Producer-Consumer Problem on an OS.

1. Problem Statement

To simulate the producer-consumer problem on an operating system as demonstrated in Fig. 5.16 of the course text. The producer generates a million items and stores it in a buffer of size one thousand. There will be 10 competing customers all of which are child processes as is the producer created using the fork() procedure.

The importance of the project is to learn the concepts of mutual exclusion and concurrency by having multiple processes communicating with each other as they compete for a limited resource on the OS. Mutual exclusion is achieved through a software means using Linux's inbuilt semaphores. This was important first step in understanding the demands of running parallel processes or threads and making an implementation safe for concurrent access from multiple processes.

2. Approach

Language used: C++11

Compiler used: g++

Editor used: micro

Debugging tools used: valgrind and gdb.

First task was to learn how to create and use shared memory in C++. This involved reading the man pages for shm_open(), mmap(), ftruncate(), munma(), and shm_unlink().

Next task was to create a circular queue class that can act as the buffer for storing the items produced by the producer and to be consumed by the customers.

Then came the definition of the structure that can hold all the shared variables between the processes and can be mapped into the shared memory area.

The producer and consumer procedures were defined using the pseudocode in figure 5.16 of the course text. The appropriate semaphores also added in the shared memory (shmem) structure (struct).

The shmem struct also holds the buffer. In the main procedure which calls on all the other procedures and forks to create multiple process also managing cleanup of the process, the shmem struct is initialized. Later the struct is mapped into a shared memory space and the virtual memory space of the process itself. The mapping is replicated in the memory space of all children processes and a change in one of them is propagated to all. This forms the basis of inter-process communication (IPC). Using the semaphores and the buffer the IPC is directed in such a way so as to simulate the producer-consumer problem defined earlier.

3. Solution

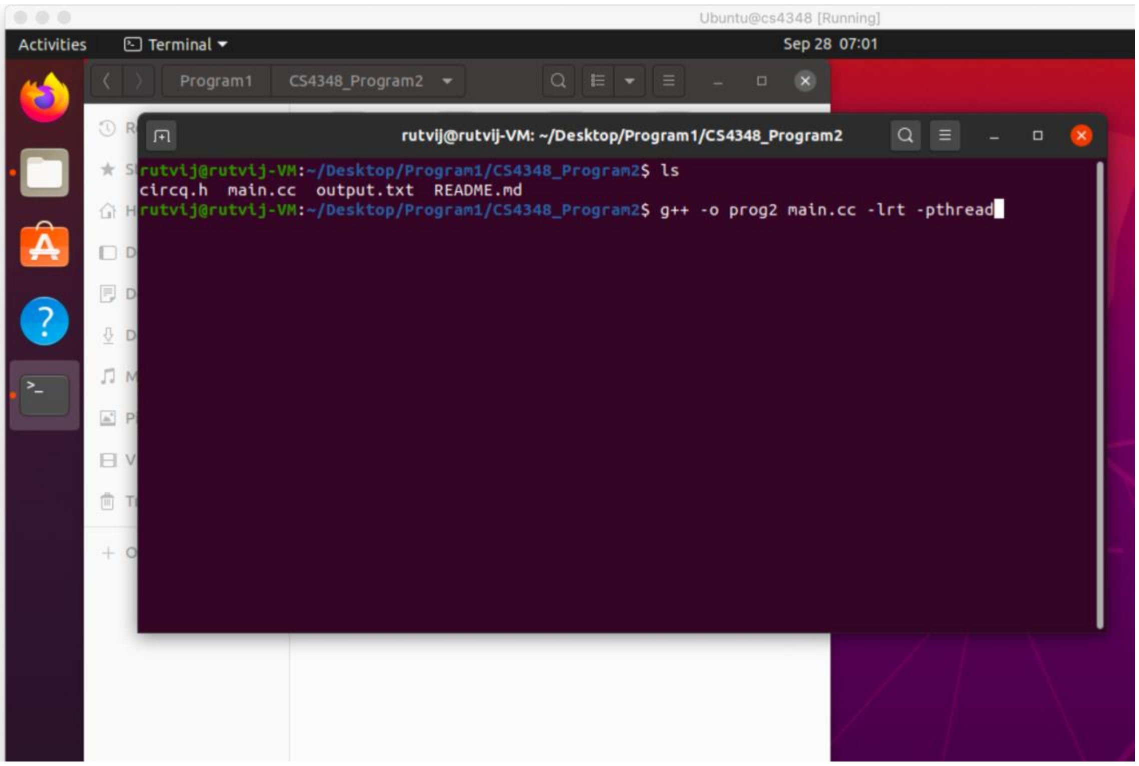
Major bugs had to be detected using valgrind and gdb, since the bug was causing a segmentation fault and was hard to isolate.

