## 一、gcc编译

### 1、gcc编译可执行程序4步骤

预处理、编译、汇编、链接（-o指定生成的文件名字）



如果报错信息有行号提示，一般是语法错误，出现在编译阶段

如果报错信息没有行号提示，或者出现链接器collect2 error，则错误出现在链接阶段

### 2、gcc常用参数

-I ：指定头文件目录，如果头文件与源码在一个文件夹里，就不需要指定

-c ：预处理、编译、汇编。**得到二进制文件**。

-g ：包含调试信息，编译时添加调试文件（没有的话无法gdb调试）

-On n = 0~3：编译优化，n越大优化得越多（默认为2）

-Wall ：提示所有警告信息

-D<DEF>：编译时定义宏，注意-D和<DEF>之间无空格

-E ：生成预处理文件

-M ：生成.c文件与头文件依赖关系，以用于Makefile，包括系统库的头文件

-MM ：生成.c文件与头文件依赖关系，以用于Makefile，不包括系统库的头文件

**【注意】：如果是.cpp文件，使用g++来编译，且不带-c参数**

## 二、静态库、动态库

静态库：对时间要求较高、对空间要求较低的核心程序中。如操作系统启动程序。

动态库：对时间要求较低、对空间要求较高。

### 1、静态库制作及使用步骤

1、将.c生成.o文件（静态库的.o文件，不是程序的.o文件）

gcc -c hello.c -o hello.o

2、使用ar工具制作静态库

ar rcs lib库名.a hello.o（库名以lib开头，.a结尾）

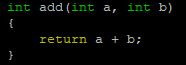
3、编译静态库到 可执行文件中

gcc test.c lib库名.a -o test.out（源码在前，库文件在后）

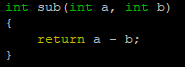
**案例：制作一个包含加法、减法、除法的静态库，并命名为libmyMath.a**

1. 准备add.c，sub.c，div1.c，test.c

#add.c



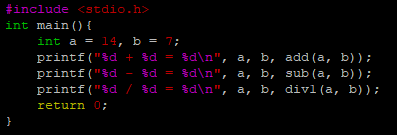
#sub.c



#div1.c



#test.c



1. 将.c生成.o文件

gcc -c add.c -o add.o

gcc -c sub.c -o sub.o

gcc -c div1.c -o div1.o

1. 使用ar工具制作静态库

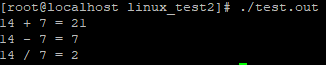
ar rcs libmyMath.a add.o sub.o div1.o

1. 编译静态库到可执行文件

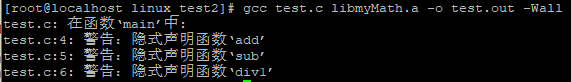
gcc test.c libmyMath.a -o test.out

1. 执行可执行文件

./test.out



**注意：上述的静态库调用方式实际上是存在问题的，执行**

****

实际上test.c文件中是没有对静态库里的3个函数做声明的，但是编译器会做隐式声明，但只会声明返回值为int的隐式声明，刚好与我们定义的函数返回值一致，所以并不会报错，只会是警告。一旦我们使用的函数不是返回值为int的，编译器就会报错，所以我们需要自己声明静态库里的函数。

**操作：**静态库里定义的函数在使用的时候需要在函数里声明，所以需要建立一个.h文件将函数声明放进去，这样每次调用静态库只需要把函数声明的.h文件引用即可。

**案例：优化上述案例**

建立一个myMath.h文件，将函数声明包含进去即可。

### 2、动态库制作及使用步骤

1、将.c生成.o文件（生成与位置无关的代码 -fPIC）（动态库的.o文件，不是程序的.o文件）

gcc -c hello.c -o hello.o -fPIC

2、使用gcc -shared制作动态库

gcc -shared -o lib库名.so hello.o

3、编译可执行文件时指定所使用的动态库 -l：指定库名 -L：指定库路经

gcc test.c -o test.out -l库名 -L库路经

**案例：制作一个包含加法、减法、除法的动态库，并命名为libmyMath.a**

1. 准备add.c，sub.c，div1.c，test.c
2. 将.c生成.o文件
3. 使用gcc -shared制作动态库

gcc – shared -o libmyMath.so add.o sub.o div1.o

1. 编译可执行文件

gcc test.c -o test.out -l myMath -L ./

1. 执行可执行文件

**依照上述步骤执行可执行文件时报错**！！！



**原因：**

**链接器**：工作于链接阶段，工作时需要 -l -L 支持

**动态链接器**：工作于程序运行阶段，工作时需要提供动态库所在目录位置，动态链接器会到固定的地方寻找动态库，如果没有找到就会报错，所以需要将服务于动态链接器的环境变量LD\_LIBRARY\_PATH指定为动态库地址

**解决方式：**

【1】通过环境变量，针对上述案例的具体操作就是在执行可执行文件前：

export LD\_LIBRARY\_PATH=动态库路径

./test.out

【2】永久生效，写入终端配置文件~/.bashrc，建议使用绝对路径

第1种方式只是**临时**的（终端重启就会失效），如果需要永久改动动态库的环境变量需要到终端配置文件~/.bashrc文件里修改export LD\_LIBRARY\_PATH=动态库路径。然后还需要运行一次bashrc使其生效，此时可以执行..bashrc或者source .bashrc，或者重启终端

1. vi ~/.bashrc
2. 写入export LD\_LIBRARY\_PATH=动态库路径 保存
3. ..bashrc或者source .bashrc或者重启终端——>让修改后的 .bashrc生效
4. ./test.out成功！

【3】拷贝到自定义动态库到标准C库所在目录里（不推荐）

sudo cp libmyMath.so /lib

./test.out

## 三、数据段合并

内存布局图（32位操作系统为例）



## 四、gdb调试工具

g++ test.cpp -o a.out -g 或者 g++ test.cpp -g（默认生成a.out）

gdb a.out进入gdb调试页面

### 1、基础命令

list/l n：从第n行开始显示源码，不会显示完全，按l会接着显示后面的

break/b n：在第n行设置断点

run/r：运行，程序停止在断点位置（断点处尚未执行）

next/n：下一个，即进入下一行

step/s：单步（与next的区别在于step遇到函数调用会进入函数内部，而next会执行完函数进入下一行，如果函数是系统函数要用next，避免进入系统函数内部）

util n：跳转到第n行（如果不小心step进入系统函数内部，可以使用该命令出来）

print/p 变量i：查看变量i的值

continue：结束当前断点，继续运行断点后的程序

quit/q：退出当前gdb调试

### 2、其他命令

使用run去查找段错误出现位置

**start：**单步执行，运行程序，停在第一行执行语句

**finish：**结束当前函数，返回到函数调用点

**backtrace/bt：**查看函数的调用的栈帧和层级关系（随着函数调用而在stack上开辟的一片内存空间，用于存放函数调用时产生的局部变量和临时值，函数调用结束时栈帧消失）

**info/i：**查看gdb内部局部变量的数值 info b：查看断点信息表

**frame/f：**切换函数的栈帧

**set args：**设置变量的值 set var n=100 set args aa bb cc

**run argv[1] argv[2]：**调试时命令行传参

**ptype 变量i：**查看变量i的类型

**display 变量i：**跟踪变量i，避免每次都print

**undisplay 变量i的编号：**取消跟踪变量i

**delete/d breakpoint 断点编号：**删除断点

**段错误：非法访问内存**

访问非访问区，比如编号地址为1000的内存区域

对只读区域进行写操作

数组越界

访问不是自己申请的区域

### 3、常见错误

如果使用gdb调试出现报错如下：



或者

**说明**在生成可执行文件a.out时没有加 -g 的调试选项

**解决办法：**

可以退出再次生成可执行文件，也可用file指明另一个有调试选项的可执行文件，比如这段程序还有另一个可执行文件test.out（可以调试），只需file test.out即可，不必退出gdb

## 五、Makefile项目管理

把一系列命令集合放到一个文件中批量处理

### 1、基本规则

命名：makefile Makefile

1. 若想生成目标，检查规则中的依赖条件是否存在，如果不存在，则寻找是否有规则里生成该依赖文件（如果依赖条件不存在，则需要在命令中编写生成依赖条件的代码，make命令自己寻找）
2. 检查规则中的目标是否需要更新，必须先检查它的所有依赖，依赖中有任一个被更新，则目标必须更新

### 2、1个规则：

目标：依赖关系

（一个Tab缩进）命令

Makefile默认只会将makefile文件里的第一个命令当作终极目标，一旦终极目标执行完成，下面的指令并不执行。

**解决办法**是在makefile文件头端加上”ALL: 终极目标”（如ALL:a.out），此时命令摆放顺序就不会影响了。

**案例1：使用makefile编译main.cpp为main**

1. 新建Makefile文件（名称必须为makefile或者Makefile，否则无法使用make命令）



1. 在Makefile文件里依据规则编写makefile命令



1. 使用make指令执行makefile（会生成main可执行文件）





1. 运行可执行文件main

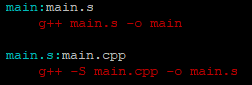


**案例2（模拟规则1）**：**使用makefile编译main.s为main**

1. 新建makefile文件，编写命令



1. 此时，找不到依赖条件main.s，所以需要生成main.s



1. 使用make指令执行makefile





1. 运行可执行文件main

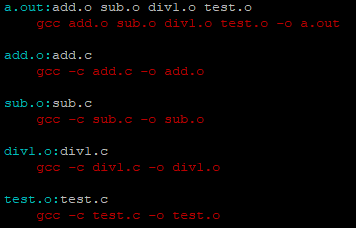


**案例3（模拟规则2）：使用makefile实现联合编译（add.c sub.c div1.c test.c）**

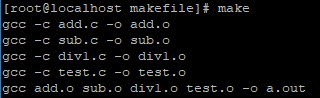
1. 编写makefile文件



这里有一个问题就是，如果我改变某一个函数（比如add.c），那么每次执行make都会将add.c sub.c div1.c test.c都编译，而实际上sub.c div1.c test.c都没有改动，不需要重新编译。因此makefile文件可以这么写



1. 使用make指令执行makefile

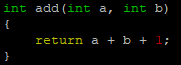


可以看到，共执行了5步

1. 执行a.out可执行文件（成功）



1. 倘若add发生变化，根据规则2，应该只会执行生成add.o和a.out这两步（将add.c改动为原结果+1）

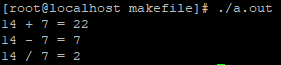


1. 重新执行make指令



如预期一样，只执行了2步

1. 执行a.out可执行文件



加法结果确实加了1

**根据结果可知，这种makefile编写可以节约系统时间消耗，也符合规则2。**

### 3**、2个函数**

src = $(wildcard \*.c)

找到当前目录下所有后缀为.c的文件，赋值给src

obj = $(patsubst %.c,%.o, $(src))

把src变量里的所有后缀为.c的文件替换成.o

clean:

-rm -rf $(obj) a.out rm前的“-”作用是删除不存在文件时，不报错，顺序执行结束

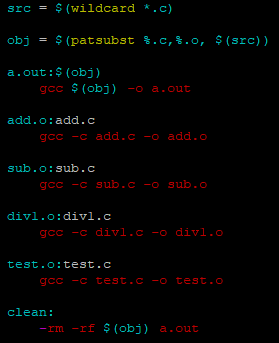
**注意**：可以在make时指定**make clean -n**函数来先执行一次clean函数，删除指定的东西，**-n**的作用时预删除，但还未删除，从而可以提示用户要删除哪些东西。

**单独执行**make里面的某些指令，可以使用”**make 指令名**”，然后可以加上”**-n**”参数来模拟执行make指令。

**比如**想单独运行makefile里的add.o命令，可以使用”make add.o -n”来查看会执行什么命令，如果没有错误就使用” make add.o”命令

**案例：以上述案例3为例，**

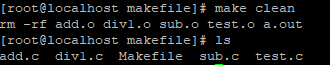
1. 可以将makefile修改为如下：



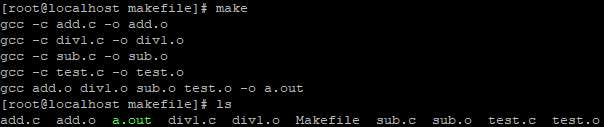
1. 执行前先使用make clean -n来删除以前的\*.o 和a.out文件



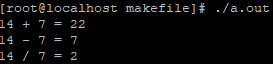
1. 确定只是删除无误后在真正执行删除



1. 然后执行make指令



1. 最后执行a.out



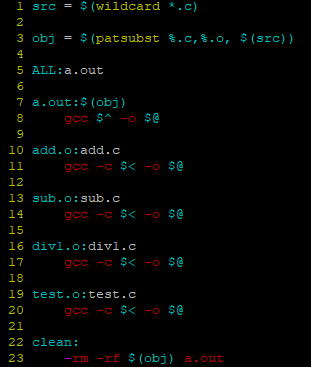
### 4、3个自动变量

**$@**：在规则命令中表示**目标**（只出现在规则的命令中）

**$<**：在规则命令中表示的**第一个依赖条件**

**$^**：在规则命令中表示的**所有的依赖条件**，如果将该变量应用在模式规则中，它可将依赖条件列表中的依赖依次取出，套用模式规则

上述的案例的makefile文件可以修改为；



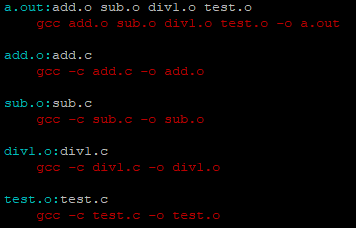
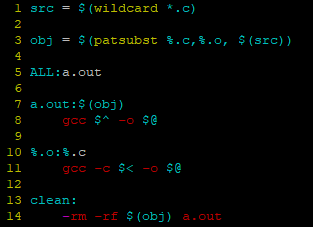
### 5、模式规则

上述makefile文件里add函数、sub函数，div1函数，test函数的模式都是一样的，都是

%.o:%.c

gcc -c $< -o $@

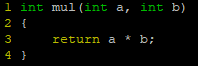
**操作**：可以合并这三个函数的描述，将上述代码加入makefile中，替换原来的4个函数



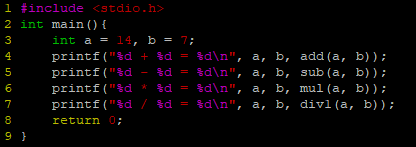
至此，最初始版本的makefile与现在的makefile已大有不同，现在这种是**可扩展的**。

**案例（可扩展性）：比如还想添加一个乘法函数，如果按照原来的makefile的写法，就不仅需要构造mul.c文件，还要更改makefile文件。而现在这种就不需要改动makefile文件了。**

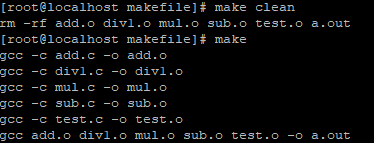
1. 编写mul.c函数



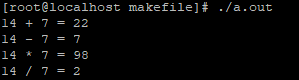
1. 更改test.c文件，添加乘法输出



1. 重新make



1. 运行a.out



**成功！！！**

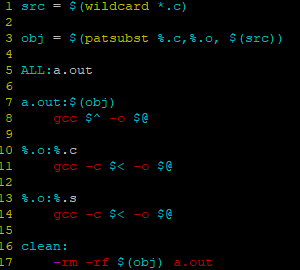
### 6、静态模式规则

指定模式规则给谁用

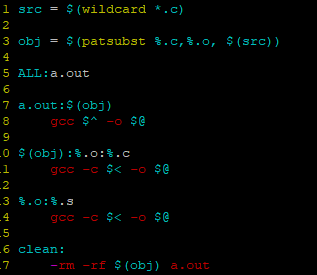
$(obj):%.o:%.c

gcc -c $< -o $@

例如



第7行找模式规则%.o文件，但是生成.o文件的规则有两个，不知道要用哪一个。此时可以将要使用的规则改为静态模式规则，指明该规则给谁用。



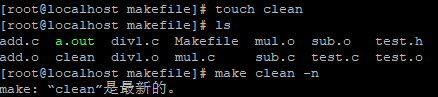
上述表明第10行的规则指定给生成a.out的规则（第7行）使用。

### 7、伪目标

如果当前目录中包含有一个clean文件或者ALL文件，执行make就不会成功，此时需要在makefile中指明伪目标，操作就是在makefile行尾加上.PHONY:clean ALL

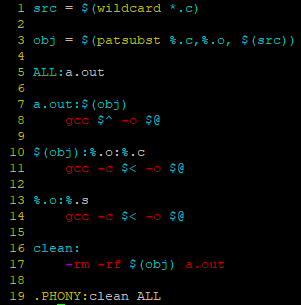
**案例：**

1. 创建clean文件



可以看到，make clean失败

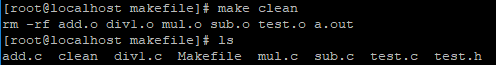
1. 更改makefile文件



1. 执行make clean -n成功



1. 执行make clean



**成功！！！**

**注意：**makefile里面还可以定义变量，就像shell脚本一样

### 8、2个参数

-n：模拟执行make、make clean命令

-f：指定文件执行Make命令(有些makefile文件并不叫makefile)

## 六、文件IO

### 1、open/close函数

#### 函数原型：





**头文件：<unistd.h>**

#### 参数说明：

flags：指明打开文件的权限，包括O\_RDONLY（只读）、O\_WRONLY（只写）、O\_RDWR（读写）、O\_NOCTTY、O\_TRUNC（把文件清零）、O\_CREAT（创建文件）、O\_EXCL（判断文件是否存在）、O\_APPEND（追加）、O\_NONBLOCK（非阻塞）

mode：当创建文件时，mode以数字形式来表明文件权限

fd：文件描述符，一个整数

**注意：**创建文件时，指定文件访问权限。权限同时受umask影响。结论是：

**文件权限 = mode & ~umask**

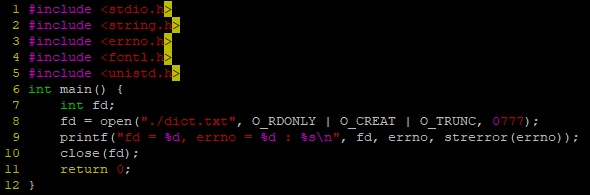
#### 返回值

成功：文件描述符

失败：-1，设置errno

**案例：打开dict.txt文件，如果存在，则把文件清零；如果不存在，则创建该文件，并设置权限为0777。**

1. 编写test.c文件，使用open函数



上述程序中，

<unistd.h>是open函数头文件，

<fcntl.h>是O\_RDONLY的头文件，

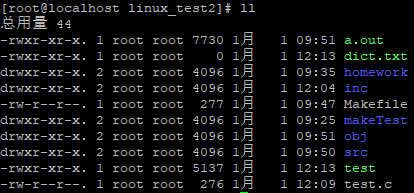
<errno.h>是errno头文件，

<string.h>是strerror()函数头文件

1. gcc编译test.c文件，并执行可执行文件



1. 查看是否生成dict.txt文件



1. 观察可知，dict.txt文件权限并不是777，而是755，查看umask



mode=0777，即0111 0111 0111，umask=0022，即0000 0010 0010，umask按位取反得~umask = 1111 1101 1101，将mode和~umask按位与得0111 0101 0101，即**755**，刚好对应上dict.txt文件权限为755，**文件权限 = mode & ~umask。**

#### 常见错误

1. 打开文件不存在





可以看到open函数返回-1，表示打开文件失败，error number为2，错误描述是不存在该文件。

1. 以写方式打开只读文件（打开文件没有对应权限）





在我自己的实验中打开是成功的，但是课程上面讲的是会报errno=13:Permission denied

1. 以只写方式打开目录





可以看到open函数返回-1，表示打开文件失败，error number为21，错误描述是这是个文件目录。

#### 错误处理函数：

与errno有关

1、printf(“xxx error: %d\n”, errno);

2、char \*strerror(int errnum);

printf(“xxx error: %s\n”, strerror(errno));

3、void perror(const char\* s);

perror(“xxx error”);

### 2、read函数

#### 函数原型：



将大小为count字节的内容，从文件fd读入缓冲区buf，失败时返回-1

#### 参数说明

fd：文件描述符

buf：存数据的缓冲区

count：缓冲区大小

#### 返回值

0：读到文件末尾

成功：读到的字节数

失败：-1，设置errno

如果返回-1，且errno = EAGAIN或EWOULDBLOCK，说明不是read失败，而是在以非阻塞方式读一个设备文件或网络文件，且文件无数据

errno = EAGAIN or EWOULDBLOCK：设置了非阻塞方式读。没有数据到达

errno = EINTR：慢速系统调用被中断

errno = ECONNRESET：说明连接被重置，需要close，移除监听队列

errno = “其他情况”异常

### 3、write函数

#### 函数原型：



将缓冲区buf的count字节的内容，写入进fd所描述的文件里，失败时返回-1

#### 参数说明

fd：文件描述符

buf：待写出数据的缓冲区

count：写出数据的大小

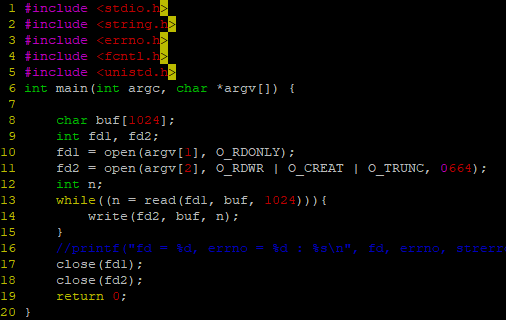
#### 返回值

成功：写入的字节数

失败：-1，设置errno

**案例：将test.c的文件通过read/write的方式写入到另一个文件里**

1. 编写mycopy.c文件



1. 编译mycopy.c文件（make mycopy会默认只把mycopy.c进行make编译）

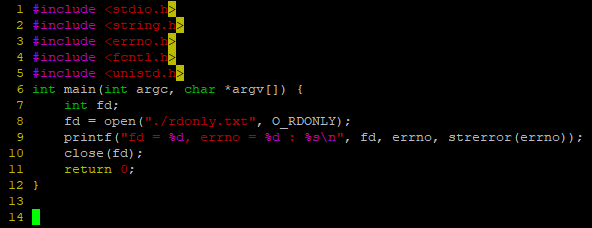




1. 执行mycopy可执行程序



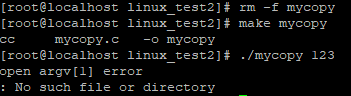
1. 查看生成的test2.c，与test一样



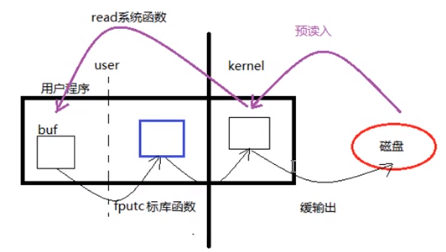
1. 添加出错模块perror（头文件<pthread.h>）



1. 执行mycopy，并传入一个错误的文件



### 4、预读入缓输出



上述案例是使用read/write来实现文件复制，read/write函数直接实现user区和kernel区之间的数据传送，相比于传统的fgetc/fputc函数（调用read/write函数），是否read/write函数的实现更快一些呢？

为此，做个实验，分别使用read/write函数和fgetc/fputc函数来复制一个4.7M大小的txt文件，并使用strace命令来追踪两者对系统资源的调用。fgetc/fputc函数每次按1字节读取写入，所以将read/write测试文件里的buff的size设为1。结果表明read/write反倒更慢一些。

原因就在于，read/write版本，每次user区的buf有一个字节就会往kernel区的buf传送数据，而fgetc/fputc版本会在user区内另建一个buf区（图中蓝色方框），每次都将一个字节存入该buf，只有当该buf满了4096字节，才会往kernel区传送。

因为跨越user区和kernel区的数据传送要比user区内部数据传送耗时多得多，这就导致read/write版本要跨越4.7M次user区和kernel区，而fgetc/fputc版本只需要跨越1204次即可，因而时间反倒更短。

