

# [75.08] Sistemas Operativos

María Inés Parnisari

19 de diciembre de 2013

## Índice

<b>I Parte Teórica</b>	<b>2</b>
1. Introducción	2
2. Procesos	5
3. Threads	13
4. Sistemas operativos móviles	17
5. Memoria	18
6. Linkedición	26
7. Bibliotecas	30
8. Virtualización	33
9. Archivos y Sistemas de archivo	36
<b>II Parte Práctica</b>	<b>53</b>
10. BASH	54
11. GREP	57
12. sed	62
13. PERL	63

## Parte I

# Parte Teórica

## 1 Introducción

### 1.1 El Zoo

Tipo	Características
Mainframes	Gran capacidad de I/O, seguridad, disponibilidad. <ul style="list-style-type: none"><li>■ Procesamiento por lotes,</li><li>■ de transacciones,</li><li>■ tiempo compartido.</li></ul>
Servidores	Ofrecen servicios a través de una red
Supercomputadoras	Velocidad de procesamiento, muchos procesadores. Uso: <i>high performance computing</i> (simuladores, cosmología...)
PC	Facilidad de uso, flexibilidad. Soportan multiprogramación
Tablets, PDA	
Consolas	
Embebidas	Diseñadas para cumplir una sola función, no aceptan software nuevo
Cluster	Computadoras conectadas a una LAN de alta velocidad (usos: alta disponibilidad, balance de carga, cálculo)
Tiempo Real	Las aplicaciones tienen <i>deadlines</i> , su uso debe ser previsible. Hay dos clases: <i>soft</i> y <i>hard</i> . Las de tipo <i>hard</i> , si no cumplen un <i>deadline</i> puede provocar un error fatal.

Cuadro 1: Tipos de computadoras

**Cloud Computing** tipo de computación en el cual recursos virtuales que son dinámicamente escalables se proveen como servicios sobre Internet.

- **IAAS** (*Infrastructure as a Service*): Amazon Web Services
- **PAAS** (*Platform as a Service*): Google Code, Azure
- **SAAS** (*Software as a Service*): PayPal, Google Apps, Steam

**Ley de Amdahl** El incremento en velocidad de un programa utilizando múltiples procesadores en paralelo está limitado por el tiempo que necesita la fracción secuencial de dicho programa. *For example, if a program needs 20 hours using a single processor core, and a particular portion of the program which takes one hour to execute cannot be parallelized, while the remaining 19 hours (95 %) of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour. Hence the speedup is limited up to 20x.*

**Ley de Moore** La cantidad de transistores en un circuito integrado de duplica cada dos años.

### 1.2 Definiciones

**Sistema operativo**

- Intermediario entre usuario y hardware.
- Entorno para ejecutar programas.
- Proporciona a los programadores un conjunto abstracto de recursos simples.
- Programa que administra el los recursos de una computadora.

Funciones del sistema operativo:

1. Identificar y localizar **archivos** mediante jerarquía de **directorios**.
2. Permitir a usuarios o grupos de usuarios establecer **permisos** de acceso a archivos.
3. Asignar **espacio** en discos, administrando la ocupación y liberación de archivos.
4. Coordinar la relación entre uno o varios procesos en ejecución, y su deseo de acceder al mismo dispositivo de almacenamiento simultáneamente (**conurrencia**).
5. Coordinar la comunicación entre la CPU (que maneja tiempos electrónicos rápidos) y los dispositivos de almacenamiento (que maneja tiempos mecánicos lentos), mediante el uso de **buffers**.

Sistema operativo como máquina extendida	Sistema operativo como administrador de recursos
	Mediador/coordinador: resuelve conflictos en las demandas de recursos
El sistema operativo le oculta el <i>hardware</i> al usuario y le presenta una perspectiva mucho más entendible	Proteger usuarios entre ellos (y de ellos mismos)
Un procesador parece como varios	Proveer una asignación, en forma ordenada y controlada, de los procesadores, memorias, y dispositivos de E/S entre todos los programas que compitan por ellos
Una memoria parece como varias	Multiplexaje de recursos en tiempo y en espacio

**Modos de la cpu:** restricciones a las instrucciones que pueden ejecutarse en un momento. Un sistema operativo puede tener partes corriendo en cada uno de los modos. Un bit en la PSW (*Program Status Word*) controla el modo.

- **Modo Kernel:** modo con menos restricciones.
- **Modo Usuario:** los programas de usuario solo pueden correr en este modo. No se permiten instrucciones que impliquen operaciones de E/S y protección de memoria.

## 1.3 Interrupciones

**Interrupciones** se utilizan para pasar entre modo usuario y modo kernel.

- Sincrónicas: un programa quiere hacer algo que no tiene permitido.
- Asincrónicas: generadas por un evento externo.
  - Interrupción de reloj
  - Interrupción de I/O
  - Interrupción externa (intentos de dividir por cero)

Dentro del ciclo de instrucción (*instruction cycle*) se encuentra el ciclo de interrupción (*interrupt cycle*), en el cual el procesador chequea si ocurrió alguna interrupción. Si no hay interrupciones pendientes, el procesador carga la siguiente instrucción del proceso actual. Si una interrupción está pendiente, el procesador:

1. Guarda el contexto del programa que está siendo ejecutado (registros, código de condición, dirección de retorno, etc.).
2. Setea el contador de programa (*program counter*) a la dirección donde comienza un programa de manejo de interrupciones (*interrupt handler*).

El procesador ahora carga la primer instrucción del programa de manejo de interrupciones, el cual atenderá la interrupción. Una interrupción podría estar seguida de un cambio del proceso en ejecución, aunque también podría suceder que, luego de la interrupción, vuelva a reanudarse el mismo proceso que venía ejecutándose.

## 1.4 Estructura de un sistema operativo

Sistemas monolíticos	Sistemas en capas	Microkernel	Cliente-servidor
Todo el sistema operativo se ejecuta como un solo programa en modo kernel.	<i>Some systems have more layers and are more strictly structured.</i>	<i>The idea is to have the kernel, i.e. the portion running in supervisor mode, as small as possible and to have most of the operating system functionality provided by separate processes. The microkernel provides just enough to implement processes.</i>	<i>Uses the microkernel approach, in which the microkernel just handles communication between clients and servers, and the main OS functions are provided by a number of separate processes.</i>
<i>The system switches from user mode to kernel mode during the poof and then back when the OS does a return (an RTI or return from interrupt).</i>	<i>The actual layers of an early layered system were</i> <ul style="list-style-type: none"><li>■ <i>The user process</i></li><li>■ <i>User programs</i></li><li>■ <i>I/O mgt</i></li><li>■ <i>User console—process communication</i></li><li>■ <i>Memory and drum management</i></li></ul>	<i>This means that when a (real) user process makes a system call there are more processes switches</i>	
Es poco manejable y difícil de comprender.			

## 1.5 Unix

Los recursos del sistema son administrados por el kernel. El kernel implementa los servicios esenciales del sistema operativo:

1. Administración de memoria
2. Administración de procesos
3. Concurrencia

Todo en Unix es un archivo (directorios, terminales, dispositivos...)

Unix es *full duplex*: todos los caracteres que se tipean se envían al kernel y luego a la terminal (excepto que se indique lo contrario).

**Shell** interfaz entre el usuario y el sistema operativo. Programa con el que interactúan los usuarios. Está basado en texto.

Funciones:

- Interpretar comandos
- Determinar formas de ejecución

Foreground con proceso hijo	Foreground sin proceso hijo	Background con proceso hijo
Necesita permiso de ejecución	No necesita permiso de ejecución	Necesita permiso de ejecución
No devuelve el control hasta que no finaliza	No devuelve el control hasta que no finaliza	Devuelve el control en el momento
\$ ./script.sh	\$ . ./script.sh	\$ ./script.sh &

Cuadro 2: Formas de ejecución de un script en Unix

- Expandir caracteres comodines
- Expandir variables de ambiente

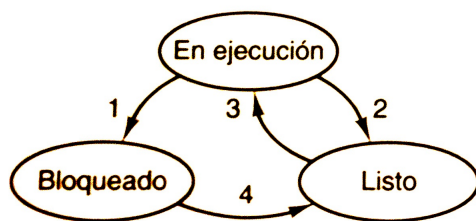
## 2 Procesos

**Modelo de procesos** todo el software ejecutable está organizado en un número de procesos secuenciales.

**Proceso** abstracción de la CPU. Instancia de un programa en ejecución, incluyendo el *program counter*, registros y variables. Cada proceso tiene:

- su propia CPU virtual,
- un espacio de direcciones,
- una **tabla de archivos (user file table)**: una tabla del proceso que almacena información acerca de los archivos que tiene abiertos el proceso (solo almacena los *file descriptors* - los datos del archivo se guardan en una estructura en el kernel),
- uno o más *threads* bajo su control.

## 2.1 Estados de un proceso



1. El proceso se bloquea para recibir entrada
2. El planificador selecciona otro proceso
3. El planificador selecciona este proceso
4. La entrada ya está disponible

Figura 1: Estados de un proceso

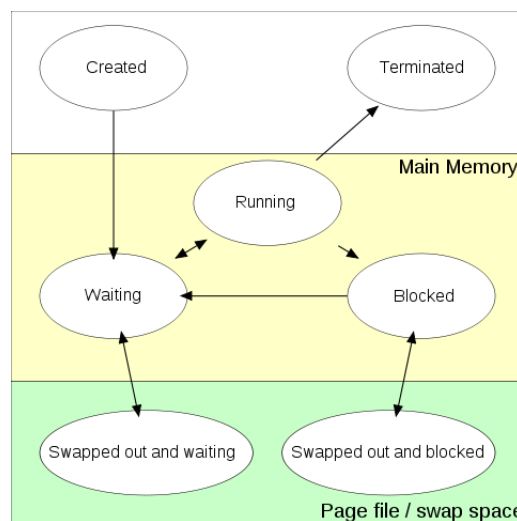


Figura 2: Estados de un proceso, ampliado

1. First, the process is "created" - it is loaded from a secondary storage device (hard disk or CD-ROM...) into main memory. After that the process scheduler assigns it the state "waiting".
2. While the process is "waiting" it waits for the scheduler to do a so-called context switch and load the process into the processor. The process state then becomes "running", and the processor executes the process instructions.
3. If a process needs to wait for a resource (wait for user input or file to open ...), it is assigned the "blocked" state. The process state is changed back to "waiting" when the process no longer needs to wait.
4. Once the process finishes execution, or is terminated by the operating system, it is no longer needed. The process is removed instantly or is moved to the "terminated" state. When removed, it just waits to be removed from main memory

### States

- **Created**: the process awaits admission to the "ready" state. This admission will be approved or delayed by a long-term scheduler. Typically in most desktop computer systems, this admission will be approved automatically, however for real-time operating systems this admission may be delayed.
- **Waiting / Ready**: process has been loaded into main memory and is awaiting execution on a CPU (to be context switched onto the CPU by the dispatcher, or short-term scheduler). There may be many "ready" processes at any one point of the system's execution. A ready queue or run queue is used in computer scheduling.

- **Running**: process moves into the running state when it is chosen for execution. The process's instructions are executed by one of the CPUs (or cores) of the system. There is at most one running process per CPU or core.
- **Blocked**: process that is blocked on some event (such as I/O operation completion or a signal).
- **Terminated**: process may be terminated, either from the "running" state by completing its execution or by explicitly being killed. In either of these cases, the process moves to the "terminated" state. If a process is not removed from memory after entering this state, it may become a Zombie process.

## 2.2 Multiprogramación

**Monoprogramación** hay un solo proceso corriendo a la vez en un sistema operativo.

**Multiprogramación** hay un solo proceso corriendo a la vez en un sistema operativo, pero se cambia rápido de un proceso a otro. Se multiplexa la CPU en el tiempo mediante interrupciones de reloj.

El tiempo es dividido en pequeños segmentos denominados *time slices* (que son variables en casi todos los sistemas). Cuando el *time slice* termina, el *dispatcher* le permite al *scheduler* actualizar el estado de cada proceso, y seleccionar el siguiente proceso a ejecutar.

- Cuando se produce una interrupción de reloj, el **scheduler** decide a qué proceso en estado *ready* darle el control. Para hacerlo, se mantiene una cola de PCBs correspondientes a procesos en estado *ready*.

Objetivos del *scheduler*:

1. *Fairness*: dar una participación adecuada del reparto de tiempo de CPU
2. *Load Balancing*: equilibrar el uso de recursos
3. Aplicar las políticas generales del sistema (prioridades, seguridad)

Respecto a la forma de manejar una interrupción de reloj, un algoritmo de planificación puede ser:

- **Apropiativo**: selecciona un proceso y deja que se ejecute por un máximo de tiempo. Si luego de ese intervalo de tiempo se sigue ejecutando, se suspende y se selecciona otro proceso.
- **No apropiativo**: selecciona un proceso para ejecutarlo y deja que se ejecute hasta que se bloquee o hasta que libera la CPU en forma voluntaria.

Las decisiones de *scheduling* se pueden tomar cuando:

1. El proceso se bloquea por esperar una operación de E/S. (*running* a *blocked*). (no-apropiativa)
2. El proceso se crea. (apropiativa)
3. Ocurre una interrupción de E/S (*blocked* a *ready*). (apropiativa)
4. El proceso termina. (no apropiativa)

El *scheduler* tiene en cuenta los siguientes items a la hora de tomar decisiones:

- Cantidad requerida de recursos.
  - Cantidad actualmente disponible de recursos.
  - Prioridad del trabajo o proceso.
  - La cantidad de tiempo de espera.
- El **dispatcher** le da a un proceso el control de la CPU. Implica las siguientes operaciones:
    - Conmutar el contexto (*context switching*).

**Context switch** *process of storing and restoring the state (context) of a process so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system. What constitutes the context is determined by the processor and the operating system. Context switches are usually computationally intensive, and much of the design of operating systems is to optimize the use of context switches. Switching from one process to another requires a certain amount of time for doing the administration - saving and loading registers and memory maps, updating various tables and lists, etc.*

*A context switch can mean a register context switch, a task context switch, a stack frame switch, a thread context switch, or a process context switch.*

El tiempo que dura el *context switch* es desperdiciado (*overhead*) ya que el sistema no puede aprovecharlo para hacer otra cosa.

El proceso de *context switching* consta de los siguientes pasos:

1. Se almacena el contexto del procesador, incluyendo el *program counter* y otros registros.
2. Se actualiza el bloque de control del proceso (PCB) que está actualmente en el estado *running*. Esto implica cambiar el estado del proceso a alguno de los otros estados (*ready*, *blocked*, etc.).
3. Se mueve el PCB de este proceso a la cola apropiada (*ready*, *blocked*, etc.).
4. Se selecciona otro proceso para ejecución (esto implica llamar al *scheduler*).
5. Se actualiza el PCB del proceso seleccionado. Esto implica cambiar el estado de este proceso a *running*.
6. Se actualizan estructuras de datos para administración de la memoria.
7. Se restaura el contexto del procesador que existía en el momento en que el proceso seleccionado fue sacado del estado *running* por última vez. Esto implica cargar los valores del contador de programa y otros registros.

- Cambiar a modo usuario.
- Saltar a la posición apropiada en el programa de usuario para ejecutar ese programa.

**Process Control Block (PCB)** registro especial donde el sistema operativo agrupa toda la información que necesita conocer respecto a un proceso particular. Cada vez que se crea un proceso el sistema operativo crea el PCB correspondiente para que sirva como descripción en tiempo de ejecución durante toda la vida del proceso.

La información almacenada en un PCB incluye típicamente algunos o todos los campos siguientes:

- **Process State** - *Running, waiting, blocked, etc.*
- **Process ID, Parent Process ID.**
- **CPU registers and Program Counter** - *These need to be saved and restored when swapping processes in and out of the CPU.*
- **CPU-Scheduling information** - *Such as priority information and pointers to scheduling queues.*
- **Memory-Management information** - *E.g. page tables or segment tables.*
- **Accounting information** - *user and kernel CPU time consumed, account numbers, limits, etc.*
- **I/O Status information** - *Devices allocated, open file tables, etc.*

## 2.3 Algoritmos de planificación de procesos

**Long-term scheduling** decide qué proceso de la *ready queue* pasará a memoria.

**Medium-term scheduling** remueve temporalmente un proceso de la memoria y lo almacena en almacenamiento secundario (*swapping in-out*).

**Short-term scheduling** decide qué proceso en estado *ready* (en memoria) será ejecutado a continuación de una interrupción de reloj, de I/O, una *system call* o una señal. Toma decisiones luego de cada *time slice*. Este *scheduler* puede ser apropiativo (*preemptive* - remueve forzosamente procesos de la CPU) o no apropiativo (*non-preemptive*).



Los procesos que están en estado *ready* y esperando a ser ejecutados son mantenidos en una lista llamada *ready queue*. El encabezado de la lista contendrá punteros al primer y último PCB en la lista. La *ready queue* podría ser implementada como una cola FIFO, una cola de prioridad, un árbol, una pila, o simplemente una lista desordenada. Conceptualmente todos los procesos en la *ready queue* están listos y esperando por una chance para usar la CPU. Si un proceso que se está ejecutando debe esperar que se complete una operación de I/O, entonces es colocado en otra cola, llamada *device queue*. Cada dispositivo tiene su propia *device queue*.

<b>FIFO (<i>First In, First Out</i>)</b>	No apropiativo	La CPU se asigna a los procesos en el orden en el que la solicitan. Si se bloquea un proceso, se ejecuta el primer proceso en la cola. Si un proceso bloqueado pasa al estado <i>ready</i> , se coloca al final de la cola.
<b>SJF (<i>Shortest Job First</i>)</b>	No apropiativo	Se selecciona primero el proceso con menor tiempo de ejecución.
<b><i>Round Robin</i></b>		A cada proceso se le asigna un intervalo de tiempo ( <b>quántum</b> ), durante el cual puede ejecutarse. Si al finalizar el cuántum sigue ejecutándose, la CPU es apropiada para dársela a otro proceso, y el anterior se coloca al final de la cola de procesos. Si se bloquea antes de que termine el cuántum, se conmuta el proceso inmediatamente.
<b>Múltiples colas</b>		Se establecen <b>clases de prioridades</b> . Los procesos en la clase más alta se ejecutan durante 1 cuántum, los de la siguiente clase más alta en 2 cuántums, y así sucesivamente. Cada vez que un proceso utiliza todos sus cuántums, se mueve una clase hacia abajo en la jerarquía.

Cuadro 3: Algoritmos de planificación

## 2.4 Segmentos en un proceso

*When an executable program is read into system memory by the kernel and executed, it becomes a process. We can consider system memory to be divided into two distinct regions. One is user space, and the other is kernel space. Every process has its own user space (about 1GB virtual space) and are prevented from interfering with one another. The mode change which is from user mode to kernel mode is called a context switch.*

*The computer program memory is organized into the following:*

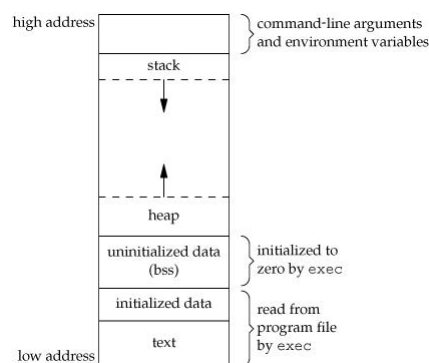


Figura 3: Txt, data, stack, heap

### 1. **Data Segment:**

- a) **Initialized data:** *global and static variables used by the program that are explicitly initialized with a value.*

- b) **Uninitialized data - BSS (Block Started by Symbol)**: global variables and static variables that are initialized to zero or do not have explicit initialization in source code.
  - c) **Heap**: The heap area is shared by all shared libraries and dynamically loaded modules in a process. The heap area begins at the end of the BSS segment and grows to larger addresses from there. The heap area is managed by `malloc`, `realloc`, and `free`, which may use the `brk` and `sbrk` system calls to adjust its size.
2. **Stack**: contains the program stack, a LIFO structure. Stores information about the active subroutines of a computer program. The stack segment is used by the process for the storage of automatic identifier, register variables, and function call information. The stack grows towards the uninitialized data segment.
  3. **Code Segment**: The Text segment (a.k.a the Instruction segment) contains the executable program code and constant data. The text segment is marked by the operating system as read-only and can not be modified by the process. Multiple processes can share the same text segment.
  4. **User Area**: the OS maintains for each process a region called the u area (User Area). The u area contains information specific to the process (e.g. open files, current directory, signal action, accounting information) and a system stack segment for process use. If the process makes a system call, the stack frame information for the system is stored in the system stack segment. Again, this information is kept by the OS in an area that the process doesn't normally have access to. Thus, if this information is needed, the process must use special system call to access it. Like the process itself, the contents of the u area for the process are paged in and out by the OS.

