

Client Server and Parallel Programming

31666

Spring 2013, Ort Braude College
Electrical Engineering Department



Course Program

- **Lecturer:** Dr. Samy Zafrany
- **Credits:** 5.0
- **Hours:** 3 lecture, 2 laboratory
- **Grade Composition:**
 - 20% - mid-term exam
 - 30% - laboratory projects
 - 50% - final exam
- **Prerequisites:** 31616 (Programming)

Course Web Site

tinyurl.com/samyz/cliserv/index.html

This is a temporary location until we move
To the college Moodle system

Slides and most figures and images are based
on the Slides of Tanenbaum Book:

**Computer Networks, Fourth Edition,
Andrew S. Tanenbaum, Prentice Hall 4th
Edition, Teacher Complimentary Materials**

Course Description

- Client/server application architecture
- Interface, Protocols, Basic Networking Concepts (TCP/IP, UDP) and basic networking tools
- Socket programming
- Internet, WWW, SQL, and client/server systems
- Multitasking, multithreading, and distributed programming
- Database systems, distributed systems, distributed programming
- Client technologies, languages and tools
- Server technologies, languages and tools
- Security and social issues of client/server systems.

Course Outline

- Client/Server systems overview: www client/server, email, ftp, File Server (NFS), DBMS, SQL, RPC
- Networking concepts: protocols, TCP/IP, UDP, MIME, POP, SMTP, DNS, HTML, HTTP, XML
- Networking concepts: OSI model
- Operating systems, processes, and threads Overview. Multithreading models. Threading issues.
- Socket Programming. Synchronous vs. Asynchronous socket calls.
- Networking testing tools: ping, nslookup, ipconfig, traceroute, netstat
- Distributed system structures. Network Structure. Network Topologies. Communication Structure. Communication Protocols.
- Client/Server system design: chat client/server, simple DBMS client/server, Poker game client/server
- Client/Server system implementation: chat client/server, simple DBMS client/server, Poker game client/sever
- Communication Security. Social issues. Cryptography. SSL.

Lab Projects

- Multi processing and multithreading (parallel programming)
- File system search/indexing using single process, multiple processes, and multithreading
- Client communication with server
- Multiple clients communicating with server (Chat server, simple DBMS, Poker game server)
- RPC client/server
- Implement a simple distributed parallel algorithm

Expected Learning Outcomes

- Students will get familiar with basic networking concepts, the basic structure and organization of networking
- Common types of networking paradigms, and common Internet applications and protocols
- Particular emphasis will be put on the prevalent client/server model, and its associated parallel programming computing methods
- Multitasking, multithreading, and distributed programming
- Ability to apply solid engineering principles and methods in building network-aware applications.

Bibliography

- *Silberschatz and Galvin*. Operating Systems Concepts. 8th edition, 2008, John Wiley & Sons, Inc.
- *Andrew S. Tanenbaum*. Computer Networks, 5th Edition, 2010, Prentice Hall.
- *W. Richard Stevens, Bill Fenner, Andrew Rudoff*. UNIX network programming, 3rd edition, 2003, Prentice Hall.
- *Allen B. Downey*. Think Python, O'Reilly 2012, <http://www.greenteapress.com/thinkpython>
- *Mark Pilgrim*. Dive into Python, Apress 2004, <http://www.diveintopython.net>
- *John Goerzen, Brandon Rhodes*. Foundations of Python Network Programming. 2nd Edition, 2010, Apress.
- www.python.org

Software

- All needed software should be downloaded from
 - <https://dl.dropbox.com/u/60773652/PYTHON/index.html>
- Into a personal flash drive (diskonkey)
 - at least 2GB drive is needed
- All software can be executed from the flash drive on any standard Windows PC
- So you can do all your coding work at home and everywhere you have an access to a windows PC
- We may however need a session or two in the College Linux labs

Computer Networks

- The old model of a single computer serving all of the organization's computational needs has been replaced by one in which a large number of separate but interconnected computers do the job.
- **"computer network"** is a collection of autonomous computing devices interconnected by a single technology
- Connection is achieved by:
 - Copper wires (Ethernet cables)
 - Fiber optics
 - Microwaves
 - Infrared,
 - Communication satellites
- Computing devices: personal computers, tablets, smart phones, routers, blade servers, car controllers, televisions, refrigerators, cameras, ewatches, hard drive controllers, robot systems (unmanned aerial vehicle), etc.

Goals of Networking

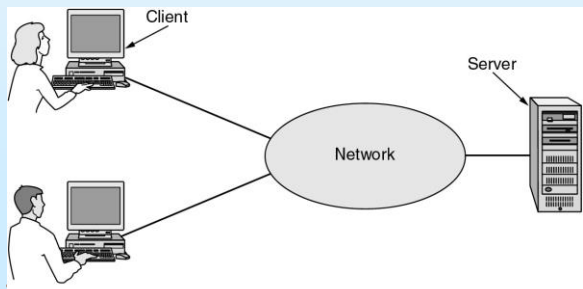
- Resource and load sharing and balancing
 - Programs do not need to run on a single machine
 - Files can span several disks (even on different continents – Hadoop)
 - Reduced cost
 - Several machines can share printers, tape drives, etc.
- Reliability & Redundancy:
 - If a machine goes down, another takes over
 - If a file or disk is damaged, data can be recovered
- Social Connectivity: mail, chat, messages, video, multimedia business, games, recreation (YouTube, Facebook, Twitter, Steam)
- Business applications: DB sharing, e-commerce, m-commerce (Amazon, eBay), Banking, Stock market, Sensor networks
- Mobile applications: tablets, smart phones, VOIP
- Scientific applications
 - knowledge bases
 - distributed computing
 - shared information systems, telelearning (education)

Computer Network & Distributed System

- In a distributed system, a collection of independent computers appears to its users as a single coherent system.
- In a computer network, users are exposed to the actual machines
 - If the machines have different hardware and different operating systems, that is fully visible to the users
 - If a user wants to run a program on a remote machine, he has to log onto that machine and run it there.
- In effect, a distributed system is a software system built on top of a network
- A well-known example of a distributed system is the **World Wide Web**. It runs on top of the Internet and presents a model in which everything looks like a document (Web page).

Client-Server System

- network architecture in which two computers are connected in such a way that one computer (the client) sends service requests to another computer (the server).
- Examples: WWW, Email, Waze
- Usually, the server is a powerful computer to which many less powerful personal computers or workstations (clients) are connected. The clients run programs and access data that are stored on the server.
- Usually on distant locations but can be also on the same machine

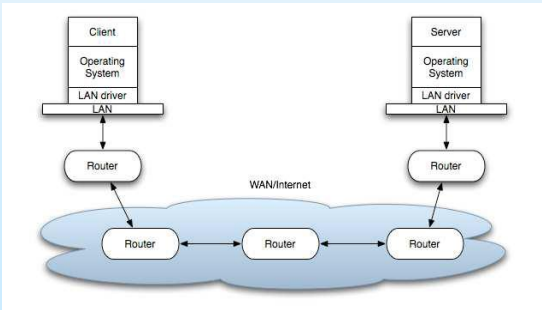
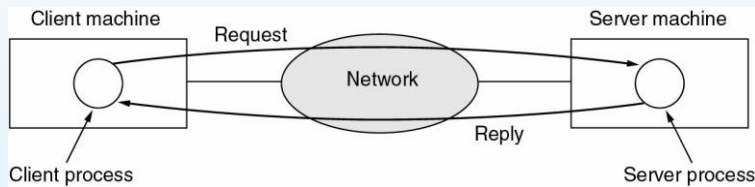


Client Server Programming

13

Client Server Data Flow

The client-server model involves requests and replies.

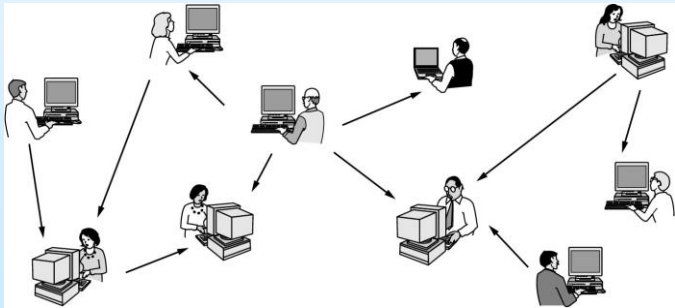


Client Server Programming - Slide Figures /quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

14

Peer-to-Peer System

- In peer-to-peer system there are no fixed clients and servers
- Any node can be sometimes a client and sometimes a server
- Examples: Napster, Kazaa, Emule, BitTorrent (content exchange)
- DEC president, Ken Olsen, 1977: "There is no reason for any individual to have a computer in his home."
 - **Digital Equipment Corporation** no longer exists



Some forms of e-commerce

Tag	Full name	Example
B2C	Business-to-consumer	Ordering books on-line
B2B	Business-to-business	Car manufacturer ordering tires from supplier
G2C	Government-to-consumer	Government distributing tax forms electronically
C2C	Consumer-to-consumer	Auctioning second-hand products on-line
P2P	Peer-to-peer	File sharing

Network Hardware

- Personal Area Networks (PAN)
- Local Area Networks (LAN)
- Metropolitan Area Networks (MAN)
- Wide Area Networks (WAN)
- Wireless Networks (LAN/WiFi)
- Home Networks (LAN/WiFi)
- Internetworks

Networks Classification

- Network are usually classified according to transmission technology and Scale
- there are two types of transmission technology that are in widespread use:
 - **broadcast** links
 - **point-to-point** links.
- **Broadcast network**: the communication channel is shared by all the machines on the network; packets sent by any machine are received by all the others
- **Point-to-point network**: shortest routes between two peers are used for communications

Interconnected Processors by Scale

Interprocessor distance	Processors located in same	Example
1 m	Square meter	Personal area network
10 m	Room	
100 m	Building	Local area network
1 km	Campus	
10 km	City	Metropolitan area network
100 km	Country	Wide area network
1000 km	Continent	
10,000 km	Planet	The Internet

Personal Area Network (PAN)



- (a) Wired connection

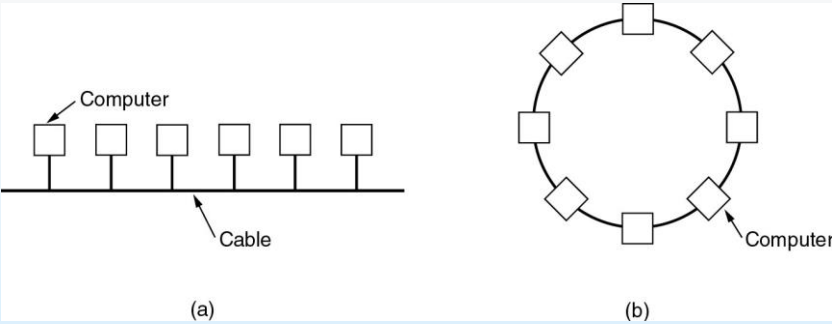
(b) Bluetooth configuration

(c) Wireless connection
- (a) Wireless keyboard/mouse/headset

(b) Wireless Printers

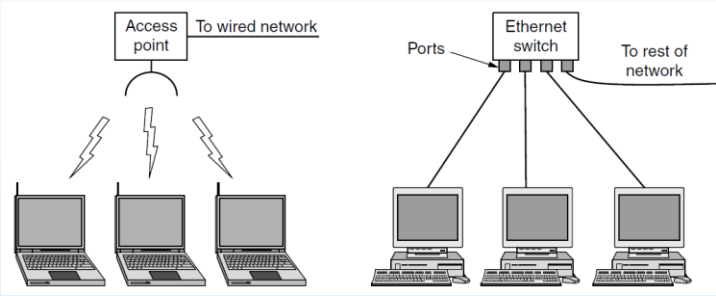
(c) External disks

Local Area Network (LAN)



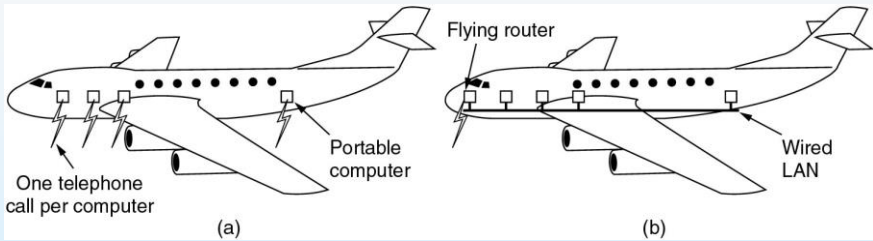
Two broadcast networks
(a) Bus
(b) Ring

Wireless and wired LANs



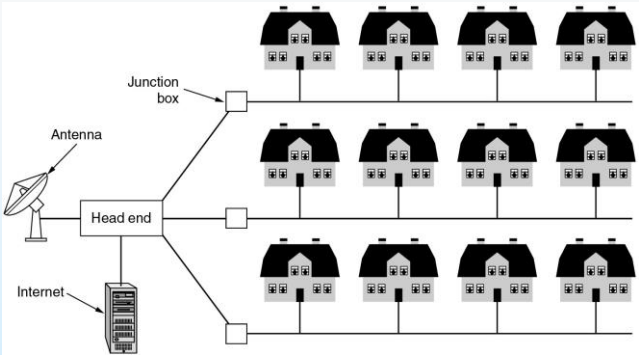
Wireless LAN: IEEE 802.11 (WiFi)
(a) 1-100 Mbps, 10 Gbps
(b) Copper wires, optical fibers
- faster than wireless LAN
(c) 802.3 (Ethernet) most popular LAN

Flying LAN



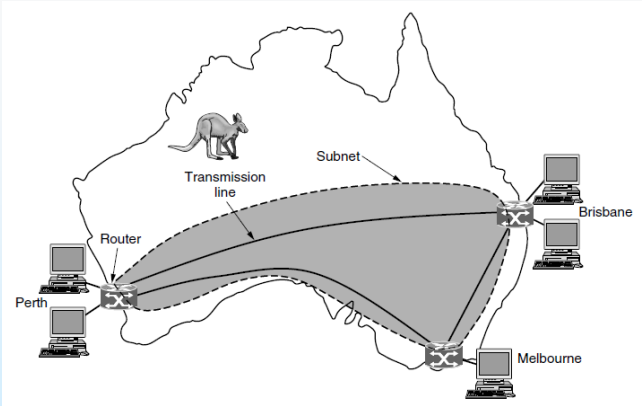
- (a) Individual mobile computers
- (b) Tablets, smartphones
- (c) Other small factor devices

Metropolitan Area Networks (MAN)



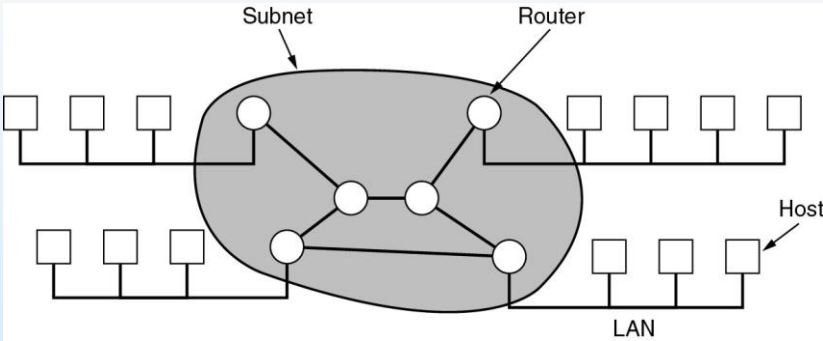
- (a) A metropolitan area network based on cable TV
- (b) New MAN: IEEE 802.16 (WiMax)
 - Worldwide Interoperability for Microwave Access
- (c) Related standards: GSM, 3G (3rd generation of mobile technology)

Wide Area Networks (WAN)



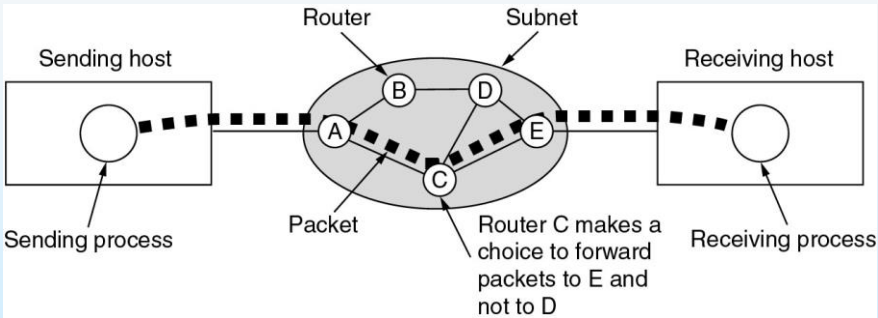
WAN that connects three branch offices in Australia
Transmission lines: copper, optical fiber, radio links
Switching elements: computers that connect two or more transmission lines (routers) - **internetworks**

Wide Area Networks (WAN)



Relation between hosts on LANs and the subnet.

Wide Area Networks (2)



A stream of packets from sender to receiver.

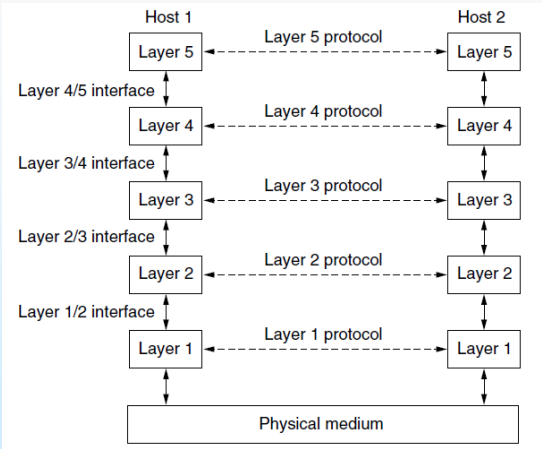
Network Software

- Communication Protocol Hierarchies
- Design Issues for the Layers (OSI Model)
- Connection-Oriented and Connectionless Services
- Service Primitives
- The Relationship of Services to Protocols

Protocol Hierarchies

- “Abstraction—the hiding of details behind a well-defined interface—is the fundamental tool used by system designers to manage complexity”
Larry L. Peterson and Bruce S. Davie, Computer Networks
- To reduce design complexity networks are organized as a stack of layers
- The purpose of each layer is to offer certain services to the higher layers while shielding those layers from the details of how the offered services are actually implemented
- AKA: information hiding, abstract data types, data encapsulation, and object-oriented programming
- Conversation between layer n on one machine with layer n on another machine: the rules and conventions used in this conversation are collectively known as the **layer n protocol**

Layers, protocols, and interfaces

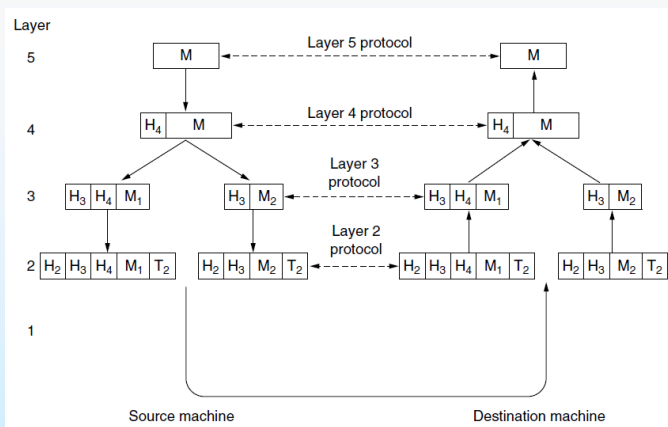


Real data is transferred only at the physical layer!
All other dotted lines are virtual!

