Ian Kenny

February 1, 2023

5CCS2OSC Operating Systems and Concurrency

More concurrency, issues with concurrency

In this lecture

- The `Lock` interface
- `ReentrantLock`
- `synchronized` vs `ReentrantLock`
- Deadlock
- The Dining Philosophers problem
- Eliminating deadlock
- Semaphores

`synchronized`

In the previous week we saw that we can use a `synchronized` method or block to create a critical section to try and avoid threads interfering with each other.

`synchronized` is a simple but effective way of avoiding problems such as race conditions.

However, synchronization, and locking in general, can lead to some issues that we will discuss in this lecture.

The `Lock` interface

The `synchronized` keyword allows for convenient locking but it is sometimes too blunt a tool. `synchronized` can only be applied to entire methods or to blocks within methods.

Java also supports other locks that have more flexibility. These locks implement the `Lock` interface.

These explicit locks have to be created, locked and unlocked by the programmer. That is the price to pay for enhanced flexibility.

The program on the next slide revisits one of the earlier examples but uses an implementation of the `Lock` interface called `ReentrantLock`.

The Lock interface

```
public class LockDemo implements Runnable {
   private int counter = 0;
   private Lock lock = new ReentrantLock();
   public void run() {
      for (int i = 0; i < 100_000; i++) {
         addToCounter(1);
      }
      System.out.println("Final Counter: "+
          Thread.currentThread().getName() + " " +
          counter);
   }
   private void addToCounter(int amount) {
      if (amount >= 0) {
         try {
            lock.lock();
            counter += amount;
         }
         finally {
            lock.unlock();
         }
      }
   }
   public static void main(String[] args) {
      LockDemo runnable1 = new LockDemo();
      Thread thread1 = new Thread(runnable1);
      Thread thread2 = new Thread(runnable1);
      thread1.start();
      thread2.start();
   }
}
```

Listing 1: LockDemo.java

LockDemo

You will see that in this program, the class has an instance variable of type `ReentrantLock`. We will discuss the concept of *reentrancy* shortly.

```
private Lock lock = new ReentrantLock();
```

Listing 2: LockDemo.java

This decouples the creation of the lock from particular methods and blocks. The lock is now just an object that can be used explicitly anywhere in the class.

LockDemo

In the `addToCounter()` method we now explicitly `lock` the lock to open the critical section, and then `unlock` it afterwards.

The lock acquisition and release are encapsulated in a `try-finally` block. This is not compulsory but recommended because the code in the critical section might throw an exception and the `finally` block will ensure that we unlock the lock regardless.

```
private void addToCounter(int amount) {
   if (amount >= 0) {
      try {
         lock.lock();
         counter += amount;
      }
      finally {
         lock.unlock();
      }
   }
}
```

Listing 3: LockDemo.java

Reentrancy

Java `synchronized` blocks (and methods) are *reentrant*. This means that once a thread has obtained an object's lock it can enter other synchronized blocks (and methods) on the same object.

This makes sense if you consider that `synchronized` methods, for example, might need to call other `synchronized` methods. Without the facility for reentrancy, this would not be possible.

Consider the case in which a thread calls synchronized method `f()` on object `A` and that method `f()` calls synchronized method `g()` also on object A. Since the thread has been granted the lock upon entering `f()`, without reentrancy, it would be blocked from entering `g()` until it is granted the lock, which it already holds!

`ReentrantLock` is a basic lock that has the property of reentrancy.

Reentrancy

```
public class Reentrancy {
   public synchronized void f() {
      g();
   }
   public synchronized void g() {
   }
}
```

Listing 4: Reentrancy.java

Java `synchronized` blocks are reentrant. The code above would be impossible if that was not the case. Java `ReentrantLock` is also reentrant, as its name suggests ...

`synchronized` vs `ReentrantLock`

On the face of it, the behaviour of a `synchronized` block or method is similar to that of `ReentrantLock`, so how would you choose between them?

As already stated, a `ReentrantLock` is not restricted to a single method as with `synchronized`.

A `ReentrantLock` can be locked and unlocked across *different* methods.

`synchronized` vs `ReentrantLock`

Another aspect of a `synchronized` block is that the JVM makes no attempt to ensure fairness between the threads competing to enter the block.

A `ReentrantLock` can be made to give access to threads in the order in which they arrive at the critical section.

