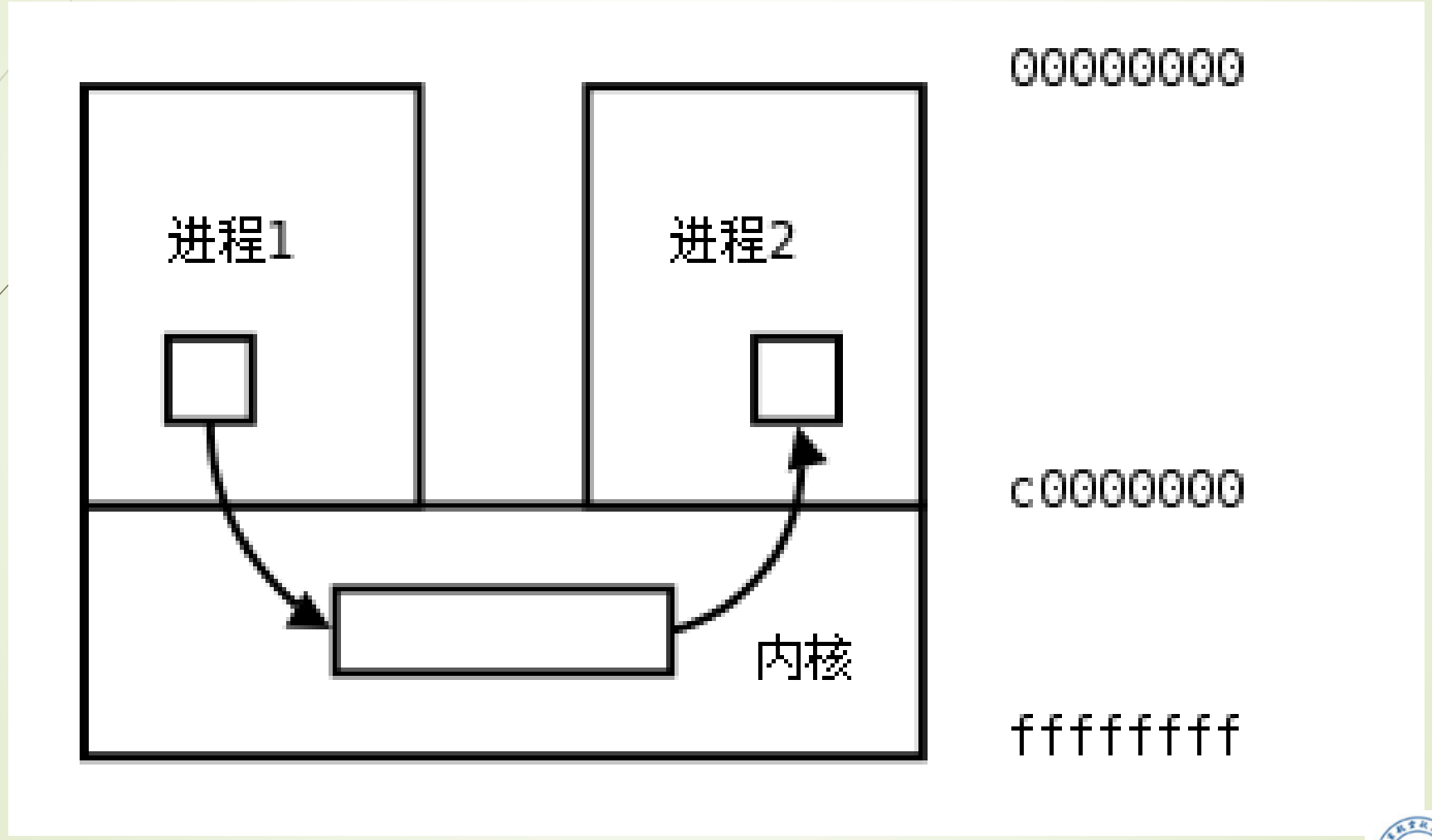
Review： 进程及其通信

- 进程：代表一个程序的运行，有自己独立的空间
- 进程间通信除共享内存外，要跨空间
  - 管道，跟普通文件的使用及应用场合对比
  - 信号
  - 信号量
  - 消息队列
- 同步和互斥的概念
  - 用信号量实现同步互斥

