INDEX

Acknowledgment

I have completed my assignment by learning from the following sources

1 Given Support Materials
   a) https://hpc-tutorials.llnl.gov/posix/

2 https://www.geeksforgeeks.org/multithreading-c-2/

3 https://www.geeksforgeeks.org/use-posix-semaphores-c/

4 https://www.ibm.com/docs/en/i/7.2?topic=lf-atol-atoll-convert-character-string-long-long-long-integer

5 https://iitd-plos.github.io/os/2020/lec/l25.html

6 https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

7 https://sites.cs.ucsb.edu/~chris/teaching/cs170/projects/proj2.html

8 https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

9 https://toodle.cs.huji.ac.il/cs14/file.php/67808/demo_jmp.c

10 http://www.sis.pitt.edu/jjoshi/courses/IS2620/Fall14/Lecture4.pdf

11 http://www.csl.mtu.edu/cs4411.ck/common/Coroutines.pdf

12 http://poincare.matf.bg.ac.rs/~ivana/courses/ps/sistemi_knjige/pomocno/apue/APUE/0201433079/ch10lev1sec15.html

13 https://github.com/bhaargav006/User-Thread-Library/commit/12bd0d66c541ccf222c3e2787da4f47af6c329a9

14 https://homepage.cs.uiowa.edu/~dwjones/opsys/threads/

15 https://www.cse.iitd.ac.in/~suban/csl373/demo.c

16 https://gist.github.com/elliott-beach/c52300a30db2c6dc744a5c9e5e107b22

17 https://github.com/asad82/User-Level-Thread-Library

18 Other Books and study materials

19 My previous submitted assignments

NOTE –

I have not copied anything from these sites, such as programs,

Only have taken the idea of the general syntax and working of the programme

I have taken the same code as my previous assignments in the matrix multiplication part

Pre-Requisite

How To Make Executable and Run Programme

1) Unzip File at location x
2) Open Terminal and change directory to that location x
3) Now enter command make.
   This will create Executables
4) Enter Command make run to run the executables bounded buffer and matrix and make runTest to run all the test
5) Alternatively, you can type
   export LD_LIBRARY_PATH=${pwd} //give the path of the present working directory.
6) Then type executable name with the arguments
7) For plot, type make plot

Structure Of Programme

Program is made up of directory structure which includes

1) src – Source Folder, It contains all necessary .c files to make object files
   a) test.c files – contains all test file which shows that our thread library function is working properly
   b) matmul.c - contains calling code of matrix multiplication
   c) matrix.c/matrixorig.c - contains code of matrix multiplication
   d) printMatrix - contains code to print matrix
   e) time.c – code to note the time of an instance when Time() is called
2) obj – contains object file created from above code
3) inc – contains header file useful at the time of linking
4) lib – contains thread libraries and queue library
5) data – contains running data from which plot can be made

Threads/Critical Section

Threads are simpler programmes by dividing a complex programme into subparts and executing them parallel or serially. Threads share almost everything except PC, SP, Register and some other specific part

In parallel, several resources of the CPU are used simultaneously to make better performance.

In serial, when a particular thread is blocked for some reason, another thread is scheduled to gain performance.

It is based on the concept to reduce CPU idle time and use resources as much as possible

Types of Thread

Kernel level thread - Threads are made by OS and run for better performance; extra context switching time is needed for moving from one thread to another.

User-level thread – Threads are made by USERS. Need less time for moving from one thread to another as no kernel switching is required. OS may use these to identify and create threads and use them for parallel processing.

Context Switch

When threads are running in parallel or serial with different scheduling, it may be the case when some process is using some resource/data, and at the same time, another process may want the same.
This may lead to a race condition.

To avoid these conditions, we may use Semaphores, Mutex (Locks), Condition variables to ensure concurrency.

Essential Programming Points

SigJMP

It is used with three construct

1. sigsetjmp -It is used to save context like register, PC, SP, etc., into jmp_buffer. It also saves signal masks. Return 0 on success and 2nd parameter of longjmp if called through it. Mask action is not defined in normal setjmp.
2. jmp_buf – It is an integer array that stores context info. It is usually of size 8.
3. siglongjmp – It is used to go to the point where sigsetjmp was set and continue from there; also, it restores context. It returns one on success after completing all steps of jmp.

Setitimer

It contains three-parameter – Timer type (real-time, CPU time, etc.), new struct timer to be implemented, store struct of old-timer which was executing till this point for this signal. It returns 0 on success and -1 on error.

Sigaction

It contains three-parameter – signal type (decides which signal to be noted), Struct sigaction, old struct sigaction to store previous struct sigaction. It returns 0 on success and -1 on error.

Struct sigaction

It contains four-parameter – handler (a function which is called when the signal is caught), the mask of signal (used to store mask of the signal which cannot affect the working of this handler with the help of sigset), flags (denotes the behaviour of signal, NODEFER is used to not

to hold other signals when the current handler is invoked), the action of queued signals.

Sig Set Action

1. sigaddset – it is used to add a signal to mask of set
2. sigemptyset-used to the empty mask set
3. sigprocmask- define the behaviour of signal mask of the set (take same set or union in case of the function call)

Struct itimerval

Contains four-parameter – microsecond of interval and first-timer and second for interval and first-timer

Note, usually union of signal mask set of previous function and mask set of defined function and calling signal is taken as mask set for new function and previous mask set is restored when the function ends.

Result

1) I have made a user Thread Library that performs all significant functions like any other Thread Library
2) The library works for a single thread (test 1), as shown below
3) In this, it goes into thread and print something and then go out of thread

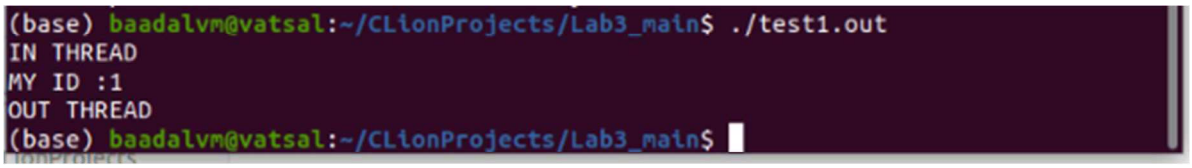

Fig1. Test 1

4) The library works for Multiple threads (test 2), as shown below
5) In this, the programme runs each thread in a FIFO manner (RR scheduling), prints something, and then goes out of the thread.

