CS 421 **Program 2** **Spring 2023**

Due: Wednesday, March 1.

Objectives:

Build your own shell.

Background:

This project consists of modifying a C program which serves as a shell interface that accepts user commands and then executes each command in a separate process. A shell interface provides the user a prompt after which the next command is entered. The example below illustrates the prompt sh> and user's next command: "cat prog.c". This command displays the contents of the file prog.c on the terminal using the UNIX cat command.

```
sh> cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (i.e. cat prog.c), and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to what is illustrated in Figure 3.8 (10th edition) or Figure 3.10 (9th edition) in the text. However, UNIX shells typically also allow the child process to run in the background (concurrently with the parent) as well by specifying the ampersand (&) at the end of the command. By rewriting the above command as:

```
sh> cat prog.c &
```

The parent and child processes now run concurrently. The separate child process is created using the fork( ) system call and the user's command is executed by using one of the system calls in the exec( ) family (as described in the text, Section 3.3.1).

A Simple Shell

A C program that provides the basic operation of a command line shell is supplied in Code I. This program is composed of two functions: main( ) and setup( ). The setup( ) function reads in the user's next command (which can be up to 80 characters), and then parses it into separate tokens that are used to fill the argument vector for the command to be executed. (If the command is to be run in the background, it will end with '& ', and setup( ) will update the parameter "background" so the main( ) function can act accordingly. This program is terminated when the user enters 'exit'.

The main( ) function presents the prompt COMMAND-> and then invokes setup( ) , which waits for the users to enter a command. The contents of the command entered by the user are loaded into the args array. For example, if the user enters “ls –l” at the COMMAND-> prompt, args[0] becomes equal to the string “ls” and args[1] is set to the string to “-l”. (By “string”, we mean a null-terminated, C-style string variable.)

```
#include <stdio.h>
#include <unistd.h>
#define MAX_LINE 80
/* setup( ) reads in the next command line, separating it into distinct tokens using whitespace as
delimiters.  setup( ) modifies the args parameter so that it holds pointers to the null-terminated
strings that are the tokens in the most recent user command line as well as a NULL pointer,
indicating the end of the argument list, which comes after the string pointers that have been
assigned to args. */

void setup (char inputBuffer[], char *args[], int *background)
{
/* full source code posted separately */
}

int main (void)
{
  char inputBuffer[MAX_LINE] ;   /* buffer to hold command entered */
  int background; /* equals 1 if a command is followed by ‘& ’ */
  char *args [MAX_LINE/2 + 1];  /* command line arguments */

