<https://codegym.cc/groups/posts/220-whats-the-difference-between-a-mutex-a-monitor-and-a-semaphore>

What's the difference between a mutex, a monitor, and a semaphore?

Hi! When you studied multithreading on CodeGym, you frequently encountered the concepts of "mutex" and "monitor". Without peeking, can you say how they differ? :) If yes, well done! If not (this is most common), that's no surprise. "Mutex" and "monitor" are actually related concepts. Additionally, when you read lessons and watch videos about multithreading on other websites, you'll come across another similar concept: "semaphore". It also has a very similar function to monitors and mutexes. That's why we're going to investigate these three terms. We'll look at a few examples and come to a definitive understanding of how these concepts differ from one another :)

## Mutex

A mutex (or lock) is a special mechanism for synchronizing threads. One is "attached" to every object in Java — you already know that :) It doesn't matter if you use standard classes or create your own classes, e.g. Cat and Dog: **all objects of all classes have a mutex**. The term "mutex" comes from "MUTual EXclusion", which perfectly describes its purpose. As we said in one of our previous lessons, **a mutex makes it possible to ensure that only one thread at a time has access to the object.** A popular real-life example of a mutex involves toilets. When a person enters a toilet partition, he locks the door from the inside. The toilet is like an object that can be accessed by multiple threads. The lock on the partition door is like a mutex, and the line of people outside represents threads. The lock on the door is the toilet's mutex: it ensures that only one person can get inside. In other words, only one thread at a time can work with shared resources. Attempts by other threads (people) to gain access to occupied resources will fail. A mutex has several important features. **First**, only two states are possible: "unlocked" and "locked". This helps us understand how it works: you can draw parallels with Boolean variables (true/false) or binary numbers (0/1). **Second**, the state cannot be controlled directly. Java has no mechanism that would let you explicitly take an object, get its mutex, and assign the desired status. In other words, you can't do something like:

Object myObject = **new** Object();

Mutex mutex = myObject.getMutex();

mutex.free();

This means that you can't release an object's mutex. Only the Java machine has direct access to it. Programmers work with mutexes through the tools of the language.

## Monitor

A monitor is an additional "superstructure" over a mutex. In fact, **a monitor is a chunk of code that is "invisible" to the programmer.** When we spoke about mutexes earlier, we gave a simple example:

**public** **class** Main {

**private** Object obj = **new** Object();

**public** **void** doSomething() {

// ...some logic, available for all threads

**synchronized** (obj) {

// Logic available to just one thread at a time

}

}

}

In the code block marked with the **synchronized** keyword, the mutex of our **obj** object is acquired. Great, we can acquire the lock, but how exactly is the "protection" provided? When we see the word synchronized, what prevents the other threads from entering the block? The protection comes from a monitor! The compiler converts the **synchronized** keyword into several special pieces of code. Once again, let's return to our example with the **doSomething()** method. We'll add to it:

**public** **class** Main {

**private** Object obj = **new** Object();

**public** **void** doSomething() {

// ...some logic, available for all threads

// Logic available to just one thread at a time

**synchronized** (obj) {

/\* Do important work that requires that the object

be accessed by only one thread \*/

obj.someImportantMethod();

}

}

}

Here is what happens "under the hood" after the compiler converts this code:

**public** **class** Main {

**private** Object obj = **new** Object();

**public** **void** doSomething() **throws** InterruptedException {

// ...some logic, available for all threads

// Logic available to just one thread at a time:

/\* as long as the object's mutex is busy,

all the other threads (except the one that acquired it) are put to sleep \*/

**while** (obj.getMutex().isBusy()) {

Thread.sleep(1);

}

// Mark the object's mutex as busy

obj.getMutex().isBusy() = true;

/\* Do important work that requires that the object

be accessed by only one thread \*/

obj.someImportantMethod();

// Free the object's mutex

obj.getMutex().isBusy() = false;

}

}

Of course, this isn't a real example. Here, we used Java-like code to depict what happens inside the Java machine. That said, this pseudo-code gives an excellent understanding of what actually happens with the object and threads inside the synchronized block and how the compiler converts this keyword into several statements that are "invisible" to the programmer. Basically, **Java uses the synchronized keyword to represent a monitor**. All the code that appears instead of the synchronized keyword in the last example is the monitor.

