Process Synchronisation

Background (1)

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Producer

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

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

Consumer

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

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

Background (2)

- 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:

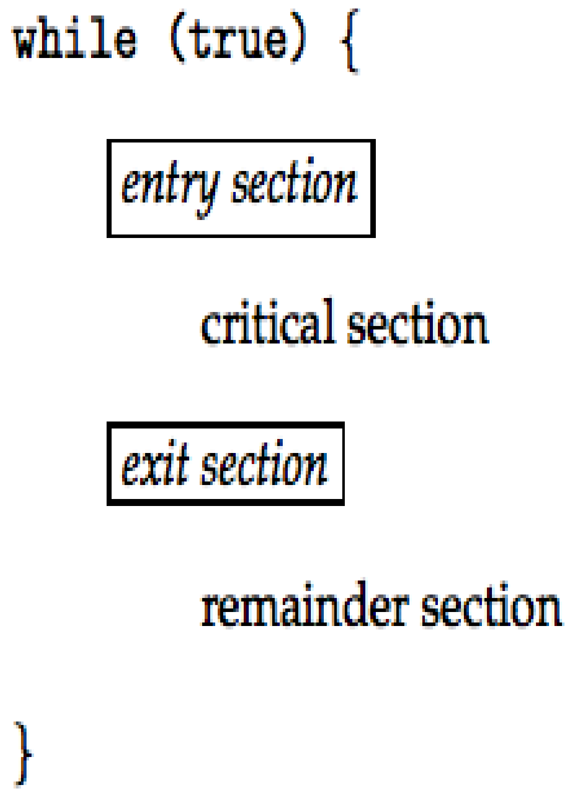
```
S0: producer execute register1 = count   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 = 6}
S2: consumer execute register2 = count   {register2 = 5}
S3: consumer execute register2 = register2 - 1   {register2 = 4}
S4: producer execute count = register1   {count = 6 }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

- Solution: ensure that only one process at a time can manipulate variable count
- Avoid interference between changes

Critical Section Problem

- Critical section: a segment of code in which a process may be changing common variables
  - Only one process is allowed to be executing in its critical section at any moment in time
- Critical section problem: design a protocol for process cooperation
- Requirements for a solution
  - Mutual exclusion
  - Progress
  - Bounded waiting
- No assumption can be made about the relative speed of processes
- Handling critical sections in OS
  - Pre-emptive kernels (real-time programming, more responsive)
    - Linux from 2.6, Solaris, IRIX
  - Non-pre-emptive kernels (free from race conditions)
    - Windows XP, Windows 2000, traditional UNIX kernel, Linux prior 2.6

Process structure

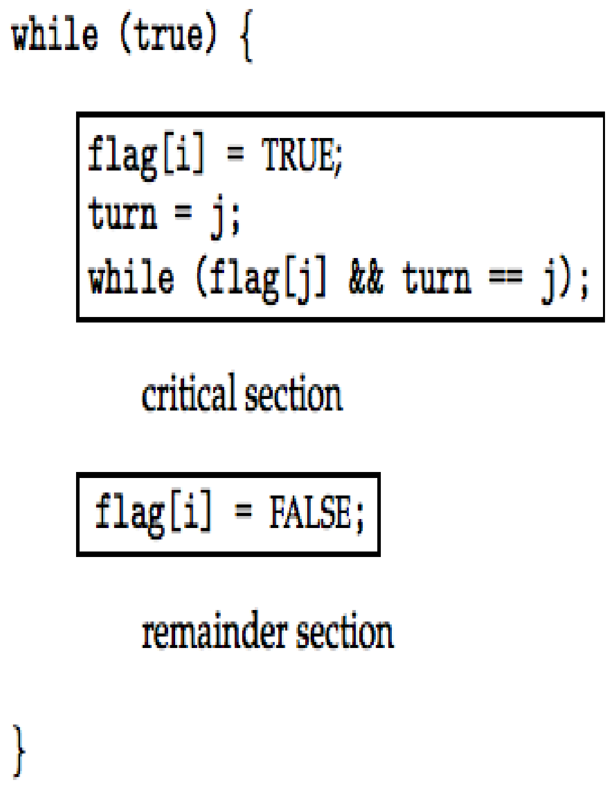


```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

Peterson’s Solution

- Two process solution
  - Mutual exclusion is preserved?
  - The progress requirements is satisfied?
  - The bounded-waiting requirement is met?
- Assumption: LOAD and STORE instructions are atomic, i.e. cannot be interrupted

Process Pi



```
while (true) {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);

            critical section

        flag[i] = FALSE;

            remainder section

}
```

Synchronisation Hardware (1)

- Simple solution: use a lock

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

- Prevent interrupts from occurring while a shared variable is being modified
  - Non-pre-emptive kernels
- Atomic instructions
  - Test and modify a word
  - Swap the contents of two words

```
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);
   }
}
```

Synchronisation Hardware (2)

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

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

    criticalSection();
    lock.set(false);
    remainderSection();
}
```

```
// 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);
    remainderSection();
}
```

Semaphores (1)

- An integer variable only access through two atomic operations

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

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

- Counting semaphore: integer value ranges over an unrestricted domain
- Binary semaphore (mutex lock): integer value ranges between 0 and 1

```
Semaphore S = new Semaphore();

S.acquire();

   // critical section

S.release();

   // remainder section
```

Semaphores (2)

```java
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.remainderSection(name);
       }
    }
}
```

```java
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();
    }
}
```

Semaphores (3)

- Implementation
  - Main disadvantage: busy waiting
    - Spinlock – in multiprocessor systems no context switch required
  - Block and wakeup

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

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

Link field in each PCB

FIFO queue – do not rely on this!

Semaphores (4)

- Must guarantee that no two processes can execute `acquire()` and `release()` on the same semaphore at the same time
- 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

Semaphores (5)

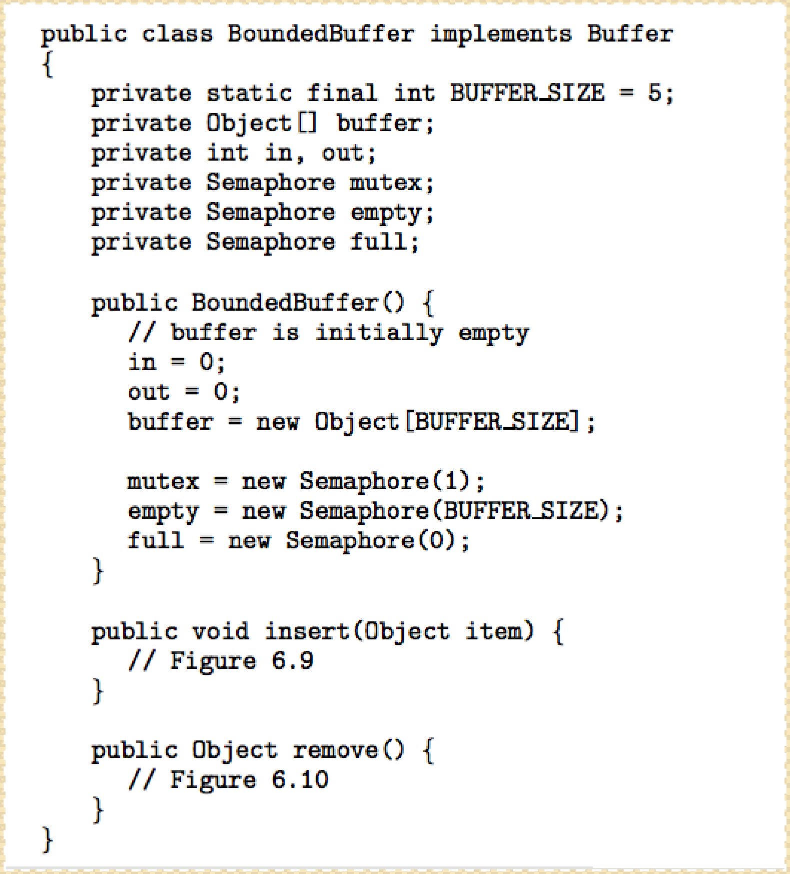
- Deadlock: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
        P0                      P1
    S.acquire();            Q.acquire();
    Q.acquire();            S.acquire();
        .                       .
        .                       .
        .                       .
    S.release();            Q.release();
    Q.release();            S.release();
```

