

CMPUT 379 Lab

ETLC E1003: Tuesday, 5:00 – 7:50 PM.

Tianyu Zhang, Peiran Yao

CAB 311: Thursday, 2:00 – 4:50 PM.

Max Ellis, Aidan Bush

Last Week...

- Handling Processes
- Sending and handling signals
- Got familiar with Interprocess Communication (IPC)
- Information Sharing
- More in depth piping

Today's Lab

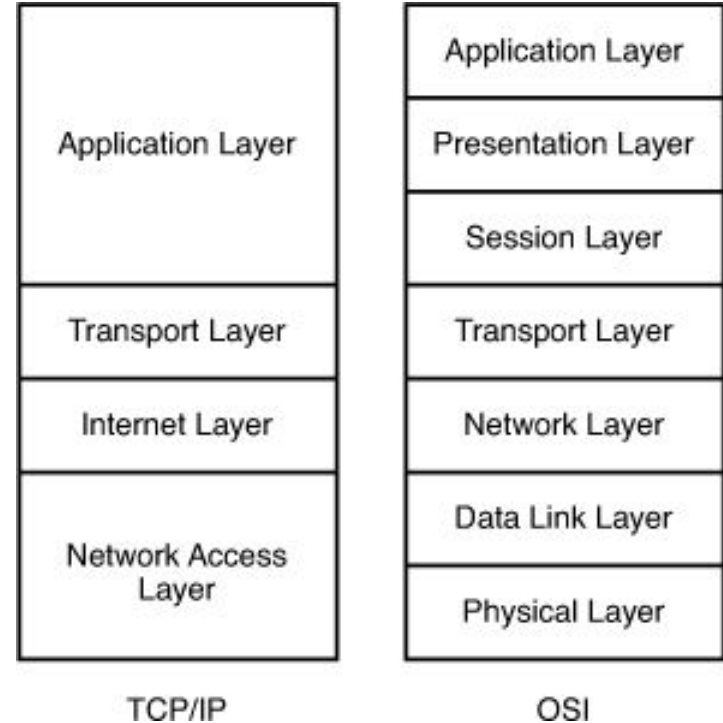
- Introduction to networking
- Creating and using sockets
- Getting familiar with TCP and UDP
- A brief look at pthreads
- Race conditions and deadlocks

Networking Models

Network protocols are built on many layers to allow for understanding and separation of tasks

We will cover the two (main) **transport layer protocols** TCP and UDP

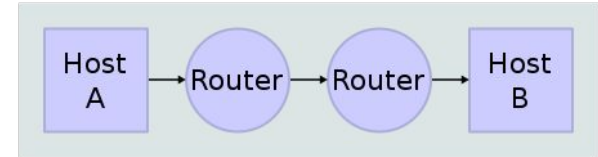
Transport layer relies on network layer and serves the application layer



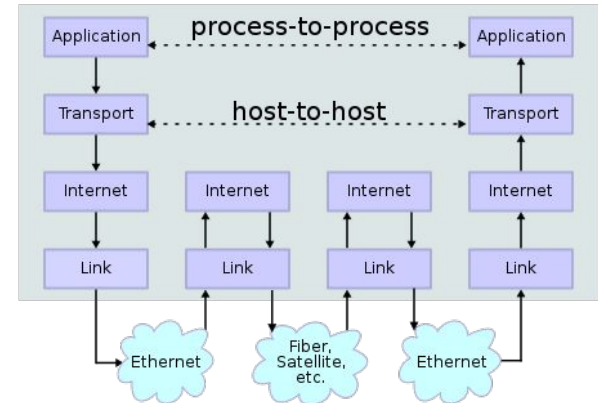
Networking

Allows computers to communicate

Network Topology



Data Flow



Ports

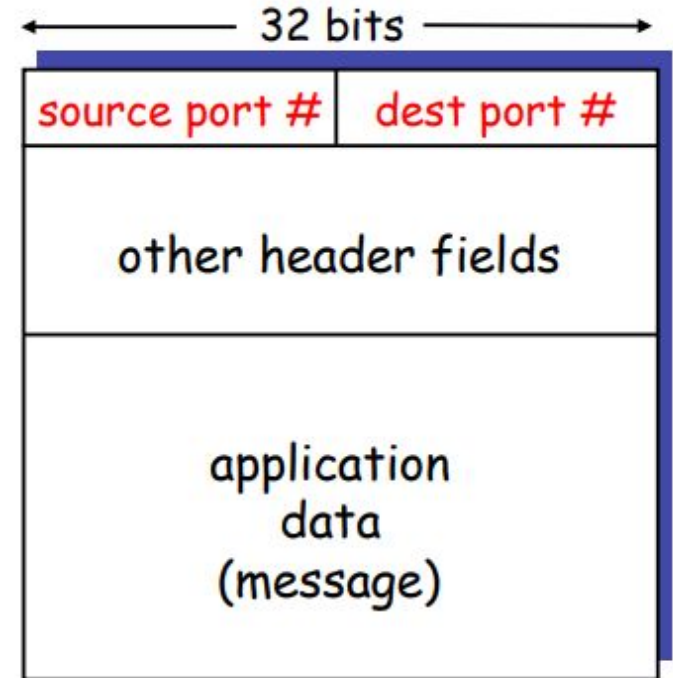
- Need to decide which application gets which packets
 - Solution: use port ports to specify the destination application
- Client and servers must know eachothers ports
- Separate 16-bit port address space for UDP and TCP
 - (src IP, src port, dst IP, dst port) uniquely identifies TCP connection