  while (1) {
     background = 0 ;
     printf (“ COMMAND ->”) ;
     setup (inputBuffer, args, &background) ;
     /** the steps are:
    (0) If command is built-in, print command results to terminal.  If command is exit, print
statistics and exit()
    (1) fork a child process using fork( )
    (2) the child process will invoke execvp( )
    (3) if background == 1, the parent will wait,
    otherwise it will invoke the setup ( ) function again. */
  }
}
```

Code I. Outline of simple shell.

This project is organized into two parts: (1) creating the child process and executing the command in the child, and (2) modifying the shell to add built-in commands.

Creating a Child Process

The first part of this project is to modify the main( ) function in Code I so that upon returning from setup( ), a child process is forked and executes the command specified by the user

As noted above, the setup( ) function loads the contents of the args array with the command specified by the user. This args array will be passed to the execvp( ) function, which has the following interface:
execvp (char *command, char *params[ ]) ;
where "command" represents the command to be performed and "params" stores the parameters to this command. For this project, the execvp( ) function should be invoked as execvp(args[0], args); be sure to check the value of background to determine if the parent process is to wait for the child to exit or not.

Implementing built-in commands
The next task is to modify the program in Code I so that it provides several built-in commands. Unix shells provide commands to support job control. This allows the user to suspend jobs, start them back up, terminate jobs, and move them between the foreground and background. Bash supports these commands:

- jobs: List the running and stopped background jobs.
- bg <job>: Change a stopped job to a running background job.
- fg <job>: Change a stopped job to a running foreground job.
- kill <job>: Terminate a job.

The shell keeps a list of active jobs and uses signals to change their status. In Code I, we will need to do a string compare of the first argument in the args array to see if it is equal to any of our built-in commands. If so, our code will need to take the appropriate action. In this case, no child process will be forked off. We'll implement some command similar to the above.

Using Signals to Communicate with Processes
UNIX systems use signals to notify a process that a particular event has occurred. Signals may be either synchronous or asynchronous, depending upon the source and the reason for the event being signaled. Signals caused by the process receiving the signal are considered synchronous (ex. segmentation violation.) Signals sent by another process are considered asynchronous. We'll mostly be using asynchronous signaling here since the shell will be sending signals to its children.

A process can send another process a signal using the kill() function.
int kill(pid_t pid, int sig);

Commonly used signals are:

SIGQUIT	Quit	<Control> <\>
SIGINT	Interrupt	<Control><c>
SIGTSTP	Stop	<Control><z>
SIGCONT	Continue	Can't produce on keyboard
SIGCHLD	Child complete	Can't produce on keyboard
SIGKILL	Kill	Can't produce on keyboard

https://www.gnu.org/software/libc/manual/html_node/Job-Control-Signals.html

This page describes the transitions a bit more: https://www.baeldung.com/linux/process-states

Once a signal has been generated by the occurrence of a certain event (e.g., kill function called, illegal memory access, user entering <Control><\>, etc.), the signal is delivered to a process where it must be handled. A process receiving a signal may handle it by one of the following techniques:

- Ignoring the signal
- Using the default signal handler or
- Providing a separate signal-handling function.

Signals may be handled by first setting certain fields in the C structure provided to the sigaction( ) function. Signals are defined in the include file /usr/include/sys/signal.h. For example, the signal SIGQUIT represents the signal for aborting a program with the control sequence <Ctrl><\>. The default signal handler for SIGQUIT is to abort execution of the program.

Alternatively, a program may choose to set up its own signal-handling function by setting the sa_handler field in struct sigaction to the name of the function which will handle the signal and then invoking the sigaction( ) function, passing it (1) the signal we are setting up a handler for, and (2) a pointer to struct sigaction.

In Code II we show a C program that uses the function handle_SIGINT( ) for handling the SIGINT signal. This function prints out the message "Caught Control-c" and then invokes the exit ( ) function to terminate the program. (We must use the write( ) function for performing output rather than the more common printf ( ) as the former is known as being signal-safe, indicating it can be called from inside a signal-handling function; Such guarantees cannot be made of printf ( ) .) This program will run in the while (1) loop until the user enters the sequence <Control><c>. When this occurs, the signal-handling function handle_SIGINT( ) is invoked.

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#define BUFFER_SIZE 50
char buffer[BUFFER_SIZE] ;

/*the signal handling function */
void handle_SIGQUIT( )
{
   write (STDOUT_FILENO, buffer,strlen (buffer) ) ;
   exit (0) ;
}

int main (int argc, char *argv[ ] )
{
   /* set up the signal handler */
   struct sigaction handler ;
   handler. sa_handler = handle_SIGQUIT ;
   sigaction (SIGQUIT, &handler , NULL);
```

```
    /*Generate the output message */
    strcpy(buffer, “Caught Control-backslash\n”) ;
    /*loop until we receive <Control><\> */
    while (1)
        ;
    return 0;
}
```
Code II. Signal-handling program

The signal-handling function should be declared above main() and, because control can be transferred to this function at any point, no parameters may be passed to this function. Therefore, any data that it must access in your program must be declared globally, i.e. at the top of the source file before your function declarations. Before returning from the signal-handling function, it should reissue the command prompt.

Details for our class:

A shell is a command line interpreter which often has some functionality built in, and spawns off processes to handle other functions. You will be writing a simple shell which spawns off processes to handle most commands. Your shell will also provide some job control functions via built-in functionality that you write. Your shell must:

1. When your shell starts up, it must print a greeting and its pid. You can use getpid() to get the pid. Example:

    "Welcome to kbshell. My pid is 26357."

