CS518 A0: Uthreads

Project Report

By

Victor Sankar Ghosh (vsg23)
Rhema Keren Marneni (rkm110)

TABLE OF CONTENTS

1. INTRODUCTION

This project attempts to implement user threads and handle them using different functions and scheduling policies. Below, we list our approach to the functions given to implement, and the other supporting functions we wrote in our code.

2. DATA STRUCTURES IMPLEMENTED

Enum types

`p_thread_state`

Describes different thread states:

RUNNING	The thread is currently under execution
READY	The thread is scheduled to be run by the processor
DONE	The thread completed execution
WAITING	The thread is blocked and is waiting to be scheduled
NEW	A new thread is created and is not used yet

`mutex_State`

Describes the status of a mutex lock for a thread:

UNLOCKED	The mutex is available for a thread waiting on it
LOCKED	The mutex has been acquired by a thread, other threads cannot use it

`scheduler_type`

Describes the scheduling policy to run:

RR	Round Robin
PSJF	Priority Shortest Job First
MLFQ	Multilevel Feedback Queue

Struct types

`threadControlBlock`

Defines the Thread Control Block for a thread. Stores various attributes as follows:

`int threadID`	A thread is identified by a unique ID (tid)
`enum pthread_State threadStatus`	The current state of the thread
`int waitingOn`	The tid of another thread it is waiting on
`int beingWaitingOnBy`	The tid of another thread waiting on it
`int waitingOnMutex`	Holds mutex ID of mutex if waiting on a mutex, -1 otherwise
`int quantumsElapsed`	Number of time quantums elapsed
`ucontext_t threadContext`	Stores context of the thread
`void* threadStack`	Pointer to the thread stack
`void* returnValue`	Pointer to the return value

`mypthread_mutex_t`

Describes mutex values:

`mutexId`	Each mutex has a unique ID
`lockState`	Set to **`LOCKED`** if mutex acquired or **`UNLOCKED`** state otherwise

`queue_node`

For each node in the circular linked list for MLFQ:

`mypthread_t id`	Thread ID
`node next*`	Pointer to the next thread in the queue

`queue`

Circular linked list for MLFQ:

`node* tail`	Each mutex has a unique ID
`int size`	Size of queue

`Mlfq`

`queue qs[LEVELS]`	**`LEVELS`** number of Queue levels
`queue fcfs`	A queue for the FCFS case of MLFQ
`tcb* last_picked`	Thread enqueued in a level
`int last_picked_level`	Updated as thread is enqueued in a level

Array

`tcb runningQueue[]`

This queue stores all the threads ever created. Each index of the queue points to a different thread control block with unique tid.

`const int LEVEL_QUANTUM[]`

This array stores the four levels of quantum for which MLFQ is executed.

3. SCHEDULING POLICIES

3.1 Round Robin Scheduling (RR)

A pool of threads are scheduled to run in a fair manner. In that, each thread gets to run for a specified interval of time, referred to as ‘time quantum’. After each interval, the thread is preempted out and the next scheduled thread is given execution time. Eventually, all threads run to completion. We chose to test with three different quanta i.e., 25, 40, 50 milliseconds.

3.2 Preemptive Shortest Job First Scheduling (PSJF)

Preemptive Shortest Job First Scheduling allows the thread with the least burst time to execute first. Shortest jobs are given more priority. However, we would typically require foreknowledge of the burst times of each thread, which is impossible. So, we implement this by giving priority to the thread that has run for the least time. Jobs with the shortest elapsed time are given more priority.

3.3 Multilevel Feedback Queue Scheduling (MLFQ)

Schedule a job in a queue that uses a scheduling algorithm appropriate to the job and its behavior. It swap jobs based on their priority and performance. Can allow jobs to be classified and run using a scheduling algorithm that is well-tuned for that particular job's characteristics. We implement this by picking different time quanta. If a thread did not run to completion in one level, it is sent to the next level with a higher quantum. This continues until it completes its execution. At the highest quantum level, the remaining threads run on FCFS.

