# linux进程学习笔记

摘自：http://www.cnblogs.com/mickole/category/496206.html

## [linux系统编程之进程（一）：进程与程序](http://www.cnblogs.com/mickole/p/3185610.html)

本节目标：

* 什么是程序
* 什么是进程
* 进程数据结构
* 进程与程序区别与联系

### 一，什么是程序？

程序是完成特定任务的一系列指令集合

### 二，什么是进程？

* 从用户的角度来看进程是程序的一次动态执行过程
* 从操作系统的核心来看，进程是操作系统分配的内存、CPU时间片等资源的基本单位。
* 进程是资源分配的最小单位
* 每一个进程都有自己独立的地址空间与执行状态。
* 像UNIX这样的多任务操作系统能够让许多程序同时运行，每一个运行着的程序就构成了一个进程

### 三，进程数据结构

进程的静态描述：由三部分组成:PCB、有关程序段和该程序段对其进行操作的数据结构集。

进程控制块（PCB)：用于描述进程情况及控制进程运行所需的全部信息，是操作系统用来感知进存在的一个重要数据结构。

代码段：是进程中能被进程调度程序在CPU上执行的程序代码段。

数据段：一个进程的数据段，可以是进程对应的程序加工处理的原始数据，也可以是程序执行后产生的中间或最终数据

进程=代码段（编译后形成的一些指令）+数据段（程序运行时需要的数据）+堆栈段（程序运行时动态分配的一些内存）+PCB（进程信息，状态标识等）

数据段包括：

* 只读数据段：常量
* 已初始化数据段：全局变量，静态变量
* 位初始化数据段（bss)（0初始化段）：未初始化的全局变量和静态变量（实际上不分配内存，因为都为0，只有一些标记信息）

### 四，进程与程序区别与联系

* 进程是动态的，程序是静态的
* 进程的生命周期是相对短暂的，而程序是永久的。
* 进程数据结构PCB
* 一个进程只能对应一个程序，一个程序可以对应多个进程。

## [linux系统编程之进程（二）：进程生命周期与PCB（进程控制块）](http://www.cnblogs.com/mickole/p/3185889.html)

本节目标：

* 进程状态变迁
* 进程控制块
* 进程创建
* 进程撤消
* 终止进程的五种方法

### 一，进程状态变迁

进程的三种基本状态

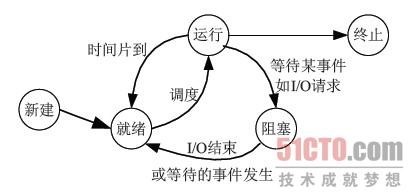
**就绪(Ready)状态**

当进程已分配到除CPU以外的所有必要的资源，只要获得处理机便可立即执行，这时的进程状态称为就绪状态。

**执行（Running）状态**   
当进程已获得处理机，其程序正在处理机上执行，此时的进程状态称为执，行状态。

**阻塞(Blocked)状态**   
正在执行的进程，由于等待某个事件发生而无法执行时，便放弃处理机而处于阻塞状态。引起进程阻塞的事件可有多种，例如，等待I/O完成、申请缓冲区不能满足、等待信件(信号)等。

状态转换图：

[](http://images.cnitblog.com/blog/529981/201307/12110200-df9756a6cb444de0985f3eef61cfbff1.jpg)

一个进程在运行期间，不断地从一种状态转换到另一种状态，它可以多次处于就绪状态和执行状态，也可以多次处于阻塞状态。

(1) 就绪→执行   
处于就绪状态的进程，当进程调度程序为之分配了处理机后，该进程便由就绪状态转变成执行状态。

(2) 执行→就绪   
处于执行状态的进程在其执行过程中，因分配给它的一个时间片已用完或更高优先级的进程抢占而不得不让出处理机，于是进程从执行状态转变成就绪状态。

(3) 执行→阻塞   
正在执行的进程因等待某种事件发生而无法继续执行时，便从执行状态变成阻塞状态。

(4) 阻塞→就绪   
处于阻塞状态的进程，若其等待的事件已经发生，于是进程由阻塞状态转变为就绪状态。

(5) 运行→终止

程序执行完毕，撤销而终止

以上是最经典也是最基本的三种进程状态，但现在的操作系统都根据需要重新设计了一些新的状态。

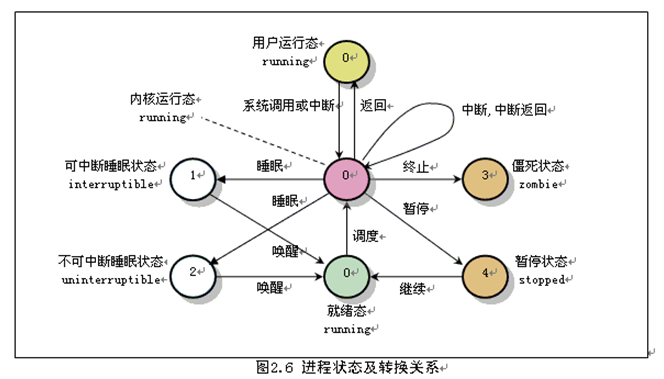
如linux：

* 运行状态（TASK\_RUNNING）：是运行态和就绪态的合并，表示进程正在运行或准备运行，Linux 中使用TASK\_RUNNING 宏表示此状态
* 可中断睡眠状态（浅度睡眠）（TASK\_INTERRUPTIBLE）：进程正在睡眠（被阻塞），等待资源到来是唤醒，也可以通过其他进程信号或时钟中断唤醒，进入运行队列。Linux 使用TASK\_INTERRUPTIBLE 宏表示此状态。
* 不可中断睡眠状态（深度睡眠状态）（TASK\_UNINTERRUPTIBLE）：

其和浅度睡眠基本类似，但有一点就是不可被其他进程信号或时钟中断唤醒。Linux 使用TASK\_UNINTERRUPTIBLE 宏表示此状态。

* 暂停状态（TASK\_STOPPED）：进程暂停执行接受某种处理。如正在接受调试的进程处于这种状态，Linux 使用TASK\_STOPPED 宏表示此状态。
* 僵死状态（TASK\_ZOMBIE）：进程已经结束但未释放PCB，Linux 使用TASK\_ZOMBIE 宏表示此状态

状态转换图如下：

[](http://images.cnitblog.com/blog/529981/201307/12110202-609ccaf975ee44ceb92ef8b4ffa18cb6.png)

      就绪态在支持虚拟内存的操作系统中可能分为：在内存中就绪和在交换空间就绪，因为支持虚拟内存的操作系统会将当前不太活跃的一些进程交换到交换空间，交换空间就绪进程要运行要先换人内存出入内存就绪态，在由cpu调度执行。

### 二，进程控制块（PCB)

##### 进程控制块包括：

进程描述信息：

* 进程标识符用于唯一的标识一个进程（pid，ppid）。

进程控制信息：

* 进程当前状态
* 进程优先级
* 程序开始地址
* 各种计时信息
* 通信信息

资源信息：

* 占用内存大小及管理用数据结构指针
* 交换区相关信息
* I/O设备号、缓冲、设备相关的数结构
* 文件系统相关指针

现场保护信息（cpu进行进程切换时）：

* 寄存器
* PC
* 程序状态字PSW
* 栈指针

##### 进程标识：PID

* 每个进程都会分配到一个独一无二的数字编号，我们称之为“进程标识”(process identifier),或者就直接叫它PID.
* 是一个正整数，取值范围从2到32768

     可以通过：cat /proc/sys/kernel/pid\_max 查看系统支持多少进程

* 当一个进程被启动时，它会顺序挑选下一个未使用的编号数字做为自己的PID
* 数字1一般为特殊进程init保留的

         init进程实际上是用户进程，它是一个程序，在/sbin/init，linux启动的第一个进程

        实际上linux中还存在0号进程（内核进程），它是一个空闲进程，它进行空闲资源的统计及交换空间的换入换出，1（init）进程是由0号进程创建的。

### 三，进程创建

* 不同的操作系统所提供的进程创建原语的名称和格式不尽相同，但执行创建进程原语后，操作系统所做的工作却大致相同，都包括以下几点：
* 给新创建的进程分配一个内部标识（pcb），在内核中建立进程结构。
* 复制父进程的环境
* 为进程分配资源， 包括进程映像所需要的所有元素（程序、数据、用户栈等），
* 复制父进程地址空间的内容到该进程地址空间中。
* 置该进程的状态为就绪，插入就绪队列。

