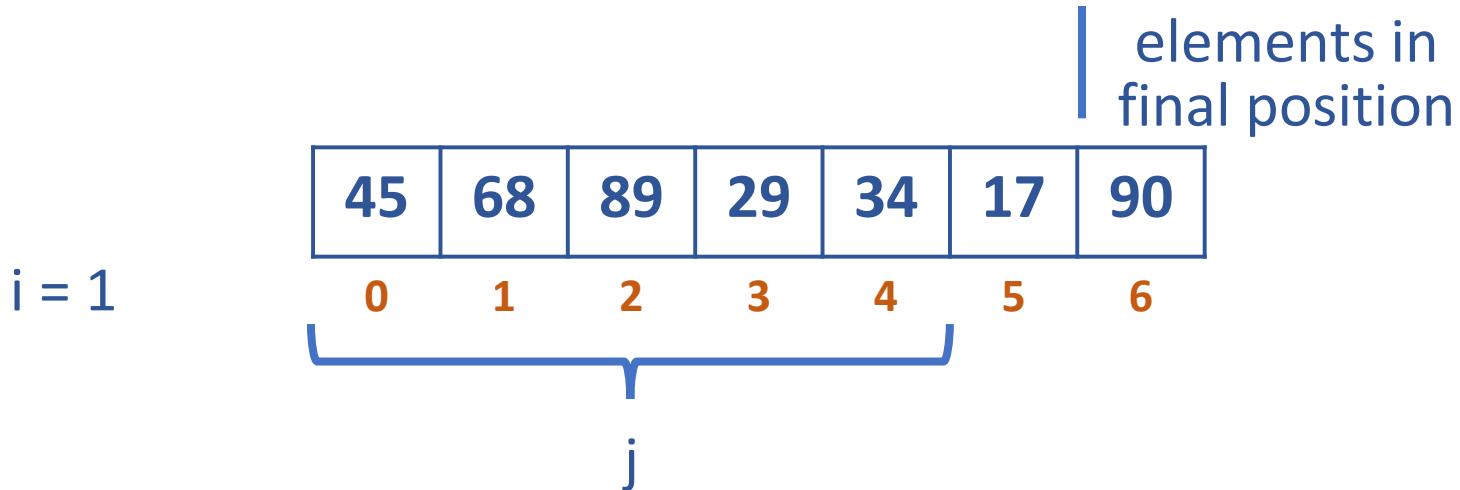
17 < 90  swap

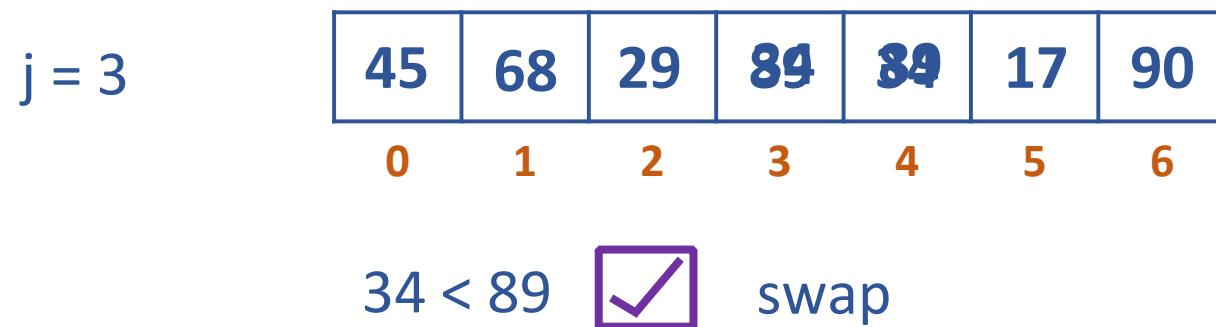
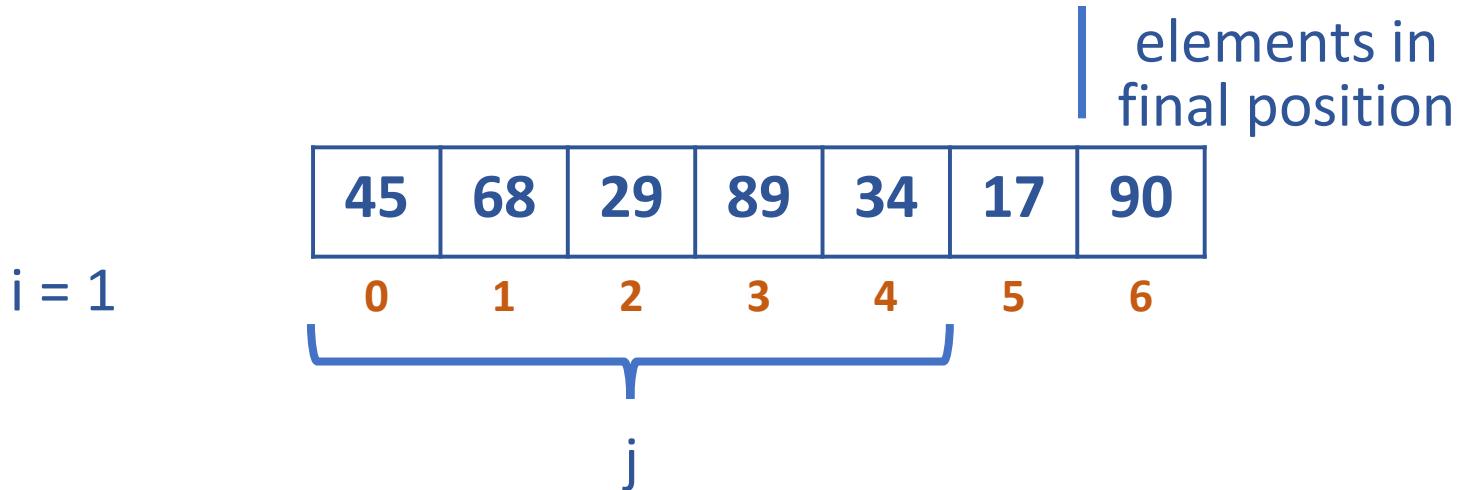
Bubble Sort

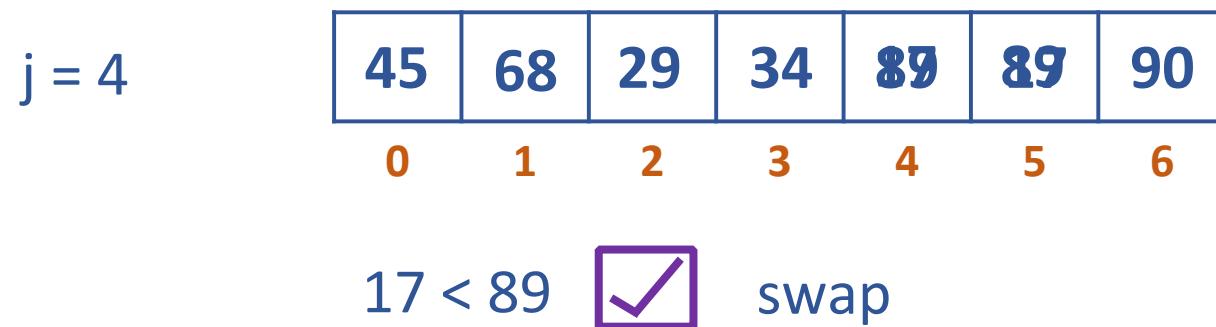
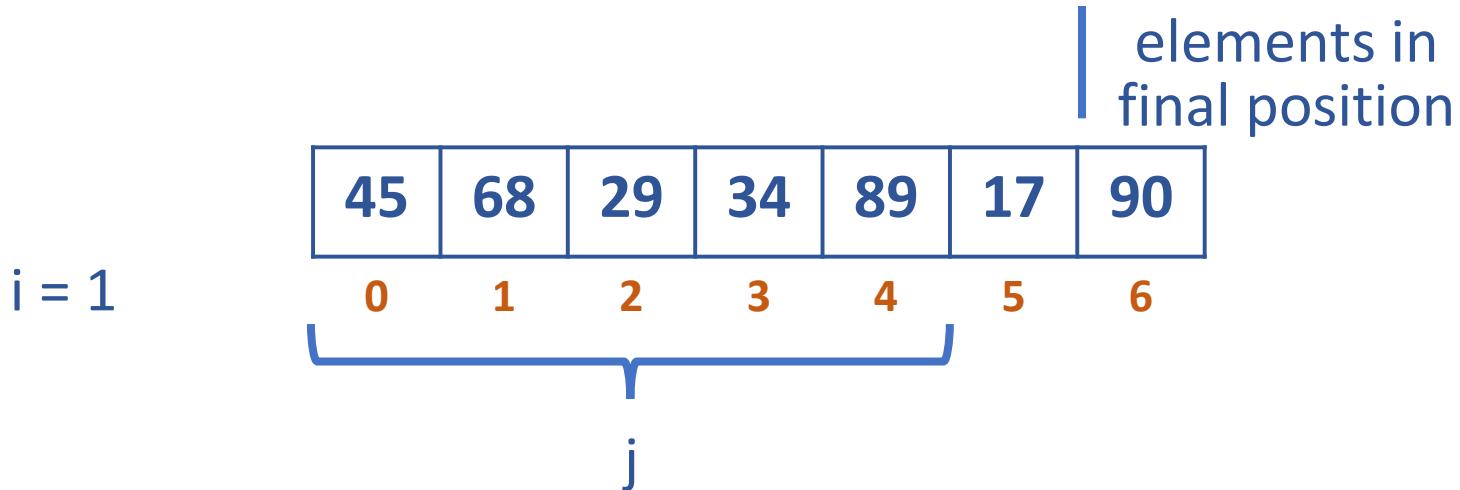


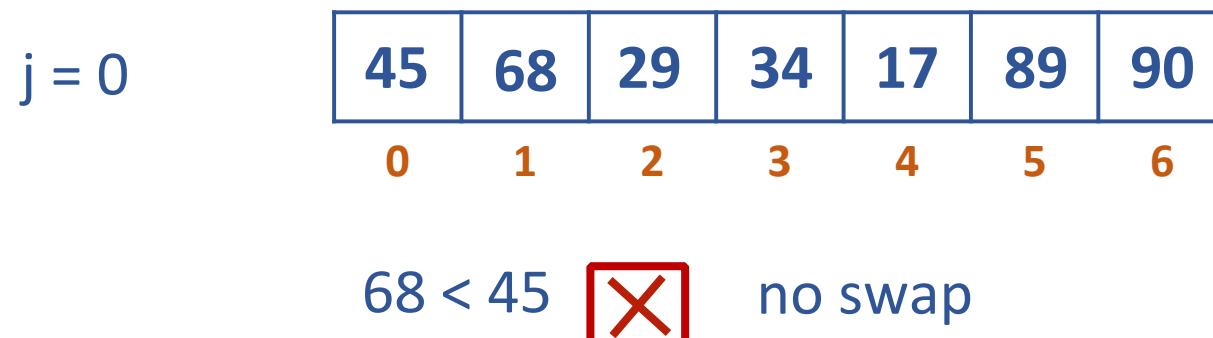
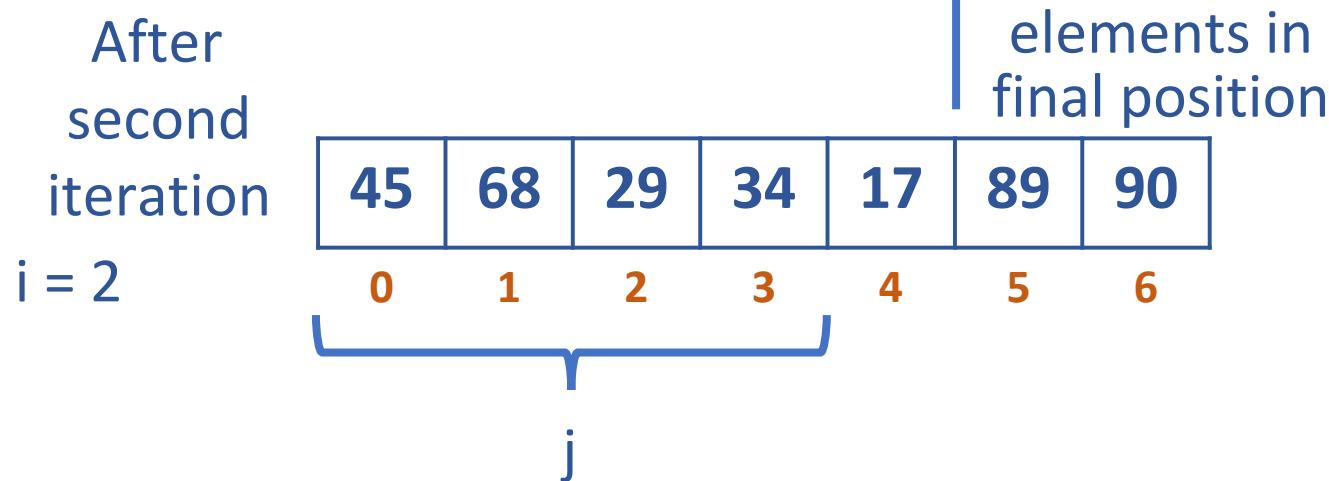
$68 < 45$  no swap

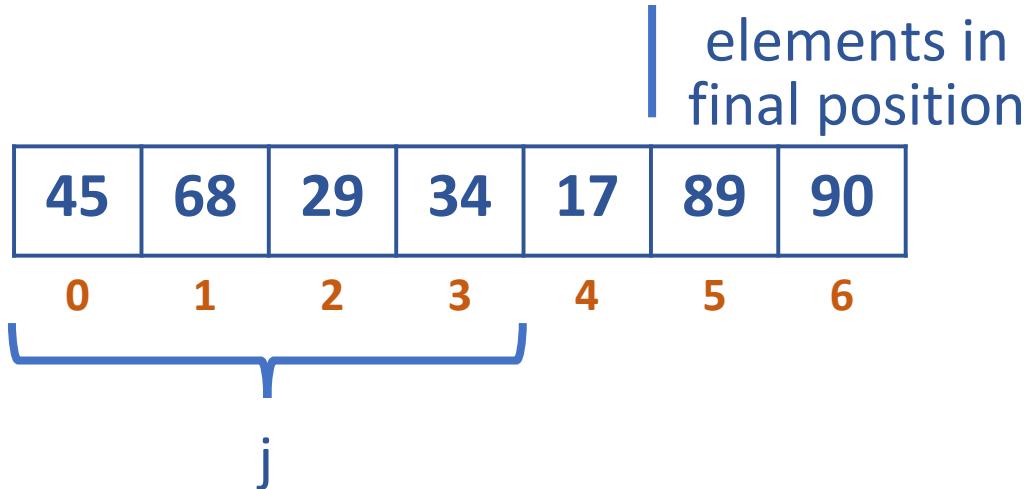








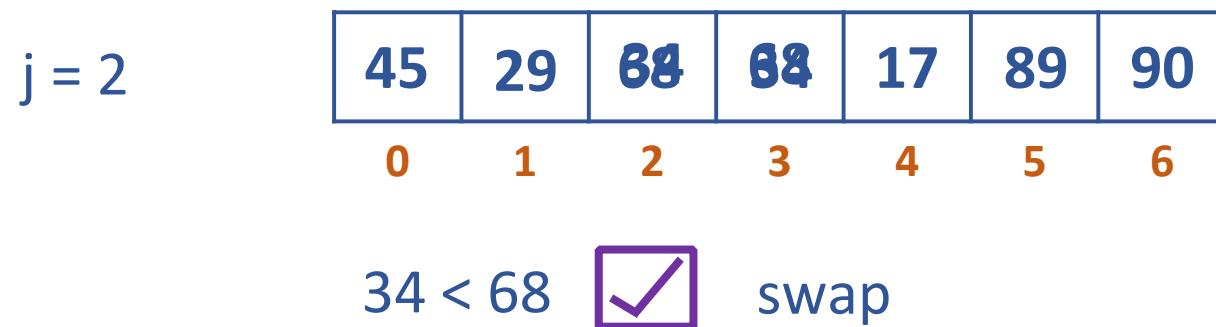
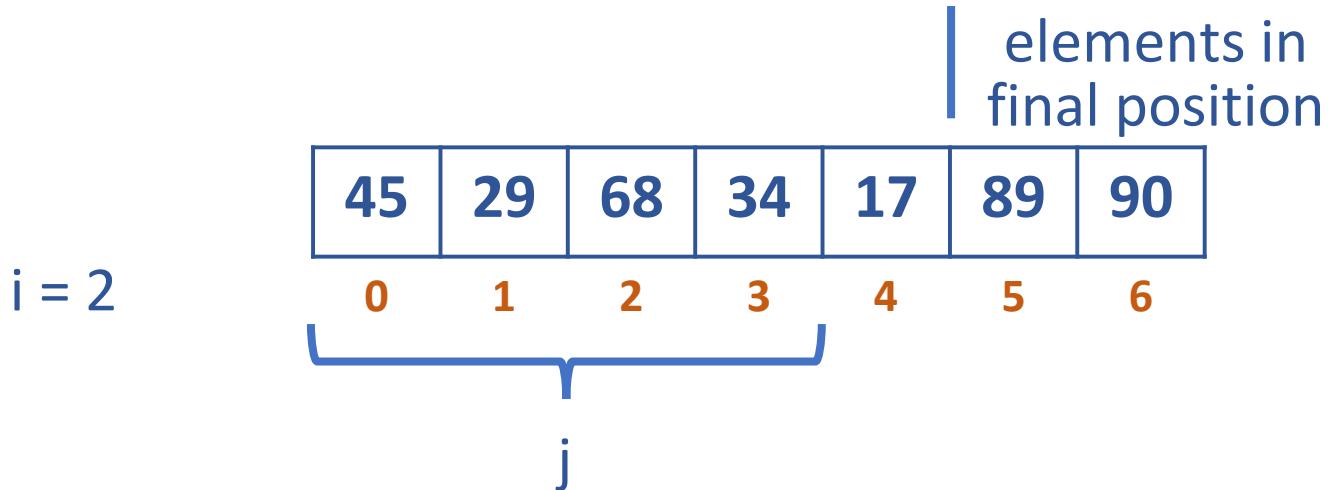


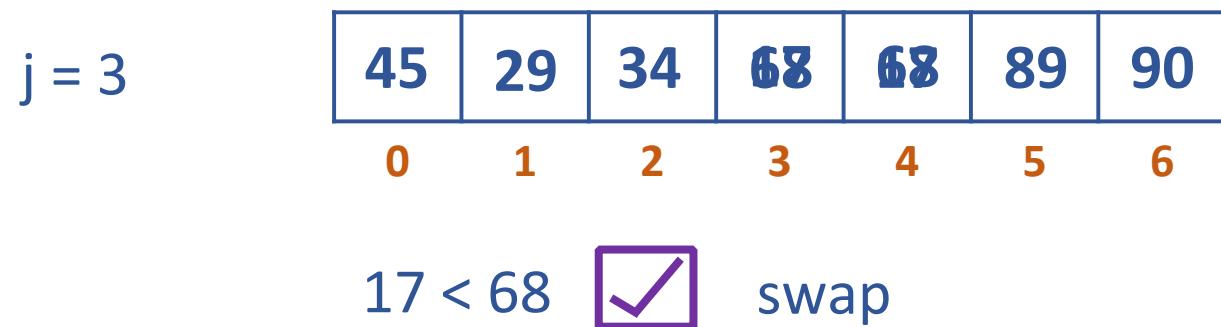
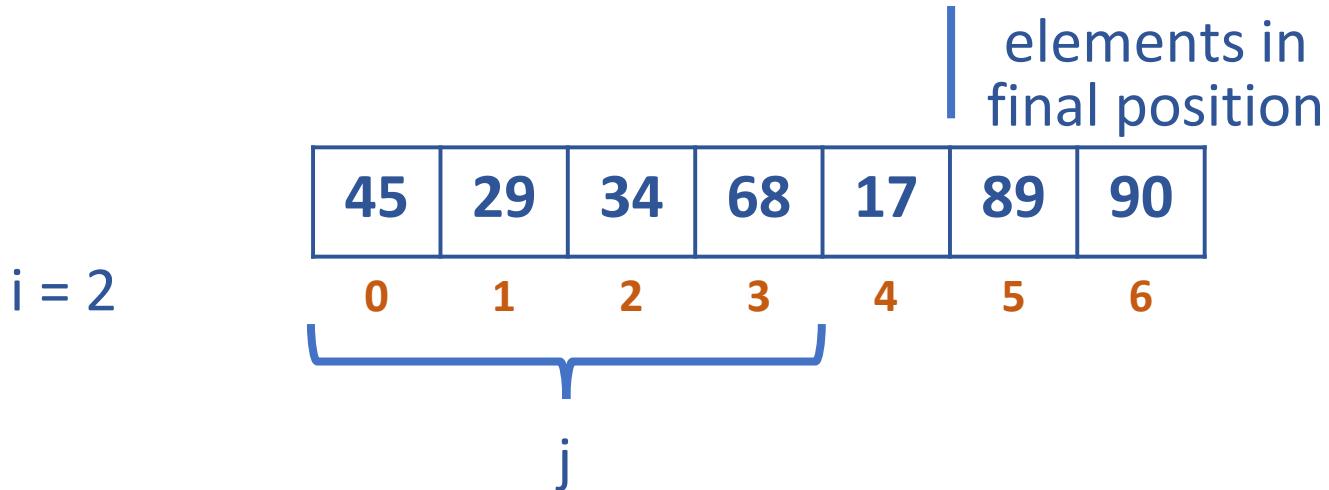


j = 1

45	68	29	34	17	89	90
0	1	2	3	4	5	6

$29 < 68$  swap



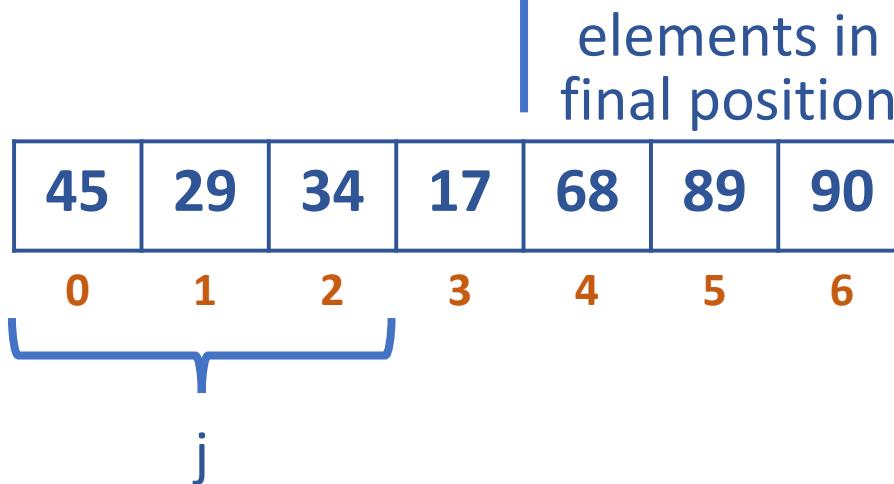


DESIGN AND ANALYSIS OF ALGORITHMS

Bubble Sort

After
third
iteration

$i = 3$



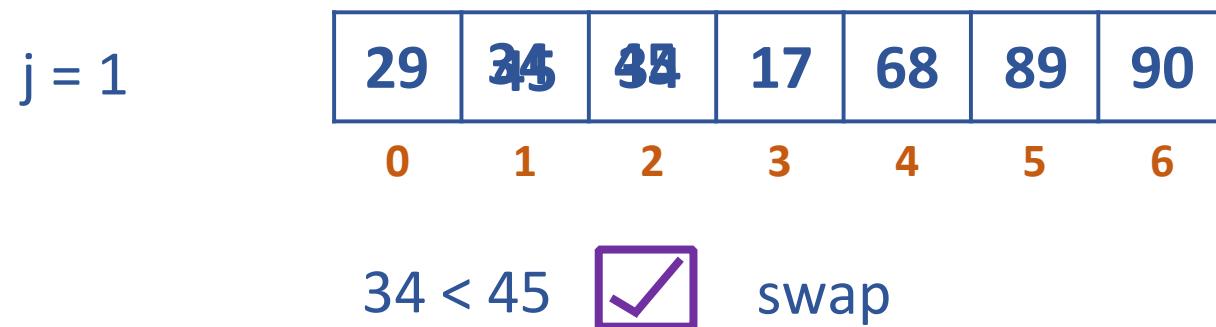
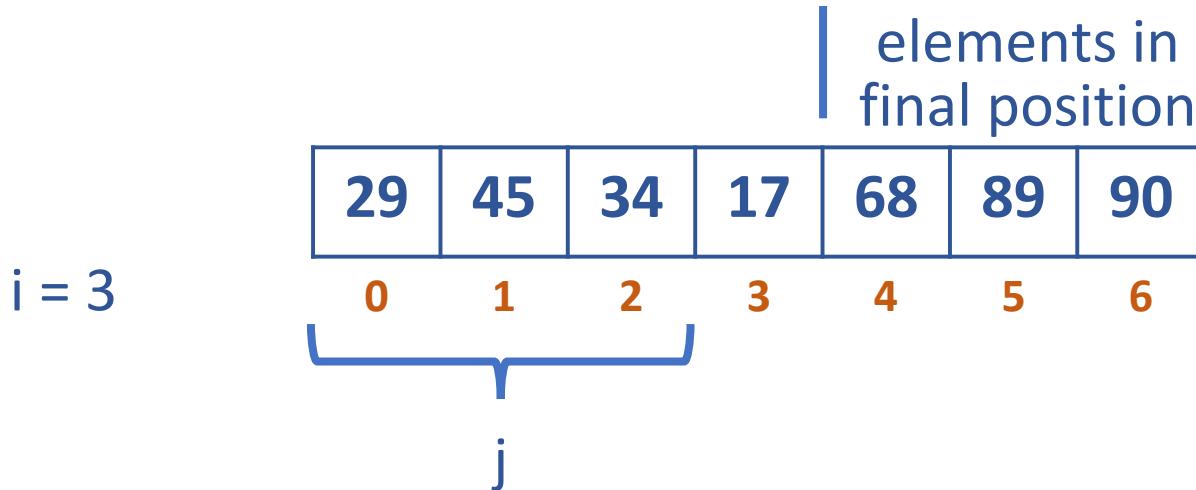
$j = 0$

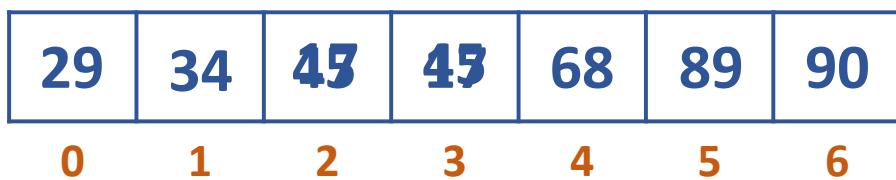
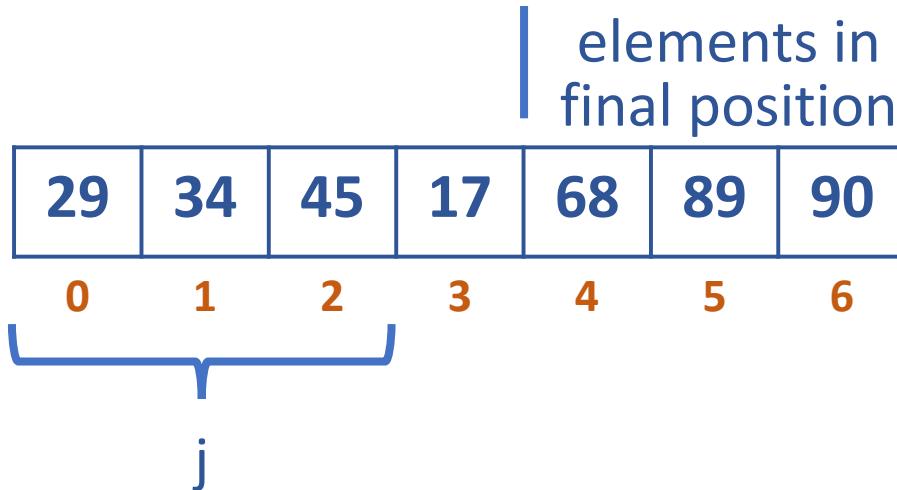
29	45	34	17	68	89	90
0	1	2	3	4	5	6

$29 < 45$  swap

DESIGN AND ANALYSIS OF ALGORITHMS

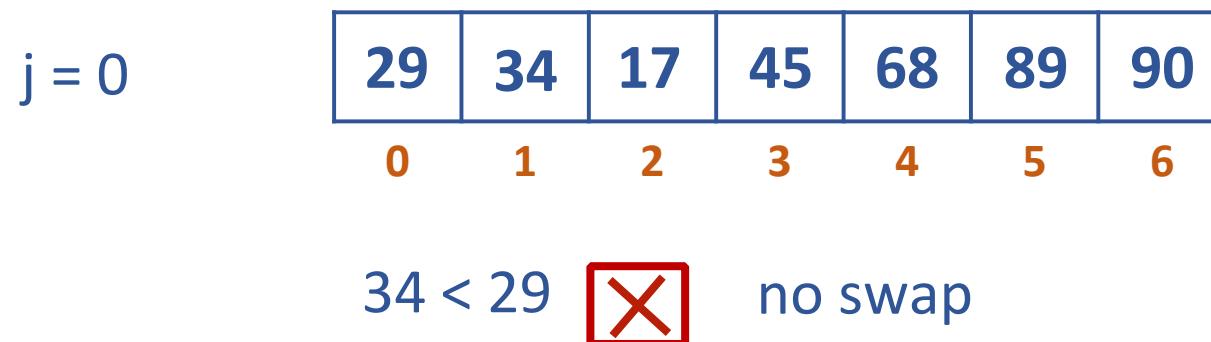
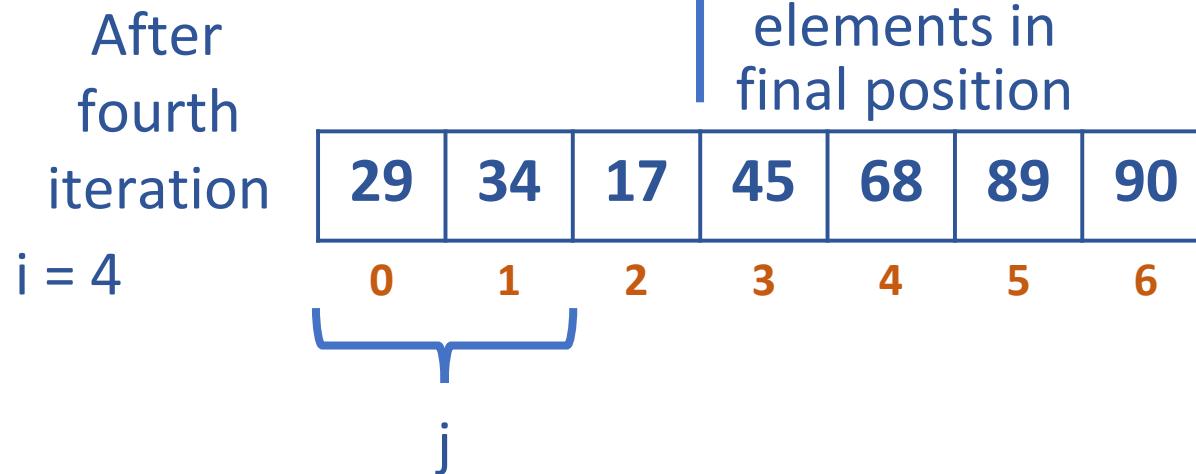
Bubble Sort

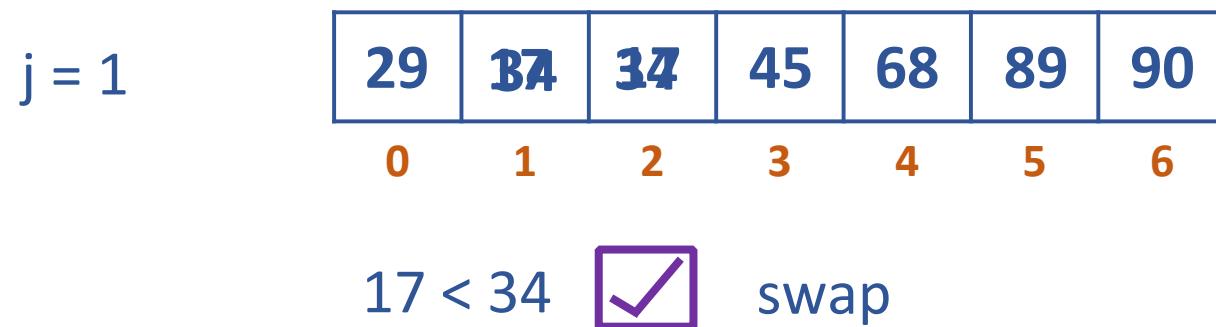
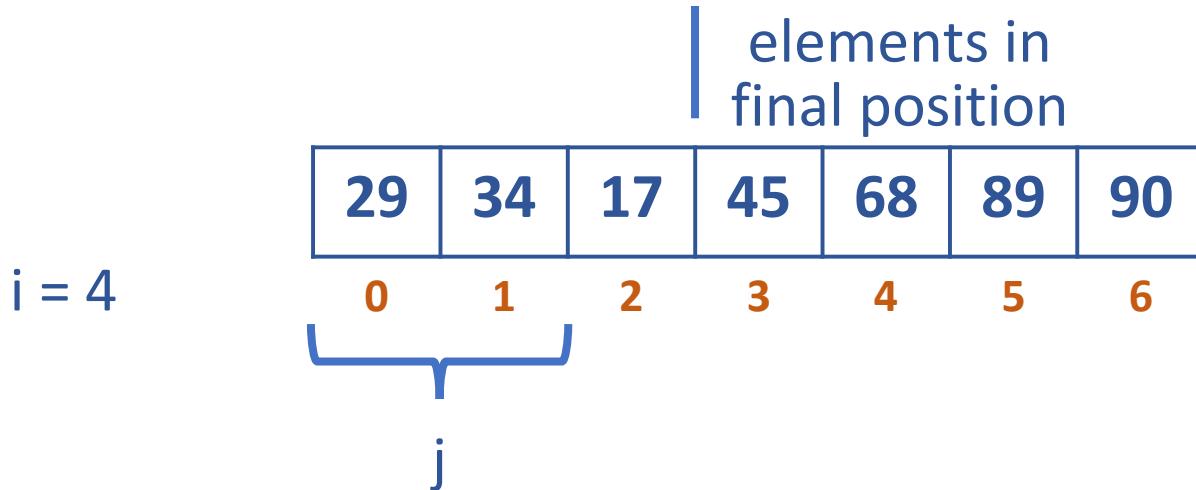




$17 < 45$  swap

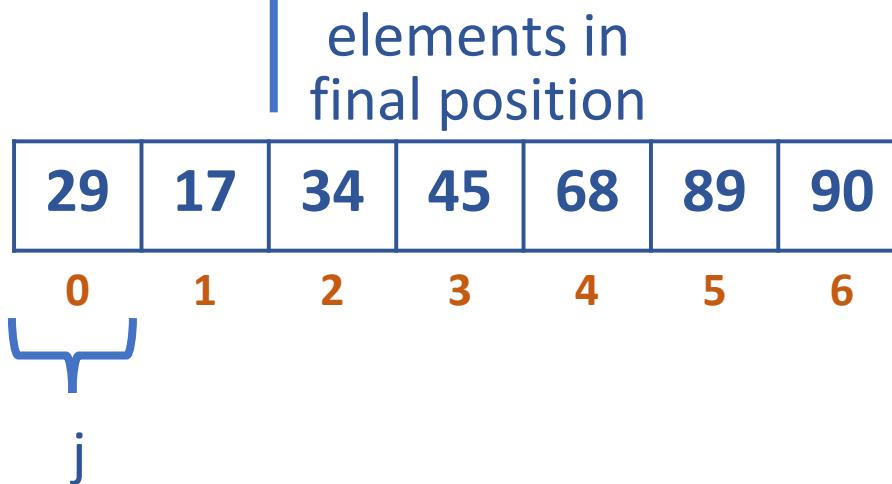
Bubble Sort





Bubble Sort

After
fifth
iteration



j = 0

29	29	34	45	68	89	90
0	1	2	3	4	5	6

17 < 29  swap

Bubble Sort

After sixth iteration	elements in final position	17	29	34	45	68	89	90
		0	1	2	3	4	5	6

ALGORITHM BubbleSort($A[0 .. n - 1]$)

//Sorts a given array by bubble sort in their final positions

//Input: An array $A[0 .. n - 1]$ of orderable elements

//Output: Array $A[0 .. n - 1]$ sorted in ascending order

for $i \leftarrow 0$ to $n - 2$ do

 for $j \leftarrow 0$ to $n - 2 - i$ do

 if $A[j + 1] < A[j]$ swap $A[j]$ and $A[j + 1]$

Bubble Sort Analysis

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n - 2 - i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n - 1 - i) = \frac{(n - 1)n}{2} \in \Theta(n^2)$$

Bubble Sort is a $\Theta(n^2)$ algorithm

DESIGN AND ANALYSIS OF ALGORITHMS

Sequential Search

- Compares successive elements of a given list with a given search key until:
 - A match is encountered (Successful Search)
 - List is exhausted without finding a match (Unsuccessful Search)
- An improvisation to the algorithm is to append the key to the end of the list
- This means the search has to be successful always and we can eliminate the end of list check

DESIGN AND ANALYSIS OF ALGORITHMS

Sequential Search

- Sequential / Linear Search

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

- For key = 33, 6 is returned
- For key = 50, -1 is returned

DESIGN AND ANALYSIS OF ALGORITHMS

Sequential Search

```
ALGORITHM SequentialSearch2(A[0 .. n ], K)
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0 .. n -1] whose value is
// equal to K or -1 if no such element is found
A[n]<---K
i<---0
while A[i] ≠ K do
    i<--- i + 1
if i < n return i
else return -1
```

