Operating Systems, Assignment 0

This assignment includes a few programming problems and a chance to read some of the online documentation for POSIX system calls. When grading, we will compile and test your programs on Gradescope. In general, we'll compile with options like the following. In some cases, we will need to add some extra options for a particular programming assignment. If a programming problem needs different compiler options, we'll let you know about those in the problem description.

```
gcc -Wall -g -std=c99 -o program program.c
```

In the command above, the -g option tells the compiler to include symbol information in the executable, to help debuggers do their job. The -Wall option turns on lots of compiler warnings. This may help you to find errors in your code, even if the errors aren't enough to prevent compilation. The -std=c99 option enables some C99 language features you're probably accustomed to.

If you develop a program on a different system, you'll want to make sure it compiles and runs correctly on Gradescope - you can submit your code unlimited amount of times before the deadline to see the feedback from the auto-grader, and after the deadline, we will only grade your last submission. Also, remember that your programs need to be commented and consistently indented, and that you **must** document your sources and write at least half your code yourself from scratch.

1. (16 pts) Log in to one of the EOS Linux machines (one of the Linux desktop machines in EB 3, in the library, in 111 Lampe (formerly Daniels Hall), or via remote.eos.ncsu.edu) and use the online manual pages to answer each of the following questions. To read the online manual page for a particular command, function or system call, you can usually just type `man` *call*, where *call* is the call you want to read about. If two different manual pages have the same name, you can specify the section of the manual to make sure you get the page you want. Section 1 of the manual is for shell commands, and section 2 is for system calls. So, **man 1 stat** will tell you about the stat shell command, and **man 2 stat** will tell you about the system call with the same name.

   Write your answers into a text file, convert it to a pdf file called `hw0_p1.pdf` and submit it to HW0_Q1 on Gradescope.

   (a) The open() system call can be used with two parameters or with three parameters. What determines whether the operating system will expect two parameters or three parameters?

   (b) When you call the read() system call, it may encounter an error. As a programmer, how can you tell the difference between when read reaches the end-of-file and when it encounters some type of error?

   (c) It's possible for the fork() system call to fail because there are already too many processes. If this happens, how can the programmer tell that fork() failed for this particular reason.

   (d) Imagine you create several child processes with fork() then call wait(). The call to wait() will return after any one of the child processes terminates. After wait() returns, how can you tell which of the child processes terminated?

2. (10 pts) Provide brief answers to the following questions. Write your answers into a text file, convert it to a pdf file called `hw0_p2.pdf` and submit it to HW0_Q2 on Gradescope.

   (a) (3 pts) What is the difference between temporal and spatial locality?

   (b) (3 pts) Does the following code snippet exhibit temporal and/or spatial locality and why?

   ```
   for (int i = 0; i < 10; i++) {
     printf(some_array[i]);
   }
   ```

(c) (4 pts) Why are interrupts and traps essential to modern operating system design? Which aspects of system functionalities are dependent on them?

3. (40 pts) For this problem, you will be implementing a program called `exclude.c` using the Unix system call interface. We're going to do this without any functions from the C Standard Library; this will let us see a little bit of how the C library is implemented on a Unix system. Internally, this library must do everything it does using only C code and calls to the operating system.

   Once your `exclude` program is working, you should be able to run it like the following example:

   ```
   $ ./exclude input-1.txt output.txt 3
   ```

   The first argument gives the name of an input file, and the second argument gives the name of an output file the program should write to. The last argument should be positive integer giving the number of a line from the input file. The job of the program is to copy over everything from the input file to the output file, excluding the line indicated by the number on the command line. Lines are numbered counting from 1 (the first line of the input file).

   In the example above, we are asking the program to copy all lines except line 3 from input-1.txt to output.txt. A sample input file, input-1.txt, is provided with the assignment. The following figure shows what it contains and what the output file would contain after running the exclude program as shown above. The output file looks just like the input file, except the 3rd line is missing.

input-1.txt	output.txt
This is a sample input file for the exclude program. You can give a command-line argument to tell it what line to omit when copying to the output.	This is a sample input file for the You can give a command-line argument to tell it what line to omit when copying to the output.

