Operating Systems

Chapter 5 Mutual Exclusion(互斥) and Synchronization(同步)

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- **5.3 Semaphores**
- **5.4 Monitors**
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

Review section1&2

- Concurrency and Problems
  - Race condition/deadlock/livelock/starvation

- Solutions : Mutual Exclusion
  - 概念：
    - Critical section 临界区 , Atomic Operation 原子操作
    - thdcas_demo
  - 实现
    - 1 、 Hardware approaches
      - Interrupt Disabling 、 Special Machine Instructions
    - 2 、 Software approaches
      - Dekker ， Peterson

Review section1&2

- Lab06 Part I:
  - 阅读实验指导： 4.1 Linux 线程 - 互斥锁 pthread_mutex_t

线程互斥锁 pthread_mutex_t 的实现原理：

```
pthread_mutex_lock:
atomic_dec(pthread_mutex_t.value);
if(pthread_mutex_t.value!=0)
futex(WAIT)
else
success

pthread_mutex_unlock:
atomic_inc(pthread_mutex_t.value);
if(pthread_mutex_t.value!=1)
futex(WAKEUP)
else
success
```

Review section1&2

- Lab06 Part I:
  - 先完成实验指导： 4.1 Linux 线程 - 互斥锁 pthread_mutex_t
  - 体会生产数据和使用数据间的配合：
    - 互斥满足
    - 但数据同步，不满足
- 问题：如何既互斥又同步

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

Agenda

- 5.3 Semaphores
- 5.4 Monitors

5.3 Semaphores

信号量的定义和实现

- 5.3.1 Overview
- 5.3.4 Implement of Semaphores

信号量的应用

- 5.3.2 Mutual Exclusion 互斥
- 5.3.3 The Producer/Consumer problem 同步

5.3.1 Semaphores(信号量) Overview(1/3)

- Fundamental principle(基本原理):
  - Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal.(两个或者多个进程可以通过简单的信号进行合作，一个进程可以被迫在一个位置停止，直到它收到一个信号)

5.3.1 Semaphores(信号量) Overview(2/3)

- For signaling, special variables called **semaphores** are used(一种称为 *信号量*的特殊变量用来传递信号)

- If a process is **waiting** for a signal, it is **blocked** until that signal is **sent** (如果一个进程在等待一个信号，它会被阻塞，直到它等待的信号被发出)

5.3.1 Semaphores(信号量) Overview(3/3)

- Semaphore : **sem 变量 + 2 atomic operations 原子操作**
  - **Sem**(integer value 整数值)
    - initialize to a nonnegative number (非负数)?
    - **初始值问题： 0,1,…n ？**

①semWait (P) ：

①--sem

②If sem < 0, the process executing the semWait is blocked.

③else , process continue execute

②semSignal (V) ：

①++sem

②If sem<=0, then a process blocked by a semWait operation is unblocked.

③else , process continue execute

5.3 Semaphores

信号量的定义和实现

- 5.3.1 Overview
- 5.3.4 Implement of Semaphores

信号量的应用

- 5.3.2 Mutual Exclusion 互斥
- 5.3.3 The Producer/Consumer problem 同步

5.3.4 Implement of Semaphores(1/6)

Semaphore Primitives(原语被假定为原子操作)

```
struct semaphore {
      int count;
      queueType queue;
}

void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0)
      {
            place this process in s.queue;
            block this process
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0)
      {
            remove a process P from s.queue;
            place process P on ready list;
      }
}
```

Figure 5.3 A Definition of Semaphore Primitives

5.3.4 Implement of Semaphores(2/6)

- Semaphore operations are atomic operations.
  - 时间非常短
  - Implement in hardware or firmware(固件)
  - Implement in software, e.g. Dekker or Peterson
  - Implement by inhibit interrupts(中断禁止) for a single-processor system

5.3.4 Implement of Semaphores(3/6)

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set
s.flag to 0) */;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

(a) Compare and Swap Instruction

```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow inter-
rupts */;
   }
      allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

(b) Interrupts

5.3.4 Implement of Semaphores(4/6)

- OpenEuler 信号量的实现


```
1.    //源代码：kernel/include/linux/semaphore.h
2.    struct semaphore {
3.        raw_spinlock_t lock;
4.        unsigned int count;
5.        struct list_head wait_list;
6.    };
```

图 6-40　信号量的数据结构

