Presentación

Introducción a los Sistemas Operativos/Conceptos de sistemas operativos

Facultad de Informática
Universidad Nacional de La Plata

2025

- Sitio Web: http://catedras.info.unlp.edu.ar (único medio de comunicación)

- Bandas Horarias:
  - Turno Mañana:
    - Jueves de 8:30 a 10:00 - Aula 5
    - Viernes de 13:00 a 14:30 - Aula 5
  - Turno Tarde:
    - Jueves de 19:00 a 20:30 - Aula 5
    - Viernes de 17:30 a 19:00 - Aula 5

  - JTPs:
    - Martin Baez: mbaez@mail.info.unlp.edu.ar
    - Leonardo Otonelo: leonardo.otonelo@info.unlp.edu.ar

- Entran todos los temas vistos en la práctica (enunciados de TP, explicaciones, material adicional brindado por la cátedra, respuestas al foro)
- La materia consta de 2 parciales:
  - Primer Parcial (Prácticas 1 y 2):
    - Primera fecha → 11 de Octubre
    - Segunda fecha → 25 de Octubre
  - Segundo Parcial (Prácticas 3 y 4):
    - Primer fecha → 6 de Diciembre
    - Segunda fecha → 20 de Diciembre
  - Recuperatorio general: Febrero
- Para rendir cada parcial se deben contestar las autoevaluaciones de las prácticas correspondientes (**¡OBLIGATORIAS!**)
- Para aprobar la cursada hay que aprobar ambos parciales

Conceptos Generales

Explicación de práctica 1

Introducción a los Sistemas Operativos/Conceptos de sistemas operativos

Facultad de Informática
Universidad Nacional de La Plata

2025

¿Qué es un Sistema Operativo?

- Es parte esencial de cualquier sistema de cómputo
- Es un programa que actúa, en principio, como intermediario entre el usuario y el hardware
- Su propósito: crear un entorno cómodo y eficiente para la ejecución de programas
- Su obligación: garantizar el correcto funcionamiento del sistema
- Sus funciones principales
  - Administrar la memoria
  - Administrar la CPU
  - Administrar los dispositivos

¿Qué es un Sistema Operativo? (cont.)

- Según Wikipedia:
  "...Es un conjunto de programas de computación destinados a realizar muchas tareas..."
- Según un usuario estándar: "Lo que aparece cuando prendo la PC"
- ...

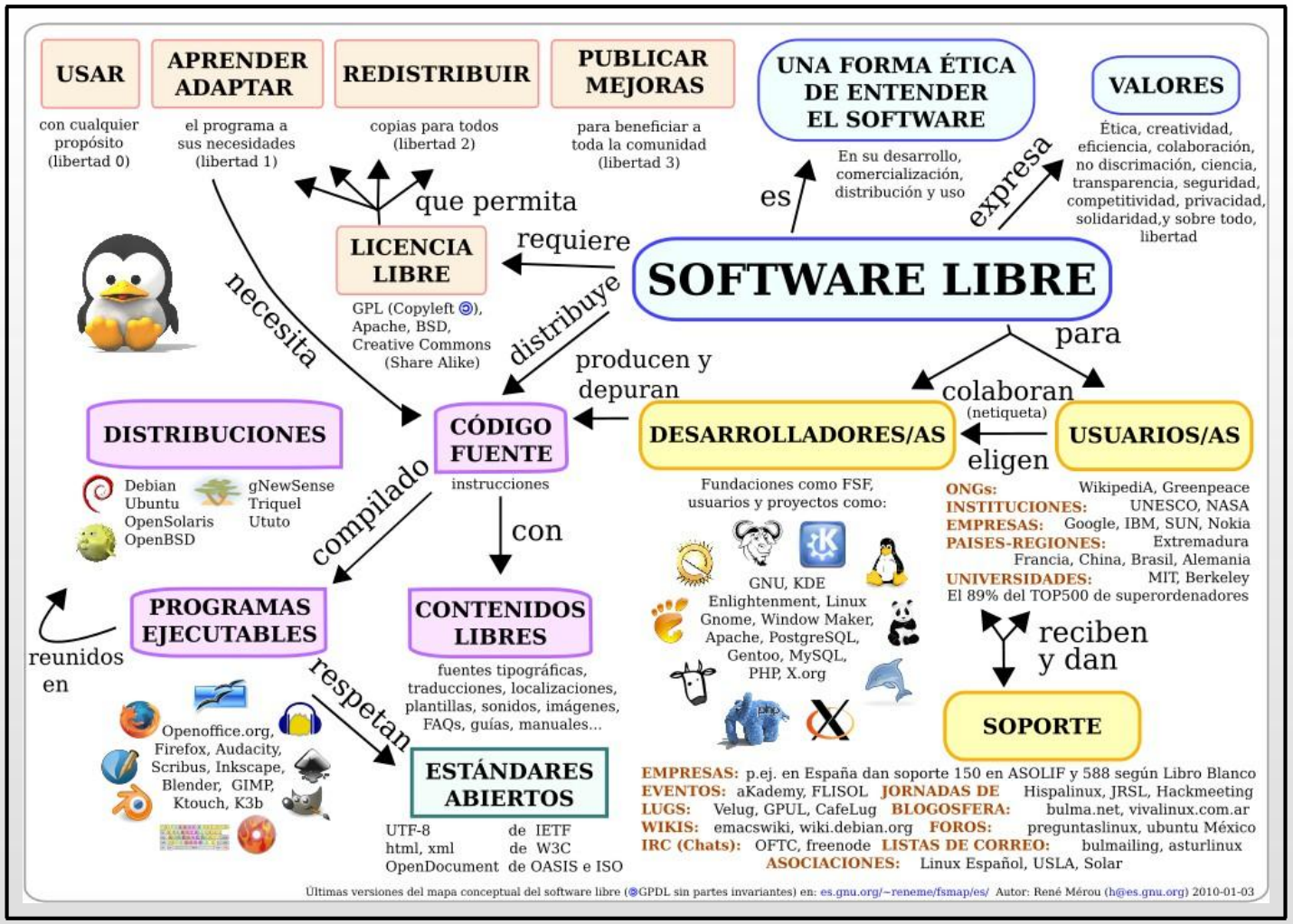
- Es un Sistema Operativo tipo *Unix* (Unix like), pero libre
- S.O. diseñado por miles de programadores
- S.O. gratuito y de libre distribución (se baja desde la Web, CD, etc.)
- Existen diversas distribuciones (customizaciones)
- Es código abierto, lo que nos permite estudiarlo, personalizarlo, auditarlo,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

Podemos ver cómo está hecho!!!

- Hablar de código abierto es referirse 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
  - Caracteríisticas que se refieren a la libertad de los usuarios para ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software

- 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
  - Menos necesidad de técnicos especializados

- GNU = **G**NU **N**o es **U**nix

- 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)
- 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.
- 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 distribuiría bajo licencia GPL
- En el año 1992, Torvalds y Stallman deciden fusionar ambos proyectos, y es allí donde nace *GNU/Linux*
- GNU/Linux pertenece al desarrollo del software libre

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

Características generales de GNU/Linux

- Es multiusuario
- Es multitarea y multiprocesador
- Es altamente portable
- Posee diversos intérpretes de comandos, de los cuales algunos son programables
- Permite el manejo de usuarios y permisos
- Todo es un archivo (hasta los dispositivos y directorios)
- Cada directorio puede estar en una partición diferente (/temp, /home, etc.)
- Es case sensitive
- Es código abierto

Características generales de GNU/Linux

- Iniciado en 1991 por Linus Torvalds
- Se encuentra basado en Minix
- La versión 1.0 apareció en 1994
- Su mascota oficial es Tux 
- Se puede descargar de https://www.kernel.org/
- Actualmente ya supera la versión 6.0

- 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
- Soporte para diversos File Systems
- Memoria virtual = RAM + *SWAP*
- Desarrollo mayoritario en C y assembler
- Otros lenguajes: java, perl, python, etc.

Estructura básica del S.O. - Núcleo

- 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 si, 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
  - Lo que lo hace híbrido es la capacidad de cargar y descargar funcionalidad a través de módulos
- Está licenciado bajo la licencia GPL v2

Nomenclatura: **A.B.C (.D)**

- **A**: Denota versión. Cambia con menor frecuencia. En 1994 (versión 1.0), en 1996 (versión 2.0), etc
- **B**: Denota mayor revisión. Antes de la versión 2.6, los números impares indican 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 en adelante, 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 (ksh), Bourne Again Shell (bash)(autocompletado, history, alias)

- 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
  - Defragmentación

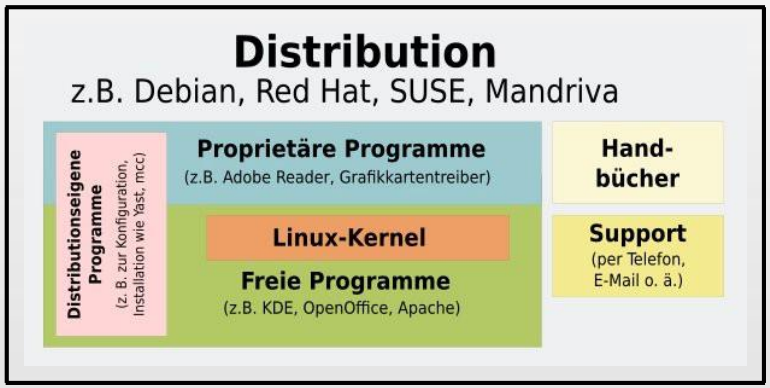
Sistemas de archivos - Directorios más importantes

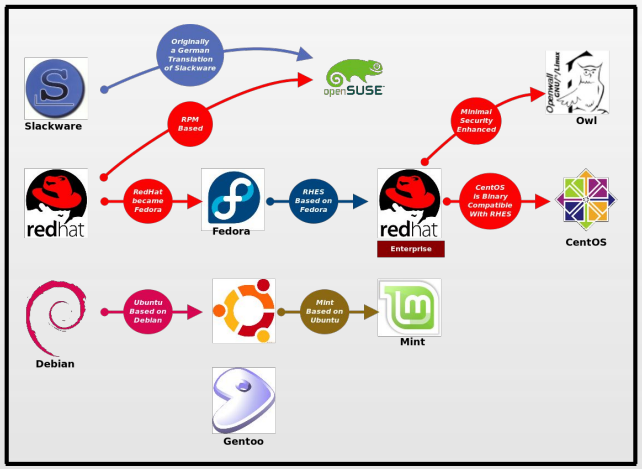
- Directorios más importantes según FHS (Filesystem Hierarchy Standard)
  - / Tope de la estructura de directorios. Es como el C:\
  - /home Se almacenan archivos de usuarios (Mis documentos)
  - /var Información que varía de tamaño (logs, BD, spools)
  - /etc Archivos de configuración
  - /bin Archivos binarios y ejecutables
  - /dev Enlace a dispositivos
  - /usr Aplicaciones de usuarios

Estructura básica del S.O. - Utilidades

- Paquete de software que permite diferenciar una distribución de otra.
- Editores de texto:
  - vi
  - emacs
  - joe
- Herramientas de networking:
  - wireshark
  - tcpdump
- Paquetes de oficina:
  - LibreOffice
- Interface gráficas:
  - GNOME / CINNAMON
  - KDE
  - LXDE

- Una distribución es una customización de GNU/Linux formada por una versión de kernel y determinados programas con sus configuraciones

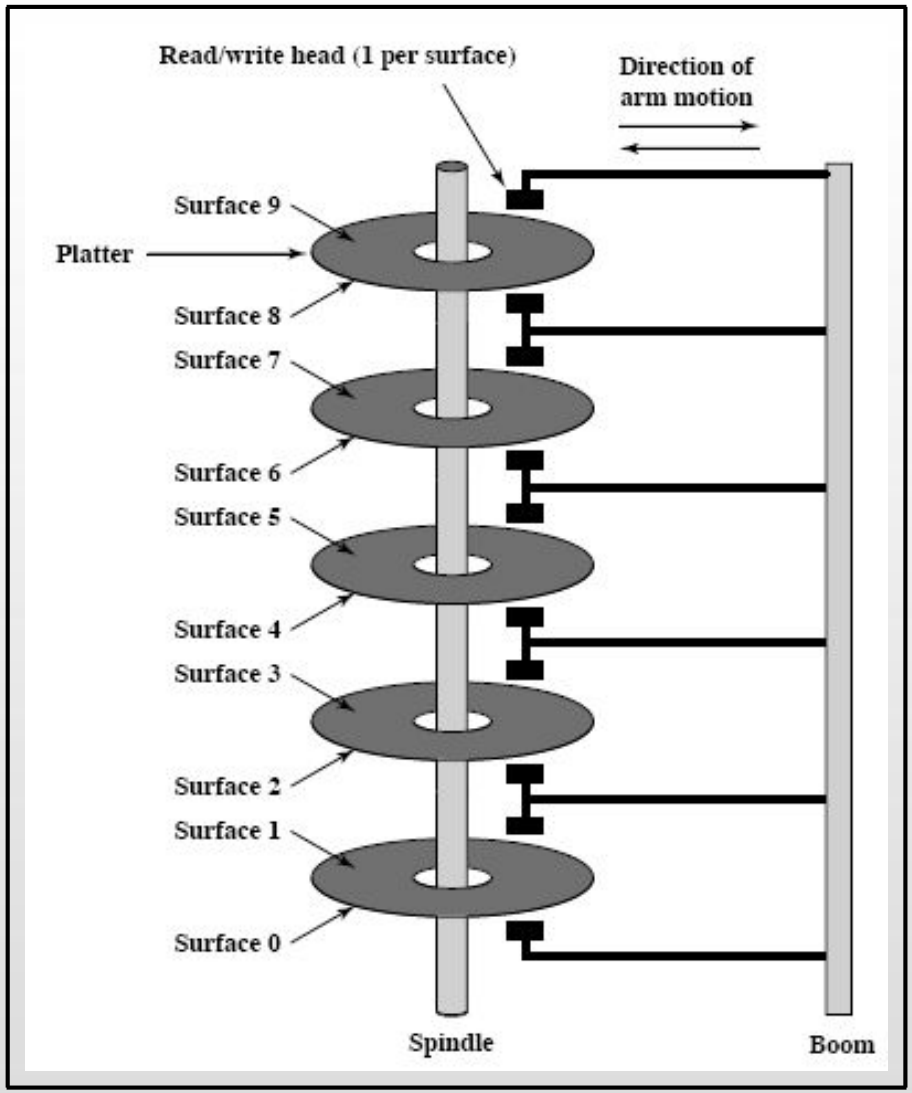




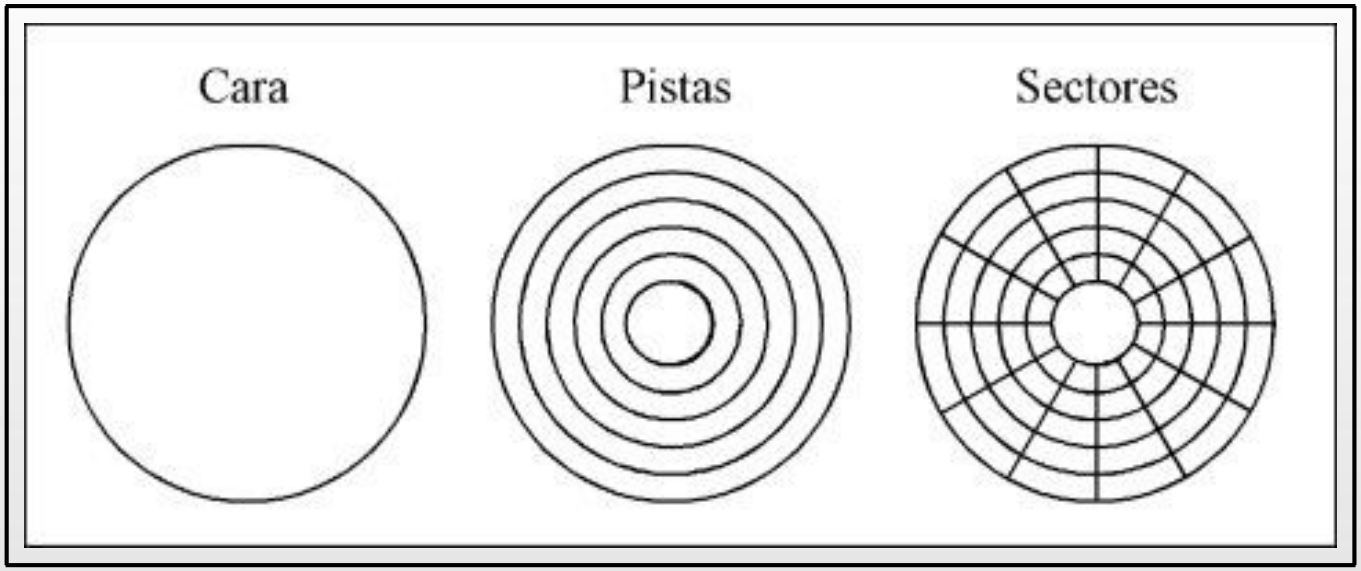
Conceptos para la instalación - **MBR**

- Sector reservado del disco físico (cilindro 0, cabeza 0, sector 1)
- Existe un MBR en todos los discos
- Si existiese más de un disco rígido en la máquina, solo uno es designado como *Primary Master Disk*

