Module 6: Process Synchronization

Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (count == BUFFER.SIZE)
   ; // do nothing

// add an item to the buffer
buffer[in] = item;
in = (in + 1) % BUFFER.SIZE;
++count;
```

Consumer

```
while (count == 0)
  ; // do nothing

// remove an item from the
buffer item = buffer[out];
out = (out + 1) % BUFFER.SIZE;
--count;
```

Race Condition

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with “count = 5” initially:

  T0: producer execute register1 = count {register1 = 5}
  T1: producer execute register1 = register1 + 1 {register1 = 6}
  T2: consumer execute register2 = count {register2 = 5}
  T3: consumer execute register2 = register2 - 1 {register2 = 4}
  T4: producer execute count = register1 {count = 6 }
  T5: consumer execute count = register2 {count = 4}

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Structure of a Typical Process

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

```
while (true) {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);
        critical section
        flag[i] = false;
        remainder section
}
```

Solution to Critical-Section Problem Using Locks

```
while (true) {
        acquire lock
                critical section
        release lock
                remainder section
}
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Data Structure for Hardware Solutions

```
public class HardwareData
{
    private boolean value = false;

    public HardwareData(boolean value) {
       this.value = value;
    }

    public boolean get() {
       return value;
    }

    public void set(boolean newValue) {
       value = newValue;
    }

    public boolean getAndSet(boolean newValue) {
       boolean oldValue = this.get();
       this.set(newValue);

       return oldValue;
    }

   public void swap(HardwareData other) {
      boolean temp = this.get();

      this.set(other.get());
      other.set(temp);
    }
}
```

Solution using GetAndSet Instruction

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
    while (lock.getAndSet(true))
       Thread.yield();

    // critical section
    lock.set(false);
    // remainder section
}
```

Solution using Swap Instruction

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

// each thread has a local copy of key
HardwareData key = new HardwareData(true);

while (true) {
    key.set(true);

    do {
       lock.swap(key);
    }
    while (key.get() == true);

    // critical section
    lock.set(false);
    // remainder section
}
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify **S: acquire()** and **release()**
  - Originally called **P()** and **V()**
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
acquire() {
    while value <= 0
       ; // no-op
    value--;
}

    release() {
        value++;
    }
```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks

```
Semaphore sem = new Semaphore(1);

sem.acquire();

   // critical section

sem.release();

   // remainder section
```

Java Example Using Semaphores

```
public class Worker implements Runnable
{
    private Semaphore sem;

    public Worker(Semaphore sem) {
       this.sem = sem;
    }

    public void run() {
       while (true) {
          sem.acquire();
          criticalSection();
          sem.release();
          remainderSection();
       }
    }
}
```

Java Example Using Semaphores

```
public class SemaphoreFactory
{
    public static void main(String args[]) {
       Semaphore sem = new Semaphore(1);
       Thread[] bees = new Thread[5];

       for (int i = 0; i < 5; i++)
          bees[i] = new Thread(new Worker(sem));
       for (int i = 0; i < 5; i++)
          bees[i].start();
    }
}
```

Semaphore Implementation

- Must guarantee that no two processes can execute **acquire ()** and **release ()** on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of **acquire()**:

```
acquire(){
    value--;
    if (value < 0) {
        add this process to list
        block;
    }
}
```

- Implementation of **release()**:

```
release(){
    value++;
    if (value <= 0) {
        remove a process P from list
        wakeup(P);
    }
}
```

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
      P0                P1

S.acquire();      Q.acquire();
Q.acquire();      S.acquire();
      .                 .
      .                 .
      .                 .
S.release();      Q.release();
Q.release();      S.release();
```

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

Bounded-Buffer Problem

```
public class BoundedBuffer<E> implements Buffer<E>
{
    private static final int BUFFER_SIZE = 5;
    private E[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer() {
       // buffer is initially empty
       in = 0;
       out = 0;
       mutex = new Semaphore(1);
       empty = new Semaphore(BUFFER_SIZE);
       full = new Semaphore(0);

       buffer = (E[]) new Object[BUFFER_SIZE];

