Multithreading
Introduction

It is often useful for a program to carry out two or more tasks at the same time. A Web browser is a typical example of a multithreaded application. Within a typical browser, you can scroll a page while it’s downloading an applet or an image, play animation and sound concurrently, print a page in the background while you download a new page, or watch three sorting algorithms race to the finish.

All programs have at least one thread. A multithreaded program allows more than one thread to run through the code at once. In some programs, the flow of control splits and executes your program as though it were two or more separate processes, rather than sticking to a single sequential order. FlashGet (A download manager) is an example of this. In this module, you will see how you can implement this behaviour by running tasks in multiple threads, and how you can ensure that the tasks access shared data in a controlled fashion. This module covers the contents of the textbook Chapter 23.

Objectives

• 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

Understand Threads and Multithreading

A thread is a program unit that is executed independently of other parts of the program. A single thread has a beginning, a sequence, and an end. At any given time during the runtime of the thread, there is a single point of execution. However, a thread itself is not a program; a thread cannot run on its own. Rather, it runs within a program. The following figure shows this relationship.

A multithreaded program allows multiple threads to run at the same time and perform different tasks in a single program. This use is illustrated in the next figure.
When a computer contains a single CPU, it can execute only one computer instruction at a time, regardless of its processor speed. When you use a computer with multiple CPU’s (such as a very large mainframe or supercomputer), the computer can execute multiple instructions simultaneously, this is called parallel computing.

Java’s multithreading allows you to use multiple processors to execute threads simultaneously, or to use a single processor to simulate simultaneous execution. In other words, it is the combination of hardware and software that allows multithreading. The following figures illustrate how multithreading executes in a multiprocessor system and in a single-processor system.

Multithreading is used to improve the performance of your Java programs. Multithreaded programs often run faster and are more user-friendly.

### Learning about a Thread’s Life Cycle

Technically, every program you have created is a thread. An application that contains only a main( ) method contains a thread that runs from beginning to end. A thread can be in one of five states during its life:

- New – A thread that has not yet started
- Runnable or ready – A thread is ready to run
- Running – A thread that is running
• Inactive – An inactive thread might be blocked, waiting indefinitely, or waiting for a specified time
• Terminated, finished, or dead – A thread that has completed its run

An application in which you create a thread to run actually contains two threads – the main thread of the application and the thread you created.

A thread can be in only one state at a given point in time. When you create a Thread, it is in the new state; the only method you can use with a new Thread is the method to start it. When you call the Thread’s start( ) method, the Thread enters the ready state. A ready Thread is runnable, which means that it can run. However, a runnable Thread might not be in the running state because the CPU might be busy elsewhere. A runnable Thread cannot run until the CPU allocates some time to it. When the operating system assigns a processor to a thread, it is dispatching the thread. When a Thread begins to execute, it is in the running state. A Thread runs until it becomes inactive or finishes. A Thread enters the inactive state for many reasons. For example, a Thread becomes inactive when you call the Thread’s sleep( ) or suspend( ) method, or it might become inactive if it must wait for another Thread to finish and for the CPU to have available time. When a Thread completes the execution of its run( ) method, it is in the finished or dead state. The isAlive( ) method can be used to check if a Thread is currently alive.

Subclassing Thread and Overriding the run() method

In Java, you can define the task statement of a thread in two ways: By extending a Thread class and overriding its run( ) method or by implementing a Runnable interface. The Runnable interface has only one method – run( ) which you are supposed to implement. This section introduces the first way of creating a Java Thread, i.e. by subclassing a Thread’s superclass. Implementing the Runnable interface will be introduced in the next section.

The Thread class contains a method named run( ). When you extend the Thread class, you override the run( ) method in your extended class to tell the system how to execute the Thread. You also inherit the start( ) method. You use the start( ) method with an instantiated Thread object, it tells the system to start execution of the Thread.

