第14章 同步机制

实验目的

- 深入理解程序并发执行的本质，进程间的互斥和同步概念
- 了解Linux内核并发性和各种同步机制
- 了解信号量实现中涉及的关键技术：进程状态转换和等待队列
- 学会向内核添加新的同步机制

主要内容

- 背景知识
  - 进程同步与同步机制
  - Linux内核并发性和同步机制
- 实验内容
  - 设计并实现新的同步原语

进程同步与同步机制

❖进程互斥和同步的概念

- 操作系统中引入并发程序设计技术，并发执行的进程由于共享系统资源和协同完成任务需要交互，从而产生进程间的相互依赖及相互制约关系。
- 并发进程的交互如果不加以控制，可能出现与时间有关的错误和不正确的运算结果，为了确保系统正常运转，必须提供某种机制来解决并发进程之间的制约关系。

进程同步与同步机制

❖进程互斥和同步的概念

- 并发进程之间的交互存在两种基本关系：
  - ✓竞争关系和协作关系。
- 进程之间的互斥关系，
  - ✓由于计算机中的资源有限，众多进程需要共享资源引起的，当两个进程访问同一个独占型资源时，一个进程向操作系统提出资源申请请求，而另外一个进程只能等待资源被释放后再申请资源。
- 在竞争关系中，由于进程间共享资源而产生制约关系，这是间接制约关系，又称互斥关系。互斥机制是解决进程间竞争关系的手段，进程使用互斥机制向系统提出资源申请，谁先向系统提出申请，谁就能先执行。

进程同步与同步机制

❖进程互斥和同步的概念

- 进程之间的协作关系，
  - ✓同步是并发进程之间共同完成一项任务时直接发生的制约关系，又称协作关系，这些具有协作关系的进程在执行的时间次序上必须遵循确定规律，需要相互协作的进程在某些关键点上协调各自的工作。
  - ✓当其中的一个到达关键点后，在尚未得到其伙伴进程发来的消息或信号之前应阻塞自己，等待协作者发来信号或消息后方被唤醒并继续执行。
- 同步机制是解决进程间同步关系的手段，让并发进程基于某个条件来协调它们的活动，通过合适的方法排定执行的先后次序。

进程同步与同步机制

❖进程互斥和同步的概念

- 例子
  - ✓著名的生产者-消费者问题(producer consumer problem)是计算机操作系统中并发进程内在关系的一种抽象，是典型的进程同步问题。
  - ✓生产者-消费者问题表述如下：有n个生产者和m个消费者，连接在一个有k个单位缓冲区的有界环形缓冲上，故又叫有界缓冲问题。其中，pi和cj都是并发进程，只要缓冲区未满，生产者pi生产的产品就可投入缓冲区；类似地，只要缓冲区不空，消费者进程cj就可从缓冲区取走并消耗产品。

进程同步与同步机制

- 进程互斥和同步的概念信号量和**PV**操作
  - 信号量由荷兰计算机科学家Edsger.Dijkstra在1965年提出。
    - ✓ 并发进程通过信号量展开交互，一个进程在某一关键点上被迫停止执行直到发现信号量值被改变，通过这一设施任何复杂的进程交互要求可得到满足。
    - ✓ 信号量实际上是一个整型数，进程在信号量上的操作仅有两种，
      - 一种称为P操作，另一种称为V操作，
      - 在Linux中，P操作也被称为Down操作，V操作也被称为Up操作。

进程同步与同步机制

❖进程状态

- 进程是一个程序的执行过程，它的动态性质是由其状态变化决定的，进程的状态是会发生变化的，状态及其转换体现了进程的动态性。按进程在执行过程中的不同情况至少要定义3种不同的进程状态:
  - ✓运行（running）态：进程占有处理器正在运行的状态。
  - ✓就绪（ready）态：进程具备运行条件，等待系统分配处理器以便运行的状态。
  - ✓等待（wait）态：又称阻塞(blocked)态或睡眠(sleep)态，指进程不具备运行条件，正在等待某个事件完成的状态。

