# 进程控制

进程是操作系统的一个核心的概念。每个进程都有自己的唯一标识进程ID，也有自己的生命周期。一个典型的进程的生命周期如下图所示：



本章会介绍进程ID，进程的层次，已经进程生命周期内的各个阶段。

## 5.1进程ID

Linux下每个进程都会有一个非负整数表示的唯一进程ID，简称pid。Linux提供了getpid函数来获取进程的进程ID，同时提供了getppid函数来获取父进程的ID。

#include <sys/types.h>

#include <unistd.h>

pid\_t getpid(void);

pid\_t getppid(void);

每个进程都有自己的父进程，父进程又会有自己的父进程，最终都会追溯到1号进程init进程。这就决定了操作系统上所有的进程必然组成树状结构，就像一个家族的家谱一样。可以通过pstree 的命令来查看进程的家族树。

procfs文件系统，为每一个进程都在/proc下建立了目录，目录名为进程的PID。目录下有很多文件，记录进程的运行情况和统计信息等。因为进程有创建，有终止，所以/proc/下以进程ID为名的目录也在变化。

➜ /root ll /proc

总用量 0

dr-xr-xr-x 9 root root 0 4月 1 06:56 1

dr-xr-xr-x 9 root root 0 4月 1 06:56 10

dr-xr-xr-x 9 root root 0 4月 1 06:56 100

dr-xr-xr-x 9 root root 0 4月 1 06:56 101

dr-xr-xr-x 9 root root 0 4月 1 06:56 102

dr-xr-xr-x 9 root root 0 4月 1 06:56 103

dr-xr-xr-x 9 root root 0 4月 1 06:56 1039

dr-xr-xr-x 9 root root 0 4月 1 06:56 104

……

操作系统保证，在某一时刻，每一进程ID是唯一的，不会存在两个进程有相同的进程ID。虽然进程ID是唯一的，但是进程ID可以重用。进程退出以后，其进程ID就可以再次分配给其他进程使用。那么问题就来了，内核是如何分配进程ID的？

Linux分配进程ID的算法不同于给进程分配文件描述符的最小可用算法，采用了延迟重用的算法，即分配给新创建进程的ID尽量不与最近终止的进程重复。这种做法可以防止将新创建的进程误判为使用相同进程ID的已经退出的进程。

如何实现延迟重用？内核采用的方法如下：

1. 位图记录进程ID的分配情况（0为可用，1为已占用）
2. 将上次分配的进程ID记录到last\_pid，下次寻找可用ID，从last\_pid+1开始找起，从位图中寻找可用的ID
3. 如果找到位图集合的最后一位，仍不可用，则回滚到位图集合的起始位置，从头找起。

既然是位图记录进程ID的分配情况，那么位图的大小就必须要考虑。位图的大小决定了系统允许同时存在的最大进程的个数。这个最大进程个数在系统中称为pid\_max。

上面第三步提到，回滚到位图集合的起始位置，从头寻找可用的进程ID。事实上，也不是从0开始找起，而是从300找起。内核定义了RESERVED\_PIDS为300，300以下的PID被系统占用，不能分配给用户进程。

*kernel/pid.c*

*-------------*

define RESERVED\_PIDS 300

int pid\_max = PID\_MAX\_DEFAULT;

可以通过查看procfs来确定当前Linux所能分配的最大进程ID。

cat /proc/sys/kernel/pid\_max

32768

该上限值是可以调整的，可以将pid\_max值调大，但是内核也设置了硬上限：

root@manu-hacks:~# echo 4194304 > /proc/sys/kernel/pid\_max

root@manu-hacks:~# cat /proc/sys/kernel/pid\_max

4194304

root@manu-hacks:~# echo 4194305 > /proc/sys/kernel/pid\_max

bash: echo: write error: Invalid argument

上面操作可以看出，在我的Linux系统上，系统进程数的硬上限为4194304（4M）。内核如何决定系统进程个数的硬上限？内核定义了如下的宏：

#define PID\_MAX\_LIMIT (CONFIG\_BASE\_SMALL ? PAGE\_SIZE \* 8 : \

(sizeof(long) > 4 ? 4 \* 1024 \* 1024 : PID\_MAX\_DEFAULT))

从上面代码中可以看出决定系统进程个数硬上限的逻辑为：

* 选择了CONFIG\_BASE\_SMALL编译选项，则为页面（PAGE\_SIZE）的位数（页面的大小，与架构有关，X86平台，PAGE\_SIZE为4KB即32768个位）。
* 如果选择了CONFIG\_BASE\_FULL编译选项：
  + 32位系统，系统进程个数硬上限为32768（即32K）
  + 64位系统，系统进程个数硬上限为4194304（即4M）

从上面逻辑可以看出，正常情况下，系统进程个数的硬上限为4M个，足够使用了。

对于单线程的进程，进程ID比较好理解，就是唯一标识进程的数字。对于多线程的程序 ，每一个线程调用getpid函数，返回值都是一样的，即进程的ID。

## 5.2进程的层次

上一节提到，每个进程都有父进程，父进程也有父进程，这就形成了一个以init进程为根的家族树。除此以外，进程还有其他的层次关系：进程、进程组和会话。

进程组和会话在进程之间形成了两级的层次：进程组是一组相关进程的集合，会话是一组相关进程组的集合。用人来打比方，会话如同一个公司，进程组如同公司里的部门，进程如同部门里的员工。尽管每个员工都有父亲，但是不影响员工同时属于某个公司下的某个部门。

这样说来，一个进程会有如下ID：

* PID：进程的唯一标识。对于多线程程序而言，所有线程调用getpid()函数返回的值相同，都是线程组ID。线程组内线程，各自有各自的线程ID，或者简称为TID（第一个线程的TID的值等于线程组ID）。
* PGID：进程组ID。每个进程都会有进程组ID，表示该进程所属的进程组。默认情况下新创建的进程会继承父进程的进程组ID。
* SID：会话ID。每个进程也都有会话ID。默认情况下，新进程会父进程的会话ID。

可以调用如下指令查看所有进程的层次关系：

ps -ejH

ps axjf

对于进程而言，可以通过如下调用获取到其进程组ID和会话ID

#include <unistd.h>

pid\_t getpgrp(void);

pid\_t getsid(pid\_t pid);

前面提过，新进程默认继承父进程的进程组ID和会话ID，如果都是默认情况的话，那么追根溯源，所有的进程应该有共同的进程组ID和会话ID。但是调用ps axjf可以看到，实际情况并非如此，存在很多不同的会话，每个会话下也有不同的进程组。为何如此?

就像家族企业，如果从创业之初，所有家族成员都墨守成规，循规蹈矩，默认情况下，只有一个公司，一个部门。但是也有些叛逆的子弟，愿意为家族公司开疆拓土，愿意成立新的部门。这些新的部门就是新创建的进程组。如果存在子弟离经叛道，甚至不愿意在家族公司里，他别开天地，另创了一公司，这个新公司就是新创建的会话组。有此可见，系统须有改变设置进程组ID和会话ID的函数接口，否则，系统中只会存在一个会话，一个进程组。

进程组和会话这两个概念，是为shell作业控制而生的，无论调用login（可以通过调用ctrl+alt+F1~F6登陆终端），还是通过ssh登陆，Linux都会创建一个会话。

PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND

1 1097 1097 1097 tty1 3864 Ss 0 0:00 /bin/login --

1097 3849 3849 1097 tty1 3864 S 1000 0:00 \\_ -bash

3849 3864 3864 1097 tty1 3864 S+ 1000 0:00 \\_ sleep 1000

…

1 994 994 994 ? -1 Ss 0 0:00 /usr/sbin/sshd –D

994 2397 2397 2397 ? -1 Ss 0 0:00 \\_ sshd: manu [priv]

2397 2475 2397 2397 ? -1 S 1000 0:00 \\_ sshd: manu@pts/7

2475 2476 2476 2476 pts/7 4068 Ss 1000 0:00 \\_ -bash

2476 4068 4068 2476 pts/7 4068 R+ 1000 0:00 \\_ ps -ajxf

5.2.1进程组

修改进程的进程组ID的接口如下：

#include <unistd.h>

int setpgid(pid\_t pid, pid\_t pgid);

这个函数的含义是，把进程ID为pid的进程的进程组ID修改为pgid，如果pid的值为0，那么表示要修改调用进程的进程组ID。这个接口很容易称为创建进程组的接口：

setpgid(0,0)

setpgid(getpid(),0)

setpgid(getpid(),getpid())

上述三个接口含义一致，都是创立新的进程组，并且指定的进程会成为进程组的首进程。如果参数pid和pgid的值不匹配，那么setpgid函数会将一个进程从源进程组迁移到目的进程组。

setpgid函数有很多限制：

* pid参数必须指定为调用setpgid函数的进程或者其子进程，不能随意修改不相关进程的进程组ID，违反这条规则，则返回-1，并置errno为ESRCH
* pid参数可以指定调用进程的子进程，但是子进程如果已经执行了exec函数，不能修改子进程的进程组ID。违反这条规则，则返回-1，并置errno为EACCESS
* 在进程组间移动，调用进程，pid指定的进程，及目标进程组必须在同一个会话之内。这个比较好理解，不加入公司（会话），就无法加入公司下属的部门（进程组），否则部门是造反的节奏。违反这条规则，返回-1，并置errno为EPERM
* pid指定的进程，不能是会话首进程。违反这条规则，则返回-1，并置errno为EPERM

有了创建进程组的接口，新创建的进程组就不必继承父进程的进程组ID了。最常见的创建进程组的场景就是在shell执行管道命令，当然，shell和login也会创建新的进程组ID，但是它们更主要的是创建会话。

cmd1 | cmd2 | cmd3

在shell中执行上面的命令，会发现，因为在同一个shell中，所以cmd1、cmd2和cmd3三个进程的会话ID相同。三个命令组成了管道命令（前一个命令的标准输出作为后一个命令的标准输入），这三者的进程组ID相同。

下面用一个简单的测试验证下：

manu@manu-hacks:~$ sleep 10 | sleep 20 |sleep 30 |sleep 40 &

[1] 6417

manu@manu-hacks:~$ ps -o pid,ppid,pgid,sid,cmd |cat

  PID  PPID  PGID   SID CMD

 4892  2391  4892  4892 bash

 6414  4892  6414  4892 sleep 10

 6415  4892  6414  4892 sleep 20

 6416  4892  6414  4892 sleep 30

 6417  4892  6414  4892 sleep 40

 6418  4892  6418  4892 ps -o pid,ppid,pgid,sid,cmd

 6419  4892  6418  4892 cat

 可以看出，sleep 10，sleep 20，sleep 30，sleep 40通过管道相连（图中的箭头表示管道的流向）。通过管道相连的一组shell命令组成了一个进程组。ps和cat两个进程也是通过管道相连，两者也组成了一个进程组。

其中调用sleep管道命令时，尾部加入了&符号，表示把进程组放到后台执行，该进程组为会话的一个后台进程组。共享一个控制终端的会话里面，可以有很多后台进程组，但是在任意时刻只能有一个前台进程组。只有前台进程组中的进程组才可以在控制终端读取输入。用户在终端中输入信号生成终端字符后，信号会发送给前台进程组的所有进程，不会影响到后台进程组的进程。所谓信号生成终端字符，即信号一章会提到的：

* Ctrl+c负责生成SIGINT信号
* Ctrl+\ 负责生成SIGQUIT信号
* Crtl+z 负责生成SIGTSTP信号

前文提到子进程一旦执行exec，父进程就无法调用setpgid函数来设置子进程的进程组ID，这条规则会影响shell的作业控制。为了安全计，一般父进程调用fork创建子进程后，父进程会调用setpgid函数设置子进程的进程组ID，子进程也要调用setpgid函数来设置自身的进程组ID。这两次调用，有一次是冗余的，但是这样做，能够保证，无论从父进程来看，还是从子进程来看，子进程一定已经进入了指定的进程组。如果不这样做，由于fork之后，父子进程的执行顺序不一定，就会造成在一定的时间窗口内，无法确定子进程是否进入相应的进程组。

可以通过跟踪bash进程的系统调用来证明这一点，下面的2258是bash，我们在该bash上执行sleep 100，在执行之前，在另一个终端用strace跟踪bash的系统调用：

manu@manu-hacks:~$ sudo strace -f -p 2258

Process 2258 attached

…

[pid 2258] setpgid(2509, 2509 <unfinished ...>/\*父进程调用setpgid函数\*/

…

[pid 2509] setpgid(2509, 2509 <unfinished ...>/\*子进程调用setpgid函数\*/

…

[pid 2509] execve("/bin/sleep", ["sleep", "200"], [/\* 31 vars \*/]) = 0 /\*子进程执行exec\*/

…

5.2.2 会话

会话是一个或者多个进程组的集合。

系统提供setsid函数来创建会话：

#include <unistd.h>

pid\_t setsid(void);

如果这个函数的调用进程不是进程组组长，调用该函数会发生以下事情：

1. 创建一个新会话，会话ID等于进程ID，调用进程成为会话的首进程
2. 创建一个进程组，进程组ID等于进程ID，调用进程成为进程组的组长
3. 该进程没有控制终端，如果调用setsid前，该进程有控制终端，这种联系会断掉

shell提供了setsid命令，可以在新的会话中执行命令，我们可以通过该命令很容易地验证上述三点：

manu@manu-hacks:~$ setsid sleep 100

manu@manu-hacks:~$ ps ajxf

PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND

…

1 4469 4469 446 9 ? -1 Ss 1000 0:00 sleep 100

从输出可以看出，系统创建了新的会话4469，新的会话下创建了新的进程组，会话ID和进程组ID都等于进程ID，而该进程已经不再任何终端上了（TTY对应的值为？表示进程没有控制终端）。

如果调用进程是进程组的组长，函数调用会失败，返回-1，并置errno为EPERM。这个限制是比较合理的。如果允许进程组组长迁移到新的会话，那么进程组的其他成员仍然在老的会话中，就会出现同一个进程组的进程，分属不同的会话之中，这就破坏了进程组和会话的严格层次关系了。

常用的调用setsid函数的场景是提到过的login和shell。除此以外创建daemon进程也要调用setsid函数。

如何变成一个daemon进程？daemon和会话有什么关系，daemon和控制终端有什么关系？在创建daemon进程这一节会介绍。

## 5.3进程的创建之fork()

Linux系统下，进程可以调用fork函数来创建新的进程。调用进程为父进程，被创建的进程为子进程。子进程获得了父进程的栈，数据段，堆和可执行文本段的拷贝。

#include <unistd.h>

pid\_t fork(void);

和普通函数不同，fork函数会返回两次。一般说来，创建两个完全相同的进程，并没有太多的价值。大部分的情况下，父子进程执行不同的代码分支。fork函数的返回值是区分父子进程的关键。fork函数向子进程返回0，并将子进程的进程ID返给父进程。当然了，如果fork失败，该函数返回-1，并设置errno。

所以一般而言，调用fork的程序，大多如此。

ret = fork();

if(ret == 0)

{

…//此处是子进程的代码分支

}

else if(ret > 0)

{

…//此处是父进程的代码分支

}

else

{

…// fork失败，执行error handle

}

fork之后，父子进程，谁先获得CPU资源，得以运行？当然，在多处理器系统中，父子进程可能同时获得CPU资源。内核从2.6.32开始，父进程成为fork之后，默认情况下优先调度的对象。POSIX标准和Linux都没有公开宣布优先调度父进程。因此在应用中，决不能对父子进程执行的顺序做任何的假设。如果确实需要某一特定的执行顺序，那么需要使用进程间同步的手段。

5.3.1 fork之后父子进程的内存关系

fork之后的子进程，完全拷贝的父进程的地址空间，包括栈，堆，代码段。通过下面示例代码，来查看父子进程的内存关系。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <errno.h>

#include <sys/types.h>

#include <wait.h>

int g\_int = 1;

int main()

{

int local\_int = 1;

int \*malloc\_int = malloc(sizeof(int));