The following example demonstrates how a HelloThread and a Goodbye thread work together by extending the Thread class.

```java
public class HelloThread extends Thread
{
    private static final int TIMES = 100;
    public void run()
    {
        for (int x = 0; x < TIMES; ++x)
            System.out.print(" Hello ");
    }
}
```
public class GoodbyeThread extends Thread
{
    private static final int TIMES = 100;
    public void run ()
    {
        for (int x = 0;x < TIMES; ++x)
        System.out.print("Goodbye ");
    }
}

public class DemoHelloGoodbyeThreads
{
    public static void main(String[] args)
    {
        HelloThread hello = new HelloThread();
        GoodbyeThread bye = new GoodbyeThread();
        hello.start();
        bye.start();
    }
}

If you run above programs several times, you will see that the appearances of “Hello” and “Goodbye” vary from time to time. This is determined by the operating system based on the available resources. You have no guarantee as to the exact pattern of execution; you only guarantee is that the threads execute concurrently.

Implementing the Runnable Interface

A modified example in the following programs demonstrates how a HelloThread and a Goodbye thread work together using a Runnable interface. Actually, the class Thread implements Runnable too.

public class HelloThread implements Runnable
{
    private static final int TIMES = 100;
    public void run()
    {
        for (int x=0;x<TIMES;++x)
        System.out.print(" Hello ");
    }
}
public class GoodbyeThread implements Runnable
{
    private static final int TIMES = 100;
    public void run()
    {
        for(int x=0;x<TIMES;++x)
            System.out.print("Goodbye ");
    }
}

Making your thread a subclass of the Thread class is fine if your thread has no other superclass to inherit. This is because a Java class can only extend one superclass. Otherwise, if your thread has to extend another superclass, you should use implementing the Runnable interface instead.

**Using the sleep () method**

You use the sleep() method to pause a Thread for a specified number of milliseconds. For example, writing sleep(500); within a Thread class’s run() method causes the execution to rest for 500 milliseconds.

You might use the sleep() method to pause a program to give a user time to respond to a question. You need to catch InterruptedException when using the sleep() method.

**Setting Thread Priority**

Every Java Thread has a priority, in terms of preferential access to the operating system’s resources. How programs execute varies in different operating systems because the systems schedule threads differently. **Thread scheduling** is the process by which an operating system allocates processor time to each running thread, such as timesliced threads scheduling (Windows) or non-timesliced (Solaris) scheduling.
In Java, each Thread object’s priority is represented by an integer in the range of 1 to 10. The default priority for a thread object is 5. You can change / find a thread’s priority by using setPriority( ) and getPriority( ) methods. When you run a Java program, the runnable Thread with the highest priority runs first. If several Threads have the same priority, they run in rotation. The methods definition for setPriority( ) and getPriority( ) can be found in the API.

Generally, when ThreadA has higher priority than ThreadB, ThreadA runs and ThreadB waits. However, sometimes Java chooses to run ThreadB to avoid starvation. **Starvation** occurs when a Thread cannot make any progress because of the priorities of other threads (like an infinite loop). The ultimate form of starvation is called deadlock. **Deadlock** occurs when two Threads must wait for each other to do something before either can progress. The DeadLock condition must be avoided.

### Terminating Threads

A thread terminates when its run( ) method terminates. However, if you want to terminate a running thread, to notify a thread that it should clean up and terminate, you use the interrupt ( ) method. Details are available in chapter 23.2.

### Synchronizing Threads and Race Conditions

Sometimes, multiple threads need to use a shared object. For example, if two threads are allowed to schedule a meeting on your calendar after determining that 10:30 on Tuesday is an available time slot, or if two threads update your bank account balance with automatic payroll deposits from two part-time jobs you hold. Unless shared objects are managed correctly, errors and unpredictable results can occur. To manage threads and prevent this sort of inconsistency, you can give exclusive use of a shared object to one Thread at a time. In other words, one thread has access to an object while all the other threads are kept waiting. This process of mutual exclusion is called **thread synchronization**. Synchronization is needed when multiple threads share a common resource.

In the traditional Producer-Consumer example (see diagram below), a Producer thread produces numbers repeatedly, puts the numbers into a shared CubbyHole object. A Consumer thread gets the numbers from the CubbyHole object. Consumer ideally should get each value produced by the Producer once and only once.

Assume for a moment that these two threads make no arrangements for synchronization, the potential problems that might arise:
1. When the Producer is quicker than the Consumer and generates two numbers before the Consumer has a chance to consume the first one. In this situation, the Consumer misses a number.

