1. 不带缓冲io
   1. open

打开一个文件，可创建文件，返回文件描述符

#include<fcntl.h>

int open(const char\* path,int oflag,.../\*mode\_t mode\*/);

path:文件名

oflag:打开方式，如读写打开

mode:访问权限，如用户读

* 1. close

关闭一个文件，成功返回0，失败返回-1

#include<unistd.h>

int close(int fd);

fd:文件描述符

进程关闭一个文件会释放它加在该文件上的所有记录锁。

当一个进程终止时，内核会自动关闭它所有的打开的文件。

* 1. lseek

设置文件偏移量,返回新的文件偏移量，失败返回-1

#include<unistd.h>

off\_t lseek(int fd, off\_t offset,int whence);

fd:文件描述符

offset：偏移量

whence：偏移模式，如SEEK\_SET表示将该文件的偏移量设置为距离文件开始处`offset`个字节

* 1. read

读取文件

#include<unistd.h>

ssize\_t read(int fd,void \*buf,size\_t nbytes);

fd:文件描述符

buf:读取内容存放缓冲区

nbytes：读取长度

成功返回读取长度

失败返回-1

* 1. write

写入文件

#include<unistd.h>

ssize\_t write(int fd,const void \*buf,size\_t nbytes);

fd：文件描述符

buf：写入内容

nbytes；写入长度

成功返回写入长度

失败返回-1

* 1. 文件共享

内核使用三种数据结构描述打开文件。它们之间的关系决定了一个进程与另一个进程在打开的文件之间的相互影响。

- 内核为每个进程分配一个进程表项（所有进程表项构成进程表），进程表中都有一个打开的文件描述符表。每个文件描述符占用一项，其内容为：

- 文件描述符标志

- 指向一个文件表项的指针

- 内核为每个打开的文件分配一个文件表项（所有的文件表项构成文件表）。每个文件表项的内容包括：

- 文件状态标志（读、写、添写、同步和阻塞等）

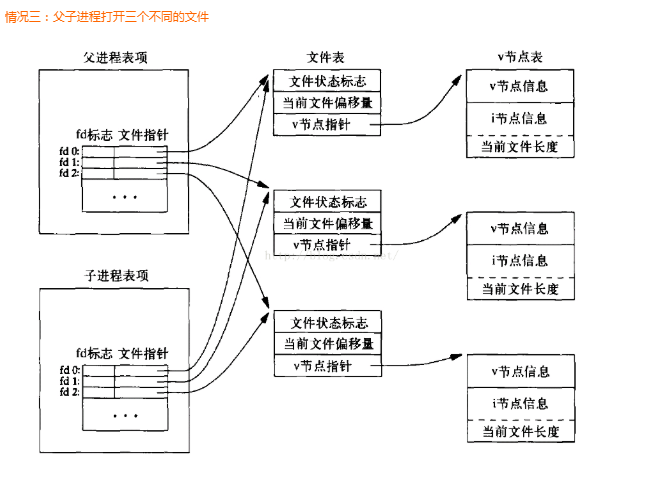
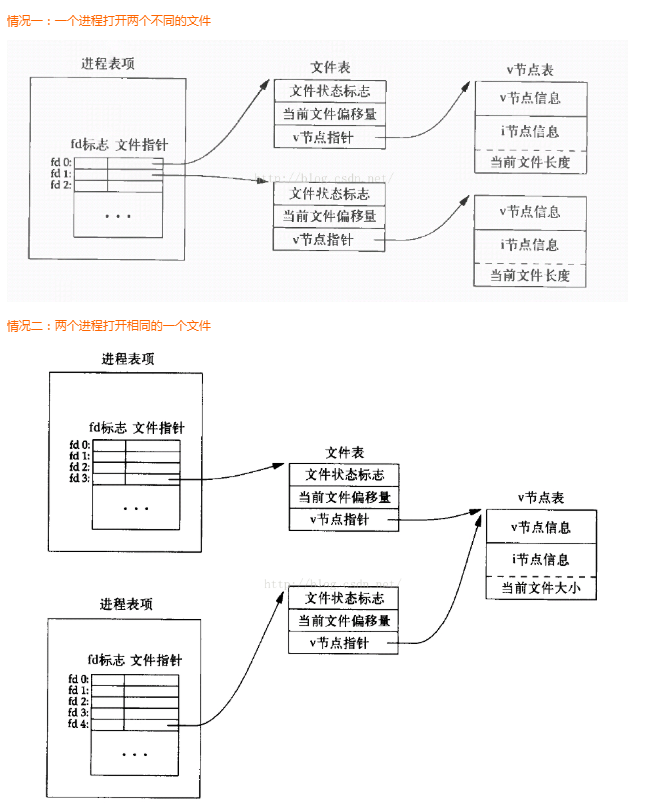
- 当前文件偏移量

