Lab #2: **Linux Review**

Introduction

This lab is a review of the bash shell in Linux, scripting, and C programming. Go through this material before coming to class on Friday.

Learning Objectives

1. Review the bash shell and basic shell commands.
2. Review the use of a Unix file system and Unix processes
3. Review shell scripts and the C programming language

Basic command shell and file system

For background information, read these articles on bash and the vi editor.

Look over the gnu.org bash site. Download the bash manual and read pages 1-9. The manual goes over bash in great detail; scan the table of contents so you know what topics are in this manual.

File Permissions

A typical text file may have the following permissions:

```
-rw-r--r-- 1 cs429 csis  7 2020-08-23  file1
```

Reading the permissions string left to right, the first '–' indicates that this is a regular text file. Other options are `d` for directory, and `l` for a link (or shortcut). The nine characters after this shows us the permissions for this file: they are in 3 sets of 3 permissions:

```
rw-  r--  r--
```
user groups others

The first set of three are permissions for the "user" or owner of the file (`cs429` in the above example). The next set are for the other users who belong to the group assigned to the file (`csis` in the above example). The third set are for the remaining users who are neither the owner and nor in the file's group. Each set of 3 permissions control **r**ead, **w**rite and e**x**ecute permissions: **rwx**. We can do 3 things to an ordinary file:

`r` – read the contents of the file but not change it
`w` – write to (change the contents of) the file
`x` – execute the file if it is an executable file.

Similarly, we can do 3 things with a directory:

`r` – read the contents of the directory (using ls, for example) but not change it
`w` – write to (change the contents of) the directory by creating new files or removing or writing to old ones.
`x` – change into the directory.

In the above example, the owner can read and write to this file but not execute it; users in the same group can only read this file and everybody else can only read it as well.
If we use the command **`ls -l`**, we may see an entry that looks like:

```
drwxr-xr-- 2 cs429 csis 4096 2020-08-23 Test1
```

the `d` indicates that `Test1` is a directory (a folder). Some execute permissions for this file are set and for directories, what this means is that a user can cd to the directory. In this specific example, the owner and users in her/his group can cd to the directory.

The **`chmod`** command changes access permissions on a file. Take a look at the man page for this command first. Some examples:

`chmod g=rw file` gives all users in our group read & write permissions only

`chmod g+w file` turns on write permission for users in our group and leaves the other permissions as they were.

`chmod ug-w file` turns off write permission for us and our group.

For more information on file permissions, read this article, look at the man page (online or using the **man** command – **man chmod** - in a terminal window) for **chmod**. Make sure that you know what a string like **-rw-r-xr--** means. Also make sure you know how to make a directory that you can

- `cd` into but not be able to list the contents of the directory
- write files to but not actually list the contents of.

Wildcards in file names:

Most Unix/Linux shells allow pattern matching to, for example, list all files with names that start with a 'pc' or have an 'mus' somewhere in the file name.

The most commonly used wildcard character is '*'. For example,

```
ls -l pc*
```

lists all files that start with a 'pc' and

```
ls -l *chat*
```

lists all files that have the sequence `chat` anywhere in their name.

Another wildcard character is '?' which is used to match just one character. For example,

```
ls -l pc201?d
```

would find `pc2010d` and `pc2019d` but not `pc2020d`.

A third pattern-matching method uses the square brackets: '[...]'. For example,

```
ls -l pc201[789]d
```

would find `pc2017d` and `pc2019d` but not `pc2016d`

For more information on wildcard characters in file names, see this short article (you can find a list of wildcard characters here). Make sure you know what globbing is.

Unix command input and output redirection

Try the following:

```
cal 9  2020 > s2020
cal 10 2020 > o2020
cal 11 2020 > n2020
cat s2020 o2020 n2020
cat s2020 o2020 n2020 > fall2020
cal 12 2020 >> fall2020
date
date;  cat s2020
date;  cat s2020 > agenda1
(date; cat s2020) > agenda2
```

Make sure you know the difference between the last two commands. Use the **diff** and **sdiff** commands to look at differences between any two files. Make the Terminal window wide enough to see the sdiff output clearly and look at the differences between `agenda1` and `agenda2`.