\*malloc\_int = 1;

pid\_t pid = fork();

if(pid == 0) /\*子进程\*/

{

local\_int = 0;

g\_int = 0;

\*malloc\_int = 0;

fprintf(stderr,"[CHILD] child change local global malloc value to 0\n");

free(malloc\_int);

sleep(10);

fprintf(stderr,"[CHILD] child exit\n");

exit(0);

}

else if(pid < 0)

{

printf("fork failed (%s)",strerror(errno));

return 1;

}

fprintf(stderr,"[PARENT] wait child exit\n");

waitpid(pid,NULL,0);

fprintf(stderr,"[PARENT] child have exit\n");

printf("[PARENT] g\_int = %d\n",g\_int);

printf("[PARENT] local\_int = %d\n",local\_int);

printf("[PARENT] malloc\_int = %d\n",local\_int);

free(malloc\_int);

return 0;

}

故意定义了一个位于数据段的全局变量，一个位于栈上的局部变量，还有一个通过malloc动态分配，位于堆上的变量，三者的值都是1。然后调用fork，创建出子进程，将桑格变量的值都改成0。

按照fork的语义，子进程完全拷贝了父进程的数据段，栈，堆上的内存，如果父子进程对相应数据进行修改，两个进程是并行不悖互不影响的。因此，上面示例代码，尽管子进程将三个变量的值都改成了0，对父进程而言这三个值都没变化，仍然是1：

[PARENT] wait child exit

[CHILD] child change local global malloc value to 0

[CHILD] child exit

[PARENT] child have exit

[PARENT] g\_int = 1

[PARENT] local\_int = 1

[PARENT] malloc\_int = 1

实验结果的确如此。

前文提到过，很少有创建出来的父子进程执行一模一样的代码。Linux提供了execve系统调用，构建在这个系统调用之上，有一个exec系列库函数。这个系列函数会丢弃现存的程序代码段，并且会重新为新的进程构建数据段，栈以及堆。通常情况下，调用fork，然后在子进程中调用execve系统调用或者exec家族库函数，是更常见的。在这种情况下，子进程完全copy父进程的地址空间，这就是一种浪费，因为接下来的exec系列函数会毫不留情的抛弃刚刚辛苦拷贝的内存，因此，fork时彻底拷贝父进程的数据段，栈和堆，性能是比较低的。为了解决这个问题，Linux引入了写时拷贝(copy-on-write)的技术。

写时拷贝是指，子进程的页表项指向和父进程相同的物理内存页，这样只拷贝父进程的页表项就可以了，当然要把这些页面标记成只读。如果父子进程都不修改内存地址，大家相安无事，共用一份物理内存页，但是一旦父子进程中任何一方尝试修改，内核就会发现进程尝试修改一个只读页面，从而引发缺页异常(page fault)。内核尝试为该页面创建一个新的物理页面，并将内容真正的复制到新的物理页面中，让父子进程真正地各自拥有自己的物理内存页，然后将页表中相应的表项标记为可写。



从上面的描述可以看出，对于没有修改的页面，内核并没有真正地复制物理内存页，仅仅是复制了父进程的页表。这种机制的引入，提升了fork的性能，从而使内核可以快速地创建一个新的进程。

从内核代码层面讲，调用关系如下：

 在这个调用关系中，到了copy\_page\_range，就到了复制父进程的页表。Linux的内存管理使用四级页表。上图中的第二行的函数名顾名思义也就知道这些函数的作用了。



在最后的copy\_one\_pte函数中有如下代码，将页表设置为写保护，一旦父子进程尝试修改，就会触发缺页异常。可以写测试代码，fork之后，修改变量的值，是否会引发缺页异常[[1]](#footnote-1)。

/\*如果是写时拷贝，无论是初始页表，还是拷贝的页表，都设置写保护

\*后面无论父子进程，修改页表对应位置的内存时，会触发page fault\*/

if (is\_cow\_mapping(vm\_flags)) {

ptep\_set\_wrprotect(src\_mm, addr, src\_pte);

pte = pte\_wrprotect(pte);

}

当父子进程中，任意一个由于尝试修改写保护的页面时，引发缺页中断，最后内核会走向do\_wp\_page函数，该函数会负责创建副本，并插入到进程的页表中。

通过写时拷贝技术，极大的提升了fork的性能，一定程度上让vfork成为了鸡肋。

5.3.2 fork之后父子进程与文件的关系

执行fork()时，内核会复制父进程所有的文件描述符，所以父进程打开的所有文件，对于子进程完全可以和父进程一样操作这些文件。那么父子进程同时操作同一个文件，是并行不悖，还是互相影响呢？

说的好像比较笼统，举个例子。glibc的read函数并没有将偏移量作为入参，但是每次调用read函数或者write函数，都能接着上次读写的位置，继续读写。原因是已经将偏移量信息记录在fd相关的数据结构。确切地说，内核维护了fd对应文件的相关信息，如文件偏移量f\_pos。那么问题来了，父子进程是共用一个文件偏移量，还是各自有各自的文件偏移量呢？

/\*read 和write 都没有将pos信息作为入参\*/

ssize\_t read(int fd, void \*buf, size\_t count);

ssize\_t write(int fd, const void \*buf, size\_t count);

我们用事实说话，给出一个例子：

#include <stdio.h>

#include <string.h>

#include <strings.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <errno.h>

#define INFILE "./in.txt"

#define OUTFILE "./out.txt"

#define MODE S\_IRUSR |S\_IWUSR|S\_IRGRP|S\_IWGRP|S\_IROTH

int main(void)

{

int fd\_in,fd\_out;

char buf[1024];

memset(buf, 0, 1024);

fd\_in = open(INFILE, O\_RDONLY);

if(fd\_in < 0 )

{

fprintf(stderr,"failed to open %s, reason(%s)\n",INFILE,strerror(errno));

return 1;

}

fd\_out = open(OUTFILE,O\_WRONLY|O\_CREAT |O\_TRUNC,MODE);

if(fd\_out < 0)

{

fprintf(stderr,"failed to open %s, reason(%s)\n",OUTFILE,strerror(errno));

return 1;

}

fork();

while(read(fd\_in, buf, 2) > 0)

{

printf("%d: %s",getpid(),buf);

sprintf(buf, "%d Hello,World!\n",getpid());

write(fd\_out,buf,strlen(buf));

sleep(1);

memset(buf, 0, 1024);

}

}

INFILE的内容是：

1

2

3

4

5

6

上面的父子进程都会去读INFILE，如果父子进程各自维护各自的文件偏移量，那么父子进程都会打印出1到6。

事实如何呢？输出如下：

manu@manu-hacks:~/code/self/c/fork$ ./fork\_file

6602: 1

6603: 2

6602: 3

6603: 4

6602: 5

6603: 6

有时候打印是这样的：

manu@manu-hacks:~/code/self/c/fork$ ./fork\_file

6610: 1

6611: 2

6610: 3

6611: 4

6610: 5

6611: 5

6610: 6

如果父子进程各自各自的文件偏移量，那么一定是打印出两套1到6。但是事实并非如此。无论父进程还是子进程通过read导致文件偏移量后移，都会被对方获知，这表明父子进程共用了一套文件偏移量。

对于第二个输出，为什么父子进程都打印5呢？这是因为我的机器是多核，父子进程同时被调度，发现当前文件偏移量是4\*2，然后各自去读了第八第九字节也就是“5\n”。如果我们将程序绑定在一个核上执行，如下面执行方法，就不会出现这种现象。

taskset -c 0x00 ./fork\_file

对于写文件也是一样，如果fork之前打开某文件，父子进程如果写入同一个文件描述符而又不采取任何同步的手段，因为共享文件偏移量，就会使输出相互混合，不可阅读。

除了文件偏移量，还有文件状态标志（file status flags）。文件状态标志在open的时候设置，某些标志位可以调用fcntl函数的F\_SETFL操作来改变。文件标志位分成以下三类：

* 获取模式（access mode）：简单地说就是控制文件描述符是否可读可写。这类标志位有只读（O\_RDONLY），只写（O\_WRONLY）和读写（O\_RDWR）.
* 打开时标志：这种标志位只在打开时起作用，主要指导当打开文件时，遇到不同情况如何处理。如O\_CREAT标志位，如果置位，文件不存在时，创建文件；如O\_EXCL标志位，，当O\_EXCL和O\_CREAT同时置位，文件存在时，会返回错误EEXIST；如O\_TRUNC，打开时，发现文件长度不是0，将文件截断成空文件等等。
* IO操作模式（I/O operating modes）：这些标志位是指导IO操作的。如O\_APPEND标志位，总是在文件的末尾追加写入要写的数据；如O\_NONBLOCK标志位，打开文件的非阻塞模式。比如read，如果当前没有可以读的内容，立刻返回失败而不是阻塞与此等待输入。

上面三种标志位，第一类和第二类不能被fcntl修改，但是第三类可以open以后，调用fcntl函数进行修改。这些标志位，也是fork之后父子进程共享的。

除了文件状态标志，文件描述符还有一个文件描述符标志（file descriptor flags）。目前只定义了一个标志位：FD\_CLOSEXEC，这是close\_on\_exec标志位。细心阅读open函数手册也会发现，open有一个标志位是O\_CLOSEXEC，该标志位也是设置文件描述符标志的。那么这个标志位是干什么的呢？顾名思义，就是调用exec时自动关闭这个文件。为什么需要这么个标志位呢？主要出于安全的考虑。

对于fork之后，子进程执行exec家族的函数这种场景，如果子进程可以操作父进程打开的文件，这会带来安全的隐患[[2]](#footnote-2)[[3]](#footnote-3)。一般来讲，调用exec的子进程，完全另起炉灶，丢弃旧的程序，从父进程拷贝来的栈，堆，数据段被新程序的相应部件替换，这种情况下，父进程的打开的文件描述符也应该一并关闭，但事实上并没有。试想如下场景，Webserver会首先以root权限启动，打开只有root权限才能打开的端口和日志等文件，再降到普通用户，fork出一些worker进程，在进程中进行解析脚本，写日志，输出结果等操作。由于子进程完全可以操作父进程打开的文件，子进程中的脚本只要继续操作这些文件描述符，就能越权操作root用户才能操作的文件。

为了解决这个问题，Linux引入了close on exec机制。设置了FD\_CLOSEXEC标志位的文件，在子进程调用exec家族函数时，会将相应的文件关闭。而设置该标志位的方法有两种：

* open时，带上O\_CLOSEXEC标志位
* open时如果没设置，可以调用fcntl函数的F\_SETFD操作。

这两种方法中，建议使用第一种，原因是，第二种方法在某些时序条件下并不那么绝对的安全。我们考虑如下场景：Thread1还没有来得及将FD\_CLOSEXEC置位，由于Thread2已经fork，这时候fork出来的子进程就不会关闭相应的文件。尽管Thread1后来调用了fcntl的F\_SETFD操作，但是为时已晚，文件已经泄漏出去了。



前面提到，执行fork时，子进程会获取父进程所有文件描述符的副本，但是测试结果表明，父子进程共享了文件的很多属性。这到底是怎么回事。让我们深入内核，一探究竟。

5.3.3 文件描述符复制的内核实现

在内核的进程描述符task\_struct结构体中，和打开文件相关的变量如下所示：

struct task\_struct {

/\* open file information \*/

struct files\_struct \*files;

}

对于fork出的子进程而言，处理打开文件相关的流程在copy\_files函数：

static int copy\_files(unsigned long clone\_flags,

struct task\_struct \*tsk)

{

struct files\_struct \*oldf, \*newf;

int error = 0;

oldf = current->files;

if (!oldf)

goto out;

/\*创建线程和vfork，都不用复制父进程的文件描述，增加引用计数即可\*/

if (clone\_flags & CLONE\_FILES) {

atomic\_inc(&oldf->count);

goto out;

}

/\*对于fork而言，需要复制父进程的文件描述符\*/

newf = dup\_fd(oldf, &error);

if (!newf)

goto out;

tsk->files = newf;

error = 0;

out:

return error;

}

CLONE\_FILES标志位，用来控制是否共享父进程的文件描述符。如果该标志位置位，表示，不要费劲复制一份父进程的文件描述符了，增加引用计数，直接共用一份就可以了。对于vfork函数和创建线程的pthread\_create函数都是如此。但是fork函数不同，调用fork函数时，该标志位为0，表示需要为子进程拷贝一份父进程的文件描述符。文件描述符的拷贝是内核的dup\_fd函数完成的

struct files\_struct \*dup\_fd(struct files\_struct \*oldf,

int \*errorp)

{

struct files\_struct \*newf;

struct file \*\*old\_fds, \*\*new\_fds;

int open\_files, size, i;

struct fdtable \*old\_fdt, \*new\_fdt;

\*errorp = -ENOMEM;

newf = kmem\_cache\_alloc(files\_cachep, GFP\_KERNEL);

if (!newf)

goto out;

dup\_fd函数上来就分配了一个file\_struct结构体。然后做了一些赋值操作。这个结构体是进程描述符与打开文件相关的数据结构，定义如下：

struct files\_struct {

atomic\_t count;

struct fdtable \_\_rcu \*fdt;

struct fdtable fdtab;

/\*

\* written part on a separate cache line in SMP

\*/

spinlock\_t file\_lock \_\_\_\_cacheline\_aligned\_in\_smp;

int next\_fd;

struct embedded\_fd\_set close\_on\_exec\_init;

struct embedded\_fd\_set open\_fds\_init;

struct file \_\_rcu \* fd\_array[NR\_OPEN\_DEFAULT];

};

struct fdtable

{

unsigned int max\_fds;

struct file \_\_rcu \*\*fd; /\* current fd array \*/

fd\_set \*close\_on\_exec;

fd\_set \*open\_fds;

struct rcu\_head rcu;

struct fdtable \*next;

};

struct embedded\_fd\_set {

unsigned long fds\_bits[1];

};

这个file\_struct就是管理进程打开文件的结构体，每一个打开的文件都会记录在这个结构体中。



初看之下struct fdtable的内容和struct files\_struct的内容颇多重复，如exec时关闭的文件描述符位图和打开文件描述符位图及file指针数组。事实上并非如此。struct files\_struct中的成员是相应数据结构的实例，而struct fdtable中的成员是相应的指针。在初始化时，执行如下步骤

* fdtable结构体中的fd指针指向一个file\_struct结构体中的fd\_array成员变量，该成员变量是指针数组，每一个指针都可以指向一个struct file结构体
* fdtable结构体中的open\_fds指针指向file\_struct结构体中的open\_fd\_init成员变量，该成员变量是个位图，每一个bit来表示对应位置的文件描述符是否已经分配出去。
* fdtable结构体中的close\_on\_exec指针指向file\_struct结构体中的close\_on\_exec\_init成员变量，该成员变量也是一个位图，每一bit记录对应位置的文件描述符执行exec时，是否需要关闭该文件描述符。

Linux系统假设，大多数情况下，进程打开的文件数不会太多。Linux选择了一个long类型的bit数（32位系统下为32，64位系统下为64位）作为门槛。

以64位系统为例讲述，file\_struct结构体自带了可以容纳64个struct file类型指针的数组，也自带了两个大小是64的位图。只要Linux打开的文件个数小于64。file\_struct结构体就足以满足需要。下面是open系统调用的基本流程，分析系统如何将文件与进程描述符关联起来：

1. 调用open函数打开文件时，第一步是寻找一个可用的文件描述符。比如找到的空闲文件描述符为9，在这一步骤中会将open位图和对应的close\_on\_exec位图做相应的改变。

fd = get\_unused\_fd\_flags(flags);

1. 分配struct file结构体，打开文件。

struct file \*f = do\_filp\_open(dfd, tmp, &op);

1. 将进程的文件描述符和内核的struct file结构体关联起来。对于我们的例子，就是将struct file\_struct结构体中的fd\_array[9]指向struct file结构体

fd\_install(fd, f);

