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

Due: 10/23/2024; 100 points + extra-credits part (15 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 thread 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. In this project, you will implement a user-level thread library with an interface similar to the pThread Library. For the multi-thread environment, you must also implement pthread mutexes, which are used for exclusive access inside critical sections that we discussed in class. There is also an **optional** extra-credit scheduling championship (described in part 3).

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. You are responsible for converting and extending the logical steps into a working code.

Part 1. Thread Library (50 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 (10 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), similar to a process control block we discussed in class. The thread control block is represented using the *TCB* structure (see *thread-worker.h*). You can add all the 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 to run the thread on a CPU. The context is also a part of TCB. So, once a The TCB structure is set and allocated, and 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 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 using *makecontext*, you must update the context structure. You can read more about the 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 and 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 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 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 Scheduler and Main Contexts Beyond the thread context, you would also need a separate context to run 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 will leave it to you to decide the number of contexts other than the work contexts.

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 to enqueue or dequeue worker threads from the scheduler queue.

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

We have attached 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: Although you used signal() in the previous project to register a signal handler, you should use sigaction() instead, 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, scheduling mechanisms, modification, and 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 (10 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 can 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(). The *setcontext()* does not save the current context. It immediately swaps to the next context.

1.3 Thread Exit (10 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. Consider what things you should clean up or change in the worker thread and scheduler states when a thread is exiting.

1.4 Thread Join (10 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 (10 points) Only creating worker threads is insufficient. Access to data across threads must be synchronized. In this project, you will design *worker_mutex*, 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 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 it will not be able to run until it 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 (40 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) (15 points)

For the first scheduling algorithm, you must implement a pre-emptive SJF (PSJF), also known as *STCF*. Unfortunately, you may have noticed that our scheduler DOES NOT know how long a thread will run for the completion of the 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 or more than one quantum. To implement a mechanism like this, you should also keep track of how much quanta each thread has run 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 the 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 and (b) calculate the entire test application's average response and turnaround times**. We have already added a print message with global variables as shown below. 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 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 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, as well as the overall average turnaround time and response time.

2.2 Multi-level Feedback Queue (25 points)

The second scheduling algorithm you need to implement is MLFQ. In this algorithm, you must maintain a multiple-level queue structure. Remember, the higher the priority, the shorter the 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 are 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 (i.e., level), you need multiple levels of runqueues. Each runqueue represents a priority value.

2) The benchmark applications set threads to four different priority values. We have defined 4 priority values in the header file as macros.

```
#define NUMPRIO 4

#define HIGH_PRIO 3
#define MEDIUM_PRIO 2
#define DEFAULT_PRIO 1
#define LOW_PRIO 0
```

3) Your MLFQ benchmark code calls functions *worker_setschedprio* to set the priority for threads.

```
#ifdef MLFQ
                priority = i % NUMPRIO;
                pthread_setschedprio(thread[i], priority);
#endif
```

4) So, you must implement a simple function in *thread-worker.c* to set the priority value to a worker thread's TCB and use it when adding to a level. If needed, you can change the arguments for worker_setschedprio and experiment with more levels than the priority values.

5) Initially, jobs are added to the runqueue (i.e., level) that represents its thread priority.

6) To schedule the jobs in the same runqueue, please recall that MLFQ with 1 queue is just a round-robin scheduler.

7) 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.

8) 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 in the book and the lecture slides.

9) To prevent threads at the lowest priority always stuck at the lowest runqueue level, you must periodically move the low priority threads to the highest runqueue level to prevent starvation. We will call this "refresh quantum." The refresh quantum is the time after which you would like to move the threads at the lowest runqueue to the highest runqueue. You can define "refresh quantum" as a macro in thread-worker.h. One easy way is to define "refresh quantum" in multiples of time quantum.

10) Experiment with different values for the number of levels, "time quantum," and "refresh time" for rule 5 of MLFQ. Include some results in your report. Ideally you would state the overall runtime of your benchmark for different number of levels, time quantum values, and refresh time, and state the values for which you get the best performance.

11) Your report must clearly describe all the macros defined in the header files.

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 worker 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

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 (STCF or MLFQ) 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 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 STCF as the default scheduler. To change the scheduling policy when compiling your thread library, pass variables with make:

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

5. Benchmark Code

The code skeleton also includes a benchmark that helps you verify your implementation and study the performance of your thread library. There are three programs in the benchmark folder (parallel_cal and vector_multiply are CPU-bound and external_cal 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 SCHED=MLFQ
(or)
make SCHED=PSJF

> ./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 will 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 as you intended. Compiling the test program is done in the same way the other benchmarks are compiled:

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

6. Output - Please follow instructions carefully

After a 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 for the following stats.** Everytime you issue a print (other than for debugging purposes), 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 the results of your thread library with different configurations of worker thread numbers.
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, so you might need to read it a couple of times patiently.

Step 1: Try out all the sample codes and make sure you understand the logic for creating, swapping, and getting contexts and using timers and signals.

Step 2. Design important data structures for your thread library, such as 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 STCF and MLFQ scheduling and test using the benchmark results and compare against the Linux pthread library.

9. Part 3: Extra Credit Scheduling Championship (15 points)

For this assignment, we will run an extra credit championship for those who have completed the project. The goal is very simple: multiply two randomly generated matrices (ensuring the result is correct) using a custom scheduling algorithm in the least amount of time. Your job is to write the matrix multiplication algorithm (in benchmarks/matrix.c) , as well as the scheduling algorithm in the thread scheduling library. For the extra credit, your team will be assigned a team number once you report your first result to us.

The rules are very simple:

1. Your team must have completed the entire project (Parts 1 and 2) and ensured your code compiles, works and runs correctly before starting the extra credit (although you don't have to have scored a 100). If we see that any part of the project is unimplemented, we won't grade the extra credit.
2. You must first develop a matrix multiplication benchmark (*matrix.c*). The benchmark would perform a 3000 x 3000 matrix multiplication that is randomly generated (you can't multiply the identity matrix!). You need to use at least 40 threads to multiply, and you can't tell your threads to sleep.
3. You will then develop a scheduling algorithm and compile it similarly to other scheduling algorithms (e.g., make SCHED=matrix).
4. Your team can manually report your results (time taken for the multiplication) to us over email at any time (making sure to CC both TAs and the instructor). We will occasionally update a simple webpage with the leaderboard, but we will not check your implementation. Make sure to let us know if you manage to improve your results. In the end, we will run your projects ourselves and verify that the times we see are roughly what you reported.
5. When grading the projects, we will take the meantime reported by all teams on the leaderboard and award the full extra credit (15 points) to all those equal to or above it. Those below the mean will receive some points based on the scheduler performance and optimizations. We will make a further announcement about this soon. You should report the improvement in performance with each optimization.
6. This is intended as a fun side activity, so feel free to implement whichever optimizations you want or use whichever scheduling algorithm you want, even if we have not discussed it in class or if you made it up. As long as it adheres to the rules, if it's fast, it's fast!

10. Submission

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

11. Tips and 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