更多嵌入式 Linux 学习资料，请关注：一口 Linux 回复关键字:1024

关于进程和线程的关系，之前一口君写过这几篇文章，大家可以参考下。

本文从头带着大家一起学习 Linux 进程

《搞懂进程组、会话、控制终端关系，才能明白守护进程干嘛的？》

《[粉丝问答 6]子进程进程的父进程关系》

《多线程详解，一篇文章彻底搞懂多线程中各个难点》

《一个多线程的简单例子让你看清线程调度的随机性》

Linux 进程篇

一、进程相关概念

了解进程的时候先来了解几个问题，明白以下问题，就懂了进程的概念

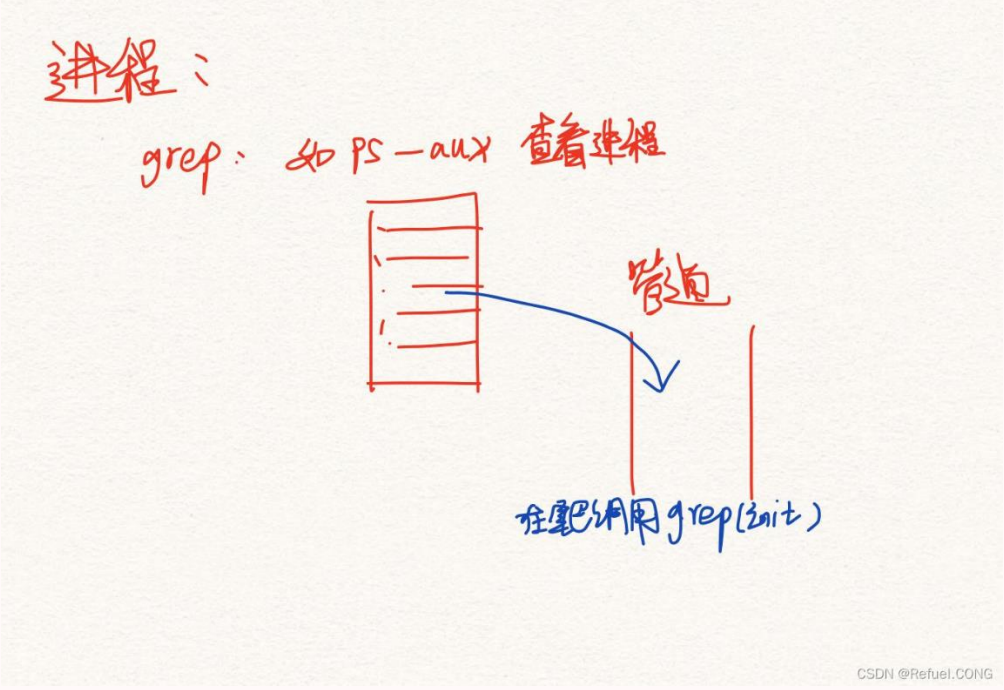
1. 什么是程序，什么是进程，两者之间的区别？

1. **程序**是静态的概念，gcc xxx.c -o pro 磁盘中生成 pro 文件，叫做程序 程序如：电脑上的图标
2. **进程**是程序的一次运行活动， 通俗点说就是程序跑起来了，系统中就多了一个进程

2. 如何查看系统中有哪些进程？

1. 使用 ps 指令查看 ： ps-aux 在 ubuntu 下查看， 在实际工作中，配合 grep 来查找程序中是否存在某一个进程

   grep 过滤进程 ： ps -aux | grep init 就只把带有 init 的进程过滤出来



2. 使用 top 指令查看，类似 windows 任务管理器

3. 什么是进程标识符？

每一个进程都有一个非负整数表示的唯一 ID，叫做 pid，类似身份证

pid =0 ：称为交换进程（swapper） 作用： 进程调度 pid=1 ：init 进程 作用： 系统初始化

- **编程调用 getpid 函数获取自身的进程标识符;**

```
#include<sys/types.h>
#include<unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

getpid 示例代码：

```
#include<stidio.h>
#include<sys/types.h>
#include<unistd.h>

int main()
```

```
{
    pid_t pid;
    pid = getpid();
   printf("my pid is %d\n",pid);
   return 0;
}
```

- **getppid 获取父进程的进程标识符;**

4. 第一个进程 init 进程

```
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 Mar23 ?        00:00:02 /sbin/init
root           2       0  0 Mar23 ?        00:00:00 [kthreadd]
root           3       2  0 Mar23 ?        00:00:01 [ksoftirqd/0]
root           5       2  0 Mar23 ?        00:00:00 [kworker/0:0H]
root           7       2  0 Mar23 ?        00:01:07 [rcu_sched]
root           8       2  0 Mar23 ?        00:00:48 [rcuos/0]
root           9       2  0 Mar23 ?        00:00:33 [rcuos/1]
root          10       2  0 Mar23 ?        00:00:33 [rcuos/2]
root          11       2  0 Mar23 ?        00:00:27 [rcuos/3]
```

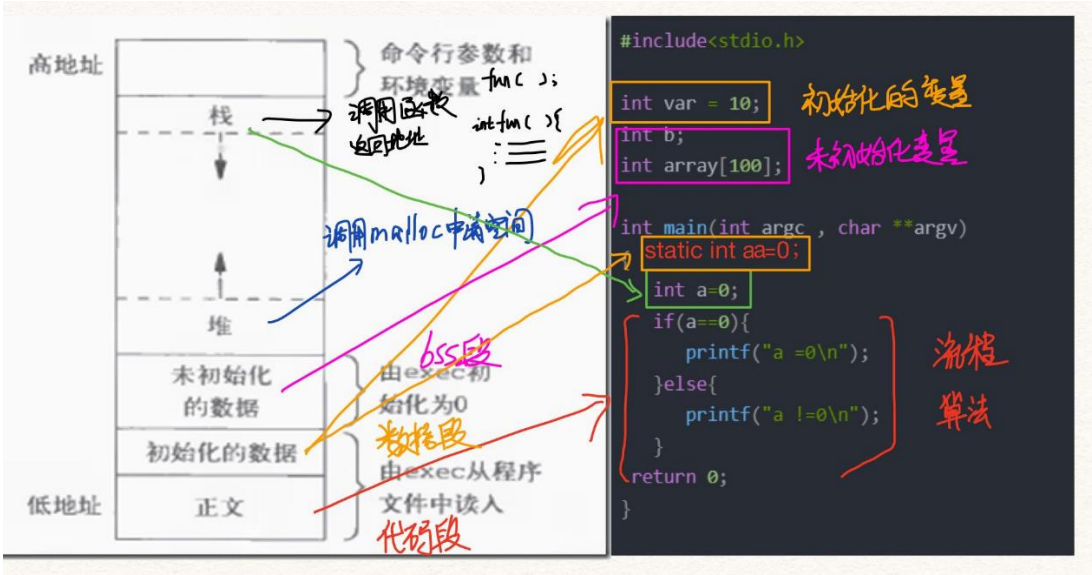

Linux 内核启动之后，会创建第一个**用户级进程 init**，由上图可知， **init 进程 (pid=1)** 是除了 **idle 进程(pid=0，也就是 init_task)** 之外另一个比较特殊的进程，它是 Linux 内核开始建立起进程概念时第一个通过 kernel_thread 产生的进程，其开始在内核态执行，然后通过一个系统调用，开始执行用户空间的 / sbin/init 程序。

5.什么叫父进程，什么叫子进程？

进程 A 创建了进程 B，那么 A 叫做父进程，B 叫做子进程，父进程是相对的概念，理解为人类中的父子关系

6. c 程序的存储空间是如何分配的？

gcc xxx.c -o a.out 当执行 ./a.out 时候，操作系统会划分一块内存空间，如何分配呢？ 如下图：



二、创建进程函数 fork 的使用

==pid_t fork(void);== 功能：使用 fork 函数创建一个进程

fork 函数调用成功，返回两次 返回值为 0 ，代表当前进程是子进程 返回值非负数，代表当前进程为父进程 调用失败 ，返回-1

1. fork();示例代码

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

int main()
{
    pid_t pid;
    pid = getpid();

    fork();

   printf("my pid is %d\n",pid);
    return 0;
}
```