Port numbers

- Well-known ports (0-1023): everyone agrees which services run on these ports
 - e.g., ssh:22, http:80
 - on UNIX, must be root to gain access to these ports (why?)
- Ephemeral ports(most 1024-65535)

Multiplexing and Demultiplexing

- Sharing of network resources
- To share resources in packet switching networks, IP addresses and ports are used to direct packets



Transport-layer protocols

- Protocols:
 - User Datagram Protocol (UDP)
 - Transmission Control Protocol (TCP)
 - Most internet traffic is TCP
- Transport-layer principles:
 - Share network resources
 - Error detection (TCP only)
 - Reliable delivery (TCP only)
 - Flow control (TCP only)

Transport-layer protocols

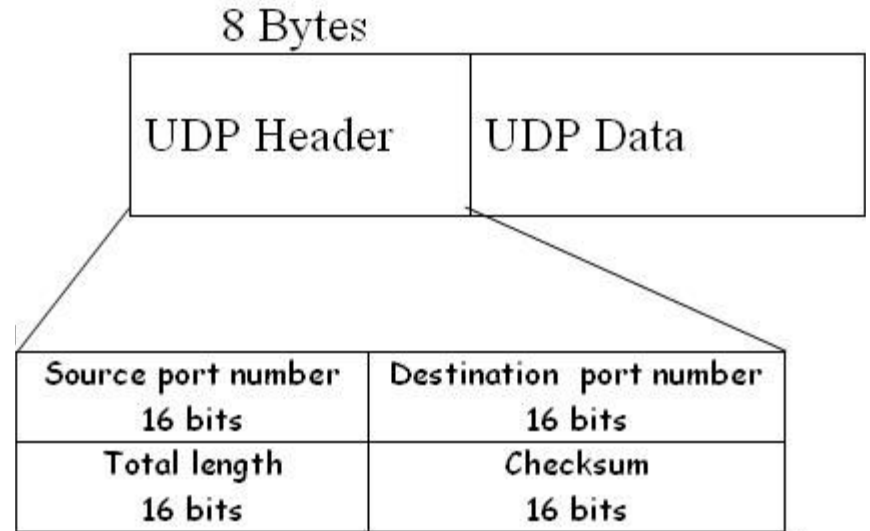
- Endpoints:
 - **Sender:** breaks application messages into packets, and passes to network layer
 - **Receiver:** reassembles packets into messages, passes to application layer

Transport-layer protocols

- UDP - Datagram *messaging* service:
 - Simply sends packets as they are created
- TCP - Reliable, *streaming*, in-order delivery:
 - Connection set-up
 - Discarding of corrupted packets
 - Retransmission of lost packets
 - Flow control
 - Congestion control

User Datagram Protocol (UDP)

- Lightweight
- Avoids extra overhead and delays caused by ordered reliable delivery



User Datagram Protocol (UDP)

- Fast - send data immediately
- Low delay
- Small packet header (8 bytes)
- No connection setup
- Unreliable
 - No error checking
 - Does not ensure packets arrive at the destination
- Good for real-time communications

UDP Examples

- Live multimedia streaming
 - Retransmitting lost/corrupted packets is not worthwhile
 - By the time the packet is retransmitted, it's too late
- Domain Name System
 - Overhead of connection establishment is overkill
 - Easier to have application retransmit if needed

Transmission Control Protocol (TCP)

- Connection oriented
 - Explicit set-up and tear-down of TCP session
- Streaming service
 - Sends and receives a stream of bytes, not messages
 - Behaves like file I/O

Transmission Control Protocol (TCP)

- Reliable, in-order delivery
 - Detects corrupted data (checksum)
 - Acknowledgments & retransmissions
- Flow control
 - Prevents overflow of the receiver's buffer space
- Congestion control
 - Prevents overflow and fair use of the network resources

Transmission Control Protocol (TCP)

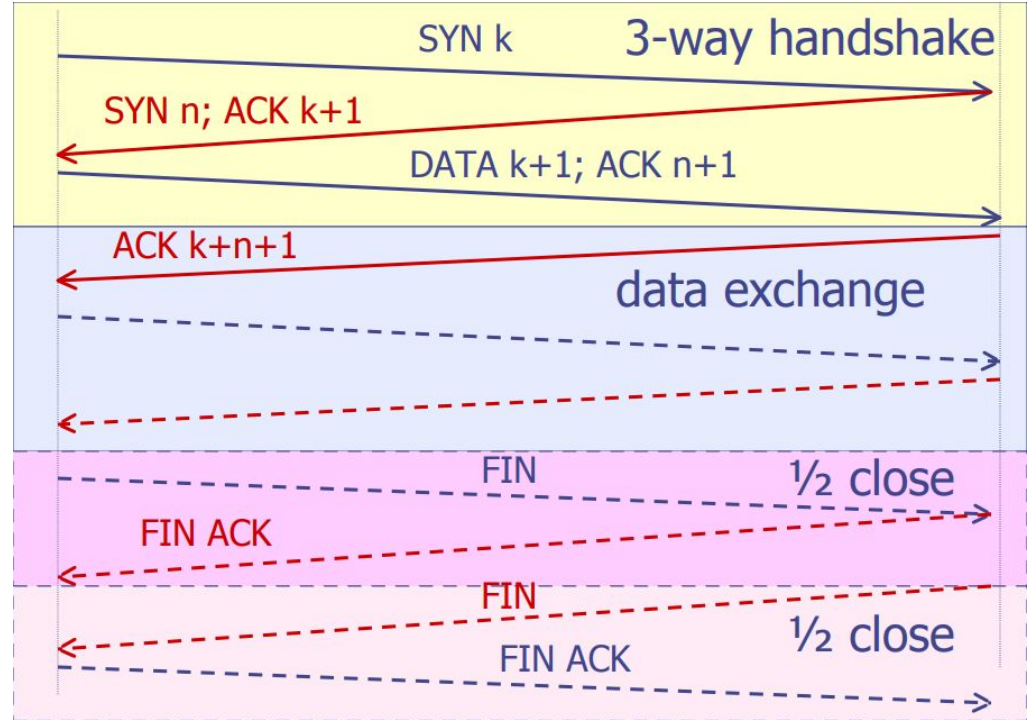
- Similar to phone calls in the real world
- What would you do if you could not hear clearly?
- What would you do if you do not get response in a while?

