

Taller de Bash

Alejandro Aguayo Ortiz

12 de septiembre de 2023

1. ¿Qué es Bash?

Bash es una Unix Shell. . . ¿Qué es una Unix shell? Es un intérprete de comandos, el cual consiste en la interfaz de usuario tradicional de los sistemas operativos basados en Unix (como Linux o Mac).

Bash es un intérprete de comandos por defecto de la mayoría de las distribuciones GNU/Linux, además de macOS. Es un programa informático cuya función consiste en interpretar órdenes a través de una consola. Es una de las formas más primitivas y útiles de comunicarse con la computadora.

Bash fue creado el 8 de junio de 1989 por Brian Fox y su última versión estable es del 30 de enero de 2018. Es un intérprete programado en C, por lo que hereda muchas de las formas sintácticas y estructurales de este lenguaje.

A groso modo, es un procesador de comandos que corre en una ventana de texto (terminal), donde el usuario escribe comandos que realizan alguna acción. En él se puede:

- Definir variables.
- Recibir argumentos.
- Programar.
- Hacer cálculos matemáticos.
- etc. . .

La ventaja que tiene trabajar en bash es que, mediante instrucciones sencillas, uno puede comunicarse con el núcleo (kernel) y controlar el funcionamiento de la computadora. . . De toda la computadora.

Los usuarios normales (la gente común) crecimos con las computadoras cuando ya eran usadas por el público en general. Para esto se desarrollaron entornos gráficos (como GNOME, creado por el mexicano Miguel de Icaza), con el fin de hacer mucho más amigable el trato entre usuario y computador. Sin embargo, al introducir las ventanas y botones y que todo esté al alcance de un click, se perdió la posibilidad del usuario de poder modificar, controlar y personalizar su computadora, ya que todo está de alguna forma preestablecido.

El objetivo de este taller es, suponiendo un conocimiento básico de comandos de Bash, poder profundizar un poco en los que sean menos conocidos y llevar todo eso un paso adelante: hacer programación en terminal, hacer combinaciones de comandos usando *pipe* (|) y finalmente, crear ejecutables; ésto es de lo más importante y ventajoso que tiene trabajar en bash.

También se darán a conocer una serie de programas de UNIX que son *software-libre* y pueden resultar como alternativa a algunos programas con licencia de macOS o Windows, o que resultan como alternativa al uso de algunas páginas de internet, así como simplemente otros que son curiosos y divertidos.

2. Terminal

La terminal es un entorno sin gráficos que permite únicamente el procesamiento de textos por medio de un teclado. Actualmente, debido a que los entornos gráficos tienen mucho “bum” entre el público en general, se ha permitido a las terminales tener características como el uso del mouse, muchas pestañas, poder mezclarse con el tema de escritorio, etc. Pero su funcionamiento y objetivo siempre es el mismo: ser el traductor entre usuario y sistema.

Al abrir la terminal aparecerá algo de este estilo:

```
usuario@nombre-de-la-compu:~$
```

donde **usuario** se refiere al nombre del usuario en el que está iniciada la sesión, **nombre-de-la-compu** es el nombre de la computadora que, si estamos conectados vía remota equivaldría a la dirección IP. El símbolo **\$** indica que estamos como un usuario normal con privilegios normales, como crear carpetas e instalar algunos programas en nuestra sesión, pero no podemos modificar nada fuera de nuestro usuario.

Todo en la computadora son carpetas, y los usuarios no son la excepción. Las carpetas de usuario, donde se encuentran nuestros directorios **Documentos**, **Imágenes**, **Escritorio**, ..., se encuentran en la carpeta **/home**, que es una entre muchas otras carpetas como **/bin**, **/usr**, **/var**, **/tmp**, **/root**, **/boot**, **/sys**, ... Estas son las carpetas a las que no cualquier usuario tiene acceso, sólo aquel que tenga privilegios de “superusuario”.