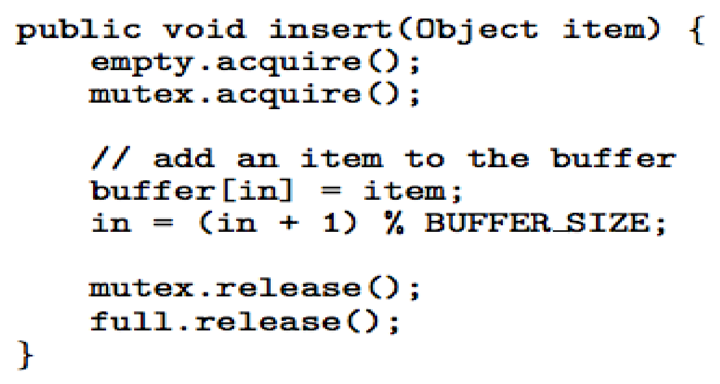
- Starvation: indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
  - LIFO queue

Classic Problems of Synchronisation (1)

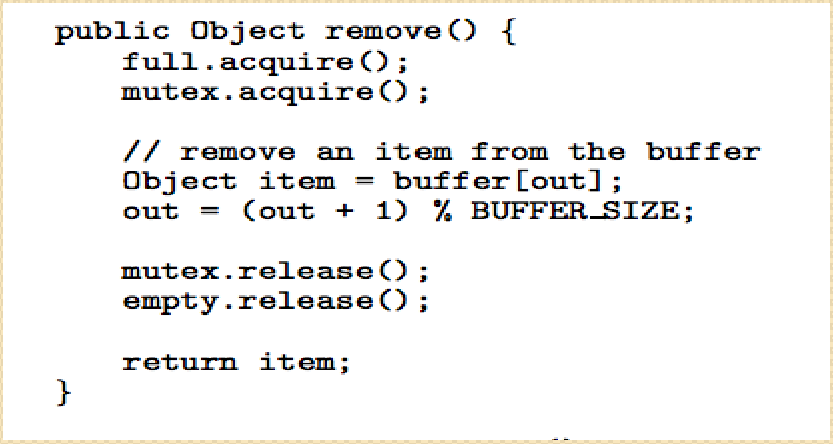
Bounded-Buffer Problem



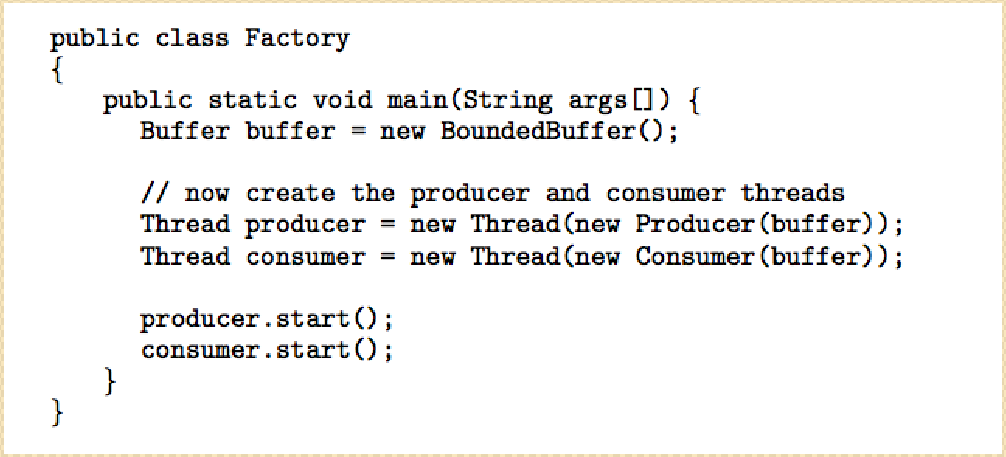
```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private Object[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    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) {
       // Figure 6.9
    }

    public Object remove() {
       // Figure 6.10
    }
}
```



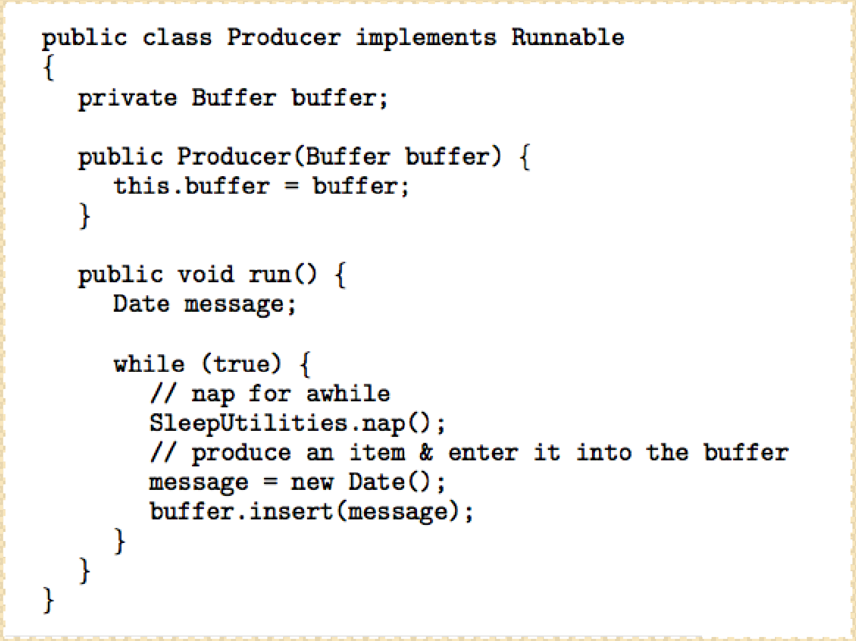
```
public void insert(Object item) {
     empty.acquire();
     mutex.acquire();

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

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



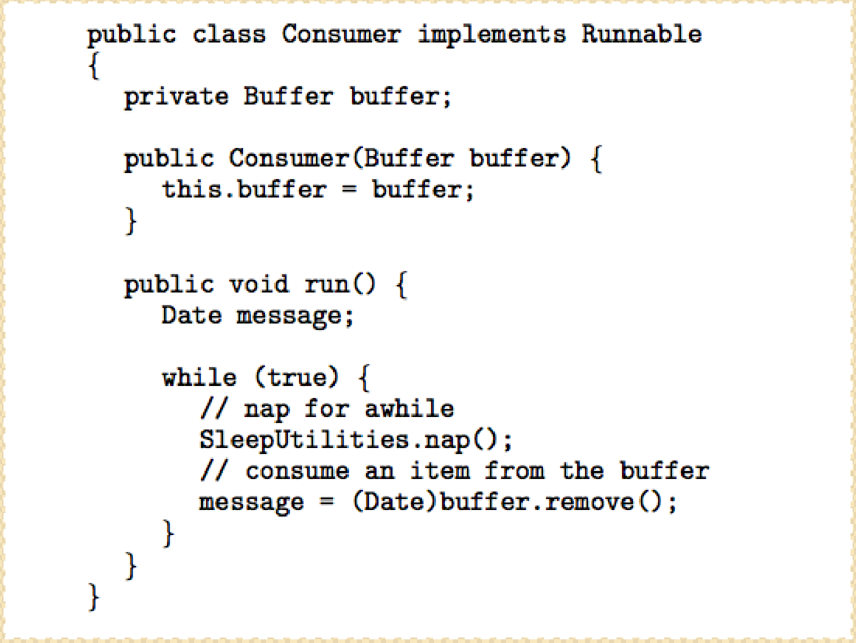
```
public Object remove() {
     full.acquire();
     mutex.acquire();