对于打开数目小于64的情况，无论是open位图，close\_on\_exec位图还是指向struct file的指针，都是struct files\_struct 里的成员变量。一旦进程打开的文件超过64个，struct files\_struct就不能满足需要了。举个例子说， 家中餐具可以招待八个客人，将要到来的客人数目未知，客人陆续到来。在客人不超过八个情况下，在家中招待。随着第九个客人的到来，不得不去酒店订一个大包间了，并将客人请到酒店。

这种情况下，内核会分配一个新的struct fdtable。在新的结构体中，无论两个位图还是struct file指针数组都足够大，然后将位图信息和struct file指针从老的fdtable拷贝到新的fdtable，将files\_struct中的fdt指针指向新的fdtable。这部分逻辑在内核的get\_unused\_fd\_flags函数中调用的expand\_files函数实现。

fork函数中，子进程会复制父进程的文件描述符，分配了file\_struct结构体后，也会初始化file\_struct中自带的fdtable，如下代码所示

atomic\_set(&newf->count, 1);

spin\_lock\_init(&newf->file\_lock);

newf->next\_fd = 0;

new\_fdt = &newf->fdtab;

new\_fdt->max\_fds = NR\_OPEN\_DEFAULT;

new\_fdt->close\_on\_exec = (fd\_set \*)&newf->close\_on\_exec\_init;

new\_fdt->open\_fds = (fd\_set \*)&newf->open\_fds\_init;

new\_fdt->fd = &newf->fd\_array[0];

new\_fdt->next = NULL;

如果父进程打开的文件特别多，如果超过了64个，那么子进程struct files\_struct内置的指针数组和两个位图就不能满足要求了，这时候，需要为子进程分配新的能管理更多文件的fdtable。

spin\_lock(&oldf->file\_lock);

old\_fdt = files\_fdtable(oldf);

open\_files = count\_open\_files(old\_fdt);

/\*如果父进程打开文件超过NR\_OPEN\_DEFAULT\*/

while (unlikely(open\_files > new\_fdt->max\_fds)) {

spin\_unlock(&oldf->file\_lock);

/\* 如果不是自带的fdtable，而是曾经分配的fdtable，需要释放

\* 对于fork而言，条件不会满足，不会执行\_\_free\_fdtale \*/

if (new\_fdt != &newf->fdtab)

\_\_free\_fdtable(new\_fdt);

/\*创建新的fdtable\*/

new\_fdt = alloc\_fdtable(open\_files - 1);

if (!new\_fdt) {

\*errorp = -ENOMEM;

goto out\_release;

}

/\*超出了系统限制，返回EMFILE\*/

if (unlikely(new\_fdt->max\_fds < open\_files)) {

\_\_free\_fdtable(new\_fdt);

\*errorp = -EMFILE;

goto out\_release;

}

spin\_lock(&oldf->file\_lock);

old\_fdt = files\_fdtable(oldf);

open\_files = count\_open\_files(old\_fdt);

}

空间分配好了，指向struct file的指针够用了，位图也够用了，接下来需要把父进程的两个位图信息和打开文件的struct file指针拷贝到子进程。

old\_fds = old\_fdt->fd; /\*父进程的struct file 指针数组\*/

new\_fds = new\_fdt->fd; /\*子进程的struct file 指针数组\*/

/\* 拷贝打开文件位图 \*/

memcpy(new\_fdt->open\_fds->fds\_bits,

old\_fdt->open\_fds->fds\_bits, open\_files/8);

/\* 拷贝 close\_on\_exec位图 \*/

memcpy(new\_fdt->close\_on\_exec->fds\_bits,

old\_fdt->close\_on\_exec->fds\_bits, open\_files/8);

for (i = open\_files; i != 0; i--) {

struct file \*f = \*old\_fds++;

if (f) {

get\_file(f); /\* f对应的文件的引用计数加1 \*/

} else {

FD\_CLR(open\_files - i, new\_fdt->open\_fds);

}

/\* 子进程的struct file类型指针，

\*指向和父进程同一个struct file 结构体\*/

rcu\_assign\_pointer(\*new\_fds++, f);

}

spin\_unlock(&oldf->file\_lock);

/\* compute the remainder to be cleared \*/

size = (new\_fdt->max\_fds - open\_files) \* sizeof(struct file \*);

/\*将尚未分配到的struct file结构的指针清零\*/

memset(new\_fds, 0, size);

/\*将尚未分配到的位图区域清零\*/

if (new\_fdt->max\_fds > open\_files) {

int left = (new\_fdt->max\_fds-open\_files)/8;

int start = open\_files / (8 \* sizeof(unsigned long));

memset(&new\_fdt->open\_fds->fds\_bits[start], 0, left);

memset(&new\_fdt->close\_on\_exec->fds\_bits[start], 0, left);

}

rcu\_assign\_pointer(newf->fdt, new\_fdt);

return newf;

out\_release:

kmem\_cache\_free(files\_cachep, newf);

out:

return NULL;

}

再次强调，父子进程之间，拷贝的是struct file的指针，并非struct file实例，父子进程的struct file类型的指针，都指向同一个struct file实例。看下struct file成员变量：

struct file{

…

unsigned int f\_flags

fmode\_t f\_mode

loff\_t f\_pos; /\*文件位置指针的当前值，即文件偏移量\*/

}

到此处，就可以理解为何上一小节，父子进程为何共享文件偏移量，共享文件的文件标志位，因为父子进程的指针都指向了同一个struct file。

## 5.4进程的创建之vfork()

早期的实现，fork不是写时拷贝，而是会将父进程的数据段，堆，栈实施严格的拷贝。效率十分低下。很多的程序在fork一个子进程后，紧接着执行exec家族函数，更是一种浪费。所以BSD引入了vfork。既然fork之后，会执行exec函数，拷贝父进程的内存数据就变成了无意义的行为，所以引入了vfork，压根不拷贝父进程的内存数据，直接共享。再后来Linux引入了写时拷贝的机制，其效率提高了很多，所以vfork其实可以退出历史舞台。

vfork会创建一个子进程，子进程会共享父进程的内存数据，系统将保证子进程先于父进程获得调度。子进程会共享父进程的地址空间，父进程会一直挂起，直到子进程退出或者执行exec。

注意，vfork之后，子进程如果返回，不要调用return，而应该使用\_exit函数。如果使用return，会有诡异的错误[[4]](#footnote-4)。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int glob = 88 ;

int main(void) {

int var;

var = 88;

pid\_t pid;

if ((pid = vfork()) < 0) {

printf("vfork error");

exit(-1);

} else if (pid == 0) { /\* 子进程 \*/

var++;

glob++;

return 0;

}

printf("pid=%d, glob=%d, var=%d\n", getpid(), glob, var);

return 0;

}

调用子进程，如果用return返回，就意味着main函数return了，因为栈是父子进程共享的，所以程序的函数栈发生了变化。main函数return之后，通常会调用exit系的函数，父进程收到子进程exit之后，开始从vfork返回，但是整个main函数的栈都已经不复存在了，所以父进程压根无法执行。栈会返回一个诡异的栈地址，对于某些内核版本，直接报栈错误就跪安了，但是某些内核版本，有可能会再次进出入main，于是进入一个无线循环，知道vfork返回错误。对于我的Ubuntu，就是后者。

子进程用return返回时不行的，一般来说，vfork创建的子进程会exec，执行完exec后应该调用\_exit返回。注意是\_exit而不是exit。因为exit会导致父进程stdio缓冲区的冲刷和关闭。我们会在下一节讲述exit和\_exit的区别。

## 5.5 daemon进程的创建（double-fork magic）

daemon进程又被称为守护进程，一般来说它有以下两个特点

1. daemon进程生命周期很长，一旦启动，正常情况不会终止，一直运行到系统退出。凡事无绝对：daemon进程也可以停止，如很多daemon提供stop选项，执行stop可以终止daemon，或者手工发送信号杀死，或者daemon进程代码存在bug，异常退出，这些退出一般都是手工操作或者异常引发的。
2. daemon进程在后台执行，并且不拥有任何控制终端。即使daemon进程是从终端命令行启动，终端相关的信号如SIGINT、SIGQUIT和SIGTSTP以及关闭终端，都不会影响到daemon进程的继续执行

如何使一个进程变成daemon进程，或者说编写daemon进程，需要遵循那些规则或者步骤？

一般来讲，创建一个daemon进程的步骤被概括的称为double-fork magic。细细说来，需要下步骤：

1. 执行fork()函数，父进程退出，子进程继续。

这一步原因有二：

* 1. 父进程有可能是进程组的组长（在命令行启动的情况下），不能够执行后面要执行的setsid函数，子进程继承了父进程的进程组ID，并且拥有自己的进程ID，一定不会是进程组组长，所以子进程一定可以执行后面要执行的setsid函数。
  2. 如果daemon从终端命令行启动，父进程退出会被shell检测到，shell会显示shell提示符，让子进程后台执行

1. 子进程执行如下三件事，摆脱和环境的关系
   1. 修改进程的当前目录为根目录（/），这样做是有目的的，因为daemon一直运行，如果当前工作路径上包含有根文件系统以外的其他文件系统，因为daemon进程的存在，这些文件系统无法卸载。因此，常规是将当前工作目录切换成根目录，当然也可以是其他目录，只要确保所在当前工作目录所在的文件系统不会被卸载即可。

chdir(“/”)

* 1. 调用setsid函数。这个函数的目的是切断于控制终端的所有关系，并且创建一个新的会话。

这一步比较关键，因为这一步确保子进程不再归属于控制终端所关联的会话。因此无论在终端上是否发送SIGINT、SIGQUIT或SIGTSTP信号，也无论终端是否断开，都与要创建的daemon进程无关，不会影响到daemon进程的继续执行。

* 1. 设置文件模式创建掩码为0。

umask(0)

这一步的目的是daemon进程创建文件的权限属性和shell脱离关系。因为默认情况下，进程的umask来源于父进程shell的umask。如果不执行umask(0)，那么父shell的umask就会影响到daemon进程的umask，如果用户改变了shell的umask，那么也就改变了相当于改变了daemon的umask，就会造成daemon进程每次执行umask信息可能不一致。

1. 再次执行fork，父进程退出，子进程继续。

执行完前面两步之后，可以说已经比较圆满了：新建会话，进程是会话的首进程，也是进程组的首进程，进程ID，进程组ID，会话ID，三者值相同，进程和终端无关联。为何还要再执行一次fork函数？

原因是，daemon进程有可能会打开一个终端设备，即daemon进程可能根据需要，会还执行：

int fd = open("/dev/console", O\_RDWR);

这个打开的终端设备是否会成为daemon的进程的控制终端，取决于两点

* 1. daemon进程是否是会话的首进程，会话首进程才有此种可能
  2. 取决于系统实现。（BSD风格的实现不会成为daemon进程的控制终端，但是POSIX标准说这由具体实现决定）

既然如此，为了确保万无一失，只有确保daemon进程不是会话的首进程，才能保证打开的终端设备不会自动成为控制终端。因此，不得不执行第二次fork，fork之后，父进程退出，子进程继续。子进程不再是会话的首进程，也不是进程组的首进程。

1. 关闭标准输入（stdin），标准输出（stdout）和标准错误（stderr）。因为文件描述符0、1和2指向的就是控制终端。Daemon进程已经不和任意控制终端关联，因此这三者没有意义。一般来讲，关闭了之后，会打开/dev/null，并还行dup2函数，将0,1,2重定向到/dev/null。这个重定向是有意义的，防止后面程序在0,1,2上执行I/O库函数导致报错。

至此，完成了daemon进程的创建，至此，进程可以执行自己真正的工作了。

## 5.6进程的终止

不考虑线程，进程的退出有一下五种方式

正常退出有三种：

* 从main函数返回
* 调用exit
* 调用\_exit

异常退出有两种：

* 调用abort
* 接收到信号，由信号终止

5.6.1 \_exit函数

#include <unistd.h>

void \_exit(int status);

\_exit函数的status参数定义了进程的终止状态，父进程可以通过wait()获取到该状态值。

需要注意的是返回值，虽然status是int型，但是仅有低8位可以被父进程所用。所以我们写exit(-1)结束进程时，在终端执行$?发现返回值是255，就是这个道理。

如果是shell相关的编程，bash需要获取进程的退出值，那么退出值最好不要大于128。如果退出值大于128的值，会给bash带来困扰。因为按照bash的规定的：

* 如果进程正常退出，$?返回status的值。
* 如果进程收到信号退出，$?返回128+信号编号

manu@manu-hacks:~/code/me/exit$ sleep 10000

^C

manu@manu-hacks:~/code/me/exit$ $?

130：未找到命令

上述命令被SIGINT信号（signo=2）中断，返回了130。如果某程序返回130，bash无法判断是收到信号SIGINT还是进程正常退出值是130。

用户调用\_exit函数，本质会调用exit\_group系统调用。在第四章已经详细介绍过，再次不赘述。

5.6.1 exit函数

这个函数更常见一些

#include <stdlib.h>

void exit(int status);

exit()函数的最后，也会调用\_exit()函数，但是exit在调用\_exit之前，做了其他的工作：

* 执行用户通过调用atexit函数或者on\_exit定义的清理函数。
* 关闭所有打开的流(stream)，所有缓冲的数据被写入(flush)，通过tmpfile创建的临时文件都会别删除；
* 调用\_exit



第一步是执行用户注册的清理函数。用户可以通过调用atexit()函数或者on\_exit()函数来定义清理函数。这些清理函数在调用return或者调用exit的时候会被执行。执行顺序与函数注册的顺序相反。当进程收到致命信号，进程退出时，注册的清理函数不会执行，当进程调用\_exit退出时，注册的清理函数不会执行，当执行到某个清理函数出现收到致命信号或者清理函数调用了\_exit()函数，那么该清理函数不会返回，导致排在后面需要执行的清理函数都会被丢弃。

其次是冲刷stdio库提供的缓冲，关闭流。glibc提供的很多I/O相关的函数都提供了缓冲区，来缓存大块数据。缓冲有三种方式：无缓冲(\_IONBF)、行缓冲(\_IOLBF)和(\_IOFBF)。

* 无缓冲：就是没有缓冲区，每次调用stdio库函数都会立刻调用read/write系统调用。
* 行缓冲：对于输出流，收到换行符之前，一律缓冲数据，除非缓冲区满了。对于输入流，每次读取一行数据。
* 全缓冲：就是缓冲区满之前，不会调用read/write系统调用进行读写。

对于后两种，可能会出现这种情况，进程调用exit的时候，缓冲区里面还有数据，比如行缓冲迟迟没有等到换行符，比如全缓冲没有等到缓冲区满。尤其是后者，因为glibc的缓冲区大小，默认是8192字节。exit函数在关闭流之前，将缓冲区的数据冲刷。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

void foo()

{

fprintf(stderr,"foo says bye.\n");

}

void bar()

{

fprintf(stderr,"bar says bye.\n");

}

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

{

atexit(foo);

atexit(bar);

fprintf(stdout,"Oops ... forgot a newline!");

sleep(2);

if (argc > 1 && strcmp(argv[1],"exit") == 0)

exit(0);

if (argc > 1 && strcmp(argv[1],"\_exit") == 0)

\_exit(0);

return 0;

}

注意上面的示例代码，fprintf打印的字符串是没有换行符的，对于标准输出流stdout，采用的是行缓冲，收到换行符之前是不会输出的。输出情况如下

manu@manu-hacks:~/code/self/c/exit$ ./test exit

bar says bye.

foo says bye.

Oops ... forgot a newline!manu@manu-hacks:~/code/self/c/exit$

manu@manu-hacks:~/code/self/c/exit$

manu@manu-hacks:~/code/self/c/exit$ ./test

bar says bye.

foo says bye.

