6.7 Monitors
Problems with Semaphores

- Incorrect use of semaphore operations

…. signal (mutex) …. wait (mutex) …. （顺序颠倒）	…. wait (mutex) …. wait (mutex) …. (signal误为wait)	…. wait (mutex) …. …. Omitting of wait (mutex) or signal (mutex) (or both)

信号量及wait、signal操作存在的问题

- 信号量及wait、signal操作使用不当，会违反同步机制应遵循的规则。
  - wait与signal位置倒置－mutual exclusion is violated ；
  - 将signal误写成wait－deadlock will occur;
  - 遗漏wait或signal－ mutual exclusion is violated or deadlock will occur;
  - wait的顺序不当－ deadlock will occur;
- solution
  - 提供更高层的方便用户同步机制，系统（e.g.compiler）将其映射到底层的信号量及wait、signal操作;
    - monitor

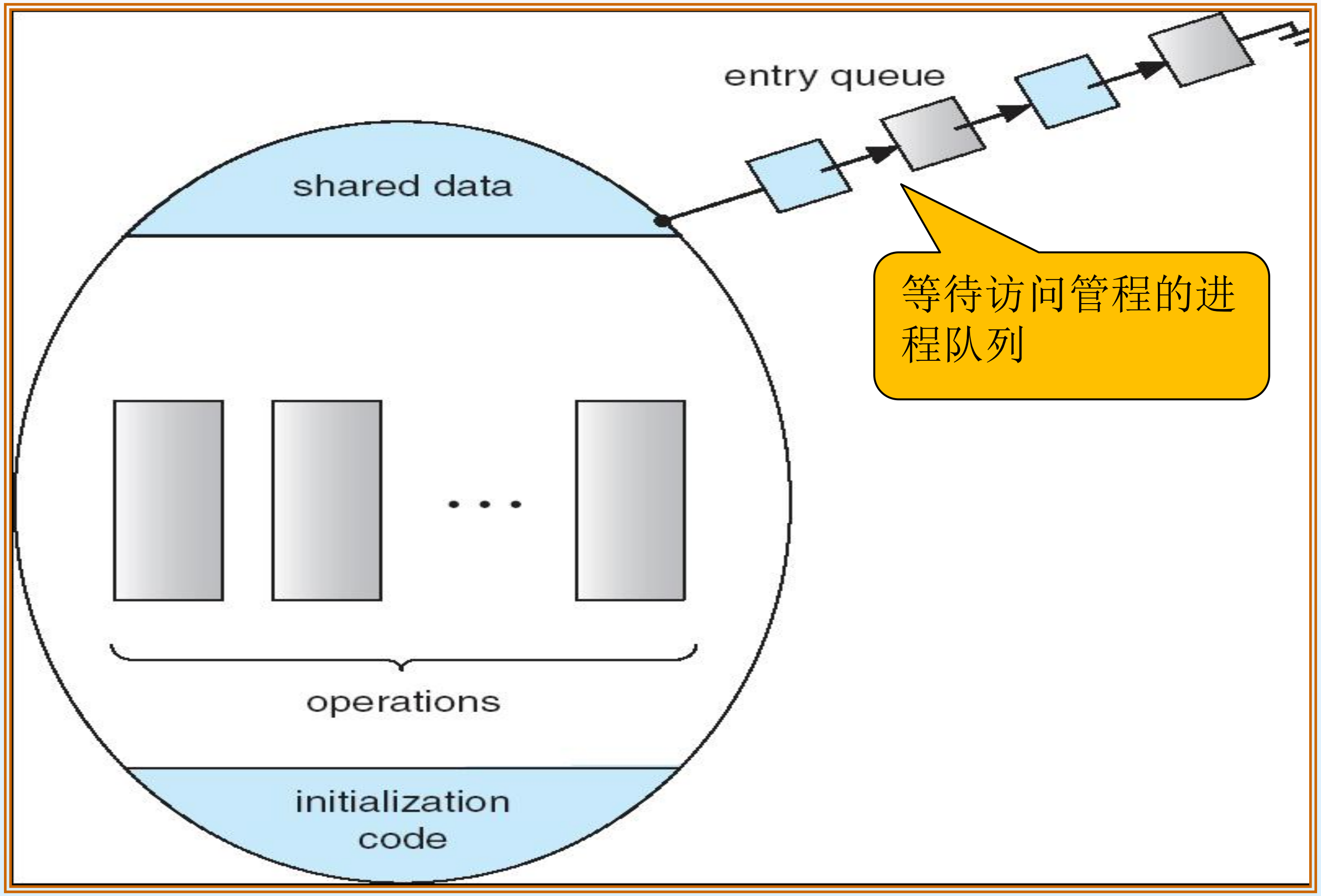
6.7.1 Usage

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization ;
- Only one process may be active within the monitor at a time ;
  - Monitor要互斥

