Scheduling

This lab will be exploring the design of several scheduling algorithms and compairing their response times and other features. First, **please** take a look around the provided code in the project folder. Nearly all of this has been finshed for you; however, it is good to look through it and understand how the different parts work.

Background on scheduling

An intrinsic part of the execution model of an operating system, scheduling is needed to make sure that a task that requires a resource gets access to that resource to complete. To perform the scheduling, a scheduler is needed. A scheduler may try to schedule the tasks in such a way that it maximizes throughput, minimizes latency, maximizes fairness, etc. To achieve these goals, different algorithms for scheduling the tasks are used. Some of the scheduling algorithms are summarized below.

First Come, First Served

The First Come, First Served(FCFS) scheduling algorithm schedules tasks to run on the processor in the order in which they arrive. Each task that is scheduled then runs until a block, regardless of how long the task takes.

Round Robin

The Round Robin(RR) scheduling algorithm schedules tasks in a similar fashion to FCFS, in that the first task that arrives gets to go next; but the tasks have a fixed period of time, referred to as a quantum, to execute on the processor. If the task has exhausted its quantum, it is then pre-empted, and the next task in the queue is allowed to execute on the processor. The pre-empted task then gets inserted back into the end of the queue, where it will eventually get its turn to execute again.

Priority Round Robin

Priority Round Robin(PRR) is an extension of the Round Robin scheduling algorithm. Each process now has a priority value, and the tasks which are part of the processes with the higher priority get scheduled before the tasks which are part of the processes with lower priority, i.e., a task for process A which has priority 3 will be scheduled before a task for process B which has priority 2, if they both arrive at the same time.

PRR has a RR queue for each priority level. Tasks within the higher priority queues must be exhausted before tasks within the lower priority queues can be scheduled. Each queue may have its own quantum value that can be different from the quantum values of the other queues. Usually the quantum of the higher priority queues are smaller than the quantum of the lower priority queues. The tasks within each queue is scheduled and executed in a round robin fashion.

Shortest Remaining Time Next

Also known as Shortest Job Next(SRTN or SJN), is a scheduling algorithm where when either a task completes execution or a new task arrives, all the available tasks to execute are evaluated based on remaining time of execution. The task with the least remaining execution time is selected to execute next. This is not a scheduling algorithm that can be implemented in the real world, due to the fact that tasks generally do not know how long they will be executing before blocking.

Setting up the lab

In this lab you will notice a number of sub-directories. The directories are listed below with a brief description.
Scheduler: this directory contains the simulator that will test various scheduling algorithms. The code in this directory is finished and has been provided for you.
SimTaskGen: This contains a program that will create a simulated list of tasks to provide to the simulator. There are a number of knobs you can use to create different task workloads which are in the `main.c` file.
Algorithms: this directory contains the skeleton code for the algorithms you will write for this lab. The First Come First Serve (FCFS) has been finished for you as an example.
Wavedrom: The wavedrom program which will view the output waves

of the simulator.

Compile Scheduler

Enter the `Scheduler` directory and run `make` to compile the simulator. The simulator requires c++ to be installed if you are working on your personal computer.

Generate a task list

Next enter the `SimTaskGen` directory and run `make` to compile the task generator. Run the generator with the default options by issuing `./main`. This will generate 26 tasks with random arrival time, run times, and priorities. The output is `out.json`.

Compile the algorithm

Enter the `/Algorithm/fcfs` directory. This contains the source code for the FCFS scheduling algorithm. More detail on this example will be given later, but for now compile it by running `make`. The output will be a file called `sched.mod` which is a runtime loadable module which can be used by the scheduler simulator.

Testing the algorithm

Return to the `Scheduler` directory. In this directory is another file called `fcfs.cfg` which is the configuration for the First Come First Serve algorithm. The four fields in this file are:

Algorithm: the `sched.mod` file that is this algorithm.
Tasks: the output of the SimTaskGen program. In this case it is a symbolic link to that output
Wave: The name of the output wave file
Log: the name of the output log file.

To test the algorithm call the simulator with the following command:

```
$ ./main -c fcfs.cfg
Registered scheduler FCFS
```

Looking in the `Scheduler` directory you should notice two new output files: `fcfs-wave.json` and `fcfs.log`. You can view the log using a text editor. The log lists each time a task arrived, finished, blocked, and unblocked. Also at the end it lists each task along with a number of properties. To view the waveform open `index.html` in the `Wavedrom` directory in a web browser. Select `Choose File` and select the `fcfs-wave.json` file in the `Scheduler` directory. It may take several seconds to render the waveform. The top line shows the state of what the CPU is actually running, and the subsequent lines show state of each task in the system. A solid color with a letter in it is a running task, a hashed out section is a task blocking for I/O, and a solid line is a task ready to run but not being given time on the CPU. The dotted lines are tasks that have either not arrived or that have finished.

