LINUX 进程控制

1. Linux 进程概述

进程是一个程序一次执行的过程，它和程序有本质区别。程序是静态的，它是一些保存在磁盘上的指令的有序集合；而进程是一个动态的概念，它是一个运行着的程序，包含了进程的动态创建、调度和消亡的过程，是 Linux 的基本调度单位。那么从系统的角度看如何描述并表示它的变化呢？在这里，是通过进程控制块（PCB）来描述的。进程控制块包含了进程的描述信息、控制信息以及资源信息，它是进程的一个静态描述。

内核使用进程来控制对 CPU 和其他系统资源的访问，并且使用进程来决定在 CPU 上运行哪个程序，运行多久以及采用什么特性运行它。调度进程使用 CPU 的默认模型是循环时间共享。内核的调度器负责在所有的进程间分配 CPU 执行时间，每个进程轮流使用 CPU 一段时间，这段时间称为时间片(time slice)，它轮流在每个进程分得的时间片用完后从进程那里抢回控制权。

循环时间共享满足了交互式多任务系统的两个重要需求。

- 公平性：每个进程都有机会用到 CPU。
- 响应度：一个进程在使用 CPU 之前无需等待太长时间。

1.1. 进程标识

OS 会为每个进程分配一个唯一的整型 ID，做为进程的**标识号(pid)**。进程除了自身的 ID 外，还有**父进程 ID(ppid)**，所有进程的祖先进程是同一个进程，它叫做 **init 进程**，ID 为 1，init 进程是内核自举后的一个启动的进程。init 进程负责引导系统、启动守护（后台）进程并且运行必要的程序。

进程的 pid 和 ppid 可以分别通过函数 **getpid()**和 **getppid()**获得。

示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
        printf("pid:%d      ppid:%d\n",getpid(),getppid());
        return 0;
}
```

1.2. 进程的用户 ID 与组 ID(进程的运行身份)

进程在运行过程中，必须具有一类似于用户的身份，以便进行进程的权限控制，缺省情况下，哪个登录用户运行程序，该程序进程就具有该用户的身份。例如，假设当前登录用户为 gotter，他运行了 ls 程序，则 ls 在运行过程中就具有 gotter 的身份，该 ls 进程的用户 ID 和组 ID 分别为 gotter 和 gotter 所属的组。这类型的 ID 叫做进程的真实用户 ID 和真实组 ID。真实用户 ID 和真实组 ID 可以通过函数 **getuid()**和 **getgid()**获得。

与真实 ID 对应，进程还具有有效用户 ID 和有效组 ID 的属性，内核对进程的访问权限检查时，它检查的是进程的有效用户 ID 和有效组 ID，而不是真实用户 ID 和真实组 ID。缺省情况下，用户的（有效用户 ID 和有效组 ID）与(真实用户 ID 和真实组 ID)是相同的。有效用户 id 和有效组 id 通过函数

geteuid()和 getegid()获得。

示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
        printf("uid:%d gid:%d euid:%d
egid:%d\n",getuid(),getgid(),geteuid(),getegid());
        return 0;
}
```

```
shell>id
uid=500(ghaha) gid=500(ghaha) groups=500(ghaha)
```

编译生成可执行文件 a.out，程序文件的属性可能为：

```
-rwxrwxr-x    1 ghaha    ghaha       12132 Oct  7 09:26 a.out
```

执行结果可能为：

```
shell>./a.out
uid:500 gid:500 euid:500 egid:500
```

现在将 a.out 的所有者可执行属性改为 s

```
shell>chmod u+s a.out
shell>ll
-rwsrwxr-x    1 ghaha    ghaha       12132 Oct  7 09:26 a.out
```

此时改另外一个用户 gotter 登录并运行程序 a.out

```
shell>id
uid=502(gotter) gid=502(gotter) groups=502(gotter)
shell>./a.out
uid:502 gid:502 euid:500 egid:502
```

可以看到，进程的有效用户身份变为了 ghaha,而不是 gotter 了，这是因为文件 a.out 的访问权限的所有者可执行为设置了 s 的属性，设置了该属性以后，用户运行 a.out 时,a.out 进程的有效用户身份将不再是运行 a.out 的用户，而是 a.out 文件的所有者。

s 权限最常见的例子是

/usr/bin/passwd 程序，它的权限位为

```
shell>ll /usr/bin/passwd
-r-s--x--x    1 root     root        16336 Feb 13  2003 /usr/bin/passwd
```

