

0921 进程控制

上节课我们学了exec1

```
int main()
{
    pid_t id = fork();
    if(id == 0)
    {
        //子进程
        //现在想让子进程执行 ls -a -l
        //exec系列的函数，不需要进行返回值判断！
        printf("子进程开始运行， pid: %d\n",getpid());
        execl("/usr/bin/ls","ls","-a","-l",NULL);
        exit(1);
    }
    else
    {
        //父进程
        printf("父进程开始运行， pid: %d\n",getpid());
        int status = 0;
        pid_t id = waitpid(-1,&status,0); //阻塞等待 -- 子进程运行完毕 -- 然后父进程再执行
        if(id > 0)
        {
            //等待成功
            printf("wait success, exit code: %d\n",WEXITSTATUS(status));
        }
    }
    return 0;
}
```

现在想让子进程执行一个ls
然后父进程等子进程执行完

```
• [yufc@VM-12-12-centos 0921]$ ./exec
父进程开始运行， pid: 4570
子进程开始运行， pid: 4571
total 28
drwxrwxr-x 2 yufc yufc 4096 Jan 21 14:19 .
drwxrwxr-x 9 yufc yufc 4096 Jan 21 14:03 ..
-rwxrwxr-x 1 yufc yufc 8656 Jan 21 14:19 exec
-rw-rw-r-- 1 yufc yufc 776 Jan 21 14:15 exec.c
-rw-rw-r-- 1 yufc yufc 57 Jan 21 14:09 Makefile
wait success, exit code: 0
○ [yufc@VM-12-12-centos 0921]$
```

为什么我们要创建子进程？

如果不创建，那么我们替换的进程只能是父进程，如果创建了，替换的进程就是子进程，而不影响父进程了！

我们之前讲的：

加载新程序之前，父子的数据和代码的关系？ --- 代码共享，数据写时拷贝

那么现在当子进程加载新程序的时候，不就是一种“写入呢”？

此时要不要发生写时拷贝？要不要将父子的代码分离？ --- 必须分离 – 所以要写时拷贝

此时程序替换是替换所有的代码和数据。

因此当发生程序替换的时候，父子进程在代码和数据上就彻底分开了！

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl(const char *path, const char *arg,
    ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
    char *const envp[]);
```

命名理解

这些函数原型看起来很容易混,但只要掌握了规律就很好记。

- l(list): 表示参数采用列表
- v(vector): 参数用数组
- p(path): 有p自动搜索环境变量PATH
- e(env): 表示自己维护环境变量

execl 后面的 l 我们可以理解成 list

execv 后面的 v 我们可以理解成 vector

当使用 execl 的时候, 我们把东西参数一个一个传进去

当使用 execv 的时候, 我们要把东西包装好再传进去
这些很好理解

其中 execv 第二个参数 char *const argv[] 是一个指针数组

execl 和 execv 除了传参方式之外, 没有任何区别

```
10 int main()
11 {
12     pid_t id = fork();
13     if(id == 0)
14     {
15         //子进程
16         //现在想让子进程执行 ls -a -l
17         //exec系列的函数, 不需要进行返回值判断!
18         printf("子进程开始运行, pid: %d\n", getpid());
19         //execl("/usr/bin/ls", "ls", "-a", "-l", NULL);
20
21         char* const _argv[NUM] = {"ls", "-a", "-l", NULL};
22         execv("/usr/bin/ls", _argv);
23         exit(1);
24     }
25     else
26     {
27         //父进程
```

```
[yufc@VM-12-12-centos 0921]$ ./exec
父进程开始运行, pid: 9801
子进程开始运行, pid: 9802
total 28
drwxrwxr-x 2 yufc yufc 4096 Jan 21 14:39 .
drwxrwxr-x 9 yufc yufc 4096 Jan 21 14:03 ..
-rwxrwxr-x 1 yufc yufc 8656 Jan 21 14:39 exec
-rw-rw-r-- 1 yufc yufc 897 Jan 21 14:39 exec.c
-rw-rw-r-- 1 yufc yufc 57 Jan 21 14:09 Makefile
wait success, exit code: 0
```

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execlx(const char *path, const char *arg,
    ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
    char *const envp[]);
```

