# CS416/518 Project 2: User-level Thread Library and Scheduler

**Due: 10/24/2025 (100 points)**

**Please read the description and instructions carefully.** In this project, you will learn how to implement scheduling mechanisms and policies. In project 1, you used the Linux pThread library for multi-threaded programming. In project 2, you will get a chance to implement a thread library, scheduling mechanism, and policies inside a thread library. You are required to implement a user-level thread library that has a similar interface compared to the pThread Library. For the multi-thread environment, you must also implement pthread mutexes, used for exclusive access inside critical sections.

Please start early! Also, see Section 10 for tips and resources and Section 11 for plagiarism policies.

### Code Structure

You are given a code skeleton structured as follows:

- thread-worker.h: contains worker thread library function prototypes and definition of important data structures
- thread-worker.c: contains the skeleton of the worker thread library. All your implementation goes here.
- Makefile: used to compile your library. Generates a static library (thread-worker.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.
- sample-code: a folder with sample codes for your understanding (discussed below).

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 the logical steps into a working code.

### Part 1. Thread Library (35 points)

Threads are logical concurrent execution units with their own context. In the first part, you will develop a pthread-like library with each thread having its own context.

**1.1 Thread creation (15 points)** The first API to implement is worker thread creation. You will implement the following API to create a worker thread that executes a function. You could ignore attr for this project.

```
int worker_create(worker * thread, pthread_attr_t * attr,
          void *(*function)(void*), void * arg);
```

Thread creation involves three parts.

**1.1.1 Thread Control Block** First, every worker thread has a thread control block (TCB), which is similar to a process control block that we discussed in class. The thread control block is represented using the *TCB* structure (see *thread-worker.h*). Add all necessary information to the TCB. You might also need a thread ID (a unique ID) to identify each worker thread. You could use the *worker_t* inside the TCB structure to set this during worker thread creation.

**1.1.2 Thread Context** We will start with thread contexts. Each worker thread has a context, needed 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 worker thread context. Because we are developing (and emulating) our scheduler in the userspace, Linux provides APIs to create a user-level context (**ucontexts**) and switch contexts. Briefly, each thread needs a context to save and restore the execution state (as a part of the TCB structure). *ucontext* is a structure that can be used to store the execution state (e.g. CPU registers, pointers to stack, etc.). You will need this to store arbitrary states of execution corresponding to the different threads you will handle.

During worker thread creation (worker_create), *makecontext()* will be used. Before the use of *makecontext*, you will need to update the context structure. You can read more about a context here: *http://man7.org/linux/man-pages/man3/makecontext.3.html*

**Intialization:** During the creation of a worker thread, one can initialize a context using makecontext(), then swap between contexts using setcontext()/swapcontext(). We have added examples of setting up a ucontext using setcontext/swapcontext as well as how getcontext() works (makecontext.c, swapcontext.c, getcontext.c).

*Note 1:* When setting up a new *ucontext*, it is important that a new *ucontext* needs its own stack so that a particular thread of execution has its own space to work on. We recommend allocating a stack via malloc() to avoid any segmentation faults.

*Note 2:* Make sure you allocate enough space for the stack, either allocate a few tens of kilobytes or just use the defined value SIGSTKSZ to specify the number of bytes to allocate. If you allocate a very small amount of space for the stack, you'll run into some issues when a particular thread runs out of stack space. It may or may not try to go out of bounds.

*Note 3:* The sample codes are for conceptual understanding. You need to figure out how to use these concepts in project 2's implementation.

**CAUTION** The sample code only provides basic info for contexts, and you might need to store other information in your context depending on how you implement your scheduler.

Sample Context Codes

- sample-code/makecontext.c
- sample-code/swapcontext.c
- sample-code/getcontext.c

Additional References:

- https://en.wikipedia.org/wiki/Setcontext
- https://linux.die.net/man/3/makecontext
- https://linux.die.net/man/3/swapcontext

**1.1.3 Scheduer and Main Contexts** Beyond the thread context, you would also need a context for running the scheduler code (i.e., scheduler context). The scheduler context can be initialized the first time the worker thread library is called (for example, when worker_create is invoked for the first time). After the scheduler context creation, you would have to switch to the scheduler context (using swapcontext()) anytime you have to execute the logic in the scheduler (for example, after a timer sends a signal for scheduling another thread). Beyond the scheduler context, you can use one more context for the main benchmark thread (that creates workers) and use the context to run the benchmark code. Optionally, another approach is to use one common context for the main benchmark and the scheduler logic. We recommand to use two separate contexts, refer to Section 10 for more hints.