Oops ... forgot a newline!manu@manu-hacks:~/code/self/c/exit$

manu@manu-hacks:~/code/self/c/exit$

manu@manu-hacks:~/code/self/c/exit$ ./test \_exit

manu@manu-hacks:~/code/self/c/exit$

尽管缓冲区里的数据没有等到换行符，但是无论是调用return返回还是调用exit返回，缓冲区里的数据被刷新，Oops ... forgot a newline!都会被输出。因为exit()函数会负责此事。从测试代码的输出也可以看出，exit()函数首先执行的是用户注册的清理函数，然后才执行了缓冲区的冲刷。

如果存在临时文件，exit会负责将临时文件删除，这在第三章标准I/O库中已经介绍，不在赘述。

exit函数的最后，调用了\_exit()函数，最终殊途同归，走向了内核清理。

**return 函数**

return是我们更常见的一种终止进程的方法。return(n)等同于执行exit(n)，因为调用main()的运行时函数会将main的返回值作为exit的参数。

## 5.7 wait子进程

### 5.7.1 僵尸进程

进程就像一个生命体，通过fork()函数，子进程呱呱坠地，有的子进程子承父业，继续执行一样的程序(相同的代码段，尽管可能是不同程序分支)，有的子进程比较叛逆，通过exec离家出走，走和父进程完全不同的道路。

令人悲伤的是，如同所有的生命体一样，进程也会消亡。进程退出走到内核清理，基本是释放进程所有的资源，这些资源包括内存资源，文件资源，信号量资源，共享内存资源，或者引用计数减一，或者彻底释放。进程的退出，并没有将所有的资源释放，保留了少量的资源，进程的PID依然占用，不可被系统分配，此时的进程不可运行，事实上也没有地址空间让其运行，此时进程进入僵尸状态。

为什么进程退出之后不将所有的资源释放，从此灰飞烟灭，一了百了，反而非要保留少量资源，进入僵尸状态。看看僵尸进程依然占有系统资源，我们就能获得答案。僵尸进程依然保留的资源有进程控制块task\_struct，内核栈等。这些资源不释放，是为了提供出来很多重要的信息，如进程为何退出，是收到信号退出还是正常退出，进程退出码是多少，进程一共消耗了多少系统CPU时间，多少用户CPU时间？收到多少信号，发生多少次上下文切换，最大内存驻留集是多少，产生多少缺页中断？这些信息，就像墓志铭，总结了进程的一生。如果进程没有这个僵尸状态，进程立刻灰飞烟灭，所有这些信息也会随之流逝，系统将没有机会获知进程何时消亡，进程执行了多久，死亡原因，退出码为何等等信息。因此进程退出后，保留少量的资源，等待父进程前来搜集这些信息。一旦父进程搜集了这些信息之后(调用下面提到的wait/waitpid等函数)，这些残存的资源就完成了它的使命，就可以释放了，进程就脱离僵尸状态，彻底灰飞烟灭了。

从上面的讨论可以看出，制造出一个僵尸进程是很容易的事情，只要父进程调用fork创建子进程，子进程退出后，父进程并不调用wait或者waitpid获取子进程的退出信息，此时子进程就会沦为僵尸进程。

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

pid\_t pid;

pid=fork();

if(pid<0)

{

/\* 如果出错 \*/

printf("error occurred!\n");

}

else if(pid==0)

{

/\* 子进程 \*/

exit(0);

}

else

{

/\* 父进程 \*/

sleep(300); /\* 休眠300秒 \*/

wait(NULL); /\* 获取僵尸进程的退出信息 \*/

}

return 0;

}

上面例子中父进程休眠300秒，之后才会调用wait获取子进程的退出信息。而子进程退出之后，变成僵尸状态，苦苦等待父进程来获取退出信息。在这300秒左右的时间里，子进程就是一个僵尸进程。

如何查看一个进程是否是僵尸状态呢？ps命令输出的进程状态Z，表示的就是进程处于僵尸状态，另外procfs提供的status信息里面的State给出的值是Z(zombie)也表明进程处于僵尸状态。

ps ax

......

3940 pts/10 S 0:00 ./zombie

3941 pts/10 Z 0:00 [zombie] <defunct>

cat /proc/3941/status

Name: zombie

State: Z (zombie)

Tgid: 3941

Ngid: 0

Pid: 3941

PPid: 3940

.......

进程进入僵尸状态，就进入了一种刀枪不入的状态，杀人不眨眼的kill -9也无能为力，因为没有办法杀死一个已经死去的进程，除非父进程或者init进程调用wait获取其退出信息。如果要清除僵尸进程，只能等待父亲天良未丧，为子进程收尸，或者干掉父进程，等待init进程来收尸。

一般而言，我们不希望大量进程长期处于僵尸状态，因为浪费系统资源。除了内存资源外，比较明显的是进程ID，僵尸进程并没有将进程ID归还给系统，依然占有这个进程ID，系统不能将该ID分配给其他进程。前文提过，默认情况下进程ID的总数为3万多。

对于编程来说，如何防范大量僵尸进程的产生？答案是具体情况具体分析。

如果我们不关心子进程的退出状态，就应该将父进程对SIGCHLD的处理函数设置为SIG\_IGN，或者调用sigaction函数时设置SA\_NOCLDWAIT标志位，这两者都是明确告诉子进程，父进程很绝情，不会为子进程收尸，当子进程退出的时候，请自行了断，不要进入僵尸状态了。子进程退出的时候，内核会检查父进程的SIGCHLD信号处理结构体是否显式设置了SA\_NOCLDWAIT标志位或者将信号处理函数显式设为SIG\_IGN。如果是autoreap会为true，子进程发现autoreap为true也就死心了，调用release\_task函数自行了断了。

*kernel/signal.c do\_notify\_parent函数*

psig = tsk->parent->sighand; /\*父进程的信号处理结构体\*/

spin\_lock\_irqsave(&psig->siglock, flags);

if (!tsk->ptrace && sig == SIGCHLD &&

(psig->action[SIGCHLD-1].sa.sa\_handler == SIG\_IGN ||

(psig->action[SIGCHLD-1].sa.sa\_flags & SA\_NOCLDWAIT))){

autoreap = true;

if (psig->action[SIGCHLD-1].sa.sa\_handler == SIG\_IGN)

sig = 0;

}

如果父进程关心子进程的退出信息，应该在流程上妥善设计，能够及时的调用wait，使子进程处于僵尸状态的时间不会太久。

对于创建很多子进程的应用来说，知道子进程的返回值是有意义的。比如说父进程维护一个进程池，通过进程池里的子进程来提供服务。当子进程退出的时候，父进程需要了解子进程的返回值来确定子进程的死因，从而采取更有针对性的措施。

5.7.2等待子进程之wait()

Linux提供了wait()函数来获取子进程的退出状态。

include <sys/wait.h>

pid\_t wait(int \*status);

成功时，返回已退出子进程的进程ID；失败时，返回-1。

注意父子进程是两个进程，子进程退出和父进程调用wait()函数获取子进程退出状态是两个时间上独立的事件，就会出现下面两种情况：

1. 子进程先退出，父进程后调用wait()函数；
2. 父进程先调用wait()函数，子进程后退出；

对于第一种情况，子进程依然销毁了自己几乎所有的资源，只留下少量的信息，苦苦等待父进程来收尸，我们称这种进程为僵尸进程。当父进程调用wait()函数的时候，苦守寒窑十八载的子进程终于等到了父进程来收尸，这种情况下，父进程获取到子进程的状态信息，立刻返回。

对于第二种情况，父进程先调用wait()函数，调用时并无子进程退出，该函数调用就会阻塞，直到某个子进程退出。

wait()函数，等待的是任意一个子进程。当存在多个子进程都处于僵尸状态，wait函数获取到其中一个子进程的信息后立刻返回。wait()函数由于不会接受pid\_t类型的入参，所以wait()函数无法等待指定进程，获取其退出信息。

一个进程如何等待所有子进程退出呢？wait()函数返回有三种可能性，

* 等到了子进程退出，获取其退出信息，返回子进程的进程ID
* 等待过程中，收到了信号，信号打断了系统调用，并且注册信号处理函数时，并没有设置SA\_RESTART标志位，系统调用不会被重启，wait()函数返回-1，并且errno为EINTR；
* 已经成功等待了所有子进程，没有子进程的退出信息需要接收，这种情况下，wait()函数返回-1，errno为ECHILD。

Linux/UNIX系统编程手册给出下面的代码来等待所有子进程退出

while((childPid = wait(NULL)) != -1)

continue;

if(errno !=ECHILD)

errExit(“wait”);

上面这种方法是不全面的，因为上面方法忽略了信号中断这种情况，如果wait()函数被信号中断，上面代码并不能成功等待所有子进程。

只需要将上面的wait函数封装一下，变成信号中断后，自动重启系统调用的r\_wait()函数就完备了。如下面代码。

pid\_t r\_wait(int \*stat\_loc)

{

    int retval;

    while(((retval = wait(stat\_loc)) == -1 &&

 (errno == EINTR))

;

    return retval;

}

while((childPid = r\_wait(NULL)) != -1)

continue;

If(errno != ECHILD)

{

/\*some error happened\*/

}

如果父进程调用wait()函数时，已经有多个子进程退出，处于僵尸状态，哪一个进程先被处理，是不一定的。标准并未规定顺序。

wait()函数存在一定的局限：

* wait()函数不能等待特定的子进程。如果进程存在多个子进程，而只想获取某个子进程的退出状态，并不关心其他子进程的退出状态，此时wait()只能一一等待，查看返回值是否是关心的子进程。
* 如果不存在子进程退出，wait()只能阻塞，有些时候，仅仅是想尝试获取退出子进程的退出状态，如果不存在子进程退出，就立刻返回，不需要阻塞等待,类似与trywait的概念。wait()函数没有这个接口。
* wait()函数只能发现子进程的终止事件，如果子进程因某信号停止，或者停止的子进程收到SIGCONT信号恢复执行，这些事件wait()函数无法获知。换言之，wait能够探知子进程的死亡，但是不能探知子进程的昏迷(暂停)，也无法探知子进程从昏迷中苏醒（恢复执行）。

由于上述三个缺点的存在，所以引入了waitpid()函数。

5.7.3 等待子进程之waitpid()

waitpid()函数接口如下：

#include <sys/wait.h>

pid\_t waitpid(pid\_t pid, int \*status, int options);

先说和wait()函数相同的地方：

返回值的含义相同，都是终止子进程或者因信号停止或者因信号恢复执行的子进程的进程ID。

status的含义相同，都是用来记录子进程的相关事件，后面一节会详细介绍。

接下来介绍waitpid()函数专有的功能：

第一个参数是pid\_t类型，可以想见，waitpid肯定具备了精确打击的功能。waitpid函数可以指定等待哪一个子进程的退出（以及停止和恢复执行）。事实上，扩展不仅仅如此：

* pid>0:表示等待进程ID为pid的子进程，就是我们说的精确打击；
* pid=0:表示等待与调用进程同一个进程组的所有子进程；因为子进程可以设置自己进程组，所以子进程不一定和父进程归属同一个进程组。
* pid=-1:表示等待任意子进程，同wait类似。waitpid(-1,&status,0)与wait(&status)完全等价。
* pid< -1:等待所有子进程中，进程组ID与pid绝对值相等的子进程；

内核之中，wait()函数和waitpid函数，调用的都是wait4系统调用。wait4系统调用不仅可以支持waitpid这样复杂全面的需要，还会提供进程系统资源应用的信息，如前文提到的子进程的系统CPU事件，用户CPU事件，上下文切换次数等信息。只不过是wait和waitpid这两个函数都不要获取这些信息。

下面函数中间部分代码，就是根据pid的正负或者是否是0和-1来定义wait\_opts类型变量wo，后面会根据wo来控制到底关心那些进程的事件。

SYSCALL\_DEFINE4(wait4, pid\_t, upid, int \_\_user \*, stat\_addr,

int, options, struct rusage \_\_user \*, ru)

{

struct wait\_opts wo;

struct pid \*pid = NULL;

enum pid\_type type;

long ret;

if (options & ~(WNOHANG|WUNTRACED|WCONTINUED|

\_\_WNOTHREAD|\_\_WCLONE|\_\_WALL))

return -EINVAL;

if (upid == -1)

type = PIDTYPE\_MAX; /\*任意子进程\*/

else if (upid < 0) {

type = PIDTYPE\_PGID;

pid = find\_get\_pid(-upid);

} else if (upid == 0) {

type = PIDTYPE\_PGID;

pid = get\_task\_pid(current, PIDTYPE\_PGID);

} else /\* upid > 0 \*/ {

type = PIDTYPE\_PID;

pid = find\_get\_pid(upid);

}

wo.wo\_type = type;

wo.wo\_pid = pid;

wo.wo\_flags = options | WEXITED;

wo.wo\_info = NULL;

wo.wo\_stat = stat\_addr;

wo.wo\_rusage = ru;

ret = do\_wait(&wo);

put\_pid(pid);

/\* avoid REGPARM breakage on x86: \*/

asmlinkage\_protect(4, ret, upid, stat\_addr, options, ru);

return ret;

}

在内核的do\_wait函数中会根据wait\_opts类型的wo变量来控制到底在关心那些子进程的状态。

对于当前进程的线程组(thread group，注意不是进程组)的每一个线程(在内核实现，线程就是进程，每个线程都有独立的task\_struct)，都会遍历其子进程。在内核中，task\_struct中的children成员变量是个链表头，该进程的所有子进程都会链入该链表，遍历起来比较方便。

static int do\_wait\_thread(struct wait\_opts \*wo, struct task\_struct \*tsk)

{

struct task\_struct \*p;

list\_for\_each\_entry(p, &tsk->children, sibling) {

/\*遍历进程所有的子进程\*/

int ret = wait\_consider\_task(wo, 0, p);

if (ret)

return ret;

}

return 0;

}

但是我们知道，并不是所有子进程都是我们关心的。当调用wait()函数或者waitpid()函数第一个参数pid等于-1的时候，任意子进程我们都关心。但是waitpid()函数的其他情况不同，我们关心某些子进程或者某个子进程。内核对关心子进程的过滤在eligible\_pid函数中完成。

/\* 当waitpid的第一个参数为-1是，wo->wo\_type 赋值为PIDTYPE\_MAX

\* 其他三种情况，由task\_pid\_type(p, wo->wo\_type)== wo->wo\_pid检验

\* 或者检查PID是否相等，或者简称进程组ID是否等于指定值 \*/

static int eligible\_pid(struct wait\_opts \*wo, struct task\_struct \*p)

{

return wo->wo\_type == PIDTYPE\_MAX ||

task\_pid\_type(p, wo->wo\_type) == wo->wo\_pid;

}

第三个参数options，是标志位，可以同时存在多个标志。当options没有设置任何标志位时，行为和wait类似，阻塞等待与pid匹配的子进程退出。options的标志位可以是如下标志位的组合：

* WUNTRACE：除了返回终止子进程的信息，也关注因信号而停止的子进程信息。
* WCONTINUED：除了返回终止子进程的信息，也关心那些收到SIGCONT信号而回复执行的已停止的子进程的状态信息。
* WNOHANG：指定的子进程并未发生状态变化，立刻返回，不会阻塞。这种情况下返回值是0。如果调用进程并没有与pid匹配的子进程，那么返回-1，并且设置errno为ECHILD，根据返回值和errno可以区分这两种情况。

前面两个标志位，作用是扩大关注事件，传统的wait只关注子进程的终止，waitpid通过这两个标志位可以检测子进程的停止和从停止中恢复这两种事件。

讲到这里，需要解释一下什么是使进程停止，什么是使进程继续，以及为什么需要这个。设想如下的场景，假如机器上编译一个大型项目，编译需要消耗很多CPU资源和磁盘IO资源，如果我暂时需要用机器做其他事情，可能只需要几分钟，这几分钟内用户体验会非常糟糕，那怎么办？当然杀掉编译项目的进程是一个选择，但是这个方案并不友好。因为编译可能比较耗时，杀死进程，你可能不得不从头编译起。这时候，我们需要的仅仅是让编译大型项目的进程停下来，把CPU资源和磁盘IO资源让给我，让我从容做自己想做的事情，几分钟后，我用完了，在让编译的进程继续工作。

