FIT2100 Practical #1 Introduction to Unix/Linux, running C Programs Week 2 Semester 2 2019

August 1, 2019

Revision Status:

`$Id: FIT2100-Practical-01.tex, Version 2.0 2018/07/24 18:30 Jojo $`

`Updated by Daniel Kos, July 2019.`

Contents

1 Background

The purpose of this first practical is to provide you with some basic experience on the Unix/Linux system and some general-purpose Unix/Linux commands and utilities.

There are some pre-lab preparation (Section 2) that you should complete before attending the lab. The practical tasks specified in Section 3 will be assessed in the lab.

2 Pre-lab Preparation (2 marks)

2.1 A Brief History of Unix/Linux

A brief history of Unix/Linux is available on the FIT2100 Moodle site. Please have a good read yourself.

Let's get started with a Unix/Linux system by learning the basics of how to drive it from a command shell (i.e. a command line).

2.2 Manual Pages for Unix

The Unix manual pages (also called **man** pages) provide information about most of the commands, programs and libraries on Unix/Linux systems.

To find and display manual pages, the `man` command is used. For instance, to display the `man` page for the `man` command itself, type the following command:

```
1  $ man man
```

(Note: The $ at the start of a command line is the *shell prompt* in Unix/Linux systems. Please do not type $ as part of your command. However, your shell prompt can be different depending on the shell that you use.)

The `man` pages do not contain information about all of the programs installed on the system. You should also realise that the contents of many of the `man` pages is aimed at experienced users, and if you are new to Unix/Linux you may find this a little overwhelming.

Navigating the manual pages The `man` command enters into the so-called a *pager* mode, and will only display the first 20 (or so) lines of the `man` page. You can use a number of commands in the pager to scroll the text:

Command	Description
Press `[enter]`	to scroll forward to the next line of text
Press `[space]`	to scroll forward to the next page of text
Press `h`	to display a list of commands that man pager accepts
Press `/`	followed by a search query to find a word in the man page
Press `n`	after a search to find the next search result
Press `N`	after a search to find the previous search result
Press `q`	to quit and return back to the shell prompt

Pre-class exercise: (0 marks)

- Find out how to scroll backwards a page and then scroll back up to the top of the `man` page.
- Use the `man` command to find more information about the following commands:
  (a) `ls`
  (b) `more`
  (c) `cat`
  (d) `vim`

2.3 C Programming Manual

If you desire, you can also install additional *sections* into the man page system, including a reference guide for each of the standard C library functions (section 3) and OS-specific functions in C (section 2). To install these on your system, enter the following command into the Unix prompt:

```
$ sudo apt install manpages-dev manpages-posix-dev
```

You will be prompted to enter your login password for the Linux system. The password will not be displayed. When asked to continue, enter `Y` and press Enter.[1]

[1] If you get an error, run: `sudo apt-get update` first, and leave it to run until it completes. You will require internet access, and cannot install software while other updates are running.

Try `man printf`. Is this helpful? Ooops! This is the man page for the Linux command `printf`. We want the man page for the C function `printf` instead! So press `q` to quit, then try `man 3 printf` to specify that you want a man page for the C standard library (section 3). While man pages are always highly technical, this man page is more complicated than most! There is a whole family of different `printf` functions in C, such as `fprintf`.

Beware! The top of the man page shows how the function prototypes are *declared* in C (e.g. the data type of each argument to be provided), not literally how to call them. The very bottom of the man page usually contains an example or two on how to actually call one of the functions described. This is reference material: you generally won't want to read the whole man page at once! These man pages are very useful for quickly looking up details such as what #include statements are required for the function you need, what arguments are required, and the data type of the function's return value.

2.4 Unix Commands

Unix commands are essentially executable files representing programs — mainly written in the `C` language. These program files are stored in certain directories such as `/bin`. (We will explore the Unix file system in the next subsection.)

Unix commands are in lowercase (remember that case is significant in Unix). The syntax for Unix commands is invariably of the form:

```
$ command -option1 -option2 ... other-arguments
```

Note: You can press the UP arrow key to repeat a previous command, which is especially handy for repeating long command lines.

You can also *auto-complete* commands, filenames, and directory names by typing the start of the name and pressing TAB. If nothing happens or the filename is not completed fully, it means there are multiple files with similar names available; just press TAB a second time for a list of possibilities. Learning to use the TAB key will save you a lot of typing when navigating the command environment. You may even find it quicker than using the desktop file manager!