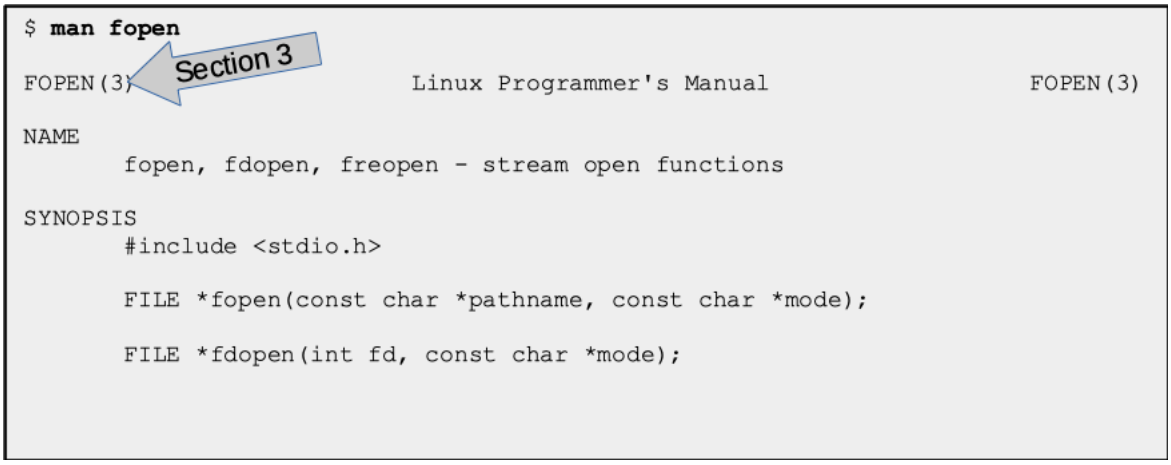
Implementation

You'll solve this problem using only Unix system calls. You can't use any functions from the C standard library. This will give you a chance to see what the system call interface looks like and how the C standard library adds functionality to make it more portable and easier to use.

Using just the system call interface means that you can't use functions like fopen(), fscanf(), fprintf() and fclose(). You'll need to use system calls like open(), read(), write() and close() instead. Likewise, you can't use functions like atoi() or strlen(). If you need to convert from a string to an integer or to measure the length of a string, you'll need to write your own function or block of code to do that. It's not too difficult.

To avoid accidentally using something from the C library, make sure you don't included any of its headers. That way, if you try to use a standard library function, the compiler will complain. The standard library headers are listed here: `https://en.cppreference.com/w/c/header` If you're unsure about a particular function, you can check its online documentation using the `man` command. Like it

shows below, if a function is part of the C standard library, the man command will tell you it's in section 3 of the manual. This should be in parentheses up near the top.

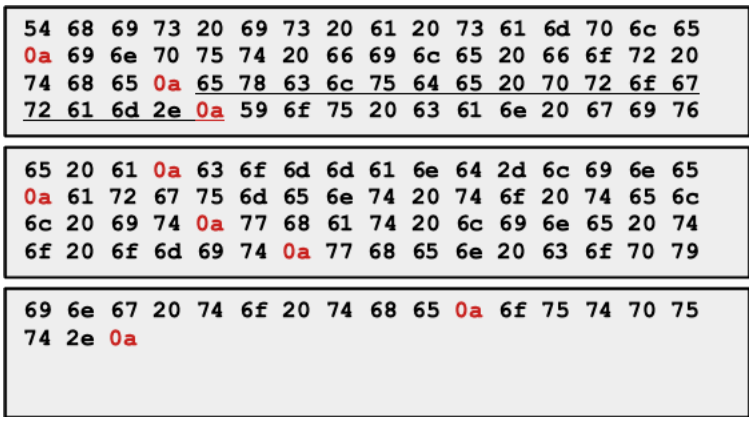


Without the standard library, you'll need to write your own code to convert the last command-line argument from a string to an integer. You can do this by going through that argument a character at a time and converting each character from an ASCII character code to a decimal value. Since it's a decimal number, its last (rightmost) digit is the one's place. The next one to the left is the tens place, so it's worth 10 times as much. The next one is the hundreds place, and so on. You should be able to write a little loop to check the validity of the last command-line argument and convert it from a sequence of ASCII characters to an integer value. It's OK if the line number has some extra zeros on the left; you don't have to flag this as an error. So, for example, 017 would be a legal way to give line number 17, but 5x2 and -904 would not be legal line numbers.

