**CMPS 12M**
**Introduction to 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.ic.ucsc.edu`. (If you don't know how to do this, you will need to attend a lab session to learn how.) From within your home directory, create a directory called `private`, then set access permissions on the new directory so that other users cannot view it's contents. Do all this by typing the following lines. The unix prompt is depicted below as `%`, although it may look different in your login session. Those lines without the unix prompt are what is typed at you by the operating system.

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

Here `foobar` will be replaced by your own username. The last three lines say 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. This creates a space where your own work can be kept secure and confidential. If you are unfamiliar with any unix command, you can view its manual page by typing: `%man <command name>`. For instance `%man mkdir` brings up the man pages for `mkdir`. Man pages can be notoriously cryptic, 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. In general

```
% fs setacl <some directory> <some username> <some subset of rlidwka 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 .
Normal rights:
  system:anyuser rl
  foobar rlidwka
```

Here again you will see your own username in place of `foobar`. Note that `.` always refers to your current working directory, i.e. the directory in which you are presently located. (Also `..` refers to the parent of your current working directory, while `~` refers to your home directory.) The group `system:authuser` refers to anyone with an IC Unix computer account, and `system:anyuser` refers to anyone anywhere in the world running AFS.

Now that you have a protected space for your files, `cd` (change directory) into that directory, and organize your files for this course by creating a hierarchy of directories.

```
% cd private
% mkdir cmps012m
% mkdir cmps012b
% cd cmps012m
% mkdir lab1
```

It is not necessary to reset the ACL's on these child directories, since they are governed by the parent directory `private`. 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.

**Jar Files**

Use your favorite editor to copy the following file to `~/private/cmps012m/lab1`. (Recall `~` always refers to your home directory.)

```
// hello. java
// Prints "Hello World" to stdout, then prints out some
// environment information.

import static java.lang.System.*;

class hello{

   public static void main( String[] args ){
      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 home = System.getProperty("java.home");
      double freemem = Runtime.getRuntime().freeMemory();
      long time = currentTimeMillis();

      System.out.println("Hello, World!");
      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: "+home);
      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 hello.java` then run with `%java hello`. Java provides a utility called `jar` for creating compressed archives of `.class` files. This utility can also be used to create an executable `jar` file which 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 which specifies the entry point for program execution, i.e. which `.class` file contains the main method to be executed. Create a file called Manifest containing just one line:

```
Main-class: hello
```

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

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

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

```
%jar cvfm myHello Manifest hello.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`. Following the manifest file is the list of `.class` files to be archived, which in our example consists of just one file, `hello.class`. The name of the executable jar file (second argument) can be anything you like. In this example it is called `myHello` to emphasize that it need not have the same name as the corresponding `.class` file. (For that matter, the manifest file need not be called `Manifest`.) At this point we should be able to run the executable jar file `myHello` 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 myHello
```

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

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

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 which can automate this and other processes.

**Makefiles**

Large programs are often distributed throughout many files which depend on each other in complex ways. Whenever one file changes, all the files which depend on that file must be recompiled. When working on such a program it can be difficult and tedious to keep track of all the dependency relationships. The **make** utility automates this process. **Make** looks at dependency lines in a file named "makefile" stored in the current working directory. The dependency lines indicate relationships among files, specifying a **target** file that depends on one or more **prerequisite** files. If a prerequisite file has been modified more recently than its target file, **make** updates the target file based on **construction commands** that follow the dependency line. **Make** normally stops 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** line *must* start with a tab character and must follow the dependency line. Start an editor and copy the following lines into a file called "Makefile".

```
# A simple makefile
myHello: hello.class
	echo Main-class: hello > Manifest
	jar cvfm myHello Manifest hello.class
	rm Manifest
	chmod +x myHello

hello.class: hello.java
	javac -Xlint hello.java

clean:
	rm -f hello.class myHello

submit: README makefile hello.java
	submit cmps012b-pt.u14 lab1 README makefile hello.java
```

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

The make utility has a macro facility that enables 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 and `list` is a list of filenames. Then `${list}` can then be used to refer 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 simple makefile with macros
JAVASRC    = hello.java
SOURCES    = README makefile ${JAVASRC}
MAINCLASS  = hello
CLASSES    = hello.class
JARFILE    = myHello
JARCLASSES = ${CLASSES}
SUBMIT     = submit cmps012b-pt.u14 lab1

all: ${JARFILE}

${JARFILE}: ${CLASSES}
	echo Main-class: ${MAINCLASS} > Manifest
	jar cvfm ${JARFILE} Manifest ${JARCLASSES}
	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 which take place before **make** interprets the file. Study this new makefile until you understand exactly what substitutions are taking place. Now create your own Hello World program and call it `hello2.java`. It can say anything you like, but just have it say something different than the original. Add `hello2.java` to the `JAVASRC` list, and add `hello2.class` to the `CLASSES` list. Also change `MAINCLASS` to `hello2`.

```
# Another makefile with macros
JAVASRC    = hello.java hello2.java
SOURCES    = README makefile ${JAVASRC}
MAINCLASS  = hello2
CLASSES    = hello.class hello2.class
JARFILE    = myHello
JARCLASSES = ${CLASSES}
SUBMIT     = submit cmps012b-pt.u14 lab1

all: ${JARFILE}

${JARFILE}: ${CLASSES}
      echo Main-class: ${MAINCLASS} > Manifest
      jar cvfm ${JARFILE} Manifest ${JARCLASSES}
      rm Manifest
      chmod +x ${JARFILE}

${CLASSES}: ${JAVASRC}
      javac -Xlint ${JAVASRC}

clean:
      rm ${CLASSES} ${JARFILE}

submit: ${SOURCES}
      ${SUBMIT} ${SOURCES}
```

This new makefile now compiles both Hello World classes (even though neither one depends on the other.) However the program execution entry point has been changed to be the main method in your `hello2.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 webpage.

**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, date, and a short description of its role in the project, followed by 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 myself or the grader.

See the webpage for instructions on using the submit command. Submit the following files to the assignment name lab1:

```
README
makefile
hello.java
hello2.java
```

You can either type the submit command directly:

```
% submit cmps012b-pt.u14 lab1 README makefile hello.java hello2.java
```

or use the makefile itself:

```
% make submit
```

Your makefile can be with or without macros, although the macro version is recommended. In any case your makefile should make both class files (`hello.class` and `hello2.class`), and an executable jar file called `myHello` which runs your `hello2.class`. Your makefile should also include `clean` and `submit` utilities, as in the above examples.