**Department of Computer Science and Engineering**

|  |  |
| --- | --- |
| **Course Code: CSE 321** | **Credits: 1.5** |
| **Course Name: Operating Systems** | **Semester: Fall 18** |

**Lab 05  
Thread Synchronization**

1. **Overview:**

In this lab students will learn about thread synchronization, it’s importance and difference between unsynchronized data sharing and synchronized data sharing, producer/consumer relationship with synchronization.

1. **Lesson Fit:**

Basic thread knowledge and programming knowledge is required for this lab.

1. **Learning Outcome:**

After this lab, students will be able to learn how to work with threads, what are the functionalities we can perform with synchronized thread.

1. **Anticipated Challenges and Possible Solutions**
   1. Students may misunderstand the concurrent work of thread and the output may vary among the students which they can seem difficult to understand.

**Solutions:** Actually JVM decides which thread will run and there is no control over it. Without setting any priority you can’t be sure that a thread will run at the beginning. As a result the output may vary.

1. **Acceptance and Evaluation**

Students will show their progress as they complete each task. They will be marked according to their lab performance.

**Activity Detail**

* 1. **Hour: 1  
     Discussion:**

1. Discussion on thread synchronization and how it works.
2. In which situation we should use synchronization.
3. Producer/ consumer relationship with thread synchronization.
   1. **Hour: 2-3**
4. Hands on work with thread synchronization.
5. Evaluation based on their lab works.

**Thread Synchronization:**

When multiple threads share an object and it’s modified by one or more of them, indeterminate results may occur (as we’ll see in the examples) unless access to the shared object is managed properly. If one thread is in the process of updating a shared object and another thread also tries to update it, it’s unclear which thread’s update takes effect. When this happens, the program’s behavior cannot be trusted—sometimes the program will produce the correct results, and sometimes it won’t. In either case, there’ll be no indication that the shared object was manipulated incorrectly.

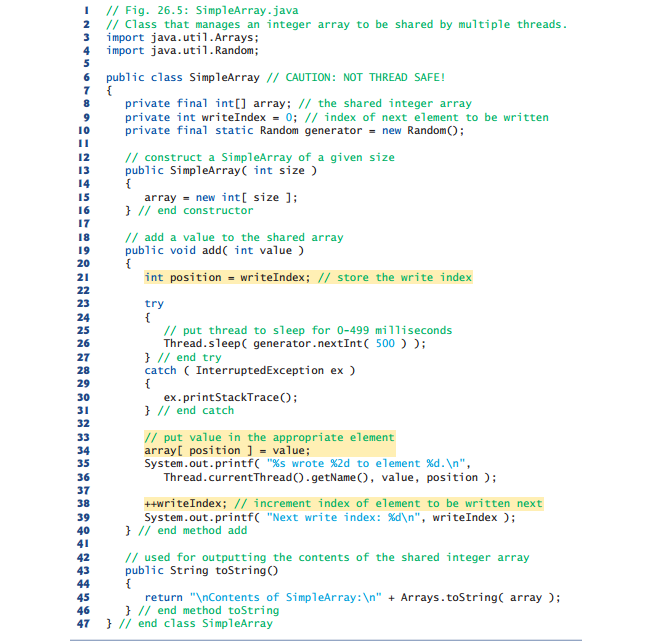
The problem can be solved by giving only one thread at a time *exclusive access* to code that manipulates the shared object. During that time, other threads desiring to manipulate the object are kept waiting. When the thread with exclusive access to the object finishes manipulating it, one of the threads that was waiting is allowed to proceed. This process, alled thread synchronization, coordinates access to shared data by multiple concurrent threads. By synchronizing threads in this manner, you can ensure that each thread accessing a shared object excludes all other threads from doing so simultaneously—this is called mutual exclusion.

**Unsynchronized data sharing:**

First, we illustrate the dangers of sharing an object across threads without proper synchronization. In this example, two Runnables maintain references to a single integer array. Each Runnable writes three values to the array, then terminates. This may seem harmless, but we’ll see that it can result in errors if the array is manipulated without synchronization.

