CZ4102 – High Performance Computing

Lecture 10:

Multi-threading

- Dr Tay Seng Chuan

Reference:

(i) "Introduction to Parallel Computing" – Chapter 7.

(ii) "Introduction to Java Programming", Y. Daniel Liang, Quek E&T, ISBN 1-58076-255-7, Chapter 12.

1

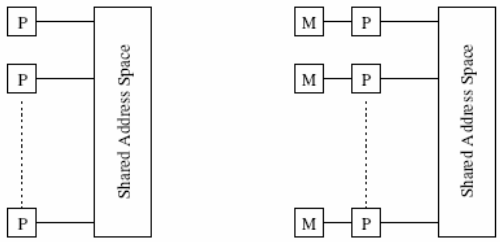
Topic Overview

- Executions of Single Thread and Multiple Threads
- Thread Synchronization and Protocols
- Over Consumption and Deadlock
- Producer-Consumer Paradigm
- JAVA Thread Examples

2

Overview of Programming Models

- Programming models provide support for expressing concurrency and synchronization.
- Process based models assume that all data associated with a process is private by default, unless otherwise specified.
- Lightweight processes and threads assume the existence of global memory.

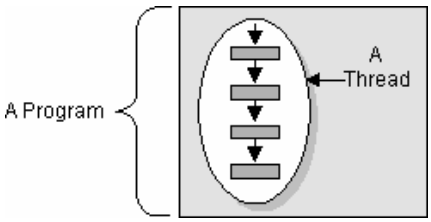


- Directive based programming models extend the threaded model by facilitating creation and synchronization of threads.

3

What Is a Thread?

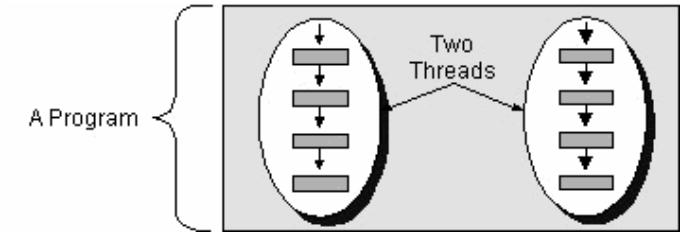
- A sequential program has a beginning, an execution sequence, and an end. At any given time during the runtime of the program there is a single point of execution.
- A thread is similar to the sequential programs but a thread itself is not a program; it cannot run on its own. Rather, it runs within a program. The following figure shows this relationship.



4

Multiple Threads

- Multiple threads can be implemented in a single program running at the same time and performing different tasks. Its execution can be in interleaved manner if it is run on unprocessed platform, or in parallel if more than one processor is available. This is illustrated in the following figure.



5

Thread Example

- A *thread* is a single stream of control in the flow of a program. A program like:

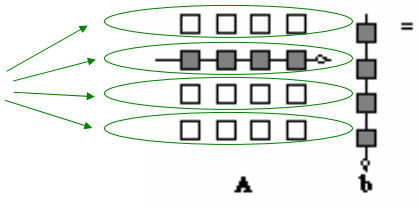
```
for (row = 0; row < n; row++)
  for (column = 0; column < n; column++)
          c[row][column] =
          dot_product( get_row(a, row),
                                  get_col(b, col));
```

can be transformed to:

```
for (row = 0; row < n; row++)
  for (column = 0; column < n; column++)
          c[row][column] =
          create_thread( dot_product(get_row(a, row),
                  get_col(b, col)));
```

In this case, one may think of the thread as an instance of a function that returns before the function has finished executing.

Multiple threads are created.



6

Thread Basics

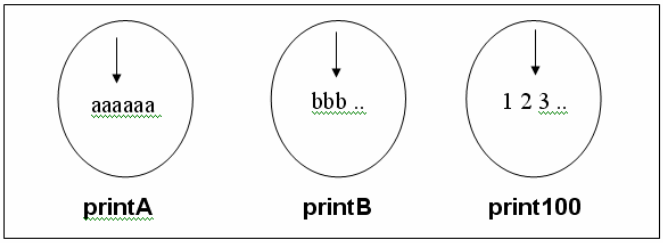
- Threads provide software portability.
- It provides inherent support for latency hiding.
- It facilitates static or dynamic scheduling and load balancing.
- It can simplify programming algorithm.
- It has extensive use in applet programming, eg, concurrent user input and update of screen graphics.

