educative

I/O Bound vs CPU Bound

We delve into the characteristics of programs with different resource-use profiles and how that can affect program design choices.

I/O Bound vs CPU Bound

We write programs to solve problems. Programs utilize various resources of the computer systems on which they run. For instance a program running on your machine will broadly require:

- CPU Time
- Memory
- Networking Resources
- Disk Storage

Depending on what a program does, it can require heavier use of one or more resources. For instance, a program that loads gigabytes of data from storage into main memory would hog the main memory of the machine it runs on. Another can be writing several gigabytes to permanent storage, requiring abnormally high disk i/o.

CPU Bound

Programs which are compute-intensive i.e. program execution requires very high utilization of the CPU (close to 100%) are called CPU bound programs. Such programs primarily depend on improving

CPU speed to decrease program completion time. This could include programs such as data crunching, image processing, matrix multiplication etc.

If a CPU bound program is provided a more powerful CPU it can potentially complete faster. Eventually, there is a limit on how powerful a single CPU can be. At this point, the recourse is to harness the computing power of multiple CPUs and structure your program code in a way that can take advantage of the multiple CPU units available. Say we are trying to sum up the first 1 million natural numbers. A single-threaded program would sum in a single loop from 1 to 1000000. To cut down on execution time, we can create two threads and divide the range into two halves. The first thread sums the numbers from 1 to 500000 and the second sums the numbers from 500001 to 1000000. If there are two processors available on the machine, then each thread can independently run on a single CPU in parallel. In the end, we sum the results from the two threads to get the final result. Theoretically, the multithreaded program should finish in half the time that it takes for the single-threaded program. However, there will be a slight overhead of creating the two threads and merging the results from the two threads.

Multithreaded programs can improve performance in cases where the problem lends itself to being divided into smaller pieces that different threads can work on independently. This may not always be true though.

I/O Bound

I/O bound programs are the opposite of CPU bound programs. Such programs spend most of their time waiting for input or output operations to complete while the CPU sits idle. I/O operations can consist of operations that write or read from main memory or network interfaces. Because the CPU and main memory are physically separate a data bus exists between the two to transfer bits to and fro. Similarly, data needs to be moved between network interfaces and CPU/memory. Even though the physical distances are tiny, the time taken to move the data across is big enough for several thousand CPU cycles to go waste. This is why I/O bound programs would show relatively lower CPU utilization than CPU bound programs.

Notes

Both types of programs can benefit from concurrent architectures. If a program is CPU bound we can increase the number of processors and structure our program to spawn multiple threads that individually run on a dedicated or shared CPU. For I/O bound programs, it makes sense to have a thread give up CPU control if it is waiting for an I/O operation to complete so that another thread can get scheduled on the CPU and utilize CPU cycles. Different programming languages come with varying support for multithreading. For instance, Javascript is single-threaded, Java provides full-blown multithreading and Python is *sort* of multithreaded as it can only have a single thread in running state because of its global interpreter lock (GIL) limitation. However, all three languages support asynchronous programming models which is another way for programs to be concurrent (but not parallel).

For completeness we should mention that there are also memory-bound programs that depend on the amount of memory available to speed up execution.

← Back
Synchronous vs Asynchronous

Next →
Throughput vs Latency

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/io-bound-vs-cpu-bound__the-basics__java-multithreading-for-senior-engineering-interviews)

Throughput vs Latency

This lessons discusses throughput and latency in the context of concurrent systems.

Throughput

> Throughput is defined as the *rate of doing work* or how much work gets done per unit of time. If you are an Instagram user, you could define throughput as the number of images your phone or browser downloads per unit of time.

Latency

> Latency is defined as the *time required to complete a task or produce a result*. Latency is also referred to as *response time*. The time it takes for a web browser to download Instagram images from the internet is the latency for downloading the images.

Throughput vs Latency

> The two terms are more frequently used when describing networking links and have more precise meanings in that domain. In the context of concurrency, throughput can be thought of as time taken to execute a program or computation. For instance, imagine a program that is given hundreds of files containing integers and asked to sum up all the numbers. Since addition is commutative each file can be worked on in parallel. In a single-threaded environment, each file will be sequentially processed but in a concurrent system, several threads

can work in parallel on distinct files. Of course, there will be some overhead to manage the state including already processed files. However, such a program will complete the task much faster than a single thread. The performance difference will become more and more apparent as the number of input files increases. The throughput in this example can be defined as the number of files processed by the program in a minute. And latency can be defined as the total time taken to completely process all the files. As you observe in a multithreaded implementation throughput will go up and latency will go down. More work gets done in less amount of time. In general, the two have an inverse relationship.

← Back
I/O Bound vs CPU Bound

Next →
Critical Sections & Race Conditions

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question (https://discuss.educative.io/tag/throughput-vs-latency__the-basics__java-multithreading-for-senior-engineering-interviews)

educative

Critical Sections & Race Conditions

This section exhibits how incorrect synchronization in a critical section can lead to race conditions and buggy code. The concepts of critical section and race condition are explained in depth. Also included is an executable example of a race condition.

> A program is a set of instructions being executed, and multiple threads of a program can be executing different sections of the program code. However, caution should be exercised when threads of the same program attempt to execute the same portion of code as explained in the following paragraphs.

Critical Section

> Critical section is any piece of code that has the possibility of being executed concurrently by more than one thread of the application and exposes any shared data or resources used by the application for access.

Race Condition

> Race conditions happen when threads run through critical sections without thread synchronization. The threads ***"race"*** through the critical section to write or read shared resources and depending on the order in which threads finish the "race", the program output changes. In a race condition, threads access shared resources or program variables that might be worked on by other threads at the same time causing the application data to be inconsistent.

As an example consider a thread that tests for a state/condition, called a predicate, and then based on the condition takes subsequent action. This sequence is called **test-then-act**. The pitfall here is that the state can be mutated by the second thread just after the test by the first thread and before the first thread takes action based on the test. A different thread changes the predicate in between the **test and act**. In this case, action by the first thread is not justified since the predicate doesn't hold when the action is executed.

Consider the snippet below. We have two threads working on the same variable **`randInt`**. The modifier thread perpetually updates the value of **`randInt`** in a loop while the printer thread prints the value of **`randInt`** only if **`randInt`** is divisible by 5. If you let this program run, you'll notice some values get printed even though they aren't divisible by 5 demonstrating a thread unsafe verison of **test-then-act**.

Example Thread Race

The below program spawns two threads. One thread prints the value of a shared variable whenever the shared variable is divisible by 5. A race condition happens when the printer thread executes a *test-then-act* if clause, which checks if the shared variable is divisible by 5 but before the thread can print the variable out, its value is changed by the modifier thread. Some of the printed values aren't divisible by 5 which verifies the existence of a race condition in the code.

```
import java.util.*;

class Demonstration {

    public static void main(String args[]) throws InterruptedException {
            RaceCondition.runTest();
    }
}

```

```
10  class RaceCondition {
11
12      int randInt;
13      Random random = new Random(System.currentTimeMillis());
14
15      void printer() {
16
17          int i = 1000000;
18          while (i != 0) {
19              if (randInt % 5 == 0) {
20                  if (randInt % 5 != 0)
21                    System.out.println(randInt);
22              }
23              i--;
24          }
25      }
26
27      void modifier() {
28
```

Race Condition Example

Even though the if condition on **line 19** makes a check for a value which is divisible by 5 and only then prints **`randInt`**. It is just **after the if check and before the print statement** i.e. in-between **lines 19** and **21**, that the value of **`randInt`** is modified by the modifier thread! This is what constitutes a race condition.

