6SENG002W Concurrent Programming

Lecture 8

Java Thread Synchronization & Monitors

Java Thread Synchronization & "Monitors"

Aim of this lecture is to describe Java *thread synchronization* mechanisms by:

- reviewing the multi-threaded programming issues of *synchronization*, *fairness*, *starvation* & *deadlock*;
- illustrating the need for *thread synchronization* by means of the *Producer/Consumer* problem by illustrating the problems that can arise without synchronization;
- showing how these problems can be solved by using *thread synchronisation* & the concept of a *monitor*;
- describing the *monitor* concept & *Java's version* of it:
  - the *properties* & *attributes* of a Java monitor,
  - how *Java monitors* are defined,
  - **`synchronized`** methods & the **`synchronized`** statement,
  - monitor methods: `wait()`, `notify()` & `notifyAll()`,

PART I

Multi-threaded Programming Issues

Review of Multi-threaded Programming Issues

Scenario: several threads *share data* via a file, e.g. threads *write data to the file*, & at the same time, other threads *read data from the file*.

Interference: will occur, if the read & write actions of the threads are **not coordinated** then the shared file will inevitably be **corrupted**.

Synchronization: is required to *coordinate* the read & write actions of these threads to stop them interfering with each other & ensuring *"data integrity"*.

Starvation: occurs when one or more of the threads is **blocked** from accessing the file & **cannot make progress**.

Deadlock: is the ultimate form of starvation & occurs when two or more threads are **waiting on a condition that cannot be satisfied**.

Fairness: ensures that all threads *competing for access* to the file, will eventually have their *"turn"* & be granted access to it.
A **fair** system does not allow for *starvation* or for *deadlock*; & ensures each thread gets enough access to *limited resources* to make *reasonable progress*.

Synchronising Threads

So far we have only considered examples of *independent, asynchronous threads*.

That is, each thread

- contained all of the *data & methods* required for its execution, &
- did **not** require any *outside resources or methods*.

Further, the threads in previous examples **ran at their own pace** without concern over the state or activities of any other concurrently running threads.

However, in most situations separate concurrently running threads *do share data* & must consider the *state & activities* of other threads.

To explore these issues & the problems that can arise in these situations we shall now examine one of the “classic” concurrent programming scenarios known as the *Producer/Consumer Problem*.

PART II

The Producer/Consumer Problem

The Producer/Consumer Problem

The Producer/Consumer problem is a scenario where the `Producer` generates a stream of data which is then consumed by a `Consumer`.

For example, the `Producer` thread writes data to a file while the `Consumer` thread reads data from the same file.

Example uses *concurrent threads* that *"share a common resource"*, i.e. a file.

Thus the actions of the `Producer` & `Consumer` must be **synchronised to avoid the corruption of the data**.

The general form of the problem is:

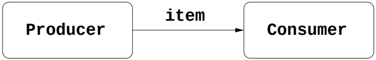

Figure : 8.1 Producer/Consumer Problem.

The *flow* of data is evened out by introducing a *buffer*:

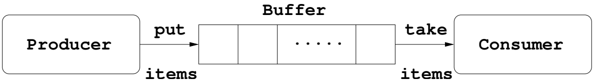

Figure : 8.2 Producer/Consumer Problem – with a Buffer.

An example of the Producer/Consumer problem

We shall now consider a simple example of the Producer/Consumer problem, using a *mailbox* containing a single integer, as the *shared buffer*.

We shall define the following code:

- **`Mailbox`** *interface*,
- `SimpleMailbox` class,
- `Producer` class,
- `Consumer` class,
- `ProdConSimpleMailbox` class is the *main program*.

The **`Mailbox`** Interface

The **`Mailbox`** interface specifies the *interface to the mailbox* in terms of the following two methods **`put`** & **`take`**:

```
interface Mailbox
{
    // put an item into the mailbox

    public void put( int value ) ;

    // take (i.e. remove & return) an item
    // from the mailbox

    public int take( ) ;
}
```

The `SimpleMailbox` class

The `SimpleMailbox` class *implements* the **`Mailbox`** interface.