TCP Reliability

- Receiver
 - Sends ACKs to confirm arrival
 - May send a NACK to ask for a re-transmission (generally not used)
- Re-transmission by sender:
 - After receiving “NACK”
 - After a timeout

TCP Lifecycle

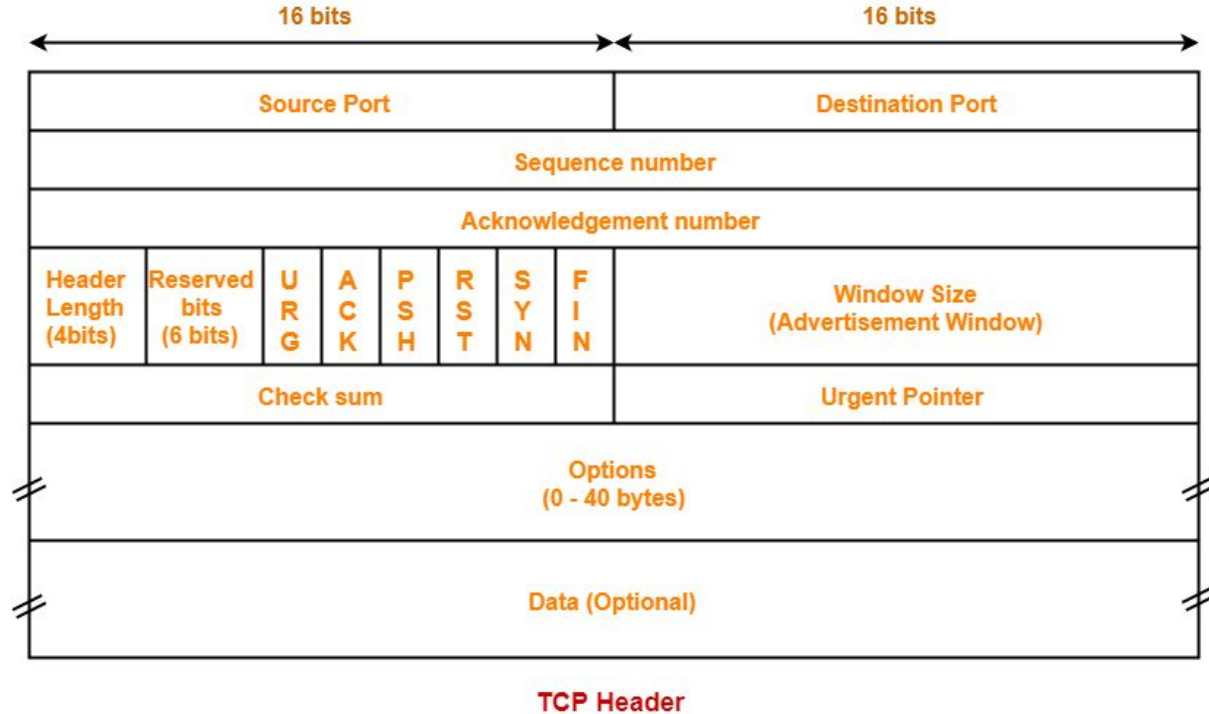
- SYN
 - establishing TCP connection
- FIN
 - terminating a TCP connection
- ACK
 - Acknowledgement



TCP Transmission Management

- TCP uses a window of packets
 - Used to ensure flow and congestion control
- If room in the window packets are sent when:
 - Segment full (Max Segment Size)
 - Not full, but times out
 - “Pushed” by application

TCP Packet structure



UDP and TCP comparison

UDP

- Minimal
- Connectionless
- Datagram oriented
- Not ordered
- No error detection / handling
 - Bit or lost packets

TCP

- Reliable
- Connection oriented
- Stream oriented
- Ordered
- Error Checked

Sockets

- Interfaces to protocols such as TCP or UDP
- Endpoints of connections
- Similar to pipes

