CSE 451/851 Programming Assignment 3:

Scheduler with Signals

1. Overview

In this PA, we are to simulate a time-sharing system by using signals, timers, and the Round Robin scheduling algorithm. Instead of using iterations to model the concept of "time slices", 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 PA, only the timer and scheduler send signals. The threads passively handle the signals without signaling back to the scheduler. This assignment consists of implementing the scheduler and answering 9 questions.

2. Due Dates and Handin Instructions.

Due Date: Monday Apr 2 at 11:59PM.

Use web handin to handin your assignments. Handin this assignment as Programming Assignment #3. Include the Makefile provided with your assignment, running make in your project directory generates an executable called scheduler.

Include a README file in with your code. Use the README file to:

1. Document any problems you had or special instructions to run your program.
2. Include a paragraph on what you learned in doing this programming assignment.

Please include all the source files provided into a zip file with the following naming convention: **<cse_account_name>_pa3.zip**

3. Discussion

The goal of this assignment 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.

The Round Robin scheduling policy alternates between executing the two available threads, until they run out of work to do. A sample executable rr_fifoscheduler is available on canvas; where you can see how round robin works in comparison to FIFO scheduling policy.

Here's an example that shows how round robin works:

```
./rr_fifoscheduler -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

An example how FIFO works:

```
./rr_fifoscheduler -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

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. In general, the Round Robin scheduling policy can alternate between executing up to queue size number of available threads in the scheduler queue.

4. Program Specification

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 auto-grader 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.
```

You can download the sample program “scheduler” from here canvas.

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 your program with specific parameters by running "scheduler -test gen ..." where the ellipsis contains the parameters you would pass to scheduler. The basic code that parses command line arguments and creates the worker threads is provided, you are to implement the scheduler with signals and timers.

You would be making use of the following code:

list.h
Defines the basic operations on a bidirectional linked list data structure. The elements of the list allow you to 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.

smp5_tests.c, testrunner.c, testrunner.h
Test harness, defines test cases for checking your solution.

scheduler.c
Implements the scheduling.

scheduler.h
Describes the interface to which your scheduler implements. Please take a look at the source files and familiarize yourself with how they work. Think about how structures containing function pointers in comparison 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 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.

You can download these files from canvas.

Do not modify any of the test programs and makefile.

5. 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 accomplish 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().

Validation

Please validate the program with the auto-grader provided.

The auto grader does not check for the validity of the total wait and run time. Please ensure that the total wait and run times produced by your scheduler matches (approximately) with the total wait and run times produced by the scheduler provided.

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. 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 8. We only implemented Round Robin in this PA. 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 9. 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?