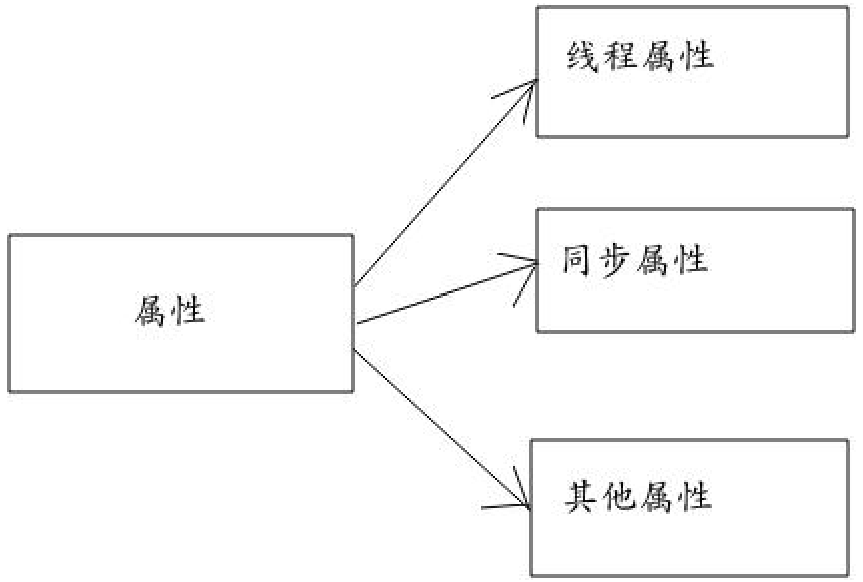
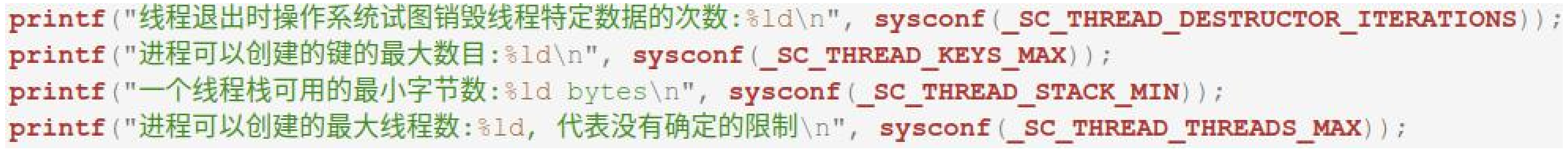
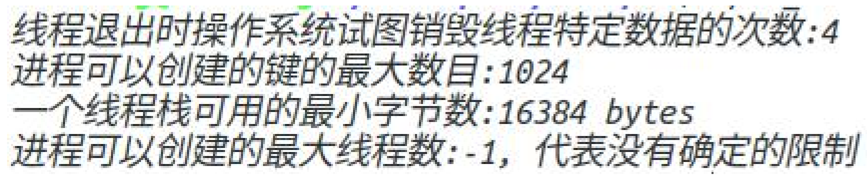

属性
线程属性
同步属性
其他属性

线程资源系统限制



```c
printf("线程退出时操作系统试图销毁线程特定数据的次数:%ld\n", sysconf(_SC_THREAD_DESTRUCTOR_ITERATIONS));
printf("进程可以创建的键的最大数目:%ld\n", sysconf(_SC_THREAD_KEYS_MAX));
printf("一个线程栈可用的最小字节数:%ld bytes\n", sysconf(_SC_THREAD_STACK_MIN));
printf("进程可以创建的最大线程数:%ld, 代表没有确定的限制\n", sysconf(_SC_THREAD_THREADS_MAX));
```

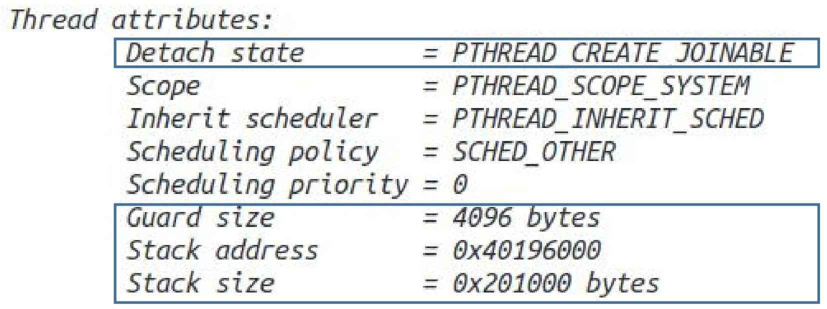


```
线程退出时操作系统试图销毁线程特定数据的次数:4
进程可以创建的键的最大数目:1024
一个线程栈可用的最小字节数:16384 bytes
进程可以创建的最大线程数:-1, 代表没有确定的限制
```

线程属性

1. 应用位置：pthread_create的第二个参数

2. 线程属性的内容



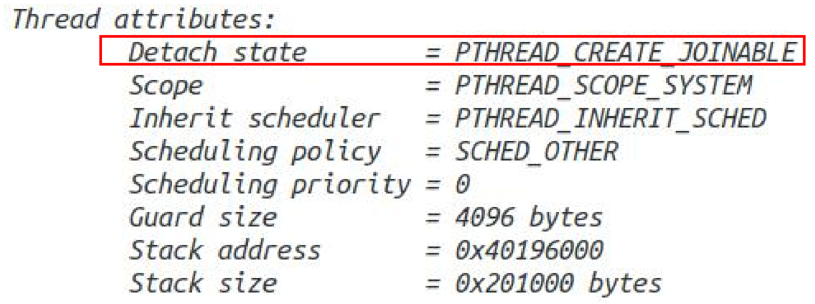
```
Thread attributes:
        Detach state        = PTHREAD_CREATE_JOINABLE
        Scope               = PTHREAD_SCOPE_SYSTEM
        Inherit scheduler   = PTHREAD_INHERIT_SCHED
        Scheduling policy   = SCHED_OTHER
        Scheduling priority = 0
        Guard size          = 4096 bytes
        Stack address       = 0x40196000
        Stack size          = 0x201000 bytes
```

