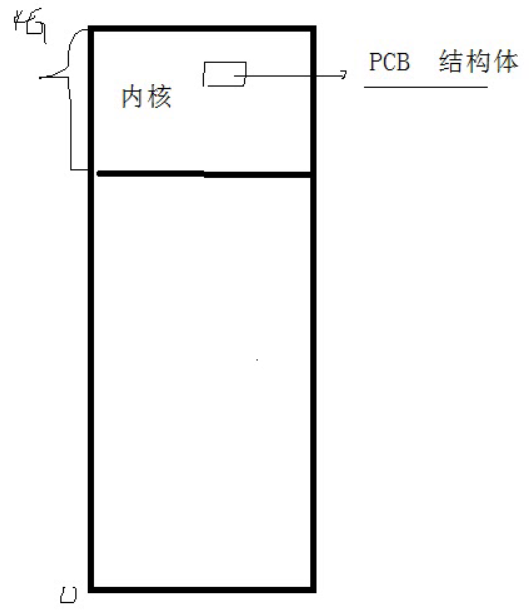
1.多进程

(1)**进程(process)：是一个程序一次执行的过程，是动态的**。 它和程序有本质区别。程序是静态的，它是一些保存在磁盘上的指令的有序集合； 而进程是一个动态概念，它是**一个运行着的程序**，包含进程的动态创建、调度和消亡的过程，**进程是Linux的基本调度单位**。进程：资源管理的基本单位；线程(带方括号的)：不负责资源管理，只负责资源使用

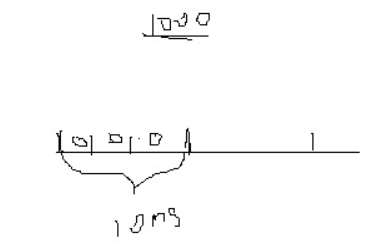
(2)进程是通过**进程控制块（PCB结构体名：task\_struct）**来描述的。进程控制块包含了进程的描述信息、控制信息以及资源信息，它是**进程某个瞬间的一个静态描述**。

当进程启动时，内核就会为这个进程申请一个PCB结构体的空间。故一个活着的进程，在内核中唯一对应一个PCB进程控制块。



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

CPU的调度周期很短：早期只有单核CPU时：就可以开个QQ、打游戏、还能去迅雷下载(相当于三个进程)，让人感觉是同时进行的，实际并没有真正的同时，就是因为CPU的一个调度周期是非常短，人类是无法感知出来的。 在一个调度周期内，会根据进程的优先级，将这1000个时间片分配给这三个进程。



(4) 进程标识号

OS会为每个进程分配一个唯一的整型ID，做为**进程的标识号(pid)**。进程除了自身的ID外，还有**父进程ID(ppid)**，pid和ppid启动起来便被决定了，不可改变的(相当于身份证号)

所有进程的祖先进程是同一个进程，它叫做**init进程，pid为1**， init进程是内核自举后的第一个启动进程。init进程负责引导系统、启动守护进程（后台）并且运行必要的程序。

1)**进程的pid和ppid可以分别通过函数 getpid() 和 getppid() 获得**：是因为PCB结构体中就存着pid和ppid的值，通过系统调用getpid()、getppid()来获得。

**内核为了安全性，不能自己定义一个指针去访问内核数据，一定要通过系统调用来访问。**

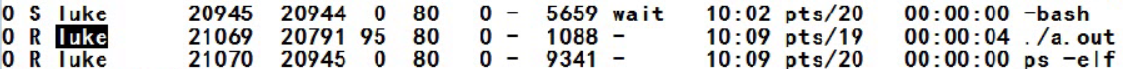
如： printf("pid:%d ppid:%d\n",getpid(),getppid());

(5)进程的 用户ID和组ID (进程的运行身份)

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

**2)真实用户ID 和 真实组ID 可以通过函数 getuid() 和 getgid() 获得**。

如图：a.out进程 就是由 luke用户(缺省情况下)启动的。



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

**3)有效用户id和有效组id通过函数 geteuid() 和 getegid() 获得**。

如：

printf("uid:%d gid:%d euid:%d egid:%d\n",getuid(),getgid(),geteuid(),getegid());

注意 **su命令**切换：切换的是真实用户ID (通过setuid()来改变真实用户ID)

4)有效用户ID可用于权限提升，如passwd、sudo的实现。

5) ps -elf 的第三列UID ：不是真实用户ID，而是进程的 有效用户ID

2.

(1) 内核只有一份，即只占用一块物理内存。

**内核只有一份，Why每个进程都有内核那**? (即每个进程地址空间最上面均为：内核区)

是因为 **把内核的这一块物理内存，分别映射到了进程的虚拟地址空间中**， 故每一个进程都看到了相同的内核。

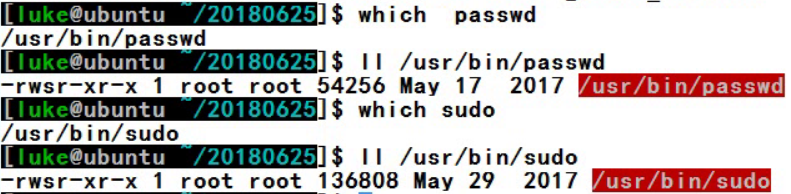
(2) 单一职责(高内聚低耦合)： 如操作系统就是由多个进程运行起来的，但大部分进程都在睡觉，当用到时再去唤醒。 Windows xp大获成功，就是因为有了WindowsNT内核(很少蓝屏)

3. 权限提升(类似sudo，改变了有效用户ID为root): 基于内核对进程的访问权限检查时，它检查的是进程的有效用户ID和有效组ID

(1)

我们知道，用户的**用户名和密码是保存在/etc/passwd**下的 (后来专门将**密码保存在/etc/shadow**，它是根据/etc/passwd文件来生成/etc/shadow的，它把所有口令从/etc/passwd中移到了/etc/shadow中。这里用到的是影子口令，它将口令文件分成两部分：/etc/passwd 和 /etc/shadow， 此时/etc/shadow就是影子口令文件，它保存的是加密的口令，而/etc/passwd中的密码全部变成x)

通过ls –l查看/etc/passwd这个文件，你会发现，这个文件普通用户都没有可写的权限，那我们执行passwd命令 的时候确实能够修改密码，那么这是怎么回事呢？也就是说，任何一个用户运行该程序时，该程序的有效身份都将是root（用普通身份去执行这个操作的时候，它会暂时得到文件拥有者root的权限）,而这样passwd程序才有权限读取/etc/passwd文件的信息: 通过权限提升。



**重点！！！**

stat结构体成员：st\_mode 用%x打印16进制数：

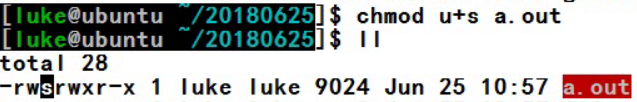
最高的4位：文件类型；

其中的低12位：权限， 而其中的**高3位：是用于权限提升**。

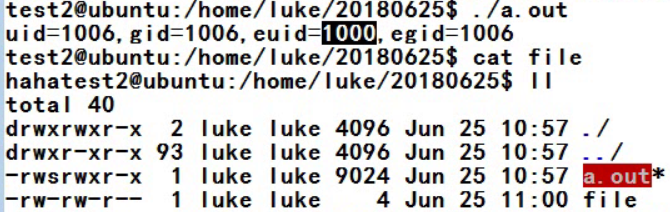
例如0775：000 111 111 101 ：

(2) 低12位中的前三位中，第一位：用户的权限提升：**chmod u+s a.out**

低12位变为了：100 111 111 101，则文件创建者权限 rw- 变为了 rws , 文件a.out的访问权限的所有者可执行权限设置了**s的属性**，设置了该属性后，用户运行a.out时,a.out进程的**有效用户ID将不再是运行a.out的用户，而是a.out文件的拥有者。**



如下图：test2用户的有效用户ID，变为了1000, 即a.out的有效用户ID：由test2用户的1006变为了luke用户的1000了 （即test2用户是借助luke身份来实现的）

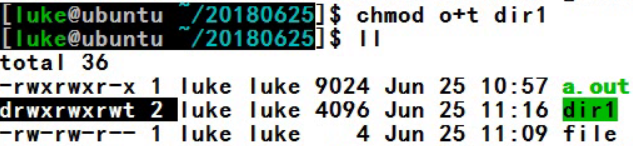


(3) 低12位中的前三位中，第二位：用户组的权限提升：**chmod g+s a.out**





(4)低12位中的前三位中，第三位：粘滞位(如 /tmp目录)：**chmod o+t dir1**



创建一个777权限的目录dir1 ,则任何用户都可以写、删除此目录下的文件，而当文件加了粘滞位后，则不可删除此文件了：加了粘滞位后，会检查文件是谁拥有的，谁拥有的谁才可以删 (相当于一个公共空间，谁都可以放东西，但是只能删除自己的东西)

(5)chown 改变文件的拥有者：

**chown luke:luke a.out** ：改变a.out为luke用户、luke组所拥有的。



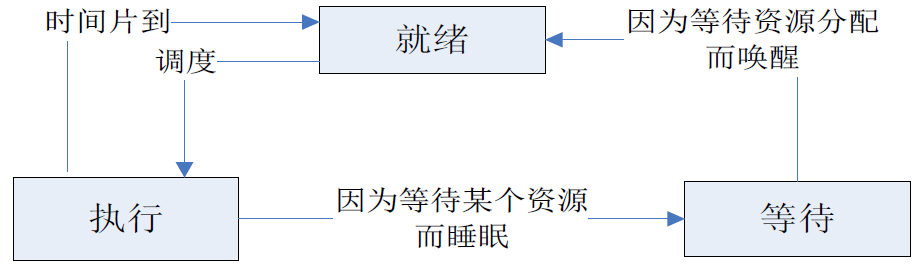
4. 进程的状态：

进程是程序的执行过程，根据它的生命周期可以划分成3种状态：

 **执行态R**：该进程正在运行，即进程正在占用CPU。

 **就绪态R**：进程已经具备执行的一切条件，正在等待分配CPU的处理时间片（就绪队列）

 **等待态S**：进程不能使用CPU，若等待事件发生（等待的资源分配到）则可将其唤醒(睡眠队列)



ps -elf查看进程状态：

**R**：正在执行、或正在就绪队列。

**S**：在睡眠队列。

**T**：暂停状态：用于gdb调试

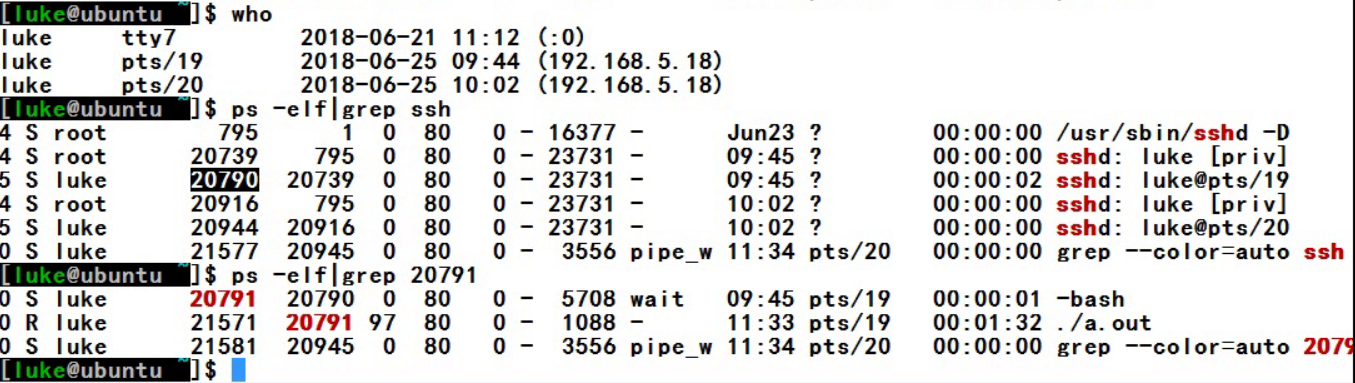
**Z**：僵尸进程: 查看ps -elf|grep Z (进程已退出，但并没有回收free其PCB结构体)

**D**：内核不可中断状态

5.

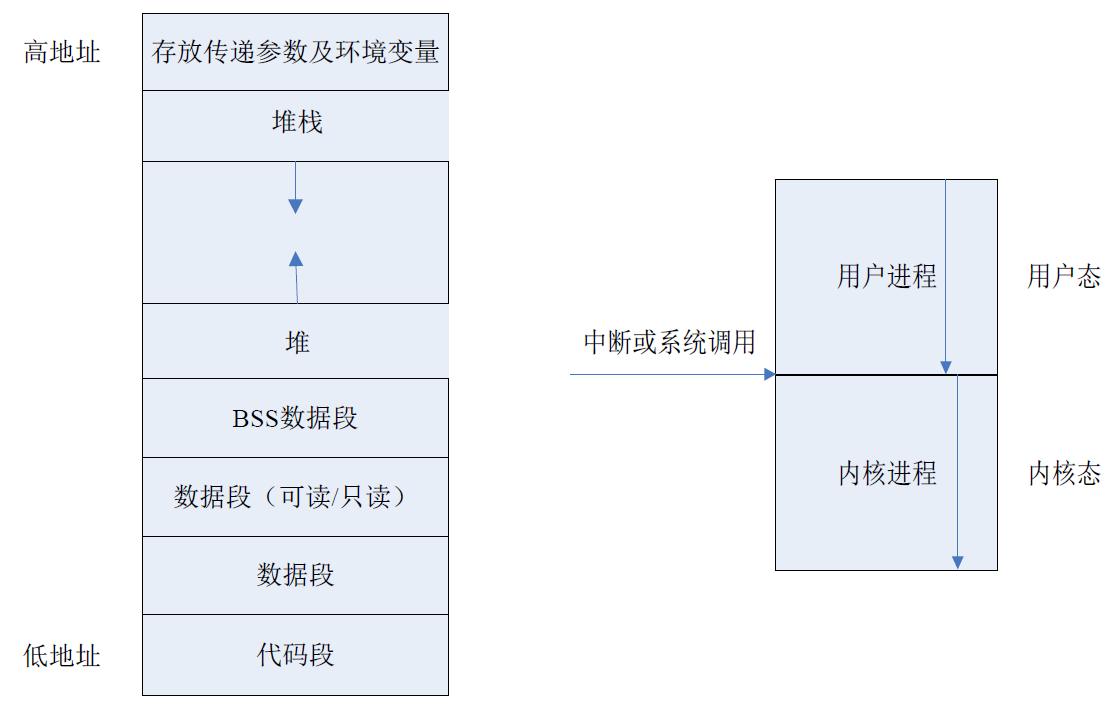
用xshell连接Linux服务器时，首先启动的是sshd的进程(用于通信的)；再用此进程启动一个 -bash

,用来解析命令。



6.