和execl的区别是
第一个参数一个是path
一个是file

一个问题：
要找到可执行程序，必须先找到程序！
所以我们要带上路径，
那么不带路径可以找到程序吗？？？

execlp 的 p 我们可以理解成
我会自己在环境变量PATH中进行查找，你不用告诉我你要执行的程序在哪里？

PATH



```
10 int main()
11 {
12     pid_t id = fork();
13     if(id == 0)
14     {
15         //子进程
16         //现在想让子进程执行 ls -a -l
17         //exec系列的函数，不需要进行返回值判断！
18         printf("子进程开始运行，pid: %d\n", getpid());
19         //execl("/usr/bin/ls", "ls", "-a", "-l", NULL);
20
21         //char* const _argv[NUM] = {"ls", "-a", "-l", NULL};
22         //execv("/usr/bin/ls", _argv);
23
24         execlp("ls", "ls", "-a", "-l", NULL);
25         exit(1);
26     }
27     else
28     {
```

```
• [yufc@VM-12-12-centos 0921]$ ./exec
父进程开始运行，pid: 12604
子进程开始运行，pid: 12605
total 28
drwxrwxr-x 2 yufc yufc 4096 Jan 21 14:47 .
drwxrwxr-x 9 yufc yufc 4096 Jan 21 14:03 ..
-rwxrwxr-x 1 yufc yufc 8656 Jan 21 14:47 exec
-rw-rw-r-- 1 yufc yufc 948 Jan 21 14:47 exec.c
-rw-rw-r-- 1 yufc yufc 57 Jan 21 14:09 Makefile
wait success, exit code: 0
• [yufc@VM-12-12-centos 0921]$
```

```
//char* const _argv[NUM] = {"ls", "-a", "-l", NULL};  
//execv("/usr/bin/ls", _argv);  
  
execlp("ls", "ls", "-a", NULL);  
exit(1);  
}
```

现在我们抛出一个疑问？

为什么有两个ls

这两个ls一样吗？

为什么要传两次？ --- 第一个ls代表【找到】，第二个ls表示【如何执行找到的这个程序】

```
int execl(const char *path, const char *arg, ...);  
int execlp(const char *file, const char *arg, ...);  
int execlx(const char *path, const char *arg,  
            ..., char * const envp[]);  
int execv(const char *path, char *const argv[]);  
int execvp(const char *file, char *const argv[]);  
int execvpe(const char *file, char *const argv[],  
            char *const envp[]);
```

一样道理 --- 按照之前的写法就行了

这里不做演示了

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl(const char *path, const char *arg,
          ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

我们重点讲解

execl 学会这个 execvpe 也是一个道理的

环境变量

即

我们可以通过execl（execvpe）向替换的程序传递环境变量！

但是对于这个函数
我们很不好演示
所以我们基于上面已经讲完的4个函数
我们先回答一些问题

1. 如何执行其他我自己写的C、C++二进制程序
2. 如何执行其他语言的程序

假如我们现在写一个 mycmd.c
然后让exec来调用mycmd的可执行

第一个问题：

Makefile 怎么一次性形成多个可执行呢？

1: Makefile+

```
1 .PHONY:all
2 all: exec mycmd
```

利用一个伪目标
伪目标：不需要依赖文件
只需要依赖方法

```
3
4 exec:exec.c
5     gcc -o $@ $^
6 mycmd:mycmd.c
7     gcc -o $@ $^
8 .PHONY:clean
9 clean:
10    rm -f exec mycmd
```

1: Makefile+

```
1 exec:exec.c
2     gcc -o $@ $^
3 mycmd:mycmd.c
4     gcc -o $@ $^
5 .PHONY:clean
6 clean:
7     rm -f exec
```

我们首先想到的肯定是这种方法：

但是此时如果直接make
是只会形成exec的
因为makefile从上到下扫描只会执行第一个

```
• [yufc@VM-12-12-centos 0921]$ make
gcc -o exec exec.c
gcc -o mycmd mycmd.c
• [yufc@VM-12-12-centos 0921]$ ls
exec  exec.c  Makefile  mycmd  mycmd.c
• [yufc@VM-12-12-centos 0921]$
```



```

1: mycmd.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 //mycmd -a/-b/-c ... 必须带一个选项, 否则不能执行
6 int main(int argc, char* argv[])
7 {
8     if(argc != 2)
9     {
10         printf("can not execute! \n");
11         exit(1);
12     }
13     if(strcmp(argv[1], "-a") == 0)
14     {
15         printf("hello a!\n");
16     }
17     else if(strcmp(argv[1], "-b") == 0)
18     {
19         printf("hello b!\n");
20     }
21     else
22     {
23         printf("default!\n");
24     }
25     return 0;
26 }

