Lab 6

Motivation

In this lab you are going to learn and implement a job control, where a process is run under a job, and the job can run both in the background (e.g. when running `kate &` a new process is created belonging to a new job and the shell blocks) and in the foreground (e.g. when running `kate` a new process is created belonging to a new job and it doesn't block the shell, the shell keeps responding to incoming commands).

Running processes with a pipe between them, means that they all belong to the same process group. For Example, `ls|wc -l`, both the process that runs `ls` and the process that runs `wc -l` are considered a single job and hence have the same process group, though each of them has its own process id. They are all connected, one process' input depends on the other's output, and therefore we'd like a way to relate to them together and perform operations on them, like bringing them to the foreground/background, and this is exactly what job control offers. **For this reason we use the group id to wait for a job or change its state instead of using a process id, where it affects a single process instead of the whole job.**

Lab 6 tasks

Take the final version of your lab 5 and use it as a starting point for this lab. All the tasks in this lab must be added to your shell code from lab5.

Task 0a

Write a signal handler that prints the signal that the shell receives with a message saying it was ignored. The signals you need to address are: SIGQUIT, SIGTSTP, SIGCHLD. Use strsignal (see: `man strsignal`) to get the signal name. See signal(2) you will need it to set your handler to handle these signals.

Task 0b

Get yourself acquainted with job control, by running the commands as described by the following link: http://linuxg.net/how-to-manage-background-and-foreground-processes/

Task 0c

Note
We again revisit our shell program. Namely, the code you write in this task is added to the shell.

Having learned how to create a pipeline, we now wish to implement a pipeline in our own shell. In this task you will extend your shell's capabilities to support pipelines that consist of an unlimited number of processes. To achieve this goal, you will re-implement your pipeline `execute` function in recursive form.

`execute` will receive an array of two integers - the pipe's file descriptors. A left pipe and a right pipe. `execute` will follow these steps:

1. fork a child process from the shell

   <u>Child process</u>
   a. If the right pipe is non-NULL, redirect output to its write-end.
   b. If the left pipe is non-NULL, redirect input to its read end.
   c. Execute child (input/output redirection and execvp).

   <u>Shell</u>
   a. If the right pipe is non-NULL, close its write-end
   b. If the left pipe is non-NULL, close its read end and free the pipe
   c. If there's another command in the pipeline, create a new pipe
   d. Recursively call `execute` with the next command in the linked list, the old right pipe as the left pipe and the new pipe as the right pipe

A wrapper function will handle the initial call and waiting for the last child.

Notes:

- The line parser automatically generates a list of cmd_line structures to accommodate pipelines. For instance, when parsing the command **"ls | grep .c"**, two chained cmd_line structures are created, representing **ls** and **grep** respectively.
- Your shell must support all previous features, including input/output redirection. It is important to note that commands utilizing both I/O redirection and pipelines are indeed quite common (e.g. **"cat < in.txt | tail -n 2 > out.txt"**).

Extend your shell's capabilities to support an unlimited number of processes.
For example, if the command **"ls | grep .c | tail -n 2"** is entered, you are expected to create three processes and two pipes. Each successive pair of child processes will be connected by one pipe, similarly to the technique presented in Task 2. The first pipe allows the **ls** process to communicate with the **grep** process, and the second pipe allows **grep** to communicate with **tail**.

Remember:

- Test your shell with various pipelines. For example: **"ls | cat | grep .c | wc -l"**.
- Make sure there's only a single function responsible for execute (and not separate functions for, zero, single and multiple pipes).

Task 1 - Job Control

Before you start working on your implementation for Job Control tasks, make sure you read through this link in the reading material. It shows what job control is and how it is used by shell users.

Each process executed by the shell has a process group. Only one process group can run in the foreground, the rest run in the background or are suspended. Only the foreground process can receive signals (interrupts) and read from stdin. Managing these process groups and manipulating which of them runs in the background and which runs in the foreground is called job control, see reading material. In this task you will implement a simplified version of job control in the shell.

Your job control implementation must support pipes.

Another useful command that is a bit similar to jobs, but offers different functionality is the `history` command. The shell saves the history of shell command lines. The shell's history also allows you to run a command from the history by typing its number, instead of typing the whole command again. It can be useful for example when you need to run valgrind, but you don't remember all the flags. You can run: `history|grep valgrind` and it will print all the commands in the history that have the word valgrind in them.

