Lecture 4: Monitors

- Introduction (Operations & Signalling Mechanisms);
- The Readers-Writers Problem SR;
- Emulating Semaphores with Monitors & Vice Versa
- The Dining Philosophers problem in SR;
- The Sleeping Barber Problem;
- Monitors in Java:
  - Recap on Basic Concurrency in Java
  - Queue Class in Java
  - Readers/Writers Problem

Monitors

- The main disadvantage with semaphores is that they are a low level programming construct.
- In a many programmers project, if one forgets to do `V()` operation on a semaphore after a CS, then the whole system can deadlock.
- What is required is a higher level construct that groups the responsibility for correctness into a few modules.
- *Monitors* are such a construct. These are an extension of the monolithic monitor found in OS for allocating memory etc.
- They *encapsulate* a set of procedures, and the data they operate on, into single modules (monitors)
- They guarantee that only one process can execute a procedure in the monitor at any given time (mutual exclusion).
- Of course different processes can execute procedures from different monitors at the same time.

Monitors (cont'd): Condition Variables

- Synchronisation is achieved by using *condition variables*, data structures that have 3 operations defined for them:
- *wait (C)* The process that called the monitor containing this operation is suspended in a FIFO queue associated with C. Mutual exclusion on the monitor is released.
- *signal (C)* If the queue associated with C is non-empty, then wake the process at the head of the queue.
- *non-empty (C)* Returns true if the queue associated with C is non-empty.
- Note the difference between the **`P`** in semaphores and **`wait(C)`** in monitors: latter always delays until **`signal(C)`** is called, former only if the semaphore variable is zero.

Monitors (cont'd): Signal & Continue

- If a monitor guarantees mutual exclusion:
  - A process uses the *signal* operation
  - Thus awakens another process suspended in the monitor,
  - So aren't there 2 processes in the same monitor at the same time?
  - Yes.
- To solve this, several signalling mechanisms can be implemented, the simplest is *signal & continue mechanism*.
- Under these rules the procedure in the monitor that signals a condition variable is allowed to continue to completion, so the *signal* operation should be at the end of the procedure.
- The process suspended on the condition variable, but is now awoken, is scheduled for *immediate resumption* on the exiting of procedure which signalled the condition variable.

Readers-Writers Using Monitors

```
_monitor (RW_control)
op request_read ( )
op release_read ( )
op request_write ( )
op release_write ( )

_body (RW_control)
   var nr:int := 0, nw:int := 0
   _condvar (ok_to_read)
   _condvar (ok_to_write)

   _proc (request_read ( ))
        do nw > 0 ->
                _wait (ok_to_read)
        od
        nr := nr + 1
   _proc_end

   _proc (release_read ( ))
        nr := nr - 1
        if nr = 0 ->
                _signal(ok_to_write)
        fi
   _proc_end

   _proc (request_write ( ))
      do nr > 0 or nw > 0 ->
        _wait (ok_to_write)
      od
      nw := nw + 1
   _proc_end

   _proc (release_write ( ))
      nw := nw -1
      _signal (ok_to_write)
      _signal_all (ok_to_read)
   _proc_end
_monitor_end
```

File rw_control.m

Readers-Writers Using Monitors (cont'd) Resource Main (main.sr)

```
resource main ( )
      import RW_control

      process reader (i:= 1 to 20)
            RW_control.request_read( )
            Read_Database ( )
            RW_control.release_read( )
      end

      process writer (i := 1 to 5)
            RW_control.request_write( )
            Update_Database ( )
            RW_control.release_write( )
      end
end
```

Emulating Semaphores Using Monitors

- Semaphores/monitors are concurrent programming primitives of equal power: Monitors are just a higher level construct.

```
_monitor semaphore
    op p ( ), v ( )
_body semaphore
    var s:int := 0
    _condvar (not_zero)
    _proc (p ( ))
        if s=0 -> _wait(not_zero) fi
                # only _wait if s=0
        s := s - 1
    _proc_end

    _proc (v ( ))
        if not_empty(not_zero)=true->
        _signal (not_zero)
        #only _signal if suspended processes
                [] else -> s := s + 1
        # else increment s
        fi
    _proc_end
_monitor_end
```

Emulating Monitors Using Semaphores

- Firstly, need blocked-queue semaphores (SR is OK)
- Secondly, need to implement *signal and continue* mechanism.
- Do this with
  - a variable **c_count**,
  - one semaphore, **s**, to ensure mutual exclusion
  - & another, **c_semaphore**, to act as the condition variable.
- **_wait** translates as:

```
c_count := c_count + 1
V (s)
P (c_semaphore)              #_wait always suspends
c_count := c_count - 1       # 1 less process in monitor
```

