

Práctica 1 - Conceptos de Sistemas Operativos

SISTEMA OPERATIVO

Es parte esencial de cualquier sistema de cómputo.

- Es un programa que actúa como intermediario entre el usuario y el hardware.
- Su propósito es crear un entorno cómodo y eficiente para la ejecución de programas.

Por esto es que decimos que es de propósito general, para que pueda utilizarlo cualquier persona para realizar cualquier tarea.

Para dispositivos con propósitos más específicos existían los microcontroladores. El programador del microcontrolador debía interactuar con el HW del microcontrolador, es decir que no había un SO de propósito general mediando entre el programador y el HW.

- Su obligación es garantizar el correcto funcionamiento del sistema
 - Sus funciones principales
 - *Administrar la memoria*
 - *Administrar la CPU*
 - *Administrar los dispositivos (E/S)*

GNU/LINUX

- Es un Sistema Operativo tipo Unix (Unix like), pero libre
- S.O. diseñado por miles de programadores
- S.O. gratuito y de libre distribución (se baja desde la Web, CD, etc.)
- Existen diversas distribuciones (customizaciones)
- Es código abierto, lo que nos permite estudiarlo, personalizarlo, auditarlo, aprovecharnos de la documentación, etc.
- Podemos ver cómo está hecho

GNU

- **GNU = GNU NO es Unix**
- Iniciado por Richard Stallman en 1983 con el fin de crear un Unix libre (el sistema GNU)
- Para asegurar que el mismo fuera libre, se necesitó crear un marco regulatorio conocido como GPL (General Public License de GNU).
 - *Todo el SW debe distribuirse bajo los términos de una licencia*
 - *GPL → una de las licencias con la cual se distribuye el software libre*
- En 1985, Stallman crea la FSF (Free Software Foundation), con el fin de financiar el proyecto GNU
- En 1990, GNU ya contaba con un editor de textos (Emacs), un compilador (GCC) y gran cantidad de bibliotecas que componen un Unix típico.
 - *Había programas pero faltaba la parte esencial: el Sistema operativo.*
 - *Es así que surge el Kernel*
- Faltaba el componente principal → El Núcleo (Kernel)
- Si bien ya se venía trabajando en un núcleo conocido como TRIX, es en 1988 que se decide abandonarlo debido a su complejidad (corría en hardware muy costoso)
- En este momento se decide adoptar como base el núcleo MACH para crear GNU Hurd, el cual tampoco prosperó
- Linus Torvalds ya venía trabajando desde 1991 en un Kernel denominado Linux, el cual se distribuía bajo licencia GPL
 - El Kernel con el que trabajó era monolítico, es decir, corría todo en el mismo proceso y era viable de correrse en el HW de la época.
- En el año 1992, Torvalds y Stallman deciden fusionar ambos proyectos, y es allí donde nace GNU/Linux
 - Se tuvo entonces la combinación de Kernel con GNU/Linux con los productos, ideas y el marco regulatorio de la GNU.
- GNU/Linux pertenece al desarrollo del software libre

- GNU se refiere a 4 libertades principales de los usuarios del software:
 - Libertad de usar el programa con cualquier propósito
 - Libertad de estudiar su funcionamiento
 - Libertad para distribuir sus copias
 - Libertad para mejorar los programas

“Los programas son una forma de expresión de ideas. Son propiedad de la humanidad y deben ser compartidos con todo el mundo

Software libre

Características del software libre:

- Una vez obtenido, puede ser usado, copiado, estudiado, modificado y redistribuido libremente.
- Generalmente es de costo nulo ← Es un gran error asociar el software libre con el software gratuito ← Pensar en software gratis que se distribuye con restricciones
- Es común que se distribuya junto con su código fuente
- Corrección más rápida ante fallas
- Características que se refieren a la libertad de los usuarios para ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software

Lo que NO se puede hacer con el software libre es RESTRINGIR SU LIBERTAD DE USO

Software propietario

Características del software propietario:

- Generalmente tiene un costo asociado
- No se lo puede distribuir libremente
- Generalmente no permite su modificación
- Normalmente no se distribuye junto con su código fuente
- La corrección de fallas está a cargo del propietario, *a diferencia del software libre que la corrección está a cargo de la comunidad.*
- Menos necesidad de técnicos especializados *a diferencia del SW libre.*

Hay menos necesidad de técnicos especializados, es por eso que con los productos de SW libre hay muchas empresas que su modelo de negocio consiste en dar soportes sobre el SW que distribuye.

GPL

- **Generic Public License**
- Licencia Pública General de GNU
- Creada en el año 1989 por la FSF
- Su objetivo principal es proteger la libre distribución, modificación y uso del software GNU
- Su propósito es declarar que todo software publicado bajo esta licencia, es libre y está protegido teniendo en cuenta las 4 libertades principales ya vistas
- La versión actual de la licencia es la 3

Existen muchas más licencias propietarias y también semi-libres, por lo general detrás de grandes licencias hay también grandes fundaciones.

Como la fundación apache, posee marco regulatorio propio similar pero con algunas diferencias con respecto a la licencia GPL.

GNU/LINUX

- **Información importante**

Es un sistema operativo de tipo UNIX.

Estos implementan determinados estándares, por ejemplo el estándar POSIX, el cual define cómo deben ser los SO para ser compatibles entre sí, de manera tal que un programa compilado en LINUX pueda ser precompilado y generar un programa ejecutable sin alterar el código para otro LINUX.

Por ejemplo para otras variantes de UNIX que no son LINUX → bsd o freebsd

- **Características**

- **Es multiusuario** → debido a que en un instante dado, puede haber varios usuarios ejecutando procesos en un SO UNIX.
- **Es multitarea** → debido a que puede haber varias tareas.
- **Es multiprocesador** → porque el SO puede correrse en equipos que poseen varios procesadores.
- **Es altamente portable** → es portable sobre todo a nivel de código fuente.
Un programa que fue desarrollado en UNIX podría ser compilado en otro UNIX sin ninguna alteración y ejecutarse de manera correcta.
- **Posee diversos intérpretes de comandos, de los cuales algunos son programables** →
- **Permite el manejo de usuarios y permisos** → permite dar permisos particulares a usuarios y a archivos del Fyle System
- **Todo es un archivo (hasta los dispositivos y directorios)** → cualquier componente con el que interactúa el SO es visto como un archivo.
Por ejemplo: un dispositivo → mouse, impresora, etc.
Este componente al cual se lo ve como un archivo se lo guarda en el directorio /dev, de manera tal que un programa en C interactúa con un dispositivo a través de las mismas funciones de C que se utilizan para escribir y leer bytes hacia archivos, es por eso que el SO maneja una

atracción a nivel de archivo.