Creating and Setting up Sockets

getaddrinfo() - Help setup socket arguments

socket() - Create socket

setsockopt() - Modify / configure socket

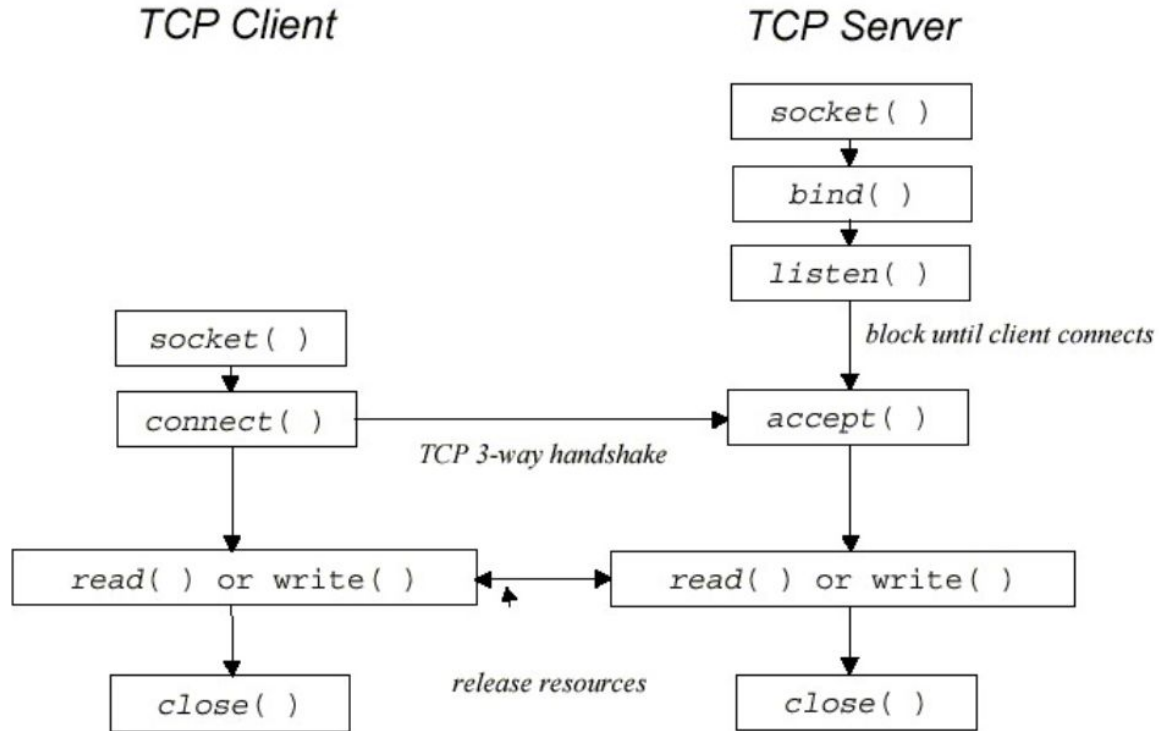
bind() - Bind to a port

listen() - (TCP only) Start listening for new connections

connect() - (TCP only) Connect to server

freeaddrinfo() - Clean up addr info

Using TCP Sockets



Simple TCP server

```
int main() {
    int listenfd, connfd;
    struct sockaddr_in cliaddr, servaddr; bzero(&servaddr, sizeof(servaddr));
    socklen_t clilen;
    listenfd = socket(AF_INET, SOCK_STREAM, 0); /* create a listen socket */
    servaddr.sin_family = AF_INET; /* use IPv4 protocol */
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* accept any incoming address */
    servaddr.sin_port = htons(SERV_PORT); /* set server port */
    bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)); /* bind address */
    listen(listenfd, 10); /* start listening */
    while (1) {
        connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &clilen); /* accept incoming conn */
        echo(connfd);
        close(connfd); /* close connection */
    }
}
```

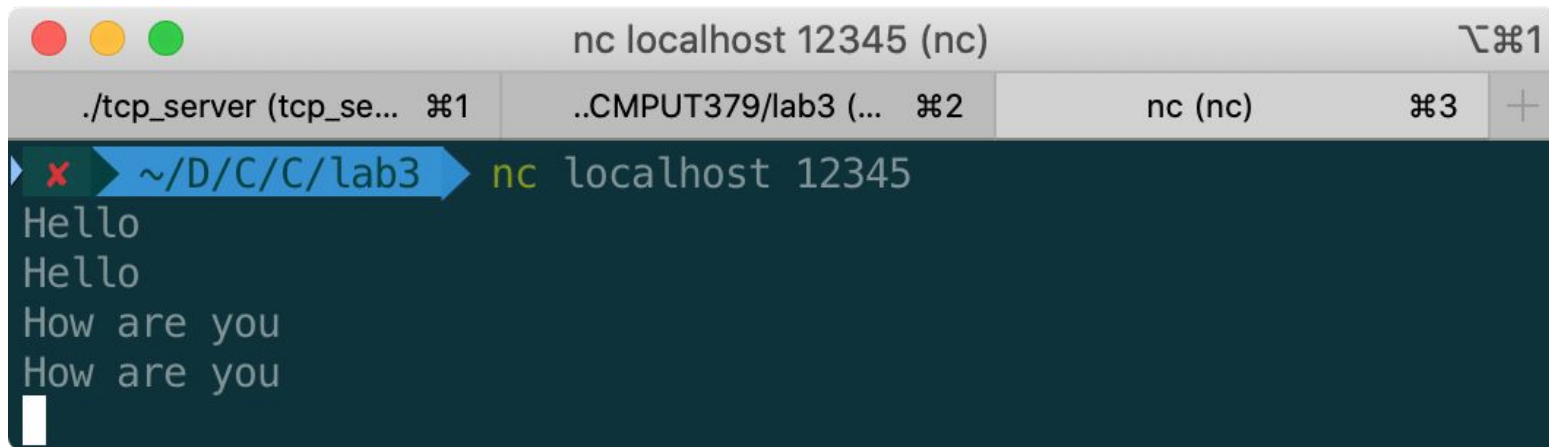
Simple TCP server

```
void echo(int sockfd) {  
    /* read content into a buffer, and write them back */  
    ssize_t n;  
    char buf[MAXLINE];  
    /* sockets can also use read(), write() and close() */  
    while ((n = read(sockfd, buf, MAXLINE)) > 0) {  
        write(sockfd, buf, n);  
        if (n < 0) perror("write");  
    }  
    return;  
}
```