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

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

     return item;
}
```



```
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();
    }
}
```



```
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);
      }
   }
}
```



```
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();
      }
   }
}
```

Classic Problems of Synchronisation (2)

Classic Problems of Synchronisation (3)

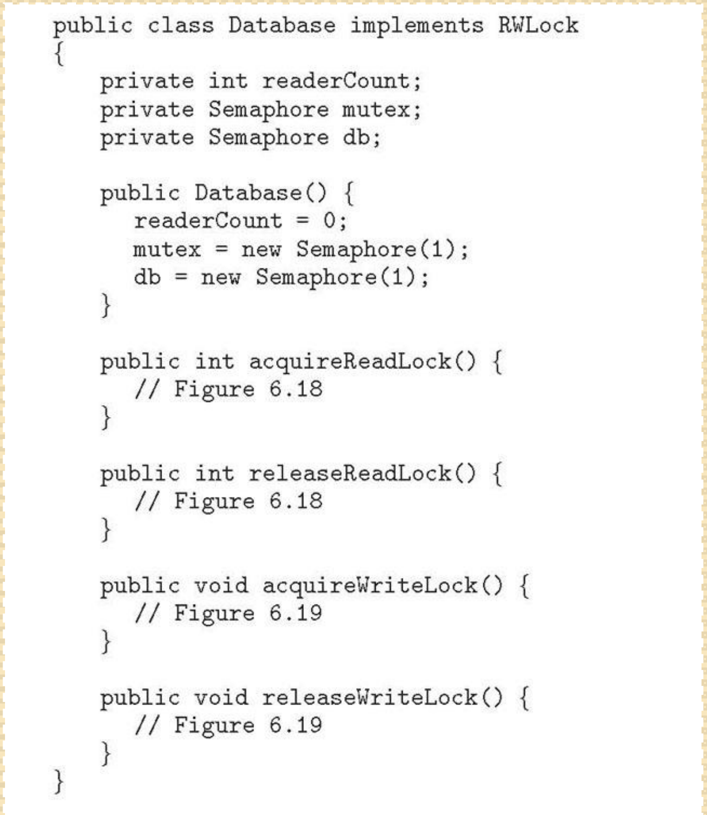
Readers-Writers Problem

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

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

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

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

         db.acquireReadLock();

         // you have access to read from the database
         SleepUtilities.nap();

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

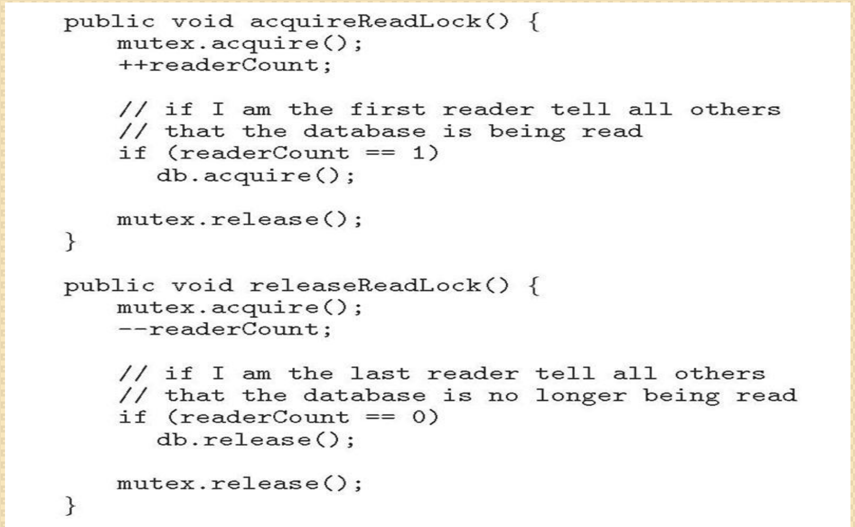
```
public class Writer implements Runnable
{
   private RWLock db;

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

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

         db.acquireWriteLock();

         // you have access to write to the database
         SleepUtilities.nap();

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

Classic Problems of Synchronisation (4)



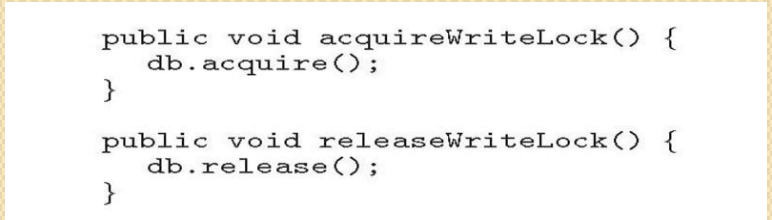
```
public class Database implements RWLock
{
   private int readerCount;
   private Semaphore mutex;
   private Semaphore db;

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

   public int acquireReadLock() {
      // Figure 6.18
   }

   public int releaseReadLock() {
      // Figure 6.18
   }

   public void acquireWriteLock() {
      // Figure 6.19
   }

   public void releaseWriteLock() {
      // Figure 6.19
   }
}
```

Figure 6.17 The database for the readers–writers problem.



```
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();
}
```

Figure 6.18 Methods called by readers.

