

进程控制

1.子进程的创建

1.1 fork函数的概念

在Linux中fork函数是非常重要的函数，它从已存在进程中创建一个新进程。新进程为子进程（子进程的PID是0），而原进程为父进程。

```
#include <unistd.h>
pid_t fork(void);
```

返回值：fork创建子进程成功后，会给父进程返回子进程的PID，给子进程返回0，失败则返回-1。

为什么要让父进程拿到子进程的PID？因为子进程变为僵尸状态后，需要父进程读取子进程的退出信息并回收资源。

操作系统将会给创建成功的子进程：

- 1、给子进程分配新的内存块和内核数据结构（PCB、进程地址空间、页表等，并构建对应的映射关系）；
- 2、将父进程的部分数据结构内容拷贝至父进程；
- 3、把子进程添加到系统进程列表中；
- 4、fork返回，调度器开始调度。

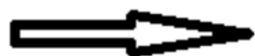
1.2 如何理解fork有两个返回值

fork的原理

```
Pid_t fork()
```

```
{
```

1. 给子进程分配新的内存块和内核数据结构
2. 给父进程的部分数据结构内容拷贝至父进程
3. 把子进程添加到系统进程列表中；
4.



在fork函数中就已经有了父子进程
父子进程返回各自的pid

```
return pid_t
```

```
}
```

CSDN @谁怕？一蓑烟雨任平生

在fork函数return之前，就已经有了父子两个进程，给父进程返回子进程的PID，给子进程返回0，失败则返回-1。

利用这个特性，我们可以用变量接收返回值，根据fork返回值不同让父子进程执行不同的代码。

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    pid_t id=fork();
    if(id==0)
    {
        printf("子进程:pid=%d,ppid=%d | grobal_val=%d,&grobal_val=%p\n",getpid(),getppid(),grobal_val,&grobal_val);
    }
    else if(id>0)
    {
        printf("父进程:pid=%d,ppid=%d | grobal_val=%d,&grobal_val=%p\n",getpid(),getppid(),grobal_val,&grobal_val);
        sleep(1);
    }
    else
```

```
{  
    printf("fork error\n");  
    return 1;  
}  
return 0;  
}
```

pid_t id=fork()这句代码父子进程谁先返回不确定。谁先返回，谁就在虚拟内存中写入id的值，后返回的进程由于进程的独立性将会发生写时拷贝。所以，我们可以看到父子进程的id变量的虚拟地址是一样的，但是内容却不一样。

1.3 fork调用失败的场景

系统中的进程数达到了最大限制。

2. 进程的终止

2.1 进程退出的常见场景

有如下三种退出的方式：

- 1、代码跑完，结果正确
- 2、代码跑完，结果不正确
- 3、代码没跑完，程序异常了

2.2 进程退出码

我们先前写C/C++代码的时候，都会在入口函数main函数开始写，我们总是喜欢在结尾的时候给上一个return 0，继而引发出了如下的两个问题：

return 0，给谁return？

为何是0？其它值可以吗？

下面一次解决：

1、return 0，给谁return？

给父进程，具体理由在下面会有讲解。

2、为何是0？其它值可以吗？

返回值代表的是进程代码跑完，结果是否正确，如果是0，则成功，非零则失败。所以我们在写一个程序的时候，如果测试结果正确，这里我们可以给上return返回值0，可如果不正确，我们return的应该是其他值以此表示结果失败，只不过我们平时都无脑return 0了，准确说是不太正确的。

此外，失败虽是用非零值表示，可也是有讲究的，结果成功都是用0表示，结果失败反倒用不同的数字来表示，以此表示失败的不同原因。

所以我们把main函数的return返回值称之为进程退出码！！进程退出码表征了进程推出的信息，此信息是要给父进程去读取的。

示例：

我们可以通过如下的指令查看退出码：