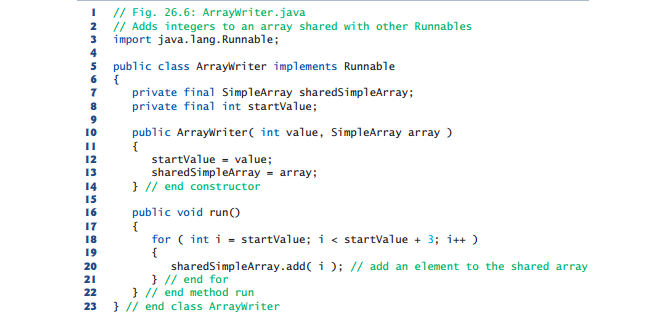
***Class SimpleArray:***

A SimpleArray object (Fig. 26.5) will be *shared* across multiple threads. SimpleArray will allows new values to be inserted at the end of the array. Line 21 stores the current writeIndex value. Line 26 puts the thread that invokes add to sleep for a random interval from 0 to 499 milliseconds. This is done to make the problems associated with *unsynchronized access to shared data* more obvious. After the thread is done sleeping, line 34 inserts the value passed to add into the array at the element specified by position. Lines 35–36 output a message indicating the executing thread’s name, the value that was inserted in the array and where it was inserted. The expression Thread.currentThread.getName() (line 36) first obtains a reference to the currently executing Thread, then uses that Thread’s getName method to obtain its name. Line 38 increments writeIndex so that the next call to add will insert a value in the array’s next element. Lines 43–46 override method toString to create a String representation of the array’s contents.enable those threads to place int values into array (declared at line 8). Line 9 initializes variable writeIndex, which will be used to determine the array element that should be written to next. The constructor (lines 13–16) creates an integer array of the desired size. Method add (lines 19–40)

