Simulator Lab: Scheduling Policies

Introduction

In this assignment, you will be simulating a set of scheduling policies within a discrete event simulation framework. The only files you will be modifying are:

- linked_list.c
- linked_list.h (optional)
- schedulerFCFS.c
- schedulerLCFS.c
- schedulerSJF.c
- schedulerPLCFS.c
- schedulerPSJF.c
- schedulerSRPT.c
- schedulerPS.c
- schedulerFB.c

You should NOT make any changes in any file besides these ten files. Changing linked_list.h is optional if you want to adjust the struct definitions or add additional functions, but you should NOT change any of the existing function definitions. We have marked locations in the code where you should make changes.

In the first part of the assignment, you will implement the linked list and FCFS/LCFS/SJF/PLCFS scheduling policies. In the second part of the assignment, you will implement the PSJF/SRPT/PS/FB scheduling policies. For each scheduling policy, you will implement the following functions:

- Create – creates scheduler info struct, which we already defined in the file
- Destroy – destroys the scheduler struct info
- ScheduleJob – handles a new job arrival to the queue
- CompleteJob – handles completing a job and removing it from the queue

As a discrete event simulator, events (i.e., job arrivals and completions) are simulated as discrete events that occur over simulated time. So the ScheduleJob and CompleteJob functions are called at the job arrival and completion times, and you are passed the current time. Job arrivals are scheduled according to input trace files, which contain arrival timestamps. Job completions are scheduled by your code. So your main goal in this assignment is to **calculate the completion time of the next job to complete**. In both the ScheduleJob and CompleteJob functions, you will likely need to figure out when the next job will complete under that scheduling policy based on the currently queued jobs. (Hint: you probably want to track what jobs are queued).

You use the schedulerScheduleNextCompletion and schedulerCancelNextCompletion functions to manage the completion event. When the completion event occurs, the event is removed and your CompleteJob function will be called. There can only be one completion event at any time, so you are only concerned with scheduling the next completion, not completions far into the future. When you get to that completion, you can then schedule the next completion thereafter. In cases where you want to change the completion (e.g., due to preemption), then you would use the schedulerCancelNextCompletion function to cancel the completion event, which would allow you to reschedule the next completion.

All the code relies upon the linked list, so you should start coding that. The linked list supports a sorted mode based on a user-defined comparison function. It is expected that insertions will be O(n), and that is acceptable for this assignment.

You are welcome to define other functions in those files to help structure the code.

Support routines

The framework for managing the discrete event simulation has already been provided for you, and you won't need to call most of the functions. Here are the most relevant functions you should be using:

- linked list functions that you're implementing
- job.h functions that provide you job info. **You should use the job functions, not the job struct directly**. In the final grading, we may change the job field names to break your code. We only guarantee that the function API will not change.
- schedulerScheduleNextCompletion described in scheduler.h
- schedulerCancelNextCompletion described in scheduler.h
- common library functions such as malloc/free

You likely won't need to use any other functions, so if you start randomly calling other functions in the starter code, you may run into unexpected behavior.

Important details

- In all policies that require breaking ties, you should use the job id as a tie-breaker. For example, in SJF, if two jobs have the same size, then you can use the job id to break the tie, where the smaller job id would be the smaller size.
- You need to be careful with divisions as they may truncate, which may cause some time to not be accounted for properly. Your code should use mod (%) to track the leftover time and properly account for it in the scheduling policy. There are edge cases in correctly addressing this, so you'll need to think through the details.

Evaluation and testing your code

You will receive zero points if:

- You violate the academic integrity policy (sanctions can be greater than just a 0 for the assignment). **All work must be your own without any collaboration of any form.** If you are unsure whether something is allowed, ask the instructor for clarification.
- **You don't show your partial work by periodically adding, committing, and pushing your code to GitHub**
- Your code does not compile/build
- Your code crashes the grading script

Your code will be evaluated primarily for correctness. You are responsible for generating additional test cases to test your code. This involves generating trace files and expected output.

Trace files are located in the traces directory and are named starting with the policy_ and ending with .txt. For example, FCFS_1.txt is a trace for the FCFS policy.

The trace files contain three columns separated by a comma and space. You can refer to the existing trace files for examples. The first column is the job id, and it should be unique and increasing. The second column is the arrival time, and it must be in sorted order. So earlier jobs must be in earlier lines in the trace. The third column is the job size. All the data should be suitable for the uint64_t type, so you cannot use floating point numbers or negative numbers.

The expected output is also located in the traces directory and are named starting with the trace filename followed by ".expected". For example, the expected output for FCFS_1.txt is FCFS_1.txt.expected.

The expected output files contain two columns separated by a comma and space. The first column contains the job id in sorted order. The second column contains the completion time.

To run the trace file, you would run make to compile the program and run:

```
./simulator traceFile outFile scheduler
```

For example, the following runs a trace with the FCFS policy:

```
./simulator traces/FCFS_1.txt traces/FCFS_1.txt.out FCFS
```

You can then compare the results by performing a diff with the expected output:

```
diff traces/FCFS_1.txt.out traces/FCFS_1.txt.expected
```

In addition to testing with trace files, we also have provided a linked list test, which is compiled as the linked_list_test program.

To automatically run all of the tests including all traces in the traces directory (assuming they're appropriately named), then you would run the following command in the project directory:

```
make test
```

make test will run grade.py, which copies your code to a sandbox and compiles/runs all of your code.

Handin

Similar to the last assignment, we will be using GitHub for managing submissions, and **you must show your partial work by periodically adding, committing, and pushing your code to GitHub.** This helps us see your code if you ask any questions on Canvas (please include your GitHub username) and also helps deter academic integrity violations.

Additionally, please input the desired commit number that you would like us to grade in Canvas. You can get the commit number from github.com. In your repository, click on the commits link to the right above your files. Find the commit from the appropriate day/time that you want graded. Click on the clipboard icon to copy the commit number. Note that this is a much longer number than the displayed number. Paste your long commit number and only your commit number in this assignment submission textbox.

Hints

- This assignment exposes you to event based programming, which requires a different mental model. When the events (arrival, completion) occur, your code should just be reacting to that event. You do not need to think about how everything fits together. You just need to address the current event and consider how it might affect the next completion event.
- You should only be performing operations that apply to the current simulated time. If something happens in the future, it will be addressed at that future time.
- In event based programming, there is not a linear flow of function calls. Everything is based on callbacks, where function pointers are called at the appropriate time based on when the event was scheduled (by an earlier event). For example, the program starts by scheduling the first job arrival, and at the first job arrival, the second job arrival is scheduled. The sequence of job arrivals and completions is thus dynamically managed in a simulator queue of events.
- The job has a remaining time field that you can set through the jobSetRemainingTime function. This is NOT automatically set. It is up to your code to track this info if needed. You are not required to use this field, but for some policies, it may be helpful to track.
- If you want a sorted list, you will likely need to create comparison functions. You can just define one in the associated scheduler file and create a linked list with this comparison function.

- Do not directly use struct fields in the starter code. Functions have already been provided to access anything you may need. The functions that we've provided have been intentionally designed to shield you from accidentally messing with the struct elements in unexpected ways.
- Similarly, we've also designed the code so that only the functions listed in the support routines section are needed. This is meant to help you avoid using functions in unintended ways, as connecting all the components requires complex interactions with callback function pointers.