A `ReentrantLock` also has a wider range of possible acquisition policies. For example, it can choose to only acquire the lock if it is immediately available, or available within a certain time period. It can also allow itself to be interrupted whilst waiting to get the lock.

A `ReentrantLock` object can also give information about its 'waiting list', etc.

Locks

Both intrinsic locks (acquired via `synchronized`) and explicit locks such as `ReentrantLock` are convenient ways to create critical sections to protect shared data.

If we only use locks in *one* object in our programs then we are unlikely to get many problems.

However, in a realistic application, there will usually be multiple threads trying to acquire multiple locks and, if there are *dependencies* in the locking arrangements, then we could get into problems.

Deadlock

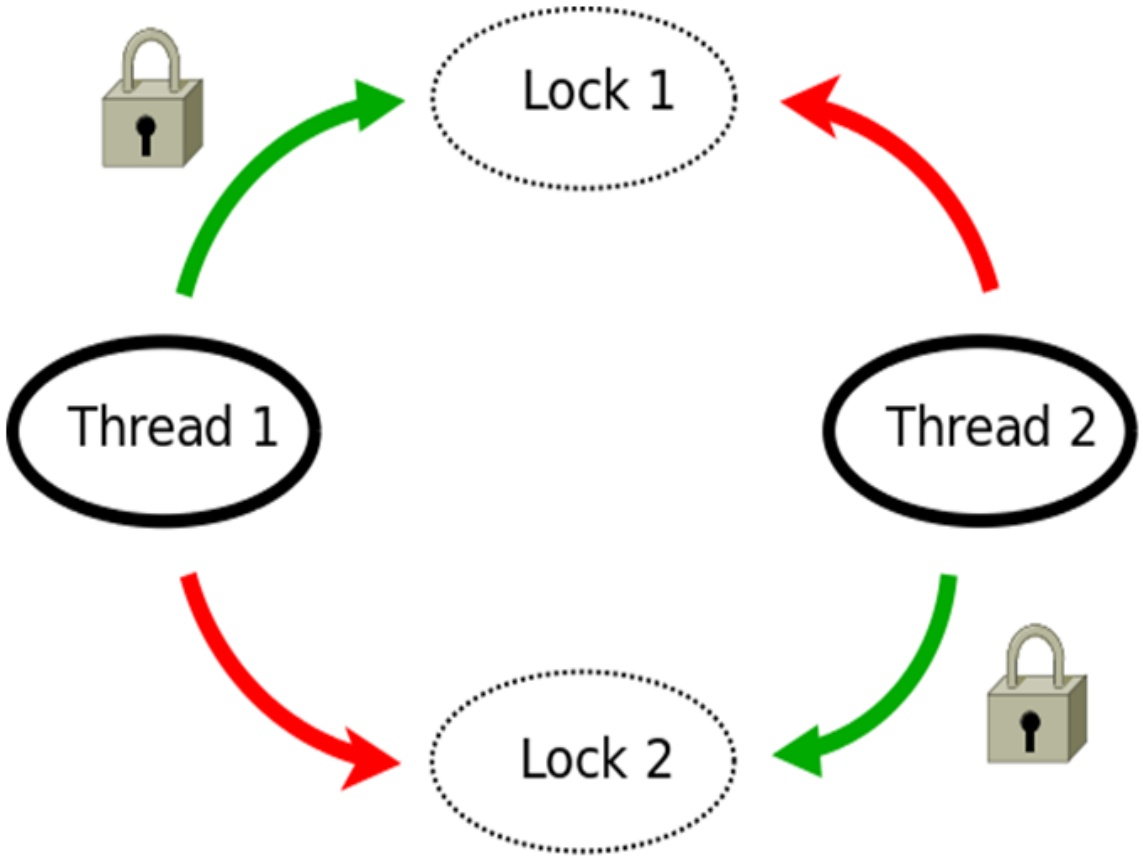
Consider the following scenario: thread A is holding lock X and thread B is holding lock Y. Thread A needs to acquire lock Y to proceed and thread B needs to acquire lock X to proceed.

Both threads will be stuck. They can't continue until they acquire a lock that the other thread is holding.

This is illustrated on the next slide. In the figure, a green arrow means 'holds this lock' whilst a red arrow means 'needs this lock'.

We can see that, in this image, both threads are stuck because they need a lock that the other one holds.

Deadlock



https://www.modernescpp.com/index.php/the-risk-of-mutexes

Conditions for deadlock

Deadlock can occur if the following conditions are all true.

