进程

学习目标

理解进程概念
复制进程 fork
替换进程映像 exec
守护进程

linux 进程概述

程序与进程

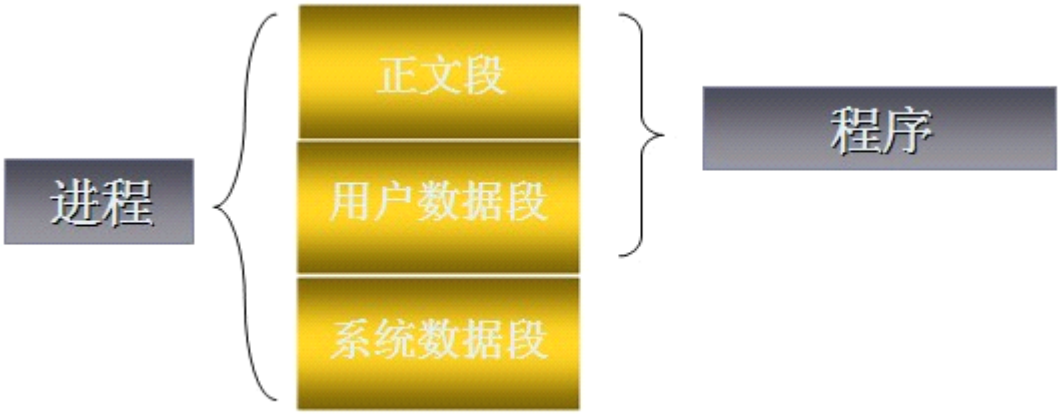
程序是一个普通文件，是机器代码指令和数据的集合，这些指令和数据存储在磁盘上的一个可执行映像（Executable Image）中。

- 进程是一个独立的可调度的任务
  - 进程是一个抽象实体。当系统在执行某个程序时，分配和释放的各种资源

- 进程是一个程序的一次执行的过程

- 进程和程序的区别
  - 程序是静态的，它是一些保存在磁盘上的指令的有序集合，没有任何执行的概念
  - 进程是一个动态的概念，它是程序执行的过程，包括创建、调度和消亡

- 进程是程序执行和资源管理的最小单位



进程不仅包括程序的指令和数据，而且包括程序计数器值、CPU 的所有寄存器值以及存储临时数据的进程堆栈。

进程结构

在 linux 系统中，每一个进程都是拥有自己的虚拟地址空间，都运行在独立的虚拟地址空间上。这也就是说，进程间是分离的任务，拥有各自的权利和责任。在 linux 系统中运行着多个进程，其中一个进程发生异常，它不会影响到系统中的其它进程。

Linux 中的进程包括了 3 个段，分别为“数据段”、“代码段”和“堆栈段”。

数据段：存放的数据为全局变量、常数及动态数据分配的数据空间（如 malloc 函数取得的空间）等

代码段：存放的是程序代码数据。

堆栈段：存入的是子程序返回地址、子程序的参数以及程序的局部变量。

补充知识：

一个由 c/C++编译的程序占用的内存分为以下几个部分

1、栈区（stack）—— 由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2、堆区（heap）—— 一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

3、全局区（静态区）（static）—— 全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。（程序结束后由系统释放）

4、文字常量区—— 常量字符串就是放在这里的。（程序结束后由系统释放）

5、程序代码区—— 存放函数体的二进制代码。

见例子程序：

```
//main.cpp

int a = 0; //全局初始化区
```

```
char *p1; //全局未初始化区
main()
{
int b; //栈
char s[] = "abc"; //栈
char *p2; //栈
char *p3 = "123456"; //123456\0 在常量区，p3 在栈上。
static int c =0;  //全局（静态）初始化区
p1 = (char *)malloc(10);
p2 = (char *)malloc(20);
//分配得来得 10 和 20 字节的区域就在堆区。
strcpy(p1, "123456"); /*123456\0 放在常量区，编译器可能会将
它与 p3 所指向的"123456"优化成一个地方。*/
}
```

进程属性

1. 进程标识

进程最主要的属性就是进程号（PID,process ID）和它的父进程号(PPID,parent process ID).PID 和 PPID 都是非零正整数。从进程 ID 的名字就可以看出，它就是进程的身份证号码，每个人的身份证号码都不会相同，每个进程的进程 ID 也不会相同。系统调用 getpid()就是获得进程标识符。