You'll need to read and write files using the read() and write() system calls. These read and write data as a block of bytes. You'll read the file in blocks of up to 64 bytes. The last read may not fill the 64-byte buffer, since the file size may not be a multiple of 64 bytes.

The read() and write() system calls don't use null terminated strings and they don't care where the line breaks fall within the file. Lines in the file are just sequences of bytes with a newline character marking the end. To exclude the line with the given number, you'll need code to look through each block after you read it and notice where the lines end. Each time you pass a newline character, you can count that as a line. For writing to the output file, you can copy over everything except the bytes in the one line specified by the last argument.

The following figure shows what the program would read on the example shown above. Each byte of the input-1.txt file is shown as two hexadecimal digits. The red ones are the newline characters, marking the ends of lines. The underlined bytes are line three. For the example above, these are the bytes that shouldn't be copied to the output file.



Invalid Arguments

Given invalid command-line arguments, the program should print the following usage message to standard error and terminate with an exit status of 1.

```
usage: exclude <input-file> <output-file> <line-number>
```

Command-line arguments would be invalid if the user didn't provide the right number of arguments, if the last argument wasn't a positive integer or if the program couldn't open one of the given files. It's OK if the given number is larger than the number of lines in the file. This isn't considered an error; it will just produce an output file that's identical to the input file.

Without the standard library, you won't be able to use fprintf() or stderr to print this message, and you won't be able to call exit() to terminate the program. For printing the usage message, you can use the write() system call to send the contents of the message to the file descriptor for the error output stream. Fortunately, the unistd.h header defines a constant, `STDERR_FILENO`, that gives the file descriptor for the standard error stream (it's 2).

For terminating the program, you can call the _exit() system call. It works just like exit() from the standard library, but it's a Unix system call rather than a standard library function.

Submission

When you're done, submit your source file, **exclude.c**, under the assignment named HW0_Q3 on Gradescope. You don't need to submit any additional files for this problem, and don't put your source file in an archive or a subdirectory. Just submit the file, `exclude.c`, directly to the HW0_Q3 submission locker.

4. (40 pts) Your own shell program: `stash.c`.

   The shell is what we call the command-line interpreter on a Unix system. When you log in to a terminal window and you type commands like "ls" or "cd", the shell is reading in your command, parsing it and deciding what to do. Shell programs are usually given names that end with "sh", like the `sh` (bourne shell), `bash` (the bourne again shell), `csh` (C shell), `ksh` (korn shell), or `zsh` (Z shell).

   For this assignment, we're going to write our own shell program, named `stash`. We'll say this stands for "simple toy assignment shell". It will have just a few of the features of a typical shell program, but this will be enough to show how a program can use system calls to start new programs and wait for them to finish.

For this problem, you're not restricted to using just system calls. You can use a combination of POSIX system calls and C standard library features as needed. That's typically how you might write a program. You'll use system calls where you need to and other, higher-level interfaces where possible.

Command Input

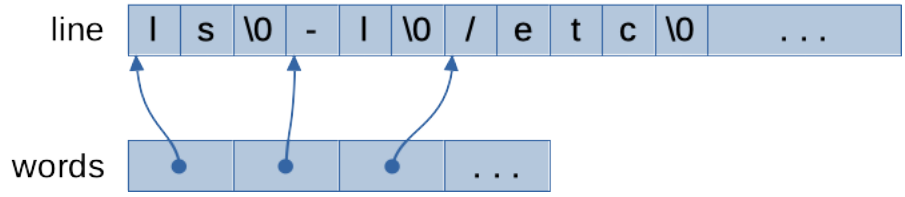
The stash.c program will prompt the user for commands using a prompt that looks like the following. There's a single space after the greater-than sign. You can't see it below, but you can see it in the sample execution later in this section.

```
stash>
```

The user can type a command of up to 1024 characters in length (not counting line termination or any null termination).

