LFD108x

Linux Tools for Developers

Version 2022-01-19

THE LINUX FOUNDATION | Training & Certification

Version 2022-01-19

Published by:

the **Linux Foundation**
https://www.linuxfoundation.org

Contents

Chapter 1

Essential Command Line Tools

Exercise 1.1: Using find

- In the following we give some examples of things you can do with the **find** command; your task is to experiment with these examples and extend them.
- Find all files under the `/tmp` directory that are newer than an already existing file and give a detailed listing:

  ```
  $ find /tmp -newer /tmp/tstfile -ls
  ```

  where it is assumed you will substitute the name of an existing file for `/tmp/tstfile`.
- Find all files under the `/etc` directory that have a suffix of `.conf`:

  ```
  $ find /etc -name "*.conf"
  ```

- Find all subdirectories under the `/etc` directory

  ```
  $ find /etc -type d
  ```

- Find all backup files on the system (ending in `.bak`) and delete them. (Do not do this if you need any such files!)

  ```
  $ find / -name "*.bak" -exec rm {} ';'
  ```

Exercise 1.2: Using grep

- In the following we give some examples of things you can do with the **grep** command; your task to is to experiment with these examples and extend them.
- Find all entries in `/etc/services` that include the string **ftp**:

  ```
  $ grep ftp /etc/services
  ```

- Restrict to those that use the **tcp** protocol:

  ```
  $ grep ftp /etc/services | grep tcp
  ```

  or to those that do not use the **tcp** protocol, while printing out the line number:

```
$ grep -n ftp /etc/services | grep -v tcp
```

- get all strings that start with **ts** or end with **st**:

```
$ grep '^ts' /etc/services
$ grep 'st$' /etc/services
```

cd

Chapter 2

Text Operations

Exercise 2.1: tee-ing a File

The **tee** utility is very useful for saving a copy of your output while you are watching it generated.

Execute a command such as doing a directory listing of the `/etc` directory:

```
$ ls -l /etc
```

while both saving the output in a file and displaying it at your terminal.

Solution 2.1

```
$ ls -l /etc | tee /tmp/ls-output
$ less /tmp/ls-output
```

Exercise 2.2: Parsing files with awk (and sort and uniq)

Generate a column containing a unique list of all the shells used for users in `/etc/passwd`. You may need to consult the manual page for `/etc/passwd` as in:

```
$ man 5 passwd
```

Which field in `/etc/passwd` holds the account's default shell (user command interpreter)? How do you make a list of **unique** entries (with no repeats).

Solution 2.2

The field in `/etc/passwd` that holds the shell is #7. To display the field holding the shell in `/etc/passwd` using **awk** and produce a unique list:

```
$ awk -F: '{print $7}' /etc/passwd | sort -u
```

or

```
$ awk -F: '{print $7}' /etc/passwd | sort | uniq
```

For example:

```
$ awk -F: '{print $7}' /etc/passwd | sort -u
```

```
/bin/bash
/bin/sync
/sbin/halt
/sbin/nologin
/sbin/shutdown
```

Exercise 2.3: Using wc (Word Count)

Find out how many lines, words and characters there are in all files in `/var/log` that have the `.log` extension.

Solution 2.3

```
$ sudo wc /var/log/*.log
```

```
    376    2675   21570 /var/log/boot.log
    805    4988  126977 /var/log/dnf.librepo.log
   8839   71774  760826 /var/log/dnf.log
   7164   32605  516106 /var/log/dnf.rpm.log
      1       6      60 /var/log/hawkey.log
     30     265    3195 /var/log/kdump.log
      3      15     102 /var/log/vbox-setup.log
    544    5836   42262 /var/log/Xorg.9.log
  17762  118164 1471098 total
```

Note you have do this with **sudo** to get every file counted; try without it!

Exercise 2.4: Searching and substituting with sed

Search for all instances of the user command interpreter (shell) equal to `/sbin/nologin` in `/etc/passwd` and replace them with `/bin/bash` using **sed**. **Do not** overwrite `/etc/passwd`!

Solution 2.4

To get output on standard out (terminal screen):

```
$ sed  s/'\/sbin\/nologin'/'\/bin\/bash'/g /etc/passwd
```

or to direct to a file:

```
$ sed  s/'\/sbin\/nologin'/'\/bin\/bash'/g /etc/passwd > passwd_new
```

Note this is kind of painful and obscure because we are trying to use the forward slash (/) as both a string and a delimiter between fields. One can do instead:

```
$ sed s:'/sbin/nologin':'/bin/bash':g /etc/passwd
```

where we have used the colon (`:`) as the delimiter instead. (You are free to choose your delimiting character!) In fact when doing this we don't even need the single quotes:

```
$ sed s:/sbin/nologin:/bin/bash:g /etc/passwd
```

works just fine

Exercise 2.5: Simple Searching with grep

Search for your username in file `/etc/passwd`.