3. 线程属性初始化与销毁

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destory(pthread_attr_t *attr);
```

//成功，返回0，　失败，返回错误编号

线程属性



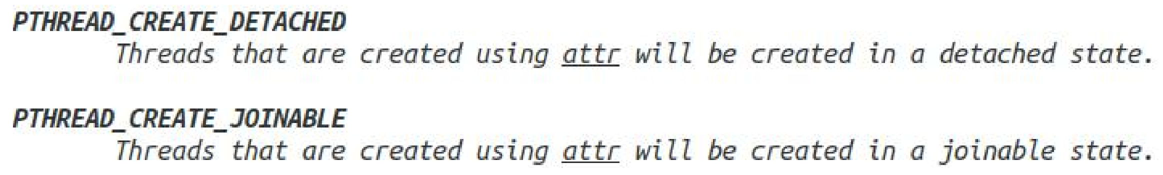
```
Thread attributes:
        Detach state        = PTHREAD_CREATE_JOINABLE
        Scope               = PTHREAD_SCOPE_SYSTEM
        Inherit scheduler   = PTHREAD_INHERIT_SCHED
        Scheduling policy   = SCHED_OTHER
        Scheduling priority = 0
        Guard size          = 4096 bytes
        Stack address       = 0x40196000
        Stack size          = 0x201000 bytes
```

1. 设置线程属性_分离状态

2. 分离态：对现有的某个线程并不关心其结束状态，可以设置其为分离态，这样其会结束后自动回收资源。

3. 线程分离属性的设置和获取

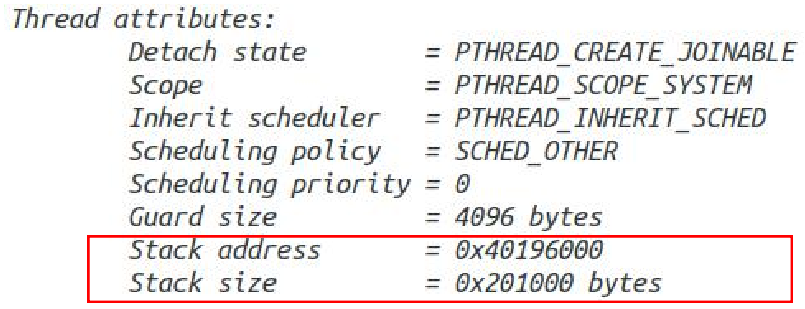
```
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int *detachstate);
//成功，返回 0 ，失败，返回错误编码
```



```
PTHREAD_CREATE_DETACHED
       Threads that are created using attr will be created in a detached state.

PTHREAD_CREATE_JOINABLE
       Threads that are created using attr will be created in a joinable state.
```

线程属性



```
Thread attributes:
        Detach state        = PTHREAD_CREATE_JOINABLE
        Scope               = PTHREAD_SCOPE_SYSTEM
        Inherit scheduler   = PTHREAD_INHERIT_SCHED
        Scheduling policy   = SCHED_OTHER
        Scheduling priority = 0
        Guard size          = 4096 bytes
        Stack address       = 0x40196000
        Stack size          = 0x201000 bytes
```

1. 设置线程属性_线程栈属性_栈最低地址和栈大小。

2. 为什么需要设置，加入虚拟内存空间中的栈地址用完了，可以使用这个把新开辟的线程安排到别的地方。

3. 线程栈属性的设置和获取

```
int pthread_attr_setstack(pthread_attr_t *attr,
                          void *stackaddr, size_t stacksize);
int pthread_attr_getstack(const pthread_attr_t *attr,
                          void **stackaddr, size_t *stacksize);
```

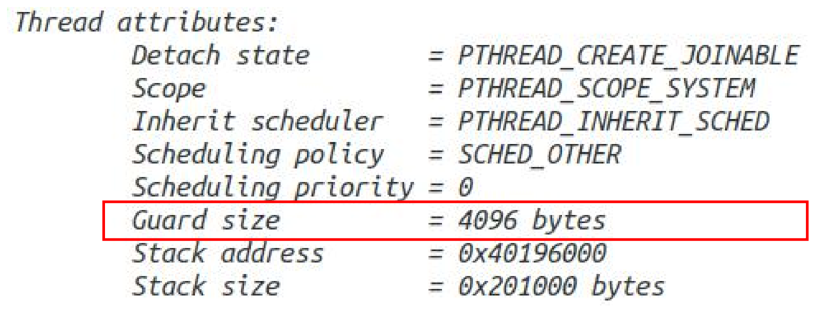
//成功，返回 0 ，失败，返回错误编码

4. 只修改线程栈的大小，但是不干涉线程栈地址的分配，采用下面的方法

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize);
```

//成功，返回 0 ，失败，返回错误编码

线程属性



```
Thread attributes:
        Detach state        = PTHREAD_CREATE_JOINABLE
        Scope               = PTHREAD_SCOPE_SYSTEM
        Inherit scheduler   = PTHREAD_INHERIT_SCHED
        Scheduling policy   = SCHED_OTHER
        Scheduling priority = 0
        Guard size          = 4096 bytes
        Stack address       = 0x40196000
        Stack size          = 0x201000 bytes
```