Linux提供了SIGSTOP(信号值19)和SIGCONT(信号值18)两个信号，来完成暂停和恢复的动作，可以通过执行kill -SIGSTOP或者kill -19来暂停一个进程的执行，通过执行kill -SIGCONT或者kill -18来让一个暂停的进程恢复执行。

waitpid()函数通过WUNTRACE标志位可以关注停止事件，如果有子进程收到信号处于暂停状态，waitpid可以返回。

同样道理，通过WCONTINUED标志位可以关注恢复执行事件，如果有子进程收到SIGCONT信号而恢复执行，waitpid也可以返回。

但是上述两种事件和子进程的终止事件是或的关系，waitpid成功返回的时候，可能是等到了子进程的终止事件，也可能是等待到了暂停或者恢复执行事件。这需要通过status的值来区分。现在是时候分析status的值了。

5.7.4 等待子进程之等待状态值

无论是wait()函数还是waitpid()函数，都有一个status变量。这个变量是个int型的指针。可以传递NULL，表示不关心子进程的状态信息。如果不为空，根据填充的status的值，可以获取到子进程的很多信息：



根据上图，直接根据status可以获得进程的退出方式，但是为了可移植性，不应该直接解析status获取退出状态。系统提供了相应的宏（macro），用来解析返回值。下面分别介绍各种情况。

1. 进程是正常退出

有两个宏与正常退出相关：

WIFEXITED(status)：如果子进程正常退出，返回true，否则返回false

WEXITSTATUS(status)：如果子进程是正常退出，则本宏用来获取退出状态。

#define \_\_WEXITSTATUS(status) (((status) & 0xff00) >> 8)

所谓取退出状态8~15位的值，也就是exit\_group系统调用用户传入的int型的值。当然只有最低的8位。

1. 进程收到信号，导致退出

有三个宏与这种情况相关

WIFSIGNALED(status)：如果进程时被信号杀死，返回true，否则返回false。

WTREMSIG(status)：如果进程是被信号杀死，返回杀死进程的信号的值。

WCOREDUMP(status)：如果子进程产生了core dump，返回true。否则返回false。

1. 进程收到信号，被停止。

有两个宏与这种情况相关

WIFSTOPPED(status)：子进程收到相关信号，暂停执行，处于停止状态，则返回true，否则返回fasle。

WSTOPSIG(status)：如果子进程处于停止状态，这个宏返回导致子进程停止的信号的值。

之所以需要这个宏来返回导致子进程停止的信号值，是因为不仅一个信号可以导致子进程停止：SIGSTOP，SIGTSTP，SIGTTIN，SIGTTOU，都会引起进程停止。

1. 子进程恢复执行

有一个宏与这种情况相关

WIFCONTINUED(status)：如果由于SIGCONT信号的递送，子进程恢复执行，则返回true，否则返回false。

为何没有返回信号值的宏？原因是只有SIGCONT能够使子进程从停止状态中恢复过来。如果子进程恢复执行，只能是收到了SIGCONT信号，所以不需要宏来取信号。

下面代码给出了判断子进程终止的示例代码。等待子进程暂停或者恢复执行的情况，可以根据下面示例，自行实现。

void print\_wait\_exit(int status)

{

printf("status = %d\n",status);

if(WIFEXITED(status))

{

printf("normal termination,exit status = %d\n",

WEXITSTATUS(status));

}

else if(WIFSIGNALED(status))

{

printf("abnormal termination,signal number =%d%s\n",

WTERMSIG(status),

#ifdef WCOREDUMP

WCOREDUMP(status)?"core file generated" : "");

#else

"");

#endif

}

}

尽管waitpid函数对wait函数做了很多的扩展，waitpid函数还是存在不足之处：

waitpid固然通过WUNTRACE和WCONTINUED标志位，增加了对子进程停止事件和子进程恢复执行事件的支持，但是这种支持并不完美，这两种事件都和子进程的终止事件混在一起。

wait和waitpid函数，都会调用wait4系统调用中，无论用户传递的参数为何，总会添上WEXITED事件。如果

wo.wo\_flags = options | WEXITED;

如果用户不关心子进程的终止事件，只关心子进程的停止事件，能否使用waitpid()明确做到。答案是不行。当waitpid返回时，可能是因为子进程终止，也可能是因为子进程停止。这是waitpid和wait的致命缺陷。

为了解决这个缺陷，wait家族的终极大boss，waitid()函数就要闪亮登场了。

5.6.5 等待子进程之waitid()

前面提到，waitpid函数是wait函数的超集，wait函数能干事情，waitpid函数都能做到。但是waitpid函数控制不太精确，用户无论是否关心相关子进程的终止事件，终止事件都可能会返回给用户。Linux提供了waitid系统调用。glibc封装了waitid系统调用实现了waitid函数。尽管目前普遍使用的是wait和waitpid两个函数，但是waitid函数的设计明显更加合理。

#include <sys/wait.h>

int waitid(idtype\_t idtype, id\_t id,siginfo\_t \*infop, int options);

用第一个入参idtype和第二个入参id用来选择关心的子进程。

* idtype == P\_PID：精确打击，等待进程ID等于id的进程。
* idtype == P\_PGID：在所有子进程中等待进程组ID等于id的进程。
* idtype == P\_ALL：等待任意子进程，第二个参数id被忽略。

这不是waitid的改进所在，waitid的改进在于第四个参数options。options参数是下面标志位的按位或：

* WEXITED：等待子进程的终止事件
* WSTOPPED：等待子进程的被信号暂停事件
* WCONTINUED：等待先前被暂停，但是被SIGCONT信号恢复执行的子进程

这三个标志位互相独立，就解决了waitpid的致命缺陷。

|  |  |
| --- | --- |
| **waitpid的标志位** | **等价的waitid的标志位** |
| WUNTRACED | WEXITED | WSTOPPED |
| WCONTINUCED | WEXITED | WCONTINUED |

WNOHANG：这个标志位是老相识了，语义和waitpid一致，与id匹配的子进程并无状态信息需要返回，则不阻塞，立刻返回，返回值是0。如果调用进程并无子进程与id匹配，返回-1，并且设置errno为ECHILD。

WNOWAIT：这个标志位是waitid独门绝招，waitpid和wait函数都不支持。通过前面讨论可以知道wait并不仅仅是获取子进程的状态信息，而且会改变子进程的状态。最典型的是子进程的退出。wait函数返回之前，子进程处于僵尸状态，取走信息之后，内核负责调用release\_task函数将僵尸子进程的最后残存资源释放，子进程彻底消失。WNOWAIT标志位指示内核，只负责获取信息，不要改变子进程的状态。带WNOWAIT标志位调用waitid函数，稍后还可以调用wait或者waitpid或者waitid再次获得同样信息。

第三个参数infop本质是返回值，系统调用负责将子进程的相关信息填充到infop指向的结构体中。如果成功取到信息，下面字段将会填充：

* si\_pid：子进程的进程ID，相当于wait和waitpid成功时的返回值
* si\_uid：子进程的真正用户ID
* si\_signo：该字段总被填成SIGCHLD
* si\_code：指示子进程发生的事件，该可能的取值是CLD\_EXIT(子进程正常退出)，CLD\_KILLED(子进程被信号杀死)，CLD\_DUMPED(子进程被信号杀死，并且产生了core dump)，CLD\_STOPPED(子进程被信号暂停)，CLD\_CONTINUED(子进程被SIGCONT信号恢复执行)，CLD\_TRAPPED(子进程被跟踪)。其中最后一个值主要用于ptrace机制的支持。
* si\_status：status的值的语义和wait函数及waitpid函数一致。

对于返回值，有两种情况会返回0：

1. 成功等待到子进程的变化，取回相应信息
2. 设置了WNOHANG标志位，并且子进程无状态变化

如果区分这两种情况？

解决的方法就是判断返回的siginfo\_t结构体中的si\_pid，如果子进程状态变化导致的返回，si\_pid不等于0，是子进程的进程ID，对于没有子进程状态有变化，si\_pid 等于0。但是标准并没有规定，waitid函数负责将siginfo\_t结构体的内容清零，所以为了正确区分两种情况，唯一安全的做法是首先将siginfo\_t结构体清零，返回后，判断si\_pid是否为0。

siginfo\_t info ;

memset(&info,0,sizeof(siginfo\_t));

if(waited(idtype,id,&info,options | WNOHANG) == -1)

{

/\*发生错误\*/

}

else if(info.si\_pid == 0)

{

/\*没有子进程发生变化\*/

}

else

{

/\*存在子进程状态发生变化，进一步处理之\*/

}

5.7.6 进程退出和等待的内核实现

Linux引入多线程之后，为了支持线程组所有线程整体退出，内核引入exit\_group系统调用。对于进程而言，无论是调用exit()函数，调用\_exit()函数还是在main函数中return，最终都会调用exit\_group系统调用，要线程组所有成员退出。对于Linux而言，用户态的线程，在内核中是进程，都会有独立的进程控制块task\_struct。

SYSCALL\_DEFINE1(exit\_group, int, error\_code)

{

do\_group\_exit((error\_code & 0xff) << 8);

/\* NOTREACHED \*/

return 0;

}

exit\_group系统调用，把所有的事情，委托给了do\_group\_exit()函数去做。不仅仅是exit\_group\_exit系统调用会用到这个函数，对于进程收到致命但是不产生内核转储文件的信号（如SIGKILL），但内核选择信号递送给进程时，也会调用do\_group\_exit。详情见get\_signal\_to\_deliver。也就是说通过kill -9来杀死进程，本质也会走到do\_group\_exit函数，让整个线程组退出。

kernel/signal.c get\_signal\_to\_deliver

do\_group\_exit(info->si\_signo);

如果此说来，do\_group\_exit函数就比较重要了。

NORET\_TYPE void

do\_group\_exit(int exit\_code)

{

struct signal\_struct \*sig = current->signal;

BUG\_ON(exit\_code & 0x80); /\* 会产生core dump的不会走到此处\*/

if (signal\_group\_exit(sig))

exit\_code = sig->group\_exit\_code;

else if (!thread\_group\_empty(current)) {

struct sighand\_struct \*const sighand = current->sighand;

spin\_lock\_irq(&sighand->siglock);

if (signal\_group\_exit(sig))

/\* Another thread got here before we took the lock. \*/

exit\_code = sig->group\_exit\_code;

else {

/\*第一个到达此处的线程，负责置位SIGNAL\_GROUP\_EXIT

\* 其他线程走到do\_group\_exit就直接走if 分支\*/

sig->group\_exit\_code = exit\_code;

sig->flags = SIGNAL\_GROUP\_EXIT;

zap\_other\_threads(current);

}

spin\_unlock\_irq(&sighand->siglock);

}

do\_exit(exit\_code);

/\* NOTREACHED \*/

}

后面多线程会讲到，对于线程组中的所有线程，会共用一个signal\_struct结构体，也就是代码中task->signal。线程组中的第一个走进do\_group\_exit的线程会走else分支，将共用的signal\_struct结构体置上SIGNAL\_GROUP\_EXIT标志位，第一个线程有责任通知其他线程，线程组的所有兄弟们，我们玩完了，请你们也退出。方法就在zap\_other\_threads函数中：

int zap\_other\_threads(struct task\_struct \*p)

{

struct task\_struct \*t = p;

int count = 0;

p->signal->group\_stop\_count = 0;

/\*遍历线程组的所有线程，除了已经在退出的，全部挂上SIGKILL信号

\* 当内核执行get\_signal\_to\_deliver，递送信号给其他线程时，

\* 发现下一个要递送的信号是SIGKILL时，会调用do\_group\_exit，

\* 对该线程而言，走if分支，直接调用do\_exit函数\*/

while\_each\_thread(p, t) {

task\_clear\_jobctl\_pending(t, JOBCTL\_PENDING\_MASK);

count++;

/\* Don't bother with already dead threads \*/

if (t->exit\_state)

continue;

sigaddset(&t->pending.signal, SIGKILL);

signal\_wake\_up(t, 1);

}

return count;

}

zap\_other\_threads负责给每一个线程挂上SIGKILL信号，后面内核调用get\_signal\_to\_deliver给其他线程递送信号时，发现了这个SIGKILL信号，就会调用do\_group\_exit。这样，线程组的其他线程也会走进do\_group\_exit。

无论是单线程的进程，还是多线程的进程，每一个线程都会走进do\_exit()函数来释放进程的资源。对于do\_exit当中，重点分析与wait相关的部分。

退出进程释放了几乎所有的资源，但是该进程并不甘心，它还有两桩心愿未了：

1. 该进程一个父进程，它可能有子进程，进程退出以后，谁来为它的子进程收尸
2. 该进程是一个子进程，它需要通知它的父进程来为自己收尸。

这部分代码在do\_exit函数的exit\_notify部分。

static void exit\_notify(struct task\_struct \*tsk, int group\_dead)

{

bool autoreap;

/\*forget\_original\_parent负责给自己子进程找个好人家\*/

forget\_original\_parent(tsk);

exit\_task\_namespaces(tsk);

write\_lock\_irq(&tasklist\_lock);

if (group\_dead)

kill\_orphaned\_pgrp(tsk->group\_leader, NULL);

if (unlikely(tsk->ptrace)) {

int sig = thread\_group\_leader(tsk) &&

thread\_group\_empty(tsk) &&

!ptrace\_reparented(tsk) ?

tsk->exit\_signal : SIGCHLD;

autoreap = do\_notify\_parent(tsk, sig);

} else if (thread\_group\_leader(tsk)) {

autoreap = thread\_group\_empty(tsk) &&

do\_notify\_parent(tsk, tsk->exit\_signal);

} else {

autoreap = true;

}

tsk->exit\_state = autoreap ? EXIT\_DEAD : EXIT\_ZOMBIE;

/\* mt-exec, de\_thread() is waiting for group leader \*/

if (unlikely(tsk->signal->notify\_count < 0))

wake\_up\_process(tsk->signal->group\_exit\_task);

write\_unlock\_irq(&tasklist\_lock);

/\* If the process is dead, release it - nobody will wait for it \*/

if (autoreap)

release\_task(tsk);

}

首先是调用forget\_original\_parent()函数来给自己的子进程找到新的父亲。原则是如果是多线程的程序，就把子进程托孤给同一线程组的其他线程，如果是单线程的程序，就托孤给1号进程即init进程。为子进程寻找新的父进程的工作就交给了find\_new\_reaper()函数。然后遍历自己所有的子进程，将子进程的父亲设为新的父亲。

static void forget\_original\_parent(struct task\_struct \*father)

{

struct task\_struct \*p, \*n, \*reaper;

LIST\_HEAD(dead\_children);

write\_lock\_irq(&tasklist\_lock);

/\*

\* Note that exit\_ptrace() and find\_new\_reaper() might

\* drop tasklist\_lock and reacquire it.