```
1.    //源代码：kernel/locking/semaphore.c
2.    int down_interruptible(struct semaphore * sem) {
3.        unsigned long flags;
4.        int result = 0;
5.        raw_spin_lock_irqsave(&sem->lock, flags);
6.        if (likely(sem->count > 0))
7.            sem->count--;
8.        else
9.            result = __down_interruptible(sem);      //对等待线程的操作
10.       raw_spin_unlock_irqrestore(&sem->lock, flags);
11.       return result;
12.   }
```

图 6-41　down 原语的实现

5.3.4 Implement of Semaphores(5/6)

- OpenEuler 信号量的实现


```
1.  //源代码：kernel/locking/semaphore.c
2.  static inline int __sched __down_common(struct semaphore * sem,
3.                                          long state,long timeout){
4.      struct semaphore_waiter waiter;                    //描述被推入等待队列的线程
5.      //将当前线程加入等待队列
6.      list_add_tail(&waiter.list, &sem->wait_list);
7.      waiter.task = current;
8.      waiter.up = false;                                 //将唤醒标志设置为 false
9.      for (;;) {
10.     ...
11.         //将线程状态设置为 TASK_UNINTERRUPTIBLE
12.         __set_current_state(state);
13.         raw_spin_unlock_irq(&sem->lock);               //释放锁
14.         timeout = schedule_timeout(timeout);           //主动让出 CPU
15.         raw_spin_lock_irq(&sem->lock);                 //上锁
16.         if (waiter.up)                                 //线程被唤醒
17.             return 0;
18.     }
19.     ...
20. }
```

图 6-42　阻塞队列的关键代码

5.3.4 Implement of Semaphores(6/6)

- OpenEuler 信号量的实现


```
1.    //源代码：kernel/locking/semaphore.c
2.    void up(struct semaphore * sem) {
3.        unsigned long flags;
4.        raw_spin_lock_irqsave(&sem->lock, flags);
5.        if (likely(list_empty(&sem->wait_list)))
6.            sem->count++;          //空闲资源数目加 1
7.        else
8.            __up(sem);             //唤醒线程
9.        raw_spin_unlock_irqrestore(&sem->lock, flags);
10.   }
```

图 6-43　up 原语的实现

```
1.    //源代码：kernel/locking/semaphore.c
2.    static noinline void __sched __up(struct semaphore * sem) {
3.        struct semaphore_waiter * waiter = list_first_entry(&sem->wait_list,
4.                                            struct semaphore_waiter, list);
5.        list_del(&waiter->list);          //将阻塞线程移出等待队列
6.        waiter->up = true;                //设置唤醒标志为 true
7.        wake_up_process(waiter->task); //修改线程状态,并加入就绪队列中
8.    }
```

图 6-44　__sched __up()的实现

5.3 Semaphores

信号量的定义和实现

- 5.3.1 Overview
- 5.3.4 Implement of Semaphores

信号量的应用

- 5.3.2 Mutual Exclusion 互斥
- 5.3.3 The Producer/Consumer problem 同步

5.3.2 Mutual Exclusion(1/9)

- 用 Semaphore 完成上一节练习的互斥

```
int a=1,b=1;
void* P1(void *arg)
{
        int i=0;
        while(i<1e2){
            i++;
            a=a+1;
                b=b+1;
        }
}
```

```
void* P2(void *arg)
{
        int i=0;
        while(i<1e2){
                i++;
                    b=2*b;
                    a=2*a;
        }
}
```

5.3.2 Mutual Exclusion(2/9)

- Semaphore application
  - Solution: Semaphore for Mutual Exclusion

```
int a=1,b=1;
semaphore sem =?
void* P1(void *arg)
{
        int i=0;
        while(i<1e2){
            i++;
            semWait(sem);
            a=a+1;
                b=b+1;
            semSignal(sem);
 }}
```