1. 设置线程属性_线程栈之间警戒缓冲区的大小

2. 为什么需要设置，如果线程栈溢出，不会直接破坏临近线程的栈，而是进入缓冲区，并且程序接收到错误信号

3. 线程警戒缓冲区属性的设置和获取

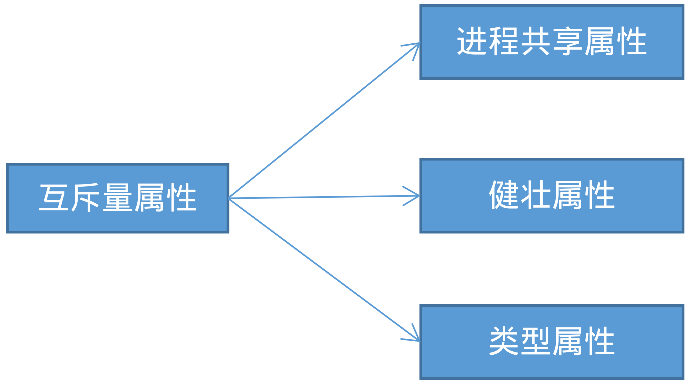
```
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
int pthread_attr_getguardsize(const pthread_attr_t *attr, size_t *guardsize);
```

//成功，返回 0 ，失败，返回错误编码

同步属性_互斥量属性

1．互斥量属性的初始化和销毁

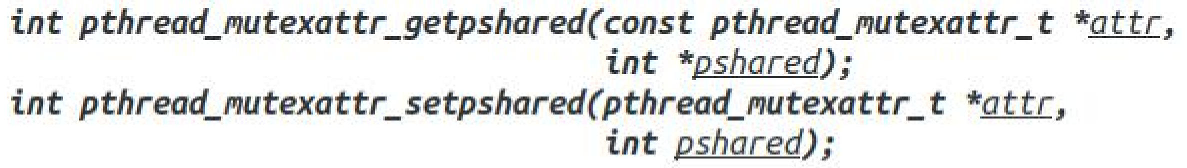
```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destory(pthread_mutexattr_t *attr);
//成功，返回 0 ，失败，返回错误编码
```



同步属性_互斥量属性

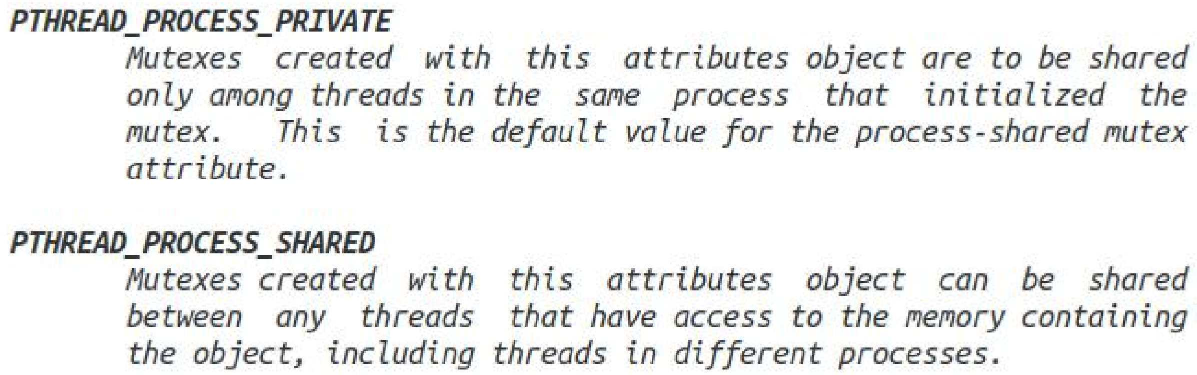
进程共享属性

1. 互斥量属性—进程共享属性



```
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr,
                                 int *pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                 int pshared);
```

//成功，返回 0 ，失败，返回错误编码



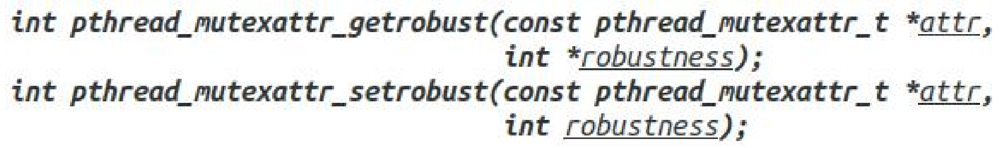
PTHREAD_PROCESS_PRIVATE
Mutexes created with this attributes object are to be shared only among threads in the same process that initialized the mutex. This is the default value for the process-shared mutex attribute.

PTHREAD_PROCESS_SHARED
Mutexes created with this attributes object can be shared between any threads that have access to the memory containing the object, including threads in different processes.

同步属性_互斥量属性

健壮属性

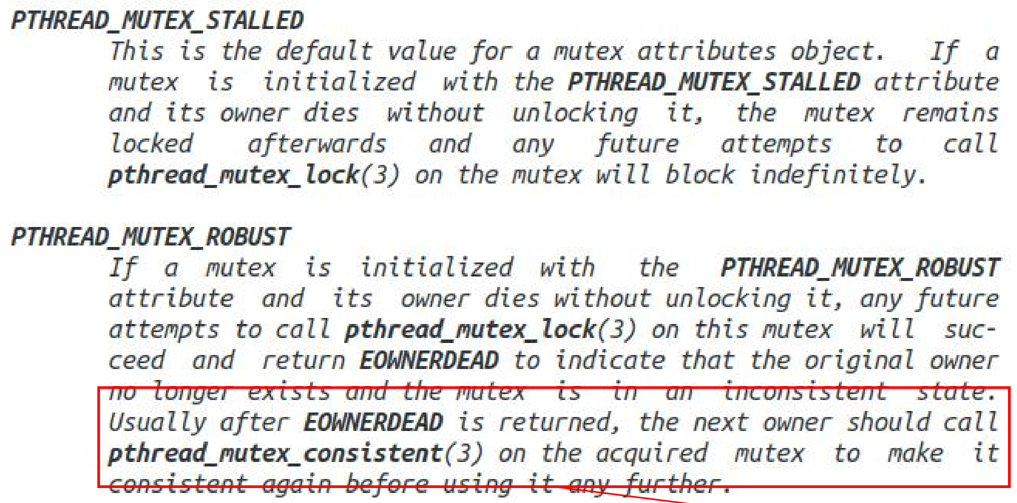
1． 互斥量属性——健壮属性



```
int pthread_mutexattr_getrobust(const pthread_mutexattr_t *attr,
                                int *robustness);
int pthread_mutexattr_setrobust(const pthread_mutexattr_t *attr,
                                int robustness);
```