Cuando tenemos privilegios de “superusuario”, la terminal aparecerá como:

```
root@nombre-de-la-compu:/home/usuario#
```

y podremos modificar los ejecutables a conveniencia. Esto es un arma de dos filos: si sabes lo que haces, puedes sacarle muchísimo provecho a tu computadora, pero si no tienes idea de qué está pasando, puedes echarla a perder por completo.

3. Comandos

En esta sección se hará una descripción rápida a manera de guía de los comandos más importantes de Bash, así como un ejemplo básico de qué es lo que realiza. Empezando por los más conocidos, hasta los menos utilizados. En estos últimos, y los que tengan más importancia para el taller, se enfatizará un poco más con ejemplos más complejos.

3.1. ¿Qué es un comando y cómo funciona?

Un comando es un conjunto de caracteres que al ser escrito en la terminal, es dirigido a una carpeta específica de la computadora que contiene un archivo ejecutable. Este archivo ejecutable (que puede estar codificado o no) recibe una serie de argumentos y opciones.

Las opciones nos permiten obtener distintos atributos del comando, ya sea un distinto formato de presentación del output del comando, o indicarle que va actuar sobre un directorio/archivo, etc. Los argumentos son, por lo general, archivos, directorios o variables sobre las cuales se va aplicar el comando.

Ejemplos:

- **\$ ls -l**: este comando despliega todos los archivos en la carpeta en la que nos encontremos y la opción **-l**, nos los escribe en forma de lista.
- **\$ libreoffice archivo.doc**: el comando **libreoffice** es un binario que redirige a la aplicación gráfica LibreOffice (la versión *free the Office*) y abre el archivo **archivo.doc**, el cual entra como argumento del binario.

Se puede incluir una lista de argumentos. Por ejemplo si queremos abrir con libreoffice todos los archivos de la carpeta que terminen en “.doc”, basta con teclear `$ libreoffice *.doc &`. El “*” indica “todo los patrones que encuentres antes de “.doc“. Si se pone de arguemnto sólo “*”, el comando trata de acceder a TODOS los ficheros que estén en esa carpeta.

Normalmente se escriben las opciones antes de los argumentos, pero esto puede variar de comando en comando, por lo que es conveniente siempre leer los manuales de ayuda del comando.

3.2. Output de los comandos

El output de los comandos, es decir, lo que arrojan al ingresarlos en la terminal, normalmente es texto que se despliega ahí mismo en la terminal. Muchas veces es conveniente redireccionar estas salidas a archivos.

Ejemplo.

```
$ ls > lista.txt
```

manda el output del comando `ls` a un archivo llamado `lista.txt` que puede o no existir previamente.

3.3. Comandos de ayuda

Para empezar a trabajar con los comandos de Bash, hay que destacar que una ventaja que tienen es que los manuales de operación, así como ayuda y ejemplos de cómo utilizarlos de cada uno de ellos se encuentra al alcance del teclado.

- **man:** Muestra el manual de cualquier comando que le indiquemos.

Ejemplo.

```
$ man comando
```

- **whatis:** Da una breve descripción de qué hace el comando.

Ejemplo.

```
$ whatis comando
```

3.4. Comandos para archivos o directorios

En esta subsección, abordaré los más importantes y, aquellos que son menos conocidos, tendrán varios ejemplos de su funcionamiento:

- **ls:** Enlista los archivos y directorios. Opciones: `-l`, lo hace con fecha, permisos y tamaño.
- **cd:** Mueve entre directorios. El directorio `.`, es donde nos encontramos y el directorio `..` es el anterior.

Ejemplo 1.

```
~/Documentos/Tesis$ cd .  
~/Documentos/Tesis$
```

no hace nada.

Ejemplo 2.

```
~/Documentos/Tesis$ cd ..  
~/Documentos$
```

se regresa al directorio anterior.

Ejemplo 3.

```
~/Documentos/Tesis$ cd  
~$
```

se va al directorio */home/usuario*.