TCP Server and Client Demo

Test the server

Netcat (nc) is a versatile utility for working with TCP or UDP data.
More information: <<https://nmap.org/ncat>>.



```
nc localhost 12345 (nc)
./tcp_server (tcp_se... #1
..CMPUT379/lab3 (... #2
nc (nc) #3
~ /D/C/C/lab3 nc localhost 12345
Hello
Hello
How are you
How are you
█
```

However, only one client can connect to the server at the same time

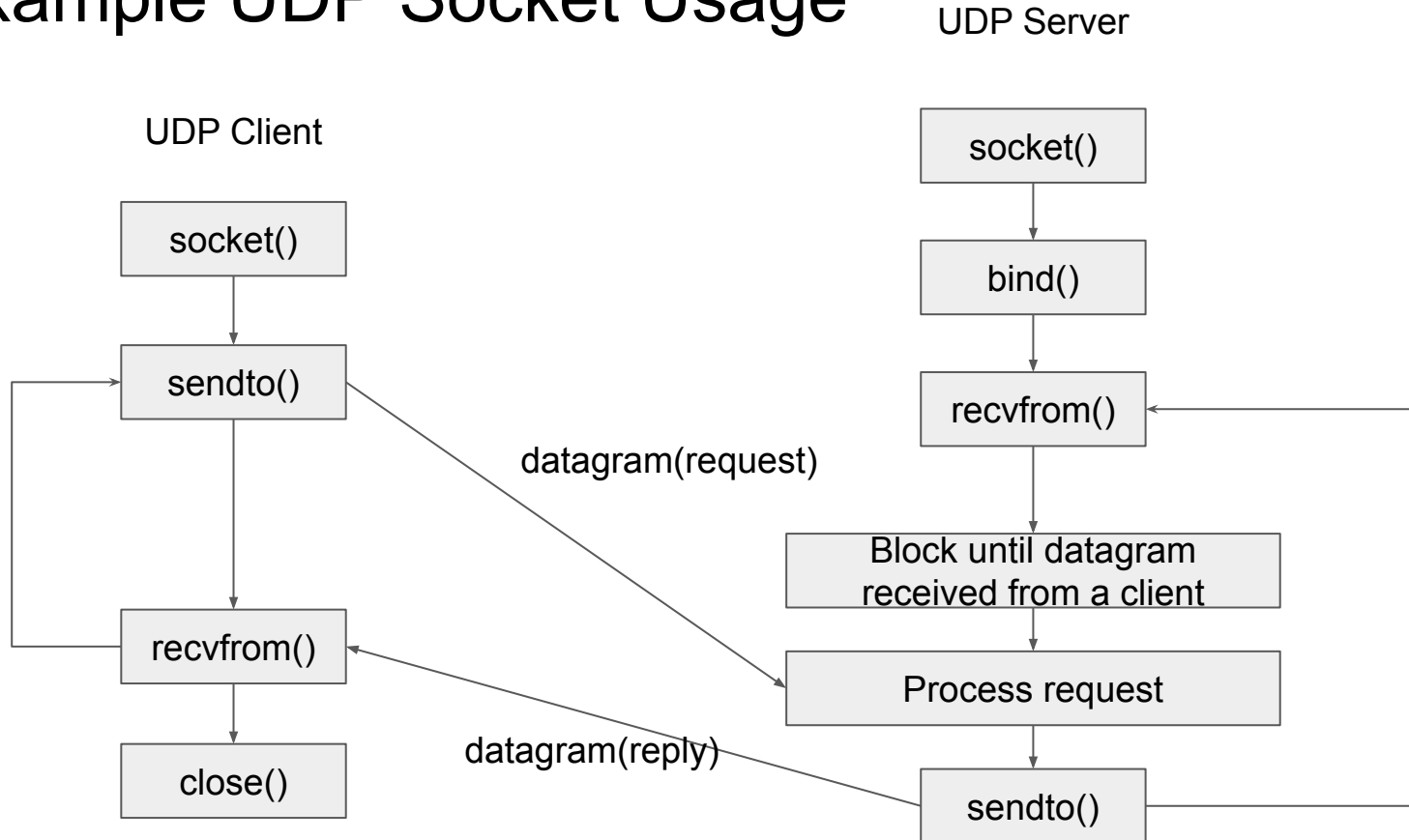
Concurrent TCP server using processes

```
if (listen(listenfd, 10) < 0) perror("listen");
while (1) {
    connfd = accept(listenfd, (struct sockaddr*) &cliaddr, &clilen);
    /* start a new child process for each connection*/
    if ((childpid = fork()) == 0) {
        close(listenfd); /* close listen socket in child */
        echo(connfd); /* handle connection in child */
        close(connfd);
        exit(0);
    }
    close(connfd);
}
```