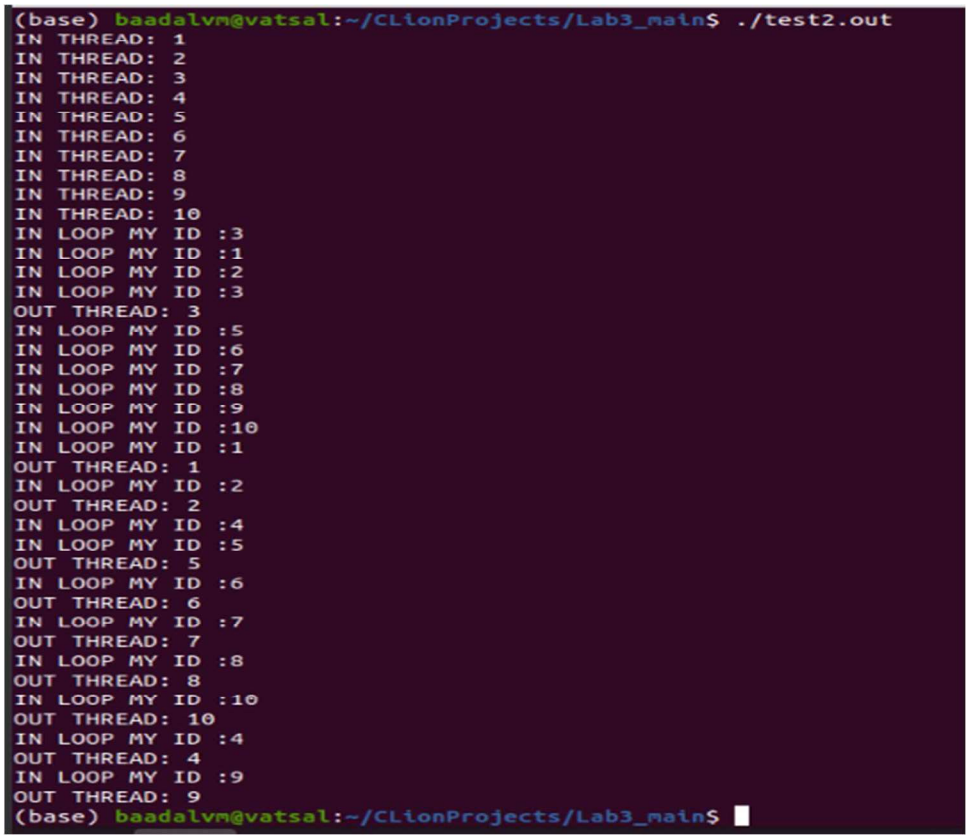

Fig2. Test 2

6) The library works for Multiple threads with yield (test 3), as shown below
7) In this, the programme runs each thread in a FIFO manner (RR scheduling) and prints something and then yields and then runs again, after all thread has run and then go out of thread.

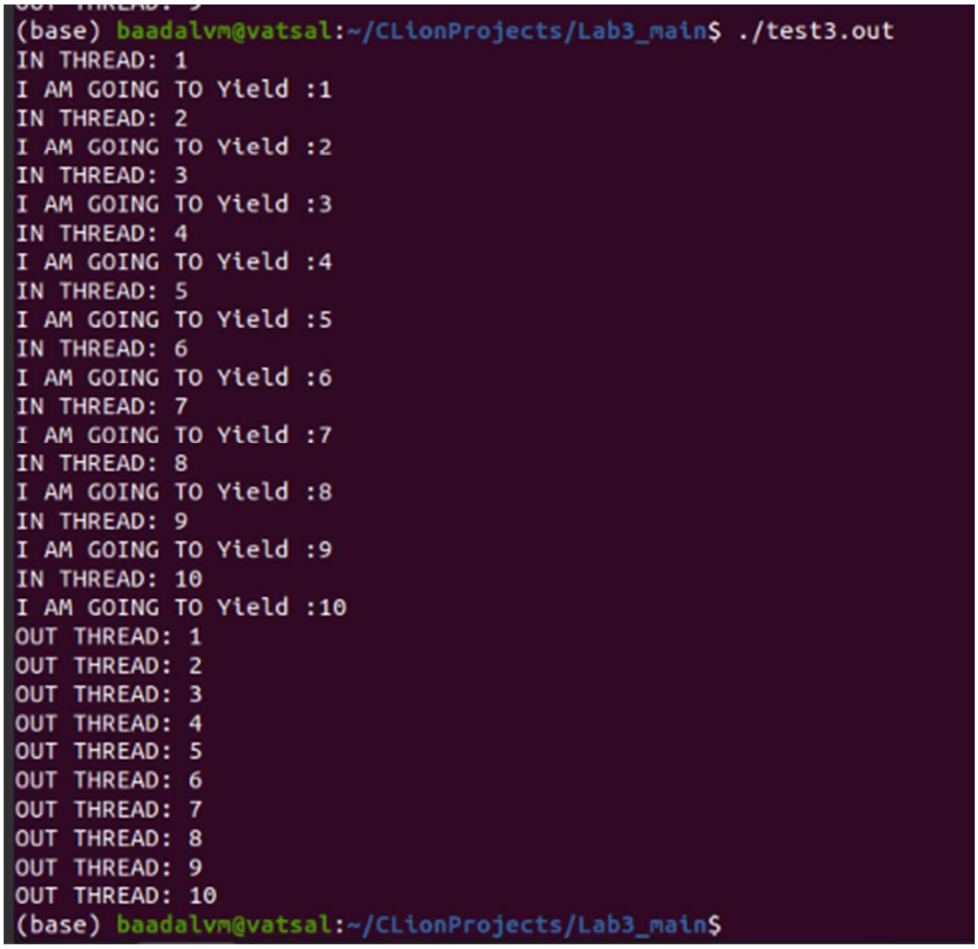

Fig3. Test 3

8) The library works for Multiple threads with lock (test 4), as shown below
9) In this, the programme runs each thread in a FIFO manner (RR scheduling) and acquire the lock, do its work, leave the lock, print something, and then go out of the thread.

