

Java Threads

In Java, the concepts about threads, threads stacks, and starting threads applies. The primary difference is that in Java, a thread is defined as a special class as part of the core Java Language. In other languages, like C/C++, threads are part of a separate “add-on” library and were not defined originally as part of the language definition.

In Java, the thread class implements an **interface** called the **Runnable** interface, which defines a single abstract method called “run”. An interface in Java is like an abstract base class in C++ with all pure virtual methods and no local data.

```
// in the java.lang.Runnable file you will find the following interface
package java.lang; // part of the core java language namespace
```

```
public interface Runnable {
    public void run(); // no impl. just like a pure virtual function in C++
}
```

Since we are defining a Java interface, the run method is implicitly a “abstract” method (pure virtual). As we have seen, interfaces in Java define a set of methods with NO implementation. Some subclass must implement the Runnable interface and provide an implementation of the run method, which is the method that is started by a thread. This is a great example of the concept of separation of interface from implementation, as discussed earlier during the OOP part of the course.

When you provide a class that implements the Runnable interface, you provide an implementation of the run method.

```
public class ConcurrentReader implements Runnable {
    ...
    public void run() { /* code here executes concurrently with caller */ }
    ...
}
```

Defining a Thread in Java

To start this thread you need to first create an object of type `Runnable`, bind it to a new `Thread` object, and then start it. Calling `start` creates the thread stack for the thread, and then invokes the `run()` method of the `Runnable` object as the first procedure on that new thread stack.

```
ConcurrentReader readerThread = new ConcurrentReader();
Thread t = new Thread(readerThread); // create thread using a Runnable object
t.start(); // start automatically calls ConcurrentReader.run()
```

The `java.lang.Thread` class has a constructor that takes an object of type `Runnable`:

```
Thread(Runnable object); // must provide an object that implements run
```

Alternatively, we can define a subclass of the class `Thread` directly.

```
class ConcurrentWriter extends Thread {
    public void run() {
        // you provide the code here to run as a separate thread of control
    }
}
```

To start this thread you just need to do the following:

```
ConcurrentWriter writerThread = new ConcurrentWriter();
writerThread.start(); // start calls run() automatically
```

java.lang.Thread

Here is a fragment of the code for the Java Thread class. Note the constructors, default run() method and the “native” start method. A native method in Java has its implementation inside of the Java virtual machine. In the case of the start() method, its job is to setup the run-time stack on which the run() method will execute, so that is why it is a native method---it does all of the “dirty” work of managing the operating system and machine specific tasks of creating the execution environment for the thread.

```
public class Thread implements Runnable {
    private char name[];
    private Runnable target;
    ...
    public final static int MIN_PRIORITY = 1;
    public final static int NORM_PRIORITY = 5;
    public final static int MAX_PRIORITY = 10;

    private void init(ThreadGroup g, Runnable target, String name) {...}

    public Thread() { init(null, null, "Thread-" + nextThreadNum()); }
    public Thread(Runnable target) {
        init(null, target, "Thread-" + nextThreadNum());
    }
    public Thread(Runnable target, String name) { init(null, target, name); }

    public synchronized native void start();

    public void run() {
        if (target != null) {
            target.run();
        }
    }
}
```

java.lang.Thread

There are a number of methods defined on the Thread class that allow you to query the thread to find its priority, to put it to sleep, cause it to yield to another thread, stop, suspend its execution, resume its execution, etc. All of these methods allow the programmer to manage running threads. When one thread yields or is suspended, then another thread can execute. We call this a **context-switch**. The context of a thread is the set of register values (general purpose and floating point registers) that are associated with the execution state of a thread. When a thread yields or is suspended, this state must be saved so that it can be restored when the thread resumes execution.

```
public class Thread implements Runnable {
    ...
    public static native Thread currentThread();
    public static native void yield();
    public static native void sleep(long millis) throws InterruptedException;
    public static int enumerate(Thread tarray[])

    public static boolean interrupted() { ... }
public boolean isInterrupted() { ... }
    public final native boolean isAlive();
    public String toString() {
    public void interrupt() { ... }
    public void interrupt() { ... }
    public final void stop() { ... }
    public final void suspend() { ... }
    public final void resume() { ... }
    public final void setPriority(int newPriority) {
    public final int getPriority() {
    public final void setName(String name) { ... }
    public final String getName() { return String.valueOf(name); }
    public native int countStackFrames();
    public final synchronized void join() throws InterruptedException {...}
    public void destroy() { throw new NoSuchMethodError(); }
}
```

Extending Class Thread vs Implementing Interface Runnable

Q: Why does Java allow two different ways to provide thread objects? How do you decide when to extend the Thread class versus implementing the Runnable interface?