```
echo $?  
//$?表示在bash中，最近一次执行完毕时，对应进程的退出码！
```

```
#include <stdio.h>
```

```
int main()  
{  
    return 666;  
}
```

CSDN @谁怕？一蓑烟雨任平生

```
[yeyushengfan@VM-24-6-centos pro2]$ echo $?  
0  
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest  
[yeyushengfan@VM-24-6-centos pro2]$ echo $?  
6  
CSDN @谁怕？一蓑烟雨任平生
```

再比如我们平时在命令行输入的指令，诸如ls、cd.....类的，其退出码均为0，表示结果正确，可是当你随便输入一条错误指令的时候，其退出码则是某一数字表示结果错误：

问：一般而言，失败的的非零值我该如何设置呢？以及默认表达的含义？

```
#include <stdio.h>
#include <string.h>
int main()
{
    for(int i=0;i<100;i++)
    {
        printf("%d :%s\n", i,strerror(i));
    }
    return 0;
}
```

```
[yeyushengfan@VM-24-6-centos pro2]$ make
gcc -o mytest mytest.c
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest
0: Success
1: Operation not permitted
2: No such file or directory
3: No such process
4: Interrupted system call
5: Input/output error
6: No such device or address
7: Argument list too long
8: Exec format error
9: Bad file descriptor
10: No child processes
11: Resource temporarily unavailable
12: Cannot allocate memory
13: Permission denied
14: Bad address
15: Block device required
16: Device or resource busy
17: File exists
18: Invalid cross-device link
19: No such device
20: Not a directory
21: Is a directory
22: Invalid argument
23: Too many open files in system
24: Too many open files
```

CSDN @谁怕？一蓑烟雨任平生

总结：错误码退出码可以对应不同的错误原因，方便定位问题

2.3进程退出的常见情况

正常终止：

- 在main函数中return代表进程退出，非main函数return代表函数调用结束，这两点要注意。
- 在自己的代码任意地点中，调用exit()。
- _exit

异常终止：

- ctrl + c，信号终止

库函数exit

函数原型：

```
#include <stdlib.h>
void exit(int status);
```

库函数exit在进程退出后，会主动刷新缓冲区。

系统调用接口_exit

函数原型：

```
#include <unistd.h>
void _exit(int status);
```

系统调用_exit在进程退出后，并不会主动刷新缓冲区。

3.进程等待

3.1进程等待的必要性

子进程退出后会进入僵尸状态，父进程通过进程等待的方式，获取子进程的退出信息，回收子进程资源，让子进程结束僵尸状态。当然，在子进程没有退出时，父进程只能阻塞等待子进程变成僵尸状态。

3.2进程等待的方法

头文件：

```
#include <sys/types.h>
#include <sys/wait.h>
```

3.2.1系统调用wait

函数原型：

```
pid_t wait(int* status);
```

返回值：等待成功被返回等待进程的pid，失败返回-1。

参数：status输出型参数，获取子进程退出码和退出状态,不关心则可以设置成为NULL。通过wait让父进程获取子进程的PID。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    pid_t id = fork();
    if(id == 0)
    {
        int cnt = 5;
        while(cnt--)
        {
            printf("子进程: %d, 父进程: %d\n", getpid(), getppid())
            sleep(1);
        }
        exit(0);
    }

    sleep(5);
    pid_t ret = wait(NULL);
    if(ret > 0)
    {
        //父进程
        printf("等待成功\n");
    }
}
```

```
    return 0;  
}
```

```
[sqy@hecs-354086 lesson11.8]$ ./test  
子进程：27981,父进程：27980  
子进程：27981,父进程：27980  
子进程：27981,父进程：27980  
子进程：27981,父进程：27980  
子进程：27981,父进程：27980  
子进程：27981,父进程：27980  
等待成功
```

