GUIs and multithreading

Michelle Kuttel

Single-threaded GUIs

GUI applications have their own peculiar threading issues

Nearly all GUI toolkits **are single threaded subsystems**

– Jave, Qt, MacOS Cocoa, X Windows…

- This means that all GUI activity is **confined to a single thread**
    - event dispatch thread (EDT) handles GUI events

GUI objects are kept consistent **not by synchronization**, but **by thread confinement**.

Why are GUIs single threaded?

The many attempts to write multithreaded GUI frameworks were plagued by race conditions and deadlock

- deadlock because of interaction between input event processing and object-oriented modelling of GUI components:
  - actions from the user “bubble-up” from OS to application
  - application-initiated actions “bubble-down” from application to OS

Why are GUIs single threaded?

Tendency for activities to access the same GUI objects in opposite order + locks required for thread safety = **inconsistent lock ordering**

Recipe for deadlock!

fundamental conflict here between a thread wanting to go "up" and other threads wanting to go "down",

- while you can fix individual point bugs, you can't fix the overall situation.

Confirmed by the experience of nearly every GUI toolkit development effort

Why are GUIs single threaded?

Other source of deadlock is prevalence of Model-View-Controller (MVC) pattern

- simplifies implementing GUI designs
- prone to inconsistent lock ordering:
  - controller calls into model, which notifies the view that something has changed
  - controller can also call view, which may call back into the model to query the model state

In his weblog,[1] Sun VP Graham Hamilton nicely sums up the challenges, describing why the multithreaded GUI toolkit is one of the recurring "failed dreams" of computer science.

[1] http://weblogs.java.net/blog/kgh/archive/2004/10

I believe you can program successfully with multithreaded GUI toolkits if the toolkit is very carefully designed; if the toolkit exposes its locking methodology in gory detail; if you are very smart, very careful, and have a global understanding of the whole structure of the toolkit. If you get one of these things slightly wrong, things will mostly work, but you will get occasional hangs (due to deadlocks) or glitches (due to races). This multithreaded approach works best for people who have been intimately involved in the design of the toolkit.

Unfortunately, I don't think this set of characteristics scales to widespread commercial use. What you tend to end up with is normal smart programmers building apps that don't quite work reliably for reasons that are not at all obvious. So the authors get very disgruntled and frustrated and use bad words on the poor innocent toolkit.

Why are GUIs single threaded?

- all eventually arrived at a single-threaded event queue model, where a dedicated thread fetches events off a queue and dispatches them to application-defined event handlers
- achieve thread safety via thread confinement:
  - all GUI objects, including visual components and data models, are accessed exclusively from the event thread.

Single-threaded GUIs

- Of course, this just pushes some of the thread safety burden back onto the application developer, who must make sure these objects are properly confined.
- e.g. For safety, certain tasks must run in the Swing event thread
  - Swing data structures are NOT thread safe, so must be confined here

Sequential Event Processing

GUI applications are oriented around processing fine-grained events:

- mouse clicks, key presses, or timer expirations.

Events are a kind of task:

the event handling machinery provided by AWT and Swing is structurally similar to an **Executor.**

Sequential Event Processing

Task are processed **sequentially:**

- one task finishes before the next one begins
- no two tasks overlap.

Upside for programmer:

- you don't have to worry about interference from other tasks.

Downside for user:

- if one task takes a long time to execute, other tasks must wait until it is finished. If those other tasks are responsible for responding to user input or providing visual feedback, the application will appear to have **frozen.**

Sequential Event Processing

Tasks that execute in **the event thread** must return control to the event thread **quickly.**

A long- running task

- spell-checking a large document, searching the file system, or fetching a resource over a network, -

must run **in another thread** so control can return quickly to the event thread.

To update a progress indicator while a long-running task executes or provide visual feedback when it completes, you again need to execute code in the event thread.

This can get complicated quickly!

Sequential Event Processing

Tasks that execute in **the event thread** must return control to the event thread **quickly.**

A long- running task

- spell-checking a large document, searching the file system, or fetching a resource over a network, -

must run **in another thread** so control can return quickly to the event thread.

To update a progress indicator while a long-running task executes or provide visual feedback when it completes, you again need to execute code in the event thread.

This can get complicated quickly!

Thread confinement in Swing

Swing single-thread rule:

- Swing components and models should be created, modified, and queried only from the event-dispatching thread.

Thread confinement in Swing

All Swing components

- JButton and JTable

and data model objects

- TableModel and TReeModel

are confined to the event thread.

Any code that accesses these objects must run in the event thread.

Thread confinement in Swing

Upside:

- tasks that run in the event thread need not worry about synchronization when accessing presentation objects

Downside :

- you cannot access presentation objects from **outside the event thread at all.**

Thread confinement in Swing: exceptions to the rule

- A small number of Swing methods may be called safely from any thread; these are clearly identified in the Javadoc as being thread-safe.
- Other exceptions to the single-thread rule include:
- SwingUtilities.isEventDispatchThread
  - determines whether the current thread is the event thread;
- SwingUtilities.invokeLater
  - schedules a Runnable for execution on the event thread
  - callable from any thread
- SwingUtilities.invoke And Wait
  - schedules a Runnable task for execution on the event thread and blocks the current thread until it completes (callable only from a non-GUI thread);
- methods to enqueue a repaint or revalidation request on the event queue (callable from any thread);
- methods for adding and removing listeners (can be called from any thread, but listeners will always be invoked in the event thread).