### 5、文件描述符



#### PCB进程控制块

用于描述进程的（比如a.out），本质是一个结构体，有很多成员，文件描述符表就是其一（存在一个指针，该指针指向文件描述符表）。

#### 文件描述符表

成员是文件描述符，分别为0/1/2/3/4/5/6/……/1023（分配给用户的从3开始），每次打开一个文件获得的文件描述符就是表中可用的最小的那个。

0 — STDIN\_FILENO 标准输入

1 — STDOUT\_FILENO 标准输出

2 — STDERR\_FILENO 标准出错

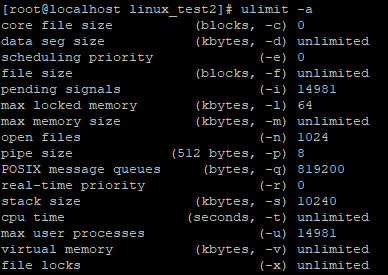
（头文件<unistd.h>）

对于这3个系统自己的文件描述符，不推荐直接使用0/1/2来表示，推荐使用它们的宏

#### 最大打开文件数

一个进程最多打开1024个文件，最大文件描述符值为1023。

命令查看ulimit -a 查看open files对应值。默认为1024



可以使用ulimit -n 4096修改

当然也可以通过修改系统配置文件永久修改该值，但不建议这样的操作

#### FILE结构体

主要包含文件描述符、文件读写位置、IO缓冲区三部分内容。

struct file {

…

文件的偏移量；

文件的访问权限；

文件的打开标志；

文件内核缓冲区的首地址；

struct operation \* f op;

};

查看方法：

1. /usr/src/kernel/linux-header-3.16.0-30/include/linux/fs.h
2. lxr：百度lxr → lxr.oss.org.cn → 选择内核版本 → 点击File Search进行搜索

→ 关键字：”include/linux/fs.h” → Ctrl+F查找“struct file {“

→ 得到文件内核中结构体定义

→ “struct file operations“文件内容操作函数指针

→ “struct inode operations“文件属性操作函数指针

### 6、阻塞和非阻塞

读常规文件是不会阻塞的，不管读多少字节，read一定会在有限的时间内返回。从终端设备或网络读则不一定，如果从终端输入的数据没有换行符，调用read读终端设备就会阻塞，如果网络上没有接收到数据包，调用read从网络读就会阻塞，至于阻塞时间的长短则是不确定的，如果一直没有数据到达就一直阻塞在那里。同样，写常规文件是不会阻塞的，而向终端设备或网络写则不一定。

阻塞（Block）：当进程调用一个阻塞的系统函数时，该进程被置于睡眠（Sleep）状态，这时，内核调度其他进程运行，直到该进程等待的事件发生了（比如网络上接收到数据包，或者调用Sleep指定的睡眠时间到了）它才有可能继续运行。与睡眠状态相对的时运行状态（Running），在Linux内核中，处于运行状态的进程分为两种情况：

**正在被调度执行**。CPU处于该进程的上下文环境中，程序计数器（eip）里保存着该进程的指令地址，通常寄存器里保存着该进程运算过程的中间结果，正在执行该进程的指令，正在读写该进程的地址空间。

**就绪状态**。该进程不需要等待什么事件发生，随时都可以执行，但CPU暂时还在执行另一个进程，所以该进程在一个就绪队列中等待被内核调度。系统中可能同时有多个就绪的进程，那么该调度谁执行呢？内核的调度算法是基于优先级和时间片的，而且会根据每个进程的运行情况动态调整它的优先级和时间片，让每个进程都能比较公平地得到机会执行，同时兼顾用户体验，不能让用户交互的进程响应太慢。

#### **产生阻塞的场景：**

读设备文件、读网络文件。（读常规文件无阻塞）

/dev/tty——终端文件

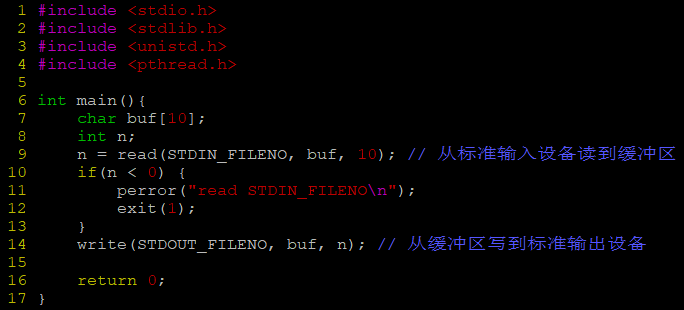
**案例：**

**阻塞读终端：** 【block\_readtty.c】

**非阻塞读终端：**  【nonblock\_readtty.c】

**非阻塞读终端和等待超时：** 【nonblock\_timeout.c】

1. **block\_readtty.c**



编译运行该程序，文件会处于阻塞状态一直等待用户终端输入



用户输入后会同等输出在终端上

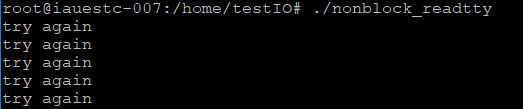


通过open函数指定文件打开方式可以以非阻塞状态读终端

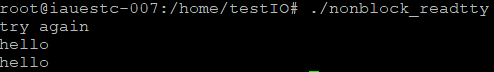
1. **nonblock\_readtty.c**



编译运行该程序，文件会处于非阻塞状态，每2s提示用户一次，一直等待用户终端输入

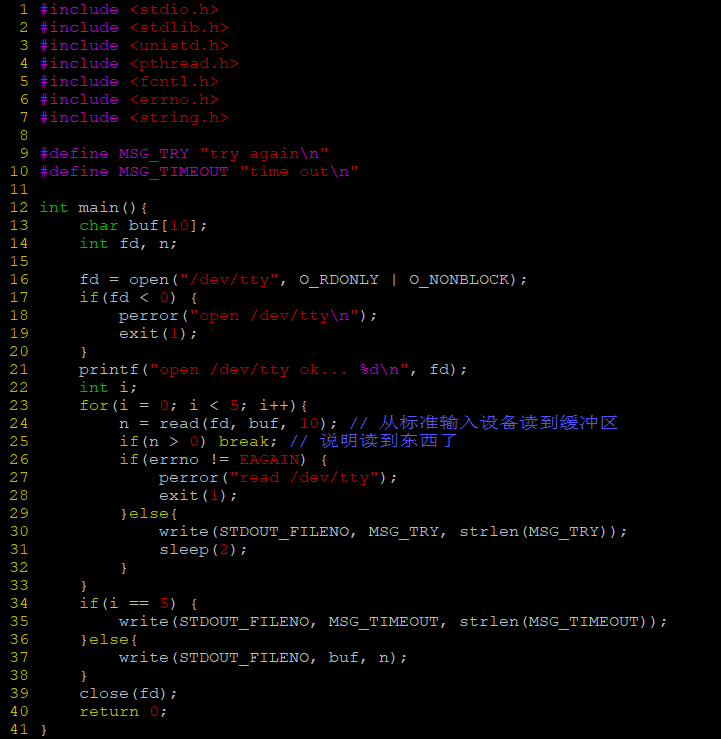


用户输入后会同等输出在终端上



因为非阻塞会一直占用终端，可以加一个timeout

**3、nonblock\_timeout.c**



超过10s终端一直没有输入的话就会结束



**注意：阻塞与非阻塞是对于文件而言的。而不是read、write等的属性。阻塞是设备文件、网络文件的属性。read终端，默认为阻塞。**

### 7、fcntl函数

改变一个【已经打开】的文件的访问控制属性（头文件<fcnl.h>）

重点掌握两个参数的使用，F\_GETFL和F\_SETFL

#### 函数原型：



#### 参数说明

fd：文件描述符

cmd：命令

…：根据命令决定后续参数

#### 返回值

flags：文件状态，int值

int flags = fcnl(fd, F\_GETFL);

获取文件状态：F\_GETFL

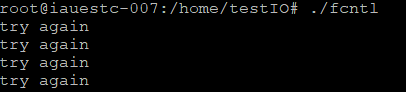
设置文件状态：F\_SETFL

**案例：通过flags在文件打开的情形下修改文件状态，将终端由默认的阻塞状态变为非阻塞状态**



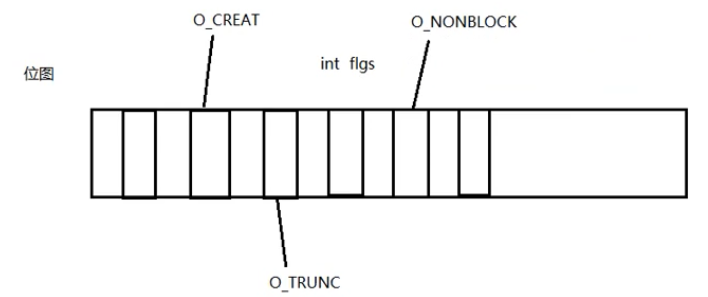
案例中，先获取标准输入设备（STDIN\_FILENO）的状态，默认为阻塞，那么就应该是等待用户输入。然后19行修改flags，并将STDIN\_FILENO的状态设置为flags，如果设置成功，那么就应该是每3s提示用户try again。

运行程序



表明终端状态修改成功。

**背后的原理就是位图。**



1. 获取文件的位图



1. 将O\_NONBLOCK位设置为1



1. 将新的位图设置给STDIN\_FILENO



### 8、lseek函数

#### 文件偏移：

Linux中可使用系统函数lseek来修改文件偏移量（读写位置）。

每个打开的文件都会记录着当前读写位置，打开文件时读写位置是0，表示文件开头，通常读写多少个字节就会将读写位置往后偏移多少个字节。但是有一个例外，如果以O\_APPEND方式打开，每次写操作都会在文件末尾追加数据，然后将读写位置移到新的末尾。lseek和标准I/O库的fseek函数类似，可以移动当前读写位置（偏移量）。

**【注意】：**文件“读”和“写”使用同一偏移量

#### 函数原型：



#### 参数说明：

fd：文件描述符

offset：偏移量（矢量）

whence：偏移起始位置

whence取值：

SEEK\_SET：起始位置

SEEK\_CUR：当前位置

SEEK\_END：末尾位置

#### 返回值：

【返回】起始位置开始的向后偏移量，错误返回-1并设置errno

#### lseek常用应用：

1. 使用lseek拓展文件：IO（write）操作才能实质性地拓展文件。单lseek是不能拓展文件的

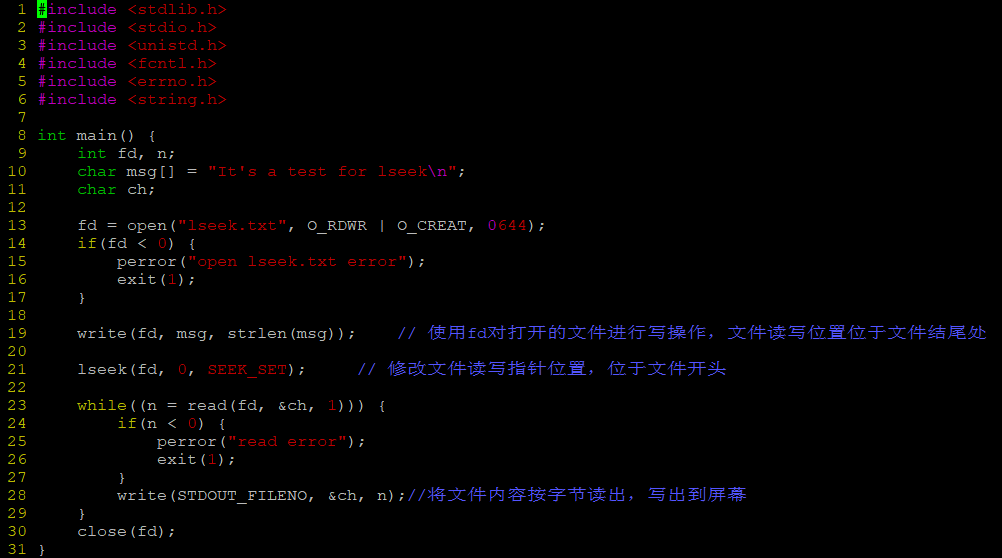
一般：write(fd, “a”, 1);

od –tcx filename 查看文件的16进制表示形式

od –tcd filename 查看文件的10进制表示形式

1. 通过lseek获取文件的大小：lseek(fd, 0, SEEK\_END);

**案例1：将一个字符串写进lseek.txt文件里，并读出到终端**



运行程序后终端输出字符串，且txt文件被建立并写入字符串

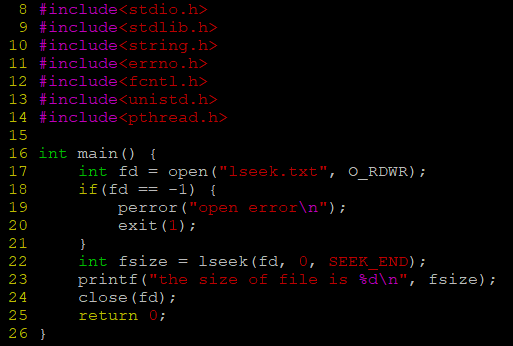


若将lseek注释掉，则txt文件依然被写入成功，但终端不会输出字符串

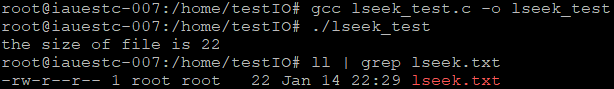


【原因】：因为文件的“读”和“写”使用同一偏移量，在对txt进行写操作后，读写位置均位于文件末尾，此时再进行读操作就会从末尾开始，并没有数据，所以终端不显示数据。而加上lseek后，就将读写位置重新调整到文件开头处，此时终端输出正常。

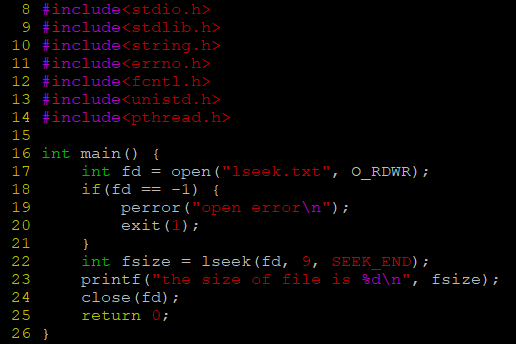
**案例2：使用lseek获取lseek.txt文件的大小（对应常用应用2）**



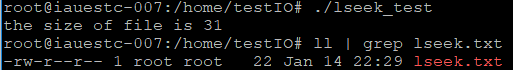
执行程序并查看文件大小，lseek获取大小与文件本身大小一致，获取成功



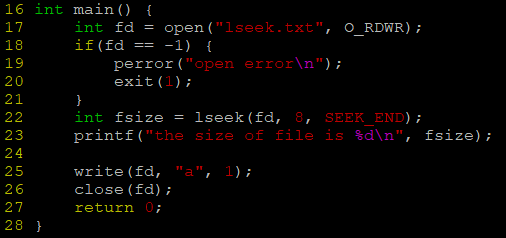
**案例3：拓展文件大小（对应常用应用1）**

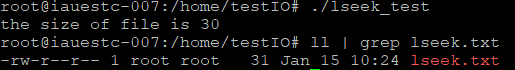


将lseek初始偏移量设置为9，那么输出文件大小应该为31



但是文件真实大小依然为22，若要实质性拓展文件，需要结合write使用



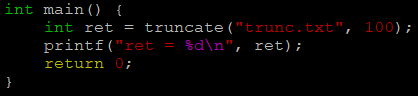


此时文件大小被真正拓展为31了。查看lseek.txt文件

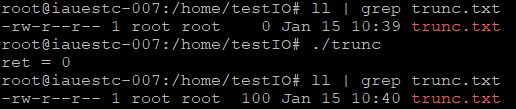


里面的”^@”就是字符”\0”

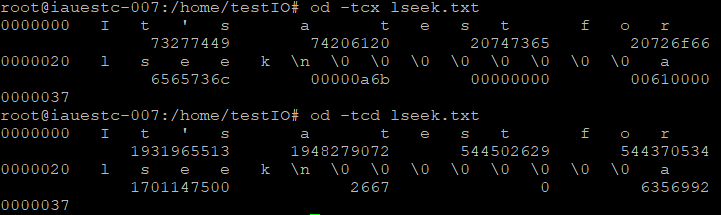
**拓展：**使用**truncate**函数也能拓展文件大小，且不需要打开文件



将trunc.txt大小拓展为100



**案例4：就是以16进制和10进制查看文件**



### 9、传入传出参数

#### 传入参数：

1. 指针作为函数参数
2. 通常由const关键字修饰
3. 指针指向有效区域，在函数内部做读操作

比如open函数里的pathname参数：



#### 传出参数：

1. 指针作为函数参数
2. 在函数调用之前，指针指向的空间可以无意义，但必须有效
3. 在函数内部，做写操作
4. 函数调用结束后，**充当函数返回值**

比如strcpy里的dest参数：



#### 传入传出参数：

1. 指针作为函数参数
2. 在函数调用之前，指针指向的空间有实际意义
3. 在函数内部，先做读操作，后做写操作
4. 函数调用结束后，**充当函数返回值**

比如strtok\_r函数的saveptr参数：



### 10、文件系统

#### inode：

其本质是结构体，存储文件的属性信息。如：权限、类型、大小、时间、用户、盘块位置。。。也叫做文件属性管理结构，大多数的inode都存储在磁盘上

少量常用、近期使用的inode会被缓存到内存中。

创建硬链接，inode是一样的，但会使inode计数加1。

#### dentry：

**目录项**，其本质依然是结构体，重要成员白能量有两个{文件名，inode号, …}，而文件内容(data)保存在磁盘盘块中。通过inode号能找到inode，通过inode能找到磁盘存储位置。

**ln命令实际上就是创建了新的目录项。**

#### 文件系统

文件系统是，一组规则，规定对文件的存储及读取的一般方法。文件系统在磁盘格式化过程中指定。

常见的文件系统有：fat32、ntfs、exfat、ext2、ext3、ext4

### 11、文件操作：

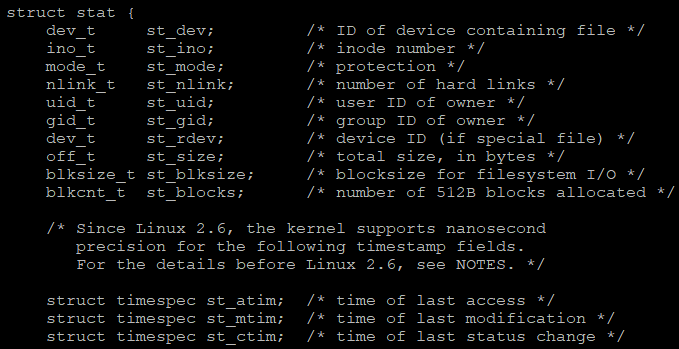
#### stat函数

获取文件属性，（**从inode结构体中获取**）主要使用是获取大小和类型

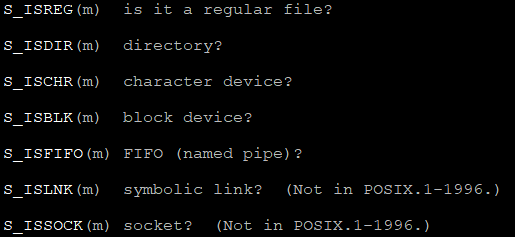


成功返回0，失败返回-1

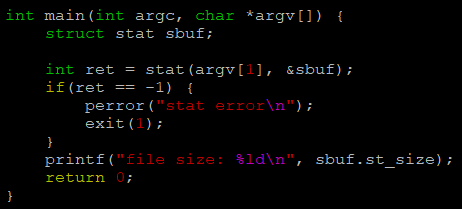
stat 结构体成员：



**注意：**这里的**st\_mode**是表示文件类型，当使用st\_mode时可以使用以下几个宏函数：



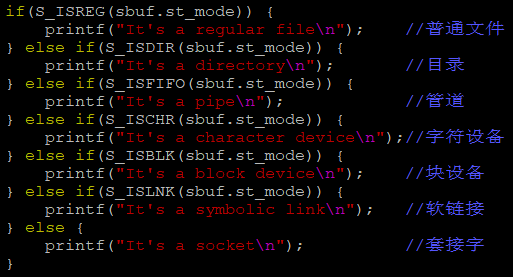
**案例1：获取指定文件的大小**



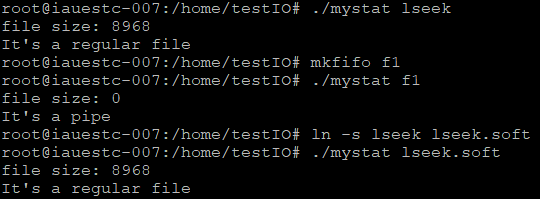
运行程序，获取trunc.c的大小



**案例2：获取指定文件的类型**



运行程序，获取指定文件的类型



发现一个问题，lseek.soft是一个软链接文件，但是打印出来却显示是普通文件（打印的实际上是软链接对应的本尊的类型），这被称为**stat穿透**。

如果不想穿透，使用**lstat函数**，其与stat函数**唯一区别**就在于不能穿透。

#### 特殊权限位

包含三个二进制位。依次是：设置组ID位setGID；设置用户ID位setID；黏住位sticky

##### 黏住位

早期计算机内存紧张，只有精要的常用的程序可以常驻物理内存，剩下的要暂存磁盘中。当内存不够用的时候会将该部分程序存回磁盘，腾出内存空间。若文件设置了黏住位，那么即使在内存条比较吃紧的情况下，也不会将该文件回存到磁盘中。由于现阶段操作系统的虚拟内存管理分页算法完善。该功能已经被弃用。

但我们仍然可以对目录设置黏住位。被设置了该位的目录，其内部文件只有：

1. 超级管理员
2. 该目录所有者
3. 该文件的所有者

以上三种用户有权限做删除、修改操作。其他用户可以读、创建，但不可以随意删除。

##### setID位

进程有两个ID：EID（有效用户ID），表示进程履行哪个用户的权限。

UID（实际用户ID），表示进程实际属于哪个用户。

多数情况下，EID和UID相同。但是，当文件的setID被设置后两个ID则有可能不一样。

例如：当进程执行一个root用户的文件，若该文件的setID位被设置为1，那么执行该文件时，进程的UID不变。EID变为root，表示进程开始履行root用户权限。

#### access函数

测试指定文件是否存在/拥有某种权限



成功，具备该权限：0；失败，不具备该权限：-1，设置errno为相应值

参数2：R\_OK、W\_OK、X\_OK

通常使用access函数来测试某个文件是否存在。F\_OK

#### chmod函数

修改文件的访问权限



成功：0；失败：-1，设置errno为相应值

#### truncate函数

截断文件长度成指定长度。常用来拓展文件大小，代替lseek



成功：0；失败：-1，设置errno为相应值

#### link函数

**实现ln命令的功能**

思考，为什么目录项要游离于inode之外，画蛇添足般地将文件名单独存储呢？

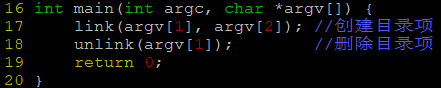
其目的是为了实现文件共享。Linux允许多个目录项共享一个inode，即共享盘块（data）。不同文件名，在人类眼中将它理解为两个文件，但是在内核眼里是同一个文件。

link函数，可以为已经存在的文件创建目录项（**硬链接**）。

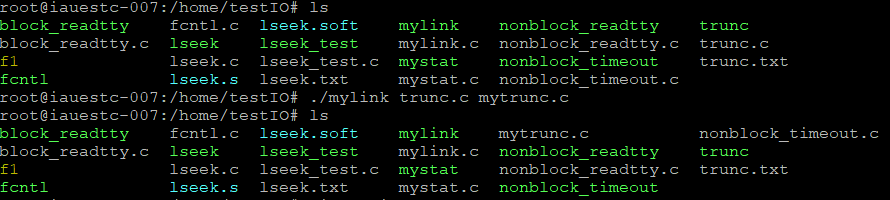


注意：由于两个参数可以使用“相对/绝对路径+文件名”地方式来指定，所以易出错。如：link(“../inc/a.c”, “../dev/b.c”)若a.c，b.c都对，但inc和dev目录不存在也会失败。

**案例：使用link函数实现mv功能**



执行程序，实现mv trunc.c mytrunc.c



#### unlink函数

删除一个文件的目录项：



成功：返回0；失败：返回-1，设置errno为相应值

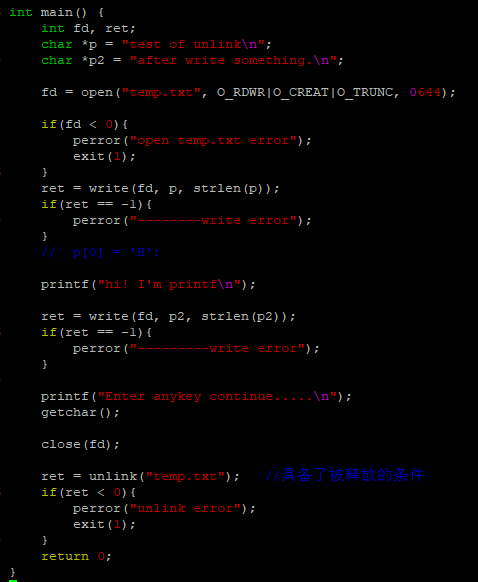
注意：Linux下删除文件的机制：不断将st nlink 减1，直至减到0为止。无目录项对应的文件，将会被操作系统择机释放。（具体时间由系统内部调度算法决定）

