Project 2

UNIX Shell Programming

Linux Kernel Module for Task Information

Introduction

The objective of this project is to implement a UNIX shell and a Linux kernel module that can be used to display information about the tasks running on the system.

The shell should be able to:
- Create child processes according to the user's input
- Provide history of the commands entered by the user
- Provide input and output redirection
- Use pipes to connect multiple commands
- Run commands in the background

The kernel module should be able to:
- Write a process ID to the `/proc/pid` file
- Read the command name, pid and state of the process from the `/proc/pid` file.

Implementation

Shell

The main structure of the shell is a loop that reads the user's input, parses it, and executes the commands. The data structures and the main function of the shell are as follows:

```c
struct shell_info
{
  char *username;
  char *cwd;
  FILE *history_file;
  char  last_command[BUFFER_SIZE];
  int   should_run;
};

struct command
{
  int   argc;
  char *argv[MAX_ARGS];
  int   in_fd;
  int   out_fd;
};

struct commands
```

```
{
  char           input_buffer[BUFFER_SIZE];
  struct command cmds[MAX_COMMANDS];
  int            cmd_cnt;
  char          *redirect_in;
  char          *redirect_out;
  int            background;
};

int main(void)
{
  init_shell();

  while (shell_info.should_run)
  {
    prompt_and_input();
    parse_input();
    exec_commands();
  }

  return 0;
}
```

- The `shell_info` structure holds information about the shell, - such as the username, current working directory, history file, and the last command entered by the user.
- The `command` structure represents a single command with its arguments and file descriptors for input and output.
- The `commands` structure holds multiple commands, input buffer, and redirection information.
- The main function of the shell reads the user's input, parses it, and executes the commands in a loop until the user exits the shell.

The details of the implementation are as follows:

```
void prompt_and_input(void)
{
  // Print the prompt and read the user's input
  // If the input is empty, return
  // If the input is '!!', copy the last command to the
  //                       input buffer
  // Copy the input buffer to shell_info.last_command,
  //                          commands.input_buffer,
  //                          and the history file
```

```
}

void parse_input(void)
{
  // Initailization
  char           *token;
  struct command *cmd;
  char            parse_buffer[BUFFER_SIZE];

  commands.cmd_cnt      = 1;
  commands.redirect_in  = NULL;
  commands.redirect_out = NULL;
  commands.background   = 0;
  strcpy(parse_buffer, commands.input_buffer);

  // Pick the first command and get the first token
  token      = strtok(parse_buffer, " \n");
  cmd        = commands.cmds;
  cmd->argc = 0;

  // Parse the input by tokenizing it
  while (token != NULL)
  {
    // Input redirection, read the next token as the input file
    if (strcmp(token, "<") == 0)
      commands.redirect_in = strtok(NULL, " \n");
    // Output redirection, read the next token as the output file
    else if (strcmp(token, ">") == 0)
      commands.redirect_out = strtok(NULL, " \n");
    // Pipe, move to the next command
    else if (strcmp(token, "|") == 0)
    {
      cmd->argv[cmd->argc] = NULL;
      ++cmd;
      cmd->argc = 0;
    }
    // Background process, set the flag
    else if (strcmp(token, "&") == 0)
      commands.background = 1;
    // Add the token to the current command's arguments
    else
      cmd->argv[cmd->argc++] = token;
    token = strtok(NULL, " \n");
  }
```

```
  // Terminate the last command's argument list
  cmd->argv[cmd->argc] = NULL;
  commands.cmd_cnt     = cmd - commands.cmds + 1;
}

void exec_commands(void)
{
  // If the command is a background process,
  // fork the shell and return
  int background_pid = 1;
  if (commands.background)
  {
    background_pid = fork();
    if (background_pid < 0)
    {
      printf("osh: fork failed\n");
      return;
    }
    // The parent process return to the main loop
    else if (background_pid > 0)
      return;
  }

  // Set the input and output file descriptors for
  // the first and last commands to STDIN and STDOUT
  // The file descriptors in the middle commands will be
  // set to the pipe file descriptors later
  commands.cmds[0].in_fd =
    commands.redirect_in != NULL ?
      open(commands.redirect_in, O_RDONLY) :
      STDIN_FILENO;
  commands.cmds[commands.cmd_cnt - 1].out_fd =
    commands.redirect_out != NULL ?
      open(commands.redirect_out,
           O_WRONLY | O_CREAT | O_TRUNC, 0666) :
      STDOUT_FILENO;

  // Execute the commands
  for (int i = 0; i < commands.cmd_cnt; ++i)
  {
    // If the command is a built-in command, execute it
    struct command *cmd = commands.cmds + i;
    if (check_builtin_commands(cmd))
```

```
      continue;

    // Create a pipe for the commands except the last one
    int pipe_fd[2];
    if (i != commands.cmd_cnt - 1)
    {
      if (pipe(pipe_fd) == -1)
      {
        printf("osh: pipe failed\n");
        return;
      }
      cmd->out_fd       = pipe_fd[1];
      (cmd + 1)->in_fd = pipe_fd[0];
    }

    // Fork the shell and execute the command
    pid_t pid = fork();
    if (pid < 0)
    {
      printf("osh: fork failed\n");
      return;
    }
    else if (pid == 0)
    {
      // Set the input and output file descriptors and
      // call execvp to execute the command
      dup2(cmd->in_fd, STDIN_FILENO);
      dup2(cmd->out_fd, STDOUT_FILENO);
      if (execvp(cmd->argv[0], cmd->argv) < 0)
            printf("osh: %s: command not found\n", cmd->argv[0]);
      exit(1);
    }

    // The parent process waits for the child process to finish
    waitpid(pid, NULL, 0);
    if (cmd->in_fd != STDIN_FILENO)
      close(cmd->in_fd);
    if (cmd->out_fd != STDOUT_FILENO)
      close(cmd->out_fd);
  }

  // If the shell is a background process, exit
  if (background_pid == 0)
    exit(0);
```

```
}

int check_builtin_commands(struct command *cmd)
{
  // Check if the command is a built-in command
  // If it is, calls the corresponding function
}

// Implementations of the built-in commands cd
void cd(struct command *cmd) {...}
// Implementations of the built-in commands history
void history(struct command *cmd) {...}
```

