# 进程控制

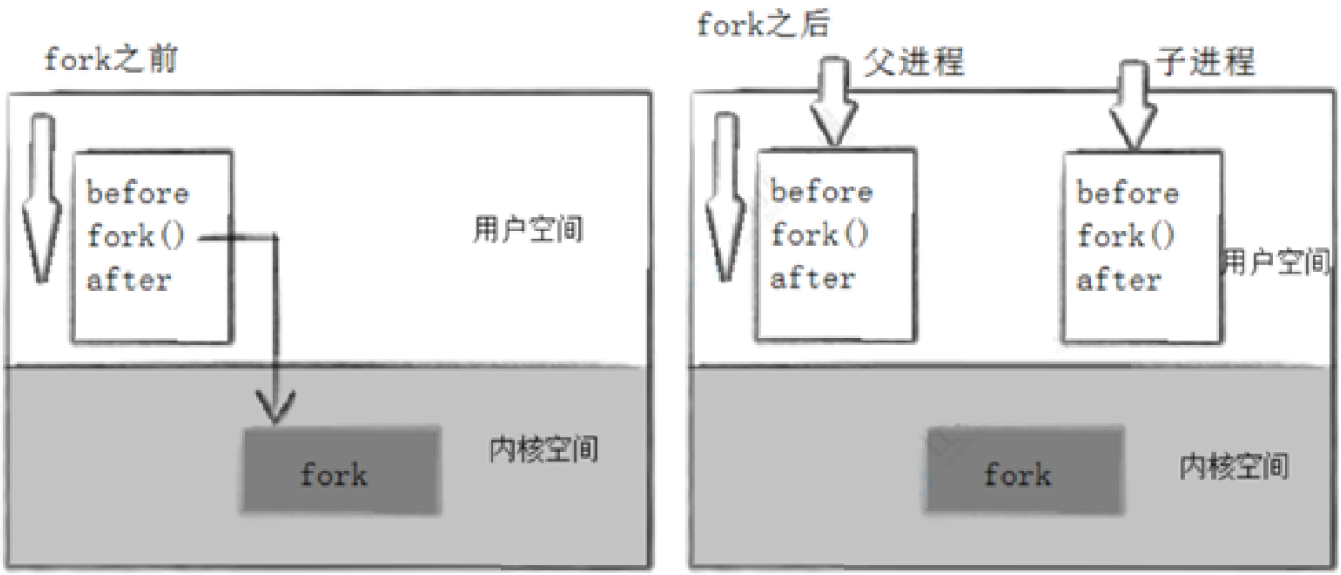
## 1. 进程创建

### 1-1 fork函数初识

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， 而原进程为父进程。

**进程调用 fork ，当控制转移到内核中的 fork 代码后，内核做：**

- 分配新的内存块和内核数据结构给子进程
- 将父进程部分数据结构内容拷贝至子进程
- 添加子进程到系统进程列表当中
- fork 返回，开始调度器调度



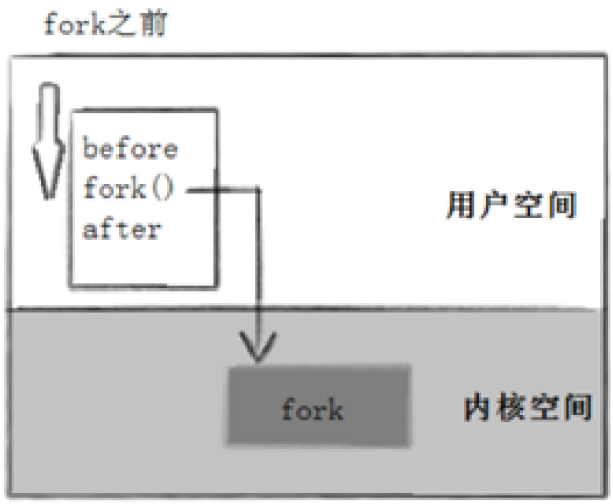

当一个进程调用fork之后，就有两个二进制代码相同的进程。而且它们都运行到相同的地方。但每个进程都将可以开始它们自己的旅程，看如下程序。

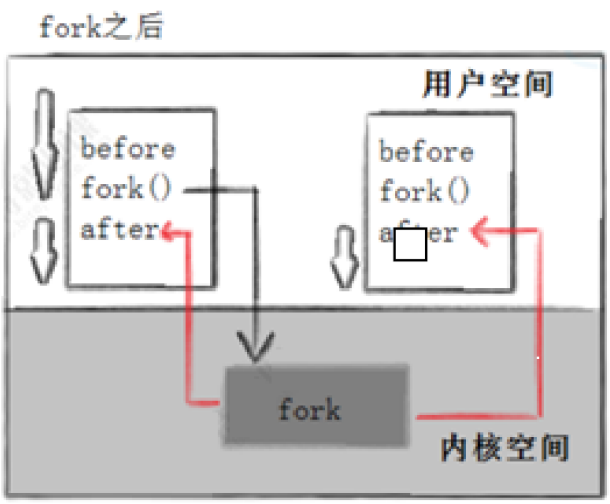
代码块

```
1   int main(void)
2   {
3       pid_t pid;
4       printf("Before: pid is %d\n", getpid());
5
6       if ((pid = fork()) == -1)
```

```
7           perror("fork()"), exit(1);
8
