Asst3 Readme

1. <u>Actions that Involve System Calls</u>
   a. Client/Server Communication
      - When WTF client issues a command, it may communicate with the WTF server via a socket, depending on the command.
      - The server creates a thread to serve each client, and may serve multiple clients concurrently.
      - The client and server each have one function to handle each WTF command in both client and server programs except for configure, add or remove, which happen on the client side only.
      - Client and server functions that handle the same WTF are handshaking, which means passing data between each other.
      - Mutexes are used to allow multiple client threads to take turns to access the critical section of code in the server.
      - Signal handler for SIGINT (CTRL-C) is registered.

   b. Socket Communication

      The steps involved in establishing a socket on the client side are:

      1) Create a socket with the socket() system call
      2) Connect the socket to the address of the server using the connect() system call
      3) Send and receive data - there are a number of ways to do this, such as to use the read() and write() system calls.

      The steps involved in establishing a socket on the server side are:

      1) Create a socket with the socket() system call
      2) Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
      3) Listen for connections with the listen() system call
      4) Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
      5) Send and receive data

   c. Implementation of Mutexes

        - Mutexes are used for thread synchronization to prevent threads from simultaneously executing on some particular program segment known as the critical section.
        - Mutexes are initialized for each project when the server is up.
        - A mutex lock is acquired at the project level for any commands involving accessing and changing files on the server side, and the lock is released when the command completes.
    d. The SHA-1 cryptographic hash function is used to generate a unique hash code for each file so that you can compare the uniqueness of any two files.

2. <u>File Structure</u>
    a. WTF.c handles all client side commands - configure, create, checkout, update, upgrade, commit, push, add, remove, destroy, currentversion, history, rollback.
    b. WTFserver.c contains functions handling signals or requests from the client side and handling action on the requests - create, checkout, update, upgrade, commit, push, destroy, currentversion, history, rollback.
    c. WTF_util.c and WTFserver_util.c contain utility functions for linked list methods, file/folder methods, file hashing using the SHA1 function, and receiving and sending one file via a socket.
    d. Running "make all" will compile the WTF and WTFserver programs in the Asst3 root folder. The folder structure is:

```
Asst3
    WTF
    WTFserver
    Project1 (client project1)
    Project2 (client project2)
    ...
    Projectn (client project)
    Repository (server side)
        Project1 (server project1)
            master (current version)
                dir1
            file1, file2 ...
        dir2
            subfolder1
                file3, file4 ...
    v0.tar.bz2 (archived version 0)
    v1.tar.bz2 (archived version 1)
    ...
    vn.tar.bz2 (archived version n)
Project2 (server project2)
    master
    v0.tar.bz2
    v1.tar.bz2
...
```

3. <u>Important Files involved</u>
   a. .Manifest – exists on both of the client and server sides, and records the current project version, file path and names, file version, and file hash code

      projNum, newline
      path/name1,tab,fileVersionNum,tab,hashcode,newline
      path/name2,tab,fileVersionNum,tab,hashcode,newline
      ...

   b. .Commit – records the files on the client side that need to be Added (Add), Removed (Remove) or Copied (Copy) to the server, and may exist on both the client and server side.

      Add,tab,path/name1,tab,fileVersionNum,tab,hashcode,newline
      Remove,tab,path/name2,tab,fileVersionNum,tab,hashcode,newline
      Copy,tab,path/name2,tab,fileVersionNum,tab,hashcode,newline
      …

   c. .Update – records the files on the server side that need to be Added (A), Removed (D) or Copied (M) to the client, and may exist on both the client and server side.

