Lecture 23: Threads & Timers

UI Hall of Fame or Shame?

Here's our hall of fame or shame example. StatCounter is a web site that tracks usage statistics of your web site, using a hit counter. This is a sample of its statistics page, which shows you how many visitors your site had over a certain period of time.

To help focus discussion, let's think about:

- color design

- visual variables

- the controls underneath the graph

Today's Topics

- Long-running tasks
- Threads
- Timers
- Event loop hacks

Today's lecture is another in our series about implementing UIs. We'll be talking about how to deal with long-running system tasks – i.e., computation or I/O that takes more than a short time to do, so you can't simply do it in response to a user input event. We'll look at three techniques for implementing these tasks: threads, timers, and event loop hacking, and discuss their applicability, advantages, and disadvantages.

Long-Running Tasks

- Some system tasks run for a long time
  - Getting a URL, running a database query, compiling a program
- Event handlers should return quickly
  - Because repainting and input handling is blocked
- But everything in a GUI is triggered by an event handler
  - e.g., pressing the button that starts the task
- So long tasks need to run in the background

First, some motivation. Why do we need to do background processing in graphical user interfaces? Even though computer systems are steadily getting faster, we're also asking them to do more. Many programs need to do operations that may take some time: retrieving URLs over the network, running database queries, scanning a filesystem, doing complex calculations, etc.

But graphical user interfaces are event-driven programs, which means (generally speaking) everything is triggered by an input event handler. For example, in a web browser, clicking a hyperlink starts loading a new web page. But if the click handler is written so that it actually retrieves the web page itself, then the web browser will have very poor usability. Why? Because its interface will actually **freeze up** until the click handler finishes retrieving the web page and returns to the event loop. It won't repaint any damage to the window, and it won't respond to any further user input. Most importantly, even if it displays a Stop button or a Cancel button for interrupting the download, the button will be completely unresponsive while the click handler is running.

So long tasks need to run in the background. What's a "long task"? Certainly anything shorter than the perceptual fusion interval, 100 msec, can be run directly in an event handler. But tasks longer than 1-5 sec should be cancellable. If a task needs to be cancelled, then it has to be run in the background, allowing the event loop to continue running so that the user can actually cancel it.

Know Your Task's Bottleneck

- Input from user
  - Most desktop and web UIs spend almost all their time waiting for the user to do something
- Output to user
  - Animation
- CPU/memory
  - Compute-bound processing
- I/O
  - Network, filesystem, devices

Before discussing the techniques for background processing, it's important to be aware of what kind of task you're running. What is its performance bottleneck – i.e., how does it spend most of its time?

For most GUIs, the bottleneck is actually **user input**. The program spends most of its time sitting in the event loop, waiting patiently for the user to do something. When the user does provide some input, the program's event handlers fire, compute a change, and repaint the result. Then the program sits and waits again. Users are so slow relative to the computer system that this kind of system doesn't need any background processing; it can do everything in the event loop.

When **user output** is the bottleneck, the program spends most of its time drawing on the screen, because the display is changing independently of the user's input. 3D games, video players, and animation-heavy programs typically face this problem. We'll talk more about animation in a future lecture.

The other two bottlenecks are the most relevant to today's lecture. **CPU** or **memory** might be a bottleneck for systems that need to do a lot of local processing. Searching for the shortest path through an enormous graph might be a compute-bound task, for example. **I/O** might be a bottleneck for tasks that need to read or write a lot of data from the network or filesystem, or interact with a device like a CD burner. For I/O-bound tasks (just like user-input-bound tasks), the CPU spends most of its time waiting – e.g., waiting for a remote web server to respond, or waiting for more bytes to arrive across the network, or waiting for a disk to seek to the right track.

Note that many tasks might involve both CPU work and I/O; what we care about is which one is the *bottleneck*. If a task involves a small file from the local disk and then proceeds to decrypt it using an expensive decryption algorithm, then overall the task is CPU-bound. If, on the other hand, the task involves fetching a large file over a distant network connection and then unzipping it with a fast decompressor, then overall the task is probably I/O-bound.

