ECEN 5623

REAL-TIME EMBEDDED SYSTEMS

HOMEWORK 2

SUBMITTED BY:

SOWMYA RAMAKRISHNAN

108684769

Q.1

PROBLEM 2.5

Implement a Linux process that is executed at the default priority for a user-level application and waits on a binary semaphore to be given by another application. Run this process and verify its state using the ps command to list its process descriptor. Now, run a separate process to give the semaphore causing the first process to continue execution and exit. Verify completion.

ANSWER

A Linux process that is executed at the default priority for a user-level application and waits on a binary semaphore to be given by another application was written and implemented (“waiting.c”).

The code is as follows:

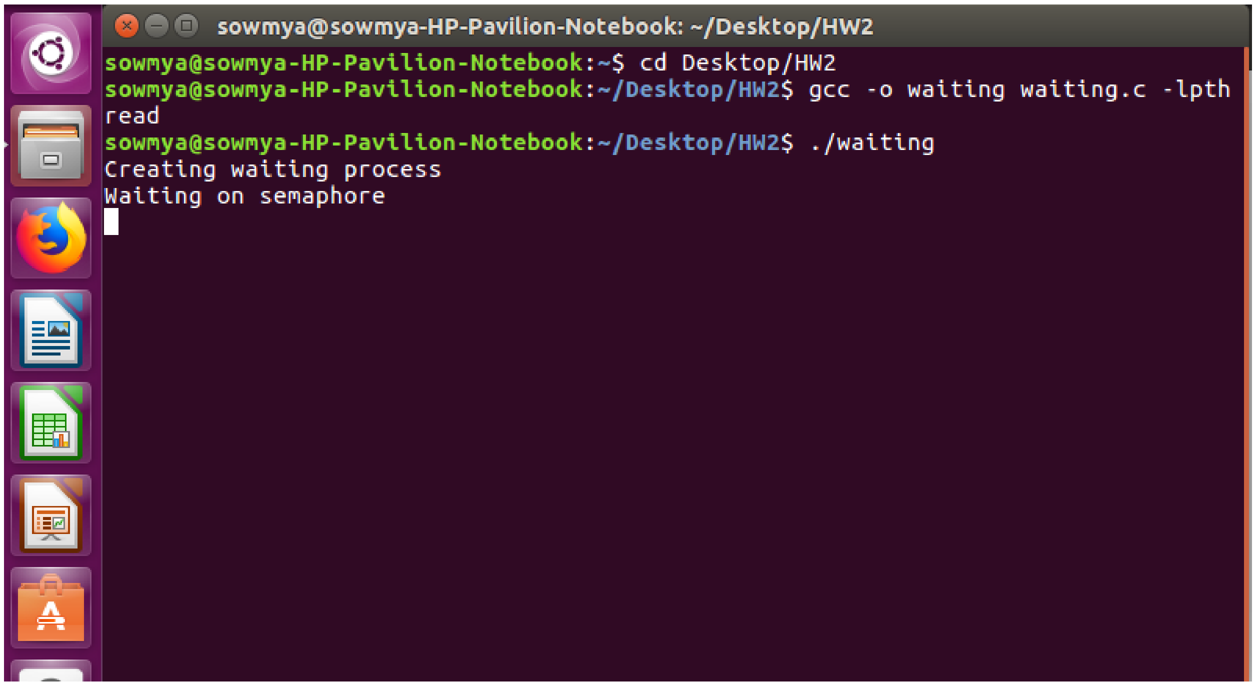
```
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <fcntl.h>

#define SEMNAME "mysemaphore"

sem_t *sem; //Semaphore Definition

int main()
{
  sem=sem_open(SEMNAME,O_CREAT,0644,0); //Creating, Initializing
semaphore to 0 and opening it
  printf("Creating waiting process\n");
  printf("Waiting on semaphore\n");
  sem_wait(sem); //Making the semaphore wait until the other process unlocks it
  printf("No longer waiting on semaphore\n"); //After the other process unlocks
the semaphore
  sem_destroy(sem); //Destroying semaphore
  exit(EXIT_SUCCESS);
}
```

