6SENG002W Concurrent Programming

Lecture 9

Java Monitors: Theory, Usage & Comparison

Java Monitors: Theory, Usage & Comparison

Aim of this lecture is to:

- Explain in detail the aspects of a *Java monitor*:
  - the versions of `notifyAll()` & `wait()` methods,
  - relationship between a monitor & the `notifyAll()` & `wait()` methods.
- *Categories* of monitor methods.
- General form of a Java *monitor method*.
- Designing a monitor method's *"guard"*.
- *"Re-entrant"* monitors.
- Compare *Hoare's* monitor with *Java's* monitor.
- *Monitor solutions* for the following standard concurrency problems:
  - *Producer/Consumer* problem
  - *Readers & Writers* problem
  - *Dining Philosophers* problem.

PART I

Details of the Monitor Methods:

`wait(),wait(ms),wait(ms, ns)`

&

`notify(),notifyAll()`

The `notifyAll()` Method

In the `MailboxMonitor`'s `take` method, the last thing it does just before returning is to call the `notifyAll()` method.

`notifyAll()` wakes up **all** the threads waiting on the same monitor held by the current thread, i.e. the `MailboxMonitor`'s monitor.

So in general, the waiting threads move from either the **`WAITING`** or **`TIMED_WAITING`** state to the **`RUNNABLE`** state.

These *newly woken threads compete for the monitor* along with other threads who are *trying to enter the monitor for the first time*.

One thread gets the monitor & the others may either:

- remain in the **`RUNNABLE`** state & wait to be scheduled; or
- try & fail to acquire the monitor & enter the **`BLOCKED`** state.

For example, the `Consumer` calls `take`, so it *holds the monitor* for the `MailboxMonitor` during the execution of `take`.

At the end of `take`, the call to `notifyAll()` *wakes up* the `Producer` which is the only thread waiting on the `MailboxMonitor`'s monitor.

So the `Producer` can acquire the `MailboxMonitor` monitor & proceed.

The `notify()` Method

The `Object` class has another method – `notify()`, that has a similar effect as `notifyAll()`.

Except that it wakes up **just one thread** (*chosen at random*) rather than *all* the threads in the *wait set*.

The awoken thread will not be able to proceed until the current thread relinquishes the lock on this object.

The awoken thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object.

For example, the awoken thread has **no special privilege or disadvantage** in being the next thread to lock this object.

That is a *"notified"* thread competes with:

- threads trying to *acquire the monitor for the first time*, by calling one of the monitor's methods; &
- other threads that *were waiting* but have been *notified* & are now trying to *re-acquire* the monitor.

The “Timed” `wait(t)` Methods

So far we have only seen the *untimed* version of *waiting* used in the Producer/Consumer example – `wait()`, that *waits indefinitely for notification*.

But, the `Object` class has two *timed* versions of the waiting method:

`wait( long millis )`
: causes the current thread to wait for notification or until the timeout period has elapsed – `millis` is measured in milliseconds.

`wait( long millis, int nanos )`
: causes the current thread to wait for notification or until the timeout period plus the additional nanoseconds has elapsed.

Notes:

1. For a call of `wait(0)` (`t = 0`) the time is ignored & is the same as calling `wait()`.
2. Java implementations of `wait( long millis, int nanos )` **usually ignore** the nanoseconds argument, since this degree of temporal accuracy is not always achievable.

Effect of “Calling” the `wait()` Method

All versions of `wait()` have a similar effect on the *“current thread”* that calls them.

We shall refer to the current thread as “**CT**”.

The effect of *thread* **CT** calling a `wait()` method is as follows:

1. `wait()` causes **CT** to be put in the *wait set* (a set of threads), for this object.
2. **CT** *relinquishes any & all synchronization claims* on this object, i.e. it **unlocks the object**.
3. **CT** becomes *disabled for thread scheduling purposes*, i.e. it enters one of the two **“not runnable”** states:
   - **WAITING**, if it called `wait()` or
   - **TIMED_WAITING**, if it called `wait(t)` ($t > 0$).

`wait()` – Unlocking & Locking a Monitor

When the thread enters the `wait()` method, for example, from within the `while`-loop of both the `put` & `take` methods:

- The `wait()` method, as it places the current thread into the *wait set* for this monitor (object), **unlocks** *only this monitor*.
- The monitor is **released atomically**, i.e. it is *unlocked*.
- Any other monitors on which the current thread may be synchronized, i.e. *holds a lock*, **remain locked** while the thread waits.
  This could result in **deadlock**!
- The thread **can only complete** the call to `wait()`, after the monitor has been **acquired again**, i.e. it is has been *re-locked* by this thread.

