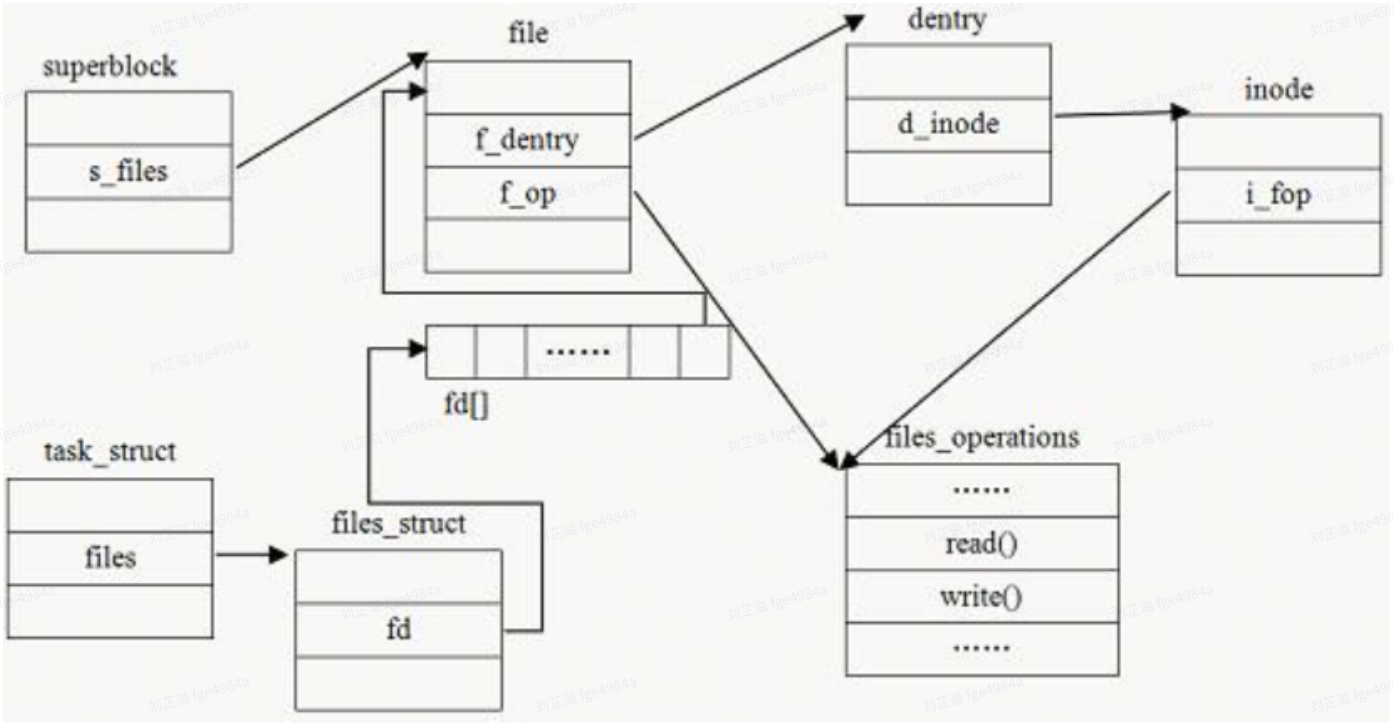
11-线程

11.1 线程概念

- 典型的UNIX进程可以看成一个控制线程：一个进程在某一个时刻只能做一件事情；多个线程，**每个线程可以处理各自独立的任务**

  线程优点

  - 为每种事件分配单独的处理线程，**简化处理异步事件的代码**；每个线程进行事件处理时采用同步编程模式
  - **多个线程可以自动访问相同的存储空间和文件描述符**；多个进程必须使用OS提供的复杂机制来实现



  - 任务可以分解成独立的子任务，用多个线程执行**提高程序吞吐量**
  - 交互程序使用多线程来**改善响应时间**

- 线程包含执行环境必需信息
  - 线程ID
  - 寄存器
  - 栈
  - 调度优先级和策略
  - 信号屏蔽字

- errno变量
  - **__thread修饰的变量在每一个线程中都有一份独立实例，各线程值是互不干扰的。**

```
extern int *__errno_location (void) __THROW __attribute__ ((__const__));
extern __thread int errno attribute_tls_model_ie;

// glibc/csu/errno-loc.c
#include <errno.h>
#include <tls.h>

int *
__errno_location (void)
{
  return &errno;
}
libc_hidden_def (__errno_location)
```

