

G4012223

2024 / 2025

# SISTEMAS OPERATIVOS

2º Ingeniería Informática  
1º Cuatrimestre

Escrito por:  
Lúa Gil Gómez  
Adrián Quiroga Linares  
Xabier Núñez Gómez



# 1

## Introducción - Empieza la Aventura

Una computadora moderna es un sistema complejo, y el trabajo de administrar todos sus componentes y utilizarlos de manera óptima es una tarea muy desafiante. Por esta razón, las computadoras están equipadas con una capa de software llamada **sistema operativo**, cuyo trabajo es proporcionar a los programas de usuario un modelo de computadora mejor, más simple y pulcro, así como encargarse de la administración de todos sus recursos, entre ellos, el hardware.

Algunos de los sistemas operativos más usados son Windows, Linux o Mac OS. El programa con el que los usuarios generalmente interactúan se denomina **Shell**, cuando está basado en texto, y **GUI (Graphical User Interface)** cuando utiliza elementos gráficos o iconos. Aunque no forma parte del sistema operativo, este lo utiliza para llevar a cabo su trabajo. Es el nivel más bajo del software en modo usuario y permite la ejecución de otros programas.

### 1.1. ¿Qué es un sistema operativo?

Los sistemas operativos realizan dos funciones básicas que no están relacionadas: proporcionar a los programadores de aplicaciones un conjunto abstracto de recursos simples, en vez de los complejos conjuntos de hardware; y administrar estos recursos de hardware.

Los verdaderos clientes del sistema operativo son los programas de aplicación (a través de los programadores de aplicaciones). Son los que tratan directamente con el sistema operativo y sus abstracciones. En contraste, los usuarios finales tienen que lidiar con las abstracciones que proporciona la interfaz de usuario, ya sea un shell de línea de comandos o una interfaz gráfica.

Por otra parte, el sistema operativo también está presente para administrar todas las piezas de un sistema complejo. Las computadoras modernas constan de procesadores, memorias, temporizadores, discos, ratones, interfaces de red, impresoras y una amplia variedad de otros dispositivos. El trabajo del sistema operativo es proporcionar una asignación ordenada y controlada de los procesadores, memorias y dispositivos de E/S, entre los diversos programas que compiten por estos recursos.

Los sistemas operativos modernos permiten la ejecución simultánea de varios programas. Su tarea es llevar un registro de que programa está utilizando qué recursos, de otorgar las peticiones de recursos, de contabilizar su uso y de mediar las peticiones en conflicto provenientes de distintos programas y usuarios. Esta administración de recursos incluye el **multiplexaje** (compartir) de recursos en dos formas distintas: en el tiempo (p.e. una sola CPU) y en el espacio (p.e. memoria principal).

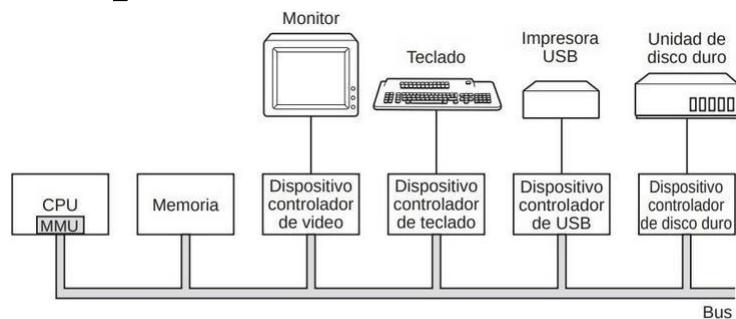


FIGURA 1.1. Algunos de los componentes de una computadora personal simple.

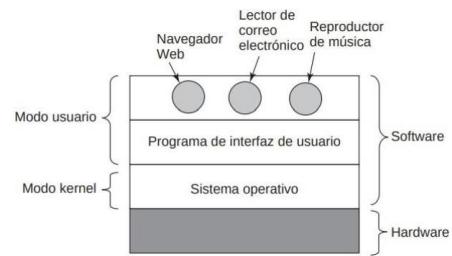
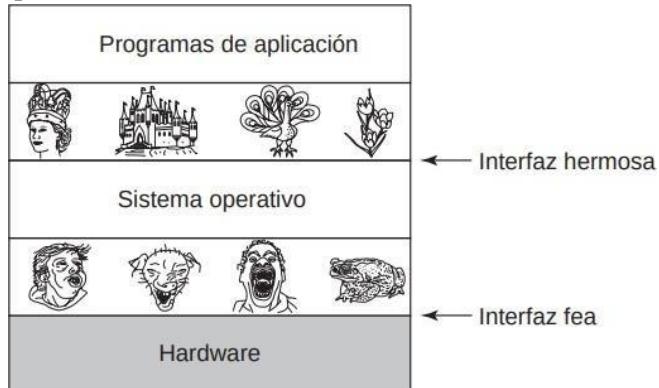


FIGURA 1.2. Ubicación del sistema operativo.

La mayoría de las computadoras tienen dos modos de operación: modo kernel y modo usuario. El sistema operativo es la pieza fundamental del software y se ejecuta en **modo kernel**. En este modo, el sistema operativo tiene acceso completo a todo el hardware y puede ejecutar cualquier instrucción que la máquina sea capaz de ejecutar. El resto del software se ejecuta en **modo usuario**, en el cual sólo un subconjunto de las instrucciones máquina es permitido. En particular, las instrucciones que afectan el control de la máquina o que se encargan de la E/S (entrada/salida) están prohibidas para los programas en modo usuario.

Para obtener servicios del sistema operativo, un programa usuario debe lanzar una **llamada al sistema** (*system call*), la cual se atrapa en el kernel e invoca al sistema operativo. La instrucción TRAP cambia del modo usuario al modo kernel e inicia el sistema operativo. Las computadoras tienen otros traps aparte de la instrucción para ejecutar una llamada al sistema. La mayoría de los demás traps son producidos por el hardware para advertir acerca de una situación excepcional.



**FIGURA 1.3.** Los sistemas operativos ocultan el hardware feo con abstracciones hermosas.

## 1.2. Revisión del hardware

Un sistema operativo está íntimamente relacionado con el hardware de la computadora sobre la que se ejecuta. Para trabajar debe conocer muy bien el hardware, por lo menos en lo que respecta a cómo aparece para el programador.

### 1.2.1. El procesador

El “cerebro” de la computadora es la CPU, que obtiene las instrucciones de la memoria y las ejecuta. El ciclo básico de toda CPU es obtener la primera instrucción de memoria, decodificarla para determinar su tipo y operandos, ejecutarla y después obtener, decodificar y ejecutar las instrucciones subsiguientes. El ciclo se repite hasta que el programa termina. De esta forma se ejecutan los programas.

Cada CPU tiene un conjunto específico de instrucciones que puede ejecutar. Como el acceso a la memoria para obtener una instrucción o palabra de datos requiere mucho más tiempo que ejecutar una instrucción, todas las CPU contienen ciertos registros en su interior para contener las variables clave y los resultados temporales. Debido a esto, el conjunto de instrucciones generalmente contiene instrucciones para cargar una palabra de memoria en un registro y almacenar una palabra de un registro en la memoria. El sistema operativo debe de estar al tanto de todos los registros, ya que cada vez que se detiene un programa en ejecución, debe guardar todos los registros para poder restaurarlos cuando el programa continúe su ejecución.

Además de los registros generales utilizados para contener variables y resultados temporales, la mayoría de las computadoras tienen varios registros especiales que están visibles para el programador. Uno de ellos es el **contador de programa** (*program counter*), el cual contiene la **dirección de memoria de la siguiente instrucción** a obtener. Una vez que se obtiene esa instrucción, el contador de programa se actualiza para apuntar a la siguiente.

Otro registro es el **apuntador de pila** (*stack pointer*), el cual apunta a la parte superior de la **pila** (*stack*) actual en la memoria. La pila contiene un conjunto de valores por cada procedimiento al que se ha entrado pero del que todavía no se ha salido. El conjunto de valores en la pila por procedimiento contiene los parámetros de entrada, las variables locales y las variables temporales que no se mantienen en los registros.

Otro de los registros es **PSW** (*Program Status Word; Palabra de estado del programa*). Este registro contiene los bits de código de condición, que se asignan cada vez que se ejecutan las instrucciones de comparación, la prioridad de la CPU, el modo (usuario o kernel) y varios otros bits de control. Los programas de usuario pueden leer normalmente todo el PSW pero por lo general sólo pueden escribir en algunos de sus campos. El PSW juega un papel importante en las llamadas al sistema y en las operaciones de E/S.

Muchas CPUs modernas cuentan con medios para ejecutar más de una instrucción al mismo tiempo. Por ejemplo, una CPU podría tener unidades separadas de obtención, decodificación y ejecución. A dicha organización se le conoce como **canalización** (*pipeline*); la figura 1.4.(a) ilustra una canalización de tres etapas.

Aún más avanzada que el diseño de una canalización es la CPU **superescalar**, que se muestra en la figura 1.4.(b). En este diseño hay varias unidades de ejecución; por ejemplo, una para la aritmética de enteros, una para la aritmética de punto flotante y otra para las operaciones Booleanas.

A medida que incrementa el número de transistores en los chips surge el problema de qué hacer con todos ellos. Una solución son las arquitecturas superescalares, con múltiples unidades funcionales, pero se puede hacer todavía más. Además de colocar cachés más grandes en el chip de la CPU, ya que en cierto momento se llega al punto de rendimiento decreciente, se multiplican no sólo las unidades funcionales, sino también parte de la lógica de control. Algunos chips de CPU tienen esta propiedad, conocida como **multihilamiento**.

(*multithreading*) o **hiperhilamiento** (*hyperthreading*). Lo que hace es permitir que la CPU contenga el estado de dos hilos de ejecución (*threads*) distintos y luego alterne entre uno y otro con una escala de tiempo en nanosegundos (un hilo de ejecución es algo así como un proceso ligero, que a su vez es un programa en ejecución).

Mas allá del multihilamiento, tenemos chips de CPU con dos, cuatro o más procesadores completos, o **núcleos** (*cores*) en su interior. Los chips de multinúcleo (*multicore*) de la figura 1.5. contienen efectivamente cuatro minichips en su interior, cada uno con su propia CPU independiente. Para hacer uso de dicho chip multinúcleo se requiere en definitiva un sistema operativo multiprocesador.

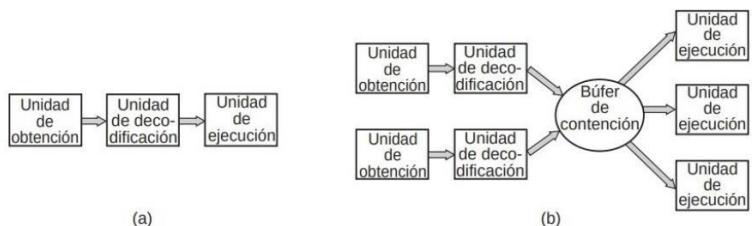


FIGURA 1.4. (a) Canalización de tres etapas; (b) CPU superescalar.

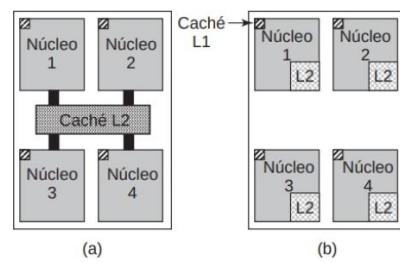


FIGURA 1.5. (a) Un chip de cuatro núcleos (*quad-core*) con una caché L2 compartida.  
(b) Un chip de cuatro núcleos con cachés L2 separadas.

## 1.2.2. La memoria

El segundo componente importante en cualquier computadora es la memoria. En teoría, una memoria debe ser en extremo rápida (más rápida que la velocidad de ejecución de una instrucción, de manera que la memoria no detenga a la CPU), de gran tamaño y muy económica.

Ninguna tecnología en la actualidad cumple con todos estos objetivos, por lo que se adopta una solución distinta. El sistema de memoria está construido como una jerarquía de capas, como se muestra en la figura 1.6. Las capas superiores tienen mayor velocidad, menor capacidad y mayor costo por bit que las capas inferiores, a menudo por factores de mil millones o más.

La capa superior consiste en los registros internos de la CPU, la capacidad de almacenamiento de estos registros suele ser de 64x64 bits.

El siguiente nivel es la **memoria caché**, que el hardware controla de manera parcial. La memoria principal se divide en líneas de caché, que por lo general son de 64 bytes, con direcciones de 0 a 63 en la línea de caché 0, direcciones de 64 a 127 en la línea de caché 1 y así sucesivamente. Las líneas de caché que se utilizan con más frecuencia se mantienen en **una caché de alta velocidad, ubicada dentro o muy cerca de la CPU**. Cuando el programa necesita leer una palabra de memoria, el hardware de la caché comprueba si la línea que se requiere se encuentra en la caché. Si es así (*a lo cual se le conoce como acierto de caché*), la petición de la caché se cumple y no se envía una petición de memoria a través del bus hacia la memoria principal. Los aciertos de caché por lo general requieren un tiempo aproximado de dos ciclos de reloj. Los fallos de caché **tienen que ir a memoria, con un castigo considerable de tiempo**. La memoria caché está limitada en tamaño debido a su alto costo.

La memoria principal viene a continuación en la jerarquía de la figura 1-6. Es el “caballo de batalla” del sistema de memoria. Por lo general a la memoria principal se le conoce como **RAM** (*Random Access Memory, Memoria de Acceso Aleatorio*). Todas las peticiones de la CPU que no se puedan satisfacer desde la caché pasan a la memoria principal.

Además de la memoria principal, muchas computadoras tienen una pequeña cantidad de memoria de acceso aleatorio no volátil. A diferencia de la **RAM**, la memoria no volátil no pierde su contenido cuando se desconecta la energía. La **ROM** (*Read Only Memory, Memoria de sólo lectura*) se programa en la fábrica y no puede modificarse después. Es rápida y económica. En algunas computadoras, el cargador de arranque (bootstrap loader) que se utiliza para iniciar la computadora está contenido en la **ROM**.

## 1.2.3. La memoria secundaria: disco

El siguiente lugar en la jerarquía corresponde al disco magnético (disco duro). El almacenamiento en disco es dos órdenes de magnitud más económico que la RAM por cada bit, y a menudo es dos órdenes de magnitud más grande en tamaño también. El único problema es que el tiempo para acceder en forma aleatoria a los datos en ella es de cerca de tres órdenes de magnitud más lento. Esta baja velocidad se debe al hecho de que un disco es un dispositivo mecánico, como se muestra en la figura 1.7.

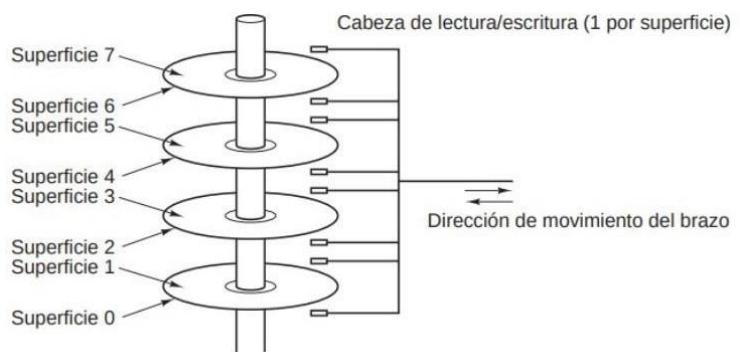
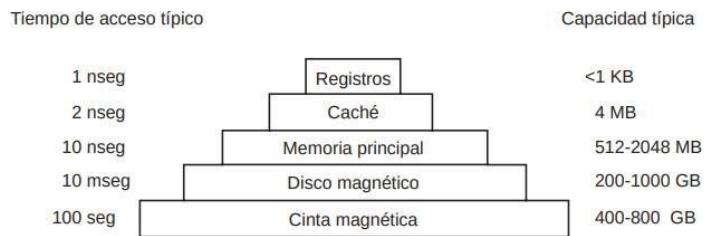


FIGURA 1.7. Estructura de una unidad de disco.

Un disco consiste en uno o más platos que giran a 5400, 7200 o 10,800 rpm. Un **brazo mecánico**, con un punto de giro colocado en una esquina, se mueve sobre los platos de manera similar al brazo de la aguja en un viejo tocadiscos. La información se escribe en el disco en una serie de **círculos concéntricos**. En cualquier posición dada del brazo, cada una de las cabezas puede leer una región anular conocida como **pista** (*track*). En conjunto, todas las pistas para una posición dada del brazo forman un **cilindro** (*cylinder*).

Además el **disco** introduce el concepto de **memoria virtual**, que se puede entender como usar el disco como una extensión de la **RAM**. La memoria virtual nos va a permitir la ejecución de programas más grandes que la memoria física al colocarlos en el disco y usar la **RAM como una especie de caché** para las partes que más se usan. Este esquema requiere la reasignación de direcciones de memoria al instante, para convertir la dirección generada por el programa a la RAM. Esta **traducción** se realiza en el procesador mediante la **MMU**.

La **presencia de la caché y la MMU** pueden tener un gran impacto en el rendimiento. En un sistema de **multiprogramación**, al cambiar de un programa a otro (*lo que se conoce comúnmente como cambio de contexto o context switch*), puede ser necesario vaciar todos los bloques modificados de la caché y modificar los registros de asignación en la **MMU**. Ambas operaciones son costosas y los programadores se esfuerzan bastante por evitarlas. Más adelante veremos algunas de las consecuencias de sus tácticas.

#### 1.2.4. Los dispositivos de E/S

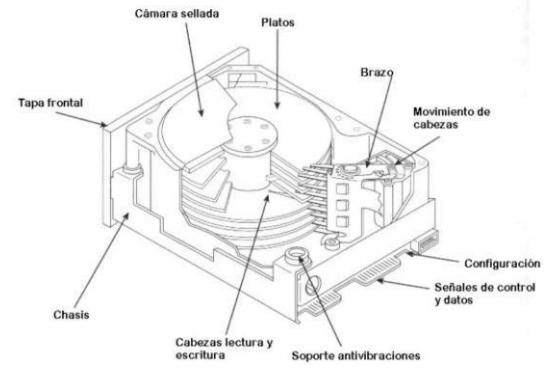
Los dispositivos de E/S también interactúan mucho con el sistema operativo. Como vimos en la figura 1.1., los dispositivos de E/S generalmente constan de dos partes: un **dispositivo controlador** y el **dispositivo en sí**. El dispositivo controlador es un chip o conjunto de chips que controla físicamente el dispositivo. Por ejemplo, acepta los comandos del sistema operativo para leer datos del dispositivo y los lleva a cabo. La otra pieza es el dispositivo en sí.

Como cada tipo de **dispositivo controlador** es distinto, se requiere software diferente para controlar cada uno de ellos. El software que se comunica con un dispositivo controlador, que le proporciona comandos y acepta respuestas, se conoce como **driver** (*controlador*). Cada fabricante de dispositivos controladores tiene que suministrar un driver específico para cada sistema operativo en que pueda funcionar.

Para utilizar el driver, se tiene que **colocar en el sistema operativo** de manera que pueda ejecutarse en modo **kernel**. Hay varias formas en que el driver se pueda colocar en el kernel, pero nos vamos a centrar en la que lo carga de forma dinámica. Esta forma consiste en que el sistema operativo acepte nuevos drivers mientras los ejecuta e instala al instante, sin necesidad de reiniciar el sistema. Los dispositivos **conectables en caliente** (*hot-pluggable*), como los dispositivos USB e IEEE 1394, siempre necesitan drivers que se cargan en forma dinámica.

Todo dispositivo controlador tiene un **número pequeño de registros** que sirven para comunicarse con él. Por ejemplo, un dispositivo controlador de disco con las mínimas características podría tener registros para especificar la dirección de disco, dirección de memoria, número de sectores e instrucción (lectura o escritura). Para activar el dispositivo controlador, el **driver recibe un comando del sistema operativo** y después lo traduce en los valores apropiados para escribirlos en los registros del dispositivo controlador. La colección de todos los registros del dispositivo controlador forma el **espacio de puertos de E/S**.

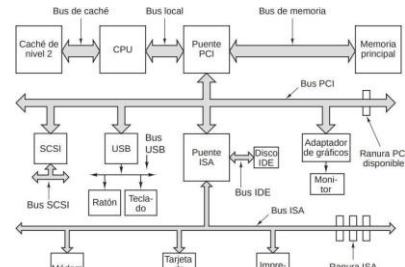
En ciertas computadoras, los registros de dispositivo tienen una correspondencia con el espacio de direcciones del sistema operativo (las direcciones que puede utilizar), de modo que se puedan leer y escribir en ellas como si fuera en palabras de memoria ordinarias. En dichas computadoras no se requieren instrucciones de E/S especiales. En otras computadoras, los registros de dispositivo se colocan en un espacio de puertos de E/S



especial, donde cada registro tiene una dirección de puerto. En estas máquinas hay instrucciones IN y OUT especiales **disponibles en modo kernel** que permiten a los drivers leer y escribir en los registros. El primer esquema elimina la necesidad de instrucciones de E/S especiales, pero utiliza parte del espacio de direcciones. El segundo esquema no utiliza espacio de direcciones, pero requiere instrucciones especiales.

### 1.2.5. El Bus

Consiste en un **conjunto de cables** eléctricos que conecta los distintos dispositivos de la **computadora**. Antes había **un único bus** que conectaba todos los dispositivos, pero eso acabo siendo inmanejable. Ahora hay **8 buses** (caché, local, memoria, PCI, SCSI, USB, IDE e ISA) cada uno con su propia **velocidad de transferencia y función distintas**.



### 1.2.6. El arranque de la computadora

En la placa base hay un programa conocido como **BIOS (Basic Input Output System, Sistema básico de entrada y salida)** del sistema. El BIOS contiene software de E/S de bajo nivel, incluyendo procedimientos para leer el teclado, escribir en la pantalla y realizar operaciones de E/S de disco, entre otras cosas. Hoy en día está contenido en una RAM tipo flash que es no volátil pero el sistema operativo puede actualizarla cuando se encuentran errores en el BIOS.

Al encender la computadora, el **BIOS** ejecuta pruebas de hardware, como verificar la cantidad de RAM instalada y la respuesta de dispositivos básicos como el teclado. Luego, explora los buses ISA y PCI para detectar dispositivos conectados, tanto los heredados como los de tipo **plug and play**, asignando configuraciones cuando sea necesario.

El **BIOS** consulta una lista en la memoria **CMOS** para determinar el dispositivo de arranque. Los dispositivos probados en orden suelen ser el **disco flexible, CD-ROM y el Disco Duro**.

Se carga el **Sistema Operativo**, el BIOS lee el primer sector del dispositivo de arranque y lo coloca en memoria. Este sector contiene un programa que examina la **tabla de particiones** para determinar qué partición está activa. Se lee y ejecuta el **cargador de arranque secundario** de esa partición, que carga el sistema operativo desde la partición activa.

El sistema operativo consulta al **BIOS** para obtener información de configuración y verifica si los **drivers** de los dispositivos están presentes. Si faltan, pide al usuario que inserte un CD-ROM con los drivers correspondientes. Una vez que los drivers están cargados en el **kernel**, el sistema operativo inicializa las tablas necesarias, crea procesos de segundo plano (llamados **demonios**) y arranca un programa de inicio de sesión o **GUI**.

### 1.2.7. Tipos de Sistemas Operativos

#### Sistemas operativos de mainframe

Están orientados hacia el procesamiento de muchos trabajos, sobre todo de muchas operaciones de E/S, a la vez. Se usan en servidores web de alto rendimiento, como en UNIX.

#### Sistemas operativos de servidores

Dan servicio a varios usuarios a la vez y les permiten compartir los recursos de hardware y software. Tenemos Linux, Windows, Solaris.

#### Sistemas operativos de multiprocesadores

Sistemas de varias CPUs. Linux, Windows.

## Sistemas operativos de computadoras personales (PC)

Su trabajo es proporcionar un buen soporte a un **sólo usuario**. Linux, Windows y MacOS.

## Sistemas operativos de computadoras de bolsillo (PDA)

Proporcionan servicios de telefonía, fotografía digital, etc. Android, IOS, Windows Phone.

## Sistemas operativos empotrados (embedded)

Tienen un **conjunto cerrado de aplicaciones** y no se pueden instalar nuevas. Se usan en microondas, reproductores de música, etc. QNX, VxWorks.

## SISTEMAS OPERATIVOS EN TIEMPO REAL

Su parámetro clave es el **tiempo**. Se usan en fábricas.

# 1.3 Conceptos Básicos de los Sistemas Operativos

## 1.3.1. Procesos

Un concepto clave en todos los sistemas operativos es el **proceso**. Un proceso es en esencia un programa en ejecución. Cada proceso tiene asociado un **espacio de direcciones**, una lista de **ubicaciones de memoria** que va desde algún mínimo (generalmente 0) hasta cierto valor máximo, donde el proceso puede leer y escribir información. El espacio de direcciones contiene el programa ejecutable, los datos del programa y su pila. También hay asociado a cada proceso un conjunto de recursos, que comúnmente incluye registros (el contador de programa CP y el apuntador de pila SP, entre ellos), una lista de archivos abiertos, alarmas pendientes, listas de procesos relacionados y toda la demás información necesaria para ejecutar el programa. En esencia, un proceso es un recipiente que guarda toda la información necesaria para ejecutar un programa.

Los procesos tienen su memoria dividida en 3 segmentos (lo de llamar a esto segmento es muy confuso y cuanto más sepáis sobre memoria más os va a confundir esto, sin embargo, pensad que en sistemas con segmentación esto es cierto, pero en sistemas con paginación se pueden interpretar como conjuntos de páginas independientes, si es la primera vez que lees esto no te preocupes, haz como que no lo has leído):

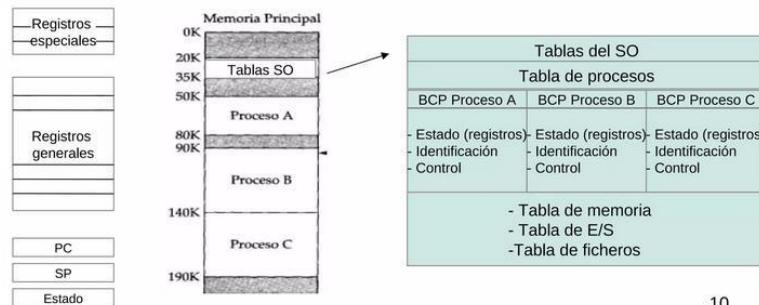
- Segmento de **texto** (*código del programa*).
- Segmento de **datos** (*variables*).
- Segmento de **pila**.

Para darse una buena idea de lo que es un proceso, podemos pensar en un sistema de multiprogramación. Cuando tenemos varios procesos activos, cada cierto tiempo el sistema operativo decide detener la ejecución de un proceso y empezar a ejecutar otro; por ejemplo, debido a que el primero ha utilizado más tiempo del que le correspondía de la CPU en el último segundo. Por lo que la CPU **commuta entre procesos rápidamente**, esto se conoce como un **cambio de contexto**.

Cuando un proceso se suspende temporalmente, toda la información relevante (*como los archivos abiertos y su posición*) se guarda para poder reiniciar el proceso en el mismo estado cuando vuelva a ejecutarse. Esta información se almacena en la **tabla de procesos** del sistema operativo, la cual es un array (*o lista enlazada*) de estructuras, una para cada proceso que se encuentre actualmente en **existencia**. **Esta tabla es global para todo el sistema y está cargada en la RAM**.

La **tabla de procesos** es esencial en un sistema de tiempo compartido porque:

- **Almacena el estado de los procesos:** Contiene información sobre cada proceso, como su ID, estado (ejecutando, listo, bloqueado), registros, contador de programa, y recursos asignados.
- **Facilita la multitarea:** Permite al sistema operativo gestionar varios procesos a la vez, realizando cambios de contexto de manera eficiente.
- **Rastrea recursos:** Ayuda a garantizar que los recursos asignados a cada proceso (como memoria y archivos abiertos) estén correctamente administrados.



10

En un sistema de **multiprogramación**, varios procesos pueden estar activos simultáneamente. El sistema operativo se encarga de **interrumpir un proceso y asignar tiempo de CPU a otro**, gestionando el intercambio entre procesos mediante suspensiones temporales.

Las llamadas al sistema de administración de procesos clave son las que se encargan de la creación y la terminación de los procesos. Considere un ejemplo común. Un proceso llamado **intérprete de comandos** o **shell** lee comandos de una terminal. El usuario acaba de escribir un comando, solicitando la compilación de un programa. El shell debe entonces crear un proceso para ejecutar el compilador. Cuando ese proceso ha terminado la compilación, ejecuta una llamada al sistema para terminarse a sí mismo.

Si un proceso puede crear uno o más procesos aparte (conocidos como **procesos hijos**) y estos procesos a su vez pueden crear procesos hijos, llegamos rápidamente la estructura de árbol de procesos de la figura 1.9. Los procesos relacionados que cooperan para realizar un cierto trabajo a menudo necesitan comunicarse entre sí y sincronizar sus actividades. A esta comunicación se le conoce como **comunicación entre procesos** mediante llamadas.

En algunas ocasiones se tiene la necesidad de transmitir información a un proceso en ejecución que no está esperando esta información. Cuando ha transcurrido el número especificado de segundos, el sistema operativo envía una **señal de alarma** al proceso. La señal provoca que el proceso suspenda en forma temporal lo que esté haciendo, almacene sus registros en la pila y empiece a ejecutar un procedimiento **manejador de señales especial**, por ejemplo, para retransmitir un mensaje que se considera perdido. Cuando termina el manejador de señales, el proceso en ejecución se reinicia en el estado en el que se encontraba justo antes de la señal. Las señales son la analogía en software de las interrupciones de hardware y se pueden generar mediante una variedad de causas además de la expiración de los temporizadores.

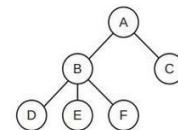


FIGURA 1.9. Un árbol de proceso. El proceso A creó dos procesos hijos, B y C. El proceso B creó tres procesos hijos, D, E y F.

Cada persona autorizada para utilizar un sistema recibe una **UID** (*User Identification*, Identificación de usuario) que el administrador del sistema le asigna. Cada proceso iniciado tiene el UID de la persona que lo inició. Un proceso hijo tiene el mismo UID que su padre. Los usuarios pueden ser miembros de grupos, cada uno de los cuales tiene una **GID** (*Group Identification*, Identificación de grupo). Una UID conocida como **superusuario** (*superuser* en UNIX) tiene poder especial y puede violar muchas de las reglas de protección.

### 1.3.2 Espacio de direcciones

Cada computadora tiene cierta **memoria principal que utiliza para mantener los programas en ejecución**, por lo que se deduce que los espacios de direcciones de los programas se mantienen en esta **memoria principal**. En un sistema operativo muy simple sólo hay un programa a la vez en la memoria. Para ejecutar un segundo programa se tiene que quitar el primero y colocar el segundo en la memoria.

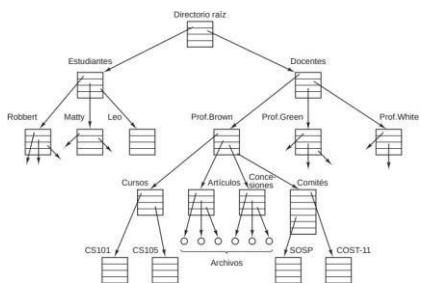
Los sistemas operativos más sofisticados permiten colocar varios programas en memoria al mismo tiempo. Para evitar que **interfieran** unos con otros (y con el sistema operativo), se necesita cierto mecanismo de protección. Aunque este mecanismo tiene que estar en el hardware, es controlado por el sistema operativo.

El problema surge cuando tenemos varios procesos que superan el tamaño de la memoria física. Para manejar esta situación, los sistemas operativos modernos utilizan una técnica llamada **memoria virtual**. Esta técnica permite que un proceso tenga un espacio de **direcciones más grande que la memoria física real disponible**, dividiendo el espacio de direcciones en partes que se mantienen en la memoria principal y en el disco duro. El sistema operativo gestiona el **movimiento de estos fragmentos entre la memoria principal y el disco** según sea necesario.

### 1.3.3 Sistema de archivos

Otro concepto clave de casi todos los sistemas operativos es el sistema de archivos. Sin duda se requieren las llamadas al sistema para crear los archivos, eliminarlos, leer y escribir en ellos. Antes de poder leer un archivo, localizarse en el disco para abrirse y una vez que se ha leído información del archivo debe cerrarse, por lo que se proporcionan llamadas para hacer estas cosas.

Un **directorio** es una forma de agrupar archivos, y puede contener tanto **archivos como otros directorios**, esto da lugar a una **estructura de árbol**. Se necesitan llamadas al sistema para crear y eliminar directorios. También se proporcionan llamadas para poner un archivo existente en un directorio y para eliminar un archivo de un directorio. Las entradas de directorio pueden ser archivos u otros directorios. Este modelo también da surgimiento a una jerarquía (el sistema de archivos) como se muestra en la figura 1.10.



Para especificar cada archivo dentro de la jerarquía de directorio, se proporciona su **nombre de ruta** de la parte superior de la jerarquía de directorios, el **directorio raíz**. Dichos nombres de ruta absolutos consisten de la lista de directorios que deben recorrerse desde el directorio raíz para llegar al archivo, y se utilizan barras diagonales para separar los componentes.

En cada instante, cada proceso tiene un **directorio de trabajo** actual, en el que se buscan los nombres de ruta que no empiecen con una barra diagonal. Como ejemplo, en la figura 1.10. si */Docentes/Prof.Brown* fuera el directorio de trabajo, entonces el uso del nombre de ruta *Cursos/CS101* produciría el mismo archivo que el nombre de ruta absoluto antes proporcionado. Los procesos pueden modificar su directorio de trabajo mediante una llamada al sistema que especifique el nuevo directorio de trabajo.

Otro concepto importante en UNIX es el **sistema de archivos montado**. Consiste en incluir otros **sistemas de archivos externos** a la jerarquía del sistema de archivos raíz del disco duro. En principio, no se puede acceder a los archivos de los sistemas de almacenamiento externos pues no existe manera de especificar sus nombres de ruta.

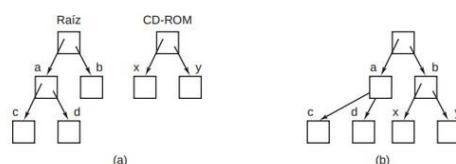


FIGURA 1.11. (a) Antes de montarse, los archivos en el CD-ROM no están accesibles.  
(b) Despues de montarse, forman parte de la jerarquía de archivos.

La llamada al sistema *mount* permite adjuntar el sistema de archivos externo al sistema de archivos raíz en donde el programa desea que esté.

### 1.3.4 Protección

Las computadoras contienen grandes cantidades de información que los usuarios comúnmente desean proteger y mantener de manera confidencial. Es responsabilidad del sistema operativo administrar la seguridad del sistema de manera que los archivos, por ejemplo, sólo sean accesibles para los usuarios autorizados.

Como un ejemplo simple, sólo para tener una idea de cómo puede funcionar la seguridad, considere el sistema operativo UNIX. Los archivos en UNIX están protegidos debido a que cada uno recibe un código de protección binario de 9 bits. El código de protección consiste en tres campos de 3 bits, uno para el propietario, uno para los demás miembros del grupo del propietario (el administrador del sistema divide a los usuarios en grupos) y uno para todos los demás. Cada campo tiene un bit para el acceso de lectura, un bit para el acceso de escritura y un bit para el acceso de ejecución.

Estos 3 bits se conocen como los **bits rwx**. Por ejemplo, el código de protección rwxrx--x indica que el propietario puede leer (**r**), escribir (**w**) o ejecutar (**x**) el archivo, otros miembros del grupo pueden leer o ejecutar (pero no escribir) el archivo y todos los demás pueden ejecutarlo (pero no leer ni escribir). Para un directorio, x indica el permiso de búsqueda. Un guion corto indica que no se tiene el permiso correspondiente.

### 1.3.5 El intérprete de comandos (*Shell*)

El **shell** es el intérprete de comandos de UNIX. Aunque **no forma parte del sistema operativo**, utiliza con frecuencia muchas características del mismo y, por ende, sirve como un buen ejemplo de la forma en que se pueden utilizar las llamadas al sistema. También es la interfaz principal entre un usuario sentado en su terminal y el sistema operativo, a menos que el usuario esté usando una interfaz gráfica de usuario. Existen muchos shells, incluyendo *sh*, *csh*, *ksh* y *bash*.