Review: 可用于空间理解的示意图



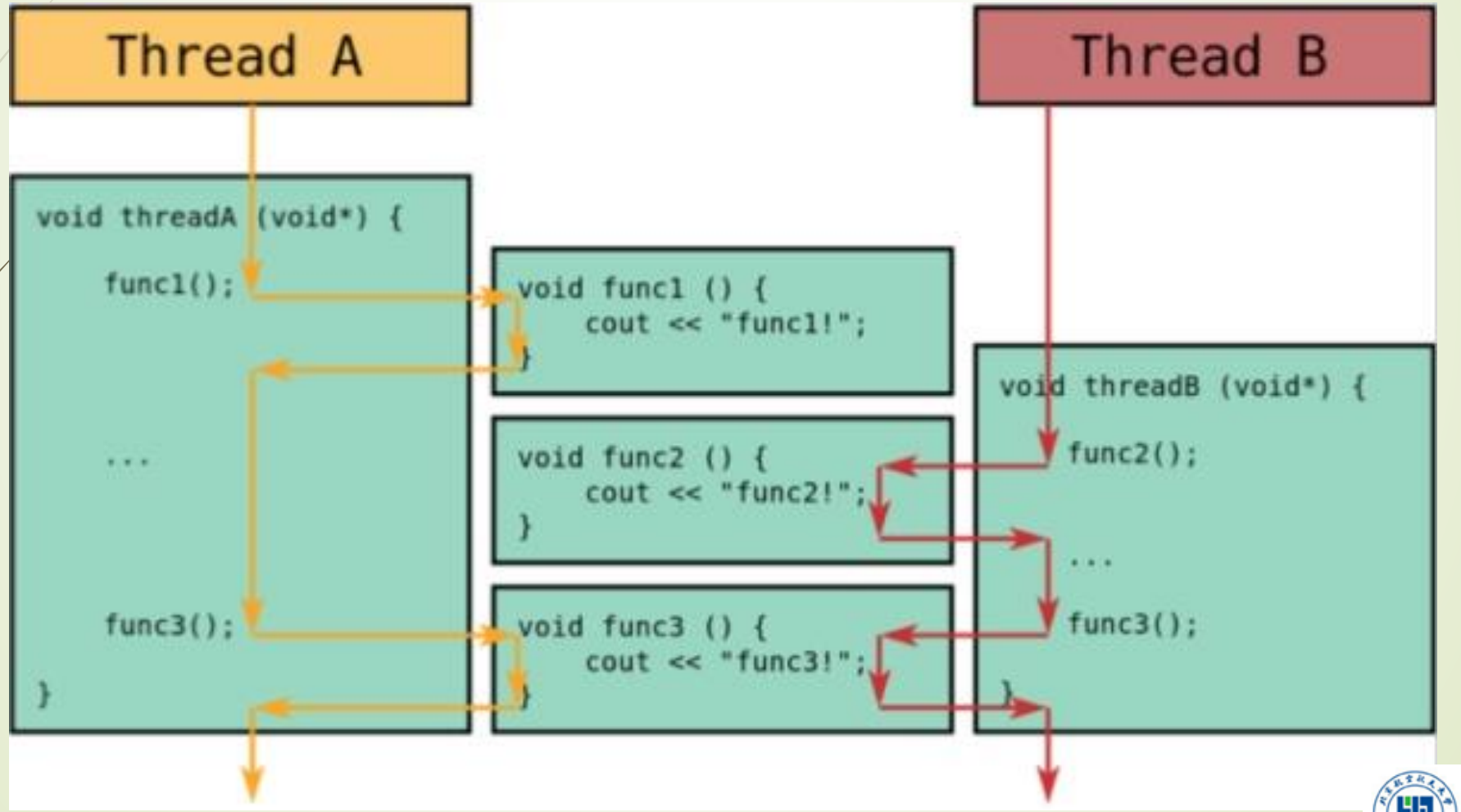
北京航空航天大學 软件学院

第8讲 多线程编程

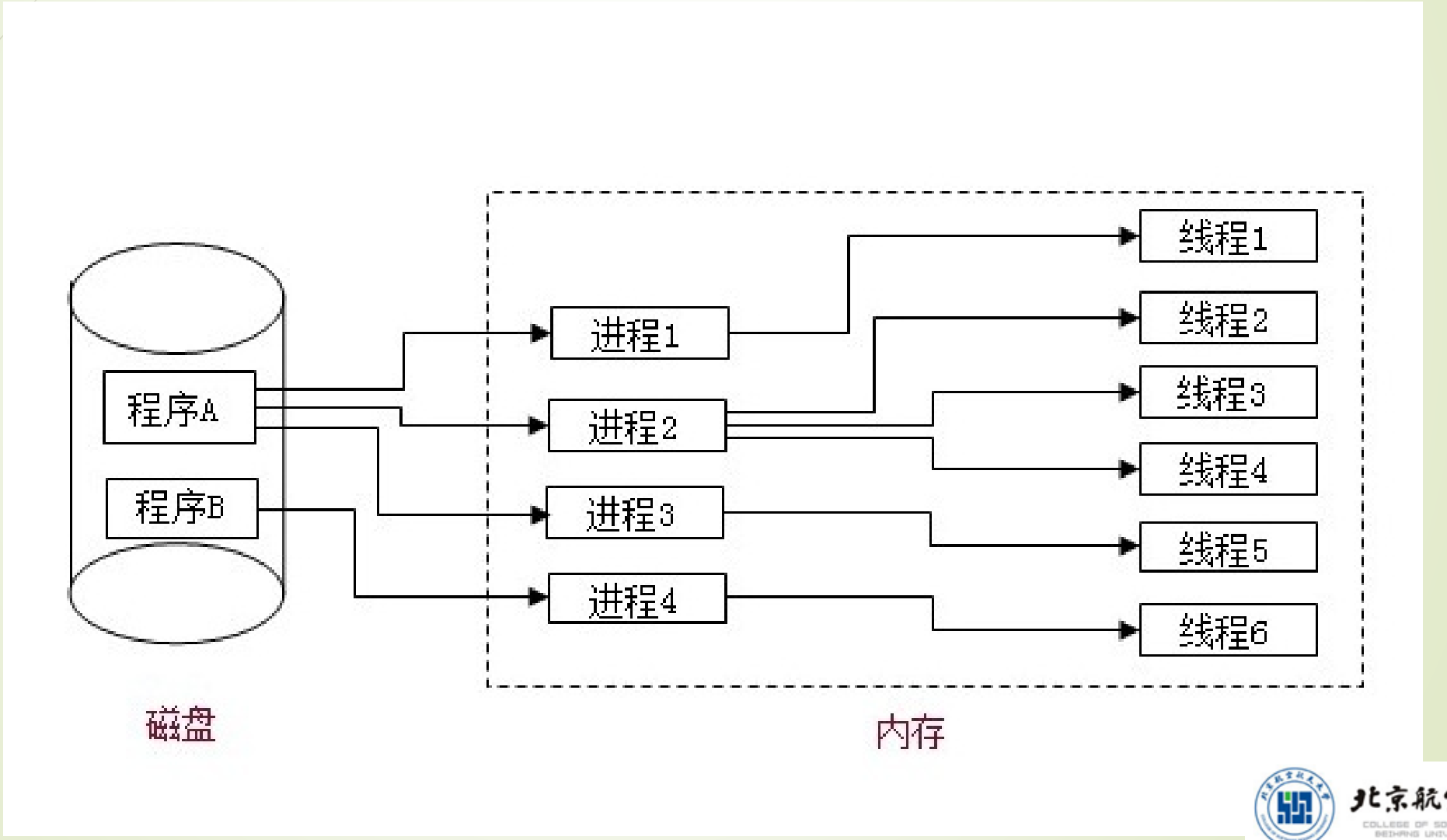
第八讲 线程

- 8.1 线程的概念
- 8.2 线程状态与编程接口
  - 线程的创建与参数传递
  - 线程的终止 pthread_exit
  - 线程的挂起 pthread_join
  - 线程的取消 phtread_cancel
- 8.3 线程的同步与互斥
  - 互斥量及其使用方法
  - 条件变量及其使用方法