When we used the "**>**" symbol, we were redirecting the output of the command (on the left) into the file (on right). For example, enter this `cat` command and the following inputs:

```
cat > testf
q
w
```

```
e
r
<Ctrl>d
```

redirects the output of the cat command into a file called `testf`. Run the following commands:

- `sort testf`
- `grep q testf`
- `grep e testf`
- `grep qw testf`
- `who`
- `wc testf`
- `wc -c testf`
- `wc -l testf`

Use **`man sort`** to see the man page for the **`sort`** command and then look at the man pages for **`grep`**, **`who`** and **`wc`** to see the basic functions of these commands.

Just as the ">" symbol is used to put the output of one command into a file, Unix has a way to take the output of one command and use that as the input to a second command:

- `cat testf | sort`
- `cat testf | wc -l`
- `cat testf | grep w`

These commands use a “pipe” (the key for the "|" character is above the “Enter” button on most keyboards). The above 3 commands are longer than the previous versions in section 5 but the real use of the pipe operator is in commands like:

- `ls -l | grep tes`
- `who | wc -l`
- `ls /dev | more`
- `ls /dev | tail`

For example: make sure you know what the following commands are checking for:

- `cal 2 1900 | grep 29`
- `cal 2 2000 | grep 29`
- `cal 2 2020 | grep 29`

In the last three commands, we only see the final output since the output of the `cal` command is used as input for the `grep` command. If we wanted to save the output of `cal` we could use the **`tee`** command to take an input and send it to two places – the “terminal” window and a file :

- `ls` *(make sure there isn't a file named `feb1900` in the current directory)*
- `cal 2 1900 | tee feb1900 | grep 29`
- `ls`

Unix Processes

Take a look at the man page for the “**`sleep`**” command and try the following:

```
sleep 1
sleep 5
sleep 10
sleep 300
```

This last one will take some time to return and we can stop the command by using <Control>-C.
Take a look at the man page for "**`echo`**" and try:

```
(sleep 10; echo OK, I am up)
(sleep 10; echo OK, I am up) &
```

Make sure you understand what this last command did.

Use a longer sleep interval to keep the **process running in the background** while we type in commands at the shell prompt:

```
(sleep 100; echo OK, I am up)
 <Ctrl>z
ps
jobs
(sleep 120; echo OK, I am up)
 <Ctrl>z
jobs
bg %2
jobs
bg %1
jobs
```

Look at the man page for the **`kill`** command. **`ps`** tells us the process ID of the sleep program. Use the **`kill`** command to terminate the sleep program. We may have to hit return a few times after the kill to see some output.

> Remember that a **process** is an instance of a running program. The commands we type in at the bash shell prompt are programs. When we run a program like **`ls`**, the bash shell creates a process for the **`ls`** program to run in and suspends itself. If there were five users on a particular Linux system and all 5 ran the ls command, there would be 5 processes (not just 1) each one being an instance of the **`ls`** program.
>
> If we were running a program that took a long time, say more than an hour, and we had other commands we want to run, we may not want to wait until the long program was done to run the next one. In such a situation, we can run a program "in the *background*". One way to do this is to use the ampersand (**&**) character after the command to indicate to bash that we want to run the program in the background and that bash should give us a prompt so that we can enter the next command.
>
> If we have already started the program running, we can "suspend" the running process by using the ***<Ctrl>z*** character; at that point the program is suspended (paused, not running) but we get a bash prompt so that we can enter commands. We can use the **`bg`** command at this point to tell bash to will then continue running (un-pause) the program but **in the background**. Make sure you know how to have programs running in the background and why this is useful.

For more information on Unix processes, read this article and look up the man pages for the **`sleep`**, **`echo`**, **`ps`**, and **`kill`** commands. Search the **`bash`** man page for information on the "built-in" commands: `jobs`, `bg`, and `fg`.

The `vi` editor

A text editor allows us to edit files containing text data, shell scripts, C or Java code, or any other raw text. It does not allow us to control the appearance of letters as a word processor does. There are many Unix text editors but the most common one is **`vi`** – found on most Linux, BSD, and Unix systems.