Kernel Module

The kernel module is implemented mainly with `proc_write` and `proc_read` functions.

- The `proc_write` function receives the user input, which should be a process ID, and converts it to an integer. The pid is stored as a global variable.
- The `proc_read` function reads the global pid variable and invokes system calls to get the command name, pid, and state of the process. The information is then written to the user buffer, which will be printed to the console by the system.

Correctness

Shell

The shell has been tested with various commands, including input/output redirection, pipes, background processes, and built-in commands. The shell correctly executes the commands and provides the expected output. The history feature also works as expected, storing and displaying the user's command history. The results of the tests are as follows:

```
ceryl@ubuntu:~/tmp$ ./osh
ceryl@osh> /home/ceryl/tmp$ ls -al
total 52
drwxrwxr-x  3 ceryl ceryl  4096 Apr 15 08:54 .
drwxr-xr-x 20 ceryl ceryl  4096 Apr 15 08:13 ..
-rw-rw-r--  1 ceryl ceryl   163 Apr 15 08:14 Makefile
-rwxrwxr-x  1 ceryl ceryl 23432 Apr 15 08:54 osh
-rw-rw-r--  1 ceryl ceryl  7456 Apr 15 08:15 osh.c
-rw-rw-r--  1 ceryl ceryl     0 Apr 15 08:54 .osh-history
-rw-rw-r--  1 ceryl ceryl  2513 Apr 15 08:14 pid.c
drwxrwxr-x  2 ceryl ceryl  4096 Apr 15 08:54 .vscode
ceryl@osh> /home/ceryl/tmp$
```