MLFQ cases for first three quantum levels:

Existing Thread				Outcome		
Present	**Finished**	**Running**	**Quota Exhausted**	**Thread Switch**	**Move Queue**	**Schedulable**
N				Start Next thread	Move in same queue	Schedulable
Y	Y			Start Next thread	Move in same queue	Schedulable
Y	N	Y	N	Keep Current thread	Move in same queue	Schedulable
Y	N	N	N	Swap to next thread	Move in same queue	Blocked = Non-schedulable
	N	Y	Y	Swap to next thread	Move to next queue	Schedulable
Y	N	N	Y	Swap to next thread	Move to next queue	Blocked = Non-Schedulable

MLFQ for FCFS case (last quantum level):

Existing Thread			Outcome		
Present	**Finished**	**Running**	**Thread Switch**	**Move Queue**	**Schedulable**
N			Start Next thread	Move in same queue	Schedulable
Y	Y		Start Next thread	Move in same queue	Schedulable
Y	N	Y	Keep Current thread	Move in same queue	Schedulable
Y	N	N	Swap to next thread	Move in same queue	Blocked = Non-schedulable

4. FUNCTION DESCRIPTIONS

4.1 Thread Functions

```
int mypthread_create(mypthread_t * thread, pthread_attr_t * attr,
void *(*function)(void*), void * arg);
```

A function to create a thread. Returns 0 on success. Before creating a new thread, a counter **threadIDCounter** checks if it is the first ever thread to be created by the program. For value 0, the first thread is created and all the Thread Control Blocks in **runningQueue** are set to uninitialized, status is NEW and return values are set to NULL. The first thread will be the main thread and the **threadIDCounter** is incremented. Here, the timer is also starts. When the counter increases to 1 and further, a new TCB is created at that index of **runningQueue**, and a stack will be malloc'd to the thread. This is done using **makecontext()**. Then the counter increment to give a unique tid for the next thread. After the main thread exits, all the heap data must be freed. So, **cleanup()** is called during **atexit()**. Whenever the timer has to go off after **QUANTUM** amount of time (i.e., 10 milliseconds), the signal is handled by calling **swapToScheduler()** function.

```
void mypthread_yield();
```

A function to let other threads get CPU execution. It takes no arguments. The current thread is taken away from the processor and sent to Ready state. As a result, a context switch occurs, where in, the context of the thread will be saved in its Thread Control Block and changed to the scheduler context. A variable **justExited** is set to 0, to let the scheduler know that the last thread did not finish exiting, but just changed its state.

```
void mypthread_exit(void *value_ptr);
```

A function to handle thread termination. As the thread exits, it has no more execution time, so pause the timer. Change the status of the thread to DONE. If there are any threads that called **join()** on this thread, they need to be sent to the scheduler to be run. Their threadStatus is set to READY and **waitingOn** is set to -1, to make them runnable by the scheduler. The argument ***value_ptr** will update **returnValue** in the Thread Control Block. If it is NULL, it does not need to do that. Finally, since the thread has finished exiting, **justExited** is set to 1, to let the scheduler know it has finished execution. The timer resumes and scheduler runs.

```
int mypthread_join(mypthread_t thread, void **value_ptr);
```

A function to let the threads waiting on a calling thread, get execution time after the calling thread finishes execution. Firstly, we check if thread exited. In case it did, scheduler runs and **justExited** will be set to 0, letting the scheduler know it has not exited, so that the threads waiting on it can be executed. If ****value_ptr** is not NULL, it will be set to thread's **returnValue** attribute. Its **beingWaitedOnBy** attribute is set to the id of calling thread (referred to as **currentThread**) which will be contacted once thread exits. Calling thread is set to WAITING status with **waitingOn** set to tid of thread so that it will not be scheduled until the thread has exited. Next, scheduler will run and **justExited** is set to 0.

```
int mypthread_mutex_init(mypthread_mutex_t *mutex, const
pthread_mutexattr_t *mutexattr);
```

