Jacob Taggart
Week 3 Day 2 Research

Week 3 Day 2 Research

1. Define the following as it relates to threading:

    a) Locks

    Lock is a word used to describe the access a thread has to a specific resource based on if the resource is being accessed by another thread or not at the same time. For example, if two threads are asynchronously trying to access the same resource at the same time, the resource will be locked as the two operations trying to get the resource will result in a conflict known as race condition.

    b) Mutex

    First off, mutex stands for mutual exclusion. Mutex locking is a way to ensure that two threads which would be in a race condition, actually only access the resource both wants one at a time. Whichever thread that is used as the first one to access the mutex lock has to release the lock upon finishing of the use and then and only then will the lock be open for the second thread to access the resource. Then, the second thread will have to perform its operation using the resource before leaving and unlocking the mutex for any further operation. Mutex locks work via the underlying code using a way to coordinate the timing of the release and access to the resource as well as the other being coordinated to enter upon the first thread releasing the lock to the resource.

    c) Semaphores

    Semaphores are often confused in the way they are interpreted. The goal overall is similar to mutex locks. By and large, both ways of locking and accessing resources aim to extinguish the problem of race condition. There is a significant difference between the two, though. Whereas mutex patterns access and release a resource lock only via the thread chosen to access it at the time completing the use of the resource, semaphores rely on signaling from one thread to another in order to indicate whether the lock is open or closed. The thread which is accessing a lock is the one signaling to the other thread which is receiving/consuming the

signal. Upon release of the lock, the first thread signals directly to the waiting thread that the lock to the resource has been released and is open for use.

d) Synchronized

"Synchronized" is a keyword used to indicate to a block of code inside an asynchronous method that the block being tagged can only access the necessary resource(s) one at a time. Often times this can be carried out inside the otherwise asynchronous method by immediately calling the "synchronized" keyword before accepting an object or multiple (in this case, it would be "this" since it is surrounding the body of the asynchronous method which encloses the synchronized task body as well). Doing this tells that whatever portion of the code is surrounded by the synchronized block of the body can only access the necessary resource(s) sequentially. Otherwise without the "synchronized" keyword and enclosed statements, the enclosed code will result in multiple threads trying to access the same resource(s) at once. While helpful in simpler scenarios, the "synchronized" keyword is the most basic way to synchronize otherwise asynchronous operations when developing in Android, which means it can be problematic when threading gets more complex.

e) Volatile

The "volatile" keyword is, above all else, concerned with getting the correct variable values from the cache when using multi-threading. In the English language, "volatile" signifies that whatever is being described has a constant possibility for change and cannot be predicted because of this state of flux. This is exactly what the keyword is telling the operating system when used in development. There is no way of knowing when the Java JVM will save certain objects or variable state's to main memory or when the JVM will load the necessary variable states or objects into the CPU cache memory. So for multiple threads, each has its own cache memory which needs to pull from the main memory and which needs to save to main memory. But because there's no easy way to predict when or how to coordinate this data exactly, if one thread needed to access another thread's data for an operation, there would be no reasonable guarantee the retrieved data will return what would be expected logically. The reason "volatile" can be necessary when using an asynchronous pattern is because of this unpredictability. One way to think of it is if two different threads asynchronously were using the same name for a counter variable. Obviously the two different thread operations may be using that variable name for different

iterative reasons and, thus would expect a value different from the other thread's cache memory. But because of the volatility of JVM loading and saving into and out of the cache, there is no knowing with certainty what value will be gotten from the cache. So even though it causes the efficiency to slow down, the keyword "volatile" ensures that any variable or object tagged by it will be immediately saved to the main memory immediately. So any time the data value of a variable or object is changed, it is saved to the main memory directly - ensuring that variable and object data consistency across threads is achievable when mulit-hreading ensues. However, this method still isn't without its flaws - namely that from the same counter variable example. The miniscule time frame between one thread reading and one thread writing to the main memory for the same counter variable can cause a race condition.

f) Atomic

Atomic variables are similar to volatile variables with one exception, the atomic variables have a method to bypass the need for a "synchronized" keyword. That's because atomic variables have the "compare and swap" pattern built in. Compare and swap works how it sounds. It solves the race condition issue from volatile variables by comparing to see if the value of a variable is the same as what is in the main memory before getting the new value and replacing it. This ensures that a different thread is not currently trying to alter the same variable in the main memory at the same time the first thread is.

2. What is a deadlock condition?

A deadlock condition is when multiple threads are accessing different resources at the same time, but for their next operations they need the next resource which one of the other multiple threads is currently occupying. Each individual thread realizes that the next resource is currently accessed and waits for it to be released. If there are a set of resources that all of the threads need at some point in their operations, this can cause a deadlock condition very easily. One way to think of it is gridlock. All the cars on the interstate need to use the same resources, but they are all occupied. The difference between the metaphor and the actual deadlock is that in gridlock there is usually a small flow that eventually allows movement while in deadlock, there will be absolutely no movement of threads to resources as the threads have nowhere they can go unless one or more resources are freed up for whatever reason.

3. What is a race condition?

A race condition in multiple-thread operations is an issue in which two threads need to use the same resource at the same time to carry out an operation. When two threads try to access the same “lock”, neither gets access to it as the “lock” is trying to be accessed by two “keys” (threads) at once.

4. What is a memory leak?

   Memory leak is generally a scenario in which an object or multiple objects are unused for a long period of time, but the JVM garbage collection is unable to mark it for deallocation because in the code, the object(s) is still being referenced. One common way a memory leak occurs is by declaring a complex object as a static field. This makes it so the memory container is always existent in the memory and if the object is not removed somehow, the last object state passed will remain there for a potentially very long time. Another common and consequential way to leak memory is by leaving data streams unclosed. It’s easy to forget to use the close() method, but it’s absolutely crucial to avoid a memory leak. A hashSet with no implementation of the hashCode() method and no implementation of the equals() method can result in memory leak. One way to detect possible memory leaks is by enabling verbose garbage collection in Java.

5. What is an ANR and what are some common causes?

   ANR is an acronym which means “Application Not Responding”. If the application is running in the foreground of the application stack, a system dialogue will open to ask if the user would like to force the app to close. This happens when the main thread cannot properly respond to an action that changes the user interface. They can be triggered when five seconds pass and the application has not yet responded to a UI interaction or a broadcast receiver operation. This can also happen when an activity which is not in the foreground takes a considerable amount of time to finish completing a broadcast receiver application. Note that ANR’s are different from standard application crashes. These stem from things like Edit Texts requiring user input on the intent side of an activity, but the code on the sending end of an activity not actually ensuring those fields are filled in and then sending it to the intent.