9       printf("After:pid is %d, fork return %d\n", getpid(), pid);
10
11      sleep(1);
12      return 0;
13  }
14
15  运行结果:
16  [root @localhost linux] #./ a.out
17  Before : pid is 43676
18  After : pid is 43676, fork return 43677
19  After : pid is 43677, fork return 0
```

这里看到了三行输出，一行before，两行after。进程43676先打印before消息，然后它又打印after。另一个after消息有43677打印的。注意到进程43677没有打印before，为什么呢？如下图所示：



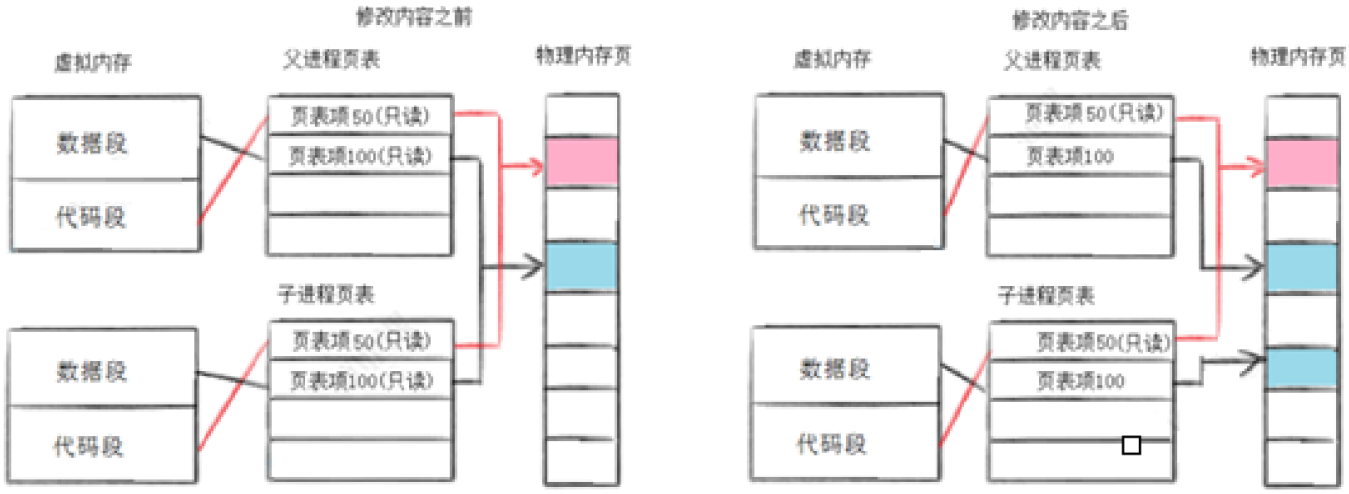

所以，fork之前父进程独立执行，fork之后，父子两个执行流分别执行。注意，fork之后，谁先执行完全由调度器决定。

## *1-2 fork函数返回值*

- 子进程返回0
- 父进程返回的是子进程的pid。

## *1-3 写时拷贝*

通常，父子代码共享，父子再不写入时，数据也是共享的，当任意一方试图写入，便以写时拷贝的方式各自一份副本。具体见下图:




## 1-4 fork常规用法

· 一个父进程希望复制自己，使父子进程同时执行不同的代码段。例如，父进程等待客户端请求，生成子进程来处理请求。

· 一个进程要执行一个不同的程序。例如子进程从fork返回后，调用exec函数。

## 1-5 fork调用失败的原因

· 系统中有太多的进程

· 实际用户的进程数超过了限制

# 2. 进程终止

进程终止的本质是释放系统资源，就是释放进程申请的相关内核数据结构和对应的数据和代码。

## 2-1 进程退出场景

· 代码运行完毕，结果正确

· 代码运行完毕，结果不正确

· 代码异常终止

## 2-2 进程常见退出方法

**正常终止（可以通过 echo $? 查看进程退出码）：**

1. 从main返回
2. 调用exit
3. _exit

**异常退出：**

· ctrl + c，信号终止

## 2-2-1 退出码

退出码（退出状态）可以告诉我们最后一次执行的命令的状态。在命令结束以后，我们可以知道命令是成功完成的还是以错误结束的。其基本思想是，程序返回退出代码 0 时表示执行成功，没有问题。

代码 1 或 0 以外的任何代码都被视为不成功。

Linux Shell 中的主要退出码：

| 退出码 | 解释 |
| --- | --- |
| 0 | 命令成功执行 |
| 1 | 通用错误代码 |
| 2 | 命令（或参数）使用不当 |
| 126 | 权限被拒绝（或）无法执行 |
| 127 | 未找到命令，或 PATH 错误 |
| 128+n | 命令被信号从外部终止，或遇到致命错误 |
| 130 | 通过 Ctrl+C 或 SIGINT 终止（终止代码 2 或键盘中断） |
| 143 | 通过 SIGTERM 终止（默认终止） |
| 255/* | 退出码超过了 0-255 的范围，因此重新计算（LCTT 译注：超过 255 后，用退出取模） |

> · 退出码 0 表示命令执行无误，这是完成命令的理想状态。
>
> · 退出码 1 我们也可以将其解释为 “不被允许的操作”。例如在没有 sudo 权限的情况下使用 yum；再例如除以 0 等操作也会返回错误码 1，对应的命令为 let a=1/0
>
> · 130 （SIGINT 或 ^C）和 143 （SIGTERM）等终止信号是非常典型的，它们属于 128+n 信号，其中 n 代表终止码。
>
> · 可以使用 strerror 函数来获取退出码对应的描述。