```

```

2
3 #include <stdio.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6
7
8 #define NUM 16
9
10 const char *myfile = "/home/yufc/bit/0921/mycmd"; //这里要把我们要替换的文件路径写好
11
12 int main()
13 {
14     pid_t id = fork();
15     if(id == 0)
16     {
17         //子进程
18         printf("子进程开始运行, pid: %d\n", getpid());
19
20         execl(myfile, "mycmd", "-a", NULL);
21
22         exit(1);
23     }
24     else
25     {
26         //父进程

```

```

• [yufc@VM-12-12-centos 0921]$ make
gcc -o exec exec.c
gcc -o mycmd mycmd.c
• [yufc@VM-12-12-centos 0921]$ ./exec
父进程开始运行, pid: 26191
子进程开始运行, pid: 26192
hello a!
wait success, exit code: 0
• [yufc@VM-12-12-centos 0921]$

```

这里我们使用的是绝对路径

我们可以改成相对路径试一下

见下一页

```

9
10 //const char *myfile = "/home/yufc/bit/0921/mycmd";//这里要把我们要替换的文件
11
12 const char *myfile = "./mycmd";//如果使用相对路径呢?
13
14
15 int main()
16 {
17     pid_t id = fork();
18     if(id == 0)
19     {
20         //子进程
21         printf("子进程开始运行, pid: %d\n", getpid());
22
23         execl(myfile, "mycmd", "-a", NULL);
24
25         exit(1);
26     }
27     else
28     {

```

```

1. 编译并执行 mycmd
• [yufc@VM-12-12-centos 0921]$ make
gcc -o exec exec.c
gcc -o mycmd mycmd.c
• [yufc@VM-12-12-centos 0921]$ ./exec
父进程开始运行, pid: 27349
子进程开始运行, pid: 27350
hello a!
wait success, exit code: 0
• [yufc@VM-12-12-centos 0921]$

```

绝对路径和相对路径都是可以的！

1. 如何执行其他我自己写的C、C++二进制程序
2. 如何执行其他语言的程序

第一个问题我们就解决了

第二个问题我们举一些小例子

```
/usr/bin/python
• [yufc@VM-12-12-centos 0921]$ vim test.py
• [yufc@VM-12-12-centos 0921]$ ls /usr/bin/python -al
lrwxrwxrwx 1 root root 7 Jan  8  2021 /usr/bin/python -> python2
○ [yufc@VM-12-12-centos 0921]$
```

稍微看下python的版本
是个python2
不过我们直接写个简单的打印就行

```
1: test.py
1
2
3 #! /usr/bin/python3.6
4
5 print("hello python!");
```

```
1: test.sh
1
2
3 #! /usr/bin/bash
4
5 echo "hello shell!"
```

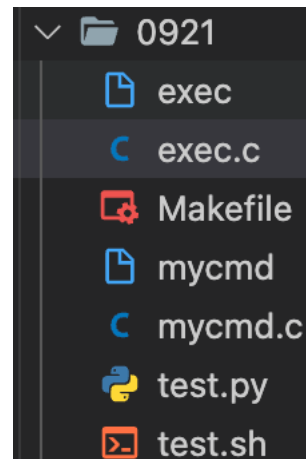
```
• [yufc@VM-12-12-centos 0921]$ bash test.sh
hello shell!
○ [yufc@VM-12-12-centos 0921]$
```

这里我们要了解一个概念：

C/C++是纯纯的编译性语言
我们是要编译 然后得到可执行程序
然后执行的

而python/JAVA/bash这些不是
这些事已经安装了一个解释器
比如/usr/bin/python这个就是一个解
释器
我们传入参数test.py 然后在python
这个软件就会帮我们执行