Network Architecture

- A set of layers and protocols is called a **network architecture**
- Neither the details of the implementation nor the specification of the interfaces is part of the architecture
- A list of the protocols used by a certain system, one protocol per layer, is called a **protocol stack**
- **Typical flow:**
 - ◆ A message, M , is produced by an application process running in layer 5 and given to layer 4 for transmission
 - ◆ Layer 4 puts a **header** in front of the message to identify the message and passes the result to layer 3
 - ◆ The header includes control information, such as address/port, to allow layer 4 on the destination machine to deliver the message
 - ◆ Other examples of control information used in some layers are sequence numbers, sizes, and times
 - ◆ layer 3 must break up the incoming messages into smaller units, packets, prepending a layer 3 header to each packet

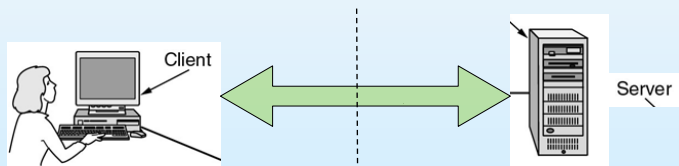
Communication Flow



- Layer 3 decides which lines to use and passes the packets to layer 2
- Layer 2 adds to each piece not only a **header** but also a **trailer**, and gives the resulting unit to layer 1 for physical transmission
- At the receiving machine the message moves upward, from layer to layer, with headers being stripped off as it progresses

Communication Protocol

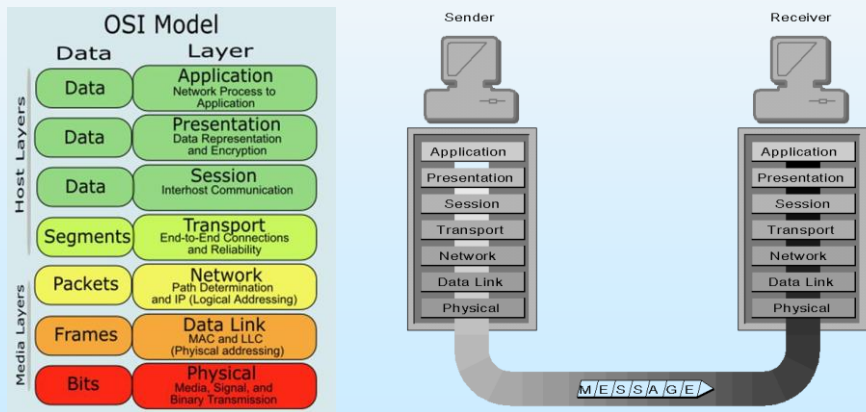
- Definition 1: A **protocol** is an agreement between the communicating parties on how communication is to proceed
- Definition 2: A **protocol** is a set of communication "rules" between two processes.
- Example: A "grades database query" protocol
 - ◆ (We may make a small project out of it later ...)



Client: HELLO	Server: READY
Client: NAME 051883261\n	Server: DAN HACKER\n
Client: GRADE MATH\n	Server: 87\n
Client: GRADE HISTORY\n	Server: 93\n
Client: END	Server: BYE

OSI Model

- Open Systems Interconnection (OSI)
- Proposed by the International Standards Organization (ISO)
- The OSI model has seven layers



Application Layer

- The closest layer to the user: Outlook, Explorer, Firefox, Skype (HTTP, POP, SMTP, FTP, TELNET).
- In this layer that a user interacts with the software application that does data transfer
- The main tasks:
 - ◆ Identify/authenticate the user who wants to communicate
 - ◆ determine whether the data and networks sources are available
 - ◆ synchronize communication between the two nodes

Presentation Layer

- Convert the data into a format that could be easily recognized by the application layers of other end users.
- For example: translation between ASCII and EBCDIC machines as well as between different floating point and binary formats. Integer size (16,32, or 64 bit?). Floating point representations.
- Compression/decompression, conversion, encryption/decryption, coding, decoding, etc.
- Converts the data obtained from the application layer into a format that can be easily identified by other network layers.

Session Layer

- In practice, this layer is often not used or services within this layer are sometimes incorporated into the transport layer
- Establishing, maintaining and terminating the connection between two end nodes (not used in TCP/IP)
- Controls the communication between the source user and the destination user and also decides the time of communication
- It determines one-way or two-way communications and manages the dialog between both parties; for example, making sure that the previous request has been fulfilled before the next one is sent
- Any error report related to application layer, presentation layer and session layer, are provided by this layer

Transport Layer

- Responsible for delivering the data or the messages between the two nodes
- Divide the data in packets at the sender side
- Re-assemble packets at the receiver side
- Third task: error free data transmission
 - ◆ Uses checksums for error correction or rejection
 - ◆ Drop corrupt packets and requests retransmission
- Fourth task: guarantee data integrity
 - ◆ Make sure all packets have arrived
- UDP, SPX, TCP are some of the protocols that operate on this layer with one exception: UDP is unreliable

Network Layer

- Provide switching technologies and routing technologies:
It is the *network* layer's job to figure out the *network* topology, handle routing and to prepare data for transmission
- Establishes the route between the sending and receiving nodes for data transmission (also known as virtual circuits)
- Encapsulation of transport data into network layer protocol data units
- Also responsible for handling errors, packet sequencing, controlling network congestion and addressing
- In short: this layer is responsible for the setting up the required network for transferring data from one node to other.

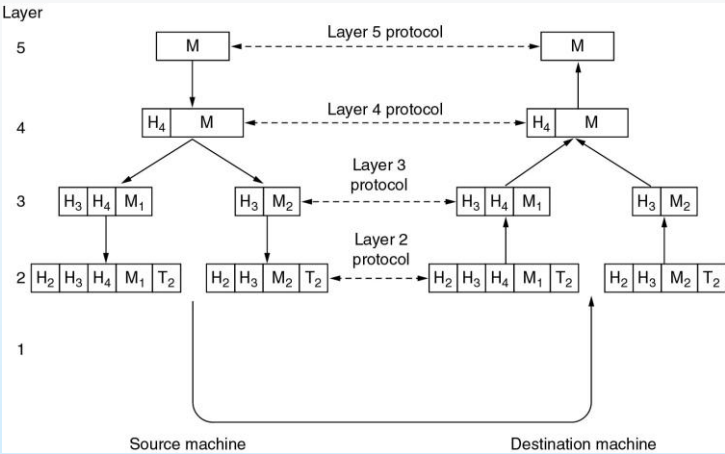
Data Link Layer

- Encoding and decoding of data frames into bits (as the physical layer may use waves or other type of media). At the receiving side: Collects a stream of bits into a larger aggregate called a *frame*.
- Segmentation of upper layer datagrams (packets) into frames in sizes that can be handled by the communications hardware
- Takes care of any errors in the physical layer (electricity presence, voltage drop, no power, connection, etc.)
- Provides reliable transit of the data through a physical network
- Synchronization of various physical devices that will transmit the data
- It makes sure that the frames are transferred in correct order and asks for retransmission in case of error
- The frame formatting issues such as stop and start bits, bit order, parity and other functions are handled here. Management of big-endian/little-endian issues are also managed at this layer.
- Usually implemented on Hardware (network interface card):

Physical Layer

- Deals with the physical components of a network
- Activation, maintenance and deactivation of various physical links that act in data transmission
- Electrical signals, voltage levels, cables, data transmission rates, etc., are some of the major elements defined by the physical layer
- It is also responsible for passing and receiving bytes from the physically connected medium
- Implemented on hardware (network interface card)

Information Flow



The peer processes in layer 4 (for example) conceptually think of their communication as being "horizontal," using the layer 4 protocol. Each one is likely to have procedures called something like *SendToOtherSide* and *GetFromOtherSide*, even though these procedures actually communicate with lower layers across the 3/4 interface, and not with the other side.

Design Issues - Accuracy

- Packet traveling through the network: there is a chance that some bits will be flipped, or even get lost, or new ones will be added:
 - ◆ fluke electrical noise
 - ◆ random wireless signals
 - ◆ hardware flaws
 - ◆ software bugs (and so on ...)
- Is it possible to detect and even fix these errors?
- Must separate between two targets:
 - **Error Detection**
 - ◆ Easy mechanisms for detecting errors (with very high probability)
 - **Error Correction**
 - ◆ This is possible but very costly (space, time, resources)

Design Issues - Reliability

- Finding a working path through a network:
 - ◆ Usually there are multiple paths between a source and destination
- In a large network, there may be broken links, hosts, and routers
- If the network is down in Germany: packets sent from London to Rome via Germany will not get through, but packets from London to Rome via Paris may get through ... ?
- A network should automatically detect the problem and make this decision. This topic is called **routing**. How this is done? We'll see later ...
- Not all communication channels preserve the order of packets sent on them, and packets can also get completely lost

Design Issues – Flow Control

- **Congestion:** how to keep a fast sender from swamping a slow receiver?
- Overloading of the network is called **congestion**: too many computers want to send too much traffic, and the network cannot deliver it all
- One strategy is for each computer to reduce its demand when it experiences congestion
- **Starvation:** fast receivers against slow senders (fast clients vs. slow server)
- **Quality of service** is the name given to mechanisms that reconcile these competing demands
- Applications: video streaming, VOIP, media recorders ("buffer overrun")
 - ◆ Balancing senders and receivers speeds in such cases is very crucial

Design Issues – Security

- Network must be secured by defending it against different kinds of threats:
- **Confidentiality:** prevent unauthorized access to information (snooping)
- **Authentication:** prevent someone from impersonating someone else (Phishing)
- **Integrity:** prevent surreptitious changes to messages:
"debit my account \$10" → "debit my account \$1000"
- Solution designs are heavily based on **cryptography**

Connection-Oriented and Connectionless Services

Connection-oriented	Service	Example
	Reliable message stream	Sequence of pages
	Reliable byte stream	Remote login
Connection-less	Unreliable connection	Digitized voice
	Unreliable datagram	Electronic junk mail
	Acknowledged datagram	Registered mail
	Request-reply	Database query

Connection-Oriented

- Connection is established, the sender, receiver, and subnet conduct a **negotiation** about the parameters to be used, such as
 - ◆ Maximum message size
 - ◆ Quality of service required, and other issues
- Typically, one side makes a proposal and the other side can accept it, reject it, or make a counter proposal.
- A **circuit** is another name for a connection with associated resources (after the telephone model ...)
- Reliability: do not lose data – e.g., the receiver acknowledge the receipt of each message
 - so the sender is sure that it arrived
- TCP – Transmission Control Protocol is connection oriented
- Text documents, email, image attachments

Connectionless Service

- In contrast to connection-oriented service, **connectionless** service is modeled after the postal system
- Each message (letter/package) carries the full destination address and each one is routed through the intermediate nodes inside the system independent of all the subsequent messages
- UDP – User Datagram Protocol – unreliable
- Unreliable (meaning not acknowledged) connectionless service is often called **datagram** service, in analogy with telegram (service, which also does not return an acknowledgement to the sender)
- Video streaming, Video conference, VOIP, Digital TV transmission (Idan+)

Co-existence of both kinds

- reliable communication may not be available in a given layer
- For example, Ethernet does not provide reliable communication. Packets can occasionally be damaged in transit
- It is up to higher protocol levels to recover from this problem. In particular, many reliable services are built on top of an unreliable datagram service. Second,
- Both reliable and unreliable communication usually coexist.

Connection-oriented Service Primitives

Primitive	Meaning
LISTEN	Block waiting for an incoming connection
CONNECT	Establish a connection with a waiting peer
RECEIVE	Block waiting for an incoming message
SEND	Send a message to the peer
DISCONNECT	Terminate a connection

- ❑ Minimal example of service primitives that provide a reliable byte stream
- ❑ A service is formally specified by a set of **primitives** (operations) available to user processes to access the service
- ❑ These primitives tell the service to perform some action or report on an action taken by a peer entity (usually as operating system calls)
- ❑ Modeled after the Berkeley socket interface

Service Primitives (2)

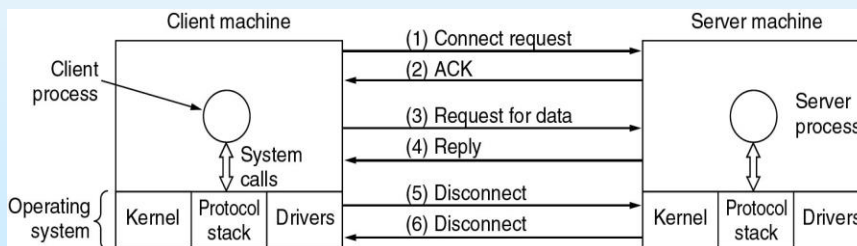
- **LISTEN** is usually implemented by a block system call - the server process is blocked until a request for connection appears
- **CONNECT** is usually implemented by a connection request to a server
 - ◆ The **CONNECT** call may need to specify the server's address
 - ◆ The operating system then typically sends a packet to the peer asking it to connect
- The client process is suspended until there is a response
- When the packet arrives at the server, the operating system sees that the packet is requesting a connection
 - ◆ It checks to see if there is a listener
 - ◆ If so it unblocks the listener (wake-up call)
 - ◆ The server process may accept the connection with the **ACCEPT** call
- This sends a response back to the client process to accept the connection

Service Primitives (3)

- Next step: **RECEIVE**
 - ◆ The server prepares to accept the first client request
 - ◆ The **RECEIVE** call blocks the server
- Then the client executes **SEND** to transmit its request (data or action) followed by the execution of **RECEIVE** by the server (and then blocks)
- The arrival of the request packet at the server machine unblocks the server so it can handle the request
- After it has done the work, the server uses **SEND** to return the answer to the client
- The arrival of this packet unblocks the client, which can now inspect the answer. If the client has additional requests, it can proceed immediately.
- When the client is done, it executes **DISCONNECT** to terminate the connection. Usually, a **DISCONNECT** is a blocking call, suspending the client and sending a packet to the server saying that the connection is no longer needed

Service Primitives (4)

- When the server gets the client disconnect packet, it also issues a server **DISCONNECT** of its own, acknowledging the client and releasing the connection
- When the server's packet gets back to the client machine, the client process is released and the connection is broken
- In a nutshell, this is how connection-oriented communication works:



The TCP/IP Reference Model

- TCP is Transmission Control Protocol.
- IP is Internet Protocol.
- Only 4 layers:

1	Application Layer
2	Transport Layer
3	Internet Layer
4	Link Layer (network)

The Tanenbaum Reference Model

- The model used in Tanenbaum book adds a Physical layer (page 48). Also used by others.
- But we will stick to the official TCP/IP model since the physical layer is out of the course scope

1	Application Layer
2	Transport Layer
3	Internet Layer
4	Link Layer (network)
5	Physical Layer

Layer 4: The Application Layer

- Higher-level protocols such as: TELNET, FTP, SMTP, DNS, HTTP, POP2, POP3.
- These are the protocols that are used by applications like MS internet explorer, Google Chrome, MS outlook, Skype, Waze, etc.
- This layer is essentially the same as the **OSI Model** layer 7

Layer 3: The Transport Layer (TCP / UDP)

- This layer implements layers 4, 5, and 6 of the **OSI model** (session, presentation, and transport)
- Handles full messages (long documents, multimedia, etc.)
- Nevertheless, in many cases **OSI layer 6** makes sense (encryption, compression, data representation) and used in analysis
- The most used protocols are: TCP, UDP (but there are additional 15 new ones)
- Usually implemented at the operating system kernel (Unix and Windows) (why?)

Layer 2: The Internet Layer (IP)

- Connectionless internetwork layer (IP Protocol)
- Packet-switching: blocks of data constrained to a fixed size
- permitting hosts to send packets into any network and have them travel independently to the destination, potentially on a different network.
- Implemented at the operating system, at routers hardware, gateways, bridges, etc.
- A computer can act sometimes as a router or a gateway, so the operating system includes special modules to handle network operations
- Major interface: `SEND_IP_PACKET`, `RECEIVE_IP_PACKET`

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

59

Layer 1: Link/network Layer (Ethernet/wireless)

- Almost everything below the internet layer is not defined in the TCP/IP reference model
- The network layer essentially performs the functions of the OSI physical and data link layers
- Usually implemented by network device drivers: Ethernet, Ring or Star card drivers (with the help of the device drivers of course)

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

60

OSI and TCP/IP Reference Models

	OSI	TCP/IP
7	Application	Application
6	Presentation	
5	Session	
4	Transport	
3	Network	Internet
2	Link	Link/Network
1	Physical	

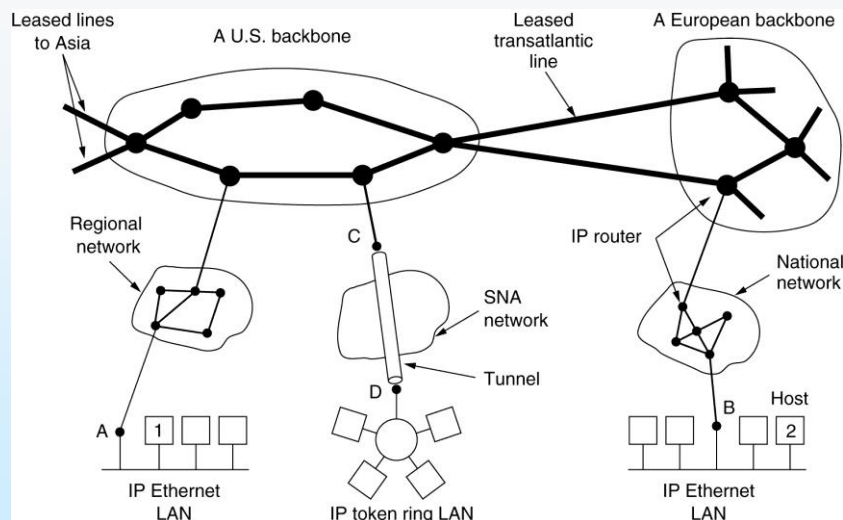
TCP/IP Family

- TCP/IP refers to an entire communication protocol family based on the
 - ◆ Transmission Control Protocol (TCP)
 - ◆ The Internet Protocol (IP)
- It defines protocols at the network layer and the transport layer
- The TCP/IP suite has six basic elements:
 - ◆ Applications
 - ◆ The Transmission Control Protocol (TCP)
 - ◆ The User Datagram Protocol (UDP)
 - ◆ The Internet Protocol (IP)
 - ◆ Auxiliary protocols like the Internet Control, Message Protocol (ICMP), and the Address Resolution Protocol (ARP).

TCP/IP Family: IP

- IP major role is to route packets from a process in one machine to another process at another machine (possibly the same machine)
- For that IP uses an IP address and a Port number
 - ◆ The port number determines the specific process to which the packet belongs
- When an application sends a data packet to another machine
 - ◆ IP determines to which network the packet should go
 - ◆ if necessary, IP routes the packet from one network to another
- IP figures out where to send a packet based on the IP address of the recipient
- At some hops, IP may fragment a large packet to smaller packets ("fragments") if that network cannot handle large packets (link with a smaller MTU - maximum transmission unit)

Internet: Collection of Subnetworks



The IP Protocol

- Packet delivery service (host-to-host).
- IP provides connectionless, unreliable delivery of IP datagrams.
- Connectionless: each datagram is independent of all others.
- Unreliable: there is no guarantee that datagrams are delivered correctly or at all.

IP Addressing (v4)

Every host on the internet is assigned a unique IP address which consists of 32 bits.

Example:

 |←----- 32 bits -----→|
 address = 11000111110010111001100000001010
The IP address consists of two parts: Network ID + Host ID

1-8	9-16	17-24	25-32	

Class A:	0nnnnnnn	hhhhhhhh	hhhhhhhh	0-127
Class B:	10nnnnnn	nnnnnnnn	hhhhhhhh	128-191
Class C:	110nnnnn	nnnnnnnn	nnnnnnnn	192-223
Class D:	1110mmmm	mmmmmmmm	mmmmmmmm	224-239
Class E:	11110rrr	rrrrrrrr	rrrrrrrr	240-247

n = network bit
h = host bit
m = multicast
r = reserved for future use

IP Addressing (v4)

- A Network ID is assigned to an organization by a global authority (**ICANN** - Internet Corporation for Assigned Names and Numbers)
- Host IDs are assigned locally by a system administrator or automatically by a DHCP server
- Both the Network ID and the Host ID are used for routing
- Very few organizations are assigned Class A addresses (USA military, government, Boeing, large banks, ...)
 - ◆ But they do not use all possible host ids
- Many universities and companies were assigned class B addresses, but most of them do not use more than 1000 or 2000 host ids (out of the 64K possible host ids).

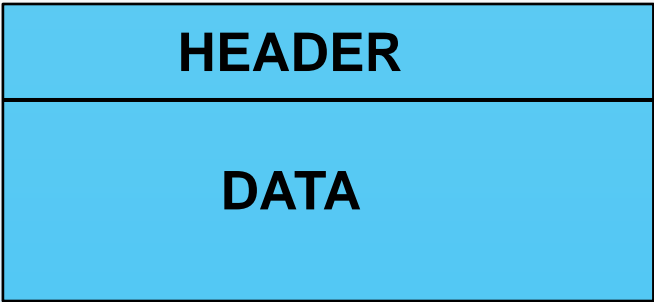
IP Addresses

- An IP address is assigned per network interface, not host!
- So a host that belongs to two networks must have two network interfaces and thus two IP addresses!

EXAMPLE

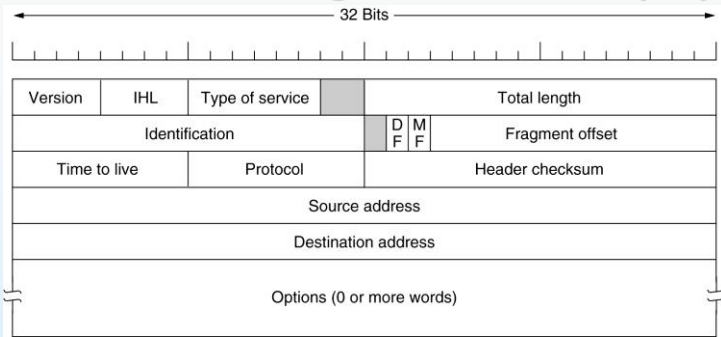
- The IP number of Netanya College Linux mail server, moon.netanya.ac.il is a 32 bits binary integer:
11000111110010111001100000001010
- It is better viewed 4 bytes:
11000111.11001011.10011000.00001010
- Even better as: **199.203.152.6**
- Since it starts in "110" it is a class C address, and therefore its network mask is: **11111111.11111111.11111111.00000000**
- The network number is 199.203.152.0.
- Broadcast address is 199.203.152.255.
- Broadcast mask is 255.255.255.255.

