Synchronisation in Java - Java Monitor

- Every object and class is logically associated with a monitor
  - the associated monitor protects the variable in the object/class

- The monitor of an object/class can also be associated with the methods in the object/class
  - The synchronized codes will be regarded as indivisible operations
  - The synchronized methods are associated with the monitor of the object/class

- The synchronised codes are called the monitor region
Synchronization of Java threads

- Two types of synchronization
  - mutual exclusion
  - cooperation

- Use keyword “Synchronized” to implement mutual exclusion
  - Synchronized method
  - Synchronized statements

- Use “wait” and “notify” method to achieve cooperation
  - wait, notify, notifyAll
Synchronized methods and statements

→ Synchronized method

```java
public class syn_statement_ex {
    private long c1 = 0;
    private long c2 = 0;

    public synchronized void addc1() { c1++; }
    public synchronized void addc2() { c2++; }
}
```

- addc1 and addc2 methods are associated with the monitor of ClassA

→ Synchronized statements must specify the object that provides the lock

```java
synchronized(object_name) {
    statements;
    ...
}
```

- The statements are associated with the monitor of the object
Synchronized statements

Synchronized statements can be used to improve concurrency with fine-grained synchronization;

see the following example and note the difference if using synchronized methods

```java
public class syn_statement_ex {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void addc1() {
        synchronized(lock1) { c1++; }
    }

    public void addc2() {
        synchronized(lock2) { c2++; }
    }
}
```
Java Monitor

Figure 20-1. A Java monitor.
How to generate threads in Java

- Using one of the two ways to implement threads in Java
  - Either extend the `Threads` class
  - Or implement the associated `Runnable` interface
- Programmer must extend (implement) a `run` method that specifies what the thread should do
- Create a new instance of the class for each thread and invoke the `start` method
- `start` initializes a new thread of control that executes the `run` method
public class MyThread extends java.lang.Thread
{
    int threadNum; // data

    public MyThread(int num) // constructor
    {
        threadNum = num;
    }

    public void run()
    {
        for (int i = 0; i<20; i++)
        {
            System.out.println("Hello from thread "+ threadNum);
        }
    }
}
public static void main(String[ ] args)
{
    System.out.println("Simple Thread Demonstration");
    System.out.println("Extending Thread");
    for (int i = 0; i < 4; i++)
    {
        MyThread newThread = new MyThread(i);
        newThread.start();
    }
}
Example of using Runnable interface

class RunBasicThread implements Runnable{
    char c;
    RunBasicThread(char c) {
        this.c = c;
    }
    // override run() method in interface
    public void run() {
        for(int i=0; i<100; i++) {
            System.out.print(c);
            try{
                Thread.sleep((int)(Math.random() * 10));
            } catch( InterruptedException e ) {
                System.out.println("Exception caught");
            }
        }
    }
    
    public static void main(String[] args) {
        RunBasicThread bt = new RunBasicThread('!');
        RunBasicThread bt1 = new RunBasicThread('*');
        // start RunBasicThread objects as threads
        new Thread(bt).start();
        new Thread(bt1).start();
    }
}
The previous examples show independent, asynchronous threads:

1. each thread has all the data and methods it needs
2. threads run at their own pace, without concern for the progress of others

In many cases threads share data, and rely on the activities of other threads.

Independent threads can run asynchronously if they only use local (private) data, but to share data they must synchronize.

Consider the following producer-consumer example.
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer (CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run () {
        for (int i = 0;  i < 10;  i++)  {
            cubbyhole.put(number, i);
            try  {
                sleep ((int) (Math.random( ) * 100));
            }  catch (InteruptedException e) {  }
        }
    }
}

Generates 0 <= i < 9
Stores it in a Cubbyhole
Sleeps for a while
Generates the next i

public class CubbyHole  {
    private int contents;
    private boolean available = false;

    public int get (int number) {
        … }
    public void put (int number, int value) {
        … }
    … }

public int get (int number) {
{ … }

public void put (int number, int value) {
{ … }
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer (CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run ( ) {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get(number);
        }
    }
}

- Consumes the integers from the Cubbyhole object
- As quickly as they become available
Producer / Consumer Example

- Shared memory (CubbyHole object) example
- Ideally Consumer will get each value only once produced by Producer

