# MyPthread User-level Thread Library and Scheduler:
Understanding Scheduling & Synchronization Complexity

## Introduction

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 to critical sections. This assignment is intended to illustrate the mechanics and difficulties of scheduling tasks within an operating system.

**Note: Make sure you read the whole project description before starting or asking questions. For those registered for CS518, you will be required to implement an additional scheduling policy and provide a more comprehensive report.**

## Code Structure

You are given a code skeleton structured as follows:

- ***mypthread.h***: contains thread library function prototypes and definitions of important data structures
- ***mypthread.c***: contains the skeleton of thread library, all your implementation goes here
- ***Makefile***: used to compile your library, generates a static library (mypthread.a).
- **Benchmark**: includes benchmarks and a Makefile for the benchmarks to verify your implementation and do performance a 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.

# 1 Thread Library

The following describes the different parts of the threading library you will need to implement.

### 1.1 Thread Creation

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.

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

Thread creation involves three parts:

1. **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 *mypthread.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 the mypthread_t inside TCB structure to set this during thread creation.

2. **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 set and allocated, 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. Before the use of *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

3. **Runqueue:**
   Finally, once the thread context is set, you might 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 the project, you will need a multi-level scheduler queue. So, we suggest to keep your code modular for enqueuing or dequeuing threads from the scheduler queue.

## 1.2 Thread Yield

void mypthread_yield();

The *mypthread_yield* function (API) enables the current thread to voluntarily give up the CPU resource to other threads. That is to say, the thread context will be swapped out (read about Linux *swapcontext()*), and the scheduler context will be swapped in so that the scheduler thread could put 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

void mypthread_exit(void *value_ptr);

This *mypthread_exit* function is an explicit call to the *mypthread_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

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

The *mypthread_join* ensures that 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

In Project 2, you will be designing *mypthread_mutex*. Mutex serializes access to a function or function states by synchronizing access across threads.

The first step is to fill the *mypthread_mutex_t* structure defined in *mypthread.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 mypthread_mutex_init(mypthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
```

This function initializes a *mypthread_mutex_t* created by the calling thread. The 'mutexattr' can be ignored for the purpose of this project.

#### 1.5.2 Thread Mutex Lock and Unlock

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

This function sets the lock for the given mutex and other threads attempting to access this mutex will not be able run until the mutex is released (recollect *pthread_mutex* use).

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

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

# 2 Scheduler

Since your thread library is managed totally in user-space, 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 Pre-emptive SJF (PSJF)

For the first scheduling algorithm, you are required to implement a pre-emptive SJF, which is also known as **STCF**. Unfortunately, you may have noticed that our scheduler DOES NOT know how long a thread will run for completion of job. Hence, in our scheduler, we could book-keep the time quantum each thread has to run; **this is on the assumption that the more time quantum a thread has run, the longer this job will run to finish**. Therefore, you might need a generic "QUANTUM" value defined possibly in *mypthread.h*, which denotes the minimum window of time after which a thread can be context switched out of the CPU. Let's assume each quantum is 5 milliseconds; depending on your scheduler logic, one could context switch out a thread after one quantum or more than one quantum. To implement a mechanism like this, you might also need to keep track of how many time quantums each thread has ran for.

Some hints to implement this particular scheduler:

- In the TCB of each thread, maintain an "elapsed" counter, which indicates how my time quantum has expired since the time thread was scheduled. After the first time quantum finishes, increment the counter for the running thread by one.
- Because we do not know the actual runtime of a job, to schedule the shortest job, you will have to find the thread that currently has the minimum counter value, remove the thread from the runqueue, and context switch the thread to CPU.

## 2.2 Other General Hints for Implementing Scheduling:

**Invoking the Scheduler Periodically:**

For both of the two scheduling algorithms, you will have to set a timer interrupt for some time quantum (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

**Picking a time quantum :**

There is no exact time quantum you should use, you should use one small enough to allow for enough multiplexing between all the threads. You should aim to use a time quantum of 5, 10, or 20 milliseconds. When you finish implementing your library, you should play around with the time quantum to see which yields the best performance for the benchmarks.

**How schedule() should work:**

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 PSJF (or MLFQ) to pick a new job.

**Important: Really 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           // e.g., thread->status = READY;
#define SCHEDULED 1
#define BLOCKED 2
```

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

**Watch out for interval timers and timer's going off when not wanted:**

With *setitimer()*, it is possible to set an interval timer that triggers every $x$ seconds no matter what. The timer does this by resetting itself after it goes off. Although initially this may seem like a good idea, it could cause some bad things if the timer so happens to go off while scheduler code is running (perhaps the scheduler is invoked due to a thread yielding instead of the timer going off). A better approach would be just to reset the timer manually every time right before you *swapcontext()* into the next thread that's going to run. On top of that, to be extra cautious, it would be a good idea also turn off the timer every time the scheduler is entered so it's not possible for the scheduler to be interrupted in any way.