DESIGN AND ANALYSIS OF ALGORITHMS

Sequential Search

Sequential Search Analysis

- Sequential Search is a $\Theta(n)$ algorithm

DESIGN AND ANALYSIS OF ALGORITHMS

Brute – Force String Matching

String Matching - Terms

- pattern:
a string of m characters to search for
- text:
a (longer) string of n characters to search in
- problem:
find a substring in the text that matches the pattern

DESIGN AND ANALYSIS OF ALGORITHMS

String Matching Idea

Step 1: Align pattern at beginning of text

Step 2: Moving from left to right, compare each character of pattern to the corresponding character in text until:

- all characters are found to match (successful search); or
- a mismatch is detected

Step 3: While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

DESIGN AND ANALYSIS OF ALGORITHMS

String Matching

```
ALGORITHM BruteForceStringMatch(T[0 .. n -1], P[0 .. m -1])
//Implements brute-force string matching
//Input: An array T[0 .. n - 1] of n characters representing a text
// and an array P[0 .. m - 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//matching substring or -1 if the search is unsuccessful
for i ← 0 to n-m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j+1
    if j = m return i
return -1
```

DESIGN AND ANALYSIS OF ALGORITHMS

String Matching Example

DESIGN AND ANALYSIS OF ALGORITHMS

String Matching Analysis

Worst Case:

- The algorithm might have to make all the 'm' comparisons for each of the $(n-m+1)$ tries
- Therefore, the algorithm makes $m(n-m+1)$ comparisons
- Brute Force String Matching is a $O(nm)$ algorithm

DESIGN AND ANALYSIS OF ALGORITHMS

Exhaustive Search

- Exhaustive Search is a brute – force problem solving technique
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints and then finding a desired element
- The desired element might be one which minimizes or maximizes a certain characteristic
- Typically the problem domain involves combinatorial objects such as permutations, combinations and subsets of a given set

DESIGN AND ANALYSIS OF ALGORITHMS

Exhaustive Search - Method

- Generate a list of all potential solutions to the problem in a systematic manner
- Evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- When search ends, announce the solution(s) found

DESIGN AND ANALYSIS OF ALGORITHMS

Travelling Salesman Problem

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

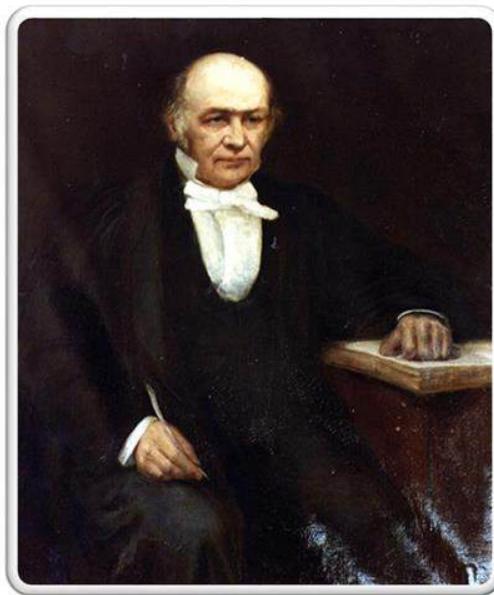
Alternative way to state the problem:

- Find the shortest Hamiltonian Circuit in a weighted connected graph

DESIGN AND ANALYSIS OF ALGORITHMS

TSP: History and Relevance

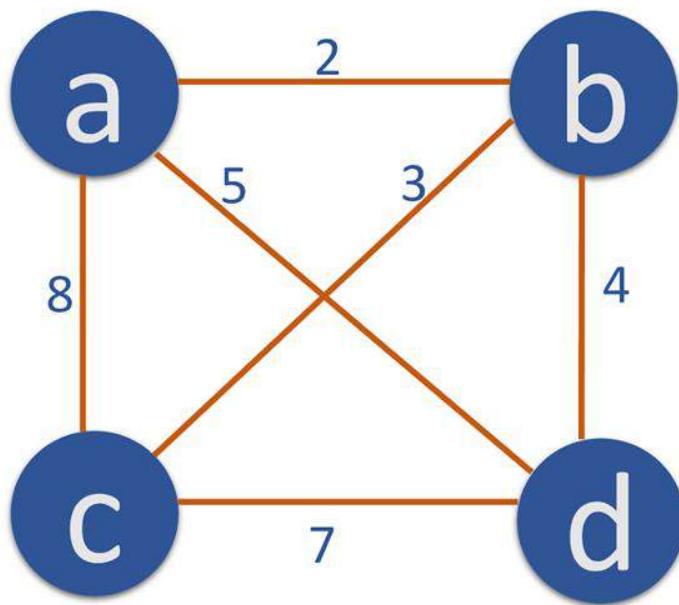
- The Travelling Salesman Problem was mathematically formulated by Irish Mathematician Sir William Rowan Hamilton
- It is one of the most intensively studied problems in optimization
- It has applications in logistics and planning



DESIGN AND ANALYSIS OF ALGORITHMS

Travelling Salesman Problem

Example



Tour	Length
a → b → c → d → a	$2+3+7+5 = 17$
a → b → d → c → a	$2+4+7+8 = 21$
a → c → b → d → a	$8+3+4+5 = 20$
a → c → d → b → a	$8+7+4+2 = 21$
a → d → b → c → a	$5+4+3+8 = 20$
a → d → c → b → a	$5+7+3+2 = 17$

DESIGN AND ANALYSIS OF ALGORITHMS

Travelling Salesman Problem

Efficiency

- The Exhaustive Search solution to the Travelling Salesman problem can be obtained by keeping the origin city constant and generating permutations of all the other $n - 1$ cities
- Thus, the total number of permutations needed will be $(n - 1)!$

DESIGN AND ANALYSIS OF ALGORITHMS

Knapsack Problem

Given n items:

- weights: $w_1 \ w_2 \dots \ w_n$
- values: $v_1 \ v_2 \dots \ v_n$
- a knapsack of capacity W

Find the most valuable subset of items that fit into the knapsack

DESIGN AND ANALYSIS OF ALGORITHMS

Knapsack Problem

Example

Knapsack Capacity W = 16

Item	Weight	Value
1	2	20
2	5	30
3	10	50
4	5	10

DESIGN AND ANALYSIS OF ALGORITHMS

Knapsack Problem

Subset	Total Weight	Total Value
{1}	2	20
{2}	5	30
{3}	10	50
{4}	5	10
{1, 2}	7	50
{1, 3}	12	70
{1, 4}	7	30
{2, 3}	15	80
{2, 4}	10	40
{3, 4}	15	60
{1, 2, 3}	17	Not Feasible
{1, 2, 4}	12	60
{1, 3, 4}	17	Not Feasible
{2, 3, 4}	20	Not Feasible
{1, 2, 3, 4}	22	Not Feasible

Knapsack Problem by
Exhaustive Search

DESIGN AND ANALYSIS OF ALGORITHMS

Knapsack Problem

- The Exhaustive Search solution to the Knapsack Problem is obtained by generating all subsets of the set of n items given and computing the total weight of each subset in order to identify the feasible subsets
- The number of subsets for a set of n elements is 2^n
- The Exhaustive Search solution to the Knapsack Problem belongs to $\Omega(2^n)$

DESIGN AND ANALYSIS OF ALGORITHMS

The Assignment Problem

- There are n people who need to be assigned to n jobs, one person per job. The cost of assigning person i to job j is $C[i, j]$.
Find an assignment that minimizes the total cost

DESIGN AND ANALYSIS OF ALGORITHMS

The Assignment Problem

Example

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

DESIGN AND ANALYSIS OF ALGORITHMS

The Assignment Problem

Algorithmic Plan

1. Generate all legitimate assignments
2. Compute their costs
3. Select the cheapest one

DESIGN AND ANALYSIS OF ALGORITHMS

The Assignment Problem

The Assignment Problem by Exhaustive Search

Assignment	Cost
1, 2, 3, 4	$9 + 4 + 1 + 4 = 18$
1, 3, 4, 2	$9 + 8 + 9 + 7 = 33$
1, 4, 3, 2	$9 + 6 + 1 + 7 = 23$
1, 4, 2, 3	$9 + 6 + 3 + 8 = 26$
1, 3, 2, 4	$9 + 8 + 3 + 4 = 24$
1, 2, 4, 3	$9 + 4 + 9 + 8 = 30$

etc.,

DESIGN AND ANALYSIS OF ALGORITHMS

The Assignment Problem

Efficiency

- The Assignment Problem is solved by generating all permutations of n
- The number of permutations for a given number n is $n!$
- Therefore, the exhaustive search is impractical for all but very small instances of the problem

Decrease-by-a-Constant-Factor Algorithms:

```
Algorithm BinarySearchRec (A[0..n-1] , K)
    if(n <= 0)
        return -1
    m = [n/2]
    if(k = A[m])
        return m
    if(k < A[m])
        return BinarySearchRec (A[0..m-1] , K)
    else
        return BinarySearchRec (A[m+1..n-1] , K)
```

Worst Case Time complexity: $O(\log_2 n)$

Fake-Coin Problem

Among n identical-looking coins, one is fake. With a balance scale, we can compare any two sets of coins.

The problem is to design an efficient algorithm for detecting the fake coin.

The most natural idea for solving this problem is to divide n coins into two piles of $n/2$ coins each, leaving one extra coin aside if n is odd, and put the two piles on the scale.

If the piles weigh the same, the coin put aside must be fake; otherwise, we can proceed in the same manner with the lighter pile, which must be the one with the fake coin.

$$W(n) = W(n/2) + 1 \text{ for } n > 1, \quad W(1) = 0.$$

$$W(n) = \log_2 n.$$

It would be more efficient to divide the coins not into two but into *three* piles of about $n/3$ coins each.

Russian Peasant Multiplication

Let n and m be positive integers whose product we want to compute, and let us measure the instance size by the value of n .

If n is even, an instance of half the size has to deal with $n/2$, and we have an obvious formula relating the solution to the problem's larger instance to the solution to the smaller one:

$$n \cdot m = (n/2) * 2m$$

If n is odd, we need only a slight adjustment of this formula:

$$n \cdot m = ((n - 1)/2) \cdot 2m + m.$$

Using these formulas and the trivial case of $1 \cdot m = m$ to stop, we can compute product $n \cdot m$ either recursively or iteratively.

<i>n</i>	<i>m</i>
50	65
25	130
12	260 (+130)
6	520
3	1040
1	2080 (+1040)
	2080 +(130 + 1040) = 3250

(a)

<i>n</i>	<i>m</i>
50	65
25	130 130
12	260
6	520
3	1040 1040
1	2080 <u>2080</u>
	3250

(b)

FIGURE 4.11 Computing $50 \cdot 65$ by the Russian peasant method.

Josephus problem

Named for Flavius Josephus, a famous Jewish historian who participated in and chronicled the Jewish revolt of 66–70 c.e. against the Romans.

Josephus, as a general, managed to hold the fortress of Jotapata for 47 days, but after the fall of the city he took refuge with 40 diehards in a nearby cave. There, the rebels voted to perish rather than surrender. Josephus proposed that each man in turn should dispatch his neighbor, the order to be determined by casting lots. Josephus contrived to draw the last lot, and, as one of the two surviving men in the cave, he prevailed upon his intended victim to surrender to the Romans.

Josephus Problem

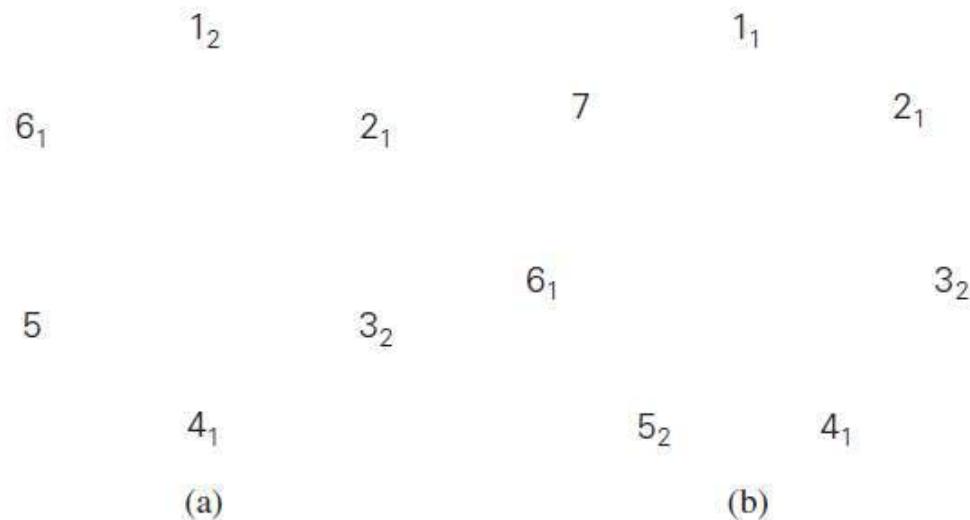
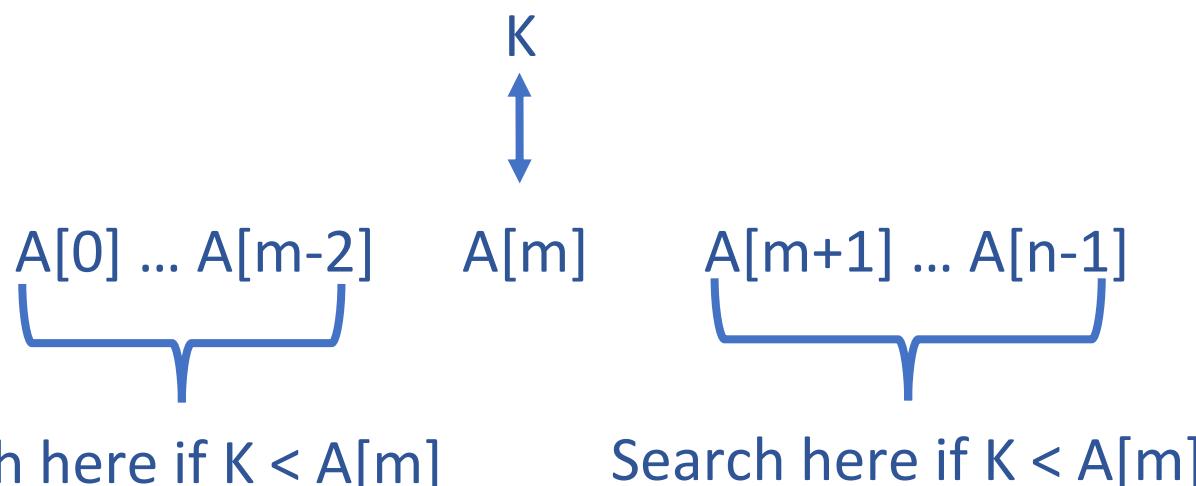


FIGURE 4.12 Instances of the Josephus problem for (a) $n = 6$ and (b) $n = 7$. Subscript numbers indicate the pass on which the person in that position is eliminated. The solutions are $J(6) = 5$ and $J(7) = 7$, respectively.

- Binary Search is a remarkably efficient algorithm for searching in a sorted array
- It works by comparing the search key K with the array's middle element $A[m]$
- If they match, the algorithm stops
- Otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$ and for the second half if $K > A[m]$



```
ALGORITHM BinarySearch(A[0 .. n -1], K)
// Implements non recursive binary search
// Input: An array A[0 .. n - 1] sorted in ascending order and a
// search key K
// Output: An index of the array's element that is equal to K or
// -1 if there is no such element
l ⊑ 0; r ⊑ n-1
while l ≤ r do
    m ←  $\lfloor(l + r)/2\rfloor$ 
    if K = A[m] return m
    else if K < A[m] r ⊑ m-1
    else l ⊑ m+1
return -1
```

DESIGN AND ANALYSIS OF ALGORITHMS



Binary Search - Example

Search Key K = 70

DESIGN AND ANALYSIS OF ALGORITHMS

Binary Search Vs Linear Search

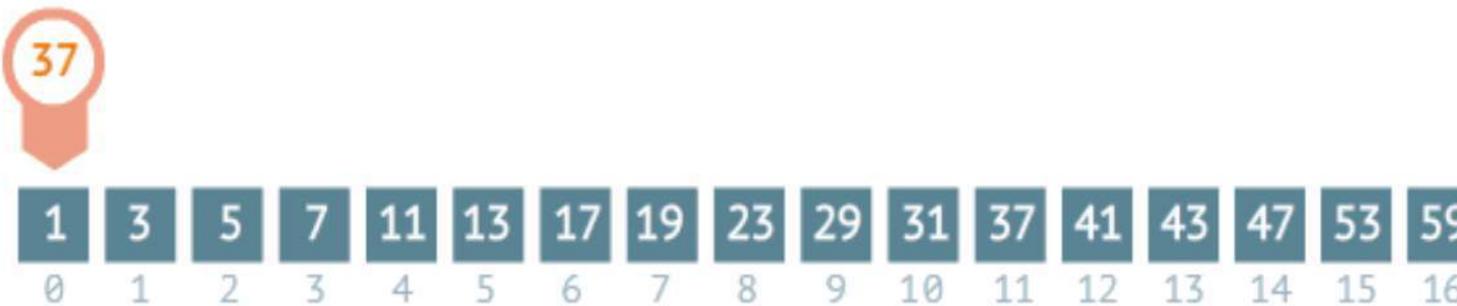
Binary search

steps: 0



Sequential search

steps: 0



The basic operation is the comparison of the search key with an element of the array

The number of comparisons made are given by the following recurrence:

$$C_{worst}(n) = C_{worst}\left(\left\lfloor n/2 \right\rfloor\right) + 1 \text{ for } n > 1, C_{worst}(1) = 1$$

For the initial condition $C_{worst}(1) = 1$, we obtain:

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1$$

For any arbitrary positive integer, n:

$$C_{worst}(n) = \lceil \log_2 n \rceil + 1$$

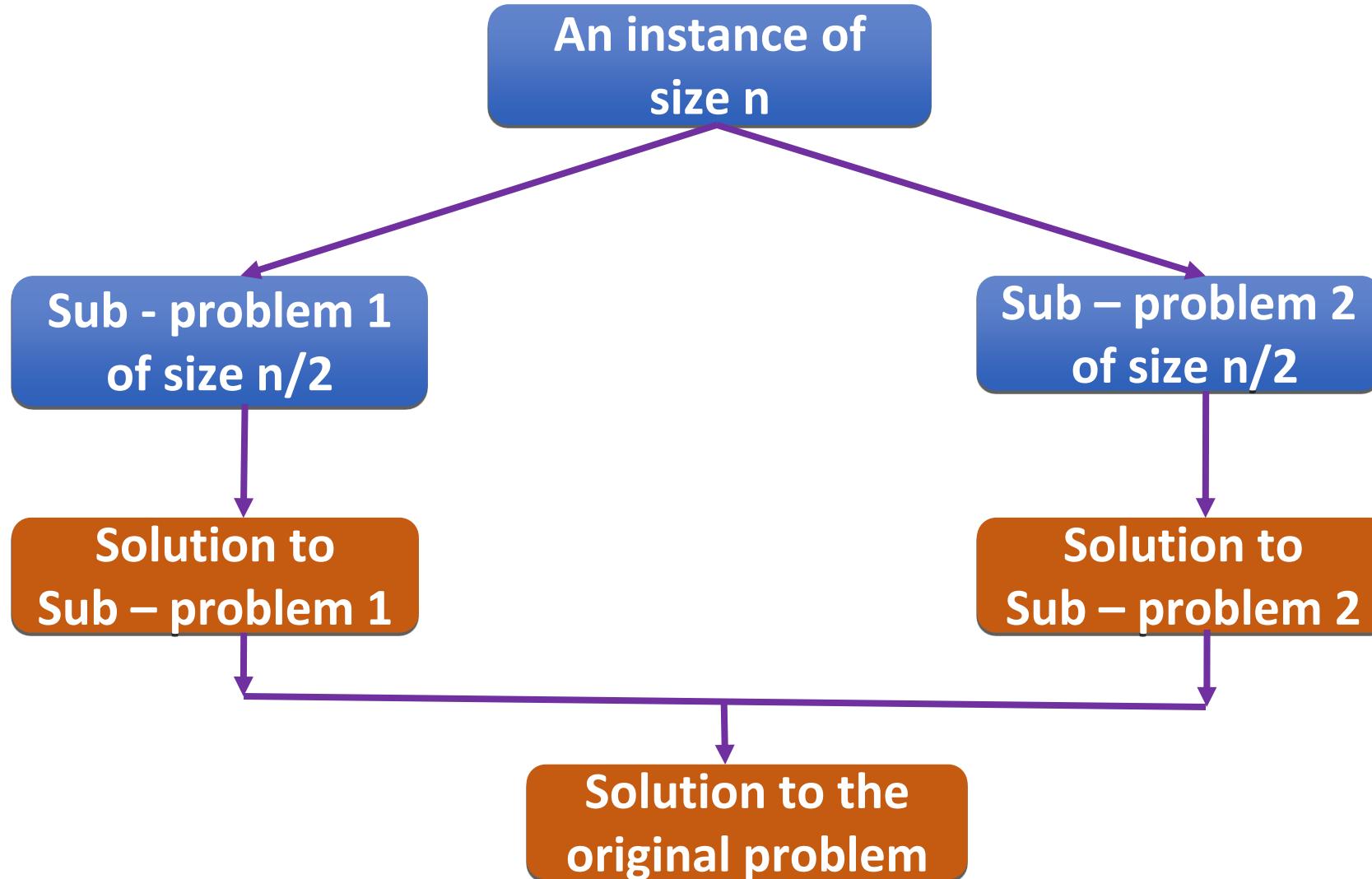
$$C_{avg} \approx \log_2 n$$

Divide and Conquer – Idea

- Divide and Conquer is one of the most well – known algorithm design strategies
- The principle underlying Divide and Conquer strategy can be stated as follows:
 - Divide the given instance of the problem into two or more smaller instances
 - Solve the smaller instances recursively
 - Combine the solutions of the smaller instances and obtain the solution for the original instance

Divide and Conquer – Idea

- Divide and Conquer



Recurrence

- In the most typical cases of Divide and Conquer, a problem's instance of size n can be divided into b instances of size n/b , with a of them needing to be solved
- Here a and b are constants; $a \geq 1$ and $b \geq 1$
- Assuming that size n is a power of b , we get the following recurrence for the running time:

$$T(n) = a * T(n/b) + f(n)$$

- $f(n)$ is a function that accounts for the time spent on dividing the problem and combining the solutions

Recurrence

- For the recurrence:

$$T(n) = a * T(n/b) + f(n)$$

- If $f(n) \in \Theta(n^d)$, where $d \geq 0$ in the recurrence relation, then:

- If $a < b^d$, $T(n) \in \Theta(n^d)$
- If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
- If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

- Analogous results hold for O and Ω as well!

DESIGN AND ANALYSIS OF ALGORITHMS

Quick Sort

- Select a pivot (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first s positions are smaller than or equal to the pivot and all the elements in the remaining $n-s$ positions are larger than or equal to the pivot



- Exchange the pivot with the last element in the first (i.e., \leq) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

DESIGN AND ANALYSIS OF ALGORITHMS

Quick Sort - Algorithm

```
ALGORITHM Quicksort(A[l .. r])
// Sorts a subarray by quicksort
// Input: A subarray A[l ... r] of A[0 .. n -1], defined by its left and
// right indices l and r
// Output: Subarray A[l .. r] sorted in non decreasing order
if l < r
    s ← Partition(A[l .. r])      //s is a split position
    Quicksort(A[l .. s - 1])
    Quicksort(A[s + 1 .. r])
```

DESIGN AND ANALYSIS OF ALGORITHMS

Quick Sort - Algorithm

ALGORITHM Partition($A[l .. r]$)

// Partitions a subarray by using its first element as a pivot

// Input: A subarray $A[l..r]$ of $A[0 .. n - 1]$, defined by its left and right indices l and r ($l < r$)

// Output: A partition of $A[l .. r]$, with the split position returned as this function's value

$p \leftarrow A[l]$

$i \leftarrow l; j \leftarrow r + 1$

repeat

 repeat $i \leftarrow i + 1$ until $A[i] \geq p$

 repeat $j \leftarrow j - 1$ until $A[j] \leq p$

 swap($A[i], A[j]$)

until $i \geq j$

swap($A[i], A[j]$) //undo last swap when $i \geq j$

swap($A[i], A[j]$)

return j

DESIGN AND ANALYSIS OF ALGORITHMS

Quick Sort - Example

5 3 1 9 8 2 4 7

2 3 1 4 5 8 9 7

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

DESIGN AND ANALYSIS OF ALGORITHMS

Quick Sort - Analysis: Best Case

- The number of comparisons in the best case satisfies the recurrence:
- $C_{\text{best}}(n) = 2C_{\text{best}}(n/2) + n \quad \text{for } n > 1, C_{\text{best}}(1) = 0$
- According to Master Theorem

$$C_{\text{best}}(n) \in \Theta(n \log_2 n)$$

DESIGN AND ANALYSIS OF ALGORITHMS

Quick Sort - Analysis: Worst Case

- The number of comparisons in the worst case satisfies the recurrence

$$C_{\text{worst}}(n) = (n + 1) + n + \dots + 3 = \frac{(n + 1)(n + 2)}{2} - 3 \in \Theta(n^2)$$

DESIGN AND ANALYSIS OF ALGORITHMS

Quick Sort - Analysis: Average Case

Let $C_{avg}(n)$ be the number of key comparisons made by Quick Sort on a randomly ordered array of size n

$$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1$$

The solution for the above recurrence is:

$$C_{avg}(n) \approx 2n \ln n \approx 1.38n \log_2 n$$

The third principal variety of decrease-and-conquer, the **size reduction pattern** varies from one iteration of the algorithm to another.

This algorithm helps optimize:

1. Computing a Median and the Selection Problem
2. Interpolation Search
3. Searching and Insertion in a Binary Search Tree
4. The Game of Nim

We will look at **Computing a Median and the Selection Problem** in detail.

Computing a Median and the Selection Problem

Objective: Find the median of an odd array in the best time.

What is the Median?

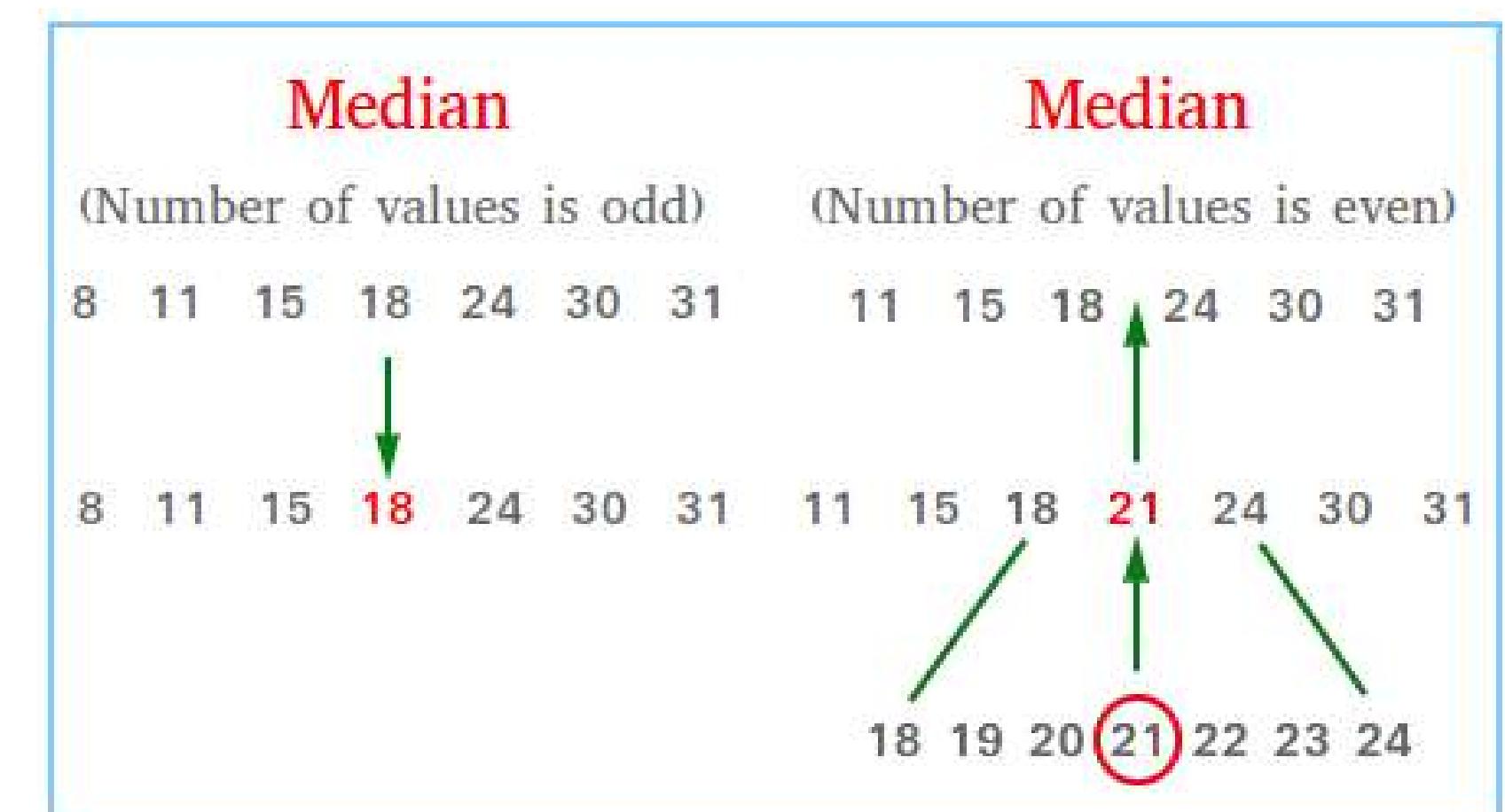
The median is the middle value of a sorted array.

Naïve Approach:

1. Sort the array.
2. Find the element in the middle position.

Time complexity:

- Sorting: $O(n \log n)$
- Selecting the middle element: $O(1)$
- Total: $O(n \log n)$



Computing a Median and the Selection Problem

Issues with the Naïve Approach:

- Sorting places all elements in their correct positions.
- We only need the middle element, so sorting the entire array is unnecessary.

What can be optimized:

- Instead of sorting all elements, directly try to find the element at the $n/2$ position.
- This problem of finding the k -th smallest or largest element is called the **Selection Problem**.

Proposed Solution:

- QuickSelect to solve the Selection Problem

Computing a Median and the Selection Problem

Quick Select:

- Similar to Quick Sort, we pick a **pivot** element and **partition** the array:
 - Elements **smaller** than the pivot go to the **left**.
 - Elements **larger** than the pivot go to the **right**.
 - The **pivot** is placed in its **correct** position.
 - This partition technique is called **Lomuto Partition**.
- Difference from Quick Sort:
 - Quick Sort recursively sorts the entire array.
 - Quick Select **stops** once the pivot is at the k-th position ($n/2$ for median).

Algorithm for QuickSelect

Input : List, left is first position of list, right is last position of list and k is k-th smallest element.

Output : A new list is partitioned.

```
quickSelect(list, left, right, k)
```

```
    if left = right
```

```
        return list[left]
```

```
// Select a pivotIndex between left and right
```

```
pivotIndex <= partition(list, left, right, pivotIndex)
```

```
if k = pivotIndex
```

```
    return list[k]
```

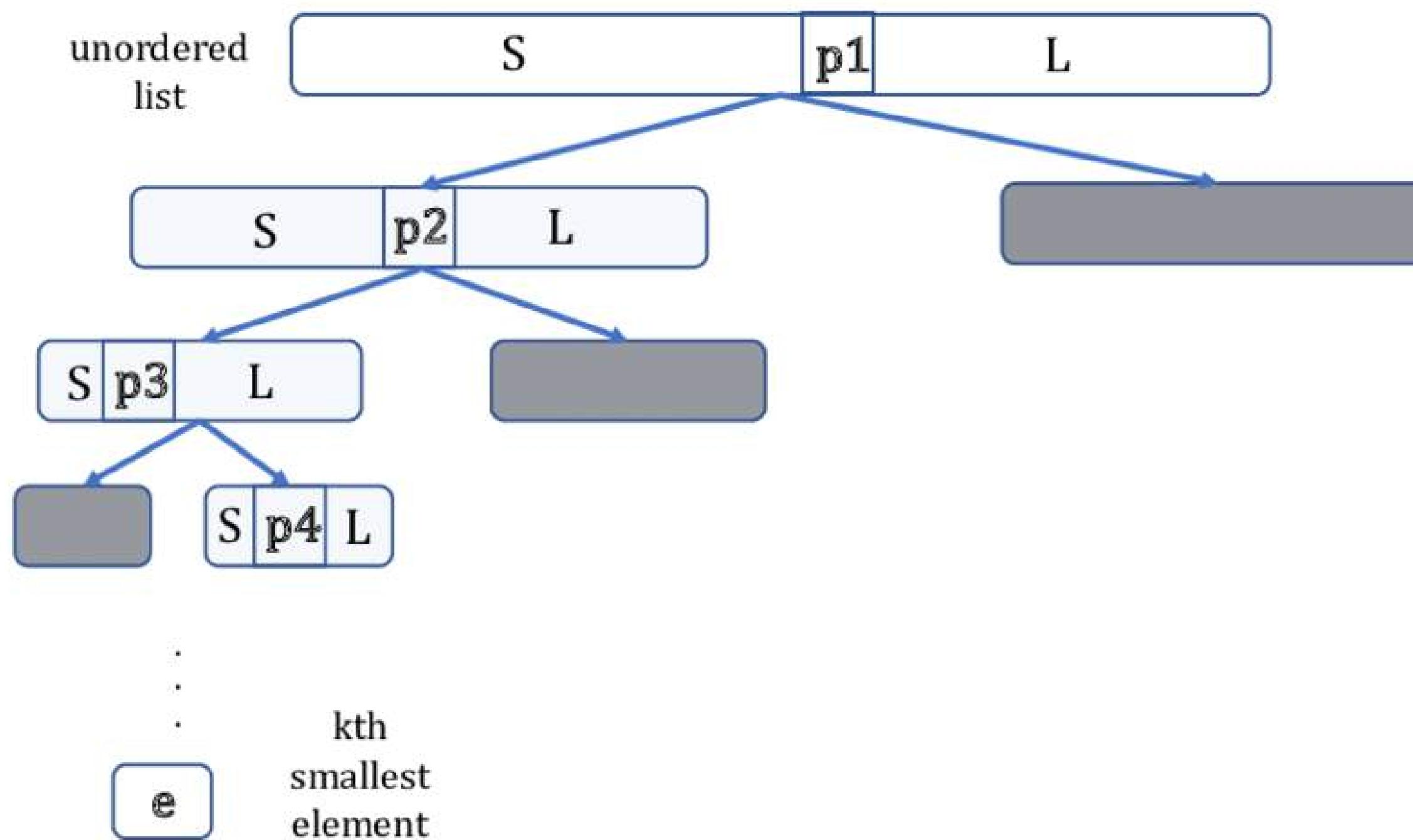
```
else if k < pivotIndex
```

```
    right <= pivotIndex - 1
```

```
else
```

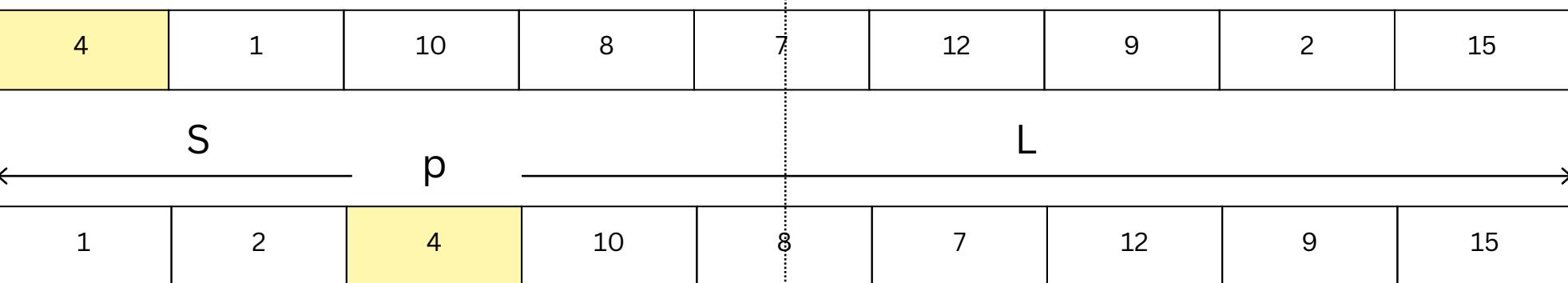
```
    left <= pivotIndex + 1
```

QuickSelect



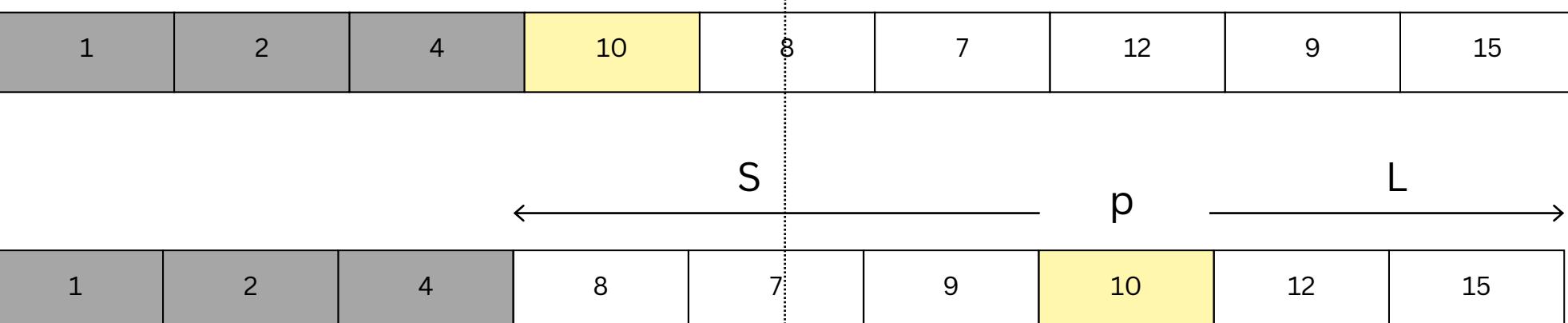
Find the 5th smallest element in the given array

pivot(first element) = 4



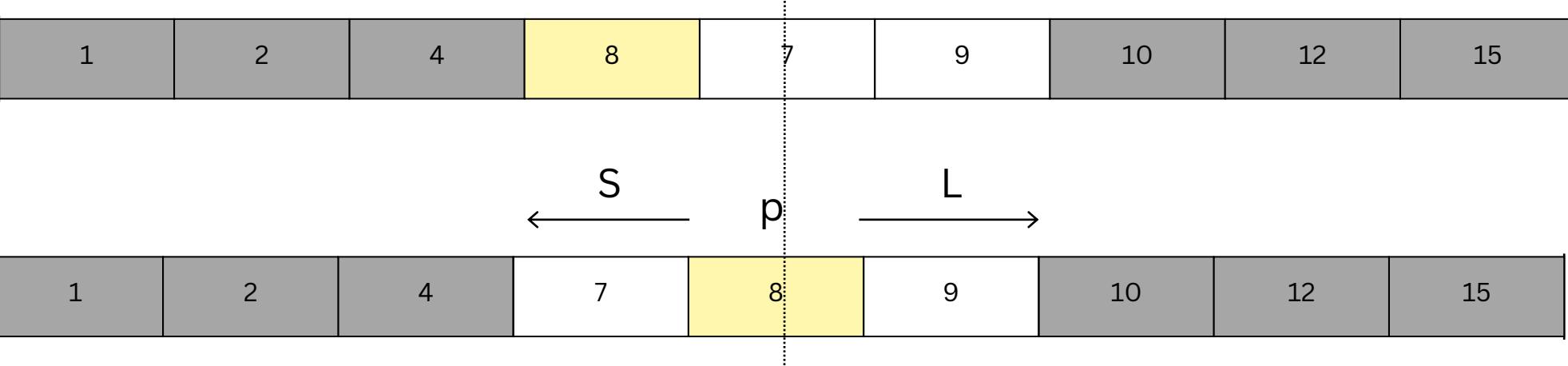
$p < k$, repeat for right array

pivot(first element) = 10



$p > k$, repeat for left array

pivot(first element) = 10



5th smallest element is 8

Time Complexities for QuickSelect

Best Case:

- The best case occurs when the pivot chosen is always the median of the array. This perfectly divides the array in half at each step.
- The recurrence relation: $T(n)=T(n/2)+O(n)$
- Solving this gives $O(n)$ time complexity, and the problem size is reduced exponentially.