Task 1a - Representation

We will need to store a list of all running/suspended jobs. To do that we use a linked list, where each node is a struct job:

```
typedef struct job{
    char *cmd;                          /* the whole command line as typed by the user, including input/output redirection and pipe
    int idx;                            /* index of current job (starting from 1) */
    pid_t pgid;                         /* process group id of the job*/
    int status;                         /* status of the job */
    struct termios *tmodes;             /* saved terminal modes */
    struct job *next;                   /* next job in chain */
} job;
```

The field *status* can have one of the following values:

```
#define DONE -1
#define RUNNING 1
#define SUSPENDED 0
```

We provide you an implementation of a job list linked list: job_control.h and job_control.c, which includes creating a job List and adding nodes to it, removing nodes from a job list, and removing a job list from memory. However some of the parts are missing. You need to implement the following functions:

- `job* initialize_job(char* cmd);` : Receive a cmd (command line), initialize job fields (to NULLs, 0s etc.), and allocate memory for the new job and its fields: tmodes (will be used to save the shell attributes) and cmd.
- `void free_job(job* job_to_remove);` : free all memory allocate for the job, include the job struct itself.
- `job* find_job_by_index(job * job_list, int idx);` : Receive a job list and an index, and return the job that has the given index.
- `void update_job_list(job **job_list, int remove_done_jobs);` This function is used to update the status of jobs running in the background to DONE. For each job, check if it is done by running waitpid and checking its return status (by using WIFEXITED and WIFSIGNALED, see man for waitpid). Update the job's status to done if needed. waitpid in this case must not block (see WNOHANG in the man page) and it should return immediately. If waitpid fails (returns -1) then there are no processes with the given process group id. if remove_done_jobs is set to 1 (TRUE) then DONE jobs are printed in the same format as in print_jobs and are then removed from the job list.

Feel free to change signatures if needed.

Task 1b - Adding Jobs

In this task, we will support adding jobs to the list, and printing them:

- First, add any new job to the job list and set its status to `RUNNING`, whenever a user enters a new command to run a program.
- Now, implement the `jobs` shell command: whenever the user enters this command, call the print_jobs function. Now test this as mentioned below.
- Finally, add code that frees the job list, before exiting the shell.

Task 1c - Initialization

To keep your shell running at all times, we are going to change the handling of signals in the shell, and set back the handlers to default in the child . In the shell, you'll use your signal handler from task 0, instead of the default handler. To be able to move jobs from running in the foreground to running in the background and vise versa, we need to set the process group id of each process, follow the steps under mandatory requirements.

Steps:

- Shell initialization: **At the beginning of the program** (at the beginning of the main!)
  - Ignore the following signals: SIGTTIN, SIGTTOU, SIGTSTP, so they can reach the foreground child process rather than the shell.
  - Use your signal handler from task0b to handle the following signals: SIGQUIT, SIGCHLD (We're not including SIGINT here so you can kill the shell with ^C if there's a bug somewhere)
  - Set the process group of the shell to its process id (getpid).
  - Save default terminal attributes to restore them back when a process running in the foreground ends.
- New processes: After each fork
  - In the child: Set the signal handlers back to default.
  - In both the child and the parent (to avoid a racing condition): Set the process **group id** (pgid) of the new process to be the same as the child process id (first child id, if there's a pipe), and save the group id in the job.

At this point, when your shell waits (if line->blocking is set), it should wait on the group instead of a single process (using `waitpid(-<group_id>, &status, options)`). This should be done in a loop, and the loop should continue as long as there are running children (using WNOHANG and checking for -1 return value).

Mandatory Requirements

- Signals: To set signals back to default use the command signal, see signal(2).
  - Relevant signal handler: SIG_DFL, SIG_IGN, and your signal handler from task0b.
- Process groups: use the following commands to set and get process groups: setpgid(2), getpgid (2).

- Use tcgetattr with STDIN_FILENO as fd, to save the shell terminal attributes.
- **Your job control implementation must support pipes.**

Test your code using the following scenario, in your shell:

```
$>test1&
$>Start of test1
test2&
$>Start of test2
test3&
$>Start of test3
test3|test2|test1&
$>Start of test1
jobs
[1]      Running                 test1&
[2]      Running                 test2&
[3]      Running                 test3&
[4]      Running                 test3|test2|test1&
$>End of test3
End of test2
End of test1
End of test1
jobs
[1]      Done                 test1&
[2]      Done                 test2&
[3]      Done                 test3&
[4]      Done                 test3|test2|test1&
$>jobs
$>
$>ls|cat
[result of ls|cat]
$>quit
```