How the Thread **CT** is “Woken Up”

Thread **`CT`** remains *dormant* in one of the **not runnable** states (**`WAITING`** or **`TIMED_WAITING`**) until one of four things happens:

- Some other thread calls `notify()` for this object & **`CT`** happens to be arbitrarily chosen as the thread to be awoken.
- Some other thread calls `notifyAll()` for this object & all waiting threads are woken up.
- Some other thread interrupts thread **`CT`**.
- If **`CT`** called `wait(t)`, the specified amount of real time has elapsed, more or less.

  Unless, **`CT`** called `wait(0)` then the time is ignored & is the same as calling `wait()` & the thread simply waits until notified.

At which point thread **`CT`** is removed from the *wait set* for this object & re-enabled for thread scheduling, i.e. reenters the **`RUNNABLE`** state.

Thread **`CT`** Trying to “Re-acquire” the Monitor

- After thread **`CT`** *reenters* the **`RUNNABLE`** state it then competes in the usual manner with other threads to *synchronize on the object*.
- However, there is **no guarantee** that it will immediately gain control of the monitor when it starts to execute.
- For example, if another thread has *gained control of the monitor* before **`CT`** can do so then **`CT`** will be put in the **`BLOCKED`** state.
- **`CT`** will stay in the **`BLOCKED`** state *until the monitor is released*, i.e. unlocked.
- When the monitor is *released* (unlocked) **`CT`** moves back to the **`RUNNABLE`** state, & attempts to acquire control of it again.
- This cycle could be repeated indefinitely.

Thread **`CT`** on “Re-gaining Control” of the Monitor

- Once thread **`CT`** has *gained control of the object*, **all its synchronization claims on the object are restored** to the situation as of the time that the `wait()` or `wait(t)` was called.
- Thread **`CT`** then returns from the invocation of the `wait()` method, i.e. to the line of code immediately after the call of `wait()`.
- On return from `wait()` the *synchronization state of the object* & of thread **`CT`** is exactly as it was when the `wait()` (or `wait(t)`) method was invoked.
- If the current thread is *interrupted* by another thread while it is waiting, then an `InterruptedException` is thrown & it will execute the `try`’s `catch` block.
- This *exception is not thrown* until the lock status of this object has been restored as described above.

An Important Fact about: `wait()`, `wait(ms)`, `wait(ms, ns)` & `notify()` & `notifyAll()`

Note: that

- `notify()` `notifyAll()` &
- `wait()`, `wait(ms)`, `wait(ms, ns)`

can only be called from within a `synchronized` **method or statement**.

So these methods **can only be called by a thread that holds the synchronisation lock on the object**, i.e. has acquired the lock associated with its monitor.

If a thread does not own the lock on the object, & it attempts to call any of the above methods on the object then an `IllegalMonitorStateException` is thrown.

PART II

Designing Monitor Methods

Different types of Monitor Methods

As an example, consider how to design a Java monitor to manage a *shared printer*, that obviously requires *mutually exclusive* control.

So consider the following standard actions for a printer:

- *print a document*,
- *refill paper tray*,
- *replace toner cartridge*,
- *restart it*,
- *service it*,
- *upgrade driver software*, etc.

If we had to define a monitor to *"manage"* a printer that supported the above actions we would need to decide how the various *monitor methods* that implemented the actions should be structured.

We would need to do this as it is **very unlikely** that all of these actions should be *implemented using the same "synchronisation behaviour"* for each monitor method.

Categories of Monitor Methods

To help us deal with the different types of monitor methods we may need to design we can *categorise* them as follows:

Obligatory-Guarded: an action *must be performed*, but if the resource is **not** in the right state *wait* until it is & then do it.

Obligatory-Unguarded: an action *must be performed*, **irrespective of the state of the resource**, i.e. **do not wait but just perform it**.

Optional: an action should be performed if the resource is in the right state, but if it is not **abort** the action, i.e. **do not wait & do not do it**.

Sometimes referred to as *"baulking"*.

Miscellaneous: some variation of the above.