- Problem arises if Producer is quicker than Consumer
- Producer generates two numbers before Consumer consumes the first
- E.g. misses the number 4

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer</td>
<td>#1 got: 3</td>
<td></td>
</tr>
<tr>
<td>Producer</td>
<td>#1 put: 4</td>
<td></td>
</tr>
<tr>
<td>Producer</td>
<td>#1 put: 5</td>
<td></td>
</tr>
<tr>
<td>Consumer</td>
<td>#1 got: 5</td>
<td></td>
</tr>
</tbody>
</table>


## Producer / Consumer Example

- If Producer is slower than Consumer we also have a problem
- Consumer gets two numbers before Producer generates second
- E.g. gets the number 4 twice

- Example of a race condition – reading and writing shared data concurrently; success depends on timing
- Need synchronization

<table>
<thead>
<tr>
<th>Producer</th>
<th>#1 put: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer</td>
<td>#1 got: 4</td>
</tr>
<tr>
<td>Consumer</td>
<td>#1 got: 4</td>
</tr>
<tr>
<td>Producer</td>
<td>#1 put: 5</td>
</tr>
</tbody>
</table>
Synchronization of Java Threads

- The `put` and `get` in the CubbyHole code are the so-called critical section (monitor region).

- These should therefore be marked as `synchronized`.

```java
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get(int who) { ... }

    public synchronized void put(int who, int value) { ... }
}
```

- Running synchronised method locks object.
- Unlocked when method terminates.
Synchronization of Java Threads

- So now we have mutual exclusion, but how do Producer and Consumer cooperate?

- Consumer needs to wait until Producer has put something new, and Producer needs to notify Consumer after the data is put— and vice versa

- i.e. `put` and `get` need to wait on and notify each other of their activities

```java
public synchronized int get {
    while (available == false) {
        try { // wait for producer to put value
            wait ();
        } catch (InterruptedException e) { }
    }
    available = false;
    // Notify Producer value has been retrieved
    notifyAll ();
    return contents;
}
```

- available=false means the data has not been put

- Loops until Producer ready with new value

- `wait` relinquishes lock

- When Producer ready to put, it does `notifyAll` and consumer comes out of wait state and get method can collect value
Synchronization of Java Threads

- So now we have locking, but how do Producer and Consumer cooperate?
- Consumer needs to wait until Producer has put something, at which point Producer needs to notify Consumer – and vice versa
- i.e. put and get need to wait on and notify each other of their activities

```java
public synchronized void put (int value) {
    while (available == true) {
        try { // wait for consumer to get value
            wait ();
        } catch (InterruptedException e) { }
    } catch (InterruptedException e) { }
    contents = value;
    available = true;
    // Notify Consumer that value has been sent
    notifyAll ();
}
```

- available=true means the data has been put
- Loops until Producer ready with new value
- wait relinquishes lock
- When Producer ready to put, it does notifyAll and consumer comes out of wait state and get method can collect value
Synchronization of Java Threads

- `notifyAll` wakes up all threads waiting on object in question (CubbyHole)
- Awakened threads compete for lock (ownership of the monitor)
- Those threads that don’t get the lock continue to wait
- `notify` will allow one (rather than all) thread to be woken
Synchronization in Java (Monitors)

- Java provides built-in mutual exclusion and thread synchronization capabilities (monitors).
- They are part of the programming language, so the compiler can generate correct code to implement the mutex locks.
- Only one thread at a time can be active in a monitor.
- The thread with the current lock can signal other waiting threads when the lock is released.
- Higher level and less error prone than forcing programmers to explicitly handle locks and mutexes.

For more information, see:
http://java.sun.com/docs/books/tutorial/essential/threads
Distributed Shared Memory

Distributed Shared Memory (DSM) Systems build the shared memory abstract on top of the distributed memory machines.

The users have a virtual global address space and the message passing underneath is sorted out by DSM transparently from the users.

Then we can use shared memory programming techniques.

Software of implementing DSM
http://www.ics.uci.edu/~javid/dsm/page.html
Three types of DSM implementations

- Page-based technique

- The virtual global address space is divided into equal sized chunks (pages) which are spread over the machines.

- Page is the minimal sharing unit.

- The request by a process to access a non-local piece of memory results in a page fault.

- A trap occurs and the DSM software fetches the required page of memory and restarts the instruction.

- A decision has to be made whether to replicate pages or maintain only one copy of any page and move it around the network.