## 2-3-2 _exit函数

```
#include <unistd.h>
void _exit(int status);
参数：status 定义了进程的终止状态，父进程通过wait来获取该值
```

· 说明：虽然status是int，但是仅有低8位可以被父进程所用。所以_exit(-1)时，在终端执行$?发现返回值是255。

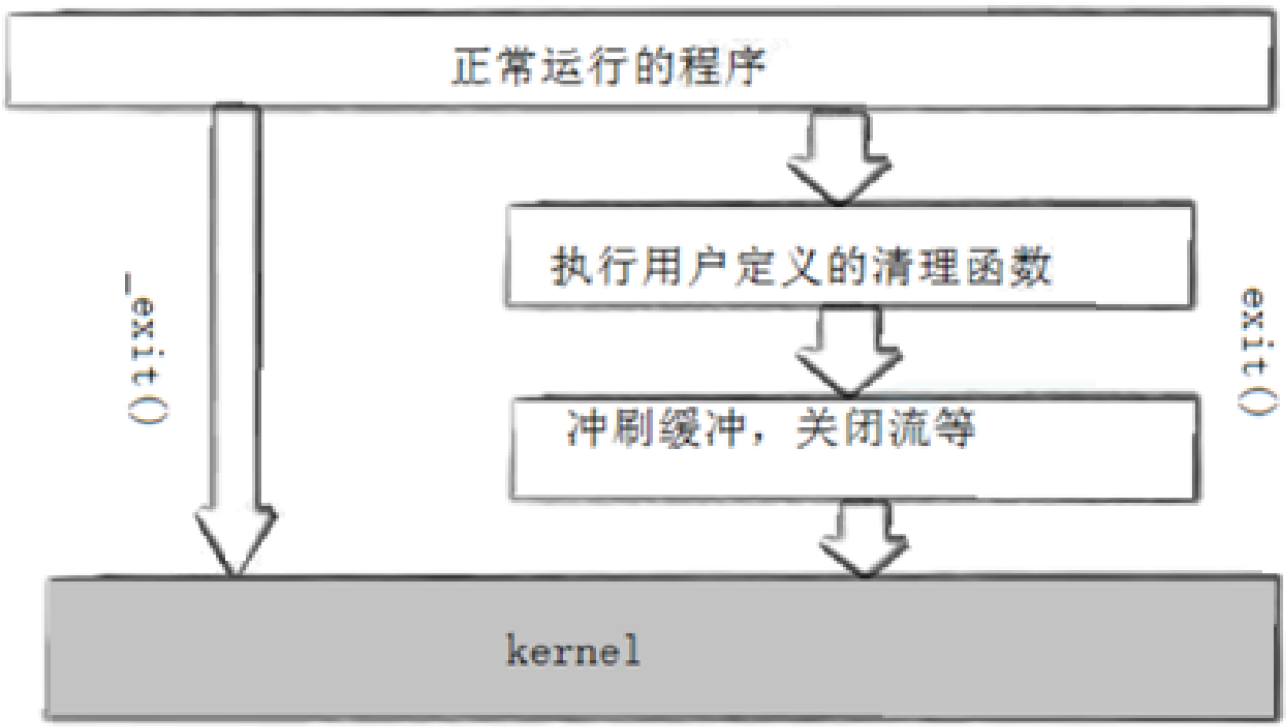
## 2-3-3 exit函数

代码块

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

exit最后也会调用_exit, 但在调用_exit之前，还做了其他工作：

1. 执行用户通过 atexit或on_exit定义的清理函数。
2. 关闭所有打开的流，所有的缓存数据均被写入
3. 调用_exit




代码块

```
int main()
```

```
{
    printf("hello");
    exit(0);
}

运行结果:
[root@localhost linux]# ./a.out
hello[root@localhost linux]#

int main()
{
    printf("hello");
    _exit(0);
}
运行结果:
[root@localhost linux]# ./a.out
[root@localhost linux]#
```

### 2-3-4 return退出

return是一种更常见的退出进程方法。执行return 等同于执行exit(n),因为调用main的运行时函数会将main的返回值当做 exit的参数。

# 3. 进程等待

## 3-1 进程等待必要性

· 之前讲过，子进程退出，父进程如果不管不顾，就可能造成‘僵尸进程’的问题，进而造成内存泄漏。

· 另外，进程一旦变成僵尸状态，那就刀枪不入，“杀人不眨眼”的kill -9 也无能为力，因为谁也没有办法杀死一个已经死去的进程。

· 最后，父进程派给子进程的任务完成的如何，我们需要知道。如，子进程运行完成，结果对还是不对，或者是否正常退出。

· 父进程通过进程等待的方式，回收子进程资源，获取子进程退出信息

## 3-2 进程等待的方法

### 3-2-1 wait方法

```
代码块
```

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

pid_t wait(int* status);

返回值:
    成功返回被等待进程pid，失败返回-1。

参数:
    输出型参数，获取子进程退出状态,不关心则可以设置成为NULL
```

## 3-2-2 waitpid方法

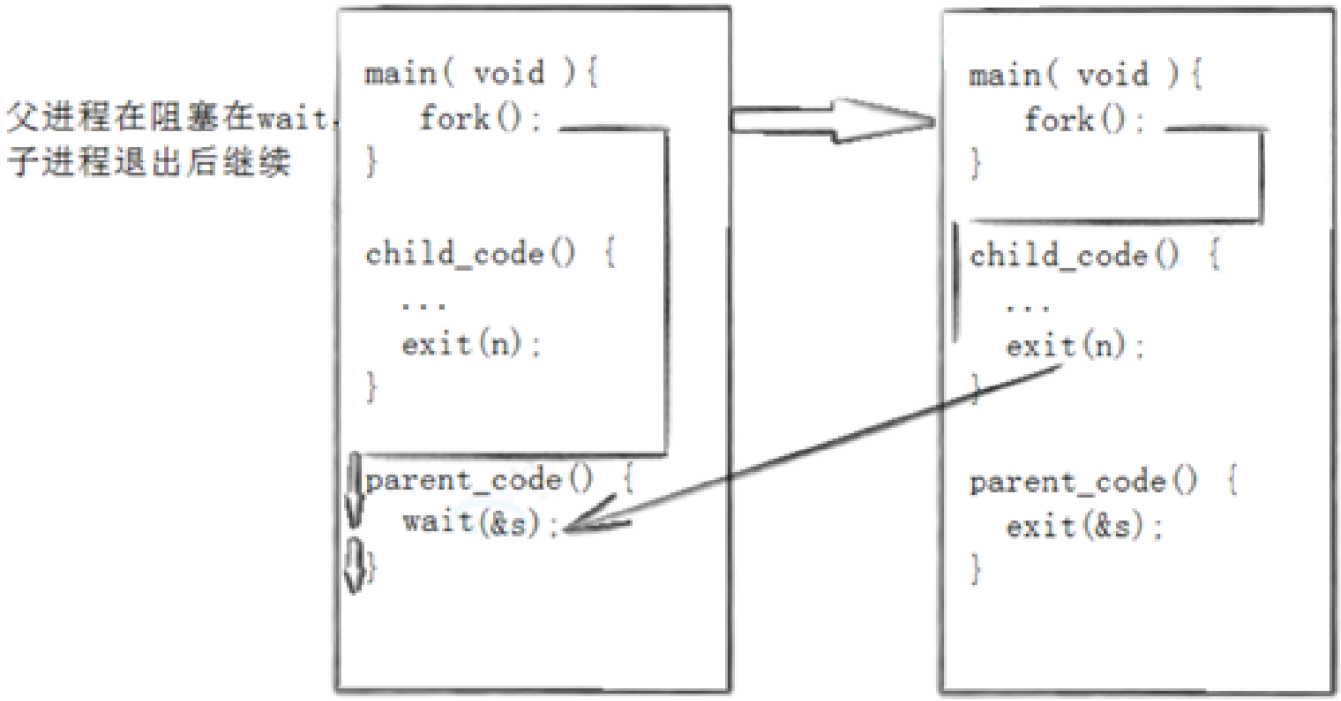
代码块

