线程引入带来的好处

1. 简化异步事件的处理
2. 相比于进程数据共享，线程共享更为简单
3. 将多个任务交叉进行，提高整个程序的吞吐量
4. 程序功能分离，便于程序设计

线程标识

1. 线程ID

与进程ID不同，线程ID只有在所属的上下文中才有意义

```
#include <pthread.h>

pthread_t pthread_self(void);

Compile and link with -pthread.
```

```
DESCRIPTION
       The pthread_self() function returns the ID of the calling thread.
```

获取线程ID

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);

Compile and link with -pthread.
```

```
DESCRIPTION
       The pthread_equal() function compares two thread identifiers.
```

比较线程ID

线程创建

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

成功，返回0， 失败，返回错误编号

attr: 线程属性， 设置为NULL， 表明使用默认属性

start_routine: 线程的函数地址

arg： 传送给线程该函数的参数

创建一个新的线程后，不保证那一个线程会先运行，新线程可以访问线程的地址空间，并且继承调用进程的浮点环境和信号屏蔽字

线程终止

整个进程退出

进程中任一一个线程调用exit, _exit, _Exit

主控线程使用return

任一一个线程接收到终止进程的信号

单个线程退出

非主控线程return

被其他进程取消

线程调用pthread_exit

不存在主控线程单独退出的情况， 主控线程退出后，整个进程都将退出

线程终止

```
void pthread_exit(void *rval_ptr);
//退出线程，rval_ptr中是返回的一个数据指针， 可以用pthread_join来接收

int pthread_join(pthread_t thread, void **rval_ptr);
//成功，返回0， 否则，返回错误编号

int pthread_cancel(pthread_t tid);
// 返回值：若成功， 返回0， 否则， 返回错误编码
// 只是向线程提出了一个终止的请求，线程可以选择忽略和控制被取消的过程
```

pthread_join

1. 一直阻塞， 直到指定的线程调用pthread_exit， 从启动历程返回或者被取消
2. 如果线程简单的返回，那么rval_ptr中包含返回码，如果线程被取消，那么rval_ptr指向的内存单元设置为PTHREAD_CANCELED
3. 如果线程处于分离态， 那么pthread_join将会失败， 此时返回EINVAL。
4. 可以设置rval_ptr为NULL， 等待线程终止，但是并不关心线程状态

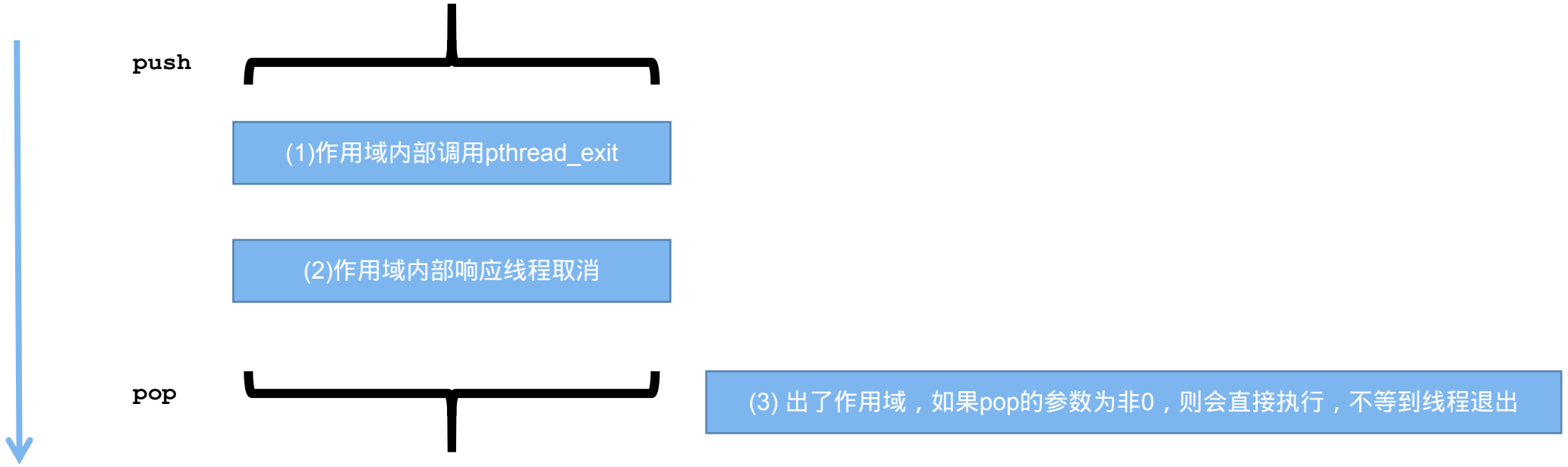
```c
void * thread_func(void *p)
{
    int *i = (int *)malloc(sizeof(int));
    *i = 10;
    return (void*)i;
}

