CS-518: Assignment 0

USER THREAD LIBRARY

Tanya Sharma (tds104) and Sanchay Kanade (sk2656)

1. INTRODUCTION

This project implements a user defined thread library containing thread functions, synchronization methods and scheduler functions to imitate the functioning of the Linux POSIX thread library. User defined thread libraries can be used to create efficient threads with customized functions.

2. DATA STRUCTURES

- typedef uint **mypthread_t;**
  Unsigned integer variable that represents a single thread in our system. It is used to identify threads through their thread IDs.

- typedef struct **threadControlBlock**{};
  The thread control block describes all significant parameters that define a thread in the system. These are described below:
  - **tID**: the thread ID is the unique identifier of a thread
  - **tContext**: holds the thread context during execution
  - **tPriority**: priority value assigned to the thread, on the basis of which it is placed in a queue when we use multi-level feedback queues
  - **join_id:** contains ID of the thread for which this thread has called join() and is waiting to finish so that it can start executing
  - **value_ptr:**
  - **mutexID**: holds the ID of the mutex held by the thread
  - **timeQuantum**: time quantum during shortest job first scheduling

- typedef struct contextNode{}
  - **threadBlock**: a node representing a thread
  - **next**: a pointer pointing to the address of the next thread in the queue

- typedef enum states{}
  - **NONE**: no change in state, current thread is executing as expected
  - **TIMER**: timer has counted up to a certain time and now the scheduler will switch out current thread for the next one
  - **YIELD**: thread yields its time to another thread because it has completed its job and still has runtime available
  - **EXITING**: a thread has completed execution and is leaving the running queue and the system
  - **JOINING**: thread has called join() on another thread and is waiting for it to complete execution
  - **INITIALTHREAD**: first thread in the running queue
  - **WAITING**: thread is waiting to acquire a mutex

- typedef struct mypthread_mutex_t{}
  - **tID**: ID of the thread that holds the mutex
  - **lockedState**: stores status of the mutex lock (0 when unlocked and 1 when locked)
  - **isWaiting**: flag to check if a thread is waiting for the mutex to be free
  - **mutexID**: unique identifier of the current mutex

- typedef struct exitNode{}
  - **tID**: thread ID of the exiting thread
  - **value_ptr**: stores the return value of the exiting thread
  - **next**: pointer to next node

- typedef struct queueType{}
  - **front**: stores the head of the queue
  - **back**: stores the tail of the queue

- typedef struct plevels{}
  - **rqs**: array of queues for multi level feedback queues where each index denotes a different priority.

3. GLOBAL VARIABLES

Variable name	Description
runningQueue	Multilevel running queue that stores threads as node and where each index indicates a priority level. Higher the index value, lower the priority of the thread. For scheduling algorithms not using multilevel queues, we use the 0th index to use just a single queue for scheduling
waitingQueue	Wait queue storing nodes waiting to acquire a mutex
current	Current thread that is being scheduled
joinQueue	Join queue storing context of threads that have used join() on another thread and are waiting for them to finish and return
exitList	Stores details about the exiting thread
status	Status of the current thread, to handle different scheduling states, set to the INITIALTHREAD which starts scheduling with the first thread being added to the running queue
firstThread	Flag to indicate whether the current thread is the first thread of not
Cycles	We are periodically counting number of scheduling cycles to provide feedback and increase priority of low priority threads which tend to be starved
itimer	Timer variable used to set timer off and call scheduler after a fixed duration
threadCount	Counts the number of running threads
mutexCount	Counts the number of mutexes currently engaged by a thread
SCHED_TYPE	Enumerates different scheduling algorithms
TYPE	Handles which scheduler is being used for a particular program

4. THREAD FUNCTIONS

- **mypthread_create**

  This function when called, creates a new thread. It allocates a new stack for this thread and initializes it by setting values for the stack size, base pointer, and flags. It also creates a new context and thread control block for the thread, which are also initialized in this method. The very first thread that is created is treated as a special case, since it triggers the initialization of each of the running, waiting and join queues. The first thread is then pushed into the running queue. Each subsequent thread that is created, always starts with the highest priority and is pushed to the end of the ready/running queue.

- **mypthread_yield**

  The current thread that was allotted some run time, has either finished its execution before the time has elapsed or is voluntarily surrendering its runtime to be used by other threads. This function changes the current thread's state from running to ready. If the thread has completed all its execution and is ready to exit, then the method sets a new context of the next thread scheduled to run. If the thread has not finished its execution and is being switched out due to a timer interrupt or some other condition, then it swaps contexts with the next thread scheduled to run and still saves the current thread's context to be used when it is scheduled the next time.

- **mypthread_exit**

  The current thread has finished its execution or is being terminated and is going to exit the scheduler processes by being deallocated. This method uses a special data structure called the exitNode to aid the exit process of the thread and ensure that its value pointer is preserved. The thread is dequeued from the running/ready queue and we free up all thread specific dynamic memory associated with the exiting thread.