For the impatient, the fix is presented below where we guard the read and write of the **`randInt`** variable using the **`RaceCondition`** object as the monitor. Don't fret if the solution doesn't make sense for now, it would, once we cover various topics in the lessons ahead.

```
39
40      public static void runTest() throws InterruptedException {
41
42
43          final RaceCondition rc = new RaceCondition();
44          Thread thread1 = new Thread(new Runnable() {
45
46              @Override
```

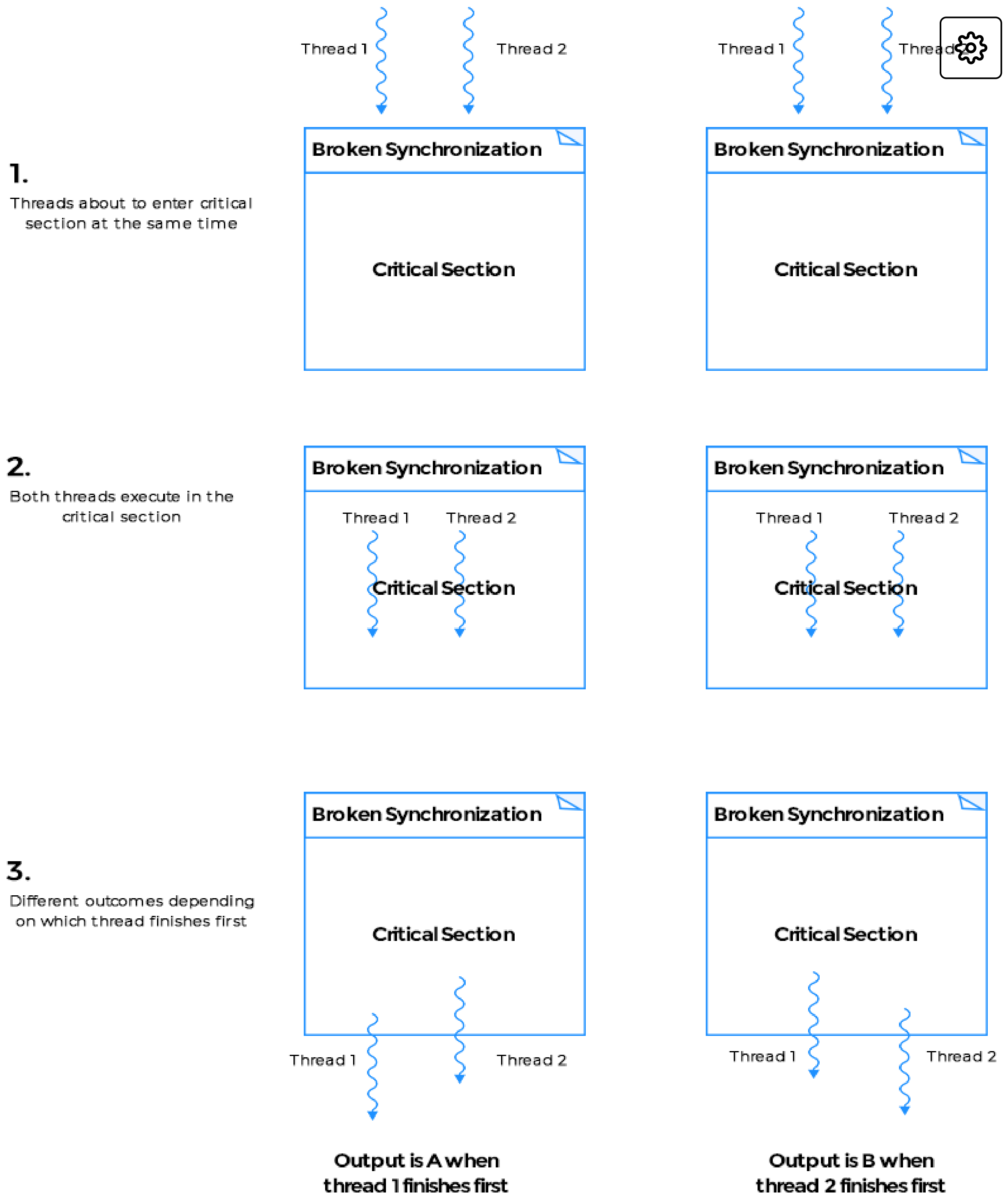
```java
            @Override
            public void run() {
                rc.printer();
            }
        });
        Thread thread2 = new Thread(new Runnable() {

            @Override
            public void run() {
                rc.modifier();
            }
        });


        thread1.start();
        thread2.start();

        thread1.join();
        thread2.join();
    }
}
```

Below is a pictorial representation of what a race condition, in general, looks like.



← Back

Throughput vs Latency

Next →

Deadlocks, Liveness & Reentrant Locks

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Deadlocks, Liveness & Reentrant Locks

We discuss important concurrency concepts deadlock, liveness, live-lock, starvation and reentrant locks in depth. Also included are executable code examples for illustrating these concepts.

> Logical follies committed in multithreaded code, while trying to avoid race conditions and guarding critical sections, can lead to a host of subtle and hard to find bugs and side-effects. Some of these incorrect usage patterns have their names and are discussed below.

DeadLock

> Deadlocks occur when two or more threads aren't able to make any progress because the resource required by the first thread is held by the second and the resource required by the second thread is held by the first.

Liveness

> Ability of a program or an application to execute in a timely manner is called liveness. If a program experiences a deadlock then it's not exhibiting liveness.

Live-Lock

A live-lock occurs when two threads continuously react in response to the actions by the other thread without making any real progress. The best analogy is to think of two persons trying to cross each other in a hallway. John moves to the left to let Arun pass, and Arun moves to his right to let John pass. Both block each other now. John sees he's blocking Arun again and moves to his right and Arun moves to his left seeing he's blocking John. They never cross each other and keep blocking each other. This scenario is an example of a livelock. A process seems to be running and not deadlocked but in reality, isn't making any progress.

Starvation

Other than a deadlock, an application thread can also experience starvation, when it never gets CPU time or access to shared resources. Other **greedy** threads continuously hog shared system resources not letting the starving thread make any progress.

Deadlock Example

```
void increment(){

  acquire MUTEX_A
  acquire MUTEX_B
    // do work here
  release MUTEX_B
  release MUTEX_A

}


void decrement(){

  acquire MUTEX_B
  acquire MUTEX_A
    // do work here
  release MUTEX_A
  release MUTEX_B

}
```

The above code can potentially result in a deadlock. Note that deadlock may not always happen, but for certain execution sequences, deadlock can occur. Consider the below execution sequence that ends up in a deadlock:

```
T1 enters function increment

T1 acquires MUTEX_A

T1 gets context switched by the operating system

T2 enters function decrement

T2 acquires MUTEX_B

both threads are blocked now
```

Thread **T2** can't make progress as it requires **MUTEX_A** which is being held by **T1**. Now when **T1** wakes up, it can't make progress as it requires **MUTEX_B** and that is being held up by **T2**. This is a classic text-book example of a deadlock.

You can come back to the examples presented below as they require an understanding of the `synchronized` keyword that we cover in later sections. Or you can just run the examples and observe the output for now to get a high-level overview of the concepts we discussed in this lesson.

If you run the code snippet below, you'll see that the statements for acquiring locks: **lock1** and **lock2** print out but there's no progress after that and the execution times out. In this scenario, the deadlock occurs because the locks are being acquired in a nested fashion.

```java
class Demonstration {

    public static void main(String args[]) {
        Deadlock deadlock = new Deadlock();
        try {
            deadlock.runTest();
        } catch (InterruptedException ie) {
        }
    }
}

class Deadlock {

    private int counter = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    Runnable incrementer = new Runnable() {

        @Override
        public void run() {
            try {
                for (int i = 0; i < 100; i++) {
                    incrementCounter();
                    System.out.println("Incrementing " + i);
                }
            } catch (InterruptedException ie) {
            }
        }
    };

    Runnable decrementer = new Runnable() {

        @Override
        public void run() {
            try {
                for (int i = 0; i < 100; i++) {
                    decrementCounter();
                    System.out.println("Decrementing " + i);
                }
            } catch (InterruptedException ie) {
            }

        }
    };

    public void runTest() throws InterruptedException {

        Thread thread1 = new Thread(incrementer);
        Thread thread2 = new Thread(decrementer);

        thread1.start();
        // sleep to make sure thread 1 gets a chance to acquire lock1
        Thread.sleep(100);
        thread2.start();

        thread1.join();
        thread2.join();

        System.out.println("Done : " + counter);
    }

    void incrementCounter() throws InterruptedException {
        synchronized (lock1) {
            System.out.println("Acquired lock1");
            Thread.sleep(100);

            synchronized (lock2) {
                counter++;
            }
        }
    }

    void decrementCounter() throws InterruptedException {
        synchronized (lock2) {
            System.out.println("Acquired lock2");

            Thread.sleep(100);
            synchronized (lock1) {
                counter--;
            }
        }
    }
}
```

Example of a Deadlock

Reentrant Lock

Re-entrant locks allow for re-locking or re-entering of a synchronization lock. This can be best explained with an example. Consider the NonReentrant class below.

