**User level thread library**

1. **contains the following files:**
   1. main.c
   2. my\_pthread\_t.c
   3. my\_pthread\_t.h
   4. my\_mutex\_and\_que.c
   5. my\_mutex\_and\_que.h
   6. readme.pdf
   7. logOutputFinal (very verbose output with lot of printfs to check accuracy)
   8. benchmarkPthread.c (to test against the standard pthread lib)
   9. benchmarkSeq.c (to test the sequential version of the program (no threads))
2. **How to run:** 
   1. Compile the code using the following command:

***gcc main.c my\_pthread\_t.c my\_mutex\_and\_que.c -lrt***

* 1. Execute the code with the below command

***./a.out***

1. **How the scheduler works:**

* The scheduler uses a multilevel priority queue for scheduling.
* There are three levels of ready queues and there is a priority associated with each level (0,1 and 2) respectively. 0 being the highest priority and 2 being the lowest. There is a time slice associated with each priority level too. (Everything discussed here is in terms of milliseconds)

Queue 0- 50 ms

Queue 1-100 ms

Queue 2-200 ms

* Every newly created thread is added to the highest priority queue to start with.
* The scheduler (also the signal handler for the SIGALRM) picks the next thread to be scheduled from the highest priority queue that is not empty after every quanta expiration. Lower priority queues are not checked until the higher priority queues are empty.
* If a thread finishes its assigned time quanta in a particular queue (without yielding or exiting), then the priority of that thread is decreased and it is pushed to the next lower priority queue (Unless it was in the lowest priority queue, in which case it remains there and added to the end of that queue again).
* Yields: If a thread yields before its quanta expires, it is allowed to be stayed in the same queue.
* Waiting Queue: If a thread tries to lock the mutex and the mutex is already locked for some other thread, the requesting thread is removed from the scheduling queue (equivalent to ready queue) and is put in a waiting queue. The threads in the waiting queue are added back to the highest priority scheduling queue if the mutex they were waiting on is unlocked. This shift happens in the very next scheduler call after the time slice expires of the thread which unlocked the mutex (or it yields or exits in the same time slice).
* There is also a maintenance cycle that is executed after every 400ms (during the next scheduler call) to do some interesting stuff discussed below.

1. There are a couple of issues that can arise and in the following section I address the strategies I have used to deal/remedy these issues:
2. **Problem -Starvation**: If the system is getting a lot of high priority threads continuously, there is chance that a low priority thread can starve out in a low priority queue.

**Solution implemented: Thread Aging**- During every maintenance cycle, the scheduler scans the lowest priority queue and checks if there is a thread that has been sitting in that queue for more than 500MS. All such threads are moved to the highest priority level. This gives them a chance to get some CPU time and if lucky, they can finish during this quantum. If they use up the time slice in the highest priority queue and do not finish, they will again bubble down to the lowest priority queue after some time (until they again spend 500MS there and the maintenance cycle comes to the rescue again). This prevents their indefinite starvation.

1. **Problem -Priority Inversion**: The problem can arise if a high priority thread is waiting on a resource (mutex in this case) that is assigned to a low priority process. If a medium priority process comes in and locks the mutex as soon as it is freed by the low priority process, the high priority process will again have to wait in the waiting queue.

**Solution implemented: Priority Inheritance**- When a thread tries to lock a mutex that is locked by another thread, we record the priority of the thread that is trying to lock the mutex. During each maintenance cycle, we check if the priority of the thread that currently holds the mutex is lower than the priority of the highest priority thread that is waiting on the mutex. If yes, the thread that currently holds the mutex temporarily inherits the priority of the highest priority thread that is waiting on the mutex. This gives the lower priority thread hogging the mutex a chance to run and finish execution and free the mutex for the use of the high priority thread in the waiting queue. Since if the mutex is unlocked, the wait queue is scanned and threads are moved to the highest priority ready queue before the scheduling of the next thread, there is no chance that a medium priority process can come in between and lock the mutex.

1. **Problem- Smart voluntary yield attack**: A smart programmer can keep the priority of a process high by performing voluntary yields just before its time quantum expires in the highest priority queue. If the programmer knows (or somehow approximates) the time slice of the highest priority queue (if the quantum for the queue is 50ms and the thread yields after executing 49ms every time, it can stay in the highest priority queue for as long as it wants), a thread can be programmed to yield just before the expiration of the time slice, and since it yielded, its priority will not be decreased.

**Solution implemented:**  **Total Yield time allowed in a queue**: If a thread is yielding continuously, we check the total time it has spent in the queue it is yielding in. If this time is more than a specific amount (0.002ms here), than the priority of the thread is decreased even if it is yielding.

1. **Performance comparisons (benchmarks):**The performance (with respect to time) of the user level thread library is compared with the standard pthread library and also against the sequential version of the same code.

The comparisons can be made as follows:

1. To get execution time using user level thread library run the following command:

***time ./a.out***

1. To get the time using standard pthread library, run the following 2 commands in the sequence given:

***gcc benchmarkPthread.c -o benchmarkPthread.out -lpthread***

***time ./benchmarkPthread.out***

1. To get the time of the sequential version of the code run the following two commands in sequence given:

***gcc benchmarkSeq.c -o benchmarkSeq.out***

***time ./benchmarkSeq.out***

The following table gives the results of the above testing:

|  |  |  |
| --- | --- | --- |
| User level thread lib | Pthread library | Sequential version |
| 6452 ms | 6397 ms | 6470 ms |