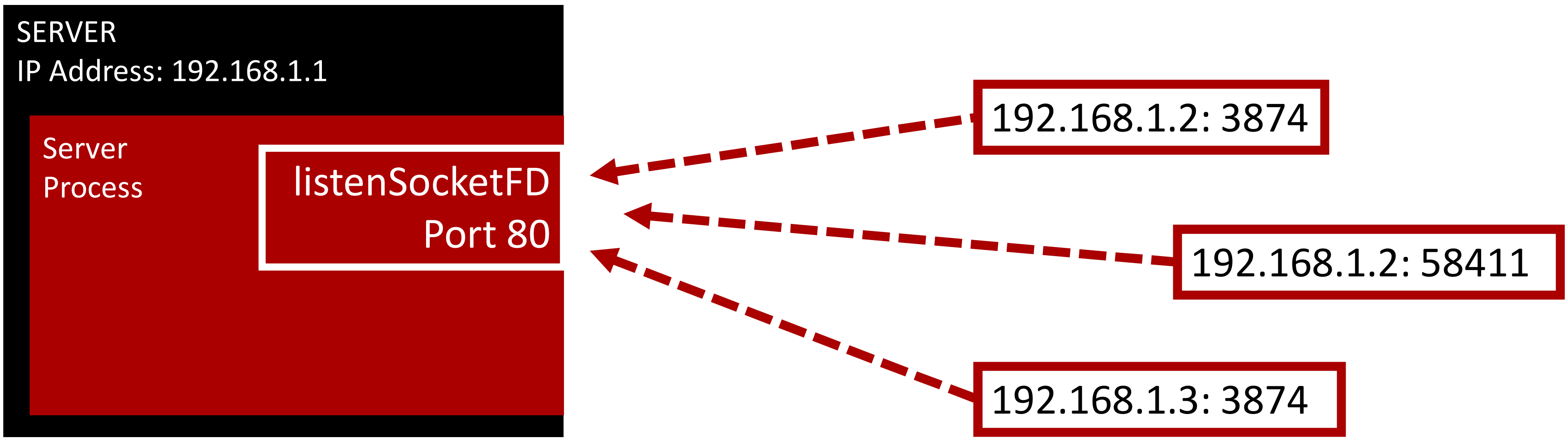
Network Servers

Benjamin Brewster

Network Servers

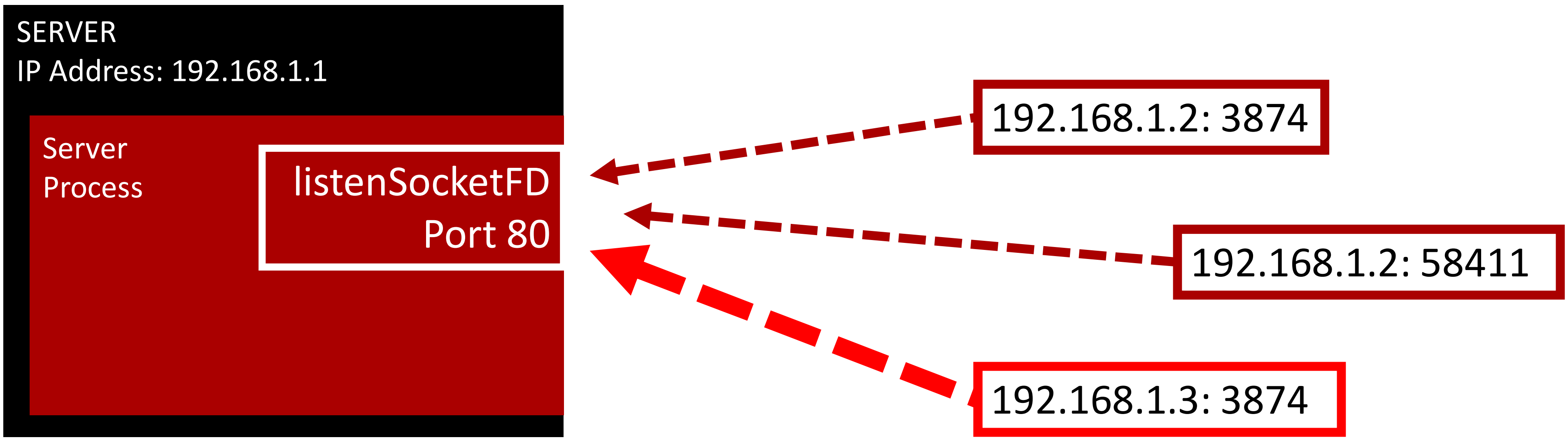
- This lecture covers
  - Setting up network sockets and connecting clients to them
  - Demo working server code
  - Server concurrency methodologies
  - Knowing when data is available