A minor modification was made to the classic solution presented in the text since at the end of production, due to parallelism and pipelining of multiple instructions, I suspected that the consumers were calling on the fillCount semaphore even after the stop of production and end of consumption, probably due to the lag time in the Boolean flag updating and the instructions being pipelined. This caused a deadlock after the end of all consumption since the processes were suspended because they were attempting to overconsume and thus blocked by the semaphore fillCount. And since the producer was no longer increasing the fillCount to release the semaphores, the consumers remained blocked permanently.

This issue was resolved by the addition of a check within the loops body to ensure there is no decrement of fillCount if the queue is empty AND the production has stopped. The solution is further described in the appropriate section of the code.

To build the code g++ is invoked and linker flags are supplied as follows (Figure 1).

```
$ g++ -o prog2 main.cc -lrt -pthread
```

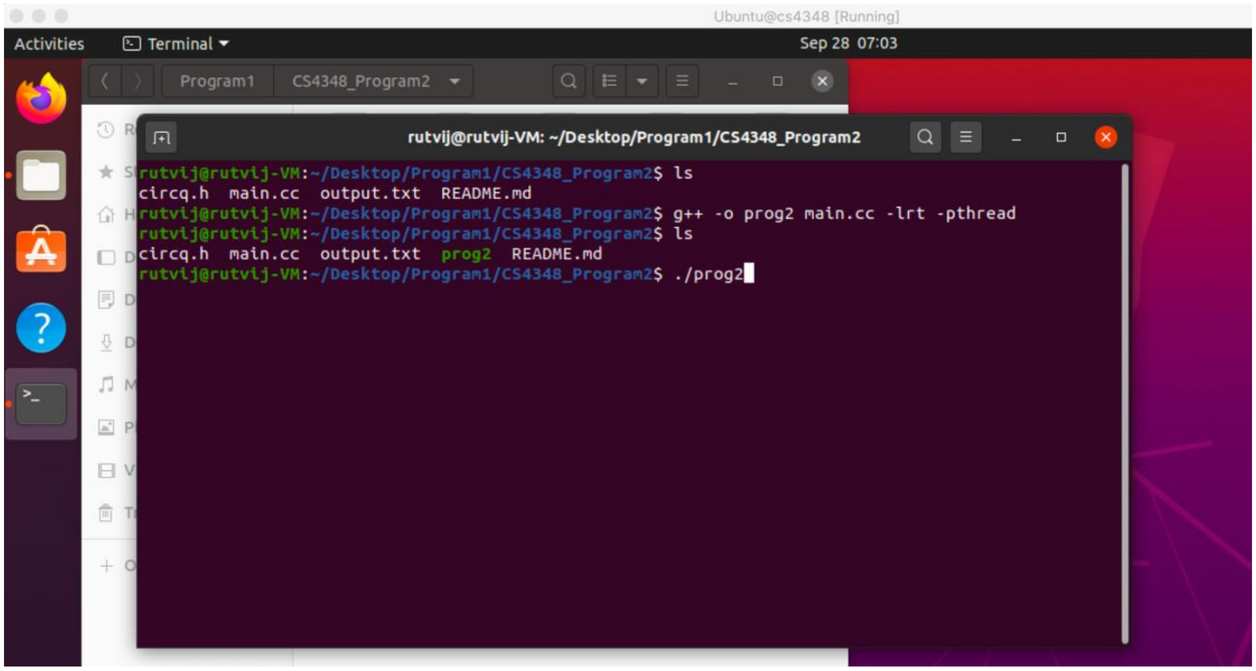


```
rutvij@rutvij-VM:~/Desktop/Program1/CS4348_Program2$ ls
circq.h  main.cc  output.txt  README.md
rutvij@rutvij-VM:~/Desktop/Program1/CS4348_Program2$ g++ -o prog2 main.cc -lrt -pthread
```

Figure 1

To run the code the following command is supplied (Figure 2).