Cuando cualquier usuario inicia sesión, se inicia un shell. El shell tiene la terminal como entrada y salida estándar. Empieza por escribir el **indicador de comandos (prompt)**, un carácter tal como un signo de dólar, que indica al usuario que el shell está esperando aceptar un comando. El usuario puede especificar que la salida estándar sea redirigida a un archivo, y la salida de un programa se puede utilizar como entrada para otro, si se conectan mediante un canal.

En el shell además podemos realizar las siguientes acciones:

- **Redireccionamiento:** se puede redirigir la entrada y salida estándar con los operandos < y>, respectivamente. Date > archivo
- **Canalización:** la salida de un programa se puede usar como entrada para otro operando | . cat archivo1 archivo2 archivo3 |sort >/dev/lp
- **Comodines:** el shell lo sustituye por todas las posibles combinaciones de caracteres provenientes del directorio en cuestión.

## 1.4 Llamadas al Sistema

Si un programa de usuario en modo usuario necesita un servicio del sistema, como leer del disco ejecuta un **trap** para pasar el SO. Después este averigua que quiere el **proceso llamador**, lleva a cabo la **llamada al sistema** y devuelve el control a la siguiente instrucción. La verdadera mecánica relacioneada con la acción de emitir una llamarada al sistema es muy **dependiente de la máquina** (y a menudo se expresa en **ensamblador**), se proporciona al programado una **biblioteca de procedimientos** para poder realizar llamaradas al sistema desde programas en C y otros lenguajes.

Una **llamada al sistema** (system call) es el mecanismo que permite a los programas en modo usuario solicitar servicios o recursos del sistema operativo en modo kernel.

Ejemplo:

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

La llamarada al sistema (y el procedimiento de biblioteca) devuelve el número de bytes que se leen en cuenta. Si la llamarada al sistema no se puede llevar a cabo, ya sea debido a un parámetro inválido o a un error del disco, *cuenta* se establece a -1 y el número de error se coloca en una variable global llamada *errno*.

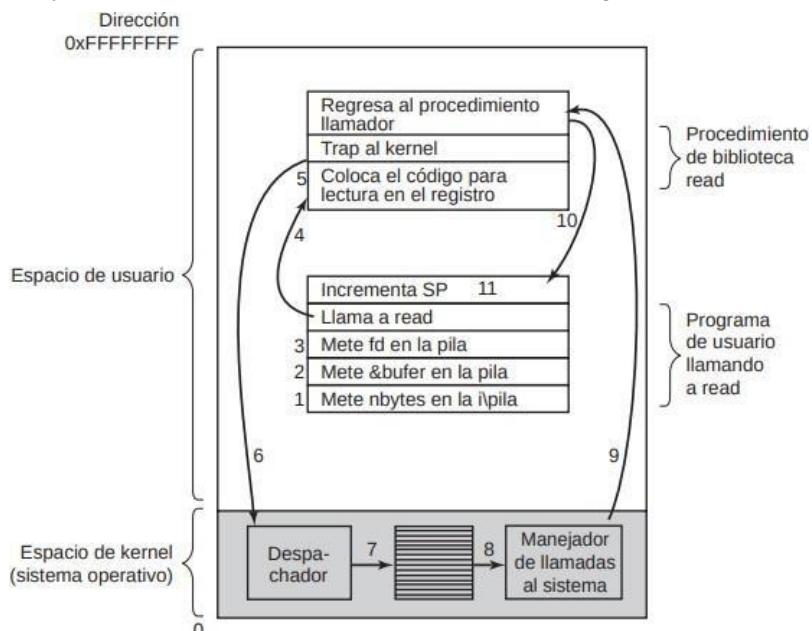


FIGURA 1.12. Los 11 pasos para realizar la llamada al sistema read(fd, bufer, nbytes).

- El programa de usuario **llamador** introduce los datos en la **pila**
- Se llama al procedimiento **read** de la **biblioteca** correspondiente (que seguramente esté en **ensamblador**).
- El código de la librería coloca en un **registro** el número de llamada correspondiente para que el SO pueda determinar qué hacer con ella.
- El procedimiento de biblioteca ejecuta un **trap** para cambiar el modo de usuario a **kernel** y empezar la ejecución en una dirección fija dentro del núcleo.  
**LA INSTRUCCIÓN DE TRAP SIEMPRE SALTA A UNA DIRECCIÓN FIJA.**
- En el **kernel**, el **despachador** analiza el número de la llamada y a través de una **tabla de apuntadores a manejadores de llamadas** indexados en base al número de llamada, se pasa el control al manejador de llamada correspondiente
- Se ejecuta el **manejador de esa llamada**.
- Se devuelve el control (o no) al procedimiento de biblioteca, en la instrucción de después del trap.
- Despues este procedimiento de biblioteca vuelve al programa de usuario.
- El programa de usuario limpia la pila.

## 1.5 Llamadas al sistema para la administración de procesos

### Administración de procesos

Llamada	Descripción
pid = fork()	Crea un proceso hijo, idéntico al padre
pid = waitpid(pid, &statloc, opciones)	Espera a que un hijo termine
s = execve(nombre, argv, entornp)	Reemplaza la imagen del núcleo de un proceso
exit(estado)	Termina la ejecución de un proceso y devuelve el estado

### Administración de archivos

Llamada	Descripción
fd = open(archivo, como, ...)	Abre un archivo para lectura, escritura o ambas
s = close(fd)	Cierra un archivo abierto
n = read(fd, bufer, nbytes)	Lee datos de un archivo y los coloca en un búfer
n = write(fd, bufer, nbytes)	Escribe datos de un búfer a un archivo
posicion = lseek(fd, desplazamiento, dedonde)	Desplaza el apuntador del archivo
s = stat(nombre, &buf)	Obtiene la información de estado de un archivo

### Administración del sistema de directorios y archivos

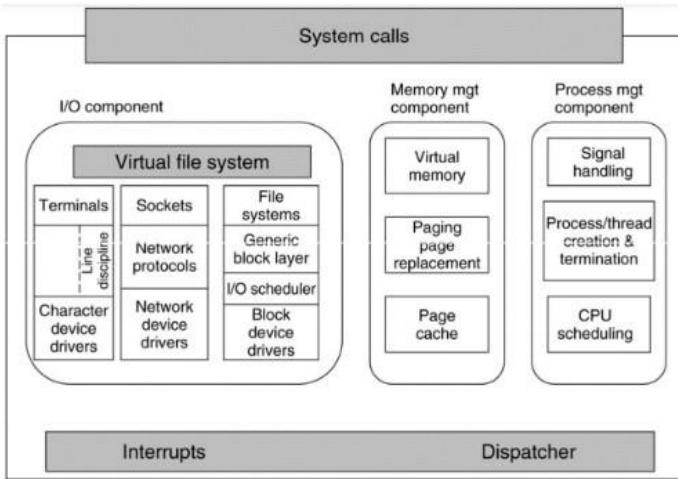
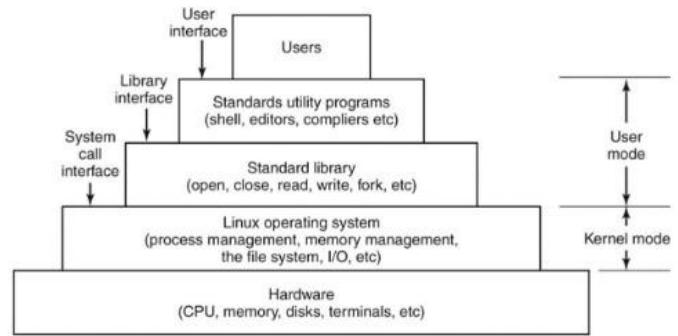
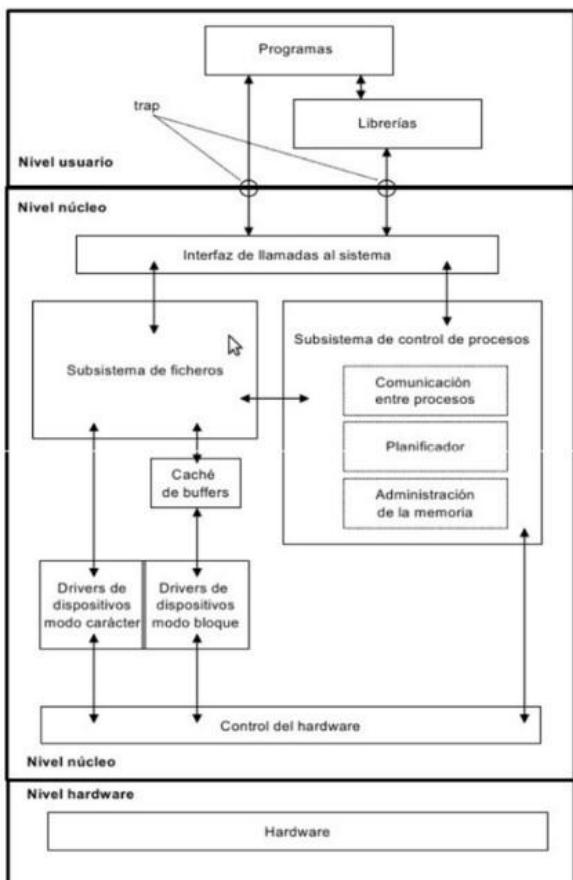
Llamada	Descripción
s = mkdir(nombre, modo)	Crea un nuevo directorio
s = rmdir(nombre)	Elimina un directorio vacío
s = link(nombre1, nombre2)	Crea una nueva entrada llamada nombre2, que apunta a nombre1
s = unlink(nombre)	Elimina una entrada de directorio
s = mount(especial, nombre, bandera)	Monta un sistema de archivos
s = umount(especial)	Desmonta un sistema de archivos

**FIGURA 1.13.** Algunas de las principales llamadas al sistema POSIX. El código de **Llamadas varias**

Llamada	Descripción
s = chdir(nombredir)	Cambia el directorio de trabajo
s = chmod(nombre, modo)	Cambia los bits de protección de un archivo
s = kill(pid, senial)	Envía una señal a un proceso
segundos = tiempo(&segundos)	Obtiene el tiempo transcurrido desde Ene 1, 1970

retorno *s* es -1 si ocurrió un error. Los códigos de retorno son: *pid* es un id de proceso, *fd* es un descriptor de archivo, *n* es una cuenta de bytes, *posicion* es un desplazamiento dentro del *archivo* y *segundos* es el tiempo transcurrido.

## 1.6 Estructura del Sistema Operativo



## 1.7 Cosas importantes de este Tema

La definición de Sistema Operativo.

Saberse el 1.4 perfectamente.

Entender los siguientes conceptos que en el Tanembaum explica por encima y no se terminan de entender (esta información la sacamos tras mandarle un correo al profesor):

- **Trap:** salto para ejecutar algún código del kernel del SO, esto incluye syscalls, interrupciones, excepciones, etc.
- **Llamada al sistema:** es producida por **software** (generada por algunas funciones en C o instrucciones en assembly como las del MIPS). Están definidas en las librerías, que pasan el nº de llamada al SO en modo kernel (por lo que son **Traps**) y se ejecutan en el kernel. Se puede pensar que son como una herramienta que permite al usuario acceder a instrucciones privilegiadas, prácticamente cualquier función de librería de C son syscalls (hay una lista en internet de todas ellas para el que se ralle con eso).
- **Interrupciones:** son producidas por **hardware** externo al procesador (Disco duro, puertos de E/S, etc.). Son asíncronos y se producen en cualquier momento, el controlador de interrupciones las espera pasivamente. Las interrupciones implican un **Trap** porque el dispositivo controlador que las recibe y la **resolución de la interrupción es un Trap**. Por si no queda claro, el trap no se produce cuando por ejemplo presionas una tecla del teclado, se produce cuando el SO gestiona esa interrupción.
- **Excepción:** se producen al intentar ejecutar una instrucción ilegal como una división por 0 o un **fallo de página** (más adelante se explica lo que es porque es muy importante). Una excepción es un **Trap** porque implica pasar a modo kernel. Además sabemos que al ocurrir un fallo de página se debe de traer una página del disco, lo ademas genera una interrupción cuya resolución implica otro Trap.

## Procesos e hilos - Perdiendo la Cordura

El concepto más importante en cualquier sistema operativo es el de **proceso**, una **abstracción** de un programa en ejecución; todo lo demás depende de este concepto. Los procesos son una de las abstracciones más antiguas e importantes que proporcionan los sistemas operativos: proporcionan la capacidad de operar (pseudo) **concurrentemente**, incluso cuando hay sólo una CPU disponible. Convierten una CPU en varias CPUs virtuales.

### 2.1. Concepto de proceso

En cualquier sistema de multiprogramación, la CPU conmuta de un proceso a otro con rapidez, ejecutando cada uno durante décimas o centésimas de milisegundos: hablando en sentido estricto, en cualquier instante la CPU está ejecutando sólo un proceso, y en el transcurso de 1 segundo podría trabajar en varios de ellos, dando la apariencia de un paralelismo (o **pseudoparalelismo**, para distinguirlo del verdadero paralelismo de hardware de los sistemas **multiprocesadores** con dos o más CPUs que comparten la misma memoria física). Es difícil para las personas llevar la cuenta de varias actividades en paralelo; por lo tanto, los diseñadores de sistemas operativos han evolucionado con el paso de los años a un modelo conceptual (procesos secuenciales) que facilita el trabajo con el paralelismo.

En concepto, cada proceso tiene su propia CPU virtual. En realidad, los procesos compiten por una o varias CPUs reales, que conmutan de un proceso a otro con rapidez. Así, aunque en todos los instantes la CPU está ejecutando un único proceso, nos da la apariencia de **pseudoparalelismo**. El algoritmo de **planificación de procesos** determina cuándo se debe detener el trabajo en un proceso para dar servicio a otro y qué proceso será el nuevo.

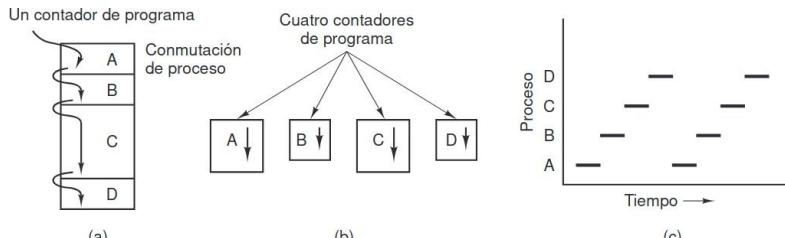
La diferencia clave entre un programa y un proceso es que:

- **Programa:** Es **estático**, un conjunto de instrucciones **almacenadas** (por ejemplo, una receta de cocina).
- **Proceso:** Es **dinámico**, la actividad de ejecutar un programa con entradas y salidas específicas (como seguir la receta para hornear un pastel).

#### 2.1.1. El modelo del proceso

La figura 2.1.(a) muestra una computadora **multiprogramando** cuatro programas en memoria; la figura 2.1(b) lista **cuatro procesos**, **cada uno con su propio flujo de control** (es decir, su propio contador de programa lógico) y cada uno ejecutándose en forma independiente. Desde luego que **sólo hay un contador de programa físico**, por lo que cuando se ejecuta cada proceso, **se carga su contador de programa lógico en el contador de programa real**.

ejecutándose en forma independiente. Desde luego que **sólo hay un contador de programa físico**, por lo que cuando se ejecuta cada proceso, **se carga su contador de programa lógico en el contador de programa real**. Cuando termina (*por el tiempo que tenga asignado*), el contador de programa físico se guarda en el contador de programa lógico almacenado del proceso en memoria. En la figura 2.1(c) podemos ver que durante un intervalo suficientemente largo todos los procesos han progresado, pero en cualquier momento dado sólo hay un proceso en ejecución.



La diferencia entre un proceso y un programa es sutil pero crucial. La idea clave es que un proceso es una actividad de cierto tipo: tiene un programa, una entrada, una salida y un estado. Varios procesos pueden compartir un solo procesador mediante el uso de un algoritmo de planificación para determinar cuándo se debe detener el trabajo en un proceso para dar servicio a otro. En resumen, un proceso es un programa en ejecución o un programa cargado que incluye códigos, datos, pilas, espacio de direcciones, señales, archivos, etc. Un programa, en cambio, es una secuencia de instrucciones y estructuras de datos necesarias para la ejecución, y se almacena en un fichero ejecutable llamado código máquina.

## 2.2. Gestión de procesos

### 2.2.1. Creación de un proceso

Los sistemas operativos necesitan cierta manera de crear procesos. Hay cuatro eventos principales que provocan la creación de un proceso:

1. El arranque del sistema.
2. La ejecución, desde un proceso, de una llamada al sistema para creación de procesos (fork).
3. Una petición del usuario para crear un proceso (abrir la app del navegador).
4. El inicio del trabajo por lotes

Generalmente, cuando se arranca un sistema operativo se crean varios procesos. Algunos de ellos son procesos en primer plano; es decir, procesos que interactúan con los usuarios (humanos) y realizan trabajo para ellos. Otros son procesos en segundo plano, que no están asociados con usuarios específicos sino con una función específica. Los procesos que permanecen en segundo plano para manejar ciertas actividades se conocen como **demonios** (*demons*). Estos procesos se inician mediante una secuencia de comandos de shell cuando se inicia el sistema.

```
//Crear un arbol de procesos de tamaño N
int main(){
    pid_t pid;
    for(int i=0; i<N; i++){
        pid=fork();
    }
}

//Crear un padre con N hijos
int main(){
    int espadre;
    pid_t arr[i];

    for(int i=0; i<N; i++){
        espadre=1;
        for(int j=0; j<N; j++){
            if(!H[j]) espadre=0;
        }

        if(espadre){
            arr[i]=fork();
        }
    }
}

//Crear un padre de n hijo, nieto, bisnieto...
int main(){
    pid_t arr[i];
    int espadre=1;
    for(int i=0; i<n; i++){
        if(espadre){
            arr[i]=fork();
            if(!arr[i]){
                espadre=0;
            }
        }
    }
}
```

En Linux, los procesos se crean de una forma especialmente simple. La **llamada al sistema fork** crea una copia exacta del proceso original. El proceso que va a realizar la bifurcación es el **proceso padre**. Al nuevo proceso se le conoce como **proceso hijo**. El padre y el hijo tienen cada uno sus propias imágenes de memoria privadas. Si el padre cambia después una de sus variables, los cambios no son visibles para el hijo y viceversa. La llamada al sistema fork devuelve un 0 para el hijo y un valor distinto de cero para el padre, el **PID** (*Process Identifier*, Identificador de proceso) del hijo, al padre.

Los procesos padre e hijo tienen la misma **memoria**, las mismas **variables** y los mismos **archivos abiertos**. Sin embargo, tienen **espacios de direcciones** distintos, por lo que si uno de ellos modifica una palabra en su espacio de direcciones, la modificación no es visible para el otro.

**Carrera Crítica:** como el padre y el hijo comparten el mismo puntero en el archivo, si ambos actualizan sobre él al mismo tiempo pueden aparecer errores de lectura o escritura, pues es imposible predecir cuál ejecutará primero el SO.

### 2.2.2. Terminación de procesos

Una vez que se crea un proceso, empieza a ejecutarse y realiza el trabajo al que está destinado. Tarde o temprano el nuevo proceso terminará, por lo general debido a una de las siguientes condiciones:

1. **Salida normal (voluntaria)** (exit(0))

2. **Salida por error** (*voluntaria*)
3. **Error fatal** (*involuntaria*) (segfault)
4. **Eliminado por otro proceso** (*involuntaria*) (kill(Pid))

La mayoría de los procesos terminan debido a que han concluido su trabajo. Cuando un compilador ha compilado el programa que recibe, ejecuta una llamada al sistema para indicar al sistema operativo que ha terminado. Esta llamada es **exit** en UNIX. La segunda razón de terminación es que el proceso descubra un error. Por ejemplo, si un usuario escribe el comando para compilar el programa *foo.c* y no existe dicho archivo, el compilador simplemente termina. La tercera razón de terminación es un error fatal producido por el proceso, a menudo debido a un error en el programa. Algunos ejemplos incluyen el ejecutar una instrucción ilegal, hacer referencia a una parte de memoria no existente o la división entre cero. La cuarta razón por la que un proceso podría terminar es que ejecute una llamada al sistema que indique al sistema operativo que elimine otros procesos. En UNIX esta llamada es **kill** (en caso de que kill no especifique otra señal manda la señal **SIGTERM**, que mata al proceso).

**Exit implícitos:** si un proceso finaliza sin realizar un *exit* el compilador lo incluirá de manera automática. Además, si tenemos una función que devuelve un tipo distinto de void y en algún caso no especificamos en valor de retorno, esa función va a devolver un valor aleatorio por normal general, aunque si probáis con códigos de este estilo dará la salida esperada, aunque no tendría por qué darla.

Ahora vamos a explicar cómo esperar a la terminación de un proceso. Considere el caso de un shell. Lee un comando de la terminal, bifurca un proceso hijo, espera a que el hijo ejecute el comando y después lee el siguiente comando cuando el hijo termina. Para esperar a que el hijo termine, el padre ejecuta una llamada al sistema **waitpid**, que sólo espera a que el hijo termine (cualquier hijo, si existe más de uno). Waitpid tiene tres parámetros. El primero permite al proceso que hizo la llamada esperar a un hijo específico. Si es -1, puede ser cualquier hijo anterior (*es decir, el primer hijo que termine*). El segundo parámetro es la dirección de una variable que se establecerá con el estado de salida del hijo (*terminación normal o anormal y valor de salida*). El tercero determina si el proceso que hizo la llamada se bloquea o regresa si ningún hijo ha terminado.

🔗 **Importante**

**Tipos de procesos:**

- **Proceso Zombi:** proceso que **ya ha terminado su tarea**, es decir, está muerto, pero no ha sido eliminado del sistema, es decir, sigue en la tabla de procesos. Esto se debe a que los procesos solo se borran de la tabla cuando su padre finaliza o hasta que ejecute un **wait** o **waitpid** asociado a él. Todos los procesos al acabar pasan a ser **zombies** hasta que se realice una de las dos acciones que provocan su eliminación de la tabla de procesos
- **Proceso huérfano:** es posible que un proceso padre cree a un hijo y **finalice antes el padre que el hijo**, creando así un hijo **huérfano**. Cuando un huérfano finaliza, **no se convierte en zombi** porque ya no tiene un parente que esté esperando por él, a pesar de haber sido adoptado, por que lo una vez terminado **se elimina directamente de la tabla de procesos**. Los procesos huérfanos son adoptados por el **proceso init**

En el caso del shell, el proceso hijo debe ejecutar el comando escrito por el usuario. Para ello utiliza la llamada al sistema **exec**, la cual hace que la imagen de todo su núcleo se reemplace por el archivo nombrado en su primer parámetro. En la figura 2.4. se muestra un shell muy simplificado, en el que se ilustra el uso de **execve**. Normalmente tras crear los procesos todos ejecutan un cambio de imagen.

**Exec** es la llamada al sistema más compleja. El resto de las llamadas son más simples. Como ejemplo de una llamada simple considere **exit**, que los procesos deben usar al terminar su ejecución. Tiene un parámetro: el estado de salida (0 a 255), el cual se devuelve al parente en la variable **status** de la llamada al sistema **waitpid**. El byte de orden inferior de **status** contiene el estado de terminación, donde 0 es la terminación normal y los

otros valores son diversas condiciones de error. El byte de orden superior contiene el estado de salida del hijo (0 a 255), según lo especificado en la llamada del hijo a exit.

### 2.2.3 Señales

Un proceso puede enviar lo que se conoce como una **señal** a otro proceso. Las señales sirven para identificar a los procesos sobre los **eventos** que ocurren en el sistema. Se identifican mediante una constante simbólica o con un entero. Pueden ser generadas por excepciones, otros procesos (kill), interrupciones de terminal, control de tareas, notificaciones de E/S y alarmas.

```
#include <unistd.h>
#include <stdio.h>

int main() {
    char *argv[] = { "/bin/ls", "-l", NULL };
    char *envp[] = { NULL };

    if (execve("/bin/ls", argv, envp) == -1) {
        perror("execve failed");
    }
    return 0;
}
```

Cada señal tiene asignado un comportamiento por defecto. Esta acción se realiza en el proceso receptor de la señal. Los procesos pueden indicar al sistema lo que quieren que ocurra cuando llegue una señal. Las opciones son ignorarla, atraparla o dejar que la señal elimine el proceso (la opción predeterminada para la mayoría de las señales). Si un proceso elige atrapar las señales que se le envían, debe especificar un procedimiento para el manejo de señales. Las señales también se utilizan para otros fines. Por ejemplo, si un proceso está realizando operaciones aritméticas de punto flotante, y de maneja inadvertida realiza una división entre 0, recibe una señal SIGFPE (excepción de punto flotante).

Las **señales** son un mecanismo de comunicación **asíncrono** utilizado en los sistemas operativos basados en POSIX para notificar a un proceso de eventos del sistema o para sincronización entre procesos. Son esenciales para **manejar interrupciones, errores y acciones específicas del sistema operativo o de otros procesos**. Se parecen a las interrupciones, pero son lanzadas por software y se puede definir qué rutina se ejecuta cuando se reciben

Señal	Descripción
<b>SIGABRT</b>	Solicita que el proceso aborte y genere un núcleo de depuración ( <i>core dump</i> ).
<b>SIGNALRM</b>	Indica que expiró un temporizador configurado por el proceso.
<b>SIGFPE</b>	Error matemático (como división por cero o error de punto flotante).
<b>SIGHUP</b>	Indica que se desconectó la terminal o línea telefónica asociada.
<b>SIGILL</b>	Instrucción ilegal detectada en el proceso.
<b>SIGKILL</b>	Termina el proceso inmediatamente (no se puede capturar ni ignorar).
<b>SIGPIPE</b>	Escrutura en un <i>pipe</i> sin lectores activos.
<b>SIGSEGV</b>	Violación de memoria (acceso a una dirección no válida).
<b>SIGTERM</b>	Solicita la finalización ordenada del proceso.
<b>SIGUSR1</b>	Señal personalizada definida por el usuario.
<b>SIGUSR2</b>	Otra señal personalizada para propósitos definidos por el usuario.

La señal **SIGINT** (*Signal Interrupt*) es una interrupción que se envía a un proceso cuando el usuario presiona **CTRL+C** en la terminal. Esta señal le indica al proceso que debe finalizar de manera controlada. Si no se maneja explícitamente, el sistema termina el proceso de forma predeterminada. La señal **SIGCHLD** se envía a un proceso padre cuando uno de sus procesos hijo cambia de estado. El proceso padre puede usar esta señal para manejar eventos del hijo. Es ignorada por defecto, esto significa que, si un proceso padre no la maneja explícitamente ni espera a los procesos hijos con funciones como wait, los procesos hijos que terminan pueden quedar en estado zombie hasta que se recolecten.

Las señales pueden generarse por diversas razones, como excepciones del sistema, comandos de otros procesos

(como el uso de ***kill***), interrupciones desde la terminal (por ejemplo, al presionar ***CTRL+C***), la finalización de un proceso hijo en el control de tareas, la gestión de cuotas, notificaciones relacionadas con operaciones de E/S (siempre que no sean interrupciones generadas por controladoras hardware), y alarmas programadas. La recepción de estas señales implica que el proceso receptor ejecute la acción correspondiente, incluyendo, en algunos casos, su propia terminación.

Para ello, es esencial que el proceso esté correctamente planificado para responder adecuadamente al evento señalado. Para anunciar que un proceso está dispuesto a atrapar una señal, el proceso puede usar la llamada al sistema ***sigaction***. El primer parámetro es la señal que va a atrapar. El segundo es el apuntador a una estructura que proporciona un apuntador al procedimiento de manejo de señales, así como otros bits y banderas. El tercero apunta a una estructura en la que el sistema desenvuelve la información sobre el manejo de señales que está en efecto, en caso de que se tenga que restaurar después.

```
#include <signal.h>
#include <stdio.h>

// Definimos un manejador (gestor) para las señales
void gestor(int signo) {
    printf("Señal recibida: %d\n", signo);
}

int main() {
    // Declaramos conjuntos de señales para bloquear y verificar señales pendientes
    sigset(SIG_BLOCK, &bloqueo, &pendientes);

    // Estructura para manejar las acciones de las señales
    struct sigaction sa;

    // Configuramos el manejador para las señales
    sa.sa_handler = gestor;           // Asignamos la función 'gestor' como manejador de la señal
    sa.sa_flags = SA_RESTART;         // Reinicia llamadas al sistema interrumpidas por señales
    sigemptyset(&sa.sa_mask);        // Limpia la máscara de señales de esta acción

    // Asignamos la acción a las señales SIGUSR1, SIGUSR2 y SIGINT
    sigaction(SIGUSR1, &sa, NULL);
    sigaction(SIGUSR2, &sa, NULL);
    sigaction(SIGINT, &sa, NULL);

    // Inicializamos el conjunto de señales de bloqueo y añadimos SIGUSR1
    sigemptyset(&bloqueo);
    sigadd(SIG_BLOCK, &bloqueo, SIGUSR1);

    // Bloqueamos las señales del conjunto 'bloqueo' (en este caso, SIGUSR1)
    sigprocmask(SIG_BLOCK, &bloqueo, NULL);

    // Verificamos señales pendientes
    sigpending(&pendientes); // Llenamos 'pendientes' con las señales que están en espera

    // Comprobamos si SIGUSR1 está en el conjunto de señales pendientes
    if (sigismember(&pendientes, SIGUSR1)) {
        printf("SIGUSR1 está pendiente\n");
    }

    // Desbloqueamos SIGUSR1
    sigprocmask(SIG_UNBLOCK, &bloqueo, NULL);

    return 0;
}
```

El manejador de señales se puede ejecutar todo el tiempo que quiera. Sin embargo, en la práctica es común encontrar manejadores de señales muy cortos. Cuando termina el procedimiento de manejo de señales, regresa al punto desde el cual se interrumpió. La llamada al sistema ***sigaction*** también se puede utilizar para hacer que se ignore una señal, o para restaurar la acción predeterminada, que es eliminar el proceso. La llamada al sistema ***kill*** permite que un proceso envíe una señal a otro proceso relacionado. Para muchas aplicaciones en tiempo real, hay que interrumpir un proceso después de un intervalo específico para realizar una acción. Para manejar esta situación se utiliza la llamada al sistema ***alarm***. El parámetro especifica un intervalo en segundos, después del cual se envía una señal ***SIGALARM*** al proceso. Un proceso sólo puede tener una alarma pendiente en cualquier momento.

### 2.3.4. Jerarquías de procesos

Un proceso y todos sus hijos, junto con sus posteriores descendientes, forman un grupo de procesos. Cuando un usuario envía una señal del teclado, esta se envía a todos los miembros del grupo de procesos. De manera individual, cada proceso puede atrapar la señal, ignorarla o tomar la acción predeterminada que es ser eliminado por la señal.

Veamos la forma en que UNIX se inicializa a sí mismo cuando se enciende la computadora. Hay un proceso especial (llamado *init*) en la imagen de inicio. Cuando empieza a ejecutarse, lee un archivo que le indica cuántas terminales hay. Después utiliza fork para crear un proceso por cada terminal. Estos procesos esperan a que alguien inicie la sesión. Si un inicio de sesión tiene éxito, el proceso de inicio de sesión ejecuta un shell para aceptar comandos. Estos pueden iniciar más procesos y así sucesivamente.

En contraste, Windows no tiene un concepto de una jerarquía de procesos. Todos los procesos son iguales. La única sugerencia de jerarquía de procesos es que, cuando se crea un proceso, el padre recibe un indicador especial un *token* (llamado **manejador**) que puede utilizar para controlar al hijo. Sin embargo, tiene la libertad de pasar este indicador a otros procesos, con lo cual invalida la jerarquía.

### 2.3.5. Estados de un proceso

Aunque cada proceso es una entidad independiente, con su propio contador de programa y estado interno, a menudo los procesos necesitan interactuar con otros. Un proceso puede generar cierta salida que otro proceso utiliza como entrada. En el comando de shell

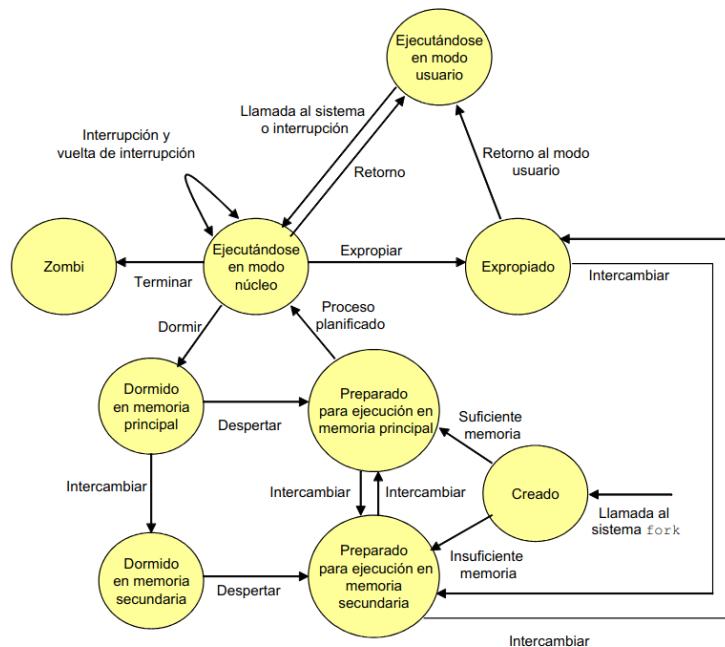
```
cat capítulo1 capítulo2 capítulo3 | grep arbol
```

el primer proceso, que ejecuta cat, concatena tres archivos y los envía como salida. El segundo proceso, que ejecuta grep, selecciona todas las líneas que contengan la palabra “arbol”. Dependiendo de la velocidad relativa de los dos procesos (que dependen tanto de la complejidad relativa de los programas, como de cuánto tiempo ha tenido cada uno la CPU), puede ocurrir que grep esté listo para ejecutarse, pero que no haya una entrada esperándolo. Entonces debe bloquear hasta que haya una entrada disponible.

Cuando un proceso se bloquea, lo hace debido a que por lógica no puede continuar, comúnmente porque está esperando una entrada que todavía no está disponible. También es posible que un proceso, que esté listo en concepto y pueda ejecutarse, se detenga debido a que el sistema operativo ha decidido asignar la CPU a otro proceso por cierto tiempo.



Si utilizamos el modelo de los procesos, es mucho más fácil pensar en lo que está ocurriendo dentro del sistema. Algunos de los procesos ejecutan programas que llevan a cabo los comandos que escribe un usuario; otros son parte del sistema y se encargan de tareas como cumplir con las peticiones de los servicios de archivos o administrar los detalles de ejecutar una unidad de disco. Cuando ocurre una interrupción de disco, el sistema toma una decisión para dejar de ejecutar el proceso actual y ejecutar el proceso de disco que está bloqueado esperando esta interrupción. Así, en vez de pensar en las interrupciones, podemos pensar en los procesos de usuario, procesos de disco, procesos de terminal, etc., que se bloquean cuando están esperando a que algo ocurra. Cuando se ha leído el disco o se ha escrito el carácter, el proceso que espera se desbloquea y es elegible para continuar ejecutándose. El nivel más bajo del sistema operativo es el planificador, con una variedad de procesos encima de él. Todo el manejo de las interrupciones y los detalles relacionados con iniciar y detener los procesos se ocultan en lo que aquí se denomina planificador.