The vi editor allows us to set up simple text files, edit shell scripts and C programs. Try out **`vi`** with your Ubuntu VM. The improved vi, vim, is better and once you install it using the command:

```
sudo apt install vim vim-runtime
```

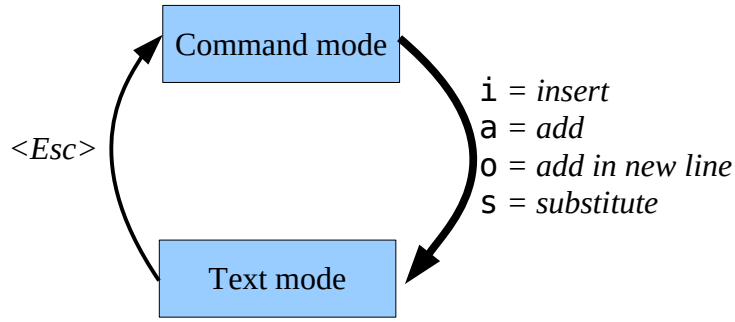
the command `vi` will refer to `vim`.

Open a Terminal Window: Application → Accessories → Terminal.

In the terminal window, start `vi` by typing the command **`vi`** or **`vi filename`** at the bash command prompt. To quit `vi`, we have to save any unsaved data to a file and then enter **`:q`** or **`:wq`** to quit.

```
cs429@Ubuntu$ vi file1
. . .
:wq
cs429@Ubuntu$
```

When vi starts, it is in "command mode"; *i.e.*, characters we enter from the keyboard are interpreted as commands, not as things that go into the file.
For example, in command mode, the <Return> key does not generate a new line of text in the file, it moves us down to the next line. The Space bar in command mode moves us one character to the right. To get into "text mode" so that keyboard input appears in the file, we have to use an appropriate command first: **`i`**, or **`a`** (insert or append) are the most commonly used ones.
These commands put us into the "text" mode; we can get back to command mode by using the **<Esc>** key. There is no menu or any obvious way to choose commands.
We have to "remember" the commands.



1. Use the Unix command "**`cal`**" to generate a text file:
```
cal 9 2020 > s2020
ls -l
cat s2020
```
2. Edit the `s2020` file generated in the previous step using the command **`vi s2020`**
   Insert the following text at the top:
   Your Name here
   CSIS 429
   Lab #2

3. Save the file as "`lab2`" using the command "`:w lab2`". Quit vi and note the size of `lab2`.

Besides commands to go into text mode, there are commands to
move around in the file: `h`, `j`, `k`, and `l` or the arrow keys.
delete a character: `x`
delete (cut) a line: `dd`
yank (copy) a line: `yy`
paste: `p`
We can also copy to and paste from various buffers. The command `"x4yy` will yank (copy) 4 lines into the buffer named 'x'. To past these four lines, use: `"xp`.
More commands:
join two lines: `J`
undo: `u`
set line numbers: **`:set nu`**

If, at some point, a file you are attempting to edit is beyond repair, you can quit without saving by using the command "`:q!`".

More commands:

to go to a particular (n^{th}) line: `:n`<*Enter*>

for example, to go to line 8, use `:8`<*Enter*>

forward search: `/searchtext/`

backward search: `?searchtext?`

search again: **`n`**, **`/`**<***Enter***>** or **`?`**<***Enter***>**

search & replace first occurrence on current line: **`:s/`***`existingtext/newtext/`*

search & replace all occurrences on current line: **`:s/`***`existingtext/newtext/`***`g`**

search & replace all occurrences in entire file: **`:%s/`***`existingtext/newtext/`***`g`**

Bash shell scripts

Shell scripts allow us to set up programs that use other programs.

The Wikipedia article on shell scripts has a good overview and an example script that uses a program called "`convert`" which converts a single jpeg file to a png file. `convert` and the script that uses it are good examples of the Unix philosophy: "**Write programs that do one thing and do it well**":

> Suppose we had 100s or 1000s of jpeg files that we had to convert to the png format. We could write a complicated (as in bloated) `convertAll` program that reads entire directories full of jpeg files to convert.
>
> The Unix philosophy would tell us that we should write a simple C program (`convert`) that takes a single file as input and creates a png output file. Instead of creating a complicated convertAll program, we can use a script like this one to set up a loop in which we use the simple program to convert each file in a directory.
>
> Adding a feature like reporting on the sizes of the jpeg files vs. the corresponding png files is easy to do in a script and gives the user added value. In such a situation, two simple programs are better, faster to develop (and more flexible) than one big complicated one.

