FIT2100 Assignment #2 Interprocess Communication with C Programming Week 10 Semester 2 2018

Mr Daniel Kos
Admin Tutor, Faculty of IT.
Email: Daniel.Kos@monash.edu

September 14, 2018

Revision Status

```
$Id: FIT2100-Assignment-02.tex, Version 1.0 2017/10/02 12:50 Jojo $

$Id: FIT2100-Assignment-02.tex, Version 2.0 2018/09/05 13:03 Daniel $
$Id: FIT2100-Assignment-02.tex, Version 2.1 2018/09/06 21:36 Daniel $
$Id: FIT2100-Assignment-02.tex, Version 2.2 2018/09/09 15:11 Daniel $
$Id: FIT2100-Assignment-02.tex, Version 2.3 2018/09/11 13:52 Jojo $
$Id: FIT2100-Assignment-02.tex, Version 2.4 2018/09/14 13:00 Daniel $
```

Contents

1 Introduction

This second programming assignment is due on **7th October 2018 (Sunday) by 5:00pm**. It is worth **15% of the total unit marks**. A **penalty of 5% per day** will apply for late submission. Refer to the FIT2100 Unit Guide for the policy on extension or special consideration.

Note that this is **an individual assignment** and **must be your own work**. Please pay attention to Section 4.2 of this document on the University's policies on the *Academic Integrity, Plagiarism and Collusion*.

This assignment consists of **three implementation tasks**. Each task should be implemented separately as individual C programs. All the program files and any supporting documents should be compressed into one single file for submission. (The submission details are given in Section 4.)

Assessment: Each of the three tasks has an equal weightage of marks.

2 Background

2.1 What is a display server?

A big part of an operating system's job is sharing of resources among processes.

Consider your computer's display. Most laptops only have one screen, so operating environments like *Windows* provide a utility to allow multiple processes to share the display at the same time. This utility is known as a *display server*.

One program (the 'server') has exclusive access to the computer's display. Although it may appear that you have multiple process windows sharing your screen at once, this is actually one process handling the display of all information on behalf of the other applications.

Other programs ('clients' such as Calculator, Notepad, Firefox, etc.) connect to the display server via inter-process communication and send any information to be displayed on the screen. If multiple clients are running at once, the display server will display each client application in a different 'window' area of the screen.[1]

Your assignment is to build your own simple display server, displaying information from multiple clients on one screen, based on *sockets* as you have explored in the Interprocess Communication practical. Task 1 is a simple display server supporting only a single client application, building up to multiple clients in Task 3.

2.2 Interprocess Communication

Interprocess communication is a Unix facility to enable two or more distinct processes to communicate with each other within the same computer system as well as across a network system.

As presented in Practical 5 (Week 9), there are a number of mechanisms that can be deployed to support interprocess communication as a **client-server** model. Under the setting of this model, one process is referred to as the *server* whose responsibility is to fulfill or satisfy the requests put forward to it by other processes, referred to as *clients*. In this assignment, we will focus on *sockets*.

[1] Prior to the development of operating environments like Windows 1.0 and Macintosh System 1 during the 1980s, only one application could use the screen at a time. In modern versions of Windows, the display server is known as *GDI+* and in modern versions of macOS, the display server component is known as *Apple Quartz*. In many Linux environments, the display server is known simply as *X*.

The server process must already be running prior to starting a client process, or the client will have nothing to connect to. It can be helpful to run the server and client(s) inside different terminal windows.

2.3 How to access the Virtual Terminals in Linux

While using the regular Linux desktop environment in your VM, you have only been using one of the *virtual terminals* (*VTs*) that Linux provides.

You can switch to another virtual terminal to run commands if your desktop environment crashes, or you can run two different desktop environments at the same time.

Use your VirtualBox *host key* (this is often the RightCtrl key on your keyboard, but can be changed in your VirtualBox settings), together with the Function keys (F1 to F8) at the top of your keyboard, to switch between your virtual terminals. See Table 1 and the notes below — your system may be different.