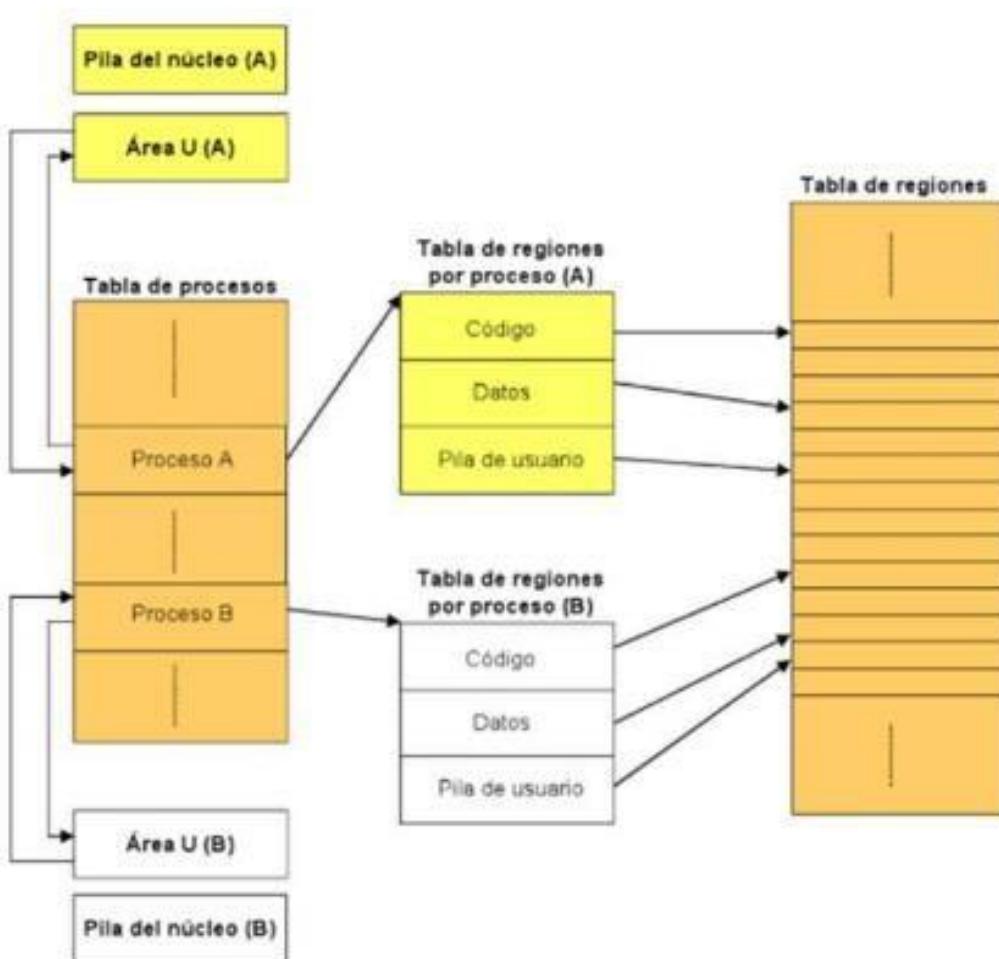
#### Motivos para la suspensión de un proceso:

<b>Swapping</b>	El sistema operativo necesita liberar suficiente memoria principal para traer un proceso en estado Listo de ejecución.
<b>Otras razones del sistema operativo</b>	El sistema operativo puede suspender un proceso en segundo plano o de utilidad o un proceso que se sospecha puede causar algún problema.
<b>Solicitud interactiva del usuario</b>	Un usuario puede desear suspender la ejecución de un programa con motivo de su depuración o porque está utilizando un recurso.
<b>Temporización</b>	Un proceso puede ejecutarse periódicamente (por ejemplo, un proceso monitor de estadísticas sobre el sistema) y puede suspenderse mientras espera el siguiente intervalo de ejecución.
<b>Solicitud del proceso padre</b>	Un proceso padre puede querer suspender la ejecución de un descendiente para examinar o modificar dicho proceso suspendido, o para coordinar la actividad de varios procesos descendientes.

### 2.3.6. Implementación de los procesos

Para implementar el modelo de procesos, el sistema operativo mantiene una tabla llamada **tabla de procesos**, con sólo una entrada por cada proceso. Esta entrada contiene información importante acerca del estado del proceso, incluyendo su contador de programa, apuntador de pila, asignación de memoria, estado de sus archivos abiertos, información de contabilidad y planificación, y todo lo demás que debe guardarse acerca del proceso cuando éste cambia del estado en *ejecución* a *listo* o *bloqueado*, de manera que se pueda reiniciar posteriormente como si nunca se hubiera detenido.

Administración de procesos	Administración de memoria	Administración de archivos
Registros Contador del programa Palabra de estado del programa Apuntador de la pila Estado del proceso Prioridad Parámetros de planificación ID del proceso Proceso padre Grupo de procesos Señales Tiempo de inicio del proceso Tiempo utilizado de la CPU Tiempo de la CPU utilizado por el hijo Hora de la siguiente alarma	Apuntador a la información del segmento de texto Apuntador a la información del segmento de datos Apuntador a la información del segmento de pila	Directorio raíz Directorio de trabajo Descripciones de archivos ID de usuario ID de grupo

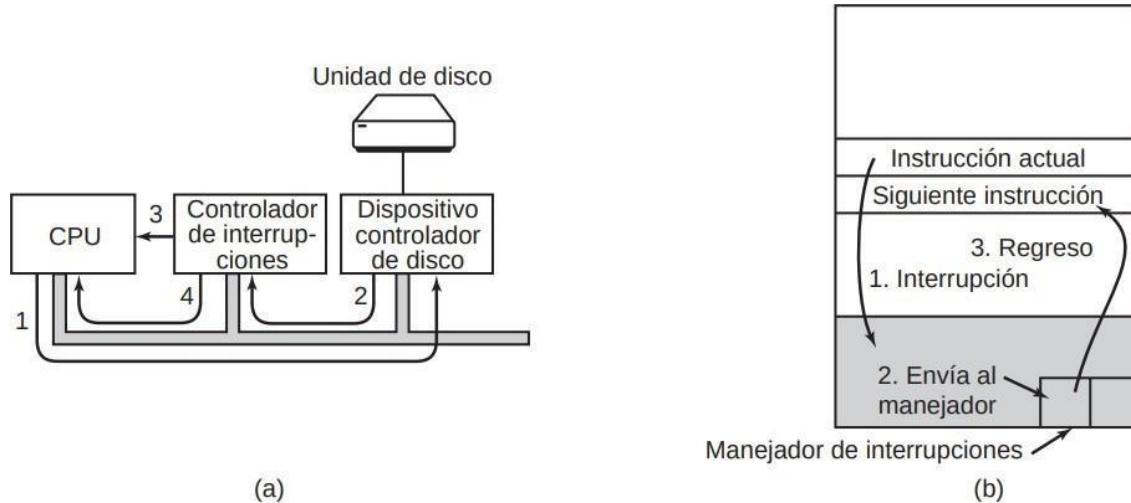


- **Pila del núcleo**: almacena funciones o rutinas invocadas durante la ejecución de un proceso en modo núcleo.
- **Área U**: almacena información de control necesaria para que el kernel gestione el proceso mientras se está ejecutando. Contiene un puntero a la entrada a la entrada del proceso en la tabla de procesos.
- **Tabla de regiones**: almacena una entrada para cada región (código, datos y pila de usuario) asignada a algún proceso

### 2.3.7 Introducción a las Interrupciones

Ahora que hemos analizado la tabla de procesos, es posible explicar un poco más acerca de cómo la ilusión de varios procesos secuenciales se mantiene en una (o en varias) CPU.

Cada clase de operación de E/S tiene una entrada en el **VECTOR DE INTERRUPCIÓN** del sistema, que contiene la dirección del procedimiento de interrupción para esa clase. El procedimiento de interrupción de un tipo de operación de E/S sirve para manejar las interrupciones de dicha clase.



**FIGURA 2.11.** (a) Los pasos para iniciar un dispositivo de E/S y obtener una interrupción.  
(b) El procesamiento de interrupciones involucra tomar la interrupción, ejecutar el manejador de interrupciones y regresar al programa de usuario.

- Se recibe una interrupción desde una **controladora E/S** (o desde un reloj)
- El hardware guarda en la **pila del proceso en ejecución** actual ciertos registros como el **PC**, el **PSW**, etc.
- El hardware carga como nuevo PC la **dirección especificada en la entrada adecuada del vector de interrupción** para ejecutar el procedimiento de interrupción.
  - o Rutina 1 (en ensamblador): igual para todas las interrupciones de todas las clases.
    - o Guarda los registros en la entrada de la tabla de procesos del proceso en ejecución actual
    - o Establece una nueva pila temporal, que usará el procedimiento de interrupción.
  - o Rutina 2 (en C): específica para cada interrupción.
- El planificador decide qué proceso va a ejecutar a continuación.
- El control se devuelve al código ensamblador para cargar los registros y el mapa de memoria del proceso seleccionado
- Se ejecuta el proceso seleccionado.

### 2.3.7 Modelación de la multiprogramación

Cuando se utiliza la multiprogramación, el uso de la CPU se puede mejorar. Dicho en forma cruda: si el proceso promedio realiza cálculos sólo 20 por ciento del tiempo que está en la memoria, con cinco procesos en memoria a la vez la CPU deberá estar ocupada todo el tiempo. Sin embargo, es-

te modelo es demasiado optimista, ya que supone que los cinco procesos nunca estarán esperando la E/S al mismo tiempo.

Un mejor modelo es analizar el uso de la CPU desde un punto de vista probabilístico. Suponga que un proceso gasta una fracción  $p$  de su tiempo esperando a que se complete una operación de E/S. Con  $n$  procesos en memoria a la vez, la probabilidad de que todos los  $n$  procesos estén esperando la E/S (en cuyo caso, la CPU estará inactiva) es  $p^n$ . Entonces, el uso de la CPU se obtiene mediante la fórmula:

$$\text{Uso de la CPU} = 1 - p^n$$

Dónde  $n$  es el número de procesos y  $p$  es la fracción de tiempo esperando a completar una operación de E/S.

## 2.4. Hilos

En los sistemas operativos tradicionales, cada proceso tiene un espacio de direcciones y un solo hilo de control. De hecho, ésa es casi la definición de un proceso. Sin embargo, con frecuencia hay situaciones en las que es conveniente tener varios hilos de control en el mismo espacio de direcciones que se ejecuta en cuasi-paralelo, como si fueran procesos (casi) separados (excepto por el espacio de direcciones compartido).

### 2.4.1. Uso de hilos

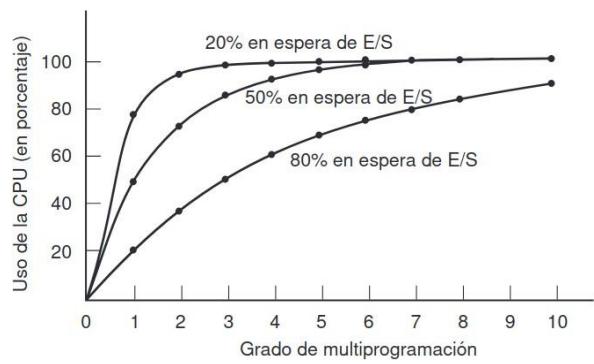
Hay varias razones de tener estos miniprocesos, conocidos como **hilos**. La principal razón de tener hilos es que en muchas aplicaciones se desarrollan varias actividades a la vez. Algunas de éas se pueden bloquear de vez en cuando. Al descomponer una aplicación en varios hilos secuenciales que se ejecutan en cuasi-paralelo, el modelo de programación se simplifica.

Esta precisamente la justificación de tener procesos. En vez de pensar en interrupciones, temporizadores y conmutaciones de contexto, podemos pensar en procesos paralelos. Sólo que ahora con los hilos agregamos un nuevo elemento: la habilidad de las entidades en paralelo de compartir un espacio de direcciones y todos sus datos entre ellas. Esta habilidad es esencial para ciertas aplicaciones, razón por la cual no funcionará el tener varios procesos (con sus espacios de direcciones separados).

Un segundo argumento para tener hilos es que, como son más ligeros que los procesos, son más fáciles de crear (es decir, rápidos) y destruir. En muchos sistemas, la creación de un hilo es de 10 a 100 veces más rápida que la de un proceso. Cuando el número de hilos necesarios cambia de manera dinámica y rápida, es útil tener esta propiedad.

Una tercera razón de tener hilos es también un argumento relacionado con el rendimiento. Los hilos no producen un aumento en el rendimiento cuando todos ellos están ligados a la CPU, pero cuando hay una cantidad considerable de cálculos y operaciones de E/S, al tener hilos estas actividades se pueden traslapar, con lo cual se agiliza la velocidad de la aplicación.

Por último, los hilos son útiles en los sistemas con varias CPUs, en donde es posible el verdadero paralelismo. Es más fácil ver por qué los hilos son útiles si utilizamos ejemplos concretos. Como primer ejemplo considere un procesador de palabras.

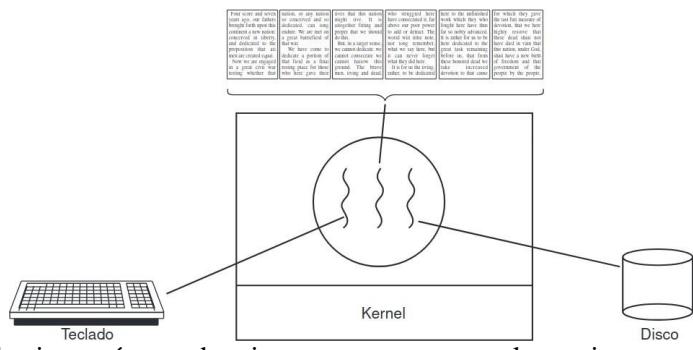


Por lo general, los procesadores de palabras muestran el documento que se va crear en la pantalla exactamente como aparecerá en la página impresa. En especial, todos los saltos de línea y de página están en sus posiciones correctas y finales, de manera que el usuario pueda inspeccionarlas y cambiar el documento si es necesario.

Suponga que el usuario está escribiendo un libro. Desde el punto de vista del autor, es más fácil mantener todo el libro en un solo archivo para facilitar la búsqueda de temas, realizar sustituciones globales, etc. También, cada capítulo podría estar en un archivo separado; sin embargo, tener cada sección y subsección como un archivo separado puede ser una verdadera molestia si hay que realizar cambios globales en todo el libro, ya que entonces tendrían que editarse cientos de archivos en forma individual.

Ahora considere lo que ocurre cuando el usuario repentinamente elimina un enunciado de la página 1 de un documento de 800 páginas. Después de revisar que la página modificada esté correcta, el usuario desea realizar otro cambio en la página 600 y escribe un comando que indica al procesador de palabras que vaya a esa página. Entonces, el procesador de palabras tiene que volver a dar formato a todo el libro hasta la página 600 en ese momento, debido a que no sabe cuál será la primera línea de la página 600 sino hasta que haya procesado las demás páginas. Puede haber un retraso considerable antes de que pueda mostrar la página 600 y el usuario estaría descontento.

Aquí pueden ayudar los hilos. Suponga que el procesador de palabras se escribe como un programa con dos hilos. Un hilo interactúa con el usuario y el otro se encarga de volver a dar formato en segundo plano. El proceso de volver a dar formato se completará antes de que el usuario pida ver la página 600, para que pueda mostrarse al instante Ahora agregamos un tercer hilo. Muchos procesadores de palabras tienen la característica de guardar de manera automática todo el archivo en el disco cada cierto número de minutos, para proteger al usuario contra la pérdida de todo un día de trabajo en caso de un fallo en el programa, el sistema o la energía. El tercer hilo se puede encargar de los respaldos en disco sin interferir con los otros dos.



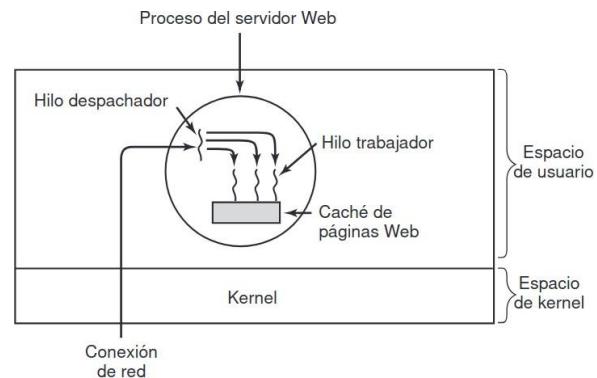
Si el programa tuviera sólo un hilo, entonces cada vez que iniciara un respaldo en el disco se ignorarían los comandos del teclado y el ratón hasta que se terminara el respaldo. El usuario sin duda consideraría esto como un rendimiento pobre. De manera alternativa, los eventos de teclado y ratón podrían interrumpir el respaldo en disco, permitiendo un buen rendimiento, pero produciendo un modelo de programación complejo, controlado por interrupciones. Con tres hilos, el modelo de programación es mucho más simple. El primer hilo interactúa sólo con el usuario, el segundo proceso vuelve a dar formato al documento cuando se le indica y el tercero escribe el contenido de la RAM al disco en forma periódica. Debemos aclarar que aquí no funcionaría tener tres procesos separados, ya que los tres hilos necesitan operar en el documento. Al tener tres hilos en vez de tres procesos, comparten una memoria común y por ende todos tienen acceso al documento que se está editando.

Ahora considere otro ejemplo más de la utilidad de los hilos: un servidor para un sitio en World Wide Web. Las solicitudes de páginas llegan y la página solicitada se envía de vuelta al cliente. En la mayoría de los sitios Web, algunas páginas se visitan con más frecuencia que otras. Considere la forma en que podría escribirse el servidor Web sin hilos. El resultado neto es que se pueden procesar menos peticiones/segundo.

Por ende, los hilos obtienen un aumento considerable en el rendimiento, pero cada hilo se programa de manera secuencial, en forma usual.

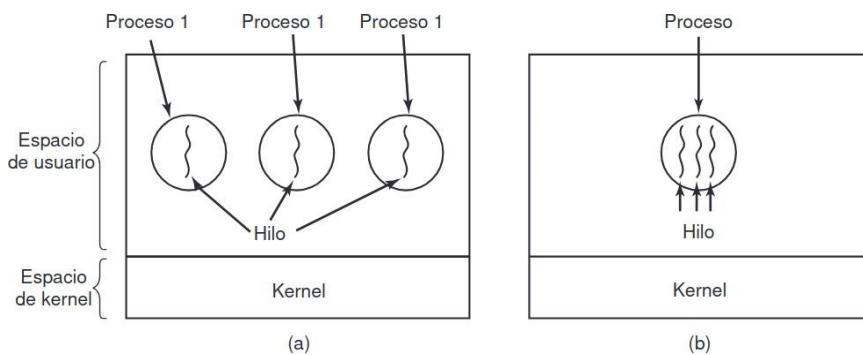
## 2.4.2. El modelo clásico de hilo

El modelo de procesos se basa en dos conceptos independientes: agrupamiento de recursos y ejecución. Una manera de ver a un proceso es como si fuera una forma de agrupar recursos relacionados. Un proceso tiene un espacio de direcciones que contiene texto y datos del programa, así como otros recursos. Estos pueden incluir archivos abiertos, procesos hijos, alarmas pendientes, manejadores de señales, información contable y mucho más. Al reunirlos en forma de un proceso, pueden administrarse con más facilidad.



El otro concepto que tiene un proceso es un **hilo de ejecución**. El hilo tiene un contador de programa que lleva el registro de cuál instrucción se va a ejecutar a continuación. Tiene registros que contienen sus variables de trabajo actuales. Tiene una pila, que contiene el historial de ejecución, con un conjunto de valores para cada procedimiento al que se haya llamado, pero del cual no se haya devuelto todavía. Los procesos se utilizan para agrupar los recursos; son las entidades planificadas para su ejecución en la CPU.

Lo que agregan los hilos al modelo de procesos es permitir que se lleven a cabo varias ejecuciones en el **mismo entorno del proceso**, que son en gran parte independientes unas de las otras. Tener varios procesos ejecutándose en paralelo en un proceso es algo similar a tener varios procesos ejecutándose en paralelo en una computadora. En el primer caso, los hilos **comparten un espacio de direcciones y otros recursos**; en el segundo, los procesos comparten la memoria física, los discos, las impresoras y otros recursos. Como los hilos tienen algunas de las propiedades de los procesos, algunas veces se les llama **procesos ligeros**.



El término **multihilamiento** también se utiliza para describir la situación de permitir varios hilos en el mismo proceso. Cuando se ejecuta un proceso con multihilamiento en un sistema con una CPU, los hilos toman turnos para ejecutarse. La CPU conmuta rápidamente entre un hilo y otro, dando la ilusión de que los hilos se ejecutan en paralelo, aunque en una CPU más lenta que la verdadera.

Los distintos hilos en un proceso **no son tan independientes como los procesos**. Todos los hilos tienen el mismo espacio de direcciones, lo cual significa que también comparten las mismas variables globales. Como cada hilo puede acceder a cada dirección de memoria dentro del espacio de direcciones del proceso, un hilo puede leer, escribir o incluso borrar la pila de otro hilo. No hay protección entre los hilos debido a que (1) es imposible y (2) no debe ser necesario. A diferencia de tener procesos diferentes, que pueden ser de distintos usuarios y hostiles entre sí, un proceso siempre es propiedad de un solo usuario, quien se supone que ha creado varios hilos para que puedan cooperar, no pelear. Además de compartir un espacio de direcciones, todos los hilos pueden compartir el mismo conjunto de archivos abiertos, procesos hijos, alarmas y señales, etc.

Es importante tener en cuenta que cada hilo tiene su propia pila. La pila de cada hilo contiene un conjunto de valores para cada procedimiento llamado, pero del que todavía no se ha regresado. Este conjunto de valores contiene las variables locales del procedimiento y la dirección de retorno que se debe utilizar cuando haya terminado la llamada al procedimiento.

Aunque obviamente si un hilo conoce la dirección de la pila de otro hilo la puede modificar.

Importante

**VARIABLES:**

- **Accesibles por todos los hilos:**
  - Variables globales
  - Archivos abiertos
  - Memoria dinámica (heap)
  - Otros recursos compartidos del sistema (como señales)
- **Privadas a cada hilo:**
  - Variables locales
  - Pila del hilo (incluyendo el contexto de ejecución y las variables locales de las funciones)

Al igual que un proceso tradicional (es decir, un proceso con sólo un hilo), un hilo puede estar en uno de varios estados: en ejecución, bloqueado, listo o terminado. Un hilo en ejecución tiene la CPU en un momento dado y está activo. Un hilo bloqueado está esperando a que cierto evento lo desbloquee. Un hilo listo se programa para ejecutarse y lo hará tan pronto como sea su turno.

Cuando hay multihilamiento, por lo general los procesos empiezan con un solo hilo presente. Este hilo tiene la habilidad de crear hilos mediante la llamada a un procedimiento de biblioteca, como *pthread\_create*. Comúnmente, un parámetro para *pthread\_create* especifica el nombre de un procedimiento para que se ejecute el nuevo hilo. El hilo creador generalmente recibe un identificador de hilo que da nombre al nuevo hilo.

Cuando un hilo termina su trabajo, puede salir mediante la llamada a un procedimiento de biblioteca, como *pthread\_exit*. Después desaparece y ya no puede planificarse para volver a ejecutarse. En algunos sistemas con hilos, un hilo puede esperar a que un hilo (específico) termine mediante a llamada a un procedimiento, por ejemplo, *pthread\_join*. Este procedimiento bloquea al hilo llamador hasta que un hilo (específico) haya terminado. Otra llamada de hilos común es *pthread\_yield*, que permite a un hilo entregar voluntariamente la CPU para dejar que otro hilo se ejecute. **Cuando se llama a *pthread\_yield* el hilo pasa de estar en estado de ejecución a estado listo, es decir, NO SE BLOQUEA y será planificable nada más se llame a la función.**

### 2.4.3. Hilos en POSIX

Llamada de hilo	Descripción
<i>pthread_create</i>	Crea un nuevo hilo
<i>pthread_exit</i>	Termina el hilo llamador
<i>pthread_join</i>	Espera a que un hilo específico termine
<i>pthread_yield</i>	Libera la CPU para dejar que otro hilo se ejecute
<i>pthread_attr_init</i>	Crea e inicializa la estructura de atributos de un hilo
<i>pthread_attr_destroy</i>	Elimina la estructura de atributos de un hilo

```

int variableGlobal = 45;

void *funcionHilo(void *arg) {
    int n = *(int *)arg;
    printf("El hilo recibe la variable local: %d\n", n);
    n = 145234;
    printf("Nuevo valor de la variable local cambiada por el hilo: %d\n", n);

    printf("Valor de la variable global en el hilo: %d\n", variableGlobal);
    variableGlobal = 2384378;
    printf("Nuevo valor de la variable global en el hilo: %d\n", variableGlobal);

    pthread_exit(NULL);
}

int main() {
    pthread_t hilo;
    int variableLocal = 34;

    printf("Valor de la variable local en el hilo principal: %d\n", variableLocal);
    printf("Valor de la variable global en el hilo principal: %d\n", variableGlobal);

    pthread_create(&hilo, NULL, funcionHilo, &variableLocal);
    pthread_join(hilo, NULL); // Espera a que el hilo termine

    printf("Valor de la variable local en el hilo principal después del hilo: %d\n", variableLocal);
    printf("Valor de la variable global en el hilo principal después del hilo: %d\n", variableGlobal);

    return 0;
}

```

## 2.4.4. Implementación de hilos en el espacio de usuario

Hay dos formas principales de implementar un paquete de hilos: en espacio de usuario y en el kernel. Cribaremos estos métodos, junto con sus ventajas y desventajas. El primer método es colocar el paquete de hilos **completamente en espacio de usuario**. El **kernel no sabe nada acerca de ellos**. En lo que al kernel concierne, está administrando procesos ordinarios con un solo hilo. La primera ventaja, la más obvia, es que un paquete de hilos de nivel usuario **puede implementarse en un sistema operativo que no acepte hilos**. Con este método, los hilos se implementan mediante una biblioteca. Cuando los hilos se administran en espacio de usuario, cada proceso necesita **su propia tabla de hilos privada** para llevar la cuenta de los hilos en ese proceso.

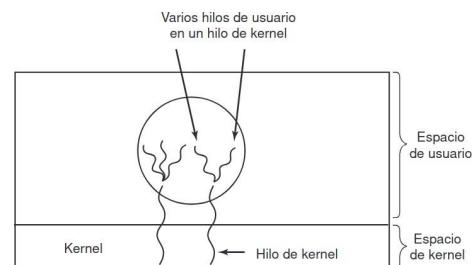
Los hilos se ejecutan encima de un sistema en tiempo de ejecución, el cual es una colección de procedimientos que administran hilos (biblioteca de procedimientos).

Cuando un hilo hace algo que puede ponerlo en estado bloqueado en forma local llama a un procedimiento del sistema en **tiempo de ejecución**. Para **evitar** que el hilo bloquee a todo un proceso de usar la **ENVOLTURA**, que es un procedimiento que se llama en tiempo de ejecución cuando un hilo pretende hacer algo que lo puede bloquear.

- El procedimiento comprueba si el hilo se va a bloquear.
- De ser así, cede la CPU a otro hilo.
- Se almacenan los registros del hilo actual en la tabla de hilos del proceso.
- Se busca en la tabla un hilo listo para ejecutarse (será un hilo **del mismo proceso**).
- Se cargan los registros con los valores del nuevo hilo.
- Se ejecuta el nuevo hilo.

Hacer esta **commutación de hilos** de esta manera es **mucho más veloz** que la commutación de procesos que se tendría que hacer si se bloquease el proceso entero.

Los hilos de nivel usuario también tienen otras ventajas. Las **funciones de manejo** de hilos y el **planificador** de hilos son procedimientos del espacio de usuario, por lo que invocarlos es mucho más rápido que invocar el **trap** al kernel que se usa en la implementación en el núcleo (*el trap a kernel es más costoso pues implica realizar un cambio de contexto, vaciar la cache, etc*). Permiten que cada proceso tenga su propio **algoritmo de planificación personalizado**. Tienen una mejor **escalabilidad** que el kernel, ya que la tabla de hilos central puede llegar a ocupar mucho espacio cuando hay muchos procesos con muchos hilos.



Sin embargo, no se puede impedir que un **fallo de página** (que es un bloqueo impredecible) en un hilo bloquee a todo su proceso, pues no se puede programar una envoltura. Los hilos **no cederán CPU automáticamente** nunca pues dentro de un mismo proceso no hay interrupciones de reloj. Por tanto, cuando se comienza a ejecutar un hilo, no se podrá ejecutar ningún otro hasta que esta ceda la CPU voluntariamente.

#### 2.4.5. Implementación de hilos en el kernel

Ahora vamos a considerar el caso en que el **kernel sabe acerca de los hilos y los administra**. No se necesita un **sistema en tiempo de ejecución para ninguna de las dos acciones**. Además, no hay tabla de hilos en cada proceso. En vez de ello, el **kernel tiene una única tabla de hilos** que lleva la cuenta de todos los hilos en el sistema. Cuando un hilo desea crear un nuevo hilo o destruir uno existente, realiza una **llamada al kernel**, la cual se encarga de la creación o destrucción mediante una actualización en la tabla de hilos del kernel. No es necesario crear **envolturas**.

La tabla de hilos del kernel contiene los registros, el estado y demás información de cada hilo. Esta información es la misma que con los hilos de nivel usuario, pero ahora se mantiene en el kernel, en vez de hacerlo en espacio de usuario (*dentro del sistema en tiempo de ejecución*). Todas las llamadas que podrían gestionar un hilo se implementan como llamadas al sistema, a un costo considerablemente mayor que una llamada a un procedimiento del sistema en tiempo de ejecución.

Cuando un hilo se bloquea (*se encuentra con un fallo de página*), el kernel, según lo que decida, **puede ejecutar otro hilo del mismo proceso** (*si hay uno listo*) o **un hilo de un proceso distinto**. Con los hilos de nivel usuario, el sistema en tiempo de ejecución ejecuta hilos de su propio proceso hasta que el kernel le quita la CPU (*o cuando ya no hay hilos para ejecutar*). Debido al costo considerablemente mayor de crear y destruir hilos en el kernel, algunos sistemas optan por un método ambientalmente correcto, reciclando sus hilos.

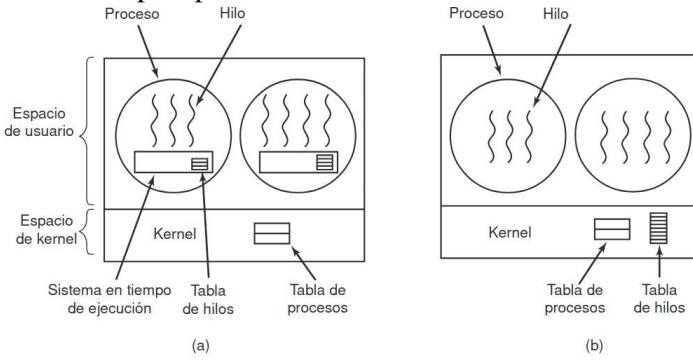


Figura 2-16. (a) Un paquete de hilos de nivel usuario. (b) Un paquete de hilos administrado por el kernel.

Las **funciones de manejo** de hilos y el **planificador** son llamadas al sistema, así que su invocación será más lenta que en la implementación en espacio de usuario. Un trap al kernel es más costoso pues implica relajizar un cambio de contexto, vaciar la caché, etc.

Todos los hilos usarán el **mismo planificador**. Tienen una **mala escalabilidad**, ya que la tabla de hilos central puede llegar a ocupar mucho espacio cuando hay muchos procesos con muchos hilos.

#### 2.4.6. Implementaciones híbridas

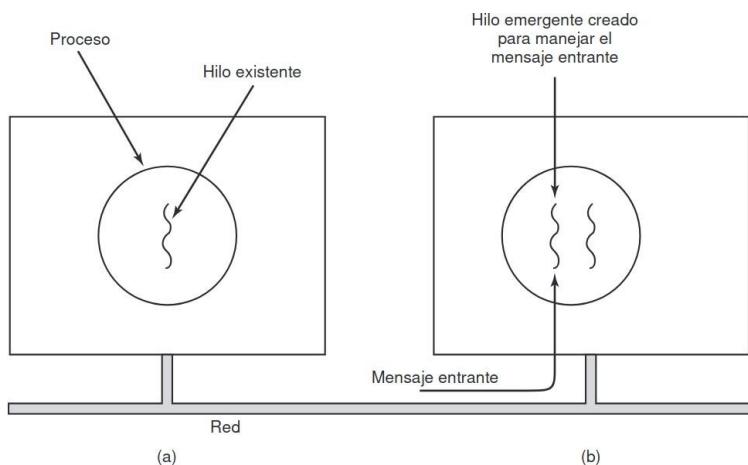
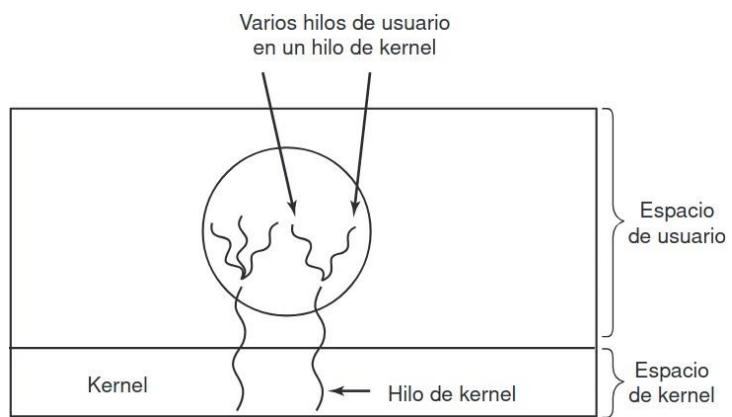
Se han investigado varias formas de tratar de combinar las ventajas de los hilos de nivel usuario con los hilos de nivel kernel. Una de esas formas es utilizar hilos de nivel kernel y después multiplexar los hilos de nivel usuario con alguno o con todos los hilos de nivel kernel. Cuando se utiliza este método, el programador

puede determinar cuántos hilos de kernel va a utilizar y cuántos hilos de nivel usuario va a multiplexar en cada uno. El kernel está consciente sólo de los hilos de nivel kernel y los planifica.

#### 2.4.7. Hilos emergentes

EL método **tradicional** para manejar los mensajes entrantes de un **sistema distribuido** es hacer que un proceso o hilo, que está bloqueado en una llamada al sistema para recibir mensajes, espere al mensaje entrante.

Una alternativa es que la **llegada de un mensaje** haga que el sistema **cree un nuevo hilo**, un **HILO EMERGENTE**, para manejar el mensaje. Como este hilo es completamente nuevo, no tiene ninguna información que restaurar, así que se **crea muy rápido**.



#### 2.4.8. Hilos en Linux

Linux es un SO con **implementación de hilos en el núcleo**. La llamada al sistema `clone(accion, pila, flags, arg)` permite disolver la distinción entre hilos y procesos, creando un nivel de abstracción superior a ellos. En función de las banderas pasadas, creará un hilo o un proceso, pero en principio no hay manera de saberlo.

Su salida en caso de existir es el pid del hijo, -1 en caso de fracaso.

- Acción: función que ejecutará el hilo/proceso creado.
- Pila: tamaño de la pila para el nuevo hilo/proceso.
- Flags: especifican qué se comparte y qué se mantiene en privado. También permiten especificar si el hilo se creará en el proceso actual o en uno nuevo.
- Arg: único argumento de la función acción.

Flag	Meaning when set	Meaning when cleared
<code>CLONE_VM</code>	Create a new thread	Create a new process
<code>CLONE_FS</code>	Share umask, root, and working dirs	Do not share them
<code>CLONE_FILES</code>	Share the file descriptors	Copy the file descriptors
<code>CLONE_SIGHAND</code>	Share the signal handler table	Copy the table
<code>CLONE_PID</code>	New thread gets old PID	New thread gets own PID
<code>CLONE_PARENT</code>	New thread has same parent as caller	New thread's parent is caller

### 2.5. Planificación de procesos

Cuando una computadora se multiprograma, con frecuencia tiene varios procesos o hilos que compiten por la CPU al mismo tiempo. Esta situación ocurre cada vez que dos o más de estos procesos se encuentran al mismo tiempo en el estado listo. Si sólo hay una CPU disponible, hay que decidir cuál proceso se va a ejecutar a continuación.

La parte del sistema operativo que realiza esa decisión se conoce como **planificador de procesos** y el algoritmo que utiliza se conoce como algoritmo de planificación. El planificador sólo considera como candidatos los procesos que están en estado listo. Siempre habrá como mínimo un proceso en estado listo, el proceso inactivo (o idle).

