临界区，忙等待

忙等待：是否允许进入？若不允许，则该进程将原地等待，直到允许为止。

缺点：浪费CPU时间；优先级反转。

IPC原语：在无法进入临界区时将阻塞，而不是忙等待。

sleep是一个将引起调用进程阻塞的系统调用，即被挂起，直到另外一个进程将其唤醒。

wakeup调用有一个参数，即要被唤醒的进程。

两个经典的同步/互斥问题

- 生产者与消费者（有界缓冲区）

- 写者与读者

Sleep and Wakeup

```
#define N 100                                    /* number of slots in the buffer */
int count = 0;                                   /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                              /* repeat forever */
          item = produce_item( );                /* generate next item */
          if (count == N) sleep( );              /* if buffer is full, go to sleep */
          insert_item(item);                     /* put item in buffer */
          count = count + 1;                     /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);      /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                              /* repeat forever */
          if (count == 0) sleep( );              /* if buffer is empty, got to sleep */
          item = remove_item( );                 /* take item out of buffer */
          count = count – 1;                     /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer);  /* was buffer full? */
          consume_item(item);                    /* print item */
     }
}
```

The producer-consumer problem with a fatal race condition.

含有严重竞争条件的生产者-消费者问题

竞争条件：其原因是对count的访问未加限制。

1. 缓冲区为空，消费者刚刚读取count的值发现它为0。
2. 此时调度程序决定暂停消费者并启动运行生产者。
3. 生产者向缓冲区中加入数据项，调用wakeup唤醒消费者。
4. 消费者此时在逻辑上并未睡眠，所以wakeup信号丢失。
5. 当消费者运行时，测试先前读到的count值为0，于是睡眠。
6. 生产者迟早会填满整个缓冲区，然后睡眠。

问题的实质：发给（尚）未睡眠进程的wakeup信号丢失了。

唤醒等待位：wakeup信号的一个小仓库。几个？

Semaphore（信号量）

- Semaphore
  - 1965 E. W. Dijkstra
- PV operations (down和up)
  - Atomic operations: very quick and uninterruptable
  - Also known as down/up or wait/signal operations
- Semaphore can be operated by PV operation only.
  - The only exception is the initialization.
  - Even read the value is prohibited

```
struct semaphore
  {
   int value;            //使用整型变量来累计唤醒次数，供以后使用。
   pointer_PCB queue;//阻塞在该信号量的各个进程的标识
  }
```

semaphore

- What’s is semaphore?
  - An integer variable with a wait queue (usu. first in first out) for the process
- Variable range has different definitions:
  - Whole integer
    - Positive: the number of resources available
    - Zero: no resource available, no waiting process
    - Negative: the number of processes waiting to use the resources
  - Zero and positive integer
    - Positive: the number of resources available
    - Zero: no resource available
  - Zero and One only
    - Binary semaphores

PV in Sleep and Wakeup

```
P (Semaphore s)
{
  if (s > 0)
    s --;
  else   if(s == 0)
  {added to the semaphore's
      queue and sleep; }
}
```

```
V (Semaphore s)
{
  if (s's queue > 0)
  {wakeup the waiting process  in
    the semaphore's queue;}
  else if(s's queue == 0)
      s ++;
}
```

P & V

- 对一信号量执行P操作：检查其值是否大于0。

1. 若大于0，则将其值减1（用掉一个保存的唤醒信号）并继续；
2. 若为0，则进程将睡眠，而且此时P操作并未结束。

- V操作对信号量的值增1。

1. 如果一个或多个进程在该信号量上睡眠，无法完成一个先前的P操作，则由系统选择其中的一个并允许该进程完成它的P操作。
2. 于是，对一个有进程在其上睡眠的信号量执行一次V操作之后，该信号量的值仍旧是0，但在其上睡眠的进程却少了一个。
3. 信号量的值增1和唤醒一个进程同样也是不可分割的。不会有某个进程因执行V而阻塞，正如在前面的模型中不会有进程因执行wakeup而阻塞一样。