### 四，进程撤销

进程终止时操作系统做以下工作：

* 关闭软中断:因为进程即将终止而不再处理任何软中断信号；
* 回收资源：释放进程分配的所有资源，如关闭所有已打开文件，释放进程相应的数据结构等；
* 写记帐信息：将进程在运行过程中所产生的记帐数据（其中包括进程运行时的各种统计信息）记录到一个全局记帐文件中；
* 置该进程为僵死状态:向父进程发送子进程死的软中断信号，将终止信息status送到指定的存储单元中；
* 转进程调度:因为此时CPU已经被释放，需要由进程调度进行CPU再分配。

### 五，终止进程的五种方法

* 从main函数返回：从return返回，执行完毕退出
* 调用exit：C函数库，实际上也是调用系统调用\_exit完成的，在任何一个函数调用exit函数都可使得进程撤销
* 调用\_exit：系统调用
* 调用abort：调用abort()函数使得进程终止，实际上该函数是产生一个SIGABRT信号，
* 由信号终止：发送一些信号如SINGINT等信号

## [linux系统编程之进程（三）：进程复制fork，孤儿进程，僵尸进程](http://www.cnblogs.com/mickole/p/3186441.html)

本节目标：

* 复制进程映像
* fork系统调用
* 孤儿进程、僵尸进程
* 写时复制

### 一，进程复制（或产生）

     使用fork函数得到的子进程从父进程的继承了整个进程的地址空间，包括：进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设置、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端等。

子进程与父进程的区别在于：

1、父进程设置的锁，子进程不继承（因为如果是排它锁，被继承的话，矛盾了）

2、各自的进程ID和父进程ID不同

3、子进程的未决告警被清除；

4、子进程的未决信号集设置为空集。

### 二，fork系统调用

包含头文件 <sys/types.h> 和 <unistd.h>

函数功能:创建一个子进程

函数原型

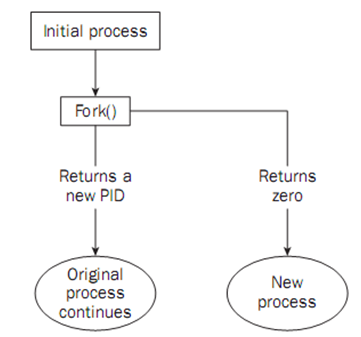
pid\_t fork(void);  //一次调用两次返回值，是在各自的地址空间返回，意味着现在有两个基本一样的进程在执行

参数：无参数。

返回值:

* 如果成功创建一个子进程，对于父进程来说返回子进程ID
* 如果成功创建一个子进程，对于子进程来说返回值为0
* 如果为-1表示创建失败

流程图：

[](http://images.cnitblog.com/blog/529981/201307/12154305-d2424314b5df43a6bcfc28abba7c512c.png)

父进程调用fork（）系统调用，然后陷入内核，进行进程复制，如果成功：

1，则对调用进程即父进程来说返回值为刚产生的子进程pid，因为进程PCB没有子进程信息，父进程只能通过这样获得。

2，对子进程（刚产生的新进程），则返回0，

这时就有两个进程在接着向下执行

如果失败，则返回0，调用进程继续向下执行

注：fork英文意思：分支，fork系统调用复制产生的子进程与父进程（调用进程）基本一样：代码段+数据段+堆栈段+PCB，当前的运行环境基本一样，所以子进程在fork之后开始向下执行，而不会从头开始执行。

示例程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#define ERR\_EXIT(m) \

do\

{\

perror(m);\

exit(EXIT\_FAILURE);\

}\

while (0)\

int main(void)

{

pid\_t pid;

printf("before calling fork,calling process pid = %d\n",getpid());

pid = fork();

if(pid == -1)

ERR\_EXIT("fork error");

if(pid == 0){

printf("this is child process and child's pid = %d,parent's pid = %d\n",getpid(),getppid());

}

if(pid > 0){

//sleep(1);

printf("this is parent process and pid =%d ,child's pid = %d\n",getpid(),pid);

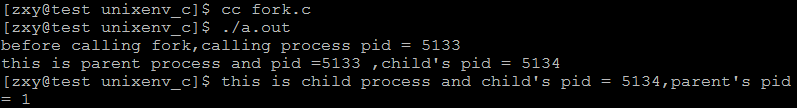
}

return 0;

}

[复制代码](javascript:void(0);)

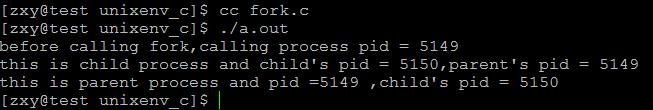
运行结果：

[](http://images.cnitblog.com/blog/529981/201307/12154306-45b4a6bcab6d4821afe339c69c040df6.png)

当没给父进程没加sleep时，由于父进程先执行完，子进程成了孤儿进程，系统将其托孤给了1（init）进程，

所以ppid =1。

当加上sleep后，子进程先执行完：

[](http://images.cnitblog.com/blog/529981/201307/12154307-06d5f4fc3e1f44a78162f2a9f128de06.png)

这次可以正确看到想要的结果。

### 三，孤儿进程、僵尸进程

fork系统调用之后，父子进程将交替执行，执行顺序不定。

如果父进程先退出，子进程还没退出那么子进程的父进程将变为init进程（托孤给了init进程）。（注：任何一个进程都必须有父进程）

如果子进程先退出，父进程还没退出，那么子进程必须等到父进程捕获到了子进程的退出状态才真正结束，否则这个时候子进程就成为僵进程（僵尸进程：只保留一些退出信息供父进程查询）

示例：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#define ERR\_EXIT(m) \

do\

{\

perror(m);\

exit(EXIT\_FAILURE);\

}\

while (0)\

int main(void)

{

pid\_t pid;

printf("before calling fork,calling process pid = %d\n",getpid());

pid = fork();

if(pid == -1)

ERR\_EXIT("fork error");

if(pid == 0){

printf("this is child process and child's pid = %d,parent's pid = %d\n",getpid(),getppid());

}

if(pid > 0){

sleep(100);

printf("this is parent process and pid =%d ,child's pid = %d\n",getpid(),pid);

}

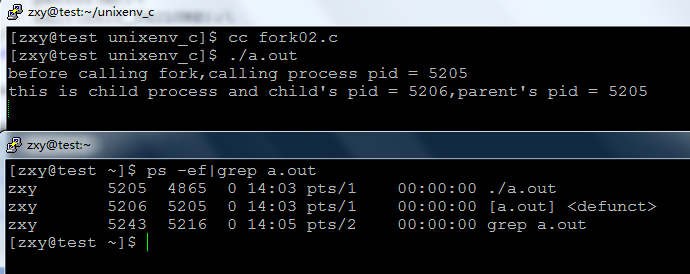
return 0;

}

[复制代码](javascript:void(0);)

以上程序跟前面那个基本一致，就是让父进程睡眠100秒，好让子进程先退出

运行结果：

[](http://images.cnitblog.com/blog/529981/201307/12154307-bd092460fb3a4060a5d89ba520ce7334.png)

从上可以看到，子进程先退出，但进程列表中还可以查看到子进程，[a.out] <defunct>，死的意思，即僵尸进程，如果系统中存在过多的僵尸进程，将会使得新的进程不能产生。

### 四，写时复制

linux系统为了提高系统性能和资源利用率，在fork出一个新进程时，系统并没有真正复制一个副本。

如果多个进程要读取它们自己的那部分资源的副本，那么复制是不必要的。

每个进程只要保存一个指向这个资源的指针就可以了。

如果一个进程要修改自己的那份资源的“副本”，那么就会复制那份资源。这就是写时复制的含义

#### fork 和vfork：

在fork还没实现copy on write之前。Unix设计者很关心fork之后立刻执行exec所造成的地址空间浪费，所以引入了vfork系统调用。

vfork有个限制，子进程必须立刻执行\_exit或者exec函数。

即使fork实现了copy on write，效率也没有vfork高，但是我们不推荐使用vfork，因为几乎每一个vfork的实现，都或多或少存在一定的问题

vfork：

Linux Description   
    vfork(), just like fork(2), creates a child process of the calling pro-   
    cess.  For details and return value and errors, see fork(2).

    vfork()  is  a special case of clone(2).  It is used to create new pro-   
    cesses without copying the page tables of the parent process.   It  may   
    be  useful  in performance-sensitive applications where a child will be   
    created which then immediately issues an execve(2).

    vfork() differs from fork(2) in that the parent is suspended until  the   
    child  terminates (either normally, by calling \_exit(2), or abnormally,   
    after delivery of a fatal signal), or it makes  a  call  to  execve(2).   
    Until  that point, the child shares all memory with its parent, includ-   
    ing the stack. The child must not return from the current function  or   
    call exit(3), but may call \_exit(2).

