sockets

For a *much* better and more verbose description of sockets programming in C, please consult **Beej's Guid to Network Programming**: https://beej.us/guide/bgnet/html/

Additionally, see the following man pages: `socket(3P)`, `accept(3P)`, `bind(3P)`, `connect(3P)`, `listen(3P)`, `recv(3P)`, `send(3P)`.

I will simply try to summarize through a simple exercise, and perhaps tie in topics which we have already covered in class and assignments.

Understand though, that this is a complex topic, and not everything can/will be covered in a quick preparation document.

Sockets are a form of inter-process communication (IPC). So far we have covered signals for very simple IPC, and pipes for communication between strictly related processes. What I mean by *strictly related* is that processes which communicate via pipes must have some sort of relationship thorugh a series of parents and/or children. Sockets, on the other hand, allow for programs which are unrelated to speak with each other, and are the fundamental building block upon which things like web servers and daemons work, not to mention dozens of other classes of programs.

Now we cannot base communcation via sockets on proccess ID's, as these change every time a process is stopped and started. Also, we need to be able to communicate over the network. Therefore, to communicate with an unrelated process, you must know its IP address, as well as what port that process is listening to. For processes local to a machine, the IP address is simply `127.0.0.1` for IPv4 or `::1` for IPv6. I may use either those numbers or simply say `localhost` interchangeably. They mean the same thing.

We will just be looking only at localhost for now, but the idea extends in the same way to non-localhost addresses and ports.

So, the question remains, what really *is* a socket? Well, as far as your programs are concerned, it is just a file descriptor. Underneath, the operating system does some magical things to connect it in a way that it can communicate with other unrelated processes. This means that you can use many of the same functions on sockets that you've been using for your past programs in this course. Unlike a pipe, though, a socket is bi-directional.

It is important to mention though, before we get into the details of how to use sockets, that instead of using `read()` and `write()` on sockets, you will use `recv()` and `send()` respectively. `send()` and `recv()` work exclusively on socket file descriptors, and allow more fine-grained control than `read()` and `write()`, although you *can* use `read()` and `write()` on sockets as well if you so choose. For the purposes of this course, `write()` is equivalent to `send()` where the flags passed to `send()` is `0`. The same thing applies for `read()` and `recv()`.

Anyway, in this document we will be making a server that, when passed a string, sorts each word in the string and passes it back to the client. And, of course, we will be creating the corresponding client to connect to this server. What I have done is pasted the code for each at the bottom of this document as both are quite lengthy, as well as the makefile to build both. Then I will talk my way through first the server and then the client above all the code.

Server

The purpose of the server here is, given a string of space-separated words, to sort the words alphabetically and send them back to the client. The sorting function I have written is not the object of this document, and so I leave it to you to understand how the function `sort_string()` works. It is using many programming practices you have seen before. I have not error checked it for brevity.

Moving on to the server code. We see two new headers we have not seen before: `arpa/inet.h` and `sys/socket.h`. `arpa/inet.h` is used for handling networking, and `sys/socket.h` is used for creating sockets.

So at the top of `main()`, we see that we have a few variables we setup at first. `server_socket` will be the socket file descriptor for our server to listen on, `client_socket` is used to hold onto a file descriptor for communicating with a client, and the two `struct sockaddr_in`'s `server_addr` and `client_addr` are used similarly.

Now, while `server_socket` and `client_socket` are just file descriptors, a `struct sockaddr_in` is a bit more complex.

I have, for your convenience, taken the definitions of the struct `sockaddr_in`, and a struct inside it `in_addr` and put it here for you. I found it in `man 7 ip`.

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```

These structs are used to hold information about a target process, as identified by the family (IPv4 = `AF_INET` or IPv6 = `AF_INET6`), the port, and the IP address.

Notice these mentions of network byte order. This has to do with the order in which bytes are stored (big-endian vs. little-endian) on the host vs what is used on the network. For portability, always assume that your host byte order will not correspond to the network byte order, and wherever necessary, convert from host-to-network (`man 3 htons`) or back from network-to-host (`man 3 ntohs`).