P & V 原子操作

检查数值、修改变量值以及可能发生的睡眠操作均作为一个单一的、不可分割的原子操作完成。

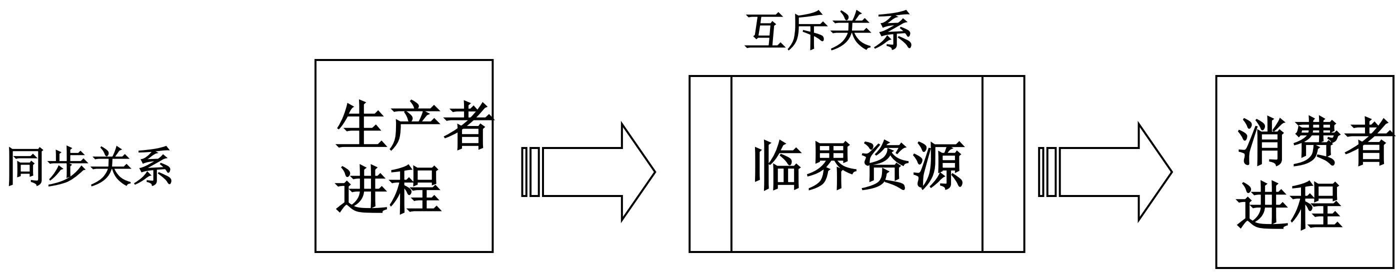
保证一旦一个信号量操作开始，则在该操作完成或阻塞之前，其他进程均不允许访问该信号量。

所谓原子操作，是指一组相关联的操作要么都不间断地执行，要么都不执行。

P和V作为系统调用实现，而且操作系统只需在执行以下操作时暂时屏蔽全部中断：测试信号量、更新信号量以及在需要时使某个进程睡眠。由于这些动作只需要几条指令，所以屏蔽中断不会带来什么副作用。

生产者与消费者问题

- 模型的抽象化与进程分析
- 用信号量解决丢失的wakeup问题



信号量的初始化设置

mutex=1　临界区互斥，1表示可以进　（互斥用）

empty=n　记录空的缓冲槽数目　（同步用）

full=0　记录充满的缓冲槽数目　（同步用）

Semaphores

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

Mutexes

- 如果不需要信号量的计数能力，有时可以使用信号量的一个简化版本，称为互斥量（mutex）。

- 互斥量仅仅适用于管理共享资源或一小段代码。

- Variable range has different definitions:
  - Zero and One only　　解锁和加锁
    - Binary semaphores

Mutexes (在用户级线程中)

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again
ok:     RET                         | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET                         | return to caller
```

取锁失败时，调用thread_yield将CPU放弃给另一个线程。这样，就没有忙等待。在该线程下次运行时，它再一次对锁进行测试。

Implementation of mutex lock and mutex unlock.

The TSL Instruction

