Multithreading

What is Multithreading?

- Multithreading is similar to multi-processing.
- A multi-processing Operating System can run several processes at the same time
  - Each process has its own address/memory space
  - The OS's scheduler decides when each process is executed
    - Only one process is actually executing at any given time. However, the system appears to be running several programs simultaneously
- Separate processes to not have access to each other's memory space
  - Many OSes have a shared memory system so that processes can share memory space
- In a multithreaded application, there are several points of execution within the same memory space.
  - Each point of execution is called a thread
  - Threads share access to memory

Why use Multithreading?

- In a single threaded application, one thread of execution must do everything
  - If an application has several tasks to perform, those tasks will be performed when the thread can get to them.
  - A single task which requires a lot of processing can make the entire application appear to be "sluggish" or unresponsive.
- In a multithreaded application, each task can be performed by a separate thread
  - If one thread is executing a long process, it does not make the entire application wait for it to finish.
- If a multithreaded application is being executed on a system that has multiple processors, the OS may execute separate threads simultaneously on separate processors.

What Kind of Applications Use Multithreading?

- Any kind of application which has distinct tasks which can be performed independently
  - Any application with a GUI.
    - Threads dedicated to the GUI can delegate the processing of user requests to other threads.
    - The GUI remains responsive to the user even when the user's requests are being processed
  - Any application which requires asynchronous response
    - Network based applications are ideally suited to multithreading.
      - Data can arrive from the network at any time.
      - In a single threaded system, data is queued until the thread can read the data
      - In a multithreaded system, a thread can be dedicated to listening for data on the network port
      - When data arrives, the thread reads it immediately and processes it or delegates its processing to another thread

How does it all work?

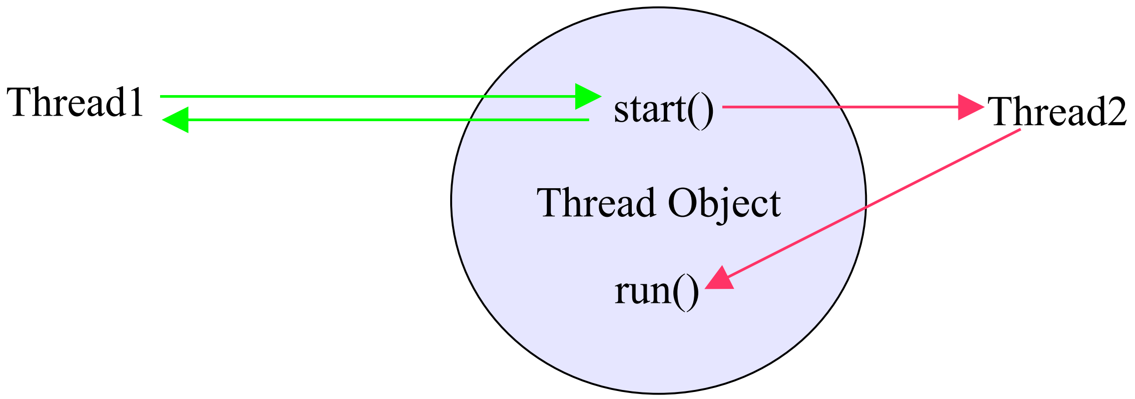
- Each thread is given its own "context"
  - A thread's context includes virtual registers and its own calling stack
- The "scheduler" decides which thread executes at any given time
  - The VM may use its own scheduler
  - Since many OSes now directly support multithreading, the VM may use the system's scheduler for scheduling threads
- The scheduler maintains a list of ready threads (the run queue) and a list of threads waiting for input (the wait queue)
- Each thread has a priority. The scheduler typically schedules between the highest priority threads in the run queue
  - Note: the programmer cannot make assumptions about how threads are going to be scheduled. Typically, threads will be executed differently on different platforms.

Thread Support in Java

- Few programming languages directly support threading
  - Although many have add-on thread support
  - Add on thread support is often quite cumbersome to use
- The Java Virtual machine has its own runtime threads
  - Used for garbage collection
- Threads are represented by a Thread class
  - A thread object maintains the state of the thread
  - It provides control methods such as interrupt, start, sleep, yield, wait
- When an application executes, the main method is executed by a single thread.
  - If the application requires more threads, the application must create them.

