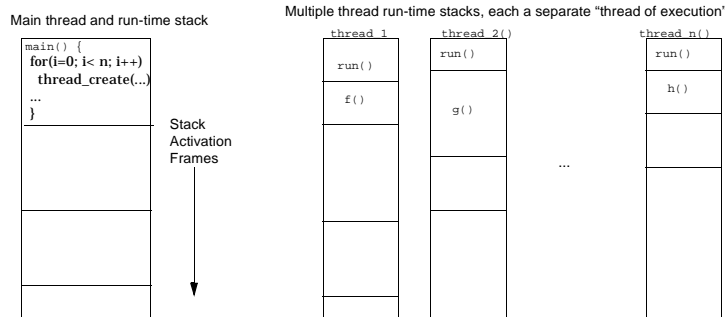


Concurrent Programming using Threads

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" procedure starts a thread.

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, 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 concurrently, each on a separate run-time stack



Example of POSIX Thread Creation in C/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 returns. A thread "dies" when the procedure that started the thread returns. The main procedure waits for each child thread to return by "joining" with it. (NOTE: On Solaris, you link with -lthread to use POSIX threads). Pthread_self() returns the current thread id.

Q: What happens to the dynamically allocated thread stack when a thread dies?

```
#include <pthread.h>

const int NUM_THREADS = 5;
const int SLEEP_TIME = 10;

void sleeping(int interval)
{
  printf ("thread %d sleeping %d seconds ...\n", pthread_self(), sleep_time);
  sleep (interval);
  printf ("\nthread %d awakening\n", pthread_self());
}

int main( int argc, char *argv[] )
{
  thread_t tid[NUM_THREADS]; /* array of thread IDs */
  for ( int i = 0; i < NUM_THREADS; i++)
    pthread_create (&tid[i], NULL, sleeping, (void*)SLEEP_TIME);
  for ( int j = 0; j < NUM_THREADS; j++)
    pthread_join (tid[j], NULL);

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

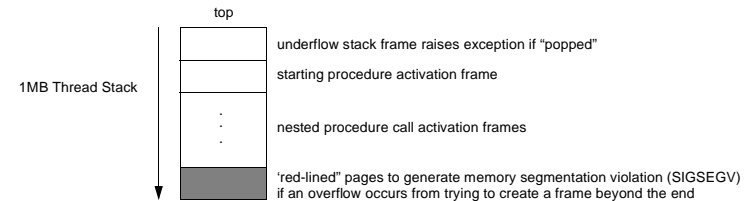
Thread Stacks

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 concurrent program, a program starts on the system run-time stack where the main procedure 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 some procedure (usually calling a special "thread_create" procedure), then a new run-time stack is usually dynamically allocated from the heap and the activation frames for the procedures are allocated on this new stack.

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.

In general, it is not tractable to know in advance how big each thread stack should be. 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. The allocation includes extra pages for some thread book-keeping information, and also special stack frames to detect stack underflow/overflow:



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 system class. 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

```
public interface Runnable {
  /**
   * The method that is executed when a Runnable object is activated. The run() method
   * is the "soul" of a Thread. It is in this method that all of the action of a
   * Thread takes place.
   */
  public void run(); // just like a pure virtual function in C++
}
```

Since we are defining an interface, run is implicitly an "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

```
// in java.lang.Thread
public class Thread implements Runnable {
  ...
}
```

If you want to create a thread, you **extend** the Thread class and **implement** the run() method in your specialized thread class. The run method is invoked by calling a special start() method on an object of type Thread.

Defining a Thread in Java

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

To start this thread you need to do the following:

```
Actor a = new Actor();
a.start();
```

Another way to create a thread is by using the Runnable interface. This way any object that implements the Runnable interface can be run as a thread. For example:

```
class MyOwnThreadObject implements Runnable {
    public void run() {
        ... // compute something concurrently
    }
}
```

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.

```
MyOwnThreadObject thread = MyOwnThreadObject();
Thread t = new Thread(thread); // create a thread and initialize it with aRunnable object
t.start();
```

The java.lang.Thread class has a constructor as follows, which takes an object of type Runnable.

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

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 is a type of *synchronization policy* called **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 reacquire 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 state must result in consumable condition being true
    notifyAll(); // notify all waiting threads to try consuming
}
```

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. Which thread actually gets the lock is **non-deterministic** and also 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. *Sort of like trying to register for CS classes-everyone has a chance, but the scheduling process is not "fair"!*

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?

We know that Java only allows single class inheritance, so if we have a class that needs to inherit from another class, but also needed to run as a thread, then it would make sense to extend the other class and implement the Runnable interface. In other words, the fact that an object has to run as a separate thread of control is really a separate issue from the type inheritance relationships that may exist among classes in your program.

So, it is quite common for a class X to extend another class Y and implement the Runnable interface, like so:

```
class X extends Y implements Runnable {
    ... // private data
    ... // constructors and other methods

    public synchronized void do_something() { ... }

    public void run() { do_something(); } // can be run as a separate 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
```

A Shared Queue Example

This is an example of a "Producer-Consumer" shared resource. Consumers wait until something is in the queue that they can consume, and producers insert things into the queue and notify the waiting consumers that there is something to consume. 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 { // wait could be interrupted
            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
    }
}
```