- & **_signal** as:

```
if c_count > 0 ->
     V (c_semaphore)         # only _signal if
[] else -> V (s)             # waiting processes
fi
```

Dining Philosophers Using Monitors

```
_monitor (fork_mon)
   op take_fork (i:int),
   op release_fork (i:int)
_body (fork_mon)
   var fork [5]:int := ([5] 2)
   _condvar (ok2eat, 5)
# define an array of
# condition variables

   _proc (take_fork (i))
        if fork [i] != 2 ->
                _wait (ok2eat[i])
        fi
        fork [(i-1) mod 5]:=
                fork[(i-1) mod 5]-1
        fork [(i+1) mod 5] :=
                fork[(i+1) mod 5]-1
   _proc_end

   _proc (release_fork (i))
        fork [(i-1) mod 5] :=
                fork[(i-1) mod 5]+1
        fork [(i+1) mod 5] :=
                fork[(i+1) mod 5]+1

        if fork[(i+1)mod 5]=2 ->
          _signal(ok2eat[(i+1)mod 5])
        fi      #rh phil can eat

        if fork[(i-1) mod 5]= 2 ->
          _signal(ok2eat[(i-1)mod 5])
        fi      #lh phil can eat

   _proc_end

_monitor_end
```

Dining Philosophers Using Monitors (cont'd)

```
resource main ( )
      import fork_mon

      process philosopher (i:= 1 to 5)
            do true ->
                  Think ( )
                  fork_mon.take_fork (i)
                  Eat ( )
                  fork_mon.release_fork(i)
            od
      end
end
```

- Using monitors yields a nice solution, since with semaphores you cannot test two semaphores simultaneously.
- The monitor solution maintains an array fork which counts the number of free forks available to each philosopher.

Dining Philosophers:Proof of No Deadlock

Theorem Solution Doesn't Deadlock

- *Proof:*
  - Let $\#E$ be the number of philosophers who are eating, and have therefore taken both forks. Then the following invariants are true from the program:

$$Non-empty(\mathbf{ok2eat}[\mathbf{i}]) \Rightarrow \mathbf{fork}[\mathbf{i}] < 2 \qquad \text{eqn (1)}$$

$$\sum_{\mathbf{i}=1}^{5} fork[\mathbf{i}] = 10 - 2(\#E) \qquad \text{eqn (2)}$$

- Deadlock implies $\#E = 0$ and all philosophers are enqueued on `ok2eat` and none are eating:
  - If they are all enqueued then (1) implies $\sum \mathbf{fork[i]} \leq 10$
  - If no philosopher is eating, then (2) implies $\sum \mathbf{fork[i]} \leq 5$.
- Contradiction implies that the solution does not deadlock.
- But individual starvation can occur. How? How to avoid?

Monitors: The Sleeping Barber Problem

- A small barber shop has two doors, an entrance and an exit.
- Inside is a barber who spends all his life serving customers, one at a time.

1. When there are none in the shop, he sleeps in his chair.
2. If a customer arrives and finds the barber asleep:
   - he awakens the barber,
   - sits in the customer's chair and sleeps while his hair is being cut.
3. If a customer arrives and the barber is busy cutting hair,
   - the customer goes asleep in one of the two waiting chairs.
4. When the barber finishes cutting a customer's hair,
   - he awakens the customer and holds the exit door open for him.
5. If there are waiting customers,
   - he awakens one and waits for the customer to sit in the barber's chair,
   - otherwise he sleeps.

Monitors: The Sleeping Barber Problem (cont'd)

- The barber and customers are interacting processes,
- The barber shop is the monitor in which they react.