Solution 2.5

```
$ grep your-username /etc/passwd
```

Exercise 2.6: Using grep in More Detail

In the following we give some examples of things you can do with the **grep** command; your task is to experiment with these examples and extend them.

1. Find all entries in `/etc/services` that include the string **ftp**:
2. Restrict to those that use the **tcp** protocol:
3. Now restrict to those that do not use the **tcp** protocol, while printing out the line number:
4. get all strings that start with **ts** or end with **st**:

Solution 2.6

1.
```
$ grep ftp /etc/services
```
2.
```
$ grep ftp /etc/services | grep tcp
```
3.
```
$ grep -n ftp /etc/services | grep -v tcp
```
4.
```
$ grep ^ts /etc/services
$ grep st$ /etc/services
```

Note: Cutting and pasting the `"^"` character from a pdf is unlikely to work properly. Please type it.

Chapter 3

Bash Scripting

Exercise 3.1: Creating Simple Bash Shell Scripts

- In the Labs directory (or anywhere you choose), create the following file in your favorite text editor and call it `nproc.sh`:

nproc.sh

```
#!/bin/sh
nproc=$(ps | wc -l)
echo "You are running $nproc processes"
exit 0
```

In this example, we have used the following commands:

  – **wc** counts the number of lines, words, and characters in a file
  – **ps** gives information about running processes

Now make it executable with:

```
$ chmod +x nproc
```

and then run it with

```
$ ./nproc
```

- Type **ps** at the command line. You will notice there is one extra line of headings; thus we have over-counted by one. Edit the script to correct this.

  (**Hint:** you can use either of these two forms:)

```
nproc=$(($nproc - 1 ))
nproc=$(expr $nproc - 1)
```

- Can you do this exercise without using any variables (i.e., do it in one line?)

  (**Hint:** you will find it easier with the `$(...)` construct than with the `` `....` `` construct.)

Solution 3.1

Please see SOLUTIONS/s_03/lab1_nproc.sh

Exercise 3.2: A Simple Backup Utility

- Construct a shell script that works as a basic backup utility. It should be invoked as:

```
$ Backup   Source  Target
```

- For each directory under `Source`, a directory should be created under `Target`.
- Each directory in `Target` should get a file named `BACKUP.tar.gz` which contains the compressed contents of the directory.
- You should not need permission to write in the `Source` directory area, but obviously you will need permission to write in `Target`.
- A good way to test it might be to use `/var` as the source.

Please Note

- Functions can be called recursively, but you do not have to do so.
- Some of the utilities and commands you might need are: **tar, gzip, find, pushd, popd, cp, echo, mkdir ...**.

- **EXTRA CREDIT:** try making it an incremental backup; only do those directories which have changed since the last backup. This can be made very complicated, but just consider cases where there are files newer than when the backup was made.

Solution 3.2

Please see SOLUTIONS/s_03/lab_backup_nor.sh
Please see SOLUTIONS/s_03/lab_backup_r.sh

Chapter 4

Networking

Exercise 4.1: Static Configuration of a Network Interface

Please Note

You may have to use a different network interface name than `eth0`. Furthermore, you can most easily do this exercise with **nmtui** or your system's graphical interface. We will present a command line solution, but beware details may not exactly fit your distribution flavor or fashion.

1. Show your current IP address and default route for `eth0`.
2. Bring down `eth0` and reconfigure to use a static address instead of **DCHP**, using the information you just recorded.
3. Bring the interface back up,
4. Make sure your configuration works after a reboot.

You will probably want to restore your configuration when you are done.

Solution 4.1

1.
   ```
   $ ip addr show eth0
   ```
   or
   ```
   $ ifconfig eth0
   ```
2. Assuming the address was `192.168.1.100` :
   ```
   $ sudo ip link set eth0 down
   $ sudo ip addr add 192.168.1.200 dev eth0
   $ sudo ip link set eth0 up
   ```
   or
   ```
   $ sudo ifconfig eth0 down
   $ sudo ifconfig eth0 up 192.168.1.100
   ```

3.
   ```
   $ sudo ip link set eth0 up
   $ sudo dhclient eth0
   ```

or

```
$ sudo ifconfig eth0 up
$ sudo dhclient eth0
```

4. `$ sudo reboot`

Chapter 5

Processes and System Monitoring

Exercise 5.1: Getting Uptime and Load Average

Ascertain how long your system has been up, and also display its load average.