It is just a simple mailbox that holds a single integer value:

```
class SimpleMailbox implements Mailbox
{
  private int contents = 0 ;

  public void put( int value )
  {
     contents = value ;
  }

  public int take()
  {
     return contents ;
  }
}
```

Notes:

1. `SimpleMailbox` is **not a thread**.
2. We are employing *data hiding/encapsulation* of the `contents` variable, i.e. it should be either `private` or `protected`.

The `Producer` Thread

The `Producer` generates integers from 0 to 9 & puts them in an object that implements the **`Mailbox`** interface.

```
class Producer extends Thread
{
  private final Mailbox mailbox ;        // Mailbox "interface"
  private final int numberOfItems ;

  public Producer( Mailbox mailbox, int Pid, int numberOfItems )
  {
     super( "Producer #" + Pid ) ;       // Thread( thrd_name )
     this.mailbox = mailbox ;
     this.numberOfItems = numberOfItems ;
  }

  public void run()
  {
    for (int i = 0; i < numberOfItems; i++)  {
       mailbox.put( i ) ;
       System.out.println(getName() + " put:   " + i) ;

       try {  sleep( (int)(Math.random() * 100) ) ;  }
       catch ( InterruptedException e ){ }
    }
  }
}
```

The `Consumer` Thread

The `Consumer` consumes all integers from an object that implements the **`Mailbox`** interface.

```
class Consumer extends Thread
{
  private final Mailbox mailbox ;    // Mailbox "interface"
  private final int numberOfItems ;

  public Consumer( Mailbox mailbox, int Cid, int numberOfItems )
  {
     super( "Consumer #" + Cid ) ;   // Thread( thrd_name )
     this.mailbox = mailbox ;
     this.numberOfItems = numberOfItems ;
  }

  public void run()
  {
    int value = 0 ;
    for (int i = 0; i < numberOfItems; i++)
    {
      value = mailbox.take() ;
      System.out.println(getName() + " taken: " + value) ;
    }
  }
}
```

The `ProdConSimpleMailbox` Main Program

```
class ProdConSimpleMailbox
{
  public static void main( String args[] )
  {
    final int NUMBITEMS = 10 ;

    Mailbox smb = new SimpleMailbox() ;

    Producer p1 = new Producer( smb, 1, NUMBITEMS ) ;
    Consumer c1 = new Consumer( smb, 1, NUMBITEMS ) ;

    p1.start() ;
    c1.start() ;
  }
}
```

The main program is a stand-alone Java application that:

- creates one `SimpleMailbox` (**non-thread**) object;
- creates & starts the `Producer` & `Consumer` threads.

Important the *same* mailbox object, i.e. **`smb`** (`SimpleMailbox`), is passed as an actual parameter to **both** the `Producer` & `Consumer` threads, hence they *communicate* via **`smb`**.

Diagrammatic View of `ProdConSimpleMailbox`

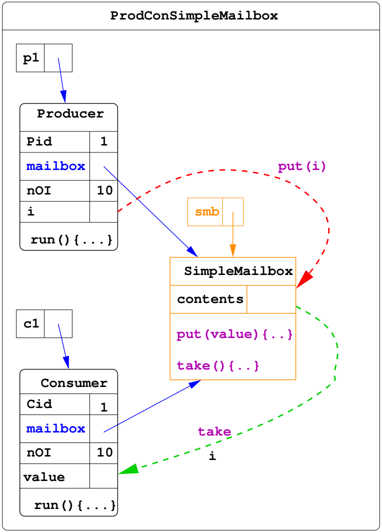

Figure : 8.3 `ProdConSimpleMailbox` Program Structure

The Output

An example of the output of `ProdConSimpleMailbox`:

```
Producer #1 put:   0
Consumer #1 taken: 0
Consumer #1 taken: 0
Consumer #1 taken: 0
Consumer #1 taken: 0
Consumer #1 taken: 0
Consumer #1 taken: 0
Consumer #1 taken: 0
Consumer #1 taken: 0
Consumer #1 taken: 0
Consumer #1 taken: 0
Producer #1 put:   1
Producer #1 put:   2
Producer #1 put:   3
Producer #1 put:   4
Producer #1 put:   5
Producer #1 put:   6
Producer #1 put:   7
Producer #1 put:   8
Producer #1 put:   9
```

This is **definitely not** what is required!

