Lab #3
Automating Installation & Introduction to Make
Due in Lab, September 17, 2003

Name: ____

Lab Time: ____

Grade: __________/10

Error Checking

In this lab you will be writing a shell script to automate the installation of a system. The system that your shell script will install is the gnuchess system that you installed by hand in lab 1. Since commands are not being typed interactively by a user, a shell script needs to avoid making assumptions. Error messages generated during the execution of a shell script are likely to go unnoticed as the output scrolls quickly across the screen. The programmer of a shell script, therefore, should be aware of, and try to prevent, possible error conditions that might occur when the shell script executes.

One possible source of errors is the directory structure. A shell script should avoid using non-existent directories. As such, it needs to check for the existence of any directory it intends to use, and be able to create missing directories as needed. Of course, there is a limit to how much checking a shell script can or should do. For example, shell scripts rarely need to check for the existence of a user's home directory.

The amount of error checking required is, therefore, a design decision that must be made by the script's programmer. Before making this decision, the programmer should consider the script's users. How likely are they to encounter a particular error, and will they be able to deal with it, if it is not handled by the shell script? In addition, a programmer may have external requirements imposed by a manager, or customer, specifying the type of errors that must be handled.

Your shell script should not make assumptions about the existence of architecture-specific subdirectories (such as `.../arch/x86/build/...`, etc.) when building and installing the gnuchess program. If any required directory does not exist, your shell script needs to create it. This allows your shell script to proceed with confidence in the automated installation.

Automatically Setting up Your Architecture Specific Directories

1. To set up your architecture specific directories create a shell script called `archdir-setup` and put it in your `~/csci3308/bin` directory (recall, this is your architecture-independent `bin` directory).

2. This shell script should be written so that it can be run from any directory. You should either use absolute paths, or change directory so that you know where you are before using relative paths.

3. In this shell script you can assume that the directory `~/csci3308/arch` exists. Add the following commands to your shell script to create the architecture specific subdirectories:

   ```
   cd ~/csci3308/arch/$ARCH
   mkdir bin build include lib man tmp
   ```

4. Make this script executable. Then type `rehash`. This will reset the hashtable the shell uses to optimize finding commands. Now use `which` or `where` to make sure that `archdir-setup` is in your path.

5. Now execute the script on an `x86` machine. You should see a few error messages (up to six) as the script tries to create directories that already exist. If you do not see any error messages, then run the script again (and also check that your $ARCH variable is defined). These error messages are not very serious, given our purpose for this script. We are merely double checking that these directories exist. If they already exist that's fine, but the user should not have to see this type of error message.

6. To avoid these error messages, rewrite your script so that it first checks if a directory exists, and only calls `mkdir` if the directory doesn't exist. To do this you will need to use the `if` statement, and file inquiry operators. You can learn more about file inquiry operators on page 639 of the "Linux Shells by Example" textbook.

7. Log in to a machine of a different architecture (using the `rlogin` command) and check to see if your script runs correctly. The machines in the Whitewater lab are Linux machines with an architecture of "x86." In 1B54 there are Sun Ultras that have an architecture of "sun4," so you should be able to remotely access one of these machines and run your script there. The names of the Sun Ultra machines are:

   ```
   bfs, csel2, ahau, cauac, etznab, cib, men, ix, ben, eb,
   muluc, caban, oc, manik, lamat, cimi, chicchan, ik, akbal,
   chuen, kan, imix
   ```

   When you run your script, it may fail with an error message like this:

   ```
   .../csci3308/arch/sun4:  No such file or directory
   ```

   Since we have been using `x86` machines for the entire semester, we have yet to create an architecture-specific subdirectory for the `sun4` architecture. Our script should be able to set up our architecture-specific directories on whatever machine you are currently on. Therefore, modify the `archdir-setup` script to create the architecture-specific subdirectory, if needed.

8. Run your script again on this new machine to make sure that it works correctly. (e.g. it should create the `.../csci3308/arch/sun4` directory and then create the `bin`, `build`, and other directories beneath it). Run the script a second time to make sure it doesn't produce error messages when the directories already exist.

9. Print out your `archdir-setup` script and turn it in with the lab. Be sure to logout of the `sun4` machine (using the `exit` command) and perform the rest of this lab on a machine in the Whitewater lab.

Writing the Installation Shell Script

10. You will now write the shell script that installs gnuchess automatically. This script should be called `gnuchess-install`, and it should be in your *architecture independent* `bin` directory. In order to work, your shell script will require that the gnuchess distribution has already been downloaded and unpacked into your `src` directory.

11. The first thing your shell script should do is to run your `archdir-setup` script to be sure that all of the directories it needs exist.

12. Before building gnuchess your shell script needs to make a new subdirectory of the architecture-specific `build` directory. Add the command(s) to your shell script to create this directory, but be sure not to print an error message if this directory already exists.

13. The following steps need to be taken to install gnuchess. For each step add the necessary command(s) to your `gnuchess-install` shell script. Look back at lab 1 to help you remember the exact commands.

    (a) Localizing (configure)

    (b) Building (make)

    (c) Installing (make install)

14. Now finish up your shell script by making it executable, and making sure of the following things. The shell script should be in your path, and it should work regardless of what directory you are in when you run the script. I.e. you cannot assume that you start in the build directory.

15. Test your script starting with a "clean" environment. Change to the directory `~/csci3308/arch/$ARCH` and remove all of its subdirectories. Run your `gnuchess-install` program and check to see that it automatically creates the needed directories, builds gnuchess, and installs it in the correct directory.

16. Print out your `gnuchess-install` script and turn it in with the lab.

Introduction to Make

17. Type `where make`. For this and later labs you must use the GNU version of make located in `/tools/cs/gnu_util/bin/make`. If this version is not first in your path you need to change the order of directories in your path so that it is first, or create an alias to the full directory path to this version.

18. Change to your `~/csci3308/tmp` directory. You will be creating some temporary files to learn about the "make" command.

19. Copy the file `~csci3308/src/lab03.makefile` to your `tmp` directory. The contents of this makefile are listed below.

```
file1:  file2
        echo file1 not up to date

file3:  file4
        echo file3 not up to date
        touch file3
```

20. The command `touch` is used to create a blank file, or to change the timestamp of an existing file to the current time. Type the command `touch file2`.

21. Now type `make -f lab03.makefile`. The `-f` option tells the make program which file to use as the makefile. Which of the rules in the makefile were run, what happened, and why?

22. Type `make -f lab03.makefile` again. You already ran make once, why isn't `file1` up to date?

23. Modify the first rule of the makefile by adding a command to touch file1. It should look like this:

```
file1:  file2
        echo file1 not up to date
        touch file1
```

24. Type `make -f lab03.makefile` again. What files are in your directory?

25. Type `make -f lab03.makefile` again. What happened differently and why?

26. Now type `make file3 -f lab03.makefile`. This tells make to use the target `file3` as the goal. What error was produced, and what could you do to eliminate this error?

27. Modify the makefile to add `file3` as a dependency of `file1`. The first rule should look like this:

```
file1:  file2 file3
            echo file1 not up to date
            touch file1
```

28. Now type `touch file4`, and `make -f lab03.makefile`. In what order were the rules run, and why?