CSE 451/851 ~~Programming Assignment 4~~ Extra Credit

Scheduler with Signals

Assigned: Mar 19, 2020
Due: May 8, 2020 23:59:59

Introduction

In this programming assignment, we will simulate a time-sharing system by using signals, timers, and a round robin scheduling algorithm. Instead of using iterations of a `while()` loop to model the concept of "time slices," we will 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. Only the timer and scheduler send signals. The threads passively handle the signals without signaling back to the scheduler.

In this assignment you will be modifying code we give you to provide your full solution.

Submission procedure

Do the following:

1. Make sure the Makefile in the `PA4/code` folder generates an executable called `scheduler` when you type `make` at the command line.
2. Modify/fill in the README file.
3. Include all the source files and README and zip the whole directory up into a zip file named `<UNL username>_pa4.zip` and submit the single zip file to CSE web handin.

Getting Started

1. Download the zip file `PA4.zip` and unzip it. Use the directory structure, README, and Makefile provided. You shouldn't need to modify the Makefile.
2. As in previous programming assignments, you can use the provided binary, `scheduler`, to see what your program **should do**. `rr_fifoscheduler` is

Grading

In Section 4 below you will read about how to test your program. There are 5 tests. Each test will count for ~~20~~ 10 points. If you pass the test you get the full points, if you fail, you get 0 points.

Preliminary discussion

The goal of this assignment is to help you understand how signals and timers work, as well as how scheduler performance can be assessed. You will implement the time-sharing system using timers and signals, and then 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`

Make sure you understand how the POSIX interval timer works. You can read about it in the man pages `http://man7.org/linux/man-pages/man2/timer_create.2.html`.

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 in the `PA4` directory you downloaded. This lets you see how round robin works in comparison to a FIFO scheduling policy (`rr_fifoscheduler`).

Examples

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 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.

1 Procedure

Program Specification The program takes a variable number of arguments:

Arg1 determines the number of jobs created (threads implemented)
Arg2 specifies the queue size of the scheduler
Arg3-ArgN gives the duration (the required number of time slices) of each job. Hence if we create 2 jobs, we should supply an 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.

Files The basic code that parses command line arguments and creates the worker threads is provided and you are to implement the scheduler with signals and timers. Please take a look at the source files and familiarize yourself with how they work. You will be making use of the following files:

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 which your scheduler implements.

Do not modify any of the test programs or makefile.

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

We will 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()` 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 scheduling, 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.

1.2 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 also need 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.

1.3 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()`.

1.4 Testing and validation

There are several ways you can test your program:

- You can test your program with specific parameters by running `./scheduler -test gen ...` where the ellipsis contain the parameters you would pass to scheduler (e.g., `./scheduler -test` `gen 3 2 3 2 3`). If you use this you must be careful to have the right number of arguments as described above in this section. If you have the wrong number of arguments the test defaults to `pass` (false positive).

- You can also run the individual built in tests by running `./scheduler -test <testname>` where <testname> is one of:
  - `test_3_1_2_2_2`
  - `test_2_2_2_2`
  - `test_5_7_1_2_1_2_1`
  - `test_4_1_1_2_3_4`
  - `test_3_3_4_3_2`
- Finally, you can run all the built-in grading tests by running `./scheduler -test -f0 rr`. This runs the 5 tests above and will be how we determine your grade according to the "Grading" section above.

For grading we will use `./scheduler -test -f0 rr`. However, the grader does not check for the validity of the total wait and run time. **Make sure** the total wait and run times produced by your scheduler approximately matches the total wait and run times produced by the scheduler provided.

Appendix: Example output

Here is an example of program output using `scheduler` in the `PA4` directory you downloaded:

```
foo@cse:~/PA4> ./scheduler 3 2 3 2 3
Main: running 3 workers with queue size 2 for quanta:
 3 2 3
Main: detaching worker thread 140449784538880.
Main: detaching worker thread 140449776146176.
Scheduler: waiting for workers.
Scheduler: waiting for workers.
Scheduler: waiting for workers.
Scheduler: waiting for workers.
Scheduler: waiting for workers.
Scheduler: waiting for workers.
Scheduler: waiting for workers.
Scheduler: waiting for workers.
Scheduler: waiting for workers.
Scheduler: waiting for workers.
Main: detaching worker thread 140449767753472.
Main: waiting for scheduler 140449792931584.
Scheduler: waiting for workers.
Scheduler: waiting for workers.
Scheduler: waiting for workers.
Thread 140449776146176: in scheduler queue.
Thread 50599680: suspending.
Thread 140449784538880: in scheduler queue.
Thread 58992384: suspending.
Scheduler: scheduling.
Scheduler: resuming 140449776146176.
Thread 50599680: resuming.
Scheduler: suspending 140449776146176.
Scheduler: scheduling.
Scheduler: resuming 140449784538880.
Thread 50599680: suspending.
Thread 58992384: resuming.
Scheduler: suspending 140449784538880.
```

```
Scheduler: scheduling.
Thread 58992384: suspending.
Scheduler: resuming 140449776146176.
Thread 50599680: resuming.
Scheduler: suspending 140449776146176.
Thread 140449776146176: leaving scheduler queue.
Thread 50599680: terminating.
Scheduler: scheduling.
Scheduler: resuming 140449784538880.
Thread 58992384: resuming.
Thread 140449767753472: in scheduler queue.
Thread 42206976: suspending.
Scheduler: suspending 140449784538880.
Scheduler: scheduling.
Thread 58992384: suspending.
Scheduler: resuming 140449767753472.
Thread 42206976: resuming.
Scheduler: suspending 140449767753472.
Scheduler: scheduling.
Thread 42206976: suspending.
Scheduler: resuming 140449784538880.
Thread 58992384: resuming.
Scheduler: suspending 140449784538880.
Thread 140449784538880: leaving scheduler queue.
Thread 58992384: terminating.
Scheduler: scheduling.
Scheduler: resuming 140449767753472.
Thread 42206976: resuming.
Scheduler: suspending 140449767753472.
Scheduler: scheduling.
Scheduler: resuming 140449767753472.
Thread 42206976: suspending.
Thread 42206976: resuming.
Scheduler: suspending 140449767753472.
Thread 140449767753472: leaving scheduler queue.
Thread 42206976: terminating.
The total wait time is 12.001326 seconds.
The total run time is 7.999737 seconds.
The average wait time is 4.000442 seconds.
The average run time is 2.666579 seconds.
```