- **mypthread_join**

  This method handles a thread that is waiting for another thread to finish executing or exit. We preserve the value pointer of the exiting thread and deallocate the dynamic memory that was created by the joining thread.

5. SYNCHRONIZATION FUNCTIONS

- **mypthread_mutex_init**

  Thread synchronization method to initialize the mutex lock. It receives a pointer to the mutex and a configuration mutex, and returns 0 if it is possible for the mutex to be created. For the purpose of our mutex implementation, we initialize the mutex lockedState (to 0 implying that the mutex is currently unlocked), thread ID to a non-permissible thread ID value, a flag that checks if there is a thread waiting to acquire the mutex to false and the mutex ID is set to the sequence in which this mutex was created.

- **mypthread_mutex_lock**

  Uses built in test-and-set atomic function to test if the mutex can acquire the lock using the variable lockedState (where 1 denotes that the mutex is locked). If the mutex is already acquired by a different thread, then the current thread is pushed onto the waiting queue and context is switched back to the current thread at the scheduler. If the thread is able to acquire the mutex then it the mutex thread ID reflects the thread's ID with a 0 returned on success.

- **mypthread_mutex_unlock**

  Before a mutex can be unlocked we test multiple scenarios to see if this operation is even valid or permissible by then current thread that has called this function. To unlock the mutex, it must not already be unlocked, and the thread ID stored in the mutex must match the thread ID of the thread that has called the unlock function, If these conditions are fulfilled, we then update the mutex's data to indicate that it has been unlocked. If there is another thread waiting to acquire the mutex, the first thread that was in the waiting queue, is put into the scheduler running queue, ready to be allotted runtime since it can now acquire the mutex lock.

- **mypthread_mutex_destroy**

  Before we decide to deallocate space, we check if the mutex fulfills certain conditions that will prevent us from accidentally destroying incorrect mutexes. The mutex must not be in the locked state because that would mean that is is being used by a thread to synchronize its access to a shared resource and destroying the mutex at that time would lead to data consistency and integrity issues. If there is another thread waiting to acquire the mutex, in the waiting queue, then the mutex should not be destroyed because it is still needed by another thread during its execution. If these conditions are fulfilled, we can go ahead with deallocation of the dynamic memory allotted for a mutex during runtime.

6. SCHEDULER OPERATION

MULTI LEVEL FEEDBACK QUEUE (MLFQ)

The multi-level feedback queue mechanism implemented in this project uses an array of running queues where each index indicates a priority level. Priority level decreases with increasing index values. The queue with the highest priority is found at the 0^{th} index of our multi-level queue data structure. When new threads are created, they are assigned the highest priority and are pushed into the top-most queue. The algorithm schedules threads in the decreasing order of their priority which in our case is convenient because it is in increasing order of the indices of our multi-level queue array. At each level, there is a simple round robin mechanism implemented with a different time quantum used for each level of priority. We start with the smallest time quantum for our top-most priority queue and gradually increase the time quantum of execution depending on the corresponding priority level queue the thread belongs to. We use the timer class to set the time interval using the following formula:

Time Quantum= (10+10*t)*1000 (microseconds) where t-> thread priority

This formula was chosen because it gradually increases the running time of a thread. Each time a thread is scheduled to run and does not finish execution, its priority is increases and it is pushed

down to the queue below it. We also have a thread starvation prevention function. This function counts how many cycles of thread execution have occurred and when this count reaches a threshold level, it flushes out the threads at the lowest priority level to the highest priority level queue so that they can obtain some runtime and not starve. If these threads, still don't finish executing then they make their way down the priority levels as normal. In this way, decreasing thread priority is dependent on how many times a thread has been scheduled and starvation of old threads is prevented by periodically allocating some run time to low priority threads by increasing their priority.

ROUND ROBIN (RR)

Round Robin queue scheduling is a simple and straightforward implementation where each thread is allotted some predetermined time quantum of time to run, and their execution is done sequentially. The time quantum chosen in our project is 25ms because it has been observed that the algorithm does not suffer from the expense of too many context switches at this interval and because the time is sufficient for most threads to finish execution. If a thread is exiting, a new thread context is set for execution and if the thread has not finished executing, we swap out its context for the context of the next thread scheduled to run.

PREEMPTIVE SHORTEST JOB FIRST (PSJF)

In the preemptive shortest job first algorithm, whenever a new thread enters the ready queue, we choose the thread with the shortest running time from the entire queue and run it. This involves knowledge of thread execution times or burst times which we don't know when a thread first enters the queue, but we can make an approximation of its running time by observing how long it was running the previous time it was scheduled. We start by running a thread for one time quantum and increasing the time quantum it should be scheduled for, each time it has been given runtime. Parallelly, we have a count for how long a thread has been waiting. Whenever a thread is executed, the waiting time of all other threads increments and is only reset once it has been allotted some runtime. If there is a tie between the minimum waiting time of two threads, we choose the one that has been waiting for a longer time to manage aging threads.