Ejemplo 4.

```
~$ cd /ruta  
~/ruta$
```

se va a la */ruta*, que le proporcionemos (ver siguiente punto).

Ejemplo 5.

```
~/Documentos/Tesis$ cd ~/ruta  
~/ruta$
```

se va a cualquier */ruta* dentro del directorio */home/usuario*. El símbolo *~*, indica siempre la ruta */home/usuario*, que es el usuario que tiene iniciada la sesión.

- **pwd**: Muestra la */ruta*, donde nos encontramos, es decir, el directorio en el que estamos parados.

Ejemplo.

```
$ pwd  
/home/Pepito/Imagenes/Hola
```

si estamos en la carpeta *Hola*, dentro del directorio *Imágenes* del usuario *Pepito*.

- **mkdir** *dir*: Crea el directorio */dir*, en donde nos encontremos.
- **rm**: Borra archivos. Opciones: *-r*, borra archivos y directorios que contengan archivos u otros directorios; *-f* borra archivos de manera forzada; *-rf* borra archivos y directorios de manera forzada.
- **cp**: Copia archivos. Opciones: *-r*, copia archivos y directorios que contengan archivos u otros directorios.
- **mv**: Mueve archivos entre directorios sin copiarlos en el que estaban inicialmente. Normalmente es usado para renombrar. Opciones: *-r*, aplica sobre directorios también.
- **date**: Muestra la fecha y hora.

Ahora, se presentaran comando que no son tan usados, pero que tienen muchísima utilidad.

- **cat**: Muestra todo el contenido de uno o varios archivos en terminal, pero si quieres leerlo desde el principio, no es muy conveniente ya que habría que regresarse al principio. La ventaja que tiene **cat**, es que puede ser utilizado para concatenar archivos.

Ejemplo.

```
$ cat archivo1.txt archivo2.txt > Archivo.txt
```

lo que hace este comando es concatenar los archivos `archivo1.txt` y `archivo2.txt` ordenandolos uno abajo del otro con jerarquia de argumento, es decir, primero `archivo1.txt` y luego `archivo2.txt`, y todo es mandado a un nuevo archivo llamado `Archivo.txt`, que puede o no existir previamente.

- **paste**: Este comando es muy importante para nosotros ya que permite juntar archivos separados que contienen tablas, en un sólo archivo que contenga una “super tabla” creada con las anteriores.

Ejemplo.

```
$ paste tabla1.txt tabla2.txt > SuperTabla.txt
```

- **less**: Muestra el contenido de los archivos desde el principio.
- **more**: Muestra el contenido de los archivos página por página.
- **chmod**: Este comando permite cambiar los privilegios de lectura “r”, escritura “w” y ejecución “x” de los archivos. Los permisos se pueden dar al propietario del archivo “u”, al grupo al que pertenece “g” o a todos los usuarios “a”.

Ejemplo 0.

```
$ chmod abc archivo
```

le da privilegios “abc” al archivo *archivo*. En este ejemplo “a”, “b” y “c” son números que van de 0 (sin permisos) a 7 (permiso para leer, escribir y ejecutar) . En ese orden, “a” representa los permisos de propietario, “b” del grupo y “c” de todos.

- 1: Opción “--x”, sólo ejecutable.
- 2: Opción “-w-”, sólo escritura.
- 3: Opción “-wx”, escritura y ejecución.
- 4: Opción “r--”, sólo lectura.
- 5: Opción “r-x”, lectura y ejecución.
- 6: Opción “rw-”, lectura y escritura.
- 7: Opción “rwx”, lectura, escritura y ejecución.

Ejemplo 1.

```
$ chmod 426 archivo
```

le da permisos de lectura al propietario, de escritura al grupo y de lectura y escritura a los demás “-r---w-rw-”.