The IP Datagram Structure



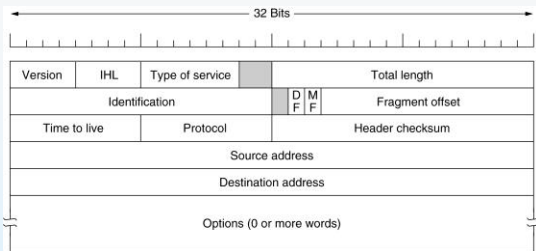
- Header length is 20 bytes minimum and 60 bytes maximum
- Packet size can range from 40 bytes to 64K bytes depending on networking software and networking hardware
- The data part is usually a small fragment of the total message which the TCP (or UDP) protocol is trying to transmit
- TCP and UDP are the drivers of the IP protocol

The IP Datagram Header (v4)



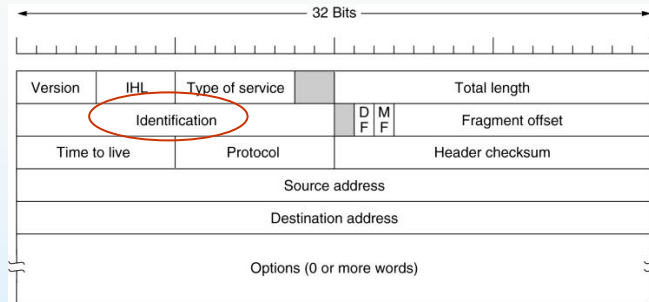
- Has a 20 bytes fixed part and a variable length optional part
- Version – IP Protocol Version (v4, v5, v6)
- IHL – (4 bits) The number of 32-bit words in the header (min=5W, max=15W). That is, the header can be at most 60 bytes!
- Total Length - total length of the datagram in bytes
 - ◆ size of the data = total length - header length"

Type of service (also called: Differentiated Services)



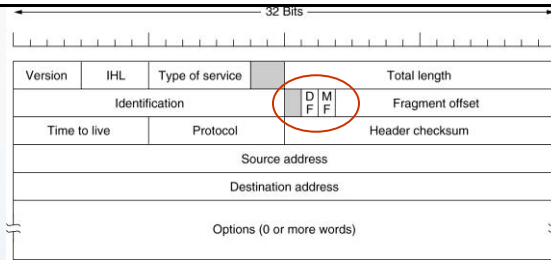
- Consists of 6 bits:
 - ◆ 1000 - minimize delay
 - ◆ 0100 - maximize throughput
 - ◆ 0010 - maximize reliability
 - ◆ 0001 - minimize monetary cost
 - ◆ The other two bits used to record congestion history but now used for VOIP
- This is a "hint" to the physical layer to which path to use
- Not supported in most implementations. Some implementations have extra fields in the routing table to indicate delay, throughput, reliability, and monetary cost.

Identification



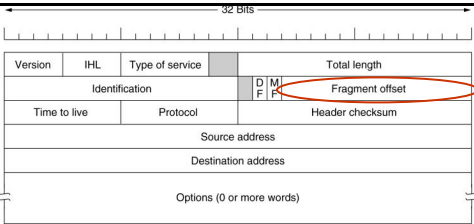
- Uniquely identifies the datagram
- Usually incremented by 1 each time a new datagram is sent
 - ◆ Puts a max limit on packet sequence: $2^{16} * (\text{packet_length}) \sim 4\text{G}$
- All fragments of a datagram contain the same identification value
- This allows the destination host to determine which fragment belongs to which datagram

FLAGS



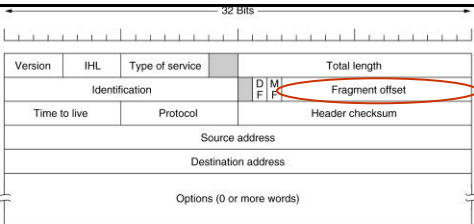
- Used for fragmentation
- DF means “do not fragment”
 - ◆ It is a request to routers not to fragment the datagram since the destination is incapable of putting the pieces back together
 - ◆ Can be use for MTU detection
- MF means “more fragments to follow”
 - ◆ All fragments except the last one have this bit set!
 - ◆ It is needed to know if all fragments of a datagram have arrived
- The bit to the left of DF is still unused ... (electrical waste ...)
 - ◆ Required to be 0

Fragment Offset

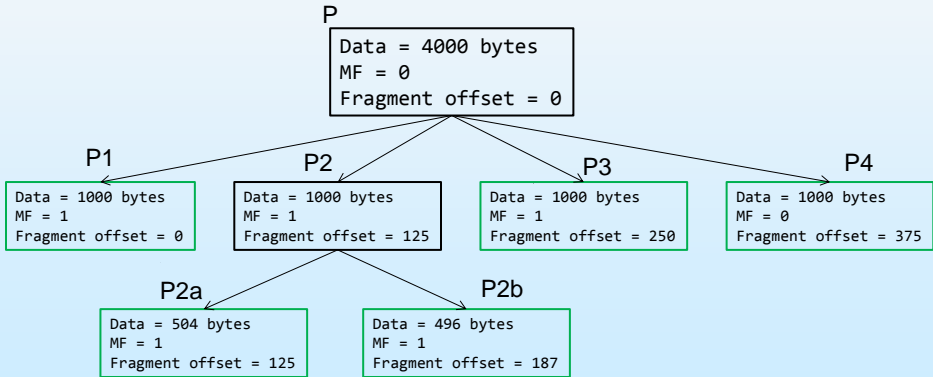


- Initial state: **fragment offset=0, MF=0**
- A router may divide a packet to small fragments, if next hop MTU is small
- Each fragmented packet will have to change these fields:
 - ◆ The **total length** field = fragment size
 - ◆ The **more fragments** (MF) flag is set for all fragments except the last one
 - ◆ The **fragment offset** field is set to the offset of the fragment in the original data payload (measured in units of eight-byte blocks)
- The **header checksum** field is re-calculated
- **fragment offset** = number of **eight-byte blocks** relative to the start of the original data payload
- Maximum **fragment offset** = $(2^{13} - 1) \times 8 = 65,528$ bytes

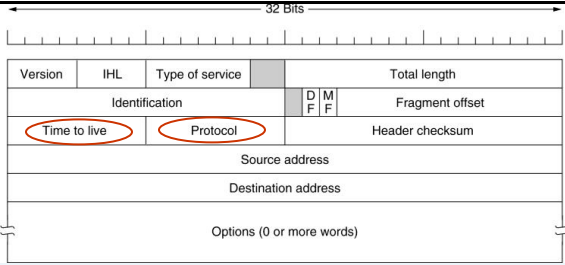
Fragment Offset



Packet P has reached a router at Albania and got fragmented to 4 Fragments: P1, P2, P3, P4
Packet P2 has reached a router at Micronesia and got fragmented
To: P2a, P2b



Time to Live & Protocol

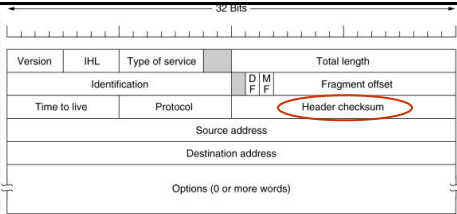


- Upper limit of routers to pass
- Usually set to 32 or 64
- Decrement by each router that processes the packet
- Router discards the datagram when TTL = 0

Protocol

- Tells IP where to send the datagram up to
 - ◆ 6 means TCP
 - ◆ 17 means UDP

Header checksum



- Only covers the header, not the data!
- How the checksum is computed?
 - ◆ Put a 0 in the checksum field
 - ◆ Add each 16-bit value together
 - ◆ Add in any carry
 - ◆ Inverse the bits and put that in the checksum field
- To check the checksum:
 - ◆ Add each 16-bit value together (including the checksum)
 - ◆ Add in carry
 - ◆ Inverse the bits
 - ◆ The result must be 0
- The ttl field changes at each hop so this needs to be recomputed on each hop
- Probability for error?

Example of IP Header

- What is IP Version? IHL? Type of Service

Start Python console and run:

```
>>> bin(0x45) = 0100,0101
>>> bin(0x6c) = 0110,1100
>>> bin(0x92) = 1001,0010
>>> bin(0xcc) = 1100,1100
```

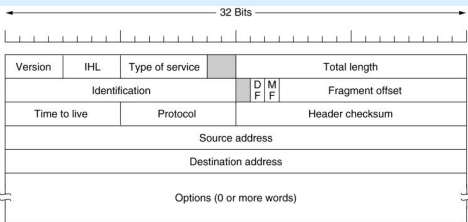
- Convert binary to decimal:

```
45:      Version = v4
        IHL = 5
00:      Type of Service = 0000
00 6c:   Total Length = 108
92 cc:   Identification = 1001001011001100
92 cc:   Checksum = 0x00
...
```

Note: When we build the IP header
We start with checksum=0x00 (RED)
And then calculate the checksum and
Write it back in that place

IP HEADER

45	00	00	6c
92	cc	00	00
38	06	00	00
92	95	ba	14
a9	7c	15	95



Checksum Calculation

first add all 16-bit values together,
adding in the carry each time:

```
4500 ←
+ 006c
----
456c
+ 92cc
----
d838
+ 0000
----
d838
+ 3806
----
1103e ← We have a carry here !
103e   Remove the leading 1 and add back
+ 1
----
103f
```

IP HEADER

45	00	00	6c
92	cc	00	00
38	06	00	00
92	95	ba	14
a9	7c	15	95

Checksum Calculation

103f
+ 0000

103f
+ 9295

a2d4
+ ba14

15ce8
5ce9
+ a97c

10665
0666
+ 1595

1bfb
1bfb = 0001 1011 1111 1011
e404 = 1110 0100 0000 0100
e404

← Again we have a carry here !
← Remove the leading 1 and add back

← Again we have a carry here !
← Remove the leading 1 and add back

← Now we have to inverse the bits:
1bfb = 0001 1011 1111 1011
e404 = 1110 0100 0000 0100
← This is the Checksum !

IP HEADER

45	00	00	6c
92	cc	00	00
38	06	00	00
92	95	ba	14
a9	7c	15	95

IP HEADER

45	00	00	6c
92	cc	00	00
38	06	e4	04
92	95	ba	14
a9	7c	15	95

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

83

Checksum Validation

- The receiver must validate the checksum
- It uses exactly the same algorithm, but this time it starts with "e404" and must end with "0000"
- If the computation does not end with "0000", the receiver does not accept the packet

IP HEADER

45	00	00	6c
92	cc	00	00
38	06	e4	04
92	95	ba	14
a9	7c	15	95

IP HEADER

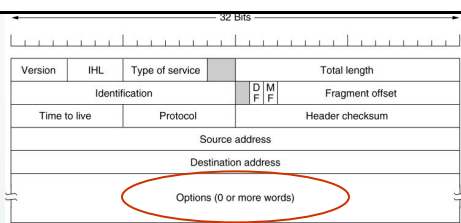
45	00	00	6c
92	cc	00	00
38	06	00	00
92	95	ba	14
a9	7c	15	95

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

84

42

Options



- Each option consists of 4 bytes
- The first byte is the option control block

0	1	2	3	4	5	6	7
copy flag	option class		option number				

- Copy flag: if 1, then copy option to fragments
- Option classes are
 - ◆ 0 - control
 - ◆ 1 - reserved
 - ◆ 2 - debugging and measurement
 - ◆ 3 - reserved
- The second byte designates the size of the entire option in bytes (including the control fields) and the other bytes are the option data.
- A padding to fill out the 32 bit words may be needed after all options
- There is room for at most 40 bytes for options (IP header max words = 15 words)

Our First Project !

Design a Python class IpDatagram with the following Interface:

```
hexstr ="4500006c92cc00003806e4049295ba14a97c1595217a6f2c"

# Class constructor
p = IpDatagram(hexstr)

# Class members
p.version = 4
p.ihl = 5
p.length = 40 (bytes)

# Class methods
p.source() = 192.68.25.7
p.destination() = 157.29.41.2
p.protocol() = 17
p.ttl() = 32
p.header() = The hex string of header part
p.data() = the hex string of the data part
p.checksum() = 0xe404
p.option(n) = Hex string of option n
>>>> MORE TO COME SOON ... (at the course web site)
```

TCP = Transport Control Protocol

- A reliable end-to-end byte stream over an unreliable internetwork
- Independent of network architecture, topology, speed
- Robust in the face of many kinds of failures
- Defined in RFC's 793, 1122, 1323
- A machine that supports TCP must have a single "TCP entity" as part of the operating system on top of the IP layer
- TCP sometimes mean a protocol, and sometimes it means a running computer process (operating system service)
- A bidirectional Protocol!
 - ◆ The peers (sender and receiver) exchange data in the same TCP segment format in both directions

TCP Connections

- Two machines establish a TCP connection by creating (or using) connection end-points that are called sockets
- A **socket** is fully identified by **network IP** and a **Port number**
 - ◆ But it has more structure and operations
- Port numbers are assigned by the OS as 16-bit number
- Each machine can have up to 65535 ($2^{16}-1$) open ports
- So it is possible to have many connections between two machines (how many in principle?)
- One port can be involved in many connections
 - ◆ with different ports on the other host
 - ◆ with the same port on different hosts
 - ◆ Several browsers on the same host connected to ynet http server

TCP Segments (1)

- TCP receives data from the Application layer (explorer, gmail, etc.)
- It may send it immediately or buffer it until it collects a large amount to send at once
- If urgent, it is possible to force TCP to flush its buffers
 - ◆ Socket flush method (sender side)
 - ◆ special bit in the TCP packet (receiver side)
- TCP breaks the data into segments (TCP packets)
- Each segment is shipped separately from the others
- may even take a different route than others
- may arrive to their destination out of order
- some of them may be lost
- It's up to the TCP entity at the other end to reassemble, report missing segments, etc, and deliver the data to the receiving process.

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

89

TCP Segments (2)

- TCP breaks the data into segments ("TCP packets")
- Each segment is shipped separately from the others
- Each segment may take a different route than the others
- Segments may arrive to their destination out of order
- Some segments may get lost and not reach their destination
- It is up to the TCP entity at the other end to
 - ◆ Acknowledge received segments
 - ◆ Ignore corrupt segments (no ack is required)
 - ◆ Reassemble segments to full message
 - ◆ Deliver the data to the receiving process.

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

90

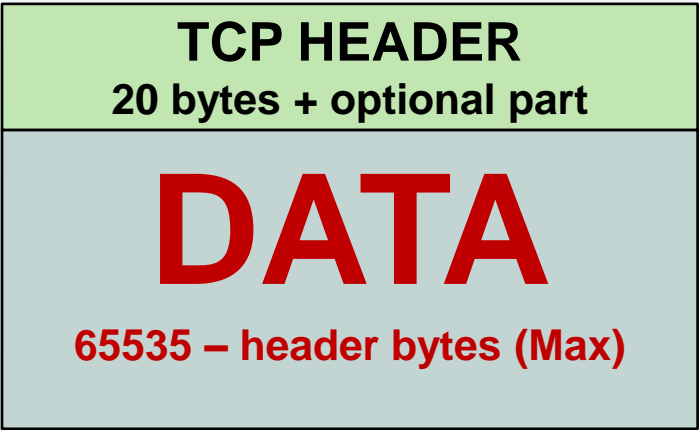
TCP Connection Control

- After TCP sends a segment it maintains a timer for receipt of an acknowledgment from the other end
 - ◆ Every received segment is acknowledged
 - ◆ Timeout/retransmission is adaptive
 - ◆ Checksum on TCP pseudo-header
 - ◆ A bad segment is discarded without a NAK
- Duplicate segments are discarded by the receiving TCP
 - ◆ IP may deliver duplicate datagrams
- Sender times out and retransmits (if no ack. received)
- Flow control (sliding windows algorithm) Ensures that a fast sender does not swamp a slow receiver

TCP Congestion Control

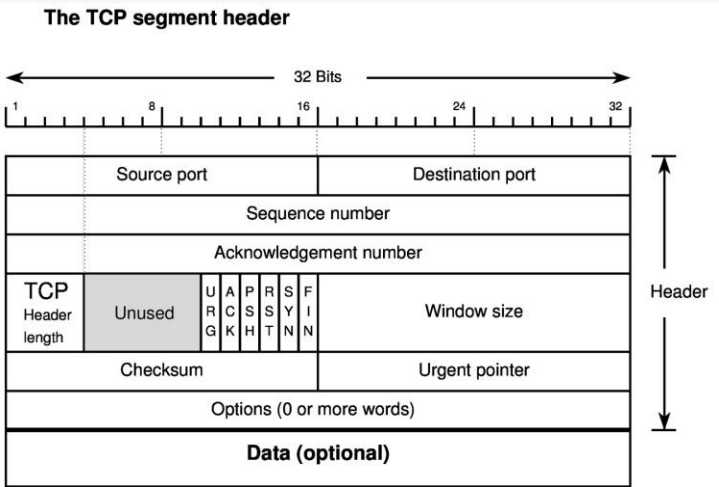
- Congestion control (host-network interaction) Prevents too much data from being injected into the network
- TCP avoids sending small packets by accumulating octets until a buffer is full or until a timer expires (default 2 ms).
- Each data byte has a sequence number!
 - ◆ Used to reassemble segments in order
- Each sequence number must be acknowledged
 - ◆ This is done by acknowledging the id of the first byte of the next TCP packet (it is indicated at its header ack. 16 bits number)
- Initial sequence numbers should be assigned randomly to minimize problems with duplicate numbers from different connections

TCP Segment Structure



In real life TCP packets are much smaller 500 bytes to 4K, and often Just header with no data at all!

TCP Segment Structure



The TCP segment header

■ The TCP header consists of:

- ♦ Minimum 20-byte (5 words) of fixed-format info
- ♦ Optional part (always an integer multiple of 4-bytes)

■ The TCP Data has at most $65535 - 20 - 20$ minus the options length (bytes)

- ♦ The second -20 comes from IP header

■ Thus any TCP segment can have at most $65535 - 20$ ($2^{16} - 21$) bytes in total

■ However this number is usually severely limited by the network MTU (maximum transfer unit) which is usually 1500 bytes

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides) 95

TCP PORTS

■ A port is a logical address for intercrosses communication node

■ Ports provide multiple destinations within one host computer, and even within the same process!