The bash manual goes over bash in detail and is also a good reference for scripting.

Boot up your Ubuntu VM and open a Terminal window. **`cd`** to the `/etc/init.d` directory. List the files there. When the operating system boots up, it looks in this directory for many of the services it needs to run, for example, sound, X11 graphics, networking, setting up peripheral devices, etc. Most of the files here are scripts. For example, take a look at the file `acpid`. This is a shell script and while it may be hard to understand everything this file does, you can get some idea of what it does. Most of the other files are **`sh`** (or `dash`) scripts.

Now **`cd`** to the `/etc` directory and use the command **`file rc*`** to see what these `rc` things are: directories. **`cd`** to some of these directories and what we will see is that these directories are populated with scripts that control various parts of the operating system running "above" the kernel.

Now, **`cd`** to your home directory and set up a simple script file called "simple" using:

```
cat > simple
x=Falcon
echo $x $HOME
<Ctrl>-d
```

Run the script file using the command:

```
bash simple
```

Next, make a copy of `simple` called `second` and run second using the command:

```
second
```

Make sure you understand what happened when you did this.

Spell out to bash that we want to run the program in the *current directory* (`.`):

```
./second
```

Make sure you understand what happened this time. See if you can figure out how to run the `second` script. Hint: see what `ls -l` tells us.

Make a copy of `second`, called `third`, with the following line of text at the top of the "`third`" file:

```
#!/bin/bash
```

Make sure you can run the "`third`" script as well. Try these commands and scripts:

1. Find out what shell we are using by entering the following commands:

```
echo $SHELL
echo $0
```

Any other shells available? Try

```
dash
echo $0
csh

sh
echo $0
```

Exit these shells using the exit command until you are back in your bash shell.

2. Every shell has a number of environment variables. In most shells, we can use the **`set`**, **`env`**, or **`setenv`** command to look at the list of environment variables. We can also use the **`echo`** command to look at the values of individual shell variables:

```
echo $SHELL
echo $HOME
echo $HOSTNAME
env
```

3. Get back to the original bash shell. Look up the man pages for the commands "**`tr`**" and "**`cut`**" to see what the following commands do:

```
who
who | tr "cs" "AA"
ls -l
ls -l | cut -c5-7,11-13
```

4. Create text files named "`myfile`", "`myfile1`", "`myfile2`", and "`myfile3`" and enter some text into each, the more the better; for example, we can enter all the text in this page into the file by copying and pasting. Look up the man pages for the commands **"`grep`"**, **"`grep`"** and **"`sort`"** and see what the following commands do:

```
echo myfile?
echo These are my files: myfile*
echo Two possiblities: myfile[23]

grep can myfile
grep -n can myfile
grep -l can my*

sort myfile
sort -r myfile
```

```
ln myfile x
ls -ai
ls -i | sort
ls -i | sort | tail -2
```

5. The `bash` profile: when a user logs in, the `bash` shell first loads common system-wide startup files for all users set up by the administrator:

   ```
   /etc/profile
   /etc/bash.bashrc
   ```

   and then start-up files in the user's home directory:

   ```
   .bash_profile
   .profile
   ```

   if they exist and finally

   ```
   .bashrc
   ```

   also in the user's home directory.
   Interactive sub-shells execute the commands in `.bashrc`. As a result there are many shell variable set in different start-up files.
   The commands in `.bash_profile` are executed once when we log in. Edit this file (create it if it does not exist) and add the following command at the end of the file:

   ```
   echo "Starting up bash ..."
   ```

   Now, run this file by using the command:

   ```
   source .bash_profile
   ```

   or the dot command is equivalent to the source built-in command. Both load the named script file.

   ```
   . .bash_profile
   ```

   What happens?

   The shell variable `PS1` is the shell prompt and is usually "`bash 4.3>`" or the username and machine name. Change the prompt by entering the assignment operation:

   ```
   PS1="Now what? >"
   ```

   What happens?

   We can change the prompt for all bash shells by putting the following lines into our `.bash_profile`:

   ```
   PS1="Now what? >"
   export PS1
   ```

   and then sourcing the startup file as before.