程序员感觉的线程：能从自己开始运行的函数



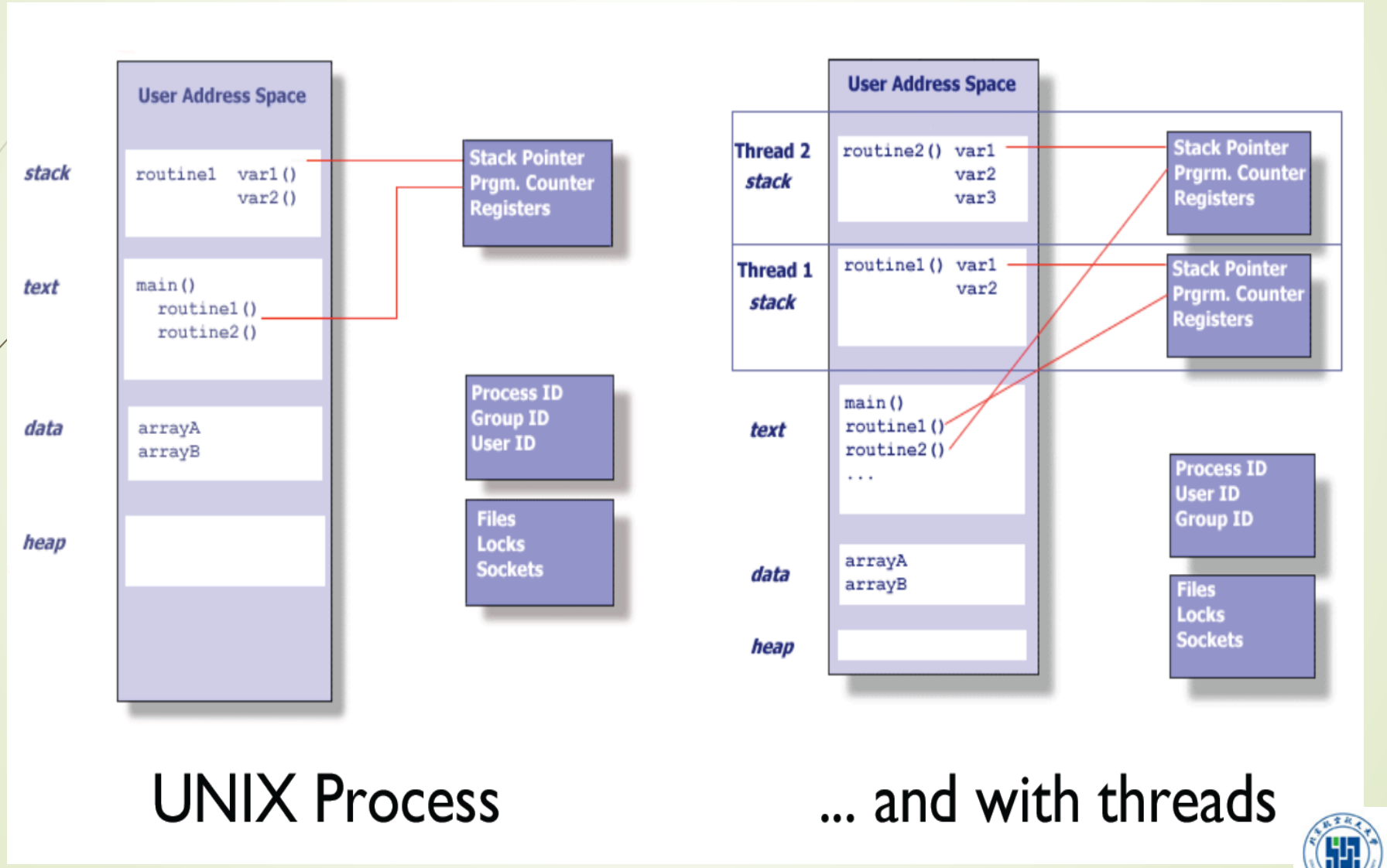
程序、进程和线程的关系



8.1 线程的概念

- 线程是CPU调度的单位。
- 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

（了解）进程和线程的代码视角

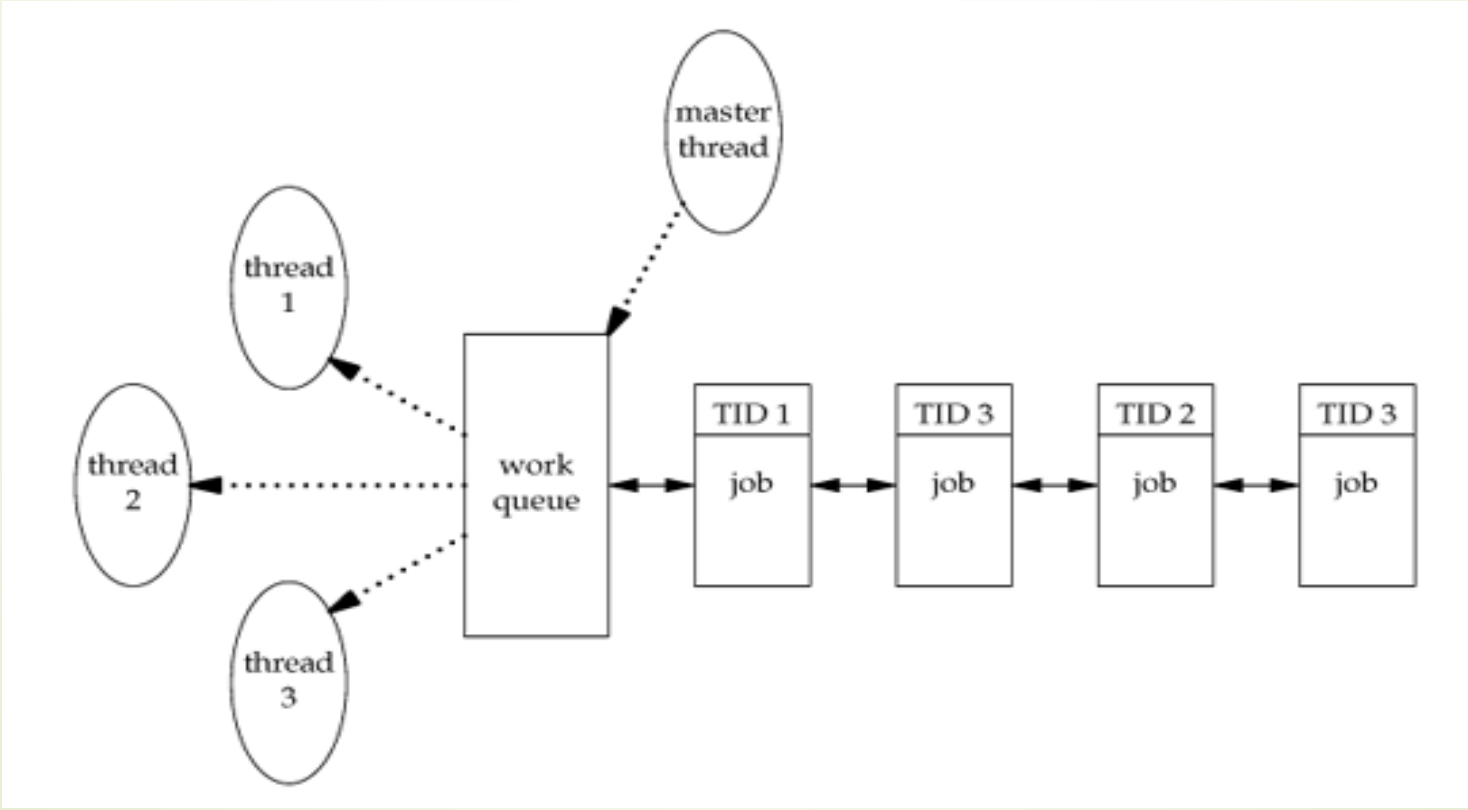


线程的概念

- 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;

线程的一种工作模式

- Scalar和Erlang，Actor



北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

第八讲 线程

- 8.1 线程的概念
- 8.2 线程状态与编程接口
  - 线程的创建与参数传递
  - 线程的终止 pthread_exit
  - 线程的挂起 pthread_join
  - 线程的取消 phtread_cancel
- 8.3 线程的同步与互斥
  - 互斥量及其使用方法
  - 条件变量及其使用方法

线程的创建

- pthread_create函数用于创建一个线程
- 函数原型

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

- 参数与返回值
  - tidp：当pthread_create成功返回时，该函数将线程ID存储在tidp指向的内存区域中

pthread_create函数

- 参数与返回值
  - attr：用于定制各种不同的线程属性，通常可设为NULL，采用默认线程属性
  - start_rtn：线程的入口函数，即新创建的线程从该函数开始执行。该函数只有一个参数，即arg，返回一个指针
  - arg：作为start_rtn的第一个参数
  - 成功返回0，出错时返回各种错误码

线程的创建

- 新创建的线程，将继承调用pthread_create函数的线程的信号屏蔽字，但新线程的未决信号集将被清除
- 示例8.1
  - 下页
- 示例8.2
  - 下下页

```
/*thread_creat.c*/
#include <pthread.h> #include<stdlib.h>  #include <string.h>
typedef struct student {  int age;   char name[20];  } STU;
void *create(void *arg) {  //线程将要执行的代码
        STU *temp=(STU *)arg;     printf("The following is transferred to thread\n");
        printf("STU age is %d\n",temp->age);    printf("STU name is %s\n",temp->name);
}
int main(int argc,char *argv[]){
     pthread_t tidp;   int error;
     STU *stu=(STU *)malloc(sizeof(STU));
     stu->age=20;
     strcpy(stu->name,"abcdefg");
     error=pthread_create(&tidp,NULL,create,(void *)stu);
     if(error!=0)    {   printf("pthread_create failed "); return -1;   }     //  return 0;
     pthread_join(tidp,NULL);
   }例8-1
```

```
/*multhread.c*/  #include <pthread.h> #include <stdio.h> #define MAX 100
int global=1;
void *t1_execute(void *arg){  //线程1执行的代码
      while(global<MAX) {   fprintf(stdout,"The global value is %d\n",global);   fflush(stdout); }
}
void *t2_execute(void *arg){//线程2执行的代码a
      while (global<MAX){ global=global+1;   //改变进程变量i的值。 sleep(1);   }
}
void main(){
      pthread_t pid1,pid2;
      int error1,error2;

      error1=pthread_create(&pid1,NULL,t1_execute,NULL);
      error2=pthread_create(&pid2,NULL,t2_execute,NULL);

      pthread_join(pid2,NULL);
      pthread_join(pid1,NULL);
}例8-2
```

线程ID的获取

- pthread_self函数可以使调用线程获取自己的线程ID
- 函数原型

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

- 返回调用线程的线程ID

```
/*threadid.c*/
#include <pthread.h>  #include <stdio.h>  #include <unistd.h>
void *t1_exe(void *arg){
    printf("In new created thread\n") ;
    printf("My pid is %d and my pthread is %lu\n",getpid(),(unsigned long int)pthread_self());
}
void main() {
    pthread_t pid1;  int error1;
    error1=pthread_create(&pid1,NULL, t1_exe,NULL);
    if (error1!=0)    { printf("pthread_create failed ");    return ;   }
    printf("In main process\n") ;
    printf("My pid is %d and my pthread is %lu\n",getpid(),(unsigned long int)pthread_self());
    pthread_join(pid1,NULL);
     return ;
}//例8-6
```

第八讲 线程

- 8.1 线程的概念
- 8.2 线程状态与编程接口
  - 线程的创建与参数传递
  - 线程的终止 pthread_exit
  - 线程的挂起 pthread_join
  - 线程的取消 phtread_cancel
- 8.3 线程的同步与互斥
  - 互斥量及其使用方法
  - 条件变量及其使用方法

线程的终止

- 进程中任一线程调用exit、_Exit、_exit，都会导致整个进程终止
- 当线程接收到信号，若信号的处理动作是终止进程，则进程将被终止（后面分析信号与线程的交互）
- 如何只终止某个线程，而不终止整个进程？

线程的终止

- 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

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

```
/*thread_exit.c*/  #include <pthread.h>   #include <stdio.h>  #include<stdlib.h>
void *t1_execute(void *arg){
        while(1) {     printf("in thread1\n");  }
}
void *t2_execute(void *arg){
        sleep(1);
        pthread_exit(NULL); //用exit(0);语句替换
}
void main(){
        pthread_t pid1,pid2;
        int error1,error2;
        error1=pthread_create(&pid1,NULL, t1_execute,NULL);
        error2=pthread_create(&pid2,NULL, t2_execute,NULL);
        if(error1!=0 | | error2!=0)    {
            printf("pthread_create failed ");        return ;
        }
        pthread_join(pid1,NULL);
        pthread_join(pid2,NULL);
        return ;
}//例8-3
```

线程取消时的资源释放编程

- 外界取消操作是不可预见的，因此的确需要一个机制来简化用于资源释放的编程。

- 线程为了访问临界资源而为其加上锁，但在访问过程中被外界取消
  - 如果线程处于响应取消状态，且采用异步方式响应
  - 或者在释放独占锁以前的运行路径上存在取消点
- 则该临界资源将永远处于锁定状态得不到释放。

pthread_cleanup_push()/pop()函数

- POSIX提供的一对自动释放资源的函数
- 从pthread_cleanup_push()的调用点到pthread_cleanup_pop()之间的程序段中的终止动作
  - 包括调用 pthread_exit()和取消点终止
  - 都将执行pthread_cleanup_push()所指定的清理函数。
- 定义如下：
  - void pthread_cleanup_push(void (*routine) (void *), void *arg)
  - void pthread_cleanup_pop(int execute)

push()/pop()函数的运行过程

- pthread_cleanup_push()/pthread_cleanup_pop()采用先入后出的栈结构管理
  - void routine(void *arg)函数在调用pthread_cleanup_push()时压入清理函数栈
- 多次对pthread_cleanup_push()的调用将在清理函数栈中形成一个函数链
  - 在执行该函数链时按照压栈的相反顺序弹出

push()/pop()函数的运行过程（续）

- execute参数表示执行到pthread_cleanup_pop()时是否在弹出清理函数的同时执行该函数
  - 为0表示不执行，非0为执行
  - 这个参数并不影响异常终止时清理函数的执行
    - 当pthread_cleanup_pop()函数的参数为0时，仅仅在本线程调用pthread_exit函数或者其它线程对本线程调用 pthread_cancel函数时，才在弹出“清理函数”的同时执行该“清理函数”

```
/*mulfunc.c*/  #include <pthread.h> #include <stdio.h>  #include<stdlib.h>
void  clean_1(void  *arg){     printf("%s\n" ,(char *)arg);  }
void *t1_execute(void *arg){
    pthread_cleanup_push(clean_1,"thread  first handler");
    pthread_cleanup_push(clean_1,"thread  second hadler");
    if ( *(int*)arg==1 )    printf("abc\n");
    else    exit(0);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
}
void main(){
    pthread_t pid1;  int error1;  int i=1;
    error1=pthread_create(&pid1,NULL, t1_execute,(void *)&i);
    if(error1!=0)    {
        printf("pthread_create failed ");
        return ;
     }
     pthread_join(pid1,NULL);
     return ;
} //例8-4
```