我们知道，用户的用户名和密码是保存在/etc/passwd（后来专门将密码保存在/etc/shadow，它是根据/etc/passwd 文件来生成/etc/shadow 的，它把所有口令从/etc/passwd 中移到了/etc/shadow 中。这里用到的是影子口令，它将口令文件分成两部分：/etc/passwd 和/etc/shadow，此时/etc/shadow 就是影子口令文件，它保存的是加密的口令，而/etc/passwd 中的密码全部变成 x）下的。通过 ls －l 查看/etc/passwd 这个文件，你会发现，这个文件普通用户都没有可写的权限，那我们执行 passwd 的时候确实能够修改密码，那么这是怎么回事呢？也就是说，任何一个用户运行该程序时，该程序的有效身份都将是 root（用普通身份去执行这个操作的时候，它会暂时得到文件拥有者 root 的权限）,而这样 passwd 程序才有权限读取/etc/passwd 文件的信息。

我们也来实现以下 passwd 的功能，实现步骤如下：

1. 用 touch 创建一个 a.txt 输入内容“hello” 类似于/etc/passwd
2. 写一个程序 1.c 如下：

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

```
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <errno.h>
int main()
{
        printf("uid : %d    gid : %d\n", getuid(), getgid());
        printf("eudi: %d    egid: %d\n", geteuid(), getegid());

        FILE* fp = fopen("a.txt", "a");//注意这里是以追加形式打开，说明 a.txt 要具有可写权限
        if(fp == NULL)
        {
                perror("fopen error");
                exit(-1);
        }
        fputs("world", fp);
        fclose(fp);
        return 0;
}
```

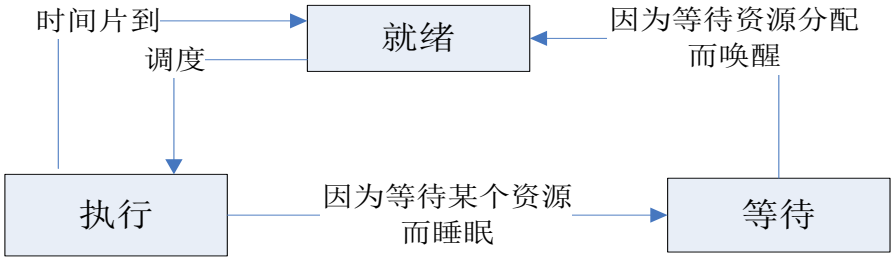
3. 编译 gcc -o 1 1.c 生成可执行程序 1 此时 1 类似于/usr/bin/passwd 文件
4. 用 ll 查看 a.txt 发现权限是-rw-r—r—的权限（跟/etc/passwd 权限一样），说明普通用户没有可写权限，如果直接切换到普通用户执行./1 会报错。
5. 用 root 身份将 1 的权限修改为-rwsr-xr-x (操作命令：chmod u+s 1，此时跟/usr/bin/passwd 的权限一样)，此时再切换到普通用户 wangxiao，执行./1 发现可以执行成功，因为此时./1 进程的有效用户 id 变成 root 了，也就是说普通用户是借助 root 身份来实现的。

6.S 是 chmod u-x a.out 之后的显示效果，t 是 chmod o+t dir1,T 是 chmod o-x dir1 之后的效果

1.3. 进程的状态

进程是程序的执行过程，根据它的生命周期可以划分成 3 种状态。

- 执行态：该进程正在运行，即进程正在占用 CPU。
- 就绪态：进程已经具备执行的一切条件，正在等待分配 CPU 的处理时间片。
- 等待态：进程不能使用 CPU，若等待事件发生（等待的资源分配到）则可将其唤醒。



