

Learn Unix Fast (L.U.F)

1 Some basics

The purpose of this handout is to introduce you to the nuisances of the Unix operating system. The operating system of a computer is what Administration is to Caltech, only operating systems usually do a better job. The operating system is responsible for administrating the processes that run on CPU (which involves handling memory usage, interfacing interprocess communication, exterminating processes, swapping, etc...), interfacing processes with external devices like the filesystem, audio devices, terminal devices, etc.. For networked machines, like the ones we use, the operating system is also responsible for networking. (Making and responding to connections from other machines, delivering email and so on.). In short the operating system is what makes the machine “alive”.

The most basic thing about any operating system, and UNIX in particular is the *shell*. The shell is a program that prompts the user for commands, accepts them, parses them and calls the appropriate programs/processes to carry out the order. With the exception of a very limited number of “incantations” most things you type on the prompt are the names of programs that will be loaded from the hard disk and run. For instance if you type

```
gluttony-50: who
jalano      pty/ttyp0   Sep 30 15:26
napalm     pty/ttyp1   Sep 29 01:13
napalm     pty/ttyp2   Sep 29 14:44
maxim      pty/ttyp3   Sep 30 15:14
garylfay   pty/ttyp4   Sep 30 15:00
et         pty/ttyp6   Sep 28 11:15
lf         pty/ttyp7   Sep 30 15:03
```

your shell will call a program called “who” which will tell who is logged in on the same machine as you are.

The most elementary Unix commands are:

- `cat <filename> <filename> ...`

The catenate command opens files listed, reads their contents and spits the output on the terminal. The files are opened in the order by which they are listed in the commandline.

- `mkdir <directory-name>`

This command creates a new directory. A directory in Unix is a *special* file which points to a list of files, some of which can be directories themselves.

- `cd <directory-name>`

This command enters a directory that’s listed under the current directory and makes that your current directory. When you first log in, your current directory can be referenced as `/home/username` where `username` is your username. This is your *home directory*. If you get lost, to return to your home directory, you simply say:

```
gluttony-36: cd
```

You can also obtain your current directory with the `pwd` command:

```
gluttony-37: pwd
/net/envy/root/mnt/scsi_2/ufs/lf
```

The gibberish in the output have to do with the specific drive that my home directory is located. `/home/lf` is a *link* which pretends to be my home directory. When you try to enter it, it sends you off to the real location of my home directory. The current directory can be addressed by a single dot.

```
gluttony-51: cd .
```

will make you enter the current directory, which is stupid because you are already there. The directory up one level can be addressed by a double dot:

```
gluttony-65: pwd
/net/envy/root/mnt/scsi_2/ufs/lf/tex
gluttony-66: cd ..
gluttony-67: pwd
/net/envy/root/mnt/scsi_2/ufs/lf
```

- `mv <filename1> <filename1>`
`mv <filename1> <filename2> ... <directory-name>`
The first incantation renames file 1 to file 2. The second incantation will move the files listed under the directory, listed at the end. Note that because Unix treats directories as special files, you can also move entire directories under a directory.
- `cp <filename1> <filename1>`
`cp <filename1> <filename2> ... <directory-name>`
Same as the `mv` command. The difference is that the files are duplicated rather than “moved” with a different name, or a different directory.
- `rm <filename1> <filename2> ...`
This command will delete the files listed. It will refuse to remove directories. It is possible to force it to remove directories, but if you find out how to do that **BE CAUTIOUS!** If you happen to delete a file that has been around for a while by accident, there may be a chance to recover it from the periodic backups of the cluster. You would need to talk with a system administrator. Sysadmin’s username is usually `root`. Don’t make deleting files accidentally a habit though. To remove a directory you should enter the directory first and delete everything inside. Then, get out of it and say

```
gluttony-55: rmdir dirname
```

where `dirname` is the directories name. If you really really need to wipe out a big directory tree, and you don’t feel like doing all this fuss, there is an option that forces `rm` to wipe out the dir-tree for you. We will tell you later how to find out this information.

