Lab C

Lab C builds on top of the code infrastructure of the "shell" from lab 2. Naturally, you are expected to use the code you wrote for lab 2 in this lab, and extend it. This do-at-home lab may be done in pairs.

Motivation

In this lab you will enrich the set of capabilities of your shell by adding job control, pipes, and history. Note that all parts below, except for part 1, are features that should be added to your shell as one program (in however many C functions you wish), that supports all these features.

Part 0: preparation

History

Check out the "history" mechanism in the Linux shell. For example, see this link and try it out in a Linux shell.

Reexamine your command interpreter (shell) code from lab 2, and if not done so before, reorganize it to be as modular and extensible as possible,

Lab C Parts

Part 1: An Exercise in Pipes

Note
This part is independent of the shell, and a preparation for implementing a pipe command in the shell. You should not use the LineParser functions in this task, nor read any command lines. However, you need to declare an array of "strings" containing all of the arguments and ending with 0 to pass to execvp() just like the one returned by parseCmdLines().

Here we wish to explore the implementation of a pipeline. In order to achieve such a pipeline, one has to create a pipe and properly redirect the standard output and standard input of processes.
Please refer to the 'Introduction to Pipelines' section in the reading material.

Your task: Write a short program called **mypipeline** which creates a pipeline of 2

child processes. Essentially, you will implement the shell command line **"ls -l | tail -n 2"**.
(A question: what does "ls -l" do, what does "tail -n 2" do, and what should their combination produce?)

Follow the given steps as closely as possible to avoid synchronization problems:

1. Create a pipe.
2. Fork a first child process (child1).
3. In the child1 process:
    1. Close the standard output.
    2. Duplicate the write-end of the pipe using **dup** (see man).
    3. Close the file descriptor that was duplicated.
    4. Execute "ls -l".
4. **In the parent process: Close the write end of the pipe.**
5. Fork a second child process (child2).
6. In the child2 process:
    1. Close the standard input.
    2. Duplicate the read-end of the pipe using **dup**.
    3. Close the file descriptor that was duplicated.
    4. Execute "tail -n 2".
7. **In the parent process: Close the read end of the pipe.**
8. Now wait for the child processes to terminate, in the same order of their execution.

After implementing the above code, debug and test it as follows: Compile and run the code and make sure it does what it is supposed to do, by adding debug messages printed to stderr as follows:

- In the parent process:
    - Before forking, "(parent_process>forking…)"
    - After forking, "(parent_process>created process with id:)"
    - Before closing the write end of the pipe, "(parent_process>closing the write end of the pipe…)"
    - Before closing the read end of the pipe, "(parent_process>closing the read end of the pipe…)"
    - Before waiting for child processes to terminate, "(parent_process>waiting for child processes to terminate…)"
    - Before exiting, "(parent_process>exiting…)"
- In the 1st child process:

        - "(child1>redirecting stdout to the write end of the pipe…)"
        - "(child1>going to execute cmd: …)"
    - In the 2nd child process:
        - "(child2>redirecting stdin to the read end of the pipe…)"
        - "(child2>going to execute cmd: …)"

3. How does the following affect your program:
    1. Comment out step 4 in your code (i.e. on the parent process:**do not** Close the write end of the pipe). Compile and run your code. (Also: see "man 7 pipe")
    2. Undo the change from the last step. Comment out step 7 in your code. Compile and run your code.
    3. Undo the change from the last step. Comment out step 4 and step 8 in your code. Compile and run your code.

Part 2: Implementing a Pipe in the Shell

Having learned how to create a pipe between 2 processes/programs in Part 1, we now wish to implement a pipeline **inside** our own shell. In this part you will extend your shell's capabilities to support pipelines that consist of just one pipe and 2 child processes. That is, support a command line with one pipe between 2 processes resulting from running executable files mentioned in the command line. The scheme uses basically the same mechanism as in part 1, except that now the program to be executed in each child process is determined by the command line.

Your shell must be able now to run commands like: ls|wc -l which basically counts the number of files/directories under the current working dir. The most important thing to remember about pipes is that the write-end of the pipe needs to be closed in all processes, otherwise the read-end of the pipe will not receive EOF, unless the main process terminates.

Notes:

- The line parser automatically generates a list of cmdLine structures to accommodate pipelines. For instance, when parsing the command **"ls | grep .c"**, two chained cmdLine structures are created,
  representing **ls** and **grep** respectively.

- Your shell must still support all previous features, including input/output redirection from lab 2. Obviously, it makes no sense to redirect the output of the left--hand-side process (as then nothing goes into the pipe), and this should be considered an error, and likewise redirecting the input of the right-hand-side process is an error (as then the pipe output is hanging). In such cases, print an error message to stderr without generating any new processes. 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"**).
- As in previous tasks, you must keep your program free of memory leaks.

Part 3: Process Manager

Every program you run using the shell runs as a process. You can get a list of the running processes using the `ps` program (see: `man 1 ps` and `man 2 ps`). In this task we are going to add to your shell an internal "process manager" to manage the process we run in our shell (everything you fork). The process manager will provide 4 operations:

- `procs` - prints current processes including sleeping, running, and "freshly" terminated processes.
- `wake <process id>` - wakes up a sleeping process.
- `suspend <process id>` - suspends a running process.
- `kill <process id>` - terminates a running/sleeping process (was implemented in lab 2).

Part 3a - Process List

In this part we will create and print a list of all processes that have been forked by your shell.