一个 PID 唯一地标识一个进程。一个进程创建一个新进程称为创建了子进程，创建子进程的进程称为父进程。

在 Linux 中获得当前进程的 PID 和 PPID 的系统调用函数为 getpid()和 getppid()。常常在程序获得进程的 PID 和 PPID 后，可以将其写入日志文件以做备份。运用 getpid 和 getppid 获得当进程 PID 和 PPID 的例子。

```
/*pidandppid.c*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
printf("PID = %d\n", getpid());
```

```
printf("PPID = %d\n", getppid());
exit(0);
}
```

通过编译后，运行程序得到以下结果（该值在不同的系统上会有不同的值）

```
[root@localhost process]#./pidandppid
PID=18985
PPID=14481
```

进程中还有真实用户 ID（UID）和有效的用户 ID（EUID）。进程的 UID 就是其创建者的用户标识号，或者说就是复制了父进程的 UID 值，通常，只允许创建者(也称为属主)和超级用户对进程进行操作。EUID 是“有效"的用户 ID，这是一个额外的 UID，用来确定进程在任何给定的时刻对哪些资源和文件具有访问权限。它们的函数分别是 getuid 和 geteuid.

2. 进程状态

为了对进程从产生到消亡的这个动态变化过程进行捕获和描述，就需要定义进程各种状态并制定相应的状态转换策略，以此来控制进程的运行。

因为不同操作系统对进程的管理方式和对进程的状态解释可以不同，所以不同操作系统中描述进程状态的数量和命名也会有所不同，但最基本的进程状态有三种：

（1）运行态:进程占有 CPU，并在 CPU 上运行。

（2）就绪态:进程已经具备运行条件,但由于 CPU 忙而暂时不能运行

（3）阻塞态（或等待态）:进程因等待某种事件的发生而暂时不能运行。(即使 CPU 空闲,进程也不可运行)。

进程在生命期内处于且仅处于三种基本状态之一,如图 6_1 所示。

这三种状态之间有四种可能的转换关系：

（1）运行态→阻塞态：进程发现它不能运行下去时发生这种转换。这是因为进程发生 I/O 请求或等待某件事情。

（2）运行态→就绪态：在系统认为运行进程占用 CPU 的时间已经过长,决定让其它进程占用 CPU 时发生这种转换。这是由调度程序引起的。调度程序是操作系统的一部分，进程甚至感觉不到它的存在。

