ECSE 427 / COMP 310
Programming Assignment #1: Processes and threads
Due date: Check My Courses

This assignment has two parts. The first part consists of short answer questions that relate to the lectures on processes and threads, and the second part consists of a c program. The C program is a multipart question, and you are advised to start working on it as soon as possible.

Part 1: Short answer questions

P1Q1. Describe the difference between uniprogramming, multiprogramming and timesharing systems. (5%)

P1Q2. Suppose you have three programs, A, B, and C, that are either running or waiting on a resource. Each program runs for 2msec, waits on a resource for 10msec, and then runs for 4msec. Suppose that the first program starts running at t=0msec, at what time will all 3 programs complete in a uniprogramming ~~system?~~ What about a multiprogramming system? (5%)

P1Q3. Consider the following simple C program, which outputs a line of code on the screen. Modify the program using system calls in order to redirect the output to a file called redirect.txt. (5%)

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

int main()
{
    printf("A simple program output.");
    return 0;
}
```

Part 2: A simple shell

General requirements

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. In order to complete the assignment, you will need to implement the following built-in commands using system calls:

ls: list files in the current directory 

cd: change directory
cat: Example usage in shell:

```
sh> cat file1
```

cp: copy file. Example usage in shell:

```
sh> cp file1 file2
```

fg: foreground a background job
jobs: list background jobs
exit: quit the shell

The shell will allow commands to either be run in the foreground or the background by specifying the ampersand (&) at the end of the command.

In addition to the built-in commands, you are also to implement output redirection. As an example, if you type

```
sh> cat file1 > file2
```

The output of the **`cat file1`** command should be sent to the **`file2`** file. See the class notes on how to implement this feature.

Details that would guide you through the implementation are shown below.

Implementation guidelines

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.

```
#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... */
      }
}
```

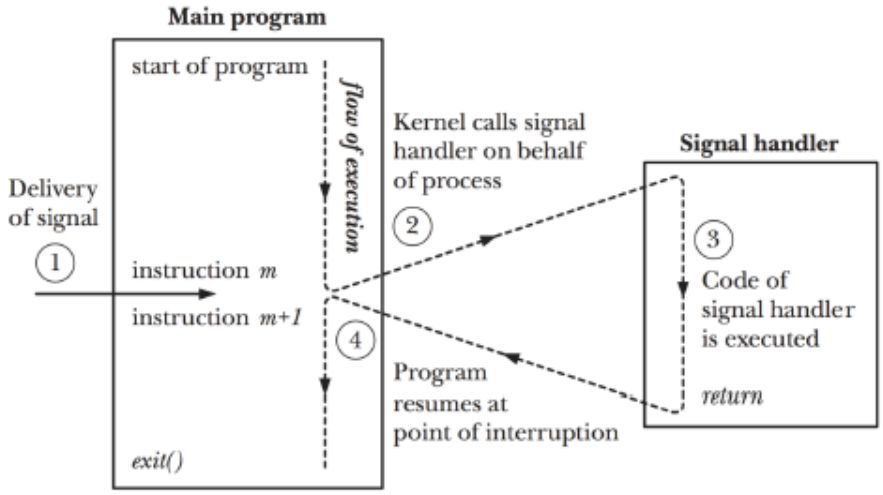
As noted above, the getcmd() function loads the contents of the args array with the command line given by the user. This args array will be passed to the execvp() function, which has the following interface:

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.

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 a 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. That is for SIGINT you kill the process that is running within the shell. For Ctrl+Z, you just ignore it so that your shell would not react to it using the default action.

There are two ways to setup 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 hookup the function to act as the signal handler using the signal() system call. The following example program should give you the general idea.

```
2  #include <signal.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6
7  static void
8  sigHandler(int sig) {
9      printf("Hey! Caught signal %d\n", sig);
10 }
11
12
13 int  main() {
14
15     if (signal(SIGINT, sigHandler) == SIG_ERR) {
16         printf("ERROR! Could not bind the signal hander\n");
17         exit(1);
18     }
19
20     while(1) {
21         printf(".....");
22         sleep(1);
23         printf("\n");
24     }
25 }
```

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. 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. 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 into 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 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 (as explained above).

Other useful information

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

Marking scheme for the programming part

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%
Memory leak problems	5%
Code quality and general documentation	5%

```
#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)*.