    }

    public void insert(E item) {
       // Figure 6.10
    }

    public E remove() {
       // Figure 6.11
    }
}
```

Bounded-Buffer insert()

```
// Producers call this method
public void insert(E item) {
    empty.acquire();
    mutex.acquire();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    mutex.release();
    full.release();
}
```

Figure 6.10 The insert() method.

Bounded-buffer remove()

```
// Consumers call this method
public E remove() {
    E item;

    full.acquire();
    mutex.acquire();

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.release();
    empty.release();

    return item;
}
```

Bounded-buffer producer

```
import java.util.Date;

public class Producer implements Runnable
{
   private Buffer<Date> buffer;

   public Producer(Buffer<Date> buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // produce an item & enter it into the buffer
         message = new Date();
         buffer.insert(message);
      }
   }
}
```

Bounded-buffer consumer

```
import java.util.Date;

public class Consumer implements Runnable
{
   private Buffer<Date> buffer;

   public Consumer(Buffer<Date> buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // consume an item from the buffer
         message = (Date)buffer.remove();
      }
   }
}
```

Bounded-buffer factory

```java
import java.util.Date;

public class Factory
{
    public static void main(String args[]) {
       Buffer<Date> buffer = new BoundedBuffer<Date>();

       // Create the producer and consumer threads
       Thread producer = new Thread(new Producer(buffer));
       Thread consumer = new Thread(new Consumer(buffer));

       producer.start();
       consumer.start();
    }
}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore db initialized to 1
  - Integer readerCount initialized to 0

Readers-Writers Problem

- Interface for read-write locks

```
public interface ReadWriteLock
{
   public void acquireReadLock();
   public void acquireWriteLock();
   public void releaseReadLock();
   public void releaseWriteLock();
}
```

Readers-Writers Problem (Cont.)

- The structure of a writer

```
public class Writer implements Runnable
{
    private ReadWriteLock db;

    public Writer(ReadWriteLock db) {
       this.db = db;
    }

    public void run() {
       while (true) {
          // nap for awhile
          SleepUtilities.nap();

          db.acquireWriteLock();

          // now write to write to the database
          SleepUtilities.nap();

          db.releaseWriteLock();
       }
    }
}
```

Readers-Writers Problem (Cont.)

- The structure of a reader

```
public class Reader implements Runnable
{
    private ReadWriteLock db;

    public Reader(ReadWriteLock db) {
       this.db = db;
    }

    public void run() {
       while (true) {
          // nap for awhile
          SleepUtilities.nap();

          db.acquireReadLock();

          // now read from the database
          SleepUtilities.nap();

          db.releaseReadLock();
       }
    }
}
```

Readers-Writers Problem (Cont.)

- The database

```
public class Database implements ReadWriteLock
{
   private int readerCount;
   private Semaphore mutex;
   private Semaphore db;

   public Database() {
      readerCount = 0;
      mutex = new Semaphore(1);
      db = new Semaphore(1);
   }

   public void acquireReadLock() {
      // Figure 6.19
   }

   public void releaseReadLock() {
      // Figure 6.19
   }

   public void acquireWriteLock() {
      // Figure 6.20
   }

   public void releaseWriteLock() {
      // Figure 6.20
   }
}
```

Readers-Writers Problem (Cont.)

