



西安邮电大学
XI'AN UNIVERSITY OF POSTS & TELECOMMUNICATIONS

Linux 编程技术



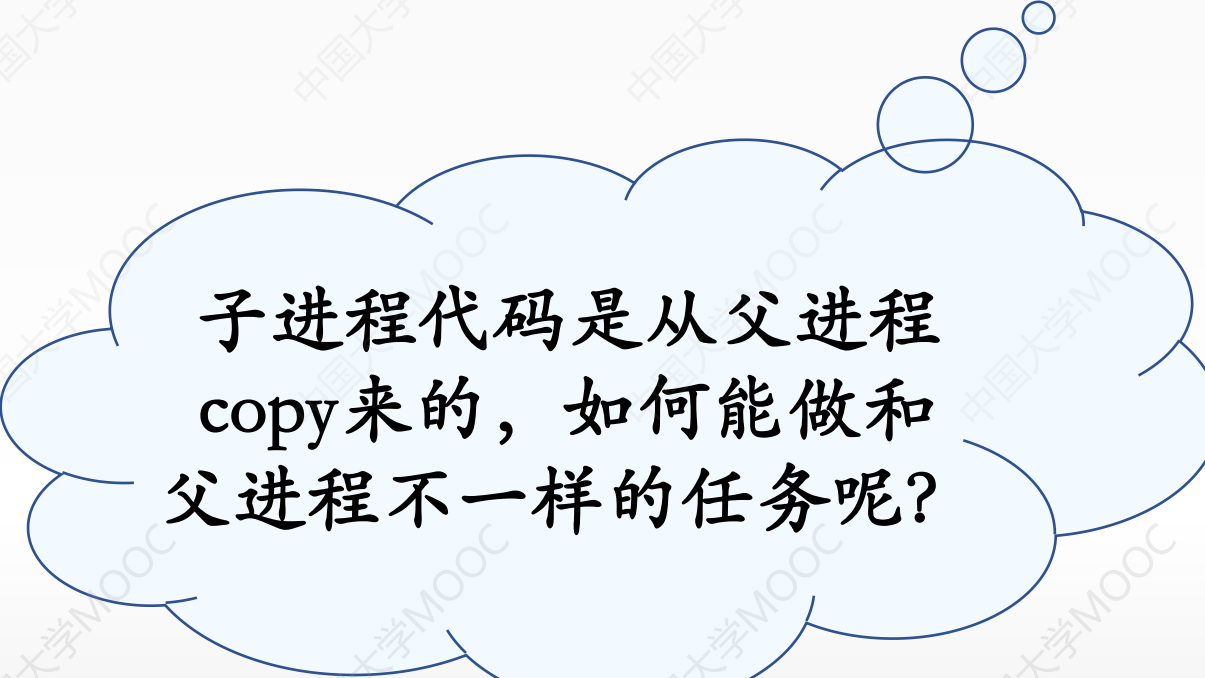
第3章 进程管理

——执行进程与vfork



- 程序中，通过fork函数创建子进程
- 区分父子进程的目的：

让父子进程完成不同的任务。



子进程代码是从父进程
copy来的，如何能做和
父进程不一样的任务呢？

eXec族函数

execvp函数的接口规范说明

execvp	
功能	运行另一个程序
头文件	/usr/include/unistd.h
函数原型	int execvp(char *file, char * argv[]);
参数	file 待运行的程序名
	argv[] 运行时的参数（以NULL结尾）
返回值	无返回值 成功
	-1 失败

以NULL
为结果，
否者报错

示例1: execvp的使用

```
//exp_execvp.c
```

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

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

```
int main() {
```

```
    char * argv[] = {"cp", "/etc/passwd", "tmppass", NULL};
```

```
    printf("Let's use execvp.\n");
```

```
    execvp("cp", argv);
```

```
    printf("*****This is the end*****");
```

```
}
```

为什么没有
输出?

```
[huangru@xiyoulinux chap3]$ ./ exp_
```

```
Let's use execvp.!
```

