

Linux 应用编程

作者：向仔州

Atexit 进程中终止处理函数	2
进程环境变量使用	3
进程 ID 获取	4
Fork 创建父进程和子进程	5
父子进程同时对文件的操作	7
进程的消亡,僵尸进程,孤儿进程	13
进程回收 wait 和 waitpid 函数的区别	14
Exec 函数族	17
进程几种状态	26
System 函数	27
守护进程	28
Syslog 日志写入	30
如何让进程不重复运行	31
无名管道 两个进程间通信	34
有名管道	35
信号的使用	39
消息队列	41
Linux 进程信号量	51
共享内存	56
网络通信流程	59
Linux 高级 IO 使用, 阻塞和非阻塞应用场景	70
Linux 线程使用	78
sem_init 初始化参数问题	85
Linux 线程互斥锁	87
Linux 条件变量解决信号量, 互斥锁处理线程同步问题	89
线程结束时, 执行线程结束子函数	82
互斥锁在有些地方使用的一些问题, 这些问题得用读写锁来解决	93

Atexit 进程中止处理函数

函数原型 `int atexit(void(*func)(void));` 给该函数传入函数指针

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

void functl1(void)
{
    printf("functl1\n");
}

int main(void)
{
    printf("process enter\n");
    atexit(functl1);
    printf("process enter2\n");
    return 0;
}
```

atexit 函数不会按照顺序执行，只有进程执行完成后，才执行该函数，所以你把 atexit 写在进程程序中间也没有用，也是最后执行

```
root@ubuntu:/home/xiang/app/process# ./process
process enter
process enter2
functl1
root@ubuntu:/home/xiang/app/process#
```

如果我们进程结束后一个，atexit 函数可能执行不完里面的程序，我们可以用两个 atexit

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include <unistd.h>
4
5 void functl1(void)
6 {
7     printf("functl1\n");
8 }
9 void functl2(void)
10 {
11     printf("functl2\n");
12 }
13
14 int main(void)
15 {
16     printf("process enter\n");
17     atexit(functl1);
18     atexit(functl2);
19     printf("process enter2\n");
20     return 0;
21 }
```

你会发现 atexit 函数和栈一样是先进后出，也就是你第一个注册的函数不会先执行，会先执行你注册的最后一个 atexit 函数，然后在运行上一个注册的 atexit 函数

```
root@ubuntu:/home/xiang/app/process# ./process
process enter
process enter2
functl2
functl1
root@ubuntu:/home/xiang/app/process#
```

这个 atexit 有个问题，我用 Ctrl+C 信号强制退出该程序，这个 atexit 是不会执行的。只有进程自动执行完成后 atexit 才会执行。下面我们来看看，`return`, `exit` 和 `_exit` 的区别。

```
printf("process enter\n");
atexit(functl1);
printf("process enter2\n");
exit(0);

printf("process enter\n");
atexit(functl1);
printf("process enter2\n");
_exit(0);
```

```
root@ubuntu:/home/xiang/app/process# ./process
process enter
process enter2
functl1
root@ubuntu:/home/xiang/app/process#
```

```
root@ubuntu:/home/xiang/app/process# ./process
process enter
process enter2
functl1
root@ubuntu:/home/xiang/app/process#
```

结论就是，`return` 和 `exit` 是一样的，程序退出会去执行进程中止函数。而 `_exit` 是不会去执行进程中止函数的

进程环境变量使用

比如 PATH 环境变量，该变量就是几个文件和路径指定的变量。使用该变量就使用指定的这些文件

```
root@ubuntu:/home/xiang/app# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/opt/frees
ultilib-2011.12/fsl-linaro-toolchain/bin
root@ubuntu:/home/xiang/app#
```

export 就查看了系统中定义了多少环境变量

```
root@ubuntu:/home/xiang/app# export
declare -x CLUTTER_IM_MODULE="xim"
declare -x COMPIZ_CONFIG_PROFILE="ubuntu"
declare -x DBUS_SESSION_BUS_ADDRESS="unix:abstract=/tmp/dbus-L8a
declare -x DEFAUTLS_PATH="/usr/share/gconf/ubuntu.default.path"
declare -x DESKTOP_SESSION="ubuntu"
declare -x DISPLAY=:0"
declare -x GDMSESSION="ubuntu"
declare -x GDM_LANG="en_US"
declare -x GNOME_DESKTOP_SESSION_ID="this-is-deprecated"
declare -x GNOME_KEYRING_CONTROL=""
declare -x GNOME_KEYRING_PID=""
declare -x GPG_AGENT_INFO="/home/xiang/.gnupg/S.gpg-agent:0:1"
declare -x GTK2_MODULES="overlay-scrollbar"
declare -x GTK_IM_MODULE="ibus"
declare -x GTK_MODULES="gail:atk-bridge:unity-gtk-module"
declare -x HOME="/root"
declare -x IM_CONFIG_PHASE="1"
declare -x INSTANCE="Unity"
declare -x JOB="gnome-session"
declare -x LANG="en_US.UTF-8"
declare -x LANGUAGE="en_US"
declare -x LESSCLOSE="/usr/bin/lesspipe %s %s"
declare -x LESSOPEN="| /usr/bin/lesspipe %s"
declare -x LOGNAME="root"
declare -x LS_COLORS="rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=0
;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31
;01:31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;3
*.*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=
.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z
*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xp
25:*.pcx=01;35:*.pcd=01;35:*.pcn=01;35:*.pcv=01;35:*.pct=01;35:*
```

但是这些环境变量又可以放在另外一个变量中归纳起来，比如 environ 变量

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

void main()
{
    extern char **environ; //该变量定义就可以直接使用，不需要赋值，因为它是系统环境变量
    int i=0;
    while(NULL!=environ[0]){
        printf("%s\n",environ[i]);
        i++;
    }
}
```

```
root@ubuntu:/home/xiang/app/environ# ./environ
XDG_VTNR=7
XDG_SESSION_ID=c2
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/xiang
CLUTTER_IM_MODULE=xim
GPG_AGENT_INFO=/home/xiang/.gnupg/S.gpg-agent:0:1
VTE_VERSION=4205
TERM=xterm-256color
SHELL=/bin/bash
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=58720266
GNOME_KEYRING_CONTROL=
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/7639
GTK_MODULES=gail:atk-bridge:unity-gtk-module
USER=root
QT_ACCESSIBILITY=1
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33:01:cd=4
2:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz
z=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.dz=0
1:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01
31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01
```

environ 是不是和 export 一样是个环境变量，而且最重要的一点是该环境变量可以被 C 语言代码直接调用，所以在 C 语言里面定义其它变量的时候一定要注意不要和系统的环境变量重合了，或者定义的时候加 static，不要加 extern 外部声明

所以我们的程序中可以无条件使用操作系统的环境变量，你的程序一旦使用了操作系统的环境变量，那么你的程序就和操作系统的环境有关了。

进程 ID 获取

getpid 是获取本次运行程序的进程号，getppid 是获取父进程的进程号。

```
1 #include<stdio.h>
2 #include<sys/types.h> //获取进程ID函数必须包含这个头文件
3 #include<unistd.h>    //获取进程ID函数必须包含这个头文件
4 #include<stdlib.h>     //这个是exit函数需要的头文件
5 void main()
6 {
7     pid_t p1=-1,p2=-1;
8
9     printf("PID...\n");
10    p1=getpid(); //获取本身这个程序的进程号
11    printf("pid = %d\n",p1);
12    p2=getppid(); //获取调用我这个程序的进程号，也就是父进程号
13    printf("parent = %d \n",p2);
14    exit(0);
15 }
```

执行后是下面这样，`./(file)`这个命令其实就是父进程程序

```
root@ubuntu:/home/xiang/app/PID# ./getpid
PID...
pid = 10927
parent = 10326
root@ubuntu:/home/xiang/app/PID#
```

10927 进程 ID 就是子进程
10326 进程 ID 就是父进程

我们 ps 之后发现父进程就是一个 bash 进程

```
root@ubuntu:/home/xiang/app/PID# ps
 PID TTY          TIME CMD
10325 pts/18    00:00:00 su
10326 pts/18    00:00:00 bash
10939 pts/18    00:00:00 ps
root@ubuntu:/home/xiang/app/PID#
```

这个`./getpid` 其实就是 bash
进程调用创建 getpid 进程
所以 getpid 进程是 bash 进程调用创建的。

然后进程执行完成后是不会回收的，如果再次重复执行该进程，那么进程 ID 会跟着加 1；

```
root@ubuntu:/home/xiang/app/PID# ./getpid
PID...
pid = 10958
parent = 10326
root@ubuntu:/home/xiang/app/PID# ./getpid
PID...
pid = 10959
parent = 10326
root@ubuntu:/home/xiang/app/PID# ./getpid
PID...
pid = 10960
parent = 10326
root@ubuntu:/home/xiang/app/PID# ./getpid
PID...
pid = 10961
parent = 10326
root@ubuntu:/home/xiang/app/PID#
```

你看我不停的重复调用 getpid 进程，进程号在不断的增加，而不是继续使用上次进程的进程号，所以进程执行完成后操作系统没有回收进程号，而是开辟新的进程号。Bash 进程号没变是因为 bash 进程一直在运行没有结束，你把 bash 进程结束了试试，绝对会变

Fork 创建父进程和子进程

Linux 创建进程的方法是拿老进程复制一份新进程出来。

用 fork 函数创建进程。fork 函数还有个重要的地方就是 fork 执行一次，返回值返回两次。

fork()=0 表示子进程，fork()>0 表示父进程，fork()<0 表示进程创建失败。

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<unistd.h> //fork函数需要包含这个头文件
4 #include<sys/types.h>
5
6 void main()
7 {
8     pid_t p1=-1;
9     p1=fork();
10    if(p1==0)
11    {
12    }
13    if(p1>0)
14    {
15    }
16    if(p1<0)
17    {
18    }
19    exit(0);
20 }
```

我们看到当
fork 之后就会
复制一份父进
程的代码到子
进程

P1=fork();

if(p1=0)

if(p1>0)

if(p1<0)

父进程

if(p1=0)

if(p1>0)

子进程

我们可以加代码运行试试

```
void main()
{
    pid_t p1=-1;
    p1=fork();
    if(p1==0)
    {
    }
    if(p1>0)
    {
    }
    if(p1<0)
    {
    }
    printf("fork run... pid = %d\n",getpid());
    exit(0);
}
```

你看我明明写的一个 printf，但
是运行了两次 printf，证明复制
了一个子进程和父进程在同时运
行同样的代码

```
t@ubuntu:/home/xiang/app/flock# ./fork
fork run... pid = 12065
fork run... pid = 12066
t@ubuntu:/home/xiang/app/flock#
```

这个子进程
就是复制的
代码

所以我们没有办法区分我们这个代码到底是在父进程还是在子进程，有时候我们希望代码在父进程，有时候我们希望代码在子进程。

```
void main()
{
    pid_t p1=-1;
    p1=fork();
    if(p1==0)
    {
        printf("sub process .. pid = %d\n",getpid());
    }
    if(p1>0)
    {
        printf("parent process .. pid = %d\n",getpid());
    }
    if(p1<0)
    {
    }
    exit(0);
}
```

我们把子进程代码放到 if(p1<0)里

把父进程代码放在 if(p1>0)里，这样就解决了进程
使用问题。修改子进程代码不会像上面那样影响父
进程

```
root@ubuntu:/home/xiang/app/flock# ./fork
parent process .. pid = 12208
sub process .. pid = 12209
root@ubuntu:/home/xiang/app/flock# ./fork
parent process .. pid = 12210
sub process .. pid = 12211
root@ubuntu:/home/xiang/app/flock#
```

这个 fork 的返回值到底是上面样的？我们来看看，p1=fork()；p1 到底等于多少

我们说 p1=0 是子进程，P1>0 是父进程，但是这个 P1 大于 0 是大于好多呢？

```
pid_t p1=-1;
p1=fork();

if(p1==0)
{
    printf("sub process .. pid = %d\n",getpid());
}

if(p1>0)
{
    printf("parent process .. pid = %d\n",getpid());
    printf("parent process p1.. pid = %d\n",p1);
}

if(p1<0)
{
}

...,
```

我们不难发现，P1>0，其实就是 P1 等于子进程的 ID

```
root@ubuntu:/home/xiang/app/fock# ./fork
parent process .. pid = 12297
parent process p1.. pid = 12298
sub process .. pid = 12298
```

我们在子进程查看下父进程的 ID 是多少

```
pid_t p1=-1;
p1=fork();

if(p1==0)
{
    printf("sub process .. pid = %d\n",getpid());
    printf("sub process cat parentid .. ppid = %d\n",getppid());
}

if(p1>0)
{
    printf("parent process .. pid = %d\n",getpid());
    printf("parent process p1.. pid = %d\n",p1);
}

if(p1<0)
{
}

...,
```

```
root@ubuntu:/home/xiang/app/fock# ./fork
parent process .. pid = 12353
sub process .. pid = 12354
sub process cat parentid .. ppid = 12353
parent process p1.. pid = 12354
root@ubuntu:/home/xiang/app/fock# ./fork
parent process .. pid = 12355
parent process p1.. pid = 12356
sub process .. pid = 12356
sub process cat parentid .. ppid = 7639
```

我们发现大部分时候子进程查看父进程 ID 和父进程查看自己的 ID 是一样的，但是有小部分时候子进程查看父进程 ID 和父进程自己查看的 ID 不一样。 这是个原因是要看父进程和子进程谁先死，如果父进程先死，那么父进程权限就会移交给 init 进程做为父进程，所以这时候子进程查看到的父进程号是 7639 init 进程号。如果子进程先死，父进程后死，那么这个问题就不会出现，就是正常 ID 号。

父子进程同时对文件的操作

void perror (const char * str); 函数用来将上一个函数发生错误打印出来, *str 是要打印字符串的头字符, 打印后面还要跟系统文件字符

```
1 #include<stdio.h>      //perror会用到该头文件
2 #include<stdlib.h>
3 #include<sys/types.h>   //open会用到该头文件
4 #include<sys/stat.h>    //open会用到该头文件
5 #include<fcntl.h>        //open会用到该头文件
6 #include<unistd.h>      //fork会用到该头文件
7 int main(void)
8 {
9     int fd = -1;
10    pid_t pid;
11
12    fd=open("1.txt",O_RDWR,O_TRUNC);
13    if(fd<0)
14    {
15        perror("iii");
16        return -1;
17    }
18}
```

没有实现创建 1.txt 文件, 就会 open 错误然后 perror 就可以报出来, 当然也可以用 printf, 看你自己

```
root@ubuntu:/home/xiang/app/process_creatfile# ./process_creat
iii: No such file or directory
```

perror 里面传入的字符头就是打印冒号前面的字符, 冒号后面的字符是系统输出的。

```
#include<sys/stat.h> //open会用到该头文件
#include<fcntl.h>      //open会用到该头文件
#include<unistd.h>      //fork会用到该头文件
int main(void)
{
    int fd = -1;
    pid_t pid;

    fd=open("1.txt",O_RDWR,O_TRUNC);
    if(fd<0)
    {
        perror("open");
        return -1;
    }

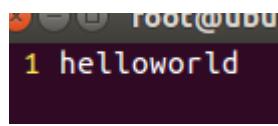
    pid=fork();
    if(pid>0)    //父进程
    {
        printf("parent\n");
        write(fd,"hello",5);
    }
    if(pid==0)   //子进程
    {
        printf("sub\n");
        write(fd,"world",5);
    }
    if(pid<0)
    {
        perror("fork");
        return -1;
    }

    close(fd);
    return 0;
}
```

执行代码

```
root@ubuntu:/home/xiang/app/process_creatfile# ./process_creat
parent
sub
root@ubuntu:/home/xiang/app/process_creatfile#
```

文件输出



这是正常输出 helloworld

再次执行

```
root@ubuntu:/home/xiang/app/process_creatfile# ./process_creat
parent
sub
root@ubuntu:/home/xiang/app/process_creatfile# ./process_creat
parent
sub
```

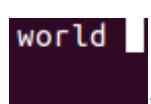
文件输出



这是不正常输出

再次执行

```
root@ubuntu:/home/xiang/app/process_creatfile# ./process_creat
parent
sub
root@ubuntu:/home/xiang/app/process_creatfile# ./process_creat
parent
sub
root@ubuntu:/home/xiang/app/process_creatfile# ./process_creat
parent
sub
```



这是不正常输出

以上输出一会儿正确一会儿不正确这种问题就是进程调度造成的，怎么理解呢？

```
#include<sys/stat.h> //open会用到该头文件
#include<fcntl.h>      //open会用到该头文件
#include<unistd.h>     //fork会用到该头文件
int main(void)
{
    int fd = -1;
    pid_t pid;

    fd=open("1.txt",O_RDWR,O_TRUNC);
    if(fd<0)
    {
        perror("open");
        return -1;
    }

    pid=fork();           ← 当进程创建之后，我的 CPU 如果先执行的父进程，父进程执行完成后
    if(pid>0)    //父进程
    {
        printf("parent\n");
        write(fd,"hello",5);

    }
    if(pid==0) //子进程
    {
        printf("sub\n");
        write(fd,"world",5);

    }
    if(pid<0)
    {
        perror("fork");
        return -1;
    }

    close(fd); ←
    return 0;
}
```

当进程创建之后，我的 CPU 如果先执行的父进程，父进程执行完成后

并不是按照规矩去执行子进程，而是直接就执行
close(fd)进程结束，然后 CPU 再去调用子进程，但是这
个时候文件已经关闭了，子进程往文件里面写不了东
西，所以就丢失了，为什么只有 hello 没有 world 就是这
个原因，也有可能 CPU 先执行子进程，然后再执行父进
程，那么就会出现子进程执行结束直接 close，这样就只
有 world 没有 hello，所以进程那个先执行是 CPU 的调
度，但是问题就是这样出现的

结论：两个进程往同一个文件里面写入数据时，一定要做安全操作处理。

为了解决上面所谓的安全问题，我们用父进程和子进程都来打开文件

```

int main(void)
{
    int fd = -1;
    pid_t pid;

    pid=fork();
    if(pid>0)      //父进程
    {
        fd=open("1.txt",O_RDWR); //在父进程打开OPEN
        if(fd<0)
        {
            perror("open");
            return -1;
        }
        printf("parent\n");
        write(fd,"hello",5);
    }

    if(pid==0)    //子进程
    {
        fd=open("1.txt",O_RDWR); //在子进程打开OPEN
        if(fd<0)
        {
            perror("open");
            return -1;
        }
        printf("sub\n");
        write(fd,"world",5);
    }

    if(pid<0)
    {
        perror("fork");
        return -1;
    }

    close(fd);
    return 0;
}

```

该程序不会出现上面的问题，该程序就是在每个进程打开文件，然后写入。因为每个进程都是单独 OPEN 的所以不会存在一个进程打开文件然后关闭了，另外一个进程再次写这个文件，但是文件没有先打开，数据写不进去的问题。

为什么每个进程里面没有 close，而是要共享最后一个 close 呢？因为 fork 之后，操作系统就复制了一份同样的程序给子进程，所以子进程是有 close 的，只是你看不见，你就可以把主进程的 close 看成执行两次就对了。也可以为了直观在每个进程里面加 close

程序执行

```

root@ubuntu:/home/xiang/app/process_creatfile2# ./process_creat2
parent
sub
root@ubuntu:/home/xiang/app/process_creatfile2#

```

执行后的 txt 文件

1 world

为什么是 world，那么 hello 呢，你看 OPEN 里用的属性是 O_RDWR，这个属性是读写不是 O_APPEND 所以每次执行程序，打开 txt 文件后不是在文件已有的字符后面增加，而是直接从开头再写。

我们可以先修改 txt 文件，在文件里面加很多 1

```
root@ubuntu: ~
1 11111111111111
~
```

然后再次执行程序

```
root@ubuntu:/home/xiang/app/process_creatfile2# ./process_creat2
parent
sub
root@ubuntu:/home/xiang/app/process_creatfile2#
```

txt 文件得到的结果是

```
root@ubuntu: ~
1 world11111111
```

因为 open 没有 O_APPEND 追加属性，所以只有从 txt 文件第一个开始写入。而不是接着 1 后面继续写 world，如果要接着写就需要用 O_APPEND

```
pid=fork();
if(pid>0)    //父进程
{
    fd=open("1.txt",O_RDWR|O_APPEND); //在父进程打开OPEN
    if(fd<0)
    {
        perror("open");
        return -1;
    }
    printf("parent\n");
    write(fd,"hello",5);
}

if(pid==0)  //子进程
{
    fd=open("1.txt",O_RDWR|O_APPEND); //在子进程打开OPEN
    if(fd<0)
    {
        perror("open");
        return -1;
    }
    printf("sub\n");
    write(fd,"world",5);
}

if(pid<0)
{
    perror("fork");
    return -1;
}
```

加入了 O_APPEND 属性

第一次执行程序

```
root@ubuntu:/home/xiang/app/process_creatfile2# ./process_creat2
parent
sub
root@ubuntu:/home/xiang/app/process_creatfile2#
```

```
1 world1111111  
2 helloworld
```

你看就是接受上面 1111111 下面增加了一行

再次执行

```
root@ubuntu:/home/xiang/app/process_createfile2# ./process_createfile2  
arent  
ub  
root@ubuntu:/home/xiang/app/process_createfile2# ./process_createfile2  
arent  
ub
```

你看就在后面接着增加 helloworld

```
1 world1111111  
2 helloworldhelloworld
```

这就是有 O_APPEND 和没有 O_APPEND 的区别。

还有一个属性叫 O_TRUNC

```
int main(void)
{
    int fd = -1;
    pid_t pid;

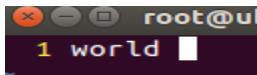
    pid=fork();
    if(pid>0)      //父进程
    {
        fd=open("1.txt",O_RDWR|O_TRUNC); //在父进程打开OPEN
        if(fd<0)
        {
            perror("open");
            return -1;
        }
        printf("parent\n");
        write(fd,"hello",5);
    }

    if(pid==0)    //子进程
    {
        fd=open("1.txt",O_RDWR|O_TRUNC); //在子进程打开OPEN
        if(fd<0)
        {
            perror("open");
            return -1;
        }
        printf("sub\n");
        write(fd,"world",5);
    }
}
```

这是执行前的 txt 文件

这是执行程序后的 txt 文件

```
root@ubuntu:/home/xiang/app/process_creatfile2# ./process_creat2
parent
sub
root@ubuntu:/home/xiang/app/process_creatfile2#
```



进程的消亡

进程在运行过程中打开了文件和 IO 设备，但是在执行到一定的时候突然进程异常中止了，这时候进程虽然退出了，但是打开的文件或者 IO 设备并未有释放掉，这样就形成了设备丢失，其它程序再访问同样的文件或者设备的时候就会报错。所以进程结束我们一定要回收申请的资源，至于怎么回收看下面。

僵尸进程

子进程比父进程先结束

子进程在运行过程中 malloc 申请了内存，打开了文件，然后子进程结束了之后操作系统会自动帮你释放内存和关闭设备，所以你忘记释放内存或者设备都没有关系。但是有一点就是 fork 子进程之后，子进程自己占用的 8K 内存空间是没有清理掉的。**所以进程本身是需要回收的。**

父进程可以用 wait 或者 waitpid 来回收子进程。

父进程也可以不用 wait 和 waitpid 回收子进程 8K 内存。父进程自己结束了，操作系统会自动回收子进程

孤儿进程

父进程先死子进程还没有死。这时候子进程就是孤儿进程了。

孤儿进程的处理方法是操作系统自动给子进程也就是孤儿进程找个父亲，这个父亲就是 init 进程。

```
void main()
{
    pid_t p1=-1;
    p1=fork();
    if(p1==0)
    {
        printf("sub process .. pid = %d\n",getpid());
        printf("sub process cat parentid .. ppid = %d\n",getppid());
    }
    if(p1>0)
    {
        printf("parent process .. pid = %d\n",getpid());
        printf("parent process p1.. pid = %d\n",p1);
    }
    if(p1<0)
    {
        }
    exit(0);
}
```

正常情况下父进程和子进程都没有中止

```
root@ubuntu:/home/xiang/app/fock# ./fork
sub process .. pid = 9397
sub process cat parentid .. ppid = 9396
parent process .. pid = 9396
parent process p1.. pid = 9397
root@ubuntu:/home/xiang/app/fock#
```