How does a Thread run?

- The thread class has a run() method
  - run() is executed when the thread's start() method is invoked
- The thread terminates if the run method terminates
  - To prevent a thread from terminating, the run method must not end
  - run methods often have an endless loop to prevent thread termination
- One thread starts another by calling its start method
  - The sequence of events can be confusing to those more familiar with a single threaded model.



Creating your own Threads

- The obvious way to create your own threads is to subclass the Thread class and then override the run() method
  - This is the easiest way to do it
  - It is not the recommended way to do it.
- Because threads are usually associated with a task, the object which provides the run method is usually a subclass of some other class
  - If it inherits from another class, it cannot inherit from Thread.
- The solution is provided by an interface called Runnable.
  - Runnable defines one method - public void run()
- One of the Thread classes constructor takes a reference to a Runnable object
  - When the thread is started, it invokes the run method in the runnable object instead of its own run method.

Subclass Thread

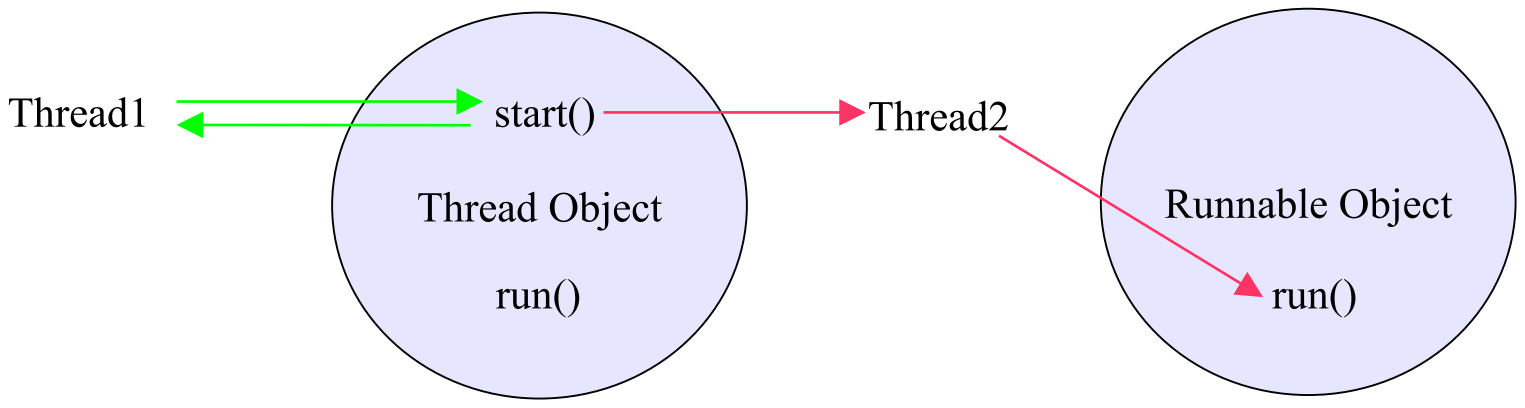
The Thread class itself implements `Runnable`, though its `run` method does nothing.

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        Thread t = new HelloThread();
        t.start();
    }
}
```

Using Runnable

- In the example below, when the Thread object is instantiated, it is passed a reference to a "Runnable" object
  - The Runnable object must implement a method called "run"

- When the thread object receives a start message, it checks to see if it has a reference to a Runnable object:
  - If it does, it runs the "run" method of that object
  - If not, it runs its own "run" method



Runnable

The Runnable interface defines a single method, `run`, meant to contain the code executed in the thread.

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        HelloRunnable rb = new HelloRunnable();
        Thread t = new Thread(rb);
        t.start();
    }
}
```

Properly Terminating Threads

- In Java 1.1, the Thread class had a stop() method
  - One thread could terminate another by invoking its stop() method.
  - However, using stop() could lead to deadlocks
  - The stop() method is now deprecated. DO NOT use the stop method to terminate a thread
- The correct way to stop a thread is to have the run method terminate
  - Add a boolean variable which indicates whether the thread should continue or not
  - Provide a set method for that variable which can be invoked by another thread

Pausing a Thread

- Thread.sleep causes the current thread to suspend execution for a specified period.
  - This is an efficient means of making processor time available to the other threads of an application.