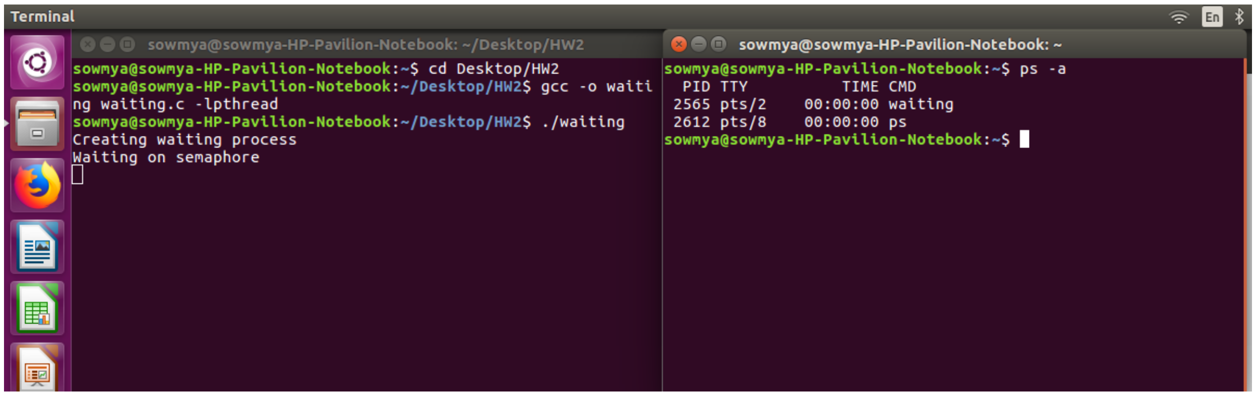
The execution of this process is shown as a screenshot as follows.



```
sowmya@sowmya-HP-Pavilion-Notebook:~$ cd Desktop/HW2
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ gcc -o waiting waiting.c -lpth
read
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ ./waiting
Creating waiting process
Waiting on semaphore
```

The state of the process was verified to be “Running” using the ps command to list its process descriptor.

The screenshot showing the process verification is as follows.


Terminal
sowmya@sowmya-HP-Pavilion-Notebook: ~/Desktop/HW2
sowmya@sowmya-HP-Pavilion-Notebook: ~

```
sowmya@sowmya-HP-Pavilion-Notebook:~$ cd Desktop/HW2
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ gcc -o waiti
ng waiting.c -lpthread
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ ./waiting
Creating waiting process
Waiting on semaphore
```

```
sowmya@sowmya-HP-Pavilion-Notebook:~$ ps -a
  PID TTY          TIME CMD
 2565 pts/2    00:00:00 waiting
 2612 pts/8    00:00:00 ps
sowmya@sowmya-HP-Pavilion-Notebook:~$
```

It can be seen that the process "waiting" is running/in active/ implementation stage.

Then, a separate process to give the semaphore causing the first process to continue execution and exit ("post.c") was written and executed.

The code is as follows.