Producer/Consumer Synchronization

Note that neither the `Producer` nor the `Consumer` *make any effort whatsoever* to ensure that the `Consumer` is taking each value produced once & only once, as can be seen from the example output.

Remember that we require the `Consumer` to take each integer produced by the `Producer` exactly once.

That is:

- No data is **lost**.
- No data is **created**.
- Ensures *FIFO usage*.

It is clear that some kind of *synchronization* is required to achieve the desired result.

The **lack of synchronization** between these two threads can result in two kinds of problems.

Synchronization Problems

One problem arises when the `Producer` is *quicker* than the `Consumer` & generates two numbers before the `Consumer` has a chance to consume the first one.

Thus the `Consumer` would **lose a number**:

```
    . . .
Consumer #1 taken: 3
Producer #1 put:   4
Producer #1 put:   5
Consumer #1 taken: 5
    . . .
```

Another problem occurs when the `Consumer` is *quicker* than the `Producer` & **consumes the same value twice**:

```
    . . .
Producer #1 put:   4
Consumer #1 taken: 4
Consumer #1 taken: 4
Producer #1 put:   5
    . . .
```

Either way, the result is **wrong**.

Outline of the Solution

The above problems are examples of *interference* as a result of a *lack of protection*, *non-coordinated access* & *race conditions*.

These arise from multiple, asynchronously executing threads accessing an object at the same time & getting the wrong result.

To prevent this interference in the Producer/Consumer example:

> *the putting of a new integer into a mailbox object by the `Producer` **must be synchronised with** the retrieval of an integer from the mailbox by the `Consumer`.*

The `Consumer` must consume each integer exactly once.

Solution

The Java run time system provides *thread synchronization* through the use of the programming language mechanism known as a *monitor*.

Thus, our Java solution to the Producer/Consumer problem will use *monitors*.

The *monitor* will provide the necessary:

- *protection* of the shared data;
- *synchronised coordinated access* to the shared data; &
- *resolution* of any *race conditions* that may arise.

The advantage of *encapsulating a shared resource within a monitor* is that all processes that access the resource do so in a *controlled way*, thus minimising the possibility of errors.

PART III

A Brief History & Introduction to "Monitors"

A (Very) Brief History of Monitors

Monitors were invented by Per Brinch Hansen in 1972, he initially used the term *"shared class"* to refer to them.

The monitor concept was later improved by C. A. R. Hoare in 1974, the version used in his paper became the *"standard"* definition of a monitor.

Brinch Hansen added monitors to the programming language Pascal to produce *Concurrent Pascal* in 1975.

Monitors were developed as an operating system structuring concept.

They provided a *mechanism for safe resource sharing* within an operating system, by allowing *localised control* over how processes were scheduled when *accessing shared resources*.

We shall look at Hoare's version of a monitor in the next lecture; see also the following papers (available on the module web site):

1. P. Brinch Hansen, *Structured multiprogramming*, Communications of the ACM, Vol. 15, Issue 7, Pages 574–578, 1972.
2. C. A. R. Hoare, Monitors: *An Operating System Structuring Concept*, Communications of the ACM, Vol. 17, Issue 10, Pages 549–557, 1974.
3. P. Brinch Hansen, *The programming language Concurrent Pascal*, IEEE Trans. Software Eng., SE-1 (1975), Pages 199–207.

The Aim of a Monitor

One of the main problems in writing an operating system is allowing concurrent processes to *safely* share resources within the operating system.

A very difficult task & requires skill & knowledge to do correctly & efficiently.

Thus one of the main aims of a *monitor* was to make the writing of concurrent systems, in particular, operating systems safer & easier.

This was to be achieved by *allowing localised control over how processes were scheduled when accessing shared resources*.

In particular, a *monitor* provides protection of shared resources via *encapsulation* & *synchronised access* by:

- *encapsulating* a shared resource within a monitor; &
- requiring all processes that access the resource do so in a *controlled way*, i.e. such that **there is no interference, etc.**

By using this approach it was hoped to minimise the possibility of errors when writing concurrent programs/systems.