6. Newlines are sometimes not desirable. Take a look at these commands:

   ```
   date
   echo Today is
   date
   echo Today is ; date
   echo -n Today is ; date
   echo -n Today is " " ; date
   ```

   In a script, we may need to have a command or series of commands on separate lines. If we need to enter a long command that does not fit on a single line, we can use the "back slash" character, "\" to tell bash not to interpret the new line as the end of the command:

```
echo -n Today is " " ; \
 date
```

Write a script with the above two lines and make sure it works.

Scripts with logic

The following steps walk you through creating a series of bash shell scripts.

1. Create a script file using the cat command and enter the following into the script file:

```
#!/bin/bash -v
x=Falcon
echo $x $HOME
echo $0 $1 $2 $3 $4 $5 $6 $7 $8 $9
```

Note: copying and pasting from this PDF will probably not work – PDFs have formatting characters. Save and run the script file with different numbers of command line arguments and observe the output.

2. Try running the following script, using more than nine arguments at the command line:

```
#!/bin/bash -v
x=Falcon
echo $x $HOME
echo $#
echo $*
ls -l
```

See if you can figure out the meanings of `$#` and `$*`

3. Write a shell script that will take one argument which will be interpreted as the name of a file. Our script should change the permissions of that file to have execute permission and print out a long listing of that file to show that the permissions have been changed.

For example, if the script is called `tn` and is executed as:

```
cs429@Ubuntu$ ./tn myfile
```

the script should print:

```
myfile is now executable
-rwxr--r--   1 tony  tony  7 Aug 27 09:21 myfile
```

We can use this to make other script files executable.

4. Save the following commands in a script file:

```
#!/bin/bash -v
x=3
y=`expr $x + 2`
let z=$y*10
let a=$y-1
let b=$a/$x
let c=$z%$x
```

echo Results are: $x $y $z $a $b $c

and run it. Next, have the script ask the user to enter a number. The commands

```
echo Enter a number
read p
```

will ask the user for a number and store the user's input in the variable, p. Modify the above script to ask the user for a value to store in x rather setting it to 3.

5. Save the following script file:

```
#!/bin/bash -v
echo -n "Enter a number> "
read x
if test $x -lt 0
then
     echo $x is less than 0
else
     echo $x is greater than or equal to 0
fi
```

Modify the above script to also check if the number is equal to 0 (use `-eq`) and print out an appropriate message.

6. The following are operators we can use with the test operator to check files:

`test` *f* command options	returns 0 if file *f*
`-e`	exists
`-f`	is a regular file (not a directory or device file)
`-s`	is not zero size
`-d`	is a directory
`-b`	is a block device (DVD, disk, etc.)
`-c`	is a character device (keyboard or serial interface)
`-p`	is a pipe
`-r`	has user read permission (for the user running `test`)
`-w`	has user write permission
`-x`	has user execute permission
`-O`	user is the owner
`-G`	has the same group-id as user
`-N`	modified since last read
f1 `-nt` *f2*	*f1* newer than *f2*
`f1 -ot f2`	*f1* older than *f2*
`f1 -ef f2`	*f1* and *f2* are hard links to the same actual file

Examples (`&&` separates two statements – the second is executed if the first is true):

```
test -d $1 && echo "$1 is a directory file"
[ -f $1 ]  && echo "$1 is a regular file"
[ $1 -nt $2 ]  && echo "$1 is newer than $2"
```

Use these to write a script file that will accept a file name as a parameter and print a message indicating whether the file is a directory or not.

7. The following can be used to set up a simple loop that will print the numbers 1-20:

```
i=1
while [ $i -le 20 ]
do
     echo $i
     let i=$i+1
done
```

Use this to write a script file that will ask the user for a number and print out the 10 subsequent numbers.

8. The following uses a for loop to list all the files in the current directory with their absolute path. Use it with an if statement to print out the names of all **directories** in the current directory.

```
for file in *
do
        echo "`pwd`/$file"
done
```

9. Modify the following script:

```
#!/bin/bash -v
echo -n "Enter a number> "
read x
if test $x -lt 0
then
     echo $x is less than 0
else
     echo $x is greater than or equal to 0
fi
```

so that it will ask the user for a number and, assuming it is a positive number, print out that number and successive numbers until we get to 0. For example, if the user enters 5, print out

```
5 4 3 2 1 0
```

10. Set up a script file called "**`catthis`**" with the following contents:

```
#!/bin/bash
while read line
do
   echo $line
done < catthis
```

Make sure you understand what it does. In your report, compare it with the `cat` command.