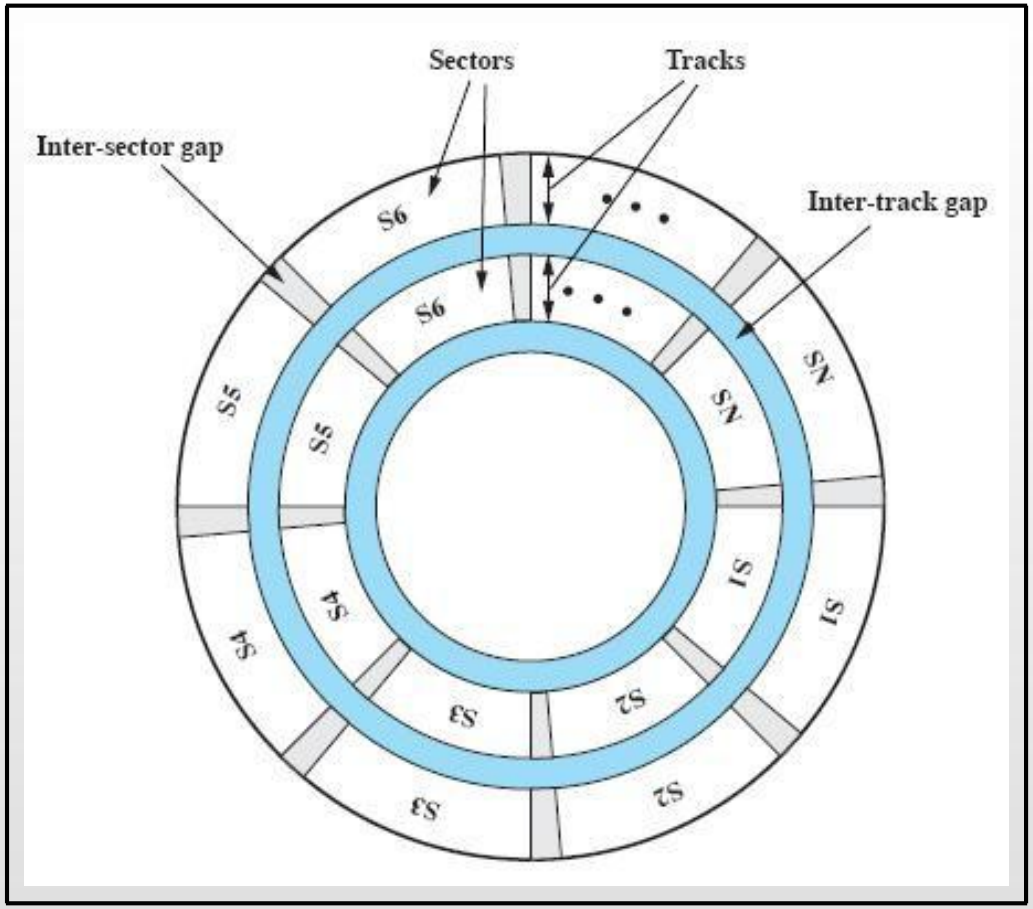
Organización física de discos



Organización física de discos (cont.)



Organización física de discos (cont.)



Conceptos para la instalación - MBR (cont.)

- El tamaño del MBR coincide con el tamaño estándar de sector, 512 bytes:
  - Los primeros bytes corresponden al *Master Boot Code* (MBC)
  - A partir del byte 446 se encuentra la tabla de particiones. Es de 64 bytes
  - Al final existen 2 bytes libres o para firmar el MBR
- Es creado con algún utilitario
- El MBC es un pequeño código que permite arrancar el SO
- La última acción del *BIOS* es leer el MBC. Lo lleva a memoria y lo ejecuta
- Si se tiene un sistema instalado → bootloader de MBC típico. Sino → uno diferente (*multietapa*)

Conceptos para la instalación - Particiones

- Es una forma de dividir lógicamente el disco físico:
  - *DOS* y *W95* no pueden manejar filesystems mayores a 2GB
  - Cada sistema operativo es instalado en una partición separada
  - Cada partición se formatea con un tipo de filesystem destino (*fat*, *ntfs*, *ext*, etc.)
  - Es una buena práctica separar los datos del usuario de la aplicaciones y/o sistema operativo instalado
  - Tener una partición de *restore* de todo es sistema
  - Poder ubicar el Kernel en una partición de solo lectura, o una que ni siquiera se monta (no está disponible para los usuarios)
  - Particionar demasiado un disco puede tener desventajas: ¡pensar..!

Conceptos para la instalación - Particiones (cont.)

- Debido al tamaño acotado en el MBR para la tabla de particiones:
  - Se restringe a 4 la cantidad de particiones primarias
  - 3 primarias y una extendida con sus respectivas particiones lógicas
- Una de las 4 particiones puede ser extendida, la cual se subdivide en volúmenes lógicos
- Partición primaria: división cruda del disco (puede haber 4 por disco). Se almacena información de la misma en el MBR
- Partición extendida: sirve para contener unidades lógicas en su interior. Solo puede existir una partición de este tipo por disco. No se define un tipo de FS directamente sobre ella
- Partición lógica: ocupa la totalidad o parte de la partición extendida y se le define un tipo de FS. Las particiones de este tipo se conectan como una lista enlazada

Conceptos para la instalación - Particiones (cont.)

- Como mínimo es necesario una partición (para el /)
- Es recomendable crear al menos 2 (/ y *SWAP*)
- Para crearlas, se utiliza software denominado **particionador**. Existen 2 tipos:
  - Destructivos: permiten crear y eliminar particiones (fdisk)
  - No destructivo: permiten crear, eliminar y modificar particiones (fips, gparted) ← generalmente las distribuciones permiten hacerlo desde la interfaz de instalación
- ¿Para qué podríamos crear otras particiones?

Nuestro ambiente de trabajo

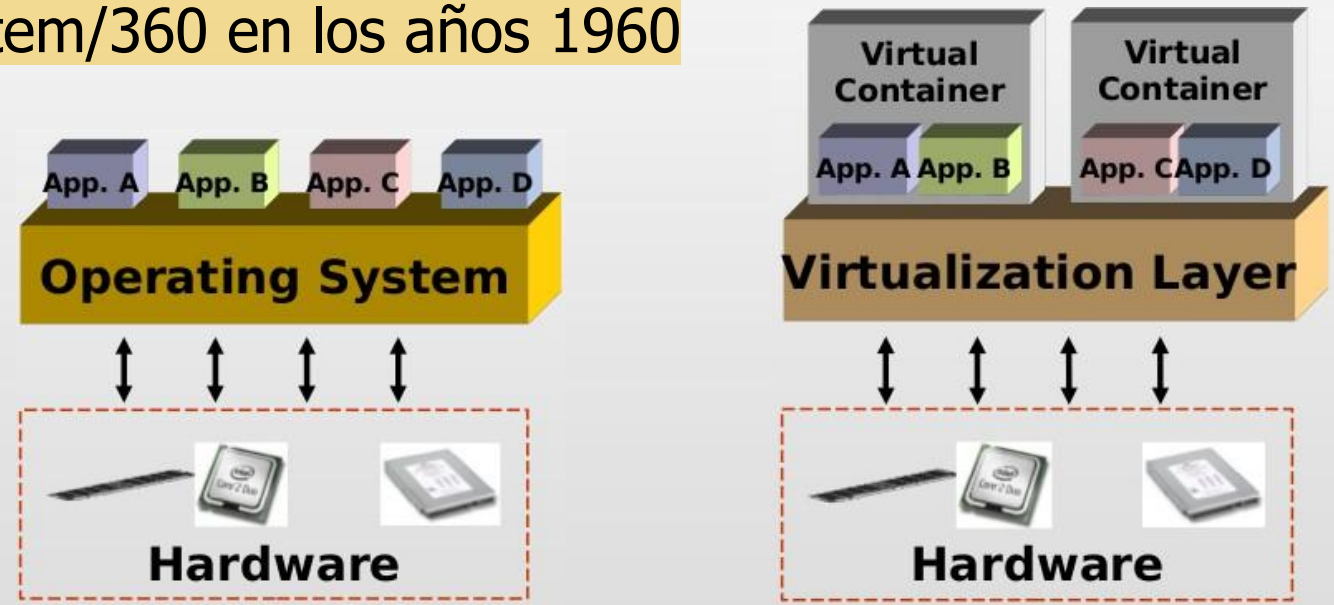
- Vamos a trabajar en un ambiente controlado → *VirtualBox*
- Necesitamos crear una maquina virtual y asignarle recursos
- Booteamos la máquina virtual iniciando desde algún medio de instalación
- Seguimos las instrucciones de instalación
- Verificamos que podamos arrancar el/los sistemas operativos instalados

- Particionando - 3 escenarios posibles:
  - Usar espacio libre no particionado
  - Usar particion no usada
  - Usar espacio libre de una partición activa (más complicado):
    - Cambio destructivo
    - Cambio no destructivo
- En nuestra instalación:
  - /dev/hda1: DOS con Windows (2 GB) ← ¡es mucho!
  - /dev/hda2: /boot → 60 MB aproximadamente
  - /dev/hda3: / → 6 GB aproximadamente
  - /dev/hda4: área de intercambio (SWAP)

Verificando la configuración de nuestro disco

- Desde Windows: utilizamos el administrador de discos
- Desde DOS: usamos fdisk
- Para instalar un nuevo SO necesitamos espacio libre sin particionar
- Si no lo tenemos, debemos generarlo. Para esto existen diversos escenarios:
  - Eliminar una partición existente
  - Redimensionar una partición
- ¿Qué ocurre en cada caso? ¿Qué software vamos a usar?

- Para los fines de este curso, permiten virtualizar plataformas
- Permite que en un equipo puedan correr varios SO en forma simultánea compartiendo recursos de hardware
- El pionero en esta tecnología fue *IBM* con el IBM System/360 en los años 1960



Emuladores/Virtualizadores (cont.)

- Básicamente se pueden considerar 3 tipos:
  - Emulación:
    - Emulan hardware
    - Tienen que implementar todas las instrucciones de la CPU
    - Es muy costosa y poco eficiente
    - Permite ejecutar arquitecturas diferentes a las soportadas por el hardware
  - Virtualización completa:
    - Permiten ejecutar SO huéspedes en un sistema anfitrión (host)
    - Utilizan en el medio un hypervisor o monitor de máquinas virtuales
    - El SO huésped debe estar soportado en la arquitectura anfitriona
    - Es más eficiente que la emulación (*Intel-VT* y *AMD-V*)
  - Paravirtualización:
    - Permite correr SOs modificados exclusivamente para actuar en entornos virtualizados
    - Mayor eficiencia que la virtualización

Emuladores/Virtualizadores (cont.)

- Las principales diferencias entre ellos son:
  - Los virtualizadores aprovechan el CPU sobre la que están trabajando, lo cual los hace más veloces
  - En un emulador se puede correr cualquier arquitectura. En un virtualizador solo se puede correr la arquitectura virtualizada

- La finalidad del bootloader es la de cargar una imagen de *Kernel* (sistema operativo) de alguna partición para su ejecución
- Se ejecuta luego del código del BIOS
- Existen 2 modos de instalación:
  - En el MBR (puede llegar a utilizar *MBR gap*)
  - En el sector de arranque de la partición raíz o activa (*Volume Boot Record*)
- *GRUB*, *LILO*, *NTLDR*, *GAG*, *YaST*, etc.

