Lab #0
Getting Started
Due In Your Lab, August 27, 2003

Name: ______________________________

Lab Time: ______________________________

Grade: __________/10

Man

UNIX systems contain on-line manuals called man pages that are accessed through the program man. To find out how to use man type `man man`. Those of you already familiar with man pages may still want to re-read the man man page to learn about advanced features that you may not already know. Regardless, you must know at least enough about man to answer the following questions.

1. What command would find the man page for `exit` in section 2 of the manual?

2. What command would list all the man pages related to the key word `network`?

Emacs

The text editor emacs has several features that we will use in this course such as integrating with the compiler and debugger. Therefore, you must learn and use emacs for this course. The more you use emacs early on the easier it will be to use when you are required to use emacs for its extra features. You can find a pointer to the emacs reference card in the reference materials section of the class web site.

To find out how to use emacs run `emacs`, and type `CTRL-h t`, or in emacs parlance `C-h t`. This will start the emacs tutorial. The complete emacs manual can be found online at

`http://www.gnu.org/manual/emacs-20.3/emacs.html`

Once again, those of you already familiar with emacs may want to look at the tutorial for features that you don't yet know. You must at least be able to answer the following questions.

3. What are the commands to move forward and backward a word at a time?

4. How do you give a numeric argument to emacs commands?

5. How do you undo?

The Kernel and the Shell

It is important for you to understand how the UNIX operating system is organized. When you are logged on, the computer is running something called a *kernel* and something else called a *shell*. These are separate things, and we will discuss them one at a time.

The kernel controls all access to hardware. Whenever a key is pressed, the mouse is clicked, or something is printed to the screen the kernel must be involved. However, the kernel is a very low level component. Users don't interact with it directly. It does not print your prompt, and it does not interpret the commands you type. You can interact with the kernel when you write programs, for example when you open and close files.

You can call the kernel directly with *system calls*, but normally you don't do this. Instead, you call a *library procedure* which calls the kernel for you. Look up the man page for `open` in section 2 of the manual, and the man page for `fopen` in section 3 of the manual. Both of these open files. `Open` is a system call directly to the kernel, and `fopen` is a library procedure. Many programs use `fopen` because it handles details automatically and presents a simpler interface. You can think of calling the kernel directly as low level programming like assembly language, and library procedures as high level programming like C++.

Strictly speaking, the UNIX operating system consists of just the kernel. However, there is another program called the shell that is very important. The

shell is the program that prints your prompt, and starts up programs when you type their names. The shell is a normal program like Netscape or Emacs. It has to use the kernel to accomplish what it does. For example, here's what happens when you type `ls *.cxx` to print all `.cxx` files:

a. Each time you type a key, the kernel receives a hardware interrupt stating that a key has been pressed. It places that character in an input buffer for the shell. The kernel has no idea that the two characters `ls` are the name of a program that you want to run.

b. The shell reads its input buffer until it gets to the end of the line. (You read from the input buffer in your C++ programs when you use the `cin` command.) The shell interprets the line as words separated by spaces. The shell assumes that the first word, `ls`, is a program that you want to run. The shell has to ask the kernel if a program named `ls` exists, and if so, to please start it up.

c. The shell also interprets `*.cxx`. The shell asks the kernel for the names of all the files in the current directory. The shell knows that `*` matches anything, so it checks each one to see if it ends with `.cxx`. If so, it passes the file name on to `ls`.

d. The `ls` program receives a list of file names. `ls` has no idea that you ever typed `*`. It prints out each of the files and exits.

e. The shell prints another prompt, and waits for the user.

It is important to know what parts of the computer do what in case something goes wrong. At some point, you may think you have a bug in your program when in fact the problem is that the shell can't find your program, or there is a problem with interpreting your arguments.

One final point is that you can run many shells, but there is only one kernel. If you have two windows open, and each window has a prompt, then you have two shells running, but they are both talking to the same kernel. Several people can be remotely logged on to a single computer. Each of them could have several shells, but they would all access the same kernel.

There are also several types of shell. To find out what shell you are using type `echo $shell`.

