Advanced Concurrency and Data-Structures

By Mohit Kumar

Contents

- Proper way to Stop a Thread.
- Exception Handling in a Thread
- Limitations with “synchronized” keyword
- Synchronization Primitives
  - Locks
  - Barriers
  - Countdown latches
  - Condition variables.

To Stop a Thread

- Problem with “Stop()”
  - This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack).
  - If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior.
  - Many uses of stop should be replaced by code that simply modifies some variable to indicate that the target thread should stop running.
  - The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running.

To Stop a Thread

- If the target thread waits for long periods (on a condition variable, for example), the interrupt method should be used to interrupt the wait.

To Stop a Thread

- ThreadDeath Exception
  - It extends from java.lang.Error.
  - An instance of ThreadDeath is thrown in the victim thread when the stop method with zero arguments in class Thread is called.
  - An application should catch instances of this class only if it must clean up(extremely difficult!!!! Why ?? later) after being terminated asynchronously. If ThreadDeath is caught by a method, it is important that it be rethrown so that the thread actually dies.
  - The top-level error handler does not print out a message if ThreadDeath is never caught.

To Stop a Thread

- A thread can throw a ThreadDeath exception *almost anywhere*. All synchronized methods and blocks would have to be studied in great detail, with this in mind.
- A thread can throw a second ThreadDeath exception while cleaning up from the first (in the catch or finally clause). Cleanup would have to repeated till it succeeded. The code to ensure this would be quite complex.
- In sum, it just isn't practical.

So How to stop a thread

- Use a blinker

```
private volatile Thread blinker;
public void stopThread() {
        blinker = null;
}
public void run() {
        Thread thisThread = Thread.currentThread();
        while (blinker == thisThread) {
                try {
                        thisThread.sleep(interval);
                } catch (InterruptedException e){ }
                        repaint();
        }
}
```

So How to stop a thread

- Sometimes however the blinker(for elegance sake) need not be desirable.
- The Thread class however provides an inherent blinker(*interrupted status*) without physically declaring it in our derived class.
- The *interrupted status* can be set to true by "public void interrupt()" method.
- And can be inquired by "public static boolean **interrupted**()" method.
  - The side effect is the *interrupted status* of the thread is cleared by this method.
  - In other words, if this method were to be called twice in succession, the second call would return false (unless the current thread were interrupted again, after the first call had cleared its interrupted status and before the second call had examined it). To be called from within the thread.

So How to stop a thread

```
public void run(){
    while (!Thread.interrupted())
    {
        System.out.println("Some UseLess work");
        synchronized(somelist){
        // Stop method could leave the list in an inconsistent state;
        //exposed for some other thread to discover the damaged object
        // ***** however the interrupt method would finish the work and
    //then go down.
        somelist.add(new String());
    }
    System.out.println(Thread.interrupted());// would print false because the flag has
    //been cleared
}
```

So How to stop a thread

- But what if Thread is waiting or sleeping and has not checked the flag.

```
public void run(){
    while(!Thread.interrupted()){
        try{
                Thread.sleep(10000);
        }catch(InterruptedException ie){
        // if interrupted from sleep or wait the flag is cleared quite like
        // "interrupted()" method
        ie.printStackTrace();
        // break cause some body wants the thread to end.
        // same would apply for the wait() method as well;
        break;
    }
}
```

So How to stop a thread

- Another way to check the flag “public boolean **isInterrupted**()”
  - Tests whether this thread has been interrupted. The *interrupted status* of the thread is unaffected by this method. Normally to be called from a thread which needs to check on the interrupted thread

Handlers for uncaught Exceptions

- What if the thread gets terminated by an unchecked Exception.The point is there is no catch clause to propagate the exception object to. The “run()” method’s signature cant be changed.
- “public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh) ” Sets the handler and is invoked when this thread abruptly terminates due to an uncaught exception.
- A thread can take full control of how it responds to uncaught exceptions by having its uncaught exception handler explicitly set.
- The handler object implements the “Thread.UncaughtExceptionHandler” interface and overrides the “uncaughtException(Thread t, Throwable e) ” method and do the required handling

Handlers for uncaught Exceptions

```
class ThreadUncheckedExceptionHandler implements
java.lang.Thread.UncaughtExceptionHandler
{
        public void uncaughtException(Thread t,Throwable e){
                // do something ,probably log the error and exit
                // just for your knowledge the thread which threw up
                // is still alive.
        }
}
```

Handlers for uncaught Exceptions

- There is one more method to set a default handler for all threads. Preference would be given to thread-specific handler if installed by the previous method.
- If thread-specific handler is not installed then the default handler will take over which can be set by the “public static void **setDefaultUncaughtExceptionHandler**(Thread.UncaughtExceptionHandler eh)” method.

Threads in "SMP" boxes!!!???

- **Read at Your Own Risk**

Threads in "SMP" boxes!!!???

- Consider this simple looking code.
- It has got obvious problems. To start with it might end up initializing the resource twice.

```
class SomeClass {
      private Resource resource = null;

      public Resource getResource() {
        if (resource == null)
          resource = new Resource();
        return resource;
      }
    }
```

Threads in "SMP" boxes!!!???

- So we could synchronize the method. But if we do that the lock would be acquired every time.

```
// Correct multithreaded version
class SomeClass {
    private Resource r = null;
    public synchronized Resource getResource() {
        if (r == null)
            r = new Resource();
            return r;
        }
    }
    // other functions and members...

}
```

Threads in "SMP" boxes!!!???

- Some smart guy came with up with this solution……(It is called a DCL(Double check Lock))

```
class SomeClass {
      private Resource resource = null;

      public Resource getResource() {
        if (resource == null) {
          synchronized (whatever){
            if (resource == null)
              resource = new Resource();
          }
        }
        return resource;
      }
    }
```

- Except it doesn't work

Threads in "SMP" boxes!!!???

- WHY???
- We need to go down to the processors to understand this….

Instruction Reordering

- There are two interrelated aspects
  - Instruction Reordering
    - In the beginning, a CPU could write directly to memory simply by wiggling the voltages on a few wires that connected the CPU chip to the memory chip. **All was well with the world**.
    - Even with multithreaded systems, only one path to memory existed, and reads and writes to memory always occurred whenever the CPU executed the associated machine instruction.
    - The introduction of memory caches didn't fundamentally change that model . Indeed, the cache is transparent to the program if it's implemented correctly.

Instruction Reordering

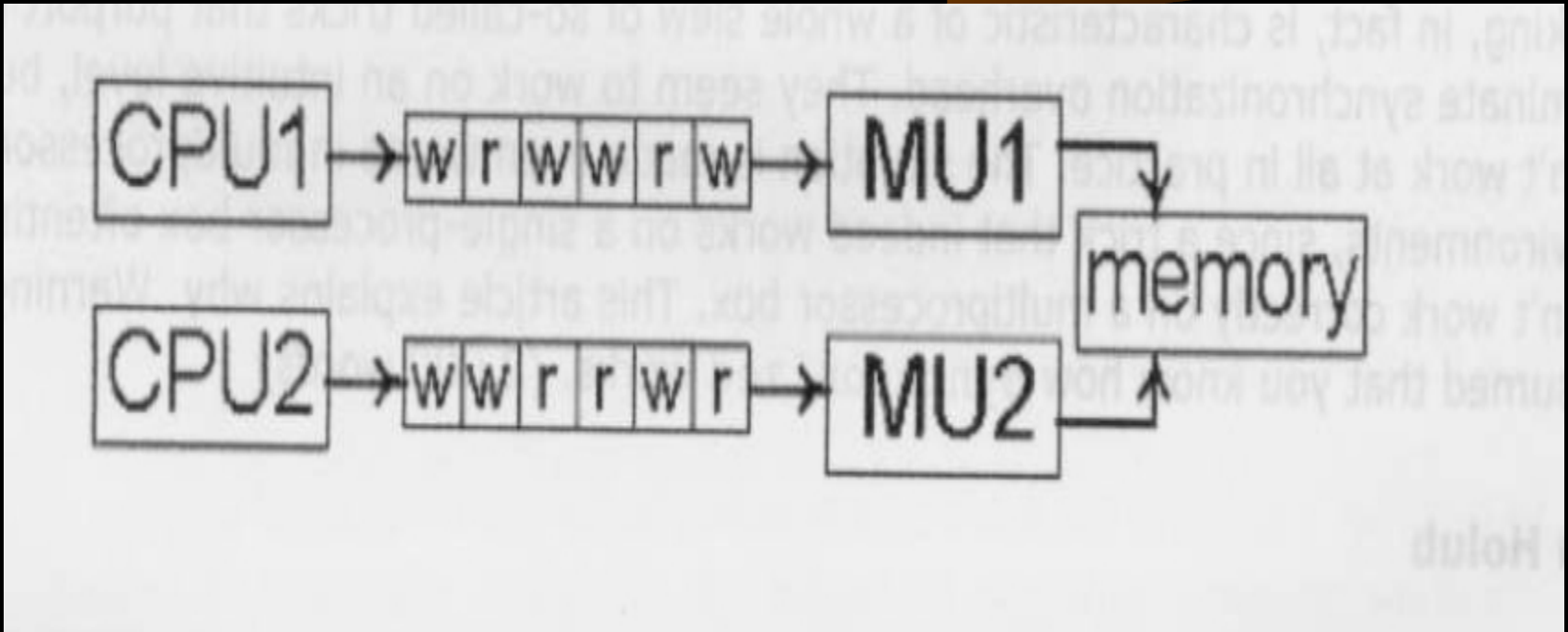
- That simple memory model -- the CPU issues an instruction that modifies memory with an immediate effect -- remains in most programmers' minds.

- Then somebody had the bright idea that two processors could run in the same box at the same time, sharing a common memory store (Suddenly, the world became much more complicated).
- In that situation, a given CPU can no longer access memory directly because another CPU might be using the memory at the same time. To solve the problem, along came a traffic-cop chip, called a *memory unit.*

Instruction Reordering

- Each CPU was paired with its own memory unit, and the various memory units coordinated with each other to safely access the shared memory.
- Under that model, a CPU doesn't write directly to memory but requests a read or write operation from its paired memory unit, which updates the main memory store when it can get access. Those early memory units effectively managed simple read or write request queues.

Instruction Reordering



Instruction Reordering

- All was well with the world until some bright engineer (I believe at DEC -- at least the Alpha, to my knowledge, was the earliest machine that worked like that) noticed a way to optimize memory operations with the hardware.
- Consider this code:

```
int a[] = new int[n];
int b[] = new int[n];
for( int i = 0; i < a.length; ++i )
    b[i] = a[i];
```

- The resulting read/write-request queue shown next. If we allow the memory units to shuffle around the order of pending read/write requests while they're waiting to get access to the main memory store , then we can turn all these atomic moves into one big block move." That is, the memory unit can transform the requests and can be sinfully fast.

Instruction Reordering

- That reordering strategy became known as a *relaxed* memory model.
- Now things are a lot faster."
- "But, but, but, now the order in which we write things to memory may not be the order in which the writes actually occur!" (More correctly, the writes might become visible in nonsource code order.)
- Memory Barriers are ways to ensure from the programmers perspective that reordering does not propagate unchecked.Requests issued between two memory barriers can be reordered, but no request can move past a memory barrier." (I'm simplifying a bit, since sometimes movement is possible, but that's the central concept.****)
- Requests are not permitted to migrate across the barrier. That way, at least we can guarantee that a write can occur before a subsequent read request is issued.

Instruction Reordering

- The point is the compiler will not change the meaning of the code for the executing “Thread” but for a different thread probably executing on a different processor the code changes would be visible in non-source code order.

Instruction Reordering

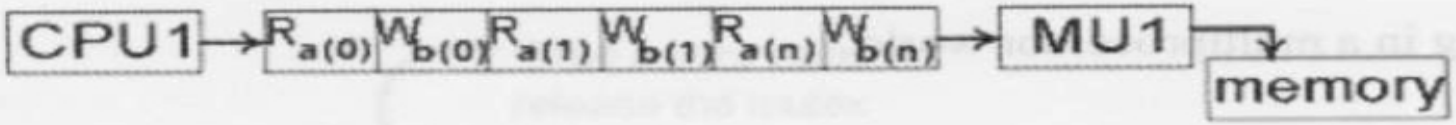


Figure 2. Copying an array: the initial state

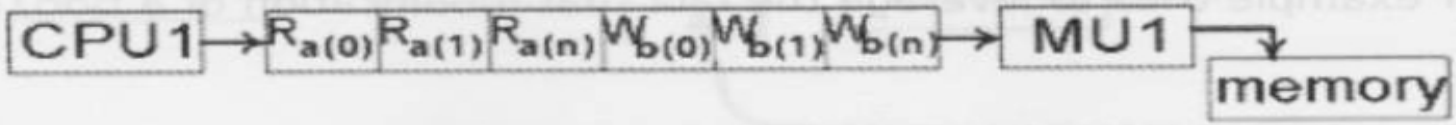
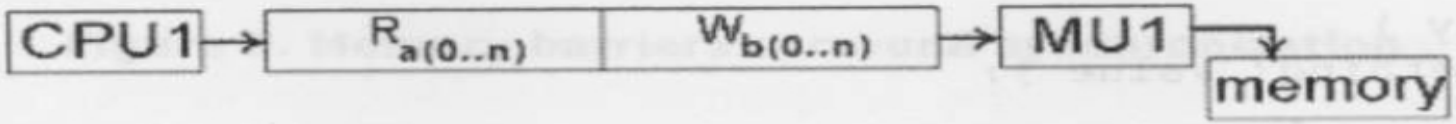


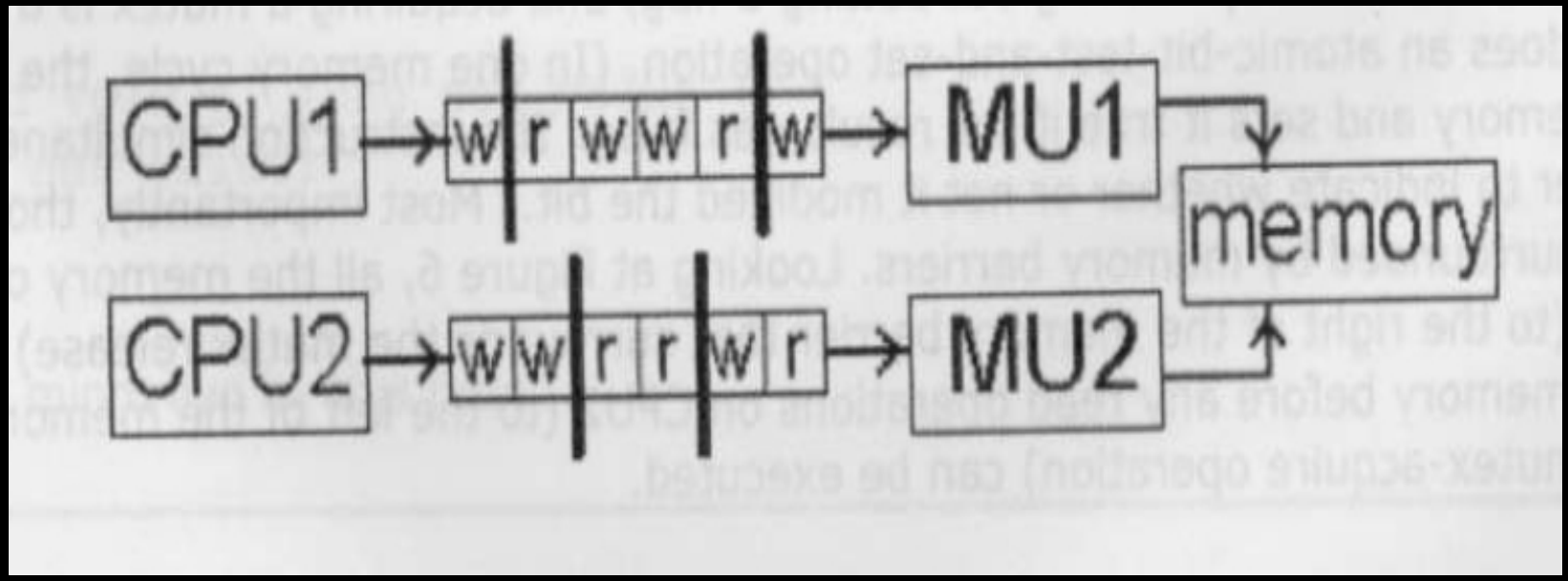
Figure 3. Copying an array: operations rearranged



Instruction Reordering

- Requests are not permitted to migrate across the barrier. That way, at least we can guarantee that a write can occur before a subsequent read request is issued.

Instruction Reordering



Instruction Reordering

- So why, you might ask, do I care about all that? The answer: programmers are too clever for their own good.
- They read how synchronized works: it goes out to the OS and acquires an exclusion semaphore (or mutex); an expensive operation, so too-clever programmers try to work around it.
- A classic too-clever example tries to leverage the fact that modification of a boolean is atomic.

```
boolean is_okay;
long    value;
void f()
{
   if( is_okay )
      do_something( value );
}
void g( long new_value )
{
   value = new_value;
   is_okay = true;
}
```

Instruction Reordering

- In the previous code excerpt, the f() method is called from one thread, and the g() method is called from another.
- The problem is that, when the two threads run on separate processors, the modifications of value and is_okay are found within the same memory barriers and are therefore subject to reordering.
- The is_okay field can effectively be set true *before* the new value of value is available. (don't quibble with me about visibility versus actual modification -- the effect is the same.)
- Note that the reordering would not matter in a single-threaded situation, where g() would complete in its entirety before either value could be used.

Instruction Reordering

- So why doesn't it work

```
class SomeClass {
      private Resource resource = null;

      public Resource getResource() {
        if (resource == null) {
          synchronized (whatever){
            if (resource == null)
              resource = new Resource();
          }
        }
        return resource;
      }
    }
```

Instruction Reordering

- The problem: the assignments that occur within the constructor(of Resources class) occur within the same memory barrier as the assignment to **resource**.
- As a consequence, it's possible for resource to be non-null, even though the assignments that occurred in the constructor are yet not visible.
- If one thread gets as far as the statement **resource = new Resource(),** and the assignment to resource is visible but the assignments within the constructor are not yet visible, then a second thread that calls **getResource**() method can return a pointer to an uninitialized object.
- This problem could occur in Single CPU boxes as well because the entire constructor could be inlined

Instruction Reordering

- **Is something broken?**
  When first confronted with the possibility of unpredictable behavior associated with DCL, many programmers worry that Java might be broken somehow.
- One reader wrote:

  Everything that I have heard about DCL gives me the creepy feeling that all sorts of strange and unpredictable things can happen when compiler optimizations interact with multithreading. Isn't Java supposed to protect us from accessing uninitialized objects and other unpredictable phenomena?
- The good news is no, Java is not broken, but multithreading and memory coherency are more complicated subjects than they might appear.
- Fortunately, you don't have to become an expert on the JMM. You can ignore all of this complexity if you just use the tool that Java provides for exactly this purpose -- synchronization.