所以python test.py



```
0921
├── exec
├── exec.c
├── Makefile
├── mycmd
├── mycmd.c
├── test.py
└── test.sh
```

```

int main()
{
    pid_t id = fork();
    if(id == 0)
    {
        //子进程
        printf("子进程开始运行, pid: %d\n",getpid());

        //execl(myfile,"mycmd","-a",NULL);
        //执行其他语言的程序
        execlp("python","python","test.py",NULL);

        exit(1);
    }
    else
    {
        //父进程
    }
}

```

```

[yufc@VM-12-12-centos 0921]$ make
gcc -o exec exec.c
[yufc@VM-12-12-centos 0921]$ ./exec
父进程开始运行, pid: 1331
子进程开始运行, pid: 1332
hello python!
hello python!
hello python!
hello python!
hello python!
hello python!
wait success, exit code: 0
[yufc@VM-12-12-centos 0921]$

```

```

1: test.py
1
2
CC 3 #! /usr/bin/python3.6
4
5 print("hello python!")
6 print("hello python!")
7 print("hello python!")
8 print("hello python!")
9 print("hello python!")
10 print("hello python!")
11 print("hello python!")

```

当然 --- 执行python这些文件
除了
python test.py 这个方法

我们还可以通过给test.py添加执行权限
然后 ./test.py 来执行 --- 这样相当于系
统直接调用了python test.py

```

[yufc@VM-12-12-centos 0921]$ chmod +x test.py
[yufc@VM-12-12-centos 0921]$ ls
exec  exec.c  Makefile  mycmd  mycmd.c  test.py  test.sh
[yufc@VM-12-12-centos 0921]$

```

此时, 我们的 test.py 就是一个可执行程序了

```

if(id == 0)
{
    //子进程
    printf("子进程开始运行, pid: %d\n",getpid());

    //execl(myfile,"mycmd","-a",NULL);
    //执行其他语言的程序

    //execlp("python","python","test.py",NULL);
    execlp("./test.py","test.py",NULL);
    exit(1);
}
else

```

所以本质上 - exec系列函数可以执行任何程序!
本质 -- 就是加载器!

现在来看看最后这两个函数

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl(const char *path, const char *arg,
          ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

```
• [yufc@VM-12-12-centos 0921]$ ./exec
父进程开始运行, pid: 7585
子进程开始运行, pid: 7586
获取环境变量: MY_105_VAL: (null)
hello a!
wait success, exit code: 0
○ [yufc@VM-12-12-centos 0921]$
```

```
1: mycmd.c+ 1
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 //mycmd -a/-b/-c ... 必须带一个选项, 否则不能执行
6 int main(int argc, char* argv[])
7 {
8     if(argc != 2)
9     {
10         printf("can not execute! \n");
11         exit(1);
12     }
13
14     //你不是说 -- execl这个函数可以获取环境变量吗?
15     //那我们这里打印一下 MY_105_VAL 这个环境变量
16     //我们知道 MY_105_VAL 这个环境变量目前不存在
17     printf("获取环境变量: MY_105_VAL: %s\n", getenv("MY_105_VAL"));
18
19     if(strcmp(argv[1], "-a") == 0)
20     {
21         printf("hello a!\n");
22     }
23     else if(strcmp(argv[1], "-b") == 0)
24     {
25         printf("hello b!\n");
26     }
27     else
28     {
29         printf("default!\n");
30     }
31     return 0;
32 }
```

此时我们这个环境变量肯定是找不到的

```

15 int main()
16 {
17     pid_t id = fork();
18
19     char *const _env[NUM] = {"MY_105_VAL=888777666",
20                             NULL};
21
22
23     if(id == 0)                设置了一个环境变量
24     {
25         //子进程
26         printf("子进程开始运行, pid: %d\n",getpid());
27
28         execle(myfile,"mycmd","-a",NULL,_env);
29
30         //执行其他语言的程序
31
32         //execlp("python","python","test.py",NULL);
33         //execlp("./test.py","test.py",NULL);
34         exit(1);
35     }
36     else
37     {
38         //父进程
39         printf("父进程开始运行, pid: %d\n",getpid());
40         int status = 0;

```

```

• [yufc@VM-12-12-centos 0921]$ vim exec.c
• [yufc@VM-12-12-centos 0921]$ make
gcc -o exec exec.c
• [yufc@VM-12-12-centos 0921]$ ./exec
父进程开始运行, pid: 9724
子进程开始运行, pid: 9725
获取环境变量: MY_105_VAL: 888777666
hello a!
wait success, exit code: 0
• [yufc@VM-12-12-centos 0921]$