你会发现父进程是 9396，子进程是 9397，子进程在父进程后面一点点

```
void main()
{
    pid_t p1=-1;
    p1=fork();
    if(p1==0)
    {
        sleep(1); //这里的延时是让父进程比子进程早执行完，让子进程成为孤儿进程
        printf("sub process .. pid = %d\n",getpid());
        printf("sub process cat parentid .. ppid = %d\n",getppid());
    }

    if(p1>0)
    {

        printf("parent process .. pid = %d\n",getpid());
        printf("parent process p1.. pid = %d\n",p1);

    }
    if(p1<0)
    {

    }
    exit(0);
}
```

当我将子进程延时，让父进程先执行完。这时候子进程就是孤儿进程了。

```
root@ubuntu:/home/xiang/app/fock# ./fork
parent process .. pid = 9416
parent process p1.. pid = 9417
root@ubuntu:/home/xiang/app/fock# sub process .. pid = 9417
sub process cat parentid .. ppid = 7698
root@ubuntu:/home/xiang/app/fock#
```

操作系统就自动给子进程找了个父进程 7698，也就是 init 进程。

进程回收 wait 和 waitpid 函数的区别

Wait 可以不指定进程号默认回收子进程

Waitpid 可以指定进程号，或者不指定进程号回收进程，而且还有阻塞和非阻塞功能。

函数原型 `pid_t wait(int *status);`

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

`pid_t`: 返回本次回收子进程的 PID

`pid_t pid`: 指定回收哪个 ID 号的子进程，如果传入 -1 就是和 wait 一样自动选择回收的进程

```
int *status
```

int options : 这个选项用来选择阻塞或非阻塞, WNOHANG 为非阻塞, 0 为阻塞

这里重点讲一下 `pid_t=waitpid()` 如果返回-1 表示没有这个进程, 如果返回 0 表示子进程还在执行没有结束, 如果返回>0 也就是子进程 ID 号就表示子进程运行结束。

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>//使用waitpid, 这个头文件必须包好

void main()
{
    pid_t p1=-1;
    pid_t ret=-1;
    int status;

    p1=fork();
    if(p1>0)
    {
        ret=waitpid(-1,&status,0);//阻塞方式回收子进程
        printf("parent huisou subprocess PID = %d\n",ret);
        printf("subprocess status = %d\n",WIFEXITED(status));//WIFEXITED(status) 如果进程子进程正常结束, 返回一个非零值
    }
    if(p1==0)
    {
        printf("subprocess = %d\n",getpid());
    }
    if(p1<0)
    {

    }
    exit(0);
}
```

执行结果为

```
root@ubuntu:/home/xiang/app/wait# ./wait
subprocess = 47239
parent huisou subprocess PID = 47239
subprocess status = 1
root@ubuntu:/home/xiang/app/wait#
```

回收的子进程号

```

p1=fork();
if(p1>0)
{
    ret=waitpid(p1+4,&status,0);//阻塞方式回收子进程
    printf("parent huisou subprocess PID = %d\n",ret);
    printf("subprocess status = %d\n",WIFEXITED(status));//WIFEXITED(status) 如果进程子进程正常结束，返回一个非零值
}
if(p1==0)
{
    printf("subprocess = %d\n",getpid());
}
if(p1<0)
{
}

exit(0);

```

在 fork 后 p1 其实就是子进程号，现在要求回收 p1+4 的进程

```

root@ubuntu:/home/xiang/app/wait# ./wait
parent huisou subprocess PID = -1
subprocess status = 1
subprocess = 47266
root@ubuntu:/home/xiang/app/wait#

```

没有这个子进程号可以回收

```

void main()
{
    pid_t p1=-1;
    pid_t ret=-1;
    int status;
    p1=fork();
    if(p1>0)
    {
        while(1){
            ret=waitpid(-1,&status,0);//阻塞方式回收子进程

            printf("parent huisou subprocess PID = %d\n",ret);
            printf("subprocess status = %d\n",WIFEXITED(status));//WIFEXITED(status) 如果进程子进程正常结束，返回一个非零值
            sleep(1);
        }
    }

    if(p1==0)
    {
        while(1){
            printf("subprocess = %d\n",getpid());
            sleep(1);
        }
    }
    if(p1<0)
    {

    }
    exit(0);
}

```

阻塞方式回收进程，子进程没结束，父进程卡在这里

执行后只打印了子进程，父进程卡在 waitpid 哪里了

```

root@ubuntu:/home/xiang/app/wait# ./wait
subprocess = 8815
^C
root@ubuntu:/home/xiang/app/wait#

```

```
pid_t p1=-1;
pid_t ret=-1;
int status;

p1=fork();
if(p1>0)
{
    while(1){
        ret=waitpid(-1,&status,WNOHANG); //阻塞方式回收子进程

        printf("parent huisou subprocess PID = %d\n",ret);
        printf("subprocess status = %d\n",WIFEXITED(status));
        sleep(1);
    }
}

if(p1==0)
{
    while(1){
        printf("subprocess = %d\n",getpid());
        sleep(1);
    }
}

if(p1<0)
{

}
exit(0);

```

```
root@ubuntu:/home/xiang/app/wait# gcc -o wait wait.c
root@ubuntu:/home/xiang/app/wait# ./wait
parent huisou subprocess PID = 0
subprocess status = 1
subprocess = 8848
parent huisou subprocess PID = 0
subprocess status = 1
subprocess = 8848
parent huisou subprocess PID = 0
subprocess status = 1
subprocess = 8848
parent huisou subprocess PID = 0
subprocess status = 1
subprocess = 8848
parent huisou subprocess PID = 0
subprocess status = 1
subprocess = 8848
```

为什么父进程回收的是 0，那是因为子进程还没有结束

```
p1=fork();
if(p1>0)
{
    while(1){
        ret=waitpid(-1,&status,WNOHANG); //阻塞方式回收子进程

        printf("parent huisou subprocess PID = %d\n",ret);
        printf("subprocess status = %d\n",WIFEXITED(status));//W
        sleep(1);
    }
}

if(p1==0)
{
    printf("subprocess = %d\n",getpid());
    sleep(1);
}

if(p1<0)
```

```
root@ubuntu:/home/xiang/app/wait# ./wait
parent huisou subprocess PID = 0
subprocess status = 1
subprocess = 8888
parent huisou subprocess PID = 0
subprocess status = 1
parent huisou subprocess PID = 8888
subprocess status = 1
parent huisou subprocess PID = -1
subprocess status = 1
parent huisou subprocess PID = -1
```

父进程成功回收子进程，
后面父进程再回收就是-1
了，因为没有子进程可以
回收了

Exec 函數族

Exec 函数族的优点在于，你的多进程程序不用再一个 C 文件里面全部实现，也就是说我不用再 if(PID>0) 父进程里面写很多函数，也不用再 if (PID==0) 子进程里面写很多函数，如果是前面讲的那样，看起来一个 C 文件就很乱了。代码太多。我们可以把我们要执行的子进程代码写在另外一个 C 文件里面，然后

编译成可执行程序，然后在主程序 C 文件里面，创建进程，然后在进程里面用 exec 函数去调用其他执行文件来充当子进程的角色，这样代码看起来就要好看得多。结论就是不管用不用 exec 函数族来做多进程，结果是一样的，只是看你方不方便管理。

```
1 #include<stdio.h>
2 #include<sys/types.h>
3 #include<unistd.h>
4 #include<stdlib.h>
5 #include<sys/wait.h> //使用waitpid，这个头文件必须包好
6
7 void main()
8 {
9     pid_t p1=-1;
10    pid_t ret=-1;
11    int status;
12
13    p1=fork();
14    if(p1>0) //父进程
15    {
16        .
17        .
18        .
19        .
20        .
21        .
22        .
23        .
24    }
25    if(p1==0) //子进程
26    {
27        .
28        .
29        .
30        .
31        .
32        .
33        .
34        .
35    }
36    if(p1<0)
37    {
38        .
39        .
40    }
41    exit(0);
42
43 }
```

这是没有 exec 函数族的程序，一个 C 文件很繁杂

有 exec 函数族的程序

```
root@ubuntu:/home/xiang/app/exec# ls  
1...c 2...c  
root@ubuntu:/home/xiang/app/exec#
```

用多个 C 文件来写繁杂的程序

```
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h> //使用waitpid, 这个头文件必须包好

void main()
{
    pid_t p1=-1;
    pid_t ret=-1;
    int status;

    p1=fork();
    if(p1>0) //父进程
    {
        execl(1...); ——————
    }
    if(p1==0) //子进程
    {
        execl(2...); ——————
    }
    if(p1<0)
    {

    }
    exit(0);
}
```

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

main()
{
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
}
```

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<unistd.h>
4 #include<stdlib.h>
5 #include<sys/wait.h>
6
7
8 main()
9 {
10     .....c
11     .....
12     .....
13     .....
14     .....
15     .....
16     .....
17     .....
18     .....
19 }
```

这样主进程只负责调用子进程的文件，主进程不做其他功能

这样主进程代码看起来就要舒服很多

比如说我想在我的子进程也就是 C 文件里面执行 shell 命令,

怎么办? exec 函数族就是把编译好的可执行程序直接加载运行

exec 函数:

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

*path 参数: 是我们要执行的程序在哪个位置

*arg 参数: 我们要执行的程序

.....参数: 我们要传入的参数, 可以传入很多个参数, 用 NULL 结尾证明参数已经传完。

我们以 ls 命令为例子来看

```
1 #include<stdio.h>  
2 #include<stdlib.h>  
3 #include<unistd.h> // 这个头文件exec函数族会用到  
4 #include<sys/types.h>  
5  
6  
7 void main()  
8 {  
9     pid_t p1=-1;  
10    p1=fork();  
11    if(p1==0)  
12    {  
13        execl("/bin/ls","ls",NULL);  
14    }  
15    if(p1>0)  
16    {  
17    }  
18    if(p1<0)  
19    {  
20    }  
21    exit(0);  
22 }
```

```
root@ubuntu:/home/xiang/app/exec# which ls  
/bin/ls
```

我们先找到 ls 命令在 linux 系统的路径

```
root@ubuntu:/home/xiang/app/exec# ./exec  
root@ubuntu:/home/xiang/app/exec# exec exec.c
```

这就是执行 exec 程序的结果 就是个 ls

然后我们给 ls 加参数

```
void main()  
{  
    pid_t p1=-1;  
    p1=fork();  
    if(p1==0)  
    {  
        execl("/bin/ls","ls","-a",NULL);  
    }  
    if(p1>0)  
    {  
    }  
    if(p1<0)  
    {  
    }  
    exit(0);  
}
```

加了 -a 打印隐藏文件

```
root@ubuntu:/home/xiang/app/exec# ./exec  
root@ubuntu:/home/xiang/app/exec# ... exec exec.c .exec.c.swp
```

这就是和 ls -a 一样

```
execl("/bin/ls","ls","-a",NULL);
```

我们上面说了后面可以跟很多个参数

比如我要列表查询

```
void main()
{
    pid_t p1=-1;
    p1=fork();
    if(p1==0)
    {
        execl("/bin/ls","ls","-a","-l",NULL);
    }
    if(p1>0)
    {
    }
    if(p1<0)
    {
    }
    exit(0);
}
```

又多跟了个-l

```
root@ubuntu:/home/xiang/app/exec# total 36
drwxr-xr-x 2 root root 4096 Jun 2 06:56 .
drwxr-xr-x 10 root root 4096 May 29 22:29 ..
-rwxr-xr-x 1 root root 8704 Jun 2 06:56 exec
-rw-r--r-- 1 root root 270 Jun 2 06:56 exec.c
-rw-r--r-- 1 root root 12288 Jun 2 06:56 .exec.c.swp
```

execv 函数

```
int execv(const char *path, char *const argv[]);
```

*path 参数：是我们要执行的程序在哪个位置

argv[]参数：是将 ls -a -b -c 写在一个数组里面，然后在传入 execv 函数，执行结果和 execl 一样

```
7 void main()
8 {
9
10     pid_t p1=-1;
11     p1=fork();
12     if(p1==0)
13     {
14         char *const arg[]={ "ls","-a","-l",NULL};
15         execv("/bin/ls",arg);
16     }
17
18     if(p1>0)
19     {
20
21         execl("/bin/ls","ls","-a","-l",NULL);
22     }
23     if(p1<0)
24     {
25
26     }
27
28     exit(0);
29 }
```

```
root@ubuntu:/home/xiang/app/exec# ./exec
total 36
drwxr-xr-x 2 root root 4096 Jun 2 07:23 .
drwxr-xr-x 10 root root 4096 May 29 22:29 ..
-rwxr-xr-x 1 root root 8752 Jun 2 07:23 exec
-rw-r--r-- 1 root root 336 Jun 2 07:23 exec.c
-rw-r--r-- 1 root root 12288 Jun 2 07:23 .exec.c.swp
root@ubuntu:/home/xiang/app/exec# total 36
drwxr-xr-x 2 root root 4096 Jun 2 07:23 .
drwxr-xr-x 10 root root 4096 May 29 22:29 ..
-rwxr-xr-x 1 root root 8752 Jun 2 07:23 exec
-rw-r--r-- 1 root root 336 Jun 2 07:23 exec.c
-rw-r--r-- 1 root root 12288 Jun 2 07:23 .exec.c.swp
root@ubuntu:/home/xiang/app/exec#
```

你看我将 execl 放在父进程，将 execv 放在子进程，执行结果是一样的。根据你爱好选择一个函数族。
用 execl 来执行自己的程序

首先创建一个自己的应用程序，然后编译成可执行文件

```
1 #include<stdio.h>
2
3 void main()
4 {
5
6     printf("\n");
7     printf("hello world\n");
8 }
```

用 GCC 编译 hello 文件，这样在目录下就有两个文件了

```
root@ubuntu:/home/xiang/app/exec# ls
exec  exec.c  hello  hello.c
root@ubuntu:/home/xiang/app/exec#
```

```
void main()
{
    pid_t p1=-1;
    p1=fork();
    if(p1==0)
    {
        execl("hello","aaa",NULL); //在用execl执行自己的时候中间必须传入一个参数，否则会出现编译警告
    }

    if(p1>0)
    {

    }
    if(p1<0)
    {

    }
    exit(0);
}
```

在 exec 文件主程序里面
用 execl 函数调用编译
好的 hello 程序文件

执行 exec 程序

```
root@ubuntu:/home/xiang/app/exec#
root@ubuntu:/home/xiang/app/exec# ./exec
root@ubuntu:/home/xiang/app/exec#
hello world
```

前面讲到的向 execl 传参数用在自己程序上是个什么效果？

```
#include<stdio.h>
int main(int argc,char **argv)
{
    int i=0;

    printf("argc = %d\n",argc);//传入多少个参数
    while(NULL!=argv[i])
    {
        printf("argv[%d] = %s\n",i,argv[i]);//每个参数占用一个数组串 argv[0]是打印hello程序本身，程序本身占用一个参数,argv[1]才是打印传入的参数
        i++;
    }

    return 0;
}
```

```
root@ubuntu:/home/xiang/app/exec# ./hello
argc = 1
argv[0] = ./hello
root@ubuntu:/home/xiang/app/exec# ./hello aaa
argc = 2
argv[0] = ./hello
argv[1] = aaa
root@ubuntu:/home/xiang/app/exec#
```

这是没有传入参数的

这是传入参数的

我们来看看用 execl 函数族来向 hello 传入参数是上面效果？

```

void main()
{
    pid_t p1=-1;
    p1=fork();
    if(p1==0)
    {
        execl("hello","aaa",NULL); //这个"aaa"就类似于执行./hello aaa
    }

    if(p1>0)

    }

    if(p1<0)
    {

    }

    exit(0);
}

```

向 hello 程序传入的参数

```

root@ubuntu:/home/xiang/app/exec# ./exec
root@ubuntu:/home/xiang/app/exec# argc = 1
argv[0] = aaa
root@ubuntu:/home/xiang/app/exec#

```

注意一点就是在 execl 里第一个 “aaa” 是在 argv[0]里面，其实这里应该写成 hello，但是都一样。

```

7 void main()
8 {
9     pid_t p1=-1;
10    p1=fork();
11    if(p1==0)
12    {
13        execl("hello","aaa","bbb",NULL); //这个"aaa"就类似于执行./hello aaa
14    }

15    if(p1>0)

16    }

17    if(p1<0)
18    {

19    }

20    exit(0);
21 }

```

向 hello 程序传入多个参数

```

root@ubuntu:/home/xiang/app/exec# argc = 2
argv[0] = aaa
argv[1] = bbb
root@ubuntu:/home/xiang/app/exec#

```

execv 执行自己的程序

```

void main()
{
    pid_t p1=-1;
    p1=fork();
    if(p1==0)
    {
        execl("hello","aaa","bbb",NULL);
    }

    if(p1>0)
    {
        char *const arg[]={ "aaa", "bbb", "ccc", "ddd", NULL };
        execv("hello",arg);
    }

    if(p1<0)
    {

    }

    exit(0);
}

```

其实 execv 和 execl 执行效果是一样的。只是格式区别

```
root@ubuntu:/home/xiang/app/exec# ./exec
argc = 4
argv[0] = aaa
argv[1] = bbb
argv[2] = ccc
argv[3] = ddd
root@ubuntu:/home/xiang/app/exec# argc = 2
argv[0] = aaa
argv[1] = bbb
```

父进程 execv

子进程 execv

exec 的使用

exec 和 exec 的区别是， exec 只需要在 path 上加绝对路径或者相对路径就可以执行，但是不加路径是执行不了的。但是 exec 的功能是在当前目录没找到，可以在 path 指定的路径下去查找，所以不用在代码 path 位置加路径。

```
7 void main()
8 {
9     pid_t p1=-1;
10    p1=fork();
11    if(p1==0)
12    {
13        execl("ls","ls",NULL);
14    }
15
16    if(p1>0)
17    {
18        printf("parent \n");
19    }
20    if(p1<0)
21    {
22    }
23    exit(0);
24 }
```

execl 函数如果把 ls 的路径去掉

```
root@ubuntu:/home/xiang/app/exec# gcc -o exec
root@ubuntu:/home/xiang/app/exec# ./exec
parent
root@ubuntu:/home/xiang/app/exec#
```

父进程执行了，子进程无法执行，是因为没有 ls 路径。

```
execl("/bin/ls","ls",NULL);
```

这样加上路径就可以正常执行。

所以我们用 execp 来解决这个不加路径的问题。

```

6
7 void main()
8 {
9
10     pid_t p1=-1;
11     p1=fork();
12     if(p1==0)
13     {
14         execlp("ls","ls","-l",NULL);
15     }
16
17     if(p1>0)
18     {
19         printf("parent \n");
20     }
21     if(p1<0)
22     {
23
24     }
25     exit(0);
26
27 }

```

```

root@ubuntu:/home/xiang/app/exec# ./exec
parent
root@ubuntu:/home/xiang/app/exec# total 32
-rwxr-xr-x 1 root root 8752 Jun 2 21:17 exec
-rw-r--r-- 1 root root 282 Jun 2 21:17 exec.c
-rwxr-xr-x 1 root root 8608 Jun 2 20:29 hello
-rw-r--r-- 1 root root 343 Jun 2 20:35 hello.c

```

你看 ls 不加路径也能执行，因为 execlp 函数先在环境变量路径下找 ls 命令，如果没有再到当前目录下找

```

root@ubuntu:/home/xiang/app/exec# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/opt/freescale/usr/local/gcc-4.6.2-glibc-2.13-linaro-multilib-2011.1
2/fsl-linaro-toolchain/bin

```

环境变量里面指定的目录 execlp 都会去找，如果环境变量目录和当前目录都没有 ls 命令 execlp 会报错。

execle 函数的使用

```

int execle(const char *path, const char *arg, ... , char * const envp[]);

```

这个函数前面三个参数和 exec 是一样的，就是最后多加了一个 envp[] 环境变量参数

我们先来看看给 main 传入第三个参数

```

void main(int argc,char **argv,char **env) //main程序是可以传入第三个参数的，env是linux系统的环境变量
{
    int i=0;

    printf("argc = %d\n",argc);
    while(NULL!=argv[i])
    {
        printf("argv[%d] = %s\n",i,argv[i]);
        i++;
    }
    i=0;
    while(NULL!=env[i])
    {
        printf("env[%d] = %s\n",i,env[i]);
        i++;
    }
}

```

这个程序编译之后是 main3

根据下面的打印我们发现 env 打印的是系统环境变量包含的值，如果我们给 env 赋值上新的参数，那么系统环境变量将被覆盖

我给 env 赋值上 AA=aaaa BB=abcd, 这时候就可以用 execle 函数

```
void main()
{
    pid_t p1=-1;
    p1=fork();
    if(p1==0)
    {
        char *const envp[]={ "AA=aaaa", "BB=abcd", NULL};
        execle("main3", "main3", "-l", NULL, envp);
    }

    if(p1>0)
    {
        printf("parent \n");
    }
    if(p1<0)
    {

    }
    exit(0);
}
```

环境变量 env 就被修改的，NULL 是环境变量结束标志，这样 env 就不会打印系统的环境变量，执行 main3 程序

-l 是传入 main 函数的 argv

环境变量 env 就被修改的，NULL 是环境变量结束标志，这样 env 就不会打印系统的环境变量，执行 main3 程序

-l 是传入 main 函数的 argv

如果 main3 程序没有传入新的环境变量，那么 env 就执行系统默认的环境变量值，打印结果就和上面一样，但是我们这里修改了 env 环境变量，所以就是我们修改的值。

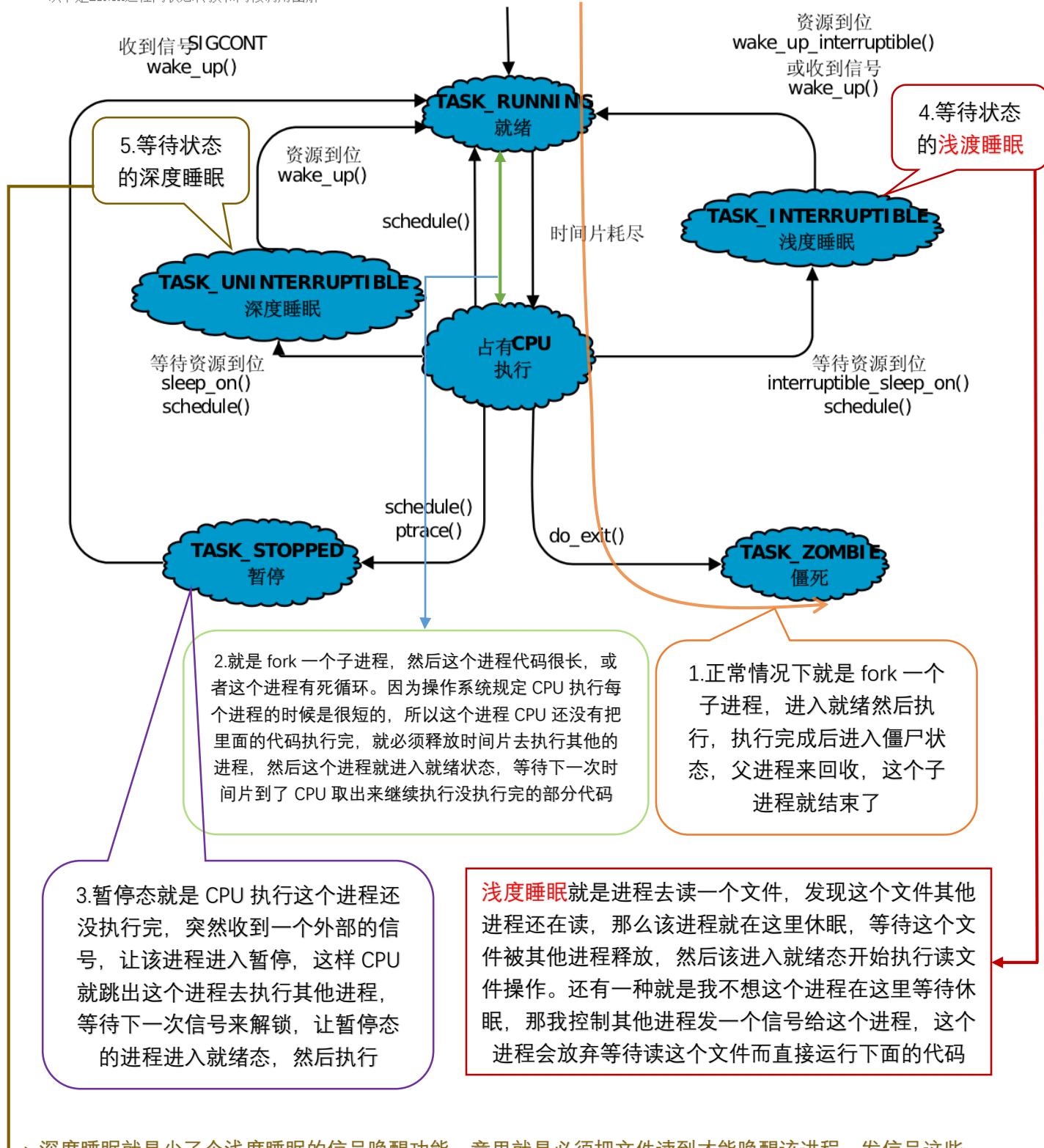
```
root@ubuntu:/home/xiang/app/exec# ./exec
parent
root@ubuntu:/home/xiang/app/exec# argc = 2
argv[0] = main3
argv[1] = -l
env[0] = AA=aaaa
env[1] = BB=abcd
root@ubuntu:/home/xiang/app/exec#
```

环境变量已经被修改

进程几种状态

进程状态有：就绪态，运行态，僵尸态，等待态，暂停态

以下是LINUX进程间状态转换和内核调用图解



→ 深度睡眠就是少了个浅度睡眠的信号唤醒功能，意思就是必须把文件读到才能唤醒该进程，发信号这些提前唤醒对深度睡眠是没有用的，深度睡眠就是一根筋。

System 函数

System 函数=fork+exec 函数

但是为什么要用 system 呢？因为 system 是原子操作，在执行这个函数过程中不会被其他进程打断。不像 fork 之后执行 exec 函数，可能中途会被其他进程占用时间片，但是 system 的坏处还是有的，就是占用 CPU 时间比较长，让其它进程情何以堪，所以自己折中选择。

