Module 7a: Classic Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Shared-memory solution to bounded-butter problem (Chapter 4) has a race condition on the class data **count**.

Race Condition

The Producer calls

```
while (1) {
        while (count == BUFFER_SIZE)
                ; // do nothing
        // produce an item and put in nextProduced
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Race Condition

The Consumer calls

```
while (1) {
        while (count == 0)
                ; // do nothing
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        // consume the item in nextConsumed
}
```

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:

```
S0: producer execute register1 = count   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 = 6}
S2: consumer execute register2 = count   {register2 = 5}
S3: consumer execute register2 = register2 - 1   {register2 = 4}
S4: producer execute count = register1   {count = 6 }
S5: 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

Two-task Solution

- Two tasks, T_0 and T_1 (T_i and T_j)
- Three solutions presented. All implement this MutualExclusion interface:

```
public interface MutualExclusion
{
     public static final int TURN 0 = 0;
     public static final int TURN 1 = 1;

     public abstract void enteringCriticalSection(int turn);
     public asbtract void leavingCriticalSection(int turn);
 }
```

Algorithm Factory class

Used to create two threads and to test each algorithm

```
public class AlgorithmFactory
{
   public static void main(String args[]) {
      MutualExclusion alg = new Algorithm 1();
      Thread first = new Thread( new Worker("Worker 0", 0, alg));
      Thread second = new Thread(new Worker("Worker 1", 1, alg));

      first.start();
      second.start();
   }
}
```

Worker Thread

```
public class Worker implements Runnable
{
      private String name;
      private int id;
      private MutualExclusion mutex;

      public Worker(String name, int id, MutualExclusion mutex) {
        this.name = name;
        this.id = id;
        this.mutex = mutex;
      }
      public void run() {
        while (true) {
                mutex.enteringCriticalSection(id);
                MutualExclusionUtilities.criticalSection(name);
                mutex.leavingCriticalSection(id);
                MutualExclusionUtilities.nonCriticalSection(name);
        }
      }
}
```

Algorithm 1

- Threads share a common integer variable turn
- If turn==i, thread i is allowed to execute
- Does not satisfy progress requirement
  - Why?

Algorithm 1

```
public class Algorithm_1 implements MutualExclusion
{
   private volatile int turn;

   public Algorithm 1() {
    turn = TURN 0;
   }
   public void enteringCriticalSection(int t) {
    while (turn != t)
        Thread.yield();
   }
   public void leavingCriticalSection(int t) {
    turn = 1 - t;
   }
}
```

Algorithm 2

- Add more state information
  - Boolean flags to indicate thread’s interest in entering critical section
- Progress requirement still not met
  - Why?

Algorithm 2

```
public class Algorithm_2 implements MutualExclusion
{
      private volatile boolean flag0, flag1;
      public Algorithm 2() {
        flag0 = false; flag1 = false;
      }
      public void enteringCriticalSection(int t) {
         if (t == 0) {
               flag0 = true;
               while(flag1 == true)
                       Thread.yield();
        }
        else {
                flag1 = true;
               while (flag0 == true)
                       Thread.yield();
        }
      }
      // Continued On Next Slide
```

Algorithm 2 - cont

```
    public void leavingCriticalSection(int t) {
        if (t == 0)
            flag0 = false;
        else
            flag1 = false;
    }
}
```

Algorithm 3

- Combine ideas from 1 and 2
- Does it meet critical section requirements?

Algorithm 3

```
public class Algorithm_3 implements MutualExclusion
{
      private volatile boolean flag0;
      private volatile boolean flag1;
      private volatile int turn;
      public Algorithm_3() {
        flag0 = false;
        flag1 = false;
        turn = TURN_0;
       }
      // Continued on Next Slide
```

Algorithm 3 - enteringCriticalSection

```
public void enteringCriticalSection(int t) {
  int other = 1 - t;
  turn = other;
  if (t == 0) {
        flag0 = true;
        while(flag1 == true && turn == other)
              Thread.yield();
  }
  else {
        flag1 = true;
        while (flag0 == true && turn == other)
              Thread.yield();
  }
}
// Continued on Next Slide
```