Worst Case:

- The worst case occurs when the pivot selection is poor, e.g., always choosing the smallest or largest element. This results in highly unbalanced partitions, where one side has $n - 1$ elements and the other has 0.
- The recurrence becomes: $T(n)=T(n-1)+O(n)$
- Solving this gives $O(n^2)$ time complexity.

Improve the Worst Case Complexity

There are two key ways to achieve better worst-case complexity:

1. Choosing a Better Pivot Selection Strategy such as Median of Medians Algorithm.
2. Using an Improved Partitioning Algorithm such as using Hoare's Partition Scheme instead of Lomuto's Partition Scheme can improve efficiency by reducing swaps.

Median of Medians Algorithm (Guaranteed $O(n)$ Worst-Case):

We need to **choose a pivot close to the median** consistently. This ensures better partitioning and prevents highly unbalanced cases.

- Divide the array into groups of 5 elements (or another small fixed size).
- Find the median of each group.
- Recursively find the median of these medians to use as the pivot.
- Ensures a pivot that is reasonably close to the true median, avoiding worst-case $O(n^2)$.
- Guarantees $O(n)$ worst-case time complexity.

By utilizing **variable-size Decrease and Conquer**, we optimized the Selection Algorithm, making median computation more efficient.

Unlike traditional divide-and-conquer, where the problem size is reduced by a fixed factor (e.g., halving in binary search), **variable-size** reduction allows for **adaptive problem shrinking** at each step. In Quick Select, this means:

- Instead of processing the entire array, we eliminate a portion based on the pivot's position.
- The size reduction varies **dynamically**, depending on where the pivot is placed.
- This flexibility enables an **$O(n)$** average-time complexity, significantly improving over naive sorting methods.

By using variable Decrease and Conquer, we were able to directly target the element we needed, making the solution far more efficient.

1.3 Master theorem

The master theorem is a formula for solving recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$ and $b > 1$ and $f(n)$ is asymptotically positive. (Asymptotically positive means that the function is positive for all sufficiently large n .)

This recurrence describes an algorithm that divides a problem of size n into a subproblems, each of size n/b , and solves them recursively. (Note that n/b might not be an integer, but in section 4.6 of the book, they prove that replacing $T(n/b)$ with $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ does not affect the asymptotic behavior of the recurrence. So we will just ignore floors and ceilings here.)

The theorem is as follows:

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■

The master theorem compares the function $n^{\log_b a}$ to the function $f(n)$. Intuitively, if $n^{\log_b a}$ is larger (by a polynomial factor), then the solution is $T(n) = \Theta(n^{\log_b a})$. If $f(n)$ is larger (by a polynomial factor), then the solution is $T(n) = \Theta(f(n))$. If they are the same size, then we multiply by a logarithmic factor.

Be warned that these cases are not exhaustive – for example, it is possible for $f(n)$ to be asymptotically larger than $n^{\log_b a}$, but not larger by a polynomial factor (no matter how small the exponent in the polynomial is). For example, this is true when $f(n) = n^{\log_b a} \log n$. In this situation, the master theorem would not apply, and you would have to use another method to solve the recurrence.

1.3.1 Examples:

To use the master theorem, we simply plug the numbers into the formula.

Example 1: $T(n) = 9T(n/3) + n$. Here $a = 9$, $b = 3$, $f(n) = n$, and $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9 - \epsilon})$ for $\epsilon = 1$, case 1 of the master theorem applies, and the solution is $T(n) = \Theta(n^2)$.

Example 2: $T(n) = T(2n/3) + 1$. Here $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^0 = 1$. Since $f(n) = \Theta(n^{\log_b a})$, case 2 of the master theorem applies, so the solution is $T(n) = \Theta(\log n)$.

Example 3: $T(n) = 3T(n/4) + n \log n$. Here $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. For $\epsilon = 0.2$, we have $f(n) = \Omega(n^{\log_4 3 + \epsilon})$. So case 3 applies if we can show that $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n . This would mean $3\frac{n}{4} \log \frac{n}{4} \leq cn \log n$. Setting $c = 3/4$ would cause this condition to be satisfied.

Example 4: $T(n) = 2T(n/2) + n \log n$. Here the master method does not apply. $n^{\log_b a} = n$, and $f(n) = n \log n$. Case 3 does not apply because even though $n \log n$ is asymptotically larger than n , it is not polynomially larger. That is, the ratio $f(n)/n^{\log_b a} = \log n$ is asymptotically less than n^ϵ for all positive constants ϵ .

DESIGN AND ANALYSIS OF ALGORITHMS

Multiplication of large integers - Idea

- Let the two numbers being multiplied be a and b
- a and b are n -digit integers, where n is a positive even number
- Let the first half of a 's digits be a_1 and second half be a_0
- Similarly, let the first half of b 's digits be b_1 and second half be b_0
- In these notations, $a = a_1a_0$ implies $a = a_1 * 10^{n/2} + a_0$ and $b = b_1b_0$
implies $b = b_1 * 10^{n/2} + b_0$

DESIGN AND ANALYSIS OF ALGORITHMS

Multiplication of large integers - Idea

$$\begin{aligned}c &= a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\&= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\&= c_2 10^n + c_1 10^{n/2} + c_0\end{aligned}$$

where

$c_2 = a_1 * b_1$ is the product of their first halves

$c_0 = a_0 * b_0$ is the product of their second halves

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's

halves and the sum of the b's halves minus the sum of c_2 and c_0

DESIGN AND ANALYSIS OF ALGORITHMS

Multiplication of large integers - Analysis

- $M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$
- Solving it by backward substitutions for $n = 2^k$ yields:

$$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2M(2^{k-2})$$

$$= \dots = 3^i M(2^{k-i}) = \dots = 3^k M(2^{k-k}) = 3^k$$

- Since $k = \log_2 n$: $M(n) = 3^{\log_2 n} = n^{\log_2 3} = n^{1.585}$
- The number of additions is given by:

$$A(n) = 3A(n/2) + cn \text{ for } n > 1, A(1) = 1$$

$$A(n) \text{ belongs to } \Theta(n^{\log_2 3})$$

DESIGN AND ANALYSIS OF ALGORITHMS

Multiplication of large integers - Example

$$2135 * 4014$$

$$= (21*10^2 + 35) * (40*10^2 + 14)$$

$$= (21*40)*10^4 + c1*10^2 + 35*14$$

where $c1 = (21+35)*(40+14) - 21*40 - 35*14$, and

$$21*40 = (2*10 + 1) * (4*10 + 0)$$

$$= (2*4)*10^2 + c2*10 + 1*0$$

where $c2 = (2+1)*(4+0) - 2*4 - 1*0$, etc.,

DESIGN AND ANALYSIS OF ALGORITHMS

Strassen's Matrix Multiplication

- This algorithm was published by V Strassen in 1969
- The principal insight of the algorithm lies in the discovery that we can find product of two 2 – by – 2 matrices A and B with seven multiplications as opposed to the eight required by the Brute – Force algorithm
- This is accomplished by the following formulae:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$
$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

DESIGN AND ANALYSIS OF ALGORITHMS

Strassen's Matrix Multiplication

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$

$$m_2 = (a_{10} + a_{11}) * b_{00}$$

$$m_3 = a_{00} * (b_{01} - b_{11})$$

$$m_4 = a_{11} * (b_{10} - b_{00})$$

$$m_5 = (a_{00} + a_{01}) * b_{11}$$

$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$$

$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$$

DESIGN AND ANALYSIS OF ALGORITHMS

Strassen's Matrix Multiplication – General Formula

For any two matrices A and B of size n – by – n, we can divide A, B and the product C as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

The sub – matrices can be treated as numbers to get the correct product

DESIGN AND ANALYSIS OF ALGORITHMS

Strassen's Matrix Multiplication – Analysis

- If $M(n)$ is the number of multiplications made by Strassen's algorithm in multiplying two matrices $n \times n$, we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \text{ for } n > 1, M(1) = 1$$

Since $n = 2^k$,

$$\begin{aligned} M(2^k) &= 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \dots \\ &= 7^i M(2^{k-i}) \dots = 7^k M(2^{k-k}) = 7^k \end{aligned}$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$$

DESIGN AND ANALYSIS OF ALGORITHMS

Strassen's Matrix Multiplication – Analysis

- The number of additions are given by the following recurrence:

$$A(n) = 7 A(n/2) + 18(n/2)^2 \quad \text{for } n>1, A(1) = 0$$

- According to Master's Theorem, $A(n)$ belongs to $\Theta(n^{\log_2 7})$

Decrease and Conquer

- **Decrease** or reduce problem instance to smaller instance of the same problem and extend solution.
- **Conquer** the problem by solving smaller instance of the problem.
- **Extend** solution of smaller instance to obtain solution to original problem .
- **Exploit** the relationship between a solution to a given instance of a problem and a solution to its smaller instance.

- Can be implemented either top-down or bottom-up
- Also referred to as *inductive* or *incremental* approach

Decrease and Conquer

3 Types of Decrease and Conquer

Decrease by a constant (usually by 1):

- insertion sort
- graph traversal algorithms (DFS and BFS)
- topological sorting
- algorithms for generating permutations, subsets

Decrease by a constant factor (usually by half)

- binary search and bisection method
- exponentiation by squaring
- multiplication à la russe

Variable-size decrease

- Euclid's algorithm
- selection by partition
- Nim-like games

This usually results in a recursive algorithm.

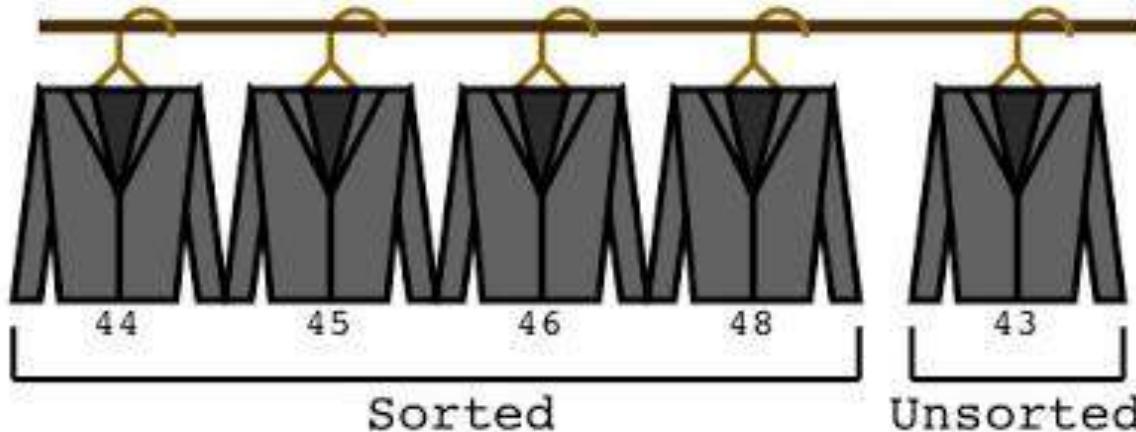
Insertion Sort

Imagine a card game

Cards in your hand are sorted.

The dealer hands you exactly one new card.

How would you rearrange your cards



Insertion Sort

- Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.
- grows the sorted array at each iteration
- compares the current element with the largest value in the sorted array.
- If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position.
- This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead

Insertion Sort

To sort array $A[0..n-1]$, sort $A[0..n-2]$ recursively and then insert $A[n-1]$ in its proper place among the sorted $A[0..n-2]$

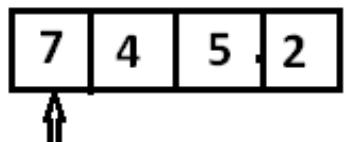
Usually implemented bottom up (nonrecursively)

Example: Sort 6, 4, 1, 8, 5

```
6 | 4  1  8  5  
        4  6 | 1  8  5  
        1  4  6 | 8  5  
        1  4  6  8 | 5  
        1  4  5  6  8
```

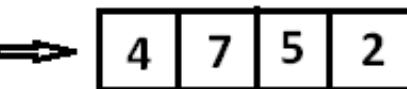
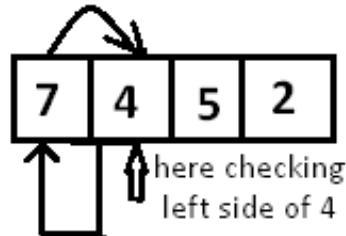
Insertion Sort

STEP 1.



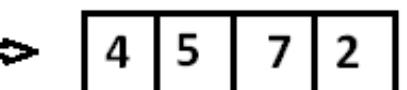
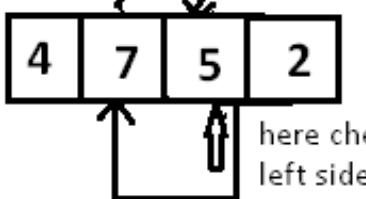
No element on left side of 7, so no change in its position.

STEP 2.



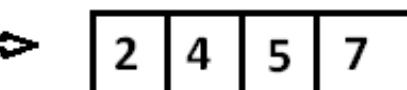
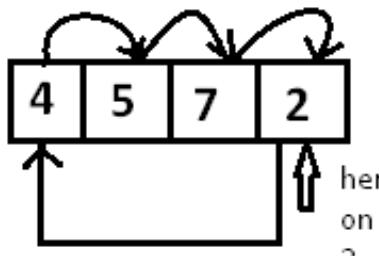
As $7 > 4$, therefore 7 will be moved forward and 4 will be moved to 7's position.

STEP 3.



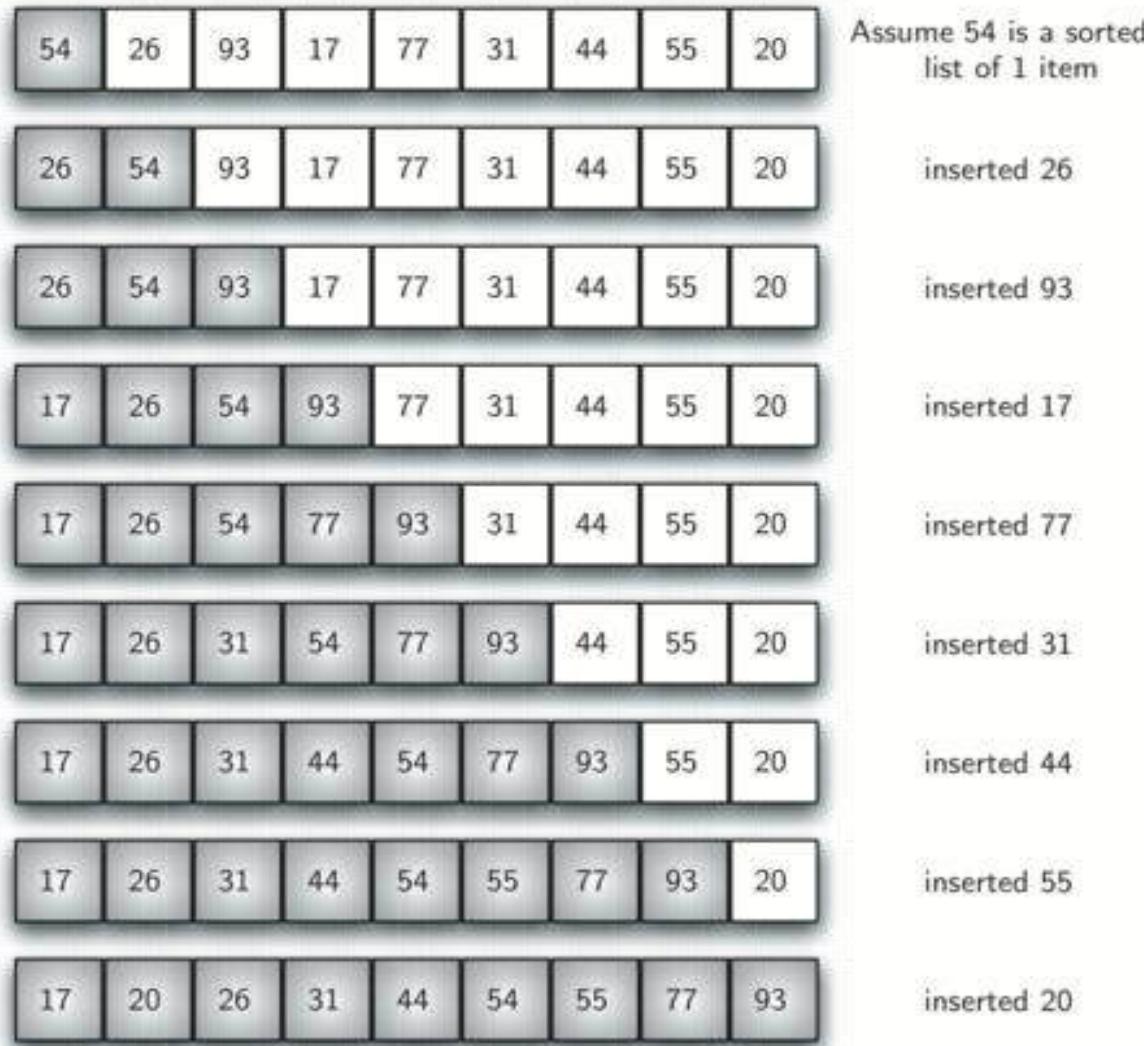
As $7 > 5$, 7 will be moved forward, but $4 < 5$, so no change in position of 4. And 5 will be moved to position of 7.

STEP 4.



As all the elements on left side of 2 are greater than 2, so all the elements will be moved forward and 2 will be shifted to position of 4.

Insertion Sort



ALGORITHM *InsertionSort(A[0..n - 1])*

//Sorts a given array by insertion sort

//Input: An array $A[0..n - 1]$ of n orderable elements

//Output: Array $A[0..n - 1]$ sorted in nondecreasing order

for $i \leftarrow 1$ **to** $n - 1$ **do**

$v \leftarrow A[i]$

$j \leftarrow i - 1$

while $j \geq 0$ **and** $A[j] > v$ **do**

$A[j + 1] \leftarrow A[j]$

$j \leftarrow j - 1$

$A[j + 1] \leftarrow v$

Time efficiency

$$C_{worst}(n) = n(n-1)/2 \in \Theta(n^2)$$

$$C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$$

$$C_{best}(n) = n - 1 \in \Theta(n) \text{ (also fast on almost sorted arrays)}$$

Space efficiency: in-place

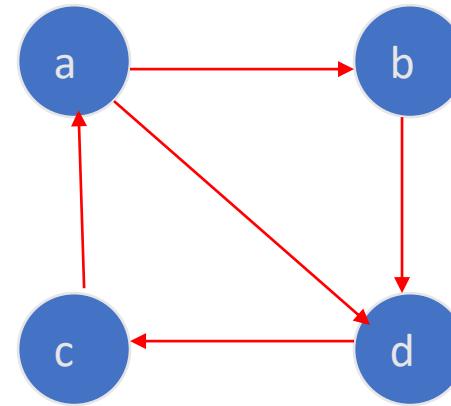
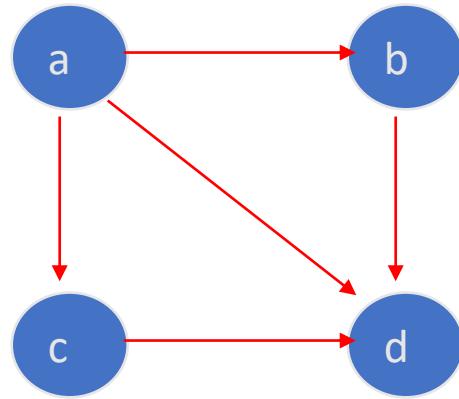
Stability: yes

Best elementary sorting algorithm overall

Binary insertion sort

DAGs and Topological Sorting

DAG: a directed acyclic graph, i.e. a directed graph with no (directed) cycles



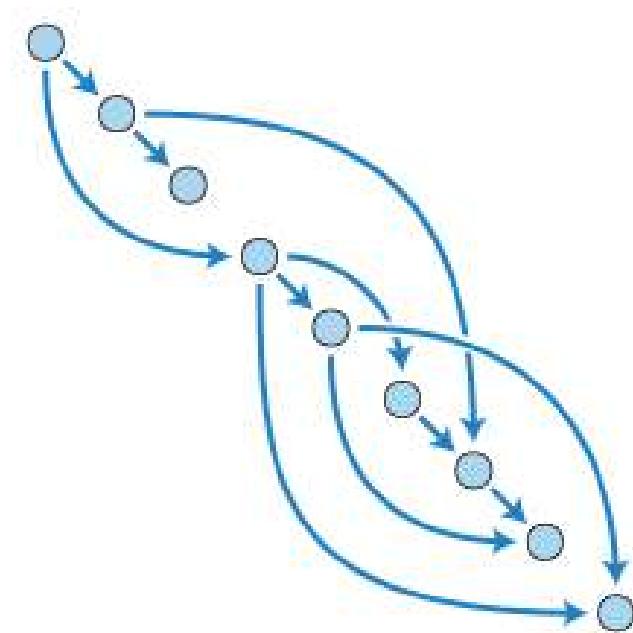
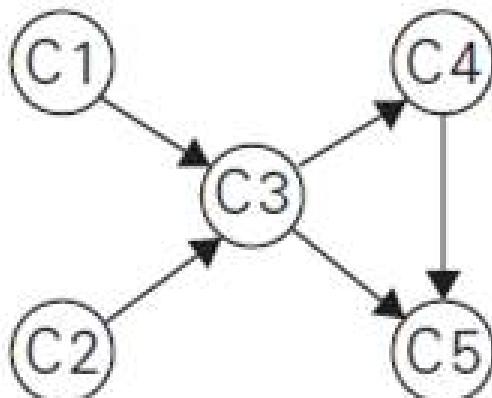
Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)

Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (topological sorting). Being a dag is also a necessary condition for topological sorting to be possible.

Topological Sorting

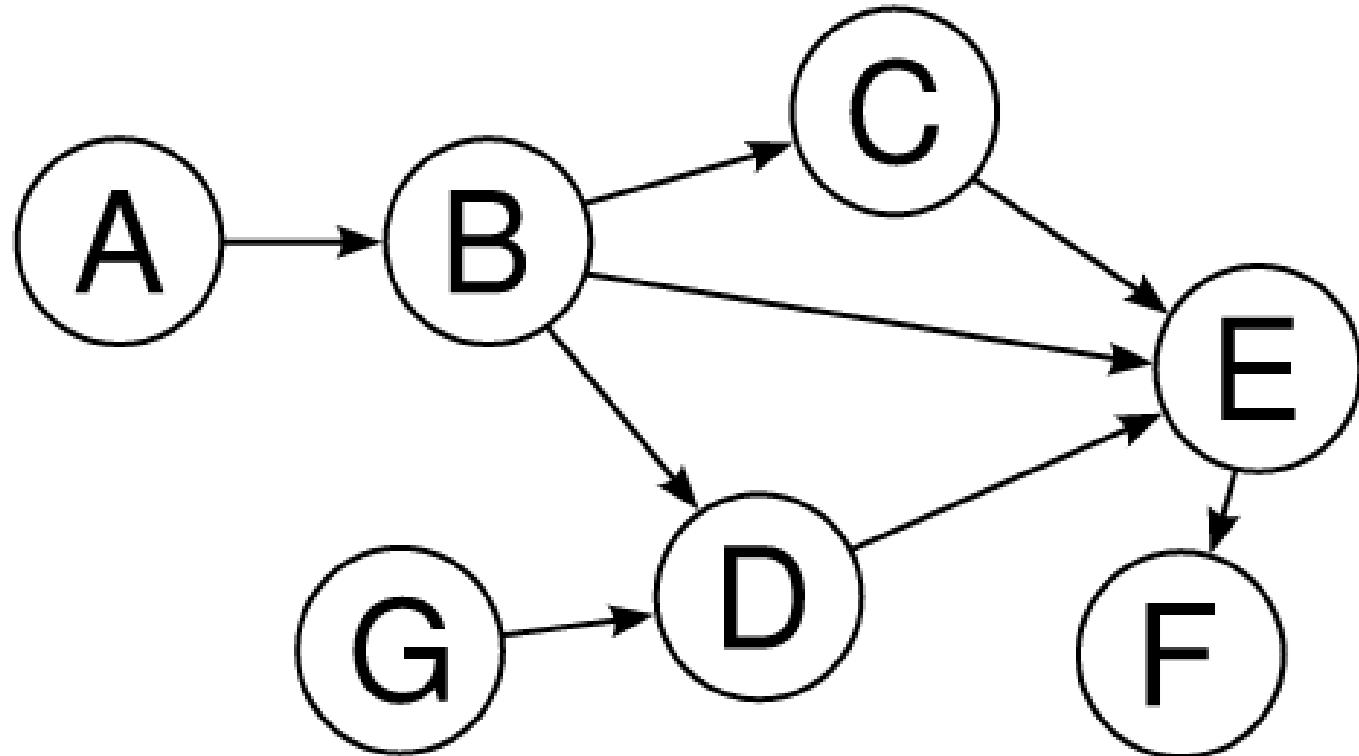
Topological Sorting: is listing vertices of a directed graph in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.

A digraph has a topological sorting iff it is a **dag**.



Finding a **Topological Sorting** of the vertices of a dag:

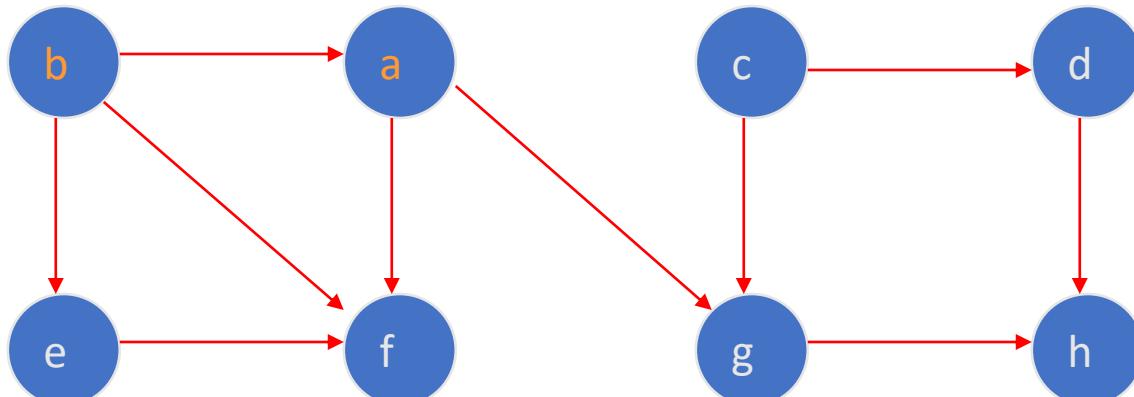
- **DFS-based** algorithm
- **Source-removal** algorithm



DFS-based algorithm for topological sorting

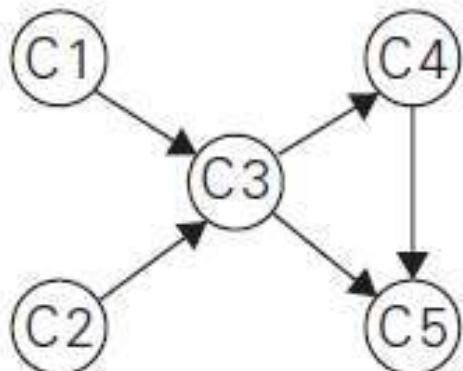
- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered? → NOT a dag!

Example:



Efficiency: The same as that of DFS.

DFS-based algorithm for finding Topological Sorting



(a)

C₅₁
C₄₂
C₃₃
C₁₄ C₂₅

(b)

The popping-off order:
C₅, C₄, C₃, C₁, C₂
The topologically sorted list:
C₂ → C₁ → C₃ → C₄ → C₅

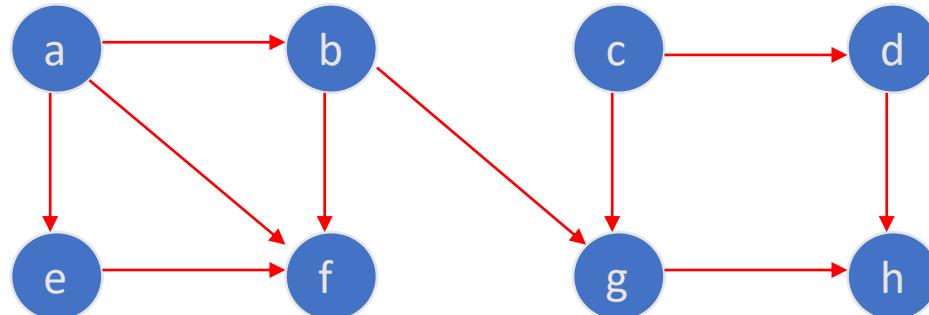
(c)

Decrease and Conquer

Source removal algorithm

Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left or there is no source among the remaining vertices (not a dag)

Example:



“Invert” the adjacency lists for each vertex to count the number of incoming edges by going thru each adjacency list and counting the number of times that each vertex appears in these lists. To remove a source, decrement the count of each of its neighbors by one.

Algorithm SourceRemoval_Toposort(V, E)

L \leftarrow Empty list that will contain the sorted vertices

S \leftarrow Set of all vertices with no incoming edges

while S is non-empty **do**

remove a vertex v from S

add v to *tail* of L

for each vertex m with an edge e from v to m **do**

remove edge e from the graph

if m has no other incoming edges **then**

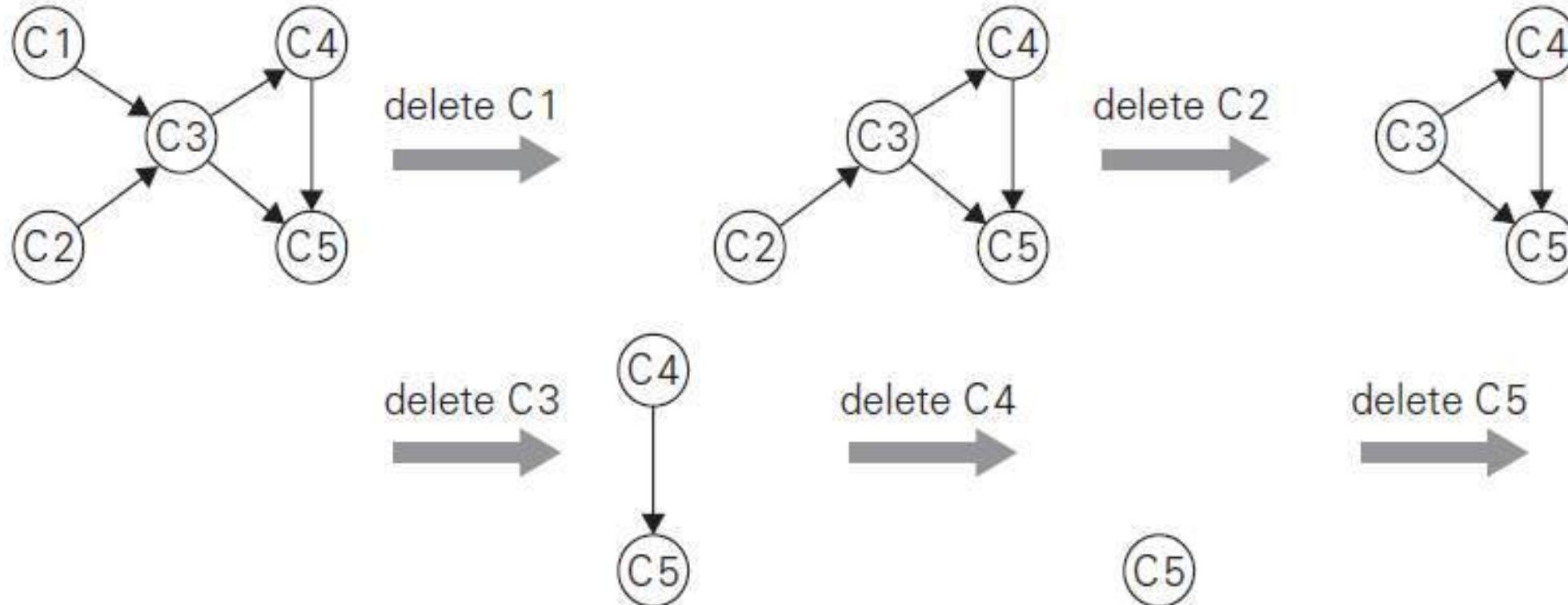
insert m into S

if graph has edges **then**

return error (not a DAG)

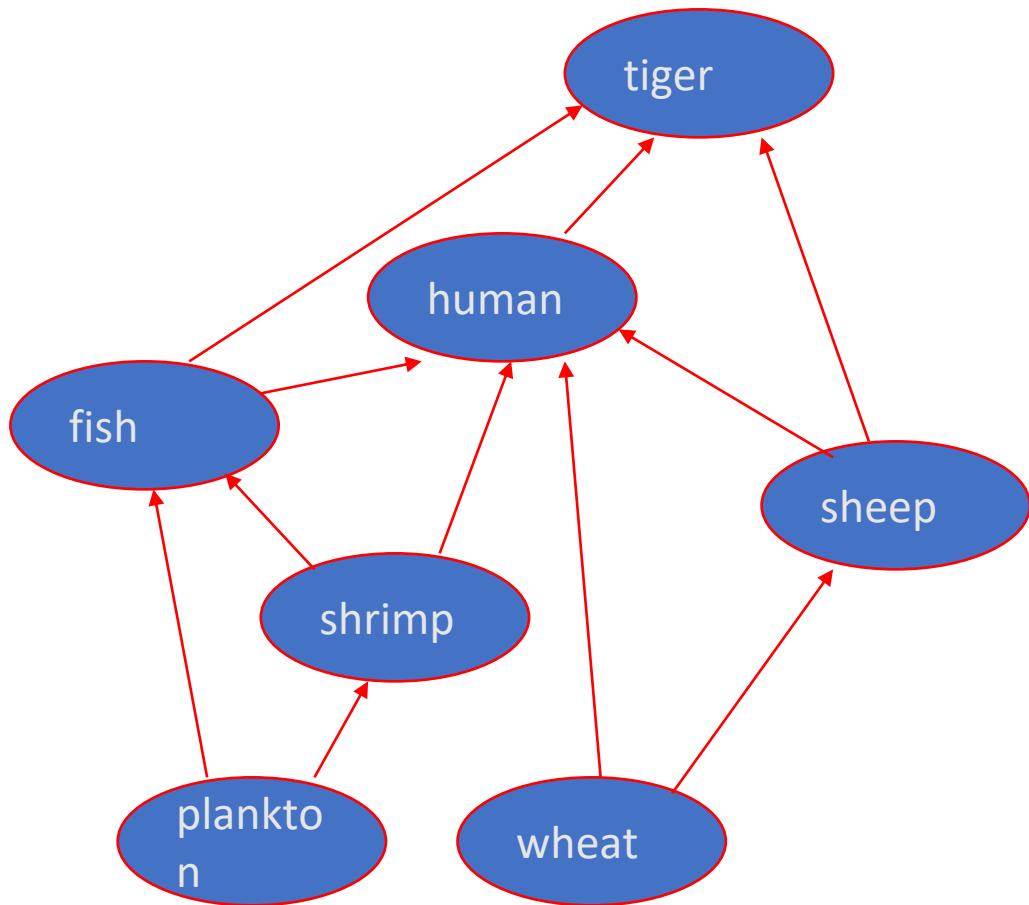
else return L (a topologically sorted order)

Decrease and Conquer



The solution obtained is C1, C2, C3, C4, C5

Order the following items in a food chain



Combinatorial Objects

- Permutations
- Combinations
- Subsets of a given set

Generating Permutations

- Underlying set elements are to be permuted
- Decrease and conquer approach
- Satisfies the minimal change requirement
- Example: Johnson- Trotter algorithm

Generating Permutations

ALGORITHM *JohnsonTrotter(n)*

//Implements Johnson-Trotter algorithm for generating permutations

//Input: A positive integer n

//Output: A list of all permutations of $\{1, \dots, n\}$

initialize the first permutation with $\overset{\leftarrow}{1} \overset{\leftarrow}{2} \dots \overset{\leftarrow}{n}$

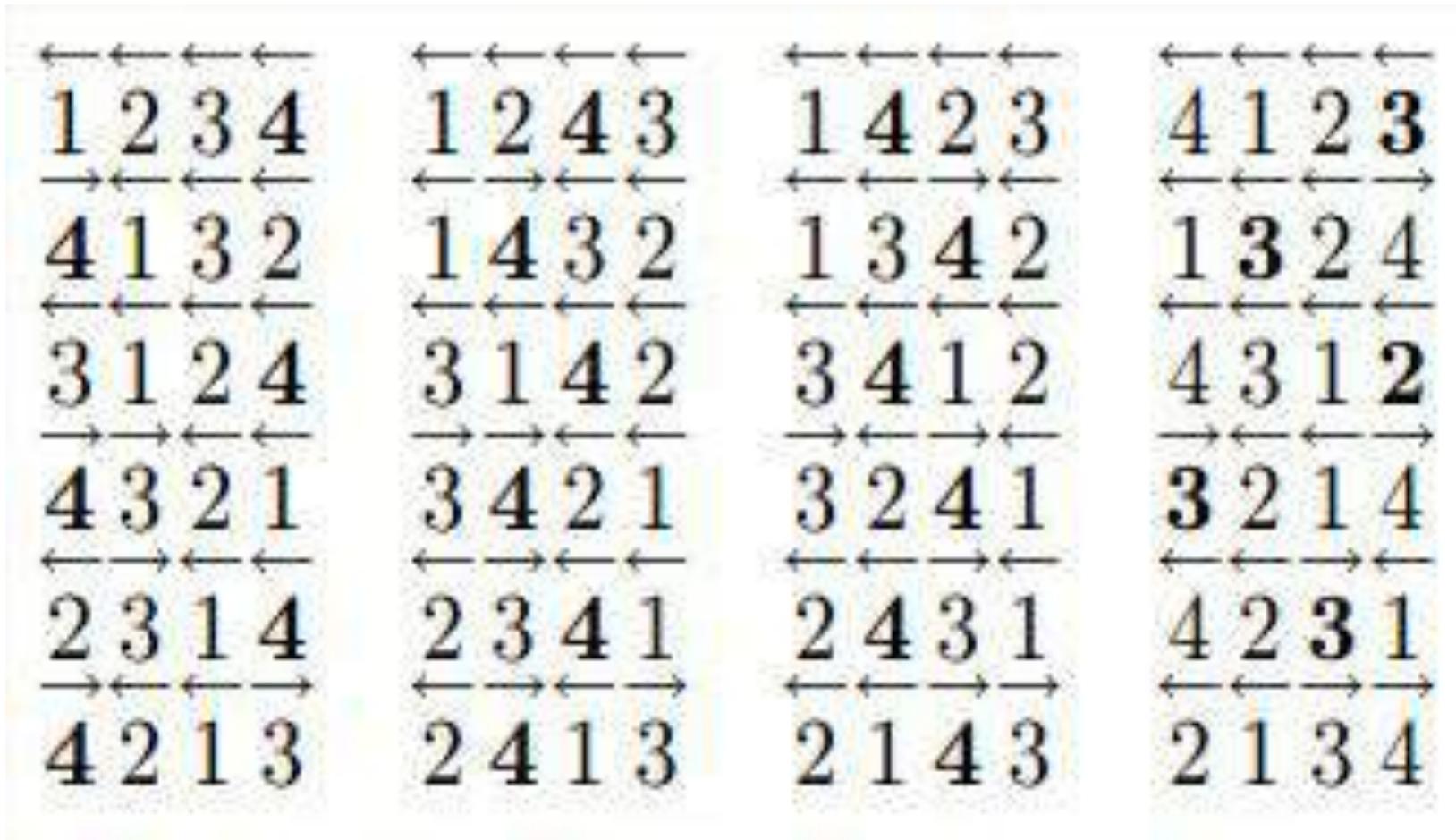
while the last permutation has a mobile element **do**

 find its largest mobile element k

 swap k with the adjacent element k 's arrow points to

 reverse the direction of all the elements that are larger than k

 add the new permutation to the list



ALGORITHM *LexicographicPermute(n)*

//Generates permutations in lexicographic order

//Input: A positive integer n

//Output: A list of all permutations of $\{1, \dots, n\}$ in lexicographic order

initialize the first permutation with $12\dots n$

while last permutation has two consecutive elements in increasing order **do**

 let i be its largest index such that $a_i < a_{i+1}$ // $a_{i+1} > a_{i+2} > \dots > a_n$

 find the largest index j such that $a_i < a_j$ // $j \geq i + 1$ since $a_i < a_{i+1}$

 swap a_i with a_j // $a_{i+1}a_{i+2}\dots a_n$ will remain in decreasing order

 reverse the order of the elements from a_{i+1} to a_n inclusive

 add the new permutation to the list

Generating Subsets:

Knapsack problem needed to find the most valuable subset of items that fits a knapsack of a given capacity.