```
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ID;
    如果设置了选项WNOHANG,而调用中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，等待任意一个子进程。与wait等效。
        Pid > 0，等待其进程ID与pid相等的子进程。
    status: 输出型参数
        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: 默认为0，表示阻塞等待
        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。
```

· 如果子进程已经退出，调用wait/waitpid时，wait/waitpid会立即返回，并且释放资源，获得子进程退出信息。

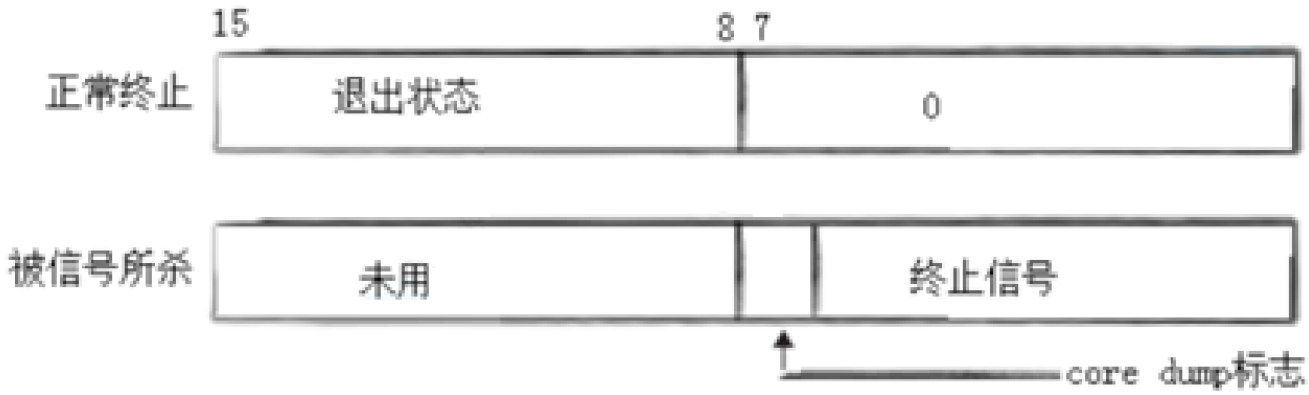
· 如果在任意时刻调用wait/waitpid，子进程存在且正常运行，则进程可能阻塞。

· 如果不存在该子进程，则立即出错返回。




## 3-2-3 获取子进程status

- 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不能简单的当作整形来看待，可以当作位图来看待，具体细节如下图（只研究status低16比特位）：




代码块

```
1   #include <sys/wait.h>
2   #include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main(void)
{
    pid_t pid;
    if ((pid = fork()) == -1)
        perror("fork"), exit(1);

    if (pid == 0)
    {
        sleep(20);
        exit(10);
    }

    else
    {
        int st;
        int ret = wait(&st);
        if (ret > 0 && (st & 0X7F) == 0)
        { // 正常退出
            printf("child exit code:%d\n", (st >> 8) & 0XFF);
        }

        else if (ret > 0)
        { // 异常退出
            printf("sig code : %d\n", st & 0X7F);
        }
    }
}
```

**测试结果：**

```
#./ a.out #等20秒退出
child exit code : 10
#./ a.out #在其他终端kill掉
sig code : 9
```

## 3-2-4 阻塞与非阻塞等待

· 进程的阻塞等待方式：

代码块