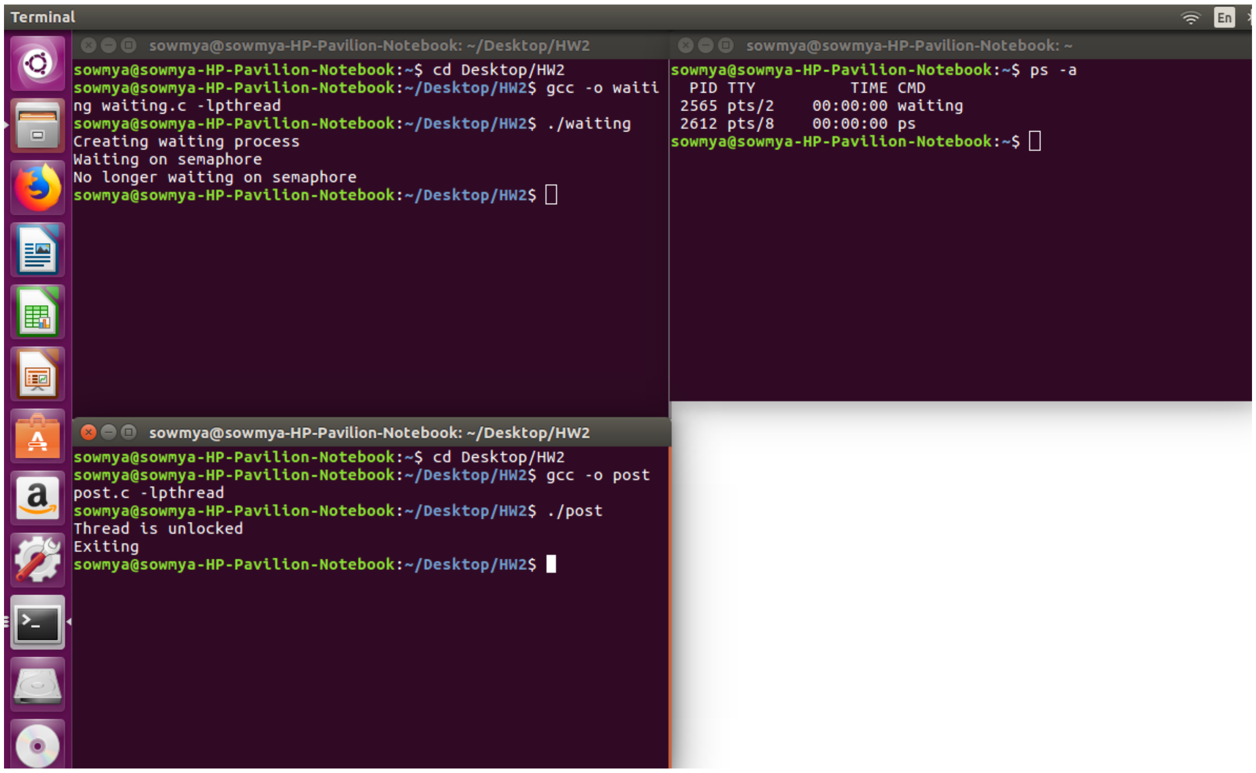
```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>
#include<sys/sem.h>
#include<sys/stat.h>
#include<fcntl.h>

#define SEMNAME "mysemaphore"

int main()
{
	sem_t *sem=sem_open(SEMNAME,0); ////Initializing semaphore to 0 and opening it
	printf("Thread is unlocked\n");
	sem_post(sem); //Incrementing the semaphore after bringing it out of wait state
	printf("Exiting\n");
	return 0;
}
```

This process essentially unlocks the waiting semaphore (brings it out of the "wait" state) and increments it, after which it goes into completion state (no longer running) and exits.

The screenshot showing the execution of this process is as follows.



Terminal

sowmya@sowmya-HP-Pavilion-Notebook: ~/Desktop/HW2

```
sowmya@sowmya-HP-Pavilion-Notebook:~$ cd Desktop/HW2
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ gcc -o waiti
ng waiting.c -lpthread
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ ./waiting
Creating waiting process
Waiting on semaphore
No longer waiting on semaphore
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ 
```

sowmya@sowmya-HP-Pavilion-Notebook: ~

```
sowmya@sowmya-HP-Pavilion-Notebook:~$ ps -a
  PID TTY          TIME CMD
 2565 pts/2    00:00:00 waiting
 2612 pts/8    00:00:00 ps
sowmya@sowmya-HP-Pavilion-Notebook:~$ 
```

