

# **Introducción a los sistemas operativos**

Isabel Salcedo de Vicente



# índice

1. Introducción
2. Gestión de procesos
3. Seguridad y protección
4. Sistemas de ficheros
5. Administración de memoria
6. Gestión de entrada/salida

Martes 10 Septiembre 2024

# introducción

## EVALUACIÓN:

### teoría

2 opciones  
↑

- Examen 4 primeros temas (60% de la nota teoría, semana del 22 nov) Será tipo test al completo. En algunas habrá que razonar.
- Examen de los 2 últimos temas (40% de la nota de teoría, puntuación mínima 4 pts) 18 de diciembre por la tarde.

### práctica

- Examen de guiones shell, 50% de la nota de prácticas, 25 octubre.
- Examen de administración, 50%, 13 de diciembre, nota mínima 4.

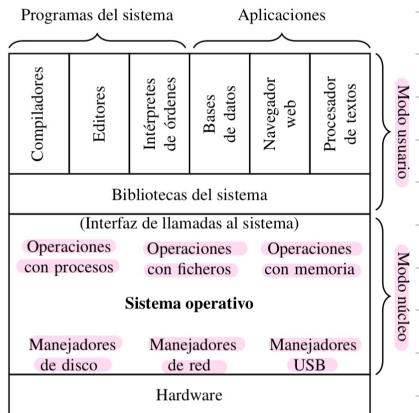
# INTRODUCCIÓN

## Tema 1

### 1. Concepto

Un sistema operativo es un componente software que se encuentra justo sobre el hardware y por debajo de los programas del sistema y aplicaciones. Es un intermediario entre los usuarios y el hardware. Son muy complejos, pero, de manera general, los podemos ver desde dos puntos de vista:

- Como máquina virtual: el SO es un programa que oculta el funcionamiento del hardware al programador y presenta una interfaz sencilla y amigable.
- Como controlador de recursos: controla todas las piezas de un sistema complejo y proporciona una asignación ordenada y controlada de los procesadores, memoria, dispositivos de E/S, etc.



### 2. Protección del SO

Se consigue a través del hardware y 3 mecanismos:

#### modos de ejecución

El procesador dispone de dos modos:

- Modo núcleo: el procesador puede ejecutar cualquier instrucción, en este modo se ejecuta el SO.
- Modo usuario: es el que usan los programas de usuario, no están disponibles todas las instrucciones.

#### protección memoria

El hardware debe tener algún mecanismo que impida que un programa acceda a memoria RAM que no le pertenece.

Si no se protege la RAM, un programa puede acceder al vector de interrupciones.\*

#### interrupciones

Para que un programa no se haga con el control de la CPU es necesario permitir que el SO tome el control de la máquina cada cierto tiempo, y así, asignar la CPU a otro programa.

\* contiene para cada interrupción la dirección del código para tratarla.

LA PROTECCIÓN  
SOLO FUNCIONARÁ  
SI EXISTEN ESTOS  
TRES MECANISMOS

### 3. Historia y evolución

#### PRIMERA GENERACIÓN (1945-1955): VÁLVULAS Y CONEXIONES

Grandes máquinas construidas con válvulas de vacío y cables. Consumían mucha energía y tenían muy poca fiabilidad. Un solo grupo de personas diseñaba, construía, programaba y mantenía cada máquina.

Los programas se ejecutaban instrucción a instrucción usando tarjetas perforadas (representaban ceros y unos). El código debía incluir código para controlar los dispositivos de E/S.

Había muchos períodos en los que la máquina no se utilizaba, lo cual era desfavorable, pues era cara y había que aprovecharla al máximo.

#### SEGUNDA GENERACIÓN (1955-1965): TRANSISTORES Y SISTEMAS POR LOTES

Surgieron los lectores de tarjetas, impresoras de líneas, cintas magnéticas y de software: ensambladores, cargadores y enlazadores. Gracias al transistor los ordenadores se volvieron fiables. Se dividió al personal de mantenimiento, diseño, programadores... etc.

Surgen también las bibliotecas de E/S, ya no se necesitaba incluir código de E/S.

Se contrató a un operador profesional (sistema operativo humano) que era el intermediario entre la máquina y el programador (se pierde la interacción directa con la máquina).

El operador es el que prepara la máquina, mete los trabajos y transmite los resultados.

Con esto se evitaba perder tiempo.

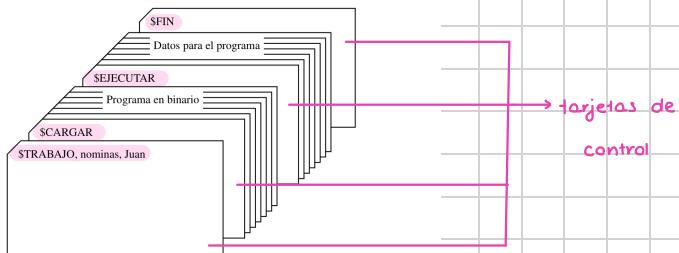
Otra medida fue ir compilando los programas uno tras otro (se escribían en FORTRAN).

Sin embargo, cuando se detenía un trabajo, el operador debía determinar por qué y realizar un volcado de memoria si fuera necesario; luego volver a preparar el ordenador.

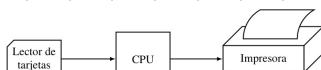
Durante la transición de un programa a otro la CPU quedaba inactiva. Para evitar esto, se desarrolló el sistema de procesamiento por lotes. Así, se crean los primeros sistemas operativos rudimentarios como el monitor residente. Este sistema interpretaba las tarjetas de control insertadas entre las tarjetas de datos.

##### Elementos del monitor residente.

- Interpretador de tarjetas.
- Cargador de programas.
- Manejadores de E/S.



Otro concepto de esta generación fue la **operación fuera de línea**: la CPU interactúa con dispositivos rápidos de E/S como cintas directamente, e indirectamente con los dispositivos lentos (impresoras, lectores de tarjetas, etc.).



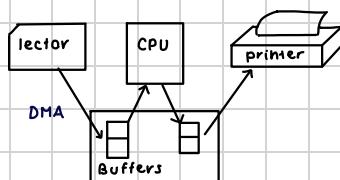
(a) Operación en línea.

La operación fuera de línea es útil cuando hay varias máquinas que desarrollan los trabajos de la 1<sup>a</sup> y 3<sup>a</sup> etapa. De lo contrario, la máquina principal puede quedar ociosa.

Este avance dio lugar a la **independencia de dispositivo de E/S**, es decir, al programa le da igual el tipo de dispositivo de E/S con el que está interactuando. El SO hace corresponder los "dispositivos lógicos" (almacén de tarjetas) con los físicos (lector de tarjetas, cintas...).

Además, se consiguió que la E/S y la CPU funcionasen en paralelo, pero al ser su coste bastante elevado, se buscó conseguir un parallelismo parecido en una sola máquina.

Un **buffer** es una zona de memoria con almacenamiento temporal.



Cuando la CPU está lista para usar datos, comienza la siguiente lectura.

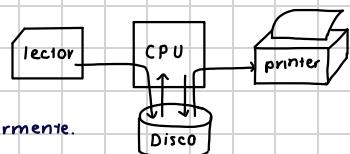
Conseguimos que la E/S de un programa se realice de forma simultánea con el cómputo del MISMO programa.

Es útil cuando los promedios de procesamiento de datos de la CPU y el dispositivo de E/S son parecidos. El manejo de buffers es una función del SO.

El mecanismo de **spoolers** (simultaneous peripheral operation on-line) permite la **superposición** de la E/S de unos programas con el cómputo de otro.

Se debe de disponer de acceso aleatorio al almacenamiento.

La CPU lee datos y los almacena para ser procesados posteriormente.



## TERCERA GENERACIÓN (1965-1980): CIRCUITOS INTEGRADOS Y MULTIPROGRAMACIÓN

Gracias a los circuitos integrados, los ordenadores se vuelven más pequeños y fiables.

Encontramos 2 conceptos importantes en esta época:

La **multiprogramación** consiste en tener varios programas en memoria. Cuando el programa que usa la CPU no pueda continuar (por ejemplo porque espera una operación de E/S) se le dará la CPU a otro. Para esto, debe existir un planificador de programas.

El **tiempo compartido o multitarea** consiste en realizar un cambio rápido entre programas lo que permite que cada usuario interactue con el programa que está ejecutando y tener la sensación de que él es el único usuario. (pseudoparalelismo)

Será necesario disponer de mecanismos que obliguen a un programa a dejar la CPU tras cierto tiempo.

Se recupera la interactividad.

Aparecen sistemas operativos como: IBM 360, MULTICS, OS/360.

## CUARTA GENERACIÓN (1980-1995): ORDENADORES PERSONALES

Aparecen los circuitos LSI, VLSI (millones de transistores en un chip). Aparece el ordenador personal (PC). Se populariza la informática.

Sistemas operativos: MS-DOS, Unix, OS/2, Windows 95/98, etc.

Además surgen nuevos tipos de SO: los de red y los distribuidos, y de tiempo real.

## QUINTA GENERACIÓN (1995-NOW): INTERNET

Además del Internet, surge el cloud computing. Se proliferan los smartphones, el primero siendo el Nokia N9000, aunque se popularizaron por el iPhone (2007).

SOS: Windows 2000, Windows XP, Linux, macOS, Android, iOS, etc.

## 4. Tipos de sistemas operativos

### DE PROPÓSITO GENERAL

Destacan por su variedad para realizar tareas y flexibilidad. En los **supercomputadores** el SO más usado es Unix. Los supercomputadores son sistemas formados por un **gran nº de**

elementos de todo tipo conectados entre sí mediante una red.

Unix también es el SO más común en los **mainframes**; que son capaces de procesar **trabajos** que requieren gran cantidad de **E/S**: por lotes o batch, transacciones y multitarea.

En los **servidores** encontramos tanto Unix, como Windows, como macOS. Los servidores sirven a muchos usuarios a la vez a través de una red para **compartir recursos** (hw y sw).

Estos 3 SO's los encontramos también en **ordenadores personales** y teléfonos.

## DE RED

Los SO's mencionados anteriormente también son de red. Los usuarios son conscientes de la existencia de varios ordenadores conectados entre sí mediante una red. Cada máquina es independiente, **ejecuta su SO local**, pero gracias a la red, puede interactuar con las máquinas.

## DISTRIBUIDOS

Se ofrece a los usuarios una imagen única del sistema. El usuario no es consciente de donde están sus ficheros ni donde se ejecutan sus programas. A esto se le conoce como **transparencia de localización**. Razones para su existencia:

- Compartición de recursos.
- Aceleración de cálculos.
- Tolerancia a fallos: una máquina falla → el resto sigue.

Ejemplos: Amoeba, Plan 9, etc.

## DE TIEMPO REAL

El parámetro clave son las **restricciones temporales**. Los hay rigurosos (estrictos) y no rigurosos (aceptable no cumplir un plazo de vez en cuando, si el comportamiento general se ajusta a las necesidades). Ejemplos: VxWorks y QNX.

## PARA TARJETAS INTELIGENTES

Tienen grandes limitaciones de potencia y memoria. Disponen de una máquina virtual Java (JVM) que ejecuta applets que se cargan en la tarjeta. En algunos casos, es posible cargar y ejecutar varios applets a la vez.

## 5. Componentes y servicios

Para construir el entorno de un SO este se divide en pequeños módulos y definimos una interfaz.

Es posible que diversos SO tengan una interfaz idéntica aunque su estructura interna sea distinta. Vamos a hablar de diversos conceptos a través de 3 puntos de vista:

### COMPONENTES

- Administración de procesos. t2
- Administración de la memoria principal. t5
- Administración de la E/S. t6
- Administración de ficheros. t4
- Sistema de protección t3

### SERVICIOS

- Ejecución de programas
- Operaciones de E/S, ya que los programas de usuario no los pueden llevar a cabo.
- Manipulación del sistema de ficheros.
- Comunicaciones: un proceso puede necesitar intercambiar info con otro. Existen la memoria compartida (direcciones dentro de un proceso donde otro proceso puede leer o escribir) y tuberías.
- Detección de errores.