Solution 5.1

A very simple method is just to use the **uptime** utility:

```
$ uptime
```

```
10:26:40 up  3:19,  5 users,  load average: 1.46, 1.40, 1.19
```

A second method is to look at the first line of output from **top**:

```
$ top | head
```

```
top - 10:28:11 up  3:20,  5 users,  load average: 1.93, 1.52, 1.25
Tasks: 313 total,   1 running, 312 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.0 us,  0.3 sy,  0.0 ni, 98.2 id,  0.5 wa,  0.0 hi,  0.0 si,  0.0
KiB Mem : 16284472 total,  6556792 free,  1029760 used,  8697920 buff/cache
KiB Swap:  8290300 total,  8290300 free,        0 used. 10364220 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2615 coop      20   0  504536 186312  65488 S   6.7  1.1   6:28.30 skype-b+
18248 coop      20   0  655804  50816  30884 S   6.7  0.3   0:20.11 emacs
    1 root      20   0  204912   6508   3956 S   0.0  0.0   0:00.92 systemd
```

A third method is to use **w**:

```
 10:30:51 up  3:23,  5 users,  load average: 0.55, 1.11, 1.14
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
coop     :0       :0               07:08   ?xdm?  16:51   0.19s gdm-session-
coop     pts/0    :0               07:09    2:22m  0.12s  0.12s bash
coop     pts/1    :0               07:09    1:37m  0.42s  0.42s bash
coop     pts/2    :0               07:09    0.00s 51.09s  0.00s w
coop     pts/3    :0               07:09   27:08   0.25s  0.25s bash
```

Exercise 5.2: Background and Foreground Jobs

We are going to launch a graphical program from a terminal window, so that one can no longer type in the window. **gedit** is an easy choice but you can substitute any other program that does this.

1. Open **gedit** on a new file as in

   ```
   $ gedit somefile
   ```

   You can no longer type in the terminal window.

2. While your pointer is over the terminal window, hit `CTRL-Z`.

   ```
   ^Z
   ```

   ```
   [3]+  Stopped                 gedit somefile
   ```

   You can no longer type in the **gedit** window.

3. With **jobs -l**, see what processes have been launched from this terminal window:

   ```
   $ jobs -l
   ```

   ```
   [1]  17705 Running                 evince *pdf &
   [2]- 18248 Running                 emacs /tmp/hello.tex &
   [3]+ 19827 Stopped                 gedit somefile
   ```

4. Now put the most recent job (**gedit somefile**) in background:

   ```
   $ bg
   ```

   ```
   [3]+ gedit somefile &
   ```

   Now you should be able to type in the **gedit** window.

5. Put the process in foreground again:

   ```
   $ fg
   gedit somefile
   ```

   Note you can no longer type in the terminal window.

6. To clean up, suspend the process again and then use **kill** to terminate it:

   ```
   ^Z
   ```

   ```
   [3]+  Stopped                 gedit somefile
   ```

   ```
   $ jobs -l
   ```

   ```
   [1]  17705 Running                 evince *pdf &
   [2]- 18248 Running                 emacs /tmp/hello.tex &
   [3]+ 19827 Stopped                 gedit somefile
   ```

   ```
   $ kill -9 19827
   $ jobs -l
   ```

   ```
   [1]  17705 Running                 evince *pdf &
   [2]- 18248 Running                 emacs /tmp/hello.tex &
   [3]+ 19827 Killed                  gedit somefile
   ```

   ```
   $ jobs -l
   ```

   ```
   [1]- 17705 Running                 evince *pdf &
   [2]- 18248 Running                 emacs /tmp/hello.tex &
   ```

Exercise 5.3: Using at for Batch Processing in the Future

Schedule a very simple task to run at a future time from now. This can be as simple as running **ls** or **date** and saving the output. (You can use a time as short as one minute in the future.)

Note that the command will run in the directory from which you schedule it with **at**.

Do this:

1. From a short **bash** script.
2. Interactively

Solution 5.3

1. Create the file `testat.sh` containing:

   ```
   #!/bin/bash
   date > /tmp/datestamp
   ```

   and then make it executable and queue it up with **at**:

   ```
   $ chmod +x testat.sh
   $ at now + 1 minute -f testat.sh
   ```

   You can see if the job is queued up to run with **atq**:

   ```
   $ atq
   ```

   ```
   17          Wed Apr 22 08:55:00 2015 a student
   ```

   Make sure the job actually ran:

   ```
   $ cat /tmp/datestamp
   ```

   ```
   Wed Apr 22 08:55:00 CDT 2015
   ```

   What happens if you take the `>/tmp/datestamp` out of the command? (Hint: type **mail** if not prompted to do so!)

