Project 5: Unix Process Control, Pipes, and Message Queues

Due: Monday February 17, 2014 23:59:00 Pacific USA time zone.

Points on this assignment: 305 points with 30 bonus points available.

Work submitted late will be penalized as described in the course syllabus. You must submit your work twice for this and all other homework assignments in this class. Ecampus wants to archive your work through Blackboard and EECS needs you to submit through TEACH to be graded. If you do not submit your assignment through TEACH, it cannot be graded (and you will be disappointed with your grade). Make sure you submit your work through TEACH. Submit your work for this assignment as a *single tar.bzip file* through TEACH. The same single tar.bzip file should also be submitted through Blackboard.

Place all of the files you produce for this assignment in a single directory called `Homework5`.

In this assignment you will explore Unix process creation and control (lots of it), the use of Unix pipes for inter-process communication and the use of Unix System V message queues for inter-process communication. The Unix pipes piece (Problem 2) of this assignment is modestly complex, maybe 200-250 lines of C code. The message queue portion (Problem 3) implementation will have about 750 lines of C code. You have 2 weeks to complete this assignment. I urge you to not delay beginning it. It will be the largest most complex homework assignment in the class. It will push your time management skills.

> For the pipes piece of the assignment (problem 2), you are going to want to become very familiar with chapters 24-27 and 44 from TLPI. Section 44.4 is especially good to (re)read. If the text has source code (and I know it does), read that too. Knowing how to read code makes you better at writing code. You can download all the source code from the book from the author's web site.
>
> For the message queue portion of the assignment (problem 3), you are going to want to become very comfy with chapter 46 from TLPI. Sections 46.1 and 46.2 are especially good to (re)read. If the text has source code (and we both know it does), read that too. Knowing how to read code makes you better at writing code.

In **all** your source files (including the Makefile), you need to have 4 things at the top of every file as comments:

1. Your name

2. Your email address (ONID or engineering)
3. The class name and section (this is CS311-400)
4. The assignment number (this is homework #5).

IMPORTANT: You are gong to be making a lot of calls to the `fork()` command for this homework. Use of the `fork()` call can be a bit tricky at first, especially when you are exec-ing other commands following the `fork()` call. One thing you don't want to do is leave a bunch of unclaimed processes running on the system. To help you identify them, run this command:

```
ps aux | grep $LOGNAME | grep -v root
```

This command will show you all of the processes that you have running on the system. If you see a bunch of dangling processes out that are yours, kill them, using the pid (second column) from the above command. It is easy to wind up with a bunch of lost processes, particularly with Problem 2, in this assignment. I suggest you do this after each time you run your program for Problem 2. Once you get it debugged, it won't be necessary, but it is still a good idea. You should **be certain to do it before you logout**.

Remember that the programming work in this class is intended to be individual work, not group work.

1. **5 points**. When you are ready to submit your files for this assignment, make sure you submit a *single bzip file*. Review homework #1, problem #1 if you need a refresher on how to do this. If your file is not a single bzip file, you cannot receive points on this homework assignment.

2. **100 points.** Write a C program (that runs on `eos-class`) called `pipeline` that reads text from `stdin` and generates output from a 5-stage filter of processes connected by pipes. The input is from `stdin`, the output is to `stdout` **and a file named on the command line**. I have provided a sample bash shell script (`pipeline.bash`) and 3 sample data files that will allow you to test the output of your program. They can be found in:
   `/usr/local/classes/eecs/winter2013/cs311-400/src/Homework5/Problem2.`

   The effect of running your program on the input data will be the much the same as this command line:

   ```
   (rev | sort | uniq -c | tee outfile2| wc) < simple1.txt > outfile1
   ```

   where `simple1.txt` is the name of one of the sample input files and `outfile1` is the name of the output file you can use to capture the result of your program. **Also notice** the `tee` program ***lurking*** in the middle of the pipeline. You'll want to read the man pages for `tee` to see what it does. If you copy and paste the above command into a shell and get back the ugly message "uniq: -c: No such file or directory" it means that the "-c" characters have been munged by MS Word and the dash is not

really a dash (hyphen, em-dash, en-dash, minus symbol, negation symbol, whatever); you'll need to actually type it.

Your main program will create pipes and will then `fork()` new child processes, plumbing the pipes correctly so the processes connect through `stdin` and `stdout`. The child processes will `exec` the appropriate command for their position in the chain of pipe-connected processes.