## 2.5 Creación y terminación de procesos

Creación	Terminación
<ul style="list-style-type: none"> <li>■ Al iniciar el sistema</li> <li>■ Con <i>system calls</i></li> </ul>	<ul style="list-style-type: none"> <li>■ Salida normal (voluntaria)</li> <li>■ Salida por error (voluntaria)</li> <li>■ Error fatal (involuntario)</li> <li>■ Muerte por otro proceso (involuntario)</li> </ul>

In Unix terminology, there are two system calls: **kill** and **exit**.

- **kill** sends a signal to another process. If this signal is not caught (via the signal system call) the process is terminated. There is also an uncatchable signal.
- **exit** is used for self termination and can indicate success or failure.

**Proceso huérfano** proceso cuyo padre ha terminado, pero que sigue corriendo. No se convierten en procesos zombies, son adoptados por `init` (proceso con PID 1), que espera a sus hijos.

**Proceso zombie** proceso que ha terminado su ejecución (estado *terminated*) pero que aún tiene una entrada en la tabla de procesos. Esta entrada se necesita para permitirle al proceso padre leer el *exit status* del hijo. La *system call* `kill` no tiene efecto en un proceso zombie.

Un proceso está en estado zombie cuando ejecutó una llamada a **exit** y aún no se limpiaron las estructuras de datos usadas por el proceso.

Una forma de obtener un proceso *zombie* es haciendo que el proceso hijo termine su ejecución con `exit()`, y que el proceso padre no ejecute `wait()`.

**fork** crea un proceso nuevo, duplicado del original, incluyendo todos los *file descriptors* y registros. Devuelve 0 en el proceso "hijo" y el PID del "hijo" en el "padre" (PID hijo  $\neq$  PID padre).

**exec** reemplaza la imagen de un proceso por un archivo que está en el disco.

**Imagen** copia de un programa.

---

**Algoritmo 1** Shell muy simplificada.

---

```
1 while (true) {
2     display_prompt();
3     read_command(command, parameters);
4     if (fork() != 0) {
5         // Proceso padre
6         // Se bloquea este proceso esperando al hijo
7         waitpid(-1, &status, 0); //simply removing the waitpid(...) gives background jobs.
8     } else {
9         // Proceso hijo
10        // User area distinta al padre
11        execve(command, parameters, environment);
12    }
13 }
```

---

	<b>fork()</b>	<b>exec()</b>
<b>Diferencias</b>	El proceso hijo es una copia exacta del padre. Se copian todos los datos ( <i>file descriptors</i> , registros, etc.). El proceso padre se bloquea esperando a que termine el hijo (si se utiliza <code>wait()</code> )	Carga una imagen del programa desde un archivo, en el proceso actual. Se reemplazan todos los datos, excepto los <i>file descriptors</i> .
<b>Estado del PCB</b>	Antes: hay un solo PCB (el del padre). Después: hay dos PCB, del padre y del hijo. El del hijo tiene la mayor parte de los campos iguales a los campos del padre (UID, registro, archivos abiertos, y algunos son distintos (PID, PPID))	Antes: hay un solo PCB (el del proceso). Después: hay un solo PCB (el del mismo proceso) con casi todos los mismos campos
<b>Estado del TXT</b>	Después: se copia todo del padre al hijo, se comparte entre ambos procesos.	Después: una vez ejecutado el <b>exec</b> , se reemplaza en el proceso actual, su área de TXT por las instrucciones del programa a ejecutar.
<b>Estado del U_AREA</b>	Antes: hay un U_AREA del proceso padre. Después: al nacer el proceso hijo, habrá dos U_AREA distintos (porque el U_AREA es propio de cada proceso)	Después: el proceso sigue siendo el mismo, solo cambió su imagen. Por lo tanto, su U_AREA es la misma.
<b>Estado de U_File Table</b>	La tabla se copia al proceso hijo.	No cambia.
<b>Estado de BSS</b>		Se modifica porque se cargan los valores estáticos del programa.
<b>Estado del Stack</b>		Cambia porque es utilizado por el programa que se carga.

## 2.5.1 Proceso de boot en Linux

1. Se carga el *First Stage Boot Loader* (ejemplo: GRUB - *Grand Unified Bootloader*).
2. Un *prompt* al usuario obtiene los datos de la partición y del kernel a bootear.
3. El kernel se carga como una imagen. Se descomprime. Se hace una inicialización de sus estructuras.
4. Se transfiere el control al proceso 0.

5. El proceso 0 detecta el tipo de CPU y hace una inicialización que depende de ésta.
6. El proceso 0 lanza la funcionalidad independiente de la arquitectura, llamando a `start_kernel()`. Se lanza a "init". "init" es el proceso 1.
7. "init" crea los procesos según `/etc/inittab`. Chequea y monta los filesystems según `/etc/fstab`.
8. "init" espera un login para lanzar un shell de usuario.

## 2.6 System y library calls

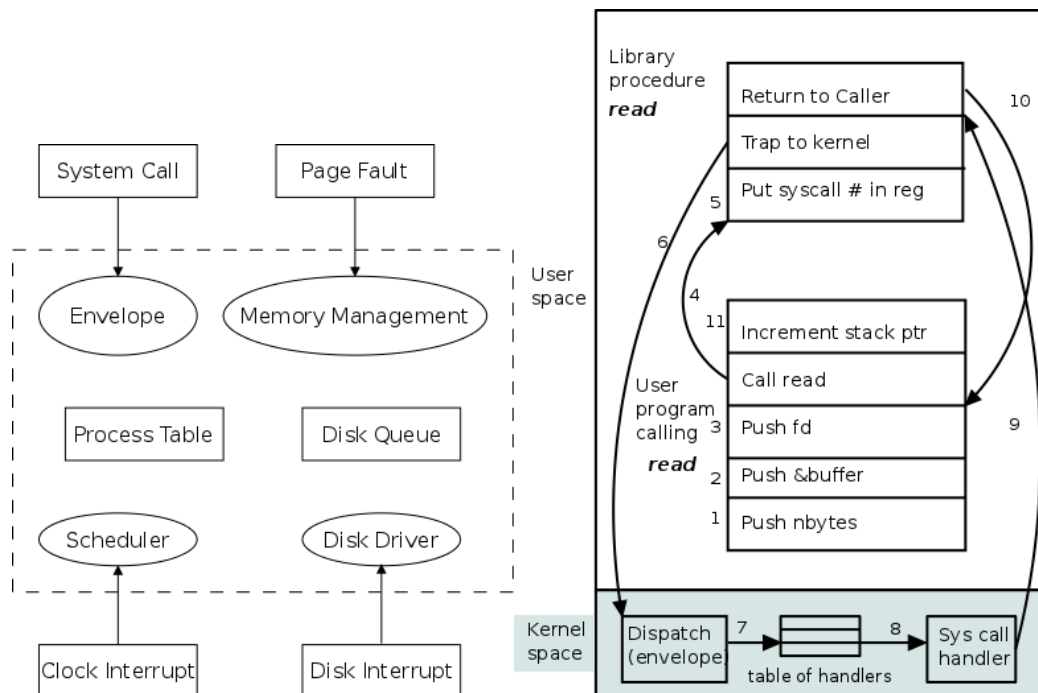


Figura 4: System calls

Los procesos se comunican con el kernel mediante *system calls*, que son funciones con un prototipo estandarizado.

- Ejemplos de *system calls*: `open`, `write`, `mmap`, `read`, `fstat`, `close`, `getpid`, `execve`
- Ejemplos de *library calls*: `printf`, `getpid`
- Para ver las *system calls* que hace un programa: `strace`. Para ver las *library calls* que hace un programa: `ltrace`.

**Blocking system call** System call that must wait until the action can be completed. `read()` would be a good example - if no input is ready, it'll sit there and wait until some is (provided you haven't set it to non-blocking, of course, in which case it wouldn't be a blocking system call). Obviously, while one thread is waiting on a blocking system call, another thread can be off doing something else.

Pasos para ejecutar la *system call* `read`:

```
1 int cuenta = read (fd, bufer, nbytes);
```

1. El programa mete los parámetros en la *stack* (primero `nbytes`,

2. luego **bufer**,
3. luego **fd**).
4. Se llama al procedimiento de biblioteca **read**.
5. El procedimiento **read** coloca el número de la *system call* **read** en un registro, para que lo lea el sistema operativo.
6. El procedimiento **read** realiza un *trap* para cambiar al **modo kernel**, saltando a una dirección fija.
7. El despachador busca en una tabla de handlers al handler que maneja esta llamada al sistema.
8. El manejador de llamadas al sistema maneja la llamada.
9. El manejador de llamadas al sistema le regresa el control al procedimiento de biblioteca **read**. Vuelve a **modo usuario** del procesador, el sistema operativo sigue en modo kernel.
10. El procedimiento de biblioteca le regresa el control al programa. El sistema operativo sale de modo kernel.
11. El programa limpia la pila, incrementando el *stack pointer*.

### 3 Threads

**Thread** hilo de ejecución de un proceso, que comparte el espacio de direcciones con otros *threads* del proceso.

*How threads differ from processes:*

- *processes are typically independent, while threads exist as subsets of a process*
- *processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources*
- *processes have separate address spaces, whereas threads share their address space*
- *processes interact only through system-provided inter-process communication mechanisms*
- *context switching between threads in the same process is typically faster than context switching between processes.*
- *there is no hierarchy between threads*
- *clock interruptions do not give the CPU to other threads; threads must give it voluntarily*
- *threads do not compete for resources like processes do*

*How threads are similar to processes:*

- *they can be in the same states (running, blocked, waiting)*
- *they are programmed in a sequential form*
- *they have a program counter, registers, stack*

**Multithreading** múltiples *threads* dentro de un mismo proceso.

Ventajas	Desventajas
<p><b>Responsiveness</b> - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.</p> <p><b>Resource sharing</b> - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.</p> <p><b>Economy</b> - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.</p> <p><b>Scalability</b>, i.e. Utilization of multiprocessor architectures. A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. (Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)</p>	<p><b>Identifying tasks</b> - Examining applications to find activities that can be performed concurrently.</p> <p><b>Balance</b> - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.</p> <p><b>Data splitting</b> - To prevent the threads from interfering with one another.</p> <p><b>Data dependency</b> - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.</p> <p><b>Testing and debugging</b> - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.</p>

Ejemplos de aplicaciones:

1. Servidor que lanza un *thread* por cada pedido.
2. Procesador de texto que tiene un *thread* que corrige la gramática, y otro *thread* que cuenta la cantidad de páginas del archivo.
3. Manejo de interfaces gráficas.

Implementación:

- Los *threads* comparten el espacio de memoria del proceso (TXT, DATA).
- Cada *thread* mantiene su propia información de estado en un *Thread Control Block* (TCB).

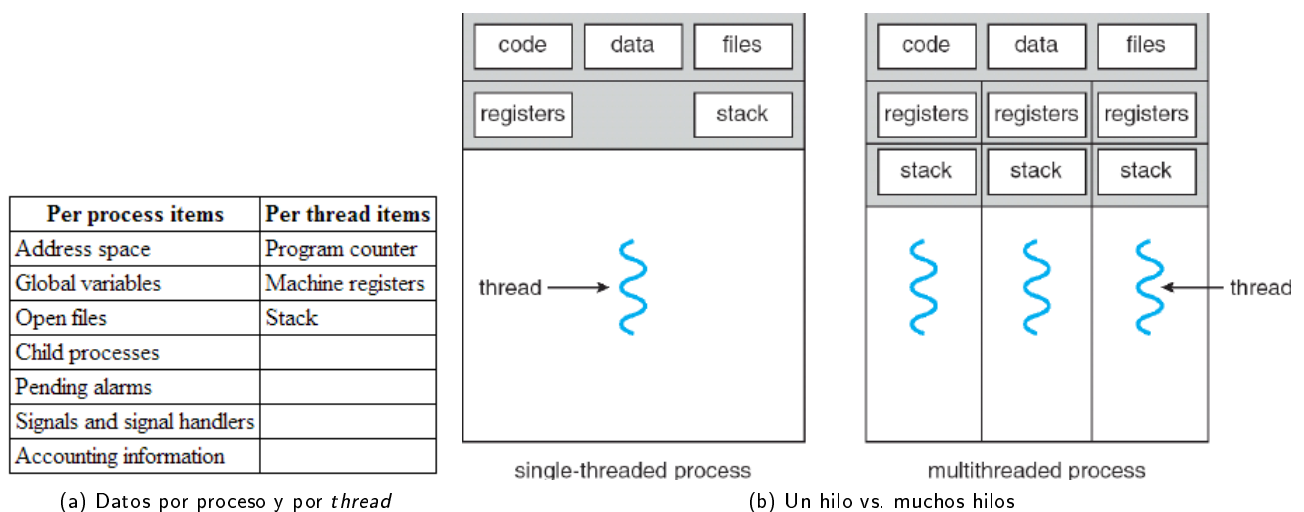


Figura 5: *Threads*

---

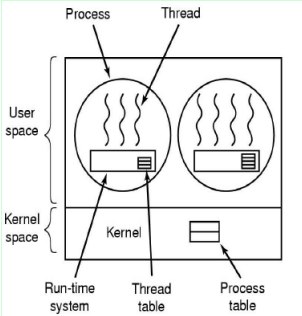
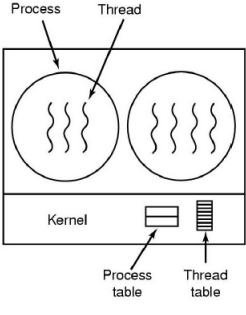
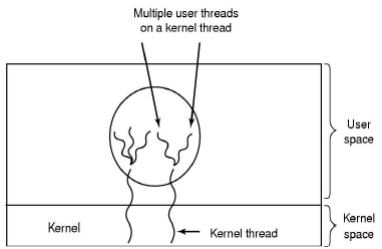
**Algoritmo 2** Programa de ejemplo que usa *threads*

