Linux下exec函数族详解

对于exec函数族来说，它的作用通俗来说就是使另一个可执行程序替换当前的进程，当我们在执行一个进程的过程中，通过exec函数使得另一个可执行程序A的数据段、代码段和堆栈段取代当前进程B的数据段、代码段和堆栈段，那么当前的进程就开始执行A中的内容，这一过程中不会创建新的进程，而且PID也没有改变。

一般exec函数族的用途有以下两种：

1. 当进程不需要再往下继续运行时，调用exec函数族中的函数让自己得以延续下去。

2. 如果当一个进程想执行另一个可执行程序时，可以使用fork函数先创建一个子进程，然后通过子进程来调用exec函数从而实现可执行程序的功能。

通过man命令来看一下exec函数族：

首先exec并不是一个函数名，而指一个函数族，之所以叫函数族就说明它有很多个不同的函数，但是这些函数的功能是一样的，只不过参数不同使用的方式也略不相同。那么在man命令下看到的exec函数原型是这样的：

#include <unistd.h>

extern char \*\*environ;

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

int execlp(const char \*file, const char \*arg, .../\* (char \*) NULL \*/);

int execle(const char \*path, const char \*arg, ...

/\*, (char \*) NULL, 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[]);

它们都是以exec为前缀，那么不同的之后后面的一些字符，l表示命令行参数列表、p表示PATH环境变量、v表示使用参数数组、e使用环境变量数组。其中execvpe和execle一般不常用，下面就以例子来看看具体的用法以及所展示出来的效果是怎么样的，便于更好的理解exec函数的作用，先来看一下下面的这个代码：

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

execl("/bin/ls", "ls", "-l", NULL);

perror("execl");

exit(1);

}

这个程序很简单，就是用当前的进程调用ls这个可执行程序，并添加了-l参数。由于execl成功调用后这个进程的代码段都被替换了，自然下面的代码就不会再执行了，所以也就没有返回值了，但是当调用失败后就会返回-1并设置errno值。那么在成功调用后实际上这个进程就变成了ls，然后执行ls -l的命令，因为我们用的是execl函数，所以第一个参数就需要用ls的所在目录，第二个参数其实没有实际意义，因为已经指定了ls的所在位置，所以第二个参数随便设置就可以但是不可以没有，第三个参数就是你所需要的功能，这里我用-l来举例，最后用NULL表示结束。那么运行结果如下：

total 16

-rw-r--r-- 1 charles charles 163 Feb 27 15:49 a.c

-rwxr-xr-x 1 charles charles 8384 Feb 27 15:49 test

如果是用execlp，那么第一个参数就可以不用加ls的路径了，直接是ls就可以了，因为系统会去PATH中查找。如果是execv的话，后面的参数就要是一个指针数组的形式，可以看下面的代码：

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main(void)

{

char \*argv[] = {"ls", "-l", NULL};

execv("/bin/ls", argv);

perror("execl");

exit(1);

}

一般的exec函数族的错误原因：

1. 找不到文件或者路径，此时errno为ENOENT。

2. 数组argv和envp(环境变量数组)没有以NULL结尾，此时errno为EFAULT。

3. 没有对应可执行文件的运行权限，此时errno为EACCES。

下面用exec函数来实现一个简单的程序b，我们先写一个这样的程序，getchar获取输入的小写字母，然后将其转换成大写输出出来，代码如下：

#include <stdio.h>

#include <ctype.h>

int main(void)

{

char s;

while((s = getchar())!=EOF){

putchar(toupper(s));

}

return 0;

}

然后我们再写一个程序a，它的作用是将一个文件打开，然后读取文件中的内容，然后调用exec函数打开这个转换大写字母的程序并将原文件中的内容输出。这里这个程序用到了dup2函数来进行重定向，直接看代码吧，不难理解。

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

int main(int argc, char \*argv[])

{

if(argc != 2){ // 接收到两个参数

printf("Open error!\n");

exit(1);

}

int fd = open(argv[1], O\_RDONLY); // 以只读的方式打开文本文件

if(fd < 0){

perror("open file");

exit(1);

}

dup2(fd, STDIN\_FILENO); // 输入重定向，使STDIN\_FILENO指向fd所指的文件

close(fd);

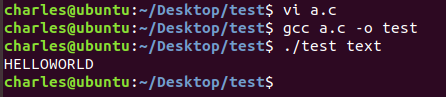
execv("./upper", "upper", NULL); // 调用upper可执行文件

perror("execl");

exit(1);

}

首先我们编译第一个程序生成可执行程序upper，然后再创建一个文本文件text，里面内容为helloworld，然后我们运行程序a，得到下面的运行结果：



Linux中exec族函数解析

1.exec函数说明

fork函数是用于创建一个子进程，该子进程几乎是父进程的副本，而有时我们希望子进程去执行另外的程序，exec函数族就提供了一个在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新程序的内容替换了。另外，这里的可执行文件既可以是二进制文件，也可以是Linux下任何可执行脚本文件。

2.在Linux中使用exec函数族主要有以下两种情况：

a. 当进程认为自己不能再为系统和用户做出任何贡献时，就可以调用任何exec 函数族让自己重生。

b. 如果一个进程想执行另一个程序，那么它就可以调用fork函数新建一个进程，然后调用任何一个exec函数使子进程重生。

3.exec函数族语法

实际上，在Linux中并没有exec函数，而是有6个以exec开头的函数族，下表列举了exec函数族的6个成员函数的语法。

所需头文件： #include <unistd.h>

函数说明： 执行文件

函数原型：

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

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

int execle(const char \*path, const char \*arg, ..., char \*const envp[])

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

int execlp(const char \*file, const char \*arg, ...)

int execvp(const char \*file, char \*const argv[])

函数返回值：成功 -> 函数不会返回，出错 -> 返回-1，失败原因记录在error中。

这6 个函数在函数名和使用语法的规则上都有细微的区别，下面就可执行文件查找方式、参数表传递方式及环境变量这几个方面进行比较说明。

查找方式：上表其中前4个函数的查找方式都是完整的文件目录路径，而最后2个函数（也就是以p结尾的两个函数）可以只给出文件名，系统就会自动从环境变量“$PATH”所指出的路径中进行查找。

参数传递方式：exec函数族的参数传递有两种方式，一种是逐个列举的方式，而另一种则是将所有参数整体构造成指针数组进行传递。在这里参数传递方式是以函数名的第5位字母来区分的，字母为“l”（list）的表示逐个列举的方式，字母为“v”（vertor）的表示将所有参数整体构造成指针数组传递，然后将该数组的首地址当做参数传给它，数组中的最后一个指针要求是NULL。读者可以观察execl、execle、execlp的语法与execv、execve、execvp的区别。

环境变量：exec函数族使用了系统默认的环境变量，也可以传入指定的环境变量。这里以“e”（environment）结尾的两个函数execle、execve就可以在envp[]中指定当前进程所使用的环境变量替换掉该进程继承的环境变量。

PATH环境变量说明：PATH环境变量包含了一张目录表，系统通过PATH环境变量定义的路径搜索执行码，PATH环境变量定义时目录之间需用用“:”分隔，以“.”号表示结束。PATH环境变量定义在用户的.profile或.bash\_profile中，下面是PATH环境变量定义的样例，此PATH变量指定在“/bin”、“/usr/bin”和当前目录三个目录进行搜索执行码。PATH=/bin:/usr/bin:. export $PATH

进程中的环境变量说明: 在Linux中，Shell进程是所有执行码的父进程。当一个执行码执行时，Shell进程会fork子进程然后调用exec函数去执行执行码。Shell进程堆栈中存放着该用户下的所有环境变量，使用execl、execv、execlp、execvp函数使执行码重生时，Shell进程会将所有环境变量复制给生成的新进程；而使用execle、execve时新进程不继承任何Shell进程的环境变量，而由envp[]数组自行设置环境变量。

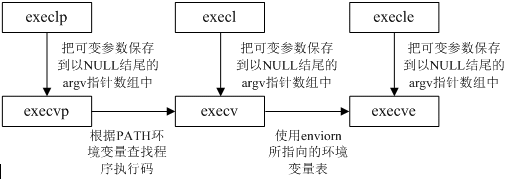
exec函数族关系:

前4位统一为：exec

第5位：l：参数传递为逐个列举方式：execl、execle、execlp；v：参数传递为构造指针数组方式：execv、execve、execvp。

第6位：e：可传递新进程环境变量：execle、execve；p：可执行文件查找方式为文件名：execlp、execvp。

事实上，这6个函数中真正的系统调用只有execve，其他5个都是库函数，它们最终都会调用execve这个系统调用，调用关系如下图所示：



4.exec调用举例如下：

char \*const ps\_argv[] ={"ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL};

char \*const ps\_envp[] ={"PATH=/bin:/usr/bin", "TERM=console", NULL};

execl("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);

execv("/bin/ps", ps\_argv);

execle("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL, ps\_envp) ;

execve("/bin/ps", ps\_argv, ps\_envp);

execlp("ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);

execvp("ps", ps\_argv);

请注意exec函数族形参展开时的前两个参数，第一个参数是带路径的执行码（execlp、execvp函数第一个参数是无路径的，系统会根据PATH自动查找然后合成带路径的执行码），第二个是不带路径的执行码，执行码可以是二进制执行码和Shell脚本。

5.exec函数族使用注意点：

在使用exec函数族时，一定要加上错误判断语句。因为exec很容易执行失败，其中最常见的原因有：

① 找不到文件或路径，此时errno被设置为ENOENT。

② 数组argv和envp忘记用NULL结束，此时errno被设置为EFAULT。

③ 没有对应可执行文件的运行权限，此时errno被设置为EACCES。

6.exec后新进程保持原进程以下特征

环境变量（使用了execle、execve函数则不继承环境变量）；

进程ID和父进程ID；

实际用户ID和实际组ID；

附加组ID；

进程组ID；

会话ID；

控制终端；

当前工作目录；

根目录；

文件权限屏蔽字；

文件锁；

进程信号屏蔽；

未决信号；

资源限制；

tms\_utime、tms\_stime、tms\_cutime以及tms\_ustime值。

对打开文件的处理与每个描述符的exec关闭标志值有关，进程中每个文件描述符有一个exec关闭标志（FD\_CLOEXEC），若此标志设置，则在执行exec时关闭该描述符，否则该描述符仍打开。除非特地用fcntl设置了该标志，否则系统的默认操作是在exec后仍保持这种描述符打开，利用这一点可以实现I/O重定向。

7.execlp函数举例:

#include <stdio.h>

#include <unistd.h>

int main()

{

if (fork() == 0)

{

if (execlp("env", "env", NULL) < 0)

{

perror("execlp error!");

return -1;

}

}

return 0;

}编译 gcc execlp.c –o execlp。

执行 ./execlp，执行结果如下：

HOME=/home/test

DB2DB=test

SHELL=/bin/bash

……

由执行结果看出，execlp函数使执行码重生时继承了Shell进程的所有环境变量，其他三个不以e结尾的函数同理。

8.execle函数举例

利用函数execle，将环境变量添加到新建的子进程中去。

execle.c源代码如下：

#include <unistd.h>

#include <stdio.h>

int main()

{

/\*命令参数列表，必须以 NULL 结尾\*/

char \*envp[]={"PATH=/tmp","USER=sun",NULL};

if(fork()==0){

/\*调用 execle 函数，注意这里也要指出 env 的完整路径\*/

if(execle("/usr/bin/env","env",NULL,envp)<0)

{

perror("execle error!");

return -1 ;

}

}

return 0 ;

}

编译:gcc execle.c –o execle。执行./execle，执行结果如下：

PATH=/tmp

USER=sun

可见，使用execle和execve可以自己向执行进程传递环境变量，但不会继承Shell进程的环境变量，而其他四个exec函数则继承Shell进程的所有环境变量。

**Linux 之exit() 进程退出函数**

进程的退出

linux下进程数量太多会导致系统崩溃，在使用完一个进程之后要及时终止它。

进程退出一般有三种方法：

1、在main函数中使用 return关键字 ，使用 return 后系统会调用 exit()函数来终止进程。

2、手动调用 exit() 来终止进程。

3、调用 \_exit() 来终止进程。

exit() 和 \_exit() 函数

头文件

#include<unistd.h>

#include<stdlib.h>

原型

void \_exit(int status);

void exit(int status);

\_exit()

os立刻把管理内存的结构体、虚拟内存释放掉。

exit()

不会马上干掉结构体，先看先看当前进程有没有文件缓存区，若有则会先处理缓存区中的数据，然后销毁结构体。

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main()

{

pid\_t res;

res = fork();

if(res == -1)

{

printf("fork err\r\n");

}

if(res == 0)

{

//不加回车换行符，让输出的字符串数据保留在io缓存区里面

printf("son");

//退出状态设置为0，表示正常退出。

\_exit(0);

}

else if(res > 0)

{

printf("parent");

exit(0);

}

}

编译后执行结果

jl@jl-virtual-machine:~/test$ ./a.out

parentjl@jl-virtual-machine:~/test$

jl@jl-virtual-machine:~/test$

注意第二行开头的parent，因为在父进程中用的是 exit() 函数，在进程退出前会先处理完缓存区中的数据，所以缓存区中滞留的parent在推出前被打印出来。

使用\r\n输出字符串则会被立即打印。

exit和\_exit函数都是用来终止进程的。当程序执行到exit或\_exit时，系统无条件的停止剩下所有操作，清除包括PCB在内的各种数据结构，并终止本进程的运行。但是，这两个函数是有区别的。

exit()函数的作用是：直接使用进程停止运行，清除其使用的内存空间，并清除其在内核中的各种数据结构；\_exit()函数则在这一基础上做了一些包装。在执行退出之前加了若干道工序。exit()函数与\_exit()函数最大区别就在于exit()函数在调用exit系统之前要检查文件的打开情况，把文件缓冲区的内容写回文件。

由于Linux的标准函数库中，有一种被称作“缓冲I/O”的操作，其特征就是对应每一个打开的文件，在内存中都有一片缓冲区。每次读文件时，会连续的读出若干条记录，这样在下次读文件时就可以直接从内存的缓冲区读取；同样，每次写文件的时候也仅仅是写入内存的缓冲区，等满足了一定的条件（如达到了一定数量或遇到特定字符等），再将缓冲区中的内容一次性写入文件。

这种技术大大增加了文件读写的速度，但也给编程代来了一点儿麻烦。比如有一些数据，认为已经写入了文件，实际上因为没有满足特定的条件，它们还只是保存在缓冲区内，这时用\_exit()函数直接将进程关闭，缓冲区的数据就会丢失。因此，要想保证数据的完整性，就一定要使用exit()函数。

exit的函数声明在stdlib.h头文件中。

\_exit的函数声明在unistd.h头文件当中。

下面的实例比较了这两个函数的区别。printf函数就是使用缓冲I/O的方式，该函数在遇到“\n”换行符时自动的从缓冲区中将记录读出。实例就是利用这个性质进行比较的。

exit.c源码

#include <stdlib.h>

#include <stdio.h>

int main(void)

{

printf("Using exit...\n");

printf("This is the content in buffer");

exit(0);

}

输出信息：

Using exit...

This is the content in buffer

#include <unistd.h>

#include <stdio.h>

int main(void)

{

printf("Using exit...\n");

printf("This is the content in buffer");

\_exit(0);

}

则只输出：

Using exit...