2. The shell prompt must be "xxshell[n]:", where xx are your initials, and n is the number of prompts shown so far (starting at 1 and increasing). Example:

    kbshell[1]:

3. The shell must support the following built-in commands:

- stop <pid> - which stops a running background task by sending it the SIGSTOP signal.
- bg <pid> - which instructs a stopped process to resume running in the background by sending it the SIGCONT signal.
- fg <pid> - which instructs a stopped process to resume running in the foreground by sending it the SIGCONT signal and waiting for it to complete. waitpid() is the right function to use here. Print a message “Child process complete” when it completes.
- kill <pid> - which kills a stopped or background process by sending it the SIGKILL signal.
- exit - which prints a message “xxshell exiting” and exits.
- Ctrl-\ – Ctrl-\ would normally terminate a program to which it is sent. So, if you typed control-\ while running your shell, it would receive the signal and terminate. Instead, we

would like our shell to catch the signal and pass it on to the foreground process (if it exists). So, the shell would continue executing but the child would terminate. This is what bash does for the foreground child. The shell should print “Caught Ctrl-\” when it catches the signal.

4. For any other command, the shell should fork() a new process and execvp() the command. You are given quite a bit of code including the setup() function which tokenizes the user input and places the arguments into a vector (perfect for execvp). The code for setup() and a skeleton of main() is given in shell.c. Your shell must provide background and foreground options controlled by a "&" sign. If a user command ends with an "&", it should be run in the background. This means that the parent does not wait for the command to complete before issuing a new prompt. If there is no "&", then the parent does wait for the child to complete.

In either case, immediately after forking, the parent should print a message indicating the pid of the child and whether the child is running in the background or not (ex. [Child pid = xxxx, background = TRUE]). If the command statement ends with an "&", the shell must run the command in the background. In this case, the shell should immediately offer a new prompt after launching the previous command. If background operation is not indicated, the shell should waitpid() for the child to complete before printing "Child process complete" and offering a new prompt. (Note: you need to use waitpid() rather than wait() because wait() will return if **any** child is done not just the one you want to wait for.) Example:

```
kbshell[1]: time sleep 8 &
[Child pid = 26358, background = TRUE]
kbshell[2]: date

[Child pid = 26359, background = FALSE]          /* Note that the order of the next couple
statements depends on the scheduler */
Wed Feb 11 22:48:42 PDT 2021
Child process complete
kbshell[3]:
real    0m8.018s                                 /* Note this is the child output which is
printed after the parent has completed another command */
user    0m0.001s
sys     0m0.001s
kbshell[4]: time sleep 8
[Child pid = 26360, background = FALSE]
real    0m8.018s                                 /* This is the child output before the parent
prints another prompt. */
user    0m0.001s
sys     0m0.001s
Child process complete
kbshell[5]:
```

Note that, in the foreground case, since we don't know whether the child or parent will execute first after the fork, the pid message may come before or after child's output.

Note: The Linux OS gets displeased when the read() call in setup() is interrupted in order to print out messages in the signal handlers (meaning that the OS will kill your process). You can fix this by adding the following line before the call to sigaction() when initializing your signal handler:
handler.sa_flags = SA_RESTART;
This will cause the read() system call to restart after the interrupt.
Here are a couple links which explain this issue:
http://beej.us/guide/bgipc/output/html/multipage/signals.html
http://www.gnu.org/s/libc/manual/html_node/Flags-for-Sigaction.html

---

Notes:
If you are unfamiliar with C function calls, here is an excellent link to on-line man pages, specifically for Linux. http://www.die.net/doc/linux/man/

If you haven't developed code in the UNIX environment before, here is a good link:
http://www.ece.utexas.edu/~bevans/talks/software_development/
and this one has some tutorials on the editors available:
http://www.cisl.ucar.edu/docs/access/tools.html

It's always a good idea to use a makefile. You put all your file dependencies in the file and type "make" on the command line to build the project. Here is an example called "makefile":

```
# This is a comment.  OS Program 2
# The 2nd line below must begin with a tab.
kbshell: kbshell.c
	gcc -Wall kbshell.c -o kbshell
```

If you are unfamiliar with C, try starting with the fork() example program on page 113 (10th edition) or page 118 (9th edition) of your book. Get that working and then add functionality.