Non-Concurrent Server Connection Overview



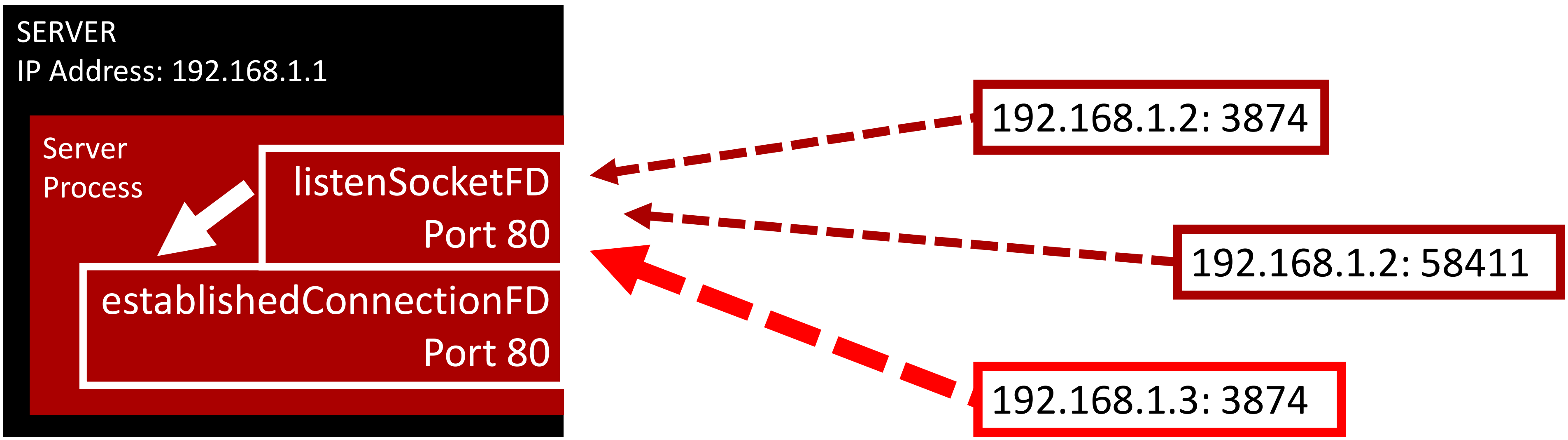
- A server socket listens on a specified port
- Many different clients may be connecting to that port
- The server needs to *differentiate* between and *communicate* with each client

Non-Concurrent Server Connection Overview



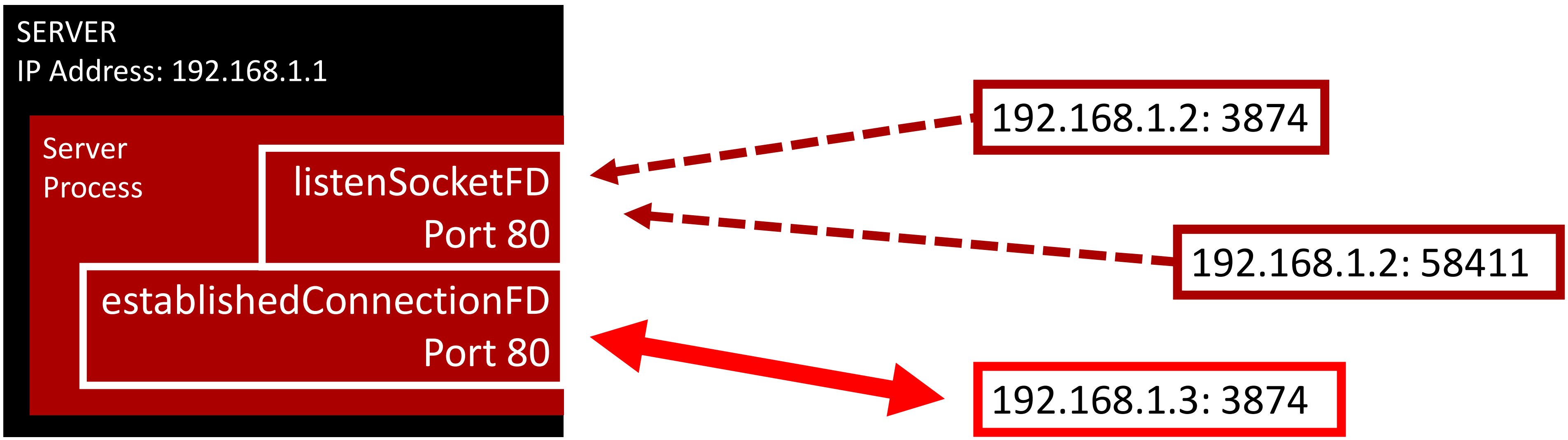
- Step 1: The server chooses the next connection to deal with

