

0914进程控制

```

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

int main()
{
    pid_t id = fork();
    if(id == 0)
    {
        //子进程
        int cnt = 5;
        while(cnt)
        {
            printf("我是子进程: %d\n",cnt);
            sleep(1);
        }
        exit(11);//仅仅用来测试
    }
    else
    {
        //父进程
        int status = 0;
        //注意: 只有子进程退出的时候, 父进程才会waitpid函数, 进行返回! 【此时, 父进程还活着呢!!!】
        pid_t result = waitpid(id, &status, 0); //默认是在阻塞状态去等待子进程状态变化 (就是退出)
        if(result > 0)
        {
            printf("父进程等待成功, 退出码: %d, 退出信号: %d\n", (status>>8)&0xFF, status&0x7F);
        }
    }
    return 0;
}

```

我们可以得出今天的第一个结论：

waitpid/wait 可以在目前的情况下，让进程退出具有一定的顺序性！
将来可以让父进程进行更多的收尾工作

其中这里的id

id > 0 等待指定进程

id == 0 我们稍后再讲

id == -1 等待任意一个子进程退出，等价于wait()

```

int status = 0;
//注意: 只有子进程退出的时候, 父进程才会waitpid函数, 进行返回! 【此时, 父进程还活着呢!!!】
pid_t result = waitpid(id, &status, 0); //默认是在阻塞状态去等待子进程状态变化 (就是退出)
if(result > 0)
{

```

```
//注意：只有子进程退出的时候，父进程才会waitpid函数，进行返回！【此时，父进程还活着呢!!!】
pid_t result = waitpid(id, &status, 0); //默认是在阻塞状态去等待子进程状态变化（就是退出）
if(result > 0)
{
    printf("父进程等待成功，退出码： %d， 退出信号： %d\n", (status>>8)&0xFF, status&0x7F);
}
return 0;
```

对于上层使用的人来说，我使用的时候还要对操作系统内核有一定了解，还得知道status的构成，我才能提取信息
这样未免太麻烦了！

其实操作系统给我们提供了宏的，其实宏的底层实现就是我们学的位运算方法

pid>0, 等待其进程ID与pid相等的子进程。

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）

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ID。

```
[yufc@VM-12-12-centos 0919]$ make
gcc -o myproc myproc.c
[yufc@VM-12-12-centos 0919]$ ./myproc
我是子进程：4
我是子进程：3
我是子进程：2
我是子进程：1
我是子进程：0
子进程执行完毕，子进程的退出码：11
[yufc@VM-12-12-centos 0919]$
```

现在我们可以这么写

```
//父进程
int status = 0;
//注意：只有子进程退出的时候，父进程才会waitpid函数，进行返回！【此时，父进程还活着呢!!!】
pid_t result = waitpid(id, &status, 0); //默认是在阻塞状态去等待子进程状态变化（就是退出）
if(result > 0)
{
    //printf("父进程等待成功，退出码： %d， 退出信号： %d\n", (status>>8)&0xFF, status&0x7F);
    if(WIFEXITED(status))
    {
        printf("子进程执行完毕，子进程的退出码： %d\n", WEXITSTATUS(status));
    }
    else
    {
        printf("子进程异常退出： %d\n", WIFEXITED(status));
    }
}
```

现在我们知道，当父进程在等待子进程死的时候，父进程是阻塞式等待，也就是说，父进程在等待的时候啥都没干，那么此时，如果我们想让父进程继续去做一些事情呢？

**此时我们需要设置第三个参数，
第三个参数默认为0，表示阻塞等待！**

WNOHANG表示非阻塞等待

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ID。

阻塞等待和非阻塞等待：

非阻塞等待：

如果父进程检测子进程的退出状态，发现子进程没有退出，我们的父进程通过调用waitpid来进行等待。

如果子进程没有退出，我们waitpid这个系统调用立马返回！

这种阻塞等待，其实是在系统调用接口中阻塞
也就是内核中阻塞

比如以前我们的scanf, cin
我们从不从键盘输入

发现缓冲区里面资源没有就绪
系统就把我们的进程挂起！
此时，表面上看，就是卡住了！

而在哪里卡住？
由冯诺依曼体系我们知道，
键盘这些输入硬件，到显示器这种输出硬件中，肯定会经过操作系统
所以其实本质上也是在内核中卡住了！
Scanf, cin必定封装了系统调用！

