Savvy Scheduler

Entire Assignment due **2019-04-17 23:59**
Graded files:

- libscheduler.c

Content

Learning Objectives

The learning objectives for Savvy Scheduler are:

- scheduling algorithms
- preemptive vs non-preemptive
- Relating the different algorithms with a priority queue

Introduction

In this lab you'll be writing a scheduler. Rather than interacting directly with the operating system, we've whipped up a userspace threading library!

Before you start

Think about how to implement a scheduler!

Try to answer these questions…

1. What do you do for incoming jobs?
2. How do you sort your job so that you can find the next job according to different scheme?
3. What kind of data structures do you need?

What a scheduler does is to put all the jobs it gets in a queue and then sort them in some order (related to scheme). Scheduler gives them to CPU one by one. The key in scheduler is the scheme it use, and the choice of scheduling algorithm depends on many factors. For example, First Come First Serve (FCFS) is really easy to implement but might keep a short job waiting really long for a long process at the front.

So now we know that a scheduler puts jobs in a queue, sort them, and give them to CPU in some order. Then what will be the best data structure to store these jobs? Priority queue can do this job really well! A priority queue is a queue with a really nice feature. It puts every incoming node in correct position so that the queue is always ordered. Therefore, you don't need to call `sort()` every time you get a new node (job). And you can simply give them out by pulling out the first element of the queue.

An important question now is: "What do you mean by **ordered**?" Let's take FCFS scheduling for example. Ideally, scheduler should be able to:

- Receive a job.
- Put it in a queue with some priority.
- Pull the job with highest priority and give to the CPU.

And since we are doing FCFS, we want the element that comes first to be at the front of the queue. So you should give jobs arriving earlier higher priority and jobs arriving afterwards lower priority.

Take Shortest Job First (SJF) for another example. You can give those jobs that can be finished faster higher priority. As you can see here, the key to implement a scheduler is to decide **PRIORITY**. And the way to decide priority in a priority queue is by giving a comparator function. By defining a job A is better than a job B in a priority queue, we mean that A has higher priority than B.

So basically, half of your job in this lab is simply writing a comparator function that helps you decide which job has higher priority.

Mission

Background: Priority Queue

To build a scheduler, a fundamental data structure is a priority queue. You do not need to implement one, but should read and understand `libpriqueue`, our priority queue library. You will be using this library in your scheduler.

Scheduler

You will need to implement scheduling callbacks for a userspace threading library.

The scheduling algorithms you are going to implement are:

- First Come First Served (FCFS)
- Preemptive Priority (PPRI)
- Priority (PRI)
- Preemptive Shortest Remaining Time First (PSRTF)
- Round Robin (RR)
- Shortest Job First (SJF)

You can read up on scheduling in the scheduling chapter (http://cs241.cs.illinois.edu/coursebook/Scheduling)

You should use the priority queue that we provided to help you complete this part of the lab.

To complete this lab, you must implement the six comparator functions and eight scheduler functions (and one optional one) defined in `libscheduler.c`. These functions are self-descriptive, but a full function outline for each function is provided for you in the file. These functions will be utilized by the green threading library (Check out the section on Gthreads below).

You might want to understand how scheduler works. So we put a detailed explanation in the bottom of this webpage.

Directions

To help you finish this lab efficiently, we recommend you to follow these steps:

1. Understand when your function will be called.
2. Try to write pseudocode for each comparator first and see what kind of information you will need. For example, you probably need the arrival time of each job so you can implement FCFS by setting priority according to time.
3. Create data members in `job_info` you need for step 2.
4. Go back and complete your comparator functions.

The second part of the lab is to set up scheduler itself and manage incoming jobs and completed jobs. Now you should implement those functions related to the CPU (like `scheduler_new_job()`, `scheduler_job_finished()`, `scheduler_quantum_expired()`).

1. Take a look at all these functions, write some pseudocode to realize your thoughts.
2. You might need to implement some helper functions to help you write these functions.
3. Finish these functions.

The last part of your job is computing stats and clean-up, which is fairly trivial. You may need some extra variables to help you keep track of these stats.