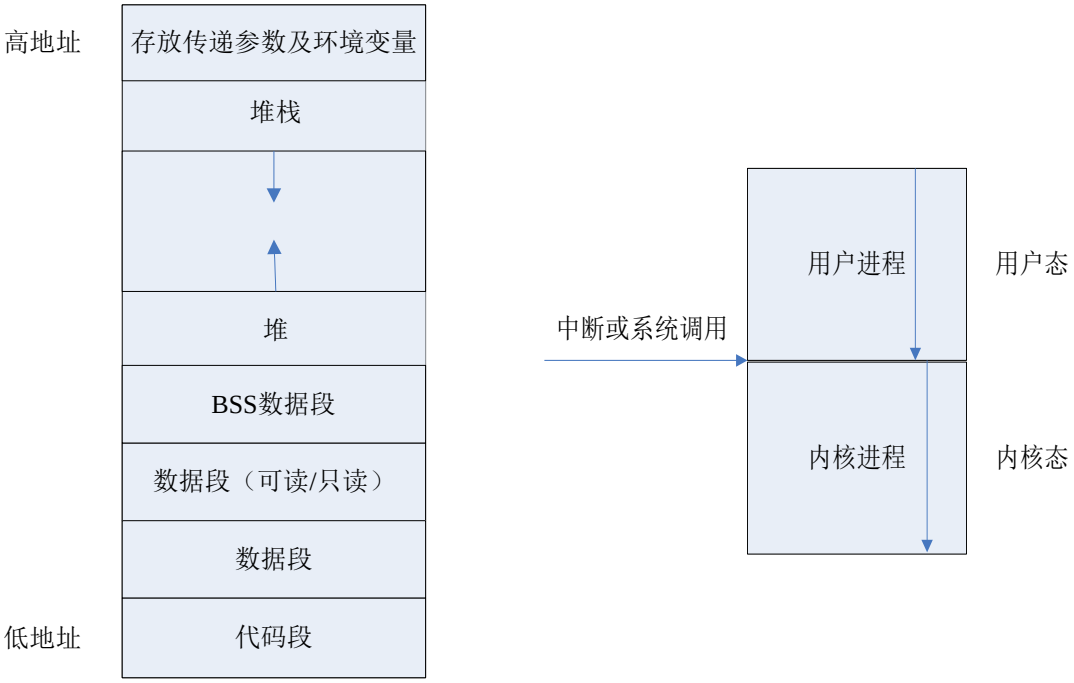
1.4. Linux 下的进程结构

Linux 系统是一个多进程的系统，它的进程之间具有并行性、互不干扰等特点。也就是说，进程之

间是分离的任务，拥有各自的权利和责任。其中，每个进程都运行在各自独立的虚拟地址空间，因此，即使一个进程发生了异常，它也不会影响到系统的其他进程。

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

- “数据段”放**全局变量、常数以及动态数据分配的数据空间**。数据段分成**普通数据段（包括可读可写/只读数据段，存放静态初始化的全局变量或常量）、BSS 数据段（存放未初始化的全局变量）**以及**堆（存放动态分配的数据）**。
- **“代码段”存放的是程序代码的数据**。
- **“堆栈段”存放的是子程序的返回地址、子程序的参数以及程序的局部变量等**。



1.5. Linux 下的进程管理

启动进程：手工启动 调度启动

命令	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	向进程发送信号（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。
bg	将挂起的进程放到后台执行

备注：

进程 process：是 os 的最小单元 os 会为每个进程分配大小为 4g 的虚拟内存空间，其中 1g 给内核

空间　3g 给用户空间{**代码区　数据区　堆栈区**}

ps 查看活动进程　ps －aux 查看所有的进程　**ps -aux| grep 'aa'** 查找指定（aa）进程　ps －ef 可以显示父子进程关系

进程状态：执行　就绪　等待状态

ps -aux 看%cpu(cpu 使用量)　%mem（内存使用量）　stat 状态{**S 睡眠　T 暂停　R 运行　Z 僵尸**}

top 显示前 20 条进程，动态的改变

```
top - 16:24:25 up 284 days,  4:59,  1 user,  load average: 0.10, 0.05, 0.01
Tasks: 115 total,   1 running, 114 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.1%us,  0.0%sy,  0.0%ni, 99.8%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:   4074364k total,  3733628k used,   340736k free,   296520k buffers
Swap:  2104504k total,    40272k used,  2064232k free,   931680k cached

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
11836 root      15   0  2324 1028  800 R  0.3  0.0   0:00.02 top
27225 root      25   0 1494m 696m  11m S  0.3 17.5   2304:03 java
1 root      18   0  2072  620  532 S  0.0  0.0   7:04.48 init
```

第一行分别显示：

```
16:24:25 当前时间、
up 284 days,  4:59 系统启动时间、
1 user 当前系统登录用户数目、
load average: 0.10, 0.05, 0.01 平均负载（1 分钟,10 分钟,15 分钟）。
```

平均负载（load average）,一般对于单个 cpu 来说，负载在 0～1.00 之间是正常的，超过 1.00 须引起注意。在多核 cpu 中，系统平均负载不应该高于 cpu 核心的总数。