一般都是内核中阻塞，等待被唤醒

非阻塞等待！

阻塞等待和非阻塞等待：

伴随着被切换！

CPU本来就很忙！

要么是在阻塞队列中，要么是等待被调度！

我们的父进程通过调用waitpid来进行等待，如果子进程没有退出，我们waitpid这个系统调用，立马返回！

父进程调用

```
waitpid (child_id, status, flag);
```

EIP寄存器

// 下面是内核中waitpid的实现，属于操作系统的
// 检测子进程退出状态，查你的task_struct 中子进程的运行信息
if(status == 退出){
 return child_pid; status |= child->sig_number; status |= ((child->exit_code)<8)
}
else if(status == 没退出){
 if(flag == 0) 挂起父进程; // father_pcb->等待队列中
 else if(flag == WNOHANG) return 0; //不阻塞进程!
 return 0;
}
else{
 // 出错了等其他原因

等待完成了，从这里继续向下执行！

阻塞时，在这里卡住！

进程阻塞，本质是在内核中阻塞！

进程阻塞，本质，是进程阻塞在系统函数的内部！

后面的代码不执行了

当条件满足的时候，父进程被唤醒，从哪里唤醒？
waitpid重新调用，还是从if的后面

如果不是阻塞等待
在这里直接返回，不用把父进程的pcb放到等待队列中！

20:06 2022/9/19

那么非阻塞调用的时候，直接就返回了。

那我们怎么知道子进程的状态呢？

我们会每隔一段时间去检测一下子进程的状态，如果监测到子进程结束，就释放子进程
这个叫做 --- 基于非阻塞调用的轮询检测方案

为什么我们要重点学习阻塞和非阻塞？

因为我们未来编写代码的内容，大部分是网络代码，大部分都是IO类别，不断面临阻塞和非阻塞接口！

```
//下面的方案是处理非阻塞调用的
int quit = 0;
while(!quit)
{
    int status = 0;
    pid_t res = waitpid(-1,&status,WNOHANG); //以非阻塞方式等待
    if(res > 0)
    {
        //等待成功 -- 子进程退出
        printf("等待子进程退出, 退出码: %d\n", WEXITSTATUS(status));
        quit = 1; //让循环break一下
    }
    else if(res == 0)
    {
        //等待成功 -- 但是子进程并未退出
        printf("子进程还在运行中, 暂时还没有退出, 父进程可以再等一等, 处理一下其他事情\n");
    }
    else
    {
        //等待失败
        //其实就是waitpid等待失败, 比如我们id写错了, 写了1234, 但没有1234这个进程
        printf("wait失败!\n");
        quit = 1; //让循环break一下
    }
    sleep(1);
    printf("此时父进程还可以做事情! \n");
}
#endif
```

这次，父进程在等子进程的时候，并没有被挂起！
而是可以做其他事情！
这就是非阻塞等待！

```
[yufc@VM-12-12-centos 0919]$ ./myproc
子进程还在运行中, 暂时还没有退出, 父进程可以再等一等, 处理一下其他事情
我是子进程: 4
此时父进程还可以做事情!
子进程还在运行中, 暂时还没有退出, 父进程可以再等一等, 处理一下其他事情
我是子进程: 3
此时父进程还可以做事情!
我是子进程: 2
子进程还在运行中, 暂时还没有退出, 父进程可以再等一等, 处理一下其他事情
此时父进程还可以做事情!
我是子进程: 1
子进程还在运行中, 暂时还没有退出, 父进程可以再等一等, 处理一下其他事情
我是子进程: 0
此时父进程还可以做事情!
子进程还在运行中, 暂时还没有退出, 父进程可以再等一等, 处理一下其他事情
此时父进程还可以做事情!
子进程还在运行中, 暂时还没有退出, 父进程可以再等一等, 处理一下其他事情
此时父进程还可以做事情!
等待子进程退出, 退出码: 11
此时父进程还可以做事情!
[yufc@VM-12-12-centos 0919]$
```