第八讲 线程

- 8.1 线程的概念
- 8.2 线程状态与编程接口
  - 线程的创建与参数传递
  - 线程的终止 pthread_exit
  - 线程的挂起 pthread_join
  - 线程的取消 phtread_cancel
- 8.3 线程的同步与互斥
  - 互斥量及其使用方法
  - 条件变量及其使用方法

pthread_join函数

- 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);
```

- 调用该函数的线程将一直阻塞，直到指定的线程调用pthread_exit、从启动例程中返回、被取消

pthread_join函数

int pthread_join(pthread_t thread，void **rval_ptr);

- 参数
  - thread：需要等待的线程ID
  - rval_ptr：
    - 若线程从启动例程返回，rval_ptr中将包含返回码
    - 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
- 返回值
  - 成功返回0，否则返回错误编号

pthread_join第二个参数 void **rval_ptr

- 为什么pthread_join的第二个参数类型是指针的指针？
  - 指针的指针基本原理？传值与传指针的区别？
  - pthread_exit的一个目标是，把一个指针传递给pthread_join函数
  - pthread_join函数的思路是：通过参数的返回值，将该指针值返回给pthread_join的调用者

回调函数的自定义参数处理

- 什么是回调函数
  - 系统的回调
  - 自定义的回调，观察者模式
- 回调函数往往提供自定义参数
  - 如pthread_create、pthread_exit
  - 当需要传递复杂数据时，可以将结构体指针或者对象指针，作为自定义参数传递
  - 但必须要保证，该指针指向的内存区域在回调函数中有效
  - pthread_create的处理。何时删除对象？谁分配谁释放？

