Assignment 3 – Threads

CMPT 300 – Operating Systems – Instructor: Nate Payne

Part 1 – Thread Scheduler

```
Please submit a zip folder with the following naming conventions to canvas:

LastName_FirstName_StudentNumber_Assig3

This file should include all code, and a text file called answers.txt that
includes answers to all questions. There may be questions for both part 1 and
part 2, so make sure that you address all questions.

INSTRUCTIONS
============

1. OVERVIEW
===========

In this MP, you will write a user-mode thread scheduler.  The basic purpose
of a scheduler is to multiplex use of the computer across several threads
of execution.  This MP deals with two different scheduling policies: FIFO
and Round Robin.  You will implement both, for use in a simple cooperative
multi-threading system.  Along the way, you'll also learn about implementing
object-oriented constructs in low-level procedural languages like C.

This assignment consists of implementing the core functionality of the
scheduler (Step 4) and answering 10 questions (Step 5).  Code for
Step 4 goes in sched_impl.c and sched_impl.h.

2. THEORY OF OPERATION
======================

The given code in the MP defines the skeleton of a scheduler together with a
parameterized dummy workload.  The idea is when you run the MP, you specify
a scheduling policy, scheduler queue size, some number of worker threads to
create, and, optionally, the number of iterations for which the worker
threads should run.  The basic code that parses command line arguments and
creates these worker threads is provided in the MP, but you must implement
the core synchronization and scheduling operations.

As provided, the MP only includes the "dummy" scheduling algorithm, which
doesn't even try to do anything.  You can run it like this:
  make
  ./scheduler -dummy 0 N   # where N is some number of worker threads
All threads run right away regardless of the queue size (even zero!), and
are scheduled by the operating system.  The goal of this MP is to create
scheduler implementations which are a bit more controlled and predictable.

For example, once you have completed the MP, the following should work:

  ./scheduler -fifo 1 2 3
  Main: running 2 workers on 1 queue_size for 3 iterations
```

```
  Main: detaching worker thread 3075984304
  Main: detaching worker thread 3065494448
  Main: waiting for scheduler 3086474160
  Thread 3075984304: in scheduler queue
  Thread 3075984304: loop 0
  Thread 3075984304: loop 1
  Thread 3075984304: loop 2
  Thread 3075984304: exiting
  Thread 3065494448: in scheduler queue
  Thread 3065494448: loop 0
  Thread 3065494448: loop 1
  Thread 3065494448: loop 2
  Thread 3065494448: exiting
  Scheduler: done!
```

The command line options used above specify:

- -fifo Use FIFO scheduling policy
- 1 One thread can be in the scheduler queue at a time
- 2 Create 2 worker threads
- 3 Each thread runs for 3 time slices

Here's another example:

```
  ./scheduler -rr 10 2 3
  Main: running 2 workers on 10 queue_size for 3 iterations
  Main: detaching worker thread 3075828656
  Main: detaching worker thread 3065338800
  Main: waiting for scheduler 3086318512
  Thread 3075828656: in scheduler queue
  Thread 3065338800: in scheduler queue
  Thread 3075828656: loop 0
  Thread 3065338800: loop 0
  Thread 3075828656: loop 1
  Thread 3065338800: loop 1
  Thread 3075828656: loop 2
  Thread 3065338800: loop 2
  Thread 3075828656: exiting
  Thread 3065338800: exiting
  Scheduler: done!
```

The command line options used above specify:

- -rr Use Round Robin scheduling policy
- 10 Ten threads can be in the scheduler queue at a time
- 2 Create 2 worker threads
- 3 Each thread runs for 3 time slices

Things to observe:

In both examples, the worker threads are created at the beginning of execution. But in the case with queue size 1, one of the threads has to wait until the other thread exits before it can enter the scheduler queue (the "in scheduler queue" messages). Whereas in the case with queue size 10, both threads enter the scheduler queue immediately.

The FIFO policy would actually have basically the same behavior even with a larger queue size; the waiting worker threads would simply be admitted to the queue earlier.