打印出了两遍 my pid 说明，有了两个进程！执行了两次打印 pid

```
CLC@Embed_Learn:~$ ./demo
my pid is 8758
my pid is 8758
```

2. 查看父进程/子进程代码：

```c
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

int main()
{
    pid_t pid;
    pid_t pid2;

    pid = getpid();
    printf("brfore fork pid is %d\n",pid);

    fork();

    pid2 = getpid();
    printf("brfore fork pid is %d\n",pid2);

    if(pid == pid2){
        printf("this is father print\n");
    }else{

        printf("this is child print , child pid is =%d\n",getpid());
    }

   return 0;

}
```

```
CLC@Embed_Learn:~$ ./demo
brfore fork pid is 8915
after fork pid is 8915
this is father print
after fork pid is 8916
this is child print , child pid is =8916
```

父子进程都会进入 if 中，但是输出结果会不同 在 fork 之前的 pid 是 8915 是父进程 ，fork 之后 pid 是子进程 8916

3. 用返回值来判断父/子进程代码(1)：

返回值为 0 ，代表当前进程是子进程 返回值非负数，代表当前进程为父进程

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>

int main()
{

    pid_t pid;

    printf("father: id=%d\n",getpid());

    pid = fork();

    if(pid > 0){
        printf("this is father print ,pid =%d\n",getpid());
    }else if (pid == 0){
        printf("this is child print, child pid = %d\n",getpid());
    }

    return 0;
}
```

```
father: id=9026
this is father print ,pid =9026
this is child print, child pid = 9027
```

4. 用返回值来判断父子进程代码(2)：

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>


int main()
```

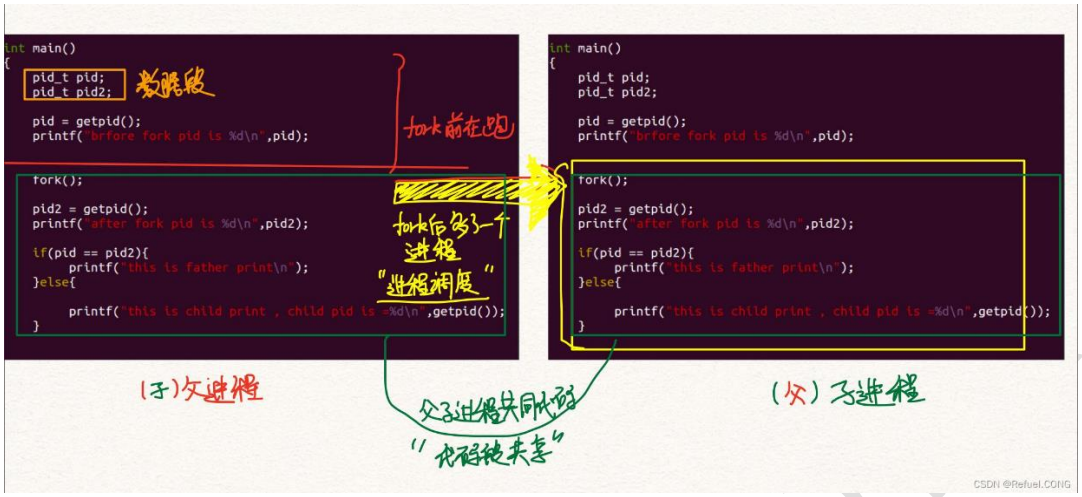
```
{

    pid_t pid;
    pid_t pid2;
    pid_t retpid;

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


    retpid = fork();


    pid2 = getpid();
    printf("after fork:pid = %d\n",pid2);


    if(pid == pid2){
        printf("this is father print :retpid = %d\n",retpid);
    }else{
        printf("this is child print :retpid =%d,child pid= %d\n",retpid,pid2);
    }

    return 0;
}
```

```
CLC@Embed_Learn:~$ ./demo2
before fork: pid = 9114
after fork:pid = 9114
this is father print :retpid = 9115
after fork:pid = 9115
this is child print :retpid =0,child pid= 9115
```

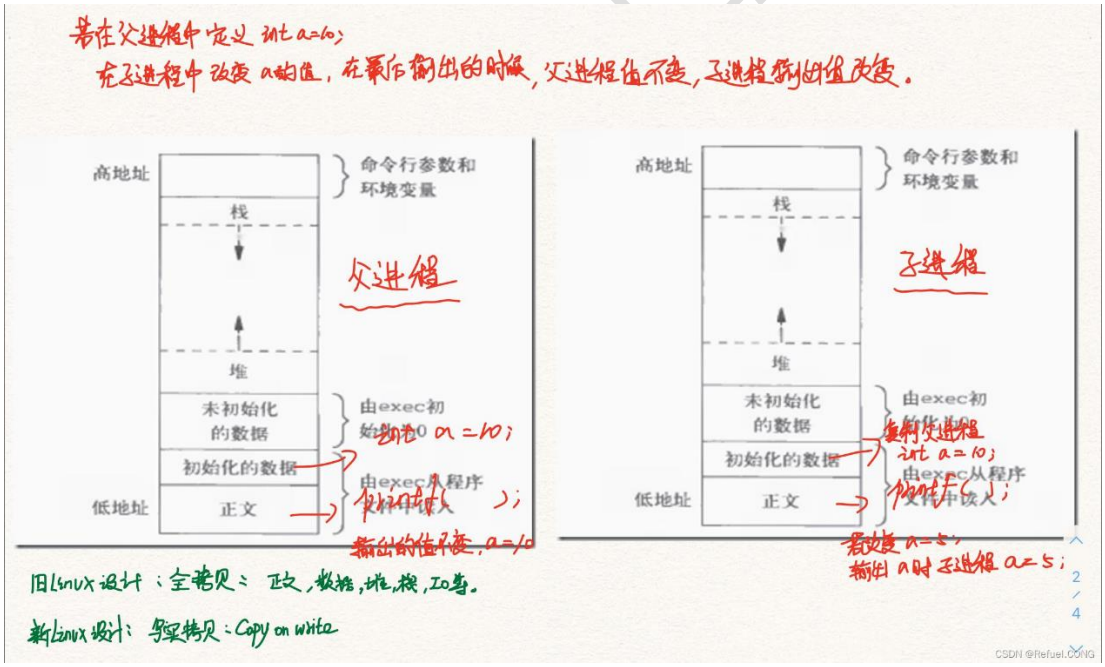
这样更清楚明了的看到

fork 返回值：9915>0 是父进程 父进程号是 9114 fork 返回值：=0 是子进程 子进程号是 9915

三、进程创建后 发生了什么事?