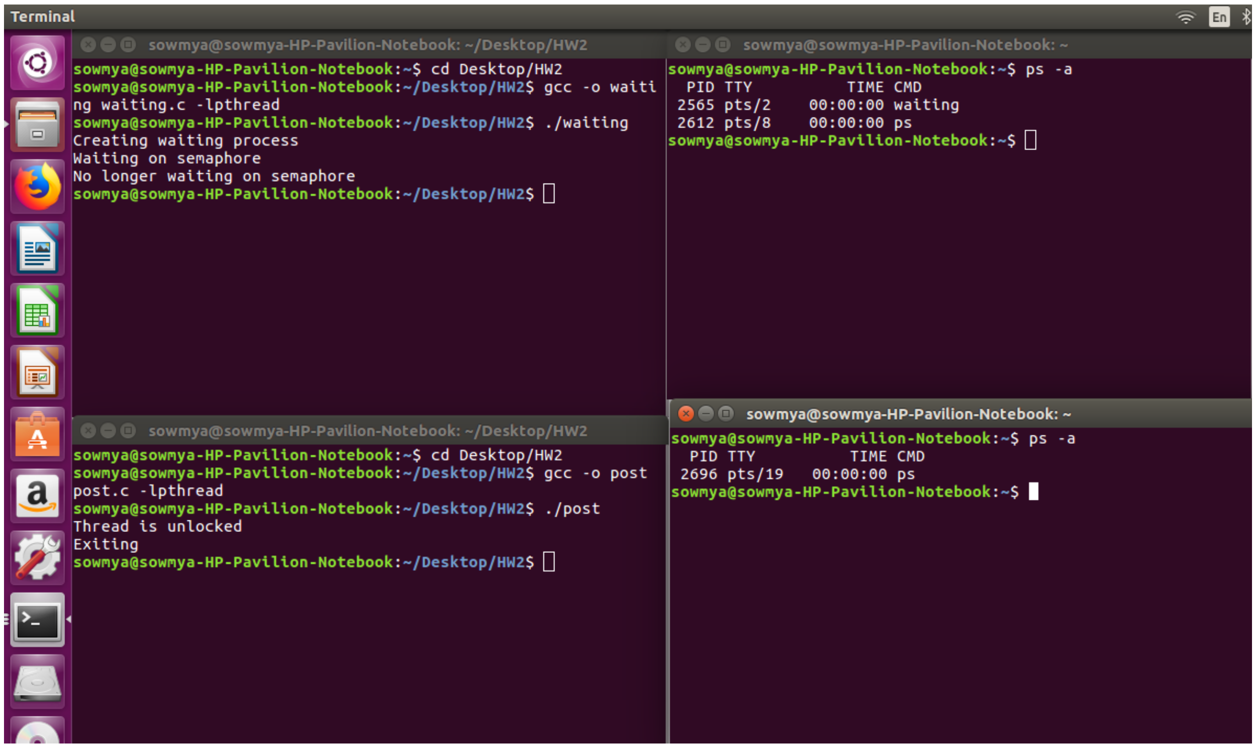
sowmya@sowmya-HP-Pavilion-Notebook: ~/Desktop/HW2

```
sowmya@sowmya-HP-Pavilion-Notebook:~$ cd Desktop/HW2
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ gcc -o post
post.c -lpthread
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ ./post
Thread is unlocked
Exiting
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ 
```

As soon as this process is implemented, in the screen where “waiting” process is running, we see the “No longer waiting on semaphore” output, which means that the semaphore has been successfully unlocked using the second process (“post”).

The completion is verified by using the ps command – printing the process descriptor.

The screenshot showing the successful verification of completion is as follows.