Java only allows single class inheritance, so if we have a class that needs to inherit from another class, but also needs to run as a thread, then we extend the other class and implement the Runnable interface. So, it is quite common for a class X to extend some class Y and implement the Runnable interface:

```
class X extends Y implements Runnable {  
  
    public synchronized void do_something() { ... }  
    public void run() { do_something(); } // can be run a thread if needed  
}
```

By implementing the Runnable interface, rather than extending the Thread class, you are communicating to the user of the class X that you expect that an object of type X will run as a thread, but it does not **HAVE TO** run as a thread. Since all the run() method does, in this case, is call another public method that could be called without running a thread, it gives the user the option of either having an object of type X run concurrently, or sequentially.

```
X obj = new X();  
obj.do_something(); // runs sequentially in the current thread  
Thread t = new Thread(new X()); // create an X and run as a thread  
t.start(); // start() calls run() which calls do_something()
```

Note: A **synchronized** method is one that “locks” an object so that no other thread can execute inside the object while the method is active. By locking the object, we have a guarantee that the state of the object will remain consistent during the duration in which the lock on the object is held by a thread. When the lock is released, then another thread can lock the object and update its state.

Synchronized Methods, Wait, Notify, and NotifyAll

An very interesting features of Java objects is that they are all “lockable” objects. Every object in Java is a subtype of the **Object** class in Java. The Object class implements an implicit locking mechanism that allows any Java object to be locked during the execution of a **synchronized method** or **synchronized block**, so that the thread that holds the lock gains exclusive access to the object for the duration of the method call or scope of the bloc. No other thread can “acquire” the object until the thread that holds the lock “releases” the object. This *synchronization policy* provides **mutual exclusion**.

Synchronized methods are methods that lock the object on entry and unlock the object on exit. The Object class implements some special methods for allowing a thread to explicitly release the lock while in the method, **wait** indefinitely or for some time interval, and then try to reacquire the lock when some condition is true. Two other methods allow a thread to signal waiting thread(s) to tell them to wakeup: the **notify** method signals one thread to wakeup and the **notifyAll** method signals all threads to wakeup and compete to try to re-acquire the lock on the object. This type of synchronized object is typically used to protect some shared resource, using two types of methods:

```
public synchronized void consume() {
    while (!consumable()) {
        wait();    // release lock and wait for resource
    }
    ... // have exclusive access to resource to consume
}
public synchronized void produce() {
    ... // change of state must result in consumable condition being true
    notifyAll(); // notify all waiting threads to try consuming
    // could also call just "notify()" and notify 1 thread at a time
}
```

Synchronized Method vs Synchronized Block

The synchronized method declaration qualifier is syntactic sugar for the fact that the entire of scope of the procedure is to be governed by the mutual exclusion condition obtained by acquiring the object's lock:

```
public void consume () {  
    synchronized(this) {  
        // code for consuming  
    }  
}
```

A synchronized block allows the granularity of a lock to be finer-grained than a procedure scope. The argument given to the synchronized block is a reference to an object.

What about recursive locking of the same object?

```
public class Foo {  
    ...  
    public void synchronized f() { ... }  
    public void synchronized g() { ...; f(); ... }  
}
```

If g() is called, and it then calls f(), what happens? What happens in the following case?

```
Foo f = new Foo;  
synchronized(f) { ...; synchronized (f) { ... } ... }
```

It turns out that locks on objects in Java are **recursive locks**, so that if a thread holds a lock on an object and the same thread requests to lock the object again, no deadlock occurs and the object is granted the lock since it already holds it.