- **GRand Unified Bootloader**: gestor de arranque múltiple más utilizado
- En el MBR solo se encuentra la fase 1 del que solo se encarga de cargar la fase 1.5
- Fase 1.5: ubicada en los siguientes 30 KB del disco (*MBR gap*). Carga la fase 2
- Fase 2: interfaz de usuario y carga el Kernel seleccionado
- Se configura a través del archivo /boot/grub/menu.lst
- Algunas líneas:
  - default: define el SO por defecto a bootear
  - timeout: tiempo de espera para cargar el SO por defecto

```
title Debian GNU/Linux
root (hd0,1) #(Disco,Particion)
kernel /vmlinuz-2.6.26 ro quiet
root=/dev/hda3 initrd /initrd-2.6.26.img
```

- Utilizado en la mayoría de las distribuciones
- Dentro de sus mejoras, incluye el soporte a nuevas arquitecturas, soporte de caracteres no ASCII, idiomas, customización de menúes, etc.
- En Grub 2 la fase 1.5 ya no existe más
- El archivo de configuración ahora es /boot/grub/grub.cfg y no debería editarse manualmente → update-grub
- Más información en: https://help.ubuntu.com/community/Grub2

Proceso de arranque

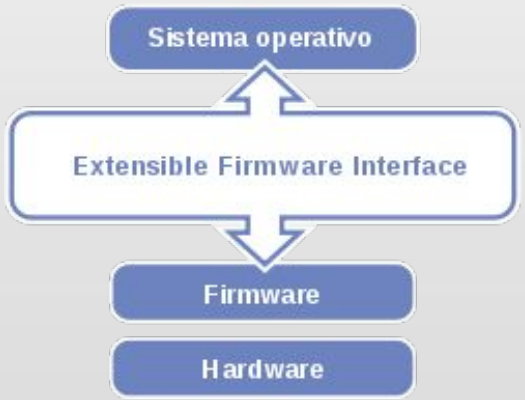
- Es el proceso de inicio de una máquina y carga del sistema operativo y se denomina *bootstrap*
- En las arquitecturas x86, el **BIOS (Basic I/O System)** es el responsable de iniciar la carga del SO a través del MBC:
  - Está grabado en un chip (*ROM*, *NVRAM*)
  - En otras arquitecturas también existe, pero se lo conoce con otro nombre:
    - Power on Reset + IPL en *mainframe*
    - OBP (OpenBoot PROM): en *SPARC*
- Carga el programa de booteo (desde el MBR)
- El gestor de arranque lanzado desde el MBC carga el Kernel:
  - Prueba y hace disponibles los dispositivos
  - Luego pasa el control al proceso **init**
- El proceso de arranque se ve como una serie de pequeños programas de ejecución encadenada

Problemas del arranque basado en BIOS

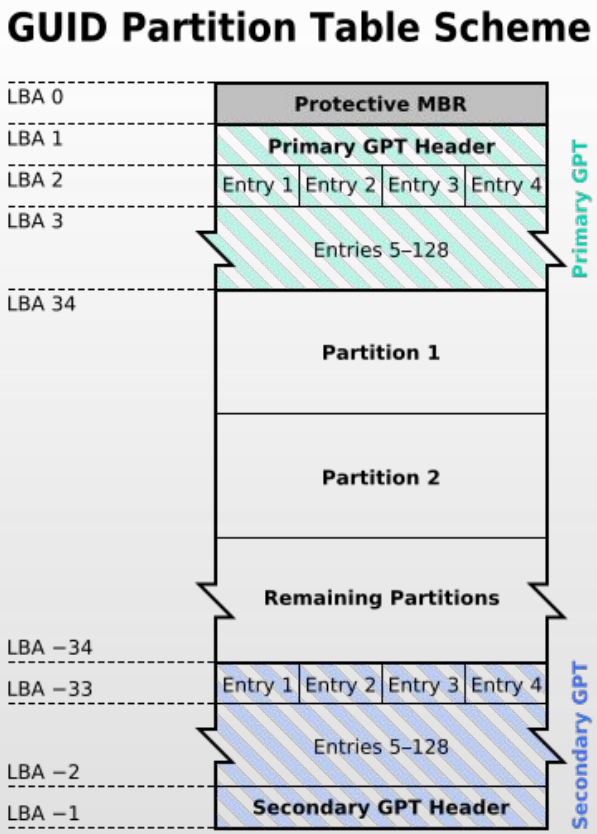
- Es de la década del 80
- La última acción del BIOS es leer el MBC del MBR
- El firmware del BIOS no facilita la lectura de filesystems
- El MBC no puede ocupar más de 446 bytes
  - Por ejemplo Grub, utiliza los 446 bytes del MBR y luego utiliza sectores del disco adyacentes que deberían estar libres (*MBR gap*)

Extensible Firmware Interface

- EFI: estándar para comunicación entre el SO y el firmware
- Utiliza el sistema **GPT** (GUID partition table) para solucionar limitaciones del MBR, como la cantidad de particiones
- GPT especifica la ubicación y formato de la tabla de particiones en un disco duro
- Es parte de EFI. Puede verse como una sustitución del MBR
- La especificación EFI es propiedad de Intel



- Se mantiene un MBR para tener compatibilidad con el esquema BIOS
- GPT usa un modo de direccionamiento lógico (logical block addressing **LBA**) en lugar de *cylinder-header-sector*
- El MBR "heredado" se almacena en el LBA 0
- En el LBA 1 está la cabecera GPT. La tabla de particiones en sí está en los bloques sucesivos
- La cabecera GPT y la tabla de particiones están escritas al principio y al final del disco (redundancia)


GUID Partition Table Scheme
LBA 0
LBA 1
LBA 2
LBA 3
LBA 34
LBA −34
LBA −33
LBA −2
LBA −1
Protective MBR
Primary GPT Header
Entry 1
Entry 2
Entry 3
Entry 4
Entries 5–128
Partition 1
Partition 2
Remaining Partitions
Entry 1
Entry 2
Entry 3
Entry 4
Entries 5–128
Secondary GPT Header
Primary GPT
Secondary GPT

- UEFI Forum
  - Alianza entre varias compañías con el objetivo de modernizar el proceso de arranque
  - Representantes de AMD, American Megatrends, Apple, HP, Dell, IBM, Insyde Software, Intel, Lenovo, Microsoft, Phoenix Technologies
  - EFI es propiedad de Intel
  - UEFI es propiedad del UEFI Forum
  - UEFI aporta criptografía, autenticación de red y una interface gráfica

UEFI (cont.)

- Define la ubicación de gestor de arranque
- Define la interfaz entre el gestor de arranque y el firmware
- Expone información para los gestores de arranque con:
  - Información de hardware y configuración del firmware
  - Punteros a rutinas que implementan los servicios que el firmware ofrece a los bootloaders u otras aplicaciones UEFI
  - Provee un *BootManager* para cargar aplicaciones UEFI (e.j.: Grub) y drivers desde un UEFI filesystem
  - El booloader ahora es un tipo de aplicación UEFI:
    - El Grub sería una aplicación UEFI, que reside en el UEFI filesystem donde están los drivers necesarios para arrancar el sistema operativo (FAT32)
    - Para el Grub deja de ser necesario el arranque en varias etapas.

- Propone mecanismos para un arranque libre de código malicioso
- Las aplicaciones y drivers UEFI (imágenes UEFI) son validadas para verificar que no fueron alteradas
- Se utilizan pares de claves asimétricas
- Se almacenan en el firmware una serie de claves públicas que sirven para validar que las imágenes están firmadas por un proveedor autorizado
- Si la clave privada está vencida o fue revocada la verificación puede fallar

¿Preguntas?

Introducción a GNU/Linux (continuación)

Explicación de práctica

Introducción a los Sistemas Operativos
Conceptos de Sistemas Operativos

Facultad de Informática Universidad Nacional de La Plata

2025

Características - Configuración de discos

- Configuración de discos IDE (*Integrated Device Electronics*):
  - Master o Slave
  - Primer y Segundo bus IDE
- Denominación de los discos basada en los buses:
  - **/dev/hd<u>a</u>**: configurado como Master en el 1º bus IDE
  - **/dev/hd<u>b</u>**: configurado como Slave en el 1º bus IDE
  - **/dev/hd<u>c</u>**: configurado como Master en el 2º bus IDE
  - **/dev/hd<u>d</u>**: configurado como Slave en el 2º bus IDE
- Particiones primarias → 1 a 4
- Particiones lógicas → de 5 en adelante

Características - Configuración de discos

- Configuración de discos SCSI (*Small Computer System Interface*):
  - se basa en *LUN*.
- Denominación de los discos basada en la identificación de los buses:
  - **/dev/sd<u>a</u>**
  - **/dev/sd<u>b</u>**
  - **/dev/sd<u>c</u>**
  - **/dev/sd<u>d</u>**
  - **...**
- Particiones primarias:
  - Se numeran de la 1 a la 4.
  - Sólo estas se pueden marcar como activas (booteables).
- Particiones extendidas:
  - Sus particiones lógicas se numeran a partir de la 5.

Características - Configuración de discos

- Configuración de discos SATA (*Serial Advanced Technology Attachment*):
  - Denominación de los discos basada en la identificación de los buses, al igual que SCSI (/dev/sda, /dev/sdb, …).
- Particiones primarias:
  - Se numeran de la 1 a la 4.
  - Sólo estas se pueden marcar como activas (booteables).
- Particiones extendidas:
  - Sus particiones lógicas se numeran a partir de la 5.

Características - Configuración de discos

- Nueva nomenclatura utilizada:
  - Con la evolución de las distribuciones GNU/Linux, se comenzó a utilizar **"udev"** (*.rules*) como gestor de dispositivos:
    - Su función es controlar dinámicamente los archivos que hay en /dev, solo en base al hardware detectado.
    - Soporta **Persistent Device Naming**.
    - Motiva su uso, el no poder garantizar que tras distintos arranques del SO, los dispositivos se sigan llamando de la misma manera.
    - Reemplaza a *devfs* y *hotplug*.
    - No se basa en **Major** y **Minor Number**.
    - Se basa en eventos y permite que nuevos dispositivos sean agregados luego del arranque.

Características - Configuración de discos

- Desde Debian/Squeeze todos los dispositivos llamados **hdX** se pasaron a denominar **sdX**.
- Por esta y otras razones se adoptan 4 mecanismos nuevos para nomenclar:
  - Nombres persistentes por **UUID** (Universal Unique Identifier):

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

  - Utilizando **labels**

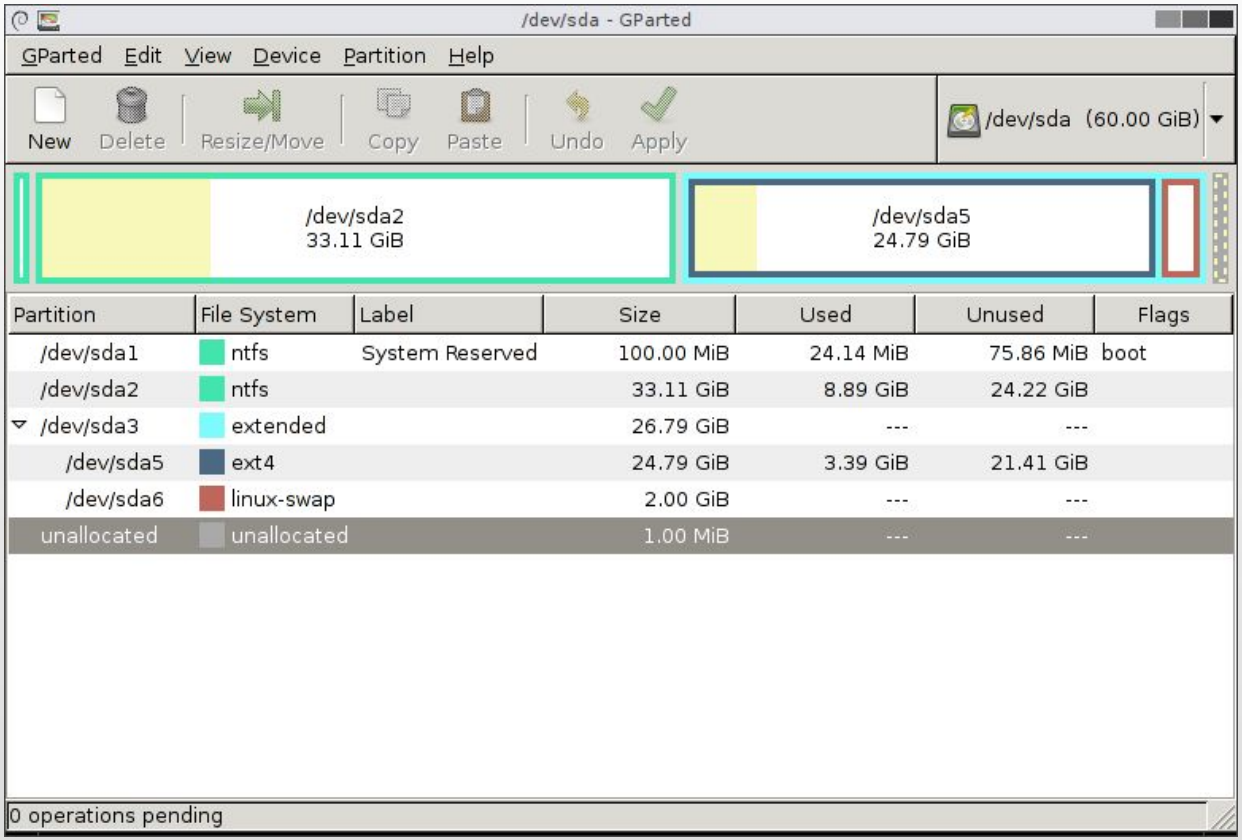
```
$ ls -l /dev/disk/by-label
data -> ../../sdb2
data2 -> ../../sda2
```

Soporte de instalación

- Existen diversos modos de instalar GNU/Linux:
  - Debemos tener en cuenta la arquitectura de hardware:
    - **amd64**: Arquitectura de 64 bits
    - **arm / armel**: Advanced Risc Machine
    - **i386**: Arquitectura de 32 bits
    - **ia64**: Intel Itanium o Intel Architecture-64
    - Otras muchas más
  - Podemos instalarlo desde una imagen ISO descargada de la web.
  - Podemos instalarlo desde un USB.
    - UNetbootin permite crear instaladores o LiveCD utilizando unidades USB.

Herramientas para particionar

- El particionado de un disco se lo puede realizar mediante:
  - Software **destructivo**: *fdisk*.
  - Software **no destructivo**: *fips*, *gparted*.



GNU/Linux: Características

- No existe el concepto de *extensión* en el nombre de un archivo.
- Los subdirectorios no se separan con el caracter \ (contrabarra) - se usa / (barra).
- Es case sensitive.
- En la línea de comandos (*CLI*), los elementos o tokens de una instrucción se separan utilizando espacios en blanco. Por ejemplo, entre un comando y sus parámetros debemos dejar obligatoriamente un espacio en blanco.
- Existe una separación entre el entorno gráfico y el de texto (*tty*).

*GNU/Linux: Editor de texto **vim***

- Presente en la mayoría de las distribuciones.
- Posee 3 modos de ejecución:
  - Modo Insert (**i**)
  - Modo Visual (**v**)
  - Modo de Órdenes o Normal (**Esc**)
- 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

GNU/Linux: Otros editores de texto

- El editor que se usa es siempre una decisión personal.
- La curva de aprendizaje de editores como **vim** o **emacs** puede ser un desafío al comienzo.
- Si resulta muy complejo para comenzar, siempre se pueden utilizar alternativas menos potentes pero más sencillas, como:
  - **nano**
  - **micro**
  - **ne** (nice editor)

Usuarios

- Todo usuario debe poseer credenciales para acceder al sistema
  - **root**: es el administrador del sistema (superusuario)
  - usuarios regulares del sistema
  - usuarios regulares del sistema que pueden realizar tareas de superusuario (**/etc/sudoers**)
- Los usuarios se identifican con un nombre de usuario, que es único en el sistema.
- Cada usuario pertenece a uno o más *grupos*.
- Los usuarios pueden tener un directorio personal o *home*.