因此，我们删除文件，从某种意义上说，**只是让文件具备了被释放的条件。**

unlink函数的特征：清除文件时，如果文件的硬链接数到0了，没有dentry对应，但该文件仍然不能被立马释放。要**等到所有打开该文件的进程关闭该文件**，系统才会挑时间将该文件释放掉。

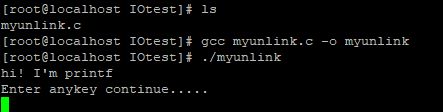
**案例1：使用unlink来删除文件**

1. 编写myunlink.c文件

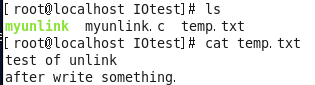


该程序功能是创建一个临时文件temp.txt，然后将两个字符串p和p2写进文件中，然后用户在终端输入一个字符后就会删除temp.txt文件。

1. 执行该程序

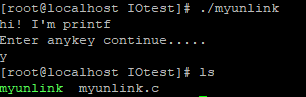


1. 查看是否产生temp.txt文件



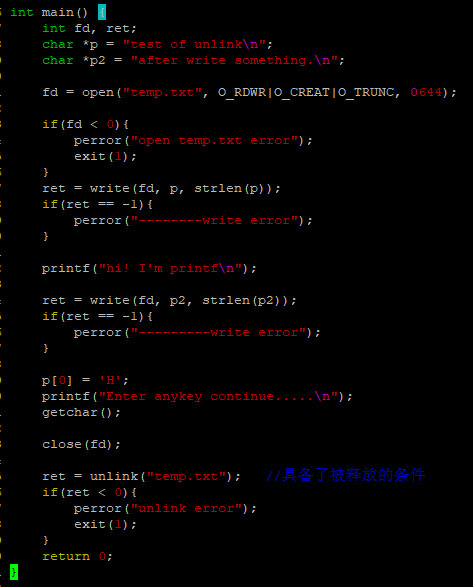
可以看出，此时temp.txt是存在的

1. 输入字符，执行unlink函数

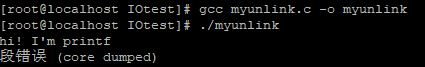


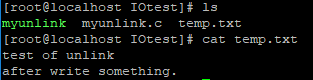
可以看出，temp.txt文件已经被删除。

**案例2：在上述程序中添加段错误**



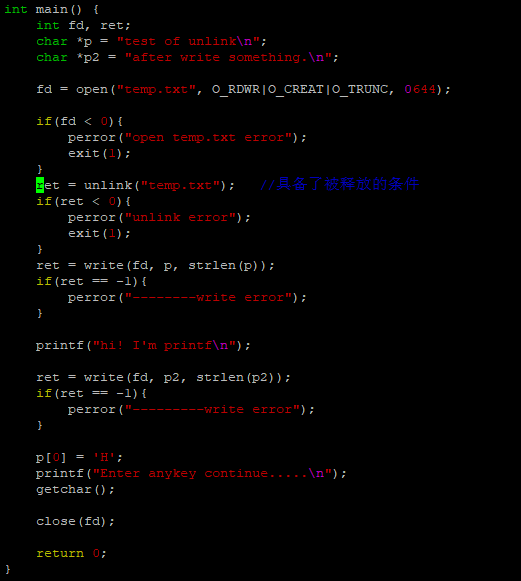
在close前添加一个段错误



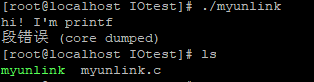


这个时候，虽然发生了段错误，但是temp.txt文件已经创建了，但并没有执行unlink进行删除，**这是我们不希望的**，我们希望的是即使发生了段错误，也应该删除临时文件。

此时可以将unlink函数的使用放到open后面，此时文件并没有被释放，等到文件使用完毕才会删除。



执行程序，虽然报错，但是临时文件是删除的



对应了上述的unlink的**性质**，清除文件时，如果文件的硬链接数到0了，没有dentry对应，但该文件仍然不能被立马释放。要**等到所有打开该文件的进程关闭该文件**，系统才会挑时间将该文件释放掉。

#### 隐式回收

当进程结束运行时，所有该进程打开的文件会被关闭，申请的内存空间会被释放。系统的这一特性称之为隐式回收系统资源。类似于程序里使用malloc但是没有使用free，程序结束后系统会自动free（前提是程序得结束，系统才会自动回收）。

#### symlink函数

创建一个符号链接



成功：0；失败：-1，设置errno为相应值

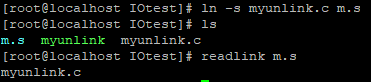
#### readlink函数

读取符号链接文件本身内容，得到链接所指向的文件名



成功：返回实际读到的字节数；失败：-1设置errno为相应值

将读到的信息放到大小为bufsize的buf缓冲区



#### rename函数

重命名一个文件



成功：0；失败：-1，设置errno为相应值

类似于mv的功能

### 12、目录操作

工作目录：“./”代表当前目录，指的是进程当前的工作目录，默认是进程所执行的程序所在的目录位置。

#### getcwd函数

**获取**进程当前工作目录



成功：buf中保存当前进程工作目录位置。失败返回NULL。

#### chdir函数

**改变**当前进程的工作目录



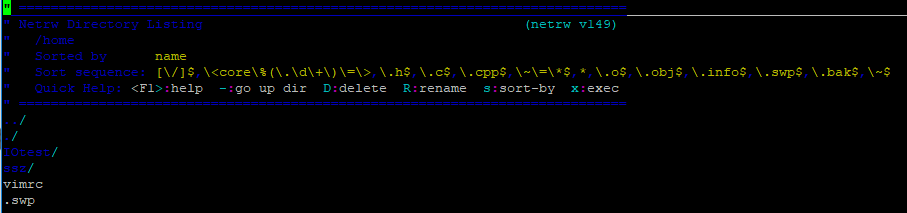
成功：0；失败：-1，设置errno为相应值

#### 文件、目录权限

注意：目录文件也是“文件”。其文件内容是该目录下所有子文件的目录项dentry，可以尝试用vim打开一个目录。

**案例：vim打开一个目录**





**文件、目录权限**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 读r | 写w | 执行x |
| 文件 | 文件的内容可以被查看  cat、more、less… | 内容可以被修改  vi、>… | 可以运行产生一个进程  ./文件名 |
| 目录 | 目录可以被浏览  ls、tree… | 创建、删除、修改文件  mv、touch、mkdir… | 可以被打开、进入  cd |

目录设置黏住位：若有w权限，创建不变，删除、修改只能由root、目录所有者、文件所有者操作。

#### opendir函数

根据传入的目录名打开一个目录（库函数（用man 3 opendir来看函数介绍）），头文件<dirent.h>

 DIR\* 类似于FILE\*（目录结构体指针）

成功返回指向该目录结构体指针，失败返回NULL

参数支持相对路径、绝对路径两种方式：例如：打开当前目录：① getcwd(), opendir() ② opendir(“.”)。

#### closedir函数

关闭打开的目录



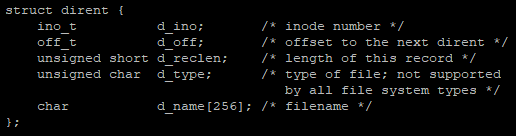
成功：0；失败：-1，设置errno为相应值

#### readdir函数

读取目录，注意输入的是DIR指针，即opendir的返回值，输出的是dirent指针

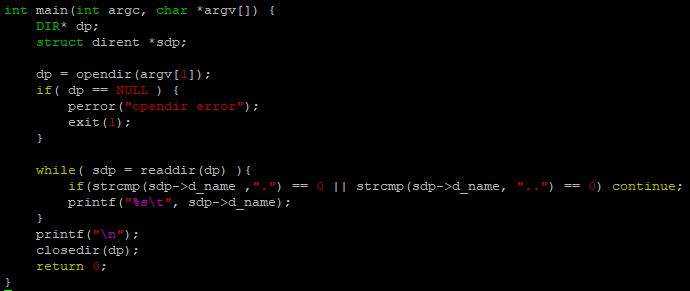


dirent结构体

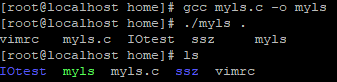


取目录名，不要超过255字节

**案例：实现ls功能**



执行程序



#### rewinddir函数

回卷目录读写位置至起始



#### telldir/seekdir函数

**获取**目录读写位置



成功：与dirp相关的目录当前读写位置；失败：-1，设置errno

**修改**目录读写位置



参数offset一般由telldir函数的返回值来决定

**案例：递归遍历目录，实现ls -R**

1. 判断命令行参数，获取用户要查询的目录名。argc, argv[1]

如果用户没有指定目录，默认使用的是当前目录，即”./”，argc==1 —> ./

1. 判断用户指定的是否是目录。stat函数，S\_ISDIR
2. 读目录

opendir()

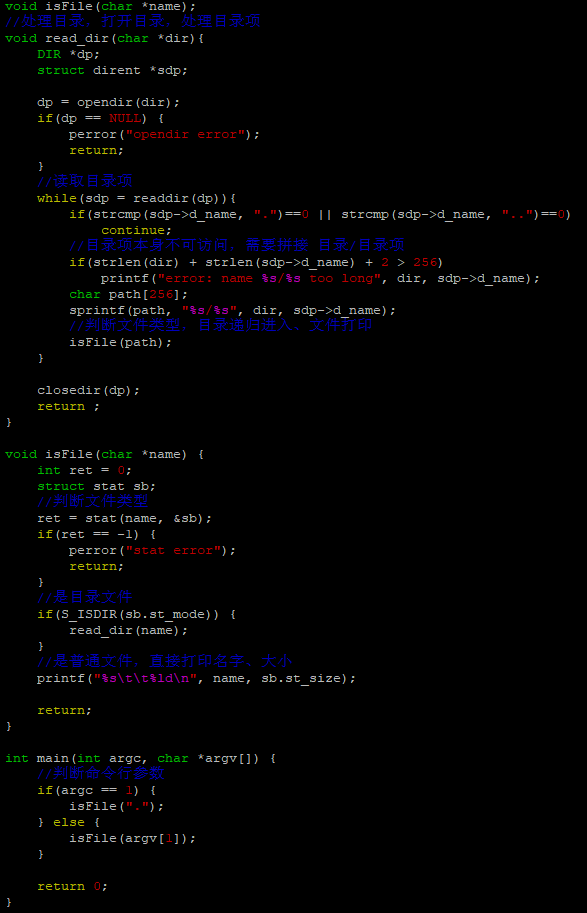
while(readdir()) {

普通文件，直接打印

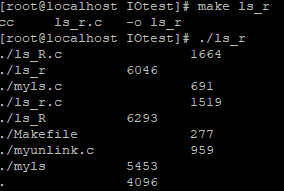
目录：拼接目录访问绝对路径，递归调用自己

}

closedir()



执行程序



### 13、重定向

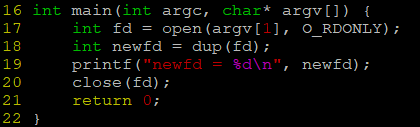
dup和dup2函数



成功：返回一个新文件描述符；失败：-1，设置errno为相应值



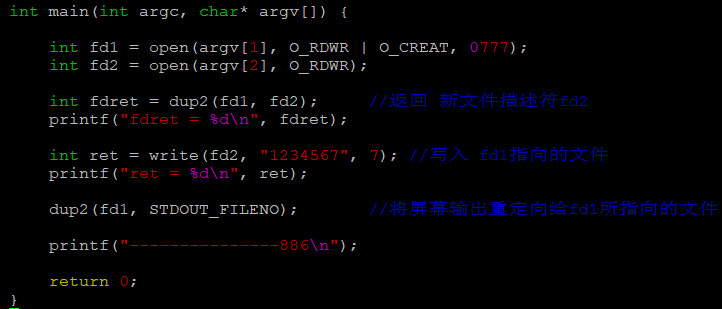
**案例1：验证dup函数**



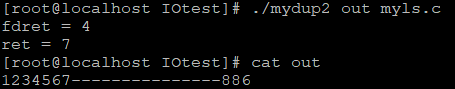
返回新的文件描述符，应该为4，运行程序



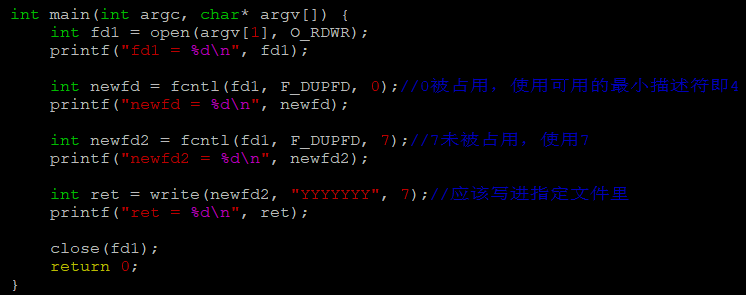
**案例2：使用dup2实现输出重定向功能（将屏幕输出到out文件里）**



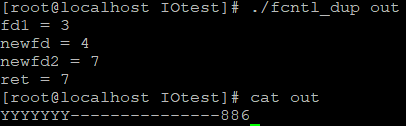
程序中将fd1复制给fd2，即fd2也指向fd1指向的文件，此时fdret应该为fd2的值4；然后向fd2所指的文件里写入1234567，ret应该为7，最后将标准输出设备标识符指向fd1指向的文件，下一行打印的-----------------886应该输出到fd1所指的文件里。



**案例3：利用fcntl函数实现dup2功能**



运行程序



## 七、进程

### 1、进程相关概念

#### 1.1、程序和进程

程序，是指编译好的二进制文件，在磁盘上，不占用系统资源（CPU、内存、打开的文件、设备、锁……）

进程，是一个抽象的概念，与操作系统原理联系紧密。进程时活跃的程序，占用系统资源。在内存中执行。（程序运行起来，产生一个进程）

程序 —> 剧本（纸） 进程 —> 戏（舞台、演员、灯光、道具……）

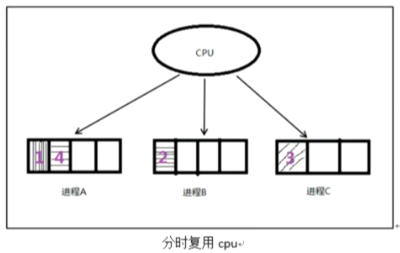
同一个剧本可以在多个舞台同时上演，同样，同一个程序也可以加载为不同的进程（彼此之间互不影响）

如：同时两个终端。各自都有一个bash但彼此ID不同。

#### 1.2、并发

并发，在操作系统中，一个时间段中有多个进程都处于已启动运行到运行完毕之间的状态。但是，任一个时刻点上只有一个进程在运行。

例如，当下，我们使用计算机可以边听音乐边聊天边上网。若笼统的将他们均看作一个进程的话为什么可以运行呢，因为**并发**。



#### 1.3、单道程序设计

所有进程一个一个排队执行，若A阻塞，B只能等待，即使CPU处于空闲状态。而在人机交互时阻塞的出现是必然的。所有这种模型在系统资源利用上极其不合理，在计算机发展历史上存在不久，大部分便被淘汰了。

#### 1.4、多道程序设计

在计算机内存中同时存放几道相互独立的程序，它们在管理程序控制之下，相互穿插的运行。多道程序设计必须有硬件基础作为保证。

**时钟中断**即为多道程序设计的理论基础。并发时，任意进程在执行期间都不希望放弃CPU。因此系统需要一种强制让进程让出CPU资源的手段。时钟中断有硬件基础作为保障，对进程而言不可抗拒。操作系统中的中断处理函数，来负责调度程序执行。

有多道程序设计模型中，多个进程轮流使用CPU（分时复用CPU资源）。而当下常见CPU为纳秒级，1秒可以执行大约10亿条指令。由于人眼的反应速度是毫秒级，所以看似同时在执行。

1s = 1000ms， 1ms = 1000μs，1μs = 1000ns

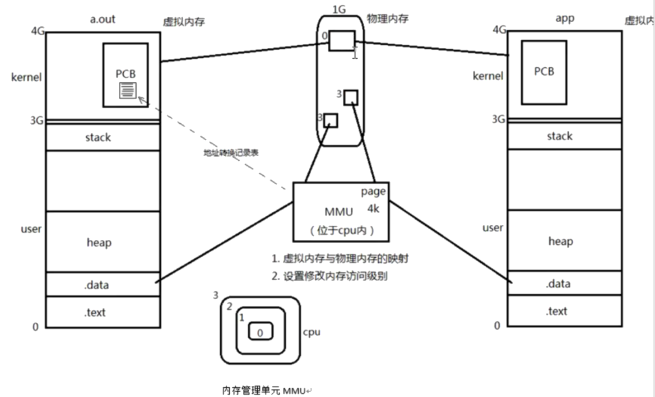
实质上，并发是宏观并行，微观串行。

#### 1.5、CPU和MMU

对于每一个进程在32位操作系统中都会产生一个4G的虚拟内存，如图，a.out和app两个可执行程序都会产生4G的虚拟内存。对于两者的.data区的数据会通过**MMU**（内存管理单元）映射到物理内存的不同区域。而两者的内核区内的PCB进程控制块则映射到物理内存的同一区域，只是在这一区域内有两个结构体而已，并不会覆盖。

其中，.text区也会映射到物理内存，而.rodata区和.data区需不需要映射到物理内存需要视情况而定，比如程序没有常量不需要映射.rodata区，没有全局变量不需要映射.data区。

**提示：**一个寄存器大小为4字节。MMU大小为4KB。



#### 1.6、进程控制块PCB

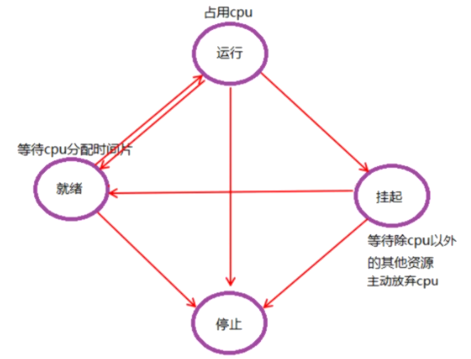
我们知道，每个进程在内核中都有一个进程控制块PCB来维护进程相关的信息，Linux内核的进程控制块是task\_struct结构体。

/usr/src/kernel/linux-header-3.16.0-30/include/linux/sched.h文件可以查看struct task\_struct结构体定义。其内部成员有很多，我们重点掌握以下部分即可：

* 进程id。系统中每个进程都有唯一的id，在C语言中用pid\_t类型表示，其实就是一个非负整数。
* 进程的状态，有就绪、运行、挂起、停止等状态。
* 进程切换时需要保存和恢复的一些CPU寄存器。
* 描述虚拟地址空间的信息（MMU地址）。
* 当前工作目录（Current Working Directory）。
* umask掩码。
* 文件描述符表，包含很多指向file结构体的指针。
* 和信号相关的信息。
* 用户id和组id。
* 会话（Session）和进程组
* 进程可以使用的资源上限（Resource Limit）

#### 1.7、进程状态

进程基本状态有5种。分别为初始态、就绪态、运行态、挂起态和终止态。其中初始态为进程准备阶段，常与就绪态结合来看。



### 2、环境变量

环境变量，是指在操作系统中用来指定操作系统运行环境的一些参数。通常具备以下特征：

① 字符串（本质）；② 有特殊的格式：名=值[:值]；③ 值用来描述进程环境信息

**存储形式**：与命令行参数类似。Char \*[]数组，数组名environ，内部存储字符串，NULL作为哨兵结尾。

**使用形式**：与命令行参数类似。

**加载位置**：与命令行参数类似。位于用户区，高于stack的起始位置。

**引入环境变量表**：须声明环境变量。extern char \*\* environ。

#### 2.1、常见环境变量

按照惯例，环境变量字符串都是name=value这样的形式，大多数name由大写字母加下划线组成，一般把name的部分叫做环境变量，value的部分则是环境变量的值。环境变量定义了进程的运行环境，一些比较重要的环境变量的含义如下：

##### PATH

可执行文件的搜索路径。ls命令也是一个程序，执行它不需要提供完整的路径名/bin/ls，然而通常我们执行当前目录下的程序a.out却需要执行完整的路径名/a.out，这是因为PATH环境变量的值里面包含了ls命令所在的目录/bin，却不包含a.out所在的目录。PATH环境变量的值可以包含多个目录，用:号分开。在shell中用echo命令可以查看这个环境变量的值：

echo $PATH

##### SHELL

当前shell，它的值通常是/bin/bash。

##### TERM

当前终端类型，在图形界面终端下它通常是xterm，终端类型决定了一些程序的输出方式，比如图形界面终端可以显示汉字，而字符终端一般不行。

##### LANG

语言和locale，决定了字符编码以及时间、货币等信息的显示格式。

##### HOME

当前用户主目录的路径，很多程序需要在主目录下保存配置文件，使得每个用户在运行程序时都有自己的一套配置。

**注意：**env命令查看所有环境变量

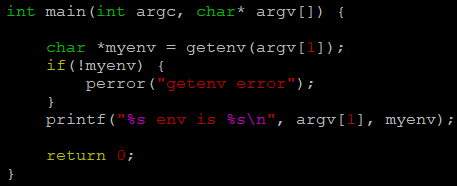
#### 2.2、getenv函数

获取环境变量值

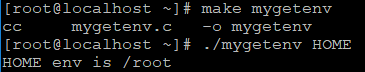


成功：返回环境变量的值；失败：NULL（name不存在）

**案例1：获取HOME的环境变量值**



执行程序



#### 2.3、setenv函数

设置环境变量值



添加环境变量name，其值为value，如果name不存在，则直接添加；如果name存在，则视overwrite值决定是否更改name的值为value，若overwrite非零，则更改其值为value。

成功：返回0，失败：返回-1。

#### 2.4、unsetenv函数

删除环境变量

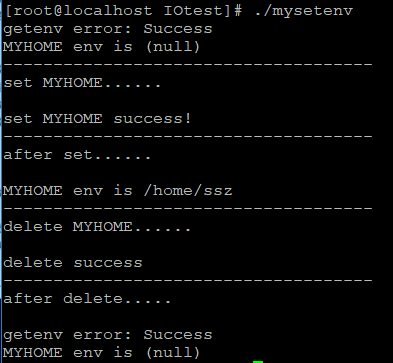


成功：返回0；失败：返回-1并设置errno

**案例2：设置环境变量MYHOME=/home/ssz**



执行程序



：一开始不存在MYHOME，所以获取环境变量失败，然后设置MYHOME后，获取成功，接着删除成功，然后再获取MYHOME失败。达到预期效果。

### 3、进程控制

#### 3.1、fork函数

创建一个子进程。



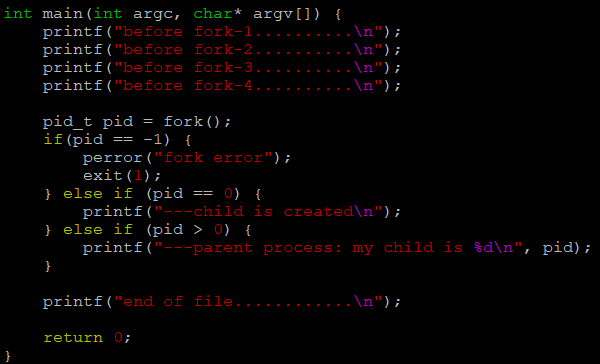
失败：返回-1；成功：返回：① 父进程返回子进程的ID（非负）② 子进程返回0

pid\_t类型表示进程ID，但为了表示-1，它是有符号整型。（0不是有效进程ID，int最小，为1）

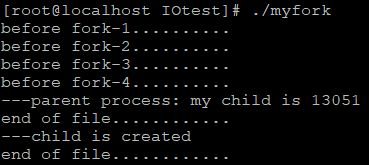
注意返回值，不是fork函数能返回两个值，而是fork后，fork函数变为两个，父子需【各自】返回一个。