Wait, Notify, and NotifyAll

Every object has access to wait, notify, and notify methods, which are inherited from class Object. The Object class in Java also implicitly contains a mutex lock and a condition variable. Synchronized methods “lock” the mutex lock on entry to the scope of the method, and automatically unlock the mutex lock on exit. The wait, notify and notifyAll methods operate on the implicit condition variable that is associated with each Java object. So when a thread “waits” on an object, it is enqueued on the condition variable that is present by the fact that all Java objects inherit from the Object class.

```
public class Object {
    ...
    public final native void notify();
    public final native void notifyAll();

    public final native void wait(long timeout) throws InterruptedException;
    public final void wait() throws InterruptedException { wait(0); }
    public final void wait(long timeout, int nanos)
        throws InterruptedException { ... }
}
```

The Object.wait() method implicitly releases the object’s lock and the thread then waits on an internal queue associated with each object. The thread waits to be notified of when it can try to re-acquire the lock and test the condition again.

The Object.notify() method signals the highest priority thread closest to the front of the wait queue to wakeup. Object.notifyAll() wakes up all waiting threads, and they compete for the lock. The thread actually gets the lock is **non-deterministic** and not necessarily “fair”. E.g., high priority threads in the wait queue could always win-out over lower priority threads, resulting in starvation since low priority threads never get access to the resource.

A Shared Queue Example

This is an example of a “Producer-Consumer” shared resource. Note that the `wait()`, `wait(timeout)`, `notify()`, and `notifyAll()` method can only be called from a synchronized method, or a method called by a synchronized method. I.e., the object must be in the locked state.

```
class SharedQueue {
    private Element head, tail;

    public boolean empty() { return head == tail; }

    public synchronized Element remove() {
        try { while (empty()) wait(); } // wait for an element in the queue
        catch (InterruptedException e) { return null; }
        Element p = head; head = head.next;
        if (head == null) tail = null;
        return p;
    }

    public synchronized void insert(Element p)
        if (tail == null) head = p;
        else tail.next = p;
        p.next = null;
        tail = p;
        notify(); // let one waiter know something is in the queue
    }
}
```

Overview of POSIX Threads

In C and C++, threads are provided by a library called POSIX Threads or Pthreads. The full Pthreads API is specified in the Linux manual pages. Pthreads are a standard API for programming with threads on all Unix systems (Solaris, AIX, HP/UX, Mac OS X, Linux, etc). Windows uses its own threading library called NT Threads, but there are some compatibility libraries that map Pthreads onto NT Threads. In C++, there exist C++ classes that map onto the more low-level C API for Pthreads, which makes programming with Pthreads more like programming with Java threads. To use pthreads, you `#include` the `<pthread.h>` header file, which incorporates the API interface definitions. It is also useful to `#include` `<unistd.h>` to obtain some standard unix definitions.

The main Pthread library calls to know are:

pthread_create - create a new thread giving it a “starting” procedure to run along with a single argument.

pthread_self - ask the currently running thread for its thread id.

pthread_join - join with a thread using its thread id (an integer value)

pthread_mutex_init - initialize a mutex structure

pthread_mutex_destroy - destroy a mutex structure

pthread_mutex_lock - lock an initialized mutex, if already locked suspend execution and wait

pthread_mutex_trylock - try to lock a mutex and if unsuccessful, do not suspend execution

pthread_mutex_unlock - unlock a mutex that was locked by the current thread

pthread_cond_init - initialize a condition variable structure

pthread_cond_destroy - destroy a condition variable structure

pthread_cond_wait - block the currently running thread on a condition variable indefinitely

pthread_cond_timedwait - block the currently running thread on a condition variable for a specific time

pthread_cond_signal - wakeup one thread blocked on a condition variable

pthread_cond_broadcast - wakeup all threads blocked on a condition variable

You should lookup the API definitions for each of these procedures by invoking the corresponding manual page for each on Linux and reading the descriptions, noting the types and kinds of arguments that each one requires. There are many more library class, as POSIX threads is a complete threading system, but knowing just the ones above allows you to write usefull concurrent programs in C and C++ using POSIX threads.