进程同步与同步机制

❖进程状态 (Linux)

- 可运行态（TASK_RUNNING）：
  - ✓正在运行或准备运行，处于这个状态的所有进程组成可运行队列。
- 可中断睡眠态（TASK_INTERRUPTIBLE）：
  - ✓进程正在睡眠，等待资源可用时被唤醒，也可以由其他进程通过信号或时钟中断唤醒后，进入运行队列。
- 不可中断睡眠态（TASK_UNINTERRUPTIBLE）：
  - ✓与可中断睡眠态类似，但是有一个例外，不可以有其他进程通过信号或时钟中断唤醒。

进程同步与同步机制

- 进程状态 (Linux)
  - 暂停态（TASK_STOPPED）：
    - ✓ 进程暂时停止执行来接受某种状态。例如，当进程接受到SIGSTOP，SIGTSTP，SIGTTIN，SIGTTOU等信号时，进程进入该状态。
  - 僵死态（TASK_ZOMBIE）：
    - ✓ 进程执行结束但尚未消亡的状态。此时，进程已经运行结束并释放大部分资源，但尚未释放进程控制块。这个状态使得其他进程可以在进程消亡之前获得该进程的一些信息。

进程同步与同步机制

❖等待队列

- 内核在管理进程时，需要把处于不同状态的进程进行分类组织。处于可运行态的进程被组织在一起。暂停态和僵死态的进程不链接在专门的链表中。
- 等待态的进程分成很多类，每一类对应一个特定的事件。处于同一类等待态的进程通过等待队列链接在一起。等待队列实现了在事件上的条件等待：希望等待特定事件的进程把自己放进合适的等待队列，并放弃处理器。
- 等待队列表示一组睡眠的进程，当某一条件变成真时，由内核唤醒睡眠的进程。

进程同步与同步机制

❖等待队列

➢数据结构(1)

✓等待队列由循环链表实现，其元素包括指向进程描述符的指针。每一个等待队列都有一个等待队列头（wait queue head），等待队列头是一个类型为wait_queue_head_t的数据结构，内核中定义在<linux/wait.h>文件中：

```
struct __wait_queue_head {
  spinlock_t lock;
  struct list_head task_list;
};
typedef struct __wait_queue_head
 wait_queue_head_t;
```

进程同步与同步机制

❖等待队列

➢数据结构

✓等待队列中的每个元素为wait_queue_t类型，该数据结构也定义在<linux/wait.h>文件中：

```
-struct __wait_queue {
-        unsigned int flags;
-#define WQ_FLAG_EXCLUSIVE 0x01
-  void *private;
-  wait_queue_func_t func;
-  struct list_head task_list;
-};
-typedef struct __wait_queue wait_queue_t;
```

进程同步与同步机制

- 等待队列
  - 数据结构
    - 等待队列中的元素还可以采用DEFINE_WAIT宏进行声明。该宏将private设置为当前进程，将func设置成autoremove_wake_function( )。autoremove_wake_function( )将调用default_wake_function( )，并将该等待队列元素从等待队列中删除。开发者在定义好等待队列元素后，可以使用内核提供的函数直接对等待队列进行操作。

进程同步与同步机制

❖等待队列

- 数据结构
  - add_wait_queue( ): 将一个非互斥进程插入等待队列;
  - add_wait_queue_exclusive( ): 将一个互斥进程插入等待队列;
  - remove_wait_queue( ): 将一个进程从等待队列中删除;
  - waitqueue_active( ): 检查等待队列是否为空。

进程同步与同步机制

- 等待队列
  - 睡眠进程
    - ✓ 内核提供一组函数让进程可以睡眠到某一个事件发生，这些函数使得开发者不需要直接对等待队列进程插入、删除操作。
    - ✓ sleep_on( ): 该函数对当前进程起作用，并需要使用一个等待队列头作为参数。该函数在内核中的代码为:

```
void sleep_on(wait_queue_head_t *wq)
{
        wait_queue_t wait;
        init_waitqueue_entry(&wait, current);
        current->state = TASK_UNINTERRUPTIBLE;
        add_wait_queue(wq,&wait);
        schedule( );
        remove_wait_queue(wq, &wait);
}
```

