



**The University of
Chicago**
Department of
Computer Science

CMSC 16200 – Honors Introduction to Computer Science 2
Winter Quarter 2007
Lab #1 (01/08/2007)

Name:

Student ID:

Lab Instructor:

Borja Sotomayor

Do not write in this area				
1	2	3	Doc	TOTAL
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

Maximum possible points: 40

NOTE: The lab session will take place in the Linux machines of the Maclab. This lab assumes that you know your way around a UNIX system. If you don't, please read the addendum "Getting Acquainted". However, take into account that you are free to do the lab exercises on Mac machines.



Writing and running shell scripts

[This part of the lab will not be graded. Its purpose is to help you get acquainted with the steps involved in running a shell script]

There are many different shells available for UNIX systems. The Linux machines in the Maclab use BASH (Bourne Again Shell). If you have a UNIX system installed in your personal computer, you are encouraged to use BASH or a Bourne-compatible shell.

To get acquainted with this shell, we will start by writing and running the following simple shell script:

```
#!/bin/bash  
echo "Hello, world!"
```

A shell script is nothing more than a text file with a list of shell commands which, in theory, could be written line by line into the shell prompt. However, a shell script is much more than just a sequence of commands. BASH scripts can include conditional statements, loops, subroutines, and many other programming constructs.

Using your favorite editor, type the above code and save it in a file called *hello.sh*. We suggest you start creating an directory structure in which to place your lab files. For example, inside your home directory create a *cmsc16200* directory, then a *labs* directory inside it and, finally, a *lab01* directory. So, your *hello.sh* file would have the following path:

```
~/cmsc16200/labs/lab01/hello.sh
```

To run the shell script, you will need to mark it as executable:

```
chmod u+x hello.sh
```

Now, try to run the script:

```
./hello.sh
```

You should see the following:

```
Hello, world!
```



Now, repeat the above steps for the following program (save this file as *hello2.sh*):

```
#!/bin/bash

if [ $# -gt 1 ]
then
    echo "ERROR: Wrong number of parameters."
    echo "Usage: $0 [name]"
    exit 1
fi

if [ $# -eq 1 ]
then
    echo "Hello, $1"
elif [ $# -eq 0 ]
then
    echo "Hello, world!"
fi

exit 0
```

BASH is very picky about spaces and the exact position of each keyword (if, then, fi, ...). Make sure you type in the script exactly as shown above.

For the remainder of the lab you will be asked to solve three shell script programming exercises. For some of them you will be able to draw from material covered in the class lectures, but you will also need to refer to BASH documentation to figure out how to perform certain tasks. In fact, the exercises in this lab are pretty simple (as far as shell scripting is concerned), and the real challenge lies in finding the adequate UNIX command to perform a certain task, the exact syntax of a particular BASH programming construct, etc. In this regard, you will probably find the following material useful:

BASH Programming Introduction

<http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>

Advanced Bash Scripting

<http://www.tldp.org/LDP/abs/html/>



Exercise 1 <<10 points>>

Write a script **list_contents.sh** that takes a directory path as an argument. The script must loop through the contents of that directory and...

- For each directory, print the name of the directory and the number of files in that directory.
- For each file, print the name of the file and number of lines in that file (you can assume that all files are text files, not binary files).

For example:

```
./list_contents.sh /home/borja
```

```
Directory docs has 7 files  
Directory files has 57 files  
File foobar.txt has 7 lines  
Directory html has 5 files  
File grades.txt has 21 lines  
File sample.txt has 5 lines
```

Exercise 2 <<20 points>>

System administrators often have to react to Denial-of-Services (DoS) attacks, where a server is bombarded with so many requests that it cannot process all of them in a timely manner, often disabling the server in the process. When the attacker uses several machines for the attack, this is a Distributed DoS (DDoS) attack. In these attacks it is important to identify the machines involved in the attack, so connections originating from those machines are immediately dropped (so they will not consume any of the server's resources).

You are provided with the request log file of a web server undergoing a DDoS, and will have to process the file using UNIX tools to determine what machines are attacking the web server. We will describe the log format to you, and tell you how you can detect the attack (it will be up to you to write the script that detects the attack).

Log format

Web servers, such as the Apache HTTP Server (<http://httpd.apache.org/>), keep a log of all visits to a website in a text file called the *access log*. One common format for this file is the *Combined Log Format*, which stores information such as the file accessed, the date of access, the IP address making the request, and other useful information. A line in Combined Log Format could look like this:



```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0"  
200 2326 "http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I ;  
Nav) "
```

The general format is the following:

IP_addr ident user [date] "request" http_code file_size "referrer" "browser"

For this exercise, we will only be concerned with the following fields:

- *IP_addr*: The IP address that originated the request.
- *request*: The file requested. For example, if my website is:
<http://www.foobar.com/>
and I accessed the URL:
<http://www.foobar.com/info/index.html>
The *request* field would be `"/info/index.html"`.

