第十一章 线程

冯 晶 E-mail: gfeng@whu.edu.cn

1. 线程的概念

- 在单进程环境中执行多个任务
  - 简化异步事件处理的代码，每个线程在进行事件处理时可以采用同步编程模式
  - 多个进程使用操作系统提供的复杂机制才能实现内存和文件描述符的共享；多线程自动可以访问相同的存储地址空间和文件描述符
  - 分解问题，提高整个程序的吞吐量
  - 交互程序可以通过多线程来改善响应时间，可以把程序的输入输出和其它部分分开

1. 线程的概念

- 在单进程环境中执行多个任务
  - 每个线程包含有表示执行环境所必需的信息，包括线程ID、一组寄存器值、栈、调度优先级和策略、信号屏蔽字、errno变量以及线程私有数据。
  - 一个进程的所有信息对该进程的所有线程都共享，包括可执行程序的代码、程序的全局内存和堆内存、栈以及文件描述符

2. 线程标识

- 比较两线程ID

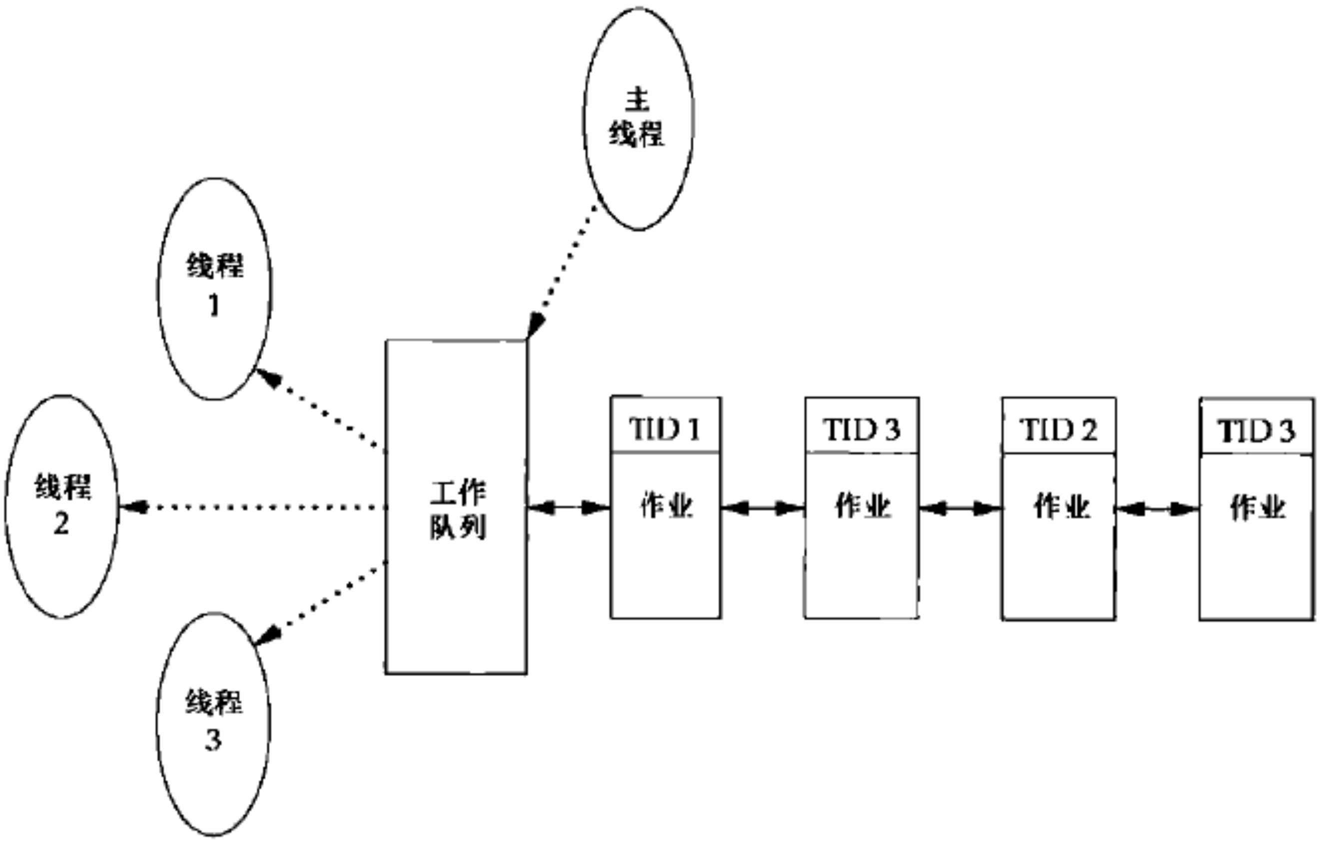
```
#include <pthread.h>
int  pthread_equal(pthread_t tid1, pthread_t tid2);
```