    Signal  handlers  are inherited, but not shared.  Signals to the parent   
    arrive after the child releases the parent’s memory  (i.e.,  after  the   
    child terminates or calls execve(2)).

示例程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#define ERR\_EXIT(m) \

do\

{\

perror(m);\

exit(EXIT\_FAILURE);\

}\

while (0)\

int main(void)

{

pid\_t pid;

int val = 1;

printf("before calling fork, val = %d\n",val);

//pid = fork();

pid = vfork();

if(pid == -1)

ERR\_EXIT("fork error");

if(pid == 0){

printf("chile process,before change val, val = %d\n",val);

val++;

//sleep(1);

printf("this is child process and val = %d\n",val);

\_exit(0);

}

if(pid > 0){

sleep(1);

//val++;

printf("this is parent process and val = %d\n",val);

}

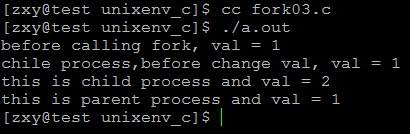
return 0;

}

[复制代码](javascript:void(0);)

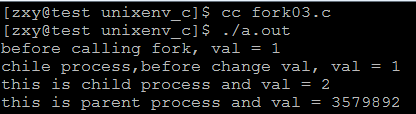
当调用fork时:

运行结果：

[](http://images.cnitblog.com/blog/529981/201307/12154308-b3080da11e6f4fd79530a43bbaeef759.png)

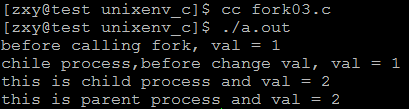
可知写时复制

当使用vfork但子进程没使用exit退出时：

[](http://images.cnitblog.com/blog/529981/201307/12154309-f9e1133756a84d55aebe8f1ea30a82c4.png)

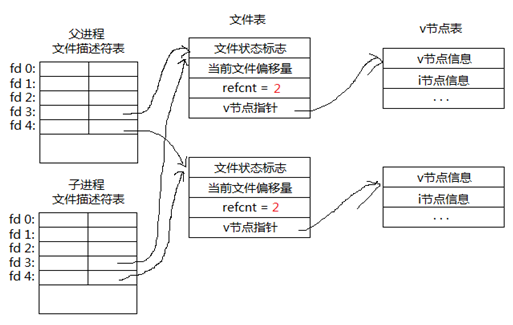
结果出错了，

使用vfork且exit退出：

[](http://images.cnitblog.com/blog/529981/201307/12154310-75b65d273a9c4a1eb9348b33d16563da.png)

结果正常，父子进程共享

#### fork之后父子进程共享文件：

[](http://images.cnitblog.com/blog/529981/201307/12154311-ea17078d51304044909b760d7815925d.png)

fork产生的子进程与父进程相同的文件文件描述符指向相同的文件表，引用计数增加，共享文件文件偏移指针

示例程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

#define ERR\_EXIT(m) \

do\

{\

perror(m);\

exit(EXIT\_FAILURE);\

}\

while (0)\

int main(void)

{

pid\_t pid;

int fd;

fd = open("test.txt",O\_WRONLY);

if(fd == -1)

ERR\_EXIT("OPEN ERROR");

pid = fork();

if(pid == -1)

ERR\_EXIT("fork error");

if(pid == 0){

write(fd,"child",5);

}

if(pid > 0){

//sleep(1);

write(fd,"parent",6);

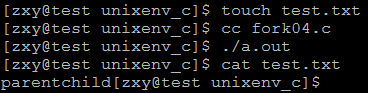
}

return 0;

}

[复制代码](javascript:void(0);)

运行结果：

[](http://images.cnitblog.com/blog/529981/201307/12154312-90bba6070f3b4aa78dcb3a585606f57a.png)

可知父子进程共享文件偏移指针，父进程写完后文件偏移到parent后子进程开始接着写。

## [linux系统编程之进程（四）：进程退出exit，\_exit区别即atexit函数](http://www.cnblogs.com/mickole/p/3186606.html)

### 一，进程终止有5种方式：

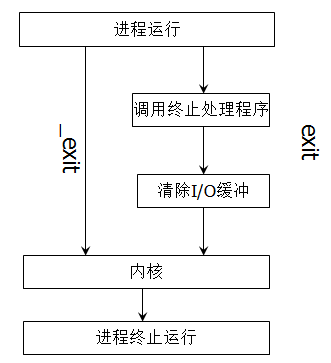
正常退出：

* 从main函数返回
* 调用exit
* 调用\_exit

异常退出：

* 调用abort
* 由信号终止

### 二，exit和\_exit区别：

[](http://images.cnitblog.com/blog/529981/201307/12164357-7fc5492821244faaa2a6ffd703850007.png)

##### 关于\_exit()：

       #include <unistd.h>

       void \_exit(int status);

       #include <stdlib.h>

       void \_Exit(int status);

DESCRIPTION   
       The function \_exit() terminates the calling process "immediately".  Any   
       open file descriptors belonging to the process are closed; any children   
       of the process are inherited by process 1, init, and the process’s par-   
       ent is sent a SIGCHLD signal.

       The value status is returned to the parent  process  as  the  process’s   
       exit  status,  and  can be collected using one of the wait(2) family of   
       calls.

       The function \_Exit() is equivalent to \_exit().

##### 关于exit()：

#include <stdlib.h>

void exit(int status);

DESCRIPTION   
       The  exit() function causes normal process termination and the value of   
       status & 0377 is returned to the parent (see wait(2)).

       All functions registered with atexit(3) and on\_exit(3) are  called,  in   
       the  reverse  order  of their registration.  (It is possible for one of   
       these functions to use atexit(3) or on\_exit(3)  to  register  an  addi-   
       tional  function  to be executed during exit processing; the new regis-   
       tration is added to the front of the list of functions that  remain  to   
       be  called.) If one of these functions does not return (e.g., it calls   
       \_exit(2), or kills itself with a signal), then none  of  the  remaining   
       functions is called, and further exit processing (in particular, flush-   
       ing of stdio(3) streams) is abandoned.  If a function has  been  regis-   
       tered  multiple  times using atexit(3) or on\_exit(3), then it is called   
       as many times as it was registered.

       All open stdio(3) streams are flushed and  closed.   Files  created  by   
       tmpfile(3) are removed.

       The  C standard specifies two constants, EXIT\_SUCCESS and EXIT\_FAILURE,   
       that may be passed to exit() to  indicate  successful  or  unsuccessful   
       termination, respectively.

**和exit比较一下，exit()函数定义在stdlib.h中，而\_exit()定义在unistd.h中，**

注：exit()就是退出，传入的参数是程序退出时的状态码，0表示正常退出，其他表示非正常退出，一般都用-1或者1，标准C里有EXIT\_SUCCESS和EXIT\_FAILURE两个宏，用exit(EXIT\_SUCCESS);

**\_exit()函数的作用最为简单：直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构；exit() 函数则在这些基础上作了一些包装，在执行退出之前加了若干道工序。   
exit()函数与\_exit()函数最大的区别就在于exit()函数在调用exit系统调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，就是"清理I/O缓冲"。**

exit()在结束调用它的进程之前，要进行如下步骤：   
1.调用atexit()注册的函数（出口函数）；按ATEXIT注册时相反的顺序调用所有由它注册的函数,这使得我们可以指定在程序终止时执行自己的清理动作.例如,保存程序状态信息于某个文件,解开对共享数据库上的锁等.

2.cleanup()；**关闭所有打开的流，这将导致写所有被缓冲的输出**，删除用TMPFILE函数建立的所有临时文件.

3.最后调用\_exit()函数终止进程。

\_exit做3件事（man）：   
1，Any  open file descriptors belonging to the process are closed   
2，any children of the process are inherited  by process 1, init   
3，the process's parent is sent a SIGCHLD signal

exit执行完清理工作后就调用\_exit来终止进程。

### 三，atexit()

atexit可以注册终止处理程序，ANSI C规定最多可以注册32个终止处理程序。

终止处理程序的调用与注册次序相反

       #include <stdlib.h>

       int atexit(void (\*function)(void));

DESCRIPTION   
       The atexit() function registers the given function to be called at nor-   
       mal process termination, either via exit(3) or via return from the pro-   
       gram’s main(). Functions so registered are called in the reverse order   
       of their registration; no arguments are passed.

       The  same  function may be registered multiple times: it is called once   
       for each registration.

       POSIX.1-2001 requires that an implementation allow at least  ATEXIT\_MAX   
       (32) such functions to be registered.  The actual limit supported by an   
       implementation can be obtained using sysconf(3).

       When a child process is created via fork(2), it inherits copies of  its   
       parent’s  registrations.   Upon a successful call to one of the exec(3)   
       functions, all registrations are removed.