The Round Robin scheduling policy alternates between executing the two available threads, until they run out of work to do.

3. FILE LAYOUT

The MP distribution consists of the following source files:

scheduler.c
  Includes the skeleton of a scheduler (sched_proc()) and a parameterized dummy workload (worker_proc()). The main() function accepts several parameters specifying the test workload (see description below). The scheduler relies on a scheduler implementation (sched_impl_t) to implement the specifics of its scheduling policy (to be provided by you in sched_impl.[hc])

scheduler.h
  Describes the interface to which your scheduler implementation must adhere. The structures containing function pointers are similar to Java interfaces or C++ pure virtual base classes. This file declares that you must define two sched_impl_t structures, sched_fifo and sched_rr in another file (sched_impl.c).

dummy_impl.c
  Implements the dummy scheduling algorithm, which just lets the OS schedule all threads, regardless of queue size.

sched_impl.h (define your data structures here)
  This is where you will define the data structures stored per scheduler instance (struct sched_queue) and per worker thread (struct thread_info). This will likely include synchronization constructs like semaphores and mutexes, and a list of threads available to be scheduled.

sched_impl.c (implement your code here)
  This is where you will define the functions implementing the core behavior of the scheduler, including the FIFO and Round Robin scheduling policies. The only way functions defined in this file are made available to the main program (scheduler.c) is by placing function pointers in the sched_impl_t structures sched_fifo and sched_rr.

list.h
  Defines the basic operations on a bidirectional linked list data structure. The elements of the list, of type list_elem_t, include a void *datum where you can store pointers to whatever kind of data you like. You don't have to use this linked list library, but it will probably come in handy.

list.c
  Implements the linked list operations.

smp4_tests.c
testrunner.c
testrunner.h
  Test harness, defines test cases for checking your MP solution.

Please take a look at the source files and familiarize yourself with how they work. Think about how structures containing function pointers compare to classes and virtual methods in C++. If you'd like to learn more, read about the virtual function table in C++. The struct containing function pointers technique employed in this MP is also used by C GUI libraries like GTK+ and to define the operations of loadable modules, such as file systems, within the Linux kernel.

4. PROGRAMMING

Now you're ready to implement the core of the scheduler, including the FIFO and Round Robin scheduling algorithms. For this purpose, you should only modify sched_impl.h and sched_impl.c. Please see scheduler.h for the descriptions of what functions you must implement. You are free to put whatever you want in the thread_info and sched_queue structures. Note that the only way that the functions you implement are made available to the main program is through the sched_impl_t structures sched_fifo and sched_rr. See dummy_impl.c for a completed example of how to fill in a sched_impl_t.

Suggested approach:

4.1 Create stub versions of all of the functions you will need to implement in sched_impl.c, and statically initialize sched_fifo and sched_rr.

4.2 Figure out how you will implement the scheduler queue, add the appropriate fields to struct sched_queue, and fill in the appropriate queue-related operations in the functions you created in (4.1). Remember that we provide a doubly-linked list in list.[hc].

4.3 Implement scheduler queue admission control, so that only the requested number of threads can be in the scheduler queue at once. Create the appropriate synchronization constructs to prevent threads not in the queue from executing (look at the implementation of worker threads in scheduler.c:worker_proc()).

4.4 Implement the queue operations for selecting the next thread to execute. This will be different for FIFO vs. Round Robin scheduling.

4.5 Add in synchronization constructs to make sure only the selected thread executes at any given time.

4.6 Fill in any gaps that might remain.

When you think you're done, you can test your program using the command "make test". For more thorough testing, the fifo_var and rr_var tests accept queue_size, num_workers, and num_iterations arguments just like the main program (but <num_iterations> is mandatory for the test case):

```
./scheduler -test fifo_var <queue_size> <num_workers> <num_iterations>
./scheduler -test rr_var   <queue_size> <num_workers> <num_iterations>
```

5. QUESTIONS

Q 1 What are some pros and cons of using the struct of function pointers

approach as we did in the MP to link different modules? Does it significantly affect performance? Give some examples of when you would and wouldn't use this approach, and why.