1 在内存空间中 fork 后发生了什么?



2. ./demo4 运行的程序父进程是谁?

```
int main(int argc, const char *argv[])
{
        while(1);
        return 0;
}
```

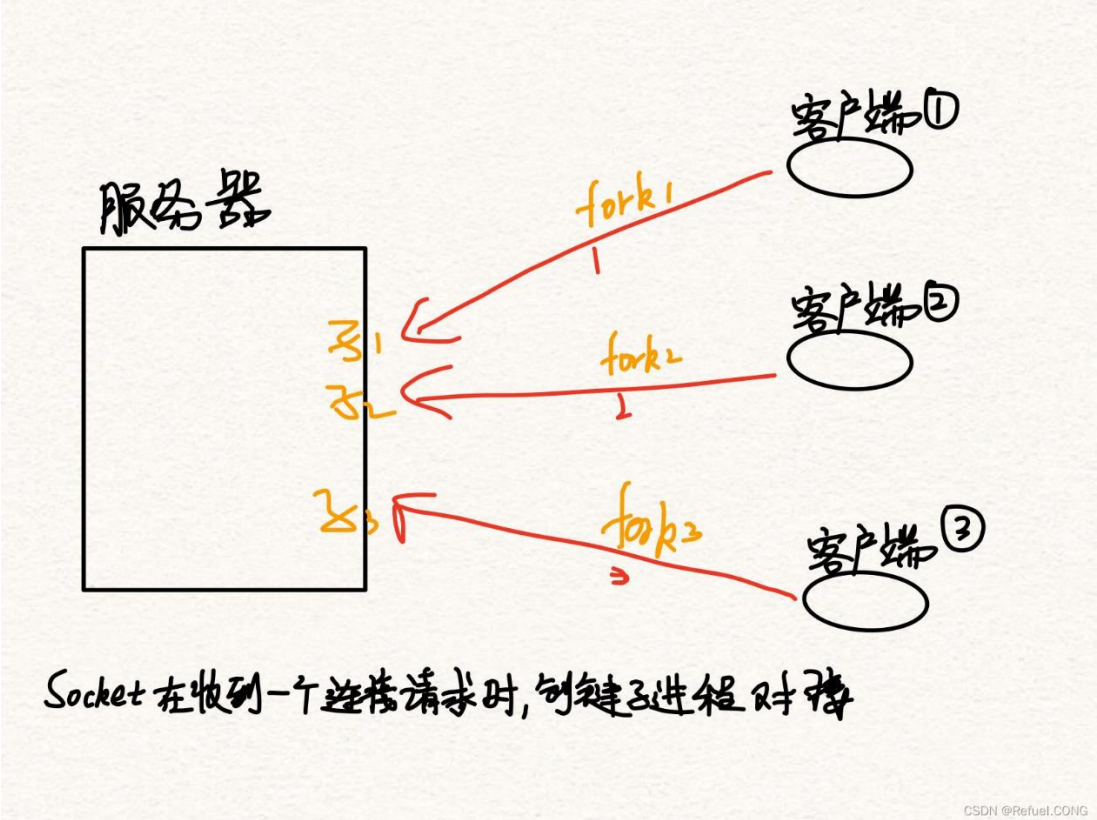
./ demo4 编译运行后，我们 ps -ef 查看进程 ID

由上图可知，./demo4 进程的进程 ID 是 12677，父进程 ID 是 12587，即进程 bash： ==**bash 的父进程是 gnome-terminal，所以我们打开 1 个 Linux 终端，其实就是启动了 1 个 gnome-terminal 进程。我们在这个终端上执行./a.out 其实就是利用 gnome-terminal 的子进程 bash 通过 execve()将创建的子进程装入 a.out:**==

四、创建新进程的实际应用场景

1. fork 创建子进程的一般目的：

- 一个父进程希望复制自己，使父、子进程同时执行不同的代码段。这在网络服务进程中是常见的——父进程等待客户端的服务请求。当这种情求达到时，父进程调用 fork，使子进程处理此请求。父进程则继续等待下一个服务请求到达。
- 一个进程要执行一个不同的程序。这对 shell 是常见的情况，在这种情况下子进程从 fork 返回后立即调用 exec。



2. 模拟 socket 创建进程（服务器对接客户端的应用场景）

示例代码：

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>

int main()
{
    pid_t  pid;
    int data;

    while(1){
        printf("please input a data\n");
        scanf("%d",&data);

        if(data ==1 )
        {
            pid = fork();

            if(pid >0){
```

```
            }
            else if(pid == 0){
                    while(1){
                            printf("do net request,pid=%d\n",getpid());
                            sleep(3);
                    }
            }
        }
        else{
            printf("wait, do noting\n");
        }
    }
    return 0;
}
```

输入非 1 时候，模拟没有客户端进行交互

```
please input a data
2
wait, do noting
please input a data
3
wait, do noting
please input a data
```

输入 1 时候，模拟有客户端进行交互 ，创建子进程来进行交互，子进程号为：9756

```
please input a data
1
please input a data
do net request,pid=9756
```

模拟多个客户端进行交互时 ，创建多个子进程来进行交互，子进程号为：9756 / 9758 / 9759

```
please input a data
1
please input a data
do net request,pid=9759
do net request,pid=9756
do net request,pid=9758
```

查看系统进程：