TCP server - Caveats

- TCP is stream-oriented. There are no explicit boundaries between “messages”
 - If you call **write()** twice, the data might arrive in one **read()**
- **read()** and **write()** might return **BEFORE** the given number of bytes are received or sent. Remember to resume in these cases

Example UDP Socket Usage



UDP server

```
int main(int argc, char **argv) {  
    int sockfd;  
    struct sockaddr_in servaddr, cliaddr;  
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);           // use SOCK_DGRAM  
    bzero(&servaddr, sizeof(servaddr));  
    servaddr.sin_family = AF_INET;  
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);  
    servaddr.sin_port = htons(SERV_PORT);  
    bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));  
    echo(sockfd, (struct sockaddr *) &cliaddr, sizeof(cliaddr));  
}
```

UDP server

```
void echo(int sockfd, struct sockaddr *pcliaddr, socklen_t clilen) {  
    int n;  
    socklen_t len;  
    char mesg[MAXLINE];  
    while (1) {  
        len = clilen;  
        n = recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);  
        sendto(sockfd, mesg, n, 0, pcliaddr, len);  
    }  
}
```

UDP server

Key differences:

- Socket type is **SOCK_DGRAM** instead of SOCK_STREAM
- Use **recvfrom()** and **sendto()** instead of **read()** and **write()**
- Receives and send a single and complete message at a time

UDP server - Caveats

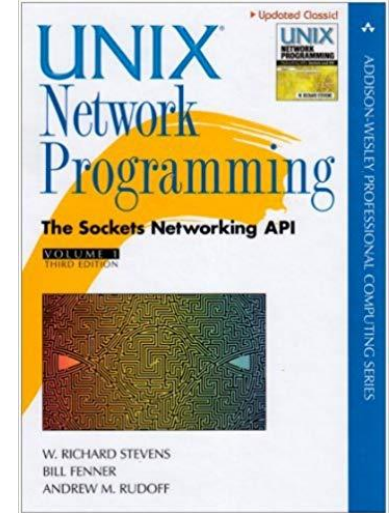
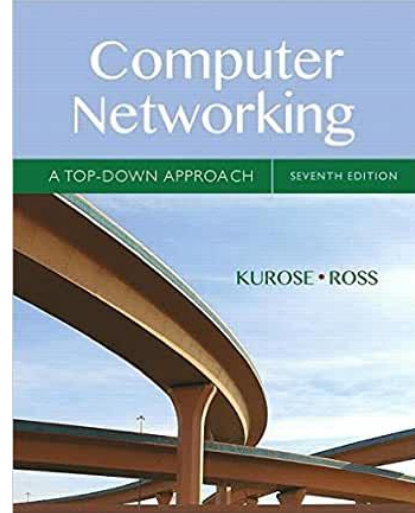
- Datagrams sent via UDP may be **lost**, **duplicated**, or **corrupted** during transit

Unix Domain Sockets

- Similar to internet sockets, but for local processes
- Can behave like a TCP or UDP socket
- Useful for daemons
- Example: X11, docker, systemd

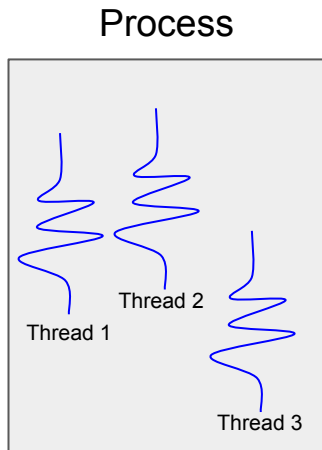
Learn more

- Computer Networking: A Top-Down Approach (7th Edition). J. Kurose, K. Ross. Pearson.
- Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition). W. Richard Stevens, B. Fenner, A. Rudoff. Addison-Wesley.



Concurrency through threads

- Threads: A sequential execution stream within a process
 - The process still contains a single address space
 - Threads share the same memory
 - The process is just a container for threads!



Concurrency through processes

- Why not just use multiple processes for each concurrent task?
- Process creation is expensive...
 - Trap a system call
 - Allocate memory
 - Setup virtual memory
 - Copy data and I/O state from parent

pthread

- POSIX Thread
- **#include <pthread.h>**
- Compiling pthreads: **gcc -Wall -o helloworld helloworld.c -lpthread**
- With make use **LDFLAGS=-lpthread**

pthread

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start_routine) (void *), void *arg);
```

Start a thread from **start_routine**, and pass the argument **arg**

- **thread** - pointer to thread id which will be set
- **attr** - set to 0 if default thread attributes are used
- **function** - pointer to the function to be threaded
 - It has a single argument, which points to data we want the thread to process
- **arg** - pointer to argument of function
 - Multiple arguments can be used by passing the address of a struct or class object

pthread

```
int pthread_join(pthread_t th, void **retval);
```

Wait for a thread to exit, and store the return value at **retval**

- **th** - The current thread is suspended until the thread identified by th terminates
- **retval** - If retval is not 0, the return value of th is stored in the location pointed to by retval

pthread

```
void pthread_exit(void *retval);
```

Exit the current thread, and return **retval**.

- **retval** - Return value of thread

pthread: basic usage

