Threading in Java

Getting started Here is a how to create a thread in Java.

```
public class ThreadHelloWorld {
    public static class MyThread implements Runnable {
        @Override
        public void run() {
            System.out.println("Hello world");
        }
    }

    public static void main(String[] args) {
        Thread t = new Thread(new MyThread());
        t.start();
    }
}
```

We first create a static class that implements the Runnable interface. Anything that implements that interface must have a run method. That method contains the code we want the thread to run. This is analogous to the target function in Python. Then to create and run the thread, we make an object from our class and call its `start` method.

A program with multiple threads Here is an example of a program that creates and runs two threads.

```
public class TwoThreads {
    public static class MyThread1 implements Runnable {
        @Override
        public void run() {
            System.out.println("Hello from thread 1.");
        }
    }

    public static class MyThread2 implements Runnable {
        @Override
        public void run() {
            System.out.println("Hello from thread 2.");
        }
    }

    public static void main(String[] args) {
        Thread t1 = new Thread(new MyThread1());
        Thread t2 = new Thread(new MyThread2());

        t1.start();
        t2.start();
    }
}
```

Note that each thread has a different class. If we want both threads to run the exact same code, it would be okay to make them both objects from the same class.

Creating a list of threads Sometimes we want to create a bunch of threads. Rather than having a separate variable for each, we can create a list, like below:

```
public class ListOfThreads {

    public static class MyThread implements Runnable {
        private String name;

        public MyThread(String name) {
            this.name = name;
        }
        @Override
        public void run() {
            System.out.println("Hello from thread " + name);
        }
    }

    public static void main(String[] args) {
        List<Thread> threads = new ArrayList<Thread>();
        for (int i=0; i<10; i++) {
            Thread t = new Thread(new MyThread(""+i));
            threads.add(t);
            t.start();
        }
    }
}
```

For this program, to give each thread its own “name”, we introduce a class variable called name into the `MyThread` class and initialize it in a constructor. If you run this, notice that things don’t always print out in the same order. This is a fact of life of threads—you’re at the mercy of the OS scheduler in terms of when things will run.

The join method The `join` method is used to wait for a thread to finish. The code after the `join` method is called will not be run until after the thread is finished. Below is an example. Java requires us to do something about a potential exception, which is why there is a `throws` statement in the `main` method.

```
public class ThreadJoin {
    public static class MyThread implements Runnable {
        @Override
        public void run() {
            System.out.println("Hello world");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new MyThread());
        t.start();
        t.join();
        System.out.println("This won't print until after the thread is done.");
    }
}
```

Locks

To create a lock called `lock`, use the line below:

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

A reentrant lock is like an ordinary lock except that it allows the same thread to reacquire a lock it already has. This is useful if the thread’s code is recursive, and it also avoids us having to check to make sure we haven’t acquired the lock already, which can get tricky. Note that Python also has a reentrant lock, though we never talked about it.

Locks have two methods we will use: `lock` and `unlock`. Surround critical sections with `lock` and `unlock`. The `lock` method is called by a thread right before the critical section when it wants the lock. If the lock is free, then the thread can run the code in the critical section. If the lock is in use, the thread has to wait to run that code until the lock is free. The lock is freed by the `unlock` method.

In the example below, we use locks to protect a shared counter variable to avoid a well-known race condition. There are two threads each running the same code that adds 1 to a counter 10 million times. The code should print out 20000000. Try removing the lock and notice that you won’t necessarily get 20000000.

```
public class ThreadLock {
    static int count = 0;
    static Lock lock;
    public static class MyThread implements Runnable {
        @Override
        public void run() {
            for (int i=0; i<10000000; i++) {
                lock.lock();
                count++;
                lock.unlock();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        lock = new ReentrantLock();
        Thread t1 = new Thread(new MyThread());
        Thread t2 = new Thread(new MyThread());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(count);
    }
}
```

A different approach Java provides a feature called the `synchronized` keyword that allows us to do the above program a little differently.

```
public class ThreadLock2
{
    static int count = 0;

    public static class MyThread implements Runnable {
        @Override
        public void run() {
            for (int i=0; i<100000000; i++)
                increment();
        }
    }

    public static synchronized void increment() {
        count++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new MyThread());
        Thread t2 = new Thread(new MyThread());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(count);
    }
}
```

The `synchronized` keyword sort of acts like a lock that doesn't allow more than one thread to run the `increment` function at a time.