- **diff**: Muestra las diferencias que hay entre dos archivos. Esto resulta muy útil cuando escribiste un código, es muy largo y le hiciste alguna modificación y de pronto ya no funciona. En este caso el comando `diff arch1-backup.c arch1-nuevo.c`, te permitiría ver si hay diferencias entre ambos archivos para saber en qué te equivocaste.
- **grep**: Te permite buscar palabras o patrones específicos en un archivo. Opciones: -f, le especifica que va a buscar sólo en archivos, para que no pase por todos los directorios, lo cual puede hacer lenta la búsqueda.

Ejemplo.

```
$ grep "Hola" -f ./*.dat
```

este comando te permite buscar la palabra “Hola” dentro de todos los archivos que terminen en “.dat” que estén en la carpeta en la que te encuentras.

- **locate**: Localiza buscando patrones.

Ejemplo.

```
$ locate 'hola'
```

busca todos los archivos, carpetas y rutas en general que contengan la palabra “hola” en su nombre, es muy poderoso ya que encuentra todo, y sirve si requieres una búsqueda global, pero debido a que también despliega rutas (directorios padre), si quieres buscar sólo un archivo específico, no es muy conveniente.

- **find**: A diferencia de locate, este comando te permite especificar: tipo de archivo “-type f” para archivo “-type d” para directorio, te permite especificar nombre “-iname (nombre sin distinción de mayúsculas y minúsculas) '*hola*' (palabras con patrón *hola* dentro)”.

Ejemplo

```
$ find . -type f -iname '*hola*'
```

- **tar**: Comprime y descomprime directorios.

Ejemplo 1.

```
$ tarx -czvf empaquetado.tar.gz /carpeta/
```

comprime.

Ejemplo 2.

```
$ tarx -xzvf empaquetado.tar.gz
```

descomprime.

- **sed**: Este comando es una herramienta poderosa pero complicada de utilizar. Su función es buscar palabras o patrones dentro de uno o varios archivos, y reemplazarlas por otras.

Ejemplo 1. \$ sed 's/hola/adios/g' *.dat

```
$ sed 's/hola/adios/g' *.dat
```

reemplaza la palabra “hola” por la palabra “adios” (sin importar si tienen o no caracteres antes o después) de todos los archivos que terminen en “.dat” dentro de la carpeta en la que te encuentras. El resultado de este cambio lo despliega en la terminal, sin embargo, los archivos originales NO SE MODIFICAN.

Ejemplo 2.

```
$ sed 's/hola/adios/g' viejo > nuevo
```

hace lo mismo que en el caso anterior pero sólo para el archivo “viejo” y el resultado lo manda al archivo “nuevo”, dejando a “viejo” igual. ¿Habría forma de hacer esto para todos los archivos “*.dat” y que genere “*-nuevo.dat”?

Ejemplo 3.

```
$ sed -i 's/hola/adios/g' viejo
```

este comando hace lo mismo que los anteriores pero ahora el archivo original sí se modifica. Ya que este comando cambia todos los patrones y la sintaxis es complicada, es preferible siempre crear archivos nuevos, o bien, tener “*.bak” (backups) de tus archivos.

- **echo:** Así sólo es un comando que manda a pantalla lo que tú le escribas entre “”. Pero, debido a que su característica es desplegar una cadena de caracteres, se puede extender su utilidad en varios casos, como se verá más adelante.

3.5. Comandos de red

Hay una gran variedad de comandos de cuyo uso y configuración es bastante complicado por lo que no será necesario para este taller. Pero se anexará un archivo de ello eventualmente.

3.6. Comandos compuestos

Primero hablaremos de los símbolos como comandos:

- **~:** Apaga el equipo.
- **.:** Ubica el directorio actual.
- **>:** Como ya se mencionó anteriormente, redirecciona las salida de los comandos.
- **2>:** Este comando redirecciona los errores que surjan, cosa que es muy útil en Linux.
- **& :** Colocado al final de la línea de comando, lo ejecuta en segundo plano.
- **&& orden1 && orden2:** Ejecuta “orden2” si “orden1” terminó correctamente.
- **|| orden1 || orden2:** Ejecuta “orden2” si “orden1” no terminó correctamente.