```
1 #include<stdio.h>
2 #include<stdlib.h> //system函数要包含这个头文件
3
4 void main()
5 {
6
7     system("ls -a");
8 }
```

执行结果

```
root@ubuntu:/home/xiang/app/system# ./system
. .. system system.c
root@ubuntu:/home/xiang/app/system#
```

你也可以多加几个 shell 参数

```
1 #include<stdio.h>
2 #include<stdlib.h> //system函数要包含这个头文件
3
4 void main()
5 {
6
7     system("ls -a -l");
8 }
```

System 函数里面应该有 execlp 这个函数，你看我没有加路径，直接执行 ls 都没有问题，所以 system 应该去 PATH 路径下去找的 ls 命令

```
root@ubuntu:/home/xiang/app/system# ./system
total 24
drwxr-xr-x  2 root root 4096 Jun  3 21:41 .
drwxr-xr-x 11 root root 4096 Jun  3 21:34 ..
-rwxr-xr-x  1 root root  8608 Jun  3 21:41 system
-rw-r--r--  1 root root   115 Jun  3 21:41 system.c
root@ubuntu:/home/xiang/app/system# vim system.c
```

所以 system 就很方便的在 C 语言代码里面调用 shell 命令，然后执行。让 C 语言和 shell 混合使用。

比如要 echo -n “…… 3… ……BCM …” wifi 网卡操作啊 查看系统文件这些就可以用 system

守护进程

先看看 ps 命令

```
root@ubuntu:/home/xiang/app/system# ps
  PID TTY      TIME CMD
8923 pts/0    00:00:00 su
8924 pts/0    00:00:00 bash
9765 pts/0    00:00:01 ps
root@ubuntu:/home/xiang/app/system#
```

运行 ps 不加参数，只能显示你现在打开终端运行的进程，但是还有很多正在运行的进程你看不见

我们来给 ps 加参数

```
root@ubuntu:/home/xiang/app/system# ps -ajx
PPID  PID  PGID  SID  TTY   TPGID STAT   UID   TIME  COMMAND
  0    1    1     1 ?    -1 Ss    0    0:04 /sbin/init auto noprompt
  0    2    0     0 ?    -1 S    0    0:00 [kthreadd]
  2    3    0     0 ?    -1 S    0    0:00 [ksoftirqd/0]
  2    5    0     0 ?    -1 S<   0    0:00 [kworker/0:0H]
  2    7    0     0 ?    -1 S    0    0:06 [rcu_sched]
  2    8    0     0 ?    -1 S    0    0:00 [rcu_bh]
  2    9    0     0 ?    -1 S    0    0:00 [migration/0]
  2   10    0     0 ?    -1 S    0    0:00 [watchdog/0]
  2   11    0     0 ?    -1 S    0    0:00 [watchdog/1]
  2   12    0     0 ?    -1 S    0    0:00 [migration/1]
  2   13    0     0 ?    -1 S    0    0:01 [ksoftirqd/1]
  2   15    0     0 ?    -1 S<   0    0:00 [kworker/1:0H]
  2   16    0     0 ?    -1 S    0    0:00 [kdevtmpfs]
  2   17    0     0 ?    -1 S<   0    0:00 [netns]
  2   18    0     0 ?    -1 S<   0    0:00 [perf]
  2   19    0     0 ?    -1 S    0    0:00 [khungtaskd]
  2   20    0     0 ?    -1 S<   0    0:00 [writeback]
  2   21    0     0 ?    -1 SN   0    0:00 [ksmd]
  2   22    0     0 ?    -1 SN   0    0:02 [khugepaged]
  2   23    0     0 ?    -1 S<   0    0:00 [crypto]
  2   24    0     0 ?    -1 S<   0    0:00 [kintegrityd]
  2   25    0     0 ?    -1 S<   0    0:00 [bioset]
  2   26    0     0 ?    -1 S<   0    0:00 [kblockd]
  2   27    0     0 ?    -1 S<   0    0:00 [ata_sff]
  2   28    0     0 ?    -1 S<   0    0:00 [md]
  2   29    0     0 ?    -1 S<   0    0:00 [devfreq_wq]
  2   33    0     0 ?    -1 S    0    0:00 [kswapd0]
  2   34    0     0 ?    -1 S<   0    0:00 [vmstat]
  2   35    0     0 ?    -1 S    0    0:00 [tsnotify_mark]
  2   36    0     0 ?    -1 S    0    0:00 [ecryptfs-kthrea]
  2   52    0     0 ?    -1 S<   0    0:00 [kthrotld]
  2   53    0     0 ?    -1 S<   0    0:00 [acpi_thermal_pm]
  2   54    0     0 ?    -1 S<   0    0:00 [bioset]
  2   55    0     0 ?    -1 S<   0    0:00 [bioset]
  2   56    0     0 ?    -1 S<   0    0:00 [bioset]
  2   57    0     0 ?    -1 S<   0    0:00 [bioset]
  2   58    0     0 ?    -1 S<   0    0:00 [bioset]
```

PPID 是父进程
PID 是子进程
PGID 是进程组

我们发现父进程 ID 号 2 生成了好多个子进程，ps -ajx 就是查看操作系统的整个正在运行的进程

ps -aux 是查看进程占用 CPU 多少，占用内存多少

```
root@ubuntu:/home/xiang/app/system# ps -aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START  TIME  COMMAND
root        1  0.0  0.2 119564  5688 ?        Ss   20:00  0:04 /sbin/init auto noprompt
root        2  0.0  0.0      0     0 ?        S     20:00  0:00 [kthreadd]
root        3  0.0  0.0      0     0 ?        S     20:00  0:00 [ksoftirqd/0]
root        5  0.0  0.0      0     0 ?        S<   20:00  0:00 [kworker/0:0H]
root        7  0.0  0.0      0     0 ?        S     20:00  0:06 [rcu_sched]
root        8  0.0  0.0      0     0 ?        S     20:00  0:00 [rcu_bh]
root        9  0.0  0.0      0     0 ?        S     20:00  0:00 [migration/0]
root       10  0.0  0.0      0     0 ?        S     20:00  0:00 [watchdog/0]
root       11  0.0  0.0      0     0 ?        S     20:00  0:00 [watchdog/1]
root       12  0.0  0.0      0     0 ?        S     20:00  0:00 [migration/1]
root       13  0.0  0.0      0     0 ?        S     20:00  0:01 [ksoftirqd/1]
root       15  0.0  0.0      0     0 ?        S<   20:00  0:00 [kworker/1:0H]
```

编写一个脱离控制台的守护进程

```
1 #include<stdio.h>
2 #include<unistd.h> //setsid, chdir ,sysconf函数要使用这个头文件
3 #include<stdlib.h>
4 #include<sys/types.h> //umask函数要使用这个头文件
5 #include<sys/stat.h> //umask函数要使用这个头文件
6 #include<fcntl.h> //OPEN函数要使用这个头文件
7
8 void create_daemon() //这个函数作用就是把调用该函数的进程变成守护进程
9 {
10     pid_t pid=-1;
11     pid=fork();
12     if(pid<0)
13     {
14         perror("fork");
15         exit(-1);
16     }
17     if(pid>0)
18     {
19         exit(0);
20     //我fork之后操作系统会复制这个代码给两个进程，但是父进程在这里已经退出了，  

21     //所以下面就算是有代码也无法执行，只有子进程能继续执行下去
22     }
23
24     pid=setsid(); //setsid将我们当前进程设置成会话期，也就是现在这个子进程
25     if(pid<0)
26     {
27
28         perror("setsid");
29         exit(0);
30     }
31
32     chdir("/"); //设置当前进程工作目录
33     umask(0); //确保进程将来创建的文件有最大的权限
34
35     int cnt=sysconf(_SC_OPEN_MAX); //该操作系统最大文件描述符
36     for(int i=0;i<cnt;i++) //关闭操作系统的文件描述符，根据文件号来关闭
37     {
38         close(i);
39     }
40
41     open("/dev/null",O_RDWR);
42     open("/dev/null",O_RDWR);
43     open("/dev/null",O_RDWR);
44 }
45 }
```

```
void main()
{
    create_daemon();
    while(1)
    {
        printf("i am runing\n");
        sleep(1);
    }
}
```

```
root@ubuntu:/home/xiang/app/dbus# ./dbus
root@ubuntu:/home/xiang/app/dbus# ./dbus
```

root	10292	0.0	0.0	4360	84	?	Ss	04:59	0:00	./dbus
root	10294	0.0	0.0	4360	80	?	Ss	04:59	0:00	./dbus

Syslog 日志写入

```
void openlog(const char *ident, int option, int facility);
```

const char *ident : 随便取个应用程序的名字，当然最好取和你现在进程文件名字一样的名字，这样方便在日志查看是哪一个进程写入的这句话，用 NULL 就是默认使用该文件的名字。

```
1 #include<stdio.h>
2 #include<syslog.h>
3
4 void main()
5 {
6     openlog("syslog",LOG_PID|LOG_CONS,LOG_USER);
7     syslog(LOG_INFO,"this my log info");
8     closelog();
9 }
```

想要在 log 文件里面显示的进程名

要写入日志的信息

选择优先级，我就选 LOG_INFO

执行该 log 程序

```
root@ubuntu:/home/xiang/app/syslog# ./syslog
root@ubuntu:/home/xiang/app/syslog#
```

然后查看我们写入的日志，如何查看？

一般来说 log 信息文件在 /var/log/messages 这个 message 文件下保存的。

但是 Ubuntu 系统的 log 文件是 /var/log/syslog 这个 syslog 文件下保存，所以 arm-linux 开发板要注意。

```
root@ubuntu:/var/log# ls
alternatives.log      apport.log.5.gz  bootstrap.log    dpkg.log.3.gz   kern.log.1
alternatives.log.1    apport.log.6.gz  btmp            dpkg.log.4.gz   kern.log.2.gz
alternatives.log.2.gz apport.log.7.gz  btmp.1          faillog       kern.log.3.gz
alternatives.log.3.gz apt             cups            fontconfig.log kern.log.4.gz
apport.log           auth.log        dist-upgrade   fsck          lastlog
apport.log.1          auth.log.1     dmesg          gpu-manager.log lightdm
apport.log.2.gz        auth.log.2.gz  dpkg.log       hp
apport.log.3.gz        auth.log.3.gz  dpkg.log.1    installer
apport.log.4.gz        auth.log.4.gz  dpkg.log.2.gz  kern.log
                                         kern.log.1
```

在 syslog 最后一行就是你现在写入的信息，下次写入就继续在最后一行换行追加。

```
188 Jun  6 07:17:01 ubuntu CRON[9496]: (root) CMD (cd / && run-parts --report /etc/cron.hourly)
189 Jun  6 07:17:36 ubuntu syslog[9510]: this my log info
```

这个是进程名字，其实是人为定义的

这个是写该条 log 信息程序的进程号

如果我将 syslog 改成 log

```
void main()
{
    openlog("log",LOG_PID|LOG_CONS,LOG_USER);
    syslog(LOG_INFO,"this my log info");
    closelog();
}
```

所以这个名字最好和进程文件名一样，否则你不容易记住是哪一个进程写的

```
207 Jun  6 07:27:33 ubuntu log[9604]: this my log info
```

你看我不想再日志中显示进程 PID 我就直接在 openlog 取消 LOG_PID 就行了

```
212 Jun  6 07:32:13 ubuntu log: this my log info
```

一般 syslog 是在任何一个进程的开始执行 openlog，然后在运行过程中不停的写 syslog，然后在进程结束时写 closelog。一个进程的完整运行状态就记录下来了。

如何让进程不重复运行

问题：前面的守护进程我点击几次就在后台生成几个守护进程

```
root@ubuntu:/home/xiang/app/dbus# ./dbus
root@ubuntu:/home/xiang/app/dbus# ./dbus
root      10292  0.0  0.0    4360    84 ?          Ss   04:59   0:00 ./dbus
root      10294  0.0  0.0    4360    80 ?          Ss   04:59   0:00 ./dbus
```

但是我不想同样的程序在后台运行好几个，就像有道一样，打开了一次，关闭窗口后，有道词典后台还在运行，但是再去打开有道词典时，操作系统会提示软件正在运行，而不像 QQ 那样可以打开很多个。
就是要做一个在终端执行一个程序，在另外一个终端无法再执行这个程序，必须前面终端将这个程序执行完，另外一个终端才可以执行这个程序。

具体做法：用一个文件的存在与否来做进程标志。

```
1 #include<stdio.h>
2 #include<fcntl.h>
3 #include<unistd.h>
4 #include<errno.h> //errno变量会用到这个头文件
5
6 #define FILE "./texxxxt"
7
8 void main()
9 {
10     int fd = -1;
11     open(FILE,O_RDWR|O_TRUNC|O_CREAT|O_EXCL,0664);
12     if(fd<0)
13     {
14         if(errno==EEXIST)
15         {
16             printf("file open failed\n");
17         }
18     }
19
20     printf("open success..\n");
21     /*
22     while(1){
23         printf("I am runing...\n");
24         sleep(1);
25     }
26     */
27 }
```

如果打开的文件没有创建那么就用 open
自动创建，剩去了 creat 文件创建函数

当我文件创建之后，下次再执行该程序，如果同名的文件还在就 open 返回-1

```
root@ubuntu:/home/xiang/app/dbus# ./busd
open success..
root@ubuntu:/home/xiang/app/dbus# ls
busd  busd.c  dbus  dbus.c  texxxxt
root@ubuntu:/home/xiang/app/dbus# ./busd
file open failed
open success..
root@ubuntu:/home/xiang/app/dbus#
```

你看加了 O_EXCL 宏之后，因为已经第一次创建了该文件了，再次创建该同名的文件就会失败，返回-1；

如果我们取消 O_EXCL 会怎么样呢？

```
void main()
{
    int fd = -1;
    open(FILE,O_RDWR|O_TRUNC|O_CREAT,0664);
    if(fd<0)
    {
        if(errno==EXIST)
        {
            printf("file open failed\n");
        }
    }

    printf("open success..\n");
/*
while(1){
    printf("T am running...\n");
}
*/
root@ubuntu:/home/xiang/app/dbus# ls
busd  busd.c  dbus  dbus.c
root@ubuntu:/home/xiang/app/dbus# ./busd
open success..
root@ubuntu:/home/xiang/app/dbus# ls
busd  busd.c  dbus  dbus.c  texxxt
root@ubuntu:/home/xiang/app/dbus# ./busd
open success..
root@ubuntu:/home/xiang/app/dbus# 
```

你会发现其实文件已经建立了，但是不会进入 fd<0 里面的程序，那是因为我没有给 OPEN 函数加文件创建重名就报错的功能。这就是 O_EXCL 文件重名报错功能的好处

```

void delete_file();
int main()
{
    int fd = -1;
    open(FILE,O_RDWR|O_TRUNC|O_CREAT|O_EXCL,0664);
    if(fd<0)
    {
        if(errno==EEXIST)
        {
            printf("file open failed\n");
            return -1;
        }
    }

    printf("open success..\n");
    int i=0;
    for(i=0;i<10;i++)
    {
        printf("I am runing...\n");
        sleep(1);
    }
    return 0;
}

```

```

root@ubuntu:/home/xiang/app/dbus# ./busd
open success..
I am runing...
root@ubuntu:/home/xiang/app/dbus# ls
busd busd.c dbus dbus.c texxxt
root@ubuntu:/home/xiang/app/dbus# ./busd
file open failed
root@ubuntu:/home/xiang/app/dbus#

```

这种方式程序就只能运行一次，如果再运行就会报错，因为文件 texxxt 已经创建，但是没有删除。

为了解决上面这个问题，让程序在一个终端运行的时候，其他终端不能运行该程序，当该终端程序运行完成之后，其他终端可以执行该程序。所以我们要在程序退出时加删除文件函数。

```

void delete_file();
int main()
{
    int fd = -1;
    open(FILE,O_RDWR|O_TRUNC|O_CREAT|O_EXCL,0664);
    if(fd<0)
    {
        if(errno==EEXIST)
        {
            printf("file open failed\n");
            return -1;
        }
    }

    printf("open success..\n");
    atexit(delete_file); //第1节讲过的程序退出执行函数
    int i=0;
    for(i=0;i<10;i++)
    {
        printf("I am runing...\n");
        sleep(1);
    }
    return 0;
}

void delete_file()
{
    remove(FILE); //remove里面就带有rm命令，文件删除功能
}

```

程序可以反复执行了

程序在另外一个终端上无法马上执行

```
environ flock hello.c process process_create
root@ubuntu:/home/xiang/app# cd dbus/
root@ubuntu:/home/xiang/app/dbus# ls
busd busd.c dbus dbus.c
root@ubuntu:/home/xiang/app/dbus# ./busd
file open failed
root@ubuntu:/home/xiang/app/dbus#
```

```
root@ubuntu:/home/xiang/app/dbus# ./busd  
open success.  
I am running...  
root@ubuntu:/home/xiang/app/dbus# ls  
busd busd.c dbus dbus.c texxxt  
root@ubuntu:/home/xiang/app/dbus# ls  
busd busd.c dbus dbus.c texxxt  
root@ubuntu:/home/xiang/app/dbus# ls  
busd busd.c dbus dbus.c texxxt  
root@ubuntu:/home/xiang/app/dbus# ls
```

一个终端执行完另一个终端才可以再执行

```
app          Downloads    Pictures   vi
C-C++ Grammar_Improve examples.desktop Public    vi
Desktop      IMX        Templates  ye
root@ubuntu:/home/xiang$ cd app/
root@ubuntu:/home/xiang/app# ls
dbus         exec hello PID process_createfile sy
environ      flock hello.c process process_createfile2 sy
root@ubuntu:/home/xiang/app# cd dbus/
root@ubuntu:/home/xiang/app/dbus# ls
busd        busd.c dbus  dbus.c
root@ubuntu:/home/xiang/app/dbus# ./busd
file open failed
root@ubuntu:/home/xiang/app/dbus# ./busd
open success..
I am runung...
I am runung...
I am runung...
I am runung...
```

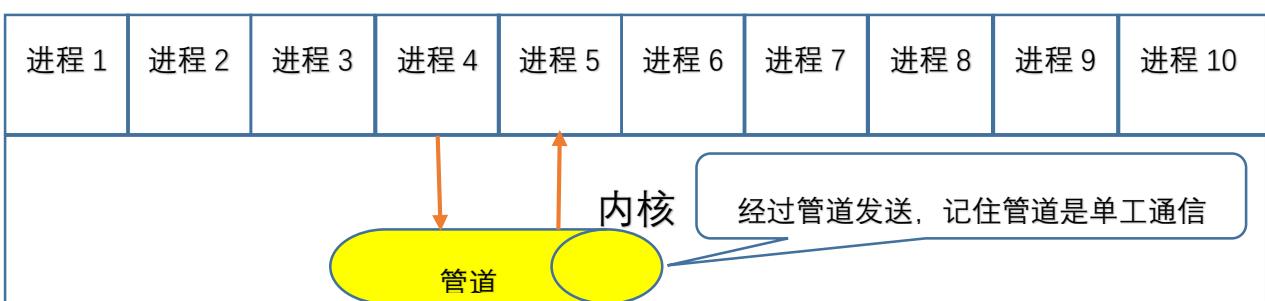
无名管道 两个进程间通信

管道是个什么东西？

进程 4 的数据发给进程 5，直接这样发送是不行的



进程 4 必须经过内核发送数据到进程五，所以要在内核里面建立一个管道。



`int pipe(int pipefd[2]);` 创建一个管道

pipefd[0] //是读端的 fd
 pipefd[1] //是写端的 fd
 read(pipefd[0], &buf, 1); 读管道，执行到这一句如果管道里面没有数据进程会阻塞，直到有数据读出来
 write(pipefd[1], argv[1], strlen(argv[1])); 写管道

```

1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6 #include <string.h>
7
8 Void main()
9 {
10     int pipefd[2]; //建立管道标识符
11     pid_t pid;
12     if(pipe(pipefd) == -1) { //打开管道，也就是管道已经在内核映射了
13         perror("pipe");
14         exit(EXIT_FAILURE);
15     }
16
17     pid=fork();
18
19     if(pid>0)
20     {
21         write(pipefd[1], "helloworld", 11); //向管道里面写数据pipefd[1]
22         wait(NULL); //你写了数据之后要等子进程来读这个数据，子进程没有来读这个数据，你就用wait来阻塞父进程。等到子进程读走数据为止。
23         //当然你不等到子进程来读这个数据，执意要继续write，那么前面的数据就会被覆盖，所以建议wait一下
24         close(pipefd[1]);
25     }
26     if(pid==0)
27     {
28         char buf[20];
29         read(pipefd[0], &buf, 11); //子进程读管道里面的数据
30         printf("%s\n",buf);
31     }
32     if(pid<0)
33     {
34         printf("fork failed\n");
35     }
36 }
37
38
  
```

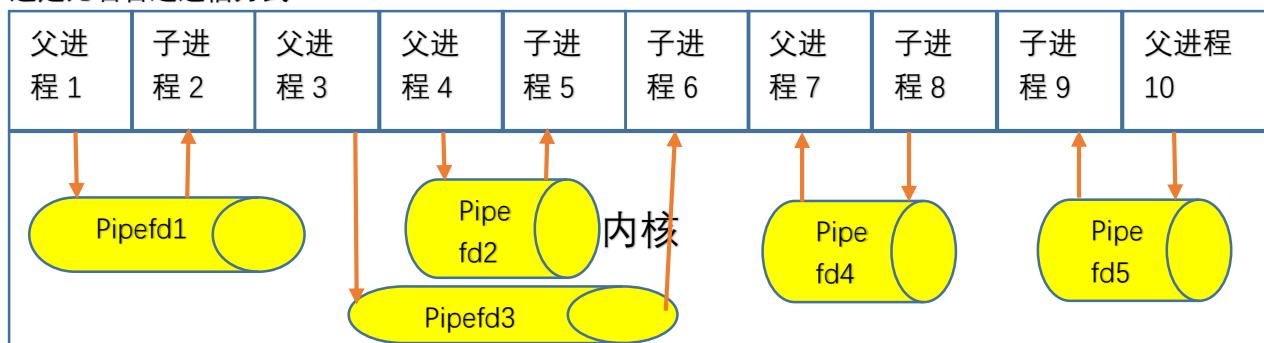
```

root@ubuntu:/home/xiang/app/pipe# ./pipe
helloworld
root@ubuntu:/home/xiang/app/pipe# ./pipe
helloworld
root@ubuntu:/home/xiang/app/pipe# ./pipe
helloworld
  
```

有名管道

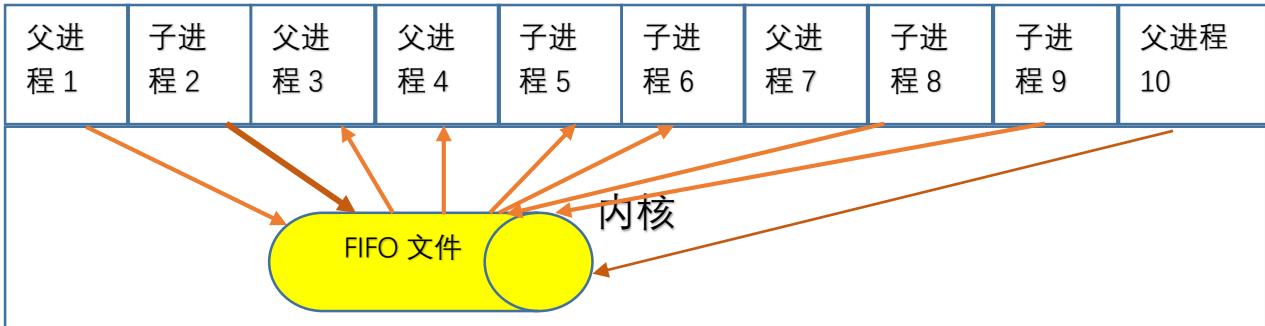
有名管道：在目录下创建 fifo 文件，两个进程通过 fifo 文件通信，不会存在无名管道那种无文件通信，所以无名管道只能父子进程通信。但是有名管道的 fifo 文件任何进程都可以使用。所以有名管道可以多进程通信。

这是无名管道通信方式



你看无名管道就是麻烦多个进程通信，就必须建立多个 pipe 变量，而且还不能乱伦通信，必须父子关系

我们看看有名管道



这就是有名管道可以多个进程通信，只要你创建了 FIFO 文件，任何进程都可以访问你这个 FIFO 管道。