---

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define NUMERO_DE_HILOS 10
6
7 void* imprimir_hola_mundo (void* tid) {
8     printf("Hola_mundo._Saludos_del_hilo_%d0_\n", (int)tid);
9     pthread_exit(NULL);
10 }
11 int main (int argc, char* argv[]) {
12     pthread_t hilos [NUMERO_DE_HILOS];
13     int estado, i;
14     for (i = 0; i < NUMERO_DE_HILOS; i++) {
15         printf("Aqui_main._Creando_hilo_%d0_\n", i);
16         estado = pthread_create(&hilos[i], NULL, imprimir_hola_mundo, (void*)i);
17         if (estado != 0) {
18             printf ("Ups._pthread_create_devolvió_el_código_de_error_%d0_\n", estado);
19             return -1;
20         }
21     }
22     return 0;
23 }
24
25 // Salida
26 $ gcc -pthread threads.c -o threads
27 $ ./threads
28 Aqui main. Creando hilo 00
29 Aqui main. Creando hilo 10
30 Aqui main. Creando hilo 20
31 Hola mundo. Saludos del hilo 00
32 Aqui main. Creando hilo 30
33 Hola mundo. Saludos del hilo 10
34 Aqui main. Creando hilo 40
35 Hola mundo. Saludos del hilo 30
36 Aqui main. Creando hilo 50
37 Hola mundo. Saludos del hilo 40
38 Hola mundo. Saludos del hilo 20
39 Aqui main. Creando hilo 60
40 Hola mundo. Saludos del hilo 50
41 Aqui main. Creando hilo 70
42 Hola mundo. Saludos del hilo 60
43 Aqui main. Creando hilo 80
44 Aqui main. Creando hilo 90
45 Hola mundo. Saludos del hilo 80
```

---

### 3.1 Implementación de threads

En espacio de usuario	En kernel	Híbrido
<p>Cada proceso necesita su tabla privada de <i>threads</i>. El kernel desconoce que hay <i>threads</i>. Se utilizan bibliotecas.</p> 	<p>No hay tabla de <i>threads</i> en cada proceso. El kernel tiene una tabla de <i>threads</i>.</p> 	<p>Utilizar hilos en espacio de usuario y después multiplexar los hilos de nivel usuario con alguno o con todos los hilos de nivel kernel. El kernel solo puede ver y planificar los hilos de nivel kernel.</p> 
<p>El kernel toma cada proceso y lo ejecuta en un <i>quantum</i> (intervalo de tiempo que se le asigna a un proceso para que se ejecute); por ejemplo, ejecuta el proceso A. El <i>scheduler</i> de <i>threads</i> dentro del proceso A decide qué <i>thread</i> ejecutar, por ejemplo A1. Este <i>thread</i> se ejecutará tanto como él quiera (ya que los <i>threads</i> no son interrumpidos), hasta que se consuma el <i>quantum</i>, y el kernel decida tomar otro proceso para ejecutar. Cuando el proceso A vuelva a ser ejecutado, el <i>thread</i> A1 continuará hasta volver a consumir todo el <i>quantum</i> del proceso, o hasta que termine su trabajo.</p>	<p>El kernel toma un <i>thread</i> y lo ejecuta. No tiene en cuenta a qué proceso pertenece dicho <i>thread</i>. Al <i>thread</i> se le da un <i>quantum</i>, y es forzado a suspenderse en caso de exceder dicho intervalo de tiempo. El kernel sabe que cambiar entre dos <i>threads</i> pertenecientes a distintos procesos es más costoso que si los <i>threads</i> pertenecen al mismo proceso (porque se requiere de un <i>context switch</i> completo). Por lo tanto, puede tener en cuenta esta información para decidir qué <i>thread</i> ejecutar en un determinado momento.</p>	
<p>(-) Si un <i>thread</i> comienza su ejecución, ningún otro <i>thread</i> va a correr en CPU hasta que el primero la ceda voluntariamente</p>	<p>(+) El kernel le puede sacar tiempo de CPU para otorgárselo a otro <i>thread</i></p>	
<p>(-) Si un <i>thread</i> realiza una <i>system call</i> bloqueante, los otros <i>threads</i> en el proceso no pueden correr hasta que la llamada termine</p>	<p>(+) Los <i>threads</i> no requieren <i>system calls</i> no bloqueantes. Cuando un <i>thread</i> bloquea, el kernel puede correr otro <i>thread</i>, tanto del mismo proceso como de otro</p>	
<p>(+) Se puede hacer <i>switching</i> de un <i>thread</i> a otro sin necesidad de usar <i>traps</i>, que son muy costosos. Se puede implementar el propio algoritmo de planificación de hilos</p>	<p>(-) Costo elevado para crear y destruir <i>threads</i>. (Se puede implementar "reciclado de <i>threads</i>")</p>	
<p>(+) Pueden ser implementados en un sistema operativo que no soporta <i>threads</i></p>	<p>(-) Cuando un proceso con <i>multithreading</i> realiza un <i>fork</i>, ¿cuántos <i>threads</i> debe tener el proceso nuevo?</p>	
<p>(-) No se beneficia de procesadores multihilo ni multiprocesadores: nunca hay más de un <i>thread</i> ejecutándose al mismo tiempo.</p>	<p>(-) Si un proceso recibe una señal, ¿qué <i>thread</i> debe manejarla?<sup>1</sup></p>	

Cuadro 4: Implementación de *threads*



## 4 Sistemas operativos móviles

### 4.1 Android

#### Capas de la arquitectura

1. *Applications* (home, contacts, phone, browser)
2. *Frameworks* (Activity Manager, Window Manager, Package Manager, Telephone Manager...)
3. *Libraries* (OpenGL, SSL, SQLite...) + *Android Runtime* (Dalvik VM<sup>2</sup>)
4. *Linux Kernel* (drivers)

**Aplicación** corre en su propio proceso con su propia copia de Dalvik. Vienen empaquetados en un **apk**. *An APK file contains all of that program's code (such as .dex files), resources, assets, certificates, and manifest file.*

Una vez que se instala la aplicación, tiene su *sandbox*. Cada apk es un usuario de Linux con permisos y directorios propios.

Para mantener la respuesta del sistema, Android puede matar sin previo aviso a un proceso y las aplicaciones contenidas.

**Componentes de una aplicación** descritos en el archivo *AndroidManifest.xml*. Se activan con un mensaje llamado *Intent*.

1. **Activities**: *An activity represents a single screen with a user interface. For example, an email application might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email application, each one is independent of the others.* Solo puede haber una *activity* activa a la vez, el resto se guarda en un *stack*. La prioridad de una *activity* se determina mediante su proceso. El programador extiende de la clase *Activity* y programa los eventos a los que responde.

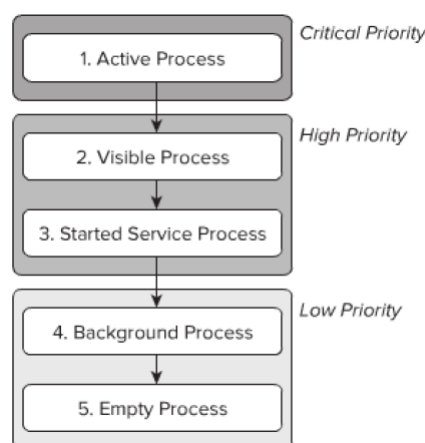


Figura 6: Prioridades de una *activity*

Ciclo de vida:

*Active*: está al tope de la *stack*, interactuando con el usuario.

<sup>2</sup>Los programas se escriben en Java, se compilan para generar *bytecode*, y se convierten de archivos *.class* a archivos *.dex* (Dalvik Executable).

*Paused:* está visible, pero sin foco.

*Stopped:* está en memoria pero ya terminó. Es candidata al *kill*.

*Inactive:* no está en memoria.

2. **Services:** *A service is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different application, or it might fetch data over the network without blocking user interaction with an activity.*
3. **Content providers:** *A content provider manages a shared set of application data. You can store the data in the file system, an SQLite database, on the web, or any other persistent storage location your application can access. Through the content provider, other applications can query or even modify the data (if the content provider allows it). For example, the Android system provides a content provider that manages the user's contact information. As such, any application with the proper permissions can query part of the content provider (such as `ContactsContract.Data`) to read and write information about a particular person.*
4. **Broadcast receivers:** *A broadcast receiver is a component that responds to system-wide broadcast announcements. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Applications can also initiate broadcasts—for example, to let other applications know that some data has been downloaded to the device and is available for them to use. Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs. A broadcast receiver is implemented as a subclass of `BroadcastReceiver` and each broadcast is delivered as an `Intent` object.*

## 4.2 iOS

Presenta cuatro capas de abstracción:

1. **Cocoa Touch:** *provides the key frameworks for developing applications on devices running iOS. Some of these key frameworks are: AirDrop (lets users share photos, documents and other data with nearby devices), TextKit (set of classes for handling text and fine typography), Multitasking, Storyboards (way of designing an app's user interface), Gesture recognizers*
2. **Media:** manejo de audio, video y gráficos.
3. **Core Services:** servicios fundamentales utilizados por las aplicaciones (iCloud, SQLite, XML...)
4. **Core OS:** frameworks de bajo nivel (*Accelerate, Core Bluetooth, Security, System...*)

Los procesos corren bajo dos UIDs: root (0) y mobile (501). Al pasar al *background*, los procesos quedan en estado suspendido y no ejecutan más código.

Apple trata de que no se usen *threads* en forma directa. *Instead of relying on threads, OS X and iOS take an asynchronous design approach to solving the concurrency problem. One of the technologies for starting tasks asynchronously is Grand Central Dispatch (GCD). It is an implementation of task parallelism based on the thread pool pattern. GCD works by allowing specific tasks in a program that can be run in parallel to be queued up for execution and, depending on availability of processing resources, scheduling them to execute on any of the available processor cores. Grand Central Dispatch still uses threads at the low level but abstracts them away from the programmer, who will not need to be concerned with as many details.*

## 5 Memoria

**Problema del direccionamiento absoluto** cuando un programa hace referencia a la memoria física absoluta, se pueden crear problemas.

- **Protección:** Puede estropearse el sistema operativo.

- **Reubicación:** Es difícil tener varios programas en ejecución a la vez.

Soluciones:

- **Reubicación estática:** modificar el programa a medida que se carga en memoria. Para ello se necesita información adicional en los programas, para indicar qué es una dirección (reubicable) y qué es una constante.
- **Reubicación dinámica:** asociar el espacio de direcciones de cada proceso sobre una parte distinta de la memoria física. Formas de implementación:
  - Registros base ( $b$ ) y límite ( $l$ ): cuando un proceso referencia una dirección absoluta  $x$ , la CPU verifica que  $b + x < l$ .  $b$  es la dirección física donde empieza el programa en memoria, y  $l$  es la longitud del programa.

**Problema de la sobrecarga de memoria** cuando un programa (o varios) son muy grandes y no caben completamente en la memoria.

Soluciones:

- **Swapping:** ubicar cada proceso completo en memoria, dejar que este corra un tiempo, y luego ubicarlo en el disco. Los procesos inactivos se encuentran en el disco, para que no consuman memoria innecesaria. Problemas:
  - Cuando por *swapping* se producen múltiples agujeros en la memoria, dichos agujeros se pueden combinar mediante una técnica (costosa) llamada **compactación de memoria**.
- **Memoria virtual:** permitir que un proceso se ejecute aun cuando solo se encuentra en la memoria de forma parcial.
- **Sobrepuestos (Overlays):** la construcción del programa consiste en dividir manualmente el mismo en bloques auto contenidos, dispuestos como una estructura de árbol.

**Espacio de direcciones** abstracción de la memoria física. Representa un conjunto de direcciones que un proceso puede utilizar.

## 5.1 Variables

- **Declarar** una variable: introducir el identificador.
- **Definir** una variable: asignarle memoria y posiblemente un valor inicial.
- **Ambiente:** porción de código durante el cual una variable está declarada.
- **Vida (Lifespan):** intervalo de ejecución en el que una variable tiene memoria asignada.
- **Ámbito (Scope):** variable en su ambiente y en su tiempo de vida.

*What happens to local variables when they go out of scope? Nothing physical happens. The memory formerly occupied by the variable continues to remain reserved in the stack until the function exits.*