- 线程私有数据

11.2 线程标识

- 线程ID只有在它所属的进程上下文才有意义
- pthread_t类型；用结构来实现

```
// linux
/* Thread identifiers.  The structure of the attribute type is not
   exposed on purpose.  */
typedef unsigned long int pthread_t;

// windows
typedef struct {
    void * p;                  /* Pointer to actual object */
    unsigned int x;            /* Extra information - reuse count etc */
 } ptw32_handle_t;
 typedef ptw32_handle_t pthread_t;
```

- **比较线程ID**

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
// 相等返回非零数值，否则返回0
```

- **获取线程自身ID**

```
#include <pthread.h>
pthread_t pthread_self();
```

- 使用线程池时，可以利用上述两个函数；让工作线程只能移出自己线程ID的作业

11.3 线程创建

- 失败返回错误码，不设置errno
- 获取自身TID，使用pthread_self，而不是使用共享内存；如果新线程在主线程调用pthread_create返回前运行；新线程看到的ID是未初始化的ID

./threadid

```
#include <pthread.h>
int pthread_create(pthread_t *restrict tid, const pthread_attr_t *restrict
attr, void *(*start_run)(void*), void *restrict arg);
// 成功返回0，失败返回错误编号
```

- restrict关键字：在该指针的生命周期内，其指向的对象不会被别的指针所引用（编译器可以优化，将内存中的数据放到寄存器，后续对该数据计算不访存）

```
0000000000001180 <add2>:
int add2(int *a, int *b) {
    1180:   f3 0f 1e fa             endbr64
  *a = 10;
    1184:   c7 07 0a 00 00 00       movl   $0xa,(%rdi)
  *b = 12;
    118a:   c7 06 0c 00 00 00       movl   $0xc,(%rsi)
  return *a + *b;
    1190:   8b 07                   mov    (%rdi),%eax
    1192:   83 c0 0c                add    $0xc,%eax
}

0000000000001180 <add2>:
int add2(int *__restrict a, int *__restrict b) {
    1180:   f3 0f 1e fa             endbr64
  *a = 10;
    1184:   c7 07 0a 00 00 00       movl   $0xa,(%rdi)
```

```
18  }
19      118a:   b8 16 00 00 00          mov    $0x16,%eax
20    *b = 12;
21      118f:   c7 06 0c 00 00 00       movl   $0xc,(%rsi)
22  }
```

11.4 线程终止

- 从启动例程返回，返回值是线程退出码
- 被同一进程的线程取消
- 调用pthread_exit退出

```
1  #include <pthread.h>
2  void pthread_exit(void *rval_ptr);
3  // 从启动例程返回，返回退出码；取消，内存单元设置为PTHREAD_CANCELED
```

- 线程可以通过pthread_join访问void *rval_ptr

```
1  #include <pthread.h>
2  int pthread_join(pthread_t thread, void **rval_ptr);
3  // 成功返回0，失败返回错误编号
```

./exitstatus

- pthread_create和pthread_exit结构必须保证**内存在调用者完成调用后仍然有效（不要用栈）**

./badexit, ./correctexit

- pthread_cancel可以取消同一进程的其他线程
- 不等待线程终止，只是提出请求
- 将返回值(void* rval_ptr)单元设置为**PTHREAD_CANCELED**

```
1  #include <pthread.h>
2  int pthread_cancel(pthread_t tid);
3  // 成功返回0，否则返回错误编号
4  #define PTHREAD_CANCELED ((void*)-1)
```

./threadcancel

线程清理程序

- 执行顺序与注册顺序相反
- rtn被调用
  - 调用pthread_exit
  - 响应取消请求时
  - 用非零execute参数调用pthread_cleanup_pop函数
- execute函数设置为0，清理函数不被调用；**正常返回线程，不调用清理函数**

```
#include <pthread.h>
void pthread_cleanup_push(void (*rtn)(void*), void* arg);
void pthread_cleanup_pop(int execute);
```

./cleanup

线程分离

- **分离后，线程的底层存储资源可以在线程终止时立即被收回**

