CS111

Week 3

Slides by: Rishab, Nikita, Karen

Things we are going to discuss:

- Recap of Project 1A
- Illustration of Project 1A and Project 1B
- Overview of Project 1B:The Big Picture
- Introduction
- Socket
  - Address Domain
  - Socket Type
  - Reference:
    - https://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html
    - https://www.youtube.com/watch?v=LtXEMwSG5-8
- TCP/UDP
- Important functions for socket programming + Demo
- Ports
- Options

And as you know:

- Project 1B
  - Spec: http://web.cs.ucla.edu/~harryxu/courses/111/winter20/ProjectGuide/P1B.html
  - Due on Wednesday (01/29/2020) at 11:59 PM
- **Late Policy** : Exponential as discussed previously
- Please remember to **download your submissions** and check if you submitted the correct files. Empty submissions - or submissions in the wrong format - cannot be graded and will therefore be scored with a 0

Project 1B: Compressed Network Communication

In this project, you will build a multi-process telnet-like client and server. This project is a continuation of Project 1A. It can be broken up into two major steps:

- Passing input and output over a TCP socket.
- Compressing communication between the client and server.

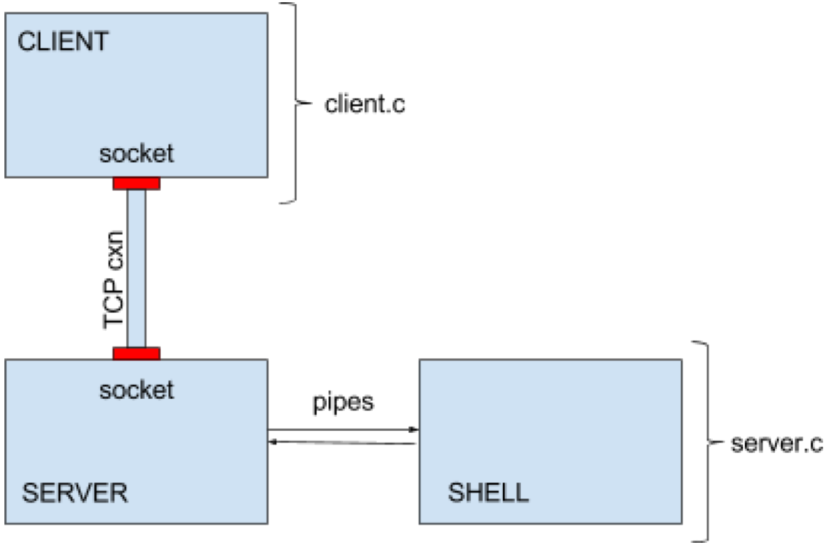
Project Objectives:

- Demonstrate the ability to research new APIs and debug code that exploits them
- Demonstrate the ability to do basic network communication
- Demonstrate the ability to exploit a new library
- Develop multi-process debugging skills

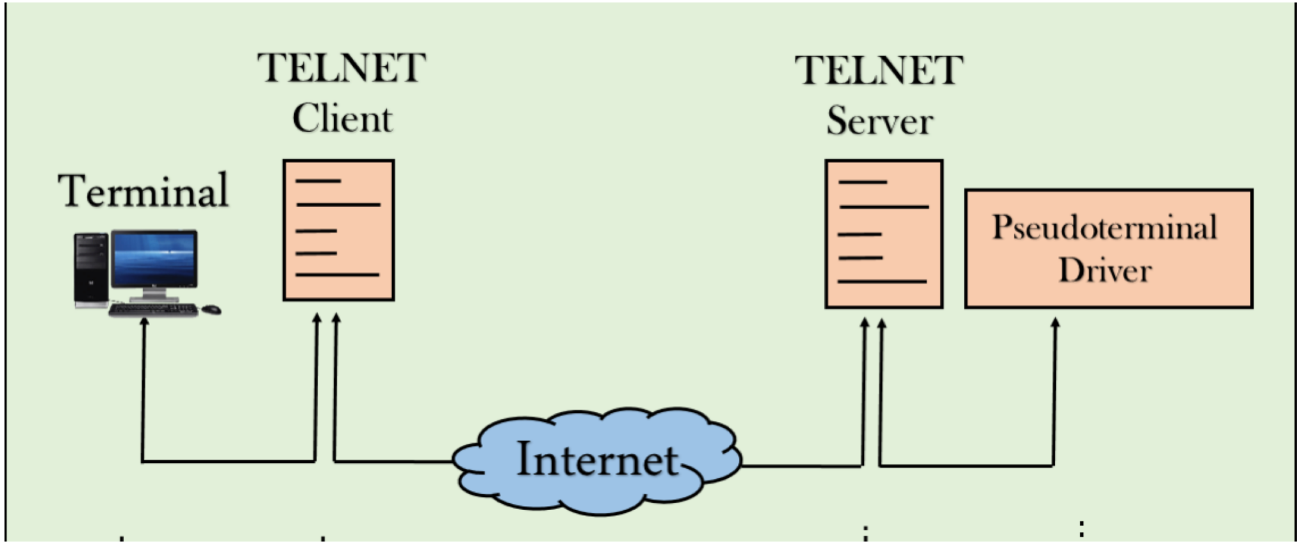
The big picture

Using Project 1A’s --shell option as a starting point create a client program (lab1b-client) and a server program (lab1b-server), both of which support a mandatory --port=port# switch.

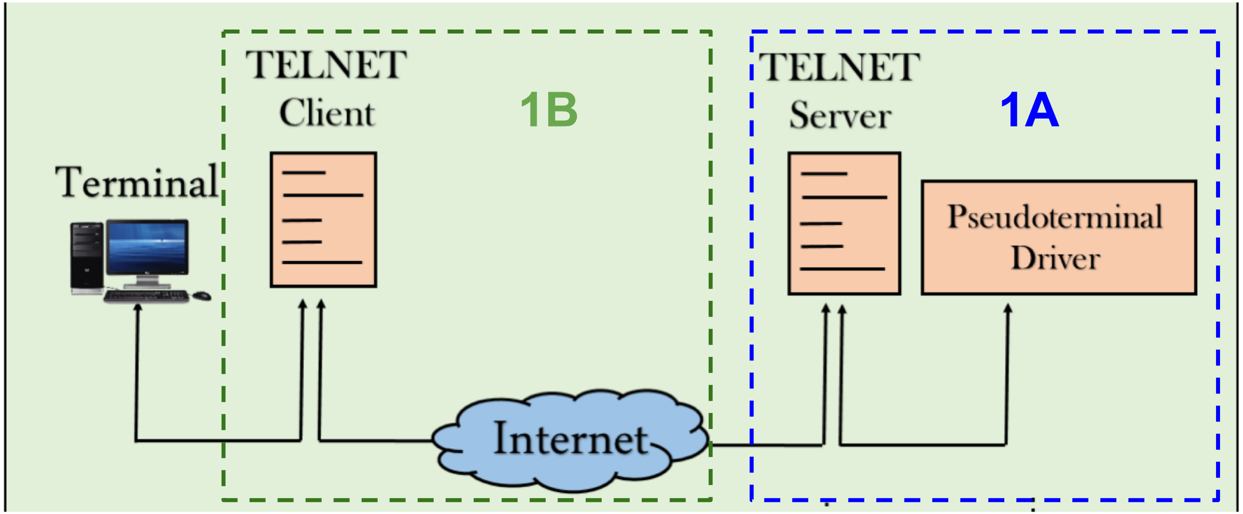
Your overall program design will look like the diagram to the right. You have a client process, a server process, and a shell process. The first two are connected via TCP connection, and the last two are connected via pipes.



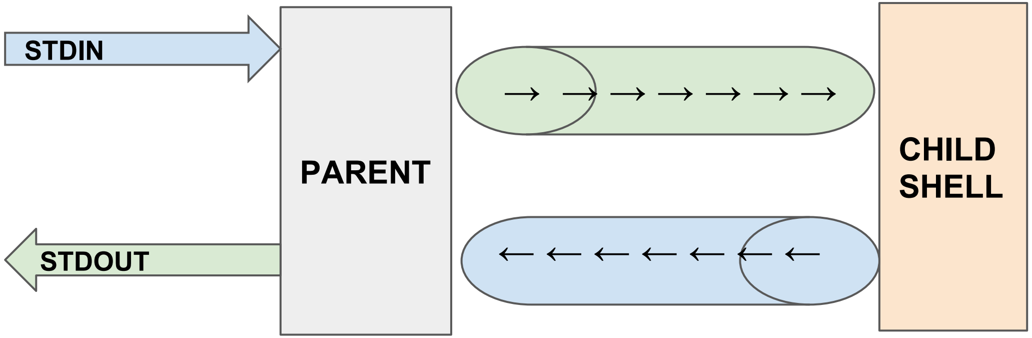
Project 1 - Telnet

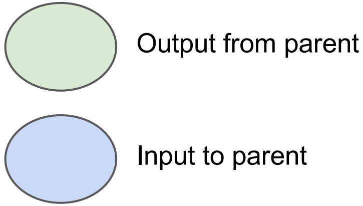


Project 1A - Where are we?



