CS2106: Lab 6

Lab 6 is to think about how a scheduler works and shows that it is feasible to build a usable *user mode* scheduler. In Linux, we have at least, the following system calls or APIs which are useful for controlling processes and getting information about processes:

- `waitpid()`: more powerful form of `wait()` to check child process status
- `kill()`: process with pid `p` can be stopped or restarted with the `kill(SIGSTOP)` (stop process) and `kill(SIGCONT)` (continue from stopped state) system calls.[1]
- `usleep()`: more precise form of `sleep()` which suspends the execution of a process using a timing granularity of microseconds.[2]
- `nice()`: set the Unix nice priority of the process.

To use the above Linux APIs, please read the man pages (for `kill`, you do not need to understand about the general usage of signals).

In this lab, we will write a number of user-mode schedulers which assume the following definitions:

- the basic time unit used by the Lab 6 schedulers is called a *user-tick* and the default value is **0.01** seconds. However, that can be adjusted by command line options.
- a process runs for a certain *user-quantum* time measured in *user-ticks*.[3]

To test your schedulers, you may need to run test processes which run for certain amount of time. The test programs will output periodically so that you can distinguish what the processes are doing and whether or not they are running. The user timing code from Lab 5 will be useful for testing Lab 6. In all the exercises below, you will need to create appropriate test programs which be used to exercise and test your scheduler.

By default, the `stderr` stream is not buffered so writes to the `stdio stderr` stream are not delayed by buffering. The example code, `lab6-errlog.c`, shows the use of the `stderr` stream. `lab6-errlog` logs a command to be executed to `stderr` and that executes the command given in the command line arguments. For example, (the redirection is to redirect `stderr` which is file descriptor 2 by default to `errlog.txt`)

```
$ ./lab5-errlog echo hello 2>errlog.txt
hello
$ cat errlog.txt
echo hello
```

[1] For this lab, it is not necessary to understand the general uses of the `kill` system call. A quick explanation is that `SIGSTOP` is one type of "*signal*" in Unix and signals are the Unix OS-level analog of a software interrupt. There are many signals, e.g. `man 7 signal`.

[2] `usleep()` is a library function and not a system call.

[3] As user-ticks is integral, the total process runtime may be less than the rounded up number of user-ticks, e.g. a process whose user CPU runtime is equivalent (recall the Lab 5 Exercise 1 delay loop gives you computation which is a certain number of user time units) to 10.5 user-ticks will have user CPU runtime between 10 to 11 user-ticks.

For simplicity, testing of your code will not involve testing system call errors.

Exercise 1: Running a process more slowly than normal (Graded: 20%)

Goals: Execute long running CPU-bound jobs more slowly than Linux so that very long running jobs, e.g. several days/weeks/months will not slowdown the system for other proceses

Usage: The usage of `lab6ex1` follows the following syntax as per `man man` which we saw in Lab 5.

```
lab6ex1 [-s W] [-t T] ...
```

The arguments in (...) are a process to be executed together with it's arguments. The T option specifies the user-tick in microseconds.[4]

The option W specifies how slowly the process runs:

- $W = 1$: the process runs at close to normal speed
- $W = 2$: the process runs about half as slow
- $W = 4$: the process runs about four times as slow

Write `lab6ex1.c` which works to control the speed of the process as follows. `lab6ex1` controls the execution of process p so that it only runs for around 1 user-tick for every W user-ticks, For example, with $W = 2$: we can have at:

$t = 0 : p$ runs; $t = 1 : p$ is not running; $t = 2 : p$ runs; ...

The specification says "about" because as this is a user mode scheduler, precise timing of when the scheduler is run is not guaranteed. However, if the system is not heavily loaded, we should expect this to work well enough.

Notice that the slowdown with W affects the "real time" and not the "user time". If a process needs a certain user time to terminate, that is will be unchanged. Scheduling with `lab6ex1` only makes it run more slowly.

You can see that `lab6ex1` is a useful (scheduling) tool for running processes which will take a long time, e.g. consider computing some large number of digits of π or running a climate change simulator. We also see the reason why there are certain system calls/APIs which are useful for controlling processes.

`lab6ex1` can be contrasted with controlling the process p with `nice()` alone. The difference is that in Exercise 1 we have a specific definition of the slowdown factor, whereas the nice value of the process affects the scheduler in a way which is not precisely specified. The effect is only known to the internals of the kernel scheduler implementation. This can vary between different Linux kernels.

Hint: to turn a string into an integer, `atoi()` may be useful.

[4] One microsecond is one millionth of a second (us).

Exercise 2: (Graded: 20%)

Goals: This is only a step towards Exercise 3—this exercise implements round robin scheduling of non-terminating processes.