      A,tab,path/name1,tab,fileVersionNum,tab,hashcode,newline
      D,tab,path/name2,tab,fileVersionNum,tab,hashcode,newline
      M,tab,path/name2,tab,fileVersionNum,tab,hashcode,newline
      …

d. .History – a log file that records all successful push and rollback history

   Push
   1
   Add,tab,path/name1,tab,fileVersionNum,tab,hashcode,newline
   Remove,tab,path/name2,tab,fileVersionNum,tab,hashcode,newline
   …
   Rollback
   0

e. .Configure – stores the port and server IP or name on the client side for future connection to a server

4. <u>Data Structures</u>
   a. Linked List
      i. Information about the file path/name, file version number, and file hash code is loaded from a .Manifest file into a linked list
      ii. The linked list facilitates adding and removing files by inserting and deleting nodes
      iii. The linked list facilitates comparing files on the client and server side to see whether any of the file have been added, deleted, or changed
   b. Array
      i. Stores information about the project name and the mutex that corresponds to the project
5. <u>Implementation of commands</u>

a. configure
   1) On the client side, store the server port and IP address in a .Configure file so that you can use this information for any subsequent client to server command connection
   2) Any server-bound client command needs to check the existence of such a file. If the file does not exist, the command will produce an error

b. create
   1) The client side sends this command to the server with a project name to create.
   2) The server checks if the project pre-exists, if so, it will send an error the client, and the command will exit.

    3) Acquire a mutex lock.
    4) On the server side, the server will create a project folder under "Repository" (which will also be created if does not pre-exist), a master folder under the project folder, and a .Manifest file inside the master folder indicating that you just created version 0 of the project.
    5) The server sends the .Manifest file to the client, which will create the project folder and place the manifest file inside it.
    6) Release the mutex lock.

c. checkout
    1) The client side will first check if a project folder already exists with the intended checkout project name, and if so, it will produce an error.
    2) The server side will check whether the project exists, and if not, it will produce an error.
    3) Acquire a mutex lock.
    4) The server side checkout function will tar-zip the master folder for the requested project, and send the zipped file to the client.
    5) Release the mutex lock.
    6) The client will un-tarzip the file and place the project folder into the client folder.

d. update
    1) Check if the project exists on the server side, and raise an error if it does not.
    2) Acquire a mutex lock.
    3) Request a .Manifest file from the server named as .serverManifestForUpdate, and load its contents into a linked list
    4) Release the mutex lock.
    5) Load the contents of the client .Manifest file into a linked list
    6) Re-generate the hash code for all client files in the .Manifest file and update the hash codes in the client linked list
    7) Compare the client and server linked lists, and write the differences between them in the .Update file on the client side. M stands for modify

where a file exists on both client and server side but with different version number and hash code, A means that a file is in the server's .Manifest file but not in the client's, and D means that a file is in the client's .Manifest file but not the server's.

8) Report that the "client is up to date" if no differences are found between the .Manifest files.

e. upgrade

1) Check if the project exists on the server side, and raise an error if it does not.
2) Check if .Update exists on the client side, and raise an error if it does not.
3) Acquire a mutex lock.
4) The client will apply the changes listed in the .Update file by deleting files listed as "D", and requesting files from the server which are listed as "A" or "M"
5) Release the mutex lock.
6) Delete the .Update file.

f. commit

1) Check if the client side has a .Update file for the project, and raise an error if it does.
2) Check if the project exists on server side and if the .Manifest file exists for the project on the server side, and if it does not, raise an error
3) Acquire a mutex lock.
4) The server sends the .Manifest file to the client side as .serverManifest.
5) Load the contents of the .serverManifest file into a linked list.
6) Load the contents of the client .Manifest file into a linked list.
7) Compare the client and server project versions, and raise an error if they are different.
8) Re-generate the hash code for all client files in the .Manifest file and update the hash codes in the client linked list.