Instruction Reordering

- The Java architects strove to allow Java to perform well on cutting-edge hardware -- at the cost of a somewhat heavyweight and hard-to-understand synchronization model.
- This complicated model has led people to try and outsmart the system by concocting clever schemes to avoid synchronization, such as DCL. But the problems with DCL result from the failure to use synchronization, not with the Java Memory Model itself.
- **If you synchronize every access to a variable that might have been written, or could be read by, another thread, you will have no memory coherency problems.**

Instruction Reordering

- **Don't try to fool the compiler**
  - By far the most common category of suggested DCL fixes are those that try to fool the compiler into performing certain operations in a specific order.
  - Attempting to trick the compiler proves dangerous for many reasons, the most obvious being that you might succeed in only fooling yourself into thinking that you've fooled the compiler.
  - Believing that you've tricked the compiler can give you a false sense of confidence, when maybe you've only fooled *this* version of *this* compiler in *this* case.
  - The *Java Language Specification* (*JLS*) gives Java compilers and JVMs a good deal of latitude to reorder or optimize away operations.

Instruction Reordering

- Java compilers are only required to maintain *within-thread as-if-serial* semantics, which means that the executing thread must not be able to detect any of these optimizations or reorderings.
- However, the *JLS* makes it clear that in the absence of synchronization, *other* threads might perceive memory updates in an order that "may be surprising."

Instruction Reordering

```
public Resource getResource() {
   if (resource == null) {
     synchronized {
       if (resource == null) {
         Resource temp = new Resource();
         resource = temp;
       }
     }
   return resource;
 }
```

- Above, you might think that as-if-serial semantics requires that the construction complete before resource is set, but that view is only from the perspective of the executing thread.
- Actually, the compiler is free to completely optimize away the **temporary variable(**temp**)**.

Instruction Reordering

- Though numerous tricks have been suggested to prevent the compiler from optimizing away the temporary variable, such as making it public, the compiler can still vary the order in which assignments are made inside the synchronized block as long as the executing thread can't tell the difference.

Instruction Reordering

```
private volatile boolean initialized = false;

public Resource getResource() {
  if (resource == null || !initialized) {
    synchronized {
      if (resource == null)
        resource = new Resource();
    }
    initialized = (resource != null);//****
  }
  return resource;
}
```

- At first, the approach taken above looks promising. Because initialized is set after the synchronized block exits, it appears that resource will have been fully written to memory before initialized is set. The above code even attempts to ensure that initialized is not set until resource is set by making initialized's value depend on resource's value. However, synchronization doesn't work quite so literally.

Instruction Reordering

- The compiler or JVM can move statements *into* synchronized blocks to reduce the cache-flush penalties associated with synchronization. But this means that from the perspective of other threads, initialized could still appear to be set before resource, and all of resource's fields are flushed to main memory.

Instruction Reordering

- **Even if you can fool the compiler, you can't fool the cache**
  - It's hard to fool the compiler, even if you can think like one. But if you do succeed in tricking the compiler, you still aren't guaranteed correct programs with respect to the JMM.
  - Compiler optimizations are only one source of potential reorderings; the processor and the cache can also affect the order in which other threads perceive memory updates.
  - A modern processor can execute multiple instructions simultaneously, or out of order, as long as it can determine that the results of one operation are not required for another operation. Also, write-back caches might vary the order in which memory writes are committed to main memory.
  - As an example, consider this simple class:

```
public class SomeObject {
      int a;
     public SomeObject() {
        a = 1;
      }
}
```

Instruction Reordering

- Even if you were assured that the processor would execute these instructions in exactly this order, other threads -- running on other processors -- examining main memory might not see them happen in that order.
- Even though FirstWrite(**a = 1**) executes before SecondWrite(**assigning of reference**), the cache on the executing processor could flush the results of SecondWrite to main memory before it flushes the results of FirstWrite.
- As a result, another thread could see someField initialized to a partially constructed SomeObject.
- This example might shed some light on a common misperception about the JMM and memory access reordering: the reorderings occur at the statement, method, or byte-code level.
- In reality, you should be more concerned about reorderings at the memory-fetch level.

Instruction Reordering

- After the Java compiler emits its byte code, and the JIT compiles it to machine code, the machine code will execute on a real processor with a real cache. The JMM specifies what sort of hardware-based reorderings it will tolerate -- and it is simply not the case that the JMM expects **memory coherency** across threads.
- If you are not familiar with the specifics of what happens in modern processors(super scalar heavily pipelined processors(can execute more than one instructions at a time and seeming out of order(Most modern processors fall into that criteria))) and caches, you might find this sort of non-determinism surprising and even disturbing.
- Why are these sorts of nondeterminism in processors and caches tolerated? Because they provide us with better performance.

Instruction Reordering

- Many of the recent advances in computing performance have come through increased parallelism. That is why the JMM doesn't assume that memory operations performed by one thread will be perceived as happening in the same order by another thread -- so as not to hamstring Java's performance on modern hardware.

Memory Coherency

- **Even if you fool the compiler *and* the cache... there is memeory coherency…(hahaha)**
  - Even if you were able to guarantee that writes to memory are completed in the desired order, that *still* might not be enough to render your programs correct with respect to the JMM.
  - It is actually possible, under the current JMM, to force Java to update several variables to main memory in a specific order.
  - You could do this::

Memory Coherency

```
class FullMemoryBarrierSingleton {
        private static boolean initialized = false;
        private static Resource resource = null;
        private static Object lock = new Object();

        public static Resource getResource() {
          if (!initialized) {
            synchronized (lock) {
              if (!initialized && resource == null)
                resource = new Resource();
            }
            synchronized (lock) {
              initialized = true;
            }
          }
          return resource;
        }
      }
```

Memory Coherency

- The current JMM does not permit the JVM to merge two synchronized blocks. So another thread could not possibly see initialized set before resource and all its fields are fully initialized and written to main memory. So have we solved the DCL problem?
- FullMemoryBarrierSingleton appears to avoid synchronization on the most common code path, without any obvious memory model hazards. Unfortunately, this is not exactly true.

Memory Coherency

- Some processors exhibit *cache coherency(obviously some don't(the one's that don't are an order of times faster than the one's that do))*, which means that regardless of the contents of any given processor's cache, each processor sees the same values of memory(conversely for processors that don't exhibit cache coherency they could see different values of the same memory).
- (Of course, depending on what is cached where, access to certain memory locations might be faster from some processors than from others.(NUMA architectures))
- While cache coherency certainly makes life easier for programmers, it substantially complicates the hardware and limits how many processors can connect to the same memory bus.

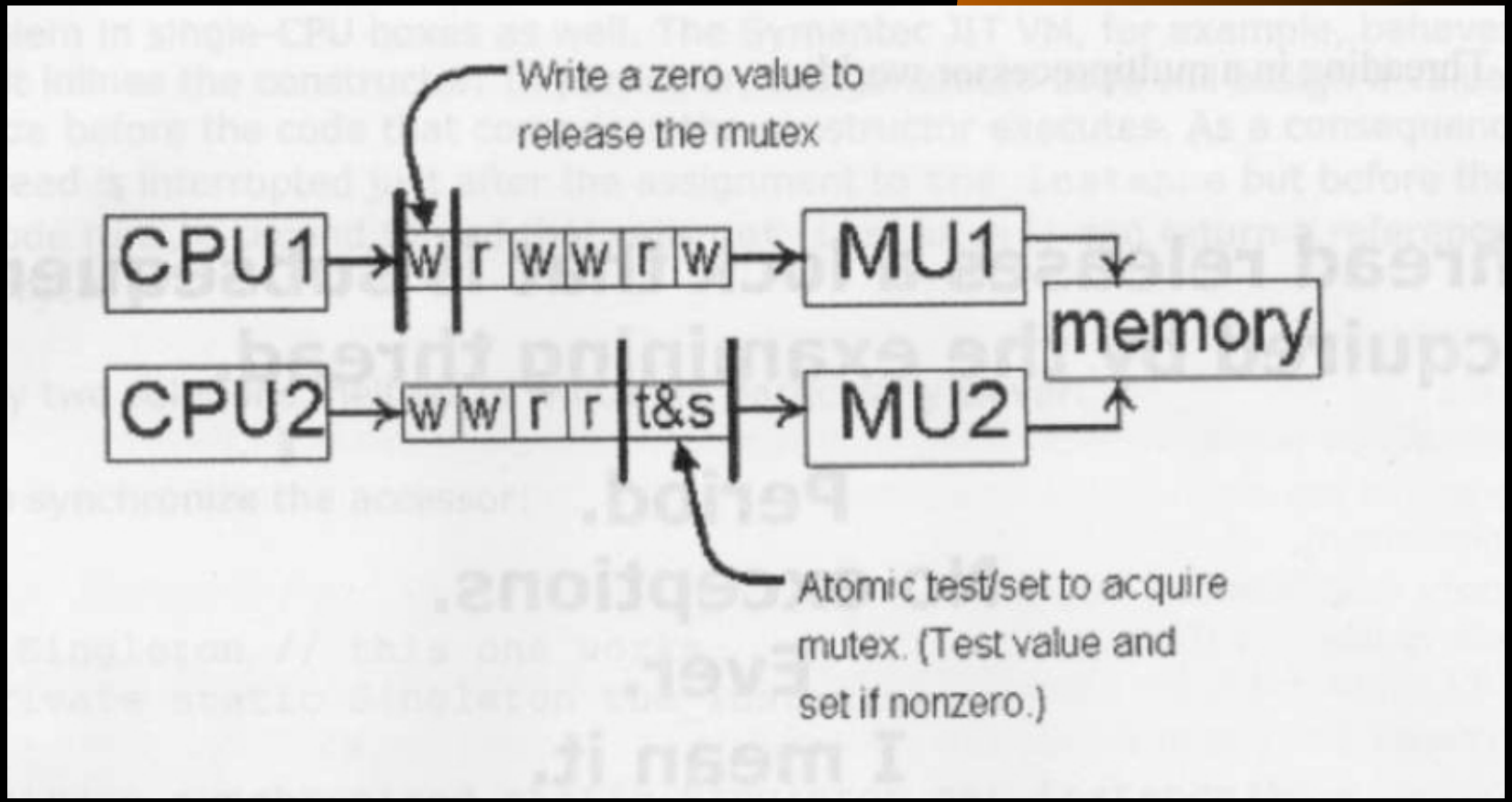
Memory Coherency

- An alternative architectural approach to cache coherency is to allow each processor to update its own cache independently, but offer some sort of synchronization mechanism to make sure that updates by one processor become visible to other processors in a deterministic manner.
- That mechanism is generally the *memory barrier*, and many processors, such as Alpha, PowerPC, and Sparc offer explicit memory barrier instructions. Generally, there are two types of memory barrier instructions:
  - A read barrier, which invalidates the contents of the executing processor's cache so that changes made by other processors can become visible to the executing processor
  - A write barrier, which writes out the contents of the executing processor's cache, so that changes made by the executing processor can become visible to others

Memory Coherency

- Most importantly, those operations are effectively surrounded by memory barriers. Looking at next picture, all the memory operations that occur on CPU1 (to the right of the memory barrier that surrounds the mutex release) must be visible in main memory before any read operations on CPU2 (to the left of the memory barrier that surrounds the mutex-acquire operation) can be executed.

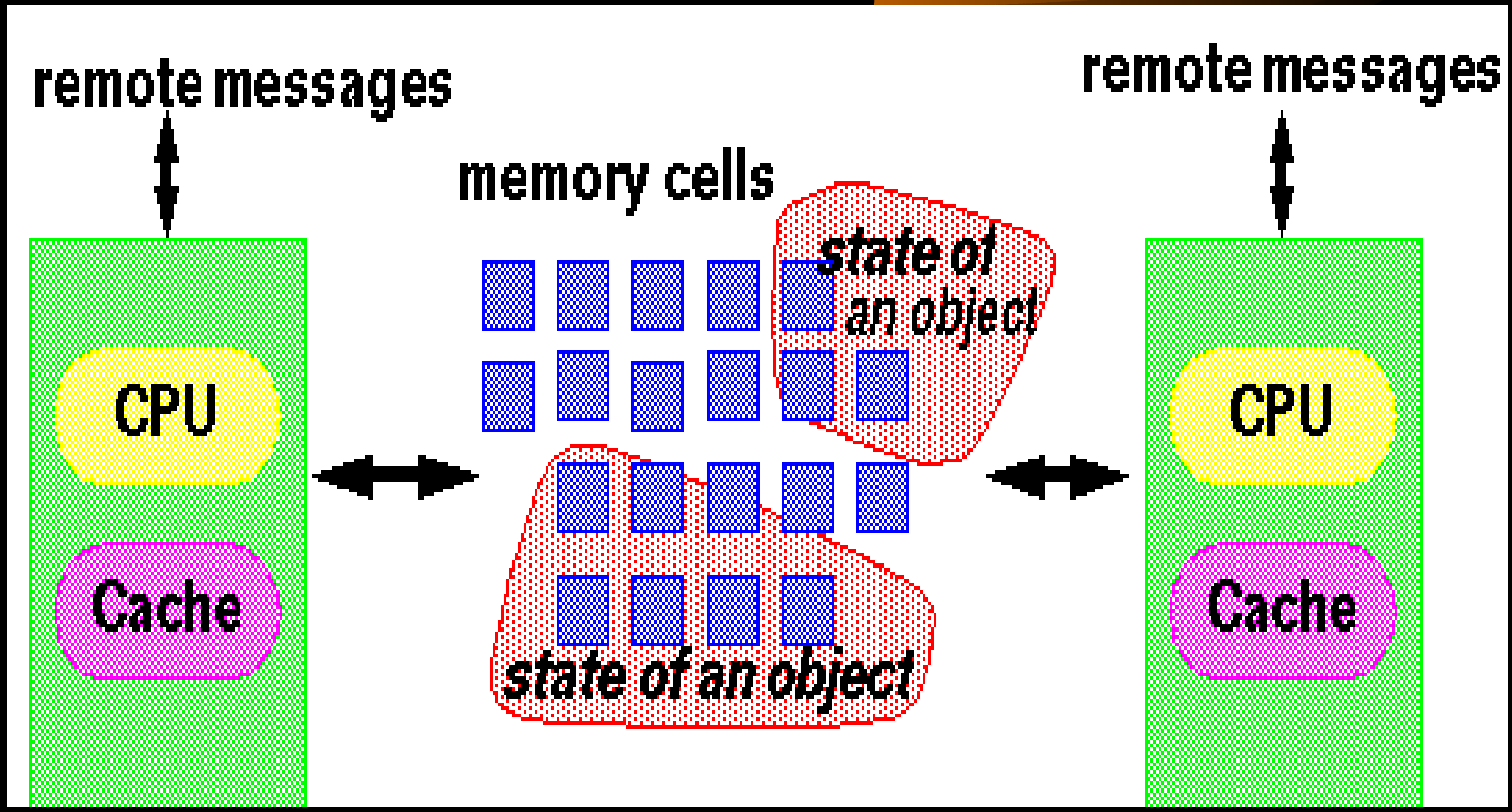
Memory Coherency



Memory Coherency

- On architectures without cache coherency, two processors can potentially fetch a value from the same memory address and see different results. To avoid this problem, either the programmer or compiler must use memory barrier instructions.
- The JMM was designed to support architectures both with and without cache coherency. T
- he JMM requires that a thread perform a read barrier after monitor entry and a write barrier before monitor exit.
- FullMemoryBarrierSingleton does indeed force the initializing thread to perform two write barriers so that resource and initialized are written to main memory in the proper order. So what could be wrong?
- The problem is that the other threads don't necessarily perform a read barrier after determining that initialized is set, so they could possibly see stale values of resource or resource's fields.

Memory Coherency



Conclusion

- DCL, and other techniques for avoiding synchronization, expose many of the complexities of the JMM. The issues surrounding synchronization are subtle and complicated -- so it is no surprise that many intelligent programmers have tried, but failed, to fix DCL.

Memory Coherency-a simple example

- We already know about the data garbling of global Objects and variable. But what we see next is much finer example of garbling in which even the copying of data from right to left need not be atomic.
- NOTE: You can actually peek at the virtual machine bytecodes that execute each statement in our class.