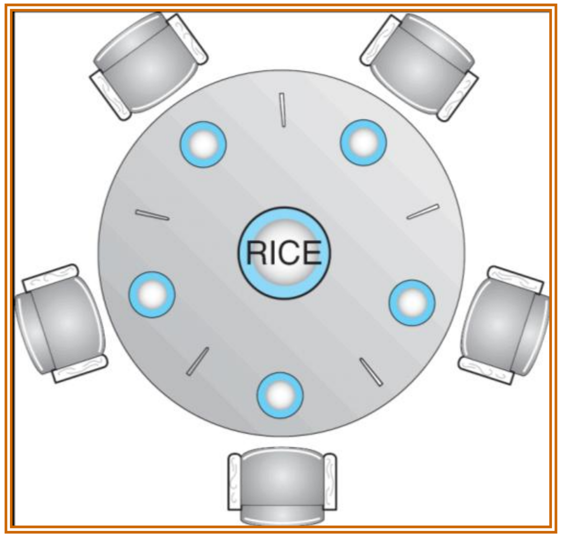


```
public void acquireWriteLock() {
   db.acquire();
}

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

Figure 6.19 Methods called by writers.

Classic Problems of Synchronisation (5)

- Dinning-Philosophers Problem
  - Semaphore chopStick [5] initialized to 1
  - Deadlock!
    - Allow up to 4 philosophers to sit at the table
    - Only allow them to pick a chopstick if both are available (within a critical section)
    - Asymmetry
  - Starvation



```
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();
}
```

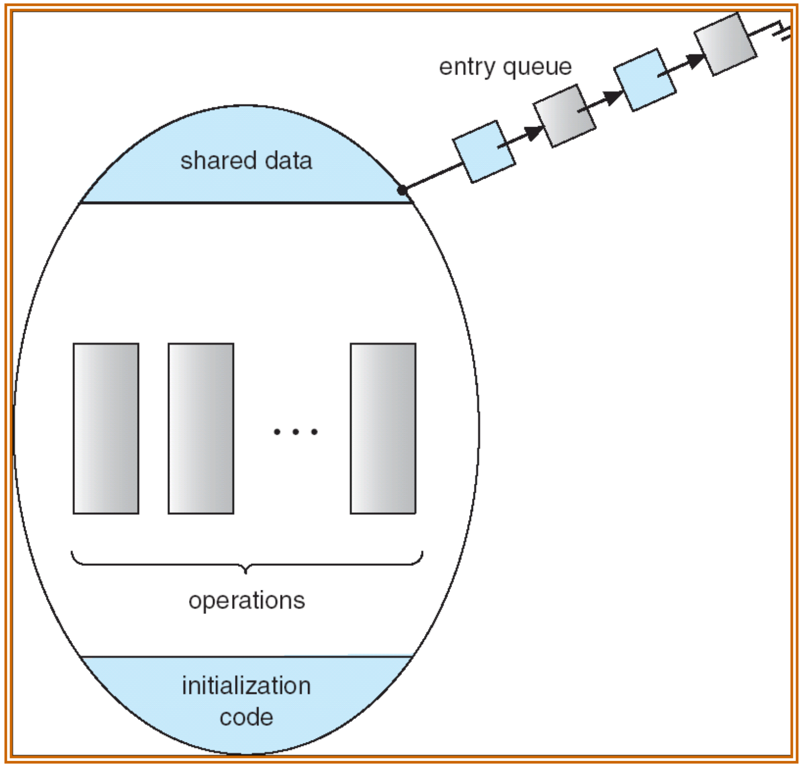
Monitors (1)

- Semaphore problems
  - Misbehaving processes
    - `mutex.release();` … critical section … `mutex.acquire();`
    - `mutex.acquire();` … critical section … `mutex.acquire();`
    - … critical section … `mutex.release();`
    - `mutex.acquire ();` … critical section …
  - Solution: introduce high-level language constructs - monitor

Monitors (2)

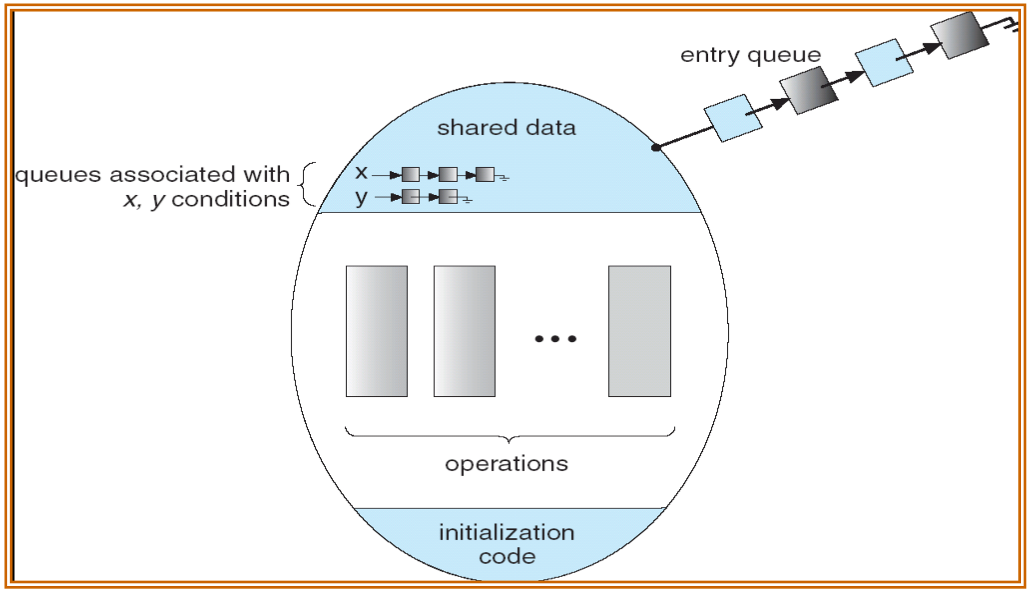
- Only one process may be active within the monitor at a time

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

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

  public P1 ( . . . ) {
     . . .
  }

  public P2 ( . . . ) {
     . . .
  }
          .
          .
          .
  public Pn ( . . . ) {
     . . .
  }
}
```