3.7. Comando pipe |, línea de comandos, comandos compuestos.

El comando pipe, de ahora en adelante |, tiene la tarea de redireccionar comandos. Es decir, al colocar | al final de una línea de comandos, el output que de ahí provenga, será enviado como input a la línea de comandos que le preceda.

Ejemplo 1.

```
$ ls -l | grep "archivo-especifico"
```

el comando anterior ejecuta el comando `ls -l` que, como ya vimos, despliega los archivos junto con sus permisos, pero no imprime el output en pantalla, sino que esa lista la manda como input a `grep ‘archivo-específico’`, que, como ya vimos, lo que hace es buscar, en esa lista, el patrón “archivo-específico”. Por lo que sólo mostrará los archivos que tengan ese patrón en su nombre.

Ejemplo 2.

```
$ find . -type -iname "*" | xargs grep -l 'patron'
```

este comando busca todos los archivos, de todos los directorios que esten en “.”, es decir, en el directorio actual y el output lo redirecciona a grep para que en esa lista busque el patrón. Para no llenar el buffer se utiliza `xargs`, para que lo haga directorio por directorio.¹

Ejemplo 3.

```
$ find . -type f -iname "*.txt" | xargs rm
```

un comando similar pero ahor lo que hace es borrar todos los archivos “.txt” que tengamos en el directorio actual.

4. Programación en Bash

Bash le proporciona al usuario la posibilidad de hacer programación. Como en C, Python, Fortran, etc, Bash tiene condicionales (if), estructuras de control (while, for, do) y se pueden declarar variables. Y una gran ventaja es que todo lo puedes hacer en una línea de comandos.

4.1. Declaración de variables

Para declarar una variable basta con hacer lo siguiente:

Ejemplo.

```
$ var=15
```

y con eso hemos de clarado “var” igual a 15. El signo igual debe de ir pegado al 15. Para acceder a ella se utiliza el signo “\$”.

Ejemplo.

```
$ echo \${var}
15
```

Las variables se pueden agrupar en listas. Por ejemplo, al escribir “`var=*.txt`” lo que hacemos es guardar todos los nombres de los archivos en una lista.

También hay variables ya preestablecidas como `$HOME`, cuya declaración es la carpeta `/home` del usuario `/home/usuario`. Y otro muy importante llamado `$PATH`, que veremos más adelante.

4.2. Condicional if

La estructura básica del condicional “if” en Bash es:

```
1 if [ "algo pasa" ]
2 then
3     "comandos"
4 fi
```

Es muy importante que lo que vaya dentro de los corchetes esté separado de los corchetes. En el siguiente ejemplo se muestra un uso rápido de if, así como un ejemplo muy sencillo de cómo programar en una línea.

Ejemplo.

¹En este ejemplo aparece el comando `xargs`. Lo que hace este comando es que en lugar de que `grep` actúe sobre todo los outputs de `find`, actúa por fichero. Esto es útil ya que el buffer reservado no puede almacenar listas muy largas.


```
$ x=15; if [ \ $x -ge 12 ]; then echo 'Nel'; else echo 'si'; fi
```

Los operadores son =, !=, <, > para cadenas alfanuméricas; -lt, -le, -eq, -ge, -gt, -ne para valores numéricos; -d, -e, -f para comprobar si el fichero existe y es directorio, existe o es un archivo regular, respectivamente. Entre muchos otros para los cuales hay que checar el manual de Bash.

4.3. For, do y while

Juntare estas estructuras en una sola sección, ya que muchas veces van de la mano. Para hacer bucles se utiliza “for” con la siguiente estructura:

```
1  for x in LISTA
2      do
3          'comandos'
4      done
```

Las listas se pueden escribir como:

\$ for x in 1 2 3 4: Lista de 1 a 4.