```

1 void function()
2 { // comienza el ambiente
3   list* lista;           // declaracion
4   lista = NULL;          // definicion
5   while (true)
6   {
7     lista = (list*)malloc(100); // comienzo de lifespan
8
9     //....
10
11    free(lista);           // fin de lifespan
12  }
13 } // fin de ambiente

```

## Tipos de variables

- **Externas:** se declara pero no se define en el bloque.
- **Estáticas:** su vida se extiende a la duración de todo el programa.
- **Dinámica automática.** Se manejan con un *stack*.
- **Dinámica controlada:** se manejan con un *heap*.
  - **Sincrónico:** liberación manual con `free`, `delete`
  - **Asincrónico:** con el *garbage collector*

## 5.2 Administración de memoria libre

*Since the precise location of the allocation is not known in advance, the memory is accessed indirectly, usually through a pointer reference. The specific algorithm used to organize the memory area and allocate and deallocate chunks is interlinked with the kernel, and may use any of the following methods:*

- Bloques de tamaño fijo: usa una lista de bloques libres en memoria.
- *Buddy blocks*: la memoria se asigna en cantidades potencias de dos.

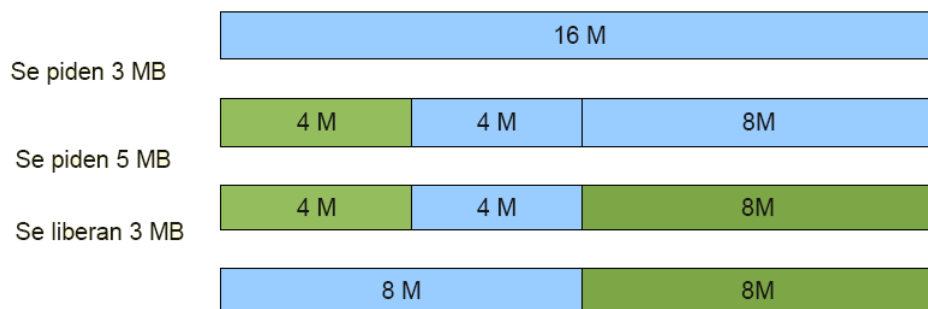


Figura 7: Buddy blocks.

	Bloques de tamaño fijo	Buddy blocks
Ventajas	<ul style="list-style-type: none"><li>■ Buena performance.</li></ul>	<ul style="list-style-type: none"><li>■ Permite una recuperación rápida de huecos grandes.</li><li>■ Implementación sencilla.</li></ul>
Desventajas	<ul style="list-style-type: none"><li>■ Sufre de fragmentación.</li></ul>	<ul style="list-style-type: none"><li>■ Sufre de fragmentación interna.</li></ul>

Algoritmos de alojamiento:

- *Best fit* (buscar el hueco más ajustado)
- *Worst fit* (buscar el hueco más holgado)
- *First fit* (buscar el primer hueco en que quepa)

## 5.3 Memoria virtual

**Memoria virtual** mecanismo diseñado para correr programas que son tan grandes que no caben en memoria.

**Paginado** es la herramienta más simple para eliminar el requisito de memoria física contigua.

El programa se divide lógicamente en partes de tamaño fijo llamadas **páginas**. Esta división es invisible para el usuario. Cada página es un rango continuo de direcciones. Estas páginas son mapeadas en memoria física, pero no todas las páginas deben estar en memoria para correr el programa. Cuando el programa referencia a una parte de su espacio de direcciones que no está en la memoria física, se produce un **page fault**: el sistema operativo recibe una alerta para bucar la página faltante y volver a ejecutar la instrucción que falló.

La memoria RAM se divide en partes de tamaño fijo llamados **frames**. El tamaño de página es igual al tamaño de *frame*.

Los programas referencian la memoria mediante **direcciones virtuales**. Esta dirección contiene un índice (número de página) y un *offset* (dentro de esa página). Pero el hardware solo entiende de **direcciones físicas**. Por lo tanto, la traducción de dirección virtual a real se realiza mediante la **Memory Management Unit** (MMU). La MMU puede encontrarse en el CPU o cerca del mismo.

Esta traducción puede ser lenta, por lo que se utiliza un **Translation Lookaside Buffer** (TLB).

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figura 8: TLB

La TLB es un cache (generalmente se encuentra dentro de la MMU) que contiene las últimas traducciones realizadas por la MMU. Se utiliza para no tener que acceder a la tabla de páginas, ya que ésta se encuentra en memoria. Hay *hit* cuando el numero de página que viene en la dirección virtual existe en la TLB, y hay un *miss* cuando no está.

Cómo funciona la TLB:

1. Cuando una dirección virtual se presenta en la MMU, el *hardware* chequea si el *virtual page number* se encuentra en la TLB, comparando todas las entradas simultáneamente.
2. Si es así, y el acceso no viola los bits de protección, el *page frame* es extraído directamente de la TLB sin usar la *page table*.
3. Si la página está presente pero hay una violación de permisos, se produce un *protection fault* tal como hubiera ocurrido con la *page table*.
4. Si la página no se encuentra, la MMU hace una búsqueda ordinaria en la *page table*:
  - a) Si la página solicita no está en la *page table*, se produce un *page fault*.
    - 1) Se selecciona un *frame* que se utilice poco.
    - 2) Si está "sucio", se escribe dicho *frame* en el disco. Si está limpio, simplemente se lo borra.
    - 3) Se obtiene la página del disco y se la guarda en el *frame* que se acaba de liberar.
    - 4) Luego sobrescribe alguna entrada de la TLB con la nueva página solicitada.

- 5) Se reinicia la instrucción que originó el *page fault*.

**Working Set** es el set de páginas que un proceso está utilizando en ese momento. Si un proceso tiene la totalidad de su *working set* en memoria, el mismo podrá ejecutarse sin producir ningún *page fault*.

**Thrashing** fenómeno que se produce cuando hay poco espacio en la memoria, y se produce un *page fault* por cada instrucción. El proceso pasa más tiempo paginando que ejecutándose.

### 5.3.1 Tabla de páginas “directa”

Se mantiene una **tabla de páginas** en memoria, que contiene una entrada por cada página. La tabla se indexa por número de página; por lo tanto, no almacena el número de página. Esta tabla suele ser muy grande, por lo que no se la almacena en memoria. **Cada proceso necesita su propia tabla de páginas.**

El mapeo se lleva a cabo de la siguiente forma:

1. Con los primeros  $n$  bits de la dirección virtual (*virtual page*) se accede a la correspondiente entrada en la *page table*.
2. Dada la entrada, se verifica si el *page frame* correspondiente a dicha virtual page se encuentra en memoria.
  - a) De ser así, la dirección física en memoria se obtiene concatenando el número de *page frame* y el *offset*.
  - b) En caso de no encontrarse en memoria el *page frame* (*page fault*), es necesario leerlo de disco. Previamente debe elegirse otro *page frame* (que no haya sido usado recientemente), grabarlo en disco (si fue modificado), para escribir el nuevo *page frame* en ese lugar de memoria.

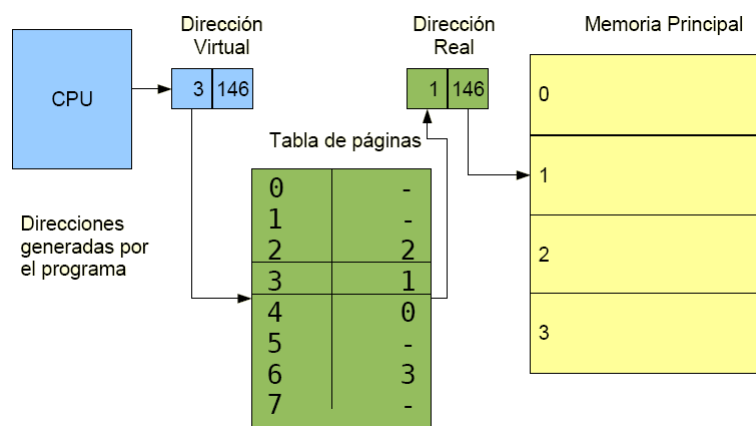


Figura 9: Tabla de páginas directa

Cada entrada contiene los siguientes campos:

- **The Frame Number.** This field is the main reason for the table. It gives the virtual to physical address translation. It is the only field in the page table for (non-demand) paging.
- **The Valid bit.** This tells if the page is currently loaded (i.e., is in a frame). If set, the frame number is valid. If a page is accessed whose valid bit is unset, a page fault is generated by the hardware.
- **The Modified or Dirty bit.** Indicates that some part of the page has been written since it was loaded. This is needed when the page is evicted so that the OS can tell if the page must be written back to disk.
- **The Referenced bit.** Indicates that some word in the page has been referenced. Used to select a victim: unreferenced pages make good victims by the locality property.

- **Protection bits.** For example one can mark text pages as execute only. This requires that boundaries between regions with different protection are on page boundaries. Normally many consecutive (in logical address) pages have the same protection so many page protection bits are redundant. Protection is more naturally done with segmentation, but in many current systems, it is done with paging (since the systems don't utilize segmentation, even though the hardware supports it).

### 5.3.2 Tablas de páginas “invertida” y “multinivel”

**Tabla de páginas “invertida”** Contiene una entrada por cada *frame*. Cada entrada lleva el registro de quién (proceso, página virtual) se encuentra en dicho *frame*.

La traducción de dirección virtual a dirección física es mucho más difícil. Cuando el proceso  $n$  hace referencia la página  $p$ , el hardware debe buscar una entrada  $(n, p)$  en **toda** la tabla de páginas invertida. Esta búsqueda debe hacerse en cada referencia a memoria, no solo en los *page fault*. Para solucionar esto, puede tenerse una tabla de hash arreglada según el hash de la dirección virtual.

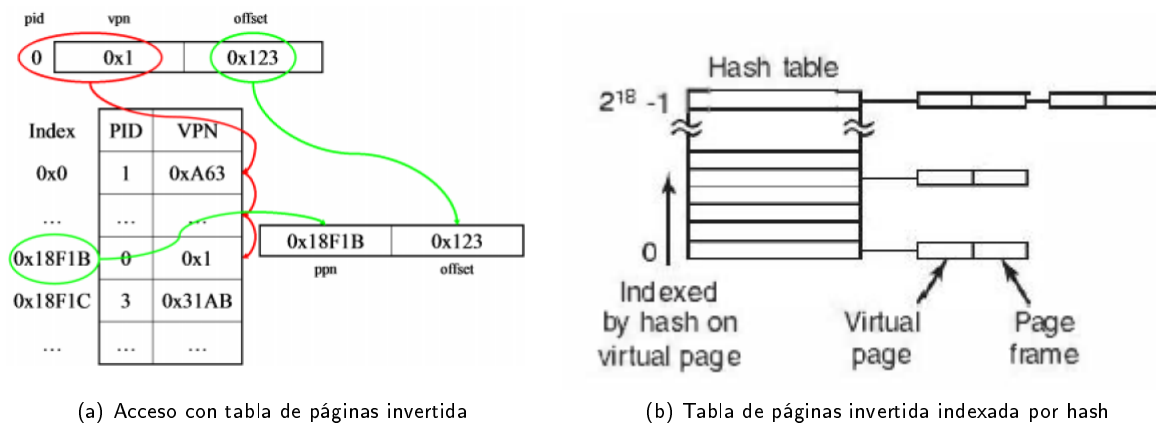


Figura 10: Tabla de páginas invertida

**Tabla de páginas “multinivel”** Dado que el espacio de direcciones virtuales puede ser muy grande la *page table* también resulta serlo. En un modelo en el que la tabla se encuentra en su totalidad en memoria, esto se traduce en un gasto inmenso de recursos. Como solución puede implementarse el uso de **Multilevel Page Table**.

Consiste de una tabla (en memoria) que referencia a todas las tablas (que están en memoria o no), y así se pueden agregar más niveles también. Cada vez que se pide una tabla que está en disco, se la deja en memoria para acceder más rápidamente a ella.

De la dirección virtual se identifican varios campos que son puntero a tablas y el *offset*. Si suponemos que tenemos 2 niveles, tenemos un campo para la tabla 1, otro para la tabla 2 y el *offset*. Se accede a la tabla 1 mediante el primer campo directamente (como tabla directa). De allí se obtiene un puntero a tabla 2, que nos indica cuáles de las tablas de segundo nivel debemos acceder (y se la carga en memoria). Mediante el campo de tabla 2 accedemos a esta segunda tabla, y ahí obtenemos la traducción (numero de *frame*) y si está en disco o en memoria. Con el numero de *frame* y el *offset* obtenemos la dirección física.

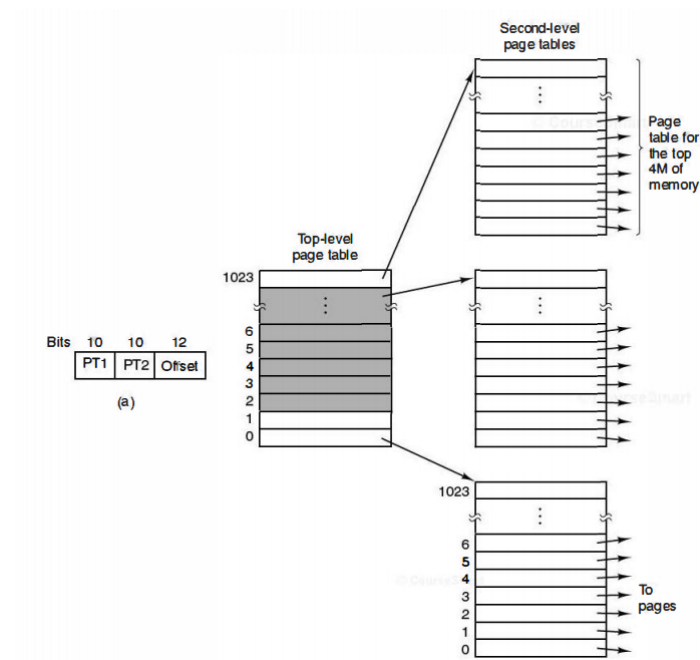


Figura 11: Tabla de páginas multinivel

### 5.3.3 Algoritmos de paginación

**Paginado por demanda** In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it (i.e., if a **page fault** occurs). It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages is located in physical memory.

When a page fault occurs, the system has a lot of work to do:

1. Software trap that blocks the process.
2. Choose a free frame, if one exists. What if there is no free frame? Make one! Choose a victim frame. Write the victim back to disk if it is dirty. Update the victim PTE to show that it is not loaded. Now we have a free frame.
3. Copy the referenced page from disk to the free frame.
4. Update the PTE of the referenced page to show that it is loaded and give the frame number.
5. Do the standard paging address translation ( $p\#, off \rightarrow f\#, off$ ).

**Pre-paginado** A technique whereby the operating system in a paging virtual memory multitasking environment loads all pages of a process's working set into memory before the process is restarted.

Under demand paging a process accesses its working set by page faults every time it is restarted. Under prepaging the system remembers the pages in each process's working set and loads them into physical memory before restarting the process. Prepaging reduces the page fault rate of reloaded processes and hence generally improves CPU efficiency.

### 5.3.4 Algoritmos de reemplazo de páginas

- **Óptimo:** elegir la página que se referencia más tarde. Es imposible de llevar a la práctica porque no se suele saber cuándo se va a referenciar una página, a menos que el proceso se ejecute dos veces y se saquen estadísticas en la primera corrida.



- **Not Recently Used (NRU)**: a cada página se le asocian dos bits de estado:

- R: se setea en 1 cuando se referencia la página (lectura o escritura)
- M: se setea en 1 cuando se escribe en la página (se modifica)

El bit R se borra en cada interrupción de reloj para diferenciar las páginas a las que no se ha hecho referencia recientemente.

Cuando ocurre un *page fault*, el sistema operativo inspecciona todas las páginas y las divide en 4 categorías con base en los valores actuales de sus bits R y M:

Clase 0	No ha sido referenciada, no ha sido modificada
Clase 1	No ha sido referenciada, ha sido modificada
Clase 2	Ha sido referenciada, no ha sido modificada
Clase 3	Ha sido referenciada, ha sido modificada

El algoritmo consiste en eliminar una página al azar de la clase de menor numeración que no esté vacía.

- **First In, First Out (FIFO)**: el sistema operativo mantiene una lista de todas las páginas actualmente en memoria, en donde la llegada más reciente está en la parte final y la menos reciente en la parte frontal. En un *page fault*, se elimina la página que está en la parte frontal de la lista, y la nueva página se agrega al final de la lista.

- Desventaja: podrían descartarse páginas de uso frecuente.

- **Segunda oportunidad**: es una modificación al algoritmo FIFO. Se inspecciona el bit R de la página más antigua. Si es 0, la página es antigua y no se ha utilizado, por lo que se sustituye de inmediato. Si el bit R es 1, el bit se borra, la página se pone al final de la lista de páginas (como si acabara de llegar a la memoria).

- **Reloj**: modificación al algoritmo de segunda oportunidad. Se mantienen los *frames* en una lista circular en forma de reloj. La manecilla apunta a la página más antigua.

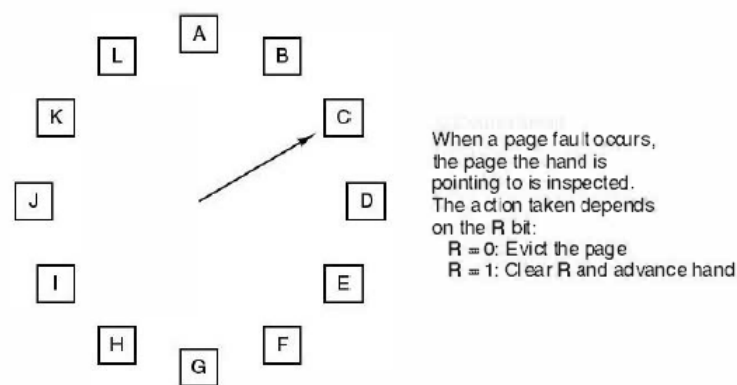


Figura 12: Algoritmo del reloj

- **Least Recently Used (LRU)**: cuando ocurre un *page fault*, se descarta la página que no se haya utilizado durante la mayor longitud de tiempo. Para esto es necesario mantener una lista enlazada de todas las páginas en memoria.

## 6 Linkediación

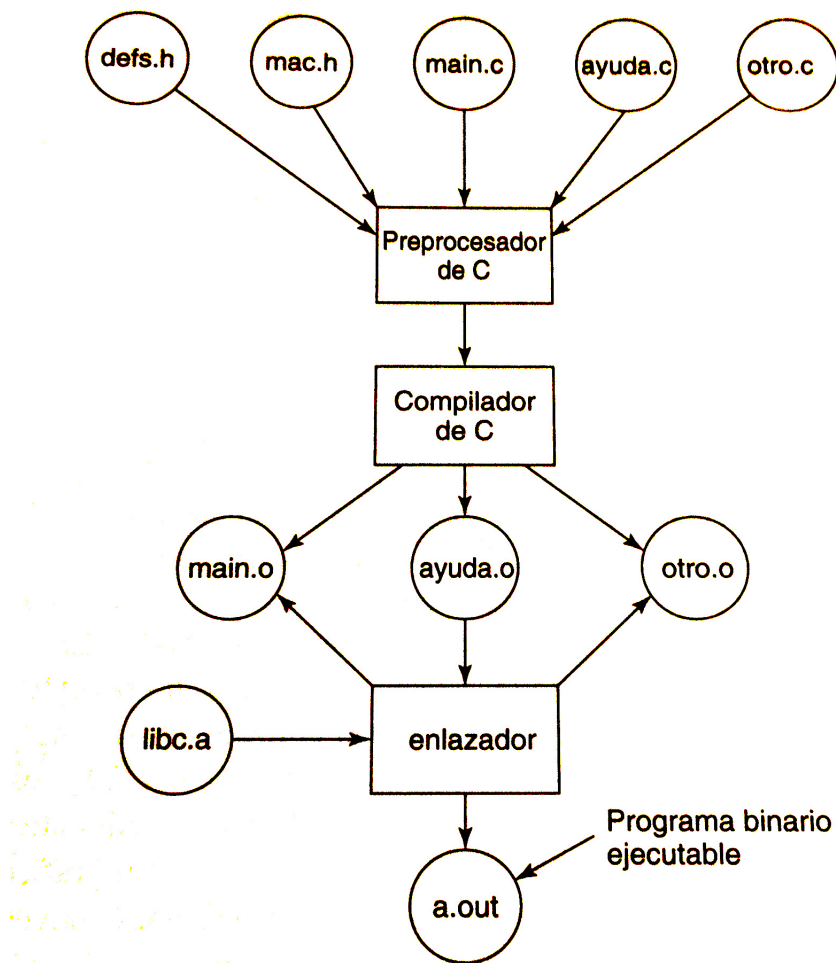


Figura 13: Compilación, ensamble, linker, loader.

**Traducción** proceso mediante el cual el código fuente de un programa se convierte en código objeto.

Si el lenguaje es ensamblador, la traducción es el **ensamblado**.

Si el lenguaje es de alto nivel, la traducción es una **compilación**.

- El compilador genera una **tabla de relocación** (*relocation table*), que es una tabla cuyas entradas son punteros a direcciones en el código objeto que deben ser modificadas cuando el *loader* realice el programa.
- El compilador genera una **tabla de símbolos** (*symbol table*), que es una lista de "items" en el archivo objeto que pueden ser utilizados por otros archivos. Pueden ser: nombres de funciones, o variables globales.

Un **object file** puede contener tres tipos de **símbolos**:

1. Símbolos definidos: le permiten ser llamado por otros módulos,
2. Símbolos no definidos: le permiten llamar a otros módulos que sí los tienen definidos
3. Símbolos locales: usados internamente para facilitar la relocación.

**Linker/Link editor** programa que toma uno o más programas objeto generados por un compilador y los combina en un solo programa ejecutable:

- Resuelve símbolos
- Relocaliza código que asume una dirección base específica

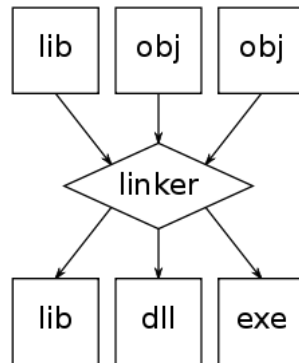


Figura 14: Proceso de *link*

**Linkediación** proceso de mezclar las direcciones de cada programa objeto en un único espacio de direcciones.

Entradas posibles: ejecutables, programa objeto, biblioteca estática.

Salidas posibles: ejecutables, programa objeto, biblioteca.

#### Procedimiento

1. Tomar el TXT de cada .o y ponerlos juntos.
2. Tomar el DATA de cada .o y ponerlos juntos, al final de los TXT.
3. Resolver referencias con la tabla de relocación

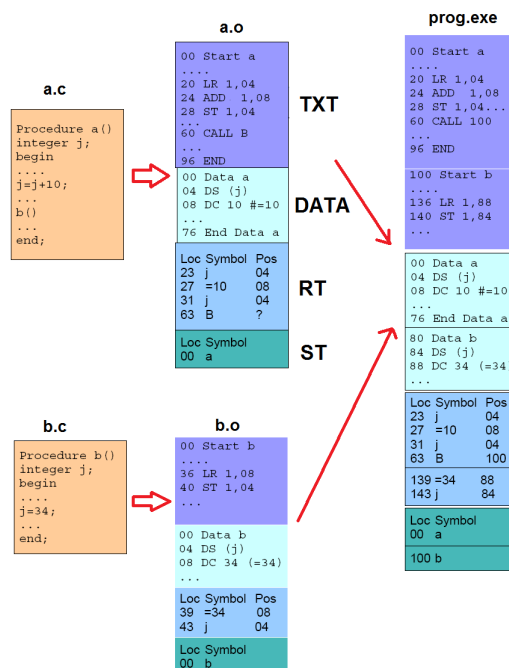


Figura 15: Procedimiento de compilación y linkediación

**Loader** parte de un sistema operativo que es el responsable de cargar programas en memoria para ser ejecutados.

El *loader* de Linux hace las siguientes tareas:

1. Validación (permisos, requisitos de memoria, etc.)
2. Copiar la imagen del programa del disco a la memoria principal
3. Copiar los argumentos por línea de comandos a la *stack*
4. Inicializar registros (el *stack pointer*, entre otros)
5. Bifurcar al punto de entrada del programa (*\_start*)

**Binding** proceso de resolver las llamadas a funciones de biblioteca. Puede darse en tiempo de compilación, en tiempo de carga, en tiempo de ejecución.

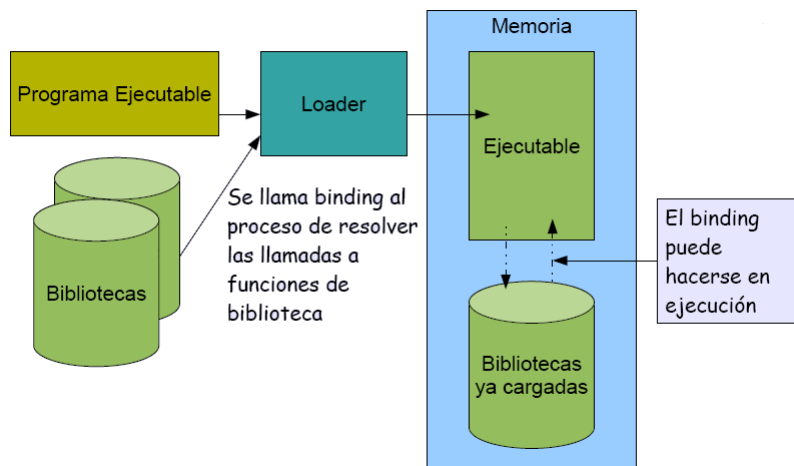


Figura 16: *Loader y binding*

## 6.1 Object File Formats

**Object File Format (OFF)** formato de archivo utilizado para el almacenamiento de código objeto e información relacionada.

- Afectan la performance del linker.
- Se puede utilizar el mismo formato para ejecutables, objetos y bibliotecas.
- Que dos sistemas operativos tengan el mismo OFF no significa que los programas de uno puedan correr en el otro, porque las llamadas al sistema pueden ser distintas.
- Los OFF soportan el uso de *headers*, segmento de texto, segmento de datos, información de relocación, definiciones externas, referencias para el *linking* e información de *linking* dinámico.

Un *object file* contiene 5 tipos de información:

- **Header information:** información general acerca del archivo, tales como el tamaño del código, nombre del archivo fuente del cual fue traducido y fecha de creación.
- **Código objeto:** instrucciones binarias generada por un compilador o ensamblador.
- **Relocation table:** una lista de los lugares en el código objeto que tienen que ser modificados cuando el linker cambia la dirección del código objeto.

- **Symbol table:** símbolos globales definidos en el módulo, símbolos a ser importados de otros módulos o definidos por el linker.
- **Debugging information:** otra información sobre el código objeto, no necesitada para el linkeo pero sí para el uso de un *debugger*. Esto incluye archivos fuentes e información sobre números de línea, símbolos locales, descripción de estructuras de datos usadas por el código objeto, tales como definiciones de estructuras de C.

### 6.1.1 com (*Command File*)

- Creado por Microsoft.
- No tiene previsión para relocación en memoria. No hay tabla de relocación. Se cargan en una dirección fija de memoria (0x100).
- El código y los datos (lo único que tiene el archivo, ya que no hay ni *header* ni metadatos) están en el mismo segmento.
- Su tamaño máximo es de casi 64 KB.

### 6.1.2 exe

- Creado por Microsoft.
- Los archivos comienzan con los caracteres "MZ" (4D 5A), las iniciales del diseñador del formato, Mark Zbikowski.
- El archivo tiene un *header*, la tabla de relocación, y la imagen binaria del TXT.

### 6.1.3 coff (*Common Object File Format*)

- Creado por Unix.
- El archivo tiene secciones separadas por *headers*. El archivo tiene información de *debugging*.

### 6.1.4 pe (*Portable Executable*)

- Versión modificada del formato coff.
- Tiene definido espacio para *resources* (metadata de solo lectura: (iconos, menús, *bitmaps*, *templates*, *fonts*, etc.)
- Tiene definido tablas para el uso de bibliotecas compartidas.
- Hay herramientas de análisis.

### 6.1.5 elf (*Executable and Linkable Format*)

- Sirve para ejecutables y bibliotecas.
- Tiene previsiones para emulación.

## 7 Bibliotecas

**Biblioteca (*Library*)** colección de subprogramas usados en el desarrollo de software. Contienen “código auxiliar” que brindan servicios a distintos programas, facilitando la modularización del código.

	Biblioteca estática	Biblioteca dinámica o compartida
	El link-editor incluye la biblioteca en el ejecutable.	El link-editor solo indica las llamadas. Éstas se resuelven: durante la carga o durante la ejecución. Pueden cargarse: <ul style="list-style-type: none"> <li>■ Antes de <i>loading</i></li> <li>■ En tiempo de <i>loading</i></li> <li>■ En tiempo de ejecución</li> </ul>
Ventajas	El ejecutable contiene todo lo que se necesita. Facilita la portabilidad.	<ul style="list-style-type: none"> <li>■ Las bibliotecas (ejemplo: STL en C++) solo se necesitan guardar en un lugar en memoria. Por ende, el ejecutable es mucho más pequeño.</li> <li>■ Si un error en una función de biblioteca se arregla reemplazando la biblioteca, todos los programas que la usen dinámicamente se benefician de la corrección reiniciándolos (no hace falta re-linkear)</li> <li>■ Las funciones de biblioteca que no son necesarias, no se cargan en memoria.</li> </ul>
Desventajas	<ul style="list-style-type: none"> <li>■ El ejecutable contiene <i>toda</i> la biblioteca, incluso si hay partes que no se usarán.</li> <li>■ Si la biblioteca se actualiza, no obtendremos los beneficios a menos que recompilemos con el nuevo código fuente.</li> </ul>	<ul style="list-style-type: none"> <li>■ En Windows, una DLL actualizada puede romper ejecutables que dependían del funcionamiento de la versión vieja de ese DLL</li> <li>■ Un programa y las bibliotecas que usa sólo se pueden validar como un paquete, no si sus componentes se pueden reemplazar</li> <li>■ Hay pérdida de tiempo en la ejecución, porque hay que linkear</li> <li>■ Si la biblioteca no es PIC, deben crearse páginas para cada proceso que comparta la biblioteca</li> </ul>
Compilación en Linux	<code>ld *.o --static libfoo.a</code>	<code>ld *.o -luno -ldos</code>

Cuadro 5: Tipos de bibliotecas

---

**Algoritmo 3** Load-Time Dynamic Linking (Windows)

---

```
1 #include <windows.h>
2
3 extern "C" int __cdecl myPuts(LPWSTR); // a function from a DLL
4
5 int main(VOID)
6 {
7     int Ret = 1;
8     // If the system cannot locate a required DLL, it terminates the process
9     // and displays a dialog box that reports the error to the user.
10    Ret = myPuts(L"Message_sent_to_the_DLL_function\n");
11    // The DLL is mapped into the virtual address space of the process
12    // during its initialization and is loaded into physical memory only when needed.
13    return Ret;
14 }
```

---

---

**Algoritmo 4** Run-Time Dynamic Linking (Windows)

---

```
1 // A simple program that uses LoadLibrary and
2 // GetProcAddress to access myPuts from Myputs.dll.
3 #include <windows.h>
4 #include <stdio.h>
5
6 typedef int (__cdecl *MYPROC)(LPWSTR);
7
8 int main( void ) {
9     HINSTANCE hinstLib;
10    MYPROC ProcAdd;
11    BOOL fFreeResult, fRunTimeLinkSuccess = FALSE;
12
13    // Get a handle to the DLL module.
14    hinstLib = LoadLibrary(TEXT("MyPuts.dll"));
15    // If the handle is valid, try to get the function address.
16    if (hinstLib != NULL)
17    {
18        ProcAdd = (MYPROC) GetProcAddress(hinstLib, "myPuts");
19        // If the function address is valid, call the function.
20        if (NULL != ProcAdd)
21        {
22            fRunTimeLinkSuccess = TRUE;
23            (ProcAdd) (L"Message_sent_to_the_DLL_function\n");
24        }
25        // Free the DLL module.
26        fFreeResult = FreeLibrary(hinstLib);
27    }
28    // If unable to call the DLL function, use an alternative.
29    if (! fRunTimeLinkSuccess)
30        printf("Message_printed_from_executable\n");
31
32    return 0;
33 }
```

---

---

**Algoritmo 5** Run-Time Dynamic Linking (Linux)

---