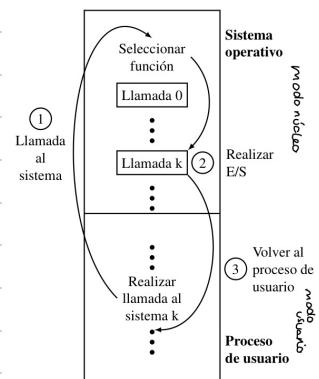
Además, existen otras funciones que aseguran un funcionamiento eficiente del sistema, como son:

- Asignación de recursos.
- Contabilidad: recopilar estadísticas, etc.
- Protección.

### LLAMADA AL SISTEMA

Estas llamadas son generalmente, instrucciones en ensamblador.

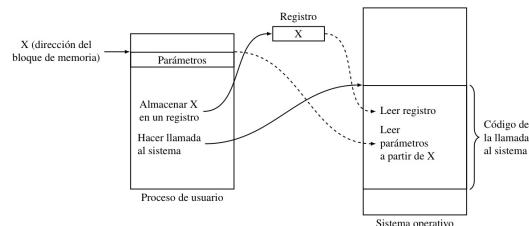
- ① Se ejecuta la llamada, el procesador pasa a modo núcleo y el SO analiza los parámetros de la llamada.
- ② Se realiza el procedimiento correspondiente a esa llamada.
- ③ El control regresa al proceso de usuario.



Podríamos decir que un SO no es más que un conjunto de llamadas al sistema.

El paso de parámetros al SO durante una llamada se puede realizar de 3 formas:

- En **registros** (puede haber + parámetros que registros)
- En un **bloque de memoria**.
- En la **pila**.



## PROGRAMAS DEL SISTEMA

- Manipulación de ficheros.
- Información de estado.
- Apoyo a lenguajes de programación.
- Comunicaciones.
- Programas de aplicación: hojas de cálculo, procesadores de texto, etc.

Desde el punto del SO no existe diferencia entre los **programas de usuario** y los **del sistema**.

Quizás, el programa más importante es el **intérprete de órdenes** (shell).

## 6. Estructura

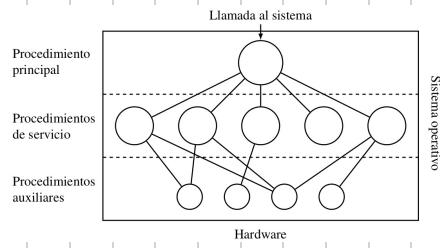
### SISTEMAS MONOLÍTICOS

Es la organización más común. Está caracterizada por:

- Colección de procedimientos que se llaman unos a otros.
- Cada procedimiento tiene una interfaz muy clara.
- No tiene una estructura bien definida.
- No existe ocultación, todo procedimiento es visible a los demás.

Ejemplos: Unix, Windows...\*

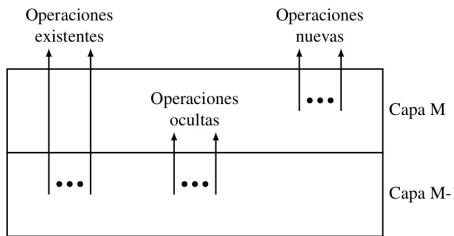
\*también podemos decir que tiene **estructura modular**



## SISTEMAS CON CAPAS

Capa más baja  $\rightarrow$  0, capa más alta  $\rightarrow$  N (interfaz con el usuario). La principal ventaja es la modularidad:

- Cada capa solo utiliza las funciones de la capa inferior.
- Una vez se ha implementado y depurado una capa, se pasa a la siguiente.
- Cada capa oculta los detalles de bajo nivel a las capas superiores.



Capa	Descripción
5	Programas de usuario
4	Administración de la E/S
3	Consola del operador
2	Administración de memoria
1	Planificación de la CPU
0	Hardware

### ESTRUCTURA DE THE

El mayor problema son las dependencias de capas que pueden surgir. Se opta por tener pocas capas.

## MODELO CLIENTE-SERVIDOR

Mover parte del SO a capas superiores y mantener un núcleo mínimo (microkernel).

Creadores  $\rightarrow$  solicitan servicios

Servidores  $\rightarrow$  realizan los trabajos solicitados y devuelven respuestas.

Un proceso servidor puede ser cliente si solicita a otro servidor.

Los servicios se ejecutan en modo usuario. Si hay un fallo en, p.ej. el sistema de ficheros solo hay que reiniciar el proceso servidor.

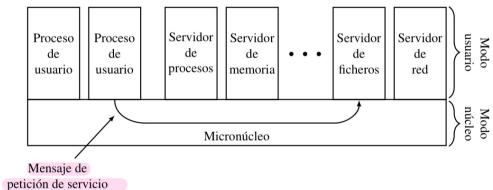
Se adapta muy bien a los sistemas distribuidos.

¿Qué pasa cuando queremos hacer una operación privilegiada?

a) incluimos servidor en el núcleo (mismo núcleo de antes): (

b) mensajes especiales enviados desde servidores al micronúcleo; este verificará que tiene permiso y realizará la operación.

Ejemplos: QNX, Minix3 o macOS.



## CONCEPTO Y PROTECCIÓN

un sistema operativo es un intermediario entre el hardware y los usuarios. Se puede ver como:

- ① **Una máquina virtual**: que oculta la complejidad del hardware mostrando una interfaz sencilla y amigable.
- ② **Controlador de recursos**: asigna ordenadamente procesadores, memoria, dispositivos de E/S y media conflictos.

Para proteger del resto de programas. modos de ejecución → **núcleo** + protección de memoria + interrupciones periódicas

## historia

### 1º GEN ('45-'55)

Válvulas de vacío  
Lenguaje máquina  
Desarrollo LENTO

### 2º GEN ('55-'65)

Transistores  $\text{t}_\text{p}$ , bibliotecas E/S, Monitor residente, operaciones fuera de línea, INDEPENDENCIA EN.



no S.O's

Operador profesional (S.O humano)

### 3º GEN ('65-'80)

Circuitos integrados  
Multiprogramación  
↓ planifica  
Tiempo compartido

IBM 360 MULTICS

### 4º GEN ('80-'95)

LSI y VLSI  
millones de transistores  
PC's for everyone!!

- \$

MS-DOS UNIX Windows95

### 5º GEN (now)



smartphones

## COMPONENTES Y SERVICIOS

Componentes

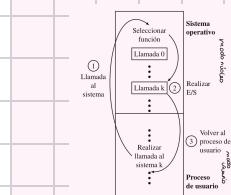
- de procesos
- Administración → de memoria
- + protección → de E/S
- de Ficheros

Servicios

- ejecución de programas
- operaciones E/S
- manipulación ficheros
- comunicaciones
- detección errores
- asignación recursos
- contabilidad
- protección

para eficiencia

## Llamadas al sistema



## Programas del sistema

Manipulación de ficheros, información de estado, apoyo a lenguajes de programación, comunicaciones, programas de aplicación

# TEMA 1 INTRODUCCION

## TIPOS DE SO'S

### Propósito general

Flexibles, adaptables.

Supercomputadores y mainframes : Unix

Servidores, PC's, teléfonos... : Windows, macOs, Unix.

### De red

Los usuarios son conscientes de que hay varias ordenadores conectados mediante red.

### Distribuidos

Igual a lo anterior pero el usuario tiene una imagen única del sistema, sirven por ejemplo para acelerar cálculos.

### De tiempo real

El parámetro clave son las restricciones temporales: hay rigurosas y no rigurosas.

### Para tarjetas inteligentes

Disponen de una máquina virtual

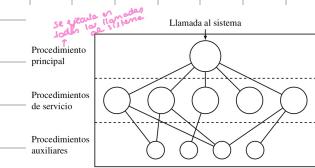
JAVA para ejecutar applets.

Limitaciones de potencia y memoria.

## ESTRUCTURA

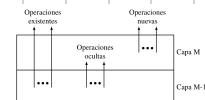
### 1 Monolíticos

Una colección de procedimientos que se llaman unas a otras, no tiene estructura bien definida y no ocultación.



### Por capas

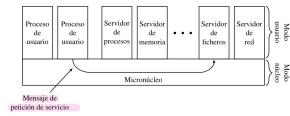
Las capas utilizan la de la capa anterior



### 2 Cliente - Servidor

Micronúcleo.

Un fallo en un sistema no afecta a todo el sistema.



# GESTIÓN DE PROCESOS

## Tema 2

### 1. introducción

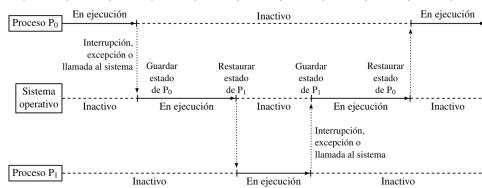
Cada CPU es capaz de ejecutar solo un proceso a la vez, por tanto, para dar la sensación de paralelismo al usuario, se ha desarrollado un **modelo de procesos** en el cual el software ejecutable del ordenador se organiza en torno al concepto de proceso.

#### CONCEPTO

Es un programa en ejecución. Un programa es algo estático, es el contenido de un fichero; mientras que un proceso es dinámico (en cada instante tiene un estado).

Un proceso ocupa espacio en disco. Un proceso requiere de más recursos. memoria (para código y pila), CPU y espacio en disco.

Un proceso puede constar de varios programas y viceversa.



La figura muestra un **pseudoparalelismo** entre los procesos. Hay paralelismo real cuando hay varias CPU's o se tienen varios núcleos. Aun así sigue existiendo tanto el pseudo como el paralelismo en los sistemas actuales.

La ejecución concurrente tiene las siguientes ventajas: compartición de recursos (físicos o lógicos), aceleración de cálculos, modularidad y comodidad.

Nota: Los procesos NUNCA deben programarse con hipótesis implícitas de tiempo.

#### CAMBIO DE PROCESO, CONTEXTO Y MODO

- **Cambio de proceso:** pasar de ejecutar de un proceso a otro.
- **Cambio de contexto:** cada vez que se guarda el contexto. Un cambio de proceso requiere cambios de contexto pero un cambio de contexto no siempre requiere un cambio de proceso.
- **Cambio de modo:** implica un cambio de contexto, pero no de proceso.

Sin embargo, un cambio de contexto no implica cambio de modo (hilos en modo usuario, cambio de un hilo a otro, sección 4).

## CREACIÓN Y DESTRUCCIÓN

Para crear un proceso en Unix se usa la llamada al sistema `fork()`, que crea una copia idéntica del proceso que llama (mismo código, pila...).

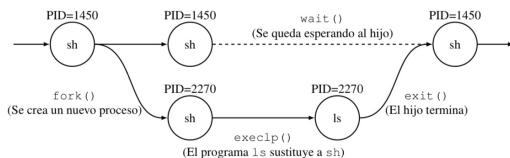
Se diferencian en que el valor devuelto por la función en el hijo es 0, y en el padre es el PID (Process Identifier) del nuevo proceso.

Al ser idénticas padre e hijo, para que cada uno haga una cosa distinta primero mirarán el valor devuelto por `fork()` para saber si son padre o hijo. Después se puede, por ejemplo, ejecutar la llamada al sistema `execve()` u otra de la familia `exec()` que sustituye las datos y código de un proceso por otro (además el usará una pila nueva). Una llamada a esta función nunca regresa si tiene éxito, y no es obligatorio que siempre la realice un proceso hijo. Además, esta ~~mutación de código~~ se realiza dentro del proceso, `exec()` no crea uno nuevo.

Lo que sí se conserva a pesar de la llamada:

- Los ficheros abiertos (aunque hay forma de cerrar ciertos descriptoros).
- El tratamiento de las señales.
- El PID y otras propiedades.

En Windows estas dos llamadas se hacen con una sola: `CreateProcess()`. A diferencia de Unix, en Windows un proceso padre puede transferirle la propiedad de su hijo a otro proceso.