En los PC hay pocos procesos activos, por lo que la planificación es poco importante. En los servidores, hay muchos procesos activos, por lo que la planificación es crítica. En cualquier caso, el principal objetivo del planificador es buscar un uso eficiente de la CPU.

### 2.5.1. Cambio de contexto

El planificador debe escoger muy bien cuándo realizará un cambio de contexto, pues es una operación **muy cara computacionalmente**.

- Se pasa de modo usuario a modo núcleo.
- Se guarda el estado del proceso actual (incluyendo los registros) en la tabla de procesos y el mapa de memoria.
- Se selecciona un nuevo proceso a ejecutar mediante el algoritmo de planificación.
- Se cargan los datos del nuevo proceso.
- Se inicia el nuevo proceso.
- Probablemente sucedan muchos fallos de caché pues, desde que este nuevo proceso perdió en su momento la CPU hasta ahora, es posible que toda la información que usaba en la caché ya no esté en ella (pues los procesos que se ejecutaron entre tanto la fueron eliminado).

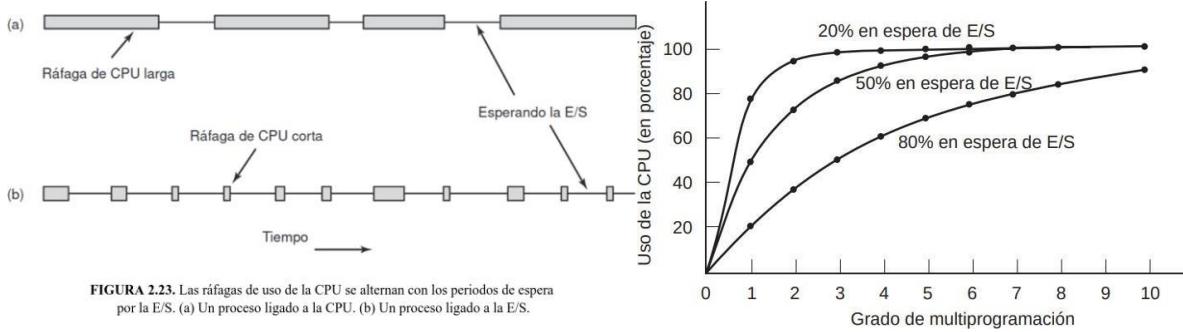
Por tanto, si las conmutaciones de procesos son **muy frecuentes**, puede llegar a consumir mucho tiempo de CPU, que podría haber sido invertido en avanzar procesos.

### 2.5.2. Tipos de procesos y planificación

Casi todos los procesos alternan ráfagas de cálculos con peticiones de E/S (de disco), como se muestra en la figura 2.23. Lo importante a observar acerca de la figura 2.23. es que algunos procesos, como el que se muestra en la figura 2.23.(a), invierten la mayor parte de su tiempo realizando cálculos, mientras que otros, como el que se muestra en la figura 2.23.(b), invierten la mayor parte de su tiempo esperando la E/S. A los primeros se les conoce como **limitados a cálculos**; a los segundos como **limitados a E/S (I/O-bound)**.

Por lo general, los procesos limitados a cálculos tienen ráfagas de CPU largas y, en consecuencia, esperas infrecuentes por la E/S, mientras que los procesos limitados a E/S tienen ráfagas de CPU cortas y, por ende, esperas frecuentes por la E/S. Observe que el factor clave es la longitud de la ráfaga de CPU, no de la ráfaga de E/S.

Los procesos limitados a E/S Los procesos limitados a E/S están limitados a la E/S debido a que no realizan muchos cálculos entre una petición de E/S y otra, no debido a que tengan peticiones de E/S en especial largas. Vale la pena observar que, a medida que las CPUs se vuelven más rápidas, los procesos tienden a ser más limitados a E/S. La idea básica aquí es que, si un proceso limitado a E/S desea ejecutarse, debe obtener rápidamente la oportunidad de hacerlo para que pueda emitir su petición de disco y mantener el disco ocupado. Como se ve en la figura 2.24., cuando los procesos están limitados a E/S, se requieren muchos de ellos para que la CPU pueda estar completamente ocupada.



La figura deja claro que, si los procesos gastan 80% de su tiempo esperando las operaciones de E/S, por lo menos debe haber 10 procesos en memoria a la vez para que el desperdicio de la CPU esté por debajo de 10%. Si el proceso promedio realiza cálculos sólo 20% del tiempo que está en la memoria, con cinco procesos en memoria a la vez la CPU deberá estar ocupada todo el tiempo.

$$Uso\ de\ la\ CPU = 1 - p^n$$

## 2.5.2. Cuando planificar un proceso

Una cuestión clave relacionada con la planificación es saber cuándo tomar decisiones de planificación. Resulta ser que hay una variedad de situaciones en las que se necesita la planificación. En primer lugar, **cuando se crea un nuevo proceso** se debe tomar una decisión en cuanto a si se debe ejecutar el proceso padre o el proceso hijo. Como ambos procesos se encuentran en el **estado listo**, es una decisión normal de programación y puede ejecutar cualquiera; es decir, el programador de procesos puede elegir ejecutar de manera legítima, ya sea el padre o el hijo.

En segundo lugar, se debe tomar una decisión de planificación cuando **un proceso termina**. Ese proceso ya no se puede ejecutar (debido a que ya no existe), por lo que se debe elegir algún otro proceso del conjunto de procesos listos. Si no hay un proceso listo, por lo general se ejecuta un **proceso inactivo** suministrado por el sistema.

En tercer lugar, cuando **un proceso se bloquea**, hay que elegir otro proceso para ejecutarlo. Algunas veces la razón del bloqueo puede jugar un papel en la elección. Sin embargo, el problema es que el planificador comúnmente no tiene la información necesaria para tomar en cuenta esta dependencia.

En cuarto lugar, cuando **ocurre una interrupción de E/S** tal vez haya que tomar una decisión de planificación. Si la interrupción proviene de un dispositivo de E/S que ha terminado su trabajo, tal vez ahora un proceso que haya estado bloqueado en espera de esa operación de E/S esté listo para ejecutarse. Es responsabilidad del planificador decidir si debe ejecutar el proceso que acaba de entrar al estado listo, el proceso que se estaba ejecutando al momento de la interrupción, o algún otro.

Si un **reloj de hardware proporciona interrupciones periódicas**, se puede tomar una decisión de planificación en cada interrupción de reloj o en cada  $k$ -ésima interrupción de reloj. Los algoritmos de planificación se pueden dividir en dos categorías con respecto a la forma en que manejan las interrupciones del reloj. Un algoritmo de programación **no apropiativo (nonpreemptive)** selecciona un proceso para ejecutarlo y después sólo deja que se ejecute hasta que **el mismo se bloquee o hasta que libera la CPU en forma voluntaria**. Por el contrario, un algoritmo de planificación **apropiativa** selecciona un proceso y deja que se ejecute por un **máximo de tiempo fijo (quantum)**.

### 2.5.3. Metas de los Algoritmos de Planificación

El planificador debe cumplir las siguientes **condiciones** dependiendo del tipo de sistema.

En **todos los sistemas**:

- **Equidad:** los procesos comparables deben recibir un servicio comparable.
- **Aplicación de políticas del sistema:** por ejemplo, darle más prioridad a un tipo de procesos que a otro.
- **Balance:** se debe intentar mantener ocupadas todas las partes del sistema cuando sea posible (tanto la CPU como los dispositivos de E/S).

En los sistemas de **procesamiento por lotes**:

- **Maximizar el rendimiento:** en rendimiento es el número de trabajos por hora que completa el sistema.
- **Minimizar el tiempo de retorno:** el tiempo de retorno es la media aritmética de los tiempos de respuesta de todos los procesos.
- **Utilización de la CPU:** se debe intentar mantener ocupada a la CPU todo el tiempo

En los sistemas con **usuarios interactivos**:

- **Minimizar el tiempo de respuesta:** el tiempo de respuesta es el tiempo que transcurre entre emitir un comando y obtener su respuesta.
- **Proporcionalidad:** se debe intentar cumplir las expectativas de los usuarios.

En los sistemas de **tiempo real**:

- **Cumplir con los plazos:** los plazos son el límite de tiempo que tiene una tarea para finalizar.
- **Predictibilidad:** se debe evitar la degradación repentina de calidad en los sistemas multimedia.

### 2.5.4. Algoritmos de planificación

No es sorprendente que distintos entornos requieran algoritmos de planificación diferentes. Esta situación se presenta debido a que las diferentes áreas de aplicación (y los distintos tipos de sistemas operativos) tienen diferentes objetivos. Tres de los entornos que vale la pena mencionar son:

1. Procesamiento por lotes.
2. Interactivo.
3. De tiempo real.

En los **sistemas de procesamiento por lotes** no hay usuarios que esperen impacientemente en sus terminales para obtener una respuesta rápida a una petición corta. En consecuencia, son aceptables los algoritmos no apropiativos (o apropiativos con largos períodos para cada proceso). Este método reduce la conmutación de procesos y, por ende, mejora el rendimiento.

En un **entorno con usuarios interactivos**, la apropiación es esencial para evitar que un proceso acapare la CPU y niegue el servicio a los demás. Aun si no hubiera un proceso que se ejecutara indefinidamente de manera intencional, podría haber un proceso que deshabilitara a los demás de manera indefinida, debido a un error en el programa. La apropiación es necesaria para evitar este comportamiento.

En los sistemas con restricciones de **tiempo real**, aunque parezca extraño, la apropiación a veces es no necesaria debido a que los procesos saben que no se pueden ejecutar durante períodos extensos, que por lo

general realizan su trabajo y se bloquean con rapidez. La diferencia con los sistemas interactivos es que los sistemas de tiempo real sólo ejecutan programas destinados para ampliar la aplicación en cuestión.

#### Todos los sistemas

- Equidad - Otorgar a cada proceso una parte justa de la CPU
- Aplicación de políticas - Verificar que se lleven a cabo las políticas establecidas
- Balance - Mantener ocupadas todas las partes del sistema

#### Sistemas de procesamiento por lotes

- Rendimiento - Maximizar el número de trabajos por hora
- Tiempo de retorno - Minimizar el tiempo entre la entrega y la terminación
- Utilización de la CPU - Mantener ocupada la CPU todo el tiempo

#### Sistemas interactivos

- Tiempo de respuesta - Responder a las peticiones con rapidez
- Proporcionalidad - Cumplir las expectativas de los usuarios

#### Sistemas de tiempo real

- Cumplir con los plazos - Evitar perder datos
- Predictibilidad - Evitar la degradación de la calidad en los sistemas multimedia

### 2.5.5. Planificación en sistemas de procesamientos por lotes

#### 2.5.5.1. Primero en entrar, primero en ser atendido

Su funcionamiento se basa en una **cola de procesos listos**.

1. Mientras se ejecuta un proceso, todos los que van llegando se introducen al final de la cola.
2. Cuando se bloquea el proceso en ejecución, se **escogerá siempre como proceso a ejecutar el primero de la cola**.
3. Cuando el proceso que se había bloqueado pase a estar listo otra vez se colocará al final de la cola como todos los demás.

Es muy **sencillo y fácil de implementar** mediante una **lista enlazada** de procesos listos.

Si se ejecutan **muchos procesos limitados a E/S**, cada vez que un proceso limitado a CPU se bloquee (por ejemplo, por un fallo de página), tendrá que esperar a que terminen de ejecutarse todos los de E/S antes de poder continuar

#### 2.5.5.2. Trabajo más corto primero

Su funcionamiento se basa en **ejecutar primero el proceso con menor tiempo de ejecución**.

- El tiempo de ejecución del primer trabajo ejecutado repercutirá en el tiempo de respuesta de todos los demás, por lo que debe ser el más veloz.

Naturalmente, se deben conocer de antemano los tiempos de ejecución de todos los procesos.

- Normalmente se consigue pidiéndole al usuario que lanza el proceso una estimación de cuánto tardará.

**Aumenta el rendimiento** del sistema pues realiza muchos procesos cortos.

**Sólo es óptimo** en términos del tiempo de retorno si todos los trabajos están disponibles al mismo tiempo

Considere el caso de cuatro trabajos, con tiempos de ejecución de  $a, b, c$  y  $d$ , respectivamente. El primer trabajo termina en el tiempo  $a$ , el segundo termina en el tiempo  $a + b$ , y así en lo sucesivo. El tiempo promedio de respuesta es  $(4a + 3b + 2c + d)/4$ . Está claro que  $a$  contribuye más al promedio que los otros

tiempos, por lo que debe ser el trabajo más corto, con  $b$  a continuación, después  $c$  y por último  $d$  como el más largo, ya que sólo afecta a su tiempo de retorno.

Como ejemplo contrario, considere cinco trabajos (del  $A$  al  $E$ ) con

tiempos de ejecución de 2, 4, 1, 1 y 1, respectivamente. Sus tiempos de llegada son 0, 0, 3, 3 y 3. Al principio sólo se pueden elegir  $A$  o  $B$ , ya que los otros tres trabajos no han llegado todavía. Utilizando el trabajo más corto primero, ejecutaremos los trabajos en el orden  $A, B, C, D, E$  para un tiempo de espera promedio de 4.6. Sin embargo, al ejecutarlos en el orden  $B, C, D, E, A$  hay una espera promedio de 4.4.

### 2.5.5.3. El menor tiempo restante a continuación

Su funcionamiento se basa en **seleccionar el proceso con menor tiempo de ejecución restante**.

Cuando llega un nuevo proceso:

1. El tiempo total del nuevo proceso se compara con el tiempo restante del que se estaba ejecutando.
2. Si necesita menos tiempo, se suspende el proceso actual y se guarda su tiempo restante.
3. Se inicia el nuevo proceso.

Naturalmente, se deben **conocer de antemano** los **tiempos de ejecución** de todos los procesos.

-Normalmente se consigue pidiéndole al usuario que lanza el proceso una estimación de cuánto tardará.

Da **muy buen servicio** a los procesos cortos

## 2.5.6. Planificación en sistemas interactivos

### 2.5.6.1. Planificación por turno circular- round robin (apropiativo)

Su funcionamiento se basa en **asignar un quantum a cada proceso y ejecutarlos secuencialmente**.

Cuando el proceso actual pierda la CPU se escogerá como nuevo proceso al **primero** de la lista, y el actual se pondrá al final. Es **muy fácil de implementar** mediante una **lista enlazada** de

procesos listos. El problema es que considera que todos los procesos tienen la misma **importancia**.

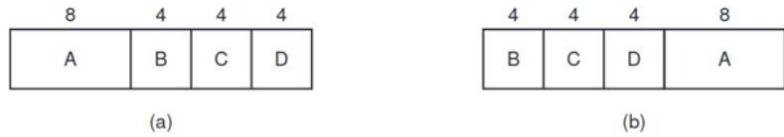
La conmutación de procesos lleva un tiempo que colabora a la **sobrecarga administrativa** (overhead), que es el tiempo de CPU gastado en ejecutar código que no es de procesos. Por lo que se debe escoger muy bien la longitud del quantum.

Si se coge un quantum demasiado **corto**, se realizan demasiado cambios de contexto, por lo que el overhead es una proporción grande del tiempo de CPU, aunque los procesos tengan que esperar meno para poder ejecutarse. Si se escoge un quantum demasiado **largo**, se realizan menos cambios de contexto innecesarios pues muchos procesos no agotan sus quantums, pero si hay muchos procesos los tiempos de esperar serán muy largos.

### 2.5.6.2. Planificación por prioridad

Su funcionamiento se basa en **asignar una prioridad a cada proceso y ejecutar aquel proceso listo con prioridad más alta**. Se debe **evitar** que los procesos con alta prioridad **acaparen la CPU**. Hay 2 métodos para conseguir esto:

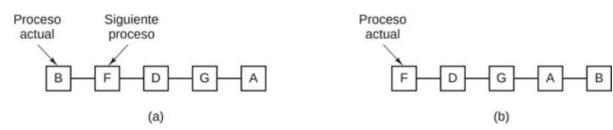
- Envejecimiento: cada vez que el proceso en ejecución agote un quantum, se reduce su prioridad y si otro proceso tiene alguna más alta, se conmuta.



**FIGURA 2.26.** Un ejemplo de planificación tipo el trabajo más corto primero.

(a) Ejecución de cuatro trabajos en el orden original.

(b) Ejecución de cuatro trabajos en el orden del tipo “el trabajo más corto primero”.



**FIGURA 2.27.** Planificación de turno circular. (a) La lista de procesos ejecutables.

(b) La lista de procesos ejecutables una vez que  $B$  utiliza su **quantum**.

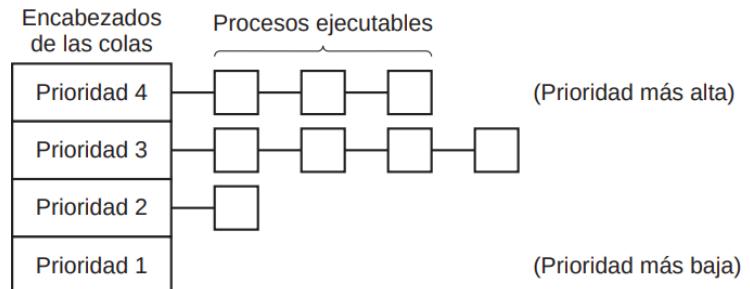
- Cuotas de CPU: cada vez que el proceso en ejecución agote un quantum, se cambia al siguiente proceso con más prioridad.

La **asignación de prioridad** a los procesos se puede hacer de dos formas:

- Estática: se asigna la prioridad del proceso cuando este comienza y se mantiene toda su ejecución
- Dinámica: la prioridad del proceso puede cambiar durante su ejecución. Por ejemplo, se le asigna a cada proceso una prioridad de  $1/f$  donde  $f$  es la fracción del último quantum usada por el proceso. Así nos aseguramos de darle **más prioridad a los procesos limitados a E/S**, como habíamos explicado antes.

Se **agrupan** los procesos en **colas de prioridad** y se usa planificación por prioridad entre clases, o planificación round robin dentro de cada clase.

Mientras haya proceso en la clase de mayor prioridad, se ejecutarán estos en round robin. SI la clase de mayor prioridad está vacía, se ejecutarán los de la siguiente, y así sucesivamente. Para asegurarse de que incluso los procesos de prioridad más baja consiguen ejecutarse, se deben **ajustar las clases dinámicamente**.



### 2.6.6.3. Otros algoritmos de planificación

El **proceso más corto a continuación** es una adaptación del trabajo más corto primero. realizar estimaciones con base en el comportamiento anterior y ejecutar el proceso con el tiempo de ejecución estimado más corto. Suponga que el tiempo estimado por cada comando para cierta terminal es  $T_0$ . Ahora suponga que su siguiente ejecución se mide como  $T_1$ . Podríamos actualizar nuestra estimación mediante una suma ponderada de estos dos números, es decir,  $aT_0 + (1 - a)T_1$ . Por medio de la elección de  $a$  podemos decidir hacer que el proceso de estimación olvide las ejecuciones anteriores rápidamente o que las recuerde por mucho tiempo.

Si nos dan  $a = \frac{1}{2}$ ,  $T_0 = 1ms$ ,  $T_1 = 4ms$ ,  $T_2 = 3ms$  y  $T_3 = 2ms$ :

$$\begin{aligned} 1^{\text{a}}\text{Estimacion} &= T_0 \\ 2^{\text{a}}\text{Estimacion} &= \frac{1}{2}T_0 + \frac{1}{2}T_1 \\ 3^{\text{a}}\text{Estimacion} &= \frac{1}{2}2^{\text{a}}\text{Est} + \frac{1}{2}T_2 \\ 4^{\text{a}}\text{Estimacion} &= \frac{1}{2}3^{\text{a}}\text{Est} + \frac{1}{2}T_3 \end{aligned}$$

La **planificación garantizada** es un método completamente distinto para la planificación que consiste en hacer promesas reales a los usuarios acerca del rendimiento y después cumplirlas. Una de ellas, que es realista y fácil de cumplir es: si hay  $n$  usuarios conectados mientras usted está trabajando, recibirá aproximadamente  $1/n$  del poder de la CPU. De manera similar, en un sistema de un solo usuario con  $n$  procesos en ejecución, mientras no haya diferencias, cada usuario debe obtener  $1/n$  de los ciclos de la CPU.

Se puede utilizar otro algoritmo distinto a la panificación garantizada para producir resultados similares con una implementación mucho más sencilla. Este algoritmo se conoce como **planificación por sorteo**. La idea básica es dar a los procesos boletos de lotería para diversos recursos del sistema, como el tiempo de la CPU. Cada vez que hay que tomar una decisión de planificación, se selecciona un boleto de lotería al azar y el proceso que tiene ese boleto obtiene el recurso.

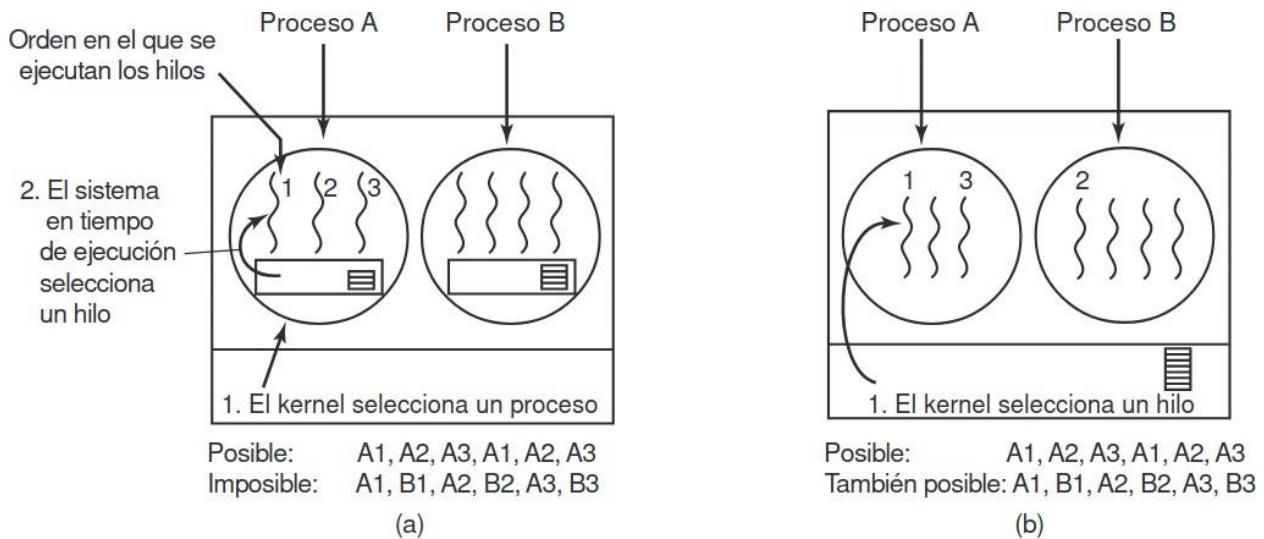
Algunos sistemas toman en consideración quién es el propietario de un proceso antes de planificarlo. En este modelo, a cada usuario se le asigna cierta fracción de la CPU y el planificador selecciona procesos de tal forma que se cumpla con este modelo. Esto se conoce como **planificación por partes equitativas**.

## 2.7. Planificación de hilos

Cuando varios procesos tienen múltiples hilos cada uno, tenemos dos niveles de paralelismo presentes: procesos e hilos. La planificación en tales sistemas difiere en forma considerable, dependiendo de si hay soporte para hilos a nivel usuario o para hilos a nivel kernel (o ambos).

En los hilos a nivel de usuario el **kernel planifica los procesos**, pues no es consciente de la existencia de hilos. Cuando se planifica un proceso, el **planificador de hilos del sistema en ejecución** de ese proceso **selecciona un hilo** en concreto. Mientras dure el quantum de ese proceso, seguirá seleccionando sus hijos. Permite crear **planificadores específicos** adaptados a las características de una aplicación concreta.

En los hilos a nivel de núcleo, el **kernel planifica hilos**, en principio sin tener en cuenta de qué proceso son (aunque puede hacerlo). Puede darle más importancia a planificar hilos del proceso que se está ejecutando actualmente pues commutar a un hilo de otro proceso sería más caro. CUando un hilo se **bloquea**, no suspende todo el proceso.



**FIGURA 2.29.** (a) Posible planificación de hilos a nivel usuario con un quántum de 50 mseg para cada proceso e hilos que se ejecutan durante 5 mseg por cada ráfaga de la CPU.  
(b) Posible planificación de hilos a nivel kernel con las mismas características que (a).

## 2.8 Cosas importantes de este Tema

Saber sobre señales y sobre cómo usar los procesos tanto como hilos tema teoría le suele dar más igual. Lo del sigaction y todas esas funciones raras lo suele preguntar. Sobre los algoritmos saber cómo funcionan y las fórmulas porque, aunque no suele, las puede preguntar. Las diferentes implementaciones de hilos son muy importantes al igual que los estados de los procesos.



# Administración de Memoria - Métodos de Tortura Medieval

Una de las principales tareas del SO es **crear y administrar abstracciones de la memoria** del sistema.

## 3.1. La memoria y el Sistema Operativo

A través de los años se ha elaborado el concepto de **jerarquía de memoria**, de acuerdo con el cual, las computadoras tienen unos cuantos megabytes de memoria caché, muy rápida, costosa y volátil, unos cuantos gigabytes de memoria principal, de mediana velocidad, a precio mediano y volátil, unos cuantos terabytes de almacenamiento en disco lento, económico y no volátil, y el almacenamiento removible, como los DVDs y las memorias USB. El trabajo del sistema operativo es abstraer esta jerarquía en un modelo útil y después administrarla.

La parte del sistema operativo que administra (parte de) la jerarquía de memoria se conoce como **administrador de memoria**. Su trabajo es administrar la memoria con eficiencia: **llevar el registro de que partes de la memoria están en uso, asignar memoria a los procesos cuando la necesiten y desasignarla cuando terminen**. Como generalmente el hardware es el que se encarga de administrar el nivel más bajo de memoria caché, nos concentraremos en el modelo del programador de la memoria principal y en cómo se puede administrar bien.

## 3.2. Sin abstracción de memoria

La abstracción más simple de memoria es ninguna abstracción. Cada programa ve simplemente **la memoria física**. Cuando un programa ejecuta una instrucción como

```
MOV REGISTRO1, 1000
```

la computadora sólo mueve el contenido de la ubicación de memoria física 1000 a *REGISTRO1*. Así, el modelo de programación que se presenta al programador era simplemente la memoria física, un conjunto de direcciones desde 0 hasta cierto valor máximo, en donde cada dirección correspondía a una celda que contenía cierto número de bits, comúnmente ocho.



Bajo estas condiciones, no era posible tener dos programas ejecutándose en memoria al mismo tiempo. Si el primer programa escribía un nuevo valor en, por ejemplo, la ubicación 2000, esto borraría cualquier valor que el segundo programa estuviera almacenando ahí. Ambos programas fallarían de inmediato.

No obstante, aun sin abstracción de memoria es posible ejecutar varios programas al mismo tiempo. Lo que el sistema operativo debe hacer es guardar todo el contenido de la memoria en un archivo en disco, para después traer y ejecutar el siguiente programa. **Mientras sólo haya un programa a la vez en la memoria no hay conflictos.**

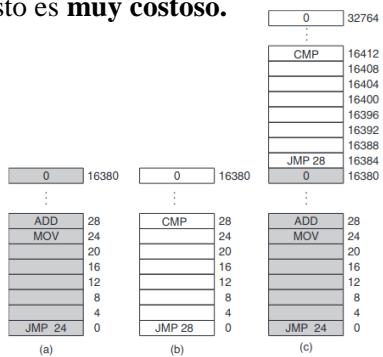
Sin embargo, esta solución tenía una gran desventaja, que se ilustra en la figura 3.2. Como muestran las figuras 3.2(a) y (b), se tienen dos programas. Cuando los dos programas se cargan consecutivamente en la memoria, empezando en la dirección 0, tenemos la situación de la figura 3-2(c). Para este ejemplo, suponemos que el sistema operativo está en la parte alta de la memoria y no se muestra.

Hay dos formas de ejecutar múltiples programas sin abstracción de memoria.

- **Intercambio:** cada programa se carga en la totalidad de la memoria principal.

Por tanto, cada vez que se cambia de programa se tiene que guardar todo el contenido de la memoria en el disco, y sustituirlo por el nuevo programa. Esto es **muy costoso**.

- **Carga consecutiva:** los programas se cargan enteros en la memoria, uno detrás de otro. Como consecuencia, las referencias a posiciones de memoria dejan de ser correctas, ya que se escribieron considerando que se disponía de la memoria entera. Para solucionar esto se usa la **reubicación estática**: cuando se carga un programa, se les suma a todas sus referencias a memoria la posición donde se cargó. Esto soluciona el problema, pero también hace mucho **más lenta la carga** de programas.



**Figura 3-2.** Ilustración del problema de reubicación. (a) Un programa de 16 KB. (b) Otro programa de 16 KB. (c) Los dos programas cargados consecutivamente en la memoria.

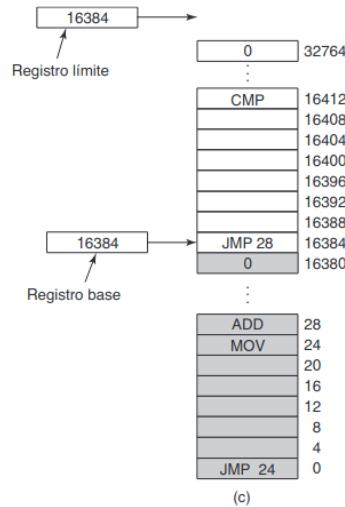
### 3.3. Abstracción de memoria: espacios de direcciones

### **3.3.1. La noción de un espacio de direcciones**

Con todo, exponer la memoria física a los procesos tiene varias desventajas. En primer lugar, si los programas de usuario pueden direccionar cada byte de memoria, pueden estropear el sistema operativo con facilidad, ya sea intencional o accidentalmente, con lo cual el sistema se detendría en forma súbita. Este problema existe aun cuando sólo haya un programa de usuario (aplicación) en ejecución. En segundo lugar, con este modelo es difícil tener varios programas en ejecución a la vez.

Hay que resolver dos problemas para permitir que haya varias aplicaciones en memoria al mismo tiempo sin que interfieran entre sí: protección y reubicación. Una solución es inventar una nueva abstracción para la memoria: el espacio de direcciones. Un **espacio de direcciones** (*address space*) es el conjunto de direcciones que puede utilizar **un proceso para direccionar la memoria**. Cada proceso tiene su **propio espacio de direcciones, independiente** de los que pertenecen a otros procesos (*excepto en ciertas circunstancias especiales en donde los procesos desean compartir sus espacios de direcciones*).

Algo un poco difícil es proporcionar a cada programa su propio espacio de direcciones, de manera que la dirección 28 en un programa indique una ubicación física distinta de la dirección 28 en otro programa. La solución sencilla utiliza una versión muy simple de la **reubicación dinámica**. Lo que hace es asociar el espacio de direcciones de cada proceso sobre una parte distinta de la memoria física, de una manera simple. La solución clásica, es equipar cada CPU con dos registros de hardware especiales, conocidos comúnmente como los registros **base** y **límite**.



**Figura 3-3.** Se pueden utilizar registros base y límite para dar a cada proceso un espacio de direcciones separado.

En cada **referencia** a memoria, el hardware suma el valor base a la dirección referencia, comprueba si el resultado está más allá del final del espacio (dirección base + límite). Si es así, genera un fallo y aborta el acceso. Como consecuencia, los **accesos a memoria son más lentos**, pues implican una comprobación (rápida)

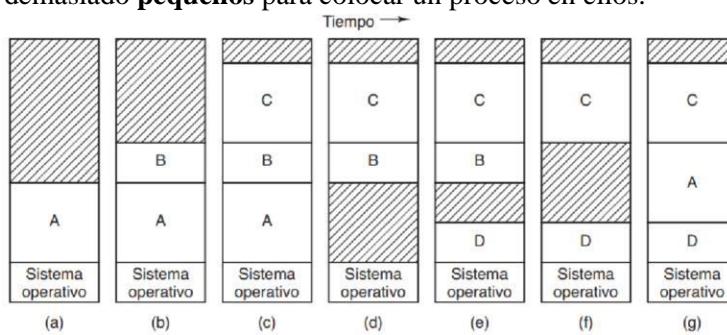
y una suma (lenta). Como mínimo, se accede a memoria una vez por ciclo, para realizar el fetch de la instrucción a ejecutar.

Este esquema funciona cuando la **memoria física** es lo **suficientemente grande** como para contener todos los procesos en ejecución en un momento dado. En la práctica, la memoria que requieren los procesos es mucho mayor que la memoria física disponible. Hay dos esquemas alternativos que solucionan esto: el **intercambio** y la **memoria virtual**.

### 3.3.2. Intercambio

Si la memoria física de la computadora es lo **bastante grande como para contener todos los procesos**, los esquemas descritos hasta ahora funcionarán en forma más o menos correcta. Pero en la práctica, la cantidad total de RAM que requieren todos los procesos es a menudo mucho mayor de lo que puede acomodarse en memoria. Para mantener todos los procesos en memoria todo el tiempo se requiere una gran cantidad de memoria y no puede hacerse si no hay memoria suficiente.

El **intercambio**, consiste en llevar cada proceso completo a memoria, **ejecutarlo durante cierto tiempo** y después **regresarlo al disco**. El mismo proceso podrá estar en distintos puntos del tiempo en distintas posiciones de memoria, por lo que se necesitará usar los registros para manejar las referencias a la memoria. Los procesos inactivos mayormente son almacenados en disco, de tal manera que no ocupan memoria cuando no se están ejecutando. El intercambio de procesos en memoria provoca que se vaya **fragmentando**, pues se van generando **huecos demasiado pequeños** para colocar un proceso en ellos.



**FIGURA 3.4.** La asignación de la memoria cambia a medida que llegan procesos a la memoria y salen de esta. Las regiones sombreadas son la memoria sin usar.

Cuando la fragmentación es muy alta, se puede usar la **compactación**, que combina todos los huecos pequeños en un hueco grande desplazando procesos lo más abajo posible. Es un procedimiento **muy lento**. Un aspecto que vale la pena mencionar es la cantidad de memoria que debe asignarse a un proceso cuando éste se crea o se intercambia. Si los procesos se crean con un tamaño fijo que nunca cambia, entonces la asignación es sencilla: el sistema operativo asigna exactamente lo necesario, ni más ni menos.

Info

**Fragmentación interna:**  
La fragmentación interna ocurre cuando un bloque de memoria asignado a un proceso es más grande de lo necesario para contener los datos del proceso. El espacio restante dentro del bloque no puede ser utilizado por otros procesos, lo que genera un desperdicio.

**Fragmentación externa:**  
La fragmentación externa ocurre cuando hay suficiente memoria libre para satisfacer una solicitud, pero esa memoria está dividida en pequeños bloques no contiguos, lo que impide la asignación de un espacio único lo suficientemente grande.

No obstante, los procesos pueden **crecer**, por ejemplo, para acomodar su pila o asignar memoria dinámicamente en el heap, como en muchos lenguajes de programación, ocurre un problema cuando un proceso trata de crecer.

Si un proceso necesita crecer intentará:

- Ocupar huecos adyacentes
- Si no hay huecos adyacentes, se moverán a un hueco lo suficientemente grande
- Si no hay huecos lo suficientemente grandes, se intentará compactar la memoria para crear uno y se moverá allí.
- Si nada de esto funciona, el proceso se tendrá que suspender hasta que se libere la memoria suficiente.

Para evitar tener que hacer todo eso cada vez que un proceso necesite crecer, será conveniente asignar **memoria en exceso** cuando se carga o mueve uno en memoria. En los procesos con **dos segmentos en crecimiento** (pila y heap), estos pueden crecer en direcciones opuestas es en el espacio para el crecimiento hasta que ambos se encuentren. Si el espacio para crecimiento se **agota**, habrá que mover o suspender el proceso como antes.

