操作系统 Project 1 报告

(Hermit Crab Shell)

隋清宇
5090309011

2011-11

目录

1 概述

本项目名为 Hermit Crab Shell,目的是实现一个简单的 shell。

本程序可以在 Minix 下编译运行,也可以在 Linux 下编译运行;需要 readline 库的支持。详细编译参数请参照 makefile。

2 项目结构

本项目包含多个文件,每个文件的作用如下:

2.1 shell.c

本文件为 shell 的启动文件,包含 main 函数。其中包括了 shell 运行的整个流程:读入命令——处理命令——执行命令。

2.2 const.h

本文件中定义了 shell 所需要的几个全局常量,包括一条命令的最大长度等。可以于编译前修改此文件以更改这些设置。

2.3 debug.h

这个头文件在每个文件前都会被包括,作用是定义一个 DEBUG 宏,以开启一些调试信息。

2.4 dir.c/h

这两个文件中包含了一些和目录处理有关的函数。

2.5 inline.c/h

这两个文件中包含了内部命令的具体实现。

2.6 command.c/h

这两个文件中定义了一系列和命令相关的数据结构,并实现了运行一条命令的函数。

2.7 prompt.c/h

这两个文件中定义了一些和获取当前命令提示符相关的函数。

2.8 tokens.c/h

这两个文件中定义了一系列和 token 相关的数据结构,以及它们的构造和析构函数(需要手动调用)。

2.9 lexer.lex/h

这两个文件是词法分析器的实现,使用 Flex 生成。

2.10 parser.c/h

这两个文件是语法分析器的实现,实现中手动遍历了语法树,并提取出相关信息传递给 command.c 或 inline.c 中的函数执行。

2.11 makefile

这是整个项目的 makefile,可以使用 make 编译。

3 项目功能

3.1 基本界面

本 shell 由显示当前路径的输入提示符开头,可以在之后输入命令的内容。

3.2 执行可执行文件

可以执行包括参数的可执行文件。

3.3 简单的错误提示

如果出现语法错误会报错并提示错误号。
错误号的具体对应详见 parser.h 内的枚举类型 PARSE_STATUS。

3.4 内置简单 shell 命令

内置有 cd 和 exit 命令,分别用来更改当前路径与退出 shell。
exit 命令支持一个正整数作为参数,表示退出时的错误号。

3.5 I/O 重定向

使用 >、< 或 >> 符号进行 I/O 重定向。

3.6 管道

使用 | 符号进行无名管道连接。

3.7 后台运行

使用 & 符号使一个程序后台运行。

3.8 顺序执行多个命令

使用; 符号顺序执行多个命令。

3.9 多个重定向、管道、后台运行与执行命令的组合

以上几种连接符可以交替使用,运行效果与 bash 一致。

3.10 历史记录与自动补全

使用 readline 和 history 库实现,与 bash 的用法基本一致。使用上、下键可以切换历史记录,使用 tab 键进行补全。

3.11 字符串支持

支持双引号字符串,连接方式与 bash 类似,实现了一部分通配符。
例如:执行命令

```
e"ch"o "abc""def"
```

可以输出字符串 “abcdef”(不含引号)。

4 设计思路

最开始,我试图不使用词法和语法分析器完成这个 shell。但是很可惜,随着代码的编写,我发现这样有很多扩展将很难完成。于是我更改了程序的架构,使用 Flex 生成了词法分析器,并自己编写了语法分析器,从而使程序的可扩展性上升了一个级别。程序也是从这时开始被提交到 GitHub 维护的。

整个程序的大致运行过程为:

1. 显示提示符并读入命令;
2. 对命令进行词法分析得到 token 流;
3. 对 token 流进行语法分析得到 command;
4. 调用函数执行 command;
5. 在 command 全部执行后进行 waitpid 等操作;
6. 重复这个过程。

5 设计细节

5.1 自动补全和历史记录

这一部分的完成上,我使用了 readline 库。在控制历史记录的时候,出现了一些小麻烦——运行过程中有很多历史记录变成了乱码。

经过调试,我发现 add_history 这个函数并不会自动拷贝传入的字符串到新的内存。所以我将代码改为不 free 传入的内存,成功。(在注释中有说明:“do NOT free single_line!”)

5.2 词法分析和语法分析

词法分析器的部分,我是用 Flex 完成的。但是即使这样,还是遇到了不小的麻烦。

因为编译原理的大作业我是使用 ANTLR 完成的,所以对 Flex 和 yacc 并不熟悉。一开始的时候,我完全是在“摸着石头过河”。而这时我还没有查询 Flex 和 yacc 怎么连接,这直接导致了我的 token 符号定义是用了自己的 tokens.h 文件。

而在查阅了更多资料之后,我发现如果想要让 Flex 和 yacc 连接,必须使用 yacc 生成的 tokens 文件,这样要对代码做不小的改动。于是,我决定自己手写一个语法分析器。

语法分析器的实现中,遇到的问题并不大。我设计了几个函数,每个函数可以分析一个特定的语法单元,并调用其它的函数进行进一步处理。

在最开始我有一个做法是,每分析出一个 command 随即在 parser 中执行。但是这样做实际上会在程序规模扩大后导致增加新功能的花费增多,并不是一个很好的做法。所以,我在之后的版本中将 command 抽象成了一个数据结构,这样就可以用一个更普适的处理函数去处理执行命令的步骤,也增强了代码的内聚性。

5.3 内存泄露

调试代码的时候,我遇到的最大问题就是严重的内存泄漏。因为之前很少用 C 编写代码,突然从 Python 和 C# 等有内存管理的语言转到 C 语言,有一些不太熟悉。

直到我找到了一个叫做 valgrind 的工具,这个问题才得以解决。这个工具可以检查 C/C++ 程序中的内存泄露并报错,非常适合调试程序。

但实际上,一些更好的编程风格会使得内存泄露问题不再出现的那么频繁。比如将大部分对象定义为指针类型,并为它们提供构造和析构函数;拷贝对象的时候提供函数,在内部进行深拷贝等。

6 其它信息

6.1 最新版本

可以在 GitHub 上获取到本程序的最新版本。

地址:https://github.com/sqybi/OS-Homework/tree/master/Chapter2/p02

6.2 协议