- `ls`

`ls -al`

The `ls` command shows the contents of the current directory. By adding the `-al` flag, it lists the files with more information attached to them.

```
gluttony-72: ls
luf.aux  luf.dvi  luf.log  luf.tex
gluttony-73: ls -al
total 36
drwxr-xr-x  2 lf      users      1024 Sep 30 15:39 ./
drwxr-xr-x 22 lf      users      2048 Sep 30 14:32 ../
-rw-r--r--  1 lf      users         8 Sep 30 19:04 luf.aux
-rw-r--r--  1 lf      users     6360 Sep 30 19:04 luf.dvi
-rw-r--r--  1 lf      users      846 Sep 30 19:04 luf.log
-rw-r--r--  1 lf      users     5308 Sep 30 19:04 luf.tex
```

These are only meant to be a starting point. For more information, you should read the *man pages*. For instance, to find about all the bells and whistles associated with the `rm` command, you should just type:

```
gluttony-76: man rm
```

Usually man pages at the end reference to other commands that have something to do with the command you just looked at. This is a good way for beginners to explore Unix. An even better way is to look for a good book in the bookstore, but familiarity with man pages is essential. For the user's convenience, you can get a list of things you can `man` about a *topic* by using the `-k` flag. For instance

```
gluttony-77: man -k convert
```

will give you a list of stuff that have something to do with conversions. You will be amazed at the abundance of information that there is in the man pages. If you feel adventurous you can try:

- `man csh`

The man page for the `csh` shell.

- `man man`

The man page of the `man` command itself.

- `man sex`

This is not a joke! Some sites actually carry that! ;-)

Finally, there exists a program local to our cluster, which is very cool for reading man pages. Try

```
gluttony-78: tkman&
```

and a fancy viewer with lots of buttons and stuff will pop up. Note that there exist man pages for the C calls. If you try

```
gluttony-89: man printf
```

you will get the man page for `printf`. Notice that sometimes there can be confusion. For instance, there exists a Unix command called `unlink`. and a C function also called `unlink`. Um, in cases like that, you may not get the man page you want if you simply say:

```
gluttony-90: man unlink
```

In cases like that, you do this to go around this obstacle:

```
gluttony-97: man -k unlink
link, unlink (1m)    - exercise link and unlink system calls
unlink (2)           - remove directory entry; delete file
gluttony-98: man 2 unlink
gluttony-99: man 1m unlink
```

2 Some features of the shell

As we said earlier, when you type something on the shell's prompt, the shell will load a program that is called so and run it. However the shell can do more than that for you. An essential feature of almost all shells is *pipes* and *redirection*.

Take a look at the following incantation:

```
gluttony-79: cat luf.tex | more
```

This call involves two programs: `cat` and `more`. What the call does is run the first program and feed it's output as input to the second program. In this case, the second program is `more`, a *pager*. If you just direct the output of `cat` on the screen, the screen will be scrolling for ever until the file is exhausted. If you send it to a *pager* the pager will make sure to wait for the user to press a key between pages. A better pager you can use is `less`

```
gluttony-80: cat luf.tex | less
```

Try them and see which one you like best. Two other useful things you can use with a construction like the above are `tee` and `mail`. `tee` will take input and send it both on the screen and on a file. This is extremely useful for keeping a log of your session with a program. If you say

```
gluttony-81: prog | tee log
```

where `prog` is a program of yours, it will execute fine, but silently the output will be logged on a file called `log`. The command `mail` can be used in this way to send copies of file over *email* to another user. For instance, you can send your homework to your TA by typing:

```
gluttony-82: cat homework | mail evilta@inquisition.caltech.edu
```

or something like that :-). (address may vary). To have a "Subject" show up along with the message, you'd better do something like:

```
gluttony-83: cat extension.req | mail -s "Extension request" evilta@inquisition.caltech.edu
```

The quote marks are essential for forcing the shell to treat the two words “*Extension request*” as one argument. Omitting them will treat the word “request” as an email address and the message will bounce back. Also, a very useful tool to use with pipes is `grep`. `grep` will read its input line by line, and only send to the screen the lines that contain a certain word. For instance,

```
gluttony-83: last |less
```

will list the logs of who logged in when, in the current machine. If you are specifically interested in user `evilta`, you can insert a `grep` filter between `last` and the pager:

```
gluttony-84: last |grep evilta |less
```