RETURN VALUE   
       The atexit() function returns the value 0 if successful;  otherwise  it   
       returns a non-zero value.

示例程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

void fun1()

{

printf("fun1 is called\n");

}

void fun2()

{

printf("fun2 is called\n");

}

int main(void)

{

printf("main function\n");

atexit(fun1);

atexit(fun2);

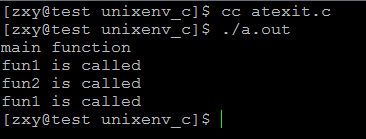
atexit(fun1);

exit(EXIT\_SUCCESS);

}

[复制代码](javascript:void(0);)

运行结果：

[](http://images.cnitblog.com/blog/529981/201307/12164358-fecc2c9fdc24494e82af9767bef3d89b.png)

当调用fork时，子进程继承父进程注册的atexit：

示例程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#define ERR\_EXIT(m) \

do\

{\

perror(m);\

exit(EXIT\_FAILURE);\

}\

while (0)\

void fun1()

{

printf("fun1 is called\n");

}

void fun2()

{

printf("fun2 is called\n");

}

int main(void)

{

pid\_t pid;

pid = fork();

atexit(fun1);

atexit(fun2);

atexit(fun1);

if(pid == -1)

ERR\_EXIT("fork error");

if(pid == 0){

printf("this is child process\n");

}

if(pid > 0){

printf("this is parent process\n");

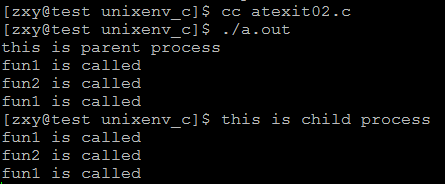
}

return 0;

}

[复制代码](javascript:void(0);)

运行结果：

[](http://images.cnitblog.com/blog/529981/201307/12164359-45c40d0ac180454483740d88aaec2762.png)

当atexit注册的函数中有一个没有正常返回或被kill，则后续的注册函数都不会被执行

示例程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <signal.h>

void fun1()

{

printf("fun1 is called\n");

}

void fun2()

{

printf("fun2 is called\n");

kill(getpid(),SIGINT);

}

int main(void)

{

printf("main function\n");

if(signal(SIGINT,SIG\_DFL) == SIG\_ERR){

perror("signal error");

exit(EXIT\_FAILURE);

}

atexit(fun1);

atexit(fun2);

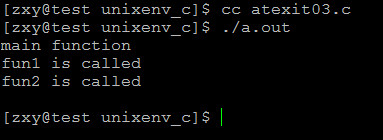
atexit(fun1);

exit(EXIT\_SUCCESS);

}

[复制代码](javascript:void(0);)

运行结果：

[](http://images.cnitblog.com/blog/529981/201307/12164400-cecbda540aa841178ce7bc6a75ebff3f.png)

可见最后那个fun1没有执行

## [linux系统编程之进程（五）：exec系列函数（execl,execlp,execle,execv,execvp)使用](http://www.cnblogs.com/mickole/p/3187409.html)

本节目标：

* exec替换进程映像
* exec关联函数组（execl、execlp、execle、execv、execvp）

### 一，exec替换进程映像

在进程的创建上Unix采用了一个独特的方法，它将进程创建与加载一个新进程映象分离。这样的好处是有更多的余地对两种操作进行管理。

当我们创建了一个进程之后，通常将子进程替换成新的进程映象，这可以用exec系列的函数来进行。当然，exec系列的函数也可以将当前进程替换掉。

例如：在shell命令行执行ps命令，实际上是shell进程调用fork复制一个新的子进程，在利用exec系统调用将新产生的子进程完全替换成ps进程。

### 二，exec系列函数（execl、execlp、execle、execv、execvp）

包含头文件<unistd.h>

功能：

    用exec函数可以把当前进程替换为一个新进程，且新进程与原进程有相同的PID。exec名下是由多个关联函数组成的一个完整系列，

头文件<unistd.h>

extern char \*\*environ;

原型：

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[]);

参数：

path参数表示你要启动程序的名称包括路径名

arg参数表示启动程序所带的参数，一般第一个参数为要执行命令名，不是带路径且arg必须以NULL结束

返回值:成功返回0,失败返回-1

注：上述exec系列函数底层都是通过execve系统调用实现：

       #include <unistd.h>

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

DESCRIPTION：   
       execve() executes the program pointed to by filename.  filename must be   
       either a binary executable, or a script starting with  a  line  of  the form

以上exec系列函数区别：

1，带l 的exec函数：execl,execlp,execle，表示后边的参数以可变参数的形式给出且都以一个空指针结束。

示例：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

printf("entering main process---\n");

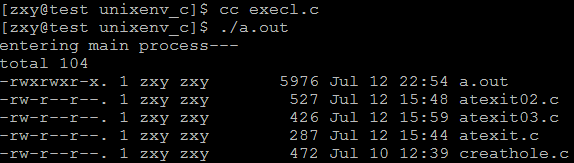
execl("/bin/ls","ls","-l",NULL);

printf("exiting main process ----\n");

return 0;

}

[复制代码](javascript:void(0);)

[](http://images.cnitblog.com/blog/529981/201307/12235434-d52eca31e0b24983a074594ddb3d46a3.png)

利用execl将当前进程main替换掉，所有最后那条打印语句不会输出

2，带 p 的exec函数：execlp,execvp，表示第一个参数path不用输入完整路径，只有给出命令名即可，它会在环境变量PATH当中查找命令

示例：

当不带p但没给出完整路径时：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

printf("entering main process---\n");

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

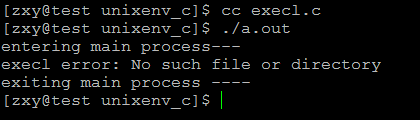
printf("exiting main process ----\n");

return 0;

}

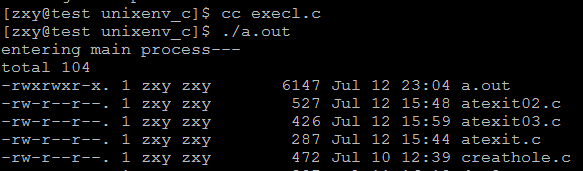
[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/12235435-1ad467bb6b2e4ecdb4efe52444c38c23.png)

结果显示找不到，所有替换不成功，main进程继续执行

现在带p：

[](http://images.cnitblog.com/blog/529981/201307/12235436-bd5b7af00c2043c79e365886eecad3df.png)

替换成功

3，不带 l 的exec函数：execv,execvp表示命令所需的参数以char \*arg[]形式给出且arg最后一个元素必须

是NULL

示例：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

printf("entering main process---\n");

int ret;

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

ret = execvp("ls",argv);

if(ret == -1)

perror("execl error");

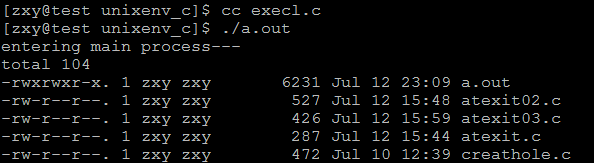
printf("exiting main process ----\n");

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/12235436-31b9ed4b2ced4a48ad92a7a9c98c971d.png)

进程替换成功

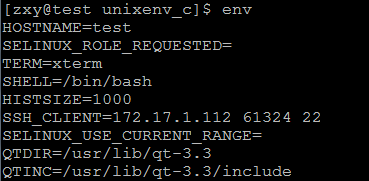
4，带 e 的exec函数：execle表示，将环境变量传递给需要替换的进程

从上述的函数原型中我们发现：

extern char \*\*environ;

此处的environ是一个指针数组，它当中的每一个指针指向的char为“XXX=XXX”

environ保存环境信息的数据可以env命令查看：

[](http://images.cnitblog.com/blog/529981/201307/12235437-917eb8467032470c983b06a1c0fa37dc.png)

它由shell进程传递给当前进程，再由当前进程传递给替换的新进程

示例：execle.c

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

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

{

//char \* const envp[] = {"AA=11", "BB=22", NULL};

printf("Entering main ...\n");

int ret;

ret =execl("./hello", "hello", NULL);

//execle("./hello", "hello", NULL, envp);

if(ret == -1)

perror("execl error");

printf("Exiting main ...\n");

return 0;

}

[复制代码](javascript:void(0);)

hello.c

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <stdio.h>

extern char\*\* environ;

int main(void)

{

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

int i;

for (i=0; environ[i]!=NULL; ++i)

{

printf("%s\n", environ[i]);