A function to initialize a new mutex. Each mutex must have a distinct id. To maintain that, we use a counter variable **mutexIDCounter**. The mutex id is set to this value and its state is unlocked as it has not been used yet. The counter will then increment for the next mutex created to have a unique id.

```
int mypthread_mutex_lock(mypthread_mutex_t *mutex);
```

A function to acquire a mutex. We use **__atomic_test_and_set()** for the calling thread to keep checking the state of the mutex in a while loop. As long as its status is LOCKED, the status will be WAITING and the **waitingOnMutex** attribute indicates that it is waiting. So, it cannot be scheduled. The scheduler runs and **justExited** is set to 0. If the loop breaks, that is the mutex is no longer locked, the calling thread changes the **lockState** to LOCKED so that it alone can use it.

```
int mypthread_mutex_unlock(mypthread_mutex_t *mutex);
```

A function to release the mutex. Firstly, set the state to UNLOCKED. Loop over **runningQueue** to check if any other threads are waiting for the mutex lock. Such threads, if any, will have their **waitingOnMutex** attribute indicate that they are waiting, and be sent to READY state for the scheduler to pick them up.

```
int mypthread_mutex_destroy(mypthread_mutex_t *mutex);
```

A function to destroy a mutex. Destroying works the same as the mutex unlock function, in that, we change the state to UNLOCKED and check for any threads waiting on the mutex. That itself indicates destroying the mutex as there was no dynamic memory allocation in the process of initializing a mutex.

4.2 Scheduling Functions

(In our code, we manually set **`scheduler_type`** *to which scheduling policy to run.)*

`static void sched_RR()`

Follows the concept of Round Robin Scheduling Algorithm. Each thread is allotted an interval of time i.e., **`ALLOWED_NUMBER_OF_QUANTUMS`**, to execute after which, it is context-switched out for another thread that does the same. The quanta chosen in this project is 25, 40 and 50 milliseconds. *Each time, we set the time quantum manually.* We are careful to not perform context switch if the thread is schedulable, is actively running and still has some time left to finish. If the execution is done, set **`justExited`** to 1 and look for next thread. Otherwise, set **`justExited`** to 0 and context switch to next thread.

`static void sched_PSJF()`

To implement preemptive based SJF, foreknowledge of burst times of threads is required. However, as that is not possible, the scheduler here gives priority to the thread that has the least elapsed time. The thread that has executed for the least number of quantums gets more priority. The function loops over the threads and execute them as long as they are schedulable, and not waiting for another thread or mutex. **`LowestQuant`** holds the value of the thread that has executed for the least amount of time. To make it preemptive, **`justExited`** is set to 1 if the job is fully executed, otherwise, **`justExited`** is set to 0, the thread is preempted out and the next priority thread swaps in.

`static void sched_MLFQ()`

Implements Multilevel Feedback Queues Scheduling Algorithm. Each round uses a Circular Linked List with a dedicated time quantum. The threads that do not finish execution in one level, are moved to the next queue level (with a higher quantum) to complete their execution. We considered four quantum levels of 20, 40, 100 and 180 in our function. One function for the first three levels, and one for the last case, where the execution ends up in FCFS policy.

`static void schedule()`

Based on the type of scheduling function, that scheduler is invoked. In our code, we manually set which scheduling policy to run.

4.3 Additional Helper Functions

Helper function for Round Robin:

```
mypthread_t RR_find_next_thread_id(mypthread_t tid)
```

Helper function for **sched_RR()**.It selects the next schedulable thread for the next time quantum and returns that to **sched_RR()**.