For example, be willing to wait for a specific time, using `wait(t)`, or limit the number of calls to `wait()` "*re-tries*" using a counter.

Obligatory-Guarded Monitor Method

We shall focus on the most complicated category of monitor method – *Obligatory-Guarded*.

Others are variations on this, i.e. parts are either omitted or simplified.

Obligatory-Guarded action: *print a document*.

- Uses a *"pre-condition guard"* – sufficient paper & toner to print the document.
- If when it is attempted the resource is **not in the correct state to do it** – *pre-condition guard* is **false**, then it *waits* indefinitely.
- Using a `while` loop, repeatedly tries to perform the monitor action; & repeats the cycle – while not in correct state wait, after *returning* from the `wait()`, rechecks state, etc, etc.
- When the **guard is true**, it drops out of `while` loop & *completes the action* & *exits* the monitor method.
- Calling thread **has** "*delegated*" the (unlimited) re-tries to the monitor method.

Obligatory-Unguarded, Optional & Miscellaneous Monitor Method

Obligatory-Unguarded action: *Reset the printer*.

- The action is performed irrespective of the state the resource is in.
- Does not require a `while` loop guard in the monitor method.

Optional action: *Replace toner cartridge*.

- If when it is attempted the resource is in the *correct state* it does it, but if **not** then it **does not** `wait()` & just exits.
- It leaves it up to the calling thread to decide if it wants to attempt it later.
- In essence the calling thread **has not** "*delegated*" the re-tries to the monitor method, but handles it itself.

Miscellaneous action: are there any?

- Combinations of the above, e.g. use a `while` loop guard & `wait(time)`.
- Adds a *"re-try attempts limit"* to the `while` loop guard's "resource state" condition, e.g. just try a few times & after that exit method.

General Form of a Java Obligatory-Guarded Monitor Method

```
public synchronized void operation()
{
   // CHECK if resource is in correct state

   while ( ``test resource_state'' )
   {
      // NOT in correct state to do "operation"
      try {  wait() ;  }
      catch (InterruptedException e){ }
   }
   // NOW in correct state to do "operation"

   // UPDATE resource variables
   resource = ... ;

   // UPDATE state indicator
   resource_state_indicator = ... ;

   // NOTIFY waiting threads of change of state
   notifyAll() ;
}
```

(Ignoring the passing & returning of parameters.)

Obligatory-Guarded Monitor Method: Why use a `while` loop?

Recall the `put` & `take` methods in the `MailboxMonitor` & the `while` loop:

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

Q: What are the consequences of replacing the **`while`** loop with an **`if`**?

```
public synchronized void put(int value)
{
   if ( available )
   {
      try {  wait() ;  }  catch(InterruptedException e){ }
   }
   contents = value ;
   available = true ;
   notifyAll() ;
}
```

Notes about Java Monitor Methods

- **MUST ALWAYS USE** a `while` loop, as **NO guarantee** the resource will be in the *correct state* to do the operation *after returning from* `wait()`.
- Using a `while` loop ensures the resource's state is **always checked before attempting to do the operation**, even after *re-acquiring the monitor* on returning from `wait()`.
- So must use *"resource state indicator"* variable(s) to check if the resource is in the correct state to do the monitor's methods, e.g. define a *helper* function that returns *"state"* of the resource.
- The `while` loop & its *condition* are known as the **"guard"**, since it must be *"passed"* before the operation can be performed.
- It is possible to *"derive"* the `while`'s *"guard"* condition.
- For example, the `take` operation can only be performed when the `MailboxMonitor` contains a new value.

How to Derive a Monitor Method's "Guards" (1)

We shall now show how to *"derive"*, the `while` loop's boolean *"guard"* condition for any monitor method.

We do this by :

1. Using a *case analysis* of the possible **"distinct" states** of the monitor's resource.
2. *Identifying the set of states* that each of the monitor's operations can **"safely"** be performed in.

Assuming the *resource encapsulated in the monitor* can have all of its *possible "distinct" states* divided up into n *different cases* (by doing a case analysis) & that it has m *operations* then:

Resource_State ::= S1 | S2 | . . . Sn

Where each Si is a *condition* that *"characterises"* one of the different states of the resource.