Linux系统是一个多进程的系统，它的进程之间具有并行性、互不干扰等特点。也就是说，进程之间是分离的任务，拥有各自的权利和责任。其中，**每个进程都运行在各自独立的虚拟地址空间，因此，即使一个进程发生了异常，它也不会影响到系统的其他进程**。



(1)

数据段：存放全局变量、常数，其分成：

1) 普通数据段（包括可读可写/只读数据段，存放静态初始化的全局变量、常量）

2) BSS数据段 （存放未初始化的全局变量）

(2)

代码段：存放的是程序代码的数据。

(3)

堆栈段 (栈区)：存放的是子程序的返回地址、子函数的形参、以及程序的局部变量等。

(4)

内存分为：

用户态：访问的栈、堆、数据段 都是在用户态的内存。

内核态：访问内核态的内存，如read、write

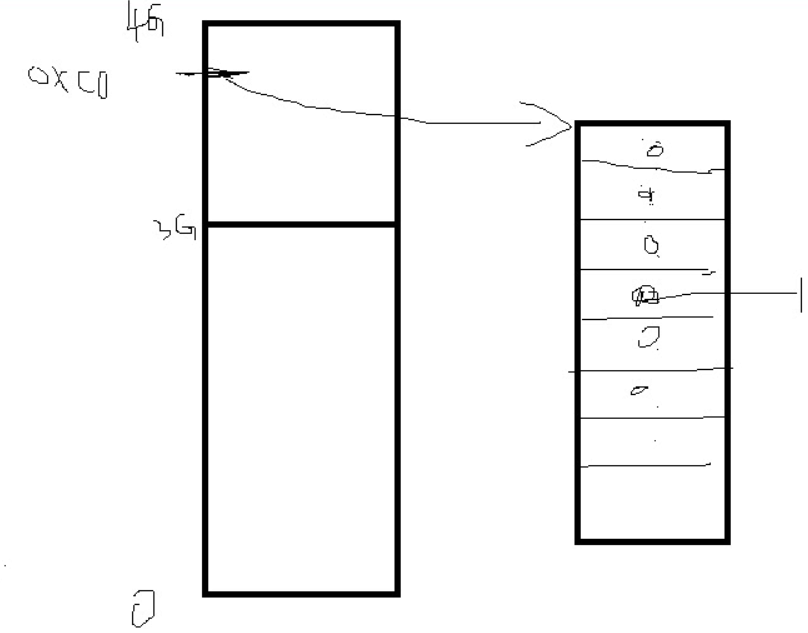
(5)

进入内核态两种方式 ：中断(硬中断、软中断)、系统调用(open、read、write等)。

7. **系统调用(操作硬件) 是如何实现的？** 为什么要尽量避免使用系统调用，很耗时 ？？

因为系统调用还存在二次偏移。 内核会有个地址：**陷入门/中断门** (会发生虚拟地址和物理地址的映射)，这个地址是系统调用的唯一入口，也是系统调用的所有函数存放的起始地址： 即所有系统调用的函数的入口地址，都存在于一个 **函数指针数组**(陷入门)中。

故要先从陷入门去函数指针数组中拿到地址，才能真正跳转到系统调用。 这样便无法通过解引用来访问内核，确保了安全性，但也比普通的函数消耗的时间更长。



内核区：3G ~ 4G的区域

8. 查看 **ps -elf**

(1) 调度策略优先级**PRI**： 数字越小，优先级越高.（man sched）

1) **实时调度策略**(多用于嵌入式)：**SCHED\_FIFO**(先进先出) 、**SCHED\_RR**(轮转)

优先级(100个): 0 ~ 99 ， Ubuntu中是 -40 ~ 59

2) **普通调度策略**：**SCHED\_OTHER**

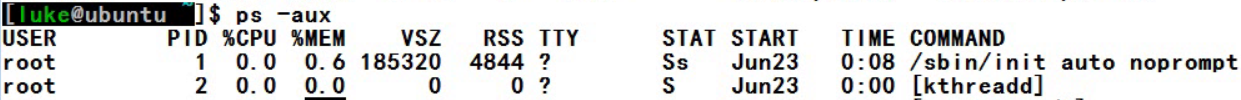
优先级(40个) ：100 ~ 139 ，Ubuntu中是 60 ~ 99

(2) **NI**: 优先级发生过了调整。-20代表优先级提高了20(数字越小，优先级越高)

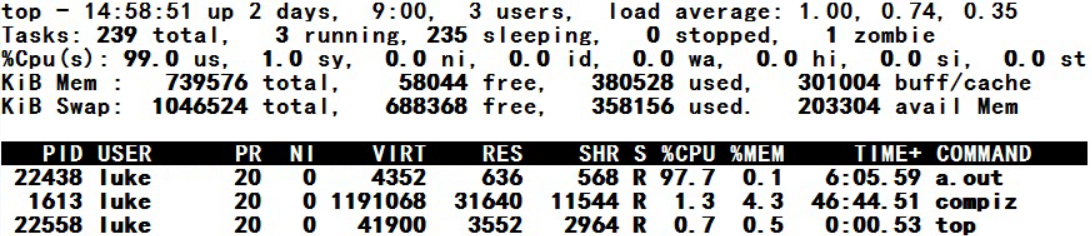
(3) **TIME:** 指进程的累计运行时间 (此时间指：消耗了时间片的时间，睡眠是不会增加的)



9. 查看 **ps -aux** : 查看当前进程的CPU占用率、内存占用率



10.查看 **top** : 显示前20条进程，动态的改变

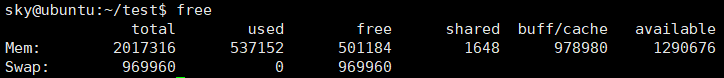


11. 虚拟内存 (Linux下叫SWAP分区)：**本质上是一块磁盘，当内存不足时**，需要用到虚拟内存。

(1)Windows下虚拟内存：如下图



(2)Linux下虚拟内存：Swap分区

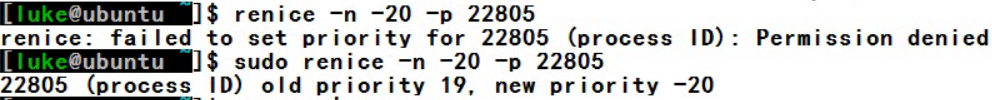


12.

操作系统的关机过程：依次把每个进程都结束。

kill -9 pid //杀死此进程

renice命令： 调整正在运行的进程的优先级



13.

查看磁盘有多少分区： **sudo fdisk -l**命令

查看当前的物理内存： **free** 命令

查看当前有多少在后台运行的进程： **jobs**命令

让原本**暂停(**如按**ctrl+z**)的进程，重新在后台运行： **bg**命令(将挂起的进程放到后台执行)

把进程从后台拉到前台： **fg**命令

让程序在后台启动运行加 &：  **./a.out &** (后台运行)

14. 进程的创建： Linux下有四类创建子进程的函数：system()、 fork()、 exec\*()、 popen()， **而内核提供的系统调用只有两种：fork()、execl()**

1. **fork()系统调用**： **pid\_t fork(void)**;

//**返回2个值，后面一定要写if、else来控制**

它从已存在进程中创建一个新进程。新进程为子进程，而原进程为父进程。它和其他函数的区别在于：它执行一次返回两个值：因为**此时有了两个进程了**，**其中父进程的返回值是子进程的进程号**，而**子进程的返回值为0**，若出错则返回-1.因此可以通过返回值来判断是父进程还是子进程。 fork()时会有dup机制; 父子进程的缓冲区是相互独立的。

fork函数创建子进程的过程为：使用fork函数得到的子进程是父进程的一个**完美复制品(PCB结构体的复制)，它从父进程继承了进程的地址空间**，包括进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设定、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端， 而子进程所独有的只有它的**进程号**(一般为父进程的pid+1)、**独立的资源使用(栈、堆、数据段的数据)**、计时器等。 通过这种复制方式创建出子进程后，原有进程和子进程都从函数fork返回，各自继续往下运行，但是原进程的fork返回值与子进程的fork返回值不同，在原进程中，fork返回子进程的pid,而在子进程中， fork返回0, 如果fork返回负值，表示创建子进程失败。

注意：

fork会给子进程 复制套接字描述符， 但并 **不会给子进程复制套接字** （套接字属于操作系统，不属于进程）

若杀掉子进程的父进程，则子进程会换一个爹ppid： 因为对于子进程来说，必须要有一个父进程来回收子进程的尸体(PCB结构体)。

一个进程便会返回一个值，故返回2个值； 当进入main函数时，便有了父进程，执行到fork后才会有子进程,故父进程的执行速度快； 子进程的pid一般比父进程的pid大1

(2) exec\* 由一组函数组成, 是系统调用。

**int execl(const char \*path, const char \*arg, ...)**//鸠占鹊巢

//**path**：要执行的文件的路径, 即**新进程的**(可执行程序a.out) 的路径。

**exec会拉起这个路径的新进程**，并用此新进程的代码段覆盖掉当前进程的代码段

注意：path参数要加上 ‘ ./ ’ ；其他exec的file参数：不用加‘./’,只用写文件名

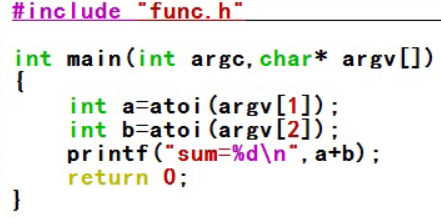
//**arg**： 执行该文件时传递的参数列表：argv[0],argv[1]，argv[2]... 最后一个参数要必须用空指针NULL作结束。 (其中argv[0]是可执行程序的路径)

exec函数族的工作过程与fork完全不同，fork是在复制一份原进程，而**exec函数是用exec的 第一个参数指定的程序 覆盖掉当前的进程空间**（也就是说**执行exec族函数后，它后面的所有代码不再执行了**，因为原后面的代码被覆盖掉了、不存在了）

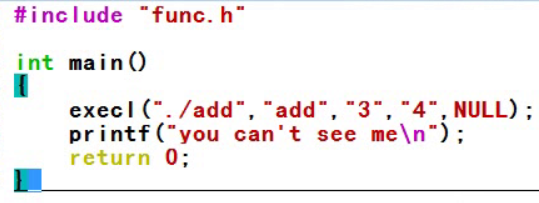
其实**exec并没有创建新进程，只是拉起一个新的进程**，并用新进程的代码段覆盖掉当

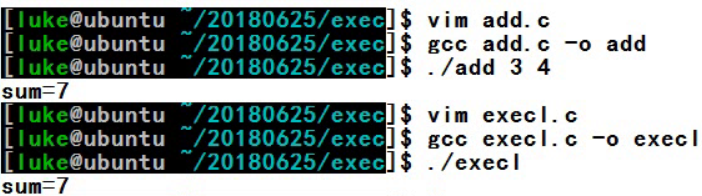
前进程的代码段，即原有进程代码不存在了; 注意：**拉起的新进程的 进程pid、 ppid仍是原进程的pid、ppid的值没有变，仅变了代码段**。

1)add.c



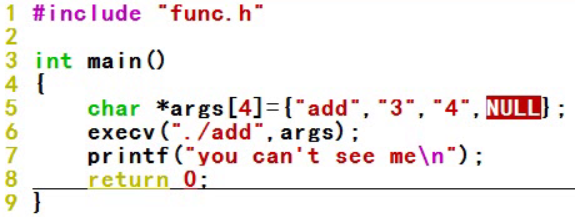
2)execl.c





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

//通过指针数组来传递参数，即把execl中要传的参数都放到指针数组中。



注意：故知道了 bash是如何执行execl、fork、ls等进程的 ？

当启动一个新进程 ./ 的时候，先fork()一个子进程，在子进程中再去执行这些操作,即通过exec\* 把代码段覆盖到当前子进程中。

(3) system函数

**int system(const char \*string);** //通过**fork()、再exec\*()**实现的

1)

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

2) 示例 **system()中嵌套脚本**：

#include <stdio.h>

#include <stdlib.h>

int main()

{

system("ls -l "); //先fork(), 再在子进程中，用exec\*() 把“ls -l”进程拉起来，并显示到当前进程。

return 0;

}

3)

此外，**system函数的参数还可以是一个可执行程序**，例如：

system(“/home/wangxiao/a.out”); 如果想要执行system后面进程的时候，不至于对当前进程进行阻塞，可以利用& 将/home/wangxiao/a.out调到后台运行。

4)

由此可知道C语言是如何嵌套其他语言的？ 通过启动一个新进程

因为每一个语言都可以编成一个可执行程序的，而C语言去嵌其他语言， 就是把这个可执行程序当作一个新进程来启动并执行的。 所以C语言嵌套其他语言效率不好。

15. **虚拟地址和物理地址的转换：通过缺页异常**

(1)每个进程都有独立的0 ~ 4G的进程地址空间：这是虚拟的地址空间，是编址。

(2)平时能看到的地址、%p打印的地址、malloc的地址等：都是虚拟地址，真正要发生写操作时，才会映射到物理地址。只有搞嵌入式编程，能打开设备，才能看到物理地址值。

(3)为什么能把相同的虚拟地址，可以映射到不同的物理地址上(如写时复制) ？

因为每个进程的叶目录的起始物理地址是不相等的。

(4)为什么malloc的地址空间一定要free()掉？

因为不free的话，malloc的虚拟地址内存，其映射到的物理地址内存，就不能分配给其他进程用了，自己不用还不free，其他人也用不了，就造成了内存泄漏。

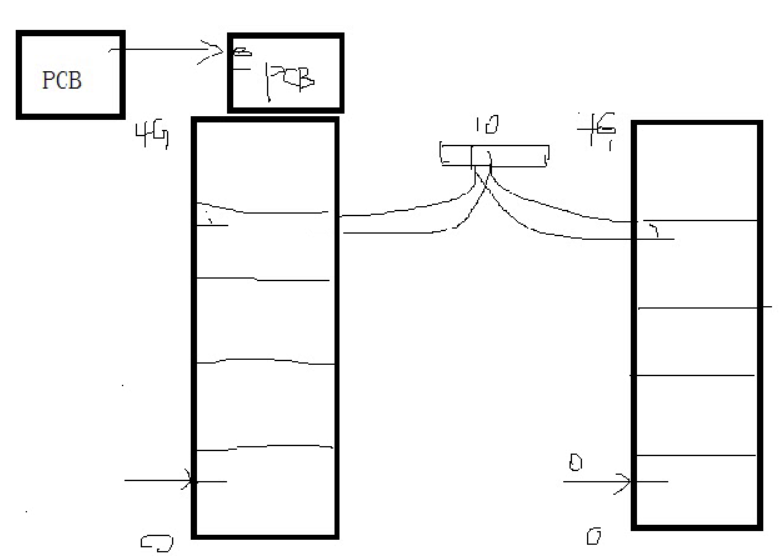
16. **写时复制: 虚拟地址不变，而是改变了物理地址**

(1)当父进程/或子进程 发生写操作时，需要重新申请一块物理内存，并把其原有值10先copy到新的物理内存中。此时父进程/子进程的虚拟地址会重新映射到这个新的物理内存中去，并把内存中的值10改为新的值5。

(2)当发生写时：子进程、父进程的虚拟地址相同，但映射的物理内存已经不同了。

(3)当只读不写时(没有发生修改时)： 父、子进程读的是同一块物理内存。

即如下图：父、子进程映射的是同一个物理内存，为10。



(4)父进程和子进程 是相互独立、各自进行的：即父进程改变时，不会影响子进程，就是因为发生了写时复制；

(5) 用fork继承打开的文件

父子进程唯一有联系的地方是：文件操作，当打开同一文件时，光标偏移另一个也会偏移(是同一个偏移的)。 即**父、子进程针对于同一文件，使用的是同一个引用计数+1**。

独立性在于： **fork()后的子进程会自动继承父进程的打开的文件(即dup机制)，**继承以后，父进程关闭打开的文件不会对子进程造成影响。

**linux操作系统每个进程的地址空间都是独立的，其实这里的独立说得是物理空间上的独立。**进程可以使用相同的虚拟地址，这不奇怪，因为转换后的物理地址并非相同的。

17.进程的控制

(1)**孤儿进程：父进程先退出，子进程被1号进程接管。**

用fork函数启动一个子进程时，子进程就有了它自己的生命并将独立运行。 如果父进程先于子进程退出，则子进程成为孤儿进程，此时将自动被PID为1的进程（即init）接管。 孤儿进程退出后，它的清理工作有祖先进程init自动处理。但在init进程清理子进程之前，它一直消耗系统的资源，所以要尽量避免。

(2) **僵尸进程：子进程先退出，父进程忙碌，子进程PCB未被清理。**

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

清理子进程：系统调用

**pid\_t wait(int \*status);** //父进程在**等待子进程**退出

**pid\_t waitpid(pid\_t pid, int \*status, int options);**

参数：

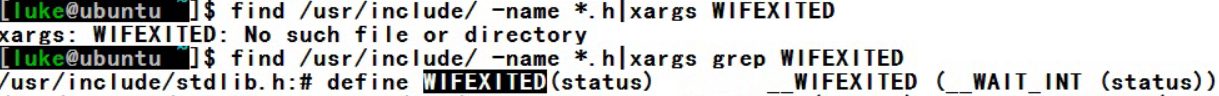
1. status参数：传出参数，存放子进程的退出状态(status要取地址)；

一般写**NULL**： 表示等待第一个子进程，并且不会看返回值。

通常用下面的两个宏，来获取子进程的返回状态：查看是正常退出还是异常退出？？？

**WIFEXITED(status)** 如果子进程正常结束，它就返回一个非0值 (传入整型值status)

**WEXITSTATUS(status)** 如果WIFEXITED值非零，它返回子进程 return 的退出码。



区别 宏(#define) 和 函数 ：宏没有返回值return，如上图

2)如果参数pid: 等待进程号为**pid**的子进程(定向回收); 若pid为**-1**, 表示等待所有子进程

1. options参数：用于改变waitpid的行为

其中最常用的是**WNOHANG**，它表示无论子进程是否退出都将立即返回，不会再将父进程的执行挂起； options为**0**时，是一直等待子进程。

4)