Before trying out some commands, create a directory (or folder) named `PRAC01` under the FIT2100 folder (`~/Documents/FIT2100`), then change to this `PRAC01` directory, using the following commands:

```
$ cd Documents/FIT2100
$ mkdir PRAC01
$ cd PRAC01
```

After completing the above commands, confirm which directory you are in with the following command:

```
$ pwd
```

Now, we are going to try out some UNIX commands. Before that, if you would like to clear the contents on your screen, the `clear` command does the job.

```
$ clear
```

2.4.1 Basic Commands

Try to understand the following commands. These commands can be used to find out who the users are since Unix is a system used by multiple users. What are the differences between these commands (check their `man` pages)?

```
$ who
$ w
$ users
```

Sometimes, you may forget about your username, especially when you have a number of accounts on a particular system. Try the following command:

```
$ whoami
```

To find out your terminal name, the following command is used. (Note that a hardware device is also a file in the Unix/Linux file system.)

```
$ tty
```

Sometimes, you would like to know just what a command does and not get into its syntax. For example, what does the cp command do? The `whatis` command provides a one-line answer:

```
$ whatis cp
cp    -- copy files and directories
```

Often times, once you have identified the command you need, you can use `man` command to get further details.

The `whereis` command can be used to locate the executable file represented by a command. As mentioned above, all Unix commands are essentially executable files representing programs. Try the following command and see what is does.

```
$ whereis pwd
```

The `apropos` command performs a *keyword* look-up to locate commands. For example, if you wonder what the command to copy a file is, the following command could be used:

```
$ apropos "copy files"
```

The `man` command together with the option `-k` is an alternative to `apropos`. You can try this to verify:

```
$ man -k "copy files"
```

The `uname` command is useful for finding out the type of operating system (OS) running on the machine that you are using. The option `-n` provides you the machine name, while the option `-r` shows the version number of the OS.

```
$ uname
$ uname -n
$ uname -r
```

To display the system date and the calendar, try the following:

```
$ date
$ cal
$ cal 7 2001
$ cal 7 2018
$ cal 7 2999
```

The `wc` is the command for counting the number of lines, words, and characters in a file. Try the following command and explain the output:

```
$ wc /etc/passwd
```

Generally the `passwd` file, which is a text file, contains information about all the users registered on the system, with each line referring to a different user. Even if you're the only user on the system, a lot of background processes running on the operating system run under their own dedicated user accounts for security reasons. Let's take a look.

```
$ cat /etc/passwd
```

Often times, you may want to search a file for a pattern and display all the lines (of the file) that contain the given pattern. `grep` is a useful utility for locating words (or other patters) in files. If your username on the system is 'student', let's try to find information about your account inside the `passwd` file. Try the following command with the `grep` command:

```
$ grep student /etc/passwd
```

Pre-class exercise: (2 marks)

- Which Unix command converts a PDF file to `jpeg` format? (Note: jpeg is a popular data compression method.)
- Which Unix command displays the status of disk space (i.e. the number of free disk blocks) on a file system?
- Which Unix command displays the type of a file (e.g. ascii text, executable, etc.)?

2.4.2 Combining Commands

So far, you have been executing individual commands separately. In fact, Unix allows you to specify more than one command in the same command line. Each command has to be separated from the other by a semicolon (;).

```
$ who; date; cal
```

You can even *redirect* the output of these commands to a single file. For example:

```
$ who; date; cal > newfile
```

You can then view the contents of this file with the cat command:

```
$ cat newfile
```

Suppose that you inadvertently pressed the [enter] key just after entering cat:

```
$ cat [enter]
```

There is no action here; the command simply waits for you to enter something. You can use the [ctrl-D] key to get back to the shell prompt.

(Note: To *interrupt* a running program, you can use the [ctrl-C] key. This will then cause the running program to terminate. This is a good way to terminate a program that is stuck in an infinite loop.)

2.4.3 More Useful Commands

In addition to the cat command, there are a number of commands available in Unix for displaying the contents of files on the screen.

The more command is to display the contents of a file in one screenful at a time. (Press the [space] key to get the next screen.)

```
$ man man > test.txt
$ more test.txt
```

A similar command to more is called less. Can you find the difference between more and less?