How to Derive a Monitor Method's "Guards" (2)

For each monitor method, i.e. operation on the resource, we can tabulate which states that operation can be **performed in** & which states it can **not be performed in** as follows:

Operation	**Resource States** (S1 - Sn) **CAN DO Operation**	**Resource States** (S1 - Sn) **CAN NOT DO Operation**
Operation-1	S1, S2	S3 - Sn
Operation-2	S1, S2, S3, S5, S8	S4, S6, S7, S9 - Sn
⋮	⋮	⋮
Operation-m	Sn	S1 - Sn-1

Example: consider students trying to print & technicians topping up the paper or replacing the toner cartridge.

Deriving the `while` loop Conditions

Now using the previous table we can tabulate the `while` loop *"guard" conditions* that control when the operation **can be performed** & when it should **not be performed**.

m **Operation**	`while` loop **Guard Condition to**	
	DO Operation	**NOT DO Operation**
Operation-1	Do_Op1	NotDo_Op1
...	...	...
Operation-i	Do_Opi	NotDo_Opi
...	...	...
Operation-m	Do_Opm	NotDo_Opm

Where for example

- Do_Op1 = S1 $\vee$ S2,
- NotDo_Op1 = S3 $\vee$ S4 $\vee \ldots \vee$ Sn.

We want to ensure that when **NotDo_Op1** is false that **Do_Op1** is true[1], i.e. it is *safe* to perform Operation-1.

[1]Formally that would mean that: ! NotDo_Opi $\Rightarrow$ Do_Opi.

Using the Monitor Method Guard

Then for each *Operation-i* that has Do_Opi as its *pre-condition*, i.e. it **must be true**, the monitor method's form would be:

```
public synchronized void Operation-i ( )
{
   // Check NotDo_Opi, while it is true BLOCK this operation

   while ( NotDo_Opi )
   {
      try {  wait() ;  }
      catch(InterruptedException e){ }
   }

   // NotDo_Opi is now false,
   // so provided Do_Opi is true
   // it is safe to do this operation

   // Do Operation-i

   ...
   notifyAll() ;
   ...
}
```

Example: Designing the `MailboxMonitor` Methods

The number of states that the mailbox has is just *two*:

1. Contains a *new* value that has **not** been consumed.
2. Contains an *old* value that has been consumed, & awaiting a new value.

Therefore, since this resource has just 2 states, which state it is in can be represented by a *single boolean value*: `new_value`.

- `new_value` is *true* when the `MailboxMonitor` contains: **a new value that has not been consumed**.
- `new_value` is *false* when it contains: **an old value that has been consumed**.

Operation	**Condition to**	
	DO Operation	**NOT DO Operation**
`put`	contains an old value `new_value` = false	contains a new value `new_value` = true
`take`	contains a new value `new_value` = true	contains an old value `new_value` = false

The put Method

So for **put(i)** we have:

```
public synchronized void put( int i )
{

   // Check new-value
   while ( new-value )
   {
      try {
            wait() ;
          }
      catch(InterruptedException e){ }
   }

   // new-value is now false
   // so ``new-value == false''
   // so safe to do put(i)

   // Do put(i)

   ...
   notifyAll() ;
   ...
}
```

The `take` Method

And for **`take()`** we have:

```
public synchronized int take( )
{
   // Check new-value
   while ( ! new-value )
   {
      try {
            wait() ;
          }
      catch(InterruptedException e){ }
   }

   // ``! new-value'' is now false
   // so ``new-value'' is true
   // so safe to do take()

   // Do take()

   ...
   notifyAll() ;
   ...
}
```

PART III

Java Monitors are Re-entrant

Java Monitors are Re-entrant

The Java run time system allows a thread to *re-acquire a monitor* that it already holds because Java monitors are *“re-entrant”*.

Meaning, a thread can call a `synchronized` method on an object for which it already holds the monitor, there by re-acquiring the monitor.

Re-entrant monitors are important because they eliminate the possibility of a **single thread deadlocking itself** on a monitor that it already holds.

Consider class `ReentrantTest`:

```
class ReentrantTest
{
  public synchronized void a()
  {
      System.out.println("Starting method: a()") ;
      b() ;
      System.out.println("Finishing method: a()") ;
  }

  public synchronized void b()
  {
      System.out.println("Executing method: b()");
  }
}
```

How `ReentrantTest` is Executed

`ReentrantTest` contains two synchronized methods: **`a`** & **`b`**.

The first synchronized method, **`a`**, calls the other synchronized method, **`b`**.