Un proceso puede terminar de forma:

- Voluntaria: ha terminado la tarea o ha encontrado un error. `exit()` Unix      `ExitProcess()` Windows
- Involuntaria: es finalizado por el SO debido a un error fatal (división por cero, acceso incorrecto a memoria...), o finalizado por otro proceso. `Kill()` Unix      `TerminateProcess()` Windows

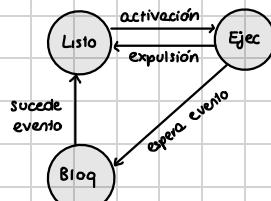
Siendo usuario, no se pueden matar procesos del root. La llamada `Kill()` no solo sirve para matar procesos, si no también para enviar señales.

El proceso `systemd` es el que tiene el PID 1 y es la raíz de todos los procesos, por ello puede realizar acciones especiales que otros no pueden.

## 2. estados de un proceso

Si simplificamos mucho, un proceso puede estar en tres estados:

- En ejecución.
- Listo: a la espera de recibir la CPU.
- Bloqueado: se encuentra a la espera de que suceda un evento externo (que le lleguen datos, que un hijo termine...)



La mayoría de procesos en un sistema están bloqueados

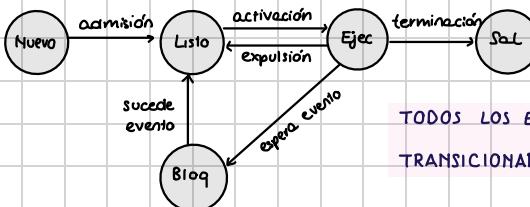
### Transiciones

De en ejecución a bloqueado → un proceso no puede continuar.

De Listo a en ejecución y viceversa → se debe al planificador. Se expulsa cuando excede su tiempo de uso, o hay un proceso importante que se debe ejecutar. Y lo mismo al revés, siempre debe haber un proceso en ejecución.

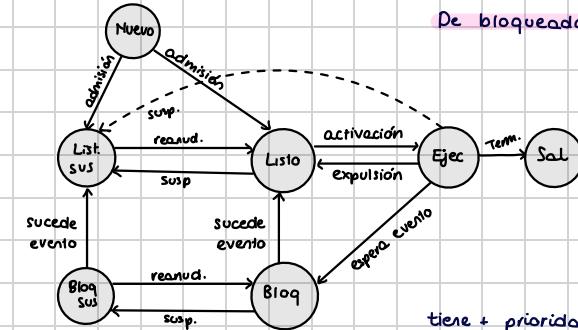
De bloqueado a Listo → el proceso ya dispone de lo que necesitaba.

Añadimos los estados de "Nuevo" (se acaba de crear un proceso) y "Saliente" (antes de que desaparece).



TODOS LOS ESTADOS PUEDEN TRANSICIONAR A SALIENTE !!

Deberemos añadir la situación en la que un proceso es "suspendido" o "reanudado" ya que muchas veces un programa se suspende para depurarlo, para liberar memoria o cuando el sistema está funcionando mal



De bloqueado a bloq.sus → para liberar memoria.

De bloq.sus a listo sus. → sucede el evento que necesita.

De listo sus. a listo → no quedan procesos listos, hay memoria libre o tiene + prioridad que uno listo.

De listo a listo sus. → cuando un proceso bloqueado tiene + prioridad que uno listo y hay que reducir memoria.

### Transiciones

De nuevo a lista suspendido → no hay memoria.

De bloq.sus. a bloq → no suele ocurrir, para ello debe tener muy alta prioridad.

De en ejecución a lista sus. → un proceso de prioridad alta despierta de bloq.sus. y no hay memoria suficiente.

### 3. implementación de procesos

Existe una tabla de procesos con una entrada por cada proceso (PCB, Process Control Block).

El PCB guarda todo aquello que sea necesario para que un proceso continúe su ejecución tras haber recuperado la CPU.

Administración de procesos	Administración de memoria	Administración de ficheros
Registros	Dirección del segmento de texto	Directorio raíz
Contador del programa	Dirección del segmento de datos	Directorio de trabajo
Palabra de estado del programa	Dirección del segmento BSS	Descriptores de fichero
Puntero de pila	Dirección del segmento de pila	Identificación de usuario
Estado del proceso		Identificación de grupo
Prioridad		
Parámetros de planificación		
Identificador de proceso		
Proceso padre		
Hora de inicio del proceso		
Tiempo utilizado de CPU		

### CREACION DE UN PROCESO

Estos son los pasos que se siguen:

1. El proceso padre realiza la llamada fork(), se salta al núcleo.

2. EL núcleo busca una entrada en la tabla de procesos y le asigna un PID.

→ no es el índice del PCB

3. Se copia toda la información del padre en la tabla del hijo excepto PID, segmentos, datos, pila...

4. Se asigna memoria para los segmentos de datos, pila del hijo y se copia el segmento del padre en ellos. Un proceso NUNCA debe modificar su código en memoria. El segmento de CÓDIGO es COMPARTIDO entre padre e hijo.

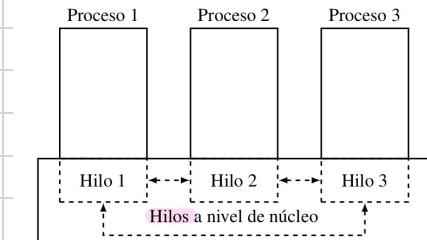
5. Se incrementan los contadores de los ficheros abiertos por el padre ya que también son abiertos por el hijo.

6. Se asigna al proceso hijo el estado "listo". Se devuelve el PID del hijo al padre, y el valor 0 al hijo.

## ESTRUCTURA DE UN PROCESO

En Unix, cuando un proceso realiza una llamada al sistema, se considera que ese mismo proceso pasa de modo usuario a modo núcleo; por tanto, cada proceso tiene dos partes: la de usuario (fuera del SO) y la de núcleo (dentro del SO).

La parte del núcleo es común para todos los procesos, pues hay una ÚNICA COPIA del SO en memoria.



Cada proceso tiene su propia pila y contador de programa dentro del núcleo, para que si hay varios en ejecución no interfieran entre sí.

Si un proceso se bloquea dentro del núcleo, cuando vuelva a tener la CPU continuará su ejecución dentro del núcleo.

## CAMBIO DE PROCESO

Los pasos para el cambio de proceso varían en función del hardware y el sistema operativo, aquí veremos una simplificación:

1 Al producirse la llamada al sistema el hardware almacena el contador de programa del proceso que la realiza en la pila de este (la que utiliza en modo usuario).

2 El hardware pasa a modo núcleo y carga el nuevo contador de programa desde el vector de interrupciones.



3 Ejecuta un procedimiento en lenguaje ensamblador guarda el "contexto" (registros y otros datos) en el PCB del proceso que realiza la llamada. Se pueden actualizar datos del PCB (tiempo de uso, estado...).

4 Un procedimiento en lenguaje ensamblador configura la nueva pila que va a utilizar el NÚCLEO. Cada proceso tiene una pila para el modo núcleo. (En sistemas actuales los procesos usan memoria virtual en modo usuario pero no en modo núcleo).

(muchas SO's están escritas en C)

5) Tras comprobar que la llamada al sistema existe, el procedimiento en ensamblador llama al procedimiento en C que lo implementa. Si durante la llamada el proceso se bloquea su estado debe cambiar.

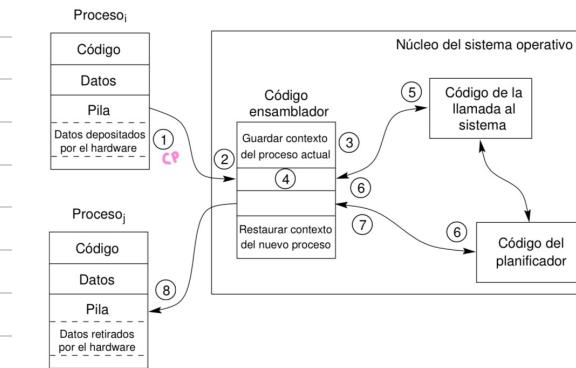
escrita en C

6) Se vuelve a un procedimiento en ensamblador que consulta si hace falta el planificador (si no se ha invocado antes) y lo ejecuta; este decide que proceso de las "listas" se ejecuta mirando su PCB. Después se vuelve al procedimiento en ensamblador.

7) Se restaura el contexto del proceso a ejecutar, prepara la pila que usará en modo usuario incluyendo el contador de programa del nuevo proceso.

La parte del SO que ENTREGA la CPU al proceso se llama DESPACHADOR.

8) Se cambia a modo usuario y se regresa de la llamada. Se desapila la dirección de la siguiente instrucción a ejecutar.

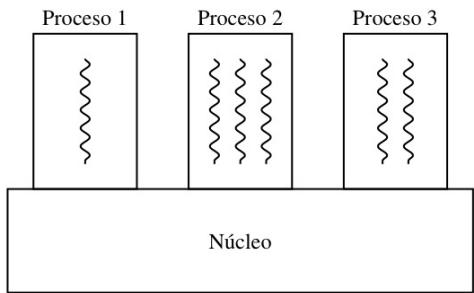


## 4. hilos

Hemos visto que el concepto de proceso engloba 2 características:

- **Unidad de propiedad de recursos:** porción de memoria que se le asigna, ficheros..
- **Unidad de planificación y ejecución:** entidad planificada y ejecutada por el SO.

Estas características son independientes, el SO las puede tratar por separado. Hablaremos de hilos refiriéndonos a la unidad de planificación y ejecución; y **proceso**, cuando hablamos de unidad de propiedad de recursos. Varios hilos existen dentro de un proceso.



Al inicio de un proceso, se ejecuta el hilo principal. Este hilo ejecuta la función main(), la cual crea, a su vez, un hilo hijo con pthread\_create(), uno de sus parámetros es el nombre de otra función del programa.  
gettid() → TID (thread identifier)  
getpid() → PID, mismo para hilos del = proceso

## ELEMENTOS

### Proceso

- Zona de memoria o espacio de direcciones
- Acceso protegido al procesador, otros procesos, ficheros y recursos E/S
- Variables globales, procesos hijos, alarmas, señales...
- Información contable

### Hilo

- Estado
- Contexto (PC register and others)
- Pila de ejecución
- Almacenamiento variables locales
- Acceso a memoria y recursos (compartido)

Los hilos no son tan independientes como los procesos. NO EXISTE PROTECCIÓN ENTRE HILOS, porque es imposible y no debe ser necesaria (el programador es el que decide sobre la creación de hilos).

El SO proporciona distintos mecanismos de sincronización entre hilos como mutex y semáforos.

## APLICACIONES

Razones para su utilización:

- Se pueden comunicar entre ellos rápidamente sin la intervención del núcleo.
- Se pueden bloquear; permiten el solapamiento de la E/S y el cómputo del proceso (unos hilos usan la CPU, otros la E/S).
- Es más rápida la creación y terminación de hilos, ya que no hay que liberar recursos. Se puede hacer un cambio de contexto entre dos hilos sin ir al núcleo.
- Se aprovechan mejor los cores y las CPU's si hay varias.
- Facilitan la construcción de programas

En un procesador de texto, un hilo puede activarse cada minuto para escribir en disco las modificaciones que hay en RAM y así evitar pérdidas.

## HILOS EN MODO USUARIO Y EN MODO NÚCLEO

• aunque se podrían programar llamadas no bloqueantes

### Implementación en modo usuario

#### ventajas

- El S.O. no sabe que existen, se pueden usar en núcleos que no los implementan.
- Los cambios de contexto son muy rápidos.
- Cada proceso puede tener su algoritmo de planificación para los hilos.

#### inconvenientes

- Cuando se bloquea un proceso se bloquean todos los hilos. NO SE PUEDE SOLAPAR E/S CON CÓMPUTO.
- Un fallo de página bloquea todo el proceso.
- La CPU tiene que repartirse entre los hilos.
- Si hay varias CPU no se obtiene paralelismo real porque la CPU se reparte entre procesos no hilos.