CSDN @平凡的小苏

```
ep myfork | grep -v grep; sleep 1; done  
PPID PID PGID SID TTY      TPGID STAT   UID    TIME COMMAND  
14886 18270 18270 14886 pts/0      18270 S+    1001  0:00 ./myfork  
18270 18271 18270 14886 pts/0      18270 S+    1001  0:00 ./myfork  
PPID PID PGID SID TTY      TPGID STAT   UID    TIME COMMAND  
14886 18270 18270 14886 pts/0      18270 S+    1001  0:00 ./myfork  
18270 18271 18270 14886 pts/0      18270 Z+    1001  0:00 [myfork] <defunct>  
PPID PID PGID SID TTY      TPGID STAT   UID    TIME COMMAND  
14886 18270 18270 14886 pts/0      18270 S+    1001  0:00 ./myfork  
18270 18271 18270 14886 pts/0      18270 Z+    1001  0:00 [myfork] <defunct>  
PPID PID PGID SID TTY      TPGID STAT   UID    TIME COMMAND  
PPID PID PGID SID TTY      TPGID STAT   UID    TIME COMMAND  
PPID PID PGID SID TTY      TPGID STAT   UID    TIME COMMAND
```

CSDN @平凡的小苏

```
[sqy@hecs-354086 lesson11.8]$ ./test  
子进程：27981,父进程：27980  
子进程：27981,父进程：27980  
子进程：27981,父进程：27980  
子进程：27981,父进程：27980  
子进程：27981,父进程：27980  
等待成功
```

CSDN @平凡的小苏

113.2.2系统调用waitpid

函数原型：

```
pid_t waitpid(pid_t pid,int* status,int options);
```

返回值：当正常返回的时候waitpid返回收集到的子进程的进程PID；如果设置了选项WNOHANG,而调用waitpid发现没有已退出的子进程可收集,则返回0；如果调用中出错,则返回-1,这时errno会被设置成相应的值以指示错误所在；

参数：

PID：

PID=-1, 等待任一个子进程。与wait等效。PID>0, 等待进程为PID的子进程。

status:

`WIFEXITED(status)`: 若为正常终止子进程返回的状态，则为真。（查看进程是否是正常退出）

`WEXITSTATUS(status)`: 若WIFEXITED非零，提取子进程退出码。（查看进程的退出码）

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    pid_t id = fork();
    if(id < 0) {
        perror("fork");
        return 1;
    }
    else if(id == 0)
    {
        //int *p = NULL;
        // child
        int cnt = 5;
        while(cnt)
        {
            printf("I am child, pid:%d, ppid:%d, cnt: %d\n",
                   id, getppid(), cnt);
            cnt--;
            sleep(1);
            /*p = 100;
        }
        exit(11);
    }
    else
    {
        int status = 0;
        pid_t ret = waitpid(id, &status, 0);
        if(ret > 0)
            if(WIFEXITED(status))
                printf("Child exited with status %d\n", WEXITSTATUS(status));
    }
}
```

```

{
    // 7F: 0111 1111
    // printf("wait success, ret: %d, exit sig: %d, e
    if(WIFEXITED(status))
    {
        printf("进程是正常跑完的, 退出码:%d\n", WEXITSTAT
    }
    else{
        printf("进程出异常了\n");
    }
}
else if(ret < 0)
{
    printf("wait failed!\n");
}

}
return 0;
}

```

options:

WNOHANG : 若PID指定的子进程没有结束，则`waitpid()`函数返回0，不予以等待。若正常结束，则返回该子进程的PID。

设置WNOHANG表示非阻塞等待，设置为0为阻塞等待

3.2.3阻塞和非阻塞等待

阻塞式等待：当父进程调用`wait/waitpid`（第三个参数为0）等待子进程，如果子进程暂未退出，父进程会被阻塞，暂停运行，如果父进程刚好没事干，可以选择使用阻塞等待。

非阻塞式等待：当父进程调用`waitpid`（第三个参数为WNOHANG）等待子进程，如果父进程检测到子进程未退出，父进程并不会原地等待，而是继续执行自己的代码。如果使用`while`循环，便能达到轮询的效果。

非阻塞式等待不会占用父进程的精力，父进程可以在轮询的过程中做其他事情：