Job Struct

We have provided a job struct defined in libscheduler.h. You do not need to modify the **state**, **ctx**, or **stack_start** fields. The only field you will be using or modifying is **metadata**, where you must insert your **job_info** struct that you will define in **libscheduler.c**.

Functions you need to implement

The only graded file is **libscheduler.c**. You will need to augment the `job_info` struct and implement the following functions:

void scheduler_start_up(scheme_t s)

This function is implemented for you, but you can add some code to it if you need to initialize any global variables.

int comparer_fcfs(const void *a, const void *b)

Comparer function for fcfs scheme

int comparer_ppri(const void *a, const void *b)

Comparer function for ppri scheme

int comparer_pri(const void *a, const void *b)

Comparer function for pri sscheme

int comparer_psrtf(const void *a, const void *b)

Comparer function for psrtf scheme

int comparer_rr(const void *a, const void *b)

Comparer function for rr scheme

int comparer_sjf(const void *a, const void *b)

Comparer function for sjf scheme

void scheduler_new_job(job *newjob, int job_number, double time, scheduler_info *sched_data)

This function is responsible for setting up a new job. You must populate `newjob->metadata` with the information found in `scheduler_info` . The contents of `newjob->metadata` cannot be a pointer to `sched_data` since this variable may be on the stack of the calling function. Once you've set up `newjob` offer it to the queue.

job *scheduler_quantum_expired(job *job_evicted, double time)

This function is called at the end of every time quantum. If there is no job currently running, `job_evicted` will be NULL. If the current scheme is not preemptive and `job_evicted` is not NULL, return `job_evicted` . In all other cases, if `job_evicted` is not NULL, place it back on the queue and return a pointer to the next job that should run. (Note, it is possible for the next job to be the same as `job_evicted`)

void scheduler_job_finished(job *job_done, double time)

This function will be called when a job is finished. You should update any statistics you are collecting and free any buffers you may have allocated for this job's metadata.

double scheduler_average_waiting_time()

This function returns the average waiting time across all jobs so far.

double scheduler_average_turnaround_time()

This function returns the average turnaround time across all jobs so far.

double scheduler_average_response_time()

This function returns the average response time across all jobs so far.

void scheduler_show_queue()

This is an optional function that you can implement for debugging purposes.

Testing

We have provided three interfaces to testing this lab. The first testing interface is **scheduler_test.c**. This file allows you to directly test the functions you've implemented in libscheduler.c. We recommend using this one, and have already populated it with a few test cases of our own! This is how the autograder will be testing your code. There are 10 tests that we have provided you. You can run each test with the following:

```
./scheduler_test [test_no]
```

On success the test will return zero (you can check the return code with `echo $?`). On failure one of the asserts will cause an error.

The second testing interface is write your own tests similar to the ones in **main.c** in **gtest.c**. You can then run `./gtest` and inspect gthreads.log or any printouts to check if your implementation works.

The final testing interface should be thought of as more of a way to gain intuition behind the concepts rather than a way to test your code for correctness. This method is to run your scheduler with the green threading library. You can see example code in **main.c** which you will not be able to edit. The expected outputs for the various scheduling algorithms are stored in **examples/** and you will be able to diff your output to see if your scheduler produces the correct output. There is an example below.

```
./main fcfs > test.txt
diff test.txt examples/expected_fcfs.txt
```

For your convenience, we've wrapped this with the bash script **testall.sh**. Running `./testall.sh` will run all the schemes and diff them with the expected output to check if your implementation is correct. If you'd like to test specific schemes, you can pass those in as arguments, for example `./testall.sh rr fcfs pri` will only test round robin, first-come-first-serve and priority.

You can test the output of gthread.log for certain scheduling schemes using the script `check_scheduler.py`. To do this, you'll first have to run `./main fcfs`, and then `python3 check_scheduler.py fcfs` - it will check if your **gthread.log** file is correct for the given scheduling scheme. This is more reliable than the **testall.sh** script. Currently, only FCFS, SJF, and PRI scheduling schemes are supported.

