Objective

Multi-threading

- To understand how multiple threads can execute in parallel
- To learn how to implement threads
- To understand race conditions and deadlocks
- To be able to avoid corruption of shared objects by using locks and conditions
- To be able to use threads for programming animations

Study Chapter 22 (Web Only) : Early Object 7th editon.

Threads

- A thread is a program that is executed independently of other parts of the program
- The Java Virtual Machine executes each thread in the program for a short amount of time (Time Slice)
- This gives the impression of parallel execution

Creating and Running a Thread

Develop a class that implements the `Runnable` interface

```
public interface Runnable
{
   void run();
}
```

Place the code for your task into the `run` method of your class

```
public class MyRunnable implements Runnable
{
   public void run()
   {
      // Task statements go here
      . . .
   }
}
```

Running a Thread

Create an object of your subclass

```
Runnable r = new MyRunnable();
```

Construct a `Thread` object from the runnable object.

```
Thread t = new Thread(r);
```

Call the `start` method to start the thread.

```
t.start();
```

`GreetingRunnable` Outline

```
public class GreetingRunnable implements Runnable
{
   public GreetingRunnable(String aGreeting)
   {
      greeting = aGreeting;
   }

   public void run()
   {
      // Task statements go here
      . . .
   }
   // Fields used by the task statements
   private String greeting;
}
```

Example

Example 1

```
Thu Dec 28 23:12:03 PST 2004 Hello, World!
Thu Dec 28 23:12:04 PST 2004 Hello, World!
Thu Dec 28 23:12:05 PST 2004 Hello, World!
Thu Dec 28 23:12:06 PST 2004 Hello, World!
Thu Dec 28 23:12:07 PST 2004 Hello, World!
Thu Dec 28 23:12:08 PST 2004 Hello, World!
Thu Dec 28 23:12:09 PST 2004 Hello, World!
Thu Dec 28 23:12:10 PST 2004 Hello, World!
Thu Dec 28 23:12:11 PST 2004 Hello, World!
Thu Dec 28 23:12:12 PST 2004 Hello, World!
```

Output

Thread Actions

To wait a thread, use the sleep method of the Thread class

```
sleep(milliseconds)
```

A sleeping thread can generate an **`InterruptedException`**

- Catch the exception
- Terminate the thread

Generic `run` Method

```
public void run()
{
   try
   {
      Task statements
   }
   catch (InterruptedException exception)
   {
   }
   finally{

     //Clean up, if necessary

   }
}
```

File GreetingRunnable.java

```
01: import java.util.Date;
02:
03: /**
04:    A runnable that repeatedly prints a greeting.
05: */
06: public class GreetingRunnable implements Runnable
07: {
08:    /**
09:       Constructs the runnable object.
10:       @param aGreeting the greeting to display
11:    */
12:    public GreetingRunnable(String aGreeting)
13:    {
14:       greeting = aGreeting;
15:    }
16:
17:    public void run()
18:    {
```

File GreetingRunnable.java

```
19:         try
20:         {
21:            for (int i = 1; i <= REPETITIONS; i++)
22:            {
23:               Date now = new Date();
24:               System.out.println(now + " " + greeting);
25:               Thread.sleep(DELAY);
26:            }
27:         }
28:         catch (InterruptedException exception)
29:         {
30:         }
31:      }
32:
33:      private String greeting;
34:
35:      private static final int REPETITIONS = 10;
36:      private static final int DELAY = 1000;
37:   }
```

To Start the Thread

- Construct an object of your runnable class

```
Runnable r = new GreetingRunnable("Hello World");
```

- Then construct a thread and call the `start` method.

```
Thread t = new Thread(r);
t.start();
```

Extend Thread

Another method to create and run a thread:

```
public class MyThread extends Thread{
   public void run(){
      // Task statements go here
   }
}
```

Then construct an object of the class and call start() method.

Check example 2

```
Thread t = new MyThread();
t.start()
```

File GreetingThreadTester.java

```
01: import java.util.Date;
02:
03: /**
04:    This program tests the greeting thread by running two
05:    threads in parallel.
06: */
07: public class GreetingThreadTester
08: {
09:    public static void main(String[] args)
10:    {
11:       Thread t1 = new GreetingRunnable("Hello, World!");
          Thread t2 = new GreetingRunnable("Goodbye, World!");
          t1.start();
          t2.start();
17:    }
18: }
```

What would be the output of this program?

Thread Scheduler

- The thread scheduler runs each thread for a short amount of time (a *time slice*)
- Then the scheduler activates another thread
- There will always be slight variations in running times especially when calling operating system services (e.g. input and output)
- There is no guarantee about the order in which threads are executed

Self Check

- What happens if you change the call to the `sleep` method in the `run` method to `Thread.sleep(1)`?
- What would be the result of the program if the `main` method called

```
r1.run();
r2.run();
```

instead of starting threads?