```

#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>

```

```
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
    pid_t id = fork();
    if (id == 0)
    {
        //子进程
        int cnt = 5;
        while (cnt--)
        {
            printf("我是子进程， 我的PID：%d, 我的PPID：%d\n", get
sleep(1);
        }
        exit(104);
    }
    else if (id > 0)
    {
        //父进程
        //基于非阻塞的轮询等待方案
        int status = 0;
        while (1)
        {
            pid_t ret = waitpid(-1, &status, WNOHANG);
            if (ret > 0)
            {
                printf("等待成功, %d, 退出信号是：%d, 退出码是：%d\
break;
            }
            else if (ret == 0)
            {
                //等待成功了，但是子进程没有退出
                printf("子进程好了没，奥，还没，那么我父进程就做其他事
sleep(1);
            }
            else
            {
                //出错了，暂时不处理
            }
        }
    }
}
```

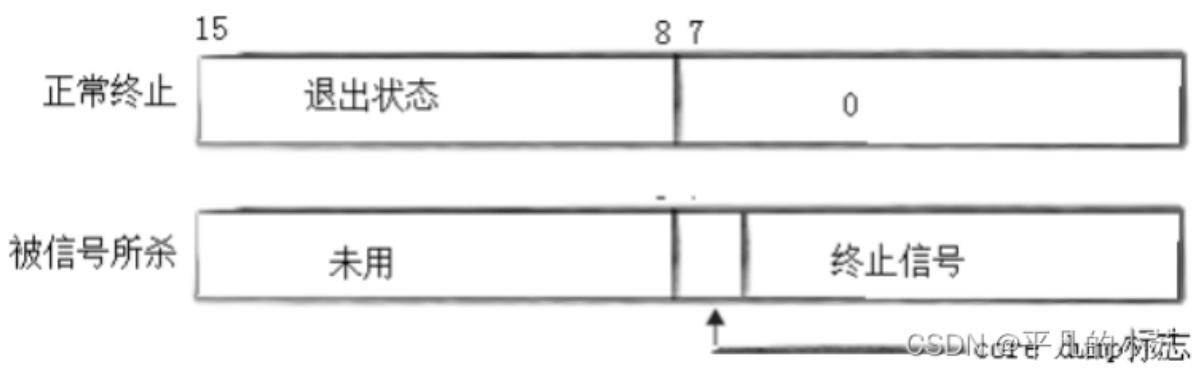
```
        }
    }
}
else
{
    //do nothing
}
return 0;
}
```

```
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest
子进程好了没，奥，还没，那么我父进程就做其他事情啦...
我是子进程，我的PID: 24204, 我的PPID: 24203
子进程好了没，奥，还没，那么我父进程就做其他事情啦...
等待成功，24204，退出信号是：0，退出码是：104
```

| **注：ret为0时等待成功，但是子进程没有退出，循环等待，大于0
则子进程退出，小于0为等待失败**

3.3status的意义

wait和waitpid，都有一个status参数，该参数是一个输出型参数，由操作系统填充。如果传递NULL，表示不关心子进程的退出状态信息。否则，操作系统会根据该参数，将子进程的退出信息反馈给父进程。status不能简单的当作整型，可以当作位图来看待。（只研究status低16比特位）：



解释上图：

- 在status的低16比特位当中，高8位表示进程的退出状态，即退出码。进程若是被信号所杀，则低7位表示终止信号，而第8位比特位是core dump标志。
- core dump表示是否正常结束

4. 进程程序替换

4.1 进程程序替换的概念

用fork创建子进程后执行的是和父进程相同的程序(但有可能执行不同的代码分支),子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时,该进程的用户空间代码和数据完全被新程序替换,从新程序的启动例程开始执行。调用exec并不创建新进程,所以调用exec前后该进程的id并未改变。

替换函数：

