CSE 3100: Exam 2 Date: 11/03/2023
Instructor: Kriti Bhargava Duration: 1 hour 50 minutes

Instructions:

This is a 110-minute exam. You may use your notes, lecture material, textbook, labs, and problem sets. You may also use blank sheets of paper to scribble. You cannot use Google or any other online sources. You may not use material from any other course.

Cheating on the midterm will lead to an immediate 'F'.

There are two questions on this exam. The grade breakdown is given below.

Question	Points
Q1	40
Q2	60
Total	**100**

Read the description carefully and attentively. Think about the problem and what is being asked before diving into the code. The solutions are short. If you find yourself writing a substantial amount of code, you are most likely making this too hard on yourself.

Do not hard code values to match the output. You will not earn any grade if you do this.

All the best! 😊

CSE 3100: Exam 2 Date: 11/03/2023
Instructor: Kriti Bhargava Duration: 1 hour 50 minutes

Q1. Pipe-prefixsum (40 points)

In this problem, we use two child processes and pipes to compute the prefix sum of n integers, generated in the parent process. The array's prefix sum at index i is the sum of all elements from index 0 to i (inclusive). The starter code is given in pipe-prefixsum.c.

Specifically, the parent process takes n integers from the user, where n must be a **positive even number**, and saves them in the array `arr`. Then it creates two child processes so that child process 1 computes the prefix sum of the first half of the array `arr` and child process 2 computes the prefix sum of the second half.

Both child processes just call `computePrefixSum()` (given to you) to compute the prefix sum and store it in the array `prefixSum`. Then they send the `prefixSum` of the computed indices to the parent process through pipes.

The parent process receives the prefix sum from both child processes through pipes and saves them in 2 arrays `childPrefixSum1` and `childPrefixSum2`. Then it combines the two prefix sums to compute the prefix sum of all n numbers and stores it in the array `prefixSum`. The code to combine the prefix sums has been given to you.

For instance, given the array [1, 2, 3, 4, 5, 6, 7, 8], the computation would proceed as follows:

- *Divide the array into two halves: [1, 2, 3, 4] and [5, 6, 7, 8].*
- *Parallel Prefix Sum: Child 1 computes [1, 3, 6, 10] and child 2 computes [5, 11, 18, 26].*
- *Merge: The parent combines the two results. Final result [1, 3, 6, 10, 15, 21, 28, 36].*

You need to complete the `main()` function. Search TODO to find the location. Read the comments and code carefully. Note that the file descriptors that are not needed in a process should be closed. Otherwise, some processes may not terminate properly.

Below are some sample sessions of running the program. The program takes as input the number of integers, `n`, followed by the list of n numbers. The program first prints the array of integers and then prints the prefix sum. ***Remember, n is an even value for convenience*.**

```
[krb20002@krb20002-vm Exam2-1]$ cc pipe-prefixsum.c -o pipe-prefixsum
[krb20002@krb20002-vm Exam2-1]$ ./pipe-prefixsum 6
Please input 6 integers
1 2 3 4 5 6
Original Array: 1 2 3 4 5 6
Prefix Sum Array: 1 3 6 10 15 21
[krb20002@krb20002-vm Exam2-1]$ ./pipe-prefixsum 10
Please input 10 integers
1 2 3 4 5 6 7 8 9 10
Original Array: 1 2 3 4 5 6 7 8 9 10
Prefix Sum Array: 1 3 6 10 15 21 28 36 45 55
[krb20002@krb20002-vm Exam2-1]$
```

Q2. Weather station (60 points)

In this question, you will implement a simple weather station using a pair of programs – weather station server (`server.c`) and weather station client (`client.c`). The pair follow a well-known pattern.

Firstly, the server program can be executed as

```
./server <port>
```

For instance,

```
./server 9091
```

would start the server and wait for inbound "let's talk" requests from clients on port 9091 (for the above example). When an inbound request arrives from a client, the server adds the socket of the client to an array of sockets to listen to and then immediately goes back to accept. Assume that our server can only support a maximum of 10 clients. What the server ***does not do*** is fork a child process. Instead, the server handles all the communications directly thanks to the handy `select` API. Indeed, given `n` client sockets and the server socket, the server simply needs to listen to all of them and when one socket has data ready to read, it can handle that socket.

- If it is the server socket, it is yet another client trying to connect. As mentioned above, the server adds the socket of the client to an array of sockets to talk with and then immediately goes back to accept. At most, it communicates to 10 clients at a time.
- If it is a client socket, then it can read the message from that client. If the message is "GET_WEATHER", server updates the variable `current_weather` by incrementing temperature, humidity, and pressure by 1, 2, and 3 respectively. It then sends the updated weather to all clients. If the message is "EXIT", the server closes all sockets and terminates. It is a bit rude to terminate at the behest of a single user, but it is simple to implement! If the message received is any other string, the server sends the string "Invalid command" to the client who sent the message.

When a client wishes to talk to a server, one can execute a client from a terminal with:

```
./client <port>
```

For instance,

```
./client 9091
```

would connect to a server running on localhost on port 9091. A client simply reads messages from the keyboard and sends them to the server. If the message is EXIT, the client terminates after sending the string to the server. When a response comes from the server, it displays the

response on its standard output. If the response from the server consists of 0 bytes, the client closes its socket and terminates.

To test your program, you should use 3 or more terminals as shown below. Recall that this should work with up to 10 clients. Below is a sample session of running the application with 2 clients. The clients connect to the server. The first client sends the GET_WEATHER command to the server. Both clients display the response. The first client then sends the string “qwerty” to the server and receives the response Invalid command. The second client then sends the GET_WEATHER request to the server. Both clients receive the process. This is repeated by the first client. Finally, the EXIT command from the second client terminates all processes.

```
[krb20002@krb20002-vm Exam2-1]$ make
cc -g    -c -o server.o server.c
cc -g   -o server server.o
cc -g    -c -o client.o client.c
cc -g   -o client client.o
[krb20002@krb20002-vm Exam2-1]$ ./server 9091
We accepted a socket: 4
We accepted a socket: 5
[krb20002@krb20002-vm Exam2-1]$
```

```
[krb20002@krb20002-vm Exam2-1]$ ./client 9091
GET_WEATHER
Server Response: Temperature: 1.00, Humidity: 2.00, Pressure:
3.00
qwerty
Server Response: Invalid command
Server Response: Temperature: 2.00, Humidity: 4.00, Pressure:
6.00
GET_WEATHER
Server Response: Temperature: 3.00, Humidity: 6.00, Pressure:
9.00
[krb20002@krb20002-vm Exam2-1]$
```

```
[krb20002@krb20002-vm Exam2-1]$ ./client 9091
Server Response: Temperature: 1.00, Humidity: 2.00, P
ressure: 3.00
GET_WEATHER
Server Response: Temperature: 2.00, Humidity: 4.00, P
ressure: 6.00
Server Response: Temperature: 3.00, Humidity: 6.00, P
ressure: 9.00
EXIT
[krb20002@krb20002-vm Exam2-1]$
```