- Sleep time can be specified in millisecond or nanosecond.
  - However, these sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS.
  - Also, the sleep period can be terminated by interrupts

Pausing a Thread

```
public class SleepMessages {
    public static void main(String args[])  throws InterruptedException

    {
        String importantInfo[] = { "Mares eat oats", "Does eat oats", 
"Little lambs eat ivy", "A kid will eat ivy too" };

        for (int i = 0; i < importantInfo.length;i++) {
            //Pause for 4 seconds
            Thread.sleep(4000);
            //Print a message
            System.out.println(importantInfo[i]);
        }
    }
}
```

Notice that main declares that it throws InterruptedException. This is an exception that sleep throws when another thread interrupts the current thread while sleep is active

Interrupts

- An interrupt is an indication to a thread that it should stop what it is doing and do something else.
  - Programmer decides how a thread responds to an interrupt,
  - common approach is to terminate the thread

Interrupts

Invoke interrupt: Multiple approaches-

- If the thread is invoking a method that throw InterruptedException (such as sleep method), it simply returns from the run method after it catches that exception.

```
for (int i = 0; i < importantInfo.length; i++) {
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

Interrupts

- What if a thread goes a long time without invoking a method that throws InterruptedException?
    - Then it must periodically invoke Thread.interrupted, which returns true if an interrupt has been received. For example:

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

Interrupts

- call `Thread.currentThread().interrupt()` to set the interrupted status

- If you call the `sleep` method when the interrupted status is set, it doesn’t sleep. Instead, it clears the status (!) and throws an `InterruptedException`.

- Calling the `interrupted` method clears the interrupted status of the thread.

- `isInterrupted` method is an instance method that you can use to check whether any thread has been interrupted. Calling it does not change the interrupted status

Join

- The join method allows one thread to wait for the completion of another.
- If t is a Thread object whose thread is currently executing,

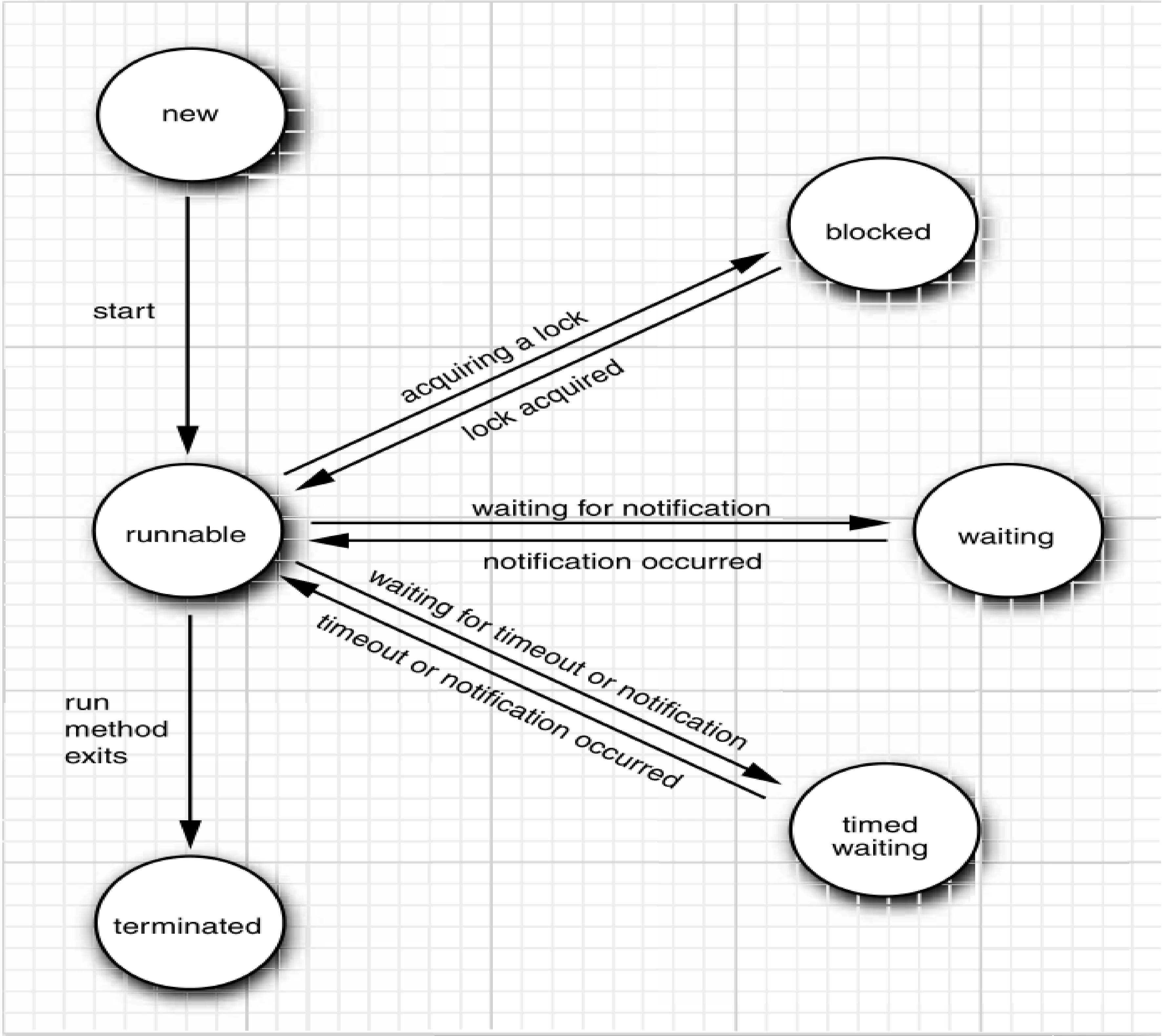
```
t.join();
```

causes the current thread to pause execution until t's thread terminates.

- Overloads of join allow the programmer to specify a waiting period.
- However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.
- Like sleep, join responds to an interrupt by exiting with an InterruptedException.

Thread States

- New
- Runnable
- Blocked
- Waiting
- Timed waiting
- Terminated


new
start
blocked
acquiring a lock
lock acquired
runnable
waiting for notification
waiting
notification occurred
waiting for timeout or notification