- **Cada directorio puede estar en una partición diferente (/temp, /home, etc.)** → distintas partes del Filesystem pueden estar montadas en particiones distintas del disco.
- **Es case sensitive** → hace distinción entre mayúsculas y minúsculas (comandos por lo general se escriben en minúscula).
- **Es código abierto**

- **Diseño**

- **Fue desarrollado buscando la portabilidad de los fuentes**
- **Desarrollo en capas**
 - Separación de funciones
 - Cada capa actúa como una caja negra hacia las otras
 - Posibilita el desarrollo distribuido

Las capas colocadas más abajo son las que tienen menor nivel de abstracción y las que están más cercanas al HW. De tal manera que las capas de más abajo abstraen de detalles a las capas de más arriba.

- **Soporte para diversos File Systems** → por la arquitectura modular del SO. Es fácil trabajar en UNIX con distintos Filesystems.
- **Memoria virtual = RAM + SWAP**
- **Desarrollo mayoritario en C y assembler**
- **Otros lenguajes: java, perl, python, etc.**

- **Estructura básica - Kernel**

- También conocido como Kernel
- Ejecuta programas y gestiona dispositivos de hardware
- Es el encargado de que el software y el hardware puedan trabajar juntos
- Sus funciones más importantes son la administración de memoria, CPU y la E/S
- En sí, y en un sentido estricto, es el sistema operativo
- Es un núcleo monolítico híbrido:
 - Los drivers y código del Kernel se ejecutan en modo privilegiado y todos en un mismo proceso → es decir, cada componente del SO corre en el mismo proceso
 - Lo que lo hace híbrido es la capacidad de cargar y descargar funcionalidad a través de módulos → los drivers se cargan y descargan dinámicamente, es decir, no hay que recompilar el kernel de linux
Drivers → pedazo de código que se inserta en el Kernel. Cada vez que el Kernel detecta que hay una aplicación de usuario que quiere leer o escribir hacia un dispositivo lo que hace es delegar esa tarea al Kernel, el cual se ejecuta en modo privilegiado
- Está licenciado bajo la licencia GPL v2

- **HISTORIA DEL KERNEL → min 32**

- **Núcleo - Rama 2.6 y 3.x**

Es importante entender cuál es la manera de versionar el sistema operativo.

- **A: Denota mayor versión.** Cambia con menor frecuencia. En 1994 (versión 1.0) y en 1996 (versión 2.0)
- **B: Denota mayor revisión.** Antes de la versión 2.6, los números impares indicaban desarrollo, los pares producción
- **C: Denota menor revisión.** Solo cambia cuando hay nuevos drivers o características
- **D: Cambia cuando se corrige un grave error sin agregar nueva funcionalidad** ← Casi no se usa en las ramas 3.x y 4.x, viéndose reflejado en C

Intérprete de comandos

- También conocido como CLI (Command Line Interface)
- Modo de comunicación entre el usuario y el SO
- Ejecuta programas a partir del ingreso de comandos
- Cada usuario puede tener una interfaz o shell
- Se pueden personalizar
- Son programables
- Bourne Shell (sh), Korn Shell (sh), Bourne Again Shell (bash)(autocompletado, history, alias)

A diferencia de Windows se puede seleccionar un intérprete de comandos para cada usuario que use el SO.

También puede haber usuarios que no utilicen el CLI. Por ejemplo, un usuario que va a conectarse a un equipo simplemente para utilizarlo como repositorio de archivos ftp no necesita ejecutar un intérprete de comandos

Sistema de archivos

- Organiza la forma en que se almacenan los archivos en dispositivos de almacenamiento (fat, ntfs, ext2, ext3, reiser, etc.)
- El adoptado por GNU/Linux es el Extended (v2, v3, v4)
- Hace un tiempo se está debatiendo el reemplazo de ext por Btrfs (B-tree FS) de Oracle
 - Soporte de mayor tamaño de archivos
 - Más tolerante a fallas y comprobación sin necesidad de desmontar el FS
 - Indexación
 - Snapshots
 - Compresión
 - Desfragmentación
- Directorios más importantes según FHS (Filesystem Hierarchy Standard):

FHS → estándar que siguen todos los SO UNIX para organizar el filesystem.

Al organizar el Filesystem nos referimos a que hay determinados directorios que son muy importantes y tienen que estar presentes en cualquier variante del SO UNIX.

Básicamente es la información que se guarda en determinados directorios.

 - */ → tope de la estructura de directorios. Es como el C:*
 - */home → se almacenan archivos de usuarios (directorios personales de los usuarios, Mis Documentos)*
 - */var → información que varía de tamaño (logs → info que generan los programas continuamente, BD, spools)*
 - */etc → archivos de configuración, si pongo `ls -l /etc` → me muestra en lista todo lo que hay en ese directorio. 5*
 - */bin → archivos binarios y ejecutables*
 - */dev → archivos donde se almacenan dispositivos.*
 - */usr → aplicaciones de usuarios*

Estructura básica del S.O.

- Paquete de software que permite diferenciar una distribución de otra.
- Editores de texto:
 - vi
 - emacs
 - joe
- Herramientas de networking:
 - wireshark
 - tcpdump
- Paquetes de oficina:
 - OpenOffice
- Interface gráficas: permite tener el entorno visual en el que habitualmente trabajamos
 - GNOME / CINNAMON
 - KDE
 - LXDE

Distribuciones

- Organizaciones toman el Kernel de Linux con algunas aplicaciones y arman los que es una distribución.
- Fedora está basado en RedHat
- Ubuntu está basado en Debian

Proceso de arranque

El proceso de arranque es el proceso que lleva a cabo la carga de un kernel a memoria para que el SO esté operativo.

Para esto debemos recordar como es la estructura de un disco. El disco es una serie de platos, los cuales se dividen en pistas concéntricas que se dividen en sectores. Para que esto funcione todos los discos deben almacenar la misma cantidad de información.

En el arranque basado en MBR (viejo), hay un sector reservado del disco (cilindro 0) existe un sector que contiene:

- Un programa llamado Master Boot Code (MBC) que permite arrancar el SO.

- Se tiene una partición que se utiliza para almacenar la información que va a llevar a cabo la carga del SO.

Caso más habitual: arranque de un SO. El sector de arranque posee el MBC que examina la tabla de particiones, en esa tabla hay una partición que está marcada como booteable y ejecuta lo que está en el primer sector de la partición booteable. Se supone que lo que está en el primer sector de la partición marcada como booteable depende del SO.