When control enters method **`a`**, the current thread acquires the monitor for the `ReentrantTest` object.

Now, **`a`** calls **`b`** & because **`b`** is also synchronized the thread attempts to acquire the same monitor again.

Because Java supports re-entrant monitors, this works.

The current thread can acquire the `ReentrantTest` object's monitor again & both **`a`** & **`b`** execute to conclusion, producing the output:

```
Starting method:  a()
Executing method: b()
Finishing method: a()
```

In systems that do not support re-entrant monitors, this sequence of method calls **would result in deadlock**.

PART IV

Hoare's "Standard" Monitor (1974)
vs.
Java's Monitor (1996)

Hoare Monitor: Synchronization

Synchronization is achieved explicitly by using *two primitives* which operate on a new data type called a *Conditional Variable – "c"* & a *function* which allows its *state* to be examined.

`wait(c)` **current process is suspended** & kept in a FIFO queue, i.e. *"c"*, until another process performs a `signal(c)`.
After it has called `wait(c)`, it *leaves the monitor procedure*, allowing other processes to execute monitor procedures.

`signal(c)` **resumes a suspended process** that has been waiting at the *front of the queue c*.
The `signal(c)` operation does not have any effect if no process is suspended, i.e. *c*'s queue is empty.

`empty(c)` is a boolean function returns *true* if the queue of processes suspended on *c* is *empty*, otherwise it returns *false*.

These are the only operations allowed on a *conditional variable*.

A conditional variable *c*, can be thought of as a queue of processes, i.e. those which have performed `wait(c)`.

Hoare Monitor: Using `wait(c)` & `signal(c)`

A common scenario is the following:

- When a process is granted access to a monitor, it may request access to a resource.
- If the resource is *busy* or is in an inappropriate state to be used by the called procedure, a `wait(c)` command is executed, suspending the process & placing it into the conditional variable `c`'s queue.
- This permits other processes to execute the monitor procedures & eventually put the resource into a suitable state for the suspended process to use.
- After a process has done this it should perform a `signal(c)` which would *wake–up* the suspended process, which can now use the resource since it is in a suitable state.

Hoare's (& Brinch Hansen's) Monitors vs. Java's Monitors

There are **several important differences** between Hoare's *"standard"* monitor & Java's monitors.

Concept	**Hoare**	**Java**
encapsulated resource	By definition it is "private"	Must be enforced by `private` &/or `protected`
monitor procedures	All "visible" methods are mutually exclusive (ME) monitor methods.	At least one `synchronized` (ME) method, **BUT** can have non-`synchronized` (non-ME) methods.
condition variables	As many as required, accessed via `wait(c)`, `signal(c)`, `empty(c)`.	Only one *wait set* per monitor object, accessed via `wait()`, `notify()` & `notifyAll()`.
monitor guards	Use "`if ( guard )`", since *multiple* condition variables.	Must use "`while ( guard )`", since *only one* condition variable.
"Safe Usage"	Yes.	**NO**, as ALL aspects rely on programmer doing it right!

Conclusion

Hoare proposed his improved version of Brinch Hansen's monitors way back in 1974.

From the above comparison it is clear that —

Java's implementation of the monitor concept is a very poor one indeed!

This is mainly due to the fact that:

- It is **far less secure** because the "monitor" concept is **NOT enforced** at all by the Java language.

  It relies solely on the programmer following a *set of "guidelines"* to construct a Java "monitor".

- Hoare's version can have as *many Conidtion Variables as required*, this allows for far more flexible, efficient & sophisticated monitors.

- Java's version of a monitor is **NOT as sophisticated, flexible or easy to use** as Hoare's, largely due to just using one *"wait set"*.

- But using Java's low level concurrency features (e.g. lock classes) it is possible to "handcraft" a Java monitor similar to a Hoare monitor, but is even messier & more error prone than Java's standard monitor.

PART V

Classic Problems Solved Using Monitors

Classic Problems Solved Using Monitors

We shall now show how *Hoare style monitors* can be used to solve the following standard concurrency problems:

- *Producer/Consumer* problem
- *Readers & Writers* problem
- *Dining Philosophers* problem

NOTE: Java style monitor solutions can either be found on the module web site on the web or textbooks.

As an exercise you could translate them into Java style monitors!

Producer/Consumer Problem using Monitors

The Producer/Consumer problem arises when there is a difference in the relative speeds of execution of a producer of data & the consumer of the data, i.e. when the producer is faster than the consumer.

