


Linux系统编程-作业2

- 学号：1120192092
- 班级：07111905
- 姓名：曾群鸿

功能介绍

本次作业 shell 起名为 zsh，来自我的名字的第一个字母（此 zsh 非彼 zsh ）。本次作业不仅完成了作业的基本要求，而且在基本要求上进行扩展，模仿 bash 的基本命令，已经能够取得一个较友好且可用的 shell 体验（在自行测试的时候会不自觉地按TAB键）。

✓ ls 命令

- 支持 -a 选项，打印隐藏文件（默认不打印文件）
- 仿bash，支持目录打印颜色为绿色，普通文件使用默认终端颜色

✓ mv 命令

- 支持移动一个文件
- 支持递归移动目录
- 仿bash，不添加 -r 选项，但支持目录文件和普通文件

✓ rm 命令

- 支持删除普通文件
- 支持 -r 选项，递归删除目录文件

- 支持 `rm file1 file2 file3 ...` , 删除多个文件
- 支持 `rm -r file1 file2 file3 ...` , 删除多个目录 (包括普通文件)

✓ `cp` 命令

- 支持复制普通文件
- 支持 `-r` 选项, 递归复制目录文件

✓ `touch` 命令

- 支持 `touch file1 file2 file3 ...` , 创建多个文件
- 方便在终端中测试 `cp` `mv` `rm` 等命令

✓ `mkdir` 命令

- 支持 `mkdir dir1 dir2 dir3 ...` , 创建多个目录

✓ `history` 命令

✓ `pwd` 命令

✓ `cd` 命令

✓ `exit` 命令

使用方法

下载源码后, 进入文件目录, 文件结构如下:

```
.
├─ Makefile
├─ readme.md
├─ readme.pdf
└─ src
    ├─ my_cp.c
    ├─ my_cp.h
    ├─ my_rm.c
    ├─ my_rm.h
    ├─ utils.c
    ├─ utils.h
    └─ zsh.c
```

运行 `makefile` 命令:

```
make
```

此时文件目录结构如下

```
.
├─ bin
│   └─ zsh
├─ Makefile
├─ obj
│   ├─ my_cp.o
│   ├─ my_rm.o
│   ├─ utils.o
│   └─ zsh.o
├─ readme.md
└─ readme.pdf
```

```
└─ src
    ├── my_cp.c
    ├── my_cp.h
    ├── my_rm.c
    ├── my_rm.h
    ├── utils.c
    ├── utils.h
    └─ zsh.c
```

其中，`bin/zsh` 为可执行程序，使用如下命令执行：

```
./bin/zsh
```

设计方案

主体结构

主体的控制结构主要包括三个部分，首先读取用户输入，然后将用户输入通过空格切分成若干个 `token`，最后将用户输入 `token` 与系统命令匹配，选择对应的命令执行。

考虑到本次实现的命令较多，且为了后续扩展的方便性，我设计了函数指针数组的形式来维护 `shell` 中的命令，采用这一设计就可以简化程序逻辑，不需要使用大量的 `if-else` 语句来判断用户输入的命令是什么。

具体来说，通过如下的方式存储函数：

```
// 当前shell支持的命令名
char *shell_funcname[] = {
```

```
    (char*)"ls",
    (char*)"pwd",
    (char*)"cd",
    (char*)"mkdir",
    (char*)"rm",
    (char*)"cp",
    (char*)"mv",
    (char*)"history",
};

// 当前shell支持的命令对应的函数指针
int (*shell_function[]) (char **) = {
    &my_ls,
    &my_pwd,
    &my_cd,
    &my_mkdir,
    &my_rm,
    &my_cp,
    &my_mv,
    &my_history,
};
```

然后就可以通过如下的方式进行命令匹配和调用：

```
int zsh_func_num = sizeof(shell_funcname) /
sizeof(char*);
for (int i = 0; i < zsh_func_num; ++i)
{
    if (!strcmp(args[0], shell_funcname[i]))
    {
        // 查找到用户命令，通过函数指针调用对应命令执行
        int ret = (*shell_function[i])(args);
        break;
    }
}
```