```c
int main()
{
    pid_t pid;
```

```
    pid = fork();
    if (pid < 0)
    {
        printf("%s fork error\n", __FUNCTION__);
        return 1;
    }

    else if (pid == 0)
    { // child
        printf("child is run, pid is : %d\n", getpid());
        sleep(5);
        exit(257);
    }

    else
    {
        int status = 0;
        pid_t ret = waitpid(-1, &status, 0); // 阻塞式等待，等待5S
        printf("this is test for wait\n");
        if (WIFEXITED(status) && ret == pid)
        {
            printf("wait child 5s success, child return code is
                    :%d.\n",WEXITSTATUS(status));
        }

        else
        {
            printf("wait child failed, return.\n");
            return 1;
        }
    }

    return 0;
}

运行结果：
[root@localhost linux]# ./a.out
child is run, pid is : 45110
this is test for wait
wait child 5s success, child return code is :1.
```

· 进程的非阻塞等待方式：

代码块

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

```cpp
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <vector>
typedef void (*handler_t)();      // 函数指针类型
std::vector<handler_t> handlers; // 函数指针数组

void fun_one()
{
    printf("这是一个临时任务1\n");
}

void fun_two()
{
    printf("这是一个临时任务2\n");
}

void Load()
{
    handlers.push_back(fun_one);
    handlers.push_back(fun_two);
}

void handler()
{
    if (handlers.empty())
        Load();

    for (auto iter : handlers)
        iter();
}

int main()
{
    pid_t pid;
    pid = fork();

    if (pid < 0)
    {
        printf("%s fork error\n", __FUNCTION__);
        return 1;
    }

    else if (pid == 0)
    { // child
        printf("child is run, pid is : %d\n", getpid());
        sleep(5);
```

```
49          exit(1);
50      }
51
52      else
53      {
54          int status = 0;
55          pid_t ret = 0;
56
57          do
58          {
59              ret = waitpid(-1, &status, WNOHANG); // 非阻塞式等待
60              if (ret == 0)
61              {
62                  printf("child is running\n");
63              }
64              handler();
65          }
66          while (ret == 0);
67
68          if (WIFEXITED(status) && ret == pid)
69          {
70              printf("wait child 5s success, child return code is :%d.\n",
71                     WEXITSTATUS(status));
72          }
73
74          else
75          {
76              printf("wait child failed, return.\n");
77              return 1;
78          }
79      }
80
81      return 0;
82  }
```

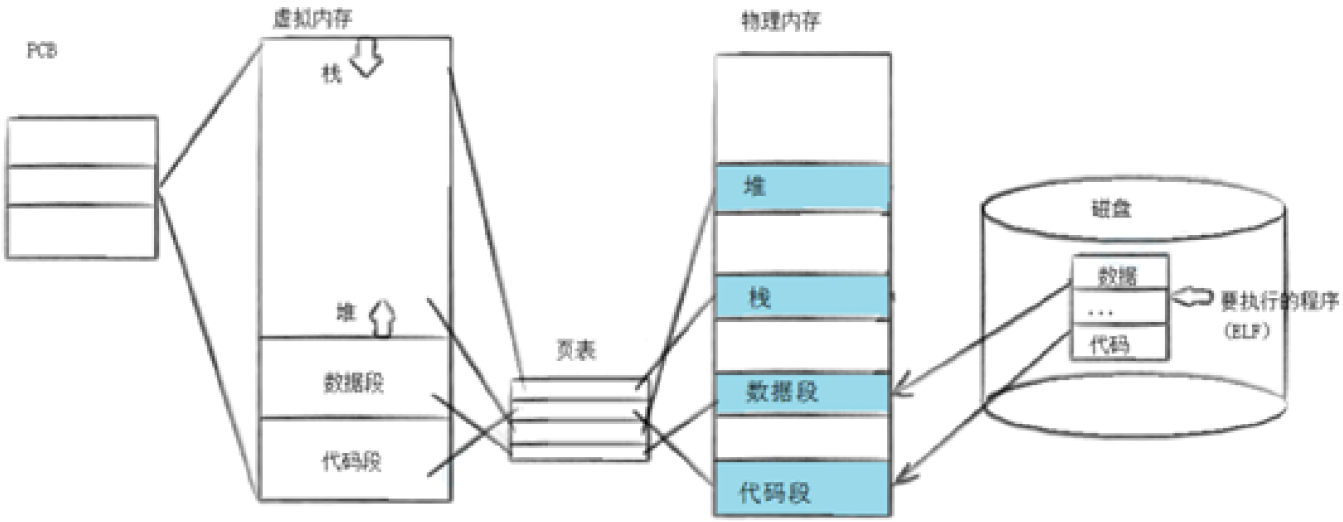
# 4. 进程程序替换

fork() 之后,父子各自执行父进程代码的一部分如果子进程就想执行一个全新的程序呢？进程的程序 替换来完成这个功能!

程序替换是通过特定的接口，加载磁盘上的一个全新的程序(代码和数据)，加载到调用进程的地址空间中!

## 4-1 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。




## 4-2 替换函数

其实有六种以exec开头的函数,统称exec函数:

代码块

