CS3104 – Operating Systems

Assignment: P1 – A scheduler in user space

Deadline: 25 October 2022 *Credits*: 50% of coursework mark

MMS is the definitive source for deadline and credit details

You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturer regarding any queries well in advance of the deadline.

Aim / Learning objectives

The purpose of this assignment is three-fold:

- to provide hands-on experience with implementing scheduling algorithms;
- to provide further insight into signals on Unix-like operating systems;
- to gain experience in implementing data structures in C.

Requirements

In this coursework exercise you will implement a simple user space scheduler. To accomplish this, you are **required to use the C programming language** to write a program which can execute an arbitrary list of other programs and control them by sending them signals to ensure that only one of these programs is running at any given time. The choice of scheduling algorithm and data structures for representing processes is left to you.

The features required of your implementation are as follows:

- Your program, once compiled, should be contained in a binary named `sched`.
- It should accept a single command line parameter: path to a configuration file which contains a list of programs in the format specified below.
- It should execute each of the listed programs by using a combination of `fork`() and `exec`()[1].
- All processes should be stopped by the parent immediately after being launched, by sending them the SIGSTOP signal.
- Information about each of the processes should be recorded in a data structure (consisting of *Process Control Blocks*) which contains relevant information (e.g. the path to the executable, parameters and process ID/pid).
- The parent process should then process the data structure containing process control blocks and activate one process at a time by sending it the SIGCONT signal.

The simplest possible approach could simply let each process finish before starting the next one (batch processing). A good solution could implement a variant of round robin scheduling, by letting each process run for a predefined amount of time before stopping it and letting another process have a go. Excellent solutions will incorporate some type of priority scheduling to determine a schedule.

Format of the configuration file

The configuration file loaded by your program should be a simple text file which contains multiple lines, with each line describing a single process to be executed. The format is as follows:

```
<priority> <path> <arguments>
```

Priority is a number between 0 and 20, with 0 representing highest priority and 20 the lowest. Path is the full path to a program to be executed (e.g. "`/usr/bin/ls`"). Arguments are any command-line parameters passed to this program. An example of a possible configuration file is given below, where "`ls`" has higher priority than "`minebitcoin`":

[1] Or any of the variants: `execl`(), `execve`(), etc.

```
3 /usr/bin/ls ~/Documents
6 /usr/bin/firefox
19 ./minebitcoin --value=2000
```

An example file is also provided with the starter code discussed below. Priority scheduling is not required for a basic solution, but your program should be able to read a priority value from the file and store it in an appropriate data structure in all cases.

Source code and getting started

You are provided with starter code which starts an external program and controls its execution by sending SIGSTOP and SIGCONT signals. The starter code can be found here:

http://studres.cs.st-andrews.ac.uk/CS3104/Coursework/P1-Scheduler/starter.zip

Approach

You should start with a minimal solution and then gradually improve it. A high-quality submission based on a round-robin scheduler could warrant a mark of up to 16. You are **strongly encouraged** to make sure that this works first before attempting advanced algorithms like priority queues, multi-level feedback queues or balanced binary trees!

The recommended approach is to follow a sequence of steps. In the first step, you can hard-code a small number of `fork`() and `exec`() calls to an appropriate program (e.g. the `printchars` program supplied with the starter code) and making sure that you can stop and restart them from the parent process by using the `kill`() function to send SIGSTOP and SIGCONT signals.

In the second step, you could develop a data structure to hold information about the processes. A minimal data structure could be a linked list of structs, where each struct contains the ID of the process returned by `fork`(), the path to its executable, priority, and any parameters.

In a third step, you could write the code to read in the contents of the configuration file by using a combination of `getline`() and `strtok`() and populate the data structure you designed. Then you can write a simple FIFO batch scheduler which waits for a process to finish before moving on.

In a fourth step, you could turn this into a round-robin scheduler with a fixed quantum by waking a process, sleeping for the duration of the quantum, and then stopping the process and moving to the next one. You should also make sure that the code is clean, correct, well-tested, well-commented and well-documented in your report.

There are numerous possibilities beyond this, in terms of scheduling algorithms and policies (priority queues, ageing schemes to prevent starvation, prioritisation based on program types) and data structures (multi-level feedback queues, balanced binary trees, hash tables). There are also interesting system calls that could be useful, such as using `ptrace` to intercept system calls, or using specialised system calls to get information about a process's resource utilisation to inform your scheduler.

Hint: you can check if a child process has finished by using the waitpid() function with the WNOHANG flag:

```
int status;
pid_t result = waitpid( process_id, &status, WNOHANG );

if( result == 0)  { /* process is running */ }
else {  /* process has terminated and we can read its status using
waitpid() */ }
```

Submission

You are required to submit an archive file containing a source directory and your report in ***PDF***. The source directory must have a working makefile which must produce an executable called `sched`. The makefile must support the "`clean`" and "`all`" targets and your submission should build and run without problems when extracted on a lab machine. You should also include a directory containing configuration files you used for testing, which should also work on the lab machines.

The report must:

- Explain the design and implementation of your scheduler. Where appropriate, this should include pseudocode (not C code)
- Document the performance of your scheduler by measuring the average waiting time, average turnaround time etc., for a schedule. You can use a diagram to illustrate your explanation.
- Include a discussion of strengths and weaknesses of your approach. This could include references to similar existing implementations (e.g. any OSes or libraries using a similar strategy).
- Explain the structure and motivation behind any data structures you use (e.g. the structure of your process control block and any list/array used to organize them). Where appropriate, you can use diagrams to help explain structures such as trees or linked lists.
- Describe any additional features that you have chosen to implement and how they relate to the scheduling task.
- Describe your testing strategy and any problems faced during development.
- Discuss any limitations in your implementation and describe possible improvements.

Autochecking

This practical will not be autochecked. Sufficient evidence of testing should be provided in the report and submitted code.

Assessment

Marking will follow the guidelines given in the school student handbook (see link in next section). Some specific descriptors for this assignment are given below:

Mark range	**Descriptor**
1 - 6	A submission that does not compile or run, or perform operations related to scheduling.
7 - 10	A partial solution which contains a sensible data structure and is able to launch, pause and resume at least one external program.
11 - 13	An implementation which is mostly correct, but has some significant problems. This could include corruption of the data structure after a period of use, programs running concurrently rather than one at a time, or a scheduler which stops working in some situations. Submissions based on poor code and poor reports can also fall in this category.
14 - 16	A well-written implementation which can load a configuration file and run an arbitrary number of programs with no major bugs. The scheduler should be pre-emptive, i.e. it should stop and restart processes instead of simply waiting for them to finish on their own. A clearly written report which evidences testing, explains all design decisions and demonstrates understanding of the underlying principles.
17 - 18	An excellent implementation which goes beyond the original specification by implementing one or more extensions such intercepting syscalls or more advanced data structures, and is accompanied by an excellent report. A solution in this grade band must implement priority scheduling.
19-20	An exceptional implementation showing independent reading and research, several extensions and excellent quality of code, accompanied by an exceptional, insightful report.

Policies and Guidelines

Marking
See the standard mark descriptors in the School Student Handbook:
http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

Lateness penalty
The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):
http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good academic practice
The University policy on Good Academic Practice applies:
https://info.cs.st-andrews.ac.uk/student-handbook/academic/gap.html