```
/*join.c*/  #include <pthread.h>  #include <stdio.h>
void *t1_exe(void *arg){    printf("The first thread: \n") ;    int i;
    for(i=1 ;i<6 ;i++)       printf("%d\n",i) ;
}
void *t2_exe(void *arg){    int i;
    pthread_join(*(pthread_t*)arg, NULL);  printf("The second thread: \n") ;
    for(i=6 ;i<11 ;i++)     printf("%d\n",i) ;
}
void main() {
    pthread_t pid1,pid2;
    int error1,error2;
    error1=pthread_create(&pid1,NULL, t1_exe,NULL);
    error2=pthread_create(&pid2,NULL, t2_exe,(void *)&pid1);
    if (error1!=0||error2!=0)    {    printf("pthread_create failed ");   return ;    }
    pthread_join(pid2,NULL);
    return ;
}  //例8-5  线程2等线程1运行完再运行
```

第八讲 线程

- 8.1 线程的概念
- 8.2 线程状态与编程接口
  - 线程的创建与参数传递
  - 线程的终止 pthread_exit
  - 线程的挂起 pthread_join
  - 线程的取消 phtread_cancel
- 8.3 线程的同步与互斥
  - 互斥量及其使用方法
  - 条件变量及其使用方法

取消点

- 类比，正在打球(将被中止的线程），过程中不看手机的（不处理信号）
  - （在取消点）看到手机通知开会，中止打球并退出
- 根据POSIX标准，pthread_join()、pthread_testcancel()、pthread_cond_wait()、pthread_cond_timedwait()、sem_wait()、sigwait()等函数以及read()、write()等会引起阻塞的系统调用都是Cancelation-point，而其他pthread函数都不会引起Cancelation动作。

为什么要用pthread_testcancel()

- 但是pthread_cancel的手册页声称，由于LinuxThread库与C库结合得不好，因而目前C库函数都不是Cancelation-point；
- 但CANCEL信号会使线程从阻塞的系统调用中退出，并置EINTR错误码，因此可以在需要作为Cancelation-point的系统调用前后调用 pthread_testcancel()，从而达到POSIX标准所要求的目标，即如下代码段：
  - pthread_testcancel();
  - retcode = read(fd, buffer, length);
  - pthread_testcancel();

pthread_cancel函数

- 线程通过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
  - tid：需要取消的线程ID
- 返回值
  - 成功返回0， 出错返回错误编号

pthread_cancel函数

- 在默认情况下
  - pthread_cancel函数会使得线程ID等于tid的线程，如同其调用了参数为PTHREAD_CANCELED的pthread_exit
- 但pthread_cancel并不等待线程终止，它仅仅是提出请求
  - 线程可以选择忽略取消方式或者控制取消方式

北京航空航天大学 软件学院

第八讲 线程

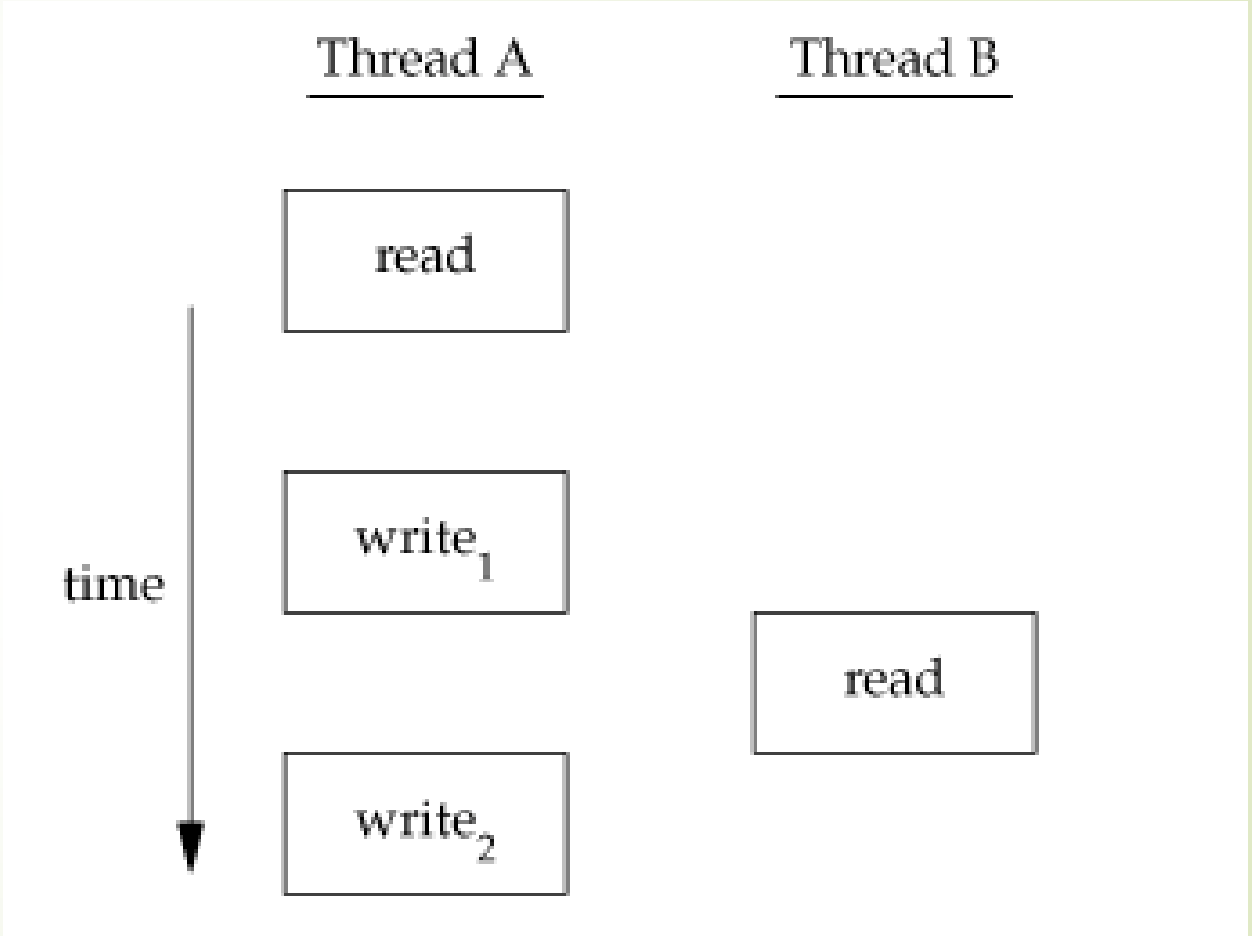
- 8.1 线程的概念
- 8.2 线程状态与编程接口
  - 线程的创建与参数传递
  - 线程的终止 pthread_exit
  - 线程的挂起 pthread_join
  - 线程的取消 phtread_cancel
- 8.3 线程的同步与互斥
  - 互斥量及其使用方法
  - 条件变量及其使用方法

线程同步的概念

- 与进程涵义相同，但具体实现方式有区别
- 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

线程同步的示例

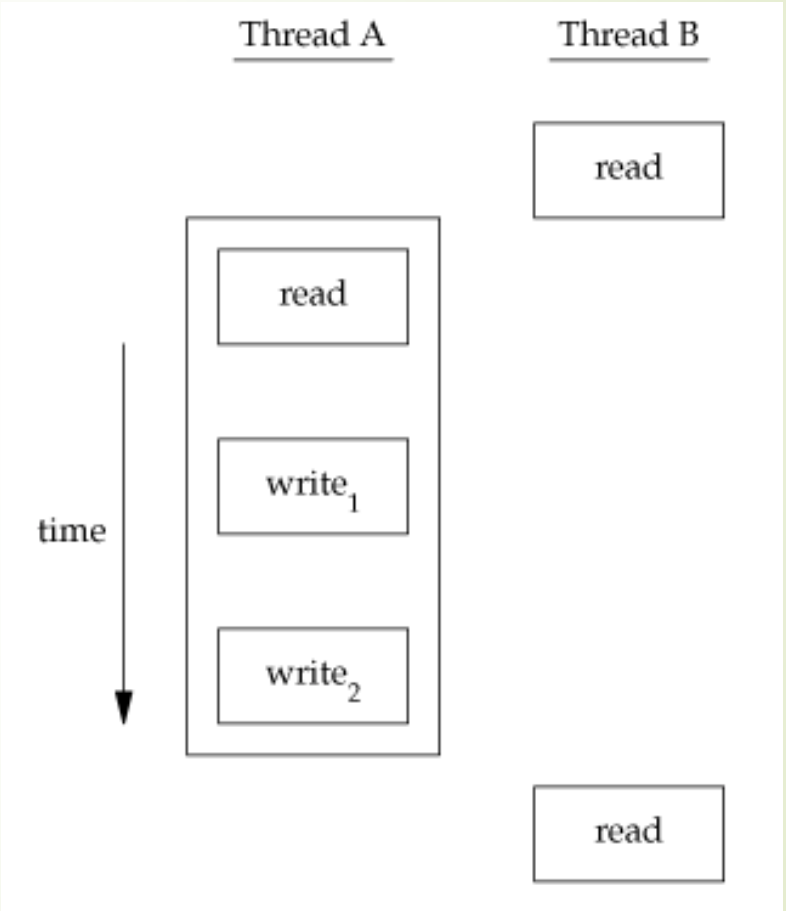
- 假设读操作需要一个总线周期
- 写操作需要两个总线周期
- 线程B和线程A冲突



北京航空航天大学 软件学院

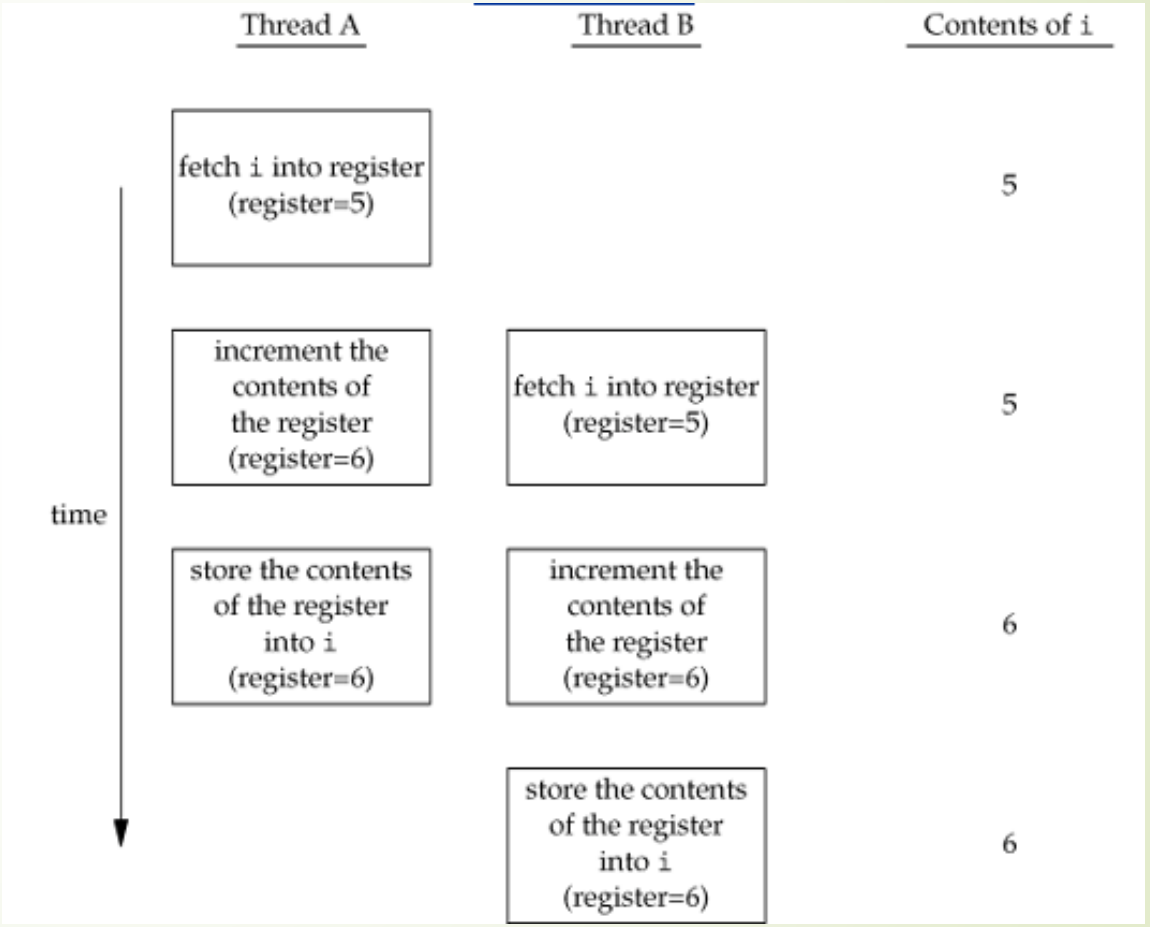
解决上述问题的方法

- 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



线程同步的必要性

- 通常，对一个存储单元的访问，要经历三个步骤
  - 将内存单元中的数据，读入寄存器
  - 对寄存器中的值进行运算
  - 将寄存器中的值，写回内存单元
- 无锁时的情况
  - 交叉存取



总结与思考

- 单线程的程序，需要对存储同步访问吗？
- 若需要，能用锁的机制吗？

第八讲 线程

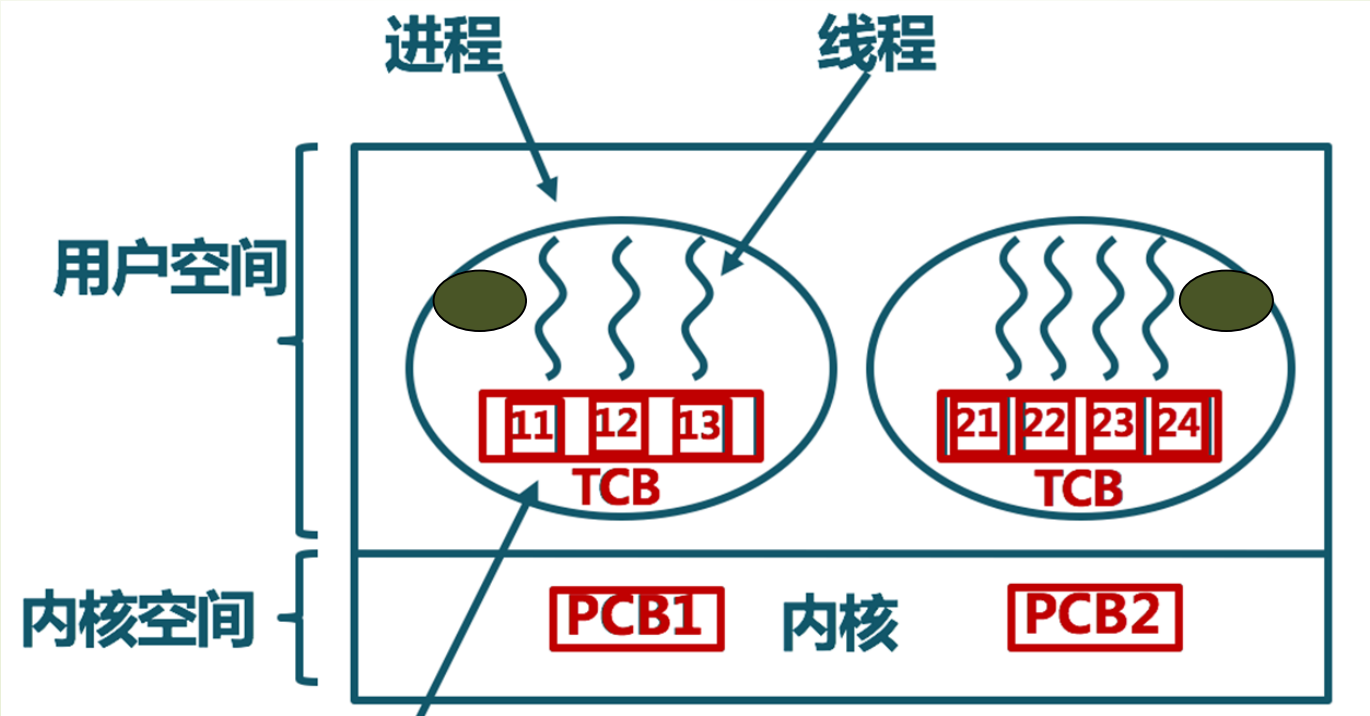
- 8.1 线程的概念
- 8.2 线程状态与编程接口
  - 线程的创建与参数传递
  - 线程的终止 pthread_exit
  - 线程的挂起 pthread_join
  - 线程的取消 phtread_cancel
- 8.3 线程的同步与互斥
  - 互斥量及其使用方法
  - 条件变量及其使用方法

互斥量

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

互斥量mutex

- 线程占用互斥量，然后访问共享资源，最后释放互斥量。实现互斥机制
- 只能被一个线程持有
- 排队等待



北京航空航天大学 软件学院

互斥量使用时的操作顺序

- 定义一个互斥量pthread_mutex_t
- 调用pthread_mutex_init初始化互斥量
- 调用pthread_mutex_lock或者pthread_mutex_tryplock对互斥量进行加锁操作
- 调用pthread_mutex_unlock对互斥量解锁
- 调用pthread_mutex_destroy销毁互斥量

互斥量的操作函数

- POSIX定义了一个宏PTHREAD_MUTEX_INITIALIZER来静态初始化互斥量
- 动态初始化方式是采用int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)

- int pthread_mutex_lock(pthread_mutex_t *mutex)
- int pthread_mutex_unlock(pthread_mutex_t *mutex)
- int pthread_mutex_trylock(pthread_mutex_t *mutex)

互斥量的初始化函数

- 互斥量在使用前，必须要对互斥量进行初始化
- 函数原型

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

- 参数1
  - mutex：即互斥量，类型是pthread_mutex_t

    注意：mutex必须指向有效的内存区域

互斥量的初始化函数（续）

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
```