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
              …
    procedure Pn (…) {……}

    Initialization code ( ….) { … }
              …
    }
}
```

Schematic view of a Monitor



Only one process may be active within the monitor at a time；

Figure 6.17 Schematic view of a Monitor

管程下的wait and signal operation

- 可把管程的定义理解为一个类定义； 与一般的类不同的是，管程有条件变量，用于控制进程之间的同步；
- 并发的进程要*互斥*访问管程；
  - Only one process may be active within the monitor at a time ;
- 当某进程通过管程请求临界资源而未满足时，管程调用**wait**原语使该进程等待，并将它排在等待队列上；
- 当另一进程访问完并释放之后，管程调用**signal**原语唤醒等待队列中的某个进程；

Condition Variable

- 通常，进程等待的原因有多个，为了区分这些原因，引入条件变量；
- 例如在生产者－消费者问题中，进程可以在**empty**、**full**或**mutex**信号量对应的等待队列中等待；
  - 在不同信号量的等待队列中的进程，等待的原因是不同的；
- 管程中对每个条件变量，都予以声明；
  - **Condition x，y**
- 该变量置于**wait**和**signal**之前，即可表示为
  - **x.wait, x.signal**
  - **e.g. empty.wait**：等待空缓冲区

    empty.signal：唤醒等待空缓冲区的进程

Condition Variable

- To allow a process to wait within the monitor, a **condition** variable must be declared, as:
  condition x, y;
- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation **x.wait()** means that the process invoking this operation is **suspended until another process invokes x.signal();**
  - The **x.signal operation resumes exactly one suspended process (if any) that invoked x.wait ()**. *If no process is suspended, then the **signal** operation has no effect.*

Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()
    - 根据x.signal ()的语义，如果没有在条件变量x中等待的进程，也可以执行x.signal()，只是执行x.signal()后不会产生任何效果

x.signal()

- 当一个进程**P执行了x.signal()**，而进程Q正在条件变量x的等待队列中，Q将被唤醒，并可被调度执行
- **由于管程需要互斥访问**，因此P与Q两个进程中只有一个能够运行，否则管程中会有两个进程同时在执行（多处理机系统中）；
- Two possibilities exist:
  - **1. Signal and wait.** *P* either waits until Q leaves the monitor or waits for another condition.
  - P执行x.signal()后，唤醒了进程Q，则P进入等待
    - P要么等到Q执行后离开管程，要么等待另一个条件。
  - **2. Signal and continue. Q** either waits until *P* leaves the monitor or waits for another condition.
  - P执行x.signal()后，唤醒了进程Q，则P继续执行，Q进入等待
    - Q要么等到P执行后离开管程，要么等待另一个条件

Monitor with Condition Variables

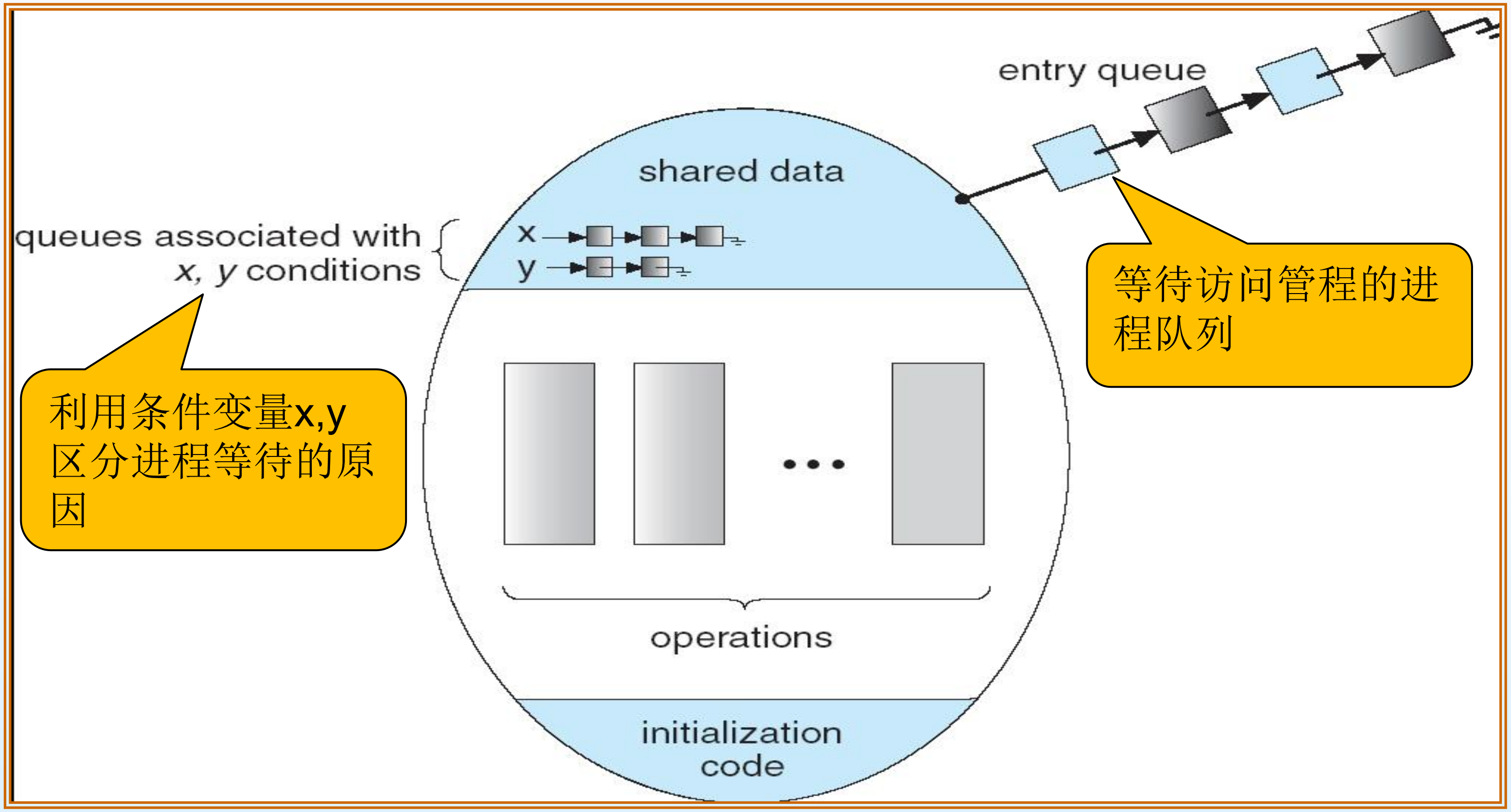


Figure 6.18 Monitor with Condition Variables

例：管程的使用--利用管程解决P-C问题

- 建立一个管程，命名为PC。
- 其中，管程包含两个过程：
  - Put(item)
    - 生产者利用该过程，将自己生成的消息放到缓冲池中；
    - 并利用整型变量count来表示在缓冲池中已有的消息数；
    - 当count ≥n 时，表示缓冲池已满，生产者等待；
  - get(item)
    - 消费者利用该过程，从缓冲池中获取一个消息；
    - 当count ≤0 时，表示缓冲池中无可用消息，消费者等待；

利用管程解决生产者—消费者问题

Producer: //生产者