- The granularity of the pages has to be decided before implementation.
Three types of DSM implementations

- Shared-variable based technique
  - only the variables and data structures required by more than one process are shared.
  - Variable is minimal sharing unit
  - Trade-off between consistency and network traffic
Three types of DSM implementations

- **Object-based technique**
  - memory can be conceptualized as an abstract space filled with objects (including data and methods)
  - Object is minimal sharing unit
  - Trade-off between consistency and network traffic
Summary

Techniques used for Shared Memory Architecture

- Multiprocessing
- User space multithreading
- Operating system-supported (or kernel) multithreading
- Distributed Shared Memory
OpenMP

- OpenMP stands for Open specification for Multi-processing
- used to assist compilers to understand and parallelise the serial code better
- Can be used to specify shared memory parallelism in Fortran, C and C++ programs
- OpenMP is a specification for
  - a set of compiler directives,
  - RUN TIME library routines, and
  - environment variables
- Started mid-late 80s with emergence of shared memory parallel computers with proprietary directive-driven programming environments
- OpenMP is industry standard
OpenMP

- OpenMP specifications include:
  - OpenMP 1.0 for Fortran, 1997
  - OpenMP 1.0 for C/C++, 1998
  - OpenMP 2.0 for Fortran, 2000
  - OpenMP 2.0 for C/C++, 2002
  - OpenMP 2.5 for C/C++ and Fortran, 2005
  - OpenMP 3.0 for C/C++ and Fortran, 2008

- OpenMP Architecture Review Board: Compaq, HP, IBM, Intel, SGI, SUN
OpenMP programming model

An implementation of thread models

Shared Memory, thread-based parallelism

Explicit parallelism

Fork-join model
OpenMP code structure in C

```c
#include <omp.h>

int var1, var2, var3;

Serial code

/*Beginning of parallel section. Fork a team of threads. Specify variable scoping*/

#pragma omp parallel private(var1, var2) shared(var3)
{
    Parallel section executed by all threads

        ...

    All threads join master thread and disband

}

Resume serial code

```
OpenMP code structure in Fortran

PROGRAM HELLO

INTEGER VAR1, VAR2, VAR3

Serial code

...!

Beginning of parallel section. Fork a team of threads. Specify variable scoping

!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)

Parallel section executed by all threads

...!

All threads join master thread and disband

!$OMP END PARALLEL

Resume serial code

...!

END
### OpenMP Directives Format

#### C/C++

<table>
<thead>
<tr>
<th>#pragma omp</th>
<th>directive-name</th>
<th>[clause, ...]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required for all OpenMP C/C++ directives.</td>
<td>A valid OpenMP directive. Must appear after the pragma and before any clauses.</td>
<td>Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.</td>
</tr>
</tbody>
</table>

#### Fortran

<table>
<thead>
<tr>
<th>sentinel</th>
<th>directive-name</th>
<th>[clause ...]</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Fortran OpenMP directives must begin with a sentinel. The accepted sentinels depend upon the type of Fortran source. Possible sentinels are: !$OMP, C$OMP, <em>$OMP</em></td>
<td>A valid OpenMP directive. Must appear after the sentinel and before any clauses.</td>
<td>Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.</td>
</tr>
</tbody>
</table>
OpenMP features

- OpenMP directives are ignored by compilers that don’t support OpenMP, so codes can also be run on sequential machines

- Compiler directives used to specify
  - sections of code that can be executed in parallel
  - critical sections
  - Scope of variables (private or shared)

- Mainly used to parallelize loops, e.g. separate threads to handle separate iterations of the loop

- There is also a run-time library that has several useful routines for checking the number of threads and number of processors, changing the number of threads, etc
Fork-Join Model