# 3 Compilation

As you may find in the code and Makefile, your thread library is compiled with PSJF as the default scheduler. To compile simply run *make*:

Command Line

```
$ make
gcc -pthread -g -c mypthread.c
ar -rc libmypthread.a mypthread.o
ranlib libmypthread.a
```

If you are part of CS518 which requires you to implement MLFQ in Section 6, to change the scheduling policy when compiling your thread library, pass *SCHED=MLFQ* with *make*:

Command Line

```
$ make SCHED=MLFQ
gcc -pthread -g -c -DMLFQ mypthread.c
ar -rc libmypthread.a mypthread.o
ranlib libmypthread.a
```

# 4 Benchmark Code

The code skeleton also includes a benchmarks that will help you to verify your implementation and study the performance of your thread library. There are three programs in the benchmark folder:

1. parallelCal (CPU-bound program)
2. vectorMultiply (CPU-bound program)
3. externalCal (IO-bound program)
4. test (empty test program file)

## 4.1 Compiling Benchmarks

To compile the benchmark programs (e.g. parallelCal, vectorMultiply, externalCal, and test) simply run *make* within the benchmark directory:

Command Line

```
$ make
gcc -g -w -pthread -o parallel_cal parallel_cal.c -L../ -lmypthread
gcc -g -w -pthread -o vector_multiply vector_multiply.c -L../ -lmypthread
gcc -g -w -pthread -o external_cal external_cal.c -L../ -lmypthread
gcc -g -w -pthread -o test test.c -L../ -lmypthread
```

## 4.2 Running the benchmarks

**To run the benchmark programs, please see README in the benchmark folder to read understand how to run each one.** Here is an example of running the parallelCal benchmark program with the number of threads (*4*) to run as an argument:

Command Line

```
$ ./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.

## 4.3 Test Program

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 mypthread 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:

Command Line

```
$ make
...
$ ./test
...
```

## 4.4 Seeing how it should run with default Linux pthread library

To understand how the benchmarks work with the default Linux pthread, you could comment out the following MACRO in mypthread.h and the code would start using the default Linux pthread library:

```
#define USE_MYTHREAD 1
```

To use your thread library to run the benchmarks, make sure to uncomment the MACRO in mypthread.h above recompile your thread library and benchmarks.

# 5 Report

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

1. Names and NetIDs of all group members.
2. Detailed logic of how you implemented each API function and scheduler.
3. Benchmark results of your thread library with different configurations of thread numbers.
4. What was the most challenging part of implementing the user-level thread library/scheduler?
5. Is there any part of your implementation where you thought you can do better?

# 6 For CS518 Students

The following are for those registered for the graduate version of Operating Systems Design (CS518). Your job is to expand the scheduling capabilities of your user-level thread library and do a little more analysis.

## 6.1 Expanding scheduling with Multi-level Feedback Queue (MLFQ)

The second scheduling algorithm you need to implement is MLFQ. In this algorithm, you have to maintain a queue structure with multiple levels. Remember, the higher the priority, the shorter time slice its corresponding level of runqueue will be. More description and logic of MLFQ is clearly stated in Chapter 8 of the textbook.

Some hints:

- 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.
- When a thread has finished one "time quantum," move it to the next level of runqueue. Your scheduler should always pick a thread at the top-level of runqueue.

## 6.2 Expanding writeup

Along with the regular writeup details described in Section 5, expand your writeup with the following:

- Detailed description of how you designed your MLFQ scheduler and why.
- Expand benchmark results with MLFQ scheduler
- A short analysis of the benchmark results and comparison of your thread library with Linux's pthread library (see Section 4.4 on how to run with Linux's pthread library)

# 7 Suggested Steps

1. Designing important data structures for your thread library. For example, TCB, Context, and Runqueue.
2. Finishing *mypthread_create*, *mypthread_yield*, *mypthread_exit*, *mypthread_join*, and scheduler functions (with a simple FCFS policy).
3. Implementation of mypthread mutex.
4. Extending your scheduler function with preemptive SJF and MLFQ scheduling policy.

## Submission

For this assignment you will be able to work in groups of up to 2, and in the end your group will have to submit your completed implementations along with a report. To submit your assignment, have one person within your group submit the following files as is. **(Do not compress the files)**:

1. **mypthread.h**
2. **mypthread.c**
3. **Makefile (even if you didn't modify it)**
4. **Any other supporting source files you created**
5. **A report in .pdf format, completed, and with both partners names and NETIDs on the report**

**Important: If one member of the group is registered CS418 and the other is registered for CS518, please upload separate submissions with separate reports. If both members of the group are registered for CS518, having group member submit is okay.**

**Note: If any of the submission rules are not followed, points will be reduced. No exceptions.**

## Other Important Notes

When completing your assignment make sure to keep the following things in mind:

1. **Make sure your programs can compile and run on the iLab machines.** We will be grading your assignments on the iLab machines, so if your programs are unable to compile or run on the iLab machines, points will be deducted. No exceptions. compile the test programs via make
2. **Make sure you can compile you programs with make.** We will be using the Makefile to compile your programs. If for some reason we can not compile your program with make, points will be deducted. No exceptions.

**When in more doubt, Google:** If you have an issue or your program is throwing an error that you don't know how to fix, Google It. Someone, somewhere, probably faced the same issue at some point.

## Frequently Asked Questions

1. **Can I use helper functions in my implementations?** Yes, as long as the library/programs works as intended.
2. **Can I use other *.c or *.h files in my implementation?** Yes, as long as we can use the makefile to compile your programs before running.
3. **Can I modify the makefile?** Yes.
4. **Can we implement the programs on our own computer?** Yes, as long as it can compile and run on the iLab machines as that is where they will be compiled, ran, and graded.

## Additional Questions

If you have any questions about the assignment or are having any issues, post your questions on Piazza.