Q1:

1. Blocking Algorithms: Blocking algorithms block the thread until the requested action can be performed.

2. Non-Blocking Algorithms: Non-blocking algorithms notify the thread requesting the action that the action cannot be performed.

3. Intrinsic Lock is an implicit internal entity associated with each instance of objects. And explicit Lock is explicitly requested for a record or table

4. Intrinsic locks use blocking structure. It enforces mutual exclusion and ensures that the actions of one thread executing a synchronized block are visible to other threads that later execute a synchronized block protected by the same lock.

5. Compare and swap 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.

6. compareAndSet is an api, it calls compareAndSwapInt of the unsafe class.

7. AppDynamics detects thread contention based on the thread state of the instrumented application. It identifies these block or waiting states in the JVM:

Acquiring a lock (MONITOR\_WAIT)

Waiting for a condition (CONDOR\_WAIT)

Sleeping (OBJECT\_WAIT)

A blocking I/O operation

8. By nonblocking algorithms. Uses the CPU's CAS instruction and JNI to complete Java's non-double algorithm

9. It does not reallt fail. And it does not cause for all run time operations to fail. Because when multiple threads try to use CAS to update the same variable at the same time, only one of the threads can update the value of the variable, while the other threads fail. The failed threads do not really fail, but are notified of the failure in the competition. And they can try again.

10. True:

1) They are all locked in synchronization;

2) Both are reentrant locks;

3) Blocking synchronization; that is, if a thread acquires an object lock and enters a synchronization block, other threads accessing the synchronization block must block outside the synchronization block and wait.

11. No. Some things cannot be done with built-in locks. Explicit locks can solve some flexible needs. ReentrantLock can be timing, timing, conditional queue, conditional predicate.

12. Because it locks synchronous resources and multiple threads do not share a lock.

13. For fine-grained operations, there is an alternate approach that is often more efficient—the optimistic approach, whereby you proceed with an update, hopeful that you can complete it without interference. This approach relies on collision detection to determine if there has been interference from other parties during the update, in which case the operation fails and can be retried (or not).

14. The key to synchronization is to assume that two processors perform the exchange operation (atomic exchange) at the same time. The atomic exchange operation is inseparable, and the order of the two exchange operations will be determined by the write order mechanism, which also ensures that the two threads cannot acquire the synchronous variable lock at the same time.

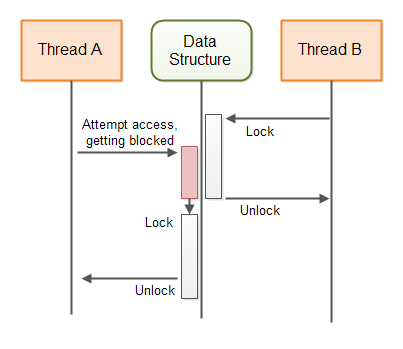
15. CAS is an optimistic locking technology. When multiple threads try to use CAS to update the same variable at the same time, only one of the threads can update the value of the variable, while the other threads fail. The failed thread is not suspended, but Inform this competition that it failed and can try again.

Q2

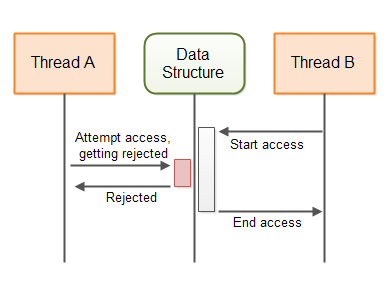
The main difference between blocking and non-blocking algorithms lies in the second step of their behaviour as described in the above two sections. In other words, the difference lies in what the blocking and non-blocking algorithms do when the requested action cannot be performed:

Blocking algorithms block the thread until the requested action can be performed. Non-blocking algorithms notify the thread requesting the action that the action cannot be performed. With a blocking algorithm a thread may become blocked until it is possible to perform the requested action.

Blocking Concurrency Algorithms:



Non-blocking Concurrency Algorithms:



Q3

1. **Creating an AtomicReference:**

AtomicReference atomicReference = new AtomicReference();

1. **Creating a Typed AtomicReference：**

AtomicReference<String> atomicStringReference =

new AtomicReference<String>();

1. **Getting the AtomicReference Reference：**

You can get the reference stored in an AtomicReference using the AtomicReference's get() method. If you have an untyped AtomicReference then the get() method returns an Object reference. If you have a typed AtomicReference then get() returns a reference to the type you declared on the AtomicReference variable when you created it.