```
#include <pthread.h>
int pthread_detach(pthread_t pid);
// 成功返回0，否则返回错误编号
```

进程与线程原语比较

进程原语	线程原语	描述
fork	pthread_create	创建新的控制流
exit	pthread_exit	从现有控制流退出
waitpid	pthread_join	从控制流中得到退出状态
atexit	pthread_cleanup_push	注册退出控制流时执行的函数
getpid	pthread_self	获取控制流ID
abort	pthread_cancel	请求控制流的非正常退出

11.5 线程同步

- 多个线程共享内存时，需要保证使用的变量不被其他线程修改
- **修改是原子操作，不存在竞争；但存储访问需要多个总线周期，不能保证原子操作**
  - 增量操作包含：1. 从内存单元读入寄存器，2. 寄存器自增， 3. 将新值写回内存单元

互斥量

- 本质上是一把锁（mutex）；
  - **通过休眠阻塞进程/线程，确保同一时间只有一个线程访问数据**
  - 如果阻塞在该互斥锁上的线程有多个，当锁可用时，所有线程都会变成可运行状态，**第一个变为运行的线程，就可以对互斥量加锁，其他线程则再次等待锁而进入休眠。**
- 需要初始化，赋值为PTHREAD_MUTEX_INITIALIZER或调用pthread_mutex_init
- 动态分配互斥量，释放内存前需要调用pthread_mutex_destroy

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
pthread_mutexattr_t *restrict attr);
// attr设置为NULL，默认初始化互斥量
int pthread_mutex_destroy(pthread_mutex_t *mutex);
// 成功返回0，否则返回错误编号
```

- pthread_mutex_trylock不会造成线程阻塞；如果互斥量已经锁住，**该函数直接返回EBUSY**

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
// 成功返回0，否则返回错误编号
```

- 绑定超时时间的互斥量
  - 指定愿意等待的超时时间；timespec用秒和纳秒描述时间
  - 超时返回ETIMEOUT

./timedlock

```
#include <pthread.h>
#include <time.h>
```

```
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const struct timespec *restrict tsptr);
// 成功返回0，否则返回错误编号
```

避免死锁

- 需要仔细控制互斥量加锁的顺序来避免死锁
- 如果涉及太多锁和数据结构；使用pthread_mutex_trylock；
  - 如果成功，继续执行
  - 如果失败，先释放占有的锁，清理后，重新尝试加锁

mutex1

```
struct foo {
  int             f_count;
  pthread_mutex_t f_lock;    // 一把互斥锁
  int             f_id;
  /* ... more stuff here ... */
};

struct foo *
foo_alloc(int id) /* allocate the object */
{
  struct foo *fp;
  // 分配空间
  if ((fp = malloc(sizeof(struct foo))) != NULL) {
    fp->f_count = 1;
    fp->f_id = id;
    // 初始化mutex
    if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
      free(fp);
      return(NULL);
    }
    /* ... continue initialization ... */
  }
  return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
  pthread_mutex_lock(&fp->f_lock);
  fp->f_count++;
  pthread_mutex_unlock(&fp->f_lock);
```

```
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
  pthread_mutex_lock(&fp->f_lock);
  if (--fp->f_count == 0) { /* last reference */
    pthread_mutex_unlock(&fp->f_lock);
    pthread_mutex_destroy(&fp->f_lock);
    free(fp);
  } else {
    pthread_mutex_unlock(&fp->f_lock);
  }
}
```

mutex2

- foo结构放在全局链表上，需要加锁
- foo_find，先锁住链表，在对元素加锁
- 保证上锁顺序：**先对链表上锁，再对元素上锁**

```
#define NHASH 29
#define HASH(id) (((unsigned long)id)%NHASH)
// 一个链表
struct foo *fh[NHASH];

pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;


struct foo *
foo_alloc(int id) /* allocate the object */
{
  struct foo  *fp;
  int     idx;

  if ((fp = malloc(sizeof(struct foo))) != NULL) {
    fp->f_count = 1;
    fp->f_id = id;
    if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
      free(fp);
      return(NULL);
    }
    idx = HASH(id);
    pthread_mutex_lock(&hashlock);
```