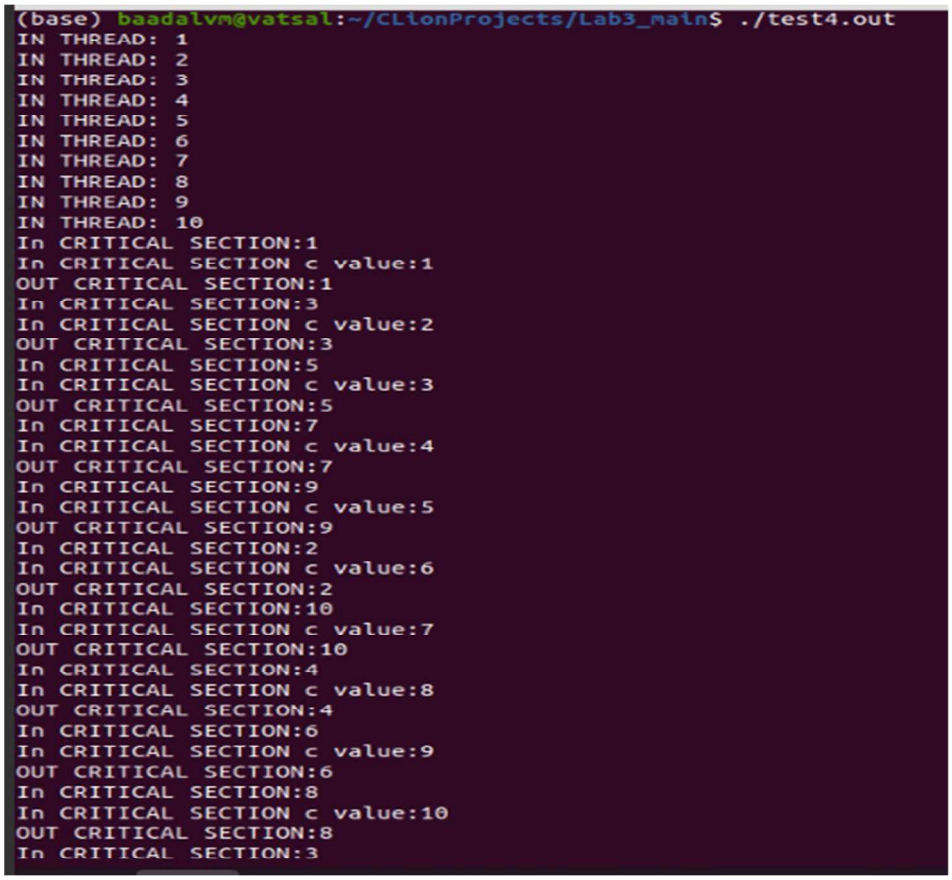

Fig4. Test 4

10) The library works for Multiple threads with lock and yield(test 5), as shown below
11) In this, the programme run each thread in FIFO manner (RR scheduling) and some thread acquire the lock, do its work, and then yield and leave lock when run again and print something and then go out of thread
12) The library works for Multiple threads with Condition Variable and lock (test 6), as shown below
13) In this, a programme run two thread in FIFO manner (RR scheduling) and 1st thread call wait and 2nd thread call signal while 1st thread waits for a quantum timeout the 2nd thread runs
14) The library works for Multiple threads with lock, Condition Variable and yield(test 7), as shown below

15) In this, programme run two thread in FIFO manner (RR scheduling) and 1st thread call wait and 2nd thread call signal. When 1st thread gets a chance, it yields.

