CS2124 Data Structures - Fall 2022
Project 1: multilevelQueueScheduler

Reminder: This project is like an exam. The program you submit should be the work of only you and, optionally, one other partner. You are not allowed to read, copy, or rewrite the solutions written by anyone other than your partner (including solutions from previous terms or from other sources such as the internet). Copying another person's code, writing code for someone else, or allowing another to copy your code are cheating, and can result in a grade of zero for all parties. If you are in doubt whether an activity is permitted collaboration or cheating, ask the instructor.

Every instance of cheating will be reported to UTSA's Student Conduct and Community Standards office (SCCS). At minimum, this will result in a zero for the project/exam and may result in failing the class.

In this project you will be simulating some of the work performed by an operating system (albeit at a super high level). Specifically you will track the simulated processes that currently need to run and then schedule them to run on the CPU according to a given set of rules.

Introduction

In "process.h" you should complete the struct process. Each process has the following data:

- An identifier (i.e., processName)
- The data that belongs to that process (i.e., a processData struct)
- A priority of either FOREGROUND or BACKGROUND
- You can add other data that will help in simulating the schedule.
    - E.g knowing on which step number the process was added to the queue may be useful

In "multilevelQueueScheduler.h" you should complete the struct schedule. Each schedule should have the following data:

- A queue of FOREGROUND processes
- A queue of BACKGROUND processes
- You can add other data that will help in simulating the schedule.
    - E.g. knowing the number of time steps since the start of the simulation may be useful

Code: createSchedule (2 points)

Mallocs and returns a new schedule. Be sure to also create queues for your schedule and initialize any other data in your schedule struct.

Code: isScheduleUnfinished (2 points)

Returns true if given schedule has any processes in either of its queues.

Code: addNewProcessToSchedule (2 points)

You are passed a process identifier and a priority You should create a process containing that data as well as the processData associated with it (call "initializeProcessData" to get this data). Once you have the process created, you should add it to the appropriate queue in the schedule.

Code: runNextProcessInSchedule (8 points)

This function first calls "attemptPromote" to promote the BACKGROUND processes and then calls "runProcess" on the next process in your schedule. See below for a description of the rules which you use to decide which process is next in line to be run. The *numSteps* variable you pass "runProcess" should contain the minimum of the following two numbers:

- the number of steps before the next BACKGROUND process is promoted
- if this is a FOREGROUND process, the number of steps until this process is moved to the back of the queue according to the round-robin schedule

The "driver.c" code repeatedly calls this function until there are no processes left in the schedule. There are important helper methods in "processSimulator.c" so be sure to read through them. Below are the rules for which process to run:

(1) If there exists a FOREGROUND process in the schedule then no BACKGROUND process is run.
(2) Rules for running a FOREGROUND process:
    (a) The queue follows a round-robin schedule scheme. In particular, a FOREGROUND process can run for at most 5 time steps. After it has been run for a total of 5 time steps you should move it to the back of the FOREGROUND queue.
    (b) If this process is complete, remove it from the queue and free the process as well as its process data.
(3) Rules for running BACKGROUND processes:
    (a) The queue follows a FCFS (first-come, first-serve) schedule scheme. Processes are completed in the order they were added to the queue. In particular, the process at the front of the BACKGROUND queue remains there until it either finishes or is promoted.
    (b) If this process is complete, remove it from the queue and free the process as well as its process data.
(4) The simulated process will return a string containing the name of another process or a NULL. Either way you do not need to do anything with the value except return it.

Code: attemptPromote (2 points)

Every process that has remained in the BACKGROUND queue for 50 time steps should be promoted to a FOREGROUND process (i.e, move it to the rear of the FOREGROUND queue). That process now behaves like a FOREGROUND process. Be sure to call the "promoteProcess" function in "processSimulator.c".

Code: freeSchedule (2 points)

Free all of the memory associated with the given schedule

Analysis: Priority Queue (2 points)

Suppose we combined the FOREGROUND and BACKGROUND queues from part one into a single priority queue.

- Would the asymptotic runtime of *runNextProcessInSchedule* be increased, decreased, or say the same?
- Specifically, what is the new asymptotic runtime of *runNextProcessInSchedule*?
- Justify your reasoning for this runtime in one to two sentences. The answer depends on how you setup your priority queue so this justification is important.

Deliverables:

Your solution to the **code** portion of the project should be submitted as "multilevelQueueScheduler.c", "multilevelQueueScheduler.h", and "process.h". The **analysis** portion should be submitted as separate .pdf file.

Upload these file to Blackboard under Project 1. **Do not zip your files**.

To receive full credit, your code must compile and execute. You should use valgrind to ensure that you do not have any memory leaks.

Getting started:

The provided code is deliberately obtuse (especially "processSimulator.c"). You should focus less on how the my functions specifically work and more on what they do and where to call them in your functions (hints to this were given in the function comment headers so be sure to read them).

When in doubt **try running your code!** Don't be afraid to add print statements to help you better understand what data is being stored in a struct or passed by a function.