Take a minute to read the code and assure yourself that any object of this class if locked twice in succession would result in a deadlock. The same thread gets blocked on itself, and the program is unable to make any further progress. If you click run, the execution would time-out.

If a synchronization primitive doesn't allow reacquisition of itself by a thread that has already acquired it, then such a thread would block as soon as it attempts to reacquire the primitive a second time.

```
class Demonstration {

    public static void main(String args[]) throws Exception {
        NonReentrantLock nreLock = new NonReentrantLock();

        // First locking would be successful
        nreLock.lock();
        System.out.println("Acquired first lock");

        // Second locking results in a self deadlock
        System.out.println("Trying to acquire second lock");
        nreLock.lock();
        System.out.println("Acquired second lock");
    }
}

class NonReentrantLock {

    boolean isLocked;

    public NonReentrantLock() {
        isLocked = false;
    }

    public synchronized void lock() throws InterruptedException {

        while (isLocked) {
            wait();
        }
        isLocked = true;
    }

    public synchronized void unlock() {
        isLocked = false;
        notify();
    }
}
```

Example of Deadlock with Non-Reentrant Lock

The statement **`"Acquired second lock"`** is never printed

Back

Critical Sections & Race Conditions

Next

Mutex vs Semaphore

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question (https://discuss.educative.io/tag/deadlocks-liveness--reentrant-locks__the-basics__java-multithreading-for-senior-engineering-interviews)

Mutex vs Monitor

Learn what a monitor is and how it is different than a mutex. Monitors are advanced concurrency constructs and specific to languages frameworks.

Continuing our discussion from the previous section on locking and signaling mechanisms, we'll now pore over an advanced concept the **monitor**. It is exposed as a concurrency construct by some programming language frameworks including Java.

When Mutual Exclusion isn't Enough

Concisely, a monitor is a mutex and then some. Monitors are generally language level constructs whereas mutex and semaphore are lower-level or OS provided constructs.

To understand monitors, let's first see the problem they solve. Usually, in multi-threaded applications, a thread needs to wait for some program predicate to be true before it can proceed forward. Think about a producer/consumer application. If the producer hasn't produced anything the consumer can't consume anything, so the consumer must ***wait on*** a predicate that lets the consumer know that something has indeed been produced. What could be a crude way of accomplishing this? The consumer could repeatedly check in a loop for the predicate to be set to true. The pattern would resemble the pseudocode below:

```
void busyWaitFunction() {
    // acquire mutex
    while (predicate is false) {
```

```
                // release mutex
                // acquire mutex
            }
            // do something useful
            // release mutex
        }
```

Within the while loop we'll first release the mutex giving other threads a chance to acquire it and set the loop predicate to true. And before we check the loop predicate again, we make sure we have acquired the mutex again. This works but is an example of ***"spin waiting"*** which wastes a lot of CPU cycles. Next, let's see how condition variables solve the spin-waiting issue.

Condition Variables

Mutex provides mutual exclusion, however, at times mutual exclusion is not enough. We want to test for a predicate with a mutually exclusive lock so that no other thread can change the predicate when we test for it but if we find the predicate to be false, we'd want to wait on a condition variable till the predicate's value is changed. This thus is the solution to spin waiting.

Conceptually, each condition variable exposes two methods **`wait()`** and **`signal()`**. The **`wait()`** method when called on the condition variable will cause the associated mutex to be atomically released, and the calling thread would be placed in a **wait queue**. There could already be other threads in the **wait queue** that previously invoked **`wait()`** on the condition variable. Since the mutex is now released, it gives other threads a chance to change the predicate that will eventually let the thread that was just placed in the wait queue to make progress. As an example, say we have a consumer thread that checks for the size of the buffer, finds it empty and invokes **`wait()`** on a condition variable. The predicate in this example would be ***the size of the buffer***.

Now imagine a producer places an item in the buffer. The predicate, the size of the buffer, just changed and the producer wants to let the consumer threads know that there is an item to be consumed. This producer thread would then invoke **`signal()`** on the condition variable. The **`signal()`** method when called on a condition variable causes one of the threads that has been placed in the **wait queue** to get ready for execution. Note we didn't say the woken up thread starts executing, it just gets ready - and that could mean being placed in the ready queue. It is only **after the producer thread which calls the `signal()` method has released the associated mutex that the thread in the ready queue starts executing.** The thread in the ready queue must wait to acquire the mutex associated with the condition variable before it can start executing.

Lets see how this all translates into code.

```
void efficientWaitingFunction() {
    mutex.acquire()
    while (predicate == false) {
      condVar.wait()
    }
    // Do something useful
    mutex.release()
}

void changePredicate() {
    mutex.acquire()
    set predicate = true
    condVar.signal()
    mutex.release()
}
```

Let's dry run the above code. Say **thread A** executes `efficientWaitingFunction()` first and finds the loop predicate is false and enters the loop. Next **thread A** executes the statement `condVar.wait()` and is be placed in a wait queue. At the same time **thread A** gives up the mutex. Now **thread B** comes along and executes `changePredicate()` method. Since the mutex was given up by **thread A**, **thread B** is be able to acquire it and set the predicate to true. Next it signals the condition variable `condVar.signal()`. This step places **thread A** into the ready queue but **thread A** doesn't start executing until **thread B** has released the mutex.

Note that the order of signaling the condition variable and releasing the mutex can be interchanged, but generally, the preference is to signal first and then release the mutex. However, the ordering might have ramifications on thread scheduling depending on the threading implementation.

Why the *while* Loop

The wary reader would have noticed us using a while loop to test for the predicate. After all, the pseudocode could have been written as follows

```
void efficientWaitingFunction() {
    mutex.acquire()
    if (predicate == false) {
      condVar.wait()
    }
    // Do something useful
    mutex.release()
}
```

If the snippet is re-written in the above manner using an `if` clause instead of a `while` then,, we need a guarantee that once the variable `condVar` is signaled, the predicate can't be changed by any other thread and that the signaled thread becomes the owner of the monitor. This may not be true. For one, a different thread could get scheduled and change the predicate back to false before the signaled thread gets a chance to execute, therefore the signaled thread must check the predicate again, once it acquires the monitor. Secondly, use of the loop is necessitated by design choices of monitors that we'll explore in the next section. Last but not the least, on POSIX systems, **spurious or fake wakeups** are possible (also discussed in later chapters) even though the condition variable has not been signaled and the predicate hasn't changed. **The idiomatic and correct usage of a monitor dictates that *the predicate always be tested for in a while loop*.**

Monitor Explained

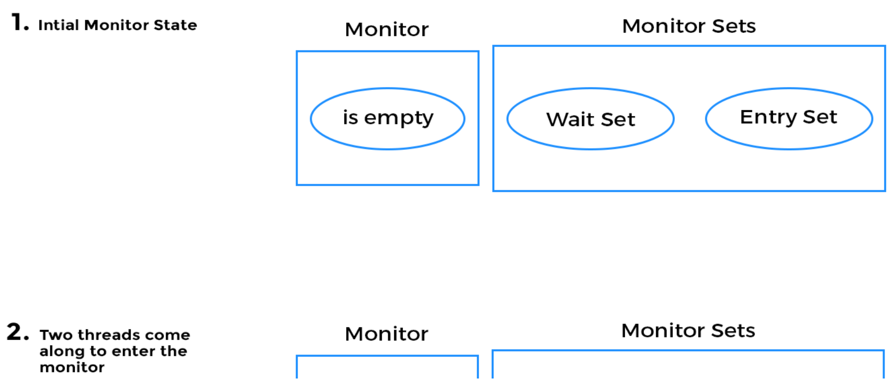
After the above discussion, we can now realize that **a monitor is made up of a mutex and one or more condition variables**. A single monitor can have multiple condition variables but not vice versa. Theoretically, another way to think about a monitor is to consider it as an entity having two queues or sets where threads can be placed. One is the **entry set** and the other is the **wait set**. When a thread A ***enters*** a monitor it is placed into the **entry set**. If no other thread ***owns*** the monitor, which is equivalent of saying no thread is actively executing within the monitor section, then thread A will ***acquire*** the monitor and is said to own it too. Thread A will continue to execute within the monitor section till it ***exits*** the monitor or calls `wait()` on an associated condition variable and be placed into the wait set. While thread A **owns** the monitor no other thread will be able to execute any of the critical sections protected by the monitor. New threads requesting ownership of the monitor get placed into the **entry set**.