The stash program will need to break this input line up into individual words. The first word gives the name of the command the user is running, and the remaining words are command-line arguments that will be passed to the command when it is run. Here, a word is just a sequence of non-whitespace characters. There may be any amount of whitespace between words or at the start or end of the input line.

Your program will modify the input string by putting a null terminator at the end of every word. It will fill in an array of pointers with one pointer pointing to the start of each word. The following figure shows what this would look like for the sample command shown above.



Running Commands

After breaking the user input line into words, you're ready to execute the command. Our shell will support two types of commands. For most commands, the shell will fork a child process and then execute a separate program. A few commands must be implemented inside the shell itself, so our shell will support a few built-in commands, ones that don't require creating a child process or executing a separate program.

Built-in Commands

Our shell will support three built-in commands. These are listed below. For these, the shell will just check the first word of the command. If it's "cd", the shell will run the built-in cd command. If it's "exit", the shell will run the built-in exit command. If the input line is empty, the shell will just ignore the command and prompt the user again.

- `cd` *path*

  The `cd` command expects one argument. It will normally be an absolute or a relative path, but you won't need to check for that. You can just pass the argument to the chdir() system call to change the current directory. The return value of chdir() will tell you if the directory was valid. If the user enters the wrong number of arguments for `cd` or if they give a path that returns an error from chdir(), the program should print the following error message to standard output and prompt the user for another command.

  Invalid command

- exit *status*

  The `exit` command terminates the shell with the given integer value as its exit status. For the exit command, you can parse the given *status* as an integer then call exit() with that value as the argument. This will terminate the shell with that value as its exit status.

  If the user gives the wrong number of arguments for exit, or if they give an exit status that can't be parsed as an integer, the program should print the following error message to standard output and prompt the user for another command.

  Invalid command

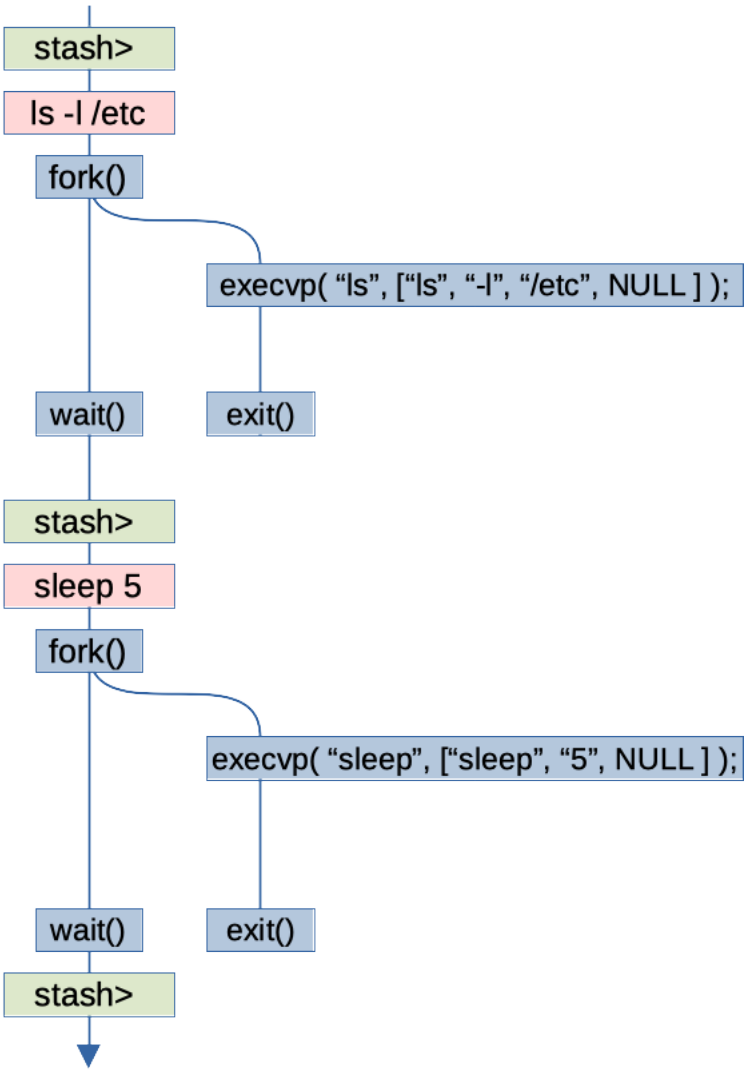
- *empty*

  If the user enters a blank line, the shell will just ignore it and prompt the user for another command.