（3）就绪态→运行态：运行进程已经用完分给它的 CPU 时间，调度程序从处于就绪态的进程中选择一个投入运行。

（4）阻塞态→就绪态：当一个进程等待的一个外部事件发生时（例如输入数据到达），则发生这种转换。如果这时没有其它进程运行，则转换③立即被触发，该进程便开始运行。

调度程序的主要工作是决定哪个进程应当运行，以及它应当运行多长时间。这点很重要，我们将在后面对其进行讨论。

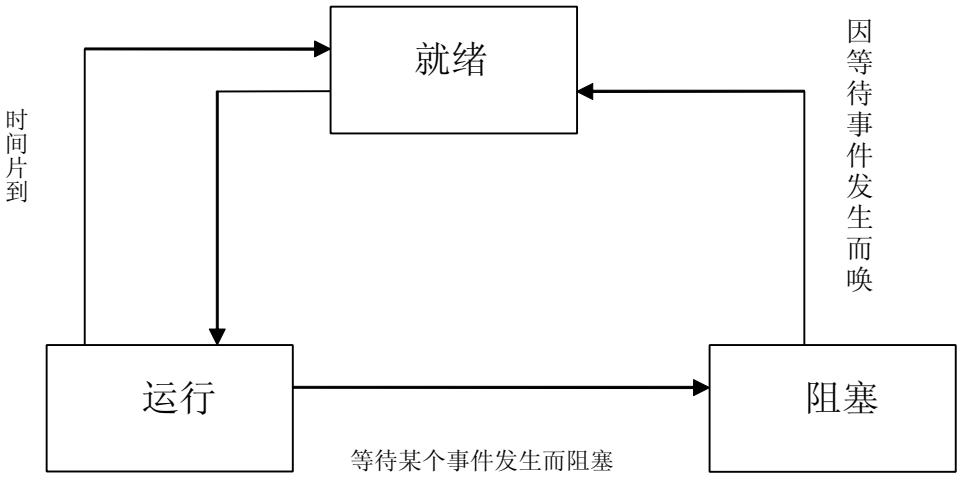


图 6_1

进程管理

1. 启动进程

进程加载有两种途径：手工加载和调度加载

手工加载：

手工加载又分为前台加载，和后台加载。

前台加载：是手工加载一个进程最常用方式。一般地，当用户输入一个命令，如“ls -l”时就已经产生了一个进程，并且是一个前台进程。

后台加载：往往是在该进程非常耗时，且用户也不急着需要结果的时候启动。常常用户在终端输入一个命令时同时在命令尾加上一个“&”符号。

调度加载：

在系统中有时要进行比较费时而且占用资源的维护工作，并且这些工作适合在深夜而无人值守时运行，这时用户就可以事先进行调度安排，指定任务运行的时间或者场合，到时系统就会自动完成一切任务。

2. 调度进程

调度进程包括对进程的中断操作，改变优先级，查看进程状态等，linux 中常见的调度进程的系统命令。如表 6_1 所示：

表 6_1 常见的调度进程的系统命令

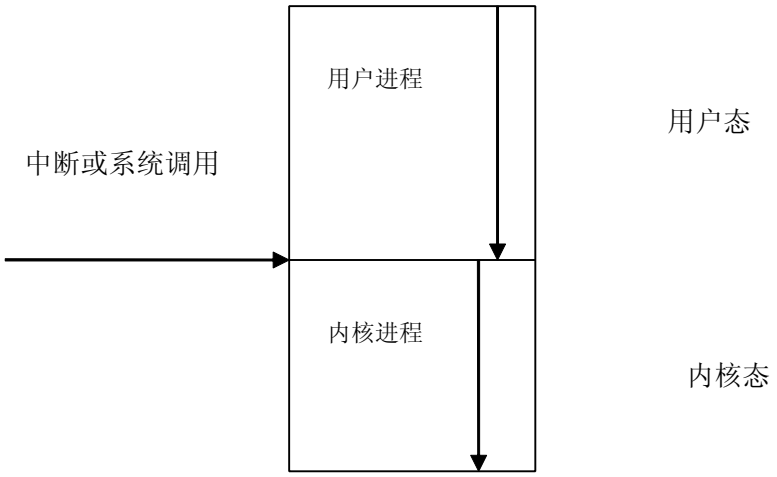
选 项	参 数 含 义
ps	查看系统中的进程
top	动态显示系统中的进程
nice	按用户指定的优先级运行
renice	改变正在运行进程的优先级
kill	终止进程（包括后台进程）
crontab	用于安装、删除或者列出用于驱动 cron 后台进程的任务。

选　项	参　数　含　义
bg	将挂起的进程放到后台执行

进程模式

在 Linux 系统中，进程的执行模式划分为用户模式和内核模式。如果当前运行的是用户程序、应用程序或内核之外的系统程序，那么对应进程就在用户模式下运行：如果在用户程序执行过程中出现系统调用或发生中断事件，那么就要运行操作系统程序，进程模式就变成内核模式。

用户进程既可以在用户模式下运行，也可以在内核模式下运行。如下图所示：



linux 进程控制

在 Linux 系统中，常用于进程控制的函数有 fork()函数、exec()函数族、exit()和 wait()函数等。

fork 函数

fork 函数和一般的函数有着很大区别，fork 函数执行一次却返回两个值。

1. fork 函数说明

在进程中使用 fork 函数，则会创建一个新进程，新进程则称为子进程，原进程称为父进程。由于 fork 函数返回两个值，则这两个进程分别带回它们各自的返回值，其中父进程的返回值是子进程的进程号，而子进程则返回 0.因此，可以通过返回值来判定该进程是父进程还是子进程。

使用 fork 函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间，包括进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设定、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，而子进程所独有的

只有它的进程号、资源使用和计时器等。因此可以看出，使用 fork 函数的代价是很大的，它复制了父进程中的代码段、数据段和堆栈段里的大部分内容，使得 fork 函数的执行速度并不很快。

2. fork 函数语法

Fork（）函数语法要点如表 6_2 所示

表 6_2 Fork（）函数语法要点

所需头文件	#include <sys/types.h> //提供类型 pid_t 的定义 #include <unistd.h>
函数原型	pid_t fork(void)
函数返回值	0：子进程
	子进程 ID（大于 0 的整数）：父进程
	-1：出错