```

当然，如果我们在上面这份代码的main里面带上env
子进程也可以继承父进程的环境变量
因为环境变量具有全局属性！！

EXEC(3)

Linux Programmer's Manual

NAME

execl, execlp, execle, execv, execvp, execvpe – execute a file

SYNOPSIS

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

```
extern char **environ;
```

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,
           ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

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

`execvpe()`: GNU SOURCE

EXECVE(2)

Linux Programmer's Manual

NAME

execve – execute program

SYNOPSIS

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

```
int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

DESCRIPTION

`execve()` executes the program pointed to by `filename`. `filename` must be

注意，上面我们学的6个接口

其实严格意义上并不完全是系统调用！

而严格意义上的系统调用只有一个！

叫做 `execve`


```
问题 输出 调试控制台 终端 端口
EXECVE(2) Linux Programmer's Manual
NAME
    execve - execute program
SYNOPSIS
    #include <unistd.h>
    int execve(const char *filename, char *const argv[],
               char *const envp[]);
DESCRIPTION
    execve() executes the program pointed to by filename. filename must be
```

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl_e(const char *path, const char *arg,
            ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

→ 它才是系统调用，系统只给我们提供了一个！

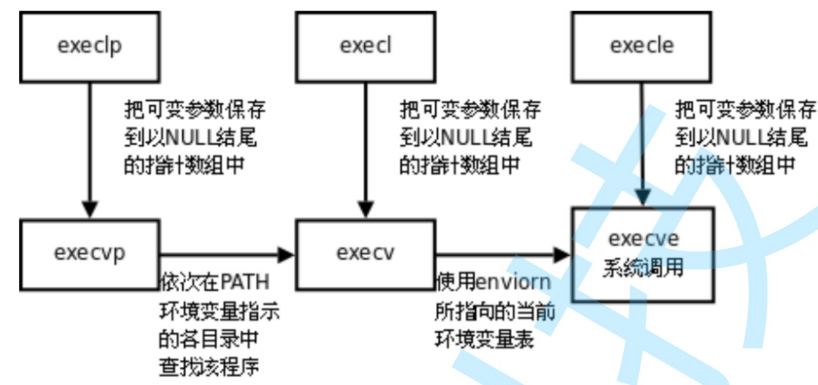


下面六个最终调用的还是上面这一个

下面这6个也可以叫系统调用

但它们是系统提供的基本封装 --- 用来满足不同的调用场景

函数名	参数格式	是否带路径	是否使用当前环境变量
execl	列表	不是	是
execlp	列表	是	是
execl_e	列表	不是	不是，须自己组装环境变量
execv	数组	不是	是
execvp	数组	是	是
execve	数组	不是	不是，须自己组装环境变量



写一个简单的shell

vim的一个技巧：现在我想要把所有的exec改成myshell

```
1: Makefile
1 exec:exec.c
2 gcc -o $@ $^
3 .PHONY:clean
4 clean:
5 rm -f exec
```

```
~
COMMAND Makefile
:%s/exec/myshell/g
```

```
1: Makefile+
1 myshell:myshell.c
2 gcc -o $@ $^
3 .PHONY:clean
4 clean:
5 rm -f myshell
```

1. 打印命令提示符

```
9 // 因此我们不怕命令在命令行中出错
10 int main()
11 {
12     //0. 命令行解释器，一定是一个常驻内存的进程 -- 不退出
13     while(1)
14     {
15         //1. 打印出提示信息 [yufc@localhost myshell]#
16         // 其实这一串是可以由系统接口来获取的 -- 不过我们不关心这些
17         printf("[yufc@localhost myshell]# ");
18         sleep(1);
19     }
20     return 0;
21 }
```

```
[yufc@VM-12-12-centos shell]$ ls
Makefile myshell.c
[yufc@VM-12-12-centos shell]$ vim Makefile
[yufc@VM-12-12-centos shell]$ vim myshell.c
[yufc@VM-12-12-centos shell]$ make
gcc -o myshell myshell.c
[yufc@VM-12-12-centos shell]$ ./myshell
```

此时，为什么我们的命令提示符没有打印？

缓冲区没有刷新！

