CS240 Operating Systems, Communications and Concurrency - Dermot Kelly
Practical 8 Concurrent Programming in Java

The Dining Philosophers problem is an example of a concurrency problem dealing with the allocation of limited resources among competing processes. The code of a solution to the dining philosophers problem was covered in class and is given overleaf. The source code is contained in the Practical 8 code directory on moodle. You will also need the Semaphore class from last week's lab.

Create a new Java project in Eclipse "CS240 Practical 8a" and drag the two source code files "Diningphilosophers.java" and "Semaphore.java" from moodle into the project's src directory.

Study the program and try to follow what you expect it to do before running.

Right click on the file DiningPhilosophers.java and select Run As -> Java Application

Observe the output. Stop the program clicking the red square icon above the console output window. Run it again. It is most likely you will get a different output sequence due to the different scheduling of the threads in the second run.

Note that from the way the program is written, the simulation could deadlock if the threads were scheduled in such a way that each philosopher picked up one chopstick. If the code deadlocks then it will stop producing output as all the threads will be blocked.

Modify the program in **two** ways (as suggested in class) to create a simulation that won't deadlock.

For each of these modifications, you should create a new Java project in Eclipse, "CS240 Practical 8b" and "CS240 Practical 8c". Put copies of the original two code files into the src folder of both of these new projects and then modify the DiningPhilosophers.java file in each to resdesign a deadlock free simulation.

```java
// This is the file DiningPhilosophers.java
// Study the class Semaphore and the class Philosopher given below.
// A Semaphore object maintains a private integer which can only be accessed by the
// operations P and V. These are declared as synchronized which means the procedures
// (methods) execute indivisibly on the semaphore's value when they are invoked by
// different threads. The Philosopher class extends the Thread class and defines a new
// run() method for it which is executed when the thread is started.
// Methods of the same name as the class in which they are defined are known as
// constructors. These are used to initialise instances of objects when they are first
// created.
// The class DiningPhilosophers contains the main program and this is where execution
// begins.

class DiningPhilosophers {
	public static void main(String args[]) {
		Semaphore chopSticks[];
		Philosopher workerThread[];

		// Create an array of five Semaphore Object Reference Handles
		chopSticks = new Semaphore[5];

		// Create five Binary Semaphore Objects and assign to the array
		for (int i=0; i<5; i++)
			chopSticks[i] = new Semaphore(1); // Semaphore initial value=1

		// Create an array of five Philosopher Thread Object Reference Handles
		workerThread = new Philosopher[5];

		// Create and initiate five Philosopher Thread Objects
		for (int i=0; i<5; i++) {
			workerThread[i] = new Philosopher(i, chopSticks);
			workerThread[i].start();
		}
	}
}
```

```
// The Philosopher class implements a run() method defining the behaviour of a Philosopher thread

class Philosopher extends Thread {
	private int myName;
	private Semaphore chopSticks[];
	//
	// This is the constructor function which is executed when a Philosopher
	// thread is first created
	//
	public Philosopher(int myName, Semaphore chopSticks[]) {
		this.myName = myName; // 'this' distinguishes the local private variable from the
parameter
		this.chopSticks = chopSticks;
	}

	//
	// This is what each philosopher thread executes
	//
	public void run() {
		while (true) {
			System.out.println("Philosopher "+myName+" thinking.");
			try {
				sleep ((int)(Math.random()*20000));
			} catch (InterruptedException e) {}

			System.out.println("Philosopher "+myName+" hungry.");
			chopSticks[myName].acquire(); // Acquire right
			chopSticks[(myName+1)%5].acquire(); // Acquire left

			System.out.println("Philosopher "+myName+" eating.");
			try { // Simulate eating activity for a random time
				sleep ((int)(Math.random()*10000));
			} catch (InterruptedException e) {}

			chopSticks[myName].release(); // Release right
			chopSticks[(myName+1)%5].release(); // Release left
		}
	}
}
```

```java
/* The Semaphore class contains methods declared as
synchronized. Java's locking mechanism will ensure
that access to Semaphore methods is mutually exclusive
among threads that invoke these methods.
*/
public class Semaphore {
	private int value;

	public Semaphore(int value) {
		this.value = value;
	}

	public synchronized void acquire() {
		while (value == 0) {
			try {
				// Calling thread waits until semaphore is free
				wait();
			} catch(InterruptedException e) {}
		}
		value = value - 1;
	}

	public synchronized void release() {
		value = value + 1;
		notify();
	}
}
```