7

Multiple Threads Example

- I am going to show you the first program that contains 3 threads: **printA**, **printB** and **print100**.

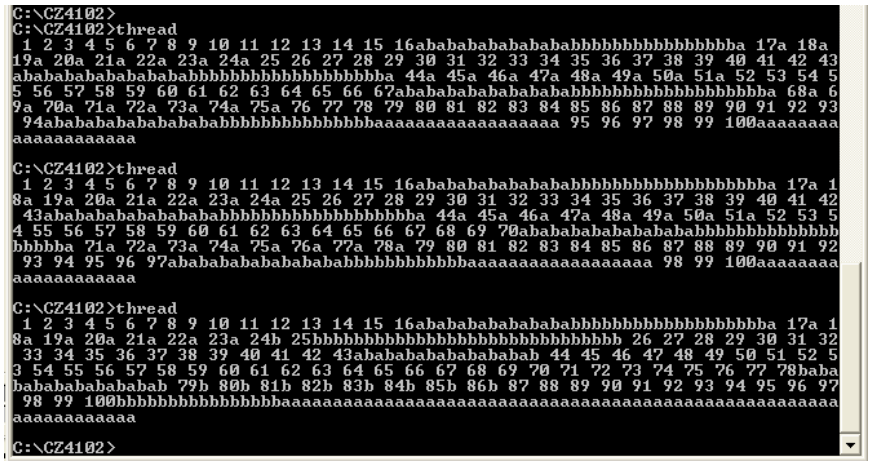
  printA will print a 100 times, **printB** will print b 100 times, and **print100** will print the integers 1, 2 , 3, …, to 100.



- What would you expect from the screen output?

8

Screen Output

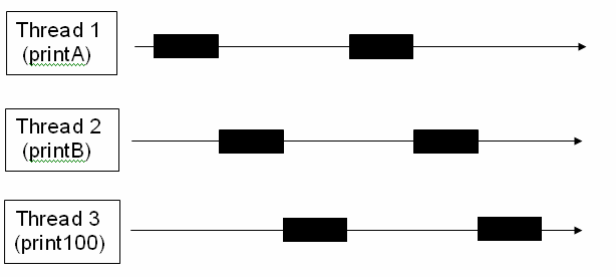


The execution is interleaved and its order changes at different execution run.

9

How are the 3 threads being executed on a uniprocessor?

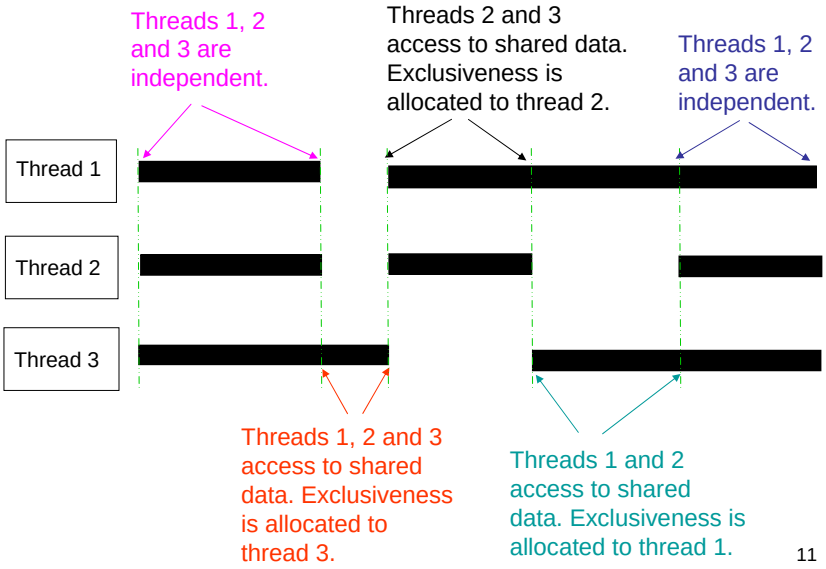
- Since the execution platform contains only one CPU, the threads are executed in concurrent manner as follows:



But at any time only one thread is active. We say the execution sequence of the three threads is interleaved. The order of execution may not be on a round robin basis.