Algo. 3 – leavingingCriticalSection()

```
 public void leavingCriticalSection(int t) {
         if (t == 0)
                 flag0 = false;
       else
                 flag1 = false;
 }
}
```

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 data;
    public HardwareData(boolean data) {
        this.data = data;
    }
    public boolean get() {
        return data;
    }
    public void set(boolean data) {
        this.data = data;
    }
    // Continued on Next Slide
```

Data Structure for Hardware Solutions - cont

```
public boolean getAndSet(boolean data) {
        boolean oldValue = this.get();
        this.set(data);
        return oldValue;
}
public void swap(HardwareData other) {
        boolean temp = this.get();
        this.set(other.get());
        other.set(temp);
}
}
```

Thread Using get-and-set Lock

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);
while (true) {
    while (lock.getAndSet(true))
        Thread.yield();
    criticalSection();
    lock.set(false);
    nonCriticalSection();
}
```

Thread 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);
            criticalSection();
            lock.set(false);
            nonCriticalSection();
    }
```

Semaphore

- Synchronization tool that does not require busy waiting *(spin lock)*
- 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(S) {
   while S <= 0
             ; // no-op
   S--;
}
release(S) {
   S++;
}
```

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
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore S; // initialized to 1

acquire(S);
criticalSection();
release(S);
```

Synchronization using Semaphores Implementation - Worker

```
public class Worker implements Runnable
{
    private Semaphore sem;
    private String name;
    public Worker(Semaphore sem, String name) {
        this.sem = sem;
        this.name = name;
    }
    public void run() {
        while (true) {
            sem.acquire();
            MutualExclusionUtilities.criticalSection(name);
            sem.release();
            MutualExclusionUtilities.nonCriticalSection(name);
        }
    }
}
```

Synchronization using Semaphores Implementation - SemaphoreFactory

```
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, "Worker " + (new Integer(i)).toString() ));
            for (int i = 0; i < 5; i++)
                bees[i].start();
     }
}
```

Semaphore Implementation

```
acquire(S){
    value--;
    if (value < 0) {
        add this process to list
        block;
    }
}
release(S){
    value++;
    if (value <= 0) {
        remove a process P from list
        wakeup(P);
    }
}
```

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
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
  - Applications may spend lots of time in critical sections
    - Performance issues addressed throughout this lecture

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

- 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

```
public class BoundedBuffer implements Buffer
{
     private static final int BUFFER SIZE = 5;
     private Object[] buffer;
     private int in, out;
     private Semaphore mutex;
     private Semaphore empty;
     private Semaphore full;

     // Continued on next Slide
```

Bounded Buffer Constructor

```
public BoundedBuffer() {
        // buffer is initially empty
        in = 0;
        out = 0;
        buffer = new Object[BUFFER SIZE];
        mutex = new Semaphore(1);
        empty = new Semaphore(BUFFER SIZE);
        full = new Semaphore(0);
}
public void insert(Object item) { /* next slides */ }

public Object remove() { /* next slides */ }
}
```

Bounded Buffer Problem: insert() Method

```
public void insert(Object item) {
    empty.acquire();
    mutex.acquire();
    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
    mutex.release();
    full.release();
}
```

Bounded Buffer Problem: remove() Method

```
public Object remove() {
    full.acquire();
    mutex.acquire();
    // remove an item from the buffer
    Object item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    mutex.release();
    empty.release();
    return item;
}
```

Bounded Buffer Problem: Producer

```
import java.util.Date;
public class Producer implements Runnable
{
    private Buffer buffer;
    public Producer(Buffer 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 Problem: Consumer

```
import java.util.Date;
public class Consumer implements Runnable
{
    private Buffer buffer;
    public Consumer(Buffer 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 Problem: Factory

```
public class Factory
{
     public static void main(String args[]) {
               Buffer buffer = new BoundedBuffer();
               // now 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: Reader

```
public class Reader implements Runnable
{
    private RWLock db;
    public Reader(RWLock db) {
            this.db = db;
    }
    public void run() {
            while (true) { // nap for awhile
                    db.acquireReadLock();

                    // you now have access to read from the database
                    // read from the database

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

Readers-Writers Problem: Writer