//成功，返回 0 ，失败，返回错误编码

健壮属性用在描述互斥量被多个进程所共享，然后被某个进程持有但是该进程死亡依旧没有释放后该如何去应对。



PTHREAD_MUTEX_STALLED
This is the default value for a mutex attributes object. If a mutex is initialized with the ***PTHREAD_MUTEX_STALLED*** *attribute and its owner dies without unlocking it, the mutex remains locked afterwards and any future attempts to call* ***pthread_mutex_lock****(3) on the mutex will block indefinitely.*

PTHREAD_MUTEX_ROBUST
If a mutex is initialized with the ***PTHREAD_MUTEX_ROBUST*** *attribute and its owner dies without unlocking it, any future attempts to call* ***pthread_mutex_lock****(3) on this mutex will succeed and return* ***EOWNERDEAD*** *to indicate that the original owner no longer exists and the mutex is in an inconsistent state. Usually after* ***EOWNERDEAD*** *is returned, the next owner should call* ***pthread_mutex_consistent****(3) on the acquired mutex to make it consistent again before using it any further.*

如果获取到锁后得到的返回值是EOWNERDERD,表示状态无法恢复， 在这种情况下，你如果对其进行解锁，会导致互斥量处于永远不可用状态。需要手动将互斥量状态进行恢复：

```
int pthread_mutex_consistent(pthread_mutex_t *mutex);
```

同步属性_互斥量属性

类型属性

1. 互斥量属性——类型属性

```
int pthread_mutexattr_settype(pthread_mutexattr_t *__attr, int __kind)
int pthread_mutexattr_gettype(pthread_mutexattr_t *__attr, int __kind)
```

//成功，返回 0 ，失败，返回错误编码

PTHREAD_MUTEX_NORMAL
//一种标准的互斥量类型，不提供错误检查和死锁检测，这个是默认的特性
PTHREAD_MUTEX_RECURSIVE
//此互斥量允许同一线程在互斥量解锁前对其进行多次加锁，然后维护锁的计数，解锁次数要和加锁次数对应
PTHREAD_MUTEX_ERRORCHECK
//提供错误类型检测，就是在你进行不合理操作[例如对未持有的锁进行解锁]会返回错误
PTHREAD_MUTEX_DEFAULT
//默认特性和行为，这个这样叫，但是不是默认的特性

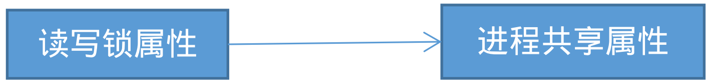
在LINUX下的pthread的源码中，发现PTHREAD_MUTEX_DEFAULT = PTHREAD_MUTEX_NORMAL

同步属性_读写锁属性

1. 读写锁属性初始化和销毁

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *__attr)
int pthread_rwlockattr_destory(pthread_rwlockattr_t *__attr)
```

//成功，返回 0 ，失败，返回错误编码



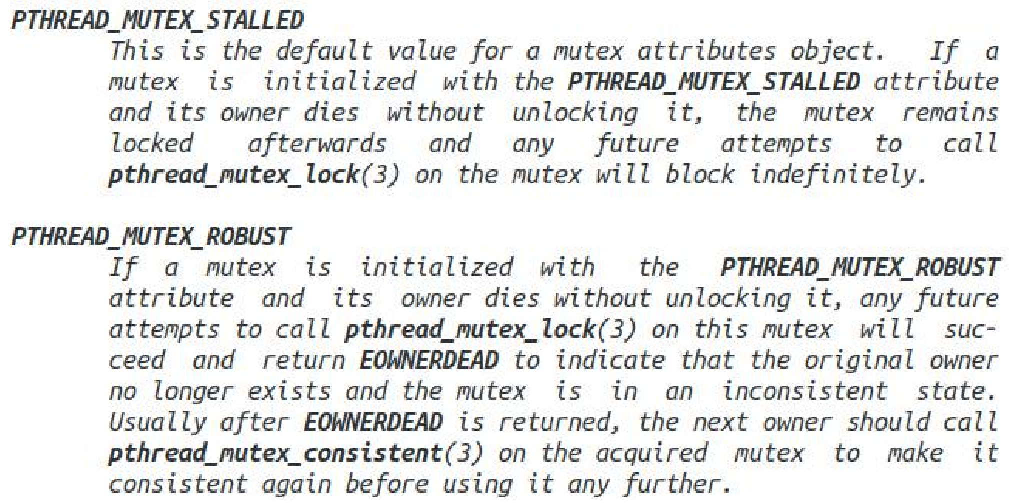
同步属性_读写锁属性

进程共享属性

1. 读写锁属性—进程共享属性

```
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int *pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

//成功，返回 0 ，失败，返回错误编码



PTHREAD_MUTEX_STALLED
    This is the default value for a mutex attributes object. If a mutex is initialized with the **PTHREAD_MUTEX_STALLED** attribute and its owner dies without unlocking it, the mutex remains locked afterwards and any future attempts to call **pthread_mutex_lock**(3) on the mutex will block indefinitely.