You can invert the operation with the `-v` flag.

```
gluttony-85: last |grep -v ftp |grep -v root |less
```

This will list all lines except for the ones that contain the words `ftp` and `root`. If you have lots of free time you can try reading the man pages for `awk`. It allows for more sophisticated filtering and is way cool. A useful tool, for things like hum papers is `wc` (word count) If you have a paper, you can say:

```
gluttony-127: cat luf.tex |wc
539 3989 23969
```

This returns the number of lines, words and characters of `luf.tex`.

At this point I should mention a very important detail. When programs send output, they do it through two channels: `stdout` and `stderr`. `stdout` is where you would normally send output, and that's where output goes with the `printf()` command. `stderr` is where you should send your error messages if the program screws up. When you do piping, as in the above constructs, *only stdout is being sent to the next program*. `stderr` is still being directed to the screen. To have your program send stuff to `stderr` you use the following C command:

```
fprintf(stderr,... );
```

Another term that should be mentioned is `stdin`. `stdin` is a reference to *standard input*. Normally this is the stuff that you type on the keyboard. If the program is part of a pipeline though (like `grep`, `last`, and so on, then `stdin` is taken from the `stdout` of the previous program in the pipeline.

Another feature of shells is *redirection*. As opposed to piping which sends the `stdout` of a process to another process, redirection will send the `stdout` (and the `stderr` if desired) into a file. The syntax for redirection is:

- `prog > filename`
Will send `stdout` of `prog` to file `filename`. It's better that file `filename` doesn't exist when this is executed.
- `prog >& filename`
Will send both `stdout` and `stderr` to the file.

More details about redirection can be found in the man pages of the shell.

Another thing that may be good to know about is the *path*. This is a list of the directories under which the shell will look for the programs it invokes. To see what your current path is, you should say:

```
gluttony-124: echo $path
/usr/ug/bin/TeX /home/lf /usr/ug/bin /usr/gnu/bin /usr/ug/bin/X11 /usr/bin/X11
/usr/local/bin /usr/contrib/bin /usr/bin /bin . /home/lf/bin
```

Do you see that little dot in the middle of all the junk? **This dot is important.** What it says is that the shell should also look in the current directory for a program. Without the dot, if you compile a program and try to run it, the shell won't find it because it won't look in the current directory. (remember that the dot refers to the current directory. If you were wondering what use that is, now you know). If your shell ever refuses to run a program, that could be for one of two reasons: The program is not in the path, which can be checked, or you need to rebuild an internal hash-table thingamajig which you can do with the following incantation:

```
gluttony-125: rehash
```

This is usually necessary, when a new executable is introduced in a directory other than the current directory. It shouldn't be necessary if you just compiled something in the current directory. It shouldn't be necessary next time you log in either. Another little toy is **which**. If you say

```
gluttony-60: which grep
/bin/grep
```

you will get the actual location of the executable **grep**. This, in case you are wondering where all those Unix commands are actually located.

3 Using an Editor

An editor is a program that allows you to create and edit text files. You would normally use an editor to enter the code for a program. Because of capitalism, you shouldn't be surprised that there are more than one editors out there. The most prevalent ones are **vi** and **emacs**. **emacs** is a huge and slow program. It's made of a ELISP interpreter, and a pile of ELISP code which is the actual editor. It's full of bells and whistles that have nothing to do with editing (including an adventure game, the game of life, Eliza, Zippy the pinhead, etc...) and has divided the world into **emacs** fanatics and **emacs** loathers. The advantage of **emacs** is that it has an easy learning curve. You can get used to it right from the beginning. It's disadvantage is that after you use it for a while, you realise that it's slow, cumbersome and annoying. When you log in to the UGCS machines, by default, two windows will pop up. One of them is the shell. The other one is EMACS. All I will explain about the EMACS window is how to get rid of it. Move the mouse inside the window and press the following key sequence: **ctrl-x ctrl-c**. The **vi** editor is smaller and more efficient to use, but it has a harder learning curve. People think it is unintuitive at first, but once you get used to it, you can do your editing with less key-strokes that in **emacs**. Also, the **vi** editor is the native Unix editor. **emacs** is an accessory. You won't find it everywhere. For these reasons, although CS 1 students are normally encouraged to use **emacs** (I mean, what would **you** make out of that silly **emacs** window that pops by itself?), this handout will discuss **vi**.