\$ for x in {1..10}: Lista de 1 a 10.

\$ for x in {1..10..2}: Lista de 1 a 10 de dos en dos.

\$ for x in file1 file2 file3: Lista de 3 archivos.

\$ for x in '*.txt': Lista de archivos terminados en “.txt”. \$ for x in \$lista: Lista contenida en una variable.

Para hacer bucles con “while”, la estructura es muy similar:

```
1  while CONDICION
2      do
3          'comandos'
4      done
```

Ejemplo 1.

```
$ for x in 1 2 3 4; do echo \ $x; done
1
2
3
4
```

Ejemplo 2.

```
$ dat=1; while [ \ $dat -le 20 ]; do echo \ $dat; dat=$((dat+1)); done
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Aquí se acaba de utilizar una operación matemática al aumentar `$dat` de uno en uno, nótese que para hacer operaciones matemáticas se tiene que poner `(())` (doble paréntesis).

Hay muchas otras herramientas para programar en bash como `case`, `continue`, `until`, etc. Pero estos son los que considero más importantes.

5. Script ejecutable

Por supuesto, cuando se tiene pensado hacer un código un poco más robusto y que no sería conveniente escribir en una sola línea, debido a los posibles errores que esto atribuya (más que nada por el “dónde” poner el punto y coma), es mejor hacer un archivo ejecutable. Así como en C o Fortran, uno puede hacer su script de código y luego correrlo desde terminal. La ventaja respecto de C o Fortran es que no requiere compilación el código, la desventaja es que no tiene todo el poder de estos dos lenguajes.

Para hacer un script yo recomiendo usar el editor de textos Vim, que es un editor en terminal, pero Gedit, Sublime o cualquier editor parecido que abra archivos `.txt` o archivos en general, es bueno.

Los que se requiere para hacer un ejecutable es:

- 1) Crear el archivo. No necesita tener una terminación en particular, puede ser `programa.txt`, `programa.exe`, `programa`, etc.
- 2) Al abrir el archivo, se debe escribir la siguiente línea: `#!/bin/bash` y con esto le diremos que vamos a usar Bash.
- 3) Posteriormente se crea el código, ya sea en una línea o bien estructurado, con indentación o como mejor le convenga al usuario.
- 4) Finalmente, se guarda el archivo y se le dan permisos de ejecución que, como vimos anteriormente se hace utilizando `$ chmod +x programa`.
- 5) Y listo. Para correr el script se teclea en terminal:
`$./programa`.

El script, como los comandos normales de Bash puede tener atributos. **Ejemplo 1.**

```
$ ./programa archivo1 archivo2
```

al ejecutar así el script los archivos `archivo1` y `archivo2` entran como variables al programa y se puede acceder a ellos mediante las variables `$1` y `$2`, respectivamente, donde el 1 y el 2 representan el lugar en el que fue colocado el atributo.

Ejemplo 2.

```
$ ./programa *.dat
```

en este caso, los atributos son una lista de archivos `.dat`. Para acceder a ellos dentro del script se utiliza `$`.

A continuación se muestra un ejemplo de un ejecutable. El siguiente programa crea un “backup” de todos los archivos dentro de una carpeta.

1. Creamos el archivo y lo abrimos.

```
$ vim programa
```

2. Escribimos el código y, de preferencia, lo comentamos.

```

1  #!/bin/bash          ###Llamamos a Bash
2
3  FILES="$@"          ###Definimos la variable FILES como una lista.
4
5  for f in $FILES      ###Iteramos sobre los archivos
6  do
7      if [ -f ${f}.bak ] ###Si ya existe el archivo (-f) .bak
8      then
9          echo "Saltando archivo $f..."
10         continue      ###y sigue...
11     fi
12
13     /bin/cp $f $f.bak  ###Hace la copia de seguridad
14 done

```

3. Se le dan permisos de ejecución al archivo.