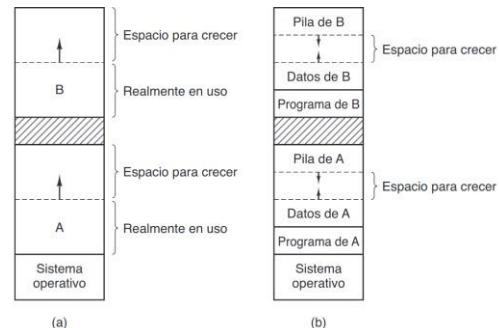


Figura 3-5. (a) Asignación de espacio para un segmento de datos en crecimiento. (b) Asignación de espacio para una pila en crecimiento y un segmento de datos en crecimiento.

### 3.3.3. Administración de memoria libre

#### 3.3.3.1. Administración de memoria con mapas de bits

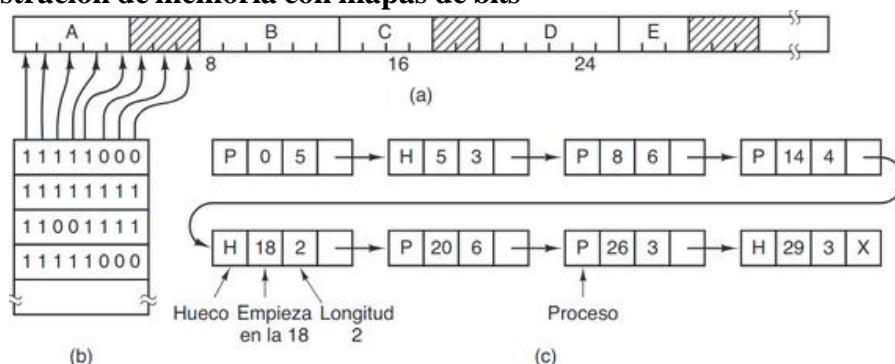


FIGURA 3.6. (a) Una parte de la memoria con cinco procesos y tres huecos. Las marcas de graduación muestran las unidades de asignación de memoria. Las regiones sombreadas (0 en el mapa de bits) están libre (b) El mapa de bits correspondiente. (c) La misma información en forma de lista.

La memoria se divide en **unidades de asignación** de un tamaño fijo. Para cada unidad de asignación en la memoria habrá un bit correspondiente en el mapa de bits, que será 0 si está libre y 1 si está ocupada. El **tamaño** del mapa será siempre **constante** pues depende del tamaño de la memoria y del de la unidad de asignación. (La unidad de asignación suele ser la unidad direccionable, explicó después lo que es la unidad direccionable por si hay lagunas de Fundamentos dos Computadores).

$$TamMapBits = \frac{TamMem}{UnidadAsig}$$

Una unidad de asignación pequeña implica un mapa de bits grande. Una unidad de asignación pequeña implica un mapa de bits más pequeño, pero se puede desperdiciar mucha memoria en la última unidad de cada proceso si su tamaño no es un múltiplo del tamaño de la unidad de asignación.

El principal problema es que cada vez que se lleva un proceso a memoria el administrador tendrá que **recorrer el mapa de bits** contando ceros hasta encontrar una **secuencia de ceros**, tan larga como el tamaño del proceso, lo cual es muy **lento**.

#### 3.3.3.2. Administración de memoria con listas ligadas

La lista está formada por **segmentos de memoria** que o bien contienen un proceso o bien son un hueco. Cada entrada especificará si se trata de un **proceso** o un **hueco**, la **dirección** donde comienza, su **longitud** y un apuntador a la **siguiente entrada**. El **tamaño de la lista no es constante**, pues depende del número de

huecos y procesos que haya en la memoria de un determinado momento. La lista está ordenada por **dirección de inicio**. De esta manera, cuando un proceso **regrese a disco** se podrá **actualizar** muy **fácilmente**.

- Si las dos entradas vecinas son procesos se marca el proceso intercambiado como hueco, sin más.
- Si una de las entradas vecinas es un hueco, la entrada del proceso intercambiado se fusiona con la del hueco, creando un hueco más grande.

Si las dos entradas vecinas son huecos, las tres entradas se fusionan, creando un hueco más grande. Fusionando los huecos, la memoria se fragmenta menos. Cuando los procesos y huecos se mantienen en una lista ordenada por dirección, se pueden utilizar varios algoritmos para asignar memoria a un proceso creado. El algoritmo más simple es el de primer ajuste: el administrador de memoria explora la lista de segmentos hasta encontrar un hueco que sea lo bastante grande. El algoritmo del primer ajuste es rápido debido a que busca lo menos posible.

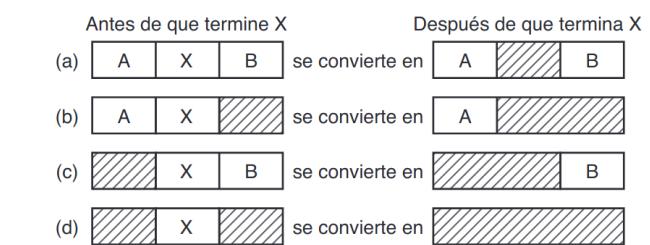


Figura 3-7. Cuatro combinaciones de los vecinos para el proceso en terminación, X.

Una pequeña variación del algoritmo del primer ajuste es el algoritmo del **siguiente ajuste**. Funciona de la misma manera que el primer ajuste, excepto porque lleva un registro de dónde se encuentra cada vez que descubre un hueco adecuado. La siguiente vez que es llamado para buscar un hueco, empieza a buscar en la lista **desde el lugar en el que se quedó la última vez**, en vez de empezar siempre desde el principio, como el algoritmo del primer ajuste. Las simulaciones realizadas muestran que el algoritmo del siguiente ajuste tiene un rendimiento ligeramente peor que el del primer ajuste.

Otro algoritmo muy conocido y ampliamente utilizado es el del **mejor ajuste**. Este algoritmo busca en toda la lista, de principio a fin y toma el hueco más pequeño que sea adecuado. En vez de dividir un gran hueco que podría necesitarse después, el algoritmo del mejor ajuste trata de buscar un hueco que esté **cerca del tamaño actual necesario**, que coincida mejor con la solicitud y los huecos disponibles.

El algoritmo del mejor ajuste es **más lento** que el del primer ajuste, ya que debe buscar en toda la lista cada vez que se le llama. De manera sorprendente, también provoca más desperdicio de memoria que los algoritmos del primer ajuste o del siguiente ajuste, debido a que tiende a llenar la memoria con huecos pequeños e inutilizables. El algoritmo del primer ajuste genera huecos más grandes en promedio.

Para resolver el problema de dividir las coincidencias casi exactas en un proceso y en un pequeño hueco, podríamos considerar el algoritmo del **peor ajuste**, es decir, tomar siempre el **hueco más grande disponible**, de manera que el nuevo hueco sea lo bastante grande como para ser útil. La simulación ha demostrado que el algoritmo del peor ajuste no es muy buena idea tampoco. Los cuatro algoritmos pueden ser acelerados manteniendo listas separadas para los procesos y los huecos. De esta forma, todos ellos dedican toda su energía a inspeccionar los huecos, no los procesos.

Si se mantienen distintas listas para los procesos y los huecos, la lista de huecos se puede mantener ordenada por el tamaño, para que el algoritmo del mejor ajuste sea más rápido. Cuando el algoritmo del mejor ajuste busca en una lista de huecos, del más pequeño al más grande, tan pronto como encuentre un hueco que ajuste, sabrá que el hueco es el más pequeño que se puede utilizar, de aquí que se le denominé el mejor ajuste. Tenemos inserción rápida, liberación lenta (los huecos adyacentes no se podrán fusionar directamente).

Un algoritmo de asignación más es el denominado de **ajuste rápido**, el cual mantiene listas separadas para algunos de los tamaños más comunes solicitados. Con el algoritmo del ajuste rápido, buscar un hueco del tamaño requerido es extremadamente rápido, pero tiene la misma desventaja que todos los esquemas que se ordenan por el tamaño del hueco: cuando un proceso termina o es intercambiado, buscar en sus vecinos para ver si es posible una fusión es un proceso costoso.

## 3.4. Memoria virtual

Existe la necesidad de ejecutar programas que son demasiado grandes como para caber en la memoria y sin duda existe también la necesidad de tener sistemas que puedan soportar varios programas ejecutándose al mismo tiempo, cada uno de los cuales cabe en memoria, pero que en forma colectiva exceden el tamaño de esta. El intercambio no es una opción atractiva, ya que es lento. Por otro lado, el proceso de dividir programas grandes en partes modulares más pequeñas (llamadas overlays o sobrepuertos) consumía mucho tiempo, y era aburrido y propenso a errores.

La solución a esto se conoce como **memoria virtual**. La idea básica detrás de la memoria virtual es que cada programa tiene su propio espacio de direcciones, el cual se divide en trozos llamados **páginas**. Cada página es un **rango contiguo de direcciones**. Estas páginas **se asocian a la memoria física**, pero **no todas tienen que estar en la memoria física para poder ejecutar el programa**. Cuando el programa hace referencia a una parte de su espacio de direcciones que está en la memoria física, el hardware realiza la asociación necesaria al instante. Cuando el programa hace referencia a una parte de su espacio de direcciones que *no* está en la memoria física, el sistema operativo recibe una alerta para **buscar la parte faltante y volver a ejecutar la instrucción que falló**. La **RAM** actúa como una especie de caché.

Así como dato podemos tener una memoria virtual cuyo tamaño sea igual que el de la memoria física y sigue siendo necesaria por si usamos varios procesos. Incluso podría ser menor que el tamaño de la memoria física, desaprovechando hardware. Por norma general es mayor.

### 3.4.1. Paginación

Primero quiero recordar que la unidad direccionable es aquel conjunto de bits o bytes a los que accedemos en bloque así grosso modo. Si por ejemplo tenemos  $2^n$  direcciones y la unidad direccionable es el bit pues implica que cada dirección accede a 1 bit por lo que la memoria va a tener un tamaño de  $2^n$  bits. Por otro lado si la unidad direccionable fuese el byte cada una de las direcciones accede en bloque a un byte (8 bits) por lo que el tamaño de la memoria va a ser  $2^n \cdot 8$  bits. Por lo que se deduce la siguiente fórmula:

$$TamMemoria = N^{\circ}Dirs \cdot UdDir$$

El espacio de direcciones de un proceso se divide en **páginas**, que son un rango contiguo de direcciones de  $2^{n-p}$  ( $\frac{2^n}{2^p}$ ) unidades direccionables (bytes o palabras). El tamaño del espacio de direcciones será de  $2^n$  unidades direccionables, siendo  $n$  la longitud de palabra del sistema (el ancho de los registros del procesador). El espacio de direcciones se divide en  $2^p$  páginas. Cada dirección del espacio de direcciones se divide en campo de página ( $p$  MSB) que indica a qué páginas pertenece y el campo de desplazamiento ( $n-p$  LSB) que indica a qué unidad direccionable de la página se refiere la dirección.

La memoria física se dividirá en **marcos de página** del mismo tamaño que las páginas del espacio de direcciones. El tamaño de la memoria física tendrá que ser, por lo tanto, un múltiplo del tamaño de página. La memoria física se divide en *tamaño\_total/tamaño\_página* marcos.

**Las páginas se asignan a marcos de manera completamente asociativa**, es decir, cualquier página de cualquier espacio de direcciones puede estar en cualquier marco. De esta manera, se provoca el mínimo número de fallo de página.

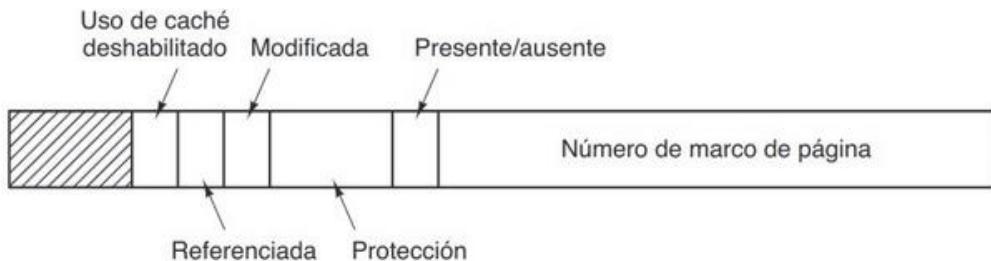
### 3.4.2. Tablas de páginas (TP)

En una implementación simple, la asociación de direcciones virtuales a direcciones físicas se puede resumir de la siguiente manera: **la dirección virtual se divide en un número de página virtual (bits de mayor orden) y en un desplazamiento (bits de menor orden)**. Por ejemplo, con una dirección de 16 bits y un tamaño

de página de 4 KB, los 4 bits superiores podrían especificar una de las 16 páginas virtuales y los 12 bits inferiores podrían entonces especificar el desplazamiento de bytes (0 a 4095) dentro de la página seleccionada. Sin embargo, también es posible una división con 3, 5 u otro número de bits para la página. Las distintas divisiones implican diferentes tamaños de página.

El número de página virtual se utiliza como **índice en la tabla de páginas** para buscar la entrada para esa página virtual. En la entrada en la tabla de páginas, se encuentra el número de marco de página (si lo hay). El **número del marco de página se adjunta al extremo de mayor orden del desplazamiento**, reemplazando el número de página virtual, para formar una dirección física que se pueda enviar a la memoria.

Por ende, el propósito de la tabla de páginas es asociar páginas virtuales a los marcos de página. Hablando en sentido matemático, la tabla de páginas es una función donde el número de página virtual es un argumento y el número de marco físico es un resultado. Utilizando el resultado de esta función, el campo de la página virtual en una dirección virtual se puede reemplazar por un campo de marco de página, formando así una dirección de memoria física.



**FIGURA 3.11.** Una típica entrada en la tabla de páginas.

- **Número de marco de página:** indica a que marco está asociada esa página (si alguno quiere saber que se almacena cuando la pagina no está en memoria pregunte al profesor).
- **Bit presente/ausente:** si es 1, significa que la página está en memoria principal. Si es 0, no está en memoria principal.
- **Bits de protección:** (r,w,x) si uno es 1, la página tiene ese permiso. Si es 0 no lo tiene. Por tanto, no se deben mezclar datos e instrucciones en la misma página, pues eso implicaría activar todos sus bits de protección.
- **Bit modificada:** se pone a 1 cuando se escribe en la página. Se usa cuando se quita la página de memoria principal para determinar si habrá que **copiar** su contenido en el disco. Si se tiene que hacer, se copia la página **entera**, pues no se sabe dónde fue modificada.
- **Bit referenciada:** se pone a 1 cuando se referencia la página (para escritura o lectura). Se usa como apoyo al **algoritmo de reemplazo**, pues las páginas que no han sido usadas desde que se cargaron serán mejores candidatas para ser reemplazadas.
- **Bit de deshabilitación de caché:** si es 1, ninguna de las palabras de la página podrá almacenarse en la caché. Se usa en páginas compartidas o páginas donde se ejecutan operaciones de lectura y escritura de E/S. Para mantener la **coherencia de caché**. Por ejemplo, si hay dos hilos de un proceso ejecutándose a la vez cada uno en un núcleo y uno de ellos pretende escribir en una página, la llevará a la caché de su núcleo. La modificación se almacenará en la caché y no será visible por el otro hilo hasta que esa línea de caché sea reemplazada.

Puede haber bastantes más bits para **ayudar a los algoritmos de reemplazo**. Normalmente, la **dirección de disco** usada para guardar la página cuando no está en memoria **no se almacena en la tabla**. Con esta información la MMU es capaz de realizar las traducciones.

El SO conoce la ubicación de las tablas gracias a unos registros. Estas tablas de páginas se almacenan en el kernel no en el espacio de usuario. Cuando se carga un proceso la tabla de páginas no genera fallo de página porque cuando se carga el proceso no se accede a ninguna dirección de memoria y por definición del Tanenbaum, un fallo de página es **una excepción** que ocurre cuando un programa llama o salta a una instrucción que no está cargada en memoria, y para llamar a una instrucción primero hay que cargar el programa. Además sabemos que al cargar el proceso también se carga su tabla de páginas por lo que no genera fallo de páginas (suponiendo que ocupe solo una página la tabla).

### 3.4.3. Traducción de dirección virtual a dirección física

Cuando se utiliza memoria virtual, las direcciones virtuales no van directamente al bus de memoria. En vez de ello, van a una **MMU** (*Memory Management Unit*, Unidad de administración de memoria) que asocia las direcciones virtuales a las direcciones de memoria físicas.

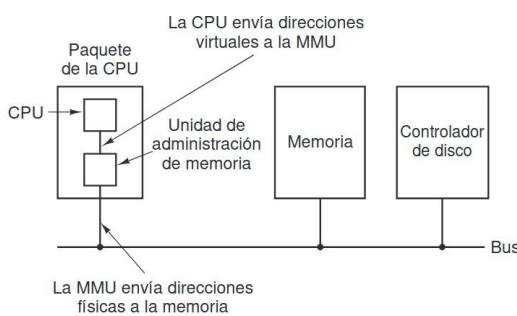


Figura 3-8. La posición y función de la MMU. Aquí la MMU se muestra como parte del chip de CPU, debido a que es común esta configuración en la actualidad. Sin embargo, lógicamente podría ser un chip separado y lo era hace años.

principal está llena, habrá que reemplazar alguna de sus páginas.

Siempre que se produce un **fallo de página** se genera una **excepción** (que si recordamos es un trap a modo kernel), y el proceso correspondiente se **bloquea** a la espera de que se gestione el fallo de página y la interrupción para poder retomar su ejecución.

Por otra parte, la MMU también gestiona el acceso a memoria mediante la verificación de los permisos de acceso de los procesos a las direcciones de memoria solicitadas, aumentando la seguridad e impidiendo que accedan a zonas de memoria que no tienen autorizadas.

En la figura 3.9 se muestra un ejemplo muy simple de cómo funciona esta asociación. En este ejemplo, tenemos una computadora que genera direcciones de 16 bits, desde 0 hasta 64 K. Éstas son las direcciones virtuales. Sin embargo, esta computadora sólo tiene 32 KB de memoria física. Así, aunque se pueden escribir programas de 64 KB, no se pueden cargar completos en memoria y ejecutarse. No obstante, una copia completa de la imagen básica de un programa, de hasta 64 KB, debe estar presente en el disco para que las partes se puedan traer a la memoria según sea necesario. En este ejemplo son de 4 KB, pero en sistemas reales se han utilizado tamaños de página desde 512 bytes hasta 64 KB. Con 64 KB de espacio de direcciones virtuales y 32 KB de memoria física obtenemos 16 páginas virtuales y 8 marcos de página.

La **MMU** comprueba la **tabla de páginas**, usando los números de página virtuales como índices de la tabla. Si el bit de presente/ausente es 1 la MMU cambia el campo de página de la dirección virtual por el número de marco asociado a ella, el campo de desplazamiento se mantiene idéntico y se coloca la dirección obtenida en el bus de memoria.

Si el bit de presente/ausente es 0 se **bloquea el proceso**, se busca y copia la dirección buscada del disco a la memoria principal y si la memoria

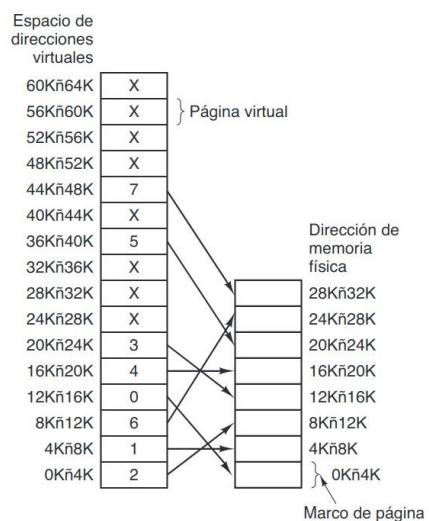


Figura 3-9. La relación entre las direcciones virtuales y las direcciones de memoria física está dada por la tabla de páginas. Cada página empieza en un múltiplo de 4096 y termina 4095 direcciones más arriba, por lo que de 4 K a 8 K en realidad significa de 4096 a 8191 y de 8 K a 12 K significa de 8192 a 12287.

Cada página contiene exactamente 4096 direcciones que empiezan en un múltiplo de 4096 y terminan uno antes del múltiplo de 4096. Por ejemplo, cuando el programa trata de acceder a la dirección 0 usando la instrucción

MOV REG,0

la dirección virtual 0 se envía a la MMU. La MMU ve que esta dirección virtual está en la página 0 (0 a 4095), que de acuerdo con su asociación es el marco de página 2 (8192 a 12287). De manera similar, la instrucción

MOV REG,8192

se transforma efectivamente en

MOV REG,24576

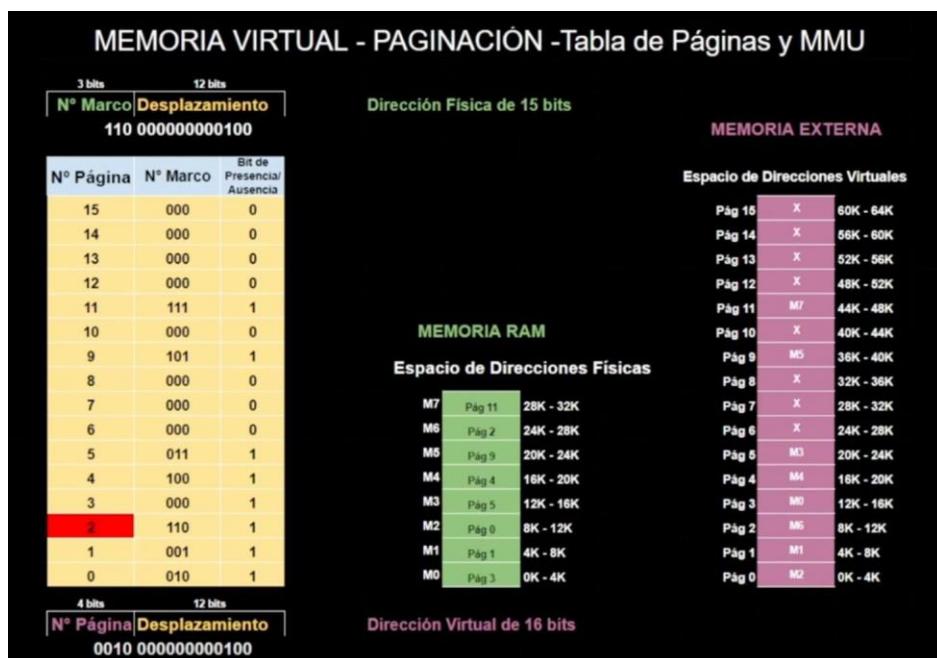
debido a que la dirección virtual 8192 (en la página virtual 2) se asocia con la dirección 24576 (en el marco de página física 6). Como tercer ejemplo, la dirección virtual 20500 está a 20 bytes del inicio de la página virtual 5 (direcciones virtuales 20480 a 24575) y la asocia con la dirección física  $12288 + 20 = 12308$ .

Como sólo tenemos ocho marcos de página físicos, sólo ocho de las páginas virtuales en la figura 3.9 se asocian a la memoria física. Las demás, que se muestran con una cruz en la figura, no están asociadas. En el hardware real, un **bit de presente/ausente** lleva el registro de cuáles páginas están físicamente presentes en la memoria. ¿Qué ocurre si el programa hace referencia a direcciones no asociadas, por ejemplo, mediante el uso de la instrucción

MOV REG,32780

que es el byte 12 dentro de la página virtual 8 (empezando en 32768)? La MMU detecta que la página no está asociada (lo cual se indica mediante una cruz en la figura) y hace que la CPU haga un trap al sistema operativo (esto es porque es una excepción y una excepción es un trap). El SO selecciona una página que desalojar para meter una nueva, guardándola en el disco y trallendo la nueva. Después reanuda la instrucción que causó el fallo.

Por ejemplo, si el sistema operativo decidiera desalojar el marco de página 1, cargaría la página virtual 8 en la dirección física 8192 y realizaría dos cambios en la asociación de la MMU. Primero, marcaría la entrada de la página virtual 1 como no asociada, para hacer un trap por cualquier acceso a las direcciones virtuales entre 4096 y 8191. Después reemplazaría la cruz en la entrada de la página virtual 8 con un 1, de manera que al ejecutar la instrucción que originó el trap, asocie la dirección virtual 32780 a la dirección física 4108 ( $4096 + 12$ ).



12 bits. Con 4 bits para el número de página, podemos tener 16 páginas y con los 12 bits para el desplazamiento, podemos direccionar todos los 4096 bytes dentro de una página.

Por si pregunta de forma explícita se usa un tamaño con potencia de 2. Se hace porque tenemos direcciones virtuales de 16, 32, 64 bytes. Las cuales la MMU va a dividir. Pongamos el ejemplo de tener direcciones virtuales de 16 bits, y 4 bits son para indicar el número de página, los 12 bits restantes son para el **offset** de la dirección y nos permiten direccionar  $2^{12} = 4096$  bytes, que para sorpresa de nadie es potencia entera de 2.

Basta con usar un contraejemplo, imaginemos que tenemos páginas de 4097 bytes, siempre nos va a quedar un byte sin poder direccionar. Y si lo queremos direccionar el número de páginas se va a reducir a la mitad y además habrá un hueco gigante de direcciones que no van a ser referenciadas.

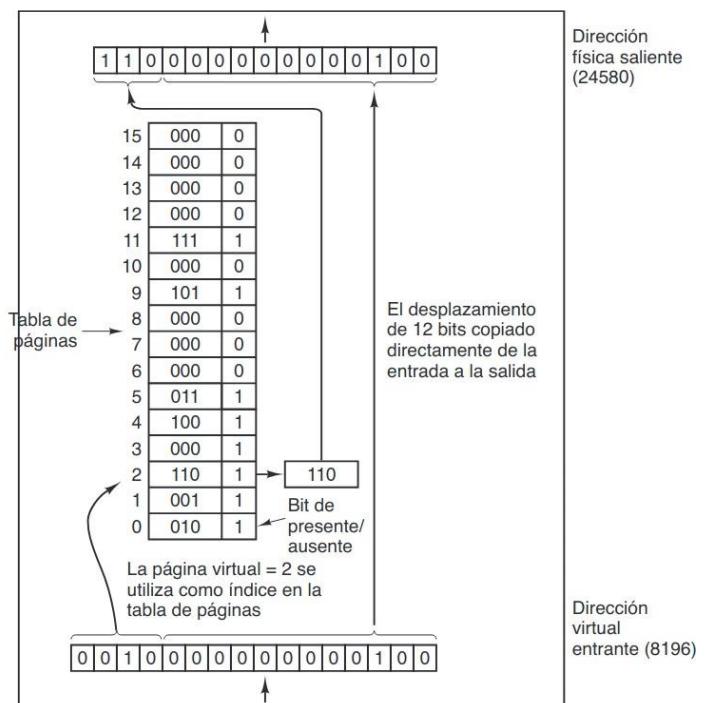


Figura 3-10. La operación interna de la MMU con 16 páginas de 4 KB.  
El desplazamiento de 12 bits copiado directamente de la entrada a la salida

### 3.4.4. Aceleración de la paginación

En cualquier sistema de paginación hay que abordar dos cuestiones principales:

1. La asociación de una dirección virtual a una dirección física debe ser rápida.
2. Si el espacio de direcciones virtuales es grande, la tabla de páginas será grande.

El primer punto es una consecuencia del hecho de que la asociación virtual-a-física debe realizarse en cada referencia de memoria. Todas las instrucciones deben provenir finalmente de la memoria y muchas de ellas hacen referencias a operandos en memoria también. En consecuencia, es necesario hacer una, dos o algunas veces más referencias a la tabla de páginas por instrucción.

El segundo punto se deriva del hecho de que todas las computadoras modernas utilizan direcciones virtuales de por lo menos 32 bits, donde 64 bits se vuelven cada vez más comunes. Por decir, con un tamaño de página de 4 KB, un espacio de direcciones de 32 bits tiene 1 millón de páginas y un espacio de direcciones de 64 bits tiene más de las que desearíamos contemplar. Con 1 millón de páginas en el espacio de direcciones virtual, la tabla de páginas debe tener 1 millón de entradas. Las tablas de páginas normales (ni multinivel ni invertidas) tienen tantas entradas como páginas. Para el tamaño de una TP basta con calcular el número de entradas de esta tabla y lo que ocupa cada entrada.

Tenemos 2 opciones para tratar las TPS:

- Mantener una **única TP** como un conjunto de **registros hardware**, uno para cada página. Cuando se inicia un proceso, el SO carga los registros con la tabla de páginas del proceso almacenada en memoria principal. Esto es muy simple, no requiere acceder a memoria principal durante la traducción. Sin embargo, es muy caro si las tablas de páginas son grandes y las comutaciones de procesos serán más lentas que implicarán la carga de su TP en los registros
- Mantener **una TP por proceso** en la **memoria principal**. Esto es lo que se hace en la mayoría de sistemas modernos. Para realizar una traducción sólo se necesita un registro que apunte el inicio de la TP del proceso en ejecución. Las comutaciones de procesos sólo implican cargar una dirección en un registro. Sin embargo, se requiere acceder a memoria principal para leer la TP cada vez que se tenga que traducir una dirección.

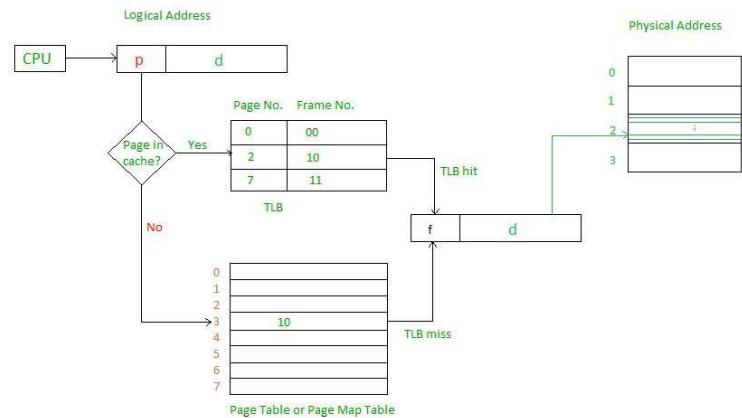
### 3.4.4.1. Búfer de traducción adelantada

Con la paginación se requiere al menos una referencia adicional a memoria para acceder a la tabla de páginas. Como la velocidad de ejecución está comúnmente limitada por la proporción a la que la CPU puede obtener instrucciones y datos de la memoria, al tener que hacer dos referencias a memoria por cada una de ellas se reduce el rendimiento a la mitad. Bajo estas condiciones, nadie utilizaría la paginación. Basicamente cada vez que queremos ver una dirección implica un acceso a memoria a mayores para ver la tabla de páginas.

Válida	Página virtual	Modificada	Protección	Marco de página
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

La solución que se ha ideado es equipar a las computadoras con un pequeño dispositivo de hardware para asociar direcciones virtuales a direcciones físicas sin pasar por la tabla de páginas. El dispositivo se llama **TLB** (*Translation Lookaside Buffer, Búfer de traducción adelantada*) o algunas veces memoria asociativa. Por lo general se encuentra dentro de la MMU y consiste en un pequeño número de entradas, ocho en este ejemplo, pero raras veces más de 64.

Cuando se presenta una dirección virtual a la MMU para que la traduzca, el hardware primero comprueba si su **número de página virtual** está presente en el TLB al compararla con todas las entradas en forma simultánea (es decir, en paralelo mediante comparadores en cada entrada del TLB). Si se encuentra el número de página en la TLB y el bit de validez es 1, se toma el número de página en la TLB **sin pasar por la tabla de páginas**. Así nos ahorraremos un acceso a memoria



Si no se encuentra el número de página en el TLB, la MMU busca el número de página en la TP, **se copia la entrada de la TP en el TLB**, si el TLB está lleno hay que reemplazar alguna de sus entradas. Si el bit de modificado de la entrada desalojada es 1, se establece el bit modificado de la entrada a la TP a 1 también (el algoritmo que usa es desconocido, preguntadle al profesor).

Puede no estar en el TLB, pero si en memoria, por lo que un fallo TLB no siempre implica traer una página de memoria secundaria

Ahora hay máquinas que gestionan el TLB mediante software en el Sistema Operativo, en vez de usar la MMU del procesador, el Sistema Operativo es el que gestiona esto todo. Cuando no se encuentra una coincidencia en el TLB, en vez de que la MMU vaya a las tablas de páginas para buscar y obtener la referencia a la página que se necesita, sólo genera un fallo del TLB (excepción) y pasa el problema al sistema operativo. El sistema debe buscar la página, eliminar una entrada del TLB, introducir la nueva página y reiniciar la instrucción que originó el fallo. Y, desde luego, todo esto se debe realizar en unas cuantas instrucciones, ya que los fallos del TLB ocurren con mucha mayor frecuencia que los fallos de página. Además, reduce el número de fallos.

Esto ayuda a que la MMU sea más simple, y que el chip de la CPU tenga más espacio para implementar cachés y otras características que mejoren el rendimiento. Los fallos de TLB son mucho más frecuentes que los fallos de página.

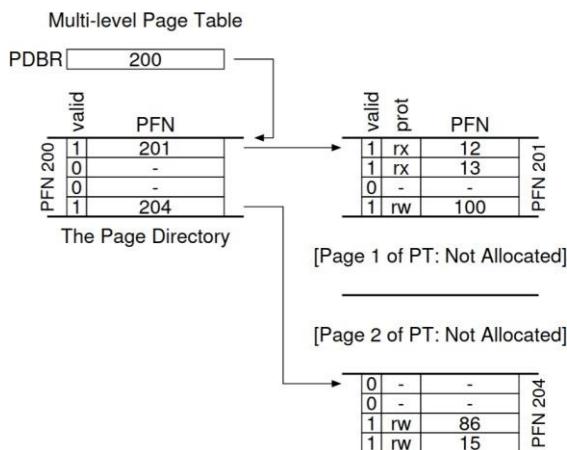
La forma normal de procesar un fallo del TLB, ya sea en hardware o en software, es ir a la tabla de páginas y realizar las operaciones de indexado para localizar la página referenciada. El problema al realizar esta búsqueda en software es que las páginas que contienen la tabla de páginas tal vez no estén en el TLB, lo cual producirá fallos adicionales en el TLB durante el procesamiento.

Estos fallos se pueden reducir al mantener una caché grande en software (por ejemplo, de 4 KB) de entradas en el TLB en una ubicación fija, cuya página siempre se mantenga en el TLB. Al comprobar primero la caché de software, el sistema operativo puede reducir de manera substancial los fallos del TLB.

Cuando se utiliza la administración del TLB mediante software, es esencial comprender la diferencia entre los dos tipos de fallos. Un fallo suave ocurre cuando la página referenciada no está en el TLB, sino en memoria. Todo lo que se necesita aquí es que el TLB se actualice. No se necesita E/S de disco. Por lo general, un fallo suave requiere de 10 a 20 instrucciones de máquina y se puede completar en unos cuantos nanosegundos. Por el contrario, un fallo duro ocurre cuando la misma página no está en memoria (y desde luego, tampoco en el TLB). Se requiere un acceso al disco para traer la página, lo cual tarda varios milisegundos. Un fallo duro es en definitiva un millón de veces más lento que un fallo suave.

### 3.4.4.2. Tablas de páginas multinivel

Como primer método, considere el uso de una tabla de páginas multinivel. En la figura 3.13 se muestra un ejemplo simple. En la figura 3.13(a) tenemos una dirección virtual de 32 bits que se partitiona en un campo *TP1* de 10 bits, un campo *TP2* de 10 bits y un campo *Desplazamiento* de 12 bits. Como los desplazamientos son de 12 bits, las páginas son de 4 KB y hay un total de  $2^{20}$ .



El secreto del método de la tabla de páginas multinivel es **evitar mantenerlas en memoria todo el tiempo**, y en especial, aquellas que no se necesitan. Por ejemplo, suponga que un proceso necesita 12 megabytes: los 4 megabytes inferiores de memoria para el texto del programa, los siguientes 4 megabytes para datos y los 4 megabytes superiores para la pila. Entre la parte superior de los datos y la parte inferior de la pila hay un hueco gigantesco que no se utiliza.

Si recordamos, cargar la tabla de páginas no genera fallo de página, sin embargo, cargar las tablas de páginas de niveles inferiores sí que genera fallos de página (en los enunciados por norma general os va a decir que las tablas de páginas de niveles inferiores ocupan exactamente lo mismo que una página). Además cada proceso tiene su propio conjunto de tablas de páginas formado por su TP1, TP2s y TP3s.