**注意：**子进程开始的地方是父进程代码中fork后面的部分，即父进程执行过的部分子进程并不执行。

**案例1：使用fork创建子进程**



执行程序



可见，“end of file……….”打印了两次，一次父进程打印，一次子进程打印，对应上述的**注意**。

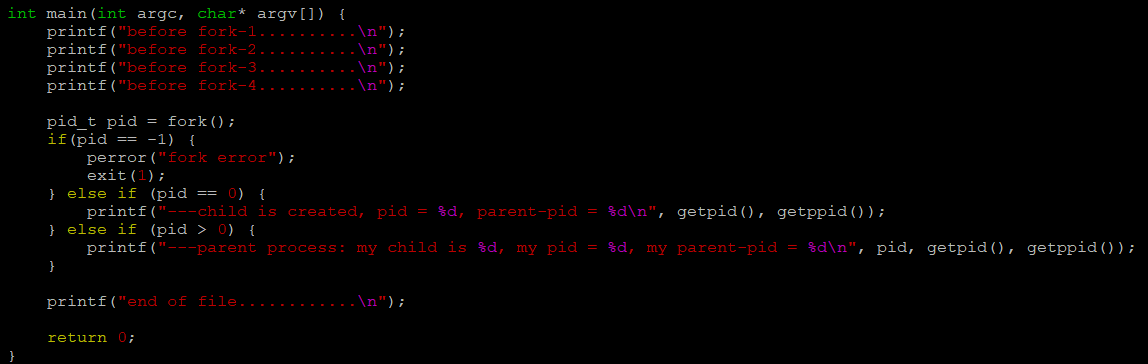
#### 3.2、getpid函数/getppid函数

获取进程id

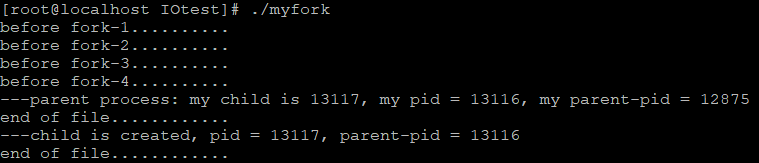


getpid获取进程本身的id，getppid获取进程的父进程id。

**案例2：在上一案例中使用getpid和getppid函数**



执行程序



观察程序结果可知，子进程是13117，父进程是13116，父进程的父进程是12875，这个12875是啥呢？



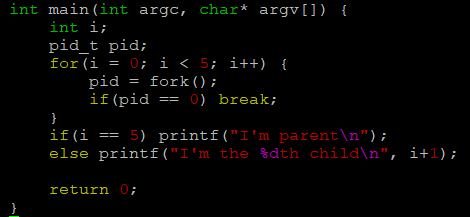
可知，12875是bash。

##### 循环创建n个子进程

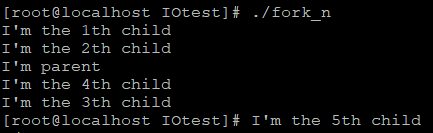
一次fork函数调用可以创建一个子进程。那么创建n个子进程应该怎么实现呢？

简单想，for(int i = 0; i < n; i++) {fork();}即可。但这样创建的是n个子进程吗？

这种创建方式产生的是个子进程。



执行程序



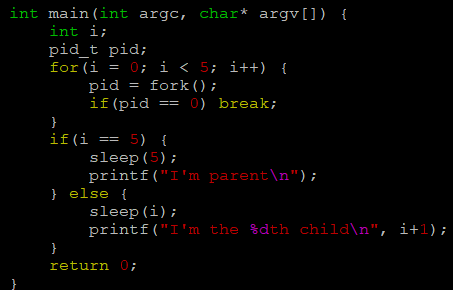
1、为什么会出现没有按照顺序打印的情况呢？

【原因】：父进程产生5个子进程实际在操作系统解释代码交给CPU去执行时，这5个进程相当于是同时产生的，因此需要争夺CPU去打印，所以顺序不一定是按照i值来的。

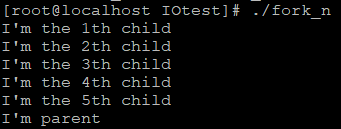
2、为什么会先出现bash才出现第5个进程打印信息？

【原因】：因为当父进程结束后才会出现bash，而子进程与父进程的父进程竞争CPU失败，导致先出现bash。

**解决办法：**加入睡眠等待



父进程等待5s，第i个子进程等待i秒



#### 3.3、getuid函数

获取当前进程实际用户ID



获取当前进程有效用户ID



#### 3.4、getgid函数

获取当前进程使用用户组ID



获取当前进程有效用户组ID



#### 3.5、进程共享

父子进程之间在fork后，有哪些相同，哪些相异之处呢？

刚fork之后：

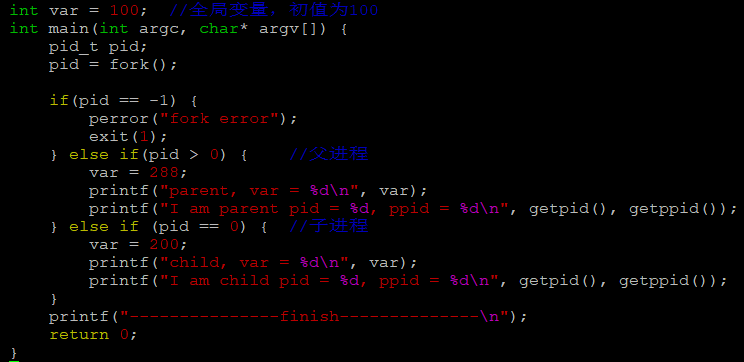
父子**相同**处：全局变量、.data、.text、栈、堆、环境变量、用户ID、宿主目录、进程工作目录、信号处理方式……

父子**不同**处：进程ID、fork返回值、父进程ID、进程运行时间、闹钟（定时器）、未决信号集

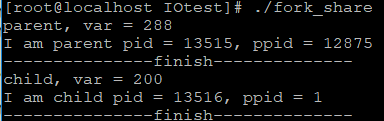
似乎，子进程复制了父进程0-3G用户空间内容，以及父进程的PCB，但pid不同。真的每fork一个子进程都要将父进程的0-3G地址空间完全拷贝一份，然后映射到物理内存吗？

当然不是！父子进程间遵循**读时共享写时复制**的原则。这样设计，无论子进程执行父进程的逻辑还是执行自己的逻辑都能节省内存开销。

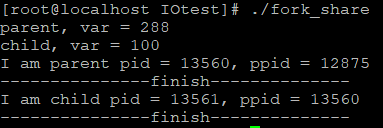
**案例3：编写测试程序，父子进程是否共享全局变量**



执行程序



可以看出，因为父进程和子进程都是写操作，所以执行复制，倘若注释掉var=200，执行结果如下



因为子进程是读操作，所以是共享全局变量，因此是100，而父进程因为是写操作，所以它更改的实际上是复制品。

**重点注意**：**躲避父子进程共享全局变量的知识误区！**

【**重点**】：**父子进程共享：1. 文件描述符（打开文件的结构体），2. mmap建立的映射区（进程间通信详解）**

特别的，fork之后父进程先执行还是子进程先执行不确定，取决于内核所使用的调度算法。

#### 3.6、gdb调试

使用gdb调试的时候，gdb只能跟踪一个进程。可以在fork函数调用之前，通过指令设置gdb调试工具跟踪父进程或者跟踪子进程。默认跟踪父进程。

set follow-fork-mode child 命令设置gdb在fork之后跟踪子进程

set follow-fork-mode parent 设置跟踪父进程

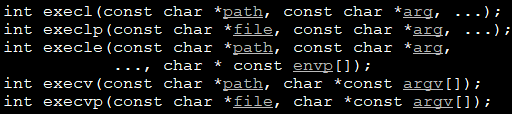
**【注意】，一定要在fork函数调用之前设置才有效。**

### 4、exec函数族

fork创建子进程后执行的是和父进程相同的程序（但有可能执行不同的代码分支），子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时，该子进程的用户空间代码和数据完全被新程序替换，从新程序的启动例程开始执行。调用exec并不创建新进程，所以调用exec前后该进程的id并未改变。

将当前进程的.text、.data替换为所要加载的程序的.text、.data，然后让进程从新的.text的第一条指令开始执行，但进程ID不变，换核不换壳。

其实有六种以exec开头的函数，统称为exec函数：





#### 4.1、execlp函数

加载一个进程，借助PATH环境变量



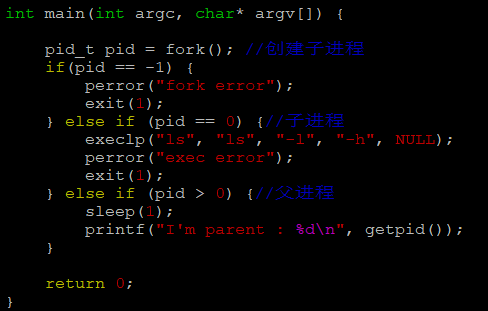
成功：无返回；失败：-1

**参数1**：要加载的程序的名字。该函数需要配合PATH环境变量来使用，当PATH中所有目录搜索后没有参数1，则出错返回。

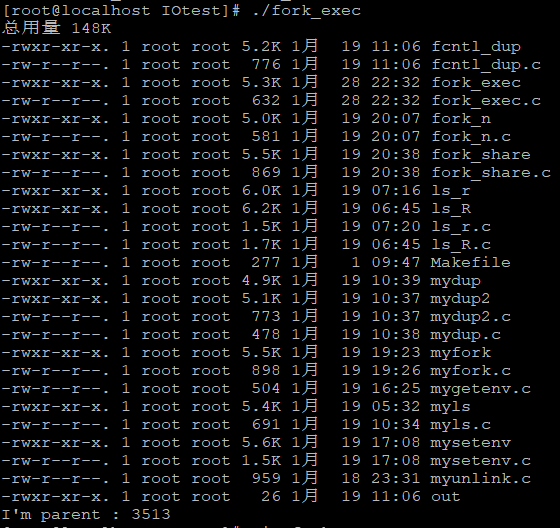
**参数2**：从argv[0]（即文件名）开始

该函数通常用来调用系统程序。如：ls、date、cp、cat等命令。

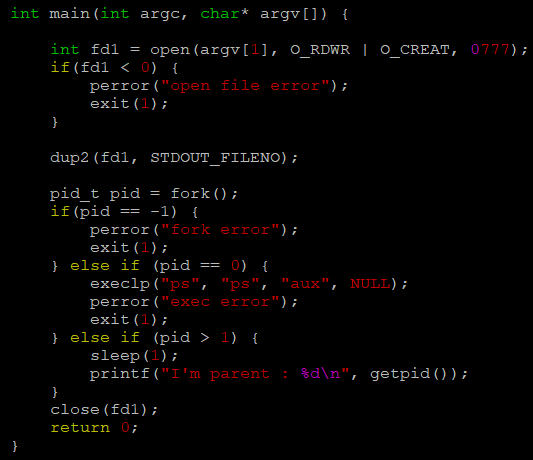
**案例1：子进程调用ls命令**



执行程序



**案例2：将当前系统中的进程信息，打印到文件中**



#### 4.2、execl函数

加载一个进程，通过 **路径+程序名** 来加载



对比execlp函数，如加载“ls”命令带有-l，-F参数

execlp(“ls”, “ls”, “-l”, “-F”, NULL); 使用程序名在PATH中搜索

execl(“/bin/ls”, “ls” , “-l”, “-F”, NULL); 使用参数1给出的绝对路径搜索

#### 4.3、execvp函数

加载一个进程，使用自定义环境变量env

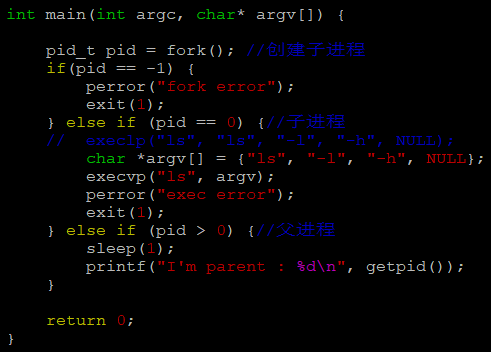


变参形式：①… ②argv[] （main函数也是变参函数）

变参终止条件：① NULL结尾 ② 固参指定

**execvp与execlp参数形式一定，原理一致**

**案例3：使用execvp实现ls功能**



#### 4.4、exec函数族一般规律

exec函数一旦调用成功即执行新的程序，不返回。只有失败才返回，错误值-1.所以通常我们直接在exec函数调用后直接调用perror()和exit()，无需if判断。

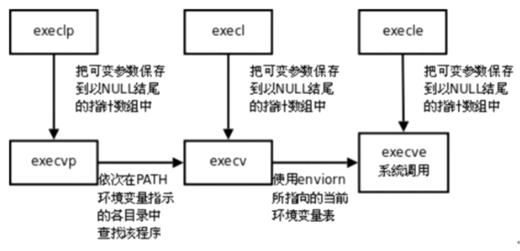
l (list) 命令行参数列表

p (path) 搜索file时使用path变量

v (vector) 使用命令行参数列表

e (environment) 使用环境变量数组，不适用进程原有的环境变量，设置新加载程序运行的环境变量

事实上，只有execve时真正的系统调用，其他五个函数最终都调用execve，所以execve在man手册第2节，其他函数在man手册第3节，这些函数之间的关系如下图：



### 5、回收子进程

#### 5.1、孤儿进程

孤儿进程：父进程先于子进程结束，则子进程称为孤儿进程，子进程的父进程称为init进程，称为init进程领养孤儿进程。

#### 5.2、僵尸进程

僵尸进程：进程终止，父进程尚未回收，子进程残留资源（PCB）存放于内核中，变成僵尸进程。

特别注意：僵尸进程是不能使用kill命令清除掉的。因为kill命令只是用来终止进程的，而僵尸进程已经终止。思考！用什么办法可以清除掉僵尸进程呢？

可以通过杀死其父进程，让该进程变为孤儿进程，进而被init进程回收。

#### 5.3、wait函数

一个进程在终止时会关闭所有文件描述符，释放在用户空间分配的内存，但它的PCB还保留着，内核在其中保存了一些信息：如果是正常终止则保存着退出状态，如果是异常终止则保存着导致该进程终止的信号是哪个。这个进程的父进程可以调用wait或waitpid获取这些信息，然后彻底清除掉这个进程。我们知道一个进程的退出状态可以在Shell中用特殊变量$？查看，因为Shell是它的父进程，当它终止时Shell调用wait或waitpid得到他的退出状态同时彻底清除掉这个进程。

**父进程**调用wait函数可以回收子进程终止信息。该函数有三个功能：

1. 阻塞等待子进程退出
2. 回收子进程残留资源
3. 获取子进程结束状态（退出原因）



成功：清理掉的子进程ID；失败：-1（没有子进程）

（status是传出参数，也就是wait返回值赋给status）

当进程终止时，操作系统的隐式回收机制会：1.关闭所有文件描述符；2.释放用户空间分配的内存。内核的PCB仍然存在。其中保存该进程的退出状态。（正常终止—>退出值；异常终止—>终止信号）

可使用wait函数传出参数status来保存进程的退出状态。借助宏函数来进一步判断进程终止的具体原因。宏函数可分为如下三组：

1. WIFEXITED(status)为非0 —> 进程正常结束

WEXITSTATUS(status)如上宏为真，使用此宏 —> 获取进程退出状态（exit的参数）

1. WIFSIGNALED(status)为非0 —> 进程异常终止

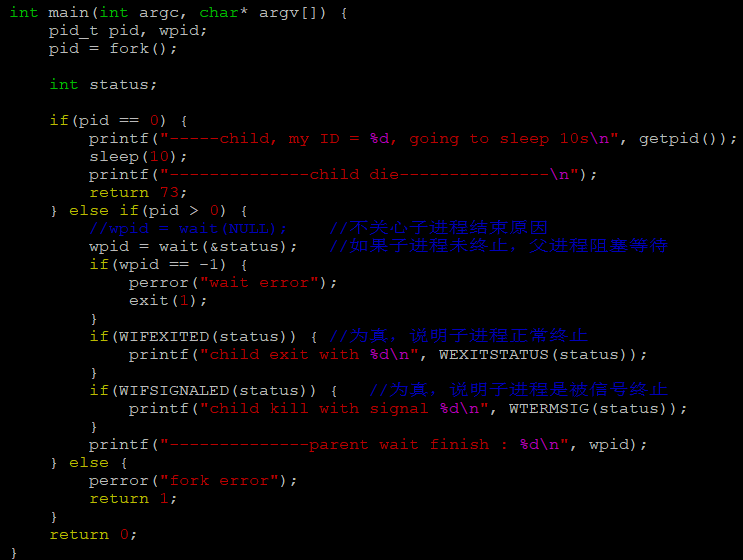
WTERMSIG(status)如上宏为真，使用此宏 —> 获取使进程终止的那个信号的编号

1. WIFSTOPPED(status)为非0 —> 进程处于暂停状态

WSTOPSIG(status)如上宏为真，使用此宏 —> 取得使进程暂停的那个信号的编号

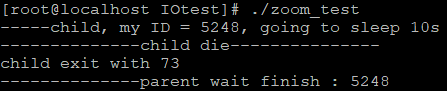
WIFCONTINUED(status)为真 —> 进程暂停后已经继续运

**案例1：验证wait函数的功能**



首先fork创建子进程，打印子进程ID，然后等待10s，父进程使用wait阻塞等待子进程退出，并使用wpid返回子进程ID。

运行程序



#### 5.4、waitpid函数

作用同wait，但可指定pid进程清理，可以不阻塞。



成功：返回清理掉的子进程ID；失败：-1（无子进程）

特殊参数和返回情况：

参数pid：

> 0 回收指定ID的子进程

-1 回收任意子进程（相当于wait）

0 回收和当前调用waitpid一个组的所有子进程

< -1 回收指定进程组内的任意子进程（-进程组id）

返回：

> 0：表示成功回收的子进程pid

0：参数options为WNOHANG，且子进程正在运行

-1：失败。设置errno

【注意】：**一次wait或waitpid调用只能清理一个子进程**，清理多个子进程应使用循环。

## 八、进程间通信

### 1、IPC方法

Linux环境下，进程地址空间相互独立，每个进程各自有不同的用户地址空间。任何一个进程的全局变量在另一个进程中都看不到，所以进程和进程之间不能相互访问，要交换数据必须通过内核，在内核中开辟一块缓冲区，进程1把数据从用户空间拷贝到内核缓冲区，进程2再从内核缓冲区把数据读走，内核提供的这种机制称为**进程间通信**（IPC）。

在进程间完成数据传递需要借助操作系统提供特殊的方法，如：文件、管道、信号、共享内存、消息队列、套接字、命名管道等。随着计算机的蓬勃发展，一些方法由于自身设计缺陷被淘汰或者弃用。现今常用的进程间通信方式有：

1. 管道（使用最简单）pipe
2. 命令管道（无血缘关系间通信）fifo
3. 信号（开销最小）signal
4. 共享映射区（无血缘关系间通信，可反复读取）mmap
5. 本地套接字（最稳定）domain

### 2、管道pipe

#### 2.1、管道的概念

管道是一种最基本的IPC机制，作用于有血缘关系的进程之间，完成数据传递。调用pipe系统函数即可创建一个管道。有如下特质：

1. 其本质是一个伪文件（实为内核缓冲区）（Linux文件类型有七种，普通文件、目录和软链接占用磁盘空间，剩下4种类型，即字符设备、块设备、管道、套接字都是伪文件，不占用磁盘空间，但占用内存）
2. 由两个文件描述符引用，一个表示读端，一个表示写端
3. 规定数据从管道的写端流入管道，从读端流出

管道的**原理**：管道实为内核使用环形队列机制，借助内核缓冲区（4K）实现。

管道的**局限性**：

1. 数据不能进程自己写、自己读。
2. 管道中数据不可反复读取。一旦读走，管道中不再存在。
3. 采用半双工通信方式，数据只能在单方向上流动。
4. 只能在有公共祖先的进程间使用管道

常见的通信方式有，单工通信、半双工通信、全双工通信

#### 2.2、pipe函数

创建、并打开管道



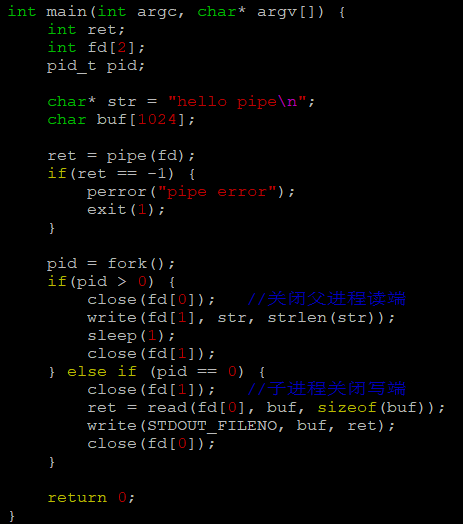
参数： fd[0]：读端 fd[1]：写端 成功：0；失败：-1，设置errno

函数调用成功返回r/w两个文件描述符。无需open，但需手动close。

管道创建成功以后，创建该管道的进程（父进程）同时掌握着管道的读端和写端。如何实现父子进程间通信，通常可以采用如下步骤：

1. 父进程调用pipe函数创建管道，得到两个文件描述符fd[0]、fd[1]指向管道的端读和写端。
2. 父进程调用fork创建子进程，那么子进程也有两个文件描述符指向同一管道。
3. 父进程关闭管道读端，子进程关闭管道写端。父进程可以向管道中写入数据，子进程将管道中的数据读出。由于管道是利用环形队列实现的，数据从写端流入管道，从读端流出，这样就实现了进程间通信。

**案例1：父子进程使用管道通信，父进程写入字符串，子进程读出字符串，打印到屏幕**



运行程序



#### 2.3、管道的读写行为

使用管道需要注意以下4种特殊情况（假设都是阻塞I/O操作，没有设置O\_NONBLOCK标志）

1. 如果所有指向管道**写端**的文件描述符都关闭了（管道写端引用计数为0），而仍然有进程从管道的读端读数据，那么管道中剩余的数据都被读取后，再次read会返回0，就像读到文件末尾一样。

2. 如果有指向管道**写端**的文件描述符没关闭（管道写端引用计数大于0），而持有管道写端的进程也没有向管道中写数据，这时有进程从管道读数据，那么管道中剩余的数据都被读取后，再次read会阻塞，直到管道中有数据可读了才读取数据并返回。

3. 如果所有指向管道**读端**的文件描述符都关闭了（管道读端引用计数为0），这时有进程向管道的写端write，那么该进程会收到信号SIGPIPE，通常会导致进程异常终止。当然也可以对SIGPIPE信号实施捕捉，不终止进程。

4. 如果有指向管道**读端**的文件描述符没关闭（管道读端引用计数大于0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写数据，那么在管道被写满时再次write会阻塞，直到管道中有空位置了，才写入数据并返回。

**总结：**

1. 读管道：1. 管道中有数据，read返回实际读到的数据

2. 管道中无数据

(1) 写端被全部关闭，read返回0（好像读到文件结尾）

(2) 写端没有被全部关闭，read阻塞等待（不久的将来可能有数据抵

达，此时会让出CPU）

1. 写管道：1. 管道读端全部被关闭，进程异常终止（也可使用捕捉SIGPIPE信号，

使进程不终止）

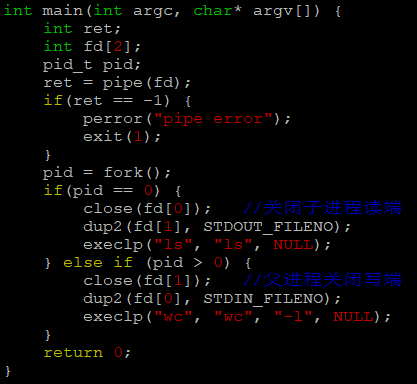
2. 管道读端没有全部关闭：

(1) 管道已满，write阻塞

(2) 管道未满，write将数据写入，并返回实际写入的字节数

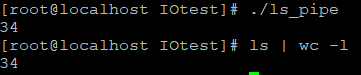
**案例2：使用管道实现父子进程通信，完成：ls | wc -l。假定父进程实现ls，子进程实现wc。**

ls命令正常会将结果集写到stdout，但现在会写入管道的写端；wc -l正常应该从stdin读取数据，但此时会从管道的读端读。



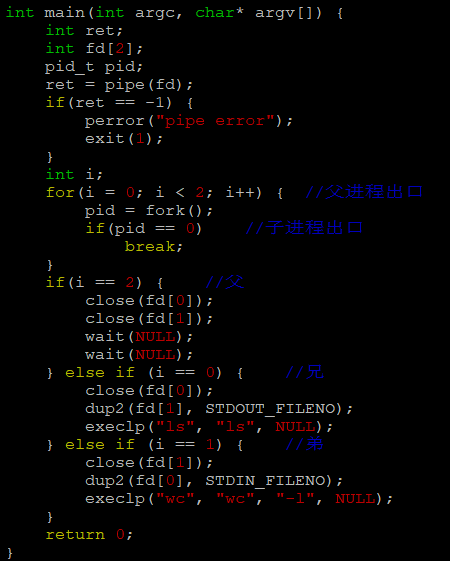
【注意】：这里是使用父进程执行wc而使用子进程执行ls，原因就在于如果倒过来会存在之前出现的bash先于程序结果显示在终端上。因为父进程执行ls，而子进程执行wc，wc的内容来源于管道的读端，而管道的内容来源于管道的写端，因此必须是父进程先执行，所以就有可能造成父进程先结束。解决办法就是让父进程晚于子进程结束，只需让二者功能调换即可。且无法使用sleep来解决，因为execlp函数会跳出程序，不会执行后面的sleep，也就无法起到延迟的作用。