11. In your report, describe what the following script does:

```
#!/bin/bash
read line < $1
echo $line
read line < $1
echo $line
read line < $1
echo $line
```

Modify it so that it will print all the contents of a file with line numbers before each line- just as the command "**`cat -n file`**" would work but without using `cat` in your script. Submit your modified script.

Basic C programming

Look up these articles on the C programming language and the Gnu Compiler Collection which we will use to compile our C language programs.

Check if the compiler is installed on your VM using the command "cc". If you get output saying that the command was not found, you will have to install it. One easy way to get a whole lot of compile tools is to use the command:

```
sudo apt install build-essential
```

Use the `man` command to see what the -c, -g, and -o options mean for the commands `cc` and `gcc`. Find out where the executables are for the `cc` and `gcc` commands on your system – the `which` command is useful for this. If any of these "executables" are links, use the `ls`, `file` and `stat` commands to track down where the actual programs (as opposed to links) are and what the differences are between them.

Use your Ubuntu VM to compile and run the following basic C programs:

1. Type the following text into a file called "`first.c`"

```
#include <stdio.h>
int main(){
   printf( "Hello, World!\n" );
   return 0;
}
```

   Compile this C program using the gnu C compiler:

```
cc first.c
```

   This creates a file called "`a.out`" which is an executable binary file. Make sure you know how to run the executable. Use the `ls -l`, `file` and `stat` commands to find out as much as possible about the source file (`first.c`) and the executable.
   Alternatively, we can compile the program using the -o option:

```
cc -o first first.c
```

   Make sure you know what this command does and how we can run the executable.

2. Type in the following program into a file called `second.c`:

```
#include <stdio.h>
#define START 1
int main(){
   int n,p,q;
   n = START;
   p = n++ * 5;
   q = ++n * 5;
   printf( "   n = %d,\n   p = %d,\n   q = %d\n", n, p, q );
   return 0;
}
```

   See if you can predict what the above program will print. Compile it, create the executable file, `second`, and run the executable.

   Note the sizes of the source file `second.c` and the executable, `second`. Note the output of the command "`file second`" and compare it with its "debug" version below.

3. Read this page on the use of `#include` statements and this article on header files. Find out what `stdio.h` is and where it is in your file system.

4. Although there are no bugs or mistakes in the above program, the output it produces may not be completely obvious to someone reading the program and so next we will use the debugger to step through this code.

   Look at the above program using the Gnu debugger (`gdb`). To do this, use the **`-g`** flag when compiling.

```
cc -g -o second second.c
```

   In your report, note the differences you see in the files produced by using the **`-g`** option and not using it. Note the output of the command "`file second`" now and how is it different from the non-debug version we had earlier.
   Start the debugger by typing in the command:

```
gdb second
```

Once the debugger starts, we will get a prompt that looks like:

```
(gdb)
```

Try entering the following commands

```
list
run
break 8
```

- *this may depend on where the statement* `n = START;` *is in your program.*

```
run
next
print n
display p
next
next
```

Use the debugger to understand how the `second` C program works and why it prints what it does. For more information on gdb, read this article.

Disassembling an executable file

Executable files created from C programs have a lot of information including the machine code that tells the computer what CPU instructions to execute to "run" the program. Machine code is almost impossible for us to read because it is raw binary or hexadecimal data.

1. Compile the `second` program without the debug flag. cat or vi the `second` executable file and you will see a lot of unprintable characters. One program that allows us to convert machine code to readable code is the **`objdump`** program. A simple way to use it would be to run the command:
   objdump -S second

   You have probably observed that we cannot read the executable file easily. A more readable version of machine code is assembly language. We can write an assembly language program using an editor like vi but like C programs, we cannot run assembly code directly.

   Once we have an assembly program, we can convert it to machine code using an "assembler" – a compiler for assembly language. When we use the gcc or g++ compilers, we are converting C or C++ programs to machine code; this machine code can be converted "back" to assembly – a process called "disassembly". The objdump program disassembles a C program executable.