- Reader methods

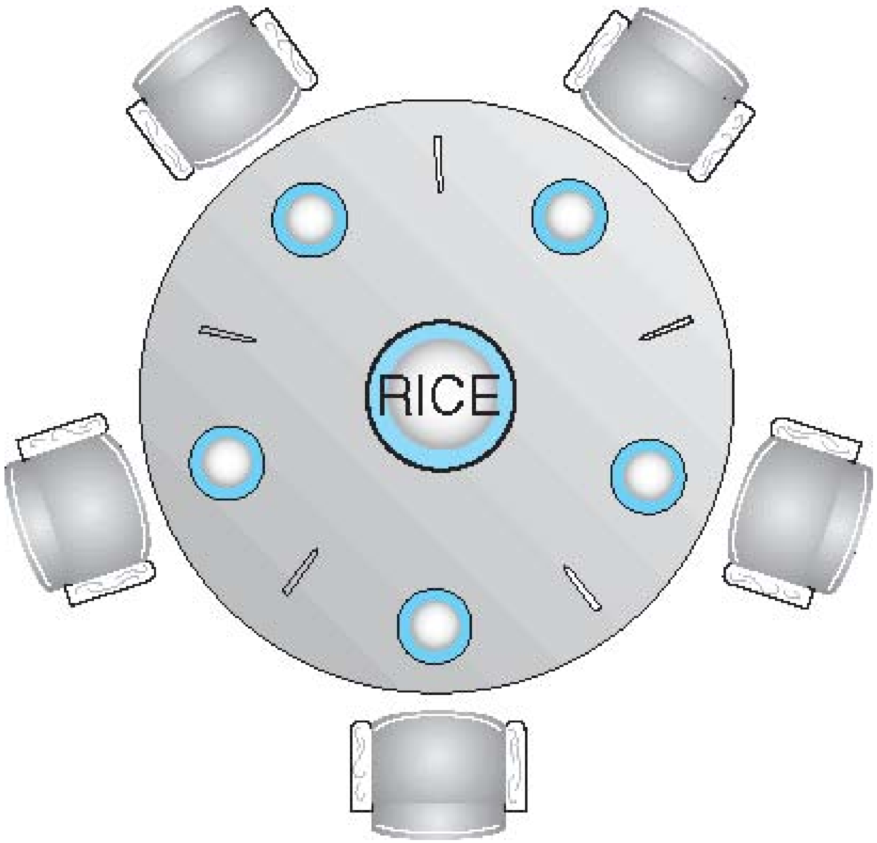
```
public void acquireReadLock() {
    mutex.acquire();

   /**
    * The first reader indicates that
    * the database is being read.
    */
    ++readerCount;
    if (readerCount == 1)
       db.acquire();

    mutex.release();
}

public void releaseReadLock() {
    mutex.acquire();

   /**
    * The last reader indicates that
    * the database is no longer being read.
    */
    --readerCount;
    if (readerCount == 0)
       db.release();

    mutex.release();
}
```

Readers-Writers Problem (Cont.)

- Writer methods

```
public void acquireWriteLock() {
   db.acquire();
}

public void releaseWriteLock() {
   db.release();
}
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopStick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
while (true) {
    // get left chopstick
    chopStick[i].acquire();
    // get right chopstick
    chopStick[(i + 1) % 5].acquire();

    eating();

    // return left chopstick
    chopStick[i].release();
    // return right chopstick
    chopStick[(i + 1) % 5].release();

    thinking();
}
```

Problems with Semaphores

- Correct use of semaphore operations:
  - Correct → mutex.acquire() …. mutex.release()
  - Incorrect → mutex.acquire () or mutex.release() (or both)
  - Omitting either mutex.acquire() or mutex.release()