2. Interactively it is basically the same procedure. Just queue up the job with:

   ```
   $ at now + 1 minute
   at> date > /tmp/datestamp
   CTRL-D
   $ atq
   ```

Exercise 5.4: Scheduling a Periodic Task with cron

Set up a **cron** job to do some simple task every day at 10 AM.

Solution 5.4

Create a file named `mycrontab` with the following content:

```
0 10 * * * /tmp/myjob.sh
```

and then create `/tmp/myjob.sh` containing:

SH

/tmp/myjob.sh

```
#!/bin/bash
echo Hello I am running $0 at $(date)
```

and make it executable:

```
$ chmod +x /tmp/myjob.sh
```

Put it in the **crontab** system with:

```
$ crontab mycrontab
```

and verify it was loaded with:

```
$ crontab -l
```

```
0 10 * * * /tmp/myjob.sh
```

```
$ sudo ls -l /var/spool/cron/student
```

```
-rw------- 1 student student 25 Apr 22 09:59 /var/spool/cron/student
```

```
$ sudo cat /var/spool/cron/student
0 10 * * * /tmp/myjob.sh
```

Note if you don't really want this running every day, printing out messages like:

```
Hello I am running /tmp/myjob.sh at Wed Apr 22 10:03:48 CDT 2015
```

and mailing them to you, you can remove it with:

```
$ crontab -r
```

If the machine is not up at 10 AM on a given day, **anacron** will run the job at a suitable time.

Exercise 5.5: Using stress or stress-ng

stress is a **C** language program written by Amos Waterland at the University of Oklahoma, licensed under the **GPL v2**. It is designed to place a configurable amount of stress by generating various kinds of workloads on the system.

stress-ng is essentially an enhanced version of **stress**, which respects its symptoms and options. It is actively maintained: see https://wiki.ubuntu.com/Kernel/Reference/stress-ng

All major distributions should have **stress-ng** in their packaging systems However, for **RHEL**/**CentOS** it needs to be obtained from the EPEL repository. As of this writing there is no package in the EPEL 8 repository, but you can install the one from EPEL 7 without a problem.

Installing from Source

If you are lucky you can install **stress** or **stress-ng** directly from your distribution's packaging system.

Otherwise, the source for **stress-ng** can be obtained using **git** from https://kernel.ubuntu.com/git/cking/stress-ng.git/. (Or you can download a tarball and use that.) To download, compile, and install:

```
$ git clone git://kernel.ubuntu.com/cking/stress-ng.git
$ cd stress-ng
$ make
$ sudo make install
```

Once installed, you can do:

```
$ stress-ng --help
```

for a quick list of options, or

```
$ info stress-ng
```

for more detailed documentation.

As an example, the command:

```
$ stress-ng -c 8 -i 4 -m 6 -t 20s
```

will:

- Fork off 8 CPU-intensive processes, each spinning on a `sqrt()` calculation.
- Fork off 4 I/O-intensive processes, each spinning on `sync()`.
- Fork off 6 memory-intensive processes, each spinning on `malloc()`, allocating 256 MB by default. The size can be changed as in `--vm-bytes 128M`.
- Run the stress test for 20 seconds.

After installing **stress-ng**, you may want to start up your system's graphical system monitor, which you can find on your application menu, or run from the command line, which is probably **gnome-system-monitor** or **ksysguard**.

Now begin to put stress on the system. The exact numbers you use will depend on your system's resources, such as the number of CPU's and **RAM** size.

For example, doing

```
$ stress-ng -m 4 -t 20s
```

puts only a memory stressor on the system.

Play with combinations of the switches and see how they impact each other. You may find the **stress-ng** program useful to simulate various high load conditions.

Exercise 5.6: Processes

1. Run **ps** with the options `-ef`. Then run it again with the options `aux`. Note the differences in the output.
2. Run **ps** so that only the process ID, priority, nice value, and the process command line are displayed.
3. Start a new **bash** session by typing **bash** at the command line. Start another **bash** session using the **nice** command but this time giving it a nice value of `10`.
4. Run **ps** as in step 2 to note the differences in priority and nice values. Note the process ID of the two **bash** sessions.
5. Change the nice value of one of the **bash** sessions to 15 using **renice**. Once again, observe the change in priority and nice values.
6. Run **top** and watch the output as it changes. Hit `q` to stop the program.

Solution 5.6

1.
```
$ ps -ef
$ ps aux
```

2.
```
$ ps -o pid,pri,ni,cmd
```

```
  PID PRI  NI CMD
 2389  19   0 bash
22079  19   0 ps -o pid,pri,ni,cmd
```

(Note: There should be no spaces between parameters.)