**wait() 和 waitpid()** ：父进程执行快，执行到此函数时**会暂停父进程**，**去等待一个要退出的子进程**，并进行清理工作； 便可以不再像之前那样，用sleep()来等子进程了。子进程退出时，会给父进程发一个信号，即SIGCHLD信号。

wait()： **随机地等待一个**已经退出的子进程，并返回该子进程的pid；

waitpid()： **等待指定pid**的子进程，**定向回收**；

5)

#include "func.h"

int main()

{

pid\_t pid;

pid=fork();

if(!pid)

{

printf("I am child,pid=%d\n",getpid());

char \*p=NULL;

\*p=5; //制造僵尸：访问空指针，子进程会挂掉，并退出。

return 5; //返回退出码 5

}else{

printf("I am parent\n");

int status;

pid\_t child\_pid;

child\_pid=**wait(&status)**; //父进程等待子进程退出

printf("child\_pid is %d\n",child\_pid);

**if(WIFEXITED(status))**

{ **//**获取子进程的返回状态

printf("exit status value=%d\n",**WEXITSTATUS(status)**);

}else{

printf("child process core dump\n");

}

return 0;

}

}

18. 进程的终止

进程的终止有5种方式：

 (1) **main函数的自然返回return**；

 (2) 子函数中调用 **exit()**函数(在子函数中直接终止): **不会清理I/O缓冲**

 (3) 调用 **\_exit()**函数 (直接封装的系统调用): 会清理I/O缓冲(若printf未加\n，就打不出来了)

 (4) 调用 **abort()**函数 (一般是大神在中间键上用)

 (5) 接收到能导致进程终止的信号：**ctrl+c** (**SIGINT) 、 ctrl+\** (**SIGQUIT)**

注意：**终止信号是发给进程组的**，进程组内的所有进程都会结束掉。

19. 守护进程 (Daemon进程)

(1)服务器上很多进程都是守护进程，**为了让它在后台默默地运行**，防止服务器在执行一个可执行程序，一个人手贱 ctrl+c把正在运行的程序给结束掉了; 同时也为了避免进程在执行过程中的信息在任何终端上显示。

Daemon运行在后台也称作“后台服务进程”。 它是没有控制终端与之相连的进程。它独立与控制终端、会话周期的执行某种任务。

(2)Windows下把没有界面的进程叫服务，Linux下都叫进程。

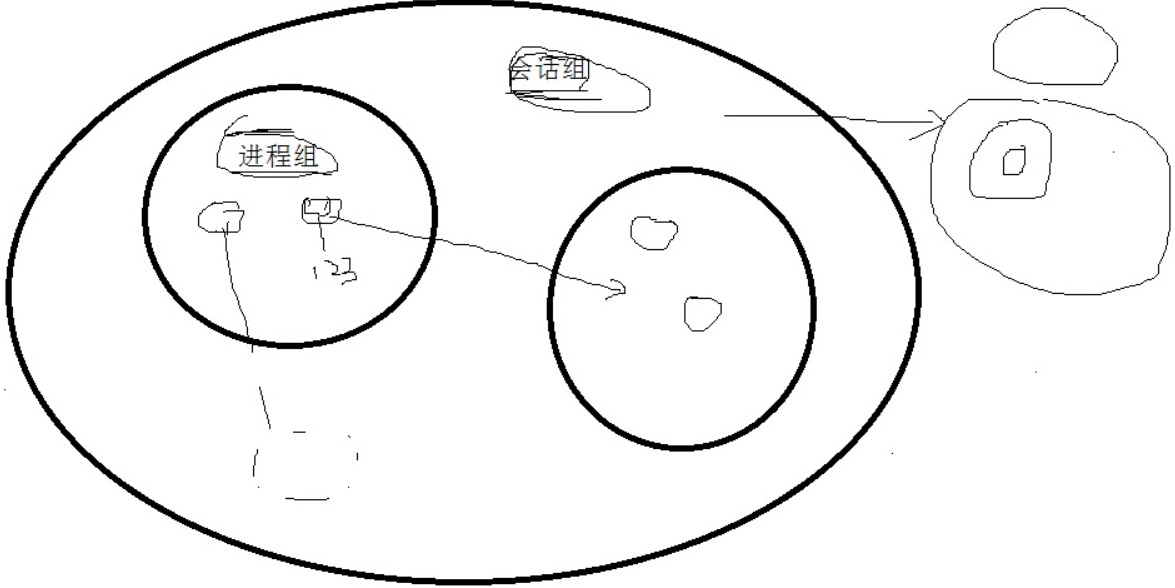
**最后一个字母为d**：就是demon进程。



(3)

由于在linux中，每一个系统与用户进行交流的界面称为终端，**每一个从此终端开始运行的进程都会依赖这个终端**，这个终端就称为这些进程的控制终端(如 xshell)。 当控制终端被关闭时，相应的进程都会自动关闭。但是守护进程却能突破这种限制，它被执行开始运转，**直到整个系统关闭时才退出**。

(4) 分组是为了便于管理：进程ID、进程组ID、会话组ID(会话周期ID)



每个xshell窗口相当于一个会话。

一个会话 (1个xshell终端窗口)：包含多个进程组，一个进程组又包含多个进程。

1. 进程的**进程组ID**：pgid：是进程组组长的进程号/也是父进程的pid **getpgid();**

2) 进程的 **会话组ID/会话周期ID**：sid **getsid();**

若杀死会话组ID (如-bash进程)，则xshell窗口就会结束。

1. 设置新的进程组ID

**int setpgid(pid\_t pid, pid\_t pgid);**

**//**设置当前的进程组ID: pid 为其他进程组ID: pgid

**int setpgid(0,0);**

//例如：子进程直接成立一个新的进程组，并且自己为组长，即pgid为自身的pid。 此时若父进程已经结束，子进程仍然可以运行(因为并不是一个进程组了)。

ctrl+c 结束原进程组：此进程并不会结束，父进程换为1号进程。

1. 设置新的会话组ID

**pid\_t setsid(void);**

**//**区别于设置进程组ID： 这个不能重新加入一个会话，只能重新建立一个会话，且会话组ID sid为自身的pid。

ctrl+c 结束原进程组：此进程并不会结束，父进程换为1号进程，**不依赖于当前窗口**了，即关掉此xshell窗口，进程并不会结束，重新再打开一个窗口，此进程仍然在。故创建demon进程就是基于此特性，让其真正运行在后台。

(5)**创建demon进程：**

1)**创建子进程，父进程退出return** (此时父进程换为1号init进程)

2)**在子进程中setsid创建新会话** (摆脱原会话的控制)

3)**改变当前目录为根目录chdir(“/”);**

4)**重设文件掩码：unmask(0);**

5)**关闭所有不需要的文件描述符**：即关闭0、1、2不需要了，已经和屏幕脱离了(后台运行)，信息记录到日志，不用打印到屏幕了。

过程分析：

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

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

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

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

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

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

close(i);

#include "func.h"

int main()

{

if(!fork())

{

setsid(); //成为新进程组组长和新会话领导，脱离控制终端

chdir("/"); //设置工作目录为根目录

umask(0);

int i;

for(i=0;i<3;i++) //尽可能关闭所有从父进程继承来的文件

{

close(i);

}

while(1)

{ //记录在系统日志中，在 /var/log/目录下

sleep(3); //每过3秒写一次日志，内容是I am dashen

syslog(LOG\_INFO,"%s%d\n" ,"I am dashen" ,i);

i++;

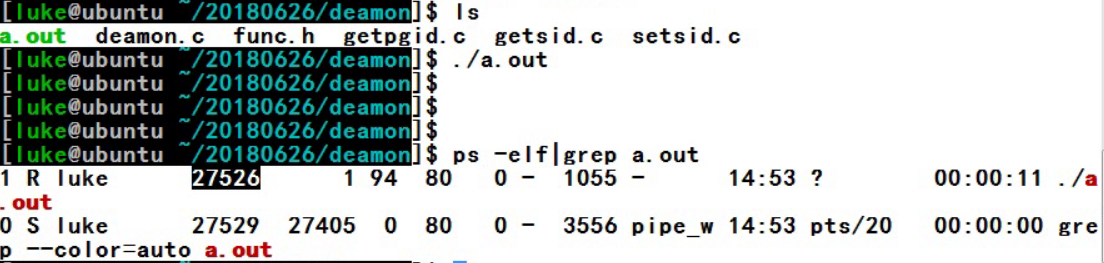
}

}else{

return 0; //父进程退出

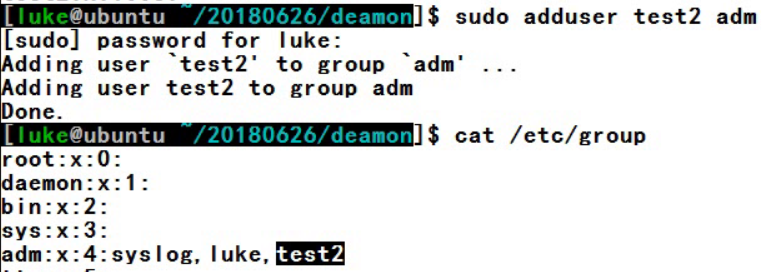
}

}



20. **查看有哪些文件组**，组中的成员有哪些：cat /etc/group

**添加成员到组里面**：sudo adduser 如下图



21. 在系统日志里面，写日志的接口：在 /var/log/目录下syslog文件：

**void syslog(int priority, const char \*format, ...);**

验证登陆的日志：vim /var/log/auth.log (登陆失败也会记录)

22.

printf(“**%04d**\n”,year); //不够4个，左边的补为0。

**struct tm \*gmtime(const time\_t \*timep);**

在返回的结构体指针所在的地址中：记录了年、月、日、时、分、秒等信息。其中：

年：从1900年开始的，要加1900

月：是0 ~ 11, 要加1

小时：要加8

23.

(1) **进程间通信**ipc 的4种方式 (为了通信)：

管道、共享内存、信号量、消息队列： 均是**system V** 标准的接口

1. **进程间同步**3种方式 (**能阻塞**：**为了控制时序**)： 管道、 消息队列、 信号量

24. 管道(3种类型)：在内核中打开的一段缓冲区

(1) 标准流管道

像文件操作有标准io 流一样，管道也支持标准文件流模式。用来创建连接到另一进程的管道，是通过函数popen 和pclose。

建立标准管道：

**FILE\* popen(const char\* command, const char\* open\_mode);**

//标准管道：先pipe()建立无名管道，再去fork()、exec\*()实现的。

//实现：**通过fork()和exec\*() 拉起一个进程，并在进程间pipe()建立一个无名管道连接**

**int pclose(FILE\* fp);**

因为是文件指针fp，故要用fread()、fwrite()等, 若用read()、write()，则需要转换：

**int fileno(FILE \*stream);**  //**转换文件指针为文件描述符**

函数popen()：允许一个程序拉起另一个程序作为新进程来启动，并可以传递数据给它或者通过它接收数据。**command 字符串：**是要拉起来的进程。 **open\_mode：** 必须是“r”或“w”。

若open\_mode 是“r”：**被拉起程序的printf输出，会被重定向到管道的写端**，即标准输出不会打印到屏幕上；原程序利用popen 函数返回的FILE\* 文件流指针，就可以通过常用的stdio 库函数（如fread）来读取被调用程序的输出；

若open\_mode 是“w”：**被拉起的程序的scanf标准输入，会被重定向到管道的读端**；原程序就可以用fwrite 向被拉起的程序发送数据。

(2) 无名管道

管道是linux 进程间通信的一种方式，查看命令： **ps -elf | grep ntp**

其中无名管道的特点：

1． 只能在**亲缘关系**进程间通信（fork父子 或 兄弟之间）

2． 半双工：只能一端读、一端写

3． 是特殊的文件，可以用read、write 等，只能在内存中

创建无名管道： **int pipe(int fds[2]);**

管道在程序中用一对文件描述符表示，其中一个文件描述符有可读属性，一个有可写的属性。**fds[0]是读（描述符：3）**； **fds[1]是写（描述符：4）**。

函数pipe 用于创建一个无名管道，如果成功，fds[0]存放可读的文件描述符，fds[1]存放可写文件描述符,并且函数返回0，否则返回-1。通过调用pipe 获取这对打开的文件描述符后，一个进程就可以从fds[0]中读数据，而另一个进程就可以往fds[1]中写数据。当然**两进程间必须有继承关系，才能继承这对打开的文件描述符**。管道不像真正的物理文件，不是持久的，即**两进程终止后，管道也自动消失**了。

(3) 命名管道

无名管道只能在亲缘关系的进程间通信大大限制了管道的使用，有名管道突破了这个限制，**有名管道通过指定路径名的方式实现不相关进程间的通信**。 创建FIFO命令：mkfifo，是通过调用的mkfifo()函数接口。

创建FIFO 文件的函数原型：

**int mkfifo(const char \*pathname, mode\_t mode);**

参数pathname： 要创建的FIFO 文件的全路径名；

参数mode ：创建的管道文件的权限； 12位: 用8进制表示(如：0666)。

如果创建成功，则返回0，否则-1。

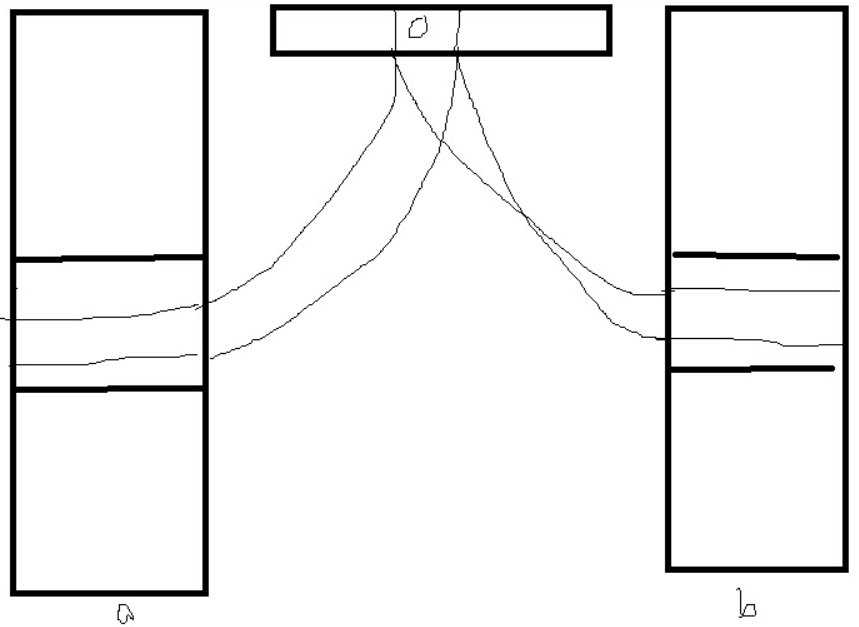
**删除文件**的函数原型为：(可删除任何文件，除了目录)

**int unlink(const char \*pathname);**

//由函数名可知：此删除接口，仅删除的是dirent信息,很快的。

25. 共享内存 (应用场景：一堆宅男看女主播)：5步

System V 共享内存机制：**shmget()、shmat()、shmdt()、shmctl()**,原理：**把同一段物理内存映射到多个不同进程的虚拟地址空间去**。 如下图：



共享内存本质是一段特殊的内存区域，进程间需要共享的数据被放在该共享内存区域中(即**同一个物理内存**)，**所有要访问该共享区域的进程，都要把该 共享区域(物理内存) 映射到本进程的虚拟地址空间中去**。 这样一个使用共享内存的进程就可以将信息写入该空间，因为都对应的同一个物理地址，另一个使用共享内存的进程就可以获取刚才写入的信息，使得两个不同进程之间进行了一次信息交换，从而实现进程间的通信。

共享内存允许一个或多个进程，通过同时出现在它们的虚拟地址空间的内存进行通信，而这块虚拟内存的页面被每个共享进程的页表条目所引用，同时并**不需要在所有进程的虚拟内存都有相同的起始地址**。 进程对象对于共享内存的访问通过key（键）来控制，同时又通过key 来进行访问权限的检查。一般工作时，key都是指定好的，不用自己创建

**key\_t ftok(const char \*pathname, int proj\_id);** //创建唯一的key值

函数ftok 用于创建一个关键字(必须是唯一的)，该关键字**用于关联一个共享内存段**：为了防止冲突，保证进程映射的都是此key关联的内存段，而不会映射到其他内存段。

