NWEN 241 Assignment 3

(Weeks 6–7 Topics)

Release Date: **22 April 2024**

Submission Deadline: **13 May 2024, 23:59**

In this assignment, you will be asked to implement a network server program. For this, you will need to complete a set of tasks to establish a client-server communication. The specifications and guidelines of the tasks are given below. Your submission must be made in files named `server.c` and `server2.c`.

You must submit the required files to the Assessment System (`https://apps.ecs.vuw.ac.nz/submit/NWEN241/Assignment_3`). Any assignment submitted up to 24 hours after the deadline will be penalised by 20%, and any assignment submitted between 24 and 48 hours after the deadline will be penalised by 40%. Any assignment submitted 48 hours or more after the deadline will not be marked and will get 0 marks.

Important: The Assessment System is configured **not to accept submissions that do not compile.** So please test that your code compiles using a C compiler before submitting it.

Full marks is 100. The following table shows the overall marks distribution:

Criteria	**Marks**	**Expectations for Full Marks**
Compilation	5	Compiles without warnings
Comments	10	Sufficient and appropriate comments
Code Quality	15	Efficient code and use of consistent coding style
Correctness	70	Handles all test cases correctly (see marks distribution below)
Total	**100**	

For the **Correctness** criteria, the following table shows the marks distribution over the different task types:

Task Type	**Marks**
Core	45
Completion	15
Challenge	10
Total	**70**

Introduction

This assignment tests whether you can apply the conceptual knowledge you have learned in Weeks 6–7 to solve practical programming tasks. You must only use the Standard C Library to perform the tasks in this part.

You will implement a network server program, a program that essentially waits for TCP connections from clients, and responds to the messages sent by the client. Unlike the previous two assignments, you will be writing a complete C program with a `main()` function.

Sample skeleton codes and text files are provided under the `files` directory in the archive that contains this file. Use the skeleton files to perform the tasks and testing.

Commenting

You should provide appropriate comments to make your source code readable. If your code does not work and there are no comments, you may lose all the marks.

Coding Style Code quality refers to (*i*) the use of efficient coding techniques and (*ii*) use of consistent coding style or standard.

When writing your source code, strive for *coding efficiency* which covers the following aspects: (*i*) elimination of unessential operations and variables; (*ii*) creation of functions to contain blocks of code that are used repeatedly; (*iii*) minimization of loop iteration; and (*iv*) avoiding the use of global variables (i.e. variables that are declared outside any function.);(*v*) use built-in functions where appropriate.

In addition, you should follow a consistent coding style when writing your source code. Coding style (aka coding standard) refers to the use of appropriate indentation, proper placement of braces, proper formatting of control constructs, and many others. Following a particular coding style consistently will make your source code more readable. There are many coding standards available (search "C coding style"), but we suggest you consult the *lightweight* Linux kernel coding style (see `https://www.kernel.org/doc/html/v4.10/process/coding-style.html`). The relevant sections are Sections 1, 2, 3, 4, 6, and 8. Note that you do not have to follow every recommendation you can find in a coding style document, you just have to apply that style consistently.

Server Program Specifications

1. The program should accept one command line argument. Excluding the program name, the argument should specify the port number that the program will use. The program should immediately terminate with a return value of -1 if the argument is not specified. The program should also immediately terminate with a return value of -1 if the specified port number is less than 1024.
2. Initiate the required socket operations to create and bind a socket.
3. Listen for TCP clients at the port specified in the command line. (During testing, you may need to use a "random" port number to avoid conflicts with other students running tests on the same server.)
4. When a client successfully establishes a connection with the server, send a `HELLO` message back to the client .
5. Wait for a message from the client.
6. Perform the following based on the message received from the client:
   (a) If the message received is `BYE` (case-insensitive), close the connection to the client and go back to step 3.
   (b) If the message received is `GET` (case-insensitive) followed by a file name (example: `GET file.txt`), open the file as text file for reading. The file should be located in the same directory as the server program. If the opening is successful, send the following back to the client:
   ```
   SERVER 200 OK\n
   \n
   (Contents of file)\n
   \n
   \n
   ```

   You must follow this format strictly. Note that there is a single empty line (a newline) after `SERVER 200 OK` and two empty lines (two newlines) after the contents of the file.

   Note: You do not need to implement any functionality to verify the validity of the file names. The file names are expected to (a) follow Linux file naming conventions and (b) not include any white space characters.

   If the opening is not successful, send the following back to the client:

   ```
   SERVER 404 Not Found\n
   ```

For any other error encountered (e.g., no file name is specified in the `GET` message), send the following back to the client:

```
SERVER 500 Get Error\n
```

(c) If the message received is `PUT` (case-insensitive) followed by a file name (example: `PUT file.txt`), open the file as text file for writing (if the file exists, existing contents must be emptied). The file should be located in the same directory as the server program. Write every message received from the client to the file. Upon receipt of two consecutive empty lines (just newlines), close the file. The last two consecutive empty lines should also be written into the file. After successfully closing the file, send the following back to the client:

```
SERVER 201 Created\n
```

For any other error encountered (e.g., failure to open file), send the following back to the client:

```
SERVER 501 Put Error\n
```

For other messages, send the following back to the client: `SERVER 502 Command Error\n`

Testing

As you are using the Linux `socket()` system call, you will need to test your program in Linux. You do not need to write a client program to test your server program. You can use the Linux program `nc` to receive and send messages to your server program. To do this:

1. Open a terminal. Compile and run your program.
2. Open another terminal. Run `nc localhost portnum` (where `portnum` is the port number that your server is using) to establish a connection with your server program. Once `nc` is connected to the server program: whatever you type in this terminal will be sent to the server program, and whatever is sent by the server program will shown in this terminal.

If you are performing the test remotely, you will need to open 2 ssh/putty connections to the same server machine. In one ssh/putty terminal, you compile and run your program. In the other ssh/putty terminal, you run `nc localhost portnum` to establish a connection with your server program.

A video will be posted in Nuku (by Friday, 26 April) to clearly demonstrate the testing process.

Task 1.

Core [45 Marks]

Implement the server program specifications 1 – 5 above inside the file `server.c`.

Task 2.

Completion [15 Marks]

Implement the server program specifications 6(a) - 6(c) above inside the file `server.c`.

Task 3.

Challenge [10 Marks]

One of the drawbacks of the server program specified in the **Program Specifications** above is that when a client is connected with the server, no other client can connect to the server.

Provide an enhancement of the server program by using the `fork()` system call as follows: At step 4 of the Program Specifications above, when a client successfully establishes connection, fork a child process. Upon successful fork, the parent process should go back to listen for TCP clients at the same port, whereas, the child process should perform the following:

1. Send a message to the client with contents `HELLO`.
2. Wait for a message from the client.
3. Perform the following based on the received message from the client:
   (a) If the message is `BYE` (case-insensitive), close the connection to the client and exit the process.
   (b) If the message is `GET` (case-insensitive) followed by a file name (example: `GET file.txt`), open the file as text file for reading. The file should be located in the same directory as the server program. Go back to Step 2. Request handling will be similar as defined above for a single client.

If the fork is not successful, the process should perform steps 4 and 5 (in the original Program Specifications) i.e. it should send a `HELLO` message back to the client and wait for a message from the client; however it is not required to support the `BYE`, `GET` and `PUT` operations.

Save the enhanced server program as `server2.c`.