COURSE:	CSC 310 (CONCURRENT PROGRAMMING)
NAME:	OLORUNFEMI-OJO DANIEL TOMIWA
MATRIC NO:	190805503

SYNCHRONIZERS

A synchronizer is any object that coordinates the control flow of threads based on its state. Thread communication happens primarily by sharing access to fields and objects. This form of communication is prone to errors caused by race condition i.e. two or more threads can access shared data and they try to change it at the same time. Synchronizers are tools that helps to prevent such errors. There are many types of synchronizers like latches, semaphores, barriers, etc, but they all share one structural property. They encapsulate state that determines whether threads arriving at the Not-Thread Safe section should be allowed to pass or forced to wait and provide methods to manipulate that state.

LATCHES

A Latch is a synchronizer that allows one or more threads to wait on the completion of operations in other threads. The thread that will wait creates a Latch and sets an initial value. The initiating thread passes the latch to the other threads then waits for the other threads to decrement the latch when done with their work. When the latch counter reaches zero the waiting thread is unblocked and continues with its work.

The manager of Company A promises to reward the most hardworking staff member. At the end of the month, three employees are eligible for the reward. The manager doesn't know who to pick. He decides to setup a voting system for staff members to choose who should be rewarded. There are fifty staff members so he is expecting fifty votes. The results will be calculated and sent to him only when every staff member has voted.

IMPLEMENTATION

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class VotingSystem {

	public static int candidateA = 0, candidateB = 0, candidateC = 0;

	public static void main(String[] args) {
		CountDownLatch voters = new CountDownLatch(50);
		ExecutorService service = Executors.newCachedThreadPool();
		for(int i = 0; i < 50; i++) {
			service.execute(new Runnable() {
				@Override
				public void run() {
					vote(voters);
```

```
                        }
                });
        }

        service.shutdown();
        try {
                voters.await();
                calculateResults();
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
    }

    public static synchronized void vote(CountDownLatch voters) {
        int choice = (int) (Math.random() * 3);
        switch (choice) {
                case 0:
                        candidateA++;
                        break;
                case 1:
                        candidateB++;
                        break;
                case 2:
                        candidateC++;
                        break;
                default:
                        break;
        }
        voters.countDown();
    }

    public static void calculateResults() {
        ArrayList<Integer> results = new ArrayList<Integer>();
        Collections.addAll(results, candidateA, candidateB, candidateC);
        System.out.println("A: " + results.get(0) + ", B: " + results.get(1) + ", C: " +
results.get(2));
        int highestVotes = Collections.max(results);
        switch (results.indexOf(highestVotes)) {
                case 0:
                        System.out.println("Candidate A won");
                        break;
                case 1:
                        System.out.println("Candidate B won");
                        break;
                case 2:
                        System.out.println("Candidate C won");
                        break;
                default:
                        break;
        }
    }
}
```

BARRIERS

A Barrier is a synchronizer that enables multiple threads to work concurrently in phases. Each thread executes until it reaches the barrier point in the code. The barrier represents the end of one phase of work. When a thread reaches the barrier, it is blocked until other threads reach the same barrier. When

all threads reach the barrier, a post-phase action is invoked. A barrier size is set and when a thread reaches the barrier, the barrier size decreases. When the expected count reaches zero, the phase completion step is executed.

Game D has a multiplayer mode. The multiplayer mode requires four players to be connected before the game begins. If the number of players connected is not four, the connected players remain in a waiting room. The four player barrier is decremented when a player connects. When the barrier is zero, the game begins.

IMPLEMENTATION

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class Game {

	public static void main(String[] args) {
		CyclicBarrier barrier = new CyclicBarrier(4);
		for(char c = 'A'; c < 'E'; c++) {
			new Thread(new Multiplayer(barrier, c)).start();
		}
	}

	public static class Multiplayer implements Runnable {

		private CyclicBarrier barrier;
		private char id;

		public Multiplayer(CyclicBarrier barrier, char id) {
			this.barrier = barrier;
			this.id = id;
		}

		@Override
		public void run() {
			waitingRoom();
		}

		public void waitingRoom() {
			try {
				System.out.println("Player " + id + " arrived");
				if(barrier.await() == 0) {
					executeGame();
				}
			} catch (InterruptedException | BrokenBarrierException e) {
				e.printStackTrace();
			}
		}

		public void executeGame() {
			System.out.println("Game begins");
		}
	}

}
```

SEMAPHORES

A semaphore limits the number of threads that can access a resource or pool of resources concurrently. The number of threads allowed to access the resource is specified on creation of the semaphore. The count on a semaphore is decremented each time a thread enters the semaphore, and incremented when a thread releases the semaphore. When the count is zero, subsequent requests block until other threads release the semaphore. When all threads have released the semaphore, the count is at the maximum value specified when the semaphore was created. Semaphores can be implemented as either a binary semaphore or counting semaphore. A binary semaphore guarantees mutually exclusive access to a resource (only one thread is in the critical section at a time). The maximum number of permits available is one. A counting semaphore is used when multiple units of a resource is available. The number of available permits will be the same as the number of the resource available.