说明：在一个进程调用了exit之后，该进程并不会马上完全小时，而是留下一个称为僵尸进程（Zombie）的数据结构。僵尸进程是一种非常特殊的进程，它几乎已经放弃了所有的内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个位置，记载该进程的退出状态等信息供其它进程收集，除此之外，僵尸进程不再占有任何内存空间。

**详解Linux下的\_exit(),exit()和atexit()函数**

当内核使用一个一个exec函数执行c程序时，在调用main函数之前先调用一个特殊的启动例程，可执行程序需将此例程指定为程序的起始地址。启动例程从内核获取命令行参数和环境变量，然后为调用mian函数做好准备。

fork函数是用于创建一个子进程，该子进程几乎是父进程的副本，而有时我们希望子进程去执行另外的程序，exec函数族就提供了一个在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新程序的内容替换了。另外，这里的可执行文件既可以是二进制文件，也可以是Linux下任何可执行脚本文件。

那么在linux中调用exec函数有两种情况：

当进程认为自己不能再为系统和用户做出任何贡献时，就可以调用任何exec 函数族让自己重生。

如果一个进程想执行另一个程序，那么它就可以调用fork函数新建一个进程，然后调用任何一个exec函数使子进程重生。

进程终止

进程终止(包括线程)的方式有8种，前5种为正常终止，后3种为异常终止：

1 从main函数返回

2 调⽤用exit函数；

3 调⽤用\_exit或\_Exit；

4 最后⼀一个线程从启动例程返回；

5 最后⼀一个线程调⽤用pthread\_exit；

6 调⽤用abort函数；

7 接到⼀一个信号并终⽌止；

8 最后⼀一个线程对取消请求做出响应。

一般我们都是使用exit函数正常终止进程

#include <stdlib.h>

void exit( int status );

void \_Exit( int status );

#include <unistd.h>

void \_exit( int status );

1

2

3

4

5

\_exit和\_Exit立即进入内核，而exit要先做一些清理工作（调用执行个终止处理程序，关闭所有标准I/O流），再进入内核。三个函数所带的整形参数称为终止状态或退出状态，如果

（a）调用这些函数不带参数

（b）main函数中的return 语句无返回值

（c）main函数没有声明返回类型为整形

则进程的终止状态是未定义的。main函数返回一个整形值与用该值调用exit是等价的。

exit()和\_exit()

exit函数运行时首先会执行由atexit()函数登记的函数，然后会做一些自身的清理工作，同时刷新所有输出流、关闭所有打开的流并且关闭通过标准I/O函数tmpfile()创建的临时文件。

1.\_exit函数

#include <unistd.h>

void \_exit(int status);

status定义了进程终止的状态，父进程可以使用wait()来获取状态。

2.exit函数

#include <unistd.h>

void exit(int status);

实际上exit函数最终还是要调用\_exit（）

执行用户通过调用atexit函数或on\_exit函数定义的清理函数----->关闭打开的流，所有缓冲区均被写入，临时文件被删除————>调用\_exit

如果在退出之前，想增加一些用户的额外功能，可以通过atexit函数来实现功能调用。否则，退出仅仅是将缓冲区中的内容清空，并回收其他系统资源。

##缓冲区（协调不同设备之间的速度差）

无缓冲区（\_INOBF）:没有缓冲区，每次调用glibc库中的函数时，会立即调用write/read系统调用（对设备的读写不经过缓冲，立即执行数据交换）

行缓冲区（\_IIOBF）：对于输出流，在收到回车换行符（\n）之前，一律进行数据冲刷，除非缓冲区满。即在向设备输出数据时，在没有遇到回车换行符号(\n)，数据只是写入缓冲区。在遇到（\n）时，才将数据写入外设。对于输入流，每次读入一行数据。

全缓冲（\_IFBF）：在系统缓冲满之前，都可以不停地向缓冲区进行IO操作，而不会立即调用write/read系统调用向外设进行数据读写。系统缓冲区的大小默认是8192字节。

exit是正常退出，在退出可以调用用户自定义函数进行执行用户任务，然后会清空缓冲区中的内容，最后释放所有资源

\_exit是强制退出，不在退出时调用其他函数，也不会将缓冲区中的内容不清空。直接强制退出。

exit(int n)、\_exit(int n)、return n

n为返回给调用者的退出状态码。调用者通过wait或waitpid函数来接收退出码。

n值最好不要大于128

0代表正常成功退出

1-125代表程序执行出现异常，具体含义可以自己定义

126找到了可执行文件，但无法运行

127找不到文件

大于等于128表示进程因为接收到信号而结束

atexit函数

atexit函数是一个特殊的函数，它是在正常程序退出时调用的函数，我们把他叫为登记函数

（函数原型：int atexit (void (\*)(void))）

一个进程可以登记若32个函数，这些函数由exit自动调用，这些函数被称为终止处理函数，atexit函数可以登记这些函数。exit调用终止处理函数的顺序和atexit登记的顺序相反，如果一个函数被多次登记，也会被多次调用。

exit()，\_exit()以及atexit()函数实例分析

源码如下

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<string.h>

#include<wait.h>

void fun1(){

fprintf(stderr,"myFun1 says bye!\n");

}

void fun2(){

fprintf(stderr,"myFun2 says bye!\n");

}

int main(int argc,char \*argv[]){

atexit(fun1);

atexit(fun2);

fprintf(stderr,"Oops~~~fogot a newline!\n");

sleep(2);

pid\_t result;

result=fork();

if(result==-1){

perror("创建子进程失败！");

exit(0);

}else if(result==0)

{

printf("测试终止进程的\_exit函数！\n");

printf("子进程： 这一行我们用缓存");

\_exit(0);

}

else{

printf("测试终止进程的exit函数！\n");

printf("父进程: 这一行我们用缓存");

wait(5);

exit(0);

}

return 0;

运行结果

————————————————

**Linux中fork()函数的详解**

一：fork()的介绍

fork（）函数通过系统调用创建一个与原来进程几乎完全相同的进程。子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）一份副本，包括代码和数据段、堆、共享库以及用户栈。子进程还获得与父进程任何打开文件描述符相同的副本，当父进程调用fork时，子进程可以读写父进程中打开的任何文件。

父进程和新创建的子进程之间最大的区别在于它们有不同的PID。

fork函数只被调用一次，却会返回两次。

以下代码的从网上找的

1）在父进程中，fork返回新创建子进程的进程ID；

2）在子进程中，fork返回0；

3）如果出现错误，fork返回一个负值

在fork函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork函数返回0，在父进程中，fork返回新创建子进程的进程ID。我们可以通过fork返回的值来判断当前进程是子进程还是父进程。

每个进程都有一个独特（互不相同）的进程标识符（process ID），可以通过getpid（）函数获得，还有一个记录父进程pid的变量，可以通过getppid（）函数获得变量的值。

通过result=1来看，这两个进程的变量都是独立的，存在不同的地址中，不是共用的。

二.fork()的二重循环的解释

在父进程中，指令执行到for循环中，i=0，接着执行fork，fork执行完后，系统中出现两个进程，分别是p3906和p3907。可以看到父进程p3906的父进程是p3886，子进程p3907的父进程正好是p3906。我们用一个链表来表示这个关系： p3886->p3906->p3907 第一次fork后，p3906（父进程）的变量为i=0，fpid=3907（fork函数在父进程中返向子进程id）

第一次fork后，p3906（父进程）的变量为i=0，fpid=3907（fork函数在父进程中返向子进程id）

p3907（子进程）的变量为i=0，fpid=0（fork函数在子进程中返回0）

父进程p3906先执行，当进入下一个循环时，i=1，接着执行fork，系统中又新增一个进程p3908，对于此时的父进程，p3886->p3906（当前进程）->p3908（被创建的子进程）。

对于子进程p3907，执行完第一次循环后，i=1，接着执行fork，系统中新增一个进程p3909，对于此进程，p1566->p3907（当前进程）->p3909（被创建的子进程）。从输出可以看到p3907原来是p1566的子进程，现在变成p3909的父进程。父子是相对的。只要当前进程执行了fork，该进程就变成了父进程了，就打印出了parent。

创建了两个进程p3908，p3909，这两个进程执行完printf函数后就结束了，因为这两个进程无法进入第三次循环，无法fork，该执行return 0;了，其他进程也是如此。 细心的读者可能注意到p3908，p3909的父进程难道不该是p3906和p3907吗，怎么会是1呢？这里得讲到进程的创建和死亡的过程，在p3906和p3907执行完第二个循环后，main函数就该退出了，也即进程该死亡了，因为它已经做完所有事情了。p3906和p3907死亡后，p3908，p3909就没有父进程了，这在操作系统是不被允许的，所以p3908，p3909的父进程就被置为p1了，p1是永远不会死亡的。

每次程序运行的结果会稍有不同，这是因为进程执行没有固定的先后顺序，哪个进程先执行要看系统的进程调度策略。

三.fork的三重循环

for i=0 1 2

father father father

son

son father

son

son father father

son

son father

son 其中每一行分别代表一个进程的运行打印结果。

————————————————

**Linux系统编程—— fork() 函数详解**

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

功能： 用于从一个已存在的进程中创建一个新进程，新进程称为子进程，原进程称为父进程。

参数：无

返回值：

成功：子进程中返回 0，父进程中返回子进程 ID。pid\_t，为无符号整型。

失败：返回 -1。

失败的两个主要原因是：

1）当前的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN。

2）系统内存不足，这时 errno 的值被设置为 ENOMEM。

测试示例如下：

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main(int argc, char \*argv[])