Table 1: Default key combinations for switching between different *VTs* in your VM environment.

Press...*	for...	You should see...
RightCtrl+F1	VT 1	Startup messages (if any), or a login prompt
RightCtrl+F2	VT 2	TTY login prompt for starting a text-only session here
RightCtrl+F3	VT 3	TTY login prompt for starting a text-only session here
RightCtrl+F4	VT 4	TTY login prompt for starting a text-only session here
RightCtrl+F5	VT 5	TTY login prompt for starting a text-only session here
RightCtrl+F6	VT 6	TTY login prompt for starting a text-only session here
RightCtrl+F7	VT 7	***** Your regular desktop environment *****
RightCtrl+F8	VT 8	Empty terminal for running a second desktop environment

***Notes:**
- On some laptop computers, you will need to hold down the [Fn] key as well, in order to access the F1-F8 keys on your keyboard.
- On a MacBook or other Apple device, the required key combination is Command+Fn+... instead of the key mentioned above.
- If you have changed your VirtualBox host key, use the key you have selected instead of the key mentioned above.
- In some cases, you might need to allow VirtualBox to 'capture' your keyboard.
- If you use Linux natively (without a VM), use Ctrl+Alt instead of RightCtrl, but note that your particular Linux distribution may manage the VTs in a different way.

In this assignment, we will be creating a (very) simple desktop environment, which will run in the 8th 'VT slot' (VT 8), but you will control it by running commands in your regular desktop environment (VT 7).

Please play around with switching between the VTs (try logging in to the text-only VTs for an old-school Unix experience), and switching back to your regular desktop environment until you get comfortable with this feature.

3 Display Server

3.1 *Task 1:* Microsoft Window?

In this task, you will implement a display server that only accepts a connection from a single client process.

To create our display server, we need to begin with a socket server program. We will also need a client application to test our server with. The server program should create a new socket and name it with a Unix domain name (pathname), such that the client program will know which socket to connect to for communication. The server will then listen to the incoming connection request from the client program.

The client program, on the other hand, should attempt to connect to the known socket setup by the server program. Once successfully connected, the client should allow the user to type input into the TTY terminal, and send the user's terminal input to the server through that socket, to be displayed on the screen by the server.

You may adapt your `socketserver.c` and `socketclient.c` code from Practical #5 for this.

Note: For Task 1, please name your main server source file as `task1server123456789.c`, and the source file for the client program as `client123456789.c`, where `123456789` is your student ID number. (The same client program will be used for all three tasks of this assignment.)

3.1.1 Requirements

You should modify your completed socket client and server code from the practical in order to satisfy the following requirements:

1. The socket file should be created as `/tmp/a2-123456789.socket` where `123456789` is your student ID.[2] You should also modify your client to connect to the socket at this location.

2. When the client connects to the server, it should first `send` its *name* to the server followed by a newline. After this, it should take all the user's input entered into the terminal and send this to the server process.

[2]The /tmp/ directory is a useful place to store temporary files, since it is automatically cleaned out each time the system is rebooted.

3. Modify the server program to use ioL so that instead of printing plain TTY output, it prints characters received from the client into a `<box>` element which should first be printed into the ioL console. The box should be created when the client's connection is accepted, and the first line of characters received (i.e. the name of the client) should be used to decorate the box with the program title. Your server program should be run using the `iol` command. An example is shown in Figure 1.