- 参数2
  - attr：互斥量的属性，通常可采用默认值，即将attr设为NULL。
  - 下页详细讨论互斥量的属性
- 返回值
  - 成功返回0，出错返回错误码

互斥量的属性

- PTHREAD_MUTEX_TIMED_NP
  - 这是缺省值，也就是普通锁。当一个线程加锁以后，其余请求锁的线程将形成一个等待队列，并在解锁后按优先级获得锁。这种锁策略保证了资源分配的公平性
- PTHREAD_MUTEX_RECURSIVE_NP
  - 嵌套锁，允许同一个线程对同一个锁成功获得多次，并通过多次unlock解锁。如果是不同线程请求，则在加锁线程解锁时重新竞争
- PTHREAD_MUTEX_ERRORCHECK_NP
  - 检错锁，如果同一个线程请求同一个锁，则返回EDEADLK，否则与PTHREAD_MUTEX_TIMED_NP类型动作相同。这样就保证当不允许多次加锁时不会出现最简单情况下的死锁
- PTHREAD_MUTEX_ADAPTIVE_NP
  - 适应锁，动作最简单的锁类型，仅等待解锁后重新竞争

Linux中Mutex的注意事项

- POSIX线程锁机制的Linux实现都不是取消点，因此，延迟取消类型的线程不会因收到取消信号而离开加锁等待。
- 值得注意的是，如果线程在加锁后解锁前被取消，锁将永远保持锁定状态，因此如果在临界区段内有取消点存在，或者设置了异步取消类型，则必须在退出回调函数中解锁
- 这个锁机制同时也不是异步信号安全的，也就是说，不应该在信号处理过程中使用互斥量，否则容易造成死锁。

互斥量的销毁

- 互斥量在使用完毕后，必须要对互斥量进行销毁，以释放资源
- 函数原型

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

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

互斥量的加锁和解锁操作

- 在对共享资源访问之前和访问之后，需要对互斥量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);
```

- 回忆锁的语义

尝试锁

- 当使用pthread_mutex_lock时，若已被加锁，则调用线程将被阻塞。有没有办法让线程不阻塞，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

互斥量的操作顺序（回顾）

- 定义一个互斥量pthread_mutex_t
- 调用pthread_mutex_init初始化互斥量
- 调用pthread_mutex_lock或者pthread_mutex_tryplock对互斥量进行加锁操作
- 调用pthread_mutex_unlock对互斥量解锁
- 调用pthread_mutex_destroy销毁互斥量

北京航空航天大学 软件学院

死锁

- 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
  - 一种不太容易察觉的情况（自己等自己）
- 多个互斥量时可能出现死锁
  - 一个线程锁住互斥量A，等等待互斥量B解锁
  - 另一个线程锁住B，而等待A （相互循环等）

死锁的避免方法

- 按照固定的顺序对互斥量进行加锁操作，如先锁A，再锁B
- 当对互斥量加锁顺序的控制很困难时，可以先使用pthread_mutex_trylock，若不能获取锁，可释放已占有的锁

```
/*nomutex.c*/    static FILE *fd ;  //文件描述符    char str[]=    "aaaaaaaaaaaaaaaaaaa
Aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbcccdddeee
fffggghhhiiiooopppdddtttsssddasdfasfasddddddddddddddddddddddddddddddddddddd";
char str2[]="1112223334445556677777777777777777777777777777";
int i=0 ;
void * t1_exe(void *arg){     if ( fd != NULL){
            while (i<100){
                  if  ( fwrite((char *)arg,strlen((char *)arg),1,fd) ==-1 ) {
                        perror("write failed: ");        pthread_exit(NULL);  } //fflush(fd);
                  i++;     }          } }
void main() {
       pthread_t pid1,pid2,pid3;       int error1,error2,error3;      fd=fopen("thread.txt","w");
       if (fd==NULL){      printf("open failed\n");       exit(0);    }
       error1=pthread_create(&pid1,NULL, t1_exe,(void *)str);    //写str
       error2=pthread_create(&pid2,NULL, t1_exe,(void *)str2); //写str2
        if (error1!=0||error2!=0)    {      printf("pthread_create failed ");         return ;     }
        pthread_join(pid1,NULL);    pthread_join(pid2,NULL);
        fclose(fd);
        return ;
}//例8-7 无锁的情况
```