Caso más atípico: es el que se reemplaza el MBC por otro código más “inteligente” que permite elegir con que SO quiero bootear. Es decir que si tengo un SO en una partición y otro en otra partición, pregunta con que SO quiero iniciar la máquina

PASOS DE PARTICIÓN EN MAQUINA VIRTUAL

CON **ctrl** SALGO DE LA MÁQUINA VIRTUAL

- fdisk → muestra la partición que ya existe

Pasos:

configuración → almacenamiento → agrego disquet (el dado por la cátedra) → entro a la maquina me pide fecha hora → luego ingreso el particionador nuevo “fips” → pongo y 2 veces y le doy el tamaño que quiero a la partición con la flecha y cuando decido toco enter:

Old partition	Cylinder	New Partition
180.4 MB	23	1866.9 MB

guardo la máquina, elimino el disquete y la vuelvo a iniciar. Pongo fdisk y muestra que la partición se hizo correctamente:

```

Display Partition Information

Current fixed disk drive: 1

Partition  Status    Type      Volume Label  Mbytes  System  Usage
C: 1        A          PRI DOS      UNLP           180    FAT16    2%
D: 2                PRI DOS                1867    UNKNOWN  23%

Total disk space is 8033 Mbytes (1 Mbyte = 1048576 bytes)

```

CONFIGURACIÓN DE DISCOS

- **Es Configuración de discos IDE (Integrated Device Electronics):**
 - **Master o Slave** → podían compartir mismo bus
 - **Primer y Segundo bus IDE**

- **Denominación de los discos basada en los buses:**

En el directorio /dev vamos a encontrar los discos y sus particiones.

abcd → hace referencia al disco en el que está guardado

- **/dev/hda: configurado como Master en el 1º bus IDE**
- **/dev/hdb: configurado como Slave en el 1º bus IDE**
- **/dev/hdc: configurado como Master en el 2º bus IDE**
- **/dev/hdd: configurado como Slave en el 2º bus IDE**

- **Particiones primarias → 1 a 4.**
- **Particiones lógicas → desde 5 en adelante**

Para identificar las particiones se le agrega un número a la nomenclatura de hda, hdb, hdc, hdd

/dev/hda1 → primera partición del disco Master del primer bus.

Vamos a tener un archivo por disco y también un archivo por partición.

-
- **Configuración de discos SCSI: se basa en LUN**
 - **Denominación de los discos basada en la identificación de los buses:**
 - **/dev/sda** → primer bus
 - **/dev/sdb** → segundo bus
 - **/dev/sdc** → tercer bus
 - **/dev/sdd**
 - ...

la letra que lo identifica es en base al bus

- **La nomenclatura para los discos SATA es la misma**
- **Particiones primarias:**

- **Se numeran de la 1 a la 4 (solo estas se pueden marcar como activas → booteables)**

Se agrega el número de partición al nombre del archivo

/dev/sda1 → primera partición del disco conectado al primer bus

- **Particiones extendidas:**

- **Sus unidades o particiones lógicas se numeran a partir de la 5**

- **Nueva nomenclatura utilizada:**

- **Con la evolución de las distribuciones GNU/Linux, se comenzó a utilizar “udev” (.rules) como gestor de dispositivos:**

La virtud de udev es poder conectar un dispositivo luego de que la máquina botee.

- **Su función es controlar dinámicamente los archivos del /dev SOLO en base al hardware detectado**
- **Soporta Persistent Device Naming** → vincula el dispositivo a partir de un nombre en vez del /dev/sda. Este avance es importante porque, además de nombrar un dispositivo con un nombre más reconocible, nos permite mantener esa referencia a lo largo del tiempo.
- **Motiva su uso, el no poder garantizar que tras distintos arranques del SO, los dispositivos se sigan llamando de la misma manera. (Suponga disco 1 y 2, que disco 1 se quita y controladoras SCSI/SATA mixtas)**
- **Reemplaza a devfs y hotplug**
- **Se desentiende del Major y Minor Number**
- **Se basa en eventos y permite que nuevos dispositivos sean agregados posteriormente al arranque**

- Desde Debian/Squeeze todos los dispositivos llamados hdX se denominan sdX → sin importancia de si son IDE o SATA
- Por estas y otras razones se adoptan 4 mecanismos nuevos para nomencilar:
 - **Nombres persistentes por UUID (Universal Unique Identifier):** ID único que es generado de tal forma que es imposible que se colisione. Enlace al disco que está definido en el /dev

```
$ ls -l /dev/disk/by-uuid/
2d781b26-0285-421a-b9d0-d4a0d3b55680 -> ../../sda1
31f8eb0d-612b-4805-835e-0e6d8b8c5591 -> ../../sda7
```

- **Utilizando labels** → definen nombres arbitrarios a los discos, por ej disco1, disco de datos

```
$ ls -l /dev/disk/by-label
data -> ../../sdb2
data2 -> ../../sda2
```

SOPORTE DE INSTALACIÓN

Hay que saber que arquitectura de HW utilizarlo antes de instalarlo.

- amd64: Arquitectura de 64 bits
- arm 'o armel: Advanced Risc Machine
- i386: Arquitectura de 32 bits
- ia64: intelItanium o Intel Architecture-64
- Otras

Puede instalarse desde un CD descargado de la web o desde un USB: Unetbootin

HERRAMIENTAS PARA PARTICIONAR

Se recomiendan mínimo dos particiones para instalar Linux, la partición principal (/) y un área de intercambio. Para particionar se puede utilizar:

- **Software destructivo: fdisk** → elimina y crea las particiones correspondientes
- **Software no destructivo: f ps, gparted** → dimensiona las particiones que ya tienen el sistema y se quieren conservar y crea las nuevas para instalar Linux.

Normalmente Debian y las distribuciones basadas en ella tienen un particionador en el momento de la instalación.

CARACTERÍSTICAS DE LINUX

- **No existe el concepto de extensión en el nombre de un archivo** → los archivos en Linux tienen extensión pero no son necesarios para que el SO entienda que formato de archivo son
- **Los subdirectorios no se separan con el carácter '\'**
- **Es case sensitive** → reconoce mayúsculas y minúsculas
- **Entre un comando y sus parámetros debemos dejar obligatoriamente un espacio en blanco**
- **Separación de entorno gráfico y texto**

EDITOR DE TEXTOS vim

Vamos a editar diversos archivos del sistema, para eso se necesita un editor de texto como vim.

Es complicado al iniciar debido a que no es como un editor común sino que tiene tres modos de ejecución distintos.

- **Presente en cualquier distribución de GNU/Linux**
- **Posee 3 modos de ejecución:**
 - **Modo Insert (Ins o i)** → cada vez que apretamos una letra va a insertarse en el texto

- **Modo Visual (v)** → vamos a poder movernos libremente por el texto y seleccionar elementos
- **Modo de Órdenes o Normal (Esc)** → permite escribir cambios, cortar, copiar, pegar, deshacer, etc.
- **Se le puede enviar una serie de comandos útiles**
 - **w:** escribir cambios
 - **q 'o q!:** salir del editor
 - **dd:** cortar
 - **y:** copiar al portapapeles
- **p:** pegar desde el portapapeles
- **u:** deshacer
- **/frase:** busca “frase” dentro del archivo

USUARIOS

Todo usuario debe poseer credenciales para acceder al sistema

- **root:** es el administrador del sistema (superusuario)
- **otros:** usuarios estándar del sistema (/etc/sudoers)

/etc → donde se almacenan todos los archivos de configuración del sistema.