```
void* P2(void *arg)
{
        int i=0;
        while(i<1e2){
            i++;
            semWait(sem);
                b=2*b;
                a=2*a;
            semSignal(sem);
    }}
```

5.3.2 Mutual Exclusion(3/9)

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true)
     {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores

5.3.2 Mutual Exclusion(4/9)

小结 1

1. **信号量互斥框架如上图**
2. **当两个线程竞争同一资源（互斥区）时，信号量初始值为 1**
   - Race condition: Winner enters and continues to execute
   - 信号量值只能为 0 或 1 ，用 Binary Semaphore(二元信号量) 即可

5.3.2 Mutual Exclusion(5/9)

- Binary Semaphore & Mutexlock
- In common:
  - Mutual Exclusion

- **Difference:**
  - MutexLock: lock and unlock operations can only be called by the same thread or process
  - Semaphore: semWait and semSignal operations can be called by different process, like producer_thread and consumer_thread

5.3.2 Mutual Exclusion(7/9)

Counting Semaphore 计数信号量 : sem= 1 ; 3 threads

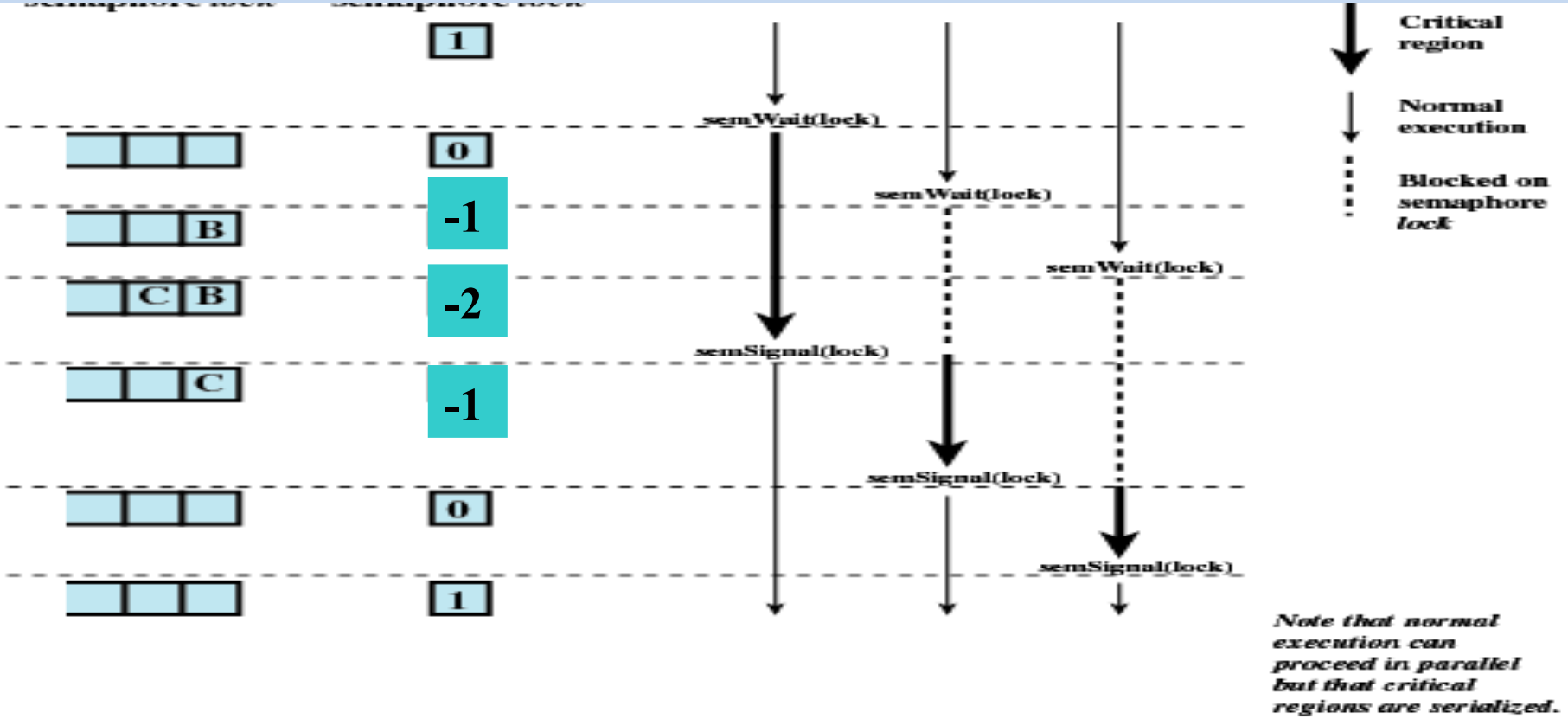


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

5.3.2 Mutual Exclusion(8/9)

小结 2 ：

当有 k(k >2) 个线程竞争同一资源（临界区）时：

- 使用 Counting Semaphore 计算信号量
- **信号量初始值为 1**

When sem is –m, m indicates the number of threads waiting to enter 为 -m 表示已经有 m 个线程在等待进入临界区 (m<k)

5.3.2 Mutual Exclusion(9/9)

- Semaphore terms (信号量术语)
  - Mutex [lock] (互斥信号量，锁)
  - Binary Semaphore (二元信号量): sem(0,1)
  - Counting Semaphore (计数信号量)/General Semaphore(一般信号量): sem >=0
  - Strong Semaphore(强信号量): 等待线程的调度遵循 FIFO
  - Weak Semaphore(弱信号量): policy undefied

5.3 Semaphores

信号量的定义和实现

- 5.3.1 Overview
- 5.3.4 Implement of Semaphores

信号量的应用

- 5.3.2 Mutual Exclusion 互斥
- 5.3.3 The Producer/Consumer problem 同步

5.3.3 Producer/Consumer Problem(1/14)

- Policy:
  - When buffer is empty, can not consume 同步
  - When buffer is full, can not produce (finite, not infinite) 同步
  - **Only one** producer or consumer may access the buffer **at any one time (mutual exclusion** 互斥 **)**
- Analyze
  - Two messages 即 2 个信号量 : full, empty
    - 如何初始化？
  - One mutex lock

5.3.3 Producer/Consumer Problem(2/14)

- One producer, one consumer , **buffer size** = 1