执行程序，结果如下



**案例3：使用管道实现兄弟间进程通信。兄：ls 弟：wc -l 父：等待回收子进程**

**要求，使用“循环创建N个子进程”模型创建兄弟进程，使用循环因子i标识**



运行程序



#### 2.4、管道缓冲区大小

可以使用ulimit -a 命令来查看当前系统中创建管道文件所对应的内核缓冲区大小。通常为：

也可以使用fpathconf函数，借助参数选项来查看



成功：返回管道的大小；失败：返回-1，设置errno

#### 2.5、管道的优劣

**优点**：简单，相比信号、套接字实现进程间通信，简单很多

**缺点**：1. 只能单向通信，双向通信需建立两个管道

2. 只能用于父子、兄弟进程间通信。该问题后来使用fifo命名管道解决。

### 3、FIFO

FIFO常被称为命名管道，以区分管道(pipe)。管道只能用于“有血缘关系”的进程间。但FIFO可以在不相关进程间交换数据。

FIFO是Linux基础文件类型中的一种。但是，FIFO文件在磁盘上没有数据块，仅仅用来标识内核中的一条通道。各进程可以打开这个文件进行read/write，实际上是在读写内核通道，这样就实现了进程间通信。

创建方式：

1. 命名：mkfifo 管道名
2. 库函数：成功：返回0；失败：返回-1



一旦使用mkfifo创建了一个FIFO，就可以使用open打开它，常见的文件I/O函数都可用于fifo。如close、read、write、unlink等。

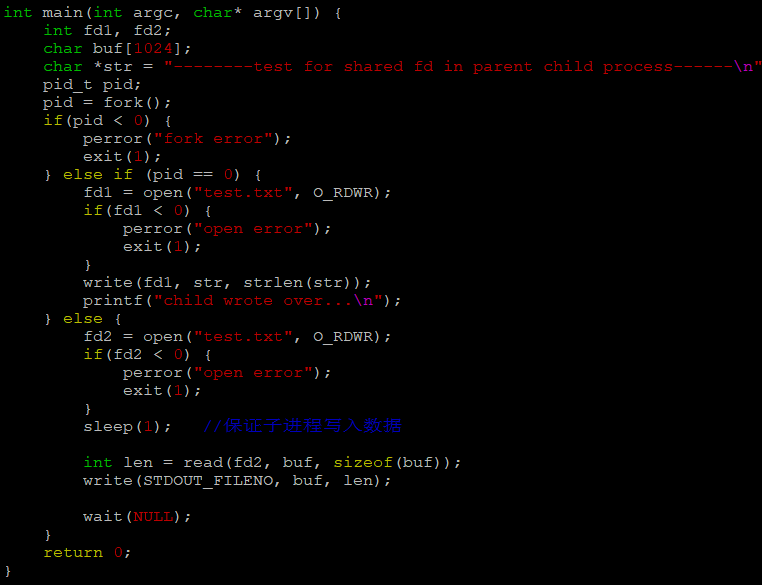
### 4、共享存储映射

#### 4.1、文件进程间通信

使用文件也可以完成IPC，理论依据是，fork后，父子进程共享文件描述符。也就共享打开的文件。

**案例：父子进程共享打开的文件，借助文件进行进程间通信。**

让子进程打开test.txt文件。然后往里面写入一串字符，接着让父进程共享打开该文件，读取该字符串并打印到屏幕上。



运行程序

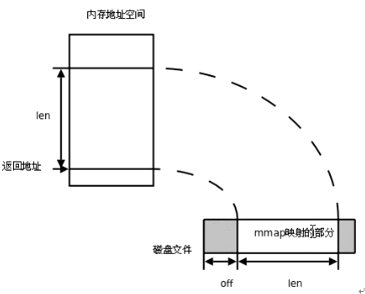


【注意】

#### 4.2、存储映射I/O

存储映射I/O（Memory-mapped I/O）使一个磁盘文件与存储空间中的一个缓冲区相映射。于是当从缓冲区中取数据，就相当于读文件中的相应字节。与此类似，将数据存入缓冲区，则相应的字节就自动写入文件。这样，就可以在不适用read和write函数的情况下，使用地址（指针）完成I/O操作。

使用这种方法，首先应通知内核，将一个指定文件映射到存储区域中。这个映射工作可以通过mmap函数来实现。



##### mmap函数

创建共享内存映射，头文件<sys/mman.h>



成功：返回创建的映射区首地址；失败：返回MAP\_FAILED宏，设置errno

**参数**：

addr：建立映射区的首地址，由Linux内核指定。使用时，直接传递NULL，系统自动分类地址。

length：欲创建映射区的大小。（<=文件实际大小）

prot：映射区权限PROT\_READ、PROT\_WRITE、PROT\_READ|PROT\_WRITE

flags：标志位参数（常用于设定更新物理区域、设置共享、创建匿名映射区）

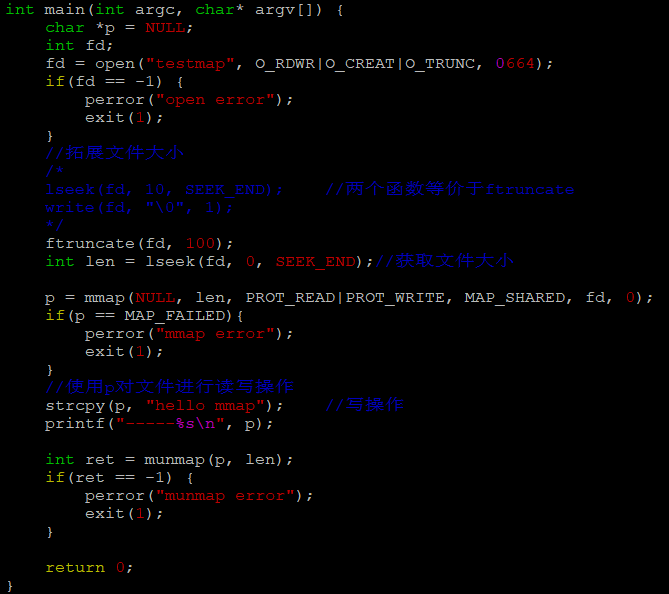
MAP\_SHARED：会将映射区所做的操作反映到物理设备（磁盘）上

MAP\_PRIVATE：映射区所做的修改不会反映到物理设备

fd：用于创建共享内存映射区的那个文件的文件描述符

offset：偏移位置，默认0，表示映射文件全部。需是4K的整数倍。

**案例：使用mmap创建映射区，实现read/write功能**



运行程序



##### mmap注意事项

**思考：**

1. 可以open的时候O\_CREAT一个新文件来创建映射区吗？
2. 如果open时，O\_RDONLY，mmap时PROT参数指定PROT\_READ|PROT\_WRITE

会怎样？无效参数错误

1. 文件描述符先关闭，对mmap映射有没有影响？有
2. 如果文件偏移量为1000会怎样？mmap错误
3. 对mem越界操作会怎样？
4. 如果mem++，munmap可否成功？不能
5. mmap什么情况下会调用失败？
6. 如果不检测mmap的返回值，会怎样？

**总结：使用mmap时务必注意以下事项：**

1. 创建映射区的过程中，隐含着一次对映射文件的**读操作**，所以文件不能只写权限。
2. 当MAP\_SHARED时，要求：**映射区的权限应<=文件打开的权限**（出于对映射区的保护）。而MAP\_PRIVATE则无所谓，因为mmap中的权限是对内存的限制，此时只需要open文件时有读权限即可。
3. 映射区的释放与文件关闭无关。只**要映射建立成功，文件可以立即关闭**。
4. 特别注意，当映射文件大小为0时，不能创建映射区。所以：用于映射的文件必须要有实际大小！！mmap使用时常常会出现**总线错误**，通常是由于共享文件存储空间大小引起的。如，400字节大小的文件，在建立映射区时offset 4096字节，则会报出总线错误。
5. munmap传入的地址一定是mmap的返回地址。**坚决杜绝指针++操作**。
6. 文件偏移量必须为4K的整数倍。
7. mmap创建映射区出错概率非常高，一定要检查返回值确保映射区建立成功后再进行后续操作。

**mmap保险调用方式：**

1. open（文件名, O\_RDWR）
2. mmap(NULL, 有效文件大小, PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0)

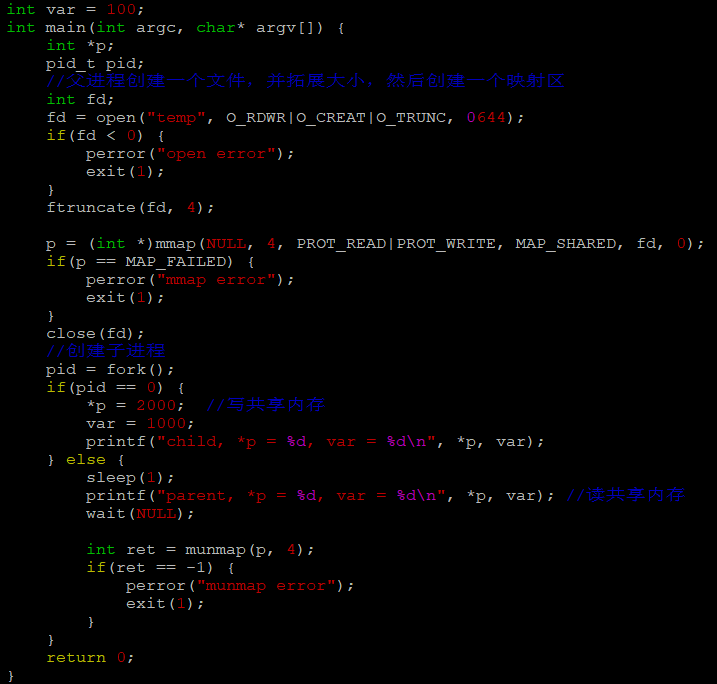
#### 4.3、mmap父子进程通信

父子等有血缘关系的进程之间也可以通过mmap建立的映射区来完成进程通信。但相应的在建立映射区时要指定对应的标志位参数flags：

MAP\_PRIVATE：（私有映射）父子进程各自独占映射区

MAP\_SHARED：（共享映射）父子进程共享映射区

**案例1：父进程创建映射区（MAP\_SHARED），子进程修改共享内容和全局变量，然后父进程读取共享内容和全局变量**



执行程序



结果，子进程更改成功后，共享的内容在父进程中也发生了改变，实现了父子进程间通信，而全局变量并没有改变，这是因为遵循“**读时共享写时复制**”原则。

**如果把权限设置为MAP\_PRIVATE的话，运行结果为**



可知，\*p并没有变化，所以并没有实现父子进程间通信。

#### 4.4、mmap无血缘关系进程间通信

实质上mmap是内核借助文件帮我们创建了一个映射区，多个进程之间利用该映射区完成数据传递，由于内核空间多进程共享，因此无血缘关系的进程间也可以使用mmap来完成进程间通信。只要设置相应的标志位参数flags为MAP\_SHARED即可。

【注意】：MAP\_ANON和/dev/zero都不能应用于非血缘关系进程间通信。只能用于亲子进程。

#### 4.5、匿名映射

通过使用我们发现，使用映射区来完成文件读写非常方便，父子进程间通信也比较容易。但缺陷是，每次建立映射区一定要依赖一个文件才能实现。通常为了建立映射区要open一个temp文件，创建好了再unlink、close掉，比较麻烦。可以直接使用匿名映射来代替。其实Linux系统给我们提供了创建匿名映射区的方法，无需依赖一个文件即可创建映射区，同样需要借助标志位flags来指定。

使用MAP\_ANONYMOUS（或MAP\_ANON）。如：



需要注意的是，MAP\_ANONYMOUS和MAP\_ANON这两个宏是Linux系统特有的宏。在类Unix系统中无该宏定义，可使用如下两步来实现匿名映射区的建立。

1. fd = open(“/dev/zero”, O\_RDWR);
2. p = mmap(NULL, size, PROT\_READ|PROT\_WRITE, MMAP\_SHARED, fd, 0);

/dev/zero ——> 字符设备，可从中想拿多少数据就可拿多少

/dev/null ——> 可向里写多少数据就写多少

## 九、信号

### 1、信号的概念

信号在我们的生活中随处可见，如：古代战争中摔杯为号；现代战争中的信号弹；体育比赛中使用的信号枪。。。。。。它们都有共性：1. 简单 2. 不能携带大量信息 3. 满足某个特设条件才发送。

信号是信息的载体，Linux/Unix环境下，古老、经典的通信方式，现下依然是主要的通信手段。

Unix早期版本就提供了信号机制，但不可靠，信号可能丢失。Berkeley和AT&T都对信号模型做了更改，增加了可靠信号机制，但彼此不兼容。POSIX.1对可靠信号例程进行了标准化。

#### 1.1、信号的机制

A给B发送信号，B收到信号之前执行自己的代码，收到信号后，不管执行到程序的什么位置，都要暂停运行，区处理信号，处理完毕再继续执行。与硬件中断类似——异步模式。但信号是软件层面上实现的中断，早期常被称为“软中断”。

**信号的特质**：由于信号是通过软件方法实现，其实现手段导致信号有很强的延时性。但对于用户来说这个延迟时间非常短，不易察觉。

每个进程收到的所有信号，都是由内核负责发送的，内核处理。

#### 1.2、与信号有关的事件和状态

**产生信号：**

1. 按键产生，如：Ctrl + C、Ctrl + Z、Ctrl + \
2. 系统调用产生：如：kill、raise、abort
3. 软件条件产生：如：定时器alarm
4. 硬件异常产生：如：非法访问内存（段错误）、除0（浮点数外）、内存对齐出错（总线错误）
5. 命令产生，如：kill命令

**递达：**递达并且到达进程

**未决：**产生和递达之间的状态。主要由于阻塞（屏蔽）导致该状态

**信号的处理方式：**

1. 执行默认动作
2. 忽略（丢弃）
3. 捕捉（调用户处理函数）

Linux内核的进程控制块PCB是一个结构体，task\_struct，除了包含进程ID、状态、工作目录、用户ID、组ID、文件描述符表，还包含了信号相关信息，主要指阻塞信号集和未决信号集。

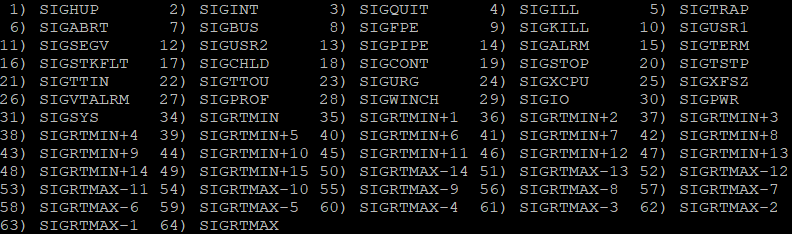
**阻塞信号集（信号屏蔽字）**：将某些信号加入集合，对他们设置屏蔽，当屏蔽x信号后，再收到该信号，该信号的处理将推后（解除屏蔽后）

**未决信号集：**

1. 信号产生，未决信号集中描述该信号的位立即翻转为1，表信号处于未决状态。当信号被处理，对应位翻转为0。这一时刻往往非常短暂。
2. 信号产生后由于某些原因（主要是阻塞）不能递达。这类信号的集合称之为未决信号集。在屏蔽解除前，信号一直处于未决状态。

#### 1.3、信号的编号

可以使用kill -l命令查看当前系统可使用的信号有哪些。



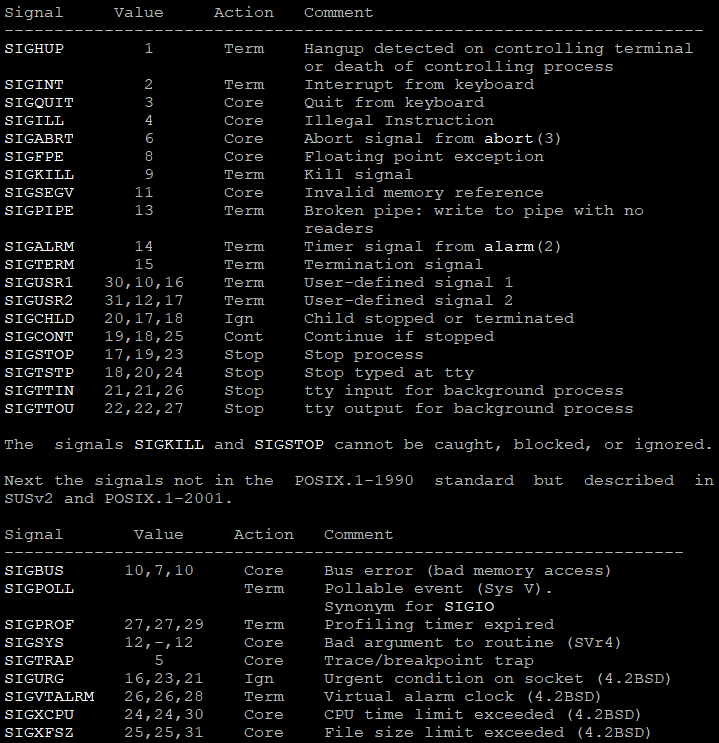
不存在编号为0的信号。其中1-31号信号称之为常规信号（普通信号或标准信号），34-64称之为实时信号，驱动编程与硬件相关。名字上区别不大。而前32个名字各不相同。

#### 1.4、信号四要素

分别为

1. 编号 2. 名称 3. 事件 4. 处理动作

可通过man 7 signal查看帮助文档获取。



在标准信号中，有一些信号有3个value，第一个值通常用对alpha和sparc架构有效，中间值针对x86、arm和其他架构，最后一个对应mips架构。一个’-’表示在对应架构上尚未定义该信号。

不同的操作系统定义了不同的系统信号。因此有些信号出现在Unix系统内，也出现在Linux系统中，而有的信号出现在FreeBSD或Mac OS中却没有出现在Linux中。这里只研究出现在Linux系统中的信号。

**默认动作：**

Term：终止进程

Ign：忽略信号

Core：终止进程，生成Core文件（查验进程死亡原因，用于gdb调试）

Stop：停止（暂停）进程

Cont：继续运行进程

这里特别强调 **9) SIGKILL和19) SIGSTOP信号，不允许忽略和捕捉，只能执行默认动作。甚至不能将其设置为阻塞。**

**另外需清楚，只有每个信号所对应的事件发生了，该信号才会被递送（但不一定递达），不应乱发信号。**

#### 1.5、Linux常规信号一览表

1. **SIGHUP**：当用户退出shell时，由该shell启动的所有进程将收到这个信号，默认动作为终止进程。
2. **SIGINT**：当用户按下<Ctrl + C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号，默认动作为终止进程。
3. **SIGQUIT**：当用户按下<Ctrl + \>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号，默认动作为终止进程。
4. **SIGILL**：CPU检测到某进程执行了非法指令。默认动作为终止进程并产生core文件。
5. **SIGTRAP**：该信号由断点指令或其他trap指令产生。默认动作为终止进程并产生core文件。
6. **SIGABRT**：调用abort函数时产生该信号。默认动作为终止进程并产生core文件。
7. **SIGBUS**：非法访问内存地址，包括内存对齐出错，默认动作为终止进程并产生core文件。
8. **SIGFPE**：在发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为0等所有的算法错误。默认动作为终止进程并产生core文件。
9. **SIGKILL**：无条件终止进程。该信号不能被忽略、处理和阻塞。默认动作为终止进程。它向系统管理员提供可以杀死任何进程的方法。
10. **SIGUSR1**：用户定义的信号。即程序员可以在程序中定义并使用该新信号。默认动作为终止进程。
11. **SIGSEGV**：指示进程进行了无效内存访问（段错误信号）。默认动作为终止进程并产生core文件。
12. **SIGUSR2**：另外一个用户自定义信号，程序员可以在程序中定义并使用该新信号。默认动作为终止进程。
13. **SIGPIPE**：Broken pipe向一个没有读端的管道写数据。默认动作为终止进程。
14. **SIGALRM**：定时器超时，超时的时间由系统调用alarm设置。默认动作为终止进程。
15. **SIGTERM**：程序结束信号，与SIGKILL不同的是，该信号可以被阻塞和终止。通常用来表示程序正常退出。执行shell命令kill时，缺省产生该信号。默认动作为终止进程。
16. **SIGSTKFLT**：Linux早期版本出现的信号，现仍保留向后兼容。默认动作为终止进程。
17. **SIGCHLD**：子进程状态发生变化时，父进程会收到这个信号。默认动作为**忽略**这个信号。
18. **SIGCONT**：如果进程已停止，则使其继续运行。默认动作为继续/忽略。
19. **SIGSTOP**：停止程序的执行。信号不能被忽略、处理和阻塞。默认动作为暂停进程。
20. **SIGTSTP**：停止终端交互进程的运行。按下<Ctrl + Z>组合键时发出该信号。默认动作为暂停进程。
21. **SIGTTIN**：后台进程读终端控制台。默认动作为暂停进程。
22. **SIGTTOU**：该信号类似于SIGTTIN，在后台进程要向终端输出数据时发生。默认动作为暂停进程。
23. **SIGURG**：套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达，如网络带外数据到达。默认动作为忽略该信号。
24. **SIGXCPU**：进程执行时间超过了分配给该进程的CPU时间，系统产生该信号并发送给该进程。默认动作为终止进程。
25. **SIGXFSZ**：超过文件的最大长度设置。默认动作为终止进程。
26. **SIGVTALRM**：虚拟时钟超时时产生该信号。类似于SIGALRM，但是该信号只计算进程占用CPU的使用时间。默认动作为终止进程。
27. **SIGPROF**：类似于SIGVTALRM，它不仅包括该进程占用CPU时间还包括执行系统调用时间。默认动作为终止进程。
28. **SIGWINCH**：窗口变化大小时发出。默认动作为忽略该信号。
29. **SIGIO**：此信号向进程指示发出了一个异步IO事件。默认动作为忽略。
30. **SIGPWR**：关机。默认动作为终止进程。
31. **SIGSYS**：无效的系统调用。默认动作为终止进程并产生core文件。
32. **SIGRTMIN** ~ 64) **SIGRTMAX**：Linux的实时信号，它们没有固定的含义（可以由用户自己定义）。所有实时信号的默认动作都为终止进程。

### 2、信号的产生

#### 2.1、终端按键产生信号

Ctrl + C ——> 2) SIGINT（终止） “INT”——Interrupt

Ctrl + Z ——> 20) SIGTSTP（暂停） “T”——Terminal

Ctrl + \ ——> 3) SIGQUIT（退出）

#### 2.2、硬件异常产生信号

除0操作 ——>8) SIGFPE（浮点数除外） “F”——float

非法访问内存 ——> 11) SIGSEGV（段错误）

总线错误 ——> 7) SIGBUS

#### 2.3、kill函数/命令产生信号

kill命令产生信号：kill -SIGKILL pid

kill函数：给指定进程发送指定信号（不一定杀死），头文件<signal.h>



成功：0；失败：-1（ID非法、信号非法、普通用户杀init进程等权级问题），设置errno

**sig**：不推荐直接使用数字，应使用宏名，因为不同操作系统信号编号可能不同，但名称一致

**pid**：> 0：发送信号给指定进程

= 0：发送信号给与调用kill函数进程属于同一进程组的所有进程

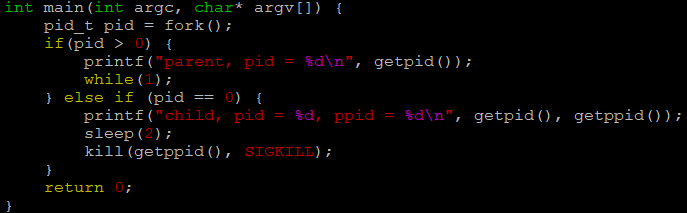
< -1：取|pid|发送给对应进程组

= -1：发送给进程有权限发送的系统中所有进程

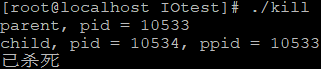
**进程组**：每个进程都属于一个进程组，进程组是一个或多个进程集合，它们相互关联，共同完成一个实体任务，每个进程组都有一个进程组长，默认进程组ID与进程组长ID相同。