Terminal
sowmya@sowmya-HP-Pavilion-Notebook: ~/Desktop/HW2
sowmya@sowmya-HP-Pavilion-Notebook: ~

```
sowmya@sowmya-HP-Pavilion-Notebook:~$ cd Desktop/HW2
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ gcc -o waiti
ng waiting.c -lpthread
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ ./waiting
Creating waiting process
Waiting on semaphore
No longer waiting on semaphore
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$
```

```
sowmya@sowmya-HP-Pavilion-Notebook:~$ ps -a
  PID TTY          TIME CMD
 2565 pts/2    00:00:00 waiting
 2612 pts/8    00:00:00 ps
sowmya@sowmya-HP-Pavilion-Notebook:~$
```

```
sowmya@sowmya-HP-Pavilion-Notebook:~$ cd Desktop/HW2
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ gcc -o post
post.c -lpthread
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$ ./post
Thread is unlocked
Exiting
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop/HW2$
```

```
sowmya@sowmya-HP-Pavilion-Notebook:~$ ps -a
  PID TTY          TIME CMD
 2696 pts/19   00:00:00 ps
sowmya@sowmya-HP-Pavilion-Notebook:~$
```

Here, we see that, when the ps command is executed after the semaphore has exited, the "waiting" process is no longer in Running state. (Inactive/Unlocked and Completed)

Thus, the codes and screenshots explain the implementation of the two processes.

The code is attached as a separate file.

Q.2

PROBLEM 3.5

If EDF can be shown to meet deadlines and potentially has 100% CPU resource utilization, then why is it not typically the hard real-time policy of choice? That is, what are drawbacks to using EDF compared to RM/DM? In an overload situation, how will EDF fail?

ANSWER

EDF scheduling has several major drawbacks that make it undesirable in real-time systems. First, EDF requires extra computation time since the deadline time (when the deadline happens) must be computed first and then priorities must be restructured based on that. The deadline time isn't always deterministic which can lead to problems and the extra computation can slow the system down. It can also be noticed that, for lightly loaded systems, EDF and RM scheduling often generate the exact same schedule for tasks. Similarly, the more harmonic a system's tasks are, the more likely that EDF and RM will produce the same scheduling. So, if the tasks of a system are made harmonic, the (simpler) RM policy can thus be used instead of EDF. One other issue is that the sufficient bound for EDF scheduling is not easily computed as is the case for RM scheduling. It is also true that for harmonic task sets, the RM policy can get an extremely high utilization. This is useful because tasks can often be made to be harmonic with minor adjustments to their computation time.

The final drawback/disadvantage is in the case of a failing system. Rate Monotonic scheduling guarantees that the highest priority task will be the first task to miss a deadline if a deadline is going to be missed. This is very convenient for real time systems because of their determinism. Since one knows how the system will fail, they can build their system to react to that failure in an intelligent manner. This is not the case for EDF scheduling. In EDF scheduling, one does not know how a system will fail. It might fail the highest period task or the lowest period task first. Since one does not know which task will fail, it becomes much more difficult to make the system fail in an "elegant" way. Basically, EDF scheduling does not fail in a deterministic way, which makes it harder to work with in case of failures and this will often result in cascading failures. These result in many more missed deadlines which might end up breaking even a soft real time system.

The difference between RM and EDF in the case of transient overload is that, under RM, an overrun in task cannot cause tasks with higher priority to miss their deadlines, whereas under EDF any task could miss its deadline. RM is thus more predictable in overload conditions, while EDF is not.

Thus, EDF is not typically the hard real-time policy of choice.

Q.3

PROBLEM 4·2

If a system must complete frame processing so that 100,000 frames are completed per second and the instruction count per frame processed is 2120 instructions on a 1GH processor core, what is the CPI required for this system? What is the overlap between instructions and IO time if the intermediate IO time is 4·5 microseconds?

ANSWER

$f = 10^9$ cycles/second , IPF = 2120 inst./frame ,

FPS = 100000 frames/sec

To find: CPI

We need 2120 x 100000 = 212,000,000 instrudions / second

$$CPI = \frac{10^{\cancel{9}3}}{212 \times \cancel{10^6}} = \frac{1000}{212} = 4.7169 = 5 \; CPI$$

Thus, we need $\underline{\underline{5}}$ cycles per instruction.