参数pathname ：一个全路径文件名，并且该文件必须可访问。

参数proj\_id ：通常传入一个非0字符，如 1。

通过pathname 和 proj\_id 组合，可以创建唯一的key： 若调用成功，返回关键字，否则返回-1。

**ipcs命令**: 来查看共享内存：其中：

key： 以十六进制表示的，ftok()返回的值。

shmid： 共享内存的ID，由系统唯一分配的。

perm： 权限。

nattch： 有几个进程，映射到了该共享内存。

status： 若为dest,表示共享内存仅被标记删除，但仍有进程连接它。

**int shmget(key\_t key, int size, int shmflg);** //创建共享内存

函数shmget 用于创建/打开一个共享内存段，**通过参数key值，来唯一地创建一个共享内存**。函数成功，则**返回唯一的共享内存标识号**（由系统分配的，唯一的标识着共享内存）， 失败返回-1。

参数key：一个与共享内存段相关联的关键字。如果事先已经存在了一个与关键字关联的共享内存段，则直接返回该内存段的标识号，表示打开； 如果不存在，则创建一个新的共享内存段。 key 的值既可以用ftok 函数产生，也可以是IPC\_PRIVATE（用于创建一个私有的共享内存，主要用于父子通信）产生key值为0。

参数size：指定共享内存段的大小**，4K的整数倍**，以字节为单位；

参数shmflg：共享内存的权限，一般是**0600**，是掩码合成值，可以是访问权限值 与 **IPC\_CREAT**或IPC\_EXCL 的合成。IPC\_CREAT 表示如果不存在该内存段，则创建它。IPC\_EXCL 表示如果该内存段存在，则函数失败返回-1，防止和其他共享内存发生冲突。

**void \*shmat(int shmid, const void \*shmaddr, int shmflg);**//映射共享内存

函数shmat 将共享内存段映射到进程空间的某一地址。如果调用成功， **返回共享内存映射到进程地址空间的首地址**，否则返回(char \*)-1。

参数shmid ：共享内存段的标识通常应该是 shmget()的成功返回值

参数shmaddr ：指共享内存映射到当前进程中的地址位置。通常是**NULL**，表示让系统来选择共享内存出现的地址位置。

参数shmflg ：权限，一组位标识，通常为**0** 即可。

**int shmdt(const void \*shmaddr);**  //解除映射，但并没有删除共享内存

函数shmdt()用于将共享内存段与进程空间分离。函数成功返回0，失败时返回-1. 注意：将共享内存分离并没删除它，只是使得该共享内存对当前进程不再可用；解除后，虚拟地址空间就可以映射别的物理内存了。

参数shmaddr：通常为 shmat()的成功返回值。

**int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);**

函数shmctl()是共享内存的控制函数，可以用来删除共享内存段，失败返回-1。

参数shmid：共享内存段标识通常是 shmget()的返回值。

参数cmd：对共享内存段的操作方式，可选为**IPC\_STAT**, **IPC\_SET**, **IPC\_RMID**。通常为IPC\_RMID,表示删除共享内存段。

参数buf：表示共享内存段的信息结构体数据，通常为**NULL**。注意：此处为**传出参数**，要放数据的，故 (1)要自己定义好指针，并malloc好空间，再传过去; (2)或定义一个结构体变量(相当于自己申请了空间)，取地址再传过去。 所以 **ipcs命令显示出的信息，可以通过此接口来获取，信息存在了buf中**。

例如：shmctl(shmid ,IPC\_RMID ,NULL) 表示删除调共享内存段shmid (标记性删除)。

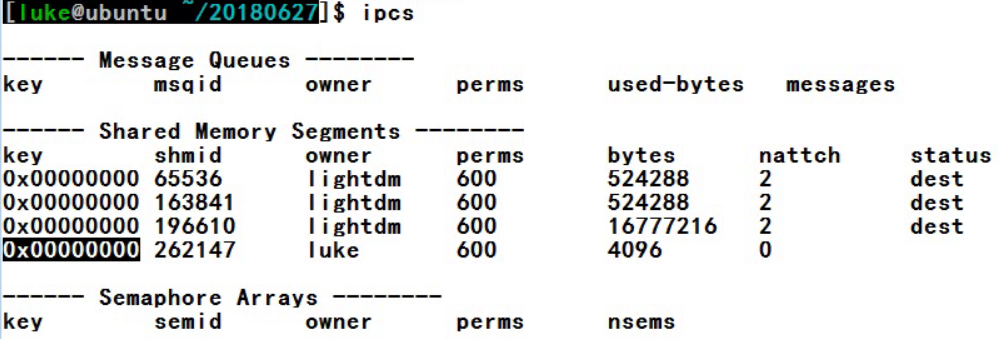
删除共享内存的命令：**ipcrm -m shmid**  //封装的 shmctl()函数

1. 创建 key值为0 的共享内存：以私有方式创建

int shmget(key\_t key, int size, int shmflg); //要在**fork()之前创建**！！！

其中参数**key要写成：IPC\_PRIVATE** (即key值会显示为0了):此共享内存只能在有亲缘关系的进程间使用（故要在fork()之前创建私有的共享内存：子进程能继承父进程的shmid了）： 故没有关系的进程不能再通过此key值映射这个共享内存了。 区别以下：

因为每次创建时若key值为0，都会重新创建一个共享内存，而不会打开这个私有的共享内存。 除此之外shmget创建时，若key值相同且不为0，则不是创建而是打开共享内存。



1. **shmctl()是标记性删除**：int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

1)若还有进程在使用共享内存，此时shmctl()删除共享内存，能删除吗？

能删除，但只是标记性删除，**仅把key值设为了0**：标记其他进程不能再用此共享内存了，此时共享内存仍然存在：要等**最后一个进程解除映射以后，才能真正去删除共享内存**

2)shmid\_ds 结构体如下：用shmctl()获取信息、设置信息。

**struct shmid\_ds** {

**struct ipc\_perm** shm\_perm; /\* Ownership and permissions \*/

size\_t shm\_segsz; /\* Size of segment (bytes) \*/

time\_t shm\_atime; /\* Last attach time \*/

time\_t shm\_dtime; /\* Last detach time \*/

time\_t shm\_ctime; /\* Last change time \*/

pid\_t shm\_cpid; /\* PID of creator \*/

pid\_t shm\_lpid; /\* PID of last shmat(2)/shmdt(2) \*/

shmatt\_t shm\_nattch; /\* Number of current attaches \*/

...

};

**struct ipc\_perm** {

key\_t \_\_key; /\* Key supplied to shmget(2) \*/

uid\_t **uid**; /\* Effective UID of owner \*/ 有效用户ID

gid\_t **gid**; /\* Effective GID of owner \*/ 有效组ID

uid\_t cuid; /\* Effective UID of creator \*/真实用户~

gid\_t cgid; /\* Effective GID of creator \*/真实组~

unsigned short **mode**; /\* Permissions + SHM\_DEST and SHM\_LOCKED flags \*/

unsigned short \_\_seq; /\* Sequence number \*/

};

注意：获取共享内存的信息：

shmctl(shmid, **IPC\_STAT**, &buf); //获取shmid\_ds结构体的信息，并放到buf中。

注意：设置共享内存的信息：

shmctl(shmid, **IPC\_SET**, &buf); //仅能改变以上结构体中加粗的成员:uid、gid、mode

26.

**原子操作（原语）**：是针对于操作系统层面的。 像原子一样不可再细分、不可被中途打断，此操作是以原子的方式被执行，要一口气执行完，执行过程**不能够被OS的其他行为打断**，是一个整体的过程。 (注意：一条机器码，本身即是个原子操作)。

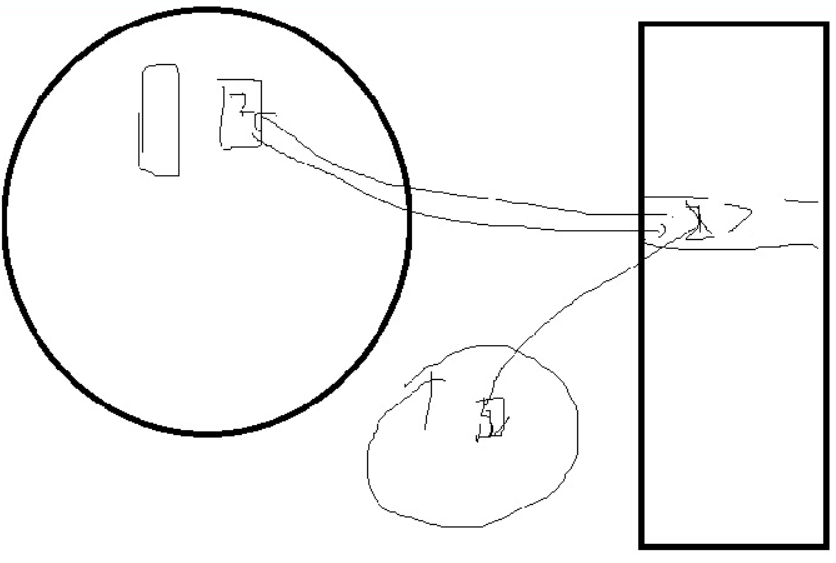
父子进程各加1000万，结果不等于2000万，为什么？？？

**p[0]=p[0]+1; 变成汇编会有三步操作**：放到CPU的寄存器中、加1操作、写回内存。

故 **p[0]=p[0]+1 并不是原子操作**，即p[0]=p[0]+1不是一条汇编指令(机器码)，故三步操作便会被打断：所以不是一个进程加1完成以后，另一个进程再去加1；而是前一个还没完成加1，另一个就去加1了。即没有控制好并发。

因为**时间片轮转**的原因：可能一个进程的三步操作只执行了2步，CPU的时间片就用完了，就从CPU上回到了就绪队列，即没有执行写回内存，CPU就去进行另一个进程了，因为上一个进程并没有到第三步写回内存，故其访问的内存数据仍是之前的。

若电脑是多核CPU，则最终两个数的和比单核CPU的和更小了：因为就一块内存，并发的概率更高了。



1. 区别：

**并发**：**有多个进程高频操作同一个资源**：有网络并发、内存并发、磁盘并发、CPU并发。

如对网络并发：即1000万人同时来买票，对后端的同一个服务器进行操作，服务器要做的是对并发进行分流：要**确保原子操作，进行加锁保护**。

**并行**：2个CPU各干各的程序。 要降低并发，提高并行性。

1. **assert断言**： **void assert( int exp );**

宏assert()用于错误检测。 如果表达式的结果为零，即为假时，**宏写错误信息到STDERR并退出程序执行**。 如果宏NDEBUG已经定义，即为真时，宏assert()将被忽略。

例如：

#include <assert.h>

int main()

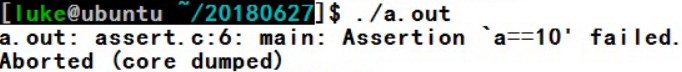
{

int a=5;

assert(a==10); //数据不符合预期时，**会调用abort()令程序主动崩溃**。

return 0;

}



1. **信号量(原子操作)：内存锁、睡眠锁**
2. 信号量（也叫信号灯）是一种用于在 不同进程间 或 一个进程的不同线程间：**提供**

**同步手段的原语**。信号量是进程/线程同步的一种方式。

有时候我们需要保护一段代码，使它 **每次只能被一个进程/线程来访问执行**，这种工作就需要一个**二进制开关**；

有时候需要 **限制一段代码可以被多少个进程/线程执行**，这就需要用到**计数信号量**。信号量开关是二进制信号量的一种逻辑扩展，两者实际调用的函数都是一样。

(2) 为了获得共享资源进程需要执行下列操作：

1）测试控制该资源的信号量。

2）若**信号量的值为正，则进程才可以使用该资源**。

进程信号量值**减1**(**关灯、加锁、通过**：关绿灯)，表示它使用了一个资源单位； 此进程使用完共享资源后对应的信号量会**加1**(**开灯、解锁、释放**：开绿灯)。以便其他进程使用。

3）若当前进程去加锁时（睡眠锁），**信号量的值为0，则当前进程进入休息状态(睡觉)**；直至**信号量值大于0，此进程马上被唤醒（在semop()处）**，返回第（1）步。

(3) 在内核中，**信号量和共享内存是在不同的队列中，故两个设的key值可以相同**，是没有影响的。

在信号量集合中，信号量的编号是从0开始的。

1. **int semget(key\_t key,int nsems,int flag);** // 创建或访问**信号量集合**

返回值：成功时返回**信号量标识符 semid**， semop和semctl 会使用它；出错时返回-1

参数key：唯一标识一个信号量的关键字，如果为**IPC\_PRIVATE**(key值为0)：表示创建一个只有创建者进程才可以访问的信号量，通常用于父子进程之间； 如果为**非0值**：表示创建一个可以被多个进程共享的信号量。

参数nsems： 要创建或使用**信号量的个数**，即**一个集合中有多少个信号量**。如果是创建新集合，则必须指定nsems。如果引用一个已存在的集合，则将nsems 指定为0。

参数flag： 一组标志，它的低9位是该信号量的权限(一般是：**0600|IPC\_CREAT**) ，与键值IPC\_CREAT 进行按位或操作，表示创建一个新的信号量；我们也可以通过IPC\_CREAT 和IPC\_EXCL 的联合使用确保自己创建出一个新的唯一的信号量，如果该信号量已经存在，就会返回一个错误。

**int semop(int semid,struct sembuf \*sops,size\_t num\_sops);** //用时1微秒

**进行原子操作**的+1(v操作)、-1(p操作) (即原语)，用于改变信号量对象中各个信号量的状态。返回值：成功返回0；失败返回-1。 **semop()系统调用，是操作系统级的原子操作**,但其保护的代码仍可被信号给打断。

参数semid： 由semget()返回的**信号量标识符**。

参数num\_sops： 为sops指向的 **sembuf结构体数组的大小(可有多个信号量集合)**

参数sops ： **指向结构体数组**的指针(**数组指针**)。 每个数组元素至少包含以下几个成员 :

**struct sembuf{**

**short sem\_num;**  //操作信号量集合中，编号是几的信号量 (编号从0开始)

**short sem\_op;**  //sem\_op 成员的值是信号量在一次操作中需要改变的数值。通常只会用到两个值，一个是 **-1，也是 p操作(通过pass)**，它等待信号量变为可用；一个是 **+1，也是 v操作(释放)**，它发送信号来通知信号量现在可用。

**short sem\_flg;**  //通常设为：**SEM\_UNDO**， 程序结束，信号量为semop调用前的值。

**};**

**int semctl(int semid, int semnum, int cmd, …);** //信号量的初始化

直接控制信号量信息：一般**用于初始化信号量的值**。 函数返回值：成功时，返回0；失败时，返回-1. 注意：要在fork()之前，就初始化好。

参数semnum： 要初始化信号量集合中的，几号信号量。 当要用到 成组的信号量时，从0开始。 一般取值为**0**，表示这是第一个也是唯一的一个信号量。 若cmd 是SETALL或GETALL，则此参数会失效, 写0即可。

参数cmd： 要执行的操作。 通常为：**IPC\_RMID**（立即删除信号量集合，唤醒所有被阻塞的进程）、**IPC\_STAT 、GETVAL（**根据semun 函数，返回信号量的值，从编号0开始：因为ipcs无法查看，故用GETVAL 来查看值是否设置好了）、**SETVAL**（设置多个信号量的值：根据semun 设定的信号值，从0 开始）、**GETALL**（获取所有信号量的值，并会将所有信号的值存入到 semun.array中）、**SETALL**（将所有semun.array 的值设定到信号集中）等。

参数 **…** (**可变参数**) ： 其是一个union semun 联合体（实现可变参数），参数… 可以是以下几个联合体的成员：

**union semun** { //联合体：用于实现可变参数的

int val; /\* Value for SETVAL，则参数… 通常设为1 \*/

struct semid\_ds \*buf; /\* Buffer for IPC\_STAT, IPC\_SET \*/

unsigned short \*array; /\* Array for GETALL, SETALL \*/ 信号量数组

};

//用于IPC\_STAT时，来查看信息：

**struct semid\_ds** {

struct ipc\_perm sem\_perm; /\* Ownership and permissions \*/

time\_t sem\_otime; /\* Last semop time \*/

time\_t sem\_ctime; /\* Last change time \*/

unsigned long sem\_nsems; /\* 集合中信号量的个数 \*/

};

