HW4 Threads

Thanks to Nick Parlante for much of this handout

HW4 has four medium sized parts -- A, B, C, D -- to explore threads in different ways. Parts A and B deal with classical threading and synchronization. Parts C and D have to do with GUI threading and networking. All the parts of HW4 are due at **6pm** on Fri Oct 31st. This is the last project where late days can be used.

A. Thread Bank

Part A builds some classical, non-GUI threaded code.

For this problem, you have an array of `Account` objects and a text file of transactions where each transaction moves an amount of money from one account to another. We start each account with a balance of 1000, then apply all the transactions to compute the final balances for all the accounts.

The trick here is to use threading to complete the operations in parallel...

- One thread reads the transactions from the file, one at a time, and adds them to a buffer object.

- There are multiple worker threads, where each worker repeatedly gets a transaction from the buffer and performs that transaction on the accounts.

Here are the classes (some skeleton code is provided in the starter files)...

Account

`Account` is a simple, classical class that encapsulates an `int balance` and `int` number of `transactions`. The ctor should take the initial `balance` and the `transactions` should start at 0. The `Account` should have some sort of `synchronized change()` method that changes the `balance` and increments the number of `transactions`. Feel free to set the number of arguments to the change method and other parts of the `Account` interface however you like. Note that using separate `get()` and `set()` methods to change the balance does not work, since some other thread could be calling get() and set() at the same time, causing problems.

Buffer

The `Buffer` object is a temporary storage area that holds the transactions before they are processed. The `Buffer` uses the `Transaction` class to store the ints `from`, `to`, and `amount` -- `from` and `to` are the account numbers and `amount` is the amount to transfer. The `Transaction` class is just a struct used for storage.

`Buffer` should respond to `void add(Transaction)` which adds a transaction object to the buffer. The thread reading transactions from the file and adding them to the buffer will call `add()`. `Buffer` should respond to a `Transaction remove()` operation that gets a transaction object from the buffer. The worker threads that process the transactions will call `remove()` to get transactions. `remove()` should provide elements in the order in which they were added, like a FIFO queue.

Buffer Implementation

Internally, the buffer should store the transactions in an `ArrayList`. To provide the elements in FIFO order, use `list.add()` to add new elements, and `list.remove(0)` removes and returns the first element. Note that the collection classes themselves are not thread safe, so you need some locking strategy to avoid calling, for example, `list.add()` and `list.remove()` at the same time. There is a `BlockingQueue`

class in the JDK that does something similar, but in this case I want you to build the `Buffer` functionality yourself as a meaningful example that combines locks and semaphores.

If a client calls `buffer.remove()` when the buffer is empty, that `remove()` should block until an element is available. When an element is available, `buffer.remove()` can wake up and return that element. Similarly, we'll say that the buffer should never hold more than 64 elements. If a `buffer.add()` tries to add a 65th element, that `add()` should block until space is available. Use `canAdd` and `canRemove` Semaphores inside the `Buffer` to implement the blocking.

Tricky issue: the `buffer.add()` and `buffer.remove()` methods should not be synchronized. Normally, they would be synchronized to protect the buffer instance variables, but in this case, the semaphore `acquire()` calls may block for a while, and it's important to not hold the buffer lock while blocked. If we held the buffer lock, then no other thread could get in to fix things and `release()`. Therefore: do the semaphore `acquire()` and `release()` without holding the buffer lock. Use a `synchronized` (something) { ... } section to lock around the list manipulation code safe (as shown in lecture).

Bank

The `Bank` class should contain an array of 20 accounts and a buffer object. From the command line, the `Bank` should take the filename of transactions and the number of worker threads to use. In `main()` (i.e. on the main thread), the bank should create and start the workers, read through the file and add transactions to the buffer. After all the transactions have been added, the Bank should add one `null` to the buffer for each worker as a signal that the input is done. When each worker gets a `null` from `remove()`, it knows the data is finished and can exit cleanly. When all the workers are finished, print a summary for each account on one line, giving its account number, balance, and number of transactions (format shown below). You can use a `toString()` in `Account` to produce the one-line account summary.

Worker

`Worker` should be an inner class of `Bank` that runs the following loop: try to get a transaction. If the transaction is `null`, exit the loop. For each transaction, withdraw the given amount from the `from` account and deposit it into the `to` account. For our solution, we will say that it is not necessary to hold the locks for the `from` and `to` accounts simultaneously -- holding the locks one at a time to manipulate each account in a thread-safe way is sufficient.

Command Line Args

The command-line arguments are passed to a Java program in the `main(String[] args)` array. The first argument is at index 0, the next at index 1, and so on. For `Bank`, the first argument is the filename of transactions, and the second argument, if present, is the number of worker threads, so the arguments "`5k.txt 4`" use that file with 4 workers. If the number of workers is not specified, it should default to `1`. In Eclipse, use the *Run*.. command to bring up the Run dialog box. Then use the *New* button, to create little run cases, each with its own name and arguments. In this way, you can set up little run cases and invoke them easily.