Goals

- Handle UI input and output constantly
  - Respond to input events within 100 msec
  - Handle repaints too
- Display task progress
  - e.g. progress bar
- Allow task to be canceled
  - Cancel button
- Don't hog the CPU

Here are the goals we want to satisfy when we implement handling for long tasks.

First, while the task is running, we want to handle UI input and output constantly. The event loop should keep spinning, reading input events from the event queue, dispatching them to event handlers, and handling repaint requests to keep the display up to date. To preserve good response times, we should make sure input events don't sit on the queue unhandled for very long – 100 msec would be the upper limit on the age of an event before it's handled.

Second, if the long task was requested by the user, or if the user is waiting for its results, then we should display its progress to the user (or at least evidence that the task is still alive and kicking). It should also be possible to cancel the task.

(Note that some long tasks may be housekeeping tasks, which are supposed to happen invisibly. Autosave is an example; garbage collection is another. These tasks don't need to display their progress and don't need to be explicitly cancelable, assuming that they really *are* invisible as far as the user interface is concerned, not modes that inhibit the user's actions while they're running.)

Finally, the task shouldn't hog the CPU unnecessarily. Some poor approaches to background processing cause the CPU to **busy-wait** on an I/O-bound task. The cost of busy-waiting is twofold: CPU resources are wasted that could have been spent on other processes (like playing MP3s in the background), and the CPU uses more power (if it could have been sleeping and saving your laptop's battery life).

Threads

- Run long tasks in a background thread
  - Event handler starts a new thread and returns immediately
  - Background thread does the work while event-handling thread continues to handle user I/O

Our first technique for handling long tasks is seemingly simple, but fraught with peril for the unprepared: using another **thread**. (If you're not clear about what threads are and how to use them in Java, see the Concurrency lesson in the Java Tutorial, http://java.sun.com/docs/books/tutorial/essential/concurrency/.)

The event loop runs in an **event-handling thread**. When an event handler needs to do a long task, it creates a new thread, starts it running, and then returns immediately to the event loop, so that further UI input and output continues to be handled. Meanwhile, the newly-created thread does the long task in the background.

Advantages of Threads

- Easy to write (seemingly)
- Ideal for I/O-bound tasks
- Can use lower-priority thread for CPU-bound tasks

Threads are appealing because they're apparently simple to write. In theory, the code inside the thread's body looks exactly the way it would if you had written it in the event handler itself (assuming it were fast enough to run there).

They're also ideal for I/O-bound tasks, in which the CPU spends most of its time waiting. Assuming the I/O code is written properly (using **blocking** calls, which we'll define later in this lecture), the background thread will simply be suspended while it's waiting for I/O, and the CPU can run other threads instead (or even go to sleep to save power). Incidentally, the event handling thread of a GUI spends most of its time suspended, waiting for a user input event to occur.

Even for CPU-bound tasks, threads are a good idea. CPU-bound threads don't wait (they usually saturate the CPU, in order to finish the task as fast as possible), but they can be given lower scheduling priority than the event-handling thread, so that when an input event *does* finally arrive, the event-handling thread can preempt the background thread and handle the input event immediately. So input events won't languish on the queue.

Disadvantages of Threads

- Need to protect shared mutable data against race conditions and deadlocks
  - Using locks (in Java, synchronized keyword)
  - This is much much harder than it looks
- Many GUI toolkits are not threadsafe
  - Including Java Swing
  - You **cannot** safely touch (read or write) any Swing object or call any Swing method from anywhere but an event handling thread

Unfortunately the apparent simplicity of threads hides subtle dangers: race conditions and deadlocks. A race condition happens when two threads read and write the same shared data structure, causing inconsistencies and corruption. Race conditions are solved by adding synchronization (locks) to the data structure, but this introduces the risk of deadlock: e.g., two threads each waiting for each other to release a lock.