■ port numbers below 256 are "well-known" ports like:

- ♦ 21 for FTP
- ♦ 23 for TELNET
- ♦ 25 for SMTP
- ♦ 80 for HTTP
- ♦ 110 for POP3

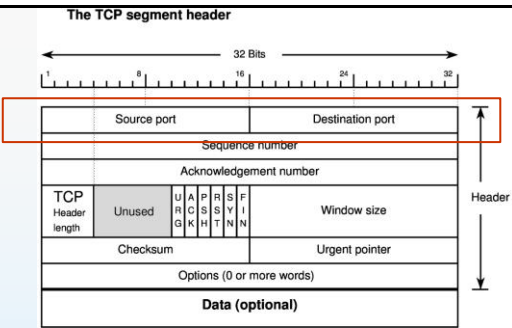
■ port numbers below 1024 are reserved for system services

- ♦ Only the administrator (like root in Unix) is allowed to allocate them

■ Port numbers from 1024 to 65535 ($2^{16} - 1$) can be used by user processes without any special permission

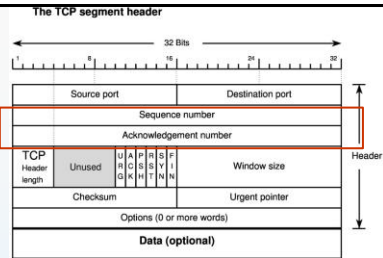
Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides) 96

TCP SOCKETS



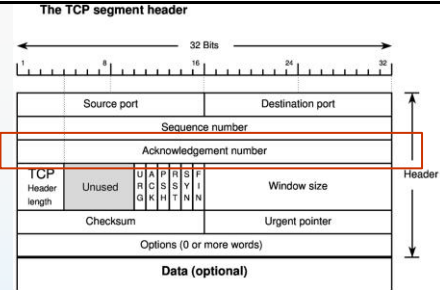
- A socket is a software object which represents a point of inter-process communication (node)
- Sometimes called: Berkeley sockets
- Sometimes called: TSAP - Transport Service Access Point
- A socket is sometimes characterized by its IP number and port number, but it has more than that (as a software unit with methods and data fields)
- Sockets provide multiple connection points within one host computer, and even within the same process!
- More on sockets in the next lecture unit

Sequence and Acknowledgement numbers



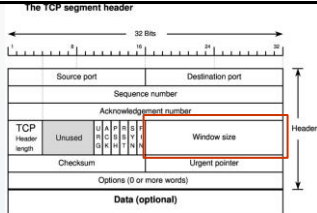
- A 32-bit number
- Every byte of the data is numbered
- The sequence number for a TCP segment is the id number of the first data byte in the segment
- It does not need to start with 1!
 - ◆ for good reasons – it better be random (after each reset)
- The range of valid sequence numbers is:
 - ◆ 0 to 4,294,967,295
 - ◆ Or: 0x0000,0000 to 0xFFFF,FFFF

Acknowledgement Number



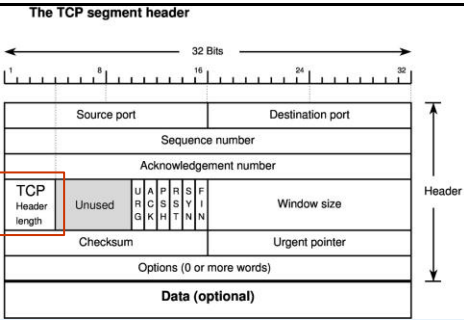
- A 32-bit number. **Valid only if the ACK bit is turned on.**
- Specifies the number of the next byte expected from the sender
 - ◆ **This the last byte correctly received + 1**
- Sent with data from the receiver to the sender
- By this, the receiver confirms to the sender that it has received all bytes below this number (ack. number)
- If this ack. segment does not arrive in certain time, the sender re-transmits the previous segment (timer timeout)

16-bit window



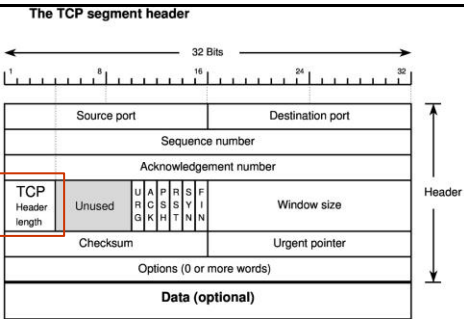
- The number of data bytes in the segment beginning with the one indicated in the acknowledgment field, which the sender of this segment is willing to receive next
- The "Acknowledgement number" field is the remaining receiver buffer size (bytes)
- Ack=0 signals that that the bytes up to acknowledgment number-1 have been received, but the receiver is incapable to accept more data at this moment
- Later, if the receiver is ready to receive more data, it sends a segment with the same acknowledgment number and a non-zero window size
- If this segment is lost, the sender re-transmits after timeout

Header Length



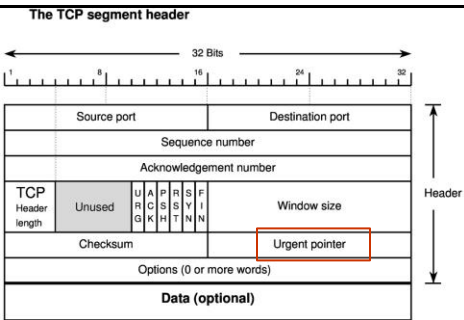
- This is the number of 32-bit in the TCP header
- This info is required since the header sometimes can be longer than 4 words
- Only 4 bits are allocated to the TCP header length field
- So it can be at most 15 words long (60 bytes)

Checksum



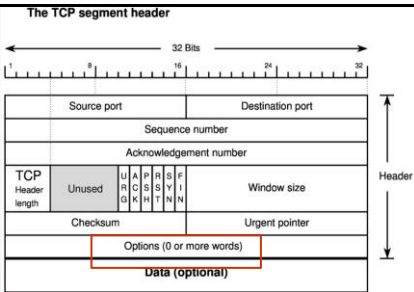
- Unlike the case of the IP datagram, checksum for TCP segment covers the whole segment including data and header
- Before computing the checksum, the algorithm zeros the checksum field and also includes a dummy IP header

Urgent Pointer



- Points to an urgent data a byte offset from the current sequence number
- Used to signal the receiver to abort broken FTP or TELNET sessions
- seldom used

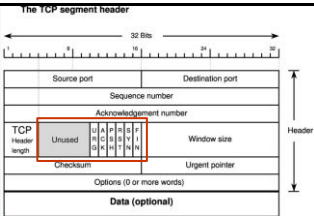
Options



- Simpler than IP options
- TCP option format:
 - ◆ A single byte for the option type
 - ◆ A length byte
 - ◆ data bytes
- If the type requires it.
- Currently implemented options are:
- End of option list indicates the end of the options, in case the end of the option bytes does not coincide with the end of the TCP headers
- Maximum segment size specifies the maximum segment size the sending TCP would like to receive

kind	length	meaning
0	-	end of option list
1	-	no operation
2	4	maximum segment size

BITS



- **URG=1** means the urgent pointer is a valid byte offset from the current sequence number at which urgent data are to be found (interrupt message)
 - ◆ Urgent mode is used when aborting rlogin or telnet connections, or ftp data transfers
- **ACK=1** means the acknowledgment number is valid
- **ACK=0** means there is no acknowledgment in this segment (usually no data)
- **PSH=1** then receiver should pass this data to the application ASAP
 - ◆ The receiver is requested to deliver the data to the application upon arrival and not buffer it
- **RST** - Reset (close) the connection
 - ◆ after a crash or errors (such as ack to a packet you never sent)
- **SYN** - Synchronize sequence numbers to begin a connection (see next slide)
- **FIN** - The sender has finished sending data (close)
- **Unused 6 bits** – too bad! (lots of electricity waste ...)
 - ◆ at some point used to debug the protocol
 - ◆ Lately used to pass performance info between hosts

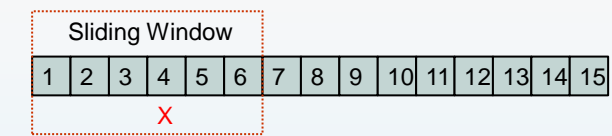
SYN: handshake (by example)

- **Step 1:** Sender sends a TCP segment with **SYN = 1**, **ACK = 0**, and **ISN=7000** (Initial Sequence Number example)
 - ◆ **SYN** is short for **Synchronize**
 - ◆ The **ISN=7000** is the beginning of the sequence numbers for data that the sender will transmit
 - ◆ **SYN** flag announces an attempt to open a connection
- If connection established then the first byte transmitted to the receiver will have the sequence number **ISN+1**
- **Step 2.** After receiving this TCP segment, the receiver returns a TCP segment with **SYN = 1**, **ACK = 1**, **ISN = 5000** (the receiver starting sequence number), and **Acknowledgment Number = 7001**
- **Step 3.** the sender sends a TCP segment to the receiver that acknowledges the receiver's **ISN**, With flags set as **SYN = 0**, **ACK = 1**, **Sequence number = 7001**, **Acknowledgment number = 5001**
- This handshaking technique is referred to as the Three-way handshake or SYN, SYN-ACK, ACK.

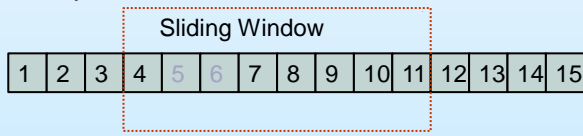
TCP Sliding Window Algorithm

- [YouTube Visualization movie](#)
- The idea: allow sender to send multiple packets without waiting for acknowledgement
- But how many packets?
- Step 1: Send 1 packet and wait for ack.
- After getting ack. 1 from the receiver, inspect the advertised "window size": this is the size of buffer that the receiver has for buffering packets
- Sender calculates how many packets can fit in window size and send all of them without waiting for ack. After that the sender waits for acks.
- This process repeats after getting each ack.
- Sender usually buffers window packets, since it may need to re-transmit some of them
- Receiver also need to buffer them in order to acknowledge early packets
- If the receiver's buffer is squeezed or finished, it may advertise a very low window size which will force the sender to slow down or stop

TCP Sliding Window Algorithm: Example



- Sender shoots 6 packets in a row with no ack. And then waits for ack. (window size is large enough to allow 6 packets)
- Receiver gets all packets except for packet 4
- Receiver sends ack. to packets 1,2,3 but cannot ack. packets 5, 6 (packet 4 was lost)
- After timeout, sender re-transmits packet 4, waits for ack. to packets 4, 5, and 6
- Receiver gets packet 4 and sends ack. for packets 4, 5, and 6
- Receiver may decrease window size of TCP header, and thus "slide" the window down



- Advanced protocols dynamically tune the window size to be suitable for both sides
- This sliding window is usually noticed when transmitting big files from one Windows machine to another, initially the time remaining calculation will show a large value and will come down later

Ethernet Frame

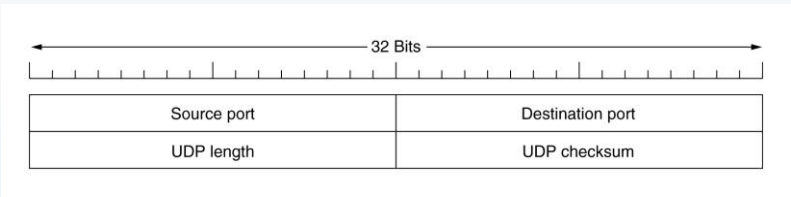
Header	Destination MAC Address Source MAC Address Protocol ID
body	DATA
Trailer	CRC Checksum

- MAC Address = Ethernet card 12 bytes id
- MAC = Media Access Control
- Example: b0:a0:92:48:72:45
- placing the CRC at the end of a frame reduces packet latency and reduces hardware buffering requirements

A DECODED ETHERNET FRAME

Ethernet Header	00 A0 92 48 72 45 00 00 0C 05 C3 58 08 00	dest. MAC address = 0:a0:92:48:72:45 source MAC address = 0:0:c:5:c3:58 network protocol = 0x0800 (IP)
IP header	4 5 00 00 29 DB FB 40 00 FE 06 7D CB 81 6E 1E 1A 81 6E 02 11	IP version = 4 header length = 5 words (word=4 bytes) type of service = 0 (normal) length = 0x29 octets = 41 bytes datagram identification don't fragment TTL = 254 transport protocol type = 6 (TCP) header checksum source IP address = 129.110.30.26 destination IP address = 129.110.2.17
TCP header	02 8B 02 03 6A 86 7B 57 B6 B6 B0 20 50 10 24 00 15 89 00 00	source port = 0x028b (651 dec.) desti. port = 0x0203 (515 dec., printer) source seqno = 1787198295 (dec.) acknowledgment no = 3065425952 (dec.) header length = 5 words indicates an ACK window size = 0x2400 (9216 dec.) TCP checksum urgent pointer off
DATA	02 54 41 4D 49 4C	Data byte Padding to make a 46 byte IP datagram
Ethernet Trailer	D7 87 6C A4	Ethernet checksum (Ethernet trailer)

UDP Header



- Protocol number = 17
- Always 8 bytes length header
- UDP Length = Header + Data length in bytes
- Maximum length = 65515 (due to IP size limit)
- Checksum cover the full packet (header+data)
- Checksum usage is optional (usually=0)
- No flow control!
- No congestion control!
- Unreliable! (up to user processes)
- Packet order, timing, and error control are usually done at the data level
- DNS I using UDP for name resolution

ICMP Protocol

- ICMP - Internet Control Message Protocol
- used by the operating systems to send error messages indicating, for example, that a requested service is not available or that a host or router could not be reached
- Another example: if a router receives a packet larger than the next hop MTU, it may drop the packet and send an ICMP message which indicates the condition "Packet too Big", or it may fragment the packet and send it over the link with a smaller MTU
- ICMP can also be used to relay query messages
- It is assigned protocol number 1
- We skip the header and other details in this course (read Tanenbaum for more details)

Part 4

TRANSPORT LAYER

SOCKET PROGRAMMING

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

113

Transport Layer

- Data transmission service goals for the application layer
 - ◆ Efficiency
 - ◆ Reliability
 - ◆ Accuracy
 - ◆ Cost-effective
- The entity that does the work is called the **transport entity**
- The **transport entity**
 - ◆ Is usually part of the operating system kernel
 - ◆ sometimes a separate library package which is loaded by the OS or even user processes
 - ◆ And sometimes even on the network interface card
- The transport entity (TCP) employs the services of the network layer (IP), and its associated software and hardware (cards and device drivers)

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

114

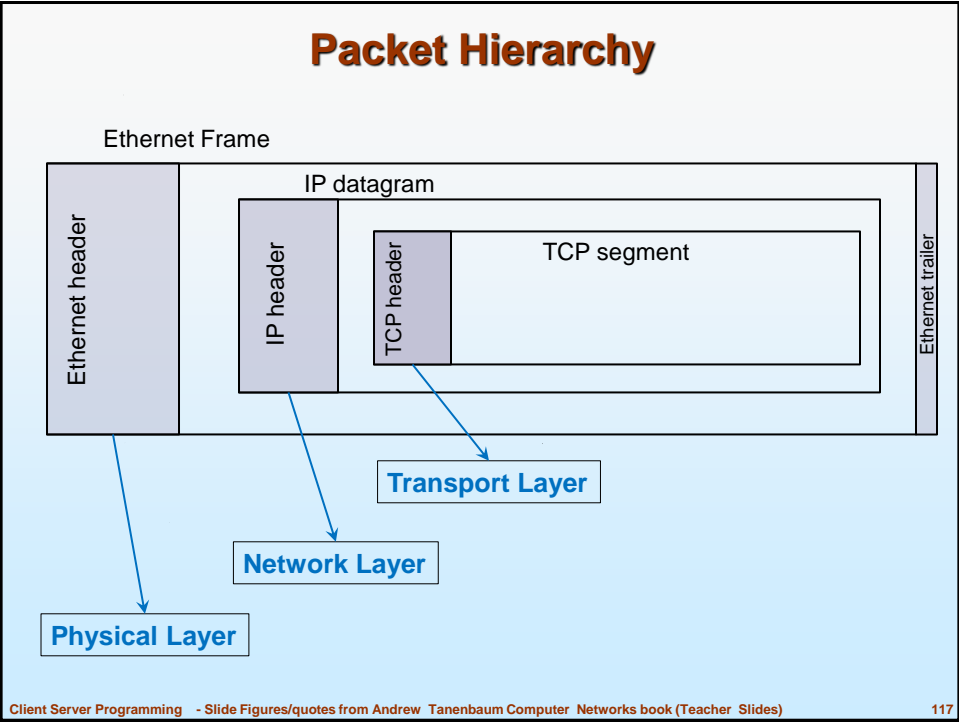
Transport Layer

- The transport entity code runs entirely on users machines, but the network layer mostly runs on routers, cards, and other bridging hardware
- Bridging hardware is inherently unreliable and uncontrollable
 - ◆ Ethernet cards, routers, and similar hardware do not contain adequate software for detecting and correcting errors
- To solve this problem we must add another layer that improves the quality of the service:
 - ◆ the transport entity detects network problems: packet losses, packet errors, delays, etc.
 - ◆ and then fixes these problems by: retransmissions, error corrections, synchronization, and connection resets
- Transport layer interface must be simple and convenient to use since it is intended for a human user

Transport Service Primitives

	Primitive	Packet sent	Meaning
Server	LISTEN	(none)	Block until some process tries to connect
Client	CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
Server/Client	SEND	DATA	Send information
Server/Client	RECEIVE	(none)	Block until a DATA packet arrives
Server/Client	DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

- These are the basic logical actions between two communication points
- A communication point is created by a process that runs on a machine
- There are several software implementations of these abstract model
- The most common is called: "Berkeley Sockets"
- Note that the "LISTEN" and "RECEIVE" actions do not involve any packet transmission! These are actually operating system states:
 - ◆ LISTEN – go to sleep until a connection arrives (OS is attending)
 - ◆ RECEIVE – go to sleep until data arrives (OS does the buffering)



Berkeley Sockets

- Sockets first released as part of the Berkeley UNIX 4.2BSD software distribution in 1983
- They quickly became popular
- The socket primitives are now widely used for Internet programming on many operating systems
- There is a socket-style API for Windows called “winsock”

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides) 118

Berkeley Socket Services

	Primitive	Meaning
Client/Server	SOCKET	Create a new communication end point
Server	BIND	Attach a local address to a socket
Server	LISTEN	Announce willingness to accept connections; give queue size
Server	ACCEPT	Block the caller until a connection attempt arrives
Client	CONNECT	Actively attempt to establish a connection
Client/Server	SEND	Send some data over the connection
Client/Server	RECEIVE	Receive some data from the connection
Client/Server	CLOSE	Release the connection

- The **SOCKET** primitive creates a new endpoint and allocates table space for it within the transport entity
- The first four primitives are executed in that order by servers
- A successful **SOCKET** call returns an ordinary **file descriptor** for use in succeeding calls, the same way an OPEN call on a file does

SERVER SOCKET

- Newly created socket has no network address (yet)
 - ◆ The machine may have several addresses (thru several interface cards)
 - ◆ It must be assigned using the **BIND** primitive method
- Once a socket has **bound** an address, remote clients can connect to it
- The parameters of the **SOCKET** call specify the addressing format to be used, the type of service desired (reliable byte stream , DGRA, etc), and the protocol.

```
import socket
# Creating a server socket on the local machine
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock.bind( '', 2525 )
sock.listen( 5 )
new_sock, (client_host, client_port) = sock.accept()
print "Client:", client_host, client_port
```

CLIENT SOCKET

- A client socket is created exactly as a server socket except that it does **not locally bound** to the machine, and it **does not listen**
- A client socket is **connecting** to an already running server socket, usually on a remote host, but also on the local host (as yet one more method of inter-process communication!)

```
import socket
# Creating a client socket
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
host = socket.gethostname()
# connect to local host at port 2525
server = (host, 2525)
sock.connect(server)
```

CONNECT & ACCEPT primitives

- When a **CONNECT** request arrives from a client to the server, the transport entity creates a **new copy of the server socket** and returns it to the **ACCEPT** method (as a file descriptor)
- The server can then fork off a process or thread to handle the connection on the new socket and go back to waiting for the next connection on the original socket
- ACCEPT returns a file descriptor, which can be used for reading and writing in the standard way, the same as for files.

```
import socket
# Creating a server socket on the local machine
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock.bind( '', 2525 ) # bind to all local interfaces
sock.listen( 5 ) # allow max 5 simultaneous connections
newsock, (client_host, client_port) = sock.accept()
print "Client:", client_host, client_port
```

SEND & RECEIVE primitives

- The **CONNECT** primitive blocks the caller and actively starts the connection process (the transport entity is in charge)
- When it completes (when the appropriate **TCP** segment is received from the server), the client process is awakened by the **OS** and the connection is established
- Both sides can now use **SEND** and **RECEIVE** to transmit and receive data over the full-duplex connection

```
# server to client:
newsock.send("Hello from Server 2525")

# client to server
server = (host, 2525)
sock.connect(server)      # connect to server
sock.recv(100)            # receive max 100 chars
```

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

123

CLOSE primitive

- When both sides have executed the **CLOSE** method, the connection is released
- Berkeley sockets have proved tremendously popular and have become the standard for abstracting transport services to applications
- The socket API is often used with the TCP protocol to provide a connection-oriented service called a **reliable byte stream**
- But sockets can also be used with a connectionless service (UDP)
- In such case, **CONNECT** sets the address of the remote transport peer and **SEND** and **RECEIVE** send and receive UDP datagrams to and from the remote peer

```
# Server:
newsock.close()

# Client
sock.close()
```

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

124

The Simplest Client/Server App

```
import socket
# Creating a server socket on the local machine
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock.bind( '', 2525 )
sock.listen( 5 )
newsock, (client_host, client_port) = sock.accept()
print "Client:", client_host, client_port
newsock.send("Hi from server 2525")
newsock.close()
```

SERVER

```
import socket
# creating a client socket
sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
host = socket.gethostname()
# connect to local host at port 2525
server = (host, 2525)
sock.connect(server)
print sock.recv(100)
sock.close()
```

CLIENT

Q: How many clients can connect to this server?