Project 1A - Recap





Project 1A - parent

Change keyboard input to character at a time, i.e., non-canonical input mode

- termios, tcssetattr, tcsgetattr

Create a child shell process and execute commands

- fork() to create, exec() to execute

Read input from keyboard pass it to child shell after processing

Read processed output from child shell

Project 1A - parent

Read input from keyboard

1. Display input on stdout (screen)
2. Send input to child-shell

Display processed output from child shell

1. Output of child-shell on processing must be shown on stdout (screen)

Project 1A - parent

Special cases:

If Ctrl+D is received from keyboard:

- Stop reading input from keyboard
- Process any remaining output from child shell
- Restore normal terminal modes
- Report the status
- Exit the program

Project 1A - parent

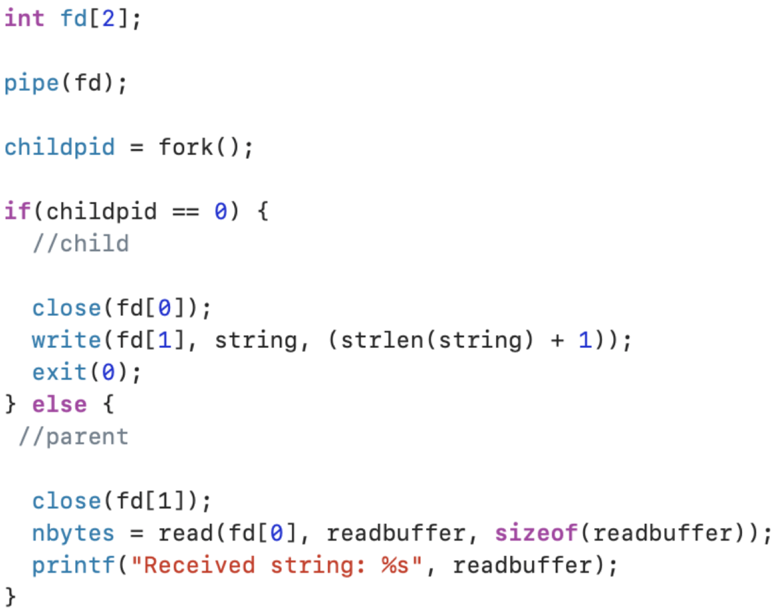
Special cases:

If Ctrl+C is received from keyboard:

- Stop reading input from keyboard
- Use kill(2) to send a SIGINT to shell program
- Process any remaining output from child shell
- Restore normal terminal modes
- Report the status
- Exit the program

Project 1A - communication - Pipes recap

- Unidirectional flow of data from one process to another.
- Buffers data till read end of pipe doesn't consume data present at write end of the pipe.



```c
int fd[2];

pipe(fd);

childpid = fork();

if(childpid == 0) {
  //child

  close(fd[0]);
  write(fd[1], string, (strlen(string) + 1));
  exit(0);
} else {
 //parent

  close(fd[1]);
  nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
  printf("Received string: %s", readbuffer);
}
```

Pipes

Writing from child to parent:

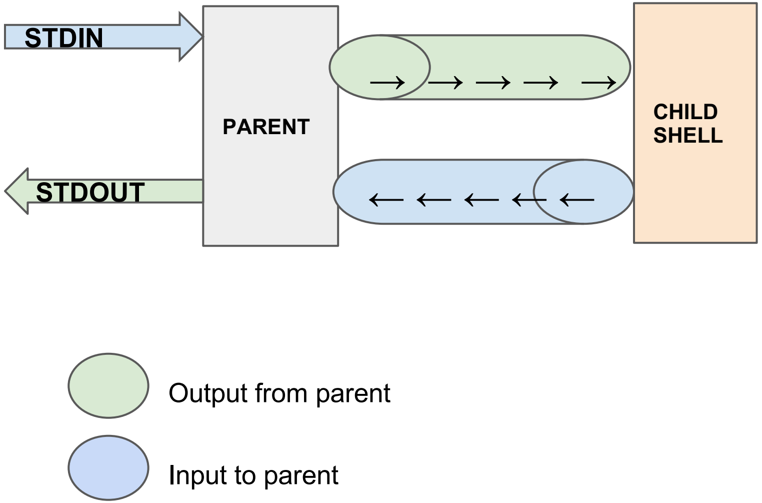
Child: I don’t need read end of pipe, I will close it

Parent: I don’t need write end of pipe, I will close it

Child: I want to send a message to parent, I will write to write end of pipe

Parent: I want to receive messages from child, I will read from read end of pipe

Project 1a - pipes recap



2 pipes:

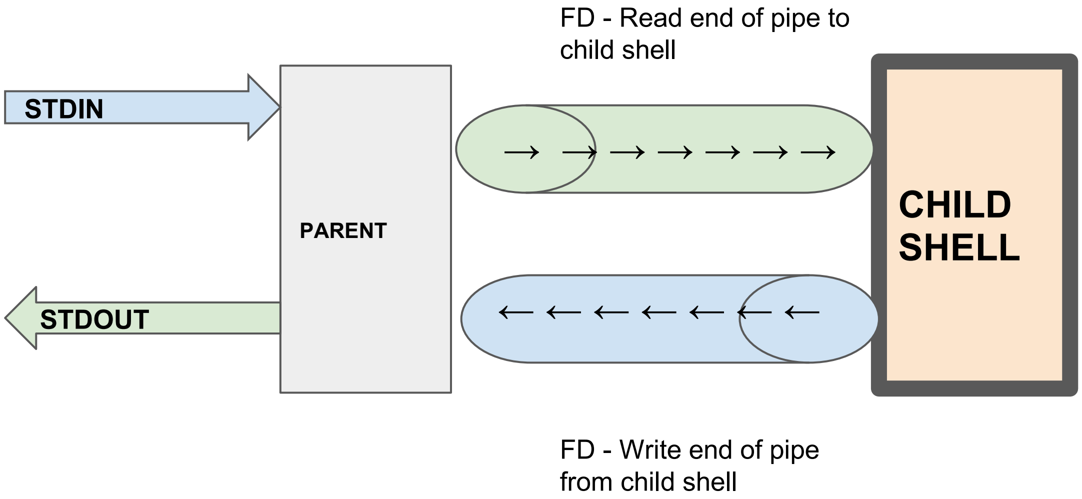
Write to child shell

Read from child shell

Parent, Child:

- I don’t need to read from which pipe?,
  I can close the read end of that pipe.
- I don’t need to send data to which pipe?,
  I can close the write end of that pipe.

Project 1a - File Descriptors - Child


FD - Read end of pipe to child shell
STDIN
PARENT
CHILD SHELL
STDOUT
FD - Write end of pipe from child shell

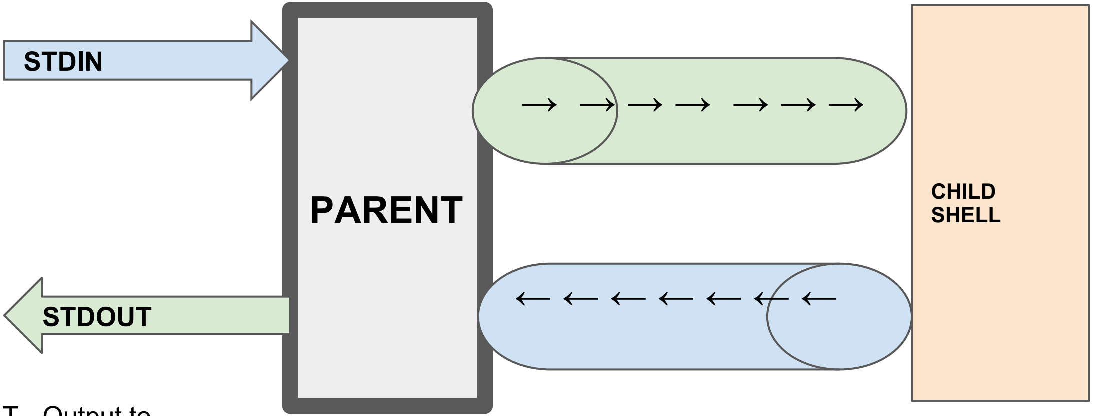
Redirect / Change stdin and stdout of child shell to the appropriate pipe ends.