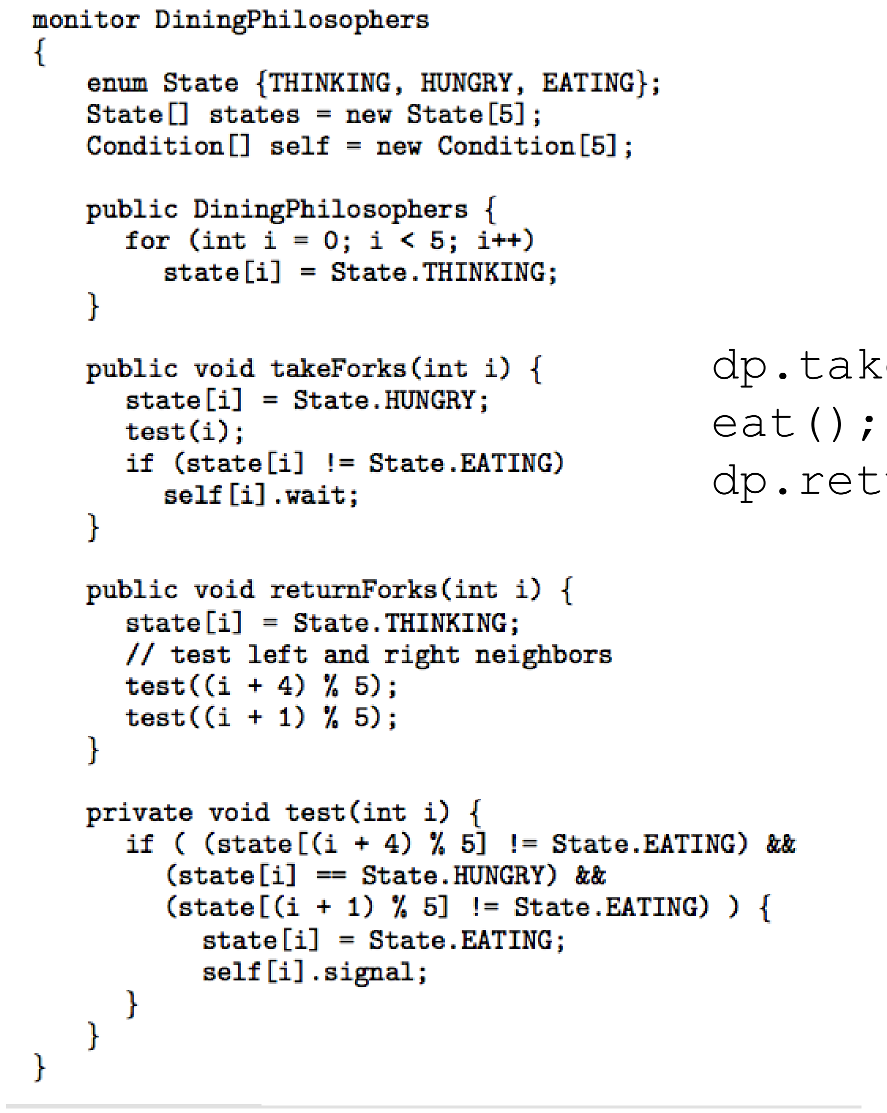


Monitors (3)

- 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 ()
- Signal and wait or signal and continue
  - Signal and immediately exit!



Monitors (4)

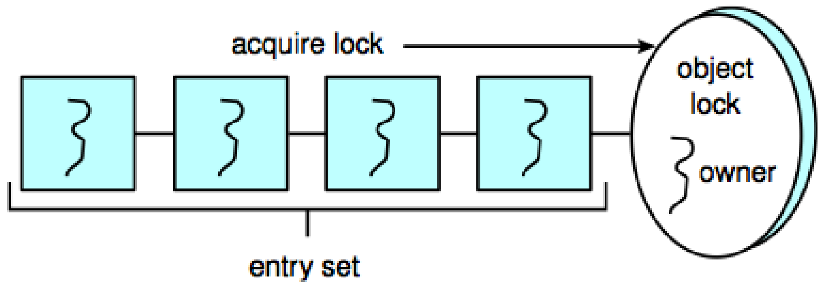


```
monitor DiningPhilosophers
{
   enum State {THINKING, HUNGRY, EATING};
   State[] states = new State[5];
   Condition[] self = new Condition[5];

   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;
      }
   }
}
```

```
dp.takeForks(i);
eat();
dp.returnForks(i);
```

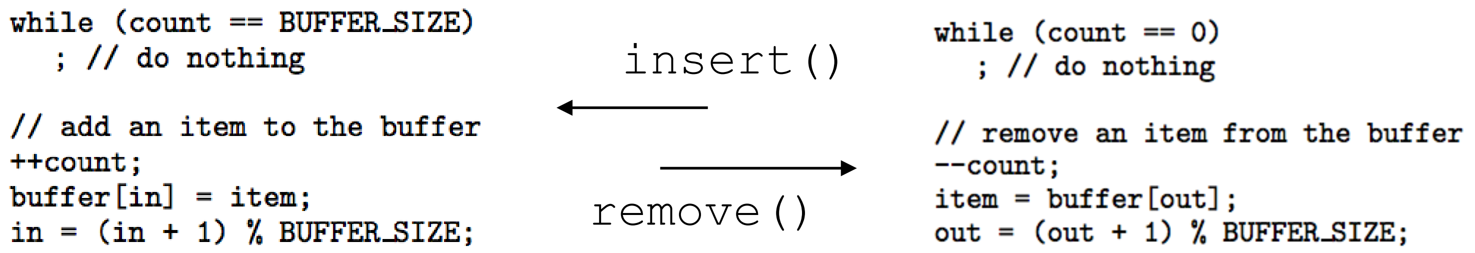
Java Synchronisation (1)

- Thread safe application ensures that data remain consistent even when accessed concurrently by multiple threads
- 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
- The JVM arbitrarily selects a thread from the entry set to be the next owner of the lock



Java Synchronisation (2)

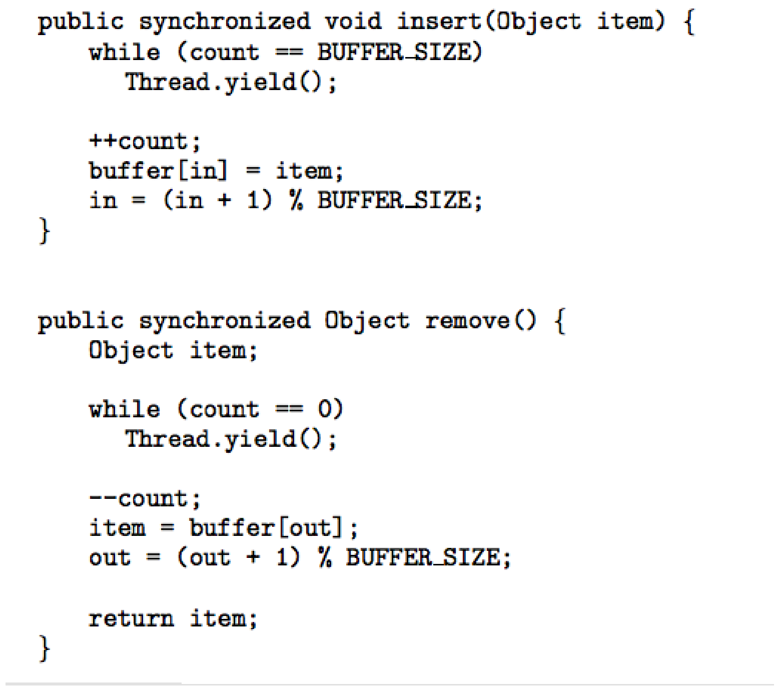
- Bounded buffer



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

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

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

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

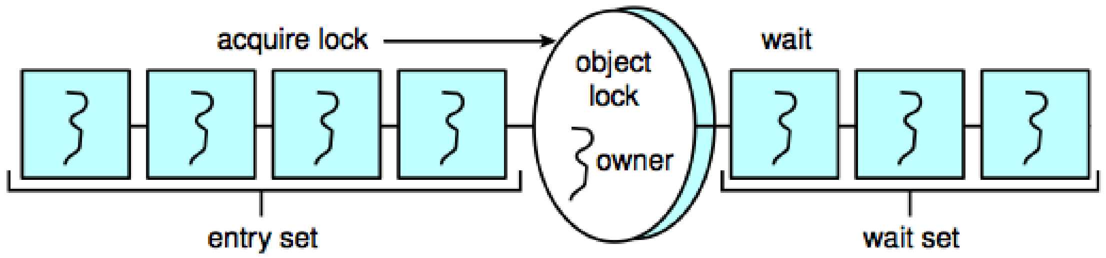
- Problem: race condition on count
- Solution: synchronised methods
  - Remove busy waiting with yield()
    - Livelock: continuously attempt
    - an action that fails
  - What if the buffer is full?
    - Deadlock!!



```
public synchronized void insert(Object item) {
   while (count == BUFFER_SIZE)
     Thread.yield();

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


public synchronized Object remove() {
   Object item;

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

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

   return item;
}
```