Socket Programming Example in C: Internet File Server

Client code using sockets:
Client program that requests a
File from a server program

```
/* This page contains a client program that can request a file from the server program
 * on the next page. The server responds by sending the whole file.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345 /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096 /* block transfer size */

int main(int argc, char **argv)
{
    int c, s, bytes;
    char buf[BUF_SIZE]; /* buffer for incoming file */
    struct hostent *h; /* info about server */
    struct sockaddr_in channel; /* holds IP address */

    if (argc != 3) fatal("Usage: client server-name file-name");
    h = gethostbyname(argv[1]); /* look up host's IP address */
    if (!h) fatal("gethostbyname failed");

    s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (s < 0) fatal("socket");
    memset(&channel, 0, sizeof(channel));
    channel.sin_family = AF_INET;
    memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
    channel.sin_port = htons(SERVER_PORT);

    c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
    if (c < 0) fatal("connect failed");

    /* Connection is now established. Send file name including 0 byte at end. */
    write(s, argv[2], strlen(argv[2])+1);

    /* Go get the file and write it to standard output. */
    while (1) {
        bytes = read(s, buf, BUF_SIZE); /* read from socket */
        if (bytes <= 0) exit(0); /* check for end of file */
        write(1, buf, bytes); /* write to standard output */
    }

    fatal(char *string)
    {
        printf("%s\n", string);
        exit(1);
    }
}
```

Socket Programming Example in C: Internet File Server (2)

Server code

```
#include <sys/types.h> /* This is the server code */
#include <sys/errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345 /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096 /* block transfer size */
#define QUEUE_SIZE 10

int main(int argc, char *argv[])
{
    int s, b, l, fd, sa, bytes, on = 1;
    char buf[BUF_SIZE];
    struct sockaddr_in channel; /* hold's IP address */

    /* Build address structure to bind to socket. */
    memset(&channel, 0, sizeof(channel)); /* zero channel */
    channel.sin_family = AF_INET;
    channel.sin_addr.s_addr = htonl(INADDR_ANY);
    channel.sin_port = htons(SERVER_PORT);

    /* Passive open. Wait for connection. */
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); /* create socket */
    if (s < 0) fatal("socket failed");
    setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));
    b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
    if (b < 0) fatal("bind failed");

    l = listen(s, QUEUE_SIZE); /* specify queue size */
    if (l < 0) fatal("listen failed");

    /* Socket is now set up and bound. Wait for connection and process it. */
    while (1) {
        sa = accept(s, 0, 0); /* block for connection request */
        if (sa < 0) fatal("accept failed");

        read(sa, buf, BUF_SIZE); /* read file name from socket */

        /* Get and return the file. */
        fd = open(buf, O_RDONLY); /* open the file to be sent back */
        if (fd < 0) fatal("open failed");

        while (1) {
            bytes = read(fd, buf, BUF_SIZE); /* read from file */
            if (bytes <= 0) break; /* check for end of file */
            write(sa, buf, bytes); /* write bytes to socket */
        }

        close(fd); /* close file */
        close(sa); /* close connection */
    }
}
```

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

127

C Socket API (1)

```
// Usually located at /usr/include/sys/socket.h

/* Create a new socket of type TYPE in domain DOMAIN, using
   protocol PROTOCOL. If PROTOCOL is zero, one is chosen automatically.
   Returns a file descriptor for the new socket, or -1 for errors. */

extern int socket (int __domain, int __type, int __protocol) __THROW;

/* Give the socket FD the local address ADDR (which is LEN bytes long). */

extern int bind (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len)
    __THROW;

/* Put the local address of FD into *ADDR and its length in *LEN. */
extern int getsockname (int __fd, __SOCKADDR_ARG __addr,
    socklen_t *__restrict __len) __THROW;

/* Open a connection on socket FD to peer at ADDR (which LEN bytes long).
   For connectionless socket types, just set the default address to send to
   and the only address from which to accept transmissions.
   Return 0 on success, -1 for errors.
   This function is a cancellation point and therefore not marked with
   __THROW. */

extern int connect (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len);
```

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

128

C Socket API (2)

```

/* Open a connection on socket FD to peer at ADDR (which LEN bytes long).
   For connectionless socket types, just set the default address to send to
   and the only address from which to accept transmissions.
   Return 0 on success, -1 for errors.

   This function is a cancellation point and therefore not marked with
   __THROW.  */

extern int connect (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len);

/* Send N bytes of BUF to socket FD.  Returns the number sent or -1.

   This function is a cancellation point and therefore not marked with
   __THROW.  */

extern ssize_t send (int __fd, const void *__buf, size_t __n, int __flags);

/* Read N bytes into BUF from socket FD.
   Returns the number read or -1 for errors.

   This function is a cancellation point and therefore not marked with
   __THROW.  */

extern ssize_t recv (int __fd, void *__buf, size_t __n, int __flags);

```

WWW Client Sockets (v1)

```

import socket, os

sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM)
google_server = ("www.google.com", 80)
sock.connect(google_server)
# HTTP protocol "GET" command
sock.send("GET / HTTP/1.0\r\n\r\n")

# Receiving the index.html file
bufsize = 4096
html_file = "c:/workspace/index.html"
f = open(html_file, "w")
while True:
    data = sock.recv(bufsize)
    if not data:
        f.close()
        break
    f.write(data)

os.system("notepad.exe " + html_file)
#os.startfile(html_file)

```

Python File Server (v1)

```
import socket, sys

servsock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
servsock.bind(("", 12345)) # bind to all local host interfaces
servsock.listen(25)       # set maximum accept rate to 25 connections

while True:
    newsock, address = servsock.accept()
    file = newsock.recv(255) # receive file name: max 255 chars
    print "File =", file
    f = open(file, "rb")     # open file for reading in binary mode
    while True:
        data = f.read(4096)
        if not data:
            f.close()
            break
        n = newsock.send(data)
        if n < len(data):
            raise Exception("send error: transmitted less than data length")
    newsock.close()
```

Unsafe!

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

131

Python File Client (v1)

```
# To be run from the command line
import socket, sys

remote_file_name = sys.argv[1]
local_file_path = sys.argv[2]

sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
sock.connect(("localhost", 12345))
sock.send(remote_file_name)
f = open(local_file_path, "wb")
while True:
    data = sock.recv(4096)
    if not data:
        f.close()
        break
    f.write(data)

sock.close()
```

Unsafe!

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

132

Conversation Techniques

- A reliable and robust communication between two sockets, can sometimes become a highly complex and fragile
- To simplify it and manage its complexity, some strict rules must be followed
- A **message** must be sent in one of the following modes:
 1. **Fixed length** (like always 40 bytes, with padding if necessary)
 2. **Delimited** (like: "name = Dan Hacker\n")
 3. **Predefined length:**

"240 message ... ends ... after ... 240 bytes"

The size itself can be of fixed length or delimited
 4. **End by shutting down the connection**
- In practice, all these 4 methods are used in combination!

Safe Socket Send

- In general this is not needed, but in some rare cases the socket send method is not guaranteed to send all the message!!
- It may send just a part of it, and therefore we must ensure sending the full message
- In most cases (short messages) this is not needed, but keep this in mind!
- The `sendall()` method has the same effect

```
def safe_send(sock, message):
    i = 0
    n = len(message)
    while i < n:
        sent = sock.send(message[i:])
        if sent == 0:
            raise RuntimeError("socket connection broken")
        i += sent
```

A Safe Socket `sendall()` method

- The socket class is already equipped with a safe `sendall()` method which does not return until it sent the whole message, or until an error is encountered
- `None` is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

```
r = sock.sendall(data)
if not r is None:
    print "Exceptional socket sendall return code:", r
    raise Exception("send error: data was not fully transmitted")
```

Receiving Fixed Size Message

- The socket `recv()` method may get less characters than requested
- To be fully safe, we need to run `recv()` several times to get the full message (provided we know the exact message size in advance!)
- The next function ensures that we get an exact number of bytes from the socket

```
def recv_fixed_size(sock, expected_size, bufsize=0):
    if bufsize == 0:
        bufsize = min(expected_size, 4096)
    message = ""
    while len(message) < expected_size:
        chunk = sock.recv(bufsize)
        if chunk == "":
            raise RuntimeError("socket connection broken")
        message += chunk
    return message
```

Receiving a Delimited Message

- Delimited message are messages that end with a delimiting character that is agreed by both sides
- The usual delimiting character is the newline character '\n', or some special character (such as '@')
- This is however slow due to the fact that we must receive 1 character at a time

```
def recv_delimited_message(sock, limit='\n'):
    message = ""
    while True:
        char = sock.recv(1)
        if char == "":
            return None
        if char == limit:
            break
        else:
            message += char
    return message
```

Receiving a Delimited Message

■ EXAMPLE

```
# client side:
sock.sendall("c:/workspace/oliver.txt" + '\n')

# server side:
file = recv_delimited_message(servsock)
# file = "c:/workspace/oliver.txt"
```

- Note that the message itself drops the delimiting char! (i.e., the delimiting char is not part of the message!)

Send and Receive with a Size Header

- A faster technique for sending and receiving messages with a known size is by appending a "fixed size header" to the message itself
- Simple "encode/decode" methods are enough to make this technique very easy and efficient to use (between a client and server that agree on it)
- Here is the key idea:
 - ◆ Compute the message size in hexadecimal form
 - ◆ Pack this size into an 8 chars hex string, possibly by adding leading zeros to it if it is too short
 - ◆ Place the header in front of the message and send it!
- Example: message = **"Hello Web Wide World"**
 - ◆ Decimal size = **20**
 - ◆ Hexadecimal = **0x14**
 - ◆ Header (8 bytes) = **"00000014"** (removed the leading 0x)
 - ◆ Send message = **"0000014Hello Web Wide World"**

Code for: send_size and recv_size

```
# convert message length to hex and chop the leading '0x'
def send_size(sock, message):
    size_string = hex(len(message))[2:]
    data = (8 - len(size_string)) * '0' + size_string + message
    sock.sendall(data)

# The receiver gets the first 8 bytes, adds a "0x"
# prefix, and converts the hex to decimal
def recv_size(sock, bufsize=0):
    hexstr = "0x" + recv_fixed_size(sock, 8)
    size = int(hexstr, 16)
    return recv_fixed_size(sock, size, bufsize)
```

- Example: `recv_size("0000014Hello Web Wide World")`
 Will first get the first 8 chars header: `"00000014"`
 Then convert it to decimal: `size=20`
 Then recv the next 20 chars which form the message itself:
"Hello Web Wide World"

File Retrieval Routine

- Retrieving a file through a socket is very common, so we better have a common function that does it effectively
- This is also a safe measure for draining the socket into a local file: we are sucking all data from the socket until it has nothing else to receive
- However this is good only if socket closes connection after sending file

```
# Dump socket output (sock.recv) to a local file
def recv_to_file(sock, filename, mode='w', bufsize=4096):
    f = open(filename, mode)
    while True:
        data = sock.recv(bufsize)
        if not data:
            f.close()
            break
        f.write(data)
```

File Retrieval Routine

- For server socket that sends many files, the standard method is:
 1. Send the file size to the client
 2. Send the file stream to the client
- The next function retrieves a fixed size stream to a file:

```
def recv_fixed_size_to_file(sock, size, file, mode="wb", bufsize=0):
    if bufsize == 0:
        bufsize = min(size, 4096)
    f = open(file, mode)
    curr_size = 0
    while curr_size < size:
        data = sock.recv(bufsize)
        if data == "":
            raise RuntimeError("socket connection broken")
        f.write(data)
        curr_size += len(data)
    f.close()
```

Send File Routine

- Sending a file through a socket is also a very common routine, which we have already encountered several times
- Here is a safe function for sending a local file from a local socket to a remote host

```
def send_file(sock, file, mode="rb", bufsize=4096):
    f = open(file, mode) # open file for reading in binary mode
    while True:
        data = f.read(bufsize)
        if not data:
            f.close()
            break
        rcode = sock.sendall(data)
        if not rcode is None:
            print "Exceptional socket sendall return code:", rcode
            raise Exception("send error: data was not fully transmitted")
```

socket_utils module

- All these new socket utilities are assembled in the in the **socket_utils** module. It can be downloaded from:
<http://tinyurl.com/samyz/cliserv/lab/socket.zip>
- You can download it and throw in your Python library, and then import it to your Python programs (see below)
- You are encouraged to improve and add new utilities to this module!
- So it is expected to change a lot until we reach Projects 4 and 5, in which we will make important use with this module! (stay tuned)

```
# if you throw it to: "c:/workspace", then:
import sys
sys.path.append("c:/workspace")
from socket_utils import *

# if you throw it to "c:\python27\lib, then it will work immediately:
from socket_utils import *

# Not that this module also imports: socket, time, hashlib, os, threading
```

WWW Client Sockets (v2)

- Here is version 2 of our www connection to Google web server
- This time we are using our `recv_to_file` utility function to drain the socket to an html file

```
from socket_utils import *

sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
google_server = ("www.google.com", 80)
sock.connect(google_server)
sock.send("GET / HTTP/1.0\r\n\r\n")
html_file = "c:/workspace/index.html"
recv_to_file(sock, html_file)
os.system("notepad.exe " + html_file)
#os.startfile(html_file)
```

WWW Client Sockets (v3)

- In version 3 we present a more interesting **GET** request:
- Google search query

```
from socket_utils import *

sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
server = ("www.google.co.il", 80)
sock.connect(server)
sock.send("GET /search?q=python+socket+programming HTTP/1.0\r\n\r\n")
html_file = "c:/workspace/index.html"
recv_to_file(sock, html_file)
os.system("notepad.exe " + html_file)
os.startfile(html_file)
```

WWW Client Sockets (v4)

- One more example with a deep path

```
from socket_utils import *

sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
html_file = "c:/workspace/index.html"

server = ("www.cs.uic.edu", 80)
sock.connect(server)
sock.send("GET /~jbell/CourseNotes/OperatingSystems/index.html HTTP/1.0\r\n\r\n")
recv_to_file(sock, html_file)
os.startfile(html_file)
```

Python File Server (v2)

```
import socket, sys

servsock = socket.socket()
servsock.bind(("localhost", 12345))
servsock.listen(20) # set maximum accept rate to 20 connections

id = 0
while True:
    newsock, address = servsock.accept()
    id += 1
    start = time.time()%1000
    file = newsock.recv(255) # receive file name: max 255 chars
    send_file(newsock, file)
    end = time.time()%1000
    print "Connection %d: File = %s, Time = %.2f-%.2f" % (id, file, start, end)
    newsock.close()
```

Notes on socket send/recv

- When a `recv()` returns 0 bytes, it means the other side has closed the connection (or is in the process of closing connection)
- You will not receive any more data on this connection! Ever!
- But you may be able to send data successfully
- Similarly: if a `send()` returns after handling 0 bytes, the connection has been closed or broken
- Example: **HTTP** uses a socket for only one transfer:
 - ◆ The client sends a request, then reads a reply. That's it.
 - ◆ The socket is discarded
 - ◆ This means: a client can detect the end of the reply by receiving 0 bytes
 - ◆ (which corresponds to the fourth type of message transfer)

Python File Client (v2)

```
# To be run from the command line
from socket_utils import *
import sys

remote_file_name = sys.argv[1]
local_file_path = sys.argv[2]

sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
file_server = ("localhost", 12345)
sock.connect(file_server)
sock.send(remote_file_name)
recv_to_file(sock, local_file_path, 'wb')
sock.close()
```

Still Unsafe!

Quality Checks

- Testing networking applications is a very critical and difficult domain
- Google invests a substantial amount of resources for testing and validating its networking infrastructure and applications
- Examples: making sure that gmail message
 - ◆ Arrive on time
 - ◆ Are not lost
 - ◆ Are not modified on their journey
 - ◆ Backup and restore
 - ◆ Performance under congested and stressful networking conditions
- To get an idea on this domain, we will write a Python program that tests our file transfer server and client

Quality Checks Plan

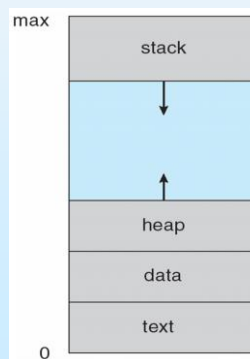
- Choose several files from different sizes for our Test Plan
 - ◆ We already have the **Oliver twist book** and our huge **db.csv** database
- Write a function that uses the file server to transfer a given file
- Write a function which loops over the previous function a large number of times (like: 20, 50, 100, and even 1000 times!)
- Our test program should check the following things:
 - ◆ The remote file and the transferred file are identical on each iteration
 - ◆ The transfer speed is reasonable and is uniform across all experiments
 - ◆ CPU consumption is not too high
 - ◆ memory usage is reasonable (no leaks or swamp)
- To be further discussed in class

Project 4: BFTP Braude File Transfer Protocol

- This is our next course project
- All 4 first versions of our small file server/client were have focused only on one operation: **GET file**
- A normal File transfer service usually have more than this operation. To list a few: **GET, PUT, LIST, PWD, CD, DELETE**, and more.
- These operations are discussed in the initial project draft. We will all make efforts to define the final project goals in the next week or two
- Please visit the course web site and read more on project 4 and try to help in defining the protocol and checking the common code
- To check the **socket_utils** code, try it on the previous small tests (1-4)

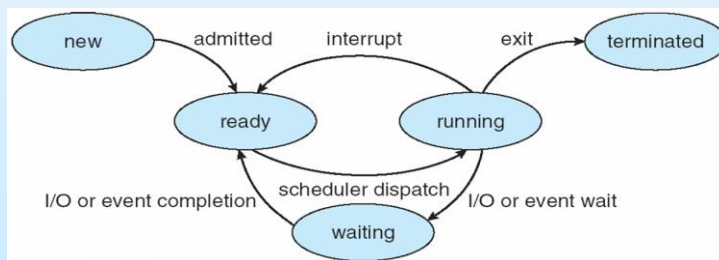
Process and Threads Concepts

- A process (or job) is a program in execution
- A process includes:
 1. Text (program code)
 2. Data (constants and fixed tables)
 3. Heap (dynamic memory)
 4. Stack (for function calls and temporary variables)
 5. Program counter (current instruction)
 6. CPU registers
 7. Open files table (including sockets)
- To better distinguish between a program and a process, note that a single Word processor program may have 10 different processes running simultaneously
- Consider multiple users executing the same Internet explorer (each has the 6 things above)
- Computer activity is the sum of all its processes



Process States

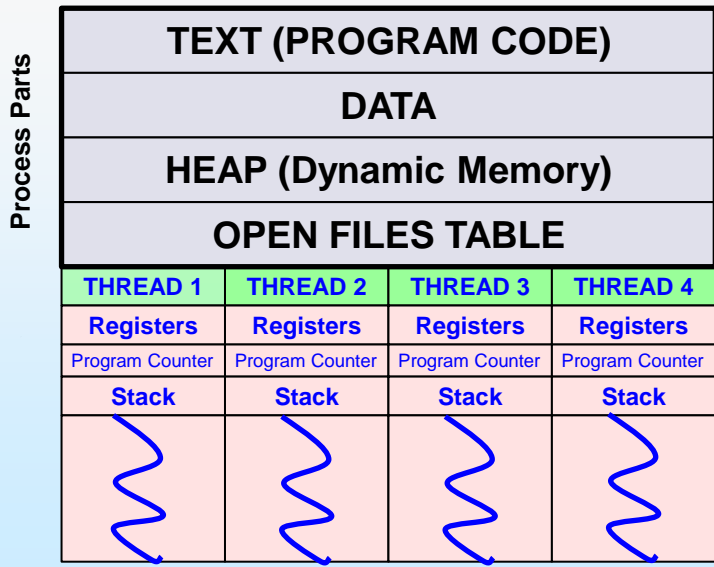
- As a process executes, it changes *state*
 - ◆ **new**: The process is being created
 - ◆ **running**: Instructions are being executed
 - ◆ **waiting**: The process is waiting for some event to occur
 - ◆ **ready**: The process is waiting to be assigned to a processor
 - ◆ **terminated**: The process has finished execution



CPU Process Scheduling

- Modern operating systems can run hundreds (or thousands) of processes in parallel !
- Of course, at each moment, only a single process can control the CPU, but the operating system is switching processes every 15 milliseconds (on average) so that at 1 minute, an operating system can switch between 4000 different process!
- The replacement of a running process with a new process is generated by an **INTERRUPT**

One Process, Many Threads!



THREADS

- A thread is a basic unit of CPU utilization consisting of
 - ◆ Program counter
 - ◆ Registers
 - ◆ Stack
 - ◆ Thread ID
- Every thread is running in the context of a parent process which have
 - ◆ TEXT (Program Code)
 - ◆ DATA (constants)
 - ◆ HEAP (Dynamic Memory)
 - ◆ Open Files Table
- A process consists of multiple threads which share these 4 things
- This means that several threads can use and share a common variable, a common open file, and even a common socket! In parallel

THREADS

- In modern operating systems, a process can be divided into several tasks that operate in parallel
- These tasks can sometimes run independently of each other, and sometimes with minimal interdependencies (or else it's better to give up threads!)
- This is particularly desirable if one of the tasks may block (and block the entire process), and then allow the other tasks to proceed without blocking
- Example: Microsoft Word process sometimes involves the following activities within a single running process:
 - ◆ A foreground thread processes user input (keystrokes)
 - ◆ Second thread makes spelling and grammar checks
 - ◆ Third thread loads images from the disk (or internet)
 - ◆ Fourth thread performs incremental backup in the background

THREADS - Notes

- Threads are easier to create than processes since they do not require a separate address space!
- Multithreading requires careful programming since threads share data structures that should only be modified by one thread at a time!
- Unlike threads, processes do not share the same address space and thus are truly independent of each other.
- Problem in one thread can cause the parent process to block or crash (and thus kill all other threads!)
- Threads are considered lightweight because they use far less resources than processes
- Threads, on the other hand, share the same address space, and therefore are interdependent
- Therefore a lot of caution must be taken so that different threads don't step on each other!

