CSCI 4210 — Operating Systems
Homework 2
Process Creation and Process Management in C

Overview

- This homework is due by 11:59:59 PM on Friday, February 28, 2020.
- This homework is to be completed **individually**. Do not share your code with anyone else.
- You **must** use C for this homework assignment, and your code **must** successfully compile via `gcc` with absolutely no warning messages when the `-Wall` (i.e., warn all) compiler option is used. We will also use `-Werror`, which will treat all warnings as critical errors.
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v18.04.2 LTS. Note that the `gcc` compiler is version 7.4.0 (`Ubuntu 7.4.0-1ubuntu1~18.04`).

Homework specifications

In this second homework, you will use C to implement a rudimentary interactive shell similar to that of `bash`. The focus of this assignment is on process creation, process management, and inter-process communication (IPC) via `fork()`, `wait()`, `waitpid()`, `pipe()`, etc.

As with Homework 1, continue to use `calloc()`, `realloc()`, and `free()` to properly and efficiently manage your memory usage. Consider using `valgrind` to verify that there are no memory leaks. We will continue to test for this on Submitty.

To properly implement your shell, create an infinite loop that repeatedly prompts the user to enter a command, parses the given command, locates the command executable, then executes the command (if found).

To execute the given command, a child process is created via `fork()`, with the child process then calling `execv()` to execute the command. In the meanwhile, the parent process calls `waitpid()` to suspend its execution and wait for the child process to terminate. This is called *foreground processing*. (And note that you must use these specific system calls.)

If instead the command is to be run with the parent process not waiting for the child process to complete its execution, then your shell will instead use *background processing*, which is achieved by using the `&` symbol (and explained in more detail on page 4).

Locating the command executable

Before executing a command entered by the user, the command executable must be found using the list of possible *paths* specified by an assignment-specific environment variable called `$MYPATH`. Do **not** use `$PATH` for this assignment (since `$PATH` is used for `bash`).

Similar to `$PATH`, this new `$MYPATH` environment variable consists of a series of paths delimited by the `:` character. And if `$MYPATH` is not set, use `/bin:.` as the default, meaning commands will be searched for first in the `/bin` directory, then the `.` (i.e., current) directory. By default, the `$MYPATH` variable is not set, so for testing, set and unset this variable manually in the `bash` shell before running your shell. Here's how:

```
bash$ export MYPATH=/usr/local/bin:/usr/bin:/bin:.
bash$ echo $MYPATH
MYPATH=/usr/local/bin:/usr/bin:/bin:.
bash$ unset MYPATH
```

To obtain `$MYPATH` (or any environment variable, e.g., `$HOME`) from within your program, use the `getenv()` function. Do **not** use `setenv()`.

Executing the command

Searching left-to-right in `$MYPATH`, if the requested command is found in one of the specified directories, your program runs the executable **in a child process** via the `fork()` and `execv()` system calls. Note that you **must** use `execv()`.

Further, **in the parent process,** you must use `lstat()` to determine whether the requested command exists (e.g., does `/bin/ls` exist?) and whether it is executable. See the `man` page and the `directories.c` example.

Commands are line-based, as in `bash`. Therefore, each command may optionally have any number of arguments (i.e., `argv[1]`, `argv[2]`, etc.). You can assume that each command read from the user will not exceed 1024 characters. Further, you can assume that each argument will not exceed 64 characters, but all memory must be dynamically allocated.

You may also assume that command-line arguments do not contain spaces. In other words, do not worry about parsing out quoted strings in your argument list, as in:

```
bash$ cat a.txt b.txt "some weird file.txt" d.txt
```

Special shell commands

Not all commands entered into the shell actually result in a call to `fork()`. For the `cd` command, if your shell did execute the command via `fork()`, your shell's current working directory would not change! Therefore, you must use the `chdir()` system call in the parent to handle this special case. Further, if the `cd` command has no arguments, then you should use the `$HOME` environment variable as the target directory.

As for wildcards and special characters, you do not need to support `*`, `?`, and `[]` in your shell, though note that these are typically expanded by the shell before calling `fork()` and `execv()`.

Finally, to exit your shell, the user enters `exit`. When this occurs, your shell must output `bye` and terminate.

