

# COSC 3360 - 23466 - Operating Systems / COSC 6310 - 25945 - Fundamentals of Operating Systems


[Dashboard](#) / [My courses](#) / [COSC3360F202323466](#) / [PROGRAMMING ASSIGNMENTS](#) / [Programming Assignment 2](#)

 [Description](#)

 [Submission](#)

 [Edit](#)

 [Submission view](#)

 **Available from:** Thursday, 28 September 2023, 12:00 AM

 **Due date:** Saturday, 28 October 2023, 11:59 PM

 **Requested files:** client.cpp, server.cpp ( [Download](#))

**Type of work:**  Individual work

**This programming assignment closes at 11:58:59 PM on 10/28/2023.**

**Similarity Threshold:** 90%

## Objective:

This assignment will introduce you to interprocess communication mechanisms in UNIX using sockets.

## Problem:

You must write two programs to implement a distributed version of the multithreaded incremental entropy algorithm you created for [programming assignment 1](#).

These programs are:

### The server program:

The user will execute this program using the following syntax:

```
./exec_filename port_no
```

where exec\_filename is the name of your executable file and port\_no is the port number to create the socket. The port number will be available to the server program as a command-line argument.

The server program does not receive any information from STDIN and does not print any messages to STDOUT.

The server program executes the following task:

- Receive multiple requests from the client program using sockets. Therefore, the server program creates a child process per request to handle these requests simultaneously. For this reason, the parent process must handle zombie processes by implementing the fireman() function call (unless you can determine the number of requests the server program receives from the client program).

Each child process executes the following tasks:

1. First, receive the input with the scheduling information of a CPU from the client program.
2. Next, use the incremental entropy algorithm proposed by Dr. Rincon to calculate the entropy of the CPU at each scheduling instant.
3. Finally, return the calculated entropies to the client program using sockets.

### The client program:

The user will execute this program using the following syntax:

```
./exec_filename hostname port_no < input_filename
```

where `exec_filename` is the name of your executable file, `hostname` is the address where the server program is located, `port_no` is the port number used by the server program, and `input_filename` is the name of the input file. The `hostname` and the port number will be available to the client as command-line arguments.

The client program receives from STDIN (using input redirection) `n` lines (where `n` is the number of input strings). Each line from the input represents the scheduling information of a CPU in a multiprocessor platform.

#### Example Input File:

```
A 2 B 4 C 3 A 7  
B 3 A 3 C 3 A 1 B 1 C 1
```

After reading the information from STDIN, this program creates `n` child threads (where `n` is the number of strings from the input). Each child thread executes the following tasks:

1. Receives the string with the scheduling information of the assigned CPU from the main thread.
2. Create a socket to communicate with the server program.
3. Send the scheduling information of the assigned CPU to the server program using sockets.
4. Wait for the entropy array from the server program.
5. Write the received information into a memory location accessible by the main thread.

Finally, after receiving the entropy values, the main thread prints the scheduling information for each CPU and the entropy. Given the previous input, the expected output is:

```
CPU 1  
Task scheduling information: A(2), B(4), C(3), A(7)  
Entropy for CPU 1  
0.00 0.92 1.53 1.42  
  
CPU 2  
Task scheduling information: B(3), A(3), C(3), A(1), B(1), C(1)  
Entropy for CPU 2  
0.00 1.00 1.58 1.57 1.57 1.58
```

**Notes:**

- You can safely assume that the input files will always be in the proper format.
- You must use the output statement format based on the example above.
- For the client program, you must use POSIX Threads and stream sockets. A penalty of 100% will be applied to submissions not using POSIX Threads and Stream Sockets.
- You must use multiple processes (fork) and stream sockets for the server program. A penalty of 100% will be applied to submissions not using multiple processes and Stream Sockets.
- The Moodle server will kill your server program after it is done executing each test case.
- You must present code that is readable and has comments explaining the logic of your solution. A 10% penalty will be applied to submissions not following this guideline.
- You cannot use global variables. A 100% penalty will be applied to submissions using global variables.
- A penalty of 100% will be applied to solutions that do not compile.
- A penalty of 100% will be applied to solutions hardcoding the output.

**RUBRIC:**

- 5 points for each test case (20 points total)
- 10 points for presenting clean and readable code.
- 10 points for using the fireman function correctly or using a for loop based on the number of requests.
- 10 points for reading information from STDIN (client).
- 25 points if each request to the server (5 points each):
  - Receives information from the client program.
  - Uses the Incremental Entropy Algorithm.
  - Returns the entropy values back to the client program.
  - Closes the socket.
- 25 points if each child thread (5 points each):
  - Creates a socket to communicate with the server.
  - Sends and receives information to/from the server.
  - Stores the information on a memory location accessible by the main thread.
  - Closes the socket.

**ADDITIONAL PENALTIES:**

100 points off for a solution not compiling (in addition to test cases not passing)

100 points off for lack of generalized solution (hard coding, no server file, etc.)

100 points off for not using POSIX threads, fork(), or Sockets.

100 points for using global variables.