Lab-3 Report

This report talks about the implementation of *User Level Threads* Library and some tests to check its correctness and feasibility. I then also implement a matrix multiplication using our threads and also using pthreads and do some timing analysis.

In this lab, I learnt the following :

- Difference between User Level Threads and Kernel Level Threads
- Use of setjmp and longjmp system calls
- Use of sub-makes
- How to make a shared library
- Use of signals in linux for inter-process communication
- Semaphores
- Practical implementation of producer consumer problem.
- How signal handlers work.

I first discuss the internal logic and design of the library, then I discuss the tests and their results, followed by graphs and data values section. I then also discuss the possible use improvements in the library to make it a more robust and better implementation of the user-level threads support.

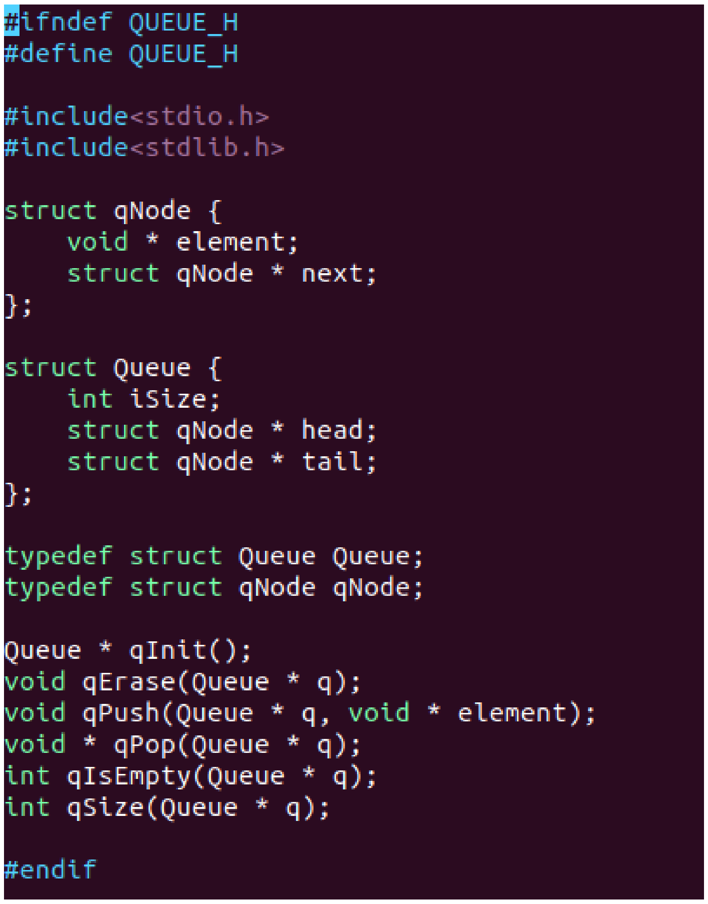
While implementing the library, I got stuck in an error which cost me a lot of time and effort. But in this process, I learnt about using valgrind to seek for memory corruptions/errors in my code. I spent a good amount of time using gdb. The pre-emption in the thread library makes it difficult to debug, and so I had to add debug print statements, to trace the execution as a log file. Finally, I was able to catch the bug, which was a pointer not being updated in the queue, when the queue becomes empty.

(please turn over)

Internal Logic & Design Of the Library

1. Queue Implementation using (void *) as elements.
2. Timer using SIGVTALRM.
3. Thread Stack Space Management
4. Scheduling using FIFO based global queue.
5. Semaphores
6. Basic Locks

Queue Implementation



```
#ifndef QUEUE_H
#define QUEUE_H

#include<stdio.h>
#include<stdlib.h>

struct qNode {
    void * element;
    struct qNode * next;
};

struct Queue {
    int iSize;
    struct qNode * head;
    struct qNode * tail;
};

typedef struct Queue Queue;
typedef struct qNode qNode;

Queue * qInit();
void qErase(Queue * q);
void qPush(Queue * q, void * element);
void * qPop(Queue * q);
int qIsEmpty(Queue * q);
int qSize(Queue * q);

#endif
```