```
#include <unistd.h>
//execve的封装
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
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
//系统调用
int execve(const char *filename, char *const argv[], char *con
```

l(list) : 表示参数采用列表；

p(path) : 带p, 不用传入地址, 传入可执行程序的名字即可, 它会自动去环境变量PATH中寻找该可执行程序的地址；

`v(vector)` : 执行参数放入数组中，统一传递；

`e(env)` : 可以传入自己写的环境变量。

4.1.1 通过execl函数调用ls命令：

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("process is running.....\n");
    execl("/usr/bin/ls"/*要执行的程序*/, "ls", "--color=auto", "-a");
    printf("process is down.....\n");//这句话并不会被打印，因为后续
    return 0;
}
```

```
[yeyushengfan@VM-24-6-centos pro2]$ ./mytest
process is running.....
total 32
drwxrwxr-x  2 yeyushengfan yeyushengfan  4096 Jan 22 19:55 .
drwx----- 13 yeyushengfan yeyushengfan  4096 Jan 22 01:18 ..
-rw-rw-r--  1 yeyushengfan yeyushengfan     61 Jan 22 12:28 makefile
-rwxrwxr-x  1 yeyushengfan yeyushengfan 15872 Jan 22 19:55 mytest
-rw-rw-r--  1 yeyushengfan yeyushengfan    368 Jan 22 19:55 mytest.c
```

4.1.2 通过execv调用另一个文件打印出环境变量

```
//mycommand.c文件
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
    pid_t id = fork();
    if(id == 0)
    {
        printf("before: I am a process, pid: %d, ppid: %d\n",
               getpid(), getppid());
        char* const myargv[] = {
```

```

        "otherExe",
        "_a",
        "_b",
        "_c",
        NULL
    };

    execv("./otherExe", myargv);
    printf("after: I am a process, pid: %d, ppid:%d\n", getppid(), getpid());
    exit(1);
}

pid_t ret = waitpid(id, NULL, 0);
if(ret > 0) printf("wait success, father pid:%d, ret id: %d\n", getppid(), getpid());
sleep(5);
return 0;
}

//otherExe.cpp文件

#include <iostream>
using namespace std;
int main(int argc, char* argv[], char*env[])
{
    cout << argv[0] << " begin running " << endl;
    cout << "这是命令行参数" << endl;
    for(int i = 0; argv[i]; i++)
    {
        cout << i << " : " << argv[i] << endl;
    }
    cout << "这是环境变量" << endl;
    for(int i = 0; env[i]; i++)
    {
        cout << i << " : " << env[i] << endl;
    }
    cout << argv[0] << " stop running " << endl;
    return 0;
}