Powerset: set of all subsets of a set. Set $A=\{1, 2, \dots, n\}$ has 2^n subsets.

Generate all subsets of the set $A=\{1, 2, \dots, n\}$.

Any **decrease-by-one** idea?

of subsets of $\{\}$ = $2^0 = 1$, which is $\{\}$ itself

Suppose, we know how to generate all subsets of $\{1, 2, \dots, n-1\}$

Now, how can we generate all subsets of $\{1, 2, \dots, n\}$?

Generating Subsets:

All subsets of $\{1, 2, \dots, n-1\}$: 2^{n-1} such subsets

All subsets of $\{1, 2, \dots, n\}$:

2^{n-1} subsets of $\{1, 2, \dots, n-1\}$ and
another 2^{n-1} subsets of $\{1, 2, \dots, n-1\}$ having 'n' with them.

That adds up to all 2^n subsets of $\{1, 2, \dots, n\}$

0	\emptyset							
1	\emptyset	$\{a_1\}$						
2	\emptyset	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	\emptyset	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

Alternate way of Generating Subsets:

Knowing the binary nature of either having n th element or not, any idea involving binary numbers itself?

One-to-one correspondence between all 2^n bit strings $b_1b_2\dots b_n$ and 2^n subsets of $\{a_1, a_2, \dots, a_n\}$.

Each bit string $b_1b_2\dots b_n$ could correspond to a subset.

In a bit string $b_1b_2\dots b_n$, depending on whether b_i is 1 or 0, a_i is in the subset or not in the subset.

000	001	010	011	100	101	110	111
\emptyset	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Generating Subsets in Squashed order:

Squashed order: any subset involving a_j can be listed only after all the subsets involving a_1, a_2, \dots, a_{j-1}

Both of the previous methods does generate subsets in squashed order.

000	001	010	011	100	101	110	111
\emptyset	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Generating Subsets in Squashed order:

Squashed order: any subset involving a_j can be listed only after all the subsets involving a_1, a_2, \dots, a_{j-1}

Can we do it with minimal change in bit-string (actually, just one-bit change to get the next bit string)? This would mean, to get a new subset, just change one item (remove one item or add one item).

Binary reflected gray code:

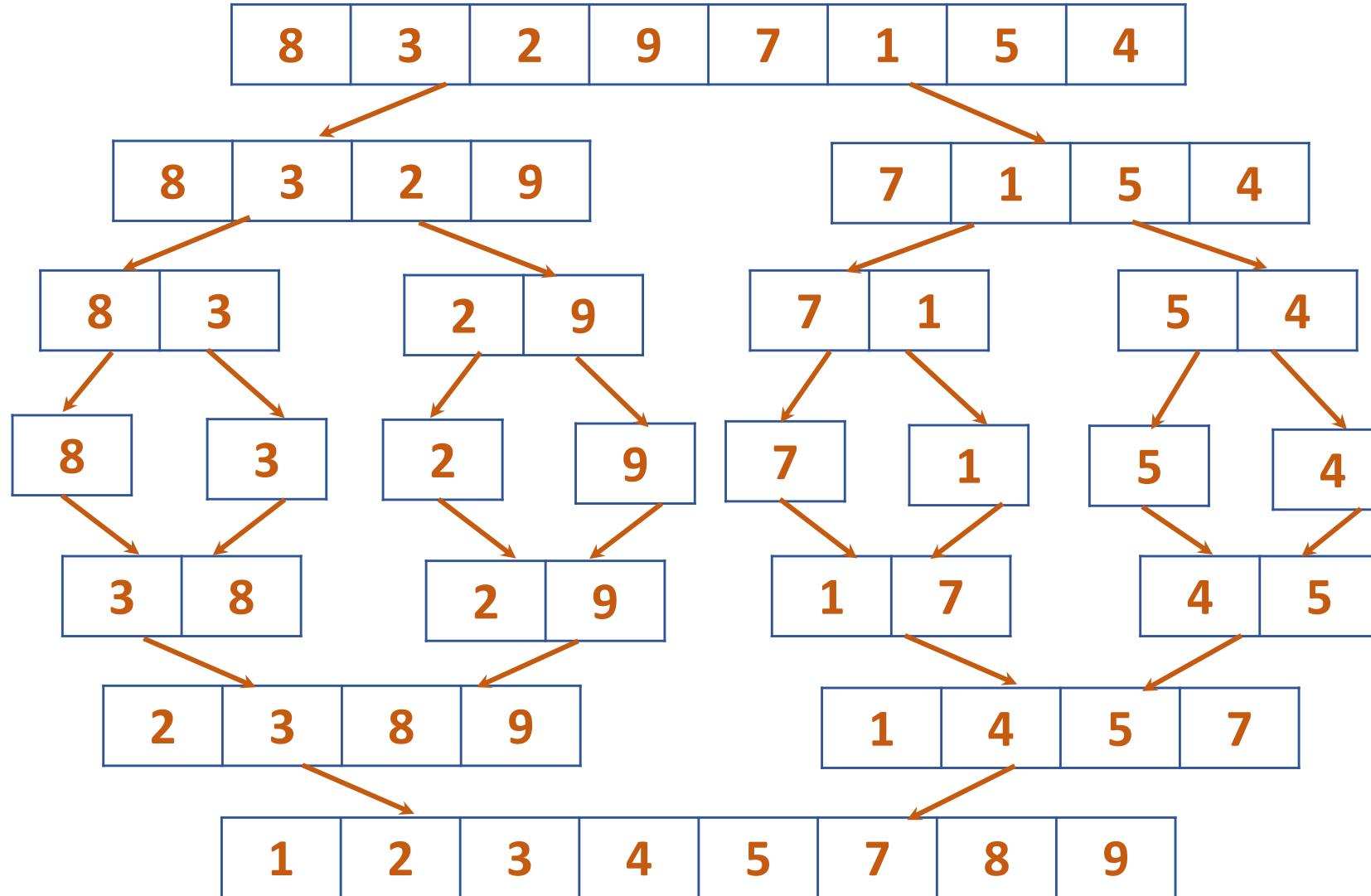
000 001 011 010 110 111 101 100

- Split array A[0..n-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
 - Repeat the following until no elements remain in one of the arrays:
 - compare the first elements in the remaining unprocessed portions of the arrays
 - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
 - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A

```
ALGORITHM Mergesort(A[0 .. n-1])
//Sorts array A[0 .. n-1] by recursive mergesort
//Input: An array A[0 .. n-1] of orderable elements
//Output: Array A[0 .. n-1] sorted in non decreasing order
if n > 1
    copy A[0 .. ⌊n/2⌋ -1 ] to B[0 .. ⌊n/2⌋ -1]
    copy A[ ⌊n/2⌋ .. n -1 ] to C[0..⌊n/2⌋-1]
    Mergesort(B[0 .. ⌊n/2⌋ - 1])
    Mergesort(C[0 .. ⌊n/2⌋ -1])
    Merge(B, C, A)
```

```
ALGORITHM Merge(B[0 .. p- 1], C[0 .. q -1], A[0 .. p + q -1])
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0 .. p -1] and C[0 .. q -1] both sorted
//Output: Sorted array A[0 .. p + q -1] of the elements of B and
C
i<-0; j<-0; k<-0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] <-B[i]; i <-i + 1
    else A[k]<-C[j]; j<-j+1
    k<-K+1
if i = p
    copy C[j .. q-1] to A[k .. p + q - 1]
else
    copy B[i .. p -1] to A[k .. p + q -1]
```

Merge Sort - Example



- Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons $C(n)$ is:

$$C(n) = 2C(n/2) + C_{\text{merge}}(n) \quad [\text{for } n > 1], \quad C(1) = 0$$

- The number of key comparisons performed during the merging stage in the worst case is:

$$C_{\text{merge}}(n) = n - 1$$

- Using the above equation:

$$C_{\text{worst}}(n) = 2C_{\text{worst}}(n/2) + n - 1 \quad [\text{for } n > 1], \quad C_{\text{worst}}(1) = 0$$

- Applying Master Theorem to the above equation:

$$C_{\text{worst}}(n) \in \Theta(n \log n)$$

DESIGN AND ANALYSIS OF ALGORITHMS

Binary Tree

- A *binary tree T* is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called as the left and right subtree of the root
- The definition itself divides the Binary Tree into two smaller structures and hence many problems concerning the binary trees can be solved using the Divide – And – Conquer technique
- The binary tree is a Divide – And – Conquer ready structure ☺



DESIGN AND ANALYSIS OF ALGORITHMS

Height of a Binary Tree

- Height of a Binary Tree: Length of the longest path from root to leaf

ALGORITHM Height(T)

//Computes recursively the height of a binary tree

//Input: A binary tree T

//Output: The height of T

if $T = \emptyset$ return -1

else return max(Height(T_L), Height(T_R)) + 1



DESIGN AND ANALYSIS OF ALGORITHMS

Height of a Binary Tree - Analysis

- The measure of input's size is the number of nodes in the given binary tree. Let us represent this number as $n(T)$
- Basic Operation: Addition
- The recurrence relation is setup as follows:

$$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1, \text{ for } n(T) > 0$$

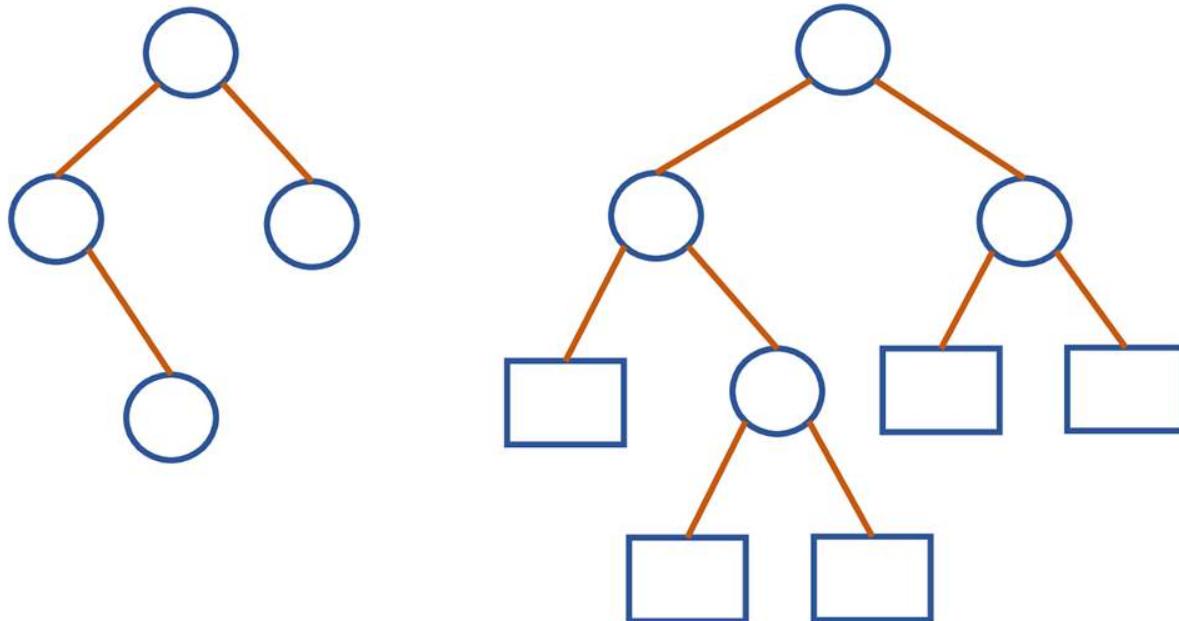
$$A(0) = 0$$



DESIGN AND ANALYSIS OF ALGORITHMS

Height of a Binary Tree - Analysis

- In the analysis of tree algorithms, the tree is extended by replacing empty subtrees by special nodes called external nodes



DESIGN AND ANALYSIS OF ALGORITHMS

Height of a Binary Tree - Analysis

- x – Number of external nodes
- n – Number of internal nodes

$$x = n + 1$$

- The number of comparisons to check whether a tree is empty or not:

$$C(n) = n + x = 2n + 1$$

- The number of additions is:

$$A(n) = n$$



DESIGN AND ANALYSIS OF ALGORITHMS

Binary Tree Traversals

- The three classic traversals for a binary tree are inorder, preorder and postorder traversals
- In the preorder traversal, the root is visited before the left and right subtrees are visited (in that order)
- In the inorder traversal, the root is visited after visiting its left subtree but before visiting the right subtree
- In the postorder traversal, the root is visited after visiting the left and right subtrees (in that order)



DESIGN AND ANALYSIS OF ALGORITHMS

Binary Tree Traversals

Algorithm Inorder(T)

if $T \neq \emptyset$

 Inorder(T_{left})

 print(root of T)

 Inorder(T_{right})

Algorithm Preorder(T)

if $T \neq \emptyset$

 print(root of T)

 Preorder(T_{left})

 Preorder(T_{right})

Algorithm Postorder(T)

if $T \neq \emptyset$

 Postorder(T_{left})

 Postorder(T_{right})

 print(root of T)



OPERATING SYSTEMS

Introduction, Computer System Organization

Suresh Jamadagni

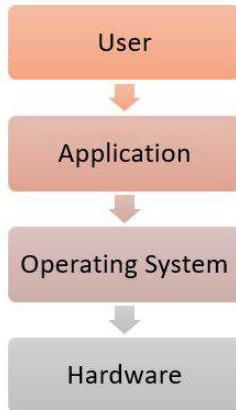
Department of Computer Science

OPERATING SYSTEMS

General Definition



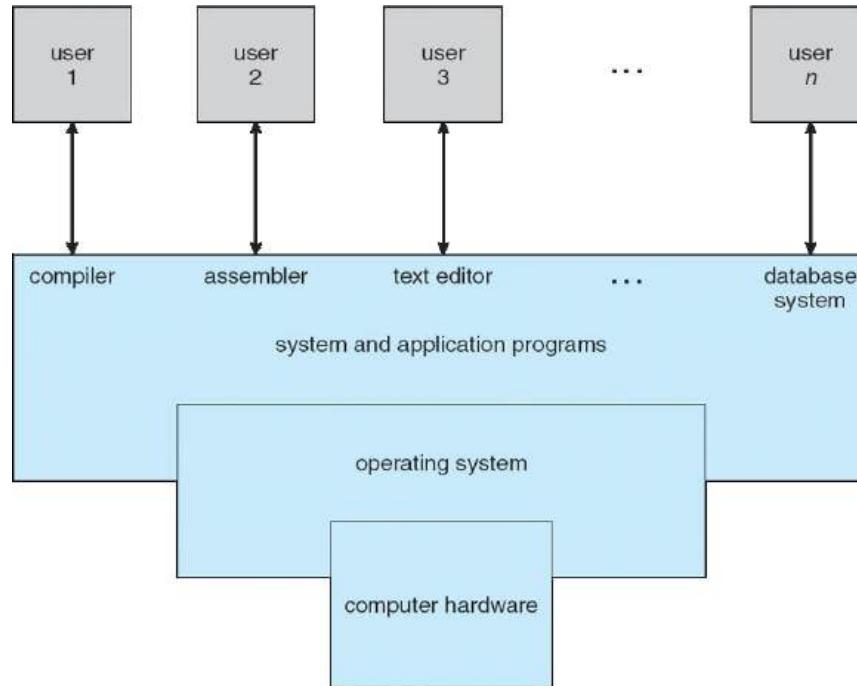
- An Operating System is a program that acts as an **intermediary** between a user of a computer and the computer hardware
- It provides a user-friendly environment in which a user may easily develop and execute programs. Otherwise, hardware knowledge would be mandatory for computer programming.



Operating System Goals

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner
- Manage resources such as
 - Memory
 - Processor(s)
 - I/O Devices

Four Components of a Computer System (Abstract view)



- Hardware – provides basic computing resources
 - 4 CPU, memory, I/O devices
- Operating system
 - 4 Controls and coordinates use of hardware among various applications and users
- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
 - 4 Word processors, compilers, web browsers, database systems, video games
- Users
 - 4 People, machines, other computers

What Operating Systems Do

- Depends on the point of view user and system
- Users want convenience, **ease of use** and **good performance**
 - Don't care about **resource utilization**
- But shared computer such as **mainframe** or **minicomputer** must keep all users happy.
 - Maximize resource utilization.
 - Available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share

What Operating Systems Do

- Users of dedicated systems such as workstations have dedicated resources but frequently use shared resources from servers.
 - resources like file, compute, and print servers are shared.
 - operating system is designed to compromise between individual usability and resource utilization
- Handheld computers are resource poor, optimized for usability and battery life.
- Some computers have little or no user interface, such as embedded computers in devices and automobiles

- OS is a **resource allocator**
 - Manages all resources
 - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
 - Controls execution of programs to prevent errors and improper use of the computer

Defining Operating System

- The OS has many roles and functions
- The fundamental goal of computer systems is to execute user programs and to make solving user problems easier.
- The common functions of controlling and allocating resources are then brought together into one piece of software: the **operating system**
- “The one program running at all times on the computer” is the **kernel**.
- Everything else is either
 - a system program (ships with the operating system) , or
 - an application program.

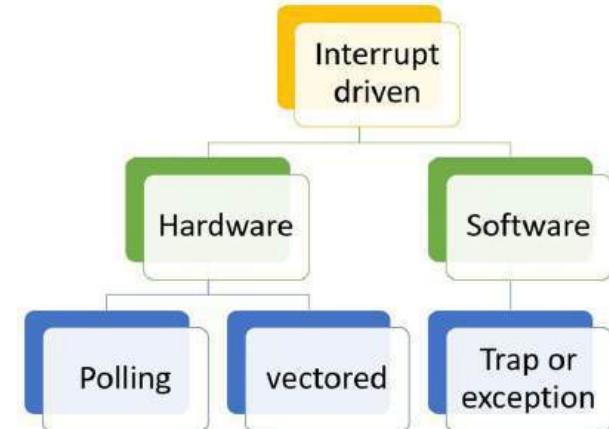
- Computer-system consists of,
 - One or more CPUs, device controllers connect through common bus providing access to shared memory
 - The CPU and the device controllers can execute concurrently, competing for memory cycles.
 - memory controller is provided to synchronize access to the memory.

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- Each device controller has registers for action (like “read character from keyboard”) to take
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an [interrupt](#)

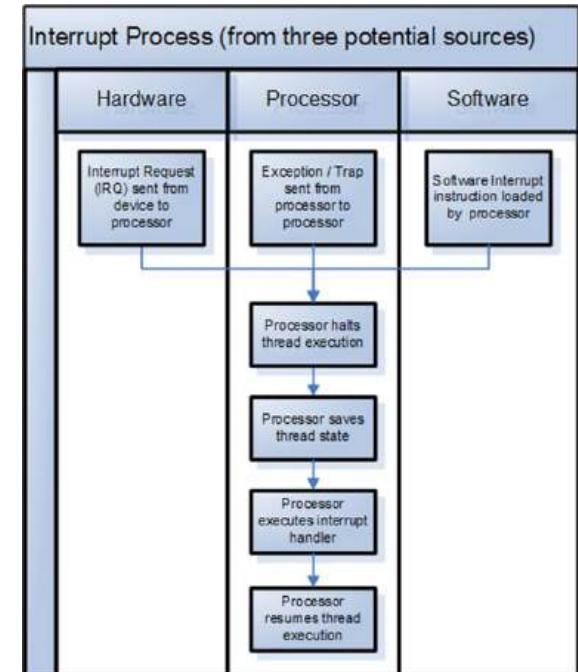
- When the system is booted, the first program that starts running is a Bootstrap.
- It is stored in read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM).
- Bootstrap is known by the general term firmware, within the computer hardware.
- It initializes all aspects of the system, from CPU registers to device controllers to memory contents.
- The bootstrap program must know how to load the operating system and how to start executing that system.
- The bootstrap program must locate and load into memory the operating system kernel.
- The first program that is created is **init**, after the OS is booted. It waits for the occurrence of event.

Common Functions of Interrupts

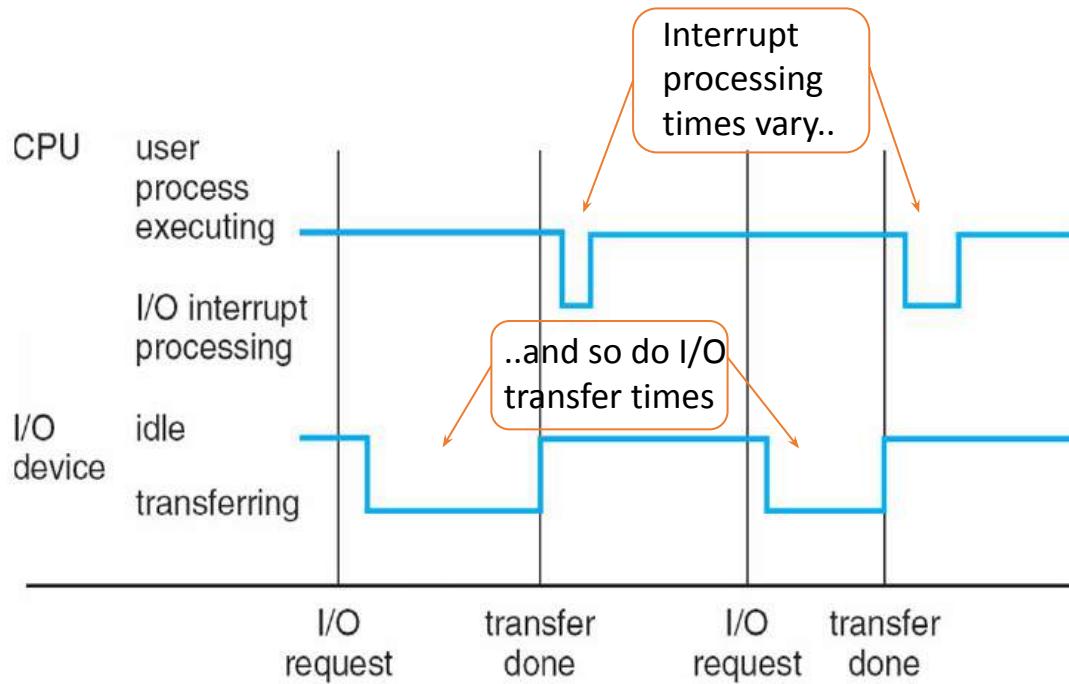
- An operating system is **interrupt driven**
- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request



- The operating system saves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
 - polling
 - vectored interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

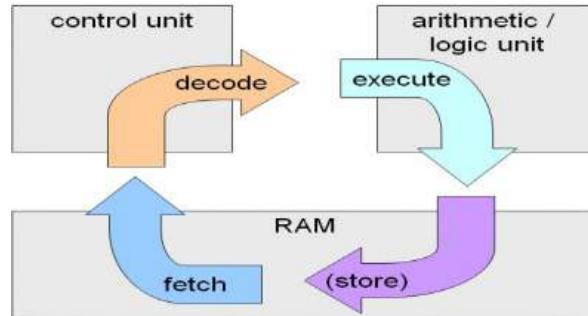


Interrupt Timeline for a single process doing output



- Main memory – only large storage media that the CPU can access directly (**Random access memory** and typically **volatile**)
 - Implemented with semiconductor technology called DRAM
- Computers use other forms of memory like ROM, EEPROM
- Smart phones have EEPROM to store factory installed programs.

- Typical instruction execution

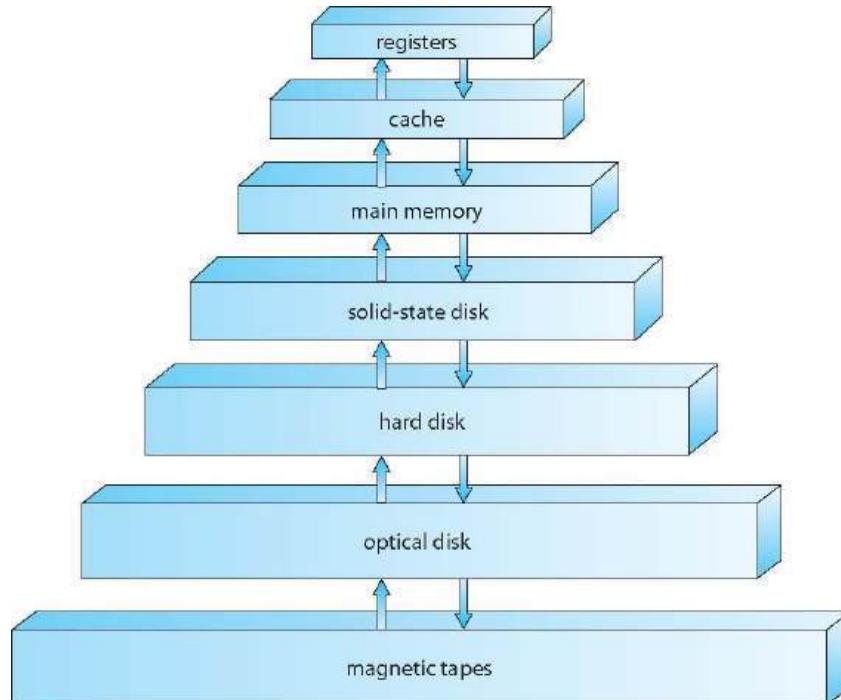


- The processor **fetches** instructions from memory, **decodes** and **executes** them.
- The Fetch, Decode and Execute cycles are repeated until the program terminates.

- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
- Hard disks – rigid metal or glass platters covered with magnetic recording material
 - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
 - The **disk controller** determines the logical interaction between the device and the computer
- **Solid-state disks** – faster than hard disks, nonvolatile
 - Various technologies and becoming more popular
 - Flash memory used in camera's PDA's

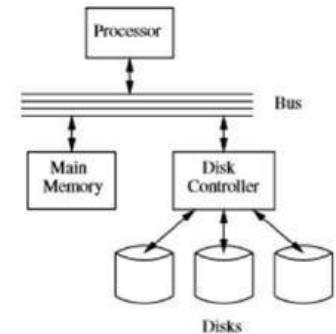
Storage Hierarchy

- Storage systems organized in hierarchy
 - Speed
 - Cost
 - Volatility
- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage



- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
 - If it is, information used directly from the cache (fast)
 - If not, data copied to cache and used there
- Cache smaller than storage being cached
 - Cache management important design problem
 - Cache size and replacement policy

- Storage is a type of I/O device
- A large portion of operating-system code is dedicated to managing I/O
 - As reliability and performance of a system is the main concern.
- General-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus.
- Each device controller is in charge of a specific type of device.
- **Device Driver** for each device controller to manage I/O
 - Provides uniform interface between controller and kernel
- **Small Computer-Systems Interface (SCSI)** controller enables to connect more devices.



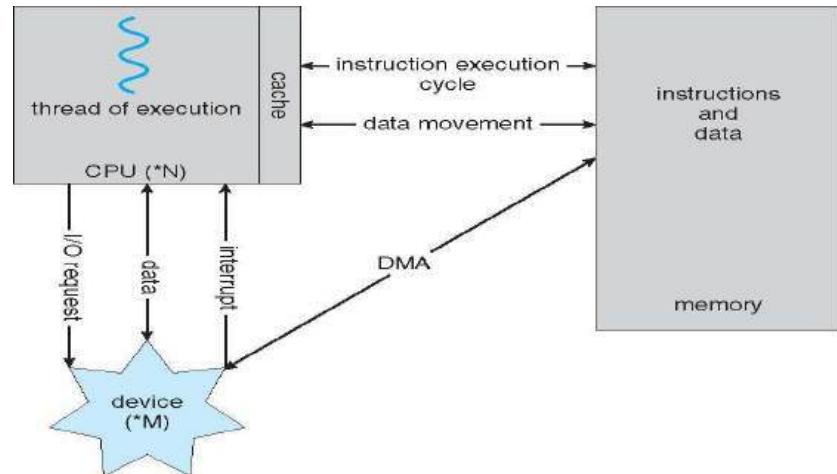
- A device controller maintains some local buffer storage and a set of special-purpose registers.
- The device controller is responsible for moving the data between the peripheral devices.

I/O Structure

- After I/O starts, control returns to user program only upon I/O completion
 - Wait instruction idles the CPU until the next interrupt
 - Wait loop (contention for memory access)
 - At most one I/O request is outstanding at a time, no simultaneous I/O processing
- After I/O starts, control returns to user program without waiting for I/O completion
 - **System call** – request to the OS to allow user to wait for I/O completion
 - **Device-status table** contains entry for each I/O device indicating its type, address, and state
 - OS indexes into I/O device table to determine device status and to modify table entry to include interrupt.

Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

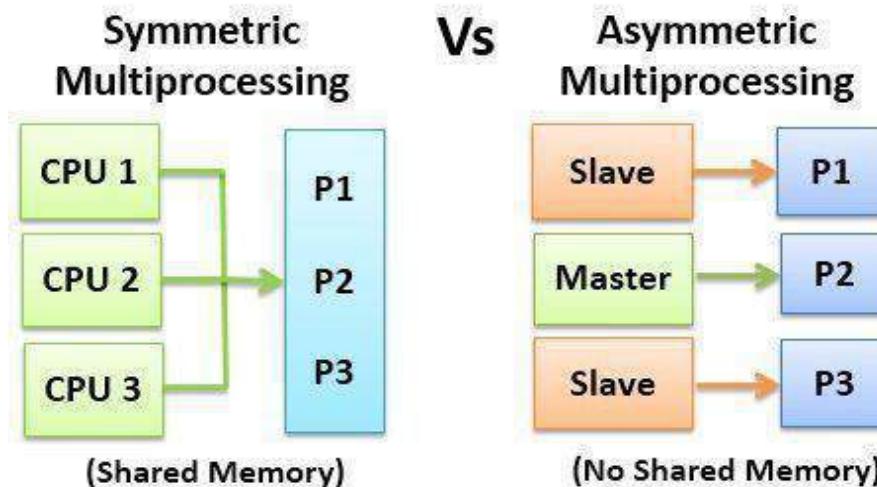


- Most systems use a single general-purpose processor
 - Most systems have other special-purpose processors as well.
 - Device specific processors like disk, keyboard, graphic controller
 - Special-purpose processors run a limited number of instructions
 - Special-purpose processors are low-level components built into the hardware
 - Managed by OS.
 - OS monitors the status.
- Example Disk controller microprocessor
 - Receives sequence of requests from CPU.
 - Implements its own disk queue and scheduling algorithm
 - Relieves the main CPU of the overhead of disk scheduling.

- **Multiprocessors** systems growing in use and importance
 - Also known as **parallel systems, tightly-coupled systems**
 - Advantages include:
 - **Increased throughput**
 - **Economy of scale**
 - **Increased reliability** – graceful degradation or fault tolerance

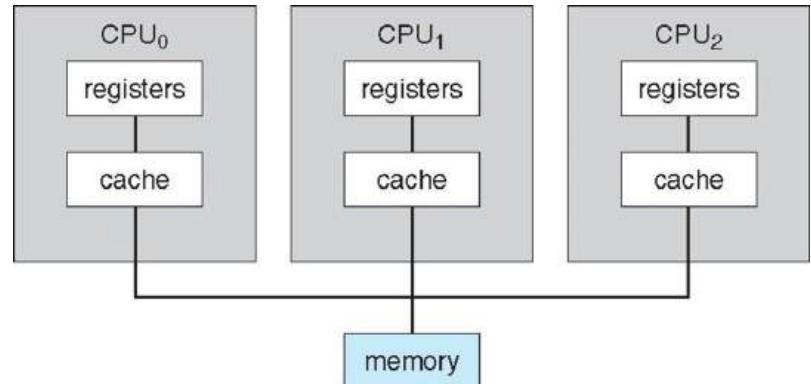
Two types of Multiprocessor Systems

1. **Asymmetric Multiprocessing** – each processor is assigned a specific task.
2. **Symmetric Multiprocessing** – each processor performs all tasks

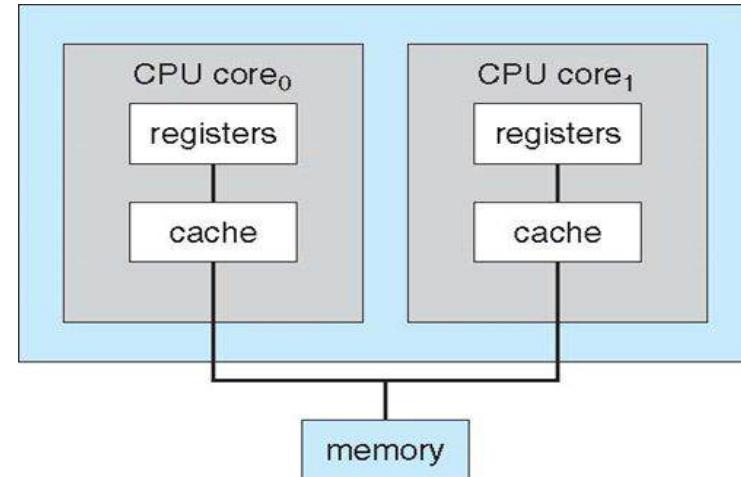


Symmetric Multiprocessing Architecture

- In SMP all processors are peers; no boss-worker relationship exists between processors.
- Each processor has its own set of registers, as well as a private or local cache.
- All processors share physical memory.



- A recent trend in CPU design is to include multiple computing cores on a single chip. Such multiprocessor systems are termed **multicore**.
- More efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication.
- One chip with multiple cores uses significantly less power than multiple single-core chips.