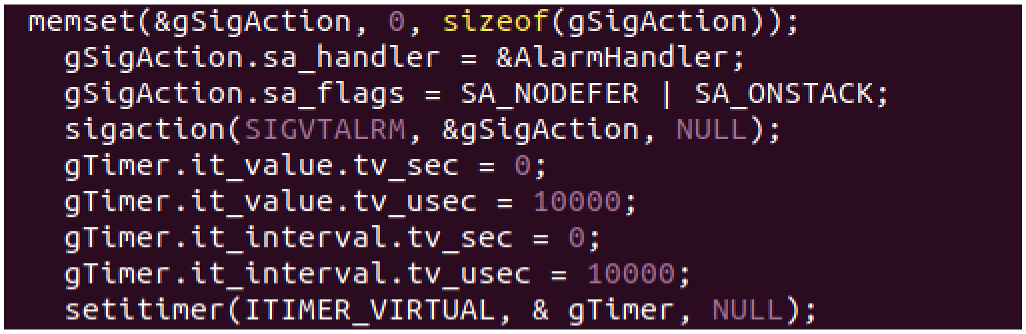
As we can see, I made a queue which could accept elements as a (void*) and thus making it effectively generic to whatever stuff I wish to put inside this queue.

Timer Using SIGVTALRM

Now, out of many signals in linux. There is

- SIGALRM :: Counts the real time as measured by a stopwatch.
- SIGVTALRM :: Counts the CPU time used by the process.
- SIGPROF :: Counts the CPU time as well as system time on behalf of the process.

Thus, SIGVTALRM looked like a good option, since I would want to give each thread an equal amount of opportunity which doesn't depend on the CPU time given to other processes.
As we can see here, I made a signal handler "AlarmHandler" function. And then I use *setitimer* to get a SIGVTALRM signal at every 10000 micro seconds.



```
memset(&gSigAction, 0, sizeof(gSigAction));
  gSigAction.sa_handler = &AlarmHandler;
  gSigAction.sa_flags = SA_NODEFER | SA_ONSTACK;
  sigaction(SIGVTALRM, &gSigAction, NULL);
  gTimer.it_value.tv_sec = 0;
  gTimer.it_value.tv_usec = 10000;
  gTimer.it_interval.tv_sec = 0;
  gTimer.it_interval.tv_usec = 10000;
  setitimer(ITIMER_VIRTUAL, & gTimer, NULL);
```

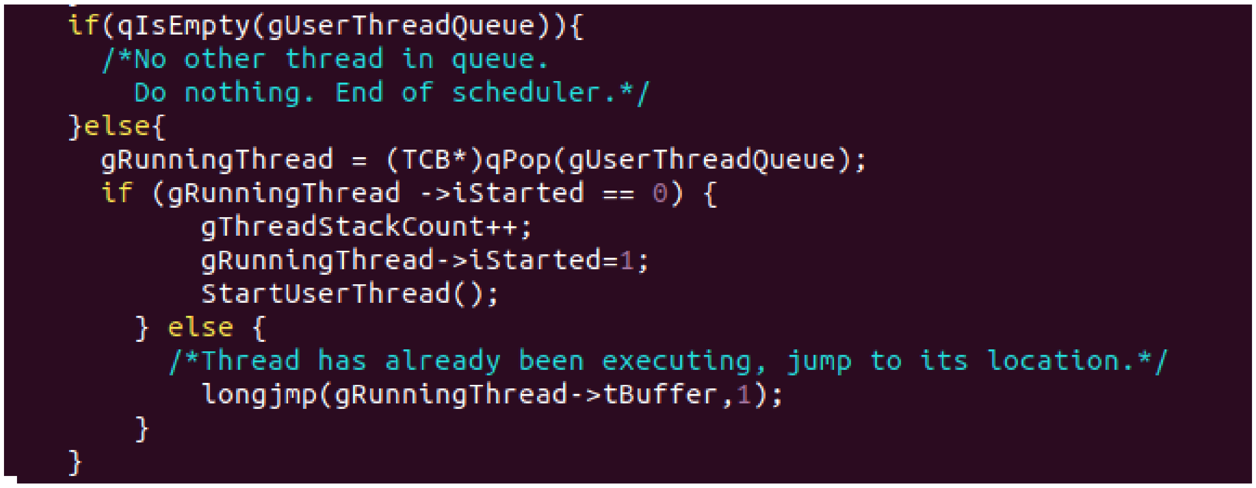
Thread Stack Space Management

This is a tricky part in the library. Now, I give stack space to each thread using the process stack. And the way it's done is that, I first allocate a large amount of space from the stack and then, I longjmp to the earlier state of process. Due to this jumping back to our previous state, our process's stack pointer now points to the address of the earlier state. Now, I incrementally provide each thread its space and similarly go forward.
Now, this has a couple of **limitations**.