Under these circumstances the producer of data must have somewhere to store it until the consumer is ready to consume it.

The *flow* of data is evened out by introducing a *buffer*: the producer puts data into the shared buffer & the consumer takes data from the shared buffer.

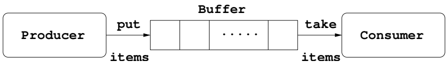

Figure : 9.1 Producer/Consumer Problem – with a Buffer.

In addition the:

Consumer: **must be prohibited from** consuming data that has not been produced, &

Producer: **must be prohibited from** over writing already produced data.

Producer/Consumer Problem using Monitors

Design of the Producer/Consumer Monitor:

- *Shared resource* encapsulated within the monitor is the *shared buffer*.
- The buffer actions to be defined as monitor procedures:
  – *"put an item to the buffer"*
  – *"take an item from the buffer"*.
- *Mutual exclusive* access to the buffer by the Producer & Consumer processes is *ensured by the semantics of a monitor*.
- **`put`** operation can only be performed when the buffer is **not full**.
  So must indicate when there are *free slots* in the buffer.
  Achieved using *conditional variable* — **`free_space`**.
- **`take`** operation can only be performed when the buffer is **not empty**.
  So must indicate when there are *items in the buffer*.
  Achieved using *conditional variable* — **`item_available`**.

Producer/Consumer Problem: Monitor Version (1)

```
Program ProducerConsumer
{
  Monitor Buffer
  {
    final int N = 10 ;         // size of the buffer ;

    Item[] buffer ;           // indexes 0 .. N-1
    int in, out ;
    int number_of_items ;

    Condition free_space, item_available ;

   public void put( Item item )
   {
     if ( number_of_items == N )  wait( free_space ) ;

     buffer[in]      = item ;
     in              = (in + 1) % N ;       // % is remainder
     number_of_items = number_of_items + 1 ;

     signal( item_available ) ;
   }
```

Producer/Consumer Problem: Monitor Version (2)

```
    public Item take( )
    {
      if (number_of_items == 0)  wait( item_available ) ;

      item = buffer[out] ;
      out  = (out + 1) % N ;
      number_of_items = number_of_items - 1 ;

      signal( free_space ) ;

      return item  ;
    }

    beginbody
      buffer = new Item[N] ;  // indexes 0 .. N-1
      in   = 0 ;              // put  buffer index
      out  = 0 ;              // take buffer index
      number_of_items = 0 ;   // initially empty
    endbody

 }  // Buffer Monitor
```

Producer/Consumer Problem: Monitor Version (3)

```
Process Producer( Buffer buf ) {
   Item item ;
   while ( true ) {
     produce(item) ;
     buf.put(item) ;
   }
}

Process Consumer( Buffer buf ) {
   Item item ;
   while ( true )  {
     item = buf.take() ;
     consume(item) ;
   } ;
}

  main()   // Main Program
  {
     Buffer buffer = new Buffer ;   // Monitor

     parbegin
        Producer( buffer ) ;
        Consumer( buffer ) ;
     parend ;
  }
} // ProducerConsumer
```

The Readers & Writers Problem

The Readers & Writers Problem is a *generalisation of the mutual exclusion problem*.

The general problem is:

- A *number of processes share some piece of data*, usually some kind of data base (e.g. bank accounts, airline reservation system, etc).
- A number of processes `n` say, *only "write" to the data base*, known as the "Writers".
- Some other processes `m` say, *only "read" from the data base*, known as the "Readers".

To maintain the **consistency of the data base** reading & writing *must be mutually exclusive*, as must writing by different "Writers".

But for the sake of efficiency *multiple readers are allowed*.

Requirements of the Readers/Writers Problem

The *Mutual Exclusion* requirements are related to the reading & writing to the data base:

- either k (≤ m) "Readers" can have *exclusive access to the data base*
- or 1 "Writer".

There are a number of *invariants* which should hold on & between the number of "Readers" ($\#Readers$) reading from the data base & the number of "Writers" ($\#Writers$) writing to the data base.