- 指向该文件 v 结点表项的指针

- 每个打开的文件或者设备都有一个 v 结点结构。 v 结点结构的内容包括：

- 文件类型和对此文件进行各种操作函数的指针。

- 对于大多数文件， v 结点还包含了该文件的 i 结点。



* 1. sync,fsync

#include<unistd.h>

int fsync(int fd);

int fdatasync(int fd);

void sync(void);

fd：文件描述符

成功返回0

失败返回-1

- `sync`：将所有修改过的块缓冲区排入写队列，然后返回，它并不等待时机写磁盘结束

- `fsync`：只对由`fd`指定的单个文件起作用，等待写磁盘操作结束才返回

1. 标准库io
   1. 缓冲

标准IO库提供了三种类型的缓冲：

- 全缓冲：此时在标准IO缓冲区被填满后，标准IO库才进行实际的IO操作。

- 行缓冲：此时当输入和输出中遇到换行符时，标准IO库执行实际的IO操作。但是注意：

- 只要填满了缓冲区，即使还没有写一个换行符，也立即进行IO操作

- 任何时候只要通过标准IO库，从一个不带缓冲的流或者一个行缓冲的流得到输入数据，则会冲洗所有行缓冲输出流。即要缓冲输入，先冲洗输出缓冲

- 不带缓冲：标准IO库不对字符进行缓冲存储。此时任何IO都立即执行实际的IO操作。

`setbuf/setvbuf`函数：设置流的缓冲类型

#include<stdio.h>

void setbuf(FILE \*restrict fp,char \*restrict buf);

int setvbuf(FILE \*restrict fp,char\* restrict buf,int mode,size\_t size);

手动冲洗一个流置内核

#include<stdio.h>

int fflush(FILE \*fp);

fp:文件流，为空时冲洗所有

成功返回0

失败返回EOF

* 1. 打开关闭文件（流）

打开文件流

#include<stdio.h>

FILE \*fopen(const char\*restrict pathname,const char\*restrict type);从名字打开流

FILE \*freopen(const char\*restrict pathname,const char\*restrict type, FILE \*restrict fp);

FILE \*fdopen(int fd,const char\*type);从文件描述符打开流

pathname：文件名

type:打开方式，如读写打开

fd:文件描述符

关闭文件流

#include<stdio.h>

int fclose(FILE \*fp);

成功返回0，失败返回-1

从流获取文件描述符

#include<stdio.h>

int fileno(FILE \*fp);

* 1. 读写流
     1. 按字符读写

读取

#include<stdio.h>

int getc(FILE\*fp);

int fgetc(FILE\*fp);

int getchar(void);

- 参数：

- `fp`：打开的文件对象指针

- 返回值：

- 成功：则返回下一个字符

- 到达文件尾端：返回`EOF`

- 失败：返回`EOF`

`getchar()`等价于`getc(stdin)`

`ungetc`函数：将字符压回流中，只压入缓冲区，不写到磁盘

#include<stdio.h>

int ungetc(int c,FILE \*fp);

c:压入字符

fp:文件流

成功返回字符c，失败返回EOF

写入

#include<stdio.h>

int putc(int c,FILE\*fp);

int fputc(int c,FILE\*fp);

int putchar(int c);

成功返回c,失败返回EOF

putchar(c)等价于putc(c,stdout)

* + 1. 按行读写

读取一行

#include<stdio.h>

char \*fgets(char \*restrict buf,int n, FILE\* restrict fp);

char \*gets(char \*buf);

buf:缓冲区

n：缓冲区长度

fp:文件流

成功返回buf,

到达文件尾返回NULL

失败返回NULL

推荐使用fgets，gets没有缓冲区长度，读取一行，该行大于缓冲区长度时缓冲区溢出

写一行

#include<stdio.h>

int fputs(const char\* restrict str,FILE\*restrict fp);

int puts(const char\*str);

