CSE 390, Fall 2010
Homework 8: Build Tools (make and ant)
Due Tuesday, Dec 7, 2010, 12:30 PM

This assignment focuses on using automated build tools such as `make` and `ant`. Turn in files named `homework8.txt` and `Makefile` using the Homework section of the course web site, along with the (optional) build.xml if you complete task 3.

Task 1 of 3: Build and Run a Program

Self-Discovery: SourceForge is a useful web site that allows developers to host their open source projects for free. SourceForge hosts each project's code and makes it available for any to download free of charge.

For this task you will download, compile, and run a piece of open source software from the web using `make`. The program is called **`jp2a`** (JPEG to ASCII converter). Here are the steps to follow:

1. Find the `jp2a` home page using your favorite search engine, and **download** the program's source code as a `.tar.gz` file from SourceForge. (The download page will have other files available for download, such as a Windows binary version. You don't want those files. Click "View all files" and look for the `.tar.gz` file.)
2. Once you've downloaded the file, **decompress** it. (See the lecture 6 slides for how to decompress a `.tar.gz`.)
3. After decompressing the archive, you must **run the `configure` program** to set up the build process for your particular computer type/architecture. The `configure` program analyzes your system and produces a `Makefile` that will work for your computer to build and install the program.

   (*Note:* The `jp2a` compilation process requires that your computer have the `gcc` and `make` programs and an installed library called `libjpeg`. Our shared `attu` server already has these, but your own Linux box might not. If you plan to work from your home machine, see the "Installation Notes" section on the next page.)

Normally `configure` sets up the `Makefile` to install the app to the directory `/usr/local/bin`, but since you may not have root access on the system you're using, you may not be able to install it there. We suggest you tell `configure` to install the app to your current directory, the directory where you decompressed the `jp2a` source files. To do so, run the command as follows (the ` marks are back-ticks):

```
./configure --prefix=`pwd`
```

If it worked properly, you'll see several lines of output such as:

```
checking for GNU libc compatible malloc... yes
   config.status: executing depfiles commands
```

4. Assuming that the `configure` program completes successfully, you are ready to **compile and install** the program using `make`. Run the `make` command, and once it completes, run `make install` . If each command works, it will output several mostly incomprehensible messages such as:

```
gcc  -g -O2 -o jp2a  html.o term.o curl.o jp2a.o image.o  -ljpeg -lcurl -lncurses
   make[2]: Leaving directory `/homes/iws/stepp/390/hw8/jp2a-1.0.6/src'
```

5. If `make install` worked properly, you should now have a `bin/` subdirectory within your `jp2a` source folder. In this folder should be a newly built executable called `jp2a`. You can **run the program** from that directory by typing:

```
./jp2a [options] filename.jpg
```

6. To complete this task you should download a `.jpg` file of your choice from the web and convert it to ASCII using `jp2a`. Use Google Image Search to find an image. You can get the file onto `attu` by using `wget` *`URL`* if needed.

The `jp2a` program should output an ASCII version of the image. Redirect the `jp2a` ASCII output to **capture the output in a file** named `homework8.txt` and submit this file as part of your assignment turnin.

(*Side note*: `jp2a` has several **options** that you can learn by typing `jp2a --help`. There isn't a `jp2a man` page because we haven't fully installed the app or its `man` files into your system. A particularly useful option is

`--background=light`, which causes white/light backgrounds to be drawn in a lighter color. We found that this option made the ASCII output of some of our test images, such as a Homer Simpson drawing, look much better.)

Task 2 of 3: Write a Makefile

For this task you will **write a `Makefile`** for a small set of C program files provided by the instructors. Download the resource file `hw8-2.tar.gz` from the course web site and decompress it to your homework directory. These files represent a linked list library stored in `linkedlist.c` and `linkedlist.h` along with some client program C files that use this library to perform simple tasks.

Your `Makefile` should have the following **six properties**:

- A target that **builds an object file named `linkedlist.o`** from the source code found in `linkedlist.c`. If `linkedlist.c` or `linkedlist.h` is modified, the `linkedlist.o` file should be rebuilt. In other words, it depends on both of those files. (You can test this by `touch`ing the `.c` or `.h` file and then re-running `make`.)
- A target that **builds an executable file named `ll`** from the source file `use_linkedlist.c` and the compiled object file `linkedlist.o`. If `linkedlist.o` or any of its dependencies are modified, `ll` should be rebuilt.
- A target that **builds an executable file named `ll2`** from the source file `use_list_2.c` and the compiled object file `linkedlist.o`. If `linkedlist.o` or any of its dependencies are modified, `ll2` should be rebuilt. (You can test the `ll` and `ll2` programs by running them once they have been compiled.)
- A **target named `clean`** that removes the `ll` and `ll2` executables along with any `.o` files from the directory.
- The `Makefile`'s **default target** should build both the `ll` and `ll2` executables.
- Use at least **one of `make`'s advanced features**. For example, declare at least one variable and use it in your rules, and/or try to use some of the special variables such as `$^` or `$<` .