**struct ipc\_perm** {

key\_t \_\_key; /\* Key supplied to semget(2) \*/

uid\_t **uid**; /\* Effective UID of owner \*/

gid\_t **gid**; /\* Effective GID of owner \*/

uid\_t cuid; /\* Effective UID of creator \*/

gid\_t cgid; /\* Effective GID of creator \*/

unsigned short **mode**; /\* Permissions \*/

unsigned short \_\_seq; /\* Sequence number \*/

}; //仅标粗的可以修改

删除信号量命令：**ipcrm -s semid**

1. **semop()实现原理**： **操作系统级的**原子操作， 而不是汇编指令级本身的原子操作 Why?

semop()是**系统调用**，对内核中某个值进行加1、减1。在执行此函数时**让时间片连续**，

从而让加、减1操作不被别人打断。

1. 若CPU是1个核, 如下：

加1：关闭中断、关闭抢占。(若没有时间片，依然不会放弃CPU，直到开启中断、抢占)

减1：开启中断，开启抢占。

1. 若CPU是多个核：除了以上操作， semop()还要调整 核间中断、核间抢占。

所以对于多核CPU来说，并不是核越多越好， 服务器科学测试后表明： 在**smp架构的4核**时，应对并发有最好的效率。

有**8核手机**：使用内存时，会把物理内存分为2部分：其中4个核操作一部分内存，另外4个核操作另一部分内存； 或大4核、小4核：即一会用4个核，一会用另外4个核。

1. **SEM\_UNDO**: 防止死锁

sembuf结构体成员sem\_flg，设为SEM\_UNDO：为了防止死锁。

加锁 执行代码 解锁： 自己写的执行代码可能会崩溃，则另一个进程要加锁(睡眠锁)时，便会一直等着前面那个已经崩溃的进程。 写上SEM\_UNDO后，若加锁、执行代码崩溃后，操作系统会自动将锁原有累计的减1、加1操作，给算回去。故另一个进程就可以唤醒加锁了。

(7) **如何判断有没有死锁？**  重新开一个窗口，查看进程状态 (ps -elf / top)

若死锁了：看两个进程是不是，都不断地处于sleep状态，并查看在sleep状态时，在

做什么函数操作的。

若没有死锁：多次查看进程状态，能看到是Running状态，少数几次处于sleep状态。

(8) 锁的类型：

**睡眠锁（互斥锁）**：当前进程去加锁(睡眠锁) 时，如果此时锁是锁着的，则当前进程就去睡觉； 当锁被打开时，就会唤醒当前进程并对其加锁。 如父子进程各加1000万时，用ps -elf查看，有时会看到进程是S睡觉状态（**信号量就属于睡眠锁**）。

**自旋锁**：当前进程去加锁(自旋锁) 时，如果此时锁是锁着的，则当前进程不睡觉，然后会不停地判断锁有没有被打开。 在内核态经常使用，好处是：没有进程或线程睡觉，故加锁的速度更快。缺点是：会一直自旋，浪费了CPU的资源。 在内核中的名字：spinlock

(9) 单核CPU：轮流访问共享资源，会因为时间片不够了，而造成不到2000万。

多核CPU：会同时读到共享资源，并发量更高了，更不会到达2000万了。

为了正确地实现信号量，信号量值的测试值的测试及减1 操作应当是原子操作（原子操作是不可分割的，在执行完毕前不会被任何其它任务或事件中断）。为此**信号量通常是在内核中实现**的。

(10) 防止车票卖成负票了： **要先加锁、再判断**

当票只剩1张时，父、子进程可能同时进入了p[0]>0判断，速度快的先加锁并卖出最后一张票； 另一个遇到睡眠锁先睡觉再被唤醒，又加锁再去减减，此时就成负票了。

改进：加锁放在if(p[0]>0)外面，解锁放在if判断中、以及else中(此时票数为0)。

(11) 为了保证 **父、子进程同步**、刚开始父进程要等子进程： 即保证2个窗口都要卖票

1)父、子进程**各sleep(1)**，睡1秒： 这样就可以实现2个窗口都在均衡地卖票，步调统一，而不是只是一个窗口在多卖票。

2)通过**管道**：父进程去读管道，卡住； 子进程再去写管道，父进程被唤醒： 父、子进程会去抢锁(是由操作系统来调度的)， 当票较少时，而父进程速度更快，可能会造成窗口2卖的票更多； 当票很多时，会比较均衡。

29. 计数信号量: 避免无效的加、解锁

由以上(10)可知：要先加锁、后判断，但当仓库中没有产品时，还要先去加锁，故可能会造成无效的加、解锁(**加锁了但并没有修改值**)： 可通过定义**两个信号量**来解决。

(1) **生产者**、**消费者**模式：对应两个进程，一个入库(生产)，一个出库(消费)。

**信号量的值用于计数**, 即两个信号量的值分别对应于：仓库剩余的空位个数、仓库中的产品个数。 用semctl()对两个信号量的值，进行初始化。

(2) 用2个信号量的值加1、减1，对应于：

生产：仓库剩余的空位个数-1 ；仓库中的产品个数 +1。

消费：仓库中的产品个数 -1； 仓库剩余的空位个数+1 。

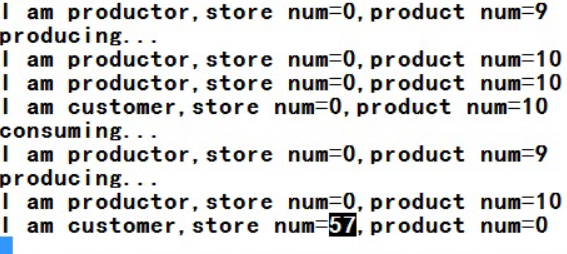
当仓库为0时，消费者会睡觉；当仓库满时，生产者会睡觉：避免了无效的加、解锁。



(3)

父子进程在生产、消费时： 若杀掉其中的 子进程消费者，会发现票数成57了，是因为加了SEM\_UNDO, 遇到程序崩溃时(如kill掉其中一个进程)，其会把减1操作给累加回去，来防止死锁。 （仅计数信号量时可明显看出此变化）

修改： 令sembuf结构体成员sem\_flg为0，不为SEM\_UNDO了。



30. 消息队列：先进先出

很多大公司在用异步消息队列的框架。 消息队列与 FIFO 很相似，都是一个队列结构，都**可多个进程往队列里面写信息**，**多个进程从队列中读取信息**。但FIFO需要读、写的两端都事先打开，才能够开始信息传递工作。而消息队列**可事先往队列中写信息，需要时再去打开，读取信息**。 但是，**消息队列若先打开读，且此时消息队列没有消息可读时，则仍会阻塞**。 注意：**消息是放在内核中的，和管道一样，其会占用内核空间**。

**int msgget(key\_t key, int msgflg);**

函数msgget()用于创建和访问一个消息队列。 该函数成功则**返回消息队列标识符**（类似于进程ID一样），失败则返回-1。

参数key： 唯一标识一个消息队列的关键字，如果为IPC\_PRIVATE(值为0)，用创建一个只有创建者进程才可以访问的消息队列，可以用于父子间通信；非0值的key(可以通过ftok函数获得)表示创建一个可以被多个进程共享的消息队列；

参数msgflg： 指明队列的访问权限和创建标志，一般是**0600|IPC\_CREAT**； 创建标志可选：IPC\_CREAT和IPC\_EXCL。如果单独指定IPC\_CREAT, msgget()要么返回新创建的消息队列id, 要么返回具有相同key值的消息队列id；如果IPC\_EXCL和IPC\_CREAT同时指明，则要么创建新的消息队列，要么当队列存在时，调用会失败并返回-1。

**int msgsnd(int msqid, struct msgbuf \*msgp, size\_t msgsz, int msgflg);**

**ssize\_t msgrcv(int msqid, struct msgbuf \*msgp, size\_t msgsz, long msgtype, int msgflg);**

函数msgsnd() 和 msgrcy(): 用来将消息添加到消息队列中、从一个消息队列中获取信息。

参数msgid： 指明消息队列的ID; 通常是msgget函数成功的返回值。

参数msgsz： 要发送、接收的消息体(即消息内容)大小：即mtext[]中存的数据大小

参数msgflg： 通常为**0** 或IPC\_NOWAIT。 若设置IPC\_NOWAIT,则msgsnd和msgrcv都不会阻塞，此时 如果队列满去调用msgsnd 或 队列空时调用msgrcv，会返回错误。

参数msgp： 是**变长结构体** (即第二个参数大小可变)，但其长度必须小于系统规定的上限。 必须以一个长整型成员变量开始，接收函数将用这个成员变量来确定消息的类型。 **因为是变长的结构体，所以必须自己定义出这个结构体**：其中第一个参数不能改，第二个参数可以自定义。如下：

**struct msgbuf msgp{**

**long mtype;**  /\* type of message \*/ 消息体的类型,**必须是正整数**

**char mtext[64];**  /\* message text \*/

**};**

结构体第一个成员 mtype： 是用户自己指定的消息类型（必须是正整数）；

该结构体第 2 个成员，仅仅是一种说明性的结构，实际上用户可以使用任何类型的数据，就是消息内容不仅仅是char类型的。

区别两个函数：1. **msgsnd()**：定义完结构体后，要给**mtype成员**赋值正整数；

2. **msgrcv()**：不用给结构体成员 赋值， 而是只用给其**参数msgtype**赋值

对于msgrcy()的参数 msgtyp有3种选项：

**msgtyp == 0**  接收队列中的第1个消息（通常为0）。

**msgtyp > 0**  接收对列中的第1个类型等于msgtyp的消息。

**msgtyp < 0**  接收其类型小于或等于msgtyp绝对值的第1个最低类型消息。

**int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);**

函数msgctl是消息队列的控制函数，常用来删除消息队列。

参数msqid： 由msgget()返回的消息队列标识符。

参数cmd： 通常为IPC\_RMID表示删除消息队列。

参数buf： 通常为NULL即可。

删除消息队列命令：**ipcrm -q msgid**

1. 消息队列和管道的区别？

每一个消息体是独立的，只能写一个、读一个； 而管道可以写多个，并能一次性全读

出来。 其次管道只能连续地去读数据，而消息队列可以有选择的读数据，即通过指定msgtype的值（正整数）。

31. 信号：为了实现有序退出

信号是进程在运行过程中，由自身产生或由进程外部发过来的消息（事件）。**信号是硬件中断的软件模拟(软中断)**。 **每个信号用一个整型常量宏表示，以SIG 开头**，比如SIGCHLD(子进程结束提醒父进程)、SIGINT(键盘中断)等，它们在系统头文件<signal.h>中定义, 也可以通过在shell 下键入kill –l 查看信号列表，或者键入man 7 signal 查看更详细的说明。 由进程的代码上的某个操作产生的信号：称为**同步信号**, 例如除以0； 由像用户按键，这样的进程外部事件而产生的信号叫做：**异步信号**。

(1) **信号的生成来自内核**，让内核生成信号的请求,来自于3个地方：

1)**用户键盘中断(异步信号)**： 用户能够通过键盘输入 ctrl+c、ctrl+\、ctrl+z，或者是终端驱动程序分配给信号控制字符的其他任何键来请求内核产生信号；

2)**执行时发生异常(同步信号)**： 当进程执行出错时，内核会给进程发送一个信号。 例如： 非法内存段存取(即虚拟地址背后没有对应的物理地址)、 浮点数溢出(浮点异常即内核除0异常) 等；

3)**其他进程kill(异步信号)**： 一个进程可以通过 kill系统调用 来给另一个进程发送信号，即实现了一个进程通过信号来和另外一个进程进行通信。

(2) 进程接收到信号以后，可以有如下3 种选择进行处理：

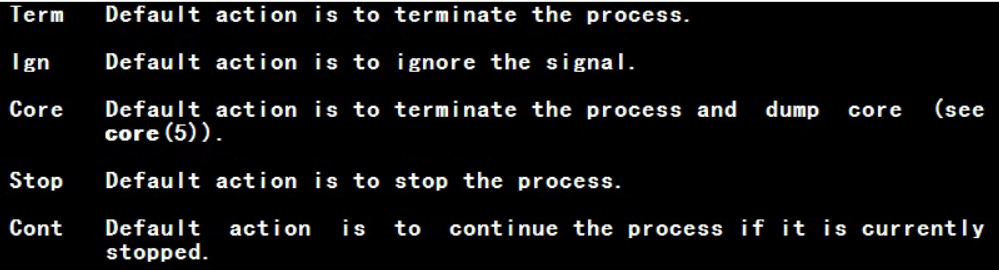
 1)接收默认处理(Default)：有五种如下图

接收**默认处理的进程通常会导致进程本身消亡**: 如连接到终端的进程，用户按一下ctrl+c，将导致内核向进程发送一个SIGINT信号，进程如果不对该信号做特殊的处理，系统将采用默认的方式处理该信号，即终止此进程的执行。

 2)忽略信号(Ign)： (区别于阻塞)进程可以通过代码，显示地忽略某个信号的处理例如：signal(SIGINT,SIG\_IGN)。 但是有些信号是不能被忽略的, 如9号信号；

 3)捕捉信号并处理： 进程可以事先注册信号处理函数(回调函数)，当接收到信号时，由信号处理函数自动捕捉并且处理信号。

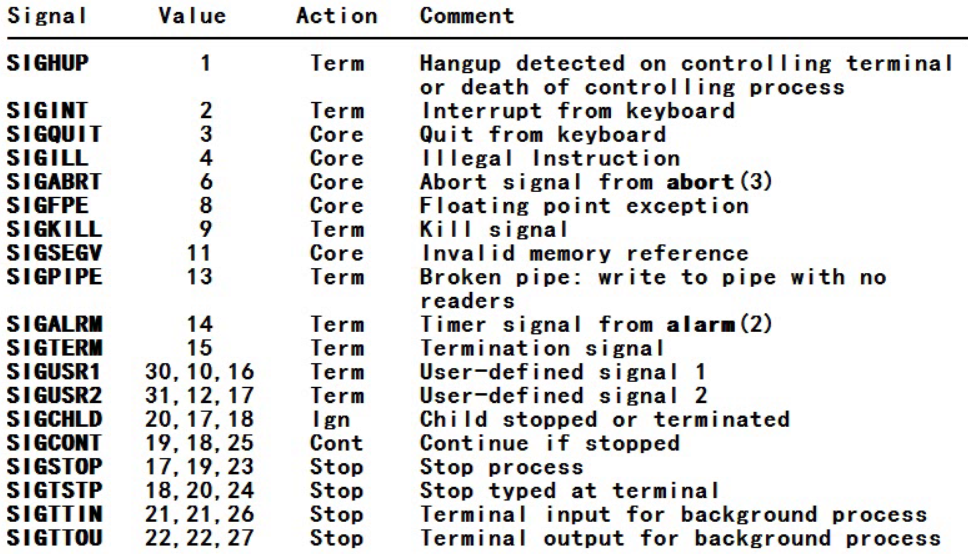
(3) 如下图默认的5种信号处理行为： 终止、忽略、 终止并产生core文件、 暂停、 继续



1）一般同步信号都会产生Core行为；

2）Stop 和 Cont类型的信号：即暂停、继续，是gdb调试要用的；

3）有两个信号**既不能忽略、也不能捕捉：** 是**SIGKILL**（终止，即平时用的kill -9）和 **SIGSTOP**（用于gdb调试）。即进程接收到这两个信号后，只能接受系统的默认处理，即终止进程。 SIGSTOP 是暂停进程。



(4)signal 信号处理机制： 注册信号捕捉函数来设定信号行为：为了不让信号进行其默认的行为，从而去实现有序退出。

**typedef void (\*sighandler\_t)(int);** // sighandler\_t是函数指针类型。

**sighandler\_t signal(int signum, sighandler\_t sigfunc);** //注册sigfunc()

第1 个参数signum ： 要捕捉的信号;

第2 个参数sigfunc： 是个函数指针（存储函数的入口地址），其表示要对该信号进行捕捉的函数。该参数也可以是**SIG\_DFL**(表示交由系统默认处理，相当于白注册了) 或 **SIG\_IGN**(表示忽略掉该信号而不做任何处理：即0改为了1，若忽略属性则会把1又改回0)。 signal调用成功，返回以前该信号的处理函数的地址(即返回函数指针)，失败返回SIG\_ERR