```
Semaphore  empty=1,full=0
void* producer(void *arg)
{
        while(1){
            semWait(empty)
            produce();
            semSignal(full)
        }}
```

```
void* consumer(void *arg)
{
        while(1){
            semWait(full)
            remove();
            semSignal(empty)
        } }
```

Threads switch?

Fine. mutual exclusion and synchronization

5.3.3 Producer/Consumer Problem(3/14)

- One producer, one consumer , **buffer size** = 1

```
pthread_mutex_t mutex;
void* producer(void *arg)
{
        while(1){
    pthread_mutex_lock(&mutex);
        produce();
    pthread_mutex_unlock(&mutex);
    }}
```

```
void* consumer(void *arg)
{
        while(1){
    pthread_mutex_lock(&mutex);
        remove();
    pthread_mutex_unlock(&mutex);
    } }
```

Unable to work effectively

5.3.3 Producer/Consumer Problem(4/14)

Finite buffer

生产
是

可否去除互斥锁？

- **int buffer[bufsize]={0};**
- **sem empty=bufsize,full=0,mutex=1;**

```
void* producer(void *arg)
{
        while(1){
          semWait(empty)
          semWait(mutex)
          produce();
          semSignal(mutex)
          semSignal(full)
}}
```

```
void* consumer(void *arg)
{
        while(1){
          semWait(full)
          semWait(mutex)
          remove();
          semSignal(mutex)
          semSignal(empty)
}}
```

5.3.3 Producer/Consumer Problem(5/14)

- Exchange the position of 2 semWaits or 2 semSignals?
  - semSignals won't block any threads, it's OK to exchange
  - But semWaits may block threads(dead lock). Don't exchange.
  - 可以交换 signal 的位置，不能交换 wait(死锁)

```
void* producer(void *arg)
{
        while(1){
          semWait(empty)
          semWait(mutex)
        produce();
          semSignal(mutex)
          semSignal(full)
}}
```

```
void* consumer(void *arg)
{
        while(1){
          semWait(mutex)
          semWait(full)
          remove();
          semSignal(mutex)
          semSignal(empty)
}}
```

5.3.3 Producer/Consumer Problem(6/14)

inFinite buffer

- **//int buffer[inf]={0};**
- **sem full=0,mutex=1;**

```
void* producer(void *arg)
{
        while(1){
          //semWait(empty)
          semWait(mutex)
          produce();
          semSignal(mutex)
          semSignal(full)
 }}
```

```
void* consumer(void *arg)
{
        while(1){
          semWait(full)
          semWait(mutex)
          remove();
          semSignal(mutex)
          //semSignal(empty)
 } }
```

5.3.3 Producer/Consumer Problem(7/14)

- **Mutual Exclusion and synchronize 同步与互斥**
  - Produce before consume ： 同步
- The buffer may be finite or infinite
- Producer & consumer ： empty sem>1 互斥
  - One producer, one consumer (已生产 1 个，生产第二个和消费需要互斥)
  - One or more producers are generating data and placing these in a buffer ； A single consumer is taking items out of the buffer, one at time.

5.3.3 Producer/Consumer Problem(8/14)

总结

- One sem releases one information
- Semaphore initialized to n:sem = n
  - n is the number of processes permit to enter; in another words: n processes is allowerd in its critical section at a time(多个进程同时在临界区内)
    - n=0, 同步问题 (消费者等待生产者)
    - n=1 ，互斥问题 (-m 表示 m 个线程在等待)
      - 二元信号量 ， 2 个线程互斥
      - 计数信号量 ， -K 个线程在等待
    - n>1, 互斥 (资源数为 n ，多个生产者、哲学家就餐，多个读者问题等)

5.3.3 Producer/Consumer Problem(9/14)

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
     while (true)
     {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1)
               semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true)
     {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0)
               semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

缓冲区中的项数

控制进入临界区

避免“超前”消费

Figure 5.9 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

5.3.3 Producer/Consumer Problem(10/14)

Table 5.4 Possible Scenario for the Program of Figure 5.9

	Producer	Consumer	s	n	Delay
1			1	0	0
2	`semWaitB(s)`		0	0	0
3	`n++`		0	1	0
4	`if (n==1) (semSignalB(delay))`		0	1	1
5	`semSignalB(s)`		1	1	1
6		`semWaitB(delay)`	1	1	0
7		`semWaitB(s)`	0	1	0
8		`n--`	0	0	0
9		`semSignalB(s)`	1	0	0
10	`semWaitB(s)`		0	0	0
11	`n++`		0	1	0
12	`if (n==1) (semSignalB(delay))`		0	1	1
13	`semSignalB(s)`		1	1	1
14		`if (n==0) (semWaitB(delay))`	1	1	1
15		`semWaitB(s)`	0	1	1
16		`n--`	0	0	1
17		`semSignalB(s)`	1	0	1
18		`if (n==0) (semWaitB(delay))`	1	0	0
19		`semWaitB(s)`	0	0	0
20		`n--`	0		0
21		`semSignalB(s)`	1	–1	0

5.3.3 Producer/Consumer Problem(11/14)