```
int mkfifo(const char *pathname, mode_t mode); //fifo 文件创建函数
```

```
const char *pathname ://fifo 文件创建路径
```

```
mode_t mode ://fifo 文件权限
```

有名管道和无名管道不一样，可以多个文件进行通信。注意读文件的 open 函数必须是 O_RDONLY，写文件的 open 函数必须是 O_WRONLY，不能用 O_RDWR 这种属性。

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

int main()
{
    int pipefd[2]; //建立普通标识符
    pid_t pid;
    if(pipe(pipefd) == -1) { //打开管道，也就是管道已经在内存映射了
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    pid=fork();
    if(pid<0)
    {
        write(pipefd[1],"hello world",11); //向管道里写入数据pipefd[1]
        wait(NULL); //你写了数据之后要等子进程来读这个数据，子进程没有来读这个数据，你就用wait来阻塞父进程。等到子进程读走数据为止。
        //当然你再等子进程来读这个数据，执意要继续write，那么前面对的数据就会被覆盖，所以建议wait一下
        close(pipefd[1]);
    }
    if(pid==0)
    {
        char buf[20];
        read(pipefd[0],&buf,11); //子进程读管道里面的数据
        printf("%s\n",buf);
    }
    if(pid>0)
    {
        printf("fork failed\n");
    }
}
```

这是前面无名管道的代码，不管多少个进程都必须在同一个 C 文件里面编写，而且还要保持父子关系。搞得代码管理起来实在困难，无法模块化。有名管道就不一样，每个进程都可以是一个 C 文件，通过 fifo 通信。

有名管道创建程序和写程序

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

int main()
{
    int fd = -1;
    if(mkfifo("./fifo",0666)==-1)
    {
        perror("mkfifo error!");
        exit(1);
    }

    fd=open("./fifo",O_WRONLY);
    write(fd,"hellofifo",10);
    close(fd);
}
```

创建 fifo 文件，然后给文件赋权限，我喜欢在当前目录下创建 fifo 文件

打开 fifo 文件，这里是写数据，所以一定要是 O_WRONLY,不是 O_RDWR

像 fifo 文件写数据

创建读 fifo 的程序

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <sys/wait.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <unistd.h>
7 #include <string.h>
8 #include <fcntl.h>
9
10
11
12 int main()
13 {
14     int fd = -1;
15     char buf[20];
16     fd=open("./fifo",O_RDONLY);
17     read(fd,buf,11);
18     printf("%s\n",buf);
19     close(fd);
20
21 }
```

打开 fifo 文件，这里读数据，所以一定要是 O_RDONLY, 不是 O_RDWR

编译出两个文件

```
root@ubuntu:/home/xiang/app/fifo# ls
fifo  fifor.c  fifow  fifow.c
root@ubuntu:/home/xiang/app/fifo#
```

执行写程序，这时候创建 fifo 管道文件，然后将数据写入 fifo 文件

```
root@ubuntu:/home/xiang/app/fifo# ls
fifo  fifor.c  fifow  fifow.c
root@ubuntu:/home/xiang/app/fifo# ./fifow
```

```
root@ubuntu:/home/xiang/app/fifo# ls
for  fifor.c  fifow  fifow.c
ot@ubuntu:/home/xiang/app/fifo#
```

我们看到写程序阻塞到这里，这是因为要等待读程序来读。

```
[^Aroot@ubuntu:/home/xiang/app/fifo#
root@ubuntu:/home/xiang/app/fifo#
root@ubuntu:/home/xiang/app/fifo#
root@ubuntu:/home/xiang/app/fifo#
root@ubuntu:/home/xiang/app/fifo#
root@ubuntu:/home/xiang/app/fifo#
root@ubuntu:/home/xiang/app/fifo#
root@ubuntu:/home/xiang/app/fifo#
root@ubuntu:/home/xiang/app/fifo#
root@ubuntu:/home/xiang/app/fifo#
```

```
root@ubuntu:/home/xiang/app/fifo# ls
fifo  fifor.c  fifow  fifow.c
root@ubuntu:/home/xiang/app/fifo# ./fifor
hellofifo
root@ubuntu:/home/xiang/app/fifo#
```

读程序执行之后，写程序就通过阻塞结束运行。整个管道的单向数据发送完成了。

```
root@ubuntu:/home/xiang/app/fifo# ls
fifo  fifor  fifor.c  fifow  fifow.c
root@ubuntu:/home/xiang/app/fifo# ls -l
total 32
prw-r--r-- 1 root root    0 Jun 10 23:35 fifo
-rwxr-xr-x 1 root root 8816 Jun 10 22:17 fifor
```

这个就是创建的 fifo 文件。如果目录里面有上次创建的 fifo 文件了，我再次执行创建 fifo 程序会怎么样？

```
root@ubuntu:/home/xiang/app/fifo# ./fifow
mkfifo error!: File exists
root@ubuntu:/home/xiang/app/fifo#
```

再次执行 fifo 创建程序会报错

只有 rm 删除 fifo 文件，再执行才能成功执行。

如果你不想每次都删除 fifo 文件，你也可以这样。

```
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <unistd.h>
7 #include <string.h>
8 #include <fcntl.h>
9
10
11 int main()
12 {
13     int fd = -1;
14 /*      if(mkfifo("./fifo",0666)==-1)
15     {
16         perror("mkfifo error!");
17         exit(1);
18     }*/
19
20     fd=open("./fifo",O_WRONLY);
21     write(fd,"hellofifo",10);
22     close(fd);
23
24 }
```

因为 fifo 文件已经第一次创建了，然后我再执行该程序的时候屏蔽创建 fifo 直接使用 fifo 也是可以的，只是这样很麻烦

```
mkfifo error!: File exists
root@ubuntu:/home/xiang/app/fifo# ls
fifo  fifor.c  fifow  fifow.c
root@ubuntu:/home/xiang/app/fifo# gcc -o fifow fifow.c
root@ubuntu:/home/xiang/app/fifo# ./fifow
root@ubuntu:/home/xiang/app/fifo# ./fifow
root@ubuntu:/home/xiang/app/fifo# ./fifow
root@ubuntu:/home/xiang/app/fifo#
```

```
root@ubuntu:/home/xiang/app/fifo# cd fifo/
root@ubuntu:/home/xiang/app/fifo# ls
fifor  fifor.c  fifow  fifow.c
root@ubuntu:/home/xiang/app/fifo# ./fifor
hellofifo
root@ubuntu:/home/xiang/app/fifo# ./fifor
hellofifo
root@ubuntu:/home/xiang/app/fifo# ./fifor
hellofifo
root@ubuntu:/home/xiang/app/fifo# ./fifor
hellofifo
```

你看我执行多少次多进程读写都没有问题

我也可以用 unlink 命令来删除 fifo 文件

```
int main()
{
    int fd = -1;
    char buf[20];
    fd=open("./fifo",O_RDONLY);
    read(fd,buf,11);
    printf("%s\n",buf);
    close(fd);
    unlink("./fifo");
}
```

这样我的两个进程就可以多次运行读写功能了

```
root@ubuntu:/home/xiang/app/fifo# ./fifow
root@ubuntu:/home/xiang/app/fifo#
root@ubuntu:/home/xiang/app/fifo# ./fifow
root@ubuntu:/home/xiang/app/fifo# ./fifow
root@ubuntu:/home/xiang/app/fifo# ./fifow
root@ubuntu:/home/xiang/app/fifo#
```

```
root@ubuntu:/home/xiang/app/fifo# ls
fifor  fifor.c  fifow  fifow.c
root@ubuntu:/home/xiang/app/fifo# ./fifor
hellofifo
```

当然你也可以删除写程序里面的创建 fifo 函数，但是每次这样创建后去删除很麻烦。所以上面这种最好。

用管道传输数据实验

The screenshot shows a terminal window with two panes. The left pane contains the source code for two C programs: 'fifow.c' and 'fifor.c'. The right pane shows the terminal command line.

```
fifow.c
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <sys/wait.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <unistd.h>
7 #include <string.h>
8 #include <fcntl.h>
9
10
11 int main()
12 {
13     int fd = -1;
14     int w=10;
15     if(mkfifo("./fifo",0666)==-1)
16     {
17         perror("mkfifo error!");
18         exit(1);
19     }
20
21     fd=open("./fifo",O_WRONLY);
22     write(fd,&w,sizeof(w));
23     close(fd);
24
25 }
26

fifor.c
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <sys/wait.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <unistd.h>
7 #include <string.h>
8 #include <fcntl.h>
9
10
11 int main()
12 {
13     int fd = -1;
14     int r;
15     fd=open("./fifo",O_RDONLY);
16     read(fd,&r,sizeof(r));
17     printf("r = %d\n",r);
18     close(fd);
19     unlink("./fifo");
20
21 }
22 }
```

```
root@ubuntu:/home/xiang/app/fifo# ./fifow
root@ubuntu:/home/xiang/app/fifo# ./fifor
r = 10
```

看来管道接受数字是没有问题的，我们在试试管道多数据传输。

信号的使用

信号的意思就是比如我在终端执行 kill，然后指定一个进程，这个进程就结束了，kill 就是一个信号。

下面是几种常见的信号：

- **SIGHUP**: 从终端上发出的结束信号
- **SIGINT**: 来自键盘的中断信号（**Ctrl-C**）
- **SIGKILL**: 该信号结束接收信号的进程
- **SIGTERM**: **kill** 命令发出的信号
- **SIGCHLD**: 标识子进程停止或结束的信号
- **SIGSTOP**: 来自键盘（**Ctrl-Z**）或调试程序的停止执行信号

```
1 SIGQUIT      输入 Quit Key 的时候 (CTRL+\) 发送给所有 Foreground Group 的进程
2 #include <signal.h> //signal调用该头文件
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h> //pause调用该头文件
6
7 void my_int(int sign)
8 {
9     printf("input SIGINT\n");
10 }
11
12 void my_quit(int sign)
13 {
14     printf("input SIGQUIT\n");
15 }
16 void main()
17 {
18
19     printf("enter singal main...\n");
20     signal(SIGINT,my_int);
21     signal(SIGQUIT,my_quit);
22
23     pause();    接收键盘敲击过来的信号
24     exit(0);
25 }
```

这个 int 形参是用来识别
键盘传入的是上面信号

接收到键盘的 SIGINT 信
号就执行 my_int 中断服
务函数，如果接收到键
盘 SIGQUIT 信号就执行
my_quit 中断服务函数

Pause 函数使让该进程挂起，直到有一个信号传入进程。然后执行信号处理函数，进程挂起结束。

看看执行结果是上面样子的

```

root@ubuntu:/home/xiang/app/singl# ./signal
enter singal main...
^Cinput SIGINT
root@ubuntu:/home/xiang/app/singl# ./signal
enter singal main...
^\\input SIGQUIT
root@ubuntu:/home/xiang/app/singl#

```

键盘 Ctrl+C

键盘 Ctrl+\

还有一种做法，其结果和上面一样。

```

void my_func(int sign)
{
    if(sign==SIGINT)
    {
        printf("input SIGINT\\n");
    }
    if(sign==SIGQUIT)
    {
        printf("input SIGQUIT\\n");
    }
}

void main()
{
    printf("enter singal main...\\n");
    signal(SIGINT,my_func);
    signal(SIGQUIT,my_func);

    pause();
    exit(0);
}

```

用一个中断函数
接收两个信号，
用形参来判断哪
一路信号

```

root@ubuntu:/home/xiang/app/singl# ./signal
enter singal main...
^Cinput SIGINT
root@ubuntu:/home/xiang/app/singl# ./signal
enter singal main...
^\\input SIGQUIT

```

键盘 Ctrl+C

键盘 Ctrl+\

```

void my_func(int sign)
{
    if(sign==SIGINT)
    {
        printf("input SIGINT\\n");
    }
    if(sign==SIGSTOP)
    {
        printf("input SIGSTOP\\n");
    }
}

void main()
{
    printf("enter singal main...\\n");
    signal(SIGINT,my_func);
    signal(SIGSTOP,my_func);

    pause();
    exit(0);
}

```

我们发现给信号函数传入 SIGSTOP 信
号，在中断函数里面是没有执行的
至于这些信号为什么不行我也不知道

```

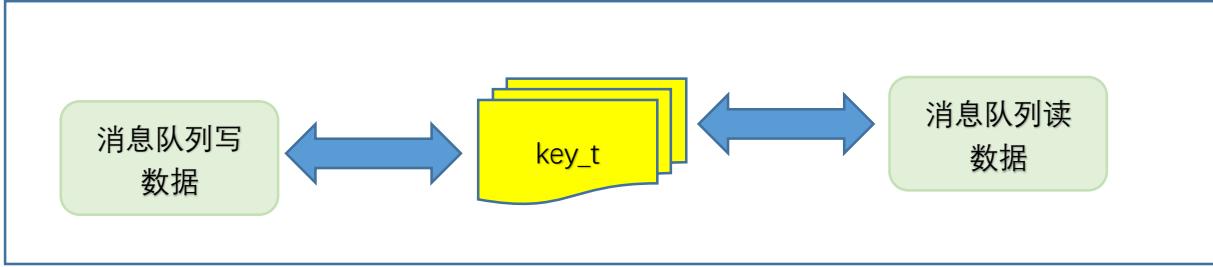
root@ubuntu:/home/xiang/app/singl# ./signal
enter singal main...
^Z
[3]+  Stopped                  ./signal
root@ubuntu:/home/xiang/app/singl#

```

消息队列

消息队列是可以传输有格式的一帧数据，而管道传输的是无格式的数据、

key_t 类型是消息队列的 ID 编号，也就是读程序和写程序都是通过 key_t 这个中转站来读写数据的。



```
1 #include<stdio.h>
2 #include<sys/types.h>
3 #include<stdlib.h>
4 #include<sys/ipc.h>
5
6
7 int main()
8 {
9     key_t key1, key2, key3;
10    if((key1=ftok("/etc/profile",0))==-1)
11    {
12
13        perror("ftok");
14        exit(1);
15    }
16    if((key2=ftok("/etc/profile",1))==-1)
17    {
18
19        perror("ftok");
20        exit(1);
21    }
22    if((key3=ftok("/etc/profile",0))==-1)
23    {
24
25        perror("ftok");
26        exit(1);
27    }
28
29    printf("key1 = %d, key2 = %d ket3 = %d\n",key1,key2,key3);
30    return 0;
31 }
```

我们随便给 ftok 载入个文件，让 ftok 根据文件创建个 key_t 的键值，也就是 ID 值

```
root@ubuntu:/home/xiang/app/queue# ./queue
key1 = 65736, key2 = 16842952 ket3 = 65736
```

这里的 key1 编号和 key3 编号是一样的，key2 编号不一样这是为什么呢？

key_t ftok(char * fname, int id)

fname 就是你指定的文件名（已经存在的文件名），一般使用当前目录，如：

key_t key;

key = ftok(".", 1); 这样就是将 fname 设为当前目录。

id 是子序号。

在一般的 UNIX 实现中，是将文件的索引节点号取出，前面加上子序号得到 key_t 的返回值。

我们来正式创建一个消息队列

```

4 #include<sys/ipc.h>
5 #include<sys/msg.h>
6 #include<errno.h>
7 #include<string.h>
8 typedef struct message
9 {
10     long int type;
11     char buffer[512];
12 }MSG;
13
14 int main()
15 {
16
17     key_t key;
18     int mesgid;
19     MSG msg;
20 /*    if((key=ftok("./que",0))==-1)
21     {
22         perror("ftok");
23         exit(1);
24     }*/
25     if((mesgid=msgget((key_t)123,IPC_CREAT|IPC_EXCL|0666))==-1)
26     {
27         if(errno != EEXIST)
28         {
29             perror("msgget");
30             exit(1);
31         }
32     }
33     if((mesgid=msgget((key_t)123,0))==-1)
34     {
35         perror("mesgid");
36         exit(1);
37     }
38
39     msg.type = 2;
40     memcpy(msg.buffer,"123456",6);
41     if(msgsnd(mesgid, (void*)&msg, strlen(msg.buffer), IPC_NOWAIT) == -1)
42     {
43         perror("msgsnd");
44     }
45
46     return 0;
47
48
49
50

```

我们没有用 ftok 来自动获取键值 ID 值

1. 我们自己设定了键值也就是 ID 值

3.IPC_EXCL 如果 key 存在, 返回失败

2.IPC_CREAT 表示当 key 所命名的消息队列不存在时创建一个消息队列, 如果 key 所命名的消息队列存在时, IPC_CREAT 标志会被忽略, 而只返回一个标识符

4.如果 msgget 失败的原因不是因为消息队列存在而失败, 而是其他原因, 那么这里会报错

5.如果是因为消息队列 key 存在的原因而失败, 那么我就再创建一次消息队列, 但是不用再去判断, key 值, 所以填 0

消息发送部分

我们在前面结构体创建的消息队列标准数据帧

```

msg.type = 2;
memcpy(msg.buffer,"123456",6);
if(msgsnd(mesgid, (void*)&msg, strlen(msg.buffer), IPC_NOWAIT) == -1)
{
    perror("msgsnd");
}

```

将要发送的数据写入 buffer

消息队列文件 fd

要发送的结构体

结构体里面 buffer 长度

非阻塞发送

接收端的 type 必须和发送端一样。都为 2, 如果发送端设置为 3, 那么接收端也必须为 3。

```

root@ubuntu:/home/xiang/app/queue# ./sendmsg
root@ubuntu:/home/xiang/app/queue# ipcs -q

----- Message Queues -----
key        msqid      owner      perms      used-bytes      messages
0x0000007b 163840      root      666          6              1
这是 key 值 123  发送程序文件 fd
key 值权限
发送多少个字节
Msgsnd 执行了几次
也就是发送了几次

```

消息接收部分

```
8 typedef struct messge
9 {
10     long int type;
11     char buffer[512];
12 }MSG;
13
14 int main()
15 {
16
17     key_t key;
18     int mesgid;
19     MSG msg;
20 /*    if((key=ftok("./que",0))==-1)
21     {
22         perror("ftok");
23         exit(1);
24    }*/
25    if((mesgid=msgget((key_t)123,IPC_CREAT|IPC_EXCL|0666))==-1)
26    {
27        if(errno != EEXIST)
28        {
29            perror("msgget");
30            exit(1);
31        }
32
33        if((mesgid=msgget((key_t)123,0))==-1)
34        {
35            perror("mesgid");
36            exit(1);
37        }
38    }
39
40    msg.type = 2;
41
42    if(msgrcv(mesgid, (void*)&msg, sizeof(msg.buffer),2, 0) == -1)
43    {
44        perror("msgrecv");
45    }
46    printf("%s\n",msg.buffer);
47
48
49    return 0;
50
51
52
53 }
54 }
```

我们没有用 ftok 来自动获取键值 ID 值

接收消息创建和发送消息程序是一样

接收消息部分

计算 buffer 长度

这里要和发送的 type 一样

```
root@ubuntu:/home/xiang/app/queue# ./recvmsg
123456
root@ubuntu:/home/xiang/app/queue# █
```

把数据接收过来了

```

root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid   owner    perms  used-bytes messages
0x00000007b 196608    root    666        6       1
root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid   owner    perms  used-bytes messages
0x00000007b 196608    root    666        0       0
root@ubuntu:/home/xiang/app/queue#

```

数据接收之后消息

就被清空了

以上方法在 ARM 开发板上面到没有什么，因为 ARM 开发板上面就我一个用户。但是在主机或者服务器开发时一台电脑可能会有好几个人用。难免会有人在做消息队列系统的时候把 key_t 值和你写重复，造成重大错误。所以我们要用自动分配 key 的方法。

先创建一个没有后缀的 key 文件

```

root@ubuntu:/home/xiang/app/queue# ls
creatqueue creatqueue.c que queue c
root@ubuntu:/home/xiang/app/queue#

```

我们创建了一个 que

```

#include<sys/msg.h>
#include<errno.h>
#include<string.h>
typedef struct msgge
{
    long int type;
    char buffer[512];
}MSG;
int main()
{
    key_t key;
    int mesgid;
    MSG msg;
    if((key=ftok("./que",0))==-1)
    {
        perror("ftok");
        exit(1);
    }
    if((mesgid=msgget(key,IPC_CREAT|IPC_EXCL|0666))==-1)
    {
        if(errno != EEXIST)
        {
            perror("msgget");
            exit(1);
        }
        if((mesgid=msgget(key,0))==-1)
        {
            perror("mesgid");
            exit(1);
        }
    }
    msg.type = 2;
    memcpy(msg.buffer,"123456",6);
    if(msgsnd(mesgid, (void*)&msg, strlen(msg.buffer), IPC_NOWAIT) == -1)
    {
        perror("msgsnd");
    }
    printf("%d\n",mesgid);
    return 0;
}

```

发送函数我们用 ftok 自动分配 key

```

root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes     messages
root@ubuntu:/home/xiang/app/queue# ./sendmsg
229376
root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes     messages
0x0001a989 229376      root      666          6             1
root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes     messages
0x0001a989 229376      root      666          0             0
root@ubuntu:/home/xiang/app/queue# 

```

```

3 #include<stdlib.h>
4 #include<sys/IPC.h>
5 #include<sys/msg.h>
6 #include<errno.h>
7 #include<string.h>
8 typedef struct messge
9 {
10     long int type;
11     char buffer[512];
12 }MSG;
13
14 int main()
15 {
16
17     key_t key;
18     int mesgid;
19     MSG msg;
20     if((key=ftok("./que",0))==-1)
21     {
22         perror("ftok");
23         exit(1);
24     }
25     if((mesgid=msgget(key,IPC_CREAT|IPC_EXCL|0666))==-1)
26     {
27         if(errno != EEXIST)
28         {
29             perror("msgget");
30             exit(1);
31         }
32         if((mesgid=msgget(key,0))==-1)
33         {
34             perror("mesgid");
35             exit(1);
36         }
37     }
38
39     msg.type = 2;
40     if(msgrcv(mesgid, (void*)&msg, sizeof(msg.buffer), 2, 0) == -1)
41     {
42         perror("msgrcv");
43     }
44     printf("%s\n",msg.buffer);
45     return 0;
46 }

```

```

root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes     messages
root@ubuntu:/home/xiang/app/queue# ./sendmsg
229376
root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes     messages
0x0001a989 229376      root      666          6             1
root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes     messages
0x0001a989 229376      root      666          0             0
root@ubuntu:/home/xiang/app/queue# 

```

被接收程序读走了

我们再来试试消息队列多数据传输

```

queue
typedef struct msgge
{
    long int type;
    char buffer[512];
}MSG;

int main()
{
    key_t key;
    int mesgid;
    MSG msg;
    if((key=ftok("./que",0))==-1)
    {
        perror("ftok");
        exit(1);
    }
    if((mesgid=msgget(key,IPC_CREAT|IPC_EXCL|0666))==-1)
    {
        if(errno != EEXIST)
        {
            perror("msgget");
            exit(1);
        }
        if((mesgid=msgget(key,0))==-1)
        {
            perror("mesgid");
            exit(1);
        }
    }
    for(int i=0;i<100;i++)
    {
        msg.buffer[i]=i;
    }
    msg.type = 2;
    if(msgsnd(mesgid, (void*)&msg, sizeof(msg.buffer), IPC_NOWAIT) == -1)
    {
        perror("msgsnd");
    }
    printf("%d\n",mesgid);
    return 0;
}