进程同步与同步机制

❖等待队列

➢睡眠进程

- ✓interruptible_sleep_on( ): 该函数与sleep_on( )类似，但是该函数将进程的状态设置成TASK_INTERRUPTIBLE。处于这个状态的进程除了在等待条件满足时会被唤醒，也会被发送到该进程的信号或时钟中断唤醒。
- ✓sleep_on_timeout( )和interruptible_sleep_on_timeout( ): 这两个函数与sleep_on( )和interruptible_sleep_on( )类似，但它们允许调用者设置一个时间间隔，过了这个时间间隔之后，进程将被内核唤醒。为了做到这一点，它们调用的是schedule_timeout( )而不是schedule( )。

进程同步与同步机制

❖等待队列

➢睡眠进程

✓ **Linux2.6中引入prepare_to_wait( )、prepare_to_wait_exclusive( )和finish_wait( )函数。程序员通过这几个函数也可以使进程睡眠，它们的通常的用法是：**

```
DEFINE_WAIT(wait);
prepare_to_wait_exclusive(&wq, &wait,
TASK_INTERRUPTIBLE);
...
if (!condition)
    schedule( );
finish_wait(&wq, &wait);
```

进程同步与同步机制

- 等待队列
  - 睡眠进程
    - ✓ 从Linux2.4开始，wait_event和wait_event_interruptible宏被引入。这两个宏将进程置于等待队列中睡眠，直到给定的条件满足。wait_event_interruptible(wq,condition)实际产生的代码与如下类似：

```
DEFINE_WAIT(_ _wait);
for (;;) {
    prepare_to_wait(&wq, &_ _wait,
TASK_UNINTERRUPTIBLE);
    if (condition)
        break;
    schedule( );
}
finish_wait(&wq, &_ _wait);
```

进程同步与同步机制

❖等待队列

- 唤醒进程
  - ✓ 内核通过wake_up( ), wake_up_nr( ), wake_up_all( ), wake_up_sync( ), wake_up_sync_nr(), wake_up_interruptible( ), wake_up_interruptible_nr( ), wake_up_interruptible_all( ), wake_up_interruptible_sync( )和 wake_up_interruptible_sync_nr( )来唤醒进程。可以从其名字知道每个宏的含义:
  - ✓ 所有宏都考虑到处于TASK_INTERRUPTIBLE状态的睡眠进程; 如果宏的名字中不含字符串“interruptible”, 则还将考虑处于TASK_UNITERRUPTIBLE状态的睡眠进程。

进程同步与同步机制

❖等待队列

- 唤醒进程
  - ✓所有宏都考虑到处于TASK_INTERRUPTIBLE状态的睡眠进程；如果宏的名字中不含字符串“interruptible”，则还将考虑处于TASK_UNITERRUPTIBLE状态的睡眠进程。
  - ✓所有宏唤醒具有所需状态的所有非互斥进程。
  - ✓名字中含有“nr”字符串的宏唤醒给定数字的具有所需状态的互斥进程；这个数字是宏的一个参数。名字中含有“all”字符串的宏唤醒具有所需状态的所有互斥进程。最后，名字中不含“nr”或“all”字符串的宏只唤醒具有所需状态一个互斥进程。
  - ✓名字中不含有“sync”字符串的宏检查唤醒进程的优先级是否高于系统中正在运行进程的优先级，并在必要时调用schedule( )。

进程同步与同步机制

- 等待队列
  - 唤醒进程
    - ✓ 例如，wake_up宏等价于下面的代码片断：

```
– void wake_up(wait_queue_head_t *q)
– {
–     struct list_head *tmp;
–     wait_queue_t *curr;
–     list_for_each(tmp, &q->task_list) {
–         curr = list_entry(tmp, wait_queue_t, task_list);
–         if (curr->func(curr,
    TASK_INTERRUPTIBLE|TASK_UNINTERRUPTIBLE,
–                 0, NULL) && curr->flags)
–             break;
–     }
– }
```