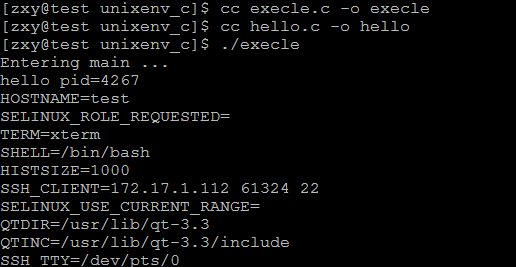
}

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/12235438-4f7fe83fe15e4f0fb46d01aef1cff883.png)

可知原进程确实将环境变量信息传递给了新进程

那么现在我们可以利用execle函数自己给的需要传递的环境变量信息：

示例程序：execle.c

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

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

{

char \* const envp[] = {"AA=11", "BB=22", NULL};

printf("Entering main ...\n");

int ret;

//ret =execl("./hello", "hello", NULL);

ret =execle("./hello", "hello", NULL, envp);

if(ret == -1)

perror("execl error");

printf("Exiting main ...\n");

return 0;

}

[复制代码](javascript:void(0);)

hello.c

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <stdio.h>

extern char\*\* environ;

int main(void)

{

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

int i;

for (i=0; environ[i]!=NULL; ++i)

{

printf("%s\n", environ[i]);

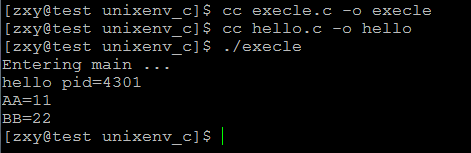
}

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/12235438-4c0f2b75cc77413da14b1db3d3bd5141.png)

确实将给定的环境变量传递过来了

### 三，fcntl()函数中的FD\_CLOEXEC标识在exec系列函数中的作用

#include <unistd.h>   
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /\* arg \*/ );

File descriptor flags   
      The following commands manipulate the  flags  associated  with  a  file   
      descriptor.   Currently, only one such flag is defined: FD\_CLOEXEC, the   
      close-on-exec flag.  If the FD\_CLOEXEC bit is 0,  the  file  descriptor   
      will remain open across an execve(2), otherwise it will be closed.

     //如果FD\_CLOEXEC标识位为0，则通过execve调用后fd依然是打开的，否则为关闭的

      F\_GETFD (void)   
             Read the file descriptor flags; arg is ignored.

      F\_SETFD (long)   
             Set the file descriptor flags to the value specified by arg.

如：fcntl(fd, F\_SETFD, FD\_CLOEXEC);

测试示例：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

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

{

printf("Entering main ...\n");

int ret = fcntl(1, F\_SETFD, FD\_CLOEXEC);

if (ret == -1)

perror("fcntl error");

int val;

val =execlp("ls", "ls","-l", NULL);

if(val == -1)

perror("execl error");

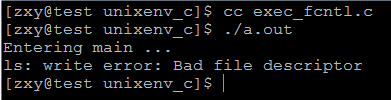
printf("Exiting main ...\n");

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/12235439-0f84e62c0cd44b9c93717205e2ed583c.png)

1关闭（标准输出关闭）ls -l无法将结果显示在标准输出

## [linux系统编程之进程（六）：父进程查询子进程的退出,wait,waitpid](http://www.cnblogs.com/mickole/p/3187770.html)

本节目标：

* 僵尸进程
* SIGCHLD
* wait
* waitpid

### 一，僵尸进程

当一个子进程先于父进程结束运行时，它与其父进程之间的关联还会保持到父进程也正常地结束运行，或者父进程调用了wait才告终止。

子进程退出时，内核将子进程置为僵尸状态，这个进程称为僵尸进程，它只保留最小的一些内核数据结构，以便父进程查询子进程的退出状态。

进程表中代表子进程的数据项是不会立刻释放的，虽然不再活跃了，可子进程还停留在系统里，因为它的退出码还需要保存起来以备父进程中后续的wait调用使用。它将称为一个“僵进程”。

### 二，如何避免僵尸进程

* 调用wait或者waitpid函数查询子进程退出状态，此方法父进程会被挂起。
* 如果不想让父进程挂起，可以在父进程中加入一条语句：signal(SIGCHLD,SIG\_IGN);表示父进程忽略SIGCHLD信号，该信号是子进程退出的时候向父进程发送的。

### 三，SIGCHLD信号

当子进程退出的时候，内核会向父进程发送SIGCHLD信号，子进程的退出是个异步事件（子进程可以在父进程运行的任何时刻终止）

如果不想让子进程编程僵尸进程可在父进程中加入：signal(SIGCHLD,SIG\_IGN);

如果将此信号的处理方式设为忽略，可让内核把僵尸子进程转交给init进程去处理，省去了大量僵尸进程占用系统资源。

示例：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#include <stdlib.h>

int main(void)

{

pid\_t pid;

if(signal(SIGCHLD,SIG\_IGN) == SIG\_ERR)

{

perror("signal error");

exit(EXIT\_FAILURE);

}

pid = fork();

if(pid == -1)

{

perror("fork error");

exit(EXIT\_FAILURE);

}

if(pid == 0)

{

printf("this is child process\n");

exit(0);

}

if(pid > 0)

{

sleep(100);

printf("this is parent process\n");

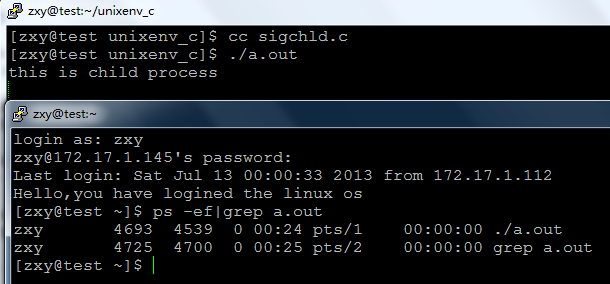
}

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/13113017-5086892aca864a0b92fb6a6a325ee7dd.png)

可知，虽然子进程先退出了，但进程表中已经不存在子进程的僵尸状态

### 三，wait()函数

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

pid\_t wait(int \*status);

进程一旦调用了wait，就立即阻塞自己，由wait自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait就会一直阻塞在这里，直到有一个出现为止。   
参数status用来保存被收集进程退出时的一些状态，它是一个指向int类型的指针。但如果我们对这个子进程是如何死掉的毫不在意，只想把这个僵尸进程消灭掉，（事实上绝大多数情况下，我们都会这样想），我们就可以设定这个参数为NULL，就象下面这样：

pid = wait(NULL);

如果成功，wait会返回被收集的子进程的进程ID，如果调用进程没有子进程，调用就会失败，此时wait返回-1，同时errno被置为ECHILD。

man帮助：

DESCRIPTION   
       All of these system calls are used to wait for state changes in a child   
       of the calling process, and obtain information about  the  child  whose   
       state  has changed.  A state change is considered to be: the child ter-   
       minated; the child was stopped by a signal; or the child was resumed by   
       a  signal.  In the case of a terminated child, performing a wait allows   
       the system to release the resources associated with  the  child;  if  a   
       wait  is not performed, then the terminated child remains in a "zombie"   
       state (see NOTES below).

       If a child has already changed state, then these calls  return  immedi-   
       ately.   Otherwise  they  block until either a child changes state or a   
       signal handler interrupts the call (assuming that system calls are  not   
       automatically restarted using the SA\_RESTART flag of sigaction(2)).  In   
       the remainder of this page, a child whose state has changed  and  which   
       has  not  yet  been  waited upon by one of these system calls is termed   
       waitable.

wait() ：   
    The wait() system call suspends execution of the calling process  until   
    one  of  its children terminates.  The call wait(&status) is equivalent   
    to:

        waitpid(-1, &status, 0);

If status is not NULL, wait() and waitpid() store status information in   
      the  int  to  which  it points.  This integer can be inspected with the   
      following macros (which take the integer itself as an argument,  not  a   
      pointer to it, as is done in wait() and waitpid()!):

      WIFEXITED(status)   
             returns true if the child terminated normally, that is, by call-   
             ing exit(3) or \_exit(2), or by returning from main().

      WEXITSTATUS(status)   
             returns the exit status of the  child.   This  consists  of  the   
             least  significant  8 bits of the status argument that the child   
             specified in a call to exit(3) or \_exit(2) or  as  the  argument   
             for  a  return  statement  in main().  This macro should only be   
             employed if WIFEXITED returned true.

      WIFSIGNALED(status)   
             returns true if the child process was terminated by a signal.

    WTERMSIG(status)   
             returns the number of the signal that caused the  child  process   
             to terminate.  This macro should only be employed if WIFSIGNALED   
             returned true.