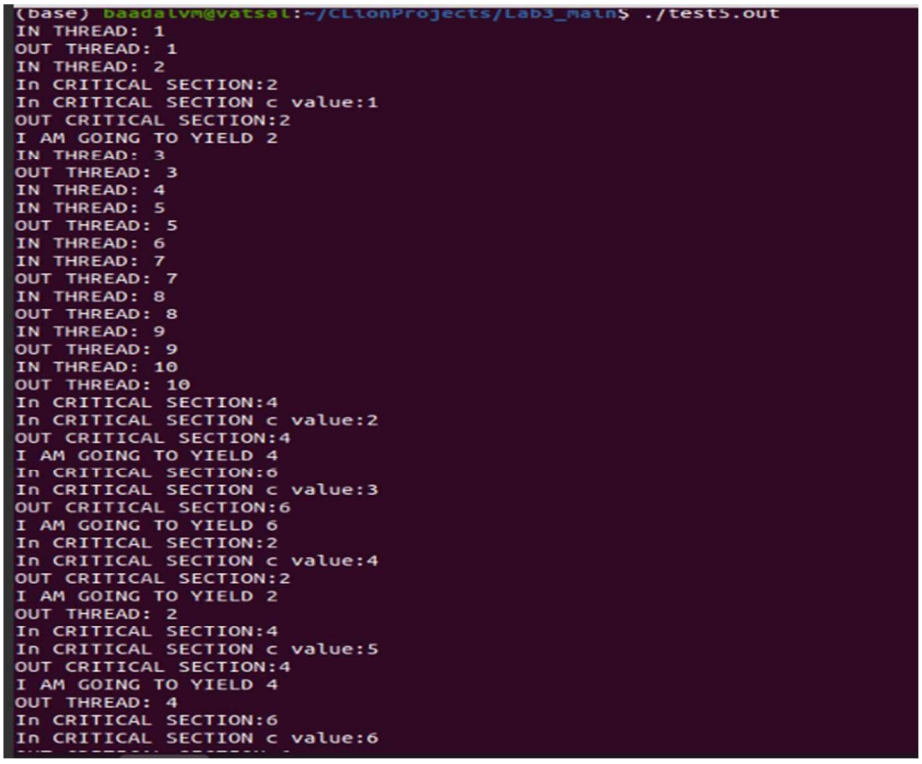

Fig5. Test 5

Fig6. Test 6

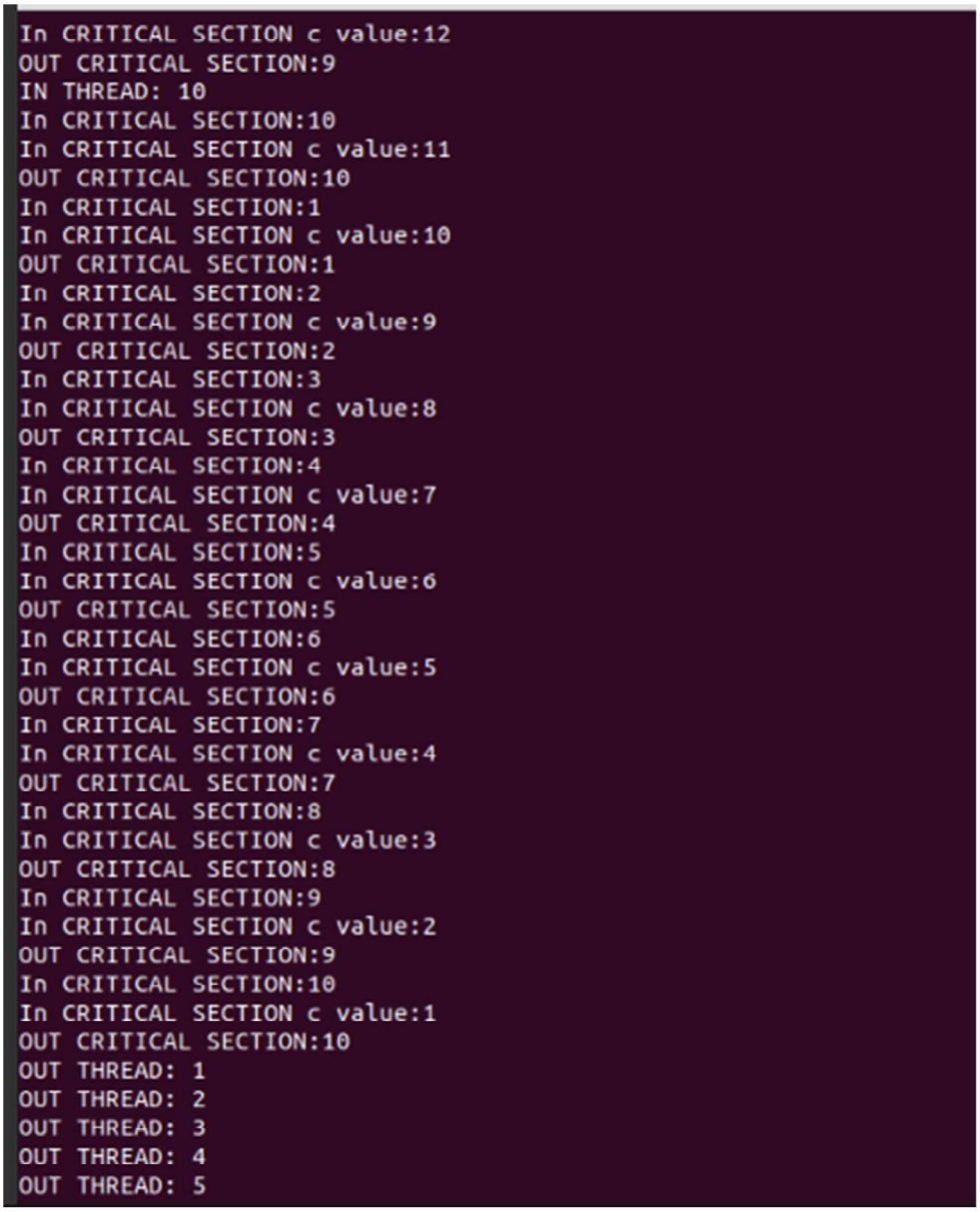

Fig7. Test 7

16) We have also implemented matrix multiplication using threads
17) In this, we have divided the number of rows into different threads and run the programme with our thread library and pthread library, both, to compare the running time of both, also; we have compared it with the number of threads and timing of matrix multiplication
18) Running time of matrix of the size of 8, 64,128,256,512 is shown below