Memory Coherency-a simple example

- Run the command “javap -c -v Bank” to decompile the Bank.class file. For example, the line accounts[to] += amount; is translated into the following bytecodes:

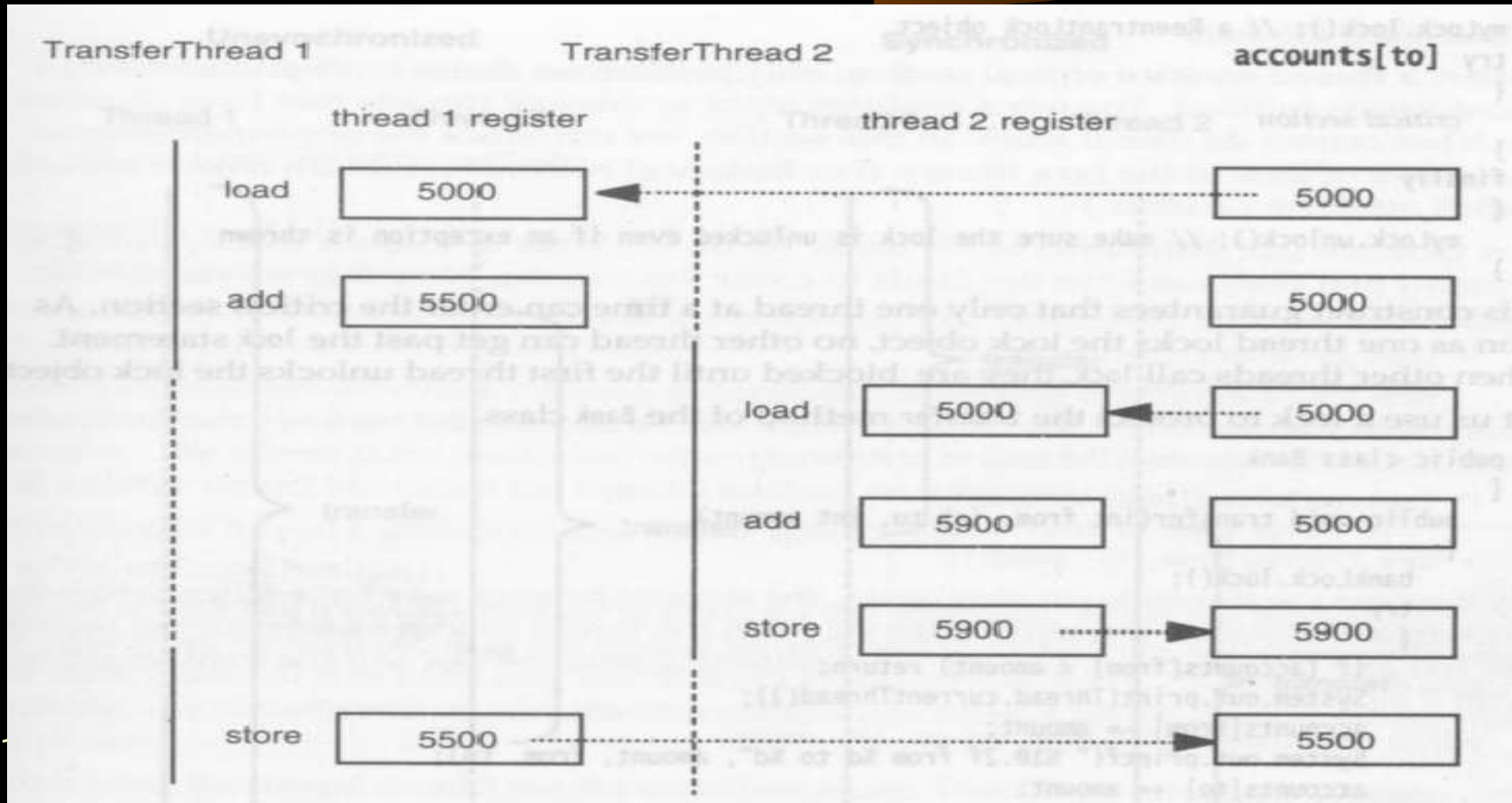
```
aload_0
getfield  #2; //field accounts:[D
iload_2
dup2
daload
dload_3
dadd
dastore
```

Memory Coherency-a simple example

- What these codes mean does not matter. The point is that the increment command is made up of several instructions, and the thread executing them can be interrupted at the point of any instruction.
- Which means the count which may be counting the number of hits to the method may not be the correct count.

Memory Coherency-a simple example

- Something like this



Memory Coherency-a simple example

- Sometimes, it seems excessive to pay the cost of synchronization just to read or write an instance field or two. After all, what can go wrong? Unfortunately, with modern processors and compilers, there is plenty of room for error:
  - Computers with multiple processors can temporarily hold memory values in registers or local memory caches. As a consequence, threads running in different processors may see different values for the same memory location!(we will just find that out!!!!!!!!!!:P)

The Old Java Memory Model

- What is a memory model, and why do I need one?
  - A memory model describes the relationship between variables in a program (instance fields, static fields, and array elements) and the low-level details of storing them to and retrieving them from memory in a real computer system.
  - Objects are ultimately stored in memory, but the compiler, runtime, processor, or cache may take some liberties with the timing of moving values to or from a variable's assigned memory location.
  - For example, a compiler may choose to optimize a loop index variable by storing it in a register, or the cache may delay flushing a new value of a variable to main memory until a more opportune time. All of these optimizations are to aid in higher performance, and are generally transparent to the user, but on multiprocessor systems, these complexities may sometimes show through

The Old Java Memory Model

- The JMM allows the compiler and cache to take significant liberties with the order in which data is moved between a processor-specific cache (or register) and main memory, unless the programmer has explicitly asked for certain visibility guarantees using synchronized or volatile.
- This means that in the absence of synchronization, memory operations can appear to happen in different orders from the perspective of different threads.

The Old Java Memory Model

- The meaning of volatile in previous JMM.
  - In order to provide good performance in the absence of synchronization, the compiler, runtime, and cache are generally allowed to reorder ordinary memory operations as long as the currently executing thread cannot tell the difference. (This is referred to as *within-thread as-if-serial semantics*.)
  - Volatile reads and writes, however, are totally ordered across threads; the compiler or cache cannot reorder volatile reads and writes with each other.
  - Unfortunately, the JMM did allow volatile reads and writes to be reordered with respect to ordinary variable reads and writes, meaning that we cannot use volatile flags as an indication of what operations have been completed.
  - Consider the following code, where the intention is that the volatile field initialized is supposed to indicate that initialization has been completed.

The Old Java Memory Model

```
Map configOptions;
char[] configText;
volatile boolean initialized = false;

. . . // In thread A
configOptions = new HashMap();
configText =readConfigFile(fileName);
processConfigOptions(configText, configOptions);
initialized = true;

. . . // In thread B
while (!initialized)
sleep();
// use configOptions
```

The Old Java Memory Model

- The idea here is that the volatile variable initialized acts as a guard to indicate that a set of other operations have completed.
- It's a good idea, but under the old JMM it didn't work, because the old JMM allowed nonvolatile writes (such as the write to the configOptions field, as well as the writes to the fields of the Map referenced by configOptions) to be reordered with volatile writes.
- So another thread might see initialized as true, but not yet have a consistent or current view of the field configOptions or the objects it references.
- The old semantics of volatile only made promises about the visibility of the variable being read or written, and no promises about other variables. While this approach was easier to implement efficiently, it turned out to be less useful than initially thought.

The Old Java Memory Model

- The meaning of immutables in Previous JMM(using final)
  - Making all fields of an object final does not necessarily make the object immutable -- all fields must *also* be primitive types or references to immutable objects.)
  - Immutable objects, like String, are supposed to require no synchronization. However, because of potential delays in propagating changes in memory writes from one thread to another, there is a possible race condition that would allow a thread to first see one value for an immutable object, and then at some later time see a different value.

The Old Java Memory Model

- How can this happen? Consider the implementation of String in the Sun 1.4 JDK, where there are basically three important final fields: a reference to a character array, a length, and an offset into the character array that describes the start of the string being represented.
- String was implemented this way, instead of having only the character array, so character arrays could be shared among multiple String and StringBuffer objects without having to copy the text into a new array every time a String is created.
- For example, String.substring() creates a new string that shares the same character array with the original String and merely differs in the length and offset fields.

The Old Java Memory Model

- The string s2 will have an offset of 4 and a length of 4, but will share the same character array, the one containing "/usr/tmp", with s1.
- Before the String constructor runs, the constructor for Object will initialize all fields, including the final length and offset fields, with their default values.
- When the String constructor runs, the length and offset are then set to their desired values.
- But under the old memory model, in the absence of synchronization, it is possible for another thread to temporarily see the offset field as having the default value of 0, and then later see the correct value of 4. The effect is that the value of s2 changes from "/usr" to "/tmp".

```
String s1 = "/usr/tmp";
String s2 = s1.substring(4);   // contains "/tmp"
```

The New Java Memory Model

- **Using a volatile variable as a "guard"**
  - The JSR 133 Expert Group decided that it would be more sensible for volatile reads and writes not to be reorderable with any other memory operations -- to support precisely this and other similar use cases.
  - Under the new memory model, when thread A writes to a volatile variable V, and thread B reads from V, any variable values that were visible to A at the time that V was written are guaranteed now to be visible to B. The result is a more useful semantics of volatile, at the cost of a somewhat higher performance penalty for accessing volatile fields.

The New Java Memory Model

- Additional ordering guarantees are created when a thread is started, a thread is joined with another thread, a thread acquires or releases a monitor (enters or exits a synchronized block), or a thread accesses a volatile variable.
- The JMM describes the ordering guarantees that are made when a program uses synchronization or volatile variables to coordinate activities in multiple threads.

The New Java Memory Model

- **Final means final**
  - In the absence of synchronization, another thread could first see the default value for a final field and then later see the correct value.
  - Under the new memory model, there are additional guarantees between the write of a final field in a constructor and the initial load of a shared reference to that object in another thread.
  - When the constructor completes, all of the writes to final fields (and to variables reachable indirectly through those final fields) become "frozen," and any thread that obtains a reference to that object after the freeze is guaranteed to see the frozen values for all frozen fields. Writes that initialize final fields will not be reordered with operations following the freeze associated with the constructor.

The New Java Memory Model

- Further, any variables that can be reached through a final field of a properly constructed object, such as fields of an object referenced by a final field, are also guaranteed to be visible to other threads as well.
- This means that if a final field contains a reference to, say, a LinkedList, in addition to the correct value of the reference being visible to other threads, also the contents of that LinkedList at construction time would be visible to other threads without synchronization.
- The result is a significant strengthening of the meaning of final -- that final fields can be safely accessed without synchronization, and that compilers can assume that final fields will not change and can therefore optimize away multiple fetches

The New Java Memory Model

- Summary of JSR133
  - JSR 133 significantly strengthens the semantics of volatile, so that volatile flags can be used reliably as indicators that the program state has been changed by another thread.
  - As a result of making volatile more "heavyweight," the performance cost of using volatile has been brought closer to the performance cost of synchronization in some cases, but the performance cost is still quite low on most platforms.
  - JSR 133 also significantly strengthens the semantics of final. If an object's reference is not allowed to escape during construction, then once a constructor has completed and a thread publishes a reference to an object, that object's final fields are guaranteed to be visible, correct, and constant to all other threads without synchronization.

The New Java Memory Model

- These changes greatly strengthen the utility of immutable objects in concurrent programs; immutable objects finally become inherently thread-safe (as they were intended to be all along), even if a data race is used to pass references to the immutable object between threads.
- Consider the following example.
  - Because writer() writes f *after* the object's constructor finishes, the reader() will be guaranteed to see the properly initialized value for f.x: it will read the value 3.
  - However,f.y is not final; the reader() method is therefore not guaranteed to see the value 4 for it.

The New Java Memory Model

```
class FinalFieldExample {//
      final int x; int y;
      static FinalFieldExample f;
      public FinalFieldExample() {
                  x = 3;
                  y = 4;
      }
      static void writer() {
                  f = new FinalFieldExample();
      }
      static void reader() {
                  if (f != null) {
                              int i = f.x; // guaranteed to see 3
                              int j = f.y; // could see 0
                  }
      }
}
```

The New Java Memory Model

- In summary, concurrent access to a field is safe in these three conditions:
  - *The field is volatile.*
  - The field is final, and it is accessed after the constructor has completed.
  - **The** field access is protected by a lock.

DCL

- **Old Volatile**
  - Making the Resource volatile and all contained field inside the Resource class volatile
  - Or only Resource is volatile but the Object must be Immutable(fields are effectively final.)
  - But the problem is even volatile will have associated costs and every time that it is accessed.

DCL

- **New Volatile**
  - Under the new memory model, this "fix" to double-checked locking renders the idiom thread-safe. But that still doesn't mean that you should use this idiom!
  - The whole point of double-checked locking was that it was supposed to be a performance optimization, designed to eliminate synchronization on the common code path, largely because synchronization was relatively expensive in very early JDKs.
  - Not only has uncontended synchronization gotten *a lot* cheaper since then, but the new changes to the semantics of volatile make it relatively more expensive than the old semantics on some platforms. (Effectively, each read or write to a volatile field is like "half" a synchronization -- a read of a volatile has the same memory semantics as a monitor acquire, and a write of a volatile has the same semantics as a monitor release.)

DCL

- So if the goal of double-checked locking is supposed to offer improved performance over a more straightforward synchronized approach, this "fixed" version doesn't help very much either.

DCL Based on new Volatile(Doesn't work)

```
class SomeClass {
        private Resource resource = null;
        private volatile boolean guard=false;
        public Resource getResource() {
          if (resource == null) {
            synchronized (whatever){
              if (resource == null)
                Resource temp = new Resource();
                 guard=true;
                 resource=temp;
            }
          }
          return resource;
        }
      }
```

Solution for DCL??

- So without synchronizing the entire code(as in synchronizing the method) is it possible to solve the DCL..
- Fortunately Yes ……Some of them obvious and some of them not so obvious…

DCL

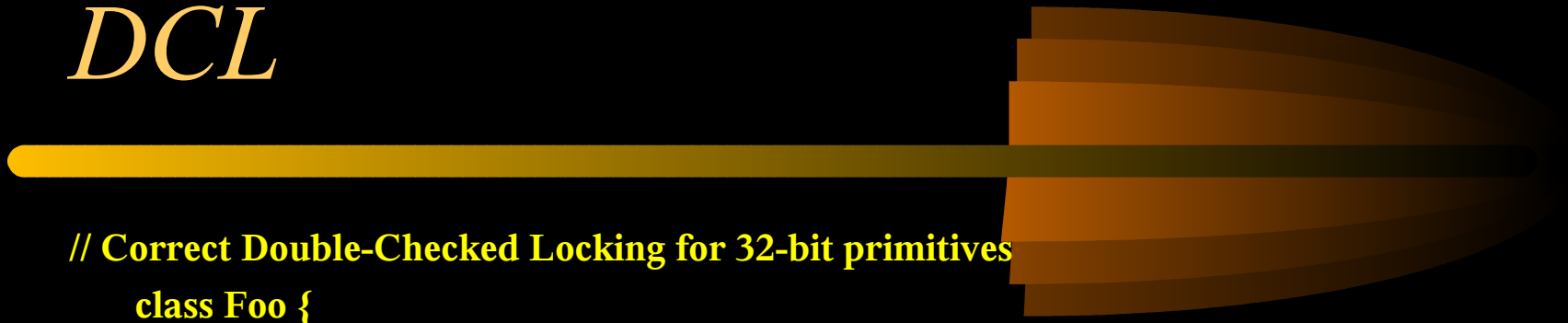
- **Making it work for static singletons**
  - If the singleton you are creating is static (i.e., there will only be one Helper created), as opposed to a property of another object (e.g., there will be one Helper for each Foo object, there is a simple and elegant solution.
  - Just define the singleton as a static field in a separate class. The semantics of Java guarantee that the field will not be initialized until the field is referenced, and that any thread which accesses the field will see all of the writes resulting from initializing that field.

```
class HelperSingleton {
    static Helper singleton = new Helper();
    public static HelperSingleton getInstance() {
            return HelperSingleton.singleton;
    }
}
```

DCL

- **It will work for 32-bit primitive values**
  - Although the double-checked locking idiom cannot be used for references to objects, it can work for 32-bit primitive values (e.g., int's or float's). Note that it does not work for long's or double's, since unsynchronized reads/writes of 64-bit primitives are not guaranteed to be atomic.

DCL



```
// Correct Double-Checked Locking for 32-bit primitives
    class Foo {
    private int cachedHashCode = 0;
    public int hashCode() {
        int h = cachedHashCode; //cache coherency will have no effect
        if (h == 0) synchronized(this) {
            if (cachedHashCode != 0) return cachedHashCode;
            h = computeHashCode();
            cachedHashCode = h;
         }
        return h;
     }
    // other functions and members...
    } // by the way good technique to write hashcode for immutables.....
```

DCL

- In fact, assuming that the computeHashCode function always returned the same result and had no side effects (i.e., idempotent), you could even get rid of all of the synchronization.

```
// Lazy initialization 32-bit primitives
// Thread-safe if computeHashCode is idempotent class Foo {
private int cachedHashCode = 0;
        public int hashCode() {
                int h = cachedHashCode;
                if (h == 0) {
                        h = computeHashCode();
                        cachedHashCode = h;
                }
                return h;
        }
        // other functions and members...
}
```

DCL-A fantastic Solution

- Enter (the fantastic)java.lang.ThreadLocal
  - This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread .
  - Could be used to maintain state of a client “outside” the Thread.
  - How does this version differ from the classic DCL implementation? Instead of checking to see if the shared resource field is nonnull, we use a ThreadLocal to store the answer to the question "Has this thread been through the synchronized block yet?"

DCL-A fantastic Solution

```
class ThreadLocalDCL {
        private static ThreadLocal initHolder = new ThreadLocal();
        private static Resource resource = null;

        public Resource getResource() {
          if (initHolder.get() == null) {
            synchronized {
              if (resource == null)
                resource = new Resource();
              initHolder.set(Boolean.TRUE);
            }
          }
          return resource;
        }
      }
```

Safe Construction Techniques

- *Don't let the "this" reference escape during construction*
  - The Java language offers a flexible and seemingly simple threading facility that makes it easy to incorporate multithreading into your applications. However, concurrent programming in Java applications is more complicated than it looks: there are several subtle (and not so subtle) ways to create data races and other concurrency hazards in Java programs.Allowing the this reference to escape during construction. This harmless-looking practice can cause unpredictable and undesirable results in your Java programs.

Safe Construction Techniques

- Most threading problems are unpredictable by their nature, and may not occur at all on certain platforms (like uniprocessor systems) or below a certain level of load. Because testing multithreaded programs for correctness is so difficult and bugs can take so long to appear, it becomes even more important to develop applications with thread safety in mind from the beginning.
- In this section, we're going to explore how a particular thread-safety problem -- allowing the this reference to escape during construction (which we'll call the *escaped reference* problem) -- can create some very undesirable results.
- We'll then establish some guidelines for writing thread-safe constructors.

Safe Construction Techniques

- Most concurrency hazards boil down to some sort of *data race*. A data race, or *race condition*, occurs when multiple threads or processes are reading and writing a shared data item, and the final result depends on the order in which the threads are scheduled .
- An example of a simple data race in which a program may print either 0 or 1, depending on the scheduling of the threads.
- The second thread could be scheduled immediately, printing the initial value of 0 for a. Alternately, the second thread might *not* run immediately, resulting in the value 1 being printed instead. The output of this program may depend on the JDK you are using, the scheduler of the underlying operating system, or random timing artifacts. Running it multiple times could produce different results.

Safe Construction Techniques

```
public class DataRace {
    static int a = 0;
    public static void main() {
        new MyThread().start();
        a = 1;
    }
    public static class MyThread extends Thread {
        public void run() {
            System.out.println(a);
        }
    }
}
```

Safe Construction Techniques

- Visibility hazards
  - There is actually another data race in Listing 1, besides the obvious race of whether the second thread starts executing before or after the first thread sets a to 1. The second race is a visibility race: the two threads are not using synchronization, which would ensure visibility of data changes across threads(the one we saw in the previous section).
  - We will only examine the first data race.

Safe Construction Techniques

- Don't publish the "this" reference during construction
  - One of the mistakes that can introduce a data race into your class is to expose the this reference to another thread before the constructor has completed.
  - Sometimes the reference is explicit, such as directly storing this in a static field or collection, but other times it can be implicit, such as when you publish a reference to an instance of a non-static inner class in a constructor.
  - Constructors are not ordinary methods -- they have special semantics for initialization safety. An object is assumed to be in a predictable, consistent state after the constructor has completed, and publishing a reference to an incompletely constructed object is dangerous.