第二行分别显示：

```
115 total 进程总数、
1 running 运行进程数、
114 sleeping 休眠进程数、
0 stopped 终止进程数、
0 zombie 僵死进程数。
```

第三行：

```
0.1%us        %us 用户空间占用 cpu 百分比;
0.0%sy        %sy 内核空间占用 cpu 百分比;
0.0%ni        %ni 用户进程空间内改变过优先级的进程占用 cpu 百分比;
99.8%id       %id 空闲 cpu 百分比，反映一个系统 cpu 的闲忙程度。越大越空闲;
0.0%wa        %wa 等待输入输出（I/O）的 cpu 百分比;
0.0%hi        %hi 指的是 cpu 处理硬件中断的时间;
0.1%si        %si 值的是 cpu 处理软件中断的时间;
0.0%st        %st 用于有虚拟 cpu 的情况，用来指示被虚拟机偷掉的 cpu 时间。
```

第四行（Mem）：

```
4074364k total          total 总的物理内存;
3733628k used used 使用物理内存大小;
340736k free  free 空闲物理内存;
296520k buffers         buffers 用于内核缓存的内存大小
```

第五行（Swap）：

```
2104504k total          total 总的交换空间大小;
40272k used   used 已经使用交换空间大小;
2064232k free free 空间交换空间大小;
931680k cached          cached 缓冲的交换空间大小
```

buffers 与 cached 区别：buffers 指的是块设备的读写缓冲区，cached 指的是文件系统本身的页面缓存。他们都是 Linux 系统底层的机制，为了加速对磁盘的访问。

然后下面就是和 ps 相仿的各进程情况列表了

第六行：
PID　　进程号
USER 运行用户
PR
优先级，PR(Priority)优先级
NI　　任务 nice 值
VIRT　　进程使用的虚拟内存总量，单位 kb。VIRT=SWAP+RES
RES　物理内存用量
SHR　共享内存用量
S　该进程的状态。其中 S 代表休眠状态；D 代表不可中断的休眠状态；R 代表运行状态；Z 代表僵死状态；T 代表停止或跟踪状态
%CPU　该进程自最近一次刷新以来所占用的 CPU 时间和总时间的百分比
%MEM　该进程占用的物理内存占总内存的百分比
TIME+　累计 cpu 占用时间
COMMAND 该进程的命令名称，如果一行显示不下，则会进行截取。内存中的进程会有一个完整的命令行

vi a.c &(&表示后台运行)，一个死循环，按 ctrl+z 可以把进程暂停，再执行[bg 作业 ID]可以将该进程带入后台。利用 jobs 可以查看后台任务，fg 1 把后台任务带到前台，这里的 1 表示作业 ID
kill -9 进程号➔表示向某个进程发送 9 号信号，从而杀掉某个进程　利用 pkill a 可以杀死进程名为 a 的进程

free 命令用来查看物理内存

fdisk　-l 查看磁盘及磁盘分区情况

2. 进程的创建

Linux 下有四类创建子进程的函数：system(),fork(),exec*(),popen()

2.1. system 函数

原型：

```
#include <stdlib.h>
int system(const char *string);
```

system 函数通过调用 shell 程序/bin/sh –c 来执行 string 所指定的命令，该函数在内部是通过调用**execve(“/bin/sh”,..)**函数来实现的。通过 system 创建子进程后，原进程和子进程各自运行，相互间关联较少。如果 system 调用成功，将返回 0。

示例：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    system("ls -l ");    //system(“clear”);表示清屏
    return 0;
}
```

此外，system函数后面的参数还可以是一个可执行程序，例如：system(“/home/wangxiao/1”);如果想要执行system后面进程的时候，不至于对当前进程进

行阻塞，可以利用&将/home/wangxiao/1调到后台运行。

2.2. fork 函数

原型：

```
#include <unistd.h>
pid_t fork(void);
```

在 linux 中 fork 函数时非常重要的函数，它从已存在进程中创建一个新进程。新进程为子进程，而原进程为父进程。它和其他函数的区别在于：它执行一次返回两个值。其中父进程的返回值是子进程的进程号，而子进程的返回值为 0.若出错则返回-1.因此可以通过返回值来判断是父进程还是子进程。

fork 函数创建子进程的过程为：使用 fork 函数得到的子进程是父进程的一个复制品，它从父进程继承了进程的地址空间，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，而子进程所独有的只有它的进程号、资源使用和计时器等。通过这种复制方式创建出子进程后，原有进程和子进程都从函数 fork 返回，各自继续往下运行，但是原进程的 fork 返回值与子进程的 fork 返回值不同，在原进程中，fork 返回子进程的 pid,而在子进程中，fork 返回 0,如果 fork 返回负值，表示创建子进程失败。（vfork 函数）

示例：