Non-Concurrent Server Connection Overview



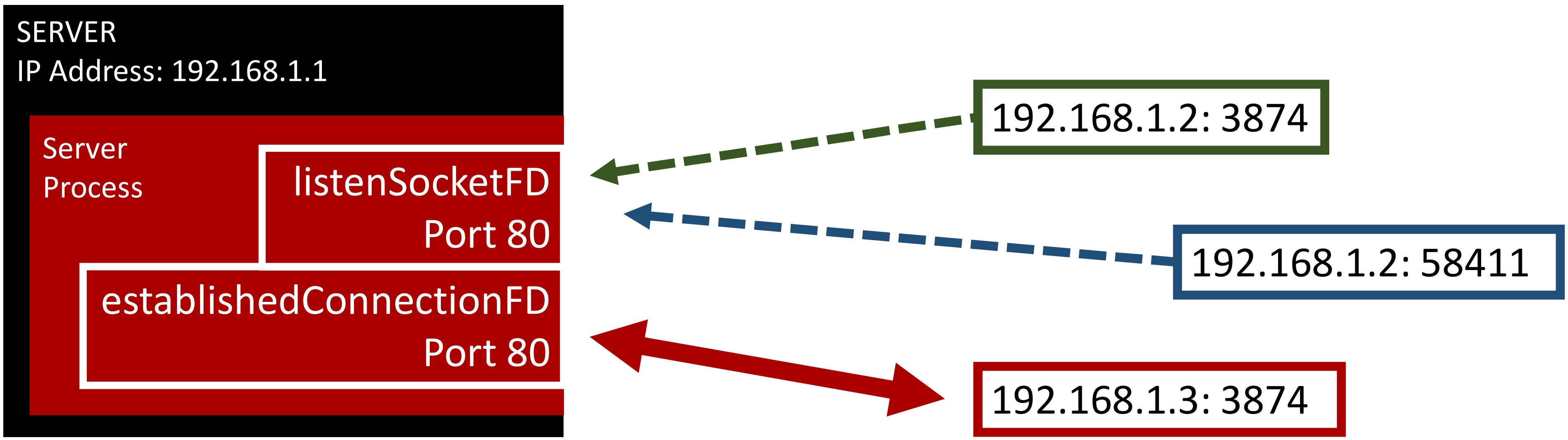
- Step 2: The server creates a new file descriptor for the chosen connection to exclusively use
- Note that the port doesn't change!

Non-Concurrent Server Connection Overview



- Step 3: The chosen client begins communication with the new file descriptor created by the server
- New connections can now be handled again by listenSocketFD

Connection Differentiation



- The OS network layer on the server *differentiates each connection* by using four pieces of the addresses, routing the network packets to the correct socket/FD based on:
  - Server IP, Server Port, Client IP, Client Port

Server Sockets API

- Server Procedure:
  1. Create a network socket/FD with `socket()`
  2. Bind the socket to a port number with `bind()`
  3. Start listening for connections on that socket/port with `listen()`
  4. Loop and accept connections on that socket/port with `accept()`, connecting them to *new* sockets for each connection's exclusive use
  5. Read and write data to and from the *newly* created sockets for connected and accepted clients using `send()` & `recv()` or `read()` & `write()`

Creating the Socket - Same Method as Client

```
int socket(int domain, int type, int protocol);
```

Returns file descriptor or -1

For general-purpose sockets that can connect across a network, use AF_INET

For sockets that are used ONLY for same-machine IPC, use AF_UNIX

For TCP, use SOCK_STREAM
For UDP, use SOCK_DGRAM

Use 0 for normal behavior

```
int socketFD = socket(AF_INET, SOCK_STREAM, 0);
if (socketFD < 0) {
     perror("Hull breach: socket()"); exit(1);
}
```