{

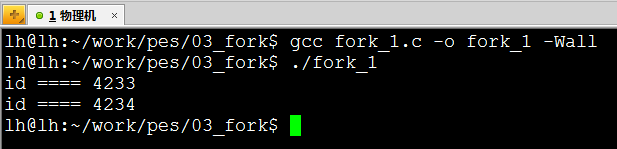
fork();

printf("id ==== %d\n", getpid()); // 获取进程号

return 0;

}

运行结果：

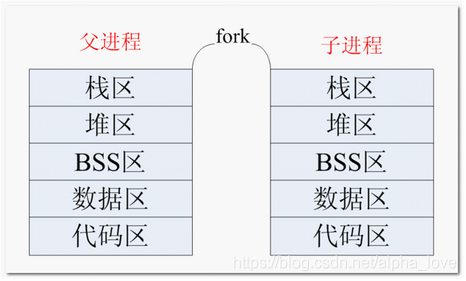


从运行结果，我们可以看出，fork() 之后的打印函数打印了两次，而且打印了两个进程号，这说明，fork() 之后确实创建了一个新的进程，新进程为子进程，原来的进程为父进程。

那子进程长什么样的呢？

使用 fork() 函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间：包括进程上下文（进程执行活动全过程的静态描述）、进程堆栈、打开的文件描述符、信号控制设定、进程优先级、进程组号等。子进程所独有的只有它的进程号，计时器等（只有小量信息）。因此，使用 fork() 函数的代价是很大的。

简单来说， 一个进程调用 fork() 函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的新进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。



实际上，更准确来说，Linux 的 fork() 使用是通过写时拷贝 (Copy- On-Write，COW技术) 实现。写时拷贝是一种可以推迟甚至避免拷贝数据的技术。内核此时并不复制整个进程的地址空间，而是让父子进程共享同一个地址空间。只用在需要写入的时候才会复制地址空间，从而使各个进行拥有各自的地址空间。也就是说，资源的复制是在需要写入的时候才会进行，在此之前，只有以只读方式共享。

子进程是父进程的一个复制品，可以简单认为父子进程的代码一样的。那大家想过没有，这样的话，父进程做了什么事情，子进程也做什么事情（如上面的例子），是不是不能实现满足我们实现多任务的要求呀，那我们是不是要想个办法区别父子进程呀，这就通过 fork() 的返回值。

fork() 函数被调用一次，但返回两次。两次返回的区别是：子进程的返回值是 0，而父进程的返回值则是新子进程的进程 ID。

测试代码：

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main(int argc, char \*argv[])

{

pid\_t pid;

pid = fork();

if( pid < 0 ){ // 没有创建成功

perror("fork");

}

if(0 == pid){ // 子进程

while(1){

printf("I am son\n");

sleep(1);

}

}else if(pid > 0){ // 父进程

while(1){

printf("I am father\n");

sleep(1);

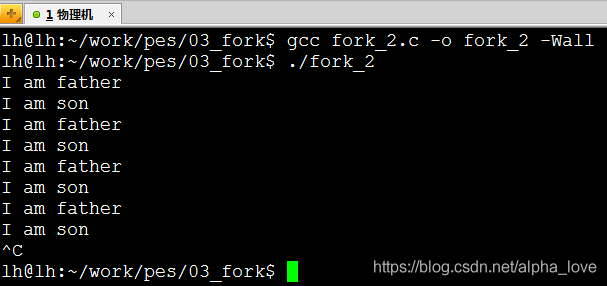
}

}

return 0;

}

运行结果：



通过运行结果，可以看到，父子进程各做一件事（各自打印一句话）。这里，我们只是看到只有一份代码，实际上，fork() 以后，有两个地址空间在独立运行着，有点类似于有两个独立的程序（父子进程）在运行着。需要注意的是，在子进程的地址空间里，子进程是从 fork() 这个函数后才开始执行代码。

一般来说，在 fork() 之后是父进程先执行还是子进程先执行是不确定的。这取决于内核所使用的调度算法

下面的例子，为验证父子进程各自的地址空间是独立的

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int a = 10; // 全局变量

int main(int argc, char \*argv[])

{

int b = 20; //局部变量

pid\_t pid;

pid = fork();

if( pid < 0 ){ // 没有创建成功

perror("fork");

}

if(0 == pid){ // 子进程

a = 111;

b = 222; // 子进程修改其值

printf("son: a = %d, b = %d\n", a, b);

}else if(pid > 0){ // 父进程

sleep(1); // 保证子进程先运行,但不保证1秒足够

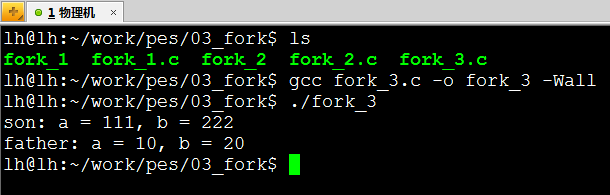
printf("father: a = %d, b = %d\n", a, b);

}

return 0;

}

运行结果：



通过得知，在子进程修改变量 a，b 的值，并不影响到父进程 a，b 的值。

在以后的编程中，fork() 之后最好是加上判断，区别哪个是子进程的空间，哪个为父进程的空间，否则，fork() 以后的代码为父子进程各有一份，对于用户而言，没有太大意义，如例子1。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdarg.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <time.h>

int tprintf(const char \*fmt,...);

int main(void)

{

printf("I'm your father,PID is %d.\n",getpid());

pid\_t pid = fork();

if(pid == 0){

printf("I'm a first son.PID is %d.\n",getpid());

exit(1);//若此处没有exit（1）， 进程也会执行 pid\_t pid2 = fork()语句,会出现孙进程

printf("You should never see this.\n");

}

pid\_t pid2 = fork();

if(pid2 == 0){

printf("I'm a second son.PID is %d.\n",getpid());

exit(1);

printf("You should never see this.\n");

}

pid\_t pid3 = fork();

if(pid3 ==0){

printf("I'm a third son.PID is %d.\n",getpid());

exit(1);

printf("You should never see this.\n");

}

else if(pid != -1){

tprintf("Parent forked child process--%d.\n",pid);

tprintf("Parent is waiting for child to exit.\n");

waitpid(pid,NULL,0);

waitpid(pid2,NULL,0);

waitpid(pid3,NULL,0);

tprintf("Child Process had exited.\n");

tprintf("Parent had exited.\n");

}

else tprintf("Everything was done without error.\n");

return 0;

}

/\*

\* 设置输出格式

\*/

int tprintf(const char\* fmt,...)

{

va\_list args;

struct tm \*tstruct;

time\_t tsec;

tsec = time(NULL);

tstruct = localtime(&tsec);

printf("%02d:%02d:%02d:%5d|",tstruct->tm\_hour,tstruct->tm\_min,tstruct->tm\_sec,getpid());

va\_start(args,fmt);

return vprintf(fmt,args);

}

参考资料：http://blog.csdn.net/tennysonsky/article/details/45165811

**linux中fork（）函数详解**

**一、fork入门知识**

     一个进程，包括代码、数据和分配给进程的资源。fork（）函数通过系统调用创建一个与原来进程几乎完全相同的进程，也就是两个进程可以做完全相同的事，但如果初始参数或者传入的变量不同，两个进程也可以做不同的事。  
    一个进程调用fork（）函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的新进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。

     我们来看一个例子：

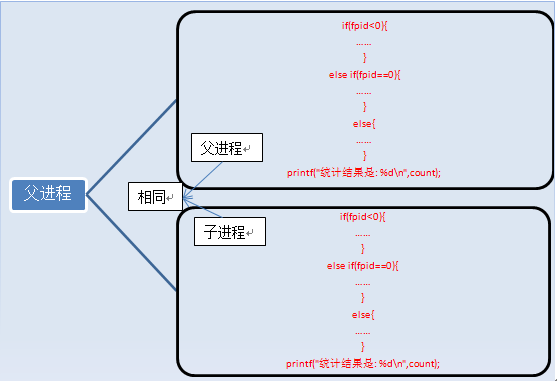
[cpp]  [view plain](https://blog.csdn.net/jason314/article/details/5640969)  [copy](https://blog.csdn.net/jason314/article/details/5640969)

