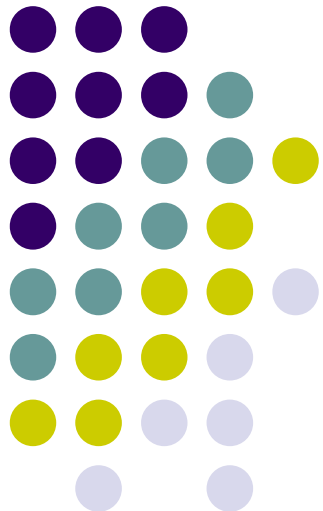
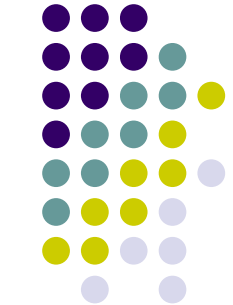
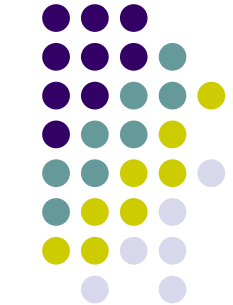
Ch4 线程编程



主要内容



- 线程概念
- 基本API函数
- 线程同步
- 线程属性



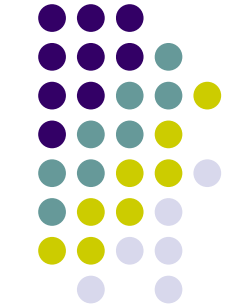
本章重点

- 线程概念
- 线程函数的介绍以及应用
- 线程互斥锁的应用
- 线程信号量的应用

Part 1 线程概念

- 进程的所有信息对该进程的所有线程都是共享的，包括可执行的程序文本、程序的全局内存、堆内存、文件描述符等
- 线程独有的：线程ID、寄存器值、栈、信号屏蔽字、errno值、线程私有数据
- 典型的UNIX进程，可以看成是只有一个线程的进程
- **线程：一个独立运行的函数**

1.1 线程概念



- 同进程一样，每个线程也有一个线程ID
- 进程ID在整个系统中是唯一的，但线程ID不同，线程ID只在它所属的进程环境中有效
- 线程ID的类型是pthread_t，在Linux中的定义：
  - 在/usr/include/bits/pthreadtypes.h中
  - typedef unsigned long int pthread_t;
  - ps ax -L
  - gcc -o -pthread

Part 2 基本API函数

2.1 线程创建——pthread_create函数

- 头文件：#include<pthread.h>
- 功能：创建一个线程
- 函数原型

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

Part 2 基本API函数

2.1 线程创建——pthread_create函数

- 参数
  - thread:线程标识符
  - attr:线程属性设置，通常可设为NULL，采用默认线程属性
  - start_routine:线程函数的起始地址
  - arg:传递给start_routine的参数
- 返回值
- 成功：0；失败：error number

Part 2 基本API函数

2.2 线程终止——pthread_exit函数

- 单个线程的三种退出方式
  - 线程从启动例程中返回，返回值是线程的退出码
  - 线程被同一进程中的其他线程取消
  - 线程调用pthread_exit函数

Part 2 基本API函数

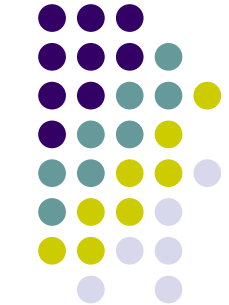
2.2 线程终止——pthread_exit函数

- 该函数让线程退出

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

- 参数
  - rval_ptr：与线程的启动函数类似，该指针将传递给pthread_join函数

Part 2 基本API函数

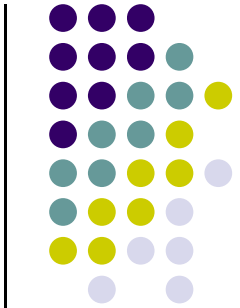


2.3 线程取消——pthread_cancel函数

- 功能：线程调用该函数可以取消同一进程中的其他线程，即让线程终止
- 函数原型

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

- 参数与返回值
  - tid：需要取消的线程ID
  - 成功返回0， 出错返回错误编号
- pthread_cancel并不等待线程终止，它仅仅是提出请求



Part 2 基本API函数

2.4 线程回收——pthread_join函数

- 功能：该函数用于等待某个线程终止
- 函数原型

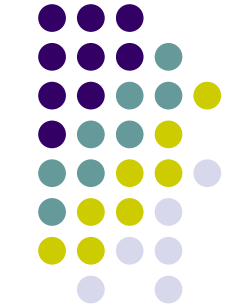
```
#include<pthread.h>
int pthread_join(pthread_t thread,  void **rval_ptr);
```

Part 2 基本API函数

2.4 线程回收——pthread_join函数

- 返回值与参数
  - 成功返回0，否则返回错误编号
  - thread：需要等待的线程ID
  - rval_ptr：
    - 若线程从启动例程返回，rval_ptr将包含返回码
    - 若线程由于pthread_exit终止，rval_ptr即pthread_exit的参数
    - 若线程被取消，由rval_ptr指定的内存单元就置为PTHREAD_CANCELED
    - 若不关心线程返回值，可将该参数设置为NULL

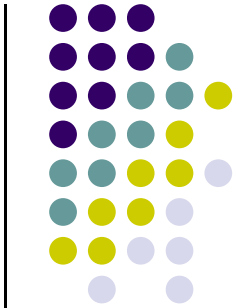
Part 2 基本API函数



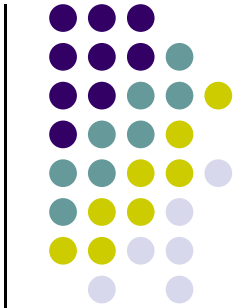
- 例4-1线程创建

  创建两个线程，线程1每秒打印全局变量的值并累计，线程2每2秒打印全局变量的值，两个线程打印5次后退出。main线程等待两个线程退出后显示两个线程已经退出。

  例4-1thrdcreat.c



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void task1(int *counter);
void task2(int *counter);
void cleanup(int counter1, int counter2);
int g1 = 0;
int g2 = 0;
int main(int argc, char *argv[])
{  pthread_t thrd1, thrd2;  int ret;  void *retval;
   ret = pthread_create(&thrd1, NULL, (void *)task1, (void *)&g1);
   printf("I'm in main function\n");
   ret = pthread_create(&thrd2, NULL, (void *)task2, (void *)&g2);
   pthread_join(thrd1,&retval);
   pthread_join(thrd2,NULL);
   cleanup(g1, g2);  getchar();
```