The Monitor Concept

A *monitor* is a concurrent programming language mechanism used as a *structuring device* for concurrent programs —

- A *shared resource* & its associated *operations* should be collected together in a program unit, e.g. a class.
- The *variables declared inside the monitor*, represent the resource, are only accessible by the *monitor's operations*.
- The *operations* on the resource are invoked by procedure (method) calls whenever required by the processes which share the resource.
- A monitor permits only one of its procedure bodies to be active at a time, this ensures *mutual exclusive* access to the resource, i.e. the procedure bodies behave like *critical sections*.
- The monitor has one or more associated *"condition variables"*.
  A condition variable is a *queue of processes* that are waiting to *regain access* to the monitor.
- Depending on the *state (condition) of the resource*, processes can place themselves in this queue & will get removed from it by other processes that have changed its state (condition).

Definition of a Monitor

A *monitor* consists of:

- A collection of declarations of *permanent variables*.

  They are used to represent & indicate the *state of the resource*.

- A collection of *procedure & function* declarations.

  These implement *operations on the resources* by manipulating the monitor variables.

  There are two types of procedures & functions

  1. *visible* outside of the monitor, i.e. interface with outside world.
  2. *hidden/invisible*, i.e. *"helper"* methods only used inside the monitor.

- A *monitor body* — which is a sequence of statements that are executed only once to *initialise the state of the monitor*, i.e. the resource.

- At least one *condition variable*, i.e. a queue of waiting processes.

- A *synchronisation lock*, used to control access to the monitor.

Once the monitor has been initialised it functions as a *package of data & procedures*.

Important Properties of Monitors

1. The monitors' *variables* are **not accessible from outside** of the monitor.
   - So they can only be read or altered via the monitor's procedures & functions.
   - Therefore, the state of the resource can only be altered through these procedures.
2. A monitor *permits only one of its visible procedure bodies to be active at a time*.
   - Ensures *mutually exclusive* access to the resource.
   - Even if two procedures (either the same or different ones) are called by two processes simultaneously, one of the calls will be delayed until the other is completed.
   - In other words, the procedure bodies together behave like a *single critical section* all controlled by the *monitor's synchronisation lock*.
3. A monitor is a *passive* object, i.e. it does nothing by itself.
   - The only way in which a monitor is executed is when one of its visible procedures is called by some process.

Java's "Version" of a Monitor

- **Java does NOT have a "`Monitor`" class or an abstract class.**
- **Nor does Java have a "`Monitor`" interface.**
- However, it uses *features built into* the `Object` class:
- This *"works"* because the `Object` class is the *"root"* of Java's *class hierarchy*.
- Therefore, in Java an instance of *any class*, i.e. an object, has the *"potential"* to be a monitor.
- This *"potential"* to be a monitor that all objects possess has to be *"enabled"* or *"switched on"*.
- This is achieved by declaring *at least one of its methods to be* **`synchronized`**.
- This is achieved by adding "`synchronized`" to the signature of the method.

Java's Version of Monitor Concepts (I)

Concept: *Encapsulated/protected (shared) resource*.

Java: Use access modifiers `private` or `protected` variable declarations in the "monitor" class.

Concept: *Monitor's procedures*.

Java: Declare at least one "monitor" class method with the `synchronized` keyword in its signature.

Concept: *Mutually exclusive execution* of monitor procedures.

Java: The bodies of the `synchronized` methods are treated as *one critical section*.

Access is controlled to these method's critical section bodies by the *monitor's synchronisation lock*.

Java's Version of Monitor Concepts (II)

Concept: *"Condition variable"*, i.e. the monitor's queue of waiting processes.

Java: Each monitor object has associated with it a *"wait set"*.

This is the set/queue of threads that were executing one of the monitor methods & had to *stop* executing it because the operation **could not be completed safely**.

As a result it *relinquishes the monitor*, by *releasing the monitor's lock* & placing itself in the *"wait set"*.

Concept: *Adding & removing processes to & from the monitor's queue.*

Java: A thread places itself in the wait set using `wait()`.

A thread is removed from the wait-set by *another thread* calling either `notify()` or `notifyAll()`.

