Programming Language Concepts: Lecture 12

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

http://www.cmi.ac.in/~madhavan/courses/pl2009

PLC 2009, Lecture 12, 04 March 2009

Concurrent Programming

Monitors [Per Brinch Hansen, CAR Hoare]

- Attach synchronization control to the data that is being protected
- Monitor is like a class in an OO language
  - Data definition — to which access is restricted across threads
  - Collections of functions operating on this data — all are implicitly mutually exclusive
- Monitor guarantees mutual exclusion — if one function is active, any other function will have to wait for it to finish

Monitors

```
monitor bank_account{

  double accounts[100];

  boolean transfer (double amount, int source, int target){
    // transfer amount accounts[source] -> accounts[target]
    if (accounts[source] < amount){ return false; }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute the total balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){ balance += accounts[i]; }
    return balance;
  }
}
```

Monitors . . .

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

- Mechanism for a thread to suspend itself and give up the monitor
- A suspended process is waiting for monitor to change its state
- Separate internal queue, as opposed to external queue where initially blocked threads wait
- Dual operation to wake up suspended processes

Monitors . . .

```
boolean transfer (double amount, int source, int target){
  while (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

What happens when a process executes `notify()`?

- Signal and exit — notifying process immediately exits the monitor
- Signal and wait — notifying process swaps roles and goes into the internal queue of the monitor
- Signal and continue — notifying process keeps control till it completes and then one of the notified processes steps in

Monitors . . .

- Makes sense to have more than one internal queue

```
monitor bank_account{

  double accounts[100];

  queue q[100];  // one internal queue for each account

  boolean transfer (double amount, int source, int target){
    while (accounts[source] < amount){
      q[source].wait();  // wait in the queue associated with source
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    q[target].notify();  // notify the queue associated with target
    return true;
  }

}
```

Monitors in Java

- Java implements monitors with a single internal queue
- Monitors incorporated within existing class definitions

Monitors in Java

- Java implements monitors with a single internal queue
- Monitors incorporated within existing class definitions
- Function declared `synchronized` is to be executed atomically
  - Trying to execute a `synchronized` function while another is in progress blocks the second thread into an external queue

Monitors in Java

- Java implements monitors with a single internal queue
- Monitors incorporated within existing class definitions
- Function declared `synchronized` is to be executed atomically
  - Trying to execute a `synchronized` function while another is in progress blocks the second thread into an external queue
- Each object has a lock
  - To execute a `synchronized` method, thread must acquire lock
  - Thread gives up lock when the method exits
  - Only one thread can have the lock at any time

Monitors in Java

- Java implements monitors with a single internal queue
- Monitors incorporated within existing class definitions
- Function declared `synchronized` is to be executed atomically
  - Trying to execute a `synchronized` function while another is in progress blocks the second thread into an external queue
- Each object has a lock
  - To execute a `synchronized` method, thread must acquire lock
  - Thread gives up lock when the method exits
  - Only one thread can have the lock at any time
- `wait()` and `notify()` to suspend and resume
  - `notify()` signals one (arbitrary) waiting process
  - `notifyAll()` signals all waiting processes
  - Java uses signal and continue

Monitors in Java . . .

```
public class bank_account{
  double accounts[100];

  public synchronized boolean
      transfer (double amount, int source, int target){
    while (accounts[source] < amount){ wait(); }
    accounts[source] -= amount; accounts[target] += amount;
    notifyAll();
    return true;
  }

  public synchronized double audit(){
    double balance = 0.0;
    for (int i = 0; i < 100; i++){ balance += accounts[i]; }
    return balance;
  }

  public double current_balance(int i){ // not synchronized!
    return accounts[i];
  }
}
```

Object locks

- Every object has a lock in Java

Object locks

- Every object has a lock in Java
- Can synchronize arbitrary blocks of code

```
public class XYZ{
  Object o = new Object();

  public int f(){
    ..
    synchronized(o){ ... }
  }

  public double g(){
    ..
    synchronized(o){ ... }
    }
  }
}
```

Object locks

- Every object has a lock in Java
- Can synchronize arbitrary blocks of code

```
public class XYZ{
  Object o = new Object();

  public int f(){
    ..
    synchronized(o){ ... }
  }

  public double g(){
    ..
    synchronized(o){ ... }
    }
  }
}
```

- `f()` and `g()` can start in parallel
- Only one of the threads can grab the lock for `o`

Object locks . . .

- Each object has its own internal queue

```
Object o = new Object();

public int f(){
  ..
  synchronized(o){
     ...
     o.wait();   // Wait in queue attached to "o"
     ...
  }
}

public double g(){
  ..
  synchronized(o){
     ...
     o.notifyAll();   // Wake up queue attached to "o"
     ...
  }
}
```

Object locks . . .

- Can convert methods from “externally” synchronized to “internally” synchronized

```
public double h(){
  synchronized(this){
    ...
  }
}
```

Object locks . . .

- Can convert methods from “externally” synchronized to “internally” synchronized