Usuarios

- La información de los usuarios se almacena en diferentes archivos de configuración:

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

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

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

Usuarios

- Comandos para el manejo de los usuarios y grupos del sistema:
  - **useradd** nombre_usuario:
    - Agrega el usuario
    - Modifica los archivos /etc/passwd
    - Alternativa → **adduser**
  - **passwd** nombre_usuario:
    - Asigna o cambia la contraseña del usuario
    - Modifica el archivo /etc/shadow
  - **usermod** nombre_usuario:
    - **-g**: modifica grupo primario (Modifica /etc/passwd)
    - **-G**: modifica grupos adicionales (Modifica /etc/group)
    - **-d**: modifica el directorio *home* (Modifica /etc/passwd)
  - **userdel** nombre_usuario: elimina el usuario
  - **groupdel** nombre_grupo: elimina el grupo

Permisos

- Se aplican a nivel de sistema de archivos (directorios y archivos).
- Existen 3 posibles permisos: R, W y X.
- Se suelen expresar en octal acorde a la siguiente tabla:

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

- Las combinaciones de permisos se expresan como la suma de los valores octales de cada permiso que se desea incluir:
  - **W + X** = 2 + 1 = **3**
  - **R + W** = 4 + 2 = **6**

Permisos

- Cada archivo o directorio del sistema de archivos tiene un usuario dueño y un grupo dueño.
- Los permisos pueden definirse sobre las siguientes agrupaciones de usuarios:
  - **Usuario**: aplicables al usuario dueño (**U**).
  - **Grupo**: aplicables a cualquier usuario perteneciente al grupo dueño que no sea el usuario dueño (**G**).
  - **Otros**: aplicables a cualquier otro usuario que no pertenezca a las agrupaciones anteriores (**O**).
- Los permisos de un archivo o directorio se gestionan mediante el comando **chmod**:

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

Entorno

- Algunos comandos útiles:
  - **ls**
  - **cd**
  - **mkdir**
  - **rmdir**
  - **rm**
  - **mv**
  - **cp**
  - **man**
  - **info**

Proceso de arranque: ***Bootloader***

- El Bootloader o cargador de arranque es un programa que permite cargar e iniciar el Sistema Operativo. Puede llegar a cargar un entorno previo a iniciar el Sistema Operativo.
- Generalmente se utilizan los cargadores multietapas, en los que varios programas pequeños se van invocando hasta lograr la carga del Sistema Operativo.
- En cierto sentido, el código del *BIOS/UEFI* forma parte del bootloader, pero el concepto está m´as orientado al código que reside en el *Master Boot Record* (MBR, 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 de cada uno.

Proceso de arranque: ***SysV init***

1. Se empieza a ejecutar el código del BIOS.
2. El BIOS ejecuta el POST.
3. El BIOS lee el sector de arranque (MBR).
4. Se carga el gestor de arranque (MBC).
5. El bootloader carga el *kernel* y el *initrd* (*initial ram disk*).
6. Se monta el *initrd* como sistema de archivos raíz y se inicializan componentes esenciales (por ejemplo, el *scheduler*).
7. El Kernel ejecuta el proceso *init* y se desmonta el *initrd*.
8. Se lee el /etc/inittab.
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 ser usado.

*Proceso de arranque: SysV init - El proceso **init***

1. Su función es cargar todos los subprocesos necesarios para el correcto funcionamiento del Sistema Operativo.
2. El proceso *init* (ejecutado desde /sbin/init) posee el PID 1.
3. En SysV init se lo configura a través del archivo /etc/inittab.
4. No tiene padre y es el padre de todos los procesos (**pstree**).
5. Es el encargado de montar los filesystems y de hacer disponible los demás dispositivos.

*Proceso de arranque: SysV init - **Runlevels***

- Es el modo en que arranca GNU/Linux (por defecto: Runlevel 3 en Redhat y Runlevel 2 en Debian).
- El proceso de arranque se divide en niveles.
- Cada runlevel es responsable de iniciar o parar una serie de servicios, ya sea al entrar al Runlevel (arranque) o al salir de éste (apagado).
- Acorde al estándar, existen 7 (numerados del 0 al 6):
  - 0 → halt (parada o apagado).
  - 1 → single-user mode (modo monousuario).
  - 2 → multi-user without network support (multiusuario sin soporte de red).
  - 3 → multi-user console mode (modo multiusuario en consola).
  - 4 → N/A (no se utiliza).
  - 5 → X11 (modo multiusuario con entorno gráfico basado en X.org).
  - 6 → reboot (reinicio).

*Proceso de arranque: SysV init - **Runlevels***

- Se encuentran definidos en el archivo /etc/inittab:
  id:runlevels:acción:proceso
  - **id**: identifica la entrada en inittab (1 a 4 caracteres).
  - **runlevels**: el/los runlevels en los que se realiza la acción
  - **acción**: indica cómo se ejecutará **proceso**
    - **wait**, **initdefault**, **ctrlaltdel**, **off**, **respawn**, **once**, **sysinit**, **boot**, **bootwait**, **powerwait**, etc.
  - **proceso**: el comando exacto que será ejecutado.

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

*Proceso de arranque: SysV init - **Runlevels***

- Los scripts que se ejecutan se suelen guardar en **/etc/init.d**.
- En **/etc/rcX.d** (donde **X** es el número de runlevel entre 0 y 6) se hacen links simbólicos a los archivos que hay en **/etc/init.d**.
- Los nombres de los links siguen este patrón:

  [S|K]<orden><nombre>

  S indica que se debe iniciar el script (se invoca con el argumento **start**).

  K indica que se debe para el script (se invoca con el argumento **stop**).

  <orden> es un valor numérico en dos dígitos para garantizar el orden de ejecución de los scripts. Puede verse como una prioridad.

  <nombre> es el nombre lógico con que identificamos el script, no es relevante para la ejecución en sí.

```
$ ls -1 /etc/rcS.d/
S55urandom S70x11-common
```

Proceso de arranque: SysV init - comando ***insserv***

- Se utiliza para administrar el orden de los enlaces simbólicos de los directorios **/etc/rcX.d**, resolviendo las dependencias de forma automática.
- Utiliza cabeceras en los scripts de /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/remover los links simbólicos.

Proceso de arranque: SysV init - comando ***insserv***

- Las dependencias se especifican mediante *facilities* → **Provides** keyword.
- Las facilities que comienzan con $ se reservan para el sistema (**$syslog**).
- Los scripts deben cumplir con el estándar *LSB init*:
  - Proveer al menos **start**, **stop**, **restart**, **force-reload** y **status**.
  - Retornar un código de salida apropiado.
  - Declarar sus dependencias.

*Proceso de arranque: SysV init - comando **insserv***

- Ejemplo de un encabezado de script LSB init:

```
### BEGIN INIT INFO
# Provides:          scriptname
# Required-Start:    $remote_fs $syslog
# Required-Stop:     $remote_fs $syslog
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Start daemon at boot time
# Description:       Enable service provided by daemon.
### END INIT INFO
```

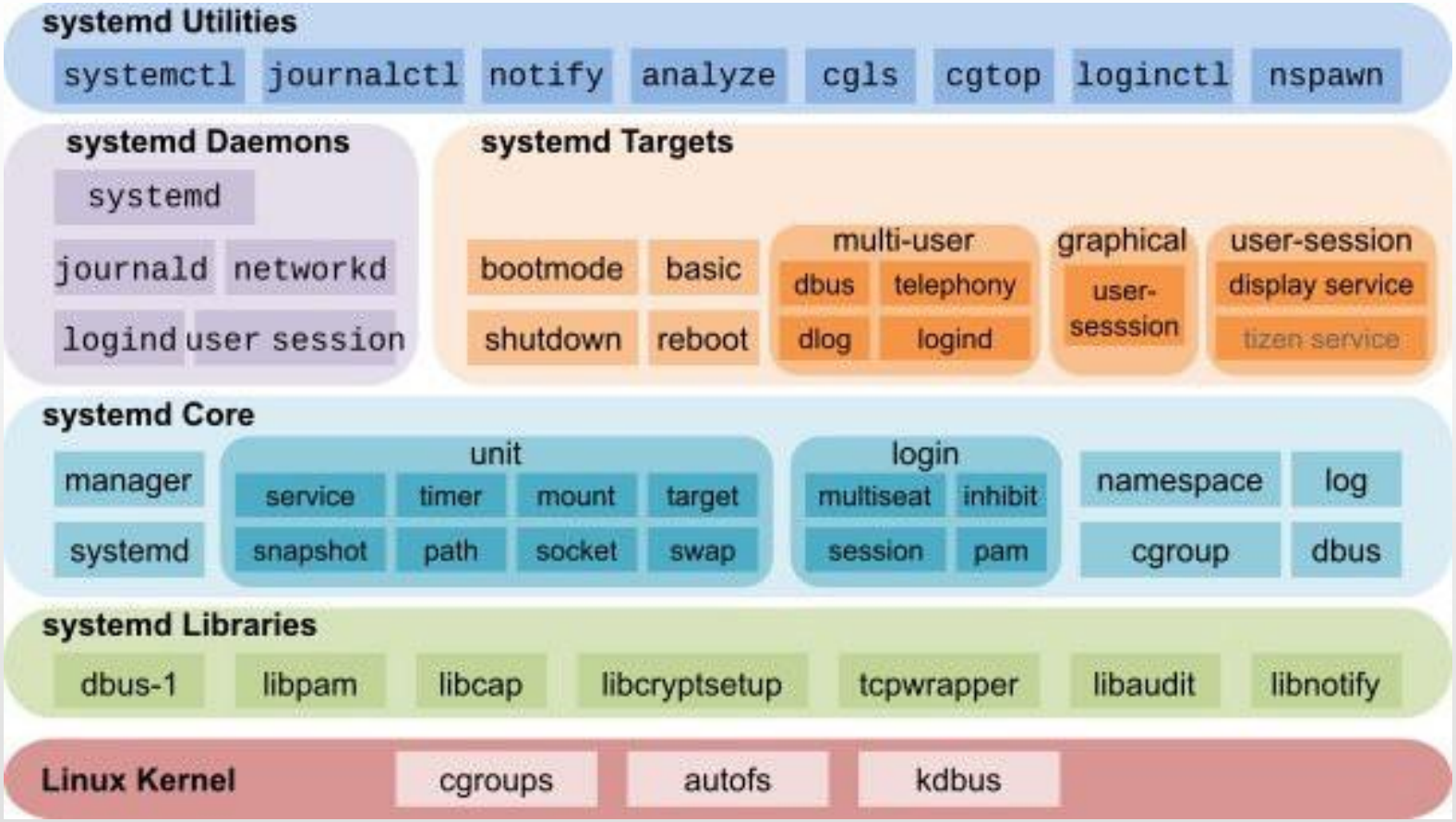
Proceso de arranque: ***SystemD***

- Es un sistema que centraliza la administración de demonios (servicios) y librerías del sistema.
- Mejora el paralelismo de arranque.
- Puede ser controlado con el comando **systemctl**.
- Compatible con SysV init → si es llamado como *init*.
- El demonio *systemd* reemplaza al proceso *init* y es el que tiene PID 1.
- Los runlevels son reemplazados por **targets**.
- No utiliza el archivo de configuración **/etc/inittab**.

Proceso de arranque: ***SystemD***

- Las unidades de trabajo son denominadas **units** y tienen distintos tipos, siendo los más relevantes:
  - **Service**: controla un servicio particular (*.service*).
  - **Socket**: encapsula *IPC*, un socket del sistema o file system *FIFO* (*.socket*) → *socket-based activation*.
  - **Target**: agrupa *units* o establece puntos de sincronización durante el arranque (*.target*) → dependencia de unidades.
  - **Snapshot**: almacena el estado de un conjunto de unidades para que pueda ser restablecido más tarde (*.snapshot*).
- Las *units* pueden tener dos estados → **active** o **inactive**.

Proceso de arranque: ***SystemD***



Proceso de arranque: SystemD - ***activación por socket***

- No todos los servicios que se inician en el booteo se utilizan:
  - Impresoras
  - Servidor en el puerto 80
  - etc.
- Es un mecanismo de inicio de servicios bajo demanda → podemos atender servicios sin que estén iniciados hasta el momento en que realmente se los necesita.
- Cuando el socket recibe una conexión, inicia el servicio y le pasa el socket para que reciba la petición.
- No hay necesidad de definir dependencias entre servicios → se inician todos los sockets en primer medida.

*Proceso de arranque: SystemD - **cgroups***

- Permite organizar un grupo de procesos en forma jerárquica.
- Agrupa conjuntos de procesos relacionados (por ejemplo, un servidor web NGINX con sus procesos dependientes)
- Tareas que realiza:
  - Seguimiento mediante subsistema *cgroups* → no se utiliza el PID individual → doble *fork* no funciona para escapar de SystemD.
  - Limita el uso de recursos (CPU, memoria, discos, etc).
  - etc.

Proceso de arranque: ***fstab***

- Define qué particiones se montan al arranque.
- Algunas opciones:
  - **user**: cualquier usuario puede montar la partición.
  - **auto**: monta la partición al inicio.
  - **ro**: read only, **rw**: read and write.
- 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
```

¿Preguntas?

Administración de Procesos

Explicación de práctica

Introducción a los Sistemas Operativos
Conceptos de Sistemas Operativos

Facultad de Informática Universidad Nacional de La Plata

2025

Tiempos de los procesos

- **CPU (T_{CPU}):** tiempo que efectivamente usa la CPU el proceso.
- **Retorno (T_R)**: tiempo que transcurre entre que el proceso llega al sistema hasta que completa su ejecución.
- **Espera (T_E)**: tiempo que el proceso se encuentra en el sistema esperando, es decir el tiempo que pasa sin ejecutarse (T_R - T_{CPU})
- **Promedios (TPR y TPE)**: tiempos promedio de Retorno y Espera. Promedio calculado de los tiempos individuales de cada proceso del lote.

Algoritmo **FIFO**

- *First come, first served*.
- Cuando hay que elegir un proceso para ejecutar, se selecciona el más antiguo.
- No favorece a ningún tipo de procesos, pero en principio podríamos decir que los *CPU Bound* terminan al comenzar su primer ráfaga, mientras que los *I/O Bound* requieren múltiples ráfagas.

Algoritmo **FIFO**

Job	Llegada	CPU	Prioridad
1	0	9	3
2	1	5	2
3	2	3	1
4	3	7	2