1. Mutual Exclusion: i.e. the existence of critical sections.
2. Hold and Wait: a thread (or process) is currently holding a resource (e.g. a lock) and is seeking to acquire others.
3. No preemption: a resource (e.g. a lock) is only released when the holding thread releases it.
4. Circular wait: as in the figure on the previous slide, there exists a cycle of threads or processes waiting on each other for resources.

Deadlock

This might seem like an unrealistic scenario but it is surprisingly easy to create.

Consider a simple example relating to a bank account on the following slides.

The first class we will look at is a simple `BankAccount` class.

Bank account example

```
public class BankAccount {
   private double balance;
   private int id;
   private String name;
   public BankAccount(int id, String name, double
       balance){
      this.balance = balance;
      this.id = id;
      this.name = name;
   }
   void withdraw(double amount){
      balance -= amount;
      System.out.println("Account " + name + " withdraw.
          New balance " + balance);
   }
   void deposit(double amount){
      balance += amount;
      System.out.println("Account " + name + " deposit.
          New balance " + balance);
   }
   public String getName() {
      return name;
   }
   public int getId() {
      return id;
   }
}
```

Listing 5: BankAccount.java

Bank account example

The next slide shows a program that uses the `BankAccount` class. The program creates two bank accounts and then starts two threads that transfer money between the accounts.

The `transfer` method is the most important one and is shown on the following slide.

(Note: this method shows yet another way of declaring and instantiating `Thread` objects.)

Bank account example

```
public static void main(String [] args) throws
    InterruptedException{
   BankAccount sid = new BankAccount(1,"Sid",10000);
   BankAccount doris = new BankAccount(2,"Doris",3000);
   Thread thread1 = new Thread("Thread1"){
      public void run(){
         transfer(sid,doris,200);
      }
   };
   Thread thread2 = new Thread("Thread2"){
      public void run(){
         transfer(doris,sid,300);
      }
   };
   thread1.start();
   thread2.start();
}
```

Listing 6: BankDeadlock.java

The `transfer()` method

```
public static void transfer(BankAccount from,
    BankAccount to, double amount){
   synchronized(from){
      synchronized(to){
         from.withdraw(amount);
         to.deposit(amount);
      }
   }
```

Listing 7: BankDeadlockNoPrint.java

(The version available on Keats has a lot of print statements in it so you can see what is happening. Those are omitted here because they makes the method more confusing to read.)

Bank account example: output

```
Thread1 need to acquire Sid lock.
Thread2 need to acquire Doris lock.
Thread1 has acquired Sid lock.
Thread2 has acquired Doris lock.
Thread1 need to acquire Doris lock.
Thread2 need to acquire Sid lock.
```

The threads are *deadlocked* trying to acquire a lock the other thread holds.

Bank account example: output

Why does this deadlock occur? You may have already seen why.

The problem is that the two threads call the `transfer()` method with the accounts in a different order. This means that when both threads successfully acquire the 'from' lock, the 'from' lock is for two different objects. Thus, the locks for "Sid" and "Doris" are now already held by the two threads (one lock each).

The next thing that the two threads attempt to do is to acquire the 'to' lock. But the lock on the 'to' objects is already owned by the threads (because both object locks are already owned). Neither can acquire the needed lock so both threads are stuck, hence the deadlock.

Deadlock avoidance

How do we prevent this situation occurring?

The best approach is to design our code so that deadlock can't occur. We will see an easy way to fix the program we have just looked at shortly. There are often simple ways of avoiding deadlock like that.

We usually can't get rid of mutual exclusion completely (i.e. have no critical sections) so we have to design our programs carefully. There are also approaches that use deadlock detection that identify when deadlock has occurred and then break it somehow.

Bank account example

We can fix the bank account program quite easily. This approach also works in many other contexts.

In scenarios where you have multiple objects acquiring multiple locks, you can create a fixed order in which the objects attempt to acquire the locks based on some property of the objects. That way, locks will always be acquired in the same order, preventing the deadlock.

In the fixed version of `transfer()` on the next slide, a bit of code has been added to make sure that the order of acquisition of the locks is always the same.