\*/

exit\_ptrace(father);

reaper = find\_new\_reaper(father);

list\_for\_each\_entry\_safe(p, n, &father->children, sibling) {

struct task\_struct \*t = p;

do {

t->real\_parent = reaper;

if (t->parent == father) {

BUG\_ON(t->ptrace);

t->parent = t->real\_parent;

}

/\*内核提供了机制，允许父进程退出时向子进程发送信号\*/

if (t->pdeath\_signal)

group\_send\_sig\_info(t->pdeath\_signal,

SEND\_SIG\_NOINFO, t);

} while\_each\_thread(p, t);

reparent\_leader(father, p, &dead\_children);

}

write\_unlock\_irq(&tasklist\_lock);

BUG\_ON(!list\_empty(&father->children));

list\_for\_each\_entry\_safe(p, n, &dead\_children, sibling) {

list\_del\_init(&p->sibling);

release\_task(p);

}

}

这部分代码比较容易引起困扰的是下面这行，我们都知道，子进程死的时候，会向父进程发送信号SIGCHLD，Linux也提供一种机制，父进程死的时候向子进程发送信号。

if (t->pdeath\_signal)

group\_send\_sig\_info(t->pdeath\_signal,

SEND\_SIG\_NOINFO, t);

读者可以man prctl，查看PR\_SET\_PDEATHSIG标志位部分。如果应用程序通过prctl设置了父进程死时，向子进程发送信号，就会执行到这部分内核代码。

接下来是第二桩未了的心愿：想办法通知父进程为自己收尸。对于单线程的程序比较简单，但是多线程的情况，就复杂了点。只有线程组的组长才有资格通知父进程，线程组其他线程终止的时候，不需要通知父进程，也没必要保留最后的资源，陷入僵尸态，直接调用release\_task函数释放所有资源就好。

为什么这样设计？细细想来，其实是合理的，父进程创建子进程时，只有子进程的主线程是父进程亲自创建出来的，是父进程的亲生儿子，父进程只关心子进程的主线程，至于子进程调用pthread\_create搞出了其他线程，父进程压根就不关心。

由于父进程只认子进程的主线程，所以线程组中，主线程一定要挺住。在用户层面，可以调用pthread\_exit让主线程先死，但是内核态，主线程的task\_struct一定要挺住，哪怕变成僵尸，也不许释放。

生命在于折腾，如果应用程序真的犯二，先将主线程退出了，其他线程还在工作，这种情况下，内核如何处理？

else if (thread\_group\_leader(tsk)) {

/\*线程组组长只有在全部线程都已退出情况下，

\*才能调用do\_notify\_parent通知父进程\*/

autoreap = thread\_group\_empty(tsk) &&

do\_notify\_parent(tsk, tsk->exit\_signal);

} else {

/\*如果是线程组的非组长线程，可以立即调用release\_task，

\*释放残余的资源，因为通知父进程这件事和它没关系\*/

autoreap = true;

}

上面代码给出答案，如果退出的进程是线程组的主线程，但是线程组中还有其他线程尚未终止（thread\_group\_empty函数返回false），那么autoreaper等于false，也不会调用do\_notify\_parent向父进程发送信号。在另一头，父进程如果调用wait，在整个线程组成员全部终止之前，父进程也不会收割自己的子进程，如下面wait\_consider\_task函数的代码所示：

#define delay\_group\_leader(p) \

(thread\_group\_leader(p) && !thread\_group\_empty(p))

/\* we don't reap group leaders with subthreads \*/

if (!delay\_group\_leader(p))

return wait\_task\_zombie(wo, p);

因为子进程的线程组中存在其他线程活着，子进程的主线程退出时不能通知父进程，错过了调用do\_notify\_parent的机会，那么父进程如何才能知晓子进程已经退出了呢？答案是，最后一个线程退出时。答案藏在内核的release\_task函数中：

leader = p->group\_leader;

if (leader != p && thread\_group\_empty(leader) && leader->exit\_state == EXIT\_ZOMBIE) {

/\*

\* If we were the last child thread and the leader has

\* exited already, and the leader's parent ignores SIGCHLD,

\* then we are the one who should release the leader.

\*/

zap\_leader = do\_notify\_parent(leader, leader->exit\_signal);

if (zap\_leader)

leader->exit\_state = EXIT\_DEAD;

}

当线程组的最后一个线程退出时，如果发现：

* 该线程不是线程组的主线程
* 线程组的主线程已经退出，处于僵尸状态
* 自己是最后一个线程

这三个条件满足的时候，该子进程就需要冒充线程组的组长，即子进程的主线程来通知父进程。注意上面代码片段中，do\_notify\_parent的第一个参数是leader，冒充线程组的组长来通知父进程。

上面讨论了一种比较折腾的场景，正常的多线程编程不应如此安排。多线程的进程，一般会主线程会等所有线程推出后，主线程才会退出。这种情况下，主线程会在exit\_notify函数中发现自己是组长，线程组里所有成员均已退出，这种情况下，主线程调用do\_notify\_parent来通知父进程。

无论怎样，子进程都走到了do\_notify\_parent函数。该函数是完成父子进程之间互动的主要函数。

bool do\_notify\_parent(struct task\_struct \*tsk, int sig)

{

struct siginfo info;

unsigned long flags;

struct sighand\_struct \*psig;

bool autoreap = false;

BUG\_ON(sig == -1);

/\* do\_notify\_parent\_cldstop should have been called instead. \*/

BUG\_ON(task\_is\_stopped\_or\_traced(tsk));

BUG\_ON(!tsk->ptrace &&

(tsk->group\_leader != tsk || !thread\_group\_empty(tsk)));

if (sig != SIGCHLD) {

/\*

\* This is only possible if parent == real\_parent.

\* Check if it has changed security domain.

\*/

if (tsk->parent\_exec\_id != tsk->parent->self\_exec\_id)

sig = SIGCHLD;

}

info.si\_signo = sig;

info.si\_errno = 0;

rcu\_read\_lock();

info.si\_pid = task\_pid\_nr\_ns(tsk, tsk->parent->nsproxy->pid\_ns);

info.si\_uid = \_\_task\_cred(tsk)->uid;

rcu\_read\_unlock();

info.si\_utime = cputime\_to\_clock\_t(cputime\_add(tsk->utime,

tsk->signal->utime));

info.si\_stime = cputime\_to\_clock\_t(cputime\_add(tsk->stime,

tsk->signal->stime));

info.si\_status = tsk->exit\_code & 0x7f;

if (tsk->exit\_code & 0x80)

info.si\_code = CLD\_DUMPED;

else if (tsk->exit\_code & 0x7f)

info.si\_code = CLD\_KILLED;

else {

info.si\_code = CLD\_EXITED;

info.si\_status = tsk->exit\_code >> 8;

}

psig = tsk->parent->sighand;

spin\_lock\_irqsave(&psig->siglock, flags);

if (!tsk->ptrace && sig == SIGCHLD &&

(psig->action[SIGCHLD-1].sa.sa\_handler == SIG\_IGN ||

(psig->action[SIGCHLD-1].sa.sa\_flags & SA\_NOCLDWAIT))) {

autoreap = true;

if (psig->action[SIGCHLD-1].sa.sa\_handler == SIG\_IGN)

sig = 0;

}

/\*子进程向父进程发送信号\*/

if (valid\_signal(sig) && sig)

\_\_group\_send\_sig\_info(sig, &info, tsk->parent);

/\* 子进程尝试唤醒父进程，如果父进程在等待其终止 \*/

\_\_wake\_up\_parent(tsk, tsk->parent);

spin\_unlock\_irqrestore(&psig->siglock, flags);

return autoreap;

}

父子进程之间的互动有两种方式

* 子进程向父进程发送信号SIGCHLD
* 子进程唤醒父进程。

对这两种方法，我们分别展开。

**父子进程互动之SIGCHLD信号**

首先是发送信号SIGCHLD。

默认情况下，父进程收到该信号，行为是置之不理。这种情况下，对于子进程来说，子进程陷入了僵尸状态，浪费了系统资源，该状态维持到父进程退出，子进程被init进程接管，init进程会等待僵尸进程，使僵尸进程释放资源。

如果父进程不太关心子进程的退出事件，听之任之不是好办法，可以采取以下办法：

* 父进程调用signal函数，或者sigaction函数，将SIGCHLD信号的处理函数设置为SIG\_IGN;
* 父进程调用sigaction函数，设置标志位时，置上SA\_NOCLDWAIT位（如果不关心子进程的暂停和恢复执行，置上SA\_NOCLDSTOP位）。

从上面内核代码看如果父进程的SIGCHLD的信号处理函数为SIG\_IGN或者sa\_flags中置上了SA\_NOCLDWAIT位，子进程运行到此处时就知道了，父进程并不关心自己的推出信息，do\_notify\_parent函数就会返回true，在外层的exit\_notify函数发现返回值是true，就会调用release\_task函数，释放残余的资源，自行了断，子进程也就不会进入僵尸状态。

默认情况下，子进程退出会向父进程发送信号，也给我们一个提示：如果父进程关心子进程的退出，父进程又不想调用wait，陷入可能的阻塞，那就可以安装SIGCHLD信号的处理函数来解决这个问题。

为SIGCHLD写信号处理函数并不简单，原因是SIGCHLD是传统的不可靠信号。信号处理函数执行期间，会将引发调用的信号暂时阻塞（除非显式指定了SA\_NODEFER标志位），在这期间收到的SIGCHLD之类的传统信号，都不会排队。因此，如果处理SIGCHLD信号时，多个子进程退出，产生了多个SIGCHLD信号，父进程只能收到一个。如果在信号处理函数中，只调用一次wait或者waitpid，会造成某些僵尸进程成为漏网之鱼。

正确的写法是，信号处理函数内，带着NOHANG标志位循环调用waitpid。如果返回值大于0，表示不断等到子进程退出，返回0表示，当前没有僵尸子进程，返回-1表示出错，最大的可能就是errno等于ECHLD，所有子进程都已退出。

while(waitpid(-1,&status,WNOHANG) > 0)

{

/\*此处处理返回信息\*/

continue;

}

信号处理函数中的waitpid可能会失败，从而改变全局的errno值，当主程序会检查errno时，就有可能冲突，所以进入信号处理函数前要现保存errno到本地变量，信号处理函数退出前，再把errno恢复。

**父子进程互动之等待队列**

前面提到了，子进程会调用\_\_wake\_up\_parent，来及时唤醒父进程。事实上，前提条件是父进程确实在等待子进程退出。如果父进程并没调用wait系列函数等待子进程的退出，那么，等待队列为空，子进程的\_\_wake\_up\_parent对父进程并无任何影响。

void \_\_wake\_up\_parent(struct task\_struct \*p, struct task\_struct \*parent)

{

\_\_wake\_up\_sync\_key(&parent->signal->wait\_chldexit,

TASK\_INTERRUPTIBLE, 1, p);

}

父进程的进程描述符的signal结构体中有wait\_childexit变量，这个变量是等待队列头。父进程调用wait系列函数时，都会创建一个wait\_opts结构体，把该结构体挂入等待队列。

static long do\_wait(struct wait\_opts \*wo)

{

struct task\_struct \*tsk;

int retval;

trace\_sched\_process\_wait(wo->wo\_pid);

/\*挂入等待队列\*/

init\_waitqueue\_func\_entry(&wo->child\_wait, child\_wait\_callback);

wo->child\_wait.private = current;

add\_wait\_queue(&current->signal->wait\_chldexit, &wo->child\_wait);

repeat:

/\*\*/

wo->notask\_error = -ECHILD;

if ((wo->wo\_type < PIDTYPE\_MAX) &&

(!wo->wo\_pid || hlist\_empty(&wo->wo\_pid->tasks[wo->wo\_type])))

goto notask;

set\_current\_state(TASK\_INTERRUPTIBLE);

read\_lock(&tasklist\_lock);

tsk = current;

do {

retval = do\_wait\_thread(wo, tsk);

if (retval)

goto end;

retval = ptrace\_do\_wait(wo, tsk);

if (retval)

goto end;

if (wo->wo\_flags & \_\_WNOTHREAD)

break;

} while\_each\_thread(current, tsk);

read\_unlock(&tasklist\_lock);

/\*找了一圈，没有找满足等待条件的的子进程，取决于WNOHANG标志位

\*如果标志位置位，不等了，直接退出，

\*如果没有置位，让出CPU，醒来后继续再找一圈\*/

notask:

retval = wo->notask\_error;

if (!retval && !(wo->wo\_flags & WNOHANG)) {

retval = -ERESTARTSYS;

if (!signal\_pending(current)) {

schedule();

goto repeat;

}

}

end:

\_\_set\_current\_state(TASK\_RUNNING);

remove\_wait\_queue(&current->signal->wait\_chldexit, &wo->child\_wait);

return retval;

}

父进程先把自己置成TASK\_INTERRUPT状态，然后开始寻找满足等待条件的子进程。找到了，将自己重置位TASK\_RUNNING状态，欢乐返回，但是如果没找到，就要根据WNOHANG标志位来决定等不等子进程。如果没有WNOHANG标志位，那么，父进程就会让出CPU资源，等待别人讲它唤醒。

回到另一头，子进程退出的时候，会调用\_\_wake\_up\_parent，唤醒父进程，父进程醒来以后，回到repeat，再次扫描。这样，子进程退出就能及时的通知到父进程，从而是父进程的wait系列函数可以及时的返回。

## 5.8 exec家族

前面讨论了进程的创建和退出，exec家族函数了在前面章节，若隐若现，现在是时候让exec家族函数亮相了。

整个exec家族有六个函数，这些函数的根基都是内核的execve系统调用。这个系统调用的作用是，将新程序加载到进程的地址空间，丢弃旧有的程序，进程的栈，数据段，堆栈会被新程序替换。

基于execve系统调用，glibc提供了一系列库函数。接口虽然各异，实现的功能却是相同，首先讲述和系统调用同名的execve函数。

### 5.8.1 execve函数

#include <unistd.h>

int execve(const char \*filename, char \*const argv[],

char \*const envp[]);

参数filename是准备执行的新程序的路径名，可以是绝对路径，也可以是相对于进程当前工作目录的相对路径。

后面的第二个参数，让我们想起C语言的main()函数的第二个参数，事实上格式也是一样的：以字符串指针组成的数组，以NULL结束。argv[0]一般对应可执行文件的文件名，也就是filename中的basename（路径名最后一个/后面的部分）。当然如果argv[0]不遵循这个约定也无妨，因为execve可以从第一个参数获取到要执行文件的路径，只要不是NULL即可。

第三个参数和C语言的main函数中第三个参数envp一样，也是字符串指针数组，以NULL结束，指针指向的字符串的格式为name=value。

一般来说，execve()函数总是和fork()函数伴随的，父进程调用fork之后，子进程执行execve函数，抛弃父进程的程序段，和父进程分道扬镳，从此各走各路。但是也可以不执行fork，单独调用execve函数。

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

int main(void)

{

char \*args[] = {"/bin/ls", "-l",NULL};

if(execve("/bin/ls",args, NULL) == -1) {

perror("execve");

exit(EXIT\_FAILURE);

}

puts("Never get here");

exit(EXIT\_SUCCESS);

}

本着贵在折腾的原则，写了一个不fork，直接调用execve的程序。调用execve后，程序就变成了/bin/sh -l。这个程序的输出如下：

total 16

-rwxr-xr-x 1 root root 8672 Dec 27 20:40 exec\_no\_fork

-rw-r--r-- 1 root root 288 Dec 27 20:40 exec\_no\_fork.c

我们看到，代码最后的Never get here没有被打印出来，这是因为execve函数的返回是特殊的。如果失败，会返回-1，但是如果成功，永不返回。这个是可以理解的。execve做的就是斩断过去，奔向新生活的勾当，如果成功，自然不可能返回回来，再次执行老程序的代码。

所以无需检查execve的返回值，只要返回，必然是-1。可以从errno判断出出错的原因。出错的可能性非常多，手册提供了19种不同的errno，罗列了22种失败的情景。很难记住，好在大部分不常见，常见的情况有以下几种：

* EACCESS：这个是我们最容易想到的，就是第一个参数filename，不是个普通文件，
* 或者该文件没有赋予可执行权限，或者目录结构中某一级目录不可搜索，或者文件所在文件系统是以MS\_NOEXEC标志挂载的。
* ENOENT：文件不存在。
* ETXTBSY：存在其他进程尝试修改filename所指代的文件。
* ENOEXEC：这个错误其实是比较高端的一种错误了，文件存在，也可以执行，但是无法执行，比如说，Window下的可执行程序，拿到Linux下，调用execve来执行，文件的格式不对，就会返回这种错误。

上面提到的ENOEXEC错误码，其实已经触及了execve函数的核心，即那些文件是可以执行的，execve系统调用又是如何执行的？会在execve系统调用的内核系统调用介绍。