6. What shell are you using?

Shell Variables and Environment Variables

To make your life easier, the shell has the ability to store variables. Each variable holds a value just like variables in an ordinary programming language. Unlike most programming languages all variables are of type string. It is legal to say `set x=25`, but this sets the value of variable `x` to be the string `"25"`, not the integer `25`. To read the value of a variable precede its name with a dollar sign. For example, `$x`. When the shell sees a dollar sign, it looks this word up to see if it is the name of a variable. If the word is a variable name, the shell replaces the word, including the dollar sign, with the value of the variable. If the word is not a variable name, the shell prints an error message. Type the following:

```
set word = quiet
echo word
echo $word
```

Echo is a command that prints out whatever it reads in. When you typed `echo word` the echo command read `word` and printed `word`. When you typed `echo $word` the *shell* saw `$word` and replaced it with `quiet`. Then the echo command read `quiet` and printed `quiet`. The effect was exactly as if you had typed `echo quiet`. Now type:

```
echo $wordly
echo "$word"ly
```

In the first command the shell couldn't find the variable `wordly`. The problem was that the shell tries to get the biggest word it can, stopping only for spaces or other special characters, so it didn't try to match `$word`. Quotes are one of the special characters that the shell recognizes so the second command tells the shell to look up `$word`, and then add `ly` to the end producing `quietly`.

Variables can also be set to lists of strings. A list is actually a single string where a special character separates list items. The shell stores this value as a normal string, but certain programs treat it like a list of items.

There are two types of variables, shell variables, and environment variables. There are only two major differences that you need to be aware of between these types of variables. The first difference is that environment variables are copied to any new child processes created from the current one while shell variables are not. For this reason environment variables are often used like global variables and shell variables are used like local variables. To distinguish them, environment variables are typically given names in all capital letters, while shell variables are given lower case names.

The other difference between shell variables and environment variables is syntax. A shell variable is defined with the set command. There is an equal sign between the variable and its value. If a value is a list the items are separated by spaces and enclosed in parentheses. For example,

```
set shopping_list = (apples oranges bananas)
```

An environment variable is set with the setenv command. There is no equal sign. If a value is a list the items are separated by colons and enclosed in quotes. For example,

```
setenv SHOPPING_LIST "apples:oranges:bananas"
```

Finally, if a lab refers to a "variable", it typically is talking about a shell variable. If a lab refers to an environment variable it will say "environment variable."

To see a list of the local variables for the current shell, type `set`. To see a list of the environment variables for the current shell, type `printenv`.

Type the following statements.

```
set a = the
set b = blue
setenv C car
set b = ($C $b)
setenv PHRASE "$a : $b : $C"
echo $b
echo $PHRASE
```

7. List all of the variables given above, and say whether each is a shell variable or environment variable.

8. What are the values of `$b` and `$PHRASE` after executing the above statements?

In the same window type `xterm`. This will open a new window which will be a child process of the window in which you typed `xterm`. In the child window type the following.

```
echo $b
echo $PHRASE
```

9. What happened any why?

Directory Paths

In UNIX the location of a file or directory is given like this

```
/usr/local/X11/doc/FAQ.X
```

This is essentially a list of directory names separated by slashes, and the last name is the name of the file. You can think of the file system as a tree. Each directory is a node in the tree. It has pointers to its parent, and all of its children, but does not know anything about other nodes in the tree. To find this file, `FAQ.X`, the kernel cannot just go directly to its immediate parent directory, `doc`, because it does not know the location of that directory on the hard disk. The kernel only knows the location of one directory, `/`, also called the root directory.

To find `/usr/local/X11/doc/FAQ.X` the kernel goes to `/`, and looks up the location of `usr`. Then it goes to `usr` and looks up the location of `local`, and so on. This hopping from one directory to another has been described as following a *path* of directories. For this reason, `/usr/local/X11/doc/` is said to be the *directory path* to the directory `doc` that contains the file `FAQ.X`.

This leads us to a very confusing naming convention in UNIX. There is an important variable named `path`. This variable actually contains a *list* of directory paths. I have probably spent an inordinate amount of space describing this, but in past semesters in this class there was sometimes confusion when someone used the word path. This usually occurred when talking about a directory path listed in the path variable. I just want everyone to be aware of this potential confusion.

When I said that the kernel only knows the location of the root directory that was not entirely accurate. It also stores a few other convenient directories where you can start directory paths. The first of these is the current working directory. When a directory path does not start with a slash it is assumed to start in the current directory. Directory paths that start in the current directory are called relative paths because the location referred to is relative to where you are at the time, and can change if you change directories. In contrast, directory paths that start at the root directory are called absolute paths because the location referred to does not change if you change directories.

There is a third starting point for directory paths called your home directory. The character `~` represents the current user's home directory. To specify a different user's home directory type `~`*`username`*. You can also specify your home directory with the environment variable `HOME`. Type `echo $HOME` to see what its value is. Using `~` does not work in every situation. If you run into problems try using `$HOME` instead.