```

将字符串长度改成
算 buffer 地址长度

这是发送

100 个数据程序

```

g/app/queue
 7 #include<string.h>
 8 typedef struct messge
 9 {
10     long int type;
11     char buffer[512];
12 }MSG;
13
14 int main()
15 {
16
17     key_t key;
18     int mesgid;
19     MSG msg;
20     if((key=fork("./que",0))==-1)
21     {
22         perror("fork");
23         exit(1);
24     }
25     if((mesgid=msgget(key,IPC_CREAT|IPC_EXCL|0666))==-1)
26     {
27         if(errno != EEXIST)
28         {
29             perror("msgget");
30             exit(1);
31         }
32
33         if((mesgid=msgget(key,0))==-1)
34         {
35             perror("mesgid");
36             exit(1);
37         }
38     }
39
40     msg.type = 2;
41     if(msgrcv(mesgid, (void*)&msg, sizeof(msg.buffer), 2, 0) == -1)
42     {
43         perror("msgrcv");
44     }
45     for(int i=0;i<100;i++){
46         printf("%d",msg.buffer[i]);
47         printf(" ");
48     }
49
50     return 0;
51

```

这是接收 100 个数据程序

```

root@ubuntu:/home/xiang/app/queue# ./recvmsg
root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes      messages
root@ubuntu:/home/xiang/app/queue# ./sendmsg
32768
root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes      messages
0x0001a989  32768      root      666          512           1

```

下面这个是接收的数据

```

root@ubuntu:/home/xiang/app/queue# ./recvmsg
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 root@ubuntu:/home/xiang/app/queue#

```

```

root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
这个 key 就是根据我那个 que 文件分配的，不是我指定的
key      msqid      owner      perms      used-bytes      messages
0x0001a989  32768      root      666          0           0
root@ubuntu:/home/xiang/app/queue# 

```

其实消息队列发送程序可以一直发送消息的，就算接收程序没来得及接收，发送也可以一直发

```

root@ubuntu:/home/xiang/app/queue# ./sendmsg
32768
root@ubuntu:/home/xiang/app/queue# ./sendmsg
32768
root@ubuntu:/home/xiang/app/queue# ./sendmsg
32768
root@ubuntu:/home/xiang/app/queue# ./sendmsg
32768
root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes      messages
0x0001a989 32768      root      666          2048                  4

root@ubuntu:/home/xiang/app/queue#

```

执行了 4 次发送程序，
就发送了 4 次消息

```

root@ubuntu:/home/xiang/app/queue# ./recvmsg
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 5
9 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 root@ubuntu:/home/xiang/app/queue#
root@ubuntu:/home/xiang/app/queue# ./recvmsg
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 5
9 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 root@ubuntu:/home/xiang/app/queue#
root@ubuntu:/home/xiang/app/queue# ./recvmsg
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 5
9 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 root@ubuntu:/home/xiang/app/queue#
root@ubuntu:/home/xiang/app/queue# ./recvmsg
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 5
9 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 root@ubuntu:/home/xiang/app/queue#
root@ubuntu:/home/xiang/app/queue# ./recvmsg
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 5
9 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 root@ubuntu:/home/xiang/app/queue#

```

这里要接收 4 次才能把队列的数据接收完

```

root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes      messages
0x0001a989 32768      root      666          0                  0

root@ubuntu:/home/xiang/app/queue# ipcrm -q 32768
root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes      messages
0x0001a989 32768      root      666          0                  0

root@ubuntu:/home/xiang/app/queue#

```

如果你想取消创建的
消息队列，就执行
ipcrm

当然这样用命令来取消消息队列，在代码运行中不太科学，所以我们就要用下面的消息队列控制函数。

一般我们是接收函数接收完数据之后再去取消消息队列。

```

int main()
{
    key_t key;
    int mesgid;
    MSG msg;
    if((key=ftok("./que",0))==-1)
    {
        perror("ftok");
        exit(1);
    }
    if((mesgid=msgget(key,IPC_CREAT|IPC_EXCL|0666))==-1)
    {
        if(errno != EEXIST)
        {
            perror("msgget");
            exit(1);
        }

        if((mesgid=msgget(key,0))==-1)
        {
            perror("mesgid");
            exit(1);
        }
    }

    msg.type = 2;
    if(msgrcv(mesgid, (void*)&msg, sizeof(msg.buffer), 2, 0) == -1)
    {
        perror("msgrecv");
    }
    for(int i=0;i<100;i++){
        printf("%d",msg.buffer[i]);
        printf(" ");
    }

    if(msgctl(mesgid,IPC_RMID,NULL)<0) // 消息队列取消函数
    {
        printf("IPC_RMID error\n");
    }
    return 0;
}

```

```

root@ubuntu:/home/xiang/app/queue# ./sendmsg
229376
root@ubuntu:/home/xiang/app/queue# ipcs -q // 发送消息队列
----- Message Queues -----
key        msqid      owner      perms      used-bytes     messages
0x0001a989 229376      root       666          512           1

```

```

root@ubuntu:/home/xiang/app/queue# ./recvmsg
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
root@ubuntu:/home/xiang/app/queue#

```

这是接收数据

```
root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes     messages
root@ubuntu:/home/xiang/app/queue#
```

这是接收之后消息队列就取消了，不像最先那样消息队列创建后还一直存在，必须用 ipcrm 来取消。
如果我是发送端取消消息队列呢？

```
int main()
{
    key_t key;
    int mesgid;
    MSG msg;
    if((key=ftok("./que",0))==-1)
    {
        perror("ftok");
        exit(1);
    }
    if((mesgid=msgget(key,IPC_CREAT|IPC_EXCL|0666))==-1)
    {
        if(errno != EEXIST)
        {
            perror("msgget");
            exit(1);
        }
        if((mesgid=msgget(key,0))==-1)
        {
            perror("mesgid");
            exit(1);
        }
    }
    for(unsigned char i=0;i<100;i++)
    {
        msg.buffer[i]=i;
    }

    msg.type = 2;
    if(msgsnd(mesgid, (void*)&msg, sizeof(msg.buffer), IPC_NOWAIT) == -1)
    {
        perror("msgsnd");
    }
    printf("%d\n",mesgid);
```

消息队列取消函数

```
if(msgctl(mesgid,IPC_RMID,NULL)<0)
{
    printf("IPC_RMID error\n");
}
return 0;
```

```
root@ubuntu:/home/xiang/app/queue# ./sendmsg
262144
root@ubuntu:/home/xiang/app/queue# ipcs -q
----- Message Queues -----
key      msqid      owner      perms      used-bytes     messages
root@ubuntu:/home/xiang/app/queue#
```

```
root@ubuntu:/home/xiang/app/queue# ./recvmsg
```

你看接收端阻塞了，因为发送端像消息队列写数据后有删除了消息队列。接收端读不到消息，因为被发送端删除了。

Linux 进程信号量

进程信号量不是用来给两个进程传输数据用的，而是用来控制两个进程的互斥或同步。

进程信号量和线程信号量代码编写方式是不一样的，但是功能差不多

创建信号量

```

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

int main()
{
    int fd;
    key_t x;
    if((x=ftok("./semmsg",0))==-1)
    {
        printf("ftok creat failed\n");
    }
    Key 值
    fd=semget(x,1,IPC_CREAT|0666);
    if(fd<0)
    {
        printf("semget error\n");
    }
    printf("semget creat success\n");
    return 0;
}

```

信号量头文件

我们 key 的 ID 值还是和
上面一样自动创建

需要创建的信号量数目

信号量权限

```

root@ubuntu:/home/xiang/app/sem# ipcs -s
----- Semaphore Arrays -----
key      semid      owner      perms nsems
root@ubuntu:/home/xiang/app/sem# ./sem
semget creat success
root@ubuntu:/home/xiang/app/sem# ipcs -s
----- Semaphore Arrays -----
key      semid      owner      perms nsems
0x00016d38 32768      root      666      1
root@ubuntu:/home/xiang/app/sem#

```

信号量没创建

信号量已创建

只有 1 个信号量

我们来做一个信号量封装程序

```
union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

int init_sem(int semnus,int value)
{
    int semid=semget(IPC_PRIVATE,semnus,IPC_CREAT|0666);
    if(semid<0)
    {
        return -1;
    }
    union semun um;
    unsigned short *array=(unsigned short*)calloc(semnus,sizeof(unsigned short));
    int i;
    for(i=0;i<semnus;i++)
    {
        array[i]=value;
    }
    um.array=array;
    if(semctl(semid,0,SETALL,um)<0)
    {
        printf("semctl error\n");
    }
    free(array);
    return semid;
}
```

IPC_PRIVATE 和 ftok 的区别在于。使用 IPC_PRIVATE 创建的 IPC 对象, key 值属性为 0 , 这样毫无关系的进程就不能通过 key 值来得到 IPC 对象的编号 , 和无名管道类似 , 只能在一 个 C 文件里面创建多进程进行信号量操作
Ftok 可以用两个 C 文件生成不同的进程来进行信号量操作

信号量个数

初始化信号量的值，其他操作函数会在这个值得基础上+或者-

Semget 创建了信号量 ID

0 表示操作所有信号量，1 表示操作信号量 1，以此类推

将 semun 里面的参数写入该 ID 的信号量

将创建的信号量 ID 返回出去给其他操作这个信号量的函数用

● IPC_STAT	获取信号量集的属性	→ buf
● IPC_SET	设置信号量集的属性	→ buf
● IPC_RMID	删除信号量集	→ buf
● GETVAL	返回信号量的值	→ val
● SETVAL	设置semnum信号量的值	→ val
● GETALL	获取所有信号量的值	→ array
● SETALL	设置所有信号量的初始值	→ array

```

void P(int semid,int semnum,int value)
{
    struct sembuf ops[]=
    {
        {semnum,-value,SEM_UNDO}
    };
    assert(value>=0);
    if(semop(semid,ops,sizeof(ops)/sizeof(struct sembuf))<0)
    {
        printf("semop error \n");
    }
}

```

上面 init 创建的信号量 ID

操作这个 ID 里面的第几个信号量

这个信号量的值是减操作

信号量多少个计算

这个信号量 ID 就是给其他进程操作
这个信号量的编号，没有这个编号
其他进程不知道操作的哪一个信号量

信号量加操作，和上面一样，只是把减好去掉

```

71 void V(int semid,int semnum,int value)
72 {
73
74     struct sembuf ops[]=
75     {
76         {semnum,value,SEM_UNDO}
77     };
78     assert(value>0);
79     if(semop(semid,ops,sizeof(ops)/sizeof(struct sembuf))<0)
80     {
81         printf("semop error \n");
82     }
83 }

```

信号量的值加操作

OPS 里面我只放了一个结构体所
以只有一个信号量，如果多放几
个结构体就是多个信号量

删除信号量

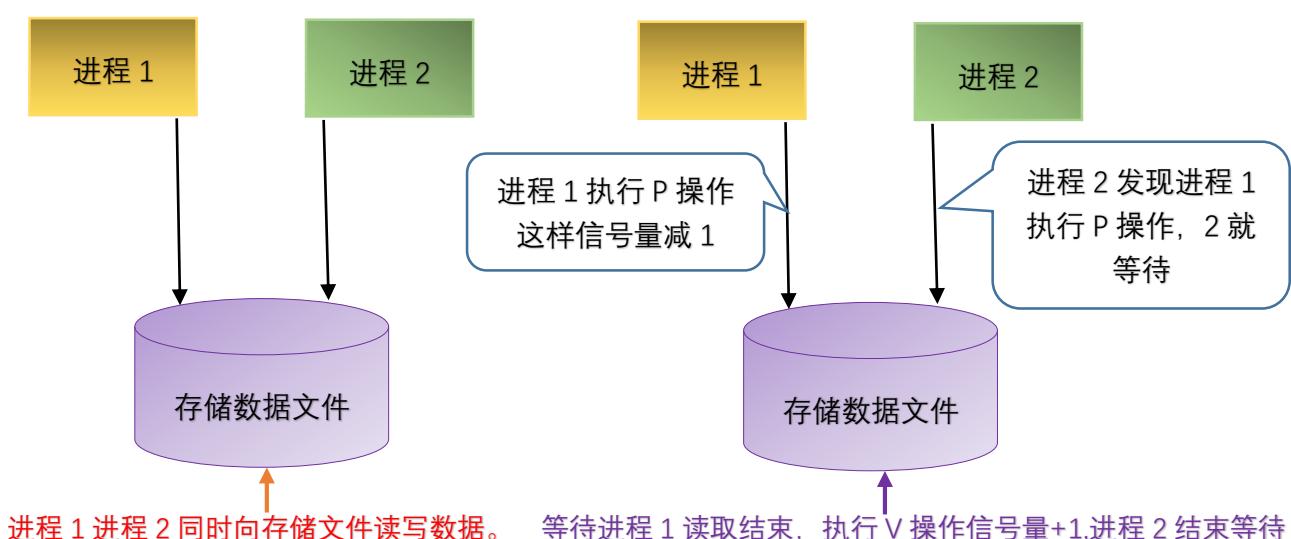
```

void delete_sem(int semid)
{
    if(semctl(semid,0,IPC_RMID,NULL)<0)
    {
        printf("undo sem error\n");
    }
}

```

上面这些信号量函数我们一般都要单独封装到 C 文件里面，免得每个文件都来写一遍麻烦

下面我们来用信号量来做 **进程互斥**



```
void deposit(int fd)
{
    P(fd, 0, 1);
    printf("deposit\n");
    V(fd, 0, 1);
}

void withdraw(int fd)
{
    P(fd, 0, 1);
    printf("withdraw\n");
    V(fd, 0, 1);
}

int main()
{
    pid_t pid;

    int fd = init_sem(1, 1);
    if(fd < 0)
    {
        printf("init_sem error\n");
    }

    pid = fork();

    if(pid == 0)
    {
        deposit(fd);
    }
    if(pid > 0)
    {
        withdraw(fd);
    }
    if(pid < 0)
    {
    }
}
```

选择信号量 ID, 第 0 的一组信号量结构, P 操作减 1

struct sembuf ops[] =
{
 {semnum, -value, SEM_UNDO}
};
...
第 0 组

选择信号量 ID, 第 0 的一组信号量结构, P 操作加 1

当子进程先执行 deposit 函数的时候, 父进程等待,
当父进程先执行 withdraw 的时候, 子进程等待。
不管哪个进程等待都要等信号量被进程加回 0 才执行

共享内存

共享内存是两个进程传输数据的方式，特点是可以传输的数据量大如视频数据传输效率比上面的队列高

共享内存操作流程

1、Int shmget(key_t key,size_t size ,IPC_CREAT|IPC_EXCL|0777)；创建共享内存

Key：共享内存键值(ID)，和队列一样，多个进程可以通过这个 key 访问同一个共享内存

Size：要分配的物理内存大小，根据自己需要来分配

Int 如果共享内存创建成功返回内核共享内存 ID

2、shmctl(int shmid,IPC_RMID,NULL);//删除共享的物理内存

3、void* shmat(shmid,char* shmaddr,0);共享内存映射

Void* 虚拟内存映射共享的物理内存成功返回这个虚拟内存的地址，操作这个地址就是操作这个物理内存

Shmid：共享内存 ID

Shmaddr：物理地址映射到进程的虚拟地址然后返回到 void，我们都是填 0，让操作系统自己去分配地址

第 3 个参数我们也设置为 0

下面我们用无名管道来操作共享内存

```
#include <unistd.h>
#include <string.h>
#include <sys/shm.h> //共享内存头文件
void main()
{
    int shmid; //创建共享内存 ID
    int pipefd[2]; //建立管道标识符
    pid_t pid;

    if((shmid=shmget(IPC_PRIVATE,1024,IPC_CREAT|IPC_EXCL|0777))<0)//创建共享内存
    {
        perror("shmid\n");
        exit(1);
    }

    if(pipe(pipefd) == -1) { //打开管道，也就是管道已经在内核映射了
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    pid=fork();

    if(pid>0) //父进程
    {

        int *pi=(int *)shmat(shmid,0,0);
        if(pi==(<int*>-1))
        {
            perror("shmat\n");
            exit(1);
        }

        *pi=100;
        *(pi+1)=200;
        shmdt(pi); //解除共享内存映射
        write(pipefd[1], "w", 1); //向管道里面写数据pipefd[1] 证明进程向共享内存写数据完成
        wait(NULL); //等到子进程读走数据。用管道 write 函数使为了实现进程
        close(pipefd[1]); //之间的同步
    }
}
```

IPC_PRIVATE 这个参数可以用在无名管道上做 Key 值，前面章节已经描述了，有名管道就必须用 ftok

分配物理内存大小，我们这里分配 1024 字节

将物理内存映射给虚拟内存，然后 pi 就获得了虚拟内存的地址，我们操作 pi 就是去写数据到虚拟内存，虚拟内存再自动传到物理内存

进程 1 写 //进程 1 向管道写入数据，同时向共享内存写入数据

进程 2 等待进程 1 写完 //进程 2 从管道读取数据，同时从共享内存读取数据

进程 1 写

进程 2 等待进程 1 写完

共享内存

这就是管道用来防止两个进程同时读写共享内存造成数据紊乱，所以要设置先来后到。也可以用信号量

```
if(pid==0) //子进程
{
    char buf[1];
    read(pipefd[0], &buf, 1); //用子进程读管道数据的阻塞方法等待父进程向共享内存写入数据
    int *pi=(int *)shmat(shmid,0,0);

    if(pi==(int*)-1)
    {
        perror("shmat\n");
        exit(1);
    }
    printf("%s ,pi = %d,pi+1 = %d\n",buf,*pi,*(pi+1));

    shmdt(pi); //解除共享内存映射
    shmctl(shmid,IPC_RMID,NULL);
    close(pipefd[0]);
}
if(pid<0)
{
    printf("fork failed\n");
}
```

用 read 来阻塞等待进程 1 写完共享内存

操作步骤和共享内存写是基本一样的

进程 1 写完

进程 2 读取共享内存

共享内存

```
root@ubuntu:/home/xiang/app/mem# ./memw
W ,pi = 100,pi+1 = 200
```

这是执行结果，读到的内存数据

```
if(pid==0) //子进程
{
    char buf[1];
    read(pipefd[0], &buf, 1); //用子进程读管道数据的阻塞方法等待父进程向共享内存写入数据
    int *pi=(int *)shmat(shmid,0,0);

    if(pi==(int*)-1)
    {
        perror("shmat\n");
        exit(1);
    }
    printf("%s ,pi = %d,pi+1 = %d\n",buf,*pi,*(pi+1));

    shmdt(pi); //解除共享内存映射
    shmctl(shmid,IPC_RMID,NULL);
    close(pipefd[0]);
}
if(pid<0)
{
    printf("fork failed\n");
}
```

在读取共享内存数据后记得消除共享内存

```

root@ubuntu:/home/xiang/app/mem# ./memw
w ,pi = 100,pi+1 = 200
root@ubuntu:/home/xiang/app/mem# ipcs -m
----- Shared Memory Segments -----
key      shmid   owner    perms      bytes  nattch   status
0x00000000 294912  xiang    600      524288     2        dest
0x00000000 655361  xiang    600     16777216     2        dest
0x00000000 425986  xiang    600      524288     2        dest
0x00000000 1441795 xiang    600      524288     2        dest
0x00000000 753668  xiang    600      524288     2        dest
0x00000000 1081349 xiang    600      524288     2        dest
0x00000000 1343494 xiang    600      524288     2        dest
0x00000000 1212423 xiang    600      524288     2        dest
0x00000000 1245192 xiang    600     67108864     2        dest
0x00000000 1540105 xiang    600      524288     2        dest
0x00000000 1802250 xiang    600      524288     2        dest

```

这样在你共享内存
使用结束后，不会
占用内存空间

如果程序运行结束后不消除共享内存

```

if(pid==0) //子进程
{
    char buf[1];
    read(pipefd[0], &buf, 1); //用子进程读管道数据的阻塞方法等待父进程向共享内存写入数据
    int *pi=(int *)shmat(shmid,0,0);

    if(pi==(int*)-1)
    {
        perror("shmat\n");
        exit(1);
    }
    printf("%s ,pi = %d,pi+1 = %d\n",buf,*pi,(pi+1));

    shmdt(pi); //解除共享内存映射
    shmctl(shmid,IPC_RMID,NULL);
    close(pipefd[0]);
}
if(pid<0)
{
    printf("fork failed\n");
}

```

```

root@ubuntu:/home/xiang/app/mem# ./memw
w ,pi = 100,pi+1 = 200
root@ubuntu:/home/xiang/app/mem# ipcs -m
----- Shared Memory Segments -----
key        shmid    owner      perms      bytes  nattch   status
0x00000000 294912  xiang      600          524288    2        dest
0x00000000 655361  xiang      600          16777216   2        dest
0x00000000 425986  xiang      600          524288    2        dest
0x00000000 1441795 xiang      600          524288    2        dest
0x00000000 753668  xiang      600          524288    2        dest
0x00000000 1081349 xiang      600          524288    2        dest
0x00000000 1343494 xiang      600          524288    2        dest
0x00000000 1212423 xiang      600          524288    2        dest
0x00000000 1245192 xiang      600          67108864   2        dest
0x00000000 1540105 xiang      600          524288    2        dest
0x00000000 1802250 xiang      600          524288    2        dest
0x00000000 3244043 root       777          1024     0        dest
root@ubuntu:/home/xiang/app/mem# ./memw
w ,pi = 100,pi+1 = 200
root@ubuntu:/home/xiang/app/mem# ipcs -m
----- Shared Memory Segments -----
key        shmid    owner      perms      bytes  nattch   status
0x00000000 294912  xiang      600          524288    2        dest
0x00000000 655361  xiang      600          16777216   2        dest
0x00000000 425986  xiang      600          524288    2        dest
0x00000000 1441795 xiang      600          524288    2        dest
0x00000000 753668  xiang      600          524288    2        dest
0x00000000 1081349 xiang      600          524288    2        dest
0x00000000 1343494 xiang      600          524288    2        dest
0x00000000 1212423 xiang      600          524288    2        dest
0x00000000 1245192 xiang      600          67108864   2        dest
0x00000000 1540105 xiang      600          524288    2        dest
0x00000000 1802250 xiang      600          524288    2        dest
0x00000000 3244043 root       777          1024     0        dest
0x00000000 3276812 root       777          1024     0        dest
root@ubuntu:/home/xiang/app/mem# 

```

那么每次执行同样的程序就会在内存中多分配一次内存，如此执行下去内存也会占满，而且分配的很多内存都是没有使用的。

bytes	nattch	status
524288	2	dest
16777216	2	dest
524288	2	dest
67108864	2	dest
524288	2	dest
524288	2	dest
1024	0	dest

当然你可以用 ipcrm -m 输入 shmid 号来删除程序运行后遗留下来的内存空间，但是这样不科学，所以还是在代码里执行 IPC_RMID 靠谱些。

bytes	nattch	status
524288	2	dest
16777216	2	dest
524288	2	dest
67108864	2	dest
524288	2	dest
524288	2	dest
1024	0	dest
1024	0	dest

上面做法只实现了共享内存的单向传输，也就是父进程写子进程读，没有子进程写父进程读的双向功能。

Linux 网络编程

网络通信流程

```

#include <sys/types.h>           /* See NOTES */
#include <sys/socket.h>

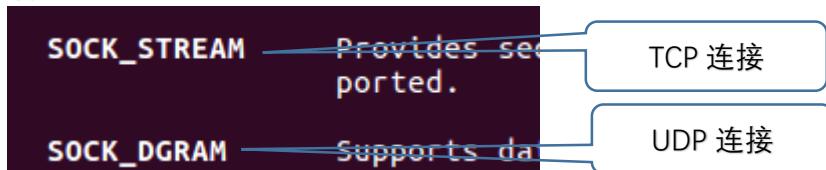
int socket(int domain, int type, int protocol);

```

domain 域：

Name	Purpose	IPV4 地址协议	Man page
AF_UNIX, AF_LOCAL	Local communication	unix(7)	
AF_INET	IPV4 Internet protocols	ip(7)	
AF_INET6	IPv6 Internet protocols	IPV6 地址协议	ipv6(7)
AF_IPX	IPX - Novell protocols		
AF_NETLINK	Kernel user interface device		netlink(7)

type 类型：



Protocol : 默认写 0

int : 返回值就是文件描述符，给其它函数使用的

bind 函数

```
#include <sys/types.h>           /* See NOTES */
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

Sockfd : 就是写入前面 socket 打开的网络文件描述符，文件 fd

const struct sockaddr : 就是绑定的 IP 地址，这个 IP 地址是服务器主机的 IP 地址，记住是服务器的 IP 地址不是客户端的 IP 地址，假设服务端有多个网卡，即会有多个 IP，这个时候我们可以选择用 bind 绑定其中一个 IP，那么服务端只接收该 IP 上某端口的数据，这个 sockaddr 是可以兼容 IPV4 和 IPV6 的。所以 bind 函数是用在服务器上面，客户端不需要这个函数。

写到这里我们就要对 IP 地址的转换做个介绍，因为 IP 地址会给 sockaddr 结构体

我们假定写的 IP 地址是 192.168.1.102 这种字符格式的，但是主机驱动会把这个字符 10 进制转换成 2 进制发送给硬件，所以我们要知道怎么把字符串转化成 16 进制，然后硬件自动帮我们转换成 2 进制。

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
```

