

简单的shell的实现

目的：主要就是为了加深对shell的底层原理的理解

```
[xifeng@VM-16-14-centos linux-learning
[xifeng@VM-16-14-centos linux-learning
[xifeng@VM-16-14-centos 简易版的shell] 可以通过接口
total 0
[xifeng@VM-16-14-centos 简易版的shell]
```

可以获取到这些主机名等等这里直接用字符串打印了

- 当我们打开一个c文件默认就打开了三个输入输出流：**stdin(标准输入)**,**stdout(标准输出)**,**stderr(标准错误)**

STDIN(3)

Linux Programmer's Manual

STDIN(3)

NAME

stdin, stdout, stderr - standard I/O streams

SYNOPSIS

#include <stdio.h>

extern FILE *stdin;

extern FILE *stdout;

extern FILE *stderr;

1. 打印提示符

1 Linux学习服务器

[xifeng@VM-16-14-centos 简易版的shell]\$ vim mini_shell.c

[xifeng@VM-16-14-centos 简易版的shell]\$./mini_shell

因为没加\n所以数据在缓冲区
中没有被刷新出来，所以这里就需要用到fflush(立即刷新缓冲区)

1 Linux学习服务器

1: mini_shell.c

1 #include<stdio.h>

2 #include<unistd.h>

3 #include<string.h>

4 #include<stdlib.h>

5 int main()

6 {

7 //我们要先理解，shell要是自己实现他一定是一个死循环的程序，这样才能够不断的读取命令并执行

8 //1. 就是获取字符串，c语言的获取字符串可以用fgets

9 for(;;)

10 {

11 printf("[xifeng@VM-16-14-centos 简易版的shell]\$ ");

12 sleep(10);

13 }

14 }

15 }

16 }

后面会换成读取字符串的函数

1 Linux学习服务器

[xifeng@VM-16-14-centos 简易版的shell]\$./mini_shell

[xifeng@VM-16-14-centos 简易版的shell]\$

能够刷新出来了

1 Linux学习服务器

1: mini_shell.c

1 #include<stdio.h>

2 #include<unistd.h>

3 #include<string.h>

4 #include<stdlib.h>

5 int main()

6 {

7 //我们要先理解，shell要是自己实现他一定是一个死循环的程序，这样才能够不断的读取命令并执行

8 //1. 就是获取字符串，c语言的获取字符串可以用fgets

9 for(;;)

10 {

11 printf("[xifeng@VM-16-14-centos 简易版的shell]\$ ");

12 if fflush(stdout);

13 sleep(10);

14 }

15 }

16 }

2. 获取命令字符串

```
[xifeng@VM-16-14-centos 简易版的shell]$ ./mini_shell
[xifeng@VM-16-14-centos 简易版的shell]$ ls -a -l -n
ls -a -l -n

[xifeng@VM-16-14-centos 简易版的shell]$
```

```
1: mini_shell.c
1 #include<stdio.h>
2 #include<unistd.h>
3 #include<string.h>
4 #include<stdlib.h>
5 #define MAX_CMD 256
6 int main()
7 {
8     char commed[MAX_CMD];
9     //我们实现一个 shell 要是自己实现他一定是一个死循环的程序，这样才能够不断的读取命令并执行
10    //1. 就是读取字符串，c语言的获取字符串可以用fgets
11    for(;;)
12    {
13        printf("[xifeng@VM-16-14-centos 简易版的shell]$ ");
14        fflush(stdout);
15        //2. 读取命令
16        fgets(commed, sizeof(commed), stdin);
17        printf("%s\n", commed);
18    }
19    return 0;
20 }
```

能够成功的获取这个字符串了

定义了一个宏

这个fgets函数就是：将stdin中读取的字符串存入commed中
sizeof是读取字符串大小的字节数

但是只是这样会有一个问题他会读取一个\n，也就是出现了上面的ls -a -l -n下面空了一行，如何去掉这一行呢？

```
commed[strlen(commed)-1]='\0';//即可
//strlen(commed)-1指向的是\n，这里是把\n替换成\0即可解决问题
```

- o fgetc()函数

```
[xifeng@VM-16-14-centos 简易版的shell]$ ^C
[xifeng@VM-16-14-centos 简易版的shell]$ clear
[xifeng@VM-16-14-centos 简易版的shell]$ man 3 fgetc
```