**权限保护**：super用户（root）可以发送信号给任意用户，普通用户是不能向系统用户发送信号的。kill -9（root用户的pid）是不可以的。同样，普通用户也不能向其他普通用户发送信号，终止其进程。只能向自己创建的进程发送信号。普通用户基本规则是：发送者实际或有效用户ID == 接收者实际或有效用户ID。

**案例：kill函数**



运行程序



#### 2.4、raise和abort函数

raise 函数：给当前进程发送指定信号(自己给自己发) raise(signo) == kill(getpid(), signo);

int raise(int sig); 成功：0，失败非0值

abort 函数：给自己发送异常终止信号 6) SIGABRT 信号，终止并产生core文件

void abort(void); 该函数无返回

#### 2.5、软件条件产生信号

##### alarm函数

设置定时器（闹钟）。在指定seconds后，内核会给当前进程发送14) SIGALRM信号。进程收到该信号，默认动作终止。

**每个进程都有且只有唯一一个定时器。**



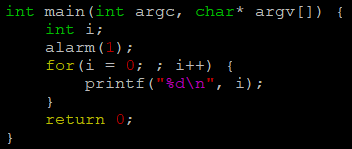
返回0或剩余的秒数，无失败。

常用：取消定时器alarm(0)，返回旧时钟余下秒数。

例：alarm(5) —> 3sec —> alarm(4) —> 5sec —> alarm(5) —> alarm(0)

定时，与进程状态无关（自然定时法）！就绪、运行、挂起（阻塞、暂停）、终止、僵尸…无论进程处于何种状态，alarm都计时。

**案例：使用alarm函数测试你的计算机一秒钟能数多少个数**

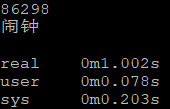


运行程序



使用**time命令**查看程序执行的时间。程序运行的瓶颈在于IO，优化程序，首选优化IO。

**实际执行时间 = 系统时间 + 用户时间 + 等待时间**



##### setitimer函数

设置定时器（闹钟）。可替代alarm函数。精度微秒μs，可以实现周期定时。



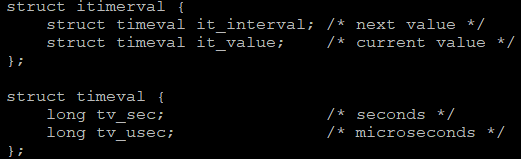
成功：返回0；失败：返回-1，设置errno

参数：which：指定定时方式

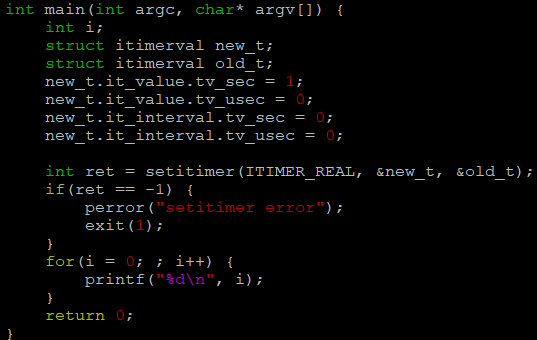
1. 自然定时：ITIMER\_REAL —> 14) SIGLARM 计算自然时间
2. 虚拟空间计时（用户空间）：ITIMER\_VIRTUAL —> 26) SIGVTALRM 只计算进程占用CPU的时间
3. 运行时计时（用户+内核）：ITIMER\_PROF —> 27) SIGPROF 计算占用CPU及执行系统调用的时间

new\_value：定时秒数

old\_value：旧时钟余下秒数



**案例：使用setitimer函数实现alarm函数测试你的计算机一秒钟能数多少个数**

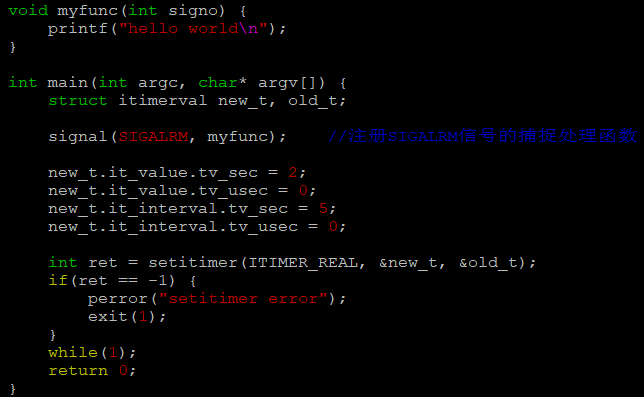


**案例：结合man page编写程序，测试it\_interval、it\_value这两个参数的作用**

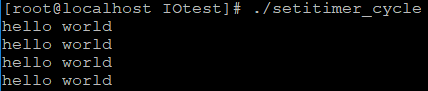
提示：it\_interval：用来设定两次定时任务之间间隔的时间

it\_value：定时的时长

两个参数都设置为0，即清零操作。

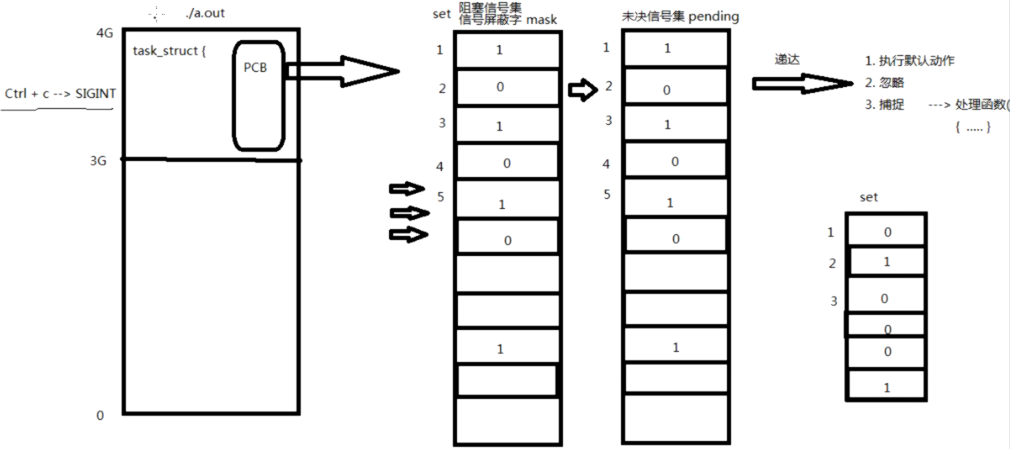


先设置it\_value为2秒，即定时2s，到了之后会打印”hello world”，设置it\_interval为5秒，即实现周期定时5s，会每隔5s，打印一次”hello world”。



### 3、信号集操作函数

内核通过读取未决信号集来判断信号是否应被处理。信号屏蔽字mask可以影响未决信号集。而我们可以在应用程序中自定义set来改变mask，以达到屏蔽指定信号的目的。



#### 3.1、信号集设定



将某个信号集清零，成功：0，失败：-1



将某个信号集置1，成功：0，失败：-1



将某个信号加入信号集，成功：0，失败：-1



将某个信号清出信号集，成功：0，失败：-1



判断某个信号是否在信号集中，返回值：在集合：1；不在：0；出错：-1

sigset\_t类型的本质是位图。但不应该直接使用位操作，而应该使用上述函数，保证跨系统操作有效。

#### 3.2、sigprocmask函数

用来**屏蔽信号**，**解除屏蔽**也可使用该函数。其本质，读取或修改进程的信号屏蔽字（PCB）。

**严格注意，屏蔽信号：只是将信号处理延后执行（延至解除屏蔽）；而忽略表示将信号丢弃处理。SIGKILL信号无法被屏蔽。**



成功：0；失败：-1，设置errno

参数：

set：传入参数，是一个位图，set中哪里为1，就表示当前进程屏蔽哪个信号。

oldset：传出参数，保存旧的信号屏蔽集

how：参数取值：假设当前的信号屏蔽字为mask

1. SIG\_BLOCK：当how设置为此值时，set表示需要屏蔽的信号。相当于mask = mask | set。
2. SIG\_UNBLOCK：当how设置为此值时，set表示需要解除屏蔽的信号。相当于mask = mask & ~set
3. SIG\_SETMASK：当how设置为此值时，set表示用于替代原始屏蔽集的新屏蔽集。相当于mask = set。若调用sigprocmask解除了对当前若干信号的阻塞，则在sigprocmask返回前，至少将其中一个信号递达。

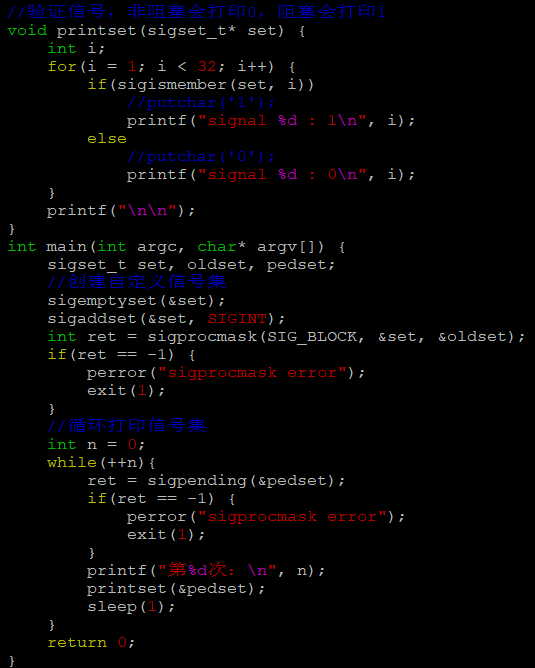
#### 3.3、sigpending函数

**读取**当前进程的未决信号集，set为传出参数

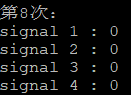


返回值：成功：0；失败：-1，设置errno

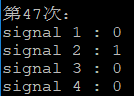
**案例：将SIGINT（Ctrl + C）设置为阻塞**



运行程序



按下Ctrl + C之后



之后再按Ctrl + C并不会终止进程，表明Ctrl + C信号被屏蔽，一开始Ctrl + C信号并未被屏蔽，是因为SIGINT信号已经被加入信号屏蔽字中，但由于没有触发SIGINT信号，所以未决信号集里并没有SIGINT。只有当按下Ctrl + C后才能在未决信号集里看到它。

### 4、信号捕捉

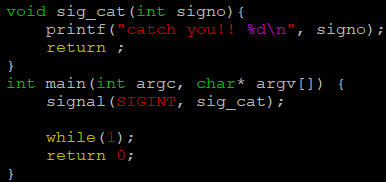
#### 4.1、signal函数

注册一个信号捕捉函数：



该函数由ANSI定义，由于历史原因在不同版本的Unix和不同版本的Linux中可能有不同的行为。因此应该尽量避免使用它，取而代之使用sigaction函数。

**案例：使用signal捕捉SIGINT信号**



执行程序



按下Ctrl + C键后，捕捉成功！！

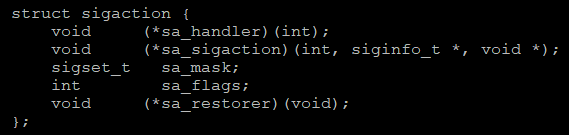


#### 4.2、sigaction函数

修改信号处理动作（通常在Linux用其来注册一个信号捕捉函数）



##### struct sigaction结构体

****

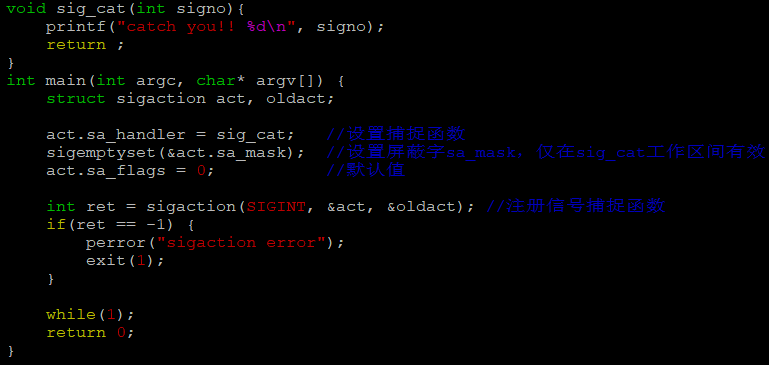
**s**a\_restorer：该元素是过时的，不应该使用，POSIX.1标准将不指定该元素。（弃用）

sa\_sigaction：当sa\_flags被指定为SA\_SIGINFO标志时，使用该信号处理程序。（很少使用）

**重点掌握：**

1. sa\_handler：指定信号捕捉后的处理函数名（即回调函数），也可赋值为SIG\_IGN表忽略或SIG\_DFL表执行默认动作。
2. sa\_mask：调用信号处理函数时，所要屏蔽的信号集合（信号屏蔽字）。注意：仅在处理函数被调用期间屏蔽生效，是临时性设置。
3. sa\_flags：通常设置为0，表使用默认属性。

**案例：使用sigaction捕捉SIGINT信号**



执行程序



按下Ctrl + C键后，捕捉成功！！



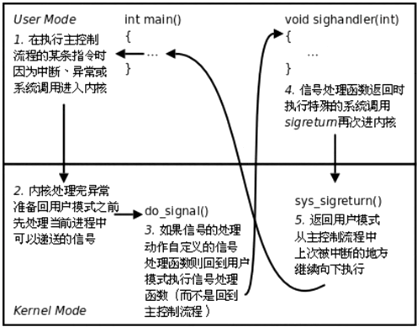
#### 4.3、信号捕捉特性

1. 进程正常运行时，默认PCB中有一个信号屏蔽字，假定为☆，它决定了进程自动屏蔽哪些信号。当注册了某个信号捕捉函数，捕捉到该信号后，要调用该函数。而该函数有可能执行很长时间，在这期间所屏蔽的信号不由☆来指定。而是由sa\_mask来指定。调用完信号处理函数，再恢复为☆。

2. xxx信号捕捉函数执行期间，xxx信号自动被屏蔽。

3. **阻塞的常规信号不支持排队，产生多次只记录一次**。（后32个实时信号支持排队）

#### 4.4、内核实现信号捕捉过程



### 5、SIGCHLD信号

#### 5.1、SIGCHLD的产生条件

子进程终止时

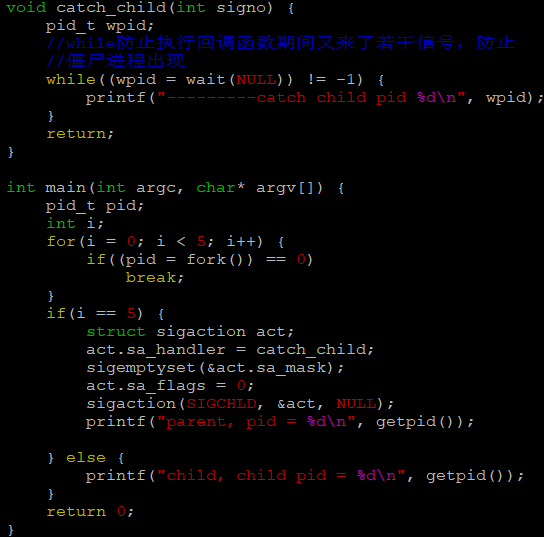
子进程接收到SIGSTOP信号停止时

子进程处于停止状态，接收到SIGCONT后唤醒

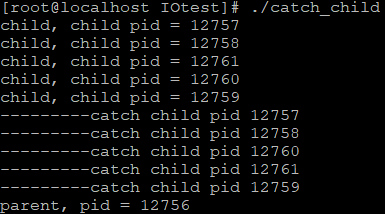
#### 5.2、借助SIGCHLD信号回收子进程

子进程结束运行，其父进程会收到SIGCHLD信号。该信号的默认处理动作是忽略。可以捕捉该信号，在捕捉函数中完成子进程状态的回收。

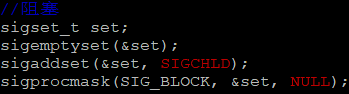
**案例：借助SIGCHLD信号捕捉回收子进程**



执行程序



存在一个问题，有可能在信号捕捉函数注册前，有的子进程已经递达成为僵尸进程，解决这个问题的方法，**一是**让子进程sleep(1)，从而保证父进程先执行捕捉函数注册；**二是**在注册前设置信号屏蔽字，使SIGCHLD无法送达，等到注册完成再解除屏蔽，具体实施是在for循环创建子进程前设置屏蔽，然后在注册信号捕捉函数之后立即解除屏蔽。





### 6、中断系统调用

系统调用可分为两类：慢速系统调用和其他系统调用。

1. 慢速系统调用：可能会使进程永远阻塞的一类。如果在阻塞期间收到一个信号，该系统调用就被中断，不再继续执行（早期）；也可以设定系统调用是否重启。如：read、write、pause、wait…
2. 其他系统调用：getpid、getppid、fork…

结合pause，回顾慢速系统调用：

慢速系统调用被中断的相关行为，实际上就是pause的行为：如：read

1. 想中断pause，信号不能被屏蔽
2. 信号的处理方式必须是捕捉（默认、忽略都不可以）
3. 中断后返回-1，设置errno为ENTER（表“被信号中断”）

可修改sa\_flags参数来设置被信号中断后系统调用是否重启。SA\_INTERRURT不重启。SA\_RESTART重启。

**扩展了解：**

sa\_flags还有很多可选参数，适用于不同情况。如：捕捉到信号后，在执行捕捉函数期间，不希望自动阻塞该信号，可将sa\_flags设置为SA\_NODEFER，除非sa\_mask中包含该信号。

## 十、守护进程

### 1、进程组和会话

#### 1.1、进程组

##### 概念和特性

进程组，也称之为**作业**。BSD于1980年前后向Unix中增加的一个新特性。代表一个或多个进程的集合。每个进程都属于一个进程组。在waitpid函数和kill函数的参数中都曾使用到。操作系统设计的进程组的概念，是为了简化对多个进程的管理。

当父进程创建子进程的时候，默认子进程与父进程属于同一进程组。进程组ID = 第一个进程ID（组长进程）。所以，组长进程标识：其进程组ID == 其进程ID。

可以使用kill -SIGKILL -进程组ID 来将整个进程组内的进程全部杀死。

组长进程可以创建一个进程组，创建该进程组中的进程，然后终止。只要进程组有一个进程存在，进程组就存在，与组长进程是否终止无关。

进程组生存期：进程组创建到最后一个进程离开（终止或转移到另一个进程）。

一个进程可以为自己或子进程设置进程组ID。

##### getpgrp函数

获取当前进程的进程组ID



总是返回调用者的进程组ID

##### getpgid函数

获取指定进程的进程组ID



成功：0；失败：-1，设置errno

如果pid = 0，那么该函数作用和getpgrp一样。

##### setpgid函数

改变进程默认所属的进程组。通常可用来加入一个现有的进程组或创建一个新进程组。



成功：0；失败：-1，设置errno

将参1对应的进程，加入参2对应的进程组中。

**注意**：

1. 如改变子进程为新的组，应fork后，exec前。

2. 权级问题。非root进程只能改变自己创建的子进程，或有权限操作的进程

#### 1.2、会话

##### 创建会话

创建一个会话需要注意以下6点注意事项：

1. 调用进程不能是进程组组长，该进程变成新会话首进程（session header）
2. 该进程成为一个新进程组的组长进程
3. 需要root权限（ubuntu不需要）
4. 新会话丢弃原有的控制终端，该会话没有控制终端
5. 该调用进程是组长进程，则出错返回
6. 建立新会话时，先调用fork，父进程终止，子进程调用setsid()（父进程是组长）

##### getsid函数

获取进程所属的会话ID



成功：返回调用进程的会话ID；失败：-1，设置errno

pid为0表示查看当前进程session ID

ps ajx命令查看系统中的进程。参数a表示不仅列出当前用户的进程，也列出所有其他用户的进程，参数x表示不仅列出有控制终端的进程，也列出所有无控制终端的进程，参数j表示列出与作业控制相关的信息。

组长进程不能成为新会话的首进程，新会话首进程必定会成为组长进程。

##### setsid函数

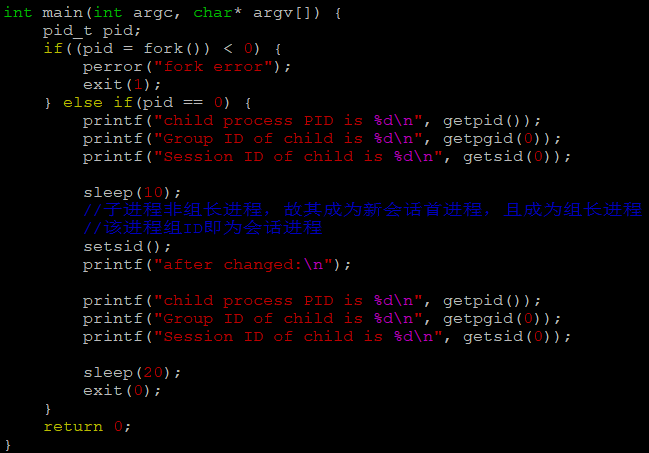
创建一个会话，并以自己的ID设置进程组ID，同时也是新会话的ID



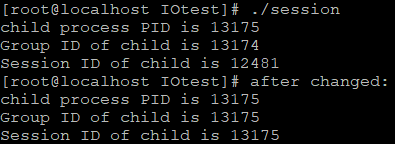
成功：返回调用进程的会话ID；失败：返回-1，设置errno

调用了setsid函数的进程，既是新的会长，也是新的组长。

**案例：fork一个子进程，并使其创建一个新会话。查看进程组ID、会话ID前后变化**



运行程序



### 2、守护进程

Daemon(精灵)进程，是Linux中的后台服务进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。一般采用d结尾的名字。比如httpd、sshd、nfsd等。

Linux后台的一些系统服务进程，没有控制终端，不能直接与用户交互。不受用户登录、注销的影响，一直在运行着，它们都是守护进程。如：预读入缓输出机制的实现；ftp服务器；nfs服务器等。

创建守护进程，**最关键**的一步是调用setsid函数创建一个新的Session，并成为Session Leader。

#### 2.1、创建守护进程模型

1. 创建子进程，父进程退出

所有工作在子进程中进行，形式上脱离了控制终端

2. 在子进程中创建新会话

setsid()函数

使子进程完全独立出来，脱离控制

3. 改变当前目录位置

chdir函数

防止占用可卸载的文件系统

也可以换成其他路径

4. 重设文件权限掩码

umask ()函数

防止继承的文件创建屏蔽字拒绝某些权限

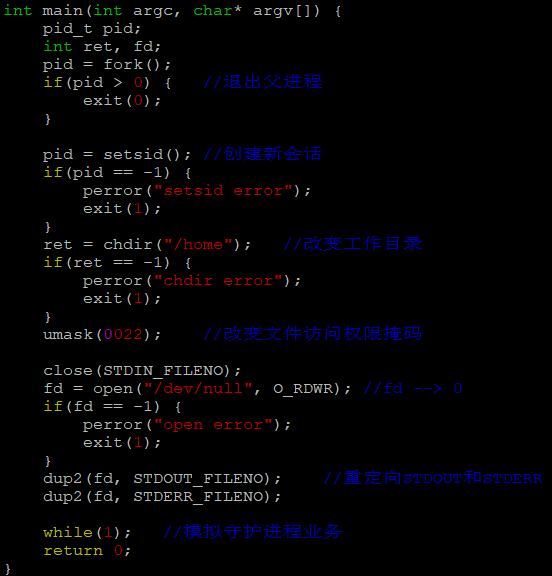
增加守护进程灵活性

5. 关闭文件描述符（0、1、2，一般是不关闭，而是重定向）

继承的打开文件不会用到，浪费系统资源，无法卸载

6. 开始执行守护进程核心工作守护进程退出处理程序模型

**案例：创建守护进程**



运行程序



程序中含有模拟守护进程业务，但是终端是没有被占用的，因为守护进程属于后台进程，使用ps ajx查看进程

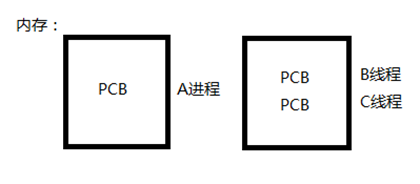


问号表示没有终端，只能使用kill命令结束进程

## 十一、线程

### 1、线程概念

#### 1.1、什么是线程

LWP：light weight process 轻量级的进程，本质仍是进程(在Linux环境下)