### 5.8.2 exec家族

从内核的角度来说，提供execve系统调用就足够了，但是从应用层编程来讲，execve函数并不那么好使：

* 第一个参数必须是绝对路径或者是相对于当前工作目录的相对路径。习惯在shell下工作的用户就会觉得不太方便，因为日常工作都ls，mkdir，没有人会写/bin/ls 或者/bin/mkdir。Shell提供了环境变量PATH，可执行程序的查找路径，对于位于查找路径里的可执行程序，我们不必写出完整路径，方便了我们使用，execv函数享受不到这个便利，使用不便
* execve函数的第三个参数值环境变量指针数组，用户编程是不得不自己负责环境变量，书写大量的“key=value”，大部分情况下不需要定制环境变量，只需要使用当前的环境变量。

正是为了提供相应的便利，所以用户层提供了6个函数，当然，这些函数本质都是调用execve系统调用，但是使用方法略有不同。

#include <unistd.h>

extern char \*\*environ;

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[]);

上述6个函数，分成上下两个半区。分类的依据是参数采用列表，还是数组。上半区采用列表，会罗列所有的参数，下半区采用数组。

每个半区之中，带p的表示可以使用环境变量PATH，带e的表示必须要自己维护环境变量，不适用当前环境变量。

exec家族函数

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名 | 参数格式 | 是否自动搜索PATH | 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，需自己组装环境变量 |
| execv | 数组 | 不是 | 是 |
| execvp | 数组 | 是 | 是 |
| execve | 数组 | 不是 | 不是，需自己组装环境变量 |

举个例子加深记忆：

#include <unistd.h>

char \*const ps\_argv[] = {"ps","-ax",NULL};

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

execl("/bin/ps","ps","-ax",NULL);

/\*带p的，可以使用环境变量PATH，无需写全路径\*/

execlp("ps","ps","-ax",NULL);

/\*带e的需要自己组拼环境变量\*/

execle("/bin/ps","ps","-ax",NULL,ps\_envp);

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

/\*带p的，可以使用环境变量PATH，无需写全路径\*/

execvp("ps",ps\_argv);

/\*带e的需要自己组拼环境变量\*/

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

### 5.8.3 execve系统调用的内核实现

上面提到ENOEXEC错误表示内核不知道如何执行对应的可以执行文件。Linux支持很多中可执行文件的格式，有渐渐退出历史舞台的aout格式，有比较通用的ELF格式的文件，有shell脚本文件，python脚本，java文件，php文件等。这些形形色色的可执行文件，内核如何正确地执行？直接将windows平台上的可执行文件拷贝到Linux下，Linux为什么不能执行（假设没有wine这个执行windows程序的工具）。这是本节需要解决的问题。要解决上述的问题，我们需要深入内核。

execve是平台相关的系统调用，除去我们不太关心的平台差异，内核都会走到do\_execve函数。

static int do\_execve\_common(const char \*filename,

struct user\_arg\_ptr argv,

struct user\_arg\_ptr envp,

struct pt\_regs \*regs)

{

struct linux\_binprm \*bprm;

struct file \*file;

struct files\_struct \*displaced;

bool clear\_in\_exec;

int retval;

const struct cred \*cred = current\_cred();

/\*

\* We move the actual failure in case of RLIMIT\_NPROC excess from

\* set\*uid() to execve() because too many poorly written programs

\* don't check setuid() return code. Here we additionally recheck

\* whether NPROC limit is still exceeded.

\*/

if ((current->flags & PF\_NPROC\_EXCEEDED) &&

atomic\_read(&cred->user->processes) > rlimit(RLIMIT\_NPROC)) {

retval = -EAGAIN;

goto out\_ret;

}

/\* We're below the limit (still or again), so we don't want to make

\* further execve() calls fail. \*/

current->flags &= ~PF\_NPROC\_EXCEEDED;

retval = unshare\_files(&displaced);

if (retval)

goto out\_ret;

retval = -ENOMEM;

bprm = kzalloc(sizeof(\*bprm), GFP\_KERNEL);

if (!bprm)

goto out\_files;

retval = prepare\_bprm\_creds(bprm);

if (retval)

goto out\_free;

retval = check\_unsafe\_exec(bprm);

if (retval < 0)

goto out\_free;

clear\_in\_exec = retval;

current->in\_execve = 1;

file = open\_exec(filename);

retval = PTR\_ERR(file);

if (IS\_ERR(file))

goto out\_unmark;

sched\_exec();

bprm->file = file;

bprm->filename = filename;

bprm->interp = filename;

retval = bprm\_mm\_init(bprm);

if (retval)

goto out\_file;

bprm->argc = count(argv, MAX\_ARG\_STRINGS);

if ((retval = bprm->argc) < 0)

goto out;

bprm->envc = count(envp, MAX\_ARG\_STRINGS);

if ((retval = bprm->envc) < 0)

goto out;

retval = prepare\_binprm(bprm);

if (retval < 0)

goto out;

retval = copy\_strings\_kernel(1, &bprm->filename, bprm);

if (retval < 0)

goto out;

bprm->exec = bprm->p;

retval = copy\_strings(bprm->envc, envp, bprm);

if (retval < 0)

goto out;

retval = copy\_strings(bprm->argc, argv, bprm);

if (retval < 0)

goto out;

retval = search\_binary\_handler(bprm,regs);

if (retval < 0)

goto out;

/\* execve succeeded \*/

current->fs->in\_exec = 0;

current->in\_execve = 0;

acct\_update\_integrals(current);

free\_bprm(bprm);

if (displaced)

put\_files\_struct(displaced);

return retval;

out:

if (bprm->mm) {

acct\_arg\_size(bprm, 0);

mmput(bprm->mm);

}

out\_file:

if (bprm->file) {

allow\_write\_access(bprm->file);

fput(bprm->file);

}

out\_unmark:

if (clear\_in\_exec)

current->fs->in\_exec = 0;

current->in\_execve = 0;

out\_free:

free\_bprm(bprm);

out\_files:

if (displaced)

reset\_files\_struct(displaced);

out\_ret:

return retval;

}

linux\_binprm是重要的结构体，它和稍候提到的linux\_binfmt联手，支持了Linux下多种可执行文件的格式。首先将程序运行需要的参数argv和环境变量搜集到linux\_binprm结构体中，比较关键的一步是:

retval = prepare\_binprm(bprm);

在prepare\_binprm函数中会读取可执行文件的头128个字节，存放在linux\_binprm结构体的buf[BINPRM\_BUF\_SIZE]中。我们知道日常写shell脚本，python脚本的时候，总是在第一行写如下语句：

#！/bin/bash

#! /usr/bin/python

#！/usr/bin/env python

开头的#！被称为shebang，又称为sha-bang，hashbang等，指的就是脚本中开始的字符。在类Unix操作系统中，运行这种程序，需要相应的解释器。使用哪种解释器，取决于shebang后面的路径。#！后面跟随的一般是解释器的绝对路径或者是相对于当前工作目录的相对路径。格式如下所示：

#! interpreter [optional-arg]

解释器是绝对路径或者是相对于当前工作目录的相对路径，这就给脚本的可移植性带来的挑战。以python的解释器为例，python可能位于/usr/bin/python，也可能位于/usr/local/bin/python，甚至有的位于/home/username/bin/python。这样编写的脚本在新的环境里面运行，用户不得不修改脚本，当大量脚本移植到新环境运行时，修改量是巨大的。为了解决这个问题，引入了如下格式：

#！/usr/bin/env python

这种格式，执行时，会从环境变量$PATH中查找python解释器。如果存在多个版本的解释器，会按照$PATH中查找路径的顺序来。

➜ hello echo $PATH

/home/manu/bin:/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games

如果执行方式是./python\_script的方式，就会优先查找/home/manu/bin/python，/usr/local/bin/python次之……，如下所示

execve("/home/manu/bin/python", ["python", "./hello.py"], [/\* 25 vars \*/]) = -1 ENOENT (No such file or directory)

execve("/usr/local/bin/python", ["python", "./hello.py"], [/\* 25 vars \*/]) = -1 ENOENT (No such file or directory)

execve("/usr/local/sbin/python", ["python", "./hello.py"], [/\* 25 vars \*/]) = -1 ENOENT (No such file or directory)

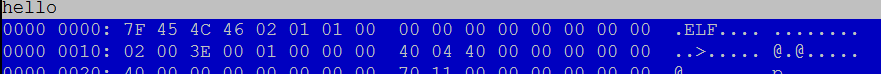
execve("/usr/local/bin/python", ["python", "./hello.py"], [/\* 25 vars \*/]) = -1 ENOENT (No such file or directory)

execve("/usr/sbin/python", ["python", "./hello.py"], [/\* 25 vars \*/]) = -1 ENOENT (No such file or directory)

execve("/usr/bin/python", ["python", "./hello.py"], [/\* 25 vars \*/]) = 0

上面提到的是脚本文件，除此外，还有其他格式的文件。有的目标文件是ELF格式，还有较早出现的a.out格式，这些文件的特点最初的128个字节中都包含了可执行文件的属性的重要信息。如下图ELF格式的可执行文件，开头4个字节为7F 45(E) 4C(L) 46(F)。将文件的开始128字节存入linux\_binprm，为了后面根据文件开头的magic number选择正确的处理方式。

➜ hello file hello

hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.24, BuildID[sha1]=657d5ef3eab6741481bb219ef6c2fb21f8e91b51, not stripped

做完准备工作后，开始执行，核心的代码位于search\_binary\_handler()函数。内核之中存在一个全局链表，名叫formats，挂到此链表的数据结构为struct linux\_binfmt:

struct linux\_binfmt {

struct list\_head lh;

struct module \*module;

int (\*load\_binary)(struct linux\_binprm \*, struct pt\_regs \* regs);

int (\*load\_shlib)(struct file \*);

int (\*core\_dump)(struct coredump\_params \*cprm);

unsigned long min\_coredump; /\* minimal dump size \*/

};

操作系统启动的时候，每个编译进内核的可执行文件的“代理人”会调用register\_binfmt函数注册，把自己挂到formats链表中。每个成员代表一种可执行文件的代理人，前面提到，会将可执行文件的头128个字节存放到linux\_binprm的buf中，同时会将运行时的参数和环境变量也存放linux\_binprm的相关结构中。formats链表中的成员依次前来认领，如果是自己代表的可执行文件的格式，后面执行的事情，就委托给了该“代理人”。

如果遍历了链表，所有的linux\_binfmt都表示不认识该可执行文件，又当如何？这种情况下根据头部的信息，查看是否有为该格式设计的，作为可动态安装的模块实现的“代理人”存在，如果有的话，把该模块安装进来挂入全局的formats链表之中，然后让formats链表中的所有成员再试一次。

上述逻辑位于search\_binary\_handler函数之中：

int search\_binary\_handler(struct linux\_binprm \*bprm,struct pt\_regs \*regs)

{

unsigned int depth = bprm->recursion\_depth;

int try,retval;

struct linux\_binfmt \*fmt;

pid\_t old\_pid;

/\* This allows 4 levels of binfmt rewrites before failing hard. \*/

if (depth > 5)

return -ELOOP;

retval = security\_bprm\_check(bprm);

if (retval)

return retval;

retval = audit\_bprm(bprm);

if (retval)

return retval;

/\* Need to fetch pid before load\_binary changes it \*/

rcu\_read\_lock();

old\_pid = task\_pid\_nr\_ns(current, task\_active\_pid\_ns(current->parent));

rcu\_read\_unlock();

retval = -ENOENT;

/\*最多尝试两次，第一次遍历formats链表中的所有成员，

\*若没找到，尝试加载动态模块，再次遍历\*/

for (try=0; try<2; try++) {

read\_lock(&binfmt\_lock);

list\_for\_each\_entry(fmt, &formats, lh) {

int (\*fn)(struct linux\_binprm \*, struct pt\_regs \*) = fmt->load\_binary;

if (!fn)

continue;

if (!try\_module\_get(fmt->module))

continue;

read\_unlock(&binfmt\_lock);

bprm->recursion\_depth = depth + 1;

retval = fn(bprm, regs);

bprm->recursion\_depth = depth;

if (retval >= 0) {

if (depth == 0)

ptrace\_event(PTRACE\_EVENT\_EXEC,

old\_pid);

put\_binfmt(fmt);

allow\_write\_access(bprm->file);

if (bprm->file)

fput(bprm->file);

bprm->file = NULL;

current->did\_exec = 1;

proc\_exec\_connector(current);

return retval;

}

read\_lock(&binfmt\_lock);

put\_binfmt(fmt);

if (retval != -ENOEXEC || bprm->mm == NULL)

break;

if (!bprm->file) {

read\_unlock(&binfmt\_lock);

return retval;

}

}

read\_unlock(&binfmt\_lock);

#ifdef CONFIG\_MODULES

if (retval != -ENOEXEC || bprm->mm == NULL) {

break;

} else {

#define printable(c) (((c)=='\t') || ((c)=='\n') || (0x20<=(c) && (c)<=0x7e))

if (printable(bprm->buf[0]) &&

printable(bprm->buf[1]) &&

printable(bprm->buf[2]) &&

printable(bprm->buf[3]))

break; /\* -ENOEXEC \*/

if (try)

break; /\* -ENOEXEC \*/

request\_module("binfmt-%04x", \*(unsigned short \*)(&bprm->buf[2]));

}

#else

break;

#endif

}

return retval;

}

是时候看下Linux当前支持的可执行文件的格式了。在Linux源码fs/Makefile，提到了以下编译选项

obj-$(CONFIG\_COMPAT) += compat.o compat\_ioctl.o

obj-$(CONFIG\_BINFMT\_AOUT) += binfmt\_aout.o

obj-$(CONFIG\_BINFMT\_EM86) += binfmt\_em86.o

obj-$(CONFIG\_BINFMT\_MISC) += binfmt\_misc.o

# binfmt\_script is always there

obj-y += binfmt\_script.o

obj-$(CONFIG\_BINFMT\_ELF) += binfmt\_elf.o

obj-$(CONFIG\_COMPAT\_BINFMT\_ELF) += compat\_binfmt\_elf.o

obj-$(CONFIG\_BINFMT\_ELF\_FDPIC) += binfmt\_elf\_fdpic.o

obj-$(CONFIG\_BINFMT\_SOM) += binfmt\_som.o

obj-$(CONFIG\_BINFMT\_FLAT) += binfmt\_flat.o

我们可以通过下面方式查看自己机器的编译选项，从而得知支持的可执行文件类型

➜ /boot grep BINFMT /boot/config-3.13.0-43-generic

CONFIG\_BINFMT\_ELF=y

CONFIG\_COMPAT\_BINFMT\_ELF=y

CONFIG\_ARCH\_BINFMT\_ELF\_RANDOMIZE\_PIE=y

CONFIG\_BINFMT\_SCRIPT=y

CONFIG\_BINFMT\_MISC=m

binfmt\_aout.c是对应a.out类型的可执行文件，这种文件格式是早期UNIX系统使用的可执行文件的格式，由AT&T设计，今天已经退出了历史舞台。

binfmt\_elf.c对应的是ELF格式的可执行文件。ELF最早由UNIX系统实验室(UNIX SYSTEM Laboratories USL)开发，目的是取代传统的a.out格式。1994年6月ELF格式出现在LINUX系统上，目前，ELF格式已经成为Linux下最主要的可执行文件格式。

binfmt\_script对应的是script格式的可执行文件，这种格式的可执行文件一般以“#！”开头，查找相应的解释器来执行脚本。比如python脚本，比如shell脚本，比如perl脚本。