```
/*withmutex.c*/
char str[]= "abcdefghijklmnopqrstuvwxyz";    pthread_mutex_t mutex; //互斥量
int index2=0;
void * t1_exe(void *arg){
      while (index2<strlen(str)-1){
        pthread_mutex_lock (&mutex);
         printf("The %dth element of array is %c\n",index2,str[index2]);
         index2++;
         pthread_mutex_unlock(&mutex);
     }
  }
void main() {
     pthread_t pid1,pid2;       int error1,error2;
     pthread_mutex_init (&mutex,NULL);  //初始化互斥量
     error1=pthread_create(&pid1,NULL, t1_exe,NULL);
     error2=pthread_create(&pid2,NULL, t1_exe,NULL);
     if (error1!=0||error2!=0)   {   printf("pthread_create failed ");   return ;   }
     pthread_join(pid1,NULL);
     pthread_join(pid2,NULL);          return ;
}//例8-7    有锁的情况
```

第八讲 线程

- 8.1 线程的概念
- 8.2 线程状态与编程接口
  - 线程的创建与参数传递
  - 线程的终止 pthread_exit
  - 线程的挂起 pthread_join
  - 线程的取消 phtread_cancel
- 8.3 线程的同步与互斥
  - 互斥量及其使用方法
  - 条件变量及其使用方法

第八讲 线程

- 8.1 线程的概念
- 8.2 线程状态与编程接口
  - 线程的创建与参数传递
  - 线程的终止 pthread_exit
  - 线程的挂起 pthread_join
  - 线程的取消 phtread_cancel
- 8.3 线程的同步与互斥
  - 互斥量及其使用方法
  - 条件变量及其使用方法

条件变量

- 现有一需求，线程A先执行某操作后（即条件），线程B才能执行另一操作，该如何实现？
- 条件变量与互斥量一起使用时，允许线程以无竞争的方式等待特定条件的发生
- 与互斥量类似，条件变量也需要初始化和销毁

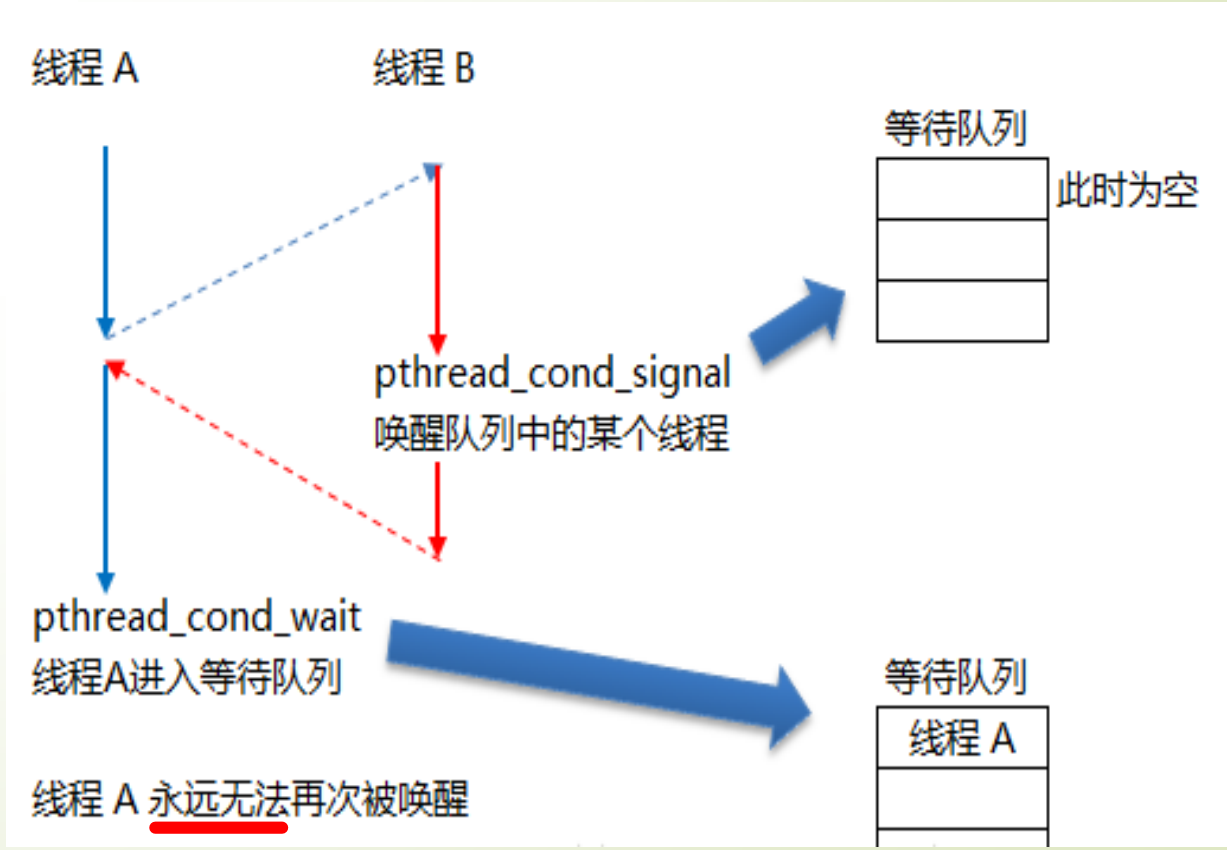
条件变量

- 条件变量是利用线程间共享的全局变量进行同步的一种机制
- 一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。
- 为什么有了互斥量，还要条件变量？
  - 互斥量可被“据为已有”，使用结束后唤醒排队等待的线程
  - 不容易做到一个线程感知另一个线程的“状态”
  - 条件变量，可让一个线程感知到另一个线程的“条件”

北京航空航天大学 软件学院

条件变量的使用

- pthread_cond_init → 初使化条件变量
- pthread_cond_destroy → 销毁条件变量
- pthread_cond_wait → 在一个条件之上等待
- pthread_cond_signal → 当条件满足的时候，可以向等待线程发起通知
- pthread_cond_broadcast → 向等待的所有线程发起通知



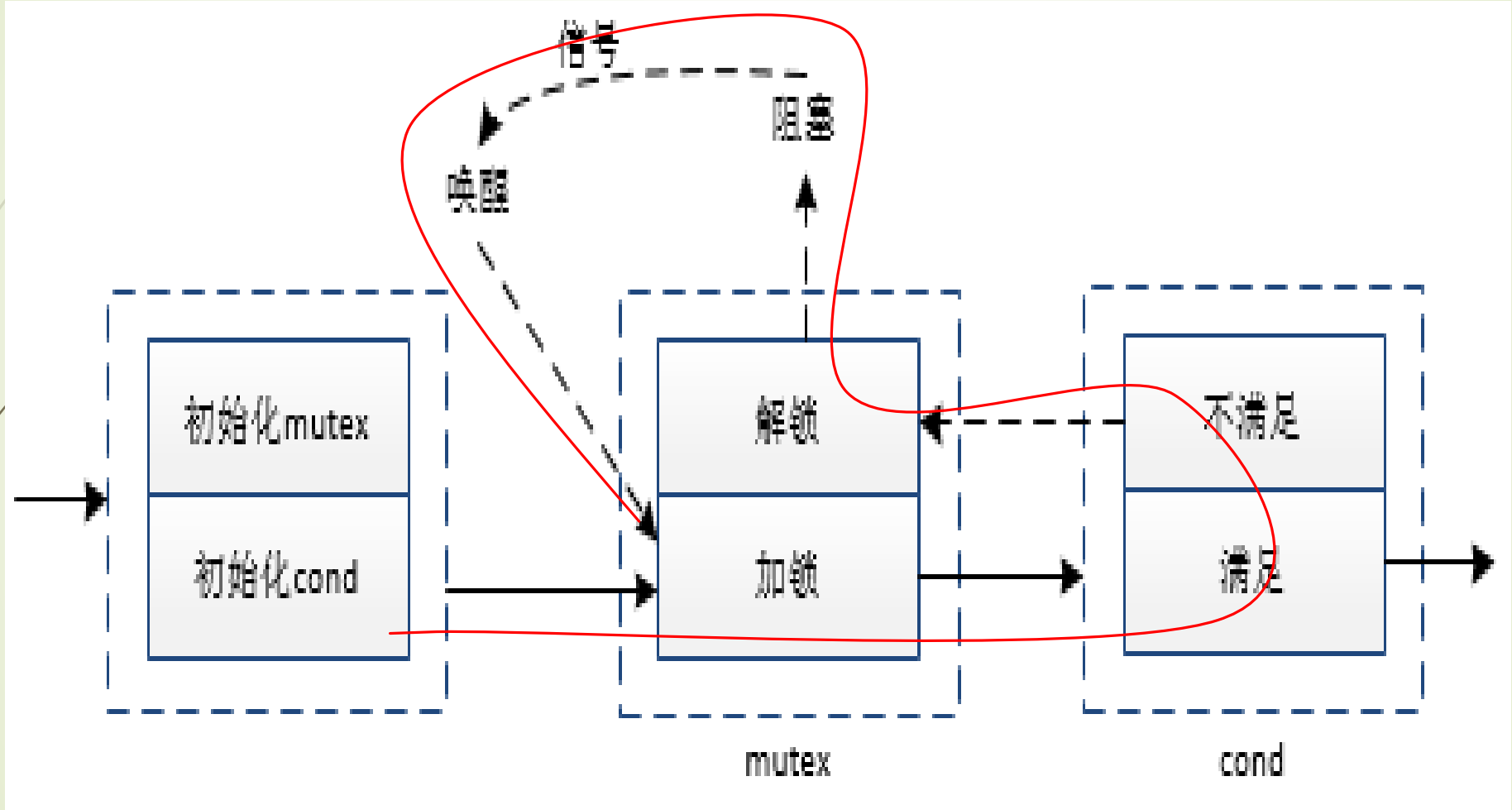
pthread_cond_wait中的互斥量

- 条件变量的使用总是和一个互斥量结合在一起
  - int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
- mutex互斥量必须是普通锁或适应锁
  - 普通锁PTHREAD_MUTEX_TIMED_NP）
  - 适应锁（PTHREAD_MUTEX_ADAPTIVE_NP）

pthread_cond_wait中的互斥量作用流程

- 图示在下页
- 在调用pthread_cond_wait()前必须由本线程加锁，而在更新条件等待队列以前，mutex保持锁定状态，并在线程挂起进入等待前解锁。
- 在条件满足从而离开pthread_cond_wait()之前，mutex将被重新加锁，以与进入pthread_cond_wait()前的加锁动作对应

pthread_cond_wait中的互斥量作用流程



北京航空航天大学 软件学院

条件变量的注意事项

- pthread_cond_wait()和pthread_cond_timedwait()都被实现为取消点，因此，在该处等待的线程将立即重新运行，在重新锁定mutex后离开pthread_cond_wait()，然后执行取消动作。
- 也就是说如果pthread_cond_wait()被取消，mutex是保持锁定状态的，因而需要定义退出回调函数来为其解锁。
- 条件变量函数不是异步信号安全的，不应当在信号处理程序中进行调用。特别要注意，如果在信号处理程序中调用 pthread_cond_signal 或 pthread_cond_boardcast 函数，可能导致调用线程死锁

北京航空航天大学 软件学院

条件变量的初始化和销毁函数

- 函数原型

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

- 参数和返回值
  - cond：条件变量
  - attr：条件变量属性，若为NULL，则使用默认属性
  - 成功返回0，出错返回错误编号

等待线程 使用等待函数 等待条件的发生

- pthread_cond_wait函数将使调用线程进入阻塞状态，直到条件为真
- 函数原型