A dual-core design with two cores placed on the same chip

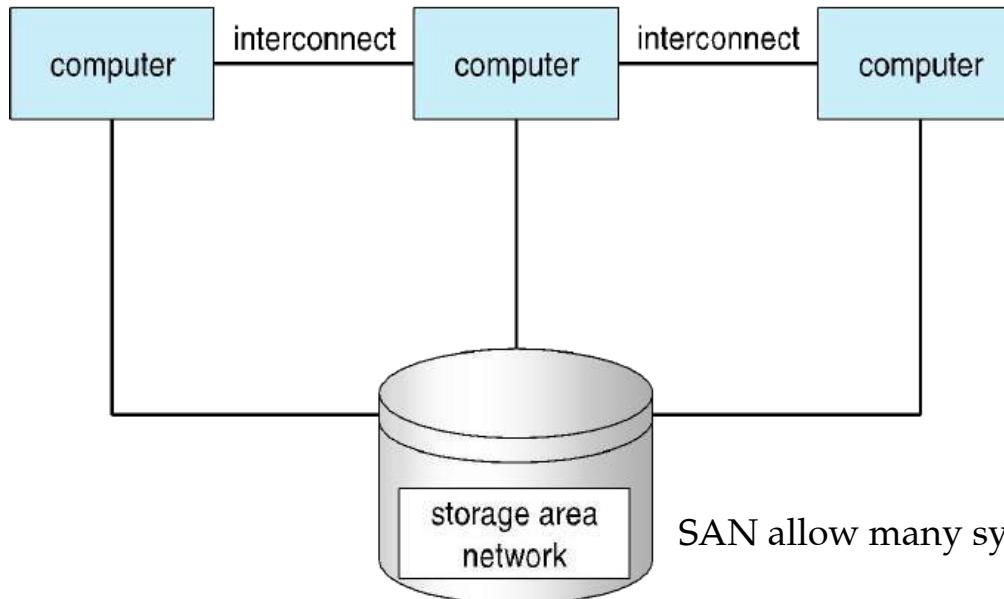
Command to know the number of cores, cache details
\$cat /proc/cpuinfo|more

Blade servers

- **Blade servers** are a recent development in which multiple processor boards, I/Oboards, and networking boards are placed in the same chassis.
 - blade-processor board boots independently and runs its own operating system.
 - Some blade-server boards are multiprocessor as well, which blurs the lines between types of computers.
 - In essence, these servers consist of multiple independent multiprocessor systems.

Clustered Systems

- Like multiprocessor systems, but multiple systems working together
 - Usually sharing storage via a **storage-area network (SAN)**
 - Provides a **high-availability** service which survives failures
 - **Asymmetric clustering** has one machine in hot-standby mode
 - **Symmetric clustering** has multiple nodes running applications, monitoring each other
 - Some clusters are for **high-performance computing (HPC)**
 - Applications must be written to use **parallelization**
 - Some have **distributed lock manager (DLM)** to avoid conflicting operations (Ex: when multiple hosts access the same data on shared storage)

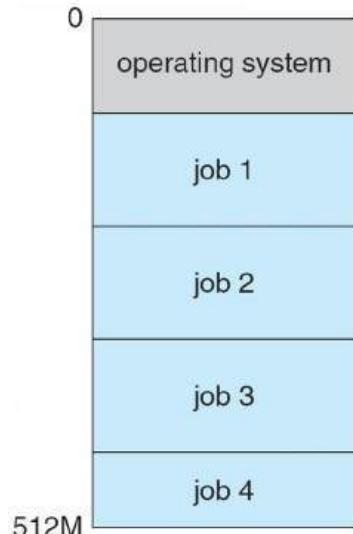


SAN allow many systems to attach to a pool of storage

General structure of a clustered system.

Operating-System Structure - Multiprogramming

- **Multiprogramming (Batch system)** needed for efficiency
 - Single user cannot keep CPU and I/O devices busy at all times
 - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
 - A subset of total jobs in system is kept in memory
 - One job selected and run via **job scheduling**
 - When it has to wait (for I/O for example), OS switches to another job



Operating-System Structure - Multitasking

- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
 - **Response time** should be < 1 second
 - Each user has at least one program executing in memory \square **process**
 - If several jobs ready to run at the same time \square **CPU scheduling**
 - If processes don't fit in memory, **swapping** moves them in and out to run
 - **Virtual memory** allows execution of processes not completely in memory

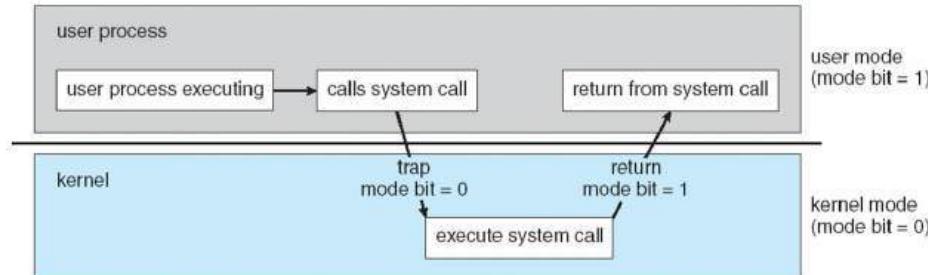
- **Interrupt driven** (hardware and software)
 - Hardware interrupt by one of the devices
 - Software interrupt (**exception** or **trap**):
 - Software error (e.g., division by zero)
 - Request for operating system service
 - Other process problems include infinite loop, processes modifying each other or the operating system

Dual-Mode and Multimode Operation

- **Dual-mode** operation allows OS to protect itself and other system components
 - **User mode** and **kernel mode**
 - **Mode bit** provided by hardware
 - Provides ability to distinguish when system is running user code or kernel code
 - Some instructions designated as **privileged**, only executable in kernel mode
 - System call changes mode to kernel, return from call resets it to user
- Increasingly CPUs support multi-mode operations
 - i.e. **virtual machine manager (VMM)** mode for guest **VMs**

Transition from user to kernel mode

- When a trap or interrupt occurs, hardware switches from user mode to kernel mode (changes the state of the mode bit to 0).
- When the request is fulfilled, the system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.



- Timer to prevent infinite loop / process hogging resources
 - Timer is set to interrupt the computer after a specified period (fixed 1/60 sec or variable 1 msec to 1 sec)
 - A **variable timer** is generally implemented by a fixed-rate clock and a counter.
 - Operating system sets the counter (privileged instruction)
 - Every time the clock ticks, the counter is decremented.
 - When counter reaches zero, an interrupt occurs
 - Timer can be used to prevent a user program from running too long (terminate the program)

Array:

- An array is a simple data structure in which each element can be accessed directly.
- Main Memory constructed with array.
- How the data is accessed?
- Items with multiple bytes are accessed as item number \times item size
- But what about storing an item whose size may vary?
- what about removing an item if the relative positions of the remaining items must be preserved?

- Standard programming data structures are used extensively in OS

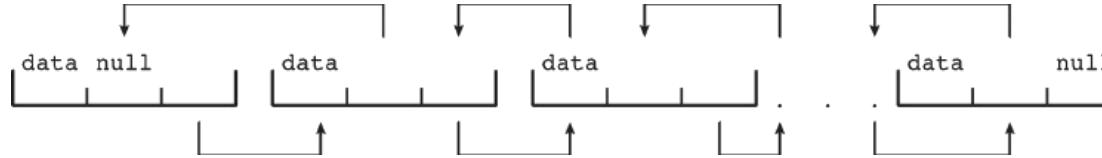
Singly linked list

- The items in a list must be accessed in a **particular order**.
- common method for implementing this structure is a linked list



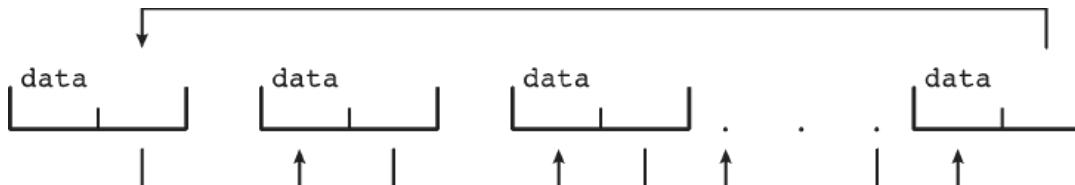
- In a **singly linked list**, each item points to its successor.

Doubly linked list



In a **doubly linked list**, a given item can refer either to its predecessor or to its successor.

Circular linked list



In a **circularly linked list**, the last element in the list refers to the first element, rather than to null.

Advantages:

- Linked lists accommodate items of varying sizes.
- Allow easy insertion and deletion of items

Disadvantages:

- Performance for retrieving a specified item in a list of size n is linear — $O(n)$, as it requires potentially traversing all n elements in the worst case.

Usage:

- Lists are used by some of the kernel algorithms
- Constructing more powerful data structures such as stacks and queues

Stack - a sequentially ordered data structure that uses **LIFO** principle for adding and removing items

- OS often uses a stack when involving function calls.
- Parameters, local variables and the return address are pushed onto the stack when a function is called
- Return from the function call pops those items off the stack

Queue - a sequentially ordered data structure that uses **FIFO** principle for adding and removing items

- Tasks waiting to be run on an available CPU are organized in queues
- Print jobs sent to a printer are printed in the order of submission

- Data structure used to represent data hierarchically.
- Data values in a tree structure are linked through parent–child relationships

Binary search tree

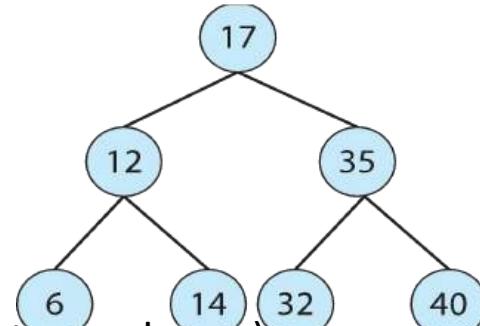
- ordering between 2 children: left \leq right

- Search performance is $O(n)$

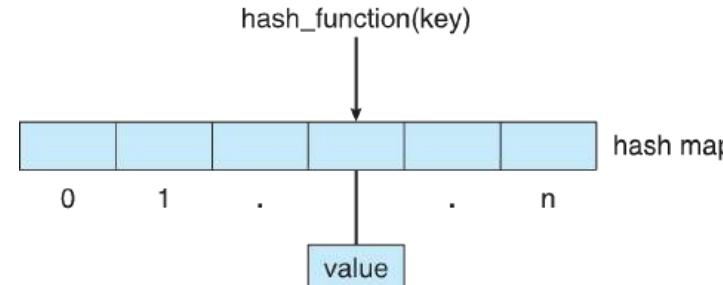
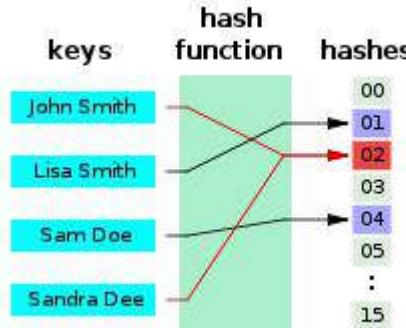
- **Balanced binary search tree –** (a tree containing n items has at most $\log n$ levels)

- Search performance is $O(\log n)$

- Used by Linux for selecting which task to run next (CPU-Scheduling algorithm)



- Hash functions can result in the same output value for 2 inputs
- **Hash function** can be used to implement a **hash map**
 - Maps or associates key:value pairs using a hash function
 - Search performance is $O(1)$

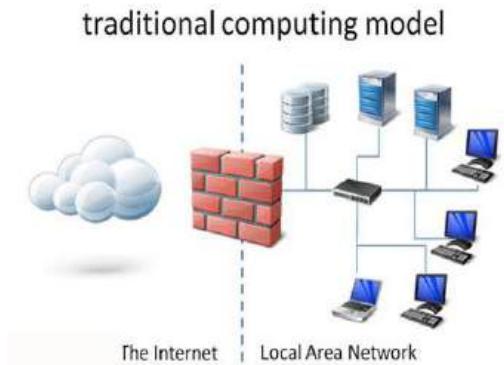


Bitmap - string of n binary digits representing the **status** of n items

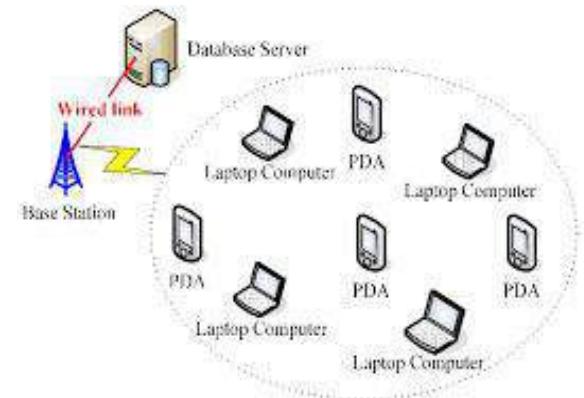
- Availability of each resource is indicated by the value of a binary digit
 - 0 – resource is **available**
 - 1 – resource is **unavailable**
- Value of the i^{th} position in the bitmap is associated with the i^{th} resource
 - Example: bitmap 001011101 shows resources 2, 4, 5, 6, and 8 are unavailable; resources 0, 1, 3, and 7 are available
- Commonly used to represent the availability of a large number of resources (**disk blocks**)

Computing Environments – Traditional

- Stand-alone general purpose machines
- But blurred as most systems interconnect with others (i.e., the Internet)
- **Portals** provide web access to internal systems
- **Network computers (thin clients)** are like Web terminals
- Mobile computers interconnect via **wireless networks**
- Networking becoming ubiquitous – even home systems use **firewalls** to protect home computers from Internet attacks

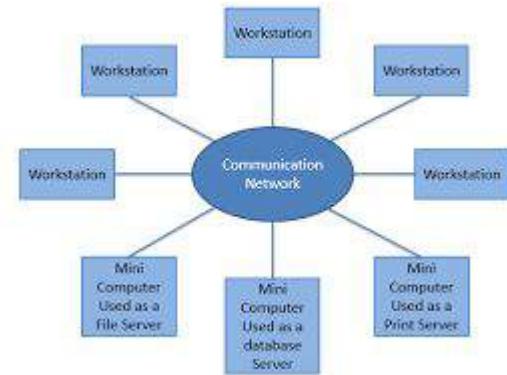


- Handheld smartphones, tablets, etc
- What is the functional difference between them and a “traditional” laptop?
- Extra feature – more OS features (GPS, gyroscope)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders are **Apple iOS** and **Google Android**



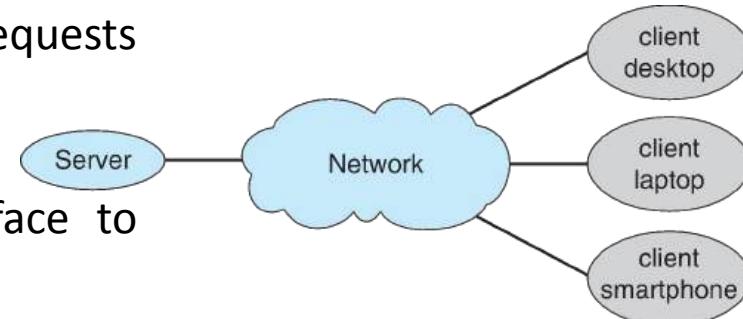
Distributed computing

- Collection of separate, possibly **heterogeneous** systems networked together
 - **Network** is a communications path, **TCP/IP** most common
 - **Local Area Network (LAN)**
 - **Wide Area Network (WAN)**
 - **Metropolitan Area Network (MAN)**
 - **Personal Area Network (PAN)**
- **Network Operating System** provides features between systems across network
 - Communication scheme allows systems to **exchange messages**
 - Illusion of a single system

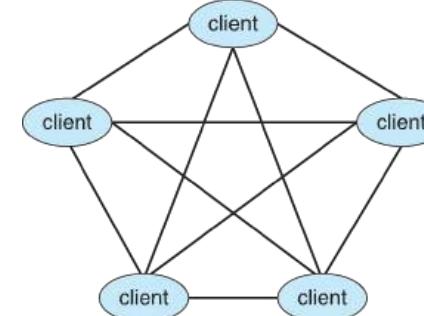


Client-Server Computing

- Dumb terminals replaced by smart PCs
- Many systems now **servers**, responding to requests generated by **clients**
 - **Compute-server system** provides an interface to client to request services (i.e., database)
 - **File-server system** provides interface for clients to store and retrieve files

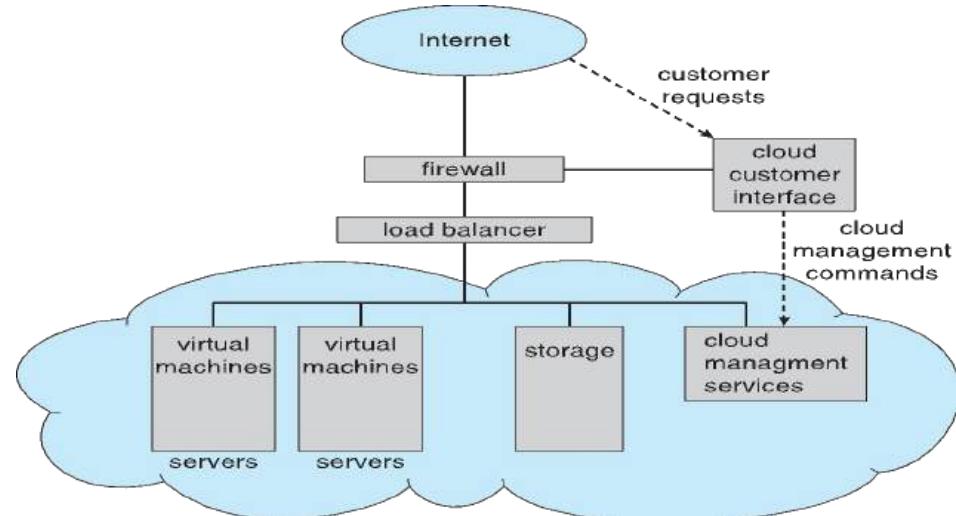


- Another model of distributed system
 - P2P does not distinguish clients and servers
 - Instead all nodes are considered peers
 - May each act as client, server or both
 - Node must join P2P network
 - Registers its service with central lookup service on network, or
 - Broadcast request for service and respond to requests for service via **discovery protocol**
 - Examples include Napster and Gnutella, **Voice over IP (VoIP)** such as Skype



- Allows operating systems to run applications within other OSes
 - Vast and growing industry
- **Emulation** used when source CPU type **different** from target type (i.e. PowerPC to Intel x86)
 - Generally slowest method
 - When computer language not compiled to native code – **Interpretation**
- **Virtualization** – OS natively compiled for CPU, running **guest OSes** also natively compiled
 - Consider VMware running WinXP guests, each running applications, all on native WinXP **host** OS
 - **VMM** (virtual machine Manager) provides virtualization services

- Cloud computing environments composed of traditional OSes, plus VMs, plus cloud management tools
 - Internet connectivity requires security like firewalls
 - Load balancers spread traffic across multiple applications



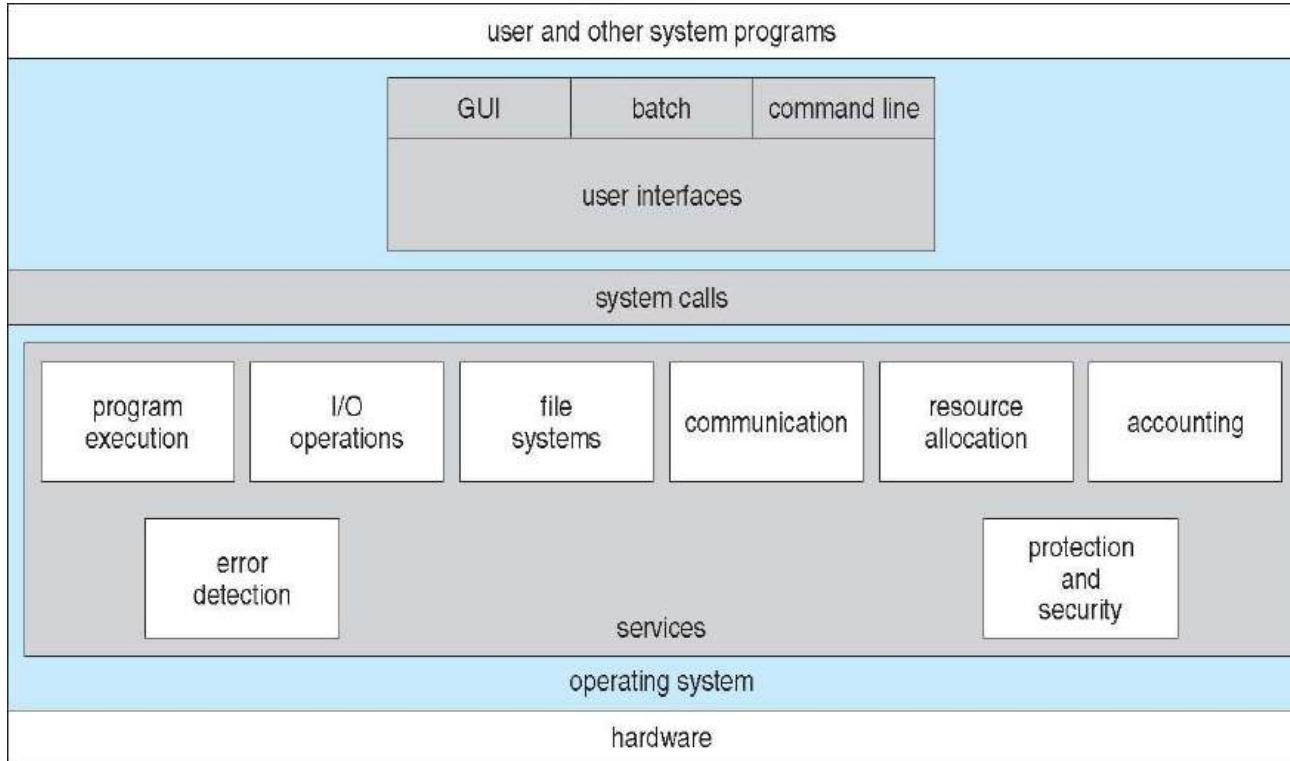
- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for its functionality.
 - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, pay based on usage
- Many types
 - **Public cloud** – available via Internet to anyone willing to pay
 - **Private cloud** – run by a company for the company's own use
 - **Hybrid cloud** – includes both public and private cloud components

- Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
- Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e., a database server)
- Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e., storage available for backup use)

- Real-time embedded systems most prevalent form of computers
 - Vary considerable, special purpose, **limited** purpose OS, **real-time OS**
 - Use expanding
- Many other special computing environments as well
 - Some have OSes, some perform tasks without an OS
- Real-time OS has well-defined **fixed time** constraints
 - Processing **must** be done within constraint
 - Correct operation only if constraints met

OPERATING SYSTEMS

A View of Operating System Services



OPERATING SYSTEMS

Services



- Operating systems provide an environment for execution of programs and services to programs and users
- Set of operating-system services provides functions that are helpful to the user:
 - **User interface** - Almost all operating systems have a user interface (**UI**).
 - **Command-Line (CLI)** - Command interpreters (shells)
 - **Graphics User Interface (GUI)**
 - **Batch**
 - **Program execution** - The system must be able to **load** a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
 - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device

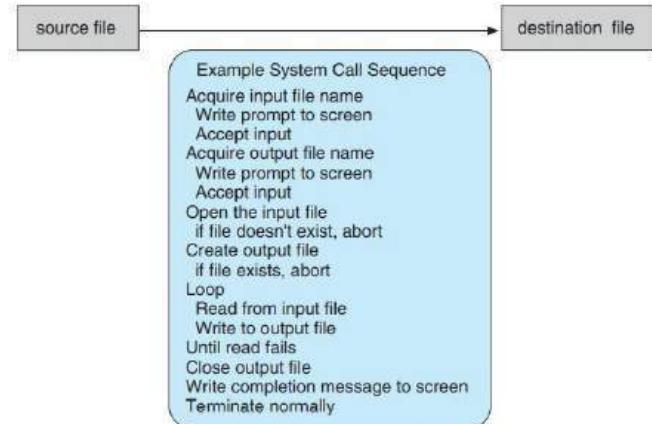
- **File-system manipulation** - The file system is of particular interest. Programs need to **read and write** files and directories, create and delete them, search them, list file information, permission management.
- **Communications** – Processes may exchange information, on the same computer or between computers over a **network**
 - Communications may be via shared memory or through **message passing** (packets moved by the OS)

- **Error detection** – OS needs to be constantly aware of possible errors
 - May occur in the CPU and memory hardware, in I/O devices, in user program
 - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
 - **Debugging** facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
 - **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
 - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
 - **Accounting** - To keep track of which users use how much and what kinds of computer resources

- **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes **should not interfere** with each other
 - **Protection** involves ensuring that all access to system **resources** is controlled
 - **Security** of the system from outsiders requires user **authentication**, extends to defending external I/O devices from invalid access attempts

- **System calls** provide an interface to the services made available by an operating system.
- These calls are generally available as **routines** written in a higher level programming language or assembly-language
- Example: Sequence of system calls used for copying contents from one file to another



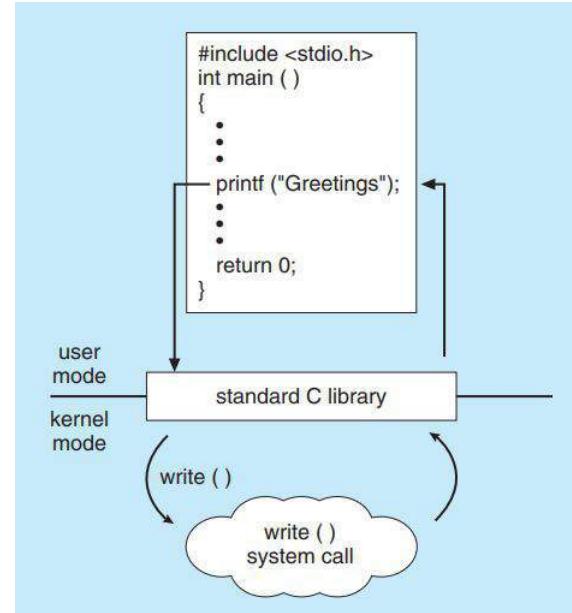
System calls can be grouped roughly into six major categories:

1. Process control
2. File manipulation
3. Device manipulation
4. Information maintenance
5. Communications
6. Protection

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

Example of a system call

- A C program invokes the printf() statement
- The C library intercepts this call and invokes the necessary system call **write()** in the operating system
- The C library takes the value returned by write() and passes it back to the user program

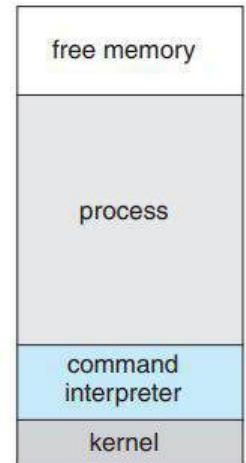


Design goals

- Start the design by defining goals and specifications
- Affected by choice of hardware and the type of system
- Requirements can be categorized as **User** goals and **System** goals
 - User goals – operating system should be **convenient** to use, easy to learn, reliable, safe, and fast
 - System goals – operating system should be easy to design, implement, and **maintain**, as well as flexible, reliable, error-free, and efficient

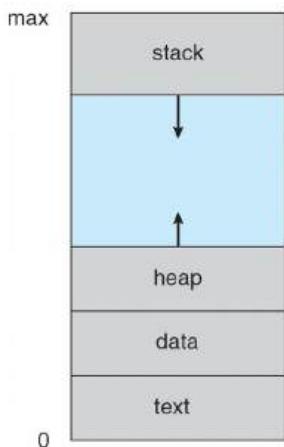
- Important principle to separate
 - Policy:** *What* will be done?
 - Mechanism:** *How* to do it?
- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer to prevent a user program from running too long)
- Specifying and designing an OS is highly creative task of **software engineering**

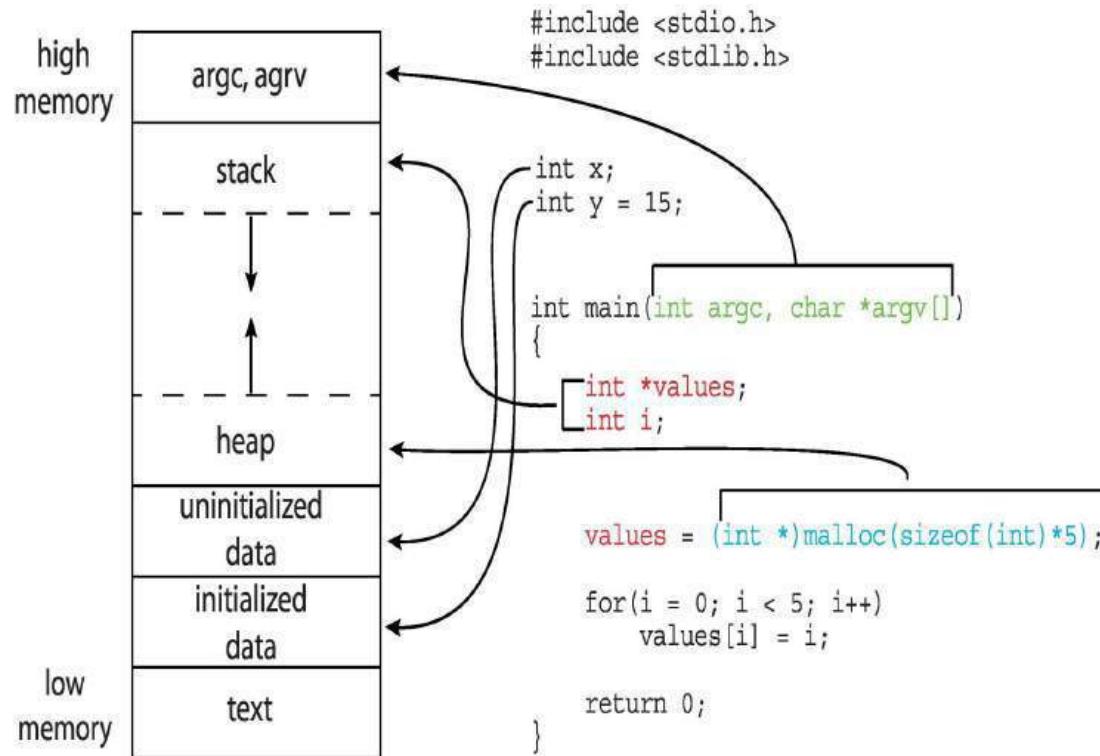
- An operating system executes a variety of programs:
 - Batch system – **jobs**
 - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms **job** and **process** almost interchangeably
- **Process** – a program in execution; process execution must **progress in sequential fashion**
- Program is **passive** entity stored on disk (**executable file**), process is **active**
 - Program becomes process when executable file **loaded** into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
 - Consider multiple users executing the same program



Structure of a process in memory

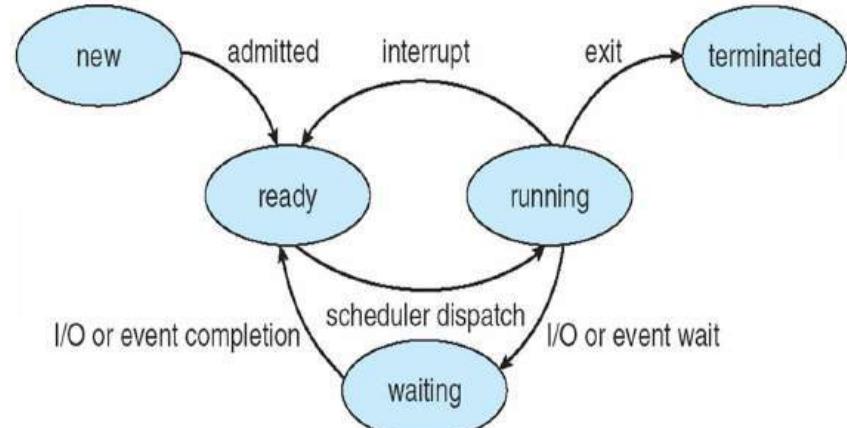
- The program code, also called **text section**.
 - Includes current activity including **program counter**, processor registers
- **Stack** containing **temporary** data
 - Function parameters, return addresses, local variables
- **Data section** containing **global** variables
- **Heap** containing memory dynamically allocated during run time



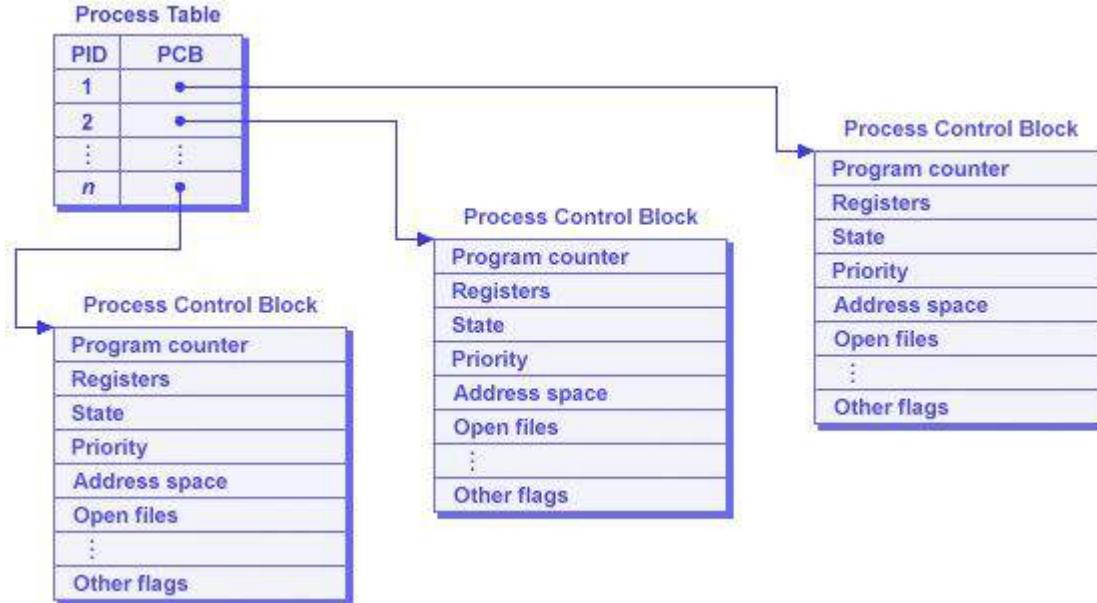


As a process executes, it changes **state**

- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution



Process Control Block (PCB)



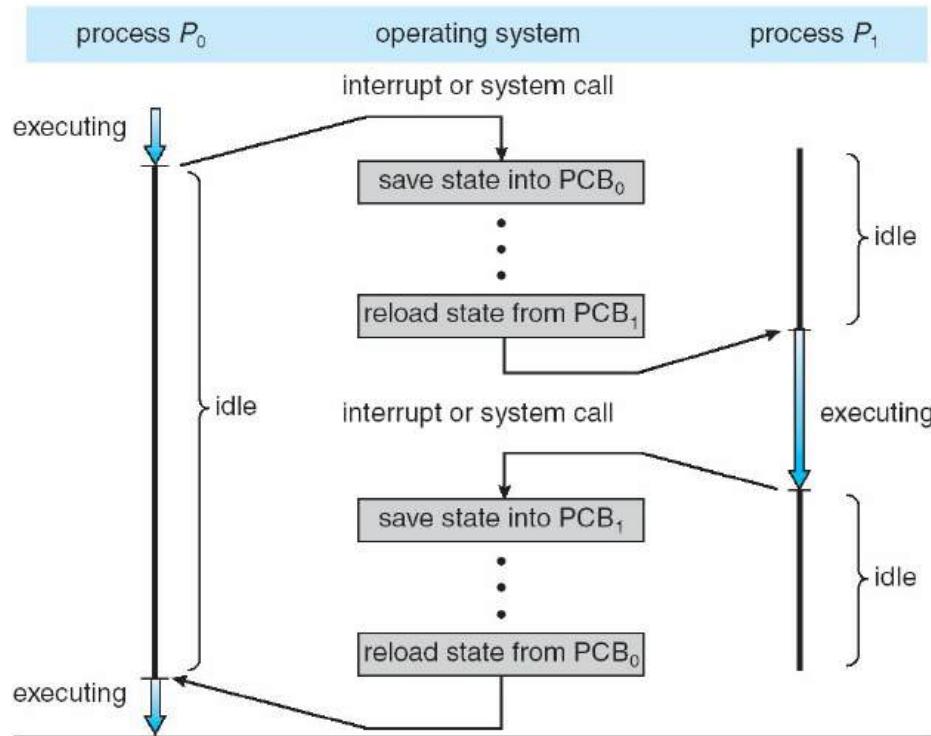
Process Control Block (PCB)

Each process is represented in the operating system by a Process Control Block (also called **task control block**)

- Process **state** – running, waiting, etc
- Program **counter** – location of instruction to next execute
- CPU **registers** – contents of all process-centric registers
- CPU **scheduling information**- priorities, scheduling **queue pointers**
- **Memory-management** information – memory allocated to the process
- Accounting information – **CPU used, clock time** elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

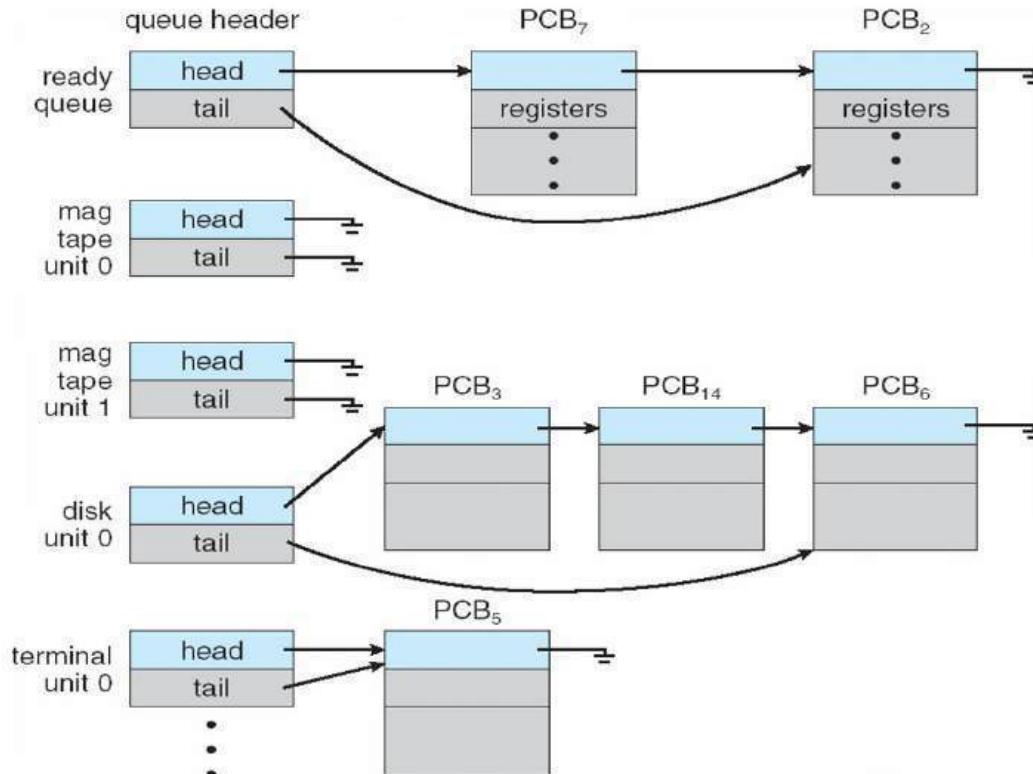
process state
process number
program counter
registers
memory limits
list of open files
...

