CS2106 AY1920S2 Lab 2 Exercise 2,3,4,5,6

In the remaining section of lab 2, we will be implementing different kinds of user-mode schedulers.

Exercise 2: Slow It Down (2 points)

Long-running batch jobs can take several days/weeks/months to complete and can degrade the performance of other processes. In this exercise, we implement a scheduer that slows down CPU-bound processes to give chance to other processes. "Slowing down" refers to the real time, not the user time. Real time and user time (and system time, which is not used here but FYI) are defined below:

- Real time: The wall-clock time. It is all the time elapsed from the start to the end of the process (includes time slices used by other processes and the time spent waiting for IO).
- User time: The amount of CPU time spent in user-mode code (outside the kernel) within the process.
- System time: The amount of CPU time spent in kernel-mode within the process.

You may find the Linux `time` command useful for understanding the above definitions (see `man 7 time`).

Your Task

Write a program `slowdown` to slowdown a single process by a given factor *W*, based on a specified user-tick *T*. `slowdown` should be used as follows:

```
./slowdown [-s W] [-t T] process_with_arguments
```

More detailed explanation of the options:

- *T* specifies the user-tick in **microseconds**. If the option is not supplied, the default value should be 10000.
- *W* specifies how slowly the process runs. If the option is not supplied, the default value should be 1. Here are some example of *W* and its meaning:
  - If *W* = 1, the process runs at normal speed
  - If *W* = 2, the process runs about half as slow
  - If *W* = 4, the process runs about four times as slow

In addition, `slowdown` should control 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: *t* = 0, *P* runs; *t* = 1, *P* is not running; *t* = 2, *P* runs; ...

The specification uses word "about", because `slowdown` is a user mode scheduler and the precise timing of when the scheduler is run depends on the system load. Therefore, you should test your solution when the system is not heavily loaded. You can also assume that the test programs will be single-threaded. Hint: check the `kill` systems call.

Exercise 3: Round Robin Scheduler (1 point)

In this exercise, we will be implementing a round robin scheduler which manages multiple processes.

Your Task

Write a program `rrsched` to perform round robin scheduling of multiple single-threaded processes on a single CPU core. The processes being scheduled should be specified in a config file. `rrsched` should be used as follows:

```
./rrsched [-t T] < path_to_config_file
```

More detailed explanation of the usage:

- *T* specifies the user-tick in **microseconds**. It is the same as the *T* option in Exercise 2.
- *path_to_config_file* specifies the path to your configuration file containing the processes to be scheduled. The first line in the config file denotes the number of cores that can be used by the set of processes (always **1** in this exercise since we are scheduling on a single core). From the second line onwards, each line in the config file is of the format `<prog_name> <program_args...>`. An example config file can be:

  ```
  1
  ./testprog1 a b
  ./testprog2 c d
  ./testprog3 e
  ```

The scheduler should satisfy the following requirements:

1. At most one user process is executing at any one time (we only use a single CPU core.)
2. The process being scheduled may either terminate or run forever.
3. The processes should be scheduled in the same order as they appear in the config file.
4. Once all scheduled processes have terminated, the scheduler should terminate. If some processes do not terminate, the scheduler should run forever. However, killing the scheduler should also kill the processes managed by the scheduler.
5. Each process runs for about one user tick before the next process is scheduled.

You might want to pin all the processes to the same CPU core using the `sched_setaffinity` system call.

Exercise 4: Biased Scheduler (1 point)

In this exercise, we will be implementing a biased scheduler which schedules the processes based on their specified CPU shares.