```
1 #include <stdio.h>
2 #include <dlfcn.h>
3 #include "ctest.h"
4
5 int main(int argc, char **argv) {
6     double (*fn)(int *);
7     int x;
8     char *error;
9
10    // Open shared library named "libctest.so".
11    // The second argument indicates the binding.
12    void* lib_handle = dlopen("/opt/lib/libctest.so", RTLD_LAZY);
13
14    if (lib_handle == NULL)
15    {
16        fprintf(stderr, "%s\n", dlerror());
17        exit(1);
18    }
19
20    // Returns address to the function which has been loaded with the shared library.
21    fn = dlsym(lib_handle, "ctest1");
22    if ((error = dlerror()) != NULL)
23    {
24        fprintf(stderr, "%s\n", error);
25        exit(1);
26    }
27
28    (*fn)(&x);
29    printf("Valx=%d\n", x);
30    dlclose(lib_handle);
31    return 0;
32 }
```

**Position-independent code (PIC)** código máquina que al ser ubicado en algún lugar de la memoria, se ejecuta correctamente independientemente de su dirección absoluta. Se suele utilizar en bibliotecas dinámicas, para que el mismo código pueda cargarse en una dirección en cada espacio de usuario de cada proceso.

Un programa PIC puede ejecutarse en cualquier dirección de memoria sin necesidad de modificarlo. Esto se diferencia del **relocatable code**, donde el link-editor o el *loader* deben modificar el programa para que pueda correr una dirección específica. El código PIC debe cumplir ciertas reglas y el compilador debe soportarlo. Las instrucciones que se refieran a posiciones de memoria específicas deben ser reemplazadas por instrucciones relativas. La indirección extra puede hacer que el código PIC sea menos eficiente.

*Data references from position-independent code are usually made indirectly, through **global offset tables (GOTs)**, which store the addresses of all accessed global variables. There is one GOT per compilation unit or object module, and it is located at a fixed offset from the code (although this offset is not known until the library is linked). When a linker links modules to create a shared library, it merges the GOTs and sets the final offsets in code. It is not necessary to adjust the offsets when loading the shared library later.*

*Microsoft Windows DLLs are not shared libraries in the Unix sense and do not use position independent code.*

## 7.1 Búsqueda de rutinas en bibliotecas



Linux	Windows
Hay una ruta de búsqueda formada por la posición fija indicada en el programa, archivo de configuración <code>/etc/ld.so.conf</code> , variable de ambiente <code>LD_LIBRARY_PATH</code>	Los ActiveX se buscan en el registro (registry). El resto en el indicado en una llamada a <code>SetDllDirectory()</code> , <code>System32</code> , <code>System</code> y <code>Windows</code>

## 8 Virtualización

**Virtualización** acto de crear una versión virtual (no real) de una cosa

Los sistemas de máquinas virtuales son capaces de virtualizar un conjunto completo de recursos de *hardware*, incluyendo al procesador (o procesadores), memoria, recursos de almacenamiento y dispositivos periféricos.

Aplicaciones:

- Aumento de confiabilidad.
- Ejecución de aplicaciones antiguas.
- Desarrollo y prueba en múltiples plataformas.
- Balanceo de cargas y escalabilidad futura.

### 8.1 Tipos de virtualización

#### 1. Virtualización de aplicaciones

Encapsula un software de aplicación del sistema operativo subyacente. La aplicación se comporta, en tiempo de ejecución, como si estuviese interactuando directamente con el sistema operativo original y los recursos que éste maneja, pero puede ser aislada. En este contexto, el término “virtualización” se refiere al objeto que está siendo encapsulado: la aplicación.

Ejemplo: *Wine* permite ejecutar aplicaciones de Windows en Linux.

#### 2. Virtualización de escritorio

Separa el ambiente de escritorio del dispositivo físico que es utilizado para acceder a él.

Ejemplo: *Remote Desktop*.

#### 3. Virtualización de recursos

Utilizar los recursos del sistema operativo *host* (red, memoria, almacenamiento...) para apoyar la ejecución del *guest*.

Ejemplos:

- RAID.
- *coLinux* permite que los kernels de Windows y Linux corran en paralelo en la misma computadora.

#### 4. Virtualización del sistema operativo

El kernel de un sistema operativo permite múltiples espacios de usuario aislados, los *containers*. Puede pensarse como una implementación avanzada del mecanismo *chroot*. El kernel suele proveer mecanismos de administración de recursos para limitar el *impacto de las actividades de un container en otros containers*.

#### 5. Virtualización de plataforma (*hardware*)

*Refers to the creation of a virtual machine that acts like a real computer with an operating system. Software executed on these virtual machines is separated from the underlying hardware resources.*

The **host** machine is the actual machine on which the virtualization takes place, and the **guest** machine is the virtual machine. The software or firmware that creates a virtual machine on the host hardware is called a **hypervisor** or **Virtual Machine Manager**.

Different types of hardware virtualization include:

- **Full virtualization**: Almost complete simulation of the actual hardware to allow software, which typically consists of a guest operating system, to run unmodified.
- **Partial virtualization**: Some but not all of the target environment is simulated. Some guest programs, therefore, may need modifications to run in this virtual environment.
- **Paravirtualization**: Presents a software interface to virtual machines that is similar, but not identical to that of the underlying hardware. The intent of the modified interface is to reduce the portion of the guest's execution time spent performing operations which are substantially more difficult to run in a virtual environment compared to a non-virtualized environment.
  - It modifies the guest operating system to eliminate the need for binary translation.
  - Requires using specially modified operating system kernels. Paravirtualization requires the guest operating system to be explicitly ported for the para-API.

## 8.2 Requerimientos de virtualización de Popek y Goldberg

Son un conjunto de condiciones suficientes para que una arquitectura de computadores soporte eficientemente la virtualización. Constituyen una manera eficaz de determinar si una arquitectura soporta eficientemente la virtualización, y proporciona líneas maestras para el diseño de arquitecturas virtualizables.

Hay tres características de interés cuando se analiza el entorno creado por un VMM:

1. **Equivalencia / Fidelidad** Un programa corriendo bajo el VMM debe exhibir un comportamiento esencialmente idéntico a aquel demostrado cuando se ejecuta directamente en una máquina equivalente.
2. **Control de recursos / Seguridad** El VMM debe estar en completo control de los recursos virtualizados.
3. **Eficiencia / Performance** Una fracción estadísticamente dominante de las instrucciones de máquina debe ser ejecutada sin la intervención del VMM.

Popek y Goldberg clasifican las instrucciones de un ISA en 3 grupos:

1. **Instrucciones privilegiadas** generan un *trap* si el procesador se encuentra en modo de usuario pero no lo hacen si se encuentra en modo kernel.
2. **Instrucciones sensibles** solo pueden ejecutarse en modo supervisor del procesador.
  - a) **de control** intentan cambiar la configuración de los recursos en el sistema.
  - b) **de comportamiento** su comportamiento o resultado dependen de la configuración de los recursos.

*Teorema. Se puede construir un VMM efectivo si el conjunto de instrucciones sensibles es un subconjunto de las instrucciones privilegiadas.*

Para poder virtualizar, la idea es generar “*containers*” donde la ejecución de una instrucción delicada provoque un software *trap*.

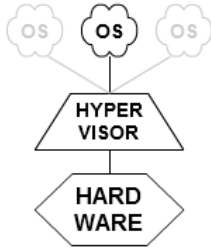
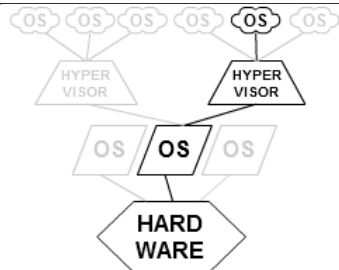
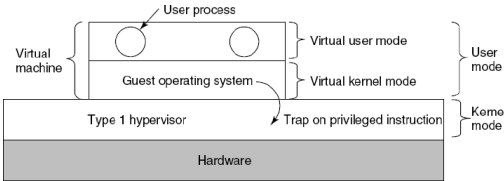
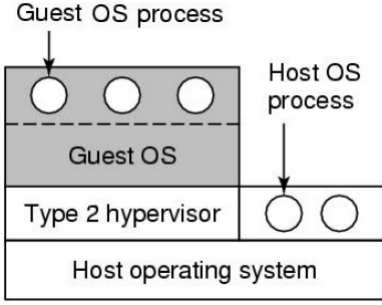
## 8.3 Hipervisores

**Hypervisor - Virtual Machine Monitor (VMM)** software, firmware o hardware que provee la abstracción de una máquina virtual. Permite crear y correr máquinas virtuales.

**Anfitrión (host)** computadora en la que un hipervisor esta corriendo una o más máquinas virtuales.

**Huésped (guest)** una máquina virtual corriendo en un anfitrión.

El hipervisor le presenta al sistema operativo huésped una plataforma virtual operativa y maneja la ejecución de los sistemas operativos huéspedes.

	Tipo 1	Tipo 2
		
		
<b>Anfitrión</b>	El hipervisor corre directamente en el hardware del anfitrión para controlar dicho hardware y administrar los sistemas operativos huéspedes. El huésped debe tener una arquitectura virtualizable.	El hipervisor corre dentro de un sistema operativo convencional. El anfitrión puede modificar el programa que está corriendo mediante la <b>traducción binaria</b> .
<b>Huésped</b>	El sistema operativo huésped corre en modo usuario. Su kernel cree haber pasado a modo supervisor, pero continúa en modo usuario.	El sistema operativo huésped corre como una tercer capa por encima del hardware.
<b>Ejecución de instrucciones delicadas</b>	Al ejecutar una instrucción delicada, se produce una <i>software trap</i> .	<i>On most modern CPUs, context-sensitive instructions are Non-Virtualizable. <b>Binary translation</b> is a technique to overcome this limitation. The sensitive instructions in the binary of Guest OS are replaced by either Hypervisor calls which safely handle such sensitive instructions or by some undefined opcodes which result in a CPU trap. Such a CPU trap is handled by the Hypervisor.</i>
<b>Software ejemplo</b>	Hyper-V	Virtual Box
<b>Ventajas y desventajas</b>	(+) El hipervisor corre sobre hardware, no necesita un sistema operativo. (+) Se pueden tener muchas VMs en el mismo hardware. (-) Requisitos de hardware son estrictos. (-) No siempre es más rápido que el tipo 2, ya que los <i>traps</i> consumen muchos recursos.	(+) Pueden virtualizar cualquier ambiente. (-) No se pueden tener muchas VMs.

Cuadro 6: Tipos de hipervisores

**Traducción binaria** emulación de un set de instrucciones por otro set de instrucciones mediante la traducción de código. *In some cases such as instruction set simulation, the target instruction set may be the same as the source instruction set, providing testing and debugging features such as instruction trace, conditional breakpoints and hot spot detection.*

- **Estática:** *aims to convert all of the code of an executable file into code that runs on the target architecture without having to run the code first.*
- **Dinámica:** *looks at a short sequence of code—typically on the order of a single basic block—then translates it and caches the resulting sequence. A **basic block** is a portion of the code within a program with only one entry point and only one exit point.*

## 9 Archivos y Sistemas de archivo

**Sistema de archivo** parte del sistema operativo que administra los archivos. Los sistemas de archivos se almacenan en disco.

Presenta dos aspectos:

- Interfaz de usuario. La interfaz exporta la noción de:
  - Directorios
  - Archivos
- Implementación.

### 9.1 Archivos

Requerimientos para el almacenamiento de información a largo plazo:

- Debe ser posible almacenar mucha información.
- La información debe sobrevivir a la terminación del proceso que la utilice.
- Varios procesos deben ser capaces de acceder a la misma información concurrentemente.

**Archivo** unidad lógica de información creada por los procesos. Abstrae las propiedades físicas del dispositivo de almacenamiento.

Tipos de archivos según la función que cumplen:

- *Archivos regulares:* contienen información del usuario. Pueden ser **ASCII** o binarios.
- *Archivos especiales de caracteres:* modelan dispositivos de E/S.
- *Archivos especiales de bloques:* modelan discos.
- *Archivos especiales de red:* para comunicar procesos.
- *Directorios:* mantienen la estructura del sistema de archivos.

Posibles atributos de archivos:

- Nombre, ubicación, tamaño, tipo.
- Atributos de **protección:** usuarios permitidos, contraseña, creador, propietario.
- Atributos **bandera** (binarios): “sólo lectura”, “oculto”, “del sistema”, “de archivo”, “ascii/binario”, “de acceso aleatorio”, “temporal”, “de bloqueo”.

- Atributos **extras**: longitud de registro, posición de la llave en el registro, longitud de la llave, hora de creación, hora de último acceso, hora de última modificación, tamaño actual, tamaño máximo.

Operaciones posibles en un archivo:

Create	Open	Read	Seek	Get attributes	Rename
Delete	Close	Write	Append	Set attributes	

- **Create** crea un archivo sin datos y establece algunos atributos.
- **Open** lleva los atributos y la lista de direcciones de disco a memoria principal. Se debe especificar un modo de apertura.
- **Close** obliga a escribir el último bloque del archivo (*flush*).
- **Read** requiere tres parámetros: el *file descriptor* (entero que referencia al archivo), cuántos datos se necesitan leer, y dónde colocarlos (un buffer).
- **Rename** evita tener que copiar el archivo en otro nuevo con el nuevo nombre.

## 9.2 Directorios

Tipos de directorio:

- *Directorios jerárquicos*, son árboles con uno o más niveles.
- *Directorio de un solo nivel*, que puede simular un directorio jerárquico si a los archivos los nombramos como si fueran un directorio. Ejemplo: "usr/ast/file".

Nombres de rutas:

- **Absoluto**: ruta desde el directorio raíz al archivo.
- **Relativa**: se utiliza en conjunto con el directorio de trabajo.

Operaciones posibles en un directorio:

Create	Open	Read	Link	Rename
Delete	Close		Unlink	

- **Link**: llamada al sistema que permite a un archivo aparecer en más de un directorio.
  - **Hard link**: relación bidireccional entre una ruta y un archivo. Dos rutas apuntan a la misma estructura que contiene metadatos sobre un archivo.  
En UNIX BSD: Al hacer un *hard link*, se incrementa el contador de *hard links* en el nodo-i del archivo. Todos los nodos-i tienen, al menos, 1 en su contador. Cuando se borra un *hard link*, el contador decrece. Cuando el contador llega a cero, el archivo se borra definitivamente.
  - **Soft link / symbolic link**: relación unidireccional entre una ruta y un archivo.  
En UNIX BSD: La carpeta original (1) apunta al nodo-i del archivo X. El siguiente directorio (2) que quiera tener una referencia al archivo, crea un archivo que contiene la ruta del archivo original X. Si se borra el archivo X, la ruta (2) tiene un puntero inválido.

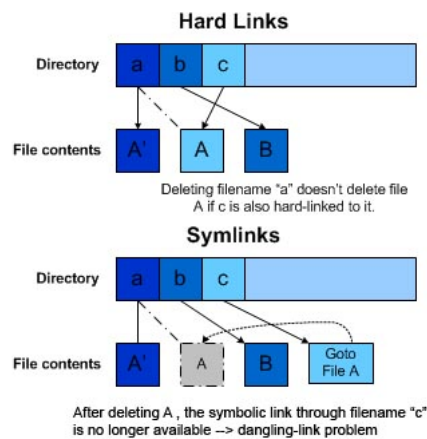


Figura 17: Hard links y soft links

Los *hard links* no requieren espacio extra en el disco, y los *soft links* sí.

Los *soft links* pueden apuntar a archivos en otras máquinas, en la misma red o por Internet. Los *hard links* solo pueden apuntar a archivos dentro de la misma partición.

## 9.3 Distribución del sistema de archivos

Ejemplo: UNIX BSD.

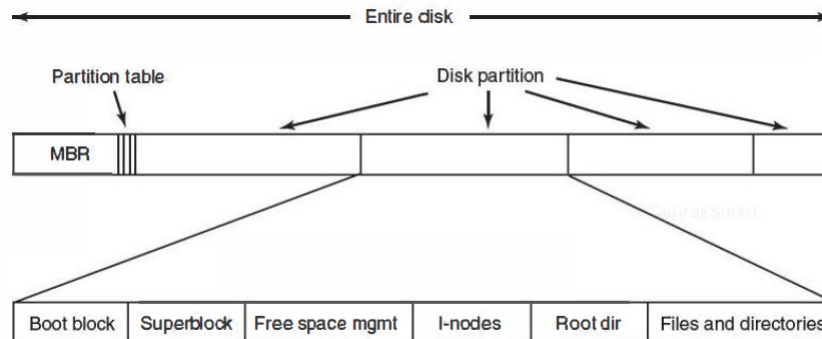


Figura 18: Distribución posible de un sistema de archivos

1. **MBR** (*Master Boot Record*): se utiliza para arrancar la computadora.
2. **Tabla de particiones**: proporciona las direcciones de inicio y fin de cada partición. Proporciona el nombre de la partición activa.
3. Para cada partición:
  - a) **Boot block**: contiene el módulo de arranque del sistema operativo de esa partición.
  - b) **Superblock**: contiene todos los parámetros clave acerca del sistema de archivos.
  - c) Administración del **espacio libre**: información acerca de los bloques libres, en forma de *bitmap* o una lista de punteros.

- d) **Nodos-i**: arreglo de estructuras de datos, uno por archivo, que indica todo acerca del archivo. La cantidad de nodos-i es **fija**, y se establece al crear la partición.
- e) Directorio raíz.
- f) Todos los otros directorios y archivos.

## 9.4 Implementación de archivos

Objetivo: mantener un registro acerca de qué bloques del disco van con cuál archivo.

Formas:

1. **Asignación contigua**: almacenar cada archivo como una serie contigua de bloques de disco. Cada archivo comienza al inicio de un nuevo bloque, y puede desperdiciarse espacio en el último bloque de cada archivo.

Ventajas:

- a) Es simple de implementar. Sólo hay que **guardar**, por cada archivo, **la dirección de disco del primer bloque y la cantidad de bloques que ocupa**.
- b) El **rendimiento de lectura** es excelente. El archivo completo se puede leer en una sola operación.

Desventajas:

- a) Con el tiempo, el disco se **fragmenta**. Cuando se quita un archivo quedan bloques libres, y no se pueden compactar en el momento.

Solución posible: mantener una lista de bloques libres.

Problema de esta solución: cuando se cree un archivo nuevo será necesario conocer su tamaño final para poder elegir el hueco apropiado.

2. **Asignación de lista enlazada**: cada archivo es una lista enlazada de bloques de disco. La primera palabra del primer bloque del archivo se usa como puntero al próximo bloque, hasta llegar a un puntero inválido (fin de archivo).

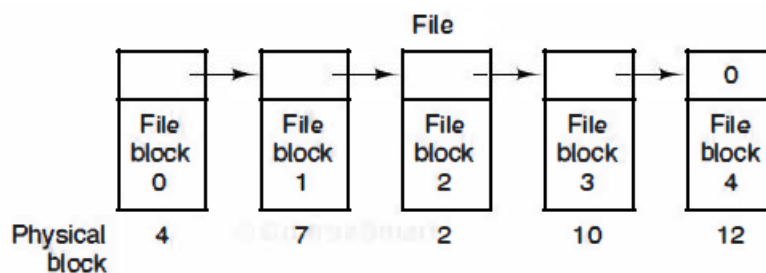


Figura 19: Asignación de lista enlazada.

Ventajas:

- a) **No se produce fragmentación**, porque cada bloque se puede aprovechar.
- b) La entrada del directorio se reduce a almacenar, para cada archivo, **la dirección de disco del primer bloque**.

Desventajas:

- a) La lectura aleatoria es **lenta**. Para llegar al bloque  $n$ , se necesita leer los  $n - 1$  bloques anteriores.
- b) El puntero ocupa mucho **espacio**.

3. **FAT / Asignación de lista enlazada utilizando una tabla en memoria:** misma estructura anterior, pero los punteros de cada bloque del disco se almacenan en una tabla en memoria. Esta tabla contiene tantas entradas como bloques de disco haya. Cada entrada posee un puntero al bloque siguiente del archivo.

Además se tiene un directorio que, para cada archivo, tiene el número del bloque inicial.

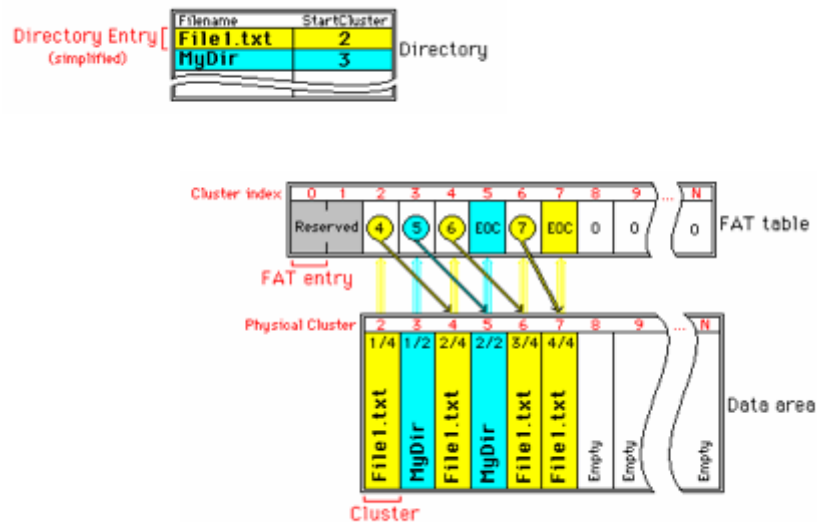


Figura 20: FAT.

Ventajas:

- El bloque completo contiene datos.
- La lectura aleatoria es mucho más rápida, porque la lista enlazada está en memoria.

Desventajas:

- Toda la tabla debe estar en la memoria todo el tiempo. Con un disco de 200 GB, se requiere una FAT que ocupa 600 MB en memoria.

4. **Nodos-i:** cada archivo y cada directorio tiene asociado una estructura de datos llamada **nodo-i**. Estos son registros de **longitud fija** con los siguientes campos<sup>3</sup>:

- Propietario y grupo del propietario,
- Tipo de archivo (regular, directorio, de caracteres, de red, de bloque)
- Permisos de acceso (9 bits),
- Fechas de acceso y modificación,
- Cantidad de registros del directorio que referencian al nodo-i (contador de *hardlinks*)
- Tamaño del archivo en bytes,
- Lista de bloques de almacenamiento del archivo:
  - Bloques 0 a 9 contienen direcciones directas
  - Bloque 10 contiene dirección indirecta simple

<sup>3</sup>El nombre del archivo NO es un campo del nodo-i. Se guarda en la entrada del directorio.