```
CLC@Embed_Learn:~$ ps -aux | grep newpro
Warning: bad ps syntax, perhaps a bogus '-'? See http://procps.sf.net/faq.html
CLC       9755  0.0  0.0   4164   352 pts/6    S+   20:23   0:00 ./newpro
CLC       9756  0.0  0.0   4164    96 pts/6    S+   20:23   0:00 ./newpro
CLC       9758  0.0  0.0   4164    96 pts/6    S+   20:23   0:00 ./newpro
CLC       9759  0.0  0.0   4164    96 pts/6    S+   20:23   0:00 ./newpro
CLC       9765  0.0  0.0  13588   940 pts/1    S+   20:24   0:00 grep --color=au
to newpro
```

3. fork 总结：

一个现有进程可以调用 fork 函数创建一个新进程。

#include cunistd.h> pid_t fork(void)；返回值：子进程中返回 0。父进程中返回子进程 ID.出错返回-1

由 fork 创建的新进程被称为子进程（child process)。fork 函数被调用一次，但返回两次。两次返回的唯一区别是子进程的返回值是 0，而父进程的返回值则是新子进程的进程 ID。将子进程 ID 返回给父进程的理由是：因为一个进程的子进程可以有多个，并且没有一个函数使一个进程可以获得其所有子进程的进程 ID。fork 使子进程得到返回值 0 的理由是：一个进程只会有一个父进程，所以子进程总是可以调用 getppid 以获得其父进程的进程 ID（进程 IDO 总是由内核交换进程使用，所以一个子进程的进程 ID 不可能为 0)。

子进程和父进程继续执行 fork 调用之后的指令。子进程是父进程的副本。例如，子进程获得父进程数据空间、堆和栈的副本。注意，这是子进程所拥有的副本。父、子进程并不共享这些存储空间部分。父、子进程共享正文段。 由于在 fork 之后经常跟随着 exec，所以现在的很多实现并不执行一个父进程数据段、栈和堆的完全复制。作为替代，使用了写时复制（Copy-On-Write，COW)技术。这些区域由父、子进程共享，而且内核将它们的访问权限改变为只读的。如果父、子进程中的任一个试图修改些区域，则内核只为修改区域的那块内存制作一个副本，通常是虚拟存储器系统中的一“页”。 Bach 和 McKusick 等对这种特征做了更详细的说明。

五、vfork 创建进程

1. vfork 函数 也可以创建进程，与 fork 有什么区别？

关键区别一： vfork 直接使用父进程存储空间，不用拷贝 **关键区别二：** vfork 保证子进程先运行，当子进程调用 exit 退出后，父进程才执行

2. fork 进程调度 父子进程：

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

int main()
{
    pid_t pid;

    pid = fork();

    if(pid >0){
        while(1){
                printf("this is father print pid is %d\n",getpid());
                sleep(3);
        }
    }else if(pid == 0){
         while(1){
                printf("this is child print pid is =%d\n",getpid());
                sleep(3);
         }
    }

   return 0;

}
```

```
CLC@Embed_Learn:~$ ./demo
this is father print pid is 9833
this is child print pid is =9834
this is father print pid is 9833
this is child print pid is =9834
this is father print pid is 9833
this is child print pid is =9834
```

3. vfork 进程调度 父子进程：

```c
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>

int main()
{
    pid_t pid;
    int cnt=0;

    pid = vfork();

    if(pid >0){
       while(1){
              printf("this is father print pid is %d\n",getpid());
              sleep(1);
       }
    }else if(pid == 0){
         while(1){
              printf("this is child print pid is =%d\n",getpid());
              sleep(1);
              cnt++;
              if(cnt == 3 ){
                  exit(0);
              }
         }
    }

   return 0;
}
```

vfork 保证子进程先运行，当子进程调用 3 次 exit 退出后，父进程才执行

```
this is child print pid is =10756
this is child print pid is =10756
this is child print pid is =10756
this is father print pid is 10755
this is father print pid is 10755
this is father print pid is 10755
```

4. 子进程改变 cnt 值，在父进程运行时候也被改变

```c
#include<stdio.h>
#include<sys/types.h>
```

```c
#include<unistd.h>
#include<stdlib.h>

int main()
{
    pid_t pid;
    int cnt=0;
    printf("cnt=%d\n",cnt);

    pid = vfork();

    if(pid >0){
       while(1){
              printf("cnt=%d\n",cnt);
              printf("this is father print pid is %d\n",getpid());
              sleep(1);
       }
    }else if(pid == 0){
        while(1){
              printf("this is child print pid is =%d\n",getpid());
              sleep(1);
              cnt++;
              if(cnt == 3 ){
                 exit(0);
              }
        }
    }

   return 0;
}
```

```
cnt=0
this is child print pid is =10825
this is child print pid is =10825
this is child print pid is =10825
cnt =3
this is father print pid is 10824
```

六、ps 常带的一些参数

下面对 ps 命令选项进行说明：

命令参数	说明
-e	显示所有进程.
-f	全格式。
-h	不显示标题。
-l	长格式。
-w	宽输出。
-a	显示终端上的所有进程，包括其他用户的进程。
-r	只显示正在运行的进程。
-u	以用户为主的格式来显示程序状况。
-x	显示所有程序，不以终端机来区分。

ps -ef 显示所有进程，全格式形式查看进程：

ps -ef 的每列的含义是什么呢？

```
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 Mar23 ?        00:00:02 /sbin/init
root           2     0  0 Mar23 ?        00:00:00 [kthreadd]
root           3     2  0 Mar23 ?        00:00:01 [ksoftirqd/0]
root           5     2  0 Mar23 ?        00:00:00 [kworker/0:0H]
root           7     2  0 Mar23 ?        00:01:07 [rcu_sched]
root           8     2  0 Mar23 ?        00:00:48 [rcuos/0]
root           9     2  0 Mar23 ?        00:00:33 [rcuos/1]
root          10     2  0 Mar23 ?        00:00:33 [rcuos/2]
root          11     2  0 Mar23 ?        00:00:27 [rcuos/3]
```


命令参数	说明
UID:	程序被该 UID 所拥有，指的是用户 ID
PID:	就是这个程序的 ID
PPID ：	PID 的上级父进程的 ID
C ：	CPU 使用的资源百分比
STIME ：	系统启动时间
TTY:	登入者的终端机位置

命令参数	说明
TIME ：	使用掉的 CPU 时间。
CMD:	所下达的指令为何

七、进程退出

1. 子进程退出方式

正常退出：

1. Mian 函数调用 return
2. 进程调用 exit()，标准 c 库
3. 进程调用_exit()或者——Exit(),属于系统调用
4. 进程最后一个线程返回
5. 最后一个线程调用 pthread_exit

异常退出：

1. 调用 abort
2. 当进程收到某些信号时候，如 ctrl+C
3. 最后一个线程对取消（cancellation）,请求作出响应

不管进程如何终止，最后都会执行内核中的同一段代码。这段代码为相应进程关闭所有打开描述符，释放它所使用的存储器等。
对上述任意一种终止情形，我们都希望终止进程能够通知其父进程它是如何终止的。对于三个终止函数（exit、_exit 和_Exit），实现这一点的方法是，将其退出状态作为参数传送给函数。**【如上面示例里面写到的 cnt==3 情况下，exit(0)；这个 0 就是子进程退出状态。】**在异常终止情况下，内核（不是进程本身）产生一个指示其异常终止原因的终止状态。在任何一种情况下，该终止进程的父进程都能用 wait 或者 waitpid 取得其终止状态。

正常退出的三个函数：

```
#include<stdlib.h>
void exit(int status);
```

```
#include<unistd.h>
```

```
void _exit(int status);
```

```
#include<stdlib.h>
void _Exit(int status);
```