```
$ chmod +x programa
```

4. Se ejecuta con sus respectivos argumentos.

```

$ ./programa *
Saltando archivo hola.bak
Saltando archivo astro.bak
$

```

Existen muchas opciones que se pueden poner después del `#!/bin/bash`, pero me parece que no es importante mencionarlas aquí.

Dentro de los archivos ejecutables, además de poder hacer códigos en Bash, puedes llamar a distintos programas con ayuda de EOF:

Ejemplo.

```

1  #!/bin/bash          ###Llamamos a Bash
2  c=$1                ###Creamos una variable de Bash que proviene de
3                      ###atributo externo
4
5  python << EOF        ###Llamamos a Python
6  "Codigo usual de python donde se usa la variable de Bash como ${c}"
7  EOF

```

Si se desea hacer de dicho ejecutable un comando que todo usuario pueda utilizar, se debe colocar dicho ejecutable en alguna de las carpetas `/bin`. Lo más adecuado es siempre colocarlo en la carpeta `/usr/local/bin`. Sin embargo, si por alguna razón se desea colocar en alguna otra carpeta `/bin`, uno puede conocer las rutas por las que Bash busca ejecutables con `echo $PATH`. La variable de entorno `$PATH` muestra en qué rutas se encuentran los ejecutables o binarios de los programas y comandos que utilizamos. Si se requiere, también se puede modificar dicha variable.

En los archivos adjuntos tenemos una serie de ejecutables y archivos con ejemplos que usan ya sea programación en una línea, como códigos más grandes para realizar diversas tareas.

6. Otros programas de UNIX

Lo mostrado anteriormente son los comandos y propiedades más importantes y, digamos, de cultura general para cualquier usuario de sistema UNIX. Sin embargo, hay muchos otros comandos y programas que

vale la pena mencionar:

- **sudo**: Te permite escribir una línea de comandos después y ejecutarla como superusuario sin meterte a “su”. Te pide la contraseña de superusuario.
- **apt-get**: En modo superusuario con **apt-get install programa**, uno puede descargarse cualquier programa en Debian y Ubuntu.
- **yum**: Lo mismo pero para Fedora/RedHat.
- **touch**: crea un archivo pero no lo abre.
- **open**: Comando de macOS para abrir programas con entorno gráfico desde la terminal. En Linux sólo hay que teclear el nombre del programa.
- **make**: Este comando al teclearlo en una carpeta que contenga el archivo “makefile”, cuyo script tenga comandos de Bash, lo que hará es ejecutar todos esos comandos. Una alternativa a crear los ejecutables.
- **top**: Te permite ver qué tarea se está consumiendo toda tu memoria ram.
- **free**: Muestra la memoria disponible en la computadora.
- **history**: Muestra el historial de comandos, muy útil cuando hiciste una línea de comando enorme con programación, variables y comandos, y ya no te acuerdas cómo era.
- **vim**: Editor de textos.
- **nano**: Otro editor de textos, pero feo (a mi parecer).
- **ssh**: Este comando es importante para cualquier persona que quiera tener acceso remoto a una computadora. El comando ssh permite conectarse a otro computador tecleando **\$ ssh usuario@nombre-de-la-compu**, donde “usuario” es la sesión a la que queremos acceder en esa compu y “nombre-de-la-compu” es, por lo general, el IP. Se utilizan comandos hijos como scp para mover archivos entre máquinas. Pero este comando merece todo un taller, así que hasta aquí lo dejaré.
- **grive**: La ventaja de tener un correo institucional, como el de la unam, es que uno tiene acceso a espacio de Drive ilimitado. Para no estar descargando los archivos del Drive, uno puede descargar el programa “grive” para sincronizar todo desde una terminal y tener tus carpetas de Drive en tu /home.
- **youtube-dl**: Este programa creado en python permite descargar vídeos de youtube en formato mp3.