Python Threads: Hello 1

```
from threading import Thread
from time import strftime

class MyThread(Thread):
    def run(self):
        threadName = self.getName()
        timeNow = strftime("%X")
        print "%s says Hello World at time: %s" % (threadName, timeNow)

# Opening 5 threads
for i in range(5):
    t = MyThread()
    t.start()
```

Python Threads: Hello 2

```
import os, time, random
from threading import Thread

def hello(tname):
    delay = 0.050 + 0.100 * random.random() # random value between 0.050 to 0.150 (seconds)
    time.sleep(delay)
    print "Delay =", delay
    print "Hello from thread %s" % (tname)

def run_threads():
    print "Process ID =", os.getpid()
    t1 = Thread(target=hello, args=('t1',))
    t2 = Thread(target=hello, args=('t2',))
    t3 = Thread(target=hello, args=('t3',))
    t4 = Thread(target=hello, args=('t4',))
    t5 = Thread(target=hello, args=('t5',))

    threads = [t1, t2, t3, t4, t5]

    for t in threads:
        print "Starting thread:", t
        t.start()

    for t in threads:
        t.join()
```

Part 5

PROTOCOL DESIGN

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

163

AGENDA

- Networking Protocol Design Principles
- Common Networking Protocol Techniques
- Learn from old and highly used internet protocols
- Introducing SMTP, POP3, and IMAP by examples

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

164

Principles of Protocol Design

- Reference: <http://nerdland.net/2009/12/designing-painless-protocols>

Protocol Design: Principle 1

Do not re-invent the Wheel!

- Try first to use existing protocols, or at least to imitate them as much as possible
- Protocols which survived many years are probably good and well thought
- They passed a lot of storms and fire tests and they are still here!
- For this, we need to get to know at least the most popular ones first

Protocol Design: Principle 2

KISSD - Keep It Simple Stupid and Deterministic

- Complicated protocols are doomed to cause chaos, complications, and eventually die!
- At every stage it should be completely clear what can happen next!
- Situations in which anything can happen lead to "code pollution" and later to horrible bugs and eventually to "protocol death"

Protocol Design: Principle 3

Prefer Human Readability

- Prefer plain simple text on short cryptic codes
- Unless speed is truly the most important factor in your system!
- Always better to sacrifice speed for readability
 - ◆ "less is more" principle
- Commands like LOGIN, GOODBYE, HELLO, QUIT are much clearer than codes like: 031, 404, 502, etc.
- If your protocol is going to contain free-form text then your protocol really should use Unicode!
- English is most definitely not the only language on the Internet!

Protocol Design: Principle 4

Make Magic Numbers Meaningful

- In many cases, numeric status codes can be useful and even human readable
- Make sure to use meaningful numbers with clear structure
- For example every HTTP response comes with a numeric status code prefix
- Everyone is familiar with:
HTTP 404 code ("File Not Found" error code)
- In most cases, it's just enough to see the number and immediately understand what happened
- The meaning embedded in this code is the first digit: 4
- User quickly catch the "400" response family

Protocol Design: Principle 4 Example

Make magic numbers meaningful

Architecture:

1xx information
2xx content
3xx redirection
4xx client error
5xx server error

Details:

200 Request was accepted and fulfilled
301 Page moved
400 Bad request
402 Payment required
403 Forbidden request
404 File not found
500 Server Error
501 Not implemented

Protocol Design: Principle 5

Scalability: Design for Expansion!

- If your protocol is good, it will be revised and extended later on (again and again!). Prepare for this from the start!
- Assign meaningful numbers or bit masks as described in principle 4, and reserve bits and fields for future use
- Indicate your protocol version immediately after handshaking (like: "HTTP/1.0")
- Force both connections to announce and match their protocol versions immediately after handshaking
- Thus if a fatal design flaws are found after a year or two, upgrade your protocol to next version and slowly deprecate the old version
- The backbone protocol of the Internet, IP, does exactly this! and that helps makes IPv6 possible! (the IP version is an integral part of the IP header!)

Protocol Design: Principle 6

Don't be stingy with information

- never hide relevant information from the other side (unless there is a security concern)
- Practically it means: each end of the connection should be able to query the other side for any relevant information
- Example: In the **BFTP** server/client project
 - ◆ the client should be able to query the server if a file exists before attempting to retrieve it, or get a list of files in a directory
 - ◆ Otherwise, we will never be able to know if a file cannot be retrieved due to server error connection problem? or it simply does not exist?
 - ◆ could be very frustrating or lead to inefficient actions

Protocol Design: Principle 7

Document your protocol precisely !!!

- Write a clear and full design specification of your protocol before you implement it
- You cannot implement a protocol which was not clearly designed and well thought
- For example, it is a bad idea to have a "restart connection" command without documenting what exactly should happen when this command is issued? What to do with partial buffers? Late packets? How many consecutive restarts are ok? etc.

Protocol Design: Principle 8

Postel's Law: "be conservative in what you do, be liberal in what you accept from others."

- This was originally coined in RFC 761, the document specifying TCP
- This is a very important, and widely known principle, yet also widely misunderstood
- The most notorious misapplication of this principle was in the implementation of early HTML parsers.
- Based on this idea, the parsers would take in any old junk that vaguely resembled HTML and try as hard as possible to display something on the browser
- The result of this extreme laxity was more than a decade of the nightmare known as "tag soup" which is only now beginning to heal from

Protocol Design: Postel's law

Postel's Law: "be conservative in what you do, be liberal in what you accept from others."

- The real meaning of the Robustness Principle is not that erroneous input should be accepted as valid, but that erroneous input should not cause catastrophic failure!
- Valid parts of a partially-erroneous input should be accepted if possible, and that diagnostics should be given for erroneous input when feasible
- An HTML parser implementation that properly followed this rule would, upon receiving "tag soup" HTML
 - ◆ produce a warning message that the HTML was invalid
 - ◆ hopefully display some information about what was wrong (e.g. unclosed anchor tag, missing doctype, etc)
 - ◆ and only then try to (or give the option to) display the parser's best approximation of what the author meant

Protocol Design: Principle 9

Design for security from the start

- Security is a common problem to many of the standard protocols, which we live with its detrimental effects every day
- These protocols, designed when the Internet was in its infancy as an academic and governmental experiment, were not designed with security in mind
- This is what facilitates spam, denial-of-service, phishing, privacy invasion, and all other sorts of Internet security problems
- Today, however, it is unacceptable to design a new protocol without giving it serious thought from the start
- Experience shows that if it is not done at the start, it may become too hard to do after a protocol has been widely deployed
- Encryption should be a layer: once the encryption layer is removed, the protocol should continue to adhere to the design principles articulated above

Learn From Examples: Common Internet Protocols

SMTP – Simple Mail Transport Protocol

Described by RFC 2821 (RFC = Request For Comments)

```
CLIENT: <<client connects to service port 25>> # HANDSHAKING
CLIENT: HELO shark.braude.ac.il # Sending host identifies itself
SERVER: 250 OK Hello shark, glad to meet you # Server acknowledges
CLIENT: MAIL FROM: <dan@braude.ac.il> # Identify sending user/domain
SERVER: 250 <dan@braude.ac.il>... Sender ok # Server acknowledges
CLIENT: RCPT TO: ran@stimpy.com # Identify target user
SERVER: 250 root... Recipient ok # Server acknowledges
CLIENT: DATA
SERVER: 354 Enter mail, end with "." on a line by itself
CLIENT: Hi Fred: Frenchy called. He wants to share
CLIENT: options, cards,
CLIENT: and a large collection of old baseball bats
CLIENT: Lehitraot,
CLIENT: Dan
CLIENT: . # End of multiline send
SERVER: 250 WAA01865 Message accepted for delivery
CLIENT: QUIT # Client (email sender) signs off
SERVER: 221 stimpy.com closing connection # Server disconnects
CLIENT: <<client hangs up>>
```

SMTP: Protocol Design

- SMTP is used for uploading mail to a mail server
- Client requests have a simple command line format:
 - ◆ HELO ...
 - ◆ MAIL ...
 - ◆ DATA ...
 - ◆ RCPT ...
- Server responses consisting of a status code followed by an informational message:
 - 250 <dan@braude.ac.il>... Sender ok
 - 221 stimp.com closing connection
- Server response consists of a status code and a human message
- Protocol software uses the status code and usually ignores the human part
- The **DATA** command sends the mail body, terminated by a line consisting of a single dot

SMTP: Main Commands

- SMTP is one of the oldest application layer protocols which is still in high use on the Internet today
- It is simple, effective, and has withstood the test of time

HELO <sendinghostname>

This command initiates the SMTP conversation.
The host connecting to the remote SMTP server identifies itself by it's fully qualified DNS host name.

MAIL From:<source email address>

This is the start of an email message.
The source email address is what will appear in the "From:" field of the message.

RCPT To:<destination email address>

This identifies the recipient of the email message.
This command can be repeated multiple times for a given message in order to deliver a single message to multiple recipients.

For more details look at: <http://the-welters.com/professional/smtp.html>

POP3 – Retrieve mail from server

```

CLIENT: <<client connects to service port 110>>
SERVER: +OK POP3 server ready <1896.6971@mailgate.dobbs.org>
CLIENT: USER bob
SERVER: +OK bob
CLIENT: PASS redqueen
SERVER: +OK bob's maildrop has 2 messages (320 octets)
CLIENT: STAT
SERVER: +OK 2 320
CLIENT: LIST
SERVER: +OK 2 messages (320 octets)
SERVER: 1 120
SERVER: 2 200
SERVER: .
CLIENT: RETR 1
SERVER: +OK 120 octets
SERVER: <the POP3 server sends the text of message 1>
SERVER: .
CLIENT: DELE 1
SERVER: +OK message 1 deleted
CLIENT: RETR 2
SERVER: +OK 200 octets
SERVER: <the POP3 server sends the text of message 2>
SERVER: .
CLIENT: DELE 2
SERVER: +OK message 2 deleted
CLIENT: QUIT
SERVER: +OK dewey POP3 server signing off (maildrop empty)
CLIENT: <<client hangs up>>

```

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

181

POP3 – Client Commands

- Client commands always start with a 4 characters code

```

USER <username>
PASS <password>
STAT
LIST
RETR <message-id>
DELE <message-id>
QUIT

```

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

182

POP3 – Server Commands

- Server has only two response modes: **+OK**, **-ERR**
- Which are essentially “+” and “-”, where “**OK**” and “**ERR**” are the “human parts”
- For some client commands, the server status line is followed by data which ends with a single “.” line

```
+OK POP3 server ready <1896.6971@mailgate.dobbs.org>
+OK bob
+OK bob's maildrop has 2 messages (320 octets)
+OK 2 320
-ERR never heard of jim
```

<http://www.pnambic.com/Goodies/POP3Ref.html>

IMAP - Internet Message Access Protocol

- A newer post office protocol designed in a slightly different style
- **IMAP** was designed to replace **POP3**
- Excellent example of a mature and powerful design worth studying and following its principles
- In the next example, user **ilanitk** is logging to a mail server to **retrieve** her email
(well, it's not Ilanit who is doing it, it's outlook or gmail client without her knowing about it)

IMAP - Internet Message Access Protocol

```

CLIENT: <<client connects to service port 143>>
SERVER: * OK iserver.com IMAP4rev1 v12.264 server ready
CLIENT: A001 USER "ilanitk" "june1987"
SERVER: * OK User ilanitk authenticated
CLIENT: A002 SELECT INBOX
SERVER: * 1 EXISTS
SERVER: * 1 RECENT
SERVER: * FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
SERVER: * OK [UNSEEN 1] first unseen message in /var/spool/mail/dan
SERVER: A002 OK [READ-WRITE] SELECT completed
CLIENT: A003 FETCH 1 RFC822.SIZE                               Get message sizes
SERVER: * 1 FETCH (RFC822.SIZE 2545)
SERVER: A003 OK FETCH completed
CLIENT: A004 FETCH 1 BODY[HEADER]                               Get first message header
SERVER: * 1 FETCH (RFC822.HEADER {1425}
<<server sends 1425 octets of message payload>>
SERVER: )
SERVER: A004 OK FETCH completed
CLIENT: A005 FETCH 1 BODY[TEXT]                               Get first message body
SERVER: * 1 FETCH (BODY[TEXT] {1120}
<<server sends 1120 octets of message payload>>
SERVER: )
SERVER: * 1 FETCH (FLAGS (\Recent \Seen))
SERVER: A005 OK FETCH completed
CLIENT: A006 LOGOUT
SERVER: * BYE iserver.com IMAP4rev1 server terminating connection
SERVER: A006 OK LOGOUT completed
CLIENT: <<client hangs up>>

```

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

185

IMAP - Internet Message Access Protocol

- The standard IMAP procedure is to leave messages on the server instead of retrieving copies
- Email is only accessible when "on-line" (from different locations, and different devices)
- Suited to a world of "always-on/anywhere" connections
- Messages remain on the server, until deleted by the user
- Messages can be accessed by multiple client computers
- Clear advantage when you use more than one computer to check your email (laptop, tablet, smartphone)
- Microsoft "MAPI" is a proprietary variation for their outlook/exchange client/server model (does not work for anything else)

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

186

IMAP - Internet Message Access Protocol

- IMAP uses the "Message Length in Advance Technique":
- instead of ending the payload with a **dot**, the payload length is sent in advance
- This makes life harder on the server a little bit:
 - ◆ messages have to be composed ahead of time
 - ◆ messages cannot be streamed after the send initiation
- But makes life easier for the client
 - ◆ Client can know in advance storage and buffer sizes it will need to process the message

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

187

IMAP - Internet Message Access Protocol

- Each response is tagged with a **sequence label** supplied by the client
- In the example above they have the form **A000n**, but the client could have generated any token into that slot
- This feature makes it possible for **IMAP** commands to be streamed to the server without waiting for the responses
- A state machine in the client can then simply interpret the responses and payloads as they come back
 - ◆ This technique cuts down on latency

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

188

RFC – Request For Comments

- Protocol design life cycle starts with an RFC
- RFC's are publications made by Internet Engineering Task Force (IETF)
- IETF develops and promotes Internet standards
- Founded by the US government around 1969 (part of the ARPANET project), but is now a very large international organization with many sub-organizations (acm, IEEE)
- Official RFC's database: <http://www.rfc-editor.org/rfc.html>
- For example, here is RFC 3501 (March 2003) for the IMAP specifications:
<http://www.rfc-editor.org/rfc/rfc3501.txt>
<http://www.rfc-editor.org/rfc/rfc4978.txt>
- (read it and write a similar doc for BFTP ...)

Client Server Programming - Slide Figures/quotes from Andrew Tanenbaum Computer Networks book (Teacher Slides)

189

PARALLEL PROGRAMMING

Adapted from David Beazley's paper:
 "An Introduction to Python Concurrency"
 Presented at USENIX Technical Conference
 San Diego, June, 2009

David Beazley: <http://www.dabeaz.com>

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

190

Code Examples and Files

- Thanks Dave Beazley for contributing his fantastic set of source code examples on Python concurrency and parallel programming
- We have also added a few more examples and rephrased Dave's examples to suite our course objectives
- Our source code repository can be retrieved from:
http://tinyurl.com/samyz/os/projects/PARALLEL_PROGRAMMING_LAB.zip
- Dave Beazley original resources can be retrieved from:
<http://www.dabeaz.com/usenix2009/concurrent/>

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

191

Concurrent Programming

- Doing more than one thing at a time
- Writing programs that can work on more than one thing at a time
- Of particular interest to programmers and systems designers
- Writing code for running on "big iron"
- But also of interest for users of multicore desktop computers
- Goal is to go beyond the user manual and tie everything together into a "bigger picture."

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

192

Examples

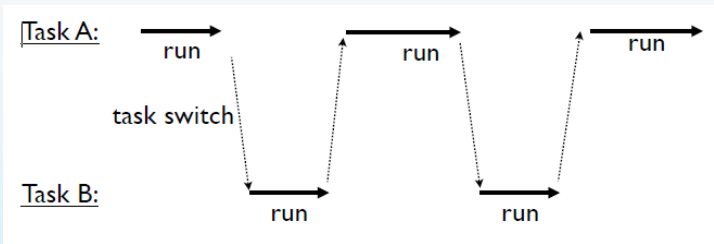
- Web server that communicates with thousand clients (Google)
- Web client (Chrome or Firefox) that displays 10 or 20 tabs
- In the same process it may do the following tasks concurrently:
 - ◆ Download several images, audio files, movies (concurrently)
 - ◆ Display an image and a movie
 - ◆ Connect to several servers
- Microsoft Word can do several tasks at the same time
 - ◆ Let the user insert text with no waits or interrupts
 - ◆ Download/upload stuff
 - ◆ Backup the document every few seconds
 - ◆ Check spelling and grammar (and even mark words as the user is typing)
- Image processing software that uses 8 CPU cores for parallel intense matrix multiplications

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

193

Multitasking

- Concurrency usually means multitasking



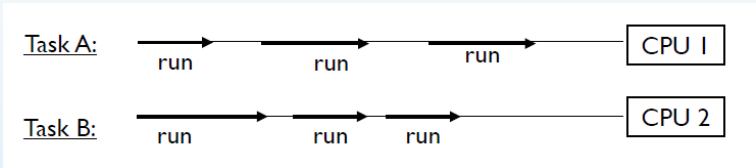
- If only one CPU is available, the only way it can run multiple tasks is by rapidly switching between them in one of two way:
 - ◆ Process context switch (two processes)
 - ◆ Thread context switch (two threads in one process)

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

194

Parallel Processing

- If you have many CPU's or CORE's then you can have true parallelism: the two tasks run simultaneously



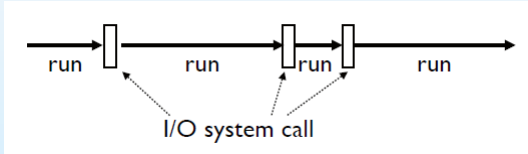
- If the total number of tasks exceeds the number of CPUs, then some CPU's must multitask (switch between tasks)

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

195

Task Execution

- Every task executes by alternating between CPU processing and I/O handling:
 - ◆ disk read/write
 - ◆ network send/receive



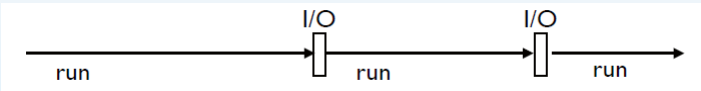
- For I/O, tasks must wait (sleep): the underlying system will carry out the I/O operation and wake the task when it's finished

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

196

CPU Bound Tasks

- A task is "CPU Bound" if it spends most of its time processing with little I/O



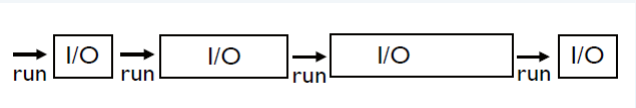
- Examples:
 - ◆ Image processing
 - ◆ Weather forecast system
 - ◆ Heavy mathematical computations

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

197

I/O Bound Tasks

- A task is "I/O Bound" if it spends most of its time waiting for I/O



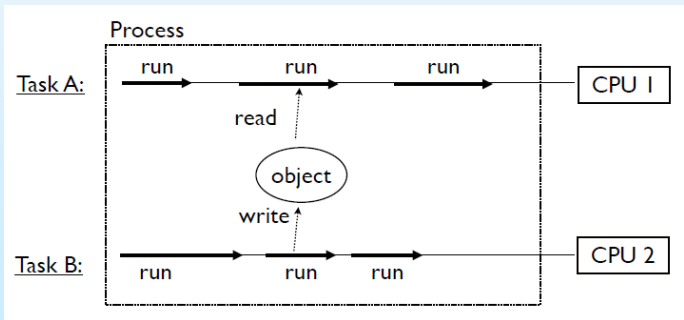
- Examples:
 - ◆ Reading input from the user (text processors)
 - ◆ Networking
 - ◆ File Processing
- Most "normal" programs are I/O bound

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

198

Shared Memory

- In many cases, two tasks need to share information ("cooperating tasks") and access an object simultaneously
- Two threads within the same process always share all memory of that process
- Two independent processes on the other hand need special mechanisms to communicate between them



Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

199

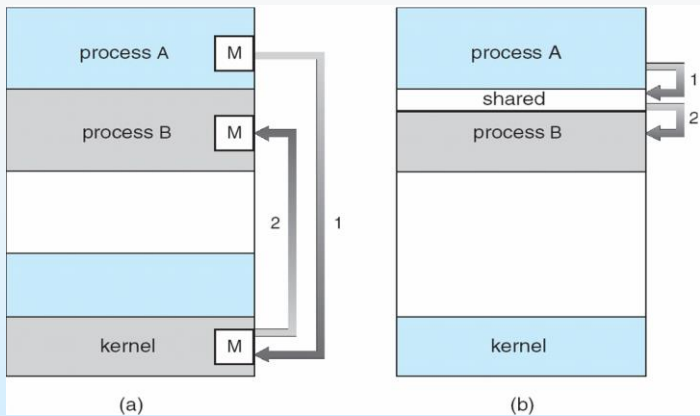
IPC – Inter Process Communication

- Processes within a system may be independent or cooperating
- Reasons for cooperating processes:
 - ◆ Information sharing
 - ◆ Computation speedup
 - ◆ Modularity
 - ◆ Convenience
- Cooperating processes need inter-process communication (IPC)
- Two models of IPC
 - ◆ Shared memory
 - ◆ Message passing

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

200

Two Types of IPC



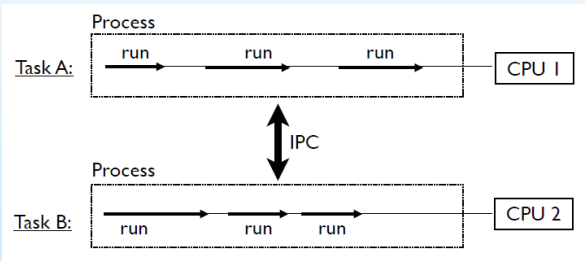
- (a) Kernel shared memory: Pipe, Socket, FIFO, mailboxes
- (b) Process shared memory (OS is not involved here!)