```
void* thread_function(void* args_p /* the input */) {
    int* num = (int *) args_p; /* cast the input to the type you want */
    printf("I'm in a new thread. I have number %d\n", *num);
    return NULL;
}

int main() {
    pthread_t thread_handle;
    int num_to_pass = 42;
    pthread_create(&thread_handle, NULL, thread_function, &num_to_pass);

    sleep(1);
    printf("I'm in the main thread!\n");
    pthread_join(thread_handle, NULL); /* wait for the thread to finish */
    return 0;
}
```

pthread: Another Example

```
#include <stdio>
#include <stdlib>
#include <pthread.h>
void *thread_func(void *ptr) {
    const char *msg = (char *)ptr;
    printf("%s\n", msg);
    return ptr; // return pointer to thread result, can't be pointing to local variable
}
int main() {
    pthread_t t1, t2;
    const char *msg1 = "I am Thread 1";
    const char *msg2 = "I am Thread 2";
    void *ret1, *ret2;
    pthread_create(&t1, nullptr, thread_func, (void*)msg1); // Create independent threads each of which will
    pthread_create(&t2, nullptr, thread_func, (void*)msg2); // execute thread_func

    pthread_join(t1, &ret1);
    pthread_join(t2, &ret2);
    printf("Thread 1 returns: %s\n", (char*)ret1);
    printf("Thread 2 returns: %s\n", (char*)ret2);
    return 0;
}
```

// Wait till threads are complete before main
// continues. Unless we wait we run the risk of
// executing an exit which will terminate the
// have completed. Thread results are stored in
// ret1 and ret2.

FYI: concurrency techniques for networking

- Pre-fork
 - fork multiple child processes right after your server set up
 - Assign incoming connections to the pre-forked child processes
- Thread pool
 - Similar to pre-fork, but use threads instead of child processes
- Event-based

Race Conditions

- Occurs when two concurrent threads access a shared resource without synchronization
- The end result depends on:
 - Timing
 - Which context switch occurs
 - Which thread ran on a context switch
 - What the threads were doing at the time

Too much milk

Consider that Peter and Greg are roommates

Time	Peter	Greg
3:00	Look in fridge, no milk	
3:05	Leave for store	
3:10	Arrive at store	Look in fridge, no milk
3:15	Buy milk	Leave for store
3:20	Arrive at home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive at home, put milk away (oof)

Race Condition - Example

```
int x = 1; // global var in shared space
int print_message_function(void *arg) {
    char *message = (char *) arg;
    printf("%s ", message);
    return ++x;
}
int main(int argc, char** argv) {
    pthread_t thread1, thread2;
    void char char *thread1result, *thread2result; // will hold results
    *message1 = "Hello";
    *message2 = "World";
    printf("Begin threads\n");
    pthread_create(&thread1, NULL, (void*) print_message_function, message1);
    pthread_create(&thread2, NULL, (void*) print_message_function, message2);
    printf("Main Thread!\n"); //when does this print?
    pthread_join(thread1, &thread1result);
    pthread_join(thread2, &thread2result);
    printf("Thread 1 returns %d\n", thread1result);
    printf("Thread 2 returns %d\n", thread2result);
}
```

Begin threads
Main Thread!
Hello World Thread 1 returns 2
Thread 2 returns 3

Begin threads
Main Thread!
World Hello Thread 1 returns 2
Thread 2 returns 3

Begin threads
Hello World Main Thread!
Thread 1 returns 3
Thread 2 returns 2

Begin threads
Main Thread!
World Hello Thread 1 returns 3
Thread 2 returns 2

Thread Synchronization

The threads library provides three synchronization mechanisms:

- **Mutexes** — Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables
- **Joins** — Make a thread wait till others are complete (terminated)
- **Condition Variables** — Used for waiting until signalled to continue by another thread

Thread Synchronization - Mutexes

Mutexes are used to prevent data inconsistencies due to operations by multiple threads upon the same memory area performed at the same time or to execute operations in a certain order.

```
// without mutex
int counter = 0;
void increment(){ // 1. load counter into register
    counter++;    // 2. increment register
}                // 3. store register to counter
```

```
// with mutex guarding the critical section
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;
void increment() {
    pthread_mutex_lock(&counter_mutex);
    counter++;
    pthread_mutex_unlock(&counter_mutex);
}
```

Thread Synchronization - Condition Variables

Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

```
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t *attr);
```

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Thread Synchronization - Condition Variables

- **pthread_cond_wait()** blocks the calling thread until the specified condition is signalled. This function should be called while mutex is locked, and it will automatically release mutex while it waits.
- **pthread_cond_signal()** function is used to signal (or wake up) another thread which is waiting on the condition variable. It must be called after mutex is locked.
- **pthread_cond_broadcast()** function should be used instead of pthread_cond_signal() if more than one thread is in a blocking wait state. All waiting threads will be woken up.

Example code available on eClass later on.

Deadlock

- A **deadlock** is a situation which occurs when a process or thread enters a waiting state because a resource requested by it is being held by another waiting process, which in turn is waiting for another resource. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process, then the system is said to be in a deadlock.
- Suppose I'm a new graduate, and I need a job, but all work requires experience. I can't get the job without having the (professional) experience and I can't get the experience without having a job.

Deadlock example

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
void *simple_thread(void *) {
    pthread_mutex_lock(&mutex2);    // lock mutex2
    pthread_mutex_lock(&mutex1);    // lock mutex1
    pthread_mutex_unlock(&mutex1);  // unlock mutex1
    pthread_mutex_unlock(&mutex2);  // unlock mutex2
    return 0;
}
int main() {
    pthread_t tid;
    pthread_create(&tid, nullptr, &simple_thread, nullptr);
    pthread_mutex_lock(&mutex1);    // lock mutex1
    pthread_mutex_lock(&mutex2);    // lock mutex2
    pthread_mutex_unlock(&mutex2);  // unlock mutex2
    pthread_mutex_unlock(&mutex1);  // unlock mutex1
    pthread_join(tid, nullptr);
}
```