- The stack space of each user thread is limited to a size allocated by the StartUserThread() function.
- Once a user thread finishes, I cannot free its stack space, since, if I free it, the stack will get corrupted and the memory addresses of other running user threads will become invalid.

Scheduling using FIFO based global queue.

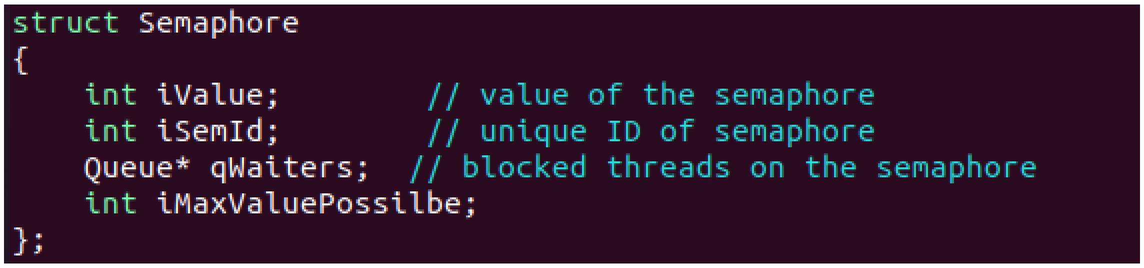
To make the scheduler run in a FIFO manner, I have made use of queue implementation and used global queues to separate working threads from blocked state(waiting state) to the ready state. There is a global variable which stores the pointer of the current running thread.
Below screenshot, represents the working of Scheduler.



```
    if(qIsEmpty(gUserThreadQueue)){
      /*No other thread in queue.
        Do nothing. End of scheduler.*/
    }else{
      gRunningThread = (TCB*)qPop(gUserThreadQueue);
      if (gRunningThread ->iStarted == 0) {
            gThreadStackCount++;
            gRunningThread->iStarted=1;
            StartUserThread();
        } else {
          /*Thread has already been executing, jump to its location.*/
            longjmp(gRunningThread->tBuffer,1);
        }
    }
```

Semaphores

Semaphores are basically counters with wait queues. It allows you to have some counter for a resource, which the multiple threads can use simultaneously without blocking. When the counter value reaches zero, the threads requesting access to the resource are put into the wait queue and are sent to ready state once the resource counter increases.
Here is the basic structure I have used for the semaphore :

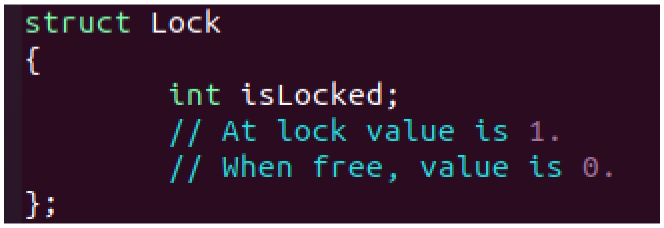


```
struct Semaphore
{
    int iValue;         // value of the semaphore
    int iSemId;         // unique ID of semaphore
    Queue* qWaiters;   // blocked threads on the semaphore
    int iMaxValuePossilbe;
};
```

Locks

Locks are a way of providing mutual exclusion to some areas of code, generally known as Critical Section. I have not added waiting queues in locks, and any of all the threads acquiring the lock can receive it without any order. Any thread which is requesting access to the lock, which has already been acquired by any other thread, busy waits on the lock. To effectively remove busy waiting I have disabled the timer for a thread which acquires the lock, thus giving it the priority to execute its critical section. User should be careful in using locks, since

- It can lead to deadlocks if not used carefully.



```
struct Lock
{
        int isLocked;
        // At lock value is 1.
        // When free, value is 0.
};
```

(please turn over)

Tests

I perform two tests over our implementation,

1) Matrix Multiplication progam
2) Bounded Buffer program

1) Matrix Multiplication Program

In this program, I multiply a matrix (NxN size) using multiple threads, to generate speed.
I perform this multiplication using our threads library and using pthreads library and observe the results. The results of this test are given in the results section down below.