- **Archivos de configuración**

- **/etc/passwd**

Estos “otros” usuarios se almacenan en un archivo llamado /etc/passwd. En este archivo vamos a tener a todos los usuarios junto con sus atributos.

El archivo está estructurado de tal manera que cada línea es un usuario y en esa misma línea vamos a tener todos los atributos de ese usuario.

```
$ cat /etc/passwd
ndelrio:x:2375:500:Nico del Rio,,,:Usuarios:/
home/admins/ndelrio:/bin/bash
```

- **/etc/group**

Se almacenan los grupos que tiene el sistema. Todo usuario pertenece a un grupo. Además puede tener grupos secundarios.

En este archivo vamos a tener los grupos secundarios de cada usuario del sistema

```
$ cat /etc/group
infraestructura:x:500:
```


- **/etc/shadow**

Se tiene el nombre de usuario y la contraseña del usuario. El archivo puede accederse únicamente por el usuario root.

Las contraseñas se encuentran encriptadas.

```
$ cat /etc/shadow
ndelrio:$1$HamkgCYM$TtgfLJLplItxutaiqh/u9
/:13273:0:99999:7:::
```

- **Comandos para el manejo de usuarios:**

Estos comandos pueden ser ejecutados por el administrador o el usuario root y también por usuarios que tengan permiso de administración.

El comando passwd puede ser ejecutado por un propio usuario para cambiar su propia contraseña.

- **useradd <nombreUsuario>:**
 - **Agrega el usuario**
 - **Modifica los archivos /etc/passwd**
 - **Alternativa → adduser**
- **passwd <nombreUsuario>:**
 - **Asigna o cambia la contraseña del usuario**
 - **Modifica el archivo /etc/shadow**
- **usermod <nombreUsuario>:**
 - **-g: modifica grupo de login (Modifica /etc/passwd)**
 - **-G: modifica grupos adicionales (Modifica /etc/group)**
 - **-d: modifica el directorio home (Modifica /etc/passwd)**
- **userdel <nombreUsuario>: elimina el usuario**
- **groupdel <nombreGrupo>: elimina el grupo**

PERMISOS

- Se aplican a directorios y archivos
- Existen 3 tipos de permisos y se basan en una notación octal:

Permiso	Valor	Octal
Lectura	R	4
Escritura	W	2
Ejecución	X	1

- Lectura: permite al usuario poder abrir un archivo para leerlo pero no modificarlo
 - Escritura: permite al usuario escribir en el archivo
 - Ejecución: permite al usuario ejecutar el archivo. Un usuario no puede ejecutar un archivo si no tiene permiso de ejecución.
- Se aplican sobre los usuarios:
 - Usuario: permisos del dueño → U
 - Usuario: permisos del grupo → G
 - Usuario: permisos de otro usuario → O

- Se utiliza el comando **chmod**:

Para asignarle permisos a un archivo o directorios se utiliza el comando chmod. Todo archivo o directorio en Linux tiene un usuario asignado el cual se denomina owner del archivo, pero también tiene un grupo el cual no necesariamente es el mismo que el del usuario

```
$ chmod 755 /tmp/script
```

El número 755 corresponde a los conjuntos de usuarios que uno quiere aplicar el permiso

- 7** → permiso para el dueño del archivo
- 5** → permiso para los usuarios del grupo de ese archivo
- 5** → para todo el resto de los usuarios

En este caso el comando chmod le aplica al archivo permisos de lectura escritura y ejecución al dueño del archivo, de lectura y ejecución a los usuarios del grupo de ese archivo y por último de lectura y ejecución al resto de los usuarios.

ENTORNO

- **Algunos comandos útiles:**
 - **ls** → *listar el contenido de un directorio*
 - **cd** → *moverse entre directorios, tanto hacia dentro como hacia fuera.*
 - **mkdir** → *crear directorio*
 - **rmdir** → *eliminar directorio*
 - **rm** → *eliminar un archivo*
 - **mv** → *mover y renombrar archivos*
 - **cp** → *copiar archivos*
 - **man** → *acceder a documentación de un comando*
 - **info** → *acceder a documentación de un comando pero con un formato más moderno.*

BOOTLOADER

- ❖ El bootloader o cargador de arranque es un programa que permite cargar el Sistema Operativo. Puede llegar a cargar un entorno previo a la carga del sistema.
- ❖ Generalmente se utilizan los cargadores multietapas, en los que varios programas pequeños se van invocando hasta lograr la carga del SO.
- ❖ En cierto sentido, el código del BIOS/UEFI forma parte del bootloader, pero el concepto está más orientado al código que reside en el Master Boot Record (512b)
- ❖ El MBR está formado por el MBC (446b) y la Tabla de Particiones (64b)
- ❖ Sólo el MBC del Primary Master Disk es tenido en cuenta
- ❖ El MBR existe en todos los discos, ya que contiene la tabla de particiones

Si bien usamos group como sinónimo de gestión de arranque, en realidad el proceso arranca desde el mismo inicio del booteo del BIOS.

Siendo group una de las etapas de lo que hablamos como cargadores multietapas, es decir que múltiples programas van a estar cargando distintas etapas del proceso de arranque

SYSTEM V PROCESO DE ARRANQUE

1. Se empieza a ejecutar el código del BIOS
2. El BIOS ejecuta el POST → *todas las pruebas que se hacen sobre el hardware al momento de prender la computadora.*
3. El BIOS lee el sector de arranque (MBR) del disco principal
4. Se carga el gestor de arranque (MBC)
5. El bootloader carga el kernel y el initrd
6. Se monta el initrd como sistema de archivos raíz y se inicializan componentes esenciales (ej.: scheduler)
initrd: sistema de archivos raíz muy liviano que monta los componentes esenciales del sistema. Es un sistema de archivos provisorio.
7. El Kernel ejecuta el proceso init y se desmonta el initrd
8. El proceso init lee el /etc/inittab (/etc → archivos de configuración)
9. Se ejecutan los scripts apuntados por el runlevel 1
10. El final del runlevel 1 le indica que vaya al runlevel por defecto
11. Se ejecutan los scripts apuntados por el runlevel por defecto
12. El sistema está listo para usarse

INIT

Padre de todos los procesos, es decir que todos los demás procesos son subprocesos del INIT y no tiene proceso padre.