```

0 int main()
1 {
2     //0. 命令行解释器，一定是一个常驻内存的进程 -- 不退出
3     while(1)
4     {
5         //1. 打印出提示信息 [yufc@localhost myshell]#
6         // 其实这一串是可以系统接口来获取的 -- 不过我们不关心这些
7         printf("[yufc@localhost myshell]# ");
8         fflush(stdout); //手动刷新
9         sleep(1);
10    }
11    return 0;
12 }

```

```

[yufc@VM-12-12-centos shell]$ vim myshell.c
[yufc@VM-12-12-centos shell]$ make
gcc -o myshell myshell.c
[yufc@VM-12-12-centos shell]$ ./myshell
[yufc@localhost myshell]# [yufc@localhost myshell]# [yufc@localhost myshell]# [yufc@localhost myshell]# [yufc@localhost myshell]# [yufc@localhost myshell]# [yufc@localhost myshell]# [yufc@localhost myshell]# [yufc@localhost myshell]# [yufc@localhost myshell]#

```

2. 获取用户命令

```

0 int main()
1 {
2     //0. 命令行解释器，一定是一个常驻内存的进程 -- 不退出
3     while(1)
4     {
5         //1. 打印出提示信息 [yufc@localhost myshell]#
6         // 其实这一串是可以系统接口来获取的 -- 不过我们不关心这些
7         printf("[yufc@localhost myshell]# ");
8         fflush(stdout); //手动刷新
9
10        //2. 获取用户的键盘输入[输入指的是各种指令和选项]
11        // 要输入 -- 我们就要提供一个输入的缓冲区
12        memset(cmd_line, '\0', sizeof cmd_line); //sizeof不是函数 -- 是运算符，所以可以不用()
13        if(fgets(cmd_line, sizeof cmd_line, stdin) == NULL)
14        {
15            //表示没有在stdin里面获取命令时出错
16            continue;
17        }
18        printf("echo: %s\n", cmd_line);
19    }
20    return 0;
21 }

```

这里把我们获取到的命令打印出来，看看对不对

```

• [yufc@VM-12-12-centos shell]$ vim myshell.c
• [yufc@VM-12-12-centos shell]$ make
gcc -o myshell myshell.c
• [yufc@VM-12-12-centos shell]$ ./myshell
[yufc@localhost myshell]# ls -al
echo: ls -al
[yufc@localhost myshell]#

```

我们发现，我们确实获得了命令
但是为什么中间多了一个换行？

因为我们输入的时候，是有回车的！

所以我们要对
cmd_line
里面把最后的 \n 去掉！

```

}
//此时要把cmd_line最后面的回车去掉
cmd_line[strlen(cmd_line)-1] = '\0';
printf("echo: %s\n", cmd_line);
}
return 0;

```

此时我们就能获取到正确的字符串了！

```

[yufc@VM-12-12-centos shell]$ make
gcc -o myshell myshell.c
[yufc@VM-12-12-centos shell]$ ./myshell
[yufc@localhost myshell]$ la -a -l
echo: la -a -l
[yufc@localhost myshell]$

```

3. 命令字符串的解析

首先

我们要把 字符串 "ls -a -l" 转成 "ls"和"-a"和"-l"

刷了这么多力扣

这个工作双指针O(n) 很简单

但是我们今天不去做这个工作

```

[yufc@VM-12-12-centos shell]$ ./myshell
[yufc@localhost myshell]$ ls -a -l
g_argv[0]: ls
g_argv[1]: -a
g_argv[2]: -l
[yufc@localhost myshell]$ ^C

```

```

//3. 拆分命令
// "ls -a -l" ----> "ls" "-a" "-l"
// strtok
// 第一次调用 -- 要传入原始字符串
// 如果还要继续解析原字符串 -- 传入NULL
g_argv[0] = strtok(cmd_line, SEP);
int idx = 1;
while(g_argv[idx++] = strtok(NULL, SEP)); //这种写法 -- 如果返回NULL, 子串提取完成
//测试一下看看提取的对不对
for(idx = 0; g_argv[idx]; idx++)
{
    printf("g_argv[%d]: %s\n", idx, g_argv[idx]);
}

```

//4. 执行这个命令

第四部分我们稍后再讲

//5. fork()