Bank account example

```
public static void transfer(BankAccount from,
    BankAccount to, double amount){
   BankAccount first = from;
   BankAccount second = to;
   if (first.getId() < second.getId()) {
      first = to;
      second = from;
   }
   synchronized(first){
      synchronized(second){
         from.withdraw(amount);
         to.deposit(amount);
      }
   }
}
```

Listing 8: BankDeadlockFixedNoPrint.java

(The version available on Keats has a lot of print statements in it so you can see what is happening. Those are omitted here because they makes the method more confusing to read.)

The dining philosophers problem

The previous example used only two threads to illustrate deadlock. However, the problem is more general than that and can occur whenever shared resources can be locked and multiple threads (or processes) are trying to access them.

Another, famous, example is the *dining philosophers problem*. This problem is usually stated as described next.

The dining philosophers problem

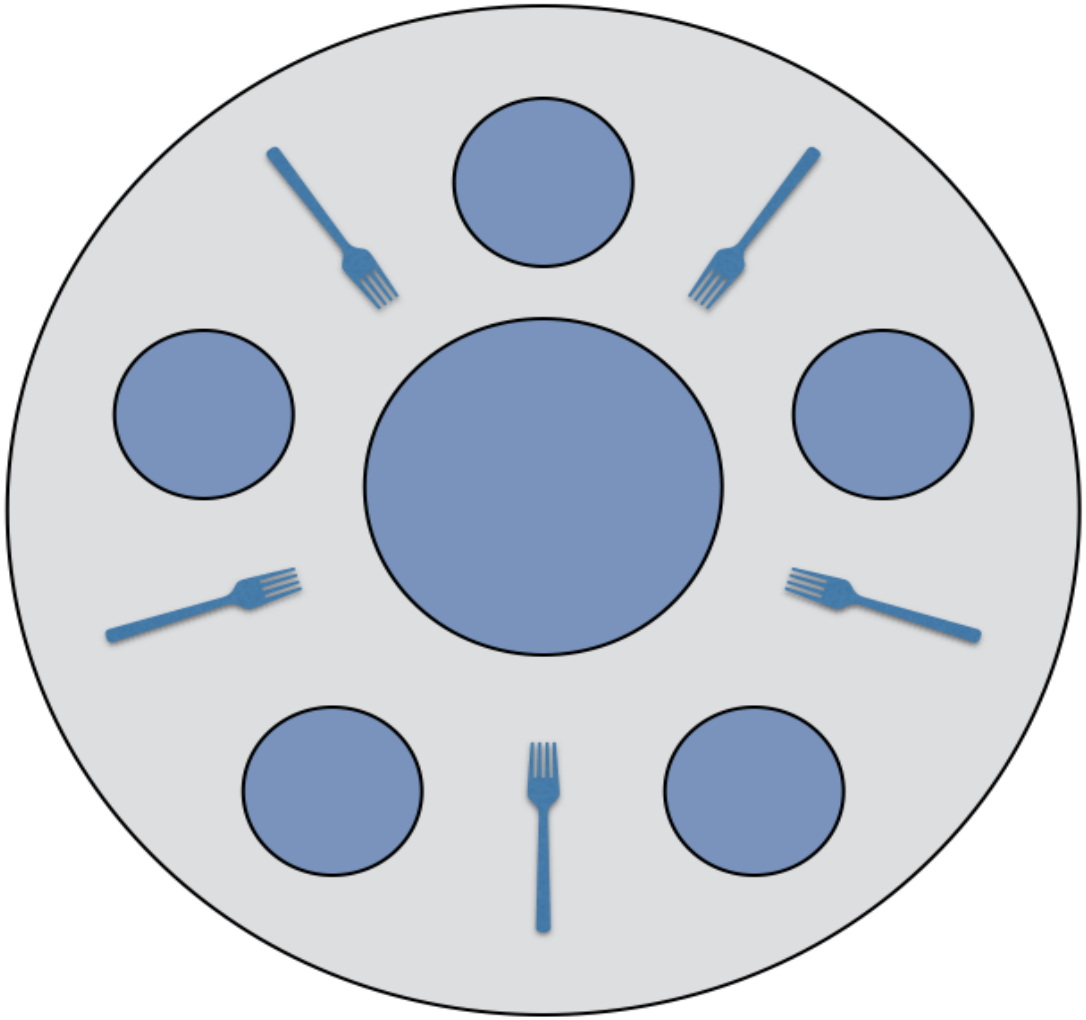
There are five philosophers sitting at a table. The philosophers alternate between thinking and eating. Each thinks for some random period of time until they feel hungry. They then attempt to eat until they are full.