En la figura 3.13(b) podemos ver cómo funciona la tabla de página de dos niveles en este ejemplo. A la izquierda tenemos la tabla de páginas de nivel superior, con 1024 entradas, que corresponden al campo *TP1* de 10 bits. Cuando se presenta una dirección virtual a la **MMU**, primero extrae el campo *TP1* y utiliza este valor como índice en la tabla de páginas de nivel superior. Cada una de estas 1024 entradas representa 4 M, debido a que todo el espacio de direcciones virtuales de 4 gigabytes (es decir, de 32 bits) se ha dividido en trozos de 4096 bytes.

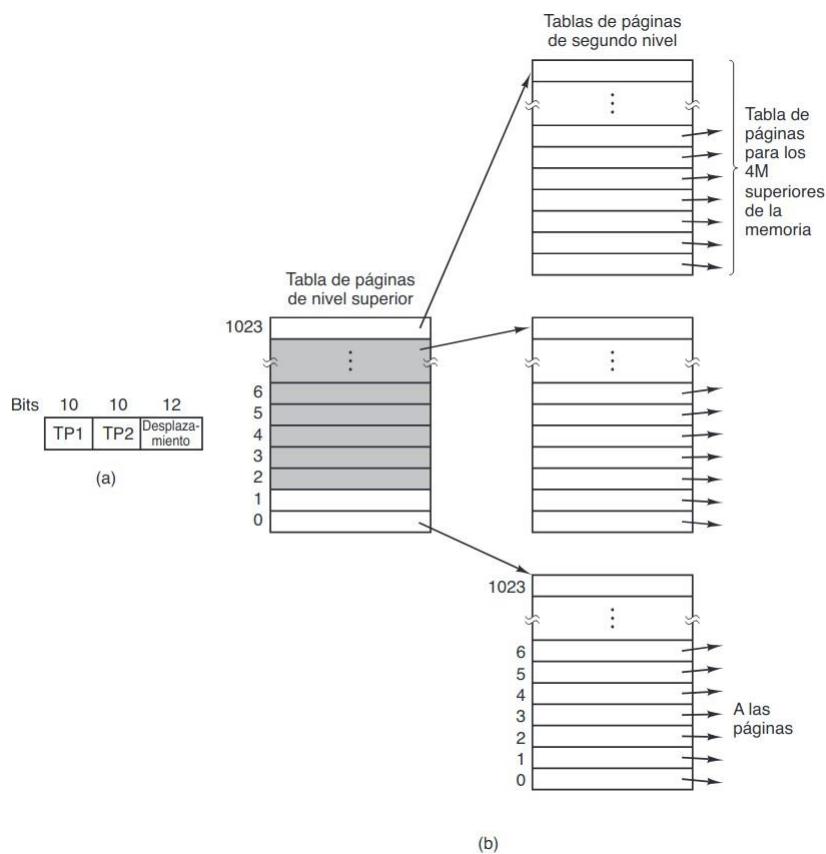
La entrada que se localiza al indexar en la tabla de páginas de nivel superior produce la dirección (o número de marco de página) de una tabla de páginas de segundo nivel. La entrada 0 de la tabla de páginas de nivel superior apunta a la tabla de páginas para el texto del programa, la entrada 1 apunta a la tabla de páginas para los datos y la entrada 1023 apunta a la tabla de páginas para la pila. Las otras entradas (sombreadas) no se utilizan. Ahora el campo *TP2* se utiliza como índice en la tabla de páginas de segundo nivel seleccionada para buscar el número de marco de página para esta página en sí.

Como ejemplo, considere la dirección virtual de 32 bits 0x00403004 (4,206,596 decimal), que se encuentra 12,292 bytes dentro de los datos. Esta dirección virtual corresponde a  $TP1 = 1$ ,  $TP2 = 3$  y *Desplazamiento* = 4. La MMU utiliza primero a *TP1* para indexar en la tabla de páginas de nivel superior y obtener la entrada 1, que corresponde a las direcciones de 4M a 8M. Después utiliza *PT2* para indexar en la tabla de páginas de segundo nivel que acaba de encontrar y extrae la entrada 3, que corresponde a las direcciones de 12288 a 16383 dentro de su trozo de 4M (es decir, las direcciones absolutas de 4,206,592 a 4,210,687). Esta entrada contiene el número de marco de la página que contiene la dirección virtual 0x00403004. Si esa página no está en la memoria, el bit de presente/ausente en la entrada de la tabla de páginas será cero, con lo cual se producirá un fallo de página. Si la página está en la memoria, el número del marco de página que se obtiene de la tabla de páginas de segundo nivel se combina con el desplazamiento (4) para construir la dirección física. Esta dirección se coloca en el bus y se envía a la memoria.

El sistema de tablas de páginas de dos niveles de la figura 3.13 se puede expandir a tres, cuatro o más niveles. Entre más niveles se obtiene una mayor flexibilidad, pero es improbable que la complejidad adicional sea de utilidad por encima de tres niveles.

Resumiendo, cada vez que se le presenta una dirección virtual a la MMU para que la traduzca:

- Se lee el campo de *TP1* de la dirección.
- Se comprueba en la entrada correspondiente de la *TP1* si la *TP2* está en memoria.
- Si la *TP2* correspondiente no está en memoria, se trae del disco como si fuera una página normal.
- Si la *TP2* correspondiente sí que está en memoria se lee el campo *TP2* de la dirección, se comprueba la entrada correspondiente de la *TP2* si la página está en memoria y se procede con normalidad.



**FIGURA 3.13.** (a) Una dirección de 32 bits con dos campos de tablas de páginas.

(b) Tablas de páginas de dos niveles.

### 3.4.4.3. Tablas de páginas invertidas

Es única para todo el sistema y siempre está en memoria.

Para los espacios de direcciones virtuales de **32 bits**, la **tabla de páginas multinivel funciona bastante bien**. Sin embargo, a medida que las computadoras de **64 bits** se hacen más comunes, la situación cambia de manera drástica. Si el espacio de direcciones es de  $2^{64}$  bytes, con páginas de 4 KB, necesitamos una tabla de páginas con  $2^{52}$  entradas. Si cada entrada es de 8 bytes, la tabla es de más de 30 millones de gigabytes (30 PB). Ocupar 30 millones de gigabytes sólo para la tabla de páginas no es una buena idea por ahora y probablemente tampoco lo sea para el próximo año, se necesita una solución diferente para los espacios de direcciones virtuales paginados de 64 bits.

Una de esas soluciones es la **tabla de páginas invertida**. En este diseño hay **una entrada por cada marco de página en la memoria real**, en vez de tener una entrada por página de espacio de direcciones virtuales. Por ejemplo, con direcciones virtuales de 64 bits, una página de 4 KB y 1 GB de RAM, una tabla de páginas invertida sólo requiere 262,144 entradas. La entrada lleva el registro de quién (proceso, página virtual) se encuentra en el marco de página.

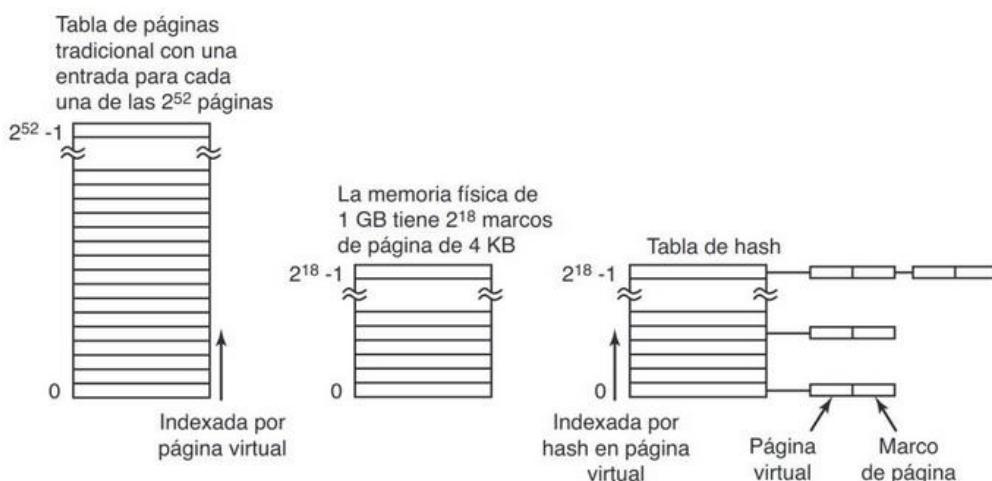
Aunque las tablas de página invertidas ahorran grandes cantidades de espacio, tienen una seria desventaja: **la traducción de dirección virtual a dirección física se hace mucho más difícil**. Cuando el proceso  $n$  hace referencia a la página virtual  $p$ , el hardware ya no puede buscar la página física usando  $p$  como índice en la tabla de páginas. En vez de ello, debe buscar una entrada  $(n, p)$  en toda la tabla de páginas invertida. La forma de salir de este dilema es utilizar el TLB. Si el TLB puede contener todas las páginas de uso frecuente, la traducción puede ocurrir con igual rapidez que con las tablas de páginas regulares. Sin embargo, en un fallo de TLB la tabla de páginas invertida tiene que buscarse mediante software. Una manera factible de realizar esta búsqueda es tener una tabla de hash arreglada según el hash de la dirección virtual. Todas las páginas virtuales que se encuentren en memoria y tengan el mismo valor de hash se encadenan en conjunto.

Pregunta muy típica del profesor en exámenes sobre si ocupan mas las TP normales o las invertidas, y como todo en la asignatura se responde con un “depende” o “por norma general”.

$$\frac{\text{TamMemVirtual}}{\text{TamPag}} = \frac{\text{TamMemFisica}}{\text{TamMarco}}$$

$$\text{TamTP} = \text{TamTPInv}$$

De esto se deduce que para darse esta situación el tamaño de la memoria física y la virtual deben ser iguales. Después por ejemplo siempre tiene otra mítica sobre una RAM de 12 Gb o algo así, y claro lo más normal es tener una RAM múltiplo de 2, creo que la respuesta correcta es decir que no es un tamaño habitual pero que sí que es usable.



**FIGURA 3.14.** Comparación de una tabla de páginas tradicional con una tabla de páginas invertida.

## 3.5. Algoritmos de reemplazo de páginas

Cuando ocurre un **fallo de página** y la memoria principal está llena, se usará un ALGORITMO DE REEMPLAZO para decidir qué página se desalojará para cargar la nueva en el marco que ocupaba.

Interesa no eliminar una **página de uso frecuente**, pues seguramente se referenciará otra vez rápidamente y habrá que volver a cargarla desde disco, lo cual es una operación muy lenta.

Existen dos tipos de algoritmos de reemplazo:

- **Algoritmos locales** → siempre seleccionarán una página del mismo proceso que la que se está intentando cargar.
- **Algoritmos globales** → pueden seleccionar una página de cualquier proceso.

El problema del reemplazo de páginas aparece también en otras áreas del diseño computacional como las memorias caché y los servidores web. En las memorias caché, la escala temporal del problema será mucho menor ya que sus fallos se resuelven en memoria principal, que es mucho más rápida que el disco.

### 3.5.1. El algoritmo de reemplazo de páginas óptimo

Selecciona la página que se vaya a **referenciar más tarde** con objeto de **posponer el fallo** de página lo máximo posible.

Es **imposible de implementar**, pues el SO no tiene manera de saber cuándo será la próxima referencia a cada página. Se puede implementar en un simulador en la segunda corrida utilizando la información de referencia de páginas recolectada durante la primera.

Se usa como **referencia**, comparándolo con otros algoritmos que sí son implementables para averiguar su rendimiento.

### 3.5.2 No Usadas Recientemente

Para permitir que el sistema operativo recolecte estadísticas útiles sobre el uso de páginas, la mayor parte de las computadoras con memoria virtual tienen dos bits de estado asociados a cada página.  $R$  se establece cada vez que se hace referencia a la página (lectura o escritura);  $M$  se establece cuando se escribe en la página (es decir, se modifica). Los bits están contenidos en cada entrada de la tabla de páginas. Es importante tener en cuenta que estos bits se deben actualizar en cada referencia a la memoria, por lo que es imprescindible que se establezcan **mediante el hardware**. Una vez que se establece un bit en 1, permanece así hasta que el **sistema operativo lo restablece**. El bit  $R$  se borra en forma periódica (en cada interrupción de reloj) para diferenciar las páginas a las que no se ha hecho referencia recientemente de las que si se han referenciado.

Cuando ocurre un fallo de página, el sistema operativo inspecciona todas las páginas del proceso 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.

Aunque las páginas de la clase 1 parecen a primera instancia imposibles, ocurren cuando una interrupción de reloj borra el bit  $R$  de una página de la clase 3. Las interrupciones de reloj no borran el bit  $M$  debido a que esta información se necesita para saber si la página se ha vuelto a escribir en el disco o no. Al borrar  $R$  pero no  $M$  se obtiene una página de clase 1.

Se puede tener una página cargada aunque no haya sido referenciada, lo que cuenta para ver si se produce un fallo o no es el **bit de váliduz**.

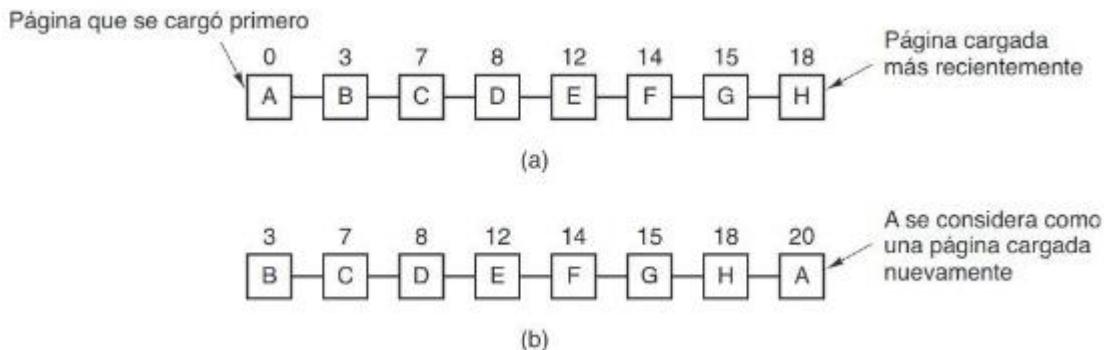
El algoritmo **NRU** (*Not Recently Used*, No usada recientemente) elimina una página al azar de la clase de menor numeración que no esté vacía. En este algoritmo está implícita la idea de que es mejor eliminar una página modificada a la que no se haya hecho referencia en al menos un pulso de reloj que una página limpia de uso frecuente. La principal atracción del NRU es que es fácil de comprender, moderadamente eficiente de implementar y proporciona un rendimiento que, aunque no es óptimo, puede ser adecuado.

### 3.5.3. Primera en Entrar, Primera en Salir (FIFO)

Otro algoritmo de paginación con baja sobrecarga es el de **Primera en entrar, primera en salir** (*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 fallo de página, se elimina la página que está en la parte frontal y la nueva página se agrega a la parte final de la lista. Se podría reemplazar una página de uso frecuente, por esta razón es raro que se utilice FIFO en su forma pura.

### 3.5.4. Segunda oportunidad

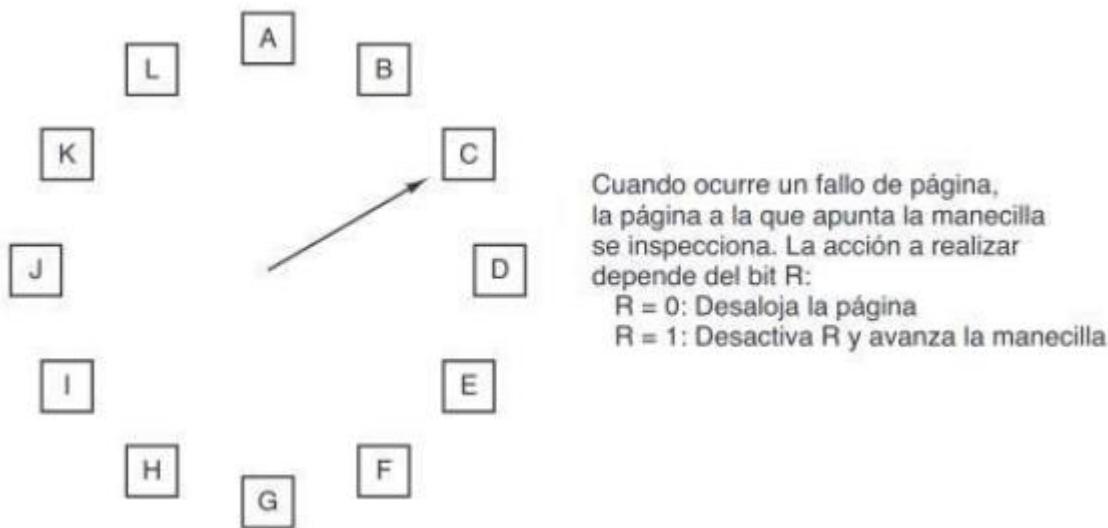
Una modificación simple al algoritmo FIFO que evita el problema de descartar una página de uso frecuente es inspeccionar el bit *R* de la página más antigua. Si es 0, la página es antigua y no se ha utilizado, por lo que se sustituye de inmediato. Si el bit *R* es 1, el bit se borra, la página se pone al final de la lista de páginas y su tiempo de carga se actualiza, como si acabara de llegar a la memoria. Después la búsqueda continúa. Este algoritmo se conoce como **segunda oportunidad**.



### 3.5.5. Reloj

Aunque el algoritmo segunda oportunidad es razonable, también es innecesariamente ineficiente debido a que está moviendo constantemente páginas en su lista. Un mejor método sería mantener todos los marcos de página en una lista circular en forma de reloj. La manecilla apunta a la página más antigua.

- Si  $R = 0 \rightarrow$  se selecciona, se inserta la nueva página en su lugar y se avanza la manecilla una posición.
- Si  $R = 1 \rightarrow$  se restablece  $R$  y se avanza la manecilla una posición.



### 3.5.6. Menos Usadas Recientemente

Se basa en el principio de **localidad temporal**, seleccionando la página que lleve **más tiempo sin ser referenciada**. Tiene varias posibles implementaciones.

Implementación por **software**:

- Se crea una lista de páginas en memoria donde la última que ha sido usada estará al principio y la que más lleve sin usarse estará al final.
- En cada referencia la página referenciada se mueve al frente de la lista.
- La página a seleccionar será la primera de la lista.
- Es una implementación muy cara, pues en cada referencia hay que buscar la página en la lista, eliminarla y después pasarlal al frente.

Implementación por **hardware sencilla**:

- Se crea un contador  $C$  que se incrementa después de cada instrucción.
- En cada referencia el valor actual de  $C$  se almacena en la entrada de la TP de la página referenciada.
- La página a seleccionar será aquella con menor valor de  $C$ .

Implementación por **hardware sofisticada**:

- Se crea una matriz  $n \times n$  inicializada a 0, donde  $n$  es el número de marcos de la memoria principal.
- En cada referencia se establecen todos los bits de la fila correspondiente a la página referenciada a 1 y todos los de la columna a 0.
- La página a seleccionar será aquella con el menor valor binario almacenado en su fila.

Página																			
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	1	1	1	0	0	1	0	0	0	1	0	0	0
2	0	0	0	0	0	0	0	0	1	1	0	1	1	1	0	1	1	0	1
3	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0

(a)	(b)	(c)	(d)	(e)																																																																																
<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	1	0	1	1	1	0	0	1	1	0	0	0	<table border="1"> <tr><td>0</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	1	1	1	0	0	1	1	0	0	0	1	0	0	0	0	<table border="1"> <tr><td>0</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td></tr> </table>	0	1	1	0	0	0	1	0	0	0	0	0	1	1	1	0	<table border="1"> <tr><td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	0	1	0	0	0	0	0	0	1	1	0	1	1	1	0	0	<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td></tr> </table>	0	0	0	0	1	0	0	0	1	1	0	1	1	1	1	0
0	0	0	0																																																																																	
1	0	1	1																																																																																	
1	0	0	1																																																																																	
1	0	0	0																																																																																	
0	1	1	1																																																																																	
0	0	1	1																																																																																	
0	0	0	1																																																																																	
0	0	0	0																																																																																	
0	1	1	0																																																																																	
0	0	1	0																																																																																	
0	0	0	0																																																																																	
1	1	1	0																																																																																	
0	1	0	0																																																																																	
0	0	0	0																																																																																	
1	1	0	1																																																																																	
1	1	0	0																																																																																	
0	0	0	0																																																																																	
1	0	0	0																																																																																	
1	1	0	1																																																																																	
1	1	1	0																																																																																	
(f)	(g)	(h)	(i)	(j)																																																																																

**FIGURA 3.17.** LRU usando una matriz cuando se hace referencia a las páginas en el orden 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

### 3.5.7. No Usadas Frecuentemente (NFU)

Es una variante del LRU que sí se puede implementar en software.

Se basa en asignar a cada página un contador (iniciado a 0) de manera que en cada interrupción de reloj el SO explora todas las páginas en memoria y le suma a su contador el valor de su bit *R* antes de restablecerlo. Así, los contadores llevan una cuenta aproximada de la frecuencia con la que se usa cada página.

Su principal **problema** es que no tiene en cuenta el tiempo que lleva sin ser utilizada cada página, sólo cuántas veces se ha usado. Entonces, páginas antiguas que se usaron hace mucho tiempo y ya no se necesitan más tendrán privilegios sobre páginas nuevas que aún no tuvieron tiempo de ser usadas pero que la CPU necesita.

**Envejecimiento:** cuando llega una interrupción de reloj, cada contador se desplaza hacia la derecha y se le agrega el bit *R* a la izquierda. Se selecciona la página de menor contador.

Bits R para las páginas 0 a 5, pulso de reloj 0		Bits R para las páginas 0 a 5, pulso de reloj 1		Bits R para las páginas 0 a 5, pulso de reloj 2		Bits R para las páginas 0 a 5, pulso de reloj 3		Bits R para las páginas 0 a 5, pulso de reloj 4	
1	0	1	0	1	1	1	1	0	0
Página									
0	1000000	1100000	1110000	1111000	0111100				
1	0000000	1000000	1100000	0110000	1011000				
2	1000000	0100000	0010000	0010000	1001000				
3	0000000	0000000	1000000	0100000	0010000				
4	1000000	1100000	0110000	1011000	0101100				
5	1000000	0100000	1010000	0101000	0010100				

(a)	(b)	(c)	(d)	(e)

**FIGURA 3.18.** El algoritmo de envejecimiento simula el LRU en software. Aquí se muestran seis páginas para cinco pulsos de reloj. Los cinco pulsos de reloj se representan mediante los incisos (a) a (e).

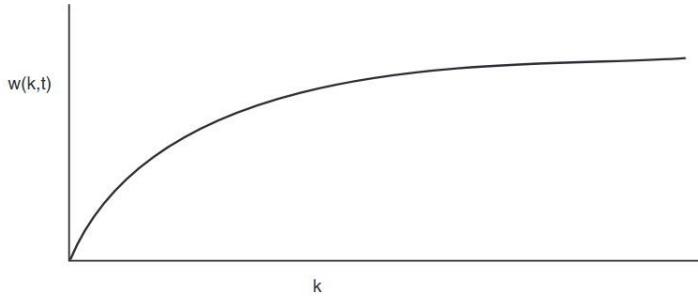
### 3.5.8. Conjunto de Trabajo

El working set es el conjunto de páginas a las que accede y utiliza un proceso en un momento dado. A más se cumpla el principio de localidad temporal (visto en Fundamentos de Computadores) en su ejecución, menor será el tamaño del Working Set al estar los datos disponibles de forma más compacta y necesitar por tanto un menor número de páginas. Si se cumple más el principio de localidad espacial, al traer más páginas a memoria principal cercanas a las anteriormente utilizadas, el working set aumentará de tamaño

En los algoritmos explicados antes se usa **paginación bajo demanda**, es decir, las páginas se cargan según las va pidiendo la CPU, no por adelantado.

Sin embargo, sabemos que la mayoría de procesos cumplen el **principio de localidad espacial** (o de referencia), que asegura que referencian una pequeña fracción de sus páginas. Esta propiedad se puede aprovechar con el concepto del CONJUNTO DE TRABAJO (WS), que es el conjunto de páginas que **referencia un proceso en un momento dado**.

- Si todo el WS de un proceso está en memoria → se ejecutará sin producir muchos fallos de página.
- Si no todo el WS de un proceso está en memoria → producirá fallos de página cada pocas instrucciones (estará **sobrepaginando**, teniendo muchos fallos de página), así que se ejecutará muy lentamente. La **sobrepaginación** sucede cuando el WS no cabe en memoria principal, bien porque esta es muy pequeña o porque el proceso referencia muchas páginas distintas.



**Figura 3-19.** El conjunto de trabajo es el conjunto de páginas utilizadas por las  $k$  referencias a memoria más recientes. La función  $w(k, t)$  es el tamaño del conjunto de trabajo en el tiempo  $t$ .

Definimos el  $w(k, t)$  de un proceso en el instante  $t$  como el conjunto de páginas utilizadas en sus últimas  $k$  referencias. Su límite conforme  $k$  avanza con un  $t$  dado es finito, pues un proceso no puede referenciar más páginas que las que hay en su espacio de direcciones.

Entonces, cada vez que el SO traiga un proceso a memoria para ejecutarlo, este producirá fallos de página hasta que se haya cargado su WS, desperdiциando tiempo de CPU. Para evitar este desperdicio se usa el MODELO DEL CONJUNTO DE TRABAJO, que consiste en asegurarse de que el WS de un proceso esté en memoria antes de comenzar a ejecutarlo.

- Se usa la **prepaginación**, se cargan las páginas del WS del proceso en memoria antes de empezar a ejecutarlo, es decir, antes de que las solicite la CPU.
- El **algoritmo de reemplazo** seleccionará las páginas que no se encuentran en el WS del proceso.

Para poder implementar este modelo, el SO necesitará conocer el WS que tiene cada proceso en todo momento. Una implementación que mantenga la definición de  $w(k, t)$  de manera estricta es **inviable**, pues implica actualizarlo en cada referencia a memoria. Se podría usar un registro de desplazamiento de longitud  $k$  que se corresponda con el  $w(k, t)$  del proceso. En cada referencia a memoria se desplaza una posición a la izquierda e inserta el número de la página referenciada a la derecha. Sin embargo, para hacer esto habría que recorrerlo, quitar todas las páginas duplicadas y ordenarlas otra vez en cada referencia, lo que sería muy costoso.

En su lugar, se usa una aproximación del modelo en la que se **redefine el WS** como el conjunto de páginas utilizadas durante los últimos  $\tau$  segundos de tiempo virtual (tiempo de ejecución) del proceso.

El algoritmo de reemplazo basado en WS, como se dijo antes, selecciona una página que no esté en el WS. Cada entrada de la TP contendrá, al menos, el tiempo virtual en el que la página fue usada por última vez, el bit  $R$  y el bit  $M$ . Partimos de que  $\tau$  abarca varios pulsos de reloj. En cada fallo de página, se recorre la TP observando el bit  $R$ :

- Si  $R = 1 \rightarrow$  la página fue usada en el último pulso de reloj, por lo que está en el WS, así que no es candidata.
- Si  $R = 0 \rightarrow$  la página no fue usada en el último pulso de reloj, así que puede ser que sea candidata o no.

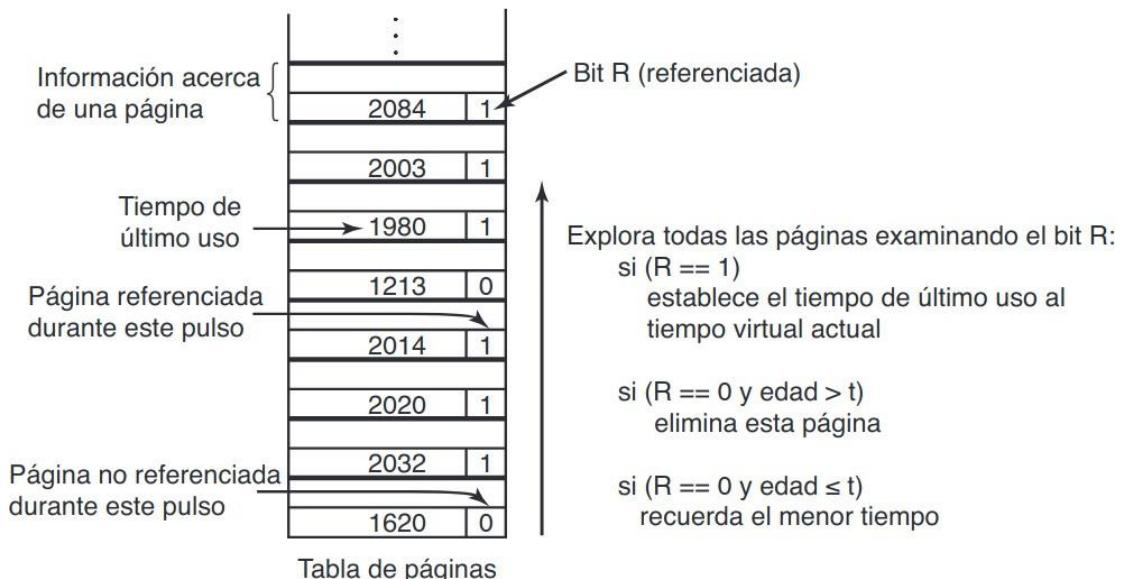
Se calcula la *edad* de la página como *tiempo virtual actual – tiempo de último uso*.

- Si  $edad < \tau \rightarrow$  está en el WS, así que no es candidata.
- Si  $edad > \tau \rightarrow$  no está en el WS, se selecciona para desalojar.

Si se explora toda la tabla sin encontrar ninguna página apta para seleccionarse, es que todas ellas están en el WS. Entonces, se seleccionará la página con  $R = 0$  más antigua. Si todas tienen  $R = 1$ , se seleccionará una al azar (de preferencia una con  $M = 0$ , si existe alguna).

$$edad = TVirtualActual - TUltimoUso$$

2204 Tiempo virtual actual



**FIGURA 3.20.** El algoritmo del conjunto de trabajo.

### 3.5.9. El algoritmo de reemplazo de páginas WSClock (Working Set Clock)

El algoritmo de reemplazo basado en WS básico es ineficiente pues implica explorar toda la TP de un proceso cada vez que se da un fallo de página. Para evitar esto, usaremos una lista circular en la que se almacenarán las entradas de la TP de las páginas que son residentes en memoria en un momento dado. En cada fallo de página, se recorre la lista observando los bits  $R$  y  $M$ :

- Si  $R = 1 \rightarrow$  la página fue usada en el último pulso de reloj, por lo que está en el WS, así que no es candidata.

1. Se restablece el bit  $R$ .
2. Se avanza la manecilla una posición.

- Si  $R = 0 \rightarrow$  la página no fue usada en el último pulso de reloj, así que puede estar en el WS o no.

- Si  $edad < \tau \rightarrow$  está en el WS, así que no es candidata.
- Si  $edad > \tau \rightarrow$  no está en el WS, así que es candidata.
  - Si  $M = 0 \rightarrow$  no está en el WS y tiene una copia en el disco, se selecciona para desalojar.
  - Si  $M = 1 \rightarrow$  no está en el WS pero no tiene una copia en el disco.
    1. Se planifica la escritura en el disco.
    2. Se avanza la manecilla una posición.

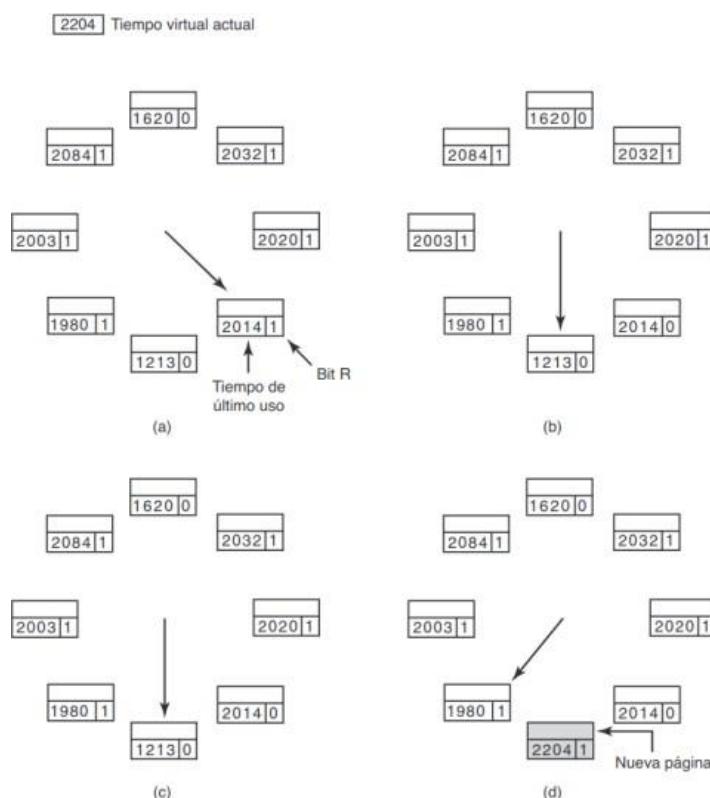


Figura 3-21. Operación del algoritmo WSClock. (a) y (b) dan un ejemplo de lo que ocurre cuando  $R = 1$ . (c) y (d) dan un ejemplo de cuando  $R = 0$ .

### 3.5.10. Resumen de los algoritmos de reemplazo de páginas

Algoritmo	Comentario
Óptimo	No se puede implementar, pero es útil como punto de comparación
NRU (No usadas recientemente)	Una aproximación muy burda del LRU
FIFO (primera en entrar, primera en salir)	Podría descartar páginas importantes
Segunda oportunidad	Gran mejora sobre FIFO
Reloj	Realista
LRU (menos usadas recientemente)	Excelente, pero difícil de implementar con exactitud
NFU (no utilizadas frecuentemente)	Aproximación a LRU bastante burda
Envejecimiento	Algoritmo eficiente que se aproxima bien a LRU
Conjunto de trabajo	Muy costoso de implementar
WSClock	Algoritmo eficientemente bueno

## 3.6. Cuestiones de diseño para los sistemas de paginación

### 3.6.1. Asignación Local contra Asignación Global

Este punto no tiene que ver con los algoritmos de reemplazo, solo con el número de marcos asignados a cada proceso.

En las políticas de **asignación global** el algoritmo de reemplazo siempre desalojará una página del mismo proceso que la que se está intentando cargar. Asignan a cada proceso una **cantidad fija de marcos** de memoria. Si en algún momento el tamaño del WS supera el número de marcos asignados, se produce sobrepaginación. Si en algún momento el tamaño del WS es menor que el número de marcos asignados se estará desperdiando memoria principal.

En las políticas de **asignación local** el algoritmo de reemplazo puede desalojar una página de cualquier proceso. Asignan a cada proceso una **cantidad dinámica de marcos** de memoria que cambia conforme a este se ejecuta. Se pueden ajustar al tamaño del WS **conforme este va cambiando**. Inicialmente, se asigna a cada proceso una cantidad de marcos proporcional a su tamaño. Se garantiza un número **mínimo** de marcos para que los procesos muy pequeños se puedan ejecutar.

Las asignaciones aumentan o disminuyen conforme se ejecutan todos los procesos. El **algoritmo PFF** (frecuencia de fallos de página) es el encargado de administrar las asignaciones, indicando cuándo se deben incrementar o decrementar. La proporción de fallos disminuye conforme se asignan más páginas, así que el

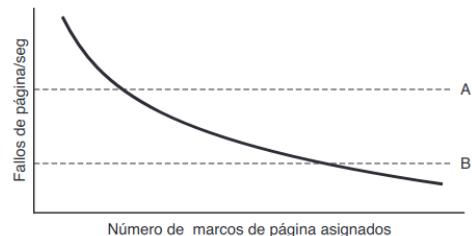


Figura 3-24. Proporción de fallos de página como una función del número de marcos de página asignados.

PFF se basa en media el número de fallos por segundo de cada proceso. En base a esto, establece un rango aceptable de proporción de fallos en el que intenta mantener todos los procesos.