3. fork 函数实例

```
/*fork.c*/
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
pid_t result;
result=fork();
if(result==-1)
{
perror("fork error");
}
else if(result==0)
{
printf("current value is %d In child process,child PID =
%d\n",result,getpid());
}
else
{
printf("current value is %d In father process,father
PID=%d\n",result,getpid());
}
}
```

编译：

```
[root@localhost fork]# gcc -o fork fork.c
```

运行的结果：

```
[root@localhost fork]#./fork
current value is 0 In child process,child PID = 21273
current value is 21273 In father process,father PID=21272
```

从结果可以看出，子进程返回值等于 0,而父进程返回子进程的进程号（>0）.

函数族

1. exec 函数族说明

fork()函数是用于创建一个子进程，该子进程几乎拷贝了父进程的全部内容。那在新进程中如何运行新的程序呢？exec 函数簇提供了一个在进程中启动另一个程序执行的方法。

exec 函数簇可以根据指定的文件名或目录找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新进程替换了。另外，这里的可执行文件既可以是二进制文件，也可以是 Linux 下任何可执行的脚本文件。

2. exec 函数族语法

在 linux 中并没有 exec 函数，而是有 6 个 exec 开头的函数族，它们之音语法有细微差别，本书在下面会详细讲解。

表 6_3　exec（）函数族成员函数语法

所需头文件	#include <unistd.h>
函数原型	int execl(const char *path, const char *arg, ...)
	int execv(const char *path, char *const argv[])
	int execle(const char *path, const char *arg, ..., char *const envp[])
	int execve(const char *path, char *const argv[], char *const envp[])
	int execlp(const char *file, const char *arg, ...)
	int execvp(const char *file, char *const argv[])
函数返回值	-1：出错

下表 7.4 再对这几个函数中函数名和对应语法做一总结，主要指出了函数名中每一位所表明的含义，希望读者结合此表加以记忆。

表 6_4　exec（）函数名对应含义

前 4 位统一为	exec	
第 5 位	l：参数传递为逐个列举方式	execl、execle、execlp
	v：参数传递为构造指针数组方式	execv、execve、execvp
第 6 位	e：可传递新进程环境变量	execle、execve
	p：可执行文件查找方式为文件名	execlp、execvp

3. exec 函数族实例

1 execlp

用 execlp 函数作实例，说明如何使用文件名的方式来查找可执行文件，同时使用参数列表的方式。

```
/*execlp.c*/
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
pid_t result;
result=fork();
if(result==0)
{
if(execlp("ls","ls","-l",NULL)<0)
{
perror("execlp error");
}
}
}
```

在该程序中，首先使用创建一个子进程，然后在程序里使用 execlp 函数。这里 execlp 使用文件名的方式进行查找，系统会在默认的环境变量 PATH 中寻找该可执行文件，参数列表是在 shell 中使用的命令和选项。参数列表最后一项应为 NULL.运行程序后，运行结果为列出当前目录下所有文件，不同的目录下有不同的结果。例如：

```
[root@localhost exec]# ./execlp
总计 12
-rwxr-xr-x   1   root  root 4954 07-05 09:48  execlp
-rw-r—r--   1   root   root 207  07-05 09:48  execlp.c
```

2 用 execl 函数作实例，说明如何使用完整的文件目录来查找对应的可执行文件（注意目录必须以“/”开头，否则将其视为文件名），同时使用参数列表的方式。

```
/*execl.c*/
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
pid_t result;
result=fork();
if(result==0)
{
if(execl("/bin/ls","ls","-l",NULL)<0)
```

```
{
perror("execlp error");
}
}
}
```

编译后，运行所得结果和运行 exelp 函数的例子一样。

3 execv

参数传递为构造指针数组方式。

```
/*execv.c*/
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
pid_t result;
char *arg[]={"ls","-l",NULL};
result=fork();
if(result==0)
{
if(execv("/bin/ls",arg)<0)
{
perror("execlp error");
}
}
}
```

运行结果为列出当前目录下的所有文件。

4 execve

e 为设置环境变量。

```
/*execv.c*/
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
pid_t result;
char *arg[]={“env”,NULL};
char *envp[]={“PATH=/tmp”,”USER=lzg”,NULL};
result = fork();
if(result==0)
{
if(execve(“/bin/env”,arg,envp)<0)
{
```

```
Perror("execlp error");
}
}
}
```