For reference, our `Makefile` is 17 lines long (10 non-blank, non-comment "substantive" lines).

Installation Notes for Tasks 1 and 2:

The `attu` server and CSE basement machines are already set up to allow you to compile C programs and use `make`. If you plan to work on this phase on your own Linux box, you may need to install the `gcc` compiler and `make` system. In Ubuntu you can do this by typing the following command in a shell:

```
sudo apt-get install gcc make libjpeg62-dev
```

In Fedora, install by clicking System->Administration->Add/Remove Software . Search for "libjpeg" and install the "Development tools for programs which will use the libjpeg library" package. Then search for "gcc" and install the "Various Compilers" package . If prompted for "other packages have to be installed...", say yes.

(Optional) Task 3 of 3: Write an Ant `build.xml` file

For this task you will **write a `build.xml` file** that can be used with the `ant` tool to compile and run a large Java application. This is optional and may require a bit of exploring ant on your own, but may be useful for those who often use Java or want to learn Ant. Download the resource file `hw8-3-pacman.tar.gz` from the course web site and decompress it to your homework directory. These files represent a Pac-Man game written in Java by Marty Stepp.

Self-Discovery: The Java compiler and runtime system use an internal variable called the "**class path**" to determine the relative working directory from which Java classes should be loaded. If your code refers to the `Foo` class, the Java virtual machine needs to go find the `Foo.class` file and load it into its memory. The default class path is the current directory, the same directory as your `.java` files. In many small Java programs, this is the best setting. But larger programs divide their classes into several packages spread across many folders; in such cases, the developer may need to explicitly set the class path when running the compiler or JVM.

The Pac-Man code in this project is large and uses several packages, so we'll need to deal a bit with class path. For example, the class `Level` is located in the `pacman.model` package, so the JVM will look for a file named ***`CLASSPATH`***`/pacman/model/Level.class` .

Your `build.xml` file should have the following **three properties**:

- A **target named `compile`** that builds all of the compiled `.class` files for the project. The source code files are located in directory `src/` , and you should use this in your target's actions. The compiled `.class` files should be placed into the directory `bin/` . Your target actions should create the `bin/` directory if it does not already exist.

(When setting up the `javac` compilation action for this target, you will need to mention the `src/` directory in two places: as its `srcdir` and as its `classpath`.)

If you complete this target successfully, it will compile 57 source files. It will create new subdirectories and files such as `bin/pacman/model/Demo.class` relative to the location of your project files and `build.xml` .

- A **target named `clean`** that removes the `bin/` output folder and all its files.
- A **target named `run`** that runs the compiled program. The class to run is named `pacman.PacManMain` .

In order to run the code, you will need to tell your `java` action to use your `bin/` directory as its Java class path. Do this using the following syntax:

```
<java ... >
    <classpath>
        <pathelement location="DIRECTORYNAME" />
    </classpath>
</java>
```

You could run the Pac-Man game on `attu`, but it will be unable to display its graphics unless you enable X11 forwarding on your `ssh` shell as described in the Lecture 6 slides (and even then, it'd be slow). It would run better on a CSE basement Linux box or your own Linux machine.

Use **relative paths**, not absolute paths, in your `build.xml` file. Don't write a full path such as `/home/username/...` .

For reference, our `build.xml` is 20 lines long (18 non-blank, non-comment "substantive" lines).

If you need syntax help, see the `make` and `ant` examples in links on the web site and in the lecture notes. See the `ant` tasks reference on the course Homework page. You will need the tasks `delete`, `java`, `javac`, and `mkdir` .

The `attu` server and CSE basement machines are already set up to allow you to use `ant`. If you plan to work on your own Linux box, you may need to install `ant`. In Ubuntu you can do this by typing the following command in a shell:

```
sudo apt-get install ant
```

In Fedora you can install `ant` by clicking System->Administration->Add/Remove Software . Search for "ant" and install the "Ant build tool for Java" package . If prompted for "other packages have to be installed...", go ahead and install them.