It's also possible to run your Java program from the command line (you can still use Eclipse to compile) -- from the directory where Eclipse puts your .class files, the command "`java Bank 5k.txt 2`" will run the Bank `main()` with the arguments "`5k.txt 2`".

```
> java Bank small.txt 4
acct:0 bal:999 trans:1
acct:1 bal:1001 trans:1
acct:2 bal:999 trans:1
acct:3 bal:1001 trans:1
acct:4 bal:999 trans:1
acct:5 bal:1001 trans:1
...
acct:17 bal:1001 trans:1
acct:18 bal:999 trans:1
acct:19 bal:1001 trans:1
```

Bad Accounting

When you have the basic account features working, add the following Bad Accounting feature. When the command line contains a third "limit" argument, the bank should keep track of "bad" transactions, which we will define precisely as a transaction that causes the balance of the "from" account to transition from being at or above the limit to being strictly below the limit (in any case, the resulting balance can be positive or negative or whatever). The `Bank` object should respond to some sort of `addBad(bad-transaction, bad-balance)` message to record the details about the bad transaction and resulting bad balance. The `addBad()` message should be sent to the `Bank` for each bad transaction. The bad transaction still goes through, but its existence is noted. Modify Bank `main()` to print out a line for each bad transaction after the regular output with the following format...

```
> java Bank test.txt 4 0      ## note limit arg of "0"
...
acct:18 bal:2140 trans:22
acct:19 bal:930 trans:8
Bad transactions...
from:7 to:2 amt:100 bal:-40
from:11 to:19 amt:160 bal:-80
...
```

Add classes, ivars, methods, and parameters as needed to support the limit/bad-transaction feature. The bank will need a way to store bad transaction information -- consider using a simple nested class with a `toString()`. If the limit command-line argument is not specified, then `addBad()` should not be called and the Bad Accounts printing should not happen. Note that which accounts violate the limit very much changes from one run to the next, since it depends on the exact order that the transactions are applied. The ending balance of each account should be the same every time however. As usual, please clean up your output formatting to look like the above before turning your code in.

The file `5k.txt` contains a 5000 transactions that "balance" -- when all done, they leave the accounts with the same balances they started with. Note that some transactions transfer from an account and to that same account. The file `100k.txt` contains 100k transactions that balance. If your code contains concurrency bugs, they may not show themselves every time. Try temporarily removing `synchronized` from places to observe that it causes the code to crash and/or get the wrong answer.

On the myth.stanford.edu machines (multi-processor), use the shell "time" command like this: "`time java Bank 100k.txt 4`". Theoretically, you could get over 100% CPU utilization -- you are using multiple CPUs at the same time -- neat!

B. Hash Cracker

The `Bank` problem deals with threads and synchronization. For this problem you will write a `Cracker` class that uses threads that are largely independent, to make full use of all the CPUs we have available.

1-Way Hash Function

A 1-way hash function takes in some bytes and computes a "hash" or "digest" value that in some sense summarizes all the data in the input bytes. Hash functions are widely used to verify data integrity and with cryptography. In this case, we will use the "SHA" hash function which produces a 20 byte hash value (in Java, the hash value will come to us as a `byte[]` array -- In Java a "byte" is like a small `int` that can only hold values in the range -128..127.) A 20 byte hash value can be printed as a string of 20, 2-hexdigit numbers, such as...

```
689fa1c433278765a476686df05cb2de9158887f
```

The key feature of a 1-way hash function is that, given the hash value, there is no easy way to compute what bytes led to that hash value -- the hash function is not "invertible", or it is said to be "one way".

Suppose we have a hash value, but we do not know the original input that gave that hash value. For example, we might have the hash value that summarizes someone's password, but we do not know that person's password. To figure out the original input value, we brute-force enumerate all the possible inputs,

compute the hash for each input, and see which input yields the hash value we are looking for. This is known as brute-force "cracking" the hash.

Of course this brute-force technique is going to be very slow if the input is at all long -- that's why you should choose passwords that are not too short! For this project, we'll write brute-force code that works on short inputs, and it will be an excuse to use threading in a neat way.

Hash Search Problem

First, the handout will describe the computation and output we want. Later on, the handout will describe the threading and implementation strategies.

Here is a CHARS array constant we will use that holds the lowercase letters, digits, and some punctuation to make up our input strings…

```
public static final char[] CHARS =
  "abcdefghijklmnopqrstuvwxyz0123456789.,-!".toCharArray();
```

Here is the sequence of strings length two or less that we can make from those chars …

```
a
aa
ab
ac
...
a-
a!
b
ba
bb
bc
...
b-
b!
c
ca
cb
...
!,
!-
!!
```

The Cracker main() should take three command-line arguments: the "target" string, the max "length" of the input strings, and the number of worker threads (defaults to 1).

When run with the target string "print", the program should print each input string followed by a space, followed by its hash value in hex form. The program should print "all done" when all the processing has been completed. So with the arguments "print 2", the program shows all the hash values for strings length 1 or 2 in this order (same order as shown above):

```
> java Cracker print 2
a 86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
aa e0c9035898dd52fc65c41454cec9c4d2611bfb37
ab da23614e02469a0d7c7bd1bdab5c9c474b1904dc
ac 0c11d463c749db5838e2c0e489bf869d531e5403
ad 4aeb195cd69ed93520b9b4129636264e0cdc0153
ae 1eabdaf488b3a3682bbca94c5f468f065cdfaf13
af d1e622507595486ee06db24b1debf11064edd2ba
ag 7edd1dd232a61b147151d657b4ad5080896f8f0d
ah fd0aa93434507bb33ff096a66a4891c2bc4fa12d
...
!9 689fa1c433278765a476686df05cb2de9158887f
!. 7484f7eb74e912f0e27018d06dc712fff8f5a65c
!, 9060de60b298c327830d874bf15577b0bd8709e4
!- 17a05c34273afb761817f81baf5a05d3fcf222c4
!! b4613f8681b1e26686a2e88299525a4dc89c46d5
all done
```

Each increase in the input length yields exponentially more possible strings (40x more for each additional character in length). The amount of output is feasible with input strings up to about length 4 or 5 (2.5 million input strings for length 4) -- after that, even just printing the output seems to take forever.