Answers

- The messages are printed about one millisecond apart.
- The first call to `run` would print 10 "Hello" messages, and then the second call to `run` would print 10 "Goodbye" messages

Pool of Threads

If you have to create many short-lived threads, then use pool of threads.

```
Runnable r1 = new MyThread(...);
Runnable r2 = new MyThread(...);
ExecutorService  pool = Executors.newFixedThreadPool(MAX);
pool.execute(r1);
pool.execute(r2);
```

This is an efficient method of creating threads for a server side program.

Terminating Threads

- A thread terminates when its **`run`** method terminates
- Do not terminate a thread using the deprecated **`stop`** method. Why?
- Instead, notify a thread that it should terminate

```
t.interrupt();
```

- `interrupt` does not cause the thread to terminate–it sets a boolean field in the thread data structure

Terminating Threads

The run method should check occasionally if it has been interrupted

- Use the `interrupted` method
- An interrupted thread should release resources, clean up, and exit

```
public void run()
{
   for (int i = 1;
         i <= REPETITIONS && !Thread.interrupted(); i++)
   {
     // Do work
   }
   // Clean up
}
```

Terminating Threads

The sleep method throws an InterruptedException when a sleeping thread is interrupted

- Catch the exception
- Terminate the thread

```
public void run() {
   try    {
      for (int i = 1; i <= REPETITIONS; i++) {
         //Do work
      }
   }
   catch (InterruptedException exception) {
   }
   finally{
      // Clean up
   }
}
```

Terminating Threads

- Java does not force a thread to terminate when it is interrupted
- It is entirely up to the thread what it does when it is interrupted
- Interrupting is a general mechanism for getting the thread's attention

Self Check

Consider the following runnable.

```
public class MyRunnable implements Runnable  {
   public void run() {
      try {
         System.out.println(1);
         Thread.sleep(1000);
         System.out.println(2);
      }
      catch (InterruptedException exception) {
         System.out.println(3);
      }
      System.out.println(4);
   }
}
```

Self Check

Suppose a thread with this runnable is started and immediately interrupted.

```
Thread t = new Thread(new MyRunnable());
t.start();
t.interrupt();
```

What output is produced?

Answers

The `run` method prints the values 1, 3, and 4.

The call to `interrupt` merely sets the interruption flag, but the **`sleep`** method immediately throws an **`InterruptedException`**.

Note that this interrupt does not terminate the program.

Race Conditions

- When threads share a common object, they can conflict with each other
- Sample program: multiple threads manipulate a bank account
- Create a BankAccount object
- Create two threads:
  - t1 deposits $100 into the bank account for 10 iterations
  - t2 withdraws $100 from the bank account for 10 iterations

Race Conditions

Here is the run method of `DepositRunnable`:

```
public void run(){
   try {
      for (int i = 1; i <= count; i++) {
         account.deposit(amount);
         Thread.sleep(DELAY);
      }
   }
   catch (InterruptedException exception) {
   }
}
```

The `WithdrawRunnable` class is similar

Sample Application

The result should be something like this:

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
. . .
Withdrawing 100.0, new balance is 0.0
```

Example 3

- But sometimes you may find that balance is not correct.
- like this: Why?

```
Depositing 100.0Withdrawing 100.0,
      new balance is 100.0, new balance is -100.0
```

Analysis

Thread: 1

```
deposit(double M){
  newB = b + M;
  b= newB;
}
```

Thread: 2

```
withdraw(double M){
  newB = b - M;
  b  = newB;
}
```

Assume balance (b) of bank account is 100, and M = 100
T1 runs: newB = 200 and then process changes T2 by OS.
T2 runs: b still is 100, and so newB=100-100 = 0, and then process changes changes to T1 by OS.
T1 runs: b=newB ➔ b= 200; process changes changes to T2 by OS.
T2 runs: b = newB ➔ b =0;

Analysis

Assume we add a flag to bank account, and process check this flag if it is being used by other process.

Thread: 1

```
deposit(double M){
  while(flag!=0)
          continue;
  flag = 1;
  newB = b + M;
  b= newB;
  flag = 0;
}
```

Thread: 2

```
withdraw(double M){
 while(flag!=0)
          continue;
  flag = 1;
  newB = b - M;
  b  = newB;
  flag = 0;
}
```

Do you think this will work?

Race Condition

- Occurs if the effect of multiple threads on shared data depends on the order in which they are scheduled
- It is possible for a thread to reach the end of its time slice in the middle of a statement
- It may evaluate the right-hand side of an equation but not be able to store the result until its next turn

Synchronizing Object Access

- To solve this problem such as the one just seen, use a *lock object*
- A lock object is used to control threads that manipulate shared resources
- In Java: **`Lock`** interface and several classes that implement it
  - **`ReentrantLock`**: most commonly used lock class
  - Locks are a feature of Java version 5.0
  - Earlier versions of Java have a lower-level facility for thread synchronization

Synchronizing Object Access

- Typically, a lock object is added to a class whose methods access shared resources, like this:

```
public class BankAccount
{
   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      . . .
   }
   . . .
   private Lock balanceChangeLock;
}
```

Synchronizing Object Access

- Code that manipulates shared resource is surrounded by calls to **`lock`** and **`unlock`**:

```
balanceChangeLock.lock();
//Code that manipulates the shared resource
balanceChangeLock.unlock();
```

Synchronizing Object Access

- If code between calls to **`lock`** and **`unlock`** throws an exception, call to **`unlock`** never happens
- To overcome this problem, place call to `unlock` into a `finally` clause:

Synchronizing Object Access

```
public void deposit(double amount)
{
   balanceChangeLock.lock();
   try
   {
      System.out.print("Depositing " + amount);
      double newBalance = balance + amount;
      System.out.println(", new balance is " + newBalance);
      balance = newBalance;
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

Example 4

Synchronizing Object Access

- When a thread calls `lock`, it owns the lock until it calls `unlock`
- A thread that calls `lock` while another thread owns the lock is temporarily deactivated
- Thread scheduler periodically reactivates thread so it can try to acquire the lock
- Eventually, waiting thread can acquire the lock

Self Check

- If you construct two `BankAccount` objects, how many lock objects are created?
- What happens if we omit the call `unlock` at the end of the `deposit` method?

Answers

- Two, one for each bank account object. Each lock protects a separate balance field.
- When a thread calls `deposit`, it continues to own the lock, and any other thread trying to deposit or withdraw money in the same bank account is blocked forever.

Deadlocks

T1:

```
deposit(double M){
   bLock.lock();
   try   {
      b=b+M;
   }
   finally   {
      bLock.unlock();
   }
}
```

Assume initial balance (b) is 0

T2:

```
withdraw((double M){
   bLock.lock();
   try   {
      while (b<=0)
        continue;
      b=b-M;
   }
   finally   {
      bLock.unlock();
   }
}
```

1. T2 runs, and lock the bLock, but b is zero and it stuck in the loop.
2. T1 runs, and it blocks because T2 locked in.
3. Both threads are deadlocked.

Avoiding Deadlocks

- A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first

Condition Objects

- To overcome problem, use a condition object
- Condition objects allow a thread to temporarily release a lock, and to regain the lock at a later time
- Each condition object belongs to a specific lock object

Condition Objects

You obtain a condition object with `newCondition` method of `Lock` interface

```
public class BankAccount
{
   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      sufficientFundsCondition
         = balanceChangeLock.newCondition();
      . . .
   }
   . . .
   private Lock balanceChangeLock;
   private Condition sufficientFundsCondition;
}
```

Condition Objects

- It is customary to give the condition object a name that describes condition to test
- You need to implement an appropriate test
- As long as test is not fulfilled, call `await` on the condition object:

Condition Objects

```
public void withdraw(double amount)
{
   balanceChangeLock.lock();
   try
   {
      while (balance < amount)
      sufficientFundsCondition.await();
      . . .
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

Condition Objects

- Calling `await`
  - Makes current thread wait
  - Allows another thread to acquire the lock object
- To `unblock`, another thread must execute `signalAll` *on the same condition object*

```
sufficientFundsCondition.signalAll();
```

- `signalAll` unblocks all threads waiting on the condition

Example 5

Condition Objects

- `signalAll`: randomly picks just one thread waiting on the object and unblocks it
- `signal` can be more efficient, but you need to know that every waiting thread can proceed
- Recommendation: always call `signalAll`

Self Check

What is the essential difference between calling `sleep` and `await`?

Why is the `sufficientFundsCondition` object a field of the `BankAccount` class and not a local variable of the `withdraw` and `deposit` methods?

Answers

A sleeping thread is reactivated when the sleep delay has passed. A waiting thread is only reactivated if another thread has called `signalAll` or `signal`.

The calls to `await` and `signal`/`signalAll` must be made *to the same object*.

Animation

Animation can be done much better using threads.

Each object runs independently.

Example 6

Summary

A thread is a program unit that is executed concurrently with other parts of the program.

The start method of the Thread class starts a new thread that executes the run method of the associated Runnable object.

The sleep method puts the current thread to sleep for a given number of milliseconds.

When a thread is interrupted, the most common response is to terminate the run method.

The thread scheduler runs each thread for a short amount of time, called a time slice.

A thread terminates when its run method terminates.

Summary

The run method can check whether its thread has been interrupted by calling the interrupted method.

A race condition occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled.

By calling the lock method, a thread acquires a Lock object. Then no other thread can acquire the lock until the first thread releases the lock.

A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first.

Summary

Calling await on a condition object makes the current thread wait and allows another thread to acquire the lock object.

A waiting thread is blocked until another thread calls signalAll or signal on the condition object for which the thread is waiting.