主要内容

- 背景知识
  - 进程同步与同步机制
  - **Linux内核并发性和同步机制**
- 实验内容
  - 设计并实现新的同步原语

Linux内核并发性和同步机制

- 中断可能在任何时刻异步发生，随时可能打断正在执行的内核代码;
- 内核调度tasklet和softirq，打断当前正在执行的代码;
- 如果内核具有抢占性，运行的内核任务会被另一个任务抢占;
- 进程被阻塞时，会唤醒调度程序工作，引起其他进程运行;
- SMP平台上，两个或多个CPU同时执行内核代码，可能访问同一共享数据结构。

Linux内核并发性和同步机制

- 原子操作
- 内核信号量
- 关中断
- 自旋锁

Linux内核并发性和同步机制

❖原子操作

- 原子整数操作。
  - ✓ 针对整型变量，定义特殊数据类型atomic_t和专门的原子整型操作函数，典型用途是实现计数器。以下是部分原子整型操作：
  - ✓ ATOMIC_INT(int i)(初始化原子变量为i)、
  - ✓ atomic_read(atomic_t *v)(读整数值v)、
  - ✓ atomic_set(atomic_t *v,int i)(把v置成i)、
  - ✓ atomic_add(int i, atomic_t *v)(把v增加i)、
  - ✓ atomic_sub(int i, atomic_t *v)(把v减去i)、
  - ✓ atomic_sub_and_test(int i, atomic_t *v)(从v中减去i，结果为0时，则返回1，否则返回0)、
  - ✓ atomic_add_negative(int i, atomic_t *v)(将v中增加i，结果为0时，则返回1，否则返回0)、
  - ✓ atomic_inc(atomic_t *v)(将v加1)和
  - ✓ atomic_dec(atomic_t *v)(将v减1)等。

Linux内核并发性和同步机制

❖原子操作

- 原子位图操作。
  - ✓针对指针变量指定的任意一块主存区域的位序列进行操作。以下是部分原子位图操作：set_bit(int nr,void *addr)(设置位图地址addr的nr位)、clear_bit(int nr,void *addr)(清除位图地址addr的nr位)、change_bit(int nr,void *addr)(反转位图地址addr的nr位)和test_bit(int nr,void *addr)(返回位图地址addr的nr位的值)等。

Linux内核并发性和同步机制

- 内核信号量
  - 在Linux内核中，也使用等待队列来实现信号量机制，内核信号量semaphore定义为：

```
struct semaphore {
        atomic_t count;
        int sleepers;
        wait_queue_head_t wait;
};
```

Linux内核并发性和同步机制

❖内核信号量

- 资源计数器count表示可用的某种资源数，若为正整数则尚有这些资源可用；若为0或负整数则资源已用完，且因申请资源而等待的进程有count绝对值个。
- sema_init宏用于初始化count为任何值，可以是二元信号量，也可以是一般信号量;
- 在DOWN（）操作中，count减1后的值若小于0，进程便进入等待队列。
- UP（）操作中，count加1后的值如果大于0，立即唤醒等待队列中的所有进程。

Linux内核并发性和同步机制

❖内核信号量

- sleepers对等待当前临界资源的进程个数进行辅助计数。
- wait用于存放等待队列链表的地址，该链表包含当前正在等待该信号量(资源)而被阻塞的所有进程。
- 内核信号量上定义的函数有：DOWN（）、DOWN_interruptible（）、DOWN_trylock（）和UP（）。此外，还提供读者-写者信号量，相应的操作有：down_read（）读者down操作、up_read（）读者up操作、down_write（）写者down操作及up_write（）写者up操作。