Invariants:

1. $0 \leq \#Writers \leq 1$
2. $0 \leq \#Readers \leq m$
3. $\#Writers = 1 \Rightarrow \#Readers = 0$
4. $\#Readers > 0 \Rightarrow \#Writers = 0$

Failed Monitor Design for Readers & Writers Problem

A monitor design for the Readers & Writers Problem that **DOES NOT WORK** is the following:

- Shared resource encapsulated within the monitor is **“obviously”** the *shared data base*.
- **“Obvious”** candidates for monitor procedures:
  – *“read from the data base”*,
  – *“write to the data base”*. .
- *Mutual exclusion* of access to the data base is ensured by the semantics of a Monitor.

This **fails** because ....?

Analysis of the First Attempt

The **BIG** problem with this solution is that it ensures *too much mutual exclusion*.

Mutual Exclusive Processes	*Required*
Writers & Readers	**Correct**
Individual Writers	**Correct**
Individual Readers	**INCORRECT!**

This is due to the definition of monitors, i.e. mutual exclusion in execution of the visible procedures.

So, if we require several Readers to be able to read the DB at once then the "reading" **CANNOT** be done inside a monitor procedure.

Therefore, the action of reading & writing must be split into three parts:

Reading	*Writing*

`start_read`
`read_data_base`
`end_read`
	`start_write`
	`write_data_base`
	`end_write`

Correct Monitor Design for the Readers & Writers Problem

- The data base is now **not encapsulated within the monitor**, but now the monitor is used to **implement** the *access protocol* to it.
- The four actions (monitor procedures) which are performed now are: `start_read`, `end_read`, `start_write`, `end_write`.
- *Mutual exclusion* of access to the data base by the Readers & Writers processes is now controlled by recording the numbers of Readers, Writers & by the semantics of Monitors.
- Need to indicate when it is permissible for a Reader to read, so use a *conditional variable* — **`OK_to_read`**.
- Need to indicate when it is permissible for a Writer to write, so use a *conditional variable* — **`OK_to_write`**.
- Behaviour of first & last Reader is different to other Readers:

  first Reader — **blocks the Writers**
  last Reader — **unblocks the Writers**

Readers/Writers Problem: Using Monitors (1)

```
Program ReadersWriters {

Monitor ReadWriteDatabase
{
   int numb_Readers ;
   bool Writing ;
   Condition OK_to_read, OK_to_write ;

   public void start_read()
   {
     if ( Writing ||  ( ! empty(OK_to_write) )
           wait(OK_to_read) ;

     numb_Readers = numb_Readers + 1 ;

     signal( OK_to_read ) ;
   }

   public void end_read()
   {
     numb_Readers = numb_Readers - 1;

     if ( numb_Readers == 0 )  signal( OK_to_write ) ;
   }
```

Readers/Writers Problem: Using Monitors (2)

```
    public void start_write();
    {
      if ( Writing || ( numb_Readers > 0 ) )  wait(OK_to_write) ;
      Writing = true ;
    }

    public void end_write( )
    {
      Writing = false ;

      if ( ! empty( OK_to_read ) )
           signal( OK_to_read ) ;
      else
           signal( OK_to_write ) ;
    }

    beginbody
        Writing      = false ;
        numb_Readers = 0 ;
    endbody

} // end Monitor ReadWrite
```

Readers/Writers Problem: Using Monitors (3)

```
Process Reader( ReadWriteDatabase rwdb ) {
   while ( true )
   {
     rwdb.start_read() ;
          read_data_base ;
     rwdb.end_read() ;
   }
}

Process Writer( ReadWriteDatabase rwdb ) {
   while ( true )
   {
     rwdb.start_write() ;
          write_data_base ;
     rwdb.end_write() ;
   }
}

main() // Main Program
{
   ReadWriteDatabase rwdb = new ReadWriteDatabase ;
   parbegin
        Reader(rwdb) ; ... ; Reader(rwdb) ;
        Writer(rwdb) ; ... ; Writer(rwdb) ;
   parend ;
}
```

The Dining Philosophers Problem

The Dining Philosophers Problem is used to illustrate various problems that can occur when many synchronized processes are competing for limited resources.

The problem is as follows:

- There are usually 5 philosophers who spend their time alternating between thinking & eating.
- After they have thought for a while they get hungry & enter the dining room & sit at their allotted place at a round table.
- In the middle of the table is a bowl of spaghetti.
- Between each pair of philosophers is one fork.
- Before an individual philosopher can take & eat some spaghetti he must have two forks – one taken from the left, & one taken from the right.
- The philosophers must find some way to share forks such that they all get to eat.