- 获取线程自身ID

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

2. 线程标识

- 线程池



3. 创建线程

- 创建线程

```
#include <pthread.h>
int  pthread_create (pthread_t  *restrict  tidp,
                     const  pthread_attr_t  *restrict  attr,
                     void  *(*strat_rtn) (void *),
                     void  *restrict  arg);
```

versity

```
pthread_t ntid;

void
printids(const char *s)
{
    pid_t      pid;
    pthread_t  tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
      (unsigned int)tid, (unsigned int)tid);
}

void *
thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}

int
main(void)
{
    int     err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_quit("can't create thread: %s\n", strerror(err));
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

4. 线程终止

- 线程终止
  - 线程可以简单的从启动例程中返回，返回值是线程的退出码。
  - 线程可以被同一进程中的其它线程取消。
  - 线程调用pthread_exit。

```
#include <pthread.h>
void pthread_exit(void *rval_ptr);
```

4. 线程终止

- 分离线程，释放资源

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **rval_ptr);
```

iversity

```
void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_quit("can't create thread 1: %s\n", strerror(err));
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_quit("can't create thread 2: %s\n", strerror(err));
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_quit("can't join with thread 1: %s\n", strerror(err));
    printf("thread 1 exit code %d\n", (int)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_quit("can't join with thread 2: %s\n", strerror(err));
    printf("thread 2 exit code %d\n", (int)tret);
    exit(0);
}
```

4. 线程终止

- 请求取消同一进程的其它线程

```
#include <pthread.h>
int  pthread_cancel(pthread_t  tid);
```

  - 被请求线程可以忽略取消
  - 被请求线程可以控制如何被取消

4. 线程终止

- 线程清理程序

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

- 以下3中情况清理函数将被执行
  - 调用pthread_exit时
  - 响应取消请求时
  - 用非零execute参数调用pthread_cleanup_pop时

4. 线程终止

- 进程和线程比较

进程原语	线程原语	描述
fork	pthread_create	创建新的控制流
exit	pthread_exit	从现有的控制流中退出
waitpid	pthread_join	从控制流中得到退出状态
atexit	pthread_cancel_push	注册在退出控制流时调用的函数
getpid	pthread_self	获取控制流的 ID
abort	pthread_cancel	请求控制流的非正常退出