Monitors: The Sleeping Barber Problem (cont'd)

```
_monitor (barber_shop)
   op get_haircut( ), finish_cut( ), get_next_customer( )
_body (barber_shop)
   var barber: int :=0, chair: int :=0, open: int:=0
   _condvar (barber_available)          # when barber > 0
   _condvar (chair_occupied)            # when chair > 0
   _condvar (door_open)                 # when open > 0
   _condvar (customer_left)             # when open = 0
```

```
_proc (get_haircut())
    do barber=0 ->
      _wait(barber_available)
    od
    barber := barber - 1
    chair := chair + 1
    _signal (chair_occupied)
    do open=0 -> _wait (door_open) od
    open := open - 1
    _signal (customer_left)
_proc_end # called by customer
```

```
_proc (get_next_customer( ))
     barber := barber +1
     _signal(barber_available)
     do chair = 0 ->
            _wait(chair_occupied)
     od
     chair := chair -1
_proc_end # called by barber

_proc (finished_cut( ))
     open := open +1
     _signal (door_open)
     do open=0 ->
            _wait(customer_left)
     od
_proc_end # called by barber
_monitor_end
```

Sleeping Barber Using Monitors (cont'd) Resource Main (**main.sr**)

```
resource main ( )
      import barber_shop

      process customer (i:= 1 to 5)
            barber_shop.get_haircut(i)
            sit_n_sleep()
      end

      process barber ()
            do true ->
                  barber_shop.get_next_customer( )
                  cut_hair ( )
                  barber_shop.finished_cut( )
            od
      end
end
```

Sleeping Barber Using Monitors (cont'd)

- For the Barbershop, the monitor provides an environment for the customers and barber to rendezvous
- There are four synchronisation conditions:
  - Customers have to wait for barber to become available to get a haircut
  - Customers have to wait for barber to open door for them
  - Barber needs to wait for customers to arrive
  - Barber needs to wait for customer to leave
- Processes
  - wait on conditions using **`wait()`**s in loops
  - **`Signal()`** at points when conditions are true

Monitors in Java

- Java implements a slimmed down version of monitors.
- Java's monitor supports two kinds of thread synchronization: *mutual exclusion* and *cooperation*:
  - *Mutual exclusion*, supported in the JVM via object locks (aka 'mutex'), enables multiple threads to independently work on shared data without interfering with each other.
  - *Cooperation*, supported in the JVM via the **`wait()`** & **`notify()`** methods of class **`Object`**, enables threads to work together towards a common goal.

Monitors in Java: Recap on Threads (cont'd)

- A Java thread is a lightweight process with own stack and execution context, and has access to all variables in its scope.
- Threads are programmed by either extending `Thread` class or implementing the `runnable` interface.
- Both of these are part of standard `java.lang` package.
- `Thread` instance is created by:

```
Thread myProcess = new Thread ( );
```

- New thread started by executing:

```
MyProcess.start ( );
```

- `start` method invokes a `run` method in the thread.
- As `run` method is undefined as yet, code above does nothing.

Monitors in Java: Recap on Threads (cont'd)

- We can define the **run** method by extending the **Thread** class:

```
class  myProcess extends Thread ( );
{
        public void run ( )
        {
        System.out.println ("Hello from the thread");
        }
}
myProcess p = new myProcess ( );
p.start ( );
```

- Best to terminate threads by letting **run** method to terminate.
- If you don't need to keep a reference to the new thread can do away with **p** and simply write:

```
new myProcess ( ).start( );
```

Monitors in Java: Recap on Threads (cont'd)

- As well as extending the **`Thread`** class, can create lightweight processes by implementing the **`Runnable`** interface.
- This has the advantage that you can make one of your own classes, or a system-defined class, into a process.
- Cannot do this with threads as Java only allows you to extend one class at a time.
- Using the **`Runnable`** interface, previous example becomes:

```
class  myProcess implements Runnable ( );
{
        public void run ( )          {
        System.out.println ("Hello from the thread");
        }
}
Runnable p = new myProcess ( );
New Thread(p).start ( );
```

Monitors in Java: Recap on Threads (cont'd)

- If a thread has nothing immediate to do (e.g it updates the screen every second) then it should be suspended by putting it to sleep.
- There are two flavours of sleep method (specifying different times)
- `join( )` waits for the specified thread to complete and provides some basic synchronisation with other threads.
- That is "join" start of a thread's execution to end of another thread's execution so that a thread will not start until other thread is done.
- If `join()` is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing:

```
try
{
        otherThread.join (1000);// wait for 1 sec
}
catch (InterruptedException e ) {}
```

Monitors in Java: Synchronization

- Conceptually threads in Java execute concurrently and therefore could simultaneously access shared variables.
- To prevent 2 threads having problems when updating a shared variable, Java provides synchronisation via a slimmed-down monitor.
- Java's keyword synchronized provides mutual exclusion and can be used with a group of statements or with an entire method.
- The following class will potentially have problems if its update method is executed by several threads concurrently.

```java
class  Problematic
{
        private int data = 0;
        public void update ( )      {
                data++;
        }
}
```

Monitors in Java: Synchronization (cont'd)

- Conceptually threads in Java execute concurrently and therefore could simultaneously access shared variables.

```
class ExclusionByMethod {
        private int data = 0;
        public synchronized void update ( ){
                data++;
        }
}
```

- This is a simple monitor where the monitor's permanent variables are private variables in the class;
- Monitor procedures are implemented as **`synchronized`** methods.
- Only 1 lock per object in Java so when a **`synchronized`** method is invoked it waits to obtain the lock, execute the method, and then releases the lock.
- This is known as *intrinsic locking*.

