2019

Further Java

CAMBRIDGE COMPUTER SCIENCE TRIPOS PART IB, PAPER 4

ASHWIN AHUJA

Table of Contents

External Sorting Algorithms

Sorting algorithms which sort data larger than the size of the RAM.

Writing to File

File Pointer: Offset at which a read or write occurs – supported by *java.io.RandomAccessFile* – instance of this class keeps track of a specific byte offset from the start of the file.

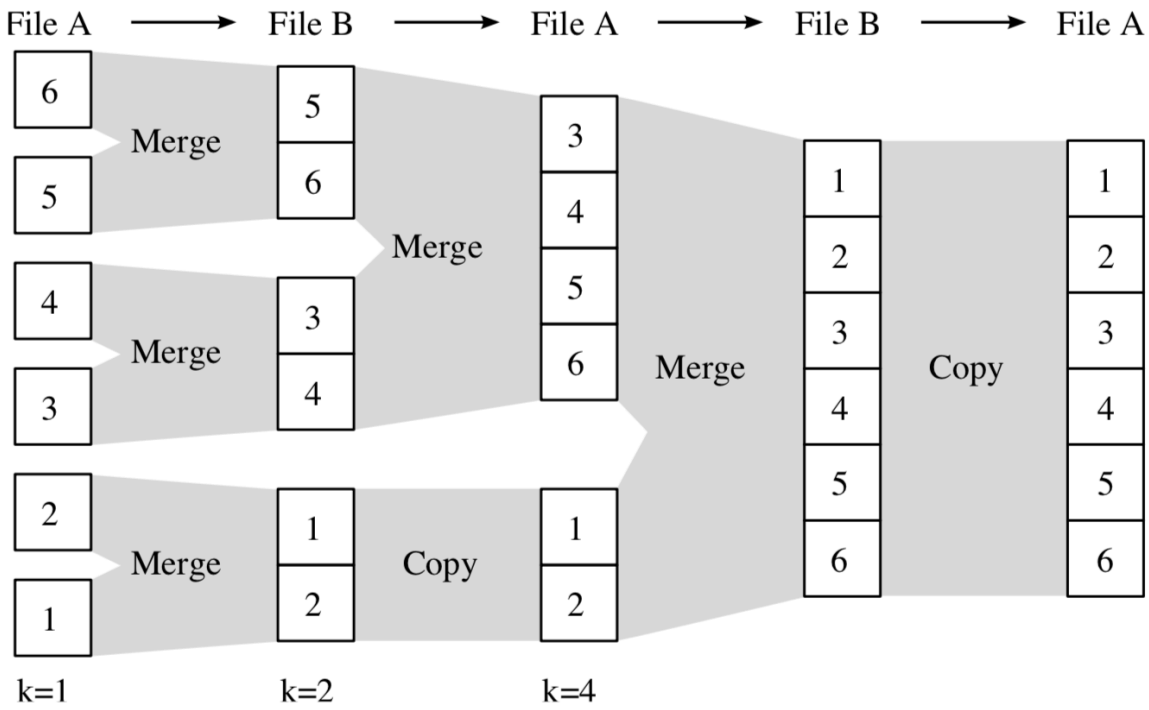
```
RandomAccessFile f = new RandomAccessFile("/home/arb33/example","rw");
f.writeInt(1); //write a "1" into the first four bytes of the file
f.writeInt(2); //write the value "2" after the value "1"
f.writeInt(3); //write the value "3" after the value "2"
f.seek(4); //file pointer now between fourth and fifth byte
System.out.println("Read four bytes as an int value "+f.readInt());
System.out.println("The file is "+f.length()+" bytes long");
```

This is very slow because every *writeInt* has to write to the hard disk. Can coalesce these into an output stream. Can also be wrapped in a *BufferedOutputStream* to create a small in-memory buffer. This can then be wrapped in a *DataOutputStream* to add support for writing primitive integers.

```
RandomAccessFile f = new RandomAccessFile("/home/arb33/example","rw");
DataOutputStream d = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream(f.getFD())));
d.writeInt(1); //write calls now only store primitive ints in memory
d.writeInt(2);
d.writeInt(3);
d.flush(); //force the contents to be written to the disk. Important!
f.seek(4);
System.out.println("Read four bytes as an int value "+f.readInt());
System.out.println("The file is "+f.length()+" bytes long");
```

External Merge Sort

Divide integers in input file into n sorted blocks, each of size k, where k is initially equal to 1 and n is equal to the number of integers stored in the file. On each pass through the file, data held in odd blocks can be merged with data held in even blocks to create n/2 blocks each of length 2k. On each pass, data read from one file and written to the other.