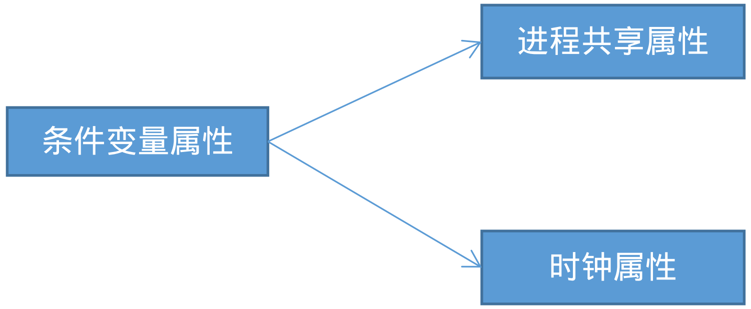
PTHREAD_MUTEX_ROBUST
    If a mutex is initialized with the **PTHREAD_MUTEX_ROBUST** attribute and its owner dies without unlocking it, any future attempts to call **pthread_mutex_lock**(3) on this mutex will succeed and return **EOWNERDEAD** to indicate that the original owner no longer exists and the mutex is in an inconsistent state. Usually after **EOWNERDEAD** is returned, the next owner should call **pthread_mutex_consistent**(3) on the acquired mutex to make it consistent again before using it any further.

同步属性_条件变量属性

1．条件变量属性初始化和销毁

```
int pthread_condattr_init(pthread_condattr_t *__attr)
int pthread_condattr_destory(pthread_condattr_t *__attr)
```

//成功，返回 0 ，失败，返回错误编码



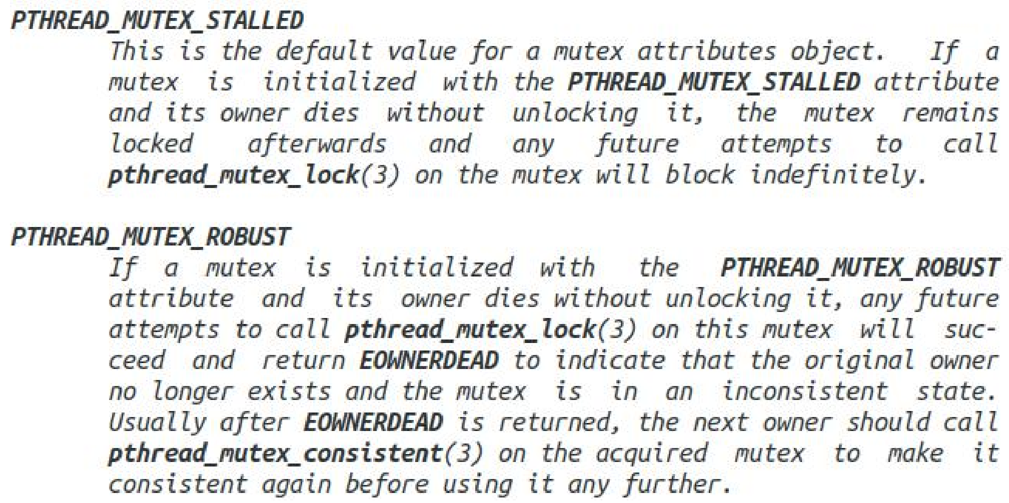
同步属性_条件变量属性

进程共享属性

1. 条件变量属性_进程共享属性

```
int pthread_condattr_getpshared(const pthread_condattr_t *__attr, int *pshared);
int pthread_condattr_setpshared(pthread_condattr_t *__attr, int pshared);
```

//成功，返回 0 ，失败，返回错误编码



```
PTHREAD_MUTEX_STALLED
       This is the default value for a mutex attributes object.  If a
       mutex  is  initialized  with the PTHREAD_MUTEX_STALLED attribute
       and its owner dies  without  unlocking  it,  the  mutex  remains
       locked    afterwards    and    any    future    attempts    to    call
       pthread_mutex_lock(3) on the mutex will block indefinitely.

PTHREAD_MUTEX_ROBUST
       If  a  mutex  is  initialized  with   the   PTHREAD_MUTEX_ROBUST
       attribute  and  its  owner dies without unlocking it, any future
       attempts to call pthread_mutex_lock(3) on this mutex  will  suc-
       ceed  and  return EOWNERDEAD to indicate that the original owner
       no longer exists and the mutex  is  in  an  inconsistent  state.
       Usually after EOWNERDEAD is returned, the next owner should call
       pthread_mutex_consistent(3) on the acquired  mutex  to  make  it
       consistent again before using it any further.
```

同步属性_条件变量属性

时钟属性

1. 条件变量属性_时钟属性——控制pthread_cond_timedwait的定时采用哪一个时钟

```
int pthread_condattr_getclock(const pthread_condattr_t *__attr, clockid_t *clock_id);
int pthread_condattr_setclock(pthread_condattr_t *__attr, clockid_t clock_id);
```

//成功，返回 0 ，失败，返回错误编码

同步属性_屏障属性

1. 屏障属性初始化和销毁

```
int pthread_barrierattr_init(pthread_barrierattr_t *__attr)
int pthread_barrierattr_destory(pthread_barrierattr_t *__attr)
```