```
  printf("return val of task1 is %d\n", (int*)retval);
  cleanup(g1, g2);
  exit(EXIT_SUCCESS);
}
void task1(int *counter)
{  while(*counter < 5)
   { printf("task1 count: %d\n", *counter);
     (*counter)++;
      sleep(1);    }
   pthread_exit((void *)100);
}
void task2(int *counter)
{   while(*counter < 5)
    {  printf("task2 count: %d\n", *counter);
       (*counter)++;
       sleep(2);    }
    pthread_exit(NULL);
}
void cleanup(int counter1, int counter2)
{ printf("total iterations: %d\n", counter1 + counter2);}
```

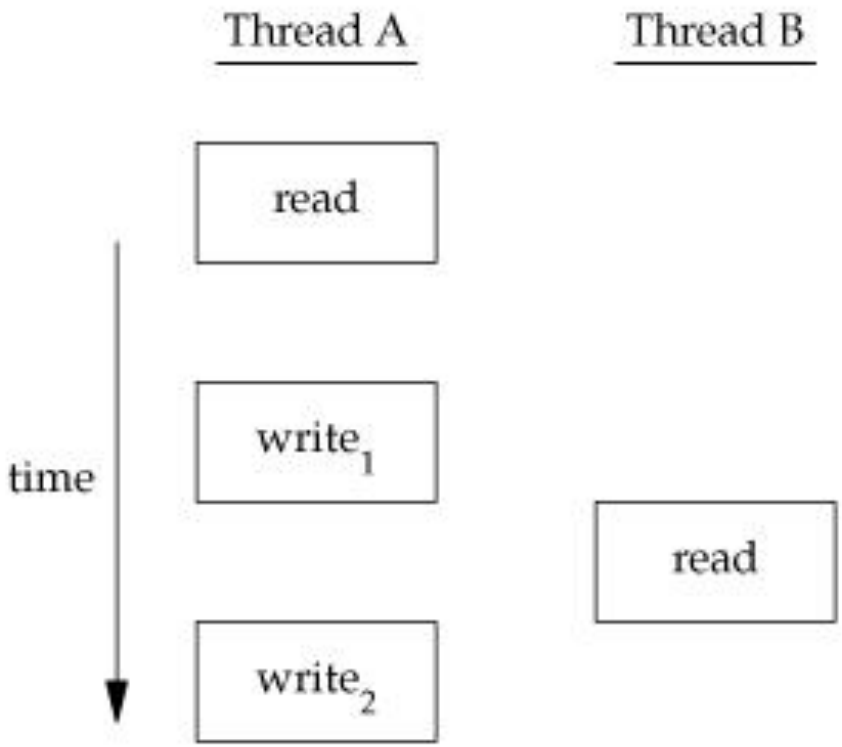
Part 3 线程同步

- 3.1线程同步的概念
- 为什么需要同步
  - 对同一个存储单元，至少存在两个执行体，其一读该单元，另一写该单元，则需要同步，避免不一致性
  - 在处理器架构中，对内存单元的修改，可能需要多个总线周期，因此读操作和写操作有可能交织在一起

Part 3 线程同步

3.1线程同步的概念

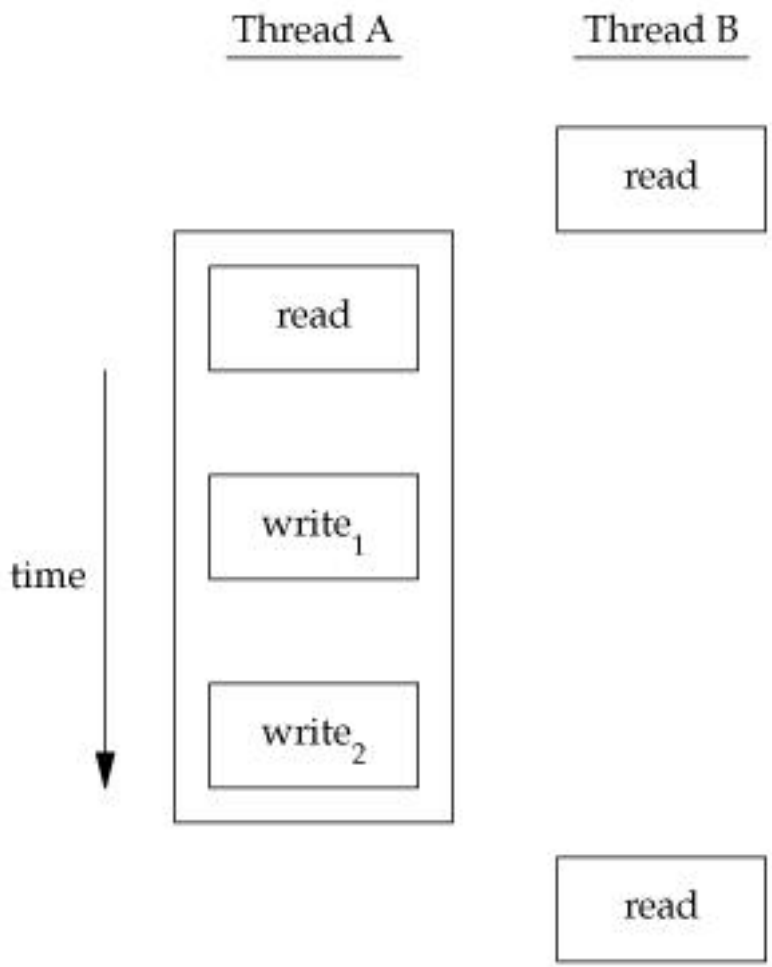
- 假设读操作需要一个总线周期
- 写操作需要两个总线周期
- 线程B和线程A冲突



Part 3 线程同步

3.1 线程同步的概念

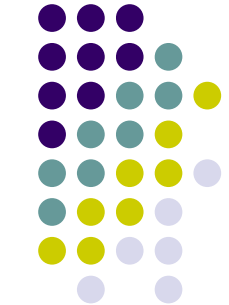
- 使用锁，以保证共享存储一次只能被一个线程访问
- 说明获取、释放锁的过程



线程同步的方法

- 互斥锁
- 读写锁
- 条件变量
- 信号量：用于线程的POSIX实时扩展

Part 3 线程同步



3.2 线程互斥锁

- 可以通过使用pthread的互斥接口保护数据，确保同一时间里只有一个线程访问数据
- 互斥锁mutex，本质上就是一把锁
  - 在访问共享资源前，对互斥锁进行加锁
  - 在访问完成后释放互斥锁上的锁
  - 对互斥锁进行加锁后，任何其他试图再次对互斥锁加锁的线程将会被阻塞，直到锁被释放

Part 3 线程同步

3.2 线程互斥锁函数

头文件：#include<pthread.h>

线程互斥锁初始化pthread_mutex_init

线程互斥锁销毁pthread_mutex_destroy

线程互斥锁加锁和解锁pthread_mutex_lock

pthread_mutex_unlock

Part 3 线程同步

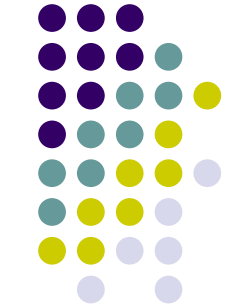
1 线程互斥锁初始化

- 互斥锁在使用前，必须要对互斥锁进行初始化
- 函数原型

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

- 参数与返回值
  - mutex：即互斥锁，类型是pthread_mutex_t
    注意：mutex必须指向有效的内存区域
  - attr：设置互斥锁的属性，通常可采用默认属性，即可将attr设为NULL。
  - 成功返回0，出错返回错误码

Part 3 线程同步



2 线程互斥锁销毁

- 互斥锁在使用完毕后，必须要对互斥锁进行销毁，以释放资源
- 函数原型

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

- 参数与返回值
  - mutex：即互斥锁
  - 成功返回0，出错返回错误码

Part 3 线程同步

4 线程互斥锁加锁和解锁

- 在对共享资源访问之前和访问之后，需要对互斥锁进行加锁和解锁操作
- 函数原型

