FullStack.Cafe - Kill Your Tech Interview

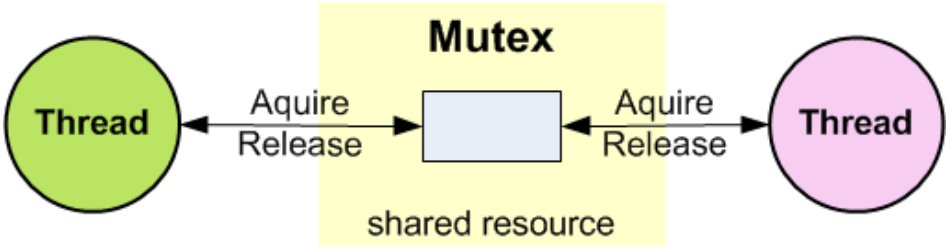
Q1: What is a Mutex? ☆☆

Topics: Concurrency

Answer:

A **Mutex** is a mutually exclusive flag. It acts as a gate keeper to synchronise two threads. When you have two threads attempting to access a single resource, the general pattern is to have the first block of code attempting access, to set the *mutex* before entering the code. When the second code block attempts access, it sees that the *mutex* is set and waits until the first block of code is complete (and un-sets the mutex), then continues.

Specific details of how this is accomplished obviously varies greatly by programming language.

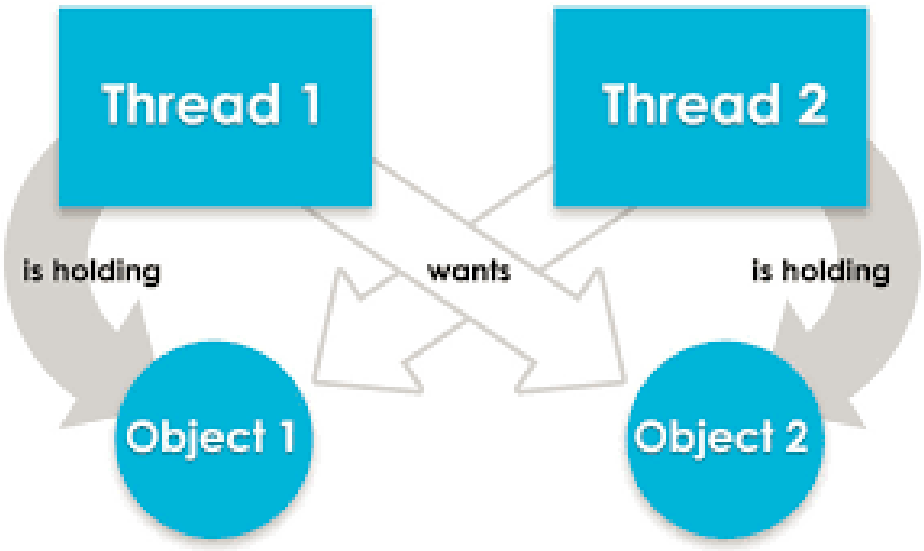


Q2: What is a Deadlock? ☆☆

Topics: Concurrency

Answer:

- A **lock** occurs when multiple processes try to access the same resource at the same time. One process loses out and must wait for the other to finish.
- A **deadlock** occurs when the waiting process is still holding on to another resource that the first needs before it can finish.



So, an example:

Resource A and resource B are used by process X and process Y

- X starts to use A.
- X and Y try to start using B
- Y 'wins' and gets B first
- now Y needs to use A
- A is locked by X, which is waiting for Y

```
Thread 1               Thread 2

Lock1->Lock();         Lock2->Lock();
WaitForLock2();        WaitForLock1();   <-- Oops!
```

The best way to avoid deadlocks is to avoid having processes cross over in this way. Reduce the need to lock anything as much as you can. In databases avoid making lots of changes to different tables in a single transaction, avoid triggers and switch to optimistic/dirty/nolock reads as much as possible.