Linux内核并发性和同步机制

❖内核信号量

- Linux内核信号量采用非忙式等待实现，当进程执行DOWN而等待时，将放入等待信号量队列；事实上，并发进程同步时，只要等待条件不满足，就必须挂起，放入相应等待队列。所以，等待队列是支持进程同步的重要数据结构，其定义为：
  - ✓ struct wait_queue {
  - ✓ unsigned int compiler_warning;
  - ✓ struct task_struct *task; /*指向等待进程的PCB*/
  - ✓ struct list_head task_list; /*等待队列链表*/
  - ✓ };

Linux内核并发性和同步机制

❖关中断

- 一个正在对内核数据结构操作的进程可以被I/O设备中断，如果设备中断处理程序恰好也要访问该数据结构，就会引起系统的不一致状态，解决这类问题的有效方法是关中断。关中断是把内核态执行的程序段作为一个临界区来保护的一种手段，主要保护中断处理程序也要访问的数据结构，如磁盘中断就需要被屏蔽，此外，关中断还能禁止内核抢占。为了防止死锁，关中断期间内核不能执行阻塞操作。在SMP环境中，关中断只能防止来自本机其他中断处理程序的并发访问，需要引入自旋锁在禁止本地中断的同时，防止来自它机的并发访问。

Linux内核并发性和同步机制

❖ 自旋锁

- Linux内核中最常见的锁是自旋锁，自旋锁最多只能被一个可执行线程持有，如果一个执行线程试图获得一个已经被锁住的自旋锁，那么该线程就会一直进行忙式等待(旋转)，等待锁重新可用，期间这个CPU不能再处理其他工作，同时等待其他CPU上运行的进程执行解锁操作，要是锁未被争用，请求锁的执行线程便能立刻锁住它，继续执行。在任意时刻，自旋锁都可以防止多于一个的执行线程同时进入临界区。

Linux内核并发性和同步机制

❖ 自旋锁

- 自旋锁是最简单的一种锁原语，锁的取值为0表示资源可用，锁的取值为1表示资源加锁。自旋锁很象二元信号量，但在实现上有区别，若一个资源得到自旋锁的保护，另一个试图取得该资源的内核例程将保持忙式等待，直到资源被解锁。而在实现二元信号量时，不必循环等待信号量，而是进入等待队列，暂时放弃对资源的请求。

Linux内核并发性和同步机制

❖ 自旋锁

- Linux中，自旋锁变量lock被定义成spinlock_t类型，只有lock成员的最低位被使用，如果锁可用，lock的最低位为0；如果上锁lock的最低位为1，初始化时，lock成员被设为0、即未上锁。自旋锁定义为：
  - ✓ typedef struct {
  - ✓ volatile unsigned int lock;
  - ✓ }spinlock_t;

Linux内核并发性和同步机制

- 自旋锁
  - ✓ void spin_unlock(spinlock_t *plock)
  - ✓ {
  - ✓ plock->lock&=~1;　　/*开锁,将lock的第0个bit置为0*/
  - ✓ }
- 在SMP系统中，自旋锁最重要的特点是内核例程在等待锁被释放时一直占据着某个CPU，一般用来保护那些只需简短访问数据结构的操作，如从双向队列链表增或删除一个元素，因此，在内核需要进程同步的位置，自旋锁用得十分频繁。

Linux内核并发性和同步机制

- 自旋锁
  - 自旋锁的上锁或解锁通过以下宏实现:
    spin_lock(spinlock_t *lock)/spin_unlock(spinlock_t *lock)(获得/释放自旋锁)、spin_lock_irq(spinlock_t *lock)/spin_unlock_irq(spinlock_t *lock)(获得自旋锁并关闭中断/释放自旋锁并开放中断)、spin_lock_bh(spinlock_t *lock)/spin_unlock_bh(spinlock_t *lock)(获得自旋锁并关闭bh的执行/释放自旋锁并开放bh的执行)和spin_lock_init(spinlock_t *lock)(初始化自旋锁)等。

Linux内核并发性和同步机制

- 自旋锁
  - void spin_lock(spinlock_t *plock)
  - {
  - int flag;
  - do{
  - flag=plock->lock&1; /*取出lock的第0个bit，若为1，已上锁*/
  - plock->lock=1; /*上锁，将lock的第0个bit置为1*/
  - }while(flag!=0); /*若已上锁则循环测试，直到成功*/
  - }

