Using UNIX[1]

The computers that we will be using for CS50 run HPUX, a variant of the UNIX operating system. An operating system is the program that manages the resources of a computer, mediates among multiple users, and insulates you (the user) from the details of the hardware and from the other users

If you have used a PC, then you might be familiar with DOS, which stands for “Disk Operating System”, or its more recent relatives, Windows 3.1, ’95 and NT. If you have used a Macintosh, then you might be familiar with the Mac Operating System. UNIX systems are different from DOS, Windows and Mac OS in that they are designed to handle many simultaneous users (called *multi-user systems*). Additionally, UNIX systems can perform many tasks at once (called *multi-tasking*).

Finally, UNIX was designed primarily as a tool for programmers. Rather than offering a complete solution to a problem (e.g., a package that lets you edit text, format it, and print it all from within a single program), UNIX provides you with a large assortment of tools that can be mixed and matched to accomplish a wide variety of tasks. You might find this approach confusing and frustrating at first, but bear with us; once you know your way around the system, you will find it a very productive environment.

The book “Learning the UNIX Operating System” (Todino & Strang) provides a fairly short, simple introduction to UNIX. This book discusses a variant of UNIX called System V, which is similar to HPUX. To find out more detailed information about any command on the system, use the on-line **`man`** pages described in the next section.

1. Logging In

The HP workstations are located in the basement of the Science Center. There are about 20 regular machines in the room to the right of the stairway and another 10 high-performance workstations in the smaller room just to the left of the doorway to the larger room. You may work at the workstations themselves (that is called “working at the console”) or you may access the machines remotely from PCs or Macintoshes in the Science Center or from a machine in your own dorm room.

In order to use the system at all, you will need an account. You can obtain an account using the register program. See the handouts in the basement of the Science Center for detailed information about creating your account.

1. Unix is a trademark of X/Open.

When you go to log into the HP workstations from the console, there will be a window displayed where you enter your logname and password. When you log in, the window manager will start up automatically. You can click on the small terminal icon to create new windows. Clicking on the background will give you a menu that will allow you to log out.

2. Getting Help on the System

UNIX includes a system of on-line manual pages (**`man`** pages) that document all the commands, so it's not necessary to remember how to use all the commands or to lug around a thick manual in order to use UNIX effectively. To find out information about a particular command, try typing:

> **`man`** *<command>*

where **`man`** is the command you are executing and *<command>* is the command about which you are seeking information.

You can also find out information about commands even if you have forgotten the exact name of the command you want. To get a list of all the commands about a certain topic, type:

> **`man -k`** *<topic>*

Sometimes you will see man pages referred to as *name*(1). The number following the name of the man page is the section in which you will find the command. Normally, you will not need to worry about the sections, but in cases where the same topic appears in multiple sections, you may need to specify the section. In this case, you can type:

> **`man`** *<n> <command>*

where n is the section of the manual you need.

In general, you can use the man pages to answer questions and help you learn about the system. They should be your first source of information when you have questions.

3. The Shell

When you type commands on a UNIX system, you are not typing directly to UNIX. Instead, you are typing commands to a program called a *shell*. A shell is a program that reads each line of text that you type, and tries to execute it as a command. The shell is responsible for finding the programs that you want to run and managing all the different programs that you are using. There are quite a number of shells available for UNIX systems. The one that we'll be using for this course is called **`tcsh`**.

For now, you can ignore shell features and just use it to type commands. As you become more comfortable with the system and want to do more complicated things, you can learn more about the shell, especially how to write in it. To find out about some of the power of the shell, look at the man pages.

4. Editing Text

When you are writing software, the tool you will probably spend more time actively using than any other is an *editor*. Therefore, we **strongly encourage** you to invest a little extra time at the beginning of the semester to become familiar with one of the text editors available on the UNIX machines.

There are a number of good programming editors available for UNIX, but by far the most popular editors are **`vi`** and **`emacs`**. Each of these editors has advantages over the other, and there are endless and pointless debates about which is better. For the purposes of this course, however, we recommend that unless you are already familiar with **`emacs`** (and don't want to take the opportunity to learn a new editor) that you learn **`vi`**. It may seem a bit confusing at first, if you are used to text processors such as Word Perfect or Microsoft word, but **`vi`** requires much less memory than **`emacs`** and will usually give you and the other users of the system better response time.

Most powerful editors, such as **`vi`** and **`emacs`**, are not completely documented by their manual pages. Instead, they have their own documentation (including on-line help). We have included in the course packet instructions on how to use **`vi`**, and an **`emacs`** tutorial is available from User Services.

5. Keeping Track of Versions of your Files

Throughout this course, you will be creating files and modifying them. Occasionally, you will make changes to a file and then regret having done so. More frequently, we will want to see a complete history of what changes you made. In order to make this simple to do, we are requiring that you use a *source code control system*. In general, source code control systems allow you to maintain a complete history of the changes you make to a file.