When testing your background jobs, you can try the command " time sleep 10 &" which will cause the child to block for 10 seconds and then print out how long it slept, ensuring that the parent will run first. You can use just “sleep 100” or “sleep 100 &” if you don’t need any output.

It’s good practice to call wait() or waitpid() when you know you have some zombie child processes around to clean them up.

You may assume that no command will have more than 4 parameters.

Type “stty –a | grep intr” to see the signals available from the keyboard.

For whatever reason, the UNIX time utility prints somewhat different output if it is run in the background or the foreground. You can correct this by using one of the options (-p), or just ignore the differences. It's fine either way. ">time -p sleep 2" and ">time -p sleep 2 &" will both give the same output.

Requirements:
You must write your program in C and run it on a Linux machine.

Your program must be well-commented including function headers and explanations of complex code. Proper indentation is expected.

To hand in:

- Your commented code
- Your typescript file (described below)
- A paragraph in which you explain the meanings of the fields and values in the "ps a" output. Look at the first line of the "ps a" output, and or each field, explain what the field is meant to describe and why the value made sense for this run of your program

---

How to generate typescript file:

1. Log on to your Linux account.
2. Start up the script command. "script"
3. Run your shell. "xxshell"
4. Issue this command to your shell. "cp xxshell tempfile"
5. Issue this command to your shell. "ls -l tempfile"
6. Issue the time command in the foreground. "time sleep 5"
7. Issue the time command in the background. "time sleep 20 &"
8. Before the child completes, issue the date command to your shell. "date"
9. Wait for the child to complete before continuing.
10. Issue this command to your shell. "sleep 100 &"
11. Before the child completes, issue the stop command. "stop <pid> where pid is the pid of the child process.
12. Issue the ps command to inspect the state of the sleep process. "ps a"
13. Issue the bg command to restart the process in the background. "bg <pid>"
14. Issue the ps command to inspect the state of the sleep process. "ps a"
15. Before the child completes, issue the stop command. "stop <pid> where pid is the pid of the child process.
16. Issue the fg command to restart the process in the foreground. "fg <pid> Wait for it to complete.
17. Issue this command to your shell. "sleep 100 &"
18. Issue the ps command to inspect the state of the sleep process. "ps a"
19. Issue the kill command to kill the process. "kill <pid>"

20. Issue the ps command to inspect the state of the sleep process. “ps a”
21. Issue this command to your shell. “sleep 100”
22. Issue the ctrl-\ command to your shell. “ctrl-\”
23. Issue the ps command to inspect the state of the sleep process. “ps a”
24. Issue the exit command. "exit"
25. Exit from script program. "exit"

The resulting script file is called "typescript". Please run the cleanup utility script2txt on this file before submitting. I provided that utility with Program #1.

How to get started:

Do your development **incrementally**. So:

I suggest doing this project in parts. Each time you get one part done, save it off to the side so that you always have something to hand in. I would suggest these steps:

1. Download myshell.c and mysignalquit.c, upload to your Linux account, compile and run.
2. Add in the startup statement for your shell and the modified command prompt.
2. Modify myshell.c to fork off a child process for every command entered and wait for each child to complete before going on the next one. You can use the fork() example from your book.
3. Look at the background variable and wait only if command is run in foreground.
4. Add in the built-in commands (stop, bg, fg, kill, and exit.)
5. Add the code for the signal handler functionality from mysignalquit.c to myshell.c, just printing some statement when Ctrl-\ is pressed.
6. Modify the signal handler to pass the signal to the foreground process.
8. Collect your output.