Our next field is the address length, which we will need later on a few times, so we just store that in a variable. And finally, we create a buffer of length `BUFLEN` for storing and manipulating strings.

Now we get into the meat of things. With a call to `socket()`, we create a socket on the system that specifies IPv4 protocol, and this variable `SOCK_STREAM`. There are two kinds of connections we will commonly see: User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). `SOCK_DGRAM` specifies UDP, while `SOCK_STREAM` specifies TCP. For the difference between the two, please see some wikipedia articles, as it is beyond the scope of this document, but it is enough to say that `SOCK_STREAM` maintains a connection while the socket between two processes is maintained.

Below that, we setup the specifics for the socket. We memset it to zero everything out, then specify `AF_INET` (IPv4), `INADDR_ANY` (listen on all network interfaces), and we specify the port on which to listen. Note that for the port, we need to convert from host byte order to network byte order! So now, our `struct sockaddr_in` is populated with the proper info. Next, we need to bind the socket we created above to the port and interfaces we just specified. To do so, we will use a call to `bind()`. At this point, the server is able to accept new connections through the specified port (in this case 6006).

Next, we listen for incoming connections via a call to `listen()`. The `SOMAXCONN` option just specifies that we should allow as many clients as possible to queue up to connect with our server.

Note that in the next line, when we `printf` the port, we have to call `ntohs` to convert from network order to host order for printing.

Now we enter a while loop where our general flow will be to accept a client connection, handle the client's request, and send a response.

First, we accept a new connection. The call to `accept()` populates the `client_addr` struct, and returns the socket file descriptor for the client. At this point, we can use the client socket as any other file descriptor.

Next, we want to receive data from the client. We will use the function `recv()` to do this, which has the

same signature as `read()` except for having an additional `flags` spot at the end. We will use `0` for flags. After our call to `recv`, `buf` will be populated with `bytes_recvd` bytes, and we append a `\0` to the end of `buf` to be safe.

Next, we make our call to `sort_string`, which will sort the words within the string, placing them back in `buf`.

We will then `send` the data back to the client. Note that the signature of `send` matches up with `write`, except that it has that additional `flags` field, which we will again set to `0`. Then we will `memset` the `buf`, and `close` the client connection. It will then repeat, accepting a connection with a new client.

Client

Much of the information in the client process is the same as in the server. This information I will gloss over, so as not to repeat myself.

For the client, we have a new function, `msg_too_long` which just takes `argv` and checks that you have no more than `4095` characters, spaces included, in your argument string. This is because we will be taking input from `argv`, and wanted a reasonable limit.

Moving more quickly now, if we move down to our client code, and we move to the line making a call to `socket()`, we see that it looks the same as in the server. This time, we only need the `struct sockaddr_in` for the server, as we won't be opening a port through the client, but rather connecting to the server's socket. So, we setup the `server_addr` (which is a `struct sockaddr_in`) to work on `AF_INET` (IPv4), and port `6006`. Note that we've also specified an IP address, and stored it in `char *addr_str`.

So, we need convert the character string representing the IP address `127.0.0.1` into the integer that the system will recognize as an IP address, storing it in `ip_conversion`.

Next, we want to establish a connection with the server using the function `connect`. This establishes a TCP connection with the server. Now, we can use calls to `send` and `recv` on the socket `client_socket`.

Next, we convert all arguments given through `argv` into a space-separated string, storing it in `buf`, and `send` it to the server. We then read the sorted string back from the server.

To give the code a shot, build the code with `make`. Then, open up two terminals. In the first, enter:

```
$ ./sort_server
```

In the other terminal, enter:

```
$ ./sort_client and this is a list of strings to sort
```