To start **vi**, you tell the shell:

```
gluttony-666: vi hello.c
```

where **hello.c** is the file you want to edit. If your compiler says that you have an error in line 251, you can invoke the editor as:

gluttony-667: vi +251 hello.c

Once into the vi editor, you can edit. There are two modes in vi: *command* mode and *insert* mode. If you are not sure which mode you are in, pressing *esc* will get you to *command* mode. In *command* mode you can move the cursor around, read/write files, and so on. In *insert* mode you are typing in text.

- **Cursor Moving commands**

- h moves one character to the left
- j moves down one line
- k moves up one line
- l moves one character to the left
- w moves forward one word
- 5w moves forward five words (get the idea?)
- b moves back one word
- G moves to the last line in the file
- 10G moves to the tenth line in the file
- :10 same thing
- \$ moves to the end of the current line
- 0 moves to the beginning of the current line

- **Deletions**

- x Deletes the character under the cursor
- dd Deletes the current line
- 4dd Deletes four lines
- dw Deletes the current word
- 8dw Deletes the next eighth words

- **Other stuff**

- :w Save the file to the disk
- :r *file* Paste a file from the disk at the cursor's current position
- :q Quit only if the file is saved
- :q! Force quit without saving the file. Good when you screw up
- :u Undo the last change.
- /*string* Search for *string*
- n Find the next occurrence of *string*
- N Find the previous occurrence of *string*
- ctrl-L Redraw the screen (for bad terminals)

- **Entering Insert mode**

- a Appends text after the cursor position
- i Inserts text at the current cursor position
- o Inserts a new line below the current line.
- O Inserts a new line above the current line.
- R Replace existing text on a line

- **Once in insert mode**
 Type in text
 esc Exit insert mode and back to command mode

4 Other goodies

Unix is a multi-tasking system. This means that the CPU is capable of running many processes at the same time. To get a list of the processes that you are running, you say:

```
gluttony-90: ps
  PID TTY          TIME COMMAND
 14811 ttyp7        0:00 ps
   9781 ttyp7        0:01 uglogind
   9782 ttyp7        0:00 csh
    814 ttyp7        0:00 badboy
```

The first process on the list is the command we just executed. The second one is a UGCS thingamajig. The third one is the shell, *csh*. PID stands for *Process ID* and it is a number that uniquely references a process. To kill a process, you issue a *kill* command:

```
gluttony-91: kill 814
```

will kill the process *badboy*. Some processes (like the shells) capture the signal that tells them to die and ignore it. To override that feature, use the *-9* flag:

```
gluttony-91: kill -9 9782
```

Killing the shell is generally stupid. The only occurrence when it can be useful is when you were logged in on a lame terminal and the terminal gets confused and you can't log out. In that case, you can go to another terminal, log in and kill the shell of your previous log in. That will log you out.

To get the entire process table type

```
gluttony-92: ps -ef
```

This works on *System V Unix*. On *BSD Unix* (like the CCO machines) you should use the *-aux*. Using *grep*, you can filter out any user's processes.

