Lab2_rest: Clarifications and Tips

Exercise 2: Slow It Down

1. The user-tick here is to simulate the system tick in OS. The system tick is a scheduling event which causes the scheduler to run. In our case, the user-tick is when the `slowdown` process decides whether to intervene the execution of its child process (its child process is the program being slown down). You can think of *T* as specifying the time interval between the occurences of two user-ticks.

2. *W* here specifies how much slower the child process *P* should be running. The intended way of slowing down *P* is to send `SIGSTOP/SIGCONT` signals with `kill` at the user-ticks. If *W* is 4, you can make *P* start running after `fork()` (let's say the start is *t*=0, where *t* is time the occurence of a user-tick), send `SIGSTOP` when *t*=1, send `SIGCONT` when *t*=4, send `SIGSTOP` when *t*=5, and so on. In this way, *P* runs for one user-tick duration per four user-tick duration.

3. This question did not specify whether the child process will terminate, and did not specify whether parent process should terminate if the child process terminates. Therefore, you don't have to consider these cases. We will test your `slowdown` program with programs which run for very long, and won't test on the situation when the child process terminates.

4. To manage time, we suggest using either the POSIX timers (for higher accuracy) or `usleep` (for lower accuracy).

5. The skeleton code given uses `getopt`. If you compile the code with `-std=c99`, there will be compilation error. To fix the error, include `-D_POSIX_C_SOURCE=200809` when you compile the code.

6. To test your program, you first need a long-running CPU-bounded program. We have provided one such program below in Appendix. There are a few ways to know that the child process *P* is slown down. You can use `top` to check the user time of *P*, or you can terminate *P* after certain amount of real time, and use `times` syscall in the parent process to check the user time of *P* and compare.

Exercise 3: Round Robin Scheduler

1. In this exercise, each child process should run for one user-tick duration. That means, when a user-tick occurs, your `rrsched` should send `SIGSTOP` to the previously running process, and send `SIGCONT` to the next process (according to the order the processes appear in the config file).

Exercise 4: Biased Scheduler

1. There is a typo in the question handout. In the usage example, `./rrsched` should be `./biasedsched`. The corrected usage example should be:

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

2. You can think of *U* as the time for one round of scheduling. Within one *U*, you should divide the time according to the CPU shares specified, and make `biasedsched` send `SIGSTOP/SIGCONT` to the schedulees.

3. If a schedulee terminates, you need to re-adjust the CPU shares for the remaining schedulees before sending `SIGCONT` to the next schedulee.

4. You can use `top` to check whether your `biasedsched` is working correctly. There is a %CPU column in the output of `top`.

Exercise 5: Schedule and Top Together

1. `schedtop` should be very similar to `biasedsched`. The only difference should be the additional printing.

2. For simplicity, the %CPU column can just be calculated from the TIME column. That means, you can calculate %CPU of process *A* as *(value for A in TIME column)/(sum of all values in TIME column)*, and convert that value into a percentage value in integer. This calculated %CPU is not very accurate at the time instance when some process just terminated, but it is good enough for this lab.

3. Since *I* might not be a multiple of *U*, you should have two threads, one for scheduling, and the other for printing the top information. You will find `pthread_create` and `pthread_join` useful.

Submission to LumiNUS

Zip the following files with the following folder structure as E0123456.zip (**replace E0123456 with your NUSNET id, NOT your student no. A012...B, and use captital 'E' as prefix**):

lab2_rest/
    slowdown.c
    rrsched.c
    biasedsched.c
    schedtop.c
    multisched.c *<include this file only if you completed the optional exercise 6>*

You should zip the folder **lab2_rest** which contains the source code files, and name your zip file as mentioned above. Upload the zip file to the "lab2-submissions" folder on LumiNUS. The deadline for submission is **Saturday, Feb 29, 8:00pm**.

Appendix: CPU-bounded code

You can use the following program as the CPU-bounded processes in testing. If a large integer is supplied as argument, the program will be wasting CPU time for a long duration.

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

double delay(int);

int main(int argc, char *argv[]) {
  double z;
  int D = atoi(argv[1]);
  if (D > 0) z = delay(D);
  exit(z);
}

// Just CPU-bounded instructions which take some time to execute.
// The information computed is not important.
double delay(int loops)
{
  int i, j;
  double z = i, y;
  z = i; // use an uninitialized value
  for (i = 0; i < loops ; i++) {
    // just some random computation to take CPU time
    for (j = 0; j < 1000; j++) {
      y = ((double) loops)/((double) (loops - i + 1));
      z = sqrt((y*y + z)*i/((double) loops));
    }
  }
  return z; // reduce the possibility of loop being optimized away
}
```