记得在结束子进程的时候要手动退出，不要使用 break；会导致数据被破坏。三种退出函数种，更推荐 exit()；exit 是 _exit 和_Exit 的一个封装，会清除，冲刷缓冲区，把缓存区数据进程处理在退出。

2. 等待子进程退出

==为什么要等待子进程退出？==

创建子进程的目的就是为了让它去干活，在网络请求当中来了一个新客户端介入，创建子进程去交互，干活也要知道它干完没有. 比如正常退出（exit/_exit /_Exit）为 完成任务 若异常退出 （abort）不想干了， 或被杀了

所有要等待子进程退出，而且还要收集它退出的状态 等待就是调用 wait 函数和 waitpid 函数

3. 僵尸进程

子进程退出状态不被收集，会变成僵死进程（僵尸进程）

正如以下例子，**就是子进程退出没有被收集，成了僵尸进程：**

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>

int main()
{
    pid_t pid;
    int cnt=0;
    printf("cnt=%d\n",cnt);

    pid = vfork();

    if(pid >0){
        while(1){
```

```
            printf("cnt=%d\n",cnt);
            printf("this is father print pid is %d\n",getpid());
            sleep(1);
        }
    }else if(pid == 0){
        while(1){
            printf("this is child print pid is =%d\n",getpid());
            sleep(1);
            cnt++;
            if(cnt == 3 ){
                exit(0);
            }
        }
    }
   return 0;
}
```

```
CLC@Embed_Learn:~$ ./demo1
cnt=0
this is child print pid is =11315
this is child print pid is =11315
this is child print pid is =11315
cnt=3
this is father print pid is 11314
cnt=3
this is father print pid is 11314
```

运行三次子进程后，退出，父进程一直运行

```
CLC@Embed_Learn:~$ ps -aux | grep demo1
Warning: bad ps syntax, perhaps a bogus '-'? See http://procps
CLC       11314  0.0  0.0   4160   360 pts/0    S+   14:51   0
CLC       11315  0.0  0.0      0     0 pts/0    Z+   14:51   0
t>
CLC       11384  0.0  0.0  13588   940 pts/1    S+   14:51   0
to demo1
CLC@Embed_Learn:~$
```

结果：在查看进程时发现，父进程 11314 正在运行 “S+” 而子进程 11315 停止运行 “z+” z 表示 zombie（僵尸）

4. 等待函数：wait(状态码)；的使用：

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

pid_t wait(int *status);     //参数status 是一个地址
pid_t waitpid(pid_t pid , int *status ,int options);
int waitid(idtype_t idtype ,id_t id ,siginfo_t  *infop, int  options);
```

- 如果其所有子进程都还在运行，则阻塞。： 通俗的说就是子进程在运行的时候，父进程卡在 wait 位置阻塞，等子进程退出后，父进程开始运行。
- 如果一个子进程已终止，正等待父进程获取其终止状态，则会取得该子进程的终止状态立即返回。
- 如果它没有任何子进程，则立即出错返回。

status 参数： 是一个整型数指针 非空： 子进程退出状态放在它所指向的地址中。 空： 不关心退出状态

检查 wait 和 waitpid 所返回的终止状态的宏

宏	说明
WIFEXITED (status)	若为正常终止子进程返回的状态，则为真。对于这种情况可执行 WEXITSTATUS（status），取子进程传送给 exit、_exit 或_Exit 参数的低 8 位
WIFSIGNALED (status)	若为异常终止子进程返回的状态，则为真（接到一个不捕捉的信号）。对于这种情况，可执行 WTERMSIG(status),取使子进程终止的信号编号。另外，有些实现（非 Single UNIX Specification）宏义宏 WCOREDUMP（status），若已产生终止进程的 core 文件，则它返回真
WIFSTOPPED (status)	若为当前暂停子进程的返回的状态，则为真，对于这种情况，可执行 WSTIOPSIG（status），取使子进程暂停的信号编号
WIFCONTINUED (status)	若在作业控制暂停后已经继续的子进程返回了状态，则为真。（POSIX.1 的 XSI 扩展，仅用于 waitpid。）
比如说： exit(3) wait (状态码); 要通过宏来解析状态码	

5. 收集退出进程状态

```
pid = vfork();

    if(pid >0){
```

```
        while(1){
                printf("cnt=%d\n",cnt);
                printf("this is father print pid is %d\n",getpid());
                sleep(1);
        }
    }else if(pid == 0){
        wait(NULL);    // 参数：status  是一个地址  为空 表示不关心退出状态
        while(1){
                printf("this is child print pid is =%d\n",getpid());
                sleep(1);
                cnt++;
                if(cnt == 3 ){
                   exit(0);
                }
        }
    }
```

wait(NULL); // 参数：status 是一个地址 为空 表示不关心退出状态

没有了 11567 子进程，这样就不是僵尸进程了

```
CLC@Embed_Learn:~$ ps -aux | grep demo
Warning: bad ps syntax, perhaps a bogus '-'? See http://procps.sf.net/faq.html
CLC      11566  0.0  0.0   4160   360 pts/0    S+   15:07   0:00 ./demo
CLC      11635  0.0  0.0  13588   940 pts/1    S+   15:07   0:00 grep --color=a
to demo
```

收集子进程退出状态示例代码：

```
int main()
{
    pid_t pid;
    int cnt=0;
    int status =10;

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

    pid = vfork();

    if(pid >0){

        wait(&status);     // 参数status 是一个地址
        printf("child out ,chile status =%d\n",WEXITSTATUS(status));  // 要解析状态码，需要借助WEXITSTATUS
        while(1){
                printf("cnt=%d\n",cnt);
                printf("this is father print pid is %d\n",getpid());
                sleep(1);
```

```
        }
    }else if(pid == 0){
        while(1){
            printf("this is child print pid is =%d\n",getpid());
            sleep(1);
            cnt++;
            if(cnt == 3 ){
                exit(5);
            }
        }
    }
```

```
int status =10;
wait(&status); // 参数 status 是一个地址
printf("child out ,chile status =%d\n",WEXITSTATUS(status)); //要解析状态码，需要借助 WEXITSTATUS
```

```
CLC@Embed_Learn:~$ ./demo
cnt=0
this is child print pid is =11691
this is child print pid is =11691
this is child print pid is =11691
child out ,chile status =5
cnt=3
this is father print pid is 11690
```

结果显示：exit(5)；就能看到子进程退出的状态 status=5

6. 等待函数：waitpid（）的使用；

wait 和 waitpid 的区别之一：

wait 使父进程（调用者）阻塞，waitpid 有一个选项 ，可以使父进程（调用者）不阻塞。

```
pid_t waitpid(pid_t pid , int *status ,int options);
```

对于 waitpid 函数种 pid 参数的作用解释如下：

pid == -1	等待任一子进程。就这一方面而言，waitpid 与 wait 等效。
pid > 0	等待其进程 ID 与 pid 相等的子进程。
pid == 0	等待其组 ID 等于调用进程组 ID 的任一子进程

pid <-1	等待其组 ID 等于 pid 绝对值的任一子进程。

waitpid 的 options 常量:

WCONTINUED	若实现支持作业控制，那么由 pid 指定的任一子进程在暂停后已经继续，但其状态尚未报告，则返回其状态（POSIX.1 的 XSI 扩展）
WNOHANG	若由 pid 指定的子进程并不是立即可用的，则 waitpid 不阻塞，此时其返回值为 0;
WUNTRACED	若某实现支持作业控制，而由 pid 指定的任一子进程已处于暂停状态。

waitpid 来使得父进程不阻塞代码：

```
int main()
{
    pid_t pid;
    int cnt=0;
    int status =10;

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

    pid = vfork();

    if(pid >0){

        waitpid(pid,&status,WNOHANG); // 参数pid 是子进程号，WNOHANG 是若由 pid 指定的子进程并不是立即可用的，则waitpid 不阻塞，此时其返回值为 0;
        printf("child out ,chile status =%d\n",WEXITSTATUS(status));
        while(1){
                printf("cnt=%d\n",cnt);
                printf("this is father print pid is %d\n",getpid());
                sleep(1);
        }
    }else if(pid == 0){
          while(1){
                printf("this is child print pid is =%d\n",getpid());
                sleep(1);
                cnt++;
                if(cnt == 3 ){
                   exit(5);
```

```
            }
        }
    }
```

```
CLC@Embed_Learn:~$ ./demo1
child out ,chile status =0
cnt=0
this is father print pid is 12274
this is child print pid is =12275
cnt=0
this is child print pid is =12275
this is father print pid is 12274
```

子进程和父进程同时进行

```
Warning: bad ps syntax, perhaps a bogus '-'? See http://procps.sf.net/faq.html
CLC       12199  0.0  0.0   4160   360 pts/0    T    21:21   0:00 ./demo1
CLC       12224  0.0  0.0   4160   360 pts/0    T    21:23   0:00 ./demo1
CLC       12250  0.0  0.0   4160   352 pts/0    T    21:25   0:00 ./demo1
CLC       12251  0.0  0.0      0     0 pts/0    Z    21:25   0:00 [demo1] <defunc
t>
CLC       12271  0.0  0.0   4160   352 pts/0    T    21:27   0:00 ./demo1
CLC       12272  0.0  0.0      0     0 pts/0    Z    21:27   0:00 [demo1] <defunc
t>
CLC       12274  0.0  0.0   4160   348 pts/0    S+   21:27   0:00 ./demo1
CLC       12275  0.0  0.0      0     0 pts/0    Z+   21:27   0:00 [demo1] <defunc
t>
CLC       12341  0.0  0.0  13592   940 pts/1    S+   21:27   0:00 grep --color=au
to demo1
CLC@Embed_Learn:~$
```


但是发现子进程 12275 在系统查询进程中 还是变成了僵尸进程 原因是 ==WNOHANG 是不等待参数，它只运行一遍== ，当他运行时候，子进程没死，等子进程死后，他没运行，就没有收到停止状态，所以成了僵尸进程。

八、孤儿进程

1. 孤儿进程的概念：

父进程如果不等待子进程退出，在子进程结束前就了结束了自己的“生命”，此时子进程就叫做孤儿进程。

2. 孤儿进程被收留：

Linux 避免系统存在过多孤儿进程，init 进程收留孤儿进程，变成孤儿进程的父进程【init 进程(pid=1)是系统初始化进程】。init 进程会自动清理所有它继承的僵尸进程。

孤儿进程的代码:

```
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>

int main()
{
    pid_t pid;
    int cnt=0;
    int status =10;
    pid = fork();

    if(pid >0){
            printf("this is father print pid is %d\n",getpid());
        }
    else if(pid == 0){
        while(1){
            printf("this is child print pid is =%d,my father pid is=%d\n",getpid(),getppid());
            sleep(1);
            cnt++;
            if(cnt == 3 ){
                exit(5);
            }
        }
    }
  return 0;
}
```

```
CLC@Embed_Learn:~$ ./guer
this is father print pid is 13098
this is child print pid is =13099,my father pid is=13098
CLC@Embed_Learn:~$ this is child print pid is =13099,my father pid is=1
this is child print pid is =13099,my father pid is=1
```

父进程运行结束前，子进程的父进程 pid 还是 13098。 父进程运行结束后，子进程的父进程变成了 init 进程(pid=1)。

```
Warning: bad ps syntax, perhaps a bogus '-'? See http://procps.sf.net/faq.html
CLC       13102  0.0  0.0  13588   936 pts/1    S+   14:52   0:00 grep --color=au
to guer
```

九、exec 族函数

1. exec 族函数的作用：

我们用 fork 函数创建新进程后，经常会在新进程中调用 exec 函数去执行另外一个程序。当进程调用 exec 函数时，该进程被完全替换为新程序因为调用 exec 函数并不创建新进程，所以前后进程的 ID 并没有改变。

2. 为什么要用 exec 族函数，有什么作用？

1. 一个父进程希望复制自己，使父、子进程同时执行不同的代码段。这在网络服务进程中是常见的——父进程等待客户端的服务请求。当这种请求到达时，父进程调用 fork，使子进程处理此请求。父进程则继续等待下一个服务请求到达。
2. 一个进程要执行一个不同的程序。这对 shell 是常见的情况。在这种情况下，子进程从 fork 返回后立即调用 exec。

3. exec 族函数定义：

功能：

exec 函数族提供了一种在进程中启动另一个程序执行的方法，它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段。在执行完之后，原调用进程的内容除了进程号外，其他全部都被替换了。 在调用进程内部执行一个可执行文件，可执行文件既可以是二进制文件，也可以是 linux 下可执行的脚本文件。**【通俗理解就是执行 demo1 的同时，执行一半去执行 demo2。】**

函数族：

execl、execlp、execle、execv、execvp、execvpe

函数原型：

```
#include<unistd.h>

extern char **environ;
int execl(char *path   , char *arg ,  ...);
int execlp(char *file   , char *arg ,  ...);
int execle(char *path   , char *arg ,  ... , char *const envp[] );
```

```
int execv(char *path   , char *const argv[] );
int execvp(char *file   , char *const atgv[] );
int execvpe(char *file   , char *const argv[] , char *const envp[]);
```

返回值：

exec 函数族的函数执行成功后不会返回，调用失败时，会设置 errno 并返回-1，然后从原程序的调用点接着往下执行。

参数说明：

path ：可执行文件的路径名字 **arg**：可执行程序所带的参数，第一个参数为可执行文件名字，没有带路径且 arg 必须以 NULL 结束。 **file**：如果参数 file 中包含/，则就将其视为路径名，否则就按 PATH 环境变量，在它所指定的各目录中搜寻可执行文件。

exec 族函数参数极难记忆和分辨，函数名中的字符会给我们一些帮助：

字符	说明
l	使用参数列表
p	使用文件名，并从 PATH 环境寻找可执行文件
v	应该先构造一个指向各参数的指针数组，然后将该数组的地址作为这些函数的参数。
e	多了 envp[]数组，使用新的环境变量代替调用进程的环境变量