int main()
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread_func, NULL);
    void *p;
    pthread_join(tid, (void**)&p);
    printf("return val: %d\n", *(int *)p);
    return 0;
}
```

线程终止

```
void pthread_cleanup_push(void (*rtn)(void*), void *arg);
void pthread_cleanup_pop(int execute);
```

安排线程退出时执行的函数，称为线程清理处理程序。一个线程可以建立多个清理处理程序，执行顺序与注册时的相反

1. push必须和pop配对使用，这已经是代码的结构层面的注意事项，因为两者都是由宏实现的，其中push包含花括号“{”,pop包含花括号“}”。
2. push和pop形成了一个作用域，称为清理函数作用域， 清理函数执行的可能情况如下：



线程终止

```
int pthread_detach(pthread_t tid)
// 设置线程状态为分离状态
// 成功，返回0， 失败，返回错误编号
```

如果一个线程结束了，其状态会保存到对该线程调用pthread_join， 如果线程被分离，线程的底层存储资源可以在线程终止时被立即回收。

线程同步

什么叫线程同步？

当一个可以修改的变量，可以同时被多个线程修改的时候，就需要对线程进行同步，确保他们访问变量的存储内容时不会访问到无效的值。

线程同步_互斥量

1. 对互斥量进行加锁之后，其他任何试图对该互斥量进行加锁的线程都会被阻塞，直到当前占有锁的线程释放锁，如果释放互斥量时有一个以上的线程阻塞在该互斥量上，那么所有该锁上的阻塞线程都会变成可运行状态，第一个加锁成功的的线程可以继续执行，其他的线程加锁失败，继续阻塞等待下一次机会。_在任何一个时刻，只能有一个线程可以向前执行

2.互斥量的初始化和销毁

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZED;
        //静态分配互斥量
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
        //动态分配互斥量
        //如果设置attr为NULL， 就是采用互斥量的默认属性
int pthread_mutex_destory(pthread_mutex_t *mutex);
        //销毁动态分配互斥量
//成功，返回0，失败，返回错误编号
```

3.互斥量的加锁和解锁

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
//不阻塞加锁，如果没法加锁，返回EBUSY,说明该互斥量已经被加锁了
int pthread_mutex_unlock(pthread_mutex_t *mutex);
//Linux下没有持有该锁，也可以释放锁，但是这样做还有什么意义？在某些解决死锁的方面还是有意义的
//成功，返回0，否则，返回错误编号
```

线程同步_互斥量

1.死锁现象？

2.死锁的成立条件？

3.如何预防死锁？

参考<<现代操作系统>>

线程同步_互斥量

```
int pthread_mutex_timedlock(pthread_mutex_t *mutex, const struct timespec *tsptr);

//指定愿意等待的绝对时间，即在时间 t 之前保持阻塞
//成功，返回 0 ，否则，返回错误编号
```

线程同步_读写锁

读模式加锁

读模式加锁，不会阻塞后续的所有读请求，但是会阻塞后续的写请求。如果一个读模式加锁，后续有一个写模式被阻塞，此时会阻塞过多的读模式来共享，从而防止写锁等待太久。

写模式加锁

写锁解锁之前，所有试图对这个锁加锁的线程都会被阻塞

不加锁

线程同步_读写锁

1. 读写锁初始化和销毁

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
        //静态初始化
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t &attr);
        //attr=NULL，采用默认属性，  动态初始化
int pthread_rwlock_destory(pthread_rwlock_t *rwlock);
//成功，返回0，失败，返回错误编号
```

2. 读写锁加锁解锁

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
//成功，返回0，失败，返回错误编号
```

3. 读写锁条件加锁

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
//成功，返回0，失败，返回错误编号
//当可以加锁，就加锁，不可以加锁，返回EBUSY
```