```
do {
        …
        produce an item in nextp;
        pc.put(nextp);
        …
    } while (1);
```

Consumer: //消费者

```
do {
        …
        pc.get(nextc);
        consumer the item in nextc;
        …
} while (1);
```

管程的定义

```
monitor pc
  {
        int in,out,count;
        item buffer[n];

        condition empty,full;   //条件变量

         void put(item i);        // following slides
         void get(item i);        // following slides

         void init() {
                in=0;
                out=0;
                count=0;
         }
  }
```

void put(item i)

```
void put(item i)
  {
    if (count >= n) empty.wait;  //缓冲池满，等待空缓冲区
    buffer[in] = i;
    in =  (in+1)%n;
    count++;
   //if  (full.queue)  full.signal;   //有消费者等待，则唤醒之
    full.signal;   //有消费者等待，则唤醒之，否则，无效果
  }
```

void get(item i)

```
void get(item i)
  {
    if (count <= 0) full.wait;    //缓冲池空，等待满缓冲区
    i = buffer[out] ;
    out =  (out+1)%n;
    count- -;
    //if  (empty.queue)  empty.signal;  //若有生产者等待，则唤醒 之
    empty.signal;  //若有生产者等待，则唤醒之,否则，无效果
  }
```

利用管程解决生产者—消费者问题

Producer: //生产者

```
do {
        …
        produce an item in nextp;
        pc.put(nextp);
        …
    } while (1);
```

Consumer: //消费者

```
do {
        …
        pc.get(nextc);
        consumer the item in nextc;
        …
} while (1);
```

习题 6.13 （ Bounded Buffer Problem）

另一种描述