4. 线程终止

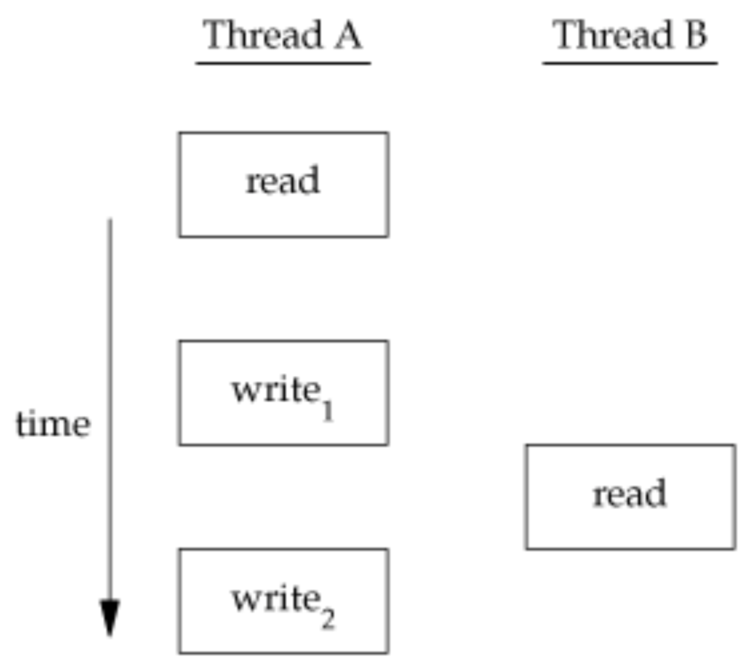
- 分离线程

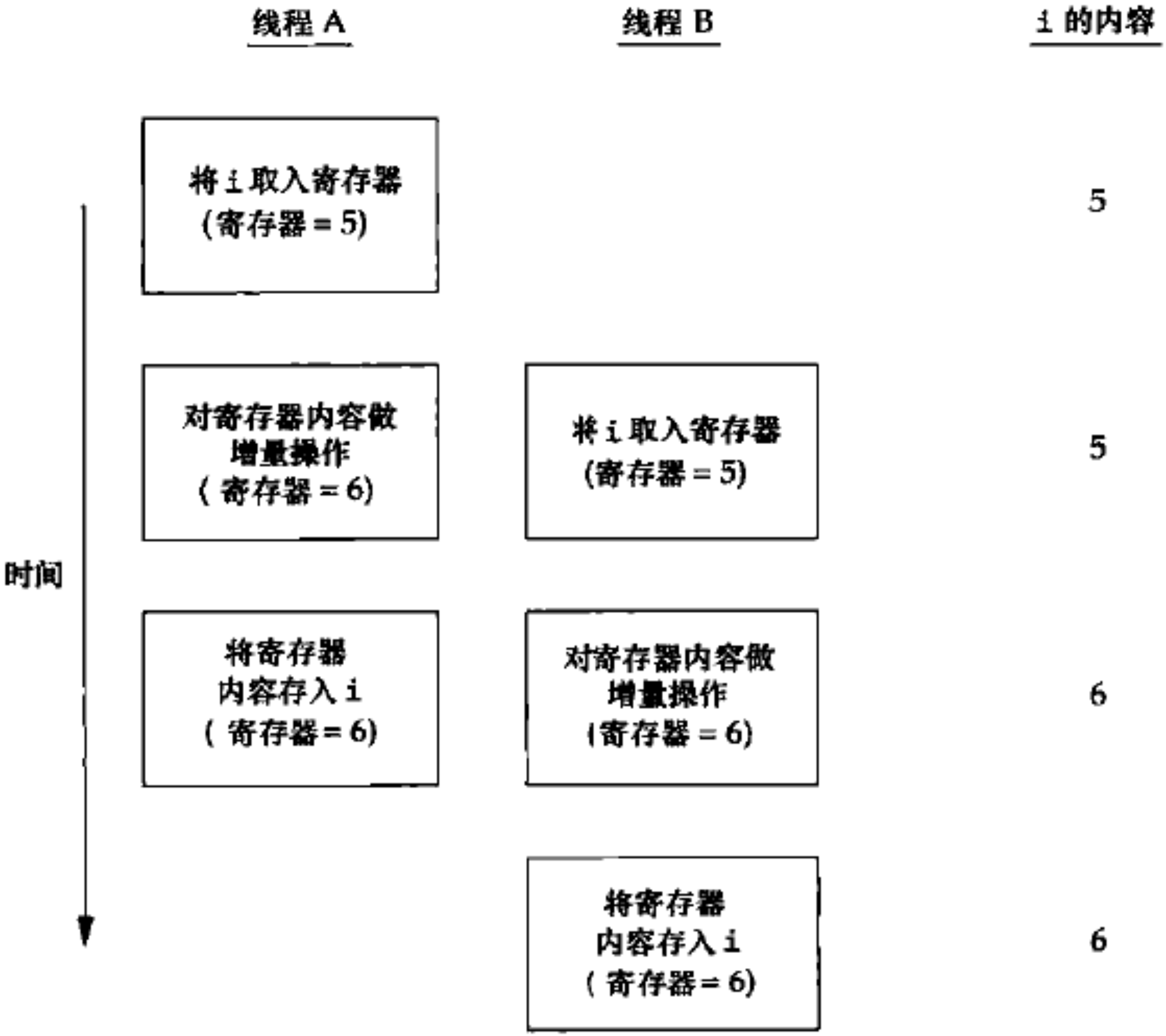
```
#include <pthread.h>
int  pthread_detach(pthread_t  tid);
```