进程：独立地址空间，拥有PCB

线程：也有PCB，但没有独立的地址空间(共享)

区别：在于是否共享地址空间。 独居(进程)；合租(线程)。

Linux下： 线程：最小的执行单位

进程：最小分配资源单位，可看成是只有一个线程的进程。

#### 1.2、Linux内核线程实现原理

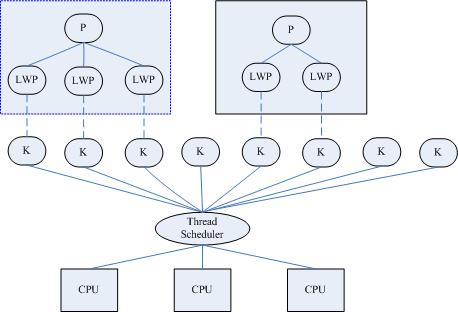
类Unix系统中，早期是没有“线程”概念的，80年代才引入，借助进程机制实现出了线程的概念，因此在这类系统中，进程和线程关系密切。

**进程**：有独立的进程地址空间，有独立的PCB

**线程**：没有独立的进程地址空间，有独立的PCB

1. 轻量级进程，也有PCB，创建线程使用的底层函数和进程一样，都是clone
2. 从内核里看进程和线程是一样的，都有各自不同的PCB，但是PCB中指向内存资源的三级页表是相同的
3. 进程可以蜕变成线程
4. 线程可以看作是寄存器和栈的集合
5. 在Linux下，**线程是最小的执行单位**，**进程是最小的分配资源单位**

查看LWP号：ps -Lf pid查看指定线程的lwp号（lwp号与线程ID不是一种东西，lwp号用于交给CPU来分配调度轮片）



**三级映射：**进程PCB —> 页目录（可看成数组，首地址位于PCB中） —> 页表 —> 物理页面 —> 内存单元

对于**进程**来说，相同的地址（同一个虚拟地址）在不同的进程中，反复使用而不冲突。原因是他们虽然虚地址一样，但，页目录、页表、物理页面各不相同。相同的虚拟地址，映射到不同的物理页面内存单元，最终访问不同的物理页面。

但，**线程**不同！！两个线程具有独立的PCB，但共享同一个页目录，也就共同一个页表和物理页面。所以两个PCB共享一个地址空间。

实际上，无论是创建进程的fork还是创建线程的pthread\_create，底层实现都是调用同一个内核函数clone。

如果复制对方的地址空间，那么就产生一个“进程”；如果共享对方的地址空间，就产生一个“线程”。

因此：Linux内核是不区分进程和线程的。只在用户层面上进行区分。所以，线程所有操作函数pthread\*是库函数，而非系统调用。

#### 1.3、线程共享资源

1. 文件描述符表

2. 每种信号的处理方式

3. 当前工作目录

4. 用户ID和组ID

5. 内存地址空间（.text/.data/.bss/heap/共享库）

#### 1.4、线程非共享资源

1. 线程ID

2. 处理器现场和栈指针（内核栈）

3. 独立的栈空间（用户栈空间）

4. errno变量（是一种全局变量）

5. 信号屏蔽字

6. 调度优先级

#### 1.5、线程优缺点

**优点：**1. 提高程序并发性 2. 开销小 3. 数据通信、共享数据方便

**缺点：**1. 库函数，不稳定 2. 调试、编写困难、gdb不支持 3. 对信号支持不好

优点相对突出，缺点均不是硬伤。Linux下由于实现方法导致进程、线程差别不是很大。

### 2、线程控制原语

#### 2.1、pthread\_self函数

获取线程ID。其作用对应进程中的getpid()函数。



返回值：成功：0；失败：无

线程ID：pthread\_t类型，本质：在Linux下为无符号整数(%lu)，其他系统可能是结构体实现

线程ID是进程内部，识别标志。（两个进程间，线程ID允许相同）

【注意】：不应使用全局变量pthread\_t tid，在子进程中通过pthread\_create传出参数来获取线程ID，而应使用pthread\_self。

#### 2.2、pthread\_create函数

创建一个线程，其作用，对应进程中的fork()函数



返回值：成功：0；失败：错误号 ---Linux环境下，所有线程特点，失败均返回错误号。

pthread\_t：当前Linux中可理解为：typedef unsigned long int pthread\_t

参数1：传出参数，保存系统为我们分配好的线程ID

参数2：通常传NULL，表示使用线程默认属性。若想使用具体属性也可以修改该参数

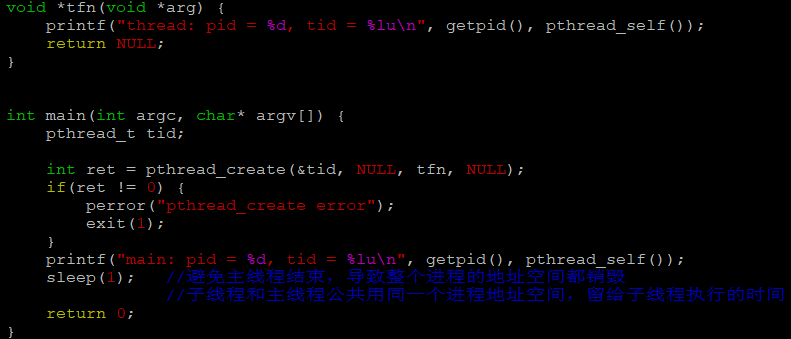
参数3：函数指针，指向线程主函数（静态函数），该函数运行结束，则线程结束

参数4：线程主函数执行期间所使用的参数

在一个线程中调用pthread\_create()创建新的线程后，当前线程从pthread\_create()返回继续往下执行，而新的线程所执行的代码由我们传给pthread\_create的函数指针start\_routine决定。start\_routine函数接收一个参数，是通过pthread\_create的arg参数传递给他的，该参数的类型为void\*，这个指针按什么类型解释由调用者自己定义。

该函数如果为类成员函数，必须为静态函数，因为普通的类成员函数、虚函数，实际上都是包含了调用他们的对象的this指针的，这就导致该函数的格式不符合pthread\_create()对线程函数的要求。

**案例1：使用pthread\_create函数创建一个子线程**

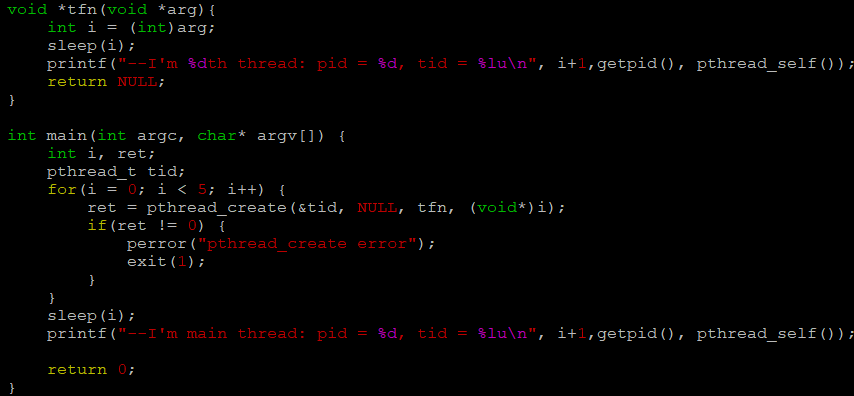


执行程序

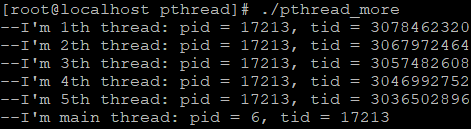


可以看出，子线程和主线程属于同一个进程，但是线程ID不同

**案例2：使用pthread\_create函数创建循环多个子线程**

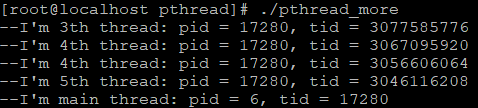


运行程序



**拓展思考**，如果将pthread\_create函数参数4修改为(void\*)&i，将线程主函数内改为 i = \*((int\*)arg)是否可以？不可以

修改后执行结果如下：



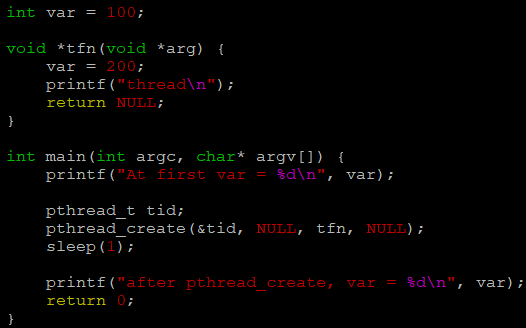
**原因**是传递的是主线程的i变量的地址，pthread\_create函数是一个用户级函数，底层会调用clone函数，clone函数是一个内核级函数，需要完成用户空间到内核空间的切换，导致子线程耗时较长，而主线程只是执行顺序逻辑。这样就存在子线程强转还未完成时i变量的值已经增加了，子线程通过地址解引用之后也就不是原来的i了。

**【注意】：创建子线程时使用值传递，然后在回调函数中使用强转。**

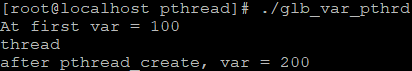
**线程间共享全局变量！**

**【牢记】**：线程默认共享数据段、代码段等地址空间，常用的是全局变量。而进程不共享全局变量，只能借助mmap。

**案例3：验证线程间共享全局变量**



运行程序



子线程与主线程共享全局变量，所以在子线程中更改全局变量的值是有效的，在主线程中打印，全局变量发生了改动，说明线程间共享全局变量。

#### 2.3、pthread\_exit函数

将单个线程退出



参数：retval表示线程退出状态，通常传NULL

**思考**：使用exit将指定线程退出，可以吗？

**结论**：线程中，禁止使用exit函数，会导致进程内所有线程全部退出。

在不添加sleep控制输出顺序的情况下，pthread\_create在循环中，几乎瞬间创建5个线程，但只有第1个线程有机会输出（或者第2个也有，也可能没有，取决于内核调度），如果第3个线程执行了exit，将整个进程退出了，所以全部线程退出了。

所以，多线程环境中，应尽量少用，或者不使用exit函数，取而代之使用pthread\_exit函数，将单个线程退出。任何线程里exit函数导致进程退出，其他线程未工作结束，主控线程退出时不能return或exit。

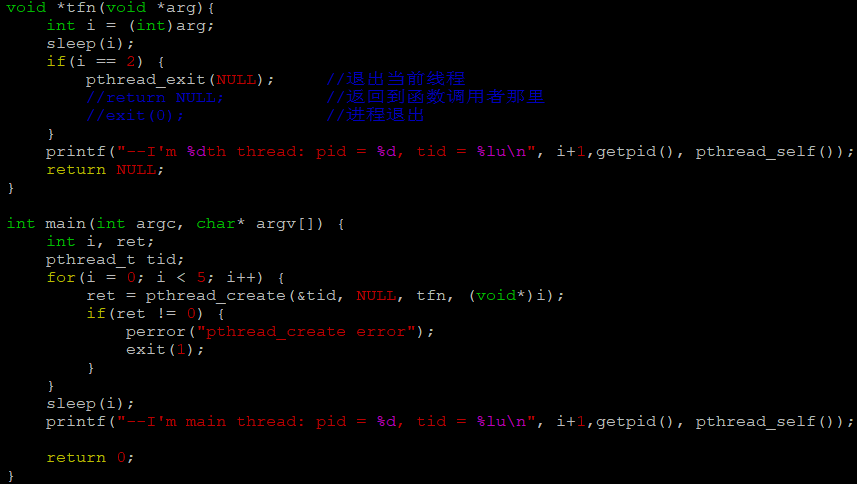
另注意，pthread\_exit或者return返回的指针所指向的内存单元必须是全局的或者是malloc分配的，不能在线程函数的栈上分配，因为其它线程得到这个返回指针时线程函数已经退出了。

**案例4：编写多线程程序，总结exit、return、pthread\_exit各自退出效果**

return、返回到函数调用者那里

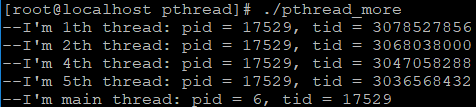
pthread\_exit：将调用该函数的线程退出

exit：将当前进程退出



程序的目的是不打印第3个子线程信息，经过验证，结果如下

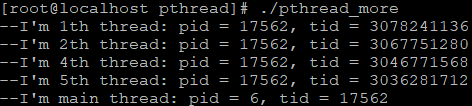
1. **return**



1. **exit**



1. **pthread\_exit**



#### 2.4、pthread\_join函数

阻塞等待线程退出，获取线程退出状态 其作用，对应进程中waitpid函数



参数：thread：线程ID（注意：不是指针）；retval：存储线程结束状态

对比记忆：

进程中：main返回值、exit参数——>int；

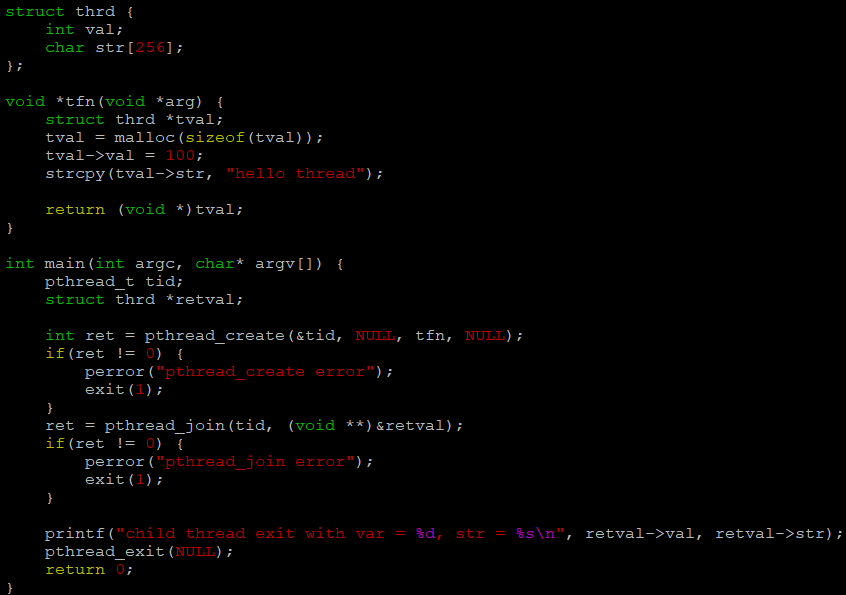
等待子进程结束wait函数参数——>int\*

线程中：线程主函数返回值、pthread\_exit——>void\*；

等待线程结束pthread\_join函数参数——>void\*\*

**案例5：主线程阻塞等待子线程退出，并获取子线程主函数的返回值**

具体操作就是，创建子线程，然后子线程调用tfn函数，在tfn函数中，返回一个结构体指针，并强转为(void \*)，然后主线程使用pthread\_join函数来回收子线程tfn函数的返回值（即结构体指针），然后打印该结构体指针指向的结构体的内容。



运行程序



#### 2.5、pthread\_cancel函数

杀死（取消）线程 其作用，对应进程中kill()函数



【注意】：线程的取消并不是实时的，而有一定的延时。需要等待线程到达某个取消点（检查点）

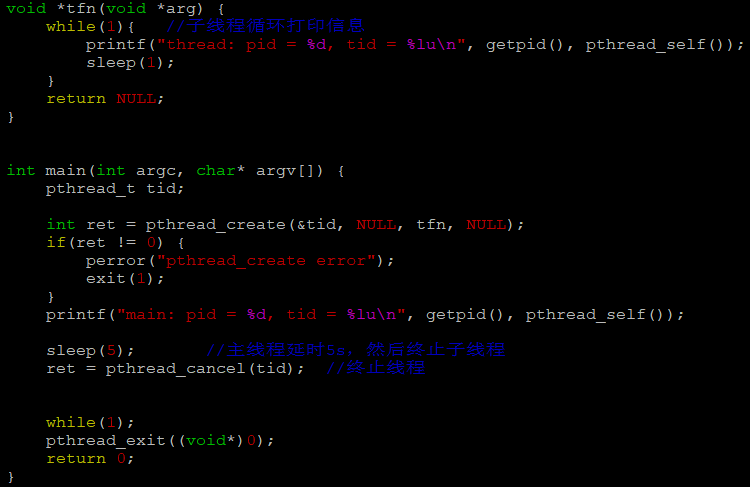
类似于玩游戏存档，必须到达指定的场所才能存储进度。杀死线程也不是立刻就能完成，必须到达取消点。

**取消点**：是线程检查是否被取消，并按请求进行动作的一个位置。通常是一些系统调用creat、open、pause、close、read、write……执行命令man 7 pthreads可以查看具备这些取消点的系统调用列表。

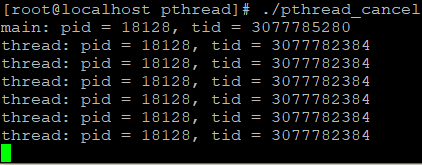
可粗略认为一个系统调用（进入内核）即为一个取消点。如线程中没有取消点，可以通过调用pthread\_testcancel函数自行设置一个取消点。

被取消的线程，退出值定义在Linux的pthread库中。常数PTHREAD\_CANCELED的值是-1。可在头文件pthread.h中找到它的定义：#define PTHREAD\_CANCELED((void \*)-1)。因此当我们**对一个已经被取消的线程使用pthread\_join回收时，得到的返回值是-1。**

**案例6：使用pthread\_cancel终止子线程**



运行程序



#### 2.6、pthread\_detach

实现线程分离



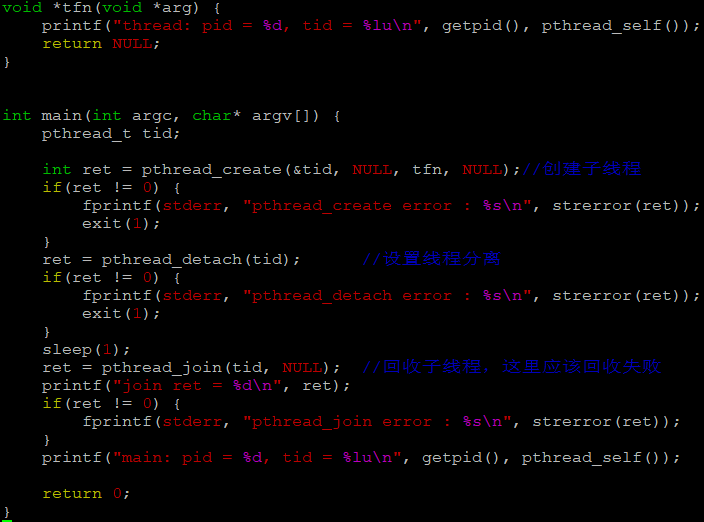
成功：0；失败：**错误号**

线程分离状态：指定该状态，线程主动与主控线程断开联系。线程结束后，其退出状态不由其他线程获取，而直接自己自动释放。网络、多线程服务器常用。

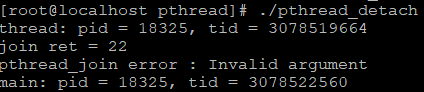
进程若有该机制，将不会产生僵尸进程。僵尸进程的产生主要由于进程死后，大部分资源被释放，一点残留资源仍存于系统中，导致内核认为该进程仍存在。

也可使用pthread\_create函数参数2（线程属性）来设置线程分离。

**案例7：使用pthread\_detach函数实现线程分离**



执行程序



可以看出pthread\_join出错，提示参数无效，原因在于线程分离后，因为sleep 1秒，所以子线程已经自动清理PCB，回收了。

一般情况下，线程终止后，其终止状态一直保留到其他线程调用pthread\_join获取它的状态为止，但是线程也可设置为detach状态，**这样的线程一旦终止就立刻回收它占用的所有资源，而不保留终止状态。**

不能对一个已经处于detach状态的线程调用pthread\_join，这样的调用将返回EINVAL错误。也就是说，如果已经对一个线程调用了pthread\_detach就不能再调用pthread\_join了。

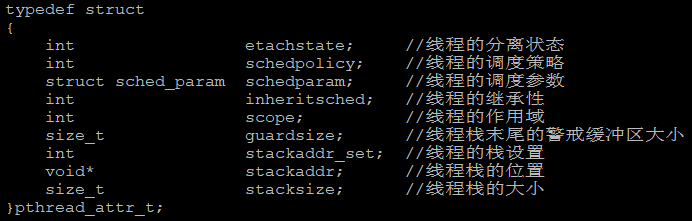
**【注意】**：检查线程出错返回不能再使用perror(“xxx error”)了，因为线程的相关函数出错的话返回的不再是-1，而是直接返回出错号，需要使用fprintf(stderr, “xxx error : %s\n”, strerror(ret))。

#### 2.7、线程/进程控制原语比较

|  |  |  |
| --- | --- | --- |
| 行为 | 线程控制原语 | 进程控制原语 |
| 创建 | pthread\_create() | fork() |
| 获取ID | pthread\_self() | getpid() |
| 退出 | pthread\_exit() | exit() |
| 回收 | pthread\_join() | wait()/waitpid() |
| 杀死 | pthread\_cancel() | kill() |
| 分离 | pthread\_detach() | 无 |

### 3、线程属性

Linux下线程的属性是可以根据实际项目需要，进行设置，之前我们讨论的线程都是采用线程的默认属性，默认属性已经可以解决绝大多数开发时遇到的问题。如我们对程序的性能提出更高的要求那么需要设置线程属性，比如可以设置线程栈的大小来降低内存的使用，增加最大线程个数。



主要结构体成员：

1. 线程分离状态
2. 线程栈大小（默认平均分配）
3. 线程栈警戒缓冲区大小（位于栈末尾）

属性值不能直接设置，须使用相关函数进行操作，初始化的函数为pthread\_attr\_init，这个函数必须在pthread\_create函数之前调用。之后必须用pthread\_attr\_destroy函数来释放资源。

线程属性主要包括如下属性：作用域（scope）、栈尺寸（stack size）、栈地址（stack address）、优先级（priority）、分离的状态（detached state）、调度策略和参数（scheduling policy and parames）。**默认的属性**为非绑定、非分离、缺省的对战、与父进程同样级别的优先级。

#### 3.1、线程属性初始化

注意：应先初始化线程属性，再pthread\_create创建线程

1. **初始化线程属性：**



成功：0；失败：错误号

1. **销毁线程属性所占用的资源：**



成功：0；失败：错误号

#### 3.2、线程的分离状态

线程的分离状态决定一个线程以什么样的方式来终止自己。

**非分离状态**：线程的默认属性是非分离状态，这种情况下，原有的线程等待创建的线程结束，只有当pthread\_join函数返回时，创建的线程才算终止，才能释放自己占用的系统资源。

**分离状态**：分离线程没有被其他线程所等待，自己运行结束了，线程也就终止了，马上释放系统资源，应该根据自己的需要，选择适当的分离状态。

线程分离状态的函数：

1. 设置线程属性，分离 or 不分离



1. 获取线程属性，分离 or 不分离



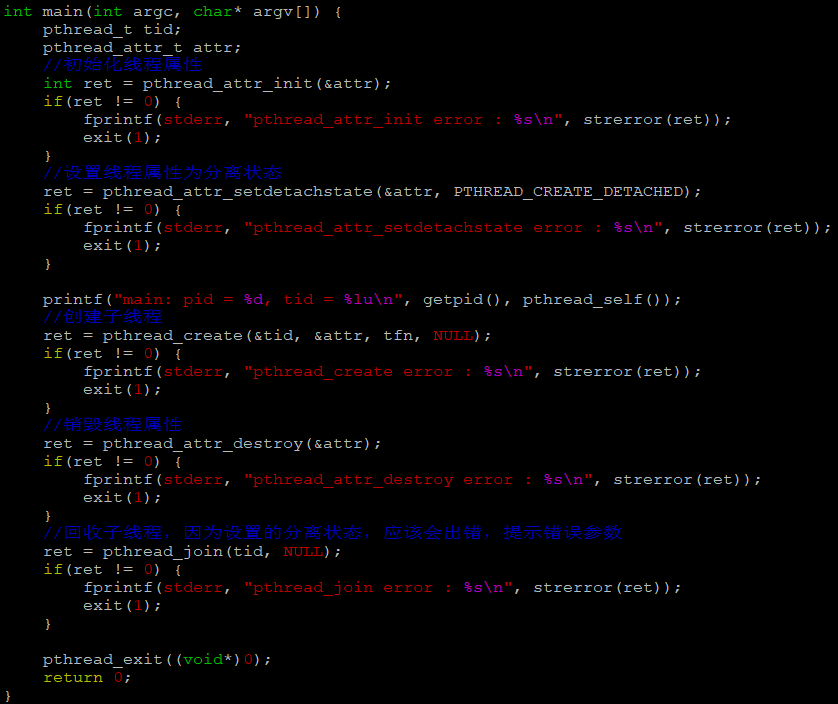
参数：attr：已初始化的线程属性

detachstate：PTHREAD\_CREATE\_DETACHED（分离线程）

PTHREAD\_CREATE\_JOINABLE（非分离线程）

这里要注意的一点是，如果设置一个线程为分离线程，而这个线程又运行的非常快，它很可能在pthread\_create函数返回之前就终止了，它终止以后就可能将线程号和系统资源移交给其他的线程使用，这样调用pthread\_create的线程就得到了错误的线程号，要避免这种情况可以采取一定的同步措施。最简单的方法之一是可以在被创建的线程里调用pthread\_cond\_timewait函数，让这个线程等待一会，留出足够的时间让函数pthread\_create返回。设置一段等待时间，是在多线程编程里常用的方法。但是注意不要使用诸如wait()之类的函数，它们是使整个进程睡眠，并不能解决线程同步的问题。