GETS(3) Linux Programmer's Manual GETS(3)

NAME

fgetc, fgetc, getc, getchar, gets, ungetc - input of characters and strings

SYNOPSIS

```
#include <stdio.h>

int fgetc(FILE *stream);

char *fgetc(char *s, int size, FILE *stream);

int getc(FILE *stream);

int getchar(void);

char *gets(char *s);

int ungetc(int c, FILE *stream);
```

DESCRIPTION

fgetc() reads the next character from stream and returns it as an unsigned char cast to an int, or EOF on end of file or error.

getc() is equivalent to fgetc() except that it may be implemented as a macro which evaluates stream more than once.

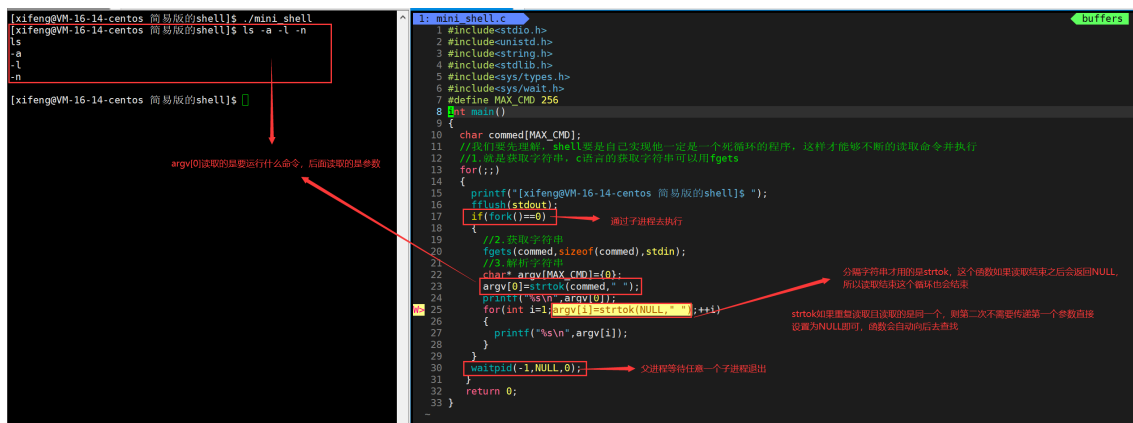
getchar() is equivalent to getc(stdin).

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

fgetc() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

ungetc() pushes c back to stream, cast to unsigned char

3. 解析命令字符串



- o strtok()

STRTOK(3)

Linux Programmer's Manual

STRTOK(3)

NAME

strtok, strtok_r - extract tokens from strings

SYNOPSIS

#include <string.h>

char *strtok(char *str, const char *delim);

char *strtok_r(char *str, const char *delim, char **saveptr);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

strtok_r(): _SVID_SOURCE || _BSD_SOURCE || _POSIX_C_SOURCE >= 1 || _XOPEN_SOURCE || _POSIX_SOURCE

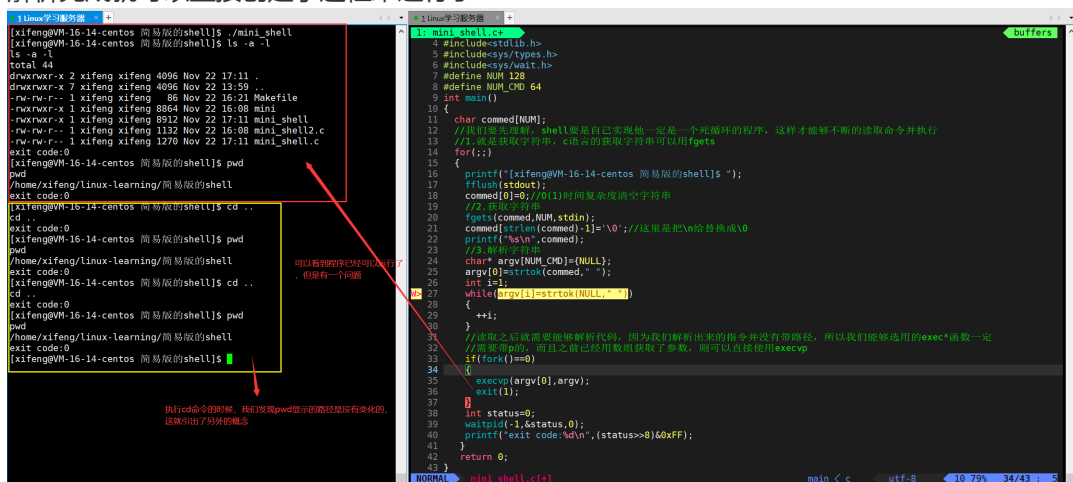
DESCRIPTION

The `strtok()` function breaks a string into a sequence of zero or more nonempty tokens. On the first call to `strtok()` the string to be parsed should be specified in `str`. In each subsequent call that should parse the same string, `str` must be NULL.

