**CMPS 12M**
**Data Structures Lab**
**Lab Assignment 1**

The purpose of this assignment is threefold: (1) get a basic introduction to the Andrew File System (AFS) which is the file system used by the ITS unix timshare, (2) learn how to create an executable jar file containing a Java program, and (3) learn to automate compilation and other tasks using Makefiles.

**AFS**

Logon to your ITS unix timeshare account at `unix.ucsc.edu`. If you don't know how to do this, ask for help at a lab session, or see Lab Assignment 1 from my CMPS 12A Winter 2016 webpage:

https://classes.soe.ucsc.edu/cmps012a/Winter16/lab1.pdf

Create a subdirectory within your home directory called `cs12b` in which you will keep all your work for both CMPS 12B and CMPS 12M. Create a subdirectory within `cs12b` called `lab1`. From within `lab1` create a subdirectory called `private`, then set access permissions on the new directory so that other users cannot view its contents. Do all this by typing the lines below. The unix prompt is depicted here as `%`, although it may look different in your login session. Those lines without the unix prompt are the output of your typed commands.

```
% mkdir cs12b
% cd cs12b
% mkdir lab1
% cd lab1
% mkdir private
% fs setacl private system:authuser none
% fs listacl private
Access list for private is
Normal rights:
  foobar rlidwka
```

Here `foobar` will be replaced by your own cruzid. The last line of output says that your access rights to directory `private` are `rlidwka` which means: read, list, insert, delete, write, lock, and administer. In other words you have all rights in this directory, while other users have none. If you are unfamiliar with any unix command, you can view its manual page by typing: `man <command name>`. (Do not type the angle brackets `<>`.) For instance `man mkdir` brings up the man pages for `mkdir`. Man pages can be very cryptic, especially for beginners, but it is best to get used to reading them as soon as possible.

Under AFS, `fs` denotes a file system command, `setacl` sets the access control list (ACL) for a specific user or group of users, and `listacl` displays the access lists for a given directory. The command

```
% fs setacl <some directory> <some user> <some subset of rlidwka or all or none>
```

sets the access rights for a directory and a user. Note that `setacl` can be abbreviated as `sa` and `listacl` can be abbreviated as `la`. For instance do `la` on your home directory:

```
% fs la ~
Access list for /afs/cats.ucsc.edu/users/a/foobar is
Normal rights:
  system:authuser l
  foobar rlidwka
```

The path `/afs/cats.ucsc.edu/users/a/foobar` will be replaced by the full path to your home directory, and your own username in place of `foobar`. Note that `~` (tilde) always refers to your home directory, `.` (dot) always refers to your current working directory (the directory where you are currently located) and `..` (dot, dot) refers to the parent of your current working directory. The group `system:authuser` refers to anyone with an account on the ITS unix timeshare. Thus by default, any user on the system can list the contents of your home directory. No other permissions are set for `system:authuser` however, so again by default, no one else can read, insert, delete, write, lock, or administer your files.

Do `fs la ~/cs12b` and verify that the access rights are the same for the child directory `cs12b`. Create a subdirectory of `private`, call it anything you like, and check its access rights are the same as for its parent. Thus we see that child directories inherit their permissions from the parent directory when they are created. To get a more comprehensive list of AFS commands do `fs help`. For instance you will see that `fs lq` shows your quota and usage statistics. For more on the Andrew File System go to

http://claymore.rfmh.org/public/computer_resources/AFSinfo.html.

**Jar Files**

Copy the following file to your `lab1` directory. (You can find this file in the Examples section of the class webpage.)

```
//-----------------------------------------------------------------------------
// HelloUser.java
// Prints greeting to stdout, then prints out some environment information.
//-----------------------------------------------------------------------------
class HelloUser{
   public static void main( String[] args ){
      String userName = System.getProperty("user.name");
      String os       = System.getProperty("os.name");
      String osVer    = System.getProperty("os.version");
      String jre      = System.getProperty("java.runtime.name");
      String jreVer   = System.getProperty("java.runtime.version");
      String jvm      = System.getProperty("java.vm.name");
      String jvmVer   = System.getProperty("java.vm.version");
      String javaHome = System.getProperty("java.home");
      long freemem    = Runtime.getRuntime().freeMemory();
      long time       = System.currentTimeMillis();

      System.out.println("Hello "+userName);
      System.out.println("Operating system: "+os+" "+osVer);
      System.out.println("Runtime environment: "+jre+" "+jreVer);
      System.out.println("Virtual machine: "+jvm+" "+jvmVer);
      System.out.println("Java home directory: "+javaHome);
      System.out.println("Free memory: "+freemem+" bytes");
      System.out.printf("Time: %tc.%n", time);
   }
}
```

