

Part 4

TRANSPORT LAYER

SOCKET PROGRAMMING

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)

Transport Layer

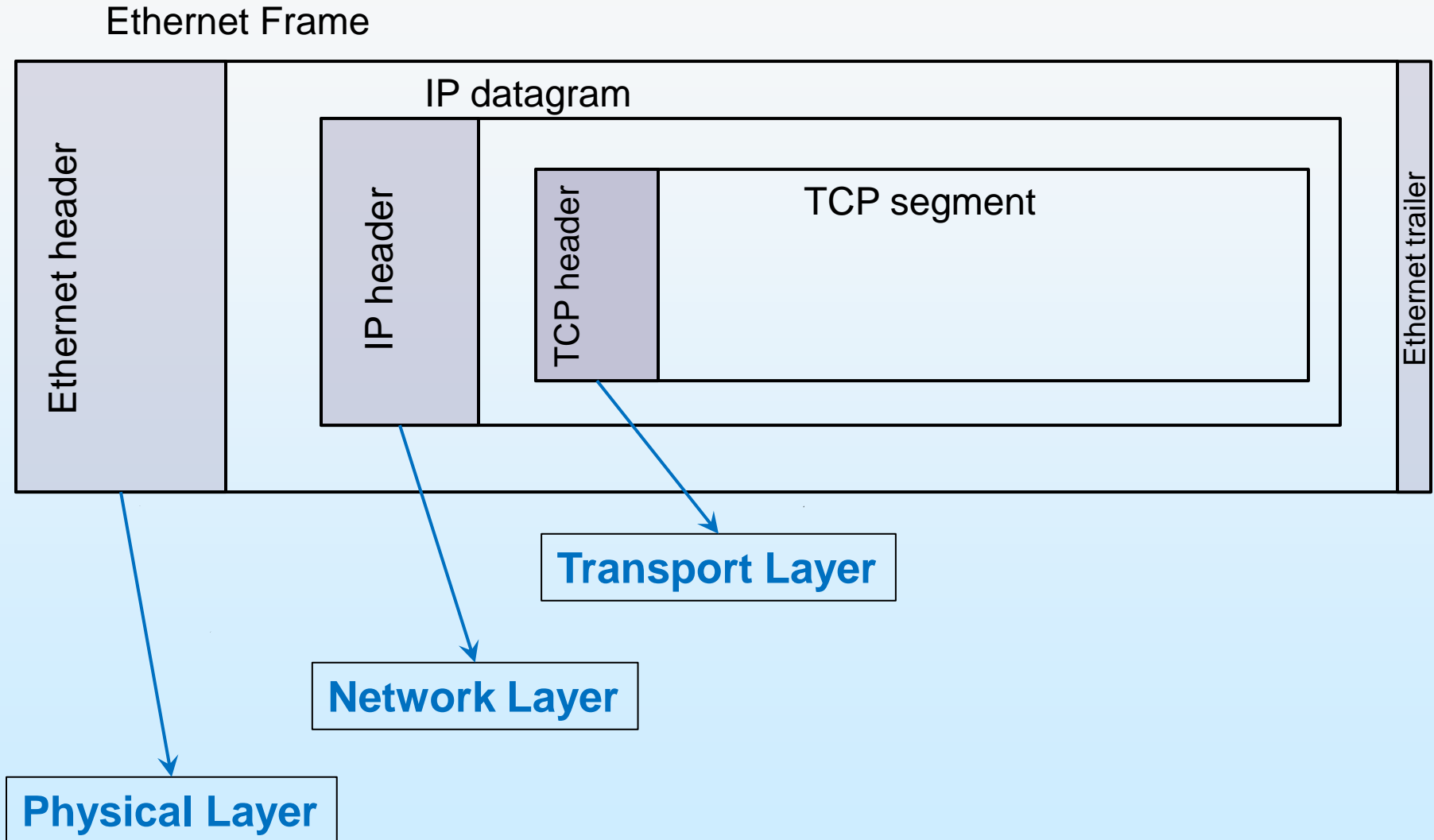
- The transport entity code runs entirely on users machines, but the network layer mostly runs on routers, cards, and other bridging hardware
- Bridging hardware is inherently unreliable and uncontrollable
 - ◆ Ethernet cards, routers, and similar hardware do not contain adequate software for detecting 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)

Packet Hierarchy



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”

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
```

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()
```

The Simplest Client/Server App

SERVER

```
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()
```

CLIENT

```
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()
```

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/fcntl.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]; /* buffer for outgoing file */
    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 */
    }
}
```

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);
```

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!