线程同步_读写锁

1. 带有超时的读写锁

```
int pthread_rwlock_timedrdlock(pthread_rwlock_t *rwlock, const struct timespec *tsptr);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *rwlock, const struct timespec *tsptr);
```

//成功，返回 0 ，失败，返回错误编号
//如果超时还没有获取锁，那么返回ETIMEDOUT错误
//时间是绝对时间

线程同步_条件变量

1．初始化条件变量

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_init(pthread_cond_t *cond);
//成功，返回 0 ，失败，返回错误编码
```

2．条件变量阻塞等待

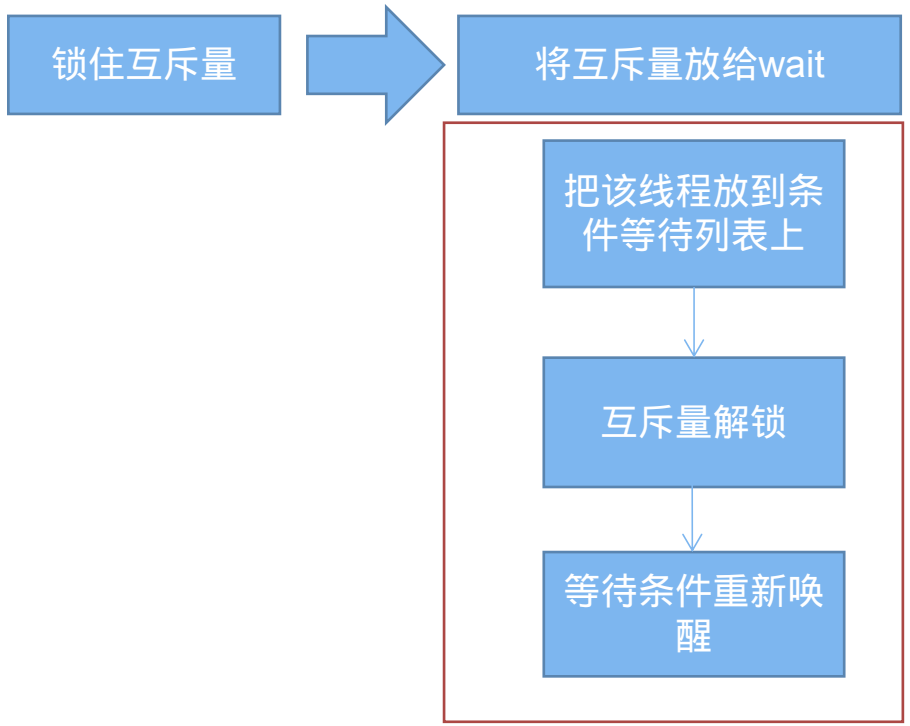
```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *tsptr);