再写一个让父进程做点别的事的例子：

```
#include<unistd.h>
#include<iostream>
#include<vector>
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>

typedef void (*handler_t)(); //函数指针类型

std::vector<handler_t> handlers; //函数指针数组

void fun_one()
{
    std::cout<<"这是任务1"<<std::endl;
}
void fun_two()
{
    std::cout<<"这是任务2"<<std::endl;
}
//设置对应的方法回调
void Load()
{
    handlers.push_back(fun_one);
    handlers.push_back(fun_two);
}
```

以后想让父进程闲着的时候执行任何方法的时候，只需要向Load里面注册，就可以让父进程执行对应的方法了！

```
int main()
{
    pid_t id = fork();
    if(id == 0)
    {
        //子进程
        int cnt = 5;
        while(cnt-->0)
        {
            printf("我是子进程: %d\n",cnt);
            sleep(1);
        }
        exit(11); //仅仅用来测试
    }
    else
    {
        //下面的方案是处理非阻塞调用的
        int quit = 0;
        while(!quit)
        {
            int status = 0;
            pid_t res = waitpid(-1,&status,WNOHANG); //以非阻塞方式等待
            if(res > 0)
            {
                //等待成功 -- 子进程退出
                printf("等待子进程退出, 退出码: %d\n",WEXITSTATUS(status));
                quit = 1; //让循环break一下
            }
        }
    }
}
```

此时，我们就通过了回调的方式，让父进程在等待子进程的时候，处理一些问题和事情

```
else if(res == 0)
{
    //等待成功 -- 但是子进程并未退出
    printf("子进程还在运行中, 暂时还没有退出, 父进程可以再等一等, 处理一下其他事情\n");
    //此时我们父进程做点别的事情呗
    if(handlers.empty())
    {
        Load();
    }
    for(auto iter : handlers)
    {
        //执行处理其他任务
        iter();
    }
}
else
{
    //等待失败
    //其实就是waitpid等待失败, 比如我们id写错了, 写了1234, 但没有1234这个进程
    printf("wait失败!\n");
    quit = 1; //让循环break一下
}
sleep(1);
}

//if 0
//下面的方案是处理阻塞调用的!
//父进程
int status = 0;
//注意: 只有子进程退出的时候, 父进程才会waitpid函数, 进行返回! 【此时, 父进程还活着呢!!!】
pid_t result = waitpid(id, &status, 0); //默认是在阻塞状态去等待子进程状态变化 (就是退出)
if(result > 0)
{
    //printf("父进程等待成功, 退出码: %d, 退出信号: %d\n", (status>>8)&0xFF, status&0x7F);
    if(WIFEXITED(status))
    {
        printf("子进程执行完毕, 子进程的退出码: %d\n", WEXITSTATUS(status));
    }
    else
    {
        printf("子进程异常退出: %d\n", WIFEXITED(status));
    }
}
#endif
}
return 0;
}
```

```
• [yufc@VM-12-12-centos 0919]$ ./myproc
子进程还在运行中，暂时还没有退出，父进程可以再等一等，处理一下其他事情
我是子进程：4
这是任务1
这是任务2
我是子进程：3
子进程还在运行中，暂时还没有退出，父进程可以再等一等，处理一下其他事情
这是任务1
这是任务2
我是子进程：2
子进程还在运行中，暂时还没有退出，父进程可以再等一等，处理一下其他事情
这是任务1
这是任务2
我是子进程：1
子进程还在运行中，暂时还没有退出，父进程可以再等一等，处理一下其他事情
这是任务1
这是任务2
我是子进程：0
子进程还在运行中，暂时还没有退出，父进程可以再等一等，处理一下其他事情
这是任务1
这是任务2
子进程还在运行中，暂时还没有退出，父进程可以再等一等，处理一下其他事情
这是任务1
这是任务2
等待子进程退出，退出码：11
○ [yufc@VM-12-12-centos 0919]$
```