```
monitor bounded_buffer {
    int items[MAX_ITEMS];
    int numItems = 0;   //满缓冲区个数
    condition empty, full;

    void produce(int v) {
        //如果缓冲池满，则等待空缓冲区
         if (numItems == MAX_ITEMS)
            empty.wait();
         items[numItems++] = v;
         full.signal();
     }
```

```
int consume() {
    int retVal;
    //如果缓冲池空，则等待满缓冲区
    if (numItems == 0)
        full.wait();
    retVal = items[--numItems];
    empty.signal();
    return retVal;
    }
} //monitor
```

讨论

- 上述示例中，没有考虑管程的互斥访问问题
- 如何保证管程的互斥访问？

讨论

- 上述示例中，没有考虑管程的互斥访问问题
- 如何保证管程的互斥访问？
- 如果系统支持管程机制，则
  - 由系统提供管程的互斥访问机制实现管程的互斥访问
  - Only one process may be active within the monitor at a time；
- 如果系统不支持管程机制，管程由用户自己定义，则
  - 可以在put()与get()中设置一个互斥信号量来实现
  - 也可以在put()与get()中使用系统提供的lock机制实现
- 参见
  - 实验6中示例程序；
  - 参见 Nachos code/monitor/ring.cc中的put()与get()

6.7.2 Dining-Philosophers Solution Using Monitors

- 筷子（资源）的分配由管程来控制；
- Each philosopher i invokes the operations pickup() and putdown() in the following sequence:

dp.pickup (i) //如果第i个哲学家自己饥饿，且左右两只筷子同时
// 空闲，则吃饭；否则，等待；

EAT;

dp.putdown (i) // 第i个哲学家放下筷子，同时测试左右邻居
// 等待吃饭；
// 如果有等待的哲学家，则唤醒之；

Solution to Dining Philosophers(cont)

- 每个哲学家的状态初始化为 state[i] = THINKING;

- 第i个哲学家具备吃饭的条件是：
  - 1、自己状态为饥饿，即state[i] = HUNGRY;
  - 2、左右两个哲学家都不在吃饭，即两边的筷子时空闲的，能同时拿起左右两只筷子
- 因此，一个哲学家想要吃饭，首先将自己的状态设置为HUNGRY，然后测试左右两只筷子是否可用；
  - 如果可用，将自己的状态设为EATING,开吃；(见pickup(i))
  - 如果不可用，自己状态维持HUNGRY(!=EATING)，并进入等待；
- 当一个哲学家吃完饭，将自己的状态设置为THINKING，然后放下筷子，然后测试左右哲学家是否在等待吃饭，如果等待，则唤醒他们。（见putdown(i)）

Solution to Dining Philosophers(cont)

```
monitor DP
  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];   //为每个哲学家分别设置一个条件变量

    void pickup (int i) {
         state[i] = HUNGRY;
        //如果一个哲学家具备吃饭的条件，则把自己的状态设为正在吃饭，
         //如果自己原来等待吃饭，则被唤醒；
         test(i);    // test(i)后，state[i] == EATING，  或 state[i] != EATING
         if (state[i] != EATING) self [i].wait;  //如果测试后发现两只筷子不能同时使用，则等待
    }

     void putdown (int i) {
         state[i] = THINKING;
              // test left and right neighbors
          test((i + 4) % 5);   //放下左筷子，测试左边的哲学家是否等待吃饭，如果是，唤醒之；
          test((i + 1) % 5);   //放下右筷子，测试右边的哲学家是否等待吃饭，如果是，唤醒之；
      }
```

Solution to Dining Philosophers (cont)

```
void test (int i) {
    //如果左右两个哲学家都不在吃饭且自己饥饿，则把自己的状态设为吃，
    //如果自己原来等待吃饭，则被唤醒
     if ( (state[(i + 4) % 5] != EATING) &&  //左边哲学家未吃饭
     (state[i] == HUNGRY) &&                 //自己饥饿
     (state[(i + 1) % 5] != EATING) )  {     //右边哲学家未吃饭
         state[i] = EATING ;                 //自己具备了吃饭的条件
          //下句由putdown(i)使用
          self[i].signal () ; //如果该哲学家不能同时拿起两只筷子,
                              //则state[i] != EATING，进入等待状态；
                              // (见void pickup (int i))
                              //当邻居哲学家放下筷子后，经过测试如果发现自己具
                              //备吃饭的条件，则被唤醒(见putdown (i))
            } //if
 } //test