## Semaphore

Another word that you'll encounter in your personal study of multithreading is "semaphore". Let's figure out what this is and how it differs from a monitor and a mutex. A semaphore is a tool for synchronizing access to some resource. **Its distinctive feature is that it uses a counter to create the synchronization mechanism.** The counter tells us how many threads can simultaneously access the shared resource. Semaphores in Java are represented by the Semaphore class. When creating semaphore objects, we can use the following constructors:

Semaphore(**int** permits)

Semaphore(**int** permits, **boolean** fair)

We pass the following to the constructor:

**int permits** — the initial and maximum value of the counter. In other words, this parameter determines how many threads can simultaneously access the shared resource;

* **boolean fair** — establishes the order in which threads will gain access. If fair is true, then access is granted to waiting threads in the order in which they requested it. If it is false, then the order is determined by the thread scheduler.

A classic example of semaphore use is the dining philosopher problem. To facilitate understanding, we'll simplify it a little bit. Imagine that we have 5 philosophers who need to eat lunch. Additionally, we have one table that can simultaneously accommodate no more than two people. Our task is to feed all the philosophers. None of them should go hungry, and none of them should "block" each other when trying to sit down at the table (we must avoid deadlock). Here's what our philosopher class will look like:

**class** Philosopher **extends** Thread {

**private** Semaphore sem;

// Did the philosopher eat?

**private** **boolean** full = false;

**private** String name;

Philosopher(Semaphore sem, String name) {

**this**.sem=sem;

**this**.name=name;

}

**public** **void** run()

{

**try**

{

// If the philosopher has not eaten

**if** (!full) {

// Ask the semaphore for permission to run

sem.acquire();

System.out.println(name + " takes a seat at the table");

// The philosopher eats

sleep(300);

full = true;

System.out.println(name + " has eaten! He leaves the table");

sem.release();

// The philosopher leaves, making room for others

sleep(300);

}

}

**catch**(InterruptedException e) {

System.out.println("Something went wrong!");

}

}

}

And here's the code to run our program:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Semaphore sem = **new** Semaphore(2);

**new** Philosopher(sem, "Socrates").start();

**new** Philosopher(sem,"Plato").start();

**new** Philosopher(sem,"Aristotle").start();

**new** Philosopher(sem, "Thales").start();

**new** Philosopher(sem, "Pythagoras").start();

}

}

We created a semaphore whose counter is set to 2 to satisfy the condition: only two philosophers can eat at the same time. That is, only two threads can run at the same time, because our Philosopher class inherits Thread! The **acquire()** and **release()** methods of the Semaphore class control its access counter. The acquire() method asks the semaphore for access to the resource. If the counter is >0, then access is granted and the counter is reduced by 1. The release() method "releases" the previously granted access, returning it to the counter (increases the semaphore's access counter by 1). What do we get when we run the program? Is the problem solved? Will our philosophers not fight while they wait for their turn? :) Here's is the console output we got:

Socrates takes a seat at the table

Plato takes a seat at the table

Socrates has eaten! He leaves the table

Plato has eaten! He leaves the table

Aristotle takes a seat at the table

Pythagoras takes a seat at the table

Aristotle has eaten! He leaves the table

Pythagoras has eaten! He leaves the table

Thales takes a seat at the table

Thales has eaten! He leaves the table

We did it! And though Thales had to dine alone, I don't think we've offended him :) **You may have noticed some similarities between a mutex and a semaphore.** Indeed, they have the same mission: to synchronize access to some resource. The only difference is that an object's mutex can be acquired by only one thread at a time, while in the case of a semaphore, which uses a thread counter, several threads can access the resource simultaneously. This isn't just a coincidence :) **A mutex is actually a semaphore** with a count of 1. In other words, it's a semaphore that can accommodate a single thread. It's also known as a "binary semaphore" because its counter can have only 2 values — 1 ("unlocked") and 0 ("locked"). That's it! As you can see, it's not so confusing after all :)