早期的内核之中，曾经为Java格式提供了专门的binfmt结构，后来取消了，原因Java并不特殊，不值得为其提供专门的binfmt结构。如果为Java提供了，其他语言就会有意见了，没有做到一视同仁。但是需要支持的可执行文件的格式越来越多，大家都可能有自己的解释器，内核支持不可能无限制增加binfmt结构，这时候，binfmt\_misc就出现了。binfmt把这个功能开放给了用户层，用户可以引入自己的可执行文件格式，只要你能定义好magic number，识别出文件是否是自己的这种格式，另外自己定义好解释器，就可以了。

binfmt\_misc这个机制非常好，提供了支持额外可执行格式的可扩展方法。举例来讲，如果想在Linux下执行Windows的.exe文件，Wine软件可以在Linux下执行Windows的exe文件。

wine application.exe

我们可以将Windows exe文件注册到binfmt\_misc，直接使用如下方法执行exe文件：

./application.exe

方法就是：

echo ':Wine:M::MZ::/usr/bin/wine:' > /proc/sys/fs/binfmt\_misc/register

如果/proc/sys/fs/binfmt\_misc目录并不存在，表明binfmt\_misc并没挂载，那么需要

mount -t binfmt\_misc binfmt\_misc /proc/sys/fs/binfmt\_misc

或者在/etc/fstab中添加如下行:

binfmt\_misc /proc/sys/fs/binfmt\_misc binfmt\_misc defaults 0 0

echo的各个字段是什么含义呢。注册某种可执行文件到binfmt\_misc的格式如下：

:Name:Type:Offset:String:Mask:Interpreter:Flags

各个字段的含义是：

* Name：产生在/proc/sys/fs/binfmt\_misc目录下的文件名，代表一种可执行文件
* Type：表示识别类型，M表示用魔数来识别，E表示扩展。
* Offset：魔数在文件中的起始偏移量。
* String：以魔数或者以扩展名匹配的字符串
* Mask：用来屏蔽掉String中的一些位的字符串
* Interpret：解释程序的完整路径名
* Flags：可选标志，控制必须怎样调用解释程序。

根据这个解释，我们echo的语句的含义是：Windows可执行文件的前两个字节是魔数MZ，由解释程序/usr/bin/wine执行这个可执行文件。

表面看，很多种类型的文件，但是最终都会归结到ELF格式或者a.out格式，这是因为，那些脚本的解释器，是ELF格式。限于篇幅，就不介绍内核如何加载执行ELF格式的可执行程序了。毛德操老师的Linux内核情景分析里面详细分析了a.out类型的可执行文件的加载执行，可以供我们参考学习。

### 5.8.4 exec与信号

exec系列函数，都是属于天生反骨仔，他会将现有进程的所有文本段抛弃，奔向新生活。叛逃之前，进程可能执行过signal或者sigaction，为某些信号注册了新的信号处理函数。一旦决裂，这些新的信号处理函数就无处可寻了。所以内核会为那些曾经改变信号处理函数的信号负责，将它们的处理函数重新设为SIG\_DFL。

这里面有一个特例，就是将处理函数设置为忽略（SIG\_IGN）的SIGCHLD信号。调用exec之后，是SIGCHLD的信号处理函数是保持为SIG\_IGN还是重置成SIG\_DFL，SUSv3语焉不详，取决于操作系统。对于Linux系统而言，采用前者：保持为SIG\_IGN。

## 5.9 system函数

前面提到了fork函数，提到了exec系列函数，提到了wait系列函数。库将这些接口糅合在一起，提供了一个system函数。程序可以通过调用system函数，来执行任意的shell命令。相信很多程序员都用过system函数，因为起到一个粘合剂的作用，可以很方便让C程序调用其他语言编写的程序。但是相信有很多程序员被system函数折磨过，当出现错误时，如何根system据函数的返回值，定位失败的原因是个比较挠头的问题。下面我们细细展开。

### 5.9.1 system函数接口

#include <stdlib.h>

int system(const char \*command);

将需要执行的命令作为command参数，传给system函数，该函数就帮你执行该命令。这样看来system最大的好处就在于方便。不需要自己来调用fork，exec，waitpid，也不需要自己处理错误，处理信号，方便省心。

但是system函数的缺点也是很明显的。首先是效率，使用system运行命令，至少要创建两个进程，一个shell进程，另外一个或者多个用于shell所执行的命令。如果对效率要求比较高，最好是自己直接调用fork和exec来执行既定的程序。

从进程的角度来看，调用system的函数，会首先创建一个子进程shell，然后shell会创建子进程来执行command，如下图所示：



调用system函数，命令是否运行成功是我们最关心的事情。但是system的返回值比较复杂，下面通过一个简化的不完备（没有处理信号）的system实现来讲述system函数的返回值。

#include<unistd.h>

#include<sys/wait.h>

#include<sys/types.h>

int system(char\* command)

{

int status ;

pid\_t child;

switch(child = fork())

{

case -1:

return -1;

case 0:

execl("/bin/sh),"sh","-c",command,NULL);

\_exit(127);

default:

while(waitpid(child,&status,0) < 0)

{

/\*如果系统调用被中断,重启系统调用\*/

if(errno != EINTR)

{

status = -1;

break

}

}

else

return status;

}

}

下面我们分别讲述system函数的返回值：

* 当command 为NULL时，返回0或者1

正常情况下，没有人会这样用system。但是command为NULL是有用的，用户可以通过调用system(NULL)来探测shell是否可用。如果shell存在并且可用，那么返回1，如果系统里面压根就没有shell，这种情况下，shell不可用，返回0。何种情况下shell不可用呢？比如system函数运行在非UNIX系统上，再比如程序调用system之前，执行过了chroot，这些情况下shell可能无法使用。

command为NULL的情况，从简化版code中看不出来，但是glibc的system函数源码中可以看出端倪：

*glibc-2.17/sysdeps/posix/system.c*

*----------------------------------*

int

\_\_libc\_system (const char \*line)

{

if (line == NULL)

/\* Check that we have a command processor available. It might

not be available after a chroot(), for example. \*/

**return do\_system ("exit 0") == 0;**

if (SINGLE\_THREAD\_P)

return do\_system (line);

int oldtype = LIBC\_CANCEL\_ASYNC ();

int result = do\_system (line);

LIBC\_CANCEL\_RESET (oldtype);

return result;

}

weak\_alias (\_\_libc\_system, system)

* 创建进程(fork)失败，或者获取子进程终止状态(waitpid)失败，返回-1

创建进程失败，比较少见，比较容易想到的也就是创建了太多的进程，超出了系统限制。但是等待子进程终止状态失败，这个失败是比较容易造出来的。

前面讲过，子进程退出的时候，如果SIGCHLD的信号处理函数是SIG\_IGN或者用户设置了SA\_NOCLDWAIT标志位，那子进程就不进入僵尸状态等待父进程wait了，直接自行了断，灰飞烟灭。但是system函数内部实现会调用waitpid获取子进程的退出状态。这就是父子之前没有协调好，总会返回ECHLD这种错误。这种情况下，system返回-1，errno为ECHLD。

这种错误的示范代码如下所示：

signal(SIGCHLD,SIG\_IGN);/\*返回-1的根源在于此处\*/

if((status = system(command) )<0)

{

fprintf(stderr,"system return %d (%s)\n",

status,strerror(errno));

return -2;

}

这种情况下，总是返回-1，错误码是ECHLD，如下所示

➜ ./t\_sys\_err "ls"

system\_return.c t\_sys t\_sys\_err t\_sys\_null t\_system.c t\_system\_null.c

system return -1 (No child processes)

所以需要调用system函数的时候，需要确认，SIGCHLD是否是SIG\_IGN。如果是，需知，system会返回-1。

* 如果子进程不能执行shell，那么system返回值会与\_exit(127)终止时一样。

case 0:

execl("/bin/sh),"sh","-c",command,NULL);

\_exit(127);

如上面代码所示，如果执行execl失败，就会执行到\_exit(127)，否则不会执行到\_exit(127)。

* 如果所有的系统调用都执行成功，system函数会返回执行command的子shell的终止状态。

因为shell的终止状态是其执行最后一条命令的退出状态。这种情况下就和获取子进程的退出状态一样了。我们在前文详细提到过，可以下面的接口来判断：

WIFEXITED(status)

WEXITSTATUS(status)

WIFSIGNALED(status)

WTERMSIG(status)

WCOREDUMP(status)

综上所述，在command不等于NULL的情况下，正确判断system返回值的方法如下：

if((status = system(command) ) == -1)

{

fprintf(stderr,"system() function return -1 (%s)\n",

strerror(errno));

}

else if(WIFEXITED(status) && WEXITSTATUS(status) == 127)

{

fprintf(stderr,"cannot invoke shell to exec command(%s)\n",command);

}

else

print\_wait\_exit(status);

其中print\_wait\_exit函数中就是前文介绍的通过宏来判断进程的终止状态。

用上面的方法我们可以测试下。下面t\_sys是我用C写的一个工具，该工具执行需要1个参数，argv[1]是接受要执行的command，用上面提到的方法判断command的执行情况：

➜ ./t\_sys "ls"

system\_return.c t\_sys t\_sys\_err t\_sys\_null t\_system.c t\_system\_null.c

status = 0

normal termination,exit status = 0

➜ ./t\_sys "sleep 100" /\*另一终端向sleep 100进程发送SIGINT信号\*/

status = 2

abnormal termination,signal number =2

➜ ./t\_sys "nosuchcmd" /\*执行一个不存在的命令\*/

sh: 1: nosuchcmd: not found

cannot invoke shell to exec command(nosuchcmd)

### 5.9.2 system函数的实现

上一小节介绍了system函数的用法，并且引入了一个system函数的简单不完备实现。之所以说是不完备，是因为没有考虑信号。如何正确处理信号，给system的实现带来了复杂度。

首先要考虑SIGCHLD。如果调用system函数的进程存在其他子进程，并且对SIGCHLD信号的处理函数也执行了wait()。这种情况下，当由system()创建子进程退出时并产生SIGCHLD信号时，主程序的信号处理函数可能先执行，导致system函数内部的waitpid无法等待子进程的退出。这就产生了竞争。自己调用fork()创建的子进程和system函数内部创建的子进程就会给程序带来困扰：

* 程序会误认为自己调用fork创建的子进程退出了
* system函数内部的waitpid返回失败，无法获取内部子进程的终止状态。

由于上面的原因，system运行期间必须要暂时阻塞SIGCHLD信号。

其他需要考虑的信号是由终端的中断操作（一般是ctrl+c）和退出操作（一般是ctrl+\）产生的SIGINT信号和SIGQUIT信号。

调用system函数会创建shell子进程，有shell子进程再创建子进程来执行command。

那么三个进程如何应对呢？SUSv3标准规定：

* 调用system函数的进程，需要忽略SIGINT和SIGQUIT信号
* system函数内部创建的进程，要恢复对SIGINT和SIGQUIT的默认处理

从逻辑上讲，当命令传入给system开始执行时，调用system函数的进程，其实已经放弃了控制权。所以调用system函数的进程不应该响应SIGINT信号和SIGQUIT信号，而应该由system内部创建的子进程负责响应。考虑到system函数执行的可能是交互式应用，交给system创建的子进程响应SIGINT和SIGQUIT信号更是合情合理。

用更通俗的话讲，就是调用system函数，在system返回之前会忽略SIGINT和SIGQUIT，无论是调用采用终端的操作（ctrl+c 或者ctrl+\）,还是采用kill来发送SIGINT或SIGQUIT信号，调用system函数的进程会不动如山。但是system内部创建的执行command的子进程，对SIGINT和SIGQUIT的响应是默认值，也就是会杀掉响应的子进程而导致system函数的返回。

Linux/UNIX系统编程手册提供了一个可读性很好的system函数实现。下面我们赏析下glibc的实现。

static int

do\_system (const char \*line)

{

int status, save;

pid\_t pid;

struct sigaction sa;

#ifndef \_LIBC\_REENTRANT

struct sigaction intr, quit;

#endif

sigset\_t omask;

sa.sa\_handler = SIG\_IGN;

sa.sa\_flags = 0;

\_\_sigemptyset (&sa.sa\_mask);

DO\_LOCK ();

if (ADD\_REF () == 0)

{ /\*调用system的进程要忽略SIGINT和SIGQUIT\*/

if (\_\_sigaction (SIGINT, &sa, &intr) < 0)

{

(void) SUB\_REF ();

goto out;

}

if (\_\_sigaction (SIGQUIT, &sa, &quit) < 0)

{

save = errno;

(void) SUB\_REF ();

goto out\_restore\_sigint;

}

}

DO\_UNLOCK ();

/\* We reuse the bitmap in the 'sa' structure. \*/

\_\_sigaddset (&sa.sa\_mask, SIGCHLD);

save = errno;

if (\_\_sigprocmask (SIG\_BLOCK, &sa.sa\_mask, &omask) < 0)

{

#ifndef \_LIBC

if (errno == ENOSYS)

\_\_set\_errno (save);

else

#endif

{

/\*阻塞SIGCHLD失败的话，恢复SIGINT和SIGQUIT，返回失败\*/

DO\_LOCK ();

if (SUB\_REF () == 0)

{

save = errno;

(void) \_\_sigaction (SIGQUIT, &quit, (struct sigaction \*) NULL);

out\_restore\_sigint:

(void) \_\_sigaction (SIGINT, &intr, (struct sigaction \*) NULL);

\_\_set\_errno (save);

}

out:

DO\_UNLOCK ();

return -1;

}

}

#ifdef CLEANUP\_HANDLER

CLEANUP\_HANDLER;

#endif

#ifdef FORK

pid = FORK ();

#else

pid = \_\_fork ();

#endif

if (pid == (pid\_t) 0)

{

/\*子进程对SIGCHLD和SIGINT及SIGQUIT采用默认值，

\* 即调用system函数之前的值. \*/

const char \*new\_argv[4];

new\_argv[0] = SHELL\_NAME;

new\_argv[1] = "-c";

new\_argv[2] = line;

new\_argv[3] = NULL;

/\* Restore the signals. \*/

(void) \_\_sigaction (SIGINT, &intr, (struct sigaction \*) NULL);

(void) \_\_sigaction (SIGQUIT, &quit, (struct sigaction \*) NULL);

(void) \_\_sigprocmask (SIG\_SETMASK, &omask, (sigset\_t \*) NULL);

INIT\_LOCK ();

/\* Exec the shell. \*/

(void) \_\_execve (SHELL\_PATH, (char \*const \*) new\_argv, \_\_environ);

\_exit (127);

}

else if (pid < (pid\_t) 0)

/\* The fork failed. \*/

status = -1;

else

/\* Parent side. \*/

{

/\* Note the system() is a cancellation point. But since we call

waitpid() which itself is a cancellation point we do not

have to do anything here. \*/

if (TEMP\_FAILURE\_RETRY (\_\_waitpid (pid, &status, 0)) != pid)

status = -1;

}

#ifdef CLEANUP\_HANDLER

CLEANUP\_RESET;

#endif

save = errno;

DO\_LOCK ();

if ((SUB\_REF () == 0

&& (\_\_sigaction (SIGINT, &intr, (struct sigaction \*) NULL)

| \_\_sigaction (SIGQUIT, &quit, (struct sigaction \*) NULL)) != 0)

|| \_\_sigprocmask (SIG\_SETMASK, &omask, (sigset\_t \*) NULL) != 0)

{

#ifndef \_LIBC

/\* glibc cannot be used on systems without waitpid. \*/

if (errno == ENOSYS)

\_\_set\_errno (save);

else

#endif

status = -1;

}

DO\_UNLOCK ();

return status;

}

我们对t\_sys对应的进程执行kill -SIGINT，进程t\_sys无动于衷。但是在另一终端，对sleep 1000对应的进程发送SIGINT信号，立刻出现如下打印：

➜ ./t\_sys "sleep 1000"

status = 2

abnormal termination,signal number =2

1. fork拿什么来证明你的写实拷贝（COW）http://blog.chinaunix.net/uid-24774106-id-3361500.html [↑](#footnote-ref-1)
2. Linux系统文件描述符继承带来的危害：http://www.80sec.com/security-issue-on-linux-fd-inheritance.html [↑](#footnote-ref-2)
3. Ulrich Drepper Secure File Descriptor Handling [↑](#footnote-ref-3)
4. 大量参考著名程序员陈皓的vfork挂掉的一个问题 [↑](#footnote-ref-4)