Diagrammatic View of a Java Monitor Object

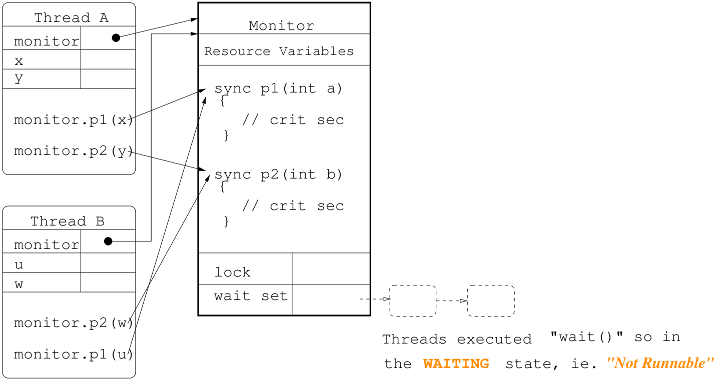

Figure : 8.4 Java Monitor Object.

Note *simultaneous calls to* `p1` & `p2` are **mutually exclusive**.

A thread *gets placed* in the *wait set* when it calls `wait()` from a monitor method, & it *gets removed* from it when another thread calls `notify()` or `notifyAll()`.

Java “Monitors” Implemented via the `Object` Class

This is possible because the `Object` class:

- has a *synchronization lock*
- has a *“wait set”* (threads queue) that is used via:
  - `wait()` – *add* a thread to it
  - `notify()` & `notifyAll()` – *remove* thread(s) from it
- its the *root* of Java’s class hierarchy.

Therefore, every object has a unique *synchronization lock* & *wait set* associated with it & thus has the *“potential”* to be a monitor.

The *“monitor”* aspect is *“switched on”* by defining a `synchronized` method.

It is an *object’s lock* that is used to *control access* to the monitor via executing its `synchronized` methods.

So, a monitor method can **only be executed** if the calling thread has **acquired the monitor’s lock**.

Other threads **cannot execute** any of the monitor’s `synchronized` methods until the *monitor is released*, i.e. the lock is released, this **guarantees mutual exclusion**.

Java Thread States & Monitors

Recall the Java thread states diagram Figure 6.1, from Lecture 6.

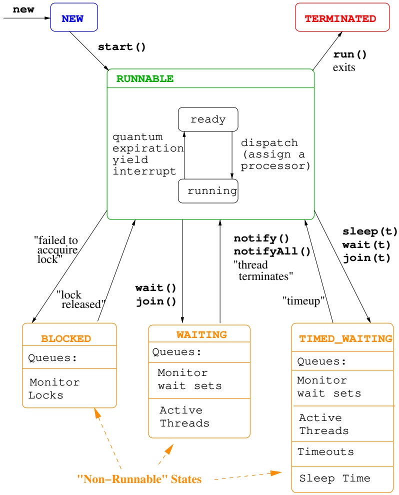

Figure : 6.1 Java Thread States

Java Thread States when using a Monitors

When a thread interacts with a monitor it can affect the thread's state (see the *Thread State* diagram) in several ways, for example:

Whenever a thread calls & starts to execute (i.e, enters) a monitor's `synchronized` method, the thread that called the method is said to *"hold"* or *" have acquired"* or *"locked"* the monitor.

Other threads can still call any of the monitor's `synchronized` methods, *but* these calls will **not be successful**, & the calling thread will enter the **`BLOCKED`** state until the monitor is *released* i.e. *unlocked*.

A monitor is *released* (*unlocked*) when a thread exits the monitor method it is executing either by completing it or by calling `wait()` & placing itself in the monitor's *wait-set*.

If a thread calls `wait()` it gets added to the *wait-set* & is moved from the **`RUNNABLE`** to **`WAITING`** state; & it **unlocks the monitor**.

The last action a thread should perform just before it completes a monitor method is to call `notifyAll()`, this **"wakes-up"** the threads that are stuck in the monitor's *wait-set*.

This results in them changing their thread state from **`WAITING`** to **`RUNNABLE`**.

Diagrammatic view of Monitors & Thread States

Threads interacting with a monitor can affect the *their states*.