Code

sort_server.c

```
#include <arpa/inet.h>
#include <errno.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define BUFLEN 4096
```

```c
#define PORT 6006

int sort_string(char *input);

int main() {
    int server_socket = -1, client_socket = -1, retval = EXIT_SUCCESS,
        bytes_recvd;
    struct sockaddr_in server_addr;
    struct sockaddr_in client_addr;
    socklen_t addrlen = sizeof(struct sockaddr_in);
    char buf[BUFLEN];

    if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "Error: Failed to create socket. %s.\n",
                strerror(errno));
        retval = EXIT_FAILURE;
        goto EXIT;
    }

    memset(&server_addr, 0, addrlen);          // Zero out structure
    server_addr.sin_family = AF_INET;          // Internet address family
    server_addr.sin_addr.s_addr = INADDR_ANY; // Internet address, 32 bits
    server_addr.sin_port = htons(PORT);        // Server port, 16 bits

    // Bind to the local address.
    if (bind(server_socket, (struct sockaddr *)&server_addr, addrlen) < 0) {
        fprintf(stderr, "Error: Failed to bind socket to port %d. %s.\n", PORT,
                strerror(errno));
        retval = EXIT_FAILURE;
        goto EXIT;
    }

    // Mark the socket so it will listen for incoming connections.
    if (listen(server_socket, SOMAXCONN) < 0) {
        fprintf(stderr,
                "Error: Failed to listen for incoming connections. %s.\n",
                strerror(errno));
        retval = EXIT_FAILURE;
        goto EXIT;
    }

    printf("Sorting server is up and running on port %d...\n",
           ntohs(server_addr.sin_port));
    while (true) {
        // Wait for the client to connect.
        if ((client_socket =
             accept(server_socket, (struct sockaddr *)&client_addr,
                    &addrlen)) < 0) {
            fprintf(stderr,
                    "Error: Failed to accept incoming connection. %s.\n",
                    strerror(errno));
            retval = EXIT_FAILURE;
            goto EXIT;
        }
```

```
        if ((bytes_recvd = recv(client_socket, buf, BUFLEN - 1, 0)) < 0) {
            fprintf(stderr,
                    "Error: Failed to receive message from client %s. %s.\n",
                    inet_ntoa(client_addr.sin_addr), strerror(errno));
            retval = EXIT_FAILURE;
            goto EXIT;
        }

        buf[bytes_recvd] = '\0';

        printf("Received following from client: %s\n", buf);

        if (sort_string(buf) == EXIT_FAILURE) {
            fprintf(stderr, "Could not execute program 'sort'. Shutting down.\n");
            retval = EXIT_FAILURE;
            goto EXIT;
        }

        printf("Sending following sorted string to client: %s\n", buf);

        if (send(client_socket, buf, BUFLEN, 0) < 0) {
            fprintf(stderr, "Error: Failed to send message to client %s. %s.\n",
                    inet_ntoa(client_addr.sin_addr), strerror(errno));
            retval = EXIT_FAILURE;
            goto EXIT;
        }
        memset(buf, 0, BUFLEN);
        close(client_socket);
    }
EXIT:
    // F_GETFD - Return the file descriptor flags.
    if (fcntl(server_socket, F_GETFD) >= 0) {
        close(server_socket);
    }

    if (fcntl(client_socket, F_GETFD) >= 0) {
        close(client_socket);
    }
    return retval;
}

/**
 * Sorts words in string, placing them in `input`. Assumed `input` is not a
 * string literal, and that strlen(input) < 4096. Returns EXIT_FAILURE and no
 * changes made to input if exec fails. Does no error checking beyond exec.
 */
int sort_string(char *input) {
    int parent2sort[2];
    int sort2parent[2];

    pipe(parent2sort);
    pipe(sort2parent);
```

```
    if (fork() == 0) {
        // sort
        close(parent2sort[1]);
        close(sort2parent[0]);

        dup2(parent2sort[0], STDIN_FILENO);
        dup2(sort2parent[1], STDOUT_FILENO);

        execlp("sort", "sort", NULL);
        fprintf(stderr, "sort failed: %s\n", strerror(errno));
        return EXIT_FAILURE;
    }


    char buf[BUFLEN];
    int bytes_read;

    close(parent2sort[0]);
    close(sort2parent[1]);

    for (int i = 0; i < (int)strlen(input); i++) {
        if (input[i] == ' ') {
            write(parent2sort[1], "\n", 1);
        } else {
            write(parent2sort[1], &input[i], 1);
        }
    }

    close(parent2sort[1]);

    bytes_read = read(sort2parent[0], buf, 4095);

    buf[bytes_read] = '\0';

    close(sort2parent[0]);

    wait(NULL);
    wait(NULL);

    strcpy(input, buf);

    for (int i = 0; i < (int)strlen(input); i++) {
        if (input[i] == '\n') {
            input[i] = ' ';
        }
    }

    return EXIT_SUCCESS;
}
```

