Assignment 2

48450 REAL-TIME OPERATING SYSTEMS

Prashant Shrestha | Roshish Karmacharya
98136563 | 11823871

Contents

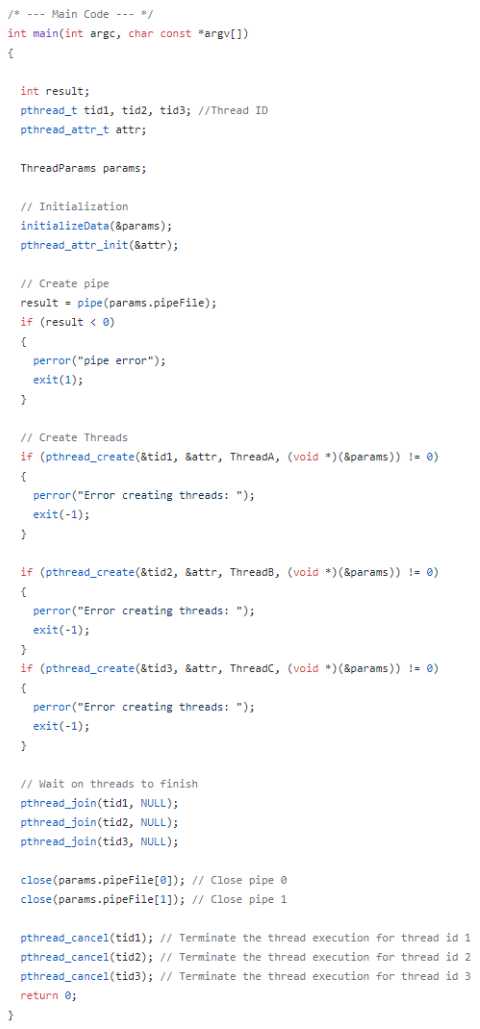
1. Introduction

1.1 Purpose

The primary purpose of this project is to utilise the “data.txt” file that has been provided from which we separate the header and content of the file and write the content in to a new text file “output.txt” using the concept of threads and pipes.

2. Code

2.1 Main



```
/* --- Main Code --- */
int main(int argc, char const *argv[])
{

  int result;
  pthread_t tid1, tid2, tid3; //Thread ID
  pthread_attr_t attr;

  ThreadParams params;

  // Initialization
  initializeData(&params);
  pthread_attr_init(&attr);

  // Create pipe
  result = pipe(params.pipeFile);
  if (result < 0)
  {
    perror("pipe error");
    exit(1);
  }

  // Create Threads
  if (pthread_create(&tid1, &attr, ThreadA, (void *)(&params)) != 0)
  {
    perror("Error creating threads: ");
    exit(-1);
  }

  if (pthread_create(&tid2, &attr, ThreadB, (void *)(&params)) != 0)
  {
    perror("Error creating threads: ");
    exit(-1);
  }
  if (pthread_create(&tid3, &attr, ThreadC, (void *)(&params)) != 0)
  {
    perror("Error creating threads: ");
    exit(-1);
  }

  // Wait on threads to finish
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  pthread_join(tid3, NULL);

  close(params.pipeFile[0]); // Close pipe 0
  close(params.pipeFile[1]); // Close pipe 1

  pthread_cancel(tid1); // Terminate the thread execution for thread id 1
  pthread_cancel(tid2); // Terminate the thread execution for thread id 2
  pthread_cancel(tid3); // Terminate the thread execution for thread id 3
  return 0;
}
```

The main function is responsible for data and thread initialisation using initializeData(&params)and pthread_attr_init(&attr). It is also helps the program create and terminate thread and pipes. The threads are created individually and returns an error when pthread_create() does not return 0. The threads and pipes are created and terminated one at time.

2.2 Initialize data

```c
/* --- Data Initialisation --- */
void initializeData(ThreadParams *params)
{
  // Initialize Sempahores
  sem_init(&(params->sem_A_to_B), 0, 1);
  sem_init(&(params->sem_B_to_A), 0, 0);
  sem_init(&(params->sem_C_to_A), 0, 0);

  params->endOfFile = 0; // Sets end of file value to 0 and initialize it
}
```

The initilizeData() is called on main the function that initialises the semaphores along with the end of file by setting it to 0. The sem_init() function s used to initialise data starting with sem_A_to_B followed by sem_B_to_A and sem_C_to_A.