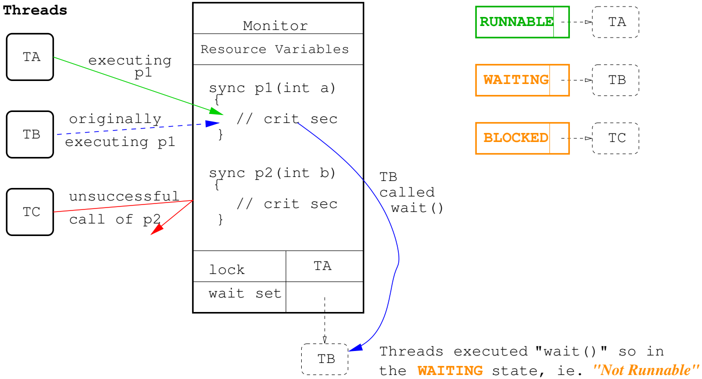

Figure : 8.5 Monitor & Thread States

Remember the *wait set* is the set of threads (i.e. `TB`) that tried to perform one of the monitor methods (i.e. `p1`) & have had to release the monitor, because **the operation could not be completed safely**.

Synchronising Threads in the JVM

The underlying implementation of Java monitors by the JVM must ensure that if two (or more) threads simultaneously try to access the resource *encapsulated* within the monitor, via its `synchronized` methods, that only one of them succeeds.

So the JVM must:

- be able to *resolve* the *race conditions* between the threads trying to access the `synchronized` methods; &
- ensure *mutually exclusive* access to the *critical section*, i.e. the combined bodies of the `synchronized` methods.

The JVM achieves this by ensuring that the acquisition (*locking*) & release (*unlocking*) of a monitor is done *automatically* and *atomically*, by using *single uninterruptible* actions.

Locking & Unlocking Synchronised Methods

For example, a `synchronized` method performs two special actions relevant only to multi-threaded operation:

1. after computing a reference to an object but before executing its body, it *locks* a *lock* associated with the object; &
2. after execution of the body has completed, either normally or abruptly, it *unlocks* that same *lock*.

It is this *atomic locking* & *unlocking* that *resolves* the race conditions & ensures *mutually exclusive* access to the monitor.

This guarantees the integrity of the monitor & the resource encapsulated inside it.

`synchronized` Methods

A class is "defined" as being a *monitor*, if one of its methods is defined with the `synchronized` keyword as a modifier for at least one of the class' methods.

For example, to turn the `Mailbox` into a monitor we would need to do this to the **`put`** & **`take`** methods:

```
public synchronized void put(int value )
{
  // Now a ``critical section''
}

public synchronized int take( )
{
  // Now a ``crucial section''
}
```

The addition of "`synchronized`" indicates that **`put`** & **`take`** are **monitor methods**, i.e. their bodies are *critical sections*.

This means that **only one of these methods can be executing at once**, i.e. they are **mutually exclusive**.

The `synchronized` Statement

Generally, *critical sections* in Java programs are methods.

You can mark smaller code segments as *critical sections* by using the `synchronized` statement.

```
synchronized ( Expression )
{
  // critical section
}
```

`Expression` must be a reference type, i.e. an object.

A `synchronized` statement acquires a *mutual-exclusion lock* on behalf of the executing thread.

It executes the code block then *releases the lock*.

While the executing thread *owns the lock*, **no other thread** may acquire it.

However, `synchronized` blocks violate the object-oriented paradigm & leads to confusing code that is difficult to debug and maintain.

For the majority of your Java programming purposes, it's **best** to use `synchronized` only at the method level.

PART IV

Producer/Consumer Problem Solution using a Monitor

Producer/Consumer Problem Solution: the `MailboxMonitor` Class

- The solution we shall adopt is to place the *synchronization* between the `Producer` & `Consumer` within the **`take`** & **`put`** methods of a mailbox object.

- We shall do this by defining a class that implements the **`Mailbox`** interface called **`MailboxMonitor`**.

- The **`MailboxMonitor`** class for the Producer/Consumer example modifies the signatures of the two methods **`put`** & **`take`** by adding the `synchronized` modifier.

- This means the two methods are now `synchronized` & as a result the **`MailboxMonitor`** class is now a *monitor*.

- **Note:** that the Java system associates a unique monitor with every instance of **MailboxMonitor**, because it has `synchronized` methods.