2. When the Consumer is quicker than the Producer and consumes the same value twice. Either way, the result is wrong because the Consumer should get each number produced by the Producer exactly once. A problem such as this is called a **race condition**. A **race condition** is a situation in which two or more threads or processes are reading or writing some shared data, and the final result depends on the timing of how the threads are scheduled. Race conditions can lead to unpredictable results and subtle program bugs.

The activities of the Producer and the Consumer can be synchronized in two ways.

1. The two threads must not simultaneously access the shared-object. A thread can prevent this from happening by **locking an object**. When an object is locked by one thread and another thread tries to call a synchronized method on the same object, the second thread will block until the object is unlocked.

2. The two threads can do some simple coordination. That is, the Producer must have a way to indicate to the Consumer that the value is ready, and the Consumer must have a way to indicate that the value has been retrieved. The Object class provides a collection of methods — wait, notify, and notifyAll — to help threads wait for a condition and notify other threads when that condition changes.

**Locking an Object**

Locking an object before Java5 is managed by a **synchronized** keyword for a method (see advance topic 23.2). From Java 5, A Lock object and its lock() and unlock() methods can be used (chapter section 23.4).

In the Producer-Consumer example, the Consumer should not access the CubbyHole when the Producer is changing it, and the Producer should not modify it when the Consumer is getting the value. So `put` and `get` methods in the CubbyHole class should be locked and unlocked.

Here’s a code skeleton for the CubbyHole class:

```java
public class CubbyHole
{
    public int get(int who)
    {
        // lock object
        // do something ...
        // unlock object
    }
    public void put(int who, int value)
    {
        // lock object
        // do something ...
        // unlock object
    }
}
```
The acquisition and release of a lock is done automatically and atomically by the Java run-time system. This ensures that race conditions cannot occur in the underlying implementation of the threads, thus ensuring data integrity.

Examine the Producer-Consumer programs below.

```java
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);
            System.out.println("Producer "+ number + " put "+ i);
        }
    }
}

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);
        }
    }
}
```
import java.util.concurrent.locks.*;

public class CubbyHole
{
    private int contents;
    private Lock theLock;

    public CubbyHole()
    {
        theLock = new ReentrantLock();
    }
    public int get(int who)
    {
        theLock.lock();
        try
        {
            return contents;
        }
        finally
        {
            theLock.unlock();
        }
    }
    public void put(int who, int value)
    {
        theLock.lock();
        try
        {
            System.out.println("Consumer "+ who + " got "+ value);
        }
        finally
        {
            theLock.unlock();
        }
    }
}

public class ProducerConsumerTest
{
    public static void main(String[] args)
    {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}
A lock object can be used for handling race condition problems. **Lock** is an interface in the API, the ReentrantLock class is the most commonly used lock class that implements a Lock interface. Typically, a lock object is added to a class whose methods accesses shared resources. The methods lock( ) and unlock( ) can be used to manipulate the shared data. Usually, you need to add a finally clause to avoid any exceptions. The text book (page 879, 880) has an example about how to use them.

**Avoiding DeadLocks**

A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first. This is like an infinite loop. Since Java 5, a **condition** object allows a thread to temporarily release a lock, so that another thread can proceed and to regain the lock at a later time. Each condition object belongs to a specific lock object. You obtain a condition object with the newCondition method of the Lock object.

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

Some methods that you need to investigate:

- await
- signalAll

**Reading**

Text book:
Chapter 23: Multithreading

**Review questions**

Review exercises:
Page 901: Exercise R23.3, R23.4, R23.7, R23.9, R23.10

Programming exercises:
Page 901: Exercise P23.1, P23.9

**References**

Solution to Programming Exercises

P23-1

ExP23_1/ConsumerRunnable.java
/**
 * This class will keep removing things from the queue
 */
public class ConsumerRunnable implements Runnable
{
    /**
     * Constructs the consumer with a queue and count.
     * @param aQueue the queue that the consumer going to consume from
     * @param count the number of time that consumer going to consume
     */
    public ConsumerRunnable(Queue aQueue, int aCount)
    {
        count = aCount;
        queue = aQueue;
    }

    public void run()
    {
        try
        {
            for (int i = 0; i < count && !Thread.interrupted(); i++)
            {
                System.out.println("Consumer: " + queue.remove());
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException exception)
        { }
    }

    private Queue queue = null;
    private int count;
    private static final int DELAY = 1000;
}