3.
```
$ bash
$ nice -n 10 bash
$ ps -o pid,pri,ni,cmd
```
```
 2389  19   0 bash
22115  19   0 bash
22171   9  10 bash
22227   9  10 ps -o pid,pri,ni,cmd
```

4.
```
$ renice 15 -p 22171
$ ps -o pid,pri,ni,cmd
```
```
  PID PRI  NI CMD
 2389  19   0 bash
22115  19   0 bash
22171   4  15 bash
22246   4  15 ps -o pid,pri,ni,cmd
```

5. `$ top`

Exercise 5.7: Monitoring Process States

1. Use **dd** to start a background process which reads from `/dev/urandom` and writes to `/dev/null`.
2. Check the process state. What should it be?
3. Bring the process to the foreground using the **fg** command. Then hit `Ctrl-Z`. What does this do? Look at the process state again, what is it?
4. Run the **jobs** program. What does it tell you?
5. Bring the job back to the foreground, then terminate it using **kill** from another window.

Solution 5.7

1. `$ dd if=/dev/urandom of=/dev/null &`

2. `$ ps -C dd -o pid,cmd,stat`
```
25899 dd if=/dev/urandom of=/dev/ R
```
Should be `S` or `R`.

3.
```
$ fg
$ ^Z
$ ps -C dd -o pid,cmd,stat
```
```
  PID CMD                         STAT
25899 dd if=/dev/urandom of=/dev/ T
```
State should be `T`.

4. Type the **jobs** command. What does it tell you?

```
$ jobs
```
```
[1]+  Stopped                 dd if=/dev/urandom of=/dev/null
```

5. Bring the job back to the foreground, then kill it using the **kill** command from another window.

```
$ fg
$ kill  25899
```

Exercise 5.8: Invoking the OOM Killer

Examine what swap partitions and files are present on your system by examining `/proc/swaps`.

Turn off all swap with the command

```
$ sudo /sbin/swapoff -a
```

Make sure you turn it back on later, when we are done, with

```
$ sudo /sbin/swapon -a
```

Now we are going to put the system under increasing memory pressure. One way to do this is to exploit the **stress-ng** program we installed earlier, running it with arguments such as:

```
$ stress-ng -m 12 -t 10s
```

which would keep 3 GB busy for 10 seconds.

You should see the **OOM** (Out of Memory) killer swoop in and try to kill processes in a struggle to stay alive. You can see what is going on by running **dmesg** or monitoring `/var/log/messages` or `/var/log/syslog`, or through graphical interfaces that expose the system logs.

Who gets clobbered first?

Chapter 6

Files and Filesystems

Exercise 6.1: umask

- Create an empty file with:

```
$ touch afile
$ ls -l afile
```

```
-rw-rw-r-- 1 coop coop 0 Jul 26 12:43 afile
```

  which shows it is created by default with read/write permissions for owner and group, and read for world.

- In fact, the default permissions given when creating a file is actually read/write for owner, group **and** world (0666); it has been modified by the current **umask**

- If you just type **umask** you get the current value:

```
$ umask
```

```
0002
```

  which is the most conventional value set by system administrators for users. This value is combined with the file creation permissions to get the actual result; i.e.,

```
0666 & ~002 = 0664; i.e., rw-rw-r--
```

- Try modifying the **umask** and creating new files and see the resulting permissions as in:

```
$ umask 0022
$ touch afile2
$ umask 0666
$ touch afile3
$ ls -l afile*
```

Exercise 6.2: Using chmod

- While we have shown you have to use **chmod** using the octal digit method of specifying permissions, it is usually preferable to use the symbolic methods that we will show here.

- It is possible to either give the permissions directly, or add or subtract permissions. The syntax is pretty obvious. Some examples (which might be silly):

  ```
  $ chmod u=r,g=w,o=x afile
  $ chmod u=+w,g=-w,o=+rw afile
  $ chmod ug=rwx,o=-rw afile
  ```

  After each step do

  ```
  $ ls -l afile
  ```

  to see how the permissions took, and try some variations.