If the target string is not "print", it will be a hex hash value. In that case, the code should iterate through all the possible inputs up to the given length, compute the hash for each, and compare each hash value to the given hash value. If the hashes match, print a line "`match:<input> <hash value>`".

So for example, the arguments "`689fa1c433278765a476686df05cb2de9158887f 2`" give the output

```
match:!9 689fa1c433278765a476686df05cb2de9158887f
all done
```

since the input "`!9`" gives exactly that hash value (as seen in the "print" output above). After finding a match, the code should continue looking for other inputs that match, although finding more than one match is profoundly unlikely.

Threading

The brute-force search nature of this problem is a natural match for concurrency. Rather than having one thread go through all the possibilities, we can fork off multiple worker threads, each searching through some part of the input string space. Since each worker has its own, isolated part of the input space, the workers do not need to coordinate with each other once they are running -- most efficient.

Our strategy will be to give each worker a different part of the input space to run through, based on the **first character** of the strings.

- If there is 1 worker thread, then it uses all the input chars, a..! (indexes 0..39 in the CHARS string) at the start of its strings.

- If there are 2 worker threads, then the first worker creates all the strings starting with chars a..t (indexes 0..19), and the second worker creates all the stings staring with chars u..! (indexes 20..39).

- Do not first generate a collection of all possible inputs, and then feed them to the workers. The workers should do their own generation of inputs, so the generation happens in parallel too.

- There are 40 chars, and so which worker gets which starting chars may not divide up exactly evenly. That's ok, just so long as every starting char is accounted for exactly once, and all the workers have roughly the same number of starting chars. If the user really cares that things work out evenly, they can give a number of workers that divides into 40 evenly, such as 4 or 8 or 10 or 20. Our scheme only works for up to 40 worker threads.

- Probably the easiest way to think about the starting chars, is that each worker has a range of index numbers (e.g. 0..19, or 20..39), and it uses the chars from the CHARS array with those indexes.

- The *print* command does not make much sense with more than 1 worker, since their `System.out` output will get mixed up, but we'll allow the user to do that if they want, and in the case the order of the printout is not defined.

- For 1 worker, the printout order should be as above. For multiple workers the order does not matter. Note that `System.out.println()` can be called by multiple concurrent threads and it does the right thing.

Implementation

Look at the docs for the Java's built-in `MessageDigest` class -- its API docs and interface are not fantastic from a client-oriented point of view, but they get the job done. Each worker should have its own `MessageDigest` object, using the "SHA" algorithm (the creation requires a `try/catch` -- you can just `printStackTrace()` in the catch). Given a `String`, use its `getBytes()` method to get a `byte[]` array of its data. Use the `MessageDigest` methods `reset()`, `update()`, and `digest()`. `update()` adds data to go into the digest. `reset()` clears the digest back to being empty. Use the `MessageDigest` static method `isEqual(byte[], byte[])` to compare the hash bytes computed with the target hash value. The starter code has utility methods `hexToString(byte[])` and `hexToArray(String)` to convert between `byte[]` arrays and their hex-string form. Just use the hex strings for input/output (making the hex strings is a little costly); the internal computation and comparison should be in terms of simple, fast `byte[]` arrays.

Command-Line Arguments

Running from the command-line can be handy for "*print*", since the Eclipse console has a hard time keeping up with such a large volume of console output. In the directory containing the generated .class files, the command "`java Cracker print 2`", runs the `Cracker` class `main()`, passing the command-line arguments "`print`" and "`2`". On Unix, the command "`java Cracker print 2 > print2.txt`" captures the output in a "`print2.txt`" file for later analysis. For the `Cracker` project, the command line can be a handy way to run the code and see its output.

Strategy and Speedup

The recursive enumeration of all the input strings is the slightly tricky, algorithmic core of this problem. Use the *print* command with lengths 2 or 3 to see that your code really is dividing up the first-char space correctly and completely. When *print* is working, the generate-hash-and-check logic is pretty simple -- you just do it once for each string. Running on a machine with multiple CPUs (myths have 2). On my 2-CPU machine, the code is not twice as fast, but it's a good 40% faster when running with 2 workers. The optimal number of workers is 1 worker for each available CPU. Try the Unix "*time*" command ("`time java Cracker …`") to see how long the code takes to run with different numbers of threads. ("`man time`" to see how it works.) If your code does not run faster with 2 threads on a lightly loaded myth, then there may be a problem with your code.