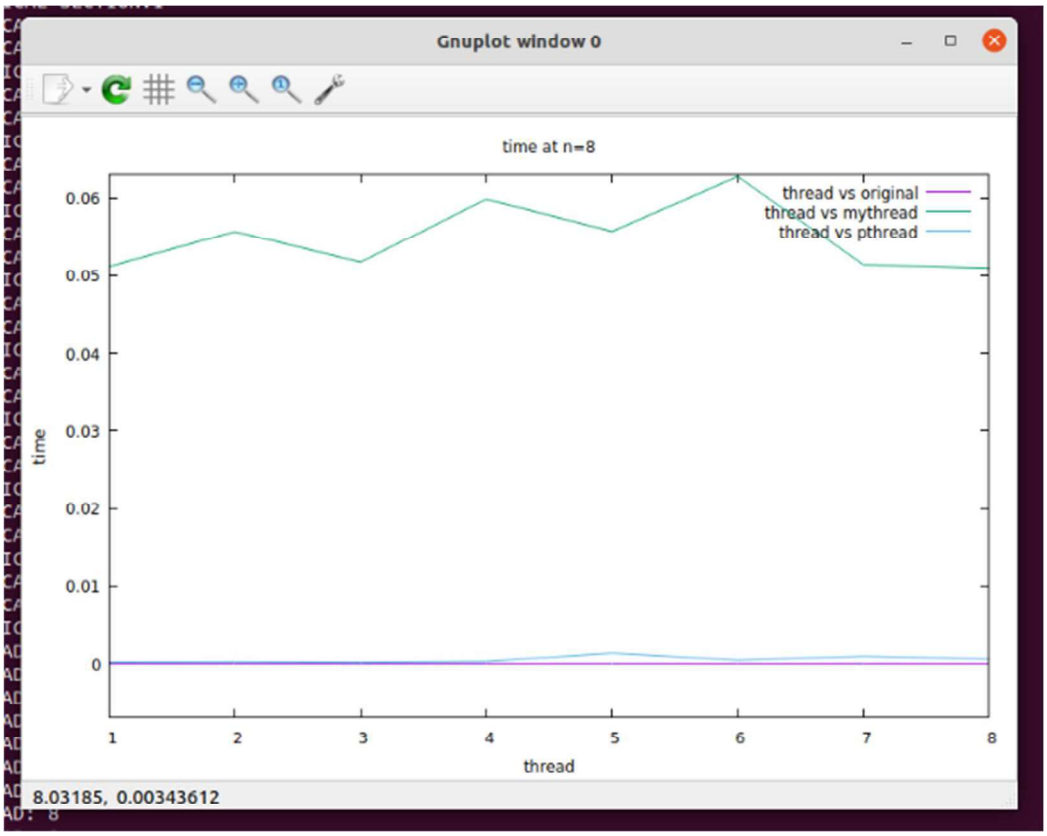

Fig8. Test matrix size 8

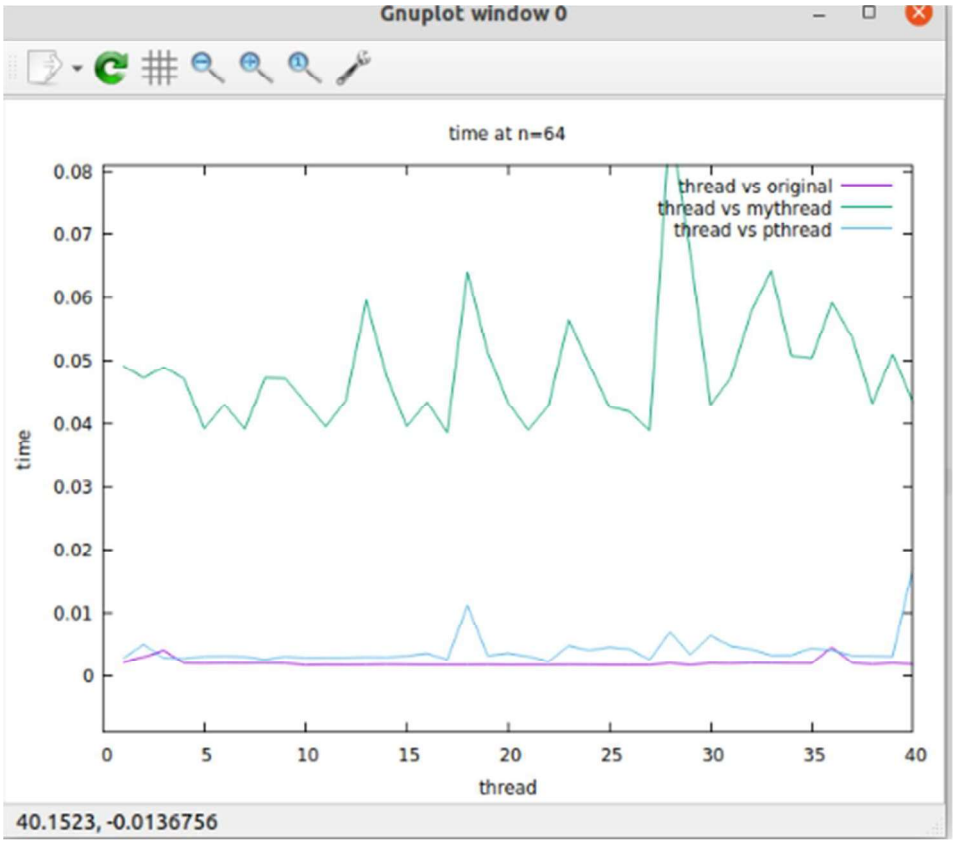

Fig9. Test matrix size 64

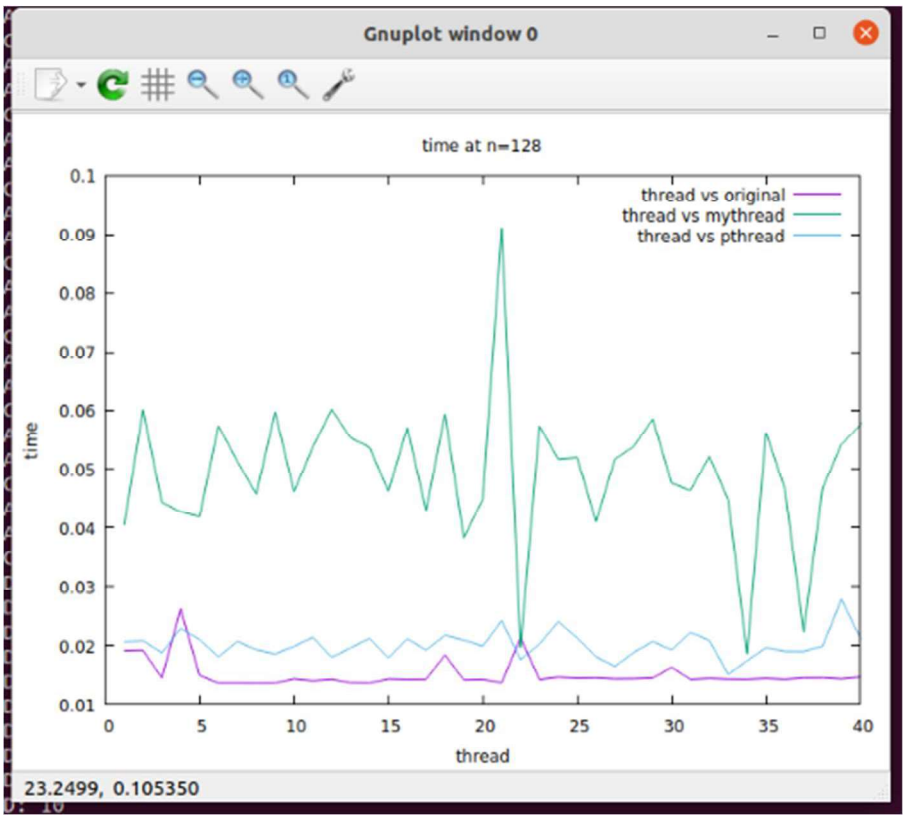

Fig10. Test matrix size 128

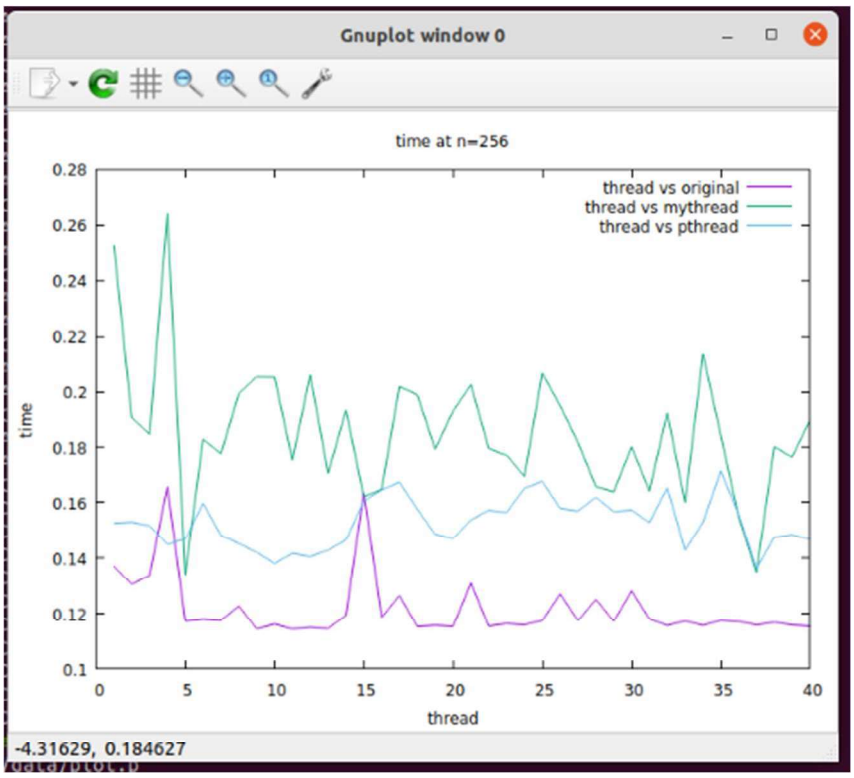

Fig11. Test matrix size 256

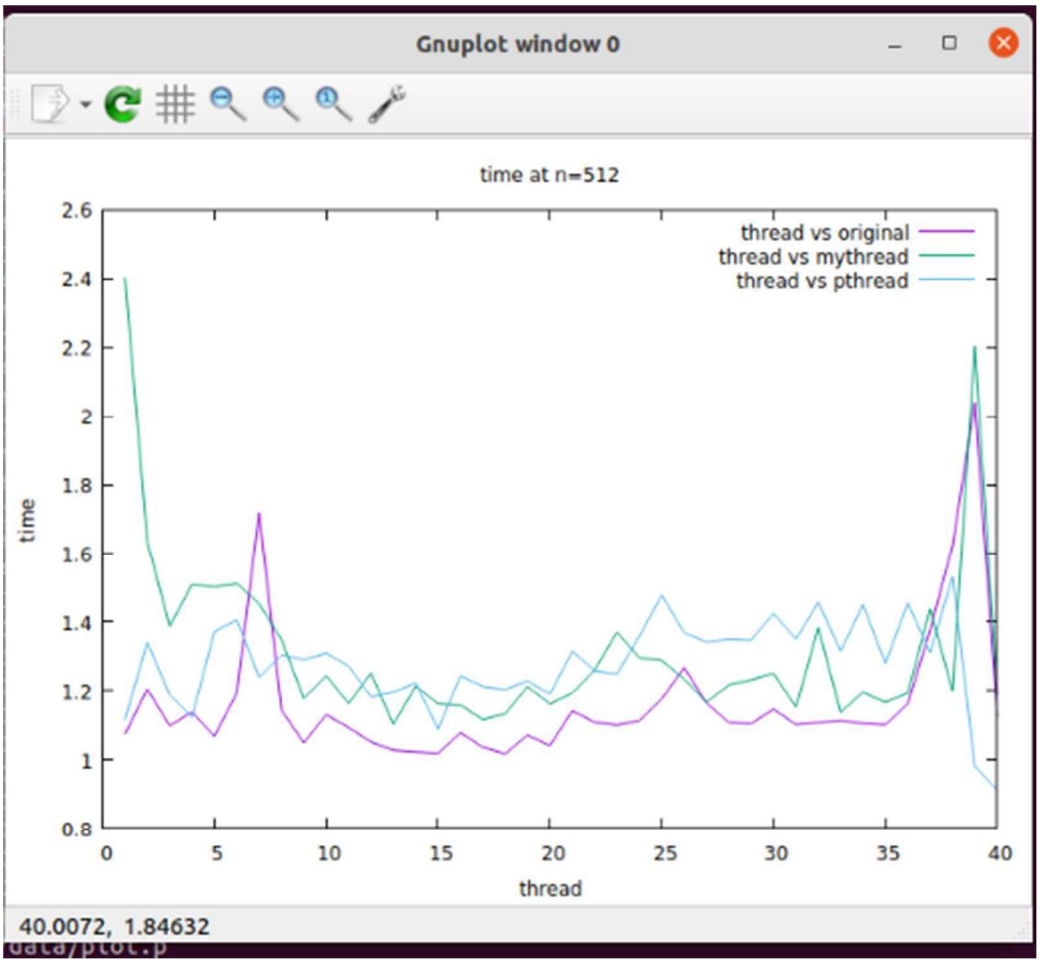

Fig11. Test matrix size 512

19) We can clearly see as the size of the matrix increases; our thread library is giving the same performance as the pthread library
20) The above case is for loaded operation; our library perform much better in case of non loaded operation
21) Also, as the size of the matrix increases, our library is performing better in comparison to the original matrix multiplication
22) At thread size between 15-20 (16), our, as well as pthread library, works best.
23) For threads of size 1 and 2, our thread performs worst
24) Also, we have compared context switch time of pthread as well as my thread for different n and thread values as given below