The Path Variable

When you type a command you are actually typing the filename of an executable file. To run that command the shell must be able to find the file. The way to tell the shell where to look for executable files is through the path variable. Type the following:

```
echo $path
```

You should see a list of directories. Most of them probably end with `bin`. Bin stands for binary and is a common name for directories used to hold executable files. Now type:

```
set path = ()
echo $path
ls
```

Oh, no! UNIX is broken, you can't even list the files in your directory! Actually, UNIX isn't broken, you just have to understand how it works. There is nothing special about `ls`, it's just a program. It has an executable file somewhere in the file system. When you want to list the files in your directory you type `ls`. The shell sees this, and looks for an executable file named `ls`. Where does it look? It looks in the directories listed in your `path` variable. When you executed `set path = ()` you set your path to an empty list. The `ls` program is still there, but your shell doesn't know where to find it. Luckily there is a way to get your path back. Type:

```
source ~/.cshrc
ls
```

You should recognize `~/.cshrc` as a directory path that starts in your home directory. When you see this you should think, "There is a file named `.cshrc` in my home directory." This file is called a startup file, and it is where the shell gets your path every time you log on. Essentially, you just re-ran your startup file to reset all of your variables including your path.

Changing Your Path

Lets say you have a new program that you want to run, but it is in a directory that is not in your path. You can change your path to add this directory. Type the following:

```
hello
set path = (~csci3308/arch/$ARCH/bin)
hello
```

The executable file `hello` is in the directory `~csci3308/arch/$ARCH/bin`. The shell couldn't find it until you put the directory in your path. Of course now you have deleted all the other directories from your path. If you type `ls` it won't work. Type `source ~/.cshrc` again to get your path back. Now type `hello`. It doesn't work because you re-initialized your path which doesn't contain `~csci3308/arch/$ARCH/bin`. What you really need is some way to add a directory to your path without destroying the rest of your path. Type the following:

```
set path = ($path ~csci3308/arch/$ARCH/bin)
hello
ls
```

Problem solved! Saying `set path = ($path ~csci3308/arch/$ARCH/bin)` is much like saying `x = x+1`. It keeps what is in the path, and adds another directory.

There is another problem which you haven't encountered yet. From that same window type `xterm` to get another window. Now try to run `hello` from this new window. It shouldn't be able to find it in your path.

10. Why can't the shell find the `hello` command in this new window?

This is where the `.cshrc` file comes in. Remember, you can re-set your path by re-running `.cshrc`. Every time you log in or create a new window your path gets set from this file. If you want to make permanent changes to your path you can change how it gets set in this file. Run `emacs ~/.cshrc`. this file is just an ordinary text file, and you can edit it with emacs. Find where it defines the path variable. If you have the default `.cshrc` given to new accounts it should be on or near line 31.

Notice that path is set here with the set command exactly how you set it on the command line. Add `~csci3308/arch/$ARCH/bin` to this list, and save the file. Now type `source ~/.cshrc` to reload your path from this file. The shell also uses a shortcut hash table to speed up processing your path. If your path ever seems like it isn't working correctly try typing `rehash` to refresh this table. Now try running `ls` and `hello` to make sure your path is correct.

Which and Where

The next thing to understand is how the shell searches your path. The shell starts with the first directory in your path. If that directory contains a file with the correct name, it tries to run it. If not, it goes on to the next directory in your path, and so on. There is a different version of hello in the directory `~csci3308/bin`. Add this directory to your path. Now type the following:

```
~csci3308/bin/hello
~csci3308/arch/$ARCH/bin/hello
hello
```

First, you typed out the entire path to the files and the shell should have run the copy of hello that you specified. When you just typed hello the shell should have run the copy in whichever directory comes first in your path. There are two useful commands for looking for programs in your path, `which` and `where`. Type the following:

```
which hello
where hello
```

`which` prints out which program will be run if you just typed its name with no directory path. `where` lists all copies of a program with that name in your path. If there are multiple copies of a program with the same name, and you care which one gets run, you may have to use `which` or `where` and the program's full directory path.

Your Directory Structure

It is a good idea to have an extensive directory hierarchy under your home directory to organize your files. There are no absolute rules about the best way to organize a directory hierarchy. Usually, you design your own hierarchy, and the most important thing is how useful it is to you. However, sometimes there are other considerations. For example, if you are working for a company, your manager might want everyone's directory hierarchy to be set up the same so that things are easy to find even under another person's home directory. In this class we will be simulating this situation where your directory hierarchy is specified for you by a manager.