 initialization_code() {
      for (int i = 0; i < 5; i++)
      state[i] = THINKING;
 }
}
```

6.7.3 Implementing a Monitor Using Semaphores

- 一般情况下，可以采用下述两种方法实现管程
  - 基于锁机制（lock）
  - 基于信号量（semaphore）
    - Hoare and Brinch-Hansen解决方案
  - 上述两种方法在Nachos的 code/monitor/synch.cc中均有实现
  - 其使用方法可参阅code/monitor/ring.cc

Implementing a Monitor Using Semaphores

- For each monitor, a semaphore mutex (initialized to 1) is provided.（to ensure the monitor is accessed mutually）
- A process
  - must execute wait(mutex) before entering the monitor and
  - must execute signal(mutex) after leaving the monitor
- Suppose a signaling process must wait until the resumed process either leaves or waits (**signal and wait**)
  - an additional semaphore, next, is introduced, initialized to 0, on which the signaling processes may suspend themselves.
  - An integer variable next_count is also provided to count the number of processes suspended on next.

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
   …
  body of F;

   …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation x.wait can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

Monitor Implementation

- The operation x.signal can be implemented as:

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

6.7.4 Resuming Processes Within a Monitor

- If several processes are suspended on condition x, and an x.signal () operation is executed by some process, then how do we determine which of the suspended processes should be resumed next?
  - FCFS
    - One simple solution is to use an FCFS ordering, so that the process waiting the longest is resumed first
  - Priority-based
    - a **priority** number is stored with the name of the process that is suspended
    - When x.signal () is executed, the process with the smallest associated priority number is resumed next
    - x.wait (c) ,where c is an integer expression that is evaluated when the wait() operation is executed.

6.8 Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems;
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable
- Windows API
  - 互斥对象：mutex（create、open、release）
  - 临界区：critical section
  - 信号量：semaphore
  - 事件event：相当于触发器，通知一个或多个线程某事件的出现；

Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections

- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

Eaxmple of pthread mutex lock

```
int value=5;
void *runner1(void *param);
void *runner2(void *param);
pthread_mutex_t mutex; //mutex lock
int main(int argc, char *argv[])
{
   pthread_mutex_init(&mutex,NULL); //create lock
   pthread_t tid1,tid2;
   pthread_attr_t attr1,attr2;
  //============
   pthread_attr_init(&attr1);
   pthread_create(&tid1,&attr1,runner1,NULL);
   //=============
   pthread_attr_init(&attr2);
   pthread_create(&tid2,&attr2,runner2,NULL);

   printf("value=%d\n",value);  //理论上:4,5,6
   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
}//main
```

问：输出结果是什么？
//理论上，该程序存在问题，缺少对共享变量value的互斥访问