```
public double h(){
  synchronized(this){
    ...
  }
}
```

- “Anonymous” `wait()`, `notify()`, `notifyAll()` abbreviate `this.wait()`, `this.notify()`, `this.notifyAll()`

Object locks . . .

- Actually, `wait()` can be "interrupted" by an `InterruptedException`
- Should write

```
try{
  wait();
}
catch (InterruptedException e) { ... };
```

Object locks . . .

- Actually, `wait()` can be "interrupted" by an `InterruptedException`
- Should write

```
try{
  wait();
}
catch (InterruptedException e) { ... };
```

- Error to use `wait()`, `notify()`, `notifyAll()` outside synchronized method
  - `IllegalMonitorStateException`

Object locks . . .

- Actually, `wait()` can be "interrupted" by an `InterruptedException`
- Should write

```
try{
  wait();
}
catch (InterruptedException e) { ... };
```

- Error to use `wait()`, `notify()`, `notifyAll()` outside synchronized method
  - `IllegalMonitorStateException`
- Likewise, use `o.wait()`, `o.notify()`, `o.notifyAll()` only in block synchronized on `o`

Java threads

- Have a class extend Thread
- Define a function run() where execution can begin in parallel

```
public class Parallel extends Thread{

  private int id;

  public Parallel(int i){ id = i; }

  public void run(){
    for (int j = 0; j < 100; j++){
      System.out.println("My id is "+id);
      try{
        sleep(1000);         // Go to sleep for 1000 ms
      }
      catch(InterruptedException e){}
    }
  }
}
```

Java threads . . .

Invoking threads

```
public class TestParallel {

  public static void main(String[] args){

    Parallel p[] = new Parallel[5];

    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       p[i].start();       // Start off p[i].run() in concurrent thre
    }

}
```

Java threads . . .

Invoking threads

```
public class TestParallel {

  public static void main(String[] args){

    Parallel p[] = new Parallel[5];

    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       p[i].start();       // Start off p[i].run() in concurrent thre
    }

}
```

- `p[i].start()` initiates `p[i].run()` in a separate thread

Java threads . . .

Invoking threads

```
public class TestParallel {

  public static void main(String[] args){

    Parallel p[] = new Parallel[5];

    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       p[i].start();       // Start off p[i].run() in concurrent thre
    }

}
```

- `p[i].start()` initiates `p[i].run()` in a separate thread
  - Directly calling `p[i].run()` does not execute in separate thread!

Java threads . . .

- `sleep(...)` is a static function in `Thread`
  - Argument is time to sleep, in milliseconds
  - Use `Thread.sleep(...)` if current class does not extend `Thread`
  - `sleep(..)` throws `InterruptedException` (like `wait()`)

Java threads . . .

- Cannot always extend `Thread`
  - Single inheritance

Java threads . . .

- Cannot always extend `Thread`
  - Single inheritance
- Instead, implement `Runnable`

```
public class Parallel implements Runnable{  // only this line
                                            //   has changed
  private int id;
  public Parallel(int i){ ... } // Constructor
  public void run(){ ... }

}
```

Java threads . . .

- To use Runnable class, must explicitly create a Thread and start() it

```
public class TestParallel {

  public static void main(String[] args){
    Parallel p[] = new Parallel[5];
    Thread t[]   = new Thread[5];

    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       t[i] = new Thread(p[i]);  // Make a thread t[i] from p[i]
       t[i].start();      // Start off p[i].run() concurrently
                          // Note: t[i].start(), not p[i].start()
    }
  }
}
```

Life cycle of a Java thread

A thread can be in four states

- New: Created but not `start()`ed.
- Runnable: `start()`ed and ready to be scheduled.
  - Need not be actually "running"
  - No guarantee made about how scheduling is done
  - Most Java implementations use time-slicing
- Blocked: not available to run
  - Within `sleep(..)` — unblocked when sleep timer expires
  - Suspended by `wait()` — unblocked by `notify()` or `notfifyAll()`.
  - Blocked on input/output — unblocked when the i/o succeeds.
- Dead: thread terminates.

Interrupts

- One thread can interrupt another using `interrupt()`
  - `p[i].interrupt();` interrupts thread `p[i]`

Interrupts

- One thread can interrupt another using `interrupt()`
  - `p[i].interrupt();` interrupts thread `p[i]`
- Raises `InterruptedException` within `wait()`, `sleep()`

Interrupts

- One thread can interrupt another using `interrupt()`
  - `p[i].interrupt();` interrupts thread `p[i]`
- Raises `InterruptedException` within `wait()`, `sleep()`
- No exception raised if thread is running!

Interrupts

- One thread can interrupt another using `interrupt()`
  - `p[i].interrupt();` interrupts thread `p[i]`
- Raises `InterruptedException` within `wait()`, `sleep()`
- No exception raised if thread is running!
  - `interrupt()` sets a status flag
  - `interrupted()` checks interrupt status and clears the flag

Interrupts

- One thread can interrupt another using `interrupt()`
  - `p[i].interrupt();` interrupts thread `p[i]`
- Raises `InterruptedException` within `wait()`, `sleep()`
- No exception raised if thread is running!
  - `interrupt()` sets a status flag
  - `interrupted()` checks interrupt status and clears the flag
- Detecting an interrupt while running or waiting