Connecting to the Internet

Java.net.Socket: Implements client sockets – endpoint for communication between two machines. An application can configure itself to create sockets.

1. Server instantiates a ServerSocket (java.net.ServerSocket) object, denoting which port number communication should occur on
2. Server invokes the accept() method of the ServerSocket class – waits until a client connects
3. Client instantiates a Socket class, specifying server name and port number
4. If connection established, client not has Socket object capable of communicating with server
5. On server side, accept() returns a reference to a new socket on the server that is connected to the client's object

Communication through I/O streams – each socket has both an Input and OutputStream (getInputStream() and getOutputStream()). InputStream has an associated read method which will pause execution until some data is available.

Server

```
// File Name GreetingServer.java
import java.net.*;
import java.io.*;

public class GreetingServer extends Thread {
   private ServerSocket serverSocket;

   public GreetingServer(int port) throws
IOException {
      serverSocket = new ServerSocket(port);
      serverSocket.setSoTimeout(10000);
   }
```

Client

```
// File Name GreetingClient.java
import java.net.*;
import java.io.*;

public class GreetingClient {

   public static void main(String [] args)
{
      String serverName = args[0];
      int port = Integer.parseInt(args[1]);
      try {
         System.out.println("Connecting to
" + serverName + " on port " + port);
```

```
   public void run() {
      while(true) {
         try {
            System.out.println("Waiting for client
on port " +
               serverSocket.getLocalPort() +
"...");
            Socket server = serverSocket.accept();

            System.out.println("Just connected to
" + server.getRemoteSocketAddress());
            DataInputStream in = new
DataInputStream(server.getInputStream());

            System.out.println(in.readUTF());
            DataOutputStream out = new
DataOutputStream(server.getOutputStream());
            out.writeUTF("Thank you for connecting
to " + server.getLocalSocketAddress()
               + "\nGoodbye!");
            server.close();

         } catch (SocketTimeoutException s) {
            System.out.println("Socket timed
out!");
            break;
         } catch (IOException e) {
            e.printStackTrace();
            break;
         }
      }
   }

   public static void main(String [] args) {
      int port = Integer.parseInt(args[0]);
      try {
         Thread t = new GreetingServer(port);
         t.start();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

```
         Socket client = new
Socket(serverName, port);

         System.out.println("Just connected
to " + client.getRemoteSocketAddress());
         OutputStream outToServer =
client.getOutputStream();
         DataOutputStream out = new
DataOutputStream(outToServer);

         out.writeUTF("Hello from " +
client.getLocalSocketAddress());
         InputStream inFromServer =
client.getInputStream();
         DataInputStream in = new
DataInputStream(inFromServer);

         System.out.println("Server says "
+ in.readUTF());
         client.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

MulticastSocket: supports multicast communications, sending the data to many computers simultaneously.

DatagramSocket: Unreliable mechanism for sending packets of data between two computers on the internet.

Threads

Threads are created either by inheriting from the client java.lang.Thread or implementing the interface java.lang.Runnable. In either case you place the code which runs inside the new thread inside a method with the following prototype: *public void run(); .start()* then the body of the method run is executed concurrently. Socket object has to be described as final.

```
// Java code for thread creation by implementing
// the Runnable Interface
```

```
class MultithreadingDemo implements Runnable
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " +
                                Thread.currentThread().getId() +
                                " is running");

        }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            Thread object = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

Serialisation

Offers an easy way to save and restore data by enabling an instance of a Java object to be turned into a platform-independent sequence of bytes which can be written to the hard disk or sent over a computer network.

Can serialise any object which implements *java.io.Serializable* interface – marker interface. The object is created by the Java runtime. **Objects related to state – such as Socket class** cannot be serialised – can ignore objects which are unserialisable inside an object to be serialised, can declare the field to be transient. We can control the version identifier given to a class by declaring the following field inside the class and providing a specific integer value: *Private static final long serialVersionUID = …*

ObjectOutputStream and ObjectInputStream: helps you serialise an object into bytes and deserialise bytes into an object