1. **execvp**调用并没有生成新进程。
2. 一个进程一旦调用**execvp**函数，它本身就“死亡”了，系统把代码段替换成新的程序代码，废弃原有的数据段和堆栈段，对新程序分配新的数据段和堆栈段。
3. 调用**execvp**的进程唯一保留的就是进程**ID**，对系统而言，还是同一个进程，不过执行的已经是另外一个程序了。

//come from /usr/include/unistd.h

int execv(const char * path, char * const argv[]);

int execl(const char * path, const char* arg,...);

int execl(const char * path, const char* arg,...);

int exevp(const char * file, char* const argv[]);

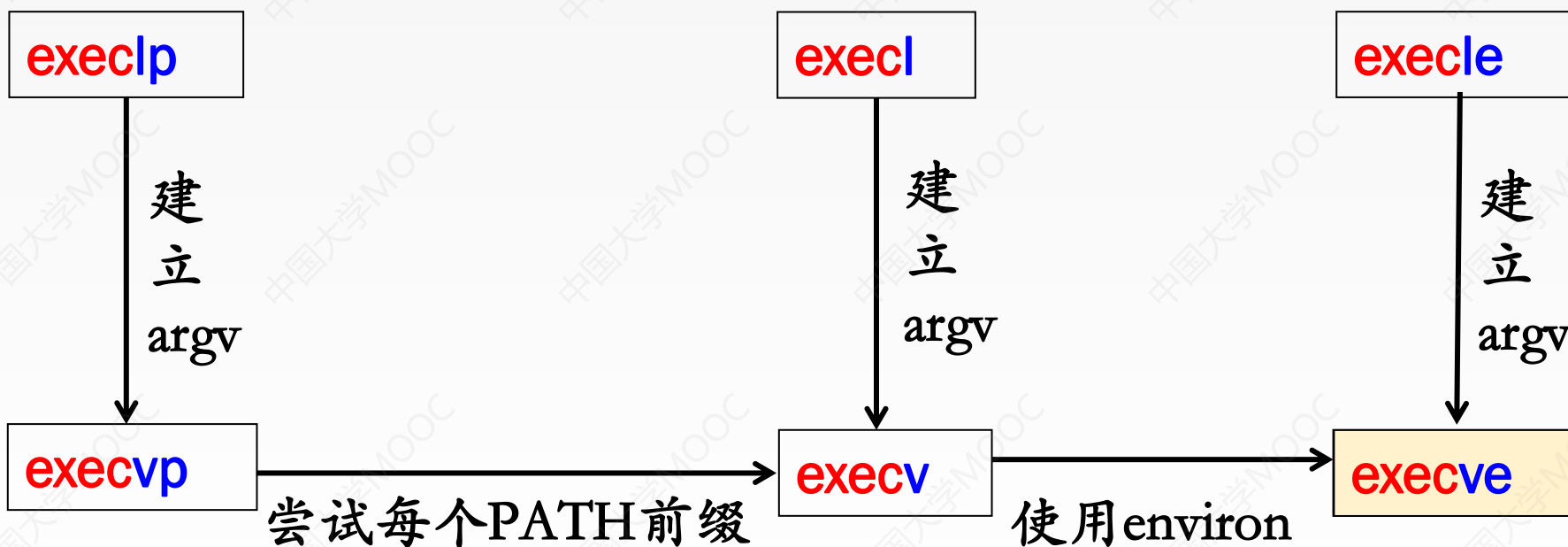
int execlp(const char * path, const char* arg,...);

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

新程序的main: **int main(int argc, char * argv[] ,char **envp);**



实际上，execvp只是eXec族函数中的一个，该族函数总共有六个。



事实上，这六个函数中真正的系统调用只有`execl_e`，其他五个都是库函数，它们最终都会调用`execl_e`这个系统调用。

在这些函数中除了exec之外的字母各有含义：

```
char * ls_argv[]={ "ls", "-1", NULL};
```

```
char * ls_envp[]={ "PATH=bin:usr/bin", "TERM=console",  
NULL};
```