```
public class Writer implements Runnable
{
    private RWLock db;
    public Writer(RWLock db) {
        this.db = db;
    }
    public void run() {
        while (true) {
            db.acquireWriteLock();
            // you have access to write to the database

            // write to the database

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

Readers-Writers Problem: Interface

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

Readers-Writers Problem: Database

```
public class Database implements RWLock
{
    private int readerCount;
    private Semaphore mutex;
    private Semaphore db;
    public Database() {
        readerCount = 0;
        mutex = new Semaphore(1);
        db = new Semaphore(1);
    }
    public int acquireReadLock() { /* next slides */ }
    public int releaseReadLock() {/* next slides */ }
    public void acquireWriteLock() {/* next slides */ }
    public void releaseWriteLock() {/* next slides */ }
}
```

Readers-Writers Problem: Methods called by readers

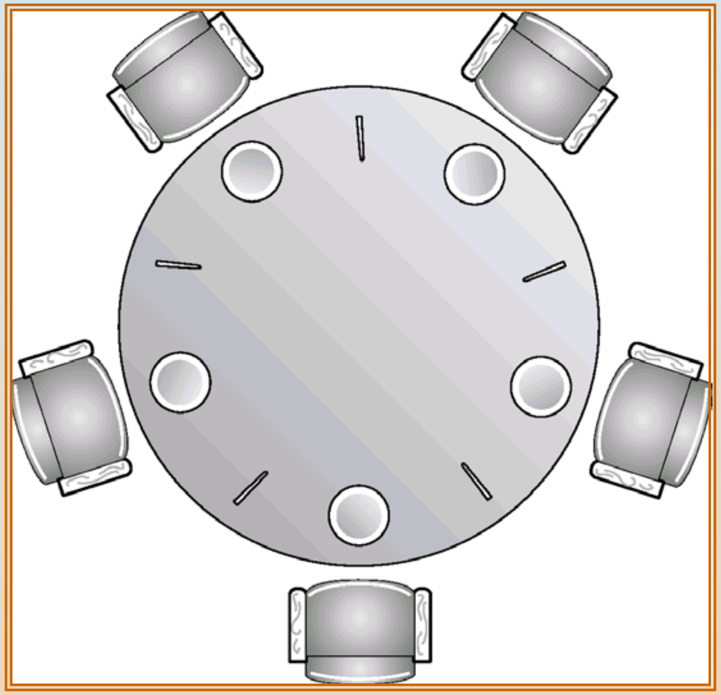
```
public void acquireReadLock() {
      mutex.acquire();
      ++readerCount;
      // if I am the first reader tell all others
      // that the database is being read
      if (readerCount == 1)
            db.acquire();
      mutex.release();
}
public void releaseReadLock() {
      mutex.acquire();
      --readerCount;
      // if I am the last reader tell all others
      // that the database is no longer being read
      if (readerCount == 0)
            db.release();
      mutex.release();
}
```

Readers-Writers Problem: Methods called by writers

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

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

Dining-Philosophers Problem



- Shared data

```
Semaphore chopStick[]  = new Semaphore[5];
```

Dining-Philosophers Problem (Cont.)