   Pipe the output of the above `objdump` command through the more command to page through the output. Note that this small C program produces a lot of assembly code. You may be able to spot the machine and assembly code for the main method. It may look like this:

```
040052d <main>:
 40052d:   55                    push   %rbp
 40052e:   48 89 e5              mov    %rsp,%rbp
 400531:   48 83 ec 10           sub    $0x10,%rsp
 400535:   c7 45 f4 01 00 00 00  movl   $0x1,-0xc(%rbp)
 40053c:   8b 45 f4              mov    -0xc(%rbp),%eax
 40053f:   8d 50 01              lea    0x1(%rax),%edx
 400542:   89 55 f4              mov    %edx,-0xc(%rbp)
 400545:   89 c2                 mov    %eax,%edx
 400547:   c1 e2 02              shl    $0x2,%edx
 40054a:   01 d0                 add    %edx,%eax
 40054c:   89 45 f8              mov    %eax,-0x8(%rbp)
```

Memory | Machine language | Assembly code

addresses

There are three main columns of output here. The column of data at the left end are the addresses in memory of the machine language instructions. The column in the middle is the machine language code as hexadecimal code (numbers in base 16). The third main column is the assembly language code corresponding to the machine language code.
Some common assembly language instructions you may see in the above steps:

- **mov** – move data from memory to CPU or other way around
- **add** – add two numbers and store the result
- **sub** – subtract two numbers and store the result
- **push** – push a number onto the "stack"

Pipe the output of the above objdump command through the **wc -l** command to see the number of lines in the output – note this number in your report.

2. The executable code we looked at above has a number of sections and many of these sections have to do with how Linux starts up a program so that the machine code can be executed.
All we have in `second.c` is a **main** function and if all we want is to see the code corresponding to the main function, we can use the command `nm second` to see what sections are in this executable. Show the output of this **nm** command in your report.

   The **nm** command tells us at which addresses the **main** function starts and we can see the same memory addresses in the output of the **objdump** command. Assuming our code starts at the hexadecimal address **40052d** (you may see a different number) and we want objdump to disassemble up to the instruction shown above – hexadecimal address **40054c**, we can use the command:

```
objdump -S --start-address=0x40052d --stop-address=0x40054f second
```

   Your addresses will probably be different. What is the address at which your `main` function in `second` starts? Use a command like the one shown above to see the assembly language corresponding to your code and show this in your report.

3. We can get even more information such as the source code corresponding to the assembly language instructions if we compile with the debug flag: `gcc -g -o second second.c`
and then do the objdump: `objdump -Sl second` (that's a lowercase letter "L" after the S in the options for objdump).
   Part of the output should look like this:

```
40052d <main>:
main():
second.c:6
/*
 * second.c
 */
#include <stdio.h>
#define START 1
int main(){
  40052d:       55                      push   %rbp
  40052e:       48 89 e5                mov    %rsp,%rbp
  400531:       48 83 ec 10             sub    $0x10,%rsp
lab61.c:9
   int n,p,q;

   n = START;
  400535:       c7 45 f4 01 00 00 00    movl   $0x1,-0xc(%rbp)
lab61.c:10
   p = n++ * 5;
  40053c:       8b 45 f4                mov    -0xc(%rbp),%eax
```

```
  40053f:      8d 50 01              lea    0x1(%rax),%edx
  400542:      89 55 f4              mov    %edx,-0xc(%rbp)
  400545:      89 c2                 mov    %eax,%edx
  400547:      c1 e2 02              shl    $0x2,%edx
  40054a:      01 d0                 add    %edx,%eax
  40054c:      89 45 f8              mov    %eax,-0x8(%rbp)
lab61.c:11
   q = ++n * 5;
  40054f:      83 45 f4 01           addl   $0x1,-0xc(%rbp)
  400553:      8b 55 f4              mov    -0xc(%rbp),%edx
  400556:      89 d0                 mov    %edx,%eax
  400558:      c1 e0 02              shl    $0x2,%eax
  40055b:      01 d0                 add    %edx,%eax
  40055d:      89 45 fc              mov    %eax,-0x4(%rbp)
```

Note that each C language statement compiles down to many machine language instructions.

4. Disassemble the entire main function in your `second` program and show the command you used. In your report show the options of `objdump` that you used to disassemble only the main function.

Submit a single tar.gz file containing

- your report document
- script files