```
/* program producerconsumer */
      int n;
      binary_semaphore s = 1, delay = 0;
      void producer()
      {
            while (true)  {
                  produce();
                  semWaitB(s);
                  append();
                  n++;
                  if (n==1) semSignalB(delay);
                  semSignalB(s);
            }
      }
      void consumer()
      {
            int m; /* a local variable */
            semWaitB(delay);
            while (true)   {
                  semWaitB(s);
                  take();
                  n--;
                  m = n;
                  semSignalB(s);
                  consume();
                  if (m==0) semWaitB(delay);
            }
      }
      void main()
      {
            n = 0;
            parbegin (producer, consumer);
      }
```

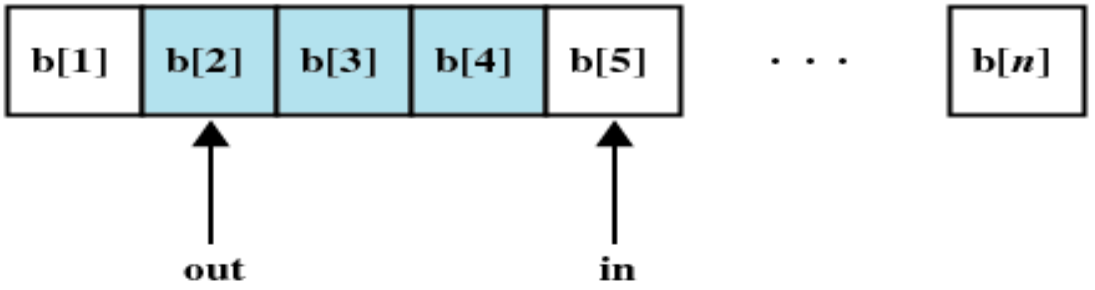
5.3.3 Producer/Consumer Problem(12/14)

- 同 34 页 A better solution: using counting semaphore
  - 读信号可以累积

```
/* program producerconsumer */
      semaphore n = 0, s = 1;
      void producer()
      {
            while (true) {
                  produce();
                  semWait(s);
                  append();
                  semSignal(s);
                  semSignal(n);
            }
      }
      void consumer()
      {
            while (true) {
                  semWait(n);
                  semWait(s);
                  take();
                  semSignal(s);
                  consume();
            }
      }
      void main()
      {
            parbegin (producer, consumer);
      }
```

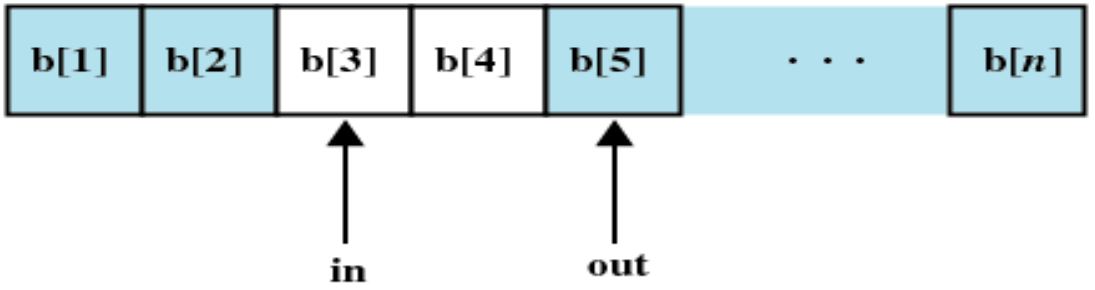
5.3.3 Producer/Consumer Problem(13/14)

Finite Circular Buffer



(a)

In 即将写入的位置
Out 即将读出的位置



(b)

Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

5.3.3 Producer/Consumer Problem(14/14)

with Circular Buffer