str以NULL结尾，NULL不写入

puts写完后会写入换行符；fputs不会，换行符需在字符串str中

* + 1. 二进制io

#include<stdio.h>

size\_t fread(void \*restrict ptr,size\_t size,size\_t nobj,FILE \*restrict fp);

size\_t fwrite(const void\*restrict ptr,size\_t size,size\_t nobj,FILE \*restrict fp);

ptr:读写地址

size:读写对象长度

nobj:读写对象个数

fp:流

成功或失败返回读写对象个数

读写时可能因为内存对齐存在异常

* 1. 临时文件

#include<stdio.h>

char \*tmpnam(char \*ptr);//获取文件路径名

FILE \*tmpfile(void);//获取临时文件流

* 1. 其它函数

创建临时文件

#include<stdlib.h>

char \*mkdtemp(char \*template);

int mkstemp(char \*template);

内存流

#include<stdio.h>

FILE \*fmemopen(void \*restrict buf,size\_t size,const char \*restrict type);

#include<stdio.h>

FILE \*open\_memstream(char \*\*bufp,size\_t \*sizep);

#include <wchar.h>

FILE \*open\_wmemstream(wchar\_t \*\*bufp,size\_t \*sizep);

1. 进程环境
   1. 退出函数

进程正常终止：main返回，调用exit，调用\_exit或\_Exit,最后一线程返回，最后一线程调用pthread\_exit.

进程异常终止：调用abort,接收信号，最后一线程对取消请求作出响应

#include<stdlib.h>

void exit(int status);

void \_Exit(int status);

#include<unistd.h>

void \_exit(int status);

exit先清理（冲洗流，关闭流），再返回内核

\_Exit和\_exit直接返回内核

命令行echo $?查看进程返回值，就是参数status值，retuan的值

atxeit函数

#include<stdlib.h>

int atexit(void (\*func) (void));

可登记函数，进程退出时自动调用

登记多次，调用多次；登记顺序和调用顺序相反，先登记，后调用

* 1. main 参数

int main(int argc,char\* argv[])

argc:参数个数

argv:参数数组，第一个参数为程序路径，默认argv[argc]为NULL

* 1. 环境表

环境指针extern char\*\* environ

环境格式 PATH=/bin:

获取单一环境：getenv(“PATH”);

* 1. 堆分配

void \*malloc(size\_t size);//初始值未定义

void \*calloc(size\_t nobj,size\_t size);//nobj对象个数，size对象长度，内存初始化为0

void \*realloc(void \*ptr,size\_t newsize);//增加ptr长度, ptr可能无效值，返回新地址

* 1. setjmp ，longjmp

跨函数goto

#include<setjmp.h>

int setjmp(jmp\_buf env);

void longjmp(jmp\_buf env,int val);

1. 进程控制
   1. 进程ID pid

交换进程（系统进程），0，内核一部分

init进程，1，不会终止，所有孤儿进程的父进程

页守护进程，2，

#include<unistd.h>

pid\_t getpid(void); // 返回值：调用进程的进程ID

pid\_t getppid(void); // 返回值：调用进程的父进程ID

* 1. fork

#include<unistd.h>

pid\_t fork(void);

子进程返回0，父进程返回子进程pid,失败返回-1

子进程拷贝父进程数据空间，堆栈，

使用标准输出，fork前缓冲区有数据时，父子进程会有两份缓冲数据，冲洗后会输出两份相同的数据。

fork父子进程复制文件描述符，不复制文件表项，只有一个文件偏移量

两次fork能解决僵尸进程问题。

* 1. wait waitpid

#include<sys/wait.h>

pid\_t wait(int \*staloc);

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

调用wait时子进程已终止，立即返回；子进程未终止，调用线程阻塞，待子进程终止后返回。返回子进程pid，staloc未子进程终止状态。

waitpid可等待指定进程，可不阻塞直接返回

* 1. exec

进程执行新的程序，当前进程的正文段、数据段、堆段和栈段丢失

#include<unistd.h>

int execl(const char \*pathname,const char \*arg0,.../\*(char \*) 0 \*/);

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

int execle(const char \*pathname,const char \*arg0,.../\*(char \*) 0

,char \*const envp[] \*/);

int execve(const char \*pathname,char \*const argv[],char \*const envp[]);

int execlp(const char \*filename,const char\*arg0,.../\*(char \*) 0\*/);

int execvp(const char \*filename, char \*const argv[]);

int fexecve(int fd,char \*const argv[],char \*const evnp[]);