```

4.1.3通过execle函数调用外部程序并使用自定义环境变量

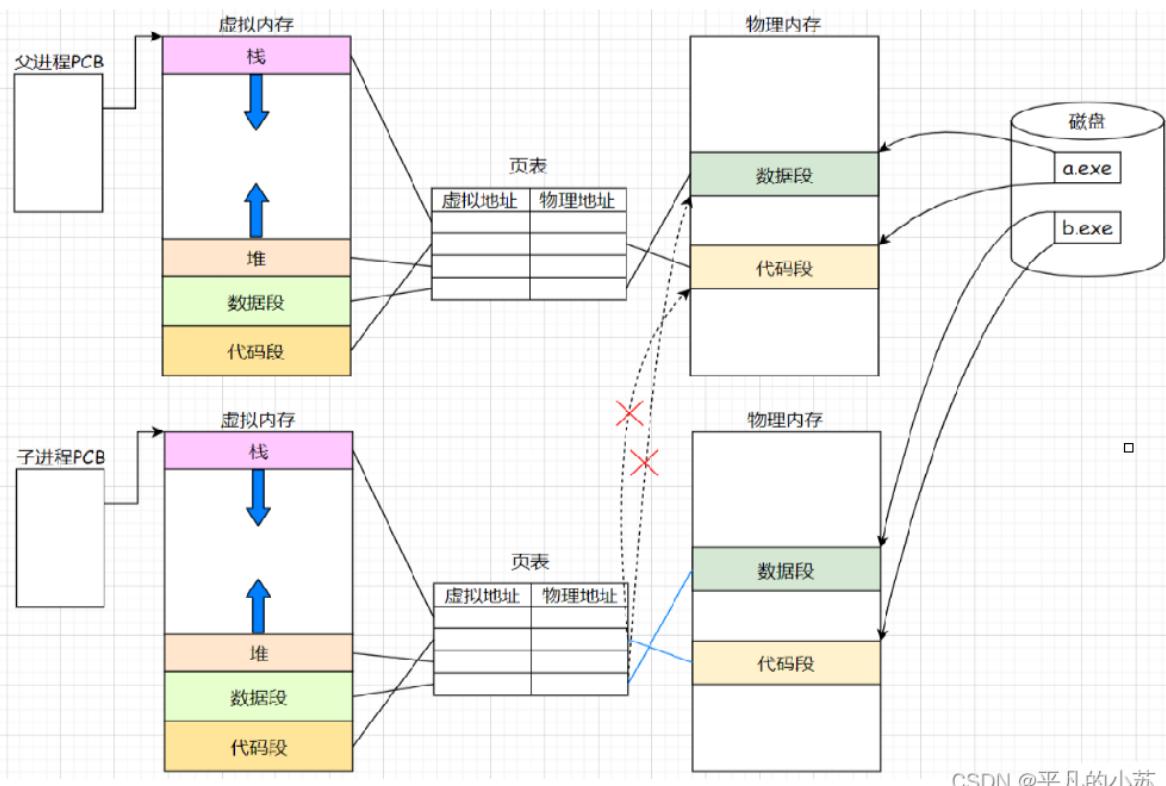
```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
    putenv((char*)"MYENV=654321");
    extern char** environ;
    pid_t id = fork();
    if(id == 0)
    {
        printf("before: I am a process, pid: %d, ppid: %d\n",
               getpid(), getppid());
        char* const myargv[] = {
            "otherExe",
            "_a",
            "_b",
            "_c",
            NULL
        };
        //使用自定义的环境变量
        char* const _env[]={(char*)"MYENV=12345",NULL};
        execle("./otherExe", "otherExe", NULL, _env);
        //extern char** environ;
        //系统环境变量不传，进程也能获取
        //execle("./otherExe", "otherExe", "-a", "-w", "-t", NULL);
        //execv("./otherExe", myargv);
        printf("after: I am a process, pid: %d, ppid:%d\n",getpid(),
               getppid());
        exit(1);
    }

    pid_t ret = waitpid(id,NULL,0);
    if(ret > 0) printf("wait success, father pid:%d, ret id: %d\n");
    sleep(5);
}
```

```
    return 0;  
}
```

4.2 进程程序替换的原理

前面我们学习到，当fork创建子进程的时候，子进程的PCB、虚拟地址空间都以父进程为模板，页表中的代码段指向的是父进程中的代码段，数据也以写时拷贝的方式来和父进程进行共享，如果现在有一个全新的程序b.exe，并且我现在不想让子进程执行任何父进程相关的代码以及访问父进程的数据，并执行的是a.exe程序，此时把b.exe的程序加载到物理内存上，让子进程重新调整自己的页表映射，使其指向新的b程序的代码和数据，这种过程就叫做程序替换。



总结程序替换的原理：

- 将磁盘中的程序，加载入内存结构

重新建立页表映射，谁执行程序替换，就重新建立谁的映射，最终达到的效果就是让父进程和子进程彻底分离，并让子进程执行一个全新的程序！！！

问1：当进行程序替换时，有没有创建新的进程？

进程序替换后，该进程对应的PCB、进程地址空间以及页表等数据结构均没有发生改变，只是重新建立了一下物理内存中的映射关系罢了，它的内核对应的数据结构没有发生变化，他的pid也没有发生变化，也就没有创建新的进程，只不过是让进程执行不同的程序罢了！！！