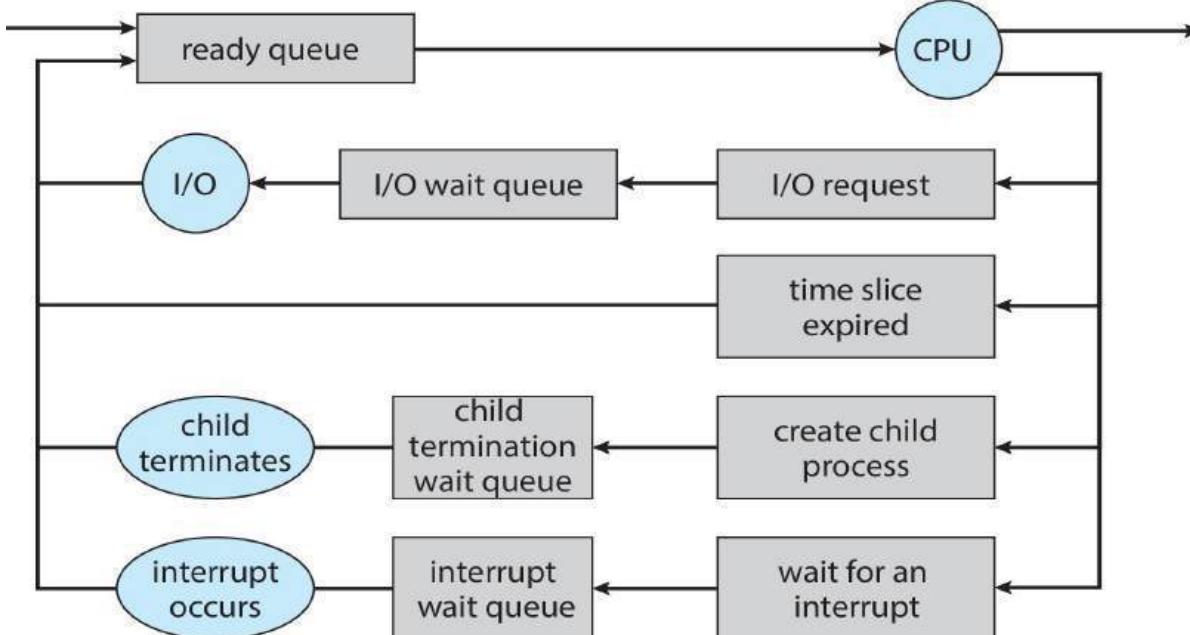
CPU switch from process to process



- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
 - **Job queue** – set of all processes in the system
 - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
 - **Device queues** – set of processes waiting for an I/O device
 - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues

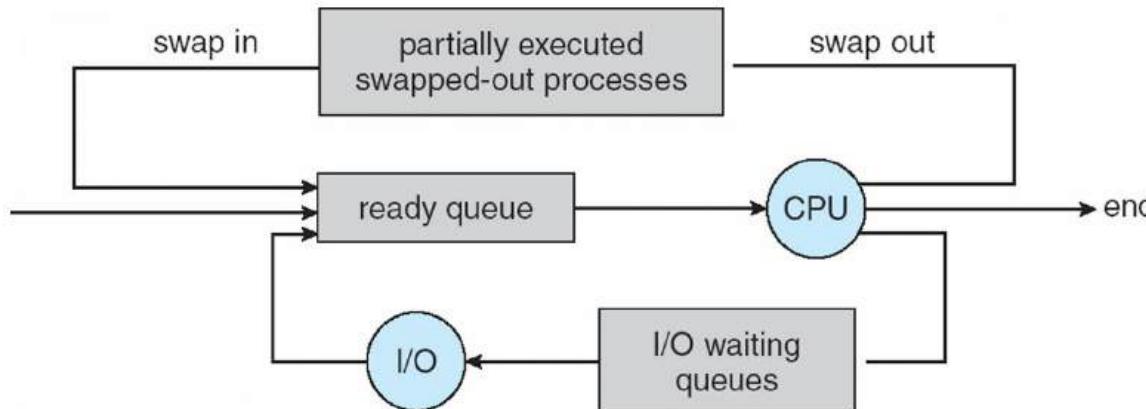




- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
 - Sometimes the only scheduler in a system
 - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
 - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
 - The long-term scheduler controls the degree of multiprogramming

- Processes can be described as either:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good *process mix*

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
 - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



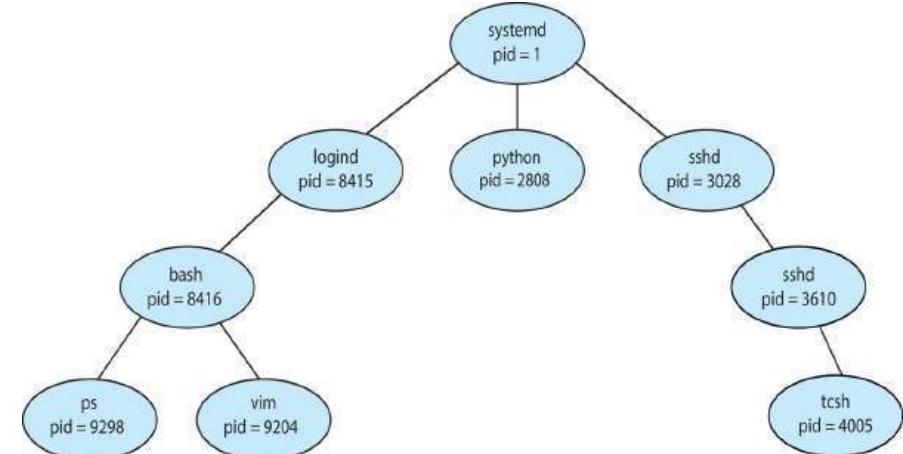
- When CPU switches to another process, the system must **save the state** of the old process and **load the saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is **overhead**; the system does no useful work while switching
 - The more complex the OS and the PCB \square the longer the context switch
- Time dependent on **hardware support**
 - Some hardware provides multiple sets of registers per CPU \square multiple contexts loaded at once

Operations on Processes

System must provide mechanisms for:

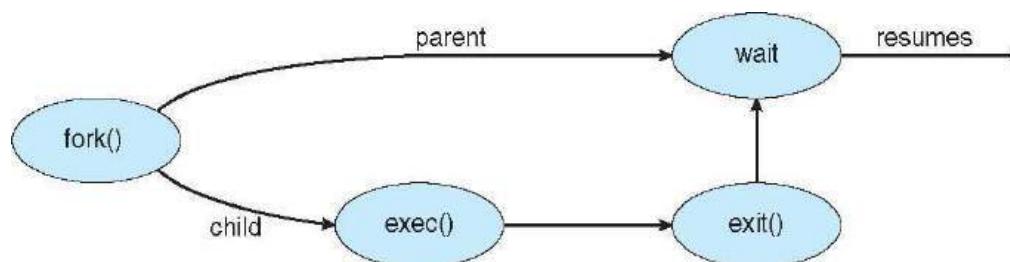
- process creation
- process termination

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution options
 - Parent and children execute concurrently
 - Parent waits until children terminate



Process creation using fork()

- Address space
 - Child **duplicate** of parent
 - Child has a **program loaded** into it
- UNIX examples
 - **fork()** system call **creates new process**
 - **exec()** system call used after a **fork()** to replace the **process' memory space** with a new program



C Program forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) { /* child process */
    execlp("/bin/ls","ls",NULL);
}
else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf("Child Complete");
}

return 0;
}
```

- Process executes **last** statement and then asks the operating system to delete it using the **exit()** system call.
 - Returns status data from child to parent (via **wait()**)
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
 - Child has **exceeded** allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting and the operating systems does not allow a child to continue **if its parent terminates**

Process Termination

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
 - **cascading termination.** All children, grandchildren, etc. are terminated.
 - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()** system call.
The call returns status information and the pid of the terminated process
 - **pid = wait(&status);**
 - If no parent waiting (did not invoke **wait()**) process is a **zombie**
 - If parent terminated without invoking **wait**, process is an **orphan**

- Every process has a unique **process ID**, a non-negative integer
- The process ID is the **only well-known identifier** of a process that is always unique
- It is often used as a **piece of other identifiers**, to guarantee uniqueness.
- Although unique, process IDs are reused. As processes terminate, their IDs become candidates for reuse.
- Most UNIX systems implement algorithms to delay reuse, so that newly created processes are assigned IDs different from those used by processes that terminated recently

- An existing process can create a new one by calling the **fork** function

```
#include <unistd.h>
pid_t fork(void);
```

- The new process created by fork is called the child process
- Return value in the child is 0
- Return value in the parent is the process ID of the new child
- Return value is -1 on error
- The child is a copy of the parent. The child gets a **copy** of the parent's data space, heap, and stack

- A process can terminate normally in five ways
 1. Executing a **return** from the main function
 2. Calling the **exit** function.
 3. Calling the **_exit** or **_Exit** function.
 4. Executing a return from the **start routine** of the last thread in the process.
 5. Calling the **pthread_exit** function from the last thread in the process

A process can terminate abnormally in three ways

- 6. Calling **abort**
- 7. When the process receives certain **signals**
- 8. The last thread responds to a **cancellation request**
- Regardless of how a process terminates, the same code in the kernel is **eventually executed**.
- This kernel code closes all the open descriptors for the process, releases the memory that it was using

- A process that calls wait or waitpid can
 - Block, if all of its children are still running
 - Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
 - Return immediately with an error, if it doesn't have any child processes
- If the process is calling wait because it received the SIGCHLD signal, wait will return immediately. But if we call it at any random point in time, it can block.

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID on success, 0 if state hasn't changed or -1 on failure

- waitid() allows a process to specify which children to wait for.
- Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- Returns: 0 if OK, -1 on error

- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function.
- The process ID does not change across an exec, because a new process is not created;
- exec merely replaces the current process — its text, data, heap, and stack segments — with a brand-new program from disk.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
```

- Return: -1 on error, no return on success

OPERATING SYSTEMS

getpid() and getppid()



```
#include <unistd.h>

pid_t getpid(void);

pid_t getppid(void);
```

- **getpid()** returns the process ID (PID) of the **calling process**.
- **getppid()** returns the process ID of the **parent** of the **calling process**.
- This will be either the ID of the process that created this process using **fork()** or if that process has already terminated, the ID of the process to which this process has been re-parented

Race condition

- A race condition occurs when multiple processes are trying to do something with **shared data** and the final outcome depends on the order in which the processes run.
- The fork function is a lively breeding ground for race conditions, if the logic after the fork either **explicitly or implicitly depends on whether the parent or child runs first after the fork.**
- In general, which process runs first cannot be predicted
- A process that wants to **wait for a child to terminate** must call one of the wait functions.
- If a process wants to wait for its parent to terminate, **a loop of the following form could be used:**

```
while (getppid() != 1)
    sleep(1);
```

- The problem with this type of loop, called **polling**, is that it wastes CPU time, as the caller is awakened every second to test the condition.
- To avoid race conditions and to avoid polling, some form of **signaling** is used between multiple processes

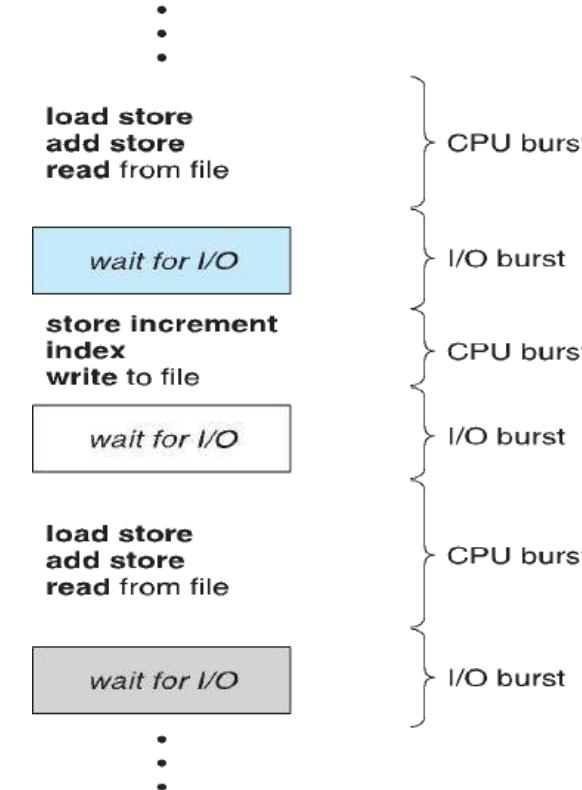
CPU Scheduling - Basic Concepts

- In a system with a single CPU core, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to **maximize CPU utilization.**
- Several processes are kept in memory at one time.
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.
- Every time one process has to wait, another process can take over use of the CPU. On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.

- A process is executed until it must wait, typically for the completion of some I/O request.
- In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively.
- Scheduling of this kind is a fundamental operating-system function.

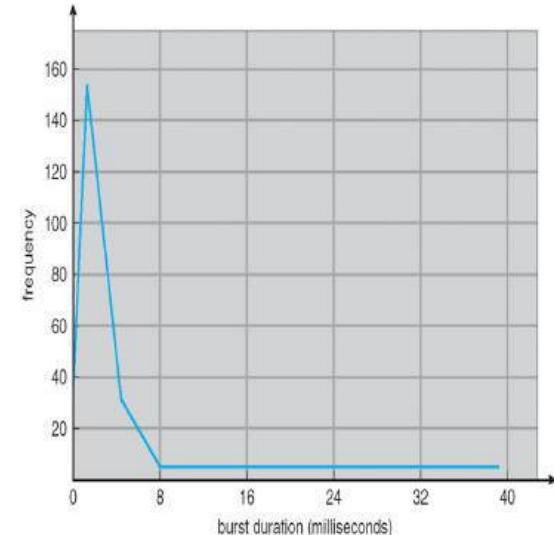
Alternating Sequence of CPU and I/O bursts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O **Burst Cycle** – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times

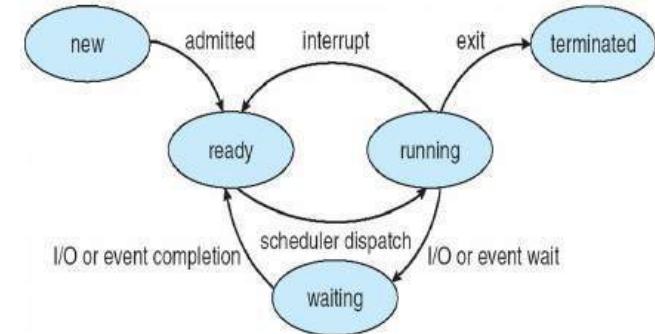
- The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in the Figure.
- An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important when implementing a CPU-scheduling algorithm.



- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
 - Queue may be ordered in various ways
 - It can be FIFO, Priority, queue, tree, unordered linked list
 - The records in the queue are PCB's of the processes

Preemptive Scheduling

- CPU scheduling decisions may take place when a process
 - 1. Switches from running to waiting state
 - 2. Switches from running to ready state
 - 3. Switches from waiting to ready
 - 4. Terminates
- Scheduling under 1 and 4 is **non-preemptive**
- All other scheduling is **preemptive**
 - Consider access to **shared** data
 - Consider preemption while in kernel mode
 - Consider interrupts occurring **during** crucial OS activities
- Scheduling algorithms used in windows3.x, non-preemptive
- Win 95 onwards used preemptive
- Preemptive Scheduling Algorithm is used in Macintosh OS



Preemptive vs Non-Preemptive Scheduling

- Unfortunately, pre-emptive scheduling can result in race conditions when data are shared among several processes. Ex: While one process is updating the shared data, it is pre-empted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.
- A pre-emptive kernel requires mechanisms such as mutex locks to prevent race conditions when accessing shared kernel data structures. Most modern operating systems are now fully pre-emptive when running in kernel mode.

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - switching context
 - switching to user mode
 - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process (performance metric)
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Process	Burst Time
P1	24
P2	3
P3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3

The Gantt Chart for the schedule is:



Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

- Average waiting time: $(0 + 24 + 27)/3 = 17$

- Suppose that the processes arrive in the order: P_2, P_3, P_1

The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- The average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly***

- **Convoy effect** - short process behind long process
 - Consider one CPU-bound and many I/O-bound processes
- FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU
- FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each process to get a share of the CPU at regular intervals.
- It is not desirable to allow one process to keep the CPU for an extended period

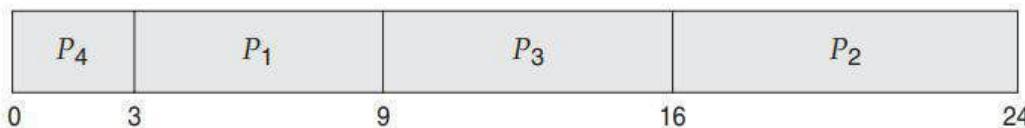
Shortest-Job-First (SJF) Scheduling

- Associate with each process the **length of its next CPU burst**
 - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
 - The difficulty is knowing the length of the **next CPU request**
 - Compute an approximation of the length of the next CPU burst

Example of SJF Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

- SJF scheduling chart



- Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$

Note: If FCFS scheduling is used, average waiting time = $(0 + 6 + 14 + 21) / 4 = 10.25 \text{ ms}$.

Determining Length of Next CPU Burst

- Can be estimated using the length of previous CPU bursts, using exponential averaging

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

where t_{n+1} is the predicted value for the next CPU burst,

The parameter α controls the relative weight of recent and past history in the prediction .

Commonly, $\alpha = 1/2$, so recent history and past history are equally weighted

The value of t_n contains most recent information

The value τ_n stores the past history

- Preemptive version is called **shortest-remaining-time-first**

Determining Length of Next CPU Burst

Calculate the exponential averaging with $T_1 = 10$, $\alpha = 0.5$ and the algorithm is SJF with previous runs as 8, 7, 4, 16.

Initially $T_1 = 10$ and $\alpha = 0.5$ and the run times given are 8, 7, 4, 16 as it is shortest job first,

So the possible order in which these processes would serve will be 4, 7, 8, 16 since SJF is a non-preemptive technique.

So, using formula: $T_2 = \alpha * t_1 + (1-\alpha)T_1$

we have,

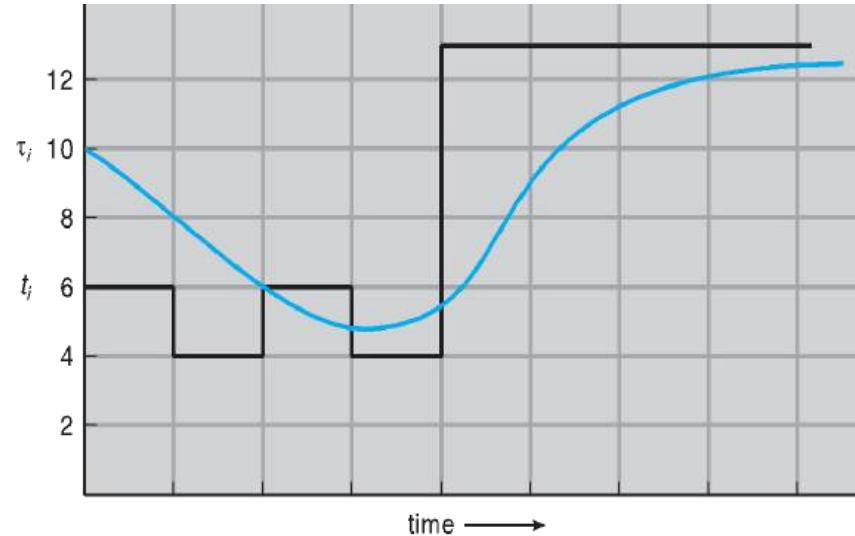
$$T_2 = 0.5 * 4 + 0.5 * 10 = 7, \text{ here } t_1 = 4 \text{ and } T_1 = 10$$

$$T_3 = 0.5 * 7 + 0.5 * 7 = 7, \text{ here } t_2 = 7 \text{ and } T_2 = 7$$

$$T_4 = 0.5 * 8 + 0.5 * 7 = 7.5, \text{ here } t_3 = 8 \text{ and } T_3 = 7$$

$$T_5 = 0.5 * 16 + 0.5 * 7.5 = 11.8, \text{ here } t_4 = 16 \text{ and } T_4 = 7.5$$

Prediction of the Length of the Next CPU Burst



$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12

Examples of Exponential Averaging

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

- $\alpha = 0$
 - $T_{n+1} = T_n$
 - Most recent CPU burst does not count
- $\alpha = 1$
 - $T_{n+1} = \alpha t_n$
 - Only the actual last CPU burst counts
- If we expand the formula, we get:
 - $T_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots$
 - $\quad \quad \quad + (1 - \alpha)^j \alpha t_{n-j} + \dots$
 - $\quad \quad \quad + (1 - \alpha)^{n+1} T_0$
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use non-preemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Burst Time
P_1	8
P_2	4
P_3	1

- What is the average wait time for these processes with the FCFS scheduling algorithm?

$$\text{Average wait time} = (0 + 8 + 12)/3 = 6.67$$

- What is the average wait time for these processes with the SJF scheduling algorithm?

$$\text{Average wait time} = (5 + 1 + 0)/3 = 2$$

Example of Shortest-remaining-time-first

- Preemptive SJF Scheduling is sometimes called SRTF
- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive SJF Gantt Chart*



- Average waiting time = $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5$ msec

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority)
 - Preemptive
 - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem \equiv **Starvation** – low priority processes may never execute
- Solution \equiv **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority Scheduling Gantt chart



- Average waiting time = $(6 + 0 + 16 + 18 + 1) / 5 = 41/5 = 8.2$

Round-Robin Scheduling

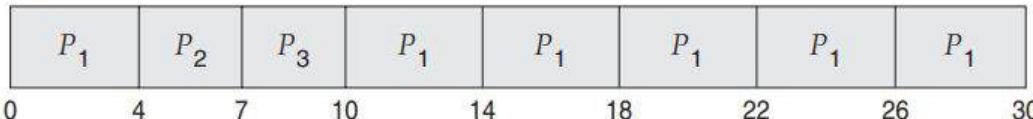
- Round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes
- A small unit of time, called a time quantum or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds in length
- The ready queue is treated as a circular queue
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum

Round-Robin Scheduling Example

- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- RR scheduling Gantt chart using a time quantum of 4 milliseconds



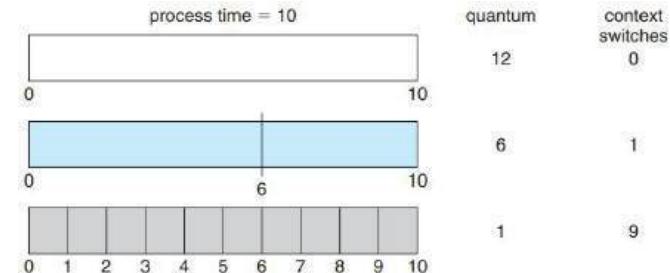
- P_1 waits for 6 milliseconds ($10 - 4$)
- P_2 waits for 4 milliseconds
- P_3 waits for 7 milliseconds.
- The average waiting time = $(6 + 4 + 7)/3 = 5.66$ milliseconds

Round-Robin Scheduling Performance

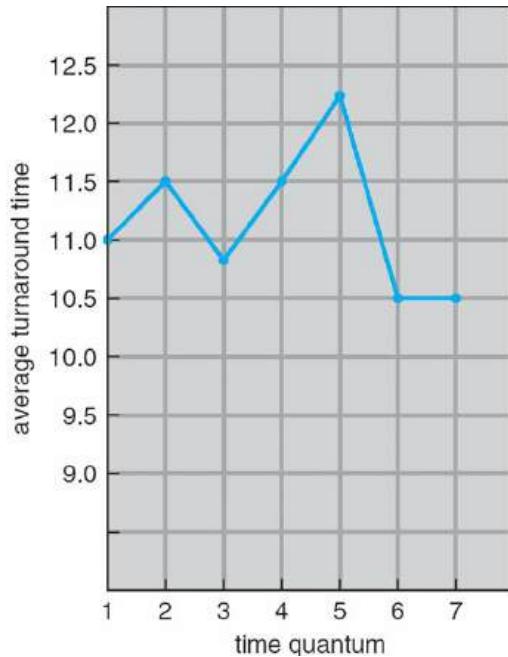
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units.
- Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum.
 - For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.
- Performance of the RR algorithm depends heavily on the size of the time quantum
- If the time quantum is extremely large, the RR policy is the same as the FCFS policy
- If the time quantum is extremely small, the RR approach can result in a large number of context switches

Example of Round-Robin Scheduling Performance

- Consider only one process of 10 time units.
- If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.
- If the quantum is 6 time units, the process requires **2 quanta**, resulting in **one context switch**.
- If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly
- In practice, most modern systems have time quanta ranging from **10 to 100 milliseconds**.
- The time required for a context switch is typically less than 10 microseconds. Thus, the **context-switch time is a small fraction of the time quantum**.



Turnaround Time Varies With The Time Quantum



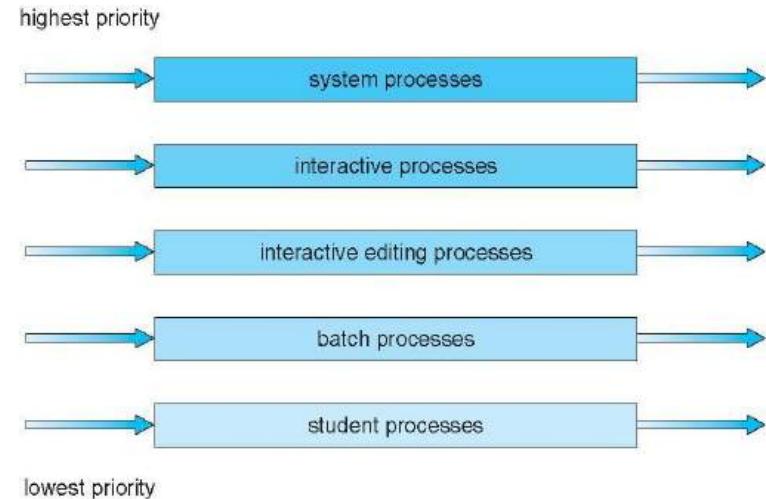
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than the time quantum

- Ready queue is partitioned into separate queues:
 - foreground (interactive)
 - background (batch)
- Process permanently assigned to a queue
- Each queue has its own scheduling algorithm:
 - foreground – RR
 - background – FCFS
- In addition, scheduling must be done between the queues
 - Fixed priority scheduling; (serve all from foreground then from background). Possibility of starvation.
 - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling

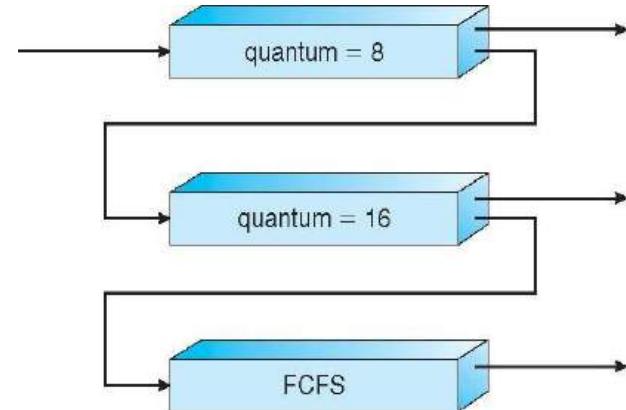
- Each queue has absolute priority over lower-priority queues
- No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty
- If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.



- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - Number of queues
 - Scheduling algorithms for each queue
 - Method used to determine when to upgrade a process
 - Method used to determine when to demote a process
 - Method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
 - Q_0 – RR with time quantum 8 milliseconds
 - Q_1 – RR time quantum 16 milliseconds
 - Q_2 – FCFS
- Scheduling
 - A new job enters queue Q_0 which is served FCFS
 - When it gains CPU, job receives 8 milliseconds
 - If it does not finish in 8 milliseconds, job is moved to queue Q_1
 - At Q_1 job is again served FCFS and receives 16 additional milliseconds
 - If it still does not complete, it is preempted and moved to queue Q_2



- Three queues:
 - Q_0 – RR with time quantum 8 milliseconds
 - Q_1 – RR time quantum 16 milliseconds
 - Q_2 – FCFS
- The scheduler first executes all processes in queue 0.
- Only when queue 0 is empty will it execute processes in queue 1.
- Processes in queue 2 will be executed **only if queues 0 and 1 are empty.**
- A process that arrives for queue 1 will preempt a process in queue 2.
- A process in queue 1 will in turn be preempted by a process arriving in queue 0.

Multiple-processor Scheduling

If multiple CPUs are available, **load sharing** becomes possible, but scheduling problems become correspondingly more complex.

1. Asymmetric multiprocessing

- CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the **master server**.
- The other processors execute only user code.
- Asymmetric multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing.

2. Symmetric multiprocessing (SMP)

- Each processor is **self-scheduling**. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
- All modern operating systems support SMP

- When a process has been running on a specific processor, the data most recently accessed by the process populate the cache of the processor.
- Successive memory accesses by the process are often satisfied in cache memory.
- If the process migrates to another processor, the contents of cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated.
- Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity**
- **Soft affinity** - OS will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors
- **Hard affinity** – OS provides system calls for process to specify a subset of processors on which it may run

Load Balancing

- On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.
- **Load balancing** attempts to keep the workload **evenly distributed** across all processors in an SMP system.
- Load balancing is typically necessary only on systems where each processor has its own **private queue of eligible processes** to execute
 1. **Push migration** - a specific task periodically **checks the load on each processor** and if it finds an **imbalance**, **evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors**
 2. **Pull migration** occurs when an idle processor **pulls a waiting task** from a busy processor.
- Push and pull migration need not be mutually exclusive

- Process Scheduling in Linux
- Linux kernel ran a variation of standard UNIX scheduling algorithm
- It did not support for SMP systems
- It had poor performance for larger processes

- Kernel moved to constant order $O(1)$ scheduling time
- Supported SMP systems - processor affinity and load balancing between processors with good performance
- Poor response times for the interactive processes that are common on many desktop computer systems

- Completely Fair Scheduler (CFS) is the default scheduling algorithm.
- Based on **Scheduling classes**
 - Each class is assigned a specific priority.
 - Using different scheduling classes, the kernel can accommodate different scheduling algorithms
 - Scheduler picks the **highest priority task** in the highest scheduling class
 - Two scheduling classes are included, others can be added
 1. A scheduling class with **CFS** algorithm
 2. A **real-time** scheduling class

Completely Fair Scheduler (CFS)

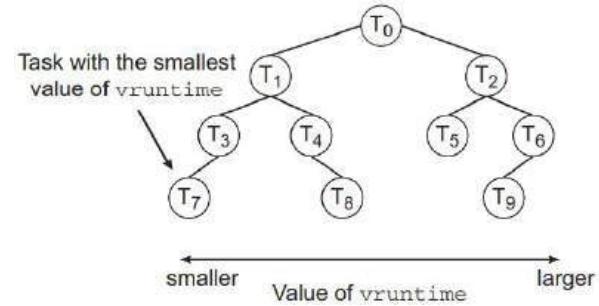
- CFS scheduler assigns a **proportion** of CPU processing time to each task.
- Proportion calculated based on **nice value** which ranges from -20 to +19
 - A numerically lower nice value indicates a higher relative priority.
 - Tasks with lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values
- Calculates **target latency** – interval of time during which task should run **at least once**
 - Proportions of CPU time are allocated from the value of targeted latency computed.
 - Target latency can increase if number of active tasks increase

Completely Fair Scheduler (CFS)

- CFS scheduler doesn't directly assign priorities
- CFS scheduler maintains per task **virtual run time** in variable **vruntime**
 - Associated with decay factor based on priority of task – lower priority is higher decay rate
 - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

Completely Fair Scheduler (CFS)

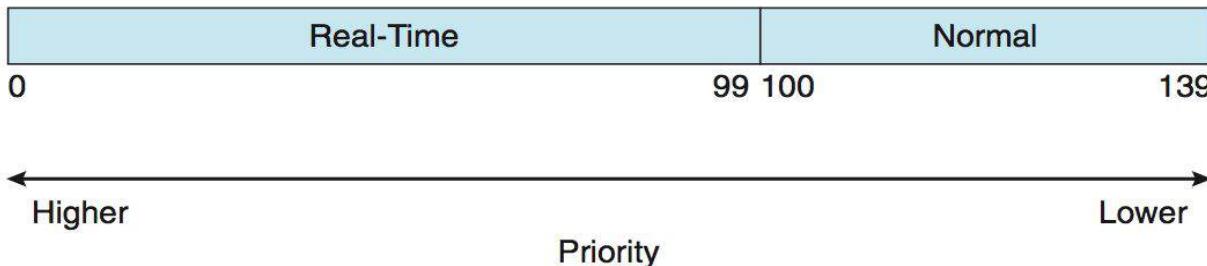
- Each runnable task is placed in a balanced binary search tree whose key is based on the value of **vruntime**
- When a task becomes runnable, it is added to the tree
- When a task is not runnable, it is deleted from the tree
- Navigating the tree to discover the task to run (leftmost node) will require $O(\lg N)$ operations (where N is the number of nodes in the tree).



Completely Fair Scheduler (CFS)

- Assume that two tasks have the same nice values.
- One task is I/O-bound and the other is CPU-bound
- The value of **vruntime** will be lower for the I/O-bound task than for the CPU-bound task, giving the I/O-bound task higher priority than the CPU-bound task.
- If the CPU-bound task is executing when the I/O-bound task becomes eligible to run, the I/O-bound task will preempt the CPU-bound task.

- Real-time scheduling according to POSIX.1b
 - Real-time tasks have static priorities (0-99)
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



- Uses priority based pre-emptive scheduling algorithm
- Scheduler ensures that the **highest-priority thread** will always run
- Windows kernel that handles scheduling is called the **dispatcher**
- Thread selected by the dispatcher runs until it is preempted by a higher-priority thread or until it terminates or until its time quantum ends or until it calls a blocking system call
- Dispatcher uses a 32-level priority scheme
 - **Variable class** is 1-15, **real-time class** is 16-31
 - Priority 0 is memory-management thread
- Queue for each priority class
- If no run-able thread, runs **idle thread**

- Windows API identifies several priority classes to which a process can belong
 - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
 - All are variable except REALTIME
- A thread within a given priority class has a relative priority
 - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

Windows Thread Priorities

- A thread within a given priority class also has a relative priority
- Priority of each thread is based on both the priority class it belongs to (top row in the diagram) and its relative priority within that class (left column in the diagram)

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Windows distinguishes between foreground process and background processes
- Foreground processes given 3x priority boost
- This priority boost gives the foreground process three times longer to run before a time-sharing preemption occurs.

What is a shell?



- Instructions entered in response to the shell prompt have the following syntax: **command [arg1] [arg2] .. [argn]**
- The brackets [] indicate that the arguments are optional. Many commands can be executed with or without arguments
- The shell parses the words or tokens (commandname , options, filenames[s]) and gets the kernel to execute the commands assuming the syntax is correct.
- Typically, the shell processes the complete line after a carriage return (cr) (carriage return) is entered and finds the program that the command line wants executing.
- The shell looks for the command to execute either in the specified directory if given (./mycommand) or it searches through a list of directories depending on your \$PATH variable.

OPERATING SYSTEMS

Common Environment variables

- **SHELL:** This describes the shell that will be interpreting any commands you type in. In most cases, this will be bash by default, but other values can be set if you prefer other options.
- **TERM:** This specifies the type of terminal to emulate when running the shell. Different hardware terminals can be emulated for different operating requirements. You usually won't need to worry about this though.
- **USER:** The current logged in user.
- **PWD:** The current working directory.
- **OLDPWD:** The previous working directory. This is kept by the shell in order to switch back to your previous directory by running cd -.
- **PATH:** A list of directories that the system will check when looking for commands. When a user types in a command, the system will check directories in this order for the executable.
- **HOME:** The current user's home directory.