```
//threads
 void *runner1(void *param)
 {
     pthread_mutex_lock(&mutex);
     value += 1;
     pthread_mutex_unlock(&mutex);
     pthread_exit(0);
 }
 void *runner2(void *param)
 {
    pthread_mutex_lock(&mutex);
    value -= 1;
    pthread_mutex_unlock(&mutex);
pthread_exit(0);
 }
```

Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions

- Windows的系统还原

System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - **collection of instructions or operations that performs single logical function**
  - Here we are concerned with changes to stable storage – disk
  - **Transaction is series of read and write operations**
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed

Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
  - Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
  - Example: disk and tape
- Stable storage – Information never lost
  - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- **Record to stable storage information about all modifications by a transaction**
- Most common is write-ahead logging
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - <T_i starts> written to log when transaction T_i starts
  - <T_i commits> written when T_i commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - Undo(T_i) restores value of all data updated by T_i
  - Redo(T_i) sets values of all data in transaction T_i to new values
- Undo(T_i) and redo(T_i) must be **idempotent**
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains <T_i starts> without <T_i commits>, undo(T_i)
  - If log contains <T_i starts> and <T_i commits>, redo(T_i)

Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
  1. Output all log records currently in volatile storage to stable storage
  2. Output all modified data from volatile to stable storage
  3. Output a log record <checkpoint> to the log on stable storage
- Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti All other transactions already on stable storage

Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- Concurrency-control algorithms provide serializability

Serializability

- Consider two data items A and B
- Consider Transactions T_0 and T_1
- Execute T_0, T_1 atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are N! valid serial schedules

Schedule 1: T_0 then T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Nonserial Schedule

- Nonserial schedule allows overlapped execute
  - Resulting execution not necessarily incorrect
- Consider schedule S, operations O_i, O_j
  - Conflict if access same data item, with at least one write
- If O_i, O_j consecutive and operations of different transactions & O_i and O_j don't conflict
  - Then S' with swapped order O_j O_i equivalent to S
- If S can become S' via swapping nonconflicting operations
  - S is conflict serializable

Schedule 2: Concurrent Serializable Schedule

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - Shared – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q
  - Exclusive – Ti has exclusive-mode lock (X) on Q, T_i can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
  - Similar to readers-writers algorithm

Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
  - Growing – obtaining locks
  - Shrinking – releasing locks
- Does not prevent deadlock

Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction T_i associated with timestamp $TS(T_i)$ before T_i starts
  - $TS(T_i) < TS(T_j)$ if Ti entered system before T_j
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j

Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
  - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
  - R-timestamp(Q) – largest timestamp of successful read(Q)
  - Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order
- Suppose Ti executes read(Q)
  - If $TS(T_i) < \text{W-timestamp}(Q)$, Ti needs to read value of Q that was already overwritten
    - read operation rejected and T_i rolled back
  - If $TS(T_i) \geq \text{W-timestamp}(Q)$
    - read executed, R-timestamp(Q) set to max(R-timestamp(Q), $TS(T_i)$)

Timestamp-ordering Protocol

- Suppose Ti executes write(Q)
  - If $TS(T_i) < \text{R-timestamp}(Q)$, value Q produced by T_i was needed previously and T_i assumed it would never be produced
    - Write operation rejected, T_i rolled back
  - If $TS(T_i) < \text{W-tiimestamp}(Q)$, T_i attempting to write obsolete value of Q
    - Write operation rejected and T_i rolled back
  - Otherwise, write executed
- Any rolled back transaction T_i is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

Schedule Possible Under Timestamp Protocol

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

- Java Synchronization

Java Synchronization

- Synchronized, wait(), notify() statements
- Multiple Notifications (notifyall())
- Block Synchronization
- Java Semaphores
- Java Monitors

synchronized Statement

- ***Every object*** has a lock associated with it.
- Calling an ordinary method the lock is ignored.
- But, calling a ***synchronized*** method requires “owning” the lock.

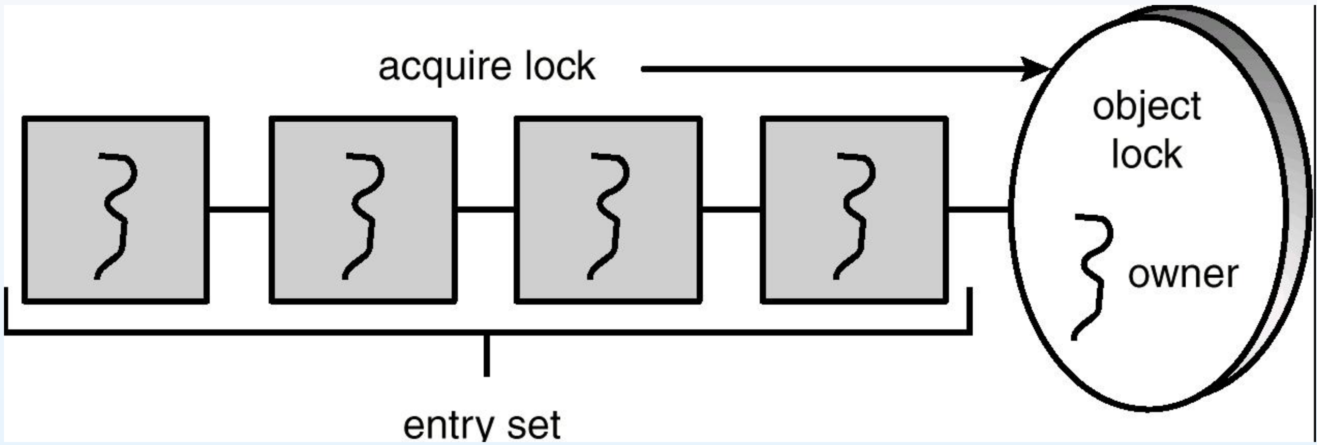
synchronized Statement

- ***If the lock is available when a synchronized method is called , the calling thread becomes the owner of the object's lock.***

- If a calling thread does not own the lock (another thread already owns it), the calling thread is blocked and is placed in the ***entry set*** for the object's lock.

- ***The lock is released when a thread exits the synchronized method,*** and the JVM selects an arbitrary thread from this set as the new owner of the lock.

Entry Set



注：**Entry set** 中的线程处于阻塞状态

也有的处于**runnable**状态（后面还要介绍）；

enter() Method - busy waiting without synchronized

