CS 131 Programming Languages

October 22, 2009

Java Thread Lifecycle

One can use the `new` operator to create a new thread.

```
t = new Thread(foo)    // resulting state of the thread is NEW
t.start();
```

Starting the thread makes an operating system allocate a piece of Java virtual machine (resources) and after all is set up, invoke thread's `run()` method. The resulting state of the thread is now `RUNNABLE`. The resulting child thread can do any ordinary stuff or do something that potentially changes its state: sleep, wait, yield, I/O. The result of the latter operation can be a change of thread's state to `BLOCKED, WAITING, TIMEDWAITING`. In these states the thread is not runnable until the operating system gets back to it.

The simplest way for the thread to exit is to return from its `run()` method, which changes its state to `TERMINATED`.

In a meantime, the parent can do something else but a common thing is to have parent wait for the thread to be terminated by calling `t.join()`. This puts the parent into the `WAITING` state. If the parent does not trust its child thread, it can pass the maximum number of milliseconds the waiting should take: `t.join(1000)`.

Parent can wait for multiple threads to complete its job. In homework 3, the parent could invoke the new thread 3 times, invoke start 3 times, and invoke join 3 times. The job is split into 3 equal pieces, each child thread has read-only access to the previous state, and write-only access to the next (current) state. Recursion can help when we'd like to split up the work among a large number of threads, such as 30,000. The parent would invoke the first few threads, which in turn would invoke next few threads each, etc.

The homework 3 assignment is an instance of what's called *embarrassing parallelism* – e.g. every thread in this application writes to a different region or the writable regions are partitioned one per thread and no thread reads from any writeable region other than its own region. Unfortunately, lots of applications are *not* embarrassingly parallel and this is a problem that needs to be solved.

Let's have a simple method that's going to screw up if the application is not embarrassingly parallel:

```
class X {
        int v;
        int next() {
                return v++;
        }
}
X p = new X();
                // let v = 1000;
i = p.next();  // in thread 1
j = p.next();  // in thread 2
```

As a result of the above code, both `i` and `j` might have gotten the same value 1000 because both threads loaded `v` out of memory at the same time.

There's a standard way in Java to avoid situations like these – a keyword `synchronized`.

```
synchronized int next() {
        …
}
```

Associated with every object in Java is an extra area there is a lock that's visible to the Java runtime system which informs whether someone is accessing the object at the time.

An issue with synchronized is that commonly accessed objects that have synchronized methods can lead to bottlenecks, particularly in high-volume applications. Sun's original Java library used lots of synchronized methods, which resulted in bottlenecks on server applications. However, current library uses no synchronized methods and it's the responsibility of the programmer to avoid conflicts among threads.

Other synchronization methods that are higher-level are available in Java. Let's look at some of these methods:

1. *Semaphore*: contains a limited number of "spaces" so that only a few threads can acquire the resources simultaneously.
    a. `s.acquire() // wait for acquire to succeed`
    b. `s.release()`
    c. `s.tryAcquire() // return bool indicating if acquire succeeded`
2. *Exchanger*: a *rendezvous* point between two threads. Two threads access an exchanger object.
    a. `v1 = x.exchange(v) // thread 1`
    b. `v = x.exchange(v1) // thread 2`
3. *CountDownLatch (N)* where N is the number of threads to count down to. Not re-usable.
4. *CyclicBarrier* is a re-usable countdown latch.

Java's Internals

Any object has a `wait()` method, which removes all locks that the thread has and wait until the object becomes available. Any object also has a `notify()` method which notifies one waiting thread that the object is available. Another primitive is `notifyAll()` which notifies all threads, giving the programmer a more fine-grained control over scheduling. Potentially, every object has a list of threads waiting on it.

Scope of Names

Scope of a name is the set of program locations where that name is visible. The scope is confined inside a block of code (defined by curly braces `{}`) and does not extend into other blocks of code. *Referencing environment* of a program location is the set of visible names. Scope of the variable *does not always* equal to its lifetime. While lifetime is a run-time notion, scope is a compile-time notion. However, there are languages in which scope is a run-time notion – those that use *dynamic scoping*

Namespaces

C has separate namespaces for: ordinary variables, structs, enums, member names (for each struct), #include file names, #defines, labels. Each namespace does not conflict with others given the same variable name. *Labels* have only one namespace per function because C designers wanted the ability to jump into *any* place within the code. *#includes* and *#defines* have one namespace per file. *Member names* have one namespace per struct. Everything else has one namespace per block of code.

Labeled namespace is a set of names and objects that is given a name, such as *interface* in Java, *structure* in ML, *package* in Ada – behaving like an API. The first idea behind labeled namespace is *abstraction* – exposing only a subset of the information to the entire system (namespace/visibility control). Abstraction lessens confusion between too many names, makes it easier to split program into small, interchangeable modules, and enables separate compilation, not having to recompile many modules unless their external namespace is changed.

There are a few methods to implement abstraction. Method A is *visibility modifiers*, which specify whether each name is *public*, *protected*, *(none)*, or *private*. In Java, public method is visible everywhere the class is visible, private method is visible only inside the class, (none) is visible inside the class and the package, and protected method is visible inside the class and its subclasses and the package.

Compilation Units and Packages

```
package edu.ucla.cs.cs131.multithreaded_automaton_synchronizer {
        class X {};
        class Y {};
}
```

The point of the *package hierarchy* is to manage/maintain the source code to see who's in charge of a package. Java also has the class/object hierarchy with `Object` on the top.

In your free time, read Java's documentation on the *class Object*.

```
public class Object {
        public Object();                        // constructor: new Object();
        public boolean equals(Object o2);       // comparison: o1.equals(o2);
        public int hashCode();                  // defaults to address of the Object
```

```
                                        // o1.equals(o2) -> o1.hashCode() == o2.hashCode()
        public final Class getClass();  // useful for debug tools that want to know the class of the object
        public String toString();       // default implementation: return the memory address of the object
        protected void finalized()      // when an object becomes garbage and GC is about to throw it out,
                throws Throwable;       // the finalize() method is called, which is empty by default.
                                        // this is to clean up resources that GC is not aware of.
        protected Object clone()        // return a copy of the object
                throws CloneNotSupportedException;    // some objects should never be cloned
        // also wait(), notify(), notifyAll();
}
```

For the midterm: read all the Java and OCaml chapters and all other chapters up to chapter 11.