//成功，返回 0 ，失败，返回错误编码

条件变量属性
进程共享属性

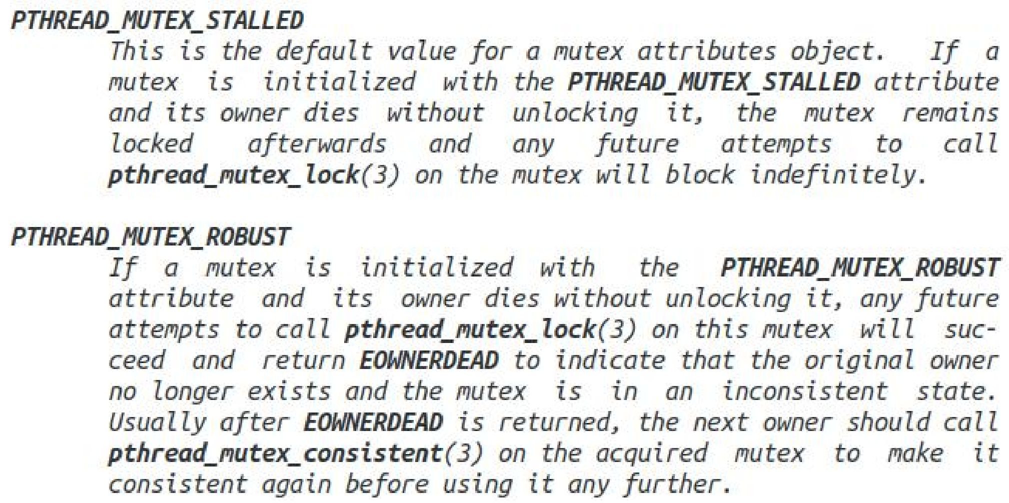
同步属性_屏障属性

进程共享属性

1. 屏障属性_进程共享属性

```
int pthread_barrierattr_getpshared(const pthread_barrierattr_t *__attr, int *pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *__attr, int pshared);
```

//成功，返回 0 ，失败，返回错误编码



PTHREAD_MUTEX_STALLED
This is the default value for a mutex attributes object. If a mutex is initialized with the ***PTHREAD_MUTEX_STALLED*** *attribute and its owner dies without unlocking it, the mutex remains locked afterwards and any future attempts to call* ***pthread_mutex_lock****(3) on the mutex will block indefinitely.*

PTHREAD_MUTEX_ROBUST
If a mutex is initialized with the ***PTHREAD_MUTEX_ROBUST*** *attribute and its owner dies without unlocking it, any future attempts to call* ***pthread_mutex_lock****(3) on this mutex will succeed and return* ***EOWNERDEAD*** *to indicate that the original owner no longer exists and the mutex is in an inconsistent state. Usually after* ***EOWNERDEAD*** *is returned, the next owner should call* ***pthread_mutex_consistent****(3) on the acquired mutex to make it consistent again before using it any further.*

重入

1. 什么叫可重入？

   可以重复进入而不产生问题，或者说可以被多个不同的任务单元同时进行访问。

   如果某个函数对于多个线程来说是可重入的，就称为是线程安全的。

   如果函数对于异步信号处理程序来说是可重入的，就称为是异步信号安全的。

线程私有数据

1. 概念
 是存储和查询某个特定线程相关数据的一种机制，我们希望每个线程可以访问它自己单独的数据副本，而不需要担心与其他线程的同步访问问题。

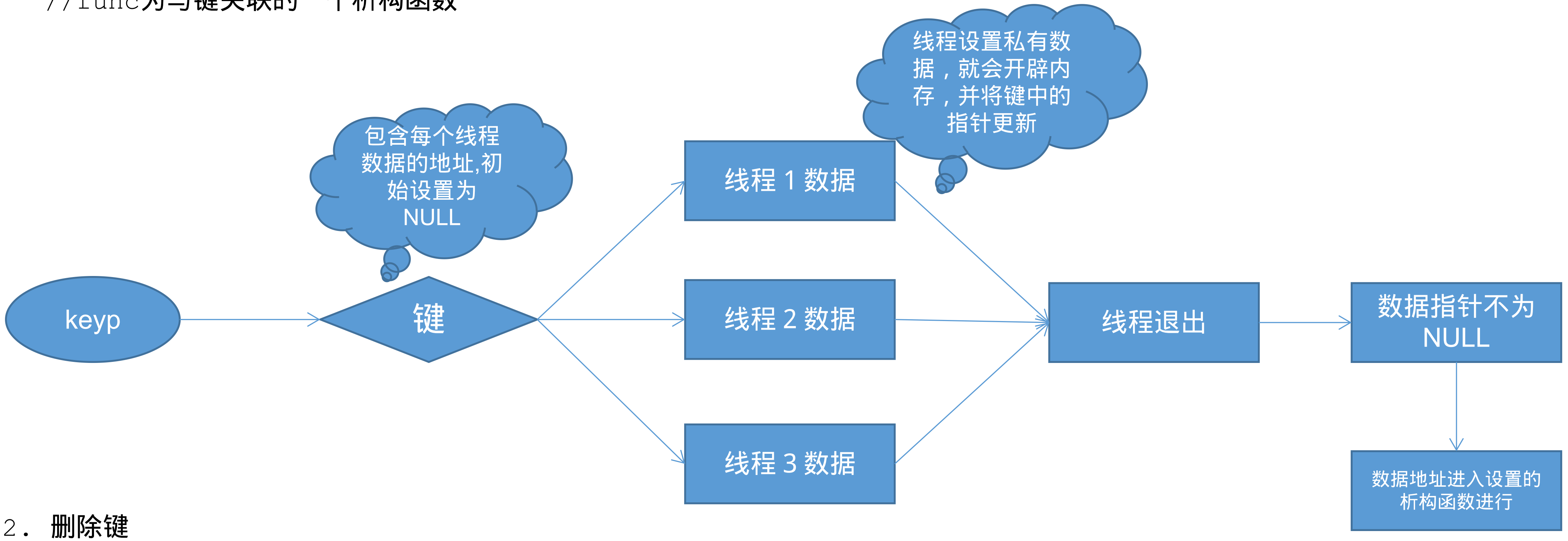
2. 线程当初有方便数据共享的优点，这里为什么又需要数据的私有？
 2.1 有时候需要维护基于每个线程的数据，而线程ID范围跨度太大，不适合用来做数据索引
 2.2 提供了让基于进程的接口适应多线程环境，使得不同线程之间不影响