### la habitual

#### inconvenientes

- Las funciones para sincronizar hilos son más costosas. Se podrían implementar **funciones de biblioteca** para la sincronización y así solo se llama al SO cuando es necesario.
- La creación y destrucción de hilos es más costosa al tener que ir al núcleo. Una posible solución es reutilizarlos o sus estructuras de datos.

### Implementación en modo núcleo

#### ventajas

- El núcleo mantiene la tabla de hilos y reparte la CPU entre ellos. Hay paralelismo si hay varias CPU.
- No se bloquea todo el proceso al bloquear un hilo.
- Los fallos de página no suponen un problema.

Algunas acciones como la suspensión (para liberar memoria) o terminación de procesos afecta a todos los hilos.

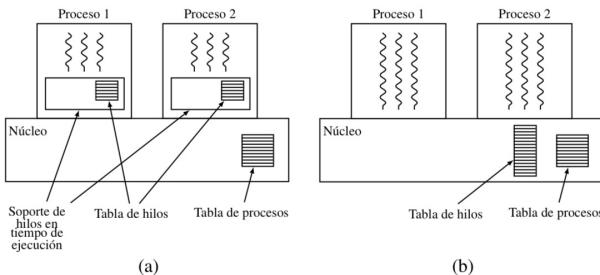


Figura 2.12. (a) Hilos implementados en espacio de usuario. (b) Hilos implementados en espacio del núcleo.

## 5. planificación de procesos

### METAS DE LA PLANIFICACIÓN

1. **Equidad:** cada proceso debe tener su proporción "justa" de CPU. Su tiempo asignado debe de ser inversa o directamente proporcional a una característica del mismo.
2. **Eficiencia:** mantener la CPU ocupada al 100% realizando trabajo útil (procesos de usuario).

$$E = \frac{T_{util}}{T_{util} + T_{carrera} + T_{gestion}}$$

3. **Tiempo de espera:** minimizar el tiempo desde que un proceso entra en la cola hasta que se le concede la CPU.
4. **Tiempo de respuesta:** el tiempo entre que se solicita algo y llegan los resultados (interactividad).
5. **Tiempo de regreso:** el tiempo entre el que se entrega un trabajo para que sea procesado y sus resultados. (Procesamiento de trabajos por lotes)
6. **Rendimiento o productividad.**

Las metas 4 y 5 son **contradicitorias**: para minimizar el tiempo de respuesta el planificador no debería de ejecutar los procesos de lotes salvo de madrugada (cuando los usuarios duermen). Es por eso que **no hay un algoritmo óptimo para todas las metas**.

### PLANIFICACIÓN APROPIATIVA Y NO APROPIATIVA

En la planificación **apropiativa** el planificador puede expulsar al proceso que esté usando la CPU para poder ejecutar otros.

En la planificación **no apropiativa** se deja que se termine o se bloquee el proceso que está usando la CPU.

Procesos interactivos → SIEMPRE apropiativa

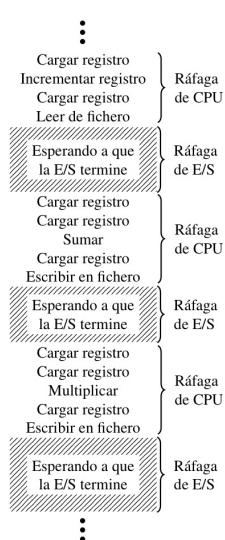
Procesos por lotes → cualquiera de los dos, pero si es apropiativa, con largos tiempos de CPU

## CICLO DE RÁFAGAS CPU Y E/S

La ejecución de un proceso consiste en un ciclo de ejecución en la CPU, un ciclo de espera de E/S y así sucesivamente. Se comienza y se termina con una ráfaga de CPU.

Esto es lo que hace posible la multiprogramación, unos procesos pueden estar en ráfagas de E/S y otro en CPU.

- Un proceso limitado por E/S pasa la mayor parte del tiempo a la espera de una operación E/S, suele tener muchas pero breves ráfagas de CPU. Interpretación de órdenes.
- Un proceso limitado por CPU necesita la CPU la mayor parte del tiempo, suele tener pocas ráfagas de CPU pero largas. Resolución de ecuaciones.



## ALGORITMOS DE PLANIFICACIÓN

FCFS: "Primer en llegar, primero en ser servido"

El más sencillo de todos. Fácil de implementar con una cola. Sin embargo el tiempo medio de respuesta puede ser bastante largo.

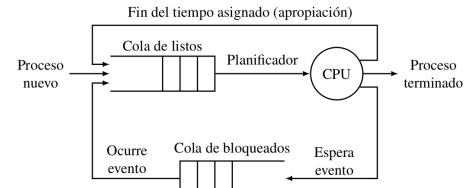
Proceso	Duración de la ráfaga
P1	24
P2	3
P3	3

$\frac{24 + 27 + 30}{3} = 27$

Ilegan los 3 a la vez

Tabla 2.1. Tres procesos y sus duraciones de ráfaga de CPU.

	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>
0		24	27



Un problema es que puede producir el efecto convoy, que ocurre cuando un proceso limitado por CPU ocupa la CPU mucho tiempo mientras que los procesos limitados por E/S quedan inactivos. Cuando por fin puedes usar la CPU lo hacen de manera muy rápida así que la CPU queda inactiva hasta que el proceso limitado por CPU vuelve a usarla.

!! Se desaprovechan recursos !!

Es NO APROPIATIVO



La locomotora representa los procesos pesados (limitados por CPU) y los vagones los limitados por E/S

### SJF: "Primero el trabajo más corto"

Es un algoritmo de planificación no apropiativo adecuado para procesos por lotes, donde los tiempos de ejecución se suelen conocer de antemano debido a ejecuciones previas.

P <sub>2</sub>	P <sub>3</sub>	P <sub>1</sub>		$\frac{3 + 6 + 30}{3} = 13$
0	3	6	30	

También se puede intentar aplicar a los procesos interactivos. Como no se conoce el tiempo exacto de ejecución debemos hacer estimaciones:

$$E_t = \alpha \cdot E_{t-1} + (1-\alpha) \cdot T_{t-1}$$

donde " $\alpha$ " es el coeficiente de credibilidad (entre 0 y 1),  $E_t$  es la estimación,  $E_{t-1}$  es la estimación anterior y  $T_{t-1}$  es el tiempo real de la ejecución anterior.

Un " $\alpha$ " pequeño hace que se olviden con rapidez los tiempos de ejecuciones anteriores.

Para que este algoritmo sea óptimo es necesario disponer de los procesos de forma simultánea.

### SRTF: "Primero el que tenga el menor tiempo restante"

Es la versión apropiativa del SJF. Se quita la CPU para dársela a otro proceso listo con rafaga de CPU menor que lo que resta el proceso que la está usando.

Si hay empate el planificador decidirá cuál de los dos usa la CPU.

### RR: "Round robin o circular"

Se atiende al orden de llegada como en FCFS, pero se asigna a cada proceso un intervalo de tiempo de ejecución llamado **quantum**. Ahora se cambia de un proceso a otro cuando el que tiene la CPU:

- Consumir su quantum.
- Termina.
- Se bloquea.

Si consume su quantum se pone al final de la cola de procesos listos. Es apropiativo.

Un proceso nunca espera más de  $(n-1) \cdot q$  uds. de tiempo para volver a ejecutarse.

Es IMPORTANTE decidir la longitud del quantum:

- Si es pequeño → mucho cambio de proceso, demasiado trabajo de gestión.
- Si es grande → la CPU no se desperdicia, pero los últimos procesos tardan en atenderse, lento el tiempo de respuesta.

## Planificación por prioridad

Cada proceso tiene una prioridad y toma la CPU el proceso de mayor prioridad. La prioridad se asigna en el momento de la creación del proceso; aunque esta asignación puede ser **estática o dinámica** (esta favorecería a los procesos limitados por E/S).

Además, hay que decidir si el algoritmo va a ser apropiativo o no.

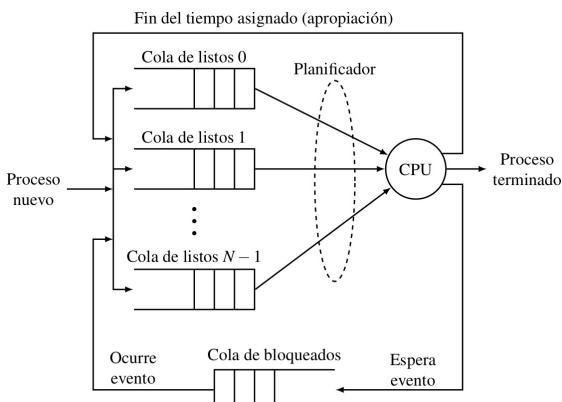
Un problema serio es el de la **inanición**, que surge cuando los procesos de baja prioridad nunca consiguen la CPU. Esto se puede solucionar de dos maneras: o bajando la prioridad del proceso que se está ejecutando, o subiendo la prioridad de aquellos que estén listos de baja prioridad.

## Planificación de múltiples colas con realimentación

Es la planificación más general y más compleja. Un proceso irá a una cola u otra en función de ciertos criterios como el **tipo de proceso, su importancia, la última cola que usó, consumo de CPU, etc.**

Al asignar la CPU habrá que elegir de qué cola se escoge y qué proceso. Existirá una planificación por colas y una dentro de cada cola. Generalmente, entre colas se suele establecer una planificación apropiativa por prioridad, se reparte el tiempo de CPU entre las colas y que cada una lo administre a su manera.

Además, **un proceso puede cambiar de una cola a otra** (realimentación)



### Parámetros:

- N° colas
- Algoritmo entre colas
- Algoritmo para cada cola
- Criterios de ascenso y descenso en colas.
- Cola inicial de un proceso nuevo

## PLANIFICACIÓN A CORTO, MEDIO Y LARGO PLAZO

Hasta ahora hemos considerado que los problemas ejecutables están en memoria, pero si no se dispone de suficiente memoria algunos estarán en disco.

Al tener esto en cuenta, debe existir un **planificador a corto plazo (PCP)**, que planifica los procesos que están en memoria, y un **planificador a medio plazo (PMP)**, que planifica el intercambio de procesos de memoria a disco y viceversa (**reanudación y suspensión de procesos**).

Criterios que puede usar el PMP:

¿Cuánto tiempo ha transcurrido desde el último intercambio?

¿Cuánto tiempo de CPU ha utilizado recientemente el proceso?

¿Qué tan grande es el proceso?

¿Qué tan alta es la prioridad del proceso?

También puede existir un **planificador a largo plazo (PLP)** que decide, de entre los trabajos por lotes, cuál entra en el sistema para su ejecución.

# SEGURIDAD Y PROTECCIÓN

## Tema 3

### 1. Introducción

La **seguridad** y la **protección** son dos cosas distintas en el ámbito de los S.O. La **protección** tiene un **carácter interno**, ya que es una tarea encargada al sistema operativo. La **seguridad**, en cambio, tiene un carácter más **general** en el que se incluyen, además de la protección, la política de copias de seguridad, la localización física del sistema, etc.

### 2. Seguridad

#### AMENAZAS A LA SEGURIDAD

Debemos tener claras las **requisitos de seguridad** que se deben cumplir en un sistema.

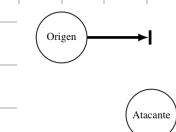
- > **Confidencialidad:** la información solo debe ser leída por usuarios autorizados. Este tipo de acceso incluye también otras formas de obtención de la información (el conocimiento de la existencia del objeto).
- > **Integridad:** exige que los elementos puedan ser modificadas solo por usuarios autorizados.
- > **Disponibilidad:** exige que los elementos estén disponibles solo para usuarios autorizados. (Para que, por ejemplo, no sean **eliminados** por otros usuarios)

#### TIPOS DE AMENAZAS

Estas amenazas se caracterizan mejor contemplando la función que del sistema como suministrador de información. Se produce un flujo de información desde un origen hacia un destino.

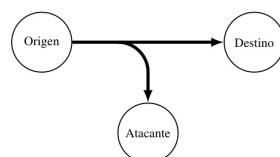
##### Interrupción

Se destruye un elemento del sistema o se hace inaccesible. Amenaza a la disponibilidad.