将字符 IP 地址串转换成 16 进制

```

#include<stdio.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>

#define IPADDR "192.168.1.102"

int main(void)
{
    in_addr_t addr=0;
    addr=inet_addr(IPADDR); //点分10进制转换成16进制
    printf("addr = 0x %x\n",addr);

    struct in_addr inaddr;
    int ret=0;
    ret=inet_nton(AF_INET,IPADDR,&inaddr); //点分10进制转换成16进制
    if(ret!=1)
    {
        printf("inet_nton error\n");
        return -1;
    }
    printf("inaddr = 0x %x\n",inaddr.s_addr);
}

return 0;
}

```

输入 IP 地址

转换成 16 进制

写入网络要使用的协议 IPV4

字符串转换成 16 进制结果

这个和 inet_addr 功能一样，只是这个函数支持 IPV6 和 IPV4

两个函数接口 inet_addr 和 inet_nton 输出结果一样

```

root@ubuntu:/home/xiang/app/net/inet# ./inet
addr = 0x 6601a8c0
inaddr = 0x 6601a8c0

```

下面我们将 2 进制转换成字符串

```

1 #include<stdio.h>
2 #include<sys/socket.h>
3 #include<netinet/in.h>
4 #include<arpa/inet.h>
5
6
7 int main(void)
8 {
9     const char *ret=NULL;
10    struct in_addr addr;
11    addr.s_addr=0x6601a8c0;
12    char buf[50];
13
14    ret=inet_ntop(AF_INET,&addr,buf,sizeof(buf));
15    if(ret==NULL)
16    {
17        printf("inet_ntop error\n");
18        return -1;
19    }
20    printf("inaddr = %s\n",buf);
21
22
23
24
25 }

```

将 16 进制转换成字符串 IP 地址

输入 16 进制

转换后的字符串写入 buf

网络协议为 IPV4

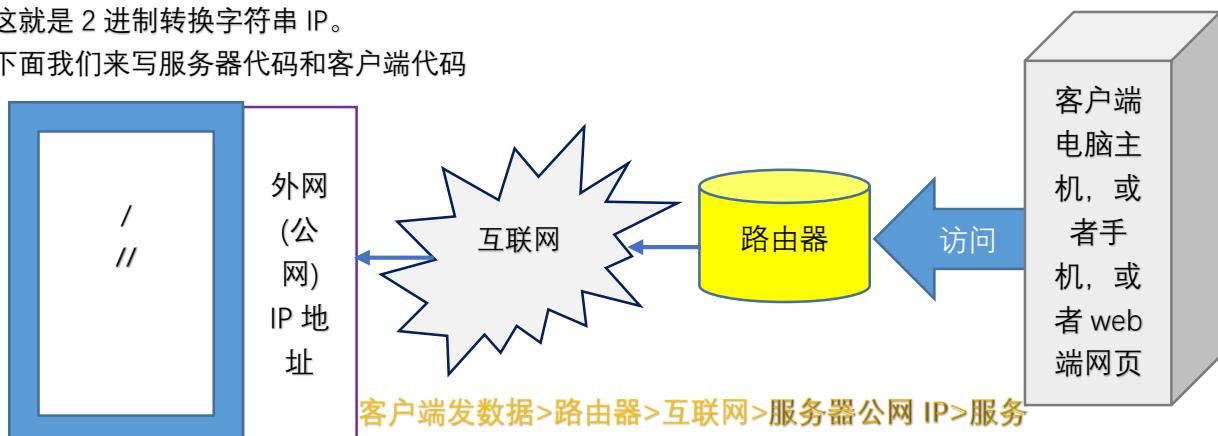
```

root@ubuntu:/home/xiang/app/net/inet# ./ntop
inaddr = 192.168.1.102
root@ubuntu:/home/xiang/app/net/inet# █

```

这就是 2 进制转换字符串 IP。

下面我们来写服务器代码和客户端代码



你在服务器上面 ifconfig 虽然看到的 IP 地址和公网 IP 地址不一样，但是这个服务器上的 IP 地址是等于公网的 IP 地址的，路由器用映射功能将公网 IP 映射到服务器主机上面，所以做服务器的主机必须是公网 IP，如果你一台服务器主机不够，要加一台服务器主机那么你就要去政府再申请一个公网 IP，两个公网 IP，两天服务器主机。注意服务器主机里面某个进程都是用端口号来表示

先创建服务器

```
1 #include<stdio.h>
2 #include<sys/socket.h> //包含socket
3 #include<arpa/inet.h> //包含sockaddr_in 包含 htons
4 #include<sys/types.h> //list
```

```
int main()
{
    int sockfd = -1;
    struct sockaddr_in serveraddr={}; //服务器IP地址和端口，协议的结构体
    struct sockaddr_in clientaddr={}; //接收到客户端的IP地址和端口号

    sockfd = socket(AF_INET,SOCK_STREAM,0);
    if(sockfd== -1)
    {
        perror("socket");
        return -1;
    }
    printf("socket ID success = %d\n",sockfd);

    serveraddr.sin_family = AF_INET; //IPv4
    serveraddr.sin_port = htons(6003); //端口号，就是服务器进程号，0-1025都被操作系统默认使用你就是使用1025以后的，最大不超过65535
    serveraddr.sin_addr.s_addr = inet_addr("192.168.1.104"); //这里我们用的inet_addr将字符串转换成2进制 这就是服务器主机IP地址，ifconfig查到的

    int ret=bind(sockfd,(const struct sockaddr *)&serveraddr,sizeof(serveraddr)); //将公网IP地址也就是服务器主机IP地址绑定到服务器程序上
    //然后将这个程序的端口也绑定到程序上，客户端就可以根据这个地址找到你这个服务器
```

第 1 步

第 2 步

当服务器接收 1 个客户端请求后，处理该客户端数据，但是在这个客户端数据没有处理完的时候又来了其它客户端的请求，这时候就只有把其它客服端放在队列里面，等待服务器把上一个客户端处理完了，在处理队列里面的其它客服端，就是去医院看医生排队是一个意思，这个 100，就是队列里面最多允许 100 个客户端等待，如果第 101 个客户端来请求就直接忽略掉，忽略掉的客服端就不处理了，其实服务器是单线程在处理客户端数据，并不是同时处理

第 3 步

第 4 步

```
//连接成功后，clientaddr将保存来链接我服务的客户端IP地址和端口，所以这个sockaddr和上面sockaddr使用上有点区别，上面那个sockaddr是写，我这个是读

    printf("connect success\n");

    return 0;
}
```

er.c" 48L, 1693C written

服务器程序成功启动了。

```
root@ubuntu:/home/xiang/app/net/sever# ./sever
socket ID success = 3
bind success
```

服务器阻塞了，等待客服端请求

我们现在来写客户端程序

```
#include<stdio.h>
#include<sys/socket.h> //包含socket
#include<arpa/inet.h> //包含sockaddr_in 包含 htons
#include<sys/types.h> //list

/*
*客户端和服务器的区别是：
*1.客户端没有端口号，因为是去链接服务器所以用服务器端口号
*2.客户端的bind的IP地址是服务器的公网IP地址
*/
int main()
{
    int sockfd = -1;
    struct sockaddr_in serveraddr={0}; //服务器IP地址和端口，协议的结构体

    sockfd = socket(AF_INET,SOCK_STREAM,0);
    if(sockfd== -1)
    {
        perror("socket");
        return -1;
    }
    printf("socket ID success = %d\n",sockfd);

    serveraddr.sin_family = AF_INET; //IPv4
    serveraddr.sin_port = htons(6003); //我们前面服务器建立的端口号
    serveraddr.sin_addr.s_addr = inet_addr("192.168.1.104"); //服务器IP地址

    int ret=connect(sockfd,(const struct sockaddr *)&serveraddr,sizeof(serveraddr));
    if(ret<0)
    {
        perror("connect");
        return -1;
    }

    printf("client connect success\n");

    return 0;
}
```

执行结果

```
root@ubuntu:/home/xiang/app/net/sever# ./sever
socket ID success = 3
bind success

connect success
root@ubuntu:/home/xiang/app/net/sever#
root@ubuntu:/home/xiang/app/net/sever#
root@ubuntu:/home/xiang/app/net/sever#
```

```
client.c
root@ubuntu:/home/xiang/app/net/client# gcc -o client.c
root@ubuntu:/home/xiang/app/net/client# ls
client.c
root@ubuntu:/home/xiang/app/net/client# ls
client.c
root@ubuntu:/home/xiang/app/net/client# ./client
socket ID success = 3
client connect success
root@ubuntu:/home/xiang/app/net/client#
```

服务器接收

客户端请求

两边链接成功

我们来用客户端给服务器发送数据

```
/*********************************************客户端******/
int main()
{
    int sockfd = -1;
    struct sockaddr_in serveraddr={0}; //服务器IP地址和端口， 协议的结构体

    sockfd = socket(AF_INET,SOCK_STREAM,0);
    if(sockfd== -1)
    {
        perror("socket");
        return -1;
    }
    printf("socket ID success = %d\n",sockfd);

    serveraddr.sin_family = AF_INET; //IPv4
    serveraddr.sin_port = htons(6003); //我们前面服务器建立的端口号
    serveraddr.sin_addr.s_addr = inet_addr("192.168.1.104"); //服务器IP地址

    int ret=connect(sockfd,(const struct sockaddr *)&serveraddr,sizeof(serveraddr));
    if(ret<0)
    {
        perror("connect");
        return -1;
    }

    printf("client connect success\n");

    char sendbuf[20];
    strcpy(sendbuf,"hello world");
    ret=send(sockfd,sendbuf,strlen(sendbuf),0); //发送数据给服务器
    printf("send char number = %d\n",ret);
    return 0;
}
```

```
/*********************************************服务器端*****/
int main()
{
    int sockfd = -1;
    struct sockaddr_in serveraddr={0};
    struct sockaddr_in clientaddr={0};

    sockfd = socket(AF_INET,SOCK_STREAM,0);
    if(sockfd== -1)
    {
        perror("socket");
        return -1;
    }
    printf("socket ID success = %d\n",sockfd);

    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(6003);
    serveraddr.sin_addr.s_addr = inet_addr("192.168.1.104");

    int ret=bind(sockfd,(const struct sockaddr *)&serveraddr,sizeof(serveraddr));
    if(ret<0)
    {
        perror("bind");
        return -1;
    }
    printf("bind success\n");

    ret = listen(sockfd,100);
    if(ret<0)
    {
        perror("listen");
        return -1;
    }

    int len;
    int clifd;
    clifd = accept(sockfd,(struct sockaddr *)&clientaddr,&len); //accept函数返回的fd和socket返回的fd不一样
    printf("server connect success\n");

    char recvbuf[100];
    ret=recv(clifd,recvbuf,sizeof(recvbuf),0); //接收客户端数据
    printf("recv number = %d\n",ret);
    printf("%s\n",recvbuf);
    return 0;
}
```

测试结果

```
root@ubuntu:/home/xiang/app/net/client# ls
client client.c
root@ubuntu:/home/xiang/app/net/client#
root@ubuntu:/home/xiang/app/net/client#
root@ubuntu:/home/xiang/app/net/client#
root@ubuntu:/home/xiang/app/net/client#
root@ubuntu:/home/xiang/app/net/client#
root@ubuntu:/home/xiang/app/net/client# ls
client client.c
root@ubuntu:/home/xiang/app/net/client#
socket ID success = 3
client connect success
send char number = 11
root@ubuntu:/home/xiang/app/net/client# 
```

```
root@ubuntu:/home/xiang/app/net/sever# ls
sever sever.c
root@ubuntu:/home/xiang/app/net/sever#
root@ubuntu:/home/xiang/app/net/sever# ./sever
socket ID success = 3
bind success
server connect success
recv number = 11
hello world
root@ubuntu:/home/xiang/app/net/sever#
root@ubuntu:/home/xiang/app/net/sever#
root@ubuntu:/home/xiang/app/net/sever# 
```

阻塞等待客户端数据
客户端数据

上面我们是客户端和服务器单次通信的，我下面做个多次通信的。

```
char sendbuf[20];
/*      strcpy(sendbuf,"hello world");
     ret=send(sockfd,sendbuf,strlen(sendbuf),0); //发送数据给服务器
     printf("send char number = %d\n",ret);*/
//*****将上面单次发送函数去掉，客户端循环发送***** */

while(1)
{
    printf("input send txt\n");
    scanf("%s",sendbuf);
    ret=send(sockfd,sendbuf,strlen(sendbuf),0); //发送数据给服务器
}
return 0;
```

```
char recvbuf[100];
ret=recv(clifd,recvbuf,sizeof(recvbuf),0); //接收客户端数据
printf("recv number = %d\n",ret);
printf("%s\n",recvbuf);
//*****服务端循环接收***** */

while(1)
{
    ret=recv(clifd,recvbuf,sizeof(recvbuf),0); //你会发现客户端的数据没有发送过来这个函数会阻塞
    printf("%s\n",recvbuf);
}

return 0;
```

测试结果

```
root@ubuntu:/home/xiang/app/net/client# ./client
socket ID success = 3
client connect success
input send txt      第一次发送
abcd
input send txt      第二次发送
eff
input send txt      第三次发送
aaaaaaa
```

```
root@ubuntu:/home/xiang/app/net/sever#
root@ubuntu:/home/xiang/app/net/sever#
root@ubuntu:/home/xiang/app/net/sever# ./sever
socket ID success = 3
bind success
server connect success
abcd
effd      第一次接收
abcd
effd      第二次接收
aaaaaaa
```

第一次发送
第二次发送
第三次发送
第一次接收
第二次接收

你会发现第二次接收出现了问题，为什么发送 eff 接收的是 effd 呢？其实那个 d 是上次 abcd 的 d，因为你接收后没有清空 recvbuf 缓存，所以字符就残留在哪里了，你后面发送的 eff 只是覆盖 abcd，但是你长度不够所以没有覆盖完，我发送一个长度够得试试。

```
client connect success
input send txt
abcd
input send txt
eff
input send txt
aaaaaaa
input send txt
```

```
root@ubuntu:/home/xiang/app/net/sever#
socket ID success = 3
bind success
server connect success
abcd
effd
aaaaaaa
```

你看发送长度够得就覆盖了。所以服务端程序要加个清空每次接受缓存的机制

```
*****服务端循环接收*****  
while(1)  
{  
    ret=recv(clifd,recvbuf,sizeof(recvbuf),0); //你会发现客户端的数据没有发送过来这个函数会阻塞  
    printf("%s\n",recvbuf);  
    memset(recvbuf,0,sizeof(recvbuf));  
}  
return 0;
```

修改服务端程序清空缓存

测试结果

```
root@ubuntu:/home/xiang/app/net/client# ./client  
socket ID success = 3  
client connect success  
input send txt  
abcd  
input send txt  
eff  
input send txt  
abcdefg  
input send txt  
abb  
input send txt
```

```
root@ubuntu:/home/xiang/app/net/sever# ./sever  
socket ID success = 3  
bind success  
  
server connect success  
abcd  
eff  
abcdefg  
abb
```

实现了实时发送接收的程序。但是这个程序还有个问题

```
input send txt  
abb  
^C  
root@ubuntu:/home/xiang/app/net/client# ./client  
root@ubuntu:/home/xiang/app/net/client# ./client  
root@ubuntu:/home/xiang/app/net/client# ./client  
root@ubuntu:/home/xiang/app/net/client# ./client
```

```
abb  
^C  
root@ubuntu:/home/xiang/app/net/sever# ./sever  
socket ID success = 3  
bind: Address already in use  
root@ubuntu:/home/xiang/app/net/sever# ./sever  
socket ID success = 3  
bind: Address already in use
```

当我 **ctrl+c** 强退服务器和客户端程序后，再次启动服务器出现了 **bind: Address already in use** 问题

这是因为我两边的程序是非正常关闭的，是人为强制退出的，正常退出主机上端口是会自动释放的，人为退出主机上端口可能还在占用着。这种情况下要过一会才可以启动服务器程序，这就是程序的 bug，程序没有加入退出机制。

下面我们让服务器给客户端发数据

前面的 socket 和链接代码都不用变，只需要改变发送和接收函数顺序就是了

```
13     int clifd;  
14     clifd = accept(sockfd,(struct sockaddr *)&clientaddr,&len); //accept函数返回的fd和socket返回的fd  
15     printf("server connect success\n");  
16  
17     char sendbuf[100];  
18     ****服务器发送数据*****  
19     strcpy(sendbuf,"server->client");  
20     ret = send(clifd,sendbuf,strlen(sendbuf),0);  
21     printf("server send number = %d\n",ret);  
22     return 0;
```

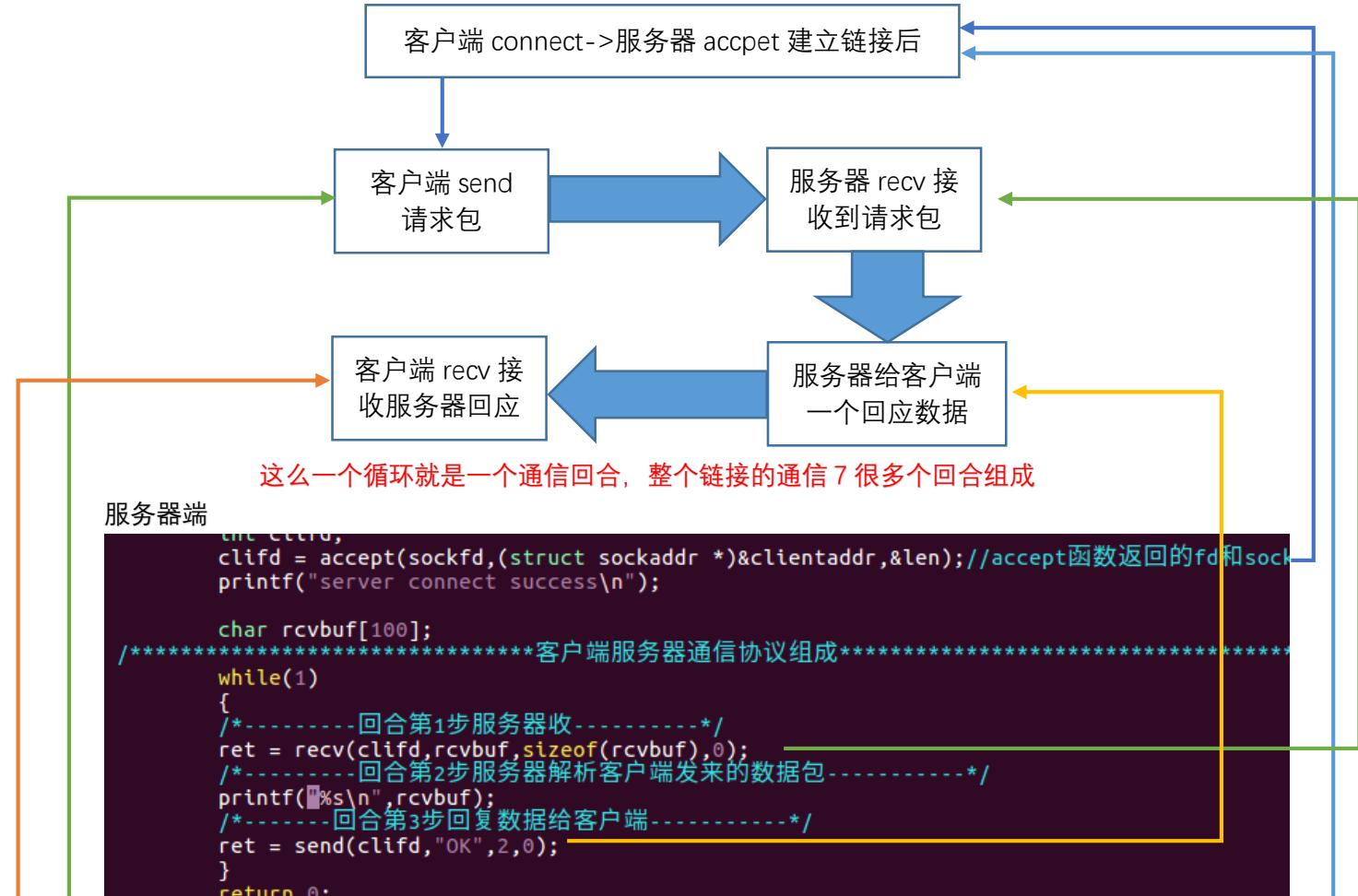
```
6     int ret=connect(sockfd,(const struct sockaddr *)&serveraddr,sizeof(serveraddr));  
7     if(ret<0)  
8     {  
9         perror("connect");  
10        return -1;  
11    }  
12  
13    printf("client connect success\n");  
14    char recvbuf[100];  
15  
16    ****客户端接收服务器数据*****  
17    ret = recv(sockfd,recvbuf,sizeof(recvbuf),0);  
18    printf("%s\n",recvbuf);  
19  
20    return 0;
```

客户端接收

```
root@ubuntu:/home/xiang/app/net/sever# ./sever  
socket ID success = 3  
bind success  
server connect success  
server send number = 14  
root@ubuntu:/home/xiang/app/net/sever#
```

以上网络程序的写法都是一方接收等待一方发送，有没有双工通信的方法呢？两边都可以随意收发。这个情况下就需要用到协议标准了，我们来自己定义一个网络协议。

1. 我们要规定一个发送和接收的协议，就是谁先发，谁后发



所有的协议都基于 http 啊，ftp 协议什么的都是基于这个回合开始向上进行封装的。

```
root@ubuntu:/home/xlang/app/net/client# ./client
socket ID success = 3
client connect success
abcdefg
send number = 100
server reply : OK
aaaaaaa
send number = 100
server reply : OK
vvvvvvvf
send number = 100
server reply : OK
zzzzzzzzaaaaaaaaaaaaaa
send number = 100
server reply : OK
bbb
send number = 100
server reply : OK
11111
send number = 100
server reply : OK
|||
```

```
root@ubuntu:/home/xlang/app/net/server#
socket ID success = 3
bind success

server connect success
abcdefg
aaaaaaa
vvvvvvvf
zzzzzzzzaaaaaaaaaaaaaa
bbb
11111
```

这就是测试结果

2. 我们要定义一个数据包的格式

```
#define CMD_REGISTER 1001 //注册学生信息
#define CMD_CHECK 1002 //检验学生信息
#define CMD_GETINFO 1003 //获取学生信息
typedef struct student
{
    char name[20];//学生名字
    int age; //学生年龄
    int cmd;
    struct other
    {
        char home_addr;//学生家庭地址
        int size; //家庭居住面积
    }oth;
}student;
```

在服务器程序和客户端程序里面定义一个这样的结构体，记住结构体两边必须是一样的。

```
clifd = accept(sockfd,(struct sockaddr *)&clientaddr,&len); //accept函数
printf("server connect success\n");

*****客户端服务器通信协议组成*****
student serst2;
while(1)
{
    /*-----回合第1步服务器收-----*/
    ret = recv(clifd,&serst2,sizeof(serst2),0);
    /*-----回合第2步服务器解析客户端发来的数据包-----*/
    if(serst2.cmd==CMD_REGISTER)
    {
        printf("register student\n");
        printf("student name = %s\n",serst2.name);
        printf("student age = %d\n",serst2.age);

        ret = send(clifd,"register OK",11,0);
    }
    if(serst2.cmd==CMD_CHECK)
    {

        ret = send(clifd,"check OK",8,0);
    }
    if(serst2.cmd==CMD_GETINFO)
    {

        ret = send(clifd,"getinfo OK",10,0);
    }
    /*-----回合第3步回复数据给客户端-----*/
}
```

服务器端通信协议，还是修改传入 recv 和 send 的数据结构

```

#define CMD_REGISTER 1001 //注册学生信息
#define CMD_CHECK 1002 //检验学生信息
#define CMD_GETINFO 1003 //获取学生信息
typedef struct student
{
    char name[20];//学生名字
    int age; //学生年龄
    int cmd;
    struct other
    {
        char home_addr;//学生家庭地址
        int size; //家庭居住面积
    }oth;
}student;