```
public void enter(Object item) {
    while (count == BUFFER_SIZE)  ;
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

remove() Method - busy waiting without synchronized

```
public Object remove() {
    Object  item;
    while (count == 0) ;
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```

Problems

- The specification for the JVM does not indicates whether threads are time-sliced or not. It is up to the particular implementation of the JVM.
- Usually, the JVM schedules threads using a preemptive, priority-based schedules algorithm.
- So if no higher priority threads arrive, the executing thread will never relinquishes control of the CPU;

problems

- Race condition:
  - shared variables: *count*, *in* and *out*
- Solution:
  - *synchronized method*

- Busy waiting
  - Maybe the executing thread never relinquishes control of the CPU;
    - waste time
    - other threads have no opportunity to run;
- Solution(以后考虑**)**
  - *yield();*

enter() Method - busy waiting with synchronized

```
public synchronized void enter(Object item) {
    while (count == BUFFER_SIZE)  ;
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
```

remove() Method - busy waiting with synchronized

```
public synchronized Object remove() {
    Object  item;
    while (count == 0)  ;
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```

problems

- Busy waiting
  - Maybe the executing thread never relinquishes control of the CPU;
- Solution:
  - *yield();*

problems

- Can leads to a deadlock;
- Assume the producer owns the lock and the buffer is full, it is busy waiting and still keeps the lock.
- When the consumer is scheduled to run, the consumer will be blocked and is placed in the *entry set* for the object's lock.

- Then
  - The producer is waiting for the consumer to free space in the buffer;
  - The consumer is blocked waiting for the producer to release the lock.

yield() method

- When a thread invokes the yield() method, the thread stays in the runnable state, but it relinquishes control of the CPU and allows the JVM to select to run anther runnable thread of ***equal*** priority.

- The yield() method makes more effective use of the CPU than ***busy waiting*** does.

synchronized enter() Method with yield

```
public synchronized void enter(Object item) {
    while (count == BUFFER_SIZE)
        Thread.yield();
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

synchronized remove() Method with yield

```
public synchronized Object remove() {
    Object  item;
    while (count == 0)
        Thread.yield();
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```

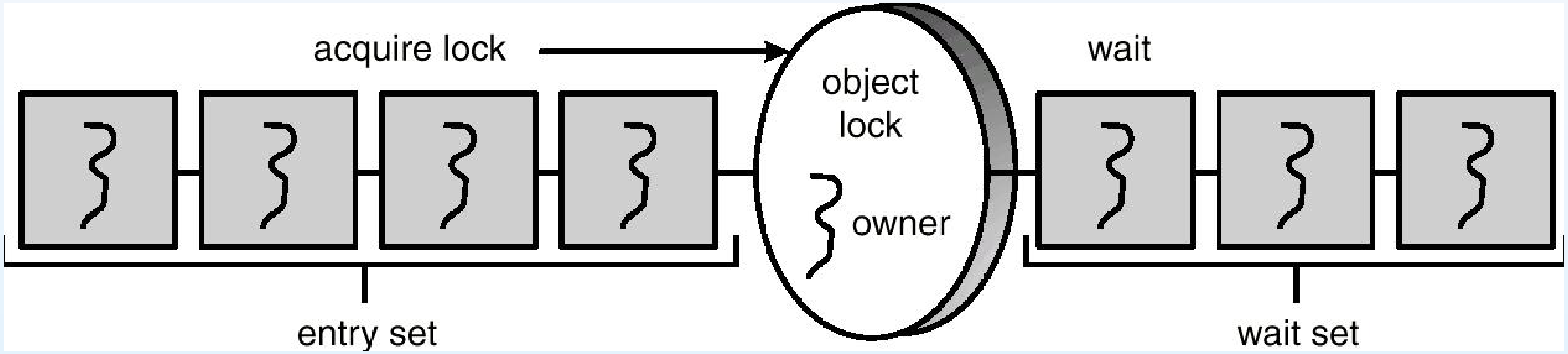
problems

- Assume the producer owns the lock and the buffer is full;
- The producer still keeps the lock;
- When the consumer has an opportunity to be scheduled to run, the consumer will be blocked and is placed in the *entry set* for the object's lock.
- Then
  - The producer is waiting for the consumer to free space in the buffer;
  - The consumer is blocked waiting for the producer to release the lock.
- So, Using ***either busy waiting or yielding*** could potentially lead to a deadlock.

The wait() Method

- When a thread calls wait(), the following occurs:
  - the thread ***releases*** the object lock.
  - thread state is set to ***blocked***.
  - thread is placed in the ***wait set***.