Java Synchronisation (3)

- `wait()` and `notify()`
  - Every Java object has an associated wait set for threads
  - Wait: (1) release lock, (2) change thread state to blocked, (3) place thread in object's wait set
  - Notify: (1) pick arbitrary thread from wait set, (2) move picked thread from wait set to entry set, (3) change picked thread state to runnable



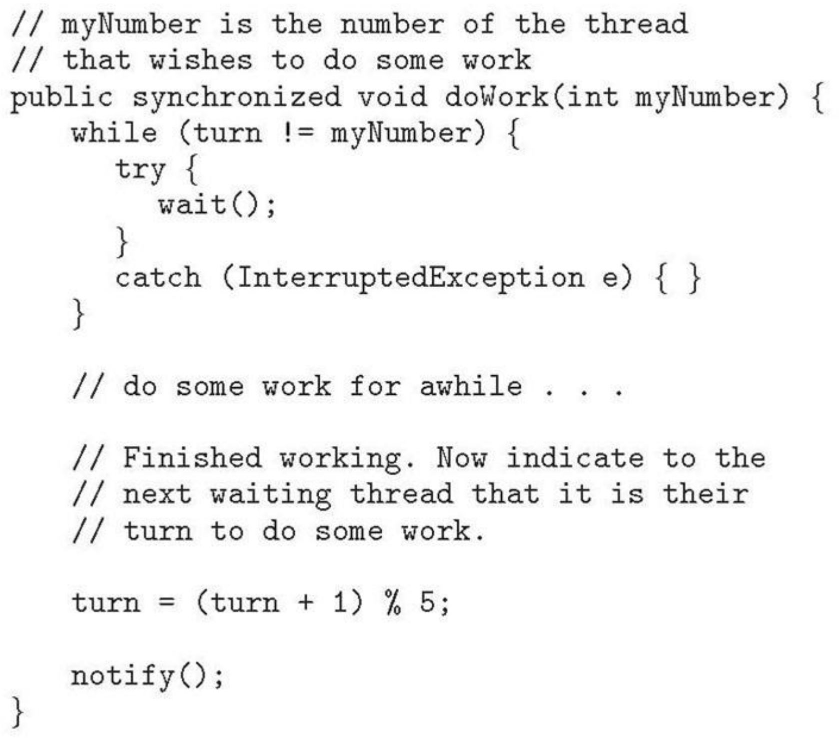
Java Synchronisation (4)

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

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

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

   public synchronized void insert(Object item) {
      // Figure 6.28
   }

   public synchronized Object remove() {
      // Figure 6.28
   }
}
```

```
public synchronized void insert(Object item) {
   while (count == BUFFER_SIZE) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

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

   notify();
}

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;
}
```

Java Synchronisation (5)

- `notifyAll()`: selects all threads in the wait set and moves them to the entry set

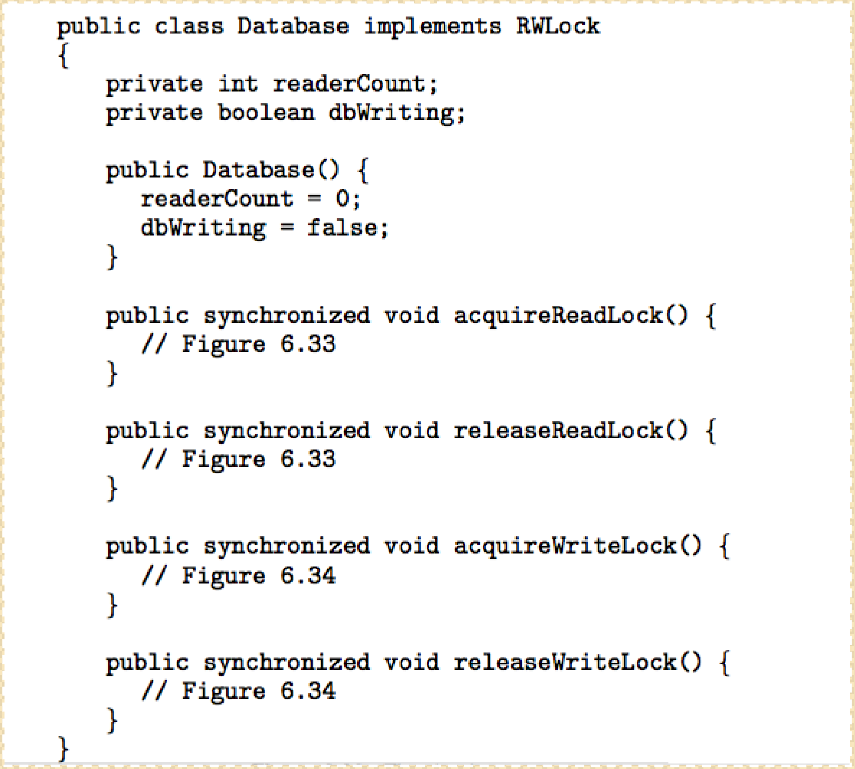


```
// myNumber is the number of the thread
// that wishes to do some work
public synchronized void doWork(int myNumber) {
   while (turn != myNumber) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

   // 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();
}
```

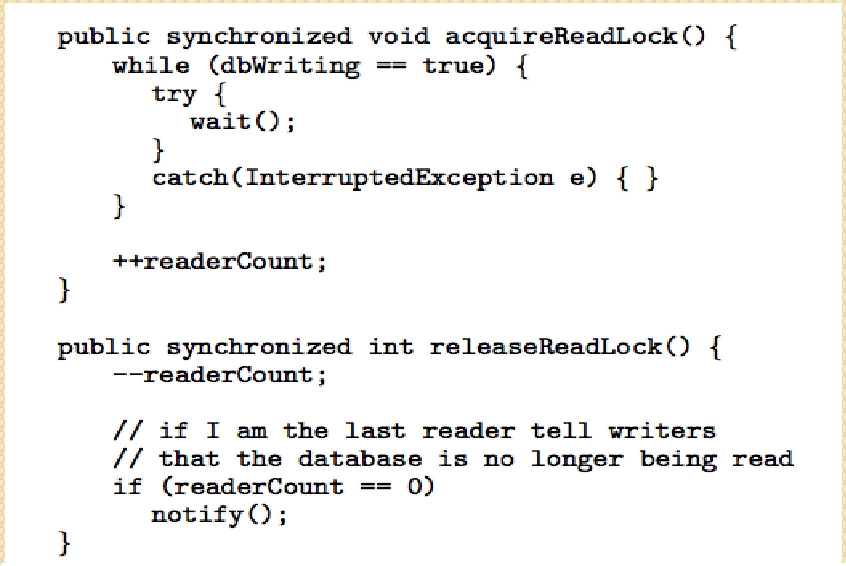
Figure 6.31 doWork() method.

Java Synchronisation (6)

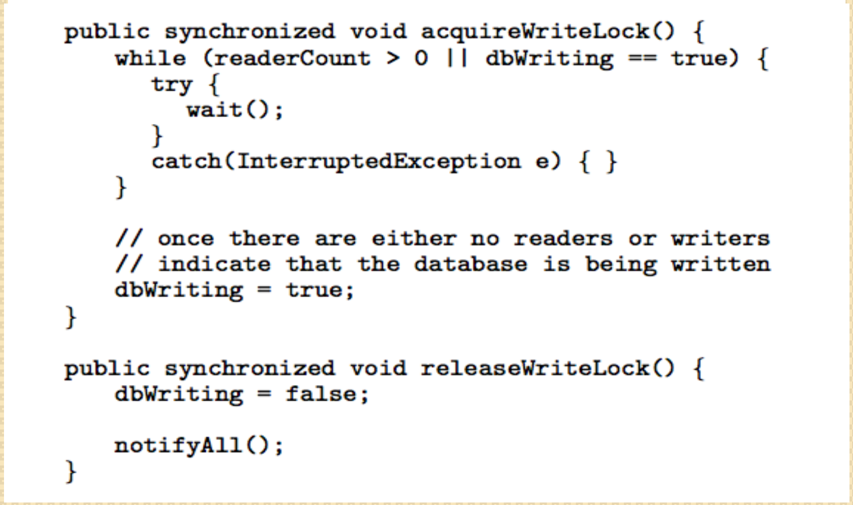
Readers-Writers



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

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

   public synchronized void acquireReadLock() {
      // Figure 6.33
   }

   public synchronized void releaseReadLock() {
      // Figure 6.33
   }

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

   public synchronized void releaseWriteLock() {
      // Figure 6.34
   }
}
```