      WCOREDUMP(status)   
             returns true if the child produced  a  core  dump.   This  macro   
             should  only  be  employed  if  WIFSIGNALED returned true.  This   
             macro is not specified in POSIX.1-2001 and is not  available  on   
             some  Unix  implementations  (e.g.,  AIX, SunOS).  Only use this   
             enclosed in #ifdef WCOREDUMP ... #endif.

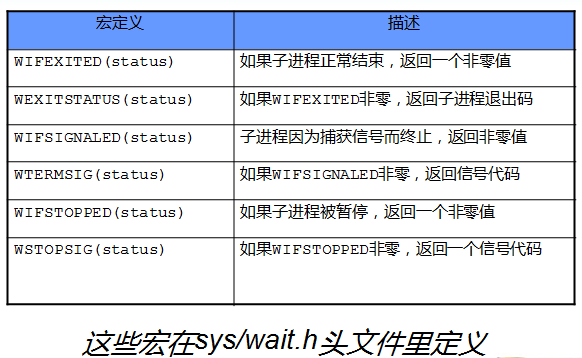
      WIFSTOPPED(status)   
             returns true if the child process was stopped by delivery  of  a   
             signal;  this  is  only possible if the call was done using WUN-   
             TRACED or when the child is being traced (see ptrace(2)).

      WSTOPSIG(status)   
             returns the number of the signal which caused the child to stop.   
             This  macro should only be employed if WIFSTOPPED returned true.

   WIFCONTINUED(status)   
       (since Linux 2.6.10) returns  true  if  the  child  process  was   
       resumed by delivery of SIGCONT.

* wait系统调用会使父进程暂停执行，直到它的一个子进程结束为止。
* 返回的是子进程的PID，它通常是结束的子进程
* 状态信息允许父进程判定子进程的退出状态，即从子进程的main函数返回的值或子进程中exit语句的退出码。
* 如果status不是一个空指针，状态信息将被写入它指向的位置

可以上述的一些宏判断子进程的退出情况：

[](http://images.cnitblog.com/blog/529981/201307/13113019-64f7b2b6adeb4402aac7f3cc5c079f0c.png)

示例程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

pid\_t pid;

pid = fork();

if(pid < 0){

perror("fork error");

exit(EXIT\_FAILURE);

}

if(pid == 0){

printf("this is child process\n");

exit(100);

}

int status;

pid\_t ret;

ret = wait(&status);

if(ret <0){

perror("wait error");

exit(EXIT\_FAILURE);

}

printf("ret = %d pid = %d\n", ret, pid);

if (WIFEXITED(status))

printf("child exited normal exit status=%d\n", WEXITSTATUS(status));

else if (WIFSIGNALED(status))

printf("child exited abnormal signal number=%d\n", WTERMSIG(status));

else if (WIFSTOPPED(status))

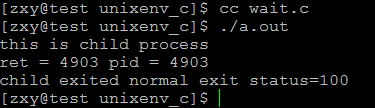
printf("child stoped signal number=%d\n", WSTOPSIG(status));

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/13113019-9f47113c9cd24867b055a15b42581229.png)

当子进程正常退出时wait返回子进程pid，且WIFEXITED(status)验证为真，可以WEXITSTATUS(status）获得返回状态码

示例2：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

pid\_t pid;

pid = fork();

if(pid < 0){

perror("fork error");

exit(EXIT\_FAILURE);

}

if(pid == 0){

printf("this is child process\n");

//exit(100);

abort();

}

int status;

pid\_t ret;

ret = wait(&status);

if(ret <0){

perror("wait error");

exit(EXIT\_FAILURE);

}

printf("ret = %d pid = %d\n", ret, pid);

if (WIFEXITED(status))

printf("child exited normal exit status=%d\n", WEXITSTATUS(status));

else if (WIFSIGNALED(status))

printf("child exited abnormal signal number=%d\n", WTERMSIG(status));

else if (WIFSTOPPED(status))

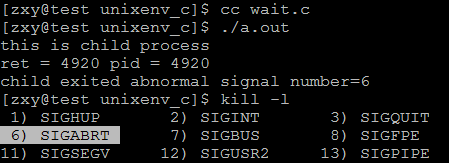
printf("child stoped signal number=%d\n", WSTOPSIG(status));

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/13113023-146111f0726046ae8d4a906d5ab7e67d.png)

当子进程异常退出时，WIFSIGNALED(status）为真，可用WTERMSIG(status)获得信号

### 四，waitpid()函数

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

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

参数:

status:如果不是空，会把状态信息写到它指向的位置，与wait一样

options：允许改变waitpid的行为，最有用的一个选项是WNOHANG,它的作用是防止waitpid把调用者的执行挂起

The value of options is an OR of zero or more  of  the  following  con-   
stants:

WNOHANG     return immediately if no child has exited.

WUNTRACED   also  return  if  a  child  has stopped (but not traced via   
            ptrace(2)).  Status for traced children which have  stopped   
            is provided even if this option is not specified.

WCONTINUED (since Linux 2.6.10)   
            also return if a stopped child has been resumed by delivery   
            of SIGCONT.

返回值：如果成功返回等待子进程的ID，失败返回-1

##### 对于waitpid的p i d参数的解释与其值有关：

pid == -1 等待任一子进程。于是在这一功能方面waitpid与wait等效。

pid > 0 等待其进程I D与p i d相等的子进程。

pid == 0 等待其组I D等于调用进程的组I D的任一子进程。换句话说是与调用者进程同在一个组的进程。

pid < -1 等待其组I D等于p i d的绝对值的任一子进程

##### wait与waitpid区别：

* 在一个子进程终止前， wait 使其调用者阻塞，而waitpid 有一选择项，可使调用者不阻塞。
* waitpid并不等待第一个终止的子进程—它有若干个选择项，可以控制它所等待的特定进程。
* 实际上wait函数是waitpid函数的一个特例。waitpid(-1, &status, 0);

示例：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

pid\_t pid;

pid = fork();

if(pid < 0){

perror("fork error");

exit(EXIT\_FAILURE);

}

if(pid == 0){

printf("this is child process\n");

sleep(5);

exit(100);

}

int status;

pid\_t ret;

ret = waitpid(pid,&status,WNOHANG);

if(ret <0){

perror("wait error");

exit(EXIT\_FAILURE);

}

printf("ret = %d pid = %d\n", ret, pid);

if (WIFEXITED(status))

printf("child exited normal exit status=%d\n", WEXITSTATUS(status));

else if (WIFSIGNALED(status))

printf("child exited abnormal signal number=%d\n", WTERMSIG(status));

else if (WIFSTOPPED(status))

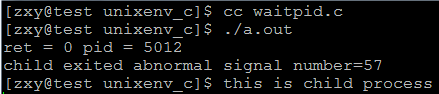
printf("child stoped signal number=%d\n", WSTOPSIG(status));

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/13113024-805acb66d6024072b4dd58401777037b.png)

可知，option设为WNOHANG，父进程不会等到子进程的退出，即不会阻塞，如果没有子进程退出则立即返回-1,

## [linux系统编程之进程（七）：system()函数使用](http://www.cnblogs.com/mickole/p/3187974.html)

#### 一，system()理解

功能：system()函数调用“/bin/sh -c command”执行特定的命令，阻塞当前进程直到command命令执行完毕

原型：

int system(const char \*command);

返回值：

如果无法启动shell运行命令，system将返回127；出现不能执行system调用的其他错误时返回-1。如果system能够顺利执行，返回那个命令的退出码。

说明：

man帮助：

       #include <stdlib.h>

       int system(const char \*command);

DESCRIPTION   
       system()  executes a command specified in command by calling /bin/sh -c   
       command, and returns after the command has been completed.  During exe-   
       cution  of the command, SIGCHLD will be blocked, and SIGINT and SIGQUIT   
       will be ignored.

RETURN VALUE   
       The value returned is -1 on  error  (e.g.   fork(2)  failed),  and  the   
       return  status  of the command otherwise.  This latter return status is   
       in the format specified in wait(2).  Thus, the exit code of the command   
       will  be  WEXITSTATUS(status).   In case /bin/sh could not be executed,   
       the exit status will be that of a command that does exit(127).

       If the value of command is NULL, system() returns non-zero if the shell   
       is available, and zero if not.

       system() does not affect the wait status of any other children.

