



CS 341

Software Design

Minilab 4

Multithreaded Server

December 6, 2007

Today we will work on modifying the multi-threaded server we discussed in class. In our first modification, we will create a pool containing Worker objects. To handle a request, the web server will get a worker from the pool, create a thread to run the worker in, and start the thread.

In our second modification, we will create and start the threads when we first create a worker. When the worker thread completes one request, it will wait for the next request to arrive before continuing on. In this case, the Web server just gets a worker from the pool and gives it a new request. There will be no need to create a new thread.

Step 0: Run the web server

The web server will serve files from a directory that you specify. The directory should contain an HTML page that has some images on it. In this way, one web page request made by you using a browser will result in multiple download requests from the web server. So, the first step is to create a directory and place in it an html file and some images. For example, you could download and save the course home page and the 2 images that appear at the top of the page.

Download the Web Server project from the course Web page. Edit the WebServer class and change the value that the `root` variable is initialized to to be the directory you just created holding the Web page.

Run the web server. It will run on port 8080. You can access web pages using it with a URL like <http://localhost:8080/>. Make sure you can download some Web pages with it. Look at the output in Eclipse's console. You should see that each request was handled by a different worker.

When you are satisfied that it works, go to the Eclipse console and click on the red square in the console's title bar, at the right side, to stop the Web server. To convince yourself that it is really stopped, try to reload the page in your Web browser. This time, it should fail.

Step 1: Create a pool of worker objects

In this version of the program, a worker pool is used to hold `worker` objects that are waiting for requests. When a request comes in from a client, the Web server should get a worker from the pool, removing it from the pool since the pool should only hold idle workers. If the

pool is empty, a new worker should be created to handle the request. When the worker completes handling the request, it should put itself back in the pool.

Begin with the `workerPool` class, which has a skeletal implementation. It already contains a complete implementation of `addWorker`. You need to complete the `getWorker` method to create a new worker if the pool is empty.

Next, modify the `webServer` class. It needs to create the `workerPool` object. Then, when a client connects, instead of creating a worker, it should get a worker from the pool. In addition to giving it the socket that the request is arriving on you must also add a call to `setRoot` to tell the worker the root to serve from. Then start a new thread for that worker as before.

Finally, you need to determine when to put the `Worker` object back into the `workerPool`. This should happen when the worker has completed handling the request it was given.

Will the `workerPool` be used by more than one thread? If so, you need to determine where to add synchronization.

When you test your program, look at the output in the console. Hopefully, you will see that some workers are handling more than one request (but only one at a time!). If not, try using a Web page with more images in it so there are more opportunities to reuse workers.

Remember to stop the server before trying to run it again. If you try to start a server when the previous one is still running, you will get `java.net.BindException: Address already in use`.

Step 2: A Pool of Worker Threads

While some efficiency benefits can be gained by reusing `Worker` objects, even more could be gained if it was not necessary to create a new thread each time a client request comes in. In this next version of the server, you will therefore start workers running in threads when you first create them. The `Worker`'s `run` method will loop so that it can handle multiple requests rather than returning after handling one request. For the loop to work, the `Worker` will need to wait after it is started until a request arrives. After handling that request, it will need to wait again until the next one arrives. To accomplish this, you will need to use the `wait` and `notify` methods from last time to get the `worker` to behave properly.

First, change where the worker thread is created and started. This should no longer happen when a request arrives, but rather when the worker is created.

Next, turn the `worker`'s `run` method into a loop. Be sure to wait for a request to arrive (that is, for the socket to be set) before attempting to handle it.

Now that you have called `wait` inside `run`, you need to add a `notify` call when a request arrives so that the thread can handle the request. Remember that `wait` and `notify` can only be called when the object holds the lock.

Now when you download a Web page, you should see that each worker only starts once, but handles multiple requests. You should never see a "Done worker" message since all the worker threads are still running when the program exits.