Intermediate IO time : 4·5 µs. ; To find overlap.

The deadline time is 10 µs - since 100 000 frames need to be completed every second at even spacing. Instruction count time is the same, based on the definition of the system.

Thus, overlap percent : $1 - \frac{10^{-5} - 4.5 \times 10^{-6}}{10^{-5}} = \underline{\underline{45\ \%}}$

Q.4

PROBLEM 4.4

Review the DVD code (also on D2L) for heap_mq.c and posix_mq.c. Write a brief paragraph describing how these two message queue applications are similar and how they are different. Make sure you not only read the code but also build it, load it, and execute it to make sure you understand how both applications work.

ANSWER

The two codes, heap_mq.c and posix_mq.c were reviewed. They were understood, built, loaded and executed and a clarity was obtained on how both the applications work.

SIMILARITIES

Both functions:

- Give the receiver a higher priority than the sender.
- Ensure a safe way for the threads or processes to communicate with each other.
- Use tasks to run the sender and receiver functions.
- Use the posix queue for storing the messages with the function calls mq_open, mq_receive and mq_send.

DIFFERENCES

The first major difference between the two implementations is that heap_mq.c creates and opens the message queue before running the sender and receiver functions. The posix_mq.c code opens and creates the message queue in the receiver function and it is again opened in the sender function.

The other major difference is that heap_mq.c uses dynamically allocated memory to be used as the buffer, while posix_mq.c uses stack memory.

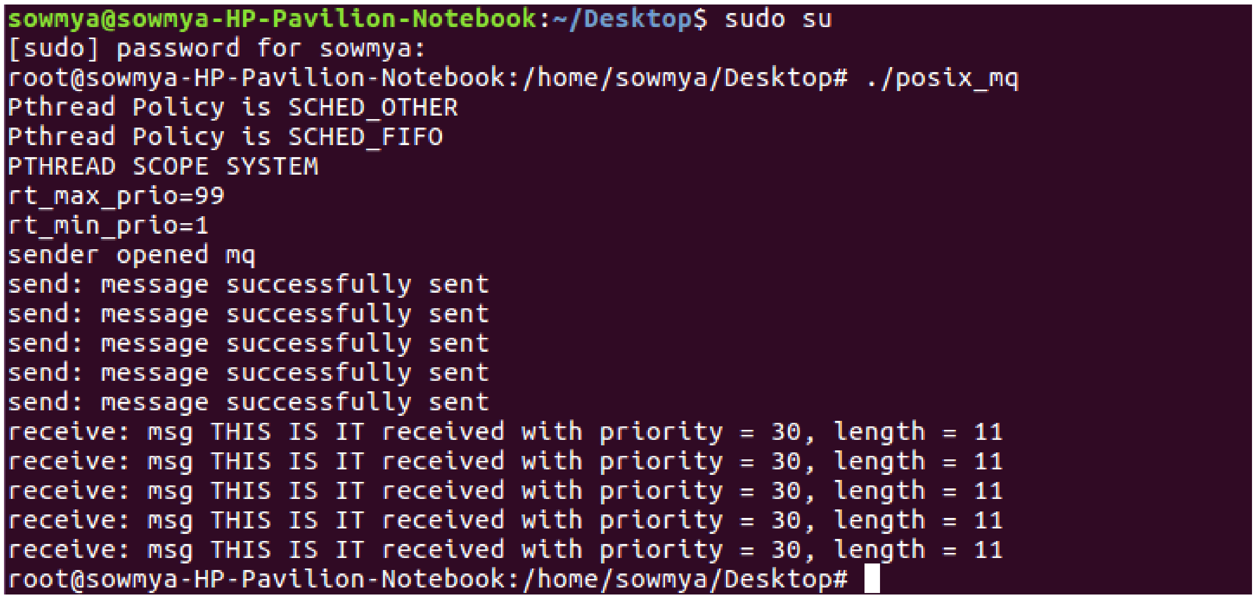
In message queue, the message transferred is the actual data and in the heap queue, the data transferred are actually pointers to messages. The pointers are set to point to a buffer allocated by the sender, and the pointer received is used to access and process the buffer. Hence in the heap queue, the queue is a buffer of pointers.