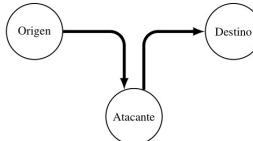
##### Intercepción

Una parte no autorizada consigue leer el elemento. Amenaza a la confidencialidad.



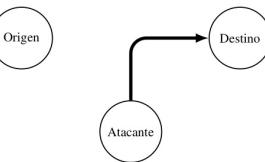
### Modificación

Una parte no autorizada accede y falsifica un elemento. Amenaza a la integridad.



### Invención

Una parte no autorizada inserta objetos falsos en el sistema. Amenaza a la integridad.



## AMENAZAS SEGÚN ELEMENTOS

Elemento	Disponibilidad	Confidencialidad	Integridad
Hardware	Robo o inutilización de equipos, eliminando el servicio. Consumo excesivo de recursos (CPU, disco, etc.).		
Software	Eliminación de programas, denegando el acceso a los usuarios.	Realización de copias no autorizadas del software.	Alteración de un programa en funcionamiento haciendo-lo fallar durante su ejecución o haciendo que realice alguna tarea no pretendida.
Datos	Eliminación de ficheros, denegando el acceso a los usuarios.	Lecturas de datos no autorizadas. Un análisis de datos estadísticos revela datos ocultos.	Modificación de ficheros existentes o invención de nuevos ficheros.
Líneas de comunicaciónes	Destrucción o eliminación de mensajes. Corte de las líneas de comunicaciónes o redes. Consumo excesivo de ancho de banda.	Lectura de mensajes. Observación del tráfico de mensajes.	Mensajes modificados, retardados, reordenados o duplicados. Invención de mensajes falsos.

## ATAQUES GENÉRICOS A LA SEGURIDAD

Para probar la seguridad de un sistema se contrata a un grupo de expertos para ver si son capaces de penetrar en el (equipo tigre o de penetración). Ataques más comunes que intenta este equipo:

- 1) Reserve espacio de memoria, disco o cinta y solo leído. Muchos sistemas no borran el espacio antes de asignarlo y podría contener info escrita por el usuario anterior.
- 2) Intenta llamadas al sistema inválidas, o llamadas válidas con parámetros inválidos, o llamadas válidas con parámetros válidos pero no razonables.
- 3) Conéctese a un sistema y oprima **DEL**, **CTRL+C** o **CTRL+PAUSA** a la mitad de la secuencia de acceso.

4. Engaña al usuario escribiendo un programa que muestre el mensaje **login:** en la pantalla y que después desaparezca. Muchos usuarios le indicarán el nombre y la contraseña, que registrará el programa.
5. Busca manuales que digan "no lleve a cabo X" e intenta tantas variaciones de X como sea posible.
6. Convence al administrador del sistema para que modifique el sistema con tal de que evite ciertas verificaciones vitales de seguridad para algunas usuarios.
7. Si todo lo anterior falla, el atacante debe encontrar al personal de administración del centro de cálculo y engañarlo o sobornarlo. No hay que subestimar los problemas que pueden causar los propios trabajadores.

## ATAQUES ESPECÍFICOS

**Bombas lógicas:** "estallan" en determinados instantes de tiempo. En muchas ocasiones, son creadas por el propio desarrollador, por ejemplo, para evitar un despido. Mientras mantenga su puesto, puede evitar que la bomba estalle.

**Puertas traseras (Backdoors):** son programas a través de los cuales es posible el acceso al sistema desde el exterior a como administrador. Se puede implementar, por ejemplo, como una cierta combinación de usuario + contraseña.

**Desbordamiento de buffer:** aprovecha los fallos de programación para sobrecargar de información al buffer. Un desbordamiento de buffer sobrescribe la pila de una función para que, cuando la función termine, ejecute un código cuidadosamente preparado, que generalmente se utiliza para convertirse en administrador.

**Caballos de Troya:** sustituyen una orden interna por una del mismo nombre pero que realiza acciones ilegales.

**Virus y gusanos:** programas cuya principal habilidad es extenderse dentro de ordenadores por medio de sistemas de almacenamiento.

- Un virus se encuentra en un programa y se copia de ahí a otros.
- Un gusano es un programa en sí que, a través de la red, se va copiando a sí mismo de un ordenador a otro.

**Spyware:** programas que se instalan sin que el usuario sea consciente y recopilan información.

**Rootkits:** Son programas que han corrompido el sistema de tal forma que no son fácilmente detectados y permiten que el atacante externo acceda como administrador. La habitual es que el rootkit proporcione una backdoor.

Normalmente, un rootkit es instalado por un atacante que ya ha accedido como administrador, llegando a modificar incluso el núcleo con nuevos drivers. La eliminación del rootkit requeriría instalar de nuevo el S.O.

## PRINCIPIOS DE DISEÑO

Los identificaron Saltzer y Schroeder (1975):

1. El diseño debe ser PÚBLICO.
2. El estado predefinido debe ser el de NO ACCESO.
3. Se debe verificar la autorización actual. Muchos sistemas verifican el permiso al abrir un fichero y no después de abrirlo.
4. Los procesos deben tener el mínimo privilegio posible que les permita seguir realizando su trabajo.
5. El mecanismo de protección debe ser simple, uniforme e integrado hasta las capas más bajas.
6. Debe ser psicológicamente aceptable, por la experiencia del usuario.

## 3. Protección

Tal como vimos en el primer tema, el hardware proporciona 3 mecanismos: hardware de direccionamiento de memoria (cada proceso tiene su espacio de direcciones), cronómetro y modo dual del procesador (núcleo y usuario).

Además, existen otros muchos elementos, tanto lógicos como físicos. El sistema debe disponer de un mecanismo que controle el acceso de los procesos a los recursos.

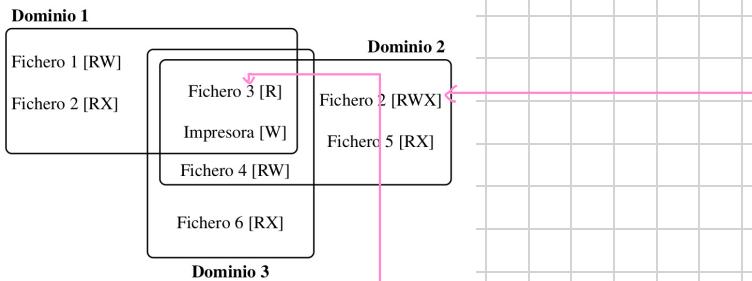
## POLÍTICA Y MECANISMO

La política indica que operaciones será posible realizar y mecanismo especifica cómo el sistema hará cumplir la política. El mecanismo es el encargado de llevar a cabo la política.

La separación de ambos conceptos es importante, ya que las políticas pueden cambiar de un sistema a otro, o con el tiempo. Por ello, es interesante contar con mecanismos generales.

## DOMINIOS DE PROTECCIÓN

Cada objeto, tanto hw como sw, tiene un nombre único mediante el que se referencia, además de un conjunto de operaciones a realizar sobre él. (p.ej: read y write para un fichero) Es necesario disponer de un mecanismo que impida que los procesos accedan a aquellos objetos sobre los que no tienen ningún permiso. Debe posibilitar que los procesos que puedan acceder a ciertos objetos realicen solo el subconjunto de operaciones permitidas.



Este mecanismo se llama dominio de protección, que es un conjunto de parejas (objeto, derechos). Un derecho representa el permiso. Los derechos son de: lectura (R), escritura (W) y ejecución (X). Un mismo objeto puede encontrarse en varios dominios con los = derechos, o con + derechos.

## MATRIZ DE ACCESO

El propio cambio de dominio se puede incluir con facilidad si se observa que un dominio también es un objeto con una operación del tipo "Entrar" que indica si puede entrar en el dominio correspondiente. La propia matriz se puede ver como un objeto fácil de modificar.

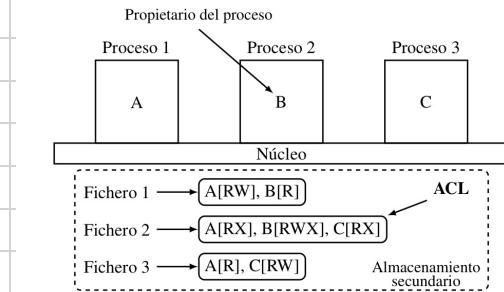
		Objetos									
		Fichero 1	Fichero 2	Fichero 3	Fichero 4	Fichero 5	Fichero 6	Impresora	Dominio 1	Dominio 2	Dominio 3
Dominios		Leer	Escribir	Leer				Escribir			Entrar
1											
2			Leer Escribir Ejecutar	Leer	Leer Escribir	Leer Ejecutar		Escribir			
3				Leer	Leer Escribir		Leer Ejecutar	Escribir	Entrar		

Sin embargo, como la mayor parte de los dominios solo tiene acceso a un nº reducido de objetos, la matriz estará casi vacía.

Existen 2 métodos que guardan solo los elementos no vacíos de la matriz, lo que nos permite REDUCIR EL ESPACIO OCUPADO: listas de control de acceso y listas de posibilidades.

## LISTAS DE CONTROL DE ACCESO

En las listas de control de acceso o **ACL** la matriz de protección se almacena **por columnas**, es decir, se asocia a cada **objeto** una lista con todos los dominios que pueden acceder al objeto y los derechos de cada uno.



Lo más razonable sería guardar las ACL en disco junto con cada fichero, **PERO** están en el núcleo para que no puedan ser manipuladas directamente por los procesos (deben enviar una solicitud al SO).

Además de los permisos básicos (rwx) el sistema puede definir otros como: borrar, copiar, ordenar, etc.

Las ACL también se pueden aplicar a grupos. Si una ACL tiene tanto entrada para usuario como entrada para grupos donde pertenece se deberá establecer un criterio para ver que derechos prevalecen.

Un fichero solo se comprueba CUANDO SE ABRE, un proceso podrá acceder hasta que se cierre aunque se le haya **revocado** el permiso.

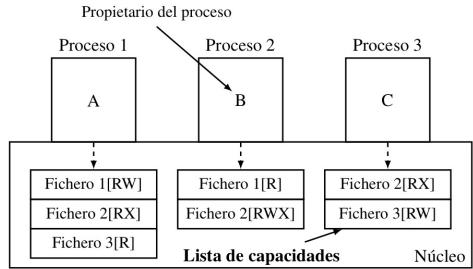
## POSIBILIDADES O CAPACIDADES

Las listas de posibilidades almacenan una matriz de protección **por filas**, por lo que, asocian a cada dominio una lista de los objetos a los cuales puede acceder junto con una indicación de las operaciones permitidas sobre cada uno.

Cada proceso tendrá su propia lista, en la que cada elemento recibe el nombre de posibilidad.

Cada entrada tiene una **posición**, de hecho, en Unix cuando se hace una operación sobre un fichero, se hace a través de su **descriptor de fichero** (que es el que indica la posición que ocupa en la lista de posibilidades).

Estas listas se almacenan en el núcleo y los procesos de usuario hacen referencia a las posibilidades mediante su **número**.



Además de los permisos habituales, las posibilidades suelen tener derechos genéricos como:

- Copiar posibilidad: nueva posibilidad para el = objeto.
- Eliminar posibilidad.
- Destruir objeto: elimina un objeto y una posibilidad.

Una característica interesante es que un proceso puede pasar su capacidad a otras procesos, generar una versión + restrictiva, etc.

Para construir una posibilidad para un objeto, Unix combina ACL con listas de posibilidades:



When a file is opened, it checks the ACL for permissions (if none, it returns an error) and creates a possibility, returning the occupied position. Subsequent accesses use this possibility to quickly verify if access is allowed. When the file is closed, the possibility is destroyed.