**void sigfunc(int signum);** 是信号捕捉函数，由signal()函数注册，此函数是**由内核来调用并传参的**，不是自己调用。 此**信号捕捉函数被注册以后，在整个进程过程中均会有效**，并且对**不同的信号可以注册同一个信号捕捉函数**。该函数只有一个整型参数，表示信号值。

(5) signal()的特性：

1） 若当前信号正在信号处理流程中(即信号捕捉函数中)，又有多个**同类型信号**到达时，来的信号不会打断当前信号处理流程，**只会多执行一次**。 故来的信号可能会发生丢失，若不想丢失信号，要用sigaction()函数来注册信号捕捉函数，其可实现无限重入。

2）若当前信号正在信号处理流程中，又有其他**不同类型信号**到达时，**可以打断当前信号处理流程并执行**，且在执行完后再去执行之前没有处理完的信号(是一个函数栈的压栈、出栈过程)。 用sigaction()函数可以使信号不被其他类型的信号打断。

3）signal()处理完一个信号之后，捕捉下一个信号时，是不需要再重新注册信号捕捉函数的； 而sigaction()是可以去实现重新注册的，使其仅一次生效，在下一次发出信号时以默认情况来处理。

4）当程序阻塞在一个系统调用(如read(...))时，若发生了一个信号：会先跳转到信号处理函数，等信号处理完毕后，系统调用再返回。

5） **内核中用sigset\_t集合(数组)来存储信号，每个信号对应一个位**。**为什么当发送多个相同信号时只会多执行一次？**

发送信号： 比如2号信号到来了，会把集合中2号信号对应位的**0改为1**。

响应信号： 去响应2号信号时，又会把对应位的**1改为0**，然后再去执行2号信号处理行为。此时若再发一次2号信号，又会把集合中对应位的0改为1，但是其得不到响应，因为其不能打断当前的信号处理行为，故多发几次是没有用的，只会不断的把集合中对应位的1改为1而已(内核PCB中对每一个信号只会存储一次)，只有当前信号执行完后，才会去判断集合中哪一位为true (1) ,并去将1改为0后再去执行。故**当前信号处理函数在没有执行完时，又发了多个相同的信号，只会多执行1次的。** 可用sigaction的SA\_NODEFER掩码来实现全部信号处理行为都执行。

6） 发送 2号、3号、2号信号，最后一次2号信号会不会打断前面的3号信号？

不会的，因为再发2号时，第一个2号还没有执行完，只是将2号对应位的0改为了1。

(6) sigaction 信号处理注册： 更加强大

**int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact);**

sigaction 也用于注册一个信号处理函数。 成功返回0，失败返回-1。

参数signum：需要捕捉的信号。

参数**act**： **传入参数** (特殊：要自己定义并申请空间，但并不是传出参数)，故此参数**可被多个sigaction()共用**。act结构体，包含了信号处理函数地址、处理方式等信息。

参数oldact： 是一个传出参数，sigaction 函数调用成功后，oldact 里面包含以前对signum 信号的处理方式的信息，即把旧的取出来放在oldact中。其通常为**NULL**。

结构体struct sigaction 原型为：(注意 sigaction结构体名称 与 sigaction()函数是重名的)

**struct sigaction**

**{**

**void (\*sa\_handler)(int);**  //老类型的信号处理函数指针，一般写**NULL**

**void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);** //新类型信号处理函数指针

**sigset\_t sa\_mask;**  //将**要被阻塞的信号集合set**

**int sa\_flags;**  //信号处理方式掩码 (通常是 **SA\_SIGINFO**)

**void (\*sa\_restorer)(void);**  //保留，不要使用，写**NULL**

**};**

注意：

1. 字段sa\_handler 是一个函数指针，用于指向原型为void handler(int) 的信号处理函数地址，即老类型的信号处理函数（如果用这个来注册信号捕捉函数，再将sa\_flags = 0, 就等同于用signal()函数）。
2. 字段sa\_handler 和sa\_sigaction 只应该有一个生效，如果想采用老的信号处理机制，就应该让sa\_handler 指向正确的信号处理函数，并且让字段sa\_flags 为0； 否则应该让sa\_sigaction 指向正确的信号处理函数，并且让字段sa\_flags 包含SA\_SIGINFO 选项。
3. 字段sa\_sigaction 也是一个函数指针，用于指向原型为

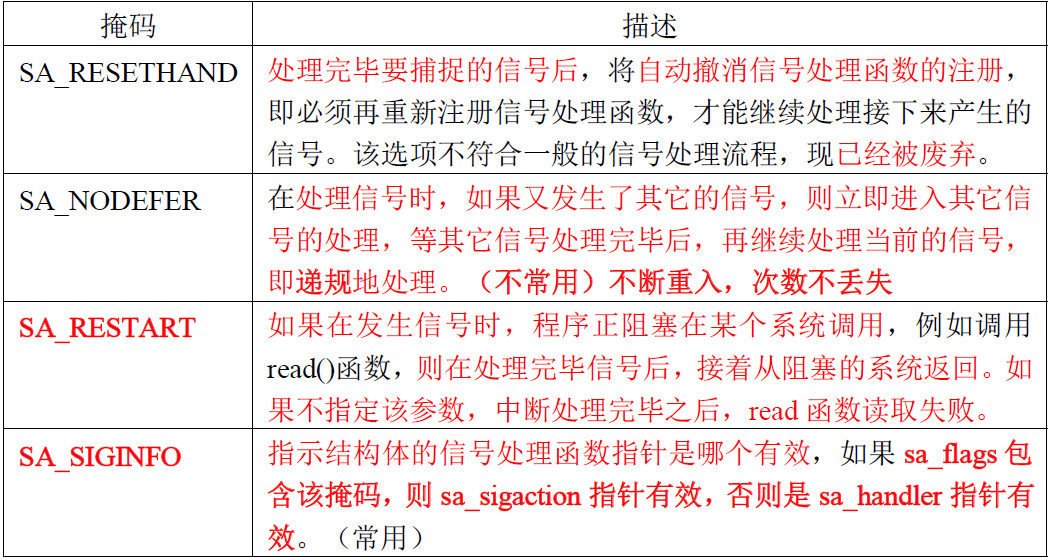
**void handler(int iSignNum, siginfo\_t \*pSignInfo, void \*pReserved)；**新类型的信号处理函数。此函数是**内核调用、并传参的**，不用自己写。

其中：参数iSignNum： 传入的信号。

参数pSignInfo： 与该信号相关的一些信息，**siginfo\_t是结构体类型**(可用于查看是谁发了信号)

参数pReserved： 保留，现没用。

4) 字段**sa\_flags 是一组掩码的合成值（按位或）**，以下宏都是位权限，均可以与其他宏按位或，指示信号处理时所应该采取的一些行为，各掩码的含义如下图：



1. sigaction结构体的 sa\_mask成员：包含**阻塞信号集合**的又一个**sigset\_t结构体**，

用于屏蔽信号，即**在当前的信号处理流程中，会屏蔽被阻塞的信号(仅暂时不执行，当前处理流程结束后仍会去执行)**。针对sigset\_t 结构体，有一组专门的函数对它进行处理，它们是如下： 函数的**前五个：位操作**； **第六个：系统调用**。

**int sigemptyset(sigset\_t \*set);**  //清空信号集合set(传入参数)

**int sigfillset(sigset\_t \*set);**  //将所有信号填充到set 中

**int sigaddset(sigset\_t \*set, int signum);**  //往set 中添加signum信号

**int sigdelset(sigset\_t \*set, int signum);**  //从set 中移除signum信号

**int sigismember(const sigset\_t \*set, int signum);** //判断signum 是否包含在set数组中(是:返回1,否:0）

**int sigpending(sigset\_t \*pend);**  //(重点)此接口是**系统调用**：

**从内核中把当前信号处理流程时被阻塞挂起的信号集合mask给拷贝出来，返回到pend中**，故参数**pend指针是传出参数，要自己定义**； 再结合sigismember()来判断集合中哪些信号是被挂起的。

其中，对于函数sigismember 而言，如果signum 在set 集中，则返回1；不在，则返回0；出错时返回-1； 其他的函数都是成功返回0，失败返回-1.

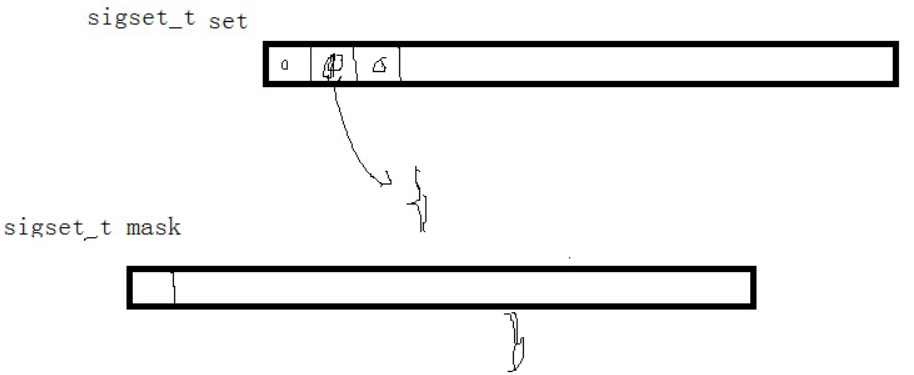
注意： 1） sa\_mask集合：用于**执行一个信号时防止其被其他信号给打断**。

注：是暂时阻塞信号而不是忽略信号： **仅在信号处理流程中被阻塞**。

例如：仅仅是在2号信号执行过程中，屏蔽了3号信号的打断；当2号信号执行完后，3号信号还是会去执行的。 要想不执行3号信号，即要达到忽略的效果，要在信号处理函数的最后加上exit();

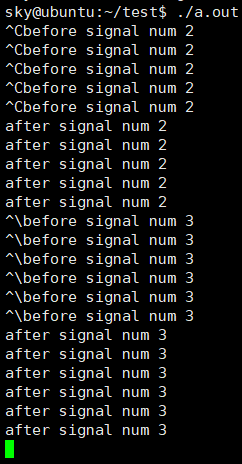
2）当一个信号被阻塞之后，在解除阻塞之前，该信号发生了多次，但是解除阻塞的时候，内核只会向进程发送一个信号而已，而不管在其阻塞期间有多少个信号产生，因为**linux 并不会对信号进行排队**。另外，这里用到了打断read 输入的中断处理，必须要加参数SA\_RESTART，对于signal 函数而言，它安装的信号处理函数，系统默认会自动重启被中断的系统调用，而不是让它出错返回。而对于sigaction 函数而言，必须要自己指定SA\_RESTART 实现重启功能，如果不指定则会read 失败，提示read 的时候中断发生。

3）**内核中用sigset\_t集合set(数组)来记录信号，每个信号对应一个位**； 除此之外，**内核中还有一个sigset\_t集合mask的阻塞信号集合，来记录阻塞了哪些信号**，被阻塞了的信号其对应位就会变为true (即为1)。 阻塞集合中是不需要自己来去进行位操作，而是通过提供的函数来去位操作的，防止小白用户出错。



4）sa\_flags为SA\_NODEFER时，处理函数中有sleep(3)，如当按了5次二号信号，3秒后会一下子直接出现5个after…。 为什么没有睡5次即15秒，只睡了3秒？

答：只会睡眠最后一次二号信号处理函数中的sleep(3)： 因为只要**有新的信号打断当前信号时，会马上唤醒当前进程的sleep函数 (即pause() 收到了信号被唤醒)**。 所以只有最后一次的2号信号没有新的信号来打断，故只会执行最后一个2号信号处理函数的sleep(3)。 //注：sleep()函数：是用alarm()和pause()实现的。



(7) sigprocmask 信号阻塞：**全程阻塞**

区别于sigaction()的阻塞： 仅在信号处理流程中时，进行信号的阻塞。

而**sigprocmask()是全程阻塞： 用于保护关键代码。**

函数sigprocmask 是全程阻塞，在sigprocmask 中设置了阻塞集合后，被阻塞的信号将不能再被信号处理函数捕捉，直到重新设置阻塞信号集合。

**int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);**

参数how：值为如下三者之一：

**SIG\_BLOCK**：将参数set信号集合，添加到进程原有的mask阻塞信号集合中(**设置阻塞**)

**SIG\_UNBLOCK**：在进程原有阻塞信号集合中，移除参数set中所包含的信号(**解除阻塞**)

**SIG\_SETMASK**：重新设置进程的阻塞信号集合为参数set的信号集合。

和上面的6个函数：位操作、系统调用，来进行组合使用。

参数set： 传入参数，阻塞信号集合set。（如上图sigset\_t结构体）：针对于sigset\_t 结构体，只能用一组专门的函数对它进行处理，防止小白出错。

参数oldset： 传出参数，存放进程中原有的信号集合，通常为NULL。

(8) **信号的响应时机**： 信号是软中断（硬件中断的软件模拟）

1．若进程正在运行：

1）当进程执行系统调用时，会先进行检查，看有没有信号。

2）当进程进行时间片轮转调度时，会进行检查，看有没有信号。

例如：while(1)时，还是会发生调度的，所以按ctrl+c才能响应此信号退出。

2. 若进程正在睡觉：

发送者发来信号时，会先把进程从睡眠队列移到就绪队列，再到上面的第1步，去进行检查。

32. **用程序发送信号**： **kill() 信号发送函数** (kill命令的实现)

**int kill(pid\_t pid, int sig);**

参数pid： 将要接受信号的进程的pid(可以通过getpid()函数获得)，来给自身发送信号，还可以发送信号给指定的进程。此时pid 有如下描述：

pid > 0 将信号发给ID 为pid 的进程；

pid == 0 将信号发送给与发送进程属于同一个进程组的所有进程；

pid < 0 将信号发送给进程组ID 等于pid 绝对值的所有进程；

pid == -1 将信号发送给该进程有权限发送的系统里的所有进程。

参数sig： 要发送的信号。

33. 计时器与信号

(1)睡眠函数， Linux 下有两个睡眠函数，原型为：

**unsigned int sleep(unsigned int seconds);** //是用信号实现的！！！

**void usleep(unsigned long usec);**

函数sleep 让进程睡眠seconds 秒，函数usleep 让进程睡眠usec 微秒。

sleep()睡眠函数，是用信号机制实现的，用到的函数有(sleep实现原理)：

**unsigned int alarm(unsigned int seconds);**

//设置一个**计时器**：**告知自身进程，要进程在seconds 秒后自动产生一个SIGALRM的信号**

**int pause(void);** //系统调用(放弃CPU)

//**将自身进程挂起并进到睡眠队列睡觉，直到被信号所打断**（任何信号都能把pause唤醒）

注意：因为sleep()函数内部是用alarm()实现的，所以在程序中最好不要sleep 与alarm 混用，以免造成混乱。

(2) 时钟处理

**Linux 默认为每个进程维护3个计时器**，分别是真实计时器、虚拟计时器和实用计时器。

 1）**真实计时器**： 计算的是程序**实际运行**的时间； ---直接

 2）**虚拟计时器**： 计算的是程序运行在**用户态Running时**所消耗的时间：

(可认为 实际时间 减掉 系统调用(内核态)、程序睡眠 所消耗的时间)---需要了解内核

 3）**实用计时器**： 计算的是程序处于 **用户态和内核态 Running时**所消耗的时间之和。 其并不会算sleep()、read()等卡住睡觉的时间。 ---最常用

例如：有一个程序运行，在用户态运行了5 秒，在内核态运行了6 秒，还睡眠了7秒，则真实计算器计算的结果是18 秒，虚拟计时器计算的是5 秒，实用计时器计算的是11秒。

**用指定的 初始间隔和重复间隔时间 为进程设定好一个计时器后**，**该计时器就会定时地向进程发送时钟信号**。以上3个计时器发送的时钟信号分别为： **SIGALRM**、 **SIGVTALRM** 和 **SIGPROF** 信号。 （3个计时器都是用不同的信号实现的）

**int getitimer(int which, struct itimerval \*value);** //获取计时器的设置

（用于获取计数器的信息，不常用）返回值：如果成功，返回0，否则-1.

参数which：指定哪个计时器，可选项为： **ITIMER\_REAL**(真实计时器)、

**ITIMER\_VIRTUAL**(虚拟计时器)、 **ITIMER\_PROF**(实用计时器)

参数value：结构体的**传出参数**，用于传出该计时器的初始间隔时间和重复间隔时间。

返回值：如果成功，返回0，否则-1.