```
24        fp->f_next = fh[idx];
25        fh[idx] = fp;
26        pthread_mutex_lock(&fp->f_lock);
27        pthread_mutex_unlock(&hashlock);
28        /* ... continue initialization ... */
29        pthread_mutex_unlock(&fp->f_lock);
30      }
31      return(fp);
32    }
33
34    void
35    foo_hold(struct foo *fp) /* add a reference to the object */
36    {
37      pthread_mutex_lock(&fp->f_lock);
38      fp->f_count++;
39      pthread_mutex_unlock(&fp->f_lock);
40    }
41
42    struct foo *
43    foo_find(int id) /* find an existing object */
44    {
45      struct foo  *fp;
46
47      pthread_mutex_lock(&hashlock);
48      for (fp = fh[HASH(id)]; fp != NULL; fp = fp->f_next) {
49        if (fp->f_id == id) {
50          foo_hold(fp);
51          break;
52        }
53      }
54      pthread_mutex_unlock(&hashlock);
55      return(fp);
56    }
57
58    void
59    foo_rele(struct foo *fp) /* release a reference to the object */
60    {
61      struct foo  *tfp;
62      int     idx;
63      // 先对fp上锁，判断f_count的值；然后释放锁，重新按顺序获取锁
64      pthread_mutex_lock(&fp->f_lock);
65      if (fp->f_count == 1) { /* last reference */
66        pthread_mutex_unlock(&fp->f_lock);
67        // 重新按顺序获取锁
68        // 先获取散列表的锁，再获取fp的锁
69        pthread_mutex_lock(&hashlock);
70        pthread_mutex_lock(&fp->f_lock);
```

```
71      /* need to recheck the condition */
72      if (fp->f_count != 1) {
73        fp->f_count--;
74        pthread_mutex_unlock(&fp->f_lock);
75        pthread_mutex_unlock(&hashlock);
76        return;
77      }
78      /* remove from list */
79      idx = HASH(fp->f_id);
80      tfp = fh[idx];
81      if (tfp == fp) {
82        fh[idx] = fp->f_next;
83      } else {
84        while (tfp->f_next != fp)
85          tfp = tfp->f_next;
86        tfp->f_next = fp->f_next;
87      }
88      pthread_mutex_unlock(&hashlock);
89      pthread_mutex_unlock(&fp->f_lock);
90      pthread_mutex_destroy(&fp->f_lock);
91      free(fp);
92    } else {
93      fp->f_count--;
94      pthread_mutex_unlock(&fp->f_lock);
95    }
96  }
```

mutex3

- 用散列表锁来保护结构引用计数；不必再纠结于释放或申请锁的顺序

```
1   pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;
2
3   struct foo {
4     int             f_count; /* protected by hashlock */
5     pthread_mutex_t f_lock;
6     int             f_id;
7     struct foo     *f_next; /* protected by hashlock */
8     /* ... more stuff here ... */
9   };
10
11  struct foo *
12  foo_alloc(int id) /* allocate the object */
13  {
14    struct foo  *fp;
15    int     idx;
```

```
16
17    if ((fp = malloc(sizeof(struct foo))) != NULL) {
18      fp->f_count = 1;
19      fp->f_id = id;
20      if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
21        free(fp);
22        return(NULL);
23      }
24      idx = HASH(id);
25      pthread_mutex_lock(&hashlock);
26      fp->f_next = fh[idx];
27      fh[idx] = fp;
28      pthread_mutex_lock(&fp->f_lock);
29      pthread_mutex_unlock(&hashlock);
30      /* ... continue initialization ... */
31      pthread_mutex_unlock(&fp->f_lock);
32    }
33    return(fp);
34  }
35
36  void
37  foo_hold(struct foo *fp) /* add a reference to the object */
38  {
39    pthread_mutex_lock(&hashlock);
40    fp->f_count++;
41    pthread_mutex_unlock(&hashlock);
42  }
43
44  struct foo *
45  foo_find(int id) /* find an existing object */
46  {
47    struct foo  *fp;
48
49    pthread_mutex_lock(&hashlock);
50    for (fp = fh[HASH(id)]; fp != NULL; fp = fp->f_next) {
51      if (fp->f_id == id) {
52        fp->f_count++;
53        break;
54      }
55    }
56    pthread_mutex_unlock(&hashlock);
57    return(fp);
58  }
59
60  void
61  foo_rele(struct foo *fp) /* release a reference to the object */
62  {
```