```
#include <stdlib.h>
#include <unistd.h>
int main()
{
        printf("Parent process id:%d\n",getpid());
        pid_t iRet = fork();
        if(iRet < 0){//出错
                printf("Create child process fail!\n");
        }else if(iRet == 0){//表示子进程
                printf("child process id:%d ppid:%d\n",getpid(),getppid());
        }else{//表示父进程
                printf("parent process success,child id:%d\n",iRet);
        }
        return 0;
 }
```

//有人可能会有疑问：这里怎么 if 和 else 里面的语句都得到执行了，和我们以前的 if...else 结构相矛盾啊？此时相当于有两份 main 函数代码的拷贝，其中一份做的操作是 if(iRet == 0)的情况；另外一份做的操作是 else(父)的情况。所以可以输出 2 句话。提问：如何创建兄弟进程和爷孙进程？

2.3. exec 函数族

exec*由一组函数组成

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

exec 函数族的工作过程与 fork 完全不同，fork 是在复制一份原进程，而 exec 函数是用 exec 的第一个参数指定的程序覆盖现有进程空间（也就是说执行 exec 族函数之后，它后面的所有代码不在执行）。

path 是包括执行文件名的全路径名

arg 是可执行文件的命令行参数，多个用，分割注意最后一个参数必须为 NULL。

例如，有个加法程序，从命令行接受两个数，输出其和 。

代码如下：

```
//add.c
#inclue<stdio.h>
#include <string.h>
int main(int argc , char * argv[])
{
    int a = atoi(argv[1]) ;
    int b = atoi(argv[2]);
    print f("%d + %d = %d" , a , b , a + b);
    return 0 ;
}
```

编译连接得到 add.exe.

```
gcc –o add.exe   add.c
```

然后在 main.exe 中调用 add.exe 程序 ,计算 3 和 4 的和。

Main.c 的源程序为 ，

```
//main.c
#include <stdio.h>
#include <string.h>
int main()
{
    execl("./add.exe" ,"add.exe" ,"3" , "4" , NULL):
    return 0 ;
}
```

编译连接得，

```
gcc –o main.exe main.c
```

然后运行 。./main.exe。

在运行 main.exe 的过程中会通过 execl 启动之前的 add.exe 程序。

当进程通过 exec 类系统调用开始某个程序的执行时，内核分配给进程的虚拟地址空间由以下内存区域组成：

1、程序的可执行代码
2、程序的初始化数据
3、程序的未初始化数据
4、初始化程序栈（即用户态栈）
5、所需共享库的可执行代码和数据
6、堆（由程序动态请求的内存）

2. 4popen 函数

popen 函数类似于 system 函数， 与 system 的不同之处在于它使用管道工作。原型为：

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

command 为可执行文件的全路径和执行参数；

type 可选参数为"r"或"w",如果为"w",则 popen 返回的文件流做为新进程的标准输入流，即 stdin， 如果为"r",则 popen 返回的文件流做为新进程的标准输出流。

如果 type 是“r”，（即 command 命令执行的输出结果作为当前进程的输入结果）。被调用程序的输出就可以被调用程序使用，调用程序利用 popen 函数返回的 FILE*文件流指针，就可以通过常用的 stdio 库函数（如 fread）来读取被调用程序的输出；如果 tpye 是“w”，（即当前进程的输出结果作为 command 命令的输入结果）。调用程序就可以用 fwrite 向被调用程序发送数据，而被调用程序可以在自己的标准输入上读取这些数据。

pclose 等待新进程的结束，而不是杀新进程。

示例：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;

    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("ps -ax", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
            buffer[chars_read - 1] = '\0';
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

内核暂停一个进程执行时，就会把几个相关处理器寄存器的内容保存在进程描述符中，这些寄存器包括：

1、 程序计数器（PC）和栈指针（SP）寄存器
2、 通用寄存器
3、 浮点寄存器
4、 包含 CPU 状态信息的处理器控制寄存器（处理器状态字）
5、 用来跟踪进程对 RAM 访问的内存管理寄存器

内核决定恢复执行一个进程时，它用进程描述符中合适的字段来装载 CPU 寄存器。

3. 进程控制与终止

3.1. 进程的控制

用 fork 函数启动一个子进程时，子进程就有了它自己的生命并将独立运行。

如果父进程先于子进程退出，则子进程成为**孤儿进程**，此时将自动被 PID 为 1 的进程（即 init）接管。孤儿进程退出后，它的清理工作有祖先进程 init 自动处理。但在 init 进程清理子进程之前，它一直消耗系统的资源，所以要尽量避免。

Example1：写一个孤儿进程：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
main()
{
    pid_t pid = fork();
    if( pid == 0)
    {
        while(1) ;
    }
    else
    {
            exit(10);
    }
}
```