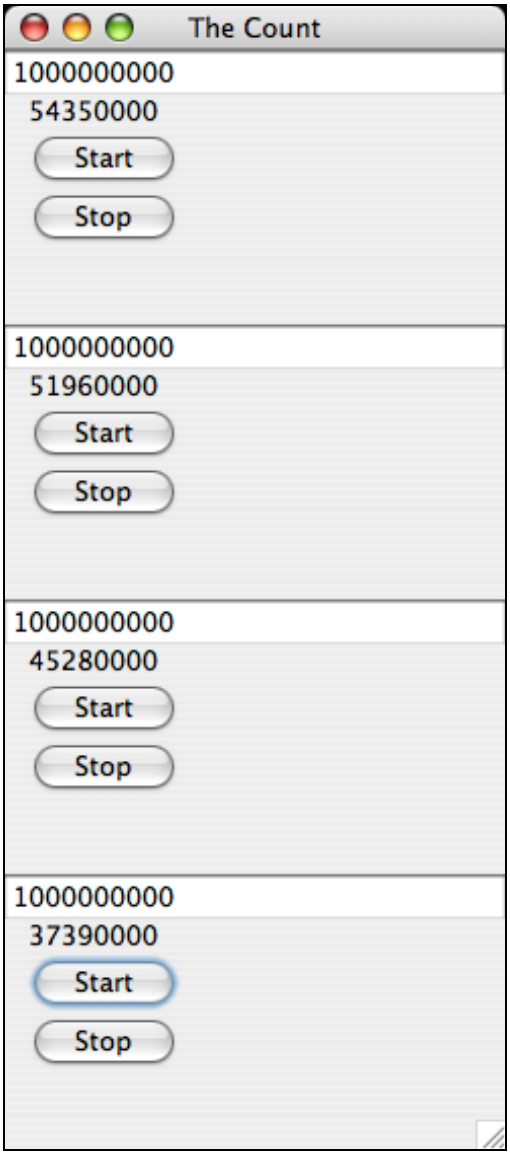
Once you have it all working, log into myth twice. Run the cracker from one shell and in the other shell run the "top" command to observe the CPU etc. use of your program as it runs. At the same time, you can observe other people running programs that top out at a pathetic 100% CPU utilization. We pity them and their antique, single-threaded software!

When your code is working nicely, use it to figure out what 4 character input yields the mystery hash "`c5e478e7da53b70f0fabcdefa082e1d1c5a2bc6d`" (it's a cute hash because it has abcdef in it).

As usual, you are free to print debugging information while working on the program, but please clean up the output to the format described above before turning it in.

C. The Count

This is a little GUI threading exercise called "The Count". It's extremely short, but it hits the major issues of forking off multiple workers and getting them to interact cleanly with the GUI. This is a warmup for the more complex GUI threading code in part (D).



Create a `JCount` subclass of `JPanel`. The `JCount` should use a vertical box layout to contain 5 off-the-shelf components: a `JTextField`, a `JLabel`, a Start button, a Stop button, and a 40 pixel vertical strut to create space at the bottom of the `JCount`. `JCount.main()` should create a frame with a vertical box layout, and install 4 `JCounts` in it.

The Box will force the `JTextField` to be wide, since the Box tries to make the things it contains uniform in width. We can live with that.

When the start button is clicked, the `JCount` should fork off a worker thread that uses a for loop to count from 1 to the bound in the text field at the time of the button click. The start button should interrupt the current worker, if there is one, before starting the new one. The stop button should interrupt the current

worker if there is one. Initially the text field should start with the value 100 million and the status field should start with the value 0. Once every ten thousand iterations, the worker thread should do the following...

1 If the worker is not interrupted, it should update the status field to show the current count value and keep looping.

2. If the worker is interrupted, then it should exit its loop silently.

3. Optional: The worker could call `Thread.yield()`. This is not required in general for thread programs, but it can help the thread scheduling system to switch among the various threads regularly. If the GUI animation is abrupt on your machine, this may help smooth it out.

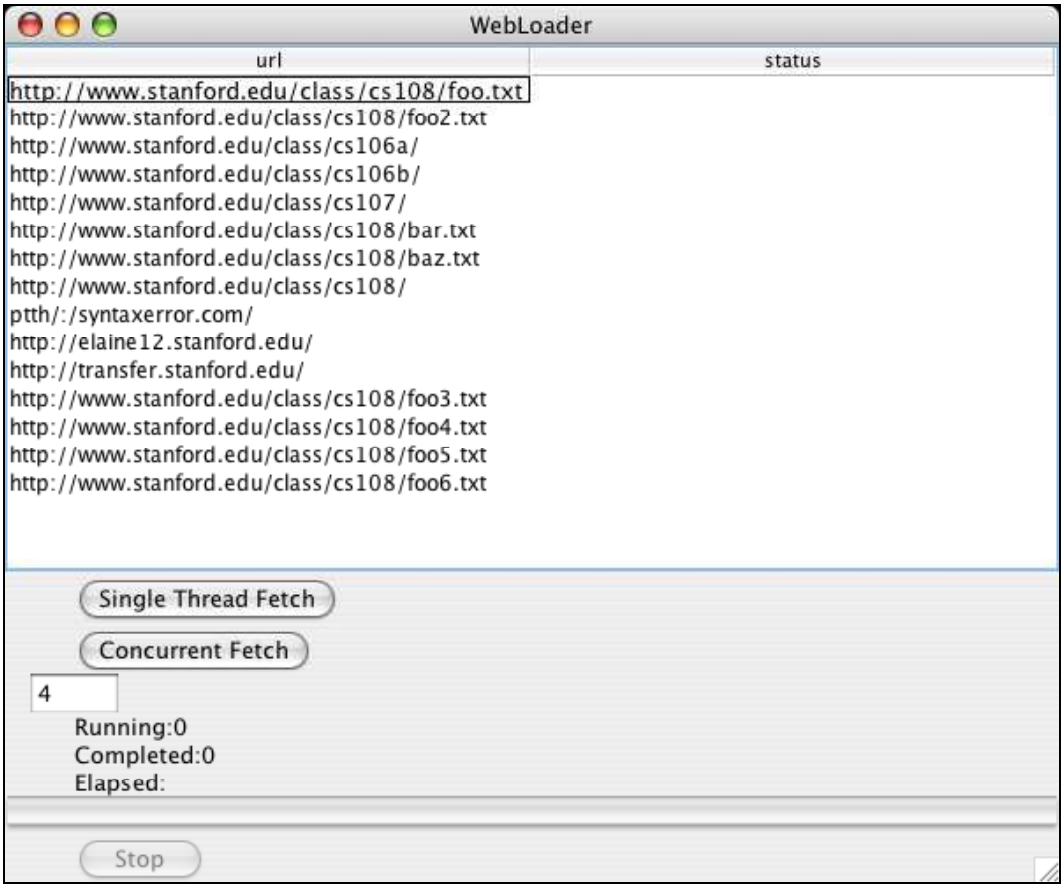
It should be possible to start all 4 `JCounts` running, start and stop each one at will, resize the window etc. while they are all running. For some reason, on MS Windows the performance of this program can be jerky, but it still basically works -- that's fine.