```
#Ejemplo 1
TAREA ''1'' PRIORIDAD=3
INICIO=0 [CPU,9]
TAREA ''2'' PRIORIDAD=2
INICIO=1 [CPU,5]
TAREA ''3'' PRIORIDAD=1
INICIO=2 [CPU,3]
TAREA ''4'' PRIORIDAD=2
INICIO=3 [CPU,7]
```

¿Cuáles serían los tiempos de retorno y espera?

Algoritmo **SJF**

- Shortest Job First
- Política *non preemptive* que selecciona el proceso con la ráfaga más corto
- Cálculo basado en la ejecución previa
- Procesos cortos se colocan delante de procesos largos
- Los procesos largos pueden sufrir *starvation* (inanicion)
- Veamos el ejemplo anterior

Algoritmo **RR**

- Round Robin
- Política basada en un reloj
- **Quantum (Q)**: medida que determina cuánto tiempo podrá usar el procesador cada proceso:
  - Pequeño: overhead de *context switch*
  - Grande: ¿pensar?
- Cuando un proceso es expulsado de la *CPU* es colocado al final de la *Ready Queue* y se selecciona otro (*FIFO circular*)

Algoritmo **RR** *(cont.)*

- Existe un "contador" que indica las unidades de CPU en las que el proceso se ejecutó. Cuando el mismo llega a 0 el proceso es expulsado
- El "contador" puede ser:
  - Global
  - Local → *PCB*
- Existen dos variantes con respecto al valor inicial del "contador" cuando un proceso es asignado a la CPU:
  - **Timer Variable**
  - **Timer Fijo**

Algoritmo **RR - Timer Variable**

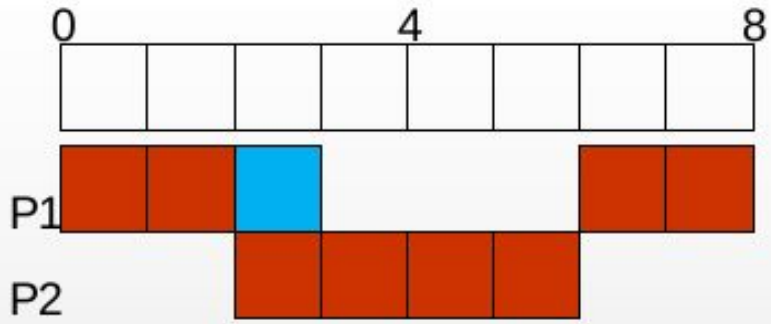
- El "contador" se inicializa en Q (contador := Q) cada vez que un proceso es asignado a la *CPU*
- Es el más utilizado
- Utilizado por el simulador
- Veamos el ejemplo 1 nuevamente

Algoritmo **RR - Timer Fijo**

- El “contador” se inicializa en Q cuando su valor es cero
  - if (contador == 0) contador = Q;
- Se puede ver como un valor de Q compartido entre los procesos

Algoritmo **RR** *(cont.)*

Timer Variable

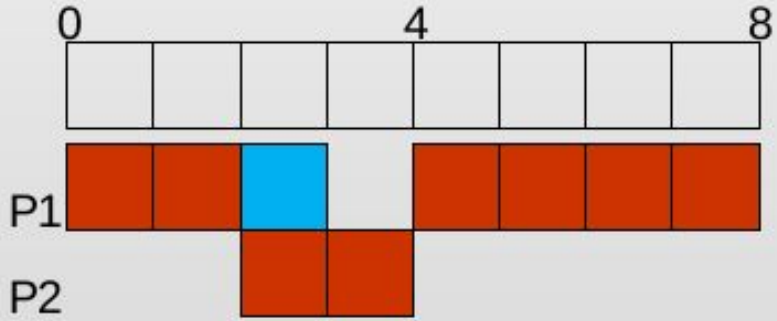


Round Robin, Q=4

= E/S

= Uso de CPU

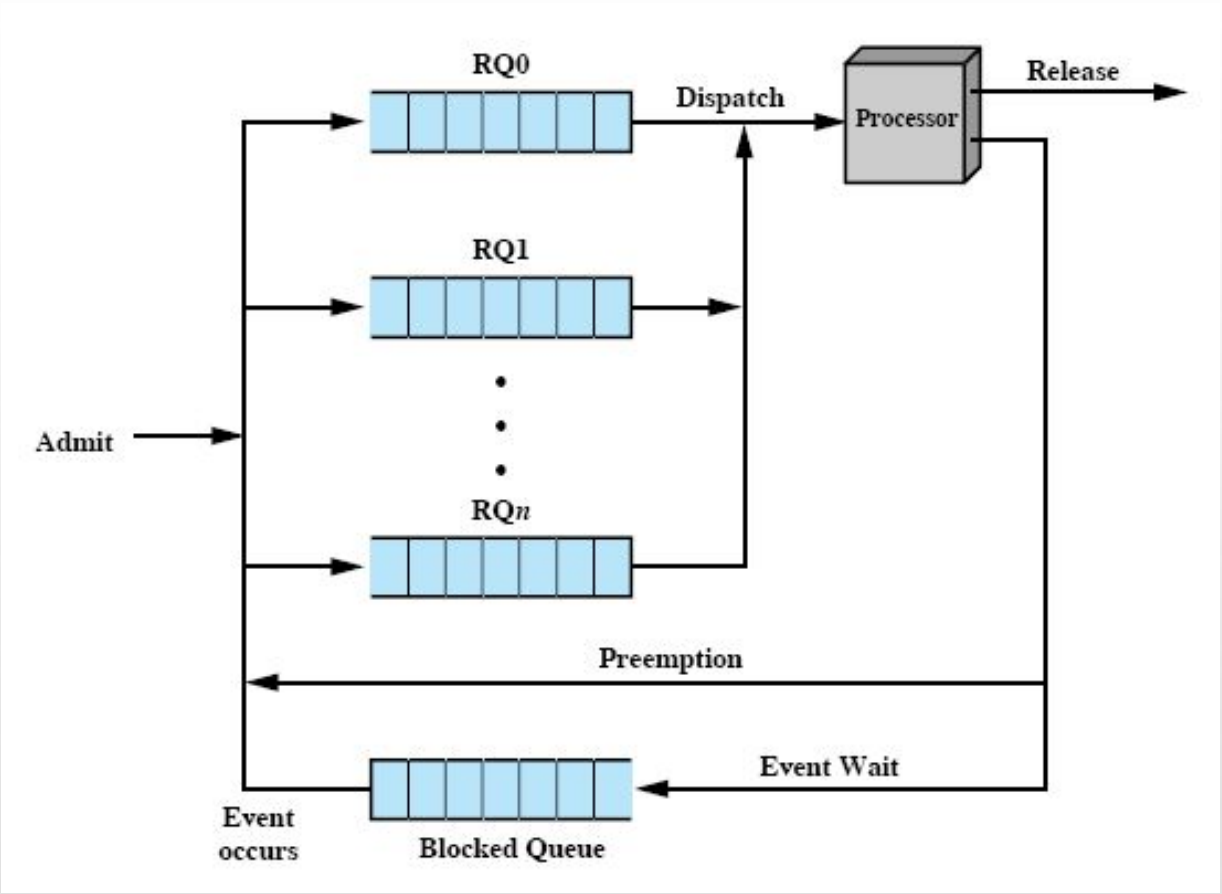
Timer Fijo



Algoritmo con **Uso de Prioridades**

- Cada proceso tiene un valor que representa su prioridad → menor valor, mayor prioridad
- Se selecciona el proceso de mayor prioridad de los que se encuentran en la *Ready Queue*
- Existe una *Ready Queue* por cada nivel de prioridad
- Procesos de baja prioridad pueden sufrir *starvation* (inanicion)
  - Solucion: permitir a un proceso cambiar su prioridad durante su ciclo de vida → **Aging o Penalty**
- Puede ser un algoritmo **preemptive** o no
- Veamos el ejemplo 1 nuevamente

Algoritmo con Uso de Prioridades (cont.)



Algoritmo SRTF

- Shortest Remaining Time First
- Versión *preemptive* de *SJF*
- Selecciona el proceso al cual le resta menos tiempo de ejecución en su siguiente ráfaga.
- ¿A qué tipos de procesos favorece? → **I/O Bound**
- Veamos el ejemplo 1 nuevamente

Algoritmo SRTF

- Shortest Remaining Time First
- Versión *preemptive* de *SJF*
- Selecciona el proceso al cual le resta menos tiempo de ejecución en su siguiente ráfaga.
- ¿A qué tipos de procesos favorece? → **I/O Bound**
- Veamos el ejemplo 1 nuevamente

Algoritmo SRTF

- Shortest Remaining Time First
- Versión *preemptive* de *SJF*
- Selecciona el proceso al cual le resta menos tiempo de ejecución en su siguiente ráfaga.
- ¿A qué tipos de procesos favorece? → **I/O Bound**
- Veamos el ejemplo 1 nuevamente

Algoritmos de planificación - CPU + I/O

- Ciclo de vida de un proceso: uso de CPU + operaciones de I/O
- Cada dispositivo tiene su cola de procesos en espera → un scheduler por cada cola
- Se considera I/O independiente de la CPU (DMA, PCI, etc.) → uso de CPU y operaciones de I/O en simultáneo

Algoritmos de planificación - Criterios de desempate

- Orden de aplicación:
  - Orden de llegada de los procesos
  - **PID** de los procesos
- Siempre se mantiene la misma política

Algoritmos de planificación - Un recurso por proceso

Job	Llegada	CPU	E/S (rec., inst., dur.)
1	0	5	(R1, 3, 2)
2	1	4	(R2, 2, 2)
3	2	3	(R3, 2, 3)

```
#Ejemplo 2
RECURSO ''R1''
RECURSO ''R2''
RECURSO ''R3''
TAREA ''1'' INICIO=0
[CPU,3] [1,2] [CPU,2]
TAREA ''2'' INICIO=1
[CPU,2] [2,2] [CPU,2]
TAREA ''3'' INICIO=2
[CPU,2] [3,3] [CPU,1]
```

Algoritmos de planificación - Recurso compartido

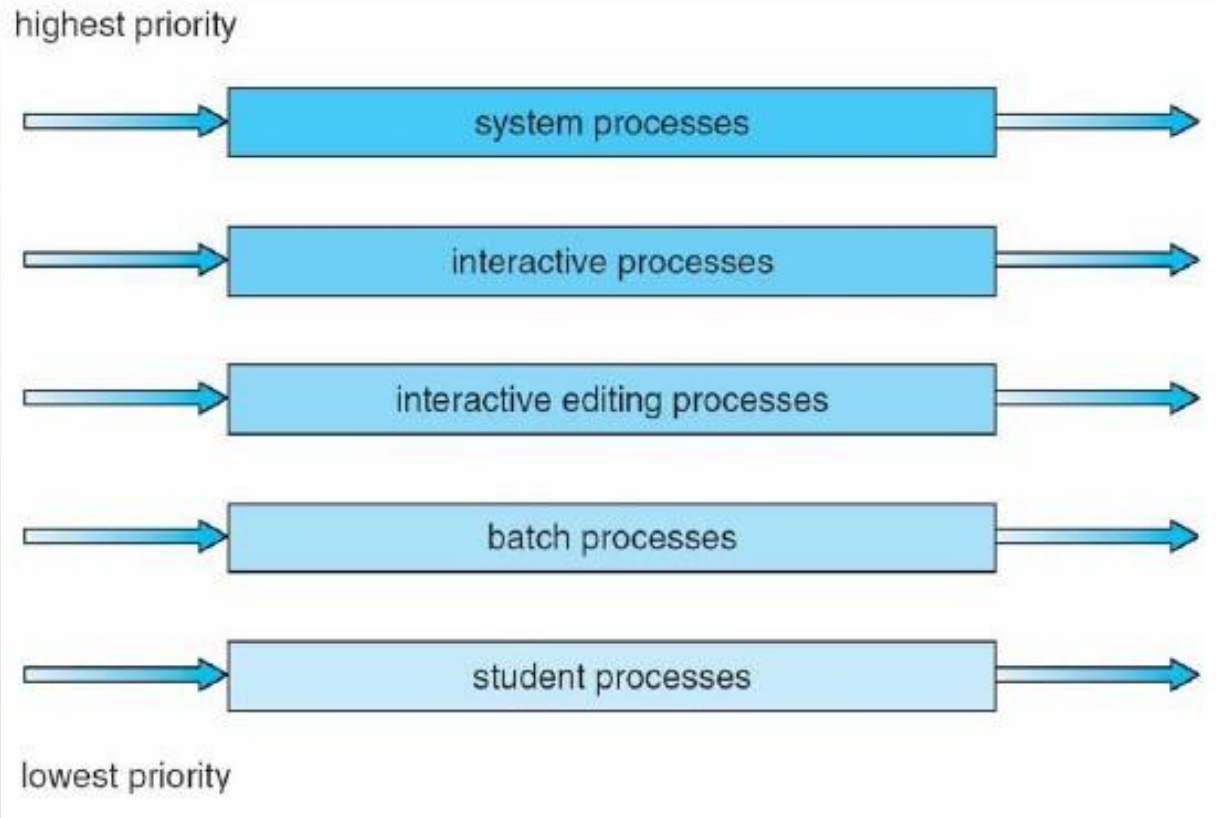
Job	Llegada	CPU	E/S (rec., inst., dur.)
1	0	5	(R1, 3, 3)
2	1	4	(R1, 1, 2)
3	2	3	(R2, 2, 3)

