Programming Assignment #1: A Simple Shell

Due: Check My Courses

IMPORTANT: You are NOT allowed to include code from any other sources (except the sample code given in this handout). If someone else turns in a copy of your code, your assignment will be flagged as a copy as well. All students with the same copy will be reported to the disciplinary officer. You can discuss problems arising in the design and implementation without sharing code. The **code you submit** should be written by **you only**.

In this assignment, you are required to create a C program that implements a shell interface that accepts user commands and executes each command in a separate process. Your shell program provides a command prompt, where the user inputs a line of command. The shell is responsible for executing the command. The shell program assumes that the first string of the line gives the name of the executable file. The remaining strings in the line are considered arguments for the command. Consider the following example.

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

The **cat** is the command that is executed with **prog.c** as its argument. Using this command, the user displays the contents of the file **prog.c** on the display terminal. If the file **prog.c** is not present, the **cat** program displays an appropriate error message. The shell is not responsible for such error checking. In this case, the shell is relying on **cat** to report the error.

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 specified otherwise, the parent process waits for the child to exit before continuing. However, UNIX shells typically also allow the child process to run in the background – or concurrently – as well by specifying the ampersand (**&**) at the end of the command. By re-entering the above command as follows the parent and child processes can run concurrently.

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

Remember that parent is the shell process and child is the process that is running **cat**. Therefore, when the parent and child run concurrently because the command line ends with an &, we have the shell running before the cat completes. So, the shell can take the next input command from the user while **cat** is still running. As discussed in the lectures, the 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 (see **man exec** for more information).

Simple Shell

A C program that provides the basic operations of a command line shell is supplied below for illustration purposes. This program is composed of two functions: **main()** and **getcmd()**. The **getcmd()** function reads in the user's next command, 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 **getcmd()** will update the background parameter so the **main()** function can act accordingly. The program terminates when the user **enters <Control><D>** because **getcmd()** invokes **exit()**.

The **main()** calls **getcmd()**, which waits for the user 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]`** is set 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.)

This programming assignment is organized into three parts: (1) creating the child process and executing the command in the child, (2) signal handling feature, and (3) additional features.

Creating a Child Process

The first part of this programming assignment is to modify the **`main()`** function in the figure below so that upon returning from **`getcmd()`** a child process is forked and it executes the command specified by the user.

```
// DISCLAIMER: This code is given for illustration purposes only. It can contain bugs!
// You are given the permission to reuse portions of this code in your assignment.
//
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
//
// This code is given for illustration purposes. You need not include or follow this
// strictly. Feel free to write better or bug free code. This example code block does not
// worry about deallocating memory. You need to ensure memory is allocated and deallocated
// properly so that your shell works without leaking memory.
//
int getcmd(char *prompt, char *args[], int *background)
{
    int length, i = 0;
    char *token, *loc;
    char *line = NULL;
    size_t linecap = 0;

    printf("%s", prompt);
    length = getline(&line, &linecap, stdin);

    if (length <= 0) {
        exit(-1);
    }

    // Check if background is specified..
    if ((loc = index(line, '&')) != NULL) {
        *background = 1;
        *loc = ' ';
    } else
        *background = 0;

    while ((token = strsep(&line, " \t\n")) != NULL) {
        for (int j = 0; j < strlen(token); j++)
            if (token[j] <= 32)
                token[j] = '\0';
        if (strlen(token) > 0)
            args[i++] = token;
    }

    return i;
}

int main(void)
{
    char *args[20];
    int bg;
```

```
    while(1) {
        bg = 0;
        int cnt = getcmd("\n>>  ", args, &bg);

        /* the steps can be..:
        (1) fork a child process using fork()
        (2) the child process will invoke execvp()
        (3) if background is not specified, the parent will wait,
             otherwise parent starts the next command... */
    }
}
```

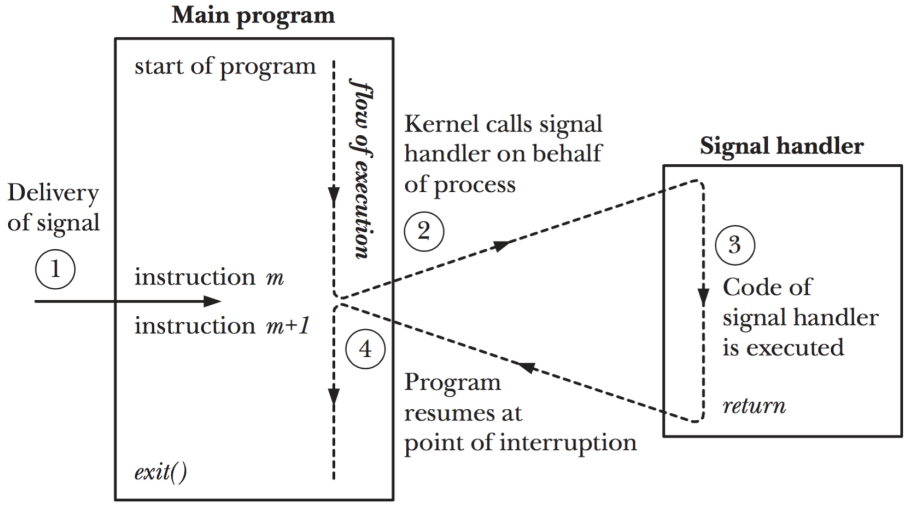
As noted above, the **`getcmd()`** function loads the contents of the args array with the command line given by the user. This args array will be passed to the **`execvp()`** function, which has the following interface:

`execvp(char *command, char *params[]);`

Where command represents the file to be executed and params store the parameters to be supplied to this command. Be sure to check the value of background to determine if the parent process should wait for the child to exit or not. You can use the **`waitpid()`** function to make the parent wait on the newly created child process. Check the man page for the actual usage of the **`waitpid()`** or similar functions that you can use.