```
63    struct foo  *tfp;
64    int     idx;
65
66    pthread_mutex_lock(&hashlock);
67    if (--fp->f_count == 0) { /* last reference, remove from list */
68      idx = HASH(fp->f_id);
69      tfp = fh[idx];
70      if (tfp == fp) {
71        fh[idx] = fp->f_next;
72      } else {
73        while (tfp->f_next != fp)
74          tfp = tfp->f_next;
75        tfp->f_next = fp->f_next;
76      }
77      pthread_mutex_unlock(&hashlock);
78      pthread_mutex_destroy(&fp->f_lock);
79      free(fp);
80    } else {
81      pthread_mutex_unlock(&hashlock);
82    }
83  }
```

读写锁（共享互斥锁）

- 三种状态：读模式加锁，写模式加锁，不加锁
- 单线程写，多线程读
- **读写锁使用前必须初始化，释放内存前必须销毁**

```
1  #include <pthread.h>
2  int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const
   pthred_rwlockattr_t *restrict attr);
3
4  int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
5  // 成功返回0，否则返回错误编号
```

- 加锁与解锁
- 有的实现可能会对共享模式下可获取的读写锁次数进行限制；**需要检测rdlock的返回值**

```
1  #include <pthread.h>
2  int pthread_rwlock_rdlock(pthread_rwlock_t  *rwlock);
3  int pthread_rwlock_wrlock(pthread_rwlock_t  *rwlock);
4  int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

```
5   // 成功返回0，否则返回错误编号
```

- Single UNIX Specification定义了读写锁的条件版本
- 可以获取锁返回0，否则返回EBUSY

```
1   #include <pthread.h>
2   int pthread_rwlock_tryrdlock(pthread_rwlock_t  *rwlock);
3   int pthread_rwlock_trywrlock(pthread_rwlock_t  *rwlock);
4   // 成功返回0，否则返回错误编号
```

- 超时的读写锁
- **超时是绝对时间，不是相对时间**
  - 获取当前时间后，加上相对时间作为超时时间

```
1   #include <pthread.h>
2   #include <time.h>
3   int pthread_rwlock_timedrdlock (pthread_rwlock_t *restrict rwlock, const
    struct timespec *restrict tsptr);
4   int pthread_rwlock_timedwrlock (pthread_rwlock_t *restrict rwlock, const
    struct timespec *restrict tsptr);
5   // 成功返回0，否则返回错误编号
```

./rwlock

条件变量

- 条件本身由互斥量保护；**线程改变条件之前必须锁定互斥量**
- 使用条件变量之前，必须对条件变量pthread_cond_t进行初始化；
  - 分配给静态变量PTHREAD_COND_INITIALIZER
  - 使用pthread_cond_init函数
- 释放空间前，使用pthread_cond_destroy来销毁

```
1   #include <pthread.h>
2   int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t
    *restrict attr);
3   int pthread_destroy(pthread_cond_t *cond);
4   // 成功返回0，否则返回错误编号
```

- 等待条件变量变为真
- 超时值是绝对值，需要在当前时间的基础上进行计算
  - 如果超时到期条件还未出现，pthread_cond_timedwait将重新获取互斥量，返回错误ETIMEOUT
  - 从pthread_cond_wait成功返回，**需要重新计算条件，因为另一个线程可能运行改变了条件**

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict
 mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t
*restrict mutex, const struct timespec *restrict tsptr);
// 成功返回0，否则返回错误编号
```

- 通知线程条件已经满足
  - 改变条件后再给线程发送信号

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);          // 至少唤醒一个线程
int pthread_cond_broadcast(pthread_cond_t *cond);       // 唤醒等待该条件的所有线程
// 成功返回0，否则返回错误编号
```