1. /\*
2. \*  fork\_test.c
3. \*  version 1
4. \*  Created on: 2010-5-29
5. \*      Author: wangth
6. \*/
7. #include <unistd.h>
8. #include <stdio.h>
9. **int** main ()
10. {
11. pid\_t fpid; //fpid表示fork函数返回的值
12. **int** count=0;
13. fpid=fork();
14. **if** (fpid < 0)
15. printf("error in fork!");
16. **else** **if** (fpid == 0) {
17. printf("i am the child process, my process id is %d/n",getpid());
18. printf("我是爹的儿子/n");//对某些人来说中文看着更直白。
19. count++;
20. }
21. **else** {
22. printf("i am the parent process, my process id is %d/n",getpid());
23. printf("我是孩子他爹/n");
24. count++;
25. }
26. printf("统计结果是: %d/n",count);
27. **return** 0;
28. }

     运行结果是：  
  i am the child process, my process id is 5574  
    我是爹的儿子  
    统计结果是: 1  
    i am the parent process, my process id is 5573  
    我是孩子他爹  
    统计结果是: 1  
    在语句fpid=fork()之前，只有一个进程在执行这段代码，但在这条语句之后，就变成两个进程在执行了，这两个进程的几乎完全相同，将要执行的下一条语句都是if(fpid<0)……  
    为什么两个进程的fpid不同呢，这与fork函数的特性有关。fork调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：  
    1）在父进程中，fork返回新创建子进程的进程ID；  
    2）在子进程中，fork返回0；  
    3）如果出现错误，fork返回一个负值；

    在fork函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork函数返回0，在父进程中，fork返回新创建子进程的进程ID。我们可以通过fork返回的值来判断当前进程是子进程还是父进程。

    引用一位网友的话来解释fpid的值为什么在父子进程中不同。“其实就相当于链表，进程形成了链表，父进程的fpid(p 意味point)指向子进程的进程id, 因为子进程没有子进程，所以其fpid为0.  
    fork出错可能有两种原因：  
    1）当前的进程数已经达到了系统规定的上限，这时errno的值被设置为EAGAIN。  
    2）系统内存不足，这时errno的值被设置为ENOMEM。  
    创建新进程成功后，系统中出现两个基本完全相同的进程，这两个进程执行没有固定的先后顺序，哪个进程先执行要看系统的进程调度策略。  
    每个进程都有一个独特（互不相同）的进程标识符（process ID），可以通过getpid（）函数获得，还有一个记录父进程pid的变量，可以通过getppid（）函数获得变量的值。  
    fork执行完毕后，出现两个进程，



    有人说两个进程的内容完全一样啊，怎么打印的结果不一样啊，那是因为判断条件的原因，上面列举的只是进程的代码和指令，还有变量啊。  
    执行完fork后，进程1的变量为count=0，fpid！=0（父进程）。进程2的变量为count=0，fpid=0（子进程），这两个进程的变量都是独立的，存在不同的地址中，不是共用的，这点要注意。可以说，我们就是通过fpid来识别和操作父子进程的。  
    还有人可能疑惑为什么不是从#include处开始复制代码的，这是因为fork是把进程当前的情况拷贝一份，执行fork时，进程已经执行完了int count=0;fork只拷贝下一个要执行的代码到新的进程。

**二、fork进阶知识**

    先看一份代码：

[cpp]  [view plain](https://blog.csdn.net/jason314/article/details/5640969)  [copy](https://blog.csdn.net/jason314/article/details/5640969)

1. /\*
2. \*  fork\_test.c
3. \*  version 2
4. \*  Created on: 2010-5-29
5. \*      Author: wangth
6. \*/
7. #include <unistd.h>
8. #include <stdio.h>
9. **int** main(**void**)
10. {
11. **int** i=0;
12. printf("i son/pa ppid pid  fpid/n");
13. //ppid指当前进程的父进程pid
14. //pid指当前进程的pid,
15. //fpid指fork返回给当前进程的值
16. **for**(i=0;i<2;i++){
17. pid\_t fpid=fork();
18. **if**(fpid==0)
19. printf("%d child  %4d %4d %4d/n",i,getppid(),getpid(),fpid);
20. **else**
21. printf("%d parent %4d %4d %4d/n",i,getppid(),getpid(),fpid);
22. }
23. **return** 0;
24. }

    运行结果是：  
    i son/pa ppid pid  fpid  
    0 parent 2043 3224 3225  
    0 child  3224 3225    0  
    1 parent 2043 3224 3226  
    1 parent 3224 3225 3227  
    1 child     1 3227    0  
    1 child     1 3226    0   
    这份代码比较有意思，我们来认真分析一下：  
    第一步：在父进程中，指令执行到for循环中，i=0，接着执行fork，fork执行完后，系统中出现两个进程，分别是p3224和p3225（后面我都用pxxxx表示进程id为xxxx的进程）。可以看到父进程p3224的父进程是p2043，子进程p3225的父进程正好是p3224。我们用一个链表来表示这个关系：  
    p2043->p3224->p3225   
    第一次fork后，p3224（父进程）的变量为i=0，fpid=3225（fork函数在父进程中返向子进程id），代码内容为：

[c-sharp]  [view plain](https://blog.csdn.net/jason314/article/details/5640969)  [copy](https://blog.csdn.net/jason314/article/details/5640969)

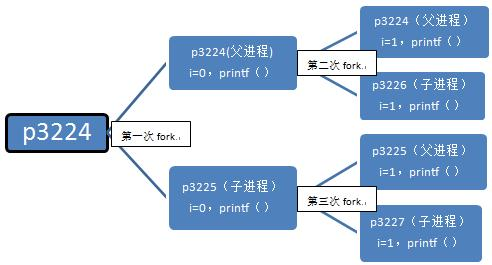
1. **for**(i=0;i<2;i++){
2. pid\_t fpid=fork();//执行完毕，i=0，fpid=3225
3. **if**(fpid==0)
4. printf("%d child  %4d %4d %4d/n",i,getppid(),getpid(),fpid);
5. **else**
6. printf("%d parent %4d %4d %4d/n",i,getppid(),getpid(),fpid);
7. }
8. **return** 0;

    p3225（子进程）的变量为i=0，fpid=0（fork函数在子进程中返回0），代码内容为：

[c-sharp]  [view plain](https://blog.csdn.net/jason314/article/details/5640969)  [copy](https://blog.csdn.net/jason314/article/details/5640969)

1. **for**(i=0;i<2;i++){
2. pid\_t fpid=fork();//执行完毕，i=0，fpid=0
3. **if**(fpid==0)
4. printf("%d child  %4d %4d %4d/n",i,getppid(),getpid(),fpid);
5. **else**
6. printf("%d parent %4d %4d %4d/n",i,getppid(),getpid(),fpid);
7. }
8. **return** 0;

    所以打印出结果：  
    0 parent 2043 3224 3225  
    0 child  3224 3225    0  
    第二步：假设父进程p3224先执行，当进入下一个循环时，i=1，接着执行fork，系统中又新增一个进程p3226，对于此时的父进程，p2043->p3224（当前进程）->p3226（被创建的子进程）。  
    对于子进程p3225，执行完第一次循环后，i=1，接着执行fork，系统中新增一个进程p3227，对于此进程，p3224->p3225（当前进程）->p3227（被创建的子进程）。从输出可以看到p3225原来是p3224的子进程，现在变成p3227的父进程。父子是相对的，这个大家应该容易理解。只要当前进程执行了fork，该进程就变成了父进程了，就打印出了parent。  
    所以打印出结果是：  
    1 parent 2043 3224 3226  
    1 parent 3224 3225 3227   
    第三步：第二步创建了两个进程p3226，p3227，这两个进程执行完printf函数后就结束了，因为这两个进程无法进入第三次循环，无法fork，该执行return 0;了，其他进程也是如此。  
    以下是p3226，p3227打印出的结果：  
    1 child     1 3227    0  
    1 child     1 3226    0   
    细心的读者可能注意到p3226，p3227的父进程难道不该是p3224和p3225吗，怎么会是1呢？这里得讲到进程的创建和死亡的过程，在p3224和p3225执行完第二个循环后，main函数就该退出了，也即进程该死亡了，因为它已经做完所有事情了。p3224和p3225死亡后，p3226，p3227就没有父进程了，这在操作系统是不被允许的，所以p3226，p3227的父进程就被置为p1了，p1是永远不会死亡的，至于为什么，这里先不介绍，留到“三、fork高阶知识”讲。  
    总结一下，这个程序执行的流程如下：