通过 ps –ef 就可以看到此时子进程一直在运行，并且父进程是 1 号进程。

如果子进程先退出，系统不会自动清理掉子进程的环境，而必须由父进程调用 wait 或 waitpid 函数来完成清理工作，如果父进程不做清理工作，则已经退出的子进程将成为**僵尸进程(defunct)**,在系统中如果存在的僵尸（zombie）进程过多，将会影响系统的性能，所以必须对僵尸进程进行处理。

函数原型：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

wait 和 waitpid 都将暂停父进程，等待一个已经退出的子进程，并进行清理工作；

wait 函数随机地等待一个已经退出的子进程，并返回该子进程的 pid；

waitpid 等待指定 pid 的子进程；如果为-1 表示等待所有子进程。

status 参数是传出参数，存放子进程的退出状态；通常用下面的两个宏来获取状态信息：

WIFEXITED(status)　　如果子进程正常结束，它就取一个非 0 值。传入整型值，非地址

WEXITSTATUS(status) 如果 WIFEXITED 非零，它返回子进程的退出码

options 用于改变 waitpid 的行为，其中最常用的是 WNOHANG，它表示无论子进程是否退出都将立即返回，不会将调用者的执行挂起。

Example1：写一个僵尸进程：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
main()
{
    pid_t pid = fork();
    if( pid == 0 )
    {
        exit(10);
    }
    else
    {
            sleep(10);
    }
}
```

通过用 ps –aux 快速查看发现 Z 的僵尸进程。

Example2：避免僵尸进程：(wait()函数)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
main()
{
    pid_t pid = fork();
    if( pid == 0 )
    {
        exit(10);
    }
    else
    {
            wait(NULL);       //NULL 表示等待所有进程
            sleep(10);  //通常要将 sleep 放在 wait 的后面，要不然也会出现僵尸进
程
    }
}
```

Example3：利用信号处理避免僵尸进程：（wait ()函数）

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
void SignChildPsExit(int iSignNo)
{
    int iExitCode;
    pid_t pid = wait(&iExitCode);     //等待子进程的退出，没有这句会出现僵尸进程
    printf("SignNo:%d      child %d exit\n",iSignNo,pid);
```

```
        if(WIFEXITED(iExitCode))
        {
            printf("Child exited with code %d\n", WEXITSTATUS(iExitCode));
        }
        sleep(10);
    }
    int main()
    {
        signal(SIGCHLD, SignChildPsExit);
        printf("Parent process id:%d\n", getpid());
        pid_t iRet = fork();
        if(iRet == 0)
            exit(3);
        Else
```

} 放在信号部分讲解

Example4：waitpid实现

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
void SignChildPsExit(int iSignNo)
{
    int iExitCode;
    pid_t pid = waitpid(-1,NULL,0); //表示等待任何进程，并阻塞。如果换成
waitpid(-1,NULL,WNOHANG);则跟没有写waitpid效果类似，此时父进程没有阻塞
    printf("SignNo:%d      child %d exit\n",iSignNo,pid);
    if(WIFEXITED(iExitCode))
    {
        printf("Child exited with code %d\n", WEXITSTATUS(iExitCode));
    }
    sleep(10);
}
int main()
{
    signal(SIGCHLD, SignChildPsExit);
    printf("Parent process id:%d\n", getpid());
    pid_t iRet = fork();
    if(iRet == 0)
        exit(3);
}
```