Unfortunately there's very often a shared mutable data structure in your GUI: the **model**. If you use background threads, you have to make sure your data structure is **threadsafe** (able to be accessed from multiple threads). This is much harder than it looks. In Java, sprinkling synchronized keywords around willy-nilly is not a principled or reliable way to do it.

Another important shared mutable data structure in your GUI is the **view hierarchy**. Some toolkits work hard to make it threadsafe (.Net, AWT), but many do not. Java Swing is not threadsafe. In general, you cannot safely call methods on a Swing object from anywhere but the event-handling thread. (See http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html for an explanation of this decision from Swing's designers.)

So if you're using Swing, your background thread:

- can't read the model, because of race conditions with the event-handling thread

- can't read or write the view hierarchy, because Swing isn't threadsafe

- can't write the model, not only because of race conditions, but also because those writes might trigger model-change events, which are usually handled by views or controllers, which would need to read or write the view hierarchy.

What to do?

A Simple Approach to Safe Threading

- Don't share any mutable data between threads
  - So don't touch the model in the background
- Do all GUI stuff in the event-handling thread
  - Use SwingUtilities.invokeLater()
- Bundle up model changes and pass them to the event-handling thread

If you want to use threads for background processing in Swing, it's best to follow a **careful discipline**. Here's a simple discipline that will be safe.

First, don't share **any** mutable data between threads. Immutable objects are OK; they can't suffer race conditions, because all you can do is read them. But don't touch mutable model objects from the background thread, either for reading or writing. Any mutable objects that the background thread uses should be created and used only by that thread, not shared with any others.

Second, do all your GUI access in the event-handling thread. When a background thread needs to make a change to the GUI – e.g., updating its progress bar – then it should force that code to be run by the event-handling thread. Single-threaded toolkits like Swing generally provide some mechanism for doing this. Swing has a method SwingUtilities.invokeLater() which takes a Runnable object and runs it (as soon as possible) in the event-handling thread. This method actually works by dropping a pseudoevent on the event queue, with your Runnable code attached to it as an event handler, so that when the event loop pops the pseudoevent off the queue, it runs your code. When a background thread needs to read data from GUI objects, you can use SwingUtilities.invokeAndWait(), which doesn't return until the event-handling thread has finished running the Runnable.

The background thread can use the same techniques to read and write the model. The essence of this discipline:

The model and the view hierarchy belong to the event-handling thread, and should be touched by no other threads.

Since this kind of communication between threads can be expensive (switching between threads can cost 100 times more than a simple procedure call), you should bundle up changes. For example, if you're downloading a file from the Web and loading it into a textbox, don't send one character at a time through invokeLater(). Send it in big chunks.

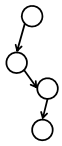
Making Your Model Threadsafe

- Synchronize carefully
  - Coarse-grained locking is simple and reliable, but blocks the event-handling thread

```
synchronized (model) {
    // all code that reads and writes model
}
```

  - Fine-grained locking may have race conditions or deadlocks

```
class Node {
    synchronized void setParent(Node n) {...}
    synchronized void setChild(Node n) {...}
    synchronized Node getChild(int i) { ... }
    ...
}
```



Unfortunately this simple discipline may be too constraining. If you can't read or write your model in the background thread, then a lot of useful long-running tasks are ruled out. How would you implement autosave, for example, if you couldn't touch the model? You'd have to make a copy of the model *before* starting the background thread. Making a copy of a large model might be more expensive than the task itself, and it would have to happen in the event-handling thread, since that's the only thread that's allowed to touch the model in the simple discipline we've just described. So putting the task in the background would be pointless.

So the next level above the simple discipline is **making your model threadsafe**, so that at least the model can be read and written directly from background threads, even if the view hierarchy can't.