* 1. 解释器

解释器文件：文本文件，以#！开头，后接解释器路径，后接参数

解释器：解释器文件第一行#！后面的字符串（可执行文件），

exec可调用解释器文件，运行解释器

* 1. system

#include<stdlib.h>

int system(const char \*cmdstring);

`system`用于将一个字符作为命令来执行。它等同于同时调用了`fork、exec、waitpid`。有三种返回值：

- `fork`失败或者`waitpid`返回除了`EINTR`之外的错误，则`system`返回 -1，并且设置`errno`以指示错误类型

- 如果`exec`失败(表示不能执行`shell`)，则其返回值如同`shell`执行了`exit(127)`一样

- 如果三个函数都执行成功，则`system`返回值是`shell`的终止状态

* 1. 进程调度 nice

#include<unistd.h>

int nice(int incr);

设置进程优先级，进程nice值+incr,nice增加，优先级降低

特权进程才可以降低nice值

#include<sys/resource.h>

int getpriority(int which,id\_t who);

int setpriority(int which,id\_t who,int value);

设置，获取指定进程nice值

1. 进程关系
   1. 登录终端

init进程读取文件`/etc/inittab`，对每个终端设备fork一次，调用getty,等待登录。

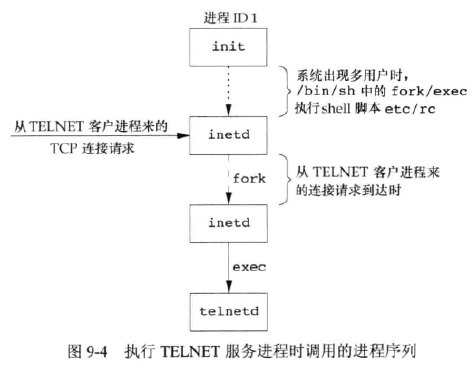
收入用户名后getty终止，调用login

* 1. 网络登录

在`BSD`系统中，由`init`执行`shell`脚本`/etc/rc`，此`shell`脚本启动`inetd`守护进程。由`inetd`负责处理网络登录

- 在`Linux`系统中，使用`xinetd`代替`inetd`进程

xinetd收到网络登录请求，fork一个telnetd,打开一个伪终端，fork为两个进程，父进程处理网络通信，子进程exec login.



* 1. 进程组

#include<unistd.h>

pid\_t getpgrp(void);

pid\_t getpgid(pid\_t pid);

获取进程组id,getpgrp()==getpgid(0);

进程组组长进程id==进程组id

#include<unistd.h>

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

pid进程加入进程组pgid,

若pid==pgid,pid变为进程组组长

pid=0,使用调用者进程id

pgid=0,进程组id为pid

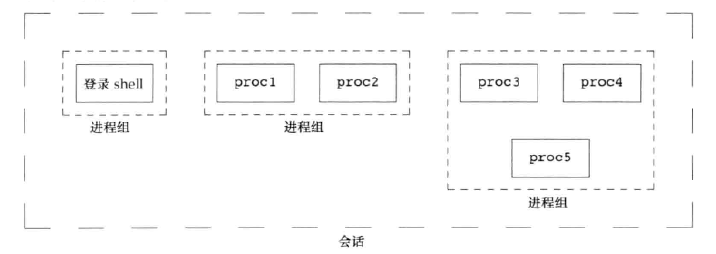
一个进程只能为它自己或者他的子进程设置进程组ID，且进程组ID只能为父进程进程组ID、父进程的进程ID或者子进程的进程ID。

- 在它的子进程调用`exec`之后，它就不再更改子进程的进程组ID

* 1. 会话

会话`session`是一个或者多个进程组的集合。

执行命令：proc1|proc2&proc3|proc4|proc5形成会话



#include<unistd.h>

pid\_t setsid(void);