- 3) Bloque 11 contiene dirección indirecta doble
- 4) Bloque 12 contiene dirección indirecta triple

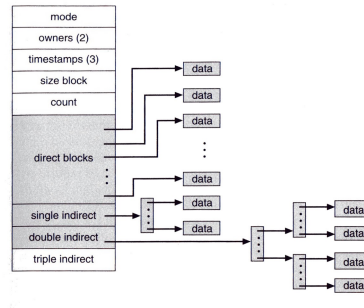


Figura 21: Nodo-i

Cada vez que un archivo cambia, cambia su correspondiente nodo-i. Pero si el contenido del nodo-i cambia, no necesariamente cambiará el archivo también.

La copia de un nodo-i a memoria contiene, además de los campos ya mencionados, los siguientes:

- Estado del nodo-i en memoria (“bloqueado/desbloqueado”, “proceso esperando por el nodo-i”, “la copia en memoria es distinta a la que está en el disco por un cambio en el nodo-i”, “la copia en memoria es distinta a la que está en el disco por un cambio en los datos del archivo”, “el archivo es un punto de montaje”),
- Número de dispositivo que contiene al archivo,
- Número de nodo-i,
- Punteros a otros nodos-i en memoria (el sistema encadena los nodos-i en memoria de igual forma que los buffers de bloques),
- Contador que indica el número de instancias del archivo que están activas.

Cuando este contador llega a 0, el nodo-i se libera y se lo agrega a la lista de nodos-i libres.

Ventajas:

- a) El nodo-i necesita estar en memoria **solo cuando está abierto el archivo**.

Desventajas:

- a) El número del nodo-i es simplemente un índice a un sector del disco que contiene al nodo-i en sí. Por lo tanto, hay una **cantidad fija de nodos-i posibles**.

## 9.5 Implementación de directorios

Implementación de archivos mediante...	Asignación contigua	Asignación de lista enlazada (en disco y en memoria)	Nodos.
Entrada de directorio debe contener...	Nombre de archivo + Dirección de disco del primer bloque + Cantidad de bloques que ocupa el archivo	Nombre de archivo + Dirección de disco del primer bloque	Nombre de archivo + <b>Número</b> del nodo-i del archivo. El directorio raíz suele ser el nodo-i 0.
Para leer un archivo...			<p>• To look up a pathname "/etc/passwd", start at root directory and walk down chain of inodes...</p> <pre> graph TD     Inode0[inode 0] --&gt; Table0[File table at inode 0]     Table0 --&gt; Inode2801[inode 2801]     Inode2801 --&gt; Table2801[File table at inode 2801]     Table2801 --&gt; Inode2859[inode 2859]     Inode2859 --&gt; FileData[File data for /etc/passwd]     </pre>

## 9.6 Ejemplos de sistemas de archivos

### 9.6.1 Linux: ext2

*Estructura del sistema de archivos:*

- Linux admite varias docenas de sistemas de archivos mediante la capa **Virtual File System** (VFS). VFS oculta de los procesos y las aplicaciones de nivel superior las diferencias entre muchos tipos de sistemas de archivos que Linux acepta. VFS define un conjunto de abstracciones básicas del sistema de archivos y las operaciones que se permiten en estas abstracciones.

VFS acepta sistemas de archivos con cuatro estructuras principales:

<i>Superbloque</i>	Contiene información sobre el sistema de archivos.
<i>Dentry</i>	Representa una entrada de directorio. Se crea mediante el sistema de archivos al instante.
<i>Nodo-i</i>	Cada nodo-i describe un archivo, directorio o dispositivo.
<i>Archivo</i>	Representación en memoria de un archivo abierto.

- Nombres de archivos de 255 caracteres.
- **Archivo:** consiste en una secuencia de 0 o más bytes. No se distingue entre archivos ASCII o binarios, o cualquier otro tipo.
- Es un sistema de archivos jerárquico.

*Sistema de archivos ext2:*

- Estructura:

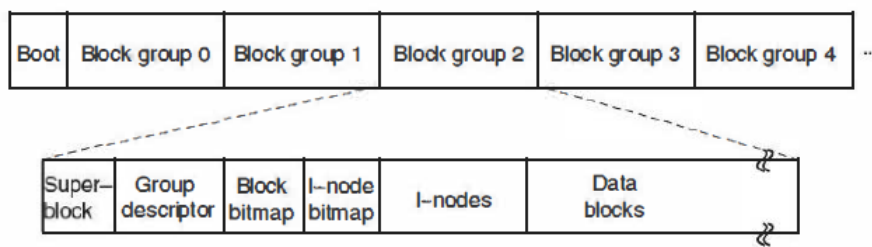


Figura 22: Sistema de archivos ext2

- *Superblock*: contiene la cantidad de nodos-i, la cantidad de bloques de disco, y el principio de la lista de bloques libres.
  - *Descriptor de grupo*: contiene la localización de los bitmaps, la cantidad de bloques y nodos-i libres, y la cantidad de directorios.
  - *Bitmap de bloques*, de 1 KB.
  - *Bitmap de nodos-i*, de 1 KB.
  - *Nodos-i*, de 128 bytes cada uno.
  - *Bloques de datos*.
- Una entrada de directorio contiene:
    - Número de nodo-i,
    - Tamaño de la entrada,
    - Tipo de entrada (archivo, directorio, dispositivo),
    - Longitud del nombre del archivo,
    - Nombre el archivo.
  - Cada proceso tiene su propia *file descriptor table* (tabla descriptora de procesos). El sistema tiene una única *open file descriptor table*.

#### Seguridad:

- Cada usuario tiene un **User ID (UID)**, un número entero desde 1 a 65.525. El usuario con UID 0 es el **root** (supervisor).
- Los usuarios se pueden organizar en grupos. Cada grupo tiene su propio **Group ID (GID)**.
- Cada proceso lleva el UID y el GID de su dueño.
- Los permisos de los **archivos** se heredan del proceso que los creó. Cada archivo tiene **9 bits de seguridad**: 3 bits **rwX** (*r=read*, *w=write*, *x=execute*) para el dueño, 3 bits *rwX* para el grupo del dueño, y 3 bits *rwX* para el resto.
- Los directorios son archivos, y por lo tanto también tienen 9 bits de seguridad, pero el bit *x* se refiere al permiso de búsqueda.

### 9.6.2 Linux: ext3

Es el mismo que ext2 pero incorpora la funcionalidad de *journaling*, por lo que ext3 es más lento que ext2.

### 9.6.3 UNIX: BSD (Berkeley Software Distribution)

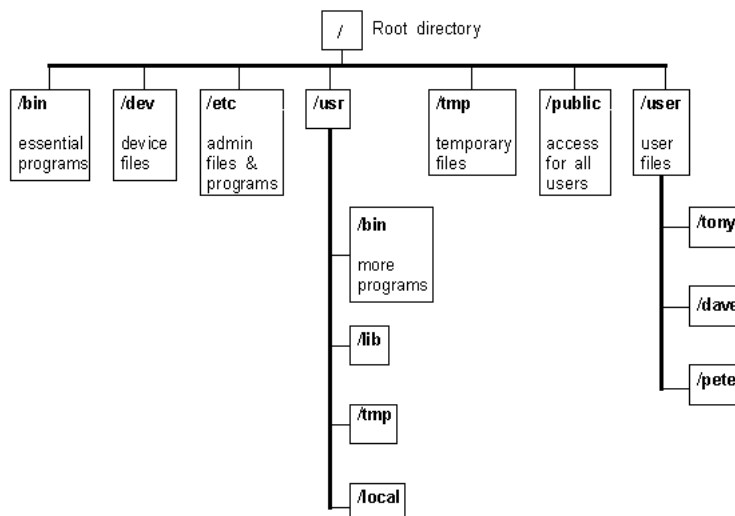


Figura 23: UNIX File System

*Estructura del sistema de archivos:*

- Los **registros de directorio** son de longitud variable. Cada uno contiene:
  - Nombre del archivo o subdirectorio,
  - Número de nodo-i del archivo o subdirectorio.
- El volumen se organiza en **bloques** (en general de 4 KB cada uno), pero para minimizar la fragmentación interna, el sistema maneja **fragmentos de bloque** que deben tener de tamaño submúltiplo del tamaño de bloque (por ejemplo, de 1 KB). Los fragmentos de bloque se utilizan para asignárselos a bloques finales de archivos. Entonces, un bloque puede tener fragmentos de distintos archivos, pero un archivo no puede tener sus fragmentos en distintos bloques.
- Un **volumen** se compone de varias secciones:
  1. **Boot block**: contiene el módulo de arranque del sistema operativo de esa partición.
  2. **Superblock**: contiene todos los parámetros clave acerca del sistema de archivos. Se encuentra replicado en distintas partes del sistema, para asegurar su integridad. Contiene:
    - a) Tamaño del sistema de archivos,
    - b) Tamaño de los bloques,
    - c) Bitmap para para control de fragmentos de bloque libres
    - d) Cantidad de nodos-i libres,
    - e) Lista de nodos-i libres,
    - f) Puntero al primer nodo-i libre,
    - g) Campos para bloquear los mapas de bloques y la lista de nodos-i libres,
    - h) Campo que indica que el superblock ha sido modificado.
  3. **Inode list**: lista de nodos-i del sistema de archivo. El tamaño de esta lista debe ser especificado por el administrador al configurar el sistema.
  4. **Data blocks**. Un block de datos puede pertenecer a uno y solo un archivo.

Los datos se dividen en áreas conformadas por cilindros adyacentes. Cada área tiene su superblock propio, para aumentar la seguridad. Cada grupo de cilindros tiene su propia lista de nodos-i. La política de asignación de espacio procura que los archivos queden localizados en una misma área. El sistema trata de ubicar los nodos-i de un directorio en una misma área.

- Cada proceso tiene su propia *user file descriptor table*. El sistema tiene una única *open file table*. Cuando un proceso abre o crea un archivo, el kernel devuelve un *file descriptor* que es un índice a la *user file descriptor table* de ese proceso.
- *User file descriptor table*: para cada archivo abierto por el proceso, contiene un puntero a la *file table* correspondiente a ese archivo.
- *Open file table*: estructura global del kernel, que para cada archivo abierto, contiene el desplazamiento en bytes en el archivo donde comenzará la próxima lectura o escritura.
- *Inode table*

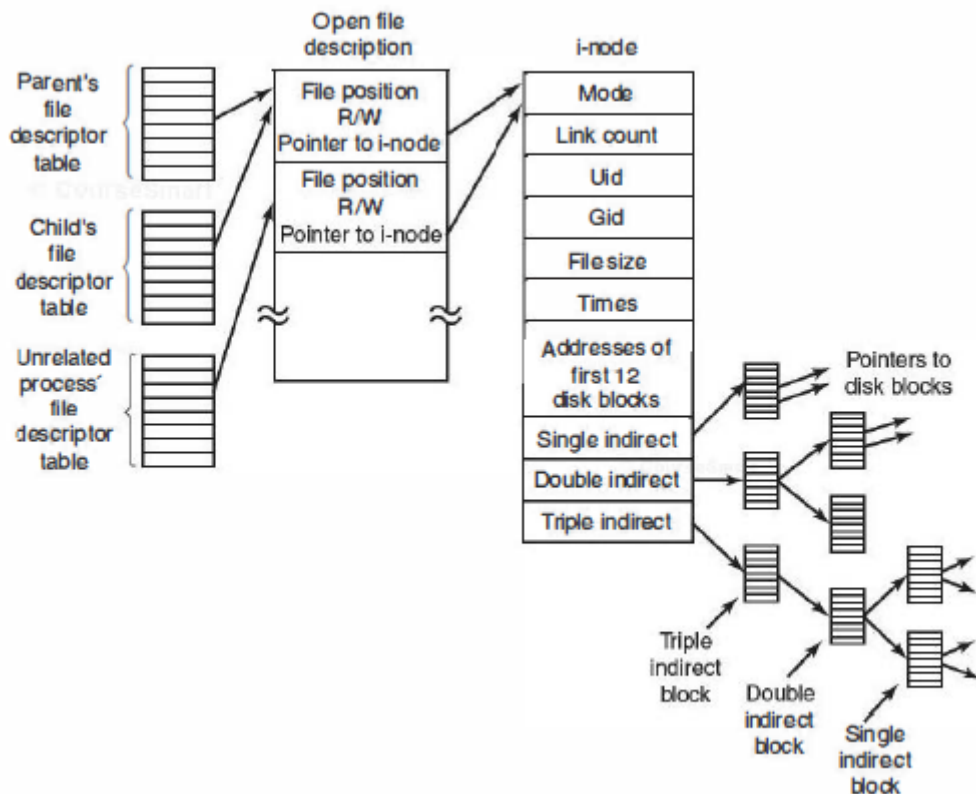


Figura 24: Relación entre las tablas

Cuando se llama a una *system call* que involucra un *file descriptor*, el *file descriptor* se utiliza como un índice a la *file descriptor table* para localizar el puntero a la *open file descriptor table* y a su vez al nodo-i correspondiente al archivo.

## 9.6.4 Microsoft: NTFS (New Technology File System)

*Estructura del sistema de archivos:*

- Es un sistema de archivos jerárquico que se desarrolló específicamente para la versión Windows NT.
- Utiliza direcciones de disco de 64 bits, y por lo tanto acepta particiones de hasta 16 exabytes.
- **Archivo:** consiste en varios atributos, y cada uno se representa mediante un flujo de bytes.

- Atributos residentes, se almacenan en la MTF
  - Atributos no residentes
- Hay soporte para *hard links*, pero sólo se pueden aplicar a archivos del mismo volumen dado que al registro del archivo en la MFT se le agrega otro registro. Además, si se le cambia el tamaño o los atributos a un archivo, las entradas de los demás links pueden no ser actualizadas hasta que se abran.
  - Hay soporte para *soft links*.
  - Cada volumen se organiza como una secuencia lineal de **clústeres** (conjuntos de **sectores**) de tamaño fijo (por lo general, 4 KB). Los **fragmentos** son conjuntos contiguos de clusters, y están especificados por:
    - *Virtual Cluster Number* (VCN): número de cluster relativo respecto del archivo.
    - *Logical Cluster Number* (LCN): número real de cluster lógico respecto del disco.
    - Cantidad de clusters contiguos a partir de LCN.
  - **Master File Table (MFT)**: principal estructura de datos de cada volumen.
    - Es un **archivo** extensible hasta  $2^{48}$  registros. Es una secuencia lineal de **registros** de 1 KB cada uno. Cada registro consiste en un **encabezado de registro** (número mágico para comprobar la validez del archivo, puntero al primer atributo, puntero al primer byte de espacio libre en el registro, número de registro base del archivo) más una secuencia de **pares** <encabezado de atributo (definición del atributo, longitud del valor), valor del atributo>.

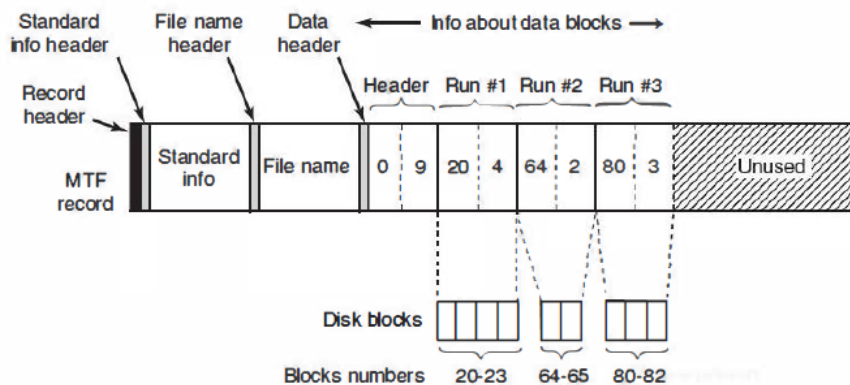


Figura 25: Ejemplo de registro de la MFT.

- Cada registro describe a un archivo o un directorio:
  - Atributos del archivo.

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in \$Extend\$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in \$Volume)
Volume information	Volume version (used only in \$Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to \$LogFile
Data	Stream data; may be repeated

Figura 26: Atributos de un registro de la MFT

- Lista de direcciones de disco donde se encuentran sus bloques.
- Los directorios chicos son registros con varias entradas de directorios, cada una de las cuales describe a un archivo o directorio. Los directorios grandes utilizan árboles B+ para listar archivos, facilitando la búsqueda alfabética y la inserción.
- Si el archivo ocupa muchos bloques en disco, el primer registro de la MFT (**registro base**) de ese archivo apunta a los otros registros de la MFT del mismo archivo.
- Como es un archivo, se lo puede colocar en cualquier parte del disco.
- Un **bitmap** lleva el registro de las entradas libres en la MFT.
- Los primeros 16 registros de la MFT se reservan para describir los archivos de metadatos de NTFS.

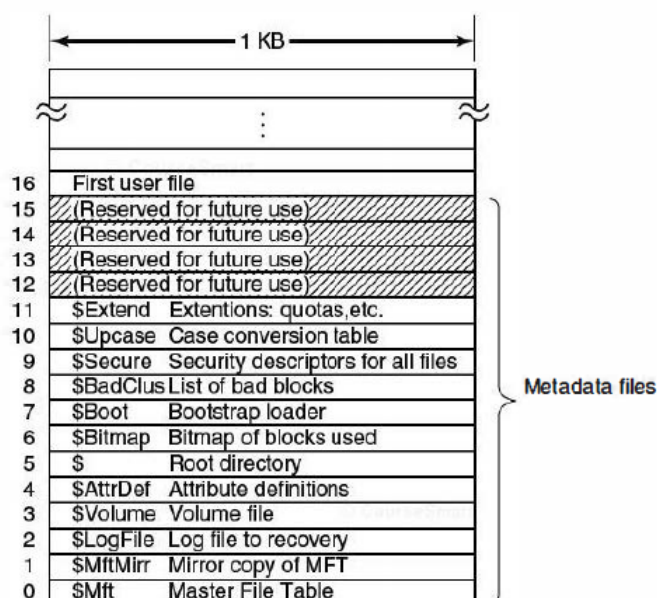


Figura 27: MFT de NTFS

Seguridad:

- NTFS utiliza un diario de cambios (*journal*): es el archivo de metadatos **\$LogFile**. Este archivo registra, cada 5 segundos, los cambios que se producen en todas las estructuras de disco, y si se completan con éxito, se elimina el registro correspondiente. Este archivo tiene dos tipos de registros:
  - UNDO: para deshacer acciones que no se pudieron completar.
  - REDO: para rehacer acciones que no se pudieron completar.
- Cada usuario y grupo se identifica mediante un **SID** (*Security ID*). Cada SID es único en el mundo. Cuando un usuario comienza un proceso, el proceso y sus hilos corren bajo el SID del usuario.
- Cada proceso tiene un **token de acceso**, que indica quién es el propietario del proceso y qué valores y poderes están asociados con él.
- Cada objeto tiene asociado en la MFT un **descriptor de seguridad**, que indica quién puede realizar operaciones sobre él.

## 9.7 Otros sistemas de archivo

**Transactional NTFS** a component of Windows Vista and later operating systems. It brings the concept of atomic transactions<sup>4</sup> to the NTFS file system, allowing Windows application developers to write file output routines that are guaranteed either to succeed completely or to fail completely.

Transactional NTFS allows for files and directories to be created, modified, renamed, and deleted atomically. Using transactions ensures correctness of operation; in a series of file operations (done as a transaction), the operation will be committed if all the operations succeed. In case of any failure, the entire operation will rollback and fail. Transactional NTFS is implemented on top of the Kernel Transaction Manager (KTM), which is a Windows kernel component first introduced in Windows Vista that provides transactioning of objects in the kernel.

A **log-structured filesystem** is a file system in which data and metadata are written sequentially to a circular buffer, called a log.

This has several important side effects:

1. Write throughput on optical and magnetic disks is improved because they can be batched into large sequential runs and costly seeks are kept to a minimum.
2. Writes create multiple, chronologically-advancing versions of both file data and meta-data. Some implementations make these old file versions nameable and accessible, a feature sometimes called time-travel or snapshotting. This is very similar to a versioning file system.
3. Recovery from crashes is simpler. Upon its next mount, the file system does not need to walk all its data structures to fix any inconsistencies, but can reconstruct its state from the last consistent point in the log.

A **versioning file system** is any computer file system which allows a computer file to exist in several versions at the same time. Thus it is a form of revision control. Most common versioning file systems keep a number of old copies of the file. Some limit the number of changes per minute or per hour to avoid storing large numbers of trivial changes. Others instead take periodic snapshots whose contents can be accessed with similar semantics to normal file access.

A **journaling file system** is a file system that keeps track of the changes that will be made in a journal (usually a circular log in a dedicated area of the file system) before committing them to the main file system. In the event of a system crash or power failure, such file systems are quicker to bring back online and less likely to become corrupted. After a crash, recovery simply involves reading the journal from the file system

---

<sup>4</sup>An example of atomicity is ordering an airline ticket where two actions are required: payment, and a seat reservation. The potential passenger must either: both pay for and reserve a seat; OR neither pay for nor reserve a seat. The booking system does not consider it acceptable for a customer to pay for a ticket without securing the seat, nor to reserve the seat without payment succeeding.



and replaying changes from this journal until the file system is consistent again. The changes are thus said to be atomic (not divisible) in that they either: succeed (succeeded originally or are replayed completely during recovery), or are not replayed at all (are skipped because they had not yet been completely written to the journal before the crash occurred).

A **virtual file system (VFS)** is an abstraction layer on top of a more concrete file system. The purpose of a VFS is to allow client applications to access different types of concrete file systems in a uniform way. A VFS can, for example, be used to access local and network storage devices transparently without the client application noticing the difference. It can be used to bridge the differences in Windows, Mac OS and Unix filesystems, so that applications can access files on local file systems of those types without having to know what type of file system they are accessing. A VFS specifies an interface (or a "contract") between the kernel and a concrete file system. Therefore, it is easy to add support for new file system types to the kernel simply by fulfilling the contract. The terms of the contract might change incompatibly from release to release, which would require that concrete file system support be recompiled.

## 9.8 Archivos mapeados a memoria

*Son un segmento de memoria virtual al que se le asignó una correlación directa byte-a-byte con alguna porción de un archivo o recurso similar a un archivo. Este recurso es, típicamente, un archivo que esta físicamente en el disco, pero también puede ser un dispositivo, objeto de memoria compartida u otro recurso que el sistema operativo puede referenciar mediante un descriptor. Una vez en memoria los archivos mapeados se ven como parte de la memoria.*

*The memory mapping process is handled by the virtual memory manager, which is the same subsystem responsible for dealing with the page file. Memory mapped files are loaded into memory one entire page at a time.*

Ventajas:

- *Aumenta la performance de I/O, especialmente con archivos grandes. Accessing memory mapped files is faster than using direct read and write operations for two reasons. Firstly, a system call is orders of magnitude slower than a simple change to a program's local memory. Secondly, in most operating systems the memory region mapped actually is the kernel's page cache (file cache), meaning that no copies need to be created in user space.*
- *Applications can access and update data in the file directly and in-place, as opposed to seeking from the start of the file or rewriting the entire edited contents to a temporary location. Since the memory-mapped file is handled internally in pages, linear file access requires disk access only when a new page boundary is crossed, and can write larger sections of the file to disk in a single operation.*

Desventajas:

- *The memory mapped approach has its cost in minor page faults - when a block of data is loaded in page cache, but is not yet mapped into the process's virtual memory space.*
- *A file larger than the addressable space can have only portions mapped at a time, complicating reading it.*

Aplicaciones:

- **Loading:** *When a process is started, the operating system uses a memory mapped file to bring the executable file, along with any loadable modules, into memory for execution. This permits the OS to selectively load only those portions of a process image that actually need to execute.*
- Permitir a varios procesos compartir la misma información en memoria.
- Acceder a archivos en disco rápidamente sin necesidad de usar *buffers*.

En C se hace utilizando la librería `sys/mman.h`, mediante las funciones `mmap` y `munmap`.

---

**Algoritmo 6** Lectura de archivos mapeados a memoria

---