D. WebLoader

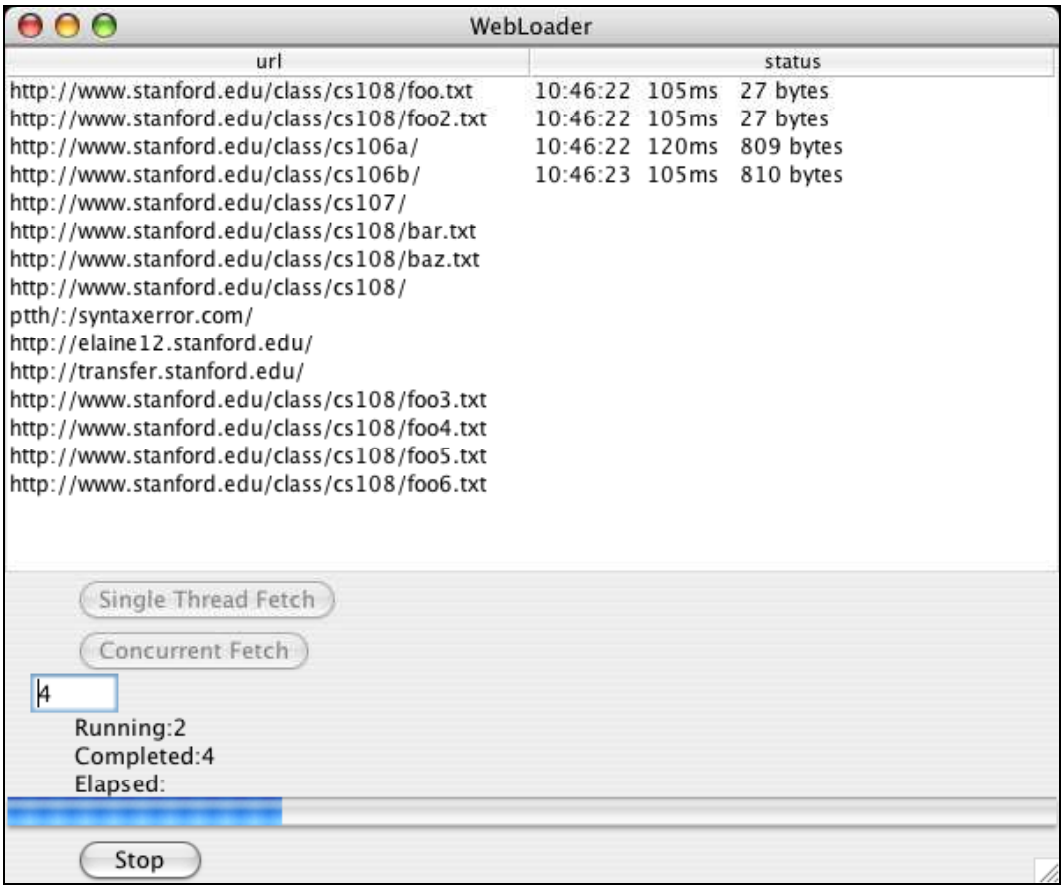
For this part, we will build a little URL downloading program that shows off the combined power of GUIs and threads. We will start with an overview of the operation of the program, and then talk about implementation strategies.

The WebLoader program loads a list url strings from a file. It presents the urls in the left column of a table. When one of the *Fetch* buttons is clicked, it forks off one or more threads to download the HTML for each url. A *Stop* button can kill off the downloading threads if desired. A progress bar and some other status fields show the progress of the downloads -- the number of worker threads that have completed their run, the number of threads currently running, and (when done running) the elapsed time.