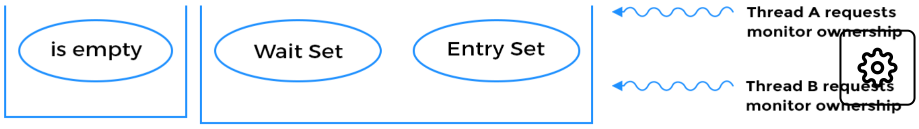
Continuing with our hypothetical example, say another thread B comes along and gets placed in the **entry set**, while thread A sits in the **wait set**. Since no other thread owns the monitor, thread B successfully acquires the monitor and continues execution. If thread B exits the monitor section without calling `notify()` on the condition variable, then thread A will remain waiting in the **wait set**. Thread B can also invoke `wait()` and be placed in the **wait set** along with thread A. This then would require a third thread to come along and call `notify()` on the condition variable on which both threads A and B are waiting. Note that only a single thread will be able to **own** the monitor at any given point in time and will have exclusive access to data structures or critical sections protected by the monitor.

Practically, in Java each object is a monitor and implicitly has a lock and is a condition variable too. You can think of a monitor as a ***mutex with a wait set***. Monitors allow threads to exercise **mutual exclusion** as well as **cooperation** by allowing them to wait and signal on conditions.

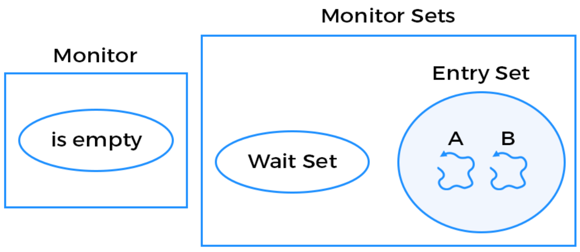
Initial Monitor State

A pictorial representation of the above simulation appears below:

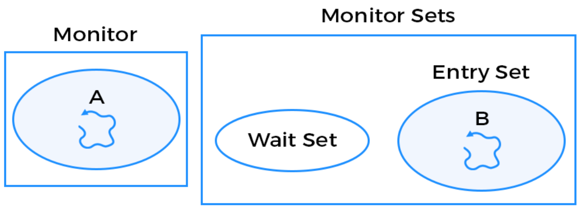



is empty
Wait Set
Entry Set
Thread A requests monitor ownership
Thread B requests monitor ownership

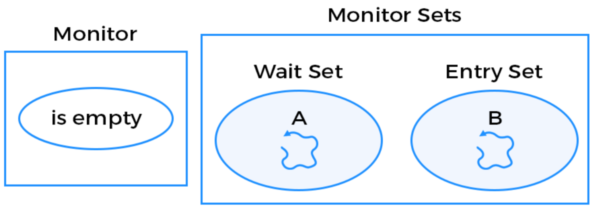
3. Thread A and B get placed in the entry set



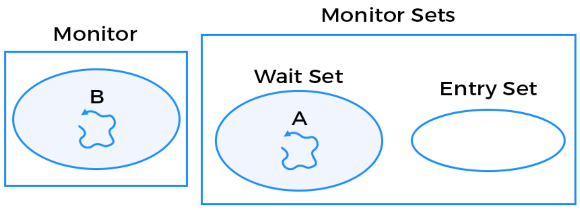
4. Thread A enters the monitor and starts execution



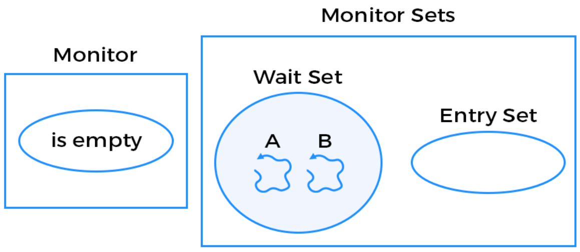
5. Thread A execution Wait() and gets placed in wait set



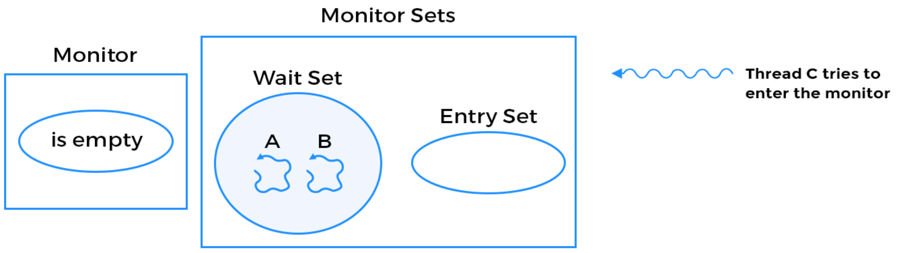
6. Thread B now able to enter the monitor



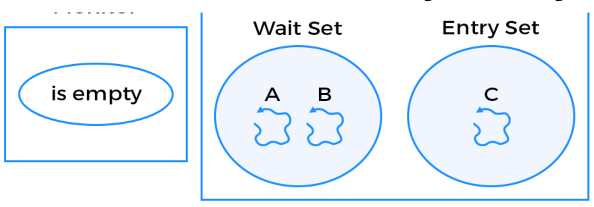
7. Thread B also invokes wait() and gets placed in the wait set



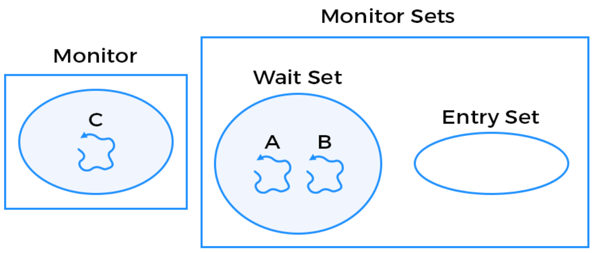
8. Thread C comes along to enter the monitor



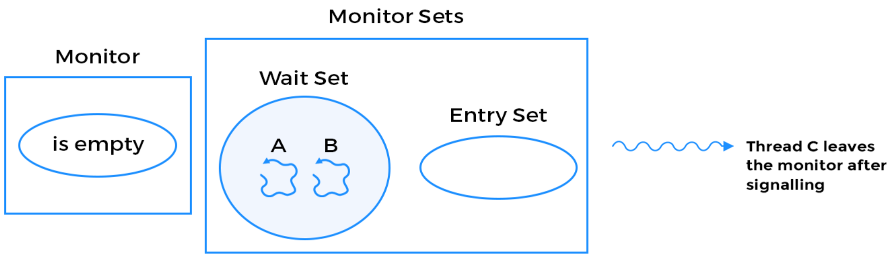
9. Thread C is placed in the entry set



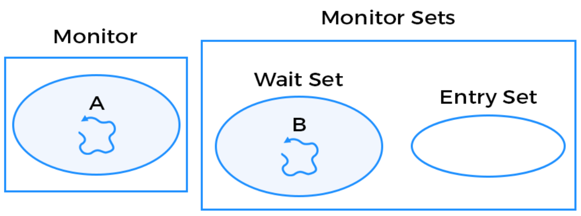
10. Thread C enters the monitor



11. Thread C exits monitor after signalling



12. Thread A resumes ownership of the monitor



← Back
Mutex vs Semaphore

Next →
Java's Monitor & Hoare vs Mesa Monit...