编译运行得到结果为

```
[root@localhost exec]# ./execve
PATH=/tmp
USER=lzg
```

(4) 使用 exec 函数族应注意的地方

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

找不到文件或路径，此时 errno 被设置为 ENOENT；

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

没有对应可执行文件的运行权限，此时 errno 被设置为 EACCES。

小知识：事实上，这 6 个函数中真正的系统调用只有 execve（），其他 5 个都是库函数，它们最终都会调用 execve（）这个系统调用。

补充知识：system 函数

（1）system 函数说明

system 函数是一个与操作系统紧密相关的函数。用户可以使用它在自己的程序中调用系统提供的各种命令。因此，使用 system 函数比使用 exec 函数族更方便。

（2）system 函数说明

表 6_5

所需头文件	#include<stdlib.h>
函数原型	int system(const char *string)
函数返回值	执行成功则返回执行 shell 命令后的返回值，调用/bin/sh 失败则返回 127，其它失败原因则返回-1，参数 string 为空(NULL),则返回非零值。

（3）system 函数实例

```
/*system.c*/
#include<stdio.h>
#include<stdlib.h>
int main()
{
int result;
result=system("ls -l");
return 0;
}
```

运行结果是列出当前目录下的所有文件。在不同的目录下运行程序会有不同的结果。

exit 和_exit 函数

1. exit 和_exit 函数说明

在系统中有大量的进程时，有可能会让系统资源消耗殆尽。因此，要在用完进程后终止进程。Linux 用到的函数为 exit 和_exit 函数。当程序执行 exit 和_exit 时，进程会条件地停止所有操作，终止本进程的运行。这两个数的区别如图 6_2 所示：

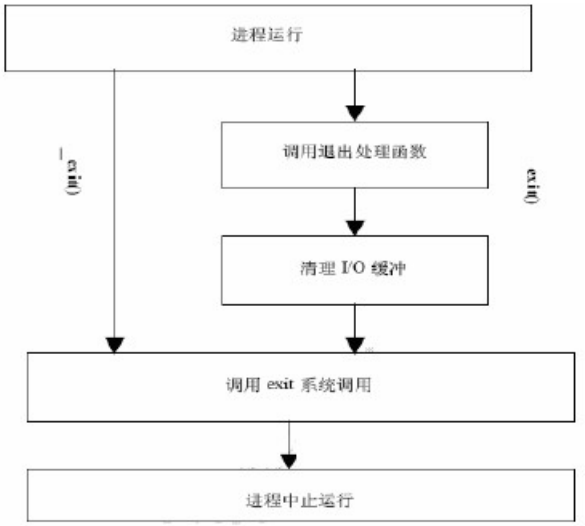


图 6_2

从图中可以看出，_exit 函数的作用是：直接使进程停止运行，清除其使用内存空间，并清除其在内核中的各种数据结构；exit 函数则在这些基础上作了一些动作，在执行退出之前加了若干道工序。Exit 函数和 _exit 函数最大的区别就在于 exit 函数在调用 exit 系统调用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。就是图中的“清理 I/0 缓冲”。

2. exit 和_exit 函数语法

表 6_6 exit 和_exit 函数语法

所需头文件	exit：#include <stdlib.h>
	_exit：#include <unistd.h>
函数原型	exit：void exit(int status)
	_exit：void _exit(int status)
函数传入值	status 是一个整型的参数，可以利用这个参数传递进程结束时的状态。一般来说，0 表示正常结束；其他的数值表示出现了错误，进程非正常结束。 在实际编程时，可以用 wait 系统调用接收子进程的返回值，从而针对不同的情况进行不同的处理

3. exit 和_exit 函数实例

print 函数使用的是缓冲 I/0 方式，该函数在遇到“\n”换行符时自动从缓冲区中将记录读出。以下实例就是利用此性质来进行比较。

```
/*exit.c*/
```

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
int main()
{
pid_t result;
result=fork();
if(result==1)

{
perror("fork fail");
exit(0);
}

else if(result==0)
{
printf("testing _exit()\n");
printf("this is the content in buffer");
_exit(0);
}

else
{
printf("testing exit()\n");
printf("this is the content in buffer");
exit(0);
}