The other fields, and their contents, are not relevant to this exercise. Ignore them.

Detecting the attack

There are many different types of DDoS, and here we concern ourselves with an attack commonly suffered by Wordpress-based websites (Wordpress is a popular weblog-publishing software package, <http://www.wordpress.org/>). Posting a comment on a Wordpress website involves requesting the file `"/wp-comments-post.php"`. Attackers will directly request that file (bypassing all other pages in the weblog, such as the index file, the pages of individual postings, etc.) and will post a large number of bogus comments in a short period of time, with the sole purpose of overloading the web server (and the weblog's database backend).

So, a rough way of detecting such an attack is processing the web server log file in search of IP addresses that request `"/wp-comments-post.php"`, but don't request any other files.

You are provided with the access log of a real web server undergoing the DDoS attack described above. You are asked to write the following scripts. Each script takes at least one parameter: the log file.

1. **list_ips.sh**: Prints to standard output all the IPs that request file `"/wp-comments-post.php"`. This list should have no repeated values.
2. **list_ips2.sh**: Hardcoding `"/wp-comments-post.php"` in your script is bad karma. Modify the script so it will accept the filename as a second parameter. Also, check that a second parameter has been provided. If not, print an error message and exit.



3. **remove_false_positives.sh**: The previous scripts will obviously produce a lot of false positives, since they still include those IPs making legitimate requests for `"/wp-comments-post.php"`. Modify the above script so it will only include those IPs that request `"/wp-comments-post.php"` and request no other files. You can also implement this in such a way that you pipe the output of `list_ips2.sh` to `remove_false_positives.sh` (in that case, please make sure you name your script **remove_false_positives_piped.sh**)

Do not overthink these three exercises. Simply by using existing UNIX commands and correctly piping them together, each script can be written with just a few lines.

Exercise 3 <<5 points>>

For auditing purposes, we wish to create a log file to keep track of the size of our home directory. This script is meant to be run every day, but we will not be concerned with the mechanism that schedules the script¹. Every time the script is run, it must append a line at the end of the log file with the following information:

[date] directory num_kb

For example:

[Mon Jan 9 16:31:14 CST 2006] /home/borja 1042

The log file is assumed to be called **usage.log** and located in your home directory. The location of the log file can be hardcoded into the script.

¹ If you're interested in how to schedule tasks, the commands to use are *at* or *cron*. In most UNIX systems you can schedule a script to be run every day by placing the script in the `/etc/cron.daily/` directory.



Documentation <<5 points>>

For this lab, you are simply required to include adequate comments in your code. BASH allows single-line comments using the # character. For example:

```
#!/bin/bash

# Script name: Personalized Hello World
# Description: Prints out a "Hello, world!" greeting, or a
#              personalized greeting
# Usage: hello2.sh [name]
# Parameters: If "name" is specified, the greeting will be personalized
#              for that name. "name" must contain no spaces.

# Check if the number of parameters is correct
if [ $# -gt 1 ]
then
    echo "ERROR: Wrong number of parameters."
    echo "Usage: $0 [name]"
    exit 1
fi

# If a parameter is specified, we print a personalized greeting.
if [ $# -eq 1 ]
then
    echo "Hello, $1"
# Otherwise, we print a standard "Hello, world!" message
elif [ $# -eq 0 ]
then
    echo "Hello, world!"
fi

exit 0
```



Addendum: Getting Acquainted

The purpose of this addendum is to provide you with the bare minimum knowledge necessary to know your way around a UNIX system, with pointers to more complete documents. If you are completely new to UNIX systems, we encourage you to read these documents and consult with your lab instructor if you have any questions.

Interacting with a UNIX system

UNIX systems in general, and Linux distributions in particular, usually provide users with two interfaces:

- Command-line interface (or *shell*): Allows the user to interact with the system through the use of commands which you must type in using the keyboard. There are many different types of shells, such as BASH, CSH, TCSH, ...
- Graphical interface: Allows the user to interact with the system through the use of graphical elements such as windows, buttons, text fields, etc. mainly using the mouse. Although the taxonomy of graphical interfaces in UNIX can be a bit complex, most new users will generally interact at first with high-level desktop environments such as KDE and GNOME, which are similar in many respects to graphical interfaces in Windows and Mac systems.