```
producer:
while (true) {
    /* produce item v */
    while ((in + 1) % n == out)
        /* do nothing */;
    b[in] = v;
    in = (in + 1) % n
}
```

```
consumer:
while (true) {
    while (in == out)
    /* do nothing */;
    w = b[out];
    out = (out + 1) % n;
    /* consume item w */
}
```

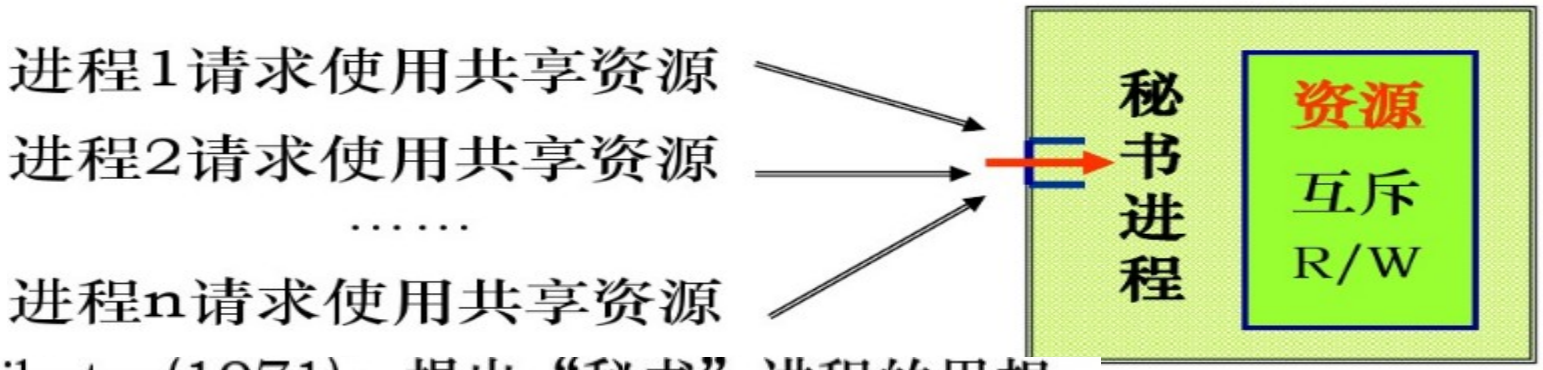
灰色部分置换为信号量 Fig 5.13 sem empty = n;

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

5.4 Monitors(1/10)

- Why Monitor?
  - Semaphore should be deployed with cautious.
    - E.g. semWait() in correct order
  - semWait, semSignal be scattered throughout the program.



Dijkstra(1971)：提出“秘书”进程的思想。

Hansen和Hoare(1973)：推广为“管程”。

5.4 Monitors(2/10)

- Definition:
  - Monitor is a software module consisting of one or more procedures, an initialization sequence, and local data(管程由一个或者多个例程、一段初始化代码和局部数据组成).

- the chief characteristics :
  - Local data variables are accessible only by the monitor
  - Process enters monitor by invoking one of its procedures 进程调用管程函数即进入管程