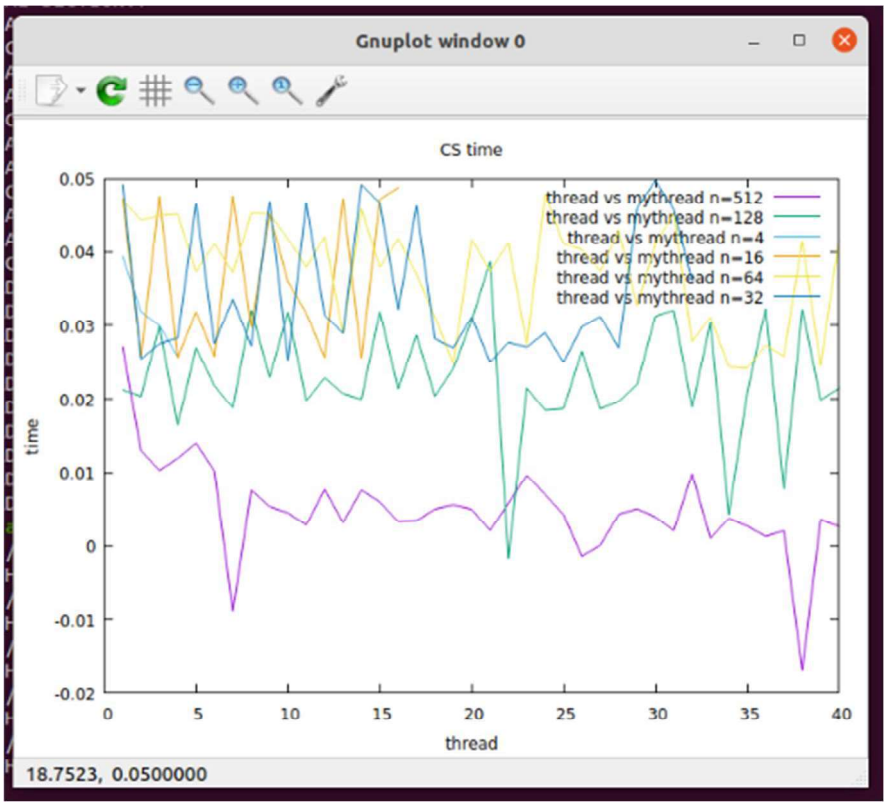

Fig12. My thread CS time

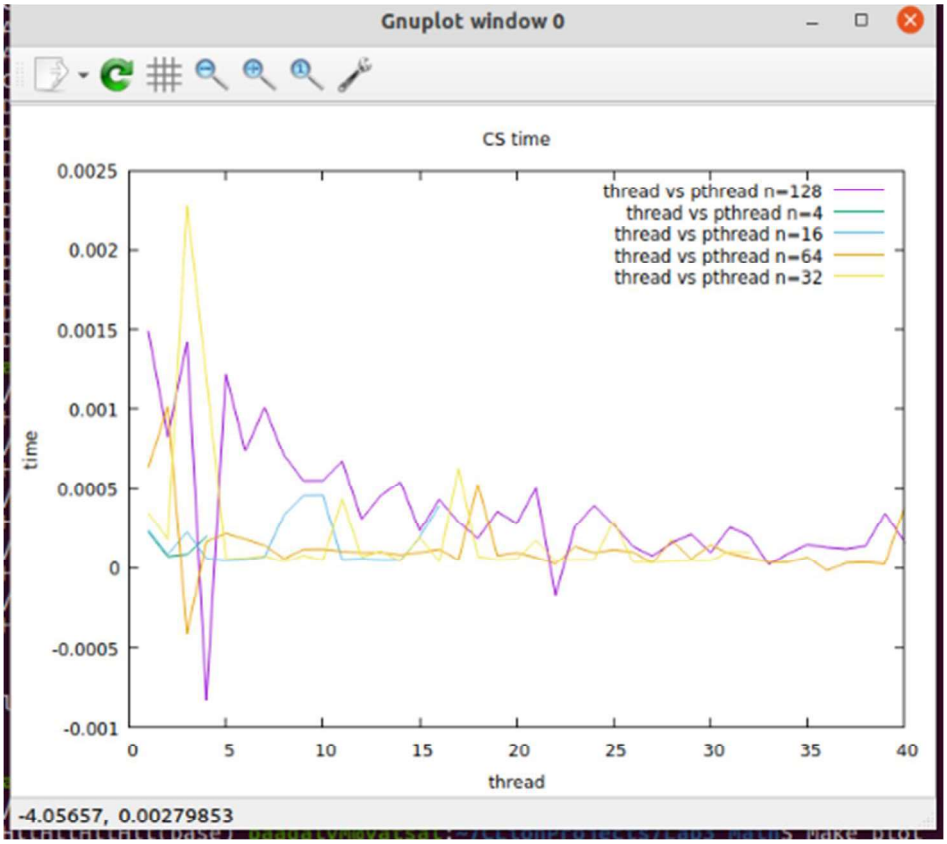

Fig12. pthread CS time

25) From above, we found out that our thread has CS time as 0.03s when there is significant load when there is no thread deletion as per above data and 0.006s in random time as for the large matrix size, parallel processing come into a role and small matrix size we do not get required time to calculate the nice average
26) All the above graphs have the worst running case
27) We considered the above value for n=128 and thread count between 15 to 30
28) For the same range, we found pthread have CS time is 0.0003s
29) Our thread has significant CS time in comparison to the pthread library
30) We have also used the bounded buffer problem to show that our condition variable and locks are working fine
31) We have used multiple producers and consumers which can have n buffer and each stop when 10*n items are produced or consumed