timeout or notification occurred
run method exits
timed waiting
terminated

Creating Multiple Threads

- Is there a maximum number of threads which can be created?
  - There is no defined maximum in Java.
  - If the VM is delegating threads to the OS, then this is platform dependent.
  - A good rule of thumb for maximum thread count is to allow 2Mb of ram for each thread
    - Although threads share the same memory space, this can be a reasonable estimate of how many threads your machine can handle.

Thread Priorities

- Every thread is assigned a priority (between 1 and 10)
  - The default is 5
  - The higher the number, the higher the priority
  - Can be set with setPriority(int aPriority)

- The standard mode of operation is that the scheduler executes threads with higher priorities first.
  - This simple scheduling algorithm can cause problems. Specifically, one high priority thread can become a "CPU hog".
  - A thread using vast amounts of CPU can share CPU time with other threads by invoking the yield() method on itself.

- Most OSes do not employ a scheduling algorithm as simple as this one
  - Most modern OSes have thread aging
    - The more CPU a thread receives, the lower its priority becomes
    - The more a thread waits for the CPU, the higher its priority becomes
  - Because of thread aging, the effect of setting a thread's priority is dependent on the platform

Yield() and Sleep()

- Sometimes a thread can determine that it has nothing to do
  - Sometimes the system can determine this. ie. waiting for I/O
- When a thread has nothing to do, it should not use CPU
  - This is called a busy-wait.
  - Threads in busy-wait are busy using up the CPU doing nothing.
    - Often, threads in busy-wait are continually checking a flag to see if there is anything to do.
- It is worthwhile to run a CPU monitor program on your desktop
  - You can see that a thread is in busy-wait when the CPU monitor goes up (usually to 100%), but the application doesn't seem to be doing anything.
- Threads in busy-wait should be moved from the Run queue to the Wait queue so that they do not hog the CPU
  - Use yield() or sleep(time)
  - Yield simply tells the scheduler to schedule another thread
  - Sleep guarantees that this thread will remain in the wait queue for the specified number of milliseconds.

Synchronization

- Threads communicate by sharing
  - access to fields and
  - the objects reference fields refer to.
- This form of communication is extremely efficient, but makes two kinds of errors possible:
  - thread interference
  - memory consistency errors.
- The tool needed to prevent these errors is synchronization.
- However, synchronization can introduce thread contention, which occurs when two or more threads try to access the same resource simultaneously and cause the Java runtime to execute one or more threads more slowly, or even suspend their execution.

Thread Interference