✓ Completed

53% completed, meet the criteria and claim your course certificate!

Claim Certificate

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/mutex-vs-monitor__the-basics__java-multithreading-for-senior-engineering-interviews)

Mutex vs Semaphore

The concept of and the difference between a mutex and a semaphore will draw befuddled expressions on most developers' faces. We discuss the differences between the two most fundamental concurrency constructs offered by almost all language frameworks. Difference between a mutex and a semaphore makes a pet interview question for senior engineering positions!

> Having laid the foundation of concurrent programming concepts and their associated issues, we'll now discuss the all-important mechanisms of locking and signaling in multi-threaded applications and the differences amongst these constructs.

Mutex

> Mutex as the name hints implies ***mutual exclusion***. A mutex is used to guard shared data such as a linked-list, an array or any primitive type. A mutex allows only a single thread to access a resource or critical section.
>
> Once a thread acquires a mutex, all other threads attempting to acquire the same mutex are blocked until the first thread releases the mutex. Once released, most implementations arbitrarily chose one of the waiting threads to acquire the mutex and make progress.

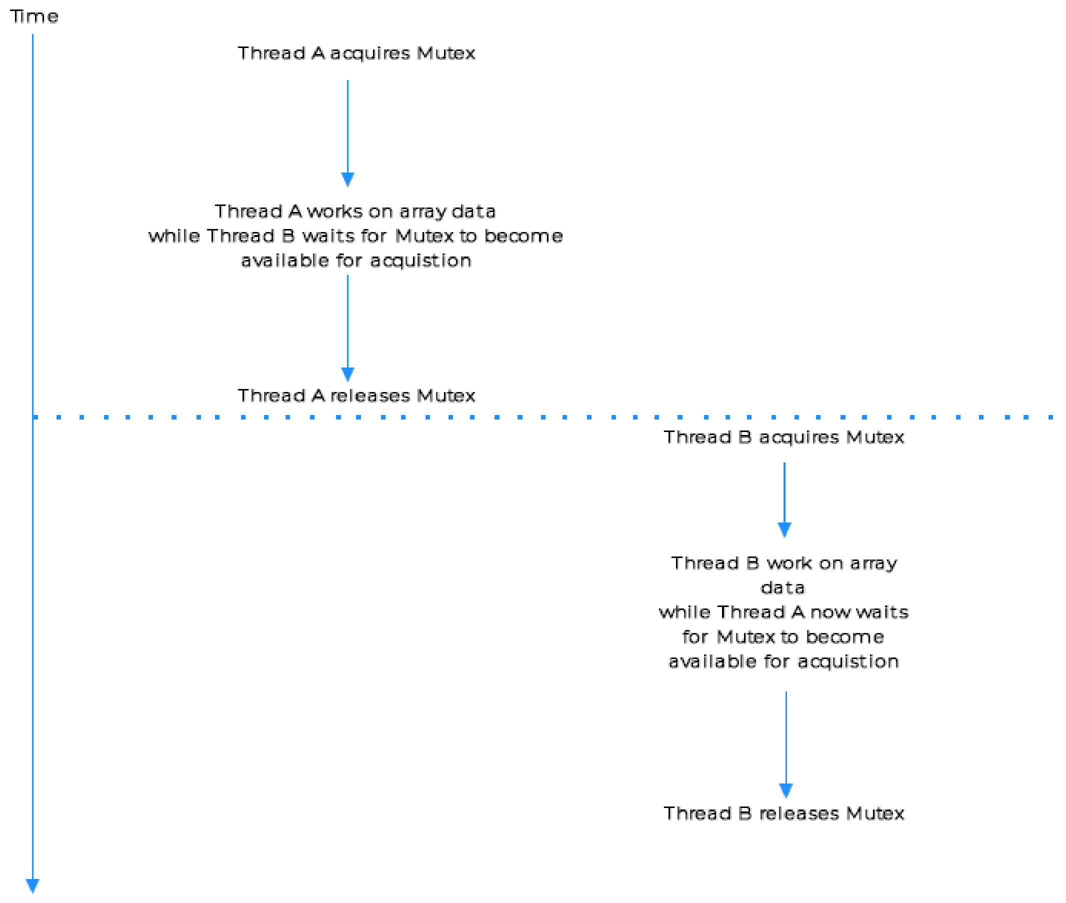
Semaphore

Semaphore, on the other hand, is used for limiting access to a collection of resources. Think of semaphore as having a limited number of permits to give out. If a semaphore has given out all the permits it has, then any new thread that comes along requesting for a permit will be blocked, till an earlier thread with a permit returns it to the semaphore. A typical example would be a pool of database connections that can be handed out to requesting threads. Say there are ten available connections but 50 requesting threads. In such a scenario, a semaphore can only give out ten permits or connections at any given point in time.

A semaphore with a single permit is called a **binary semaphore** and is often thought of as an equivalent of a mutex, which isn't completely correct as we'll shortly explain. Semaphores can also be used for signaling among threads. This is an important distinction as it allows threads to cooperatively work towards completing a task. A mutex, on the other hand, is strictly limited to serializing access to shared state among competing threads.

Mutex Example

The following illustration shows how two threads acquire and release a mutex one after the other to gain access to shared data. Mutex guarantees the shared state isn't corrupted when competing threads work on it.


Time
Thread A acquires Mutex
Thread A works on array data while Thread B waits for Mutex to become available for acquistion
Thread A releases Mutex
Thread B acquires Mutex
Thread B work on array data while Thread A now waits for Mutex to become available for acquistion
Thread B releases Mutex

When a Semaphore Masquerades as a Mutex?

A semaphore can potentially act as a mutex if the permits it can give out is set to 1. However, the most important difference between the two is that in case of a mutex ***the same thread must call acquire and subsequent release on the mutex*** whereas in case of a binary sempahore, ***different threads can call acquire and release on the semaphore***. The pthreads library documentation states this in the **`pthread_mutex_unlock()`** method's description.

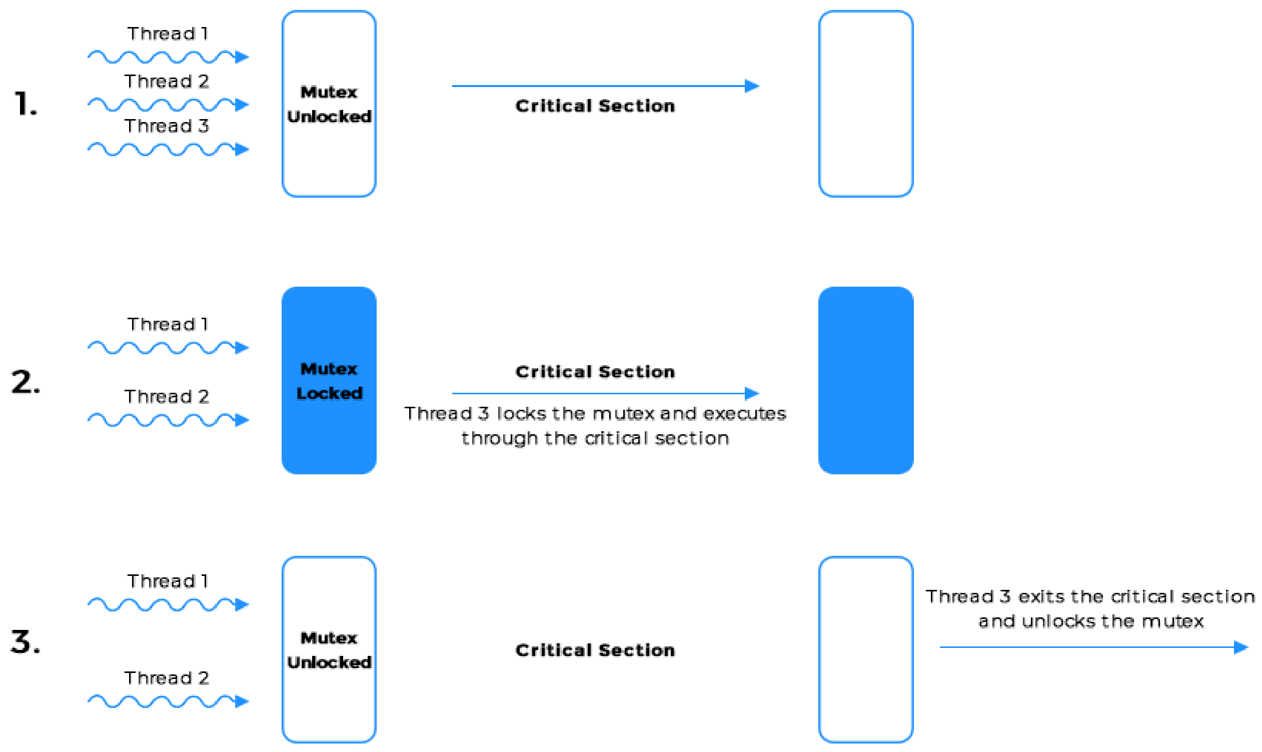
```
If a thread attempts to unlock a mutex that it has not locked or a mutex whi
```