主要内容

- 背景知识
  - 进程同步与同步机制
  - Linux内核并发性和同步机制
- 实验内容
  - 设计并实现新的同步原语

设计并实现新的同步原语

❖ 实验说明

- 设计一组新的同步原语，多个进程可以在同一个事件上等待。当其他进程产生该事件时，等待在该事件上的进程被唤醒。如果事件产生时，没有进程因这个事件而阻塞，则该事件无效。内核需要为该同步原语提供4个系统调用：
  - ✓1）int event_open(int arg)
    - 打开一个事件。如果参数arg等于0，内核创建一个新事件，并且返回新事件的ID；如果参数arg大于0，内核打开一个已有事件ID(为arg的事件)，如果该事件不存在，内核返回-1。
  - ✓2）int event_close(int)
    - 关闭已有事件。关闭成功返回0，关闭失败返回-1。如果在关闭时还有进程等待在该事件上，内核将这些进程唤醒。

设计并实现新的同步原语

- 实验说明
  - ✓3）int event_wait(int arg)
    - 进程在事件arg上等待，直到其他进程产生该事件。如果事件不存在，内核返回-1。
  - ✓4）int event_sig(int arg)
    - 进程产生事件arg，等待在该事件上的进程被唤醒。如果事件不存在，内核返回-1。

设计并实现新的同步原语

- 解决方案
  - 数据结构
    - ✓ 用户在使用该组同步原语时，可以动态地创建多个事件。用内核提供的双向链表将这些数据结构组织在一起是比较好的方法。对于链表中的每一个事件，内核首先需要为事件定义一个事件ID；由于进程可以等待在事件上，因此每个事件都需要有一个单独的等待队列，睡眠于该事件的进程便可以放入进程的该等待队列中。在事件数据结构中，还需要记录该事件是否已经发生。根据这些要求，可以将事件的数据结构定义为：

```
typedef struct __event{
        int id;
        wait_queue_head_t *wq;
        bool occur;
        struct list_head events;
} event_t;
```

设计并实现新的同步原语

- 解决方案
  - 数据结构
    - ✓ id表示事件ID；wq为等待队列头；occur表示事件是否已经发生过；events是用于将事件链在一起的双向链表元素。同时，内核需要记录事件链表的头元素，这样内核才能对链表进行搜索。事件链表的头元素定义为内核中的一个全局变量：
      - event_t *event_head = {0, NULL, 0, LIST_HEAD_INIT(event_head.events)};
    - ✓ Linux双向链表的使用可以参考本书前面的相关内容。当定义好链表头后，事件便可以插入链表中。为了加快事件的搜索速度，事件在插入链表时，按照事件ID从小到大进行排序。

设计并实现新的同步原语

- 解决方案
  - event_open( )
    - ✓ event_open系统调用是为了创建或者打开已有的一个事件。该系统调用的主要流程如图14-1所示。