```
#include<pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
Int pthread_mutex_unlock(
                        pthread_mutex_t *mutex);
```

Part 3 线程同步

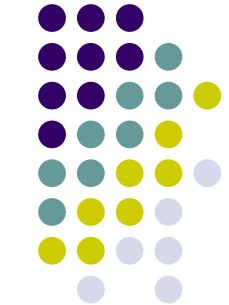
5 尝试锁

- 当使用pthread_mutex_lock时，若已被加锁，则调用线程将被阻塞。
- pthread_mutex_trylock让线程不阻塞，即实现非阻塞的语义
- 函数原型

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

- 调用该函数时，若互斥量未加锁，则锁住该互斥量，返回0；若互斥量已加锁，则函数直接返回失败，即EBUSY

Part 3 线程同步



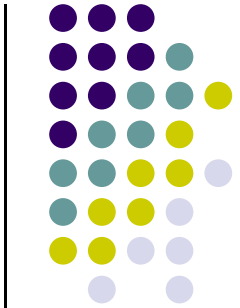
6 互斥锁的操作顺序

- 定义一个互斥锁pthread_mutex_t
- 调用pthread_mutex_init初始化互斥锁
- 调用pthread_mutex_lock或者pthread_mutex_trylock对互斥锁进行加锁操作
- 调用pthread_mutex_unlock对互斥锁解锁
- 调用pthread_mutex_destroy销毁互斥锁

Part 3 线程同步

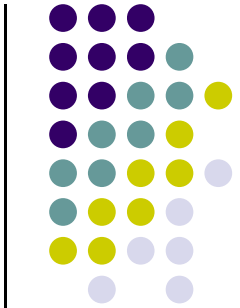
例 4-2线程互斥锁4-2lock_mutex.c

创建两个线程，线程1执行时加1，线程2减1。

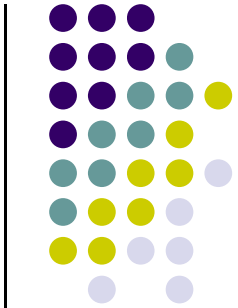


```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void task1(void);
void task2(void);
int sharedi=0;
pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZER;
int main(void) {
    pthread_t thrd1, thrd2;    int ret;
    ret = pthread_create(&thrd1, NULL, (void *)task1, NULL);
    if (ret) {      perror("pthread_create: task1");
                      exit(EXIT_FAILURE);      }
    ret = pthread_create(&thrd2, NULL, (void *)task2, NULL);
    if (ret) {      perror("pthread_create: task2");
                      exit(EXIT_FAILURE);      }
    pthread_join(thrd1, NULL);
    pthread_join(thrd2, NULL);
    pthread_mutex_destroy(&mutex);
    printf("sharedi = %d\n", sharedi);
}..
```



```
void task1(void)
{   long i,tmp;
    for(i=0; i<1000000; i++) {
/*  if(pthread_mutex_lock(&mutex) != 0) {
                perror("pthread_mutex_lock");
                exit(EXIT_FAILURE);                }
*/
    tmp = sharedi;
     tmp = tmp + 1;
     sharedi = tmp;
/*  if (pthread_mutex_unlock(&mutex) != 0) {
                perror("pthread_mutex_unlock");
                exit(EXIT_FAILURE);                }
*/
    }
}
```