- Su función es cargar todos los subprocesos necesarios para el correcto funcionamiento del SO
- El proceso init posee el PID 1 y se encuentra en /sbin/init
- En SysV se lo configura a través del archivo /etc/inittab
- No tiene padre y es el padre de todos los procesos (pstree)
- Es el encargado de montar los filesystems y de hacer disponible los demás dispositivos

RUNLEVELS

- **Modo en que arranca Linux (3 en Redhat, 2 en Debian)**
Son modos de ejecución de como el SO va a iniciar y a operar.
- **El proceso de arranque lo dividimos en niveles**
- **Cada uno es responsable de levantar (iniciar) o bajar (parar) una serie de servicios**
Lo que define un runlevel es qué servicios se van a levantar o qué servicios se van a bajar dependiendo de qué nivel nos encontramos
- **Se encuentran definidos en /etc/inittab**
id : nivelesEjecución : acción: proceso
- **Acción: describe la acción a realizar**
 - **wait:** inicia cuando entra al runlevel e init espera a que termine
 - **initdefault**
 - **ctrlaltdel:** se ejecutar'a cuando init reciba la señal SIGINT
 - **off, respawn, once, sysinit, boot, bootwait, powerwait, etc.**
- **Proceso: el proceso exacto que será ejecutado**

```
$ cat /etc/inittab
id:2:initdefault:
si::sysinit:/etc/init.d/rcS
ca::ctrlaltdel:/sbin/shutdown -t3 -r
```

- Existen 7, y permiten iniciar un conjunto de procesos al arranque o apagado del sistema
- Según el estándar:
 - **0: halt (parada)** → ejecuta secuencia de scripts que hace que el apagado de la máquina sea estable, ejecutando los script de cierre de los servicios que están levantados.
Por esto es que no se recomienda mantener el botón de apagado hasta que el sistema forzosamente se apague, porque al hacer esto el sistema no puede hacer el halt y hacer el apagado del sistema correctamente.
 - **1: single user mode (monousuario)** → permite que un único usuario utilice la computadora.
 - **2: multiuser, without NFS (modo multiusuario sin soporte de red)** → puede ser utilizado por más de un usuario pero sin soporte de red, por lo tanto no va a levantar los servicios de red al momento de cargar el runlevel.
 - **3: full multiuser mode console (modo multiusuario completo por consola)** → puede ser utilizado por más de un usuario con soporte de red pero sin interface gráfica.
 - **4: no se utiliza**
 - **5: X11 (modo multiusuario completo con login gráfico basado en X)** → tiene interface gráfica. Es el que usamos al utilizar Linux
 - **6: reboot** → es el mismo que el halt (0) pero al finalizar ejecuta los scripts de inicio.
- Los scripts que se ejecutan están en /etc/init.d → se encuentran **todos** los scripts disponibles para todos los runlevels.
- En /etc/rcX.d (donde X = 0..6) hay links a los archivos del /etc/init.d → vamos a encontrar los scripts que se van a ejecutar para ese runlevel específico.
- Formato de los links:


```
[S/K]<orden><nombreScript>
```

```
$ ls -l /etc/rcS.d/
S55urandom
S70x11-common
```

- **S: lanza el script con el argument start** → define qué operaciones va a ejecutar cuando inicie el servicio.
- **K: lanza el script con el argument stop** → le dice al script que ejecute las operaciones de kill, es decir que va a ejecutar cuando ese servicio se apague.
- **Número de orden** → orden en cual se va a ejecutar dentro de todos los scripts.

INSSERV

Se pueden crear nuestros propios scripts y agregarlos a un runlevel. También se pueden agregar scripts ya existentes al mismo.

Para esto se utiliza insserv que a grandes rasgos va a administrar esos scripts.

- Se utiliza para administrar el orden de los enlaces simbólicos del `/etc/rcX.d`, resolviendo las dependencias de forma automática
- Utiliza cabeceras en los scripts del `/etc/init.d` que permiten especificar la relación con otros scripts rc → LSBInit (Linux Standard Based Init)
- Es utilizado por `update-rc.d` para instalar/remove los links simbólicos
- Las dependencias se especifican mediante facilities → Provides keyword
- Las facilities que comienzan con \$ se reservan para el sistema (\$syslog)
- Los scripts deben cumplir LSB init script:
 - Proveer al menos 'start, stop, restart, force-reload and status'
 - Retornar un código apropiado
 - Declarar las dependencias

Los scripts deben contar con un formato determinado, que tenga un encabezado que va a hacer que insserv sepa operar con ese script.

- **LSB init script headers:**

```
### BEGIN INIT INFO
# Provides:          my_daemon
# Required-Start:    $syslog $remote_fs
# Required-Stop:     $syslog $remote_fs
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: This is a test daemon
# Description:       This is a test daemon
#                    This provides example about how
#                    to
#                    write a Init script.
### END INIT INFO
```

El script debe proveer:

- *start que defina qué va a realizar cuando inicia*
- *stop cuando finaliza*
- *restart cuando reinicie*
- *cuando se fuerce a reiniciar*
- *status que devuelva el estado del servicio*

Además de un código de error y las dependencias.

PROCESO DE ARRANQUE UPSTART

- **Upstart fue el primer reemplazo propuesto para SystemV (Ubuntu, Fedora, Debian, etc.)** → *bloqueaba distintas tareas hasta que la actual no se haya terminado, por esto surgen los jobs*
- **Permite la ejecución de trabajos en forma asincrónica a través de eventos (event-based) como principal diferencia con sysVinit que es estrictamente sincrónico (dependency-based)** → *es asincrónico basado en eventos que ejecutan jobs. Estos scripts que se ejecutaban en un runlevel específico se deben ejecutar en el momento que un evento suceda. Por ej cuando se monta un usb.*
- **Estos trabajos se denominan jobs**
- **El principal objetivo de un job es definir servicios o tareas a ser ejecutadas por init**
- **Son scripts de texto plano que definen las acciones/tareas (unidad de trabajo) a ejecutar ante determinados eventos**
- **Cada job es definido en el /etc/init (.conf)**
- **Suelen ser de dos tipos:**
 - **Task: ejecución finita (task)** → **not respawning** → **exit 0 o uso de stop**, *ejemplo: monte el sistema de archivos de un USB y finalice.*

- **Service:** ejecución indeterminada → **respawning**, *ejemplo: servidor de impresora o conexión de red*
- Los jobs son ejecutados ante eventos (arranque del equipo, inserción de un dispositivo USB, etc.):
 - Es posible crear eventos pero existen algunos de manera estándar 2
 - Definido por start on y stop on
- Es compatible con SystemV → /etc/init/rc-sysinit.conf, runlevels, scripts en /etc/init.d, objetivo start y stop
- Cada job posee un objetivo (goal start/stop) y un estado (state)
 - En base a ellos se ejecuta un proceso específico
 - Al inicio, init emite el evento startup
A diferencia de system V que tiene un parámetro de inicio y de parada y nada más.
- Un job puede tener uno o varias tareas ejecutables como parte de su ciclo de vida y siempre debe existir la tarea principal
- Las tareas de un job se definen mediante **exec o script ... end script**
Las tareas se definen en el archivo del script mediante exec si es una sola línea, o si es un bloque -script - el bloque - end script
- A través de **initctl** podemos administrar los jobs del demonio de Upstart:
 - **start** : cambia el objetivo a start del job especificado
 - **stop** : cambia el objetivo a stop del job especificado
 - **emit** : event es emitido causando que otros jobs cambien a objetivo start o stop
- No más /etc/inittab

SYSTEMD

Es asincrónico y en vez de trabajar en base a runlevels trabaja en base a targets. Estos targets pueden ser activados o desactivados a través del comando systemctl.