This leads us to the concept of **ownership**. **A mutex is owned by the thread acquiring it till the point the owning-thread releases it, whereas for a semaphore there's no notion of ownership.**

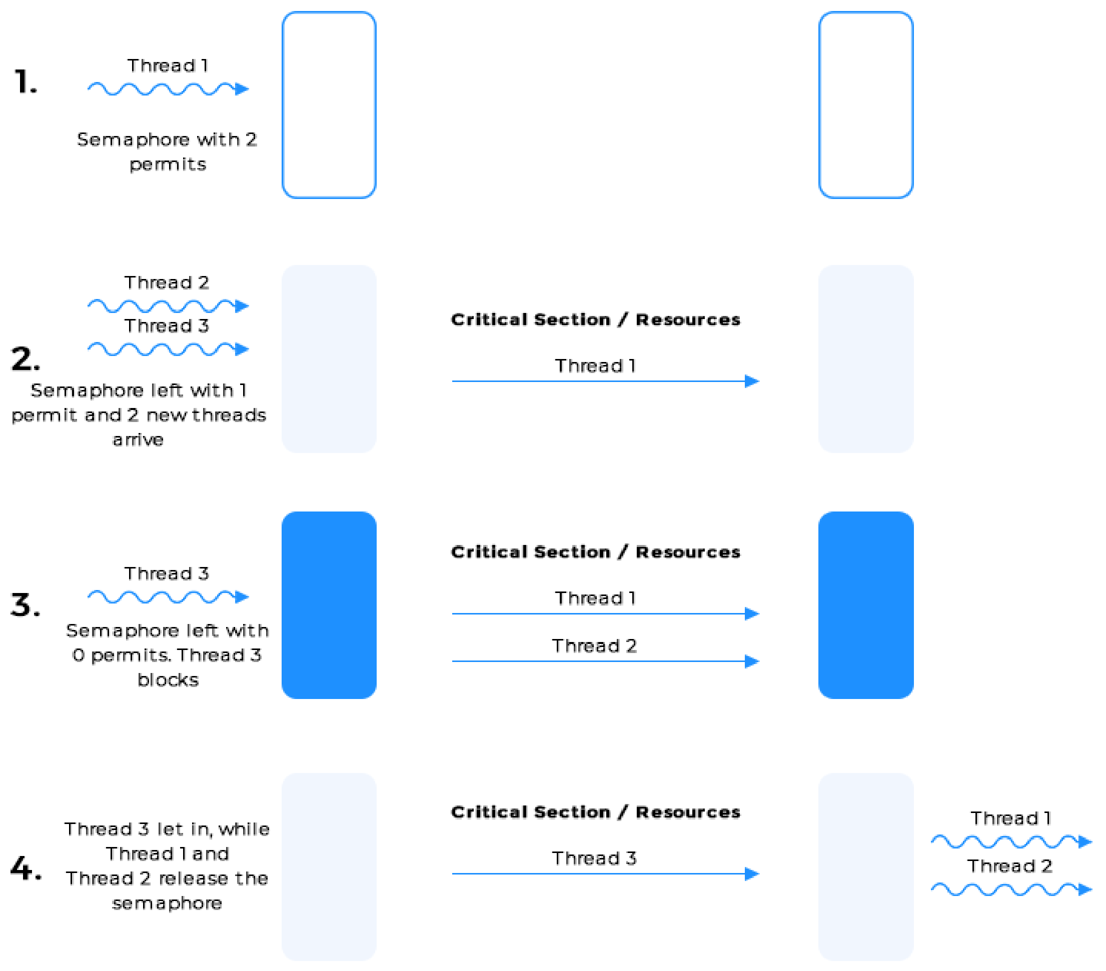
Semaphore for Signaling

Another distinction between a semaphore and a mutex is that semaphores can be used for signaling amongst threads, for example in case of the classical **producer/consumer** problem the producer thread can signal the consumer thread by incrementing the semaphore count to indicate to the consumer thread to consume the freshly produced item. A mutex in contrast only guards access to shared data among competing threads by forcing threads to serialize their access to critical sections and shared data-structures.

Below is a pictorial representation of how a mutex works.



Below is a depiction of how a semaphore works. The semaphore initially has two permits and allows at most two threads to enter the critical section or access protected resources


1.
Thread 1
Semaphore with 2 permits
2.
Thread 2
Thread 3
Semaphore left with 1 permit and 2 new threads arrive
Critical Section / Resources
Thread 1
3.
Thread 3
Semaphore left with 0 permits. Thread 3 blocks
Critical Section / Resources
Thread 1
Thread 2
4.
Thread 3 let in, while Thread 1 and Thread 2 release the semaphore
Critical Section / Resources
Thread 3
Thread 1
Thread 2

Summary

1. Mutex implies mutual exclusion and is used to serialize access to critical sections whereas semaphore can potentially be used as a mutex but it can also be used for cooperation and signaling amongst threads. Semaphore also solves the issue of **missed signals**.

2. Mutex is **owned** by a thread, whereas a semaphore has no concept of ownership.

3. Mutex if locked, must necessarily be unlocked by the same thread. A semaphore can be acted upon by different threads. This is true even if the semaphore has a permit of one

4. Think of semaphore analogous to a car rental service such as Hertz. Each outlet has a certain number of cars, it can rent out to customers. It can rent several cars to several customers at the same time but if all the cars are rented out then any new customers need to be put on a waitlist till one of the rented cars is returned. In contrast, think of a mutex like a lone runway on a remote airport. Only a single jet can land or take-off from the runway at a given point in time. No other jet can use the runway simultaneously with the first aircraft.

← Back
Deadlocks, Liveness & Reentrant Locks

Next →
Mutex vs Monitor

✓ Completed

Semaphore vs Monitor

This lesson discusses the differences between a monitor and a semaphore.

> Monitor, mutex, and semaphores can be confusing concepts initially. A monitor is made up of a mutex and a condition variable. One can think of a mutex as a subset of a monitor. Differences between a monitor and semaphore are discussed below.

The Difference

- A monitor and a semaphore are interchangeable and theoretically, one can be constructed out of the other or one can be reduced to the other. However, monitors take care of atomically acquiring the necessary locks whereas, with semaphores, the onus of appropriately acquiring and releasing locks is on the developer, which can be error-prone.

- Semaphores are lightweight when compared to monitors, which are bloated. However, the tendency to misuse semaphores is far greater than monitors. When using a semaphore and mutex pair as an alternative to a monitor, it is easy to lock the wrong mutex or just forget to lock altogether. Even though both constructs can be used to solve the same problem, monitors provide a pre-packaged solution with less dependency on a developer's skill to get the locking right.