**1.1.4 Runqueue** Finally, once the worker thread context is set, you might need to add the worker thread to a scheduler runqueue. The runqueue has active worker threads ready to run and to wait for the CPU. Feel free to use a linked list or a better data structure to back your scheduler queues. Note that you will need a multi-level scheduler queue in the second part of the project. So, we suggest writing modular code for enqueuing or dequeuing worker threads from the scheduler queue.

**1.1.5 Timers** Timers will help you periodically swap into the scheduler context when a time quantum (i.e., time slice) elapses. To do this, you will need to use setitimer() to setup a timer. When the timer goes off, it will send your program a signal for the corresponding timer.

Attached is an example (see timer.c) of how to set up a timer with setitimer() and register a signal handler via sigaction(). We suggest playing around with setting the different timer values to see how it affects the timer.

Sample timer code

- sample-code/timer.c

Regarding the timer structs, the "itimerval" struct has two "timeval" structs, it_interval and it_value. The it_interval structure is the value that the timer gets reset to once it expires, and the it_value is the timer's current value.

If you set it_interval to zero, you get a one-shot timer, meaning the timer will no longer work until you manually reset it_value back to your time quantum and call setitimer() again. If you set it_interval to a value greater than zero, it will continuously count down, even in your handler, sending a signal until you disarm it. Either one is fine but be wary of how each one will affect your program. If you initially set it_value to zero, the timer will not start after calling setitimer(). It will also kill your current timer if it is running.

*Note 1:* You should use sigaction() instead of signal(), as there may be some cases where the signal handler will be unregistered if you use signal().

*Note 2:* Notice that there are different types of timers; we recommend that you use ITIMER_PROF, as it takes into account the time the user process is running and any time where the system is running on behalf of the user process. This is important if you use a timer that doesn't take into account the system running on behalf of the user. For example, you might get some funky timing intervals if there are any system calls within any of the threads.

*Note 3:* If you use the ITIMER_PROF timer, it will send the SIGPROF signal. So before you start actually implementing your library and scheduler, we highly recommend you take a look at the code provided and start to get familiar with setting timers and creating/swapping ucontexts.

Try out the following if you don't understand using the following sample program: Creating a program that creates two ucontexts to run two functions, foo() and bar(). Within foo(), let it print out "foo" in a never-ending while loop, and within bar(), let it print out in a never-ending while loop. Then use timers and swapcontext() to swap between the two threads of execution every 1 second.

After the above steps, you might have a firm grasp of setting timers, handling the timer signals, ucontext creation, and swapping between the contexts, everything you will need to comfortably start the project. At this point, you can start slowly implementing your library and focus more on the thread creation and scheduling mechanisms and modification as well as scheduler policies.

References:

- https://linux.die.net/man/2/setitimer - http://www.informit.com/articles/article.aspx?p=23618&seqNum=14
- https://linux.die.net/man/2/sigaction - https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/20/lec.html

### 1.2 Thread Yield (5 points)

```
void worker_yield();
```

The *worker_yield* function (API) enables the current worker thread to *voluntarily* give up the CPU resource to other worker threads. That is to say, the worker 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 worker thread back to a runqueue and choose the next worker thread to run. You can read about swapping a context here:

- http://man7.org/linux/man-pages/man3/swapcontext.3.html

*swapcontext() vs setcontext():* swapcontext() saves the context you are switching from and then swaps it out for the next context. Essentially just getcontext() -> setcontext(). setcontext() does not save the current context. It immediately swaps to the next context.

### 1.3 Thread Exit (5 points)

```
void worker_exit(void *value_ptr);
```

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

### 1.4 Thread Join (5 points)

```
int worker_join(worker thread, void **value_ptr);
```

The *worker_join* ensures that the calling application 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 (5 points)** Only creating worker threads is insufficient. Access to data across threads must be synchronized. In this project, you will be designing *worker_mutex*, which is similar to pthread_mutex. Mutex serializes access to a function or function states by synchronizing access across threads.

The first step is to fill the *worker_mutex_t* structure defined in *thread-worker.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 worker thread (or thread's TCB) holding the mutex, as well as any other information.