Q3: Explain the difference between Asynchronous and Parallel programming? ☆☆

Topics: Concurrency

Answer:

When you run something **asynchronously** it means it is non-blocking, you execute it without waiting for it to complete and carry on with other things. **Parallelism** means to run multiple things at the same time, in parallel. Parallelism works well when you can separate tasks into independent pieces of work. Async and Callbacks are generally a way (tool or mechanism) to express concurrency i.e. a set of entities possibly talking to each other and sharing resources.

Take for example rendering frames of a 3D animation. To render the animation takes a long time so if you were to launch that render from within your animation editing software you would make sure it was running *asynchronously* so it didn't lock up your UI and you could continue doing other things. Now, each frame of that animation can also be considered as an individual task. If we have multiple CPUs/Cores or multiple machines available, we can render multiple frames in *parallel* to speed up the overall workload.

Q4: What is the difference between Concurrency and Parallelism? ☆☆☆

Topics: Software Architecture Concurrency

Answer:

- **Concurrency** is when two or more tasks can start, run, and complete in overlapping time **periods**. It doesn't necessarily mean they'll ever both be running **at the same instant**. For example, *multitasking* on a single-core machine.

- **Parallelism** is when tasks *literally* run at the same time, e.g., on a multicore processor.

For instance a bartender is able to look after several customers while he can only prepare one beverage at a time. So he can provide *concurrency without parallelism*.

Q5: What is a Race Condition? ☆☆☆

Topics: Concurrency

Answer:

A race condition is a situation on concurrent programming where two concurrent threads or processes compete for a resource and the resulting final state depends on who gets the resource first.

Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". E.g:

```
if (x == 5) // The "Check"
{
   y = x * 2; // The "Act"

   // If another thread changed x in between "if (x == 5)" and "y = x * 2" above,
   // y will not be equal to 10.
}
```

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing.

In order to prevent race conditions from occurring, you would typically put a lock (Mutex or Semaphores) around the shared data to ensure only one thread can access the data at a time. This would mean something like this:

```
// Obtain lock for x
if (x == 5)
{
   y = x * 2; // Now, nothing can change x until the lock is released.
              // Therefore y = 10
}
// release lock for x
```

Q6: What is the meaning of the term “Thread-Safe”? ☆☆☆

Topics: Concurrency

Problem:

Does it mean that two threads can't change the underlying data simultaneously? Or does it mean that the given code segment will run with predictable results when multiple threads are executing that code segment?

Solution:

- **Thread-safe** code is code that will work even if many Threads are executing it simultaneously within the same process. This often means, that internal data-structures or operations that should run uninterrupted are protected against different modifications at the same time.
- Another definition may be like - a class is** thread-safe** if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime

environment, and with no additional synchronisation or other coordination on the part of the calling code.

Q7: Explain Deadlock to 5 years old ☆☆☆

Topics: Concurrency

Answer:

Jack and Jill share a sparse kitchen that has only one of everything. They both want to make a sandwich at the same time. Each needs a slice of bread and each needs a knife, so they both go to get the loaf of bread and the knife from the kitchen.

Jack gets the knife first, while Jill gets the loaf of bread first. Now Jack tries to find the loaf of bread and Jill tries to find the knife, but both find that what they need to finish the task is already in use. If they both decide to wait until what they need is no longer in use, they will wait for each other forever. **Deadlock**.

Q8: Is there any difference between a Binary Semaphore and Mutex? ☆☆☆

Topics: Concurrency

Answer:

- A **mutex** (or Mutual Exclusion Semaphores) is a **locking mechanism** used to synchronize *access* to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there will be ownership associated with mutex, and only the owner can release the lock (mutex).

- **Semaphore** (or Binary Semaphore) is **signaling mechanism** ("I am done, you can carry on" kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend called you, an interrupt will be triggered upon which an interrupt service routine (ISR) will signal the call processing task to wakeup. A binary semaphore is NOT protecting a resource from access. Semaphores are more suitable for some synchronization problems like producer-consumer.

Short version:

- A **mutex** can be released only by **the thread that had acquired it**.
- A **binary semaphore** can be signaled **by any thread** (or process).