Recall dup()

Project 1a - File Descriptors - Parent

STDIN - Input from Keyboard

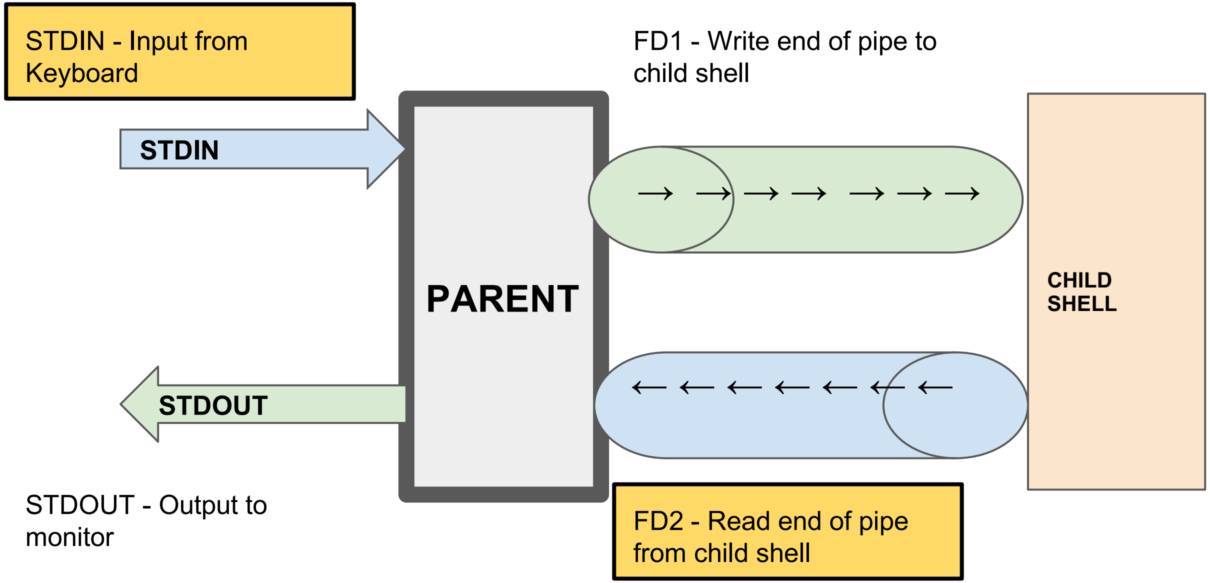
FD1 - Write end of pipe to child shell



STDOUT - Output to monitor

FD2 - Read end of pipe from child shell

Project 1a - Poll



STDIN - Input from Keyboard

FD1 - Write end of pipe to child shell

FD2 - Read end of pipe from child shell

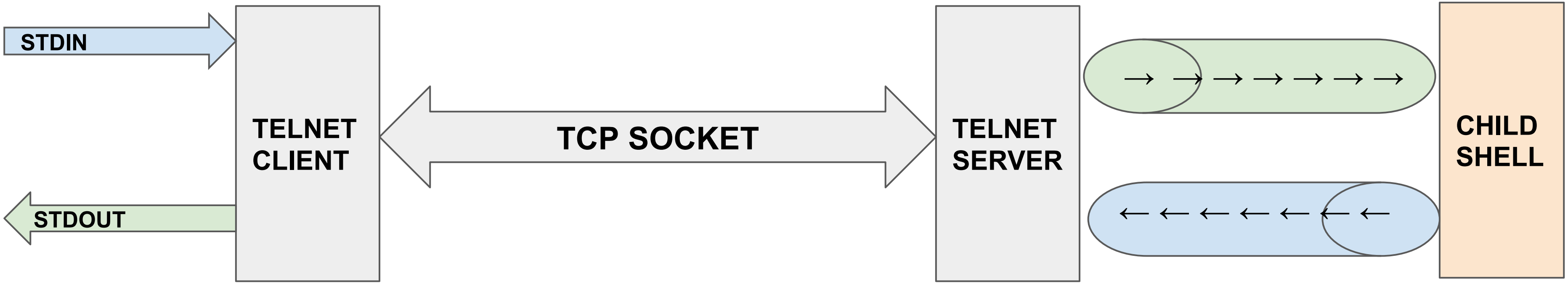
STDOUT - Output to monitor

Parent reading from 2 inputs (highlighted in yellow).

Need to poll between the file descriptors to see if there is any new input

poll()

Project 1b



STDIN

STDOUT

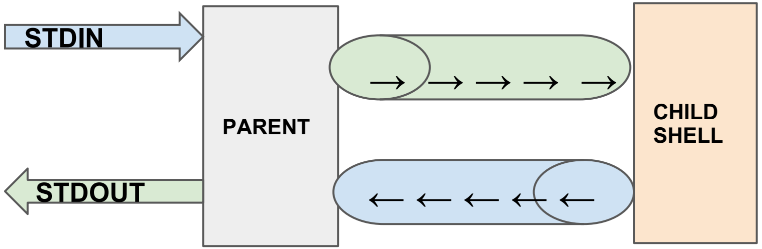
TELNET CLIENT

TCP SOCKET

TELNET SERVER

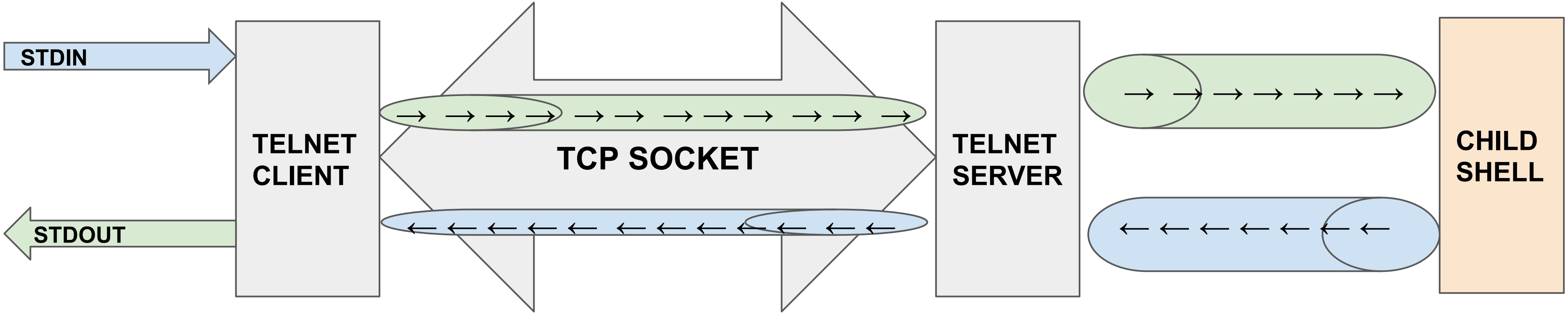
CHILD SHELL

Project 1b

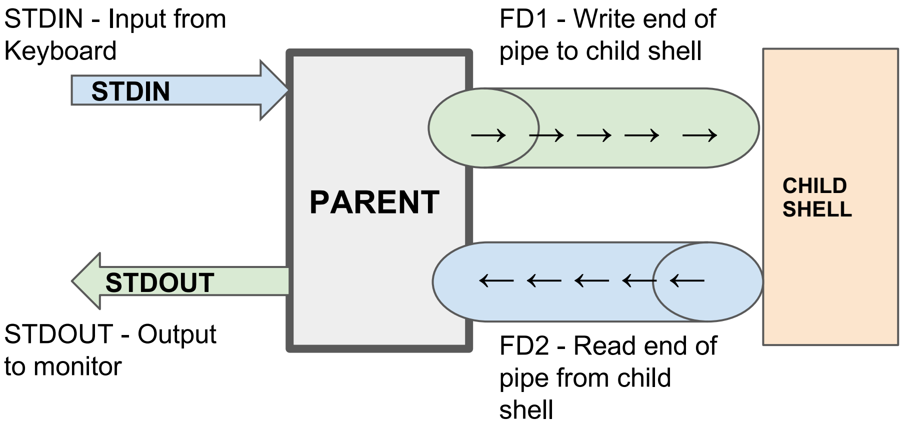


TCP Socket -

- Socket is again a file descriptor
- Full duplex - you can read and write from the same file descriptor



Project 1b - File descriptors (TELNET client)



STDIN - Input from Keyboard

STDIN

PARENT

FD1 - Write end of pipe to child shell

CHILD SHELL

STDOUT

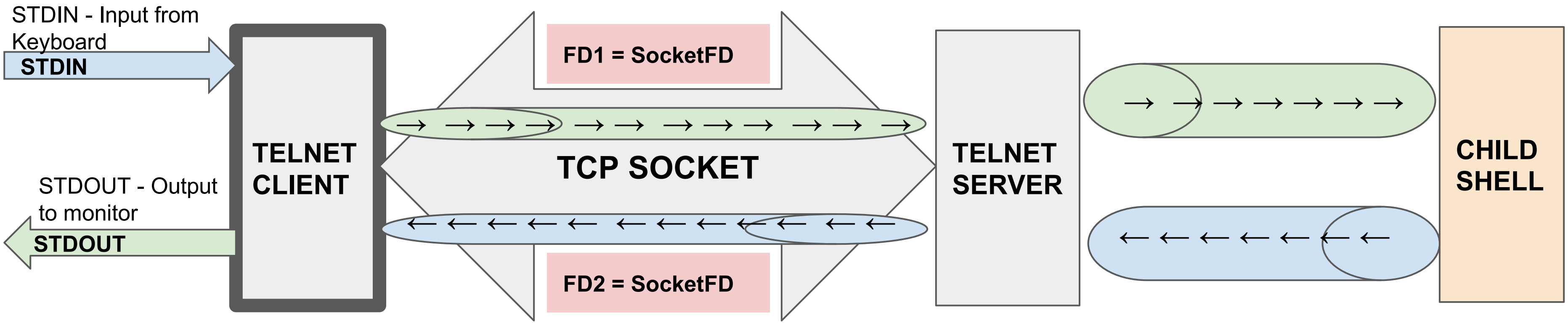
STDOUT - Output to monitor

FD2 - Read end of pipe from child shell

TCP Socket -

- Socket is again a file descriptor
- Full duplex - you can read and write from the same file descriptor