- El límite *A* establece una proporción de fallos demasiado alta: cuando un proceso lo sobrepasa se incrementa su número de marcos asignados.

- El límite *B* establece una proporción de fallos tan baja que se puede suponer que el proceso tiene demasiada memoria: cuando un proceso lo sobrepasa se decrementa su número de marcos asignados.

Algunos algoritmos de reemplazo, como FIFO o LRU, pueden funcionar con **asignación local o global**. Otros, los **basados en WS**, sólo pueden funcionar con **asignación local**. Por definición, no existe un WS para todo el sistema, sólo para un proceso concreto. Al intentar definir un WS total se perdería el fundamento del WS, el principio de localidad.

### 3.6.2. Control de carga

Aunque se use mejor algoritmo de reemplazo y una asignación óptima de marcos, el sistema sobrepaginará si la suma de los WS de todos los procesos en ejecución es más grande que la memoria principal. Un síntoma de que ha sucedido este es que el PFF determine que algunos procesos en memoria necesitan más marcos.

La única solución es deshacerse temporalmente de algunos procesos **bloqueados, intercambiándolos a disco** para liberar sus marcos y que los puedan usar los que los necesitan. Si aun así el sistema sigue sobrepaginando, se volverá a hacer hasta que deje de estarlo.

Al aplicar esto se debe tener en cuenta el **grado de multiprogramación**, pues si el número de procesos en memoria es demasiado bajo, la CPU puede estar inactiva durante largos períodos de tiempo.

### 3.6.3. Tamaño de página

El tamaño de página es un parámetro que a menudo el sistema operativo puede elegir. Para determinar el mejor tamaño de página se requiere balancear varios factores competitivos. Como resultado, no hay un tamaño óptimo en general. Para empezar, hay dos factores que están a favor de un tamaño de página pequeño. Un **segmento** de texto, datos o pila elegido al azar no llenará un número integral de páginas. En promedio, la mitad de la página final estará vacía. El espacio adicional en esa página se desperdicia. A este desperdicio se le conoce como **fragmentación interna**. Con  $n$  segmentos en memoria y un tamaño de página de  $p$  bytes, se desperdiciarán  $np/2$  bytes en fragmentación interna. Este razonamiento está a favor de un tamaño de página pequeño.

Por otro lado, tener páginas pequeñas implica que los programas necesitarán muchas páginas, lo que sugiere la necesidad de una tabla de páginas grande. Las transferencias hacia y desde el disco son por lo general de una página a la vez, y la mayor parte del tiempo se debe al retraso de búsqueda y al retraso rotacional, por lo que para transferir una página pequeña se requiere casi el mismo tiempo que para transferir una página grande.

En algunas máquinas, la tabla de páginas se debe cargar en registros de hardware cada vez que la CPU cambia de un proceso a otro. En estas máquinas, tener un tamaño pequeño de página significa que el tiempo requerido para cargar sus registros aumenta a medida que se hace más pequeña. Además, el espacio ocupado por la tabla de páginas aumenta a medida que se reduce el tamaño de las páginas.

Este último punto se puede analizar matemáticamente. Digamos que el tamaño promedio de un proceso es de  $s$  bytes y que el tamaño de página es de  $p$  bytes. Además supone que cada entrada de página requiere  $e$  bytes. El número aproximado de páginas necesarias por proceso es entonces  $s/p$ , ocupando  $se/p$  bytes de espacio en la tabla de páginas. La memoria desperdiciada en la última página del proceso debido a la fragmentación

interna es  $p/2$ . Así, la sobrecarga total debido a la tabla de páginas y a la pérdida por fragmentación interna se obtiene mediante la suma de estos dos términos:

$$\text{sobrecarga} = se/p + p/2$$

Para obtener el tamaño óptimo de página calculamos la derivada respecto de  $p$  e igualamos a 0 obteniendo:

$$p = \sqrt{2se}$$

Parece coña, pero preguntó en un final de donde salía esta fórmula.

### 3.6.4. Espacios separados de instrucciones y de datos

Todo el rato el tanembaum lo llama segmentos, lo que es un lío tremendo porque nada tiene que ver, como veremos después hay paginación, segmentación (dónde hay segmentos) y segmentación con paginación (dónde hay segmentos paginados).

En la mayoría de sistemas cada proceso tiene **un único espacio** de direcciones que contiene tanto instrucciones como datos. A menudo, este espacio es demasiado pequeño para contener toda esa información.

Para solucionar esto se crean **dos espacios** de direcciones, el **espacio I** (para las instrucciones) y el **espacio D** (para los datos). Cada uno de ellos tendrá el mismo tamaño que el original  $2^n$ , por lo que se **duplicará el espacio** de direcciones disponible. Ambos espacios se **pagan de manera independiente**. Cada uno tiene su propia TP con su propia asignación de páginas a marcos.

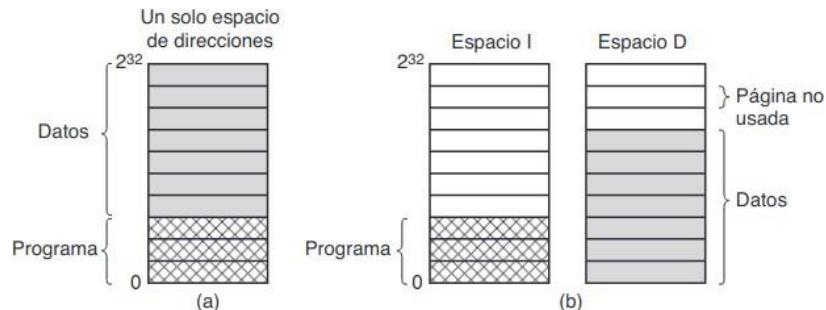


Figura 3-25. (a) Un espacio de direcciones. (b) Espacios I y D separados.

### 3.6.5. Páginas compartidas

Cuando varios usuarios ejecutan el mismo programa, compartir sus páginas es mucho más eficiente que tener dos copias de la misma página en memoria todo el tiempo. Compartir las **páginas de código** (que son sólo de lectura) es **fácil**:

- Si se admiten espacios de direcciones separados, los procesos con el mismo programa usarán la misma TP del espacio I y otra para la TP del espacio D. Para compartir la TP del espacio I, se hará que sus apuntadores apunten a la misma tabla.
- Si no hay espacios de direcciones separados, también se puede conseguir que dos procesos compartan las páginas de código, pero el mecanismo es más complicado.
- El **problema** es que al terminar o intercambiar uno de los procesos que comparten código, se retirarán todas sus páginas de memoria, provocando que el otro tenga muchos fallos de página hasta que vuelva a traer su código. Buscar en todas las TPs de todos los procesos antes de desalojar una página es muy caro, por lo que se usan estructuras de datos especiales para llevar cuenta de las páginas compartidas.

Compartir las **páginas de datos** (que son de lectura y escritura) es más **difícil**. Esto sucede en los proceso padre e hijo, que comparten tanto el código como los datos del programa.

- Cada uno tendrá su propia TP, que apuntará al mismo conjunto de páginas para evitar tener que copiarlas.
- Tan pronto como cualquiera de los dos realice alguna escritura, se hará una copia de la página en la que se escribe. Esto se conoce como **copiar en escritura**. Esto implica crear una página nueva.
- Así, aquellas páginas que nunca se modifican (incluyendo las del programa) no se copiarán.

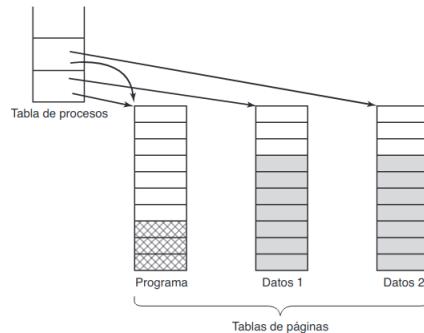
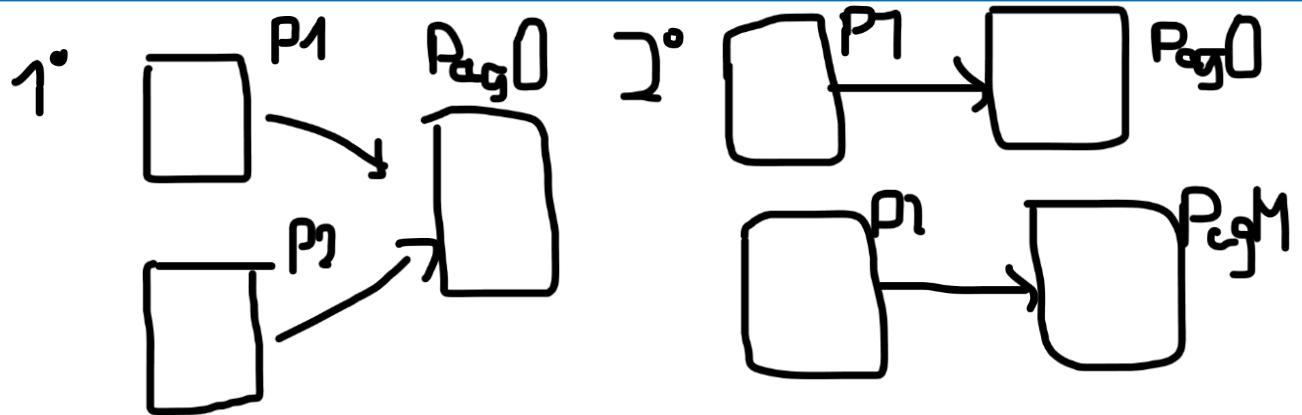


Figura 3-26. Dos procesos comparten el mismo programa compartiendo su tabla de páginas.

#### Copia en escritura:



### 3.6.6. Bibliotecas compartidas

En los sistemas modernos hay muchas **bibliotecas extensas** utilizadas por **muchos procesos**. Se pueden enlazar a los programas que las usan de dos maneras:

**Enlace estático:** las funciones usadas en las bibliotecas se incluyen en tiempo de compilación en el binario ejecutable:

- Las funciones de la biblioteca que no se llaman en el programa no se incluyen.
- El binario ejecutable contiene todo el código necesario para ejecutarse de manera independiente.
- Sin embargo, si muchos programas usan la misma biblioteca, se desperdicia mucho espacio tanto en el disco como en la memoria principal.

**Enlace dinámico:** durante la compilación se incluye una pequeña rutina auxiliar para enlazar las funciones de la biblioteca al llamarlas en tiempo de ejecución.

- Las funciones de la biblioteca se cargan en memoria cuando algún programa las llama por primera vez, por lo que las no se llaman nunca no se cargan.
- Una vez algún proceso las haya cargado, ya las pueden usar todos los demás que la tengan enlazada dinámicamente.

### 3.6.7. Archivos asociados

Un proceso puede emitir una llamada al sistema para **asociar un archivo a una porción de su espacio** de direcciones virtuales. El archivo no se carga entero al momento de la asociación, se va cargando bajo demanda. Los archivos asociados se usan como:

- **Método alternativo para la E/S** → en vez de realizar lecturas y escrituras se puede acceder al archivo como un array de datos en memoria.
- **Canal de comunicación entre procesos** → si varios procesos se asocian a la vez al mismo archivo, se pueden comunicar a través de la memoria que comparten. Las escrituras que realice uno de ellos en la memoria compartida serán inmediatamente visibles por el resto.

### 3.6.8. Política de limpieza

La **paginación** funciona **mejor** cuando hay **muchos marcos libres** que se pueden reclamar nada más ocurrir un fallo. Si todos los marcos están ocupados y han sido modificados, cuando se produzca un fallo se tendrá que realizar una copia en el disco antes de cargar la página que se necesitaba.

En el tanmebaum no se entiende nada así que tras un correo el profesor explicó lo siguiente:

El demonio de paginación es un proceso que periódicamente comprueba los marcos que son usando en RAM. Si determina que está muy llena, intenta seleccionar algunos de los ocupados por alguna página de algún proceso para liberar el correspondiente marco. Si ha sido modificada tarda un cierto tiempo en liberarla porque debe escribir en el disco. La idea es evitar fallos en el futuro. Sirve para actualizar en disco copias de páginas modificadas en RAM en momentos en los que no se usa el disco. La página deja de tener el bit M activado, lo que es bueno.

El algoritmo de reemplazo de páginas usando los marcos libres, si el algoritmo decide usar uno de estos liberados por el demonio, la información pasa al disco, pero mientras eso no ocurra la información sigue en la RAM y está disponible. Quitar una página no implica borrarla, su contenido sigue ahí hasta que no sea reemplazada (sobrescrita) por otra página.

Resumidamente el demonio escribe en disco páginas modificadas y el algoritmo de reemplazo ya determina qué reemplazar.

### **3.7. Cuestiones de implementación (no entra, pero al que le interese saberlo muy bien que nosotros nos enteramos 4 días antes del final).**

#### **3.7.1. Participación del sistema operativo en la paginación**

Hay cuatro ocasiones en las que el sistema operativo tiene que realizar trabajo relacionado con la paginación: al crear un proceso, al ejecutar un proceso, al ocurrir un fallo de página y al terminar un proceso.

Cuando se crea un proceso en un sistema de paginación, el sistema operativo tiene que determinar qué tan grandes serán el programa y los datos (al principio), y **crear una tabla de páginas para ellos**. Se debe **asignar espacio en memoria para la tabla de páginas y se tiene que inicializar**. La tabla de páginas **no necesita estar residente cuando el proceso se intercambia hacia fuera, pero tiene que estar en memoria cuando el proceso se está ejecutando**. Además, se debe asignar espacio en el área de intercambio en el disco, para que cuando se intercambie una página, tenga un lugar a donde ir. El área de intercambio también se tiene que inicializar con el texto del programa y los datos, para que cuando el nuevo proceso empiece a recibir fallos de página, las páginas se puedan traer. Algunos sistemas paginan el texto del programa directamente del archivo ejecutable, con lo cual se ahorra espacio en disco y tiempo de inicialización. Por último, **la información acerca de la tabla de páginas y el área de intercambio en el disco se debe registrar en la tabla de procesos**. Como dijimos anteriormente, la tabla de procesos tiene punteros a las tablas de páginas.

Cuando un proceso se planifica para ejecución, el TLB se vacía para deshacerse de los restos del proceso que se estaba ejecutando antes. La tabla de páginas del nuevo proceso se tiene que actualizar, por lo general copiándola o mediante un apuntador a éste hacia cierto(s) registro(s) de hardware. De manera opcional, algunas o todas las páginas del proceso se pueden traer a memoria para reducir el número de fallos de página al principio (por ejemplo, es evidente que será necesaria la página a la que apunta la PC).

Cuando ocurre un fallo de página, el sistema operativo tiene que leer los registros de hardware para determinar **cuál dirección virtual produjo el fallo**. Con base en esta información debe **calcular qué página se necesita y localizarla en el disco**. Después debe buscar un marco de página disponible para colocar la nueva página, **desalojando alguna página anterior si es necesario**.

Cuando un proceso termina, el sistema operativo debe liberar su tabla de páginas, sus páginas y el espacio en disco que ocupan las páginas cuando están en disco.

#### **3.7.2. Manejos de fallos de página**

1. El hardware hace un trap al kernel, guardando el contador de programa en la pila. En la mayor parte de las máquinas, se guarda cierta información acerca del estado de la instrucción actual en registros especiales de la CPU.
2. Se inicia una rutina en código ensamblador para guardar los registros generales y demás información volátil, para evitar que el sistema operativo la destruya.
3. El sistema operativo descubre que ha ocurrido un fallo de página y trata de descubrir cuál página virtual se necesita. El sistema operativo debe obtener el contador de programa, obtener la instrucción y analizarla en software para averiguar lo que estaba haciendo cuando ocurrió el fallo.
4. Una vez que se conoce la dirección virtual que produjo el fallo, el sistema comprueba si esta dirección es válida y si la protección es consistente con el acceso. Si la dirección es válida y no ha ocurrido un fallo de página, el sistema comprueba si hay un marco de página disponible. Si no hay marcos disponibles, se ejecuta el algoritmo de reemplazo de páginas para seleccionar una víctima.

5. Si el marco de página seleccionado está sucio, la página se planifica para transferirla al disco y se realiza una conmutación de contexto, suspendiendo el proceso fallido y dejando que se ejecute otro hasta que se haya completado la transferencia al disco. En cualquier caso, el marco se marca como ocupado para evitar que se utilice para otro propósito.
6. El sistema operativo busca la dirección de disco en donde se encuentra la página necesaria, y planifica una operación de disco para llevarla a memoria. Mientras se está cargando la página, el proceso fallido sigue suspendido y se ejecuta otro proceso de usuario, si hay uno disponible.
7. Cuando la interrupción de disco indica que la página ha llegado, las tablas de páginas se actualizan para reflejar su posición y el marco se marca como en estado normal.
8. La instrucción fallida se respalda al estado en que tenía cuando empezó, y el contador de programa se restablece para apuntar a esa instrucción.
9. El proceso fallido se planifica y el sistema operativo regresa a la rutina
10. Esta rutina recarga los registros y demás información de estado, regresando al espacio de usuario para continuar la ejecución.

### **3.7.3. Respaldo de Instrucción**

Cuando un programa hace referencia a una página que no está en memoria, la instrucción que produjo el fallo se detiene parcialmente y ocurre un **trap** al sistema operativo. Una vez que el sistema operativo obtiene la página necesaria, debe reiniciar la instrucción que produjo el **trap**. Para poder reiniciar la instrucción, el sistema operativo debe determinar en dónde se encuentra el primer byte de la instrucción.

Si en una instrucción tenemos varias referencias a memoria, con frecuencia es imposible que el sistema operativo determine sin ambigüedad en dónde empezó la instrucción. Tan mal como podría estar este problema, podría ser aún peor. Algunos modos de direccionamiento utilizan el autoincremento, lo cual significa que un efecto secundario de ejecutar la instrucción es incrementar uno o más registros. Las instrucciones que utilizan el modo de autoincremento también pueden fallar.

Por fortuna, en algunas máquinas los diseñadores de la CPU proporcionan una solución, por lo general en la forma de un registro interno oculto, en el que se copia el contador de programa justo antes de ejecutar cada instrucción. Estas máquinas también pueden tener un segundo registro que indique cuáles registros se han ya autoincrementado o autodecrementado y por cuánto. Dada esta información, el sistema operativo puede deshacer sin ambigüedad todos los efectos de la instrucción fallida, de manera que se pueda reiniciar.

### **3.7.4. Bloqueo de páginas en memoria**

La memoria virtual y la E/S interactúan en formas sutiles. Considere un proceso que acaba de emitir una llamada al sistema para leer algún archivo o dispositivo y colocarlo en un búfer dentro de su espacio de direcciones. Mientras espera a que se complete la E/S, el proceso se suspende y se permite a otro proceso ejecutarse. Este otro proceso recibe un fallo de página.

Si el algoritmo de paginación es global, hay una pequeña probabilidad (distinta de cero) de que la página que contiene el búfer de E/S sea seleccionada para eliminarla de la memoria. Una solución a este problema es bloquear las páginas involucradas en operaciones de E/S en memoria, de manera que no se eliminen.

Bloquear una página se conoce como **fijada** (*pinning*) en la memoria. Otra solución es enviar todas las operaciones de E/S a búferes del kernel y después copiar los datos a las páginas de usuario

### 3.7.5. Almacén de respaldo

No hemos dicho mucho con respecto a dónde se coloca en el disco cuando se página hacia fuera de la memoria. El algoritmo más simple para asignar espacio de página en el disco es tener **una partición de intercambio especial** en el disco o aún mejor es tenerla en un disco separado del sistema operativo. Esta partición no tiene un sistema de archivos normal, lo cual elimina la sobrecarga de convertir desplazamientos en archivos a direcciones de bloque.

Cuando se inicia el sistema, esta partición de intercambio está vacía y se representa en memoria como una sola entrada que proporciona su origen y tamaño. Cuando se inicia el primer proceso, se reserva un trozo del área de la partición del tamaño del primer proceso y se reduce el área restante por esa cantidad. Con cada proceso está asociada la dirección de disco de su área de intercambio; es decir, en qué parte de la partición de intercambio se mantiene su imagen. Esta información se mantiene en la tabla de procesos. El cálculo la dirección en la que se va a escribir una página es simple: sólo se suma el desplazamiento de la página dentro del espacio de direcciones virtual al inicio del área de intercambio.

Con cada proceso está asociada la dirección de disco de su área de intercambio; es decir, en qué parte de la partición de intercambio se mantiene su imagen. Esta información se mantiene en la tabla de procesos. El cálculo la dirección en la que se va a escribir una página es simple: sólo se suma el desplazamiento de la página dentro del espacio de direcciones virtual al inicio del área de intercambio.

Sin embargo, este simple modelo tiene un problema: los procesos pueden incrementar su tamaño antes de empezar. En consecuencia, podría ser mejor reservar áreas de intercambio separadas para el texto, los datos y la pila, permitiendo que cada una de estas áreas consista en más de un trozo en el disco. El otro extremo es no asignar nada por adelantado y asignar espacio en el disco para cada página cuando ésta se intercambie hacia fuera de la memoria y desasignarlo cuando se vuelva a intercambiar hacia la memoria. De esta forma, los procesos en memoria no acaparan espacio de intercambio.

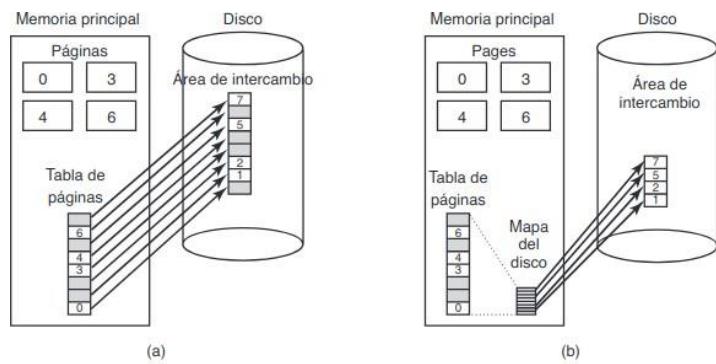


Figura 3-29. (a) Paginación a un área de intercambio estática. (b) Respaldo de páginas en forma dinámica.

En la figura 3.29(a) se ilustra una tabla de páginas con ocho páginas. Las páginas 0, 3, 4 y 6 están en memoria. Las páginas 1, 2, 5 y 7 están en disco. El área de intercambio en el disco es tan grande como el espacio de direcciones virtuales del proceso (ocho páginas). Una página que está en memoria siempre tiene una copia sombra en el disco, pero esta copia puede estar obsoleta si la página se modificó después de haberla cargado.

En la figura 3-29(b), las páginas no tienen direcciones fijas en el disco. Cuando se intercambia una página hacia fuera de la memoria, se selecciona una página vacía en el disco al momento y el mapa de disco (que tiene espacio para una dirección de disco por página virtual) se actualiza de manera acorde. Una página en memoria no tiene copia en el disco.

### 3.7.6. Separación de política y mecanismo

Una importante herramienta para administrar la complejidad de cualquier sistema es separar la política del mecanismo. Este principio se puede aplicar a la administración de la memoria, al hacer que la mayor parte del

administrador de memoria se ejecute como un proceso a nivel usuario. La política se determina en gran parte mediante el paginador externo, que se ejecuta como un proceso de usuario.

Cuando se inicia un proceso, se notifica al paginador externo para poder establecer el mapa de páginas del proceso y asignar el almacenamiento de respaldo en el disco, si es necesario. A medida que el proceso se ejecuta, puede asignar nuevos objetos en su espacio de direcciones, por lo que se notifica de nuevo al paginador externo.

Una vez que el proceso empieza a ejecutarse, puede obtener un fallo de página. El manejador de fallos averigua cuál página virtual se necesita y envía un mensaje al paginador externo, indicándole el problema. Despues el paginador externo lee la página necesaria del disco y la copia a una porción de su propio espacio de direcciones. Despues le indica al manejador de fallos en dónde está la página. Luego, el manejador de fallos desasigna la página del espacio de direcciones del paginador externo y pide al manejador de la MMU que la coloque en el espacio de direcciones del usuario, en el lugar correcto. Entonces se puede reiniciar el proceso de usuario.

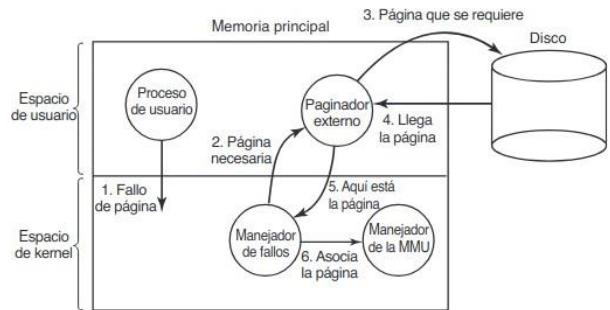
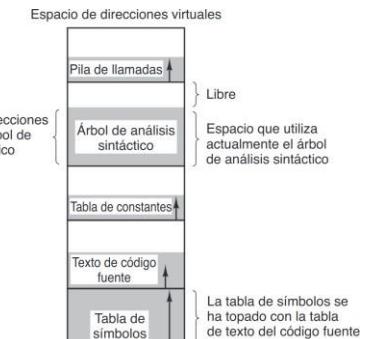


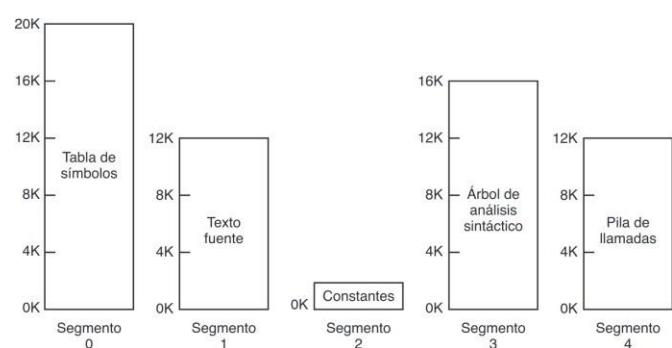
Figura 3-30. Manejo de fallos de página con un paginador externo.

## 3.8. Segmentación

Hasta ahora hemos analizado una memoria virtual **unidimensional** en la que cada proceso tiene un único espacio de direcciones. Los espacios de direcciones se dividen en varias partes cuyo tamaño cambia durante la ejecución del proceso, de manera que a una de ellas se le asigna un determinado espacio por lo que una podría llenarse e impedir la carga de más datos mientras hay espacio libre en otras. Se necesita un método que permita liberar al programador de tener que administrar la expansión de memoria.



Una solución simple es usar una memoria virtual **bidimensional** en la que cada proceso tenga **varios segmentos**. Cada **segmento** es un espacio de direcciones, es decir, una secuencia lineal de direcciones desde 0 a un máximo, completamente independiente de los demás. La longitud de cada segmento puede ser cualquier valor entre 0 y el máximo. Distintos segmentos pueden (y suelen) tener distintas longitudes. La longitud de cada segmento puede cambiar



durante la ejecución del proceso. Como cada segmento es un espacio de direcciones, puede crecer o decrecer sin afectar al resto.

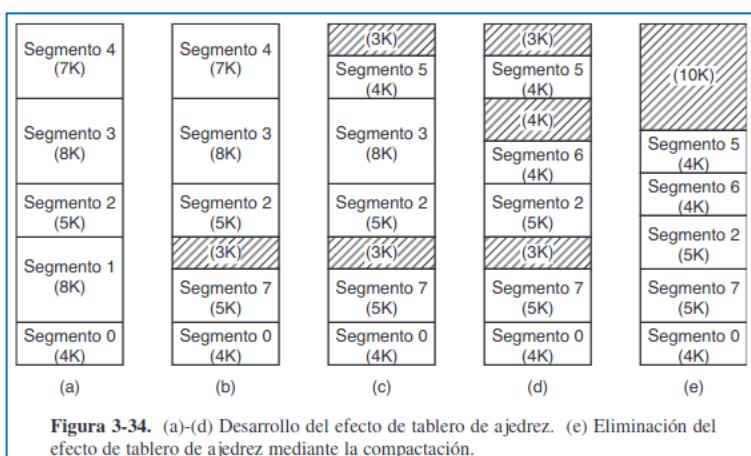
Los segmentos son **unidades lógicas** de las que el **programador es consciente**. Un segmento puede contener un procedimiento, un array, una pila, etc., pero normalmente **no contendrá una mezcla** de estas cosas. Las **direcciones** de la memoria bidimensional se dividen en dos partes, el número de segmento y la dirección dentro del segmento.

## Ventajas:

- Simplifica la administración de la expansión de la memoria
- Permite vincular procesos más eficientemente: si cada procedimiento está en su propio segmento, cuando se modifique alguno y se vuelva a compilar de manera que su longitud cambie, este no afectará a las invocaciones a otros, pues no modificará sus direcciones iniciales.
- Facilita la compartición de procedimientos, datos o bibliotecas entre procesos: se colocan en un segmento al que acceden todos los procesos que los comparten.
- Cada segmento tiene su tipo de protección: como en un segmento no se mezclarán datos con código, se asegura que los permisos sean los adecuados. En un sistema paginado no se puede garantizar que no se mezclan.

## Inconvenientes:

- Conforme se cargan y retiran segmentos en memoria, irá apareciendo huecos demasiado pequeños para que quepan la mayoría de segmentos, es decir, aparece **fragmentación externa**, se desperdicia memoria. Se soluciona mediante la **compactación** de todos los huecos pequeños en uno grande.



### 3.8.1. Comparación de la paginación y la segmentación

La principal diferencia entre la paginación y la segmentación es que **las páginas tienen un tamaño fijo y los segmentos no**.

	Paginación	Segmentación
¿Por qué se inventó esta técnica?	Para obtener un espacio de direcciones lineales grande independientemente del tamaño de la memoria física.	Para permitir que los programas y datos se dividan en espacios de direcciones lógicamente independientes, ayudando a la compartición y protección.
¿Necesita el programador ser consciente de que se está usando esta técnica?	No	Sí
¿Cuántos espacios de direcciones lineales hay?	1 por proceso	Tantos como segmentos
¿Puede el espacio de direcciones total exceder el tamaño de la memoria física?		Sí
¿Pueden los procedimientos y datos diferenciarse y protegerse por separado?	No	Sí
¿Se facilita la compartición de procedimientos entre usuarios?	No	Sí
¿Pueden las tablas de tamaño cambiante acomodarse con facilidad?	No	Sí

### 3.8.2. Segmentación con Paginación

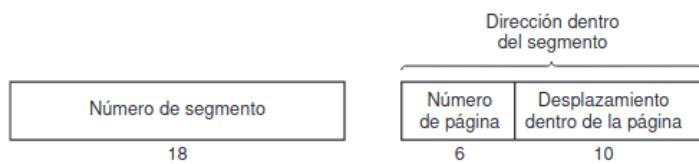
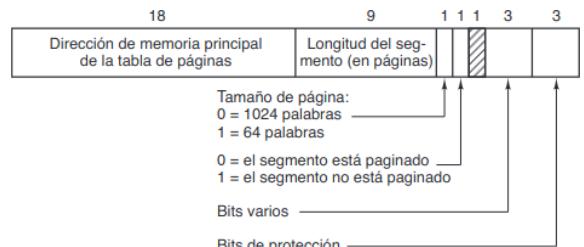
Si los **segmentos** son **extensos**, puede ser muy difícil (o imposible) mantenerlos en la memoria principal. Para solucionar esto, se usa la **segmentación con paginación**, en la que cada segmento se comportará como una memoria virtual paginada. Así, sólo las páginas que se necesiten de cada segmento estarán en memoria principal. Pretende combinar las ventajas de la segmentación explicadas antes con las de la paginación (el tamaño de página es uniforme y no hay que mantener el segmento en memoria, sólo la parte de él que se usa).

Cada proceso tendrá una **tabla de segmentos**, con un **descriptor** (entrada) por cada uno de sus segmentos. Como podría tener muchísimas entradas, la **tabla de segmentos (TS)** será también un **segmento** y como tal, estará **paginada**.

Un **descriptor** de un segmento indica si este está en memoria principal o no. Se considera que un segmento está en memoria si cualquiera de sus páginas lo está. Si un segmento está en memoria, su TP estará en memoria también. Se produce un **fallo de segmento** cuando la TP del segmento no está en memoria.

Un descriptor en una TS estará formado por:

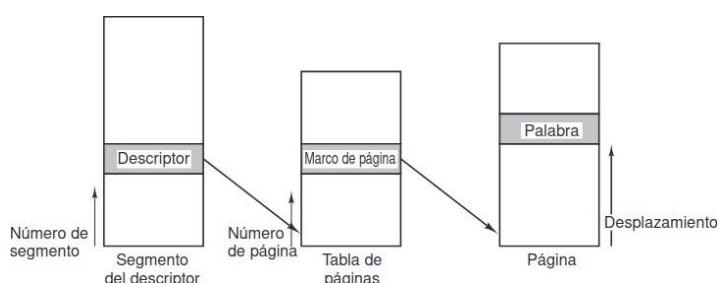
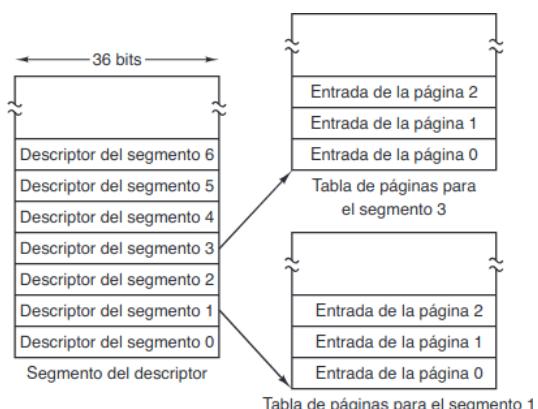
- Dirección en memoria principal de la TP del segmento.
- Longitud del segmento.
- Bits de protección del segmento
- Otros bits.



Las direcciones de memoria con segmentación paginada se dividen en número de segmento, direcciones dentro del segmento (número de página, desplazamiento en página). Cada segmento tiene su propia tabla de páginas (si es multinivel).

### TRADUCCIÓN:

- Se observa el número de segmento para encontrar su descriptor en la TS.
- Se comprueba si la TP del segmento está en memoria. Si no lo está, se produce fallo de segmento. Si lo está, se continua.
- Se observa el número de página para encontrar su entrada en la TP. Si la página no está en memoria, se produce un fallo de página. Si lo está se continua.
- Se le suma el desplazamiento al marco de página.
- Se realiza el acceso a memoria



## 3.9 Mapa de memoria de un proceso

No aparece en el tanembaum pero es importante saberlo.