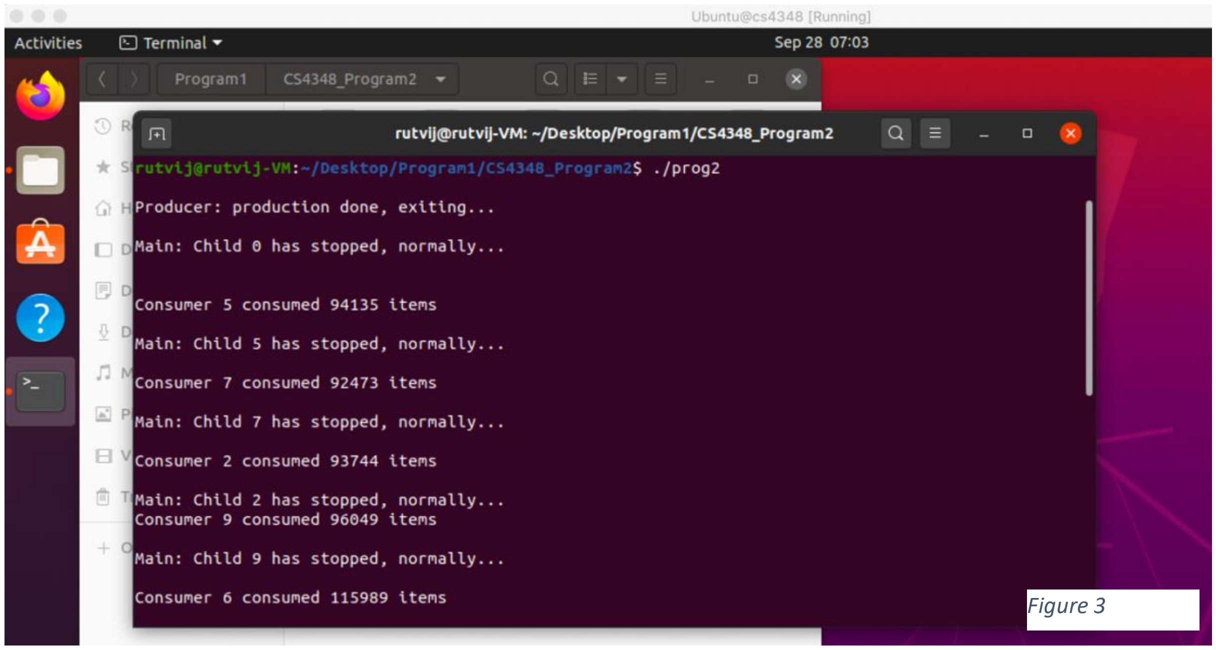
$ ./prog2



```
rutvij@rutvij-VM:~/Desktop/Program1/CS4348_Program2$ ls
circq.h  main.cc  output.txt  README.md
rutvij@rutvij-VM:~/Desktop/Program1/CS4348_Program2$ g++ -o prog2 main.cc -lrt -pthread
rutvij@rutvij-VM:~/Desktop/Program1/CS4348_Program2$ ls
circq.h  main.cc  output.txt  prog2  README.md
rutvij@rutvij-VM:~/Desktop/Program1/CS4348_Program2$ ./prog2
```