通过这样的设计大量简化了我的代码，并且高度可扩展，想象一下目前 Linux-shell 支持的命令个数，如果使用 if-else 会让代码变得极为臃肿。

通过将用户输入的命令分割成若干个 token，设置所有 shell 函数统一的参数格式接受解析后的 token 也使得整体的设计上变得统一而简单。并且后续命令的扩展，并不需要在前端解析命令后有任何修改，只需要一致地将 token 传递给新的函数指针即可。

CP命令

为了实现仿 linux 的 cp 命令，需要根据带复制文件的类型（普通文件还是目录文件）做不同的处理，实现复制的基本原理为创建目标目录下的文件，然后打开待复制的文件，读取待复制文件的内容，写入目标文件。

这样的做法会导致复制过去的文件与源文件不统一，如创建时间，修改时间等。我的解决方法为通过系统调用 `stat` 拿到待复制文件的基本信息，然后通过 `utime` 改变目标文件的时间属性，设置成与待复制文件的时间属性一致。

因为需要支持单个文件的复制，也需要支持目录文件的复制，因此我写了两个函数（见 `src/my_cp.c`），一个用于复制单个文件，一个用于递归复制目录文件。

这里需要注意目录文件时间属性的设置节点，不能在递归之前设置其时间属性，因为之后需要递归到其子目录中修改文件，会导致时间属性被我们覆盖。需要等到回溯时，即其子文件都完成复制和时间属性设置后，才能对其进行时间属性的设置。

RM与MV命令

`rm` 命令同样需要支持删除普通文件和目录文件，对于普通文件，使用系统调用 `remove` 进行删除；对于目录文件，则需要递归删除其子文件后，才能删除目录文件。递归遍历的做法与 `cp` 命令类似，使用 `opendir` 和 `readdir` 获取当前目录下所有文件，然后逐一判断进行处理，这里不再赘述。（代码见 `src/my_rm.c`）

我对 `rm` 命令做了较为友好的使用方式，支持 `rm file1 file2 ...`，也支持 `rm -r file1 file2 ...` 的形式。

`mv` 命令比较特殊，`mv source target` 的操作可以看成是两个操作：`cp source target`，然后 `rm source`。所以没有必要再实现一次 `mv` 命令，直接调用我们已经实现好的 `cp` 和 `rm` 命令就可以实现。

其他命令

`ls` 命令使用 `opendir` 和 `readdir` 获取目录下文件，然后进行判断和输出。考虑到用户体验，模仿bash对目录文件输出绿色，而普通文件则使用终端默认颜色输出。

`pwd` , `cd` , `mkdir` 则分别使用 `getcwd` , `chdir` , `mkdir` 系统调用实现，相对来说比较简单，这里不再赘述。（代码请见 `src/zsh.c`)

`mkdir` 命令同样做了较为友好的接口，允许创建任意多的目录，使用 `mkdir dir1 dir2 ...` 。

为了方便在 `shell` 中进行 `mv` , `cp` 等命令的测试，我增加了 `touch` 命令，支持 `touch file1 file2 ...` 创建多个文件，以方便测试其他命令的效果。

心得体会

经过上学期的操作系统课程中，我对其中Linux系统调用编程已经有了一些了解。因此本次作业相对容易上手，在本次作业我不仅仅局限于只实现对应命令的功能，也从软件系统设计和 `shell` 的实际可用性方面考虑，进行了一系列的扩展，代码量也来到了600多行。

但之前所了解的更多是基于理论性的，虽然有做了一些编程工作，但还不够深入。通过本次作业，多个命令的完成都需要不同的系统调用，在查看Linux系统调用API文档以及实践的过程中，让我对Linux的系统调用编程有了更进一步的认识，尤其是文件管理和错误管理的部分。

因为我对较多命令做了扩展，导致代码量较大，也切实体会到了Makefile的强大和便利，只需要写一次编译的规则，就可以通过 `make` 执行各种编译指令，极大提高了我的编程和调试的效率，也便于其他用户在拿到源码后的运行。

参考

我在之前没有使用过 `Makefile`，通过本次作业，我学习和实践了 `Makefile` 的应用，受益匪浅，下面是 `makefile` 的资料：

- 李老师的课件
- Chase Lambert等人编写的[makefile教程](#)
- 一个开源的[makefile工程实践教程](#)