- Multiple threads are created using the **parallel** construct

  For C and C++
  
  ```cpp
  #pragma omp parallel
  {
    ... do stuff
  }
  
  For Fortran
  
  !$OMP PARALLEL
  ... do stuff
  !$OMP END PARALLEL
How many threads generated

The number of threads in a parallel region is determined by the following factors, in order of precedence:

- Use of the `omp_set_num_threads()` library function
- Setting of the `OMP_NUM_THREADS` environment variable
- Implementation default - the number of CPUs on a node

Threads are numbered from 0 (master thread) to N-1
Parallelizing loops in OpenMP

- Compiler directive specifies that loop can be done in parallel
  - For C and C++
    ```c
    #pragma omp parallel for
    for (i=0; i++<N) {
        value[i] = compute(i);
    }
    ```
  - For Fortran
    ```fortran
    !$OMP PARALLEL DO
    do (i=1:N)
        value(i) = compute(i);
    end do
    !$OMP END PARALLEL DO
    ```

- Can use thread scheduling to specify partition and allocation of iterations to threads
  ```c
  #pragma omp parallel for schedule(static,4)
  ```
  - `schedule(static [,chunk])`
    Deal out blocks of iterations of size chunk to each thread
  - `schedule(dynamic [,chunk])`
    Each thread grabs a chunk iterations off a queue until all are done
  - `schedule(runtime)` Find schedule from an environment variable `OMP_SCHEDULE`
Synchronisation in OpenMP

Critical construct

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$OMP CRITICAL [ name ]</td>
<td>#pragma omp critical [ name ]</td>
</tr>
<tr>
<td>block</td>
<td>structured_block</td>
</tr>
<tr>
<td>!$OMP END CRITICAL</td>
<td></td>
</tr>
</tbody>
</table>

Barrier construct

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$OMP BARRIER</td>
<td>#pragma omp barrier</td>
</tr>
</tbody>
</table>
Example of Critical Section in OpenMP

```c
#include <omp.h>

main() {
    int x;
    x = 0;

    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        x = x+1;
    }  /* end of parallel section */
}
```
Example of Barrier in OpenMP

```c
#include <omp.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    int th_id, nthreads;
    #pragma omp parallel private(th_id)
    {
        th_id = omp_get_thread_num();
        printf("Hello World from thread %d\n", th_id);
        #pragma omp barrier
        if ( th_id == 0 )
        {
            nthreads = omp_get_num_threads();
            printf("There are %d threads\n",nthreads);
        }
    }
    return 0;
}
```
Data Scope Attributes in OpenMP

- OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped.

- These clauses are used in conjunction with several directives (e.g. PARALLEL, DO/for) to control the scoping of enclosed variables.

- Three often encountered clauses:
  - Shared
  - Private
  - Reduction
Shared and private data in OpenMP

- **private(var)** creates a local copy of var for each thread
- **shared(var)** states that var is a global variable to be shared among threads
- Default data storage attribute is shared

```c
!$OMP PARALLEL DO
 !$OMP& PRIVATE(xx,yy) SHARED(u,f)
 DO j = 1,m
   DO i = 1,n
     xx = -1.0 + dx * (i-1)
     yy = -1.0 + dy * (j-1)
     u(i,j) = 0.0
     f(i,j) = -alpha * (1.0-xx*xx) * & (1.0-yy*yy)
   END DO
 END DO
!$OMP END PARALLEL DO
```
Reduction Clause

- Reduction -

Reduction clause
`reduction (op : var)`

- e.g. add, logical OR. A local copy of the variable is made for each thread. Then local values combined to create global value through Reduction operation.

```c
double ZZ, res=0.0;
#pragma omp parallel for reduction (+:res)
private(ZZ)
for (i=1;i<=N;i++) {
    ZZ = i;
    res = res + ZZ;
}
```
Run-Time Library Routines

- Can perform a variety of functions, including
  - Query the number of threads/thread no.
  - Set number of threads
  - ...

Run-Time Library Routines

- query routines allow you to get the number of threads and the ID of a specific thread

  id = omp_get_thread_num(); // thread no.
  Nthreads = omp_get_num_threads(); // number of threads

- Can specify number of threads at runtime

  omp_set_num_threads(Nthreads);
Environment Variable

→ Controlling the execution of parallel code

→ Four environment variables

- **OMP_SCHEDULE**: how iterations of a loop are scheduled
- **OMP_NUM_THREADS**: maximum number of threads
- **OMP_DYNAMIC**: enable or disable dynamic adjustment of the number of threads
- **OMP_NESTED**: enable or disable nested parallelism
OpenMP compilers

- Since parallelism is mostly achieved by parallelising loops using shared memory, OpenMP compilers work well for multiprocessor SMPs and vector machines.

- OpenMP could work for distributed memory machines, but would need to use a good distributed shared memory (DSM) implementation.

- For more information on OpenMP, see www.openmp.org.