  - Only one process may be executing in the monitor at a time

- 5.4 Monitors(3/10)

1. Local data
2. Condition variables
3. Procedures
4. Initialization code

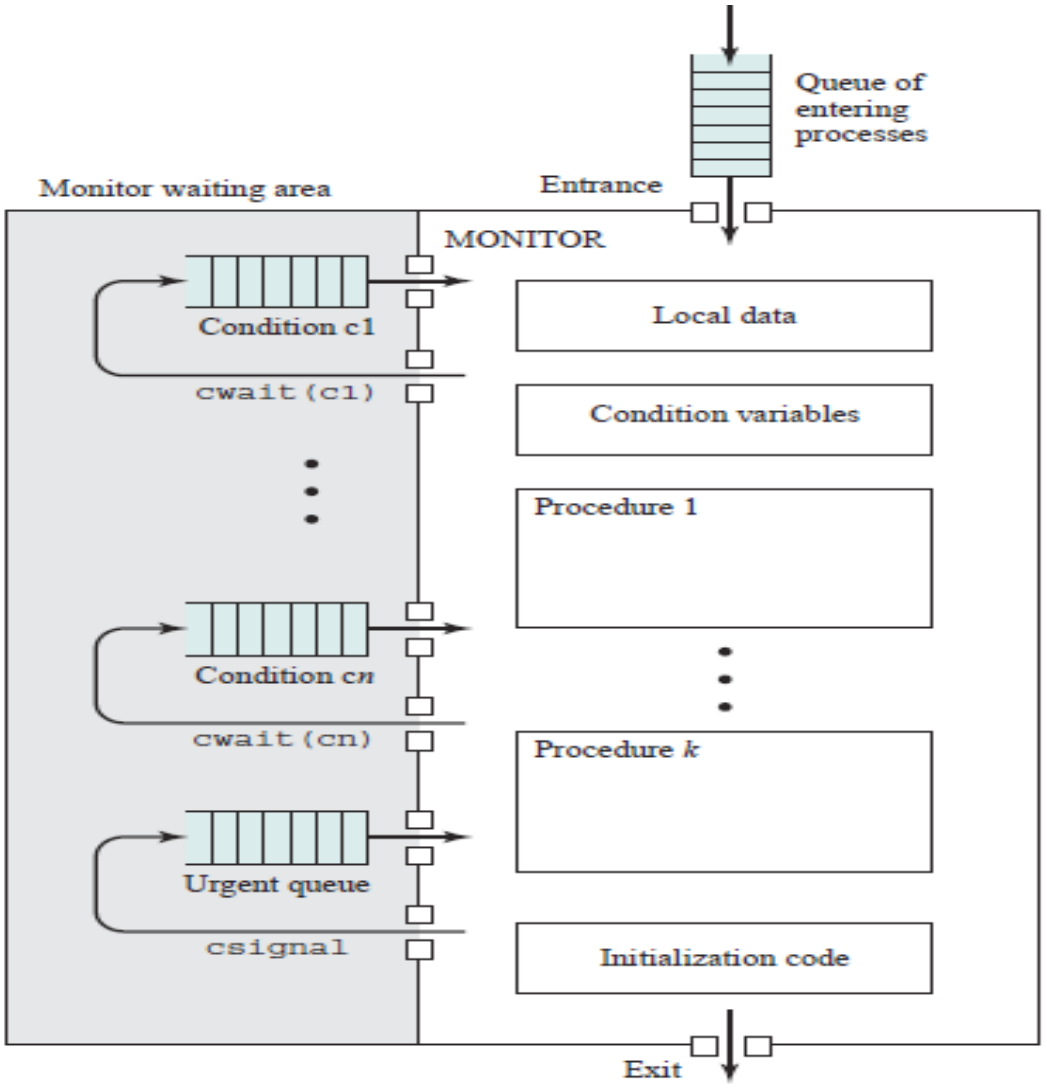


Figure 5.15 Structure of a Monitor

5.4 Monitors(4/10)

- A monitor supports synchronization by the use of **condition variables** (管程通过条件变量实现同步)
  - cwait(c): **Block** execution of the calling process on condition *c*. The monitor is now available for use by another process.
  - csignal(c): **Resume** execution of some process blocked after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

5.4 Monitors(5/10)

- 管程和信号量的 Differences ：
- monitor *wait* and *signal* vs semaphore *wait* and *signal*

- **If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost.**

5.4 Monitors(6/10)

- One monitor object
  - One lock ：一次只能有一个进程
  - Several conditional objects
    - Object: Counting variable and queue
    - Operation: cwait/csingal
  - Several functions

5.4 Monitors(7/10)

管程下的生产者消费者问题