AtomicReference<String> atomicReference =

new AtomicReference<String>("first value referenced");

String reference = atomicReference.get();

1. **Setting the AtomicReference Reference：**

AtomicReference atomicReference =

new AtomicReference();

atomicReference.set("New object referenced");

1. **Comparing and Setting the AtomicReference Reference**

String initialReference = "initial value referenced";

AtomicReference<String> atomicStringReference =

new AtomicReference<String>(initialReference);

String newReference = "new value referenced";

boolean exchanged = atomicStringReference.compareAndSet(initialReference, newReference);

System.out.println("exchanged: " + exchanged);

exchanged = atomicStringReference.compareAndSet(initialReference, newReference);

System.out.println("exchanged: " + exchanged);

Q4

In C ++ 11:

Work:Condition variables are a mechanism for synchronizing with global variables shared between threads. They mainly include two actions:

A thread hangs while waiting for "condition of condition variable to be satisfied";

Another thread makes a "condition" and gives a signal to wake up the waiting thread.

To prevent contention, the use of condition variables is always combined with a mutex; usually this lock is std :: mutex, and the management of this lock can only be std :: unique\_lock <std :: mutex> RAII template class .

Purpose: we can use condition\_variable to achieve synchronous operation between multiple threads; when the condition is not met, the related threads are blocked until a certain condition occurs, these threads will not be woken up.

Java:

Condition variables in java implement the java.util.concurrent.locks.Condition interface. The instantiation of condition variables is obtained by calling the newCondition () method on a Lock object. In this way, the condition is bound to a lock object . Therefore, condition variables in Java can only be used in conjunction with locks to control the security of concurrent programs accessing competing resources.

Q5

Condition is an interface. Condition queue is that When a thread calls the wait method, or calls the await related method through the Condition object, the thread enters the blocking state and is added to the corresponding condition queue. So condition queue calls the methods of the Condition Interface.

Q6

v = value.get() gets the current value of the counter, and if nobody else is trying to update the counter at the same time, old == v will be true, so the value is set to v+1 (i.e. it is incremented) and old is returned. The loop terminates since v == old.

Suppose someone else incremented the counter just after we did v = value.get(), then old == v would be false, and the method will immediately return old, which is the updated value. Since v != old now, the loop continues.

This way, the caller can know that the increment operation doesn't get lost by some other thread that tries to modify value at the same moment.

Q7

a) The push () method observes the current top node, constructs a new node on the stack, and then installs a new node if the top node has not changed since the initial observation. If the CAS fails, meaning that another thread has modified the stack, the process restarts.

b) top.compareAndSet 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.

c) public class ConcurrentStack<E> {

    AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();

    public void push(E item) {

        Node<E> newHead = new Node<E>(item);

        Node<E> oldHead;

        do {

            oldHead = head.get();

           newHead.next = oldHead;

        } while (!head.compareAndSet(oldHead, newHead));

    }

    public E pop() {

        Node<E> oldHead;

        Node<E> newHead;

        do {

            oldHead = head.get();

            if (oldHead == null)

               return null;

            newHead = oldHead.next;

        } while (!head.compareAndSet(oldHead,newHead));

        return oldHead.item;

    }

    static class Node<E> {

        final E item;

        Node<E> next;

        public Node(E item) { this.item = item; }

    }

}

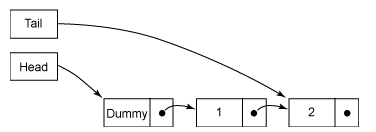
Q8

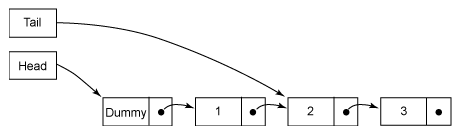
(a) Inserting elements at the end of a linked list usually involves updating two pointers: the "tail" pointer always points to the last element in the list, and the "next" pointer points to the newly inserted element from the last element in the past. Because two pointers need to be updated, two CAS are required.

(b) Link from the current last node of the queue to the new node, and move the tail pointer to the new last node.

(c) Yes, it works. Both pointers need to be updated

(d) Quiescent state and intermediate state when the new element is inserted but the tail pointer hasn’t been updated.





(e) public class LinkedQueue <E> {

    private static class Node <E> {

        final E item;

        final AtomicReference<Node<E>> next;

        Node(E item, Node<E> next) {

            this.item = item;

            this.next = new AtomicReference<Node<E>>(next);

        }

    }

    private AtomicReference<Node<E>> head

        = new AtomicReference<Node<E>>(new Node<E>(null, null));

    private AtomicReference<Node<E>> tail = head;

    public boolean put(E item) {

        Node<E> newNode = new Node<E>(item, null);

        while (true) {

            Node<E> curTail = tail.get();

            Node<E> residue = curTail.next.get();

            if (curTail == tail.get()) {

                if (residue == null) /\* A \*/ {

                    if (curTail.next.compareAndSet(null, newNode)) /\* C \*/ {

                        tail.compareAndSet(curTail, newNode) /\* D \*/ ;

                        return true;

                    }

                } else {

                    tail.compareAndSet(curTail, residue) /\* B \*/;

                }

            }

        }

    }

}