10

What if more than one processor is available (3 processors in this diagram)?



11

Exam Scope:

Your are expected to write C program and MPI program.

You **need not** write java program but you are expected to explain the java code for multithreading (ie, the effect of the multithreading algorithm).

12

Structure of a Threaded Program in JAVA (Concurrent Programming)

```
//User Defined Thread Class
class UserThread extends Thread
{...
   public UserThread()
   {
      ...
   }
      ...
   public void run()
   {
        ...
   }
}
```

```
//Client Class
public class Client
{   ...
    main()
    {UserThread ut = new
                UserThread();
        ...
      ut.start();
        ...
    }
}
```

- The Thread class implements a generic thread that, by default, does nothing. That is, the implementation of its run method is empty. This is not particularly useful, so the Thread class defines API that lets a Runnable object provide a more interesting **run** method for a thread.
- The **run** method gives a thread something to do. Its code implements the thread's running behavior. It can do anything that can be encoded in Java statements.
- To customize what a thread does when it is running, we can subclass (extend or inherit) Thread (itself a Runnable object) and override its empty **run** method so that it will do something.

13

Threaded JAVA Program

```
// TestThreads.java

import java.io.*;    // for System I/O
import java.util.*;  // for StringTokenizer
class TestThreads
{  public static void main (String[] args)
   {
      StringTokenizer stok;
      DataInputStream in = new DataInputStream (System.in);

      //declare and create threads
      PrintChar printA = new PrintChar('a',100);
      PrintChar printB = new PrintChar('b',100);
      PrintNum  print100 = new PrintNum(100);

      //start threads
      print100.start();
      printA.start();
      printB.start();

       String ch = "";
       // hold the screen
       try{
          ch = in.readLine();
        } catch (IOException e) {}
   }
}                          // P.T.O.
```

14

```
//The thread class for printing a specified character in specified times

class PrintChar extends Thread
{  private char charToPrint;  //the character to print
   private int     times;  //the times to repeat

   //The thread class constructor
   public PrintChar(char c, int t)
   {  charToPrint = c;
      times = t;
   }

   //override the run() method to tell the system what the thread will do
   public void run()
   {  for (int i=1; i <= times; i++)
         System.out.print(charToPrint);
   }
}                    // P.T.O.
```

15

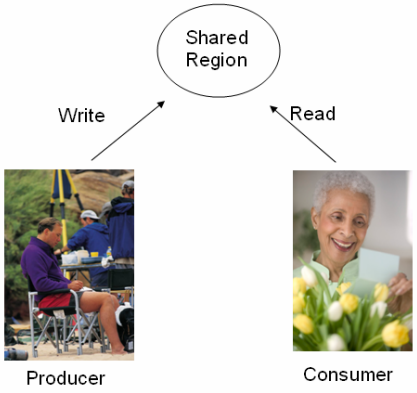
```
//The thread class for printing number from 1 to n for a given n

class PrintNum extends Thread
{  private int lastNum;

   public PrintNum(int i)
   {  lastNum = i;       }

   public void run()
   {  for (int i=1; i <= lastNum; i++)
         System.out.print(" "+i);
   }

}   // end of program
```

16

Producer-Consumer Paradigm

The producer-consumer scenario imposes the following constraints:

- The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
- The consumer threads must not pick up tasks until there is something present in the shared data structure.
- Individual consumer threads should pick up tasks one at a time.



17

Applications of Producer-Consumer Paradigm

- Memory consistency
- Delivery flow
- Parallel/distributed processing – synchronization aspect
- etc …

18

Producer-Consumer Program

```
// shared1.java

import java.io.*;    // for System I/O
import java.util.*;  // for StringTokenizer

class SharedRegion
{
   private int sharedInt = -1;
   private boolean moreData = true;

   public void setSharedInt( int val )
   { sharedInt = val; }

   public int getSharedInt() { return sharedInt; }

   public void setMoreData( boolean b )
   { moreData = b; }

   public boolean hasMoreData() { return moreData; }
}      //P.T.O.
```