The FCFS Algorithm

Return to the `fcfs.c` file in the `Algorithms/fcfs` directory. You will notice that the file closely resembles that of a Linux kernel module. The `init_module()` will be called when the module is first loaded into the simulator, and the `cleanup_module()` will be called when the module is unloaded from the simulator. The `init_module()` function is calling `register_scheduler()` which will register a new scheduling algorithm with the simulator. The function takes two parameters: a c-string name for the algorithm, and a `scheduler_operations` object. The `struct scheduler_operations` is defined in `Scheduler/include/scheduler_algorithms.h` and contains three callbacks and one variable which are defined in that file.

You will notice `struct task` a lot in this file. This is the actual task objects that are being scheduled and is declared in `Algorithm/include/task.h`. The `task` contains three fields: `next` and `prev` which can be used by the algorithm to create a linked list, and `scheduler_date` which is a null pointer that can be used to hold algorithm information about this task. There is also a read-only `task_info` field which contains important information about the task.

The First Come First Serve algorithm is very simple. It always runs the task that arrived first until that task can not run any longer. Therefore it simply can maintain a linked list of tasks where the oldest is the head and the newest is the tail. Every time a new task arrives that task is placed on the tail, and the running task is always the head. Look at `fcfs_enqueue()` and `fcfs_dequeue()` to understand how the linked list is working. The two functions are added to the `scheduler_operations` at the very bottom of the file. FCFS does not require a periodic timer so the last two fields of the `scheduler_operations` can be left as default.

Round Robin

Next copy the FCFS algorithm to a new folder called `rr`. The round robin scheduler is just like `fcfs` except the `tail` of the linked list should wrap around to the `head`. Modify the `enqueue` function to match below:

```c
struct task* rr_enqueue(struct task* r, struct task* t)
{
    if(head == NULL)
    {
        head = t;
        t->next = head;
        t->prev = head;
    }
    else
    {
        head->prev->next = t;
        t->prev = head->prev;
        head->prev = t;
        t->next = head;
    }
    return head;
}
```

The `dequeue` function is also the same as `fcfs` except care must be taken to close the loop after the head is removed. The new `dequeue` function should be

```
struct task* rr_dequeue(struct task* r)
{
    if(r->next == r)
    {
        head = NULL;
    }
    else
    {
        r->prev->next = r->next;
        r->next->prev = r->prev;
        head = r->next;
    }
    return head;
}
```

Finally, the round robin function needs a periodic timer. At each tick of the timer the head position should be moved forward one spot. The function to do this is given below:

```
struct task* rr_periodic_timer(struct task* r)
{
    if(head)
```

```c
        head = head->next;
    return head;
}
```

And to tell the scheduler what functions to call modify the `sops` object as follows:

```c
const struct scheduler_operations sops =
{
    .task_enqueue = rr_enqueue,
    .task_dequeue = rr_dequeue,
    .periodic_timer = rr_periodic_timer,
    .period = 1
};
```

The period is how many timer ticks of the system to wait before calling the `periodic_timer` function. This is sometimes referred to as the quantum of the scheduler.

Compile this by issuing the `make` command. In the `Scheduler` directory copy the `fcfs.cfg` to `rr.cfg` and make the necessary changes. Finally run the simulator and check the output.

Tasks for this lab

For this lab you should understand the `FCFS` and `RR` scheduling algorithm. You then should create some of your own algorithms. You must implement Priority Round Robin (`PRR`), and Shortest Remaining Time Next (`SRTN`). Note that for SRTN you will need to make use of `scheduler_data` field in the task structure. Also, when a task arrives in the system you do not know how long that task will run for. Therefore you must estimate the remaining time and adjust your estimate over time. There are a number of ways to do this, but it is recommended to use a weighted average approach where you guess for the first run time, and then adjust that estimate with each real run time. The first guess should be (2*(4-priority)). Each later guess should be of the form new_guess = 0.2*last_gues + 0.8*last_runtime. You can get the last_runtime by subtracting your own count of runtime from `t->task_info->runtime` . You may choose to implement other scheduling algorithms for extra credit. For more information on different scheduling algorithms look at this wiki: Scheduling Algorithms.

Note on tasks:

Tasks in this lab are similar to processes in a real system. They arrive, run for a while, and then block for some user input or system I/O. Therefore the runtime in the task_info is the total amount of time the task has run, not how much time it has run in this CPU burst. A task may run for 2 units, and then block for user input for 4 units, and then run for an additional 5 units. This task would arrive to task_enqueue twice, and at the end would have a runtime of 7 units.

Extra Credit

For extra credit you can implement your own scheduling algorithm.

Feel free to get as creative as you would like on this and follow the given examples to get started. One option is priority round robin with different quantum for each priority level.

License