The first step is adding synchronization (locks) to your model objects, so that they're safe to read and write from multiple threads. This is harder than it looks, and it's beyond the scope of this class to cover how to do it well. (The most we can really convey is a sense of peril.) But here's the essence of the problem: most model data structures consist of multiple objects, so when you design your locking strategy, you need to decide on the granularity of locking. Fine-grained locking, with one lock per object, provides the most parallelism and the least exclusion, but is also most susceptible to race conditions and deadlocks. Coarse-grained locking, with one lock for the entire model, is likely to be too exclusive; it will actually freeze the UI from updating while the background task is holding the lock.

For example, suppose the model is a tree. If there's only one lock on the tree, and the background thread is holding it while it's traversing the tree making changes (or perhaps just autosaving), then the event-handling thread will block – **freeze** – if it tries to access the model, even just to repaint. That's bad. If the background thread prevents the event-handling thread from running, then we might as well have run the long task in the event-handling thread anyway!

But if each node has its own lock, then it's possible to have race conditions or deadlocks. For example,

code that removes a node from the tree has to update two nodes: the node itself (to set its parent field to null) and its parent (to remove the child from its list of children). If this code acquires the locks for the nodes separately, then a race condition results (i.e., another thread could interrupt after the first node has been updated but before the second node has been changed, seeing an inconsistent tree). If the code acquires and holds the locks for both nodes while it's updating both nodes, then it must acquire them in a consistent order, or deadlock may result. Actually, deadlock can easily happen on a tree with individual node locks: suppose one thread is doing a depth-first search (which acquires locks from the root downward) and another thread is searching for the least common ancestor of two leaves (which acquires locks from the leaves upward). If those threads ever meet at a node, they'll deadlock, waiting for each other to release the locks they're holding.

Making Your Model Threadsafe

- Use a database and transactions instead of locks
- Fire events in the event-handling thread
  - Since views and controllers are listening for those events, and they're probably Swing objects

It's usually far easier to synchronize your model if it's stored in a database. If you surround your updates by transactions to keep them consistent, then you can achieve thread safety with much less pain.

However you secure your model, make sure that it communicates appropriately with the views and controllers listening to it, which are either Swing objects themselves or need to call lots of Swing objects. Make sure that code runs on the event-handling thread, using invokeLater().

Timers

- Break task into little chunks and run them periodically with a timer
  - Usually used for animation
  - Also useful for network I/O, but be careful not to block
- Timer events happen on event-handling thread
  - So generally no synchronization problems
  - But timer handler should return quickly (< 100ms)
  - Use javax.swing.Timer, not java.util.Timer
  - HTML/Javascript: setTimeout() or setInterval()

Hopefully we've made the point that using threads in GUI programming isn't as simple as it looks. **Timers** are an alternative technique that uses nothing but the event-handling thread, so that you needn't fear race conditions and deadlocks. A timer is an object that periodically fires an event handler. So if a long-running task can be broken up into small chunks that run in less than 100 msec (in order to keep the event-handling thread responsive), then a timer may be a simple solution.

Timers are frequently used for animation, as we'll see in a future lecture. They're also useful for I/O-bound tasks as long as I/O is buffered (as it is on every desktop OS), because the operating system can be filling the buffer while the event loop is waiting for the timer to fire. Once the timer fires, it will find data in the buffer ready to process; it processes it, and then goes back to waiting while the buffer fills up some more. As long as the buffer is large enough (or the timer interval is short enough) that the buffer never fills up and stalls the I/O transfer, then this scheme will have the same throughput as a background thread.

But timers tend to slow down CPU-bound tasks. (Why?)

Since timer events run on the event handling thread, there are generally no synchronization problems – as long as, of course, each chunk of the task makes sure that any data structures it touches are consistent (i.e., all invariants are satisfied) before it returns to the event loop.

Be careful of a pitfall here: Java has two different timer classes, javax.swing.Timer and java.util.Timer. **javax.swing.Timer** is the one you want for this technique, because it will call your handler on the event-handling thread. java.util.Timer creates a new thread to call your handler, so if you use that class, you face all the problems with threads we just discussed.

