Programming Assignment #1: A Simple Shell

Due: Check My Courses

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 as 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) modifying the shell to allow a history 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.

```
#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 writer 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.

Creating a History Feature

The next task is to modify the above program so that it provides a *history* feature that allows the user access at least up to the 10 most recently entered commands. These commands will be numbered starting at 1 and will continue to grow larger even past 10, e.g. if the user has entered 35 commands, the 10 most recent commands should be numbered 26 to 35. With this list, the user can run any of the previous 10 commands by entering just the !command-number. For example, just typing !32 will execute the command that corresponds to entry 32. If there is no entry 32 in the history window, you print "no command found in history". Any command that is executed in this fashion should be echoed on the user's screen (i.e., you write out the command that is being executed) and the command is also placed in the history buffer as the next command. You could modify **`getcmd()`** as part of the history implementation.

Built-in Commands

The **`history`** command is a built-in command because the functionality is completely built into your shell. On the other hand, the process forking mechanism was used to execute outside commands. In addition to the **`history`** command, implement the **`cd`** (change directory), **`pwd`** (present working directory), and **`exit`** (leave shell) commands. The **`cd`** command could be implemented using the **`chdir()`** system call, the **`pwd`** could be implemented using the **`getcwd()`** library routine, and the exit command is necessary to quit the shell. Other built-in commands to be implemented include **`fg`** and **`jobs`**. The command **`jobs`** should list all the jobs that are running in the background at any given time. These are jobs that are put into the background by giving the command with **`&`** as the last one in the command line. Each line in the list provided by the jobs should have a number identifier that can be used by the **`fg`** command to bring the job to the foreground.

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 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%
History feature	10%
Other built-in features	20%
Output redirection	15%
Command piping	20%
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)*.