3.2. 进程的终止

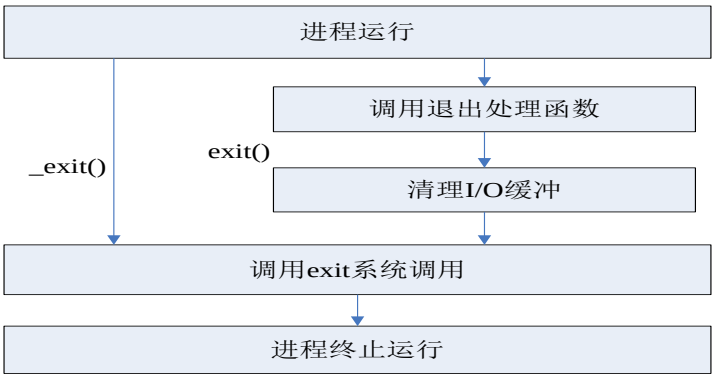
进程的终止有 5 种方式：

- main 函数的自然返回；
- 调用 exit 函数

- 调用_exit 函数
- 调用 abort 函数
- 接收到能导致进程终止的信号 ctrl+c SIGINT　ctrl+\ SIGQUIT

前 3 种方式为正常的终止，后 2 种为非正常终止。但是无论哪种方式，进程终止时都将执行相同的关闭打开的文件，释放占用的内存等资源。只是后两种终止会导致程序有些代码不会正常的执行比如对象的析构、atexit 函数的执行等。

exit 和_exit 函数都是用来终止进程的。当程序执行到 exit 和_exit 时，进程会无条件的停止剩下的所有操作，清除包括 PCB 在内的各种数据结构，并终止本程序的运行。但是它们是有区别的，exit 和_exit 的区别如图所示：



exit 函数和_exit 函数的最大区别在于 exit 函数在退出之前会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/O 缓冲”。

由于 linux 的标准函数库中，有一种被称作“缓冲 I/O”操作，其特征就是对应每一个打开的文件，在内存中都有一片缓冲区。每次读文件时，会连续读出若干条记录，这样在下次读文件时就可以直接从内存的缓冲区中读取；同样，每次写文件的时候，也仅仅是写入内存中的缓冲区，等满足一定的条件（如达到一定数量或遇到特定字符等），再将缓冲区中的内容一次性写入文件。这种技术大大增加了文件读写的速度，但也为编程带来了麻烦。比如有一些数据，认为已经写入文件，实际上因为没有满足特定的条件，它们还只是保存在缓冲区内，这时用_exit 函数直接将进程关闭，缓冲区中的数据就会丢失。因此，如想保证数据的完整性，建议使用 exit 函数。

exit 和_exit 函数的原型：

```
#include <stdlib.h>        //exit 的头文件
#include <unistd.h>        //_exit 的头文件
void exit(int status);
void _exit(int status);
```

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。

Example1：exit 的举例如下：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Using exit...\n");
    printf("This is the content in buffer");
    exit(0);
```

```
}
```

可以发现，调用 exit 函数，缓冲区中的记录也能正常输出。

Example2：_exit 的举例如下：

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("Using _exit...\n");
    printf("This is the content in buffer");
    _exit(0);
}
```

可以发现，最后的输出结果没有 This is the content in buffer，说明_exit 函数无法输出缓冲区中的记录。

4. 进程间打开文件的继承

4.1. 用 fork 继承打开的文件

fork 以后的子进程自动继承了父进程的打开的文件，继承以后，父进程关闭打开的文件不会对子进程造成影响。

示例：

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
{
    char szBuf[32] = {'\0'};
    int iFile = open("./a.txt", O_RDONLY);
    if(fork() > 0){//parent process
        close(iFile);
        return 0;
    }
    //child process
    sleep(3);      //wait for parent process closing fd
    if(read(iFile, szBuf, sizeof(szBuf)-1) < 1){
        perror("read fail");
    }else{
        printf("string:%s\n",szBuf);
    }
    close(iFile);
    return 0;


