CS 464/546, Assignment 3

This assignment can be solved in teams of no more than *three students*,
is worth *five tokens* and
is due on *15 March at 11:59 pm*

This assignment asks for the construction of a simple bulletin board server. The server accepts one-line messages from multiple clients, stores them in a local file, and serves them back on request. Messages are identified upon storage by an unique number established by the server, and by the "sender" of the message (as provided by the `USER` command explained below).

We also start assuming a production environment so that we implement a smarter concurrency control algorithm, which turns out to be relatively simple to program yet extremely effective in real-world deployments.

1 Configuration

Upon startup, our server reads a configuration file. This file contains pairs consisting of a variable name and a value for that variable, separated by at least one blank. The configuration file includes (in no particular order) the following definitions:

`THMAX` $\mathbb{T}_{\text{max}}$
`THINCR` $\mathbb{T}_{\text{incr}}$
`BBPORT` *bp*
`FDEBUG` *df*
`BBFILE` *file*

$\mathbb{T}_{\text{max}}$, $\mathbb{T}_{\text{incr}}$, and *bp* are strings representing positive numbers. The numbers $\mathbb{T}_{\text{max}}$ and $\mathbb{T}_{\text{incr}}$ are used by the concurrency management mechanism (see Section 3.2), while *bp* is the port numbers on which the bulletin board server listens for incoming clients. The flag *df* is a Boolean value that can be provided either symbolically as `true` or `false`, or numerically as `0` or `1`. When set to true, this flag causes artificial delays in file operations, see Section 3.1. Finally, *file* is a file name (given by absolute or relative path) and specifies the file that stores all the bulletin board messages (hereinafter called the *bulletin board file*).

All the lines in the configuration file except for `BBFILE` are optional; each missing line causes the respective variable to take a default value, as follows: 25 for $\mathbb{T}_{\text{max}}$, 5 for $\mathbb{T}_{\text{incr}}$, 9000 for *bp*, and false for *df*. The server never modifies the configuration file, even if it is missing or incomplete.

The default configuration file is called `bbserv.conf` and resides in the current working directory. The name can be overridden by the sole command line argument which if present specifies (using an absolute or relative path) the configuration file to be used for the respective session.

2 Application protocol

This section describes the application protocol, i.e., of the commands send by the clients as well as the answers returned by the server, In this description fixed-width font represents text which is well, fixed for the given command or response, while parameters that may vary are shown in italics.

All the messages exchanged between the server and the client are single lines. The server should accept from clients lines terminated by any combination of the characters `'\n'` and `'\r'` (with the most common combinations being `"\n"` and `"\r\n"`). In turn all the responses from the server should be terminated by `'\n'` alone.

Connection

As soon as the client connects the server sends a one-line message as follows:

```
0.0 WELCOME ver 1.0: USER READ WRITE REPLACE QUIT spoken here
```

If you implement additional commands then those commands should also be listed in this welcome message and the version number `1.0` should be modified to `1.`x for some $x > 0$.

`USER` *name*

This command identifies the client. The argument *name* is a string not containing the character `/`. All the messages posted by the respective client (see below) will be identified as posted by *name* as provided by the most recent `USER` request. Normally, a client will send this command at the beginning of the interaction, but the server should handle the case in which this command is sent more than once during the interaction with a particular client, as well as the case when a client does not send a `USER` command at all (case in which the poster will be identified as `anonymous coward`).

The server response is the line

`1.0 HELLO` *name greeting*

where *greeting* is some (possibly empty) message intended for human consumption. If for come reason the user name is not acceptable (e.g. because it contains the `/` character) then the server should send the following response:

`1.2 BAD` *name text*

where *name* is the name provided in the client request and *text* is some free-form explanatory text intended for human consumption. This response implies that the poster name remains the same as it was before the receipt of the (faulty) `HELLO` command.

`READ` *message-number*

This command asks for the message number *message-number*. In the event that this message exists on the bulletin-board, the server will send in response one line of the form

`2.0 MESSAGE` *message-number poster/message*

where *message* represents the requested message and is prefixed by its poster *poster* as identified by the `USER` command in effect at the time of posting.

If message *message-number* does not exist, then the server sends the line

`2.1 UNKNOWN` *message-number text*

where *text* is a message for human consumption. If the server encounters an internal error while serving the request (e.g., the unavailability of the bulletin board file), then the following response is sent back to the client:

`2.2 ERROR READ` *text*

Again, *text* is an explanatory message with no particular structure.

`WRITE` *message*

Sends *message* to the server for storage. The server will store the message into the bulletin board file as a line of the form:

message-number/poster/message

where *message-number* is a unique number assigned by the server, and *poster* is the user posting the message as specified by the most recent successful `USER` command issued by the respective client (`anonymous coward` if no `USER` command has been issued by that client). Upon successful storage, the server returns

`3.0 WROTE` *message-number*

When an error occurs during the storage process, the server responds

`3.2 ERROR WRITE` *text*

The receipt of such a response should guarantee that no message has been written to the bulletin board.

`REPLACE` *message-number*/*message*

Asks the server to erase message number *message-number* and replace it with *message* (which will be assigned the same message number as the original). The poster is also changed to the current poster as identified by the most recent successful `USER` command sent by the respective client (`anonymous coward` if no such command has been issued).

The server response is identical to the response to a `WRITE` request, plus

`4.1 UNKNOWN` *message-number*

when message number *message-number* does not exist in the bulletin board file (case in which no message is added to the file).