The touch commands allows you to create a new empty file.

```
$ touch test2.txt
$ cat test2.txt
```

To display the first few lines of a text file (10 lines by default), the head command is available. A similar command is tail, which displays the last few lines of a text file (10 lines by default).

```
$ head test.txt
$ tail test.txt
```

The command `history` is used for displaying all of the stored commands in the history list.

```
$ history
```

Another command which is similar to a `man` page — `info` — can be used to display the information page for a given command. For example:

```
$ info pwd
```

To find out a list of jobs (processes) started in the current shell environment, use the `jobs` command:

```
$ jobs
```

The `free` command is useful for finding out the amount of free and used memory (both physical and virtual), with basic information about how that memory is being used.

```
$ free
```

Finally, you can cause the shell to "sleep" for a specified number of seconds.

```
$ sleep 5
```

2.5 The Unix File System

In a Unix/Linux file system, everything is treated as *file* — most objects in the system can be accessed in a file-like manner.

Terminology A *file* in the system contains whatever information a user places in it. There is no format imposed on a regular file; it is just a sequence of bytes.

A *directory* contains a number of files; or it may also contain subdirectories which in turn contain more files. There is only one *root* directory. All files in the system can be traced through a path as a chain of directories starting from the root directory.

When a file is specified to the system, it may be in the form of a *path name*, which is a sequence of filenames separated by slashes. In Unix/Linux systems, a forward slash (/) is used (rather than a backslash). Any filename except the one following the last slash must be the name of a directory.

Unix files All files in a Unix/Linux file system are treated equally — i.e. the system does not distinguish between a text file, a directory file or other file types. It is up to the user to know the type of file they are using, which can result in operations being attempted on incompatible file types (e.g. printing executable output files to printers).

The command `file` can be used to give a fairly reliable description of the content of a file.

```
1 $ file /bin
2 $ file /bin/bash
3 $ file /etc/passwd
```

2.5.1 Handling Files and Directories

The Unix/Linux operating system provides a number of commands to create, modify, traverse and view the file systems. Make sure you know what each of the following commands does and how to use each command.

Command	Description
`cd`	change to another directory
`pwd`	print the present (working) directory
`ls`	list a directory's contents
`mkdir`	make a directory
`rmdir`	remove a directory
`rm`	remove a file or a number of files
`mv`	rename/move a directory or a file
`cp`	copy a file
`df`	display information about free space on the available file systems
`du`	display disk usage information (for files and directories)

The cd command without any argument takes you back to your home directory from anywhere.

The `ls` with the option `-l` will display more information about files. For example, you can see the file size with this command.

```
$ ls -l
```

Special characters Make sure you know how to use the special characters, such as '*' and '?' with the above commands.

Wildcard symbol	Description
* (asterisk)	represents all ordinary files in a directory
?	represents a single character
~ (tilde)	represents your home directory

Pre-class exercise: (0 marks) Make sure you understand what is happening as well as the output after the execution of each of the following commands.

```
$ man man > f01.txt
$ cp f01.txt f02.txt
$ cp f01.txt f001.txt
$ cp f01.txt f002.txt
$ cp f01.txt test_f01.txt
$ cp f01.txt test_f02.txt
$ cp f01.txt f1.txt
$ cp f01.txt f2.txt
```

```
$ ls
$ ls f*.txt
$ ls f?.txt
$ ls f??.txt
$ ls f???.txt
$ ls test*.txt
$ ls ??.txt
$ ls ???.txt
$ ls ~
```

After you have a good understanding of the above, remove all of these text files using the following command:

```
$ rm f*.txt t*.txt
```

2.5.2 Unix File Attributes

Each file in the Unix/Linux file system has a number of *attributes* associated with it. Some attributes that are associated with a file include:

- name
- size
- permissions/protection
- time/date of modification
- owner
- group

To investigate file attributes, change your working directory to your home directory. Get a full directory listing by typing the command: `ls -l`. Make sure you know what each piece of information means. In the left column is a list of permission bit flags for each file. For example, if a file has permission bits: `-rwxr-xr-x`, it means (reading from left to right) that the *owner* of the file can read `r`, write `w` and execute `x` the file[2], while the group of users the file belongs to can read `r` the file, cannot write `-` to it, but can execute it `x`. Everyone else (last 3 flags) can also read, cannot write, but can execute this file.