```
void task2(void)
{   long i,tmp;
    for(i=0; i<1000000; i++) {
/*          if(pthread_mutex_lock(&mutex) != 0) {
                    perror("pthread_mutex_lock");
            exit(EXIT_FAILURE);                 }
*/
            tmp = sharedi;
            tmp = tmp - 1;
            sharedi = tmp;
/*
            if (pthread_mutex_unlock(&mutex) != 0) {
                    perror("pthread_mutex_unlock");
                    exit(EXIT_FAILURE);                 }
*/
    }
}
```

Part 3 线程同步

互斥锁特点

- 两种状态：锁定和非锁定
- 死锁问题：

  1.若线程试图对同一个互斥量加锁两次，那么它自身就会陷入死锁状态

  2.多个互斥量时可能出现死锁

  - 一个线程锁住互斥量A，等待互斥量B解锁
  - 另一个线程锁住B，而等待A

Part 3 线程同步

3.3 条件变量

因为互斥锁的状态只有锁定和非锁定两种状态，无法决定线程执行先后，有一定的局限。而条件变量通过允许线程阻塞和等待另一个线程发送信号实现线程的同步。

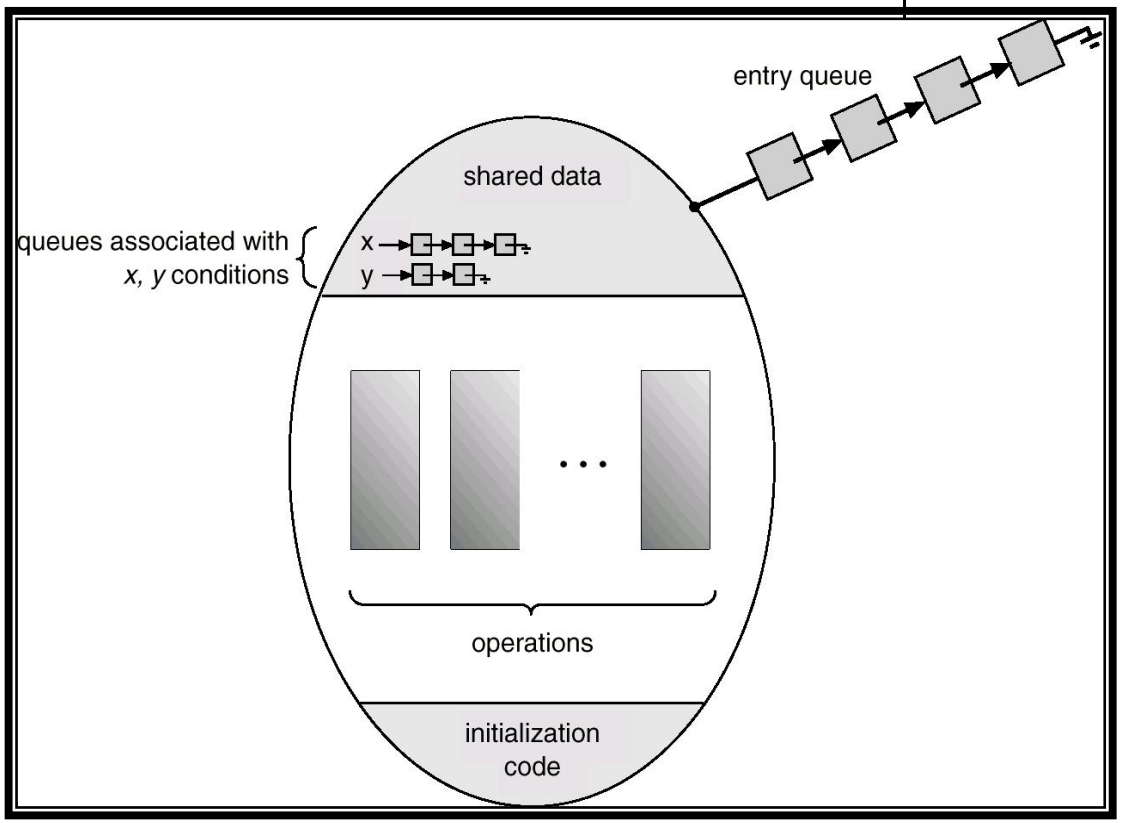
Part 3 线程同步

条件变量

condition x, y;

条件变量的操作

- 阻塞操作：wait
- 唤醒操作：signal



x.wait(): 进程阻塞直到另外一个进程调用x.signal()

x.signal()：唤醒另外一个进程

Part 3 线程同步

3.3 条件变量函数

头文件：#include <pthread.h>

初始化pthread_cond_init函数

销毁一个条件变量pthread_cond_destroy函数

阻塞等待一个条件变量pthread_cond_wait函数

唤醒某个线程pthread_cond_signal函数

唤醒所有线程pthread_cond_broadcast函数

Part 3 线程同步

3.3 条件变量函数

int pthread_cond_init (pthread_cond_t cond,pthread_condattr_t cond_attr);

功能：初始化条件变量

cond：待初始化的条件变量

cond_attr：条件变量的属性，通常为NULL，执行默认属性

Part 3 线程同步

3.3 条件变量函数

int pthread_cond_destroy(pthread_cond_t *cond);

功能：销毁条件变量

Part 3 线程同步

3.3 条件变量函数

int pthread_cond_wait (pthread_cond_t *cond,pthread_mutex_t *mutex)

功能：让调用者线程进入睡眠，并解锁一个互斥量

cond：线程睡眠的条件变量

mutex：线程睡眠前的要解锁的互斥量

必须放在pthread_mutex_lock()函数和pthread_mutex_unlock()函数之间

Part 3 线程同步

3.3 条件变量函数

int pthread_cond_signal(pthread_cond_t *cond)

功能：唤醒条件变量中的一个线程

注意：线程醒的前提条件是互斥量必须是解锁状态的，线程醒前会再次加锁，如果不能加锁就不会醒来。

Part 3 线程同步

3.3 条件变量函数

int pthread_cond_signal(pthread_cond_t *cond)

功能：唤醒条件变量中的一个线程

注意：线程醒的前提条件是互斥量必须是解锁状态的，线程醒前会再次加锁，如果不能加锁就不会醒来。

Part 3 线程同步

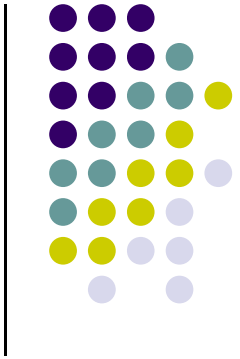
例 4-3线程条件变量4-3condition.c

创建三个线程，线程1输入，线程2读取，线程3读取

同步问题.

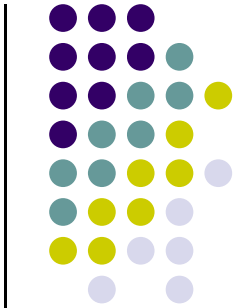
```
char buf[20] = "";
pthread_cond_t cond;
pthread_mutex_t lock;
void *thread1_handler(void *arg){
    while(1){
        fgets(buf, 10, stdin);
        buf[strlen(buf) - 1] = '\0';
        pthread_cond_signal(&cond);
    }
    pthread_exit(0);
}
void *thread2_handler(void *arg){
    while(1){
        pthread_mutex_lock(&lock);
        pthread_cond_wait(&cond, &lock);
        printf("thread2 buf:%s\n", buf);
        sleep(1);
        pthread_mutex_unlock(&lock);
    }
    pthread_exit(0);
}
```

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>
```

