Assignment 6 Instructions
CISC 220, Fall 2017

Administrative Notes: All of the administrative notes from previous assignments apply to this assignment as well. I will just summarize the main points here and you can refer back to the older instructions for more details:

- No group work allowed.
- Hand in your file to OnQ in the "Assignment 6" Dropbox area.
- Follow instructions carefully or you may lose points.
- There is a 24-hour "grace period" after the official deadline for each assignment; submissions made within this period lose 20% but are still accepted.

Deadline: The final deadline for this assignment is **Tuesday, December 5 at 11:55 p.m.**

Purpose: This assignment is intended to be a *short* exercise in creating a program using multiple processes.

High-Level Description: You must write a C program called `race.c`. It must take two parameters, which must be the names of executable programs (C programs, scripts, etc – anything that can be executed without needing parameters). Your program must execute both programs in parallel until one of them finishes and wins the race. Your program must report which program won.

With the tools we have discussed in class, it wouldn't be easy to do this in some cases – especially for programs that take less than a second to run, or pairs of programs that take almost exactly the same amount of time. We'd have to worry the "fairness" of the Linux scheduling algorithms and which program was started first. But I'm not asking you to do any of that.

Suppose someone calls `race program1 program2`. Here's what you have to do:

- Fork off a child process to execute program1. Use `execvp` or one of the related functions for running a program. These are documented in the summary sheet for topic 9 ("Processes and Signals").
- Fork off another child process to execute program2.
- The parent process should *wait* until it is told a child has finished. From the result and the parameter for wait, you'll know which child finished and whether it finished successfully.
- Kill the other child immediately.
- If the first child finished successfully, report it as the winner. If not, report that the second child is the winner by default.
- And yes, this means that if both children were going to fail the one that fails first wins. Maybe not completely fair, but it keeps things simpler.

Sample Programs: I have put several files in /cas/course/cisc220/assn6 for your use:

- `testProgram1.c` finishes successfully after approximately two seconds
- `testProgram2.c` finishes successfully after approximately three seconds
- `badProgram1.c` fails after approximately one second
- `badProgram2.c` fails after approximately 60 seconds
- `quickSort.c`: sorts several arrays using the quick sort algorithm *(takes about 1 second on CASLab when running by itself)*
- `mergeSort.c`: sorts several arrays using the merge sort algorithm *(takes about 2 seconds on CASLab when running by itself)*
- `selSort.c`: sorts one array using the selection sort algorithm *(takes about 6 seconds on CASLab when running by itself)*
- `makefile`

(Just a side note: I had to make the quick sort and merge sort programs sort a LOT of arrays just to get the into the time range to compare with just one selection sort. Empirical evidence to confirm what you have probably been taught about these algorithms – selection sort is simpler to write but is a LOT slower!)

Copy these programs to your working directory and write a `race.c`. The `makefile` will build your `race` program as well as all of the test programs.

Feel free to write your own test programs as well as using the ones I have provided. We will use different test programs while marking, just to make sure your code isn't tailored specifically to our examples.

If you try to write test programs that don't use the "sleep" function and just run for a very long time, bear in mind that one second is a very, very long time for a computer. Just counting to one million isn't going to be enough time to make a difference.

Test Run #1:

```
-------$ race testProgram1 testProgram2
testProgram2 is starting.
testProgram1 is starting.
testProgram1 is finished!
testProgram1 is the winner!
-------$ ps
  PID TTY          TIME CMD
 3834 pts/0    00:00:01 emacs
 7917 pts/0    00:00:00 ps
31552 pts/0    00:00:00 bash
```

This output shows that `testProgram1` finished before `testProgram2`, which is what is expected. The "ps" afterwards shows that `testProgram2` was killed and was not still executing after the race.

Test Run #2:

```
-------$ race testProgram1 badProgram1
testProgram1 is starting.
badProgram1 is starting.
badProgram1 has failed!
badProgram1 failed, so testProgram1 wins by default
-------$ ps
  PID TTY          TIME CMD
 3834 pts/0    00:00:01 emacs
 8581 pts/0    00:00:00 ps
31552 pts/0    00:00:00 bash
```

This output shows that `badProgram1` failed before `testProgram1` could finish, so `testProgram1` was the winner by default and was killed immediately to save cycles.

Test Run #3:

```
-------$ race testProgram1 badProgram2
badProgram2 is starting.
testProgram1 is starting.
testProgram1 is finished!
testProgram1 is the winner!
-------$ ps
  PID TTY          TIME CMD
 3834 pts/0    00:00:01 emacs
 9320 pts/0    00:00:00 ps
31552 pts/0    00:00:00 bash
```

This output shows that `testProgram1` finished successfully before `badProgram2` finished and immediately killed off `badProgram2`, so we never never found out that `badProgram2` would have failed.

Marking Scheme:

- forking off two child processes to execute the programs named in the arguments: 2
- waiting to find out which child process stopped first: 2
- stopping the other process: 2
- if both processes finished successfully, output telling which one was the winner: 2
- if one of the processes finished with an error, output saying that the other process won by default: 2
- total: 10

A program that can't be compiled without errors will get less than 5 points.

Advice: If you run out of time and one of your functions won't compile, replace it with the original "stub" version so that it will be possible for the grader to test your module and give you credit for the functions that *do* compile and work correctly.