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

   ++readerCount;
}

public synchronized int releaseReadLock() {
   --readerCount;

   // if I am the last reader tell writers
   // that the database is no longer being read
   if (readerCount == 0)
      notify();
}
```



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

   // once there are either no readers or writers
   // indicate that the database is being written
   dbWriting = true;
}

public synchronized void releaseWriteLock() {
   dbWriting = false;

   notifyAll();
}
```

Java Synchronisation (7)

Block Synchronisation

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

    synchronized(mutexLock) {
       criticalSection();
    }

    remainderSection();
}
```

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

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

- Synchronisation rules
  - Locks are reentrant (recursive)
  - A thread can simultaneously own the lock for multiple objects
    - Nesting of synchronised invocations
  - Non synchronised methods can be invoked regardless of lock ownership
  - Calls to `notify()` and `notifyAll()` when the wait set is empty have no effect
  - `wait()`,`notify()` and `notifyAll()` may only be invoked from synchronised methods or blocks
    - `IllegalMonitorStateException` is thrown otherwise

Java Synchronisation (8)

- Handling InterruptedException
    - wait() checks interruption status
    - The exception clears the interruption status
    - Handle or propagate
        - Should a thread blocked in a wait set be interrupted?

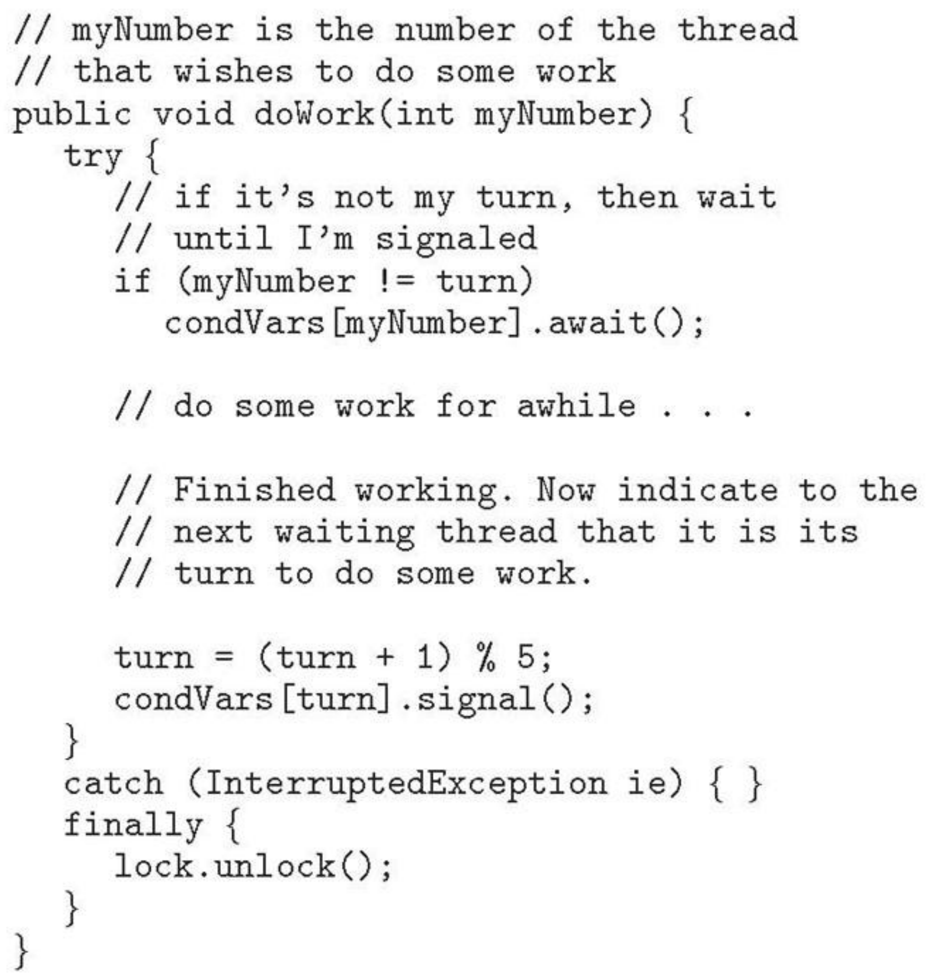
Concurrency Features in Java (1)

- `Java.util.concurrent` and `java.util.concurrent.locks`
- Reentrant locks
  - Similar to `synchronized` but with fairness feature
- Semaphores
- Condition variables
  - Similar to `wait`, `notify` and `notifyAll`
  - Must be associated with a lock

```
Lock key = new ReentrantLock();
key.lock();
try {
   //critical section
   //…
} finally {
   key.unlock();
}
```

