ЛАБОРАТОРНАЯ РАБОТА №2

Введение

Что такое shell?

Для работы в командной строке Linux предоставляет так называемую ”оболочку” или *shell*.

Оболочка — это некоторая интерактивная среда, интерпретатор. Её задача — непосредственно обрабатывать то, что вводит пользователь с помощью терминала. Можно думать о ней, как о некотором интерпретируемом языке, таком как, например, Python. Интерпретатор Python также предоставляет некоторую интерактивную среду (такие среды реализуют так называемый **REPL** (Read, Evaluate, Print and Loop)). Однако в то время как задачей среды для Python является исполнение выражений и инструкций, как правило, с целью изучения поведения некоторого кода и соответствующих структур данных, задачей оболочки является исполнение команд с целью реализовать то или иное взаимодействие с операционной системой. Эти команды бывают нескольких типов, принципиально их можно разделить следующим образом:

- **Исполняемая программа**. К этой категории относятся скомпилированные двоичные программы (например, написанные на C и C++) или программы, написанные на языках сценариев, таких как Shell, Perl, Python, Ruby. К таким программам относятся, например, `ls` и `cat`.
- **Встроенные (builtin) команды**. Команды, реализованные внутри самой командной оболочки, то есть не требующие никаких сторонних программ. Такой командой является, например, `cd`.

Подавляющее большинство команд относится именно к первой категории, более того, практически все встроенные команды сделаны такими ислючительно в целях оптимизации (зачастую удобнее или быстрее выполнить какое-то действие в рамках программы-оболочки) и могли бы быть реализованы и как внешние. Тем не менее, обойтись совсем без встроенных команд не получится - часть из них (как выше упомянутая `cd`) меняют состояние самой оболочки и потому не могут быть реализованы как исполняемые программы.

Как запускаются программы в Linux?

Важнейшим вопросом для любой операционной системы является механизм запуска программ. С точки зрения пользователя, запуск программ может осуществляться разными способами, один из которых - с помощью командной оболочки(как уже было сказано, большинство команд, выполняемых в оболочке, представляют собой как раз **исполняемые программы**). Но как именно оболочка (или любая другая программа) запускает другую программу? В Linux

для этого используется так называемый механизм *fork-exec*: он предоставляет собой последовательность системных вызовов `fork()` и `exec()` (под *процессом* пока будем понимать уже запущенную программу):

- `fork()` – создаёт копию текущего *процесса* (родительского), называемую потомком.
- `exec()` – заменяет программу в текущем процессе новой программой.

То есть когда пользователь вводит команду (например, `ls`), оболочка выполняет несколько шагов, чтобы запустить соответствующую программу:

1. **Создание нового процесса** — оболочка создаёт новый процесс с помощью системного вызова `fork()`.
2. **Замена программы** — в новом процессе вызывается `exec()`, который заменяет в нем программу на указанную (например, `ls`).

Простейший пример выполнения команды `ls` внутри программы на C:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid = fork(); // Create a child process
    if (pid == 0) { // Code executed in the child process
        execlp("ls", "ls", "-l", NULL); // Execute the "ls -l" command
        perror("exec failed"); // Print error if exec fails
    } else { // Code executed in the parent process
        wait(NULL); // Wait for the child process to complete
    }
    return 0;
}
```

Системные вызовы `fork()` и `exec()` здесь представляют собой вызовы обычных библиотечных функций языка C. Обратите внимания на проверку `if (pid == 0)` - в дочернем процессе `fork()` вернет 0, а в родительском - идентификатор дочернего. Так мы понимаем, в каком процессе мы оказались.

А вот очень упрощенный пример того, как может выглядеть фрагмент оболочки.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define MAX_LEN 1024

int main() {
    char command[MAX_LEN];

    while (1) {
        printf("my_shell> "); // Display shell prompt
        fgets(command, MAX_LEN, stdin); // Read user input
```

```
        command[strcspn(command, "\n")] = 0; // Remove newline character  from input
        if (strcmp(command, "exit") == 0) break; // Exit the shell


        pid_t pid = fork(); // Create a new process
        if (pid == 0) { // Code executed in the child process
            execlp(command, command, NULL); // Replace child process
            with the specified command
            perror("Command failed"); // Print error if exec fails
            exit(1);
        } else { // Code executed in the parent process
            wait(NULL); // Wait for the child process to finish
        }
    }

    return 0;
}
```

Эта программа считывает пользовательский ввод, представляющий собой название программы, и запускает эту программу с помощью механизма *fork-exec*. Здесь же реализована и простейшая встроенная команда `exit` - мы в этом случае не запускаем новую программу, а выполняем код внутри оболочки.

Задание

В рамках лабораторной работы необходимо в простейшем виде реализовать свой вариант командной оболочки. Оболочка должна поддерживать следующие функции:

- запуск программ
- механизм *pipe* по крайней мере для двух программ (то есть выполнение подобных инструкций `ls | grep "something"`)
- встроенные команды `cd` (смена директории) и `exit` (выход из оболочки)
- встроенную команду из списка ниже в соответствии с вариантом(свой вариант - [№ в списке группы на home.mephi.ru mod 5] + 1).

Языки реализации - на выбор C, Go, Python.

Отчет о работе должен содержать код и описание реализации, а также примеры выполнения команд.

Вариант	Команда	Функциональное определение
1	`bgrep`	**Формат**: `bgrep <pattern>` Фильтрует входной поток, оставляя только строки, соответствующие регулярному выражению `pattern`. **Вход**: стандартный поток (stdin). **Выход**: стандартный поток (stdout) с отфильтрованными строками. **Пример**: `ls | bgrep "lala"` (выведет только файлы, содержащие в названии “lala”).
2	`bcat`	**Формат**: `bcat <file1> <file2>` Выводит содержимое двух файлов в один объединённый поток. **Вход**: два аргумента — пути к файлам. **Выход**: содержимое файлов в стандартный поток (stdout). **Пример**: `bcat file1.txt file2.txt`.
3	`bwc`	**Формат**: `bwc` Подсчитывает количество строк во входном потоке данных. **Вход**: стандартный поток (stdin). **Выход**: одно число (количество строк). **Пример**: `ls | bwc` (отобразит число файлов в текущей директории).
4	`echo`	**Формат**: `echo <string>` Выводит переданную строку в стандартный поток вывода. **Вход**: аргумент `string`. **Выход**: переданный текст в стандартный поток (stdout). **Пример**: `echo "Hello, world!"` (отобразит “Hello, world!”).

№	Команда	Функциональное определение
5	`alias`	**Формат**: `alias <name>='<command>'` Создаёт псевдоним (alias) для указанной команды. **Вход**: аргумент `name` (имя псевдонима) и `command` (замещаемая команда). **Выход**: отсутствует. **Пример**: `alias ll='ls -l'` (после выполнения при вводе `ll` в оболочку будет исполняться `ls -l`).

Таблица 1 – Соответствие вариантов и спецификация встроенных команд