`QUIT` *text*

Signals the end of interaction. Upon receipt of this message the server sends back the line

`9.0 BYE` *some-text*

and closes the socket. The same response (including the socket close) is given by the server to a client that just shuts down its connection. The server always closes the socket in a civilized manner.

3 Performance and other implementation requirements

Your server must be robust, in the sense that no message shall be lost when the server is killed, except possibly a message that is currently being written to the nonvolatile storage. The bulletin board file should be considered too large to be kept completely in memory.

Your server must also be efficient, in the sense that it must not rewrite the whole bulletin board file upon the receipt of each and every message. It should use the file system as little as possible (within reason and within the robustness requirements above).

Consider providing a mechanism for rolling back the most recent transaction (write or replace). You do not need to implement such a functionality, but it is recommended that you organize your code so that rollback is possible as this would become necessary in the next assignment. In general, providing a general rollback functionality (the ability to undo arbitrarily many operations) would be nice and may result in bonus marks.

You must build a concurrent server, which is able to serve many clients simultaneously. You must provide an implementation based on POSIX threads, using concurrency management and thread preallocation (as explained in Section 3.2).

3.1 Bulletin board file access

The bulletin board file is specified in the configuration file and should be created if nonexistent. Message numbers are assigned by the server in sequence, according to the order in which the `WRITE` requests have been received. No two current messages can have the same number in any bulletin board file. In particular, if the server is stopped and started again on the same file, it should inspect the file and make sure that the next message written to the file has an associated number that does not conflict with existing message numbers.

The bulletin board file must be plain text. It should be easily readable when displayed in a terminal using e.g., the `cat` shell command.

The access control to the bulletin board file follows the *readers/writers paradigm*, a common scheme in operating systems. Simultaneous reads of the bulletin board file by different threads of execution must be allowed. However, *no other operation* (read *or* write) is allowed when a thread writes to the bulletin board file. Of course, if a write request is issued while several read operations are in progress, the write will have to wait until the current reads complete. Note that this mechanism is slightly more complicated than the normal file locking. You may want to use a structure (that records the number of current read and write operations) protected by a critical region and also a condition variable to enforce this restriction. Inefficient implementations of this access control mechanism (busy waiting loops, unnecessary delays, etc.) will be penalized. *Do not use file locking for implementing the access control since this method will not work with threads.*

When the flag *df* is set to true in the configuration file the server must:

1. use suitable `sleep()` statements to introduce different delays for reading from and writing into the file (suggested: 3 seconds for reading and 6 seconds for writing), and

2. produce to the standard output suitable messages at the beginning and at the end of each file operation (meaning the actual system or library call together with the associated `sleep()` call).

The purpose of this mechanism is to facilitate the process of ensuring correct access control for the bulletin board file. I will let you figure out by yourself how to use this facility.

3.2 Concurrency management

The server uses a concurrency management similar (but not identical) to the Apache Web server. Upon startup, $\mathbb{T}_{\text{incr}}$ threads are preallocated as client handlers.

If all the preallocated threads are active serving clients, then another batch of $\mathbb{T}_{\text{incr}}$ threads is preallocated, *unless* the number of threads currently preallocated exceeds $\mathbb{T}_{\text{max}}$. The idle threads (the ones not serving clients) are periodically cleaned up as follows: Every 20 seconds the total number n of preallocated threads as well as the number of active threads are checked. If $n > \mathbb{T}_{\text{incr}}$ and less than $n - \mathbb{T}_{\text{incr}} - 1$ threads are active, then $\mathbb{T}_{\text{incr}}$ of the idle threads quit.

The overall effect is that at any given time there will be at most $\mathbb{T}_{\text{incr}} + \mathbb{T}_{\text{max}}$ threads allocated for client-server interaction, possibly less if the traffic does not justify this many threads. The number of preallocated threads increases when the traffic demands it (but up to a given maximum) and goes down progressively when the traffic lightens, so that repeated traffic spikes do not cause the server to keep deallocating threads only to create them again.

This mechanism can be implemented either using a monitor thread or in a fully distributed manner, where each thread cleans after itself with no dedicated monitor thread. You are free to choose an implementation based on a monitor thread or a distributed implementation.

4 What to submit

Submit the source code for your server plus a makefile and appropriate documentation as specified on the course's Web site. The default target in your makefile should produce an executable named `bbserv` and residing in the root directory of your submission.

The executable should not accept any command line argument except the optional name of an alternate configuration file as specified in Section 1. You are free to accept command line switches if you find them useful, but they should all be optional (and will certainly not be used during marking).

You can test your server using the client you built for the previous assignment, or build your own, more user friendly client if you feel like it. You can also use `telnet`, which incidentally speaking is going to be my main client during marking.

You can probably test your server using a single machine such as `linux.ubishops.ca`. However, if you prefer to perform more real-world-like testing you can also use the J118 machines. Given that there will likely be multiple servers running at the same time I recommend that you choose the port number for your server as 8000 + your user ID. This port number can be found at the shell prompt (on either `linux.ubishops.ca` or any of the machines in J118) using the following command:

```
echo $((8000+$(id -u)))
```

and also within C/C++ code by evaluating the expression `8000+getuid()` (see the `getuid(2)` man page for the necessary headers, though realistically speaking you should have no reasons for obtaining such a number in your code).