Example of POSIX Thread Creation in C

The main program creates a number of children threads that will call a procedure that prints a message, sleeps for a few seconds, then wakes up and then returns. The main procedure waits for each child thread to return by “joining” with each child thread. (NOTE: On Solaris, you link with `-lthread` to use POSIX threads. Other Unix systems may or may not have POSIX threads.)

```
#include <pthread.h>
#include <unistd.h> /* sleep declaration */
#include <stdio.h> /* printf declaration */
const int NUM_THREADS = 5;

void* sleeping(void* st)
{
    int sleep_time = (int) st; /* cast void* to an int */
    printf ("thread %d sleeping %d seconds ...\n", pthread_self(), sleep_time);
    sleep(sleep_time);
    printf ("\nthread %d awakening\n", pthread_self());
}

main( int argc, char *argv[] )
{
    pthread_t tid[NUM_THREADS]; /* array of thread IDs */
    int i;

    for ( i = 0; i < NUM_THREADS; i++)
        pthread_create (&tid[i], NULL, sleeping, i+2);

    for ( i = 0; i < NUM_THREADS; i++)
        pthread_join (tid[i], NULL);

    printf ("main() reporting that all %d threads have terminated\n", i);
} /* main */
```

Example of a C++ “Wrapper” Class for Thread Synchronization

```
class Synchronized {
    pthread_mutex_t m; // mutex variable
    pthread_cond_t c; // condition variable
protected:

    /* use this class to associate the mutex lock/unlock with the scope of a procedure */
    class Scope {
        Synchronized* obj;
    public:
        Scope(Synchronized* s) : obj(s) { pthread_mutex_lock(&obj->m); }
        ~Scope() { pthread_mutex_unlock(&obj->m); }
    };

public:

    Synchronized() { // initialize the mutex and condvar on construction
        pthread_mutex_init(&m, 0);
        pthread_cond_init(&c, 0);
    }

    ~Synchronized() { // destroy the mutex and condvar on destruction
        pthread_mutex_destroy(&m);
        pthread_cond_destroy(&c);
    }

    // map Java-like wait, notify and notifyAll onto pthread equivalents

    void wait() { pthread_cond_wait(&c, &m); }
    void notify() { pthread_cond_signal(&c); }
    void notifyAll() { pthread_cond_broadcast(&c); }
};
```

Using the C++ POSIX Wrapper Class

The C++ wrapper class can be inherited by another class to provide Java like object synchronization where the methods of the class are “synchronized” on the object that inherits from the Synchronized just like synchronized methods in Java work because every Java object inherits from the class Object, which is where the mutex and the condition variable is kept for each Java object, and that is where the wait, notify and notifyAll methods are defined in Java.

```
class MySynchronozedClass : public Synchronized {
    .. // private instance variables
public:

    // when this classes constructor is called, it first invokes the
    // constructor of the Synchronized class, which initialized the
    // the mutex and condition variable by calling the corresponding
    // pthread_{mutex,cond}_init library procedures
    MySynchronozedClass() { ... }

    // Likewise on destruction, the destructor of the Synchronized class is
    // automatically called and it destroys the mutex and condition variable
    ~MySynchronozedClass() { ...}

    // to make a method “synchronized” we declare a local variable of type
    // Synchronized::Scope, which locks the mutex on entry to the procedure scope
    // and automatically unlocks the mutex on exit from the procedure scope

    int some_method(...)
    {
        Synchronized::Scope mx(this); // automatically locks the mutex
        ... // execute code under mutual exclusion
    } // mx is automatically destructed, which unlocks the mutex
    ...
};
```

Homework Due Monday, 5 Nov

1. Read the Linux manual page for POSIX Threads by running the command: “man pthreads”. Then read the manual page for each of the POSIX library procedures mentioned previously in this lecture note. You will likely be quizzed over them.
2. Using Linux, implement the C/C++ threads program on the page 11 in this lecture note and change the number of threads to 10 and pick a random sleep time between 1 and 10 seconds for each thread. Run the program several times. Note the integer value assigned to each pthread that is created and the order in which the threads run, sleep and awake. Provide a printout of both your program and a trace of the each run of the output (screen snapshot is fine). Note: to compile with POSIX threads you must use Linux. You need to include the -pthread flag to the compiler to link your program with the pthreads library: gcc -pthread homework.c.