3. 理论上来说，由于线程可以访问这个进程地址空间中的所有合法地址，不可能实现线程数据的私有，但是采用某些线程数据管理函数可以进行管理。

线程私有数据

1． 创建线程私有数据键

```
int pthread_key_create(pthread_key_t *keyp, void (*func)(void*));
//成功，返回 0 ，失败，返回错误编号
//键与线程私有数据关联，用于获取对线程特定数据的访问
//func为与键关联的一个析构函数
```

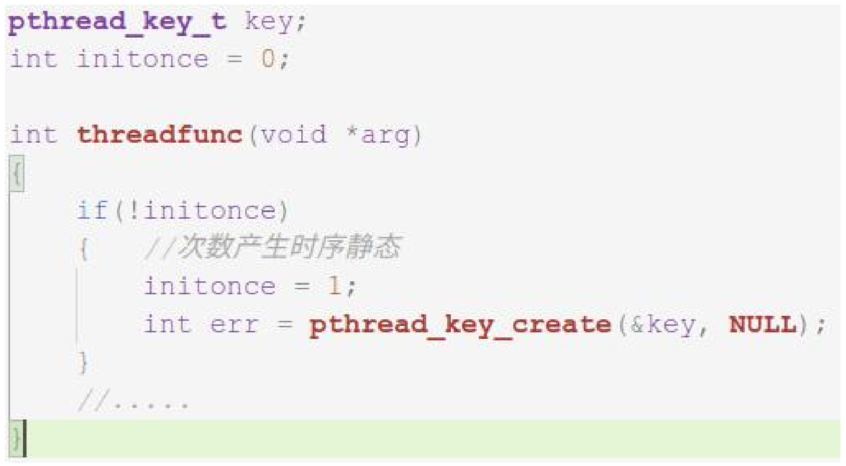

线程设置私有数据，就会开辟内存，并将键中的指针更新
包含每个线程数据的地址,初始设置为NULL
keyp
键
线程 1 数据
线程 2 数据
线程 3 数据
线程退出
数据指针不为NULL
数据地址进入设置的析构函数进行

2． 删除键

```
int pthread_key_delete(pthread_key_t key);
//成功，返回 0 ，失败，返回错误编号
//不会激活数据的析构函数
```

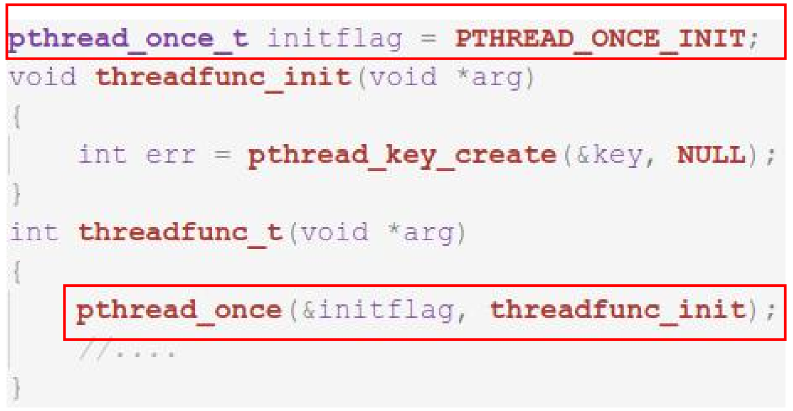
线程私有数据

3. 键初始化的时候的竞态



```
pthread_key_t key;
int initonce = 0;

int threadfunc(void *arg)
{
    if(!initonce)
    {   //次数产生时序静态
        initonce = 1;
        int err = pthread_key_create(&key, NULL);
    }
    //.....
}
```



```
pthread_once_t initflag = PTHREAD_ONCE_INIT;
void threadfunc_init(void *arg)
{
    int err = pthread_key_create(&key, NULL);
}
int threadfunc_t(void *arg)
{
    pthread_once(&initflag, threadfunc_init);
    //....
}
```

```
int pthread_once(pthread_once_t *initflag, void (*initfn)(void));
```
//成功，返回 0 ，失败，返回错误编号
//把检查标志和标志设置合并成原子操作
//pthread_once_t 必须是一个全局变量或者静态变量，且必须被初始化为PTHREAD_ONCE_INIT

线程私有数据

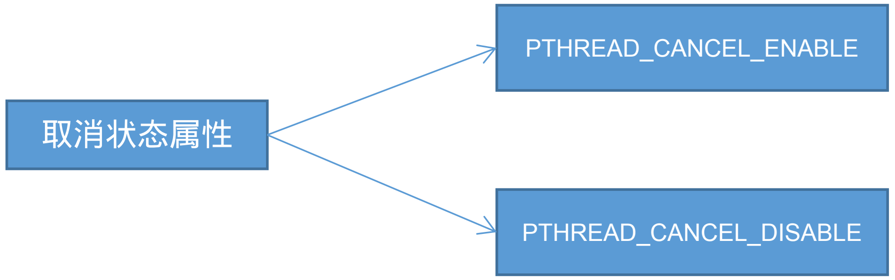
4．通过键访问数据

```
void *pthread_getspecific(pthread_key_t key);
//返回线程特定数据值，如果没有，返回NULL
int pthread_setspecific(pthread_key_t key, const void *value);
//成功，返回 0 ，失败，返回错误编号
```