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

201

IPC – Inter Process Communication

- The simplest mechanism for two processes to communicate are
 - ◆ Pipe
 - ◆ FIFO
 - ◆ Shared memory (memory mapped regions)
 - ◆ Socket



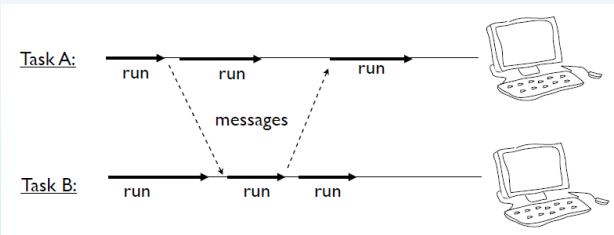
- Processes can also communicate through the file system, but it tends to be too slow and volatile (like suppose disk is full or bad)

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

202

Distributed Computing

- Tasks may be running on distributed systems
- Sometimes on two different continents



- Cluster of workstations
- Usually: communication via sockets (or MPI)

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

203

Programmer Performance

- Programmers are often able to get complex systems to "work" in much less time using a high-level language like Python than if they're spending all of their time hacking C code
- In some cases scripting solutions might be even competitive with C++, C# and, especially, Java
- The reason is that when you are operating at a higher level, you often are able to find a better, more optimal, algorithm, data structures, problem decomposition schema, or all of the above

"The best performance improvement is the transition from the nonworking to the working state."

- John Ousterhout

"Premature optimization is the root of all evil."

- Donald Knuth

"You can always optimize it later."

- Unknown

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

204

Intel VLSI Tools as an Example

- In recent years, a fundamental transition has been occurring in the way industry developers write computer programs
- The change is a transition from system programming languages such as C or C++ to scripting languages such as Perl, Python, Ruby, JavaScript, PHP, etc.

Tool	C/C++ lines	TCL/PERL/PYTHON lines
VLSI CAD TOOL 1	510,946	644,272
VLSI CAD TOOL 2	581333	368435
VLSI CAD TOOL 3	1,517,917	1,421,400
VLSI CAD TOOL 4	422,239	1,332,767

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

205

Performance is Irrelevant!

- Many concurrent programs are "I/O bound"
- They spend virtually all of their time sitting around waiting for
 - ◆ Clients to connect
 - ◆ Client requests
 - ◆ Client responses
- Python can "wait" just as fast as C
- One exception: if you need an extremely rapid response time as in real-time systems

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

206

You Can Always Go Faster

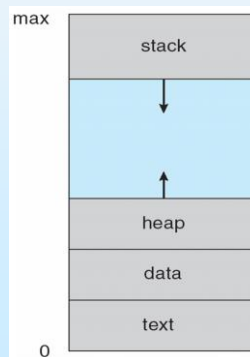
- Python can be extended with **C code**
- Look at **ctypes**, **Cython**, **Swig**, etc.
- If you need really high-performance, you're not coding Python -- you're using C extensions
- This is what most of the big scientific computing hackers are doing
- It's called: "**using the right tool for the job**"

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

207

Process Concept Review

- A process (or job) is a program in execution
- A process includes:
 1. **Text** (program code)
 2. **Data** (constants and fixed tables)
 3. **Heap** (dynamic memory)
 4. **Stack** (for function calls and temporary variables)
 5. **Program counter** (current instruction)
 6. **CPU registers**
 7. **Open files table** (including sockets)
- To better distinguish between a program and a process, note that a single Word processor program may have 10 different processes running simultaneously
- Consider multiple users executing the same Internet explorer (each has the 6 things above)
- Computer activity is the sum of all its processes

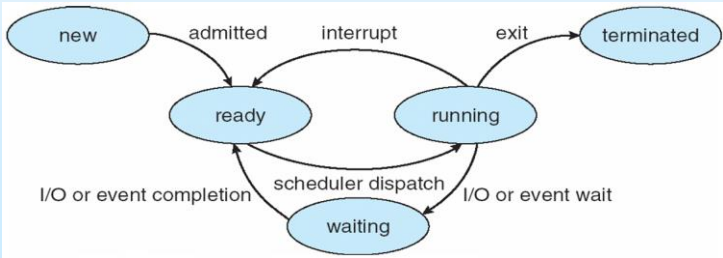


Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

208

Process States

- 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

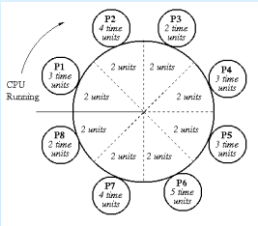


Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

209

CPU Process Scheduling

- Modern operating systems can run hundreds (and even thousands) of processes in parallel
- Of course, at each moment, only a single process can control a CPU, but the operating system is switching processes every 15 milliseconds (on average) so that at 1 minute, an operating system can swap 4000 processes!
- The replacement of a running process with a new process is generated by an **INTERRUPT**



Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

210

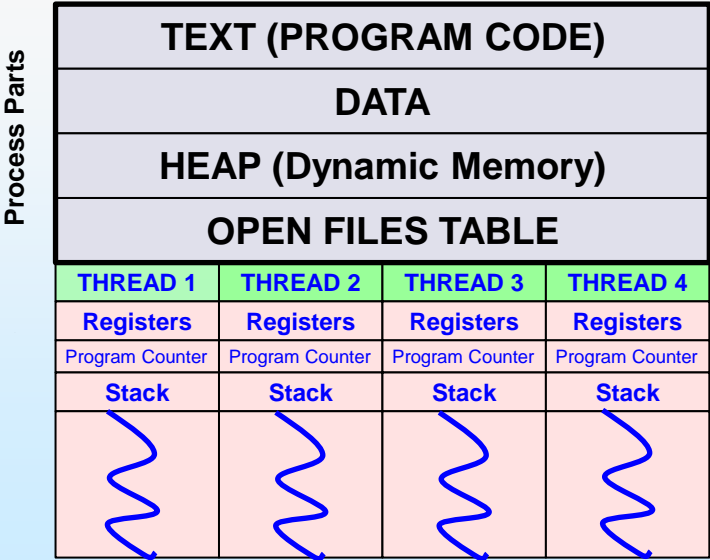
THREADS

- What most programmers think of when they hear about “concurrent programming”
- A Thread is an **independent task** running inside a program
- Shares resources with the main program (and other threads)
 - ◆ **Memory** (Program text, Data, Heap)
 - ◆ **Files**
 - ◆ **Network connections**
- Has its own **independent** flow of execution
 - ◆ **Thread stack**
 - ◆ **Thread program counter**
 - ◆ **Thread CPU registers (context)**

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

211

One Process, Many Threads!

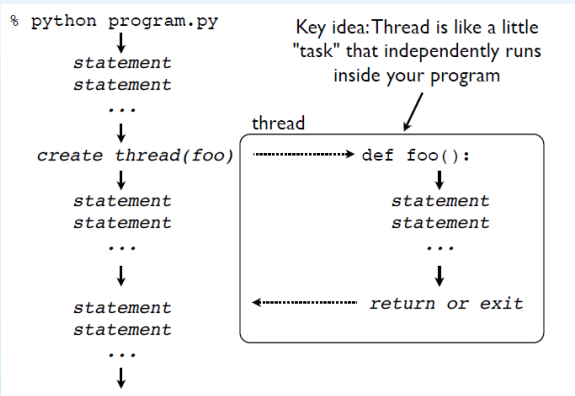


Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

212

THREADS

- Several threads within the same process can use and share
 - ◆ common variables
 - ◆ common open files
 - ◆ common networking sockets



Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

213

Threading Module

- Python threads are defined by a class
- You inherit from Thread and redefine run()

```
from threading import Thread
import time

class CountdownThread(Thread):
    def __init__(self, name, count):
        Thread.__init__(self)
        self.name = name
        self.count = count

    def run(self):
        while self.count > 0:
            print "%s:%d" % (self.name, self.count)
            self.count -= 1
            time.sleep(2)
        return
```

This code
executes in
the thread

countdown1.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

214

Launching a Thread

- To launch a thread: create a thread object and call start()

```
t1 = CountdownThread(10) # Create the thread object
t2 = CountdownThread(20) # Create another thread
t1.start()               # Launch thread t1
t2.start()               # Launch thread t2
```

countdown1.py

- Thread executes until their run method stops (return or exit)

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

215

Alternative way to launch threads

```
import time
from threading import Thread

def countdown(name, count):
    while count > 0:
        print "%s:%d" % (name, count)
        count -= 1
        time.sleep(2)
    return

t1 = Thread(target=countdown, args=("A", 10))
t2 = Thread(target=countdown, args=("B", 20))
t1.start()
t2.start()
```

countdown2.py

- Creates a Thread object, but its run() method just calls the countdown function

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

216

Joining a Thread

- Once you start a thread, it runs independently
- Use **t.join()** to wait for a thread to exit
- This only works from other threads
- A thread can't join itself!

```
t = Thread(target=foo, args=(N/2,))
t.start()
# Do some work ...
t.join()    # Wait for the thread to exit
# Continue your work ...
```

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

217

Daemonic Threads

- If a thread runs forever, make it "daemonic"
- If you don't do this, the interpreter will lock when the main thread exits - waiting for the thread to terminate (which never happens)
- Normally you use this for background tasks

```
t.daemon = True
t.setDaemon(True)
```

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

218

Access to Shared Data

- Threads share all of the data in your program
- Thread scheduling is non-deterministic
- Operations often take several steps and might be interrupted mid-stream (non-atomic)
- Thus, access to any kind of shared data is also non-deterministic
- (which is a really good way to have your head explode)

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

219

Accessing Shared Data

- Consider a shared object

```
x = 0
```

- And two threads that modify it

```
#Thread 1
x = x + 1
```

```
#Thread 2
x = x - 1
```

- It's possible that the resulting value will be unpredictably corrupted

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

220

Accessing Shared Data

- Is this a serious concern?
- YES! This is a dead serious matter!
- Look what happens in the following example !?

```
def foo():
    global x
    for i in xrange(1000000):
        x += 1

def bar():
    global x
    for i in xrange(1000000):
        x -= 1

t1 = Thread(target=foo)
t2 = Thread(target=bar)
t1.start()
t2.start()
```

RACE_WARS/race_1.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

221

The Therac-25 Accidents

- Machine for radiation therapy
 - ◆ Software control of electron accelerator and electron beam/Xray production
 - ◆ Software control of dosage
- Software errors caused the death of several patients
- A series of race conditions on shared variables and poor software design

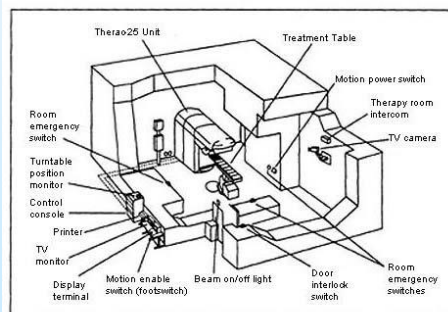


Figure 1. Typical Therac-25 facility

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

222

Race Conditions

- The corruption of shared data due to thread scheduling is often known as a "race condition."
- It's often quite diabolical - a program may produce slightly different results each time it runs (even though you aren't using any random numbers!)
- Or it may just flake out mysteriously once every two weeks

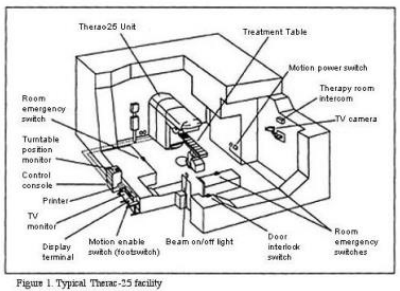


Figure 1. Typical Therac-25 facility

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

223

THREADS – Summary (1)

- Threads are easier to create than processes:
 - ◆ Threads do not require a separate address space!
- Multithreading requires careful programming!
- Threads share data structures that should only be modified by one thread at a time! (mutex lock)
- A problem in one thread can
 - ◆ Cause the parent process to block or crash
 - ◆ and thus kill all other threads!
- Therefore a lot of caution must be taken so that different threads don't step on each other!

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

224

THREADS – Summary (2)

- Unlike threads, **processes** do not share the same address space and thus are truly independent of each other.
- Threads are considered lightweight because they use far less resources than processes (no need for a full context switch)
- Threads, on the other hand, share the same address space, and therefore are interdependent
- Always remember the golden rules:
 - ◆ **Write stupid code and live longer (KISS)**
 - ◆ **Avoid writing any code at all if you don't have to! (Bjarn Stroustrup, inventor of C++)**

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

225

Thread Synchronization

- Identifying and fixing a race condition will make you a better programmer (e.g., it "builds character")
- However, you'll probably never get that month of your life back ...
- To fix : You have to synchronize threads
- **Synchronization Primitives:**
 - ◆ Lock
 - ◆ Semaphore
 - ◆ BoundedSemaphore
 - ◆ Condition
 - ◆ Event

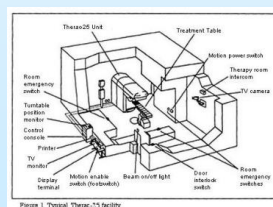


Figure 1. Typical Therese-25 facility

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

226

Mutex Locks

- Probably the most commonly used synchronization primitive
- Mostly used to synchronize threads so that only one thread can make modifications to shared data at any given time
- Has only two basic operations

```
from threading import Lock

m = Lock()
m.acquire()
m.release()
```

- Only one thread can successfully acquire the lock at any given time
- If another thread tries to acquire the lock when its already in use, it gets blocked until the lock is released

Use of Mutex Locks

- Commonly used to enclose critical sections

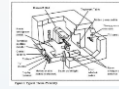
```
x = 0
x_lock = threading.Lock()

Thread-1                                Thread-2
-----                                -----
...                                     ...
x_lock.acquire()                         x_lock.acquire()
Critical Section x = x + 1                x = x - 1
x_lock.release()                         x_lock.release()
...                                     ...
```

- Only one thread can execute in critical section at a time (lock gives exclusive access)

Using a Mutex Lock

- It is your responsibility to identify and lock all "critical sections" !



```
x = 0
x_lock = threading.Lock()
```

```
Thread-1
-----
...
x_lock.acquire()
x = x + 1
x_lock.release()
...
```

```
Thread-2
-----
```

```
...
x = x - 1
...
```

If you use a lock in one place, but not another, then you're missing the whole point. All modifications to shared state must be enclosed by lock acquire()/release().

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

229

Lock Management

- Locking looks straightforward
- Until you start adding it to your code ...
- Managing locks is a lot harder than it looks!
- Acquired locks must always be released!
- However, it gets evil with exceptions and other non-linear forms of control-flow
- Always try to follow this prototype:

```
x = 0
x_lock = threading.Lock()

# Example critical section
x_lock.acquire()
try:
    statements using x
finally:
    x_lock.release()
```

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

230

Lock and Deadlock

- Avoid writing code that acquires more than one mutex lock at a time

```
mx = Lock()
my = Lock()

mx.acquire()
# statement using x
my.acquire()
# statement using y
my.release()
# ...
mx.release()
```

- This almost invariably ends up creating a program that mysteriously deadlocks (even more fun to debug than a race condition)
- Remember Therac-25 ...



Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

231

Semaphores

- A counter-based synchronization primitive
 - ◆ acquire() - Waits if the count is 0, otherwise decrements the count and continues
 - ◆ release() - Increments the count and signals waiting threads (if any)
- Unlike locks, acquire()/release() can be called in any order and by any thread

```
from threading import Semaphore
m = Semaphore(n) # Create a semaphore
m.acquire() # Acquire
m.release() # Release
```

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

232

Semaphores Uses

- **Resource control:** limit the number of threads performing certain operations such as database queries, network connections, disk writes, etc.
- **Signaling:** Semaphores can be used to send "signals" between threads
- For example, having one thread wake up another thread.

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

233

Resource Control

- **Using Semaphore to limit Resource:**

```
import requests
from threading import Semaphore

sem = Semaphore(5) #    Max: 5-threads
def get_link(url):
    sem.acquire()
    try:
        req = requests.get(url)
        return req.content
    finally:
        sem.release()
```

- Only 5 threads can execute the `get_link` function
- This make sure we do not put too much pressure on networking system

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

234

Thread Signaling

- Using a semaphore to “send a signal”:

```
sem = Semaphore(0)
```

```
# Thread 1
...
statements
statements
statements
sem.release()
...
```

```
# Thread 2
...
sem.acquire()
statements
statements
statements
...
```

- Here, acquire() and release() occur in different threads and in a different order
- Thread-2 is blocked until Thread-1 releases “sem”.
- Often used with producer-consumer problems

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

235

Threads Summary

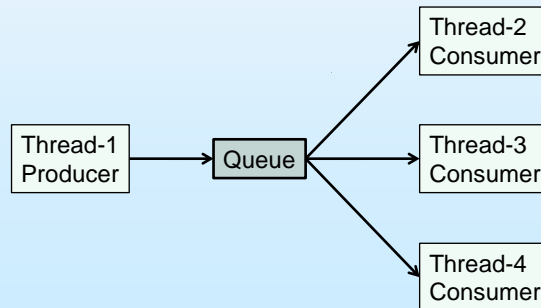
- Working with all of the synchronization primitives is a lot trickier than it looks
- There are a lot of nasty corner cases and horrible things that can go wrong
 - ◆ Bad performance
 - ◆ deadlocks
 - ◆ Starvation
 - ◆ bizarre CPU scheduling
 - ◆ etc...
- All are valid reasons to not use threads, unless you do not have a better choice

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

236

Threads and Queues

- Threaded programs are easier to manage if they can be organized into producer/consumer components connected by queues
- Instead of "sharing" data, threads only coordinate by sending data to each other
- Think Unix "pipes" if you will...



Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

237

Queue Library Module

- Python has a thread-safe queuing module
- Basic operations

```

from Queue import Queue

q = Queue([maxsize]) # Create a queue
q.put(item)           # Put an item on the queue
q.get()               # Get an item from the queue
q.empty()              # Check if empty
q.full()               # Check if full
  
```

- Usage: Try to strictly adhere to get/put operations. If you do this, you don't need to use other synchronization primitives!

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

238

Queue Usage

- Most commonly used to set up various forms of producer/consumer problems

```
from Queue import Queue
q = Queue()
```

Producer Thread

```
for item in produce_items():
    q.put(item)
```

Consumer Thread

```
while True:
    item = q.get()
    consume_item(item)
```

- Critical point : You don't need locks here !!!
(they are already embedded in the Queue object)

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

239

Producer Consumer Pattern

```
import time, Queue
from threading import Thread, currentThread

que = Queue.Queue()

def run_producer():
    print "I am the producer"
    for i in range(30):
        item = "packet_" + str(i) # producing an item
        que.put(item)
        time.sleep(1.0)

def run_consumer():
    print "I am a consumer", currentThread().name
    while True:
        item = que.get()
        print currentThread().name, "got", item
        time.sleep(5)

for i in range(10): # Starting 10 consumers !
    t = Thread(target=run_consumer)
    t.start()

run_producer()
```

Code:
producer_consumers_que.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

240

Queue Signaling

- Queues also have a signaling mechanism

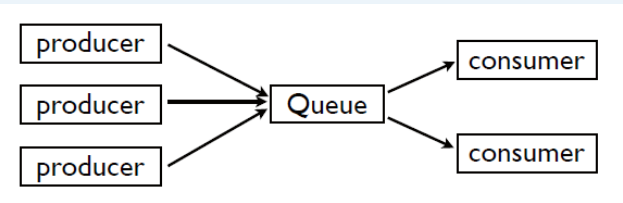
```
q.task_done()    # Signal that work is done
q.join()         # Wait for all work to be done
```

- Many Python programmers don't know about this (since it's relatively new)
- Used to determine when processing is done

<u>Producer Thread</u>	<u>Consumer Thread</u>
<pre>for item in produce_items(): q.put(item) # Wait for consumer q.join()</pre>	<pre>while True: item = q.get() consume_item(item) q.task_done()</pre>

Queue Programming

- There are many ways to use queues
- You can have as many consumers/producers as you want hooked up to the same queue



- In practice, try to keep it simple !