```
pid_t id = fork();
```

```
if(id == 0)
```

```
{
```

```
    //子进程
```

```
    //选择一个程序替换的接口
```

```
    //我们肯定要选带v和带p的 -- 因为指针数组准备好了，而且命令都在PATH中
```

```
    //为了方便 -- 带execvp是最好的
```

```
    printf("下面功能是让子进程执行的\n");
```

```
    execvp(g_argv[0],g_argv);
```

```
    exit(1);
```

```
}
```

```
//父进程 -- 这里我们不用else了 -- 子进程执行完直接退出，后面的肯定是父进程了
```

```
int status = 0;
```

```
pid_t ret = waitpid(id,&status,0);
```

```
if(ret > 0)
```

```
{
```

```
    printf("exit code: %d\n",WEXITSTATUS(status));
```

```
}
```

写到这里，基本上一个简单的shell就写完了

可以执行一些基本的命令了

```
[yufc@VM-12-12-centos shell]$ make
```

```
gcc -o myshell myshell.c
```

```
[yufc@VM-12-12-centos shell]$ ./myshell
```

```
[yufc@localhost myshell]$ ls -a -l
```

```
下面功能是让子进程执行的
```

```
total 28
```

```
drwxrwxr-x 2 yufc yufc 4096 Jan 21 17:21 .
```

```
drwxrwxr-x 3 yufc yufc 4096 Jan 21 16:25 ..
```

```
-rw-rw-r-- 1 yufc yufc 68 Jan 21 16:27 Makefile
```

```
-rwxrwxr-x 1 yufc yufc 9064 Jan 21 17:21 myshell
```

```
-rw-rw-r-- 1 yufc yufc 2871 Jan 21 17:21 myshell.c
```

```
exit code: 0
```

此时，我们抛出一个问题，如图

```
o [yufc@VM-12-12-centos shell]$ ./myshell
[yufc@localhost myshell]# pwd
下面功能是让子进程执行的
/home/yufc/bit/0921/shell
exit code: 0
[yufc@localhost myshell]# cd ..
下面功能是让子进程执行的
exit code: 0
[yufc@localhost myshell]# pwd
下面功能是让子进程执行的
/home/yufc/bit/0921/shell
exit code: 0
[yufc@localhost myshell]#
```

为什么我们的路径没有发生变化？

因为，我们执行cd
其实是子进程里面执行cd命令

并不影响父进程！

因此，我们要设置一些内置命令

如果是 cd 这些命令 --- 我们是不能创建子进程的

```
exit code: 0
[yufc@localhost myshell]# pwd
下面功能是让子进程执行的
/home/yufc/bit
exit code: 0
[yufc@localhost myshell]# cd ..
[yufc@localhost myshell]# pwd
下面功能是让子进程执行的
/home/yufc
exit code: 0
[yufc@localhost myshell]#
```

```
//4. TODO
if(strcmp(g_argv[0],"cd")==0)
{
    //让父进程执行 -- 不要创建子进程
    //内置命令（内建命令） -- 本质就是shell中的一个函数调用
    //我们用一个系统调用 -- chdir
    if(g_argv[1]!=NULL)chdir(g_argv[1]);
    continue;
}
```

可以把 ls 的颜色加一下

```
int idx = 1;
//可以把ls的颜色加一下
if(strcmp(g_argv[0], "ls") == 0)
{
    g_argv[idx++] = "--color=auto";
}
while(g_argv[idx++] = strtok(NULL, SEP)); //这种写法 -- 如果返回NULL, 子串提取完成
//测试一下看看提取的对不对
//for(idx = 0; g_argv[idx]; idx++)
```

```
[yufc@localhost myshell]# ls -al
下面功能是让子进程执行的
total 28
drwxrwxr-x 2 yufc yufc 4096 Jan 21 17:39 .
drwxrwxr-x 3 yufc yufc 4096 Jan 21 16:25 ..
-rw-rw-r-- 1 yufc yufc 68 Jan 21 16:27 Makefile
-rwxrwxr-x 1 yufc yufc 9168 Jan 21 17:39 myshell
-rw-rw-r-- 1 yufc yufc 3329 Jan 21 17:39 myshell.c
exit code: 0
```