2) Bounded Buffer Program

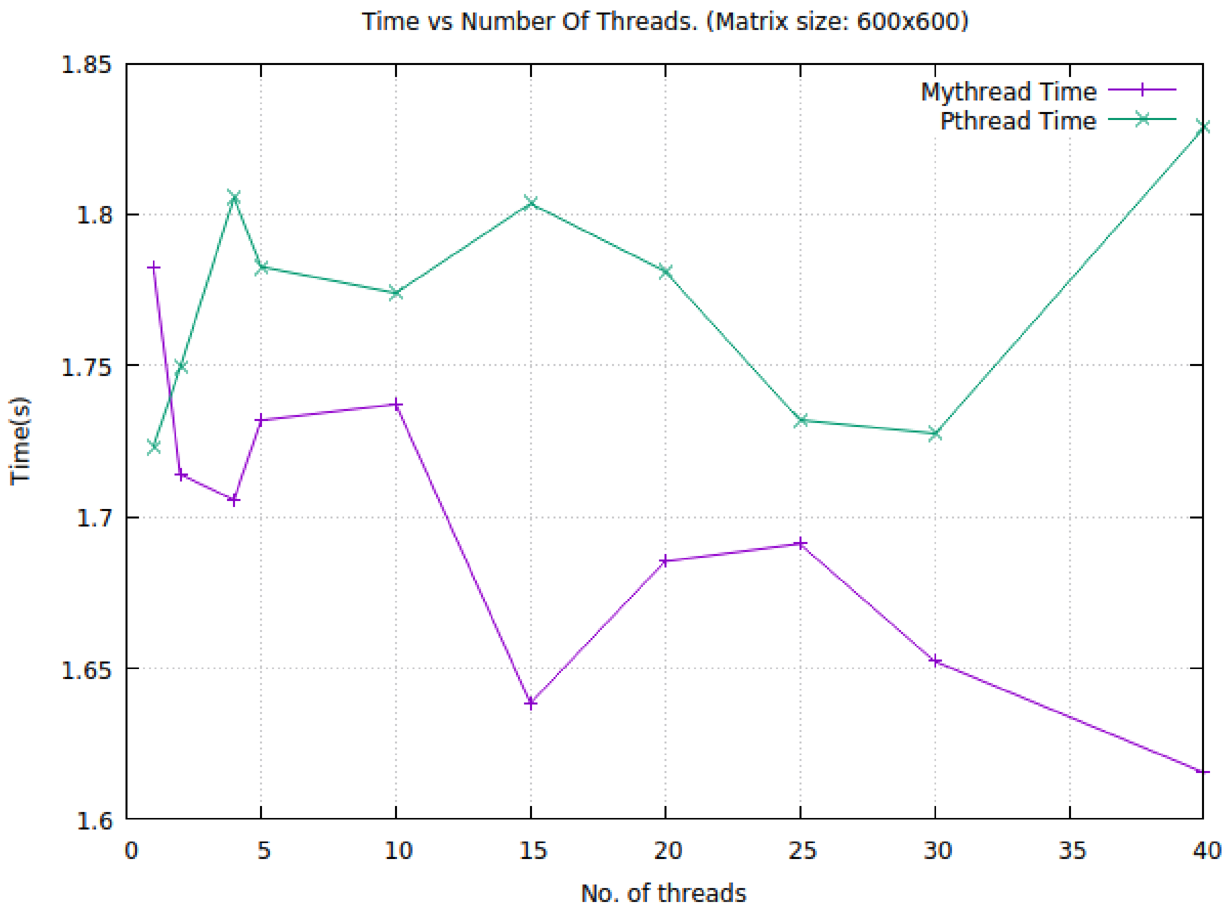
This is based on the famous Producer Consumer problem. It consists of a fixed size buffer, which the producers and consumers use to produce and consume objects respectively.
Now, this program tests the semaphores (condition variables) implementation of the library.
A consumer cannot consume an object unless it has been produced by any producer, similarly a producer cannot produce an object if the buffer has reached its maximum capacity and is no more empty. The output of the program is given in the results section down below.
I also added a for loop, so that each thread is doing some other work to slow down the producing and consuming task, thus giving other user-level thread a chance to produce/consume an object.