Figure 1: Run a simple ls command

```
ceryl@ubuntu:~/tmp$ ./osh
ceryl@osh> /home/ceryl/tmp$ ls -al | sort
drwxrwxr-x  2 ceryl ceryl  4096 Apr 15 08:54 .vscode
drwxrwxr-x  3 ceryl ceryl  4096 Apr 15 08:54 .
drwxr-xr-x 20 ceryl ceryl  4096 Apr 15 08:13 ..
-rw-rw-r--  1 ceryl ceryl     0 Apr 15 08:54 .osh-history
-rw-rw-r--  1 ceryl ceryl   163 Apr 15 08:14 Makefile
-rw-rw-r--  1 ceryl ceryl  2513 Apr 15 08:14 pid.c
-rw-rw-r--  1 ceryl ceryl  7456 Apr 15 08:15 osh.c
-rwxrwxr-x  1 ceryl ceryl 23432 Apr 15 08:54 osh
total 52
ceryl@osh> /home/ceryl/tmp$
```

Figure 2: Run a command with a pipe

```
ceryl@ubuntu:~/tmp$ ./osh
ceryl@osh> /home/ceryl/tmp$ ls
Makefile  osh  osh.c  pid.c
ceryl@osh> /home/ceryl/tmp$ ls &
ceryl@osh> /home/ceryl/tmp$ Makefile  osh  osh.c  pid.c
```

Figure 3: Run a command in the background

```
ceryl@ubuntu:~/tmp$ ./osh
ceryl@osh> /home/ceryl/tmp$ ls
Makefile  osh  osh.c  pid.c
ceryl@osh> /home/ceryl/tmp$ ps
    PID TTY          TIME CMD
  10246 pts/1    00:00:00 bash
  31108 pts/1    00:00:00 osh
  31366 pts/1    00:00:00 ps
ceryl@osh> /home/ceryl/tmp$ history
    1  ls
    2  ps
    3  history
ceryl@osh> /home/ceryl/tmp$
```

Figure 4: Display the command history

```
ceryl@ubuntu:~/tmp$ ./osh
ceryl@osh> /home/ceryl/tmp$ grep proc < ./pid.c > result
ceryl@osh> /home/ceryl/tmp$
```

Figure 5: Run a command with input and output redirection

```
tmp > C result
#include <linux/proc_fs.h>
static ssize_t proc_read(struct file *file, char __user *buf, size_t count, loff_t *pos);
static ssize_t proc_write(struct file *file, const char __user *usr_buf, size_t count, loff_t *pos);
static struct proc_ops proc_op = {
    .proc_flags = 0,
    .proc_read  = proc_read,
    .proc_write = proc_write,
static int proc_init(void)
    proc_create(PROC_NAME, 0666, NULL, &proc_op);
    printk(KERN_INFO "/proc/%s created\n", PROC_NAME);
static void proc_exit(void)
    remove_proc_entry(PROC_NAME, NULL);
    printk(KERN_INFO "/proc/%s removed\n", PROC_NAME);
static ssize_t proc_read(struct file *file, char __user *usr_buf, size_t count, loff_t *pos)
        len = sprintf(buffer, "No process with pid = [%d]\n", pid);
            len = sprintf(buffer, "No process with pid = [%d]\n", pid);
        printk(KERN_WARNING "/proc/%s copy_to_user failed\n", PROC_NAME);
static ssize_t proc_write(struct file *file, const char __user *usr_buf, size_t count, loff_t *pos)
        printk(KERN_WARNING "/proc/%s copy_from_user failed\n", PROC_NAME);
        printk(KERN_WARNING "/proc/%s kstrtoint failed\n", PROC_NAME);
module_init(proc_init);
module_exit(proc_exit);
```

Figure 6: The output file after the redirection test

```
ceryl@ubuntu:~/tmp$ ./osh
ceryl@osh> /home/ceryl/tmp$ cd ..
ceryl@osh> /home/ceryl$ cd ..
ceryl@osh> /home$ cd /dev
ceryl@osh> /dev$ cd /home/ceryl/tmp
ceryl@osh> /home/ceryl/tmp$ ls
Makefile  osh  osh.c  pid.c  result
ceryl@osh> /home/ceryl/tmp$ !!
ls
Makefile  osh  osh.c  pid.c  result
ceryl@osh> /home/ceryl/tmp$
```