Safe Construction Techniques

```
//As an example of introducing this sort of race condition into a constructor.
//It may look harmless, but it contains the seeds of serious concurrency problems
public class EventListener {
      public EventListener(EventSource eventSource) {
            // do our initialization    ...
            // register ourselves with the event source
            eventSource.registerListener(this);
      }
      public onEvent(Event e) {
          // handle the event
      }
}
```

Safe Construction Techniques

- On first inspection, the EventListener class looks harmless. The registration of the listener, which publishes a reference to the new object where other threads might be able to see it, is the last thing that the constructor does.
- But even ignoring all the Java Memory Model (JMM) issues such as differences in visibility across threads and memory access reordering, this code still is in danger of exposing an incompletely constructed EventListener object to other threads.
- Consider what happens when EventListener is subclassed .

Safe Construction Techniques

```
public class RecordingEventListener extends EventListener {
    private final ArrayList list;
    public RecordingEventListener(EventSource eventSource) {
            super(eventSource);
            list = Collections.synchronizedList(new ArrayList());
    }
    public onEvent(Event e) {
            list.add(e);
            super.onEvent(e);
    }
    public Event[] getEvents() {
            return (Event[]) list.toArray(new Event[0]);
    }
}
```

Safe Construction Techniques

- Because the Java language specification requires that a call to super() be the first statement in a subclass constructor, our not-yet-constructed event listener is already registered with the event source before we can finish the initialization of the subclass fields.
- Now we have a data race for the list field. If the event listener decides to send an event from within the registration call, or we just get unlucky and an event arrives at exactly the wrong moment,
- RecordingEventListener.onEvent() could get called while list still has the default value of null, and would then throw a NullPointerException exception. Class methods like onEvent() shouldn't have to code against final fields not being initialized.

Safe Construction Techniques

- The problem with is that EventListener published a reference to the object being constructed before construction was complete. While it might have looked like the object was *almost* fully constructed, and therefore passing this to the event source seemed safe, looks can be deceiving.
- Publishing the this reference from within the constructor, is a time bomb waiting to explode.

Safe Construction Techniques

- Don't implicitly expose the "this" reference
  - It is possible to create the escaped reference problem without using the this reference at all. Non-static inner classes maintain an implicit copy of the this reference of their parent object, so creating an anonymous inner class instance and passing it to an object visible from outside the current thread has all the same risks as exposing the this reference itself.

Safe Construction Techniques

```
public class EventListener2 {
    public EventListener2(EventSource eventSource) {
        eventSource.registerListener(
            new EventListener() {
                public void onEvent(Event e) {
                    eventReceived(e);
                }
            });
    }
    public void eventReceived(Event e) { }
}
```

Safe Construction Techniques

- The EventListener2 class has the same disease as its EventListener cousin previously:
- A reference to the object under construction is being published -- in this case indirectly -- where another thread can see it.
- If we were to subclass EventListener2, we would have the same problem where the subclass method could be called before the subclass constructor completes.

Safe Construction Techniques

- Don't start threads from within constructors
  - A special case is starting a thread from within a constructor, because often when an object owns a thread, either that thread is an inner class or we pass the this reference to its constructor (or the class itself extends the Thread class).
  - If an object is going to own a thread, it is best if the object provides a start() method, just like Thread does, and starts the thread from the start() method instead of from the constructor.
  - While this does expose some implementation details (such as the possible existence of an owned thread) of the class via the interface, which is often not desirable, in this case the risks of starting the thread from the constructor outweigh the benefit of implementation hiding.

Safe Construction Techniques

```
public class Safe {
      private Object me;
      private Set set = new HashSet();
      private Thread thread;
      public Safe() {
                // Safe because "me" is not visible from any other thread
               me = this;
               // Safe because "set" is not visible from any other thread
               set.add(this);
               // Safe because MyThread won't start until construction is complete
               // and the constructor doesn't publish the reference
               thread = new MyThread(this);
      }
      public void start() {    thread.start();  }
      private class MyThread(Object o) {   // rest ommited
}
```

Safe Construction Techniques

```
public class Unsafe {
    public static Unsafe anInstance;
    public static Set set = new HashSet();
    private Set mySet = new HashSet();
    public Unsafe() {
        // Unsafe because anInstance is globally visible
        anInstance = this;
        // Unsafe because SomeOtherClass.anInstance is globally visible
    SomeOtherClass.anInstance = this;
        // Unsafe because SomeOtherClass might save the "this" reference
        // where another thread could see it
        SomeOtherClass.registerObject(this);
        // Unsafe because set is globally visible
        set.add(this);
        // Unsafe because we are publishing a reference to mySet
        mySet.add(this);
        SomeOtherClass.someMethod(mySet);
```

Safe Construction Techniques

```
            // Unsafe because the "this" object will be visible from the new
            // thread before the constructor completes
            thread = new MyThread(this);
            thread.start();
    }
    public Unsafe(Collection c) {
            // Unsafe because "c" may be visible from other threads          c.add(this);
    }
}
```

Safe Construction Techniques

- The practices detailed above for thread-safe construction take on even more importance when we consider the effects of synchronization.
- For example, when thread A starts thread B, the Java Language Specification (JLS) guarantees that all variables that were visible to thread A when it starts thread B are visible to thread B, which is effectively like having an implicit synchronization in Thread.start().
- If we start a thread from within a constructor, the object under construction is not completely constructed, and so we lose these visibility guarantees.

Avoiding Scalability Bottleneck

- For situations like many common multithreaded applications (servers,etc), you can use other approaches, like pooling (Threads,connections, etc), for safely managing shared access. However, even pooling has some potential drawbacks from a scalability perspective.
- Because pool implementations must synchronize to maintain the integrity of the pool data structures, if all threads are using the same pool, program performance may suffer due to contention in a system with many threads accessing the pool frequently.

Avoiding Scalability Bottleneck

- Sometimes it is best not to share.
  - The ThreadLocal class appeared with little fanfare in version 1.2 of the Java platform. While support for thread-local variables has long been a part of many threading facilities, such as the Posix pthreads facility, the initial design of the Java Threads API lacked this useful feature.
  - A *thread-local variable* effectively provides a separate copy of its value for each thread that uses it. Each thread can see only the value associated with that thread, and is unaware that other threads may be using or modifying their own copies.

Avoiding Scalability Bottleneck

```
//ThreadLocal
public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            return (T)map.get(this);//

        // Maps are constructed lazily.  if the map for this thread
        // doesn't exist, create it, with this ThreadLocal and its
        // initial value as its only entry.
        T value = initialValue();
        createMap(t, value);
        return value;
    }
```

Avoiding Scalability Bottleneck

```
//ThreadLocal
public void set(T value) {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
}
void createMap(Thread t, T firstValue) {
        t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

Avoiding Scalability Bottleneck

- The benefits of ThreadLocal
  - ThreadLocal offers a number of benefits. It is often the easiest way to render a stateful class thread-safe, or to encapsulate non-thread-safe classes so that they can safely be used in multithreaded environments.
  - Using ThreadLocal allows us to bypass the complexity of determining when to synchronize in order to achieve thread-safety, and it improves scalability because it doesn't require any synchronization.
  - Under the newest version of the Java platform, version 1.4b2, performance of ThreadLocal and Thread.currentThread() has been improved significantly.
  - With these new improvements, ThreadLocal should be faster than other techniques such as pooling. Because it is simpler and often less error-prone than those other techniques, it will eventually be discovered as an effective way to prevent undesired interactions between threads.

Addressing Scalability Bottleneck

- The first associative collection class to appear in the Java class library was Hashtable, which was part of JDK 1.0. Hashtable provided an easy-to-use, thread-safe, associative map capability, and it was certainly convenient.
- However, the thread-safety came at a price -- all methods of Hashtable were synchronized. At that time, uncontended synchronization had a measurable performance cost.
- The successor to Hashtable, HashMap, which appeared as part of the Collections framework in JDK 1.2, addressed thread-safety by providing an unsynchronized base class and a synchronized wrapper, Collections.synchronizedMap.
- Separating the base functionality from the thread-safety Collections.synchronizedMap allowed users who needed synchronization to have it, but users who didn't need it didn't have to pay for it.

Addressing Scalability Bottleneck

- The simple approach to synchronization taken by both Hashtable and synchronizedMap -- synchronizing each method on the Hashtable or the synchronized Map wrapper object -- has two principal deficiencies.
  - It is an impediment to scalability, because only one thread can access the hash table at a time.
  - At the same time, it is insufficient to provide true thread safety, in that many common compound operations still require additional synchronization.
    - While simple operations such as get() and put() can complete safely without additional synchronization, there are several common sequences of operations, such as iteration or put-if-absent, which still require external synchronization to avoid data races.

Addressing Scalability Bottleneck

```
Map m = Collections.synchronizedMap(new HashMap());
List l = Collections.synchronizedList(new ArrayList());
// put-if-absent idiom -- contains a race condition
// may require external synchronization
if (!map.containsKey(key))
map.put(key, value);

// ad-hoc iteration -- contains race conditions
// may require external synchronization
for (int i=0; i<list.size(); i++) {
      doSomething(list.get(i));
}
// normal iteration -- can throw ConcurrentModificationException
// may require external synchronization
for (Iterator i=list.iterator(); i.hasNext(); ) {
     doSomething(i.next());
}
```

Addressing Scalability Bottleneck

- Conditional thread-safety
  - The synchronized collections wrappers, synchronizedMap and synchronizedList, are sometimes called *conditionally thread-safe* -- all individual operations are thread-safe, but sequences of operations where the control flow depends on the results of previous operations may be subject to data races.
  - The above snippet shows the common put-if-absent idiom -- if an entry does not already exist in the Map, add it. Unfortunately, as written, it is possible for another thread to insert a value with the same key between the time the containsKey() method returns and the time the put() method is called.
  - If you want to ensure exactly-once insertion, you need to wrap the pair of statements with a synchronized block that synchronizes on the Map m.

Addressing Scalability Bottleneck

- **False sense of confidence**
  - The conditional thread safety provided by synchronizedList and synchronizedMap present a hidden hazard -- developers assume that because these collections are synchronized, they are fully thread-safe, and they neglect to synchronize compound operations properly.
  - The result is that while these programs appear to work under light load, under heavy load they may start throwing NullPointerException or ConcurrentModificationException.

Addressing Scalability Bottleneck

Table 1. Scalability comparison between HashMap and LockPoolMap

Threads	Unsynchronized `HashMap` (unsafe)	Synchronized `HashMap`	`LockPoolMap`
1	1.1	1.4	1.6
2	1.1	57.6	3.7
4	2.1	123.5	7.7
8	3.7	272.3	16.7
16	6.8	577.0	37.9
32	13.5	1233.3	80.5

Addressing Scalability Bottleneck

- The synchronized collections classes, Hashtable and Vector, and the synchronized wrapper classes, Collections.synchronizedMap and Collections.synchronizedList, provide a basic conditionally thread-safe implementation of Map and List.
- However, several factors make them unsuitable for use in highly concurrent applications -- their single collection-wide lock is an impediment to scalability and it often becomes necessary to lock a collection for a considerable time during iteration to prevent ConcurrentModificationExceptions.

Addressing Scalability Bottleneck

- Now the idea should be to understand how scalable data structures can be constructed without compromising thread safety.
- One way Could be to reduce the lock granularity to the bucket level as opposed to a map wide lock

Addressing Scalability Bottleneck

```
public class LockPoolMap {
    private Node[] buckets;
    private Object[] locks;
    private static final class Node {
        public final Object key;
        public Object value;
        public Node next;
        public Node(Object key) {
            this.key = key;
        }
    }
    public LockPoolMap(int size) {
        buckets = new Node[size];
        locks = new Object[size];
        for (int i = 0; i < size; i++)
            locks[i] = new Object();
    }
```

Addressing Scalability Bottleneck

```
    public Object get(Object key) {
            int hash = hash(key);
            synchronized(locks[hash]) {
                    for (Node m=buckets[hash]; m != null; m=m.next)
                            if (m.key.equals(key))
                    return m.value;
            }
            return null;
    }
}
```

Addressing Scalability Bottleneck

```
public void put(Object key, Object value) {
        int hash = hash(key);
        synchronized(locks[hash]) {
                Node m;
                for (m=buckets[hash]; m != null; m=m.next) {
                        if (m.key.equals(key)) {
                                m.value = value;
                                return;
                        }
                }
        // We must not have found it, so put it at the beginning of the chain
                m = new Node(key);
                m.value = value;
                m.next = buckets[hash];
                buckets[hash] = m;
        }//end of synchronized
}
```

Addressing Scalability Bottleneck

- A structure like this performs pretty well but map wide operations like “size()” become tricky.
- Secondly a get and a set don’t count as structural modification and yet we need to synchronize because we don’t know is there is a concurrent put or remove going on.
  - With a little bit of more effort we should be able to remove synchronization for get and set operations.

Addressing Scalability Bottleneck

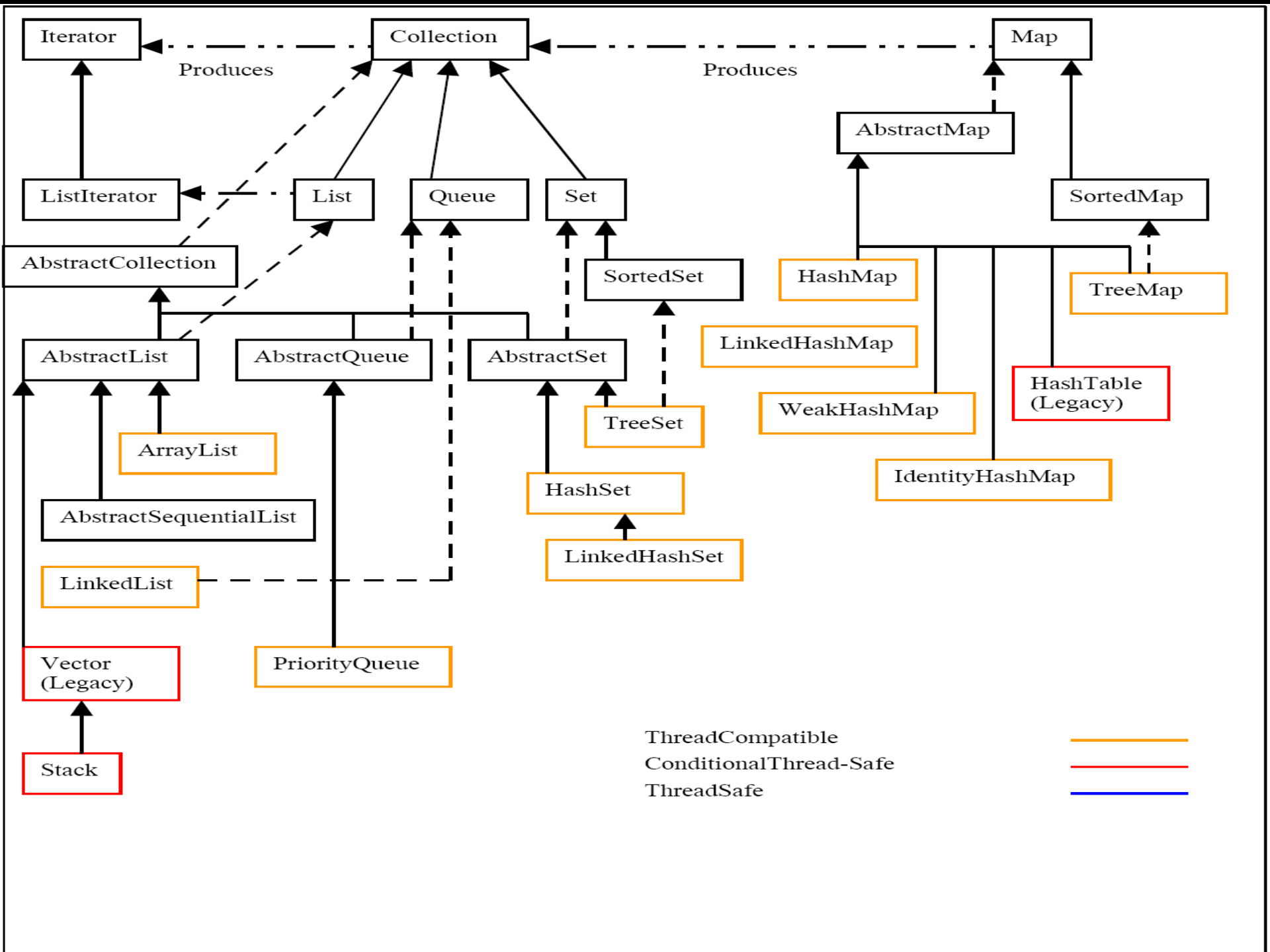
- Some Taxanomies about Thread-safety.
  - **Immutable**
    - Immutable objects are guaranteed to be thread-safe, and never require additional synchronization
  - **Thread-safe**
    - For a class to be thread-safe, it first must behave correctly in a single-threaded environment. If a class is correctly implemented, which is another way of saying that it conforms to its specification, no sequence of operations (reads or writes of public fields and calls to public methods) on objects of that class should be able to put the object into an invalid state, observe the object to be in an invalid state, or violate any of the class's invariants, preconditions, or postconditions.

Addressing Scalability Bottleneck

- **Conditionally thread-safe**
  - Conditionally thread-safe classes are those for which each individual operation may be thread-safe, but certain sequences of operations may require external synchronization.
- **Thread-compatible**
  - Thread-compatible classes are not thread-safe, but can be used safely in concurrent environments by using synchronization appropriately. This might mean surrounding every method call with a synchronized block or creating a wrapper object where every method is synchronized (like Collections.synchronizedList()). Or it might mean surrounding certain sequences of operations with a synchronized block.

Addressing Scalability Bottleneck

- **Thread-hostile**
  - Thread-hostile classes are those that cannot be rendered safe to use concurrently, regardless of what external synchronization is invoked. Thread hostility is rare, and typically arises when a class modifies static data that can affect the behavior of other classes that may execute in other threads. An example of a thread-hostile class would be one that calls System.setOut().


Iterator
Collection
Map
Produces
Produces
AbstractMap
ListIterator
List
Queue
Set
SortedMap
AbstractCollection
SortedSet
HashMap
TreeMap
LinkedHashMap
AbstractList
AbstractQueue
AbstractSet
HashTable
(Legacy)
WeakHashMap
TreeSet
ArrayList
HashSet
IdentityHashMap
AbstractSequentialList
LinkedHashSet
LinkedList
Vector
(Legacy)
PriorityQueue
Stack
ThreadCompatible
ConditionalThread-Safe
ThreadSafe

Building Blocks for Highly Concurrent Design

- **Inherent Problems with locking**
  - With locking, if one thread attempts to acquire a lock that is already held by another thread, the thread will block until the lock becomes available.
  - This approach has some obvious drawbacks, including the fact that while a thread is blocked waiting for a lock, it cannot do anything else. This scenario could be a disaster if the blocked thread is a high-priority task (a hazard known as *priority inversion*).
  - Using locks has some other hazards as well, such as deadlock (which can happen when multiple locks are acquired in an inconsistent order).
  - Another problem with lock-based algorithms is that if a thread holding a lock is delayed (due to a page fault(A linked list may have been paged out), scheduling delay, or other unexpected delay), then *no* thread requiring that lock may make progress.

Building Blocks for Highly Concurrent Design

- **Design Problem with Locks**
  - Few tasks can be truly parallelized in such a way as to require *no* coordination between threads.
  - Consider a thread pool, where the tasks being executed are generally independent of each other. If the thread pool feeds off a common work queue, then the process of removing elements from or adding elements to the work queue must be thread-safe, and that means coordinating access to the head, tail, or inter-node link pointers.
  - And it is this coordination that causes all the trouble.

Building Blocks for Highly Concurrent Design

- Even in the absence of hazards like the abovementioned, locks are simply a relatively coarse-grained coordination mechanism, and as such, are fairly "heavyweight" for managing a simple operation such as incrementing a counter or updating who owns a mutex.
- It would be nice if there were a finer-grained mechanism for reliably managing concurrent updates to individual variables;
- **and on most modern processors, there is.**

Building Blocks for Highly Concurrent Design

- Fifteen years ago, multiprocessor systems were highly specialized systems costing hundreds of thousands of dollars (and most of them had two to four processors).
- Today, multiprocessor systems are cheap and plentiful, nearly every major microprocessor has built-in support for multiprocessing, and many support dozens or hundreds of processors.
- To exploit the power of multiprocessor systems, applications are generally structured using multiple threads.
- But as anyone who's written a concurrent application can tell you, simply dividing up the work across multiple threads isn't enough to achieve good hardware utilization -- you must ensure that your threads spend most of their time actually doing work, rather than waiting for more work to do, or waiting for locks on shared data structures.

Building Blocks for Highly Concurrent Design

- As stated earlier, most modern processors include support for multiprocessing. While this support, of course, includes the ability for multiple processors to share peripherals and main memory, it also generally includes enhancements to the instruction set to support the special requirements of multiprocessing.
- **In particular, nearly every modern processor has instructions for updating shared variables in a way that can either detect or prevent concurrent access from other processors.**

Building Blocks for Highly Concurrent Design

- **Compare and swap (CAS)**
  - The first processors that supported concurrency provided atomic test-and-set operations, which generally operated on a single bit. The most common approach taken by current processors, including Intel and Sparc processors, is to implement a primitive called *compare-and-swap*, or CAS.
  - (On Intel processors, compare-and-swap is implemented by the cmpxchg family of instructions. PowerPC processors have a pair of instructions called "load and reserve" and "store conditional" that accomplish the same goal; similar for MIPS, except the first is called "load linked.")

Building Blocks for Highly Concurrent Design