## COMPARACIÓN

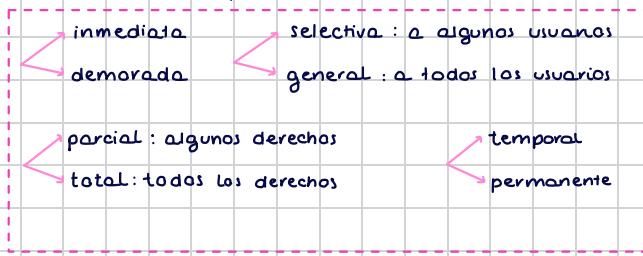
Las ACL se corresponden con las necesidades del usuario en cuanto a la creación de objetos y asignación de permisos y dominios. Sin embargo, como la información de dominio no está toda juntas, hay que comprobar cada acceso al objeto (búsqueda lenta).

Las listas de posibilidades son más complejas para el usuario pero son útiles para saber a qué objetos puede acceder un proceso.

La solución más común es combinar el uso de ambas.

## CANCELACIÓN O REVOCACIÓN

La cancelación puede ser:



Con las listas ACL la cancelación es bastante sencilla (cualquiera de las cancelaciones).

Sin embargo, en las posibilidades un objeto puede estar distribuido por el sistema.

Hacer un registro de las posibilidades

de cada objeto CONSUMIRÍA MUCHA MEMORIA. Existen dos técnicas eficientes para este problema:

1) Hacer que cada posibilidad apunte hacia un objeto indirecto; si el indirecto apunta al real el sistema puede romper esta conexión, invalidando las posibilidades.

(ninguna de las técnicas permite una cancelación selectiva ni parcial)

2) Asignar a cada objeto un número aleatorio (enorme para evitar colisiones) y almacenarlo en la posibilidad. Si el n° aleatorio y el del objeto coinciden se permite la operación. Si el propietario del objeto solicita que se cambie el n°, se invalidan las posibilidades.

## 4. Autenticación de usuarios

El principal problema de la seguridad para los S.O. es el de la **validación**. El problema de la identificación de usuarios se conoce como **"autenticación de usuarios"**.

Existen numerosas métodos, pero el más común es **identificador + contraseña**.

La contraseña "autentica" al usuario y el ID determina:

1. Si el usuario está autorizado para acceder al sistema.
2. Los privilegios acordados con el usuario. Un usuario (o grupo reducido) tendrá **estatus de supervisor, administrador o "superusuario"** que le permitirá llevar a cabo funciones especiales. Algunos SO disponen de **cuentas anónimas** con **privilegios restringidos**.

### CONTRASEÑAS

A la hora de almacenar una nueva contraseña para un usuario, se pasan 3 parámetros a la **rutina de cifrado crypt()**: la contraseña, el valor "base" y el algoritmo de cifrado a utilizar. Se guarda en este formato:

`$tipo $base $contraseña_cifrada`

El tipo indica mediante un número el algoritmo de cifrado (1 para MD5, 5 para SHA256 y 6 para SHA512).

El valor de la base está asociado con el momento en el que se asigna la contraseña (para impedir que contraseñas iguales se cifren igual).

Las contraseñas se guardaban originalmente en `/etc/passwd` pero como ese fichero necesita permiso de lectura para todos los usuarios se almacenan ahora en `/etc/shadow`.

Para acceder al sistema el SO utiliza el ID como índice en el fichero de contraseñas. Pasa los 3 parámetros a `crypt()` y el resultado se compara con la contraseña del fichero.

## ESTRATEGIAS DE ELECCIÓN DE CONTRASEÑAS

No existe una **inversa de la función hash** utilizada para el cálculo de la contraseña. Además el diseño de estas funciones evita el ataque por adivinación.

Sin embargo, el usuario debería elegir una contraseña no muy fácil de adivinar. Existen 2 técnicas para ayudar al usuario en esta tarea:

### instrucción del usuario

Se explica al usuario la importancia de seleccionar contraseñas difíciles de adivinar. Muchos sistemas solo admiten contraseñas con ciertos criterios: caracteres no alfanuméricos, mayúsculas y minúsculas., etc.

### inspección proactiva

Un usuario elige su contraseña pero si el sistema detecta que es demasiado fácil la rechaza. El administrador puede ejecutar **cracks** (programas de averiguación de contraseñas).

## 5. Protección en Unix

### DOMINIOS EN UNIX

En Unix cada usuario se identifica mediante un número (UID). Este número se asocia al login. Además, cada grupo de usuarios también tiene un identificador (GID). Se asocia el GID con el grupo en el fichero **/etc/group**, junto con la lista de usuarios del grupo.

Puede haber varios usuarios y grupos con el mismo UID o GID.

Cada par de valores (UID, GID) define un **dominio** en Unix.

Entre todos los **UID el más especial es el 0**, generalmente asociado al "root". Algunas distribuciones de Linux no permiten iniciar sesión como root, cualquier tarea administrativa se hace a través de la orden **sudo**.

### ACL RESTRINGIDAS

Cada fichero y dispositivo (que tenga una entrada asociada en /dev) pertenece a un usuario y a un grupo. Las ACL asociadas a estos ficheros son **restringidas** por estar limitadas a 3 conjuntos: **1 propietario, 2 grupo y 3 otros usuarios**.

Otros sistemas implementan listas **completas** para especificar los permisos de cada usuario y grupo en concreto.

Para comprobar el acceso se llevan a cabo los siguientes pasos:

1. Si el UID del proceso y del fichero coinciden el proceso tiene los permisos del propietario.
2. Si no, se comprueba el GID y si coinciden tiene entonces los permisos del grupo.
3. Si lo anterior tampoco se cumple, el proceso tiene los permisos del resto de usuarios.

## CAMBIO DE DOMINIO

Cada proceso se ejecuta siempre con el dominio (UID, GID) del usuario que lo crea. De forma excepcional los ejecutables pueden poseer el bit SETUID y el SETGID, que pueden cambiar el dominio final al que pertenece un proceso.

Si tiene activado este bit, al hacer la llamada execve(), cambiará su UID por el del usuario propietario del ejecutable. Lo mismo ocurre con el bit SETGID.

Un posible cambio de dominio hace que cada proceso tenga 2 parejas de identificadores: la real (UID, GID) y la efectiva (EUID, EGID). La primera corresponde con el par del proceso padre y la segunda es el par con el que el proceso accede a los ficheros de forma efectiva, y, por tanto, es el que realmente determina lo que puede hacer el proceso.

Normalmente estas parejas son iguales excepto cuando están activos los bits SETUID, SETGID.

```
-rwsr-xr-x root root 20 Abr 12:00 passwd
```

Lo podemos entender mejor con el ejemplo de cuando queremos cambiar la contraseña con passwd() y hay que modificar /etc/shadow para el que no tenemos acceso.

Aun así, con getuid() el SO sabe quién está cambiando la contraseña (solo dejará cambiar la tuya excepto si eres root).

## COMBINACIÓN DE ACL Y POSIBILIDADES

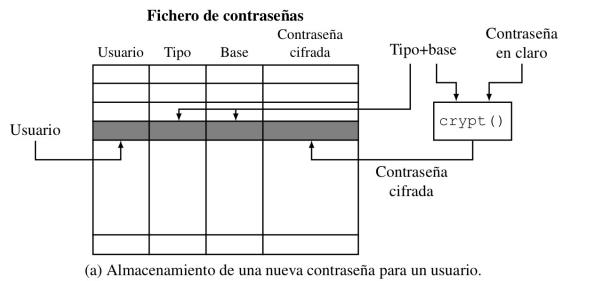
Cuando un proceso quiere abrir un fichero se comprueba en la ACL sus permisos y se crea una posibilidad. Se crea una entrada en la tabla de ficheros abiertos y devuelve un índice (descriptor de fichero).

Cuando se cierra el fichero se elimina la entrada de la tabla.

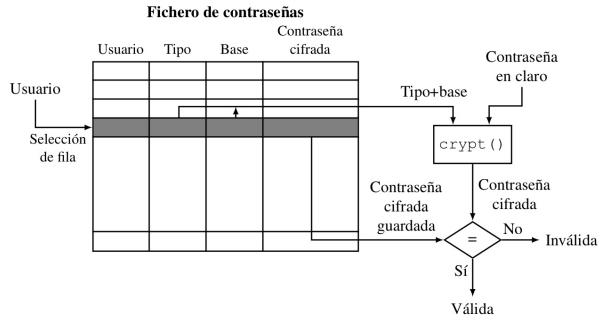
El programa de usuario solo puede acceder a los ficheros abiertos.

Si se abre una posibilidad de solo lectura no dejará escribir aunque esté en los permisos.

Solo se comprueban los permisos al abrirlo, así que si sigue abierto da igual que cambien.



(a) Almacenamiento de una nueva contraseña para un usuario.



(b) Verificación de la contraseña introducida por un usuario.

Figura 3.6. Esquema de contraseñas de Unix.

# SISTEMAS DE FICHEROS

## Tema 4

### 1. Introducción

Los sistemas actuales necesitan algún tipo de **almacenamiento secundario** para completar al almacenamiento principal que proporciona la memoria RAM, debido a estas 3 razones:

1. Para **almacenar más** cantidad de lo que cabe en RAM
2. Para **preservar la información** para que esta no desaparezca cuando termine el proceso.
3. Permitir que la información sea **accedida** de forma concurrente por varios procesos.

El almacenamiento secundario, en realidad, no es más que **un array de bloques de información**. Debido a esto, el SO ha creado **abstracciones** que facilitan a los usuarios y programadores su uso.

La unidades de información son **ficheros** que se organizan en **directorios**. Este sistema de ficheros debe estar diseñado correctamente.

### 2. Ficheros

Un fichero es la unidad lógica de almacenamiento. Sirven para guardar y leer información de un dispositivo de almacenamiento secundario.

En sistemas modernos, un fichero es una **secuencia de bytes** cuyo significado está definido por los procesos que acceden a él.

Cada fichero tiene un nombre, cuyos criterios dependerán del SO, realmente el SO no entiende la estructura interna de los ficheros, **SOLO LA DE LOS EJECUTABLES**.

### TIPOS DE FICHEROS

REGULARES	DIRECTORIOS	ESEPCIALES DE CARACTERES	ESP. DE BLOQUES
Texto ↓ ASCII UTF-8 Latin1 ...  Binarios ↓ incomprensibles al ojo humano (estruct. interna)	ficheros gestionados por el SO, organizar y registrar ficheros	relacionadas con la E/S, para referenciar y acceder a ellas	para referenciar y acceder a disp. de almacenam. sec. de acceso aleatorio.

## Ejemplos

Muchas veces el nombre de un fichero contiene una **extensión**, aunque en los sistemas modernos no es significativa. Sin embargo, programas como el **explorador de archivos**, pueden hacer uso de esta para que sepa qué acción realizar al hacer doble click sobre el fichero. Además, estas extensiones ayudan al usuario a saber el tipo de información del fichero.

Extensión	Significado
fichero.bak	Copia de seguridad
fichero.c	Código fuente en C
fichero.gif	Imagen en formato GIF
fichero.hlp	Fichero de ayuda
fichero.html	Código fuente en HTML
fichero.jpg	Imagen fija en formato JPEG
fichero.mp3	Sonido codificado en MP3
fichero.mpg	Vídeo codificado en MPEG
fichero.o	Fichero objeto (salida del compilador, todavía no enlazada)
fichero.pdf	Fichero PDF
fichero.ps	Fichero PostScript
fichero.tex	Fichero fuente de TeX
fichero.txt	Fichero de texto genérico
fichero.zip	Archivo comprimido con formato ZIP

## ACceso A UN FICHERO

Hay dos principales accesos: **secuencial** y **aleatorio**. En el secuencial los bytes se leen en orden, en el aleatorio se sigue un orden cualquiera (con llamadas como lseek). Realmente el acceso secuencial es un caso particular del acceso aleatorio. Hay ficheros donde el único acceso posible es el **secuencial**, por ejemplo: en los **Ficheros especiales de caracteres** que representan teclados o cintas, o las tuberías.

## ATRIBUTOS DE FICHERO

Además del nombre y del contenido, los SO asocian **información adicional** a cada fichero. A cada elemento se le llama atributo.