Entry and Wait Sets



1、own the lock

2、blocked or runnable

1、release the lock

2、blocked

The notify() Method

- When a thread calls notify(), the following occurs:
  - selects ***an arbitrary thread*** *T* from the ***wait set***.
  - moves T to the ***entry set***.
  - sets T to ***Runnable.***

- *T* can now compete for the object's lock again.

enter() with wait/notify Methods

```
public synchronized void enter(Object item) {
    while (count == BUFFER_SIZE)
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    notify();
}
```

remove() with wait/notify Methods

```
public synchronized Object remove() {
    Object  item;
    while (count == 0)
        try {
            wait();
        }
        catch (InterruptedException e) { }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    notify();
    return item;
}
```

problems

- notify() selects an arbitrary thread from the wait set.
  - This may not be the thread that you want to be selected.
- Java does not allow you to specify the thread to be selected.
- Consider the case where there are multiple threads in the wait set ***and*** more than one condition for which to wait. (为完成一件事情需要满足多个条件)
  - It is possible that a thread whose condition is still unmet may be the thread that receives the notification.
  - Then this thread will be blocked again;
  - For the worst case, the threads in the wait set were notified in a ***bad*** sequence, and they wait for other unmet conditions and then all of them would be blocked again.
- Can also leads to a deadlock;

Multiple Notifications

- notifyAll() removes ALL threads from the ***wait set*** and places them in the ***entry set***. This allows the threads to decide among themselves who should proceed next.

- notifyAll() is a conservative strategy that works best when multiple threads may be in the wait set.

enter() with wait/notifyall Methods

```
public synchronized void enter(Object item) {
    while (count == BUFFER_SIZE)
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    notifyall();
}
```

remove() with wait/notify Methods

```
public synchronized Object remove() {
    Object  item;
    while (count == 0)
        try {
            wait();
        }
        catch (InterruptedException e) { }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    notifyall();
    return item;
}
```

- No deadlock occurs, but low efficient

Reader Methods with Java Synchronization

```
public class Database {
   public Database() {
      readerCount = 0;
      dbReading = false;
      dbWriting = false;
   }
      public synchronized int startRead() { /* see next slides */ }
      public synchronized int endRead()  { /* see next slides */ }
      public synchronized void startWrite() { /* see next slides */ }
      public synchronized void endWrite()  { /* see next slides */ }

      private int readerCount;
      private boolean dbReading;
      private boolean dbWriting;
}
```

startRead() Method

```
public synchronized int startRead() {
    while (dbWriting == true) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
        ++readerCount;
        if (readerCount == 1)
            dbReading = true;
        return readerCount;
}
```

endRead() Method

```
public synchronized int endRead() {
    --readerCount
    if (readerCount == 0)
      db.notifyAll();
    return readerCount;
  }
```

Writer Methods

```
public void startWrite() {
      while (dbReading == true || dbWriting == true)
            try {
                  wait();
            }
            catch (InterruptedException e) { }
            dbWriting = true;
}

public void endWrite() {
      dbWriting = false;
      notifyAll();

}
```

Block Synchronization

- Blocks of code – rather than entire methods – may be declared as synchronized.
- This yields a lock scope that is typically smaller than a synchronized method.

Block Synchronization (cont)

```
Object mutexLock = new Object();
. . .
public void someMethod() {
    // non-critical section
    synchronized(mutexLock) {
            // critical  section
    }
    // non-critical section
}
```

Java Semaphores

- Java does not provide a semaphore, but a basic semaphore can be constructed using Java synchronization mechanism.

Semaphore Class

```
public class Semaphore {
   public Semaphore() {
            value = 0;
    }
   public Semaphore(int v) {
            value = v;
    }
   public synchronized void P() { /* see next slide */ }
     public synchronized void V() { /* see next slide */ }
   private int value;
}
```

P() Operation

```
public synchronized void P() {
    while (value <= 0) {
      try {
        wait();
      }
      catch (InterruptedException e) { }
    }
    value --;
  }
```

V() Operation

```
public synchronized void V() {
    ++value;

    notify();
  }
```

课后复习题

- 思考题
  - concepts
    - race condition，critical resource, critical section、atomic operation,semaphoer,wait() and signal() operation,monitor
  - 如何利用硬件TestAndSet Instruction以及swap Instruction实现临界区的互斥？
  - 给出教材中讨论的三个经典问题、以及The Sleeping-Barber Problem及Cigarette Smoker’s Problem的问题描述，说明进程之间的制约关系，利用信号量及wait、signal操作给出能正确执行的程序；
  - 课件中的例题
- Page 233

  3,4,5,6,8,9,11,13,22

+KTZY3 14

End of Chapter 6