

CS 385 –Operating Systems – Fall 2011

Homework Assignment 1

statsh – Processes and System Calls

Due: Tuesday 27 September. Electronic copy due at 3:00 P.M., optional paper copy at the beginning of class.

Overall Assignment

For this assignment, you are to write a shell named statsh that will report process statistics for all of the child processes that it has launched. The shell will incorporate the appropriate system calls to implement file I/O redirection, pipes, and background processes.

References

The following man pages will also prove useful to you in completing this assignment. Note very carefully the section of the manual given for each command – For many of them it is necessary to specify the manual section when running man, such as typing “man 3 exec” instead of just “man exec”. Also note that some of these man pages contain more than one useful command, such as dup and dup2, and 6 versions of exec.

- getline(3)
- string(3)
(strtok, strlen, strcpy, etc.)
- There may be others – follow the “see also” sections of the above man pages.
- You may also use the string class available in C++, and/or STL data structures such as the vector template. See separate documentation for the use of those tools.
- fork(2)
- exec(3)
- wait(2)
- wait4(2)
- getrusage(2)
- pipe(2)
- dup2(2)
- open(2)

Evolutionary Development

It is recommended that you develop your program in the following stages:

1. Issue a prompt, read in a line from the user, parse the tokens, and repeat until the user enters “exit”. The string commands, particularly **strtok** may be useful for this step. You will need to store the input line in a traditional C-style null-terminated array of chars, which you can do either by declaring the array of some (large) size and using **getline**, or reading in a C++ style string object and then using the `c_str()` function to generate the C-style array. You will then need to create an array of char pointers to null-terminated words as generated by strtok.
2. Implement fork and exec to create a child process, passing along any relevant arguments. Wait for the child process using the wait system call. (It is recommended to use **execvp** and **wait4**.) At this stage no I/O redirection, pipelining, or background processes should be considered.
3. Record user and system time of each child as it terminates. These should be reported for individual children as they complete, and a full history should be available by typing “stats”. The wait4 system call will return the desired information for a particular waited-for child. See the man page for **getrusage** for a full description of the **rusage** data structure.

4. Support multiple piped commands on a single line by separating the line into separate commands first using `strtok` and the “|” pipe symbol for the separator, and then separating each command into separate arguments with the space separator as was done in step 1. (You will now need **two** arrays of char pointers - One to point to the beginning of each command as parsed by `strtok('|')` and the other to point to each word of the current command as implemented in step 1 above.) Essentially this just adds an extra level of nested loop around the work already done. Fork and exec each command as a separate child as in step 2 above, without waiting for the child processes to terminate. Then wait for all the children in a separate loop as in step 3 above. (Note that at this point in your evolutionary development the input and output of the piped commands are not connected yet, so the goal here is just to launch multiple independent commands from a single command line.)
5. Then modify the program to connect the input and output of adjacent commands in the pipeline. Use **pipe(2)** to create pairs of file descriptors for use by adjacent commands in the pipeline, one for input and one for output. The best approach is to create each new pipe only as needed, and then close off the unused pipe ends as soon as possible, (as opposed to creating all the pipes at once.)

The parent shell needs to create the pipe objects one-by-one prior to forking off each child. When the fork occurs, then both the parent and child will have copies of the open file descriptors, and each will need to close off those that they are not using. The child will also need to re-direct the new file descriptors to replace the previous values of standard input and/or standard output (`stdin`, `stdout`), and then close the file descriptor that was duplicated.

- a. There may be as many as 3 open file descriptors at the time the fork occurs - One this child may read from, one it may write to, and one that the following child may read from. Any of these file descriptors that are not being used should be set to -1 (e.g. for the ends of the pipeline or for commands that do not use pipelining.)
 - b. Parent creates the pipe (2 open file descriptors) for the output of the next command to be forked, and then forks off the child process. The parent can then close off the file descriptors used by this child (either one or two) and continue the loop to start working on the next child.
 - c. Each child should first close off the file descriptor that they will definitely not use (the one to be read by the following child), unless it is set to -1.
 - d. Then each child uses **fdup2** to replace either `stdin` or `stdout` (or both) with copies of the appropriate open file descriptors (if they are not -1), and closes the original open file descriptors after they have been copied. (File descriptor 0 is `stdin`, and 1 is `stdout`.) After the copying and closing are done, then the child can exec the command as in step 2.
 - e. The parent shell does not wait for any of these children until after they are all forked. Then it waits for each child in turn, so that it can collect the usage statistics.
6. Implement file I/O redirection, by parsing the first and last commands in the pipeline for < and > respectively. If found, open the appropriate file using a system call to “**open**”, and place the file descriptor in the appropriate file descriptor variable used in step 5 above. The child code doesn’t change, because the child doesn’t know (or care) if a given file descriptor came from a **pipe** or from an **open**.

7. Implement the option of background processes, and recording the times thereof. Initially this is a simple modification – If the `&` is present, the shell uses the `WNOHANG` option with `wait4`, and continues if the child has not yet completed. However, any child that does not complete immediately must be remembered, and waited for again just prior to issuing any new prompt. (Note that there could theoretically be any number of background processes running at any given time, and if a pipeline is run as a background operation, then all the commands in the pipeline are backgrounded.