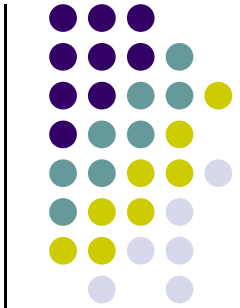


唤醒线程.

```
    pthread_mutex_lock(&lock);/*执行加锁操作*/
    pthread_cond_wait(&cond, &lock);/*线程执行阻塞，此时自动执行解锁，当线程收到唤醒信号，函数立即返回，此时在进入临界区之前，再次自动加锁*/
    printf("thread2 buf:%s\n", buf);/*临界区*/
    sleep(1);
    pthread_mutex_unlock(&lock);/*解除互斥锁*/
```



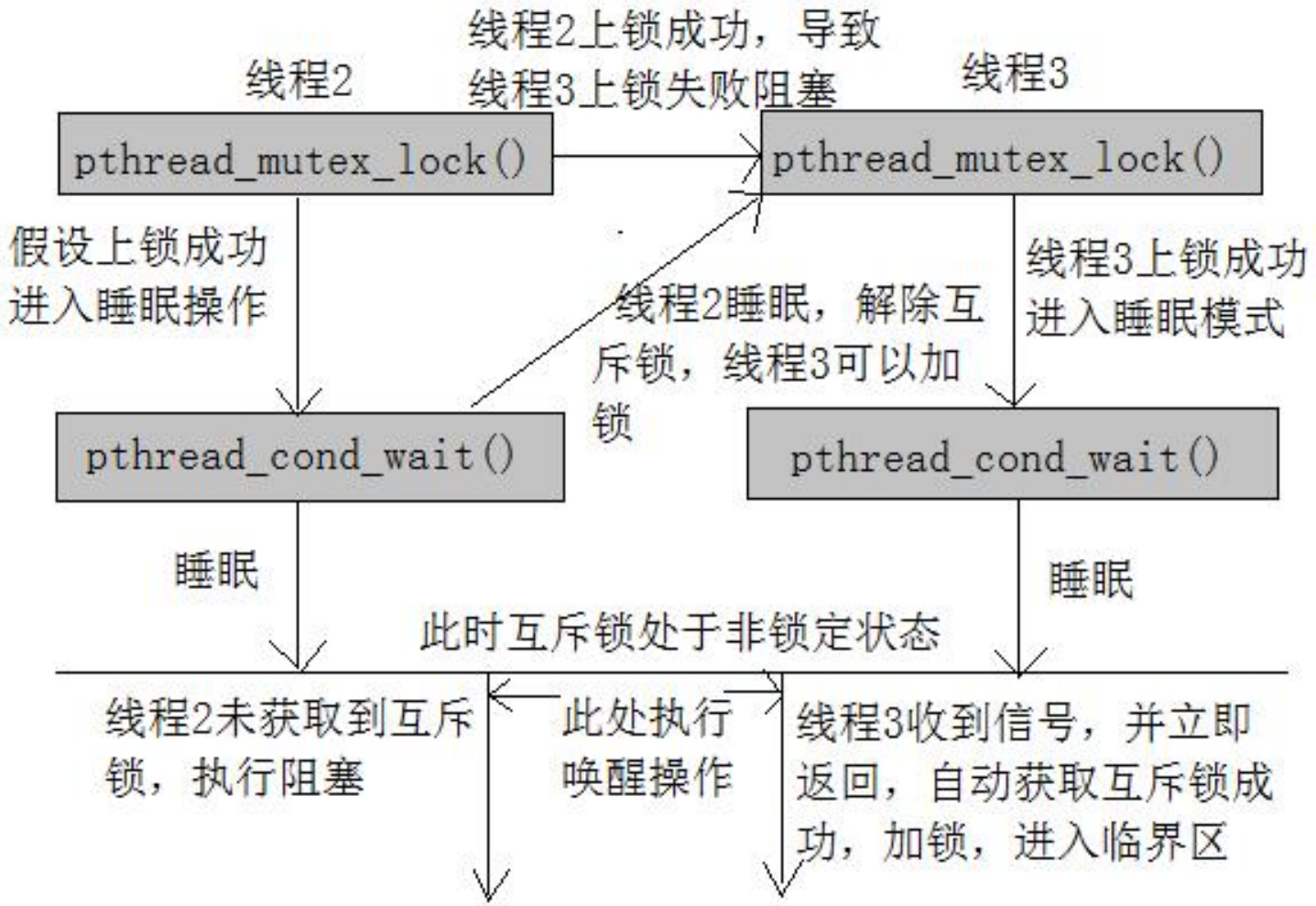
```
void *thread3_handler(void *arg){
    while(1){
        pthread_mutex_lock(&lock);
        pthread_cond_wait(&cond, &lock);
        printf("thread3 buf:%s\n", buf);
        sleep(1);
        pthread_mutex_unlock(&lock);
    }
    pthread_exit(0);
}
int main(int argc, const char *argv[])
{   pthread_t thread1, thread2, thread3;
    if(pthread_cond_init(&cond, NULL) != 0){
        perror("pthread_cond_init error");
    }
    if(pthread_mutex_init(&lock, NULL) != 0){
        perror("pthread_mutex_init error");
    }
```



```
    if(pthread_create(&thread1, NULL,
                                thread1_handler, NULL) != 0){
            perror("pthread_create1 error");
    }
   if(pthread_create(&thread2, NULL,
                                thread2_handler, NULL) != 0){
            perror("pthread_create2 error");
    }
    if(pthread_create(&thread3, NULL,
                                thread3_handler, NULL) != 0){
            perror("pthread_create3 error");
    }
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&cond);
    return 0;
}
```