You can compile this in the normal way by doing `javac HelloUser.java` then run it by doing the command `java HelloUser`. Java provides a utility called `jar` for creating compressed archives of executable `.class` files. This utility can also be used to create an executable `jar` file that can then be run by just typing its name at the unix prompt (with no need to type `java` first). To do this you must first create a manifest file that specifies the entry point for program execution, i.e. which `.class` file contains the `main()` method to be executed. Create a text file called Manifest containing just one line:

```
Main-class: HelloUser
```

If you don't feel like opening up an editor to do this you can just type

```
% echo Main-class: HelloUser > Manifest
```

The unix command `echo` prints text to stdout, and `>` redirects the output to a file. Now do

```
% jar cvfm HelloUser Manifest HelloUser.class
```

The first group of characters after `jar` are options. (c: create a jar file, v: verbose output, f: second argument gives the name of the jar file to be created, m: third argument is a manifest file.) Consult the man pages to see other options to `jar`. The second argument `HelloUser` is the name of the executable jar file to be created. The name of this file can be anything you like, i.e. it does not have to be the same as the name of the `.class` file containing function `main()`. For that matter, the manifest file need not be called `Manifest`, but this is the convention. Following the manifest file is the list of `.class` files to be archived. In our example this list consists of just one file: `HelloUser.class`. At this point we would like to run the executable jar file `HelloUser` by just typing its name, but there is one problem. This file is not recognized by Unix as being executable. To remedy this do

```
%chmod +x HelloUser
```

As usual, consult the man pages to understand what `chmod` does. Now type `HelloUser` to run the program. The whole process can be accomplished by typing five lines:

```
% javac –Xlint HelloUser.java
% echo Main-class: HelloUser > Manifest
% jar cvfm HelloUser Manifest HelloUser.class
% rm Manifest
% chmod +x HelloUser
```

Notice we have removed the now unneeded manifest file. Note also that the `–Xlint` option to `javac` enables recommended warnings. The only problem with the above procedure is that it's a big hassle to type all those lines. Fortunately there is a unix utility that can automate this and other processes.

**Makefiles**

Large programs are often distributed throughout many files that depend on each other in complex ways. Whenever one file changes all the files depending on it must be recompiled. When working on such a program it can be difficult and tedious to keep track of all the dependencies. The Unix `make` utility automates this process. The command `make` looks at dependency lines in a file named `Makefile`. The dependency lines indicate relationships between source files, indicating a *target* file that depends on one or more *prerequisite* files. If a prerequisite has been modified more recently than its target, `make` updates the target file based on *construction commands* that follow the dependency line. `make` will normally stop if it encounters an error during the construction process. Each dependency line has the following format.

```
target: prerequisite-list
      construction-commands
```

The dependency line is composed of the `target` and the `prerequisite-list` separated by a colon. The `construction-commands` may consist of more than one line, but each line *must* start with a tab character. Start an editor and copy the following lines into a file called `Makefile`.

```
# A simple Makefile
HelloUser: HelloUser.class
	echo Main-class: HelloUser > Manifest
	jar cvfm HelloUser Manifest HelloUser.class
	rm Manifest
	chmod +x HelloUser

HelloUser.class: HelloUser.java
	javac -Xlint HelloUser.java

clean:
	rm -f HelloUser HelloUser.class

submit: README Makefile HelloUser.java
	submit cmps012b-pt.s16 lab1 README Makefile HelloUser.java
```

Anything following `#` on a line is a comment and is ignored by `make`. The second line says that the target `HelloUser` depends on `HelloUser.class`. If `HelloUser.class` exists and is up to date, then `HelloUser` can be created by doing the construction commands that follow. Remember that *all* indentation is accomplished via the tab character. The next target is `HelloUser.class` which depends on `HelloUser.java`. The next target `clean` is what is sometimes called a *phony target* since it doesn't depend on anything and just runs a command. Likewise the target `submit` does not compile anything, but does have some dependencies. Any target can be built (or perhaps executed if it is a phony target) by doing `make <target name>`. Just typing `make` creates the first target listed in the Makefile. Try this by doing `make clean` to get rid of all your previously compiled stuff, then do `make` again to see it all created again. Your output from `make` should look something like:

```
% make
javac -Xlint HelloUser.java
echo Main-class: HelloUser > Manifest
jar cvfm HelloUser Manifest HelloUser.class
added manifest
adding: HelloUser.class(in = 1577) (out= 843)(deflated 46%)
rm Manifest
chmod +x HelloUser
```