➤ **l**: 以列表(list)形式列出参数;

```
execl( "/bin/ls", "ls", "-1", NULL);
```

➤ **v**: 以数组(vector)形式列出参数;

```
execv( "/bin/ls", ls_argv);
```

➤ **p**: 在系统路径PATH中查找代码程序;

```
execvp( "ls", ls_argv);
```

➤ **e**: 将环境变量设置为参数, 即把新的环境制定到新进程中。

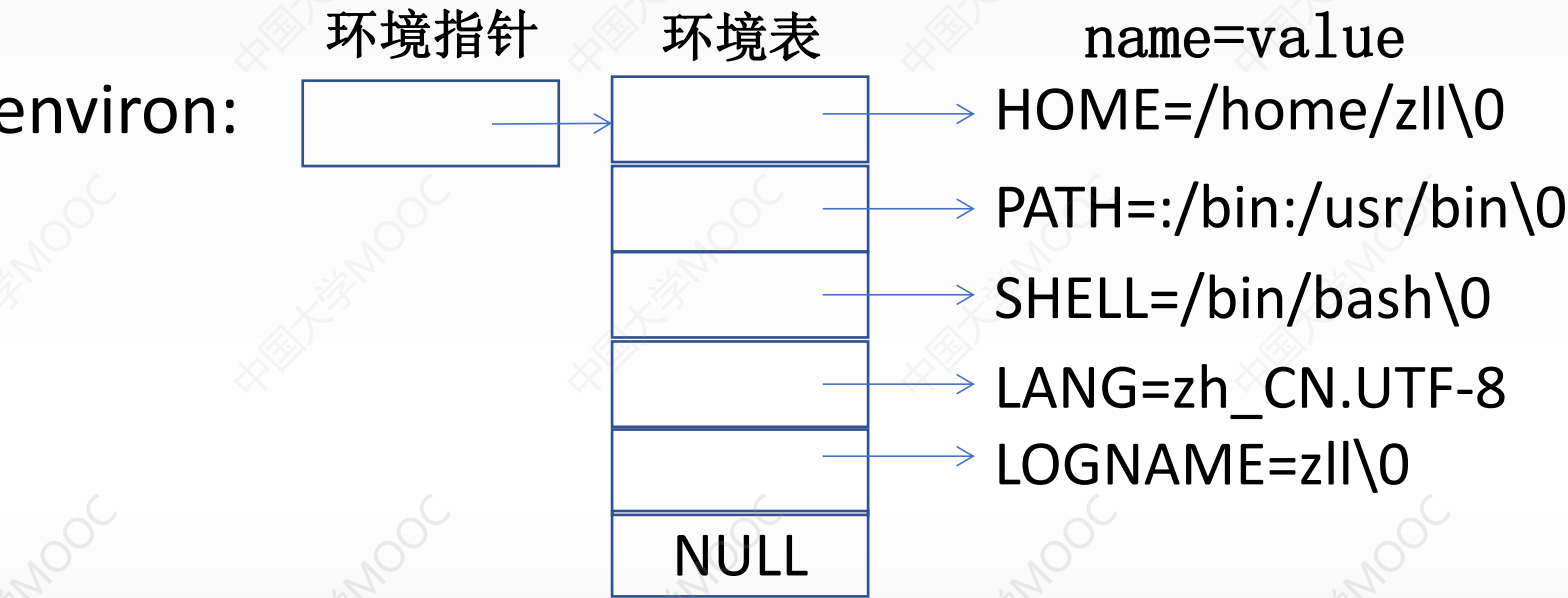
```
execve( "/bin/ls", ls_argv, ls_envp);
```


(1) 环境变量定义了用户的工作环境，包括用户的主目录、终端类型、当前目录等。

(2) 系统预定义全局变量`environ`显示各个环境变量值。

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

如：环境包含5个字符串，如图所示：



示例2：执行自定义任务

//先编写一个自定义程序hello.c

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

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

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

```
main() {
```

```
    printf("I am hello process image\n");
```

```
    printf("My pid=%d, parentpid=%d!\n",getpid(),getppid());
```

```
    printf("uid=%d,gid=%d\n", getuid(),getgid());
```

```
    return 0;
```

```
}
```