- 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();
}
```

Monitors

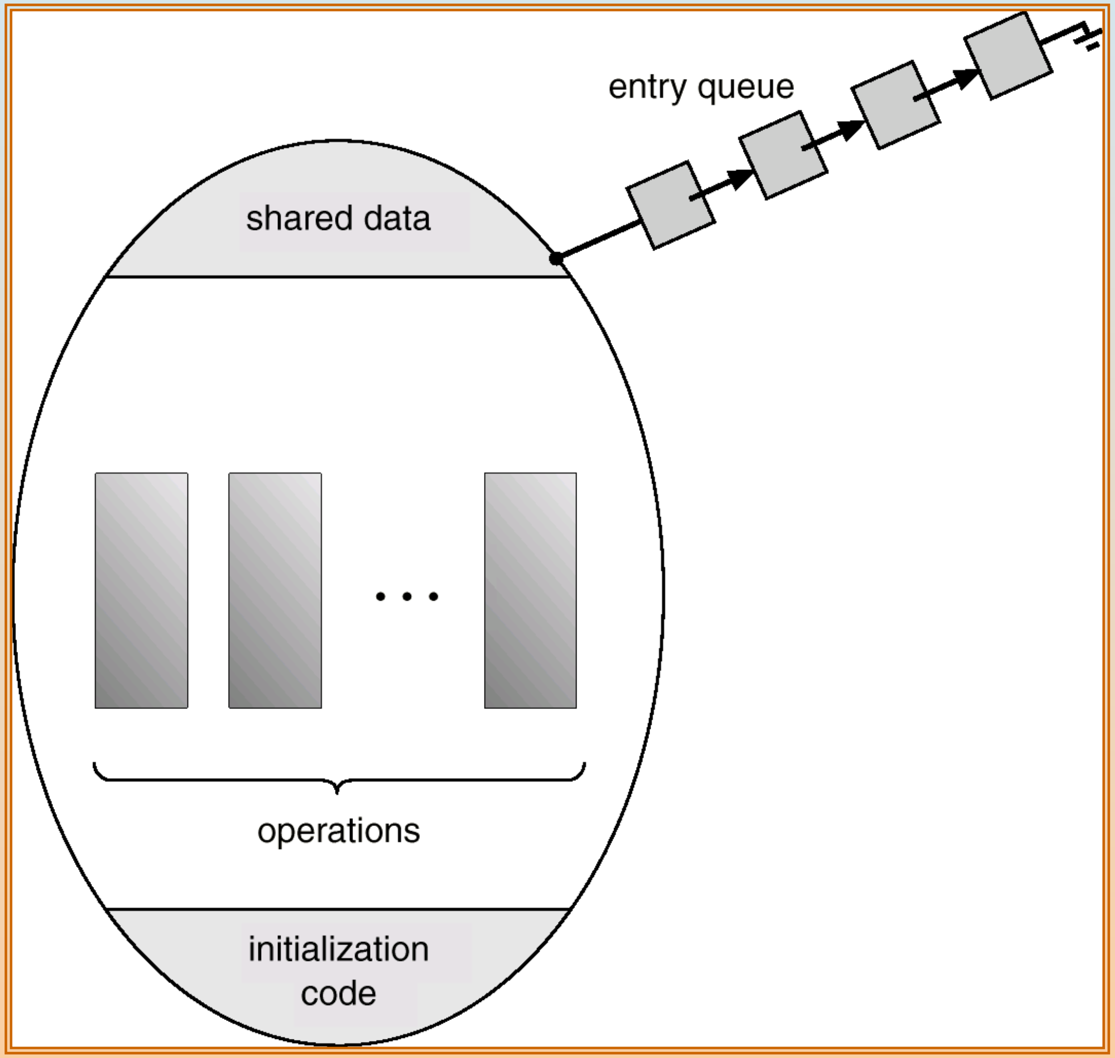
- A monitor is a high-level abstraction that provides thread safety
- Only one thread may be active within the monitor at a time

```
monitor monitor-name
{
    // variable declarations
    public entry p1(…) {
        …
    }
    public entry p2(…) {
        …
    }
}
```

Condition Variables

- condition x, y;
- A thread that invokes x.wait is suspended until another thread invokes x.signal

Monitor with condition variables



Condition Variable Solution to Dining Philosophers

```
monitor DiningPhilosophers {
    int[] state = new int[5];
    static final int THINKING = 0;
    static final int HUNGRY = 1;
    static final int EATING = 2;
    condition[] self = new condition[5];
    public diningPhilosophers {
            for (int i = 0; i < 5; i++)
                    state[i] = THINKING;
     }
    public entry pickUp(int i) {
            state[i] = HUNGRY;
            test(i);
            if (state[i] != EATING)
                    self[i].wait;
    }
    // Continued on Next Slide
```

Solution to Dining Philosophers (cont)

```
public entry putDown(int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
}
private test(int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
              (state[i] == HUNGRY) &&
              (state[(i + 1) % 5] != EATING) ) {
                    state[i] = EATING;
                    self[i].signal;
        }
}
```