进程调用`setsid`建立一个新会话。如果调用此函数的进程不是一个进程组的组长进程，则此函数创建一个新会话并且发生下面三件事：

- 该进程会变成新会话的会话首进程`session leader`。此时该进程是新会话中的唯一进程

> 会话首进程是创建该会话的进程

- 该进程成为一个新进程组的组长进程。新进程组ID就是该调用进程的进程ID

- 该进程没有控制终端。即使调用`setsid`之前该进程有一个控制终端，该联系也被切断

如果调用此函数的进程是个进程组的组长，则此函数返回出错。

> 通常是进程首先`fork`，然后父进程终止，子进程调用`setsid`继续执行。这确保了子进程不是一个进程组的组长

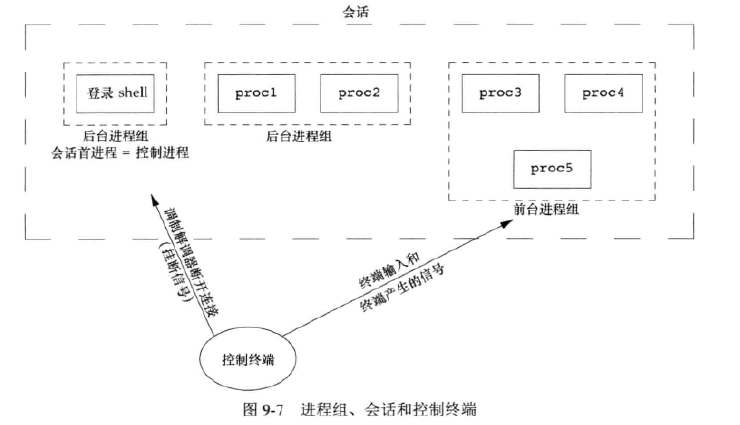
#include<unistd.h>

pid\_t getsid(pid\_t pid);

会话首进程id可视为会话id,pid=0返回调用进程会话首进程id(会话首进程进程组id)

* 1. 控制终端

一个会话可以有一个控制终端，



* 1. 设置前台进程组

2. `tcgetpgrp/tcsetpgrp`函数：获取/设置当前进程所在会话的前台进程组ID

```

#include<unistd.h>

pid\_t tcgetpgrp(int fd);

int tcsetpgrp(int fd,pid\_t pgrpid);

```

- 参数：

- `fd`：进程在`fd`这个描述符上打开的终端

- `pgrpid`：待设置的前台进程组ID

- 返回值：

- 对于`tcgetpgrp`：成功则返回前台进程组ID，失败返回 -1

- 对于 `tcsetpgrp`：成功返回 0；失败返回 -1

如果进程有一个控制终端，则该进程可以调用`tcsetpgrp`将前台进程组ID设置为`pgrpid`，其中：

- `pgrpid`必须是同一个会话的一个进程组的ID

- `fd`必须引用该会话的控制终端

注意：大多数应用程序并不直接使用这两个函数，它们通常是由作业控制`shell`调用

* 1. 作业控制

支持作业控制可分前台进程组后后台进程组，

后台进程组等待输入时，后台进程组停止作业，转为前台进程组时继续作业

命令：fg %1

后台进程组等待输出时，可由stty tostop设置，禁止输出时作业停止，否则直接输出

* 1. 孤儿进程组

孤儿进程组：一个进程组中的所有进程：

>

> - 要么其父进程不再同一个会话中

> - 要么其父进程就在同一个组中

孤儿进程组产生的时候，如果孤儿进程组中有`TASK\_STOP`的进程，那么就发送`SIGHUP`和`SIGCONT`信号给这个进程组

- 这个顺序是不能变的。我们知道进程在进程`在TASK\_STOP`的时候是不能响应信号的，只有当进程继续运行的时候，才能响应之前的信号。

- 如果先发送`SIGCONT`信号再发送`SIGHUP`信号，那么`SIGCONT`信号后，进程就开始重新进入运行态，这个和马上响应`SIGHUP`信号的用意相悖

- 所以这个时候需要在进程`TASK\_STOP`的过程中首先发送`SIGHUP`信号，为的是让进程运行之后马上执行`SIGHUP`信号。

- 这两个信号是发送给有处于`TASK\_STOP`状态的进程的进程组的所有进程的。所以进程组中正在运行的进程，如果没有建立`SIGHUP`信号处理函数，那么运行的进程就会因为`SIGHUP`退出。

1. 信号
   1. 常用信号

SIGALRM 定时器超时

SIGINT 终端中断符 delete或Ctrl+c

SIGQUIT 终端退出符 ctrl+\

SIGKILL 终止 不能捕捉或忽略

SIGSTOP 停止进程 不能捕捉或忽略

SIGCONT 继续进程

SIGSEGV 无效内存引用

SIGUSR1，SIGUSR2 用户自定义信号

SIGCHLD ,SIGCLD子进程终止或停止时发给父进程

信号处理函数 SIG\_IGN 忽略信号 SIG\_DFL 系统默认函数

* 1. signal函数

设置一个信号的处理函数，信号发生时，中断当前执行动作，执行信号处理函数，完成后返回，继续执行中断前动作

#include<signal.h>

void (\*signal(int signo,void (\*func)(int)))(int);

typedef void Sigfunc(int signo)