}
```

4.2. 守护进程

Daemon 运行在后台也称作“后台服务进程”。 它是没有控制终端与之相连的进程。它独立与控制终端、会话周期的执行某种任务。那么为什么守护进程要脱离终端后台运行呢？守护进程脱离终端是为了避免进程在执行过程中的信息在任何终端上显示并且进程也不会被任何终端所产生的任何终端信息所打断。那么为什么要引入守护进程呢？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赖这个终端，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。但是守护进程却能突破这种限制，它被执行开始运转，直到整个系统关闭时才退出。几乎所有的服务器程序如 Apache 和 wu-FTP，都用 daemon 进程的形式实现。很多 Linux 下常见的命令如 inetd 和 ftpd，末尾的字母 d 通常就是指 daemon。

守护进程的特性：

1> 守护进程最重要的特性是后台运行。

2> 其次，守护进程必须与其运行前的环境隔离开来。这些环境包括未关闭的文件描述符、控制终端、会话和进程组、工作目录已经文件创建掩码等。这些环境通常是守护进程从父进程那里继承下来的。

3> 守护进程的启动方式

daemon 进程的编程规则

- 创建子进程，父进程退出：

调用 fork 产生一个子进程，同时父进程退出。我们所有后续工作都在子进程中完成。这样做我们可以交出控制台的控制权,并为子进程作为进程组长作准备;由于父进程已经先于子进程退出，会造成子进程没有父进程，变成一个孤儿进程（orphan）。每当系统发现一个孤儿进程，就会自动由 1 号进程收养它，这样，原先的子进程就会变成 1 号进程的子进程。代码如下：

```
pid = fork();
if(pid>0)
    exit(0);
```

- 在子进程中创建新会话：

使用系统函数 setsid()。由于创建守护进程的第一步调用了 fork 函数来创建子进程，再将父进程退出。由于在调用 fork 函数的时候，子进程全盘拷贝了父进程的会话期、进程组、控制终端等，虽然父进程退出了，但会话期、进程组、控制终端并没有改变，因此，还不是真正意义上的独立开来。而调用 setsid 函数会创建一个新的会话并自任该会话的组长，调用 setsid 函数有下面 3 个作用：让进程摆脱原会话的控制，让进程摆脱原进程组的控制，让进程摆脱原控制终端的控制；

进程组：是一个或多个进程的集合。进程组有进程组 ID 来唯一标识。除了进程号(PID)之外，进程组 ID（GID）也是一个进程的必备属性。每个进程都有一个组长进程，其组长进程的进程号等于进程组 ID。且该进程组 ID 不会因为组长进程的退出而受影响。

会话周期：会话期是一个或多个进程组的集合。通常，一个会话开始于用户登录，终止于用户退出，在此期间该用户运行的所有进程都属于这个会话期。

控制终端：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赖这个控制终端。

- 改变当前目录为根目录：

使用 fork 函数创建的子进程继承了父进程的当前工作目录。由于在进程运行中，当前目录所在的文件是不能卸载的，这对以后的使用会造成很多的不便。利用 chdir("/");把当前工作目录切换到根目录。

- 重设文件权限掩码：

umask(0);将文件权限掩码设为 0,Deamon 创建文件不会有太大麻烦；

- 关闭所有不需要的文件描述符：

新进程会从父进程那里继承一些已经打开了的文件。这些被打开的文件可能永远不会被守护进程读写，而它们一直消耗系统资源。另外守护进程已经与所属的终端失去联系，那么从终端输入的字符不可能到达守护进程，守护进程中常规方法（如 printf）输出的字符也不可能在终端上显示。所以通常关闭从 0 到 MAXFILE 的所有文件描述符。

```
for(i=0;i<MAXFILE;i++)
    close(i);
```

（注：有时还要处理 SIGCHLD 信号 signal(SIGCHLD, SIG_IGN);防止僵尸进程（zombie））

下面就可以添加任何你要 daemon 做的事情

示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
void Daemon()
{
    const int MAXFD=64;
    int i=0;
    if(fork()!=0)//父进程退出
        exit(0);
    setsid();     //成为新进程组组长和新会话领导，脱离控制终端
    chdir("/");   //设置工作目录为根目录
    umask(0);     //重设文件访问权限掩码
    for(;i<MAXFD;i++) //尽可能关闭所有从父进程继承来的文件
        close(i);
}
int main()
{
    Daemon(); //成为守护进程
    while(1){
        sleep(1);
    }
    return 0;
}
```

Example：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <syslog.h>
#include <sys/stat.h>
#include <time.h>
main()
{
    int i = 0;
    if(fork() > 0)
```

```
            exit(0);
        setsid();
        chdir("/");
        umask(0);
        for(; i < 64; i++)
        {
            close(i);
        }
        i = 0;
        while(i < 10)
        {
            printf("%d\n",i);
            time_t ttime;
            time(&ttime);
            struct tm *pTm = gmtime(&ttime);
            syslog(LOG_INFO,"%d  %04d:%02d:%02d", i, (1900 + pTm->tm_year), (1 +
pTm->tm_mon), (pTm->tm_mday));
            i++;
            sleep(2);
        }
    }
```

通过查看vi /var/log/messages