# Arquitectura del Computador
# y
# Sistemas Operativos

Decimoquinta Clase


# Proceso de Arranque (o Booting)

## Alternativas

Hay dos formas de llegar desde el encendido del procesador (instante en que toda la RAM está borrada) hasta un ordenador con su Sistema Operativo cargado y listo para que el usuario lo opere.

Al proceso que se encarga de lograrlo se denomina "*Proceso de Arranque*". Este proceso hace, entre otras cosas, lo siguiente:

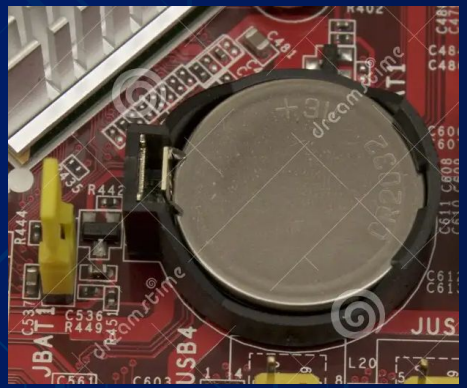
- Determina la cantidad de memoria y su estado
- Determina los dispositivos conectados y los inicializa
- Enumera los discos disponibles y de cuál de ellos cargará el SO
- Carga y le transfiere el control al "*cargador*" del SO. Este cargador es un pequeño programa escrito por el fabricante del SO que contiene las rutinas necesarias para cargar y arrancar el SO

# Proceso de Arranque (o Booting)

Al recibir energía, el procesador inicia la ejecución en una dirección específica. El fabricante del motherboard debe colocar una memoria no volátil (ROM/EPROM/FLASH) mapeada en esa posición y colocar ahí un programa.

Es importante entender, que cuando ese código se empieza a ejecutar, no se sabe qué dispositivos están conectados, ni siquiera se sabe si hay RAM ni cuánta, por lo que no se pueden usar variables. Ningún dispositivo externo (teclado, mouse, monitor, etc) está disponible porque no están inicializados.

La única memoria no volátil que tiene datos es una pequeña memoria RAM con soporte de batería que está en el mismo chip que el reloj. La batería suele ser una pila CR2032 como la de la figura.



El objetivo es mantener los valores configurados en el programa BIOS (Basic Input Output System).


# Proceso de Arranque (o Booting)

Esta rutina comienza con la primer tarea que se conoce como POST (Power On Self Test). La misma se encarga de:

1. Determinar la cantidad de RAM. Una vez conocida se la puede empezar a usar (El programa puede empezar a usar variables).
2. Como vimos, los dispositivos, requieren direcciones para intercambiar sus datos. También requieren líneas de interrupción. El proceso de arranque lee pequeñas memorias en cada dispositivo que le indican qué recursos necesitan y se los aloca. Este proceso se conoce como configuración "*Jumperless*" o "*Plug and Play*".
3. Una vez alocados los recursos solicitados por los dispositivos, se los inicializa. Como el programa es limitado algunos de ellos, como el monitor, se arrancan con un modo de video básico.
4. Se determina si presionaron alguna tecla. Algunas de ellas tienen funciones especiales. Por ejemplo si se presiona F2 se ingresa a otro programa ubicado en memoria no volátil que permite configurar opciones, entre ellas de qué disco se debe arrancar el SO.

En este punto termina el POST, y el camino ahora depende de si el disco de arranque es MBR o GPT.


# Proceso de Arranque (o Booting)

## Disco de arranque MBR

El proceso de carga del Sistema Operativo se conoce como Boot Sequence. En el caso que el disco de arranque tenga una partición MBR, se siguen los siguientes pasos:

1) Se carga el primer sector del disco de arranque a la dirección 0x7c00, que en todas las PCs tiene RAM.
2) El proceso anterior inicializó y enumeró todos los discos, se carga el registro DL con el número de disco del que se está arrancado.
3) Se transfiere el control con un JMP a la primer dirección del sector cargado.

El primer sector tiene un programa provisto por el fabricante del SO. Tiene como máximo 440 bytes así que es muy básico. Lo primero que hace es ver de cuál de las 4 particiones tiene que cargar el SO. Luego carga de disco algunos sectores adicionales sacados de direcciones CHS o LBA predefinidas que tienen código que permite interpretar el formato del disco.