Livelock

- A **livelock** is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.
- Suppose Alice and Bob meet in a corridor, and both of them decide to avoid other. They end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Deadlock & Livelock

The **valgrind** tool helgrind can help identify the lines of code which may potentially cause deadlocks

```
valgrind --tool=helgrind a.out
```

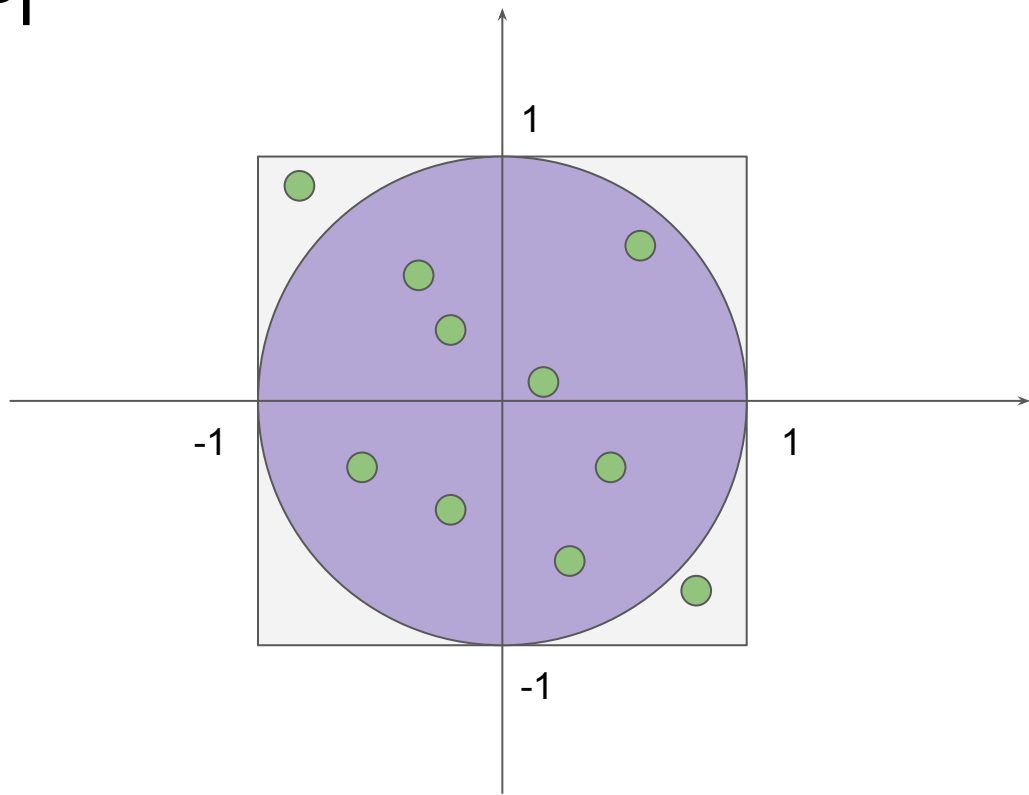
Your program should be compiled with **-g** to give valgrind access to source code information.

Learn more

- An Introduction to Parallel Programming by Peter Pacheco.
- Operating Systems: Three Easy Pieces. Chapter 25 - 34.

Practice 1: estimating Pi

- If you randomly pick up a point with the square, it will have a probability of $\pi/4$ to fall inside the circle
- You can simulate the process of picking a point by generating random numbers (Monte Carlo simulation)
- Use multiple threads to speed up the process



Practice 1: hints

- Use **rand_r()** instead of **rand()**, and make the seed a local variable in your thread routine. (**rand()** has a global lock, and therefore can't be used in parallel)
- Use **time** to measure your speed up

Practice 1: draft answer

```
#define TOTAL 1000  
#define RAND(seed) (((double) rand_r(seed)) / RAND_MAX)*2 - 1)
```

```
void* thread_function(void* args_p) {  
    int* hit = malloc(sizeof(int));  
    unsigned seed = (unsigned) time(NULL);  
    double x, y;  
    for (int i = 0; i < TOTAL; i++) {  
        x = RAND(&seed); y = RAND(&seed);  
        printf("%f %f\n", x, y);  
        if (x*x + y*y <= 1) *hit += 1;  
    }  
    pthread_exit(hit);  
}
```

```
int main() {  
    pthread_t thread_handle;  
    pthread_create(&thread_handle, NULL, thread_function, NULL);  
  
    sleep(1);  
    printf("I'm in the main thread!\n");  
    int* h;  
    pthread_join(thread_handle, (void **) &h);  
    printf("hit %d\n", *h);  
    return 0;  
}
```

Practice 2: file server and downloader

- Write a TCP server and a client
- The server reads a filename from the socket, opens and reads the file, and sends the content of the file to the client
- The client takes an input filename from stdin, sends te filename to the server, reads the content of the file from the server and save the content to a local file.