```
enter_region:
        TSL REGISTER,LOCK        | copy lock to register and set lock to 1
        CMP REGISTER,#0          | was lock zero?
        JNE enter_region         | if it was nonzero, lock was set, so loop
        RET                      | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0             | store a 0 in lock
        RET                      | return to caller
```

mutex_lock 的代码与enter_region的代码很相似，但有一个关键的区别。当enter_region进入临界区失败时，它始终重复测试锁（忙等待）。

实际上，由于时钟超时的作用，会调度其他进程运行。这样迟早拥有锁的进程会进入运行并释放锁。

Mutexes in Pthreads

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

mutex_trylock，或者获得锁或者返回失败码，但并不阻塞线程。给调用线程灵活性，用以决定下一步做什么，是使用替代办法还只是等待下去。

Some of the Pthreads calls relating to mutexes.

线程、进程中的Mutex

在用户级线程包中，多个线程访问同一个互斥量是没有问题的，因为所有的线程都在一个公共地址空间中操作。

问题：对于大多数早期解决方案，诸如Peterson算法和信号量等，都有一个未说明的前提，即这些多个进程至少应该访问一些共享内存，也许仅仅是一个字。如果进程有不连续的地址空间，如我们始终提到的，那么在Peterson算法、信号量或公共缓冲区中，它们如何共享turn变量呢？

解决：

1. 有些共享数据结构，如信号量，可以存放在内核中，并且只能通过系统调用来访问。
2. 让进程与其他进程共享其部分地址空间。缓冲区和其他数据结构可以共享。最坏的情形下，可以使用共享文件。

同步机制：条件变量

互斥量在允许或阻塞对临界区的访问上是很有用的；
条件变量则允许线程由于一些未达到的条件而阻塞。

情境：

1. 如果生产者发现缓冲区中没有空槽可以使用，它不得不阻塞起来直到有一个空槽可以使用。
2. 生产者使用互斥量进行原子性检查，而不受其他线程干扰。
3. 但是当发现缓冲区已经满了以后，生产者需要一种方法来阻塞自己并在以后被唤醒。这便是条件变量的职责。

条件变量与互斥量经常一起使用。用于让一个线程锁住一个互斥量，然后当它不能获得它期待的结果时等待一个条件变量。

另一个线程会向它发信号，使它可以继续执行。pthread_cond_wait原子性地调用并解锁它持有的互斥量。由于这个原因，互斥量是参数之一。

condition variables in Pthreads

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

cond_wait原子性地调用并解锁它持有的互斥量。由于这个原因，互斥量是参数之一。

Some of the Pthreads calls relating to condition variables.

Mutexes in Pthreads

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                     /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;                            /* buffer used between producer and consumer */

void *producer(void *ptr)                  /* produce data */
{     int i;

      for (i= 1; i <= MAX; i++) {
           pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
           while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
           buffer = i;                       /* put item in buffer */
           pthread_cond_signal(&condc);      /* wake up consumer */
           pthread_mutex_unlock(&the_mutex);/* release access to buffer */
      }
      pthread_exit(0);
}

void *consumer(void *ptr)                  /* consume data */
{     int i;

      for (i = 1; i <= MAX; i++) {
           pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
           while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
           buffer = 0;                       /* take item out of buffer */
           pthread_cond_signal(&condp);      /* wake up producer */
           pthread_mutex_unlock(&the_mutex);/* release access to buffer */
      }
      pthread_exit(0);
}

int main(int argc, char **argv)
{
      pthread_t pro, con;
      pthread_mutex_init(&the_mutex, 0);
      pthread_cond_init(&condc, 0);
      pthread_cond_init(&condp, 0);
      pthread_create(&con, 0, consumer, 0);
      pthread_create(&pro, 0, producer, 0);
      pthread_join(pro, 0);
      pthread_join(con, 0);
      pthread_cond_destroy(&condc);
      pthread_cond_destroy(&condp);
      pthread_mutex_destroy(&the_mutex);
}
```

这里的buffer最大等于1，只有一个缓存空间

Using threads to solve the producer-consumer problem.

管程（ Monitors ）

管程：一个由过程、变量及数据结构等组成的一个集合，它们组成一个特殊的模块或软件包。

进程可在任何需要的时候调用管程中的过程，但它们不能在管程之外声明的过程中直接访问管程内的数据结构。

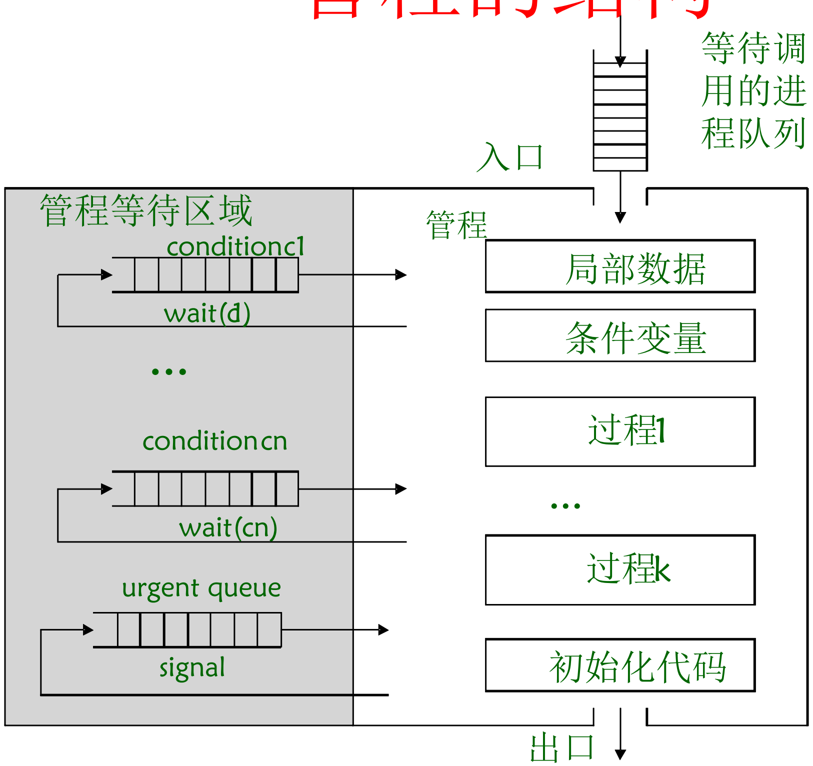
管程有一个很重要的特性，即任一时刻管程中只能有一个活跃进程，这一特性使管程能有效地完成互斥。

