Date due: Thursday, April 14, 11:59:59 p.m.

1 Introduction and purpose

In this project you will write a very small program using simple process control. The purpose is to get some firsthand experience with process control basics.

As was the case for Project #5 earlier this is a small project that has only public tests and will not be graded for style. So you have a short time to do it (to leave more time for the more difficult projects). And the time to do it was reduced even more due to the extension on Project #6. However, the project really is very short, and if you understand the lecture examples of process control it can be written in an hour– or less.

Suppose you need to write a paper, which **must** be 200 words or more, or you will get a terrible grade in one of your GENED courses. Wanting to be as efficient as possible and not waste time (because you would much rather work on your CMSC projects than write papers), you want to write the minimum number of words needed, so you have decided to write a C program that counts the number of words in a paper. A program could do this by reading lines until the end of the input, breaking them up into words using the string library functions, and counting the words. However you don't need to reinvent the wheel by implementing your own word count program– an easier way is to use process control, because there is already a standard UNIX utility named `wc` (for "word count") that counts the number of words, characters, and/or lines in its input, depending on a command–line argument. The option `-w` causes `wc` to give the number of words in its standard input. (A word is just any sequence of consecutive non–whitespace characters.) You just need to invoke `wc` properly.

This is the kind of situation where you would realistically want to use process control– you want to write a program that will run one or more other programs (namely `wc` in this case), but the output from that other program should not be printed on the screen. Instead your program must capture the output of the other program and read and process it. The simple metric that your program should use is as follows: if `wc`'s output is a number that is **less than** 200, your program must print the **exact** message `Too short!` and quit with an exit status of 1. If `wc`'s output is a number that is 200 **or more**, your program must print `Long enough!` and quit with an exit status of 0 (indicating successful termination).

Note that the processing that your program has to do of `wc`'s output is very simple– it just reads the output of `wc` (which will be a single number), prints something based upon its value, and also exits with one status or another depending on its value. But it will probably be your first experience using process control.

Before writing any project, but this one in particular, study any relevant lecture examples carefully, and make 100% sure you understand everything that they are doing. If you have questions about any of the lecture process control examples ask about them in the TAs' office hours **before** starting to code.

2 What your program must do

In this project, as in Project #2, you will not be writing functions that we will call from our main programs. Instead you will write a complete, standalone, very small program, named `papersize.c` (its name must be spelled and capitalized **exactly** that way for it to compile on the submit server). Here's what your program needs to do:

- It should first create a pipe.
- It should then create a child process. This is usually done using the `fork()` system call, however, **do not call `fork()` directly**. We have provided an object file `safe-fork.o`, which contains a function `safe_fork()`, which you should call **instead**. It prevents the possibility of having a serious fork bomb. Your `papersize.c` should include `safe-fork.h`, so the prototype of `safe_fork()` there will be visible. `safe_fork()` has the same return value as `fork()` and has no parameter, as you can see by looking at `safe-fork.h`. **Use `safe_fork()`, instead of directly calling `fork()`.**

  We set things up so your program **will not even compile** on the submit server if it calls `fork()` directly– but you could still cause a fork bomb on the Grace machines if you call `fork()` and do it incorrectly, which could cause dozens of other students to lose whatever work they were working on. **Do not** call `fork()` directly.
- Keep in mind that a child process will inherit its standard input and output from its parent process, as well as file descriptors for any open files or pipes. What the parent process needs to do is to reroute its own standard input to

come from the input end of the pipe, and the child process should reroute its standard output to go to the output end of the pipe. (We discussed how to do this in lecture the day this project is being assigned, and some of today's lecture examples show it.)

In this project it's not mandatory for either the parent or child to close unneeded ends of the pipe, because the program will not be trying to read until the end of the input, but closing anything as soon as it is no longer needed saves system resources, so it is always a good practice.

- The child process should `exec()`, or replace itself, with the `wc` program (its full pathname is `/usr/bin/wc`), using the single argument `-w`. The `wc` program will run and count the number of words in the standard input, and the result that it prints will end up going to its standard output, which has previously been redirected to go to the pipe. You can use any of the `exec` family of functions you find convenient.
- The parent process must read a single integer from its standard input, which has now been redirected to come from the pipe. Just use `scanf()` to read it.

  Your program doesn't have to use `wait()`, because `scanf()` does the wait– it blocks until input is available to be read.
- If the integer read by the parent is 200 or more it should print `Long enough!` and quit with an exit status of 0, otherwise it should print `Too short!` and quit with an exit status of 1. (Both messages must be spelled **exactly** as shown and end with a newline for the submit server to recognize your results as correct.) Just use standard C I/O to print the messages, as well as to read and write from the pipe. (You **only** need to use C standard I/O in this project, **you do not need to use low-level UNIX I/O.**)
- System calls don't usually fail, but anything can happen. If any system call fails your program should print some sort of descriptive message saying what didn't work right and where and should just quit after that. The exact wording of the message doesn't matter– the public tests don't test this situation– but you should print something anyway, so you'll know about it in case your code has a bug and ends up calling some system call incorrectly. It also doesn't matter how you cause the program to quit in this case; any way that achieves this is fine.

A Development procedure review

A.1 Obtaining the project files and compiling

Log into the Grace machines and use commands similar to those from before:

```
cd ~/216
tar -zxvf ~/216public/project07/project07.tgz
```

This will create a directory `project07` containing the necessary files for the project, including the public tests, `safe-fork.o`, and `safe-fork.h`. `cd` to the `project07` directory, create a file named `papersize.c` (spelled **exactly** that way) that will include the header file `safe-fork.h`, and write the program as described above.