Heap_mq.c also implements the sender and receiver functions as infinite loops that must be closed by deleting the tasks, while posix_mq.c only runs through the sending and receiving functions once.

In message queue, it is important to ensure that the read and write to the buffer is atomic using some semaphore or mutex. Such a restriction is not necessary in case of heap queue.

For message queue the size of the message queue always remains constant which is determined when the queue is opened, but in case of the heap queue, the sender allocates the buffer and the receiver de-allocates to avoid exhaustion of the associated buffer heap.

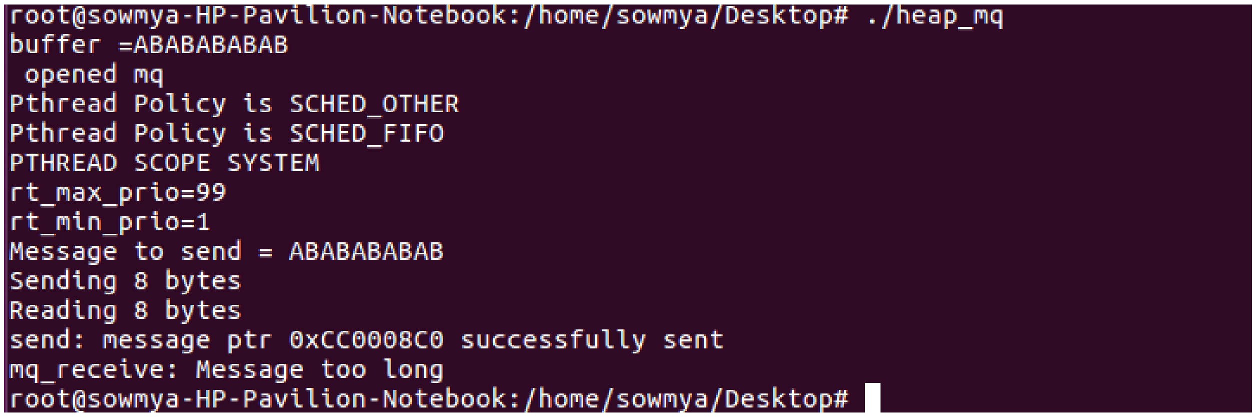
Posix_mq



```
sowmya@sowmya-HP-Pavilion-Notebook:~/Desktop$ sudo su
[sudo] password for sowmya:
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop# ./posix_mq
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
sender opened mq
send: message successfully sent
send: message successfully sent
send: message successfully sent
send: message successfully sent
send: message successfully sent
receive: msg THIS IS IT received with priority = 30, length = 11
receive: msg THIS IS IT received with priority = 30, length = 11
receive: msg THIS IS IT received with priority = 30, length = 11
receive: msg THIS IS IT received with priority = 30, length = 11
receive: msg THIS IS IT received with priority = 30, length = 11
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop#
```

In the code, the sender and receiver both open the queue. Attributes like message queue size, length and priority are specified while opening the queue. After the queue is opened, the message can be sent using the mq_send() function; mq_receive() can be used to receive the message.

Heap_mq



```
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop# ./heap_mq
buffer =ABABABABAB
 opened mq
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Message to send = ABABABABAB
Sending 8 bytes
Reading 8 bytes
send: message ptr 0xCC0008C0 successfully sent
mq_receive: Message too long
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Desktop#
```

In the code, heap queue is used to transfer the message. Here again, mq_open() is used to open the queue but in the sender, a pointer to the heap is specified so that the queue is opened in the heap. And while using mq_send(), message is sent to the heap, allocating memory each time. And while reading the message, we use mq_read() and de-allocate the memory.