问2：子进程进行进程序替换后，会影响父进程的代码和数据吗？

子进程刚被创建时，与父进程共享代码和数据，但当子进程需要进行进程序替换时，也就意味着子进程需要对其数据和代码进行写入操作，这时便需要将父子进程共享的代码和数据进行写时拷贝，此后父子进程的代码和数据也就分离了，因此子进程进行程序替换后不会影响父进程的代码和数据。

4.3 exec*()系列函数的返回值

exec()函数仅在发生错误时返回。返回值为-1，设置errno以指示错误。

可以看到exec系列函数调用成功后并没有返回值，因为exec一旦被调用成功，后续代码将被覆盖，根本没机会用到返回值

5. 实现简易版本的Shell

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>

#define LEFT "["
#define RIGHT "]"
#define LABEL "#"
#define DELIM " \t"
```

```
#define LINE_SIZE 1024
#define ARGC_SIZE 32
#define EXIT_CODE 44

int lastcode = 0;
int quit = 0;
extern char** environ;
char commandline[LINE_SIZE];
char* argv[ARGC_SIZE];
char pwd[LINE_SIZE];
char myenv[LINE_SIZE];
const char* Getusername()
{
    return getenv("USER");
}

const char* Gethostname()
{
    return getenv("HOSTNAME");
}

void Getpwd()
{
    //使用系统调用获取
    getcwd(pwd,sizeof(pwd));
}

void interact(char* cline, int size)
{
    Getpwd();
    printf(LEFT"%s@%s %s"RIGHT""LABEL" ",Getusername(),Gethos
    char *s = fgets(cline,size,stdin); //输入命令
    assert(s); //断言fgets有没有出错
    (void)s; //为了没有警告
    cline[strlen(cline)-1] = '\0'; //为了将输入命令的回车符去掉
}

int splitstring(char* cline, char* argv[])

```

```

{
    int i = 0;
    argv[i++] = strtok(cline,DELIM);
    while(argv[i++] = strtok(NULL,DELIM));//字符串解析
    return i - 1;
}

void NormalExcute(char* argv[])
{
    pid_t id = fork();
    if(id < 0)
    {
        perror("fork");
        return;
    }
    else if(id == 0)
    {
        //让子进程执行命令
        execvp(argv[0],argv);
    }
    else
    {
        int status = 0;
        pid_t rid = waitpid(id,&status,0);
        if(rid == id)
        {
            lastcode = WEXITSTATUS(status);
        }
    }
}

int buildCommand(int argc, char* argv[])
{
    if(argc == 2 && strcmp(argv[0],"cd") == 0)
    {
        chdir(argv[1]);
        Getpwd();
        sprintf(getenv("PWD"), "%s", pwd);
    }
}

```

```

        return 1;
    }
else if(argc == 2 && strcmp(argv[0],"export") == 0)
{
    strcpy(myenv,argv[1]);
    putenv(myenv);
    return 1;
}
else if(argc == 2 && strcmp(argv[0],"echo") == 0)
{
    if(strcmp(argv[1],"$?") == 0)
    {
        printf("%d\n",lastcode);
        lastcode = 0;
    }
    else if(*argv[1] == '$')
    {
        char* val = getenv(argv[1] + 1);
        if(val) printf("%s\n", val);
    }
    else
    {
        printf("%s\n", argv[1]);
    }
    return 1;
}

//特殊处理ls，使它有颜色
if(strcmp(argv[0],"ls") == 0)
{
    argv[argc++] = "--color";
    argv[argc++] = NULL;
}

return 0;
}

int main()

```

```
{  
    while(!quit)  
    {  
        //交互问题，获取命令行  
        interact(commandline,sizeof(commandline));  
  
        //子串分割的问题，解析命令行  
        int argc = splitstring(commandline,argv);  
        if(argc == 0) continue;  
  
        //内建命令  
        int n = buildCommand(argc,argv);  
  
        //普通命令的执行  
        if(!n) NormalExcute(argv);  
    }  
    return 0;  
}
```