Monitors in Java: Synchronization (cont'd)

- Another way to implement mutual exclusion is to use the `synchronized` statement within the body of a method.

```
class ExclusionByGroup  {
        private int data = 0;
        public void update ( ){
                synchronized (this) {    // lock this object for
                        data++;          // the following group of
                }                        // statements
        }
}
```

- The keyword `this` refers to the object invoking the update method.
- The lock is obtained on the invoking object.
- A `synchronized` statement specifies that the following group of statements is executed as an atomic, non interruptible, action.
- A `synchronized` method is equivalent to a monitor procedure.

Monitors in Java: Condition Variables

- While Java does not explicitly support condition variables, there is one implicitly declared for each synchronised object.
- Java's **`wait()`** & **`notify()`** resemble SR's **`wait()`** & **`signal()`** but can only be executed in **`synchronized`** code parts (when object is locked)
- The **`wait()`** method releases the lock on an object and suspends the executing thread in a delay queue (one per object, usually FIFO).
- The **`notify()`** method awakens the thread at the front of the object's delay queue.
- **`notify()`** has signal and continue semantics, so the thread invoking notify continues to hold the lock on the object.
- The awakened thread will execute at some future time when it can reacquire the lock on the object.
- Java has **`notifyAll()`** method, similar to **`signal_all()`** in SR.

Monitors in Java: Queue Class

- The use of `wait()` and `notify()` in Java can be seen in the `Queue` implementation:

```
/**
* One thread calls push() to put an object on the queue. Another calls pop() to
* get an object off the queue. If there is none, pop() waits until there is
* using wait()/notify(). wait() and notify() must be used within a synchronized
* method or block.
*/
import java.util.*;

public class Queue {
        LinkedList q = new LinkedList();  // Where objects are stored
        public synchronized void push(Object o) {
                q.add(o);          // Append the object at end of the list
                this.notify();     // Tell waiting threads data is ready
       }
        public synchronized Object pop() {
                while(q.size() == 0) {
                        try { this.wait(); }
                catch (InterruptedException e) { /* Ignore this exception */ }
                }
                return q.remove(0);
        }
}
```

Readers/Writers in Monitors: `ReadersWriters` Class

```
class ReadersWriters
{
   private int data = 0;  // our database
   private int nr = 0;

   private synchronized void startRead(){
       nr++;
   }

   private synchronized void endRead(){
       nr--;
       if (nr == 0) notify();  // wake a
   }                          //waiting writer

   public void read ( )   {
       startRead ( );
       System.out.println("read"+data);
       endRead ( );
   }
```

```
   public synchronized void write ( ) {
       while (nr > 0)
         try {
            wait ( ); //wait if any
                  //active readers
         }
         catch (InterruptedException ex){
            return;
         }
       data++;
       System.out.println("write"+data);
       notify (); // wake a waiting writer
   }
}
```

Readers/Writers in Monitors: `ReadersWriters` Class

```
class Reader extends Thread {
    int rounds;
    ReadersWriters RW;

    Reader(int rounds, ReadersWriters RW) {
        this.rounds = rounds;
        this.RW = RW;
    }

    public void run ( ){
        for (int i = 0; i < rounds; i++)
         RW.read ( );
    }
}
```

```
class Writer extends Thread {
    int rounds;
    ReadersWriters RW;

    Writer(int rounds, ReadersWriters RW) {
        this.rounds = rounds;
        this.RW = RW;
    }

    public void run ( ){
        for (int i = 0; i < rounds; i++)
         RW.write ( );
    }
}

class RWProblem {
    static ReadersWriters RW = new
        ReadersWriters ( );

    public static void main(String[] args){
        int rounds = Integer.parseInt
                (args[0], 10);
        new Reader(rounds, RW).start ( );
        new Writer(rounds, RW).start ( );
    }
}
```

- This is the *Reader Preference* Solution. How to make this fair?