     这个程序最终产生了3个子进程，执行过6次printf（）函数。  
    我们再来看一份代码：

[cpp]  [view plain](https://blog.csdn.net/jason314/article/details/5640969)  [copy](https://blog.csdn.net/jason314/article/details/5640969)

1. /\*
2. \*  fork\_test.c
3. \*  version 3
4. \*  Created on: 2010-5-29
5. \*      Author: wangth
6. \*/
7. #include <unistd.h>
8. #include <stdio.h>
9. **int** main(**void**)
10. {
11. **int** i=0;
12. **for**(i=0;i<3;i++){
13. pid\_t fpid=fork();
14. **if**(fpid==0)
15. printf("son/n");
16. **else**
17. printf("father/n");
18. }
19. **return** 0;
21. }

     它的执行结果是：  
    father  
    son  
    father  
    father  
    father  
    father  
    son  
    son  
    father  
    son  
    son  
    son  
    father  
    son   
    这里就不做详细解释了，只做一个大概的分析。  
    for        i=0         1           2  
              father     father     father  
                                        son  
                            son       father  
                                        son  
               son       father     father  
                                        son  
                            son       father  
                                        son  
    其中每一行分别代表一个进程的运行打印结果。  
    总结一下规律，对于这种N次循环的情况，执行printf函数的次数为2\*（1+2+4+……+2N-1）次，创建的子进程数为1+2+4+……+2N-1个。(感谢gao\_jiawei网友指出的错误，原本我的结论是“执行printf函数的次数为2\*（1+2+4+……+2N）次，创建的子进程数为1+2+4+……+2N”，这是错的)  
    网上有人说N次循环产生2\*（1+2+4+……+2N）个进程，这个说法是不对的，希望大家需要注意。

    数学推理见<http://202.117.3.13/wordpress/?p=81>（该博文的最后）。  
    同时，大家如果想测一下一个程序中到底创建了几个子进程，最好的方法就是调用printf函数打印该进程的pid，也即调用printf("%d/n",getpid());或者通过printf("+/n");来判断产生了几个进程。有人想通过调用printf("+");来统计创建了几个进程，这是不妥当的。具体原因我来分析。  
    老规矩，大家看一下下面的代码：

[cpp]  [view plain](https://blog.csdn.net/jason314/article/details/5640969)  [copy](https://blog.csdn.net/jason314/article/details/5640969)

1. /\*
2. \*  fork\_test.c
3. \*  version 4
4. \*  Created on: 2010-5-29
5. \*      Author: wangth
6. \*/
7. #include <unistd.h>
8. #include <stdio.h>
9. **int** main() {
10. pid\_t fpid;//fpid表示fork函数返回的值
11. //printf("fork!");
12. printf("fork!/n");
13. fpid = fork();
14. **if** (fpid < 0)
15. printf("error in fork!");
16. **else** **if** (fpid == 0)
17. printf("I am the child process, my process id is %d/n", getpid());
18. **else**
19. printf("I am the parent process, my process id is %d/n", getpid());
20. **return** 0;
21. }

*执行结果如下：  
    fork!  
    I am the parent process, my process id is 3361  
    I am the child process, my process id is 3362   
    如果把语句printf("fork!/n");注释掉，执行printf("fork!");  
    则新的程序的执行结果是：  
    fork!I am the parent process, my process id is 3298  
    fork!I am the child process, my process id is 3299   
    程序的唯一的区别就在于一个/n回车符号，为什么结果会相差这么大呢？  
    这就跟printf的缓冲机制有关了，printf某些内容时，操作系统仅仅是把该内容放到了stdout的缓冲队列里了,并没有实际的写到屏幕上。但是,只要看到有/n 则会立即刷新stdout,因此就马上能够打印了。  
    运行了printf("fork!")后,“fork!”仅仅被放到了缓冲里,程序运行到fork时缓冲里面的“fork!”  被子进程复制过去了。因此在子进程度stdout缓冲里面就也有了fork! 。所以,你最终看到的会是fork!  被printf了2次！！！！  
    而运行printf("fork! /n")后,“fork!”被立即打印到了屏幕上,之后fork到的子进程里的stdout缓冲里不会有fork! 内容。因此你看到的结果会是fork! 被printf了1次！！！！  
    所以说printf("+");不能正确地反应进程的数量。  
    大家看了这么多可能有点疲倦吧，不过我还得贴最后一份代码来进一步分析fork函数。*

[cpp]  [view plain](https://blog.csdn.net/jason314/article/details/5640969)  [copy](https://blog.csdn.net/jason314/article/details/5640969)

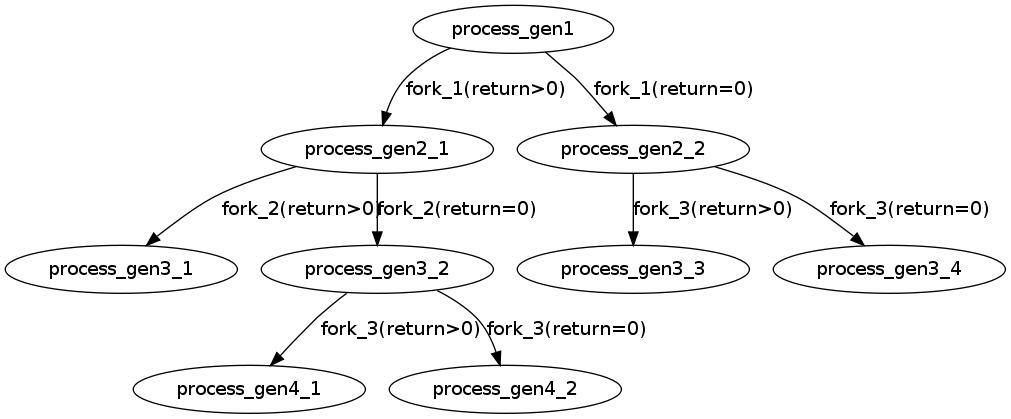
1. #include <stdio.h>
2. #include <unistd.h>
3. **int** main(**int** argc, **char**\* argv[])
4. {
5. fork();
6. fork() && fork() || fork();
7. fork();
8. **return** 0;
9. }

*问题是不算main这个进程自身，程序到底创建了多少个进程。  
    为了解答这个问题，我们先做一下弊，先用程序验证一下，到此有多少个进程。*

[c-sharp]  [view plain](https://blog.csdn.net/jason314/article/details/5640969)  [copy](https://blog.csdn.net/jason314/article/details/5640969)

1. #include <stdio.h>
2. **int** main(**int** argc, **char**\* argv[])
3. {
4. fork();
5. fork() && fork() || fork();
6. fork();
7. printf("+/n");
8. }

*答案是总共20个进程，除去main进程，还有19个进程。  
    我们再来仔细分析一下，为什么是还有19个进程。  
    第一个fork和最后一个fork肯定是会执行的。  
    主要在中间3个fork上，可以画一个图进行描述。  
    这里就需要注意&&和||运算符。  
    A&&B，如果A=0，就没有必要继续执行&&B了；A非0，就需要继续执行&&B。  
    A||B，如果A非0，就没有必要继续执行||B了，A=0，就需要继续执行||B。  
    fork()对于父进程和子进程的返回值是不同的，按照上面的A&&B和A||B的分支进行画图，可以得出5个分支。*



     加上前面的fork和最后的fork，总共4\*5=20个进程，除去main主进程，就是19个进程了。

***三、fork高阶知识***

*这一块我主要就fork函数讲一下操作系统进程的创建、死亡和调度等。因为时间和精力限制，我先写到这里，下次找个时间我争取把剩下的内容补齐。*

*原文地址：*[*http://blog.csdn.net/jason314/article/details/5640969*](http://blog.csdn.net/jason314/article/details/5640969)

**linux之wait函数**

wait函数：成功:清理掉的子进程 ID;失败:-1 (没有子进程)

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*wstatus);

wstatus是传出参数，

a.wait函数有3个功能：

(1)阻塞等待子进程退出

(2)回收子进程残留资源

(3)获取子进程结束状态(退出原因)。

b.可使用 wait 函数传出参数 status 来保存进程的退出状态。借助宏函数来进一步判断进程终止的具体原因。常用的宏函数可分为如下两组:

(1)WIFEXITED(status)为非0 -----------进程正常结束

如果宏函数WIFEXITED(status)的返回值为真（非0），则进一步调用宏函数WEXITSTATUS(status)，此函数返回的是子进程退出的值，比如exit(10),返回值就是10

(2) WEXITSTATUS(status) 如上宏为真,使用此宏---------获取进程退出状态 (exit 的参数)

(3)WIFSIGNALED(status)为非0----------进程异常终止

如果宏函数WIFSIGNALED(status)的返回值为真（非0），则进一步调用宏函数WTERMSIG(status)，此函数返回的是使子进程终止的那个信号的编号

(3) WTERMSIG(status) 如上宏为真,使用此宏----------取得使进程终止的那个信号的编号

wait函数的使用：wait.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main()

{

pid\_t pid,wpid;

int status;

pid = fork();

if(pid==0){

//让子进程异常退出

execl("abnor","abnor",NULL);

printf("---child, my parent=%d, going to sleep 3s\n",getppid());

sleep(60);

printf("--------------child die--------------\n");

//exit(76);

return 100;

}

else if(pid>0){

wpid = wait(&status);//子进程的退出状态保存在status中

if(wpid==-1){

perror("wait error:");

exit(1);

}

//子进程正常退出

if(WIFEXITED(status)){//如果宏函数WIFEXITED(status)的返回值为真，则继续调用宏函数WEXITSTATUS(status)，此函数返回的是子进程退出的值

printf("child exit with %d\n",WEXITSTATUS(status));

}

//子进程异常退出

//测试子进程异常退出(不是子进程自己异常)：运行程序之后，新开终端 ps aux ，杀掉子进程，此时会打印 child killed by ...

if(WIFSIGNALED(status)){

printf("chile killed by %d\n",WTERMSIG(status));

}

while(1){

printf("I am paraent, pid = %d, myson = %d\n",getpid(),pid);

sleep(1);

}

}

else{

perror("fork");

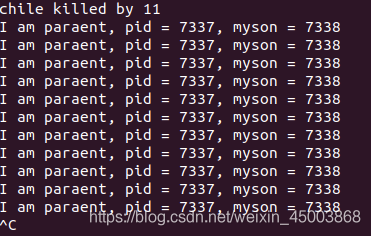
return 1;

}

return 0;

}

结果：



**Linux：进程等待wait函数**

进程等待的作用：

父进程在它的执行代码中调用进程等待的方法，等待子进程退出，防止子进程变成僵尸进程;

也就是说，进程等待是父进程调用某个接口进行等待，父进程等待子进程退出回收子进程的资源，防止子进程变成僵尸进程

进程等待的方法（如何让父进程进行进程等待）：wait函数和waitpid函数

wait函数：

pid\_t wait (int\* status)

参数是一个指针类型，但是该指针类型并不是要传递一个指针参数，而是一个输出型参数;

将wait函数内部计算的某个结果通过status变量返回给调用者；

之前所接触的输出型参数有：swap函数中的参数，交换两数的值函数进行传址，将swap的结果传到外部

在编码时有一个代码规范：

如果是输入型，参数定义成引用；

如果是输出或者输入输出参数，参数定义成指针；

wait函数的四个特性：

1.输出型参数，与其对应的有：

输入型参数→调用者给被调用函数传参；

输入输出型参数；

2.int\* status是一个指针类型占四个字节，但是实际中只使用到后两个字节，将这两个字节分为三部分：

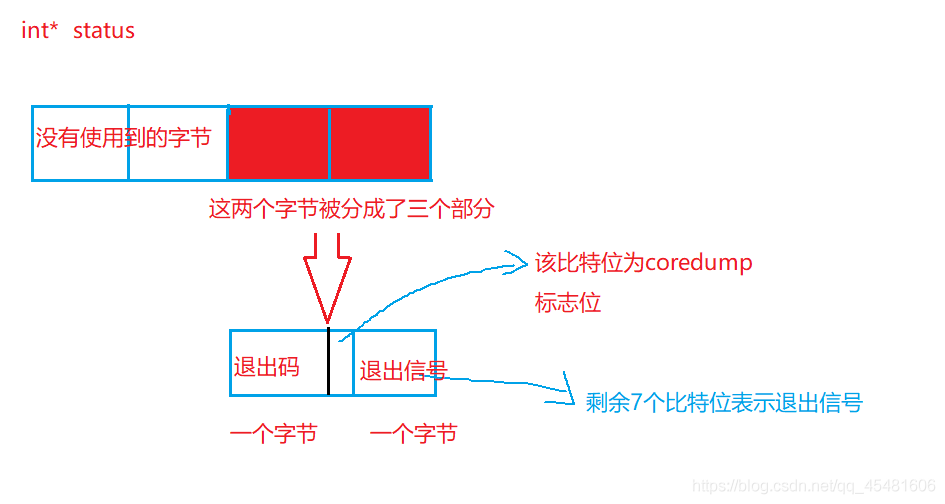
退出码 + coredump标志位 + 退出信号

高位字节表示退出码；

低位的一个字节又被分为两个部分，

第一个比特位表示coredump标志位

后七个比特位表示退出信号



退出码：程序正常退出时用到

main函数的return返回时，return后的数字就是退出码

exit和\_exit中的参数就是退出码

coredump标志位，退出信号是程序异常退出时用到:

为0表示该进程在退出时没有内存镜像文件的产生（也就是说没有coredump产生）

为1则有镜像文件产生，当前程序退出异常

退出信号：进程异常退出时收到的几号信号

进程正常退出时coredump标志位和退出信号被置为全0；

退出码则是进程正常退出给的数值；

异常退出coredump标志位和退出信号会有相应的数；

也就是说程序正常退出只有退出码起作用，异常退出则要看标志位和退出信号

用退出信号判断进程是否正常退出：

产生coredump文件不能判断进程是否正常退出的原因：

程序在运行时崩溃（例如解引用空指针），该进程会收到崩溃信号；

该信号可能会让进程产生一个coredump文件，这个文件并不是收到崩溃信号就一定会产生。

是否产生coredump文件取决于：

磁盘硬限制，空间大小不足；

系统的软限制，进程可以产生coredump文件但是系统不让其产生

1.判断是否有退出信号

正常退出时，退出信号7位全=0；异常退出，退出信号>0；

所以

int \_status=wait(&status);

status & 0xfff ffff;，等于0正常退出，>0异常退出

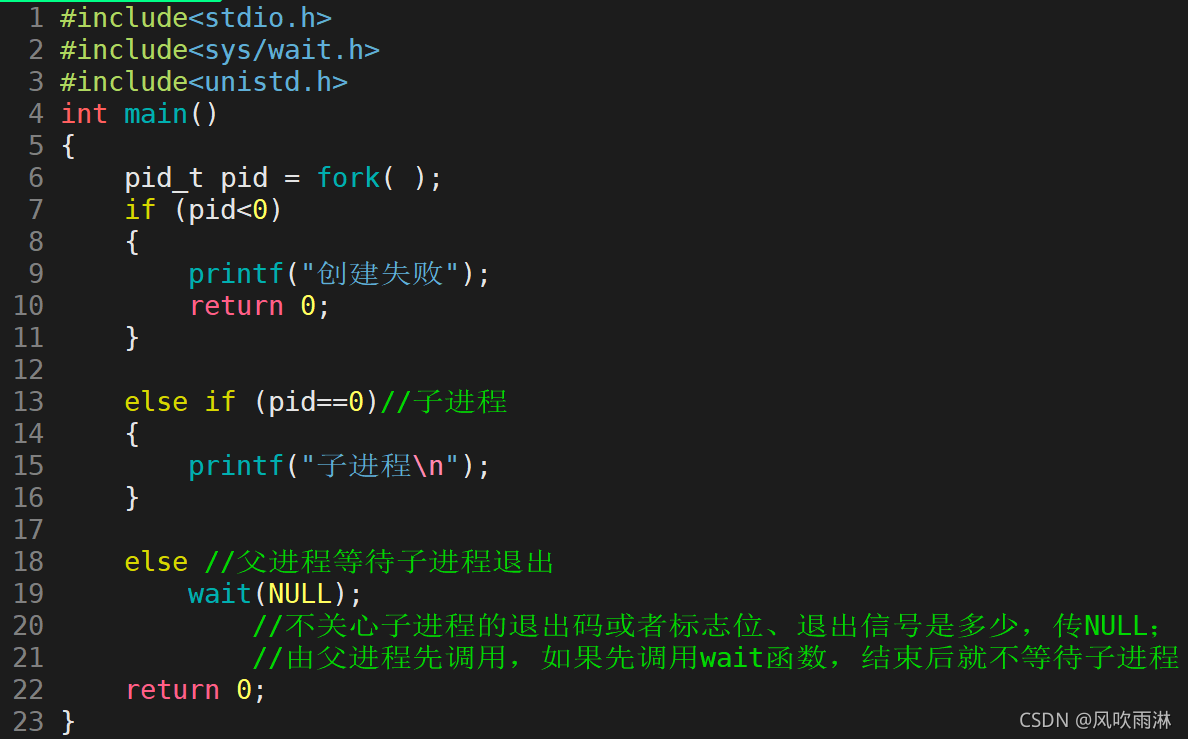
2.判断coredump标志位

(ststus >> 7) & 0x1;