#### 1.5.1 Thread Mutex Initialization

```
int worker_mutex_init(worker_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
```

This function initializes a worker_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 worker_mutex_lock(worker_mutex_t *mutex);
```

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

```
int worker_mutex_unlock(worker_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 worker_mutex_destroy(worker_mutex_t *mutex);
```

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

### Part 2: Scheduler (45 points)

Since your worker thread library is managed totally in user-space, you also need to have a scheduler and policies in your thread library to determine which worker 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) (10 points)

For the first scheduling algorithm, you are required to implement a pre-emptive SJF (PSJF), 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 must 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" variable defined in *thread-worker.h* (which we have already added), 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 10ms; 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 quantums each thread has ran for.

Here are some hints to implement this particular scheduler:

1) In a worker threads'TCB, maintain an "elapsed" counter, which indicates if the time quantum has expired since the time thread was scheduled. After the time quantum expires, move the worker thread (i.e., work) to the tail of the linked list (or a queue), and schedule a worker thread from the head of the list.

2) 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.

3) Once all worker threads finish, **you must show (a) the total context switches, (b) calculate the entire test application's average response and turnaround times**. As shown below, we have already added a print message with global variables. You are responsible for updating the global variables.

```
//DO NOT MODIFY THIS FUNCTION
/* Function to print global statistics. Do not modify this function.*/
void print_app_stats(void) {

       fprintf(stderr, "Total context switches %ld \n", tot_cntx_switches);
       fprintf(stderr, "Average turnaround time %lf \n", avg_turn_time);
       fprintf(stderr, "Average response time  %lf \n", avg_resp_time);
}
```

HINT: To calculate these metrics, you must first maintain per-thread context switches, response time, and turnaround time. Remember, turnaround time is the time it takes for a thread to complete after first being put into a runqueue. Response time is the time it takes for a thread to first be scheduled after being put into a runqueue. After each thread finishes, update a global running average response time variable and average turnaround time variable. One way to do this is to keep track of the total time for each metric and the total number of threads that have been completed thus far.

4) This is a user-level threading library running in a single kernel thread. Recall what this means and how it relates to the state threads can be in at a certain point in time (SCHEDULED, BLOCKED, READY).

For our tests, we will change the time for the QUANTUM variable and test the total number of context switches and the overall average turnaround time and response time.

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

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 have (please read section 8.5 of the textbook). More descriptions and logic for the

MLFQ scheduling policy is clearly stated in Chapter 8 of the textbook. For this implementation, follow the rules 1-5 of MLFQ from section 8.6 of the textbook. Here are some hints to help you implement:

1) Instead of a single runqueue, you need multiple levels of run queues. It could be a 4-level or 8-level queue as you like. It is suggested to define the number of levels as a macro in *thread-worker.h*.
2) When a worker thread has used up one "time quantum," move it to the next lower runqueue. Your scheduler should always pick the thread at the highest runqueue level.
3) If a thread yields before its time quantum expires, it stays in the current runqueue. But it cannot stay in its current runqueue forever; notice rule 4 of MLFQ.
4) Recall that MLFQ with 1 queue is just RR. Use this knowledge to reuse code from the RR scheduler.
5) Experiment with different values of S for rule 5 of MLFQ. Include some results in your report.

### 2.3 Completely Fair Scheduling (15 points)

The third scheduling algorithm you need to implement is CFS. In this scheduling policy, your goal is to maintain fairness among all threads by ensuring that the virtual runtime (vruntime) of each thread progresses equally. In this project, the vruntime is simply the accumulated runtime of each thread. Instead of using a queue to hold ready threads, you must use a min-max heap, which allows quick access to the thread with the minimum vruntime. Please refer to to wikipedia for more information about this structure. For simplicity, all threads are assumed to have the same priority, so weights are not considered.

**Hints for Implementation**

1. Use the targeted latency (macro TARGET_LATENCY, in milliseconds) to determine the time quantum (time slice) for each thread. The time quantum depends on the number of runnable threads used in the benchmark.
2. The minimum granularity (MIN_SCHED_GRN) is also provided as a macro. If the computed time slice is smaller than MIN_SCHED_GRN, use MIN_SCHED_GRN instead. This ensures that every thread gets to run for at least MIN_SCHED_GRN time units, regardless of the number of runnable threads.
3. When threads are created, insert them into the min-max heap. Initially, all threads have a vruntime of 0.
4. Always pick the thread with the smallest vruntime from the heap. At the beginning, this can be any thread since all runtimes are 0.
5. Run the selected thread for its computed time slice (or until it finishes).
6. Once a thread yields or is context switched out, update its vruntime by adding the time it actually ran. Reinsert it into the heap if it is still runnable.
7. Repeat this process by picking the next thread with the smallest vruntime, ensuring fairness over time.
8. Experiment with different numbers of threads. Include results in your report showing how vruntime ensures fairness across threads.

**Invoking the Scheduler Periodically** For all three 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 worker thread.

For CFS, the timeslice must be calculated based on the number of threads created at the begining of the benchmark.

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

### 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 (PSJF or MLFQ or CFS) to pick a new job.

*Think about conditions when the schedule() method must be called. Other than a timer interrupt, what are the other ways?*

**Thread States** As discussed in the class, worker threads must be in one of the following states. The states help identify a worker thread currently running on the CPU vs. worker threads waiting on the queue vs. worker threads blocked for I/O. So you could define these three states in your code and update the worker 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 PSJF as the default scheduler. To change the scheduling policy when compiling your thread library, pass variables with make:

```
make SCHED=PSJF
(or)
make SCHED=MLFQ
(or)
make SCHED=CFS
```

### 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 worker threads to run as an argument:

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

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

To understand how the benchmarks work with the default Linux pthread, you could comment the following MACRO in thread-worker.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 thread-worker.h and recompile your thread library and benchmarks.

```
#define USE_WORKERS 1
```

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

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

Please note that we will vary the number of threads when testing your code.

### 6. Output - Please follow instructions carefully

After successful run, your code will print an output of the following format. Do not modify the lines of code that print the output. We expect the same output lines. **Also, before making a final submission, please avoid printing other messages except the following stats.** Evertime you issue a print (other than for debugging purpose), the code's performance is impacted.**

```
***************************
Total run time: ... micro-seconds
Total sum is: ...
Total context switches ...
Average turnaround time ...
Average response time  ...
***************************
```

### 7. 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 function and the scheduler logic.
2. Benchmark results of your thread library with different configurations of worker thread number.
3. A short analysis of your worker thread library's benchmark results and comparison with the default Linux pthread library.

### 8. Suggested Steps

Step 0. Read the project write-up carefully and make notes. There is a lot of information. You might need to read them a couple of times patiently.

Step 1: Try out all the sample codes and make sure you understand the logic for how to create, swap, get contexts, and how to use timers and signals.

Step 2. Design important data structures for your thread library. For example, TCB, Context, and Runqueue.

Step 3. Finish worker_create, worker_yield, worker_exit, worker_join, and scheduler mechanisms (you could start with a simple FCFS policy).

Step 4. Implement worker thread library's mutex functions for synchronization.

Step 5. Extend your scheduler function with PSJF, MLFQ, and CFS scheduling and test using the benchmark results and compare against the Linux pthread library.

## 9. Submission

1. Please zip all your code files, Makefile, and benchmark code and upload to Canvas.
2. Also attach a detailed report in PDF format, and include your project partner name and NetID (if any) on the report.
3. Any other support source files and Makefiles you created or modified.

## 10. Tips and Resources

1. Always recompile the benchmarks after recompiling the thread-worker library. The thread-worker library is compiled as a static library and linked into the benchmarks. This means any changes you make to the thread-worker library or scheduler require you to recompile the benchmarks as well.

2. The scheduler must be initialized when threads are created. On the first call to worker_create(), set up the scheduler so it can begin running and scheduling threads. This includes creating the scheduler context, installing the signal handler, starting the timer, and initializing any necessary scheduler data structures.

3. You need separate contexts: one for the scheduler (to run scheduling code) and one for the main() program (to run the benchmark code). Similar to Project 1, you can use getcontext() to initialize a context from the currently running thread. Make sure you allocate and set up a context for the main thread and store it in the appropriate data structures. This ensures the scheduler can return to the main thread after running created threads, so the benchmark code continues executing.

4. Be aware that SIGPROF and ITIMER_PROF only account for CPU time (user + kernel) consumed by your process or threads. While the process is sleeping or blocked, it is not running on the CPU, so the timer does not advance and signals will not be delivered. To avoid this issue, do not rely on sleep or blocking calls when using these timers. If you need to use sleep() for debugging purposes, consider switching to ITIMER_REAL with SIGALRM, which is based on wall-clock time.

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

## 11. Plagiarism and AI Use Policy

Start Early! Systems programming assignments are time-consuming, and starting late almost always leads to incomplete or rushed submissions. Plan your work ahead and give yourself enough time to debug.

Plagiarism in any form will not be tolerated. Sharing code, copying from classmates, or using uncredited external sources is considered academic misconduct and will be reported.

Use of AI tools is allowed only as a reference when you run into issues you cannot solve. The generated code cannot be copied into your submission. This course is designed to teach you systems programming skills, which is why it is offered at the 400-level.

If you use AI tools, you must include a written disclosure in your report with the following details: 1. The name of the tool(s) used 2. The parts of the project where the tool was applied 3. The exact prompts you used 4. Failure to follow these guidelines will be treated as a violation of the academic integrity policy.