Design strategies for hw3b

HW3

Be sure to read the assignment handout first to get familiar with the functional requirements of the two programs. This handout offers some additional advice and suggestions for navigating the new areas that you will need to tackle for this assignment.

LinkTester networking

A good starting task for the LinkTester is familiarizing yourself with the URL, URLConnection, and HttpURLConnection classes. Like we suggested for Webster, it might be easiest to start with a command-line version of the program that allows the user to enter a URL string and then downloads and prints the requested document. Extend the program to handle tracking simple statistics about the download including the content-type, http response code, size and time required, overall throughput, etc.. Have the program print out a summary line after each 500-byte chunk has been read so you can watch the progress being made. Try it out on various trouble cases -- trying to load a malformed URL, access a non-existent file, or contact a non-existent host and make sure you have the appropriate handling of these error conditions.

Go on and incorporate the use of our LinkScanner class to extract and print the links found in the downloaded document. It is probably easiest to first just download the bytes, appending each read chunk into a StringBuffer, then wrap a StringReader on the String contents and push that through the LinkScanner (rather than trying to download and scan for links at the same time). Save the founds links in some sort of collection, avoiding duplicates, and then download each sequentially, printing out a final summary of the time and throughput results for all links at the end. Congratulations! You are on your way. Now the trick is making it all happen in parallel…

ThreadPool

Implementing ThreadPool is probably the next task to tackle. The ThreadPool is a generic entity, not something specific to the LinkTester. Think of it as queuing management for any Runnable objects. For each task you would like to dispatch in a separate thread, you create the Runnable object and hand it over to the ThreadPool. The ThreadPool has a maximum cap on the number of active threads within the pool if not yet over the cap, a new Thread for the Runnable objects is created and started. Otherwise, it will store the Runnable on an internally managed queue. When any previously started thread in the pool finishes, the first Runnable on the queue then gets a chance to go.

One slightly tricky part is figuring out how to make it so that last action of any pool thread is to communicate with the ThreadPool about the impending exit of the thread so it can start another. It should not be the case that the client is required to supply a specific type of Runnable object that does this (Here's a hint: think about "wrapping" the client's Runnable in what you need…)

There are some issues of synchronization here— for example, what happens if two clients try to create a new thread in the pool at the same time? What if two or more previously spawned threads finish simultaneously? The ThreadPool will need to take precautions that these simultaneously action do not interfere with each other with careful use of the synchronized keyword.

To support reporting an emptied ThreadPool, there should be a method a client can use that blocks until the ThreadPool notifies it that all threads in the pool have finished and no more remain in the queue. This is a chance to try out the wait and notify methods that allow you to coordinate the actions between two threads. One way to do it would be to have the client wait until the last thread

exits (there are other ways, but this is perhaps the most straightforward). You may need to be careful about the case where the only thread left in the pool is the one actually waiting for the pool to empty it doesn't get stuck there for eternity.

LinkTester threads

From here, you will put the ThreadPool to use in building the multi-threaded version of the command-line link tester (saving the user interface work for last). Dispatching the downloading into separate threads will involve create a Runnable that wraps around the task and sending it off to the ThreadPool. By keeping your status print requests in, you can monitor the progress of your command-line program to see how the threads are trading off and make sure the ThreadPool constraint is being properly respected.

If you're working over a fast connection or accessing small files via the local filesystem URLs, you may find that the downloads proceed too quickly for you to observe what is going and verify the right things are happening. Inserting a Thread.sleep() for 200 or so milliseconds after reading each chunk of data may come in handy while you are in development. This, of course, will completely skew the numbers for download time and throughput.

LinkTester UI

Your experience from Webster will come in handy when creating and laying out the interface for the LinkTester. The picture in the assignment handout shows one possible configuration, but it's up to you to decide how fancy you want to go. The controls for setting up the search shouldn't be too much trouble. The threads control is probably just a simple JTextField, the Go an ordinary JButton. I choose to use an editable JComboBox for selecting the starting URL because that it made it convenient during development to quickly pick from a list of common testing URLs, as well as directly enter any new URL that I wanted to test. You could instead use a JTextField, but then you would also have to re-type different URLs as needed.

By far, the most complicated part of the LinkTester user interface is the JTable. We only briefly discussed JTable in lecture, so you will want to check the on-line docs and excellent examples in the Java Tutorial to pick up some more background information.

Writing your own TableModel

JTable is designed in the MVC paradigm. The data drawn in the table cells is obtained by messaging the JTable's underlying TableModel. So a good first task is reading up up on TableModel and AbstractTableModel. You will need to write your own TableModel implementation to provide the information about the links and their ongoing progress to display in the table. A good starting point is subclassing from AbstractTableModel, so that your model inherits the model listener list and methods to post events to the listeners when data in the model changes. You will only need to fill in the abstract methods for getting the count of rows and columns, getttng each colunn name, and providing the String value for each (row,col) cell.

The data model needs to store the column labels and all the rows. Most of the data is in the collection of rows, where each row object tracks all the status for one link. The downloading thread working on that link will update the data as its progresses and each update should fire the appropriate notification from the table model about cells and/or rows changing so that the UI will properly re-draw the newly changed information.

Fundamentally, all the operations on the row link object, such as updateBytesRead or setStatusString perform the change on the data model data structure and then do the appropriate fireXXX event to notify the view. In turn, this causes the table to interrogate the data model to figure out the new state and redraw it. To make for efficient re-draw, you should take care to fire specific change notifications, e.g. fireTableRowsUpdated() or fireTableCellUpdated() instead of the more drastic fireTableDataChanged() that re-draws everything. Another good reason to avoid the big

change notifications is that ones such as fireTableStructureChanged() essentially cause the table to be built from scratch from the model which causes the table to forget state such as the column widths. Note that there should be no explicit calls to paint/repaint/update on your table, all updates should be done by broadcasting model change notifications for which table listens.

Swing and threads

Swing takes the position that actions that affect Swing components should be handled in the Swing event thread only. Most of the activities going on in the LinkTester programs don't affect the Swing components, but the few that do (most notably the posting of the fireXXXUpdated events that take action on the JTable itself) thus need to be sent through the Swing event thread. Use SwingUtilities.invokeLater(). For example, to safely post a change to the cell at (4,5) in the table from some thread other than the event thread, you would do something like this:

```
SwingUtilities.invokeLater(new Runnable() {
      public void run() {
          model.fireTableCellUpdated (4, 5); }
      });
```

A few random details

- Consideration of others and conservation of shared resources means you should not repeatedly perform large downloads of any network sites. Do most of your testing by downloading local file URLs so that you are only accessing files on your machine. Limit yourself to only a few small downloads over the network when necessary. Everyone who shares the bandwidth thanks you in advance for your proper network etiquette.
- When testing your program, we advise you to stay within conservative limits (say 20) on the maximum number of threads, given that many Java systems have constraints on the number of concurrent threads and open files and connections. We won’t push your program higher than this, and you shouldn’t either.
- One interesting thing to note is that if one of the threads run into a fatal problem (such as using a null object reference or accessing an array out of bounds), only that thread is stopped. The rest of program and the other threads keep going merrily along. So if one of your links appears to get stuck as though its thread died, you might want to check the output window to see if a fatal exception was reported for that thread so you know what you need to fix.
- The underlying implementations of threads tend to vary quite significantly among Java platforms. For this reason, we consider it a good idea for you to do some testing on Solaris, since it has one of the more reliable thread implementations and that is where we will be testing your submissions. If your program works fine on one platform and incorrectly on another, your program probably has a thread bug, but that happens to only be triggered by the circumstances on one platform or another.