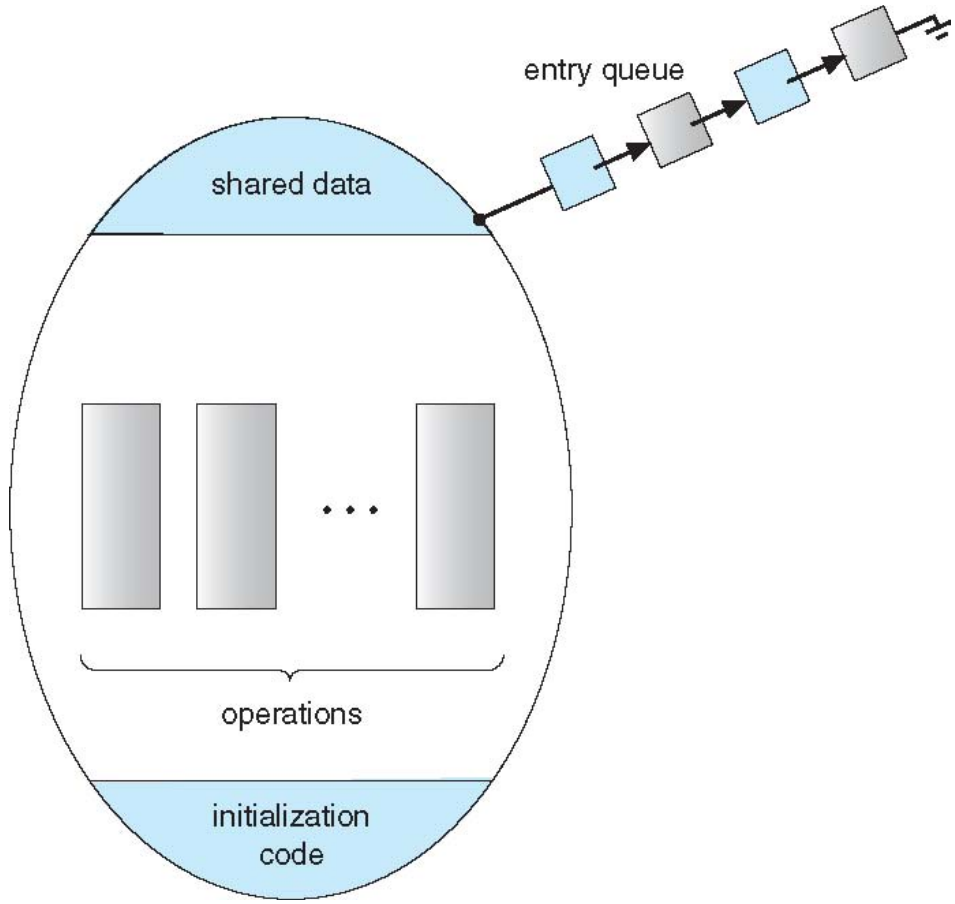
Monitors

- 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

Syntax of a Monitor

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

             .
             .
             .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

