Lecture 8: *Java Resources* Language

- Introduction
- Description of functionality of *Java Resources* (JR)
- Hello World Implementation
- Bounded Buffer Using Monitors in JR
- JR Virtual Machines
- Remote objects on Virtual Machines in JR
- Producer Consumer with ASMP in JR
- Readers/Writers in JR

Java Resources Language

- As seen above, a concurrent program specifies two or more processes working together to perform a task.
- Each process consists of a sequential program.
- The processes interact by communicating, giving rise to the need for synchronization.
- Communication/synchronization are programmed by reading/ writing shared variables or sending/receiving messages:
  - Shared variables are most appropriate for concurrent code executing on a shared-memory multiprocessor.
  - Message passing is most appropriate for distributed programs that execute on HPC Clusters or networks of workstations.
- Message passing can also be used on shared-memory machines.

Java Resources Language (cont'd)

- JR is an extension of the Java programming language with additional concurrency mechanisms based on those in the SR (Synchronizing Resources) programming language.
- Can also be used as a Distributed Language
- Java has proven to be a clean and simple (and popular) language for object-oriented programming.
- Even so, the standard Java concurrency model is limited.
- As seen above, it provides threads, a primitive monitor-like mechanism, and remote method invocation (RMI).
- Though the basic functionality has been supplemented in Java 5.0, aspects of SR still remain that are difficult to implement using the basic Java package.

Java Resources Language (cont'd)

- JR fills the gap by a richer/more flexible concurrency model than Java.
- JR adapts the following features from SR:
  - *Dynamic Remote Virtual Machine Creation*:
    - JR eliminates RMI's requirement for external setup & interaction with the program.
    - Instead, a JR program can dynamically create Remote VMs on which remote objects can be instantiated.
  - *Dynamic Remote Object Creation*:
    - With JR can populate Remote VMs with new objects through Dynamic Remote Object Creation.
    - Remote objects are created using the familiar "new" expression provided by Java.
  - *Miscellaneous*:
    - RMI, Dynamic Process Creation, support for Rendezvous and Asynchronous Message Passing, are all implemented similar to SR except that in JR, Java classes take the place of SR resources and Java methods take the place of SR `procs`.
    - JR takes a novel object-oriented approach to synchronization whereas SR is not object-oriented (as seen SR is object-based but lacks support for inheritance).

Java Resources Language (cont'd)

- So JR inherits & extends SR's powerful attribute: its expressive power.
- Communication, synchronization mechanisms listed above include the most popular and useful.
- This makes JR suitable for writing concurrent programs for both shared- and distributed-memory applications and machines.
- As well as being expressive, its easy to use for those who know Java.
- Its variety of concurrent programming mechanisms is based on only a few underlying concepts.
- These concepts are generalizations of ones that have been found useful in sequential programs.
- The concurrency mechanisms are also integrated with the sequential ones, so that similar things are expressed in similar ways.
- An important design goal has been to retain the "feel" of Java while providing a richer concurrency model.

Java Resources Language (cont'd)

- JR is implemented on top of Java, so, in principle, it can run on any platform that supports Java.
- This includes networks of workstations and shared-memory multiprocessors.
- JR programs can also be executed on single processor machines, in which case process execution is interleaved.
- The current JR implementation runs on UNIX-based (Linux, Mac OS X, and Solaris) and Windows-based systems.
- Implementation at http://www.cs.ucdavis.edu/~olsson/research/jr
- Also features PySy - a Python package (currently for Python 2.x) based on the JR programming language.
- PySy provides abstractions for expressing several concurrent programming mechanisms, e.g., rendezvous, dynamic process creation, remote method invocation (RMI), and asynchronous message passing.