- Es un sistema que centraliza la administración de demonios y librerías del sistema
- Mejora el paralelismo de booteo
- Puede ser controlado por systemctl
- Compatible con SysV → si es llamado como init
- El demonio systemd reemplaza al proceso init → este pasa a tener PID 1
- Los runlevels son reemplazados por targets
- Al igual que con Upstart el archivo /etc/inittab no existe más
- Las unidades de trabajo son denominadas units de tipo:
Maneja un conjunto de units y targets, Units van a realizar ciertas tareas y van a ser de diferentes tipos
 - **Service:** controla un servicio particular (.service) → units que tienen su start, stop y reload, y van a estar corriendo constantemente sirviendo algún tipo de tarea como un servidor web.
 - **Socket:** encapsula IPC, un socket del sistema o file system FIFO (.socket) → socket-based activation → permiten comunicarse entre units o activar o desactivar units a partir de ciertos eventos.
 - **Target:** agrupa units o establece puntos de sincronización durante el booteo (.target) → dependencia de unidades → es lo más cercano a un runlevels. Podemos hacer que ciertos units ejecuten a partir de la acción de un target.
 - **Snapshot:** almacena el estado de un conjunto de unidades que puede ser restablecido más tarde (.snapshot)
- Las units pueden tener dos estados → active o inactive

Activación por Socket

- **No todos los servicios que se inician en el booteo se utilizan:**
 - **impresora** → *se utiliza únicamente al imprimir*
 - **servidor en el puerto 80** → *solo se ejecuta cuando se le haga un requerimiento*
 - **etc.**
- **Es un mecanismo de iniciación bajo demanda** → **podemos ofrecer una variedad de servicios sin que realmente estén iniciados**
Se pueden activar o desactivar servicios bajo demandas a través de sockets.
- **Cuando el socket recibe una conexión inicia el servicio y le pasa el socket**
- **No hay necesidad de definir dependencias entre servicios** → **se inician todos los sockets en primer medida**

SYSTEMD - cgroups

Se puede agrupar un conjunto relacionado de forma jerárquica y darle un conjunto de recursos limitados.

- **Permite organizar un grupo de procesos en forma jerárquica**
 - **Agrupar conjunto de procesos relacionados (por ejemplo, un servidor web Apache con sus dependientes)**
- **Tareas que realiza:**
 - **Tracking mediante subsistema cgroups** → **no se utiliza el PID** → **doble fork no funciona para escapar de systemd**
 - **Limitar el uso de recursos**
 - **etc**

fstab

- **Define qué particiones se montan al arranque**
- **Su configuración se encuentra en /etc/fstab:**

```
$ cat /etc/fstab
# <file system> <mount point> <type> <options> <dump>
  <pass>
/dev/sda1 / ext4 errors=remount-ro 0      1

UUID=3FDE00F9523092AE /home/iso/datos ntfs user,auto
,rw,exec,uid=1000,gid=1000,umask=000 0 2

/dev/sda2 none swap sw 0 0
```

- Se ve la partición definida /dev/sda1 o por medio de UUID
 - Su punto de montaje
 - El sistema de archivos
 - Varias opciones y flags para definir
- **Opciones:**
 - **user:** cualquier usuario puede montar la partición → que usuario puede montar la partición
 - **auto:** monta la partición al inicio → si la partición se monta automáticamente o cuando recién quiere ingresar esa partición
 - **ro: read only, rw: read and write** → para definir si la partición se monta solo de lectura o de lectura y escritura.
 - **etc.**

Redirecciones

Se puede redirigir la salida estándar de una red a un archivo
 salida estándar → lo que se imprime en pantalla
 por lo tanto vamos a poder en vez de imprimir en pantalla, escribir en un archivo

- **Al utilizar redirecciones mediante > (destruktiva):**
 Al definir la redirección de un archivo, el archivo va a ser eliminado y se va a crear uno nuevo con el nuevo contenido.
 - Si el archivo de destino no existe, se lo crea →
 - Si el archivo existe, se lo trunca y se escribe el nuevo contenido
- **Al utilizar redirecciones mediante >> (no destruktiva):**
 Al definir la redirección de un archivo, el archivo va a agregar el contenido nuevo al final del archivo.
 - Si el archivo de destino no existe, se lo crea
 - Si el archivo existe, se agrega la información al final

`ls < nombreArchivo >` → si vamos al directorio vamos a ver que el `nombreArchivo` existe y tiene el contenido que el `ls` debería haber mostrado.

Uso de |

- El “|” nos permite comunicar dos procesos por medio de un pipe o tubería desde la shell
- El pipe conecta stdout (salida estándar) del primer comando con la stdin (entrada estándar) del segundo.
- **Por ejemplo:** `$ls | more`
 - Se ejecuta el comando `ls` y la salida del mismo, es enviada como entrada del comando `more`.
 - `ls` muestra el listado de archivos de un directorio y comunicarlo con la entrada del `more`, que va a partir la entrada grande de archivos en distintos segmentos
- Se pueden anidar tantos pipes como se deseen
- **¿Cómo haríamos si quisiéramos contar la cantidad de usuarios del sistema que en su nombre de usuario aparece una letra “a”?**
`$ cat /etc/passwd | cut -d: -f1 | grep a | wc -l`

Imprime o muestra en la salida estándar el estado de usuarios del sistema. Esa salida la comunica a la entrada del cat.

El cat va a quedarse con la columna inicial de ese archivo de usuarios y mandarlo a la salida estándar. Pero se va a comunicar nuevamente con la entrada estándar de grep.

*El grep se queda únicamente con los nombres de usuarios que tengan la letra “a”. No lo muestra en pantalla sino que lo manda al wc -l
wc -l va a contar la cantidad de líneas que devuelve grep,
Por lo tanto lo que hace esta línea es contar la cantidad de usuarios que poseen la letra “a”.*

Práctica 3 - Conceptos de Sistemas Operativos

Shell: intérprete de comandos, funciona de manera interactiva y permite ingresar comandos y ejecutarlos para realizar tareas.

En general todos LOS SO tienen un tipo de shell. En el caso que son derivado de UNIX son *nix, la shell es configurable, lo que quiere decir que cada usuario puede especificar su propia shell y se puede tener más de una instalada y ejecutándose a la vez en la máquina.

La shell nos provee los fundamentos básicos para poder programar los *shell scripts* (pequeños programas que se interpretan mediante la shell).

Estos *shell scripts* permiten automatizar tareas, realizar aplicaciones interactivas, tener aplicaciones con interfaces gráficas que se proporcionan con el comando zenity.