Q 2 Briefly describe the synchronization constructs you needed to implement this MP--i.e., how you mediated admission of threads to the scheduler queue and how you made sure only the scheduled thread would run at any given time.

Q 3 Why is the dummy scheduling implementation provided potentially unsafe (i.e. could result in invalid memory references)? How does your implementation avoid this problem?

Q 4 When using the FIFO or Round Robin scheduling algorithm, can sched_proc() ever "miss" any threads and exit early before all threads have been scheduled and run to completion? If yes, give an example; if no, explain why not.

Q 5 Why are the three variables in scheduler.h declared 'extern'? What would happen if they were not declared 'extern'? What would happen if they were not declared without the 'extern' in any file?

Q 6 Describe the behavior of exit_error() function in scheduler.c. Why does exit_error() not use errno?

Q 7 Does it matter whether the call to sched_ops->wait_for_queue(queue) in sched_proc() actually does anything? How would it affect correctness if it just returned right away? How about performance?

Q 8 Explain how worker_proc() is able to call the appropriate implementation of wait_for_cpu() corresponding to the scheduling policy selected by the user on the command line. Start from main() and briefly explain each step along the way.

Q 9 Is it possible that a worker thread would never proceed past the call to wa->ops->wait_for_cpu(&wa->info) when using one of the scheduling policies implemented in this MP?

Q 10 Explain how you would alter the program to demonstrate the "convoy" effect, when a large compute bound job that never yields to another thread slows down all other jobs in a FIFO scheduled system? See Page 402, Stallings, the paragraph starting "Another difficulty with FCFS is that it tends to favor processor-bound processes over I/O bound processes". Why is it difficult to show the benefits of Round Robin scheduling in this case using the current implementation in the machine problem?

Submit all code for this part of the assignment and answers to questions within you answers.txt file.

Reminder: Do not copy or plagiarize any code from any other student in the course and be sure to cite all online references.

Do not copy or plagiarize from any source online. Any student found doing so will receive a 0 for the assignment portion of the course. My goal is to maximize your learning, so please focus on that!

Part 2 - Scheduler with Signals

This part of the assignment is a variation of the previous part.

In part 1, we built a simulated OS process scheduler. The scheduler can hold only a certain number of processes (workers) at one time. Once the process has been accepted into the scheduler, the scheduler decides in what order the processes execute. We implemented two scheduling algorithms: FIFO and Round Robin.

In this MP, we are to simulate a time-sharing system by using signals and timers. We will only implement the Round Robin algorithm. Instead of using iterations to model the concept of "time slices" (as in the last MP), we use interval timers. The scheduler is installed with an interval timer. The timer starts ticking when the scheduler picks a thread to use the CPU which in turn signals the thread when its time slice is finished thus allowing the scheduler to pick another thread and so on. When a thread has completely finished its work it leaves the scheduler to allow a waiting thread to enter. Please note that in this MP, only the timer and scheduler send signals. The threads passively handle the signals without signaling back to the scheduler.

The program takes a number of arguments. Arg1 determines the number of jobs (threads in our implementation) created; arg2 specifies the queue size of the scheduler. Arg3 through argN gives the duration (the required time slices to complete a job) of each job. Hence if we create 2 jobs, we should supply arg3 and arg4 for the required duration. You can assume that the autograder will always supply the correct number of arguments and hence you do not have to detect invalid input.