Representation

Create a linked list to store information about running/suspended processes. Each node in the list is a struct process:

```
typedef struct process{
```

```
        cmdLine* cmd;                           /* the parsed command line*/
        pid_t pid;                                /* the process id that is
running the command*/
        int status;                             /* status of the process:
RUNNING/SUSPENDED/TERMINATED */
        struct process *next;                     /* next process in chain */
    } process;
```

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

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

Implementation

Implement the following functions that create and print the process list:

- `void addProcess(process** process_list, cmdLine* cmd, pid_t pid);`: Receive a process list (process_list), a command (cmd), and the process id (pid) of the process running the command. Note that process_list is a pointer to a pointer so that we can insert at the beginning of the list if we wish.
- `void printProcessList(process** process_list);`: print the processes.
- Add support for the command `procs` to the shell which prints processes using printProcessList()in the following format:
  `<index in process list> <process id> <process status> <the command together with its arguments>`

Example:

```
#> sleep 3  # foreground, takes 3 seconds until we get prompt back
#> procs
PID          Command      STATUS
14952        sleep        Terminated
#>
#> sleep 5& # background, we get prompt back immediately
#> procs
PID          Command      STATUS
14962        sleep        Running
#> # Wait for the process to finish
#>
#> procs
PID          Command      STATUS
14962        sleep        Terminated
```

Part 3b - Updating the Process List

Implement the following to add some functionality to your process list:

- `void freeProcessList(process* process_list);`: free all memory allocated for the process list.
- `void updateProcessList(process **process_list);`: go over the process list, and for each process check if it is done, you can use waitpid with the option WNOHANG. WNOHANG does not block the calling process, the process returns from the call to waitpid immediately. If no process with the given process id exists, then waitpid returns -1.
  In order to learn if a process was stopped (SIGTSTP), resumed (SIGCONT) or terminated (SIGINT), It's highly essential you read and understand how to use waitpid(2) before implementing this function
- `void updateProcessStatus(process* process_list, int pid, int status)`: find the process with the given id in the process_list and change its status to the received status.
- update `void printProcessList(process** process_list);`:
  - Run updateProcessList() at the beginning of the function.
  - If a process was "freshly" terminated, delete it after printing it (meaning print the list with the updated status, then delete the dead processes).

Part 3c - Manipulating the Processes

In this part you add to your shell process manipulation commands, one of the following (some you already implemented in lab 2):

- `suspend <process id>` - suspends a running process. Send SIGTSTP to the respective process. This is similar to typing CTRL-Z in the shell when running the process.
- `kill <process id>` - terminates a running/sleeping process. Send SIGINT to the respective process. This is similar to typing CTRL-C in the shell when running a process.
- `wake <process id>` - wakes up a sleeping process. Send SIGCONT to the respective process. This is similar to typing `fg` in a standard shell, right after typing CTRL-Z.

Use kill(), see man 2 kill, to send the relevant signal to the given process id. Check if kill() succeeded and print an appropriate message. Remember to update the status

of the process in the process_list.

Test your shell using your `looper` code from task0b in the following scenario:

```
    #> ./looper&
    #> ./looper&
    #> ./looper&
    #> procs
    PID         Command     STATUS
    18170       ./looper    Running
    18171       ./looper    Running
    18174       ./looper    Running
    #> kill 18170
    #> Looper handling SIGINT       # Message from the child process

    #> suspend 18174
    #> Looper handling SIGTSTP      # Message from the child process
    procs
    PID         Command     STATUS
    18170       ./looper    Terminated
    18171       ./looper    Running
    18174       ./looper    Suspended
    #> wake 18174
    #> Looper handling SIGCONT      # Message from the child process

    #> wake 18171   # What will happen to the process? (it is already
running)
    #> Looper handling SIGCONT      # Message from the child process
    procs
    PID         Command     STATUS
    18171       ./looper    Running
    18174       ./looper    Running
```

Part 4: Adding the History Mechanism

Here you will add a history mechanism to your shell. The history mechanism works as follows. Your shell should keep HISTLEN previous command lines in a queue, where HISTLEN is a constant with a value of 20 as a default. The history list is maintained in an array of size HISTLEN of pointers to (copies of) previous commands. Note that you need to allocate space for these copies. Also note that you should keep the UNPARSED command lines in the history list, and NOT the parsed version.

When a new command line is entered after the history list is full (already has HISTLEN entries), delete the oldest entry and insert the new one. You should implement the history as a circular queue, using "newest" and "oldest" indices.

The user can now perform the following functions as a shell command (not a process!):

- "history": print the history list (number of entry in the array and the appropriate command line), for all valid entries.
- "!!": retrieve the last command line (non-history, for clarification please refer to lab reading material) CL, enter CL again into the queue, and execute it (needs to be parsed again!).
- "!n": With n a number between 1 and HISTLEN, as in "!!" except with CL being the command line at history index n. If n is an invalid number (out of range, or no such entry yet) print an error message to stdout and ignore this command line.

Note that your shell should support history on top of all the other features: pipes, redirection, etc. This should not be hard if your code is well-designed.

Submission

Submit a zip file (named either [student-id-num].zip [student1-id-num_student2-id-num.zip] in case of pair submission) with the following files: mypipeline.c that implements the stand-alone pipe from part 1, and myshell.c, the source code of a shell supporting all features from lab 2 and from this lab (parts 2,3,4) and a makefile to compile and link them into the executables "mypipeline" and "myshell", respectively.