Filling the Address Struct for the Server

- Set the address struct so that it accepts connections from any IP address, or just one, and which specific port it will be available on:

```
struct sockaddr_in serverAddress;

serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(80);
serverAddress.sin_addr.s_addr = INADDR_ANY;
```

`htons()`: **host-to-network-short**

Converts from *host/PC* byte order (LSB) to *network* byte order (MSB)

PCs store bytes with smallest digit first, but networks expect largest digit first

Allows connections from any IP address

Bind the Socket to a Port

- Ports allow multiple processes running on a single machine to communicate across the network from only a single IP address
- A server process has to choose a port where clients can contact it on
- `bind()` associates the chosen port with a socket already created with the `socket()` command
- Subsequent calls to `bind()` using an already-bound socket will fail
- Even after the sockets are all closed, the OS does not immediately release the port; you'll need to wait many seconds for it to be available again for reuse

Binding the Socket

```
int bind(int sockfd, struct sockaddr *address, size_t add_len);
```

Returns 0 on success or -1 on error

The socket we're binding to the port

The network address struct, which identifies which port this socket will use

The size of the address struct

```
if (bind(listenSocketFD, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) < 0)
{
	perror("Hull breach: bind()"); exit(1);
}
```

Listening for Connections

- The server will ignore any connection attempts until you tell the socket to start listening for connections with listen()
- Once this has been done, the socket will begin queuing up connection requests until it reaches the connection queue limit

```
int listen(int sockfd, int queue_size);
```

Returns 0 on success or -1 on error

The socket we're enabling for connections

Maximum number of connections to queue

```
if (listen(sockfd, 5) < 0) {
        perror("Hull breach: listen()"); exit(1);
}
```

Loop and Accept

- Servers generally run continually, waiting for clients to contact them
- Thus a server has an "infinite loop" that continually processes connections from clients
- The `accept()` function takes the next connection off of the listen queue for a socket, or blocks the process until a connection request arrives

Accepting Connections

```
int accept(int sockfd, struct sockaddr* address, size_t &add_len);
```

Returns file descriptor for new connection or -1 on error

The socket we're going to get a connection from

Network address struct, into which connecting *client* information will be written

The size of the address struct

```
int establishedConnectionFD = accept(listenSocketFD,
                                     (struct sockaddr*)&clientAddress,
                                     &sizeOfClientInfo);

if (establishedConnectionFD < 0) { perror("Hull breach: accept()"); exit(1); }
```

server.c 1 of 2

This is a basic server program that can send and receive.

It is intended to pair with client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(const char *msg) { perror(msg); exit(1); } // Error function used for reporting issues

int main(int argc, char *argv[])
{
	int listenSocketFD, establishedConnectionFD, portNumber, charsRead;
	socklen_t sizeOfClientInfo;
	char buffer[256];
	struct sockaddr_in serverAddress, clientAddress;

	if (argc < 2) { fprintf(stderr,"USAGE: %s port\n", argv[0]); exit(1); } // Check usage & args

	// Set up the address struct for this process (the server)
	memset((char *)&serverAddress, '\0', sizeof(serverAddress)); // Clear out the address struct
	portNumber = atoi(argv[1]); // Get the port number, convert to an integer from a string
	serverAddress.sin_family = AF_INET; // Create a network-capable socket
	serverAddress.sin_port = htons(portNumber); // Store the port number
	serverAddress.sin_addr.s_addr = INADDR_ANY; // Any address is allowed for connection to this process

	// Set up the socket
	listenSocketFD = socket(AF_INET, SOCK_STREAM, 0); // Create the socket
	if (listenSocketFD < 0) error("ERROR opening socket");
```

server.c 2 of 2

```
    // Enable the socket to begin listening
    if (bind(listenSocketFD, (struct sockaddr *)&serverAddress, sizeof(serverAddress)) < 0) // Connect socket to port
            error("ERROR on binding");
    listen(listenSocketFD, 5); // Flip the socket on - it can now receive up to 5 connections

    // Accept a connection, blocking if one is not available until one connects
    sizeOfClientInfo = sizeof(clientAddress); // Get the size of the address for the client that will connect
    establishedConnectionFD = accept(listenSocketFD, (struct sockaddr *)&clientAddress, &sizeOfClientInfo); // Accept
    if (establishedConnectionFD < 0) error("ERROR on accept");