ExP23_1/ExP21_1.java
public class ExP21_1
{
    public static void main(String args[])
    {
        Queue q = new Queue(QUEUE_MAX_SIZE);
        ProducerRunnable producer = new ProducerRunnable(q, NUM_OF_ITERATION);
        ConsumerRunnable consumer = new ConsumerRunnable(q, NUM_OF_ITERATION);
        Thread t1 = new Thread(producer);
        Thread t2 = new Thread(consumer);
        t1.start();
        t2.start();
    }

    private static int QUEUE_MAX_SIZE = 10;
    private static int NUM_OF_ITERATION = 1000;
ExP23_1/ProducerRunnable.java

```java
import java.util.Date;

/**
 * This class will keep putting things into the queue.
 */
public class ProducerRunnable implements Runnable {
    /**
     * Constructs the producer with a queue and count.
     * @param aQueue the queue that the consumer going to consume from
     * @param count the number of time that consumer going to consume
     */
    public ProducerRunnable(Queue aQueue, int aCount) {
        count = aCount;
        queue = aQueue;
    }

    public void run() {
        try {
            for (int i = 0; i < count && !Thread.interrupted(); i++) {
                String item = new Date().toString();
                System.out.println("Producer: " + item);
                queue.add(item);
                Thread.sleep(DELAY);
            }
        } catch (InterruptedException exception) {
        }
    }

    private Queue queue = null;
    private int count;
    private static final int DELAY = 1000;
}
```

ExP23_1/Queue.java

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.ArrayList;