Esto quiere decir que *no estamos atados a que la única forma de interacción sea mediante el ingreso de texto sino que tenemos interfaces que pueden ser más ricas que eso.*

Tipos de shell

Existen distintos tipos de shell, tratan de tener cierta compatibilidad entre sí:

- sh: Shell por defecto en Unix.
- **bash**: cómoda, instalada por defecto en la mayoría de las distribuciones. Surge como una mejora de sh. Es la que vamos a utilizar en la materia
- dash: eficiente, parcialmente con bash. Es eficiente pero tiene cosas que salen de los estándares por lo tanto no es tan utilizada.
- csh: sintaxis incompatible con bash/dash.
- otros...

Diferencias con otros lenguajes

Existen distintas alternativas para hacer scripts como C, Java o Python.

El potencial de estos es mucho más grande, tienen construcciones, librerías y otras alternativas que bash no posee. Pero tienen la gran complejidad de ser compilado que requieren muchas más cosas hechas alrededor para lograr lo mismo y a su vez pueden tener ciertos problemas de portabilidad entre distintos equipos.

Ventajas:

- bash es práctico para manejar archivos
- Extremadamente simple para crear procesos y manipular sus salidas
- Independiente de la plataforma (a diferencia de C)
- Funciona en cualquier sistema operativo de tipo – nos permite ejecutar el mismo script, prácticamente sin ninguna modificación en muchas plataformas derivadas de unix
- Se puede probar en el intérprete interactivo (a diferencia de C y Java)

Elementos de lenguaje

Sobre los elementos que tenemos en el lenguaje tenemos construcciones básicas para crear los programas o scripts

- Instrucciones: comandos
 - Internos o built-in (help para verlos) → incluidos en la misma shell
 - Externos (man para verlos) → se deben instalar por separado
- Redirecciones y pipes: facilidad para manejar archivos y para hacer que los comandos se comuniquen entre sí. *Nos permiten cumplir con las premisas de Unix.*

Una de ellas es que cada comando debe ser una herramienta específica para poder resolver tareas con la combinación de estos comandos.

- Comentarios que empiezan con #
- Estructuras de control
 - if
 - while
 - for (2 tipos)
 - case
- Variables
 - Strings
 - Arreglos
- Funciones

COMANDOS ÚTILES	
cat archivo → imprimir contenido de un archivo	
echo "Holaaa" → imprimir texto	
read var → leer una línea desde entrada estándar en la variable var	
cut -d: -f1 → quedarme con la 1er columna de un texto separado por: desde entrada estándar	
wc -l → contar la cantidad de líneas que se leen desde la entrada estándar	
grep lola /tmp/* → buscar todos los archivos que contengan la cadena lola en el directorio /tmp	
grep lola /tmp/* → buscar todos los archivos que contengan la cadena lola en el directorio /tmp	
find \$HOME -name "*.doc" → buscar todos los archivos dentro del home del usuario, cuyo nombre termine en .doc. Recibe desde donde queremos buscar y recibe condiciones que cumple el archivo que tiene que encontrar. Al encontrarlo imprime la información en pantalla.	
find -type l → buscar todos los archivos dentro del directorio actual que sean enlaces simbólicos	
tar -cvf archivo.tar archivo1 archivo2 archivo3 tar -xvf archivo.tar → se unen varios archivos en uno solo.	EMPAQUETADO
gzip archivo.tar → genera archivo.tar.gz comprimido gzip -d archivo.tar.gz → descomprime el archivo.tar	COMPRESIÓN Se reduce el tamaño de arch
tar -cvzf archivo.tar.gz arch1 arch2 arch3 tar -xvzf archivo.tar.gz → tar puede invocar a gzip por nosotros (argumento "z") Nos permite realizar el empaquetado y el comprimido en una sola operación	

Ejemplo	
<pre> root@debian # echo Hola mundo Hola mundo root@debian # read variable hdsak dsjahdks ad ahjks root@debian # echo \$variable hdsak dsjahdks ad ahjks root@debian # _ </pre>	<p>→ imprime hola mundo</p> <p>→ lee todas las letras ingresadas y se almacena en "variable"</p> <p>→ echo \$variable hace referencia al contenido de la variable, por lo tanto imprime su contenido</p>

REDIRECCIONES Y PIPES: stdin, stdout, stderr

Los procesos (programas en ejecución) normalmente cuentan con 3 "archivos" abiertos.

Son 3 *lugares* especiales donde enviamos o recibimos información (bytes) que los procesos pueden utilizar para comunicarse con el ambiente/usuario.

Estos tres archivos son:

- **stdin**: Entrada estándar, normalmente el teclado.
- **stdout**: Salida estándar, normalmente el monitor.
- **stderr**: Error estándar, normalmente la salida estándar.

Se identifican con un **file descriptor** → un puntero al archivo, algo que nos dice cómo acceder a él, que son numéricos

- **0**: Entrada estándar → asociada al teclado
- **1**: Salida estándar → al monitor, es decir la consola
- **2**: Error estándar → al monitor, es decir la consola

Esto se puede cambiar especificando cosas distintas a través de las redirecciones y pipes.

REDIRECCIONES

Son de entrada o de salida.

comando > archivo → las redirecciones de salida son las que comunican un comando con un archivo. La salida de nuestro comando va a ir al archivo que le indicamos.

Estas redirecciones pueden ser:

- **> Destructiva**
 - Si el archivo no existe, se crea.
 - Si el archivo existe, sobrescribe → pisa su contenido
- **>> No destructiva**
 - Si el archivo no existe, se crea.
 - Si el archivo existe, agrega al final.

EJEMPLO: al no indicar nada delante del >> se está redirigiendo al archivo de salida estándar, si se utiliza el 2 se va a estar redirigiendo de error estándar

```
cd
ls >> /tmp/lista.txt
cd /tmp
ls >> /tmp/lista.txt
```

```
comando 2> archivo
comando 2>> archivo
comando < archivo
```

Entonces:

- **2> y 2>> Redirigen el error estándar**
- **< Hace que archivo sea la entrada de comando**

En otras palabras cuando comando intente leer entrada del teclado, en realidad, va a leer el contenido de archivo

Ejemplo: un comando que lee desde la entrada estándar (como el read), guarda en una variable. Si hago una redirección desde un archivo hacia ese comando que comunique el archivo con la entrada estándar del proceso, en lugar de leer desde el teclado (entrada estándar por defecto) va a leer desde ese archivo

PIPES

Con los **pipes** vamos a comunicar dos comandos

- Uno que genera salida.
- Otro que recibe esa salida como su propia entrada.

SINTAXIS

Conectan la salida estándar de un comando con la entrada estándar de otro. Esto siempre es en un sentido secuencial en cuanto al orden

comando | **comando2** | **comando3**

La salida estándar de **comando** se va a comunicar con la entrada estándar de comando 2, y a su vez la salida estándar de **comando2** se va a comunicar con la entrada estándar de **comando3**, finalmente la salida estándar de **comando3** va a ser la que salga en pantalla.