以后想让父进程闲着的时候执行任何方法的时候，只需要向Load里面注册，就可以让父进程执行对应的方法了！

进程替换

1. 是什么？概念+原理

2. 怎么办？操作

3. 为什么？总结

我们以前学：

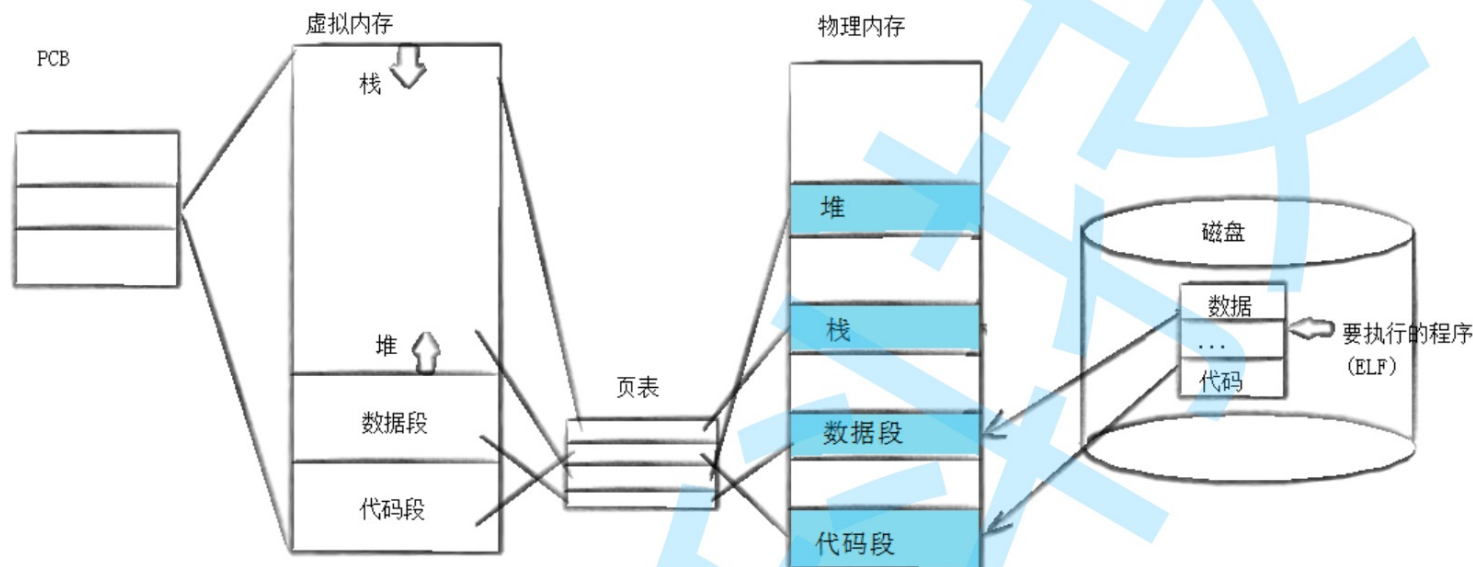
fork()之后，父子各自执行进程代码的一部分

如果子进程就像执行一个全新的程序呢？

进程替换，是通过特定的接口，加载到磁盘上的一个权限的程序（代码和数据），加载到调用进程的地址空间中！

替换原理

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

其实有六种以exec开头的函数,统称exec函数:

```
#include <unistd.h>

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

1. 进程替换, 有没有创建新的进程?
没有! 因为pcb这些结构没变!

2. 如何理解所谓的将程序放入内存中?
本质就是加载!

我们学习的所谓的exec系列的函数
本质
就是如何加载程序的函数

2. 怎么办？操作

a. 1. 不创建子进程

2. 创建子进程（只使用最简单的exec函数） -- 见见是啥东西

b. 详细展开其他函数的用法！

```
EXEC(3)                                Linux Programmer's Manual                                EX
```

NAME

execl, execlp, execl, 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 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[]);`

课件上是6个，其实有7个

不过，只要会用一个，其他的都很快能学会！

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

path: 路径+目标文件名

...: 可变参数列表

这里的可变参数列表：
我们在命令行是怎么执行的
这里的参数就按顺序一个一个填上去
而且
最后一个参数，必须是NULL，表示参数传递完毕！