线程属性扩充——取消状态属性[如何响应pthread_cancel]

1. 设置取消状态属性_有效和无效

```
int pthread_setcancelstate(int newstate, int *oldstate);
//成功，返回 0 ，失败，返回错误编号
```



默认值，代表取消请求可以直接被发送给线程，线程在接下来的取消点进行终止

将取消请求挂起，等到取消属性再一次被设置为默认值，才会接受，进行默认动作

2. 用户设置取消点

```
void pthread_testcancel(void); //如果取消状态被设置为无效，就不起作用
```

3. 设置取消类型_推迟还是立即

```
int pthread_setcanceltype(int type, int *oldtype);
//成功，返回 0 ，   失败，返回错误编号
 //PTHREAD_CANCEL_DEFERRED  和  PTHREAD_CANCEL_ASYNCHRONOUS
```

线程和信号_复杂的作用机制

线程拥有自己的信号屏蔽字，但是共享信号处理方式

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```
//成功，返回 0 ，失败，返回错误编号

进程中的信号递送给单个线程，如果是硬件导致的故障，该信号发送给引起故障的线程，否则，发送给任意一个线程

```
int sigwait(const sigset_t *set, int *sig);
```
//成功，返回 0 ，失败，返回错误编号

四个问题：

- sigwait工作基本内容？
    - 等待set中的一个或者多个信号到达，并且将sig中设置为发送信号的数量
- sigwait的使用规范？
    - 如果在set中的信号处于挂起状态，那么调用sigwait后，会无阻塞返回，并且将该信号从未决信号集合中剔除，若支持排队，那么只会剔除一个，其他的还要继续排队
- 同时使用sigaction和sigwait怎么办？
    - 由系统决定，选择其一，不会两个都收到
- 再一次理解一下进程资源的含义。
    - 例如时钟就是进程资源，不属于任何一个线程，sigaction也是针对于进程设置的，不针对与任何一个单独的线程

线程和信号_复杂的作用机制

发送信号给某个线程

```
int pthread_kill(pthread_t thread, int sig);
```

//成功，返回 0 ，失败，返回错误编号
//发送一个 0 来测试线程是否存在
//如果信号的动作是结束该进程，那么信号传递给某个线程依旧会杀死进程[因为信号的处理手段是进程资源，和线程无关，线程可以使用sigwait从进程手里偷(会将未决信号集合中的信号去掉一个,或者系统会把一个到达的信号发送给sigwait)信号自己去做处理(sigwait之后做的一些操作)]

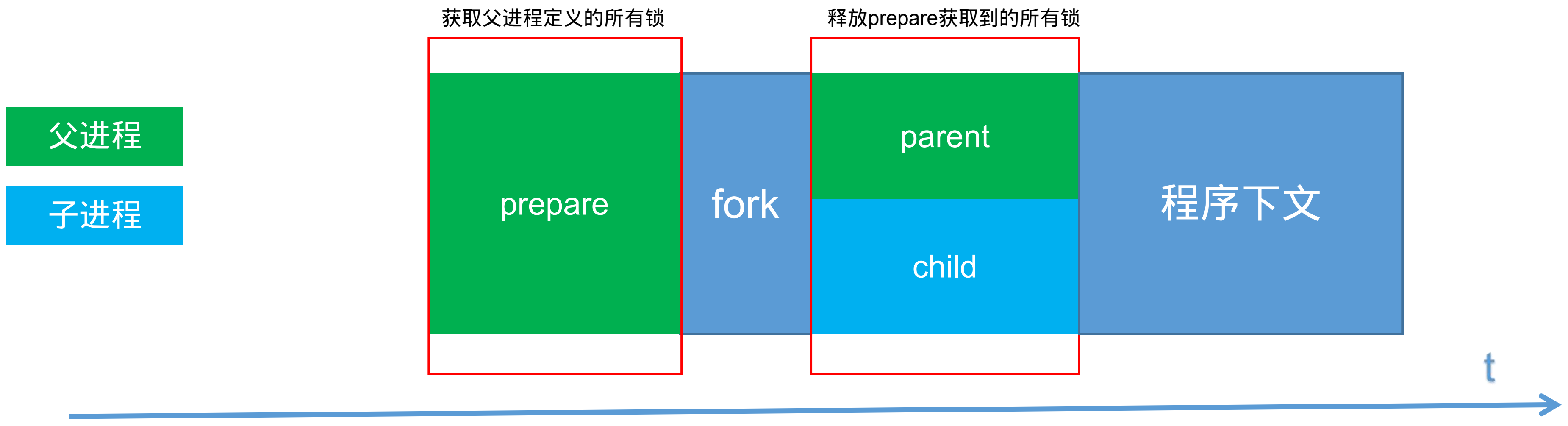
线程fork之后的行为

1. fork之后，会继承每个互斥量　、读写锁和条件变量的状态，　调用exec会清理这些锁，　否则，如果父进程包括一个以上的线程，而且没有紧接着调用execl，就有必要清理锁状态

2. 在某个线程执行fork之后，子进程只存在一个线程，就是fork的调用线程，　此时子进程没办法知道谁占有锁和需要释放什么锁。

3. 建立fork处理程序来清理锁状态

```
int pthread_atfork(void(*prepare)(void), void (*parent)(void), void (*child)(void));
//成功，返回 0 ，失败，返回错误编码
//可以多次调用，注册多组函数，其中prepare调用顺序和注册顺序相反，parent和child调用顺序和注册顺序相同
```


获取父进程定义的所有锁
释放prepare获取到的所有锁
父进程
子进程
prepare
fork
parent
child
程序下文
t