```
#include <unistd.h>
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 execve(const char *path, char *const argv[], char *const envp[]);
```

### 4-2-1 函数解释

- 这些函数如果调用成功则加载新的程序从启动代码开始执行,不再返回。
- 如果调用出错则返回 -1
- 所以 exec 函数只有出错的返回值而没有成功的返回值。

### 4-2-2 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

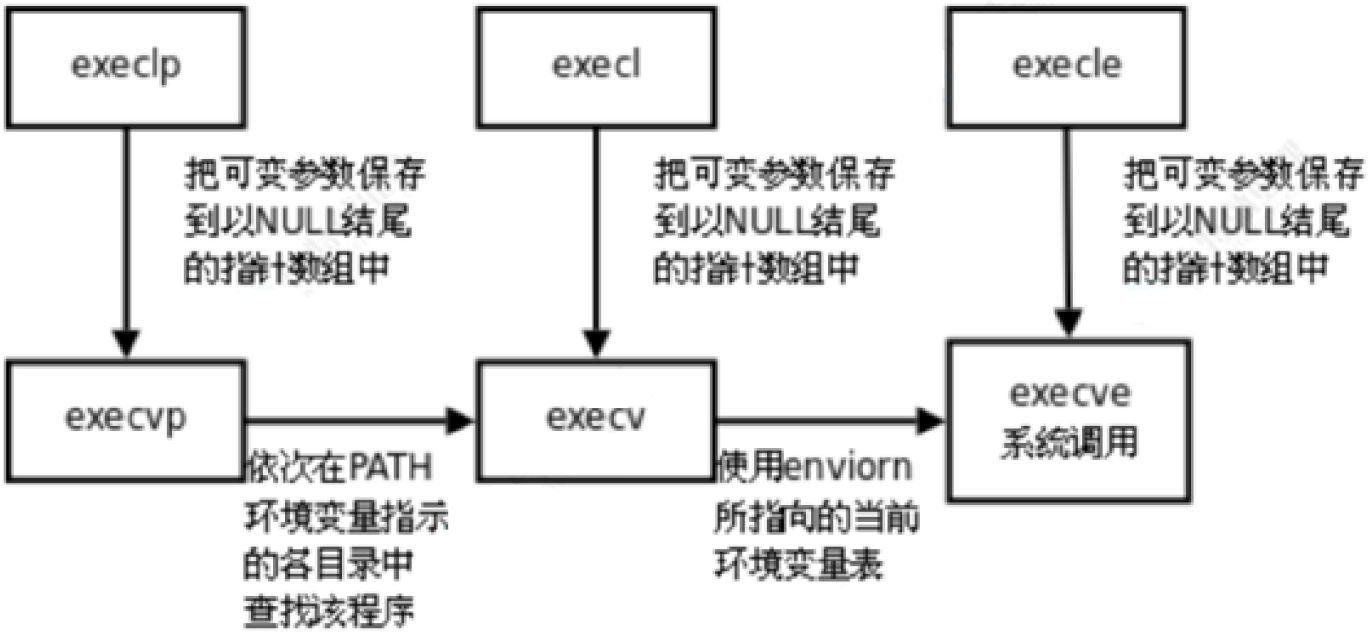
| 函数名 | 参数格式 | 是否带路径 | 是否使用当前环境变量 |
| --- | --- | --- | --- |
| execl | 列表 | 不是 | 是 |
| execlp | 列表 | 是 | 是 |
| execle | 列表 | 不是 | 不是，须自己组装环境变量 |
| execv | 数组 | 不是 | 是 |
| execvp | 数组 | 是 | 是 |
| execve | 数组 | 不是 | 不是，须自己组装环境变量 |

代码块

```
#include <unistd.h>
int main()
{
    char *const argv[] = {"ps", "-ef", NULL};
    char *const envp[] = {"PATH=/bin:/usr/bin", "TERM=console", NULL};
    execl("/bin/ps", "ps", "-ef", NULL);