Pre-class exercise (0 marks)

(a) Under the directory `~/Documents/FIT2100/PRAC01` make a subdirectory called `test_dir`.

(b) Create an ordinary file called `file.txt` under `test_dir`.

(c) Check the current file attributes (especially the permissions) of `file.txt`.

(d) Use the following commands to change the permissions of `file.txt`. Verify the effect of each command using `ls -l file.txt`. (Make sure you understand the *permissions* represented by each *octal* number. When you convert each octal digit into binary, it matches the three permission bits mentioned above. The three octal digits correspond to permission bits for the owner of the file, user group and everyone else respectively)

```
$ chmod 000 file.txt
$ chmod 777 file.txt
$ chmod 666 file.txt
$ chmod 444 file.txt
$ chmod 664 file.txt
```

[2] only executable programs and directories should normally have executable permission set

```
$ chmod 600 file.txt
$ chmod 466 file.txt
$ chmod 251 file.txt
$ chmod 111 file.txt
$ chmod 700 file.txt
```

Another way to use `chmod` is with the + (add permission) and – (remove permission) operators. For example, `chmod +x file.txt` will make the file executable for all three categories of users.

2.6 The Shell for Unix/Linux

The command-line interface you have been using inside your terminal is more correctly called a 'shell' utility.

There are several Unix shells available which largely fall within two classes — the 'Bourne' shells (`sh`, `ksh`) and the 'C' shells (`csh`, `tcsh`). The `bash` shell (short for `Bourne Again SHell`) is the default shell in your Linux environment. The shell is actually a primitive kind of programming environment (not to be confused with the C Programming Language), used to interact with the system.

2.6.1 Aside: Pre-processing of Unix commands

Let's look more closely at what happens when you type a command into the shell prompt in a Linux terminal...

- The shell places the prompt on the user terminal and goes to sleep.
- The user types a command line consisting of one or more commands. These commands may be separated by the following symbols: `;` (sequentially execute), `||` (otherwise execute), `&&` (if ok then execute).
- When the user presses `[enter]`, the shell begins processing the command line.
- First step in this pre-processing is to *parse* the first command. If there are more than one command in a line they will be processed and executed after the first command has finished execution. However, the decision will be based on the separator (`;`, `||`, or `&&`) you use between the commands.
- The command is broken into its constituent words. The end of the words is usually identified by the spaces, tabs and special symbols. The command line parser is often not as nice as those used by the programming languages. As a result, sometimes your

command may not be understood if there is an extra space or you miss a space between the words.

- Next, the shell replaces variables by their values (the shell can make use of special system variables known as 'environment variables'). These variables are shown in the command by preceding them with the symbol $.
- *Command substitution* is done next. A command substitution is indicated by enclosing the command in a pair of back-quotes (``). (Note: this quote is usually found on the left end of the top row on your keyboard.)
- The shell then performs redirection of the standard input, standard output and standard error output, if requested.
- Wild-cards are expanded next.
- Finally the command is ready to execute. The shell searches for an executable file whose name matches the command name.
- While the command executes, the shell waits.
- When the execution finishes, the shell displays next prompt on the terminal. A new cycle begins.

If you want to know what error value the last program returned when it terminated, you can use the following command. By convention, any piece of software on the machine that completes running successfully should return a value of 0.

```
$ echo $?
```

2.6.2 Writing C Programs with Unix Commands

Under the directory ~/Documents/FIT2100/PRAC01, do the following:

- Use the following command to create a file: $cat > prog01.c

```
$ cat > prog01.c

#include <stdio.h>
int main (void)
{
    printf("This is the first FIT2100 practical.\n")
```

```
7         return 0;
8     }
```

- Finally, use [ctrl-D] (which sends 'end of file') to finish the session with the cat command.

Do you notice an error in the C program above? There should be a semicolon ';' at the end of the longest line. This error is deliberate.

(Note: This is a difficult way to work with files! You will probably wish to use a text editor, either console-based such as pico, vim or joe, or the graphical editors
tt pluma, or subl (non-free) to work with text files.)

2.7 Compilation and Execution of C Programs

In this section, we will have an overview on how the C compiler and linker work in compiling executable C programs. Before that, let's have a quick recap on what we have learned in the Week 1 Tutorial.