Exercise 6.3: setuid

- Normally programs are run with the privileges of the user who is executing the program. Occasionally it may make sense to have normal users have expanded capabilities they would not normally have, such as the ability to start or stop a network interface, or edit a file owned by the superuser.
- By setting the **setuid** (**set u**ser **ID**) flag on an executable file one modifies this normal behaviour by given the program the access rights of the **owner** rather than the **user** of the program.
- We should emphasize that this is generally a **bad idea** and is to be avoided in most circumstances. It is often better to write a **daemon** program with lesser privileges for this kind of use.
- Suppose we have the following **C** program (`./writeit.c`) which attempts to overwrite a file in the current directory, `afile`:

writeit.c

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
        int fd, rc;
        char *buffer = "TESTING A WRITE";

        fd = open("./afile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
        rc = write(fd, buffer, strlen(buffer));
        printf("wrote %d bytes\n", rc);
        close(fd);
        exit(EXIT_SUCCESS);
}
```

which your instructor will provide to you, and which can be compiled simply be doing:

```
$ make writeit
```

or

```
$ gcc -o writeit writeit.c
```

- If you try to run this program on a file owned by root you will get the following sequence of events:

```
$ sudo touch afile
$ ./writeit afile
```

```
wrote -1 bytes
```

```
$ sudo ./writeit afile
```

```
wrote 15 bytes
```

Thus the root user was able to overwrite the file it owned, but a normal user could not.

- Observe that changing the owner of **writeit** to root will not help:

```
$ sudo chown root.root writeit
$ ./writeit
```

```
wrote -1 bytes
```

- By setting the **setuid** bit you can make any normal user able to do this:

```
$ sudo chmod +s writeit
$ ./writeit
```

```
wrote 15 bytes
```

- You may be asking, why did not we just write a script to do such an operation, rather than to write and compile an executable program? The reason is that under **Linux** if you change the **setuid** bit on such an executable, it will not do anything unless you actually change the **setuid** bit on the shell, which would be downright stupid.

Solution 6.3

```
Please see SOLUTIONS/s_06/lab_setuid.sh
Please see SOLUTIONS/s_06/lab_write.c
```

Chapter 7

Linux Filesystems

Exercise 7.1: Loopback Filesystems

- **Linux** systems often use **loopback** filesystems, in which a normal file is treated as an entire filesystem image.
- First create an empty file by doing:

  ```
  $ dd if=/dev/zero of=/tmp/part count=500 bs=1M
  ```

  which will create an empty 500 MB file named /tmp/part. You can adjust the size if you are short on space.
- You can then put an **ext4** filesystem on the file by doing:

  ```
  $ mkfs.ext4 /tmp/part
  ```

  which you can then mount by doing:

  ```
  $ mkdir /tmp/mntpart
  $ sudo mount -o loop /tmp/part /tmp/mntpart
  $ df -T
  ```

  ```
  Filesystem     Type   1K-blocks      Used Available Use% Mounted on
  /dev/sda5      ext4    10157148   6238904   3393960  65% /
  ....
  /dev/loop1     ext4      495844     10544    459700   3% /tmp/mntpart
  ```

- Once it is mounted you can create files on it etc, and they will be preserved across remount cycles.
- You can check the filesystem by doing:

  ```
  $ sudo umount /tmp/mntpart
  $ fsck.ext4 -f /tmp/part
  ```

  and get additional information by

  ```
  $ dumpe2fs /tmp/part
  ```

  and change filesystem parameters by doing:

  ```
  $ tune2fs /tmp/part
  ```

  For example you could change the **maximum-mount-count** or **reserved-blocks-count** parameters.

Chapter 8

Compiling, Linking and Libraries

Exercise 8.1: Shared Libraries

- To see what shared libraries **vim** uses do:

  ```
  $ ldd /usr/bin/vim
  ```

  which will show you the full shared library names and their locations on the filesystem.

- You can gain further information about the actual shared memory segments used by starting up an instance of **vim** and making sure you know the **pid** of the process, such as by doing:

  ```
  $ vim &
  ```

  ```
  [1] 25716
  ```

  which says the **pid** is 25716. You can gain this detailed information by doing either

  ```
  $ cat /proc/pid/maps    (i.e., cat /proc/25716/maps)
  ```

  or

  ```
  $ pmap -d 2 pid         (i.e., pmap -d 25716)
  ```

- While there are probably separate executables for **vi** and **vim** on your system (/bin/vi and /usr/bin/vim) you probably have a default **alias** in your command shell that invokes **vim** when you type **vi**, so to get **vi** directly you have to give the full path name. Try doing the above for /bin/vi as well.

Chapter 9

Building RPM and Debian Packages

Exercise 9.1: Lab 1: Building RPM's

On RedHat or SUSE based Systems

If you are on an **RPM** packaging system you should do the this exercise. If you are on an **APT** packaging system, you should do the following one.

- We give you the source files for a trivial `Hello world` application, in the directory `my_app_1.0.0`.
- We also give you a slightly modified version of the source in `my_app_1.0.0_PATCHED`.

Please Note

The source directory includes a `README` file; without it some versions of **RPM** will bail out in error; it is always good practice to have one anyway.

- Construct the patch file, i.e., by doing something like