- Hay atributos relacionados con la **protección**: propietario, permisos... etc.
- Las **banderas** son bits que permiten o no cierta propiedad: de ocultación, temporal, de biblioteca..., etc.
- Atributos de **tiempo**: fecha de creación, de modificación, etc.
- Atributos de **tamaño**: tamaño actual, máximo.

## OPERACIONES

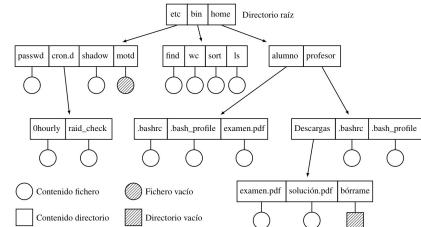
- **Create**: crea un fichero vacío.
- **Delete**: libera el espacio que ocupa un fichero, su contenido no se elimina inmediatamente.
- **Open**: abre un fichero para usarlo.
- **Close**: lo cierra.
- **Read**: lee la cantidad de bytes indicada como parámetro.
- **Write**: escribe en un fichero la info dada como parámetro.
- **Append**: añade datos al final del fichero.

- Seek: indicar una posición para leer o escribir.
- Get attributes
- Set attributes
- Rename : cambiar nombre o moverlo de directorio.
- Truncate: elimina su contenido a partir de una posición dada.

### 3. Directorios

#### SISTEMAS JERÁRQUICOS DE DIRECTORIOS

Se utiliza un árbol de directorios para organizar los ficheros. Un usuario puede tener tantos directorios como desee y agrupar los ficheros según sus necesidades.



#### RTA DE ACCESO

Cada fichero tiene una ruta de acceso absoluta, que va desde el directorio raíz hasta el fichero.

Cada fichero tiene una ruta de acceso relativa que se construye indicando los directorios que hay que recorrer para llegar desde donde estamos hasta el fichero.

En cada directorio hay 2 directorios especiales: ".." (directorío actual) y "../" (directorío padre). Estas entradas nos pueden ayudar a construir las rutas.

OJO: Debido a la existencia de ".." y "../", no hay solo una ruta absoluta.

#### OPERACIONES

- Create: crea un directorio "vacío" (con las entradas .. y .)
- Delete: elimina un directorio, siempre que este "vacío".
- OpenDir: abre un directorio para ser recorrido (obtiene una lista de los ficheros y subdirectorios que contiene).
- ReadDir: devuelve la siguiente "entrada" (nombre de fichero o directorio) de un directorio abierto.
- Rename: cambia de nombre o localización.

- Link: permite que un fichero aparezca a la vez en un directorio con distintos nombres o en varios directorios.
- Unlink: elimina una entrada del directorio. Si solo hay una equivale a borrar el directorio.

¿Por qué usamos "readdir" y no "read"?

Porque es el SO el que determina el formato de los directorios, el SO conoce su estructura interna, no los programas.

¿Por qué no existe "write"?

Escribir en un fichero equivale a crear, borrar o renombrar ficheros dentro de él.

## 4. Implementación

Para facilitar el uso de los dispositivos de almacenamiento secundario, muchos de ellos exportan una interfaz que permite verlos como un array de bloques. Otras veces es el SO el encargado de crear esa interfaz. Debido a esto, a estos dispositivos se les conoce como "dispositivos de bloques".

Tanto el tamaño como el n.º de bloques dependen del dispositivo. A cada posición del array la llamamos dirección.



## IMPLEMENTACIÓN DE FICHEROS

Todo fichero tiene asociado un conjunto de bloques. Cuando hablamos de implementación hablamos de la forma de llevar un registro de esos bloques, es decir, saber donde están.

### Asignación contigua

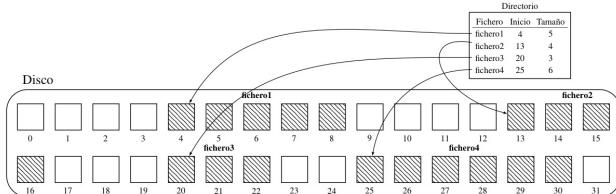
Cada fichero se almacena como un conjunto de bloques adyacentes.

#### Ventajas

- Fácil implementación. El bloque X se encontrará en D + X (D es el inicio)
- Ofrece un rendimiento excelente por el fácil recorrido. Se pueden leer varios bloques con una operación.

#### Inconvenientes

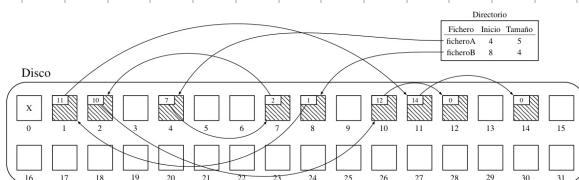
- Se necesitaría conocer el tamaño máx de cada fichero. (en DVDs si sirve)
- Fragmentación externa\*: el espacio libre está repartido en pequeños huecos.



\* Hay espacio para crear un fichero de 5 bloques, pero no se va a poder porque no hay 5 libres seguidas.

### Asignación mediante lista ligada

En cada bloque se guardará un puntero al siguiente bloque



### Ventajas

- No hay fragmentación externa.
- Solo hace falta que se guarde la dirección del primer bloque.

### Inconvenientes

- Si quisieramos añadir algo al final del fichero tendríamos que leer todos sus bloques → acceso lento
- El espacio para almacenar datos ya no es potencia de 2. El puntero ocupará espacio en el bloque. Esto es un problema porque a veces los programas leen los datos en bloques que son potencia de 2.

### Asignación mediante lista ligada e índice

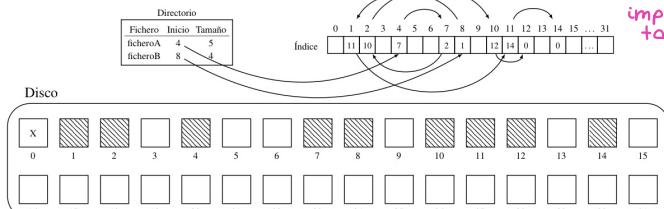
Almacenar los punteros en una tabla en memoria. Cada entrada es un bloque. La dirección de un bloque  $X$  es  $D_x$ ,  $D_{x+1}$  se almacenará en la entrada de  $D_x$ .

### Ventajas

- El acceso aleatorio es mucho más rápido.
- Los punteros ya no se guardan en el bloque.

### Inconvenientes

- Toda la tabla debe estar en memoria, lo que es un problema si la tabla es GRANDE debido a la capacidad del disco. Se podrían usar BLOQUES GRANDES pero entonces se desperdicia espacio dentro de ellos.
- Si una modificación de la tabla no se escribe en disco por cualquier problema, ciertos ficheros podrían desaparecer.



FAT32  
usa esta  
implementación  
!!

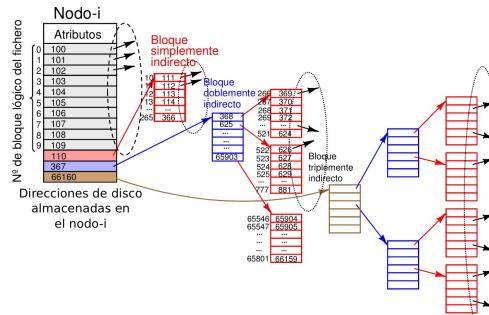
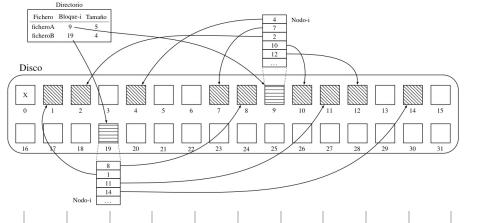
## Nodos-i

A cada fichero se le asocia una pequeña tabla llamada nodo-i (nodo índice) la cual contiene las direcciones de los bloques del fichero en orden.

El nodo-i de cada fichero se almacena en disco. Como la mayoría de ficheros son pequeños, el nodo-i guarda pocas direcciones, por lo que se usan nodos-i pequeños.

Sin embargo, ¿qué pasa con los ficheros grandes? La solución es usar **bloques indirectos**.

Una de las direcciones del nodo-i será la dirección de un bloque en disco: **bloque simplemente indirecto (BSI)**, que guardará punteros a bloques de datos. Si esto no es suficiente, otra dirección será la dirección de un **bloque doblemente indirecto (BDI)**, cuyo contenido son direcciones de BSI's. Si aun no es suficiente se pueden usar **bloques triplemente indirectos (BTI)**. Unix utiliza este esquema.



En los sistemas de ficheros que usan nodos-i de este tipo, todos los nodos se guardan juntos en bloques consecutivos de disco a los que se llama **tabla de nodos-i**. Cada nodo-i recibe un nº que es el que se guarda en la entrada de directorio correspondiente a su fichero. Sabiendo el nº del nodo-i y cuántos nodos caben en un bloque, sabremos qué bloque de la tabla de nodos-i tenemos que leer y la posición del nodo-i dentro de ese bloque.

Además de las direcciones de disco, un nodo-i en Unix guarda también **los atributos del fichero**. Esto determina la forma en la que se implementan los directorios.

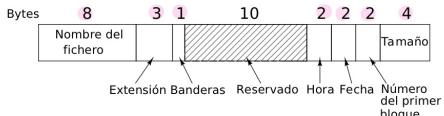
## IMPLEMENTACION DE DIRECTORIOS

Recordemos que un directorio es un fichero que maneja el SO y contiene las entradas a los ficheros que están dentro de él.

Los atributos de un fichero se guardan, o bien junto al nombre en la entrada de directorio o en una estructura aparte.

## Directorios en MS-DOS

En MS-DOS, los directorios son ficheros que almacenan una lista desordenada de entradas o registros de 32 bytes, una por fichero. Cada una se divide en varios campos.



- Nombre + extensión (8 + 3)
- Atributos (1) : varios bits se usan como banderas.
- Usos futuros (10)
- Hora de la última modificación (2) : guarda sólo los segundos pares.
- Fecha de la última modificación (2)
- Dirección del primer bloque (lista ligada con índice)
- Tamaño del fichero en bytes (hasta 4GiB)

El directorio raíz no está implementado como fichero, si no que ocupa unos bloques FIJOS en disco, lo que hace que tenga un tamaño máximo preestablecido.

MS-DOS permite crear un árbol de ficheros de tamaño arbitrario.

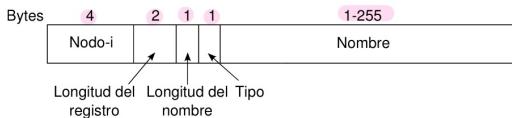
Esta estructura de directorios es utilizada (con extensiones y mejoras) por FAT32 y vFAT.

## Directorios en Unix

Encontramos una diferencia con MS-DOS que es que el directorio raíz no tiene ningún tratamiento especial. Cada entrada ocupaba 16 bytes en las PRIMERAS VERSIONES, dos para guardar el nodo-i asociado y 14 para almacenar el nombre.

Ahora, las entradas tienen la siguiente estructura:

- Nº de nodo-i (4)
- Longitud del propio registro (2)
- Longitud del nombre (1) que es hasta 255 caracteres (o menos, si se usan caracteres multibyte).
- Tipo de entrada : fichero o directorio (este campo está tmb en el nodo-i pero sirve para acelerar los listados).



¿Para qué necesitamos el campo longitud del registro?

Si renombramos una entrada y usamos un nombre más corto se desperdiciarán bytes, ya que no se generará un hueco. Si el tamaño del registro es múltiplo de 4 será frecuente que queden bytes libres en el nombre para poder renombrar sin tener que moverlo.

2. Otra razón es la gestión del espacio libre dentro del fichero que contiene al directorio. Cuando se crea un directorio, contiene ya dos entradas (.. y .) cada una se guarda en un registro. En la entrada ".." es donde se registra el espacio libre que queda en la entrada de directorio, tras crear..."

Si ahora se crea un fichero nuevo, se buscará un registro con espacio libre para crear un nuevo registro, dividiendo ".." en dos: uno para la entrada que ya existe y otro para la nueva entrada, que se quedará con el resto del espacio disponible.