Figure 7: Run a built-in command

```
ceryl@ubuntu:~/tmp$ ./osh
ceryl@osh> /home/ceryl/tmp$ ps -ael | grep sh | sort > result
ceryl@osh> /home/ceryl/tmp$
```

Figure 8: Run a complex command with multiple pipes and redirections

```
tmp > $ result
0 S  1000    1635    1407  0  80   0 - 943083 do_sys ?       00:00:04 gnome-shell
0 S  1000    1685    1407  0  80   0 - 145259 do_sys ?       00:00:00 gnome-shell-cal
0 S  1000    1738    1433  0  80   0 - 78459 do_sys ?        00:00:00 gvfsd-trash
0 S  1000    1771    1407  0  80   0 - 116184 do_sys ?       00:00:00 gsd-sharing
0 S  1000    2790       1  0  80   0 -   654 do_wai ?        00:00:00 sh
0 S  1000   32850   32849  0  80   0 -  2408 do_wai ?        00:00:00 bash
0 S  1000   32966   32869  0  80   0 -   654 pipe_r ?        00:00:00 sh
0 S  1000   33155    5848  0  80   0 -   647 do_wai pts/0    00:00:00 osh
0 S  1000    5848    2805  0  80   0 -  2690 do_wai pts/0    00:00:00 bash
1 I     0     153       2  0  60 -20 -     0 -      ?        00:00:00 zswap-shrink
1 I     0       5       2  0  60 -20 -     0 -      ?        00:00:00 slub_flushwq
1 S  1000    1583    1516  0  80   0 -  1511 -      ?        00:00:00 ssh-agent
4 S     0   32780     859  0  80   0 -  3507 -      ?        00:00:00 sshd
4 S     0     859       1  0  80   0 -  3049 -      ?        00:00:00 sshd
5 S  1000   32849   32780  0  80   0 -  3507 -      ?        00:00:00 sshd
```

Figure 9: The output of the complex command

Kernel Module

The test of the kernel module is executed with the shell written in the previous section. The result of the test is as follows:

```
ceryl@osh> /home/ceryl/tmp$ make
make -C /lib/modules/5.15.0-97-generic/build M=/home/ceryl/tmp modules
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-97-generic'
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-97-generic'
ceryl@osh> /home/ceryl/tmp$ sudo insmod pid.ko
ceryl@osh> /home/ceryl/tmp$ echo 33155 > /proc/pid
ceryl@osh> /home/ceryl/tmp$ cat /proc/pid
command = [osh] pid = [33155] state = [1]
ceryl@osh> /home/ceryl/tmp$ echo 123 > /proc/pid
ceryl@osh> /home/ceryl/tmp$ cat /proc/pid
command = [irq/50-pciehp] pid = [123] state = [1]
ceryl@osh> /home/ceryl/tmp$ echo 1 > /proc/pid
ceryl@osh> /home/ceryl/tmp$ cat /proc/pid
command = [systemd] pid = [1] state = [1]
ceryl@osh> /home/ceryl/tmp$
```

Figure 10: The output of the kernel module test

Bonus

The difference between anonymous pipes and named pipes is:

- Anonymous pipes are used between parent and child processes or between sibling processes. Named pipes can be used between any processes.
- Anonymous pipes provide a one-way communication channel. Named pipes provide a two-way communication channel.
- The lifetime of an anonymous pipe is limited to the lifetime of the processes that use it. Named pipes can be used by multiple processes even after the processes that created them have terminated.
- The data written to an anonymous pipe is lost when the pipe is closed. The data written to a named pipe is stored in the pipe until it is read by another process.
- Anonymous pipes are created using the `pipe` system call, and can be used immediately. Named pipes are created using the `mkfifo` system call, and must be openedbefore they can be used.

Conclusion

The shell and kernel module have been successfully implemented and tested. The shell can execute various commands, including input/output redirection, pipes, background processes, and built-in commands. The kernel module can write and read the process ID to and from the `/proc/pid` file. The project has provided valuable experience in UNIX shell programming and Linux kernel module development.