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 rest of the 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. Effectively, 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.

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. 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 rewriting the above command as follows the parent and child processes can run concurrently.

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

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 (see textbook or **`man exec`** for more information).

Simple Shell

A C program that provides the basic operations of a command line shell is supplied below. 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. The program terminates when the user **`enters <Control><D>`**; **`setup()`** invokes **`exit()`**.

The **`main()`** function presents the prompt **COMMAND->** and then invokes **`setup()`**, 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]`** 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.)

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 commands.

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 **`setup()`**, child process is forked and it executes the command specified by the user.

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

#define MAX_LINE 80 /* 80 chars per line, per command, should be enough. */

/**
 * setup() reads in the next command line, separating it into distinct tokens
 * using whitespace as delimiters. setup() sets the args parameter as a
 * null-terminated string.
 */

void setup(char inputBuffer[], char *args[],int *background)
{
    int length, /* # of characters in the command line */
        i,      /* loop index for accessing inputBuffer array */
        start,  /* index where beginning of next command parameter is */
        ct;     /* index of where to place the next parameter into args[] */

    ct = 0;

    /* read what the user enters on the command line */
    length = read(STDIN_FILENO, inputBuffer, MAX_LINE);

    start = -1;
    if (length == 0)
        exit(0);            /* ^d was entered, end of user command stream */
    if (length < 0){
        perror("error reading the command");
       exit(-1);           /* terminate with error code of -1 */
    }

    /* examine every character in the inputBuffer */
    for (i=0;i<length;i++) {
        switch (inputBuffer[i]){
          case ' ':
          case '\t' :               /* argument separators */
            if(start != -1){
                    args[ct] = &inputBuffer[start];    /* set up pointer */
                ct++;
            }
              inputBuffer[i] = '\0'; /* add a null char; make a C string */
            start = -1;
            break;

           case '\n':                 /* should be the final char examined */
            if (start != -1){
                    args[ct] = &inputBuffer[start];
                ct++;
            }
              inputBuffer[i] = '\0';
              args[ct] = NULL; /* no more arguments to this command */
            break;

          default :             /* some other character */
            if (start == -1)
                start = i;
              if (inputBuffer[i] == '&'){
                *background  = 1;
                    inputBuffer[i] = '\0';
            }
        }
      }
     args[ct] = NULL; /* just in case the input line was > 80 */
}

int main(void)
{
    char inputBuffer[MAX_LINE]; /* buffer to hold the command entered */
    int background;             /* equals 1 if a command is followed by '&' */
    char *args[MAX_LINE/+1];    /* command line (of 80) has max of 40 arguments */


    while (1){                  /* Program terminates normally inside setup */
        background = 0;
        printf(" COMMAND->\n");
        setup(inputBuffer,args,&background);       /* get next command */

