CS-518 - UTHREADS

Ashwin Patil (aap327) | Sammed Admuthe (ssa180)

1 Introduction

There are mainly two types of threads - User Level Threads and Kernel Level Threads. Kernel Level Threads are created and managed by the OS itself. While on the other hand, User Level Threads are created and managed by USER. In general, User Level Threads are faster and smaller compared to Kernel Level Threads. While User Level Threads are lightweight and have some advantages, there are also some disadvantages. For example, User Level Threads cannot take advantage of multicore CPUs and boost the performance. Then why/when should we go for User Level Threads? The answer is simple. User Level Threads are easy to create and manage. Since they are not using/dependent on the kernel for synchronization and scheduling, you can opt for any synchronization/scheduling mechanism you wish for. As User Level Threads are essentially just another library, we can run them on any OS we wish. Another reason to go for User Level Threads is that switching between User Level Threads is much faster and does not require any OS intervention. In this document, we have explained methods and algorithms implemented for our own User Level Thread library. We have also evaluated performance of some scheduling algorithms against various benchmarks.

2 Procedures

2.1 Threads

In this section we have explained all the methods/procedures used in implementing our own User Level Thread library (mypthread library).

2.1.1 mypthread_create

For every mypthread_create call, a Thread Control Block is initialized with respective Id, status, priority and context of thread. Whenever the first call to mypthread_create is made, a running queue is initialized which holds the identifier of these threads. The context creation for Main thread and scheduler's thread in addition to periodic timer setup is also done during the first call to mypthread_create.

2.1.2 mypthread_join

In this method, if the thread passed as argument to function mypthread_join has completed execution then, a context switch is made to scheduler.
In case if the same thread has not completed its execution, the joining thread (context that calls mypthread_join) is blocked and its thread identifier is attached to the blocking thread(waitCalledBy). Finally the context switch is made to the scheduler.

2.1.3 mypthread_exit

In this method, the status of the currently running thread is changed to COMPLETE. A previously blocked thread during join call (indicated by the waitCalledBy parameter of the current running thread) is also added to the Running queue.

2.1.4 mypthread_yield

In this method, the status of the currently running thread would be changed to READY state. This follows a context switch from the current thread to the scheduler's context.

2.2 Mutex

In this section, we have explained the implementation of mutexes using in built instructions such as test_and_set().

2.2.1 mypthread_mutex_init

Mypthread_mutex_init method initializes data structure for mutex. A call to this method initializes mutex waiting queue, lock and owner variables.

2.2.2 mypthread_mutex_lock

This method uses spin lock mechanism using __atomic_test_and_set to provide locking mechanism for shared mutex variables. Once a lock is acquired by thread, successive threads which try to acquire lock would be added into the waiting queue (changing status of thread to WAIT state)unless the lock is released by the owner thread.

SPECIAL CASE - A variable qOpDone is used to make sure that threads are added to the waiting queue irrespective of periodic clock interrupts.

2.2.3 mypthread_mutex_unlock

This method dequeues threads for waiting queues and enqueues them into running Queue at the same time updating the WAIT status of thread to READY state. A variable qOpDone handles interrupts caused by a periodic clock during the unlock phase.

2.2.4 mypthread_mutex_destroy

Mypthread_mutext_destory deallocates dynamically created memory for mutex variables and data structures.

3 Scheduling Algorithms

In this section, we have explained various thread scheduling algorithms used to implement our own User Level Threads library.

3.1 Round Robin - RR

Threads in the running queue are scheduled to run for a time quantum (5 micro-seconds). Threads at front of queue are dequeued and enqueued at the rear queue after they are done executing for defined quantum of time. In case, the concerned threads are in BLOCKED, WAITED or COMPLETED status, they are dequeued from the running queue so that they won't get run time. This follows a context switch from scheduler to the thread that was at queues front before dequeue.

Round Robin with time quantum 5 microseconds is ideal for most benchmarks. Increasing quantum time beyond 5 microseconds results increased run time due to starvation of threads. Whereas decreasing quantum below 5 microseconds also results in higher run time due to constant context switching.

3.2 Multilevel Feedback Queue - MLFQ

Design:-

MLFQ utilizes 4 queues with increasing time quantum for each queue. A priority parameter in the thread control block determines which queue the thread currently belongs to. A thread can have priority from 0 through 3. Thread with priority = 0 resides in mlfq_Q[0], thread with priority = 1 resides in mlfq_Q[1] and so on. The 0th queue represents highest priority and has lower quantum time while 3rd queue represents lowest priority queue and has higher quantum.
Time quantum for each queue is determined by QUANTUM * (Queue Number+1):-

Time quantum for 1st queue mlfq_Q[0] = QUANTUM * 1.
Time quantum for 2nd queue mlfq_Q[1] = QUANTUM * 2.
Time quantum for 3rd queue mlfq_Q[2] = QUANTUM * 3.
Time quantum for 4th queue mlfq_Q[3] = QUANTUM * 4.