Here is an example of program output, once the program is complete:

```
% scheduler 3 2 3 2 3
Main: running 3 workers with queue size 2 for quanta:
 3 2 3
Main: detaching worker thread 3075926960.
Main: detaching worker thread 3065437104.
Main: detaching worker thread 3054947248.
Main: waiting for scheduler 3086416816.
Scheduler: waiting for workers.
Thread 3075926960: in scheduler queue.
Thread 3075926960: suspending.
Thread 3065437104: in scheduler queue.
Thread 3065437104: suspending.
Scheduler: scheduling.
Scheduler: resuming 3075926960.
Thread 3075926960: resuming.
Scheduler: suspending 3075926960.
Scheduler: scheduling.
Scheduler: resuming 3065437104.
Thread 3065437104: resuming.
Thread 3075926960: suspending.
Scheduler: suspending 3065437104.
Scheduler: scheduling.
Scheduler: resuming 3075926960.
Thread 3075926960: resuming.
Thread 3065437104: suspending.
```

```
Scheduler: suspending 3075926960.
Scheduler: scheduling.
Scheduler: resuming 3065437104.
Thread 3065437104: resuming.
Thread 3075926960: suspending.
Scheduler: suspending 3065437104.
Thread 3065437104: leaving scheduler queue.
Scheduler: scheduling.
Scheduler: resuming 3075926960.
Thread 3075926960: resuming.
Thread 3065437104: terminating.
Thread 3054947248: in scheduler queue.
Thread 3054947248: suspending.
Scheduler: suspending 3075926960.
Thread 3075926960: leaving scheduler queue.
Scheduler: scheduling.
Scheduler: resuming 3054947248.
Thread 3054947248: resuming.
Thread 3075926960: terminating.
Scheduler: suspending 3054947248.
Scheduler: scheduling.
Scheduler: resuming 3054947248.
Thread 3054947248: suspending.
Thread 3054947248: resuming.
Scheduler: suspending 3054947248.
Scheduler: scheduling.
Scheduler: resuming 3054947248.
Thread 3054947248: suspending.
Thread 3054947248: resuming.
Scheduler: suspending 3054947248.
Thread 3054947248: leaving scheduler queue.
Thread 3054947248: terminating.
The total wait time is 12.062254 seconds.
The total run time is 7.958618 seconds.
The average wait time is 4.020751 seconds.
The average run time is 2.652873 seconds.
```

The goal of this MP is to help you understand (1) how signals and timers work, and (2) how to evaluate the performance of your program. You will first implement the time-sharing system using timers and signals. Then, you will evaluate the overall performance of your program by keeping track of how long each thread is idle, running, etc.

The program will use these four signals:

SIGALRM: sent by the timer to the scheduler, to indicate another time quantum has passed.
SIGUSR1: sent by the scheduler to a worker, to tell it to suspend.
SIGUSR2: sent by the scheduler to a suspended worker, to tell it to resume.
SIGTERM: sent by the scheduler to a worker, to tell it to cancel.

You will need to set up the appropriate handlers and masks for these signals.

You will use these functions:
- clock_gettime
- pthread_sigmask
- pthread_kill

```
  sigaction
  sigaddset
  sigemptyset
  sigwait
  timer_settime
  timer_create
```

Also, make sure you understand how the POSIX:TMR interval timer works.

There are two ways you can test your code. You can run the built-in grading tests by running "scheduler -test -f0 rr". This runs 5 tests, each of which can be run individually. You can also test you program with specific parameters by running "scheduler -test gen ..." where the ellipsis contains the parameters you would pass to scheduler.

Programming

Part I: Modify the scheduler code (scheduler.c)

We use the scheduler thread to setup the timer and handle the scheduling for the system. The scheduler handles the SIGALRM events that come from the timer, and sends out signals to the worker threads.

Step 1.

Modify the code in init_sched_queue() function in scheduler.c to initialize the scheduler with a POSIX:TMR interval timer. Use CLOCK_REALTIME in timer_create(). The timer will be stored in the global variable "timer", which will be started in scheduler_run() (see Step 4 below).

Step 2.

Implement setup_sig_handlers(). Use sigaction() to install signal handlers for SIGALRM, SIGUSR1, and SIGTERM. SIGALRM should trigger timer_handler(), SIGUSR1 should trigger suspend_thread(), and SIGTERM should trigger cancel_thread(). Notice no handler is installed for SIGUSR2; this signal will be handled differently, in step 8.

Step 3.

In the scheduler_run() function, start the timer. Use timer_settime(). The time quantum (1 second) is given in scheduler.h. The timer should go off repeatedly at regular intervals defined by the timer quantum.