#### 二，system()函数原理

system函数执行时，会调用fork、execve、waitpid等函数。

linux版system函数的源码：

[复制代码](javascript:void(0);)

int system(const char \* cmdstring)

{

pid\_t pid;

int status;

if(cmdstring == NULL){

return (1);

}

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

status = -1;

}

else if(pid == 0){

execl("/bin/sh", "sh", "-c", cmdstring, (char \*)0);

\_exit(127); //子进程正常执行则不会执行此语句

}

else{

while(waitpid(pid, &status, 0) < 0){

if(errno != EINTER){

status = -1;

break;

}

}

}

return status;

}

[复制代码](javascript:void(0);)

* 函数说明   
  system()会调用fork()产生子进程，由子进程来调用/bin/sh-c string来执行参数string字符串所代表的命令，此命>令执行完后随即返回原调用的进程。   
  在调用system()期间SIGCHLD 信号会被暂时搁置，SIGINT和SIGQUIT 信号则会被忽略。   
  返回值   
  =-1:出现错误    
  =0:调用成功但是没有出现子进程    
  >0:成功退出的子进程的id   
  如果system()在调用/bin/sh时失败则返回127，其他失败原因返回-1。若参数string为空指针(NULL)，则返回非零值>。如果system()调用成功则最后会返回   
  执行shell命令后的返回值，但是此返回值也有可能为 system()调用/bin/sh失败所返回的127，因此最好能再检查errno 来确认执行成功。   
  附加说明   
  在编写具有SUID/SGID权限的程序时请勿使用system()，system()会继承环境变量，通过环境变量可能会造成系统安全的问题。

##### system函数对返回值的处理，涉及3个阶段：

阶段1：创建子进程等准备工作。如果失败，返回-1。   
阶段2：调用/bin/sh拉起shell脚本，如果拉起失败或者shell未正常执行结束（参见备注1），原因值被写入到status的低8~15比特位中。system的man中只说明了会写了127这个值，但实测发现还会写126等值。   
阶段3：如果shell脚本正常执行结束，将shell返回值填到status的低8~15比特位中。   
备注1：   
只要能够调用到/bin/sh，并且执行shell过程中没有被其他信号异常中断，都算正常结束。   
比如：不管shell脚本中返回什么原因值，是0还是非0，都算正常执行结束。即使shell脚本不存在或没有执行权限，也都算正常执行结束。   
如果shell脚本执行过程中被强制kill掉等情况则算异常结束。

如何判断阶段2中，shell脚本子进程是否正常执行结束呢？系统提供了宏：WIFEXITED(status)。如果WIFEXITED(status)为真，则说明正常结束。   
如何取得阶段3中的shell返回值？你可以直接通过右移8bit来实现，但安全的做法是使用系统提供的宏：WEXITSTATUS(status)。

由于我们一般在shell脚本中会通过返回值判断本脚本是否正常执行，如果成功返回0，失败返回正数。   
所以综上，判断一个system函数调用shell脚本是否正常结束的方法应该是如下3个条件同时成立：   
（1）-1 != status   
（2）WIFEXITED(status)为真   
（3）0 == WEXITSTATUS(status)   
注意：   
根据以上分析，当shell脚本不存在、没有执行权限等场景下时，以上前2个条件仍会成立，此时WEXITSTATUS(status)为127，126等数值。   
所以，我们在shell脚本中不能将127，126等数值定义为返回值，否则无法区分中是shell的返回值，还是调用shell脚本异常的原因值。shell脚本中的返回值最好多1开始递增。

示例程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#define EXIT\_ERR(m) \

do\

{\

perror(m);\

exit(EXIT\_FAILURE);\

}\

while (0);\

int main(void)

{

int status ;

status = system("ls -l|wc -l");

if(status == -1){

EXIT\_ERR("system error");

}

else{

if(WIFEXITED(status))

{

if(WEXITSTATUS(status) == 0)

printf("run command successful\n");

else

printf("run command fail and exit code is %d\n",WEXITSTATUS(status));

}

else

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

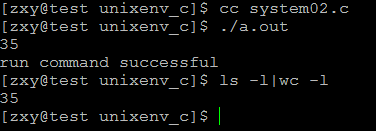
}

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/13135909-465054302e784a2c952b24965ed4ed4e.png)

## [linux系统编程之进程（八）：守护进程详解及创建，daemon()使用](http://www.cnblogs.com/mickole/p/3188321.html)

### 一，守护进程概述

Linux Daemon（守护进程）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。它不需要用户输入就能运行而且提供某种服务，不是对整个系统就是对某个用户程序提供服务。Linux系统的大多数服务器就是通过守护进程实现的。常见的守护进程包括系统日志进程syslogd、 web服务器httpd、邮件服务器sendmail和数据库服务器mysqld等。

守护进程一般在系统启动时开始运行，除非强行终止，否则直到系统关机都保持运行。守护进程经常以超级用户（root）权限运行，因为它们要使用特殊的端口（1-1024）或访问某些特殊的资源。

一个守护进程的父进程是init进程，因为它真正的父进程在fork出子进程后就先于子进程exit退出了，所以它是一个由init继承的孤儿进程。守护进程是非交互式程序，没有控制终端，所以任何输出，无论是向标准输出设备stdout还是标准出错设备stderr的输出都需要特殊处理。

守护进程的名称通常以d结尾，比如sshd、xinetd、crond等

### 二，创建守护进程步骤

首先我们要了解一些基本概念：

进程组 ：

* 每个进程也属于一个进程组
* 每个进程主都有一个进程组号，该号等于该进程组组长的PID号 .
* 一个进程只能为它自己或子进程设置进程组ID号

会话期：

会话期(session)是一个或多个进程组的集合。

setsid()函数可以建立一个对话期：

 如果，调用setsid的进程不是一个进程组的组长，此函数创建一个新的会话期。

(1)此进程变成该对话期的首进程

(2)此进程变成一个新进程组的组长进程。

(3)此进程没有控制终端，如果在调用setsid前，该进程有控制终端，那么与该终端的联系被解除。 如果该进程是一个进程组的组长，此函数返回错误。

(4)为了保证这一点，我们先调用fork()然后exit()，此时只有子进程在运行

现在我们来给出创建守护进程所需步骤：

编写守护进程的一般步骤步骤：

（1）在父进程中执行fork并exit推出；

（2）在子进程中调用setsid函数创建新的会话；

（3）在子进程中调用chdir函数，让根目录 ”/” 成为子进程的工作目录；

（4）在子进程中调用umask函数，设置进程的umask为0；

（5）在子进程中关闭任何不需要的文件描述符

说明：

1. 在后台运行。   
为避免挂起控制终端将Daemon放入后台执行。方法是在进程中调用fork使父进程终止，让Daemon在子进程中后台执行。   
if(pid=fork())   
exit(0);//是父进程，结束父进程，子进程继续   
2. 脱离控制终端，登录会话和进程组   
有必要先介绍一下Linux中的进程与控制终端，登录会话和进程组之间的关系：进程属于一个进程组，进程组号（GID）就是进程组长的进程号（PID）。登录会话可以包含多个进程组。这些进程组共享一个控制终端。这个控制终端通常是创建进程的登录终端。   
控制终端，登录会话和进程组通常是从父进程继承下来的。我们的目的就是要摆脱它们，使之不受它们的影响。方法是在第1点的基础上，调用setsid()使进程成为会话组长：   
setsid();   
说明：当进程是会话组长时setsid()调用失败。但第一点已经保证进程不是会话组长。setsid()调用成功后，进程成为新的会话组长和新的进程组长，并与原来的登录会话和进程组脱离。由于会话过程对控制终端的独占性，进程同时与控制终端脱离。   
3. 禁止进程重新打开控制终端   
现在，进程已经成为无终端的会话组长。但它可以重新申请打开一个控制终端。可以通过使进程不再成为会话组长来禁止进程重新打开控制终端：   
if(pid=fork())   
exit(0);//结束第一子进程，第二子进程继续（第二子进程不再是会话组长）   
4. 关闭打开的文件描述符   
进程从创建它的父进程那里继承了打开的文件描述符。如不关闭，将会浪费系统资源，造成进程所在的文件系统无法卸下以及引起无法预料的错误。按如下方法关闭它们：   
for(i=0;i 关闭打开的文件描述符close(i);>   
5. 改变当前工作目录   
进程活动时，其工作目录所在的文件系统不能卸下。一般需要将工作目录改变到根目录。对于需要转储核心，写运行日志的进程将工作目录改变到特定目录如/tmpchdir("/")   
6. 重设文件创建掩模   
进程从创建它的父进程那里继承了文件创建掩模。它可能修改守护进程所创建的文件的存取位。为防止这一点，将文件创建掩模清除：umask(0);   
7. 处理SIGCHLD信号   
处理SIGCHLD信号并不是必须的。但对于某些进程，特别是服务器进程往往在请求到来时生成子进程处理请求。如果父进程不等待子进程结束，子进程将成为僵尸进程（zombie）从而占用系统资源。如果父进程等待子进程结束，将增加父进程的负担，影响服务器进程的并发性能。在Linux下可以简单地将SIGCHLD信号的操作设为SIG\_IGN。   
signal(SIGCHLD,SIG\_IGN);   
这样，内核在子进程结束时不会产生僵尸进程。这一点与BSD4不同，BSD4下必须显式等待子进程结束才能释放僵尸进程。

### 三，创建守护进程

在创建之前我们先了解setsid()使用：

  #include <unistd.h>

       pid\_t setsid(void);

DESCRIPTION   
       setsid()  creates a new session if the calling process is not a process   
       group leader.  The calling process is the leader of  the  new  session,   
       the  process group leader of the new process group, and has no control-   
       ling tty.  The process group ID and session ID of the  calling  process   
       are set to the PID of the calling process.  The calling process will be   
       the only process in this new process group and in this new session.