    // Get the message from the client and display it
    memset(buffer, '\0', 256);
    charsRead = recv(establishedConnectionFD, buffer, 255, 0); // Read the client's message from the socket
    if (charsRead < 0) error("ERROR reading from socket");
    printf("SERVER: I received this from the client: \"%s\"\n", buffer);

    // Send a Success message back to the client
    charsRead = send(establishedConnectionFD, "I am the server, and I got your message", 39, 0); // Send success back
    if (charsRead < 0) error("ERROR writing to socket");
    close(establishedConnectionFD); // Close the existing socket which is connected to the client

    close(listenSocketFD); // Close the listening socket
    return 0;
}
```

Client/Server Results

```
$ gcc -o client client.c
$ gcc -o server server.c
$ ./server 51717 &
[1] 21094
$ ./client localhost 51717
CLIENT: Enter text to send to the server, and then hit enter: AWESOMESAUCE
SERVER: I received this from the client: "AWESOMESAUCE"
CLIENT: I received this from the server: "I am the server, and I got your message"
[1]+  Done                    ./server 51717
$
```

multiserver.c 1 of 2

This server lives forever by wrapping the `accept()` section in a while loop

It is also intended to pair with client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(const char *msg) { perror(msg); exit(1); } // Error function used for reporting issues

int main(int argc, char *argv[])
{
	int listenSocketFD, establishedConnectionFD, portNumber, charsRead;
	socklen_t sizeOfClientInfo;
	char buffer[256];
	struct sockaddr_in serverAddress, clientAddress;

	if (argc < 2) { fprintf(stderr,"USAGE: %s port\n", argv[0]); exit(1); } // Check usage & args

	// Set up the address struct for this process (the server)
	memset((char *)&serverAddress, '\0', sizeof(serverAddress)); // Clear out the address struct
	portNumber = atoi(argv[1]); // Get the port number, convert to an integer from a string
	serverAddress.sin_family = AF_INET; // Create a network-capable socket
	serverAddress.sin_port = htons(portNumber); // Store the port number
	serverAddress.sin_addr.s_addr = INADDR_ANY; // Any address is allowed for connection to this process

	// Set up the socket
	listenSocketFD = socket(AF_INET, SOCK_STREAM, 0); // Create the socket
	if (listenSocketFD < 0) error("ERROR opening socket");
```

multiserver.c 2 of 2

```c
    // Enable the socket to begin listening
    if (bind(listenSocketFD, (struct sockaddr *)&serverAddress, sizeof(serverAddress)) < 0) // Connect socket to port
            error("ERROR on binding");
    listen(listenSocketFD, 5); // Flip the socket on - it can now receive up to 5 connections
  while (1) {
    // Accept a connection, blocking if one is not available until one connects
    sizeOfClientInfo = sizeof(clientAddress); // Get the size of the address for the client that will connect
    establishedConnectionFD = accept(listenSocketFD, (struct sockaddr *)&clientAddress, &sizeOfClientInfo); // Accept
    if (establishedConnectionFD < 0) error("ERROR on accept");
    printf("SERVER: Connected Client at port %d\n", ntohs(clientAddress.sin_port));
    // Get the message from the client and display it
    memset(buffer, '\0', 256);
    charsRead = recv(establishedConnectionFD, buffer, 255, 0); // Read the client's message from the socket
    if (charsRead < 0) error("ERROR reading from socket");
    printf("SERVER: I received this from the client: \"%s\"\n", buffer);

    // Send a Success message back to the client
    charsRead = send(establishedConnectionFD, "I am the server, and I got your message", 39, 0); // Send success back
    if (charsRead < 0) error("ERROR writing to socket");
    close(establishedConnectionFD); // Close the existing socket which is connected to the client
  }
    close(listenSocketFD); // Close the listening socket
    return 0;
}
```

Client/Multiserver Results

Note that the order of the client and server sending text to the terminal is hard to control!

```
$ multiserver 55556 &
[1] 26889
$ client localhost 55556
CLIENT: Enter text to send to the server, and then hit enter: SERVER: Connected Client at port 38422
My Test!
SERVER: I received this from the client: "My Test!"
CLIENT: I received this from the server: "I am the server, and I got your message"
$ client localhost 55556
CLIENT: Enter text to send to the server, and then hit enter: SERVER: Connected Client at port 38424
So much text!!
SERVER: I received this from the client: "So much text!!"
CLIENT: I received this from the server: "I am the server, and I got your message"
$ kill -TERM 26889
[1]+  Terminated              multiserver 55556
```