A very cool feature of Unix is *background processes*. If you merely type

```
gluttony-92: prog
```

the program will run, and you will have to wait for it to finish before you can get your shell prompt back. This is ok for programs that are interactive. If however a program will just pop up a window and interact there, or will just do some computations silently and write the output to a file, and you would like your shell back while you are waiting for the program to finish, you should add an ampersand *&* at the end of your incantation:

```
gluttony-105: xmosaic&
[2] 15010
gluttony-106:
```

The effect of the ampersand is that it makes your program run on the background. The returned number is the PID of the spawned process and you can see that this is so if you say

```
gluttony-106: ps
  PID TTY          TIME COMMAND
 14967 ttyp7        0:00 xdvi
 15051 ttyp7        0:00 ps
   9781 ttyp7        0:01 uglogind
   9782 ttyp7        0:00 csh
 15010 ttyp7        0:01 xmosaic
```

There are some restrictions for programs that run in the background. They can't accept input from `stdin` and they can't write to `stdout` or `stderr`. However, you can use pipes and redirection to make up for the limitations. You can have your program accept input from a file `feed` and redirect its `stdout` and `stdin` to `file` like this:

```
gluttony-107: cat feed | prog >& file&
```

If you are not interested in the output of the program, you can redirect it to the bit bucket: `/dev/null`

```
gluttony-107: cat feed | prog >& /dev/null&
```

An alternative is to have the program's output mailed to you, as soon as the program terminates:

```
gluttony-108: cat feed | prog | mail lf@ugcs.caltech.edu&
```

or you can have the output written in a file, **and** mailed to the user:

```
gluttony-109: cat feed | prog | tee logfile | mail lf@ugcs.caltech.edu&
```

By now, the flexibility offered by those two simple Unix tools (piping and redirecting) should be obvious.

Another subject that you may find interesting is *file permissions*. Permissions determine, who can read, write or execute a file. (for normal files, execute means invoking them from the shell; for directories it means permitting the user to enter them with the `cd` command.) To obtain the permissions of a file, you use the `ls` command as follows:

```
gluttony-111: ls -al luf.tex
-rw-r--r--  1 lf      users      22276 Oct  1 00:32 luf.tex
```

The very first field lists the permissions. The first character (a dash in our example) tells us whether the file is a normal file or a directory. If it is a directory the first letter will be a `d`. The next 3 triplets of characters determine the permissions set up for the owner (`lf`), the group (`users`) and everybody else. Each triplet contains

- *Read bit* set to either `r` or `-`
- *Write bit* set to either `w` or `-`
- *Execute bit* set to either `x` or `-`

To change the permissions of a file, you use the `chmod` command. A detailed description of the command is in the man page. When a file is created, the default permissions is that the owner can read and write but not execute, and everybody else can only read. Compiled files have their default permissions set to read,write and execute by the owner and only read and execute for everyone else. Some users like to create at the top of their home directory a directory called `private` like this:

```
gluttony-116: mkdir private
gluttony-117: chmod go-rwx private
gluttony-118: ls -al |grep private
drwx-----  2 lf      users          24 Oct  1 00:52 private/
```

That way they can put stuff under that directory without worrying about the prying eyes of other users. The permissions for sensitive things like your email, are by default set to readable and writeable only by the owner.

Another toy that can get useful sometimes is the so called symbolic links. You can identify a symbolic link by the first letter in their permission field which is an `l`. A symbolic link is a file pointer that points to another file. When you read a symbolic link, the operating system is actually opening the file pointed to by the symbolic link. Also, when you write to a symbolic link, the operating system opens and writes the pointed file. In other words, you can edit the pointed file, by invoking an editor on it's symbolic link. Finally, symbolic links can be removed just like normal files with the `rm` command, but *that operation will leave the pointed file unaffected*. Notice that as directories are treated as special files by the Unix OS, you can have a symbolic link point to a directory. Then, when you `cd` into the symbolic link, you effectively enter the pointed directory. To create a symbolic link, you use the `ln` command:

```
gluttony-119: ln -s /ug/tmp/elef temp
```

The example command will create a symbolic link called `temp` that points to the temporary storage directory `/ug/tmp/elef`. The `-s` flag tells `ln` to create a *soft* link as opposed a *hard* link. Soft links are better because they have less limitations than hard links, and are a lot safer. Symbolic links are useful on a couple of occasions. For instance, when setting up a web server that needs to access a couple of files, it's best to reference those files through a symbolic link, so that if you are to move the files around, you just change the symbolic links, without hacking the web server all over again. The man page for symbolic links is called `symlink`.