Лабораторная работа №2, «Создание простой командной оболочки»
Группа: Б23-534
ФИО: Калашников Владимир Алексеевич
Номер в журнале: 5
Год в Москве: 2025

Описание лабораторной работы

Ссылка на GitHub

Работа выполнена на языке Си с использованием и включает в себя два файла:
1) *ProstoShell.h*, содержит в себе реализацию командной оболочки
2) *main.c*, запускает основную функцию оболочки

Также проект оснащён небольшим скриптиком *BuildAndRun.sh* для сборки с помощью CMake и последующего запуска оболочки.

Описание кода

В *main.c* действительно нет ничего примечательного, просто запуск оболочки

main.c
```c
#include "Source/ProstoShell.h"


int main() {
  shell_loop();
  return 0;
}
```

Файл *ProstoShell.h*, в его начале объявление констант и подключение необходимых библиотек.

ProstoShell.h
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <dirent.h>

#define MAX_LEN 1024
#define MAX_ARGS 64
```

Объявление всех функций в начале файла

```c
void shell_loop();

void parse_args(char *line, char **args);

void execute_command(char **args);

void execute_pipeline(char **args1, char **args2);

int is_builtin(char* cmd);

void handle_builtin(char **args);

void bgrep(char *pattern);
```

Встроенная функция *bgrep*, отсеивающая из потока ввода лишь те элементы, что содержат строку pattern.

```c
void bgrep(char *pattern) {
  char line[MAX_LEN];

  if (pattern == NULL || pattern[0] == '\0') {
    fprintf(stderr, "bgrep: missing pattern\n");
    return;
  }

  size_t pattern_len = strlen(pattern);

  while (fgets(line, MAX_LEN, stdin)) {
    if (strstr(line, pattern) != NULL) {
      printf("%s", line);
    }
  }
}
```

Функции *is_builtin* и *handle_builtin* отвечают за исполнение встроенных команд. Используются в *execute_command* и *execute_pipeline*, если функция не является встроенной, то она будет запущена силами *execvp*.

```c
int is_builtin(char* cmd) {
  return  strcmp(cmd, "cd") == 0 ||
          strcmp(cmd, "exit") == 0 ||
          strcmp(cmd, "bgrep") == 0;
}
```

```
void handle_builtin(char **args) {
  if (strcmp(args[0], "cd") == 0) {
    if (args[1] == NULL) {
      fprintf(stderr, "cd: missing argument\n");
    }
    else {
      if (chdir(args[1]) != 0) {
        perror("cd");
      }
    }
  }
  else if (strcmp(args[0], "exit") == 0) {
    exit(0);
  }
  else if (strcmp(args[0], "bgrep") == 0) {
    if (args[1] == NULL) {
      fprintf(stderr, "bgrep: missing pattern\n");
    }
    else {
      bgrep(args[1]);
    }
  }
}
```

Разбивает строку на аргументы.

```
void parse_args(char *line, char **args) {
  int i = 0;
  char* token = strtok(line, " \t\n");

  while (token != NULL && i < MAX_ARGS - 1) {
    args[i++] = token;
    token = strtok(NULL, " \t\n");
  }
  args[i] = NULL;
}
```

Функция *execute_command* проверяет, есть ли команда среди встроенных, затем или перенаправляет исполнение к *handle_builtin*, или создаёт *fork*, где команда выполняется силами *execvp*.

```
void execute_command(char **args) {
  if (args[0] == NULL) {
    return;
  }
```

```
 if (is_builtin(args[0])) {
  handle_builtin(args);
  return;
 }

 pid_t pid = fork();

 if (pid == 0) {
  execvp(args[0], args);
  perror("execvp");
  exit(1);
 } else if (pid < 0) {
  perror("fork");
 } else {
  wait(NULL);
 }
}
```

execute_pipeline реализовывает выполнение двух команд через **“pipe”**.
Сперва создаёт контейнер (тоже pipe), через который смогут общаться два процесса. Затем создаёт сами процессы для левой и правой команды, соединяет вывод левого со вводом контейнера, ввод правого с выводом контейнера, запускает процессы и ждёт конца работы. Как и *execute_command*, умеет отделять и исполнять встроенные команды.

```
void execute_pipeline(char **args1, char **args2) {
 int pipefd[2];
 pid_t pid1, pid2;

 if (pipe(pipefd)) {
  perror("pipe");
  return;
 }

 pid1 = fork();
 if (pid1 == 0) {
  close(pipefd[0]);
  dup2(pipefd[1], STDOUT_FILENO);
  close(pipefd[1]);

  if (is_builtin(args1[0]))
  {
   handle_builtin(args1);
   exit(EXIT_SUCCESS);
  }
  else
  {
   execvp(args1[0], args1);
   perror("execvp");
   exit(EXIT_FAILURE);
```

```
    }
  }

  pid2 = fork();
  if (pid2 == 0) {
    close(pipefd[1]);
    dup2(pipefd[0], STDIN_FILENO);
    close(pipefd[0]);

    if (is_builtin(args2[0]))
    {
      handle_builtin(args2);
      exit(EXIT_SUCCESS);
    }
    else
    {
      execvp(args2[0], args2);
      perror("execvp");
      exit(EXIT_FAILURE);
    }
  }

  close(pipefd[0]);
  close(pipefd[1]);
  waitpid(pid1, NULL, 0);
  waitpid(pid2, NULL, 0);
}
```