  - pthread_join自动把线程置于分离状态。如果线程已经处于分离状态，pthread_join调用会失败。可以调用pthread_detach分离线程。

5. 线程同步

- 线程同步问题
  - 多个线程读取或修改同一存储空间引起的不一致问题（变量修改时间多于一个存储器访问周期）




线程 A
线程 B
i 的内容
将 i 取入寄存器
(寄存器 = 5)
5
对寄存器内容做
增量操作
(寄存器 = 6)
将 i 取入寄存器
(寄存器 = 5)
5
时间
将寄存器
内容存入 i
(寄存器 = 6)
对寄存器内容做
增量操作
(寄存器 = 6)
6
将寄存器
内容存入 i
(寄存器 = 6)
6

5. 线程同步

- 互斥量

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

5. 线程同步

- 互斥量

```
#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);
```

5. 线程同步


```
struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(void) /* allocate the object */
{
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... continue initialization ... */
    }
    return(fp);
}
```

5. 线程同步


```
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);
    }
}
```

5. 线程同步

- 避免死锁
  - 线程试图对同一互斥量加锁两次
  - 多个进程试图对多个互斥量进行加锁

```
#include <pthread.h>
int  pthread_mutex_trylock (pthread_mutex_t  *mutex);
```

```c
#define NHASH 29
#define HASH(fp) (((unsigned long)fp)%NHASH)
struct foo *fh[NHASH];

pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    struct foo     *f_next; /* protected by hashlock */
    int             f_id;
    /* ... more stuff here ... */
};

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

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(fp);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp->f_next;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... continue initialization ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
    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);
}

struct foo *
foo_find(int id) /* find an existing object */
{
    struct foo *fp;
    int         idx;

    idx = HASH(fp);

    pthread_mutex_lock(&hashlock);
    for (fp = fh[idx]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            foo_hold(fp);
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}
```

```c
void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    struct foo  *tfp;
    int         idx;

    pthread_mutex_lock(&fp->f_lock);
    if (fp->f_count == 1) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_lock(&hashlock);
        pthread_mutex_lock(&fp->f_lock);
        /* need to recheck the condition */
        if (fp->f_count != 1) {
            fp->f_count--;
            pthread_mutex_unlock(&fp->f_lock);
            pthread_mutex_unlock(&hashlock);
            return;
        }
        /* remove from list */
        idx = HASH(fp);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        fp->f_count--;
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

5. 线程同步

- 设置超时等待锁

```
#include <pthread.h>
int  pthread_mutex_timedlock (pthread_mutex_t  *restrict mutex,
                               const  struct  timespec  *restrict  tsptr);
```