```
FileOutputStream fos = new FileOutputStream("message.jobj");
ObjectOutputStream out = new ObjectOutputStream(fos);
out.writeObject(new Message(1,"Hello, world"));
out.close();
```

Class Loaders and Reflection

Class Loader is the part of the JVM responsible for load class definitions – by default, class loader looks for .class files at various locations on disk and inside Jar files when an instance of a class is first referenced. Issue with ObjectOutputStream is having a common class definition to deserialise things.

Can extend the Java class loader to dynamically load the definition of a new class at runtime into the JVM. With the new class loaded into the JVM, it is then possible to deserialise an instance of the new class.

Reflection: can be used to get information about:

1. Class: getClass()
2. Constructors: getConstructors()
3. Methods: getMethods()

```
// A simple Java program to demonstrate the use of reflection
import java.lang.reflect.Method;
import java.lang.reflect.Field;
import java.lang.reflect.Constructor;

// class whose object is to be created
class Test
{
    // creating a private field
    private String s;

    // creating a public constructor
    public Test() { s = "Hello"; }

    // Creating a public method with no arguments
    public void method1() {
        System.out.println("The string is " + s);
    }

    // Creating a public method with int as argument
    public void method2(int n) {
        System.out.println("The number is " + n);
    }

    // creating a private method
    private void method3() {
        System.out.println("Private method invoked");
    }
}
```

```
class Demo
{
    public static void main(String args[]) throws Exception
    {
        // Creating object whose property is to be checked
        Test obj = new Test();

        // Creating class object from the object using
        // getclass method
        Class cls = obj.getClass();
        System.out.println("The name of class is " +
                           cls.getName());

        // Getting the constructor of the class through the
        // object of the class
        Constructor constructor = cls.getConstructor();
        System.out.println("The name of constructor is " +
                           constructor.getName());

        System.out.println("The public methods of class are : ");

        // Getting methods of the class through the object
        // of the class by using getMethods
        Method[] methods = cls.getMethods();

        // Printing method names
        for (Method method:methods)
            System.out.println(method.getName());

        // creates object of desired method by providing the
        // method name and parameter class as arguments to
        // the getDeclaredMethod
        Method methodcall1 = cls.getDeclaredMethod("method2",
                                                   int.class);

        // invokes the method at runtime
        methodcall1.invoke(obj, 19);

        // creates object of the desired field by providing
        // the name of field as argument to the
        // getDeclaredField method
        Field field = cls.getDeclaredField("s");

        // allows the object to access the field irrespective
        // of the access specifier used with the field
        field.setAccessible(true);

        // takes object and the new value to be assigned
        // to the field as arguments
        field.set(obj, "JAVA");
```

```
        // Creates object of desired method by providing the
        // method name as argument to the getDeclaredMethod
        Method methodcall2 = cls.getDeclaredMethod("method1");

        // invokes the method at runtime
        methodcall2.invoke(obj);

        // Creates object of the desired method by providing
        // the name of method as argument to the
        // getDeclaredMethod method
        Method methodcall3 = cls.getDeclaredMethod("method3");

        // allows the object to access the method irrespective
        // of the access specifier used with the method
        methodcall3.setAccessible(true);

        // invokes the method at runtime
        methodcall3.invoke(obj);
    }
}
```

Annotations

Annotations provide meta-data and are often used to (1) provide information to the compiler, (2) control compilation and deployment or (3) aid program execution. Definition or use is always prefixed with '@'

Typically, they are made up of name-value pairs. In order to use annotations, they must first be defined, as per the example below. Java also supports three built-in annotations:

1. @Deprecated – generate warning on use of this component
2. @Override – generate error if does not override something
3. @SuppressWarnings – suppress warnings