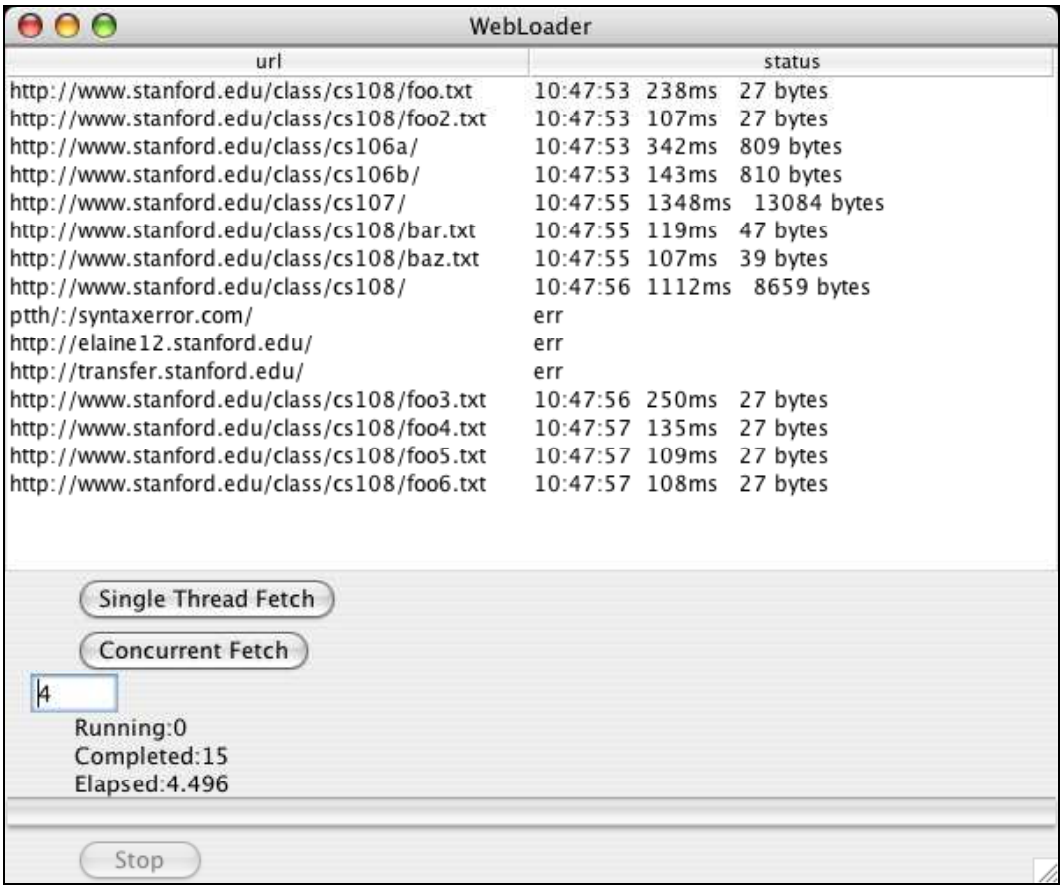
Here is the `WebLoader` interface, with no workers running, ready to be started. The *Fetch* buttons are enabled, and the stop button is disabled (grayed out).



Here, the Single Thread Fetch has been clicked, and one worker at a time, the program is proceeding through the urls. The Fetch buttons are disabled, and the Stop button is enabled. Four workers have finished and written their url's status. One worker is running. The "Running" thread count is 2, since it accounts for the one worker plus the one "launcher" thread described below.



Finally, here the fetch has been completed, with a run of 15 workers.



There are two main classes that make up the `WebLoader` program…

WebFrame

The `WebFrame` should contain the GUI, keep pointers to the main elements, and manage the overall program flow.

WebWorker

`WebWorker` is a subclass of `Thread` that downloads the content for one url. The "Fetch" buttons ultimately fork off a few `WebWorkers`.

The files "`links.txt`" and "`links2.txt`" in the starter directory have some urls for you to play with. As shown in the screen shots, some of the urls in links.txt should error out -- they are not actually web servers.

The starter files for this part are minimal -- WebWorker.java has the basic networking code shown below, and you should create your own `WebFrame`. One way to create a new class in Eclipse is with the *New..Class* command. Another handy technique is this: find an existing, similar class -- such as some other `JFrame` subclass. Copy it into your project directory, and give it the filename you want. In Eclipse, use the *Refresh* command, and it will notice the new file. Edit the file, taking advantage of the boilerplate code already in it, and make it into the class you need.

WebFrame Setup

The `WebFrame` constructor should read the file "links.txt" into a `DefaultTableModel`/`JTable` installed in a panel, like this…

```
model = new DefaultTableModel(new String[] { "url", "status"}, 0);
table = new JTable(model);
```

```
table.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);

JScrollPane scrollpane = new JScrollPane(table);
scrollpane.setPreferredSize(new Dimension(600,300));
panel.add(scrollpane);
```

`JTable` is a component that displays a table made of rows and columns. The `DefaultTableModel` is an off the shelf class that holds the data for a `JTable` its data. The `DefaultTableModel` implements an `addRow()` method that takes an array and adds its data as a row, and a `setValueAt(Object, row, col)` method to install a `String` in a particular cell. See the `DefaultTableModel` docs. (In HW5, we will write out own table model. Here, we just use `DefaultTableModel` off the shelf.)

Below the table, install the three buttons, three `JLabels`, one `JTextField` and one `JProgressBar`, with something like the appearance shown above. By default, the `JTextField` will grow to fill the width of the box, which works but does not look good. Use `setMaximumSize()` on the field to kept its size looking reasonable.