In Round-Robin, whenever the timer goes off, the scheduler suspends the currently running thread, and tells the next thread to resume its operations using signals. These steps are listed in timer_handler(), which is called every time the timer goes off. In this implementation, the timer handler makes use of suspend_worker() and resume_worker() to accomplush these steps.

Step 4.

Complete the suspend_worker() function. First, update the info->quanta value.

This is the number of quanta that remain for this thread to execute. It is initialized to the value passed on the command line, and decreases as the thread executes. If there is any more work for this worker to do, send it a signal to suspend, and update the scheduler queue. Otherwise, cancel the thread.

Step 5.

Complete the cancel_worker() function by sending the appropriate signal to the thread, telling it to kill itself.

Step 6.

Complete the resume_worker() function by sending the appropriate signal to the thread, telling it to resume execution.

Part II: Modify the worker code (worker.c)

In this section, you will modify the worker code to correctly handle the signals from the scheduler that you implemented in the previous section.

You need to modify the thread functions so that it immediately suspends the thread, waiting for a resume signal from the scheduler. You will need to use sigwait() to force the thread to suspend itself and wait for a resume signal. You need also to implement a signal handler in worker.c to catch and handle the suspend signals.

Step 7.

Modify start_worker() to (1) block SIGUSR2 and SIGALRM, and (2) unblock SIGUSR1 and SIGTERM.

Step 8.

Implement suspend_thread(), the handler for the SIGUSR1 signal. The thread should block until it receives a resume (SIGUSR2) signal.

Part III: Modify the evaluation code (scheduler.c)

This program keeps track of run time, and wait time. Each thread saves these two values regarding its own execution in its thread_info_t. Tracking these values requires also knowing the last time the thread suspended or resumed. Therefore, these two values are also kept in thread_info_t. See scheduler.h.

In this section, you will implement the functions that calculate run time and wait time. All code that does this will be in scheduler.c. When the program is done, it will collect all these values, and print out the total and average wait time and run time. For your convenience, you are given a function time_difference() to compute the difference between two times in microseconds.

Step 9.

Modify create_workers() to initialize the various time variables.

Step 10.

Implement update_run_time(). This is called by suspend_worker().

Step 11.

Implement update_wait_time(). This is called by resume_worker().

Questions
==========

Question 1.
Why do we block SIGUSR2 and SIGALRM in worker.c? Why do we unblock SIGUSR1 and SIGTERM in worker.c?

Question 2.
We use sigwait() and sigaction() in our code. Explain the difference between the two. (Please explain from the aspect of thread behavior rather than syntax).

Question 3.
When we use POSIX:TMR interval timer, we are using relative time. What is the alternative? Explain the difference between the two.

Question 4.
Look at start_worker() in worker.c, a worker thread is executing within an infinite loop at the end. When does a worker thread terminate?

Question 5.
When does the scheduler finish? Why does it not exit when the scheduler queue is empty?

Question 6.
After a thread is scheduled to run, is it still in the sched_queue? When is it removed from the head of the queue? When is it removed from the queue completely?

Question 7.
We've removed all other condition variables in SMP4, and replaced them with a timer and signals. Why do we still use the semaphore queue_sem?

Question 8.
What's the purpose of the global variable "completed" in scheduler.c? Why do we compare "completed" with thread_count before we wait_for_queue() in next_worker()?

Question 9.
We only implemented Round Robin in this SMP. If we want to implement a FIFO scheduling algorithm and keep the modification as minimum, which function in scheduler.c is the one that you should modify? Briefly describe how you would modify this function.

Question 10.
In this implementation, the scheduler only changes threads when the time quantum expires. Briefly explain how you would use an additional signal to allow the scheduler to change threads in the middle of a time quantum. In what situations would this be useful?

Submit all code for this part of the assignment and answers to questions within you answers.txt file.

Reminder: Do not copy or plagiarize any code from any other student in the course and be sure to cite all online references.

Do not copy or plagiarize from any source online. Any student found doing so will receive a 0 for the assignment portion of the course. My goal is to maximize your learning, so please focus on that!