The system that we will be using for this class is called **`RCS`** (Revision Control System). **`RCS`** will keep track of all old versions of a file while you are creating a new version. If you like the changes you have made, and you wish to make them permanent, you can use the **`checkin`** command to do so. Once you have done that, **`RCS`** believes that most recent copy of the file is the "real" one. If, instead, you decide that you do not like the changes you made, you can retrieve any previous version of the file from RCS using the **`checkout`** command. Also, if you are trying to figure out what has changed since an earlier version (because your program used to work and now it doesn't), you can use the **`rcsdiff`** command to show you what has changed.

You can find out all the details of **`RCS`** by looking at the man pages for **`rcs`**, **`ci`**, **`co`**, **`rcsdiff`**, **`rlog`**, and others. In the meantime, we have created some simplified versions of these commands that are described here.

- **`checkin`** *<filename*s> - Tell **`RCS`** that you have a new version of the files *<filenames>*. **`RCS`** will preserve them for future reference. When you execute **`checkin`**, RCS starts up an editor so that you can add comments about why you are checking in each file and any other information you want to record about this revision of the files. You will find it most helpful if you write very precise and detailed comments when you check things in.

  The editor that **`checkin`** uses by default is **`vi`**. If you prefer to use a different editor, then make sure that your environment variable **`EDITOR`** is set to the name of the editor that you prefer. For example, if you prefer **`emacs`**, add the line

  ```
  setenv EDITOR emacs
  ```

  to your **`.login`** file (if you haven't done so already).

- **`checkout`** *<filename*s> - Retrieve that last copy of files that you checked in. checkout can also be used to retrieve previous versions of files, by specifying the name or revision number that you want to check out.

- **`rlog`** *<filename>* - Display a list of all the revisions of a set of files. This will print out all the comments that you added when you ran **`checkin`**.

- **`rcsdiff`** *<filename*s> - Display the differences between the version of each file that you have checked out and the last version that you checked in. If you want to look at different versions than the most recently checked in one, consult the man page.

6. Turning in your work.

We will ask you to submit both paper and on-line copies of your homework. The paper copies make it easier for your Teaching Fellows to grade, and the on-line copies let us run your programs. The program that lets you submit your homework electronically is called **`cs50-submit.`**

Before running **`cs50-submit`**, make sure you have checked in all the files we specified, including any text files that you created. Change into the directory directly *above* your assignment. If all your cs50 materials are in **`~/cs50/`** and you are submitting assignment 1 that you have stored in **`~/cs50/my_assignment_1`**, then you should be in the **`~/cs50`** directory. Now, type the **`cs50-submit`** command. You will be asked for the project name. Type **`asstX`** where **`X`** is the number of the assignment you are submitting. Next you will be asked what to submit. Type the name of the directory in which your assignment resides, in this case **`my_assignment_1`**. The cs50-submit program will create copies of most of the files in your **`my_assignment_1`** directory, placing them where your Teaching Fellow can access them.

7. UNIX Tools

As mentioned at the beginning of this document, one of the things that makes UNIX so useful is its wide assortment of tools. You can also make the tools interact to form more powerful tools. In this section, we will present some of the most commonly used tools and also some common ways that you can "hook tools together". However, the best way for you to learn about the power of UNIX is to experiment with it. Read interesting man pages. Try to do things. Try to make the computer make your life easier!

For each command we give the command name in **`command font`** and a mnemonic (in *italics*) indicating why the command has such a bizarre name, and a description of the command. As always, you can get more information from the man page.

- **`ls`** (list) displays the names of the files in a directory.
- **`grep`** (global regular expression print) searches files for a pattern (called a regular expression).
- **`cd`** (change directory) changes your current working directory.
- **`cat`** (concatenate) displays the contents of a file.
- **`more`**, **`less`** (*show more of a file.* **`less`** *is a newer version of* **`more`***; the name is a pun on the name more*) displays the contents of a file one page at a time. Each time a page is displayed, you can execute a command that searches for text, continues paging, goes to the next file, etc. You will find this an extremely useful command.

You can save the output of any command by using a mechanism in the shell called *Input/Output Redirection*. The symbols used for redirection are "**<**" and "**>**". To save the output of a command in a file instead of having it appear on your screen, use *output redirection* ("**>**"). For example, to save the output of the **`grep`** command, you could type:

```
grep Thurs ~lib50/cs50.times > thursday_sections
```

This command would look for the string "Thurs" in the file "~lib50/cs50.times". It would then put the output of the command (what the command would normally print to the screen) in the file "thursday_sections."

On the other hand, you might have a program which expects the "user" to input a long list of words. You might get tired of typing the words in each time you run the program, so you could use input redirection ("<") to help you:

```
myprog < /usr/dict/words
```

This command line would execute the program **myprog** and give it as input the data in the file "/usr/dict/words".

