EDA093 / DIT401 Operating systems-Lab1

Lab group 22: Wenjun Tian, Mohammad Mourad, Zhuoer Shao

Date: September 24, 2024

1. Introduction

In this lab, we'll design and implement a custom shell that interprets user commands and supports both basic and advanced functionalities. All tests passed in both versions of tests.

Form now on, we will discuss the **methodology** used to build the lab, the **challenges** we faced, the **improvements** we can make and our **feedback** about the lab.

2. Methodology - How we meet the specifications

Here are how we build our code and meet all the requirements.

2.0 Code Structure Overview

Here is the overall structure of our code (following codes are pseudo-code).

```
initialize_bg_proc_list(); //To store background processes, used for CTRL-D handler
set_sig_handler_for_shell(); //Set SIGCHILD, SIGINT handlers for shell
for(;;) {
    cmd = readline();
    cmd_execute(cmd);
}
```

For the `cmd_execute()` function:

```
handle_exit_cmd();
pgm_execute(cmd, current_pgm, root); // execute pgms in the cmd recursively
//root == TRUE means it is called by the shell
```

The structure of `pgm_execute()`:

```
pid = fork();
pipe();// create pipe
if (pid == 0) {
    if (cmd -> background) {
        ignore_SIGINT(); // bg proc ingore SIGINT
    }
    if (root) {
```

```
        io_redirection(); // redirect io of the cmd in the first child of
shell
    } else {
        redirect_output_to_pipe(); // pgms communicate using pipes
    }
    pgm_execute(cmd, next_pgm, FALSE); // execute pgms recusively
    execvp(current_pgm);
} else {
    if (!root) {
        redirect_input_to_pipe(); //pgms communicate using pipes
    }
    if (cmd -> background) {
        add_bg_proc(child);
    } else {
        wait(child);
    }
}
```

2.1 Ctrl-D Handling - No orphan version

When `readline()` returns `NULL`, it indicates that the user has pressed `Ctrl-D`, and we should handle the `EOF` event. However, in our initial implementation, `exit()` was directly used to terminate the shell, but this would result in any background processes becoming orphaned and continuing to run in the background. In the improved design, `kill()` is used to send the `SIGTERM` signal to the background processes to ensure they are properly terminated.

Before terminate background processes, we need to record them first. Here we use a linked-list structure to store them.

```
typedef struct bg_proc {
    pid_t pid; //pid for bg proc, or pid == -1 indicates dummy head.
    struct bg_proc *next
} Bg_proc;
```

Actually we had a different version utilizing `process groups` to deal with this. We will discuss it latter in section 3.

When background processes are forked, we add them to the list.

```
if (command_list->background)
{
    // if child is running in background, add it to the bg_proc list.
    add_bg_process(bg_proc_head, pid);
}
```

When background processes exited before the shell, we remove them from the list in the `SIGCHILD` handler. `SIGCHILD` signal will be sent to the parent process when a child process finished execution, and the handler will be executed.

```c
void child_signal_handler()
{
    pid_t pid;
    //iterate all children
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0)
    {
        remove_bg_process(bg_proc_head, pid);
    }
}
```

The functions `add_bg_process()` and `remove_bg_process()` used in the code are simple operations to the linked-list, thus explanations are omitted.

Finally, when the shell captured `EOF`, we can kill all the background processes in the list, and then gracefully exit the shell.

```c
void EOF_handler(Bg_proc *bg_proc)
{
    //iterate all background processes
    while (bg_proc != NULL)
    {
        //exclude invalid pid, such as dummy head
        if (bg_proc->pid > 0)
        {
            if (kill(bg_proc->pid, SIGTERM) < 0) {
                perror("kill bg proc error before exiting shell");
            }
        }
        bg_proc = bg_proc->next;
    }
    //exit shell
    exit(0);
}
```

2.2 Basic Commands

We perform the basic commands by creating child processes to execute commands. `execvp()` system call is used to actually execute the command (program).

```c
execvp(current->pgmlist[0], current->pgmlist);
```

It receives the path of the program (absolute path, or environment variable `PATH` as prefix), and the parameters of the program. If execution failed, it will return `-1`.

2.3 & 2.8 Background Execution & No Zombies

When the command needs to be run in foreground, the parent process has to be blocked and call `waitpid()` or `wait()` to wait for its child to exit; when the command needs to be run in background, the parent process should not be blocked by the child process.