Task Producer

- Can be defined in a function or in a class
- Here is a simple one in a function

```
# Keep producing unlimited number of tasks
# Every task is pushed to a task_que

def task_producer(id, task_que):
    while True:
        a = random.randint(0,100)    # random int from 0 to 100
        b = random.randint(0,100)    # random int from 0 to 100
        task = "%d*%d" % (a,b)       # multiplication task
        time.sleep(3)                 # 3 sec to produce a task
        task_que.put(task)
        print "Producer %d produced task: %s" % (id, task)
```

Code:
producer_consumer_1.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

243

Worker (consumer)

- Can be defined in a function or in a class
- Here is a simple one in a function

```
# Accepts unlimited number of tasks (from task_que)
# It solves a task and puts the result in the result_que.

def worker(id, task_que, result_que):
    while True:
        task = task_que.get()
        t = random.uniform(2,3)    # Take 2-3 seconds to complete a task
        time.sleep(t)
        answer = eval(task)
        result_que.put(answer)
        print "Worker %d completed task %s: answer=%d" % (id, task, answer)
```

Code:
producer_consumer_1.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

244

Simulation: 2 producers, 3 workers

```
def simulation2():
    task_que = Queue()
    result_que = Queue()

    # Two producers
    p1 = Thread(target=task_producer, args=(1, task_que))
    p2 = Thread(target=task_producer, args=(2, task_que))

    # Three workers
    w1 = Thread(target=worker, args=(1, task_que, result_que))
    w2 = Thread(target=worker, args=(2, task_que, result_que))
    w3 = Thread(target=worker, args=(3, task_que, result_que))

    p1.start()
    p2.start()
    w1.start()
    w2.start()
    w3.start()
    # producers and workers run forever ...
```

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

Code: producer_consumer_1.py

245

Performance Test

- Consider this CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Sequential Execution:

```
count(100000000)
count(100000000)
```

- Threaded execution

```
t1 = Thread(target=count, args=(100000000,))
t1.start()
t2 = Thread(target=count, args=(100000000,))
t2.start()
```

- Now, you might expect two threads to run twice as fast on multiple CPU cores

Code: threads_perf.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

246

Performance Test

Bizarre Results

- Performance comparison (Dual-Core 2Ghz Macbook, OS-X 10.5.6)

Sequential : 24.6s

Threaded : 45.5s (1.8X slower!)

- If you disable one of the CPU cores...

Threaded : 38.0s

- Insanely horrible performance. Better performance with fewer CPU cores? It makes no sense.

Code: threads_perf.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

247

Threads Summary (1)

- To understand why this is so and how to make better use of threads, keep reading David Beazley Paper at: <http://www.dabeaz.com/usenix2009/concurrent/>
- Threads are still useful for I/O-bound apps, and do save time in these situations (which are more common than CPU-bound apps)
- For example : A network server that needs to maintain several thousand long-lived TCP connections, but is not doing tons of heavy CPU processing
- Most systems don't have much of a problem -- even with **thousands of threads**

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

248

Threads Summary (2)

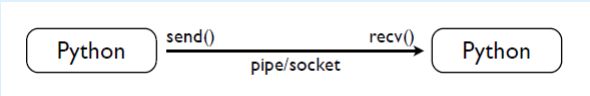
- If everything is I/O-bound, you will get a very quick response time to any I/O activity
- Python isn't doing the scheduling
- So, Python is going to have a similar response behavior as a C program with a lot of I/O bound threads
- Python threads are a useful tool, but you have to know how and when to use them
- I/O bound processing only
- Limit CPU-bound processing to C extensions (that release the GIL)
- To parallel CPU bound applications use Python's multiprocessing module ... our next topic

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

249

Multi Processing

- An alternative to threads is to run multiple independent copies of the Python interpreter
- In separate processes
- Possibly on different machines
- Get the different interpreters to cooperate by having them send messages to each other



- Each instance of Python is independent
- Programs just send and receive messages

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

250

Message Passing

- Two main issues:
 - What is a message?
 - What is the transport mechanism?
- A Message is just a bunch of bytes (buffer)
- A "serialized" representation of some data
- Could be done via files, but it's very slow and volatile

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

251

Message Transport

- Pipes
- Sockets
- FIFOs
- MPI (Message Passing Interface)
- XML-RPC (and many others)

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

252

Pipe Example 1

- The **bc.exe** (**Berkeley Calculator**) performs Math much faster than Python (think of it as a simple Matlab)
- **bc.exe** reads from **stdin** and writes to **stdout**
- It is included in the parallel programming code bundle
- Here is a **bc** program to calculate PI from term m to term n:

```
# This is not Python! This is a bc code to
# for the Gregory-Leibnitz series for of pi:
#      pi = 4/1 - 4/3 + 4/5 - 4/7 + ...

define psum(m,n) {
    auto i
    s=0
    for (i=m; i < n; ++i)
        s = s + (-1)^i * 4.0/(2*i+1.0)
    return (s)
}
```

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

253

Pipe Example 1

```
import subprocess

code = """
    define psum(m,n) {
        auto i
        s=0
        for (i=m; i < n; ++i)
            s = s + (-1)^i * 4.0/(2*i+1.0)
        return (s)
    }
"""

# This is code in a totally different language !

# Starting a pipe to the bc.exe program
p = subprocess.Popen(["bc.exe"], stdin=subprocess.PIPE, stdout=subprocess.PIPE)

# Sending code to bc by writing to the Python side of the Pipe
p.stdin.write(code)
p.stdin.write("scale=60\n")          # 60 digits precision
p.stdin.write("psum(0,1000000)\n")  # Now we do the calculation!
result = p.stdout.readline()         # Now we read the result!
p.terminate()
print result
```

IPC/pipe_to_bc_1.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

254

Pipe Example 2

- If one sub-process gets us a lot of speed, how about opening two sub-processes in parallel?

```
# Starting two pipes to the bc.exe program!
p1 = subprocess.Popen(["bc.exe"], stdin=subprocess.PIPE, stdout=subprocess.PIPE)
p2 = subprocess.Popen(["bc.exe"], stdin=subprocess.PIPE, stdout=subprocess.PIPE)

# Sending code to bc by writing to the Python side of the Pipe
p1.stdin.write(code)
p2.stdin.write(code)
p1.stdin.write("scale=60\n") # 60 digits precision
p2.stdin.write("scale=60\n")

# Now we do the calculation!
# Both processes run in parallel in the background !
p1.stdin.write("psum(0,500000)\n") # We divide the task to two parts !
p2.stdin.write("psum(500000, 1000000)\n") # Part 2

result1 = p1.stdout.readline()
result2 = p2.stdout.readline()
p1.terminate()
p2.terminate()
print Decimal(result1) + Decimal(result2)
```

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

IPC/pipe_to_bc_2.py

255

Pipe Example 3

- That worked all right, but if we want to use our 8 CPU cores, we need to be more prudent!

```
def bc_worker(a,b):
    p = subprocess.Popen(["bc.exe"], stdin=subprocess.PIPE, stdout=subprocess.PIPE)
    p.stdin.write(code)
    p.stdin.write("scale=60\n") # 60 digits precision
    p.stdin.write("psum(%d,%d)\n" % (a,b))
    return p

# 8 parallel sums of 500K terms chunks ... (total 4M terms)
procs = []
chunk = 500000
for i in range(8):
    a = i * chunk
    b = (i+1) * chunk
    p = bc_worker(a,b)
    procs.append(p)

getcontext().prec = 60
result = Decimal("0.0")
for p in procs:
    r = p.stdout.readline()
    p.terminate()
    result += Decimal(r)
```

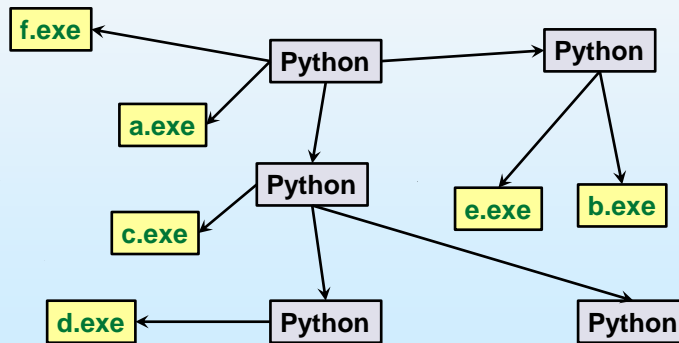
IPC/pipe_to_bc_3.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

256

The Big Picture

- Can easily have 10s-100s-1000s of communicating Python interpreters and external programs through pipes and sockets
- However, always keep the "golden rules" in mind ...



Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

Golden rule: He who has the gold makes the rules

257

The Multiprocessing Module

- This is a module for writing concurrent programs based on communicating processes
- A module that is especially useful for concurrent CPU-bound processing
- Here's the cool part:
You already know how to us multiprocessing!
- It is exactly as using Threads, just replace "Thread" with "Process"
- Instead of "Thread" objects, you now work with "Process" objects
- But! One small difference: you need to use Queue's for process communication (or else you have independent processes with no shared data at all)

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

258

Multiprocessing Example 1

- Define tasks using a Process class
- You inherit from Process and redefine run()

```
import time, os
from multiprocessing import Process

print "Parent Process id:", os.getpid()

class CountdownProcess(Process):
    def __init__(self, name, count):
        Process.__init__(self)
        self.name = name
        self.count = count

    def run(self):
        print "Child Process id:", os.getpid()
        while self.count > 0:
            print "%s:%d" % (self.name, self.count)
            self.count -= 1
            time.sleep(2)

        return
```

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

countdownp1.py

259

Multiprocessing Example 1

- To launch, same idea as with threads
- You inherit from Process and redefine run()

```
if __name__ == '__main__':
    p1 = CountdownProcess("A", 10) # Create the process object
    p1.start()                     # Launch the process

    p2 = CountdownProcess("B", 20) # Create another process
    p2.start()                     # Launch
```

countdownp1.py

- Processes execute until run() stops
- critical detail: Always launch in main as shown (or else your Windows will crash)

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

260

Multiprocessing Example 2

- Alternative method of launching processes is by using simple functions instead of classes

countdownp2.py

```
def countdown(name, count):
    print "Process id:", os.getpid()
    while count > 0:
        print "%s:%d" % (name, count)
        count -= 1
        time.sleep(2)
    return

# Sample execution
if __name__ == '__main__':
    p1 = Process(target=countdown, args=("A", 10))
    p2 = Process(target=countdown, args=("B", 20))

    p1.start()
    p2.start()
```

- Creates two Process objects, but their run() method just calls the countdown function

Does it Work ?

- Consider this CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Sequential Execution:

```
count(100000000)
count(100000000) → 24.6s
```

- Multiprocessing Execution

```
p1 = Process(target=count, args=(100000000,))
p1.start()
p2 = Process(target=count, args=(100000000,))
p2.start() → 12.5s
```

- Yes, it seems to work

Other Process Features

- Joining a process (waits for termination)

```
p = Process(target=somefunc)
p.start()
...
p.join()
```

- Making a daemon process

```
p = Process(target=somefunc)
p.daemon = True
p.start()
```

- Terminating a process

```
p = Process(target=somefunc)
...
p.terminate()
```

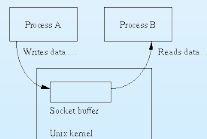
- These mirror similar thread functions

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

263

Distributed Memory

- Unlike Threads, with multiprocessing, there are no shared data structures, in fact no sharing at all !
- Every process is completely isolated!
- Since there are no shared structures, forget about all of that locking business
- Everything is focused on messaging



<http://fxa.noaa.gov/kelly/ipc/>

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

264

Pipes

- A channel for sending/receiving objects

```
(c1, c2) = multiprocessing.Pipe()
```

- Returns a pair of connection objects (one for each end-point of the pipe)

- Here are methods for communication

```
c.send(obj)           # Send an object
c.recv()              # Receive an object

c.send_bytes(buffer)  # Send a buffer of bytes
c.recv_bytes([max])   # Receive a buffer of bytes

c.poll([timeout])     # Check for data
```

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

265

Pipe Example 1

■ A simple data consumer

```
from multiprocessing import Process, Pipe

def consumer(p1, p2):
    p1.close()  # Close producer's end (not used)
    while True:
        try:
            item = p2.recv()
        except EOFError:
            break
        print "Consumer got:", item
```

■ A simple data producer

```
def producer(outp):
    print "Process id:", os.getpid()
    for i in range(10):
        item = "item" + str(i)  # make an item
        print "Producer produced:", item
        outp.send(item)
```

pipe_for_producer_consumer.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

266

Pipe Example 1

- **Launching Consumer and Producer**
- The consumer runs in a child process
- But the producer runs in the parent process
- Communication is from parent to child

```
if __name__ == '__main__':
    p1, p2 = Pipe()

    c = Process(target=consumer, args=(p1, p2))
    c.start()

    # Close the input end in the producer
    p2.close()

    run_producer(p1)

    # Close the pipe
    p1.close()
```

pipe_for_producer_consumer.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

267

Message Queues

- multiprocessing also provides a queue
- The programming interface is the same

```
from multiprocessing import Queue

q = Queue()
q.put(item)    # Put an item on the queue
item = q.get() # Get an item from the queue
```

- There is also a joinable Queue

```
from multiprocessing import JoinableQueue

q = JoinableQueue()
q.task_done()    # Signal task completion
q.join()         # Wait for completion
```

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

268

Queue Implementation

- Queues are implemented on top of pipes
- A subtle feature of queues is that they have a "feeder thread" behind the scenes
- Putting an item on a queue returns immediately
 - ◆ Allowing the producer to keep working
- The feeder thread works on its own to transmit data to consumers

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

269

Deadlocks

- Assume Alice wants to transfer money to Bob and at the same time Bob wants to transfers money to Alice
- Alice's bank grabs a lock on Alice's account, then asks Bob's bank for a lock on Bob's account
- Bob's bank locked Bob's account and is now asking for a lock on Alice's account
- Bang! you have a deadlock!



http://www.eveninghour.com/images/online_transfer2.jpg

Code:
DEADLOCK/bank_account_1.py
DEADLOCK/bank_account_2.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

270

Dining Philosophers

- 5 philosopher with 5 forks sit around a circular table
- The forks are placed between philosophers
- Each philosopher can be in one of three states:
 - ◆ **Thinking** (job is waiting)
 - ◆ **Hungry** (job ready to run)
 - ◆ **Eating** (job is running)
- To eat, a philosopher must have two forks
 - ◆ He must first obtain the **first fork** (left or right)
 - ◆ After obtaining the **first fork** he proceeds to obtain the **second fork**
 - ◆ Only after having two forks he is allowed to eat
 - ◆ (The two forks cannot be obtained simultaneously!)
- Analogy: a process that needs to access two resources: a disk and printer for example



Code: DEADLOCK/dining_philosophers.py

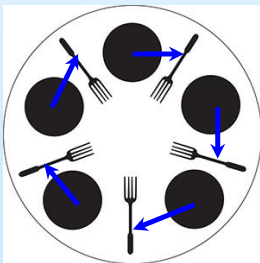
Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

271

Dining Philosophers: Deadlock

- A Deadlock is a situation in which all 5 philosophers are hungry but none can eat forever since each philosopher is waiting for a fork to be released
- Sometimes this situation is called: **full starvation**
- In operating systems, a philosopher represents a thread or a process that need access to two resources (like two files or a disc and printer) in order to proceed
- Operating system puts every process into a device Queue each time it needs to access a device (disc, memory, or CPU)

Typical deadlock situation:
Each Philosopher grabbed the left fork and waits for the right fork



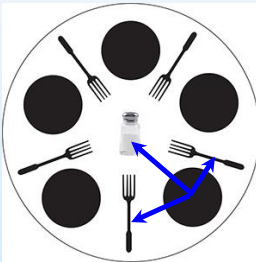
Code: DEADLOCK/dining_philosophers.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

272

Dining Philosophers: Solution 1

- A philosopher who wants to eat first picks up the salt shaker on the table
- Assume only one salt shaker exists!
- All other philosophers that do not have the salt shaker must release their forks
- The philosopher that got the salt shaker picks up his forks, eats and when finishes must put the salt shaker back at the table center
- This solution works but is not optimal: only one philosopher can eat at any given time
- if we further stipulate that the philosophers agree to go around the table and pick up the salt shaker in turn, this solution is also fair and ensures no philosopher starves.



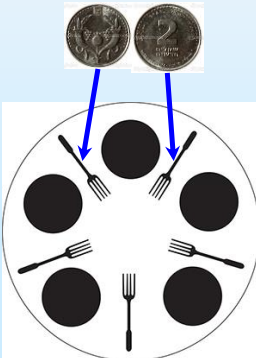
Code: DEADLOCK/dining_philosophers.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

273

Dining Philosophers: Solution 2

- Each philosopher flips a coin:
 - ◆ Heads, he tries to grab the right fork
 - ◆ Tails, he tries to grab the left fork
- If the second fork is busy, release the first fork and try again
- With probability 1, he will eventually eat
- Again, this solution relies on defeating circular waiting whenever possible and then resorts to breaking 'acquiring while holding' as assurance for the case when two adjacent philosophers' coins both come up the same.
- Again, this solution is fair and ensures all philosophers can eat eventually.

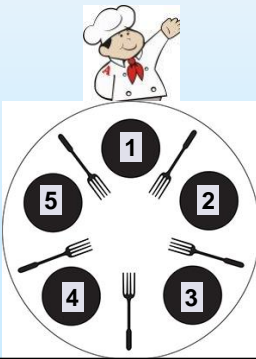


Code: DEADLOCK/dining_philosophers.py

Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

Dining Philosophers: Solution 3

- The chef that cooked the meal dictates who should eat and when to prevent any confusion. This breaks the 'blocking shared resources' condition.
- The chef assures all philosophers that when they try to pick up their forks, they will be free!
- Effectively the chef enforces a fair "fork discipline" over the philosophers
- This is the most efficient solution (no shared resources/locking involved) but is in practice the hardest to achieve (the chef must know how to instruct the philosophers to eat in a fair, interference-free fashion).
- **For example**, the chef can assign a number to each philosopher and decide that the following pairs of philosophers eat at the following order:
 $(3, 5) \rightarrow (1, 4) \rightarrow (2, 4) \rightarrow (1, 3) \rightarrow (5, 2)$
- This schedule ensures that each philosopher gets to eat twice in each round and will neither deadlock nor starve



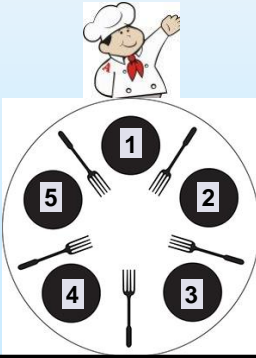
Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

Dining Philosophers Solution 3 Implementation

- A Python program model for the dining philosophers is coded in the file:

`PARALLEL_PROGRAMMING_LAB/DEADLOCK/dining_philosophers.py`

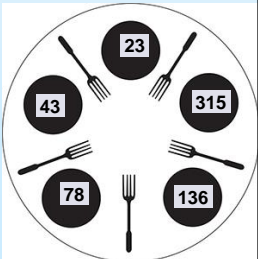
- Based on this code, try to implement a Chef Thread which monitors the 5 philosophers and solves the problem as described above
- How to go about solution 2 ?



Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>

Dining Philosophers: Solution 4

- Each philosopher behaves as usual. That is whenever it gets hungry, he is trying to acquire the two forks as usual (in whatever order he wants)
- Each Philosopher is assigned a **"Hunger Index"**
- This is roughly the time that has passed since he last ate
- As soon as the highest Hunger Index rises above a fixed threshold, the neighbors of this philosopher must release the forks near the starving philosopher (or complete their food if they were eating and then release the forks)
- This guarantees that the starving philosopher will get to eat in a short time.
- Once the starving philosopher is satiated, his "Hunger Index" drops down below the next starving philosopher
- How would you implement this solution?
Start with the file:
PARALLEL_PROGRAMMING_LAB/DEADLOCK/dining_philosophers.py



Based on David Beazley Python Concurrency Paper: <http://www.dabeaz.com>