//调用进程必须是非当前进程组组长，调用后，产生一个新的会话期，且该会话期中只有一个进程组，且该进程组组长为调用进程，没有控制终端，新产生的group ID 和 session ID 被设置成调用进程的PID

RETURN VALUE   
       On success, the (new) session ID of the calling  process  is  returned.   
       On  error,  (pid\_t) -1  is  returned,  and errno is set to indicate the   
       error.

现在根据上述步骤创建一个守护进程：

以下程序是创建一个守护进程，然后利用这个守护进程每个一分钟向daemon.log文件中写入当前时间

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <time.h>

#include <fcntl.h>

#include <string.h>

#include <sys/stat.h>

#define ERR\_EXIT(m) \

do\

{\

perror(m);\

exit(EXIT\_FAILURE);\

}\

while (0);\

void creat\_daemon(void);

int main(void)

{

time\_t t;

int fd;

creat\_daemon();

while(1){

fd = open("daemon.log",O\_WRONLY|O\_CREAT|O\_APPEND,0644);

if(fd == -1)

ERR\_EXIT("open error");

t = time(0);

char \*buf = asctime(localtime(&t));

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

close(fd);

sleep(60);

}

return 0;

}

void creat\_daemon(void)

{

pid\_t pid;

pid = fork();

if( pid == -1)

ERR\_EXIT("fork error");

if(pid > 0 )

exit(EXIT\_SUCCESS);

if(setsid() == -1)

ERR\_EXIT("SETSID ERROR");

chdir("/");

int i;

for( i = 0; i < 3; ++i)

{

close(i);

open("/dev/null", O\_RDWR);

dup(0);

dup(0);

}

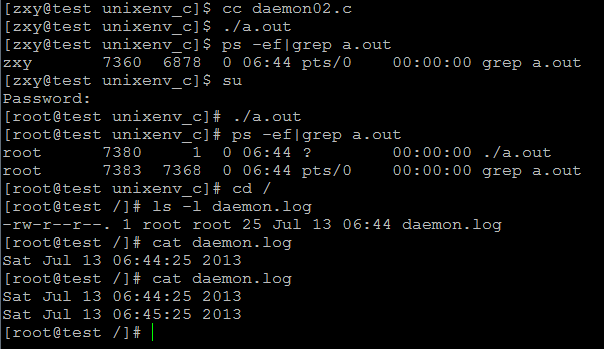
umask(0);

return;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/13191708-0a54178e53cb4b4b9beaf0962d2415fb.png)

结果显示：当我一普通用户执行a.out时，进程表中并没有出现新创建的守护进程，但当我以root用户执行时，成功了，并在/目录下创建了daemon.log文件，cat查看后确实每个一分钟写入一次。为什么只能root执行，那是因为当我们创建守护进程时，已经将当前目录切换我/目录，所以当我之后创建daemon.log文件是其实是在/目录下，那肯定不行，因为普通用户没有权限，或许你会问那为啥没报错呢？其实是有出错，只不过我们在创建守护进程时已经将标准输入关闭并重定向到/dev/null，所以看不到错误信息。

### 四，利用库函数daemon()创建守护进程

其实我们完全可以利用daemon()函数创建守护进程，其函数原型：

#include <unistd.h>

int daemon(int nochdir, int noclose);

DESCRIPTION   
       The daemon() function is for programs wishing to detach themselves from   
       the controlling terminal and run in the background as system daemons.

       If nochdir is zero, daemon()  changes  the  process’s  current  working   
       directory to the root directory ("/"); otherwise,

       If  noclose is zero, daemon() redirects standard input, standard output   
       and standard error to /dev/null; otherwise,  no  changes  are  made  to   
       these file descriptors.

功能：创建一个守护进程

参数：

nochdir：=0将当前目录更改至“/”

noclose：=0将标准输入、标准输出、标准错误重定向至“/dev/null”

返回值：

成功：0

失败：-1

现在我们利用daemon()改写刚才那个程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <time.h>

#include <fcntl.h>

#include <string.h>

#include <sys/stat.h>

#define ERR\_EXIT(m) \

do\

{\

perror(m);\

exit(EXIT\_FAILURE);\

}\

while (0);\

void creat\_daemon(void);

int main(void)

{

time\_t t;

int fd;

if(daemon(0,0) == -1)

ERR\_EXIT("daemon error");

while(1){

fd = open("daemon.log",O\_WRONLY|O\_CREAT|O\_APPEND,0644);

if(fd == -1)

ERR\_EXIT("open error");

t = time(0);

char \*buf = asctime(localtime(&t));

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

close(fd);

sleep(60);

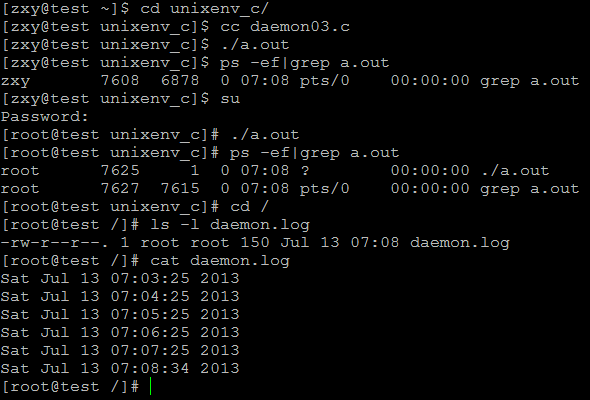
}

return 0;

}

[复制代码](javascript:void(0);)

当daemon(0,0)时：

[](http://images.cnitblog.com/blog/529981/201307/13191708-3051ebbcea5b40fd96c28062dd4e79a9.png)

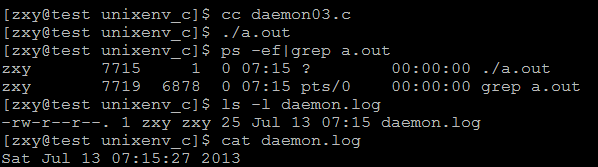
结果同刚才一样，也是只有root才能成功，普通用户执行时看不到错误信息

现在让daemon(0,1)，就是不关闭标准输入输出结果：

[QQ截图20130713190932](http://images.cnitblog.com/blog/529981/201307/13191709-480e7b8c09c74204aabec86e634cb63b.png)

可以看到错误信息

现在让daemon(1,0),就是不重定向，结果如下：

[](http://images.cnitblog.com/blog/529981/201307/13191710-365931fa46084265a6eda275929efe0d.png)

这次普通用户执行成功了，以为没有切换到/目录下，有权限

其实我们可以利用我们刚才写的创建守护进程程序默认daemon()实现：

代码如下：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <time.h>

#include <fcntl.h>

#include <string.h>

#include <sys/stat.h>

#define ERR\_EXIT(m) \

do\

{\

perror(m);\

exit(EXIT\_FAILURE);\

}\

while (0);\

void creat\_daemon(int nochdir, int noclose);

int main(void)

{

time\_t t;

int fd;

creat\_daemon(0,0);

while(1){

fd = open("daemon.log",O\_WRONLY|O\_CREAT|O\_APPEND,0644);

if(fd == -1)

ERR\_EXIT("open error");

t = time(0);

char \*buf = asctime(localtime(&t));

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

close(fd);

sleep(60);

}

return 0;

}

void creat\_daemon(int nochdir, int noclose)

{

pid\_t pid;

pid = fork();

if( pid == -1)

ERR\_EXIT("fork error");

if(pid > 0 )

exit(EXIT\_SUCCESS);

if(setsid() == -1)

ERR\_EXIT("SETSID ERROR");

if(nochdir == 0)

chdir("/");

if(noclose == 0){

int i;

for( i = 0; i < 3; ++i)

{

close(i);

open("/dev/null", O\_RDWR);

dup(0);

dup(0);

}

umask(0);

return;

}

[复制代码](javascript:void(0);)