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

Due: 4th March, 2024. Time - 8:00 am ET

Points: 100

Please read the description and instructions carefully.

0. Introduction

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 your own thread library, with scheduling mechanisms and policies inside it. You are required to implement a user-level thread library that has an interface similar to the pThread Library. For the multi-thread environment, you must also implement pthread mutexes, used for exclusive access inside critical sections.

0.1. Code Structure

You are given a code skeleton structured as follows:

- thread_worker_types.h: contains definitions for data structures needed for implementing threads.
- mutex_types.h: contains definitions for data structures needed for implementing mutexes.
- thread-worker.h: contains worker thread library function prototypes
- 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. 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 programs 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 working code.

CAUTION: Since these are userspace threads, you must not use blocking system calls in your implementation.

1. 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 a few different 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_types.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 contexts here:

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

Initialization: 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 (see 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 programs 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 Code

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

Understanding usage of swapcontext() for switching between threads:
The sample-code/yield_swapcontext.c program mentioned above gives the following output:

```
main thread swapping to thread 1
Thread 1 prints 1
Thread 1 prints 2
Thread 2 prints 3
Thread 2 prints 4
Thread 2 returning context to main
swap context back to main executed correctly
```

If you want to get a better understanding of contexts, see if you can figure out how swapcontext works in this program, and then try to modify the code to make it print in the following order:

```
main thread swapping to thread 1
Thread 1 prints 1
Thread 2 prints 3
Thread 1 prints 2
Thread 2 prints 4
Thread 2 returning context to main
swap context back to main executed correctly
```

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 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 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 implement 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 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: 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 "bar" 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 (10 points)

```
void worker_yield();
```

The worker_yield function 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 (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, the return value provided here must 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 (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 should be passed back to this pointer.

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 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 *mutex_types.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.

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 scheduling policy/policies:

- Round Robin Scheduler **(CS416 & CS518 students)**
- Multi-Level Feedback Queue Scheduler **(CS518 students only)**

2.1. Round Robin (RR) (CS416: 40 Points) (CS518: 20 points)

For the first scheduling algorithm, you are required to implement a pre-emptive Round Robin scheduler. You should specify a particular time QUANTUM value; every thread should only be allowed to run for one QUANTUM at a time. The scheduler should allocate every task available in the runqueue in a First In First Out Round Robin manner. To list out the requirements for this scheduler concisely:

- A QUANTUM value must be specified (could be global variable/macro).
- Threads are added to the back of the run queue when they are created or when the transition to the ready state.

- The scheduler must always pick the thread waiting at the front of the runqueue and schedule it.
- After a scheduled thread has run for a time period equal to the QUANTUM, this thread must be descheduled and added back to the runqueue. The scheduler must pick the next available thread to be scheduled.
- 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, RUNNING).

2.2. Multi-Level Feedback Queue (MLFQ) (CS416: Not Required) (CS518: 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 it:

- 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.
- 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.
- 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.
- Recall that MLFQ with 1 queue is just RR. Use this knowledge to reuse code from the RR scheduler.

Invoking the Scheduler Periodically: For both of the 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
- The sample code in sample-code/timer.c has an example to illustrate how this works.

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. 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 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. Compiling

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=RR
(or)
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. To run the benchmark programs, please see the README in the benchmark directory. There are six programs in the benchmark folder:

- **one_thread.c:** Main thread creates one worker thread.
- **multiple_threads.c:** Main thread creates multiple worker threads.
- **multiple_threads_with_return.c:** Main thread creates multiple worker threads. Each thread returns some value in worker_exit(), and the main thread gets this value back in worker_join().
- **multiple_threads_yield.c:** Main thread creates multiple worker threads which periodically call worker_yield().
- **multiple_threads_mutex.c:** Main thread creates multiple worker threads which all update a shared variable via synchronization using a mutex.
- **multiple_thread_different_workload.c:** Main thread creates multiple worker threads. Some workers yield() after performing short bursts of work, while some threads perform long running tasks and do not yield. (useful for checking if the MLFQ implementation works as it is supposed to).

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

```
> make
> ./multiple_threads 5
```

The above example will run the *multiple_threads* benchmark with 5 threads.

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

The benchmark programs will output various messages while they run. Before making your final submission, please make sure that your code does not print any other messages, apart from what the benchmarks already print.

7. Report (10 points)

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

1. What structures you used to implement the TCB, mutex, runqueues, any other queues or structures, etc.
2. What logic you used for implementing the thread and mutex API functions.
3. What logic you used for implementing the scheduler(s). (How it keeps track of which threads are in what states, how it handles transitions between states, how it chooses which thread to pick next, etc.)
4. Collaboration and References: State clearly all people and external resources (including on the Internet) that you consulted. What was the nature of your collaboration or usage of these resources?

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 it a couple of times patiently.

Step 1: Try out all the sample programs 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. Implement worker_create, worker_yield, worker_exit, worker_join, and basic scheduler mechanisms (you could start with a simple FCFS policy without using the timers, if it helps).

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

Step 5. Extend the scheduler logic to implement RR scheduling.

Step 6. Extend or create new data structures (runqueues etc.) for MLFQ scheduling. Ideally, you should be able to reuse many structures and functions implemented for RR.

9. Submission

1. Please add the names and NetIDs of your team members in the *thread-worker.c* file.
2. The report should be in PDF format, and must include the names and NetIDs of your team members.
3. Please create a zip file which contains
   - All your code files, Makefile, benchmark code.
   - Any other support source files and Makefiles you created or modified.
   - Report PDF.
4. Upload this zip file to Canvas. Only one person from a team should submit.

Other Important Things to Note: When completing your assignment make sure to keep the following things in mind:

- 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.

- Make sure you can compile your programs using *make*. We will be using the Makefile to compile your programs. If we cannot compile your program with make, points will be deducted. If you are modifying the Makefile, please ensure that compilation still works before submitting.

10. 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`