Common SHELL variables

- **BASHOPTS**: The list of options that were used when bash was executed. This can be useful for finding out if the shell environment will operate in the way you want it to.
- **BASH_VERSION**: The version of bash being executed, in human-readable form.
- **BASH_VERSINFO**: The version of bash, in machine-readable output.

```
ubuntu@ip-172-31-41-208:~$ echo $BASH_VERSION  
5.0.17(1)-release
```



OPERATING SYSTEMS

SHELL basics

- Creating shell variables: ubuntu@ip-172-31-41-208:~\$ `MY_VAR="Hello world"`
- Accessing the value of any shell or environmental variable:

```
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR  
Hello world
```

- Spawning a child shell process

```
ubuntu@ip-172-31-41-208:~$ bash
```

- In the child shell process, the shell variable defined in the parent is not available

```
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR
```

- To terminate the child shell process,

```
ubuntu@ip-172-31-41-208:~$ exit
```

- To pass the shell variables to child shell processes, you need to **export** the variable

```
ubuntu@ip-172-31-41-208:~$ export MY_VAR="Hello world"  
ubuntu@ip-172-31-41-208:~$ bash  
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR  
Hello world
```



OPERATING SYSTEMS

SHELL basics



- Removing a shell variable

```
ubuntu@ip-172-31-41-208:~$ unset MY_VAR  
ubuntu@ip-172-31-41-208:~$ echo $MY_VAR  
ubuntu@ip-172-31-41-208:~$ █
```

- For setting environment variables at login, edit the **.profile** file in the **\$HOME** directory and add the export command

```
ubuntu@ip-172-31-41-208:~$ cd $HOME  
ubuntu@ip-172-31-41-208:~$ vi .profile
```

OPERATING SYSTEMS

SHELL control flow - if

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```



Example: To check if the file exists

```
#!/bin/bash

if [ -f welcome.txt ]; then
    echo "File exists"
else
    echo "File does not exist"
fi
```

OPERATING SYSTEMS

SHELL control flow - loops

```
#!/bin/bash

for i in {1..5}
do
    echo "i = $i"
done|
```

Nested loop:

```
#!/bin/bash

for i in {1..5}
do

    for j in {1..3}
    do
        echo "i = $i and j = $j"
    done
done
```



```
#!/bin/bash

for i in {1..5}
do
|
    if [ $i -eq 3 ]
    then
        continue
    fi

    for j in {1..3}
    do
        echo "i = $i and j = $j"
    done
done
```

```
#!/bin/bash

for i in {1..5}
do|
    if [ $i -eq 3 ]
    then
        break
    fi

    for j in {1..3}
    do
        echo "i = $i and j = $j"
    done
done
```

OPERATING SYSTEMS

cron examples



SCHEDULE	SCHEDULED VALUE
5 0 * 8 *	At 00:05 in August.
5 4 * * 6	At 04:05 on Saturday.
0 22 * * 1-5	At 22:00 on every day-of-week from Monday through Friday.

OPERATING SYSTEMS

Interprocess Communication



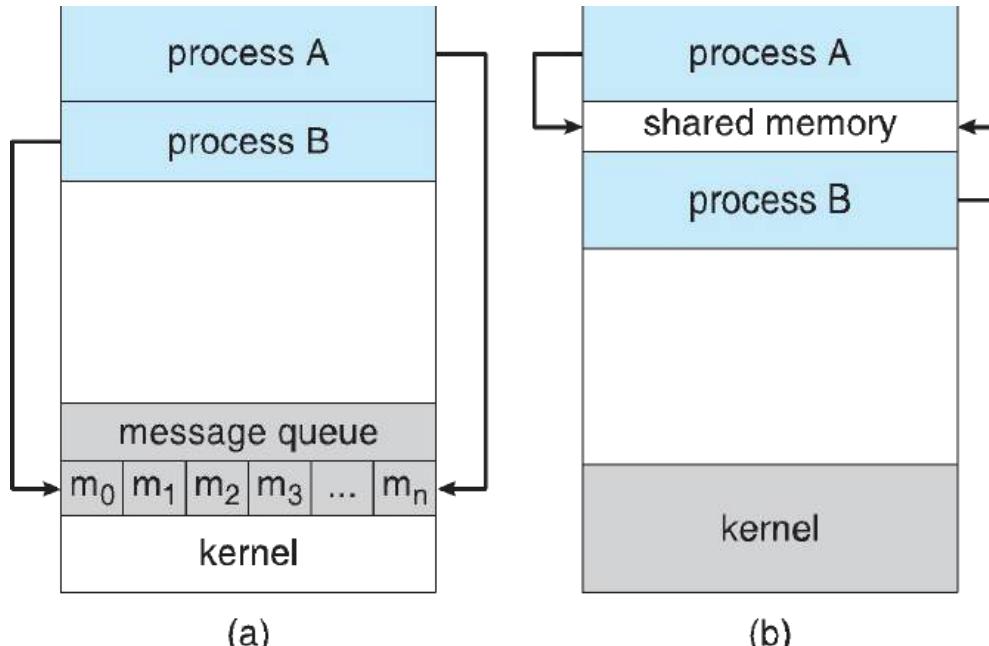
- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
 - **Shared memory**
 - **Message passing**

- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience

OPERATING SYSTEMS

Communication Models

- Two models of IPC
 - a) Message passing
 - b) Shared memory



- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
 - **unbounded-buffer** places no practical limit on the size of the buffer
 - Consumer may have to wait for new items, but the producer can always produce new items
 - **bounded-buffer** assumes that there is a fixed buffer size
 - Consumer must wait if the buffer is empty; producer must wait if the buffer is full

Bounded-Buffer – Producer Consumer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
```

```
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Shared buffer is implemented as a circular array with 2 logical pointers: **in** and **out**
- Buffer is empty when **in == out**; buffer is full when $((\text{in} + 1) \% \text{BUFFER_SIZE}) == \text{out}$
- Variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer
- Solution is correct, but can only use $\text{BUFFER_SIZE}-1$ elements

```
item next_produced;  
while (true) {  
    /* produce an item in next_produced */  
    while (((in + 1) % BUFFER_SIZE) == out)  
        ; /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
}
```

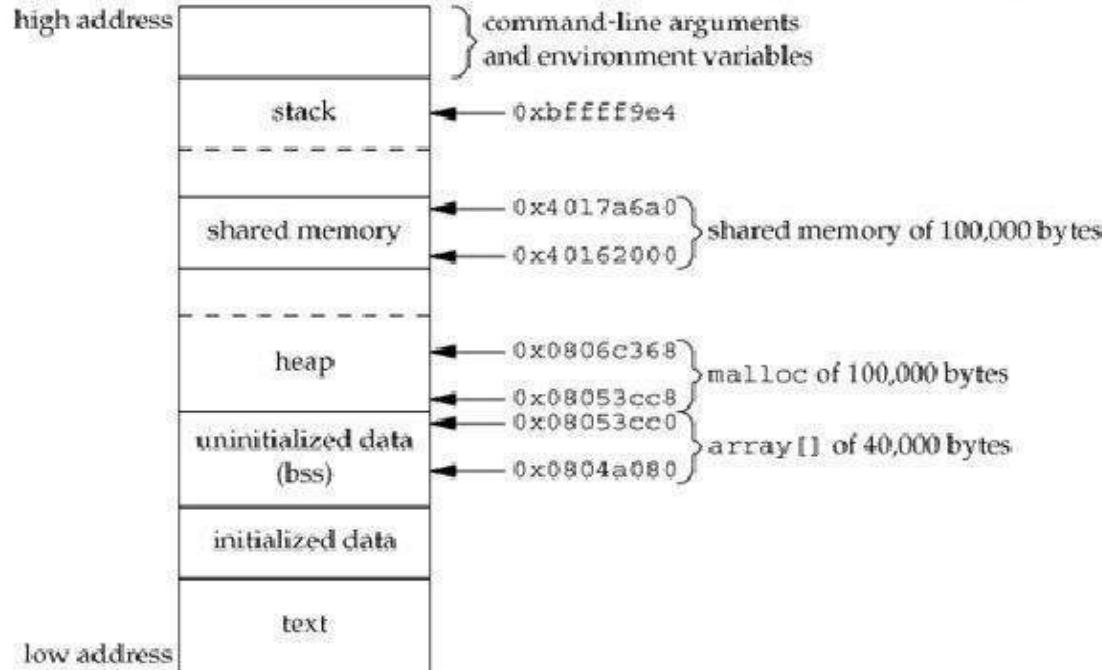
```
item next_consumed;  
while (true) {  
    while (in == out)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    /* consume the item in next_consumed */  
}
```

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

- Shared memory allows two or more processes to share a given region of memory.
- Shared memory is the fastest form of IPC, because the data does not need to be copied between the client and the server.
- The only trick in using shared memory is synchronizing access to a given region among multiple processes.
- If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done.
- Often, semaphores are used to synchronize shared memory access. (record locking can also be used.)

Memory layout on an Intel-based Linux system



Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send(message)**
 - **receive(message)**
- The *message* size is either fixed or variable

- If processes P and Q wish to communicate, they need to:
 - Establish a ***communication link*** between them
 - Exchange messages via send/receive
- Implementation issues:
 - How are links established?
 - Can a link be associated with more than two processes?
 - How many links can there be between every pair of communicating processes?
 - What is the capacity of a link?
 - Is the size of a message that the link can accommodate fixed or variable?
 - Is a link unidirectional or bi-directional?

- Implementation of communication link
 - Physical:
 - Shared memory
 - Hardware bus
 - Network
 - Logical:
 - Direct or indirect
 - Synchronous or asynchronous
 - Automatic or explicit buffering

- Processes must name each other explicitly:
 - **send** ($P, \text{ message}$) – send a message to process P
 - **receive**($Q, \text{ message}$) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional

Indirect Communication (Cont.)

- Operations
 - create a new mailbox (port)
 - send and receive messages through mailbox
 - destroy a mailbox
- Primitives are defined as:
 - **send**(*A, message*) – send a message to mailbox A
 - **receive**(*A, message*) – receive a message from mailbox A

Indirect Communication (Cont.)

- Mailbox sharing
 - P_1 , P_2 , and P_3 share mailbox A
 - P_1 sends; P_2 and P_3 receive
 - Who gets the message?
- Solutions
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Message Passing - Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send** -- the sender is blocked until the message is received
 - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** -- the sender sends the message and continue
 - **Non-blocking receive** -- the receiver receives:
 - A valid message, or
 - Null message
- Different combinations possible
 - If both send and receive are blocking, we have a **rendezvous** between the sender and the receiver

- Producer-consumer becomes trivial

```
message next_produced;
while (true) {
    /* produce an item in next_produced */
    send(next_produced);
}
```

```
message next_consumed;
while (true) {
    receive(next_consumed);

    /* consume the item in next_consumed */
}
```

- Queue of messages attached to the link (direct or indirect); messages reside in a temporary queue
- Queues can be implemented in one of three ways
 1. Zero capacity – no messages are queued on a link.
Sender must wait for receiver (rendezvous)
 2. Bounded capacity – finite length of n messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits
- Zero-capacity case is sometimes referred to as a message system with no buffering; other cases are referred to as systems with automatic buffering

- Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems.
- Pipes have two limitations.
 1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
 2. Pipes can be used only between processes that have a **common ancestor**. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

- A pipe is created by calling the **pipe()** function.

```
#include <unistd.h>
int pipe(int fd[2]);
```

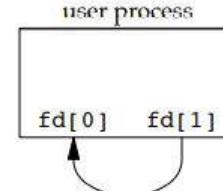
- Return value: 0 if OK, -1 on error
- Two file descriptors are returned through the fd argument:
 1. fd[0] is open for reading
 2. fd[1] is open for writing.
- The output of fd[1] is the input for fd[0]

OPERATING SYSTEMS

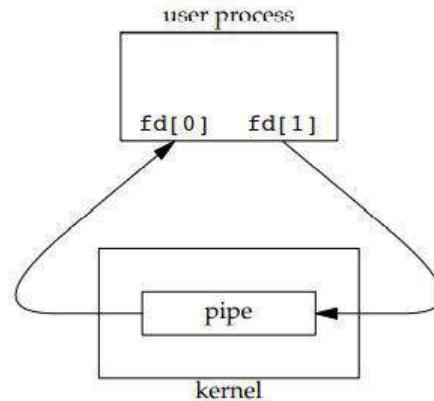
pipe()

- Two ways to picture a half-duplex pipe

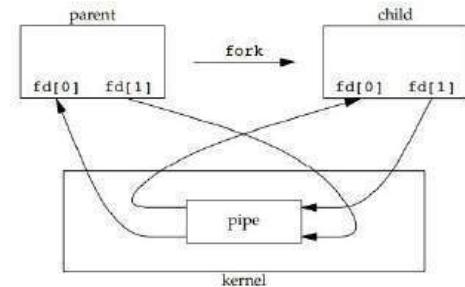
1. The two ends of the pipe connected in a single process



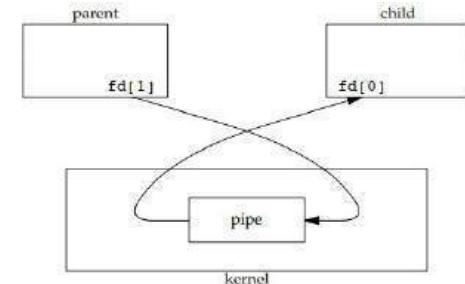
2. The data in the pipe flows through the kernel



- Normally, a process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa



- For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]) and the child closes the write end (fd[1]).
- For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0]



Pipe from parent to child

OPERATING SYSTEMS

popen() and pclose()

- A common operation is to create a pipe to another process to either read its output or send it input, the standard I/O library has historically provided the **popen()** and **pclose()** functions
- These two functions handle all the work: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
```

```
int pclose(FILE *fp);
```

- The function **popen()** does a fork and exec to execute the cmdstring and returns a standard I/O file pointer.
- If type is "r", the file pointer is connected to the standard output of cmdstring.
- If type is "w", the file pointer is connected to the standard input of cmdstring
- The function **popen()** returns file pointer if OK, NULL on error
- The function **pclose()** closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell or -1 on error

- FIFOs are sometimes called **named pipes**.
- Unnamed pipes can be used only between related processes when a common ancestor has created the pipe.
- With FIFOs, unrelated processes can exchange data
- Creating a FIFO is similar to creating a file

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
int mkfifoat(int fd, const char *path, mode_t mode);
```

- Functions mkfifo() and mkfifoat() return 0 if OK, -1 on error
- Once a FIFO has been created using mkfifo() or mkfifoat() , it can be opened using open(). Normal file I/O functions (e.g., close, read, write, unlink) all work with FIFOs.
- There are two uses for FIFOs.
 1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
 2. FIFOs are used as rendezvous points in client–server applications to pass data between the clients and the servers

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
- A new queue is created or an existing queue opened by **msgget**.

```
#include <sys/msg.h>
int msgget(key_t key, int flag);
```

- Returns message queue ID if OK, -1 on error
- New messages are added to the end of a queue by **msgsnd**.

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

- Every message has a positive long integer type field, a non-negative length, and the actual data bytes all of which are specified to **msgsnd** when the message is added to a queue.
- Messages are fetched from a queue by **msgrcv**.

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

- Messages need not be fetched in a first-in, first-out order. Instead, can be fetched based on their type field

- A semaphore is a counter used to provide access to a shared data object for multiple processes.
- To obtain a shared resource, a process needs to do the following:
 1. Test the semaphore that controls the resource.
 2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
 3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.
- When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1.
- If any other processes are asleep, waiting for the semaphore, they are awakened.

- First need to obtain a semaphore ID by calling the semget function

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
```

- Returns: semaphore ID if OK, -1 on error
- **semctl** function is the catchall for various semaphore operations.

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
```

- The function **semop** atomically performs an array of operations on a semaphore set

```
#include <sys/sem.h>
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

- The **semoparray** argument is a pointer to an array of semaphore operations
- Returns 0 if OK, -1 on error

- Shared memory allows two or more processes to share a given region of memory.
- This is the fastest form of IPC, because the data does not need to be copied between the client and the server
- The challenge in using shared memory is synchronizing access to a given region among multiple processes.
- Semaphores and mutexes are used to synchronize shared memory access
- Function **shmget** is used to obtain a shared memory identifier

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int flag);
```

- Returns shared memory ID if OK, -1 on error

- Function **shmctl** is the catchall for various shared memory operations

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- Returns 0 if OK, -1 on error
- The cmd argument specifies one of the commands to be performed on the segment specified by shmid
- Once a shared memory segment has been created, a process is attached to this memory segment by calling **shmat**

```
#include <sys/shm.h>
void *shmat(int shmid, const void *addr, int flag);
```

- Returns: pointer to shared memory segment if OK, -1 on error

Shared memory

- Function call **shmctl** will detach a shared memory segment from a process

```
#include <sys/shm.h>
int shmdt(const void *addr);
```

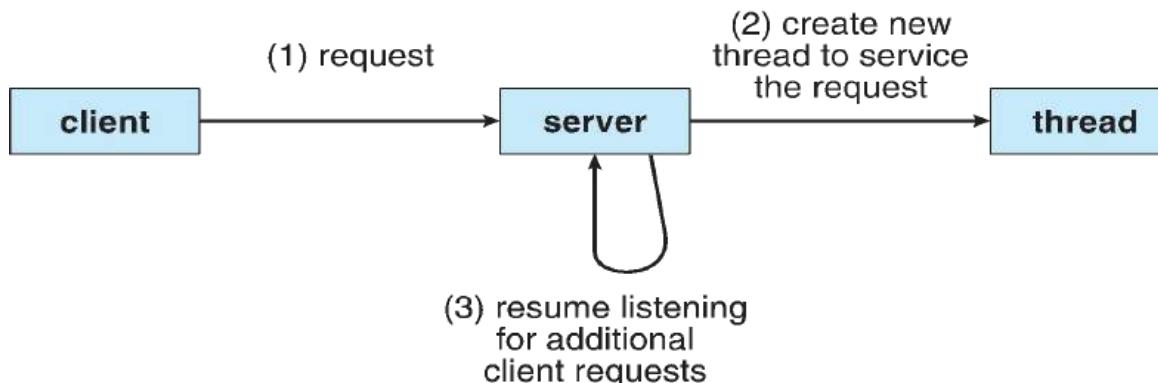
- Returns 0 if OK, -1 on error
- A shared memory segment can be removed by calling **shmctl** with a command of IPC_RMID.

Overview and Motivation

- A Thread is a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.
- It consists of thread ID, Program counter, a register set and stack
- Shares with other threads of same process its code, data, file descriptors, signals
- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads.
- Application 1: internet browser.
 - numerous tabs open at a given time
 - Multiple threads of execution are used to load content, display animations, play a video, fetch data from a network and so on.

Multithreaded Server Architecture

- Process creation is heavy-weight while thread creation is light-weight
- Process creation is time consuming, resource intensive
- Threads also play a vital role in remote procedure call (RPC) systems
- Can simplify code, increase efficiency
- Kernels are generally multithreaded



- **Responsiveness** – may allow continued execution if part of process is blocked or performing lengthy operations.
 - It is useful in designing user interfaces.
 - A multithreaded Web browser could allow user interaction in one thread while an image was being loaded in another thread.
 - Example: click on the button
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing.
 - Programmer needs to specify the techniques for sharing
 - But threads share the memory and other resources
 - Sharing of resources helps in having many threads within the same address space

- **Economy** – cheaper than process creation, thread switching lower overhead than context switching.
 - In Solaris, for example, creating a process is about thirty times slower than creating a thread, and context switching is about five times slower.
- **Scalability** – process can take advantage of multiprocessor architectures.
 - Threads can run on multiple cores parallelly

Process

- Will by default not share memory
- Most file descriptors not shared
- Don't share filesystem context
- Don't share signal handling

Thread

- Will by default share memory
- Will share file descriptors
- Will share filesystem context
- Will share signal handling

Attributes shared by Threads

- process ID and parent process ID;process group ID and session ID;
- controlling terminal;
- process credentials (user and group Ids);open file descriptors;
- record locks created using fcntl();signal dispositions;
- file system–related information: umask, cwd and root directory;
- resource limits;CPU time consumed (as returned by times());
- resources consumed (as returned by getrusage()); nice value (set by setpriority() and nice()).

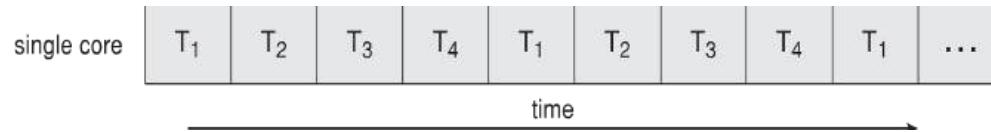
Attributes specific to Threads

- thread ID ;signal mask;
- Thread-specific data ;
- the errno variable;
- floating-point environment (see fenv(3));
- stack (local variables and function call linkage information i.e CPU registers saved in the called function's stack frame when one function calls another function and restored for the calling function when the called function returns)
- and a few more

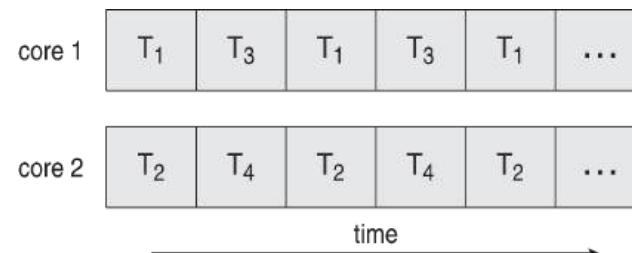
- A thread consists of the following information necessary to represent an execution context within a process.
- thread ID that identifies the thread within a process
- Every thread has a thread ID
- set of register values
- stack
- scheduling priority and policy
- signal mask
- Errno variable
- Thread-specific/thread-private data (each thread to access its own separate copy of the data, without worrying about synchronizing access with other threads)

- When a thread is created, there is no guarantee which will run first: the newly created thread or the calling thread.
- The newly created thread has access to the process address space and inherits the calling thread's floating-point environment and signal mask
- The set of pending signals for the thread is cleared.
- The pthread functions usually return an error code when they fail.
They don't set errno like the other POSIX functions.

- Concurrent execution on single-core system:

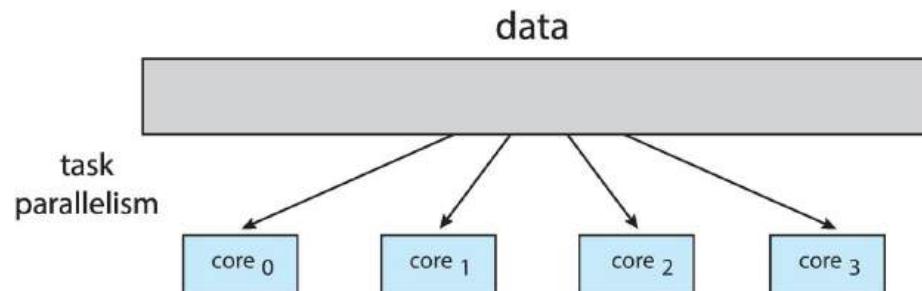
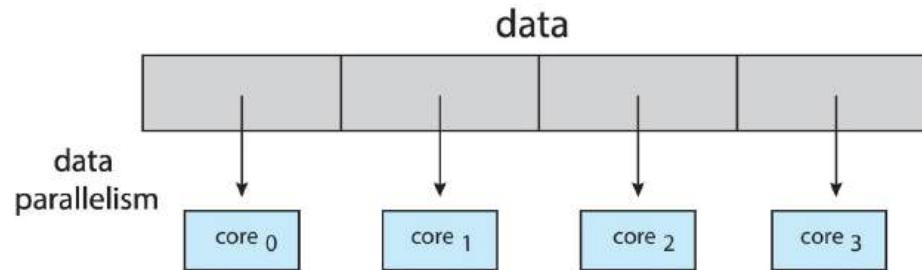


- Parallelism on a multi-core system:



- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
 - Single processor / core, scheduler providing concurrency

- Types of parallelism
 - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
 - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
 - CPUs have cores as well as ***hardware threads***
 - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

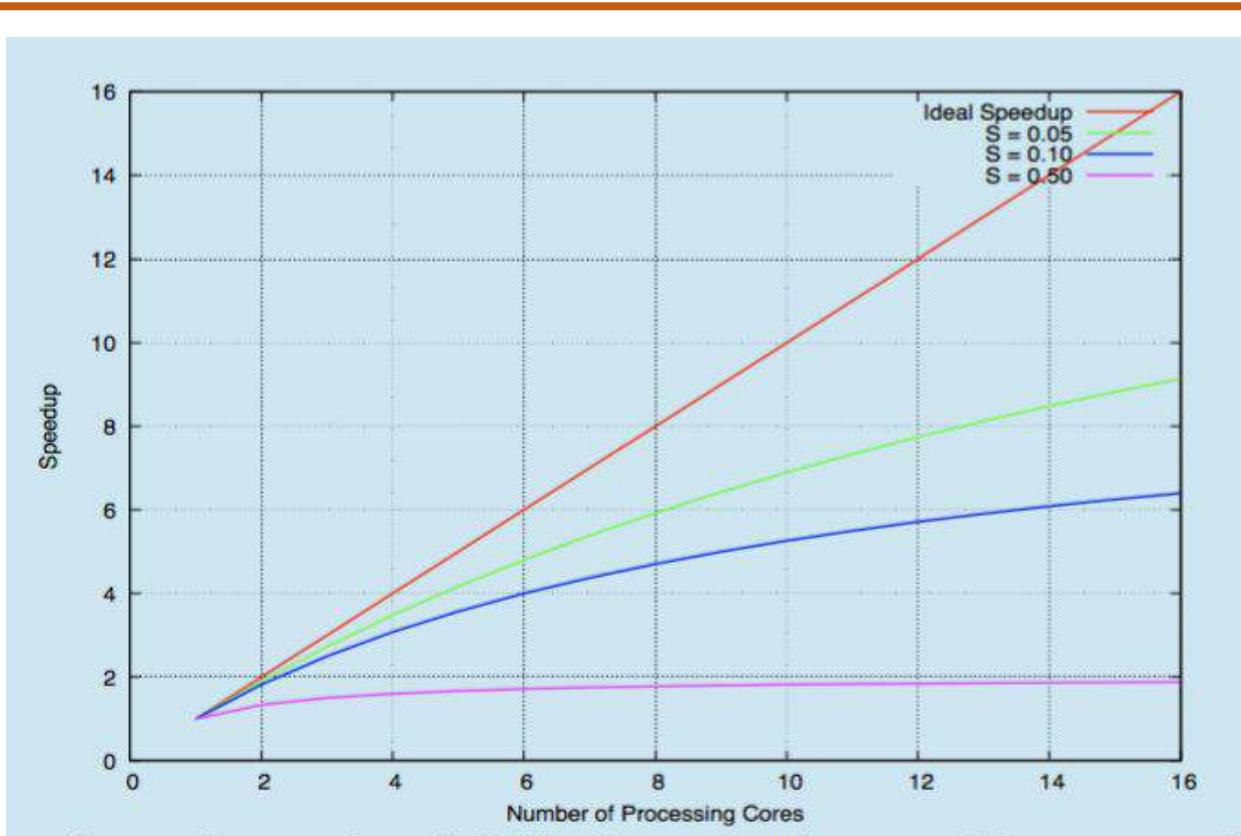


- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is portion of an application that needs to be done in serial
- N processing cores

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

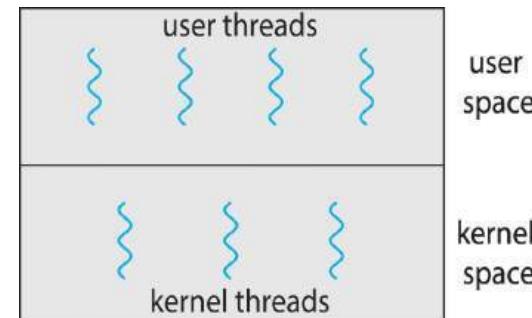
- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches $1 / S$
Serial portion of an application has disproportionate effect on performance gained by adding additional cores
- But does the law take into account contemporary multicore systems?

Amdahl's Law



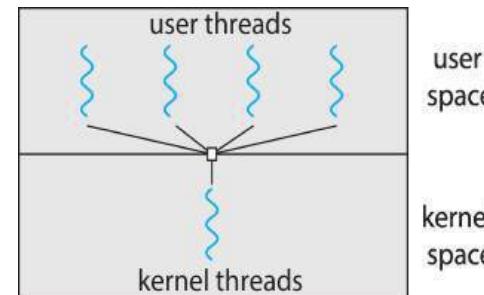
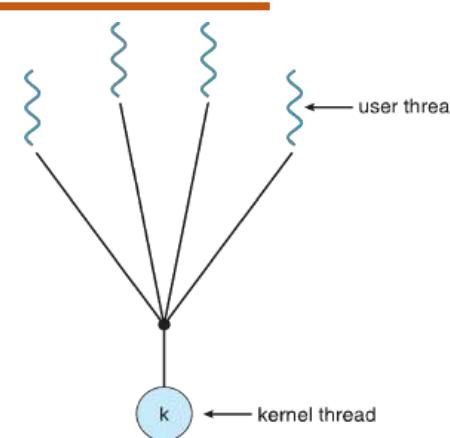
User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Windows threads
 - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
 - Windows
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X



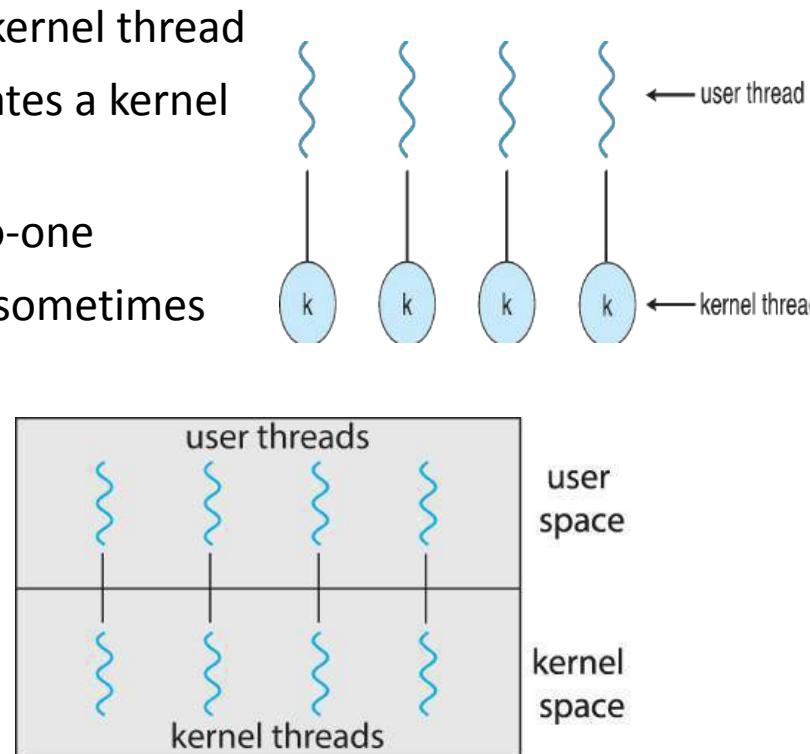
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads

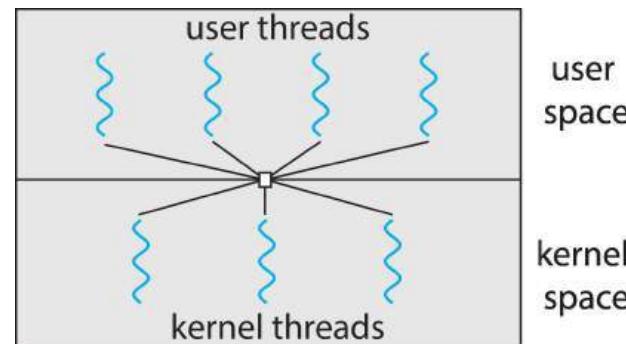
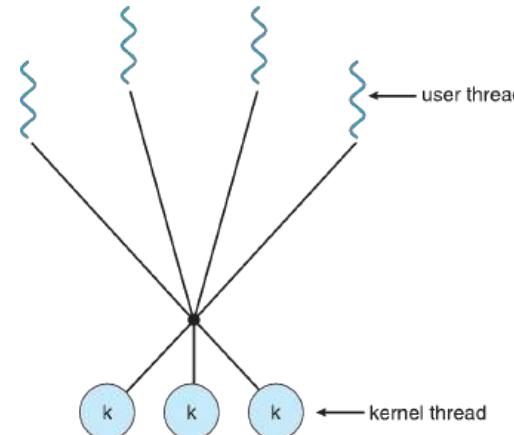


One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux
 - Solaris 9 and later

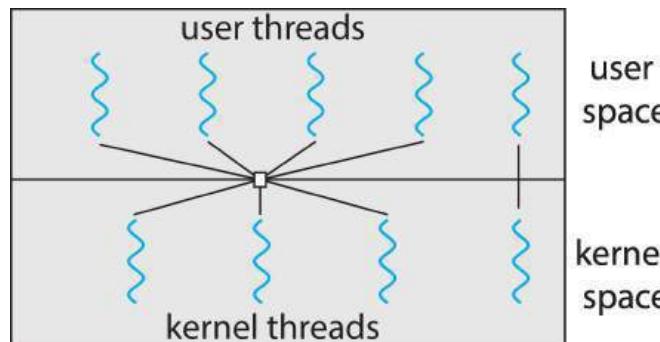
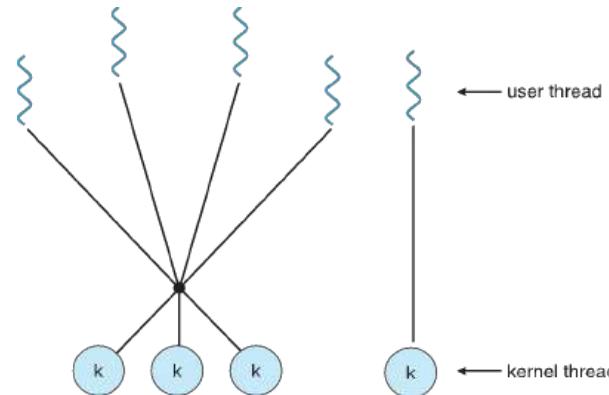


- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 and earlier



- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- *Specification, not implementation*
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }
}
```

Pthreads Example

```
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP.
- Scheduling of user level threads (ULT) to kernel level threads (KLT) via light-weight process (LWP) by the application developer.
 - **Lightweight Process (LWP) :**
Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources
 - Scheduling of kernel level threads by the system scheduler to perform different unique OS functions.

- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
 - Known as **process-contention scope (PCS)** since scheduling competition is within the process
 - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

- API allows specifying either PCS or SCS during thread creation
 - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
 - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
```

```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

- Provides the programmer with an API for creating and managing threads.
- Two primary ways of implementing a thread library.
 - Provide a library entirely in user space with no kernel support.
 - All code and data structures for the library exist in user space.
 - Implement a kernel-level library
 - Code and data structures for the library exist in kernel space.
 - Invoking a function directly by the operating system.

Thread Libraries: Pthreads

- Three main thread libraries are in use today:
 - (1) POSIX Pthreads,
 - (2) Win32,
 - (3) Java.
- Pthreads, the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library.
- The Win32 thread library is a kernel-level library available on Windows systems.
- The Java thread API allows threads to be created and managed directly in Java programs.

Thread Libraries: Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification, not implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }
}
```

Pthreads Example

```
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

- Windows implements the Win32 API as its primary API.
- A Windows application runs as a separate process, and each process may contain one or more threads.
- Windows uses the one-to-one mapping.
- Windows also provides support for a **fiber** library, which provides the functionality of the many-to-many model

- The general components of a thread include:
 - A thread ID uniquely identifying the thread
 - A register set representing the status of the processor.
 - A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode
 - A private storage area various run-time libraries and dynamic link libraries (DLLs)

- Threads are created in the Win32 API using the CreateThread() function the Win32 API.
- Using the WaitForSingleObject() function, which causes the creating thread to block until the child thread has exited

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr,"An integer >= 0 is required\n");
        return -1;
    }
}
```

Windows Multithreaded C Program (Cont.)