```
//Code illustrating the behavior (but not performance) of compare-and-swap
public class SimulatedCAS {
    private int value;
    public synchronized int getValue() {
        return value;
    }
    public synchronized int compareAndSwap(int expectedValue, int newValue) {
        int oldValue = value;
        if (value == expectedValue)
          value = newValue;
        return oldValue;
    }
}
```

Building Blocks for Highly Concurrent Design

```
//Implementing a counter with compare-and-swap
 public class CasCounter {
       private SimulatedCAS value=new SimulatedCAS();
       public int getValue() {
              return value.getValue();
      }
      public int increment() {
      int oldValue = value.getValue();
      while (value.compareAndSwap(oldValue, oldValue + 1) != oldValue)
              oldValue = value.getValue();
              return oldValue + 1;
      }
}
```

Building Blocks for Highly Concurrent Design

- Concurrent algorithms based on CAS are called *lock-free*, because threads do not ever have to wait for a lock (sometimes called a mutex or critical section, depending on the terminology of your threading platform).
- Either the CAS operation succeeds or it doesn't, but in either case, it completes in a predictable amount of time. If the CAS fails, the caller can retry the CAS operation or take other action as it sees fit.
- A CAS operation includes three operands -- a memory location (V), the expected old value (A), and a new value (B). The processor will atomically update the location to the new value if the value that is there matches the expected old value, otherwise it will do nothing.

Building Blocks for Highly Concurrent Design

- In either case, it returns the value that was at that location prior to the CAS instruction. (Some flavors of CAS will instead simply return whether or not the CAS succeeded, rather than fetching the current value.) CAS effectively says "I think location V should have the value A; if it does, put B in it, otherwise, don't change it but tell me what value is there now."

Building Blocks for Highly Concurrent Design

- Instructions like CAS allow an algorithm to execute a read-modify-write sequence without fear of another thread modifying the variable in the meantime, because if another thread did modify the variable, the CAS would detect it (and fail) and the algorithm could retry the operation.
- **The above code illustrates the behavior (but not performance characteristics) of the CAS operation, but the value of CAS is that it is implemented in hardware and is extremely lightweight (on most processors):**

Building Blocks for Highly Concurrent Design

- Substantial research has gone into wait-free and lock-free algorithms (also called *nonblocking algorithms*) over the past 15 years, and nonblocking algorithms have been discovered for many common data structures.
- Nonblocking algorithms are used extensively at the operating system and JVM level for tasks such as thread and process scheduling.
- While they are more complicated to implement, they have a number of advantages over lock-based alternatives -- hazards like priority inversion and deadlock are avoided, contention is less expensive, and coordination occurs at a finer level of granularity, enabling a higher degree of parallelism.

Building Blocks for Highly Concurrent Design

- **Atomic variable classes**
  - Until JDK 5.0, it was not possible to write wait-free, lock-free algorithms in the Java language without using native code.
  - With the addition of the atomic variables classes in the java.util.concurrent.atomic package, that has changed.
  - The atomic variable classes all expose a compare-and-set primitive (similar to compare-and-swap), which is implemented using the fastest native construct available on the platform (compare-and-swap, load linked/store conditional, or, in the worst case, spin locks).
  - Nine flavors of atomic variables are provided in the java.util.concurrent.atomic package (AtomicInteger; AtomicLong; AtomicReference; AtomicBoolean; array forms of atomic integer; long; reference; and atomic marked reference and stamped reference classes, which atomically update a pair of values).

Building Blocks for Highly Concurrent Design

- The atomic variable classes can be thought of as a generalization of volatile variables, extending the concept of volatile variables to support atomic conditional compare-and-set updates.
- Reads and writes of atomic variables have the same memory semantics as read and write access to volatile variables.
- While the atomic variable classes might look superficially like the SynchronizedCounter example, the similarity is only superficial. **Under the hood, operations on atomic variables get turned into the hardware primitives that the platform provides for concurrent access, such as compare-and-swap.**

Building Blocks for Highly Concurrent Design

- A common technique for tuning the scalability of a concurrent application that is experiencing contention is to reduce the granularity of the lock objects used, in the hopes that more lock acquisitions will go from contended to uncontended.
- The conversion from locking to atomic variables achieves the same end -- by switching to a finer-grained coordination mechanism, fewer operations become contended, improving throughput.

Building Blocks for Highly Concurrent Design

- Nearly all the classes in the java.util.concurrent package use atomic variables instead of synchronization, either directly or indirectly.
- Classes like ConcurrentLinkedQueue use atomic variables to directly implement wait-free algorithms, and classes like ConcurrentHashMap use ReentrantLock for locking where needed. ReentrantLock, in turn, uses atomic variables to maintain the queue of threads waiting for the lock.

Building Blocks for Highly Concurrent Design

```
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();
```

Building Blocks for Highly Concurrent Design

```
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; }
  }
}
```

Building Blocks for Highly Concurrent Design

- Performance considerations
  - Under light to moderate contention, nonblocking algorithms tend to outperform blocking ones because most of the time the CAS succeeds on the first try, and the penalty for contention when it does occur does not involve thread suspension and context switching, just a few more iterations of the loop.
  - An uncontended CAS is less expensive than an uncontended lock acquisition (this statement has to be true because an uncontended lock acquisition involves a CAS plus additional processing), and a contended CAS involves a shorter delay than a contended lock acquisition.

Building Blocks for Highly Concurrent Design

- Under high contention -- when many threads are pounding on a single memory location -- lock-based algorithms start to offer better throughput than nonblocking ones because when a thread blocks, it stops pounding and patiently waits its turn, avoiding further contention.
- However, contention levels this high are uncommon, as most of the time threads interleave thread-local computation with operations that contend for shared data, giving other threads a chance at the shared data. (Contention levels this high also indicate that reexamining your algorithm with an eye towards less shared data is in order.)

Building Blocks for Highly Concurrent Design

```
//Insertion in the Michael-Scott nonblocking queue algorithm
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 Node<E> dummy=new Node<E>(null,null);
private AtomicReference<Node<E>> head= new
     atomicReference<Node<E>>(dummy);
   private AtomicReference<Node<E>> tail = new
     AtomicReference<Node<E>>(dummy);
```

Building Blocks for Highly Concurrent Design