- Java monitors enforce correct locking by throwing the **`IllegalMonitorState`** exception object when methods on a condition variable are invoked without first acquiring the associated lock. The exception is in a way saying that either the object's lock/mutex was not acquired at all or that an incorrect lock was acquired.

- A semaphore can allow several threads access to a given resource or critical section, however, only a single thread at any point in time can **own** the monitor and access associated resource.

- Semaphores can be used to address the issue of **missed signals**, however with monitors additional state, called the predicate, needs to be maintained apart from the condition variable and the mutex which make up the monitor, to solve the issue of missed signals.

← Back
Java's Monitor & Hoare vs Mesa Monit...

Next →
Amdahl's Law

✓ Completed

Java's Monitor & Hoare vs Mesa Monitors

Continues the discussion of the differences between a mutex and a monitor and also looks at Java's implementation of the monitor.

We discussed the abstract concept of a monitor in the previous section and now let's see the working of a concrete implementation of it in Java.

Java's Monitor

In Java every object is a condition variable and has an associated lock that is hidden from the developer. Each java object exposes **`wait()`** and **`notify()`** methods.

Before we execute **`wait()`** on a java object we need to lock its hidden mutex. That is done implicitly through the **synchronized** keyword. If you attempt to call **`wait()`** or **`notify()`** outside of a synchronized block, an **`IllegalMonitorStateException`** would occur. It's Java reminding the developer that the mutex wasn't acquired before wait on the condition variable was invoked. **`wait()`** and **`notify()`** can only be called on an object once the calling thread becomes the ***owner*** of the monitor. The ownership of the monitor can be achieved in the following ways:

- the method the thread is executing has synchronized in its signature
- the thread is executing a block that is synchronized on the object on which wait or notify will be called

- in case of a class, the thread is executing a static method which is synchronized.

Bad Synchronization Example 1

In the below snippet, **`wait()`** is being called outside of a synchronized block, i.e. without implicitly locking the associated mutex. Running the code results in **IllegalMonitorStateException**

```
class BadSynchronization {

    public static void main(String args[]) throws InterruptedException
        Object dummyObject = new Object();

        // Attempting to call wait() on the object
        // outside of a synchronized block.
        dummyObject.wait();
    }
}
```

Illegal Monitor Exception

Bad Synchronization Example 2

Here's another example where we try to call **`notify()`** on an object in a synchronized block which is synchronized on a different object. Running the code will again result in an **IllegalMonitorStateException**

```
class BadSynchronization {

    public static void main(String args[]) {
        Object dummyObject = new Object();
        Object lock = new Object();

        synchronized (lock) {
            lock.notify();

            // Attempting to call notify() on the object
            // in synchronized block of another object
            dummyObject.notify();
        }
    }
}
```

Hoare vs Mesa Monitors

So far we have determined that the idiomatic usage of a monitor requires using a while loop as follows. Let's see how the design of monitors affects this recommendation.

```
while( condition == false ) {
    condVar.wait();
}
```

Once the asleep thread is signaled and wakes up, you may ask why does it need to check for the condition being false again, the signaling thread must have just set the condition to true?

In **Mesa monitors** - Mesa being a language developed by Xerox researchers in the 1970s - it is possible that the time gap between thread B calls **`notify()`** and releases its mutex **and** the instant at which the asleep thread A, wakes up and reacquires the mutex, ***the predicate is changed back to false by another thread different***

than the signaler and the awoken threads! The woken up thread competes with other threads to acquire the mutex once the signaling thread B **empties** the monitor. On signaling, thread B doesn't give up the monitor just yet; rather it continues to own the monitor until it exits the monitor section.

In contrast, **Hoare monitors** - Hoare being one of the original inventor of monitors - the signaling thread B ***yields*** the monitor to the woken up thread A and thread A ***enters*** the monitor, while thread B sits out. This guarantees that the predicate will not have changed and instead of checking for the predicate in a while loop an if-clause would suffice. The woken-up/released thread A immediately starts execution when the signaling thread B signals that the predicate has changed. No other thread gets a chance to change the predicate since no other thread gets to enter the monitor.

Java, in particular, subscribes to Mesa monitor semantics and the developer is always expected to check for condition/predicate in a while loop. Mesa monitors are more efficient than Hoare monitors.

← Back
Mutex vs Monitor

Next →
Semaphore vs Monitor

✓ Completed

educative

Amdahl's Law

Blindly adding threads to speed up program execution may not always be a good idea. Find out what Amdahl's Law says about parallelizing a program

Definition

No text on concurrency is complete without mentioning the ***Amdahl's Law***. The law specifies the cap on the maximum speedup that can be achieved when parallelizing the execution of a program.

If you have a poultry farm where a hundred hens lay eggs each day, then no matter how many people you hire to process the laid eggs, you still need to wait an entire day for the 100 eggs to be laid. Increasing the number of workers on the farm can't shorten the time it takes for a hen to lay an egg. Similarly, software programs consist of parts which can't be sped up even if the number of processors is increased. These parts of the program must execute serially and aren't amenable to parallelism.

Amdahl's law describes the theoretical speedup a program can achieve at best by using additional computing resources. We'll skip the mathematical derivation and go straight to the simplified equation expressing Amdahl's law:

$$S(n) = \frac{1}{(1-P) + \frac{P}{n}}$$

- ***S(n)*** is the speed-up achieved by using ***n*** cores or threads.

- ***P*** is the fraction of the program that is parallelizable
- ***(1 - P)*** is the fraction of the program that must be executed serially.

Example

Say our program has a parallelizable portion of **P** = **90%** = **0.9**. Now let's see how the speed-up occurs as we increase the number of processes

- ***n** = **1 processor***

$$S(1) = \frac{1}{(1-P)+\frac{P}{1}} = \frac{1}{1-P+P} = 1$$

- ***n** = **2 processors***

$$S(2) = \frac{1}{(1-0.9)+\frac{0.9}{2}} = \frac{1}{0.1+0.45} = 1.81$$

- ***n** = **5 processors***

$$S(5) = \frac{1}{(1-0.9)+\frac{0.9}{5}} = \frac{1}{0.1+0.18} = 3.57$$

- ***n** = **10 processors***

$$S(10) = \frac{1}{(1-0.9)+\frac{0.9}{10}} = \frac{1}{0.1+0.09} = 5.26$$

- ***n** = **100 processors***

$$S(100) = \frac{1}{(1-0.9)+\frac{0.9}{100}} = \frac{1}{0.1+0.009} = 9.17$$

- ***n** = **1000 processors***

$$S(1000) = \frac{1}{(1-0.9)+\frac{0.9}{1000}} = \frac{1}{0.1+0.0009} = 9.91$$

- ***n** = **infinite processors***

$$S(\infty) = \frac{1}{(1-0.9)+\frac{0.9}{\infty}} = \frac{1}{0.1+0} = 10$$

The speed-up steadily increases as we increase the number of processors or threads. However, as you can see the theoretical maximum speed-up for our program with 10% serial execution will be 10. We can't speed-up our program execution more than 10 times compared to when we run the same program on a single CPU or thread. To achieve greater speed-ups than 10 we must optimize or parallelize the serially executed portion of the code.

Another important aspect to realize is that when we speed-up our program execution by roughly 5 times, we do so by employing 10 processors. The utilization of these 10 processors, in turn, decreases by roughly 50% because now the 10 processors remain idle for the rest of the time that a single processor would have been busy. Utilization is defined as the ***speedup divided by the number of processors***.

As an example say the program runs in 10 minutes using a single core. We assumed the parallelizable portion of the program is 90%, which implies 1 minute of the program time must execute serially. The speedup we can achieve with 10 processors is roughly 5 times which comes out to be 2 minutes of total program execution time. Out of those 2 minutes, 1 minute is of mandatory serial execution and the rest can all be parallelized. This implies that 9 of the processors will complete 90% of the non-serial work in 1 minute while 1 processor remains idle and then one out of the 10 processors, will execute the serial portion for another minute. The rest of the 9 processors are idle for that 1 minute during which the serial execution takes place. In effect, the combined utilization of the ten processors drops by 50%.