- **maxima**: Ya que estamos trabajando todo en terminal, ¿cuál es la alternativa minimalista de Mathematica o Maple? ¡MAXIMA! Con el mismo poder y sintaxis de Wolfram pero completamente gratis.
- **pdftotext**: Si la tabla que estás buscando está un pdf, la puedes pasar a un .txt con este comando. A menos que el pdf sean fotos o imágenes escaneadas, ahí sí ni modo.
- **texmaths**: El Office de Linux es Libreoffice, al ser software-libre debe tener alguna extensión de LaTeX, y esa es TexMaths. Con esta extensión tendrán a su alcance un botón en la barra de herramientas que le permitirá introducir ecuaciones de LaTeX con los mismos comandos, sintaxis y fuente. Sólo se descarga y se agrega la extensión desde Libreoffice.
- **gimp**: El Photoshop de Linux.
- **mutt**: El correo desde la terminal, puedes configurar todas tus cuentas de correo (preferentemente gmail o equivalente para poder encontrar tutoriales en internet de cómo hacerlo). Este sólo se recomienda a aquellos que ya se la vivan en terminal, pero siempre es bueno poder mandarle un correo a tu asesor pidiendo ayuda si por error desinstalaste el entorno gráfico.
- **w3m**: Abre internet en Terminal. Cosa que igual es muy necesario si le volaste el entorno gráfico y quieres buscar en internet cómo recuperarlo.
- **cron**: Este es un comando complejo pero te permite ejecutar comandos cada cierto tiempo.

- **gnuplot**: El graficador más simple.
- **calendar**: Despliega un calendario.
- **bc**: Calculadora.
- **convert**: Convierte un archivo de cualquier formato en otro. Sólo hay que ser coherente, puede convertir un doc en un pdf, pero no un txt en un mp4.
- **pdftk**: Merge de archivos PDF en uno sólo.
- **ffmpeg**: Convierte archivos multimedia a otro formato multimedia.
- **fdupes**: Te busca archivos duplicados en la carpeta en la que te encuentres o la ruta que le indiques.
- **dpkg**: Te permite instalar repositorios de los cuales tengas el “.deb”, pero preferentemente no lo hagas si no estás seguro de que sabes cómo hacerlo.
- **shutdown**: Apaga la compu, necesitas estar en superusuario.
- **reboot**: La reinicia, necesitas estar en superusuario.
- **mount**: Si te quedaste sin entorno gráfico y sin conexión a internet, pero todo lo que necesitas para actualizar tu compu lo tienes en una USB, sigue los siguientes pasos:
`$ mkdir /media/tmp`, crea una carpeta temporal para montar la USB,
`$ fdisk -l`, inserta la USB y búscala con este comando, normalmente tiene el nombre `/dev/sdb1`, sda es tu disco duro... por lo general.
`$ mount /dev/sdb1` y listo, tu USB es un directorio más y cumplirá con todos comandos que ya vimos.
- **wget**: Descarga lo que quieras de la red.
- **arandr**: Te permite configurar tus pantallas.
- **xfig**: Como un Corel, pero de Linux. Sinceramente es muy complicado de utilizar pero probablemente es el programa para crear imágenes, figuras, diseños, logos, con más herramientas que existe... Y sí, también puedes incluirle LaTeX.

Recuerda que todos estos comandos se pueden utilizar en un script o en un loop, así que le puedes sacar todo el provecho del mundo para que tu computadora haga exactamente lo que tú quieras.

Hay que aclarar que algo importante y que no fue tocado en este taller son tanto los archivos de configuración que te permiten que los comandos de Bash o los programas se ajusten a tus necesidades, así como los archivos que están en las carpetas fuera de `/home`, cuyo manejo adecuado te permite utilizar tu sistema al máximo.

La ventaja es que TODO, absolutamente TODO esto se encuentra en internet y todo es completamente gratuito, así que ya no hay pretexto para no utilizar terminal y entornos UNIX...