However, even if the parent process cannot be block by the background child process itself, it may be blocked by `pipe` operation, since parent process may need to read the output of its child process, and if the pipe is empty, the parent will be blocked. This feature guarantees the right execution order of background commands with pipes.

For the shell, specifically, if its children are running background, there is no way they cannot block it.

```
if (command_list->background)
{
     // if child is running in background, add it to the bg_proc list.
    add_bg_process(bg_proc_head, pid);
}
else
{
    // if child is running foreground, the parent should wait for it's exit.
    int status;
    if (waitpid(pid, &status, 0) != pid) {
        perror("Wait child process failed:");
    }
}
```

The `add_bg_process()` function is to store background processes which will be used when handling `EOF`, as we previously stated in section 2.2. We just ignore it here.

However, when parent processes do not call `wait()` or `waitpid()` to retrieve children's status, children processes will continue to stay in the process table and thus become `zombie processes`. In order to prevent this, we capture the `SIGCHLD` signal to ensure that the parent process can be notified when any of its child process exits.

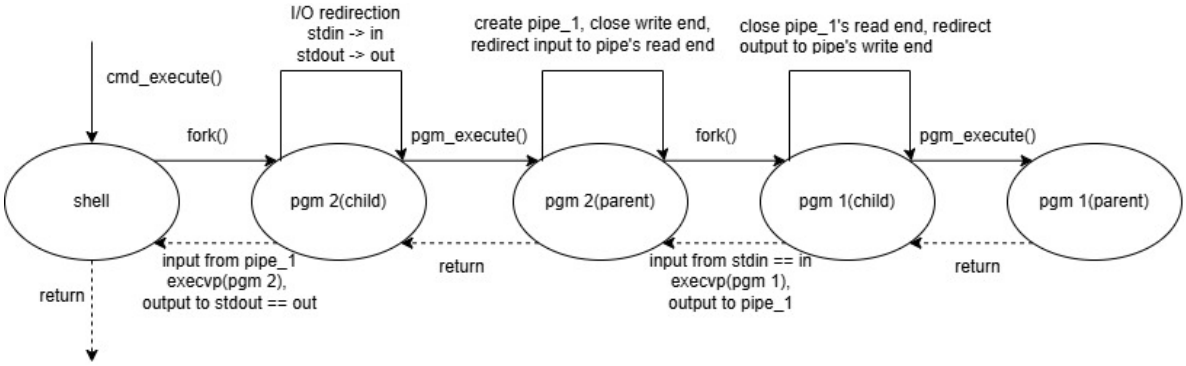
To handle the end events of child processes, in the `child_signal_handler` function, we continuously call `waitpid()` to ensure the parent process who captured `SIGCHILD` retrieves the `PIDs` of all finished child processes and remove them from the process table to avoid zombies. Moreover, we also remove all finished child processes from the background process list, which serves for `EOF` handling.

```
// SIGCHILD hanlder for the shell.
// when child process exited,
// shell captures the SIGCHILD signal and
// use 'waitpid' to remove child processes from
// the process table to avoid zombies
void child_signal_handler()
{
    pid_t pid;
    //iterate all children
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0)
    {
        remove_bg_process(bg_proc_head, pid);
    }
}
```

2.4 & 2.5. Piping & I/O Redirection

Here we discuss about the piping and I/O redirection together.

The following picture shows how data flows inside and out command `pgm1 < in | pgm2 > out` when using pipes and I/O redirection.



As we can see, I/O redirection changes file descriptor `STDIN_FILENO` to file `in` and `STDOUT_FILENO` to file `out` for the whole command. Therefore, we do the I/O redirection in the first child process of shell.

```
if (pid == 0) {
    if (root) {
        io_redirection();
    }
}
```

Now let's look into the `io_redirection()` function.

```
void io_redirection(Command *command_list)
{
    if (command_list->rstdout != NULL)
    {
        // if the last cmd, redirect to given parameter
        int out = open(command_list->rstdout, O_WRONLY | O_CREAT, 0777);
        dup2(out, STDOUT_FILENO);
        close(out);
    }
    if (command_list->rstdin != NULL)
    {
        // if the last cmd, redirect to given parameter
        int in = open(command_list->rstdin, O_RDONLY);
        dup2(in, STDIN_FILENO);
        close(in);
    }
}
```

To redirect output, firstly we call the `open()` system call to get the input file's file descriptor. We specify `O_WRONLY | O_CREAT` flags here, which means we open it in write-only mode, and if the file does not exist, we generate a empty file. After we get the file descriptor of `rstdout`: `out`, we call `dup2()` to duplicate `out` to `STDOUT_FILENO`, after which the standard output will be redirected to `rstdout`. Finally, we close the `out` file descriptor. Redirecting input can also be done in a similar way.