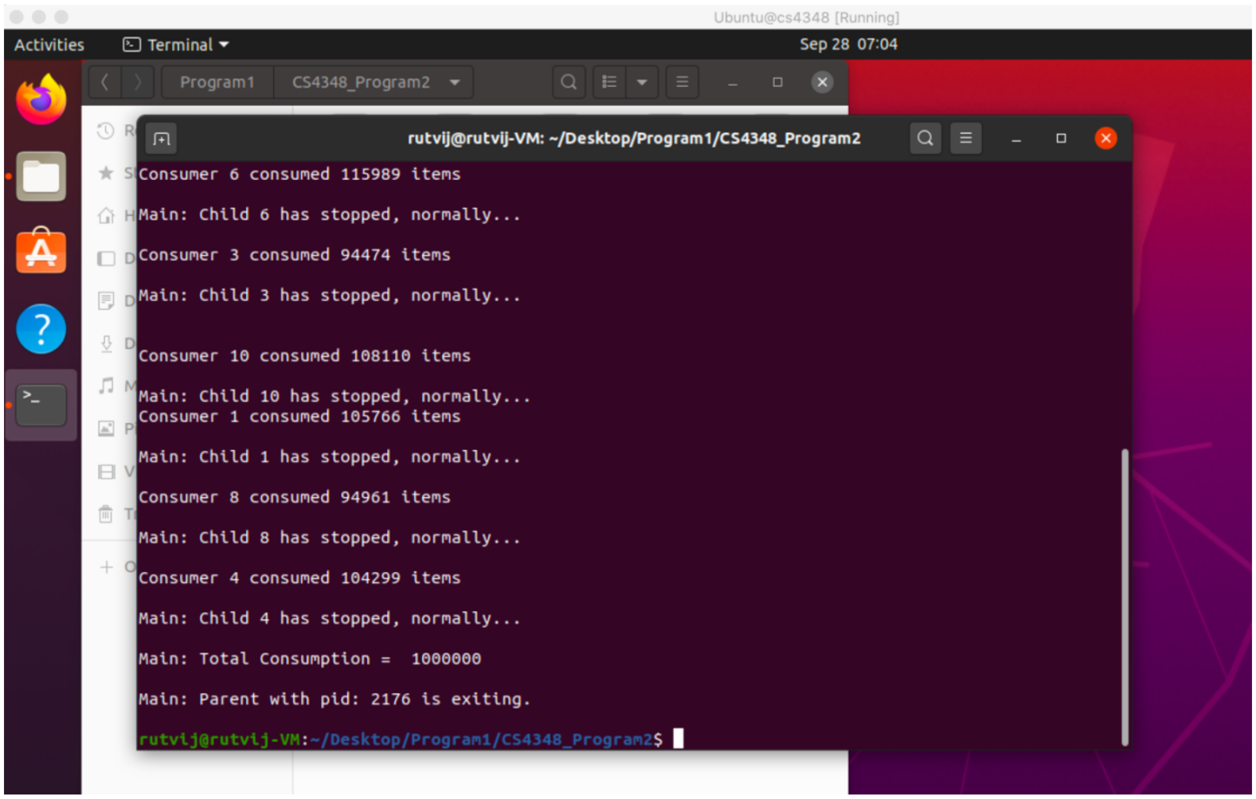
Figure 2

The following is the output generated by the code (Figures 3 & 4).



```
rutvij@rutvij-VM:~/Desktop/Program1/CS4348_Program2$ ./prog2

Producer: production done, exiting...

Main: Child 0 has stopped, normally...


Consumer 5 consumed 94135 items

Main: Child 5 has stopped, normally...

Consumer 7 consumed 92473 items

Main: Child 7 has stopped, normally...

Consumer 2 consumed 93744 items

Main: Child 2 has stopped, normally...
Consumer 9 consumed 96049 items

Main: Child 9 has stopped, normally...

Consumer 6 consumed 115989 items
```

Figure 3



```
Consumer 6 consumed 115989 items

Main: Child 6 has stopped, normally...

Consumer 3 consumed 94474 items

Main: Child 3 has stopped, normally...


Consumer 10 consumed 108110 items

Main: Child 10 has stopped, normally...
Consumer 1 consumed 105766 items

Main: Child 1 has stopped, normally...

Consumer 8 consumed 94961 items

Main: Child 8 has stopped, normally...

Consumer 4 consumed 104299 items

Main: Child 4 has stopped, normally...

Main: Total Consumption =  1000000

Main: Parent with pid: 2176 is exiting.

rutvij@rutvij-VM:~/Desktop/Program1/CS4348_Program2$
```

Figure 4

The output shows when each child process has been terminated and consequently reaped by the parent process and the consumption of each child.

The producer prints when it has finished production and is consequently terminated using exit(), this leads the parent to reap it and is signaled by main telling Child 0 has stopped normally.

After that as each consumer realizes that the queue is empty and the production has stopped, they begin being terminated via exit() and are reaped by the parent process (main). Main signals as each child from 1 to 10 corresponding to each Consumer from 1 to 10 stops and exits normally.

Each consumer also reports the total number it has consumed and as we can see it is an average of 100k items (±10% approx.) per consumer. Some consumers consume more while some get to consume less depending upon how each of those processes was scheduled. Each time the program is run the numbers vary but are within the range mentioned.

Before the parent exits, it accesses the variable in shared memory that was tracking total consumption by all consumers and prints it out. This consistently prints out to be a million items as we would expect since a million items are being produced.