Concurrent Programming using Threads

Threads are a control mechanism that enable you to write concurrent programs. You can think of a thread in an object-oriented language as a special kind of “system object” that contains information about the state of execution of a sequence of function calls that are said to “execute as a thread”. Usually, a special “run” or “start” procedure starts a separate thread of control.

Normally, when you call a function or procedure, the compiler sets-up a stack frame (also called an activation frame) on the run-time procedure call stack, pushes arguments (or puts them into registers), and calls the function. The stack is also used as temporary storage for locally allocated objects declared in the scope of a procedure.

In a sequential program, there is only one run-time stack and all activation frames are allocated in a nested fashion on the same run-time stack, corresponding to each nested procedure call. In a multi-threaded application, each “thread” represents a separate run-time stack, so you can have multiple procedure call chains running at the same time, possibly on multiple processors. Java on Solaris supports multi-processor threads.

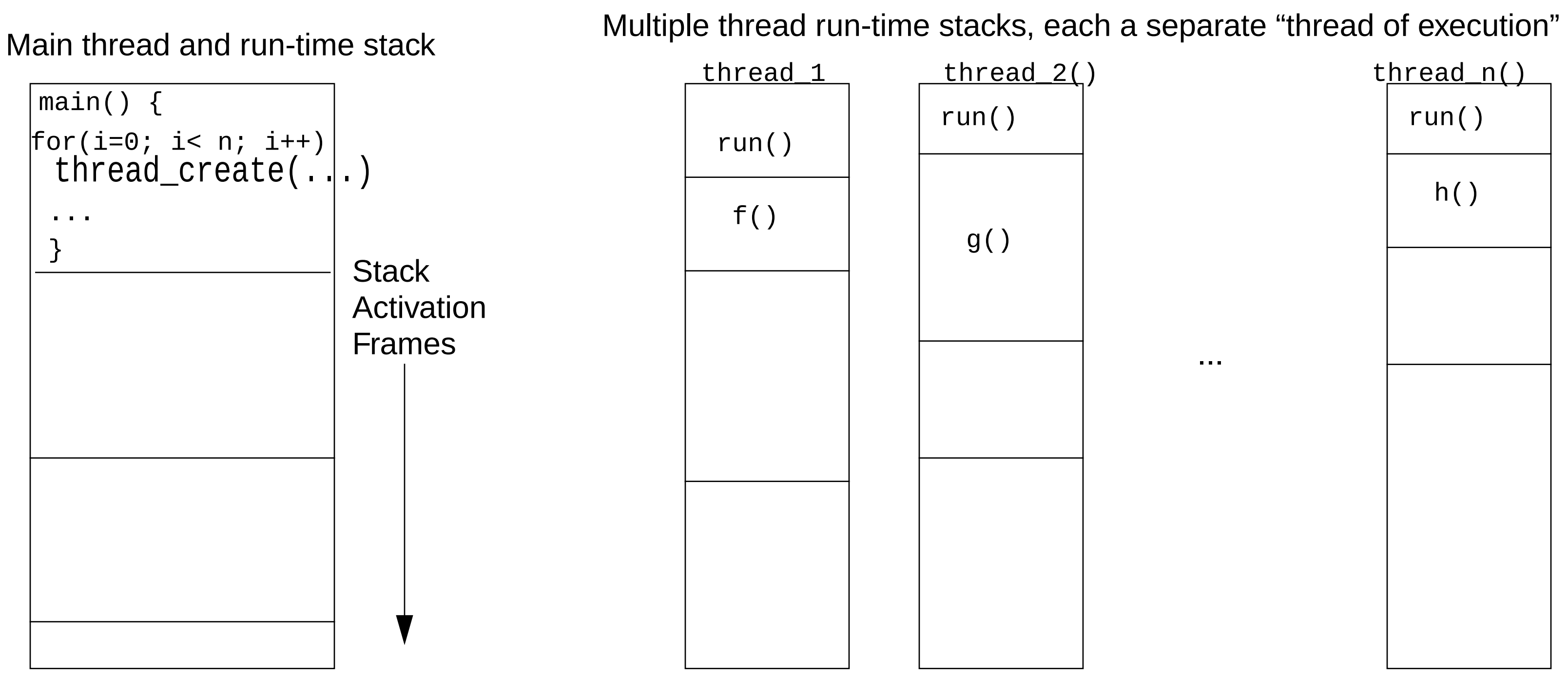
In a sequential program, the main run-time stack is allocated at program start and all procedure calls, including the initial call to “main” are made on this single run-time stack. In a multi-threaded program, a program starts on the system run-time stack where the main procedures runs. Any functions/procedures called by the main procedure have their activation frames allocated on this run-time stack.

If the main procedure creates a new thread for run some procedure (usually calling a special “thread creation or construction” procedure/method), then a new run-time stack is dynamically allocated from the heap and the activation frames for the procedures are allocated on this new stack.

