readme.pdf

Asst 3 WTF? By Manav P Patel and Kevin Shah

This project was an implementation of git version control. The idea of it is that many people can work on a single project at once, sending major changes into the server, which creates a new version of the project and stores it in there. At any time, they can revert the current project into any version before it, enabling a safe and efficient way to work.

This project touched mainly on three big ideas. One of them was parsing .Manifest files rather than comparing projects byte by byte for updates/adds/deletes. This .Manifest file contained a encrypted hash code of a file, which is guaranteed to be unique to any version of a file. Another big idea was networking. This project utilized sockets, to write and read data across server and client. This allowed a repository to be stored in one place, essentially not in the client's computer, to be access by certain calls. The last concept required for this project was thread synchronization. The server on receiving a request by a client socket, would generate a thread to handle a specific call by the client. The ingenuity of this, is that the server can theoretically can generate unlimited threads to handle unlimited amount of request by many clients simultaneously. However, the problem with this, is that the threads needed to be synchronized in order to prevent threads from reading/and writing into the same resource (files from projects) and the method used to do this will be discussed.

<u>Thread Synchronization</u>

To handle threads, we first create an array of pthreads of a set size. Upon receiving a call from a client, the server branches the client into a new thread, which is then taken to the main_process which then handles all types of requests by sending specific requests into specific functions. If the amount of pthreads approaches the set size of the array, it joins the threads together, and then resumes. This way, there aren't so many threads that the CPU gets clogged.

To stop one thread from reading/writing into a shared resource while another thread was also doing the same, we used mutexes. One of the muteness was a global mutex which held the lock for the entire repository. This lock and unlock had to be called each time the request of push or destroy were called. We did this by calling lock() and unlock() one after the other, in any other instances that it needed to read and write, and on the cases of push/destroy, the lock was first called, then the reading/writing happened, then the unlock() was called.

For locking individual projects, a struct node that matched the project name with its associated mutex was created. Each time a new project was created, a node was appended to the end of the list. Each time a a read/write was called for a project, the general idea was to search through this list for a project name. IF its as unlocked, it would lock it, and at the end of the call, it would unlock it.

We avoided deadlocks by severe error checking within the code, and if an error was detected immediately the mutex was unlocked.

<u>THE REST:</u>

The rest of the code was fairly simple. It made generous use of parsing functions to store the .Manifests in hash table form for easy search operations. All the operations were then implemented on said hash table, and then a make_manifest function would then print a .Manifest file from this.