Condition variables

To create a condition variable called `cond`, use the lines below. Note that since condition variables are built on top of locks, we first have to declare a lock and then add the condition variable to it.

```
Lock lock = new ReentrantLock();
Condition cond = lock.newCondition();
```

Since condition variables are build on locks, any time we want to do something with a condition variable, we have to surround the code with calls to `lock` and `unlock`. The important methods of condition variables are these:

- `await` — Pauses the current thread until it receives a signal from another.
- `signal` — Tell a waiting thread to stop waiting.
- `signalAll` — If multiple threads are waiting, tell all of them to stop waiting.

Here is an example. Thread 1 and Thread 2 each print hello. Thread 1 is always supposed to print hello before Thread 2 does. In the code, there is a sleep statement that causes a pause of 1 to 3 seconds. This is to simulate the fact that we might not know which thread will run first.

```java
public class ConditionExample {
    public static Lock lock;
    public static Condition cond;
    public static boolean signalSent;

    public static class A implements Runnable  {
        @Override
        public void run() {
            Random random = new Random();
            try {
                Thread.sleep(random.nextInt(2000)+1000);
            } catch (InterruptedException e1) {}
            System.out.println("Hi from A.");
            lock.lock();
            cond.signal();
            signalSent = true;
            try {
                cond.await();
            } catch (InterruptedException e) {}
            lock.unlock();
        }
    }

    public static class B implements Runnable  {
        @Override
        public void run() {
            Random random = new Random();
            try {
                Thread.sleep(random.nextInt(2000)+1000);
            } catch (InterruptedException e1) {}
            if (!signalSent) {
                lock.lock();
                try {
                    cond.await();
                } catch (InterruptedException e) {}
                lock.unlock();
            }
            System.out.println("Hi from B.");
        }
    }

    public static void main(String[] args) {
        lock = new ReentrantLock();
        cond = lock.newCondition();
        signalSent = false;

        Thread t1 = new Thread(new A());
        Thread t2 = new Thread(new B());

        t1.start();
        t2.start();
    }
}
```

Condition variables are used to make it so that Thread 1 prints hello before Thread 2 does. Once Thread 1 prints hello, it sends a signal to Thread 2. We also unfortunately need to have a global `signalSent` variable. The reason we need it is that it might happen that Thread 1 runs and sends the signal to Thread 2 before Thread 2 has started running. In that case, the Thread 2 will completely miss the signal. When it starts running, it calls the wait method and will start waiting for a signal that has already been sent (and missed). This will cause it to wait forever. The boolean variable is set to true by the Thread 1 when it sends the message, and Thread 2 checks the value of that variable before it waits. Using this `signalSent` variable is not always needed with condition variables, but in this case it is.

You might notice a lot of try/catch blocks in the code above. Java requires us to do something about the potential for a thread to be interrupted while sleeping. Here though, we're doing the minimum possible and essentially ignoring the exceptions.

Personally, I find condition variables a bit of a pain to work with. Sometimes, they are the right thing to use, but more often I prefer to work with semaphores.

Semaphores

Semaphores are a versatile tool for working with threads. They can be used as locks or as condition variables, and they can be used to limit access to a shared resource to a specific number of threads. Here is how to create a semaphore called `sem` with an initial value of 5:

```
Semaphore sem = new Semaphore(5);
```

Semaphores have two useful methods:

- `acquire` — Decreases semaphore value by 1. If the resulting value is negative, the caller will go to sleep.
- `release` — Increases the semaphore value by 1. If there are any threads waiting (i.e., the value of the semaphore is negative before the increase), one will be woken up.

Here is an example that does the same thing as the condition variable example in the previous section. We have two Threads, 1 and 2, where Thread 2 is not supposed to print hello until 1 does.

```
public class SemaphoreExample {
    public static Semaphore semaphore;

    public static class A implements Runnable  {
        @Override
        public void run() {
            Random random = new Random();
            try {
                Thread.sleep(random.nextInt(2000)+1000);
            } catch (InterruptedException e1) {}
            System.out.println("Hi from A.");
            semaphore.release();
        }
    }

    public static class B implements Runnable  {
        @Override
        public void run() {
            Random random = new Random();
            try {
                Thread.sleep(random.nextInt(2000)+1000);
            } catch (InterruptedException e1) {}
            try {
                semaphore.acquire();
            } catch (InterruptedException e) {}
            System.out.println("Hi from B.");
        }
    }
```

```
        public static void main(String[] args) {
            semaphore = new Semaphore(0);

            Thread t1 = new Thread(new A());
            Thread t2 = new Thread(new B());

            t1.start();
            t2.start();
        }
    }
```

The way this works is that the semaphore is initially set to 0. If Thread 2 runs first, when it calls `acquire`, it will have to wait until Thread 1 calls release.

Semaphores are also useful for allowing a specific number of threads to access a particular resource at a given time. The next section has an example of that.