2.3 Thread A

```c
/* --- Implementation of Thread A --- */
void *ThreadA(void *params)
{
  /* note: Since the data_stract is declared as pointer. the A_thread_params->message */
  ThreadParams *A_thread_params = (ThreadParams *)(params);
  static const char file[] = "data.txt"; // data.txt is stored into local variable filename

  FILE *fReader = fopen(file, "r"); // Open file to be read

  if (!fReader) // Validation check
  {
    perror(file);
    exit(-1);
  }

  // Wait for sem_A_to_B to finish
  while (!sem_wait(&(A_thread_params->sem_A_to_B)))
  {
    // Read the file
    if (fgets(A_thread_params->message, sizeof(A_thread_params->message), fReader) == NULL)
    {
      A_thread_params->endOfFile = 1; // set endOfFile flag to 1 after reaching end of file
      sem_post(&(A_thread_params->sem_C_to_A)); // Signal sem_C_to_A semaphore
      break;
    }

    // Write the data from reader to the pipe
    write(A_thread_params->pipeFile[1], A_thread_params->message, strlen(A_thread_params->message) + 1);
    sem_post(&(A_thread_params->sem_C_to_A)); // Signal sem_C_to_A semaphore
  }

  close(A_thread_params->pipeFile[1]); // Close Pipe 1
  fclose(fReader); // Close file

  return NULL;
}
```

Thread A starts off by opening the data.txt file with reading the content within it. Thread A takes the data from “data.txt” and writes in to a pipe which is done by the write() function.

2.4 Thread B

```c
void *ThreadB(void *params)
{

  ThreadParams *B_thread_params = (ThreadParams *)(params);

  // Wait for sem_C_to_A to finish
  while (!sem_wait(&(B_thread_params->sem_C_to_A)))
  {
    // Read the data from pipe 0 and push it to message
    read(B_thread_params->pipeFile[0], B_thread_params->message, sizeof(B_thread_params->message));
    sem_post(&(B_thread_params->sem_B_to_A)); // Signal sem_B_to_A semaphore

    if (B_thread_params->endOfFile == 1) // Break after reaching end of file
      break;
  }

  close(B_thread_params->pipeFile[0]); // Close Pipe 0
  return NULL;
}
```

The main thing that Thread B does is to read the data from pipe and push it to the message. This done by the read() function. Thread B breaks after reaching the end of file and closes the pipe once done reading

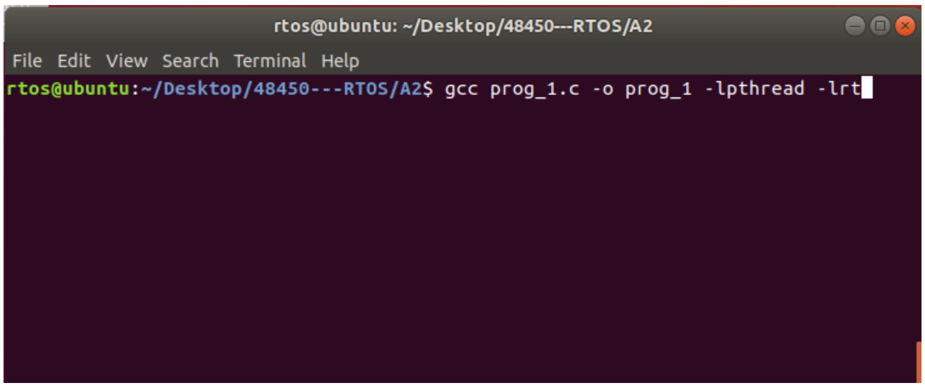
2.5 Thread C

```c
/* --- Implementation of Thread C --- */
void *ThreadC(void *params)
{
  ThreadParams *C_thread_params = (ThreadParams *)(params);
  int lineCount = 0; // Initialise line counter

  static const char file[] = "output.txt"; // output.txt is stored into local variable filename

  FILE *fWriter = fopen(file, "w"); // Open file to write

  if (!fWriter) // Validation check
  {
    perror(file);
    exit(-1);
  }

// Wait for sem_B_to_A to finish
  while (!sem_wait(&(C_thread_params->sem_B_to_A)))
  {
    if (lineCount)
    {
      // Put the data in message from fWriter file
      fputs(C_thread_params->message, fWriter);

      if (C_thread_params->endOfFile == 1) // Break after reaching end of file
        break;
    }
    else if (strstr(C_thread_params->message, "end_header")) // Check for end of message
      lineCount = 1;

    sem_post(&(C_thread_params->sem_A_to_B)); // Signal sem_A_to_B semaphore
  }

  fclose(fWriter); // Close file
  return NULL;
}
```

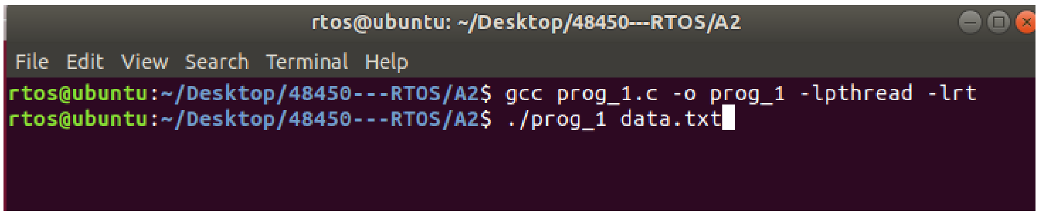
Thread C identifies whether the data from Thread B is from the file header region or not. The fputs() function puts the data in the message from file that was opened after which is stored in output.txt.

