

## CS416 Project 2: User-level Thread Library and Scheduler

**Due: 9th March 2021 (100 points + 10 Extra Credit)**

In Project 1, you learned how to use pThread library for multi-threaded programming. In Project 2, you will get a chance to exploit the logic inside a thread library. In this assignment, you are required to implement a pure user-level thread library that has a similar interface compared to the pThread Library. Since you will be providing a multi-thread environment, you will also need to implement pthread mutexes which are used to guarantee exclusive access in the critical section. This assignment is intended to illustrate the mechanics and difficulties of scheduling tasks within an operating system.

*Note:* Please maintain the partner pairings from project 1.

### Code Structure

You are given a code skeleton structured as follows:

- `rpthread.h`: contains thread library function prototypes and definition of important data structures
- `rpthread.c`: contains the skeleton of thread library. All your implementation goes here.
- `Makefile`: used to compile your library. Generates a static library (`rpthread.a`).
- `Benchmark`: includes benchmarks and a `Makefile` for the benchmarks to verify your implementation and do a performance study. There is also a test file that you can modify to test the functionalities of your library as you implement them.

You need to implement all of the API functions listed below in Part 1, the corresponding scheduler function in Part 2, and any auxiliary functions you feel you may need.

To help you towards the implementation of the entire pthread library, we have provided logical steps in each function. It is your responsibility to convert and extend them into a working code.

### Part 1. Thread Library (50 points)

**1.1 Thread creation (10 points)** The first API to implement is thread creation. You will implement the following API to create a thread that executes function. You could ignore `attr` for this project (pass `NULL`).

```
int rpthread_create(rpthread * thread, pthread_attr_t * attr,
                  void *(*function)(void*), void * arg);
```

Thread creation involves three parts.

**Thread Control Block:** First, every thread has a thread control block (TCB), which is similar to a process control block that we discussed in the class. The thread control block is represented using the `threadControlBlock` structure (see `rpthread.h`). Add all necessary information to the the TCB. You might also need a thread ID (a unique ID) to identify each thread. You could use `rpthread_t` inside the TCB structure to store thread id during thread creation.

**Thread Context:** Every thread has a context, needed by Linux for running the thread on a CPU. The context is also a part of TCB. So, once a TCB structure is allocated and populated, the next step is to create a thread context. Linux provides APIs to create a user-level context and switch contexts. During thread creation (`pthread_create`), `makecontext()` will be used to generate a context. Before using `makecontext`, you will need update the context structure. You can read more about a context here: <http://man7.org/linux/man-pages/man3/makecontext.3.html>

**Runqueue:** Finally, once the thread context is set, you will need to add the thread to a scheduler runqueue. The runqueue has active threads ready to run and waiting for the CPU. Feel free to use a linked list or any other data structure to back your scheduler queues. *Note* that in the second part of this project, you will need to implement a multi-level scheduler queue. So, we suggest to keep your code modular for enqueueing or dequeuing threads from the scheduler queue.

*Note:* We shall be testing your complete threading library with 50 to 100 threads.

## 1.2 Thread Yield (10 points)

```
void rpthread_yield();
```

The *rpthread\_yield* function (API) enables the current thread to *voluntarily* give up its CPU resource. That is to say that the thread context will be swapped out (read about Linux *swapcontext()*), and the scheduler context will be swapped in so that the scheduler thread can enqueue the current thread back to runqueue and choose the next thread to run. You can read about swapping a context here: <http://man7.org/linux/man-pages/man3/swapcontext.3.html>

## 1.3 Thread Exit (10 points)

```
void rpthread_exit(void *value_ptr);
```

This *rpthread\_exit* function is an explicit call to the *rpthread\_exit* library to end the pthread that called it. If the *value\_ptr* isn't NULL, any return value from the thread will be saved. Think about what things you should clean up or change in the thread state and scheduler state when a thread is exiting.

## 1.4 Thread Join (10 points)

```
int rpthread_join(rpthread_t thread, void **value_ptr);
```

The *rpthread\_join* ensures that the calling thread will not continue execution until the one it references exits. If *value\_ptr* is not null, the return value of the exiting thread will be passed back.