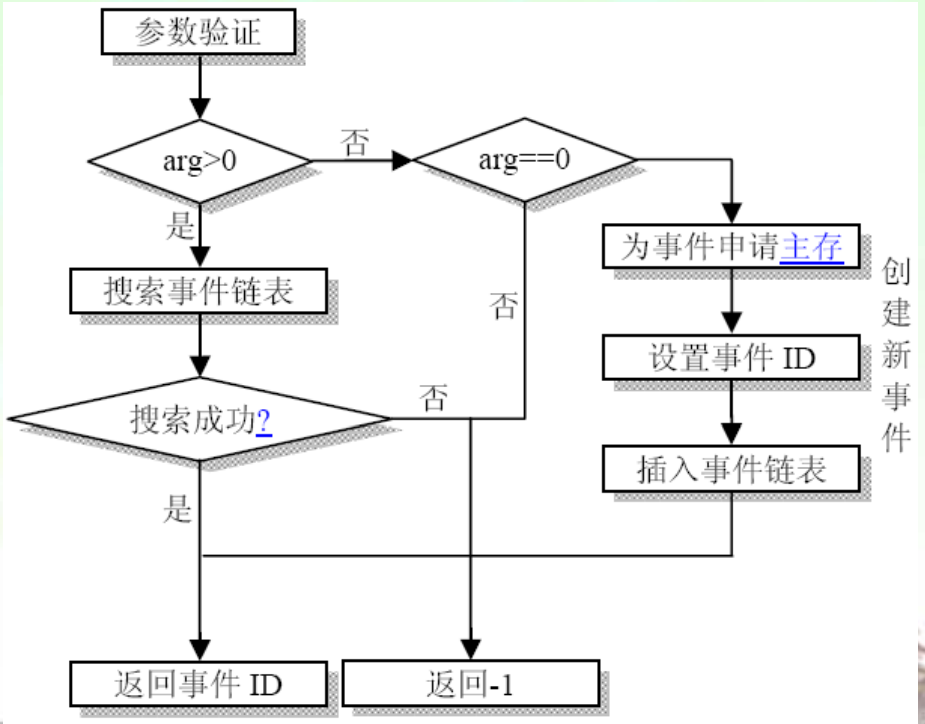

图 14-1 event_open 流程图

设计并实现新的同步原语

- 解决方案
  - 搜索事件
    - ✓ 在event_open( )中，根据用户提供的事件ID，内核需要在事件链表中搜索到相应的事件元素。由于链表采用的是Linux内核提供的双向链表，并且链表中元素按事件ID排序，搜索事件可以采用以下的代码：

```
event_t *search_event( int id )
{
        event_t *event = NULL;
        list_head *p = NULL;
        list_for_each(p, event_head) {
                event = list_entry(p,event_t, events);
                if ( event->id==id )  /*找到元素*/
                        return event;
                if ( event->id>id )   /*链表有序排列，因此不需要再继续查找*/
                        return NULL;
        }
        return NULL;
}
```

设计并实现新的同步原语

- 解决方案
  - 创建新事件
    - ✓ 创建新事件主要涉及到3个流程：为事件申请主存、为事件申请ID和将事件插入事件链表。
    - ✓ 在内核中，申请主存可以采用kmalloc( )，该内核函数的原型为：
    - ✓ void *kmalloc(size_t size, int flags);
    - ✓ size为所申请的主存字节数，flags为要分配的主存类型，返回值为申请到的主存地址。flags参数取值如下：GFP_USER代表用户分配主存，可以睡眠；GFP_KERNEL分配内核中的主存，可以睡眠；GFP_ATOMIC表示分配但不睡眠，一般在中断处理程序中使用。为事件申请主存可以使用如下代码：

```
newevent = (event_t *) kmalloc(sizeof(event_t),
GFP_KERNEL);
newevent->wq = (wait_queue_head_t *)
kmalloc(sizeof(wait_queue_head_t),
   GFP_KERNEL);
init_waitqueue_head(newevent_wq);
```

设计并实现新的同步原语

- 解决方案
  - event_wait( )
    - 当事件创建好以后，进程可以通过event_wait( )系统调用在事件上等待。Linux提供wait_event宏让进程可以在指定的等待队列中等待。该宏的使用方法是：
      - wait_event(wq, condition);
    - wq是使用到的等待队列，condition是一个布尔值的表达式。进程调用这个宏后，会在指定的等待队列中睡眠，直到指定的条件变为真。在事件的数据结构中，occur成员表明事件是否已经发生，因此该成员便是wait_event需要使用到的条件。
      event_wait( )的主要结构如下：

```
int event_wait( int id)
{
        ……
        event_t *e = search_event(id);
        wait_event(e->wq, e->occur);
        ……
}
```

设计并实现新的同步原语

❖解决方案

➢ event_sig( )