The **MailboxMonitor** Class

We define **MailboxMonitor** as a *monitor* class as follows:

```
class MailboxMonitor implements Mailbox
{
  // Shared Resource
  private int     contents  = -1 ;         // encapsulated
  private boolean available = false ;      // data


  public synchronized int take( )          // monitor method
  {
     while ( !available )
     {
        try {
              wait() ;    // add calling thread to 'wait-set'
            }
        catch( InterruptedException e ){ }
     }

     available = false ;  // change state of monitor

     notifyAll() ;        // signal state change to waiting threads

     return contents ;
  }
```

MailboxMonitor Class: put

The **put** method is also a monitor method & has a very similar structure:

```
  public synchronized void put( int value )
  {
     while ( available )
     {
       try {
             wait() ;
           }
       catch( InterruptedException e ){ }
     }

     contents  = value ;
     available = true ;

     notifyAll() ;
  }

} // MailboxMonitor
```

The `ProdConMailboxMonitor`

The main program for this solution is given below.

Note that we now instantiate **`MailboxMonitor`** as the class that implements the **`Mailbox`** interface.

```
class ProdConMailboxMonitor
{
  public static void main( String args[] )
  {
     final int NUMBITEMS = 10 ;

     // Create: MailboxMonitor, Producer & Consumer

     Mailbox mbm = new MailboxMonitor() ;

     Producer p1 = new Producer( mbm, 1, NUMBITEMS ) ;

     Consumer c1 = new Consumer( mbm, 1, NUMBITEMS ) ;

     // Start Producer & Consumer

     p1.start() ;
     c1.start() ;
    }
}
```

Diagrammatic View of `MailboxMonitor`

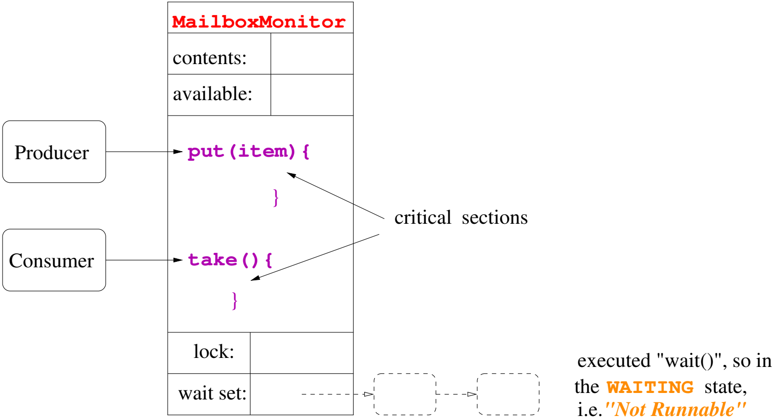

Figure : 8.6 **`MailboxMonitor`** Object

Note that there is *mutual exclusion* between executions of:

- **`put`** & **`take`**,
- multiple **`put`**s,
- multiple **`take`**s.

The Output

So now if we use the `MailboxMonitor` class instead of the `SimpleMailbox` class in the previous program then we get the following output from the `ProdConMailboxMonitor`:

```
Producer #1 put:   0
Consumer #1 taken: 0
Producer #1 put:   1
Consumer #1 taken: 1
Producer #1 put:   2
Consumer #1 taken: 2
Producer #1 put:   3
Consumer #1 taken: 3
Producer #1 put:   4
Consumer #1 taken: 4
Producer #1 put:   5
Consumer #1 taken: 5
Producer #1 put:   6
Consumer #1 taken: 6
Producer #1 put:   7
Consumer #1 taken: 7
Producer #1 put:   8
Consumer #1 taken: 8
Producer #1 put:   9
Consumer #1 taken: 9
```

Which is exactly what we require!

`MailboxMonitor` Shared Resource

The *shared resource* is represented by two *encapsulated* (`private`) variables:

- The integer `contents` which is the current contents of the mailbox.
- `available` is used to indicate the state of the *"shared resource"*, i.e. the contents of the mailbox.

  As there are **only TWO states**, (a value is available or is not available), we can use a Boolean variable.

Meaning of the Boolean values of the `available` variable are:

True:
- the current value in `contents` *can be taken* by the `Consumer`,
- a new value **cannot be put** into `contents` by the `Producer`.

False:
- a new value *can be put* into `contents` by the `Producer`,
- the current value in `contents` **cannot be taken** by the `Consumer`.