Each philosopher requires two forks to eat, but there are only five forks on the table, in the positions as shown on the next slide[1]. As soon as an adjacent fork becomes available, a philosopher will pick it up. They can't eat until they have two forks though: one in each hand.

Philosophers are selfish and greedy and do not coordinate their actions. An eating philosopher will eat until they are ready to stop and think again. They will also immediately grab an available fork, even if they can't use it yet because it's the only one they have.

[1] This arrangement is, of course, unhygienic.

The dining philosophers problem



https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/DiningPhil.html

The dining philosophers problem

The philosophers are not shown in the figure but their plates are (the five blue circles). The large blue circle is where they take the food from but that doesn't really play a part in the problem.

You can probably see the problem. A philosopher will only be able to eat if they can pick up the two forks on either side of their plate. They are competing for the forks. They will not be able to eat if the adjacent philosopher is already eating.

The dining philosophers problem

So, what each philosopher attempts to do is:

1. Think for a while.
2. Pick up the left fork when available.
3. Pick up the right fork when available.
4. Eat until full.
5. Put down the left fork.
6. Put down the right fork.
7. goto 1.

The dining philosophers problem

Since there is no coordination between the philosophers, without intervention this setup could lead to some philosophers eating a lot, some eating a little, some eating nothing or all of them eating nothing.

It's quite possible that, after a period thinking, each philosopher reaches for the left fork at roughly the same time. If this happens then the whole system is deadlocked because no philosopher will be able to pick up their right fork and they will all starve.

The dining philosophers problem

We will now look at a program that replicates the dining philosophers problem.

We first create a class called `Philosopher`. The constructor is shown on the next slide.

The dining philosophers problem

```
public Philosopher(Fork f1, Fork f2, int n) {
   number = n;
   fork1 = f1;
   fork2 = f2;
}
```

Listing 9: Philosopher.java

The `Fork` class is completely empty but it gives us what we need: an object with an intrinsic lock.

Note that each philosopher receives a reference to two forks in the constructor. These are just references to the left-hand and right-hand fork of the philosopher. It does not mean that they have yet picked up those forks. They haven't. `number` is just an ID number for each philosopher.

Philosopher.run()

Here is the `run()` method for the `Philosopher` class since each `Philosopher` will run in a thread.

```
public void run() {
   while (true) {
      act("is thinking");
      synchronized(fork1) {
         act("has picked up left fork.");
         synchronized(fork2) {
            act("has picked up right fork.");
            act("is eating.");
         }
      }
   }
}
```

Listing 10: Philosopher.java

The `act()` method simply prints out what the philosopher is currently doing and puts the current thread to sleep for a random amount of time.

`Philosopher.run()`

Each philosopher thread attempts to 'pick up' the two forks by first synchronizing on the first fork and then on the second (note that you have to obtain the locks sequentially if you want more than one).

So when a philosopher has managed to get one fork, they must wait for the other one to become available before 'eating'.

The next slide shows a program that creates the five philosophers and forks, allocates forks to philosophers and then starts each philosopher thread.

The program

```
public class DiningPhilosophers {
   private static final int SIZE = 5;
   private static Fork[] forks = new Fork[SIZE];
   private static Philosopher[] philosophers = new
       Philosopher[SIZE];
   public static void main(String[] args) {
      for (int i = 0; i < SIZE; i++) {
         forks[i] = new Fork();
      }
      for (int i = 0; i < SIZE; i++) {
         philosophers[i] = new Philosopher(forks[i],
             forks[(i + 1) % forks.length], i+1);
      }
      for (int i = 0; i < SIZE; i++) {
         new Thread(philosophers[i]).start();
      }
   }
}
```

Listing 11: DiningPhilosophers.java

Possible output

```
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 1 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 5 has picked up left fork.
Philosopher 4 has picked up left fork.
Philosopher 1 has picked up left fork.
Philosopher 2 has picked up left fork.
Philosopher 3 has picked up left fork.
```

The philosophers are deadlocked and will starve and never think again.

The dining philosophers problem

It is possible that the dining philosophers program, as shown above, will work sometimes. That is, run for a long time without deadlock. However this can't be guaranteed at all.

We can create an easy solution to this problem by reordering the forks. If all of the philosophers attempt to pick up their left fork, as usual, except the final one who picks up their right fork, we avoid the deadlock.

This works because the previous version contains a *circular wait* condition in which the philosophers are all waiting on the philosopher to the right of them.