Another advantage of timers is that they're almost universally supported by GUI toolkits, even on platforms that don't support threads at all. HTML/Javascript has no notion of threads, but it has timers: setTimeout() runs a function once after a given time has elapsed, and setInterval() runs a function repeatedly with a given period.

Listening for Idle Time

- Some toolkits let you register an event handler for idle time
  - Called whenever there are no other events waiting
  - Useful for CPU-bound tasks if they can be easily chopped up into chunks
- Can often by simulated by zero-duration timer

```
function idler() {
    // do idle code
    setTimeout(idler, 0)
}
```


Another way to run a long task in the event-handling thread is **idle time**. Some toolkits let you register an event handler that gets called when the event loop is idle, i.e. there are no other events waiting. Microsoft Foundation Classes (MFC) calls this the onIdle event.

Idle handling is very useful for running CPU-bound tasks in the event-handling thread, assuming they can be easily chopped up into small chunks so that the event loop periodically gets a chance to handle input and repaints.

Swing has no idle handling, although it can be simulated to some extent by using invokeLater() to drop idle handlers at the end of the event queue.

On HTML/Javascript, idle handling can be simulated by "zero-duration" timers. Ideally, a zero-duration timer should fire immediately, putting the event on the queue immediately, just like invokeLater() does. Alas, despite the prevalence of this idiom among Javascript programmers, current browsers don't really implement it that way. They actually create a timer with the minimum resolution supported by the OS (typically 10-15 ms). So you pay something for each zero-duration timer. If you've divided your work into 100 ms chunks, then the overhead of this technique is an additional 10%. But, alas, there's no other way to do it in HTML/Javascript at the moment. (http://lazutkin.com/blog/2008/mar/23/javascript-edp-and-0ms-timeouts/)

I/O Interfaces

- Blocking calls wait until data is available to read (or buffer has space to write)
  - InputStream.read()
  - EventQueue.getNextEvent()
- Nonblocking calls return failure immediately
  - InputStream.available()
  - EventQueue.peekEvent()
  - **Polling**: looping around nonblocking call until it succeeds
- Asynchronous calls notify an event handler when data is ready
  - java.nio package
  - XMLHttpRequest in HTML/Javascript

Let's return to thinking about I/O, since network access is such an important part of modern applications. Programming interfaces for I/O come in three flavors. **Blocking** calls wait until data arrives (for a read) or the outgoing buffer has free space (for a write). Blocking stalls the whole thread while waiting for the I/O to happen. Most of the I/O code you've probably written used blocking calls, because it's easiest to write.

Nonblocking calls return immediately whether the read/write can be done or not. If data is available, a nonblocking read returns it; otherwise it returns failure. So doing nonblocking I/O may involve **polling**, in which you try the call repeatedly in a loop until it succeeds. InputStream.available() is an example of nonblocking I/O; it doesn't strictly do a

read, but it tells you whether a read will block or not.

Finally, **asynchronous** calls are like triggering a background thread for doing the I/O. The function call returns right away, regardless of whether the data is ready or not. Once the data is ready, a callback function (that you provided) is called to notify you. The java.nio package in recent versions of Java provides asynchronous I/O. So does XMLHttpRequest in HTML/Javascript.

Blocking vs. Polling

- Blocking
  - appropriate for thread-based tasks
  - best for polite CPU sharing
  - best for performance (I/O latency and throughput)
  - bad inside in an event handler
- Polling
  - appropriate for timer-based tasks
  - may be less polite for CPU sharing
  - may have lower performance

Blocking I/O is appropriate for thread-based programming. It's simpler to code, because you can write your code as if the data is always ready and waiting for you. Blocking I/O automatically suspends the thread making the call until the I/O is ready, which leads to efficient thread scheduling and good I/O performance. Blocking I/O is polite: other threads on the computer can run at full speed while your thread is sitting patiently blocked on its I/O. (Or the CPU can go to sleep, if there are no other threads that need to run, saving energy.)