- ✓ event_sig( )系统调用的作用是唤醒在指定事件上睡眠的进程，它的主要工作流程是：
  - 1)设置事件的occur成员，表明事件已经发生；
  - 2)唤醒睡眠进程。
- ✓ 与wait_event宏类似，内核提供wake_up(wq)宏唤醒在指定等待队列中的睡眠进程，wq是指定的等待队列。event_sig( )的结构如下：

```
int event_sig(int id)
{
        ……
        event_t *e = search_event(id);
        e->occur = true;
        wake_up(e->wq);
        ……
}
```

设计并实现新的同步原语

- 解决方案
  - event_close( )
    - ✓ event_close( )系统调用需要将事件删除，并且在事件被删除前将睡眠在该事件上的进程唤醒。删除事件的主要流程是：
      - 1) 唤醒睡眠在该事件的进程;
      - 2) 将事件从事件链表中删除;
      - 3) 将事件所占用的主存释放。
    - ✓ 该流程涉及的几个内核函数有：list_del( )，kfree( )。list_del( )将指定的双向链表节点从链表中删除，kfree( )用于释放用kmalloc( )申请的主存。这两个内核函数的原型是：
      - void list_del(struct list_head *entry);
      - void kfree(const void *objp);
    - ✓ 在list_del( )内核函数中，参数entry是要删除的双向链表节点；在kfree( )内核函数中，参数objp是需要释放的主存地址。

设计并实现新的同步原语

- 解决方案
  - 封装系统调用
    - ✓ 最后一步是将这些系统调用封装成函数，读者可以参考系统调用一章的内容。

设计并实现新的同步原语

- 解决方案
  - 使用新的同步原语
    - ✓ 在定义新的系统调用及封装成函数以后，程序中就可以使用它们了。在本节中，将通过编写几个测试程序来验证。程序用SYS_eventopen、SYS_eventclose、SYS_eventwait和SYS_eventsig 4个宏表示这几个系统调用的系统调用号。测试程序包括4个文件：
      /*****open.c、close.c、sig.c、wait.c*****/

设计并实现新的同步原语

- 解决方案
  - 使用新的同步原语

```
/*open.c*/
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv)
{
        if ( argc!=2 )
                return -1;
        int id = syscall(SYS_eventopen, aroi(argv[1]);
        printf("%d\n", id);
        return 0;
}
```

设计并实现新的同步原语

- 解决方案
  - 使用新的同步原语

```
/*close.c*/
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv)
{
        if ( argc!=2 )
                return -1;
        int r = syscall(SYS_eventclose, aroi(argv[1]);
printf("%d\n", id);
        return 0;
}
```

设计并实现新的同步原语

- 解决方案
  - 使用新的同步原语

```
/*wait.c*/
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
        if ( argc!=2 )
                return -1;
        i = syscall(SYS_eventwait, atoi(argv[1]));
        printf("%d\n",i);
        return 0;
}
```

设计并实现新的同步原语

- 解决方案
  - 使用新的同步原语

```
/*sig.c*/
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
        if ( argc!=2 )
                return -1;
        i = syscall(SYS_eventsig, atoi(argv[1]));
        printf("%d\n",i);
        return 0;
}
```

设计并实现新的同步原语

- 解决方案
  - 使用新的同步原语
    - ✓ 编译源代码后获得可执行程序open、close、wait和sig。open程序用于打开一个指定的事件，如果事件ID为0，则创建一个新事件，并将事件ID打印到控制台，例如：
      - #open 0
      - 1
    - ✓ 这表明open程序创建了事件ID为1的新事件。当一个事件创建好以后，可以通过wait命令在该事件上等待：
      - #wait 1

设计并实现新的同步原语

- 解决方案
  - 使用新的同步原语
    - ✓ 执行该命令后，进程将在该事件上等待，并进入睡眠状态。如果要将该进程唤醒，需要切换到其他控制台，并通过sig程序唤醒该进程。
      - #sig 1
    - ✓ 执行该命令之后，等待事件1上的进程将被唤醒。在实验的最后，用户需要通过close程序将已经打开的事件关闭，将占用的资源释放。
      - #close 1

第14章 同步机制