Sigfunc \* signal(int signo,Sigfunc func);

signo 要处理的信号编号

func: SIG\_IGN(忽略该信号) SIG\_DFL(系统默认动作)

其它可未函数指针

成功返回原信号处理函数指针

失败返回SIG\_ERR

* 1. kill raise

#include<signal.h>

int kill(pid\_t pid,int signo); //向进程pid发信号

int raise(int signo); //向自身发信号

raise(signo)=kill(getpid(),signo);

* 1. alarm pause

#include<unistd.h>

unsigned int alarm(unsigned int seconds); //定时器 只产生一次SIGALRM信号

int pause(void); //休眠，等待信号唤醒

* 1. 信号函数

sigset\_t //信号集

#include<signal.h>

int sigemptyset(sigset\_t \*set); //清空信号集

int sigfillset(sigset\_t \*set); //填满信号集

int sigaddset(sigset\_t \*set,int signo); //添加信号

int sigdelset(sigset\_t \*set,int signo); //删除信号

int sigismember(const sigset\_t \*set,int signo); //判断包含信号

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

set为空，how无意义

set不空

how: SIG\_BLOCK 阻塞set信号，进程信号集为set和原信号集并集

how=SIG\_UNBLOCK 解除阻塞set信号，进程信号集为set和原信号集补集的并集

how=SIG\_SETMASK 进程信号集为set

oset 若非空，返回当前信号集

int sigpending(sigset\_t \*set); //返回当前信号集

int sigaction(int signo,const struct sigaction \*restrict act,struct sigaction\*restrict oact);

signo:要处理的信号编号

act:若非空，信号处理动作

oact:若非空 上一信号处理动作

struct sigaction{

void (\*sa\_handler)(int); //可以为 SIG\_IGN、SIG\_DFL或者信号处理函数的地址

sigset\_t sa\_mask; //额外需要屏蔽的信号，执行sa\_ handler时阻塞sa\_mask，sa\_ handler返回时恢复进程原信号集，可实现执行信号处理函数时避免信号中断信号处理函数

int sa\_flags; // 标志位

void (\*sa\_sigaction)(int,siginfo\_t \*,void\*); //可选的另一种信号处理函数

}

int sigsuspend(const sigset\_t \*sigmask); //用于原子性的修改信号屏蔽字然后进程睡眠

* 1. 信号名与编号

extern char \*sys\_siglist[] //信号名数组

void psignal(int signo,const char\*msg);//输出信号名，有空行

char \*strsignal(int signo); //输出信号名

1. 线程
   1. 线程互斥量属性
      1. 互斥量共享属性

默认为PTHREAD\_PROCESS\_PRIVATE，多个线程共享互斥量

PTHREAD\_PROCESS\_SHARED，多个进程共享互斥量

* + 1. 互斥量健壮（鲁棒）属性

默认PTHREAD\_MUTEX\_STALLED,当持有互斥量进程终止后，其它进程无法再获取该互斥量，

PTHREAD\_MUTEX\_ROBUST，如果其中某个进程在未释放锁的情况下退出了，另一个进程仍然可以获得锁，但是此时 pthread\_mutex\_lock 将返回 EOWNERDEAD，通知获得锁的线程，有一个其它进程的线程挂掉了，互斥量现在变成了 inconsistent 的状态。这时候，需要对互斥量做 consistent 处理，否则，一旦再次解锁后，互斥量将永久不可用。

* + 1. 互斥量类型属性

PTHREAD\_MUTEX\_NORMAL：一种标准互斥量类型，不做任何特殊的错误检查或死锁检查

PTHREAD\_MUTEX\_ERRORCHECK：此类型互斥量进行错误检查

PTHREAD\_MUTEX\_RECURSIVE：此类型互斥量运行同一线程在互斥量解锁之前对该互斥量进行多次加锁（维持了一个加锁的计数）。当解锁次数和加锁次数不相同的情况下，并不会释放锁

PTHREAD\_MUTEX\_DEFAULT：此类型互斥量提供默认特性和行为。操作系统在实现的时候可以将它映射到上面三类中的任何一种。linux默认为PTHREAD\_MUTEX\_NORMAL