```
public void run(){
  try{
    j = 0;
    while(!interrupted() && j < 100){
      System.out.println("My id is "+id);
      sleep(1000);         // Go to sleep for 1000 ms
      j++;
    }
  }
  catch(InterruptedException e){}
}
```

Interrupts

- Check another thread's interrupt status using `interrupted`
  - `t.isInterrupted()` to check status of `t`'s interrupt flag
  - Does not clear flag

Interrupts

- Check another thread's interrupt status using `interrupted`
  - `t.isInterrupted()` to check status of `t`'s interrupt flag
  - Does not clear flag
- `isAlive()` checks running status of a thread
  - `t.isAlive()` is `true` if `t` is Runnable or `Blocked`
  - `t.isAlive()` is `false` if `t` is New or `Dead`

Interrupts

- Check another thread's interrupt status using `interrupted`
  - `t.isInterrupted()` to check status of `t`'s interrupt flag
  - Does not clear flag
- `isAlive()` checks running status of a thread
  - `t.isAlive()` is `true` if `t` is Runnable or `Blocked`
  - `t.isAlive()` is `false` if `t` is New or `Dead`
- Can also `stop()`, `suspend()` and `resume()` a thread, but should not!

An example

- A narrow North-South bridge can accommodate traffic only in one direction at a time.

An example

- A narrow North-South bridge can accommodate traffic only in one direction at a time.
- When a car arrives at the bridge
  1. Cars on the bridge going in the same direction $\Rightarrow$ can cross
  2. No other car on the bridge $\Rightarrow$ can cross (implicitly sets direction)
  3. Cars on the bridge going in the opposite direction $\Rightarrow$ wait for the bridge to be empty

An example

- A narrow North-South bridge can accommodate traffic only in one direction at a time.
- When a car arrives at the bridge
  1. Cars on the bridge going in the same direction $\Rightarrow$ can cross
  2. No other car on the bridge $\Rightarrow$ can cross (implicitly sets direction)
  3. Cars on the bridge going in the opposite direction $\Rightarrow$ wait for the bridge to be empty
- Cars waiting to cross from one side may enter bridge in any order after direction switches in their favour.

An example

- A narrow North-South bridge can accommodate traffic only in one direction at a time.
- When a car arrives at the bridge
  1. Cars on the bridge going in the same direction $\Rightarrow$ can cross
  2. No other car on the bridge $\Rightarrow$ can cross (implicitly sets direction)
  3. Cars on the bridge going in the opposite direction $\Rightarrow$ wait for the bridge to be empty
- Cars waiting to cross from one side may enter bridge in any order after direction switches in their favour.
- When bridge becomes empty and cars are waiting, yet another car can enter in the opposite direction and makes them all wait some more.

An example ...

- Design a class `Bridge` to implement consistent one-way access for cars on the highway synchronization primitives
  - Should permit multiple cars to be on the bridge at one time (all going in the same direction!)

An example . . .

- Design a class `Bridge` to implement consistent one-way access for cars on the highway synchronization primitives
  - Should permit multiple cars to be on the bridge at one time (all going in the same direction!)

- `Bridge` has a public method

```
public void cross(int id, boolean d, int s)
```

  - `id` is identity of car
  - `d` indicates direction
    - `true` is North
    - `false` is South
  - `s` indicates time taken to cross (milliseconds)

An example . . .

```
public void cross(int id, boolean d, int s)
```

- Method `cross` prints out diagnostics
  1. A car is stuck waiting for the direction to change
     `Car 7 going North stuck at Thu Mar 13 23:00:11 IST 2009`
  2. The direction changes
     `Car 5 switches bridge direction to North at Thu Mar 13 23:00:14 IST 2009`
  3. A car enters the bridge.
     `Car 8 going North enters bridge at Thu Mar 13 23:00:14 IST 2003`
  4. A car leaves the bridge.
     `Car 16 leaves at Thu Mar 13 23:00:15 IST 2003`

An example . . .

```
public void cross(int id, boolean d, int s)
```

- Method `cross` prints out diagnostics
  1. A car is stuck waiting for the direction to change
     `Car 7 going North stuck at Thu Mar 13 23:00:11 IST 2009`
  2. The direction changes
     `Car 5 switches bridge direction to North at Thu Mar 13 23:00:14 IST 2009`
  3. A car enters the bridge.
     `Car 8 going North enters bridge at Thu Mar 13 23:00:14 IST 2003`
  4. A car leaves the bridge.
     `Car 16 leaves at Thu Mar 13 23:00:15 IST 2003`
- Use `java.util.Date` to generate time stamps