Laboratory 2: Sockets + Multithreading with Blocking I/O and Multiplexing I/O

Alberto ARDUSSO, Marco LUCARELLA
ardusso@eurecom.fr, lucarell@eurecom.fr

Last change: Thursday 31st March, 2016 at 02:16

Contents

1 Programming with threads

1.1 Deadlocks

In the program `deadlock.java` a deadlock occur because the threads try get a lock on two resources in two different moments. When one of these threads manage to lock the first resource, may happen that another one lock the object needed i the second moment. If that same thread, for example, needs the first resource locked by the first thread, then the two will wait indefinitely. This doesn't happen regularly because the order of the resource acquisition is not always the same.

Then a possible solution could be to fix an order for the resource acquisition. In this way it's impossible to encounter a loop of dependencies, that generates the deadlock. There are other means to avoid the problem:

1. Avoid the locking of more than one resource at a time;
2. Lock all resources needed in a single action; in this way no thread will wait for a resource while keeping another one locked.
3. Use an external mean that control all the resources and that can avoid loops. (Banker's algorithm)

Each one of this methods is not optimal because we loose in either concurrency or elasticity of our algorithm.

1.2 Race conditions

1.2.1

Running the code of `racecondition1.java`, for some threads the value of `flag` varies in a time between the setting and the testing. The solution that I propose requires a static object in `MyThread1` class. Then whenever there must be a modification of the variable `flag` we write a `synchronized` block that has as parameter the lock previously defined. We have to pay attention that each assignation and usage of the shared variable (`flag` in our case) are in the same block.

1.2.2

In the second file there are two problems:

1. The main doesn't wait for the termination of the execution of the threads; in this way the counter is printed when some thread is still running.
2. The access to the `count` variable is not synchronized. In fact the operation `count++` will be decomposed in *read*, *update* and *write*. Another thread can modify the same variable in between one of this operations.

2 Multithreaded HTTP client

Since the lab is mainly focused on the performance measurement between one thread and multi thread approach with client and server, we decided to develop a client that receive only character files in order to analyze the results easily.

2.1 Single threaded client

The `HTTPClient` class allows to create a connection to a server, get a file print it on a stream that can be a file, another socket or the standard output. This class provide three public methods:

1. `Class constructor`: creates and connect an active socket to the provided server, it also create and initialize the header structure with the default values.
2. `doGET`: configure and send the header structure in order to request a specified file, then it reads the error code returned by the server and if it is 200 it will print the requested content on the provided stream.
3. `close`: close the input and output stream and then it close the socket.

2.2 Multithreaded client

The `HTTPClientMulti` class is an upgrade of the `HTTPClient` class and in addition supports the asynchronous file transfer using threads.
The function `doGETasync` creates a thread that executes the `doGET` function in this way is possible to execute multiple request at the same time. In order to do this funciton doGET has been modified:

1. all the attribute related to the connection have become local variable inside `doGET`;
2. the connection to the server has been moved from the `Constructor` to the `doGET` in order to create different socket for each function call;
3. now the `doGET` works with filenames and not streams;
4. `TaskRequest` class has been added in order to implements different threads.

3 Multithreaded HTTP server

3.1 Blocking I/O

In our implementation of a concurrent web server we decided to collect the data related to each connection with a client in the class `ServerThread` that extends thread. In this way the synchronization to access the memory is simpler; the only shared object is a reference to the calling class that acts like the manager for the threads. In the manager itself (instance of the class `HTTPServer`) there are two collections: `threadRequests` is an array containing references for the created threads; `freeThreads` is a queue of index. When a new connection comes the main thread check if there are secondary threads in the list `freeThreads` that have finished their last job; if so, there is the method `reassign()` the allows the manager to give the thread the new connection socket. After finishing serving the client `run()` method waits over the object `newClientReady`: in the function `reassign()` this variable is notified so that the thread can start serving the new client. If there is no free thread, a new one is created and enqueued to the list.
This mechanism can lead to leakage of resources because the number of thread will be always equal to the maximum of simultaneous client ever connected. To control this problem we implemented the method `terminate()` that tells the thread to return after finishing his job, instead of waiting for another.