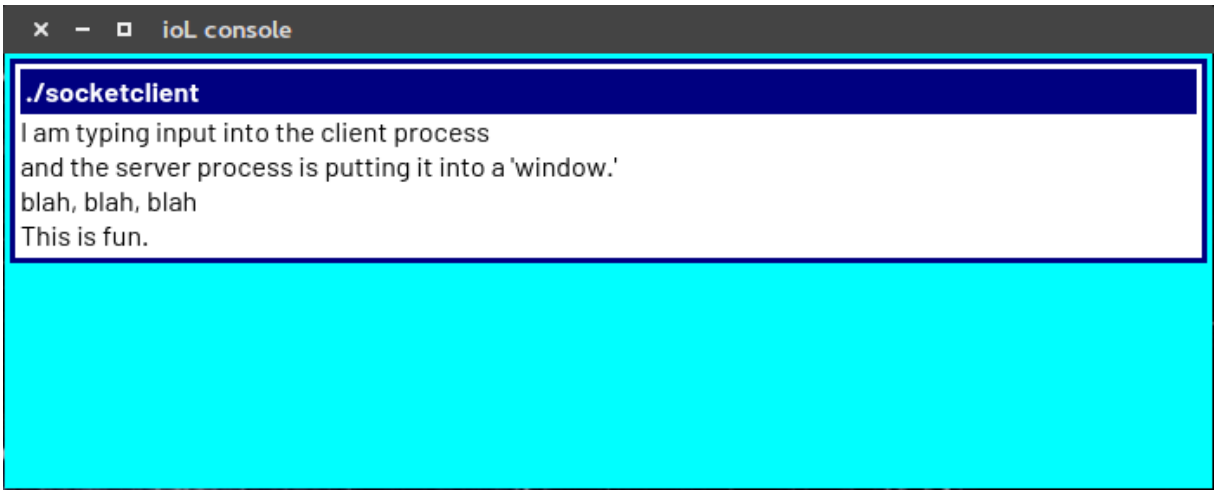

Figure 1: Sample server output when the user runs a client named `socketclient` (which connects to the server) and the user enters text into the client process. The 'window' was created using a box with `width=100,{%}` in order to fill the screen horizontally. *Note: you do not need to display the output in exactly this way as long as you visually differentiate between the title and the window contents in some way. Also note that you are not required to print each line of client input on a separate line on the server.* In Task 3, we will allow the user to create multiple windows.

3.1.2 Running your server as a stand-alone desktop environment

There is a bash script provided in Appendix A near the end of this assignment paper. This script will run your display server in the 8th virtual terminal slot (VT 8), as explained earlier in the background information.

You should modify the script to use the command needed to run your socket server program, and try running your display server as a separate desktop environment. You will need to switch back to VT 7 (your regular desktop) in order to run your client and enter input into your client. (Pressing Ctrl+C in the TTY window you ran the script from should be able to terminate your desktop environment for now. You will implement a better solution for this in Task 2.)

You should try running your desktop environment in this way for each of the tasks in order to see what a real user's experience of your desktop environment would be like.

3.2 *Task 2:* Add a single quit button

It would be good to provide a way for the user to shut down their desktop environment once they are done.

In this task, you are required to extend your display server from Task 1 by adding a single button to the screen for the user to click when they wish to shut down your display server.

Note: For your improved Task 2 server, please name your main server source file as `task2server123456789.c` where `123456789` is your student ID number.

You can use the following code to create the button, or implement it in your own way:

```
1 printf("<button {Shutdown} onclick=<putln {Q}>></n>\n");
```

This button sends the input `"Q"` (followed by a newline) from the ioL console to your display server when clicked.

Your task is to make the display server quit when the button is clicked, by reading this `Q` command from your server program's standard input and exiting the program when this happens.

But you must do so *without* disabling the functionality from Task 1.

3.2.1 Note

This small addition to the server program might sound easy but there is a catch. While we are reading input from the client, our program will continually block until there is new input available from the client, but the quit instruction doesn't come from the client, it comes from our standard input. How do we know which source of input to read from? How can we wait for two different sources of input at once?

The `select()` function in C is useful here. This blocking function allows you to specify a list of possible input sources you want to read from, and will put your program to sleep until at

least one input source is ready, and allows you to know which input source is ready *now* before you attempt to read from it. See Appendix B for an overview of how this function works.