Fetching URLs

When one of the Fetch buttons is clicked, the GUI should change to a "running" state -- fetch buttons disabled, stop button enabled, status strings reset, maximum on the progress bar set to the number of urls.

The swing thread cannot wait around to launch all the workers, so create a special "launcher" thread to create and start all the workers. We will include the launcher thread in the count of "running" threads. Having a separate launcher thread helps keep the GUI snappy — we leave the GUI thread to service the GUI. Notice that GUI reacts quickly to mouse button clicks, etc. even as the launcher and all the worker threads are working and blocking.

The launcher should do the following

* Run a loop to create and start `WebWorker` objects, one for each url.

* Rather than starting all the workers at once, we will use a limit on the number of workers running at one time. This is a common technique -- starting a thousand threads at once can bog the system down. It's better to limit the number of concurrent workers to some reasonable number. When the *Single Thread Fetch* button is clicked, limit at one worker. If the *Concurrent Fetch* button is clicked, limit at the int value in the text field at the time of the click.

* Use a semaphore to enforce the limit -- the launcher should not create/start more workers than the limit, and each worker at the very end of its run can signal the launcher that it can go ahead and create/start another worker.

* At the very start of its `run()`, each worker should increment the running-threads count. At the very end of its run, each worker should decrement the running-threads count. The launcher thread should also do this to account for itself, so the running-threads count includes the launcher plus all the currently running workers.

* You can detect that the whole run is done when the running thread count gets back down to zero. When that finally happens, compute the elapsed time and switch the GUI back to the "ready" state -- Fetch buttons enabled, Stop button disabled, and progress bar value set to 0.

When the launcher/limit system is working, the running-threads status string should hover right at or below the limit value plus one (for the launcher) at first, and gradually go down to zero.

Make a new semaphore for each Fetch run -- that way you do not depend on the semaphore state left behind by the previous run. Interruption may mess up the internal state of the semaphore.

WebWorker Strategy

The `WebWorker` constructor should take a `String url`, the `int` row number in the table that it should update, and a pointer back to the `WebFrame` to send it messages.

The `WebWorker` should have a `download()` method, called from its `run()`, that tries to download the content of the url. The `download()` may or may not succeed for any number of reasons, it will probably take between a fraction of a second and a couple seconds to run

Given a url-string, Java has classes that make parsing the URL and retrieving its content pretty easy. The standard code to get an input stream from a URL and download its content is shown below (see the `URLConnection` API docs, and this code is in the starter file). The process may fail in many different ways at runtime, in which case the flow of control jumps down to the `catch` clauses and continues from there. If the download succeeds, control gets to the "Success" line.

```
InputStream input = null;
StringBuilder contents = null;
try {
    URL url = new URL(urlString);
    URLConnection connection = url.openConnection();

    // Set connect() to throw an IOException
    // if connection does not succeed in this many msecs.
    connection.setConnectTimeout(5000);

    connection.connect();
    input = connection.getInputStream();

    BufferedReader reader  = new BufferedReader(new InputStreamReader(input));

    char[] array = new char[1000];
    int len;
    contents = new StringBuilder(1000);
    while ((len = reader.read(array, 0, array.length)) > 0) {
        contents.append(array, 0, len);
        Thread.sleep(100);
    }

    // Successful download if we get here

}
// Otherwise control jumps to a catch...
catch(MalformedURLException ignored) {}
catch(InterruptedException exception) {
    // YOUR CODE HERE
    // deal with interruption
}
catch(IOException ignored) {}
// "finally" clause, to close the input stream
// in any case
finally {
    try{
        if (input != null) input.close();
    }
    catch(IOException ignored) {}
}
```

Things to notice...

- The code should test `isInterrupted()` periodically while reading and stop trying to read in that case. This will be important later as you try to get the *Stop* button to actually stop things. The worker may not notice `isInterrupted()` instantly, since the flow of control may be down in the `read()` library code for a time, before it makes it back out to your code to notice the interruption. The other possibility is that one of the blocking operations (e.g. `read()`) will throw `InterruptedException` and the flow of control will jump to that `catch()` clause.

- The `Thread.sleep(100)` line is a required slowdown for this assignment — otherwise it all happens too fast to interpret. I want you to see the progress and interaction of the threads, stop button, progress bar, etc., and slowing the threads down a little is the best way.

- The "finally" at the end runs whether or not there is an exception -- this is a standard use of the finally clause to be sure to `close()` the input.

When the download is done reading (successfully or not), the `WebWorker` should update the GUI.

First, the worker should deal with the corresponding status string in the table. If the worker was interrupted, it should set its status to "interrupted". Otherwise, if the worker was not interrupted it should update the status as follows: If the download was successful, update the status of the corresponding row in the table with a `String` that summarizes the download, giving the wallclock time of completion, the elapsed time of the download in milliseconds, and the size in bytes of the downloaded content (use the number of chars, although in reality the number of bytes could be larger for a page with non-ASCII content). See the `Date` class to figure the current time, and the `SimpleDateFormat` class to format the current time reasonably. If the download was not successful, just update the status to "`err`". We could do something with the content, but for this program we just download it and count how many chars there were (the length of `StringBuilder`).