Thread Stacks

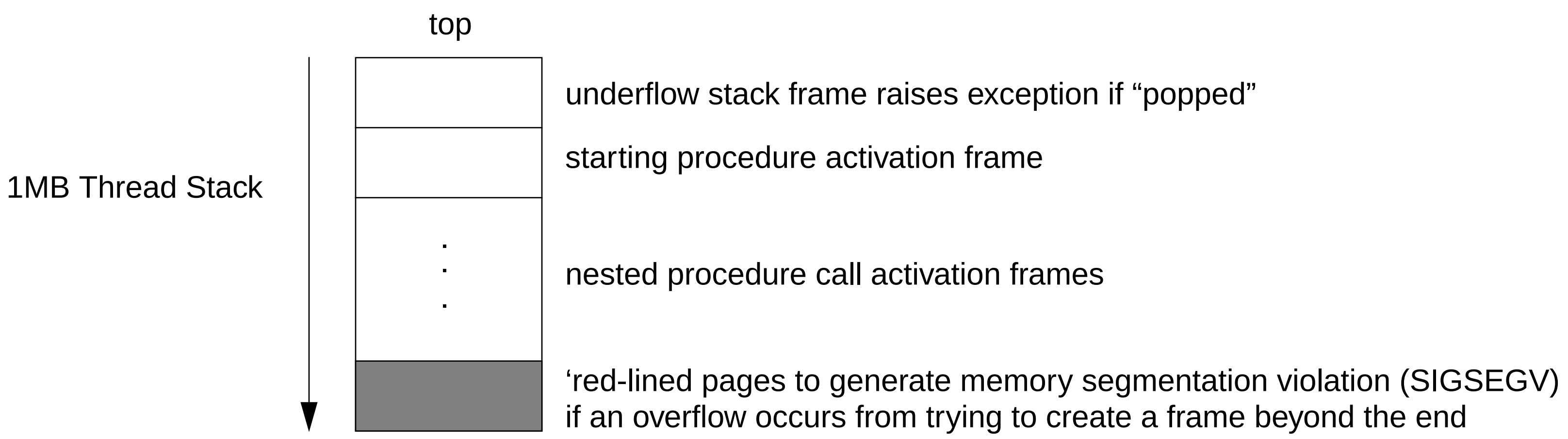
Question: How large should the heap allocated thread stack be?

A thread stack will contain the activation record of the “starting” thread procedure (e.g., called the “run” method in Java), as well as any procedures that are called by the procedure that was first started in the new thread of control. So, the thread stack needs to be large enough to hold the maximum number of bytes required to hold all the activation records of the deepest procedure call chain, as well as storage for all local variables allocated on the stack.



Thread Stack Size

On operating systems that support processes with multiple threads of control, threads stacks are typically set at 1MB, consisting of contiguous virtual memory pages, that are allocated incrementally at run-time by the system. There may also be some extra pages at the “top” or “bottom” for some thread book-keeping. There is also usually an extra pages allocated “above” the top and “below” the bottom of the stack to detect overflow:


top
1MB Thread Stack
underflow stack frame raises exception if “popped”
starting procedure activation frame
nested procedure call activation frames
‘red-lined pages to generate memory segmentation violation (SIGSEGV) if an overflow occurs from trying to create a frame beyond the end

Java Threads

In Java, the same concepts about threads, threads stacks, and starting threads applies. The primary difference is that in Java, a thread is defined as a special class in the java.lang package. The thread class implements an interface called the “Runnable” interface, which defines a single abstract method called “run”.

```
// in the java.lang.Runnable file you will find the following interface
package java.lang;

public interface Runnable {
   public void run();  // just like a pure virtual function in C++
}
```

Since we are defining an interface, run is implicitly a “abstract” method. As we have seen, interfaces in Java define a set of methods with NO implementation. Some class must “implement” the Runnable interface and provide an implementation of the run method, which is the method that is started by a thread. Java provides a “Thread” class, that implements the Runnable interface, but does not implement the run

```
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 MyOwnThreadObjects, bind it to a new Thread object, and then start it. Calling start creates the thread stack for the thread, and then invoked the run() method as the first procedure on that new thread stack.

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

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 ConcuirrentWriter();
writerThread.start(); // start calls run()
```

java.lang.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

```
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, thenwe 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. A **synchronized** method is one that “locks” an object so that no other thread can execute inside the object while the method is active.

```
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() as
```

Synchronized Methods, Wait, Notify, and NotifyAll

An very interesting features of Java objects is that they are all “lockable” objects. That is, the java.lang.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
}
```

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) { ... } ... }
```

Wait, Notify, and NotifyAll

Every object has acess to wait, notify, and notify methods, which are inherited from class Object.

```
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
    }
}
```