```
#include<pthread.h>
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

- 参数与返回值
  - cond：条件变量
  - mutex：互斥量
  - 成功返回0，否则返回错误编号

等待条件函数需要配合一个互斥量

- 为什么pthread_cond_wait需要互斥量
  - 在调用pthread_cond_wait前，需要使互斥量处于锁住状态
  - 这样pthread_cond_wait函数，可以以原子的方式，将调用线程放到等待条件的线程列表上
- pthread_cond_wait函数内的特殊操作
  - 在线程阻塞前，调用pthread_mutex_unlock
  - 在线程唤醒后，调用pthread_mutex_lock

等待条件的发生 的程序流程

- 等待线程的操作顺序
  - 调用pthread_mutex_lock
  - 调用pthread_cond_wait
  - 调用pthread_mutex_unlock

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

准备条件线程 通知 等待线程条件已满足

- pthread_cond_signal和pthread_cond_broadcast可以通知等待的线程，条件已经满足
  - pthread_cond_signal唤醒某一个等待该条件的线程
  - pthread_cond_broadcast唤醒等待该条件的所有线程

北京航空航天大学 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

通知函数 的说明

- 函数原型

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

- 参数与返回值
  - cond：条件变量
- 返回值
  - 成功返回0，否则返回错误编号

等待线程与被等待线程的流程总结

- 等待线程
  - 调用pthread_mutex_lock
  - while（判断条件）pthread_cond_wait
    - 重复检查条件是由于线程可能不是被pthread_cond_signal唤醒，可能是由信号等唤醒（mutex）
  - 调用pthread_mutex_unlock
- 被等待线程
  - 调用pthread_mutex_lock
  - 修改条件
  - 调用pthread_mutex_unlock
  - 调用pthread_mutex_broadcast等

条件变量使用注意事项

- POSIX.1说明：pthread_cond_broadcast等函数的调用无需考虑调用线程是否拥有锁，并建议在lock和unlock以外的区域调用。为什么？
- 假设系统中有线程1和线程2
  - 线程1获取mutex，在进行数据处理的时候，线程2也想获取mutex，但是此时被线程1所占用，线程2进入休眠，等待mutex被释放

条件变量使用注意事项

- 线程1做完数据处理后，调用pthread_cond_signal唤醒等待队列中某个线程，在本例中也就是线程2。线程1在调用pthread_mutex_unlock前，因为系统调度的原因，线程2获取使用CPU的权利，那么它就想要开始处理数据，但是在开始处理之前，mutex必须被获取，线程1正在使用mutex，所以线程2被迫再次进入休眠
- 然后就是线程1执行pthread_mutex_unlock（）后，线程2方能被再次唤醒。
- 十分低效

带超时功能的等待函数

- 函数原型

```
#include<pthread.h>
int pthread_cond_timedwait(
                    pthread_cond_t *cond,
                pthread_mutex_t *mutex,
                const timespec *timeout);
```

- 参数与返回值
  - cond：条件变量
  - mutex：互斥量

带超时功能的等待函数

- 参数和返回值
  - timeout：超时时间，是一个绝对时间，而非相对时间

```
struct timespec {
    time_t tv_sec;  //秒数
    long tv_nsec;   //纳秒
};
```

带超时功能的等待函数

- 超时值的设置

```
void
maketimeout(struct timespec *tsp, long minutes)
{
        struct timeval now;

        /* get the current time */
        gettimeofday(&now);
        tsp->tv_sec = now.tv_sec;
        tsp->tv_nsec = now.tv_usec * 1000; /* usec to nsec */
        /* add the offset to get timeout value */
        tsp->tv_sec += minutes * 60;
}
```

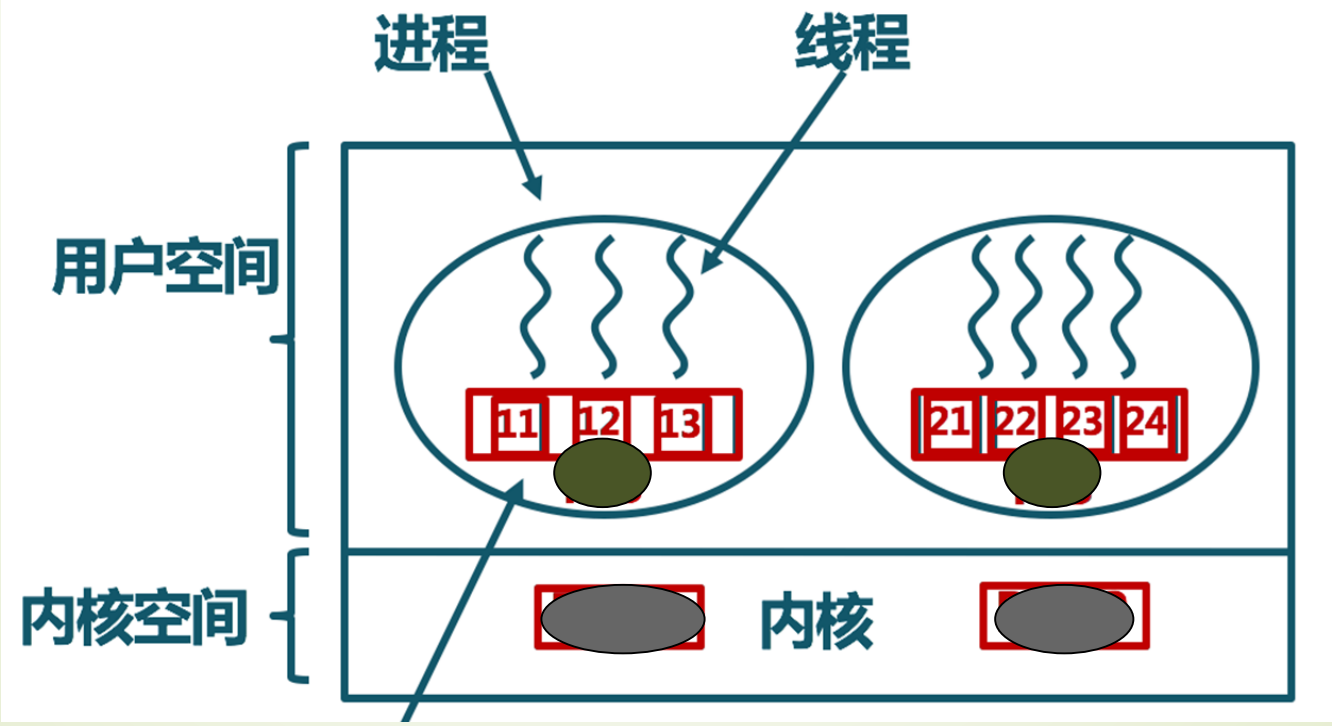
```
define MAX 100    pthread_mutex_t mutex;   pthread_cond_t cond ;
void *thread1(void *);
void *thread2(void *);
int i=1;
int main(void){
    pthread_t t_a;
    pthread_t t_b;
    pthread_mutex_init (&mutex,NULL);  //互斥量的初始化
    pthread_cond_init (&cond,NULL);//条件变量的初始化
    pthread_create(&t_a,NULL,thread2,(void *)NULL);/*创建线程t_a*/
    pthread_create(&t_b,NULL,thread1,(void *)NULL); /*创建线程t_b*/
    pthread_join(t_b, NULL);/*等待线程t_b结束*/
    pthread_join(t_a, NULL);/*等待线程t_a结束*/
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
    exit(0);
} //例8-8
```

```
void *thread1(void *flag){
    for(i=1;i<=MAX;i++)    {
        pthread_mutex_lock(&mutex);/*锁住互斥量*/
        if(i%9==0)    pthread_cond_signal(&cond);/*条件改变，发送信号，通知t_b进程*/
        else  printf("In thread1:%d\n",i);
        pthread_mutex_unlock(&mutex);/*解锁互斥量*/
        sleep(1);
    }
}
void *thread2(void *flag) {
    while(i<MAX)  {
        pthread_mutex_lock(&mutex);
        printf("In thread2 before wait\n");
        if(i%9!=0) pthread_cond_wait(&cond,&mutex);/*等待并释放mutex*/
        printf("In thread2 after wait\n");
        if (i%9==0)   printf("In thread2:%d\n",i);
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
}//例8-8
```

多进程与多线程对比

进程同步与线程同步对比

- 互斥量和条件变量放到进程间共享的内存，就可用于进程间同步



北京航空航天大学 软件学院

多进程与多线程的特性对比

维度	多进程	多线程	总结
数据共享、同步	数据是分开的:共享复杂，需要用 IPC;同步简单	多线程共享进程数据：共享简单；同步复杂	各有优势
内存、CPU	占用内存多，切换复杂，CPU 利用率低	占用内存少，切换简单，CPU 利用率高	线程占优
创建销毁、切换	创建销毁、切换复杂，速度慢	创建销毁、切换简单，速度快	线程占优
编程调试	编程简单，调试简单	编程复杂，调试复杂	进程占优
可靠性	进程间不会相互影响	一个线程挂掉将导致整个进程挂掉	进程占优
分布式	适应于多核、多机分布；如果一台机器不够，扩展到多台机器比较简单	适应于多核分布	进程占优

北京航空航天大学 软件学院

第8章 结束