Program Interaction, Testing, and Use

- The `statsh` should operate like a normal shell, providing usage statistics as each child completes or when the “`stats`” command is issued. Upon shell exit, summary statistics should be reported, including the times used for the shell processing itself. (Use `getrusage(2)`).
- Because the user input parsing is complex for this assignment, you do not need to handle unreasonable input, such as output redirection and piping in bad combinations. (E.g. “`ls > myfile | more`”)
- If you have any questions regarding how your shell should handle a certain situation, ask yourself: (A) How do other shells handle that situation?, and (B) How do I want my shell to handle that situation? Ultimately most of this is left to the programmer’s discretion, but the decisions you make are part of your design.
- For testing purposes, you can launch the `statsh` like any other program, and it will run as a child of your regular shell. When you exit out of `statsh`, you will be returned to your normal shell.
- Alternatively you can type “`exec statsh`” at your normal shell prompt, which will cause your normal shell to be **replaced by** the `statsh`. When you exit the `statsh` it will log you out of the system.
- If you **really** like the shell you wrote, you can use it as your normal shell by placing the “`exec statsh`” command in your configuration / startup file for your regular shell, so that it gets launched automatically whenever you log in. (e.g in `.tcshrc`). OR, you could ask your System Administrator to configure your account to use the `statsh` as your normal shell. (Don’t expect that to happen on the CS or ACCC computers. 😊)

Required Output

- All programs should print your name and ACCC user name as a minimum when they first start.
- Beyond that, this program should print statistics as shown above, and support normal shell I/O redirection.

Other Details:

- You must do your own **INDIVIDUAL WORK** on this assignment. This is not a group assignment, and you are not allowed to confer with any current or past students of this course. You may find relevant examples in various textbooks and on-line documentation, but ultimately you have to write your own code – Don’t just copy the examples, and above all, do not copy anyone else’s code. The MOSS anti-cheating program will be used to compare the submissions for this semester with those of both other students and of past semester submissions, and excessive similarities will be noted and dealt with harshly. If you need help with your program, come to the instructor or TA, do not copy from your buddy. (ACM tutoring is allowable, but only for general programming and conceptual issues, and the proper implementation of system calls. They should not write or debug any of your code.)

- Many students typically wait until the last week before an assignment is due before they begin to work on it or start worrying about it. **THAT WILL NOT WORK FOR THIS ASSIGNMENT.** It is recommended that you plan to complete the first three items in the evolutionary development during the first week, and the remaining four items during the second week. That will then leave you one more week to debug and polish up the program, complete the documentation, add optional enhancements, and gives you some extra time if you fall behind schedule.
- The TA must be able to build your program by typing "make statsh". If that does not work using the built-in capabilities of make, then you need to submit a properly configured makefile along with your source code. As always, you are free to develop wherever you wish, but the TA will be grading the programs based on their performance on the CS department Mac OS/X machines.

What to Hand In:

1. Your code, **including a makefile if necessary, and a readme file**, should be handed in electronically using Blackboard.
2. The purpose of the readme file is to make it as easy as possible for the grader to understand your program. If the readme file is too terse, then (s)he can't understand your code; If it is overly verbose, then it is extra work to read the readme file. It is up to you to provide the most effective level of documentation.
3. The readme file must specifically provide direction on how to run the program, i.e. what command line arguments are required and whether or not input needs to be redirected. It must also specifically document any optional information or command line arguments your program takes, as well as any differences between the printed and electronic version of your program.
4. If there are problems that you know your program cannot handle, it is best to document them in the readme file, rather than have the TA wonder what is wrong with your program.
5. A printed copy of your program, along with any supporting documents you wish to provide, (such as hand-drawn sketches or diagrams) may be handed in **at the beginning of class** on the date specified above.
6. Make sure that your **name and your ACCC account name** appear at the beginning of each of your files. Your program should also print this information when it runs.

Optional Enhancements:

It is course policy that students may go above and beyond what is called for in the base assignment if they wish. These optional enhancements will not raise any student's score above 100 for any given assignment, but they may make up for points lost due to other reasons. Note that all optional enhancements need to be clearly documented in your readme files. The following are some ideas, or you may come up with some of your own:

- Use `getrusage(2)` to report other process statistics besides just CPU usage. Effective implementation will involve thinking about what other statistics are desired, how to format the output nicely, and how the user can specify what statistics interest them. (Later in the course we will be interested in page faults, swapping, and context switches.)
- A command history feature, that will allow users to re-issue old commands using the ! and/or ↑.
- Control of the working environment, including `cd`, `pwd`, and the setting and reporting of environment variables.
- Wild card matching for file and directory names.
- Command completion.
- Any other shell features that you wish to implement, such as shell scripting support. To get ideas you can either look at the documentation for the other shells (`man` on `sh`, `csh`, `bash`, `tsh`, `ksh`, etc.), or simply use the shell and see what sort of features you wish it had.