```
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 */;
                }
            }
        }
}
```

Building Blocks for Highly Concurrent Design

- A nonblocking linked list
  - The examples so far -- counter and stack -- are very simple nonblocking algorithms and are easy to follow once you grasp the pattern of using CAS in a loop. For more sophisticated data structures, nonblocking algorithms are much more complicated than these simple examples because modifying a linked list, tree, or hash table can involve updating more than one pointer.
  - CAS enables atomic conditional updates on a single pointer, but not on two. So to construct a nonblocking linked list, tree, or hash table, we need to find a way to update multiple pointers with CAS without leaving the data structure in an inconsistent state.

Building Blocks for Highly Concurrent Design

- Inserting an element at the tail of a linked list typically involves updating two pointers: the "tail" pointer that always refers to the last element in the list and the "next" pointer from the previous last element to the newly inserted element. Because two pointers need to be updated, two CASes are needed.
- Updating two pointers in separate CAS operations introduces two potential problems that need to be considered: what happens if the first CAS succeeds but the second fails, and what happens if another thread attempts to access the list between the first and second CAS.

Building Blocks for Highly Concurrent Design

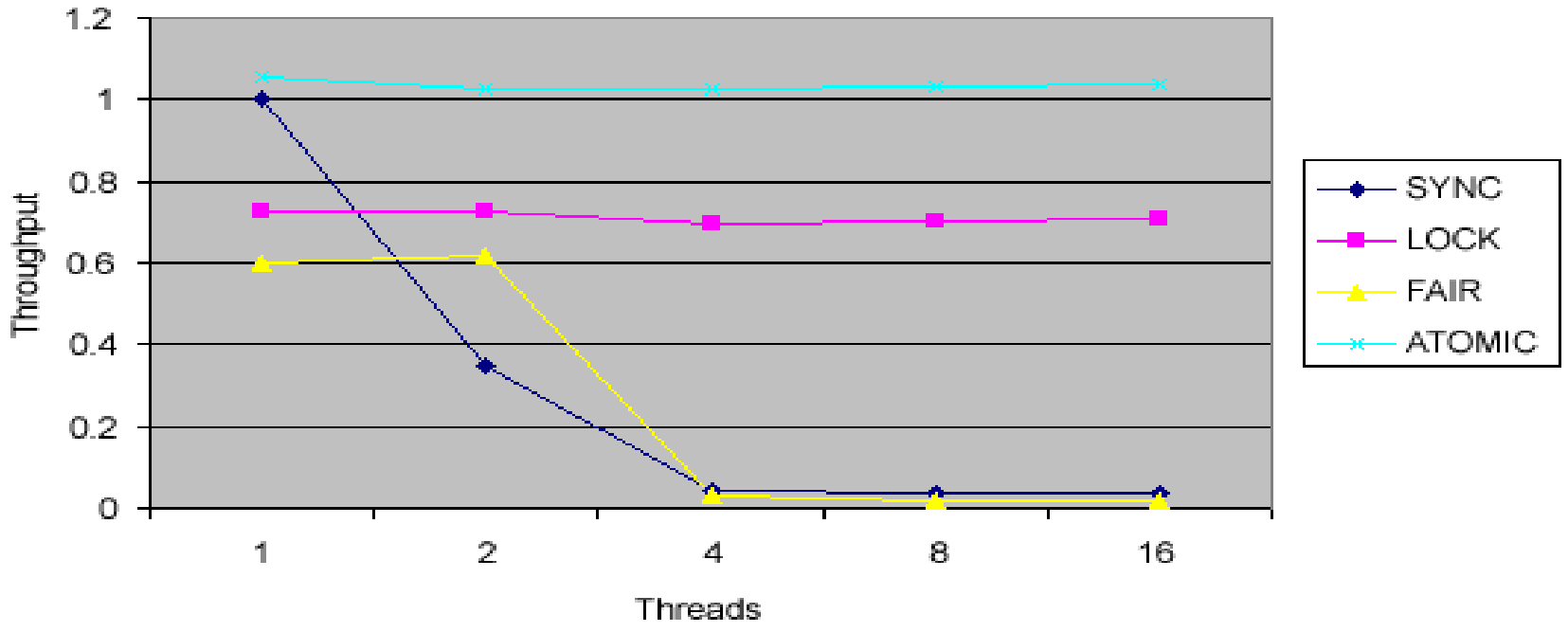
- The "trick" to building nonblocking algorithms for nontrivial data structures is to make sure that the data structure is always in a consistent state, even between the time that a thread starts modifying the data structure and the time it finishes, and to make sure that other threads can tell not only whether the first thread has finished its update or is still in the middle of it, but also what operations would be required to complete the update if the first thread went AWOL.
- If a thread arrives on the scene to find the data structure in the middle of an update, it can "help" the thread already performing the update by finishing the update for it, and then proceeding with its own operation. When the first thread gets around to trying to finish its own update, it will realize that the work is no longer necessary and just return because the CAS will detect the interference (in this case, constructive interference) from the helping thread.

Building Blocks for Highly Concurrent Design

- This "help thy neighbor" requirement is needed to make the data structure resistant to the failure of individual threads. If a thread arrived to find the data structure in mid-update by another thread and just waited until that thread finished its update, it could wait forever if the other thread fails in the middle of its operation.
- Even in the absence of failure, this approach would offer poor performance because the newly arriving thread would have to yield the processor, incurring a context switch, or wait for its quantum to expire, which is even worse.

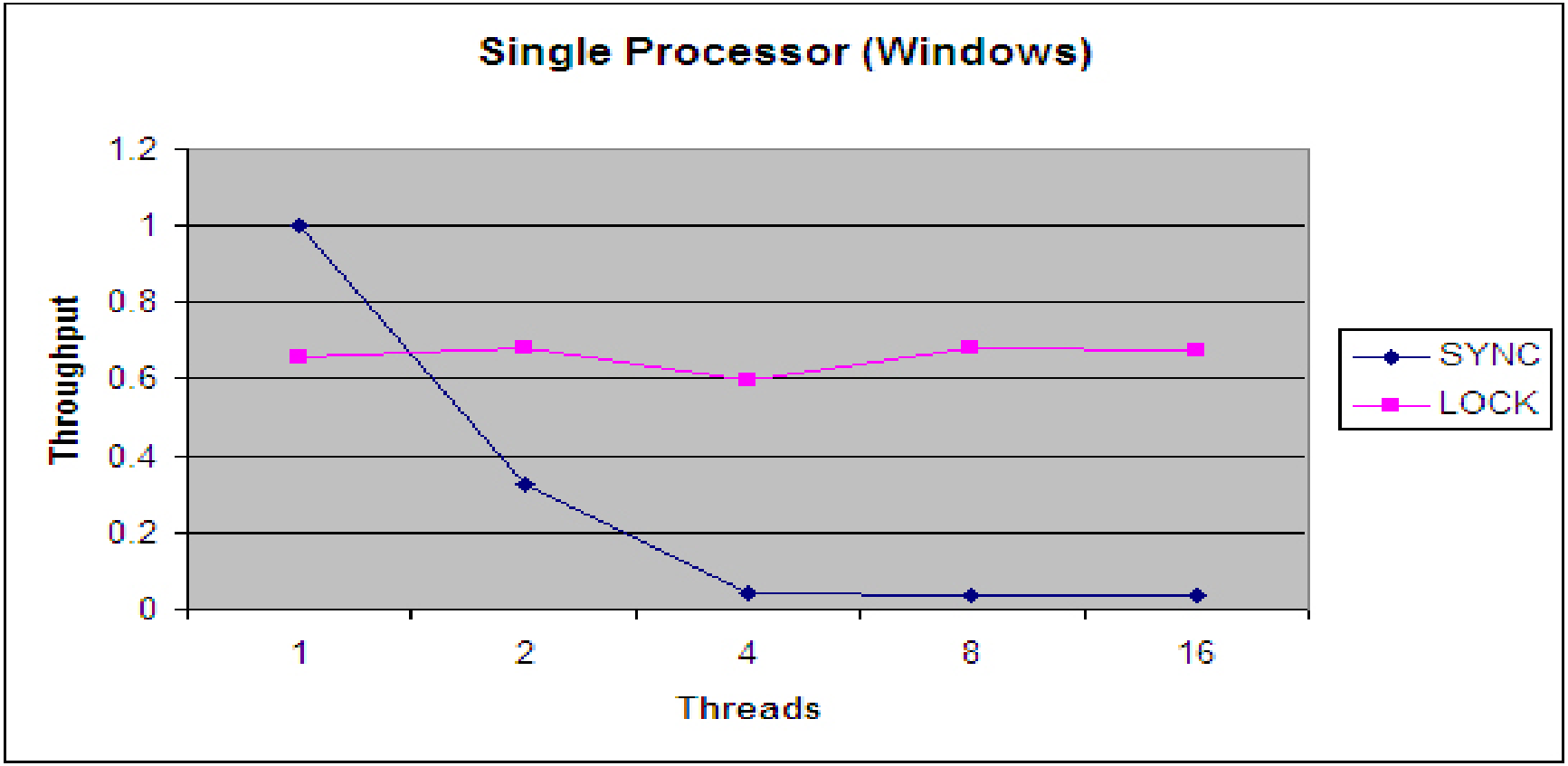
Building Blocks for Highly Concurrent Design

- Benchmark throughput for synchronization, ReentrantLock, fair Lock, and AtomicLong on a single-processor Pentium 4



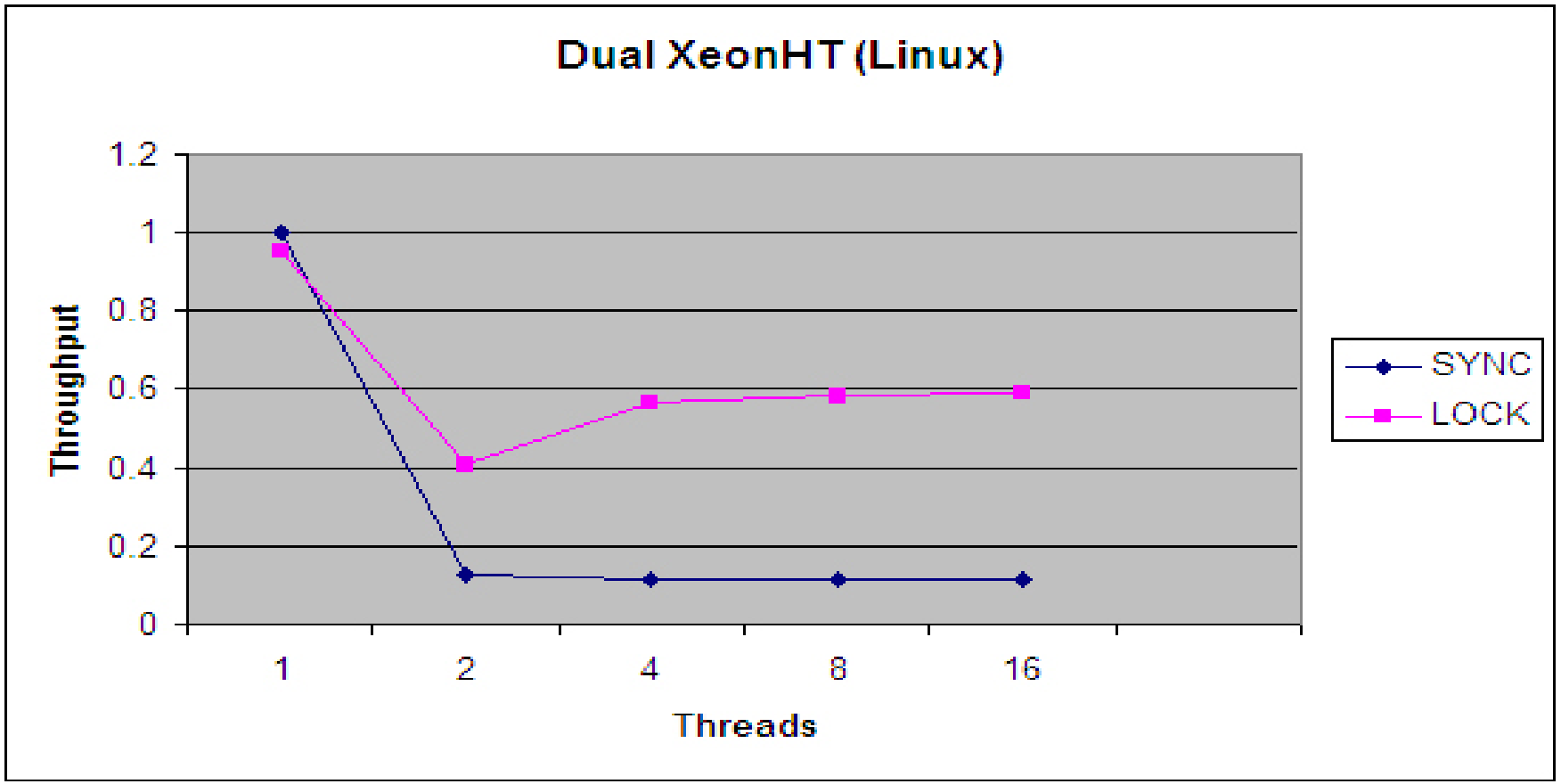
Building Blocks for Highly Concurrent Design

- Figure 2. Throughput (normalized) for synchronization and Lock, four CPUs



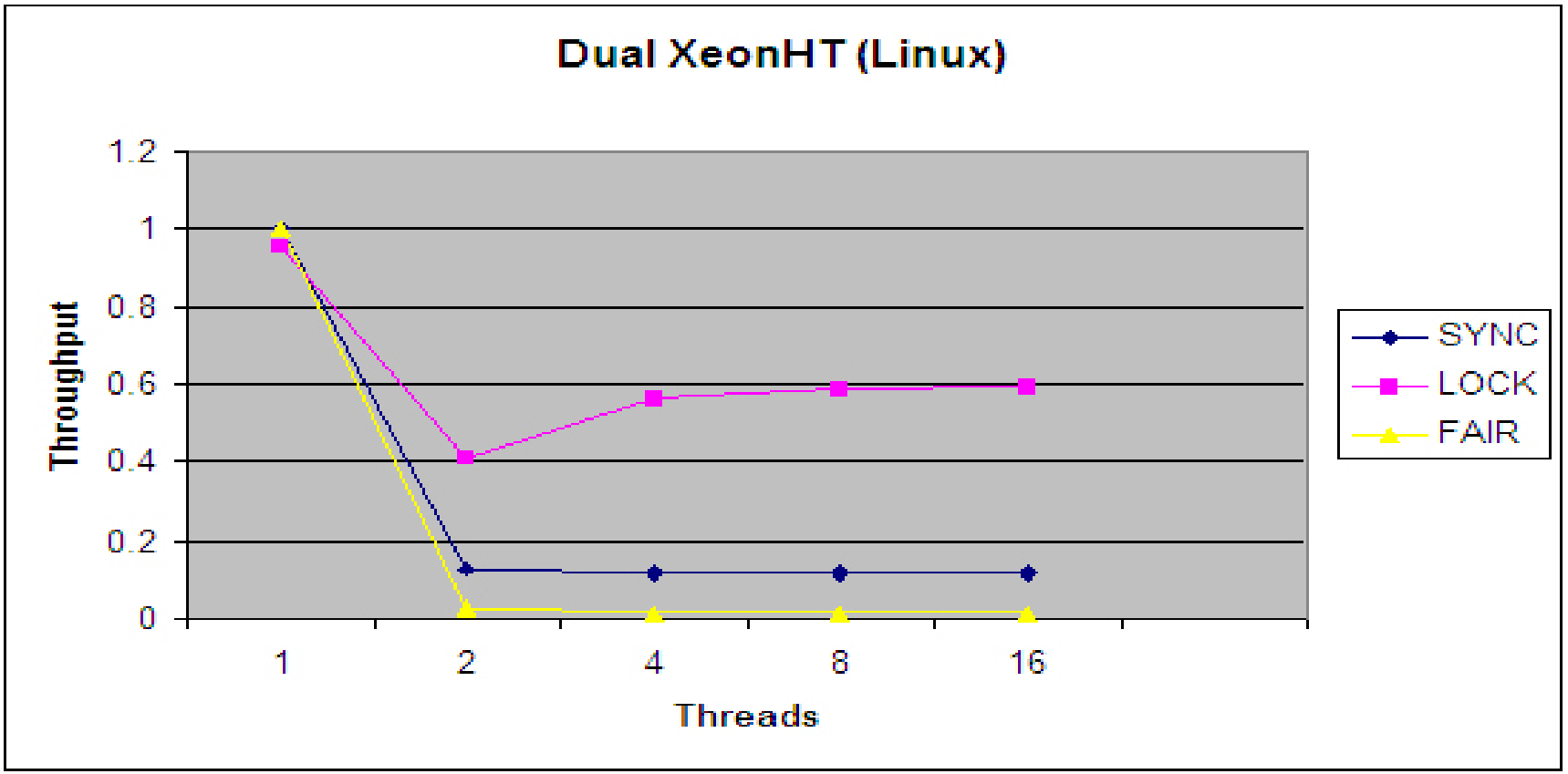
Building Blocks for Highly Concurrent Design

- Throughput for synchronization and Lock, single CPU



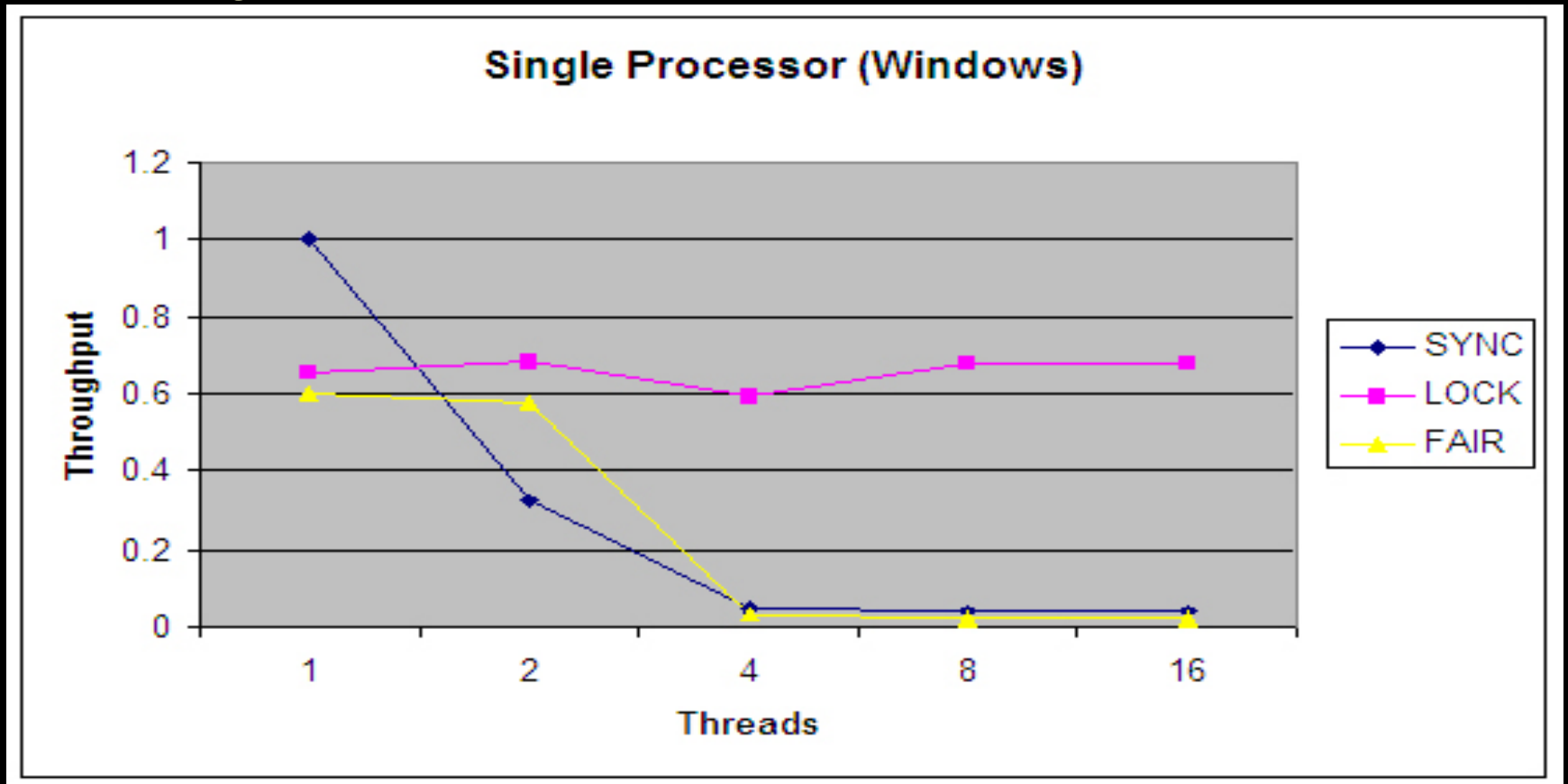
Building Blocks for Highly Concurrent Design

- Relative throughput for synchronization, barging Lock, and fair Lock, with four CPUs



Building Blocks for Highly Concurrent Design

- Relative throughput for synchronization, barging Lock, and fair Lock, with single CPU



Locks

- Why not the “synchronized” keyword.
  - Well for most times the synchronized keyword is an easy way out, but because of the abstraction it provides it has certain limitations.
  - Among them are:
    - You cannot interrupt a thread that is trying to acquire a lock.
    - You cannot specify a timeout when trying to acquire a lock.
    - Having a single condition per lock can be inefficient.
    - The virtual machine locking primitives do not map well to the most efficient locking mechanisms available in hardware.

Locks

- The Lock interface
  - ***java.util.concurrent.locks.Lock***
    - boolean trylock()tries to acquire the lock without blocking; returns true if it was successful. This method grabs the lock if it is available even if it has a fair locking policy and other threads have been waiting.
    - boolean tryLock(long time, TimeUnit unit) tries to acquire the lock, blocking no longer than the given time; returns true if it was successful.
    - void lockInterruptibly() acquires the lock, blocking indefinitely. If the thread is interrupted, throws an InterruptedException.

Locks

- void **lock**() Acquires the lock. If the lock is not available then the current thread becomes disabled for thread scheduling purposes and lies dormant until the lock has been acquired.
- void **unlock**() Releases the lock.

Locks

- java.util.concurrent. locks.Condition
  - void await() Causes the current thread to wait until it is signalled or interrupted.
  - boolean await(long time, TimeUnit unit) enters the wait set for this condition, blocking until the thread is removed from the wait set or the given time has elapsed. Returns false if the method returned because the time elapsed, true otherwise.
  - void awaitUninterruptibly() enters the wait set for this condition, blocking until the thread is removed from the wait set. If the thread is interrupted, this method does not throw an InterruptedException. If the current thread's interrupted status is set when it enters this method, or it is interrupted while waiting, it will continue to wait until signalled. When it finally returns from this method its interrupted status will still be set.

Locks

- boolean awaitUntil(Date deadline) Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.
- long awaitNanos(long nanosTimeout)
  Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
- void signal()
  Wakes up one waiting thread.
- void signalAll()
  Wakes up all waiting threads.

Locks

- Declaration

```
class Bank
{
    public Bank(int n, double initialBalance)
    {
    bankLock = new ReentrantLock();
    sufficientFunds = bankLock.newCondition();
    }
    private Lock bankLock;
    private Condition sufficientFunds;
    // other methods
}
```

Locks

- Putting them to use

```
public void transfer(int from, int to, double amount)
    throws InterruptedException
    {
            bankLock.lock();
            try
            {
                    while (accounts[from] < amount)
                    sufficientFunds.await();
                    accounts[from] -= amount;
                    accounts[to] += amount;
                    sufficientFunds.signalAll();
            }
            finally{ bankLock.unlock();}
  }
```

Locks-ReentrantLock

- Java.util.concurrent.locks.ReentrantLock
  - The implementation class has got a lot of utility methods to find out the state of the lock the threads holding the lock.
  - For e.g. the thread holding the lock, the threads waiting on the lock's queue , the threads waiting on the condition variables of the lock etc, etc.
  - If you want they support fairness…..????? But is that OK?????????

ReadWriteLock

- A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive.
- A read-write lock allows for a greater level of concurrency in accessing shared data than that permitted by a mutual exclusion lock. It exploits the fact that while only a single thread at a time (a *writer* thread) can modify the shared data, in many cases any number of threads can concurrently read the data (hence *reader* threads).
- In theory, the increase in concurrency permitted by the use of a read-write lock will lead to performance improvements over the use of a mutual exclusion lock.
- In practice this increase in concurrency will only be fully realized on a multi-processor, and then only if the access patterns for the shared data are suitable.

ReadWriteLock

- Whether or not a read-write lock will improve performance over the use of a mutual exclusion lock depends on the frequency that the data is read compared to being modified, the duration of the read and write operations, and the contention for the data - that is, the number of threads that will try to read or write the data at the same time. For example, a collection that is initially populated with data and thereafter infrequently modified, while being frequently searched (such as a directory of some kind) is an ideal candidate for the use of a read-write lock.

ReadWriteLock

- However, if updates become frequent then the data spends most of its time being exclusively locked and there is little, if any increase in concurrency. Further, if the read operations are too short the overhead of the read-write lock implementation (which is inherently more complex than a mutual exclusion lock) can dominate the execution cost, particularly as many read-write lock implementations still serialize all threads through a small section of code. Ultimately, only profiling and measurement will establish whether the use of a read-write lock is suitable for your application.

ReadWriteLock

- Java.util.concurrent.locks.ReadWriteLock
  - Lock readLock() Returns the lock used for reading.
  - Lock writeLock() Returns the lock used for writing.

ReadWriteLock

- Put them to use

```
class CachedData {
    Object data;
    volatile boolean cacheValid;
    ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

    void processCachedData() {
        rwl.readLock().lock();
        if (!cacheValid) {
            // upgrade lock manually
            rwl.readLock().unlock();   // must unlock first to obtain writelock
            rwl.writeLock().lock();
            if (!cacheValid) { // recheck
            data = ...
            cacheValid = true;
    }
```

ReadWriteLock