       /* the steps are:
        (1) fork a child process using fork()
        (2) the child process will invoke execvp()
        (3) if background == 1, the parent will wait,
             otherwise returns to the setup() function. */
    }
}
```

As noted above, the **`setup()`** 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 is to wait for the child to exit or not.

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 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 **`r x`** where **`'x'`** is the first letter of that command. If more than one command starts with **`'x'`**, execute the most recent one. Also, the user should be able to run the most recent command again by just entering **`'r'`**. You can assume that only one space will separate the **`'r'`** and the first letter and that the letter will be followed by **`'\n'`**. Again, **`'r'`** alone will be immediately followed by the **`'\n'`** character if it is wished to execute the most recent command.

Any command that is executed in this fashion should be echoed on the user's screen and the command is also placed in the history buffer as the next command. (**`r x`** does not go into the history list; the actual command that is specifies does.)

If the user attempts to use this history facility to run a command and the command is detected to be *erroneous*, an error message should be given to the user and the command not entered into the history list, and the **`execvp()`** function should not be called. (It would be nice to know about improperly formed commands that are handed off to **`execvp()`** that appear to look valid and are not, and not include them in the history as well, but that is beyond the capabilities of this simple shell program.) You should also modify **`setup()`** so it returns an **`int`** signifying if has successfully created a valid **`args`** list or not, and the **`main()`** should be updated accordingly.

Signal Handling in the Shell

Here two signals should be handled. First one is the keyboard interrupt signal sent by pressing **`<Control><C>`** and the second one is the quit signal sent by **`<Control><Z>`**. The program should ignore the keyboard interrupt signal. That is the shell should not die when the user presses **`<Control><C>`**. The **`<Control><Z>`** should put the current command in the background. That is you should be able to send an executing command to the background by issuing the quit signal. This is slightly different from normal shell behaviour where the quit signal suspends the command and then you put the command in the background using the **`bg`** command.

Following is a brief overview of signals and their usage. 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 signalled. Once a signal has been generated by the occurrence of a certain event (e.g., division by zero, illegal memory access, user entering **`<Control> <C>,`** 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 struct sigaction and then passing this structure to the sigaction() function. Signals are defined in the include file **`/usr/include/sys/signal.h`**. For example, the signal SIGINT represents the signal for terminating a program with the control sequence **`<Control><C>.`** The default signal handler for SIGINT is to terminate the program.

Alternatively, a program may choose to set up its own signal-handling function by setting the **`sa_handler`** filed 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 the figure below 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()`** in invoked.

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

#define BUFFER_SIZE 50

static char buffer[BUFFER_SIZE];

/* the signal handler function */
void handle_SIGINT() {
      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_SIGINT;
        sigaction(SIGINT, &handler, NULL);

        strcpy(buffer,"Caught <ctrl><c>\n");

        /* wait for <control> <C> */
        while (1)
                ;

        return 0;
}
```

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 it 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.

Built-in Commands

The **`history`** command is a built-in command because the functionality is completely built into the 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) and **`exit`** (leave shell) commands. The **`cd`** command should be implemented using the **`chdir()`** system call and the exit command is necessary to quit the shell because **`<Control><C>`** does not work after you capture and ignore the signal! 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 pressing **`<Control><Z>`** or 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.

Turn-in and Marking Scheme

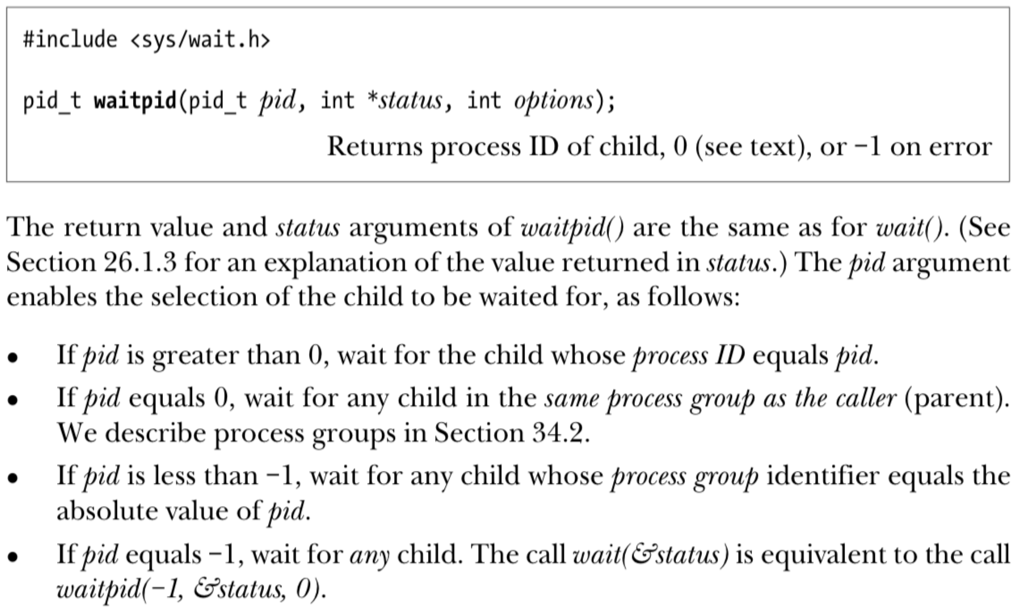
The programming assignment should be submitted via My Courses. Other submissions (including email) are not acceptable.

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.



```
#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)*.