3.2 Blocking I/O: synchronising

For this point was necessary a synchronization for writing on a stream (in our case the standard error). Each thread while serving a client has to log a message. The problem is that while a thread try to write, another one could be accessing the same stream, resulting in inconsistencies. Our solution is a class `Log` to centralize the access to the stream. Each thread has a reference to the same `Log` instance and when it wants to log a message, it calls the method `log()` over the instance passing a String. This method is defined `synchronized` so one at a time can execute it.

4 Event-driven HTTP server

4.1 A Web Server with NIO

The implementation of the Event-driven HTTP server uses many function of the previous server such as the `Log` class, and all the function to read and parse a client request and send back the requested file.
The class used for this server are:

- `HTTPServerNB`: manage the selector and the events;
- `ClientManager`: manage the request reading and parsing and the file transmission.

The `HTTPServerNB` class starts with an initialization phase:

1. Server socket channel creation;
2. Socket binding to the provided port;

3. Selector initialization;
4. Event `ACCEPT` is registered inside the select.

After this initialization the server waits for an event on the selector and every time it occurs an if statement is used to understand the kind of event and manage it:

- Accept event: the passive socket is used to accept the client and create a new channel, then a `READ` event is registered inside the selector in order to read the client request. In this stage a `ClientManager` object is created and attached to the selector key;
- Read event: the `ClientManager` attached to the key is used to read the client request. After the reading a new key is inserted inside the selector but this time associated to the `WRITE` event.
- Write event: the `ClientManager` attached to the key is used to send back the file content requested by the client. We decided to send to the client few byte at each write request in order to not block the thread on this event. When the files end this event closes the channel.

When we switched from socket to channel we also changed the read and write procedure in order to make them compatible with channels.

4.2 Combining selector and threads

In this implementation we added a new class to manage the threads, it is called `ClientSelectThread`. Moreover we split the `select` operation in two parts: the first one inside the main thread that only manage the accept event and creates threads for each new client and a second one that manage the read and write event inside a dedicated thread. When a new accept event occur the first select uses an `Hash Map` to understand if this client has already an active thread or if it is a new client. In case of a new client the main thread creates and starts a new dedicated thread, instead id that client has already an active thread the new channel is registered on the same thread.
When all the request that belong to the same client are managed the thread ends.

5 Profiling

5.1 Profiling concurrent file transfer

The tests are done with a local installation of Apache Web Server 2.4 on a laptop. We made the client download 60 times the same web page (11,104 Bytes): with the concurrent client the time required complete the task was **2,398**±1 **ms** while the sequential implementation took just **548**±1 **ms**. The results change when we test the client with a remote server. In this case we can notice that the time required to start a thread becomes almost negligible respect to network latencies. In fact for the second test the client downloaded 60 times the home page of Eurecom website (34,086 Bytes). Now the concurrent client terminated in **3,040**±1 **ms** while downloading sequentially the pages took **53,224**±1 **ms**.
With an `ExecutorService` the time elapsed was unexpectedly more than before. My test with a remote web page resulted in **3,724±1 ms**. This can be due to network variations between tests.

5.2 Profiling concurrent client services

These tests were done creating requests for our servers with this script:

```
1  #!/bin/bash
2
3  for (( i = 0; i < 103; i++ )); do
4  wget -q http://192.168.0.28:7777/file2.txt -O $i &
5  done
```

For the completely concurrent server the results were quite varying between different executions. This was because the number of thread created is not always the same, but old threads are reused. For example we had these results:

1. **84,674**±1 **ms** with 96 threads

2. **76,790**±1 **ms** with 83 threads

Results varied also when execute the server many times without pause. The non-blocking implementation is quite more efficient: to serve the same number of requests it took **29,782**±1 **ms** with the creation of one single thread.