In your home directory create a directory named `csci3308`. This directory `~/csci3308` will be called your class home directory. All files we create in this class will be placed under this directory, and this directory should only be used for this class. The reason for this is so that our files will not interfere with your own personal files, or with files from other classes. Also, for many of the labs we have set up shell scripts and make files to work under this directory.

There is an important difference between `~csci3308` and `~/csci3308`. There is a computer account for the class called csci3308. This is essentially a separate user named csci3308. This user's home directory is `~csci3308`. Many of the files you will use for this class will come from this account. However, you also have a directory named `csci3308` under your home directory. The directory path to this is `~/csci3308`. This means `~`, your home directory, `/`, a subdirectory of your home directory, and `csci3308`, the subdirectory is named csci3308. This is different than `~csci3308` which is the home directory of a different user who happens to be named csci3308. You should understand this difference and keep these two directories straight in your head.

Create subdirectories of your class home directory as shown in Figure 1. The purpose of each directory is explained below.

Note: Some printers cannot print the figures below correctly. If the figures below look weird on your printout, then view the figures on-line using the Adobe Acrobat Reader.

`arch` There are computers of several different architectures in the education lab. To find out the architecture of the machine you are on type one of the following commands:

```
/usr/local/bin/arch
echo $ARCH
```

You should be able to recognize `$ARCH` as being the value of the environment variable `ARCH`. You can tell that it is an environment variable because

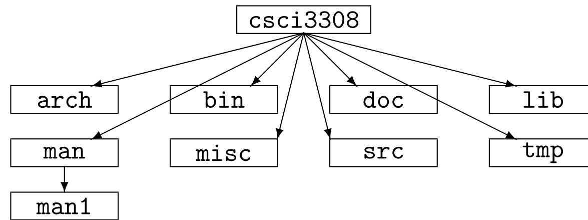

Figure 1: Directory Structure

it is capitalized.

Certain files such as binary executable files will only work for a particular architecture. These are referred to as architecture specific files. The arch directory is where you will keep any architecture specific files. Create a subdirectory of your `arch` directory for each architecture in the lab as shown in Figure 2.

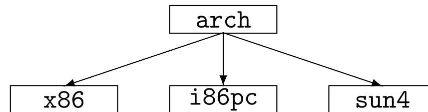

Figure 2: Architecture Specific Directories

Each architecture directory also needs certain subdirectories. Under the `x86` directory make subdirectories as shown in Figure 3.

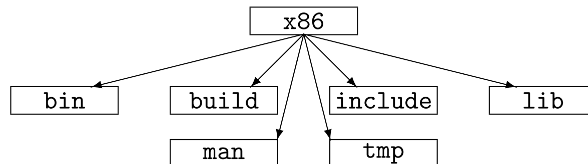

Figure 3: x86 Subdirectories

x86 is the architecture that we will use most often, and it is the only one whose directories we will set up today. Eventually, all three architectures should have the same set of subdirectories. Creating each architecture's subdirectories by hand would be tedious. Soon you will be writing a shell script to set up the architecture specific subdirectories automatically.

`arch/x86/bin` A `bin` directory is where you store executable files. Because this `bin` directory is under the `arch/x86` directory it stores architecture specific executable files for the x86 architecture. These files are also called

binaries, hence the name bin. These kind of files are the normal programs that everyone knows and loves.

Unfortunately, binary files only work for one architecture. Imagine that you had written a very useful program in C++, and you compiled it on a x86 machine. The program would work great and do its useful thing on a x86 machine, but if you tried to run it on an sun4 machine you would probably get a message like "`Exec format error. Wrong Architecture.`" To solve this problem you need to compile the program on a sun4 machine, but this will create a new executable file with the *same name* as the old one. If you create it in the same directory it will overwrite the old program, and you will no longer be able to use the program on a x86 machine without recompiling again.

Rather than recompile the program every time you switch architectures you should have multiple copies of the same program that work on different architectures. These copies have to be placed in different directories. That is the purpose of the `x86/bin`, `sun4/bin`, etc. directories.

You want the program to be in your path, but which copy? You have multiple copies of the program in different directories. Which directory should be in your path? The string `.../arch/$ARCH/bin` solves this problem. The variable `ARCH` automatically selects the bin directory that contains programs that will work on your current architecture. Add the directory `~/csci3308/arch/$ARCH/bin` to the path in your `.cshrc` file. Also, copy to this directory the hello program from `~csci3308/arch/$ARCH/bin` and use the where command to make sure your path is correct. Remember to source your `.cshrc` file, and type `rehash`.