//成功，返回 0 ，失败，返回错误编码
```

3．条件变量唤醒线程

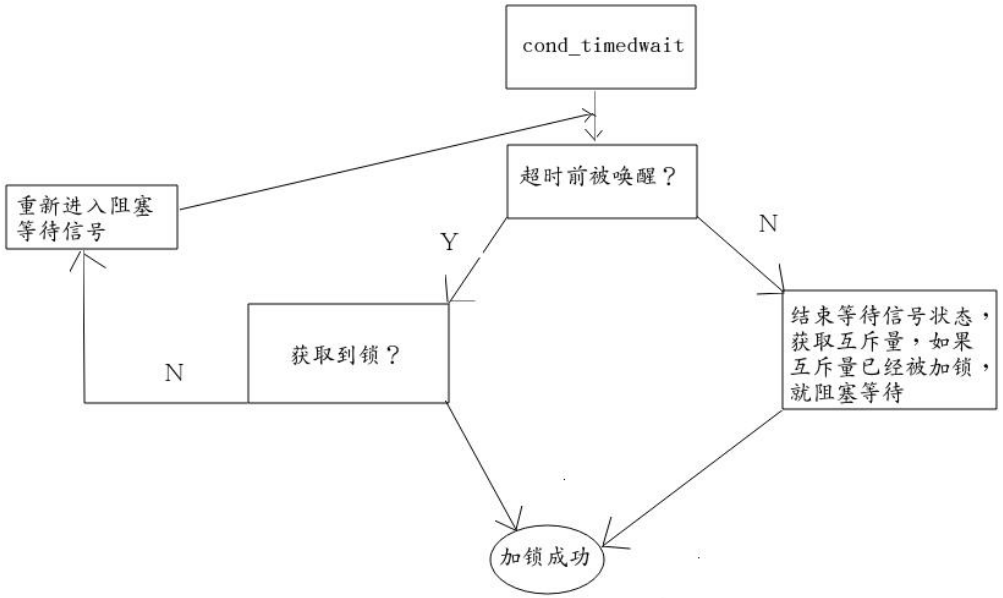
```
int pthread_cond_signal(pthread_cond_t *cond);  //向每一个信号队列中的线程发送信号，只要有一个获取锁成功，就结束
int pthread_cond_broadcast(pthread_cond_t *cond);  //要保证信号队列中每一个线程都要被唤醒并且加锁成功
//成功，返回0， 失败，返回错误编码
```

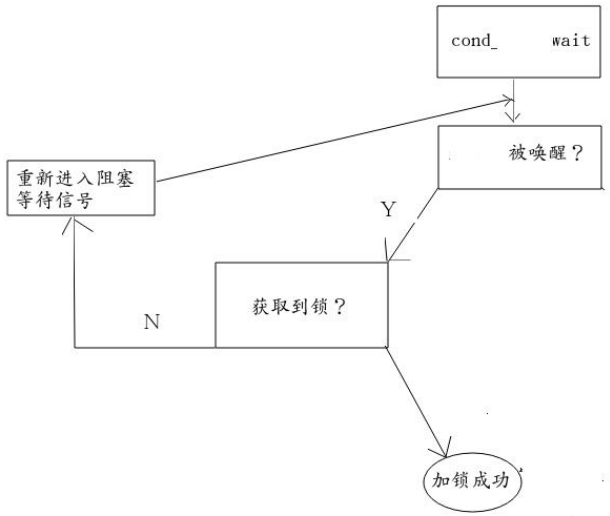
超时之前，等待在条件变量上，超时之后，等待在获取互斥量上

线程同步_条件变量



原子操作





线程同步_自旋锁

1．什么是自旋锁？

与互斥量类似，但是不是通过休眠使其阻塞，而是在获取锁之前一直处于忙等待阻塞状态。一般用于：锁被持有时间短，而且线程并不希望在重新调度上花费太多时间。

2．当自旋锁等待可用时，CPU不能做其他事情，依旧要给等待自旋锁的进程分配时间片。

3． 自旋锁初始化和销毁