Con el soporte de estas funciones se puede cargar el resto del SO. Éste, a su vez, reconfigura los dispositivos con los valores previamente definidos por el usuario, como la resolución del monitor, distribución de teclado, etc.


# Proceso de Arranque (o Booting)

## Disco de arranque GPT

Este modo de arranque es mucho más nuevo. La memoria no volátil del motherboard ya tiene las rutinas que permiten interpretar las particiones de tipo FAT12, FAT16 y FAT32.

El en la BIOS carga el módulo básico del SO de una partición formateada con sistema de archivos FATxx y le transfiere el control. Este programa, hecho por el propio fabricante del SO es el encargado de terminar de cargarlo y reconfigurar los dispositivos siguiendo la selección del usuario.


# Acceso al FileSystem desde el SO

El principal problema para integrar los filesystems con los Sistemas Operativos es la seguridad. Las otras operaciones (lectura, escritura, creación, truncado, etc) son muy parecidos entre todos los FSs y por ende fáciles de compatibilizar.

Distintos SO implementan la seguridad en forma distinta, y todos quieren integrar su forma con los archivos y/o directorios del filesystem.

Linux originalmente implementaba el estándar POSIX que permitía solamente permisos para user/group/others.

Windows, desde el comienzo usó ACLs (Access Control Lists) que era mucho más flexible.

Para resolver este problema Linux incorpora en el formato EXT2 los "*Extended Attributes*", que permiten asociar metadata con los archivos y directorios. Estos bloques de metadata son mantenidos por el filesystem, pero desconoce completamente qué significan. Queda en manos del SO


# Acceso al FileSystem desde el SO (1/2)

- **Funciones de Alto Nivel**

  Los sistemas operativos contienen drivers que permiten interpretar los diferentes formatos de disco. Estos drivers ofrecen funciones de alto nivel para manipular archivos y directorios.

- **Funciones de Bajo Nivel**

  Para que aplicaciones accedan a formatos para los cuales no hay drivers instalados, los SOs ofrecen funciones de bajo nivel que permiten acceder a sectores "*crudos*". Con esas funciones una aplicación puede interpretar "*por sí misma*" cualquier formato.


# Acceso al FileSystem desde el SO (2/2)

## APIs por nivel y Sistema Operativo

| | Drivers | Aplicaciones |
|---|---|---|
| Linux | POSIX | FUSE |
| Windows | WINAPI | WinFsp |

POSIX: Portable Operating System Interface. La X viene de Unix.

FUSE: Filesystem in UserSpace

WINAPI: Windows Application Programmer Interface

WinFsp: Windows File System Proxy


# Herramientas útiles para debugging (1/2)

Linux tiene un comando que nos permite conocer todos los llamados que una tarea hace al SO. Su nombre es *strace*.

Ejemplo: `strace cat /a.txt (el archivo a.txt tiene 2 bytes)`

```
openat(AT_FDCWD, "/a.txt", O_RDONLY)                = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=2, ...}) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL)           = 0
read(3, "1\n", 131072)                              = 2
write(1, "1\n", 21)                                 = 2
read(3, "", 131072)                                 = 0
close(3)                                            = 0
```

La herramienta nos permite ver que abrió el archivo, leyó el contenido y los copió al descriptor 1 (stdout).


# Herramientas útiles para debugging (2/2)

El file system */proc*:

Linux expone lo que parece un filesystem en el directorio /proc. Esta información no está físicamente en ningún disco, sino que expone de esa forma el estado del kernel.

Por ejemplo en el directorio raíz hay un directorio con el número de PID de cada proceso que corre. Dentro del mismo está toda la información que el kernel tiene del mismo:

- El subdirectorio *fd* tiene los datos de cada descriptor que el proceso tiene abierto.
- El symlink *cwd* apunta al directorio actual del proceso.
- El subdirectorio *map_files* contiene un archivo por cada zona de la memoria virtual del proceso compartida con otros módulos
- El subdirectorio *net* contiene toda la información de las comunicaciones del proceso.
- El symlink *exe* apunta al programa en disco.
- El archivo *cmdline* tiene los parámetros con que se arrancó la tarea

Y mucho más...

# Fin

¿Preguntas?