```
// downgrade lock
        rwl.readLock().lock();  // reacquire read without giving up write lock
        rwl.writeLock().unlock(); // unlock write, still hold read
       }

    use(data);
    rwl.readLock().unlock();
      }
 }
```

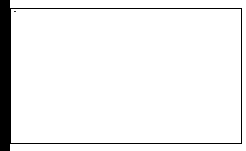
ReadWriteLock-ReentrantReadWriteLock

- An implementation of ReadWriteLock supporting similar semantics to ReentrantLock.
- This class has the following properties:
  - **Acquisition order** This class does not impose a reader or writer preference ordering for lock access. However, it does support an optional *fairness* policy. When constructed as fair, threads contend for entry using an approximately arrival-order policy.
  - **Reentrancy** This lock allows both readers and writers to reacquire read or write locks in the style of a ReentrantLock. Readers are not allowed until all write locks held by the writing thread have been released.
  - Additionally, a writer can acquire the read lock - but not vice-versa. Among other applications, reentrancy can be useful when write locks are held during calls or callbacks to methods that perform reads under read locks. If a reader tries to acquire the write lock it will never succeed.

ReadWriteLock-ReentrantReadWriteLock

- **Lock downgrading** Reentrancy also allows downgrading from the write lock to a read lock, by acquiring the write lock, then the read lock and then releasing the write lock. However, upgrading from a read lock to the write lock is **not** possible.
- **Interruption of lock acquisition** The read lock and write lock both support interruption during lock acquisition.
- **Condition support** The write lock provides a Condition implementation that behaves in the same way, with respect to the write lock, as the Condition implementation provided by ReentrantLock.newCondition() does for ReentrantLock. This Condition can, of course, only be used with the write lock.
- The read lock does not support a Condition and readLock().newCondition() throws UnsupportedOperationException.
- **Instrumentation** This class supports methods to determine whether locks are held or contended. These methods are designed for monitoring system state, not for synchronization control.

The Hierarchy



Locks

- While ReentrantLock is a very impressive implementation and has some significant advantages over synchronization, I believe the rush to consider synchronization a deprecated feature to be a serious mistake
- *The locking classes in java.util.concurrent.lock are advanced tools for advanced users and situations*. In general, you should probably stick with synchronization unless you have a specific need for one of the advanced features of Lock, or if you have demonstrated evidence (not just a mere suspicion) that synchronization in this particular situation is a scalability bottleneck.
- Why do I advocate such conservatism in adopting a clearly "better" implementation? Synchronization still has a few advantages over the locking classes in java.util.concurrent.lock.
  - For one, it is impossible to forget to release a lock when using synchronization; the JVM does that for you when you exit the synchronized block.

Locks

- Another reason is because when the JVM manages lock acquisition and release using synchronization, the JVM is able to include locking information when generating thread dumps. These can be invaluable for debugging, as they can identify the source of deadlocks or other unexpected behaviors. The Lock classes are just ordinary classes, and the JVM does not (yet) have any knowledge of which Lock objects are owned by specific threads.

Locks

- **When to select ReentrantLock over synchronized**
  - So, when should we use ReentrantLock?
  - The answer is pretty simple -- use it when you actually need something it provides that synchronized doesn't, like timed lock waits, interruptible lock waits, non-block-structured locks, multiple condition variables, or lock polling.
  - ReentrantLock also has scalability benefits, and you should use it if you actually have a situation that exhibits high contention, but remember that the vast majority of synchronized blocks hardly ever exhibit any contention, let alone high contention.
  - Lastly synchronized is a language primitive and the JVM authors may “reimplement” it using CAS. In fact it has been done in JDK 6.0(Mustang).

Highly Concurrent Data-Structures

- **ConcurrentHashMap**, part of Doug Lea's util.concurrent package, offers a higher degree of concurrency than Hashtable or synchronizedMap.
- In addition, it manages to avoid locking completely for most successful get() operations, which results in excellent throughput in concurrent applications.
- ConcurrentHashMap uses several tricks to achieve a high level of concurrency and avoid locking, including using multiple write locks for different hash buckets and exploiting the properties of the JMM to minimize the time that locks are held -- or avoid acquiring locks at all.
- It is optimized for the most common usage, which is retrieving a value likely to already exist in the map. In fact, most successful get() operations will run without any locking at all. **(Warning: don't try this at home! Trying to outsmart the JMM is much harder than it looks and is not to be undertaken lightly.**

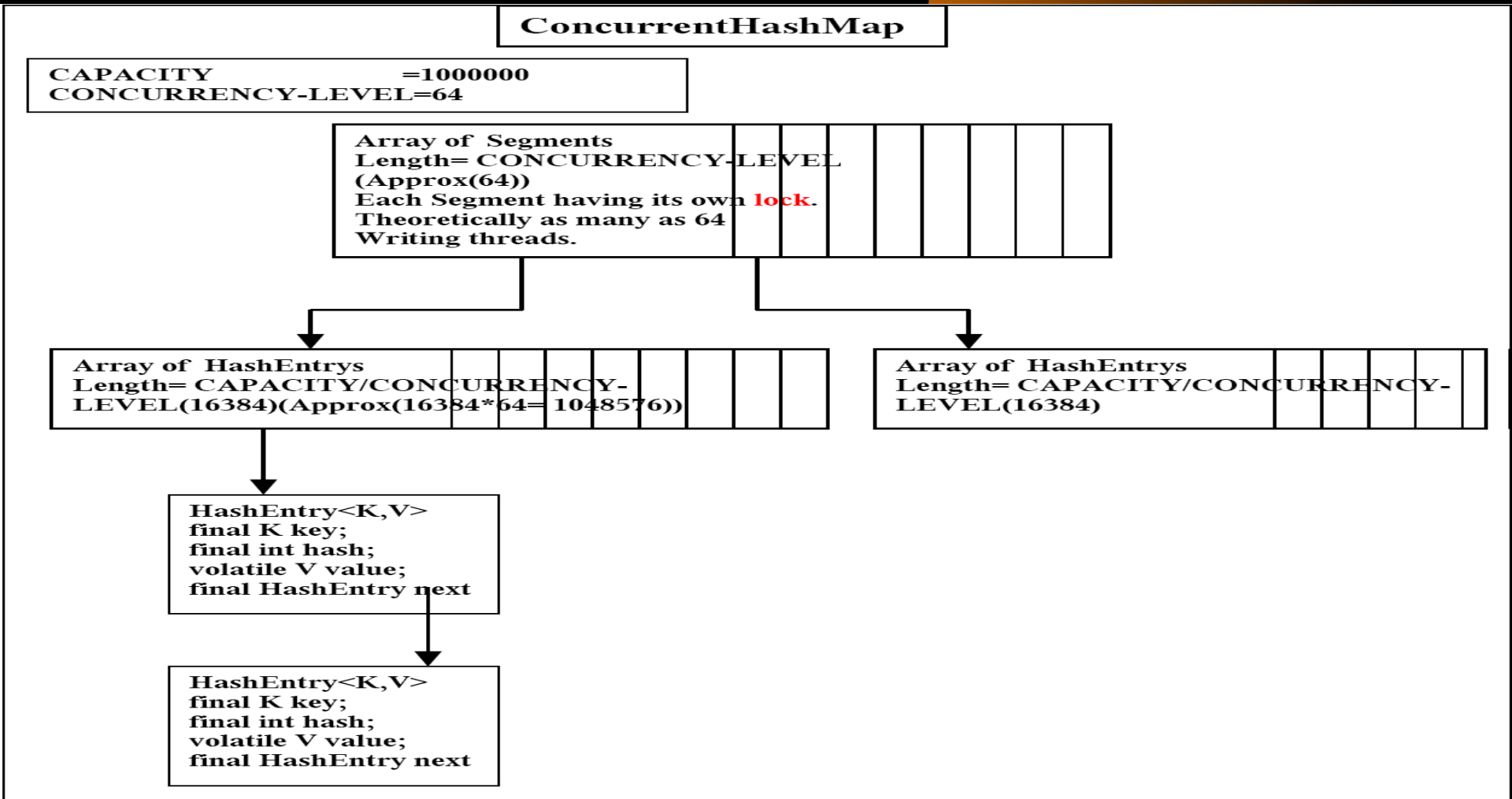
Highly Concurrent Data-Structures

- Trick1
  - ConcurrentHashMap dispenses with the single map-wide lock, instead using a collection of CONCURRENCY-LEVEL number of locks(which you could specify), each of which guards a subset of the hash buckets. Locks are primarily used by mutative (put() and remove()) operations.
  - Having CONCURRENCY-LEVEL separate locks means that a maximum of CONCURRENCY-LEVEL threads can be modifying the map at once.
  - This doesn't necessarily mean that if there are fewer than CONCURRENCY-LEVEL threads writing to the map currently, that another write operation will not block -- CONCURRENCY-LEVEL is the theoretical concurrency limit for writers, but may not always be achieved in practice.

Highly Concurrent Data-Structures

- Still, CONCURRENCY-LEVEL is a lot better than one and should be more than adequate for most applications running on the current generation of computer systems.

Highly Concurrent Data-Structures



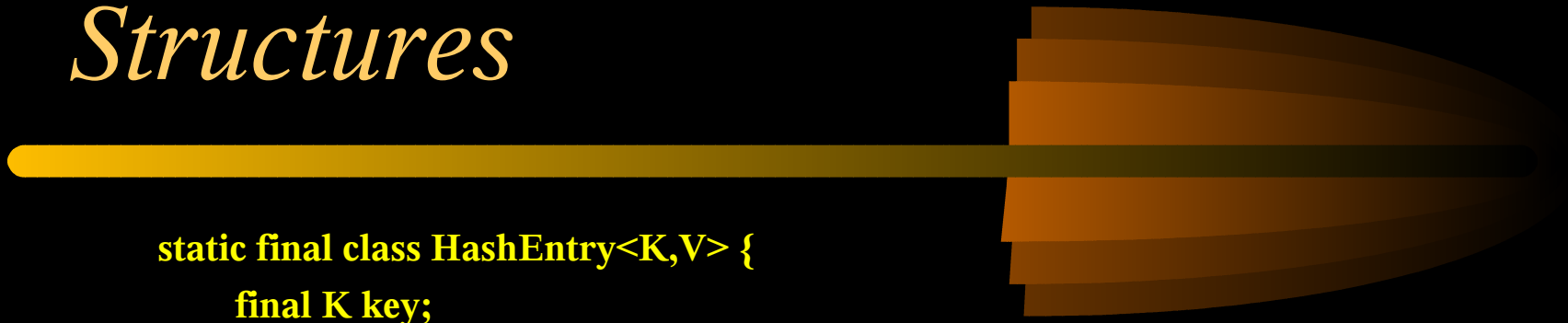
Highly Concurrent Data-Structures

- Trick2
  - **Exploiting immutability**
  - One significant source of inconsistency is avoided by making the Entry elements nearly immutable -- all fields are final, except for the value field, which is volatile.
  - This means that elements cannot be added to or removed from the middle or end of the hash chain -- elements can only be added at the beginning, and removal involves cloning all or part of the chain and updating the list head pointer.
  - So once you have a reference into a hash chain, while you may not know whether you have a reference to the head of the list, you do know that the rest of the list will not change its structure.

Highly Concurrent Data-Structures

- Also, since the value field is volatile, you will be able to see updates to the value field immediately, greatly simplifying the process of writing a Map implementation that can deal with a potentially stale view of memory.
- While the new JMM provides initialization safety for final variables, the old JMM does not, which means that it is possible for another thread to see the default value for a final field, rather than the value placed there by the object's constructor.
- The implementation must be prepared to detect this as well, which it does by ensuring that the default value for each field of Entry is not a valid value. The list is constructed such that if any of the Entry fields appear to have their default value (zero or null), the search will fail, prompting the get() implementation to synchronize and traverse the chain again.

Highly Concurrent Data-Structures



```
static final class HashEntry<K,V> {
        final K key;
        final int hash;
        volatile V value;
        final HashEntry<K,V> next;
      //remainder omited
}
```

Highly Concurrent Data-Structures

- **Retrieval operations**
- Retrieval operations proceed by first finding the head pointer for the desired bucket (which is done without locking, so it could be stale), and traversing the bucket chain without acquiring the lock for that bucket.
- If it doesn't find the value it is looking for, it synchronizes and tries to find the entry again, as shown below:
- Immutability means that get succeeds without locking in common code path.

Highly Concurrent Data-Structures

```
//get operation
      public Object get(Object key) {
            int hash = hash(key);     // throws null pointer exception if key is null
            // Try first without locking...
            HashEntry[] tab = table;
            int index = hash & (tab.length - 1);
            HashEntry first = tab[index];
            HashEntry e;
            for (e = first; e != null; e = e.next) {
                  if (e.hash == hash && eq(key, e.key)) {
                        Object value = e.value;
                        // null values means that the element has been removed

                        if (value != null)
                              return value;
                        else        break;
                  }
            }
```

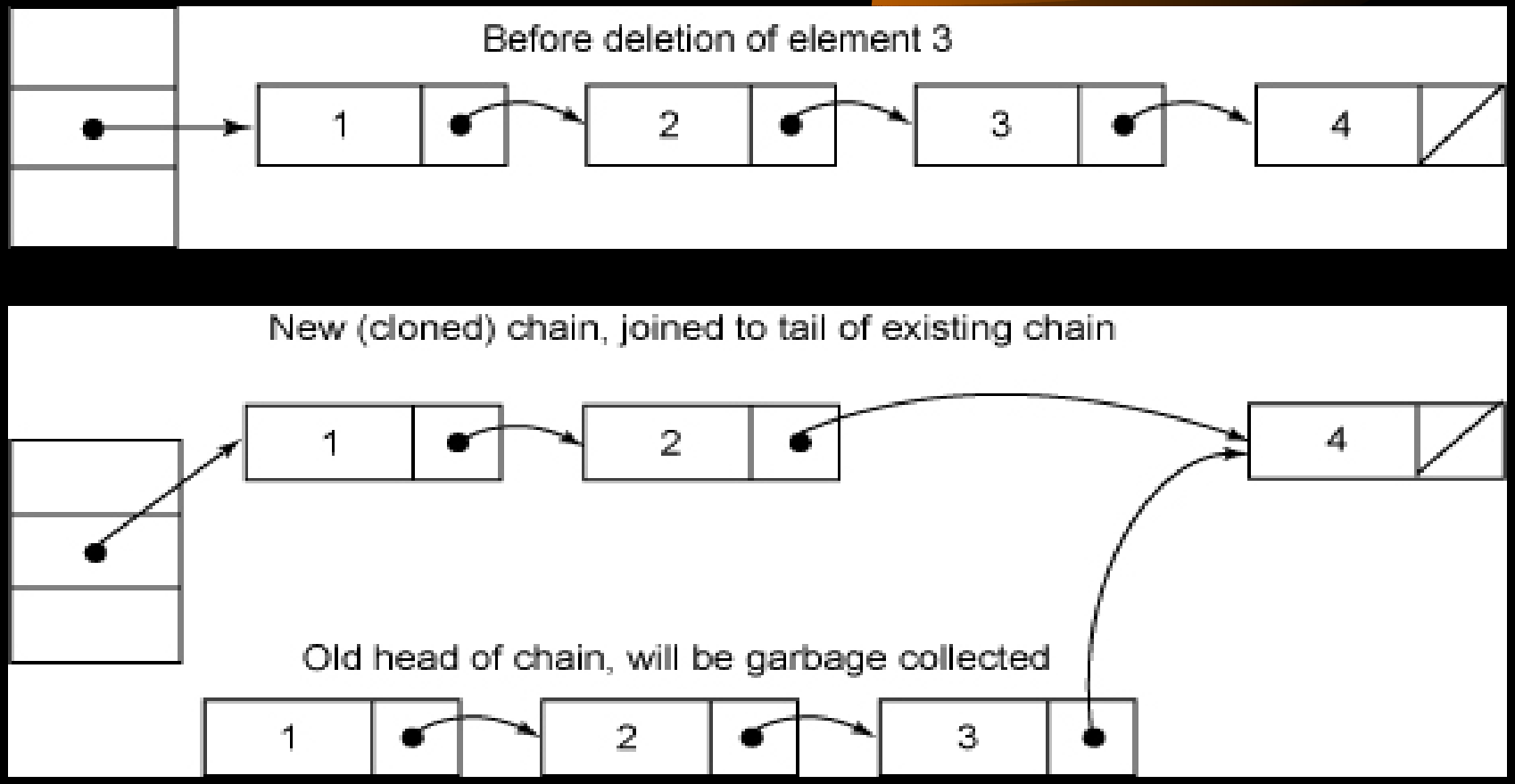
Highly Concurrent Data-Structures

```
        // Recheck under synch if key apparently not there
        //or interference (another thread in the same bucket)
        Segment seg = segments[hash & SEGMENT_MASK];
synchronized(seg) {
                tab = table;
                index = hash & (tab.length - 1);
                HashEntry newFirst = tab[index];
                if (e != null || first != newFirst) {
                        for (e = newFirst; e != null; e = e.next) {
                                if (e.hash == hash && eq(key, e.key))
                                        return e.value;
                        }
                }
                return null;
        }
}
```

Highly Concurrent Data-Structures

- **Removal operations**
- Because a thread could see stale values for the link pointers(HashEntry) in a hash chain, simply removing an element from the chain would not be sufficient to ensure that other threads will not continue to see the removed value when performing a lookup.
- Instead,, removal is a two-step process -- first the appropriate Entry object is found and its value field is set to null, and then the portion of the chain from the head to the removed element is cloned and joined to the remainder of the chain following the removed element.
- Since the value field is volatile, if another thread is traversing the stale chain looking for the removed element, it will see the null value field immediately, and know to retry the retrieval with synchronization. Eventually, the initial part of the hash chain ending in the removed element will be garbage collected.

Highly Concurrent Data-Structures



Highly Concurrent Data-Structures

```
protected Object remove(Object key, Object value) {
    /*      Find the entry, then
            1. Set value field to null, to force get() to retry
            2. Rebuild the list without this entry.
            All entries following removed node can stay in list, but all preceding ones
    need to be cloned.  Traversals rely on this strategy to ensure that elements
    will not be repeated during iteration.
    */
    int hash = hash(key);
   Segment seg = segments[hash & SEGMENT_MASK];
    synchronized(seg) {
            Entry[] tab = table;
            int index = hash & (tab.length-1);
            Entry first = tab[index];
            Entry e = first;
```

Highly Concurrent Data-Structures