Part 3 线程同步

例 4-3线程条件变量4-3condition.c



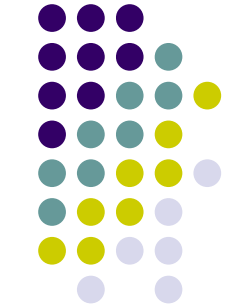
Part 3 线程同步

3.4 信号量

信号量也就是操作系统中所用到的PV 原语，它广泛用于进程或线程间的同步与互斥。信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。

Linux 实现了POSIX 的无名信号量，主要用于线程间的互斥同步。

Part 3 线程同步



3.4 信号量

- 函数定义

  #include <semaphore.h>

  int sem_init(sem_t *sem,int pshared,unsigned int value)

- 函数参数

  sem：信号量

  pshared：决定信号量能否在几个进程间共享。由于目前Linux还没有实现进程间共享信号量，所以这个值只能够取0

  value：信号量初始化值

- 函数返回值

  成功：0；　　出错：-1

Part 3 线程同步

3.4 信号量

- int sem_wait(sem_t *sem)
  int sem_trywait(sem_t *sem)
  int sem_post(sem_t *sem)
  int sem_getvalue(sem_t *sem,int * sval);
  int sem_destroy(sem_t *sem)
  函数传入值sem：信号量
  函数返回值
- 成功：0
  出错：-1

Part 3 线程同步

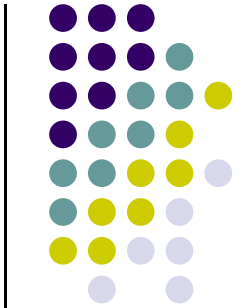
信号量举例

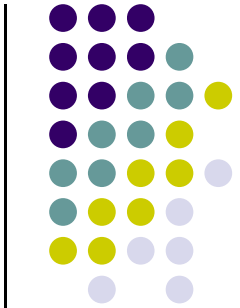
- 信号量互斥4-4sem_mutex.c
- 信号量同步 例4-5sem_syn.c

4-6sem.c

例4-4sem_mutex.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <sys/ipc.h>
#include <semaphore.h>
sem_t sem;
void task1(void);
void task2(void);
int sharedi=0;
int main(void) {
    int ret;pthread_t id1,id2;
    ret=sem_init(&sem,0,1);
    if(ret!=0) perror("sem_init");
    ret=pthread_create(&id1,NULL,(void *)task1, NULL);
    if(ret!=0)  perror("pthread create1");
    ret=pthread_create(&id2,NULL,(void *)task2, NULL);
    if(ret!=0)  perror("pthread create2");
    pthread_join(id1,NULL);
    pthread_join(id2,NULL);
    sem_destroy(&sem);
    printf("sharedi = %d\n", sharedi);
}
```

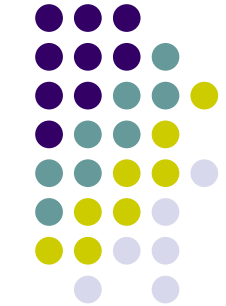


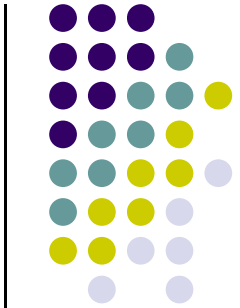


```
void task1(void)
{   long i,tmp;
    for(i=0; i<1000000; i++)
    {  sem_wait(&sem);
       tmp = sharedi;
       tmp = tmp + 1;
       sharedi = tmp;
       sem_post(&sem);     }
}
void task2(void)
{    long i,tmp;
     for(i=0; i<1000000; i++)
     {    sem_wait(&sem);
          tmp = sharedi;
          tmp = tmp + 1;
          sharedi = tmp;
          sem_post(&sem);          }
}
```

例4-5sem_syn.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>
#include <semaphore.h>
void *thread_function(void *arg);   sem_t sem;
#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int main()
{ int res;  pthread_t a_thread;  void *thread_result;
  res=sem_init(&sem,0,0);
  if(res!=0){    perror("Semophore initialization failed");
     exit(EXIT_FAILURE);   }
  res=pthread_create(&a_thread,NULL,thread_function,NULL);
  if(res!=0){    perror("Thread creation failed");    exit(EXIT_FAILURE);   }
  printf("Input some text.Enter 'end' to finish\n");
  while(strncmp("end",work_area,3)!=0){
    fgets(work_area,WORK_SIZE,stdin);    sem_post(&sem);  }
  printf("\n Waiting for thread to finish...\n");
  res=pthread_join(a_thread,&thread_result);
  if(res!=0){    perror("Thread join failed");    exit(EXIT_FAILURE);   }
  printf("Thread joined\n");
  sem_destroy(&sem);
  exit(EXIT_SUCCESS);
}
```