The Behaviour of `put` & `take`

The behaviour of the **put** & **take** monitor (`synchronized`) methods:

The `put` method:

- can only **put** a new value in the mailbox when `available` is **false**.
- will be blocked by entering the `while`-loop & calls `wait()` when `available` is **true**.
- this stops `put` from **overwriting** the value in the mailbox that has not yet been **taken** by the `Consumer`.

The `take` method:

- can only **take** the value in the mailbox when `available` is **true**.
- will be blocked by entering the `while`-loop & calls `wait()` when `available` is **false**.
- this stops `take` from either **taking a value from an "empty" mailbox** or **re-taking the value in the mailbox** that has already been taken.

The Behaviour of `put`

Whenever the `Producer` calls the **`put`** method, the `Producer` *acquires the monitor* for the **`MailboxMonitor`** thereby preventing the `Consumer` from calling the **`take`** method.

```
public synchronized void put(int value)
{  // monitor acquired by Producer
   while ( available )
   {
      try { // ``full'' so release monitor
            wait();
            // ``re-acquired'' monitor, check if Ok to do `put'
      } catch (InterruptedException e){ }
   }
   // now OK to put item
   contents = value;
   available = true;

   notifyAll();
} // monitor released by Producer
```

When the **`put`** method finishes, the `Producer` *releases the monitor* thereby **unlocking** the **`MailboxMonitor`**.

The Behaviour of **take** is Similar

Conversely, whenever the `Consumer` calls the **take** method, the `Consumer` *acquires the monitor* for the **MailboxMonitor** thereby preventing the `Producer` from calling the **put** method.

```
public synchronized int take()
{ // monitor acquired by Consumer

   while ( !available )
   {
      try {
            // 'empty' so release monitor
            wait();
            // 're-acquired' monitor, recheck if Ok to do 'take'
      }
      catch (InterruptedException e){ }
   }

   // now OK to take the item
   available = false;
   notifyAll();

   return contents;

} // monitor released by Consumer
```

Interaction between `wait()` & `notifyAll()`

The `Producer` uses **`put`** & the `Consumer` uses **`take`** to ensure that each value placed in the mailbox by the `Producer` is retrieved once & only once by the `Consumer`.

The **`put`** & **`take`** methods in the **`MailboxMonitor`** both make use of the `wait()` & `notifyAll()` methods to coordinate putting & taking values into & out of the mailbox.

In general, `wait()` is used in conjunction with `notifyAll()` to – *coordinate the activities of multiple threads using the same resource*.

The `wait()` method causes the current thread to wait (possibly forever) until another thread notifies it via `notifyAll()`, that the state of the resource has changed.

Example: Coordinating Taking & Putting

An example of how the interaction between `wait()` & `notifyAll()` works in the Producer/Consumer program is as follows:

- When the `Consumer` starts to execute **`take`** if the operation **cannot be completed** because there is no new value available to be consumed, then it has to wait (possibly forever) until one becomes available.

  This *"waiting"* is achieved by executing `wait()`.

- After the `Producer` has *changed the state* of the **`MailboxMonitor`** by putting a new value into it, it must notify the `Consumer` that the *state has changed*.

  This *"notification of a state change"* is achieved by executing `notifyAll()`.

The `wait()` Method

The **`MailboxMonitor`**'s **`take`** method contains a `while` statement that loops until `available` becomes `true`.

If `available` is `false`, then the `Consumer` knows that the `Producer` has *not yet produced a new number* & the `Consumer` *must wait until it has*.

The `Consumer` enters the `while` loop & calls `wait()`, & waits until there is a notification from the `Producer` thread.

When the `Producer` uses the **`put`** method to put in a new value, at the end it calls `notifyAll()`, this notifies the `Consumer`.

The `Consumer` *leaves the wait state* & continues within the `while` loop, rechecking the condition that would now be `false`, so it exits the `while` loop & completes **`take`**.

If the `Producer` had not generated a number, **`take`** would go back to the beginning of the loop & continue to wait until the `Producer` had generated a new number & called `notifyAll()`.

`put` works in a similar fashion waiting for the `Consumer` thread to consume the current value before allowing the `Producer` to produce a new one.