The fix

```
public class DiningPhilosophersFixed {
   private static final int SIZE = 5;
   private static Fork[] forks = new Fork[SIZE];
   private static Philosopher[] philosophers = new
       Philosopher[SIZE];
   public static void main(String[] args) {
      for (int i = 0; i < SIZE; i++) {
         forks[i] = new Fork();
      }
      for (int i = 0; i < SIZE-1; i++) {
         philosophers[i] = new Philosopher(forks[i],
             forks[(i + 1) % forks.length], i+1);
      }
      philosophers[SIZE-1] = new Philosopher(forks[(SIZE)
          % forks.length], forks[SIZE-1], SIZE);
      for (int i = 0; i < SIZE; i++) {
         new Thread(philosophers[i]).start();
      }
   }
}
```

Listing 12: DiningPhilosophersFixed.java

Possible output (partial)

```
Philosopher 3 is thinking
Philosopher 5 is thinking
Philosopher 4 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 5 has picked up left fork.
Philosopher 4 has picked up left fork.
Philosopher 2 has picked up left fork.
Philosopher 3 has picked up left fork.
Philosopher 4 has picked up right fork.
Philosopher 4 is eating.
Philosopher 3 has picked up right fork.
Philosopher 4 is thinking
Philosopher 5 has picked up right fork.
Philosopher 3 is eating.
Philosopher 5 is eating.
Philosopher 1 has picked up left fork.
Philosopher 5 is thinking
Philosopher 4 has picked up left fork.
Philosopher 3 is thinking
```

The dining philosophers problem

This fix does work in this case but we can also find solutions that use concurrency constructs.

We saw earlier that we can create `Lock` objects that give us some more flexibility than the `synchronized` block. We can make use of that additional flexibility with the dining philosophers problem.

Using `ReentrantLock`

One of the things we can do with `ReentrantLock` is to attempt to acquire the lock but not block if unsuccessful. We can either not block and move on immediately, or attempt to acquire the lock for a fixed period of time before giving up.

We will now adapt the dining philosophers program to use `ReentrantLock`. In order to check that all of the philosophers are actually getting to both think and eat, we will add some code to gather that data.

Firstly we will look at the instance data for the new class `ForkLock`.

Using ReentrantLock

```
public class ForkLock {
   private Lock lock = new ReentrantLock();
   public boolean pickUp() {
      return lock.tryLock();
   }
   public void putDown() {
      lock.unlock();
   }
}
```

Listing 13: ForkLock.java

Using `ReentrantLock`

On the next slide we will see the `run()` method of new class `PhilosopherLock`.

The most important statements are those that attempt to acquire the locks. Since we are using the method `tryLock()` inside the `ForkLock` class, these statements will attempt to acquire the lock but move on if they can't.

The run() method

```
public void run() {
   while (!quit) {
      act("is thinking");
      thinkingCount++;
      if (fork1.pickUp()) {
         announce("has picked up left fork.");
         if (fork2.pickUp()) {
            announce("has picked up right fork.");
            act("is eating");
            eatingCount++;
            fork1.putDown();
            fork2.putDown();
         }
         else {
            fork1.putDown();
         }
      }
   }
}
```

Listing 14: PhilosopherLock.java

The announce() method simply prints the a message to the screen.

Program output

You will see when you look at the source code that there is some code in there to print out the 'stats' for each philosopher. Here is the output from one run of the program.

```
Philosopher 1 thought 22 times and ate 5 times.
Philosopher 2 thought 22 times and ate 4 times.
Philosopher 3 thought 27 times and ate 5 times.
Philosopher 4 thought 27 times and ate 7 times.
Philosopher 5 thought 32 times and ate 4 times.
```

We don't expect the output to be exactly the same for each philosopher. We're not looking for absolute equity but for all philosophers to have thought and eaten. The increased flexibility of `ReentrantLock` allowed that.

Semaphores

So far we have considered cases in which access to a resource (e.g. contained in a critical section) was permitted for only one thread at a time.

It can be the case that multiple threads (or processes) can be allowed to access a resource simultaneously. As we have seen, this is not suitable when accessing the shared resource (e.g. shared object) causes the program to behave incorrectly due to thread (or process) interference. But there are other cases when this might be appropriate.

Semaphores

Consider a scenario in which an application has a DBMS on the server side. The DBMS may only support a fixed number of concurrent connections, for example 5 connections. In the case where there are more users than allowable connections, access to the connections must be restricted.