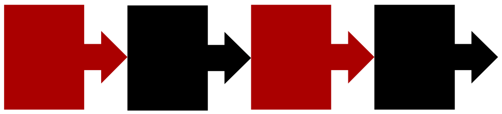
Doing it All at Once

- Many clients may need to both connect and perform tasks at the same time
- We want to minimize:
  - Response time
  - Complexity
- Want to maximize:
  - Throughput (connections serviced / second)
  - Hardware utilization (%CPU usage)

- These are all tradeoffs of each other!
- Let's look at two methods…

Iterative Servers

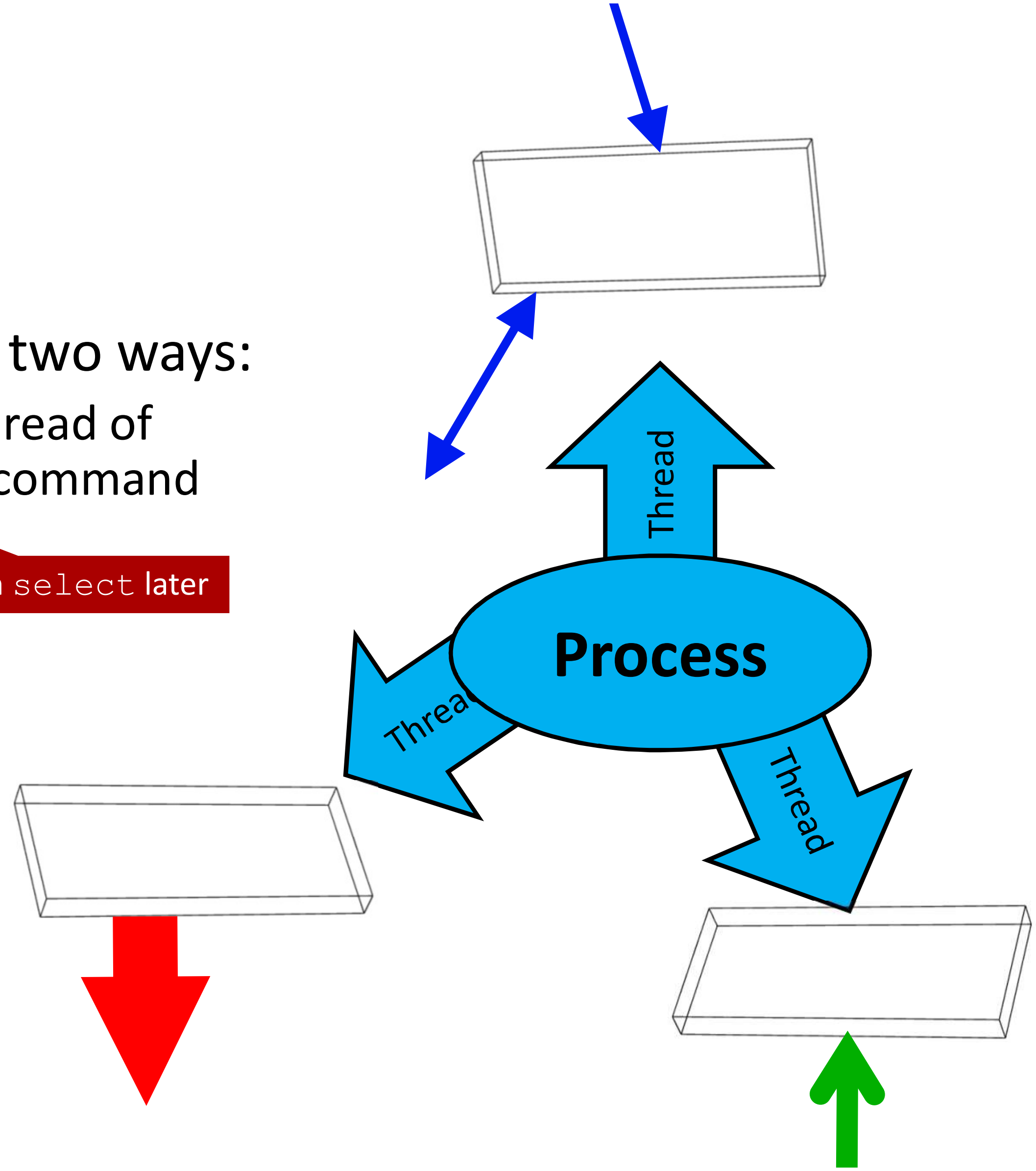
- Iterative
  - Handles only one client at a time
  - Non-preemptive: additional client must wait for all previous requests to complete
- Easy to design, implement, and maintain
- Best when:
  - Request processing time is short
  - No I/O is needed by server
  - Order matters