4. exec 函数 带 l 带 p 带 v 来说明参数特点

先写一个带参数的程序，输入参数 输出参数，在上一篇 Linux 文件编程里，main 参数我们学过。

./echoarg 代码：

```
#include<stdio.h>
int main(int argc , char *argv[])
{

    int i =0;
    for(i =0 ;i <argc;i++){
        printf("argv[%d]:%s\n",i ,argv[i]);
    }
  return 0;
}
```

在执行 a.out 代码一半的时候，调用上面的代码 echoarg

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

int main(void)
{
    printf("brfore execl\n");
    //int execl(char *path   , char *arg ,  ...);
    if(execl("/bin/echoarg","echoarg","abc",NULL)==-1)
    {
        printf("execl failed!\n");
    }
    printf("after execl \n");
    return 0;
}
```

exec 函数族的函数执行成功后不会返回，调用失败时，会设置 errno 并返回-1，然后从原程序的调用点接着往下执行。

if(execl("/bin/echoarg","echoarg","abc",NULL)==-1) 源代码：int execl(char *path , char *arg , ...); //最后一个参数是：arg 必须以 NULL 结束。

在执行 a.out 代码一半的时候，调用上面的代码 echoarg: exec 函数族的函数执行成功后不会返回，调用失败时，会设置 errno 并返回-1，然后从原程序的调用点接着往下执行。

```
CLC@Embed_Learn:~$ gcc ech.c -o ech
CLC@Embed_Learn:~$ ./ech
brfore execl
argv[0]:echoarg
argv[1]:abc
```

==perror("why"); //用来在执行错误时候，查询错误原因==

若要调用 ech 执行一般执行 ls ，同理。只需要改动

if(execl("/bin/ls","ls",NULL,NULL)==-1)

```
CLC@Embed_Learn:~$ ./ech
brfore execl
demo   demo1.c  demo2.c  demo.c  ech.c    echoarg.c       Fi
demo1  demo2    demo3.c  ech     echoarg  Fileprogramme   gu
```

若要调用 ech 执行一般执行 ls-l ，同理。

```
if(execl("/bin/ls","ls","-l",NULL)==-1)
```

```
CLC@Embed_Learn:~$ ./ech
brfore execl
total 128
-rwxr-xr-x 1 CLC book 8632 Mar 23 15:19 demo
-rwxr-xr-x 1 CLC book 8635 Mar 23 21:27 demo1
-rw-r--r-- 1 CLC book  756 Mar 24 14:48 demo1.c
-rwxr-xr-x 1 CLC book 8481 Mar 22 16:56 demo2
-rw-r--r-- 1 CLC book  466 Mar 23 13:34 demo2.c
-rw-r--r-- 1 CLC book  631 Mar 22 20:17 demo3.c
-rw-r--r-- 1 CLC book  734 Mar 24 14:31 demo.c
-rwxr-xr-x 1 CLC book 8426 Mar 24 17:26 ech
-rw-r--r-- 1 CLC book  259 Mar 24 17:26 ech.c
-rwxr-xr-x 1 CLC book 8381 Mar 24 17:10 echoarg
```


execlp 和 execl 的区别

带 p ： 可以通过环境变量 PATH 环境寻找可执行文件

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

int main(void)
{
    printf("brfore execl\n");
    //int execl(char *path  , char *arg ,  ...);
    if(execl("ls",";s",NULL,NULL)==-1)
    {
        printf("execl failed!\n");
    }
    printf("after execl \n");
    return 0;
}
```

在路径中不用写具体路径，就可以自动找到文件

```
CLC@Embed_Learn:~$ ./ech
brfore execl
demo   demo1.c  demo2.c  demo.c  ech.c    echoarg.c       Fi
demo1  demo2    demo3.c  ech     echoarg  Fileprogramme   gu
```

execvp 和 execl 的区别

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

```c
#include<unistd.h>


int main(void)
{
    printf("brfore execl\n");

    char *argv[] = {"ps",NULL,NULL};
    if(execvp("ps",argv)==-1)
    {
        printf("execl failed!\n");
    }
    printf("after execl \n");
    return 0;
}
```

```
char *argv[] = {"ps",NULL,NULL}; if(execvp("ps",argv)==-1)
```

结果与上面相同

5. 任何目录下执行程序

一个程序在目录下能运行，换一个目录就无法运行，如果把程序配置到环境变量里面去。

```
==pwd 显示当前路径 echo 查看环境变量 PATH: [pwd 显示的当前路径]==
```

就可以在任何目录下执行程序了

6. exec 配合 fork 使用

一个进程要执行一个不同的程序。这对 shell 是常见的情况。在这种情况下，子进程从 fork 返回后立即调用 exec。

1. 不用 exec 的方法： 实现功能，当父进程检查到输入为 1 的时候，创建子进程把配置文件的字段值修改掉。

```c
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>

int main()
```

```
{
    pid_t pid;
    int data = 10;

    while(1){
            printf("please input a data\n");
            scanf("%d",&data);
            if(data == 1){
                    pid = fork();
                    if(pid>0)
                    {
                      wait(NULL);
                    }
                    if(pid == 0){
                            int fdSrc;
                            char *readBuf=NULL;
                             fdSrc = open("config.txt",O_RDWR);
                             int size = lseek(fdSrc,0,SEEK_END);
                             lseek (fdSrc,0,SEEK_SET);

                             readBuf =(char *)malloc(sizeof(char)*size+8);
                             int n_read= read(fdSrc,readBuf,size);
                             char *p=strstr(readBuf,"LENG="); //找到（要修改的）位置
  //参数1 要找的源文件  2.“要找的字符串”
                             if(p==NULL){
                                     printf("not found\n");
                                     exit(-1);
                              }
                                p=p+strlen("LENG=");  //移动字符串个字节
                                  *p='0';      //*p  取内容
lseek (fdSrc,0,SEEK_SET);
                            int n_write =write(fdSrc,readBuf,strlen(readBuf));
                            close(fdSrc);
                            exit(0);
                        }

                    }else {
                                    printf("do noting\n");
                             }
            }
                return 0;
}
```

```
please input a data
2
do noting
please input a data
3
do noting
please input a data
1
please input a data
```

实现了当父进程检查到输入为 1 的时候，创建子进程把配置文件的字段值修改掉。

```
CLC@Embed_Learn:~$ vi config.txt
CLC@Embed_Learn:~$ cat config.txt
SPEED=5
LENG=8
SCORE=90
LEVEL=95
CLC@Embed_Learn:~$ cat config.txt
SPEED=5
LENG=0
SCORE=90
LEVEL=95
CLC@Embed_Learn:~$
```