进入管程时的互斥由编译器负责，但通常的做法是用一个互斥量或二元信号量。

管程提供了一种实现互斥的简便途径，还需要一种办法使得进程在无法继续运行时被阻塞。条件变量

Monitors

管程的结构



monitor *example*
    integer *i*;
    condition *c*;

    procedure *producer*( );
    .
    .
    .
    end;

    procedure *consumer*( );
    .   .   .
    end;
end monitor;

A monitor.

Monitors

```
monitor ProducerConsumer
      condition full, empty;
      integer count;
      procedure insert(item: integer);
      begin
            if count = N then wait(full);
            insert_item(item);
            count := count + 1;
            if count = 1 then signal(empty)
      end;
      function remove: integer;
      begin
            if count = 0 then wait(empty);
            remove = remove_item;
            count := count − 1;
            if count = N − 1 then signal(full)
      end;
      count := 0;
end monitor;
```

```
procedure producer;
begin
      while true do
      begin
            item = produce_item;
            ProducerConsumer.insert(item)
      end
end;
procedure consumer;
begin
      while true do
      begin
            item = ProducerConsumer.remove;
            consume_item(item)
      end
end;
```

An outline of the producer-consumer problem with monitors.

Monitors in Java

```
public class ProducerConsumer {
        static final int N = 100;       // constant giving the buffer size
        static producer p = new producer( );    // instantiate a new producer thread
        static consumer c = new consumer( ); // instantiate a new consumer thread
        static our_monitor mon = new our_monitor( );      // instantiate a new monitor

        public static void main(String args[ ]) {
           p.start( );       // start the producer thread
           c.start( );       // start the consumer thread
        }

        static class producer extends Thread {
           public void run( )  {// run method contains the thread code
              int item;
              while (true) {      // producer loop
                 item = produce_item( );
                 mon.insert(item);
              }
           }
           private int produce_item( ) { ... }       // actually produce
        }
```

A solution to the producer-consumer problem in Java.

Monitors in Java

```
static class consumer extends Thread {
   public void run( ) {run method contains the thread code
      int item;
      while (true) {      // consumer loop
         item = mon.remove( );
         consume_item (item);
      }
   }
   private void consume_item(int item) { ... }// actually consume
}

static class our_monitor {  // this is a monitor
   private int buffer[ ] = new int[N];
   private int count = 0, lo = 0, hi = 0;  // counters and indices

   public synchronized void insert(int val) {
      if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
      buffer [hi] = val; // insert an item into the buffer
      hi = (hi + 1) % N;        // slot to place next item in
      count = count + 1;        // one more item in the buffer now
      if (count == 1) notify( );        // if consumer was sleeping, wake it up
   }
```

Synchronized:保证一旦某个线程执行该方法，就不允许其他线程执行该对象中的任何synchronized方法。

Monitors in Java

```
    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
        val = buffer [lo]; // fetch an item from the buffer
        lo = (lo + 1) % N;         // slot to fetch next item from
        count = count – 1;         // one few items in the buffer
        if (count == N – 1) notify( ); // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};
  }
}
```

当生产者在insert内活动时，它确信消费者不能在remove中活动，从而保证更新变量和缓冲区的安全

lo是缓冲区槽的序号，指出将要取出的下一个数据项。
hi是缓冲区中下一个将要放入的数据项序号。
lo= hi，其含义是在缓冲区中有0个或N个数据项。count的值说明了究竟是哪一种情形。

Monitors in Java

Java没有内嵌的条件变量。

Java提供wait和notify，与sleep和wakeup等价，不过，当它们在Synchronized同步方法中使用时，不受竞争条件约束。理论上，wait可以被中断，它本身就是与中断有关的代码。

通过临界区互斥的自动化，管程比信号量更容易保证并行编程的正确性。

缺点：管程是一个编程语言概念，编译器必须要识别管程并用某种方式对其互斥做出安排。
C、Pascal以及多数其他语言都没有管程，所以指望这些编译器遵守互斥规则是不合理的。

如果一个分布式系统具有多个CPU，并且每个CPU拥有自己的私有内存，它们通过一个局域网相连，那么这些原语将失效。---消息传递

Message Passing

- Send(destination, &message);
- Receive(source, &message); // non-message? Blocked

1. Idempotence ?

   (aside from error or expiration issues) the side-effects of

   N > 0 identical operations is the same as for a single operation.