Your application has a single *required* command line parameter: `-f outfile`. The `-f outfile` parameter names the file to which the `tee` sub-process will write data. If you want to have additional command line options (such as –v), that's fine with me.

For grading, I will run your program with the following command:

```
./pipeline -f outfile2 < /usr/local/classes/eecs/winter2013/cs311-400/src/Homework5/Problem2/words.txt > outfile1
```

The challenging part of this piece of homework will not be the fork and exec (not that those are trivial), it will be using `dup()/dup2()` to get all the processes connected through `stdin` and `stdout`. You'll also need to take care that you close all the unnecessary sides of the pipes. If you don't close all the unnecessary pipes, you won't reach the correct termination at end of the input. If you find that your program has processed all the input, but does not terminate, you've forgotten to close one/some of those other pipes. Read TLPI pages 894-897 a couple of times if you get stuck on this, including the sample source code. Having your program *almost* working, but not terminating on end of input because you've not closed all the extra pipes can lead to a flattening of the forehead. Here are a couple web pages that may be helpful when you have those ends of the pipes open:

http://www.ibm.com/developerworks/aix/library/au-lsof.html
http://www.cyberciti.biz/faq/linux-find-all-file-descriptors-used-by-a-process/

My suggestion on how to start (because this is the way I wrote it) it to just have straight code that simply does the *fork* and *exec* of the `cat` command and then connect all those pipes (`dup()/dup2()`). Using the `cat` command to get started makes it easy to read the output and `cat` reads from `stdin` and writes to `stdout` just like the other commands do. Start by just having a single `cat` command, then two `cat` commands, then three, all the way until you have a straight boring pipeline of five `cat` commands. After you've got three `cat` commands linked this way, you'll see a nice reusable pattern for the remaining ones. Once you have all those hooked together through `stdin/stdout` **and** have them terminate on end of input, start making the switch to replacing the `cat` commands with the other commands (`rev, sort, uniq, tee,` and `wc`). If it helps you to make a copy of the `pipeline.bash` script and change it as you develop your C code, that's fine too.

Like in other assignments, don't just start hacking away on C code. Plan the work. Work the plan. Break the assignment into small pieces, write them, test them, and move on.

A word of advice: You'll be collecting the output from your program into a file, from `stdout`. So if you want to have diagnostic messages in your code (aka debug statements), it might be better to have your diagnostic messages go to `stderr`, not `stdout`. Sending them to `stderr` allows you to separate your diagnostic messages from your output.

You must have a `Makefile` for this assignment. If you don't have a `Makefile`, I cannot compile your code. You should use the following compiler flags for `gcc` as part of your `CFLAGS` in your `Makefile`:

```
-Wall
-Wshadow
-Wunreachable-code
-Wredundant-decls
-Wmissing-declarations
-Wold-style-definition
-Wmissing-prototypes
-Wdeclaration-after-statement
```

You will probably generate a LOT of messages from the compiler using those flags. Go into your code and clean them up. You may want to read about the C `extern` keyword. And, `extern` isn't just for variables.

Task	Number of points
Just for your code compiling.	5
Using "`-Wall -Wshadow -Wunreachable-code -Wredundant-decls -Wmissing-declarations -Wold-style-definition -Wmissing-prototypes -Wdeclaration-after-statement`" in the `Makefile` without warnings from the compiler.	10
Having a target in your `Makefile` called `test` that will run your code and compare the output to the output from the `pipeline.bash` script, using `words.txt` as input. Homework #4 had a pretty extensive explanation on how to accomplish this.	20
Terminating on the end of input	25
Producing the correct output (as the `pipeline.bash` script produces)	40
Total	**100 points**

Optional Extra Credit (worth up to 10 points): Write the same program, but use Python instead of C for the development language.

3. **200 points total.** Write a C program (that runs on `eos-class`) called `uniqify`. The duty of `uniqify` is to read a text file (as `stdin`) and output the unique words in the file to `stdout`, sorted in alphabetical order with the count of the occurrence of each word. The words will be sorted and counted by a group of processes. The input is from `stdin`, and the output is to `stdout`. I have provided a sample bash shell script (`uniqify.bash`) and five sample data files that will allow you to test the output of your program. The sample data files and the sample script can be found in: `/usr/local/classes/eecs/winter2013/cs311-400/src/Homework5/Problem3`