    // 带p的，可以使用环境变量PATH，无需写全路径
    execlp("ps", "ps", "-ef", NULL);

    // 带e的，需要自己组装环境变量
    execle("ps", "ps", "-ef", NULL, envp);
    execv("/bin/ps", argv);

    // 带p的，可以使用环境变量PATH，无需写全路径
    execvp("ps", argv);

    // 带e的，需要自己组装环境变量
    execve("/bin/ps", argv, envp);
    exit(0);
}
```

事实上,只有 execve 是真正的系统调用,其它五个函数最终都调用 execve ,所以 execve 在 man手册 第2节,其它函数在 man 手册第3节。这些函数之间的关系如下图所示。

下图exec函数簇 一个完整的例子:

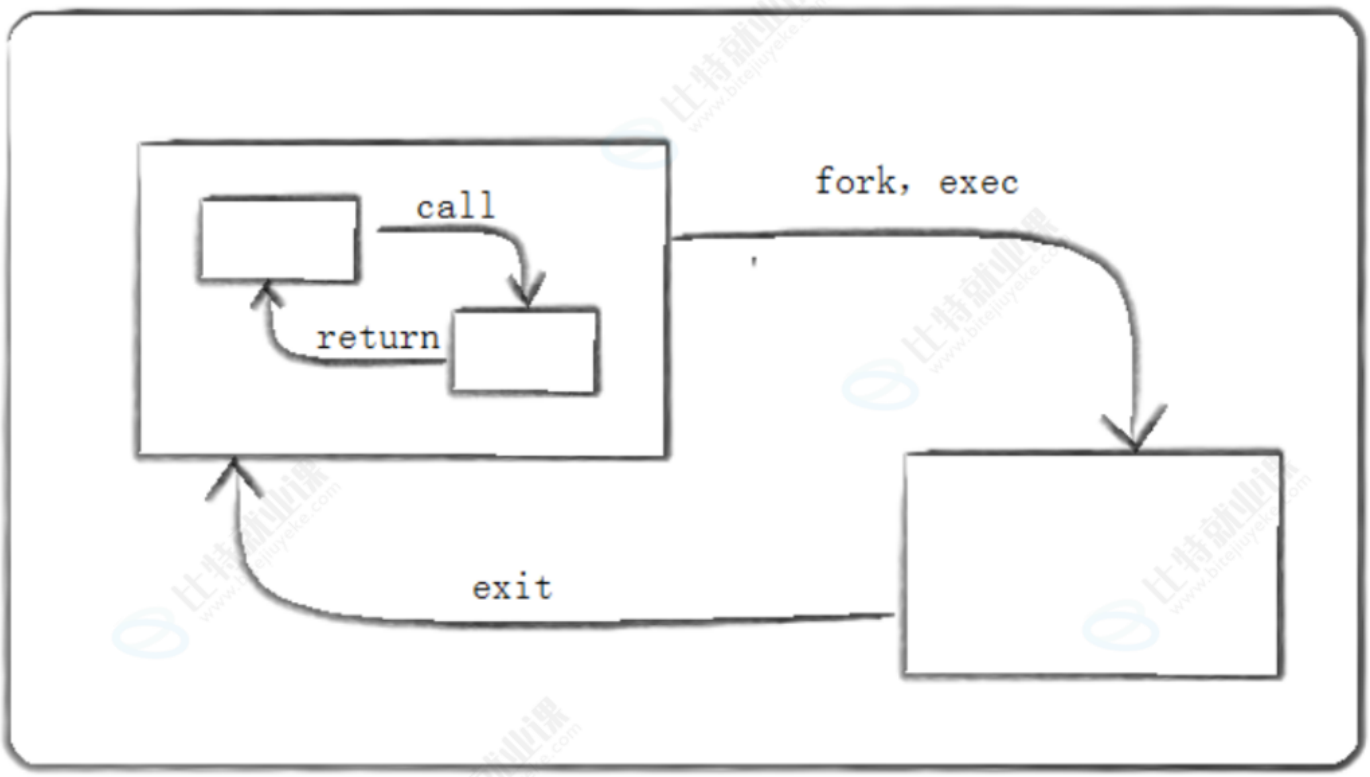



## 5-4 总结

在继续学习新知识前，我们来思考函数和进程之间的相似性 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。Linux鼓励将这种应用于程序之内的模式扩展到程序之间。如下图



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（&ret）来获取exit的返回值。