}
```

运行结果为：

```
[root@localhost exit]#./exit
testing _exit()
testing exit()
this is the content in buffer
```

从实例结果来看，exit 函数前的字符串输出了两句，_exit()函数前的字符串输出一句。这也说明调用 exit 函数时，缓冲区的记录能正常输出；而调用_exit 函数时，缓冲区中的记录无法输出。

wait 和 waitpid 函数

1. wait 和 waitpid 函数说明

wait 函数是用于使父进程阻塞，直到一个子进程终止或者该进程接到了一个指定的信号

为止。如果该父进程没有子进程或者他的子进程已经终止，则 wait 就会立即返回。

waitpid 函数的作用和 wait 一样，但它并不一定要等待第一个终止的子进程，它还有若干项，如可提供一个非阻塞版本的 wait 功能。实际上 wait 函数只是 waitpid 函数的一个特例。

2. wait 和 waitpid 函数格式说明

表 6_7 wait()函数语法

所需头文件	#include <sys/types.h> #include <sys/wait.h>
函数原型	pid_t wait(int *status)
函数传入值	这里的 status 是一个整型指针，是该子进程退出时的状态； status 若为空，则代表任意状态结束的子进程； status 若不为空，则代表指定状态结束的子进程； 另外，子进程的结束状态可由 Linux 中一些特定的宏来测定。
函数返回值	成功：已结束运行的子进程的进程号
	失败：-1

表 6_8 waitpid()函数语法

所需头文件	#include <sys/types.h> #include <sys/wait.h>	
函数原型	pid_t waitpid(pid_t pid, int *status, int options)	
函数传入值	pid	pid>0：只等待进程 ID 等于 pid 的子进程，不管已经有其他子进程运行结束退出了，只要指定的子进程还没有结束，waitpid 就会一直等下去
		pid=1：等待任何一个子进程退出，此时和 wait 作用一样
		pid=0：等待其组 ID 等于调用进程的组 ID 的任一子进程
		pid<-1：等待其组 ID 等于 pid 的绝对值的任一子进程
	status	同 wait
	options	WNOHANG：若由 pid 指定的子进程不立即可用，则 waitpid 不阻塞，此时返回值为 0
		WUNTRACED：若实现某支持作业控制，则由 pid 指定的任一子进程状态已暂停，且其状态自暂停以来还未报告过，则返回其状态
		0：同 wait，阻塞父进程，等待子进程退出
函数返回值	正常：子进程的进程号 使用选项 WNOHANG 且没有子进程退出：0 调用出错：-1	

3. waitpid 函数实例

wait 函数只是 waitpid 函数的一个特例,所以这里只举使用 waitpid 函数的实例。

```
/*waitpid.c*/
```

```
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
pid_t result1,result2;
result1=fork();（创建新进程）
if(result1<0)
printf("fork fail\n");
else if(result1==0)
{
printf("sleep 3s in child\n ");
sleep(3); （子进程暂停 3s）
exit(0);
}
else
{
 /*不断地在测试子进程是否退出*/
do
{
result2=waitpid(result1,NULL,WNOHANG);
if(result2==0)（如果子进程没有退出就返回值为 0）
{
printf("The child process has not exited\n");
sleep(1);
}

}while(result2==0);
if(result1==result2)
printf("The child process has exited\n");
}

}
```

编译运行结果为

```
[root@localhost waitpid]./waitpid
sleep 3s in child
The child process has not exited
The child process has not exited
The child process has not exited
The child process has exited
```

通过运行结果可知道父进程在没有捕获到子进程的退出信号，就会不断地循环，直到子

进程退出为止，子进程不退出,waitpid 返回值为 0。

守护进程

守护进程概述

守护进程（Daemon）是 Linux 中的后台服务进程。它是一个生存期较长的进程，通常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。守护进程常常在系统引导装入时启动，在系统关闭时终止。Linux 系统有很多守护进程，大多数服务都是通过守护进程实现的，同时，守护进程还能完成许多系统任务，例如，作业规划进程 crond、打印进程 lqd 等（这里的结尾字母 d 就是 Daemon 的意思）。

由于在 Linux 中，每一个系统与用户进行交流的界面称为终端，每一个从此终端开始运行的进程都会依附于这个终端，这个终端就称为这些进程的控制终端，当控制终端被关闭时，相应的进程都会自动关闭。但是守护进程却能够突破这种限制，它从被执行开始运转，直到整个系统关闭时才退出。如果想让某个进程不因为用户或终端或其他地变化而受到影响，那么就必须把这个进程变成一个守护进程。