Usage: The usage of `lab6ex2` is as follows

```
lab6ex2 [-t T]
```

where the T option is the same is in Exercise 1, the user-tick time.

Write `lab6ex2.c` which will do round robin scheduling of processes. The processes to be scheduled are specified as command lines in the input (standard input (`stdin`)). For simplicity (this will be relaxed in Exercise 3), the processes **will not terminate**, so they can round robin indefinitely. The processes to be run are be fixed based on the input and do not change over time, which means that you can choose to read in all the command lines first before starting scheduling. An example of using `lab6ex2` is as follows:

```
$ ./lab6ex2 -t 10000
./testprog1 a b
./testprog2 c d
^D
```

This will set the user-tick to be the same as the default. It will schedule the executables `testprog1` and `testprog2` with the arguments as given above with a round robin policy.

The scheduler will control the processes so that at most one process is running at any one time, this is effectively, simulating scheduling on a single core. Processes are to be run in the order they occur in the input. Each process runs for around 1 user-tick before running the next process and so on. As the processes do not terminate, the round robin scheduling can be continued indefinitely.[5]

Exercise 3: (Graded: 60%)

Goals: A realistic round robin scheduler which does "fair share" scheduling

Write a scheduler, `lab6ex3.c`, which extends `lab6ex2` in the following two ways:

- The processes may either terminate or run forever (like in Exercise 2).
- A form of "fair share" scheduling where each process which is running gets a specified percentage of CPU.

The idea of the form of "fair share" scheduling is that each process which is **still running** gets a specified share, i.e. a specified percentage of the CPU. Processes which have terminated donate their share to the remaining running processes. The command line is the same as `lab6ex2`.

[5]Killing the scheduler, e.g. CTRL-C, should also kill the processes managed by the scheduler. This is usually the default as killing the parent propagates to the children unless the behavior has been changed.

The percentage share is given as a number S in input command line before the actual command. For example,

```
$ ./lab6ex2 -t 10000
43 ./testprog1 a b
20 ./testprog2 c d
37 ./testprog3 e
^D
```

For each input line, the shares should sum up to 100, so S can be viewed as a percentage of CPU.[6] To understand the shares, suppose each program in the example needs 100 user-ticks of CPU time. Then after 100 user-ticks, `testprog1` should be $\sim 43\%$ completed, `testprog2` should be $\sim 20\%$ completed, etc. Note that it is still round robin so the scheduler should distribute running time uniformly as in round robin modified by the overall statistical requirements of the shares. The granularity of user-ticks can affect the result. One way to do the above example is to distribute the ticks as evenly as possible given the share requirement.

When a process terminates, the shares are redistributed as follows. Suppose initially all processes are running, then as described above, `testprog1` takes roughly 43% of the CPU time, `testprog2` takes roughly 20% each and `testprog3` takes roughly 37%. After some time, `testprog1` terminates and the other processes are still running. Then the CPU shares become, `testprog2` with 20+22=42% and `testprog3` with 37+21=58%. The share is divided equally up to an integer with any fractional excess being rounded up and given to the running process whose command line is before the command lines of the remaining running processes. The requirements from Exercise 2 still hold for `lab6ex3`, namely:

- at most one process is executing at the time
- the processes are scheduled in the same order as they appear in the input

Note that `lab6ex2`, the assumption was that processes did not terminate. This assumption does not hold for `lab6ex3`.

In addition, when a process terminates, the scheduler prints the current user tick number and command line to `stderr`. This means that the scheduler also keeps its own clock using user tick units, the currrent tick number. Suppose the current tick number is 128 when `testprog1` terminates in the previous example. The following will be output to `stderr` before continuing the remaining processes:

```
128: ./testprog1 a b
```

Note the `128` will start the output line.

Exercise 4: (Not Graded)

This is not graded. You can do at your own leisure. Enhance Exercise 3 so that processes which are sleeping (e.g. waiting for an event to occur such as I/O completion), get some extra user ticks. As this is

[6] The share here is percentage of CPU usage as the scheduler progresses. It is different from the user-tick.

not graded, it is meant for you to play with enhancing your scheduler, so the exact specifications are left to you. We see that `lab6ex4` is similar to a form of I/O boost policy.

Hint: the process state can be found from the special `/proc` filesystem.

Submission

Zip up `lab6ex1.c`, `lab6ex2.c` and `lab6ex3.c` where the individual filenames follow our usual naming convention, e.g.

Surname-*StudentID*-`lab6ex1.c`.

The zip file also follows the usual naming convention, e.g.

Surname-*StudentID*-`lab6.zip`.

In your C files, you should give your usual student information in a comment block at the start, e.g.

```
/*
* Name: Dennis Ritchie
* ID: U123Y
* file: lab6ex1.c
*/
```