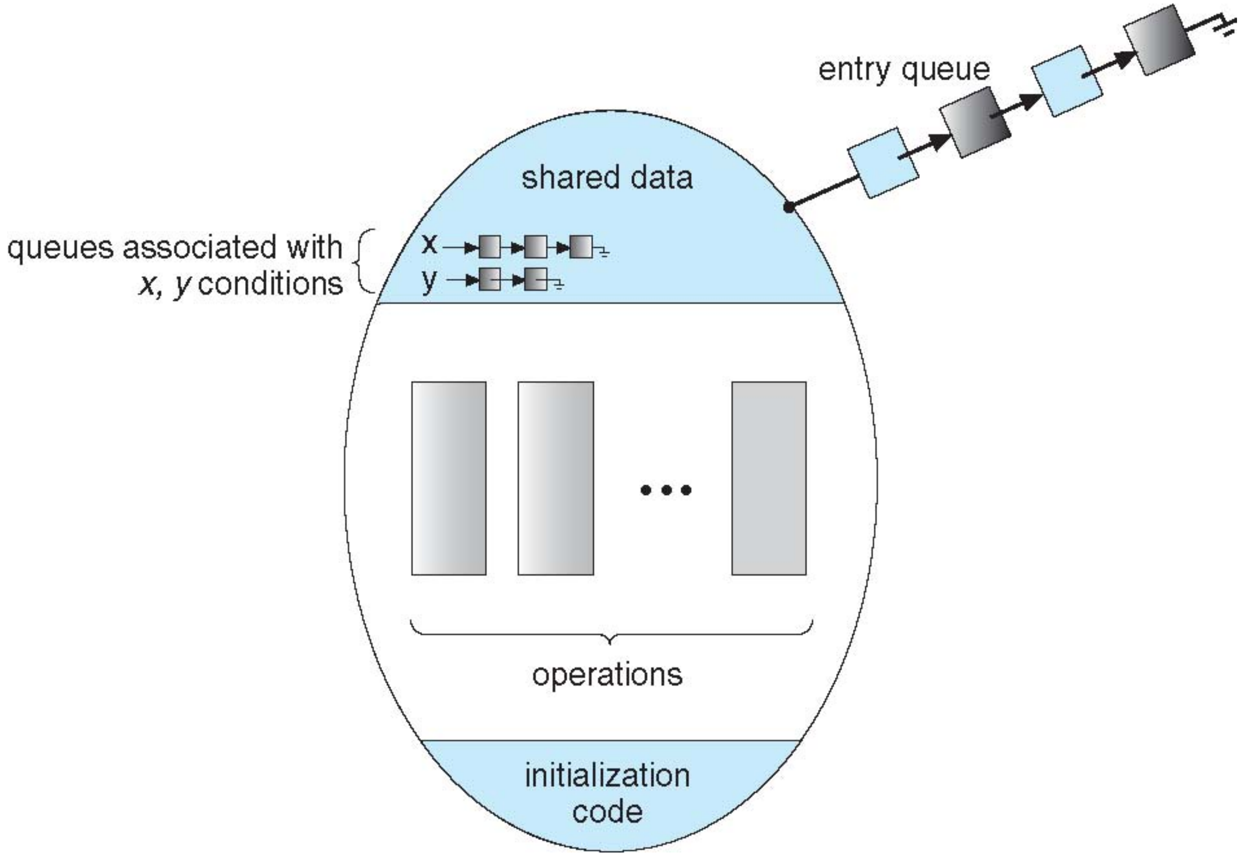
Schematic view of a Monitor



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 ()**

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum State {THINKING, HUNGRY, EATING};
   State[] states = new State[5];
   Condition[] self = new Condition[5];

   public DiningPhilosophers {
      for (int i = 0; i < 5; i++)
         state[i] = State.THINKING;
   }

   public void takeForks(int i) {
      state[i] = State.HUNGRY;
      test(i);
      if (state[i] != State.EATING)
         self[i].wait;
   }

   public void returnForks(int i) {
      state[i] = State.THINKING;
      // test left and right neighbors
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   private void test(int i) {
      if ( (state[(i + 4) % 5] != State.EATING) &&
         (state[i] == State.HUNGRY) &&
         (state[(i + 1) % 5] != State.EATING) ) {
            state[i] = State.EATING;
            self[i].signal;
      }
   }
}
```

Solution to Dining Philosophers (Cont.)

- Each philosopher *I* invokes the operations **takeForks(i)** and **returnForks(i)** in the following sequence:

  dp.takeForks (i)

  EAT

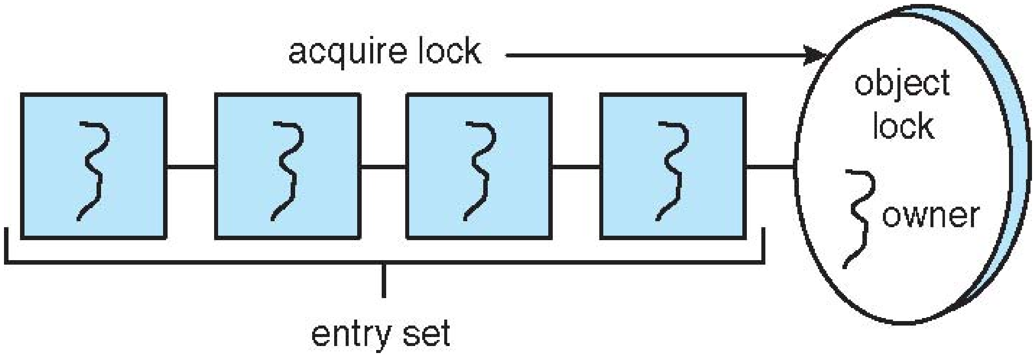
  dp.returnForks (i)

Java Synchronization

- Java provides synchronization at the language-level.
- Each Java object has an associated lock.
- This lock is acquired by invoking a **synchronized** method.
- This lock is released when exiting the **synchronized** method.
- Threads waiting to acquire the object lock are placed in the **entry set** for the object lock.

Java Synchronization

- Each object has an associated **entry set**.



Java Synchronization

- Synchronized insert() and remove() methods – Incorrect!