After discussing about I/O redirection, we are going to explain the piping feature.

Since we execute the programs in a recursive way, the `pgm 2` process will be forked first, and then it will create a pipe, close it's write end, and redirect it's `STDIN_FILENO` to the read end (which overrides the overall I/O redirection). After that `pgm 2` executes `pgm_execute()`, and `fork()` his child `pgm 1`. `pgm 1` process can now redirect it's `STDOUT_FILENO` to the write end of the pipe sharing with `pgm 2`. After that, the `pgm 1` process will call `execvp()` to actually execute

the program, and output to the pipe. After `execvp()` execution, the `pgm_execute()` called by `pgm 2` will return, and thus `pgm 2` process can execute it's program which fetches the output of `pgm 1` from the pipe as its input.

To create a pipe, redirect input/output to pipe's read/write end, and close the unused end, we use the following code.

Creating pipes:

```
    //creating pipes used to communicate with
    //preceding programs (child processes) in a pipeline fasion.
    int pipe_fd[2];
    if (pipe(pipe_fd) == -1)
    {
        perror("Pipe error:");
    }
```

`pipe()` function receive a `int[2]` to create a pipe. After creation, the file descriptors of read and write end will be respectively stored in the first and second element of the specified integer array, and return 0 if succeeded or -1 if failed.

Redirecting and closing file descriptors are explained previously in `io_redirection()` function.

2.6 Built-ins

Generally, we check if a program is inbuilt or not before executing it. If it is, we call our own implementation of it; else, we just use `execvp()` to execute it as external programs.

```
// execute inbuilt or ordinary program
if (!execute_inbuilt(current->pgmlist))
{
    // if not inbuilt, use execvp to run it.
    if (execvp(current->pgmlist[0], current->pgmlist) == -1)
    {
        // execution error
        perror(current->pgmlist[0]);
        exit(0);
    }
}
```

The `execute_inbuilt()` function check if a program is inbuilt or not. If it is inbuilt, we use our own implementation to handle it and return 1; else return 0.

```
// check if a cmd is inbuilt and run it(except for 'exit');
// if it's inbuilt, execute it and return 1;
// if it's not inbuilt, return 0

int execute_inbuilt(char **pgmlist)
{
    //check if 'cd'
    if (strcmp(pgmlist[0], "cd") == 0)
    {
        cd(pgmlist[1]);
        return 1;
    }
    return 0;
```

```
}
```

Here we only have `cd` here, because `exit` is specially treated.

We call `execute_inbuilt()` function in a child process of shell, so it's not convenient to exit shell in this place. Instead, we deal with `exit` at the very beginning.

```
//The entrance when start to execute a command.
void cmd_execute(Command *cmd)
{
    // inbuilt cmd 'exit' is specially treated here
    // since we should call exit(0) at shell process.
    if (strcmp(cmd->pgm->pgmlist[0], "exit") == 0) // exit shell
    {
        puts("Exiting lsh···");
        exit(0);
    }
    //Execute programs recursivly.
    pgm_execute(cmd, cmd->pgm, TRUE);
}
```

2.7 Ctrl-C Handling

`Ctrl-C` generates `SIGINT`.

For foreground commands, they should be killed by this signal, which is the default way of handle `SIGINT`, thus we do nothing for them.

For background commands, `SIGINT` should be ignored. Thus, when creating a background process, we set its signal handler for `SIGINt` as `SIG_IGN`.

```
// child process here
// if running in background, ignore SIGINT
if (command_list->background)
{
    signal(SIGINT, SIG_IGN);
}
```

For the shell, `SIGINT` should also be ignored.

```
// ctrl-c(SIGINT) handler for the shell.
// the shell just have to ignore it.
// Other foreground processes(cmds) just behave as default, i.e.,
// being killed by SIGINT.
// The background processes(cmds) should ignore it. This is set
// when a background process is forked. See the code below.
void int_signal_handler()
{
    printf("\n>");
}
```

2.8 No Zombies

Combined with section 2.3.