**int setitimer(int which, const struct itimerval \*value, struct itimerval \*ovalue);** //设置计时器（常用），返回值：成功返回0，失败-1.

参数which： 指定哪个计时器，可选项为ITIMER\_REAL(真实计时器)、ITIMER\_VIRTUAL(虚拟计时器)、 ITIMER\_PROF(实用计时器)

参数value： 结构体的传入参数，指定该计时器的初始间隔时间和重复间隔时间

参数ovalue： 结构体传出参数，用于传出以前的计时器时间设置。

**struct itimerval**

**{**

**struct timeval it\_interval;** /\* next value \*/ 重复间隔

**struct timeval it\_value;**  /\* current value \*/ 初始间隔

**};**

**struct timeval**

**{**

**long tv\_sec;**  /\* seconds \*/ 时间的秒数部分

**long tv\_usec;**  /\* microseconds \*/ 时间的微秒部分

**};**

34. 多线程 (轻量级进程)

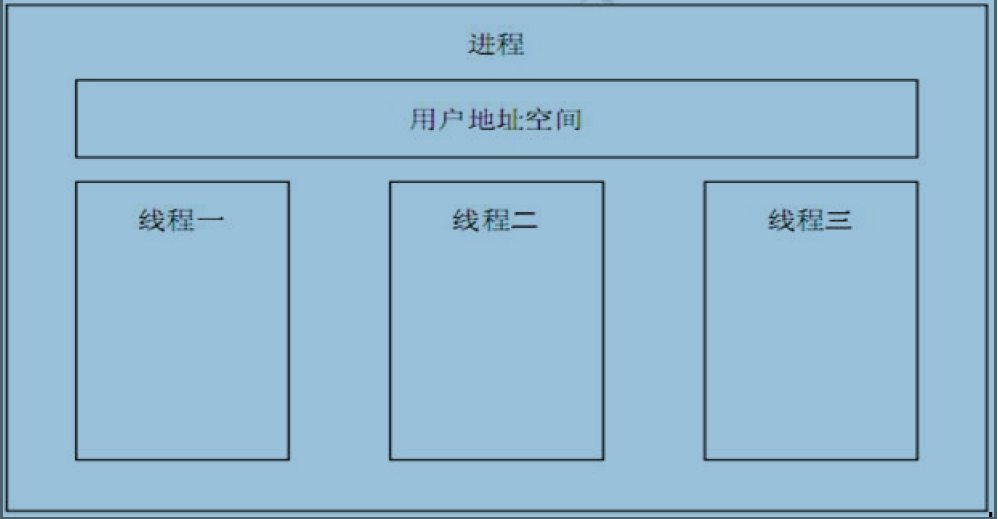
为什么有了线程的概念？ 是因为 进程的创建、切换，所耗费时间比较长。

进程是系统中程序执行和资源分配的基本单位。每个进程有自己的数据段、代码段和堆栈段。这就造成进程在进行切换等操作时都需要有比较负责的上下文切换等动作。为了进一步**减少处理器的空转时间，支持多处理器和减少上下文切换开销**，也就出现了线程。

线程通常叫做轻量级进程。线程是在共享内存空间中并发执行的多道执行路径(故**线程是不能独立存在的，必须依附于进程**)，是一个更加接近于执行体的概念，拥有独立的执行序列； 线程是**进程的基本调度单元，每个进程至少都有一个 main 线程**(**要最后退出**) 其与同进程中的其他线程共享进程的地址空间｛即共享堆空间、代码段、数据段、文件描述符、信号｝，因为其**仅仅拥有自己的栈空间，所以大大减少了上下文切换的开销**。

线程和进程各自优缺点：线程执行开销小，占用的CPU少，线程之间的切换快，但不利于资源的管理和保护； 而进程正相反。 从可移植性来讲，多进程的可移植性要好些。

同进程一样，线程也将相关的变量值放在**线程控制表**(**PCB也包含线程控制表**)内。一个进程可以有多个线程，也就是有多个 线程控制表和堆栈寄存器，但却共享一个用户地址空间。 要注意的是，由于线程共享了进程的资源和地址空间，因此，任何线程对系统资源的操作都会给其他线程带来影响（**是真正的共享**，而不是多进程的引用计数没有影响）



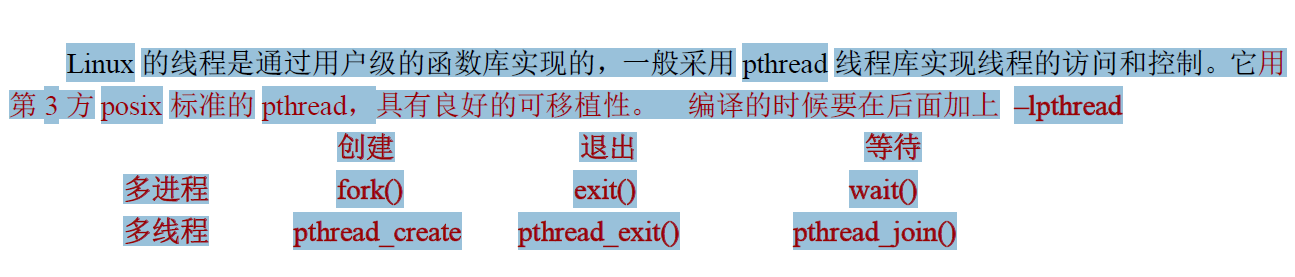
1. 线程的分类：用户级线程 和 核心级线程

**用户级线程**(类似协程，已是历史)： 内核中**只有1个协程在不断地跳转执行， 主要解决上下文切换问题**，调度算法和调度过程全部由用户决定，在运行时不需要特定的内核支持。缺点是无法发挥多处理器的优势。

**核心级线程**(目前均是核心级)： 允许不同进程中的线程 **按照同一相对优先调度方法调度，可以发挥多处理器的并发优势**。

现在大多数系统都采用用户级线程和核心级线程并存的方法。一个用户级线程可以对应一个或多个核心级线程，也就是“一对一”或“一对多”模型。

2. 进程、线程对应表： （注意：子线程间 拥有自己独立的栈空间，互不影响）



**要显式链接pthread线程库**： 即gcc要加上-l选项，当编译链接时，进行显式链接。（C库是隐式链接）

线程库用的是**nptl线程库**，遵守的posix标准（所有**pthread开头**的接口都是posix标准）；前面学的进程间通信，遵守的是system V标准 (BSD协议)

**创建线程**：即确定调用该**线程函数的入口点**，线程的创建采用pthread\_create()。

函数pthread\_exit()函数，这是**线程主动退出**行为。这里要注意的是，在使用线程函数时,不能随意使用exit()来退出函数进行出错处理，由于exit 的作用是使整个进程终止，往往一个进程包括了多个线程，所以在线程中通常使用pthread\_exit 函数来代替进程中的退出函数exit 。 （线程退出标准的方式）

由于一个进程中的多个线程是共享数据段的，因此通常在**线程退出之后，退出线程所占用的资源并不会随着线程的终止而得到释放**。 正如进程之间可以通过wait()函数系统调用来同步终止并释放资源一样，线程之间也有类似的机制，那就是pthread\_join 函数。pthread\_join 函数可以用于将当前线程挂起，等待线程的结束。这个函数是一个线程阻塞函数，调用它的函数将一直等待，直到被等待的线程结束为止，当函数返回时，被等待线程的资源被回收。 **父、子进程是有各自独立的输出的，只不过是共享一个屏幕终端；而主线程和子线程并不是相互独立的**。

查看线程命令：**ps -elLf** ,**只有 主线程的线程ID 和 进程PID 一样的**。

3. 注意：所有线程函数都不能用perror()了，要通过错误码来查看错误原因。

**int pthread\_create(pthread\_t\* thread, pthread\_attr\_t \* attr, void \*(\*start\_routine)(void \*), void \* arg);**

**void pthread\_exit(void \*retval);** //线程退出标准的方式

函数pthread\_create 用来创建线程，成功：返回0； **失败：返回对应错误码**。

函数pthread\_exit 表示线程的退出。 其**参数 retval 可被其它线程用pthread\_join 函数捕获**。各参数描述如下：

参数**thread**： **传出参数**，故要取地址，会从内核中把线程ID 给拷贝出来。

参数attr： 结构体指针，线程属性(优先级、调度策略等)，attr可以用

pthread\_attr\_init()等函数设置各成员的值，但通常为**NULL** 即可（最下面详细介绍）

注意：不同的线程，可以用同一个线程入口函数。

参数start\_routine： 函数指针，指向 新线程的入口点函数。 线程入口点函数带有一个 **void \*的参数 ,且其是由pthread\_create 的第4 个参数传入的 (来给子线程传参)**

参数arg： 用于传递给第3 个参数所指向的入口点函数，可为**NULL**，表示不传递。

4. pthread\_join()合并线程： **阻塞并等待线程退出、以及清理资源（回收线程控制表）**

(主线程要等待所有子线程，即合并掉所有子线程后，再最后退出)

线程从入口函数return自然返回；或调用pthread\_exit()函数：都可让线程正常终止。**线程函数的返回值，均可被其它线程用pthread\_join 函数获取**。

**int pthread\_join(pthread\_t th, void \*\*thread\_return);**

1. 该函数是一个**阻塞函数**，会一直等到参数th ID所指定的线程返回 (与多进程中的wait 或waitpid 类似); 若th为0 时，返回ESRCH错误。 注意：若线程被cancel了，则pthread\_join() 拿到的返回值是PTHREAD\_CANCELED ：((void \*) -1)

(2) **thread\_return**： **传出参数**，用于接收线程函数的返回值。 如果线程通过调用

pthread\_exit()终止，则pthread\_exit()中的参数retval 相当于返回值， 照样可被其它线程用pthread\_join 获取到。

**Why是二级指针**？ 因为函数指针所对应的线程函数，返回值类型是void\*, 即定义一个指针变量，来接线程函数传回来的返回值，故要取地址传过去。 用的是二级指针的传递

(3) 主线程停留在**futex\_** (**当进程/线程sleep时，会告诉停留在哪个函数**)

此系统调用由线程封装的，内核中用于加锁的系统调用是futex\_， 故man pthread类的函数是**man 3**的，指此接口封装了其他的系统调用。 man 2是直接封装的系统调用，和系统调用一模一样的，没有作额外的功能。

(4) pthread\_join()函数还有一个非常重要的作用：**清理资源, 并回收线程控制表**。

由于一个进程中的多个线程共享数据段，故在一个线程退出后，**退出的线程 其所占用的资源并不会随着线程结束而释放**。 如果th 线程并不是自动清理资源类型的，则th 线程退出后，**此线程本身的资源必须通过其它线程调用pthread\_join 来清除**,这相当于多进程程序中的waitpid

(5) pthread\_join()内部实现 只对参数thread\_return进行了一次解引用。

C:\Users\64407\AppData\Local\Temp\1555381202(1).png //对thread\_return只有一次解引用

`

(6) 注意： ps -elLf看到的线程ID不是真正的线程ID，而是proc文件系统来设计并显示的编号； 实际的线程ID是特别大的：要通过**pthread\_self()**函数返回值 来查看。

5. 线程的互斥：线程锁

在Posix Thread 中定义了一套专门用于线程互斥的mutex 函数，此锁效率很高。mutex 是一种简单的加锁的方法来控制对共享资源的存取，这个

只有两种状态（仅上锁和解锁），可以把互斥锁看作某种意义上的全局变量。 在Linux 中，互斥锁并不占用任何资源。

**为什么需要加锁？ 就是因为多个线程共用进程的资源**，要访问的是公共区间时（全局变量），当一个线程访问的时候，需要加上锁以防止另外的线程对它进行访问，实现资源的独占。

在一个时刻只能有一个线程掌握某个互斥锁，拥有上锁状态的线程能够对共享资源进行操作。 若**其他线程若想上锁一个已经上锁了的互斥锁，则该线程就会先挂起**，直到上锁的线程释放掉互斥锁为止。 实现原子操作如下：

**进程锁(信号量)**： 用的**系统调用（操作系统级）**，速度较慢。

**线程锁**： 用**汇编级指令**实现 (因为线程间共享进程地址空间，可以通过类似

全局变量来对每个线程进行判断)，速度很快，效率很高。

有两种方法创建互斥锁，静态方式和动态方式。

1. 静态初始化锁：

POSIX 定义了一个 **宏PTHREAD\_MUTEX\_INITIALIZER 来静态初始化互斥锁**：

**pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;**

在Linux Threads 实现中，pthread\_mutex\_t 是个结构体，而PTHREAD\_MUTEX\_INITIALIZER 则是一个宏常量。

(2) 动态初始化锁：

动态方式是采用pthread\_mutex\_init()函数来初始化互斥锁：

**int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr)；**

其中mutexattr 用于指定互斥锁属性（见下），通常为**NULL** ：表示使用缺省属性，即PTHREAD\_MUTEX\_TIMED\_NP属性，即为普通锁/快速锁。

(3) pthread\_mutex\_destroy()用于注销一个互斥锁：

**int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);**

销毁一个互斥锁即意味着释放它所占用的资源，且要求锁当前处于开放状态。由于在**Linux中**，**互斥锁并不占用任何资源**，因此Linux Threads中的 pthread\_mutex\_destroy()除了**检查锁状态**以外（锁定状态则返回EBUSY）没有其他动作。

(4) 锁操作：

加锁： **int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)；**

解锁： **int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex)；**

测试加锁： **int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex)；**

**pthread\_mutex\_lock**： 加锁，不论哪种类型的锁，都不可能被两个不同的线程同时得到，而必须等待解锁。 对于**普通锁**类型，解锁者可以是同进程内任何线程；

而**检错锁**则必须由加锁者解锁才有效，否则返回EPERM； 对于**嵌套锁**，文档和实现要求必须由加锁者解锁，但实验结果表明并没有这种限制，这个不同目前还没有得到解释。在同一进程中的线程，如果加锁后没有解锁，则任何其他线程都无法再获得锁。

**pthread\_mutex\_unlock**： 根据不同的锁类型，实现不同的行为：

对于快速锁，pthread\_mutex\_unlock 解除锁定；

对于递归锁，pthread\_mutex\_unlock 使锁上的引用计数减1；

对于检错锁，如果锁是当前线程锁定的，则解除锁定，否则什么也不做。

**pthread\_mutex\_trylock**：语义与pthread\_mutex\_lock()类似，不同的是，在锁已经被占据时则会返回EBUSY，而不是挂起等待。

（5）互斥锁属性，互斥锁属性的 结构体定义如下：

**typedef struct**

**{**

**int \_\_mutexkind;**  //注意这里是两个下划线

**}pthread\_mutexattr\_t;**

互斥锁的属性在创建锁的时候指定，在Linux Threads 实现中, 锁类型属性仅有一个即\_\_mutexkind，不同的锁类型在试图对一个已经被锁定的互斥锁加锁时表现不同，即是否有阻塞等待。有三个值可供选择：

·**PTHREAD\_MUTEX\_TIMED\_NP**， 这是**缺省值**（直接写**NULL** 就是表示这个缺省值），也就是普通锁(或快速锁)。 当一个线程加锁以后，其余请求锁的线程将形成一个阻塞等待队列，并在解锁后按优先级获得锁。这种锁策略保证了资源分配的公平性。

·**PTHREAD\_MUTEX\_RECURSIVE\_NP**， **嵌套锁**，允许同一个线程对同一个锁成功获得多次，并通过多次unlock 解锁。如果是不同线程请求，则在加锁线程解锁时重新竞争。

·**PTHREAD\_MUTEX\_ERRORCHECK\_NP**， **检错锁**，如果同一个线程请求同一个锁，则返回

EDEADLK，否则与PTHREAD\_MUTEX\_TIMED\_NP 类型动作相同。 这样就保证当不允许多次加锁时不会出现最简单情况下的死锁：即若**锁的类型是普通锁时，一个线程加锁之后，又加锁，则此时就是死锁**。

（6）加锁注意事项

1．**线程中如果一个锁在加锁后、解锁前，线程就被取消cancel了，则锁将会永远保持锁定状态**。 即线程被cancel了，也不会去自动解锁，所以其他线程要再加此锁时，就会加不上、死锁了。 因此，如果**在关键区段内有取消点存在时，必须定义 退出回调函数pthread\_cleanup\_push/pthread\_cleanup\_pop ，并在其清理函数中来解锁**。 同时不应该在信号处理函数中使用互斥锁，否则容易造成死锁。