---

sort_client.c

```c
#include <arpa/inet.h>
#include <errno.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define BUFLEN 4096
#define PORT 6006

bool msgs_too_long(int argc, char **argv) {
    int cumulative_len = 0;
    for (int i = 1; i < argc; i++) {
        cumulative_len += strlen(argv[i]);
        if (i < argc - 1) {
            cumulative_len++; // +1 for space
        }
        if (cumulative_len >= BUFLEN) {
            return true;
        }
    }
    return false;
}

int main(int argc, char **argv) {
    int client_socket = -1, retval = EXIT_SUCCESS, bytes_recvd, ip_conversion;
    struct sockaddr_in server_addr;
    socklen_t addrlen = sizeof(struct sockaddr_in);
    char buf[BUFLEN];
    char *addr_str = "127.0.0.1";

    if (argc < 2) {
        fprintf(stderr, "Error. Must provide at least 1 string to sort.\n");
        return EXIT_FAILURE;
    }

    if (msgs_too_long(argc, argv)) {
        fprintf(stderr,
                "Error: At most %d characters can be sent, including '\\0'!\n",
                BUFLEN);
        retval = EXIT_FAILURE;
        goto EXIT;
    }

    // Create a reliable, stream socket using TCP.
    if ((client_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "Error: Failed to create socket. %s.\n",
                strerror(errno));
        retval = EXIT_FAILURE;
```

```
    goto EXIT;
}

// Construct the server address structure.
memset(&server_addr, 0, addrlen);         // Zero out structure
server_addr.sin_family = AF_INET;         // Internet address family
server_addr.sin_port = htons(PORT);       // Server port, 16 bits

// Convert character string into a network address.
ip_conversion = inet_pton(AF_INET, addr_str, &server_addr.sin_addr);
if (ip_conversion == 0) {
    fprintf(stderr, "Error: Invalid IP address '%s'.\n", addr_str);
    retval = EXIT_FAILURE;
    goto EXIT;
} else if (ip_conversion < 0) {
    fprintf(stderr, "Error: Failed to convert IP address. %s.\n",
            strerror(errno));
    retval = EXIT_FAILURE;
    goto EXIT;
}

// Establish the connection to the sorting server.
if (connect(client_socket, (struct sockaddr *)&server_addr,
        sizeof(struct sockaddr_in)) < 0) {
    fprintf(stderr, "Error: Failed to connect to server. %s.\n",
            strerror(errno));
    retval = EXIT_FAILURE;
    goto EXIT;
}

memset(buf, 0, BUFLEN);
for (int i = 1; i < argc; i++) {
    if (strlen(buf) + strlen(argv[i]) + 1 >= BUFLEN) {
        break;
    }
    strncat(buf, argv[i], BUFLEN-1);
    if (i != argc - 1) {
        strncat(buf, " ", BUFLEN-1);
    }
}

printf("Sending string to server: %s\n", buf);
if (send(client_socket, buf, strlen(buf), 0) < 0) {
    fprintf(stderr, "Error: Failed to send message to server. %s.\n",
            strerror(errno));
    retval = EXIT_FAILURE;
    goto EXIT;
}

if ((bytes_recvd = recv(client_socket, buf, BUFLEN-1, 0)) < 0) {
    fprintf(stderr, "Error: Failed to receive message from server. %s.\n",
            strerror(errno));
    retval = EXIT_FAILURE;
    goto EXIT;
```

```
    }

    buf[bytes_recvd] = '\0';

    printf("Sorted string received from server: %s\n", buf);

EXIT:
    if (fcntl(client_socket, F_GETFD) >= 0) {
        close(client_socket);
    }
    return retval;
}
```

---

makefile

```
all:
    gcc -o sort_server sort_server.c -Wall -Werror -pedantic-errors
    gcc -o sort_client sort_client.c -Wall -Werror -pedantic-errors
```