CS 361 – Computer Systems – Fall 2017
Homework Assignment 2
Process Creation and Signal Handling

Due: **Thursday 5 October. Electronic copy due at 9:00 A.M., optional paper copy may be delivered to the TA during lab.**

Overall Assignment

For this assignment, you are to write two related programs in either C or C++ as you choose - One that acts as a simple shell, and the other of which exercises the handling of several different signals. More specifically, the second program will use the Monte Carlo method to estimate what fraction of the total address space is inaccessible, by attempting to access random locations and reporting what fraction of attempts are unsuccessful.

References

The following man pages may 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 6 versions of exec.

- getline(3)
- string(3) (strtok, strlen, strcpy, etc.)
- fork(2)
- exec(3)
- wait(2), wait4(2)
- getrusage(2)
- perror(3), errno(3)
- signal(2), signal(7)
- kill(1), kill(2)
- getppid(2)
- alarm(2), setitimer(2)
- 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.

Program I - A Simple Shell

This program shall:

1. Issue a command prompt and read in a command line from the user.
2. Parse the line into a command followed by arguments using strtok( ), creating an array of char pointers, where array[0] points to the command and the rest of the array points to the rest of the arguments. (Equivalent to argv[ ] in main( ).)
3. Fork off a child, and have that child load the requested program by passing the argument vector created in step 2 to execvp( ). The parent should report the PID of the child that was launched before proceeding to step 4.
4. Wait for the child to complete using wait4( ), then report why it ended (exited or uncaught signal), its exit value if available, the number of page faults it experiences, and the number of signals it received.
5. Repeat from step 1 forever, until the user enters a command of "exit".

Note: The above describes the initial behavior of the shell. Later it will be modified to communicate with the second program via signals, as part of developing the second program. The shell does not need to support pipelines, file redirection, background processes, or more than one child at a time. It does, however, need to support any valid command that the user might enter, (not counting shell built-ins such as "cd".)

Program II - A Monte Carlo Simulation

The basic idea behind the Monte Carlo method is to perform a large number of "evaluations" based on random inputs, and to count how many (or what percentage) of the evaluations fulfill a certain criteria. For example, by calculating random points within a unit square and counting what fraction also land within a unit circle it is possible to estimate pi, based on the ratio of the area of the circle to the area of the square.

For this particular assignment, random addresses will be generated, and a count will be kept of how many of the random addresses result in segmentation faults when an attempt is made to read from that location. The ratio of segmentation faults generated to the total number of random addresses tried will give a reasonable estimate of what fraction of the total address space is inaccessible. More specifically, this program shall:

1. Create and initialize to 0 two static global counters (unsigned long ints) for the number of evaluations attempted and the number of segmentation faults generated. Also initialize the random number generator with srand(time(null)) and parse any command line arguments that may be present.

2. Set up signal handlers to catch and respond to certain signals (see below.)

3. Enter a for( ) loop to generate and test random addresses, using the global evaluation counter for the loop counter. Eventually the upper limit on the loop counter will be the maximum possible unsigned long integer, (ULONG_MAX, see limits.h), but for initial testing you will want to use something smaller. Each time through the loop the program shall:

    a. Call setjmp( ) to remember the context at the beginning of the loop.

    b. Call rand( ), and store the return value in an int * variable.

    c. Try to copy the data pointed to by that pointer into an int variable.

4. If the program successfully completes the for( ) loop, then it should report the number of segmentation faults counted, the total number of evaluations attempted, and the percentage of attempts that resulted in segmentation faults. The program should then exit with an exit value equal to the percentage calculated, as a whole number (i.e. typecast to an int).

Signal Handlers Required

The majority of these signal handlers will be needed in the second program, though in one case handlers will be needed in both programs.

- SIGSEGV - In the event of a segmentation fault, the handler should increment both of the global counters. If the evaluation counter has not reached its limit then the handler should "return" using longjmp( ). Otherwise it should report the final results and exit, as described in step 4 above.

- SIGINT - Generated by control-C, this signal handler allows the user to stop the program "nicely" at any time. The signal handler should simply report the final results and exit as described above.

- SIGTSTP - Generated by control-Z, this signal handler should report the current results without stopping the program. Because no errors have occurred and this handler doesn't exit, it should simply return when complete.

- SIGALRM - The program shall take a floating point command line argument as the maximum number of seconds to run before stopping. If this value is a positive number, then the program should call alarm( ) before starting the for( ) loop with this time value as the alarm time. Should the alarm go off the response should be the same as for control-C, and in fact could possibly be the same handler. (Note: alarm( ) only accepts an integral number of seconds. Alternatively setitimer( ) will set an interval timer with sub-second resolution, but it is more complicated to use.)

- SIGUSR1 / SIGUSR2 - Whenever a power of 10 number of evaluations has been performed, then the accuracy of the current estimate increases by one digit of precision. For example, after 100 evaluations have been performed the answer is known to within 1/100, or two digits of precision, and when 1000 have been performed, then the answer is known to within 1/1000, or 3 digits of precision. This progress will be communicated to the parent (the shell you write) in the following manner:

    - Every time the number of evaluations passes another power of 10, the child should use the kill( ) command to send a SIGUSR1 signal to the parent. (The easiest way to do this is to set a "report" variable initially to 10, and every time the number of evaluations matches the report variable send the signal and multiply the report variable by 10.)

    - The parent will now need a signal handler to catch SIGUSR1, and increment a counter each time it arrives. If the counter exceeds a limit set on the command line when the shell was launched, then the parent uses the kill( ) system call to send a SIGUSR2 signal to the child. (The counter will need to be reset to zero sometime before launching the next child.)

    - The child will now need a signal handler to catch SIGUSR2. The response should be the same as that for control-C, and in fact could probably be the same handler.

Command Line Arguments

- The shell program shall (eventually) take a single command line argument - an integer for the number of digits to let the child calculate before stopping the calculations. (The number of SIGUSR1 signals to count before sending back a SIGUSR2.) If the argument is missing or less than one, then three should be assumed.

- The Monte Carlo simulation shall take a single floating point argument for the maximum number of seconds to calculate unless interrupted by some other stopping conditions. If this argument is absent or non-positive, then no alarm should be set.

Required Output

- All programs should print your name and ACCC user name as a minimum when they first start.
- Beyond that, the two programs should report the results described above.

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 child termination results should be reported.

3. Report the results of each terminated child as described above. 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. Develop the Monte Carlo simulation with a small number of iterations and only the segmentation fault interrupt handler in place.

5. Add additional signal handlers one by one, testing each one to make sure it works before adding the next handler.

Other Details:

- The TA must be able to build your programs by typing "make". 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 machines.

What to Hand In:

1. Your code, **including a makefile 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 **to the TA** on or shortly after the date specified above. Any hard copy must match the electronic submission exactly.
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:

- Enhance the shell with additional features, such as I/O redirection, pipes, background processes, shell built-ins, command completion, and any other features you would like your shell to have.
-