Los comandos utilizan esto para hacer comunicación entre procesos y PC.

```
cat archivo | tr a-z A-Z
cat archivo | grep hola | cut -d, -f1
cat /etc/passwd | cut -d: -f1 | grep a | wc -l
cat /etc/passwd | cut -d: -f7 | sort | uniq > res.txt
```

Se ejecuta el comando cat pasándole un archivo que queremos que procese, luego un pipe y luego otro comando. Pueden ser infinitos comandos.
Explicado aprox 35 min.

Ambos son distintas formas de comunicar los procesos

Redirecciones → hablamos de comunicar un proceso con un archivo para entrada o para salida. Siempre una redirección es de un comando a un archivo

Pipes → comunicación entre procesos, entre comandos, los cuales se comunican entre sí en línea y se van a pasar la información mediante la salida estándar y la entrada estándar.

VARIABLES

- En bash las **variables** pueden ser únicamente de tipo *String* o *arreglo*, esto no quiere decir que no puede haber números sino que son considerados como Strings.
- Son sensibles a mayúsculas y minúsculas.
- Para asignar el valor a una variable se debe escribir SIN ESPACIOS en la asignación.
Ejemplo: NOMBRE="dolores" → SIN ESPACIOS entre el igual!!!
- Para acceder al contenido se utiliza \$.
Ejemplo: echo \$NOMBRE
- Para evitar ambigüedades se pueden usar las llaves
Ejemplo: echo \$NOMBREesto_no_es_parte_de_la_variable → NO accede a NOMBRE
echo \${NOMBRE}esto_no_es_parte_de_la_variable → accede a NOMBRE
- Los nombres de variables pueden tener mayúsculas, _, minúsculas, números.
Pero no pueden empezar con número
Ejemplo: echo \$NOMBRE

ARREGLOS

- Secuencias de valores que están guardados y lógicamente conectados.
- Declaración:
arregloVacio=()
arregloLleno=(1 2 3 4 5) *siempre sin espacios*
- Asignación de un valor en una posición concreta
arregloLleno[2]=5
- Acceso de un valor del arreglo (llaves obligatorias)
echo \${arregloLleno[2]}
copia=\${arregloLleno[2]}
- Acceso a todos los valores del arreglo
echo \${arregloLleno[@]}
o
echo \${arregloLleno[*]}

- Tamaño del arreglo:

```
echo ${#arregloLleno[@]}
```

o

```
echo ${#arregloLleno[*]}
```

- Borrar un elemento del arreglo:

```
unset arregloLleno[2]
```

No hace el corrimiento por lo tanto el espacio va a seguir quedando ahí

- Los índices de los arreglos comienzan en 0

COMILLAS

- No hacen falta, a menos que:
 - el string tenga espacios.
 - que sea una variable cuyo contenido pueda tener espacios.
 - son importantes en las condiciones de los if, while, etc...

- Comillas dobles

```
var='variables'
echo "Permiten usar $var"
echo "Y resultados de comandos $(ls)"
```

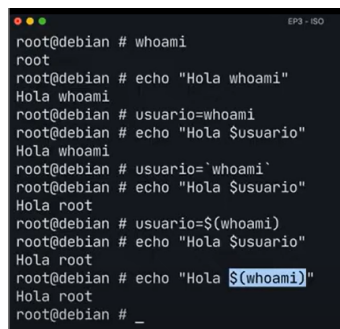
- Comillas simples

```
echo 'No permiten usar $var'
echo 'Tampoco resultados de comandos $(ls)'
```

REEMPLAZO DE COMANDOS

- Permite que la salida de un comando se incluya dentro de alguna otra parte que se especifique.

Ejemplo:



```
root@debian # whoami
root
root@debian # echo "Hola whoami"
Hola whoami
root@debian # usuario=whoami
root@debian # echo "Hola $usuario"
Hola whoami
root@debian # usuario='whoami'
root@debian # echo "Hola $usuario"
Hola root
root@debian # usuario=$(whoami)
root@debian # echo "Hola $usuario"
Hola root
root@debian # echo "Hola $(whoami)"
Hola root
root@debian # _
```

CREAR UN SCRIPT

toco ESC para volver a modo comando

wq → para cerrar y guardar

muestra contenido del script

EJECUCIÓN DE UN SCRIPT

- **absoluta**: empieza desde el directorio raíz (/).
No importa donde esté ubicado en el filesystem, siempre va a hacer referencia al mismo archivo

- **relativa:** empieza desde donde estemos posicionados

- variable de entorno PATH

busque el contenido del archivo que está en el directorio `/etc/passwd`.

Poner como cat ./etc/passwd

El punto es el directorio actual.

1. PERMISO DE EJECUCIÓN

```
dolores@dolores-VirtualBox:~$ chmod a+x mi script.sh
```

```
dolores@dolores-VirtualBox:~$ chmod a+x mi_script.sh
dolores@dolores-VirtualBox:~$ ls -l mi_script.sh
-rwxrwxr-x 1 dolores dolores 63 sep 22 13:07 mi_script.sh
dolores@dolores-VirtualBox:~$
```

2. EJECUCIÓN

- Dándole permiso de ejecución previamente
`$./mi_script.sh` → ejecuta
- otra forma sin tener que darle permiso:
`$ bash mi_script.sh`
- otra forma: debug → muestra comando que ejecuta y otra información adicional
`bash -x mi_script.sh`

```
dolores@dolores-VirtualBox:~$ bash -x mi_script.sh
+ echo 'Hola Mundo!'
Hola Mundo!
```

FUNCIONES

SINTAXIS

```
nombre_de_la_funcion() {  
    # Código de la función  
    # comandos  
}
```

PASANDO ARGUMENTOS

```
mi_funcion() {  
    echo "El primer argumento es: $1"  
    echo "El segundo argumento es: $2"  
}  
mi_funcion arg1 arg2
```

EJEMPLO

```
sumar() {  
    local resultado=$(( $1 + $2 )) → declaración de variable local de la función  
    return $resultado  
}
```

```
sumar 3 4 → parámetros  
resultado=$? → $?: contiene el código de salida de la función  
echo "El resultado de la suma es: $resultado"
```

IF	CASE
<pre> if [condición] then # Código si la condición es verdadera elif [otra_condición] then # Código si la otra condición es verdadera else # Código si ninguna de las condiciones es verdadera fi </pre>	<pre> case \$variable in v1) # Código si la variable es igual a v1 ;; v2) # Código si la variable es igual a v2 ;; *) # Código si la variable no coincide con ningún valor anterior ;; esac </pre>
WHILE	FOR
<pre> while [condición] do # Código que se ejecuta mientras la condición sea verdadera done </pre>	<pre> for ((i=0; i<n; i++)) do # Código done </pre>

SELECT
<pre> select variable in opcion1 opcion2 opcion3 do case \$variable in opcion1) # Código para la opción 1 ;; opcion2) # Código para la opción 2 ;; opcion3) # Código para la opción 3 ;; *) # Código si se selecciona una opción no válida ;; esac done </pre>