Q9: How much work should I place inside a lock statement? ☆☆☆

Topics: Concurrency

Answer:

It is first and foremost a question of correctness. You should not care so much about efficiency trade-offs, but more about correctness.

- Do as little work as possible while locking a particular object. Locks that are held for a long time are subject to contention, and contention is slow. Note that this implies that the *total* amount of code in a *particular* lock and the *total* amount of code in *all lock statements that lock on the same object* are both relevant.
- Have as few locks as possible, to make the likelihood of deadlocks (or livelocks) lower.
- If you want concurrency, use processes as your unit of concurrency. If you cannot use processes then use application domains. If you cannot use application domains, then have your threads managed by the Task Parallel Library and write your code in terms of high-level tasks (jobs) rather than low-level threads (workers).

- Benchmarking and being aware that there are more options than "lock everything everywhere" and "lock only the bare minimum".

Q10: Write a function that guarantees to never return the same value twice ☆☆☆

Topics: Concurrency Brain Teasers

Problem:

Write a function that is guaranteed to never return the same value twice. Assume that this function will be accessed by multiple machines *concurrently*.

Solution:

1. Just toss a simple (threadsafe) counter behind some communication endpoint:

```
long x = long.MinValue;
public long ID(){
 return Interlocked.Increment(ref x);
}
```

2. Let interviewer be the one that follow up with those problems:
    - Does it need to survive reboots?
    - What about hard drive failure?
    - What about nuclear war?
    - Does it need to be random?
    - How random?
3. If they made it clear that it has to be unique across reboots and across different machines, I'd give them a function that calls into the standard mechanism for creating a new GUID, whatever that happens to be in the language being used. This is basically the problem that guids solve. Producing a duplicate Guid, no matter its format, is the most difficult lottery on the planet.

Q11: Explain what is a Race Condition to 5 years old ☆☆☆☆

Topics: Concurrency

Answer:

- You are planning to go to a movie at 5 pm. You inquire about the availability of the tickets at 4 pm. The representative says that they are available.
- You relax and reach the ticket window 5 minutes before the show. I'm sure you can guess what happens: it's a full house.

The problem here was in **the duration between the check and the action**. You inquired at 4 and acted at 5. In the meantime, someone else grabbed the tickets. That's a race condition - specifically a "check-then-act" scenario of race conditions.

Q12: What is a Data Race? ☆☆☆☆

Topics: Concurrency

Answer:

A *data race* happens when there are two memory accesses in a program where both:

- target the same location
- are performed concurrently by two threads
- are not reads
- are not synchronization operations

Q13: What's the difference between Deadlock and Livelock? ☆☆☆☆

Topics: Concurrency

Answer:

- In concurrent computing, a **deadlock** is a state in which each member of a group of actions, is waiting for some other member to release a lock

- A **livelock** is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of **resource starvation**; the general definition only states that a specific process is not progressing.

Livelock is a risk with some algorithms that detect and recover from deadlock. If more than one process takes action, the deadlock detection algorithm can be repeatedly triggered. This can be avoided by ensuring that only one process (chosen randomly or by priority) takes action.

Q14: Provide some real-live examples of Livelock ☆☆☆☆

Topics: Concurrency

Answer:

- A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

- Example of livelock will happen where a husband and wife are trying to eat soup, but only have one spoon between them. Each spouse is too polite, and will pass the spoon if the other has not yet eaten.

- Deadlock detection may cause livelock. If two threads detect a deadlock, and try to "step aside" for each other, without care they will end up being stuck in a loop always "stepping aside" and never managing to move forwards.

Q15: What are some advantages of Lockless Concurrency? ☆☆☆☆

Topics: Concurrency

Answer:

Lockless concurrency eliminates *deadlocks* and provides the nice advantage *that readers never have to wait for other readers*. This is especially useful when many threads will be reading data from a single source.

Lockless programming, is a set of techniques for safely manipulating shared data *without using locks*. There are lockless algorithms available for passing messages, sharing lists and queues of data, and other tasks. Lockless

programming is pretty complicated. e.g. All purely functional data structures are inherently lock-free, since they are immutable.