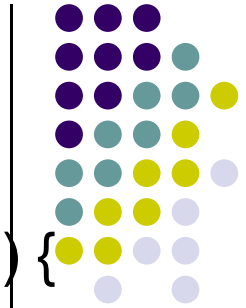




```
void *thread_function(void*arg){
  sem_wait(&sem);
  while(strncmp("end",work_area,3)!=0){
    printf("You input %d
   characters\n",strlen(work_area)-1);
    sem_wait(&sem);
  }
  pthread_exit(NULL);
}
```

例4-6sem.c

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
 #define MAXSIZE 10
int stack[MAXSIZE][2];
int size = 0;   sem_t sem;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void provide_data(void)
{   int i;
    for (i = 0; i < MAXSIZE; i++) {
            stack[i][0] = i;                stack[i][1] = i;    sem_post(&sem);    }
}
void handle_data1(void)
{   int i;
    while (pthread_mutex_lock(&mutex), ((i = size++) < MAXSIZE)) {
            pthread_mutex_unlock(&mutex);
            sem_wait(&sem);
            printf("Plus:    %d + %d = %d\n", stack[i][0], stack[i][1],
    stack[i][0] + stack[i][1]);
    }
    pthread_mutex_unlock(&mutex);
}
```



```
void handle_data2(void)
{   int i;
    while (pthread_mutex_lock(&mutex), ((i = size++) < MAXSIZE)) {
            pthread_mutex_unlock(&mutex);
            sem_wait(&sem);
            printf("Multiple: %d * %d = %d\n", stack[i][0], stack[i][1],
    stack[i][0] * stack[i][1]);          }
    pthread_mutex_unlock(&mutex);
}
int main(void)
{   pthread_t thrd1, thrd2, thrd3;
    sem_init(&sem, 0, 0);
    pthread_create(&thrd1, NULL, (void *)handle_data1, NULL);
    pthread_create(&thrd2, NULL, (void *)handle_data2, NULL);
    pthread_create(&thrd3, NULL, (void *)provide_data, NULL);
    pthread_join(thrd1, NULL);
    pthread_join(thrd2, NULL);
    pthread_join(thrd3, NULL);
    sem_destroy(&sem);
    return 0;
}
```

Part 4 线程属性

- pthread_create时，针对线程属性，传入的参数都是NULL。
- 实际上，可以通过构建pthread_attr_t结构体，设置若干线程属性
- 要使用该结构体，必须首先对其进行初始化；使用完毕后，需要销毁它

Part 4 线程属性

4.1 线程属性初始化和销毁

- 函数原型

```
#include<pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- 参数与返回值
  - 成功返回0，否则返回错误编号
  - attr：线程属性，确保attr指向的存储区域有效
  - 为了移植性，pthread_attr_t结构对应用程序是不可见的，应使用设置和查询等函数访问属性

Part 4 线程属性

4.2 设置分离状态属性

- 每个线程默认是非分离状态，线程被调用pthread_join的线程回收资源；如果一个非分离状态线程结束运行但没有被join，则它的状态类似于进程中的Zombie Process， 还有一部分资源没有被回收。
- 设置为分离属性，由系统释放资源函数，不需要其他线程调用pthread_join函数。分离属性设置方法：
  1.pthread_attr_setdetachstate函数，使调用线程处于分离状态，以便让OS在线程退出时收回它所占的资源
  2. 调用函数pthread_detach(threadid)或pthread_detach(pthread_self())分离。

Part 4 线程属性

4.2 设置分离状态属性方法1

- 函数原型

```
#include<pthread.h>
int pthread_attr_setdetachstate(
                pthread_attr_t *attr, int detachstate);
```

- 参数与返回值
  - 成功返回0，否则返回错误编号
  - attr：线程属性结构体指针
  - detachstate：
    - PTHREAD_CREATE_DETACHED：以分离状态启动线程
    - PTHREAD_CREATE_JOINABLE：正常启动线程（默认属性）

Part 4 线程属性

4.2 获取分离状态属性

- 函数原型

```
#include<pthread.h>
int pthread_attr_getdetachstate(
        const pthread_attr_t *attr, int *detachstate);
```

- 参数与返回值
  - 成功返回0，否则返回错误编号
  - attr：线程属性结构体指针
  - detachstate指向的整数，可被设置为以下值
    - PTHREAD_CREATE_DETACHED：以分离状态启动线程
    - PTHREAD_CREATE_JOINABLE：正常启动线程（默认属性）

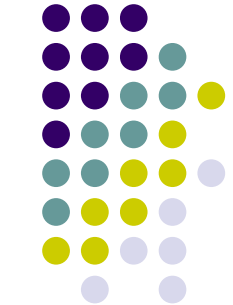
Part 4 线程属性

4.2 设置分离状态属性1举例

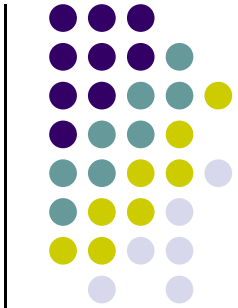
例4-7thread_detach1.c

程序头文件：

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
```

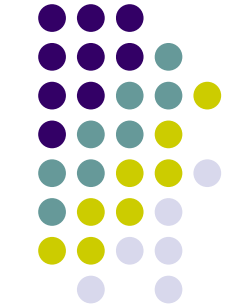


```
void *thread_function(void *arg);
char message[]="Hello World";     int thread_finished=0;
int main()
{ int res;  pthread_t a_thread;  pthread_attr_t thread_attr;
  res=pthread_attr_init(&thread_attr);
  if(res!=0){    perror("Attribute creation failed");
     exit(EXIT_FAILURE);   }
res=pthread_attr_setdetachstate(&thread_attr,PTHREAD_CREATE_D
    ETACHED);
  if(res!=0){    perror("setting detached attribute failed");
     exit(EXIT_FAILURE);   }
res=pthread_create(&a_thread,&thread_attr,thread_function,(void*)mes
    sage);
  if(res!=0){    perror("Thread creation failed");
     exit(EXIT_FAILURE);   }
  (void)pthread_attr_destroy(&thread_attr);
  while(!thread_finished){
   printf("Waiting for thread to say it's finished...\n");   sleep(1);  }
  printf("Other thread finished,bye\n");
  exit(EXIT_SUCCESS);
}
```