7. HELPER FUNCTIONS

- **Queue operations**

  To manage each of the queue data structures used in our program to handle threads states such as the running, waiting and join queue, we implement queue operations to improve queue reusability and modularity. We use the queue enqueue operation to place a thread at the back of a queue and the queue dequeue operation to extract a thread from the front of the queue in a LIFO (last in first out) manner.

- **Thread Operations**

  Helper functions to get current thread count and to deallocate dynamically allocated memory for threads and their context. We also implement a thread state handler function that handles each thread state. It returns 1 if the thread is the current thread and enables execution to continue for it. A TIMER state prompts the scheduler to decrement the priority of the current running thread and swap it's context with the next thread to be scheduled. The current thread is also placed in the next priority queue. A YIELD state means that a

thread wants to voluntary allow another thread to use its remaining runtime and swaps context with the next thread to be run. An EXITING state prompts the program to free context of the exiting thread and deallocate any dynamically allotted memory associated with the thread and its execution. A JOINING state implies that a thread is waiting for another thread to finish execution and a WAITING state means that the thread is waiting to acquire a mutex and will be placed at the end of the waiting queue.

- **Starvation Prevention Operation**

  A thread in a multi-level feedback queue may be at risk of being starved if its priority is very low and it cannot obtain any runtime to execute. To prevent starvation, this method keeps a global cycle count of the MLFQ algorithm execution and if the count hits a certain threshold, it increases the priority of threads that have been waiting for the longest time.

8. ANALYSIS

External_Calc:

Time taken by each algorithm (in micro seconds)

Thread Count	Kernel threads	RR		SJF	MLFQ		
Time quantum (micro seconds)		25000	2000		(10+10t)*1000	(25+25t)*1000	(25+25t)
2	4751	579	509	538	575	569	571
6	2526	543	596	572	592	524	551
10	2384	529	618	599	583	554	564
25	3172	518	566	580	580	550	527
50	3200	581	572	593	541	538	552
75	3220	543	528	619	526	537	586
100	3104	534	544	594	556	535	641

As we can observe, kernel threads have the best performance given that the sums that are calculating is at a higher order of magnitude than the user threads. Out of different user thread scheduling algorithms tested for different time quantum, we can conclude that the Multi-Level Scheduling Queue has the best performance when we set the time quantum to (25+25t)*1000. SJF algorithm has the worst performance and performance gets worse with an increase in the number of threads which could be because older threads have a tendency to starve.

	Time	Sum
Kernel thread		2400138723
RR	25000	458453348
	2000	165960244
SJF		153304562
MLFQ	(10+10t)*1000	201889607
	(25+25t)*1000	281763342
	(25+25t)	182666184

Parallel calc:

Time taken by each algorithm (in micro seconds)

Thread Count	Kernel threads	RR		PSJF	MLFQ	
Time quantum (micro seconds)		6	600		25+25t	(6 +6*t)*100
2	1911	4001	3457	4503	3441	4164
6	714	3375	3219	3542	3399	3520
10	460	3601	3447	3527	3358	3373
25	330	3420	3221	3644	3473	3409
50	233	3410	3730	3532	3472	3570
75	230	3238	3322	4105	3383	3334
100	178	3248	3512	4493	3387	3259

In this benchmark test, we observe that the best performance for the parallel calc benchmark test is the the Multi-Level Feedback Queue algorithm for the smaller time quantum (25+25t). Similar to the kernel threads, the algorithm speeds up as we increase the number of threads. The Round Robin algorithm also follows closely behind in terms of performance.

Vector_Multiply:

Time taken by each algorithm (in micro seconds)

Thread Count	Kernel threads	RR		PSJF	MLFQ	
Time quantum (micro seconds)		6	600		(6+6*t)*100	(6+6t)
2	140	24	24	23	32	36
6	245	33	26	25	24	36
10	229	38	36	26	25	29
25	187	44	43	44	44	57
50	313	53	39	38	38	43
75	400	58	48	48	48	38
100	413	51	34	40	32	37

For the vector multiplication benchmark test, we observe that the algorithm with the shortest runtime is the Preemptive Shortest Job First algorithm. PSJF increases more smoothly as compared to the other algorithms and imitates kernel thread behavior in doing so. The Round Robin algorithm for a larger time quantum (600 micro seconds) also gives high performance and is also an alternative that can be used.

Parallel Sum Square:

Thread Count	Kernel threads	RR		PSJF	MLFQ	
Time quantum (micro seconds)		6	600		(6+6t)	(6+6t)*100
2	23	29	28	28	29	38
6	15	29	29	29	28	29
10	8	31	36	30	28	29
25	7	29	38	41	55	29
50	5	32	38	36	48	29
75	6	29	43	45	41	29
100	7	29	34	40	42	29

From the observations we can make about the run time of the parallel sum square benchmark test, we can safely say that the Multi-Level Feedback queue algorithm with the time quantum (6+6t)*100 micro seconds provides the most consistent results. The Round Robin algorithm for the smaller time quantum (6 micro seconds) is also comparable in performance.