Each of the 4 queues uses Round Robin Scheduling Policy.

Implementation:-

Initially all the threads are added in the highest priority queue i.e. queue 0.
Round Robin Policy is implemented at each level of the queue.

When to demote threads? (Decrease priority)

- When the threads execute for the entire quantum of time, its priority decreases and it is pushed to queue with lower priority.
- The quantum count in TCB ensures that waiting threads don't get starved and thread waiting for a long time gets runtime.

When to promote threads? (Increase priority):-

Threads in the lowest priority queue are promoted to the highest priority queue i.e 0th queue after a definite time interval (1000 microseconds). Thus, **priority inversion** can be achieved using the above strategy.

3.3 Preemptive Shortest Job First - PSJF

Design:-
PSJF utilizes the quantumCount parameter of thread’s TCB to keep track of the number of quantums a particular thread has executed. A thread with lower quantum count gets higher priority over others and thus is assumed to have a shorter remaining execution time. If the threads are waiting for a resource in the waiting queue they would have the same quantum count for their stay in the waiting queue(no increment, no decrement of quantumCount). Thus when these threads are scheduled later, they would have higher priority than one that is running for a long time.

Sorting the running queue based on quantum count(Insertion sort) ensures shorter jobs are at the front of the queue. The time complexity of insertion if O(n) since the threads in queue are already sorted.

Implementation :-
When to increment quantumCount? - If the threads runs for the entire time quantum without getting into BLOCKED, COMPLETED or WAIT state, increment quantumCount.

When to decrement quantumCount? - If a thread is being dequeued because it has been added to the waiting queue, decrement quantumCount and later increment it before context switch (effectively keep quantumCount untouched for that particular thread). This would ensure that the longer **waiting process** gets priority in case of **tie-breakers** (threads with similar quantum count) when waiting thread returns to the running queue**.**

4 Structs

In this section we have explained structs that we have used to store all the information about our user level threads.

4.1 TCB - Thread Control Block

```
typedef struct threadControlBlock
{
        int Id;                 // unique thread id
        int status;             // thread status
        int priority;           // thread priority
        ucontext_t context;     // thread context including stack, stack size, stack ptr, flags,uc_link
        uint waitCalledBy;      // thread who has called join on current thread
        int quantumCount;       // thread quantum count used to evaluate shortest job
} tcb;
```

4.2 Mutex

```
typedef struct mypthread_mutex_t
{
        uint lock;                      // if lock = 1 we have acquired the lock otherwise we wait
        mypthread_t owner;              // thread id of the thread which owns the lock currently
        struct Queue* waitQueue;        // queue containing threads waiting to acquire lock
} mypthread_mutex_t;
```

4.3 Queue

```
struct Queue {
   uint front;          // pointer to front element of the queue
   uint rear;           // pointer to the rear of the queue
   uint size;           // size of the queue
   unsigned capacity; // maximum elements we can keep in the queue
   uint* array;         // array used to store mypthread_t thread identifier
};
```

5 Performance Evaluation

In this section, we have compared the performance of our user level threads against standard POSIX thread library pthread. This comparison is done using various benchmarking programs.

5.1 Vector Multiplication

A simple vector multiplication operation performed using mutex. As there are no I/O bound operations, this benchmark is CPU bound. In the vector_multiply function call, we first acquire mutex on the variable and then proceed with the operations. As a result, with increasing threads, we get low performance as all the threads are waiting for the lock.

			mypthread		
Time Quantum	Thread Count	pthread (micro seconds)	RR (micro seconds)	MLFQ (micro seconds)	PSJF (micro seconds)
2	5	256	41	39	39
	10	268	43	43	42
	100	310	62	63	63
5	5	224	43	44	42
	10	241	50	45	44
	100	305	65	64	63
10	5	260	45	44	45
	10	279	55	48	47
	100	302	61	62	59

5.2 Parallel Calculation

In this benchmark, we perform CPU bound operations. When we call a parallel_cal function, first we perform a multiplication operation without acquiring mutex, then we do an addition operation by acquiring mutex. In the case of Linux pThread library, threads are running on different cores of the CPU, we observe that turnaround time gradually decreases as we go on increasing the number of threads. But in the case of myPthread library, we do not make use of multiple cores of the CPU, resulting in a rather similar performance even if we increase the total number of threads.