```
1 int fd = open("archivo.txt", O_RDONLY, S_IRUSR);
2 int len = 1024; //100 registros de ints
3
4 void* addr = mmap(NULL, len, PROT_READ, MAP_SHARED, fd, 0);
5 close(fd); //no hace falta mas
6
7 int ar* = (int*) addr; // establezco direccionamiento
8
9 for (int i=0; i < 100; i++)
10 {
11     // imprimo el contenido de cada registro
12     cout<<"a["<<i<<"]= "<<ar[i]<<" ";
13     if (i%10 == 9)
14         cout<<endl;
15 }
16
17 munmap(addr,len); //lo sacamos de memoria. si no lo hago, exit() lo hace
```

---

---

**Algoritmo 7** Escribir en archivos mapeados a memoria

---

```
1 int fd = open("archivo.txt", O_RDWR|O_CREAT, S_IRUSR | S_IWUSR);
2 int len = 1024; //100 registros de ints
3
4 void* addr = mmap(NULL, len, PROT_WRITE, MAP_SHARED, fd, 0);
5 close(fd); //no hace falta mas
6
7 int ar* = (int*) addr; // establezco direccionamiento
8
9 for (int i=0; i < 100; i++)
10 {
11     ar[i] = 1000 + i; // en cada byte del archivo escribimos un valor
12 }
13
14 munmap(addr,len); //lo sacamos de memoria. si no lo hago, exit() lo hace
```

---

## 9.9 Clustered filesystems

**Clustered file system** sistema de archivo que es compartido por estar montado simultáneamente en múltiples servidores. Pueden proveer de servicios como *location-independent addressing* y redundancia, que mejoran la confiabilidad y reducen la complejidad de otras partes del cluster.

Proveen un mecanismo de control de concurrencia y de serialización.

**RAID (Redundant Array of Inexpensive Disks)** tecnología de almacenamiento que combina múltiples discos en una sola unidad lógica. Los datos son distribuidos a través de los discos en varias formas ("niveles RAID") dependiendo de la redundancia y *performance* requeridas, pero son vistos como un solo disco por el sistema operativo.

Provee funciones de:

- *Disk mirroring*: replicación de volúmenes de discos en discos duros separados para asegurar la disponibilidad continua.
- *Data striping*: técnica de segmentar datos secuencialmente lógicos para almacenarlos en dispositivos distintos y permitir el acceso concurrente.
- Corrección de errores

Se realiza de dos formas:

- **Software RAID / Fake RAID:** existe una capa adicional entre el *file system* y el *device driver*. El procesador debe estar ocupado haciendo las operaciones del RAID. El usuario puede ver que existen varios discos, y puede unirlos en un RAID.
- **Hardware RAID:** existe un dispositivo que maneja los discos físicos y los presenta a la computadora como una unidad lógica. El sistema operativo (y por lo tanto el usuario) solo ve un disco, que es este controlador haciéndose pasar por un disco. Provee de una interfaz front-end y back-end:
  - Front-end hacia el *host adapter* de la computadora
  - Back-end hacia los discos

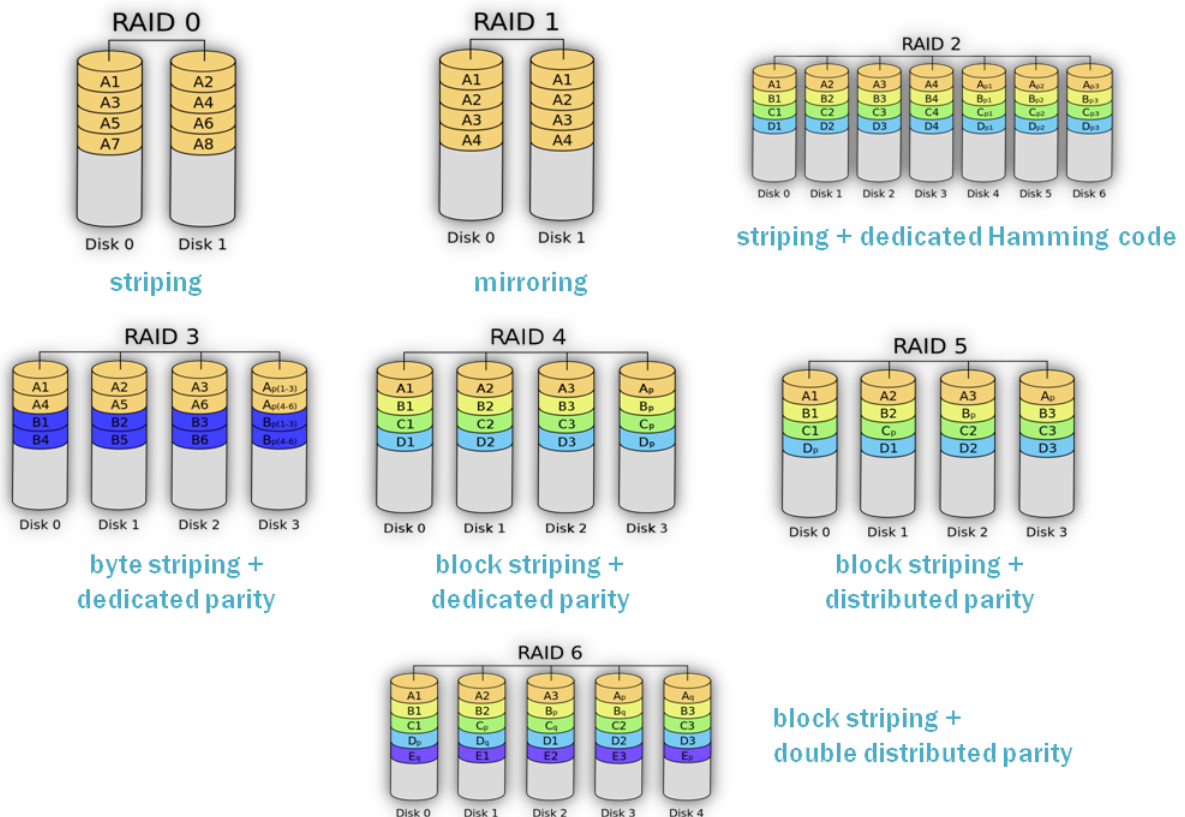


Figura 28: Niveles RAID

**Levels of nested RAID / hybrid RAID** combina dos o más niveles de RAID para ganar performance, redundancia, o ambos.

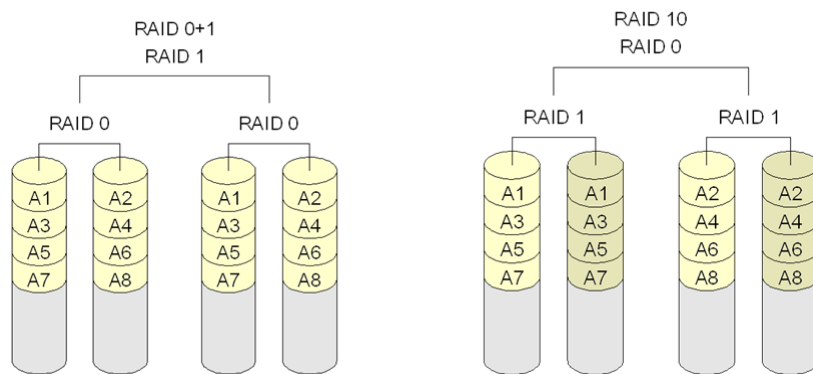


Figura 29: Nested RAID

También existen arquitecturas no RAID:

- **JBOD** (*Just a Bunch of Disks*): an array of drives, each of which can be accessed directly as an independent drive.
- **MAID** (*Massive Array of Idle Drives*): hundreds to thousands of hard drives for nearline storage. MAID is designed for "Write Once, Read Occasionally" applications.

## 9.10 Tipos de almacenamiento

*Storage Area Networks (SANs) and Network Attached Storage (NAS) complement each other very well to provide access to different types of data. SANs are optimized for high-volume block-oriented data transfers while NAS is designed to provide data access at the file level.*

**A NAS is a single storage device that operate on data files, while a SAN is a local network of multiple devices that operate on disk blocks.**

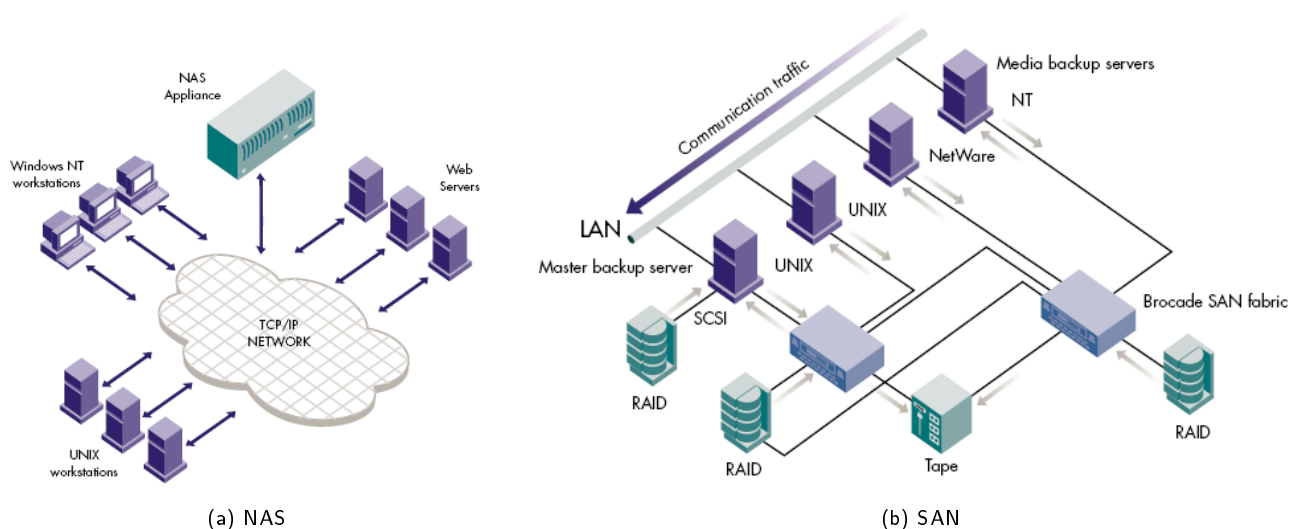


Figura 30: NAS vs SAN

**Network Attached Storage (NAS)** conecta un *File System* remoto a una red, proporcionando el acceso a clientes heterogéneos. Provee almacenamiento e implementa el software de *File System*.

**Storage Area Network (SAN)** conecta dispositivos remotos que el sistema operativo ve como locales, proporciona almacenamiento en bloques, y deja el *File System* a cargo del cliente.

Cualquiera de las dos arquitecturas soporta RAID.

	<b>SAN</b>	<b>NAS</b>
Protocolos	Fibre Channel Fibre Channel-to-SCSI	TCP/IP
Aplicaciones	<ul style="list-style-type: none"> <li>■ Procesamiento de bases de datos en transacciones</li> <li>■ Resguardo de información en forma centralizada</li> <li>■ Recupero ante desastres</li> <li>■ Consolidación de almacenamiento</li> </ul>	<ul style="list-style-type: none"> <li>■ Compartición de archivos</li> <li>■ Transferencia de datos sobre grandes distancias</li> <li>■ Acceso de solo lectura a bases de datos</li> </ul>
Ventajas	<ul style="list-style-type: none"> <li>■ Gran disponibilidad</li> <li>■ Confiabilidad de la transferencia de datos</li> <li>■ Tráfico reducido</li> <li>■ Configurable</li> <li>■ Administración centralizada</li> </ul>	<ul style="list-style-type: none"> <li>■ No hay limitación de distancias</li> <li>■ Es fácil agregar capacidad de almacenamiento</li> <li>■ Fácil de instalar y mantener</li> </ul>

*The administrator of a home or small business network can connect one NAS device to their LAN. The NAS maintains its own IP address comparable to computer and other TCP/IP devices. Using a software program that normally is provided together with the NAS hardware, a network administrator can set up automatic or manual backups and file copies between the NAS and all other connected devices. The NAS holds many gigabytes of data, up to a few terabytes. Administrators add more storage capacity to their network by installing additional NAS devices, although each NAS operates independently. Administrators of larger enterprise networks may require many terabytes of centralized file storage or very high-speed file transfer operations. Where installing an army of many NAS devices is not a practical option, administrators can instead install a single SAN containing a high-performance disk array to provide the needed scalability and performance. Administrators require specialized knowledge and training to configure and maintain SANs.*

## Parte II

## Parte Práctica

## 10 BASH

Command	Description	Optional parameters						
echo x	Print x	-n Do not output the trailing newline. -E Disable the interpretation of the following backslash-escaped characters -e Enable interpretation of the following backslash-escaped characters in each STRING: \a alert (bell) \b backspace \c suppress trailing newline \e escape \f form feed \n new line \r carriage return \t horizontal tab \v vertical tab \\ backslash						
ls	List files in directory	-l: with details -1: one line per file -a: include hidden files, "." and ".."						
chown owner file	Change file's ownership							
chgrp group file	Change file's group							
chmod mode file	Change's file permissions							
cp source destination	Copy source to destination							
more file	Show file on screen, paged. Move forward with spacebar	-d: show message "[Press space to continue, 'q' to quit.]"						
export variable	Set export attribute for shell variables							
cat file	Concatenate file(s), or standard input, to standard output.							
cut file	Print selected parts of lines from each file to standard output.	<table><tr><td>-b=LIST</td><td>select only these bytes</td></tr><tr><td>-c=LIST</td><td>select only these characters</td></tr><tr><td>-f=LIST</td><td>select only these fields</td></tr></table> <p>Use one, and only one of -b, -c or -f. Each LIST is made up of one range, or many ranges separated by commas. Selected input is written in the same order that it is read, and is written exactly once. Each range is one of: N N'th byte, character or field, counted from 1 N- from N'th byte, character or field, to end of line N-M from N'th to M'th (included) byte, character or field -M from first to M'th (included) byte, character or field</p>	-b=LIST	select only these bytes	-c=LIST	select only these characters	-f=LIST	select only these fields
-b=LIST	select only these bytes							
-c=LIST	select only these characters							
-f=LIST	select only these fields							
read name	Read a line from standard input and assign it to <i>name</i>	<ul style="list-style-type: none"><li>■ -d delim: use <i>delim</i> to terminate the input line</li><li>■ -r: backslash doesn't act as an escape character</li><li>■ -s: characters written are not echoed</li></ul>						

Cuadro 7: Commands

Environment Variable	Description
SHELL	Name of shell
PWD	Print the full filename of the current working directory.
PS1	Prompt 1
PATH	Path where to find executables
RANDOM	A random integer between 0 and 32767 is generated. Assigning a value to this variable seeds the random number generator.
USER	Get current user

Cuadro 8: Environment variables

### Algoritmo 8 Arrays