9) Compare the client and server linked lists, and write the differences between them into a .Commit file on client side. The differences include Add for new files, Remove for deleted files, and Copy for a new version of the files whose hash code differ from the server hash code and the file version number is incremented by one. Make sure that the file version number is higher than the server's file version number, and raise an error if the server's file version number is not lower than the updated client file version number.
10) Send the client .Commit file to server side and name it as .CommitActive under the project folder.
11) Release the mutex lock.

g. push

1) Check if the client side has a .Update file for the project, and if so, raise an error.
2) Check if the client side has a .Commit file, and if not, raise an error.
3) Acquire a mutex lock.
4) Send the push command to the server, and check if the project exists on the server and if the .CommitActive file exists on the server. If either is not present, raise an error.
5) The client sends a .Commit file to the server as .CommitSecond, and compares the hash code of .CommitSecond with .CommitActive. If they are different, raise an error to show that there was a push failure.
6) Compress the existing master folder, which is version [n], into v[n].tar.bz2 which is the archive of the [n]th version of the project. This v[n] can be unzipped during any future rollback.
7) Load the contents of the server .Manifest file into the linked list, and guide the client to apply commands from the .Commit file while updating the manifest linked list to the updated file version number and hash:
   'c' for copy a new version of an existing file;
   'a' for adding a new file;

'r' for removing an existing file from the project.

8) Copy the contents of the updated manifest linked list to the new .manifest file, and increase the project version number by 1.
9) Send the updated .manifest file to the client side
10) Remove the .Commit file from the client and the .CommitActive file from the server
11) Write the project version number and commands from .Commit to .History under the project folder so that you are recording the changes.
12) Release the mutex lock.

h. add

1) Check if the project exists in client, and raise an error if it does not.
2) Check if the file to be added exists in the client side, and raise an error if it does not.
3) Load the contents of the client .Manifest file to the linked list.
4) Check if the file is already in the .Manifest linked list, and raise an error if it is.
5) Add the file path and name, version 1, and the hash code of the file to the linked list as a new node.
6) Write the contents of the linked list to the .Manifest file, overwriting the old file.

i. remove

1) Check if the project exists in client, and raise an error if it does not.
2) Check if the file to be removed exists on the client side, and raise an error if it does not.
3) Load the contents of the client .Manifest file to the linked list.
4) Find the node for the file to be removed and delete it from the linked list. Raise error if you cannot find the node.
5) Write the contents of the linked list to the .Manifest file, overwriting the old file.

j. rollback

1) The client sends the request, with the project name, to server. Check if the project exists in server, and raise an error if it does not.
2) Acquire a mutex lock.
3) Check if the rollback version number is invalid, meaning that it is not smaller than current version number. Raise an error if it is.
4) Check if the rollback version archive (vn.tar.bz2) exists, and raise an error if it does not.
5) Write the rollback action into the .History file, and remove any commits that are later than the rollback version.
6) Release the mutex lock.

k. destroy

1) The client sends the request, with the project name, to the server.
2) Check if the project exists in server, and raise an error if it does not.
3) Acquire the mutex lock.
4) The server removes the folder for the specified project.
5) Release the mutex lock.
6) The server sends a status back to the client about whether the action was successful or not.

l. currentversion

1) The client sends the request, with the project name, to the server.
2) Check if the project exists in server, and raise an error if it does not.
3) Acquire a mutex lock.
4) The server sends the .Manifest file from the master folder to the client and names it as .tmp_Manifest
5) Release the mutex lock.
6) The client outputs the project name and the contents of the .tmp_Manifest file including the project version number, file path and file names, file version numbers and file hash codes

m. history

1) The client sends the request, with the project name, to the server.
2) Check if the project exists in server, and raise an error if it does not.

3) Acquire a mutx lock.
4) The server sends the .History file from the project folder to the client and names it as .tmp_History
5) Release the mutex lock.
6) The client outputs the contents of the .tmp_History file which includes all of the successful push and rollback actions

6. Implementation of Compress/Decompress
    a. In the checkout command, the server compresses the master folder into one file and sends it to the client side. The client decompresses this file into a project folder on the client side.
    b. In the push command, the existing master folder will be compressed into v[n].tar.bz2 as the archive of the [n]th version of the project. The master folder will then go through the Add/Remove/Copy actions to get the items pushed from the client.
    c. In the rollback command to [n]th version on the server side, the archived v[n].tar.bz2 will be decompressed into the master folder to replace the old master, and any archived compressed version that is older than nth version will be removed.