Helper functions for MLFQ:

```
void enqueue(queue* q, mypthread_t tid)
```

Helper function for MLFQ Scheduling. Adds the thread to a queue level.

```
mypthread_t dequeue(queue* q)
```

Helper function for MLFQ Scheduling. Removes a thread from a queue level and frees it, after it has finished executing fully. Returns the tid of that thread.

```
mypthread_t dequeue_next_schedulable(queue* q)
```

Helper function for MLFQ Scheduling. Removes a thread, that has not run to completion, from a queue level. Since it still has to execute more, we **enqueue()** to the next level (where the time quantum increases). Returns the tid of such a thread.

```
static void pickThreadFromMlfq(int level, mypthread_t next_tid)
```

Helper function for MLFQ Scheduling. Picks a thread from a queue level. If it did not finish executing after one round, its level is updated to the next i.e., moved to a higher level.

```
static void sched_MLFQ_FCFS()
```

Helper function for MLFQ Scheduling. This level separately handles the highest level of quantum, where the execution becomes a First-Come First-Served policy.

```
static void sched_MLFQ_regular(int level)
```

Helper function for MLFQ Scheduling. Here, all the quantum levels (except the FCFS case) are handled. At each level, the queue must have a thread that is specified to run in that level (**`last_picked`** is not **`NULL`**), is schedulable and is not blocked.

```
static int hasAtLeastOneScheduleable(queue* q)
```

Helper function for MLFQ Scheduling. The function looks for at least one schedulable thread in the entire queue level. Returns 1 if found, and 0 if not found.

```
static int findAvailableQueueLevel()
```

Helper function for MLFQ Scheduling. This function selects the next available queue level. If in one queue level, checks if the next queue level is available. Returns the level value on success and -1 on failure.

```
static void init_MLFQ(Mlfq* mlfq)
```

Helper function for MLFQ Scheduling. It initializes all the queue levels required for implementing MLFQ. Sets all sizes to 0 and the tail pointer to **`NULL`**.

Helper function for implementing threads:

```
int isScheduleable(tcb *t)
```

It returns true if the thread is not waiting on another thread, not waiting on a mutex, still has some execution left and is waiting to get execution time.

```
void swapToScheduler()
```

This function is run after a thread's quantum is done. It will increment the number of quantums it executed for, sets the state of current thread to **`READY`** and schedules it to run.

```
void set_next_thread(mypthread_t tid)
```

A function to switch to next schedulable thread after current thread finished its execution.

```
void swap_thread(mypthread_t x, mypthread_t y)
```

A function to context switch to next schedulable thread. Current thread still has some execution left, but swapped out for next thread.

```
void pauseTimer()
```

A function to pause the timer.

```
void resumeTimer()
```

A function to resume the timer.

```
void cleanup()
```

A function to deallocate heap space for the thread that finished execution. It runs over **`runningQueue`** to check for threads whose status is marked as DONE. Their memory stack is freed.

```
char state(mypthread_t tid)
```

A function for debugging purposes. ‘X’ indicates not schedulable and ‘S’ indicates schedulable.

5. INSTRUCTIONS

5.1 Testing Instructions

*In our code, we manually set **`scheduler_type`** to the scheduling policy we want to run.

Include the following into benchmarks/Makefile

```
build_and_run_all:
$(MAKE) clean_all && $(MAKE) -C .. all && $(MAKE) all
./external_cal
./parallel_cal
./vector_multiply
```

Open terminal on ilab3 and run the following command

```
make build_and_run_all
```

Check if sum run by the test driver is equal to verified sum generated

5.2 Debugging instructions

Set DEBUG flag in mypthread.h to 1 and run

6. PERFORMANCE REPORT

Round Robin Output:

```
vsg23@ilab3:~/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks$ make build_and_run_all
make clean_all && make -C .. all && make all
make[1]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
rm -rf testcase test parallel_cal vector_multiply external_cal *.o
make clean -C ..
make[2]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
rm -rf testfile *.o *.a
make[2]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
make[1]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
make[1]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
gcc -pthread -g -c mypthread.c
ar -rc libmypthread.a mypthread.o
ranlib libmypthread.a
make[1]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
make[1]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
gcc -g -w -DUSE_MYTHREAD -pthread -o parallel_cal parallel_cal.c -L../ -lmypthread
gcc -g -w -DUSE_MYTHREAD -pthread -o vector_multiply vector_multiply.c -L../ -lmypthread
gcc -g -w -DUSE_MYTHREAD -pthread -o external_cal external_cal.c -L../ -lmypthread
gcc -g -w -DUSE_MYTHREAD -pthread -o test test.c -L../ -lmypthread
make[1]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
./external_cal
running time: 1348 micro-seconds
sum is: -776529381
verified sum is: -776529381
./parallel_cal
running time: 7217 micro-seconds
sum is: 83842816
verified sum is: 83842816
./vector_multiply
running time: 97 micro-seconds
res is: 631560480
verified res is: 631560480
vsg23@ilab3:~/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks$
```