```
int
main(void)
{
    int err;
    struct timespec tout;
    struct tm *tmp;
    char buf[64];
    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

    pthread_mutex_lock(&lock);
    printf("mutex is locked\n");
    clock_gettime(CLOCK_REALTIME, &tout);
    tmp = localtime(&tout.tv_sec);
    strftime(buf, sizeof(buf), "%r", tmp);
    printf("current time is %s\n", buf);
    tout.tv_sec += 10; /* 10 seconds from now */
    /* caution: this could lead to deadlock */
    err = pthread_mutex_timedlock(&lock, &tout);
    clock_gettime(CLOCK_REALTIME, &tout);
    tmp = localtime(&tout.tv_sec);
    strftime(buf, sizeof(buf), "%r", tmp);
    printf("the time is now %s\n", buf);
    if (err == 0)
        printf("mutex locked again!\n");
    else
        printf("can't lock mvtex again:%s\n",strerror(err));
    exit(0);
}
```

Jniversity

5. 线程同步

- 读写锁（共享互斥锁）
  - 读模式加锁状态
  - 写模式加锁状态
  - 不加锁状态

```
#include <pthread.h>
int pthread_rwlock_init (pthread_rwlock_t *restrict rwlock,
                         const pthread_rwlockattr_t *restrict attr);

int pthread_rwlock_destory (pthread_rwlock_t *rwlock);
```

5. 线程同步

- 读写锁（共享互斥锁）
  - 读模式加锁状态
  - 写模式加锁状态
  - 不加锁状态

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

5. 线程同步

- 读写锁（共享互斥锁）
  - 读模式加锁状态
  - 写模式加锁状态
  - 不加锁状态

```
#include <pthread.h>
int  pthread_rwlock_tryrdlock(pthread_rwlock_t  *rwlock);
int  pthread_rwlock_trywrlock(pthread_rwlock_t  *rwlock);
```

5. 线程同步

- 读写锁（共享互斥锁）
  - 读模式加锁状态
  - 写模式加锁状态
  - 不加锁状态

```
#include <pthread.h>
int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tsptr);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tsptr);
```

5. 线程同步

- 条件变量
  - 条件变量与互斥锁一起使用，允许线程以无竞争的方式等待特定的条件发生
  - 条件变量本身由互斥锁保护

```
#include <pthread.h>
int pthread_cond_init (pthread_cond_t *restrict cond,
                       const pthread_condattr_t *restrict attr);
int pthread_cond_destroy (pthread_cond_t *cond);
```

5. 线程同步

- 条件变量
  - 条件变量与互斥锁一起使用，允许线程以无竞争的方式等待特定的条件发生
  - 条件变量本身由互斥锁保护

```
#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);
```

5. 线程同步

- 条件变量
  - 通知线程条件已经满足

```
#include <pthread.h>
int  pthread_cond_signal (pthread_cond_t  *cond);
int  pthread_cond_broadcast (pthread_cond_t  *cond);
```

```c
struct msg {
    struct msg *m_next;
    /* ... more stuff here ... */
};
struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void
process_msg(void)
{
    struct msg *mp;

    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);
        /* now process the message mp */
    }
}

void
enqueue_msg(struct msg *mp)
{
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_mutex_unlock(&qlock);
    pthread_cond_signal(&qready);
}
```

University

5. 线程同步

- 自旋锁
  - 自旋锁与互斥锁类似
  - 不是通过休眠使线程阻塞，而是在获取锁之前一直处于忙等（自旋）阻塞状态
  - 适用于锁被持有的时间短，线程不希望在重新调度上花费太多成本

```
#include <pthread.h>
int  pthread_spin_init (pthread_spinlock_t  *lock,  int  pshared);
int  pthread_spin_destroy (pthread_spinlock_t  *lock);
```

5. 线程同步

- 自旋锁
  - 自旋锁与互斥锁类似
  - 不是通过休眠使线程阻塞，而是在获取锁之前一直处于忙等（自旋）阻塞状态
  - 适用于锁被持有的时间短，线程不希望在重新调度上花费太多成本

```
#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);
```

5. 线程同步

- 屏障
  - 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都达到某一点，然后从该点继续执行。
  - pthread_join函数就是一种屏障，等待另一个线程的退出

```
#include <pthread.h>
int  pthread_barrier_wait (pthread_barrier_t  *barrier);
```