```
class Counter {
       private int c = 0;
       public void
   increment() {
           c++;
       }
       public void
   decrement() {
           c--;
       }
       public int value() {
           return c;
       }
   }
```

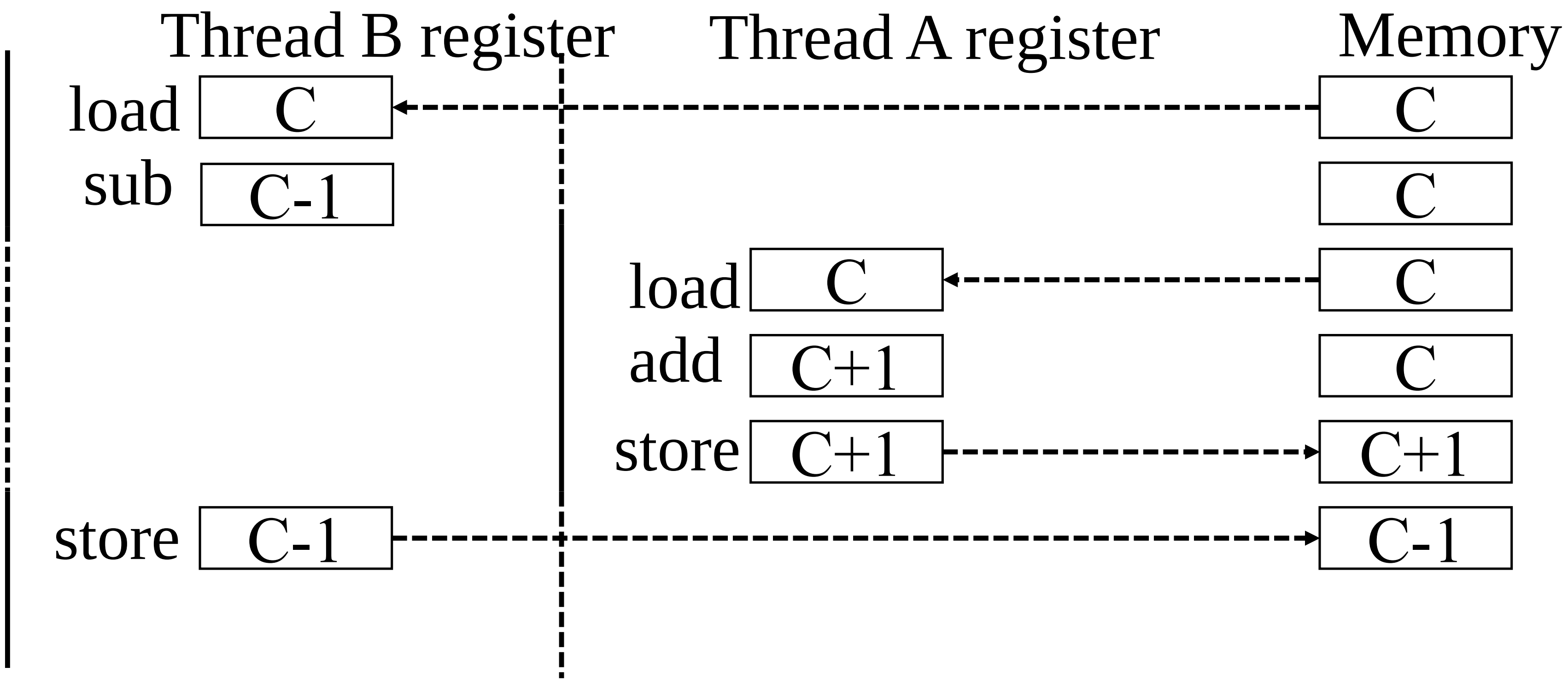
Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

Example: Single expression c++ can be decomposed into three steps:

1. Retrieve the current value of c.
2. Increment the retrieved value by 1.
3. Store the incremented value back in c.

The expression c-- can be decomposed the same way, with decrement instead of increment.

Timeline



Thread B register
Thread A register
Memory
load
C
C
sub
C-1
C
load
C
C
add
C+1
C
store
C+1
C+1
store
C-1
C-1

Thread Interference

Suppose Thread A invokes increment at about the same time Thread B invokes decrement. If the initial value of c is 0, their interleaved actions might follow this sequence:

1. Thread B: Retrieve c.
2. Thread B: Decrement retrieved value; result is -1.
3. Thread A: Retrieve c.
4. Thread A: Increment retrieved value; result is 1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

Thread A's result is lost, overwritten by Thread B.

This particular interleaving is only one possibility! Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all.

Memory Consistency Error

- Different threads may have inconsistent views of what should be the same data