2. 用 exec 的方法： 实现功能，当父进程检查到输入为 1 的时候，创建子进程把配置文件的字段值修改掉。

```c
int main()
{
    pid_t pid;
    int data = 10;

    while(1){
        printf("please input a data\n");
        scanf("%d",&data);
        if(data == 1){
                pid = fork();
                if(pid > 0){
                    wait(NULL);
                }
                if(pid == 0){
```

```
                execl("./changdata","changdata","config.txt",NULL);
                exit(0);
                    }
                }else {
                            printf("do noting\n");
                        }
            }
            return 0;
}
```

```
CLC@Embed_Learn:~$ ./demo4
please input a data
2
do noting
please input a data
3
do noting
please input a data
1
```



```
CLC@Embed_Learn:~$ cat config.txt
SPEED=5
LENG=0
SCORE=90
LEVEL=95
```

使用 execl 和 fork 结合 也能做到上面结果，而且更方便，但是在 ./changdata 可执行文件存在的情况下。

十、system 函数

1. system 函数定义：

函数原型：

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

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：

system()函数的返回值如下： **成功，则返回进程的状态值； 当 sh 不能执行时，返回 127； 失败返回-1；**

2. system 函数的使用：

用 system 也可以做到 execl 的功能 用 system 实现修改配置 数值代码：

```
int main()
{
   pid_t pid;
   int data = 10;
   while(1){
        printf("please input a data\n");
        scanf("%d",&data);
        if(data == 1){
                pid = fork();
                if(pid > 0){
                    wait(NULL);
                }
                if(pid == 0){
                  execl("./changdata config.txt");
                  exit(0);
                    }
                  }else {
                                printf("do noting\n");
                    }
          }
                return 0;
}
```

```
CLC@Embed_Learn:~$ ./demo4
please input a data
1
please input a data
^Z
[5]+  Stopped                 ./demo4
CLC@Embed_Learn:~$ cat config.txt
SPEED=5
LENG=0
SCORE=90
LEVEL=95
```


3. system 和 execl 不同的是：

sysem 运行完调用的可执行文件后还会继续执行源代码。

==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。

十一、popen 函数

1. popen 函数的定义：

函数原型：

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

参数说明：

command：是一个指向以 NULL 结束的 shell 命令字符串的指针。这行命令将被传到 bin/sh 并且使用 -c 标志 ，shell 将执行这个命令。

type： 只能是读或者写中的一种，得到的返回值（标准 I/O 流）也具有和 type 相应 的只读或只写类型。如果 type 是”r“ 则文件指针连接到 command 的标准输出；如果 type 是”w“则文件指针连接到 command 的标准输入。

返回值：

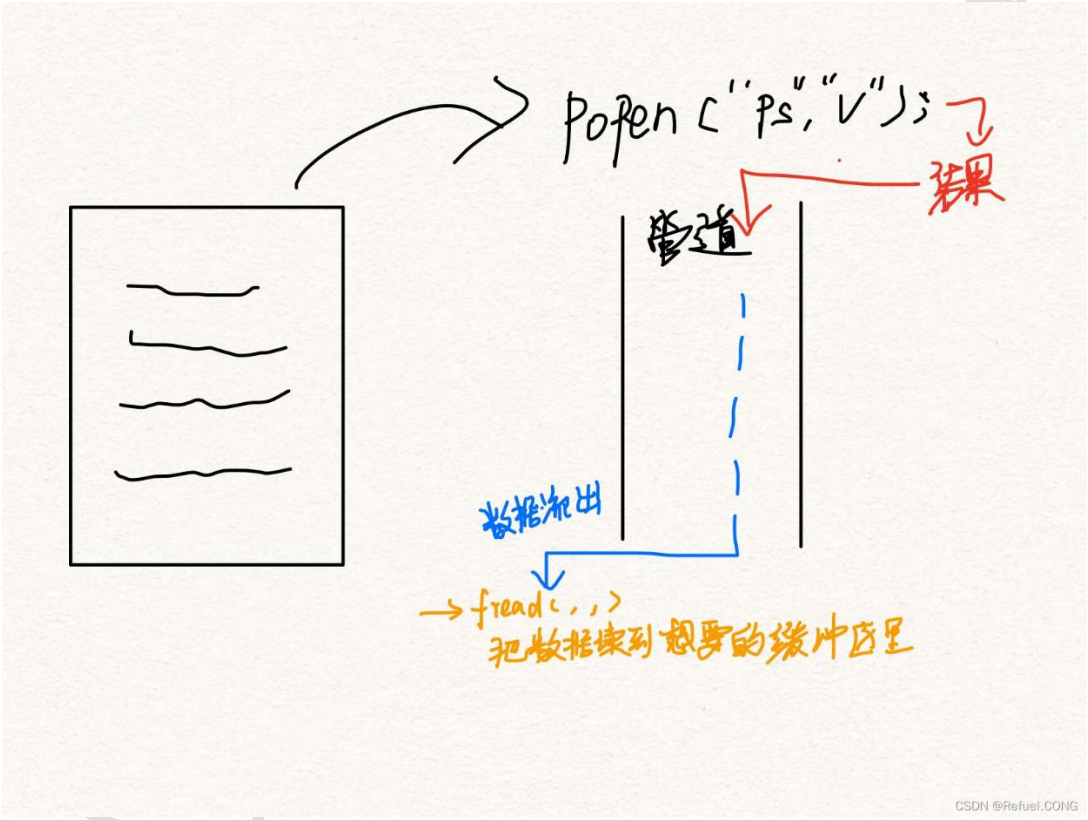
如果调用成功，则返回一个读或者打开文件的指针，如果失败，返回 NULL，具体错误要根据 errno 判断

int pclose（FILE *stream） 参数说明： stream：popen 返回对丢文件指针 返回值： 如果调用失败，返回-1

作用：

popen（）函数用于创建一个管道：其内部实现为调用 fork 产生一个子进程，执行一个 shell 以运行命令来开启一个进程这个进程必须由 pclose（）函数关闭。

popen 比 system 在应用中的好处： ==可以获取运行的输出结果==



popen 函数执行完，执行结果到管道内，数据流出的时候，在管道尾部 fread 就可以读出执行数据，就能实现把数据读到或写到想要的缓冲区里。

2. popen 函数的使用：

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

int main(void)
{
```

```
    char ret[1024]={0};
    FILE *fp;

    fp = popen("ps","r");
    int nread = fread(ret,1,1024,fp);

    printf("read ret %d byte ,ret =%s\n",nread ,ret);
    return 0;
}
```

结果发现： popen 函数结束后，ps 输出的内容， 都捕获到 ret 数组里面去了。 popen 可以获取运行的输出结果 ，可以读取也可以写入文件中。

```
CLC@Embed_Learn:~$ ./popentest
read ret 145 byte ,ret =  PID TTY          TIME CMD
14398 pts/0    00:00:00 bash
14514 pts/0    00:00:00 popentest
14515 pts/0    00:00:00 sh
14516 pts/0    00:00:00 ps
```