```
#Ejemplo 3
RECURSO ''R1''
RECURSO ''R2''
TAREA ''1'' INICIO=0
[CPU,3] [1,3] [CPU,2]
TAREA ''2'' INICIO=1
[CPU,1] [1,2] [CPU,3]
TAREA ''3'' INICIO=2
[CPU,2] [2,3] [CPU,1]
```

Esquema **Colas Multinivel**

- Schedulers actuales → combinación de algoritmos vistos
- La *ready queue* es dividida en varias colas (similar a prioridades)
- Los procesos se colocan en las colas según una clasificación que realice el sistema operativo
- Cada cola posee su propio algoritmo de planificación → **planificador horizontal**
- A su vez existe un algoritmo que planifica las colas → **planificador vertical**
- Retroalimentación → un proceso puede cambiar de una cola a la otra

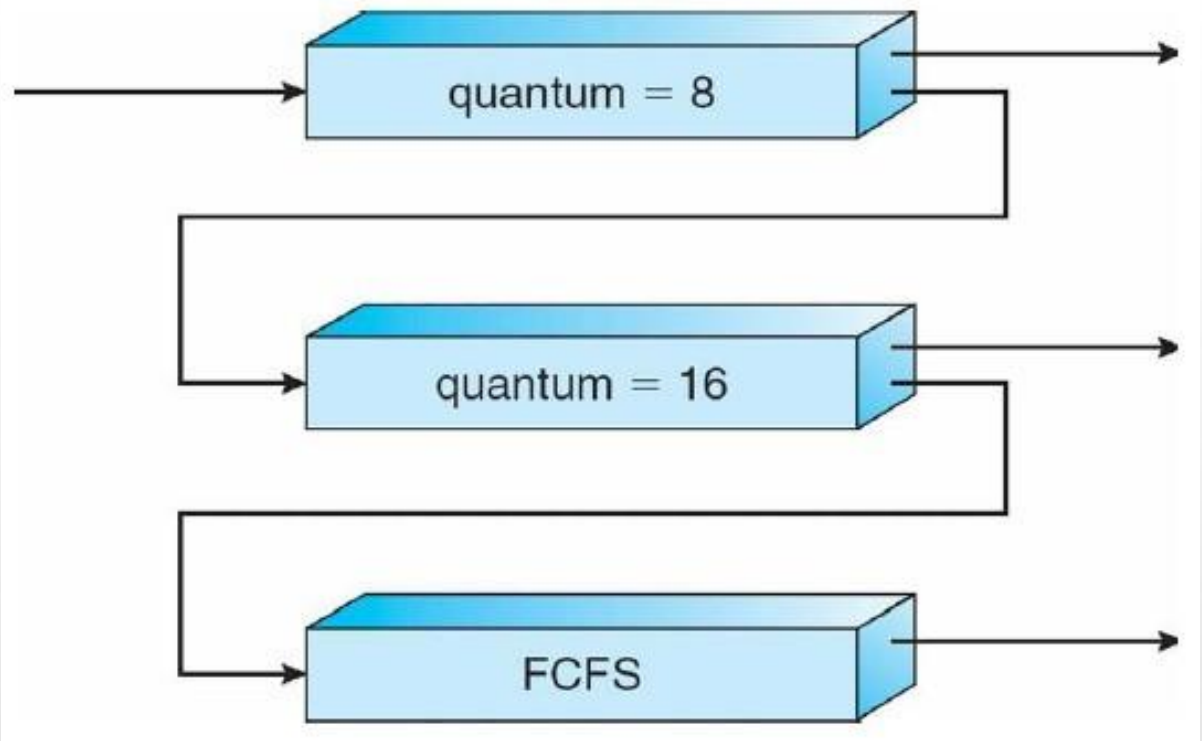
Esquema **Colas Multinivel** *(ejemplo 1)*



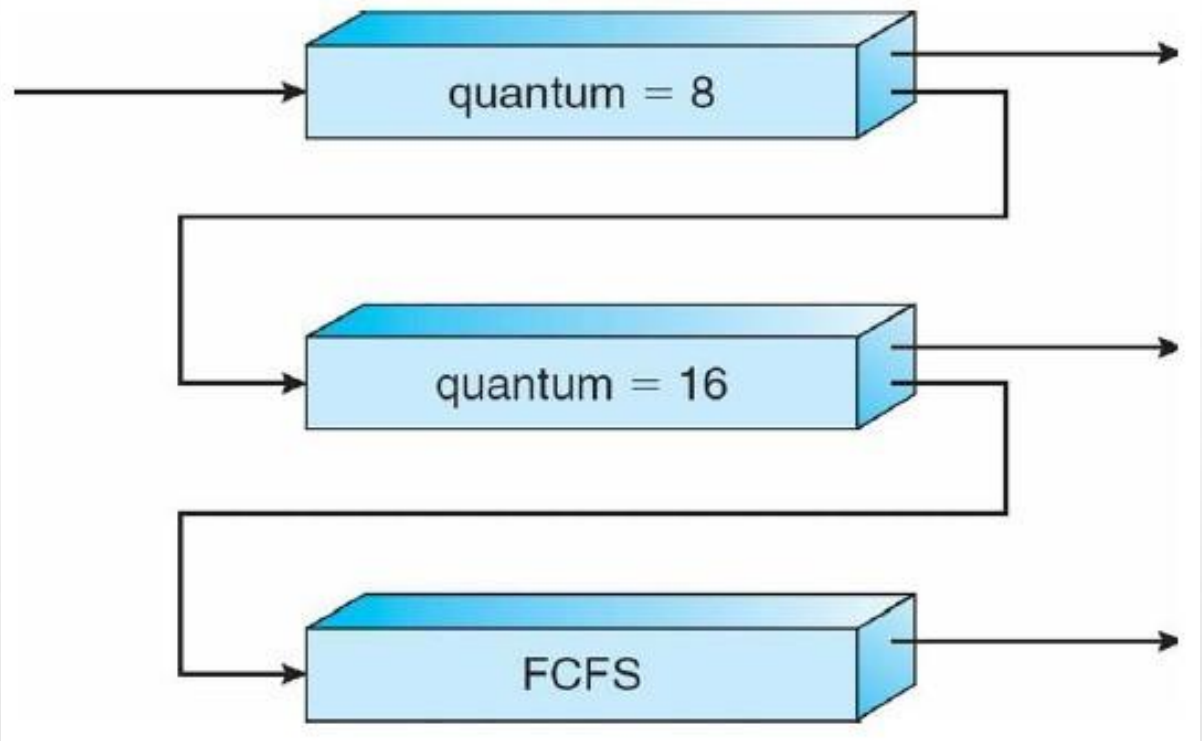
Esquema **Colas Multinivel** *(ejemplo 2)*

- El sistema consta de tres colas:
  - Q0: se planifica con *RR, q=8*
  - Q1: se planifica con *RR, q=16*
  - Q2: se planifica con *FCFS*
- Para la planificación se utilizan los siguientes criterios:
  - Los procesos ingresan en la *Q0*. Si no se utilizan los 8 quantums, el job es movido a la cola *Q1*
  - Para la cola *Q1*, el comportamiento es similar a *Q0*. Si un proceso no finaliza su ráfaga de 16 instantes, es movido a la cola *Q2*

Esquema **Colas Multinivel** *(ejemplo 2 cont.)*

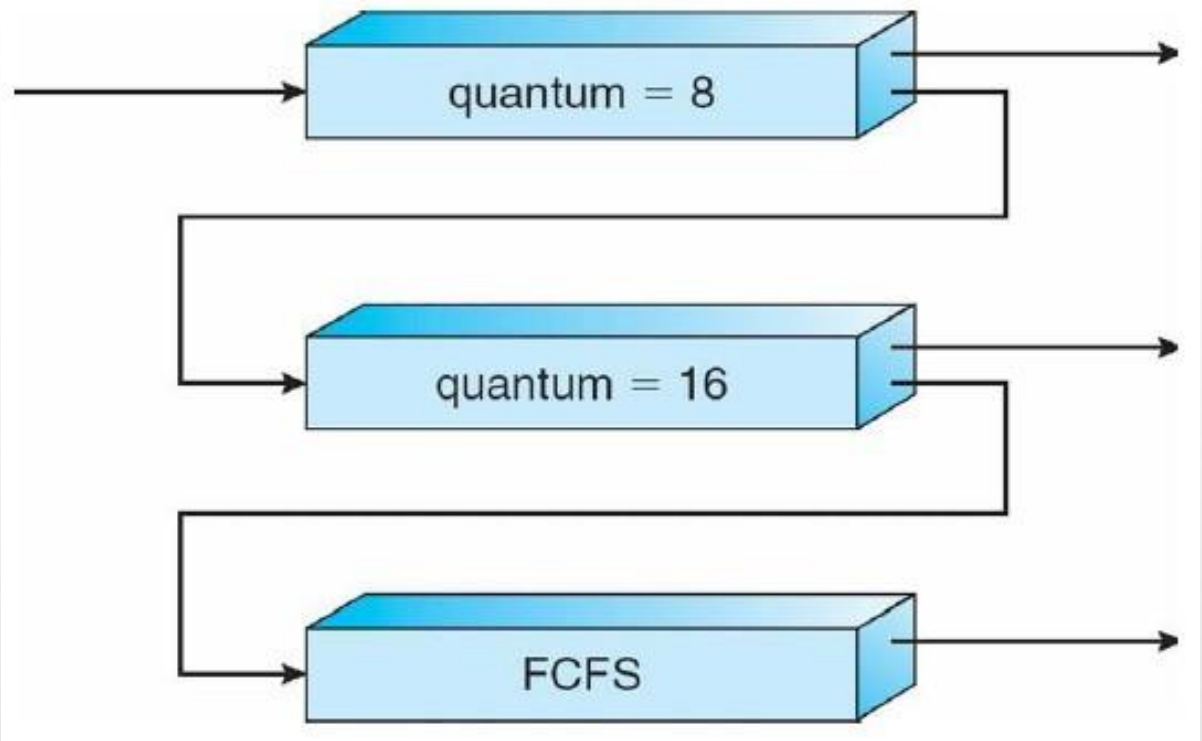


Esquema **Colas Multinivel** *(ejemplo 2 cont.)*



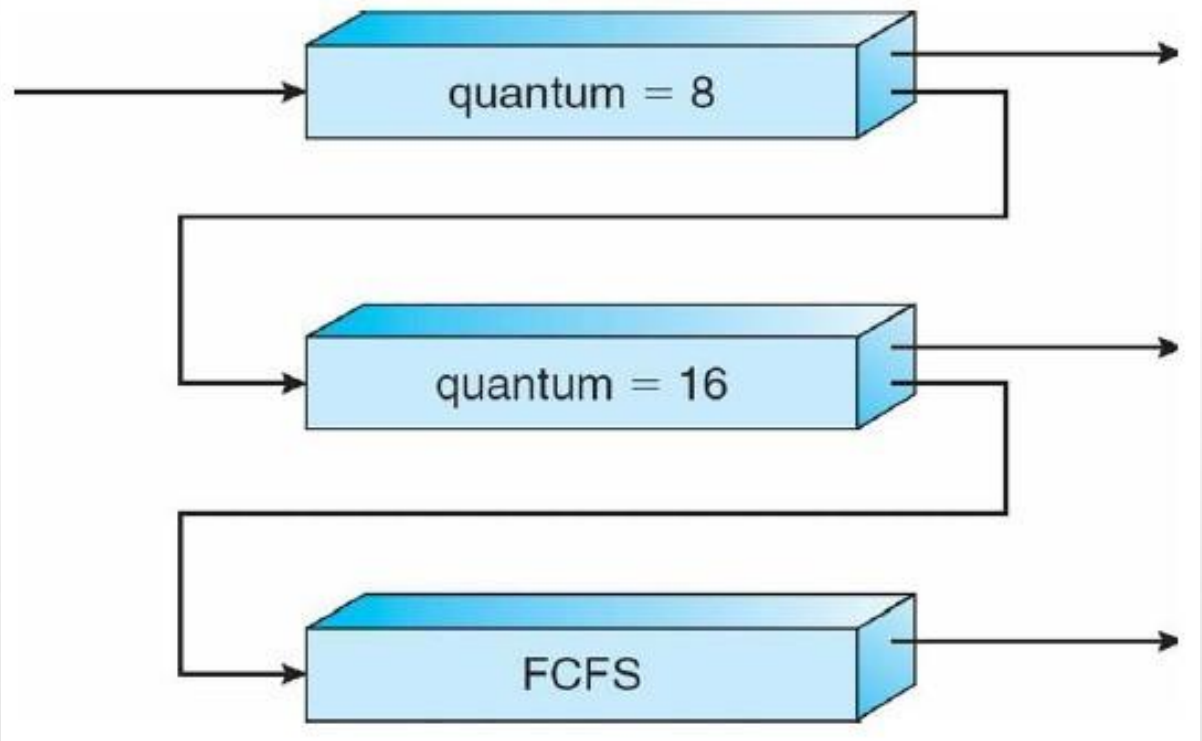
- ¿A qué procesos beneficia el algoritmo? → CPU Bound

Esquema **Colas Multinivel** *(ejemplo 2 cont.)*



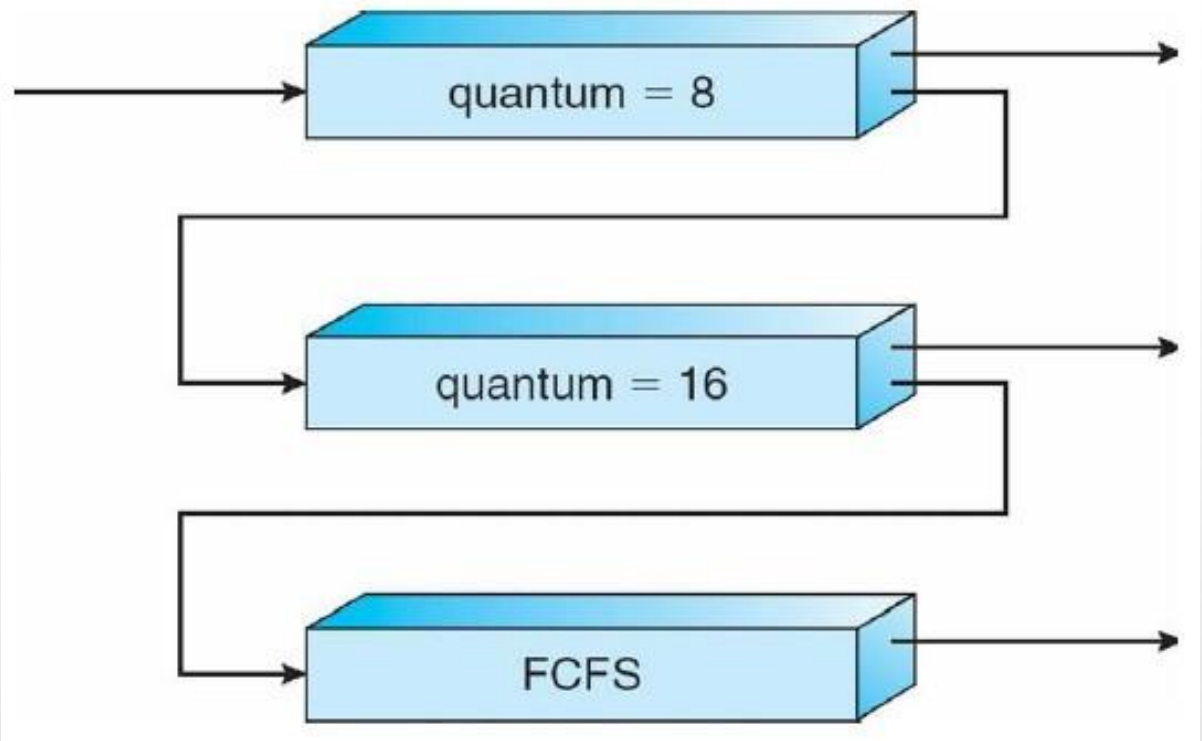
- ¿A qué procesos beneficia el algoritmo? → **CPU Bound**

Esquema **Colas Multinivel** *(ejemplo 2 cont.)*



- ¿Puede ocurrir inanición?

Esquema **Colas Multinivel** *(ejemplo 2 cont.)*



- ¿Puede ocurrir inanición? → Si, con los procesos ligados a *E/S* si siempre llegan procesos ligados a *CPU*

Planificación con múltiples procesadores

- La planificación de CPU es más compleja cuando hay múltiples CPUs
- Este enfoque fue implementado inicialmente en *Mainframes* y luego en *PC*
- La carga se divide entre distintas CPUs, logrando capacidades de procesamiento mayores
- Si un procesador falla, el resto toma el control

Planificaci´on con mu´ltiples procesadores: Criterios

- **Planificaci´on temporal** → que proceso y durante cuanto
- **Planificaci´on espacial** → en que procesador ejecutar:
  - **Huella**: estado que el proceso va dejando en la cache de un procesador
  - **Afinidad**: preferencia de un proceso para ejecutar en un procesador
- La asignaci´on de procesos a un procesador puede ser:
  - **Est´atica**: existe una afinidad de un proceso a una CPU
  - **Din´amica**: la carga se comparte → balanceo de carga
- La pol´ıtica puede ser:
  - **Tiempo compartido**: se puede cosiderar una *cola global* o una *cola local* a cada procesador
  - **Espacio compartido**:
    - **Grupos** (*threads*)
    - **Particiones**

Planificaci´on con mu´ltiples procesadores (cont.)

- *Clasificaciones*:
  - **Procesadores homog´eneos**: todas las CPUs son iguales. No existen ventajas f´ısicas sobre el resto
  - **Procesadores heterog´eneos**: cada procesador tiene su propia cola, su propio clock y su propio algoritmo de planificaci´on
- *Otra clasificaci´on*:
  - **Procesadores d´ebilmente acoplados**: cada CPU tiene su propia memoria principal y canales
  - **Procesadores fuertenemente acoplados**: comparten memoria y canales
  - **Procesadores especializados**: uno o m´as procesadores principales de uso general y uno o m´as procesadores de uso espec´ıfico

¿Preguntas?

Administración de Memoria Principal

Explicación de práctica

Introducción a los Sistemas Operativos
Conceptos de Sistemas Operativos

Facultad de Informática Universidad Nacional de La Plata

2025

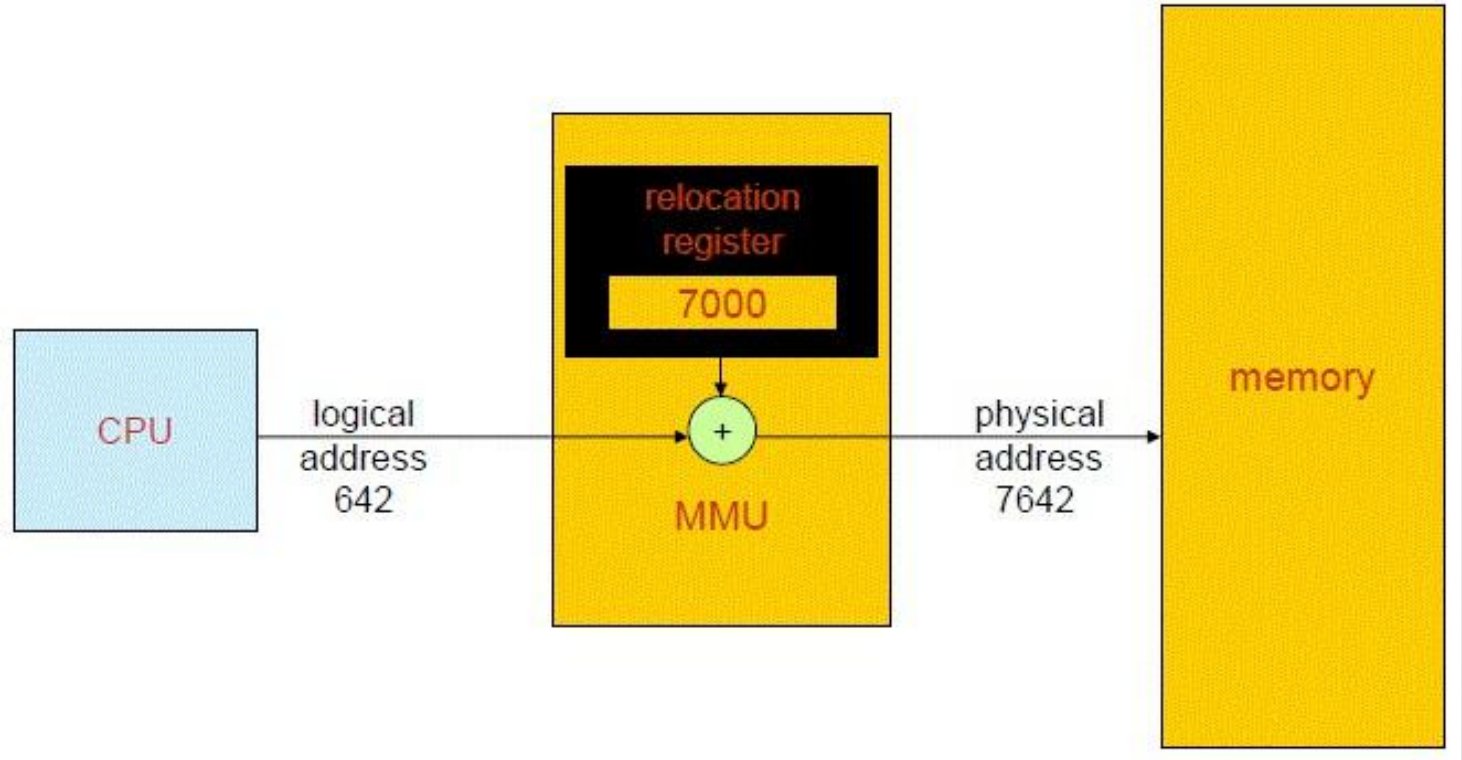
- La organización y administración de la *memoria RAM* es uno de los factores más importantes en el diseño de un SO
- Los programas y datos deben residir en ella para:
  - Poder ejecutar
  - Referenciarlos directamente

Memoria (cont.)

- La parte del SO que administra esta memoria se llama "*administrador de la memoria*":
  - Lleva un registro de las partes de la memoria que se están utilizando y de aquellas que no
  - Asigna espacio en memoria a los procesos cuando estos la necesitan
  - Libera espacio de memoria asignada a procesos que han terminado
- Se espera que el SO haga uso eficiente de esta memoria con el fin de alojar el mayor número de procesos → repercute en la *multiprogramación*

- **Dirección Lógica**:
  - Es una dirección que enmascara o abstrae una dirección física
  - Referencia a una localidad en memoria
  - Se la debe traducir a una dirección física
- **Dirección Física**:
  - Es la dirección real. Es con la que se accede efectivamente a memoria
  - Representa la dirección absoluta en memoria principal
- La CPU trabaja con direcciones lógicas. Para acceder a la memoria se deben transformar en direcciones físicas
- El mapeo entre direcciones virtuales y físicas se realiza mediante *hardware* → **MMU** (Memory Management Unit)

Traducción **MMU**



Asignación de memoria

- **Particiones Fijas**:
  - La memoria se divide en particiones o regiones de tamaño fijo → tamaños iguales o diferentes
  - Alojan un único proceso
- **Particiones Dinámicas**:
  - Las particiones varían en tamaño y número
  - Alojan un proceso cada una
  - Cada partición se genera en forma dinámica del tamaño justo que necesita el proceso
- **Selección de partición**:
  - Cada proceso se coloca en alguna partición de acuerdo a algún criterio:
    - **First Fit**
    - **Best Fit**
    - **Worst Fit**
    - **Next Fit**

¿Qué problemas se generan en cada caso?

Asignación de memoria

- **Particiones Fijas**:
  - La memoria se divide en particiones o regiones de tamaño fijo → tamaños iguales o diferentes
  - Alojan un único proceso
- **Particiones Dinámicas**:
  - Las particiones varían en tamaño y número
  - Alojan un proceso cada una
  - Cada partición se genera en forma dinámica del tamaño justo que necesita el proceso
- **Selección de partición**:
  - Cada proceso se coloca en alguna partición de acuerdo a algún criterio:
    - **First Fit**
    - **Best Fit**
    - **Worst Fit**
    - **Next Fit**

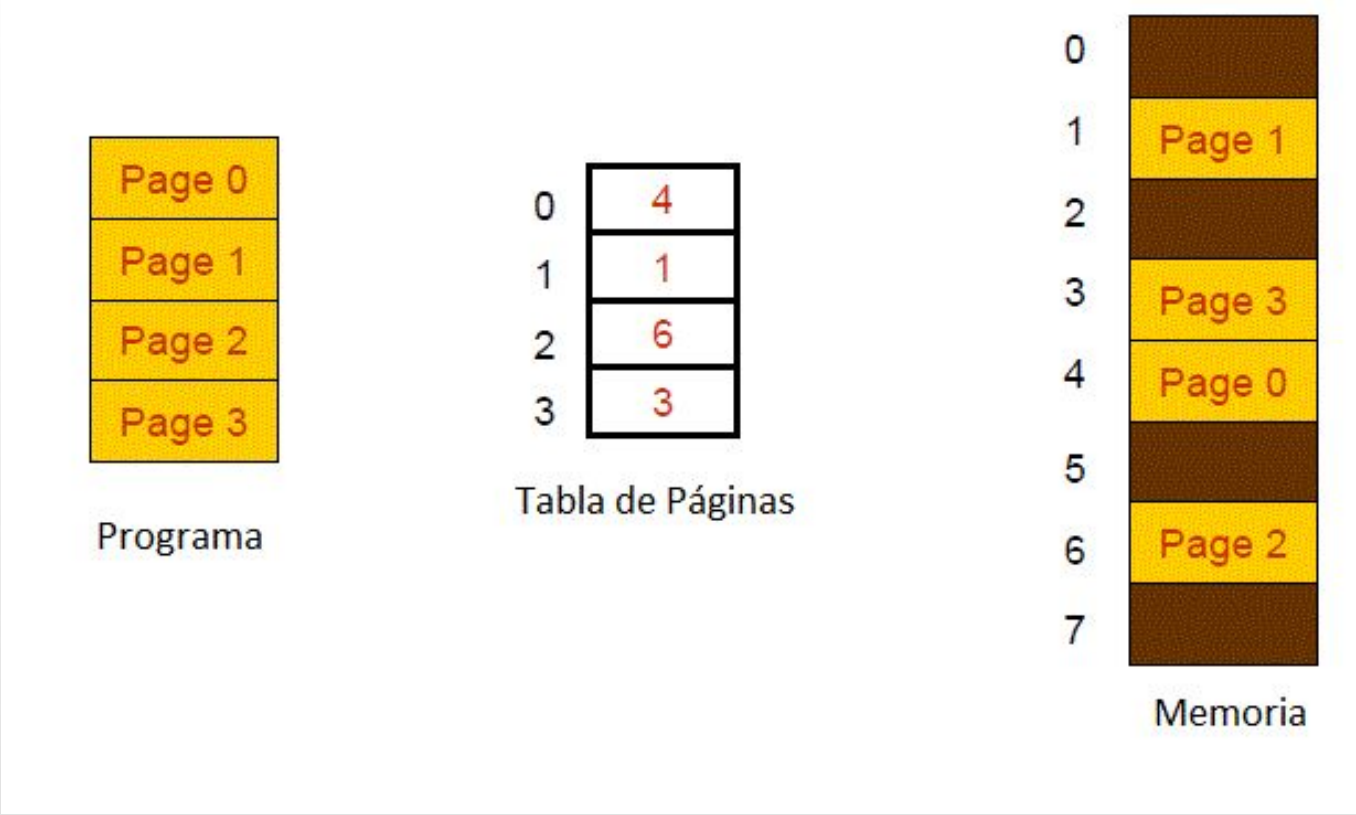
¿Qué problemas se generan en cada caso?

Fragmentación

- La fragmentación se produce cuando una localidad de memoria no puede ser utilizada por no encontrarse en forma contigua
- **Fragmentación Interna**:
  - Se produce en el esquema de particiones fijas
  - Es interna a la localidad asignada
  - Es la porción de la localidad que queda sin utilizar
- **Fragmentación Externa**:
  - Se produce en el esquema de particiones dinámicas
  - Son huecos que van quedando en la memoria a medida que los procesos finalizan
  - Al no encontrarse en forma contigua puede darse el caso de que tengamos memoria libre para alocar un proceso, pero que no la podamos utilizar
  - Solución → *compactación* → muy costosa

Paginación

- La memoria se divide en porciones de igual tamaño llamadas **marcos**
- El espacio de direcciones de los procesos se divide en porciones de igual tamaño denominadas **páginas**
- Tamaño *página* = tamaño *marco* = 512 bytes (generalmente)
- El SO mantiene una tabla de páginas para cada proceso, la cual contiene el *marco* donde se encuentra cada página
- La paginación bajo demanda es una técnica eficiente de manejar esta estrategia


Page 0
Page 1
Page 2
Page 3
Programa
0
1
2
3
4
1
6
3
Tabla de Páginas
0
1
2
3
4
5
6
7
Page 1
Page 3
Page 0
Page 2
Memoria

Paginación - **Direccionamiento**

- Un proceso en ejecución hace referencia a una dirección virtual $\rightarrow v = (p,d)$
- El SO busca la página *p* en la *tabla de páginas* del proceso y determina en qué *marco* se encuentra
- La dirección de almacenamiento real se forma por la concatenación de la resolución de *p* (dirección inicio del marco que aloca la página) y *d*, donde *p* es el número de página y *d* es el desplazamiento

Paginación - Ejemplo

- Memoria administrada por sistema de paginación
- Tamaño de página → 512 Bytes
- Cada dirección de memoria referencia a 1 Byte
- Los *marco* en memoria principal se encuentran desde la dirección física 0
- Tenemos un proceso con un tamaño de 2000 Bytes y con la siguiente tabla de páginas

Paginación - Ejemplo

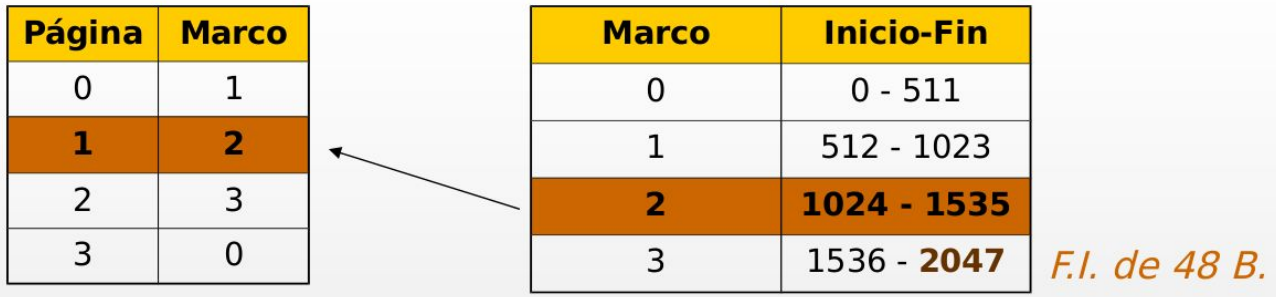
Página	Marco
0	1
1	**2**
2	3
3	0

Marco	Inicio-Fin
0	0 - 511
1	512 - 1023
2	**1024 - 1535**
3	1536 - **2047**

F.I. de 48 B.

- Si tenemos una dirección **virtual**, por ejemplo 580:
  - Para averiguar el número de página hacemos 580 div 512 = 1. Luego esta dirección corresponde a la página 1 que se encuentra en el *marco* 2
  - Para averiguar el desplazamiento hacemos 580 mod 512 = 68
  - La dirección física es 1024 + 68 = 1092

Paginación - Ejemplo



Página	Marco
0	1
1	**2**
2	3
3	0

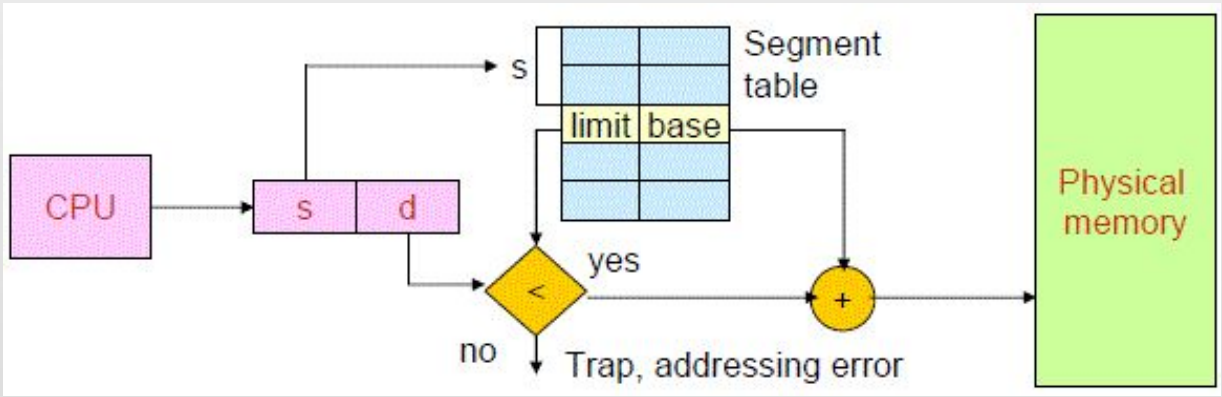
Marco	Inicio-Fin
0	0 - 511
1	512 - 1023
2	**1024 - 1535**
3	1536 - **2047**

F.I. de 48 B.

- Si tenemos una dirección **física**, por ejemplo 1092:
  - Para averiguar el número de *marco* hacemos 1092 div 512 = 2. En el *marco* número 2 tenemos la página número 1
  - Para averiguar el desplazamiento hacemos 1092 mod 512 = 68
  - La dirección virtual es 512 + 68 = 580

Segmentación

- La segmentación básicamente la podemos ver como una mejora de la paginación (*no hay F.I., sino externa*)
- Ahora la tabla de segmentos, además de tener la dirección de inicio del mismo, tiene la longitud o límite
- Las direcciones lógicas constan de dos partes → un número de segmento *s* y un desplazamiento *d* dentro del segmento
  (*0 ⟨ d ⟨ límite*)



Shell scripting
Explicación de práctica

Introducción a los Sistemas Operativos
Conceptos de Sistemas Operativos

Facultad de Informática Universidad Nacional de La Plata

2025

Agenda

1. Introducción
2. Conceptos básicos
   - Comandos
   - Redirecciones y pipes
   - Variables y sustitución de comandos
   - Reemplazo de comandos
3. Programación de scripts
   - Scripts
   - Estructuras de control
   - Comparaciones
   - Estructuras de control en detalle
   - Argumentos y valor de retorno
   - Funciones
   - Alcance y visibilidad

Agenda

1. Introducción
2. Conceptos básicos
   - Comandos
   - Redirecciones y pipes
   - Variables y sustitución de comandos
   - Reemplazo de comandos
3. Programación de scripts
   - Scripts
   - Estructuras de control
   - Comparaciones
   - Estructuras de control en detalle
   - Argumentos y valor de retorno
   - Funciones
   - Alcance y visibilidad

¿Qué es una shell?

- Intérprete de comandos
- Interactivo
- En sistemas operativos *nix es configurable
- Proveen estructuras de control que permiten programar *shell scripts*

¿Qué puedo hacer con *shell scripts*?

- Automatización de tareas
- Aplicaciones interactivas
- Aplicaciones con interfaz gráfica (con el comando `zenity`, por ejemplo)

Existen muchas *shells*. Sus diferencias consisten principalmente en sintaxis. A continuación se listan las más utilizadas:

- `sh:` *Shell* por defecto en Unix.
- `bash:` Cómoda, instalada por defecto en la mayoría de las distribuciones.
- `dash:` Eficiente, parcialmente compatible con `bash`.
- `csh:` Sintaxis incompatible con `bash`/`dash`.
- Otros...

Tip

En la materia utilizaremos `bash`.

Diferencias con otros lenguajes

¿Por qué *shell script* y no C, o Java, o Python?

- 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 *nix (distribución GNU/Linux, Mac OS X, etc.)
- Se puede probar en el intérprete interactivo (a diferencia de C y Java)

Elementos del lenguaje

- Instrucciones: comandos
  - Internos o *built-in* (`help` para verlos)
  - Externos (archivos separados `man comando`)
- Redirecciones y *pipes*
- Comentarios que empiezan con #
- Estructuras de control
  - `if`
  - `while`
  - `for` (2 tipos)
  - `case`
- Variables
  - *Strings*
  - Arreglos `()`
- Funciones

Agenda

1. Introducción
2. Conceptos básicos
   - Comandos
   - Redirecciones y pipes
   - Variables y sustitución de comandos
   - Reemplazo de comandos
3. Programación de scripts
   - Scripts
   - Estructuras de control
   - Comparaciones
   - Estructuras de control en detalle
   - Argumentos y valor de retorno
   - Funciones
   - Alcance y visibilidad

Repaso de algunos comandos útiles

- Imprimir el contenido de un archivo