```
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destory(pthread_spinlock_t *lock);
//成功，返回0,失败，返回错误编号
```

PTHREAD_PROCESS_PRIVATE

The spin lock is to be operated on only by threads in the same process as the thread that calls ***pthread_spin_init****(). (Attempting to share the spin lock between processes results in undefined behavior.)*

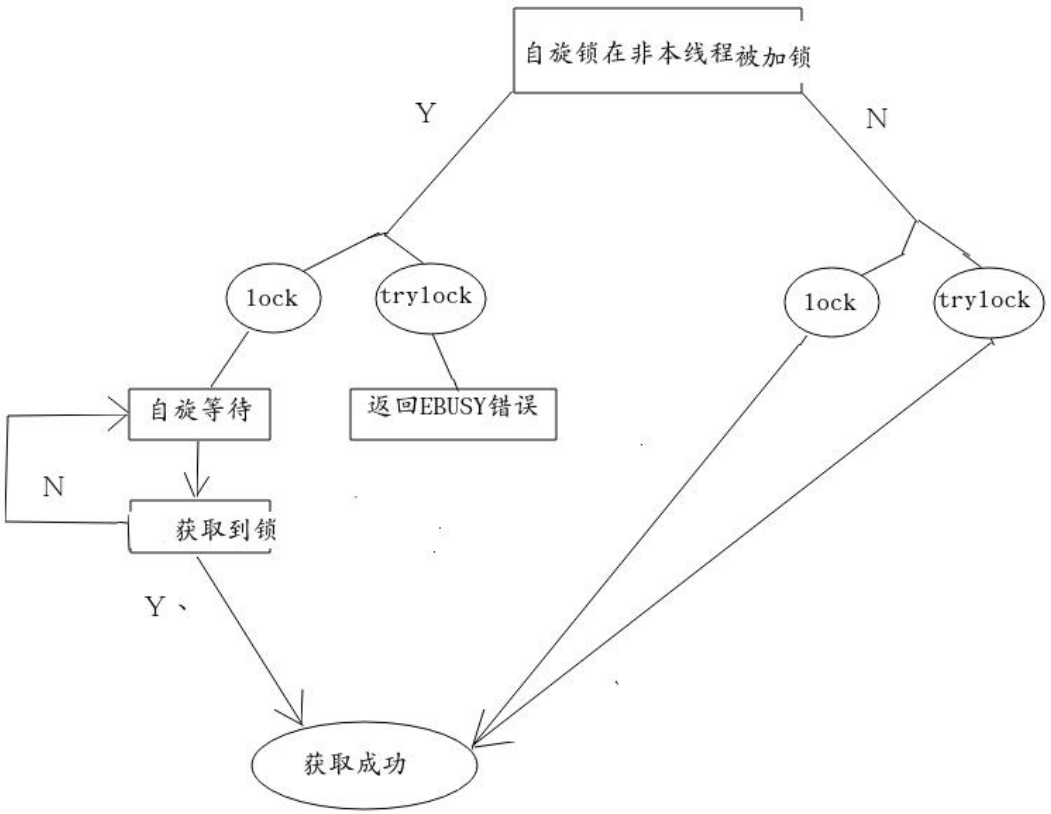
PTHREAD_PROCESS_SHARED

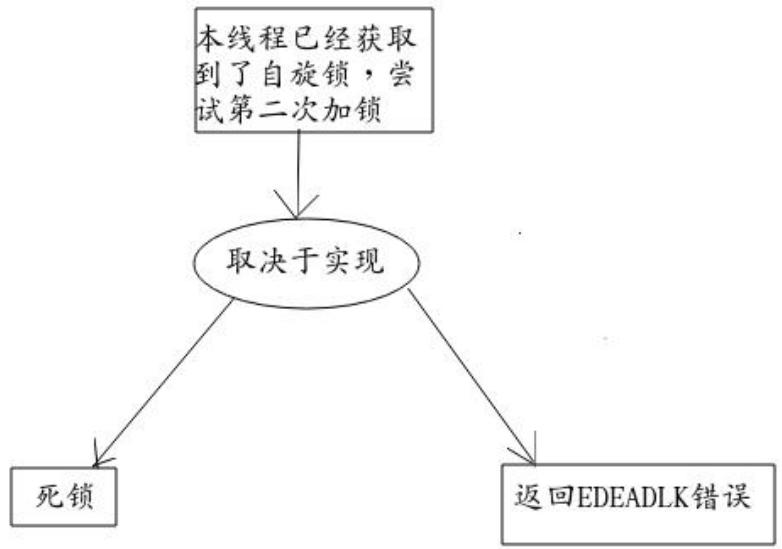
The spin lock may be operated on by any thread in any process that has access to the memory containing the lock (i.e., the lock may be in a shared memory object that is shared among multiple processes).

线程同步_自旋锁

1. 自旋锁的加锁和解锁

```
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
//成功，返回 0 ，失败，返回错误编号
```





线程同步_屏障

1. 屏障是用户协调多个线程并行工作的同步机制，屏障允许每个线程等待，直到所有的合作线程都到达某一点，然后从该点继续执行。

2. 屏障的初始化和销毁

```
int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *attr, unsigned int count);
int pthread_barrier_destory(pthread_barrier_t *barrier);
//成功，返回 0 ，否则，返回错误编号
//attr = NULL, 表示使用默认属性
//count表明该屏障需要到达的线程数目
```

3. 屏障等待

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
//成功，返回 0 或者PTHREAD_BARRIER_SERIAL_THREAD; 否则，返回错误编号
//调用该函数的所有线程在屏障计数未满足条件时，会进入休眠状态。如果该线程是最后一个调用pthread_barrier_wait的线程，就满足屏障计数，所有的线程都会被唤醒。
//对于任意一个线程，返回PTHREAD_BARRIER_SERIAL_THREAD， 剩下的线程返回值为 0 .
//屏障可以被重用，不需要重新进行初始化，但是如果想刚给计数count,需要先销毁再重新初始化，否则屏障计数不变。
```