Second, whether or not it was interrupted, the worker is about to exit and should update the GUI: decrease the count of running threads by 1, increase the count of completed threads and the progress bar by 1 (finishing for any reason -- interruption, success, err -- counts as "completion" for the thread), and work with the launcher semaphore to open up a slot for another worker. We are not counting the launcher thread in the completed count, just the workers.

The completed and current-running counts should only be changed when actually launching a thread, or when a thread is actually at the end of its `run()`. Don't just set them to zero when you suspect all the threads are done -- let the exiting of the threads manipulate them on their own. That way, the GUI will give you an accurate view of how your thread lifecycle code is working.

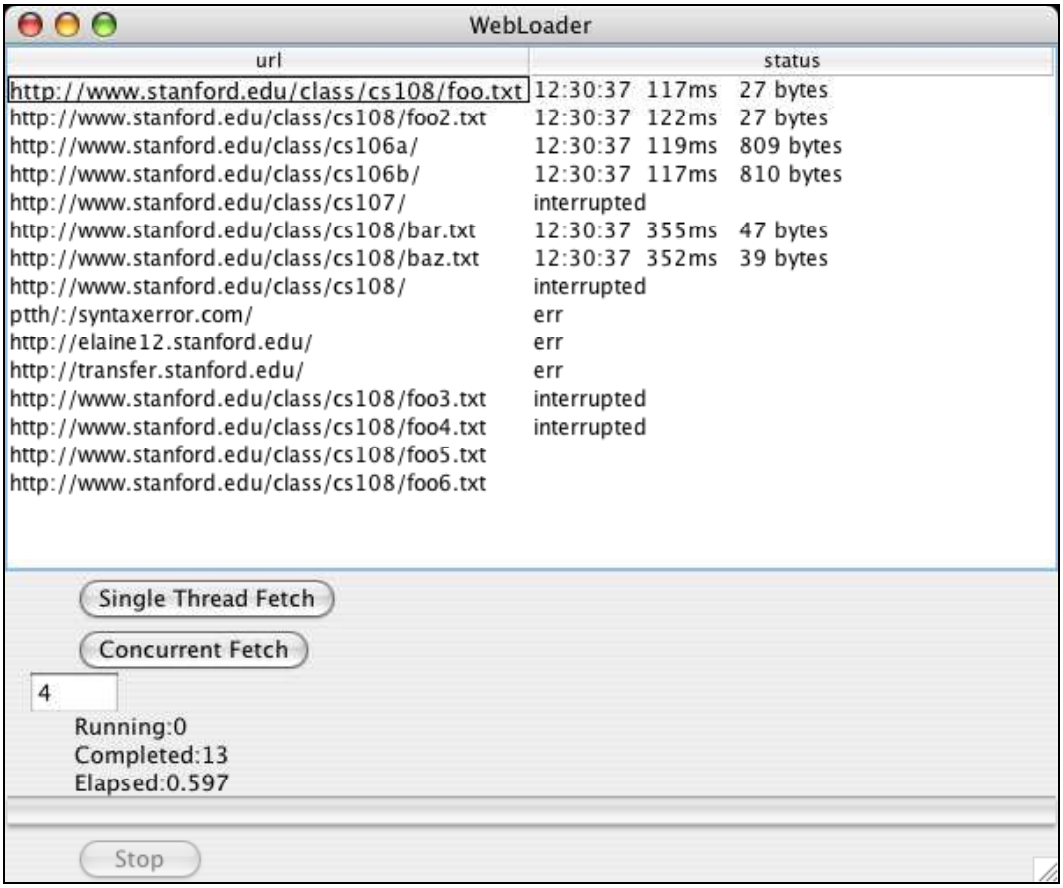
Interruption

The *Stop* button should interrupt the launcher (so it stops launching new workers) and all the other workers. The threads should notice when they have been interrupted without too much delay. Depending on the JVM implementation, there may be a little delay before the workers get to a point where they notice they have been interrupted (if they are blocked down in `read()` or `connect()` for example. Windows I/O seems to be especially slow noticing interruption).

When interrupted, the launcher should stop creating and starting new worker threads. The launcher can just exit its run. Eventually all the other workers will notice that they have been interrupted, and the running count will go down to 0. Since the labels track the running and completion of threads, you can watch the GUI as you play with the *Fetch* and *Stop* buttons to see that your threads are starting up and closing down properly.

Tricky timing case: what if the user clicks the *stop* button just as the launcher is creating and starting another worker. You can imagine a pathological case where the *Stop* button interrupts all the workers just after the launcher checks `isInterrupted()`, so the new worker doesn't get interrupted. Solution: introduce some locking so that the task of creating a new worker and the task of interrupting the launcher and all the workers cannot run at the same time.

Here is a screenshot after a Concurrent Fetch that was interrupted with the Stop button. Looking at the table, we see that 13 of the 15 rows completed and wrote their status. Four workers were downloading when the *Stop* button was clicked. The interruption happened before the launcher ever got the chance to start a worker for two bottom rows.



The Lesson Of Networking And Threads

If years from now you remember nothing else from our little thread adventure, try to remember the speedup between the single threaded and concurrent fetch. How does this speedup compare to the speedup in the others part of HW3? We'll discuss this a little in lecture. The basic idea is that concurrency and high latency activities go well together. The Internet is full of high latency activities.

Deliverables

Turn in all your HW4 project directory with a README as usual. **Please delete the *100k.txt* file before submitting** -- no need to upload such a big file!