The `delim` argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in `delim` in successive calls that parse the same string.

Each call to `strtok()` returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, `strtok()` returns NULL.

- o 解析完成就可以直接创建子进程来运行了



- o 出现这种现象的主要原因是我们通过子进程来执行cd命令，子进程是回退了，可是并不会影响父进程，pwd显示的是当前进程的路径

```
[xifeng@VM-16-14-centos 简易版的shell]$ ./mini_shell
[xifeng@VM-16-14-centos 简易版的shell]$ cd ...
cd ...
exit code:0
[xifeng@VM-16-14-centos 简易版的shell]$ cd ...
cd ...
exit code:0
[xifeng@VM-16-14-centos 简易版的shell]$
```

改变的子进程的进程

```
1 23411 20406 11755 ? -1 R 1001 0:00 ./mini_shell
total 0
dr-xr-xr-x 9 xifeng xifeng 0 Nov 22 17:23 .
dr-xr-xr-x 176 root root 0 Jan 22 2022 ..
dr-xr-xr-x 2 xifeng xifeng 0 Nov 22 17:32 attr
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 autogroup
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 auxv
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 cgroup
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 clear_refs
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:23 cmdline
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 comm
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 coredump_filter
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 cpuset
lrwxrwxrwx 1 xifeng xifeng 0 Nov 22 17:32 cwd -> /home/xifeng/linux-learning/简易版的shell
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:23 environ
lrwxrwxrwx 1 xifeng xifeng 0 Nov 22 17:23 exe -> /home/xifeng/linux-learning/简易版的shell/mini_shell
dr-x--x--x 2 xifeng xifeng 0 Nov 22 17:23 fd
dr-x--x--x 2 xifeng xifeng 0 Nov 22 17:32 fdinfo
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 gid_map
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 io
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 limits
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 loginuid
dr-x--x--x 2 xifeng xifeng 0 Nov 22 17:32 map_files
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 maps
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 mem
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 mountinfo
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 mounts
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 mountstats
dr-xr-xr-x 5 xifeng xifeng 0 Nov 22 17:32 net
dr-x--x--x 2 xifeng xifeng 0 Nov 22 17:32 ns
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 numa_maps
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 oom_adj
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 oom_score
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 oom_score_adj
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 pagemap
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 patch_state
-r--r--r-- 1 xifeng xifeng 0 Nov 22 17:32 personality
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 projid_map
lrwxrwxrwx 1 xifeng xifeng 0 Nov 22 17:23 root -> /
-rw-r--r-- 1 xifeng xifeng 0 Nov 22 17:32 sched
```

父进程的进程没有变化

- cd这种命令和ls这种命令不一样，cd这种属于内建命令，他是需要父进程来执行的，而不是子进程去执行；其他的属于第三方命令，由子进程执行(/usr/bin下面看到的都是第三方命令)
 - 所以这个程序就需要步骤4
4. 检测命令是否是是需要shell本身执行的，内建命令

这里就需要一个函数chdir来更改路径

CHDIR(2)

Linux Programmer's Manual

CHDIR(2)

NAME

chdir, fchdir - change working directory

SYNOPSIS

#include <unistd.h>

int chdir(const char *path);

int fchdir(int fd);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

fchdir():

_BSD_SOURCE || _XOPEN_SOURCE >= 500 ||

_XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED

|| /* Since glibc 2.12: */

_POSIX_C_SOURCE >= 200809L

DESCRIPTION

chdir() changes the current working directory of the calling process to the directory specified in path.

fchdir() is identical to chdir(); the only difference is that the directory is given as an open file descriptor.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