```

数据结构两边都是一样的

```

int ret=connect(sockfd,(const struct sockaddr *)&serveraddr,sizeof(serveraddr));
if(ret<0)
{
    perror("connect");
    return -1;
}

printf("client connect success\n");

char recvbuf[100];

*****客户端服务器通信协议组成*****
student st1;
while(1)
{
printf("please input student name\n");
scanf("%s",st1.name);
printf("please input student age\n");
scanf("%d",&st1.age);
st1.cmd=CMD_REGISTER;
/*-----回合第1步发送数据给服务器-----*/
ret = send(sockfd,&st1,sizeof(st1),0);
printf("send number = %d\n",ret);
/*-----回合第2步客户端接收服务器回复-----*/
memset(recvbuf,0,sizeof(recvbuf));//清空上次接收的数据
ret = recv(sockfd,recvbuf,sizeof(recvbuf),0);
/*-----回合第3步客户端解析服务器的回复-----*/

printf("server reply : %s\n",recvbuf);
}

```

和服务端一样的都是修改传入 recv 和 send 的数据结构

```

root@ubuntu:/home/xiang/app/net/client# ./client
socket ID success = 3
client connect success
abcdefg
send number = 100
server reply : OK
aaaaaaaa
send number = 100
server reply : OK
vvvvvvff
send number = 100
server reply : OK
zzzzzzzaaaaaaaaaaaa
send number = 100
server reply : OK
bbb
send number = 100
server reply : OK
111111
send number = 100
server reply : OK

```

```

root@ubuntu:/home/xiang/app/net/sever#
./sever
socket ID success = 3
bind success

server connect success
abcdefg
aaaaaaaa
vvvvvvff
zzzzzzzaaaaaaaaaaaa
bbb
111111

```

客户端输入学生信息，服务器接收后处理，服务器再返回一个数据给客户端，http, ftp 都是这样玩的。

Linux 高级 IO 使用，阻塞和非阻塞应用场景

阻塞方式

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #include<string.h>
4
5 int main(void)
6 {
7
8     char buf[100];
9     memset(buf, 0, sizeof(buf));
10    printf("input keyboard char\n");
11    read(0, buf, 2);
12    printf("read = %s\n", buf);
13
14    return 0;
15
16 }
```

打开键盘句柄，因为系统启动默认打开键盘了，所以我们不用再 OPEN，键盘默认 fd 是 0

运行试试

```
root@ubuntu:/home/xiang/app/IO# ./readkeyboard  
input keyboard char
```

我们键盘没有输入字符时 程序阻塞在 read 函数这里。

```
root@ubuntu:/home/xiang/app/IO# ./readkeyboard  
input keyboard char  
12  
read = 12  
root@ubuntu:/home/xiang/app/IO#
```

我们输入字符，字符放在缓存区

回车之后字符从缓存区打印出来

```
root@ubuntu:/home/xiang/app/IO# ./readkeyboard  
input keyboard char  
546+ _____  
read = 54           输入字符超过 read 读取个数, 会报错  
root@ubuntu:/home/xiang/app/IO# 6+  
6+: command not found  
root@ubuntu:/home/xiang/app/IO#
```

我鼠标晃动没反应

我们在来看看使用鼠标，鼠标和键盘不一样，要先找到设备，所以我在`/dev/input`下面查找

```
root@ubuntu:/home/xiang/app/I0# ls /dev/input/  
by-id by-path event0 event1 event2 event3 mice mouse0 mouse1 mouse2  
root@ubuntu:/dev/input# cat mouse1  
我鼠标晃动没反应  
root@ubuntu:/dev/input# cat mouse2  
我鼠标晃动有反应，说明是 mouse0  
root@ubuntu:/dev/input# cat mouse0  
我鼠标晃动没反应
```

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #include<string.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <fcntl.h>
7
8 int main(void)
9 {
10
11     int fd = -1;
12     char buf[100];
13
14     fd=open("/dev/input/mouse0",O_RDONLY);
15     if(fd<0)
16     {
17         perror("open");
18         return -1;
19     }
20
21     memset(buf,0,sizeof(buf));
22     printf("input mouse char\n");
23     read(fd,buf,100);
24     printf("read = %s\n",buf);
25
26     return 0;
27
28 }
```

运行程序

```
root@ubuntu:/home/xiang/app/IO# ./readmouse
input mouse char
read = 0%
```

我晃动了鼠标，有字符输出，但是为什么输出只有两个，这个和系统有关，其实标准读鼠标数据不是用 read 来做，这里只是为了方便观察。以上在单程序里面读鼠标键盘都很成功

如果我再一个程序里面同时执行鼠标和键盘功能呢？

```

2 #include<stdio.h>
3 #include<unistd.h>
4 #include<string.h>
5 #include <sys/types.h>
6 #include <sys/stat.h>
7 #include <fcntl.h>
8
9
10 int main(void)
11 {
12
13     int fd = -1;
14     char buf[100];
15     memset(buf,0,sizeof(buf));
16     printf("input keyboard char\n");
17     read(0,buf,2);
18     printf("readkeyborad = %s\n",buf);
19
20
21     fd=open("/dev/input/mouse0",O_RDONLY);
22     if(fd<0)
23     {
24         perror("open");
25         return -1;
26     }
27
28     memset(buf,0,sizeof(buf));
29     printf("input mouse char\n");
30     read(fd,buf,100);
31     printf("readmouse = %s\n",buf);
32
33 }
34 
```

```

root@ubuntu:/home/xiang/app/IO# ./readmouskey
input keyboard char
readkeyborad = 
```

我晃动鼠标没有反应

```

root@ubuntu:/home/xiang/app/IO# ./readmouskey
input keyboard char
55
readkeyborad = 55
input mouse char
readmouse = 800
root@ubuntu:/home/xiang/app/IO# 
```

我先按了键盘，然后在晃动鼠标才有反应

证明了阻塞方式只能等前面被阻塞的键盘程序执行后，后面鼠标程序才能执行，这种就不科学

针对上面鼠标键盘不能同时执行的问题我们可以采用多进程方式，当然这是解决方法的一种，但是我们今天要做的是用并行 IO 来解决。

非阻塞方式的并行 IO

```

int main(void)
{
    int fd = -1;
    char buf[100];
    int flag=-1;

    flag=fcntl(0,F_GETFL); //先获取原来OPEN文件句柄的flag
    flag|=O_NONBLOCK; //添加非阻塞属性
    fcntl(0,F_SETFL,flag); //写入文件句柄

    memset(buf,0,sizeof(buf));
    printf("input keyboard char\n");
    read(0,buf,2);
    printf("readkeyborad = %s\n",buf);

    fd=open("/dev/input/mouse0",O_RDONLY);
    if(fd<0)
    {
        perror("open");
        return -1;
    }

    memset(buf,0,sizeof(buf));
    printf("input mouse char\n");
    read(fd,buf,100);
    printf("readmouse = %s\n",buf);

    return 0;
} 
```

因为系统启动时就 OPEN 键盘为阻塞功能了，如果你用 OPEN 方式再去打开键盘会返回错误，所以我无法用 OPEN 去改键盘的阻塞方式现在唯一的方法就是获取 OPEN 键盘后的句柄，因为键盘句柄为 0 我们用 fcntl 函数获取它来操作

输出结果：

```

root@ubuntu:/home/xiang/app/IO# ./readmouskey
input keyboard char
readkeyborad =
input mouse char
readmouse =
root@ubuntu:/home/xiang/app/IO# 
```

根据输出结果来看不并不是我想要的，我连输入键盘字符机会都没有程序就执行完了。

```
int main(void)
{
    int fd = -1;
    char buf[100];
    int flag=-1;

    flag=fcntl(0,F_GETFL); //先获取原来OPEN文件句柄的flag
    flag|=O_NONBLOCK; //添加非阻塞属性
    fcntl(0,F_SETFL,flag); //写入文件句柄

    fd=open("/dev/input/mouse0",O_NONBLOCK); //直接设置鼠标文件句柄为非阻塞
    if(fd<0)
    {
        perror("open");
        return -1;
    }

    while(1){
        memset(buf,0,sizeof(buf));
        read(0,buf,2);
        printf("readkeyborad = %s\n",buf);

        memset(buf,0,sizeof(buf));
        read(fd,buf,100);
        printf("readmouse = %s\n",buf);
    }
    return 0;
}
```

```
readkeyborad =
readmouse =
rea^C
root@ubuntu:/home/xiang/app/IO#
```

```
int main(void)
{
    int fd = -1;
    char buf[100];
    int flag=-1;
    int ret = -1;

    flag=fcntl(0,F_GETFL); //先获取原来OPEN文件句柄的flag
    flag|=O_NONBLOCK; //添加非阻塞属性
    fcntl(0,F_SETFL,flag); //写入文件句柄

    fd=open("/dev/input/mouse0",O_NONBLOCK); //直接设置鼠标文件句柄为非阻塞
    if(fd<0)
    {
        perror("open");
        return -1;
    }

    while(1){
        memset(buf,0,sizeof(buf));
        ret = read(0,buf,2); //read读取数据后会返回读取数据的个数，如果没有数据read会返回0
        if(ret >0){ //如果大于0，说明有数据
            printf("readkeyborad = %s\n",buf);
        }
        memset(buf,0,sizeof(buf));
        ret = read(fd,buf,100);
        if(ret > 0){
            printf("readmouse = %s\n",buf);
        }
    }
    return 0;
}
```

```
c
root@ubuntu:/home/xiang/app/IO# ./readmouskey
ss
readkeyborad = ss
readkeyborad =
dd
readkeyborad = dd
readkeyborad =
bb
readkeyborad = bb
readkeyborad =
readmouse = 8**#
readkeyborad = ss
readkeyborad =
aa
readkeyborad = aa
readkeyborad =
11
readkeyborad = 11
readkeyborad =
readmouse = 8**#
^C
root@ubuntu:/home/xiang/app/IO#
```

这样就对了 read 有数据会执行 printf，没有数据 printf 也不会刷屏

这种死循环方式有个问题就是太耗费 CPU 资源

IO 多路复用

用 select 函数来解决上面死循环非阻塞读取键盘的问题。

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #include<string.h>
4 #include<sys/types.h>
5 #include<sys/stat.h>
6 #include<fcntl.h>
7 #include<sys/select.h> //用select函数需要这个头文件
8 #include<sys/time.h> //select函数需要用到超时，所以需要这个头文件
```

```

9 int main(void)
10 {
11
12     int fd = -1;
13     char buf[100];
14     int flag=-1;
15     int ret = -1;
16
17     flag=fcntl(0,F_GETFL); //先获取原来OPEN文件句柄的flag
18     flag|=O_NONBLOCK; //添加非阻塞属性
19     fcntl(0,F_SETFL,flag); //写入文件句柄
20
21     fd=open("/dev/input/mouse0",O_NONBLOCK); //直接设置鼠标文件句柄为非阻塞
22     if(fd<0)
23     {
24         perror("open");
25         return -1;
26     }
27
28     fd_set myfdset;
29     FD_ZERO(&myfdset);
30     FD_SET(fd,&myfdset);
31     FD_SET(0,&myfdset);
32
33     struct timeval tm;
34
35     tm.tv_sec=10;
36     tm.tv_usec=0;

```

因为一个 select 可以同时监控很多个文件 ID 是否有输入发生，所以这里要在使用 select 之前把需要监控的文件 ID 写入 fd_set 结构，我这里监控键盘和鼠标，所以就 FD_SET 两个文件

设置 select 超时时间，我设置的 10 秒，usec 是微妙，我写 0，时间到了不管有没有按键按下，我都要跳过 select 的阻塞，去执行 select 返回 0 的那个判断代码

键盘文件 ID 是 0，鼠标文件 ID 是 fd，因为键盘是最小的文件 ID，这里要求填入最大的文件 ID+1，所以我们填入鼠标的 ID

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
          fd_set *exceptfds, struct timeval *timeout);
```

```

ret=select(fd+1,&myfdset,NULL,NULL,&tm);
if(ret<0)
    printf("select error\n");
else if(ret == 0)
    printf("select timeout\n");
else
{

```

超时时间

因为我们只有读操作，所
以后面两个为 NULL

```

    if(FD_ISSET(0,&myfdset)){
        memset(buf,0,sizeof(buf));
        ret = read(0,buf,2); //read读取数据后会返回读取数据的个数，如果没有数据read会返回0
        if(ret >0){
            printf("readkeyborad = %s\n",buf);
        }
    }
    if(FD_ISSET(fd,&myfdset)){
        memset(buf,0,sizeof(buf));
        ret = read(fd,buf,100);
        if(ret > 0){
            printf("readmouse = %s\n",buf);
        }
    }
}

```

FD_ISSET 判读键盘是否
按下，按下返回 1

FD_ISSET 判读鼠标是否
按下，按下返回 1

FD_ISSET (写入你要判断的设备文件 ID，写入 fd_set 结构)

测试结果：

鼠标输入

```

root@ubuntu:/home/xiang/app/IO/IO_mulist#
root@ubuntu:/home/xiang/app/IO/IO_mulist# ./readmouskey
readmouse = 8♦
root@ubuntu:/home/xiang/app/IO/IO_mulist#
root@ubuntu:/home/xiang/app/IO/IO_mulist# ./readmouskey
ls
readkeyborad = ls
root@ubuntu:/home/xiang/app/IO/IO_mulist#
root@ubuntu:/home/xiang/app/IO/IO_mulist# ./readmouskey
select timeout
root@ubuntu:/home/xiang/app/IO/IO_mulist#

```

用 select 函数的好处是

- 1, CPU 执行到 select 没有按键按下，进程就阻塞休眠，不会像上面死循环那样一直轮询占用 CPU 资源
- 2, 按键或者鼠标按下 select 就阻塞结束，但是不会向前面那样等到按键按下才能执行后面的鼠标事件。可以相互同时执行。

用 poll 函数来解决上面死循环非阻塞读取键盘的问题。

```

1 #include<stdio.h>
2 #include<unistd.h>
3 #include<string.h>
4 #include<sys/types.h>
5 #include<sys/stat.h>
6 #include<fcntl.h>
7 #include<poll.h> //用poll函数需要这个头文件
8
9 int main(void)
0 {
1
2     int fd = -1;
3     char buf[100];
4     int flag=-1;
5     int ret = -1;
6
7     flag=fcntl(0,F_GETFL); //先获取原来OPEN文件句柄的flag
8     flag|=O_NONBLOCK; //添加非阻塞属性
9     fcntl(0,F_SETFL,flag); //写入文件句柄
0
1     fd=open("/dev/input/mouse0",O_NONBLOCK); //直接设置鼠标文件句柄为非阻塞
2     if(fd<0)
3     {
4         perror("open");
5         return -1;
6     }
7
8     struct pollfd mypollfd[2]={0}; //定义poll结构，有多少个文件需要监控，最大数组号就写多少
9
0     mypollfd[0].fd=0; //0号数组的fd监控键盘
1     mypollfd[0].events=POLLIN; //监控文件的类型是键盘，所以填写输入类型
2     mypollfd[1].fd=fd; //1号数组的fd监控鼠标
3     mypollfd[1].events=POLLIN; //监控文件的类型是鼠标，所以填写输入类型
4

```

struct pollfd {
 int fd; /* file descriptor */
 short events; /* requested events */
 }

```

34     while(1)
35     {
36         ret=poll(mypollfd,fd+1,5000);
37         if(ret<0)
38             printf("select error\n");
39         else if(ret == 0)
40             printf("select timeout\n");
41         else
42         {
43
44             if(mypollfd[0].events==mypollfd[0].revents){
45                 memset(buf,0,sizeof(buf));
46                 ret = read(0,buf,2); //read读取数据后会返回读取数据的个数，如果没有数据read会返回0
47                 printf("readkeyborad = %s\n",buf);
48
49         }
50
51             if(mypollfd[1].events==mypollfd[1].revents){
52                 memset(buf,0,sizeof(buf));
53                 ret = read(fd,buf,100);
54                 printf("readmouse = %s\n",buf);
55
56         }
57     }
58 }
59
60
61     return 0;
62 }
63 }
```

```

struct pollfd {
    int fd;          /* file descriptor */
    short events;    /* requested events */
    short revents;   /* returned events */
};
```

Poll 用 revents 判断该文件是否有按键发生

程序测试

```

root@ubuntu:/home/xiang/app/io/io_mulist# ./readmouskey
gggg
readkeyborad = gg
readkeyborad = gg
readkeyborad = g
ssss
readkeyborad = ss
readkeyborad = ss
readkeyborad = s
readmouse = 8+++
readmouse = ++
readmouse = ++
readmouse =
rc
```

程序经过测试，发现鼠标和键盘都可以同时运行，不需要等待谁运行完才能运行自己，和 select 一样的功能。

poll 和 select 其实是一样的，看你喜欢用哪个。

用软件模拟外部中断强制进程执行自己定义的中断服务程序

这就是异步 IO

```

#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include <signal.h> //异步功能signal头文件

int fd = -1;
```

```

2 void func(int sig)
3 {
4     char buf[100]={0};
5     if(sig!=SIGIO)
6         return ;
7
8     read(fd,buf,50); //read读取数据后会返回读取数据的个数，如果没有数据read会返回0
9     printf("readmouse = %s\n",buf);
10 }
11
12 int main(void)
13 {
14     int flag=-1;
15     int ret = -1;
16     char mbuf[100];
17
18     fd=open("/dev/input/mouse0",O_RDONLY); //直接设置鼠标文件句柄为非阻塞
19     if(fd<0)
20     {
21         perror("open");
22         return -1;
23     }
24
25     flag=fcntl(fd,F_GETFL); //先获取原来OPEN文件句柄的flag
26     flag|=O_ASYNC; //设置键盘文件句柄为异步通知，到时候接收到软件中断，程序会跑到异步信号服务程序里面去执行
27     fcntl(fd,F_SETFL,flag); //功能设置好后，重新写入该文件
28
29     /*哪个进程接收键盘发送过来的中断信号？*/
30     fcntl(fd,F_SETOWN,getpid()); //我设置当前进程接收键盘发送过来的信号
31
32     signal(SIGIO,func); //注册SIGIO发生后要执行的函数名
33
34     while(1)
35     {
36         memset(mbuf,0,sizeof(mbuf));
37         ret = read(0,mbuf,100);
38         printf("readkeyborad = %s\n",mbuf);
39
40     }
41
42     return 0;
43 }

```

测试结果

```

root@ubuntu:/home/xiang/app/IO/yibuo# ./readmouskey
sss
readkeyborad = sss

ffff
readkeyborad = ffff

abcdefg
readkeyborad = abcdefg

readmouse = 8♦♦
readmouse = ♦♦♦
readmouse =

```

F_SETOWN 设置该鼠标文件句柄具有接收 SIGIO 的功能，指定 SIGIO 信号发给当前进程

用异步 SIGIO 也能实现键盘鼠标同时运行，同时获取输入事件的功能，和上面 select, poll 得到的结果一样

但是把键盘放入异步信号中断，鼠标放入死循环就要出现问题，不知道为什么？

Linux 线程使用

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<unistd.h>
4 #include<stdlib.h>
5 #include<pthread.h> //pthread_create线程函数头文件
6 #include<sys/types.h>
7 #include<sys/stat.h>
8 #include<fcntl.h>
9 #include<string.h> //memset函数头文件
10
11 void func(int *arg) ←
12 {
13     char buf[50];
14     while(1)
15     {
16         memset(buf,0,sizeof(buf));
17         read(0,buf,50);
18         printf("读出键盘内容 : %s\n",buf);
19     }
20 }
21
22 void main()
23 {
24     int ret = -1,fd = -1;
25     pthread_t th = -1;
26
27     ret=pthread_create(&th,NULL,(void *)func,NULL);
28     if(ret!=0)
29         printf("func creat pthread error\n");
30
31     char buf[100];
32
33     fd=open("/dev/input/mouse0",O_RDONLY);
34     if(fd<0)
35     {
36         perror("open");
37         return ;
38     }
39     while(1){
40         memset(buf,0,sizeof(buf));
41         read(fd,buf,100);
42         printf("读出鼠标内容 : %s\n",buf);
43     }
44 }
45
46 }
```

线程函数创建

可以给线程函数传入参数

创建的线程函数

用线程同时读鼠标键盘，完成上面异步IO 和 IO 多路复用来实现同时读鼠标键盘的功能。

```

root@ubuntu:/home/xiang/app/thread# gcc -o thread thread.c -lpthread
root@ubuntu:/home/xiang/app/thread# ./thread
读出鼠标内容 : ##
读出鼠标内容 : ##
读出鼠标内容 :
读出鼠标内容 :
读出鼠标内容 :
读出鼠标内容 :
读出鼠标内容 :
aaa
读出键盘内容 : aaa
bbb
读出键盘内容 : bbb

```

编译的时候要用-l去链接动态库.so 因为 pthread 函数实现是单独的动态库实现，没有包含到 gcc 里面

根据上面的例子发现线程创建的函数和主程序可以同时运行。

Linux 进程同步方法

```

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<stdlib.h>
#include<pthread.h> //pthread_create线程函数头文件
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<string.h>/memset函数头文件

void main()
{
    char buf[200];
    int i;
    printf("输入一个字符串，请回车结束\n");
    while(scanf("%s",buf))
    {
        printf("本次输入 %d 个字符\n",i=strlen(buf));
        memset(buf,0,sizeof(buf));
    }
}

```

键盘输入字符统计程序

```

root@ubuntu:/home/xiang/app/thread# gcc -o threadtt threadtt.c -lpthread
root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串，请回车结束
aaa
本次输入 3 个字符
ccccc
本次输入 5 个字符
zzzdddaaa
本次输入 9 个字符
end
本次输入 3 个字符

```

这里有个问题就是每次输入字符按回车可以得到统计，但是输入 end 无法退出统计程序

修改代码如下

```

void main()
{
    char buf[200];
    int i;
    printf("输入一个字符串, 请回车结束\n");
    while(scanf("%s",buf))
    {
        /*为了解决输入end无法退出程序问题, 我们修改了一下*/
        if(strncmp(buf,"end",3)==0)
        {
            printf("程序结束\n");
            exit(0);
        }
        printf("本次输入 %d 个字符\n",i=strlen(buf));
        memset(buf,0,sizeof(buf));
    }
}

```

root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串, 请回车结束

aaa
本次输入 3 个字符

bbbbbb
本次输入 6 个字符

xxxxxxxxeeee

本次输入 12 个字符

end

程序结束

root@ubuntu:/home/xiang/app/thread#

输入 end 问题解决了

但是有个 bug 还是没有解决

root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串, 请回车结束

aaaa
本次输入 4 个字符

endaaaa

程序结束

root@ubuntu:/home/xiang/app/thread# ./threadtt 我不仅输入 end 后面还加入了其它字符, 程序也算结束, 其实 strncmp 是以第一个字符开始向后计算 2 个, 看这三个对不对, 至于后面加了多少字符 strncmp 不管, 死个舅子认为前面三个字符正确就行了。

我们现在将记录字符个数功能放到子线程里面

```

char buf[200];

void func(int *arg)
{
    int i;
    printf("本次输入 %d 个字符\n", i=strlen(buf));
}

void main()
{
    int ret;
    pthread_t th = -1;

    ret=pthread_create(&th,NULL,(void *)func,NULL);
    printf("输入一个字符串, 请回车结束\n");
    while(scanf("%s",buf))
    {
        /*为了解决输入end无法退出程序问题, 我们修改了一下*/
        if(strcmp(buf,"end",3)==0)
        {
            printf("程序结束\n");
            break;
        }
        memset(buf,0,sizeof(buf));
    }

    ret=pthread_join(th,NULL); //回收子线程
    if(ret!=0)
    {
        printf("子线程回收失败\n");
        exit(-1);
    }
    printf("子线程回收成功\n");
}

```

```

root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串, 请回车结束
本次输入 0 个字符
asadadasd
end
程序结束
子线程回收成功
root@ubuntu:/home/xiang/app/thread#

```

回收子线程成功返回 0

就是子线程先执行主进程还没接受到字符输入, 所以子线程只有输出空 buf

子线程创建之后就马上执行子线程, 但是我主线程还没有输入字符

而且现在问题是如果不输入 end, 那么子线程已经先执行完了, 主进程等待字符输入子线程还没有被回收那么我给子线程加死循环不就行了吗, 这样是可以, 但是子线程一直刷屏你舒服吗。

我们用信号量来解决这个问题