- 使用条件变量

```
struct msg {
  struct msg *m_next;
  char *m_data;
  /* ... more stuff here ... */
};

struct msg *workq;

pthread_cond_t qready = PTHREAD_COND_INITIALIZER;

pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

// 消费者
void *process_msg(void *arg) {
  struct msg *mp;
  for (;;) {
```

```
17        pthread_mutex_lock(&qlock);
18        while (workq == NULL) {
19          printf("waiting for produce msg...\n");
20          pthread_cond_wait(&qready, &qlock);
21        }
22        mp = workq;
23        workq = mp->m_next;
24        pthread_mutex_unlock(&qlock);
25        /* now process the message mp */
26      }
27    }
28
29    // 生产者
30    void *enqueue_msg(void *arg) {
31      struct msg *mp = (struct msg *)arg;
32      pthread_mutex_lock(&qlock);
33      // 头插法
34      mp->m_next = workq;
35      workq = mp;
36      printf("msg is comming\n");
37      pthread_mutex_unlock(&qlock);
38      pthread_cond_signal(&qready);
39    }
```

./condmsg

自旋锁

- 互斥量通过休眠使进程阻塞；不过自旋锁不是通过休眠阻塞进程，**而是在取得锁之前一直处于忙等待的阻塞状态。这个忙等的阻塞状态，也叫做自旋。**
- 适用场景
  - 锁被持有的时间短；线程不希望重新调度花费太多成本
  - 非抢占式内核，需要阻塞中断（**中断处理程序不能休眠，唯一可以使用的同步原语就是自旋锁**）
- 很多互斥量的实现在**试图获取互斥量时会自旋一段时间，当自旋计数到达一定阈值才休眠**；使互斥量性能与自旋性能接近
- pshared是进程共享属性
  - PTHREAD_PROCESS_SHARED：自旋锁可以访问锁底层内存的线程获取，不同进程线程也可以
  - PTHREAD_PROCESS_PRIVATE：只能被初始化该锁进程的线程访问

```c
#include <pthread.h>
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);
// 成功返回0，否则返回错误编号
```

- pthread_spin_lock在获取锁之前一直自旋；pthread_spin_trylock（**不能自旋**）如果不能获取锁，立即返回EBUSY错误
- 在线程已经获取锁再获取spin_lock时，**调用pthread_spin_lock会返回EDEADLK或其它错误，或者调用可能永久自旋**；具体行为取决于实现；
- 试图对没有加锁的自旋锁解锁，行为未定义（本机上没有错误）
- **不要在持有自旋锁情况下调用可能休眠的函数**；会浪费CPU资源，其他线程等待自旋锁时间增加

```c
#include <pthread.h>
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，否则返回错误编号
```

./spinlock

屏障

- 用户协调多个线程并行工作的同步机制；允许每个线程等待，直到所有的合作线程到达某一点，从该点继续执行
- pthread_join是一种屏障，允许一个线程等待，直到另一个线程退出
- **count指定允许线程允许运行前， 必须到达屏障的线程数目**

```c
#include <pthread.h>
int pthread_barrier_init(pthread_barrire_t *restrict barrier, const pthread_barrierattr_t *restrict attr, unsigned int count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
// 成功返回0，否则返回错误编号
```

- pthread_barrier_wait表明线程已经完成工作，等待其他线程；
- 调用该函数的线程**在屏障计数条件未满足时，进入休眠状态；满足后，唤醒所有线程**
- 到达屏障计数值，线程处于阻塞状态后，屏障可以被重用

```c
#include <pthread.h>
int pthread_barrier_wait(pthread_barrier_t *barrier);
// 成功返回0，否则返回错误编号
```

- 可以用互斥量和条件变量实现屏障

```c
typedef struct s_my_pthread_barrier_t {
  pthread_mutex_t lock; // 互斥锁
  pthread_cond_t cond;  // 条件变量
  unsigned int count;   // 等待的线程数
} my_pthread_barrier_t;

int my_pthread_barrier_wait(my_pthread_barrier_t *barrier) {
  pthread_mutex_lock(&barrier->lock);
  // 如果是最后一个线程，就唤醒所有等待的线程，否则就阻塞
  if (--barrier->count == 0) {
    pthread_cond_broadcast(&barrier->cond);
  }

  while (barrier->count > 0) {
    pthread_cond_wait(&barrier->cond, &barrier->lock);
  }

  pthread_mutex_unlock(&barrier->lock);
}
```

./barrier