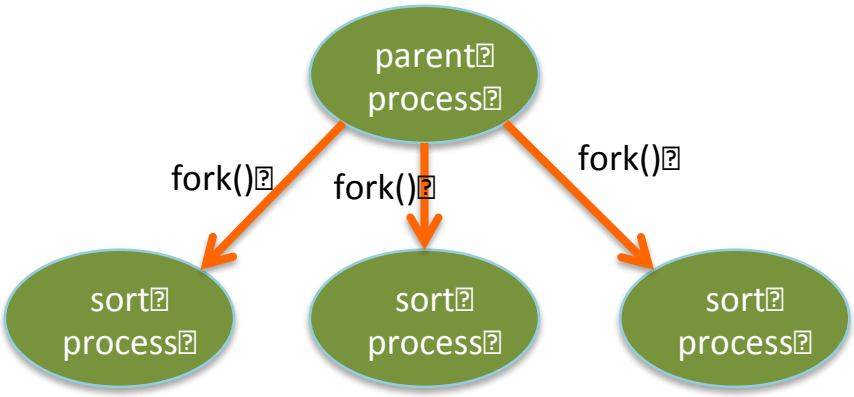
   Logically, your program should be organized into 3 stages:

   1. A parser stage that reads the input from `stdin` parsing the lines into words and sending the words to the sort sub-processes using System V message queues,
   2. A **group** of sort sub-processes process does the sorting and counting, and
   3. A combiner stage that suppresses duplicate words, sums occurrences, and writes the output to `stdout` (it is possible to reuse the process from stage 1 for this stage as well, I did).

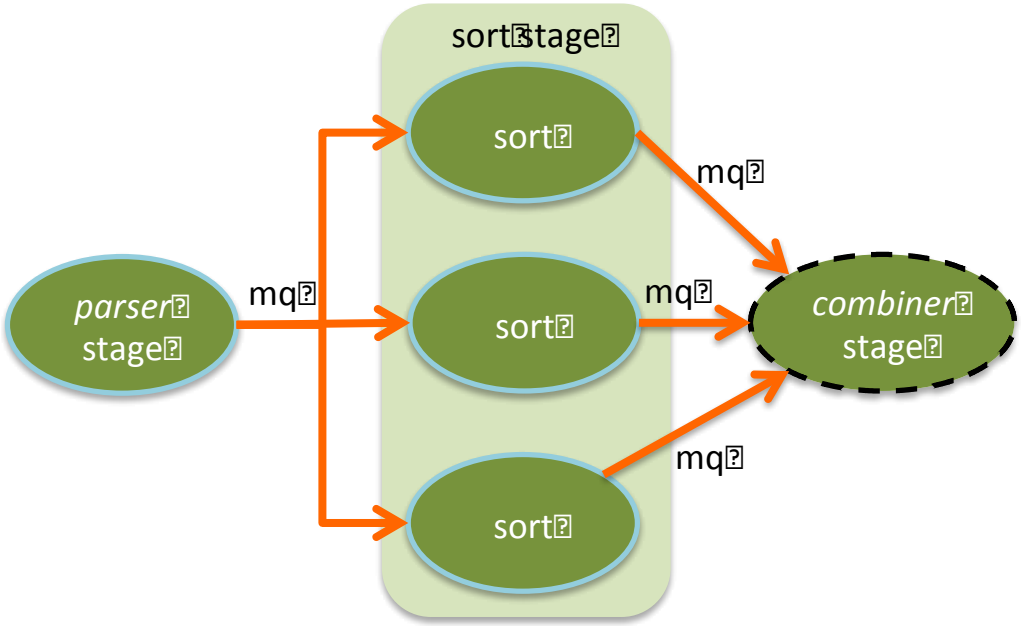
   In this assignment words are all alphabetic and case insensitive, the parser stage should convert all alphabetic characters to lower case. All non-alphabetic characters delimit words and are discarded by the parser. This means than an input of "Jesse's 3 multi-processes" is actually 4 words (for this assignment): Jesse, s, multi, process.

   You should organize the processes so that every parent process waits for its child processes to terminate (no zombies or orphans). Using a signal handler for `SIGCHLD` is a good idea, as is a `SIGINT` signal handler. Remember to clean up those dangling message queues, even when your process receives a Control-C (which is why you want that handler for `SIGINT`). Leaving your message queues out there with unknown messages still in them will lead to expressions of displeasure (some of them very loud). The `ipcs` and `ipcrm` commands will be your new best friends (you can unfriend them after this class). You could probably put something like this in your `Makefile`, to clean up the queues.

