For the default project, you are to revisit the Fabricated Files and Folders (F^3) assignment, extending it to support three main features:

* Functional permissions
* Users and groups
* Process execution and scheduling

In the original F^3 assignment, you were to implement permission strings for the objects the user created. Now you are to make those objects functional. The permissions should be enforced as follows:

**Directories:  
read allows contents to be listed (via ls)  
write allows contents to be added/modified (via touch, rm, mkdir, rmdir)  
execute allows the directory to be part of the current path (i.e., allows cd'ing into/past it)**

**Files:  
read allows the file to be read (not exercisable, but feel free to add a cat command to test this)  
write allows the file mod time to be updated (via touch)  
execute allows the file to be ran (via run)**

**Both:  
write allows attribute updates via chmod/chown/chgrp**

Recall that the permission sets were for the object owner, the object owner’s group, and everyone else. In order to make each of those sets usable, you will need to introduce users and groups, as well. At startup you will have one user and one group: Root and Users, respectively. These cannot be removed from the system. Root should operate as a super user, essentially meaning permissions do not restrict it.

When users are added to the system, they will be added to the group Users. This group will initially be the users’ *primary group*. A *primary group* is the group that receives permissions for anything that the user creates (there is an example of this below). Note: users can be members of multiple groups, but only one of those groups will be the user’s primary group. New commands needed for this:

**useradd <username>**  
Creates a new user and adds them to the Users group (and sets Users as their primary group)

**useradd -G <group[,group]> <username>**Creates a new user and adds them to the Users group (and sets Users as their primary group). Also adds the user to the additional groups indicated (if a group doesn’t exist, the command will still function for other groups, but should notify the user of the non-existent group). Note: there should be no whitespace in the comma separated group list

**chuser <username>**  
Change the active user to the one indicated. Fails if: the user doesn’t exist

**groupadd <group>**  
Creates a new group and adds the Root user to the group

**usermod -g <group> <username>**Set the primary group for the indicated user. Fails if: the user doesn’t exist, the group doesn’t exist, or the user not part of the group

**usermod -a -G <group> <username>**Add the indicated user to the indicated group. Fails if: the user doesn’t exist or the group doesn’t exist

**chown <username> <object>**  
Change the owner of the indicated object (file, directory) to the indicated user. Fails if: the active user doesn’t have write permission on the object, the indicated object doesn’t exist, the indicated user doesn’t exist

**chgrp <group> <object>**Change the group of the indicated object (file, directory) to the indicated group. Fails if: the active user doesn’t have write permission on the object, the indicated object doesn’t exist, the indicated user doesn’t exist

**userdel -G <group> <username>**Remove the indicated user from the indicated group. Fails if: the user doesn’t exist, the group doesn’t exist, the user not part of the group, the group is Users, the user is Root

**userdel <username>**  
Remove the user from the system (any owned objects become owned by the Root user). Fails if: the user doesn’t exist or the user indicated is Root

**groupdel <group>**Remove the group from the system (any objects with permissions for this group are changed to the Users group). Fails if: the group doesn’t exist or the group indicated is Users

**groups <username>**  
List the groups that the indicated user is part of. Fails if: the user doesn’t exist

**users**  
List the users known to the system.

So, given this command string immediately following startup:

1) groupadd Student  
2) groupadd Prof  
3) useradd -G Prof,Student jw  
4) usermod -g Prof jw  
5) useradd -G Student joeSchmoe  
6) usermod -g Student joeSchmoe  
7) touch Welcome.txt  
8) chmod 704 Welcome.txt  
9) chuser jw  
10) mkdir homework  
11) chmod 775 homework  
12) mkdir solutions  
13) chmod 770 solutions  
14) chuser joeSchmoe  
15) cd solutions  
16) chmod 777 solutions  
17) cd homework  
18) touch jsHomework.cpp

After this, we should have three known users:  
 -Root: member of Users (primary group), Prof, and Student  
 -jw: member of Users, Prof (primary group), and Student  
 -joeSchmoe: member of Users and Student (primary group)  
The root directory should contain two directories:   
 -homework: owned by jw and Prof with permission string rwxrwxr-x  
 -solutions: owned by jw and Prof with permission string rwxrwx---  
The root directory should contain one file:  
 -Welcome.txt: owned by Root and Users with permission string rwx---r--  
And inside homework there should be one file:  
 -jsHomework.cpp: owned by joeSchmoe and Student with default permission string  
And lastly, commands 15 and 16 should both fail (insufficient permissions for the active user).

Now for the next addition: you are to implement a COMPLETELY FAKE process execution system to your shell. This should be implemented as a second thread to the main shell. A run command will be added to the shell that will “run” any file created by the shell users (for all intents and purposes, all files are runnable processes for this assignment). When files are created with the touch command, they will be given a random number of time units needed when executed (generated by the shell/your program automatically). This number should be bound between 10 and 50 (for starters, feel free to play with the bounds though).

The process management thread will perform process scheduling in a manner similar to the Scheduler Showdown assignment (a process will be selected each timestep according to the active scheduling algorithm in the system, which you are free to select on your own). When a user “runs” a file, a process needs to be added to the pool of schedulable processes that the thread is checking at a regular rate. Feel free to use code from the scheduler framework for this, the main difference here is that processes are given to the scheduler incrementally instead of all at once in a file. New commands needed in the shell for this:

**run <file>**  
“Run” the indicated file. The running version of the file will have the active user as it’s owner (even if the user is not the owner of the file itself). Fails if: the file does not exist or the active user does not have execute permissions on the indicated file

**ps**  
View all “running” files/processes on the system. This will differ from Linux ps, you should show file/process name, file/process owner, when the process arrived, how much time the process has run, and how much time total the process will need

**kill <file>**  
Terminate an executing file/process prematurely. Fails if: the file is not currently running or the active user is not the owner of the file/process

**schedHist**  
Output the scheduling history, indicating the time step and the process (omitting any idle times).

For example, if the following commands were executed:  
 touch fileA.txt  
 touch fileB.txt  
 touch fileC.txt  
 run fileA.txt  
 run fileB.txt  
 run fileC.txt  
 kill fileB.txt

Running ps might have output along the lines:  
 Process - Owner - Time Started - Time Scheduled - Time Needed  
 fileA.txt - Root - 12 - 5 - 13  
 fileC.txt - Root - 16 - 3 - 20  
  
And running schedHist might have output like:  
 TimeStep - Process  
 12 - fileA.txt  
 13 - fileA.txt  
 14 - fileA.txt  
 15 - fileB.txt  
 16 - fileB.txt  
 17 - fileC.txt  
 18 - fileC.txt  
 19 - fileC.txt  
 20 - fileA.txt  
 21 - fileA.txt