So, we want to restrict access to this resource but not to one thread at a time but 5, in this example.

We still have to consider the effects of concurrent access but, in this case, the DBMS will manage that.

Semaphores

A semaphore is basically just an integer variable, that is limited to a fixed maximum value, with some methods to increase and decrease the value of the variable.

A semaphore is initialised to its maximum value, e.g. 5 as in the example above. In the terminology of the Java SDK, this means that the semaphore has 5 reusable 'permits' available. When a permit is requested, it is granted and the number of available permits is reduced to 4 (in this example). When another is requested, the number is reduced to 3, and so on. When the number of available permits reaches zero, no more can be granted.

Semaphores

When a process or thread, etc. has finished with the resource protected by the semaphore, then it returns the permit. If the number of available permits is zero, and one is returned, there is clearly now one available permit. The number of available permits never exceeds the maximum.

This type of semaphore is called a *counting semaphore*.

In the lab session you will implement a program that contains a *binary semaphore*. As its name suggests, a binary semaphore can only hold the values 0 and 1. A binary semaphore can be used as a lock.

Semaphores: example

The next few slides provide some example code for the use of a Java `Semaphore` to control access to a database connection.

Firstly, we will look at the database connection class.

Semaphores: example

```
public class DatabaseConnection {
   private static final int MAX_CONNECTIONS = 5;
   private Semaphore semaphore = new Semaphore(MAX_CONNECTIONS);
   public boolean connect() {
      boolean success = false;
      try {
         success = semaphore.tryAcquire(100, TimeUnit.MILLISECONDS);
      } catch (InterruptedException e) {
         e.printStackTrace();
         return success;
      }
      return success;
   }
   public void disconnect() {
      semaphore.release();
   }
   public int getNumPermitsAvailable() {
      return semaphore.availablePermits();
   }
}
```

Listing 15: DatabaseConnection.java

Semaphores: example

We can see that this database connection class has a semaphore that limits the number of permits to 5.

In the `connect()` method, which will be used by any thread trying to connect to the database, we see the use of the `Semaphore.tryAcquire()` method. This version of the method allows an attempt to last for a certain time period. In this example, that is 100 milliseconds.

We also see the `disconnect()` method which simply returns the permit.

Next we will look at a worker thread class.

Semaphores: example

```
public class DatabaseWorker implements Runnable {
   private DatabaseConnection dbconn;
   public DatabaseWorker(DatabaseConnection conn) {
      dbconn = conn;
   }
   public void run() {
      if (dbconn.connect()) {
         System.out.println("Database access granted");
         System.out.println("Working on database");
         try {
            Thread.sleep((long) (150 * Math.random()));
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
         System.out.println("Releasing database connection");
         dbconn.disconnect();
      }
      else {
         System.out.println("Database access denied");
      }
   }
}
```

Listing 16: DatabaseWorker.java

Semaphores: example

The `run()` method of the `DatabaseWorker` class attempts to connect to the database. If successful it does some work and then releases the connection.

If it is not successful it simply returns without doing anything. There are other approaches we could use here, such as looping until a connection is finally granted, etc. This version creates some possibly short-lived threads that don't achieve anything. Other threads will do something useful, however.

Finally, in the program itself, the `main()` method simply loops creating the worker threads.

Semaphores: example

```
public class CountingSemaphoreExample {
   private static DatabaseConnection dbconnection;
   public CountingSemaphoreExample() {
      dbconnection = new DatabaseConnection();
   }
   public static void main(String[] args) throws
       InterruptedException {
      new CountingSemaphoreExample();
      while (true) {
         DatabaseWorker worker = new
             DatabaseWorker(dbconnection);
         new Thread(worker).start();
         Thread.sleep(20);
         System.out.println("Permits available " +
             dbconnection.getNumPermitsAvailable());
      }
   }
}
```

Listing 17: CountingSemaphoreExample.java

Sample output (partial)

```
Working on database
Releasing database connection
Permits available 3
Database access granted
Working on database
Permits available 2
Database access granted
Working on database
Permits available 1
Database access granted
Working on database
Releasing database connection
Releasing database connection
Releasing database connection
Permits available 3
```

Semaphores

As you will see in the lab session, semaphores can be used as locks provided they are *binary*.

As you will also see in the lab, semaphores can also be used to allow literal concurrent access to a shared resource.