Time quantum taken = 40 microseconds

PSJF Output:

```
vsg23@ilab3:~/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks$ make build_and_run_all
make clean_all && make -C .. all && make all
make[1]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
rm -rf testcase test parallel_cal vector_multiply external_cal *.o
make clean -C ..
make[2]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
rm -rf testfile *.o *.a
make[2]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
make[1]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
make[1]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
gcc -pthread -g -c mypthread.c
ar -rc libmypthread.a mypthread.o
ranlib libmypthread.a
make[1]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
make[1]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
gcc -g -w -DUSE_MYTHREAD -pthread -o parallel_cal parallel_cal.c -L../ -lmypthread
gcc -g -w -DUSE_MYTHREAD -pthread -o vector_multiply vector_multiply.c -L../ -lmypthread
gcc -g -w -DUSE_MYTHREAD -pthread -o external_cal external_cal.c -L../ -lmypthread
gcc -g -w -DUSE_MYTHREAD -pthread -o test test.c -L../ -lmypthread
make[1]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
./external_cal
running time: 1359 micro-seconds
sum is: -776529381
verified sum is: -776529381
./parallel_cal
running time: 6102 micro-seconds
sum is: 83842816
verified sum is: 83842816
./vector_multiply
running time: 92 micro-seconds
res is: 631560480
verified res is: 631560480
vsg23@ilab3:~/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks$
```

MLFQ Output:

```
vsg23@ilab3:~/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks$ make build_and_run_all
make clean_all && make -C .. all && make all
make[1]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
rm -rf testcase test parallel_cal vector_multiply external_cal *.o
make clean -C ..
make[2]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
rm -rf testfile *.o *.a
make[2]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
make[1]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
make[1]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
gcc -pthread -g -c -Wdeprecated-declarations mypthread.c
ar -rc libmypthread.a mypthread.o
ranlib libmypthread.a
make[1]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign'
make[1]: Entering directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
gcc -g -w -DUSE_MYTHREAD -pthread -o parallel_cal parallel_cal.c -L../ -lmypthread
gcc -g -w -DUSE_MYTHREAD -pthread -o vector_multiply vector_multiply.c -L../ -lmypthread
gcc -g -w -DUSE_MYTHREAD -pthread -o external_cal external_cal.c -L../ -lmypthread
gcc -g -w -DUSE_MYTHREAD -pthread -o test test.c -L../ -lmypthread
make[1]: Leaving directory '/common/home/vsg23/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks'
./external_cal
running time: 1209 micro-seconds
sum is: -776529381
verified sum is: -776529381
./parallel_cal
running time: 6707 micro-seconds
sum is: 83842816
verified sum is: 83842816
./vector_multiply
running time: 77 micro-seconds
res is: 631560480
verified res is: 631560480
vsg23@ilab3:~/Documents/Rutgers_Workspace/OperatingSystemDesign/benchmarks$
```

Comparison Table

Benchmarks		RR (quantum = 40)	PSJF	MLFQ
external_cal	Running time (microseconds)	1348	1359	1209
	Sum	-776529381	-776529381	-776529381
	Verified Sum	-776529381	-776529381	-776529381
parallel_cal	Running time (microseconds)	7217	6102	6707
	Sum	83842816	83842816	83842816
	Verified Sum	83842816	83842816	83842816
vector_multiply	Running time (microseconds)	97	92	77
	Sum	631560480	631560480	631560480
	Verified Sum	631560480	631560480	631560480

Our code shows that performs better by taking lesser runtime. Round Robin and PSJF policies almost perform similarly for these benchmarks.