```

7 #include<sys/types.h>
8 #include<fcntl.h>
9 #include<string.h> //memset函数头文件
10 #include<semaphore.h> //线程信号量
11
12 char buf[200];
13 sem_t sem; // 初始化一个信号量，如果有多个不同的信号量就要多定义几个
14
15 void func(int *arg)
16 {
17     int i;
18     while(1)
19     {
20         sem_wait(&sem); 当 sem 为 0 是线程就阻塞到这里，如果 sem 的值大于 0，该阻塞就消失，而且附带对 sem 减 1 的功能
21         printf("本次输入 %d 个字符\n", i = strlen(buf));
22         memset(buf, 0, sizeof(buf));
23     }
24 }
25
26 void main()
27 {
28     int ret;
29     pthread_t th = -1;
30
31     sem_init(&sem, 0, 0); sem_init(信号量类型, 为 0 表示该信号量只有该进程可以用其他进程不能用, 信号量值)
32     ret = pthread_create(&th, NULL, (void *)func, NULL);
33     printf("输入一个字符串，请回车结束\n");
34     while (scanf("%s", buf))
35     {
36         /*为了解决输入end无法退出程序问题，我们修改了一下*/
37         if (strcmp(buf, "end", 3) == 0)
38         {
39             printf("程序结束\n");
40             break;
41         }
42         sem_post(&sem); 信号量值加 1，这样线程的 sem_wait 才不会阻塞
43     }
44
45     ret = pthread_join(th, NULL); //回收子线程
46     if (ret != 0)
47     {
48         printf("子线程回收失败\n");
49         exit(-1);
50     }
51     printf("子线程回收成功\n");
52 }

```

```

root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串，请回车结束
aaa
本次输入 3 个字符
vvvvvv
本次输入 6 个字符
end
程序结束

```

这样就解决了上面线程刷屏的问题，但是程序结束后貌似线程还没死

上面这个问题是因为线程一直处于死循环，没有退出循环机子

```

void func(int *arg)
{
    int i;
    while(1)
    {
        sem_wait(&sem);
        printf("本次输入 %d 个字符\n", i=strlen(buf));
        memset(buf, 0, sizeof(buf));
    }
}

```

因为线程没有退出循环，所以就算主进程回收子线程，但是子线程还是卡着的

唯一的解决方法就是主进程结束之前先执行子线程退出

```

char buf[200];
sem_t sem;
unsigned int flag=0;

void func(int *arg)
{
    int i;
    while(flag==0)
    {
        sem_wait(&sem);
        printf("本次输入 %d 个字符\n", i=strlen(buf));
        memset(buf, 0, sizeof(buf));
    }
}

void main()
{
    int ret;
    pthread_t th = -1;

    sem_init(&sem, 0, 0);
    ret(pthread_create(&th, NULL, (void *)func, NULL));
    printf("输入一个字符串，请回车结束\n");
    while(scanf("%s", buf))
    {
        /*为了解决输入end无法退出程序问题，我们修改了一下*/
        if(strncmp(buf, "end", 3)==0)
        {
            printf("程序结束\n");
            flag=1;
            break;
        }
        sem_post(&sem);
    }

    ret(pthread_join(th, NULL)); //回收子线程
    if(ret!=0)
    {
        printf("子线程回收失败\n");
        exit(-1);
    }
    printf("子线程回收成功\n");
}

```

加一个标志位

让 flag=0 时死循环，不等于 0 时退出循环

输入 end 退出程序时让 flag 不等于 0

```

root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串，请回车结束

```

aaa
本次输入 3 个字符

bbbbbb
本次输入 5 个字符
end
程序结束

但是这个问题还是没有解决

```

char buf[200];
sem_t sem;
unsigned int flag=0;

void func(int *arg)
{
    int i;
    while(flag==0)
    {
        sem_wait(&sem);
        printf("本次输入 %d 个字符\n", i=strlen(buf));
        memset(buf, 0, sizeof(buf));
    }
}

void main()
{
    int ret;
    pthread_t th = -1;

    sem_init(&sem, 0, 0);
    ret=pthread_create(&th,NULL,(void *)func,NULL);
    printf("输入一个字符串, 请回车结束\n");
    while(scanf("%s",buf))
    {
        /*为了解决输入end无法退出程序问题, 我们修改了一下*/
        if(strncmp(buf, "end", 3)==0)
        {
            printf("程序结束\n");
            flag=1;
            sem_post(&sem);
            break;
        }
        sem_post(&sem);
    }

    ret=pthread_join(th,NULL); //回收子线程
    if(ret!=0)
    {
        printf("子线程回收失败\n");
        exit(-1);
    }
    printf("子线程回收成功\n");
}

```

我在这里加一个 sem_post 是
因为我在执行这个退出进程
函数的时候, 子线程还在
sem_wait 等待, 不知道我退
出了所以我要用这个 post 发
信号量给子线程, 让子线程
再次循环发现 while 不等于
0, 就退出死循环了

```

root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串，请回车结束
aaaa
本次输入 4 个字符
bbbb
本次输入 4 个字符
end
程序结束
本次输入 3 个字符
子线程回收成功
root@ubuntu:/home/xiang/app/thread#

```

你看没有问题了。

sem_init 初始化参数问题



```

root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串，请回车结束

```

第一次运行程信号量为 0, 所以子线程卡在这里

当我输入字符后程序就会跳过 scanf 函数, 执行下面的 sem_post

```

root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串，请回车结束

```

```

aaaa
本次输入 4 个字符

```

因为 sem 信号加 1 了, 所以子线程不会阻塞到 sem_wait 这里会继续执行, 顺便附带减 1 个 sem
然后子线程执行完打印后, 在循环一遍发现 sem=0, 只有在此阻塞在这里了。等到键盘输入字符让 post 在次发信号给子线程, 取消阻塞。

如果我将 sem_init 设置成加 5 会怎么样？

```
sem_init(&sem,0,5); // Sem , 0 , 5
ret=pthread_create(&th,NULL,(void *)func,NULL);
printf("输入一个字符串, 请回车结束\n");
while(scanf("%s",buf))
{
    /*为了解决输入end无法退出程序问题, 我们修改了一下*/
    if(strncmp(buf,"end",3)==0)
    {
        printf("程序结束\n");
        break;
    }
    sem_post(&sem);
}

void func(int *arg)
{
    int i;
    while(1)
    {
        sem_wait(&sem);
        printf("本次输入 %d 个字符\n",i=strlen(buf));
        memset(buf,0,sizeof(buf));
    }
}
```

```
root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串, 请回车结束
本次输入 0 个字符
aaa
本次输入 3 个字符
```

为什么线程会先执行 5 次,
我没有 post 啊？那是因为我
在初始化加了 5 个信号，所
以必须等到线程把这 5 个信
号减完，sem 才会是 sem, 0,
0 这个状态，线程才阻塞，等
到进程发 post

Linux 线程互斥锁

```
void func(int *arg)
{
    int i;
    while(1)
    {
        pthread_mutex_lock(&mutex);
        printf("本次输入 %d 个字符\n", i=strlen(buf));
        memset(buf, 0, sizeof(buf));
        pthread_mutex_unlock(&mutex);
    }
}

void main()
{
    int ret;
    pthread_t th = -1;
    pthread_mutex_init(&mutex); 初始化互斥锁

    ret = pthread_create(&th, NULL, (void *)func, NULL);
    printf("输入一个字符串, 请回车结束\n");
    while(1)
    {
        pthread_mutex_lock(&mutex); 上锁
        scanf("%s", buf);
        pthread_mutex_unlock(&mutex); /*为了解决输入end无法退出程序问题，我们修改了一下*/
        if(strncmp(buf, "end", 3) == 0)
        {
            printf("程序结束\n");
            break;
        }
    }

    ret = pthread_join(th, NULL); //回收子线程
    if(ret != 0)
    {
        printf("子线程回收失败\n");
        exit(-1);
    }
    printf("子线程回收成功\n");
}
```

```
root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串, 请回车结束
aaa
bbb 出现个没有打
印的问题
```

为什么只有输入没有打印呢, 因为程序启动后就进入了主程序, 主程序上锁, 那么线程的锁就只有阻塞, 因为都是用的同一个锁, 有时候运气好就是线程先执行, 然后你看到 printf 刷屏, 输入字符没法显示, 因为线程把锁占用了, 主进程无法得到锁,

互斥锁就是为了防止几个进程同时读取一个变量，造成数据错误。而给该变量上锁，来进行保护。但是几个进程也必须同时要有这个变量的锁，才能按着规矩来访问该变量。我 linux 驱动里面上互斥锁的方式貌似方法有点浪费。

上面问题最好的解决方法是线程和主进程都加 sleep 把时间让出来。

```
void func(int *arg)
{
    int i;
    while(1)
    {
        sleep(1);
        pthread_mutex_lock(&mutex);
        printf("本次输入 %d 个字符\n", i = strlen(buf));
        memset(buf, 0, sizeof(buf));
        pthread_mutex_unlock(&mutex);
    }
}

void main()
{
    int ret;
    pthread_t th = -1;
    pthread_mutex_init(&mutex, NULL);

    ret = pthread_create(&th, NULL, (void *)func, NULL);
    printf("输入一个字符串，请回车结束\n");
    while(1)
    {
        pthread_mutex_lock(&mutex);
        scanf("%s", buf);
        pthread_mutex_unlock(&mutex);
        /*为了解决输入end无法退出程序问题，我们修改了一下*/
        if(strncmp(buf, "end", 3) == 0)
        {
            printf("程序结束\n");
            break;
        }
        sleep(1);
    }

    ret = pthread_join(th, NULL); //回收子线程
    if(ret != 0)
```

root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串，请回车结束

aaa
本次输入 3 个字符
sssss
本次输入 5 个字符
assaaaa
本次输入 7 个字符
fwf
本次输入 3 个字符
asdad
本次输入 5 个字符
sdfASF
本次输入 6 个字符
asdgfsdf
本次输入 0 个字符

本次输入 8 个字符

这样问题就解决了，
但是还是有点 BUG，
毕竟互斥锁不是用在
锁线程上，而是用来
保护关键代码

Linux 条件变量解决信号量，互斥锁处理线程同步问题

```
char buf[200];
unsigned int flag=0;
pthread_mutex_t mutex;
pthread_cond_t cond;//定义条件变量

void func(int *arg)
{
    int i;
    while(flag==0)
    {
        pthread_cond_wait(&cond,&mutex);
        printf("本次输入 %d 个字符\n",i=strlen(buf));
        memset(buf,0,sizeof(buf));
    }
}

void main()
{
    int ret;
    pthread_t th = -1;
    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&cond,NULL);
    ret(pthread_create(&th,NULL,(void *)func,NULL));
    printf("输入一个字符串，请回车结束\n");
    while(1)
    {
        scanf("%s",buf);
        pthread_cond_signal(&cond);

        if(strncmp(buf,"end",3)==0)
        {
            printf("程序结束\n");
            flag=1;
            break;
        }
    }
    ret=pthread_join(th,NULL); //回收子线程
    if(ret!=0)
    {
        printf("子线程回收失败\n");
        exit(-1);
    }
    printf("子线程回收成功\n");

    pthread_mutex_destroy(&mutex); //销毁互斥锁
    pthread_cond_destroy(&cond); //销毁条件变量
}
```

条件变量要求定义一个互斥锁，和一个条件变量

线程收到条件满足信号放弃阻塞

按道理在条件变量上下位置要加互斥锁，但是我发现互斥锁有问题，所以没有加

初始化互斥锁和条件变量

发送条件满足信号给线程

```

root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串, 请回车结束
aaaa
本次输入 4 个字符
bbbbbbbb
本次输入 9 个字符
end
程序结束
本次输入 3 个字符
子线程回收成功
root@ubuntu:/home/xiang/app/thread#

```

感觉条件变量比信号量和互斥
锁在做线程同步方面好用

条件变量除了满足条件唤醒一个线程阻塞外, 其实还可以同时唤醒多个线程, 我们下面来试试

```

char buf[200];
unsigned int flag=0;
pthread_mutex_t mutex;
pthread_cond_t cond;//定义条件变量

void func3(int *arg)
{
    while(flag==0)
    {
        pthread_cond_wait(&cond,&mutex);
        printf("线程输出3\n");
    }
}

void func2(int *arg)
{
    while(flag==0)
    {
        pthread_cond_wait(&cond,&mutex);
        printf("线程输出2\n");
    }
}

void func1(int *arg)
{
    int i;
    while(flag==0)
    {
        pthread_cond_wait(&cond,&mutex);
        printf("线程输出1... 本次输入 %d 个字符\n",i=strlen(buf));
    }
}

void main()
{
    int ret;
    pthread_t th1 = -1;
    pthread_t th2 = -1;
    pthread_t th3 = -1;
    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&cond,NULL);
    ret(pthread_create(&th1,NULL,(void *)func1,NULL));
    ret(pthread_create(&th2,NULL,(void *)func2,NULL));
    ret(pthread_create(&th3,NULL,(void *)func3,NULL));

    printf("输入一个字符串, 请回车结束\n");
    while(1)
    {
        scanf("%s",buf);
        pthread_cond_signal(&cond);
        if(strcmp(buf,"end",3)==0)
        {
            printf("程序结束\n");
            flag=1;
            break;
        }
    }
    ret(pthread_join(th1,NULL));//回收子线程
    ret(pthread_join(th2,NULL));//回收子线程
    ret(pthread_join(th3,NULL));//回收子线程
}

```

Signal 只能随机让某个线程阻
塞唤醒, 不能同时唤醒

```

root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串, 请回车结束
aaaa
线程输出1... 本次输入 4 个字符
bbb
线程输出3
ccc
线程输出2
sss
线程输出1... 本次输入 4 个字符
end
程序结束
线程输出3

```

你看输出顺序很乱

我们换一个唤醒函数试试

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

void main()
{
    int ret;
    pthread_t th1 = -1;
    pthread_t th2 = -1;
    pthread_t th3 = -1;
    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&cond,NULL);
    ret(pthread_create(&th1,NULL,(void *)func1,NULL));
    ret(pthread_create(&th2,NULL,(void *)func2,NULL));
    ret(pthread_create(&th3,NULL,(void *)func3,NULL));

    printf("输入一个字符串, 请回车结束\n");
    while(1)
    {
        scanf("%s",buf);
        pthread_cond_broadcast(&cond); //这个是条件变量全部唤醒的函数
        if(strncmp(buf,"end",3)==0)
        {
            printf("程序结束\n");
            flag=1;
            break;
        }
    }
    ret(pthread_join(th1,NULL)); //回收子线程
    ret(pthread_join(th2,NULL)); //回收子线程
    ret(pthread_join(th3,NULL)); //回收子线程
    if(ret!=0)
}
```

三个线程同时输出, 所以 pthread_cond_broadcast() 函数是线程只要有这个条件变量都必须输出。

```
root@ubuntu:/home/xiang/app/thread# ./threadtt
输入一个字符串, 请回车结束
aaaa
线程输出1... 本次输入 4 个字符
线程输出3
线程输出2
bbbb
线程输出1... 本次输入 4 个字符
线程输出3
线程输出2
end
程序结束
线程输出1... 本次输入 3 个字符
```

线程结束时，执行线程结束子函数

```
1 #include<pthread.h>
2 #include<stdio.h>
3 #include<stdlib.h>
4
5 void clear_fun(void *arg)
6 {
7     printf("清理函数1执行\n");
8     printf("%s\n", (char *)arg);
9 }
10 }
11 void *trhead_fun(void *arg)
12 {
13     int execute=1;
14     pthread_cleanup_push(clear_fun, "第一个线程清除函数");
15     printf("线程正在运行\n");
16     pthread_cleanup_pop(execute);
17
18 }
19 }
20
21 void main()
22 {
23     int err=-1;
24     pthread_t th1,th2;
25
26     if((err(pthread_create(&th1,NULL,trhead_fun,NULL))!=0)
27     {
28         printf("thread_fun create failed\n");
29     }
30
31     pthread_join(th1,NULL); //该函数是阻塞到这里，等待该函数指定的th1线程执行结束，才继续执行下去
32     printf("thread_fun run.....over\n");
33
34
35
36 }
```

将线程结束函数压入栈
将你想要传入的参数给线程结束函数
1 : 代表线程结束，执行 clear_fun 线程结束函数。0 : 代表不执行

这是执行结果

```
root@ubuntu:/home/xiang/app/thread# ./pop_ptread
线程正在运行
清理函数1执行
第一个线程清除函数
thread_fun run.....over
```

```
2 void *trhead_fun(void *arg)
3 {
4     int execute=0;
5     pthread_cleanup_push(clear_fun, "第一个线程清除函数");
6     printf("线程正在运行\n");
7     pthread_cleanup_pop(execute);
8 }
9
10
11 root@ubuntu:/home/xiang/app/thread# ./pop_ptread
12 线程正在运行
13 thread_fun run.....over
14 root@ubuntu:/home/xiang/app/thread#
```

我输入 0 看看线程结束函数执不执行
线程结束函数没有执行

```
pthread_cleanup_push
pthread_cleanup_pop
```

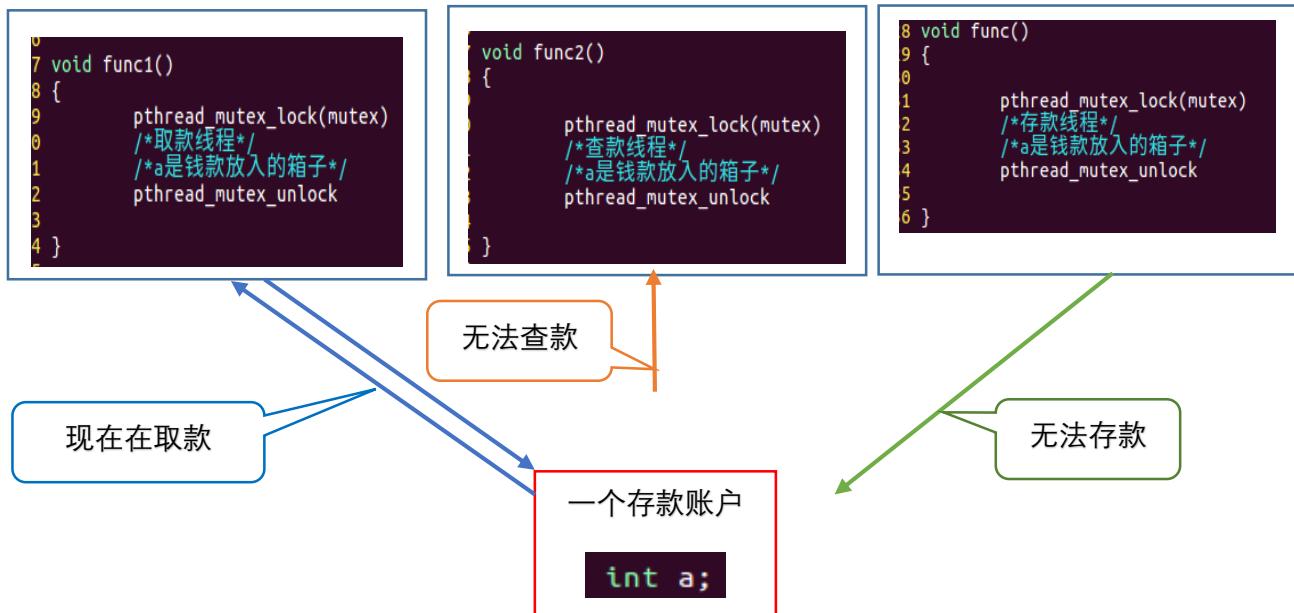
所以这两个函数是成对出现的，`pthread_cleanup_push` 决定将哪个线程结束函数压入栈，因为只有压入栈的函数在该线程结束时才执行，`pthread_cleanup_pop` 是决定线程结束时要不要执行线程结束函数。

```
void clear_fun(void *arg)
{
    printf("清理函数1执行\n");
    printf("%s\n", (char *)arg);
}
```

一般线程结束函数，都是用来释放线程分配的动态内存，和一些线程结束后还没有释放的数据，这种数据就交给线程结束函数来释放。

互斥锁在有些地方使用的一些问题，这些问题得用读写锁来解决

我们以银行账户为例



一个人在执行取款线程取款，那么其余两个执行查款线程和存款线程的人就无法对同一个账户进行操作，因为取款线程执行的时候就上了互斥锁，其他两个线程看见该账户被上锁了，那么就只有休眠，等待取款人把锁还出来，自己才能查款，或者存款。

这个就像公司在给你银行卡打工资，但是你又正好在查询银行卡余额，这时候打工资这个人就看到电脑上处于白板等待状态，等你查询完了，打工资的人才能看到打款成功，这明显不科学。

我们用读写锁来解决这个问题

```
1 #include<pthread.h>
2 #include<stdio.h>
3 #include<stdlib.h>
4
5     pthread_rwlock_t rw;//读写锁变量
6
7 void *inquire(void *arg)
8 {
9     pthread_rwlock_rdlock(&rw);//上读锁
10    printf("查询线程\n");
11    pthread_rwlock_unlock(&rw);//解锁
12 }
13
14 void *withdrawal(void *arg)
15 {
16     pthread_rwlock_rdlock(&rw);//上读锁
17     printf("取款线程\n");
18     pthread_rwlock_unlock(&rw);//解锁
19 }
20
21 void *deposit(void *arg)
22 {
23     pthread_rwlock_rdlock(&rw);//上读锁
24     printf("存款线程\n");
25     pthread_rwlock_unlock(&rw);//解锁
26 }
```

```

void main()
{
    int err=-1;
    pthread_t th1,th2,th3;

    pthread_rwlock_init(&rw,NULL); //创建读写锁

    if((err(pthread_create(&th1,NULL,deposit,NULL))!=0)//创建线程1
    {
        printf("deposit create failed\n");
    }

    if((err(pthread_create(&th2,NULL,withdrawal,NULL))!=0)//创建线程2
    {
        printf("withdrawal create failed\n");
    }

    if((err(pthread_create(&th3,NULL,inquire,NULL))!=0)//创建线程3
    {
        printf("inquire create failed\n");
    }
    while(1);
    printf("主进程执行完成\n");

    pthread_rwlock_destroy(&rw);
}

```

```

root@ubuntu:/home/xiang/app/thread# ./pop_ptread
取款线程
查询线程
存款线程

```

这个就是读写锁执行效果

```

7 void *inquire(void *arg)
8 {
9     pthread_rwlock_rdlock(&rw); //上读锁
0     printf("查询线程\n");
1     pthread_rwlock_unlock(&rw); //解锁
2
3 }

```

```

void *withdrawal(void *arg)
{
    pthread_rwlock_rdlock(&rw); //上读锁
    printf("取款线程\n");
    pthread_rwlock_unlock(&rw); //解锁
}

```

可以直接读取

可以直接读取

共享数据

可以直接读取

读写锁和互斥锁不一样的地方是，任何进程都可以直接读取共享数据，不用等着上锁的进程读取完数据扔出钥匙后，自己才能读取。随时随地都可以自由读取，不用顾忌他人。

所以说读写锁只能用于读共享数据，不能用于写共享数据，因为太随意了，会造成数据写入错误。有一个小问题我在说一下

```
void *withdrawal(void *arg)
{
    pthread_rwlock_rdlock(&rw); //上读锁
    printf("取款线程\n");
    pthread_rwlock_unlock(&rw); //解锁
}

void *deposit(void *arg)
{
    pthread_rwlock_rdlock(&rw); //上读锁
    printf("存款线程\n");
    pthread_rwlock_unlock(&rw); //解锁
}

void main()
{
    int err=-1;
    pthread_t th1,th2,th3;

    pthread_rwlock_init(&rw,NULL); //创建读写锁

    if((err(pthread_create(&th1,NULL,deposit,NULL))!=0)//创建线程1
    {
        printf("deposit create failed\n");
    }

    if((err(pthread_create(&th2,NULL,withdrawal,NULL))!=0)//创建线程2
    {
        printf("withdrawal create failed\n");
    }

    if((err(pthread_create(&th3,NULL,inquire,NULL))!=0)//创建线程3
    {
        printf("inquire create failed\n");
    }

    pthread_rwlock_destroy(&rw);

}
```

```
root@ubuntu:/home/xiang/app/thread# ./pop_ptread
root@ubuntu:/home/xiang/app/thread#
```

为什么没有输出

你看前面读写锁，我是加了死循环，线程才能输出。那是因为主进程比线程执行得快，所以主进程执行完了，线程都还没有开始执行，当然没有输出咯。

```
if((err(pthread_create(&th3,NULL,inquire,
{
    printf("inquire create failed\n")
}
pthread_join(th1,NULL);
pthread_join(th2,NULL);
pthread_join(th3,NULL));
```

```
root@ubuntu:/home/xiang/app/thread# ./pop_ptread
存款线程
取款线程
查询线程
root@ubuntu:/home/xiang/app/thread#
```

我再主程序结束前加上 pthread_join 线程阻塞，也能解决这个问题，那是因为这个函数必须等到它指定的线程执行完，这个阻塞才会消除，不会像进程那样管都不管线程，自己先执行完再说。