```
public @interface FurtherJavaPreamble {
 enum Ticker {A, B, C, D};
 String author();
 String date();
 String crsid();
 String summary();
 Ticker ticker();
}
```

```
@FurtherJavaPreamble(
author = "Maurice V. Wilkes",
date = "4th May 1949",
crsid = "mvw1",
summary = "Calculate the table of
squares between 1 and 99",
ticker = FurtherJavaPreamble.Ticker.A)
```

Concurrency

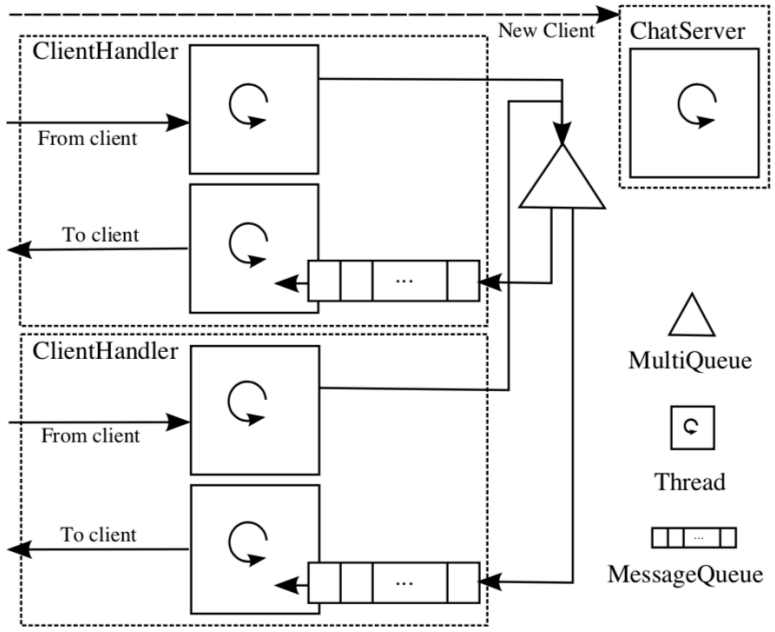
Producer-Consumer Model: producers and consumers share a FIFO queue – need to be careful about multiple producers putting items on the queue at the same time / multiple consumers consuming at the same time – can be solved using mutual exclusion.

Issue with take: programmer pauses execution until at least one item of data is available – this is the wait-sleep paradigm. Better, is the wait-notify paradigm. (this.wait and this.notify or this.notifyAll)

synchronized(object) {
// Java statements
}
This is the syntax for providing mutual exclusion – works with creation and tracking of a lock or mutex. Synchronized can also be put on a method which works like adding a synchronized(this) around the entire method.

Fine-Grained Locking: lock fewer things – for example in this case, lock only on the link so you can lock when only required to do so.

Chat Server



Server contains a single instance of a MultiQueue object, regardless of the number of clients connected to the server. It supports register and deregister methods and these are invoked by multiple instances of ClientHandler.

Vector Clocks

Data structure and associated algorithm for determining the partial order of events in a distributed system. Allows clients to determine whether, given two events A and B, if A occurred before B, after B or whether they were concurrent.

Requires each client to maintain a vector of counters, called a vector clock. The client's current vector clock is attached to events when they happen.

Since we don't know how many communicating processes, represent vector clock by java.util.Map<String, Integer> where each key in the map represents a client which is

represented by java.util.UUID class. The value in the Map should be an integer fulfilling the criteria for the vector clock algorithm.

Updating Vector Clock: A client will increment its own clock element ton any event, including when a local event occurs, or when it sends a message from another client. Therefore, task of reconstructing the order of messages is rather trivial. Rules for updating vector clocks:

1. Each chat client should maintain its own local vector clock for as long as the client is alive
2. When a chat is identifier ID, and a vector clock called client sends a message, the client must increment its own clock element by one before attaching a copy of its own vector clock to the message.
3. When a client with identifier ID receives a message, it updates the clock elements in its own vector clock to be the larger of those in its clock and the values found in the message