```
int counter = 0;

counter++;

System.out.println(counter);
```

  - If the two statements had been executed in the same thread?
  - If the two statements are executed in separate threads?

Memory Consistency Error

If the two statements had been executed in the same thread?

Value printed out would be "1".

But if the two statements are executed in separate threads?

The value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a happens-before relationship between these two statements.

Memory Consistency Error

- When a statement invokes Thread.start, every statement that has a happens-before relationship with that statement also has a happens-before relationship with every statement executed by the new thread. The effects of the code that led up to the creation of the new thread are visible to the new thread.

- When a thread terminates and causes a Thread.join in another thread to return, then all the statements executed by the terminated thread have a happens-before relationship with all the statements following the successful join. The effects of the code in the thread are now visible to the thread that performed the join.

Locks

```
public class SynchronizedCounter {
    private int c = 0;

    private Lock myLock = new ReentrantLock();
    public void increment() {

      myLock.lock();

       c++;

      myLock.unlock();
    }
    public void decrement() {
      myLock.lock();

       c--;

      myLock.unlock();
    }
}
```

Synchronized Methods

To make a method synchronized, simply add the *synchronized* keyword to its declaration:

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Synchronized Methods

If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:

- Two invocations of synchronized methods on the same object can’t interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- When a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the synchronized keyword with a constructor is a syntax error

Conditional Wait

```
ReentrantLock myLock = new ReentrantLock();
Condition myCondition = myLock.newCondition();
Foo(){
myLock.lock();
…
while (!(ok to proceed))
   myCondition.await();
}
Foo2(){
…
myCondition.signalAll();
}
```

`synchronized` and Conditional Wait

```
public synchronized void foo1() throws InterruptedException
{
  while (!ok_to_proceeed)
    wait(); // wait on intrinsic object lock's condition
  do_something();
  notifyAll(); // notify all threads waiting on the
condition
}
```

synchronized blocks

```
public void foo()
{
  synchronized (lock) // an ad-hoc lock, any object
  {
    //critical section
    …
  }
}
```

volatile

- Processors can temporarily hold memory values in registers or local memory caches. Threads running in different processors may see different values for the same memory location.

- Use volatile for such variables. If you declare a field as volatile , then the compiler and the virtual machine take into account that the field may be concurrently updated by another thread.

Deadlock

- Thread A

```
condition1.await();

do_something();

condition2.notifyAll();
```

- Thread B

```
condition2.await();

do_something();

condition1.notifyAll();
```

Locks, Interrupt and Deadlock

- The lock method cannot be interrupted.
- If a thread is interrupted while it is waiting to acquire a lock, the interrupted thread continues to be blocked until the lock is available.
- If a deadlock occurs, then the lock method can never terminate.
- If you call `tryLock` with a timeout, then an `InterruptedException` is thrown if the thread is interrupted while it is waiting.
- `lockInterruptibly` method. It has the same meaning as `tryLock` with an infinite timeout.
- `await` and `awaitInterruptibly`

Read Write Locks