Company A has a web application for its staff to perform their jobs. For security purposes, the system administrator configures the application to log the IP address of all connections made to the server. For every new connection made, the server creates a thread to maintain that session. Multiple connections are made to the server at the same time. They all try to write to the log file and it loses integrity because some threads will not be writing to the latest copy of the logs. Some logs will be lost. A semaphore with one permit is added so that only one thread can write to the log file. Instead of writing to the log file haphazardly, each thread has to wait to get the permit to write to the log file. Access to the log file is limited to one.

IMPLEMENTATION

```
import java.util.ArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

public class Server {
	public static String ip;

	public static void main(String[] args) {
		ExecutorService connections = Executors.newFixedThreadPool(5);
		for (int i = 0; i < 5; i++) {
			ip = getIP();
			connections.execute(Logger.getInstance());
		}

		connections.shutdown();
	}

	public static String getIP() {
		int digits[]= new int[4];
		for(int i = 0; i < 4; i++) {
			digits[i] = (int) (Math.random() * 255);
		}
		return Integer.toString(digits[0]) + "." + Integer.toString(digits[1]) + "." +
				Integer.toString(digits[2]) + "." + Integer.toString(digits[3]);
	}
}

class Logger implements Runnable{
```

```
        Semaphore sempahore = new Semaphore(1);
        ArrayList<String> logs = new ArrayList<String>();

        private static final Logger manager = new Logger();

        @Override
        public void run() {
                try {
                        sempahore.acquire();
                        writeLog(Server.ip);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                } finally {
                        sempahore.release();
                }
        }

        public void writeLog(String ip) {
                logs.add(ip);
                System.out.println(logs.toString());
        }

        public static Logger getInstance() {
                return manager;
        }
}
```

BLOCKING QUEUES

Blocking Queue is a thread-safe collection class that provides concurrent addition and removal of items from multiple threads. A blocking queue blocks when you try to dequeue an empty queue or if you try to enqueue items in a full queue. A thread trying to dequeue an empty queue is blocked until some other threads insert an item into the queue. Blocking queues support bounding. Bounding means that you can set the maximum capacity of the collection. An unbounded blocking queue is a type that has no limit to the amount of values that can be stored (the only limiting factor is memory size). Blocking queues support the producer-consumer design pattern. A producer-consumer design separates the identification of work to be done from the execution of that work by placing work items on a "to do" list for later processing, rather than processing them immediately as they are identified. The producer-consumer pattern simplifies development because it removes code dependencies between producer and consumer classes, and simplifies workload management by decoupling activities that may produce or consume data at different or variable rates. In a producer-consumer design built around a blocking queue, producers place data onto the queue as it becomes available, and consumers retrieve data from the queue when they are ready to take the appropriate action. Producers need not to know anything about the identity or number of consumers, or even whether they are the only producer all they have to do is place data items on the queue. Similarly, consumers need not know who the producers are or where the work came from. Bounding is important in certain scenarios because it enables you to control the maximum size of the collection in memory, and it prevents the producing threads from moving too far ahead of the consuming threads. Multiple threads or tasks can add items to the collection concurrently, and if the collection reaches its specified maximum capacity, the producing threads will block until an item is removed. Multiple consumers can remove items concurrently, and if the collection becomes empty, the consuming threads will block until a producer adds an item.

Company A has a customer support application for clients to chat with their staff. Company A assigns five people to the helpdesk platform to respond to clients. When a new client initiates a session, the client is placed at the bottom of the blocking queue and a free staff member claims the first client available in the queue. While the blocking queue is not empty, the helpdesk members will try to claim a client from the queue. The Client class is the Producer class and the HelpDesk class is the consumer class because the HelpDesk claims the products of the Producer class which are clients to chat with.

IMPLEMENTATION

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class CustomerCare {

	public static void main(String[] args) {
		BlockingQueue<Character> clientQueue = new ArrayBlockingQueue<Character>(5);
		new Thread(new Client(clientQueue)).start();
		for(int i = 0; i < 5; i++) {
			new Thread(new HelpDesk(clientQueue, i)).start();
		}
	}

	// Consumer Class
	public static class HelpDesk implements Runnable {

		private BlockingQueue<Character> clientQueue;
		private int helpdeskID;

		public HelpDesk(BlockingQueue<Character> clientQueue, int helpdeskID) {
			this.clientQueue = clientQueue;
			this.helpdeskID = helpdeskID;
		}

		@Override
		public void run() {
			while(!clientQueue.isEmpty()) {
				try {
					System.out.println("Agent " + Integer.toString(helpdeskID)
+ " is chatting with Client " + clientQueue.take());
					// Represents chat session
					Thread.sleep((int) (Math.random() * 5000));
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
		}
	}

	// Producer Class
	public static class Client implements Runnable {

		private BlockingQueue<Character> clientQueue;

		public Client(BlockingQueue<Character> clientQueue) {
			this.clientQueue = clientQueue;
		}

		@Override
		public void run() {
```

```
                        for(char c = 'A'; c <= 'J'; c++) {
                                try {
                                        clientQueue.put(c);
                                        System.out.println("New Client: " + c);
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                        }
                }
        }

}
```