Отчёт по заданию task5.

МОДЕЛИРОВАНИЕ РАБОТЫ ИНТЕРПРЕТАТОРА SHELL

Мозговых В.В., 209 группа

6 декабря 2019 г.

Постановка задачи

Реализовать интерпретатор команд shell - (интерактивную) программу, воспринимающую команды, вводимые пользователем с терминала или находящиеся в командном файле.

Команды делятся на «внутренние», которые shell выполняет непосредственно, и «внешние», для выполнения которых создаются отдельные процессы. Имя любого исполняемого файла Unix является «внешней» командой shell. Кроме того, shell позволяет соединять выполняющиеся команды каналами (создавать «конвейер»), перенаправлять ввод-вывод команд в файлы, выполнять команды в асинхронном режиме.

Интерпретатор должен выдавать приглашение на ввод очередной команды, например “=> ” или “$ ”. При наступлении ситуации «конец файла»(Ctrl-Dпри вводе с клавиатуры)интерпретатор завершается

Синтаксис my_shell

<команда_shell> ::= <список_команд>

<список_команд> ::= <конвейер> { [один из &, ;] <конвейер> } [&, ;]

<конвейер> ::= <команда> { [один из |, ||, &&] <команда>}

<команда> ::= <простая_команда> [один из |, ||, &&] (< список_команд) [< <имя_файла>] [[один из >, >>] имя_файла]

<простая_команда> ::= <имя_файла> {<аргумент>} [< <имя_файла>] [[один из >, >>] имя_файла]

Список файлов проекта

Ниже перечислен список файлов проекта и краткое содержание каждого из них.

*** main.c**

Содержит объявление переменных для работы с буфером, списком и набором команд. Построчно считывает символы и переводит их в список слов. Превращает список слов в команды и передаёт их на запуск. Обрабатывает получаемые ошибки.

*** buf.h**

Объявляет структуру buf, необходимую для накопления символов и заполнения списка слов. Также объявлены функции очищения, создания удаления буфера, добавления в буфер символа, добавления строки в список слов и функция, которая анализирует и добавляет полученный символ.

*** buf.c**

*** list.h**

Объявление структуры list, содержащей массив строк и размер этого массива. Здесь объявлены функции, обрабатывающие спикок слов.

*** list.c**

*** cmd.h**

Объявление структур command и cmd(список команд). Реализовано заполнение структуры cmd с помощью структуры list. Также здесь находятся функции, реализующие запуск команд интерпретатора.

*** cmd.c**

*** symbols.h**

Библиотека обрабатываемых символов и функции для их обработки.

*** symbols.c**

*** error.h**

Объявление типа возникающих ошибок и функции, которая о них сообщает.

*** error.c**

*** myShellHeaders.h**

Включены все необходимые стандартные библиотеки для работы с процессами, а также error.h и symbols.h.

*** Makefile**

Описание внутренних структур данных

Структура данных, содержащая информацию для исполнения одной конкретной команды:

```
typedef struct command{
 char **argv;
 char **sub_shell;
 string file_in;
 string file_out;
 int out_type;
 int background;
 int next_type;
} command;
```

`argv` - список из имени команды и аргументов
`sub_shell` - аргументы для запуска в my_shell
`file_in` - переназначенный файл стандартного ввода
`file_out` - переназначенный файл стандартного вывода
`out_type` - тип вывода данных (0 - вывод на stdout, 1 - вывод в начало файла, 2 - вывод в конец файла)
`background` - отвечает за выполнение команды в фоновом режиме (0 или 1)
`next_type` - отвечает за связь текущей команды со следующей (0 - ; или &, 1 - |, 2 - ||, 3 - &&)

Структура данных для хранения всех команд:

```
struct cmd {
 command *commands;
 int commandsCount;
};
typedef struct cmd * cmd;
```

`commands` - указатель на массив структур, описывающих команды
`commandsCount` - размер соответствующего массива

Структура данных для контроля запущенных процессов:

```
typedef struct prStack {
 pid_t pid;
 struct prStack *next;
} prStack;
```

Представляет собой связный список
`pid` - pid запущенного процесса
`next` - указатель на структуру со следующим процессом или NULL

Структура данных для хранения слов из входной строки:

```
struct list {
 string *words;
 word_type *types;
 int count;
};
typedef struct list * list
```