**案例8：设置线程属性**



执行程序



### 4、使用线程注意事项

1. 主线程退出其他线程不退出，主线程应调用pthread\_exit

2. 避免僵尸线程

pthread\_join

pthread\_detach

pthread\_create指定分离属性

被join线程可能在join函数返回前就释放完自己所有的内存资源，所以不应

当返回被回收线程栈中的值

1. malloc和mmap申请的内存可以被其他线程释放
2. 应避免在多线程模型中调用fork，除非马上exec，子进程中只有调用fork的线程存在，其他线程在子进程中均pthread\_exit
3. 信号的复杂语义很难和多线程共存，应避免在多线程引入信号机制

## 十二、线程同步

### 1、同步概念

所谓同步，即同时起步，协调一致。不同的对象，对“同步”的理解方式略有不同。如，设备同步，是指在两个设备之间规定一个共同的时间参考；数据库同步，是指让两个或者多个数据库内容保持一致，或者按需要部分保持一致；文件同步：是指让两个或者多个文件夹里的文件保持一致。

而编程中、通信中所说的同步与生活中大家印象中的同步概念略有差异。“同”字应是协同、协助、互相配合。主旨在协同步调，按预定的先后次序运行。

#### 1.1、线程同步

同步即协同步调，按预定的先后次序运行。

**线程同步，指一个线程发出某一功能调用时，在没有得到结果之前，该调用不返回。同时其他线程为保证数据一致性，不能调用该功能。**

举例1：银行存款5000。柜台，折：取3000；提款机，卡，取3000。剩余：2000

举例2：内存中100字节，线程T1欲填入全1，线程T2欲填入全0。如果T1执行了50个字节失去CPU，T2执行，会将T1写过的内容覆盖。当T1再次获得CPU继续，从失去CPU的位置向后写入1，当执行结束，内存中的100字节，既不是全1，也不是全0。

产生的现象叫做“与时间有关的错误”。为了避免这种数据混乱，线程需要同步。

“同步”的目的是为了避免数据混乱，解决与时间有关的错误。实际上，不仅线程间需要同步，进程间、信号间也都需要同步机制。

因此，**所有“多个控制流，共同操作一个共享资源”的情况，都需要同步机制。**

#### 1.2、数据混乱原因

1. 资源共享（独享资源则不会）
2. 调度随机（意味着数据访问会出现竞争）
3. 线程间缺乏必要的同步机制

以上3点中，前两点不能改变，欲提高效率，传递数据，资源必须共享。只要共享资源，就一定会出现竞争。只要存在竞争关系，数据就很容易出现混乱。

所以只能从第三点着手解决，使多个线程在访问共享资源的时候，出现互斥。

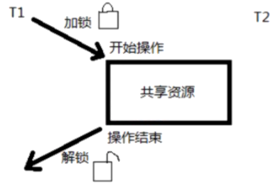
### 2、互斥量mutex

Linux中提供一把互斥锁mutex（也称之为互斥量）。

每个线程在对资源操作前都尝试先加锁，成功加锁才能操作，操作结束解锁。

资源还是共享的，线程间也还是竞争的。

但通过“锁”就将资源的访问变成互斥操作，而后与时间有关的错误也不会再产生了。



但，应注意：同一时刻，只能在一个线程持有该锁。

当A线程对某个全局变量加锁访问，B在访问前尝试加锁，拿不到锁，B阻塞。C线程不去加锁，而直接访问该全局变量，依然能够访问，但会出现数据混乱。

所以，互斥锁实质上使操作系统提供的一把“**建议锁**”（又称“协同锁”），建议程序中有多线程访问共享的时候使用该机制。但，并没有强制限定。所有线程【应该】在访问时先拿到锁再访问。

因此，即使有了mutex，如果有线程不按规则来访问数据，依然会造成数据混乱。

#### 2.1、主要应用函数

pthread\_mutex\_init函数

pthread\_mutex\_destory函数

pthread\_mutex\_lock函数

pthread\_mutex\_trylock函数

pthread\_mutex\_unlock函数

以上5个函数的返回值都是：成功：0，失败：错误号

pthread\_mutex\_t类型，其本质是一个结构体。为简化理解，应用时可忽略其实现细节，**简单当成整数看待**。

pthread\_mutex\_t mutex变量mutex只有0，1两种取值

**使用mutex（互斥量、互斥锁）一般步骤：**

1. pthread\_mutex\_t lock; 创建锁
2. pthread\_mutex\_init; 初始化
3. pthread\_mutex\_lock; 加锁
4. 访问共享数据
5. pthread\_mutex\_unlock; 解锁
6. pthread\_mutex\_destroy; 销毁锁

【注意】：尽量保证锁的粒度，越小越好。（访问数据前，加锁。访问结束，**立即**解锁）

##### pthread\_mutex\_init函数

初始化一个互斥锁（互斥量）——>初值可在看作1



参1：互斥锁 参2：锁属性

【注意】，参1中的restrict是一个关键字，用来限定指针变量。被该关键字限定的指针变量所指向的内存操作，必须由本指针完成。

也可以使用**静态初始化** pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER

##### pthread\_mutex\_destory函数

销毁一个互斥锁



##### pthread\_mutex\_lock函数

加锁



##### pthread\_mutex\_unlock函数

解锁



##### pthread\_mutex\_trylock函数

尝试加锁，成功就加锁，失败则返回错误号



#### 2.2、加锁与解锁

##### lock与unlock

lock尝试加锁，如果加锁不成功，线程阻塞，阻塞到持有该互斥量的其它线程解锁为止。

unlock主动解锁函数，同时将阻塞在该锁上的所有线程全部唤醒，至于哪个线程先被唤醒，取决于优先级、调度。默认：先阻塞、先唤醒。

例如：T1 T2 T3 T4使用一把mutex锁，T1加锁成功，其他线程均阻塞，直至T1解锁。T1解锁后，T2 T3 T4均被唤醒，并自动再次尝试加锁。

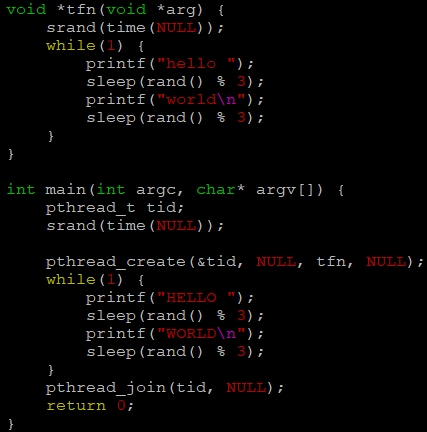
可假想mutex锁init成功初值为1。Lock功能是将mutex--。而unlock则将mutex++。

##### lock与trylock

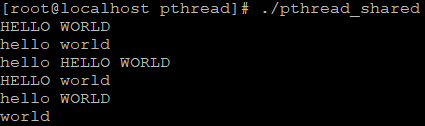
lock加锁失败会阻塞，等待锁被释放

trylock加锁失败直接返回错误号（如：EBUSY），不阻塞

**案例：父进程打印大写的HELLO WORLD，子进程打印小写的hello world**



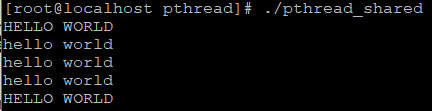
运行程序



因为主子线程中均有延迟操作，此时会让出CPU，被另一线程占用，两个线程抢占共享资源STDOUT，导致并没有如预期一样。**解决办法**就是加上互斥锁。



运行程序



### 3、死锁

**死锁**是使用锁不恰当导致的现象。

1. 线程试图对一个互斥量S加锁两次

2. 线程1拥有A锁，请求获得B锁；线程2拥有B锁，请求获得A锁

### 4、读写锁

与互斥量类似，但读写锁允许更高的并行性。其特性为：**写独占，读共享**。

#### 4.1、读写锁状态

特别强调，读写锁**只有一把**，但其具备两种状态：

1. 读模式下加锁状态（读锁）
2. 写模式下加锁状态（写锁）

#### 4.2、读写锁特性

1. 读写锁是“写模式加锁”时，解锁前，所有对该锁加锁的线程都会被阻塞。

2. 读写锁是“读模式加锁”时，如果线程以读模式对其加锁会成功；如果线程以写模式对其加锁会阻塞。

3. 读写锁是“读模式加锁”时，既有试图以写模式加锁的线程，也有试图以读模式加锁的线程。那么读写锁会阻塞随后的读模式锁请求，优先满足写模式锁。**读锁、写锁并行阻塞，写锁优先级高。**

读写锁也叫共享独占锁。当读写锁以读模式锁住时，它是以共享模式锁住的；当它以写模式锁住时，它是以独占模式锁住的。**写独占、读共享**。

读写锁非常适合于对数据结构读的次数远大于写的情况。

#### 4.3、主要应用函数

pthread\_rwlock\_init函数

pthread\_ rwlock \_destory函数

pthread\_ rwlock \_rdlock函数

pthread\_ rwlock \_tryrdlock函数

pthread\_ rwlock \_wrlock函数

pthread\_ rwlock \_trywrlock函数

pthread\_ rwlock \_unlock函数

以上7个函数的返回值都是：成功：0，失败：错误号

pthread\_ rwlock \_t类型，用于定义一个读写锁变量。

pthread\_ rwlock \_t rwlock

##### pthread\_rwlock\_init函数

初始化一把读写锁



参1：读写锁 参2：锁属性

##### pthread\_ rwlock \_destory函数

销毁一把读写锁



##### pthread\_ rwlock \_rdlock函数

读模式加锁



##### pthread\_ rwlock \_tryrdlock函数

尝试读模式加锁，成功就加锁，失败则返回错误号



##### pthread\_ rwlock \_wrlock函数

写模式加锁



##### pthread\_ rwlock \_trywrlock函数

尝试写模式加锁，成功就加锁，失败则返回错误号

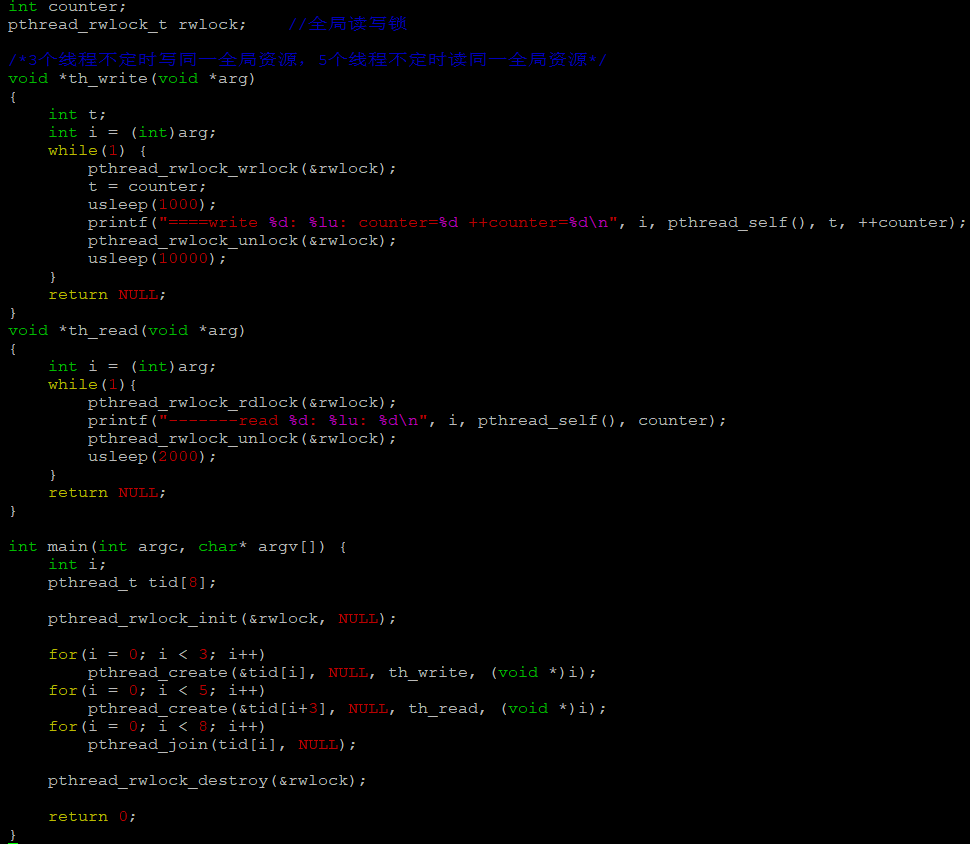


##### pthread\_ rwlock \_unlock函数

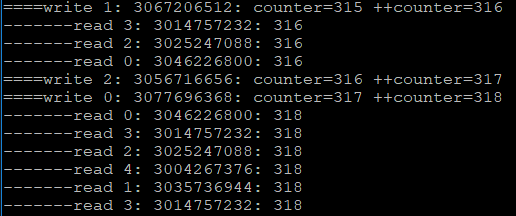
解锁



**案例：读写锁实现**



运行程序



### 5、条件变量

条件变量本身不是锁！但它可以造成线程阻塞。通常与互斥锁配合使用。给多线程提供一个会合的场所。

#### 5.1、主要应用函数

pthread\_cond\_init函数

pthread\_cond\_destroy函数

pthread\_cond\_wait函数

pthread\_cond\_timedwait函数

pthread\_cond\_signal函数

pthread\_cond\_broadcast函数

以上6个函数的返回值都是：成功：0；失败：错误号

pthread\_cond\_t类型，用于定义条件变量

pthread\_cond\_t cond

##### pthread\_cond\_init函数

初始化一个条件变量



参1：条件变量 参2：条件变量属性(默认为NULL)

也可以使用**静态初始化** pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER

##### pthread\_cond\_destroy函数

销毁条件变量



##### pthread\_cond\_wait函数

阻塞等待一个条件变量



函数作用：

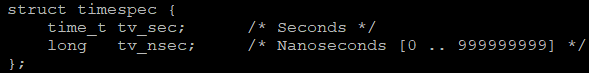
1. 阻塞等待条件变量cond满足
2. 释放已掌握的互斥锁（解锁互斥量）相当于pthread\_mutex\_unlock(&mutex)
3. 当被唤醒，pthread\_cond\_wait函数返回时，解除阻塞并重新申请获取互斥锁pthread\_mutex\_lock(&mutex)

##### pthread\_cond\_timedwait函数

限时等待一个条件变量



参3：参看man sem\_timedwait，查看timespec结构体



##### pthread\_cond\_signal函数

唤醒阻塞在条件变量上的一个线程

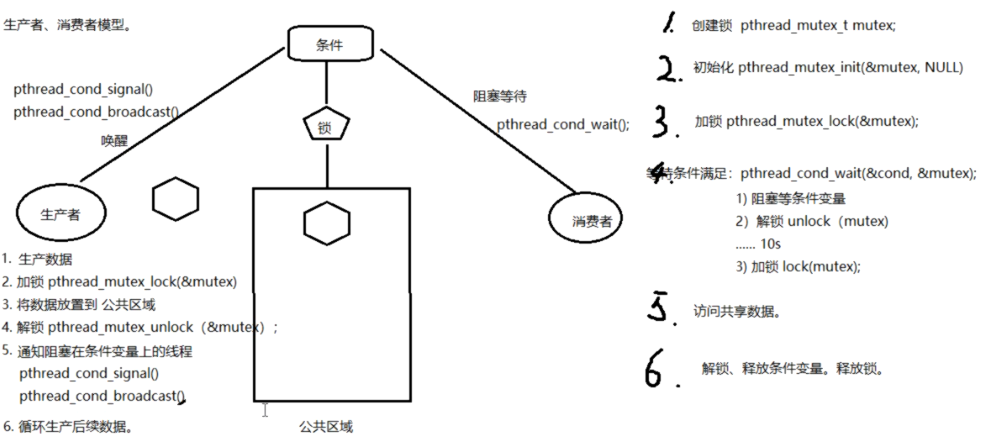


##### pthread\_cond\_broadcast函数

唤醒阻塞在条件变量上的所有线程



#### 5.2、生产者消费者模型条件变量实现



生产者生产数据，放置到公共区域，消费者从公共区域消费数据。需要对公共区域添加条件，只有条件满足时（即公共区域有数据）才允许消费者消费，条件不满足时消费者需要阻塞等待。

**消费者行为：**

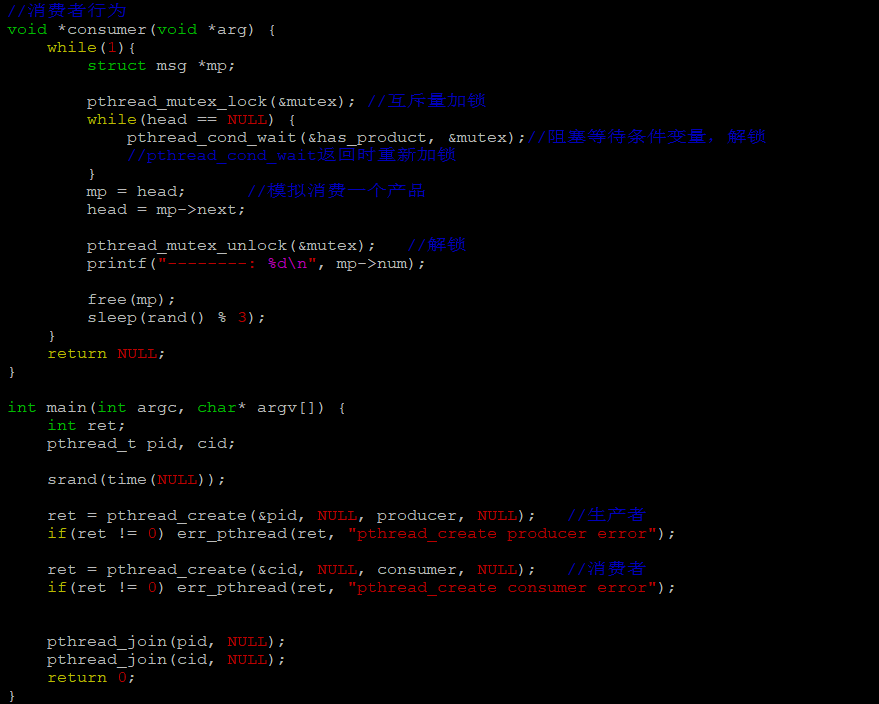
1. 加锁
2. 阻塞等待条件变量，使用pthread\_cond\_wait，会解锁，收到信号后会再次加锁
3. 收到信号后解除阻塞，访问公共数据
4. 解锁

**生产者行为：**

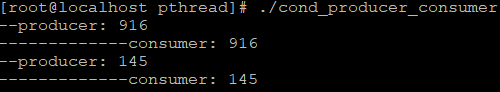
1. 本地生产数据
2. 加锁
3. 将数据写到公共区域
4. 解锁
5. 唤醒阻塞的条件变量上等待的线程即消费者
6. 循环生产后续数据

**【注意】**：公共区域无数据时消费者会阻塞等待生产者生产数据，即消费者停留在第2步，因为pthread\_cond\_wait在阻塞期间是解锁状态，所以生产者第2步可以直接加锁；然后生产者会解锁，将信号传递，此时消费者pthread\_cond\_wait函数返回时会再次加锁，所以消费者可以直接访问公共数据，并最后解锁。





运行程序



**如果有多个消费者，只需要创建多个消费者即可**。

**【注意】**：条件变量是head == NULL，在判断条件变量时使用的是while，能不能使用if？

如果只有一个消费者，是可以使用if的，如果有多个消费者就不能使用if，因为有可能有多个消费者同时在等待公共区域来数据。

#### 5.3、条件变量的优点

相较于mutex而言，条件变量可以减少竞争。

如直接使用mutex，除了生产者、消费者之间要竞争互斥量以外，消费者之间也需要竞争互斥量，但如果汇聚（链表）中没有数据，消费者之间竞争互斥锁是没有意义的。有了条件变量机制以后，只有生产者完成生产，才会引起消费者之间的竞争。提高了程序效率。

### 6、信号量

进化版的互斥锁（1——>N），相当于初始化值为N的互斥量，N表示可同时访问共享区的线程数。**既可应用于线程，也可应用于进程。**

由于互斥锁的力度比较大，如果我们希望在多个线程间对某一对象的部分数据进行共享，使用互斥锁是没有办法实现的，只能将整个数据对象锁住。这样虽然达到了多线程操作共享数据时保证数据正确性的目的，却无形中导致了线程的并发性下降。线程从并行执行，变成了串行执行。与直接使用单进程无异。

信号量，是相对折中的一种处理方式，既能保证同步，数据不混乱，又能提高线程并行。

#### 6.1、主要应用函数

sem\_init函数

sem\_destroy函数

sem\_wait函数（加锁）

sem\_trywait函数

sem\_timedwait函数

sem\_post函数（解锁）

以上6个函数的返回值都是：成功：0；失败：-1，设置errno

sem\_t类型，本质是结构体

sem\_t sem；规定信号量sem不能<0。头文件<semaphore.h>

##### 信号量基本操作

sem\_wait： 1. 信号量大于0，则信号量-- 2. 信号量等于0，则造成线程阻塞

sem\_post： 将信号量++，同时唤醒阻塞在信号量上的线程

但，由于sem\_t的实现对用户隐藏，所以所谓的++、--操作只能通过函数来实现，而不能直接++、--符号

**信号量的初值，决定了占用信号量的线程的个数。**

##### sem\_init函数

初始化一个信号量



参2：0：线程间同步；非0：进程间同步

参3：N值，指定同时访问的线程数

##### sem\_destroy函数

销毁一个信号量



##### sem\_wait函数（加锁）

类似于加锁操作，即pthread\_mutex\_lock



##### sem\_trywait函数

类似于尝试加锁操作，即pthread\_mutex\_trylock



##### sem\_timedwait函数



参2使用绝对时间（Unix创建时间1970年1月1日0时0分0秒起）

定时1秒：

time\_t cur = time(NULL); //获取当前时间

struct timespec t; //定义timespec结构体变量

t.tv\_sec = cur + 1; //定时1秒

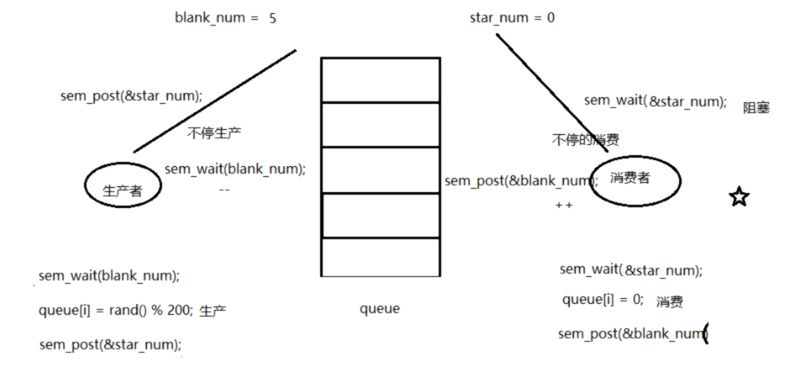
sem\_timewait(&sem, &t); //传参

##### sem\_post函数（解锁）

类似于解锁操作，即pthread\_mutex\_unlock



#### 6.2、信号量实现生产者消费者模型





运行程序