Q16: What is Starvation? ☆☆☆☆

Topics: Concurrency

Answer:

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads or threads with more "prioroty". For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

One more real live example may be this one. Imagine you're in a queue to purchase food at a restaurant, for which pregnant women *have priority*. And there's just a whole bunch of pregnant women arriving all the time. You'll soon be starving.

Q17: Compare Actor Model with Threading Model for concurrency ☆☆☆☆

Topics: Concurrency

Answer:

- The **actor model** operates on message passing. Individual processes (actors) are allowed to send messages asynchronously to each other.
- What distinguishes this from what we normally think of as the **threading model**, is that there is (in theory at least) no shared state. And if one believes (justifiably, I think) that shared state is the root of all evil, then the actor model becomes very attractive.

Q18: What is Green Thread? ☆☆☆☆

Topics: Concurrency

Answer:

A **Green Thread** is a thread that is scheduled by a virtual machine (VM) instead of natively by the underlying operating system. Green threads emulate multithreaded environments without relying on any native OS capabilities, and they are managed in user space instead of kernel space, enabling them to work in environments that do not have native thread support.

Green Thread usually used when the OS does not provide a thread API, or it doesn't work the way you need. Thus, the advantage is that you get thread-like functionality at all. The disadvantage is that green threads can't actually use multiple cores. In the context of Java specifically, there were a few early JVMs that used green threads (IIRC the Blackdown JVM port to Linux did), but nowadays all mainstream JVMs use real threads. There may be some embedded JVMs that still use green threads.

Q19: Two customers add a product to the basket in the same time whose the stock was only one (1). What will you do? ☆☆☆☆

Topics: Concurrency Software Architecture

Problem:

What is the best practice to manage the case where two customers add in the same time a product whose the stock was only 1?

Solution:

There is no perfect answer for this question and all depends on details but you have some options:

1. As a first 'defense line' I would try to avoid such situations at all, by simply not selling out articles that low if any possible.
2. You reserve an item for the customer for a fix time say (20 minutes) after they have added it to the basket – after they time they have to recheck the stock level or start again. This is often used to ticket to events or airline seats
3. For small jobs the best way is to do a final check right before the payment, when the order is actually placed. In worst case you have to tell you customer that you where running out of stock right now and offer alternatives or discount coupon.
4. Try to fulfil both orders later - just cause you don't have stock right now, doesn't mean you can't find it in an emergency. If you can't then someone has to contact the user who lucked out and apologise.

Side note:

1. The solution to do the double check when adding something to the basket isn't very good. People put a lot in baskets without ever actually placing an order. So this may block this article for a certain period of time.

Q20: What happens if you have a "race condition" on the lock itself? ☆☆☆☆☆

Topics: Concurrency

Problem:

For example, two different threads, perhaps in the same application, but running on different processors, try to acquire a lock at the exact same time. What happens then? Is it impossible, or just plain unlikely?

Solution:

Have a "race condition" on the lock itself is impossible. It can be implemented in different ways, e.g., via the *Compare-and-swap* where the hardware guarantees sequential execution. It can get a bit complicated in presence of multiple cores or even multiple sockets and needs a complicated protocol (MESI protocol) between the cores, but this is all taken care of in hardware.

Compare-and-swap (CAS) is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail.

FullStack.Cafe - Kill Your Tech Interview

Q1: What is the difference between Race Condition and Data Races? Are they the same? ☆☆☆☆☆

Topics: Concurrency

Answer:

- A **data race** occurs when 2 instructions from different threads access the same memory location, at least one of these accesses is a write and there is no synchronization that is mandating any particular order among these accesses.

- A **race condition** is a *semantic error*. It is a flaw that occurs in the timing or the ordering of events that leads to erroneous program behavior. Many race conditions can be caused by data races, but this is not necessary.

The Race Condition and Data Races are not the same thing. They are not a subset of one another. They are also neither the necessary, nor the sufficient condition for one another.