But blocking I/O is risky inside the event-handling thread. If you're using a blocking call to read from a network connection, and that network connection hangs (say, because the network goes down), then your whole UI freezes.

Conversely, nonblocking I/O is more appropriate for timer-based tasks, which run in the event-handling thread and wake up periodically to process some I/O. The timer event uses nonblocking I/O to consume data that has already arrived, but it doesn't block to wait for more. Instead, it returns to the event loop and waits there until the next timer event fires.

The event loop itself blocks if there are no events, so the event-handling thread suspends politely during this wait. But the cost of timer-based polling is that it introduces artificial delays into I/O handling. For example, if the timer interval is 50 msec, then even if data is ready after only 10 msec, the timer will waste an extra 40 msec before it gets

around to handling it. Trying to reduce this latency by increasing the timer frequency increases busy-waiting – the CPU wastes time repeatedly activating the thread to check for I/O. (With blocking I/O, by contrast, the thread costs no CPU time while it's waiting, and it can wake up almost immediately, in less than a microsecond.)

Hacking the Event Loop

- Maintain UI reponsiveness by periodically polling the event queue
  - Essentially creates a new event loop inside the main event loop
  - Modal dialogs and synchronous XMLHttpRequest do this
- Merge long-running I/O tasks into the main event loop
  - Block on both user input and other I/O simultaneously
  - Single-threaded web browsers do this

Sometimes, the best (or only) way to integrate user interface handling with long-running tasks is by changing the event loop.

Suppose you need to run a long task in the event-handling thread, and you can't break it into timer chunks. One solution is periodically polling the event queue inside your long task. In other words, every so often, your long task checks whether there are any events on the queue, and if so, it dispatches them. (In Java, you can obtain a reference to the event queue with Toolkit.getSystemEventQueue()). As long as you poll the queue faster enough (every 50-100 ms), you can keep the user interface reasonably responsive. This technique is often called **pumping the event queue**.

Modal dialog boxes effectively do this. The method call that shows a modal dialog box doesn't return until the user has dismissed the dialog, so it's effectively a long task. It avoids freezing the UI by creating its own event loop that pumps the event queue. The synchronous version of XMLHttpRequest in HTML/Javascript does something similar while it's waiting for an I/O to complete.

Another event-loop hack is to combine long-running tasks into the main event loop. This only works for GUI toolkits where the main event loop is under your control, of course, so it can't be done in Swing or HTML, but can be done in SWT (another Java toolkit) and low-level Windows and X programming. If you control the main event loop, then you control what happens during idle time. You can also block on multiple input

channels simultaneously – specifically, both the event queue and the I/O you're trying to do – using OS-specific mechanisms (like select() on Unix or WaitForMultipleObjects on Windows). Before threads were widely supported on Unix, this is how most Unix web browsers had to be written.

Lazy Computation

- Lazy evaluation means delaying parts of the long task until they're actually needed
  - Imagine a scrolling table showing 100,000 database records
  - Just load 100 (or a screenful)
  - As the user scrolls, load more screenfuls on demand
- Works well when the time to compute each chunk is short and predictable (< 100 ms)

One lesson from this lecture should be that long-running tasks in a GUI are tricky. So if you can structure your system to avoid them, so much the better. **Lazy computation** is one such technique. Rather than doing a large computation or I/O in the background, only do the part that the user can actually see, e.g., the part that's scrolled into view. When the user scrolls to another part, compute that part. This technique ensures that all computation happens on one thread (since it's fired by user input events, i.e. scrolling) and is very polite with CPU resources (since user input happens so rarely and the program does nothing between input events). But it obviously requires each chunk to be fast to compute, so that the system has good response time when the user scrolls.

Summary

- Long tasks must still keep the UI alive
- Know whether your task is CPU or I/O bound
- Threads require careful synchronization
  - Especially in non-threadsafe toolkits like Swing
- Timers are very safe but force chopping up the task and using polling for I/O