3.3 *Task 3:* Multiple clients at a time

Now extend your display server further to allow the user to run up to 5 client instances at a time. Your display server should create a new <box> element on the screen for each new connection and ensure the input received from each client is pushed into its respective box.[3]

Once again, the `select()` function is useful for dealing with inputs coming from multiple sources. See Appendix B for further details.

You are not required to clean up your windows (boxes) from clients that have terminated, but you should ensure that the display server rejects[4] any further connections after the 5th client process.

Note: For your improved Task 3 server, please name your main server source file as `task3server123456789.c` where 123456789 is your student ID number.

3.3.1 Limitations of your desktop environment

Your desktop environment only displays output but does not allow the user to interact with a client window or pass information back to the client program.

Think about how you might extend your program to allow the client programs to receive input from the user's interaction via your display server. Can you think of any other limitations of your display server? Discuss briefly (1-2 paragraphs) in your program documentation.

[3] You will need to create each <box> element with a different label in order to reference the correct box for each client after creation.

[4] The aim here is to make your implementation simpler by only allowing a fixed maximum number of client windows. Unix sockets don't provide a 'standard' way to 'reject' a connection, but accepting and then immediately closing any unwanted connections is one common approach. Ignoring any new connections once the server is 'full' — by no longer checking the socket in the `select()` call once five clients have connected — is another possible approach. You can handle unwanted extra client connections whichever way you choose; any approach that does not result in undefined behaviour is fine for the purposes of this assignment.