```
boundedbuffer bf;// 管程对象
int data;
void main(){
parbegin (producer,comsumer);
}
```

```
void producer(void *arg){
  while(1){
          data = rand()%100;
          bf.deposit(data);}
}
```

```
void consumer(void *arg){
  while(1){
          bf.remove(&data);}
}
```

5.4 Monitors(8/10)

```
class boundedbuffer{…// 管程类
Lock lock;
int count=0;
condition notfull, notempty;
deposit(void data);
remove(void *data);}
```

```
boundedbuffer::deposit(void data){
    lock.lock();
    while(count==n)
        notfull.wait(&lock)
    add data to buffer
    count++;
    notempty.signal();
    lock.release();
}
```

```
boundedbuffer::remove(void *data){
    lock.lock();
    while(count==0)
        notempty.wait(&lock)
    remove data from buffer
    count--;
    notfull.signal();
    lock.release();
}
```

5.4 Monitors(9/10)

```
class condition{
    int numWaiting = 0;// 阻塞线程计数
    waitqueue q;// 阻塞等待队列
    wait(Lock *plock);
    signal( ); }
```

```
condition::wait(Lock *plock){
    numWaiting++;
    add this thread to q;
    release(plock);
    schedule();<---Reentry point
    require(plock);
}
```

```
condition::signal( ) {
    if(numWaiting>0){
    remove a thread t from q;
    wakeup(t);
    numWaiting--;
}}
```

5.4 Monitors(10/10)

- 实例：
- 实现管程类 monitor_class
- 需要 Linux support ：

  pthread_mutex_t lock 互斥锁

  +

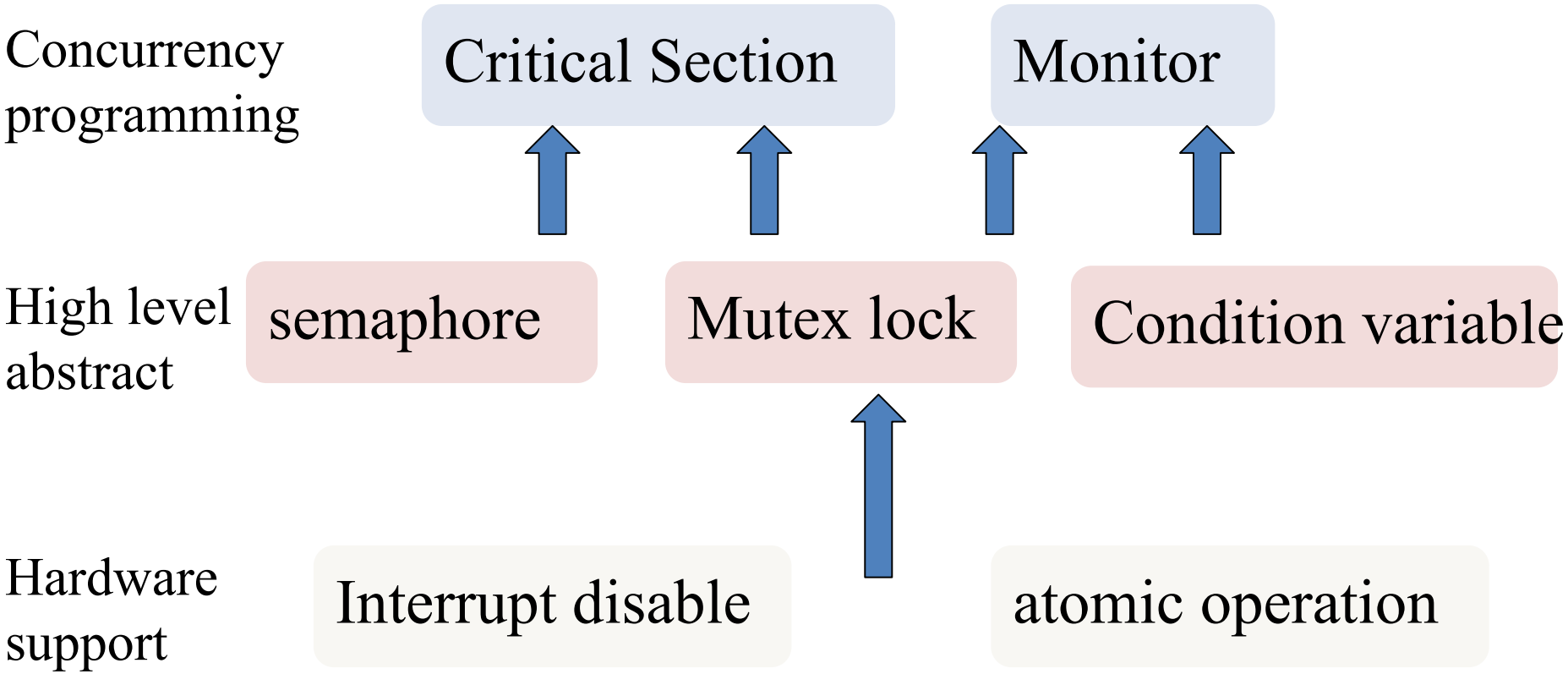
  pthread_cond_t consume_cond, produce_cond; 条件变量

summary

Table 5.3 Common Concurrency Mechanisms

Semaphore	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**.
Binary semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
Condition variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures, and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.

summary


Concurrency programming
Critical Section
Monitor
High level abstract
semaphore
Mutex lock
Condition variable
Hardware support
Interrupt disable
atomic operation