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!

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 messages 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

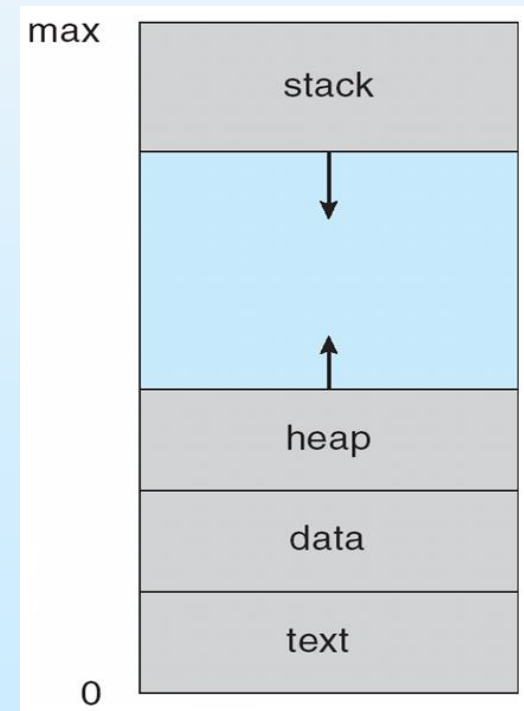
Project 4: BFTP

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)

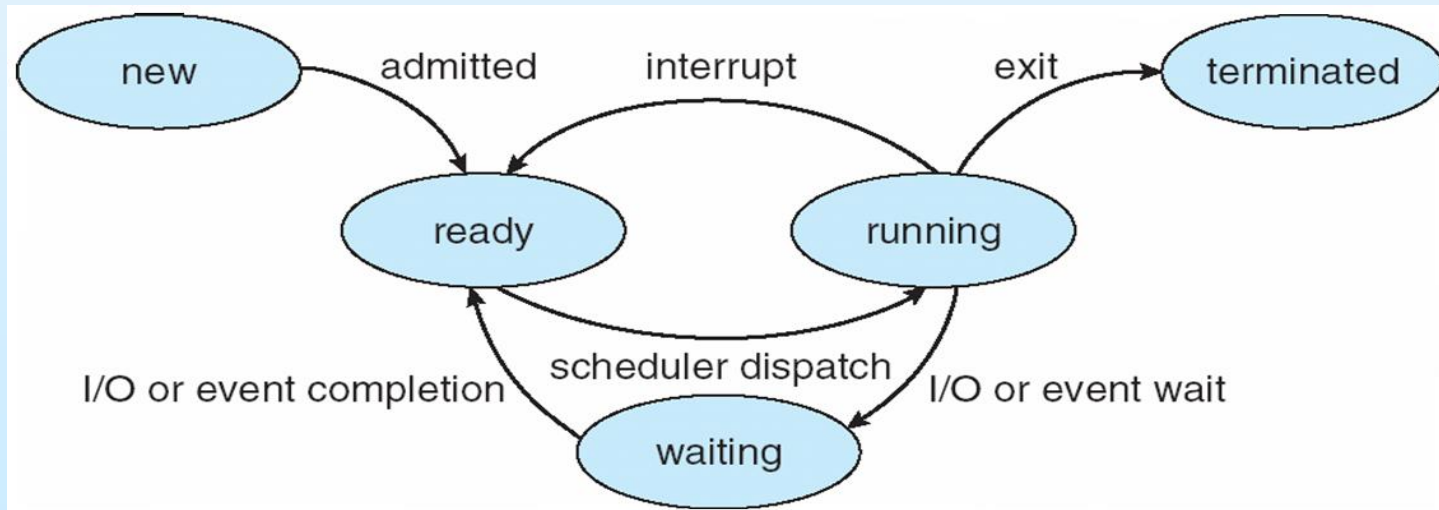
Process and Threads Concepts

- 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!

Process Parts

TEXT (PROGRAM CODE)			
DATA			
HEAP (Dynamic Memory)			
OPEN FILES TABLE			
THREAD 1	THREAD 2	THREAD 3	THREAD 4
Registers	Registers	Registers	Registers
Program Counter	Program Counter	Program Counter	Program Counter
Stack	Stack	Stack	Stack
			

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()
```