How to create a C program? You can use any text editors of your choice to type in the source code. The source file should be saved with the .c extension, for example simple.c.

How to compile a C program? Before you are able to execute a C program, the program needs to be first compiled using the C compiler, such as gcc. To compile with gcc, one of the following commands can be used:

- `gcc simple.c`
- `gcc -o simple simple.c`

Note: The first command produces the *executable* file named a.out from the source file. However, the default executable file name can be re-named using the -o argument followed immediately by the desired output file name.[3] the second command sets the name of the executable file to simple.

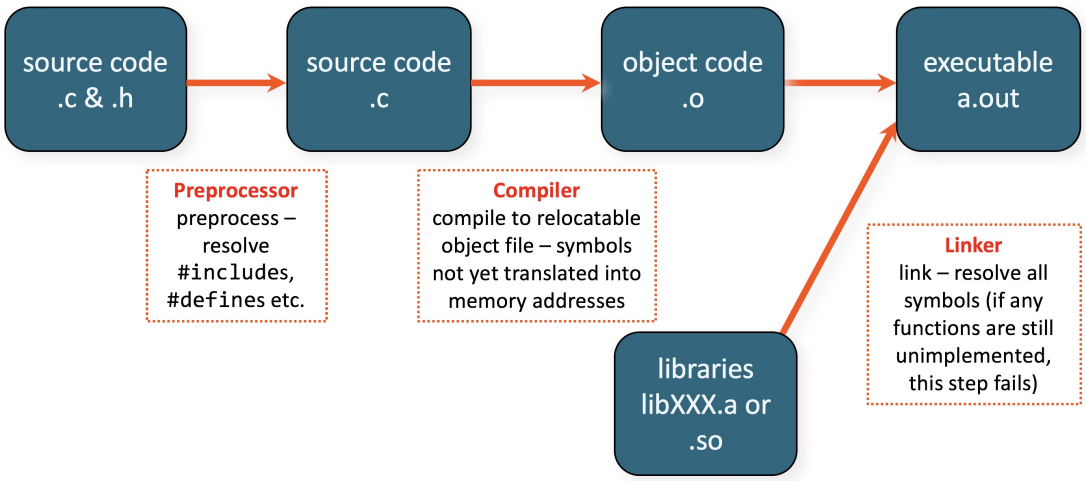
How to run a C program? You can now run the C program with one of the following commands based on the name of the executable file:

[3]The reason for the default name a.out is purely historical. It is simply short for 'assembler output.' Unlike Windows, executable files in Linux typically don't require any filename extension as long as they are set up with executable permission, and there is no such thing as a '.out' file type.

- `./a.out`
- `./simple`

2.7.1 Aside: How Do The Compiler and Linker Work?

There are three main steps involved in compiling a C source code into an executable program:



(Adopted from FIT3042 Courseware by Robyn McNamara)

Preprocessing The C source code is first given to a *preprocessor*, which looks for the *directives* (that begin with the # symbol). Directives such as `#include` and `#define` are resovled in this step.

For `#include`, the given header file is opened and its contents are copied into the current source file. For `#define`, the defined identifiers are searched are replaced with the specific values.

Compiling The modified source code now goes to the compiler, which translates it into machine instructions, also known as *object code*. The program at this step is not yet ready for exeuction; since symbols (such as variables and function names) are not yet translated into memory addresses.

Linking In the final step, the linker combines the object code produced by the compiler with any additional code that needed to produce a complete executable program. The linker attempts to resolve all the symbols and library functions at this last step.

2.7.2 GCC Compiler Options

A summary of various gcc options for quick reference.

Option	Meaning
`-c`	compile source but do not link
`-S`	stop after the compilation stage; do not assemble
`-E`	stop after the preprocessing stage; do not compile
`-g`	embed debugging information inside the executable file
`-O`	optimise the code to a specific level (e.g. `-O3`)
`-W`	turn on or off particular warnings (e.g. `-Wall` for all warnings)
`-I`	specify a directory to look for include files (e.g. `-I/usr/lib/somelib`)
`-L`	specify a directory to look for library files (e.g. `-L/usr/lib/somelib`)
`-o outfile`	place the output in `outfile`

2.8 Using valgrind to find memory errors

In C programming, you may often experience undefined behaviour due to accessing memory incorrectly, or a *segmentation fault* where the operating system shuts down a program that has tried to access a forbidden part of memory. These errors can be frustrating and hard to pin down. For example, going past the end of an array in one part of your program might accidentally corrupt a different variable in a different part of your program!

