

Design Notes

Assignment 2

Majid Ghaderi

Disclaimer

These notes are based on my own implementation. I do not claim that my implementation is the simplest or the best. Feel free to use or disregard any of these suggestions as you wish.

Tips

- Start with a single-threaded web server.
- Initially, use Telnet to test your code with well-formatted requests.
- Once basic web server works then add multi-threading.

Testing the Web Server

Start the server by simply running the `WebServer` class supplied with this assignment. To stop the server, type “quit” at the command prompt. Once the server is running, you can test it using Telnet, a web browser, or your Assignment 1 code. With Telnet, connect to the host running your web server at the specified port number. Alternatively, you can test your web server using a browser by sending the port number in the URL. For example, if the server is running on your local machine on port 2225, you can enter the URL `http://localhost:2225/obj.html` to retrieve the html file `obj.html`. Yet another approach is to modify the `Tester` program from Assignment 1 to programmatically send HTTP requests to the server using the `getObject()` method in your `UrlCache` class.

Terminating Web Server and `shutdown()` Method

While there are several techniques for terminating a thread, a simple approach is to define a flag variable in your server. While the flag variable is true, the server is listening for incoming connections. You can set the flag to `false`, e.g., from within the `shutdown()` method, to break the loop and terminate the main thread. Since `ServerSocket.accept()` is a blocking call,

you need to force the server thread to periodically time-out to return from the blocking method `accept()`, and check the status of the flag. The following pseudo-code shows you how I implemented this in my code. The method `ServerSocket.setSoTimeout()` takes a parameter to specify the timeout interval. See Java documentation for details. A timeout value of 1000 milli-seconds is a reasonable choice.

```
public class WebServer extends Thread {
    private volatile boolean shutdown = false;

    public WebServer(int port){
        - initialization
    }

    public void run(){
        - open the server socket
        - set socket timeout option using setSoTimeout(1000)

        while (!shutdown) {
            try {
                - accept new connection
            - create a worker thread to handle the new connection
            } catch (SocketTimeoutException e) {
                // do nothing, this is OK
                // allows the process to check the shutdown flag
            }

        } // while

        // optional: a good implementation will wait for all running worker threads to
        // terminate before terminating the server. You can keep track of your worker
        // threads using a list and then call join() on each of them. A better approach is
        // to use Java ExecutorService to schedule workers using a FixedThreadPool executor
        .

        - clean up (e.g., close the socket)
    }

    public void shutdown(){
        shutdown = true;
    }
}
```

OPTIONAL: In my implementation, I used the Java Executor service to create a fixed thread pool:

```
ExecutorService executor = Executors.newFixedThreadPool(POOL_SIZE);
```

where, `POOL-SIZE` specifies how many threads are allowed to run in parallel. The best way to set this number is to find out how many CPU cores the server machine has, but for simplicity you can just set it to a reasonable number, say 8. Then, to schedule a new worker thread, simply call `executor.execute()` and pass a worker object as argument. Note that you need to define your worker class to implement the `Runnable` interface. Finally, to wait for the running workers to terminate before terminating the server thread:

```
// shutdown the executor
try {
    // do not accept any new tasks
    executor.shutdown();

    // wait 5 seconds for existing tasks to terminate
    if (!executor.awaitTermination(5, TimeUnit.SECONDS)) {
        executor.shutdownNow(); // cancel currently executing tasks
    }
} catch (InterruptedException e) {
    // cancel currently executing tasks
    executor.shutdownNow();
}
```

Server Port Number

The server port number should be greater than 1024 and less than 65536. Most ports numbers less than 1024 are reserved for well-known applications.

Content Type

I used `Files.probeContentType()` to find the content type. There exist better and more versatile techniques if you look for them!