```
int main()
{
    printf("当前进程的开始代码!\n");
    // 我们想在代码中执行一个名为ls的可执行程序
    // execl("/usr/bin/ls", "ls", NULL);
    execl("/usr/bin/ls", "ls", "-a", "-l", NULL);
    printf("当前进程的结束代码!\n");
    return 0;
}
```

```
• [yufc@VM-12-12-centos 0919]$ make
  g++ -std=c++11 -o myproc myproc2.cc
• [yufc@VM-12-12-centos 0919]$ ./myproc
当前进程的开始代码！ ← 相当于执行了 ls -al
total 32
drwxrwxr-x 2 yufc yufc 4096 Jan 17 19:08 .
drwxrwxr-x 8 yufc yufc 4096 Jan 17 15:45 ..
-rw-rw-r-- 1 yufc yufc  77 Jan 17 19:06 Makefile
-rwxrwxr-x 1 yufc yufc 8448 Jan 17 19:08 myproc
-rw-rw-r-- 1 yufc yufc  402 Jan 17 19:08 myproc2.cc
-rw-rw-r-- 1 yufc yufc 2960 Jan 17 18:54 myproc.cc
• [yutc@VM-12-12-centos 0919]$
```

```
• [yufc@VM-12-12-centos 0919]$ which ls
alias ls='ls --color=auto'
/usr/bin/ls
○ [yufc@VM-12-12-centos 0919]$
```

不过我们在程序里面调用的ls其实是
带了 控制颜色 的选项的

所以我们在代码上加一个选项，我们也有颜色了

```
• [yufc@VM-12-12-centos 0919]$ make
g++ -std=c++11 -o myproc myproc2.cc
• [yufc@VM-12-12-centos 0919]$ ./myproc
当前进程的开始代码！
total 32
drwxrwxr-x 2 yufc yufc 4096 Jan 17 19:12 .
drwxrwxr-x 8 yufc yufc 4096 Jan 17 15:45 ..
-rw-rw-r-- 1 yufc yufc 77 Jan 17 19:06 Makefile
-rwxrwxr-x 1 yufc yufc 8448 Jan 17 19:12 myproc
-rw-rw-r-- 1 yufc yufc 417 Jan 17 19:12 myproc2.cc
-rw-rw-r-- 1 yufc yufc 2960 Jan 17 18:54 myproc.cc
○ [yufc@VM-12-12-centos 0919]$
```

但是我们发现一个问题：

为什么

当进程的结束代码这句话为什么不
打印？？？

原因是：

execl是程序替换，调用该函数成功之后，
会将当前进程所有的代码和数据都进行
替换！包括已经执行的和没有执行的！

一旦调用成功，后续所有的代码都不会
执行！

```

int main()
{
    printf("当前进程的开始代码!\n");
    //我们想在代码中执行一个名为ls的可执行程序
    // execl("/usr/bin/ls" "ls",NULL);
    execl("/usr/bin/lssss" "ls","--color=auto","-a","-l",NULL);
    printf("当前进程的结束代码!\n");
    return 0;
}

```

如果函数调用失败

```

[yufc@VM-12-12-centos 0919]$ ./myproc
当前进程的开始代码！
当前进程的结束代码！
[yufc@VM-12-12-centos 0919]$

```

则不会被替换，这些都很好理解！

所以我们使用execl的时候

根本也不需要对其返回值做判断

如果调用失败了，则直接在后面写个exit(1) 退出码是1 即可

如果调用成功了，exit(1)也不会被执行。

在这里我们根本不需要判断返回值！

RETURN VALUE
The `exec()` functions return only if an error has occurred. The return value is -1, and `errno` is set to indicate the error.

只有调用失败了才会有返回值！

ERRORS
All of these functions may fail and set `errno` for any of the errors specified for `execve(2)`.

为什么只有调用失败才有返回值呢？

这个也很简单 --- 都调用成功了，代码和数据都被替换了（包括execl自己）