19

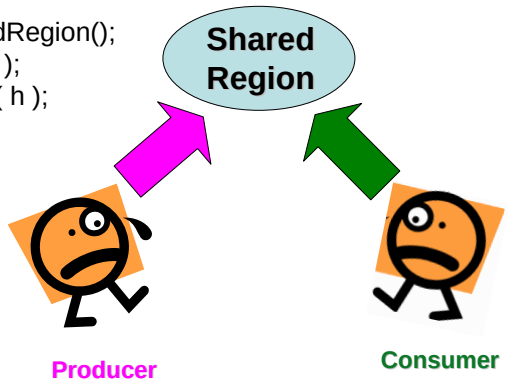
```
class Producer extends Thread
{
   private SharedRegion this1;
   public Producer( SharedRegion h )
   {
      this1 = h;
   }

   public void run()
   {
      for ( int count = 0; count < 10; count++ )
      {
         try {  // sleep for a random interval
              Thread.sleep( (int) ( Math.random() * 3000 ));
            }
         catch( InterruptedException e ) {
            System.err.println( e.toString() );
         }
         this1.setSharedInt( count );
         System.out.println("Producer set sharedInt to " + count );
         System.out.flush();
      }
      this1.setMoreData( false );
   }
}      //P.T.O.
```

20

```
class Consumer extends Thread
{
  private SharedRegion this1;
  public Consumer( SharedRegion h )
  {
    this1 = h;
  }

  public void run()
  {
    int val;
    while ( this1.hasMoreData() )
    {
      // sleep for a random interval
      try {
        Thread.sleep( (int) (Math.random() * 3000));
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }
      val = this1.getSharedInt();
      System.out.println( "Consumer retrieved " + val );
      System.out.flush();
    }
  }
}          //P.T.O.
```

21

```
public class shared1
{
  public static void main( String args[] )
  {
    StringTokenizer stok;
    DataInputStream in = new DataInputStream (System.in);

    SharedRegion h = new SharedRegion();
    Producer p = new Producer( h );
    Consumer c = new Consumer( h );
    p.start();
    c.start();

    String ch = "";
    // hold the screen
    try{
        ch = in.readLine();
      } catch (IOException e) {}
  }
}          // end of shared1.java
```



22

What will be the screen output for shared1.java?

23

Synchronization Protocol 1

Global flags: **Lock_P**, **Lock_C**;
set Lock_P to false (indicate that the shared integer is not used by producer);
set Lock_C to false (indicate that the shared integer is not used by consumers);

Producer

if Lock_C is true, keep waiting until Lock_C is false (if the consumer is using the integer, the producer will have to wait);

set Lock_P to true (indicate that the shared integer will be used by producer);

access the shared integer (write operation);

set Lock_P to false (indicate that the producer has released the shared integer);

Consumer

if Lock_P is true, keep waiting until Lock_P is false (if the producer is using the integer, the consumer will have to wait);

set Lock_C to true (indicate that the shared integer will be used by consumer);

access the shared integer (read operation);

set Lock_C to false (indicate that the consumer has released the shared integer);

24

Effect of the first set of protocol:

25

Synchronization Protocol 2

Global flags: **Lock_P**, **Lock_C**;
set Lock_P to false (indicate that the shared integer is not used by producer);
set Lock_C to false (indicate that the shared integer is not used by consumers);

Producer

set Lock_P to true (indicate that the shared integer will be used by producer);

if Lock_C is true, keep waiting until Lock_C is false (if the consumer is using the integer, the producer will have to wait);

access the shared integer (write operation);

set Lock_P to false (indicate that the producer has released the shared integer);

Consumer

set Lock_C to true (indicate that the shared integer will be used by consumer);

if Lock_P is true, keep waiting until Lock_P is false (if the producer is using the integer, the consumer will have to wait);

access the shared integer (read operation);

set Lock_C to false (indicate that the consumer has released the shared integer);

26

Effect of the second set of protocol:

27

Precautions

If you write a program in which several concurrent threads are competing for resources (e.g., the shared integer), you must take precautions to ensure fairness. A system is fair when each thread gets enough access to limited resource to make reasonable progress. A fair system prevents **starvation** and **deadlock**. **Starvation** occurs when one or more threads in your program are blocked from gaining access to a resource and thus cannot make progress.

Deadlock is the ultimate form of starvation; it occurs when two or more threads are waiting on a condition that cannot be satisfied. Deadlock most often occurs when two (or more) threads are each waiting for the other(s) to do something.