```
cat archivo
```

- Imprimir texto

```
echo "Hola mundo"
```

- Leer una línea desde entrada estándar en la variable `var`

```
read var
```

- Quedarme con la primer columna de un texto separado por `:` desde entrada estándar

```
cut -d: -f1
```

- Contar la cantidad de líneas que se leen desde entrada estándar

```
wc -l
```

Repaso de algunos comandos útiles

- Buscar todos los archivos que contengan la cadena `pepe` en el directorio `/tmp`

```
grep pepe /tmp/*
```

- Buscar todos los archivos dentro del *home* del usuario, cuyo nombre termine en `.doc`

```
find $HOME -name "*.doc"
```

- Buscar todos los archivos dentro del directorio actual que sean enlaces simbólicos

```
find . -type l
```

- **Empaquetado:** Se unen varios archivos en uno solo (`tar`)

```
tar -cvf archivo.tar archivo1 archivo2 archivo3
tar -xvf archivo.tar
```

- **Compresión:** Se reduce el tamaño de un archivo (`gzip`/`bzip2`/etc.)

```
gzip archivo.tar # Genera archivo.tar.gz comprimido
gzip -d archivo.tar.gz # Descomprime archivo.tar
```

- El comando `tar` puede invocar a `gzip` por nosotros (argumento `z`):

```
tar -cvzf archivo.tar.gz arch1 arch2 arch3
tar -xvzf archivo.tar.gz
```

Redirecciones y pipes: stdin, stdout, stderr

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

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

Estos archivos se identifican con un número, el *file descriptor* (descriptor de archivo):

0	stdin
1	stdout
2	stderr

Sintaxis básica:

```
comando > archivo
comando >> archivo
```

- Redirección destructiva (>):
  - Si archivo no existe, se crea.
  - Si archivo existe, lo sobreescribe.
- Redirección no destructiva (>>):
  - Si archivo no existe, se crea.
  - Si archivo existe, se le agrega al final.

¿Qué hacen los siguientes scripts?

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

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

Sintaxis básica:

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

- 2> y 2>> Redirigen la salida de 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*.

Sintaxis básica:

```
comando | comando2 | comando3
```

- Conectan la salida de un comando con la entrada de otro.
- Indispensables para hacer programas potentes en *shell script*.

Ejemplos:

```
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
```

- bash soporta *strings* y *arrays*
- Los nombres son *case sensitive*
- Para crear una variable:

```
NOMBRE="pepe" # SIN espacios alrededor del =
```

- Para accederla se usa $:

```
echo $NOMBRE
```

- Para evitar ambigüedades se pueden usar llaves:

```
# Esto no accede a $NOMBRE
echo $NOMBREesto_no_es_parte_de_la_variable
# Esto sí
echo ${NOMBRE}esto_no_es_parte_de_la_variable
```

Los nombres de las variables pueden contener mayúsculas, minúsculas, números y el símbolo _ (*underscore*), pero no pueden empezar con un número.

```
NOMBRE="Fulano De Tal"
facultad=Informatica
carrera_1="Licenciatura en Sistemas"
carrera_2="Licenciatura en Informatica"
echo El alumno $NOMBRE de la Facultad de $facultad cursa
$carrera_1 y $carrera_2
# imprime:
#  El alumno Fulano De Tal de la Facultad  de
Informática cursa Licenciatura en Sistemas y
Licenciatura en Informática
```

Variables: Ejemplo 2

```
nombre=Carlos
echo "Hola $nombre" # Hola Carlos
echo Hola ${nombre} # Hola Carlos
nombre=5
echo "Hola $nombre" # Hola 5
```

- Creación:

```
arreglo_a=() # Se crea vacío
arreglo_b=(1 2 3 5 8 13 21) # Inicializado
```

- Asignación de un valor en una posición concreta:

```
arreglo_b[2]=spam
```

- Acceso a un valor del arreglo (en este caso las llaves no son opcionales):

```
echo ${arreglo_b[2]}
copia=${arreglo_b[2]}
```

- Acceso a todos los valores del arreglo:

```
echo ${arreglo[@]} # o bien ${arreglo[*]}
```

- Tamaño del arreglo:

```
${#arreglo[@]}
${#arreglo[*]}
```

- Borrado de un elemento (reduce el tamaño del arreglo pero no elimina la posición, solamente la deja vacía):

```
unset arreglo[2]
```

- Los índices en los arreglos comienzan en 0

Variables: Ejemplo de arreglos

```
#!/bin/bash

arreglo=(1 2 3 5 8 13 21)

arreglo[2]=spam

echo "El primer elemento es ${arreglo[0]}"

echo "El tercer elemento es ${arreglo[2]}"

echo "La longitud: ${#arreglo[*]}"

echo "Todos sus elementos: ${arreglo[*]}"
```

- 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...
- Tipos de comillas
  - 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)'
```

```
# Un ejemplo:
variable="un texto de varias palabras"
variable_2=UnaSolaPalabra

echo "Podemos leer $variable"
echo 'No podemos leer $variable'

variable_3="Asi concateno $variable_2 a otro string"

echo $variable_3
echo variable_3
```

- ¿Qué se imprime en cada caso?

- Permite utilizar la salida de un comando como si fuese una cadena de texto normal.
- Permite guardarlo en variables o utilizarlos directamente.
- Se la puede utilizar de dos formas, cada una con distintas reglas:

```
$(comando_valido)
`comando_valido`
```

Nota: La primer forma resulta más clara y posee reglas de anidamiento de comandos más sencillas. Ejemplo:

```
arch="$(ls)"
mis_archivos="$(ls /home/$(whoami))"
```

Agenda

1. Introducción
2. Conceptos básicos
   - Comandos
   - Redirecciones y pipes
   - Variables y sustitución de comandos
   - Reemplazo de comandos
3. Programación de scripts
   - Scripts
   - Estructuras de control
   - Comparaciones
   - Estructuras de control en detalle
   - Argumentos y valor de retorno
   - Funciones
   - Alcance y visibilidad

- Crear un archivo con cualquier editor de texto.
- Indicar el intérprete (opcional) en la primer línea con:

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

  Esta línea, denominada *shebang*, especifica el intérprete que se utilizará para ejecutar *script*. Si no se especifica, se utiliza el intérprete por defecto del usuario.
- ¿Qué problemas puede traer esto?
- Escribir una serie de comandos en el archivo, de la misma manera que lo haríamos en la terminal.
- ¿Guardar el script con terminación *.sh*?
- ¿Permisos de ejecución?

Ejemplos

```
#!/bin/bash
# Si la primera línea de mi script comienza
# con la cadena #! se interpretará como el
# path al intérprete a utilizar (podréa ser
# ruby, python, php, node, etc...)
# Ahora el script en sí:

echo "Hola mundo"
```

Repaso de conceptos:

- **Path absoluto:** Empieza desde el directorio raíz `/`
- **Path relativo:** Empieza desde donde estamos posicionados
- `.` y `..` son el directorio actual y el directorio padre
- **Variable de entorno PATH**

¿Cómo ejecutar un *script*?

- ¿Le damos permisos de ejecución?
- Lo ejecutamos
  - `./mi_script.sh`
  - `bash mi_script.sh`
  - O podemos también ejecutarlo en modo *debug* (depuración)

    `bash -x mi script.sh`

Selección de alternativas:

Decisión:

```
if [ condition ]
then
  bloque
elif [ condition ]
then
  bloque
else
  bloque
fi
```

Selección:

```
case $variable in
"valor 1")
  bloque
  ;;
"valor 2")
  bloque
  ;;
*)
  bloque
  ;;
esac
```

Menú de opciones:

```
select variable in opcion1 opcion2 opcion3
do
  # en $variable está el valor elegido
  bloque
done
```

Menú de opciones:

Ejemplo:

```
select accion in Nuevo Salir
do
  case $accion in
    "Nuevo")
      echo "Seleccionado: Nuevo"
      ;;
    "Salir")
      exit 0
      ;;
  esac
done
```

- C-*style*:

```
for ((i=0; i < 10; i++))
do
  bloque
done
```

- Con lista de valores (foreach):

```
for i in valor1 valor2 valor3 valorN
do
  bloque
done
```

while

```
while [ condicion ] # Mientras se cumpla la condición
do
  bloque
done
```

until

```
until [ condition ] # Mientras NO se cumpla la condición
do
  bloque
done
```

Las condiciones lógicas normalmente se evalúan mediante:

```
[ condicion ]
test condicion
```

Operadores para `condicion`:

Operador	Con strings	Con números
Igualdad	`"$nombre" = "Maria"`	`$edad -eq 20`
Desigualdad	`"$nombre" != "Maria"`	`$edad -ne 20`
Mayor	`A > Z`	`5 -gt 20`
Mayor o igual	`A >= Z`	`5 -ge 20`
Menor	`A < Z`	`5 -lt 20`
Menor o igual	`A <= Z`	`5 -le 20`

Ejemplos

```
if [ "$USER" == root ]
then
  echo "superuser"
else
  echo "Ud es $USER"
fi
```

```
n=0
while [ $n -ne 5 ]; do
  echo $n
  let n++
done
```

```
for archivo in $(ls)
do
  echo "- $archivo"
done
```

```
for ((i=0; i < 5; i++))
do
  echo $i
done
```

Adicionalmente:

- `break [n]` corta la ejecución de n niveles de *loops*.
- `continue [n]` salta a la siguiente iteración del enésimo *loop* que contiene esta instrucción.

```
#!/bin/bash
# Imprime los números del 1 al 5
# (no es un código para nada elegante)
# true es un comando que siempre retorna 0
i=0
while true
do
  let i++ # Incrementa i en 1
  if [ $i -eq 6 ]; then
    break # Corta el loop (while)
  fi
  echo $i
done
```

¿Qué hace el siguiente script?

```
#!/bin/bash
i=0
while true; do
  let i++
  if [ $i -eq 6 ]; then
    break # Corta el while
  elif [ $i -eq 3 ]; then
    continue # Salta una iteración
  fi
  echo $i
done
```

Condiciones compuestas

```
# AND
if [ $a = $b ] && [ $a = $c ]; then

# OR
if [ $a = $b ] || [ $a = $c ]; then
```

Argumentos y valor de retorno

- Los *scripts* pueden recibir argumentos en su invocación.
- Para accederlos, se utilizan variables especiales:
  - `$0` contiene la invocación al script.
  - `$1, $2, $3, ...` contienen cada uno de los argumentos.
  - `$#` contiene la cantidad de argumentos recibidos.
  - `$*` contiene la lista de todos los argumentos.
  - `$?` contiene en todo momento el valor de retorno del último comando ejecutado.

```
if [ $# -ne 2 ]; then
  exit 1 # Error
else
  echo "Nombre: $1, Apellido: $2"
fi
exit 0 # Funcionó correctamente
```

Para terminar un script usualmente se utiliza el comando `exit`:

- Causa la terminación de un script.
- Puede devolver cualquier valor entre 0 y 255:
  - El valor 0 indica que el script se ejecutó de forma exitosa
  - Un valor distinto indica un código de error
  - Se puede consultar el *exit status* imprimiendo la variable `$?`

Las funciones permiten modularizar el comportamiento de los scripts.

- Se pueden declarar de 2 formas:
  - `function nombre { bloque }`
  - `nombre() { bloque }`
- Con la sentencia `return` se retorna un valor entre 0 y 255
- El valor de retorno se puede evaluar mediante la variable `$?`
- Reciben argumentos en las variables `$1`, `$2`, etc.

```
# Recibe 2 argumentos y devuelve:
# 1 si el primero es el mayor
# 0 en caso contrario

mayor() {
  echo "Se van a comparar los valores: $*"
  if [ $1 -gt $2 ]; then
    echo "$1 es el mayor"
    return 1
  fi
  echo "$2 es el mayor"
  return 0
}

mayor 5 6     # Invocación
echo $?       # Imprime el exit status de la función
```

Variables: alcance y visibilidad

- Las variables no inicializadas son reemplazadas por un valor nulo o 0, según el contexto de evaluación.
- Por defecto las variables son globales.
- Una variable local a una función se define con `local`

```
test() {
  local variable
}
```

- Las variables de entorno son heredadas por los procesos hijos.
- Para exponer una variable global a los procesos hijos se usa el comando `export`:

```
export VARIABLE_GLOBAL="Mi var global"
comando
# comando verá entre sus variables de
# entorno a VARIABLE_GLOBAL
```

¿Preguntas?