JR Basics: A "Hello World" Program

```
public class HelloProcess{
   static process Hello = ((int id = 0; id < 25; id++)){
        System.out.println("Hello World from thread " + id);
                        }
   public static void main string args[] { }
}
```

- This Hello World Program kicks off 25 `Hello process`
- First thing to note about the **`process`** is that they are similar but not the same as java threads.
- Confusingly, a multithreaded Java program and a multiprocess JR program both run as a single OS process.
- Instantiating a process is simpler than in Java i.e without need of instantiation, just in a declarative way letting JR handle the rest.
- To declare a **`process`**, JR introduces a new keyword **`process`**
- So a very simple bit of JR code for declaring a JR Process could be:

```
process Hello {System.out.println("It's a Process!");}
```


Bounded Buffer Using Monitors in JR

```
_monitor BoundedBuffer {              // By default signal & continue in JR
    private static final int N = 5;  //Size of the buffer

    _var String[] buffer = new String[N];_var int front;
    _var int rear; _var int count;

    _condvar notFull; _condvar notEmpty;

    _proc void deposit(String data){
        while(count == N){
            _wait(notFull);
        }
        buffer[rear] = data;
        rear = (rear + 1) % N;
        count++;
        _signal(notEmpty);
    }

    _proc String fetch(){
        while(count == 0){
            _wait(notEmpty);
        }
        String result = buffer[front];
        front = (front + 1) % N;
        count--;
        _signal(notFull);
        _return result;
     }
}
```

JR Virtual Machines

- Until now, we've always had concurrent program working in one VM,
- But with JR can declare many VMs on many physical machines.
- A JR VM contains a Java VM and a layer for the JR language.
- Having created some virtual machines, you can specify where an object will be created with a variant of the new operator.
- After that, almost all the development is transparent.
- For example, a `send` operation on an `operation` serviced by an other VM is exactly the same as if there is only one VM.
- An important thing to note is that all the VMs created contain the `static` part of the application. So `static` part is local to the VMs.

```
vm vm1 = new vm();
```

- Creates new VM, `vm1`, on same physical machine where code is run

```
vm vm2 = new wm() on "192.168.1.200";
```

- This creates a new VM on machine with IP Address shown

JR Remote Objects on VMs

- If you want to place an object on a virtual machine, you must define it with the `remote` keyword :

```
remote Person person = new remote Person();
```

- By default, the object is created on the same virtual machine where the code is executed, but you can put it in an other virtual machines :

```
remote Person person = new remote Person() on vm1;
```

- You can also combine the creation of virtual machine and remote objects in one line :

```
remote Person person = new remote Person() on new vm() on "localhost";
```

Producer/Consumer with ASMP in JR

```
class ProducerConsumer {
      private static final int N =12; //Number of producers and consumers

      private static op void deposit (String); // The channel

      public static void main (String… args){
          vm vmConsumer = new vm; // all consumers on one VM
          vm vmProducer = new vm; // all producers on other VM

           for (int i = 0; i <= N; i++) {
                    new remote Consumer on vmConsumer;
                    new remote Producer on vmProducer;
          }
      }
      public static class Consumer {
          private static process Consumer {
                    String data;
                    receive deposit(data);
                    System.out.println("Consumer " + data);
          }
      }
      public static class Producer {
          private static process Producer {
                    send deposit("Producer");
          }
      }
}
```


Readers & Writers in JR

```
public class RW { // operations: abstract communication channels between processes.
      private static op void start_read(); private static op void end_read();
      private static op void start_write(); private static op void end_write();

      private static process RWAllocator {
          int nr = 0, nw = 0; // number of readers (writers) accessing database
          while (true) {     // inni is JR's input statement, doing a multi-way receive.
                             // similar to SR's input statement in
          inni      void start_read() st nw == 0 { nr++; }
          []        void end_read() { nr--; }
          []        void start_write() st nw == 0 && nr == 0 { nw++; }
          []        void end_write() { nw--; }
          }
      }

      private static final int READERS = 4; private static final int WRITERS = 3;
      private static final int RIters = 3; private static final int WIters = 2;

      private static process reader( (int id = 0; id < READERS; id++) ) {
          for (int iters = 1; iters <= RIters; iters++) {
                    call  start_read();  //synch  call:  As  in  Hygenic  Philosophers  in  SR.
                                         //so reader awaits ok from RWAllocator, then reads.
                    send end_read();     //send asynch, continues after invocation sent.
          } System.out.println("reader done "+id); }

      private static process writer( (int id = 0; id < WRITERS; id++) ) {
          for (int iters = 1; iters <= WIters; iters++) {
                    call start_write(); // as in reader above
                    send end_write();
          } System.out.println("writer done "+id); }

      public static void main(String [] args) { } // main needn't do anything since
}                                                 // have statically created processes
```