* 1. 读写锁属性

只规定一个属性，和互斥量共享属性一样

默认为PTHREAD\_PROCESS\_PRIVATE，多个线程共享读写锁

PTHREAD\_PROCESS\_SHARED，多个进程共享读写锁

* 1. 条件变量属性

进程共享属性：条件变量的进程共享属性与互斥量的进程共享属性是相同的。

时钟属性：控制计算pthread\_cond\_timedwait函数的超时参数`tsptr`时，采用的是哪个时钟

* 1. 屏障属性

屏障支持的唯一属性是进程共享属性。它与互斥量的进程共享属性是相同的

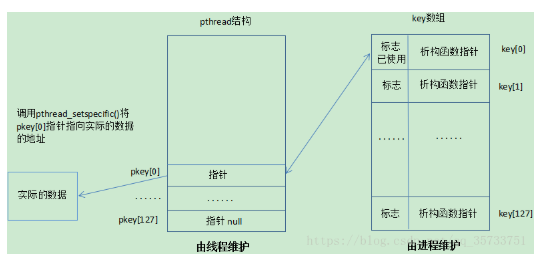
* 1. 线程私有数据

  POSIX规定为每个进程维护了一个key结构的容器（可以理解为一个数组），通过这个数组可以获取或保存线程私有数据，这个数组中的每个结构称之为一个线程特定数据元素，POSIX规定系统实现的Key结构数组必须包含不少于128个线程特定元素。

   每个线程的线程特定数据元素至少包含两项内容：标志和析构函数指针，key结构的的标志表示这个数组元素是否使用，析构函数指针表示线程退出时释放线程特定数据。

   当一个线程调用pthread\_key\_create函数创建一个key时，系统就会从key数组中关联一个未使用的key。创建的key通过pthread\_key\_create函数的参数keyp指针返回该key的地址，第二个参数是一个函数指针，指向一个析构函数。

   除了进程内key数组，系统还为进程内的每个线程维护了一个线程结构，也就是pthread结构。pthread结构维护的pkey指针是和进程的key数组的所有key相关联，指针指向的内存是线程特定数据。



* 1. 线程取消选项

pthread\_cancel函数可对线程发出取消请求

* + 1. 可取消状态

线程收到取消请求，若可取消状态为PTHREAD\_CANCEL\_ENABLE，线程到达取消点时会执行取消操作，若取消状态为PTHREAD\_CANCEL\_DISABLE,取消请求被挂起

一下系统函数会出现取消点，pthread\_testcancel可手动添加取消点

#include<pthread.h>

void pthread\_testcancel(void);

* + 1. 取消类型

#include<pthread.h>

int pthread\_setcanceltype(int type,int \*oldtype);

设置取消类型

PTHREA\_CANCEL\_DEFERRED，延迟取消，即到达取消点取消，默认设置

PTHREAD\_CANCEL\_ASYNCHRONOUS，异步取消，不用到达取消点，直接取消

* 1. 线程和信号

在多线程进程中，信号的处理会随机选择一线程处理

每个线程都有自己的信号屏蔽字，pthread\_sigmask函数可修改线程信号屏蔽字

sigwait等待线程信号，调用线程阻塞

pthread\_kill可将信号发送给线程

闹钟定时器是进程资源，并且所有的线程都共享相同的闹钟

* 1. 线程和fork

子进程通过继承整个地址空间的副本，还从父进程那儿继承了每个互斥量、读写锁和条件变量的状态

如果父进程包含了一个以上的线程，则子线程内部，只存在一个线程，它是由父进程中调用`fork`的线程的副本构成的

pthread\_atfork处理父子进程锁的问题

#include<pthread.h>

int pthread\_atfork(void (\*prepare)(void),void (\*parent)(void),void (\*child)(void));

- `prepare`：一个函数指针。该函数在父进程`fork`创建子进程之前调用

- `parent`:一个函数指针。该函数在`fork`返回之前在父进程的上下文中调用

- `child`：一个函数指针。该函数在`fork`返回之前在子进程的上下文中调用

`pthread\_atfork`是父进程在调用`fork`之前调用的。它给出了`fork`前后可以执行的三个清理函数。你可以在其中清理锁。此时：

- 在`prepare`处理程序中获取所有的锁（因此潜在的条件是：所有实用这些锁的线程都释放了锁）

- 在`parent`处理程序中释放锁

- 在`child`处理程序中释放锁