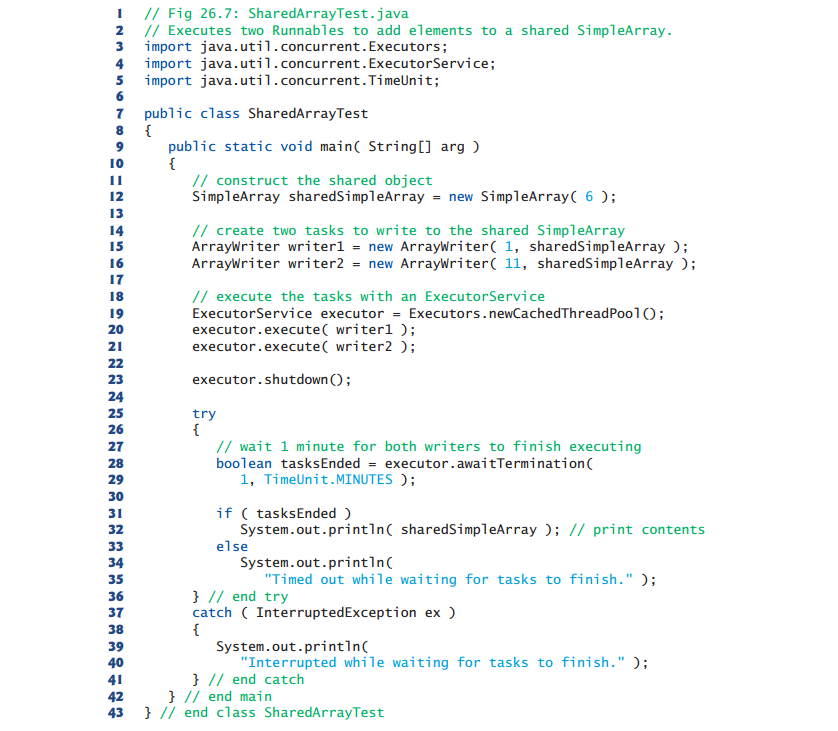
****

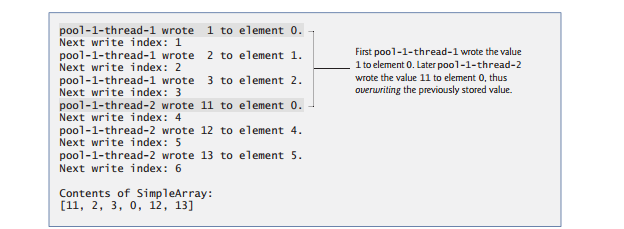
***Class ArrayWriter:***

Class ArrayWriter (Fig. 26.6) implements the interface Runnable to define a task for inserting values in a SimpleArray object. The constructor (lines 10–14) takes two arguments—an integer value, which is the first value this task will insert in the SimpleArray object, and a reference to the SimpleArray object. Line 20 invokes method add on the SimpleArray object. The task completes after three consecutive integers beginning with startValue are added to the SimpleArray object.

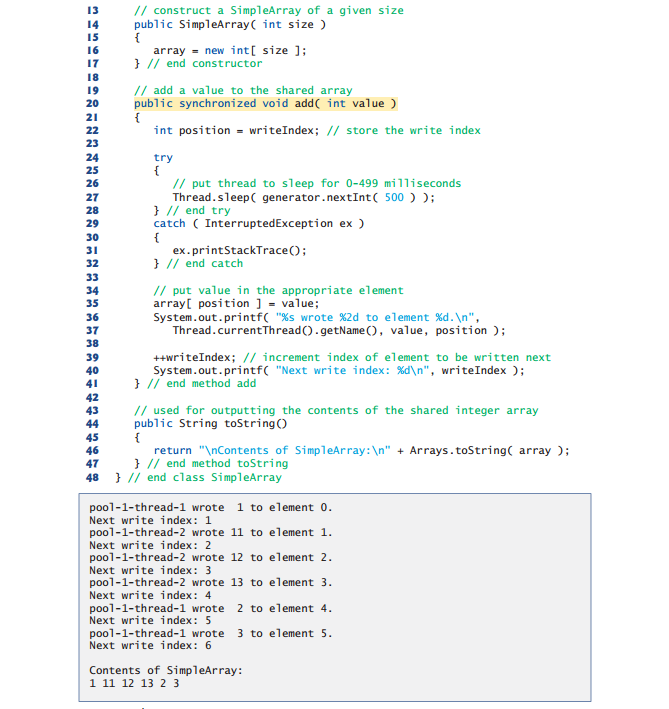


***Class SharedArrayTest:***



**

***Class SimpleArray with Synchronization***

**

**Extra Java program to demonstrate usage of Thread class**

// Java program to demonstrate

// method calls of Thread class

package generic;

class Helper implements Runnable

{

    public void run()

    {

        try

        {

            System.out.println("thread2 going to sleep for 5000");

            Thread.sleep(5000);

        } catch (InterruptedException e)

        {

            System.out.println("Thread2 interrupted");}

        }

}

public class Test implements Runnable

{

    public void run()

    {

        //thread run() method

    }

    public static void main(String[] args)

    {

        Test obj = new Test();

        Helper obj2 = new Helper();

        Thread thread1 = new Thread(obj);

        Thread thread2 = new Thread(obj2);

        // moving thread to runnable states

        thread1.start();

        thread2.start();

        ClassLoader loader = thread1.getContextClassLoader();

        Thread thread3 = new Thread(new Helper());

        // getting number of active threads

        System.out.println(Thread.activeCount());

        thread1.checkAccess();

        // fetching an instance of this thread

        Thread t = Thread.currentThread();

        System.out.println(t.getName());

        System.out.println("Thread1 name: "+thread1.getName());

        System.out.println("Thread1 ID: " + thread1.getId());

        // fetching the priority and state of thread1

        System.out.println("Priority of thread1 = " + thread1.getPriority());

        System.out.println(thread1.getState());

        thread2 = new Thread(obj2);

        thread2.start();

        thread2.interrupt();