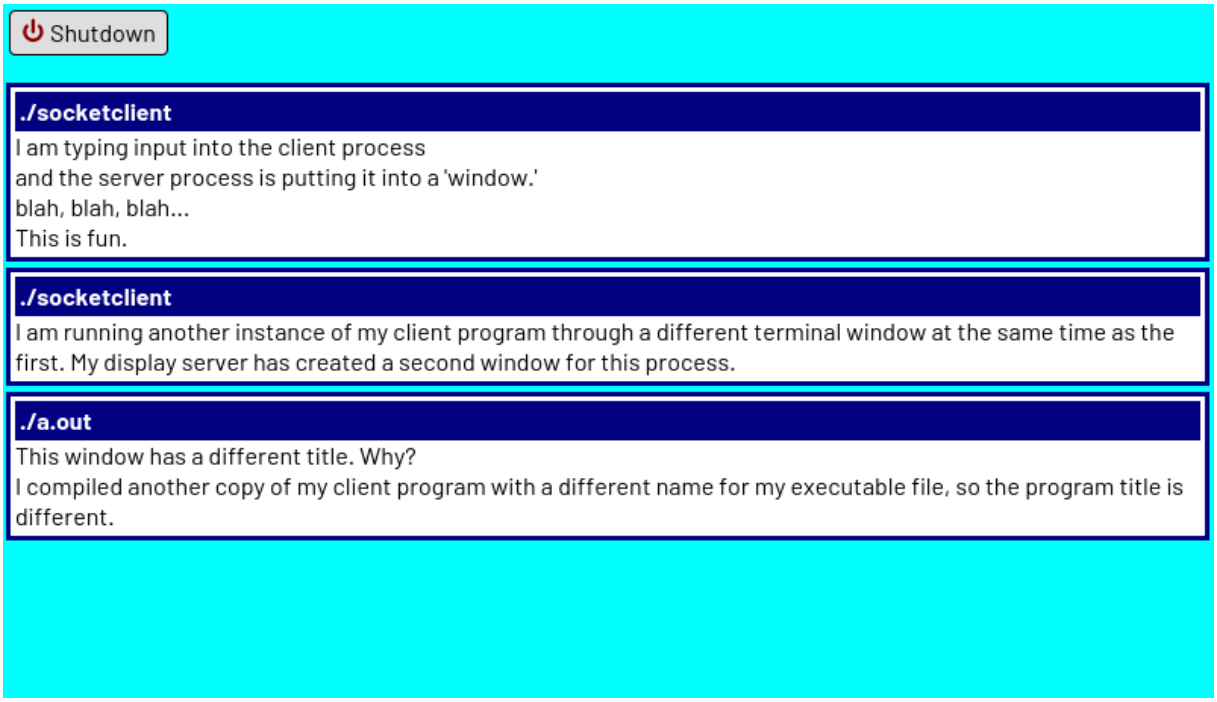

Figure 2: Sample server output for Task 3. (Also showing a quit/shutdown button which was added in Task 2.) Here, the user might be running three instances of the client program (in three different terminal windows), and has typed different stuff into each client. The 'windows' are boxes with `width=100,{%}` and a newline between each box on the screen, for a vertical-tiled appearance (you may experiment with laying out your windows in different ways if you wish to do so). Each box was created with a different name (label) to make it possible to push characters into the correct box depending on which client sent them.

3.4 Important Notes

Commenting your code is essential as part of the assessment criteria (refer to Section 3.5). You should also include comments at the beginning of your program file, which specify your name, your Student ID, the start date and the last modified date of the program, as well as with a high-level description of the program. In-line comments within the program are also part of the required documentation.

3.5 Marking Criteria

The assessment of this assignment will be based on the following marking criteria. The same marking criteria will be applied on each of the implementation tasks (Task 1 to Task 3):

- 50% for working program (for all three tasks);
- 20% for code architecture (algorithms, data structures, use of procedures and libraries, etc. in your implementations of the three tasks);
- 10% for coding style (clarity in variable names, function names, blocks of code clearly indented, etc.);
- 20% for documentation (both program comments and user documentation).

4 Submission

There will be NO hard copy submission required for this assignment. You are required to submit your assignment as a `.tar.gz` file named with your Student ID. For example, if your Student ID is 12345678, you would submit a zipped file named `12345678_A2.tar.gz`. Note that marks will be deducted if this requirement is not strictly complied with.

Your submission is via the assignment submission link on the FIT2100 Moodle site by the deadline specified in Section 1, i.e. **7th October 2018 (Sunday) by 5:00pm**.

4.1 Deliverables

Your submission should contain the following documents:

- A completed assignment submission statement for online submission via the FIT2100 Moodle site.
- An user documentation (not more than 3 pages) in plain text (`.txt`) format with clear and complete instructions on how to run your programs, as well as any references you have used, and any limitations of your implementation as appropriate.
- Electronic copies of ALL your files that are needed to run your programs.

Marks will deducted for any of these requirements that are not strictly complied with.

Note that your programs must at least run on the Linux VM environment provided for this unit. Any submission that does not compile or run accordingly will receive no marks.

4.2 Academic Integrity: Plagiarism and Collusion

Plagiarism means to take and use another person's ideas and or manner of expressing them and to pass them off as your own by failing to give appropriate acknowledgement. This includes materials sourced from the Internet, staff, other students, and from published and unpublished works.

Collusion means unauthorised collaboration on assessable work (written, oral, or practical) with other people. This occurs when you present group work as your own or as the work of

another person. Collusion may be with another Monash student or with people or students external to the University. This applies to work assessed by Monash or another university.

It is your responsibility to make yourself familiar with the University's policies and procedures in the event of suspected breaches of academic integrity. (Note: Students will be asked to attend an interview should such a situation is detected.)

The University's policies are available at: `http://www.monash.edu/students/academic/policies/academic-integrity`

5 Appendix A: bash script for standalone running

The following bash script enables you to run an ioL program (such as your display server) as a standalone desktop environment. **You are not expected to understand the code in this script**, and should only modify a single line as mentioned below in order to suit the command needed to run your program. A copy of this script is available from the FIT2100 Moodle site.