			mypthread		
Time Quantum	Thread Count	pthread (micro seconds)	RR (micro seconds)	MLFQ (micro seconds)	PSJF (micro seconds)
2	5	435	1928	1934	1916
	10	283	1925	1919	1919
	100	223	1922	1927	1911
5	5	595	1911	1911	1913
	10	270	1922	1913	1924
	100	188	1912	1876	1915
10	5	443	1921	1920	1916
	10	351	1946	1917	1914
	100	200	1915	1914	1915

5.3 External Calculation

In this benchmark, we perform I/O bound operations. As we know that I/O bound operations are slower than CPU bound operations, linux pThread library performs consistently even if the total number of threads increases.

			mypthread		
Time Quantum	Thread Count	pthread (micro seconds)	RR (micro seconds)	MLFQ (micro seconds)	PSJF (micro seconds)
2	5	1433	409	402	409
	10	1519	405	405	413
	100	1524	402	404	455
5	5	1433	405	405	401
	10	1519	408	399	410
	100	1524	426	423	407
10	5	1433	425	406	408
	10	1519	409	405	410
	100	1524	427	410	409

5.4 Atomic Operations (Banking Transactions)

In this benchmark, we test the mutex performance. This benchmark tries to imitate bank balance with deposit and withdrawal transactions. We assume a variable with some initial balance. Then we perform additions and subtractions on this initial balance. If mutex works correctly, we should get a consistent answer. Every even thread is performing addition operation and every odd thread is performing subtraction operations. As we only perform CPU bound operations, we get similar performance.

			mypthread		
Time Quantum	Thread Count	pthread (micro seconds)	RR (micro seconds)	MLFQ (micro seconds)	PSJF (micro seconds)
2	5	253	44	37	36
	10	276	46	38	38
	100	280	49	43	45
5	5	253	44	41	38
	10	276	45	40	40
	100	280	51	44	51
10	5	253	43	43	43
	10	276	44	44	45
	100	280	52	49	55

5.5 Dining Philosophers Problem

The purpose of choosing this benchmark is to avoid possible deadlock due to mutual exclusion and circular wait. The mutex lock and unlock mechanism ensures synchronization and resolve Dining Philosopher Problems.

			mypthread		
Time Quantum	Thread Count	pthread (micro seconds)	RR (micro seconds)	MLFQ (micro seconds)	PSJF (micro seconds)
2	5	12	12	11	14
	10	12	13	12	15
	100	15	14	14	13
5	5	12	13	12	11
	10	12	13	15	10
	100	15	12	16	13
10	5	12	11	12	10
	10	12	13	14	13
	100	15	16	16	14

5.6 Last Element of Fibonacci Sequence

Since this benchmark utilizes recursive stack, a context switch is involved at every stage of recursion. All the operations involved in this benchmark are CPU bound operations.

			mypthread		
Time Quantum	Thread Count	pthread (micro seconds)	RR (micro seconds)	MLFQ (micro seconds)	PSJF (micro seconds)
2	5	12	14	12	13
	10	13	14	13	13
	100	15	13	14	13
5	5	12	14	13	13
	10	13	13	11	13
	100	15	11	14	11
10	5	12	12	16	12
	10	13	13	14	13
	100	15	15	11	14

5.7 Sum of Prime Numbers

This benchmark involves lightweight computations. We calculate the sum of N prime numbers. All the operations in this benchmark are CPU bound operations. Since this benchmark uses nested for loops, this would ensure that mutex lock and unlock operations are performed and the results are consistent.

			mypthread		
Time Quantum	Thread Count	pthread (micro seconds)	RR (micro seconds)	MLFQ (micro seconds)	PSJF (micro seconds)
2	5	0	2	3	2
	10	0	3	2	3
	100	0	2	2	2
5	5	0	3	2	3
	10	0	3	2	3
	100	0	2	2	2
10	5	0	3	3	2
	10	0	2	3	2
	100	0	3	3	3

6 Analysis

Benchmarks for Vector multiplication and Parallel Calculation demonstrate better response time for mypthread library. However with increase in the number of threads the response time is lowered. It is found that the quantum of 2 and 5 is ideal for these benchmarks.

In the case of Parallel Calculation, the POSIX pthread library performs well across all time quanta compared to mypthread, this is due to the fact that parallelism is better utilized in this benchmark by multiple cores compared to single core.

The Posix pthread library majorly outperforms mypthread across multiple benchmarks as it utilizes multiple cores.

In case of benchmarks where computation is not expensive like Dining Philosopher, Atomic Operations (Banking Transactions), Sum of Prime lower time quantum like 2 milliseconds, 5 milliseconds produces better response time. Scheduling algorithms have a comparatively low effect on response time for **higher quantum** as threads are **short lived.** However, with an increase in the number of threads, the cost of **context switching** outweighs and thus produces output with higher response time.