Required output

The command prompt in the shell must show the current working directory followed by the `'$'` prompt character and one space. To obtain the current working directory for the running process, use the `getcwd()` function. And use `fgets()` to read in a command from the user.

Required output is shown below, with sample input also shown. As per usual, you must match the given output format exactly as shown.

```
/cs/goldsd/s19/os/assignments/hw2$ cocoapuffs
ERROR: command "cocoapuffs" not found
/cs/goldsd/s19/os/assignments/hw2$ ls
annoying.c  a.out  code  hw2.aux  hw2.log  hw2.out  hw2.pdf  hw2.tex
/cs/goldsd/s19/os/assignments/hw2$ ls -l
total 156
drwxrwx--x  3 goldsd goldsd  4096 May 27 15:08 .
drwxrwx--x  5 goldsd goldsd    51 May 27 11:59 ..
-rw-rw----+ 1 goldsd goldsd   197 May 27 15:08 annoying.c
-rwxrwx--x  1 goldsd goldsd  8344 May 27 15:08 a.out
drwxrwx--x  2 goldsd goldsd     6 May 27 15:04 code
-rw-rw----  1 goldsd goldsd   662 May 27 15:04 hw2.aux
-rw-rw----  1 goldsd goldsd 20507 May 27 15:04 hw2.log
-rw-rw----  1 goldsd goldsd     0 May 27 15:04 hw2.out
-rw-rw----  1 goldsd goldsd 97289 May 27 15:04 hw2.pdf
-rw-rw----+ 1 goldsd goldsd  9886 May 27 15:07 hw2.tex
/cs/goldsd/s19/os/assignments/hw2$ cat annoying.c
/* annoying.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  while ( 1 )
  {
    printf( "Hey, get back to work!\n" );
    sleep( 3 );
  }

  return EXIT_SUCCESS;
}
/cs/goldsd/s19/os/assignments/hw2$ exit
bye
```

Foreground and background processing

Normally, a shell will execute the given command via a child process, with the parent calling either `wait()` or `waitpid()` to wait for the child process to complete its command and terminate. The child process essentially calls `execv()` to execute the given command with the arguments allocated dynamically.

Your shell must be able to execute a process in the background if the user includes an ampersand (i.e., `'&'`) at the end of a command. In this case, when the child process is created, the parent does **not** wait for the child to terminate before prompting the user for the next command.

For a background process, the parent must report that the child process has been created.

The parent must also report when the child process does terminate (if it does). When you detect that the background process has terminated (i.e., before you display the prompt), display the child process ID and its exit status, as in:

```
[process 9335 terminated with exit status 0]
```

This should be the normal reporting for all children that terminate whether they return EXIT_SUCCESS or an error code.

For background child processes that terminate due to a kill signal or other abnormal termination, display the following:

```
[process 9335 terminated abnormally]
```

Note that the `&` symbol can only be included at the end of the command line; otherwise, this is a user error. Also note that the output may be interleaved with background processes, so do not expect to always match the example output exactly line for line.

```
/cs/goldsd/s19/os/assignments/hw2$ ls
annoying.c  a.out  code  hw2.aux  hw2.log  hw2.out  hw2.pdf  hw2.tex
/cs/goldsd/s19/os/assignments/hw2$ a.out &
[running background process "a.out"]
Hey, get back to work!
/cs/goldsd/s19/os/assignments/hw2$ Hey, get back to work!
Hey, get back to work!
Hey, get back to work!
/cs/goldsd/s19/os/assignments/hw2$ ls
annoying.c  a.out  code  hw2.aux  hw2.log  hw2.out  hw2.pdf  hw2.tex
/cs/goldsd/s19/os/assignments/hw2$ Hey, get back to work!
Hey, get back to work!
Hey, get back to work!
/cs/goldsd/s19/os/assignments/hw2$ exit
bye
Hey, get back to work!
Hey, get back to work!
```

For the above example, you will need to use `kill` in the `bash` shell to terminate the background process since it will continue to execute after your shell terminates. And note that for this assignment, you are required to use `waitpid()` for both foreground and background processes.