```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE)
      Thread.yield();

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;
}


// Consumers call this method
public synchronized E remove() {
    E item;

    while (count == 0)
      Thread.yield();

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    return item;
}
```

Java Synchronization wait/notify()

- When a thread invokes **wait():**

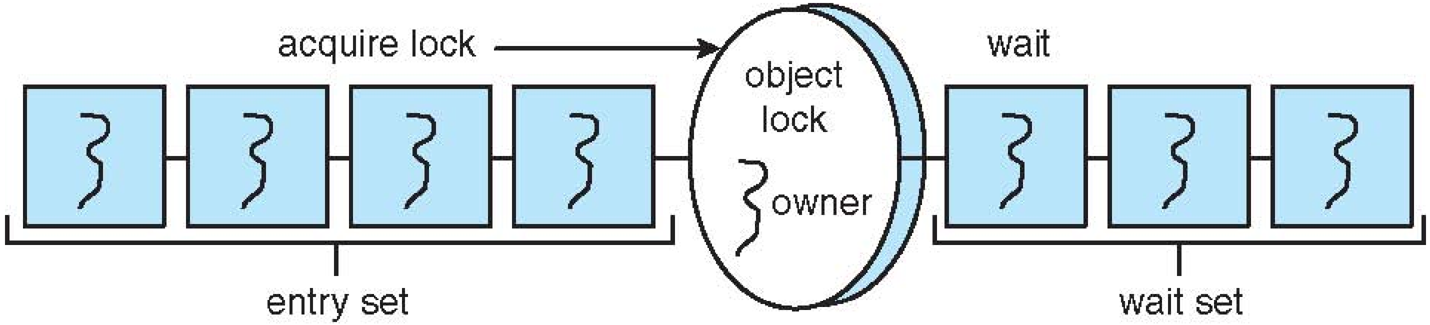
  1. The thread releases the object lock;
  2. The state of the thread is set to Blocked;
  3. The thread is placed in the **wait set** for the object.

- When a thread invokes **notify()**:

  1. An arbitrary thread T from the wait set is selected;
  2. T is moved from the wait to the entry set;
  3. The state of T is set to Runnable.

Java Synchronization

- Entry and wait sets



Java Synchronization – wait/notify

- Synchronized insert() method – Correct!

```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;

    notify();
}
```

Java Synchronization – wait/notify

- Synchronized remove() method – Correct!

```
// Consumers call this method
public synchronized E remove() {
    E item;

    while (count == 0) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    notify();

    return item;
}
```

Java Synchronization - Bounded Buffer

```
public class BoundedBuffer<E> implements Buffer<E>
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private E[] buffer;

    public BoundedBuffer() {
      // buffer is initially empty
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
    }

    public synchronized void insert(E item) {
       // Figure 6.29
    }

    public synchronized E remove() {
       // Figure 6.29
    }
}
```

Java Synchronization

- The call to **notify()** selects an aribitrary thread from the wait set. It is possible the selected thread is in fact not waiting upon the condition for which it was notified.
- The call **notifyAll()** selects all threads in the wait set and moves them to the entry set.
- In general, **notifyAll()** is a more conservative strategy than **notify()**.

Java Synchronization

```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public synchronized void doWork(int myNumber) {
    while (turn != myNumber) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    // Do some work for awhile . . .