Note:

- Since this script requires root privileges in order to set up a new virtual terminal, you will need to run via the sudo command. Example: `sudo ./runstandalone.sh`
- You will need to set execute permission in order to execute the script.
- You must modify the value of `COMMAND="..."` on line 28 to specify the command used to run your display server.
- It is safe to ignore any messages about unexpected signals when running this script.
- Use the keyboard combinations discussed in Section 2.3 to switch back to your regular desktop environment.

```
1  #!/bin/bash
2  #BASH SCRIPT FOR RUNNING SOCKET SERVER PROGRAM IN A STANDALONE ENVIRONMENT
3
4  #please run as root user
5  if [[ $EUID -ne 0 ]]; then
6      echo "This script must be run using sudo"
7      exit 1
8  fi
9
10 killall -u root hiped 2> /dev/null; rm /run/hipe.socket 2> /dev/null;
11 xinit /usr/local/sbin/hiped --css /etc/hipe-files/hipe.css --fill \
12     -- :1 vt8 &
13 BGPID=$!
14
15 until ls /run/hipe.socket 2> /dev/null > /dev/null; do
16     echo "Waiting for back-end to come up..."
17     sleep 0.5
18 done
19 chmod 766 /run/hipe.socket
20 sleep 0.5
21
22 echo "*** PLEASE USE [host key]+F7/F8 TO SWITCH BETWEEN CONSOLES ***"
23
24 #########
25 # Please modify the following value to assign the command needed to run
```

```
# your socket server program:
#
COMMAND="iol -- ./put-the-name-of-your-program-here"

sudo -u student bash -c "\
     export HIPE_KEYFILE=/run/hipe.hostkey;\
     export HIPE_SOCKET=/run/hipe.socket;\
     $COMMAND"

echo "SHUTTING DOWN..."
kill -2 $BGPID
```

6 Appendix B: Waiting for multiple input sources with select()

Many resources in Linux that can be read from or written to are allocated a unique *file descriptor* — an integer value used to reference that resource within your program. The standard input (e.g. input from console) is automatically allocated file descriptor 0 when your program starts. The socket you create in Task 1 (used to listen for new connection requests) is allocated another file descriptor value, and there is an additional descriptor allocated for each new client connection.

Many read operations (including `scanf()`) in Linux are *blocking* operations: if there is no input available yet, the OS puts your program to sleep until the input arrives. This can be problematic if you don't know where the next input will come from. For example, if your program is blocked on an instruction waiting for a new client connection that might never come, you cannot receive console input or do anything else in the meantime.

One common solution is to implement multi-threading — each thread can monitor a different input source. However, a simpler solution which avoids multi-threading is the `select()` function.

The `select()` function monitors a set of different file descriptors you provide, and blocks your program's execution until one or more of the resources changes state. You can then check exactly which resource contains data ready for *immediate* reading, and safely read from it without your program blocking (since data is already available so there is no need to wait).

The special `fd_set` data type is used to represent a set of file descriptors to be monitored by `select()`. The associated macros `FD_ZERO()`, `FD_SET()` and `FD_ISSET()` are used to manipulate variables of this type. Any file descriptors still in the set after `select()` returns

are generally the ones that have data ready for reading.

Pseudocode for the structure of a socket server using the `select` function to take input from multiple sources is shown in an *NS diagram* in Figure 3. You may also need to consult documented usage information for this function, and should cite any useful resources in your program documentation.

Here is a code sample showing usage of the `select()` function in a situation where you have two different file descriptors that you are waiting for input from at the same time:

```
//Preconditions:
// * my_file_descriptor is an int variable for a resource that we have
//   opened earlier.
// * the required C library headers have been included according
//   to the man page for select.

//...

fd_set set_of_resources;
FD_ZERO(&set_of_resources); //need to clear the set every time before use.

FD_SET(my_file_descriptor, &set_of_resources);
//add the file descriptor we opened earlier (not shown - see above comment)
    .

FD_SET(0, &set_of_resources);
//also add the file descriptor for standard input to the set of resources
//we wish to monitor.

//(If we had more file descriptors we wanted to check for input from,
//we would add them here.)

int num_ready = 0;
while(num_ready == 0) { //Not strictly needed for this use case.
    //(What kind of scenario would cause select() to return 0 though?)

    num_ready = select(my_file_descriptor+1, &set_of_resources,
        NULL, //I'm not interested in write resources
        NULL, //I'm not interested in resource exceptions
        NULL  //I'm not interested in giving up after a time expires
    );
}
if(num_ready < 0) exit(1); //some kind of error

//at this point, select_result must be the number of resources that
//are ready to be read...

while(num_ready > 0) { //number of resources with input waiting
    if(FD_ISSET(0, &set_of_resources) { //descriptor 0 is stdin...
        //(Read characters from standard input here.)
    } else if(FD_ISSET(my_file_descriptor, &set_of_resources)) {
```

```
41          //our other open resource is ready for reading.
42          //(Read characters from it here.)
43      }
44
45      num_ready--;
46  }
47
48  //...
```