If you are new to UNIX, we encourage you to use the KDE desktop environment, which is very intuitive and easy to use, specially if you have previous experience in Windows or MacOS systems. To use KDE, make sure you choose a "KDE Session" before logging in (you can do this clicking on the "Session" button in the login screen).

However, most of the lab exercises will require using a shell for most, if not all, tasks. You can bring up a console in KDE by choosing "Terminal console" in the System menu (inside the KDE menu, similar to the Windows "Start" menu).

Text editing

Text editing can be performed using text-interface programs (from a shell) or graphical programs (from a graphical interface). You are free to use any text editor you want during the course. Graphical editors (such as kedit, kate, gedit, ...) are easy to use, but are too general-purpose and lack versatility. Text-based editors (vi and emacs) are powerful and versatile, but have a greater learning curve.



The UNIX file system

The UNIX file system, like most modern file systems, provides users a layer of abstraction over the data contained in storage mediums such as hard disks. In particular, file systems allow us to think in terms of *files* and *directories*. Of special interest in the UNIX file system is the *home directory*, which is where we will be able to place our files and work with them. For now, we will be unconcerned with other directories in the file system.

The UNIX shell

As mentioned above, the shell will allow us to interact with the system through the use of commands which you must type in using the keyboard. When we bring up a shell (either because we are directly using a pure command-line interface or because we have called one up from a graphical interface), we will be shown a *prompt* where we can type in a command. In particular, our prompt will show something like this:

```
user@machine:~$
```

For example:

```
borja@classes:~$
```

This denotes that the current user is *borja*, logged into machine *mahogany*, and with the *current directory* being the home directory (the tilde character *~* is short for the home directory). The current directory is an important concept in the shell, as we will generally refer to files *relative* to the current directory (this will be explained shortly).

To start tinkering with the command-line, we can run a simple command called *fortune* that will present us with a fortune message (akin to the ones found in fortune cookies). Simply type *fortune* and press the enter key to run the command.

```
user@machine:~$ fortune
```

You should then see a message printed out. For example:

```
Sometimes a cigar is just a cigar.  
-- Sigmund Freud
```

We can invoke literally hundreds of commands from the command-line. Below is a quick overview of basic commands we need to perform basic actions. For a more complete



text on the command-line, take a look at:

<http://support.uchicago.edu/docs/misc/unix/tutorial/>

You can also get more details on a specific command by reading its *man page*. You can do this using the *man* command:

```
user@machine:~$ man command_name
```

For example:

```
user@machine:~$ man fortune
```

This will show the man page for the *fortune* command. You can navigate through the man page using the scroll keys, and exit pressing the “q” key.

```
FORTUNE(6)                                UNIX Reference Manual                                FORTUNE(6)

NAME
  fortune - print a random, hopefully interesting, adage

SYNOPSIS
  fortune [-acefilosw] [-n length] [ -m pattern] [[n%] file/dir/all]

DESCRIPTION
  When fortune is run with no arguments it prints out a random epigram.
  [...]
```

There is an easy (but somewhat limited) way of finding a command that performs a certain task: searching through all the man page titles using the *apropos* command.

```
user@machine:~$ apropos XSLT
xsltproc (1)          - command line xslt processor
```

Creating new directories

We can create new directories using the *mkdir* command. For example, suppose we want to create a *cmssc16200* directory inside our home directory. We will type the following command:

```
~$ mkdir cmssc16200
```

- For simplicity, we are omitting the “user@machine” part.
- The *mkdir* command, unlike the *fortune* command, accepts an *argument*. In



particular, the argument tells it exactly what directory to create. In general, arguments are used to pass options to the commands.

Listing files

We can see a listing of files and directories contained in the current directory by running the `ls` command.

```
~$ ls
```

If you started out with a new CS account (and, therefore, an empty home directory) you should see the following single line:

```
cmsc16200
```

The current directory

Previously, we introduced the concept of *current directory*, and said that we will generally refer to files *relative* to the current directory. This is important because there are many commands that require us to specify a file (e.g. when compiling a program, we need to specify what source file we want to compile).

For example, if the current directory is the home directory and the home directory contains a file named `foo.c`, we would refer to it from the command line simply like this:

```
foo.c
```

However, if the file were inside the `cmsc16200` directory, we would refer to it like this:

```
cmsc16200/foo.c
```

Another example: the `ls` command seen above (without any arguments) assumes that we want to see the list of files and directories contained in the *current* directory.

To change the current directory, we need to use the `cd` command, specifying the new current directory. For example, suppose we are in the home directory and want to make `cmsc16200` the current directory. We would run `cd` like so:

```
~$ cd cmsc16200
```

The prompt would change to reflect that the current directory has changed:

```
~/cmsc16200$
```