Symmetrical synchronization protocol is very easy to design, but it is very prone to deadlock. A lot of programmers design this type of algorithm.

28

Synchronization of JAVA Thread

When multiple threads attempt to manipulate the same data item, the results can often be incoherent if proper care is not taken to synchronize them.

The Java programming language provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*.

First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

29

JAVA's notify Mechanism

public final void notify()

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods. The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object. This method should only be called by a thread that is the owner of this object's monitor.

A thread becomes the owner of the object's monitor in one of three ways:
- By executing a synchronized instance method of that object.
- By executing the body of a synchronized statement that synchronizes on the object.
- For objects of type Class, by executing a synchronized static method of that class.

30

```
// shared3.java
// with synchronization
// sleep removed in shared3

import java.io.*;    // for System I/O
import java.util.*;  // for StringTokenizer

class SharedRegion
{
   private int sharedInt = -1;
   private boolean moreData = true;
   private boolean writeable = true;
```

// P.T.O.

31

```
public synchronized void setSharedInt( int val )
  {
     while ( !writeable )
     {
        try
        {
           wait();
        }
        catch ( InterruptedException e )
        {
           System.err.println( "Exception: " + e.toString() );
        }
     }
     sharedInt = val;
     writeable = false;
     notify();
  }
```

// P.T.O

32

```
    public synchronized int getSharedInt()
    {
      while ( writeable )
      {
        try
        {
          wait();
        }
        catch ( InterruptedException e )
        {
          System.err.println( "Exception: " + e.toString() );
        }
       }
       writeable = true;
       notify();
       return sharedInt;
     }
   public void setMoreData( boolean b ) { moreData = b; }
   public boolean hasMoreData() { return moreData; }
}      //  end class SharedRegion                    P.T.O.
```

33

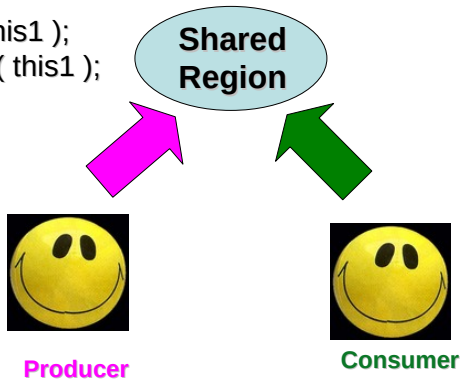
```
class Producer extends Thread
{
  private SharedRegion this1;
  public Producer( SharedRegion h )
  {
    this1 = h;
  }
  public void run()
  {
    for ( int count = 0; count < 10; count++ )
    {

      this1.setSharedInt( count );
      System.out.println( "Producer set sharedInt to " + count );
      System.out.flush();
    }
    this1.setMoreData( false );
  }
}                    // P.T.O.
```

34

```
class Consumer extends Thread
{
   private SharedRegion this1;
   public Consumer( SharedRegion h )
   {
      this1 = h;
   }

   public void run()
   {
      int val;
      while ( this1.hasMoreData() )
      {
         val = this1.getSharedInt();
         System.out.println( "Consumer retrieved " + val );
         System.out.flush();
      }
   }
}                    // P.T.O.
```

35

```
public class shared3
{
   public static void main( String args[] )
   {
      StringTokenizer stok;
      DataInputStream in = new DataInputStream (System.in);
      SharedRegion this1 = new SharedRegion();

      Producer p = new Producer( this1 );
      Consumer c = new Consumer( this1 );

      p.start();
      c.start();

       String ch = "";
       // hold the screen
       try{
          ch = in.readLine();
        } catch (IOException e) {}
   }
}          //end shared3.java
```



36

Output of shared3.java

```
Producer set sharedInt to 0
Consumer retrieved 0
Producer set sharedInt to 1
Consumer retrieved 1
Producer set sharedInt to 2
Consumer retrieved 2
Producer set sharedInt to 3
Consumer retrieved 3
Producer set sharedInt to 4
Consumer retrieved 4
Producer set sharedInt to 5
Consumer retrieved 5
Producer set sharedInt to 6
Consumer retrieved 6
Producer set sharedInt to 7
Consumer retrieved 7
Producer set sharedInt to 8
Consumer retrieved 8
Producer set sharedInt to 9
Consumer retrieved 9
```

37