(please turn over)

Results

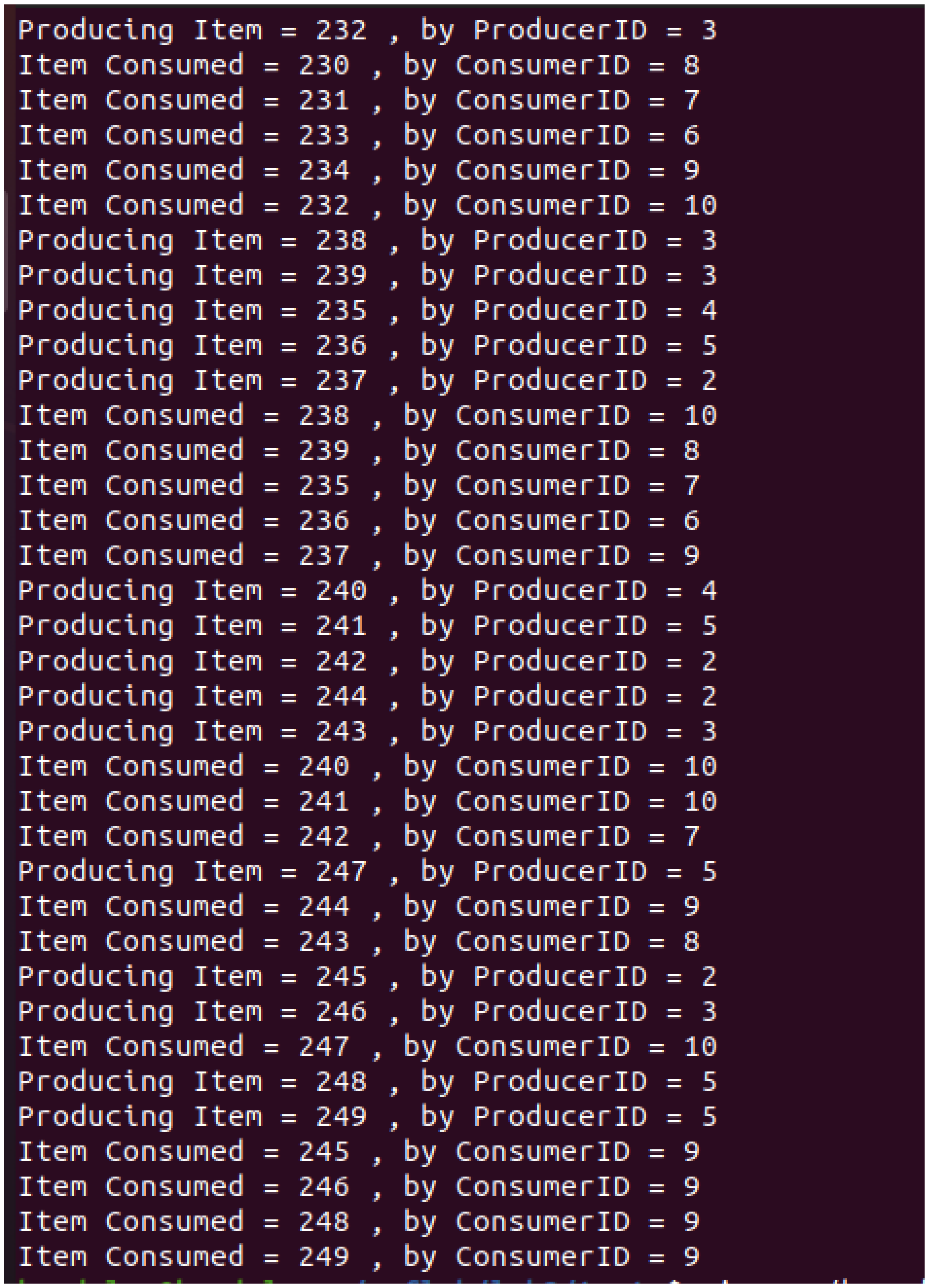
1) Matrix Multiplication Program

I observe that our threads library improves the efficiency of the multiplication and the time reduces. However the pthreads are not able to improve the efficiency much, due to a higher context switch time taken.
The below graph depicts the performance of the program, run over with different threads count.