Un registro no puede atravesar la frontera entre bloques.

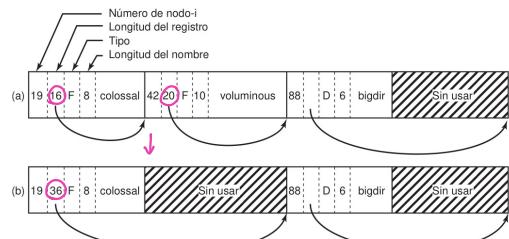
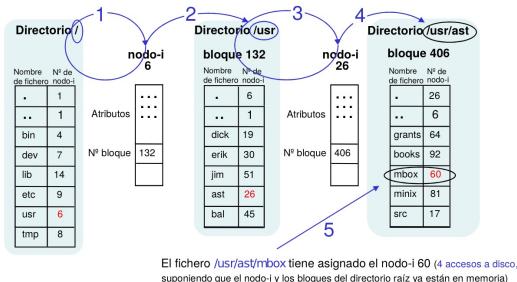


Figura 4.12. Eliminación de una entrada de directorio en Unix. (a) Tres entradas de directorio donde el campo «Longitud del registro» nos permite ir saltando fácilmente de una entrada a la siguiente. (b) Eliminación de la entrada «voluminous». Se observa que el espacio ocupado por esta entrada se añade al tamaño del registro de la entrada anterior.

## RESOLUCIÓN DE RUTAS

Vamos a ver como, dada una ruta, se accede al nodo-i asociado a ese fichero.



3. Ahora dentro buscamos la entrada *last*, nos dice que su nodo-i es el 26.

4. Del nodo-i 26 obtenemos la dirección del bloque de datos (406).

5. Finalmente, en el bloque de datos /usr/ast buscamos la entrada *mbox* que nos dice que su nodo-i es el 60. Ya podemos acceder a sus datos.

4 ACCESES A DISCO.

no existen entradas reales para ".." y ".."

(Suponemos que el nodo-i y el bloque de datos "/" están en memoria)

1. Buscamos en el bloque de datos del directorio raíz la entrada */usr*.

2. Del nodo-i 6 obtenemos la dir. del bloque de datos de */usr*.

## FICHEROS COMPARTIDOS

Cuando un fichero se puede acceder desde varios directorios con el mismo nombre o desde el mismo directorio con distinto nombre, se dice que es un **fichero compartido**.

Se pueden implementar de dos formas:

- Que los datos relativos a un fichero (atributos...) se guarden en una estructura a la que apunten las entradas de directorio. En Unix ocurre esto, varias entradas pueden guardar el mismo n.º de nodo-i (**enlace físico o hard link**). Sería necesario implementar un **contador de enlaces** en el nodo-i.
- La segunda forma es creando un nuevo tipo de fichero cuyo contenido sea la ruta de acceso del fichero al que se enlaza. Se les llama **enlaces simbólicos o soft links**. Se distinguen de los ficheros regulares usando un bit de bandera. Si el fichero se borra, el enlace simbólico simplemente fallará. El problema de estos enlaces es su **ciclo temporal** (2 rutas se deben resolver) y **espacial** (un nodo-i entero para guardar una ruta).

Sin embargo, los **enlaces simbólicos** son más flexibles. Unix no permite crear enlaces físicos de directorios, pero sí simbólicos, y no permite crear un enlace físico a un fichero de un sistema de ficheros distinto al suyo, pero sí simbólico.

Un problema de todos los enlaces es la **DUPPLICACIÓN DE DATOS**.

## ADMINISTRACIÓN DE ESPACIO EN DISCO

### Tamaño de bloque lógico

Los discos tienen como unidad mínima el **bloque físico**, suelen tener tamaños de 512, 1024, 2048 e, incluso, 4096 bytes. Hacen que estos se vean en un array lineal.

Es común, que el SO agrupe los bloques físicos para formar **bloques lógicos** o unidades más grandes (o pequeñas).

**Bloques grandes** → mucha fragmentación interna pero pocos bloques por fichero

**Bloques pequeños** → poca fragmentación interna pero muchos bloques por fichero.

El tamaño del bloque afectará al tamaño de la **tasa de transferencia** (lect/escr.).

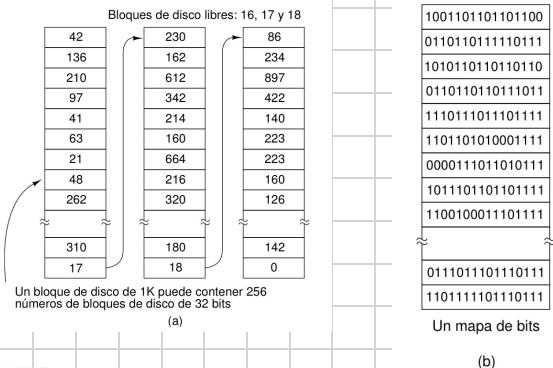
Habrá que buscar un equilibrio entre eficiencia de espacio y tasa de transferencia.

## Registro de bloques libres

Se suelen usar 2 métodos:

### Lista ligada de bloques

Cada bloque de la lista contiene tantos nº de bloques libres como pueda, además de un puntero al siguiente bloque. Por ejemplo, bloques lógicos de 1KiB y 32 bits por nº de bloque, un bloque será capaz de almacenar 256 direcciones (255 bloques libres y 1 al sig.).



### mapa de bits

Un disco con  $N$  bloques necesita un mapa de bits con  $N$  bits. Los bloques libre se representarán con un 0. El mapa de bits siempre tendrá el mismo tamaño porque depende del tamaño del disco.

De las 2 opciones, la más utilizada en la práctica es el mapa de bits (Ext3, Ext4 en Linux, o NTFS en Windows). Un mapa de bits ocupa menos espacio que la lista ligada. Solo ocupará más bloques que la lista ligada si el disco está casi lleno. Además, con el mapa podemos buscar secuencias de bloques libres.

La FAT utiliza la lista ligada con índice, no solo para registrar los bloques de cada fichero, si no para registrar también los que están libres (con alguna marca).

## 5. Caché de disco

El acceso a disco es bastante más lento que el acceso a memoria. Existen mecanismos para mejorar el rendimiento, como la **caché de disco**. La implementa el SO y usa una parte de la memoria principal y contiene bloques que pertenecen al disco pero se mantienen temporalmente en memoria principal.

Funcionamiento: cuando se lee un bloque, se comprueba si está en memoria principal, si está, no se accede a disco. Si no está, se lee y se coloca en caché. Si la caché está llena, habrá que expulsar bloques. Para decidir cuáles se expulsan, usamos un algoritmo LRU con listas ligadas; aunque aplicar un LRU puro no es recomendable si queremos mantener la consistencia del sistema de ficheros.

Habrá que modificar el LRU para tener en cuenta lo siguiente:

1. ¿Es probable que se vuelva a necesitar pronto?
2. ¿Es esencial para la consistencia del sistema?

Según la primera, los bloques recién usados que probablemente no se vuelvan a usar pronto (los indirectos) pasarán al final de la LRU.

Según la segunda, si un bloque es esencial para el sistema de ficheros (nodo-i o mapa de bits) y se modifica, se debe escribir en disco rápidamente (sin necesidad de expulsar de memoria).

Tampoco es recomendable mantener los bloques de datos en caché, para no perder información.

→ Bloques de datos: se escriben en disco cada 30 seg.

En Unix → Bloques de metadatos: se escriben en disco cada 5 seg.

Con la orden `sync`, el usuario ordena que se escriban todos los bloques que han sido modificados.

## 6. Discos y sistemas de ficheros

### PARTICIONES

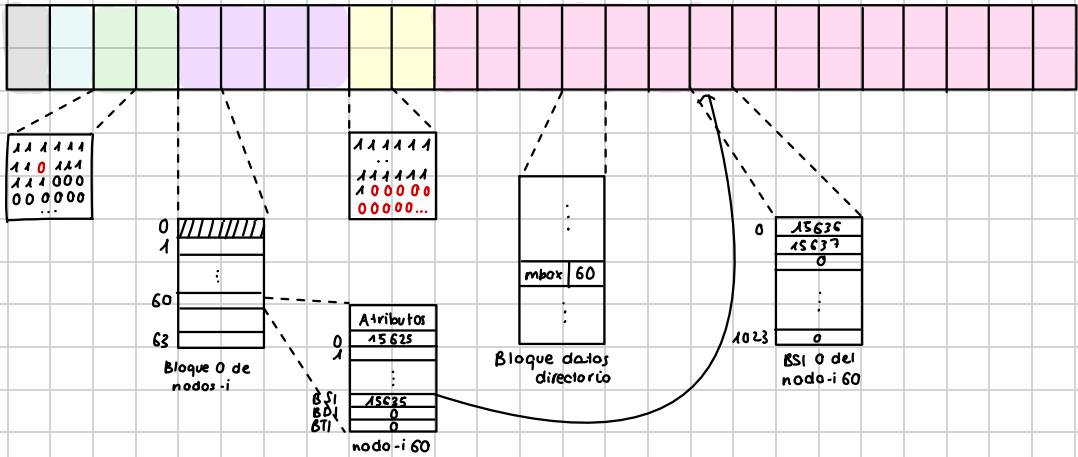
Una partición es una porción de bloques consecutivos de un disco. Las particiones son manejadas por el S.O., que las representa como un array lineal de bloques. Se ven como "pequeños discos" dentro de un disco mayor.

El nº y tamaño de las particiones, y su bloque de fin y comienzo, se guarda en el bloque 0 del disco: el **Master Boot Record**. Suele contener una porción de código de arranque que la BIOS ejecuta para arrancar un S.O. en el ordenador.

Esto sirve para, por ejemplo, tener 2 sistemas operativos, o para tener 2 sistemas de ficheros distintos (una **partición de intercambio** para guardar páginas que no caben en memoria principal).

### ESTRUCTURA DE UN SISTEMA DE FICHERO

El sistema de ficheros usa zonas de bloques consecutivos para guardar distintas estructuras de datos.



**Bloque de arranque:** es siempre el bloque 0, si hubiera un SO en la partición, contendría código de arranque, si no, estará vacío. Se puede usar su dirección como valor nulo.

**Superbloque:** es siempre el bloque 1. Contiene información **crítica** relativa a la organización del sistema de ficheros: nº total de bloques lógicos, nodos-i, etc. Su destrucción haría que el sistema quedara **ilegible**. Algunos sistemas tienen varias copias.

**Mapa de bits de nodos-i:** para saber qué nodos-i hay libres. Necesita  $\frac{I}{8 \cdot TB}$  bloques.

**Tabla de nodos-i:** contiene los nodos-i usados para ficheros. El tamaño de la tabla depende del tamaño de los nodos-i y de su número. Ocupa  $\frac{I \cdot T_b}{TB}$  bloques.

**Mapa de bits de bloques:** indica qué bloques están libres. Ocupa  $\frac{B}{8 \cdot TB}$  bloques.

**Bloques de datos:** de los ficheros, directorios, y bloques indirectos.

Ejemplo de creación del fichero /usr/ast/mbox:

Datos:

Bloques de 4KiB

0	AT
1	:
2	BSI
3	BD1
4	BD2

Direcciones 4 bytes

Nodos de 64 bytes → 13 direcciones

1. Se comprueba en el bloque de directorio /usr/ast no existe un mbox.
2. Buscamos un nodo-i libre en el mapa de bits de nodos-i (60).
3. Añadimos la entrada mbox al directorio y le asignamos el nodo-i.

4. Le vamos asignando bloques libres, marcándolos como ocupados (1) en el mapa de bits de bloques y guardando las direcciones en el nodo-i.
5. Como el fichero ocupa 45KiB ocupará 12 bloques (11.37), no caben más direcciones por lo que hay que asignarle el BSI.
6. Se escriben en el BSI las dos direcciones de datos que quedan.

Las posiciones no ocupadas contienen el valor 0, que es una "marca" de vacío ya que es una dirección de disco inválida.