```
Semaphore sem = new Semaphore(1);
try {
   sem.acquire();
   //critical section
   // …
} catch (InterruptedException ie) {
} finally {
   sem.realease();
}
```

Concurrency Features in Java (2)



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

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

Figure 6.37 doWork() method with condition variables.

For contemplation (1)

- What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.
- Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.
- Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.
- Demonstrate that monitors and semaphores are equivalent insofar as they can be used to implement the same types of synchronization problems.
- Discuss the tradeoff between fairness and throughput of operations in the readers-writers problem. Propose a method for solving the readers/writers problem without causing starvation.

For contemplation (2)

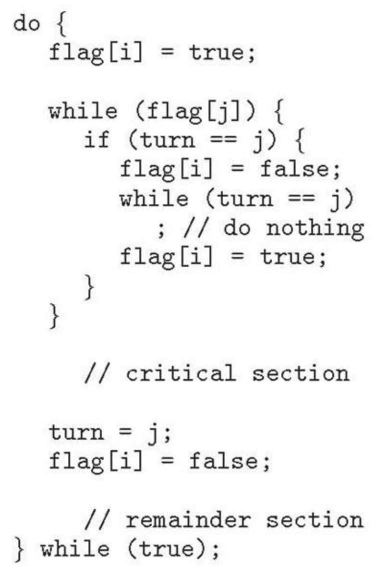
- Under which circumstances can a semaphore be used to solve the critical section problem. Clearly demonstrate that in these circumstances the semaphore solution satisfies the conditions for a solution to the critical section problem. Provide code that shows how the semaphore is used to protect the critical section.
- How does the signal() operation associated with monitors differ from the corresponding operation defined for semaphores?
- Show that, if the acquire() and release() semaphore operations are not executed atomically, then mutual exclusion may be violated.
- The wait() statement in all Java program examples in this chapter is part of a while loop. Explain why you would always need to use a while statement when using wait() and why you would never use an if statement.
- Suppose we replace the wait() and signal() operations of monitors with a single construct await(B), where B is a general Boolean expression that causes the process executing it to wait until B becomes true.
  - Write a monitor using this scheme to implement the readers–writers problem.
  - Explain why, in general, this construct cannot be implemented efficiently.
  - What restrictions need to be put on the await statement so that it can be implemented efficiently? (Hint: Restrict the generality of B; see Kessels [1977].)

For contemplation (3)

- The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, *P*0 and *P*1, share the following variables:

  boolean flag[2]; /* initially false */

  int turn;
- The structure of process *Pi* (i == 0 or 1) is shown in the figure; the other process is *Pj* (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.



```
do {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; // do nothing
         flag[i] = true;
      }
   }

      // critical section

   turn = j;
   flag[i] = false;

      // remainder section
} while (true);
```

Figure 6.42 The structure of process P_i in Dekker's algorithm.

For contemplation (4)

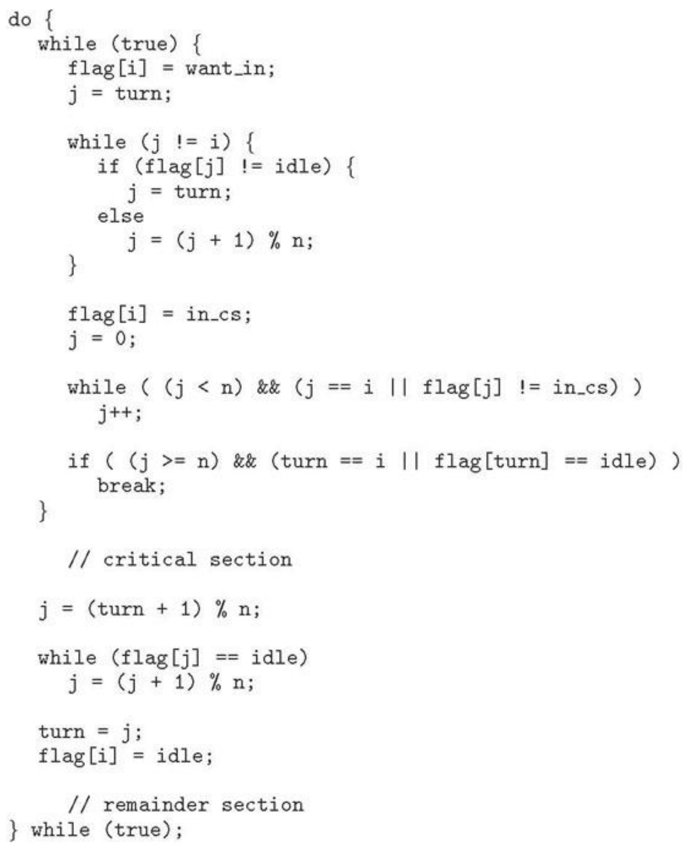
- The first known correct software solution to the critical-section problem for *n* processes with a lower bound on waiting of *n* − 1 turns was presented by Eisenberg andMcGuire. The processes share the following variables:

  enum pstate {idle, want in, in cs};

  pstate flag[n];

  int turn;

- All the elements of flag are initially idle; the initial value of turn is immaterial (between 0 and n-1). The structure of process *Pi* is shown in the figure. Prove that the algorithm satisfies all three requirements for the critical-section problem.



```
do {
   while (true) {
      flag[i] = want_in;
      j = turn;

      while (j != i) {
         if (flag[j] != idle) {
            j = turn;
         else
            j = (j + 1) % n;
      }

      flag[i] = in_cs;
      j = 0;

      while ( (j < n) && (j == i || flag[j] != in_cs) )
         j++;

      if ( (j >= n) && (turn == i || flag[turn] == idle) )
         break;
   }

      // critical section

   j = (turn + 1) % n;

   while (flag[j] == idle)
      j = (j + 1) % n;

   turn = j;
   flag[i] = idle;

      // remainder section
} while (true);
```

Figure 6.43 The structure of process P_i in Eisenberg and McGuire's algorithm.

For contemplation (5)

- The **Singleton** design pattern ensures that only one instance of an object is created. For example, assume we have a class called Singleton and we only wish to allow one instance of it. Rather than creating a Singleton object using its constructor, we instead declare the constructor as private and provide a public static method—such as getInstance() —for object creation:

  Singleton sole = Singleton.getInstance();
- The figure provides one strategy for implementing the Singleton pattern. The idea behind this approach is to use **lazy initialization**, whereby we create an instance of the object onlywhen it is needed—that is, when getInstance() is first called. However, the figure suffers from a race condition. Identify the race condition.
- The following figure shows an alternative strategy that addresses the race condition by using the **double-checked locking idiom**. Using this strategy, we first check whether instance is null. If it is, we next obtain the lock for the Singleton class and then double-check whether instance is still null before creating the object. Does this strategy result in any race conditions? If so, identify and fix them. Otherwise, illustrate why this code example is thread safe.

```
public class Singleton {
   private static Singleton instance = null;

   private Singleton { }

   public static Singleton getInstance() {
     if (instance == null)
       instance = new Singleton();

     return instance;
   }
}
```

Figure 6.44 First attempt at Singleton design pattern.

```
public class Singleton {
   private static Singleton instance = null;

   private Singleton { }

   public static Singleton getInstance() {
     if (instance == null) {
       synchronized(Singleton.class) {
         if (instance == null)
           instance = new Singleton();
       }
     }

     return instance;
   }
}
```

Figure 6.45 Singleton design pattern using double-checked locking.

For contemplation (6)

- Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and—once finished—will return them. As an example, many commercial software packages provide a given number of licenses, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such requests will only be granted when an existing license holder terminates the application and a license is returned. The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:
  - #define MAX RESOURCES 5
  - int available resources = MAX RESOURCES;
- When a process wishes to obtain a number of resources, it invokes the decrease count() function:

```
/* decrease available resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease count(int count) {
  if (available resources < count) return -1;
  else { available resources -= count; return 0; }
}
```

- When a process wants to return a number of resources, it calls the decrease count() function:

```
/* increase available resources by count */
int increase count(int count) {
  available resources += count; return 0;
}
```

- The preceding program segment produces a race condition. Do the following:
  - Identify the data involved in the race condition.
  - Identify the location (or locations) in the code where the race condition occurs.
  - Using Java synchronization, fix the race condition. Also modify decreaseCount() so that a thread blocks if there aren't sufficient resources available.