```
for (;;) {
        if (e == null) return null;
        if (e.hash == hash && eq(key, e.key))
        break;
        e = e.next;
}
Object oldValue = e.value;
if (value != null && !value.equals(oldValue))
return null;
e.value = null; //a concurrent get will see the null and retry under
//synchronization..and will block because the segment is under lock.
Entry head = e.next;
for (Entry p = first; p != e; p = p.next)
head = new Entry(p.hash, p.key, p.value, head);
tab[index] = head;      seg.count--;      return oldValue;
    }
}
```

Highly Concurrent Data-Structures

- **No locking?**
  - To say that successful get() operations proceed without locking is a bit of an overstatement, as the value field of Entry is volatile, and this is used to detect updates and removals.
  - At the machine level, volatile and synchronization often end up getting translated into the same cache coherency primitives, so there effectively is *some* locking going on here, albeit at a finer granularity and without the scheduling or JVM overhead of acquiring and releasing monitors.
  - But, semantics aside, the concurrency achieved by ConcurrentHashMap in many common situations, where retrievals outnumber insertions and removals, is quite impressive.

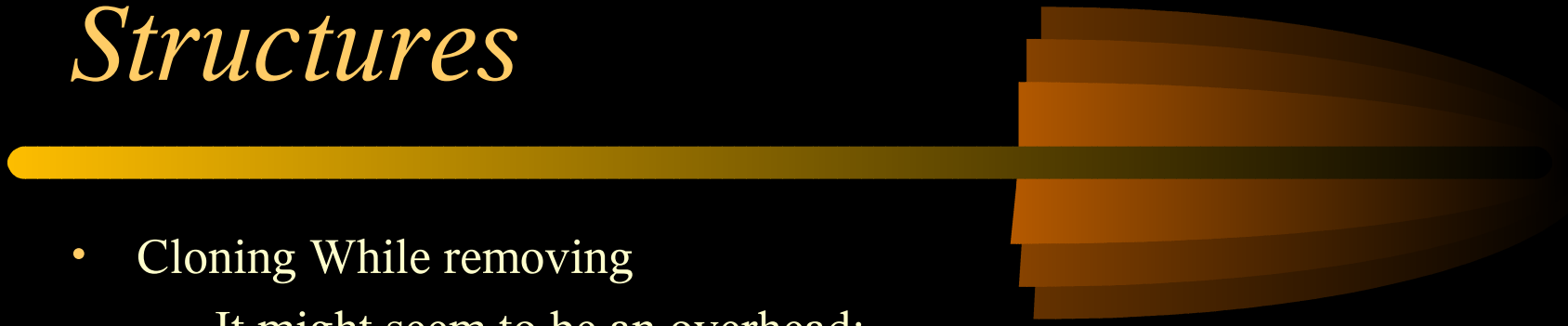
Highly Concurrent Data-Structures

- **Weakly consistent iterators**
  - The semantics of iterators returned by ConcurrentHashMap differ from those of java.util collections; rather than being *fail-fast* , they are instead *weakly consistent.*
  - When a user calls keySet().iterator() to retrieve an iterator for the set of hash keys, the implementation briefly synchronizes to make sure the head pointers for each chain are current. The next() and hasNext() operations are defined in the obvious way, traversing each hash chain and then moving on to the next chain until all the chains have been traversed.
  - Weakly consistent iterators may or may not reflect insertions made during iteration, but they will reflect updates or removals for keys that have not yet been reached by the iterator, and will not return any value more than once.The iterators returned by ConcurrentHashMap will not throw ConcurrentModificationException.

Highly Concurrent Data-Structures

- **Dynamic resizing**
  - As the number of elements in the map grows, the hash chains will grow longer and retrieval time will increase. At some point, it makes sense to increase the number of buckets and rehash the values.
  - In classes like Hashtable, this is easy because it is possible to hold an exclusive lock on the entire map. In ConcurrentHashMap, each time an entry is inserted, if the length of that chain exceeds some threshold, that chain is marked as needing to be resized.
  - When enough chains vote that resizing is needed, ConcurrentHashMap will use recursion to acquire the lock on each bucket and rehash the elements from each bucket into a new, larger hash table. Most of the time, this will occur automatically and transparently to the caller

Highly Concurrent Data-Structures



- Cloning While removing
  - It might seem to be an overhead:
    - But only the HashEntrys are cloned not the objects and the size of the array is chosen such that with the default load factor the number of links cant be more that 1.5 on an average.
    - Which means max(worst case) one object to be cloned.

Highly Concurrent Data-Structures

Table 1. Scalability comparison between HashMap, LockPoolMap and ConcurrentHashMap,

Threads	ConcurrentHashMap	Unsynchronized HashMap (unsafe)	Synchronized HashMap	LockPoolMap
1	1.00	1.1	1.4	1.6
2	2.09	1.1	57.6	3.7
4	3.00	2.1	123.5	7.7
8	7.21	3.7	272.3	16.7
16	12.58	6.8	577.0	37.9
32	27.59	13.5	1233.3	80.5

Highly Concurrent Data-Structures

- Conclusion
  - ConcurrentHashMap is both a very useful class for many concurrent applications and a fine example of a class that understands and exploits the subtle details of the JMM to achieve higher performance. ConcurrentHashMap is an impressive feat of coding, one that requires a deep understanding of concurrency and the JMM. Use it, learn from it, enjoy it

Highly Concurrent Data-Structures

- ConcurrentLinkedQueue
  - Again uses the non-blocking algorithms like CAS.

```
public class ConcurrentLinkedQueue<E>{
private static final
        AtomicReferenceFieldUpdater<ConcurrentLinkedQueue, Node>
        tailUpdater = //OMITTED……….
private static final
        AtomicReferenceFieldUpdater<ConcurrentLinkedQueue, Node>
        headUpdater = //OMITTED……….
   private boolean casTail(Node<E> cmp, Node<E> val) {
        return tailUpdater.compareAndSet(this, cmp, val);
    }

    private boolean casHead(Node<E> cmp, Node<E> val) {
        return headUpdater.compareAndSet(this, cmp, val);
    }
}
```

Highly Concurrent Data-Structures

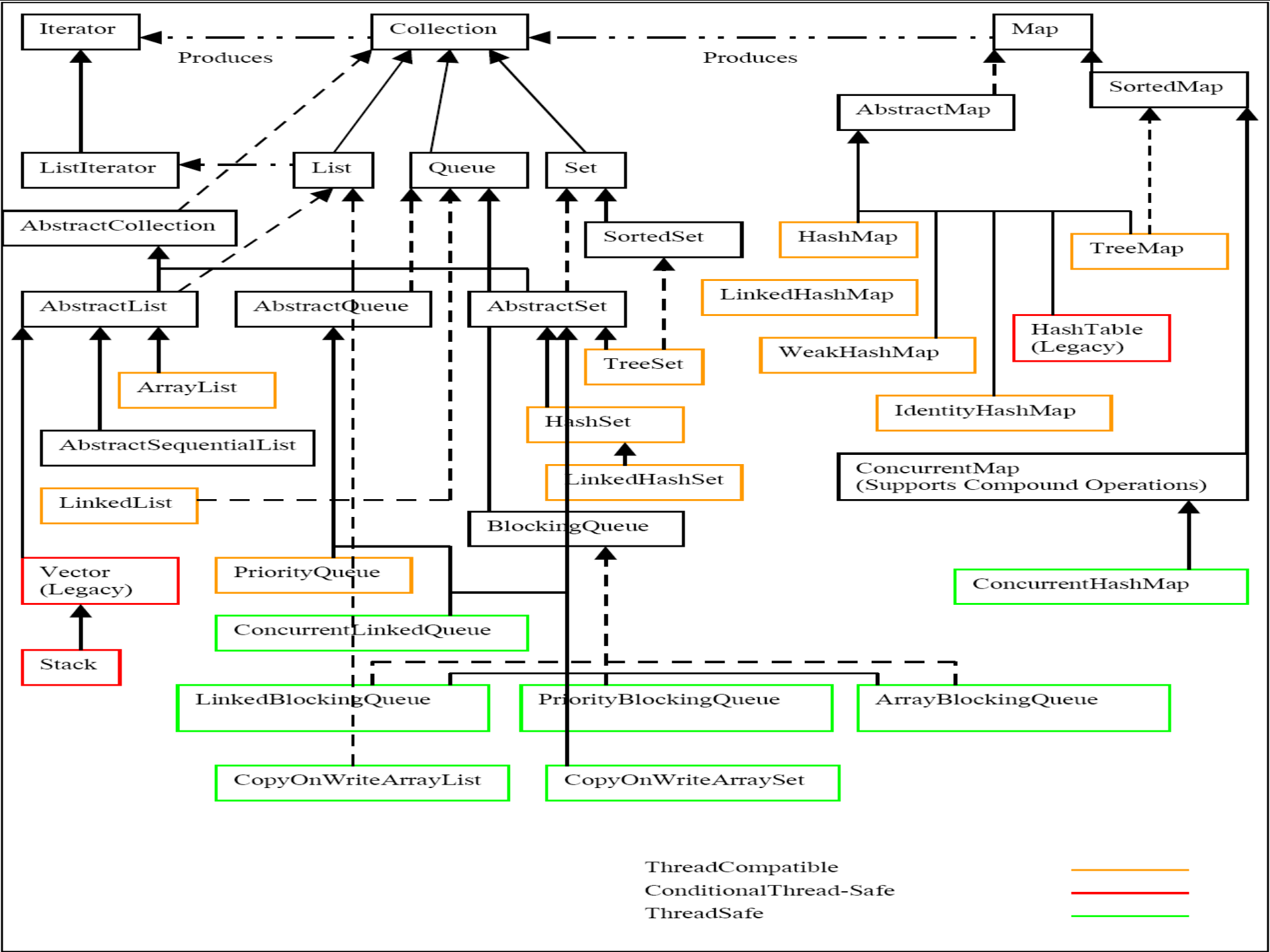
```
public boolean offer(E o) {
    Node<E> n = new Node<E>(o, null);
     for(;;) {
        Node<E> t = tail;
        Node<E> s = t.getNext();
        if (t == tail) {
           if (s == null) {
              if (t.casNext(s, n)) {
                 casTail(t, n);//non blocking calls to CAS classes..
                 return true;
              }
           } else {
              casTail(t, s);
           }
        }
     }
     //rest ommited
}
```

Highly Concurrent Data-Structures

- CopyOnWriteArrayList
  - The CopyOnWriteArrayList class is intended as a replacement for ArrayList in concurrent applications where traversals greatly outnumber insertions or removals.
  - This is quite common when ArrayList is used to store a list of listeners, such as in AWT or Swing applications, or in JavaBean classes in general. (The related CopyOnWriteArraySet uses a CopyOnWriteArrayList to implement the Set interface.)
  - If you are using an ordinary ArrayList to store a list of listeners, as long as the list remains mutable and may be accessed by multiple threads, you must either lock the entire list during iteration or clone it before iteration, both of which have a significant cost.

Highly Concurrent Data-Structures

- CopyOnWriteArrayList instead creates a fresh copy of the list whenever a mutative operation is performed, and its iterators are guaranteed to return the state of the list at the time the iterator was constructed and not throw a ConcurrentModificationException.
- It is not necessary to clone the list before iteration or lock it during iteration because the copy of the list that the iterator sees will not change.
- In other words, CopyOnWriteArrayList contains a mutable reference to an immutable array, so as long as that reference is held fixed, you get all the thread-safety benefits of immutability without the need for locking.


Iterator
Collection
Map
Produces
Produces
SortedMap
AbstractMap
ListIterator
List
Queue
Set
AbstractCollection
SortedSet
HashMap
TreeMap
LinkedHashMap
AbstractList
AbstractQueue
AbstractSet
HashTable
(Legacy)
WeakHashMap
TreeSet
ArrayList
IdentityHashMap
HashSet
AbstractSequentialList
ConcurrentMap
(Supports Compound Operations)
LinkedHashSet
LinkedList
BlockingQueue
Vector
(Legacy)
PriorityQueue
ConcurrentHashMap
ConcurrentLinkedQueue
Stack
LinkedBlockingQueue
PriorityBlockingQueue
ArrayBlockingQueue
CopyOnWriteArrayList
CopyOnWriteArraySet
ThreadCompatible
ConditionalThread-Safe
ThreadSafe

"Canned" Synchronizers

- The java.util.concurrent package contains several classes that help manage a set of collaborating threads.
- These mechanisms have "canned functionality" for common rendezvous patterns between threads.
- If you have a set of collaborating threads that follows one of these behavior patterns, you should simply reuse the appropriate library class instead of trying to come up with a handcrafted collection of locks.

"Canned" Synchronizers

Class	What It Does	When To Use
`CyclicBarrier`	Allows a set of threads to wait until a predefined count of them has reached a common barrier, and then optionally executes a barrier action.	When a number of threads need to complete before their results can be used.
`CountDownLatch`	Allows a set of threads to wait until a count has been decremented to 0.	When one or more threads need to wait until a specified number of results are available.
`Exchanger`	Allows two threads to exchange objects when both are ready for the exchange.	When two threads work on two instances of the same data structure, one by filling an instance and the other by emptying the other.
`SynchronousQueue`	Allows a thread to hand off an object to another thread.	To send an object from one thread to another when both are ready, without explicit synchronization.
`Semaphore`	Allows a set of threads to wait until permits are available for proceeding.	To restrict the total number of threads that can access a resource. If permit count is one, use to block threads until another thread gives permission.

"Canned" Synchronizers

- CyclicBarrier
  - The CyclicBarrier class implements a rendezvous called a *barrier*. Consider a number of threads that are working on parts of a computation.
  - When all parts are ready, the results need to be combined. When a thread is done with its part, we let it run against the barrier.
  - Once all threads have reached the barrier, the barrier gives way and the threads can proceed.
  - If any of the threads waiting for the barrier leaves the barrier, then the barrier *breaks. (A* thread can leave because it called await with a timeout or because it was interrupted.) In that case, the await method for all other threads throws a BrokenBarrierException. Threads that are already waiting have their await call terminated immediately.

"Canned" Synchronizers

- CountDownLatches
  - A useful special case is a latch with a count of 1. "this implements a one-time *gate*. Threads are held at the gate until another thread sets the count to 0. Imagine, for example, a set of threads that need some initial data to do their work. The worker threads are started and wait at the gate. Another thread prepares the data. When it is ready, it calls countDown, and all worker threads proceed.
  - A CountpownLatch lets a set of threads wait until a count has reached zero. It differs from a barrier in these respects:
    - Not all threads need to wait for the latch until it can be opened.
    - The latch can be counted down by external events.
    - The countdown latch is one-time only. Once the count has reached 0, you cannot reuse it.

"Canned" Synchronizers

- ***Exchangers***
  - An Exchanger is used when two threads are working on two instances of the same data buffer. Typically, one thread fills the buffer, and the other consumes its contents. When both are done, they exchange their buffers.

"Canned" Synchronizers

```
class FillAndEmpty {
    Exchanger<DataBuffer> exchanger = new Exchanger();
    DataBuffer initialEmptyBuffer = ... a made-up type
    DataBuffer initialFullBuffer = ...
    class FillingLoop implements Runnable {
        public void run() {
            DataBuffer currentBuffer = initialEmptyBuffer;
            try {
                while (currentBuffer != null) {
                addToBuffer(currentBuffer);
                if (currentBuffer.full())
                    currentBuffer =
    exchanger.exchange(currentBuffer);
                }
            } catch (InterruptedException ex) { ... handle ... }
        }
}
```

"Canned" Synchronizers

```
class EmptyingLoop implements Runnable {
        public void run() {
                DataBuffer currentBuffer = initialFullBuffer;
                try {
                        while (currentBuffer != null) {
                                takeFromBuffer(currentBuffer);
                                if (currentBuffer.empty())
                                        currentBuffer =
    exchanger.exchange(currentBuffer);
                        }
                } catch (InterruptedException ex) { ... handle ...}
        }
  }
  void start() {
        new Thread(new FillingLoop()).start();
        new Thread(new EmptyingLoop()).start();
  }
}
```

"Canned" Synchronizers

- ***Synchronous Queues***
  - A synchronous queue is a mechanism that pairs up producer and consumer threads. When a thread calls put on a SynchronousQueue, it blocks until another thread calls take, and vice versa. Unlike the case with an Exchanger, data are only transferred in one direction, from the producer to the consumer.
  - Even though the SynchronousQueue class implements the BlockingQueue interface, it is not conceptually a queue. It does not contain any elements-its size method always returns 0.
  - A blocking queue in which each put must wait for a take, and vice versa. A synchronous queue does not have any internal capacity, not even a capacity of one.
  - You cannot peek at a synchronous queue because an element is only present when you try to take it; you cannot add an element (using any method) unless another thread is trying to remove it; you cannot iterate as there is nothing to iterate.

"Canned" Synchronizers

- The *head* of the queue is the element that the first queued thread is trying to add to the queue; if there are no queued threads then no element is being added and the head is null. For purposes of other Collection methods (for example contains), a SynchronousQueue acts as an empty collection. This queue does not permit null elements.
- Synchronous queues are similar to rendezvous channels used in CSP and Ada. They are well suited for handoff designs, in which an object running in one thread must sync up with an object running in another thread in order to hand it some information, event, or task.

"Canned" Synchronizers

- Semaphores
  - A counting semaphore. Conceptually, a semaphore maintains a set of permits. Each acquire() blocks if necessary until a permit is available, and then takes it. Each release() adds a permit, potentially releasing a blocking acquirer.
  - However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly.
  - Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource.

"Canned" Synchronizers

```
class Pool {
    private static final MAX_AVAILABLE = 100;
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);

    public Object getItem() throws InterruptedException {
        available.acquire();
        return getNextAvailableItem();
    }

    public void putItem(Object x) {
        if (markAsUnused(x))
        available.release();
    }
```

"Canned" Synchronizers

```
// Not a particularly efficient data structure; just for demo
protected Object[] items = ... whatever kinds of items being managed
protected boolean[] used = new boolean[MAX_AVAILABLE];

protected synchronized Object getNextAvailableItem() {
        for (int i = 0; i < MAX_AVAILABLE; ++i) {
                if (!used[i]) {
                        used[i] = true;
                        return items[i];
                }
        }
        return null; // not reached
}
```

"Canned" Synchronizers

```
protected synchronized boolean markAsUnused(Object item) {
        for (int i = 0; i < MAX_AVAILABLE; ++i) {
            if (item == items[i]) {
                if (used[i]) {
                    used[i] = false;
                    return true;
                } else
                return false;
            }
        }
        return false;
    }

}
```

Callable and Future

- A Runnable encapsulates a task that runs asynchronously; you can think of it as an asynchro-nous method with no parameters and no return value.
- A Callable is similar to a Runnable, but it returns a value. The Callable interface is a parameterized type, with a single method call.

```
public interface Callable<V>
{
        V call() throws Exception;
}
```

Callable and Future

- The type parameter is the type of the returned value. For example, a Callable<Integer> represents an asynchronous computation that eventually returns an Integer object.
- A Future holds the *result* of an asynchronous computation. You use a Future object so that you can start a computation, give the result to someone, and forget about it. The owner of the Future object can obtain the result when it is ready.
- The Future interface has the following methods:

```
public interface Future<V>
{       //exception throwing not shown.
        V get() ;
        V get(long timeout, TimeUnit unit) ;
        void cancel(boolean oayInterrupt);
        Boolean isCancelled(); Boolean isDone();
}
```

Callable and Future

- A call to the first get method blocks until the computation is finished. The second method throws a TimeoutException if the call timed out before the computation finished.
- If the thread running the computation is interrupted, both methods throw an InterruptedException. If the computation has already finished, then get returns immediately.
- The isDone method returns false if the computation is still in progress, true if it is finished. You can cancel the computation with the cancel method. If the computation has not yet started, it is canceled and will never start.
- If the computation is currently in progress, then it is interrupted if the mayInterrupt parameter is true.

Callable and Future

- The FutureTask wrapper is a convenient mechanism for turning a Callable into both a Future and a Runnable,it implements both interfaces.

```
Callable<Integer> myComputation =…….
FutureTask<Integer> task =new Future Task<Integer>(myComputation);
Thread t = new Thread (task); // it's a Runnable
t.start();


//after some time……
Integer result = task.get(); // it's a Future
```