CMPS 12M

Introduction to Data Structures

Lab Assignment 1

Objective

The purpose of this assignment is to introduce you to your Unix ITS account, learn how to use Java from the command line, and demonstrate how to use a Makefile to automate tasks.

Setup a place to work

First you'll set up a secure directory to do your work. Login to your ITS Unix timeshare account at `unix.ic.ucsc.edu`. From your home directory, type the following commands (everything after the $ prompt):

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

Your ITS account uses AFS which has its own access control system. The fs command lets you tell the filesystem which actions to perform. The setacl sets the access control list (ACL) for a specific location, while listacl displays the access control list for a location. The above commands create a new directory then set the permissions so that you have permission to read (r), list (l), insert (i), delete (d), write (w), lock (k), and administer (a) the private directory. Other users do not have any permissions, so cannot see the files inside the directory or read their contents.

The fs setacl command works like:

```
$ fs setacl <some directory> <some username> <some subset of rlidwka or none>
```

The group system:authuser refers to anyone with an account on the system, and system:anyuser refers to anyone in the world running AFS.

If you ever have trouble with commands, you can always bring up a manual page using:

```
$ man mkdir
```

You can also bring up a list of valid fs commands with:

```
$ fs help
```

Organize directories

To keep your work organized, make some subdirectories for assignments and labs.

```
$ cd private
$ mkdir cmps012m
$ mkdir cmps012b
$ cd cmps012m
$ mkdir lab1
$ cd lab1
```

You don't need to manually set the ACL for these new subdirectories, they are inherited from the parent directory that you already set.

Hello Java

Use any editor to create a short "Hello world" Java program saved as `hello.java` with the following text:

```
// 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);
    }
}
```

Once you have the file saved, try compiling it and running it using the Java command line tools:

```
$ javac hello.java
$ java hello
```

The first command compiles the source code into a hello.class file. The second command uses the Java runtime to load and execute the class file.

Hello executable

Next create a standalone executable program. You'll do this by using the jar tool. This tool manages collections of class files. It can merge many class files into one jar file, and is also capable of creating executable files that can be run just by typing their name at the command line.

First create a manifest file. This file will specify the entry point for program execution (e.g. which class file to run). Create a file called Manifest with the following text:

```
Main-class: hello
```

You can use an editor or use the command line:

```
$ echo "Main-class: hello" > Manifest
```

This command line uses the echo command which just prints text to stdout, then redirects it into the Manifest file.

Now that you have a manifest, you can create the executable file:

```
$ jar cvfm myHello Manifest hello.class
```

The cvfm are options to the jar command, they say to create (c), with verbose output (v), files (f), with a manifest (m). The arguments are the name of the archive to create, the manifest file, then a list of class files to include (in our case just the one).

Finally we need to mark the output file as executable to be able to run it:

```
$ chmod +x myHello
$ ./myHello
```

The chmod command controls the executable bit of the file.

Java lint

It's good practice to compile Java programs with warnings turned on. Rebuild your executable, this time compiling with all recommended warnings turned on during the compile:

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

In this sequence we create the manifest file then delete it as soon as we don't need it any more.

Automating the build

Typing all those commands every time you change a line of code is ridiculous. Let the computer do the work by setting up a makefile and using gmake.

Software projects are made up of many files that all have to work together to build the final finished product. The pieces typically depend on each other in complicated ways. The simplest solution is to rebuild everything whenever anything changes. This works for very small projects but quickly becomes a huge productivity drain on larger projects.

The better strategy is to see what has changed, then compute the steps needed to only rebuild what is needed. The make utility does this. You write a makefile that specifies the dependencies of each part of the build process, then make automatically figures out the commands to run to rebuild after you make changes.

Makefile rules look like:

```
target: prerequisites
	build-commands
```

Each rule says that the target depends on the prerequisites. If the prerequisites change, then the build commands are executed to rebuild the target. Makefiles are normally put in files named Makefile.

Here is an example makefile for our Java program:

```
# My first 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
.PHONY: clean
```

Note that the indented lines *must* use a tab, not spaces. The second rule tells make how to create hello.class from hello.java using the javac command. The first rule tells make how to turn hello.class into myHello, the final executable command. The third rule is a dummy target. It lets you do:

```
$ make clean
```

The special notation .PHONY tells make that clean is not a file, it is a fake target.

Once you have this makefile you can rebuild your executable by doing:

```
$ make myHello
```

If hello.java has been modified, make will see the dependency of myHello on hello.class, which depends on hello.java. It will compile hello.java into hello.class, then run the steps to produce the executable. If you run the make command again, it will detect that everything is up to date and do nothing. To save even more typing, running make with no target defaults to the first target in the makefile.

Try deleting java.class and myHello in different combinations and running make to see how it works.

Make variables

Right now the makefile is very specific to the hello program. Make allows makefiles to set variables and refer to them in rules. This means we can abstract out the details of filenames and class names into one section of variables at the start of the makefile.

Setting a variable is done with `ID = list`, where ID is the name of the variable and list is a list of filenames or options. To refer to a variable use the syntax `${ID}`.

Here is a reorganized version of the makefile that uses variables.

```
# A simple makefile with variables
JAVASRC = hello.java
SOURCES = README makefile ${JAVASRC}
MAINCLASS = hello
CLASSES = hello.class
JARFILE = myHello
JARCLASSES = ${CLASSES}

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}

.PHONY: clean all
```

Update your Makefile to this version and verify that it still works the same.

Your hello world

Now create your own Java program that prints out a message to stdout. Your message must be different than the one included here. Print out a haiku or (clean) limerick about computer science, or any other short witty quote. Save your file as hello2.java.

Adapt the makefile to compile both versions of hello and put them in the final executable, but set the executable entry point to your modified program. You'll want to add your source file to JAVASRC, your class file to CLASSES, and update MAINCLASS.

What to turn in

All files you turn in for every assignment and lab should begin with a comment block that includes your name, CruzID, class, date, filename, short description of the file's role in the assignment, and any special instructions related to the file. Also create a file called README. The README file should have the normal comment block, then list all the files being submitted (including itself) along with any special notes to the graders.

For this lab, submit the following files:

```
README
Makefile
hello.java
hello2.java
```

To submit, use the submit command.

```
submit cmps12b-nojw.f14 lab1 README Makefile hello.java hello2.java
```

Your makefile should use variables and include a clean target. It must compile both source files and create an executable named myHello that runs your hello2.class file.