   If you were look at the parent child relationships for processes in my code, they would look like the diagram below (with 3 sort sub-processes).



A diagram showing the stages in my implementation and also depicting the System V message queues looks like the figure below. This is how the sort would look with 3 sort sub-processes. You must accept a command line option (-n #) to determine the number of sort sub-processes. I managed reuse the parent/parser process as the combiner process. If you prefer to use a separate message queue for the parser to send words to each of the sort sub-processes, that is fine with me. If you prefer to use a separate process as the combiner (and not reuse the parent/parser process), that's fine too.



For the sorting processes, you can use the `tsearch()/tfind()/twalk()` function calls. I'm not sure how much easier those calls actually make the development, but it did mean that I did not have to write the sort routines myself (again). Despite having been around for a long time, the `tsearch()/tfind()/twalk()` don't seem especially well documented. There are not a lot of good example code chunks out there on the web for them either. Some of the web-pages about `tsearch()` examples that are available are in

languages I don't speak or read. If you do use the `tsearch()` functions, I can pretty much guarantee that you'll find yourself mumbling something not acceptable in polite company and realizing that you need to have an extra * on one of your variables. Unfortunately, I cannot find again the web page that suggested that to me (it saved my scalp). If you'd rather use your own over/under/sideways-balanced B*++heap Fibonacci double hash uber quickest ninja sort routines, that's fine. Just don't let having the coolest sorting routines keep you from being on time with the assignment. If someone asks on Piazza, I can post some of the web pages I found for `tsearch()` and friends. I can also tell you that if you use your own code for the sorting, make sure you test with large input.

15 points (of the 200): Provide timings based on the size of the input file and the number of sort sub-processes. Vary the number of sort processes in this fashion: 1, 2, 3, 4, 5, 10, 15, and 20. Ensure that you test it with multiple types and sizes of files, including word lists and free form prose (the 5 sample files should just about cover it). Having a little shell script to generate that data would be really handy. These timings should be plotted in a fashion that makes sense. Use MS Excel or LibreOffice Calc to produce a spreadsheet of your timing results and create a graph showing time on the vertical axis and input size as the horizontal axis. You'll have a line in the graph for each input file. If you are building and running your code on your personal system/VM, you're are going to want to do some runtime data collection on `eos-class`, since it probably has more CPUs that your system. The `eos-class` system has 16 CPUs (that are hyper-threaded) and quite a bit of memory, though it also has a number of users. Try out the `lscpu` command.

15 points (of the 200): Create a slide using wither MS PowerPoint or LibraOffice Draw that shows the parent/child topology for <u>your</u> application and a slide that shows the input/output data flow topology for your program. Make sure you clearly label each process and edge. If you use more than one process for a stage in the pipeline, note it in your slide. Be creative; try something other than just copying mine.

This is a complex assignment. Break it into pieces. Plan the work. Work the plan. My recommendation is to:

- First solve processing the input from stdin:
  - Removing non-alpha characters
  - Down-casing all the letters, and
  - Splitting a line of words into multiple words.
- After that, you can work on the fork of the sort sub-processes and get all the System V message queues in order. Make sure you can accept a command line argument for the number of sort sub-processes.
- Once you get all the sort sub-processes message queues plumbed, start working on recombining the results. Merging the output from the sort sub-process to eliminate duplicates and NOT storing all the data and/or resorting data will take some analysis. You do not want to be doing this at 11:38 pm the night the assignment is due.

There are of course other ways to break down the work, but that is how I did it and probably how I'd do it again. If I knew I was dealing only with very large data files, I'd choose something a little different, but not much.

I suspect that even though this piece of the assignment is about processes and System V message queues, you'll spend more time on the combiner stage rather than the other stages. Make sure you **test it** with multiple size input files and several different numbers of sort sub-processes.

Command line options for your program must include "–n #". This is how you will vary the number of sort sub-processes. If –n # is omitted from the command line, it should default to a single sort sub-process. Don't allow the number of sort sub-processes to go above 50. I won't test higher than 20 sub-processes. Don't leave any zombie/orphan/unwnated processes out there. Run the

```
ps aug | grep $LOGNAME | grep -v root
```

command to make sure you don't have any. If you find them, kill them. Clean up and remove your System V message queues when done too.