Do not do anything extra to kill any child processes, in particular when the user exits your shell. And to properly "catch" background processes when they terminate, mimic the behavior of `bash` by checking for terminated background processes immediately before you display the prompt to the user. **Do not use a signal handler for this.**

IPC via pipes

Finally, add support for a pipe between two processes; you need only support one pipe per command line. Two processes (i.e., A and B) may be connected via a pipe such that the output on `stdout` from process A is the input on `stdin` to process B.

A pipe is indicated via the `|` symbol. To create a pipe, use the `pipe()` system call. Further, the two processes A and B must have your shell process as their parent process.

```
/cs/goldsd/s19/os/assignments/hw2$ ps -ef | grep goldsd
root     23553  1414  0 15:00 ?        00:00:00 sshd: goldsd [priv]
goldsd   23556     1  0 15:00 ?        00:00:00 /lib/systemd/systemd --user
goldsd   23558 23556  0 15:00 ?        00:00:00 (sd-pam)
goldsd   23714 23553  0 15:00 ?        00:00:00 sshd: goldsd@pts/0
goldsd   23715 23714  0 15:00 pts/0    00:00:00 -bash
goldsd   23716 23715  0 15:00 pts/0    00:00:00 myshell
root     23729  1414  0 15:01 ?        00:00:00 sshd: goldsd [priv]
goldsd   23813 23729  0 15:01 ?        00:00:00 sshd: goldsd@notty
goldsd   23814 23813  0 15:01 ?        00:00:00 /usr/lib/openssh/sftp-server
goldsd   24615 23716  0 15:15 pts/0    00:00:00 ps -ef
goldsd   24616 23716  0 15:15 pts/0    00:00:00 grep goldsd
/cs/goldsd/s19/os/assignments/hw2$ ls -1
annoying.c
a.out
code
hw2.aux
hw2.log
hw2.out
hw2.pdf
hw2.tex
/cs/goldsd/s19/os/assignments/hw2$ ls -1 | wc -l
8
/cs/goldsd/s19/os/assignments/hw2$ exit
bye
```

Pipes and background processes

Note that a pair of piped processes can be run in the background if the user specifies an ampersand at the end of the line. When run in the background, both processes are background processes. And when these processes terminate, show both processes.

Here is an example with `12117` and `12118` as the process IDs of the two background processes:

```
/cs/goldsd/s19/os/assignments/hw2$ ls -1 | wc -l
8
/cs/goldsd/s19/os/assignments/hw2$ ls -1 | wc -l &
[running background process "ls"]
[running background process "wc"]
/cs/goldsd/s19/os/assignments/hw2$
8
[process 12117 terminated with exit status 0]
[process 12118 terminated with exit status 0]
/cs/goldsd/s19/os/assignments/hw2$ exit
bye
```

As noted previously, output may be interleaved with background processes, so do not expect to always match the example output exactly line for line.

Relinquishing allocated resources

Be sure that all processes (i.e., the parent shell process and all child processes) properly deallocate memory via `free()`, close all opened file descriptors, etc.

Error handling

If improper command-line arguments are given, report an error message to `stderr` and abort further program execution. In general, if an error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then abort further program execution.

Error messages must be one line only and use the following format:

```
ERROR: <error-text-here>
```

Note that you should **not** abort your shell program if the user enters an invalid command (e.g., command not found) or if the child process reports an error.

Submission instructions

To submit your assignment (and also perform final testing of your code), please use Submitty, the homework submission server.

Note that this assignment will be available on Submitty a minimum of three days before the due date. Please do not ask when Submitty will be available, as you should first perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, as discussed in class, use the `DEBUG_MODE` technique to make sure you do not submit any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of x is %d\n", x );
    printf( "the value of q is %d\n", q );
    printf( "why is my program crashing here?!" );
    printf( "aaaaaaaaaaaaaggggggggghhhh!" );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE hw2.c
```

Second, as discussed in class, output to standard output (`stdout`) is buffered. To disable buffered output for grading on Submitty, use `setvbuf()` as follows:

```
setvbuf( stdout, NULL, _IONBF, 0 );
```

You would not generally do this in practice, as this can substantially slow down your program, but to ensure good results on Submitty, this is a good technique to use.