Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации Сибирский Государственный Университет Телекоммуникаций и Информатики СибГУТИ

Кафедра вычислительных систем

Лабораторная работа №2
по дисциплине «Операционные системы»

Выполнил:
Студент группы ИВ-921
Ярошев Р.А.

Работу проверил:
ассистент кафедры ВС
Петрук Е. А.

Новосибирск 2021

Задание

Реализовать упрощенную командную оболочку (аналог bash, csh, и т.д.). Оболочка должна предоставлять следующий функционал:

• Выводить приглашение для ввода команды;

• Запускать введенную команду, выводить результат;

• Поддерживать конвейер:

> command1 | command2 | command3

• Любой другой функционал на ваше усмотрение.

Системные вызовы, которые понадобятся вам при выполнении лабораторной (прочитать о них в мануале): fork, exec, wait, exit, waitpid, getpid, pipe, dup2.

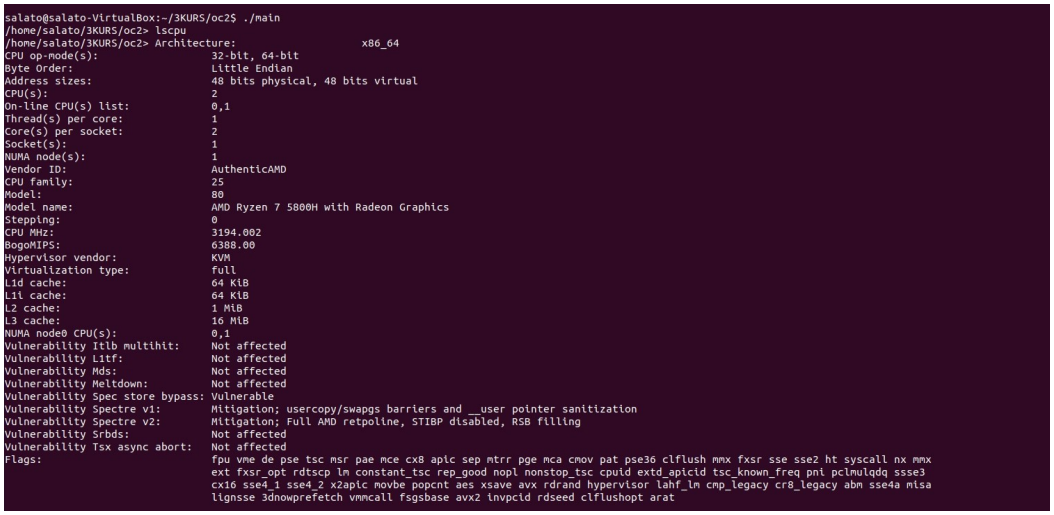
Результат работы программы

Сборка:

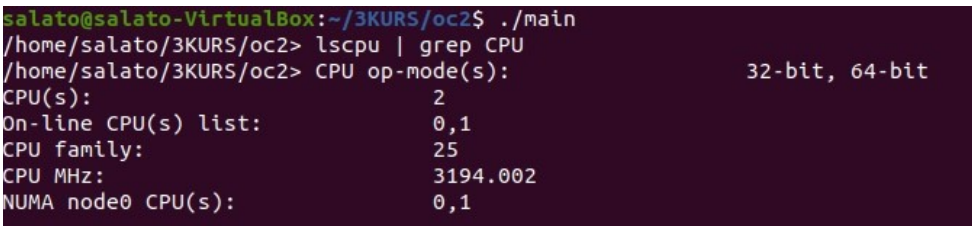
$ gcc main.c

Запуск:

$ ./main.out



Изображение 1. Сборка приложения и запуск программы



Изображение 2. Проверка работы конвейера

Листинг

main.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#define MAX_LINE_LENGTH 1024
#define BUFFER_SIZE 64
#define REDIR_SIZE 2
#define PIPE_SIZE 3
#define MAX_COMMAND_NAME_LENGTH 128
#define PROMPT_MAX_LENGTH 30
#define PIPE_OPT "|"

int running = 1;

char *get_current_dir() {
    char cwd[FILENAME_MAX];
    char*result = getcwd(cwd, sizeof(cwd));
    return result;
}

char *prompt() {
    static char *_prompt = NULL;

    if (_prompt == NULL) {
        _prompt = malloc(PROMPT_MAX_LENGTH * sizeof(char));
        if (_prompt == NULL) {
            perror("Error: Unable to locate memory");
            exit(EXIT_FAILURE);
        }
    }
}

void error_alert(char *msg) {
    printf("%s %s\n", prompt(), msg);
}

void remove_end_of_line(char *line) {
    int i = 0;
    while (line[i] != '\n') {
        i++;
    }

    line[i] = '\0';
}

void read_line(char *line) {
    char *ret = fgets(line, MAX_LINE_LENGTH, stdin);
```