//完成后，编译为hello

```
gcc -o hello hello.c
```

示例2：执行自定义任务

// 编写父程序exec.c

```
main(int argc, char * argv[], char **environ) {  
    pid_t pid; int flag;  
    char *envp[] = {"PATH=.", NULL};  
    pid=fork();  
    switch(pid) {  
    case 0:  
        printf("in child process ..... \n");  
        printf("My pid=%d, parentpid=%d! \n", getpid(), getppid());  
        printf("uid=%d, gid=%d \n", getuid(), getgid());  
        flag=execve("hello", argv, envp);  
        .....  
    default:  
        printf("Parent process is running \n");  
        break;  
    } exit(0);  
}
```

子进程

父进程

示例2：执行自定义任务

//编译并运行父程序exec

```
gcc exec -o exec.c
```

//运行结果

```
root@ubuntu:4# ./execX_demo
```

```
Parent process is running//父进程优先调度执行
```

```
in child process .....
```

```
My pid=45980, parentpid=45979!
```

```
uid=0,gid=0
```

```
root@ubuntu:4# I am hello process image//执行hello程序
```

```
My pid=45980, parentpid=1!//父进程变为init进程
```

```
uid=0,gid=0
```

子进程的父
进程改变了?

- 父进程早于子进程结束，子进程称为孤儿进程
- 人类社会里一个孩子失去了父母。人类社会是如何安置这样的孩子呢？
- 将被init进程接收

vfork	
功能	产生一个新的进程
头文件	/usr/include/unistd.h
函数原型	pid_t vfork(void);
返回值	-1 创建失败（父进程）
	>0 子进程号（父进程）
	0 创建成功（子进程）

vfork创建一个子进程，在该子进程调用exec或exit之前，子进程将在父进程地址空间中运行，而父进程在子进程调用exec或exit前，将处于等待状态。

例3：用vfork创建进程

```
//vfrok_demo.c
```

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

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

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

```
int glob=3;
```

```
int main(){
```

```
    int var=1, i; pid_t pid;
```

```
    printf("before vfork\n");
```

```
    pid=vfork();
```

```
    switch(pid){
```

```
        case 0:
```

```
            i=3;
```

```
            while(i-->0){
```

```
                sleep(1);
```

```
                glob++;
```

```
                var++;
```

```
                printf("Child's pid=%d,glob=%d,var=%d\n", getpid(),glob,var); }
```

```
            exit(0);
```

全局变量glob，数据区；
局部变量var，栈区。

子进程修改glob
和var，并打印

case -1:

```
perror("Process creation failed\n");  
exit(0);
```

default:

```
i=3;
```

```
while(i-->0) {
```

```
    sleep(1);
```

```
    glob++;
```

```
    var++;
```

```
    printf("Parent's pid=%d,glob=%d,var=%d\n",getpid(),glob,var);}
```

```
}
```

```
exit(0);
```

```
}
```

父进程修改glob
和var，并打印。

```
root@ubuntu:4# ./vfrok_hello
```

```
before vfork
```

```
Child is runnig
```

```
Child is runnig
```

```
Child is runnig
```

```
Child's pid=45098,glob=6,var=4
```

```
Parent is running
```

```
Parent is running
```

```
Parent is running
```

```
Parent's pid=45097,glob=9,var=7//父进程读变量值
```

子进程先执行，父
进程后执行，父子
进程共享glob和
var空间。

```
root@ubuntu:4# ./frok_hello  vfork变为fork
before fork
Parent is running
Child is runnig
Parent is running
Child is runnig
Child is runnig
Parent is running
Child's pid=45106,glob=6,var=4
Parent's pid=45105,glob=;6,var=4
```

父子进程执行顺序不确定，交替执行。

父子进程glob和var空间独立。

fork和vfork区别	
fork	vfork
子进程完全复制父进程的资源，子进程独立于父进程。	子进程共享父进程地址空间（代码、数据、堆栈）。
父子进程谁先运行，由CPU调度算法决定。	保证子进程先运行。
创建进程系统开销大，写时复制技术可以缓解开销大问题。	创建进程系统开销小

注意：vfork使用需要谨慎，因父进程阻塞等待，子进程在调用exec或exit前依赖父进程行为，导致死锁；父子共享地址空间，易于造成进程间同步错误。

eXec族函数：调用进程执行新执行映像

eXec族函数使用：函数名的字符含义理解其函数具体使用方法

vfork系统调用：共享父进程地址空间

fork和vfork区别：地址空间、执行顺序

谢谢大家！