🟥 Modified file descriptors



STDIN - Input from Keyboard

STDIN

TELNET CLIENT

STDOUT - Output to monitor

STDOUT

FD1 = SocketFD

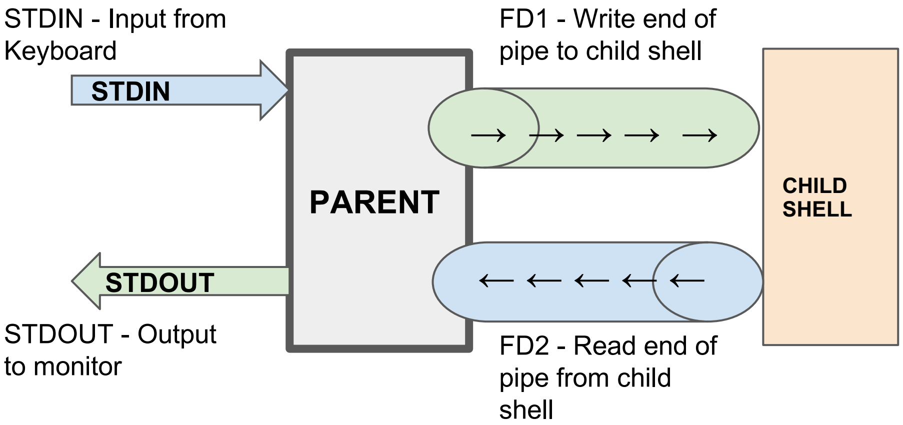
TCP SOCKET

FD2 = SocketFD

TELNET SERVER

CHILD SHELL

Project 1b - File descriptors (TELNET server)



STDIN - Input from Keyboard

STDIN

PARENT

STDOUT

STDOUT - Output to monitor

FD1 - Write end of pipe to child shell

→ → → → →

← ← ← ← ←

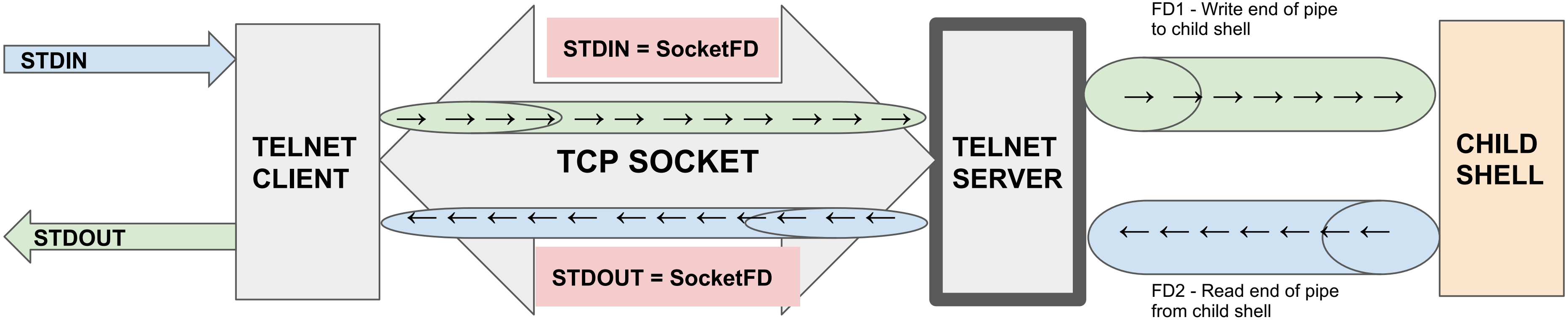
FD2 - Read end of pipe from child shell

CHILD SHELL

TCP Socket -

- Socket is again a file descriptor
- Full duplex - you can read and write from the same file descriptor

🟥 Modified file descriptors



STDIN

STDOUT

TELNET CLIENT

STDIN = SocketFD

→ → → → → → → → → → → →

TCP SOCKET

← ← ← ← ← ← ← ← ← ← ← ←

STDOUT = SocketFD

TELNET SERVER

FD1 - Write end of pipe to child shell

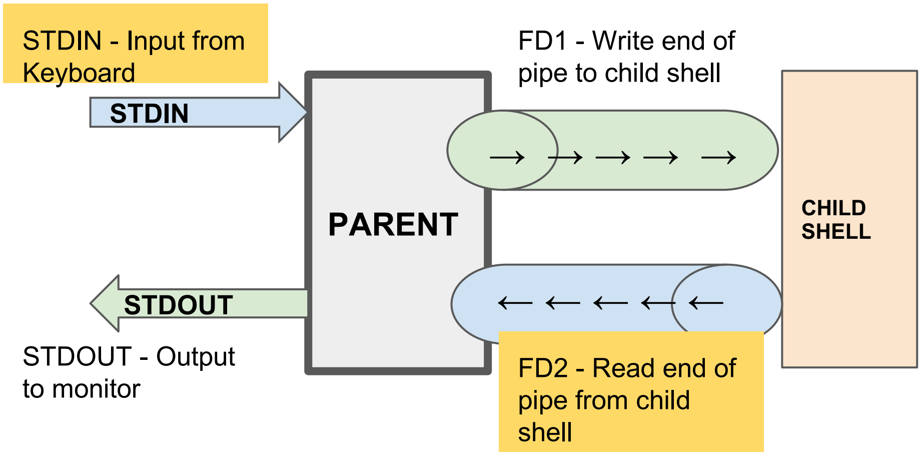
→ → → → → → → →

← ← ← ← ← ← ←

FD2 - Read end of pipe from child shell

CHILD SHELL

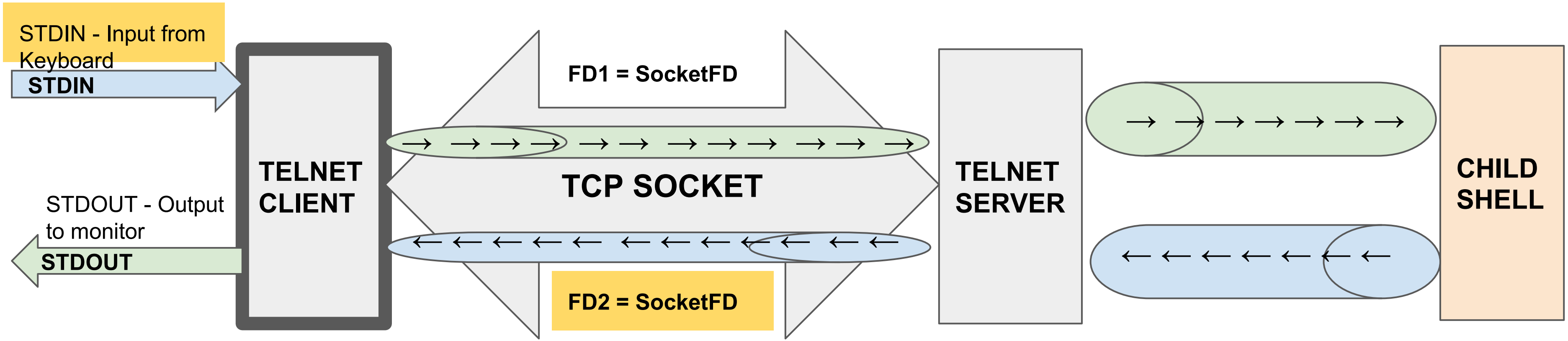
Project 1b - Poll (TELNET client)


STDIN - Input from Keyboard
STDIN
FD1 - Write end of pipe to child shell
PARENT
CHILD SHELL
STDOUT
STDOUT - Output to monitor
FD2 - Read end of pipe from child shell

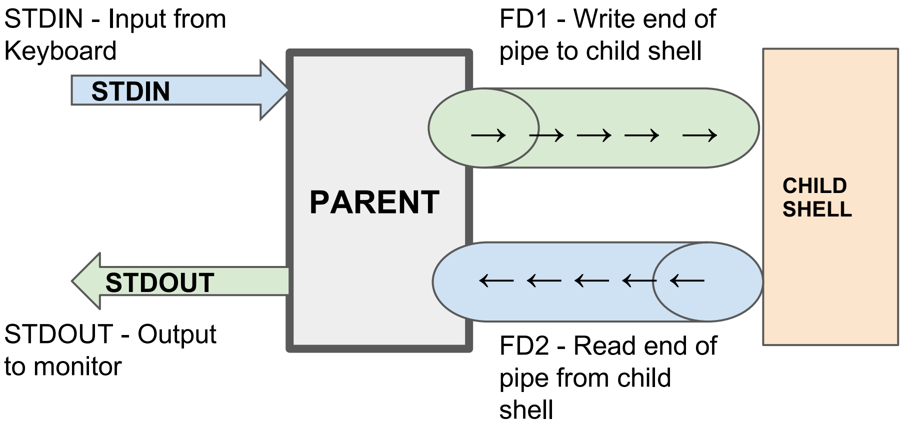
TCP Socket -

- Socket is again a file descriptor
- Full duplex - you can read and write from the same file descriptor