As ***N*** approaches infinity, the Amdahl's law takes the following form:

$$S(n) = \frac{1}{(1-P)} = \frac{1}{fraction\ of\ program\ serially\ executed}$$

One should take the calculations using Amdahl's law with a grain of salt. If the formula spits out a speed-up of 5x it doesn't imply that in reality one would observe a similar speed-up. There are other factors such as the memory architecture, cache misses, network and disk I/O etc that can affect the execution time of a program and the actual speed-up might be less than the calculated one.

The Amdahl's law works on a problem of fixed size. However as computing resources are improved, algorithms run on larger and even larger datasets. As the dataset size grows, the parallelizable portion of the program grows faster than the serial portion and a more realistic assessment of performance is given by **Gustafson's law**, which we won't discuss here as it is beyond the scope of this text.

← Back
Semaphore vs Monitor

Next →
Moore's Law

✓ Completed

educative

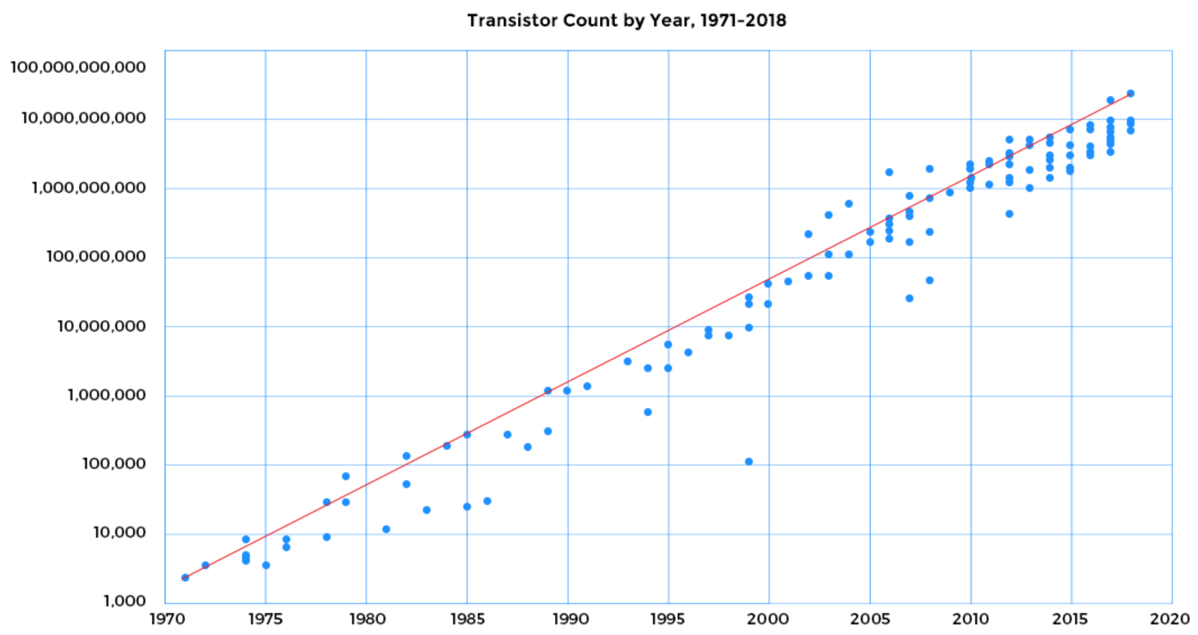
Moore's Law

Discusses impact of Moore's law on concurrency.

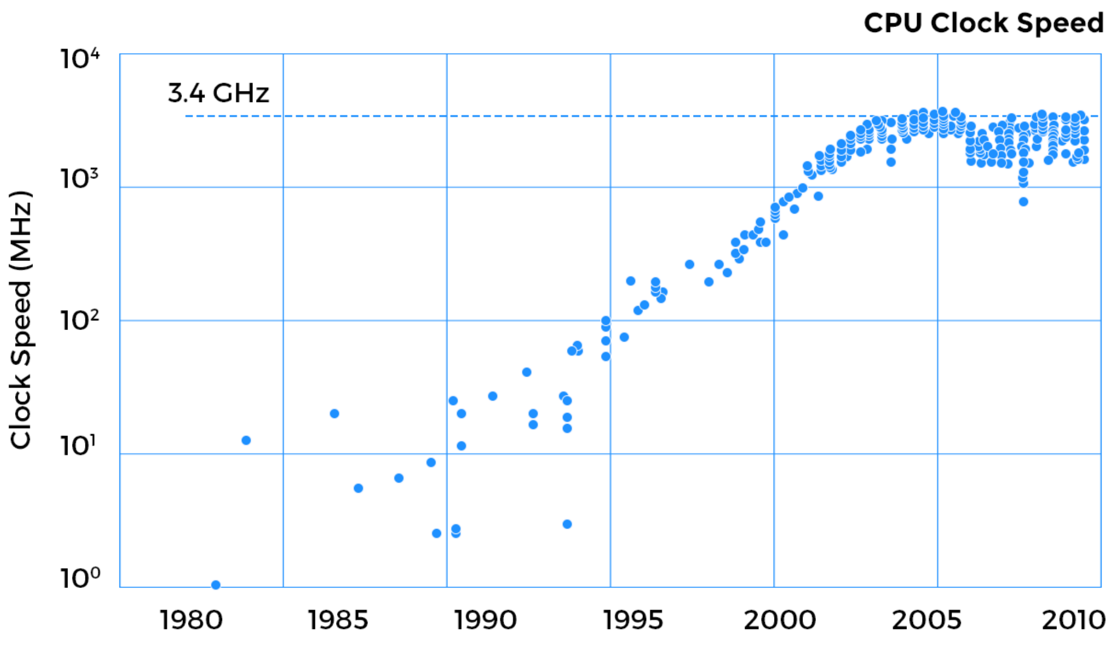
Moore's Law

In this lesson, we'll cover a high-level overview of Moore's law to present a final motivation to the reader for writing multi-threaded applications and realize that concurrency is inevitable in the future of software.

Gordon Moore (https://en.wikipedia.org/wiki/Gordon_Moore), co-founder of Intel, observed the number of transistors that can be packed into a given unit of space doubles about every two years and in turn the processing power of computers doubles and the cost halves. Moore's law is more of an observation than a law grounded in formal scientific research. It states that **the number of transistors per square inch on a chip will double every two years**. This exponential growth has been going on since the 70’s and is only now starting to slow down. The following graph shows the growth of the transistor count.



Initially, the clock speeds of processors also doubled along with the transistor count. This is because as transistors get smaller, their frequency increases and propagation delays decrease because now the transistors are packed closer together. However, the promise of exponential growth by Moore's law came to an end more than a decade ago with respect to clock speeds. The increase in clock speeds of processors has slowed down much faster than the increase in number of transistors that can be placed on a microchip. If we plot clock speeds we find that the linear exponential growth stopped after 2003 and the trend line flattened out. The clock speed (proportional to difference between supply voltage and threshold voltage) cannot increase because the supply voltage is already down to an extent where it cannot be decreased to get dramatic gains in clock speed. **In 10 years from 2000 to 2009, clock speed just increased from 1.3 GHz to 2.8 GHz merely doubling in a decade rather than increasing 32 times as expected by Moore's law.** The following plot shows the clock speeds flattening out towards 2010.



Since processors aren't getting faster as quickly as they use to, we need alternative measures to achieve performance gains. One of the ways to do that is to use multicore processors. Introduced in the early 2000s, multicore processors have more than one CPU on the same machine. To exploit this processing power, programs must be written as multi-threaded applications. A single-threaded application running on an octa-core processor will only use 1/8th of the total throughput of that machine, which is unacceptable in most scenarios.

Another analogy is to think of a bullock cart being pulled by an ox. We can breed the ox to be stronger and more powerful to pull more load but eventually there's a limit to how strong the ox can get. To pull more load, an easier solution is to attach several oxen to the bullock cart. The computing industry is also going in the direction of this analogy.

← Back
Amdahl's Law

Next →
Thready Safety & Synchronized

✓ Completed