Concurrent Servers

- Concurrency can be provided in two ways:
  - Apparent concurrency: a single thread of execution, using the `select()` command and non-blocking I/O
  - Real concurrency: multiple threads of execution, or multiple processes, each with one thread

More on `select` later

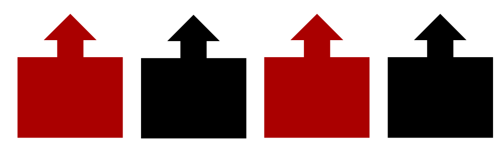


Apparent Concurrency: Details

- Only one thread, no preemption, using non-blocking I/O
- Whenever an I/O request would block, switch to another connection
- Up to a certain number of clients being server:
  - Maximizes CPU utilization
  - Increases throughput
- Complexity involves tracking connections, choosing the next to run, detecting blocking calls, etc.
- Works well if requests are short

Real Concurrency: Details

- Preemptive
  - Clients can connect anytime to the server, which uses multiple threads or processes to service connections
- Up to a certain number of connections:
  - Maximizes CPU utilization
  - Maximizes response time
  - Increases throughput
- Harder to design, implement, and *maintain:*
  - After too many concurrent connections:
    - Everything gets worse -> server eventually hangs
    - Need to put limits on concurrent connections



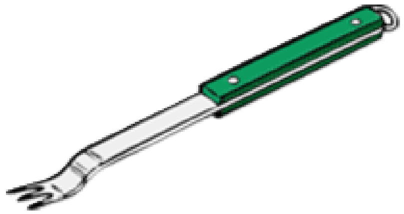
More Real Concurrency

- Four different methods
  - Create one process per client connection
  - Create a pool of available processes before clients connect
  - Use only one process, but create one thread per client connection
  - Use only one process, but create a pool of available threads before clients connect

Fork Solution #1

- One process per client connection
- Fork a new process to handle every connection
- Advantages:
  - Simple: minimal shared state to worry about
- Disadvantages:
  - Process creation via `fork()` is slow
  - Context-switching between processes is also slow, but minor compared to `fork()`

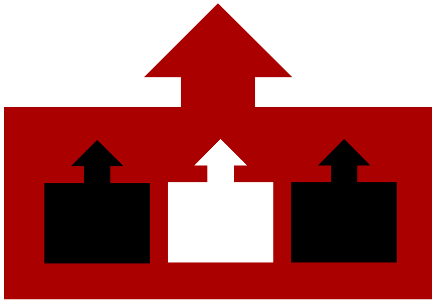
Fork Solution #2



- Create a pool of available processes for clients to use
- Advantages:
  - No longer have to fork
  - Have rapid response as long as there is an idle process available
  - Can set the pool size, so that you don't overload the hardware
- Disadvantages:
  - Still have process context switching
  - Managing the pool of processes can be complex

Threads Solutions 1 & 2

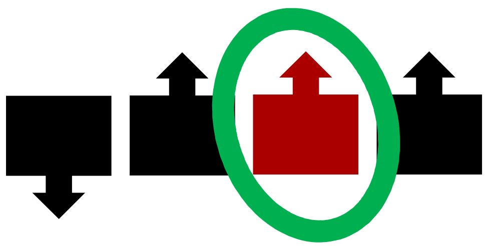
- Threads allow multiple concurrent execution contexts within a single process
- Can implement a server as a single process with multiple threads
  - Either one thread per connection, or a pool of threads
- Advantages:
  - Trades process context switches (slow) for thread switches (fast)
  - Shared address space, shared code, shared data, etc.
- Disadvantages:
  - Code must be thread-safe
  - Must always worry about inadvertent data-sharing



`select()`

- `select()` is designed for server-like applications that have many communication channels open at once (like a pool of threads)
  - Data or space may become available at any time on *any* of the channels
  - You want to minimize the delay between when data/space becomes available and your process takes action