The exact time values are available in the Data section down below.

2) Bounded Buffer Program



```
Producing Item = 232 , by ProducerID = 3
Item Consumed = 230 , by ConsumerID = 8
Item Consumed = 231 , by ConsumerID = 7
Item Consumed = 233 , by ConsumerID = 6
Item Consumed = 234 , by ConsumerID = 9
Item Consumed = 232 , by ConsumerID = 10
Producing Item = 238 , by ProducerID = 3
Producing Item = 239 , by ProducerID = 3
Producing Item = 235 , by ProducerID = 4
Producing Item = 236 , by ProducerID = 5
Producing Item = 237 , by ProducerID = 2
Item Consumed = 238 , by ConsumerID = 10
Item Consumed = 239 , by ConsumerID = 8
Item Consumed = 235 , by ConsumerID = 7
Item Consumed = 236 , by ConsumerID = 6
Item Consumed = 237 , by ConsumerID = 9
Producing Item = 240 , by ProducerID = 4
Producing Item = 241 , by ProducerID = 5
Producing Item = 242 , by ProducerID = 2
Producing Item = 244 , by ProducerID = 2
Producing Item = 243 , by ProducerID = 3
Item Consumed = 240 , by ConsumerID = 10
Item Consumed = 241 , by ConsumerID = 10
Item Consumed = 242 , by ConsumerID = 7
Producing Item = 247 , by ProducerID = 5
Item Consumed = 244 , by ConsumerID = 9
Item Consumed = 243 , by ConsumerID = 8
Producing Item = 245 , by ProducerID = 2
Producing Item = 246 , by ProducerID = 3
Item Consumed = 247 , by ConsumerID = 10
Producing Item = 248 , by ProducerID = 5
Producing Item = 249 , by ProducerID = 5
Item Consumed = 245 , by ConsumerID = 9
Item Consumed = 246 , by ConsumerID = 9
Item Consumed = 248 , by ConsumerID = 9
Item Consumed = 249 , by ConsumerID = 9
```

I see, that the program executes correctly and each thread are able to do their job.

User-level threads are a way of performing multiple tasks on a single cpu. It does not deals with parallel computation and processing on multiple cpus. Switching context between user-level threads is very low cost since only a section of the registers and pointers changes and rest of the process level information remains the same. Tasks which are more I/O bound can more effectively use user-level threads since, while some I/O happens the process can switch to some other task and perform it while the I/O is handled by the machine.
User-level threads hence allow a better performance and better utilisation of the CPU.

Data

Time Vs Number of Threads for MyThreads,Pthreads implementation.

Number Of Threads	MyThreads Time(s)	Pthread Time (s)
1	1.782811	1.723093
2	1.714043	1.749451
4	1.705645	1.805561
5	1.731921	1.782731
10	1.73722	1.773981
15	1.638468	1.80363
20	1.685283	1.78116
25	1.691102	1.732088
30	1.652357	1.727581
40	1.615571	1.82879

This above data is for matrix multiplication of size 600x600.

Further Improvements

One, can make the following improvements in the library to make it more robust:

1. Allocate a large amount of memory from heap and provide it to each thread as its stack space. This will also avoid corruption of stack in case a thread uses a large stack memory inside its execution. This way we can also release the stack memory once a thread finishes its job.
2. Use of queue in locks to avoid busy wait.
3. Using point 1, we can also remove our limitation of a max number of threads and can possibly reach a very large number of threads.

References :

1. Asad’s Implemenation of User-thread library
2. Bhargav’s Implementation of User-thread library
3. Signals IPC - Wikipedia
4. Signals in Linux document
5. Make manual
6. GeeksForGeeks (for different articles related to lab)