/**
 * This class implements the FIFO queue data structure.
 */
public class Queue {
    /**
     * Constructs the maximum size of the queue to deafult size.
     */
    public Queue() {
        size = DEAFULT_SIZE;
```
queue = new ArrayList(size);
queueLock = new ReentrantLock();
queueChangeCondition = queueLock.newCondition();
}

/**
 * Constructs the maximum size of the queue to the given size.
 * @param aSize the maximum size of the queue
 */
public Queue(int aSize)
{
    size = aSize;
    queue = new ArrayList(size);
    queueLock = new ReentrantLock();
    queueChangeCondition = queueLock.newCondition();
}

/**
 * Adds a string into the queue.
 * @param item the item to add
 */
public void add(String item) throws InterruptedException
{
    queueLock.lock();
    try
    {
        while (isFull())
            queueChangeCondition.await();
        queue.add(item);
        queueChangeCondition.signalAll(); // notify all others that the queue is not empty
    }
    finally
    {
        queueLock.unlock();
    }
}

/**
 * Removes one item from the queue.
 * @return the first item of the queue
 */
public String remove() throws InterruptedException
{
    String element = null;
    queueLock.lock();
    try
    {
        while (isEmpty())
            queueChangeCondition.await();
        String element = queue.remove(0).toString();
        queueChangeCondition.signalAll(); // notify all others that the queue is not full
    }
    finally
    {
        queueLock.unlock();
    }
    return element;
}

/**
Check if the queue is empty.
@return true if the queue is empty, else false
*/
public boolean isEmpty()
{
    return queue.isEmpty();
}
/**
    Check if the queue is full.
    @return true if queue size equals to maximum size, else false
    */
public boolean isFull()
{
    return queue.size() == size;
}

private ArrayList queue;
private int size = 0;
private static final int DEFAULT_SIZE = 10;
private Lock queueLock;
private Condition queueChangeCondition;

ExP23_2/
/**
 * This class will keep removing things from the queue
 */
public class ConsumerRunnable implements Runnable
{
    /**
     * Constructs the consumer with a queue and count.
     * @param aQueue the queue that the consumer going to consume from
     * @param count the number of time that consumer going to consume
     */
    public ConsumerRunnable(Queue aQueue, int aCount)
    {
        count = aCount;
        queue = aQueue;
    }

    public void run()
    {
        try
        {
            for (int i = 0; i < count && !Thread.interrupted(); i++)
            {
                System.out.println("Consumer[" + hashCode() + "]: " + queue.remove());
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException exception)
        {
        }
    }

    private Queue queue = null;
    private int count;
    private static final int DELAY = 1000;
}
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * A bank account has a balance that can be changed by deposits and withdrawals.
 */
public class BankAccount
{
    /**
     * Constructs a bank account with a zero balance.
     */
    public BankAccount()
    {
        balance = 0;
        balanceChangeLock = new ReentrantLock();
        sufficientFundsCondition = balanceChangeLock.newCondition();
        lessThanMaxBalanceCondition = balanceChangeLock.newCondition();
    }

    /**
     * Deposits money into the bank account.
     * @param amount the amount to deposit
     */
    public void deposit(double amount) throws InterruptedException
    {
        balanceChangeLock.lock();
        try
        {
            while (balance + amount > MAX_BALANCE)
            {
                lessThanMaxBalanceCondition.await();
                System.out.print("Depositing "+ amount);
                double newBalance = balance + amount;
                System.out.println(" , new balance is "+ newBalance);
                balance = newBalance;
                sufficientFundsCondition.signalAll();
            }
        }
        finally
        {
            balanceChangeLock.unlock();
        }
    }

    /**
     * Withdraws money from the bank account.
     * @param amount the amount to withdraw
     */
    public void withdraw(double amount)
throws InterruptedException
{
    balanceChangeLock.lock();
    try
    {
        while (balance < amount)
        {
            sufficientFundsCondition.await();
            System.out.print("Withdrawing "+ amount);
            double newBalance = balance - amount;
            System.out.println(",
    new balance is "+ newBalance);
            balance = newBalance;
        }
    }
    finally
    {
        balanceChangeLock.unlock();
    }
}
/**
 * Gets the current balance of the bank account.
 * @return the current balance
 */
public double getBalance()
{
    return balance;
}

public static final double MAX_BALANCE = 100000;

private double balance;
private Lock balanceChangeLock;
private Condition sufficientFundsCondition;
private Condition lessThanMaxBalanceCondition;
}

ExP23_9/BankAccountThreadTester.java
/**
 * This program runs four threads that deposit and withdraw
 * money from the same bank account.
 */
public class BankAccountThreadTester
{
    public static void main(String[] args)
    {
        BankAccount account = new BankAccount();
        final double AMOUNT = 100;
        final int REPETITIONS = 1000;
        final int DEPOSIT_THREADS = 1000;
        final int WITHDRAW_THREADS = 20;

        for (int i = 0; i < DEPOSIT_THREADS; i++)
        {
            DepositRunnable d = new DepositRunnable(account, AMOUNT * 5,
            REPETITIONS);
            Thread t = new Thread(d);
            t.start();
        }

        for (int i = 0; i < WITHDRAW_THREADS; i++)
        {
WithdrawRunnable d = new WithdrawRunnable(account, AMOUNT, REPETITIONS);
Thread t = new Thread(d);
t.start();
}
}

ExP23_9/DepositRunnable.java
/**
 * A deposit runnable makes periodic deposits to a bank account.
 */
public class DepositRunnable implements Runnable
{
    /**
     * Constructs a deposit runnable.
     * @param anAccount the account into which to deposit money
     * @param anAmount the amount to deposit in each repetition
     * @param aCount the number of repetitions
     */
    public DepositRunnable(BankAccount anAccount, double anAmount,
             int aCount)
    {
        account = anAccount;
        amount = anAmount;
        count = aCount;
    }

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

    private static final int DELAY = 1;
    private BankAccount account;
    private double amount;
    private int count;
}

ExP23_9/WithdrawRunnable.java
/**
 * A withdraw runnable makes periodic withdrawals from a bank account.
 */
public class WithdrawRunnable implements Runnable
{
    /**
     * Constructs a withdraw runnable.
     * @param anAccount the account from which to withdraw money
     * @param anAmount the amount to deposit in each repetition
     * @param aCount the number of repetitions
     */
    public WithdrawRunnable(BankAccount anAccount, double anAmount,
             int aCount)
public void run()
{
    try
    {
        for (int i = 1; i <= count; i++)
        {
            account.withdraw(amount);
            Thread.sleep(DELAY);
        }
    }
    catch (InterruptedException exception) {}
}

private static final int DELAY = 1;
private BankAccount account;
private double amount;
private int count;