- Overview: you call `select()` with a list of read and/or write file descriptors, and it *returns* when any one of those descriptors becomes readable or writable



select()

```
int select(
        int nfds,                   // Highest numbered FD + 1
        fd_set* readfds,            // Input FDs of interest
        fd_set* writefds,           // Output FDs of interest
        fd_set* errorfds,           // FDs where exception has occurred
        struct timeval* timeout     // when to time out if nothing happens
)
```

- The three parameters `readfds`, `writefds`, and `errorfds` are bit masks
  - Each bit of the number refers to one file descriptor
  - Bit 0 is file descriptor 0, bit 1 is file descriptor 1, etc.
- UNIX provides you with macros to manipulate bit masks:
  - `FD_ZERO()` :: Set all bits to 0
  - `FD_SET()` :: Set one specific bit to 1
  - `FD_ISSET()` :: Determine if a specific bit is set to 1
  - `FD_CLR()` :: Set one specific bit to 0

`select()` Return Values

- -1 if error
- 0 if time out: nothing ready
- Else, the return value is the *number* of file descriptors ready for reading, writing, or have had errors occur

selectDemo.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
	fd_set readFDs;
	struct timeval timeToWait;
	int retval;

	// Watch stdin (FD 0) to see when it has input
	FD_ZERO(&readFDs); // Zero out the set of possible read file descriptors
	FD_SET(0, &readFDs); // Mark only FD 0 as the one we want to pay attention to

	// Wait up to 50 seconds
	timeToWait.tv_sec = 50;
	timeToWait.tv_usec = 0;

	retval = select(1, &readFDs, NULL, NULL, &timeToWait);  // Check to see whether any read FDs have data!
	                                                        // After select returns, timeToWait is undefined
	if (retval == -1)
		perror("select()");
	else if (retval)
		printf("Data is available now!\n"); // FD_ISSET(0, &readFDs) will return true
	else
		printf("No data within 50 seconds\n");

	return(0);
}
```

This example comes from the `select()` man page

Together with returning an int, `select()` *also* overwrites your bit masks to show you *which* bits are interesting; you'll have to iterate through them to see which ones are set, though

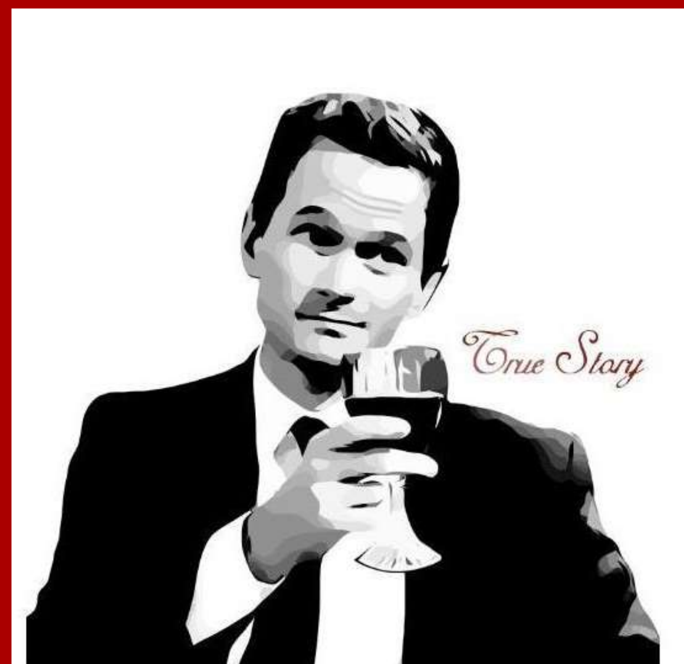
selectDemo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
        fd_set re
        struct ti
        int retva

        // Watch
        FD_ZERO(&
        FD_SET(0,

        // Wait u
        timeToWai
        timeToWai

        retval =                                                    ad FDs have data!
                                                                    oWait is undefined
        if (retva

        else if                                                     e

        else


        return(0)
}
```

In marking up this code and testing it, the double slash comments // were confusing the copy/paste functions in vim. So, I changed them all to single slashes, pasted it all into vim, and then entered this command:

```
:%s/\//\/\//g
```

… and it worked, first try.



selectDemo.c Results

```
$ mkfifo myfifo
$ selectDemo < myfifo &
[1] 17013
$ echo "text" > myfifo
Data is available now!
[1]+  Done                    selectDemo < myfifo
```

This hooks the output of the echo command to the input of the selectDemo program through a named pipe!

As soon as both ends are opened, and data is written, the pipe transfers the data

echo → myfifo → selectDemo