```

1 #!/bin/bash
2 # An entire array can be assigned by enclosing the array items in parenthesis:
3 NAMESERVERS=("ns1.nixcraft.net." "ns2.nixcraft.net." "ns3.nixcraft.net.")
4
5 # Individual items can be assigned with the familiar array syntax
6 NAMESERVERS[4]=Hello
7 NAMESERVERS[5]=World
8
9 # Get length of an array
10 tLen=${#NAMESERVERS[@]} # or tLen=${#NAMESERVERS[*]}
11
12 # Use for loop read all nameservers
13 for (( i=0; i<${tLen}; i++ ));
14 do
15     # It gets a bit ugly when you want to refer to an array item
16     echo ${NAMESERVERS[$i]}
17 done
18
19
20 ${NAMESERVERS[*]}    # All of the items in the array
21 ${!NAMESERVERS[*]}  # All of the indexes in the array

```

Inside script parameters	
\$0	Script name
\$i	Positional parameter <i>i</i> ( <i>i</i> = 1.,9)
@	Parameter list (except \$0)
#	Length of parameter list (except \$0)

File Test Operators	
-f	File exists?
-d	Directory exists?

### Algoritmo 9 Read from file

```

1 while read -r line # -r prevents backslash interpretation, use it always
2 do
3     # ... process line ...
4 done < $filename

```

---

**Algoritmo 10** For loop syntax

---

```
1 # Numeric ranges for syntax is as follows:
2 for i in 1 2 3 4 5
3 do
4     echo "Welcome_${i}_times"
5 done
6
7 for i in {1..5}
8 do
9     echo "Welcome_${i}_times"
10 done
11
12 for OUTPUT in $(Linux-Or-Unix-Command-Here)
13 do
14     # command1 on $OUTPUT
15 done
16
17 for (( i=1; i<=5; i++ ))
18 do
19     echo "Welcome_${i}_times"
20 done
```

---

---

**Algoritmo 11** Arrays en BASH

---

```
1 array=(Hello World) #contains two elements
2 echo ${array[0]} #prints "Hello", the braces are required to avoid conflicts with pathname expansion
3
4 ${array[*]} #all the items
5 ${#array[*]} #number of items
```

---



**regex** is a language for describing patterns in strings.  
**grep** filters its input against a pattern.  
**sed** applies transformation rules to each line.

In code, regular expressions describe matchable patterns over text.

They are often used to describe locations in text (e.g. all lines that match this pattern) and to transform text (e.g. transform text matching a pattern into something different text).

There is no standard for regular expressions in code, but most languages employ a dialect from a common ancestor.

## 11 GREP

The **grep** command provides a variety of ways to find strings of text in a file or stream of output.

There are two ways to provide input to **grep**, each with its own particular uses.

1. **grep** can be used to search a given file or files on a system.
2. **grep** also can be used to send output from another command that **grep** will then search for the desired content.

Although it is usually possible to integrate **grep** into manipulating text or doing “search and replace” operations, it is not the most efficient way to get the job done. Instead, the **sed** and **awk** programs are more useful for these kinds of functions.

There are two basic ways to search with **grep**:

1. searching for fixed strings and
2. searching for patterns of text. To search for text with variable content, use regular expressions.

Regular expressions are included in the **grep** command in the following format:

```
grep [options] [regexp] [filename]
```

Regular expressions are comprised of two types of characters:

1. normal text characters, called *literals*, and
2. special characters, such as the asterisk (\*), called *metacharacters*. An *escape sequence* allows you to use metacharacters as literals or to identify special characters or conditions (such as word boundaries or “tab characters”).

The desired string that someone hopes to find is a *target string*.

A *regular expression* is the particular search pattern that is entered to find a particular target string.

It is customary to place the regular expression inside single quotation marks. There are a few reasons for this.

1. The first is that normally Unix shells interpret the space as an end of argument and the start of a new one. What if the string you wish to search for has a “space” character? The quotes tell **grep** (or another Unix command) where the argument starts and stops when spaces or other special characters are involved.
2. The other reason is that various types of quotes can signify different things with shell commands such as **grep**.

For instance, using the **single quote** underneath the tilde key (also called the backtick) tells the shell to execute everything inside those quotes as a command and then use that as the string. For instance:

```
grep `whoami` filename
```

would run the `whoami` command (which returns the username that is running the shell on Unix systems) and then use that string to search.

**Double quotes**, however, work the same as the single quotes, but with one important difference. With double quotes, it becomes possible to use environment variables as part of a search pattern:

```
grep "$HOME" filename
```

## 11.1 Metacharacters

Since metacharacters help define the manipulation, it is important to be familiar with them.

Metacharacter	Name	Matches	Example
<b>Items to match a single character</b>			
.		Any one character	'r.d' would match "red", "rod", "red", "rzd", and so on
[...]	Character class	Any character listed in brackets. There are two basic ways to use character classes: to specify a range and to specify a list of characters. A combination of ranges can also be used.	'[a-f]' '[aeiou]' '[a-zA-F0-5]'
[^...]	Negated character class	Any character NOT listed in brackets	
\char	Escape character	The character after the slash literally; used when you want to search for a "special" character	'.' would match any single character and would return every piece of text in a file. '\.' would only match the actual "period" character.
<b>Items that match a position</b>			
^	Caret	Start of line	'^red' would match all lines that begin with "red"
\$		End of line. It will match every line in a stream except the final line, which is terminated by an "end of file" character instead of an "end of line" character.	'-\$' would find all lines whose last character is a dash
\<		Start of a word. It detects the beginning of a word by looking for a space or another "separation" that indicates the beginning of a new word (a period, comma, etc.).	'\<un' would match words starting with the prefix "un", such as "unimaginable," "undetected," or "undervalued." It would not match words such as "funding," "blunder," or "sun."
\>		End of a word. After the characters, it looks for a "separation" character that indicates the end of a word (a space, tab, period, comma, etc.).	'ing\>' would match words that end in "ing" (e.g., "spring"), not words that simply contain "ing" (e.g., "kingdom").
<b>Quantifiers</b>			
*		Any number (including 0); sometimes used as general wildcard	'install.*file' should output all the lines that contain "install" (with any amount of text in between) and then "file".
\?		The preceding character (or string if placed after a subpattern) is an "optional" matching pattern.	'colors\?' would match both "color" and "colors".
\+		1 or more of the preceding expression	'150\+' would match 150 with any number of additional zeroes (e.g., 1500, 15000, 1500000, etc.).
\{N\}		Match exactly N times. When placed after a character, indicate a specific number of repetitions to search for.	'150{3}\b' would match 15 followed by 3 zeroes. If the desired match is precisely "15000" and there is not a check for a word boundary "150000", "150002345" or "15000asdf" would match
\{N,\}		Match at least N times	
\{N,M\}		Match between N and M times	'150{2,3}\b' would match "1500" and "15000" and nothing else

For example, suppose you are looking for amounts that contain the dollar sign within *price.list*:

```
grep '[1-9]$" price.list
```

As a result, the search will try to match the numbers at the end of the line. This is certainly something you do not want. By using the escape character, annotated by the backslash (\), you avoid such confusion:

```
grep '[1-9]$" price.list
```

The metacharacter \$ becomes a literal, and therefore is searched in *price.list* as a string.

## 11.2 Rules

**grep** has rules of precedence for processing. Repetition is processed before concatenation. Concatenation is processed before alternation. Strings are concatenated by simply being next to each other inside the regular expression—there is no special character to signify concatenation.

For instance, take the following regular expression:

```
'pat{2}ern|red'
```

In this example, the repetition is processed first, yielding two “t”s. Then, the strings are concatenated, producing “pattern” on one side of the pipe and “red” on the other. Next, the alternation is processed, creating a regular expression that will search for “pattern” or “red”.

```
'(pat){2}ern|red'
```

```
'pat{2}(ern|red)'
```

The first example will concatenate “pat” first and then repeat it twice, yielding “patpatern” and “red” as the search strings. The second example will process the alternation subpattern first, so the regular expression will search for “pattern” and “pattred”.

A regular expression can continue as long as the single quote is not closed. For instance:

```
$ grep 'patt  
> ern' filename
```

In this case, the regular expression searches for the word “pattern”.

The following regular expression:

```
'username:''whoami'' and home directory is '$HOME'
```

would match on the string “username:bambenek and home directory is /home/bambenek”

## 11.3 Basics

There are two ways to employ **grep**. The first examines files as follows:

```
grep regexp filename
```

**grep** searches for the designated **regexp** in the given file (**filename**).

The second method of employing **grep** is when it examines “standard input.” For example:

```
cat filename | grep regexp
```

The **cat** command will display the contents of a file. The output of this command is “piped” into the **grep** command, which will then display only those lines that contain the given **regexp**.

It is important to note that “chaining” **grep** commands is inefficient most of the time. Often, a regular expression can be crafted to combine several conditions into a single search.

Example	Explanation
<code>grep -e -style doc.txt</code>	grep will look for lines that match “-style”.
<code>grep -f pattern.txt searchhere.txt</code>	grep searches for all the patterns from pattern.txt in the designated file searchhere.txt. The patterns are additive; that is, grep returns every line that matches any pattern. The pattern file must list one pattern per line.
<code>grep -v oranges filename</code>	The output would be every line in filename that does not contain the pattern “oranges”.
<code>grep -c contact.html access.log</code>	Prints just a count of how many lines matched in each input file.
<code>grep -l “ERROR:” *.log</code>	Prints just the names of input files containing the pattern. The search stops on the first match.
<code>grep -m 10 ‘ERROR:’ *.log</code>	This option tells grep to stop reading a file after 10 lines are matched.
<code>grep -b pattern filename</code>	Displays the byte offset of each matching text.
<code>grep -H pattern filename</code>	Includes the name of the file before each line printed.
<code>grep -n pattern filename</code>	Includes the line number of each line displayed
<code>grep -a pattern filename</code>	Allows a binary file to be processed as if it were a text file.

One final limitation of basic grep: the “extended” regular expressions metacharacters—?, +, {, }, |, (, and )—do not work with basic grep. The functions provided by those characters exist if you preface them with an escape.

## 11.4 Backreferences

Imagine the following text file:

```
The red dog fetches the green ball.
The green dog fetches the blue ball.
The blue dog fetches the blue ball.
```

Only the third line repeats the use of the same color. A regular expression pattern of “(red|green|blue)\*(red|green|blue)” would return all three lines. To overcome this problem, you could use backreferences:

```
grep -E '(red|green|blue).*\1' filename
```

This command matches only the third line, as intended.

It is important to note that backreferences require the use of parentheses to determine reference numbers.

## 11.5 Performance

For most routine uses, grep performance is not an issue. Even megabyte-long files can be searched quickly using any of the specific grep programs without any noticeable performance difference. Obviously, the larger the file, the longer the search takes.

The more “choices” given to grep, the longer a particular search takes. For instance:

```
grep -E '(0|2|4|6|8)' filename
grep -E '[02468]' filename
```

Comparing the two examples, the second one performs better because no alternation is used and so lines do not have to be searched multiple times. Avoid alternation when other alternatives exist that accomplish the same thing.

By far, the biggest cause of performance slowdowns when using grep is the use of backreferences.

## 11.6 Examples

### ■ IP addresses

```
$ grep -E '\b[0-9]{1,3}(\.[0-9]{1,3}){3}\b' patterns
```

- This will also find strings that aren't valid IP addresses.

```
$ grep -E '\b
((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b' patterns
```

- In this case, it makes sure to find IP addresses with an octet between 0–255 by establishing a combination of patterns that would work.

### ■ MAC addresses

```
$ grep -Ei '\b[0-9a-f]{2}
(:[0-9a-f]{2}){5}\b' patterns
```

- In this case, the additional -i option is added so no regard is given to capitalization.

### ■ Email addresses

```
$ grep -Ei '\b[a-z0-9]{1,}@*\.
(com|net|org|uk|mil|gov|edu)\b' patterns
```

## 12 sed

**sed** reads the **STDIN** into the pattern space, performs a sequence of editing commands on the pattern space, then writes the pattern space to **STDOUT**. If no flags are specified the first match on the line is replaced.

Most Unix utilities work on files, reading a line at a time. Sed, by default, is the same way.

**sed** only operates on patterns found in the incoming data. That is, the input line is read, and when a pattern is matched, the modified output is generated, and the **rest** of the input line is scanned.

The **sed** regular expressions are essentially the same as the **grep** regular expressions. They are summarized below.

<code>^</code>	Matches the beginning of the line
<code>\$</code>	Matches the end of the line
<code>.</code>	Matches any single character
<code>(character)*</code>	Match arbitrarily many occurrences of (character)
<code>(character)\?</code>	Match 0 or 1 instance of (character)
<code>(character)\+</code>	Match 1 or more instances of (character)
<code>[abcdef]</code>	Match any character enclosed in <code>[]</code> (in this instance, <code>a b c d e</code> or <code>f</code> ). Ranges of characters such as <code>[a-z]</code> are permitted
<code>[^abcdef]</code>	Match any character NOT enclosed in <code>[]</code> (in this instance, any character other than <code>a b c d e</code> or <code>f</code> )
<code>(character)\{m,n\}</code>	Match <code>m-n</code> repetitions of (character)
<code>(character)\{m,\}</code>	Match <code>m</code> or more repetitions of (character)
<code>(character)\{,n\}</code>	Match <code>n</code> or less (possibly 0) repetitions of (character)
<code>(character)\{n\}</code>	Match exactly <code>n</code> repetitions of (character)
<code>\(expression\)</code>	Group operator
<code>\n</code>	Backreference - matches <code>nth</code> group
<code>expression1\ expression2</code>	Matches <code>expression1</code> or <code>expression2</code>

Cuadro 10: sed regular expressions

The forward slash `/` is a special character in sed.

## 12.1 Substitution

The substitution command, denoted by `S`, will substitute any string that you specify with any other string that you specify.

`sed "s/pattern/replacement text/{flags}" inputfile`

The flags can be any of the following:

<code>n</code>	replace <code>nth</code> instance of pattern with replacement
<code>g</code>	replace all instances of pattern with replacement
<code>p</code>	write pattern to <code>STDOUT</code> if a successful substitution takes place
<code>w file</code>	write pattern to <code>file</code> if a successful substitution takes place

For example, to substitute instances of the regular expression `[ch]at` for `ball`, use:

```
$ sed 's/[ch]at/ball/g' < in > out
```

## 13 PERL

### 13.1 Data types

Type	Sigil	Description
Array	@	Indexable list of scalar values.
Code	&	A piece of Perl code, e.g., a subroutine.
Format		A format for producing reports.
Glob	*	All data types.
Hash	%	Associative array of scalar values.
IO		Filehandle. Used in input and output operations.
Scalar	\$	Strings, numbers, typeglobs, and references.

Cuadro 11: Data types

- Header: `#!/bin/perl`
- Comments must begin with `#`
- Instructions must end with `;`
- `$inputline = <STDIN>;`
- Command-line arguments are stored in the array `@ARGV`.
- Variable declaration:
  - Scalar: `$my_scalar`
  - Array: `@my_array` (must contain scalars)
    - Index: `$my_array[0] ... $my_array[$#my_array]`
    - Length: `$length = @my_array;`
    - Add element: `push(@my_array, $my_value);`
    - Remove element: `$element = pop(@my_array);`
  - Hash structure: `%my_hash = (user => maine, password => 1234);`
    - Add element : `$my_hash{"key"} = "value"`
    - Access element: `$my_array{key}`
    - Delete element: `delete($my_hash{key})`
    - Exists element: `exists($my_hash{key})`
    - Values: `@values = values(%my_hash)`
    - Keys: `@keys = keys(%my_hash)`

Scalar variables operators	
+	Addition
-	Substraction
*	Multiplication
/	Division
**	Power
%	Remainder
.	Concatenation
x	Repetition

- `chomp`: removes the EOL of a string
- `chop`: removes the last character of a string



Comparison operators	
Numeric	String
>	gt
>=	ge
<	lt
<=	le
==	eq
!=	ne

## 13.2 Control structures

```

1 if (boolean expression 1) {
2 }
3 elsif (boolean expression 2) {
4 }
5 else {
6 }
7
8 while (boolean expression) {
9 }
10
11 for ($i = 0; $i < $n; $i++) {
12     next; #like "continue" in C
13 }
14
15 foreach $variable (@array) {
16     last; #like "break" in C
17 }

```

## 13.3 I/O with files

```

1 open ($file_handle, "<_$_in_file");
2
3 while ($line = <$file_handle>) {
4     chomp($line);
5     # ... process line ...
6 }
7
8 close($file_handle);

```

## 13.4 Files with fields

```

1 # Get fields
2 $info = "Lou,_Reed,_71";
3 @data = split(",", $info);
4 ($name,$surname,$age) = split(",", $info);
5
6 # Join fields
7 $string = join ("", @numbers);

```

## 13.5 I/O with directories

```

1 $dir_name = $ENV{PWD};
2 if (opendir($dir_handle, $dir_name)) {
3     @files = readdir($dir_handle);
4     closedir($dir_handle);
5 }

```

```

6
7 foreach $file_name (@files) {
8     if ($file_name eq "." || $file_name eq "..") {
9         next;
10    }
11    # ... process file ...
12 }
13
14 foreach $number ($from .. $to) {
15
16 }

```

File Test Operators	
-e	File or directory exists?
-d	Directory exists?
-f	Is file?
-l	Is symbolic link?
-r	Is file readable?
-w	Is file writable?
-x	Is file executable?
-z	File has zero size?
-O	Is owned by user?

## 13.6 Subroutines

- `my $variable`: will only be used in subroutine.
- `local $variable`: will only be used in subroutine and any subroutine called by it.

```

1 # Call
2 &printnum ($number1, $number2, $number3);
3
4 sub printnum {
5     my ($number1, $number2, $number3) = @_;
6     #or... $number1 = $_[0]; etc
7     my ($total);
8     $total = $number1 + $number2 + $number3;
9     print ("Total:_$total\n");
10 }

```

## 13.7 Extras

```

1 # Get time as an array
2 ($sec,$min,$hour,$mday,$month,$year,$yday,$yday, $isdst) = localtime;
3
4 # Get time as string
5 $date = localtime;
6
7 # Get user
8 $user = getlogin();
9
10 # System calls
11 $scripts = 'ls -l *.sh';
12
13 # Comparison regexp
14 $line = "html_php_javascript";
15 if ($line =~ /^html/) {
16     print "$line_starts_with_html";
17 }
18

```

```
19 # Substitution regexp
20 $string = "Today_is_Monday.";
21 $string =~ s/Monday/Friday/;
```

Special Variables	
\$_	Last read record
\$.	Current line number for the last file handle accessed
\$0	Name of script
\$\$	PID
@ARGV	Parameters passed (doesn't include \$0)
%ENV	Environment variables