Since your program will be a self–contained standalone program, you don't need to write a makefile for this project (although you are welcome to if you like; if so it will just be ignored on the submit server). You can compile your program using the (exact) command:

```
gcc papersize.c safe-fork.o -o papersize.x
```

A.2 Checking your results and submitting

The project tarfile contains two sample papers named `document1` and `document2`. If you want to see how long they are (how many words), just use a command like "`wc -w < document1`".

When you finish writing the program, test it by running it with input from one of the two data files `document1` or `document2`, for example:

```
papersize.x < document1
```

Immediately after running the program (as the **very next command**) you can see its exit code in the shell using `echo $status`. (This shows the exit status of the **most recent command**, so if you run another command between the program and `echo $status`, it will show the other program's exit status.)

There are four public tests of this project, and no secret tests. Note that the public tests are actually just small shell scripts, named `public1`, `public2`, `public3`, and `public4`. This is because you are not writing a function that we could call from our main program, but instead a complete program. (There are other ways that things could be set up but this is the easiest.) Note you have to use the name `papersize.x` for the executable program when compiling, or the public test scripts won't work.

You can tell by looking at the public test shell scripts that the first two public tests check the output of your `papersize.c` program on a too–short paper and a long enough paper, respectively, while the third and fourth check whether it's exiting with the proper status on the same two papers. To run the tests after compiling your program to form the executable `papersize.x`, just run the script (for example `public1`)– if its only output is "`Success!`", your program passed the test. If not, run `papersize.x` with input redirected from one of the data files and check its output as in projects:

```
papersize.x < document1 | diff - public1.results
```

(Note that the output files for this project have been named `public1.results` and `public2.results`.)

However, if the third or fourth test is failing, do **not** run the program using the `diff` command– instead just run it using input redirection, and check its exit status as described above.

Because the tests are all shell scripts the `run-tests` script that your TA showed recently in discussion, that will run a project on all its tests, won't work for this project. But there is a second version for projects like this whose tests are shell scripts that is named `run-tests2`, which is just a slightly modified version of `run-tests`. Just run `run-tests2` to automate the process of running your program on all of the tests, once you have compiled your program.

Running `submit` from the project directory will submit your project, but **before** you submit you **must** make sure you have passed the public tests, by compiling your program and running them yourself as described just above.

A.3 Grading criteria

Your grade for this project will be based on:

public tests	100 points

B Project–specific requirements, suggestions, and other notes

- Of course the program could be written without explicitly using process control by just using the C library function `system()`, or by writing your own code to count words in a file instead of running `wc`. But the entire point of the project is to get some experience using some process control system calls, so you **must** write it as described. If you don't use process control to write the project, or if you use `system()`, you will **not get any credit on the project. Your program will not even compile on the submit server if it calls `system()`**.
- Your program can **not call `sleep()` anywhere**. Some lecture examples of process control used `sleep()` in order to cause the results of small example programs to be more random, so that you could see process control working. However, you do not need to make this happen artificially in this project, and there is no reason to have any process sleep in this project. Your program will **not even compile** on the submit server if it calls `sleep()` anywhere.

  Also do **not** use a loop to try to make a process wait for another one to do something. This is called busy–waiting.
- If we really wanted to do what this project does, instead of writing a C program to run another program and read its output, we might instead write a shell script to run it and capture its results (which would require some shell script concepts that were not covered in discussion section, which are not important for out purposes here). UNIX process control is extensively used, but this program is a little simple for it to be the best realistic approach. But the project is just for the purpose of getting beginning experience with basic UNIX process control.
- If your program isn't working right, carefully compare the arguments you are passing to an `exec()` system call to the arguments being passed to it in any of the lecture examples that use `exec()`. If what you're doing is different from the lecture examples, that could be a problem.
- You **cannot** modify anything in the header file `safe-fork.h` or add anything to it, because your submission will be compiled on the submit server using our version of this file.

Your code **may not** comprise any source (.c) files other than `papersize.c`, so all your code **must** be in that file. You cannot write any new header files of your own either.

- You cannot debug a shell script using `gdb`. If you want to debug, run `papersize.x` under `gdb` (compiling of course with `-g`), then look at the shell script of the test that you want to run the debugger on, to see what input file it is using for input redirection when running `papersize.x`. Then run your `papersize.x` in `gdb` redirecting input from the same file. (You can run a program under `gdb` with input redirection by just putting the `<` and input file name after the `gdb run` command). For example, suppose you want `papersize.x` to read input from the file `document1`, just like `public1` does. Just run `gdb papersize.x` and then use the `gdb` command `run < document1`. (Perhaps you may want to set some breakpoints first.)
- If you are debugging a program that uses process control to create child processes, note that `gdb` will by default continue to trace the execution of the parent process after a fork. However, if you use the `gdb` command `set follow-fork-mode child` before the program creates a child, `gdb` will trace the child instead. (If needed, `set follow-fork-mode parent` will switch back to tracing the parent when subsequent child processes are created.)

  If your `papersize.x` program ends up running child processes with the wrong command–line arguments nothing is going to work right, and it might be difficult to know why. So if things aren't working right and you're not sure why, set a breakpoint in `gdb` right before a child process is created, use `follow-fork-mode child`, set a breakpoint in the child process right before it runs another program, and use `gdb` to view the arguments at that point that the other program is about to be run with. If they're not right, that is the problem.
- For this project you will **lose one point** from your final project score for every submission that you make in excess of four submissions, for any reason.

C Academic integrity

Please **carefully read** the academic honesty section of the syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, publicly providing others access to your project code online, or unauthorized use of computer accounts, **will be submitted** to the Office of Student Conduct, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to projects. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. More information is in the course syllabus– please review it now.

The academic integrity requirements also apply to any test data for projects, which must be **your own original work**. Exchanging test data or working together to write test cases is also prohibited.