3. Results

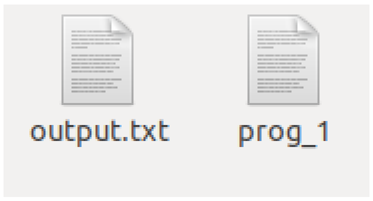
3.1 With semaphore

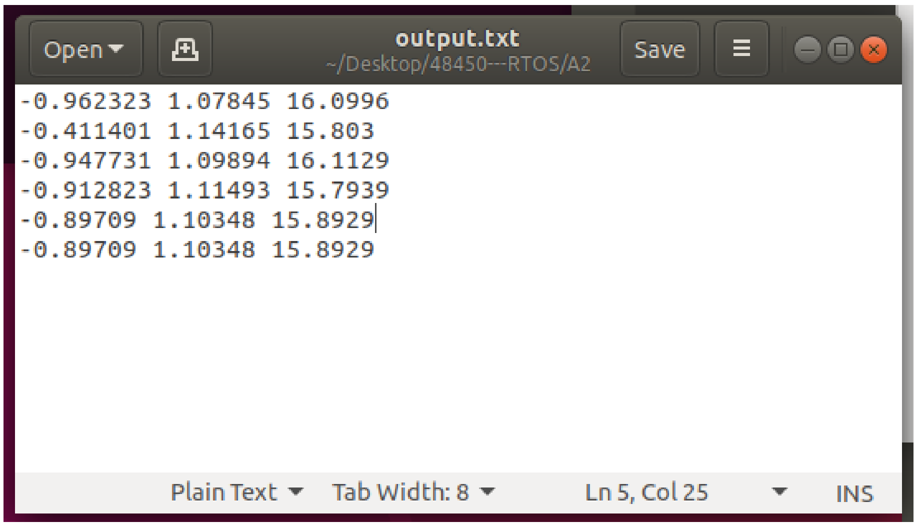

rtos@ubuntu: ~/Desktop/48450---RTOS/A2
File Edit View Search Terminal Help

```
rtos@ubuntu:~/Desktop/48450---RTOS/A2$ gcc prog_1.c -o prog_1 -lpthread -lrt
```


rtos@ubuntu: ~/Desktop/48450---RTOS/A2
File Edit View Search Terminal Help

```
rtos@ubuntu:~/Desktop/48450---RTOS/A2$ gcc prog_1.c -o prog_1 -lpthread -lrt
rtos@ubuntu:~/Desktop/48450---RTOS/A2$ ./prog_1 data.txt
```





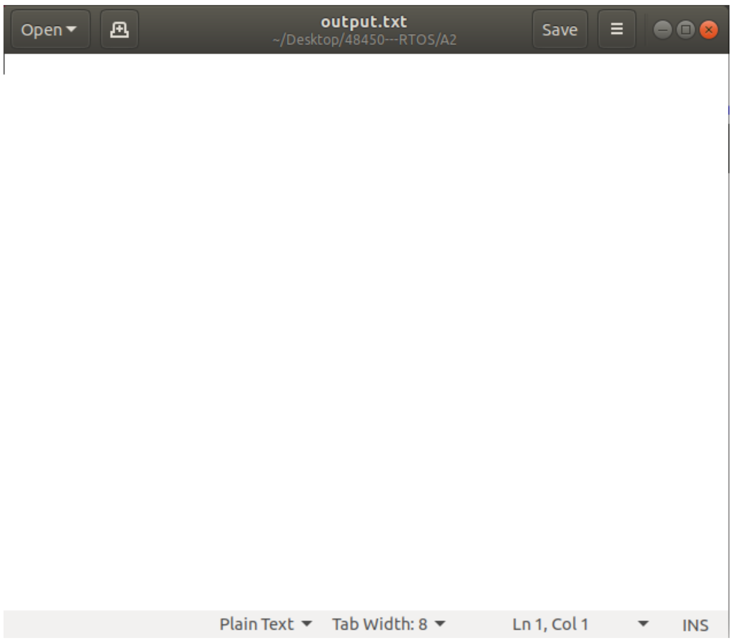
```
-0.962323 1.07845 16.0996
-0.411401 1.14165 15.803
-0.947731 1.09894 16.1129
-0.912823 1.11493 15.7939
-0.89709 1.10348 15.8929
-0.89709 1.10348 15.8929
```

Semaphores is a key that a program obtains to be able to continue the execution of various threads. They are mainly used for:

- controlling access to shared resources,
- signal when an event is occurred
- synchronising two threads

Once compiled, the program creates a file which we run to create the output.txt file with the results.

3.2 Without semaphore



When semaphore is removed, multiple threads try to read and write at the same time and the operating system will not be able to handle it which leads to data inconsistency. The output.txt file shown above is the result of semaphore not being present in the program which is empty.

4. Conclusion

We have successfully implemented the concept of threads, semaphores and pipes by creating three different threads i.e. Thread A, Thread B, Thread C which reads data from one file and writes the data to another file using the concept of pipe-line.