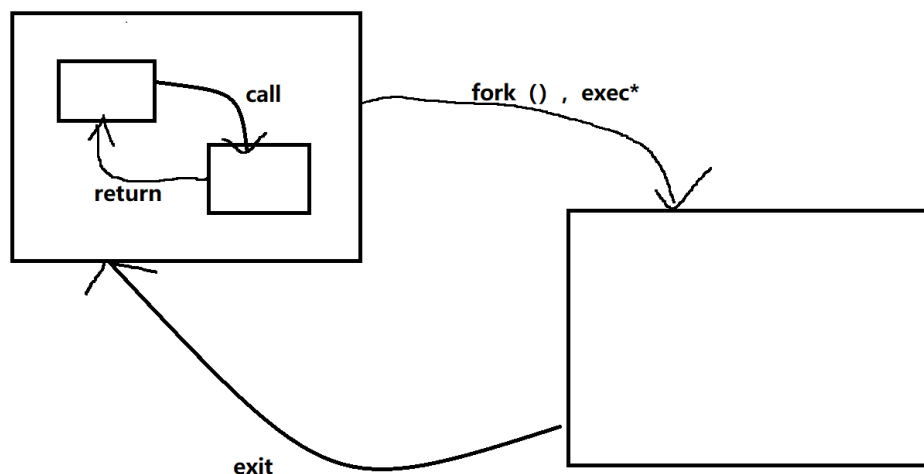
```
[xifeng@VM-16-14-centos ~]$ ./mini_shell.c
xifeng@VM-16-14-centos ~$ pwd
/home/xifeng/linux-learning/简易版的shell
xifeng@VM-16-14-centos ~$ cd ..
xifeng@VM-16-14-centos ~$ cd ..
xifeng@VM-16-14-centos ~$ pwd
/home/xifeng
xifeng@VM-16-14-centos ~$
xifeng@VM-16-14-centos ~$

//这里发生改变了，至此简易版的shell就完成了

1: mini_shell.c
0 #define NUM_CMD 64
0 int main()
10 {
11     char cmd[NUM_CMD];
12     //从stdin读取命令，shell是自己实现的，它是一个死循环的程序，这样才能不断的读取命令并执行
13     //1. 从stdin读取命令，c语言的读取字符串可以用fgets
14     for(;;)
15     {
16         printf("[xifeng@VM-16-14-centos ~]$ ");
17         fflush(stdout);
18         cmd[0] = '\0'; //0(1)时间复杂度清空字符串
19         //2. 从stdin读取命令
20         fgets(cmd, NUM_CMD, stdin);
21         cmd[strlen(cmd)-1] = '\0'; //这里是把\n给替换成\0
22         //3. 解析命令
23         char* argv[NUM_CMD] = {NULL};
24         argv[0] = strtok(cmd, " ");
25         int i = 1;
26         while(argv[i] = strtok(NULL, " "))
27         {
28             ++i;
29         }
30         //读取之后需要能够解析代码，因为我们解析出来的命令并没有带路径，所以我们能够选用的exec*函数一定需要带p的，而且之前已经用数组获取了参数，所以可以直接使用execvp
31         //4. 但是运行命令之前需要先判断是否是内建命令
32         //如果是内建命令，那么就在当前进程中执行，否则就fork一个子进程来执行
33         if(strcmp(argv[0], "cd") == 0 && argv[1] != NULL) //相等，并且带有参数
34         {
35             chdir(argv[1]);
36         }
37         if(fork() == 0)
38         {
39             execvp(argv[0], argv);
40             exit(1);
41         }
42         int status = 0;
43         waitpid(-1, &status, 0);
44         printf("exit code: %d\n", (status >> 8) & 0xFF);
45     }
46     return 0;
}
```

到这里简易的shell就完成了，有一个小问题，思考一下函数和进程之前有没有相似性呢？--->答案是有的

exec/exit就像call/return一个C程序有很多函数组成。一个函数可以调用另外一个函数，同时传递给它一些参数。被调用的函数执行一定的操作，然后返回一个值。每个函数都有他的局部变量，不同的函数通过call/return系统进行通信。这种通过参数和返回值在拥有私有数据的函数间通信的模式是**结构化程序设计的基础**



一个C程序可以fork/exec另一个程序，并传给它一些参数。这个被调用的程序执行一定的操作，然后通过exit(n)来返回值。调用它的进程可以通过wait来获取exit的返回值。