```
ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

Lock readLock = rwl.readLock();

Lock writeLock = rwl.writeLock();
```

- `readLock` gets a read lock that can be acquired by multiple readers, excluding all writers.
- `writeLock()` gets a write lock that excludes all other readers and writers.

Java concurrent data structures

- BlockingQueue::put(). Adds an element. Blocks if the queue is full.
- BlockingQueue:: take(). Removes and returns the head element. Blocks if the queue is empty
- PriorityBlockingQueue
- ArrayBlockingQueue
- LinkedBlockingQueue

Thread safe data structures

- java.util.concurrent package. ConcurrentHashMap, ConcurrentSkipListMap
- ConcurrentSkipListSet, ConcurrentLinkedQueue.
- Unlike in most collections, the size method does not necessarily operate in constant time. Determining the current size of one of these collections usually requires traversal.
- An iterator of a collection in the java.util package throws a ConcurrentModificationException when the collection has been modified after construction of the iterator.
- The collections in concurrent package return weakly consistent iterators. That means that the iterators may or may not reflect all modifications that are made after they were constructed, but they will not return a value twice and they will not throw a ConcurrentModificationException.

Swing is NOT thread safe.

Callable and Future

- `Callable` is similar to a Runnable , but it returns a value.

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

- `Future` holds the result of Callable

```
public interface Future<V>{ V get() throws . . .; }
```

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

Future

- `Future` provides **get()** method that can wait for the Callable to finish and then return the result.
- Future provides **cancel()** method to cancel the associated Callable task.
- There is an overloaded version of get() method where we can specify the time to wait for the result, it’s useful to avoid current thread getting blocked for longer time.
- There are **isDone()** and **isCancelled()** methods to find out the current status of associated Callable task.

Callable and Future

```
Callable<Integer> myC = . . .;
FutureTask<Integer> task = new FutureTask<Integer>(myC);
Thread t = new Thread(task); // it's a Runnable
t.start();
try {
   Integer I = task.get();
   // do something
}
catch (ExecutionException e) {   e.printStackTrace(); }
catch (InterruptedException e){ }
```

Thread pools

- Thread pool contains a number of idle threads that are ready to run.
- You give a Runnable to the pool, and one of the threads calls the run method.
- When the run method exits, the thread doesn’t die but stays around to serve the next request
- Use `submit` method to submit your task implementing `Callable` or `Runnable`

```
ExecutorService pool = Executors.newCachedThreadPool();

Callable<Integer> myCallable = new …

Future<Integer> result = pool.submit(myCallable);
```

Barriers

```
CyclicBarrier myBarrier = new CyclicBarrier(num_threads);


public void run()
{
//do something
myBarrier.await();
}
```

- Use barrier action to execute some code when all threads have reached the barrier

```
Runnable barrierAction = . . .;
CyclicBarrier barrier = new CyclicBarrier(num_threads,
barrierAction);
```

```
class Solver {
  class Worker implements Runnable {
  Worker(int row) { … }
   public void run() {
          processRow(myRow);
          barrier.await();
          …
   }
   public Solver(float[][] matrix) {
          barrier = new CyclicBarrier(N, new Runnable() {
                                              public void run() { mergeRows(...); }
                                            });
   for (int i = 0; i < N; ++i)
     new Thread(new Worker(i)).start();
   waitUntilDone();
…
```

Semaphore and CountDownLatch

- A semaphore can controls a number of permits.
- Calling `acquire` allows only a fixed number of threads to pass.

- A CountDownLatch lets a set of threads wait until a count has reached zero.
- The countdown latch is one-time only. Once the count has reached 0, you cannot increment it again.

Swing and MultiThreading

- Swing is not thread safe. The only method which are thread-safe in the swing are repaint() and revalidate().
- Use event dispatch thread to do your tasks in a thread safe manner

```
EventQueue.invokeLater(new Runnable(){
  public void run(){
   //do something with gui components
  }
  });
```

- InvokeAndWait is a blocking call and wait until all pending AWT events get processed and run() method completes.
- InvokeAndWait should not be called from EventDispatcher thread unlike invokeLater; it’s an error because it will result in guaranteed deadlock