The man page for `select` is another useful resource and provides an example of simple usage.[5]

Note: The `select()` function is quite versatile and takes in a number of arguments that are not necessarily useful for this assignment. For all but not the first two arguments, you can pass a `NULL` pointer if you do not require them.

[5] Note that the man page talks about 'synchronous multiplexing'. In this context, *synchronous* simply means one input source will be read at a time, and *multiplexing* means we are reading from different file descriptors in an interleaved way (reading characters from different input sources as new input become available).

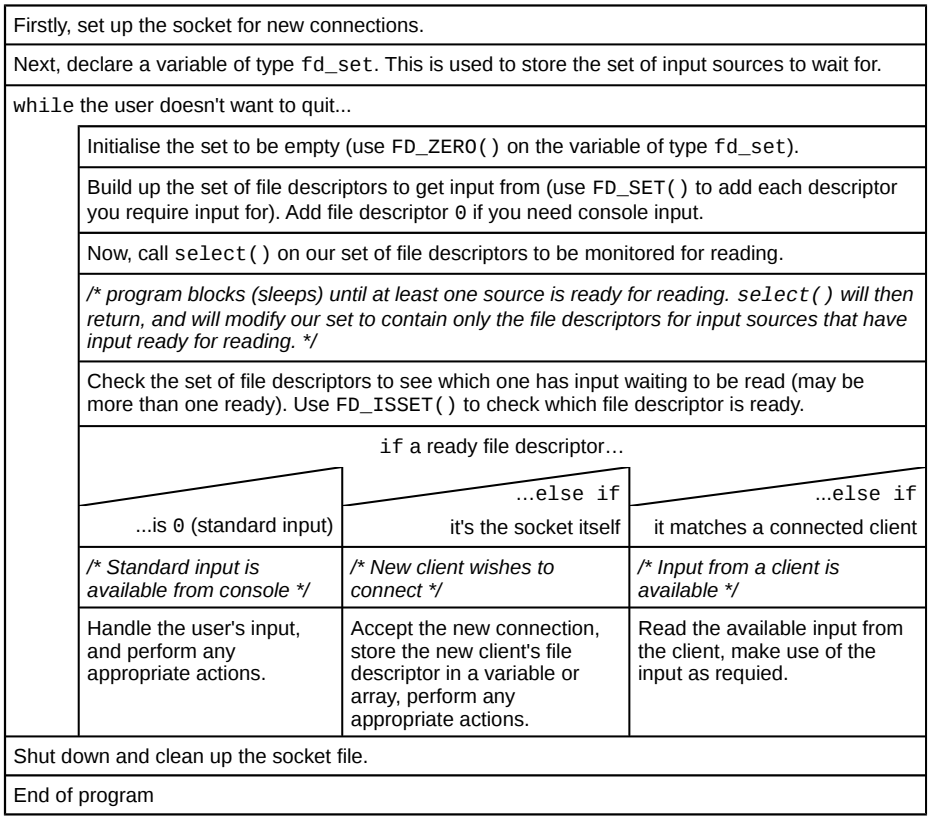

Figure 3: Rough overview of program structure for a socket server that makes use of the `select()` function to wait for input from multiple sources. After `select()` returns, the program can make a decision on what to do depending on which file descriptor has input available.