Assignment 2: Scheduler

Date: April 10th, 2020
Deadline: April 29th, 2020 23:59 CEST

Introduction

You are going to implement the most important parts of a scheduler.

We have supplied a framework which will simulate a running system where processes appear, run and terminate. It uses four queues for the different states of a process; your code will be run by the simulator to handle the processes in these queues.

Objective

You will implement two schedulers.

The first one is a memory scheduler. Processes in the `new_proc` queue are waiting for memory allocation. Your scheduler must assign memory to these processes and place them in the `ready_proc` queue. Memory is limited, so your code can decide the allocation fails. However when this happens, you shouldn't stop the allocation for other processes until the first process in the queue fits. There might be other processes which do still fit. Therefore you should implement an "N_{try}" parameter, which can be set at program start, that is used as the number of (other) processes your scheduler attempts consecutively to satisfy after a failed allocation.

The important word here is "attempts". An attempt may fail or succeed. So, after an initial failure, your code should continue until "N_{try}" consecutive attempts to allocate memory (successful or unsuccesful) have been made, before giving up. Note that it is also possible that multiple processes in the `new_proc` queue can be given memory, so do not stop after the first successful allocation either.

The second scheduler is a CPU scheduler. Processes in the `ready_proc` queue are collected in order for execution on the CPU. Your scheduler may change the order of this queue and grant time slices to the processes. For doing this you can define different scheduling algorithms. You are required to create at least one simple scheduling algorithm (e.g. Round-Robin), one memory efficient algorithm (so that memory will be made available as soon as possible) and one algorithm that is better than the simple one. Be creative! The main evaluation criterium for this last algorithm is how you explain and illustrate in your report why it is better and how you back this up with statistics.

Note that some functions for queue manipulation have already been defined in `schedule.h`. These functions start with the prefix `queue_`. You can use these functions to move processes between queues. For some examples on how to use these, you can look inside `simul2018.c` (however, you do not have to modify that file). You can also write additional queue manipulation functions yourself (for instance, for reordering the ready queue).

Avoid directly manipulating the `.prev` and `.next` pointers in your code. Instead define helper functions such as the ones defined in `schedule.h`. This will make your code more readable and less likely to contain bugs. One exception is using the `.next` pointer for iterating over all of the processes in a queue. In that case it is reasonable to use the `.next` pointer directly since you are not making complicated modifications to the queue.

The `pcb` structure, as defined in `pcb.h`, is used to represent a process in the simulation. However, for your scheduler you are only allowed to use the variables that are exposed through the `student_pcb` struct, except one. You are not allowed to use the `sim_pcb` field in that struct. Documentation on the `student_pcb` structure is provided in `schedule.h`. You are not allowed to use any variables from simul2018.c (e.g. `struct sim_pcb *first,*last`).

Structure of your report

You have to write a short report to accompany this assignment. The structure of the report will be graded (see grading scheme below). Be sure you clearly separate the explanations for your method/implementation, results and interpretation.

Explain the motivations and main guiding ideas behind your algorithms ("why you did it that way"). Tell us what went wrong or what you wanted to implement but didn't (and why you didn't). Show us tables and graphs etc. to compare your different algorithms and tell us what differences you see and why one is better than the other.

To help you with the structure of your report, you can select the following structure:

- Introduction/Introductie
- Method/Methode (here, you explain the idea behind your schedule and their implementation)
- Results/Resultaten (here, you present the results and interpret them)
- Conclusions/Conclusie (here, you conclude the report)

Submission format

Name each of your different schedulers in the same format as the skeleton and edit the makefile to include the compilation of them in the *all* statement.

The report should be short and concise, and must be no longer than 5 pages including all figures and tables. The format of the report should be similar to this file, i.e., font size 12, single spacing. The report must be submitted as a *pdf*.

Getting Started

- Read the entire assignment before starting.
- Study `schedule.h` and try and understand what each of the functions and variables is for.
- Run the program and see what happens when you change the parameters.
- Make some adjustments to `scheduler-skeleton.c` and see what happens.
- Try and implement the memory scheduler with the N_{try} idea.
- Try and implement a simple CPU scheduler.

Grading

- +0.5pt if you have submitted an archive in the right format.
- +0.5pt if your source code builds without errors and you have modified `scheduler-skeleton.c` in any way.
- +1pt if you have implemented the memory scheduler.
- +1pt if you have succesfully implemented N_{try} in the memory allocation.
- +1pt if you have implemented a simple CPU scheduler.
- +1pt if you have implemented a memory efficient CPU scheduler.
- +1pt if your report is properly structured and comprehensively documents your experimental findings.
- +2pt if you have implemented a more efficient scheduler than the simple one (explain and show in your report why it is better).
- +2pt if your report comprehensively discusses your choices and motivations.

- -1pt if either `valgrind` or GCC/Clang's AdressSanitizer reports errors while running your code.
- -1pt if `clang -W -Wall` reports warnings when compiling your code.
- -1pt if any of your source files contains functions of more than 30 lines of code.
- -1pt if your source files are not neatly indented.