2. 重点： 任何线程加了锁，其他线程都是可以直接进行解锁操作的。

6. 线程的取消 pthread\_cancel()：异步的，对应清理push/pop

（1） **线程也可被其它线程杀掉，Linux 中是一个线程被另一个线程取消(cancel)**。线程取消的方法：**一个线程向目标线程发cancel 信号，目标线程并不会马上终止掉（因为是异步的）**，**目标线程要运行到 cancelation-point(取消点) 后才会终止**。

如何处理cancel 信号是由目标线程自己决定：目标线程或忽略、或立即终止、或继续运行至cancelation-point取消点后终止。 注意1： cancel和信号一样是异步操作，其并不会马上删除，而是类似于删除共享内存 是标记性取消。

注意2： pthread\_cancel() 并不是会引起阻塞的接口（非阻塞接口）。

（2） 取消点有哪些？ (阻塞函数)

根据POSIX标准，**pthread\_join()**、**pthread\_testcancel()**、**pthread\_cond\_wait()**、

**pthread\_cond\_timedwait()**、**sem\_wait()**、**sigwait()**等函数、及 **read()类**、 **write()类等会引起阻塞的系统调用** 都是Cancelation-point取消点； 而其他pthread 函数都不会引起Cancelation 动作。 但是pthread\_cancel 的手册页声称，由于Linux 线程库与C 库结合得不好，因而目前C 库函数都不是Cancelation-point； 但CANCEL信号可以使线程从阻塞的系统调用中退出，并置EINTR 错误码，因此可在需要作为取消点的系统调用的前后面，通过调用pthread\_testcancel()，来达到POSIX 标准所要求的目标，即如下代码段：

**pthread\_testcancel();**

**retcode = read(fd, buffer, length);**

**pthread\_testcancel();**

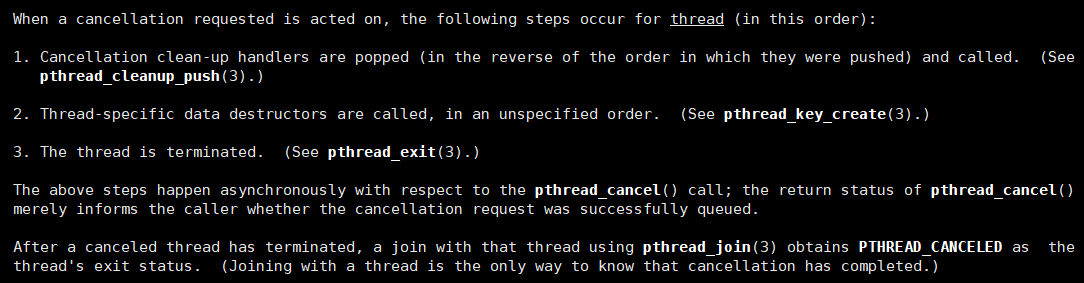
但是实际测试来看，至少有些C 库函数的阻塞函数是取消点，如**read()**, **getchar()**等，而**sleep()** 函数不管线程是否设置了pthread\_setcancelstate(PTHREAD\_CANCEL\_DISABLE,NULL)，都起到取消点作用。 总之，

**线程的取消是一个线程强行杀死另外一个线程**，从程序设计角度看并不是一种好的风格，另一方面目前Linux 本身对这方面的支持并不完善，所以在实际应用中应该谨慎使用。

**int pthread\_cancel(pthread\_t thread);**

（3） 因为**cancel是异步的**，类似标记性删除，并不是马上删除，所以**cancel一般都会成功，即返回0 ： 仅是指取消请求排队成功了**。  **若线程被cancel了，则通过 pthread\_join() 拿到的返回值是 PTHREAD\_CANCELED ： ((void \*) -1)**

（4） 当执行取消cancel请求时，线程将执行以下3步（内部实现）：



翻译：上述3个步骤 与 调用pthread\_cancel（）是异步发生; 即pthread\_cancel（）的返回值状态 仅仅是通知调用者 取消请求是否已排队成功。

7. 如何看pthread\_join(); 中的参数要强转为二级指针。

因为要改变原主线程中的值，故要取地址传过去（二级指针的传递）。

强制类型转换（如一级指针强转为二级指针）： 其不会改变程序的 执行逻辑（如引用；解引用）， 仅仅是为了编译不警告，让编译器看的。

8. **线程终止清理函数**： pthread\_cleanup\_push()/pthread\_cleanup\_pop()

（1） 不论是可预见的线程终止还是异常终止，都会存在资源释放的问题，在不考虑因运行出错而退出的前提下，如何保证线程终止时能顺利的释放掉自己所占用的资源，**特别是锁资源**，就是一个必须考虑解决的问题。 最经常出现的情形是 **资源独占锁的使用**：线程为了访问临界共享资源而为其加上锁，但在访问过程中该线程被外界取消，或者发生了中断，则该临界资源将永远处于锁定状态得不到释放。因为外界的取消操作是不可预见的，因此需要一个机制来简化用于资源释放的编程。

（2）在POSIX 线程API 中提供了一个 pthread\_cleanup\_push()/ pthread\_cleanup\_pop() 的**成对函数**，**用于自动释放资源**： **仅仅是从 pthread\_cleanup\_push()的调用点 到pthread\_cleanup\_pop() 之间的程序段中的终止动作，都将执行pthread\_cleanup\_push()所指定的清理函数**。 （下面的 **终止动作的三种场景，必须均要写在 push 、pop 中间**）

（3）API 定义如下： **用来把清理函数压栈**、**弹栈**

**void pthread\_cleanup\_push(void (\*routine) (void \*), void \*arg);**

**void pthread\_cleanup\_pop(int execute);**

参数解释如下：

pthread\_cleanup\_push()/pthread\_cleanup\_pop()采用**先入后出**的栈结构来进行管理：即**调用一次push 就会压栈一次清理函数**。

**void routine(void \*arg); 清理函数**： 在调用pthread\_cleanup\_push() 时，**压入清理函数栈**，多次对pthread\_cleanup\_push()的调用，则会在清理函数栈中形成一个函数链，在执行该函数链时会按照压栈的相反顺序弹出。

**arg** 参数： 要给清理函数传递的参数。

**execute** 参数：表示执行到 pthread\_cleanup\_pop()时是否在弹出清理函数的同时执行该函数：**为0 表示仅弹栈不执行**，**非0 表示弹栈并执行**； 此参数并不影响异常终止时清理函数的执行。pthread\_cleanup\_push()/pthread\_cleanup\_pop()是**以宏方式实现的,**以下是其在 pthread.h 中的宏定义：

**#define pthread\_cleanup\_push(routine,arg)\**

**{**

**struct \_pthread\_cleanup\_buffer \_buffer;\**

**\_pthread\_cleanup\_push (&\_buffer, (routine), (arg));**

**#define pthread\_cleanup\_pop(execute)\**

**\_pthread\_cleanup\_pop (&\_buffer, (execute));**

**}**

可见，pthread\_cleanup\_push()带有一个 "{" ， 而pthread\_cleanup\_pop()带有一个"}"， 故这**两个函数必须成对出现，且必须位于程序的同一级别的代码段中才能通过编译**

（4） push压栈的 清理函数，会得到执行的三种场景：

1. **线程要被cancel**时： 但不好确定cancel信号什么时候会到来，这是异步编程较难的地方。 注意：线程在取消点被cancel时，取消点后面的代码程序都不会再执行了

2．**pop(1)**时： 清理函数会弹出函数栈并执行；

3. 执行**pthread\_exit()**时：会自动去执行清理函数。

9. 线程的同步：类似于进程间用管道来实现同步

**条件变量： 利用线程间共享的全局变量来进行同步的一种机制**，**可以让多个线程都排队在一个条件变量上**。 一般是子线程等待在条件变量上，而主线程发信号让条件成立。

（**条件变量作用：当条件不满足时，阻塞当前线程**；直到满足条件时，线程被唤醒）

其主要包括两个动作：

（1）一个是 线程在等待条件变量的条件成立而挂起；

（2）另一个是 线程使条件成立（通过给出条件成立信号）。

为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起 （因为可能会有多个线程等待在此条件变量上，故 **wait前必须要先加锁**）

1、创建和注销

条件变量和互斥锁一样，都有静态、动态两种创建方式：

（1）静态方式使PTHREAD\_COND\_INITIALIZER 常量，如下：

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

（2）动态方式调用pthread\_cond\_init()函数，API 定义如下：

**int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t \*cond\_attr);**

尽管POSIX 标准中为条件变量定义了属性，但在Linux Threads 中并没有实现，因此参数**cond\_attr 的值通常为NULL**，且被忽略。

注销一个条件变量需要调用pthread\_cond\_destroy()， **只有在没有线程在该条件变量上等待的时候，才能注销这个条件变量，否则返回EBUSY**。 因为Linux 实现的条件变量没有分配什么资源，所以注销动作只包括检查是否有等待线程。API 定义如下：

**int pthread\_cond\_destroy(pthread\_cond\_t \*cond);**

2、等待和激发

（1）排队等待在条件变量上

等待条件有两种方式： 无条件等待pthread\_cond\_wait() 和 计时等待， 如下：

**int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);**  //是取消点

**int pthread\_cond\_timedwait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex, const struct timespec \*abstime);** //是取消点

注意事项：

1）wait内部关键机制： **线程 先解开mutex指向的锁**，**其再被条件变量cond 阻塞**。

2）计时等待方式表示：等待abstime 段时间后，即使条件变量不满足，阻塞也被解除

3）注意：timewait中 **abstime** 参数指的是： **超时 = 当前时间 + 多长时间超时**

4） **wait/timewait 内部实现机制**， 重点！！！（分为前半部、后半部）：

一、前半部：

（1）**排队在对应的条件变量上**；

（2）**解锁(故wait前必须先加锁)：为了让其他线程也能排队到此条件变量上**

（3）**放弃CPU去睡觉（被条件变量cond挂起阻塞）**

二、后半部：

（1）**被signal唤醒**

（2）**加锁**（**故醒来时锁依然是锁着的：为了调用wait的前后保持对应**）

(注意：内部加锁时可能还加不上：可能会被其他线程给抢走锁，则还要继续等，必须加上锁后，此线程才能继续往下运行)

5）无论哪种等待方式，都**必须和一个互斥锁配合，以防止多个线程同时请求pthread\_cond\_wait()**/pthread\_cond\_timedwait()的 竞争条件（Race Condition）.

6）参数mutex： 必须是普通锁（PTHREAD\_MUTEX\_TIMED\_NP）；

在pthread\_cond\_wait 前，用pthread\_mutex\_lock 进行加锁（且必须由本线程来加锁）；

而调用pthread\_cond\_wait 函数会先将锁解开； 然后将线程挂起阻塞，直到条件被 pthread\_cond\_signal 激发； 再将锁状态加锁恢复为锁定状态；

最后在调用完wait后，要再用pthread\_mutex\_unlock 进行解锁，来保持前后对应。

（2）激发条件有两种形式：

**pthread\_cond\_signal()** ： 激活一个等待在该条件上的线程，若存在多个等待线程时，则**按入队顺序激活其中的一个**。

注意：**pthread\_cond\_signal让条件成立，仅仅是调用的一瞬间成立，其并不是永久成立**；**且调用一次signal只能唤醒一个线程**。

**pthread\_cond\_broadcast()**： 激活所有等待线程，一次唤醒全部等待线程

3、注意事项

（1） pthread\_cond\_wait() 和 pthread\_cond\_timedwait()都被实现为取消点，也就是说如果pthread\_cond\_wait()被取消，则退出阻塞，然后将锁状态恢复，则此时mutex 是保持锁定状态的，而当前线程已经被取消掉，那么解锁的操作就会得不到执行，此时锁得不到释放，就会造成死锁，因而需要定义退出回调函数来为其解锁。

（2）标准的条件变量的使用方式：

回调函数保护（push、pop）、等待条件前加锁、 pthread\_cond\_wait()返回后解锁。

（3）

信号（处理函数中）： 只能自己**定义一个全局变量来传参**（因为信号处理函数中的形参是由内核来传参）； 互斥锁、条件变量等： 要用 **结构体 来传参**。

（4）重点技巧： 其实可以 **用一个进程（两个线程）来掌握 管道的读端和写端**， 来实现线程间同步。 一般用于两个线程间的同步（不用加解锁、条件变量， 用管道即可简单实现）

（5）条件变量机制不是异步信号安全的（因为使用到了全局变量）： 即 条件变量机制和互斥锁一样， 不能用于信号处理中， 在信号处理函数中调用 pthread\_cond\_signal() 或者 pthread\_cond\_broadcast() 很可能会引起死锁。

10. 线程的属性

（1） pthread\_create 的第二个参数attr 是一个结构体指针，结构中的元素分别指定新线程的运行属性,各成员属性为：

**\_\_detachstate**：表示新线程是否与进程中其他线程脱离同步，如果置位则新线程不能用pthread\_join()来同步，且在退出时自行释放所占用的资源。缺省为PTHREAD\_CREATE\_JOINABLE 状态。这个属性也可以在线程创建并运行以后用pthread\_detach()来设置，而一旦设置为PTHREAD\_CREATE\_DETACH状态（不论是创建时设置还是运行时设置）则不能再恢复到PTHREAD\_CREATE\_JOINABLE 状态。

**\_\_schedpolicy**：表示新线程的调度策略，主要包括SCHED\_OTHER（正常、非实时）、SCHED\_RR（实时、轮转法）和SCHED\_FIFO（实时、先入先出）三种，缺省为SCHED\_OTHER，后两种调度策略仅对超级用户有效。运行时可以用过pthread\_setschedparam()来改变。

**\_\_schedparam**：一个sched\_param 结构，目前仅有一个sched\_priority 整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即SCHED\_RR 或SCHED\_FIFO）时才有效，并可以在运行时通过pthread\_setschedparam()函数来改变，缺省为0。

**\_\_inheritsched**：有两种值可供选择：PTHREAD\_EXPLICIT\_SCHED 和PTHREAD\_INHERIT\_SCHED，前者表示新线程使用显式指定调度策略和调度参数（即attr 中的值），而后者表示继承调用者线程的值。缺省为PTHREAD\_EXPLICIT\_SCHED。

**\_\_scope**：表示线程间竞争CPU 的范围，也就是说线程优先级的有效范围。POSIX 的标准中定义了两个值：PTHREAD\_SCOPE\_SYSTEM 和PTHREAD\_SCOPE\_PROCESS，前者表示与系统中所有线程一起竞争CPU 时间，后者表示仅与同进程中的线程竞争CPU。目前Linux 仅实现了PTHREAD\_SCOPE\_SYSTEM 一值。

（2） 属性设置是由一些函数来完成的：

通常调用pthread\_attr\_init 函数进行初始化；

设置绑定属性的函数为pthread\_attr\_setscope；

设置分离属性的函数是pthread\_attr\_setdetachstate；

设置线程优先级的相关函数pthread\_attr\_getscehdparam（获取线程优先级）和pthread\_attr\_setschedparam（设置线程优先级）。

再设置完成属性后，调用pthread\_creat 函数创建线程。

·线程属性初始化：

int pthread\_attr\_init (pthread\_attr\_t \*attr);

attr：传出参数，表示线程属性，后面的线程属性设置函数都会用到。

返回值：成功0，错误-1。

·设置绑定属性：

pthread\_attr\_setscope(pthread\_attr\_t \*attr, init scope);

attr：线程属性

scope：PTHREAD\_SCOPE\_SYSTEM(绑定) PTHREAD\_SCOPE\_PRCESS(非绑定)

返回值：成功0，错误-1。

·设置分离属性:

pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, init detachstate);

attr：线程属性

detachstate ：PTHREAD\_CREAT\_DETACHED(分离) PTHREAD\_CREAT\_JOINABLE(非分离)

返回值：成功0，错误-1。

·获取线程优先级：

int pthread\_attr\_getschedparam(pthread\_attr\_t \*attr, struct sched\_param \*param);

attr：线程属性

param：线程优先级

返回值：成功0，错误-1。

·设置线程优先级：

int pthread\_attr\_setschedparam(pthread\_attr\_t \*attr, struct sched\_param \*param);

attr：线程属性

param：线程优先级

返回值：成功0，错误-1。

可用top查看线程(默认只能看进程)： top ，再按 ’H ’

11. 线程安全：

**可重入**函数（线程安全）： 没有使用到全局变量

**不可重入**函数（线程不安全）： 当一个函数使用到了 全局变量（地址值相同）

如： 例子ctime()返回的时间，是存在一个全局变量的数组中的。 故ctime()就不是线程安全的，要用ctime\_r() 可重入函数;