However, since this method of testing relies on outputs generated every second, it may not accurately reflect the schedulers behavior, and may falsely report your solution as correctly working. To get around this, you can also take a look at the generated log in **gthread.log**. This contains information about each thread's context switches and you can manually inspect it to see if it does what you expect.

Using the log file, we have built a visualizer for this data that will display which thread ran in approximately 500ms intervals. Note that the visualizer doesn't perfectly display all the context switches, and performs especially badly with round robin output. However, it works well for schemes such as sjf and will display the following if **gthread.log** contains the log of running `./main sjf`:

```
['d10', 'd40', 'd70', 'da0', 'dd0']
d10: ++
d40:   +++
d70:      +++++
da0:           +++++++
dd0:                  ++++++++
```

There are couple things to note about this output. The first is that each '+' represents something slightly more than half a second so we interpret every two '+'s as something close to 1s. The second thing is to note that the thread names are replaced by a uid in the log file. You can which ones corresponding to which by looking through the log for registration messages and the code to check which order the threads were created.

Spooky bugs

This lab has some strange gotchas when testing with the gthread library. For this reason, we recommend using (and augmenting) the tests in `scheduler_test.c` . If you notice any random segfaults or freezes in the program that occur non-deterministically (maybe only once or twice every 10 runs) please report this to us so we can get that patched! (This will not affect grading since the grader will directly test the functions in **libscheduler.c** as opposed to the actual context switches generated by the green threading library.

The threading model - gthreads (optional reading)

here
Initial idea and code from (https://c9x.me/articles/gthreads/code0.html)

Gthreads is an implementation of green threads in c! It uses libscheduler to schedule the threads.

NOTE: The green thread scheduler uses alarms and a SIGALRM handler, it is an error to use some other handler for SIGALRM with gthreads.

Constants

There are two constants defined as enums in gthread.h, MaxGThreads (not used, remove) and StackSize. Stack size determines the size of the stack for each green thread spawned.

Gthread Functions

gtinit

This function should be called before anything else

This function sets up the scheduler and a signal handler for SIGALRM. It is undefined behavior to call any other function in gthread before this one.

Takes in a scheme_t detailing what scheduling algorithm to be used.

gtgo

This is like pthread_create. It spawns a new green thread (won't start until it's actually scheduled).

It takes in the function to execute and a scheduler_info* to get it's scheduler attributes.

gtstart

This function starts off the scheduling process with a call to ualarm.

gtret

This function should be called from a thread to clean up and exit. If called from main it will wait for all other threads to finish before exiting with the status as the argument to gtret. If any other thread calls this function, it is equivalent to calling return.

gtsleep

This function will sleep for at least the number of seconds specified by the argument. Unlike sleep(2), this function will also call yield and allow another thread to run (if there is one on the queue).

gtdoyield

This function is a wrapper around the internal function gtyield and might perform a context switch to another thread. The return value will be true if a context switch occurred and false otherwise. The argument is not important, so long as it is not -1 or SIGALRM.

gtcurrjob

Returns a job* indicating the current running job

Okay, so how does this work?

The idea of green threads is essentially to have a user-space thread that is lighter than a pthread, but at the cost of not executing in parallel. Instead a green thread will switch between many “threads of execution” giving the illusion of parallel processing. As you might have guessed, this switch involves what we call a “context switch” that allows us to save the current state of the thread before moving to a different one.

To learn more about the concept, read this green threading intro (https://c9x.me/articles/gthreads/code0.html) article that we used as a starting point.

Now, let’s talk about what we’ve added on top of that very simple implmentation. We need a way to preempt threads. For our purposes, a signal is an acceptable solution to this problem. We have used the function ualarm(3) to schedule alarms on regular intervals. The handler then calls gtyield which will call scheduler_quantum_expired to select the next job to run.

Note that almost all the scheduling magic is implemented in libscheduler! The only exception is that the main thread will never be sent to any of the functions in libscheduler. Instead, every other quanta, gtyield will store the current job do a context switch to main, and the next time gtyield is called from main, the process will switch back to the stored job.