`words` - массив слов (тип string = char *, определён в list.h)
`types` - массив типов для каждого слова (тип word_type определён в list.h и имеет всего 2 значения - simple и special)
`count` - количество слов в списке

Структура данных (буфер) для хранения накопленного слова:

```
struct buf {
 string word;
 int length;
 state st;
};
typedef struct buf * buf
```

`word` - строка с накопленным словом или NULL
`length` - длина накопленного слова
`st` - состояние перед получением последнего символа (тип state описан в buf.h)

Синтаксический разбор

Разбиение списка слов на команды производится с помощью функции cmd_fill(). Если список пуст, функция возвращает отсутствие ошибки. Далее проверяется корректное наличие скобок. После этого идёт проверка первого слова - первым словом обязана быть либо команда, либо команда sub_shell. Далее список проверяется в цикле. Если встретили открывающую скобку или простое слово, то заполняем поля argv или sub_shell текущей команды с помощью функций cmd_pushArgv или cmd_pushSubShell соответственно. В ином случае каждое специальное слово рассматривается отдельно:

';' - следующее слово должно быть простым или открывающей скобкой, next_type устанавливается в 0
'|' - следующее слово должно быть простым или открывающей скобкой, next_type устанавливается в 1
'||' - следующее слово должно быть простым или открывающей скобкой, next_type устанавливается в 2
'&&' - следующее слово должно быть простым или открывающей скобкой, next_type устанавливается в 3
'&' - следующее слово должно быть простым или открывающей скобкой, next_type устанавливается в 0, background устанавливается в 1
'<' - следующее слово должно быть простым, file_in присваивает следующее слово
'>' - следующее слово должно быть простым, file_out присваивает следующее слово, next_type устанавливается в 1
'>>' - следующее слово должно быть простым, file_out присваивает следующее слово, next_type устанавливается в 2

Наборы тестов

Работают правильно:

* ls -l | cat -n > f.out

Записать в f.out результат работы ls -l | cat -n

* (pwd; who) >> f.out

Записать в конец файла f.out путь до текущего каталога и имя пользователя

* ((ps)) > f.out

Перезаписать файл f.out, заполняя его таблицей запущенных процессов

* cat -n < f.out

Вывести пронумерованные строки файла f.out, если он существует

* ls -l > f.out

Записать список всех файлов в текущем каталоге с правами доступа в файл f.out

* (cat -n; head) < f.out

Вывести пронумерованные строки файла f.out

* (head; cat -n) < f.out

Сначала вывести 10 строк f.out, затем оставшиеся строки, но пронумерованные

* ./my_shell && pwd

Запустить интерпретатор внутри себя, если тот успешно завершится то напечатать путь то текущего каталога

* exit 1

Завершить процесс с кодом 1 для проверки предыдущей команды

* ./my_shell || ps

* exit 1

* false || true || ls || ps || pwd

Ничего не печатает

* true && false && ls && ps || pwd

Напечатать путь до текущего каталога

* yes | yes | yes | yes | sleep 10 | pwd

Напечатать текущую директорию. Приглашение к вводу следующей команды появится не раньше, чем через 10 секунд

* cat < /dev/urandom | head -c 4096 > "file.bin"

Создаёт файл случайных чисел размером 4096 байт

Тесты, на которые интерпретатор выдаёт syntax error:

*)(
* ()
* (()
* ((()))|cat
* echo "something
* echo "\"
* echo '\' '
* > something
* cat << f.out
* ps |
* ls || pwd &&
* ls & & pwd
* cat > > f.out

Если интерпретатору не удаётся открыть файл, он выдаёт Can't open "filename"

* cat nosuchfile > f.out

Для несуществующих программ при попытке запуска интерпретатор выдаст Can't execute "process name"

* a | a

Уведомления об ошибке при использовании cd:

* cd nosuchdir
* cd arg1 arg2 ...

Уведомления об ошибке при использовании exit:

* exit arg1 arg2 ...

Примечания

Исходный код программы и прочие файлы прокта содержатся в закрытом репозитории https://bitbucket.org/mozgovihws/task5/src/master

Запуск программы осуществляется через терминал.

При запуске без аргументов программа работает как интерпретатор, при запуске с аргументами исполняет свои аргументы.