External Commands

Other than the built-in commands, all entered commands will be run as a separate programs. This means that the shell will use fork() to create a child process, then it will use execvp() to run a different executable in the child. The execvp() system call is particularly well suited for this. Its first argument should be the name of the executable you want to run. This version of exec() automatically looks in all the directories in your PATH variable to find an executable with the given name, so it should be able to find all the standard shell commands based on just the executable name. The second argument to execvp() is an array of char pointers, like the one you build when you parse the user command into words. You'll need to add a NULL pointer to the end of this array. That's how execvp() can tell how many command-line arguments there are.

The following figure shows how you will run external command. At the start, we're pretending the user enters "ls -l /etc" as the first command. The shell runs this by making a child and having it run the `ls` command via execvp(). The parent process waits for the child to finish, then prompts the user for another command. If the user enters a command like "sleep 5", the shell makes another child to run this command and then waits for that child to finish.



Invalid Commands

If execvp() fails to execute the command given by the user, the program should print the following error message to standard output and prompt the user for another command. Here, *cmd* is the name of the command they entered, the first word of the user's command.

Can't run command *cmd*

If the user enters invalid options for a command, you don't have to do anything special to detect this. After you run it with execvp(), each external command will automatically check its arguments and complain if there's anything wrong with them.

Design

Your solution should include the following functions. With these functions, I found that the rest of my implementation wasn't too complicated. Of course, you can add other functions if you want to. Just

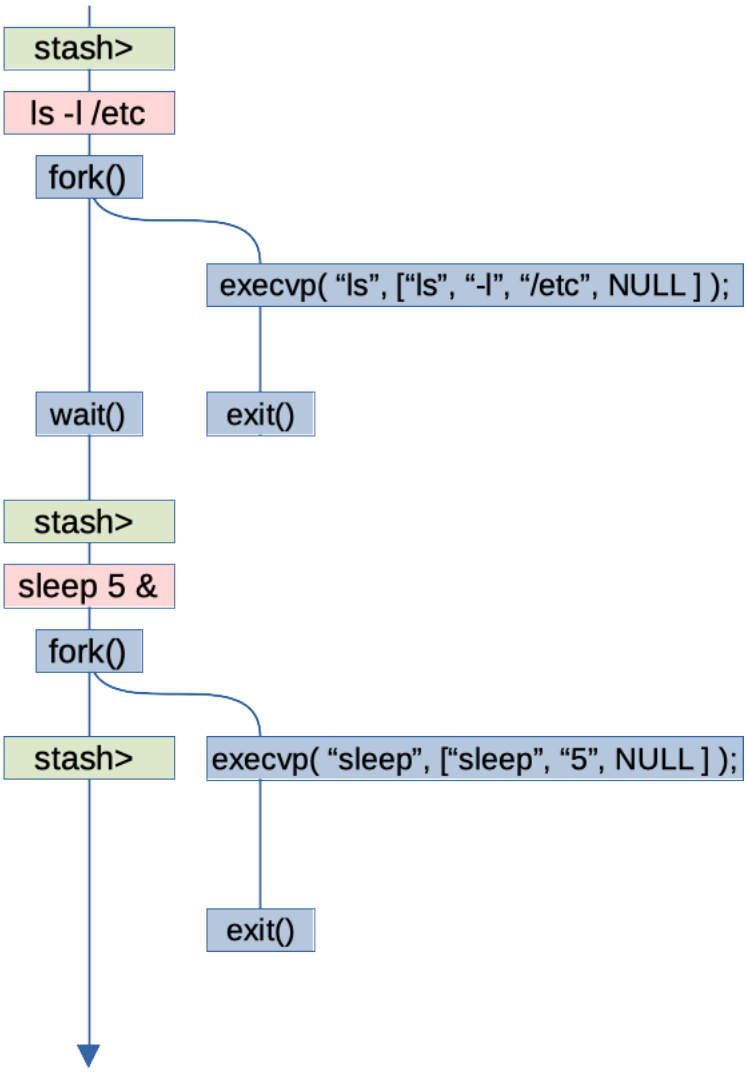
be sure to implement and use at least the following:

- `int parseCommand( char *line, char *words[] )`
  This function takes a user command (line) as input. As described above it breaks the line into individual words, adds null termination between the words so each word is a separate string, and it fills in a pointer in the words array to point to the start of each word. It returns the number of words found in the given line. The words array should be at least 513 elements in length, so it has room for the largest possible number of words that could fit in a 1024-character input line.
- `void runExit( char *words[], int count )`
  This function performs the built-in exit command. The words array is the list of pointers to words in the user's command and count is the number of words in the array.
- `void runCd( char *words[], int count )`
  This function performs the built-in cd command. As with runExit(), the parameters give the words in the command entered by the user.
- `void runCommand( char *words[], int count )`
  This function runs a (non-built-in) command by creating a child process and having it call execvp() to run the given command.

Extra Credit

For **8 points of extra credit**, you can implement support for running commands in the background. The user can run commands in the background by putting an ampersand at the end. The ampersand will be the last word of the command; you don't have to look for an ampersand that's part of some other word on the command line.

Running a command in the background means that the shell doesn't wait for the command to finish before prompting the user for another command. It runs the command then starts trying to read another command right away. The following shows the execution of the same commands as above, but with the sleep command run in the background. In this case, the shell prompts the user for another command right after starting the sleep command; it doesn't wait for sleep to finish.



When you start a command in the background, you will print out the process ID of the child inside square brackets. This gives the user a way to keep up with background commands as they are started. So, running a sleep command in the background might look like the following, if the sleep command had process ID 12345.

```
stash> sleep 5 &
[12345]
```

Background commands will eventually finish. If you do the extra credit, your shell should report any backgrounded commands that have finished after the user runs the next (non-builtin) command. (This is the first command that the user runs **after** the backgrounded command finishes, not necessarily the next command the user enters.) The output for a finished background command should give the process ID of the background command that finished, followed by the word “done”, all inside square brackets. For example, if the user runs the `date` command after the sleep command above finishes, you might get the following (if the sleep command had process ID 12345).

```
stash> date
[12345 done]
Tue Jan 11 22:12:08 EST 2023
```

Sample Execution

The following shows an execution of our shell. I've included comments below to explain what this example is doing (the lines starting with #). Don't actually type these when you're trying out your program (our shell isn't expected to be able to ignore comments).

```
# Start up our shell program.
$ ./stash

# Run a ls command (an external command).  I did this in the same
# directory where I worked on homework 0, so you can see some of
# of the other files from this assignment.
stash> ls
exclude.c  input-1.txt input-2.txt  Makefile  stash  stash.c

# Try out a shell command with some arguments.
stash> echo this is a test with multiple arguments
this is a test with multiple arguments

# Make a directory and try out the cd command.
stash> mkdir test
stash> cp input-1.txt test
stash> cd test

# Make sure we're in the new directory we just changed to.  I'm doing
# this in my account on an EOS linux machine.  You'll see a different
# pathname when you run this command.
stash> pwd
/mnt/ncsudrive/s/sjiao2/MyCSC246/hw0/test

# Have a look at the file we copied to this directory.
stash> cat input-1.txt
This is a sample
input file for the
exclude program.
You can give a
command-line
argument to tell it
what line to omit
when copying to the
output.

# Try out a blank command.  It should be ignored and we should
# get another prompt.
stash>

# Try out some invalid built-in commands.
stash> cd
Invalid command
stash> cd too many args
Invalid command
stash> exit bad-argument
```

```
Invalid command

# Try running a comamnd that doesn't exist.  The execvp()
# call should fail here.
stash> command-that-does-not-exist
Can't run command command-that-does-not-exist

# try out the exit command.
stash> exit 200

# Back in the regular shell, make sure we got the exit status
# we specified.
$ echo $?
200
```

Submission

When you're done, submit your source file, **stash.c**, under the assignment named HW0_Q4 on Gradescope. You don't need to submit any additional files for this problem, and don't put your source file in an archive or a subdirectory. Just submit the file, `stash.c`, directly to the HW0_Q4 submission locker.