3. Challenges

3.1 Ways to store background processes

If we want to clear all background processes when exiting the shell, we have to store their `PID`s.

In our first version of implementation, we chose to put all of the background processes in to a `process group`. In this case, if we want to kill all of them, we just need to kill the process group.

```
// All background processes are placed in a group different from shell
// we denote their group id as 'bg_proc_pgid'
//there are 3 states for this 'bg_proc_pgid':
// 1: val == -1, no background processes occurred
// 2: val != -1, and there exists running background processes
// 3: val != -1, and all bg processes dead.
// we use shared memory here, since after fork, the child process will
// have it's own vitrual memory space, we can't simply use ordinary
variables
// to communicate between each other.
int bg_proc_pgid_shmid;
```

After the fork of a background process, we add it to the process group.

```
void add_bg_process()
{
    pid_t *bg_proc_pgid = (pid_t *)shmat(bg_proc_pgid_shmid, (void *)0, 0);
    // the proc group exists, than add to it
    if (*bg_proc_pgid != -1 && setpgid(getpid(), *bg_proc_pgid) == 0)
    {
        return;
    }
    //else, create a new group
    if (setpgid(0, 0) == -1)
    {
        perror("bg proc group falied");
    }
    *bg_proc_pgid = getpid();
    shmdt(bg_proc_pgid);
}
```

Before exiting the shell with `EOF`, we kill the whole process group.

```
void EOF_handler(pid_t *bg_proc_pgid)
{
  //kill the background proc group
  //if the group was created, then try to kill it.
  //even if the group is empty now(all proc dead)
  //try to kill a non-exsisting group does not generate any error.
  if (*bg_proc_pgid != -1)
  {
    kill(-*bg_proc_pgid, SIGINT);
  }
  shmdt(bg_proc_pgid);
  shmctl(bg_proc_pgid_shmid, IPC_RMID, NULL);
  exit(0);
```

```
14 }
```

This implementation passed all the tests (old version). However, it has a fatal bug in it.

If the final process in the group exited, the stored `PGID` remains unchanged. If we exit the shell right now, the `kill` function will work on that group. Unfortunately, the `PGID` may be assigned to another process group by the operating system, so in this case this code will randomly kill some processes, which causes a big problem.

3.2 Blocking or not when executing background commands with both pipes?

As we discussed in section 2.3&2.8, our final implementation will make all background programs in a single command not wait for the children. However, they are executing in a specified order. Take `pgm 1 | pgm 2` for example, if the `pgm 2` do not wait for the child `pgm 1` to finish, how can `pgm 2` get its input from `pgm 1`?

In the first implementation, when executing background processes, we only let shell not to wait for children, but for programs inside one command, they have to wait for their children. After test, it does not work. The shell will still be blocked by the program waiting for its child (we are stilled confused about this right now, actually).

Finally, we realize that the pipes used in inter-process communication can definitely block the background processes before their child processes finish. This part is articulated in section 2.3&2.8.

3.3 Recursive or iterative?

Actually, in the very beginning, we do not want to execute programs recursively, since it will make pipe operations and I/O redirections more tricky to implement. Moreover, in a recursive fashion, we have to deal more issues like the exit of recursion, which may lead to infinite recursion, and generate segment faults that are almost unable to locate problems.

However, we still chose the more challenging way to solve this problem, and we did it.

4. Potential Improvements

1. Inbuilt command `exit` should work in the same way as `EOF handler` (it is easy to fix, but we are close to the deadline, unfortunately).
2. Implement `nonhup` feature to leave background processes alive when exiting shell.
3. Implement other fancy features in modern shell, such as font and color customization, automatic command completion, git status presentation, etc.
4. Improve the code structure, do some decoupling, and make the comments more professional.

5. Discussion and Feedback

1. Maybe change tests near the deadline is a little bit scary.
2. The most import part of a shell is actually the `parser` right? But it is more like a compiler thing. A little bit pity we do not have to implement it by ourselves. It must be challenging!