Your Task

Write a program `biasedsched` to schedule multiple single-threaded processes on a single CPU core, based on their specified CPU shares. `biasedsched` should be used as follows:

```
./rrsched [-u U] < path_to_config_file
```

More detailed explanation of the usage:

- *path_to_config_file* specifies the path to your config file. The format of config file should be the same as that in Exercise 3, except that each line (except the first) is now of the format `<share> <prog_name> <program_args...>`. For all processes in the config file, the shares should sum up to 100, so can be viewed as a percentage of CPU. An example config file can be:

```
1
43 ./testprog1 a b
20 ./testprog2 c d
37 ./testprog3 e
```

- *U* specifies the maximum time unit within which you should maintain the the CPU share bias, in **milliseconds**. If the option is not supplied, the default value should be 100. Suppose *U=100* and the above config file is used. Then 100ms after the start of scheduling, `testprog1` should have run for ~ 43ms, `testprog2` should have run for ~ 20ms, `testprog3` should have run for ~ 37ms. This CPU share percentage should be followed at 100ms, 200ms, 300ms time marks and so on, as long as no processes have terminated. However, within each 100ms duration, you are free to schedule the processes in any way you like.

When a process terminates, its time shares should be distributed among the remaining running processes in proportion to their existing percentages, rounded down an integer, with fractional excess given to the running process that appears earliest in the config file. Suppose initially all processes are running. Then as described above, `testprog1` takes roughly 43% of CPU time, `testprog2` takes roughly 20% and `testprog3` takes roughly 37%. After some time, `testprog1` terminates and the other processes are still running. At this point, the CPU shares should be adjusted to: `testprog2` with (20+15+1=36)%, `testprog3` with (37+27=64)%.

The scheduler should still satisfy the requirements 1-4 in Exercise 3. You can use Linux command `top` for testing.

Exercise 5: Schedule and Top Together (1 point)

In this exercise, we will be improving the scheduler in Exercise 4, so that it acts as a 'top' program for the scheduled processes as well.

Your Task

Write a program `schedtop` which performs scheduling just as the biased scheduler in Exercise 4, and addtionally prints out periodically the information about the **running** processes managed by this scheduler. `schedtop` should be used as follows:

```
./schedtop [-u U] [-i I] < path_to_config_file
```

More detailed explanation of the usage:

- *U* and *path_to_config_file* are the same as those in Exercise 4.

- *I* specifies the time interval for displaying the running processes information, in **milliseconds**. If the option is not supplied, the default value should be 1000.

The following example illustrates the format of the processes information printed. You must follow this format, with the processes displayed in the order in which they appear in the config file:

```
PID    COMMAND      TIME    %CPU
23241  ./testprog1  11231   43
12321  ./testprog2  5237    20
21231  ./testprog3  9712    37
```

If *I* is specified as 1000, your program should print out the above information once per second, with the correct entries and field values. Each line represents a currently running process, with its pid, the command (program name used when executing), its cumulative **user time** in **milliseconds**, and the percentage of CPU time among the current running processes. Values in %CPU column should always be calculated based on the TIME column. Once a process terminated, it should not be printed anymore. If *I* is a multiple of *U*, the printed %CPU values should roughly match the required CPU shares; otherwise, they don't have to match.

To simplify the implementation, we will only test your program with CPU-bound processes containing very limited number of system calls. You should create such programs for testing your solution as well. The alignment of the entries in the printed output is not critical for our purposes, but you should at least make the printed output easy to read.

Exercise 6: Take Them to More Cores (3 stars)

This is an optional exercise. You will get up to 3 stars for a correct solution.

In this exercise, we relax the single CPU core constraint in the previous exercises.

Your Task

Write a program `multisched` which follows all specifications in Exercise 5, except that it schedules the processes on multiple CPU cores. The number of CPU cores available for use is indicated on the first line of the config file. `multisched` should be used as follows (the options are the same as those in Exercise 5):

```
./multisched [-u U] [-i I] < path_to_config_file
```

Let the first line of the config file be an integer *C*, and it will be guaranteed that *1 < C <= 8*. The output from `multisched` should be the same as the output in Exercise 5, except for the %CPU column. In Exercise 5, the %CPU values should sum up to ~ 100. In this exercise, the %CPU values should sum up to ~ (100**C*)%. The CPU shares in the config file will add up exactly to (100**C*)%. Note that you would need to bind the scheduled processes to run on *C* cores.