```
$ diff -Nur my_app_1.0.0 my_app_1.0.0_PATCHED > my_app-1.0.0.patch
```

  Write a `spec` file.

  Construct source and binary **RPM**'s using **rpmbuild**.

- Install and test the binary **rpm** by doing:

```
$ sudo rpm -ivh $HOME/rpmbuild/my_app-1.0.0.x86_64.rpm
```

```
/usr/local/binmyhello
```

  and then remove it with:

```
$ sudo rpm -e my_app
```

- Also try rebuilding from the source package:

```
$ rpmbuild --rebuild  $HOME/rpmbuild/SRPMS/my_app-1.0.0-1.src.rpm
```

and test as before.

- The script `build_rpm.sh` demonstrates one possible way to do the procedure.

Solution 9.1

```
Please see SOLUTIONS/s_09/build_rpm.sh
Please see SOLUTIONS/s_09/my_app-1.0.0
Please see SOLUTIONS/s_09/my_app-1.0.0_PATCHED
Please see SOLUTIONS/s_09/my_app-1.0.0.spec
Please see SOLUTIONS/s_09/nomake.sh
```

Exercise 9.2: Building a Debian Package from Source

On Debian -based Systems

If you are on an **APT** packaging system you should do this exercise. If you are on an **RPM** packaging system, you should do the previous one.

In this exercise we will build a **Debian** package from its upstream source tarball. (Of course if you are on a non-**Debian** based system you can not perform this exercise!)

We will use a simple **hello** program package, the source of which is contained in the `SOLUTIONS` tarball you can obtain from https://training.linuxfoundation.org/cm/LFD108x.

Before beginning you may want to make sure you have the necessary utilities installed with:

```
$ sudo apt-get install dh-make fakeroot build-essential
```

The contents are:

```
$ tar xvf myappdebian-1.0.tar.gz
```

```
myappdebian-1.0/
myappdebian-1.0/Makefile
myappdebian-1.0/myhello.c
myappdebian-1.0/README
```

where:

```
$ cat README
```

```
Some very informative information should go in here :)
```

```
$ cat myhello.c
```

myhello.c

```c
#include <stdio.h>
#include <stdlib.h>
char hello_string[] = "hello world";
int main ()
{
        printf ("\n%s\n\n", hello_string);
        exit (EXIT_SUCCESS);
}
```

```
$ cat Makefile
```

```
BIN = $(DESTDIR)/usr/bin
CFLAGS :=       -O $(CFLAGS)
TARGET=         myhello
SRCFILES=       myhello.c

all: $(TARGET)

$(TARGET):      $(SRCFILES)
        $(CC) $(CFLAGS) -o $(TARGET) $(SRCFILES)

install: $(TARGET)
        install -d $(BIN)
        install $(TARGET) $(BIN)

clean:
        rm -f $(TARGET)
```

You can certainly construct a simpler `Makefile`. However, you must have the line:

```
BIN = $(DESTDIR)/usr/bin
```

and point the installation to the `BIN` directory, or the executable will not be installed as part of the package.

The main steps in the following are encapsulated in the `lab_makedeb.sh` which is also in the `SOLUTIONS` tarball for this class section:

1. Make a working directory and put a copy of the source tarball in there:

   ```
   $ rm -rf  WORK && mkdir WORK && cd WORK
   $ cp ../myappdebian-1.0.tar.gz .
   ```

2. Expand the source tarball:

   ```
   $ tar xvf myappdebian-1.0.tar.gz
   ```

3. Go into the expanded directory and build the package, using **dh_make**, one of several possible package builder programs:

   ```
   $ cd myappdebian-1.0
   $ dh_make -f ../*myappdebian-1.0.tar.gz
   $ dpkg-buildpackage -uc -us
   ```

4. Make sure the program works!

   ```
   $ ./myhello
   ```

   ```
   hello world
   ```

   Verify its contents:

   ```
   $ dpkg --contents ../*.deb
   ```

5. Take a good look at all the files in the `debian` directory and try to imagine building them all by hand!

6. Install the package:

```
$ cd ..
$ sudo dpkg --install *.deb
```

Verify the installation worked:

```
$ myhello
```

```
hello world
```

You can uninstall the package with:

```
$ sudo dpkg --remove myappdebian
```

Chapter 10

Printing

Exercise 10.1: The CUPS Web interface

1. Bring up http://localhost:631 in your favorite browser.
2. In the first column (**CUPS for Users**), you will find a wealth of documentation. Spending some time on it will be profitable.
3. In the second column (**CUPS for Administrators**), you will find a lot of configuration information. You can do things like add new printers etc (once you supply the administrator password etc.)
4. The third column is **CUPS for Developers** and is not particularly relevant for this class.

Exercise 10.2: Creating PostScript and PDF from Text Files

1. Check to see if the **enscript** package has been installed on your system, and if not install it.
2. Using **enscript**, convert the text file `/var/dmesg` to **PostScript** format and name the result `/tmp/dmesg.ps`. (As an alternative you can use any large text file on our system.)

   Make sure you can read the **PostScript** file (with **evince** for example) and compare to the original file.

   (Note: on some systems, **evince** may have problems with the PostScript file, but the PDF file you produce from it will be fine for viewing.)
3. Convert the **PostScript** document to **PDF** format, using **ps2pdf**.

   Make sure you can read the resulting **PDF** file. Does it look identical to the **PostScript** version?
4. Is there a way you can go straight to the **PDF** file without producing a **PostScript** file on the disk along the way?
5. Using **pdfinfo**, determine what is the **PDF** version used to encode the file, the number of pages, the page size, and other metadata about the file. (If you don't have **pdfinfo** you probably need to install the **poppler-utils** package.

Solution 10.2

1. Try

   ```
   $ which enscript
   ```

   ```
   /usr/bin/enscript
   ```

If you do not get a positive result, install with whichever command is appropriate for your **Linux** distribution:

```
$ [ apt-get | dnf | yum | zypper ] install enscript
```

2.
```
$ enscript -p /tmp/dmesg.ps /var/log/dmesg
$ evince /tmp/dmesg.ps
```

3.
```
$ ps2pdf /tmp/dmesg.ps
$ ls -lh /var/log/dmesg /tmp/dmesg.ps /tmp/dmesg.pdf
```

```
-rw-rw-r-- 1 coop coop 28K Apr 22 13:00 /tmp/dmesg.pdf
-rw-rw-r-- 1 coop coop 80K Apr 22 12:59 /tmp/dmesg.ps
-rw-r--r-- 1 root root 53K Apr 22 11:48 /var/log/dmesg
```

```
$ evince /tmp/dmesg.ps /tmp/dmesg.pdf
```

Note the difference in sizes. **PostScript** files tend to be large, while **PDF** is a compressed format.

4. You may want to scan the **man** pages for **enscript** and **ps2pdf** to figure out how to use standard input or standard output instead of files.

```
$ enscript -p - /var/log/dmesg  | ps2pdf -  dmesg_direct.pdf
```

```
[ 15 pages * 1 copy ] left in -
85 lines were wrapped
```

```
$ ls -l dmesg*pdf
```

```
-rw-rw-r-- 1 coop coop 28177 Apr 22 13:20 dmesg_direct.pdf
-rw-rw-r-- 1 coop coop 28177 Apr 22 13:00 dmesg.pdf
```

5.
```
c7:/tmp>pdfinfo dmesg.pdf
```

```
Title:          Enscript Output
Author:         Theodore Cleaver
Creator:        GNU Enscript 1.6.6
Producer:       GPL Ghostscript 9.07
CreationDate:   Wed Apr 22 13:00:26 2015
ModDate:        Wed Apr 22 13:00:26 2015
Tagged:         no
Form:           none
Pages:          15
Encrypted:      no
Page size:      612 x 792 pts (letter)
Page rot:       0
File size:      28177 bytes
```

Exercise 10.3: Combining PDFs

1. Given two text files (you can create them or use ones that already exist since this is non-destructive), convert them into **PDF**s.

2. Combine these two files into one **PDF** file.

   Depending on what software you have employ the following methods:

   (a) **qpdf**
   (b) **pdftk**
   (c) **gs**

3. View the result and compare to the originals.

✓ Solution 10.3

1. First, create two **PDF**s to play with, using **enscript** and then **ps2pdf**:

```
$ enscript -p dmesg.ps /var/log/dmesg
$ enscript -p boot.ps /var/log/boot.log
$ ps2pdf dmesg.ps
$ ps2pdf boot.ps
```

2. Combining files:

   (a) Method 1: Using **qpdf**

   ```
   $ qpdf --empty --pages dmesg.pdf boot.pdf -- method1.pdf
   ```

   (b) Method 2: Using **pdftk**

   ```
   $ pdftk dmesg.pdf boot.pdf cat output method1.pdf
   ```

   (c) Method 3: Using **gs**

   ```
   $ gs -dBATCH -dNOPAUSE -q -sDEVICE=pdfwrite -sOutputFile=method2.pdf dmesg.pdf boot.pdf
   ```

3. Now view them:

```
$ ls -l method?.pdf
$ evince method?.pdf
```

How do the files compare? In appearance? In size?

(Note the use of wildcards in `method`.pdf? .)