## 1.5 Thread Synchronization (10 points)

Only creating threads is insufficient. Access to data across threads must be synchronized. Recollect that in Project 1, you were a consumer of Linux pthread library's mutex operation. In Project 2, you will be designing *rpthread\_mutex*. Mutex serializes access to a function or function states by synchronizing access across threads. Read <http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf> for complete understanding of mutexes.

The first step is to fill the *rpthread\_mutex\_t* structure defined in *rpthread.h* (currently empty). While you are allowed to add any necessary structure variables you see fit, you might need a mutex initialization variable, information on the thread (or thread's TCB) holding the mutex, as well as any other information.

### 1.5.1 Thread Mutex Initialization

```
int rpthread_mutex_init(rpthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
```

This function initializes a *rpthread\_mutex\_t* created by the calling thread. The 'mutexattr' can be ignored for the purpose of this project (call with NULL).

### 1.5.2 Thread Mutex Lock and Unlock

```
int rpthread_mutex_lock(rpthread_mutex_t *mutex);
```

This function attempts to lock the mutex for a caller thread. Other threads, trying to lock the same mutex will not be able to do so until the current lock is released. (recollect pthread\_mutex usecase)

```
int rpthread_mutex_unlock(rpthread_mutex_t *mutex);
```

This function unlocks a given mutex. Once a mutex is released, other threads might be able to lock this mutex again.

### 1.5.3 Thread Mutex Destroy

```
int rpthread_mutex_destroy(rpthread_mutex_t *mutex);
```

Finally, this function destroys a given mutex. Make sure to release the mutex before destroying it.

## Part 2: Scheduler (40 points + 10 Extra Credit)

Since your thread library is managed totally in user-space (the OS only assigns 1 kernel thread), you also need to have a scheduler and policies in your thread library to determine which thread to run next. In the second part of the assignment, you are required to implement the following two scheduling policies:

### 2.1 Preemptive Round Robin (RR) (20 points)

For the first scheduling algorithm, you are required to implement a preemptive Round robin algorithm. Each thread shall be given a set amount of exclusive CPU time (“TIMESLICE”) window after which it can be context switched out. Let’s assume each timeslice is 5ms; depending on your scheduler logic, one could context switch out a thread after one timeslice or more than one timeslices. In the skeleton code, take a look at the Makefile on how to control the timeslice at compile time, without changing code.

*Note:* We shall be testing your code with different timeslice values and thread counts.

Here are some hints to implement this particular algorithm:

- 1) Maintain a job queue which can enqueue a job at the end of the queue and pop from its head.
- 2) Try to make this algorithm as modular as possible so that you don’t need to reimplement it when implementing MLFQ with RR in the next subsection.

### 2.2 Multi-level Feedback Queue (20 points)

The second scheduling algorithm you need to implement is basic MLFQ. In this algorithm, you have to maintain a queue structure with multiple levels. Here are the hints to implement:

- 1) Instead of a single runqueue, you need multiple levels of run queue. It could be a 4-level or 8-level queue as you like.
- 2) When a thread has finished one “timeslice” move it to the next level of runqueue, ie. if a thread yields its CPU before its timeslice ends, it stays in its level. Your scheduler should always pick a thread at the top-level of the runqueue.
- 3) As mentioned in section 2.1, threads in the same level should run in Round Robin. It might be a good idea to reuse the code here for each level. This also introduces the idea that each queue should have the same time slice since this is a basic MLFQ scheduler.
- 4) Again, try to keep this as modular as possible, since you will most likely want to modify MLFQ if you were to attempt the extra credit (section 2.3).

*Note:* We shall be testing your code with varying thread counts and timeslice values.