3.判断退出码

(ststus >> 8) & 0xff;也就是按位与上八个一

使用wait函数阻止子进程变成僵尸进程



运行情况：

父进程创建子进程后，子进程走else if，父进程走else语句；

因为父子进程是抢占式执行，那么可能是父进程先执行wait并退出，子进程变成孤儿进程；

也可能是子进程先执行先退出，父进程等待退出

实际不管谁先运行，父进程都是正常结束的；

因为这里牵扯到阻塞的概念，父进程一旦调用wait函数后，当子进程没有退出前，wait函数是不会返回；

父进程调用wait函数，wait函数就压到父进程的栈帧上，子进程没有退出，父进程栈帧是不会销毁的，也就是说wait函数不会出栈返回

阻塞：

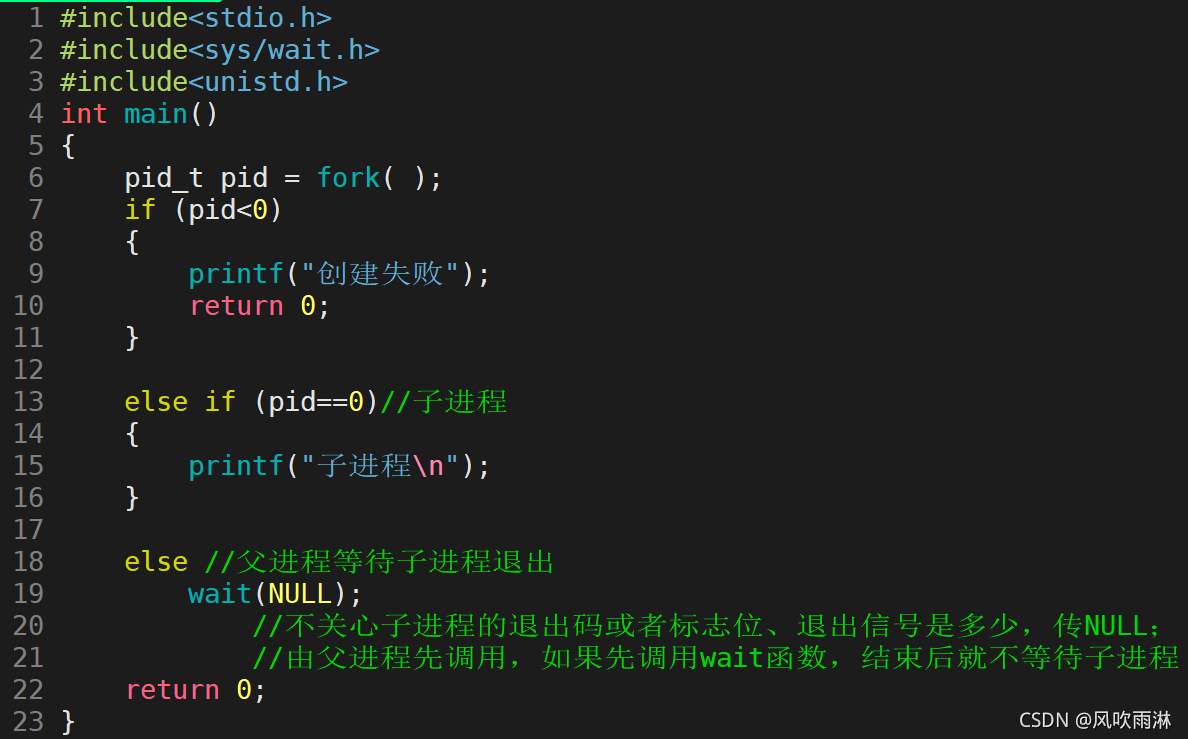
阻塞概念：当调用结果返回之前，当前的执行流会被挂起，并在得到结果之后返回

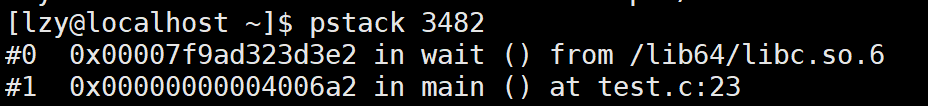
或者说：调用结果返回前，当前线程会被挂起，并在得到结果之后返回

例如：父进程被wait函数阻塞，父进程就会被挂起，必须等到子进程结束后才能返回

使用上述代码，只需给父进程处加一个死循环即可；

运行后通过pstack+父进程进程号查看父进程栈帧；





父进程一直在wait，并没有返回；

对阻塞和非阻塞理解：

1.子进程一种在运行；

阻塞：父进程等待子进程运行完后退出；这就是阻塞

非阻塞：父进程不等子进程运行是否运行完毕就退出，这样可能会导致孤儿进程；这就是非阻塞

2.子进程已退出

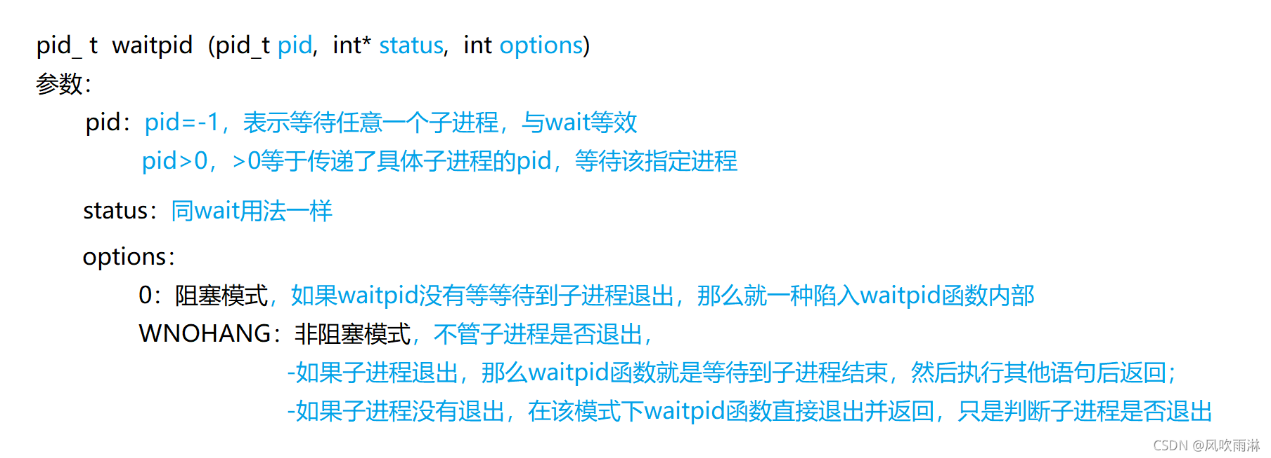
阻塞：父进程正常退出

非阻塞：父进程正常退出

对于两种非阻塞的情况，父进程都是直接退出，但是两种情况父进程退出后，一种正常一种不正常

waitpid函数

wait函数的实现是调用waitpid函数实现



非阻塞模式下waitpid函数返回是没有等待子进程的；

waitpid函数在非阻塞模式下只调用一次并且也没等待到子进程退出，再不去调用该函数；

等到子进程退出后子进程就会变成僵尸进程；

所以在非阻塞模式下需要搭配循环；

————————————————