The Dining Philosopher's Table

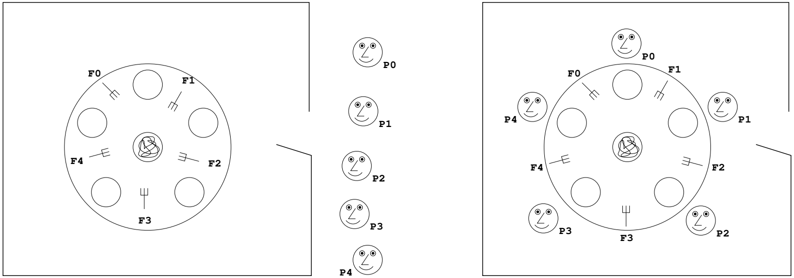

Figure : 9.2 The Dining Philosopher's Table

A Philosopher's Behaviour

The behaviour of each philosopher is a cycle:

1. think,
2. when they get hungry they enter the dining room,
3. sit down at their allotted place,
4. pickup the fork on their right,
5. pickup the fork on their left,
6. eat,
7. put down the fork on their right,
8. put down the fork on their left,
9. leave the dining room.

With this **unconstrained behaviour** it is possible for **deadlock** to occur, i.e. all the philosophers have the fork to their right.

(Non-Java) Dining Philosophers Problem: Forks Monitors

```
Monitor ForkMonitor
{
  final int NP = 5 ;
  int[] Fork ;
  Condition[] OK_to_eat ;

  public void Pick_up_Forks( int i )
  {
    if ( Fork[i] < 2 )  wait( OK_to_eat[i] ) ;
    Fork[(i+1) % NP] = Fork[(i+1) % NP] - 1 ;
    Fork[(i-1) % NP] = Fork[(i-1) % NP] - 1 ;
  }

  public void Put_down_Forks( int i )
  {
    Fork[(i+1) % NP] = Fork[(i+1) % NP] + 1 ;
    Fork[(i-1) % NP] = Fork[(i-1) % NP] + 1 ;
    if ( Fork[(i+1) % NP] == 2 )  signal( OK_to_eat[(i+1) % NP] ) ;
    if ( Fork[(i-1) % NP] == 2 )  signal( OK_to_eat[(i-1) % NP] ) ;
  }

  beginbody
     Fork = new int[NP] ;
     Condition OK_to_eat = new Condition[NP] ;
     for ( int i = 0; i < NP; i++ )  Fork[i] = 2 ;
  endbody
} // end Monitor ForkMonitor
```

How the Forks Monitor Works

- It uses the `Fork` array to keep track of *how many forks are available for each philosopher*.
  E.g. if `Fork[3] = 2` it means that philosopher 3 has 2 forks available.

- In **`Pick_up_Forks( int i )`**:
  - If philosopher i has < 2 forks available then it must **wait** by calling
    `wait( OK_to_eat[i] ).`
  - Note that philosopher `i` does **not** reduce its own count of available forks `Forks[i]`, this is done by its 2 neighbour philosophers.
  - When philosopher `i` *"picks up 2 forks"* the result is that it **reduces** the forks available to its 2 neighbours: philosopher `i-1` & philosopher `i+1`.

- In **`Put_down_Forks( int i )`**:
  - When philosopher `i` *"puts down 2 forks"* the result is that it **increases** the forks available to its two neighbours: philosopher `i-1` & philosopher `i+1`.
  - If its neighbour philosophers now have 2 forks available then **signal** they can eat by calling:

```
signal( OK_to_eat[(i-1] % NP] )
signal( OK_to_eat[(i+1] % NP] )
```

Dining Philosophers Problem: Philosophers & Main Program

```
Process Philosopher( int i , ForkMonitor fm )
{
   while ( true )
   {
     // Think() ;
     // SitDown() ;
     fm.Pick_up_Forks( i ) ;
     // Eat() ;
     fm.Put_down_Forks( i ) ;
     // GetUp() ;
   }
}

Program DiningPhilosophers
{
    ForkMonitor fm = new ForkMonitor ;

    parbegin
       Philosopher( 0, fm ) ;
       Philosopher( 1, fm ) ;
       Philosopher( 2, fm ) ;
       Philosopher( 3, fm ) ;
       Philosopher( 4, fm ) ;
    parend ;
}
```