The order of printing will not necessarily be as mentioned above. Download test1, test2, test3, and use the above scenario to test your code. Each of the files prints a start message, sleeps for awhile (30, 20, 10 seconds respectively), and then prints an end message. Obviously, you need to give these executable files execute permission before you try to run them…

Task 1d - Run in the foreground

Running jobs in the foreground is done in 2 cases:

1. If a job is run without & (non-blocking).
2. If the command fg is run.

Add support to running jobs in the foreground and to the `fg <job_number>` command that receives a job index, and runs the job in the foreground. Use find_job_by_index to get the job with the given index, and implement a function called run_job_in_foreground:

```
void run_job_in_foreground (job** job_list, job *j, int cont, struct termios* shell_tmodes, pid_t shell_pgid);
```

- Receive the job list, a pointer to the job with the given index, the shell process group, and the shell's saved attributes, see task2c.
- Check if the job is done, by running waitpid and checking its return status. waitpid in this case must **not block** (see WNOHANG in the man page). It should return immediately. If it fails (returns -1) then there are no processes with the given process **group id**. If that is the case, then it prints a Done message in the same format as in print_jobs and remove the job from the job list.
- If the job has not finished yet, then put it in the foreground using: `tcsetpgrp (STDIN_FILENO, <job pgid>);`.
- if cont is 1 and the job's status was SUSPENDED, then:
  1. set the attributes of the terminal to that of the job's using: `tcsetattr (STDIN_FILENO, TCSADRAIN, <job tmodes>);`.
  2. Use kill, see man 2 kill, to send SIGCONT signal to the process **group** of the job.
- Wait for the job to change status using waitpid (need to block). Change the status of the job to SUSPENDED if the process **group** receives: a SIGTSTP (ctrl-z) - see WUNTRACED and WIFSTOPPED in the man page of waitpid.
  If it receives a SIGINT (ctrl-c), change the job status to DONE.

  Now the child process is running in the foreground and the shell is waiting for it to complete (or be stopped). Once the shell returns from `waitpid`:

- Put the shell back in the foreground.
- Save the terminal attributes in the job tmodes.
- Restore the shell's terminal attributes using the shell tmodes, which were saved during initialization. This is done to prevent leaving the shell in an unstable mode. For example if one of the jobs changes the tmodes of the terminal (the text reader used by `man` for examples does this).
- Check for status update of jobs that are running in the background using your update_job_list() function.

You need to call this function, when the fg command is used and when a command is executed in the foreground (in the function `execute`), in a blocking mode (if a command is run without &, instead of the parent waiting for the job to end).
Hint: You'll probably need to refactor `execute` a bit for this to work.

Example:

```
$> test1
Start of test1
^Z
$> jobs
[1]     Suspended               test1
$> fg 1
End of test1
[1]     Done                test1
$>quit
```

Task 1e - Run in the background

Add support to running jobs in the background and to the `bg <job_number>` command that receives a job index, and sends a `SIGCONT` to the process group.

Steps

- Implement a function called run_job_in_background that receives a job (that has the given index using find_job_by_index), sets its status to RUNNING, and sends it a SIGCONT. `void run_job_in_background (job *j, int cont);`
- Use kill, see man 2 kill, to send SIGCONT signal to the process group of the job.

You need to call this function, when the bg command is used and when a command is executed in the background (in the function `execute`), in a non-blocking mode (if a command is run with &).

Example

```
$> test1
Start of test1
^Z
$> test2
Start of test2
^Z
$> test3
Start of test3
^Z
$> test3 | test2 | test1
Start of test3
Start of test2
Start of test1
^Z
$> jobs
[1]     Suspended               test1
[2]     Suspended               test2
[3]     Suspended               test3
[4]     Suspended               test3 | test2 | test1
$> fg 3
End of test3
[3]    Done               test3
$> jobs
[1]     Suspended               test1
[2]     Suspended               test2
$> bg 2
$> End of test2
fg 4
End of test3
End of test2
End of test1
/* Note that the shell only returns once all the children in group 4 have existed (as opposed to waiting for any child that exited) *
$>fg 1
End of test1
[1]     Done               test1
$>jobs
[2]     Done               test2
$>jobs
$> quit
```

Deliverables:

Task 1a,1b, 1c and 1d must be completed during the regular lab. Task 1e may be done in a completion lab. The deliverables must be submitted until the end of the lab session.
You must submit source files for tasks 1d and 1e and a makefile that compile them. The submitted zip file should be (+ represents a folder and '-' represents a file):
+ task1d
  - makefile
  - job_control.c
  - job_control.h
  - myshell.c
+ task1e
  - makefile
  - job_control.c
  - job_control.h
  - myshell.c