   /**
    * Finished working. Now indicate to the
    * next waiting thread that it is their
    * turn to do some work.
    */
    turn = (turn + 1) % 5;

    notify();
}
```

notify() may not notify the correct thread!

Java Synchronization - Readers-Writers

```
public class Database implements ReadWriteLock
{
    private int readerCount;
    private boolean dbWriting;

    public Database() {
       readerCount = 0;
       dbWriting = false;
    }

    public synchronized void acquireReadLock() {
       // Figure 6.34
    }

    public synchronized void releaseReadLock() {
       // Figure 6.34
    }

    public synchronized void acquireWriteLock() {
       // Figure 6.35
    }

    public synchronized void releaseWriteLock() {
       // Figure 6.35
    }
}
```

Java Synchronization - Readers-Writers

- Methods called by readers

```
public synchronized void acquireReadLock() {
   while (dbWriting == true) {
      try {
         wait();
      }
      catch(InterruptedException e) { }
   }

   ++readerCount;
}

public synchronized void releaseReadLock() {
   --readerCount;

   /**
    * The last reader indicates that
    * the database is no longer being read.
    */
   if (readerCount == 0)
      notify();
}
```

Java Synchronization - Readers-Writers

- Methods called by writers

```
public synchronized void acquireWriteLock() {
   while (readerCount > 0 || dbWriting == true) {
      try {
         wait();
      }
      catch(InterruptedException e) { }
   }

  /**
   * Once there are no readers or a writer,
   * indicate that the database is being written.
   */
   dbWriting = true;
}

public synchronized void releaseWriteLock() {
   dbWriting = false;

   notifyAll();
}
```

Java Synchronization

- Rather than synchronizing an entire method, **Block synchronization** allows blocks of code to be declared as synchronized

```
Object mutexLock = new Object();
. . .
public void someMethod() {
    nonCriticalSection();

    synchronized(mutexLock) {
      criticalSection();
    }

    remainderSection();
}
```

Java Synchronization

- Block synchronization using wait()/notify()

```
Object mutexLock = new Object();
. . .
synchronized(mutexLock) {
    try {
       mutexLock.wait();
    }
    catch (InterruptedException ie) { }
}

synchronized(mutexLock) {
    mutexLock.notify();
}
```

Concurrency Features in Java 5

- Prior to Java 5, the only concurrency features in Java were Using synchronized/wait/notify.
- Beginning with Java 5, new features were added to the API:
  - Reentrant Locks
  - Semaphores
  - Condition Variables

Concurrency Features in Java 5

- Reentrant Locks

```
Lock key = new ReentrantLock();

key.lock();
try {
   // critical section
}
finally {
   key.unlock();
}
```

Concurrency Features in Java 5

- Semaphores

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   // critical section
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Concurrency Features in Java 5

- A condition variable is created by first creating a **ReentrantLock** and invoking its **newCondition()** method:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

- Once this is done, it is possible to invoke the **await()** and **signal()** methods

Concurrency Features in Java 5

- doWork() method with condition variables

```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public void doWork(int myNumber) {
   lock.lock();

   try {
      /**
       * If it's not my turn, then wait
       * until I'm signaled
       */
      if (myNumber != turn)
         condVars[myNumber].await();

      // Do some work for awhile . . .

      /**
       * Finished working. Now indicate to the
       * next waiting thread that it is their
       * turn to do some work.
       */

      turn = (turn + 1) % 5;
      condVars[turn].signal();
   }
   catch (InterruptedException ie) { }
   finally {
      lock.unlock();
   }
}
```

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

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- 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

Atomic Transactions

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

Transactional Memory

- **Memory transaction** is a series of read-write operations that are atomic.
- We replace

```
update () {
   acquire();
   /* modify shared data */
   release();
}
```

- With

```
update () {
   atomic {
      /* modify shared data */
   }
}
```

- The **atomic{S}** statement ensures the statements in **S** execute as a transaction.

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) < 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) ≥ 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) < 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) < 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)

End of Chapter 6