🟨 FD’s being polled



Project 1b - Poll (TELNET server)



STDIN - Input from Keyboard

STDIN

PARENT

STDOUT

STDOUT - Output to monitor

FD1 - Write end of pipe to child shell

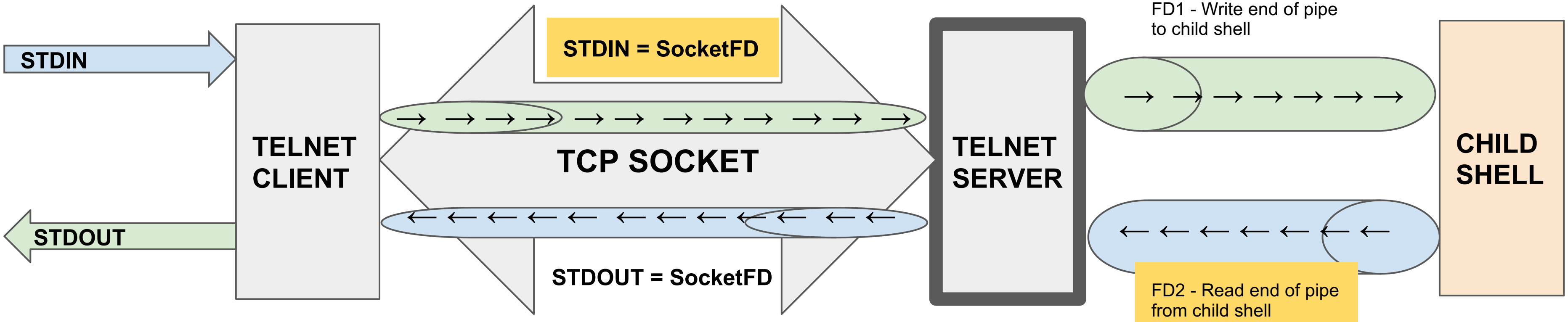
FD2 - Read end of pipe from child shell

CHILD SHELL

TCP Socket -

- Socket is again a file descriptor
- Full duplex - you can read and write from the same file descriptor

🟨 FD's being polled



STDIN

STDOUT

TELNET CLIENT

STDIN = SocketFD

TCP SOCKET

STDOUT = SocketFD

TELNET SERVER

FD1 - Write end of pipe to child shell

FD2 - Read end of pipe from child shell

CHILD SHELL

Socket Programming - Server

- Create a socket with the socket() system call
- Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the listen() system call
- Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
- Send and receive data

Socket Programming - Client

1. Create a socket with the socket() system call
2. Connect the socket to the address of the server using the connect() system call
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

Socket

- When a socket is created, the program has to specify the address domain and the socket type.

- Two processes can communicate with each other only if their sockets are of the same **type** and in the same **domain**.

- So, let’s discuss address domains and socket types.

Address Domain

- There are two widely used address domains. Each of these has its own address format.
    - *Unix domain* :
        - Two processes which share a common file system communicate
        - The address of a socket in the Unix domain is a character string which is basically an entry in the file system.
    - *Internet domain* :
        - Two processes running on any two hosts on the Internet communicate
        - The address of a socket consists of the Internet address of the host machine (its IP address)
        - In addition, each socket needs a port number on that host. Port numbers are 16 bit unsigned integers. The lower numbers are reserved in Unix for standard services.
        - For example, the port number for the FTP server is 21. It is important that standard services be at the same port on all computers so that clients will know their addresses. However, port numbers above 2000 are generally available.

Socket Type

- There are two widely used socket types, *stream sockets*, and *datagram sockets*.
  - Stream sockets:
    - Treat communications as a continuous stream of characters
    - Uses TCP (Transmission Control Protocol), which is a reliable, stream oriented protocol
  - Datagram sockets:
    - Have to read entire messages at once
    - Uses UDP (Unix Datagram Protocol), which is unreliable and message oriented.

The examples in this tutorial will use sockets in the **Internet domain** using the **TCP protocol.**

How will we use socket programming

Since telnet server and client need to communicate over internet:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

We will create TCP sockets to communicate between two processes over the internet.

TCP: A protocol that ensures that you as an application programmer need not worry about network inconsistencies.

- i.e., if a network packet is dropped due to any specific reason, TCP will ensure that it will resend that packet so that client and server need not worry about unreliable networks

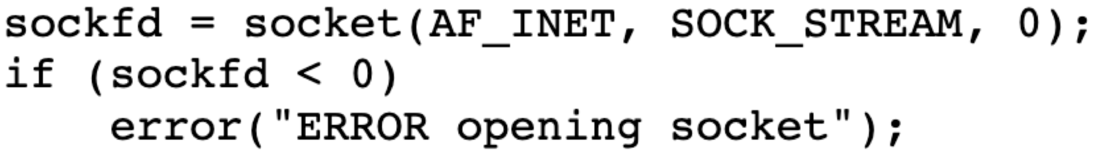
Follow up: Go through detailed explanation on socket programming here

Socket in detail

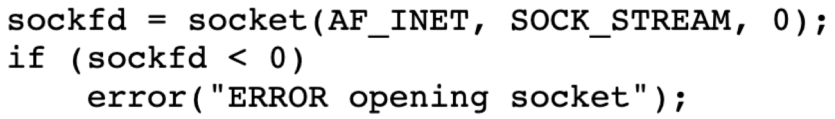
- **Function signature:** int socket(int *domain*, int *type*, int *protocol*);
- The ***socket***() function shall create an unbound socket in a communications domain, and return a file descriptor that can be used in later function calls that operate on sockets.
- The ***socket***() function takes the following arguments:
  - *Domain*
    - Specifies the communications domain in which a socket is to be created.
  - *Type*
    - Specifies the type of socket to be created.
  - *Protocol*
    - Specifies a particular protocol to be used with the socket. Specifying a *protocol* of 0 causes *socket*() to use an unspecified default protocol appropriate for the requested socket type.
- Upon successful completion, *socket*() shall return a non-negative integer, the socket file descriptor. Otherwise, a value of -1 shall be returned and *errno* set to indicate the error.

Example of a socket() call

- Before we dive into the demo, let’s take a look at a socket() call:
- Let’s see what each of these mean.



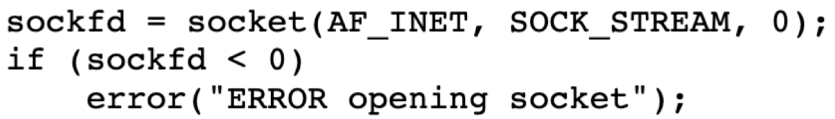
```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    error("ERROR opening socket");
```



```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    error("ERROR opening socket");
```

Example of a socket() call (continued)

- The socket() system call creates a new socket. It takes three arguments. The first is the address domain of the socket. Recall that there are two possible address domains, the unix domain for two processes which share a common file system, and the Internet domain for any two hosts on the Internet.
    - The symbol constant AF_UNIX is used for the former, and AF_INET for the latter (there are actually many other options which can be used here for specialized purposes).
- The second argument is the type of socket. Recall that there are two choices here, a stream socket in which characters are read in a continuous stream as if from a file or pipe, and a datagram socket, in which messages are read in chunks. The two symbolic constants are SOCK_STREAM and SOCK_DGRAM.



```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    error("ERROR opening socket");
```

Example of a socket() call (continued)

- The third argument is the protocol. If this argument is zero (and it always should be except for unusual circumstances), the operating system will choose the most appropriate protocol. It will choose TCP for stream sockets and UDP for datagram sockets.
- The socket system call returns an entry into the file descriptor table (i.e. a small integer). This value is used for all subsequent references to this socket. If the socket call fails, it returns -1. In this case the program displays and error message and exits. However, this system call is unlikely to fail.
- This is a simplified description of the socket call; there are numerous other choices for domains and types, but these are the most common

Socket Programming Demo

Create Server.c client.c

Compile server : gcc -o server server.c

Compile client : gcc -o client client.c

Run server : ./server 8080

Run client : ./client localhost 8080

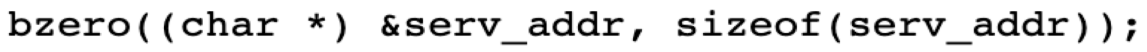
Quick look at some useful functions

- Let’s quickly run through some functions that might help us understand the ***server.c*** code better
- Functions we will discuss:
    - bzero()
    - bind()
    - listen()
    - accept()
    - read()
    - write()

bzero() - quick look at this useful function

- **Function Signature:** void bzero(void *s, int nbyte);
- This function does not return anything.
- Example:

- The function bzero() sets all values in a buffer to zero.
- It takes two arguments, the first is a pointer to the buffer and the second is the size of the buffer.
- Thus, this line initializes serv_addr to zeros.



```
bzero((char *) &serv_addr, sizeof(serv_addr));
```

bind() - a quick look at this useful function

- **Function Signature**:
  - `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
  - **bind**() assigns the address specified by ***addr*** to the socket referred to by the file descriptor ***sockfd***.
  - ***addrlen*** specifies the size, in bytes, of the address structure pointed to by ***addr***.
- When a socket is created with socket(), it exists in a namespace (address family) but has no address assigned
- The bind() system call binds a socket to an address, in this case the address of the current host and port number on which the server will run
- It takes three arguments:
  - Socket file descriptor
  - Address to which is bound: Pointer to a structure of type sockaddr, but what is passed in is a structure of type sockaddr_in, and so this must be cast to the correct type
  - Size of the address to which it is bound
- This can fail for a number of reasons, the most obvious being that this socket is already in use on this machine
- Returns 0 for success, -1 for error

listen() - a quick look at a useful function

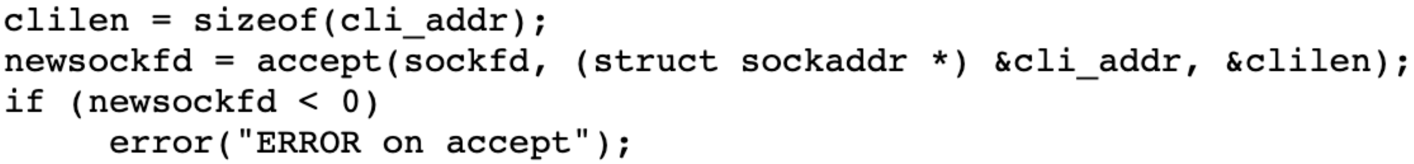
- **Function Signature**: `int listen(int sockfd, int backlog);`
- listen() marks the socket referred to by *sockfd* as a passive socket,that is, as a socket that will be used to accept incoming connection requests using accept()
- The first argument is the socket file descriptor, and the second is the size of the backlog queue, i.e., the number of connections that can be waiting while the process is handling a particular connection.
  - The *sockfd* argument is a file descriptor that refers to a socket of type SOCK_STREAM or SOCK_SEQPACKET.
  - The *backlog* argument defines the maximum length to which the queue ofpending connections for *sockfd* may grow.
- The listen system call allows the process to listen on the socket for connections.
- The backlog argument should be set to 5, the maximum size permitted by most systems.
- If the first argument is a valid socket, this call cannot fail, and so the code doesn't check for errors

accept() - a quick look at a useful function

- **Function Signature**: `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
- The **accept**() system call is used with connection-based socket types (**SOCK_STREAM**, **SOCK_SEQPACKET**).
- It extracts the first connection request on the queue of pending connections for the listening socket, ***sockfd***, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state.
- The original socket *sockfd* is unaffected by this call.
    - *sockfd* is a socket that has been created with socket(),bound to a local address with bind(), and is listening for connections after a listen().
- On success, these system calls return a nonnegative integer that is a file descriptor for the accepted socket.
- On error, -1 is returned,*errno* is set appropriately, and *addrlen* is left unchanged.

accept() - Example

```
struct sockaddr_in cli_addr;
unsigned clilen = sizeof (cli_addr);
```



```
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
if (newsockfd < 0)
     error("ERROR on accept");
```

- The accept() system call causes the process to block until a client connects to the server.
- Thus, it wakes up the server process when a connection from a client has been successfully established.
- It returns a new file descriptor, and all communication on this connection should be done using the new file descriptor.
- The second argument is a reference pointer to the address of the client on the other end of the connection, and the third argument is the size of this structure.

socket() and accept() big picture

- Each running process has a file descriptor table which contains pointers to all open i/o streams. When a process starts, three entries are created in the first three cells of the table. Entry 0 points to standard input, entry 1 points to standard output, and entry 2 points to standard error.
- Whenever a file is opened, a new entry is created in this table, usually in the first available empty slot.
- The **socket** system call returns an entry into this table; i.e. a small integer. This value is used for other calls which use this socket.
- The **accept** system call returns another entry into this table. The value returned by accept is used for reading and writing to that connection

read(), write() - some useful tips

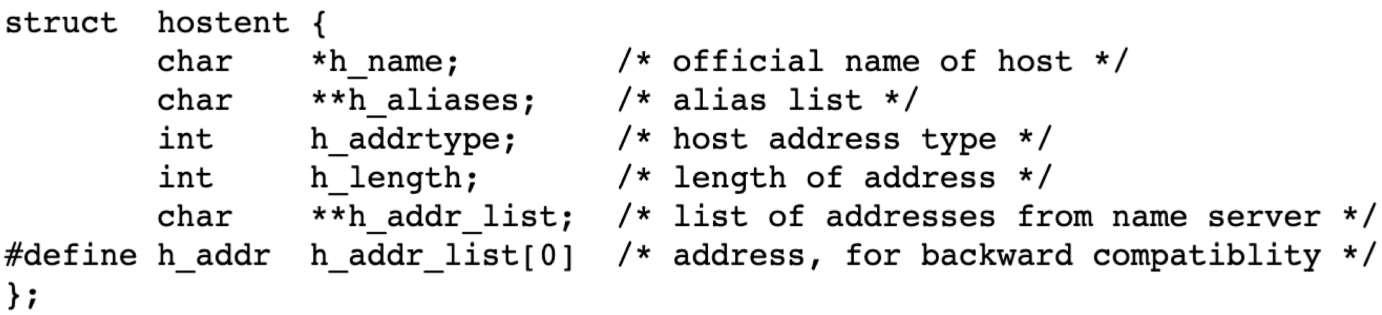
- Note that the **read** call uses the new file descriptor, the one returned by accept(), not the original file descriptor returned by socket().
- Note also that the **read()** will block until there is something for it to read in the socket, i.e. after the client has executed a **write()**.
- It will read either the total number of characters in the socket or 255, whichever is less
- Returns the number of characters read
- Once a connection has been established, both ends can both read and write to the connection.
- Naturally, everything written by the client will be read by the server, and everything written by the server will be read by the client.

Quick look at some useful functions

- Let’s quickly run through some functions that might help us understand the ***client.c*** code better
- Functions we will discuss:
    - hostent()
    - bcopy()
    - connect()

hostent()

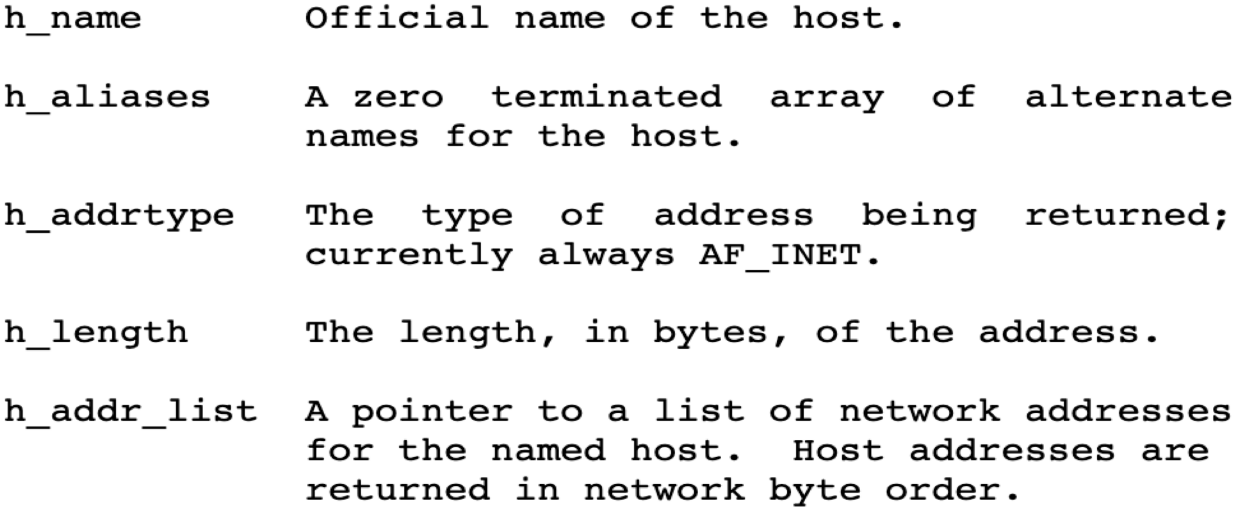
- The variable server is a pointer to a structure of type hostent. This structure is defined in the header file netdb.h as follows:



```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses from name server */
#define h_addr  h_addr_list[0]  /* address, for backward compatiblity */
};
```

hostent() (continued)

- It defines a host computer on the Internet. The members of this structure are:



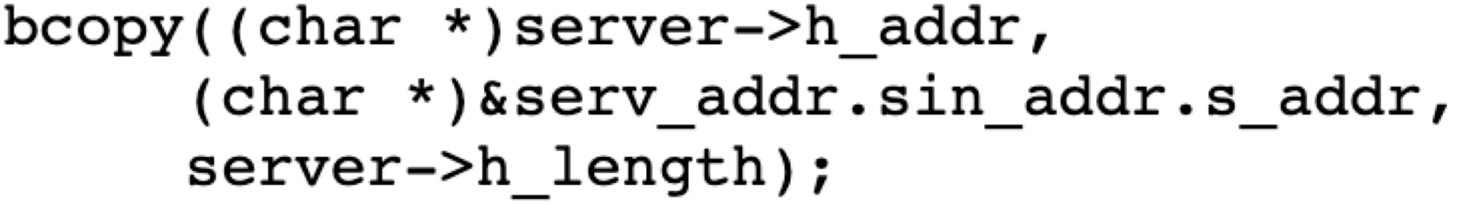
```
h_name       Official name of the host.

h_aliases    A zero  terminated  array  of  alternate
             names for the host.

h_addrtype   The  type  of  address  being  returned;
             currently always AF_INET.

h_length     The length, in bytes, of the address.

h_addr_list  A pointer to a list of network addresses
             for the named host.  Host addresses are
             returned in network byte order.
```

bcopy()

- **`void bcopy(char *s1, char *s2, int length)`** copies *length* bytes from *s1* to *s2*.
- So once we have set **`char *h_addr`**, we can copy it onto the

  (char *)&serv_addr.sin_addr.s_addr field

- Usage example:

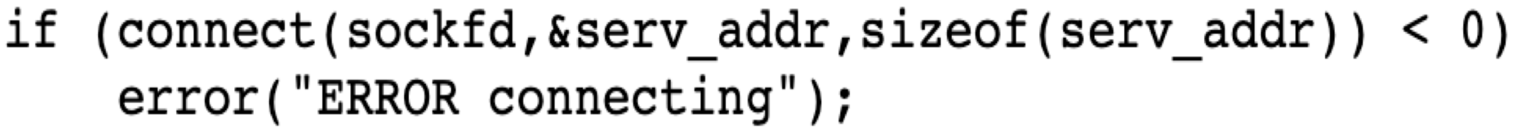


```
bcopy((char *)server->h_addr,
     (char *)&serv_addr.sin_addr.s_addr,
     server->h_length);
```

INADDR_ANY is used when you don't need to bind a socket to a specific IP. When you use this value as the address when calling bind() , the socket accepts connections to all the IPs of the machine.

connect()

- The connect function is called by the client to establish a connection to the server.
- It takes three arguments, the socket file descriptor, the address of the host to which it wants to connect (including the port number), and the size of this address.
- This function returns 0 on success and -1 if it fails.
- Example usage:



```
if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
    error("ERROR connecting");
```

What are ports?

- Any server machine makes its services available to the Internet using numbered ports, one for each service that is available on the server.
- For example, if a server machine is running a Web server and an FTP server, the Web server would typically be available on port 80, and the FTP server would be available on port 21.
- Clients connect to a service at a specific IP address and on a specific port.

What are ports? (continued)

- If the server machine accepts connections on a port from the outside world, and if a firewall is not protecting the port, you can connect to the port from anywhere on the Internet and use the service.
- Note that there is nothing that forces, for example, a Web server to be on port 80. If you were to set up your own machine and load Web server software on it, you could put the Web server on port 918, or any other unused port, if you wanted to.
  - Then, if your machine were known as xxx.yyy.com, someone on the Internet could connect to your server with the URL http://xxx.yyy.com:918.
  - The ":918" explicitly specifies the port number, and would have to be included for someone to reach your server.
- When no port is specified, the browser simply assumes that the server is using the well-known port 80.

What are ports? (continued)

- In both TCP and UDP, port numbers start at 0 and go up to 65535. Numbers in the lower ranges are dedicated to common internet protocols such as port 25 for SMTP and port 21 for FTP.
- It is important that standard services be at the same port on all computers so that clients will know their addresses.
- However, port numbers above 2000 are generally available.

Data Compression

The purpose of compression is to reduce the amount of space data takes up.

In communications scenarios, a compressed version of data will use less bandwidth to transmit the same data than its uncompressed form would use.

On the other hand, compression will require processing at the sending and receiving end before the data can be effectively used at the receiving process.

Sometimes the benefits in reduced bandwidth are worth the costs of extra processing and sometimes they are not, so compression is not used by default in network communications.

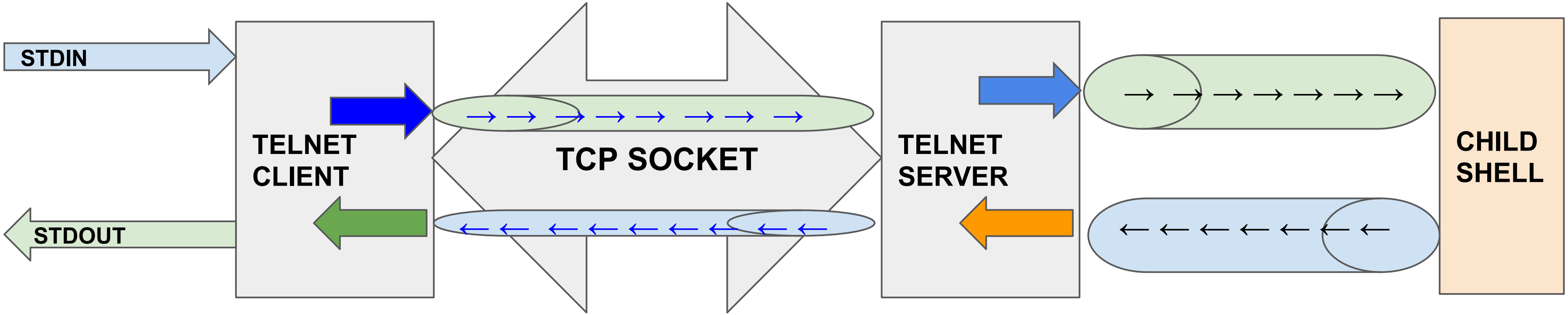
Data Compression - Server and Client

Add a --compress command line option to your client and server which, if included, will enable compression (of all traffic in both directions).

Modify both the client and server applications to compress traffic before sending it over the network and decompress it after receiving it.

Data Compression (Server and Client)

→ →	Compressed data flowing through TCP sockets
➡	Compress STDIN and send to socket
➡	Decompress socket data and send to child shell’s STDIN
⬅	Decompress socket data and send to client’s STDOUT
⬅	Compress child shell’s STDOUT and send to socket



Zlib usage

Create zpipe.c

Compile with -lz flag for including the zlib header file: `gcc -o zpipe -lz zpipe.c`

Compress a file(Deflate): `./zpipe < src > srcCompressed`

Decompress a file(Inflate): `./zpipe -d < srcCompressed > srcDecompressed`

Diff decompressed file and original file: `diff src srcDecompressed`

Zlib compression - man

Zlib Metadata: The *zlib* format has a very small header of only two bytes to identify it as a *zlib* stream and to provide decoding information, and a four-byte trailer with a fast check value to verify the integrity of the uncompressed data after decoding.

Zlib allows you to define compression levels in the range of -1 to 9. Lower compression levels result in faster execution, but less compression. Higher levels result in greater compression, but slower execution. The *zlib* constant Z_DEFAULT_COMPRESSION, equal to -1, provides a good compromise between compression and speed and is equivalent to level 6.

Zlib state is inside a struct called as z_stream_s, we are mainly responsible for these 3 fields:

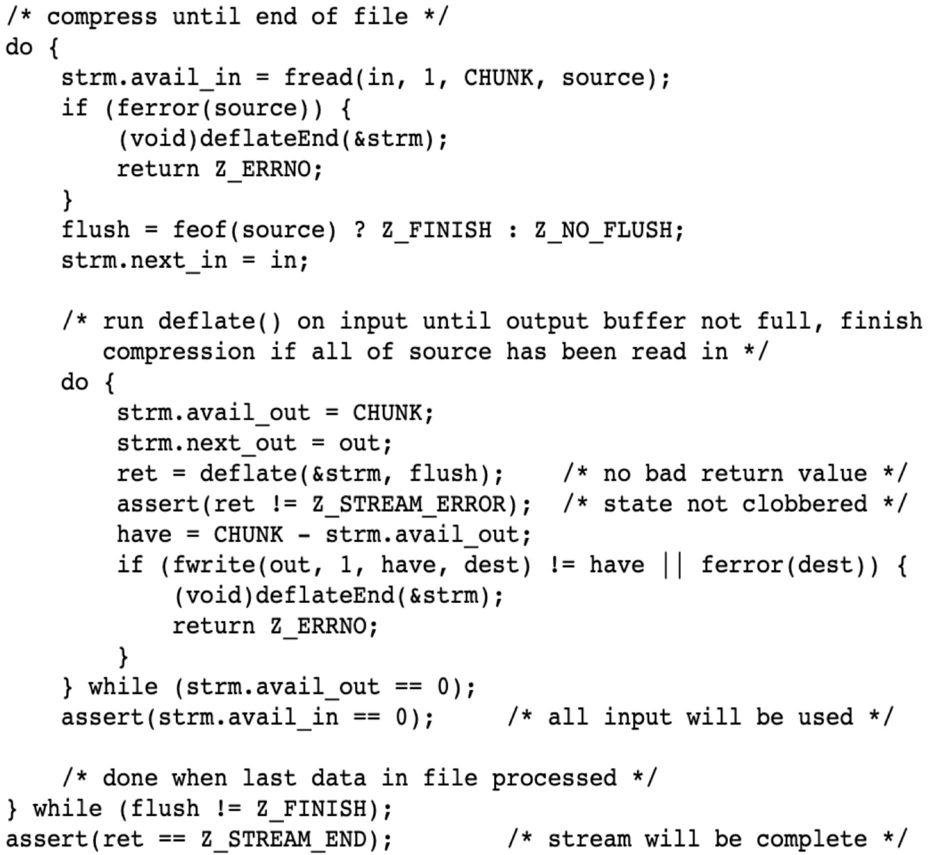
- **alloc_func zalloc; /* used to allocate the internal state */**
- **free_func zfree;   /* used to free the internal state */**
- **voidpf opaque; /* private data object passed to zalloc and zfree */**

The application must initialize zalloc, zfree and opaque before calling the init function. All other fields are set by the compression library and must not be updated by the application.

When zalloc and zfree are Z_NULL on entry to the initialization function, they are set to internal routines that use the standard library functions malloc() and free().

We will set zalloc and zfree to Z_NULL

Deflate (Compress)



```
/* compress until end of file */
do {
    strm.avail_in = fread(in, 1, CHUNK, source);
    if (ferror(source)) {
        (void)deflateEnd(&strm);
        return Z_ERRNO;
    }
    flush = feof(source) ? Z_FINISH : Z_NO_FLUSH;
    strm.next_in = in;

    /* run deflate() on input until output buffer not full, finish
       compression if all of source has been read in */
    do {
        strm.avail_out = CHUNK;
        strm.next_out = out;
        ret = deflate(&strm, flush);    /* no bad return value */
        assert(ret != Z_STREAM_ERROR);  /* state not clobbered */
        have = CHUNK - strm.avail_out;
        if (fwrite(out, 1, have, dest) != have || ferror(dest)) {
            (void)deflateEnd(&strm);
            return Z_ERRNO;
        }
    } while (strm.avail_out == 0);
    assert(strm.avail_in == 0);     /* all input will be used */

    /* done when last data in file processed */
} while (flush != Z_FINISH);
assert(ret == Z_STREAM_END);        /* stream will be complete */
```

deflate(): takes as many of the avail_in bytes at next_in as it can process, and writes as many as avail_out bytes to next_out.

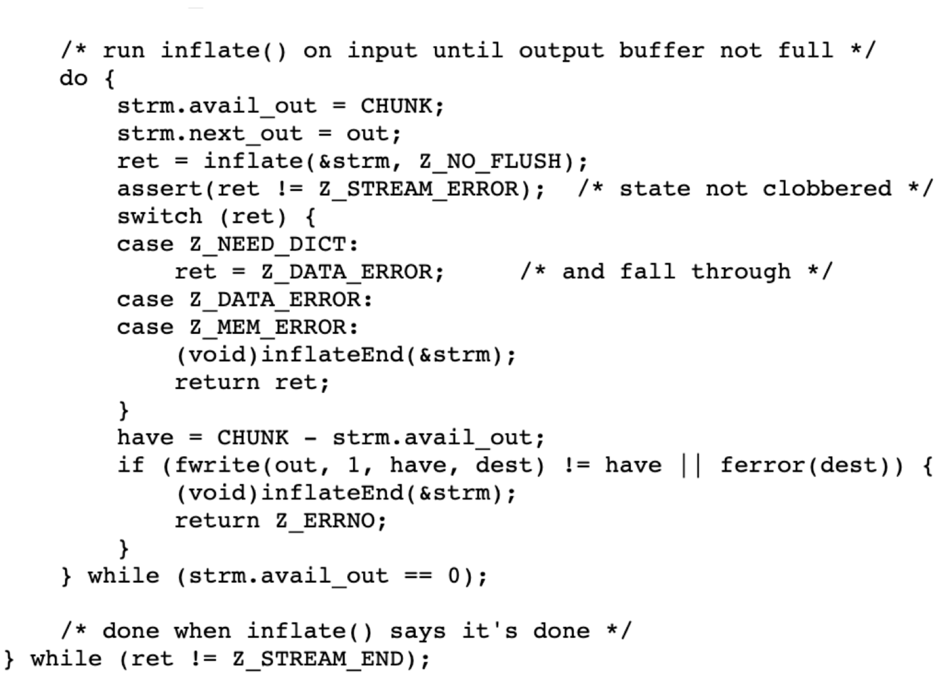
Those counters and pointers are then updated past the input data consumed and the output data written.

It is the amount of output space available that may limit how much input is consumed.

Hence the inner loop to make sure that all of the input is consumed by providing more output space each time.

Since avail_in and next_in are updated by deflate(), we don't have to mess with those between deflate() calls until it's all used up.

Inflate (Decompress)



```
    /* run inflate() on input until output buffer not full */
    do {
        strm.avail_out = CHUNK;
        strm.next_out = out;
        ret = inflate(&strm, Z_NO_FLUSH);
        assert(ret != Z_STREAM_ERROR);  /* state not clobbered */
        switch (ret) {
        case Z_NEED_DICT:
            ret = Z_DATA_ERROR;     /* and fall through */
        case Z_DATA_ERROR:
        case Z_MEM_ERROR:
            (void)inflateEnd(&strm);
            return ret;
        }
        have = CHUNK - strm.avail_out;
        if (fwrite(out, 1, have, dest) != have || ferror(dest)) {
            (void)inflateEnd(&strm);
            return Z_ERRNO;
        }
    } while (strm.avail_out == 0);

    /* done when inflate() says it's done */
} while (ret != Z_STREAM_END);
```

The outer do-loop decompresses input until inflate() indicates that it has reached the end of the compressed data and has produced all of the uncompressed output.

The inner do-loop has the same function it did in def(), which is to keep calling inflate() until has generated all of the output it can with the provided input.

The inner do-loop ends when inflate() has no more output as indicated by not filling the output buffer, just as for deflate(). In this case, we cannot assert that strm.avail_in will be zero, since the deflate stream may end before the file does.

The outer do-loop ends when inflate() reports that it has reached the end of the input *zlib* stream, has completed the decompression and integrity check, and has provided all of the output. This is indicated by the inflate() return value Z_STREAM_END. The inner loop is guaranteed to leave ret equal to Z_STREAM_END if the last chunk of the input file read contained the end of the *zlib* stream. So if the return value is not Z_STREAM_END, the loop continues to read more input.

Shut down order

If the client initiates the closing of the socket, it may not receive the last output from the shell. To ensure that no output is lost, shut-downs should be initiated from the server side:

1. an *exit(1)* command is sent to the shell, or the server closes the write pipe to the shell.
2. the shell exits, causing the server to receive an EOF on the read pipe from the shell.
3. the server collects and reports the shell's termination status.
4. the server closes the network socket to the client, and exits.
5. the client continues to process output from the server until it receives an error on the network socket from the server.
6. the client restores terminal modes and exits.

After reporting the shell's termination status and closing the network socket, the server should exit. Otherwise it would tie up the socket and prevent testing new server versions. After your test is complete, the client, server, and shell should all be gone. There should be no remaining orphan processes.

The --log option

To ensure that compression is being correctly done, we will ask you to add a **--log=***filename* option to your client. If this option is specified, all data written to or read from the server should be logged to the specified file.

Prefix each log entry with **SENT # bytes:** or **RECEIVED # bytes:** as appropriate.

(Note the colon and space between the word **bytes** and the start of the data). Each of these lines should be followed by a newline character (even if the last character of the reported string was a newline).

Before running your program in a mode that creates a log file, please use the *ulimit* command to ensure that your log file does not get too large, which could happen if you have a bug in your program. *ulimit 10000* should be sufficient, but check out the ulimit man page for further details. Failing to limit the size of log files has filled up the HSSEAS Linux servers' /tmp directories in the past, which makes the machines hard to use for everybody, including you.

Sample log format output:

```
SENT 35 bytes: sendingsendingsendingsendingsending
RECEIVED 18 bytes: receivingreceiving
```

Questions?

Do you need to save and restore both clients and servers terminal modes?

Who exits first? Client or Server?

What all errors should you handle?

Polling on server and client should happen on which file descriptors?

What events should you poll on?

What flush flag is ideal for our use case? `Z_SYNC_FLUSH` (https://www.bolet.org/~pornin/deflate-flush.html)