One of the joys of programming in C is that you get to write a few low level functions on your own. For example, I think it is justified to say that the string handling functions in C are somewhat Spartan. Depending on how you want to go, you don't actually *have* to remove/replace non-alpha characters, just use them as delimiters. There are some C functions that can make **tokenizing** a string a lot easier (Google can be your friend). Did I use the word tokenize? I think I did use the word tokenize. Did you see the word tokenize?

You must have a `Makefile` for this assignment. If you don't have a `Makefile`, I cannot compile your code. If I cannot compile your code, it is not happy code.

20 points (of the 200): In your `Makefile`, put a target called `test` that will compare the output of your program (with -n 5) against the output from the `uniqify.bash` script using the websters.txt sample file as input.

The `all` target in your Makfile should build your program. You should use the following compiler flags for `gcc` as part of your `CFLAGS` in your `Makefile` (and produce a clean compile):

```
-Wall
-Wshadow
-Wunreachable-code
-Wredundant-decls
-Wmissing-declarations
-Wold-style-definition
-Wmissing-prototypes
-Wdeclaration-after-statement
```

Notes:

1. You do not (and should not) need to store the output from all the sort sub-processes and resort it in the combiner stage. If you already know the data from the sort sub-processes is in the correct order, you can assemble the merged word lists also in order. If you re-sort in the combiner, you cannot receive the points for having the sort sub-processes. This will take some analysis on your part to see how this can be done.
2. When you've used pipes (like Problem2 in this homework), it is easy to find the termination condition and pass that to the downstream stages. With pipes, when you find EOF on the pipe, just close the pipe and you are done. Message queues are different. Remember that message queues can outlive the creating process. If you remove a message queue that another process it waiting on, you achieve a segmentation fault. We don't like segmentation faults do we? You need to plan how you will pass along the termination condition to the sort sub-processes and to the combiner process. It is (probably) not until the all the data have been read and parted out by the parser process that the sort sub-processes produce any output.
3. The final set of timing runs for your spreadsheet will probably need to be run on `eos-class`. You'll need to have multiple (physical) processors (and maybe a bit of memory) to see any meaningful speedup from the parallel sort sub-processes.
4. The sample input files can also be found on `eos-class` in: `/usr/local/classes/eecs/winter2013/cs311-400/src/Homework5/Problem3` Not everyone on `eos-class` needs to have a personal copy of these files, symbolic links are a good thing.
5. You can assume that you are working with ASCII content files (none of UTF-8 stuff). If you find any UTF-8, assume that I made a mistake and let me know.
6. One of the first things you are going to need to do is establish what your message type structure is going to contain. You don't have to call is mess_t; you can call it earl_gray_t if you want.
7. Right after that, you are going to need to establish what are the message types. Use #define value to represent message types. It makes your code easier to debug and a lot easier to read.
8. As messages arrive at your sort processes, you'll need to allocate (`malloc()` and buddies) space to store the data in the message.
9. Just as a question (with no points attached, just for thought), how might you implement this application using a single message queue for all communication?
10. The `ipcs` and `ipcrm` commands are going to be your new BFFs.
11. The message queues have a pretty limited capacity for the amount of data stored in them. So, your program could block on a write to a message queue. Don't expect to be able to pour thousands of messages into a queue without the program blocking. You'll need to be reading the other end of that queue.
12. Even though you'll need to fork new child processes, you don't necessarily have to exec over top of the child processes. I built my application within a single binary.
13. The message queue tennis code might actually be helpful on with this.

The point distribution for `uniqify` is:

Task	Number of points
Just for you C code compiling	5
Spreadsheet with graph	15
Topology slides	15
`test` target in `Makefile`, as described above (with "-n 5")	20
Creating the sort sub-processes based on the –n command line option.	65
Sorted, unique, and counted output	80
Total	**200 points**

Optional Extra Credit (worth up to 20 points): Write the same program, but use POSIX message queues instead of System V message queues. Just as a warning, you may not be able to use as many sort processes with POSIX message queues as you do with System V message queues. Tell me why (5 points of the optional 30). If you do the extra credit, put it in the same directory as the System V message queue solution. After you are done, let me know which you prefer, the System 5 message queues or the POSIX message queues. I was surprised by my answer on this.

Things to include with the assignment (in a single bzipped file):

1. C source code (.c and .h files) for the solution to the posed problems (all files).
2. A Makefile to build your code.
3. A PowerPoint/LibraOffice file showing the process topologies.
4. An Excel/LibraOffice file showing the timing results.
5. The extra credit source code (if you do that portion).

Please combine all of the above files into a single bzip file prior to submission.