```
void *thread_function(void*arg){
  printf("thread_function is running.Argument
   was %s\n",(char*)arg);
  sleep(4);
  printf("second thread setting finished flag,and
   exiting now\n");
  thread_finished=1;
  pthread_exit(NULL);
}
```

Part 4 线程属性



4.2 设置分离状态属性方法2

1. pthread_detach(threadid)函数

   功能：使线程ID为threadid的线程处于分离状态。

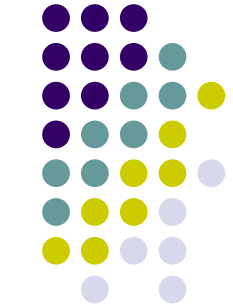
   一旦线程处于分离状态，该线程终止时底层资源立即被回收。通常是主线程使用pthread_create()创建子线程以后，调用该函数分离刚刚创建的子线程。

2. pthread_detach(pthread_self())函数

   功能：线程设置自己的分离状态。

   被创建的线程也可以自己分离自己，pthread_self() 函数返回线程自己的ID。

Part 4 线程属性

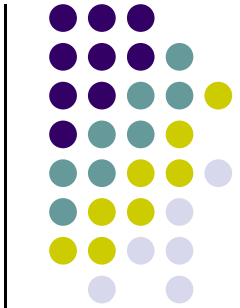


4.2 设置分离状态属性2举例

例4-8thread_detach2.c

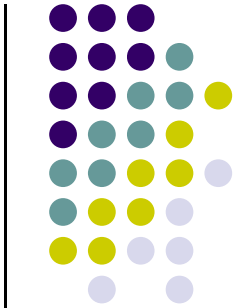
程序头文件：

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



```
void *thread_handler(void *arg){
    pthread_detach(pthread_self());

    int count = *((int *)arg);
    while(count > 0){
        printf("thread...\n");
        sleep(1);
        count--;
    }
    return NULL;
}
```



```
int main(int argc, const char *argv[])
{   pthread_t thread;
    int arg = 3;
    if(pthread_create(&thread, NULL,
    thread_handler, (void *)&arg) != 0){
          perror("pthread_create1 error");        }
    int temp = 0;
    sleep(1);
    if(pthread_join(thread, NULL) == 0){
          printf("pthread wait success\n");
          temp = 0;    }
    else{       printf("pthread wait failed\n");temp = 0;}
    return temp;
}
```

Part 4 线程属性

4.3 设置取消线程属性

- 函数原型

```
#include<pthread.h>
int pthread_setcancelstate(int *state, int *oldstate);
```

- 参数与返回值
  - 成功返回0，否则返回错误编号
  - state：取值为
    - PTHREAD_CANCEL_ENABLE：允许线程接收取消请求（默认设置）
    - PTHREAD_CANCEL_DISABLE：忽略取消请求

Part 4 线程属性

4.3 设置取消线程属性

int pthread_setcanceltype(int *type, int *oldtype);

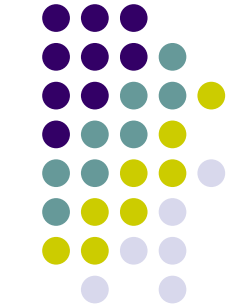
- 参数与返回值
  - 成功返回0，否则返回错误编号
  - type ：取值为
    - PTHREAD_CANCEL_ASYNCHRONOUS：收到取消请求后立即行动
    - PTHREAD_CANCEL_DEFERRED：收到取消请求后延迟行动（默认设置）

Part 4 线程属性

4.3 设置取消线程属性

void pthread_testcancel(void);

- 当线程取消功能处于启用状态且取消类型设置为延迟模式时，pthread_testcancel() 函数有效
- 在执行到pthread_testcancel的位置时，线程才可能响应cancel退出进程



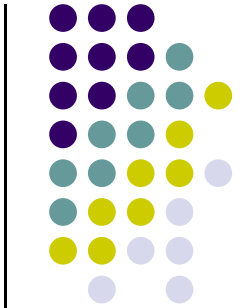
Part 4 线程属性

4.3 设置取消线程属性举例

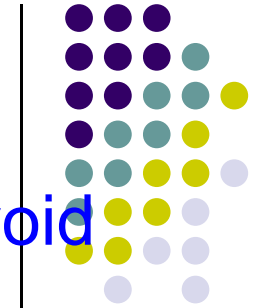
4-9thread_cancel.c

程序头文件

```
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
```



```
int g1 = 0;   int g2 = 0;  pthread_t thrd1, thrd2;
void task1(int *counter)
{   volatile int i;
    //pthread_setcancelstate(PTHREAD_CANCEL_DISABL
    E, NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHR
    ONOUS, NULL);
    while((*counter) < 5) {    printf("task1 count: %d\n",
    *counter);  (*counter)++;sleep(1);}
    printf("*counter in task1 is : %d\n",*counter);
}
void task2(int *counter)
{   while(*counter < 5) {       printf("task2 count: %d\n",
    *counter);  (*counter)++; }
}
void cleanup(int counter1, int counter2)
{   printf("total iterations: %d\n", counter1 + counter2);
}
```



```
int main(int argc, char *argv[])
{   pthread_t thrd1, thrd2;    int ret;
    ret = pthread_create(&thrd1, NULL, (void *)task1, (void
    *)&g1);
    if (ret) {      perror("pthread_create: task1");
            exit(EXIT_FAILURE);}
    ret = pthread_create(&thrd2, NULL, (void *)task2, (void
    *)&g2);
    if (ret) {      perror("pthread_create: task2");
            exit(EXIT_FAILURE);}
    sleep(2);
    pthread_join(thrd2, NULL);
    pthread_cancel(thrd1);
    printf("in main , after pthread_cancel\n");
    pthread_join(thrd1, NULL);
    cleanup(g1, g2);
    exit(EXIT_SUCCESS);
}
```

掌握的知识点

- 线程的创建、终止函数
- 线程互斥锁、信号量的应用
- 线程主要属性的设置