In addition to using the shell to redirect input and output to files, you can use a shell mechanism called a *pipe* to string commands together. For example, suppose that you wanted to look at all the five-letter words in the dictionary. You could type:

```
grep '^.....$' /usr/dict/words
```

Unfortunately, there are many five-letter words and the list would quickly scroll off your screen. What you might really like is to be able to take this list and use the **more** program to display is one screenful at a time. While you could do this by creating a temporary file:

```
grep '^.....$' /usr/dict/words > tempfile
more tempfile
```

you do not necessarily want to create the temporary file. Worse yet, suppose you were constructing a file so large that it could not fit on your disk. The shell lets you string together the grep and more command using *pipes*.

```
grep '^.....$' /usr/dict/words | more
```

The "|" symbol creates a pipeline between the **grep** command and the **more** command. The output from **grep** (your list of five-letter words) is used as the input to **more**. You can string together as many commands as you like:

```
grep '^.....$' /usr/dict/words | sort | more
```

8. Make

Large programs are often made up of many different files. As your programs become larger, it will become more difficult to keep track of all the different pieces. Fortunately, UNIX provides a tool that does this for you. This tool, **make**, allows you to define rules for how to perform a job and then uses those rules to do the job for you. For example, if you have a large software program, it will consist of many *source files* (sometimes called ".c" files). Each of those source files might use some number of *include files* (sometimes called "header" or ".h" files). In order to create a program that the computer can run, you must compile each ".c" file to create ".o" files and then link the ".o" files together to create the final program. Furthermore, each time you modify one of those ".h" files, you will have to recompile all the ".c" files that use it. You can imagine that this gets tedious pretty quickly.

So you don't have to do each of these steps manually, the **make** utility can keep track of all your ".h" and ".c" files and generate the commands that must be executed to compile and link them. It uses a set of rules to figure out how to do this. The language in which to write these rules is cryptic and difficult to understand, even for experienced programmers. Fortunately, **`make`** already knows all of the rules that you need for this class. We will usually give you a *makefile* (set of rules for **`make`** to use) so all you will need to do is type

> **`make`** *<program>*

Sometimes we will ask you to change the names of the files listed in the makefile. In any case, at this point in your programming career, you need not concern yourself with the details of make. However, you should have an understanding of what it is doing for you.

9. Getting your Programs to work Correctly

The first version of your program rarely does exactly what you intended it to do, and sometimes it can do something completely unanticipated! When this happens, your program is said to contain a bug, and the process of removing these errors is called *debugging*. Luckily, UNIX provides tools to help debug programs.

- **`lint`** - lint is a program that looks at the source code for your program and tries to find "suspicious" things that look like they will cause trouble when you run your program. The C compiler also may warn you about certain errors, but lint is usually better at finding more subtle errors. Your makefiles have rules to help you run lint. You should type **`make lint`** to run lint on your programs.

- **`gdb`** – gdb is a debugger. If you run your program from inside gdb, gdb will monitor the program as it runs. You can stop the program at any point and look at the value of variables, execute one instruction at a time, have it stop when certain conditions occur, etc. You will find that in early assignments you can find most of you bugs by running the program and looking at the code. Therefore you might be inclined to avoid learning to use the debugger. **`This is a huge mistake.`** As your programs become more complex, you will be unable to find bugs merely by looking at the code. It is far easier to learn to use **`gdb`** now when you can understand what your programs are doing, rather than waiting until your programs are very complicated.

- core dumps - If your program has a particularly nasty bug in it, UNIX may detect this while the program is running and automatically stop your program before it can harm the rest of the system (unlike the Mac or PCs, which would just crash). When this happens, UNIX creates a file named *core* in your current directory that will contain as much information about your program as it can determine. This core file can be used by **`gdb`** in order to help perform an autopsy on the program.

You can use **`gdb`** as you normally would, only you cannot execute the program from the core; you can just look at the values of variables to determine where it crashed, why it crashed, and ideally, how to make it stop crashing.

10.Communicating with other users

There are many tools designed to let users communicate via the computer. The tools that we will be using most heavily are **`mail`** and **`news`**. The mail program lets you send and receive messages with users in this class, at Harvard, or nearly anywhere in the world. You will often find *email* (electronic mail) the easiest way to contact your teaching fellows or professor.

There are a variety of different mail reading programs available on the science center machines. The most basic one is called, **`mailx`**, and is described in the Nutshell book. Refer to the man pages and the book for instructions on using mail.

Our other communication medium, netnews, lets you communicate with groups of users. There are a variety of topics (or *newsgroups*) that people read. For this class, you will want to be sure to read the newsgroups:

- **`cs50.announce`**: general announcements about the course and homework.
- **`cs50.talk`**: discussion among students (note that you should not talk about homework on the talk newsgroup).
- **`cs50.test`**: a place to master your news skills.

Refer to the man pages or the appropriate section of The Internet (the Comer book) for more information about reading news. Assignment 2 will cover some introductory material. Also, be careful! Some people find news so interesting that they do little else, including course work. Don't let this happen to you!