Long-running GUI tasks

- simple short-running tasks can stay entirely in the event thread
  - if all task are short-running, you don’t need threads at all
- for longer running tasks, some of the processing should be off-loaded to another thread
  - spell-checking, background compilation etc.

Long-running GUI tasks

- Can create an Executor to help with long-running tasks

```
ExecutorService backgroundExec = Executors.newCachedThreadPool();
...
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      backgroundExec.execute(new Runnable() {
         public void run() { doBigComputation(); }
      });
}});
```

"fire and forget" example

Long-running GUI tasks

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setEnabled(false);
        label.setText("busy");
        backgroundExec.execute(new Runnable() {
            public void run() {
                try {
                    doBigComputation();
                } finally {
                    GuiExecutor.instance().execute(new Runnable() {
                        public void run() {
                            button.setEnabled(true);
                            label.setText("idle");
                        }
                    });
                }
            }
        });
    }
});
```

visual feedback example

Cancellation and Shutdown

Michelle Kuttel

Cancellation and shutdown

Starting tasks and threads is easy and most of the time we let them stop by themselves

Sometimes we want to stop tasks earlier, e.g.:

- when a user cancelled an operation
- when the application needs to shut down

Here we talk about techniques for convincing tasks and threads to terminate prematurely.

elegant shutdown is a factor that defines a truly **robust** concurrent application

Deprecated thread primitives

- Not easy to get threads to stop safely, quickly and reliably
  - Thread.stop and Thread.suspend and Thread.resume were an attempt at doing this
  - now deprecated, as too dangerous

java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

Why is Thread.stop deprecated?

- Because it is inherently unsafe.
  - Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the ThreadDeath exception propagates up the stack.)
  - If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state.
    - Such objects are said to be *damaged*.
  - When threads operate on damaged objects, arbitrary behavior can result.
  - Unlike other unchecked exceptions, ThreadDeath kills threads silently; thus, the user has no warning that the program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future.

Why are Thread.suspend and Thread.resume deprecated?

Thread.suspend is inherently deadlock-prone.

- If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed.
- If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, deadlock results.
- Such deadlocks typically manifest themselves as "frozen" processes.

Java

Java does not now provide any mechanism for forcing a thread to stop

- instead, ask the thread to stop what it is doing through *interruption*
- **cooperative approach**
  - don't want a thread to stop immediately, since that could leave shared data and structures in an inconsistent state
  - allow threads to clean up work currently in progress and then terminate

Task cancellation

An activity is *cancellable* if external code can move it to completion before its normal completion.

Examples of Task cancellation

User-requested cancellation. e.g. pressing cancel button

Time-limited activities. e.g. an optimization must terminate after a certain period, after which the best

Application events. e.g. where different tasks are searching a problem space, when one task finds a solution, all must terminate

Errors. e.g. disk is full and all tasks must terminate.

Shutdown. when an application is shutdown, work currently being processed must be dealt with (either allowed to complete, or cancelled).

Methods for cancellation

There is not safe way to preemptively stop a task in Java

- there are only cooperative mechanisms, where the task and the code requesting cancellation follow an agreed-upon protocol.

- e.g. set a “cancellation requested” flag, which is checked periodically.
  - NB: for this to work, the flag must be **volatile**.

Cancellation policies

Specify the “how”, “when” and “what” of cancellation.

- how other code can request cancellation
- when the task checks that cancellation has been requested
- what action the task takes in response to a cancellation request

Limitations of flags

Using flags for cancellation in combination with a blocking method can cause a task to run for ever

e.g. BlockingQueue.put

- if producer blocks because queue is full, and consumer subsequently sets flag, then cancels itself, producer will never come out of suspension

Interruption

- Java provides a cooperative interruption mechanism that can be used to facilitate cancellation
  - but it is up to you to construct protocols for cancellation and use them effectively
- Using FutureTask and the Executor framework simplifies building cancellable tasks and services.

Interruption

- Each thread has a boolean *interrupted status*
  - interrupting a thread sets it status to true

Interruption

`Thread` provides the methods:

`void` **`interrupt`**`()`

Interrupts this thread.

`static boolean` **`interrupted`**`()`

Tests whether the current thread has been interrupted. The *interrupted status* of the thread is cleared by this method.

`public boolean` **`isInterrupted`**`()`

Tests whether this thread has been interrupted. The *interrupted status* of the thread is unaffected by this method.

Interruption

Interruption is a cooperative mechanism:

- thread A requests that thread B stops when convenient, it cannot force it to stop.

Blocking methods like Thread.sleep and Object.wait try to detect when a thread has been interrupted and return early

- but the JVM makes no guarantees on how quickly this will happen

Interruption

If a thread is interrupted when it is not blocked, it is up to the cancelled activity to poll the interrupted status to detect interruption

- calling interrupt on a target thread merely delivers the message that interrupt has been requested

Interruption is “sticky” – it persists until it is cleared

Interruption is usually the most sensible way to request cancellation

```
public void run()
{
   try
   {
      . . .
      while (!Thread.currentThread().isInterrupted())
      {
         do more work
      }
   }
   catch(InterruptedException e)
   {
      // thread was interrupted during sleep or wait
   }
   finally
   {
      cleanup, if required
   }
   // exiting the run method terminates the thread
```

Interruption policies

Determines how a thread interprets an interruption request

- usually: exit as quickly as possible, cleanup where necessary
- could be: pausing or resuming a service