The `valgrind` utility can help. This utility runs your program in a partly-simulated environment to check if your program is accessing memory safely. If your program has been compiled in gcc with the `-g` option, `valgrind` can even pull debugging information out of the executable file to tell you which line numbers in your original C code the problem might have come from. If your executable file is named `a.out`, you can run your program through `valgrind` like this:[4]

```
1 $ valgrind ./a.out
```

2.9 Shutting down

Like any modern computing environment, your virtual machine environment must be shut down safely when you are done using it. Always shut down your virtual machine at the end of

[4] Run `man valgrind` for the complete user manual.

your session, and make regular backups of any important work!

3 Practical Tasks (8 marks)

Complete the following tasks, writing your answers (or what you did) in a plain text file (together with the marked pre-lab preparation exercises), and get your answers marked off by your tutor before you leave.

3.1 Task 1 (3 marks)

Experiment with navigating to the following paths using the cd command. After navigating to each directory, use the ls command to look at the files in that location. Are you able to navigate to all of these paths? Does it matter what working directory you are in before running the cd command?

- `/home/student/Documents`
- `FIT2100`
- `~/Documents/FIT2100`
- `.`
- `..`
- `.././.././home/student`
- `/`

After experimenting, answer the following questions:

(a) What does it mean if a path starts with a slash / character?

(b) Can you figure out the meaning of these special 'directories': `.` `..` `~`

(c) Why is it necessary to place `./` in front of the name of a program to be run from the current directory? (Hint: what might happen if you had an executable named `ls` in your current directory?)

3.2 Task 2 (5 marks)

Enter the following program into a file named `task2.c`, then complete the following tasks.

```c

#include <stdio.h>

int main() {

    char string='a';
    //forgot how to make a string :(

    printf("Enter a word:");
    scanf("%s", string);
    //prompt the user to enter a word

    printf("You entered: %c", string);

    return 0;
}
```

(a) Using gcc, compile two slightly different executables from your `task2.c` file. The first executable, named `task2`, should be compiled as normal. The second executable, named `task2-debug`, should be compiled using the `-g` option from the table in section 2.7.2 above.

(b) You may have seen one or more warning messages. With good reason! While gcc is able to compile your code, the code is unlikely to run correctly, and may randomly produce a segmentation fault due to trying to access memory in an incorrect way.
Reading the warning messages, **identify the line number and column number of the place in your source code that triggered the warning.**
(You do not need to fix the code at this stage.)

(c) Compare the file size of the two executables generated (try `ls -lh` to see the file sizes in 'human' format). Why is one file larger than the other?

(d) Run your `task2` executable through `valgrind`. Be patient, `valgrind` is much slower than running your program natively! Interact with your program and also read the extra information `valgrind` spits out. Is the information useful? Now try running your `task2-debug` file instead.
What is the difference in Valgrind's output between the two different versions?

(e) What is the program supposed to do? What is the program actually doing?
Hint: Valgrind always displays memory addresses in hexadecimal, for example 0x61 means

hexadecimal 61. A `char` variable in C is used to store the value of a single character. What is the ASCII character code for the character 'a'?

3.3 Task 3 (Extension: 0 marks)

Fix the program from Task 2.

4 Postscript

In practice, compiler warnings should never be ignored! Note that a compiler warning will often only tell you part of what is wrong, but you should aim to treat the cause rather than the symptom. While it is usually easy to change your code (e.g. adding or removing `&` or `*` characters) to force a warning to disappear, this usually only masks the problem rather than solving it.

Valgrind is a useful utility for identifying problems such as memory leaks in larger C programs. Another debugging tool is `gdb` which will be discussed in the future.

The `bash` command-line shell utility (located in the `/bin/` directory of your system), which is set up to run automatically each time you start a new terminal session under your student account (did you notice `/bin/bash` at the end of your user account entry in the `etc/passwd` file?) is a powerful (and old-school) interface to using the operating system, but it's only one possible interface. In reality, `bash` is a C program like many of the other utilities in the Linux environment. The way it provides access to the operating system's services (like browsing directories and running programs) is by making *system calls* to the operating system. Future labs will cover various system calls in detail.