Rutgers, The State University Of New Jersey

DEPARTMENT OF COMPUTER SCIENCE

RUTGERS

THE STATE UNIVERSITY
OF NEW JERSEY

CS518 - Design of Operating Systems

Assignment 0: Uthreads

Parth Khandelwal (pk684)

Lakshit Pant (lp749)

iLab machine: ilab2

Introduction

In this report, the execution of thread libraries, policies inside them, and scheduling mechanisms have been explained. Implementation of a user-level thread (uthread) library - similar to the POSIX thread (pthread) library - is done. 3 schedulers, i.e. Round-Robin, Pre-emptive Shortest Job First (or STCF) scheduler, and Multilevel Feedback Queue are used along with the policies in the uthread library to determine which threads need to be run. A detailed explanation of each function, data structure, scheduling stages, and additional data values is given in this report.

Implementation of functions and detailed scheduler logic

- **void initialize():**

  In this function firstly we allocated memory for all the runqueues. Then we registered the signal handler pointing to the timerHandler function whenever a timer interrupt occurs. After successfully allocating memory we initialized the threadTerminated and thread_exitValue of threads which will be useful to indicate whether a thread is running or not and store the information of threads on exiting respectively. At last we set up the scheduler context by initializing it and modifying it to call the schedule function.

- **int mypthread_create():**

  This function first executes the helper function initialize() and then creates a thread. It allocates stack space for the thread, initializes the thread control block as well as its context. It also creates a single main thread by allocating space to its stack and initializing its thread control block and gives it priority 4.

- **int mypthread_yield():**

mypthread_yield() gives other user level threads the possession of the CPU. It saves the context of the current thread to its TCB and context switches to the scheduler. The threadYield value changes in the function which indicates to the scheduler that the current thread has yielded the possession voluntarily before its time slice ends.

- **void mypthread_exit():**

This function uses the thread ID of the current thread to deallocate the memory generated for this thread by using freeCurrentThread() function after the said thread finishes its execution. The value_ptr is assigned to the thread_returnValues[] array mapped by the thread ID of the thread. It allocates NULL value to the currentThread variable which stores the thread which is currently running, to let the scheduler know that the execution of this thread has been completed and to not add this thread to the queue again.

- **int mypthread_join():**

This function implements one while loop which checks the threadsTerminated array for the termination of a particular thread. Only after the execution of the current thread is completed the next thread is allowed to start its execution.

- **Mutex():**

Mutexes are a locking mechanism used to synchronize two or more threads, i.e., when multiple threads try to access a shared resource at the same time, the mutex mechanism is used to ensure that only one thread can use that resource at a time. Mutex uses a locking/unlocking mechanism, hence to initialize the mutex lock, we use a function **mypthread_mutex_init()**, which first checks for an invalid pointer, and initializes the flag to value 0, indicating its availability.

In the **mypthread_mutex_lock()** function, we first use the built-in atomic test-and-set function to test the mutex's availability and if it can be acquired. If the function returns 1 after setting it to 1, this indicates that the mutex lock has already been acquired by some other thread, and the current thread is pushed to the blocked threads list, with the isBlocked value set to 1 so that the current thread isnt pushed to the scheduler thread. In the other case, the mutex thread is assigned to the current thread, and flag is set to value 1, indicating that mutex isn't available.

In the **mypthread_mutex_unlock()** function, we are releasing the mutex. First set the mutex flag to 0, indicating its availability, and then we set the mutex's thread to null, releasing the mutex. Once it is released, other threads are able to access and lock the mutex when needed. The releaseThreads() function is then invoked to release the threads in the blocked list into the running queue.

In the **mypthread_mutex_destroy()** function we will deallocate dynamic memory which was allocated in the mypthread_mutex_init function, destroying the mutex.

- **static void schedule()**

Whenever the timer interrupts, this function is called that makes the context switch happen from thread context to the scheduler context. In this function other functions are called according to the policies, Round Robin, MLFQ or PSJF.

- **static void sched_RR():**

In Round Robin there is a specific time quanta assigned to each of the executing threads. The status of the to be executed threads is changed to SCHEDULED and isBlocked to 0 indicating that the thread is not blocked. After the specific time quanta has passed the next thread is being executed. This solves the starvation problem which occurs in MLFQ as there is no priority level but a specific time quanta after which thread context switch happens.

- **static void sched_PSJF():**

In PSJF, threads are placed into the queue as they arrive, but as soon as a thread with a shorter burst time arrives, the existing thread gets removed from execution, and the shorter job is executed instead.

- **static void sched_MLFQ():**

The scheduler picks the next available ready thread in the queue. This scheduler has priority levels 1, 2, 3 and 4, where threads in level 1 has the lowest priority level, whereas threads in level 4 have the highest priority, and every thread from the highest priority level is chosen one by one to be executed. After their time quanta is completed, their priority level decreases. If any thread yields, their priority level is unchanged. The lower level priority queues are accessed by the scheduler only when the higher level priority queues are empty. At last we used the Round Robin for the threads in same priority level queue. Round Robin scheduling is performed for jobs inside the queue (highest to lowest)

Benchmarking

Vector Multiply

Time Quanta (in ms)	Pthread (in microseconds)	mypthread(in microseconds)
5	157	171
12	357	391
20	65	178
30	102	138

Parallel cal

Time Quanta (in ms)	Pthread (in microseconds)	mypthread(in microseconds)
5	7341	8102
12	8295	9040
20	3486	3923
30	7324	7334

External cal

Time Quanta (in ms)	Pthread (in microseconds)	mypthread(in microseconds)
5	1062	1114
12	1337	1352
20	696	795
30	1247	1277

We have documented the time taken by threads for different time Quanta in the above table. The different time Quanta that we used were 5ms, 12ms, 20ms and 30ms. The efficiency of our library is almost similar to the pthread library when we run vector multiply, external_cal and parallel_cal benchmarks provided. The number of threads that we tested on were 20.