        System.out.println("Is thread2 interrupted? " + thread2.interrupted() );

        System.out.println("Is thread2 alive? " + thread2.isAlive());

        thread1 = new Thread(obj);

        thread1.setDaemon(true);

        System.out.println("Is thread1 a daemon thread? " + thread1.isDaemon());

        System.out.println("Is thread1 interrupted? " + thread1.isInterrupted());

        // waiting for thread2 to complete its execution

        System.out.println("thread1 waiting for thread2 to join");

        try

        {

            thread2.join();

        }

        catch (InterruptedException e)

        {

            e.printStackTrace();

        }

        // setting the name of thread1

        thread1.setName("child thread xyz");

        System.out.println("New name set for thread 1" + thread1.getName());

        // setting the priority of thread1

        thread1.setPriority(5);

        thread2.yield();

        // fetching the string representation of thread1

        System.out.println(thread1.toString());

        // getting list of active thread in current thread's group

        Thread[] tarray = new Thread[3];

        Thread.enumerate(tarray);

        System.out.println("List of active threads:");

        System.out.printf("[");

        for(Thread thread : tarray)

        {

            System.out.println(thread);

        }

        System.out.printf("]\n");

        System.out.println(Thread.getAllStackTraces());

        ClassLoader classLoader = thread1.getContextClassLoader();

        System.out.println(classLoader.toString());

        System.out.println(thread1.getDefaultUncaughtExceptionHandler());

        thread2.setUncaughtExceptionHandler(thread1.getDefaultUncaughtExceptionHandler());

        thread1.setContextClassLoader(thread2.getContextClassLoader());

        thread1.setDefaultUncaughtExceptionHandler(thread2.getUncaughtExceptionHandler());

        thread1 = new Thread(obj);

        StackTraceElement[] trace = thread1.getStackTrace();

        System.out.println("Printing stack trace elements for thread1:");

        for(StackTraceElement e : trace)

        {

            System.out.println(e);

        }

        ThreadGroup grp = thread1.getThreadGroup();

        System.out.println("ThreadGroup to which thread1 belongs " +grp.toString());

        System.out.println(thread1.getUncaughtExceptionHandler());

        System.out.println("Does thread1 holds Lock? " + thread1.holdsLock(obj2));

        Thread.dumpStack();

    }

}

**Output**

3

main

Thread1 name: Thread-0

Thread1 ID: 10

Priority of thread1 = 5

RUNNABLE

Is thread2 interrupted? false

Is thread2 alive? true

Is thread1 a daemon thread? true

Is thread1 interrupted? false

thread1 waiting for thread2 to join

thread2 going to sleep for 5000 ms

thread2 going to sleep for 5000 ms

Thread2 interrupted

New name set for thread 1child thread xyz

Thread[child thread xyz, 5, main]

List of active threads:

[Thread[main, 5, main]

Thread[Thread-1, 5, main]

null

]

{Thread[Signal Dispatcher, 9, system]=[Ljava.lang.StackTraceElement;@33909752,

Thread[Thread-1, 5, main]=[Ljava.lang.StackTraceElement;@55f96302,

Thread[main, 5, main]=[Ljava.lang.StackTraceElement;@3d4eac69,

Thread[Attach Listener, 5, system]=[Ljava.lang.StackTraceElement;@42a57993,

Thread[Finalizer, 8, system]=[Ljava.lang.StackTraceElement;@75b84c92,

Thread[Reference Handler, 10, system]=[Ljava.lang.StackTraceElement;@6bc7c054}

sun.misc.Launcher$AppClassLoader@73d16e93

null

Printing stack trace elements for thread1:

ThreadGroup to which thread1 belongs java.lang.ThreadGroup[name=main, maxpri=10]

java.lang.ThreadGroup[name=main, maxpri=10]

Does thread1 holds Lock? false

java.lang.Exception: Stack trace

at java.lang.Thread.dumpStack(Unknown Source)

at generic.Test.main(Test.java:111)