**Invoking the Scheduler Periodically** For both of the two scheduling algorithms, you will have to set a timer interrupt for some timeslice (say  $t$  ms) so that after every  $t$  ms your scheduler will preempt the current running thread. Fortunately, there are two useful Linux library functions that will help you do just that:

```
int setitimer(int which, const struct itimerval *new_value,
              struct itimerval *old_value);
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

More details can be found here: <https://linux.die.net/man/2/setitimer> <https://linux.die.net/man/2/sigaction>

### 2.3 Extra Credit: Prevent Gaming MLFQ (10 points)

Reading the MLFQ description, one might think of many ways of gaming the system, ie. try to get all the CPU cycles for yourself. One such method is to purposefully yield the CPU right before the Quantum ends to avoid being demoted to a lower priority. This thread will always be scheduled first since it maintains its high priority queue while other threads go down to lower priority queues.

In this part, you can be as creative as possible to detect such threads and force them down to lower priorities. Add/extend functions and/or data structures that will help you in this.

*Note:* Attempt this part if you are sure that all the previous parts work as expected. Your Report should have extensive details about how you solve gaming. The report should indicate cases tested that show why your proposed solution would not allow gaming; performance must be compared against MLFQ without the gaming logic.

### 3. Other Hints

**schedule()** The schedule function is the heart of the scheduler. Every time the thread library decides to pick a new job for scheduling, the schedule() function is called, which then calls the scheduling policy (RR or MLFQ) to pick a new job.

*Think about conditions when the schedule() method must be called.*

**Thread States** As discussed in the class, threads must be in one of the following states. These states help you to classify thread that is currently running on the CPU vs. threads waiting in the queue vs. threads blocked for I/O. So you could define these three states in your code and set update the thread states.

```
#define READY 0
#define SCHEDULED 1
#define BLOCKED 2
```

e.g., `thread->status = READY;`

If needed, feel free to add more states as required.

### 4. Compilation

As you may find in the code and Makefile, your thread library is compiled with RR as the default scheduler. To change the scheduling policy when compiling your thread library, pass variables with make:

```
make SCHED=MLFQ
```

### 5. Benchmark Code

The code skeleton also includes a benchmark that helps you to verify your implementation and study the performance of your thread library. There are three programs in the benchmark folder (parallelCal and vectorMultiply are CPU-bound, and externalCal is IO-bound). To run the benchmark programs, **please see README in the benchmark folder.**

Here is an example of running the benchmark program with the number of threads to run as an argument:

```
> make
> ./parallelCal 4
```

The above example would create and run 4 user-level threads for parallelCal benchmark. You could change this parameter to test how thread numbers affect performance.

To understand how the benchmarks work with the default Linux pthread, you could comment the following MACRO in rpthread.h and the code would start using the default Linux pthread. To use your thread library to run the benchmarks, please uncomment the following MACRO in rpthread.h and recompile your thread library and benchmarks.

```
#define USE_RTHREAD 1
```

To help you while you are implementing the user-level thread library and scheduler, there is also a program called *test.c* which is a blank file which you can use to play around with and call rpthread library functions to check if they work as you intended. Compiling the test program is done in the same way the other benchmarks are compiled:

```
> make
> ./test
```

## 6. Report (10 points)

Besides the thread library, you also need to write a report for your work. The report must include the following parts:

1. Detailed logic of how you implemented each API functions and scheduler
2. Benchmark results of your thread library with different configurations of thread number.
3. A short analysis of the benchmark results and comparison of your thread library with pthread library.

## 7. Suggested Steps

1. Designing important data structures for your thread library. For example, TCB, Context, and Runqueue.
2. Finishing `rpthread_create`, `rpthread_yield`, `rpthread_exit`, `rpthread_join`, and scheduler functions (with a simple FCFS policy).
3. Implementation of `rpthread_mutex`.
4. Extending your scheduler function with preemptive RR and MLFQ scheduling policy.

*NOTE:* Do not change the function call signatures for user facing functions such as `rpthread_create`, `rpthread_yield`, `rpthread_exit`, `rpthread_join`, `pthread_mutex_init`, `rpthread_mutex_lock`, `rpthread_mutex_unlock`, `rpthread_mutex_destroy`.

## 8. Submission

Submit the following files to Sakai, as is (**Do not compress them**): 1. `rpthread.h` 2. `rpthread.c` 3. A report in .pdf format, completed, and with both partners names and netids on the report 4. Any other support source files and Makefiles you created or modified

Please **do not compress these files**. If you did not create any new source files or modify the Makefiles, there is no need to submit any of the makefiles or benchmark source files as we already have them. Any one of the project partner may submit on sakai. There is no need for both the project partners to submit.

## 9. Resources

A POSIX thread library tutorial: <https://computing.llnl.gov/tutorials/pthreads/>

Another POSIX thread library tutorial: <http://www.mathcs.emory.edu/~cheung/Courses/455/Syllabus/5c-pthreads/pthreads-tut2.html>

Some notes on implementing thread libraries in Linux: <http://www.evanjones.ca/software/threading.html>