Signal Handling Feature

The next task is to modify the above program so that it provides a *signal* handling feature. You can think of the signals as software interrupts. The signals are sent to a process because of external events such as keyboard presses (pressing the Ctrl + C key), external programs sending signals with specific values, or abnormal conditions created by the program (segmentation faults). A process is wired to act in a default way on the receipt of a given signal. You can change this default behaviour using a system call. The figure below shows how a program reacts to delivery of a signal assuming the program is programmed to anticipate the arrival of the signal.



You need to develop a signal feature that would do the following: (a) kill a program running inside the shell when Ctrl+C (SIGINT) is pressed in the keyboard and (b) ignore the Ctrl+Z (SIGTSTP) signal.

There are two ways to set up signal handlers. We would use the legacy interface because it is easy to understand. For portable programs, the approach is not recommended because it works differently on

different Unix/Linux implementations. The general approach is very simple. You create a function for handling a particular signal and hook up the function to act as the signal handler using the **`signal()`** system call. The following example program should give you the general idea.

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void
sigHandler(int sig) {
    printf("Hey! Caught signal %d\n", sig);
}


int  main() {

    if (signal(SIGINT, sigHandler) == SIG_ERR) {
        printf("ERROR! Could not bind the signal hander\n");
        exit(1);
    }

    while(1) {
        printf("......");
        sleep(1);
        printf("\n");
    }
}
```

Built-in Commands

A command is considered built-in, when all the functionality is completely built into the shell (i.e., without relying on an external program). The main feature of the shell discussed in the first two sections of this handout is about forking a child process to run external commands. Now, we want to implement some built-in commands.

`echo` command: This command takes a string with spaces in between as an argument. It prints the argument to the screen.

`cd` command: This command takes a single argument that is a string. It changes into that directory. You are basically calling the chdir() system call with the string that is passed in as the argument. You can optionally print the current directory (present working directory) if there are no arguments. You don't need to support additional features (e.g., those found in production shells like bash).

`pwd` command: This command takes no argument. It prints the present working directory.

`exit` command: This command takes no argument. It terminates the shell. It will terminate any jobs that are running in the background before terminating the shell.

`fg` command: This command takes an optional integer parameter as an argument. The background can contain many jobs you could have started with **`&`** at the end. The **`fg`** command should be called with a number (e.g., **`fg 3`**) and it will pick the job numbered 3 (assuming it exists) and put it in the foreground. The command **`jobs`** should list all the jobs that are running in the background at any given time. These are jobs that are put in the background by running the command with **`&`** as the last character in the

command line. Each line in the list shown by the **jobs** command should have a number identifier that can be used by the **fg** command to bring the job to the foreground (as explained above).

jobs command: This command takes no argument. It lists all the jobs that are running in the background.

Simple Output Redirection

The next feature to implement is the output redirection. If you type

```
ls > out.txt
```

The output of the **ls** command should be sent to the **out.txt** file. See the class notes on how to implement this feature.

Simple Command Piping

The last feature to implement is a simplified command piping. If you type

```
ls | wc -l
```

The output of the **ls** command should be sent to the **wc -l** command. One easy way of implementing this command piping is to write the output of ls to the disk and ask the second command to read the input from the disk. However, in this assignment and in the real system, you don't implement it through the file system. You implement command piping using an inter-process communication mechanism called pipes (anonymous). You can follow our discussion in the class and it should implement a command piping that should work with the above example.

Turn-in and Marking Scheme

The programming assignment should be submitted via My Courses. Other submissions (including email) are not acceptable. **Your programming assignment should compile and run in Linux. Otherwise, the TAs can refuse to grade them**. Here is the mark distribution for the different components of the assignment.

A simple shell that runs: shows prompt, runs commands, goes to the next one, does not crash for different inputs (e.g., empty strings)	30%
Signal feature	10%
Other built-in features	20%
Output redirection	15%
Command piping	15%
Memory leak problems	5%
Code quality and general documentation (make grading a pleasant exercise)	5%

Useful Information for the Assignment

You need to know how process management is performed in Linux/Unix to complete this assignment. Here is a brief overview of the important actions.

(a) Process creation: The **fork()** system call allows a process to create a new process (child of the creating process). The new process is an exact copy of the parent with a new process ID and its own process control block. The name "fork" comes from the idea that parent process is dividing to yield two copies of itself. The newly created child is initially running the exact same program as the parent – which is pretty useless. So we use the **execvp()** system call to change the program that is associated with the newly created child process.

(b) The **exit()** system call terminates a process and makes all resources available for subsequent reallocation by the kernel. The **exit(status)** provides status as an integer to denote the exiting condition. The parent could use **waitpid()** system call to retrieve the status returned by the child and also wait for it.

(c) The **pipe()** creation system call.

(d) File descriptor duplication system call (**dup()**).

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
                                Returns process ID of child, 0 (see text), or -1 on error
```

The return value and *status* arguments of *waitpid()* are the same as for *wait()*. (See Section 26.1.3 for an explanation of the value returned in *status*.) The *pid* argument enables the selection of the child to be waited for, as follows:

- If *pid* is greater than 0, wait for the child whose *process ID* equals *pid*.
- If *pid* equals 0, wait for any child in the *same process group as the caller* (parent). We describe process groups in Section 34.2.
- If *pid* is less than −1, wait for any child whose *process group* identifier equals the absolute value of *pid*.
- If *pid* equals −1, wait for *any* child. The call *wait(&status)* is equivalent to the call *waitpid(-1, &status, 0)*.