`arch/x86/build` When you compile programs they often create many intermediate files that are only used in the compilation process. For example, to compile a C++ program the compiler first creates a `.o` (object) file for each `.cpp` source file. Then these `.o` files are combined to create the executable file. Once the executable file is created the .o files are unnecessary to running the program and can be deleted. If the program is undergoing changes then keeping `.o` files around can speed up compiling the program because only those `.cpp` files that change need to generate new `.o` files. Usually, `.o` files are kept around while a program is being changed, but are deleted when the final version is compiled.

A naive approach to keeping intermediate files around would be to compile the program in the bin directory where the executable should go. Unfortunately, this creates some problems.

a. Intermediate files clutter the bin directory and make executable programs harder to find.

b. When you want to delete intermediate files it is harder to clean up all of the unnecessary files without deleting files you want.

c. If two programs both create an intermediate file with the same name they will overwrite each other.

To solve the first two problems, programs are compiled (built) in the build directory, and to solve the third problem each program is given its own subdirectory under the build directory. The program is built in that directory, and all of the intermediate files are created there. When the final version of the program is completed the executable is copied (installed) to the `bin` directory, and the entire `build` directory for that program can be deleted without worrying about deleting anything you need.

`arch/x86/include` **and**

`arch/x86/lib` These directories hold libraries of functions that can be used and reused by different programs without rewriting, or even recompiling, the library. This topic will be discussed in more detail later in the semester.

`arch/x86/man` This directory holds man pages that are architecture specific. Believe it or not, some programs work differently on different architectures, and so need different man pages.

`arch/x86/tmp` This directory holds any architecture specific temporary files.

`bin` This directory holds files that are executable, but are architecture independent. An example of this is a shell script which is not compiled, and can be run on any architecture. Place this directory in your path. Also, copy the hello program from `~csci3308/bin` to this directory and use the where command to make sure your path is correct.

`doc` This directory holds documentation other than man pages. The man program searches man directories automatically, and expects man directories to be set up in a certain way. Therefore, it is usually best to keep non-man page documentation out of the man directory.

`lib` This directory holds reusable components that behave like libraries, but are not architecture specific. An example would be a high score file for a game. You want the program to use the same high score file no matter what architecture you are on.

`man` This directory holds man pages. Most man pages are stored in directories where you do not have write access. Thus, you have a problem if you download a new program and you want to store and access its man pages. The solution is to store its man pages in your own man directory. Under the man directory create a directory called `man1`. In this directory create a text file named `simple.1` containing the text "`This is my man page.`" or something to that effect. Now type:

```
man simple
man -M$HOME/csci3308/man simple
```

The man program can only find your simple man page if you tell it where to look. You don't want to do this every time, so there is an environment variable called MANPATH that works for man pages like path works for programs. MANPATH is set in your .cshrc file like your path. Edit your .cshrc file to add `$HOME/csci3308/man` and `$HOME/csci3308/arch/$ARCH/man` to your MANPATH. Re-source your .cshrc file and make sure man can see your simple man page.

`misc` Some files don't fit in any classification under our directory structure. Place these files in the misc directory.

`src` This directory holds source code for any programs you build. You want to keep source code, intermediate files, and executable programs separate for all of the reasons listed in the description of the build directory above. Like the build directory, the source directory will have a subdirectory for each program in case two programs have files with the same name.

`tmp` This directory holds temporary files. You should feel free to remove all of the files in your `tmp` directory at any time. Consequently, you shouldn't put anything in your `tmp` directory that you don't want deleted. Use the `misc` directory for this instead.

Your First Shell Script

A shell script is an ordinary text file containing a list of shell commands. Every time the script is run it runs all the commands listed in the file one after another. Use Emacs to create a new file containing the following text:

```
#!/usr/local/bin/tcsh
echo The current date and time are:
date
```

11. This shell script is an architecture-independent executable file. In what directory should it be placed?

Now you need to give permission to run your script. Type the following:

```
chmod u+x filename
```

where *filename* is replaced by the name of your script. If your shell script is in the right directory it should be in your path and you can access it from anywhere. Use which and where and the name of your shell script to make sure it's in your path, and then type the name of your shell script to run it.

Note: make sure that the path to tcsh is specified correctly. Type `which tcsh` to make sure you are using the correct path.