The `make` utility allows you to create and use macros within a Makefile. The format of a macro definition is `ID = list` where `ID` is the name of the macro (by convention all caps) and `list` is a list of filenames. Then `$(list)` refers to the list of files. Move your existing Makefile to a temporary file, then start your editor and copy the following lines to a new file called `Makefile`.

```
#-----------------------------------------------------------------------------
# A Makefile with macros
#-----------------------------------------------------------------------------

JAVASRC    = HelloUser.java
SOURCES    = README Makefile $(JAVASRC)
MAINCLASS  = HelloUser
CLASSES    = HelloUser.class
JARFILE    = HelloUser
SUBMIT     = submit cmps012b-pt.s16 lab1
```

```
all: $(JARFILE)

$(JARFILE): $(CLASSES)
	echo Main-class: $(MAINCLASS) > Manifest
	jar cvfm $(JARFILE) Manifest $(CLASSES)
	rm Manifest
	chmod +x $(JARFILE)

$(CLASSES): $(JAVASRC)
	javac -Xlint $(JAVASRC)

clean:
	rm $(CLASSES) $(JARFILE)

submit: $(SOURCES)
	$(SUBMIT) $(SOURCES)
```

Run this new Makefile and observe that it is equivalent to the previous one. The macros define text substitutions that happen before `make` interprets the file. Study this new Makefile until you understand exactly what substitutions are taking place. Now create your own Hello program and call it `HelloUser2.java`. It can say anything you like, but just have it say something different from the original. Add `HelloUser2.java` to the `JAVASRC` list, add `HelloUser2.class` to the `CLASSES` list and change `MAINCLASS` to `HelloUser2`. Also change the name of `JARFILE` to just `Hello` (emphasizing that the jar file can have any name.)

```
#-----------------------------------------------------------------------------
# Another Makefile with macros
#-----------------------------------------------------------------------------

JAVASRC    = HelloUser.java HelloUser2.java
SOURCES    = README Makefile $(JAVASRC)
MAINCLASS  = HelloUser2
CLASSES    = HelloUser.class HelloUser2.class
JARFILE    = Hello
SUBMIT     = submit cmps012b-pt.s16 lab1

all: $(JARFILE)

$(JARFILE): $(CLASSES)
	echo Main-class: $(MAINCLASS) > Manifest
	jar cvfm $(JARFILE) Manifest $(CLASSES)
	rm Manifest
	chmod +x $(JARFILE)

$(CLASSES): $(JAVASRC)
	javac -Xlint $(JAVASRC)

clean:
	rm $(CLASSES) $(JARFILE)

submit: $(SOURCES)
	$(SUBMIT) $(SOURCES)
```

This new Makefile compiles both HelloUser classes (even though neither one depends on the other.) Notice however the entry point for program execution has been changed to function `main()` in your

program `HelloUser2.java`. Macros make it easy to make changes like this, so learn to use them. To learn more about Makefiles follow the links posted on the class webpage.

We've discussed three `Makefiles` in this project. If you rename them `Makefile1`, `Makefile2` and `Makefile3` respectively (since you can't have three files with the same name), you'll find that the `make` command does not work since a file called `Makefile` no longer exists. Instead of renaming files to run the Makefile you want, you can use the `-f` option to the `make` command, and specify the name of your Makefile. For instance

```
% make -f Makefile2
```

runs `Makefile2`. If you want to specify something other than the first target, place it after the file name on the command line. For instance

```
% make –f Makefile3 clean
```

runs the `clean` target in `Makefile3`.

**What to turn in**

All files you turn in for this and other assignments (in both 12B and 12M) should begin with a comment block giving your name, cruzid, class (12B or 12M), date, a short description of its role in the project, the file name and any special instructions for compiling and/or running it. Create one more file called README which is just a table of contents for the assignment. In general README lists all the files being submitted (including README) along with any special notes to the grader. Use what you've learned in this project to write a `Makefile` that creates and archives both `.class` files (`HelloUser.class` and `HelloUser2.class`) in an executable jar file called `Hello` that runs function `main()` from `HelloUser2.class`. It should also include `clean` and `submit` utilities, as in the above examples. Also add a `check` utility that checks that the project was properly submitted.

See the webpage for instructions on using the `submit` command and for checking that a project was properly submitted. Submit the following files to the assignment name lab1:

| | |
|---|---|
| `README` | described above |
| `Makefile` | described above |
| `HelloUser.java` | unchanged from above |
| `HelloUser2.java` | described above |

You can either type the submit command directly:

```
% submit cmps012b-pt.s16 lab1 README Makefile HelloUser.java HelloUser2.java
```

or use the Makefile itself:

```
% make submit
```

This is not a difficult assignment, especially if you took CMPS 11 or CMPS 12A from me (see lab4 from those classes), but start early and ask questions in lab section or office hours if anything is unclear.