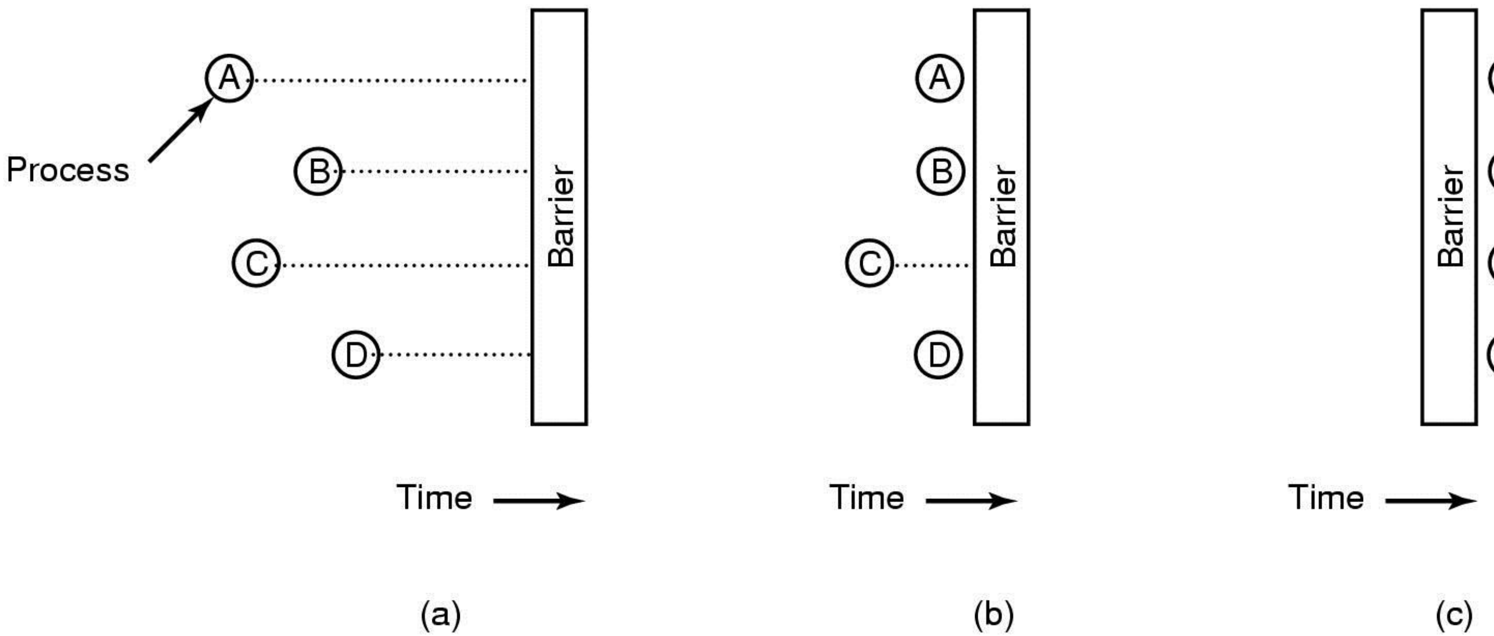
2. Authentication ?
3. Performance ?

Producer-Consumer Problem with Message Passing

```
#define N 100                           /* number of slots in the buffer */

void producer(void)
{
      int item;
      message m;                        /* message buffer */

      while (TRUE) {
            item = produce_item( );     /* generate something to put in buffer */
            receive(consumer, &m);      /* wait for an empty to arrive */
            build_message(&m, item);    /* construct a message to send */
            send(consumer, &m);         /* send item to consumer */
      }
}

void consumer(void)
{
      int item, i;
      message m;

      for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
      while (TRUE) {
            receive(producer, &m);      /* get message containing item */
            item = extract_item(&m);    /* extract item from message */
            send(producer, &m);         /* send back empty reply */
            consume_item(item);         /* do something with the item */
      }
}
```

The producer-consumer problem with N messages.

Barriers



Use of a barrier.

(a) Processes approaching a barrier.

(b) All processes but one blocked at the barrier.

(c) When the last process arrives at the barrier, all of them are let through.