The Producer-Consumer problem

This is a famous threading problem. The idea is there are two types of threads: producers and consumers. The producers produce random numbers and put them onto a buffer. Consumers read those values and remove them from the buffer.

We need to make sure that the consumers never try to remove something from an empty buffer and that producers never let the buffer get overfull. We can accomplish this using two semaphores.

Here is the code. It creates one producer and three consumers.

```
    public class ProducerConsumer {
        private static Semaphore emptySemaphore;
        private static Semaphore fullSemaphore;
        private static List<Integer> buffer;

        public synchronized static void printMessage(String m) {
            System.out.println(m);
            System.out.println("Buffer = " + buffer);
        }

        public static class Producer implements Runnable {
            private String name;

            public Producer(String name) {
                this.name = name;
            }

            @Override
            public void run() {
                Random random = new Random();
                while (true) {
                    try {
                        Thread.sleep(random.nextInt(400)+100);
                    } catch (InterruptedException e) {}
                    int r = random.nextInt(100) + 1;
                    try {
                        fullSemaphore.acquire();
                    } catch (InterruptedException e1) {}
                    buffer.add(r);
                    printMessage("Thread " + name + " produced " + r);
                    emptySemaphore.release();
                }
            }
        }

        public static class Consumer implements Runnable {
            private String name;
            public Consumer(String name) {
                this.name = name;
            }

            @Override
            public void run() {
                Random random = new Random();
                while (true) {
                    try {
                        emptySemaphore.acquire();
                    } catch (InterruptedException e1) {}
                    int v = buffer.remove(0);
                    fullSemaphore.release();
                    printMessage("Thread " + name + " consumed " + v);
                    try {
                        Thread.sleep(random.nextInt(1000)+1000);
                    } catch (InterruptedException e) {}
                }
            }
        }

        public static void main(String[] args)
        {
            buffer = new ArrayList<Integer>();
            fullSemaphore = new Semaphore(3);
            emptySemaphore = new Semaphore(0);

            Thread p1 = new Thread(new Producer("P1"));
            Thread c1 = new Thread(new Consumer("C1"));
            Thread c2 = new Thread(new Consumer("C2"));
            Thread c3 = new Thread(new Consumer("C3"));

            p1.start();
            c1.start();
            c2.start();
            c3.start();
        }
    }
```

Here are the two semaphores that we use to solve this problem:

- `fullSemaphore` — Initially set to 3. Producers call `acquire` on this when they want to put something onto the buffer. Consumers call `release` on this when they take something off of the buffer. Its job is to prevent the buffer from getting overfull (having more than 3 things on it). Since `acquire` decreases the value by 1 and `release` increases the value by 1, its value will track how many things are in the array, and once the array is full, the semaphore's value will be 0 and any thread that calls `acquire` will have to sleep until the semaphore value positive again, preventing it from adding to the list until that point.

- `emptySemaphore` — Initially set to 0. Consumers call `acquire` on this when they want to pull something off of the buffer. Producers call `release` on this when they put something on the buffer. It's job is to prevent consumers from trying to take something off of an empty buffer. It's essentially acting like a condition variable, causing threads to sleep if they want to remove from the list while it is empty.

Working through the code, we first see the `printMessage` function. This uses the synchronized keyword to prevent things from printing out in a jumbled mess.

The `produce` function loops forever. In that loop, the first thing it does is sleep for a random amount of time between .1 and .5 seconds. This is to simulate the time it takes to produce something. Then it calls `acquire` on `fullSemaphore`. It is allowed to put something onto the buffer unless the buffer is full, in which case the semaphore's value will be 0. In that case, the producer will have to wait until one of the consumers pulls something off the buffer and calls `release`. Once the producer is done appending to the buffer, it calls `release` on `emptySemaphore`. This serves as a signal to any waiting consumers that there is something on the buffer.

The `consume` function is somewhat similar. It also loops forever. The consumer calls acquire on `emptySemaphore`, which will cause it to wait until there is something on the buffer. Once it's allowed to, it pops something off the buffer and then calls `release` on the `fullSemaphore` to signal to producers that there is room on the buffer. After that, it sleeps for a random amount of time between .1 and .2 seconds to simulate the time it takes to consume the item.

It's worth running the code to see how it all works. In particular, try changing the amounts of time things sleep for to speed up or slow down the producers and consumers. You can also quickly add more producers and consumers.

You might be asking what the reason is to use semaphores here. Why not just use some if statements that check if the buffer is full or empty? The reason is race conditions. If, for instance, a consumer gets interrupted in between checking the if condition and popping, another consumer might run in the meantime and pop from the buffer. When the original consumer resumes, it will end up popping from an empty buffer, which will cause an error.