```
    remove_end_of_line(line);

    if (strcmp(line, "exit") == 0 || ret == NULL || strcmp(line, "quit") == 0) {
        exit(EXIT_SUCCESS);
    }
}

void parse_command(char *input_string, char **argv, int *wait) {
    int i = 0;

    while (i < BUFFER_SIZE) {
        argv[i] = NULL;
        i++;
    }

    *wait = (input_string[strlen(input_string) - 1] == '&') ? 0 : 1;
    input_string[strlen(input_string) - 1] = (*wait == 0) ?
input_string[strlen(input_string) - 1] = '\0'
    : input_string[strlen(input_string) - 1];
    i = 0;
    argv[i] = strtok(input_string, " ");

    if (argv[i] == NULL) return;

    while (argv[i] != NULL) {
        i++;
        argv[i] = strtok(NULL, " ");
    }

    argv[i] = NULL;
}

int is_pipe(char **argv) {
    int i = 0;
    while (argv[i] != NULL) {
        if (strcmp(argv[i], PIPE_OPT) == 0) {
            return i;
        }
        i = -~i;
    }
    return 0;
}

void parse_redirect(char **argv, char **redirect_argv, int redirect_index) {
    redirect_argv[0] = strdup(argv[redirect_index]);
    redirect_argv[1] = strdup(argv[redirect_index + 1]);
    argv[redirect_index] = NULL;
    argv[redirect_index + 1] = NULL;
}

void parse_pipe(char **argv, char **child01_argv, char **child02_argv, int
pipe_index) {
    int i = 0;
    for (i = 0; i < pipe_index; i++) {
        child01_argv[i] = strdup(argv[i]);
    }
    child01_argv[i++] = NULL;
```

```
    while (argv[i] != NULL) {
        child02_argv[i - pipe_index - 1] = strdup(argv[i]);
        i++;
    }
    child02_argv[i - pipe_index - 1] = NULL;
}

void exec_child(char **argv) {
    if (execvp(argv[0], argv) < 0) {
        fprintf(stderr, "Error: Failed to execte command.\n");
        exit(EXIT_FAILURE);
    }
}

void exec_child_pipe(char **argv_in, char **argv_out) {
    int fd[2];
    if (pipe(fd) == -1) {
        perror("Error: Pipe failed");
        exit(EXIT_FAILURE);
    }

    if (fork() == 0) {
        dup2(fd[1], STDOUT_FILENO);
        close(fd[0]);
        close(fd[1]);
        exec_child(argv_in);
        exit(EXIT_SUCCESS);
    }

    if (fork() == 0) {
        dup2(fd[0], STDIN_FILENO);
        close(fd[1]);
        close(fd[0]);
        exec_child(argv_out);
        exit(EXIT_SUCCESS);
    }

    close(fd[0]);
    close(fd[1]);
    wait(0);
    wait(0);
}

int simple_shell_cd(char **args);
int simple_shell_exit(char **args);
void exec_command(char **args, char **redir_argv, int wait, int res);

char *builtin_str[] = {
    "cd",
    "exit"
};

int (*builtin_func[])(char **) = {
    &simple_shell_cd,
    &simple_shell_exit
};
```

```c
int simple_shell_num_builtins() {
    return sizeof(builtin_str) / sizeof(char *);
}

int simple_shell_cd(char **argv) {
    if (argv[1] == NULL) {
        fprintf(stderr, "Error: Expected argument to \"cd\"\n");
    } else {
        if (chdir(argv[1]) != 0) {
            perror("Error: Error when change the process's working directory to
PATH.");
        }
    }
    return 1;
}

int simple_shell_exit(char **args) {
    running = 0;
    return running;
}

int simple_shell_pipe(char **args) {
    int pipe_op_index = is_pipe(args);
    if (pipe_op_index != 0) {
        char *child01_arg[PIPE_SIZE];
        char *child02_arg[PIPE_SIZE];
        parse_pipe(args, child01_arg, child02_arg, pipe_op_index);
        exec_child_pipe(child01_arg, child02_arg);
        return 1;
    }
    return 0;
}

void exec_command(char **args, char **redir_argv, int wait, int res) {
    for (int i = 0; i < simple_shell_num_builtins(); i++) {
        if (strcmp(args[0], builtin_str[i]) == 0) {
            (*builtin_func[i])(args);
            res = 1;
        }
    }

    if (res == 0) {
        int status;
        pid_t pid = fork();
        if (pid == 0) {
            if (res == 0) res = simple_shell_pipe(args);
            if (res == 0) execvp(args[0], args);
            exit(EXIT_SUCCESS);

        } else if (pid < 0) {
            perror("Error: Error forking");
            exit(EXIT_FAILURE);
        } else {
            if (wait == 1) {
                waitpid(pid, &status, WUNTRACED);
            }
        }
```

```
    }
}

int main(void) {
    char *args[BUFFER_SIZE];
    char line[MAX_LINE_LENGTH];
    char t_line[MAX_LINE_LENGTH];
    char *redir_argv[REDIR_SIZE];
    int wait;
    int res = 0;

    while (running) {
        printf("%s%s> ", prompt(), get_current_dir());
        fflush(stdout);
        read_line(line);
        parse_command(line, args, &wait);
        exec_command(args, redir_argv, wait, res);
    }

    return 0;
}
```