```
/* create the thread */
ThreadHandle = CreateThread(
    NULL, /* default security attributes */
    0, /* default stack size */
    Summation, /* thread function */
    &Param, /* parameter to thread function */
    0, /* default creation flags */
    &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) {
    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}
```

- Processes can execute concurrently
 - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:

Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

```
while (true) {  
    /* produce an item in next_produced */  
  
    while (counter == BUFFER_SIZE);  
        /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE; /* Variable in points to the next free position in the buffer */  
    counter++;  
}
```

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE; /* Variable out points to the first full position in the buffer */  
    counter--;  
    /* consume the item in next consumed */  
}
```

Race Condition

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

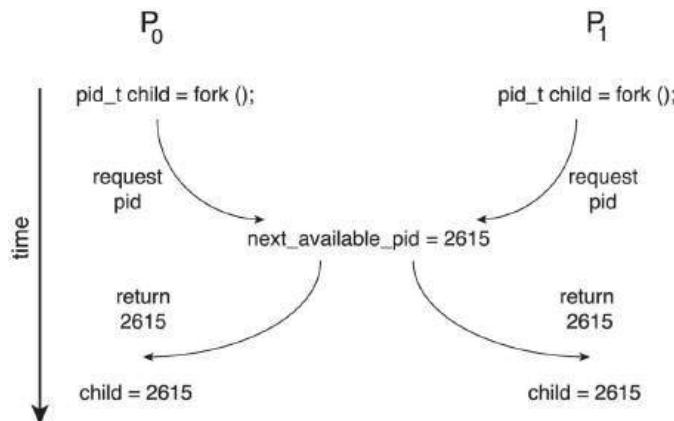
```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

```
S0: producer execute register1 = counter      {register1 = 5}
S1: producer execute register1 = register1 + 1  {register1 = 6}
S2: consumer execute register2 = counter      {register2 = 5}
S3: consumer execute register2 = register2 - 1  {register2 = 4}
S4: producer execute counter = register1       {counter = 6 }
S5: consumer execute counter = register2       {counter = 4}
```

Race Condition (Another Example)

- Processes P_0 and P_1 are creating child processes using the fork() system call
- Race condition on kernel variable next_available_pid which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable next_available_pid the same pid could be assigned to two different processes!

- Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has **critical section** segment of code
 - Process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

- General structure of process P_i

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
 - Assume that each process executes at a nonzero speed
 - No assumption concerning **relative speed** of the n processes

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode.
 - Preemptive kernels are difficult to design on SMP architectures
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
 - Essentially free of race conditions in kernel mode
 - It is free from race conditions on kernel data structures
- Preemptive kernel may be more responsive. Suitable for real-time systems.

Peterson's Solution

- Software-based solution to the Critical Section problem.
- Good algorithmic description of solving the problem
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- **Peterson's Solution** restricted to two process.
- P_i and P_j are two process, $j=1-i$
- Peterson solution requires two processes share two data items:
 - int turn;
 - Boolean flag[2]
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = true** implies that

Algorithm for Process P_i

```
while (true) {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    /* critical section */  
    flag[i] = false;  
    /* remainder section */  
}
```

The structure of process P_i in Peterson's solution

- To prove solution is correct, we need show
- Mutual exclusion is preserved

P_i enters critical section only if:

turn = i

and **turn** cannot be both 0 and 1 at the same time

- What about the Progress requirement?
- What about the Bounded-waiting requirement?

Correctness of Peterson's Solution

- Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

P_i enters CS only if:

either $\text{flag}[j] = \text{false}$ or $\text{turn} = i$

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Code for process i

```
do{  
    flag[i]=TRUE  
    turn=j  
    while(flag[j]&&turn==j);//Do-nop  
        critical section  
        flag[i]=FALSE;  
    Reminder section  
}while(TRUE)
```

Code for process j

```
do{  
    flag[j]=TRUE  
    turn=i  
    while(flag[i]&&turn==i);//Do-nop  
        critical section  
        flag[j]=FALSE;  
    Reminder section  
}while(TRUE)
```

- **Principles of Concurrency**

- relative speed of execution of processes is not predictable.
- system interrupts are not predictable
- scheduling policies may vary

- Software based solutions are not guaranteed to work on modern computer architectures
- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
 - Protecting critical regions via locks.
 - synchronization can be done through Lock & Unlock technique
 - Locking part is done in the Entry Section. After locking the process enter critical section.
 - The process is moved to the Exit Section after it is done with execution in CS.
 - Unlock is done in exit section.
 - This process is designed in such a way that all the three conditions of the Critical Sections are satisfied

Synchronization Hardware

- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - 4 Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - 4 **Atomic** = non-interruptible
 - Either test memory word and set value
 - Or swap contents of two memory words

- Test and Set Lock (TSL) is a synchronization mechanism.
- It uses a test and set instruction to provide the synchronization among the processes executing concurrently.
- It is an instruction that returns the old value of a memory location and sets the memory location value to 1 as a single atomic operation.
- If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.

Definition:

```
boolean test_and_set (boolean *target)
```

```
{
```

```
    boolean rv = *target;
```

```
    *target = TRUE;
```

```
    return rv;
```

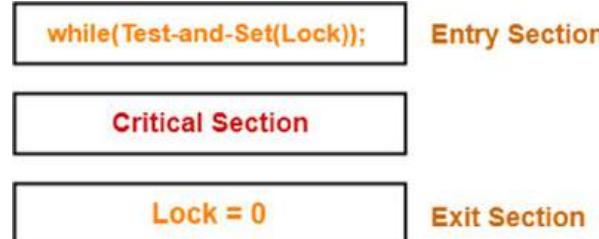
```
}
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
    /* critical section */  
    lock = false;  
    /* remainder section */  
} while (true);
```



compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
  
    return temp;  
}
```

```
do{  
    while(compare_and_swap(&lock,0,1)!=0);  
    Critical section  
    lock=0  
    Remainder section  
}while(true)
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if *value == expected. That is, the swap takes place only under this condition.
4. In the [x86](#) (since [80486](#)) and [Itanium](#) architectures this is implemented as compare and exchange (CMPSXCHG) instruction

Solution using compare_and_swap()

- Shared integer “lock” initialized to 0;
- Solution:

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        /* do nothing */
    /* critical section */

    lock = 0;
    /* remainder section */
} while (true);
```

Mutual exclusion is satisfied

Do not satisfy bounded waiting requirement

Bounded-waiting Mutual Exclusion with test_and_set

This test_and_set algorithm satisfies all the critical section requirements

The common data structures are
boolean waiting[n];
boolean lock;

```
do {
```

```
    waiting[i] = true;  
    key = true;  
    while (waiting[i] && key)  
        key = test_and_set(&lock);  
    waiting[i] = false;
```

```
/* critical section */
```

```
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
    if (j == i)  
        lock = false;  
    else  
        waiting[j] = false;
```

```
/* remainder section */
```

```
} while (true);
```

Mutex Locks

- ❑ Previous solutions i.e hardware and software solutions are complicated and generally inaccessible to application programmers
- ❑ OS designers build software tools to solve critical section problem
- ❑ Simplest is mutex lock
- ❑ Protect a critical section by first **acquire()** a lock then **release()** the lock
 - ❑ Boolean variable indicating if lock is available or not
- ❑ Calls to **acquire()** and **release()** must be atomic
 - ❑ Usually implemented via hardware atomic instructions
- ❑ But this solution requires **busy waiting**
- ❑ This lock therefore called a **spinlock**
- ❑ It is problem in real time systems
- ❑ Busy waiting wastes CPU cycles

Advantages of spinlocks

- ❖ no context switch is required when a process must wait on a lock.
- ❖ context switch may take considerable time.
- ❖ When locks are expected to be held for short times, spinlocks are useful
- ❖ These are employed on multiprocessor systems where one thread can “spin” on one processor while another thread performs its critical section on another processor.

Solution to Critical-section Problem Using Locks

```
while (TRUE) {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
}
```

- Implementation of acquire and release

- acquire() {**
 while (!available)
 ; /* busy wait */
 available = false;
}
- release() {**
 available = true;
}

OPERATING SYSTEMS

acquire() and release()



- Solution to a critical section problem using mutex

- **do {**

- acquire lock*

- critical section**

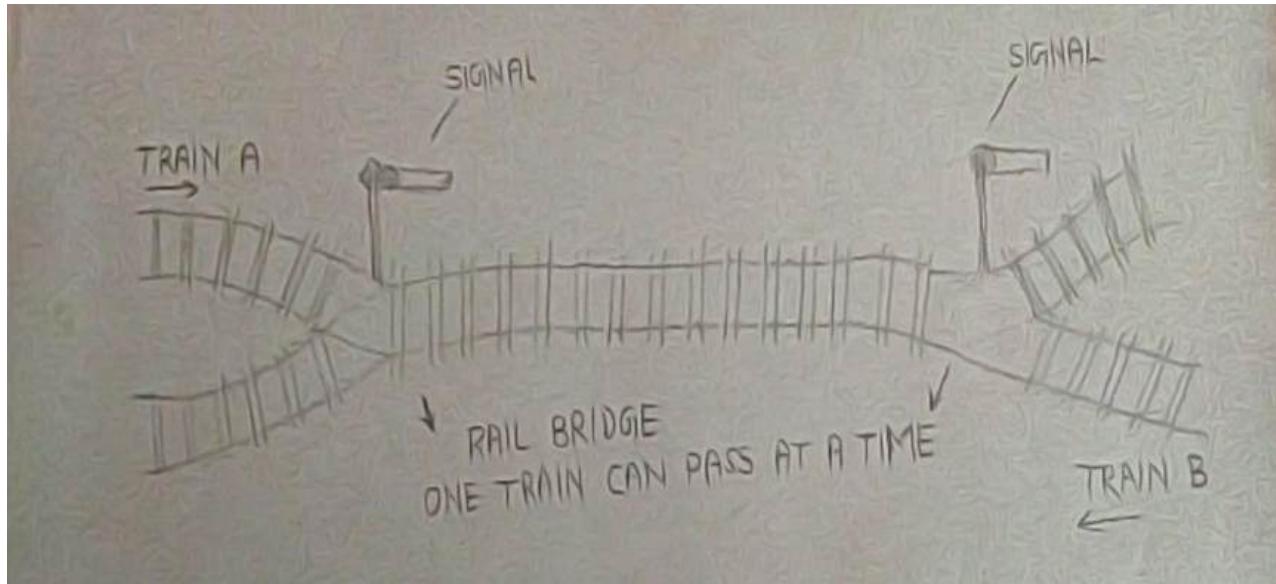
- release lock*

- remainder section**

- } while (true);**

OPERATING SYSTEMS

Scenario 2



Scenario 1

- Consider a library of an university with 10 rooms
- At a time one room can be used by only one student by informing the front desk for reading.
- Once he completes reading, he has to inform the front desk.
- Person at front desk knows how many rooms are available for use and how many are occupied, how many of them are waiting.
- Once the room is vacant, who will get the chance to occupy the room?

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
 - **wait()** and **signal()**
 - 4 Originally called **P()** and **V()**
- Definition of the **wait()** operation

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```
- Definition of the **signal()** operation

```
signal(S) {  
    S++;  
}
```

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2 . Create a semaphore “synch” initialized to 0

P1:

```
S1;  
signal(synch);
```

P2:

```
wait(synch);  
S2;
```

- Can implement a counting semaphore S as a binary semaphore

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
 - Could now have **busy waiting** in critical section implementation
 - 4 But implementation code is short
 - 4 Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
 - value (of type integer)
 - pointer to next record in the list
- Two operations:
 - block** – place the process invoking the operation on the appropriate waiting queue
 - wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- typedef struct{**

int value;

struct process *list;

} semaphore;

Semaphore Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
    if (S->value >= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    } }
```

Problems with Semaphores

- Incorrect use of semaphore operations:
 - signal (mutex) wait (mutex)
 - wait (mutex) ... wait (mutex)
 - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let **S** and **Q** be two semaphores initialized to 1

P_0 <pre>wait(S); wait(Q); ... signal(S); signal(Q);</pre>	P_1 <pre>wait(Q); wait(S); ... signal(Q); signal(S);</pre>
---------------------------------------------------------------------------------	---------------------------------------------------------------------------------

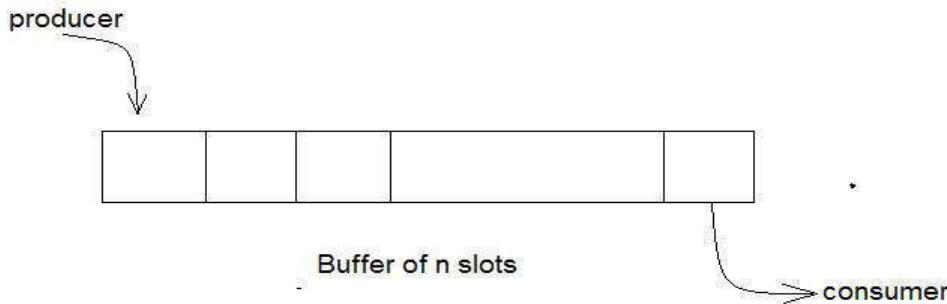
- **Starvation – indefinite blocking**
 - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - Solved via **priority-inheritance protocol**

Priority Inheritance Protocol

- When several tasks are waiting for the same critical resource, the task which is currently holding this critical resource is given the highest priority among all the tasks which are waiting for the same critical resource.
- Now after the lower priority task having the critical resource is given the highest priority then the intermediate priority tasks can not preempt this task. This helps in avoiding priority inversion.
- When the task which is given the highest priority among all tasks, finishes the job and releases the critical resource then it gets back to its original priority value (which may be less or equal).
- It allows the different priority tasks to share the critical resources.

- Consider the scenario with three processes **P1**, **P2**, and **P3**.
 - **P1** has the highest priority, **P2** the next highest, and **P3** the lowest.
- Assume that **P3** is holding semaphore **S** and that **P1** is waiting for **S** to be released
- Assume that **P2** is assigned the CPU and preempts **P3**
 - **P3** is still holding semaphore **S**
 - **P1** is waiting for **S** to be released
- What has happened is that **P2** - a process with a lower priority than **P1** - has indirectly prevented **P3** from gaining access to the resource.
- To prevent this from occurring, a **priority inheritance protocol** is used.

- n buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value n



Bounded-Buffer Problem (Cont.)

- The structure of the producer process

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty); //wait until empty > 0 and then decrement 'empty'  
    wait(mutex); //acquire lock  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex); //release a lock  
    signal(full); //increment full  
} while (true);
```

Bounded-Buffer Problem (Cont.)

□ The structure of the consumer process

```
do {  
    wait(full); // wait until full > 0 and then decrement 'full'  
    wait(mutex); // acquire the lock  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex); // release the lock  
    signal(empty); // increment 'empty'  
    ...  
    /* consume the item in next consumed */  
    ...  
} while (true);
```

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

- **solution**
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
 - Data set
 - Semaphore **rw_mutex** initialized to 1(semaphore)
 - Semaphore **mutex** initialized to 1 (mutex)
 - Integer **read_count** initialized to 0

- The structure of a writer process

```
do {  
    wait(rw_mutex);  
  
    ...  
    /* writing is performed */  
  
    ...  
    signal(rw_mutex);  
} while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);

    ...
    /* reading is performed */

    ...
    wait(mutex);
    read count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
 - Need both to eat, then release both when done
- In the case of 5 philosophers
 - Shared data
 - Bowl of rice (data set)
 - Semaphore **chopstick [5]** initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher i :

```
do {  
    wait (chopstick[i] );  
    wait (chopStick[ (i + 1) % 5 ] );  
        // eat  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5 ] );  
        // think  
} while (TRUE);
```

- What is the problem with this algorithm?

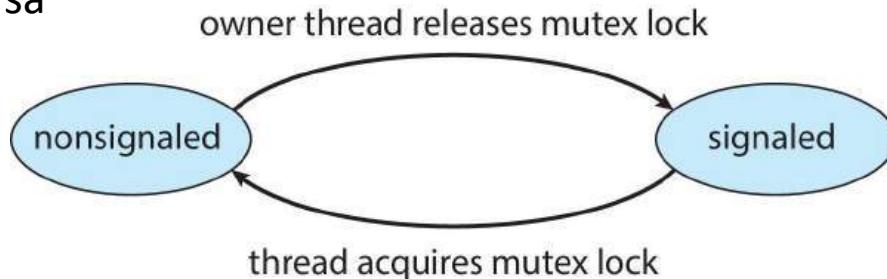
Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
 - Allow at most 4 philosophers to be sitting simultaneously at the table.
 - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section).
 - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
 - For reasons of efficiency, kernel ensures that a thread will never be preempted while holding a spinlock.
- Also provides **dispatcher objects** outside the kernel, to synchronize mutex locks, semaphores, events, and timers
 - **Events**
 - 4 An event acts much like a condition variable (i.e notify a waiting thread when a desired condition occurs)
 - Timers notify one or more thread when time expired

Windows Synchronization - Mutex dispatcher object

- Dispatcher objects may be in either a **signaled-state** (object available and a thread will not block) or a **non-signaled state** (object not available, thread will block)
- A Relationship exists between the state of a dispatcher object and the state of a thread.
- State of a thread changes from ready to waiting and vice-versa



- Linux:
 - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
 - Version 2.6 and later, fully preemptive
- Linux provides:
 - Semaphores
 - atomic integers
 - spinlocks
 - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

- Atomic variables

atomic_t is the data type for atomic integer

- Consider the variables

```
atomic_t counter;  
int value;
```

<i>Atomic Operation</i>	<i>Effect</i>
atomic_set(&counter,5);	counter = 5
atomic_add(10,&counter);	counter = counter + 10
atomic_sub(4,&counter);	counter = counter - 4
atomic_inc(&counter);	counter = counter + 1
value = atomic_read(&counter);	value = 12

- Pthreads API is OS-independent, widely used on UNIX, Linux, and macOS
- It provides:
 - mutex locks
 - semaphores
 - condition variable
- Non-portable extensions include:
 - read-write locks
 - spinlocks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores (have actual names in the file system) can be shared by multiple unrelated processes
- Unnamed semaphores can be used only by threads belonging to the same process.

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor (A high-level abstraction that provides a convenient and effective mechanism for process synchronization) , POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;  
pthread_cond_t cond_var;  
  
pthread_mutex_init(&mutex, NULL);  
pthread_cond_init(&cond_var, NULL);
```

- Thread waiting for the condition **a == b** to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments (< a few 100 instructions)
 - Starts as a standard semaphore spin-lock
 - If lock held, and by a thread running on another CPU, spins
 - If lock held by non-run-state thread (i.e. the thread holding the lock is not currently in run state), block and sleep waiting for signal of lock being released

Solaris Synchronization (Cont.)

- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
 - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

- Multiprogramming environment: several processes compete to limited number of resources
- A Process is holding a resource(R1) and is waiting for the resources(R2).
- The resource R2 is held by another process..
- Waiting state of processes will not change, as the requested resource is held by the waiting process.
- This situation is called deadlock

System Model

- System consists of finite number of resources
- Resource types R_1, R_2, \dots, R_m
 - *Physical resources: CPU cycles, memory space, I/O devices, printer, tape drives.*
 - *Logical resources: semaphores, mutex locks, files.*
- Each resource type R_i has W_i instances.
- Ex: if the system has 2 CPU's then CPU has 2 instances
- Each process utilizes a resource as follows:
 - **Request :** Process makes request to the resource. Eg. system call like request(), open(), wait(), allocate() etc
 - **Use:** operates on these resources
 - **Release:** process releases the resources. Eg. Using a system call like release(), close(), signal(), free etc.
- Request and release of semaphore, acquire and release of lock on mutex

Example 1

- Consider a system with 3 CD RW drives.
- Suppose 3 processes(p0,p1,p2) are holding one drive each.
- What happens,
 - If a process p0 makes a request for one more drive

Example 2

- Consider a system with one printer one DVD drive.
- Process Pi is holding printer and process Pj is holding DVD drive.
- What happens if
 - Process Pi request DVD and Pj requests printer

Does dead lock occur in example 1 and 2?

- Data:
 - A semaphore **S1** initialized to 1
 - A semaphore **S2** initialized to 1

- Two processes P1 and P2

- P1:

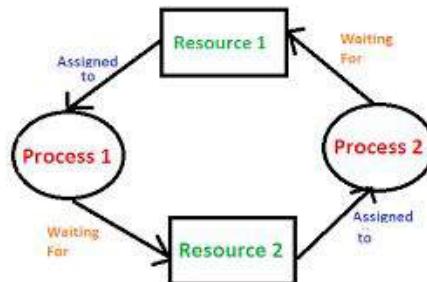
wait(s1)

wait(s2)

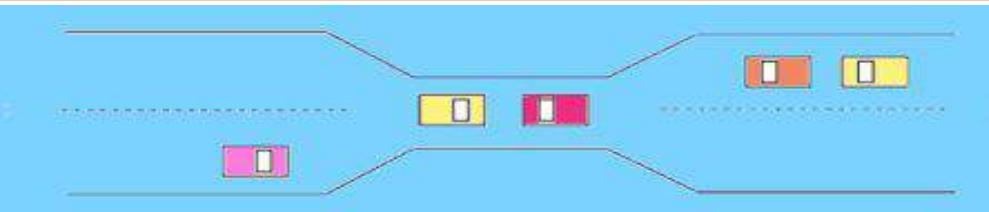
- P2:

wait(s2)

wait(s1)



Bridge crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible.
- Note – Most OSes do not prevent or deal with deadlocks

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource (sharable resources like Read-only files do not require mutually exclusive access and thus cannot be involved in a deadlock).
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .

- Deadlocks are described precisely with directed graphs called system resource-allocation graph

A graph consists of set of vertices V and a set of edges E .

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

Resource-Allocation Graph (Cont.)

A set of vertices V and a set of edges E .

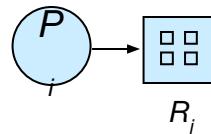
- Process



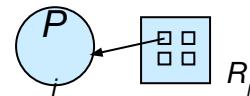
- Resource Type with 4 instances



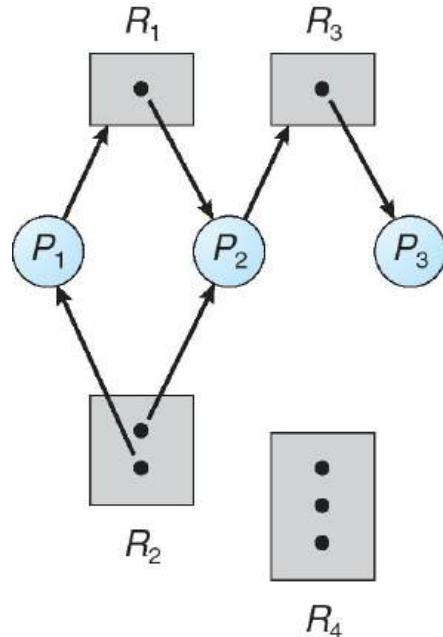
- P_i requests instance of R_j



- P_i is holding an instance of R_j



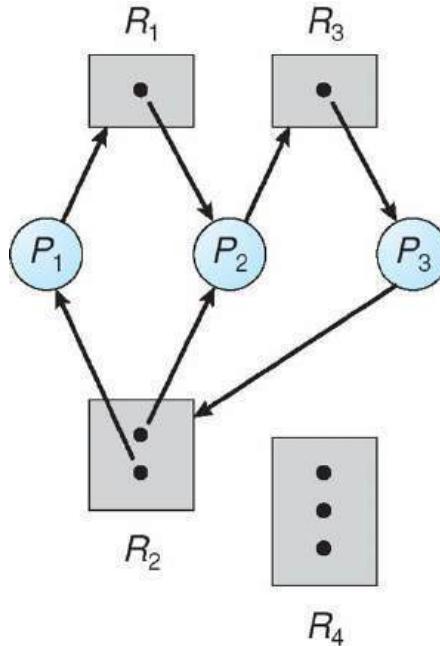
Example of a Resource-Allocation Graph



Set of process: P_1, P_2, P_3

Set of Resources: R_1, R_2, R_3

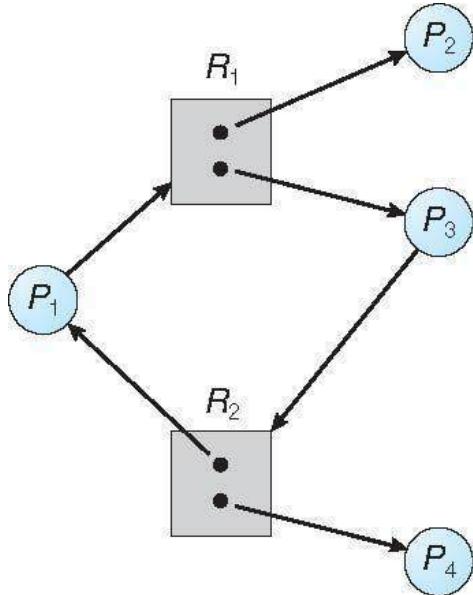
Resource-Allocation Graph With A Deadlock



$P_1 \square R_1 \square P_2 \square R_3 \square P_3 \square R_2 \square P_1$

$P_2 \square R_3 \square P_3 \square R_2 \square P_2$

Resource-Allocation Graph With A Cycle but No Deadlock



P1 → R1 → P3 → R2 → P1

- If graph contains no cycles \Rightarrow no deadlock
- If graph contains a cycle \Rightarrow
 - if only one instance per resource type, then deadlock
 - if several instances per resource type, the system may or may not be in a deadlocked state

Methods for Handling Deadlocks

- Generally speaking there are three ways of handling deadlocks:
 - Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
 - Allow the system to enter into deadlocked state, detect it, and recover.
 - Ignore the problem all together and pretend that deadlocks never occur in the system.

4 Necessary conditions for deadlock to occur

- **Mutual Exclusion**
 - At least one resource must be non sharable
 - Ex. printers and tape drives, mutex locks
 - Sharable resources do not require mutual exclusion
 - Ex . Read-only files
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
 - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
 - Low resource utilization; starvation possible

- **No Preemption –**
 - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
 - Preempted resources are added to the list of resources for which the process is waiting
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait –**
 - Each resource will be assigned with a numerical number.
 - A process can request the resources increasing/decreasing order of numbering.

- Ex., if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.
- Resources={R1,R2,.....Rm}
- Resources are assigned unique number
- Each process request resource in increasing order enumeration
- we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers
- Protocol 1: Process makes request for R_i and then for R_j . Resources R_j request is allowed if and only if $F(R_j) > F(R_i)$.
- Protocol 2: Process requesting an instance of resource type R_j , must have released any resource R_i , such that $F(R_i) \geq F(R_j)$.
- If these protocols are used then circular wait will not exist.

If these two protocols are used, then the circular-wait condition cannot hold.

Proof by contradiction:

- We can demonstrate this fact that by assuming that circular wait condition cannot hold
- Consider set of processes $P=\{P_0, P_1, \dots, P_n\}$
- Let us consider process P_0 is waiting for resource held by P_1 , P_1 waiting for P_2, \dots, P_{n-1} is waiting for resource held by P_n , P_n is waiting for resources held by P_0 .
- Generalizing this, Process P_i is waiting for resources R_i , R_i is held by P_{i+1} and it is making request for R_{i+1}
- We must have $F(R_i) < F(R_{i+1})$
- But this condition means that $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$.
- By transitivity, $F(R_0) < F(R_0)$, which is impossible
- Therefore no circular wait.

Example

- $F(\text{tape drive})=1$
- $F(\text{disk drive})=5$
- $F(\text{printer})=12$
- $F(\text{tape drive}) < F(\text{printer})$

Circular Wait

- Ex., if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.
- Invalidating the circular wait condition is most common.
- Assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:

`first_mutex = 1`

`second_mutex = 5`

code for **`thread_two`** could not be written as follows:

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

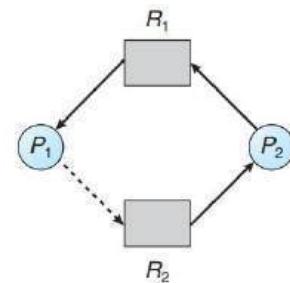
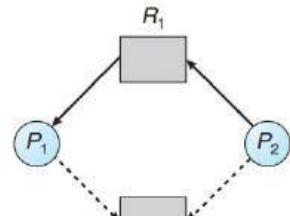
Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount) {  
    mutex lock1, lock2;  
  
    lock1 = get_lock(from);  
    lock2 = get_lock(to);  
  
    acquire(lock1);  
    acquire(lock2);  
  
    withdraw(from, amount);  
    deposit(to, amount);  
  
    release(lock2);  
    release(lock1);  
}
```

Transactions 1 and 2 execute concurrently.
Transaction 1 transfers \$25 from account A to account B, and Transaction 2 transfers \$50 from account B to account A

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

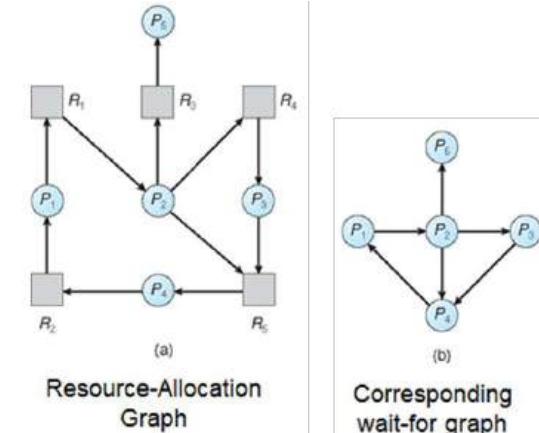
- In a resource allocation graph, a claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. This edge is represented in the graph by a dashed line.
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph



OPERATING SYSTEMS

Resource-Allocation Graph and Wait-for Graph

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.
- We obtain wait-for graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from P_i to P_j implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q .
- A **deadlock exists** in the system if and only if the **wait-for graph contains a cycle**.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations.



- **Available:** A vector of length m indicates the number of available resources of each type. If Available[j] = k, then k instances of resource type R_j are available
- **Max:** An $n \times m$ matrix defines the maximum demand of each process. If Max[i][j] = k, then process P_i may request at most k instances of resource type R_j .
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process P_i is currently allocated k instances of resource type R_j
- **Request:** An $n \times m$ matrix indicates the current request of each process. If Request [i][j] = k, then process P_i is requesting k more instances of resource type R_j .

1. Let **Work** and **Finish** be vectors of length m and n , respectively Initialize:

- (a) **Work = Available**
- (b) For $i = 1, 2, \dots, n$, if $\text{Allocation}_i \neq 0$, then
Finish[i] = false; otherwise, **Finish[i] = true**

2. Find an index i such that both:

- (a) **Finish[i] == false**
- (b) **Request_i ≤ Work**

If no such i exists, go to step 4

3. $\text{Work} = \text{Work} + \text{Allocation}_i$,
 $\text{Finish}[i] = \text{true}$
go to step 2
4. If $\text{Finish}[i] == \text{false}$, for some i , $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $\text{Finish}[i] == \text{false}$, then P_i is deadlocked. If $\text{Finish}[i] == \text{true}$ for all i , then the system is in a safe state

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4 ; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0 :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2		3	0	3	0	0	0		
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = true$ for all i

Example of Detection Algorithm (Cont.)

- P_2 requests an additional instance of type C

Request

	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?

- Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes' requests
- Deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4

OPERATING SYSTEMS

Detection-Algorithm Usage



- When, and how often, to invoke depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
 - 4 one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

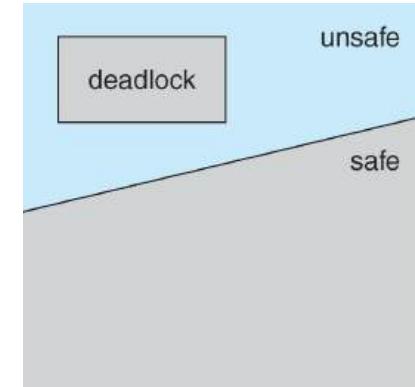
- Deadlock-prevention algorithms, prevent deadlocks by limiting how requests can be made.
- The limits ensure that at least one of the necessary conditions for deadlock cannot occur.
- Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.
- **Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.
- With this additional knowledge of complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock
- To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process

- The simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need.
- Given this apriori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.
- The resource allocation **state** is defined by the number of available and allocated resources and the maximum demands of the processes.

- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- More formally, a system is in a safe state only if there exists a **safe sequence**.
- A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i , the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j , with $j < i$.
- In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished.
- When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be **unsafe**

Safe, Unsafe and Deadlock States

- A safe state is not a deadlocked state.
- Conversely, a deadlock state is an unsafe state.
- Not all unsafe states are deadlocks.
- An unsafe state may lead to a deadlock.
- As long as the state is safe, the OS can avoid unsafe states.
- In an unsafe state, the OS cannot prevent processes from requesting resources in such a way that a deadlock occurs.
- The behavior of processes controls unsafe states.
- If a system is in safe state no deadlocks
- If a system is in unsafe state possibility of deadlock
- Goal for Avoidance ensure that a system will never enter an unsafe state.



Example - Safe, Unsafe and Deadlock States

- Consider a system with twelve magnetic tape drives and three processes: P_0 , P_1 , and P_2 .
- Process P_0 requires ten tape drives, process P_1 may need as many as four tape drives, and process P_2 may need up to nine tape drives.
- Suppose that, at time t_0 , process P_0 is holding five tape drives, process P_1 is holding two tape drives, and process P_2 is holding two tape drives. (Thus, there are three free tape drives.)
- At time t_0 , the system is in a safe state. The sequence $\langle P_1, P_0, P_2 \rangle$ satisfies the safety condition. Process P_1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process P_0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process P_2 can get all its tape drives and return them
- At time t_1 , process P_2 requests and is allocated one more tape drive. The system is no longer in a safe state.
- Since process P_0 is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process P_2 may request six additional tape drives and have to wait, resulting in a deadlock

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

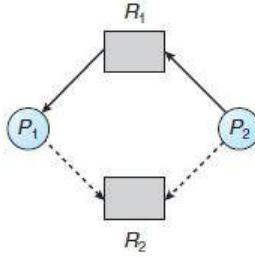
Deadlock avoidance algorithms

- Avoidance algorithms ensure that the system will never deadlock.
- The idea is simply to ensure that the system will always remain in a safe state.
- Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
- The request is granted only if the allocation leaves the system in a safe state.
- In this scheme, if a process requests a resource that is currently available, it may still have to wait.
- Thus, resource utilization may not be optimal

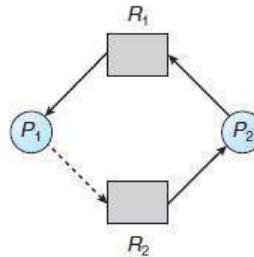
Resource Allocation Graph algorithm

- In addition to the request and assignment edges in a resource allocation graph, we introduce a new type of edge, called a **claim edge**
- A claim edge $Pi \rightarrow Rj$ indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- When process Pi requests resource Rj , the claim edge $Pi \rightarrow Rj$ is converted to a request edge. Similarly, when a resource Rj is released by Pi , the assignment edge $Rj \rightarrow Pi$ is reconverted to a claim edge $Pi \rightarrow Rj$.
- The resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph.
- Now suppose that process Pi requests resource Rj . The request can be granted only if converting the request edge $Pi \rightarrow Rj$ to an assignment edge $Rj \rightarrow Pi$ does not result in the formation of a cycle in the resource-allocation graph.

Resource Allocation Graph algorithm example



Resource-allocation graph for deadlock avoidance.



An unsafe state in a resource-allocation graph.

- Consider the above resource-allocation graph.
- Suppose that P_2 requests R_2 . Although R_2 is currently free, we cannot allocate it to P_2 , since this action will create a cycle in the graph below.
- A cycle indicates that the system is in an unsafe state.
- If P_1 requests R_2 , then a deadlock will occur.

Banker's algorithm

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
- Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system.
- We need the following data structures, where n is the number of processes in the system and m is the number of resource types

Banker's algorithm data structures

- **Available** – A vector of length m indicates the number of available resources of each type. If $\text{Available}[j]$ equals k , then k instances of resource type R_j are available.
- **Max.** - An $n \times m$ matrix defines the maximum demand of each process. If $\text{Max}[i][j]$ equals k , then process P_i may request at most k instances of resource type R_j .
- **Allocation** - An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $\text{Allocation}[i][j]$ equals k , then process P_i is currently allocated k instances of resource type R_j .
- **Need** - An $n \times m$ matrix indicates the remaining resource need of each process. If $\text{Need}[i][j]$ equals k , then process P_i may need k more instances of resource type R_j to complete its task.
- $\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$.
- Treat each row in the matrices **Allocation** and **Need** as vectorsThe vector Allocation_i specifies the resources currently allocated to process P_i ; the vector Need_i specifies the additional resources that process P_i may still request to complete its task
- These data structures vary over time in both size and value.

Banker's algorithm for determining whether or not a system is in a safe state

1. Let Work and Finish be vectors of length m and n , respectively.

Initialize $\text{Work} = \text{Available}$ and $\text{Finish}[i] = \text{false}$ for $i = 0, 1, \dots, n - 1$.

2. Find an index i such that both

- a. $\text{Finish}[i] == \text{false}$

- b. $\text{Need}_i \leq \text{Work}$

If no such i exists, go to step 4

3. $\text{Work} = \text{Work} + \text{Allocation}_i$

$\text{Finish}[i] = \text{true}$

Go to step 2.

4. If $\text{Finish}[i] == \text{true}$ for all i , then the system is in a safe state.

- This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

Banker's algorithm for determining whether requests can be safely granted

Let Request_i be the request vector for process P_i . If $\text{Request}_i[j] == k$, then process P_i wants k instances of resource type R_j .

When a request for resources is made by process P_i , the following actions are taken:

1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $\text{Request}_i \leq \text{Available}$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$$\text{Available} = \text{Available} - \text{Request}_i ;$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i ;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i ;$$

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources.

If the new state is unsafe, then P_i must wait for Request_i , and the old resource-allocation state is restored.

Banker's algorithm example

- Consider a system with five processes P_0 through P_4 and three resource types A , B , and C .
- Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances.
- Suppose that, at time T_0 , the following snapshot of the system has been taken

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

- Sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies the safety requirement. Hence, we can immediately grant the request of process P_1 .
- A request for $(3,3,0)$ by P_4 cannot be granted, since the resources are not available.
- A request for $(0,2,0)$ by P_0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

- A signal is used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received either synchronously or asynchronously depending on the source of and the reason for the event being signaled
- All signals, whether synchronous or asynchronous, follow the same pattern:
 1. A signal is generated by the occurrence of a particular event.
 2. The signal is delivered to a process.
 3. Once delivered, the signal must be handled.

Signals – synchronous and asynchronous

- Examples of synchronous signal include illegal memory access and division by 0.
- If a running program performs either of these actions, a signal is generated.
- Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).
- When a signal is generated by an event external to a running process, that process receives the signal asynchronously.
- Examples of such signals include terminating a process with specific keystrokes (such as <control><C>) and having a timer expire.
- Typically, an asynchronous signal is sent to another process.

- A signal may be ***handled*** by one of two possible handlers:
 1. A default signal handler
 2. A user-defined signal handler
- Every signal has a **default signal handler** that the kernel runs when handling that signal.
- This default action can be overridden by a **user-defined signal handler** that is called to handle the signal.
- Signals are handled in different ways. Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.

- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process.
- Delivering signals is more complicated in multithreaded programs, where a process may have several threads.
- In general, the following options exist:
 1. Deliver the signal to the thread to which the signal applies.
 2. Deliver the signal to every thread in the process.
 3. Deliver the signal to certain threads in the process.
 4. Assign a specific thread to receive all signals for the process.

- The method for delivering a signal depends on the type of signal generated.
- For example, synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process.
- However, the situation with asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (<control><C>) should be sent to all threads.
- The standard UNIX function for delivering a signal is

kill(pid t pid, int signal)

- This function specifies the process (pid) to which a particular signal (signal) is to be delivered

- Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block.
- An asynchronous signal may be delivered only to those threads that are not blocking it.
- Because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it.
- POSIX Pthreads provides the following function, which allows a signal to be delivered to a specified thread (tid):

`pthread kill(pthread t tid, int signal)`

Signals in windows

- Windows does not explicitly provide support for signals, it allows to emulate them using **asynchronous procedure calls (APCs)**.
- The APC facility enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event.
- An APC is roughly equivalent to an asynchronous signal in UNIX
- The APC facility is more straightforward, since an APC is delivered to a particular thread rather than a process

- A signal is a kind of software interrupt, used to announce asynchronous events to a process
- **SIGINT** is a signal generated when a user presses Control-C. This will terminate the program from the terminal.
- **SIGALRM** is generated when the timer set by the alarm function goes off.
- **SIGABRT** is generated when a process executes the abort function.
- **SIGSTOP** tells LINUX to pause a process to be resumed later.
- **SIGCONT** tells LINUX to resume the process paused earlier.
- **SIGSEGV** is sent to a process when it has a segmentation fault.
- **SIGKILL** is sent to a process to cause it to terminate at once.