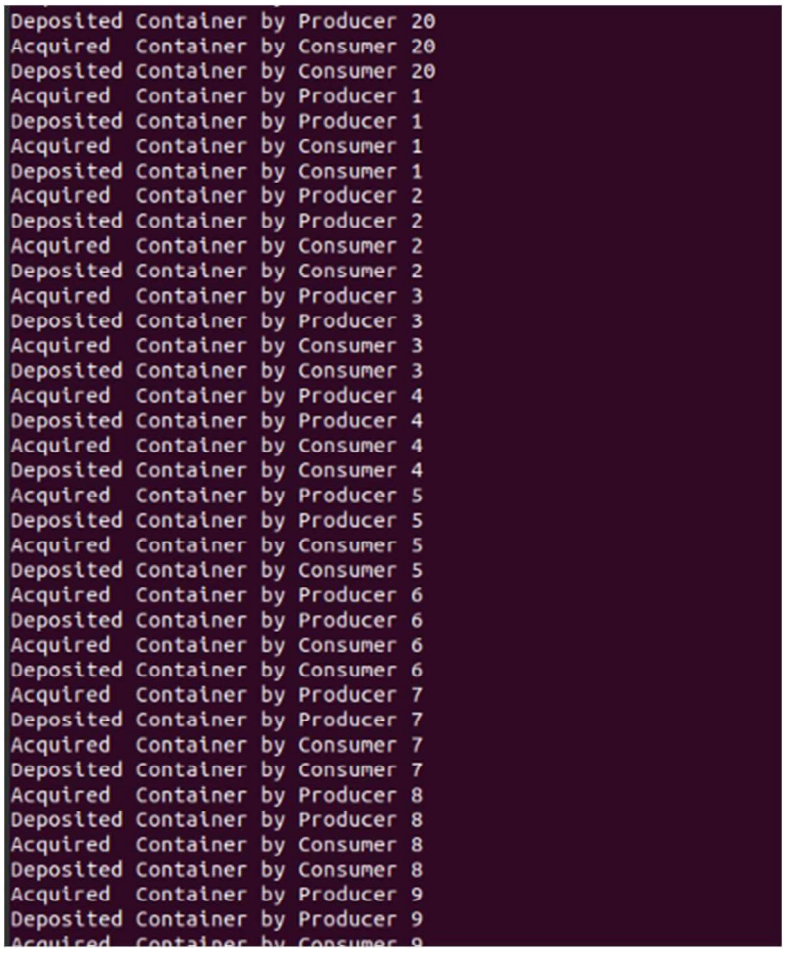

Fig13. Bounded Buffer

Conclusion

1. We are working on the concept that the scheduler is one thread. The scheduler's work is to be in the ready queue and run only when a context switch is needed. In the case of RR, it runs when quantum ends.
2. States of a process are denoted by Running, Ready, Terminated, Scheduler, Blocked, Finished
3. When the first time thread create is called, a scheduler is also initialized with all the queues it needs.
4. Pointer of TCB of every thread is kept in rdy (ready) queue, and the finished process is kept in ft queue.
5. When every thread in the ready queue is ended, it waits for 50ms to 100ms before deleting thread data. Note, thus join condition must be called in this time only. This feature time can be extended or can be turned off also.
6. Our programme works on a large number of threads ranging more than 10K in most of cases, given the constraint that memory does not overflow.
7. Our programme runs on the RR scheduler. Every time a Context switch is done, a thread context is saved, and a new thread context is restored. This is done with the help of siglongjmp and sigsetjmp and jmp_buf
8. A timer is maintained, which counts the time of CPU execution(not real-time) and give SIGVTALRM when the timer cranks.
9. Each TCB has its tid, its state, its JMP_buf, which store SP and PC, also it has function and arguments and attributes related to the thread.
10. Note, a programme jmp_buffer is protected and encrypted in Linux, so when we store new PC and SP in it manually, so we also have to encrypt them. Encryption is done with the help of the

assembly inline function, which is based on AT&T. In this, operands are reversed; that is, the result comes in the end. We use

asm volatile("xor %%fs:0x30,%0\n""rol $0x11,%0\n": "=g" (ret): "0" (addr));

In this volatile means do not compiler optimization (do not change position and always use it)
%% fs is register to hold special data structure of OS %%fs:0x30 = fs[0x30] is used to indicate process environment block. Rol is rotating on the left with carry bits, =g denoting result can be register, memory, etc., and 0 denotes input operand %0 as register.

Note encrypting feature is called mangle

11. Jmp_buffer is an integer array that holds at position six from 0 a stack pointer and the position seven a PC in x86; the Base pointer is stored at position 1
12. The lock is guaranteed with the help of an ASM code

    asm volatile("lock; cmpxchg %2, %0 \n":"+m" (*original), "+a"(old) :"r"(new));

    cmpxchg compare %2 with a series register like EAX, AX, AL and if equal %0 is stored in %2; if not equal %2 is loaded in eax(made non 0), +a is used to store old in eax, +m is used to take original as memory address which read then write and r is register for read
13. Thcount is used in our programme to maintain thread count,
    Schfun is sigacation struct , itime in time interval struct, inCpu stores running thread TCB information.
14. MyThreadSwitch initializes timer and other things.
15. CS is a function that gains attention when a timer is called, and it calls context switch and
16. ContextSwitch help in context switch and stopping the timer when rdy length is 0

17. Rest functions are the same as pthread. Get is to get a lock, and leave is to leave the lock.
18. Note, the lock is a mutex, i.e. the person get mutex, can only leave it; otherwise, deadlock can occur
19. MythreadJoin work on tid while other functions work on thread pointer
20. Wrap is a function that passes an argument to function and call myThreadExit automatically when the function ends.
21. All the test cases are working fine
    Test 1-Single thread
    Test 2-Multiple thread
    Test 3- Multiple thread+yield
    Test 4- Multiple thread+lock
    Test 5- Multiple thread+lock+yield
    Test 6- Multiple thread+CV
    Test 7- Multiple thread+CV+yield
22. NODEFER has no particular use in our programme but can be used in case of some other signal is triggered
23. We have used a setitimer for alarm function.
24. We have used the ConsumerProducer problem in which many producers and consumers are sharing N buffer and stop when n buffers are produced
25. We have also used matrix multiplication, in which the number of rows is divided for threads
26. We have created a shared dynamic library of user-level thread
27. We are calculating Average context switch time in my thread matrix multiplication by formula

$$\frac{my\ thread\ time - original\ time}{1 + \frac{my\ thread\ time}{quantum\ time}}$$

28. We are calculating Average context switch time in my pthread – by scheduling them in FIFO order in matrix multiplication by formula

$$\frac{\text{pthread time - original time}}{\text{no of threads}}$$

29. We have found the average context switch time as 0.03s in my thread when there is no thread deletion and 0.005s when there is thread deletion, and 0.0003s in pthread.
30. We have used a value of n=128 and a thread value of t=16 for this
31. We have found that our thread library performs best at thread count = 16
32. This has been seen that alarming timer after all thread completion to delete threads will be better in performance when there is load in comparison to not deleting threads
33. Our thread library performs as better as pthread when there is less load.

Future Scope

1) Serial implementation can be made faster by avoiding costlier operations like multiplication by using addition instead of it in matrix multiplication
2) We can implement the thread library to make it smarter by exponential increasing its timer when there is less load
3) Also, we can make it smart by increasing the timer when there is more load so that deadlocks do not occur
4) We can modify our lock and condition variable to avoid more deadlock situation
5) We can add extra functions like detect future threads to enhance the performance of our library
6) Lots of CS time can be saved by using some code optimization like eliminating redundant expression or calling lees function, or calling less heavier expression.