Todos los procesos tienen asignado un espacio de direcciones lógicas o virtuales. El sistema operativo debe ubicar en dicho espacio las diferentes regiones en las que un proceso estructura su memoria: un área para el código, otra(s) para datos, otra para la pila, ... Esta colección de regiones, cada una con sus propiedades, define el mapa de memoria de un proceso, es decir, la estructura de su espacio de direcciones lógico.

```
64b3250ef000-64b3250f0000 r--p 00000000 103:04 12584775 /home/adriang1/Escritorio/SOI/Practica5/Ejercicio1
64b3250f0000-64b3250f1000 r--p 00001000 103:04 12584775 /home/adriang1/Escritorio/SOI/Practica5/Ejercicio1
64b3250f1000-64b3250f2000 r--p 00002000 103:04 12584775 /home/adriang1/Escritorio/SOI/Practica5/Ejercicio1
64b3250f2000-64b3250f3000 r--p 00002000 103:04 12584775 /home/adriang1/Escritorio/SOI/Practica5/Ejercicio1
64b3250f3000-64b3250f4000 rw-p 00003000 103:04 12584775 /home/adriang1/Escritorio/SOI/Practica5/Ejercicio1
64b325924000-64b325945000 rw-p 00000000 00:00 0 [heap]
7f3d81e00000-7f3d81e28000 r--p 00000000 103:04 5782528 /usr/lib/x86_64-linux-gnu/libc.so.6
7f3d81e28000-7f3d81fb0000 r--p 00028000 103:04 5782528 /usr/lib/x86_64-linux-gnu/libc.so.6
7f3d81fb0000-7f3d81ff0000 r--p 001b0000 103:04 5782528 /usr/lib/x86_64-linux-gnu/libc.so.6
7f3d81fff000-7f3d82003000 r--p 001fe000 103:04 5782528 /usr/lib/x86_64-linux-gnu/libc.so.6
7f3d82003000-7f3d82005000 r--p 00202000 103:04 5782528 /usr/lib/x86_64-linux-gnu/libc.so.6
7f3d82005000-7f3d82012000 rw-p 00000000 00:00 0
7f3d8218a000-7f3d8218d000 rw-p 00000000 00:00 0
7f3d821a0000-7f3d821a2000 rw-p 00000000 00:00 0
7f3d821a2000-7f3d821a3000 r--p 00000000 103:04 5782340 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f3d821a3000-7f3d821ce000 r--p 00001000 103:04 5782340 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f3d821ce000-7f3d821d8000 r--p 0002c000 103:04 5782340 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f3d821d8000-7f3d821da000 r--p 00036000 103:04 5782340 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f3d821da000-7f3d821dc000 rw-p 00038000 103:04 5782340 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffc74968000-7ffc74989000 rw-p 00000000 00:00 0 [stack]
7ffc749de000-7ffc749e2000 r--p 00000000 00:00 0 [vvar]
7ffc749e2000-7ffc749e4000 r--p 00000000 00:00 0 [vds]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

### 1. Primera fila:

Contiene el segmento de texto, que corresponde al código ejecutable respaldado por el archivo binario.

### 2. Segunda fila:

Representa las variables globales inicializadas.

### 3. Área azul (Heap):

Es la zona destinada a la memoria dinámica.

### 4. Archivos .so:

Son *Shared Objects*, es decir, librerías dinámicas compartidas.

### 5. Área verde (Stack):

Corresponde al espacio donde se gestiona la memoria de la pila.

### 6. VDSO:

Es la sección proporcionada por el sistema operativo, conocida como Virtual Dynamic Shared Object.

Además, si nos fijamos estos bloques de direcciones contiguas acaban en 000, eso se debe a que abarcan varias páginas y obviamente estas páginas ocupan potencia entera de 2, si se razona en papel seguro que ya lo entendéis (esto lo preguntó en un final). Recomiendo que usando un tamaño de página de 4Kb miréis en que dirección empezáis y en cual acabáis y van a acabar todas en 000.

## 3.9.1 Librerías

**Enlace estático:** El fichero ejecutable incluye todo el código que necesita la aplicación, es decir, el código propio más el de las funciones externas que necesita. El ejecutable es autocontenido.

En el caso de que se disponga de dos versiones de la misma biblioteca, estática y dinámica, hay que usar la opción **-static** del compilador. La orden de compilación tendría la forma:

gcc programa.c -static -lXYZ -o programa

donde -lXYZ indica utilizar la librería libXYZ. Por ejemplo, -lm para utilizar la librería libm (librería matemática).

**• Enlace dinámico implícito:** La carga y el montaje de la biblioteca se lleva a cabo en tiempo de ejecución del proceso, y por tanto no se incluye explícitamente en el fichero ejecutable. Es por tanto en tiempo de ejecución cuando se han de resolver las referencias del programa a constantes y funciones (símbolos) de la

biblioteca y la reubicación de regiones. Esta es la opción por defecto del compilador (si no se especifica lo contrario) ya que busca en primer lugar la versión dinámica de la biblioteca. La orden de compilación sería como la anterior, pero sin la opción -static:

```
gcc programa.c -lXYZ -o programa
```

Los mapas de memoria resultantes de un programa serán diferentes si utiliza una función que se encuentra dentro de una biblioteca, por ejemplo, la función cos(), sin() sqrt(), ... de la biblioteca matemática, al compilarlo de forma que enlace la librería de forma estática o de forma dinámica implícita. También los tamaños de los ejecutables serán distintos, mayor para la compilación Estática.

### **3.9.2 Procesos hijos**

La llamada al sistema fork() crea un nuevo proceso hijo cuyo mapa de memoria será una réplica exacta del mapa del proceso padre. Por su parte, la llamada al sistema exec() reemplaza la imagen de memoria del proceso por la del fichero ejecutable que se especifique, con lo que el mapa de memoria también Cambiará.

### **3.9.3 Proceso con archivo proyectado**

Se pueden proyectar/mapear archivos en el mapa de memoria de procesos. Para ello el sistema operativo hace corresponder una zona del mapa de memoria del proceso con el archivo, siendo una copia idéntica del contenido de los bloques de dicho archivo en disco. Una vez que el archivo está proyectado, acceder a él es más rápido que a disco, ya que no son necesarias llamadas al sistema.

## **3.10 Cosas importantes de este Tema**

En general todo es importante, de los algoritmos sobre todo el WorkingSet. De memoria virtual todo, especialmente las tablas de páginas multinivel identificar cuantos bits se necesitan para cada campo de la dirección virtual, saber cómo funciona la TLB y en general sabérselo todo bastante bien que un 50% del final lo saca de este tema.

# El Sistema de Archivos – La Calma Después de la Tormenta

Una de las principales tareas del SO es crear y administrar abstracciones del disco del sistema.

## 4.1. Consideraciones generales

Todos los sistemas necesitan almacenar y recuperar información a largo plazo de manera que se cumplan los siguientes requisitos:

- Los procesos pueden almacenar cierta cantidad de información en su espacio de direcciones, pero de manera muy limitada.
- Debe ser posible almacenar una **cantidad muy grande** de información → si usásemos el espacio de direcciones, el tamaño estaría limitado.
- La información almacenada debe **sobrevivir a la terminación del proceso** que la utilice → si usásemos el espacio de direcciones, al terminar el proceso se perdería la información.
- **Varios procesos** deben poder acceder concurrentemente a la información → si usásemos el espacio de direcciones, sólo un proceso podría acceder a la información.

Como los espacios de direcciones no son viables para almacenar información a largo plazo, se usan **dispositivos físicos** como discos magnéticos. Por ahora, entenderemos el disco como una secuencia lineal de bloques de tamaño fijo que admiten únicamente dos operaciones, leer y escribir. Esto plantea varios problemas: ¿cómo se busca información? ¿cómo se sabe qué bloques están libres?...

El ARCHIVO es una **abstracción** que realiza el SO de los **dispositivos físicos** de almacenamiento. Son **unidades lógicas de información creadas por los procesos**. Los **procesos** pueden **leer** los archivos existentes y crear otros si es necesario. La información que se almacena en los archivos debe ser **persistente**, es decir, sobrevivir a la terminación de los procesos. Un archivo debe **desaparecer** solo cuando un **proceso autorizado** (por ejemplo, su dueño) lo elimina.

Los archivos son administrados por una parte del SO denominada SISTEMA DE ARCHIVOS.

- Desde el **punto de vista del usuario**, lo más importante del sistema de archivos es su **apariencia** (cómo se estructuran los archivos, cómo se llaman, cómo se opera con ellos, etc.).
- Desde el **punto de vista del diseñador del SO**, lo más importante del sistema de archivos son **detalles sobre su implementación** (listas enlazadas, mapas de bits, etc.).

### 4.1.1. Comparación entre disco físico y disco lógico.

Un **disco físico** es un dispositivo periférico de E/S para el **almacenamiento permanente de datos**. Se trata de un dispositivo modo bloque, que **contiene un array de bloques de tamaño fijo**. Cada uno de estos bloques posee un número identificativo denominado **número de bloque físico**.

Un **disco lógico** es una **abstracción del hardware** que el sistema operativo ve como una **secuencia lineal de bloques de tamaño fijo accesibles aleatoriamente**. Cada bloque de un disco lógico tiene asignado un número identificativo denominado **número de bloque lógico**. El driver o manejador del disco entre otras tareas se encarga de traducir los números de bloques lógicos a números de bloques físicos.

Normalmente, el disco físico se divide en varias **PARTICIONES** contiguas independientes, de manera que cada una de ellas puede actuar como disco lógico. Es responsabilidad del administrador del sistema decidir qué va a contener en cada una de ellas. Permiten que convivan varios SOs en un único disco, de manera que a cada uno de ellos se le asigna una partición. La **partición activa** será aquella en la que se busca el SO en el momento del arranque de la máquina.

Cada **sistema de archivos** se encuentra **contenido por completo en un disco lógico** y un **disco lógico** puede **contener un único sistema de archivos**. Algunos discos lógicos, en vez de contener un sistema de archivos, son usados como el área de **swapping** de la memoria principal (en otros sistemas se usa una partición dedicada para esto).

En los sistemas modernos, **varios discos físicos o particiones de distintos discos pueden combinarse en un único disco lógico** para que soporte sistemas de archivos más grandes.

#### 4.1.2. Montaje de Sistemas de Archivos.

Aunque la jerarquía de archivos de UNIX parece monolítica, se puede tener varios subárboles independientes, **cada uno de los cuales puede contener un sistema de archivos completo**. Un sistema de archivos se configura para ser el sistema de archivos raíz, y para que su **directorio raíz** sea el directorio raíz del sistema. Los otros sistemas de archivos son adjuntados a la nueva estructura montando cada nuevo sistema de archivos dentro de un directorio del árbol ya existente, al que se le denominará **directorio de montaje** o punto de montaje.

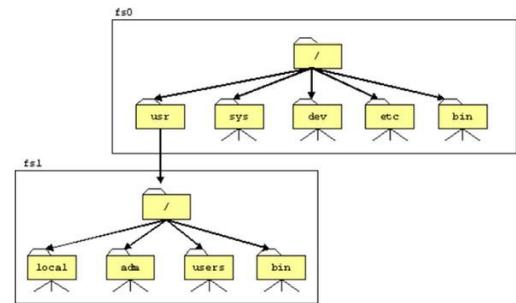


FIGURA 4.1. Montaje de un sistema de archivos en otro.

El montaje de sistemas de archivos permite **ocultar al usuario** los detalles de la organización del almacenamiento, pues el espacio de **nombres de archivos será homogéneo**, es decir, no habrá que especificar la unidad del disco como parte del nombre del archivo.

Normalmente los sistemas de archivos que usa el sistema no están cambiando frecuentemente. Entonces el núcleo usará una TABLA DE MONTAJE (normalmente en *etc/mtab*) para identificar los sistemas de ficheros que debe montar al arrancar la

#	device	directory	type	options
	/dev/hda1	/	ext2	defaults
	/dev/hda2	/usr	ext2	defaults
	/dev/hda3	none	swap	sw
	/dev/sd1	/dosc	msdos	defaults
	/proc	/proc	proc	none

FIGURA 4.2. Tabla de montaje del posible contenido del archivo.

máquina. Cada una de las líneas de esa tabla contiene la información sobre uno de los sistemas a montar: dispositivo que se monta, directorio de montaje, tipo de sistema de archivos montado y opciones de montaje.

#### 4.1.2. Llamadas al sistema y comandos asociados

La llamada al sistema `mount` permite montar un sistema de archivos desde un programa.

```
resultado = mount(dispositivo,dir,flags);
```

donde *dispositivo* es la ruta de acceso del dispositivo del disco donde se encuentra el sistema de archivos que se va a montar, *dir* es la ruta de acceso del directorio sobre el que se va a montar el sistema de archivos, y *flags* es una máscara de bits que permite especificar diferentes opciones. Si la llamada se ejecuta con éxito en resultado se almacena el valor 0. En caso contrario, se almacena el valor -1. Cuando un sistema de archivos deja de ser utilizado, puede ser desmontado.

```
resultado = umount(dispositivo);
```

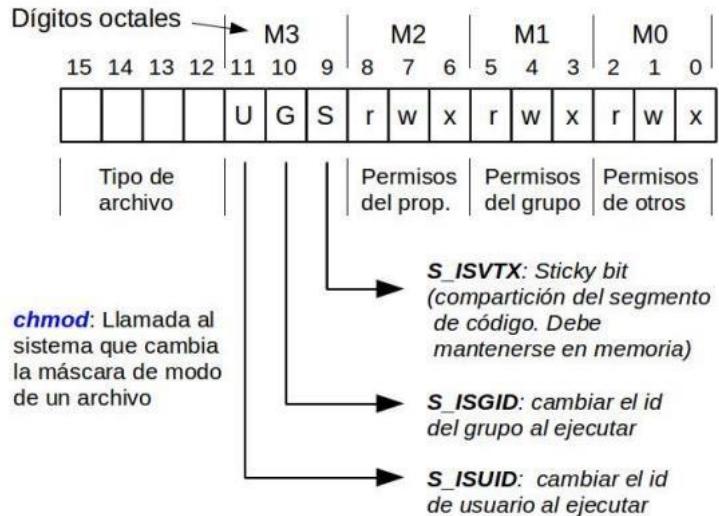
donde *dispositivo* es la ruta de acceso del archivo del dispositivo que da acceso al sistema de archivos que se desea desmontar. Las llamadas `mount` y `umount` no actualizan el archivo `/etc/mtab`, que contiene la tabla de montaje. Por lo tanto, si se decide montar un sistema de archivos desde un programa, habrá que actualizar también desde dicho programa el archivo `/etc/mtab`.

Por ejemplo, la llamada al sistema `mount("/dev/hda2","/usr",0)`; monta la partición 2 del disco duro sobre el directorio `/usr`, el sistema se monta en modo lectura/escritura. La llamada al sistema `umount("/dev/hda2")`; desmonta la partición 2 del disco duro.

#### **4.1.3. Archivos desde el punto de vista del usuario.**

#### **4.1.3.1 Máscara de Modo.**

Cada archivo en un sistema tiene asociada una máscara de 16 bits conocida como **máscara de modo**, la cual define sus permisos y tipo. Esta máscara se organiza en diferentes secciones: los bits más altos identifican el tipo de archivo, seguidos por los permisos del propietario, los permisos del grupo y, finalmente, los **permisos** para otros usuarios. Además, existen bits especiales como el S\_ISVTX (*sticky bit*), que permite compartir segmentos de código en memoria, el S\_ISGID, que cambia el ID de grupo al ejecutar, y el S\_ISUID, que cambia el ID de usuario al ejecutar. La máscara puede modificarse mediante llamadas



#### **4.1.3.1 Llamadas al sistema para operar con archivos**

`open(ruta, flags, modo)` → abre un archivo para lectura, escritura o ambos y devuelve su descriptor de archivo. Se puede hacer que, si el archivo no existe, lo cree.

*close(fd)* → cierra un archivo.

*read(fd, buffer, tamano)* → lee datos de un archivo.

`write(fd, buffer, tamano)` → escribe datos en un archivo.

*seek(d, desplazamiento, whence)* → desplaza el puntero de un archivo.

`stat(ruta, buffer)` → obtiene información sobre un archivo.

*chmod(ruta, modo)* → cambia la máscara de modo de un archivo

`rename(ruta_anterior, ruta_nueva)` → renombra un archivo

```

#include <sys/types.h>           /* incluye los archivos de encabezado necesarios */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);
/* prototipo ANSI */

#define TAM_BUFS 4096
#define MODO_SALIDA 0700
/* usa un tamaño de búfer de 4096 bytes */
/* bits de protección para el archivo de salida */

int main(int argc, char *argv[])
{
    int ent_da, sal_da, leer_cuenta, escribir_cuenta;
    char bufer[TAM_BUFS];

    if (argc != 3) exit(1);          /* error de sintaxis si argc no es 3 */

    /* Abre el archivo de entrada y crea el archivo de salida */
    ent_da = open(argv[1], O_RDONLY); /* abre el archivo fuente */
    if (ent_da < 0) exit(2);         /* si no se puede abrir, termina */
    sal_da = creat(argv[2], MODO_SALIDA); /* crea el archivo de destino */
    if (sal_da < 0) exit(3);         /* si no se puede crear, termina */

    /* Ciclo de copia */
    while (TRUE) {
        leer_cuenta = read(ent_da, bufer, TAM_BUFS); /* lee un bloque de datos */
        if (lee_cuenta <= 0) break;                  /* si llega al fin de archivo o hay un error, sale del ciclo */
        escribir_cuenta = write(sal_da, bufer, lee_cuenta); /* escribe los datos */
        if (escribir_cuenta <= 0) exit(4);            /* escribir_cuenta <= 0 es un error */
    }

    /* Cierra los archivos */
    close(ent_da);
    close(sal_da);
    if (lee_cuenta == 0)                /* no hubo error en la última lectura */
        exit(0);
    else
        exit(5);                      /* hubo error en la última lectura */
}

```

FIGURA 4.4. Un programa simple para copiar un archivo.

#### 4.1.3.1 Directorios.

Los DIRECTORIOS o carpetas son los contenedores de los archivos, que se usan para agrupar archivos relacionados de manera natural. Muchas veces, los directorios serán archivos a su vez. El sistema de archivos se organiza en una jerarquía en forma de árbol de directorios.

Cuando el sistema de archivos está organizado como un **árbol de directorios**, se necesita cierta forma de especificar los nombres de los archivos. Por lo general se utilizan dos métodos distintos. En el primer método, cada archivo recibe un **nombre de ruta absoluta** que consiste en la ruta desde el directorio raíz al archivo. Sin importar cuál carácter se utilice, si el primer carácter del nombre de la ruta es el separador, entonces la ruta es absoluta. El otro tipo de nombre es el **nombre de ruta relativa**. Éste se utiliza en conjunto con el concepto del **directorio de trabajo** (también llamado directorio actual).

La mayoría de los sistemas operativos que proporcionan un sistema de directorios jerárquico tienen dos entradas especiales en cada directorio: “.” y “..”, que por lo general se pronuncian “punto” y “puntopunto”. Punto se refiere al directorio actual; puntopunto se refiere a su padre (excepto en el directorio raíz, donde se refiere a sí mismo).

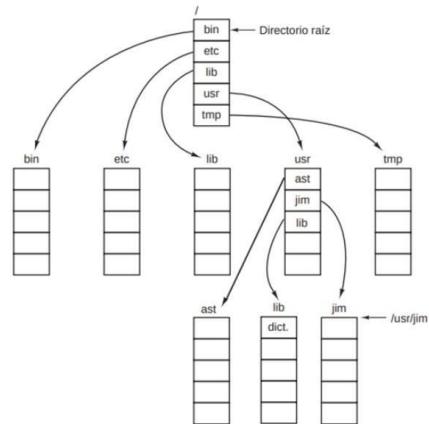


FIGURA 4.5. Un árbol de directorios de UNIX.

## 4.2. Identificadores de usuario y grupo

Puesto que UNIX es un sistema multiusuario, el sistema operativo asocia a cada proceso dos identificadores enteros positivos de usuario y dos identificadores enteros positivos de grupo. Los identificadores de usuario son el identificador de usuario real, *uid*, y el identificador de usuario efectivo, *euid*. Mientras que, para el grupo se tiene el identificador del grupo real, *gid*, y el identificador de grupo efectivo, *egid*.

El **uid** identifica al usuario que es responsable de la ejecución del proceso y el **gid** identifica al grupo al cual pertenece dicho usuario. El **euid** se utiliza, principalmente, para determinar el propietario de los ficheros recién creados, para permitir el acceso a los ficheros de otros usuarios y para comprobar los permisos para enviar señales a otros procesos. El uso del **egid** es similar al del **euid** pero desde el punto de vista del grupo.

Usualmente, el **uid** y el **euid** van a coincidir, pero si un usuario U1 ejecuta un programa P que pertenece a otro usuario U2 y que tiene activo el bit **S\_ISUID** entonces el proceso asociado a la ejecución de P por parte de U1 va a cambiar su **euid** y va a tomar el valor del **uid** del usuario U2. Es decir, a efectos de comprobación de permisos sobre P, U1 va a tener los mismos permisos que tiene el usuario U2. Para el identificador de grupo efectivo **egid** se aplica la misma norma.

Las llamadas al sistema **getuid**, **geteuid**, **getgid** y **getegid** permiten determinar qué valores toman los identificadores **uid**, **euid**, **gid** y **egid**, respectivamente. Su sintaxis es similar a la de la llamada al sistema **getpid**. Para cambiar los valores que toman estos identificadores, es posible utilizar las llamadas al sistema **setuid** y **setgid**.

```
salida = setuid(param);
```

Cuando un archivo ejecutable tiene el bit **S\_ISUID** activado, cualquier usuario que lo ejecute obtiene temporalmente los privilegios del propietario del archivo durante la ejecución del programa. Esto es útil para permitir que un programa realice tareas que requieren privilegios específicos, incluso si el usuario que lo ejecuta no tiene esos privilegios normalmente. Lo mismo pasa con **S\_ISGID**.

La llamada al sistema **setuid** permite asignar el valor par al **euid** y al **uid** del proceso que invoca a la llamada. Se distinguen dos casos. En el primero, el identificador de usuario efectivo del proceso que efectúa la llamada es el del superusuario. En este caso **uid = parametro** y **euid = parametro**. En el segundo caso, el identificador del usuario efectivo del proceso que efectúa la llamada no es el del superusuario. En este caso **euid = parametro** si se cumple que el valor del parámetro **parametro** coincide con el valor del **uid** del proceso, o, esta llamada se está invocando dentro de la ejecución de un programa que tiene su bit **S\_ISUID** activado y el valor del parámetro **par** coincide con el valor del **uid** del propietario del programa. Si la llamada se ejecuta con éxito entonces salida vale 0. Si se produce un error salida vale -1.

Supóngase que, en un cierto directorio, se tienen tres archivos. Primero, el programa ejecutable *ejemident*, cuyo código es el indicado en el programa de la figura 4.6., que pertenece al usuario USUARIO1, este fichero tiene la siguiente máscara simbólica de permisos – *rws rwx rwx*, es decir, todos los usuarios pueden leer, escribir y ejecutar este archivo, además su bit **S\_ISUID** se encuentra activado. Segundo, el fichero de texto *fichero1.txt* que pertenece al usuario USUARIO1, este fichero tiene los siguientes permisos – *rw- --- ---*, es decir, únicamente el propietario del fichero puede leer y escribir en dicho fichero. Por último, el fichero de texto *fichero2.txt* que pertenece al usuario USUARIO2, este fichero tiene los siguientes permisos – *rw- --- ---*, es decir, únicamente el propietario del fichero puede leer y escribir en dicho fichero.

Supónganse además que USUARIO1 tiene **uid = 501** y que USUARIO2 tiene **uid = 503**. Se van a considerar tres casos. En el primer caso, el USUARIO1 ejecuta *ejemident* y se obtiene la traza de la figura 4.7. en pantalla. En el segundo caso, el USUARIO2 ejecuta el archivo *ejemident* y se obtiene la traza de la figura 4.8. en pantalla. En el tercer y último caso, el USUARIO2 ejecuta el archivo *ejemident*, se supone que ahora que su bit **S\_ISUID** no está activado, es decir, su máscara simbólica es – *rwx rwx rwx*, y se obtiene la traza de la figura 4.9. en pantalla.

En el caso 2 como **IS\_UID** está activado, si el USUARIO2 ejecuta ese archivo adquiere temporalmente los permisos del propietario del archivo (USUARIO1).

```

#include <fcntl.h>

main(){
    int x, y, fd1, fd2;
    x=getuid(); y=/geteuid();
    printf("\nUID=%d, EUID=%d\n", x, y);
    fd1=open("ficherol.txt", O_RDONLY);
    fd2=open("ficher02.txt", O_RDONLY);
    printf("fd1=%d, fd2=%d\n", fd1, fd2);

    setuid(x);
    printf("UID=%d, EUID=%d\n", getuid(),geteuid());

    fd1=open("ficherol.txt", O_RDONLY);
    fd2=open("ficher02.txt", O_RDONLY);
    printf("fd1=%d, fd2=%d\n", fd1, fd2);

    setuid(y);
    printf("UID=%d, EUID=%d\n", getuid(),geteuid());
}

```

- El programa pertenece a USUARIO1
  - Máscara **rwx rwx rwx**
  - S\_ISUID=1
- fichero1.txt pertenece a USUARIO1:
  - Máscara **rw - - - - -**
- fichero2.txt pertenece a USUARIO2:
  - Máscara **rwx - - - - -**
- uid de USUARIO1 = 501  
uid de USUARIO2 = 503

**1** USUARIO 1 ejecuta el programa

[1] UID=501, EUID=501	[2] fd1=3, fd2=-1
[3] UID=501, EUID=501	[4] fd1=4, fd2=-1
[5] UID=501, EUID=501	

**2** USUARIO 2 ejecuta el programa

[1] UID=503, EUID=501	[2] fd1=3, fd2=-1
[3] UID=503, EUID=503	[4] fd1=-1, fd2=4
[5] UID=503, EUID=501	

**3** USUARIO 2 ejecuta el programa y su bit S\_ISUID=0

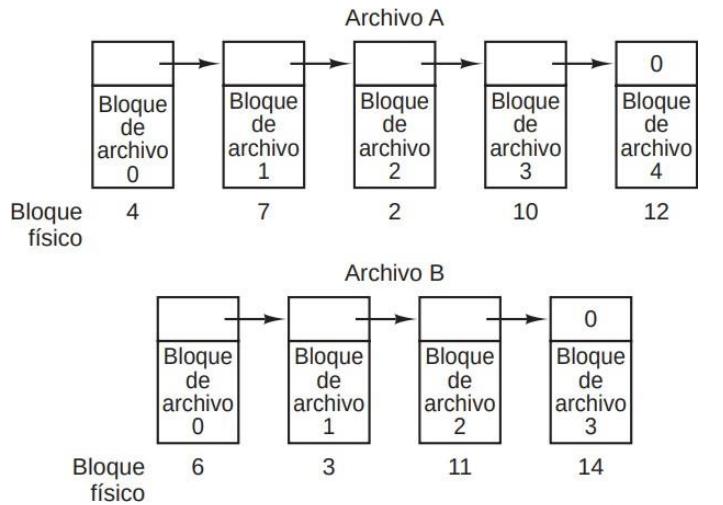
[1] UID=503, EUID=503	[2] fd1=-1, fd2=2
[3] UID=503, EUID=503	[4] fd1=-1, fd2=4
[5] UID=503, EUID=503	

## 4.3. Implementación del sistema de archivos

A continuación, se presentan distintas opciones de distribución de los bloques de un archivo para poder mantener un registro acerca de qué bloques del disco pertenecen a qué archivo.

### 4.3.1. Asignación de Lista Enlazada

Se mantiene cada archivo como **una lista enlazada de bloques del disco**, de manera que la primera palabra de cada bloque se usa como apuntar al siguiente. El último bloque del archivo tiene el apuntado a un valor no válido como 0 para indicar el fin de la cadena. A diferencia de lo que sucede en la asignación contigua, permite usar todos los bloques del disco, por lo que **no** se pierde espacio debido a la **fragmentación externa**. Para localizar un archivo sólo hace falta almacenar la **dirección de disco de su primer bloque**, el resto se pueden encontrar a partir de ella. Sin embargo, el acceso aleatorio es muy lento, pues para llegar al bloque  $k$  de un archivo, se tendrá que comenzar desde el primero y leer los  $k-1$  siguientes desde el disco. En muchos sistemas, el tamaño de página es múltiplo del tamaño del bloque, por lo que para leer una sola página del disco habría que acceder a 2 bloques.



### 4.3.2. Usando una Tabla FAT.

Versión de lista enlazada que soluciona sus problemas. Se mantiene en memoria en todo momento **una tabla FAT** (tabla de asignación de archivos) con **tantas entradas como bloques tenga el disco** de manera que en

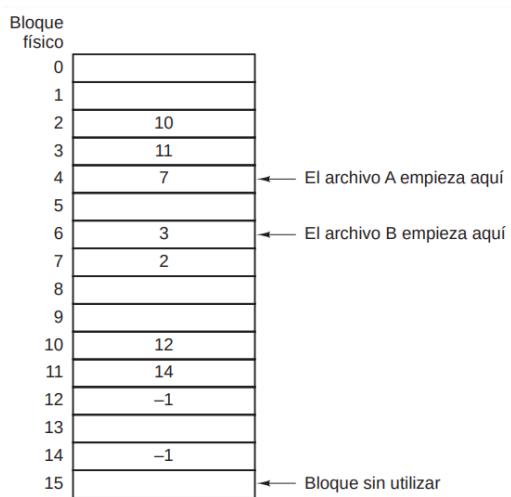
cada una de ellas se almacena el apuntador al siguiente bloque del archivo. La última entrada del archivo tiene en el apuntador un valor no válido como -1 para indicar el fin de la cadena.

El **acceso aleatorio** es más rápido porque, aunque haya que seguir la cadena hasta llegar al bloque deseado, ahora los punteros están en memoria principal, que es más rápida. La cantidad de datos que se pueden almacenar en un bloque es potencia de **2**.

Toda la tabla tiene que estar en memoria todo el tiempo para que funcione, y su tamaño es proporcional al del disco, por lo que la idea escala muy mal con el tamaño del disco. Se usa en Windows.

Un disco de  $n$  bloques implica  $n$  entradas.

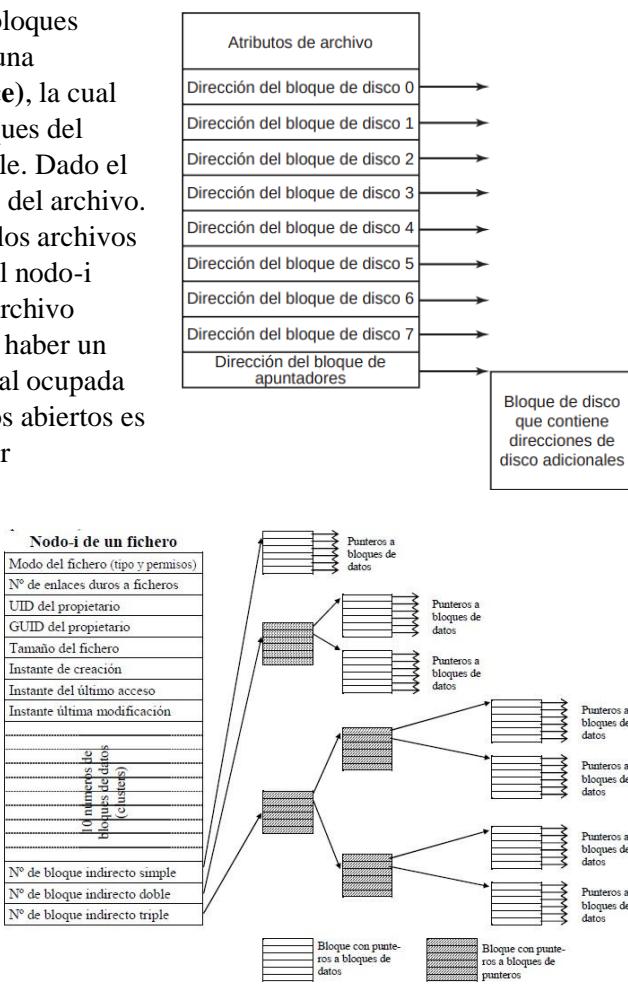
$$TamTabFAT = \frac{TamDisc}{TamBloque}$$



#### **4.3.3. Asignación con Nodos-i.**

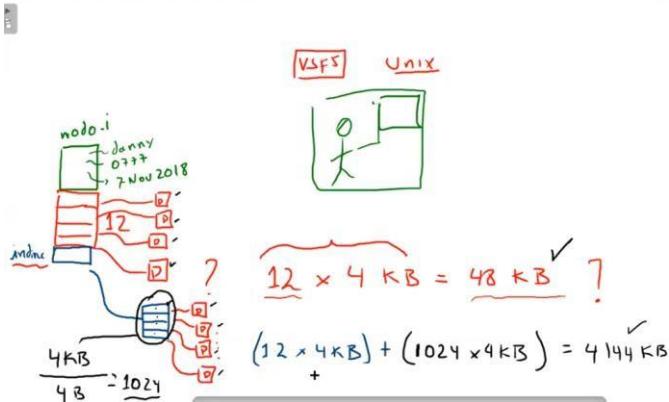
Nuestro último método para llevar un registro de qué bloques pertenecen a cuál archivo es asociar con cada archivo una estructura de datos conocida como **nodo-i** (**nodo-índice**), la cual lista los atributos y las direcciones de disco de los bloques del archivo. En la figura 4.12. se muestra un ejemplo simple. Dado el nodo-i, entonces es posible encontrar todos los bloques del archivo. La gran ventaja de este esquema, en comparación con los archivos vinculados que utilizan una tabla en memoria, es que el nodo-i necesita estar en memoria sólo cuando está abierto el archivo correspondiente. Si cada nodo-i ocupa  $n$  bytes y puede haber un máximo de  $k$  archivos abiertos a la vez, la memoria total ocupada por el arreglo que contiene los nodos-i para los archivos abiertos es de sólo  $kn$  bytes. Sólo hay que reservar este espacio por adelantado.

Por lo general, este arreglo es mucho más pequeño que el espacio ocupado por la tabla de archivos descrita en la sección anterior. La razón es simple: la tabla para contener la lista enlazada de todos los bloques de disco es proporcional en tamaño al disco en sí. Si el disco tiene  $n$  bloques, la tabla necesita  $n$  entradas. A medida que aumenta el tamaño de los discos, esta tabla aumenta linealmente con ellos. En contraste, el esquema del nodo- $i$  requiere un arreglo en memoria cuyo tamaño sea proporcional al número máximo de archivos que pueden estar abiertos a la vez.



Un problema con los nodos-i es que, si cada uno tiene espacio para un número fijo de direcciones de disco, ¿qué ocurre cuando un archivo crece más allá de este límite? Una solución es reservar la última dirección de disco no para un bloque de datos, sino para la dirección de un bloque que contenga más direcciones de bloques de disco, como se muestra en la figura 4-13. Algo aún más avanzado sería que dos o más de esos

bloques contuvieran direcciones de disco o incluso bloques de disco apuntando a otros bloques de disco llenos de direcciones.



## 4.4 Administración y optimización de sistemas de archivos

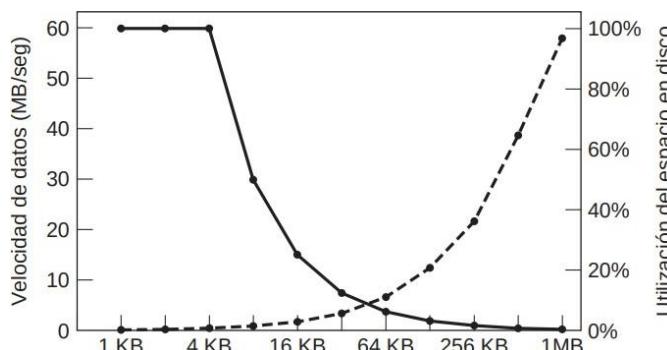
### 4.4.1 Tamaño de bloque

Existen dos maneras de almacenar un archivo de  $n$  bytes en el disco:

- Asignarle  $n$  bytes consecutivos → provoca fragmentación externa y, si crece, habría que moverlo a otro lugar del disco, lo cual es muy costoso.
- Dividir el archivo en varios bloques de un determinado tamaño no necesariamente consecutivos.

Por tanto, casi todos los sistemas escogen la segunda alternativa. Para poder implementarla, hay que decidir cuál será el tamaño de estos bloques. Tendría sentido escoger un tamaño múltiplo del de sector, pista o cilindro del disco, pero estos valores dependen del dispositivo, lo cual es inconveniente.

- Si el tamaño de bloque es demasiado grande → el último bloque de cada archivo quedará mayormente vacío, aparece fragmentación interna.
- Si el tamaño de bloque es demasiado pequeño → la mayoría de archivos ocuparán muchos bloques, así que para leerlos se necesitarán muchas búsquedas y retrasos rotacionales.



**FIGURA 4.13.** La curva punteada (escala del lado izquierdo) da la velocidad de datos del disco. La curva sólida (escala del lado derecho) da la eficiencia del espacio de disco.  
Todos los archivos son en promedio de 4 KB.

#### Observaciones de la imagen:

##### -Velocidad de datos del disco (curva punteada)

La velocidad de datos aumenta casi de forma lineal con el tamaño del bloque (hasta que las transferencias tardan tanto que el tiempo de transferencia empieza a ser importante)

##### -Eficiencia del espacio en disco (curva continua)

Cuanto más se aumenta el tamaño del bloque más disminuye la utilización del espacio. En realidad, pocos archivos son un múltiplo exacto del tamaño de bloque del disco, por lo que siempre se desperdicia espacio en el último bloque de un archivo.

Las curvas se cruzan en los 64 KB, pero aun así ninguno de los dos valores es bueno. Históricamente se

hacían bloques de 1KB a 4KB, pero con discos de 1 TB donde no importa tanto el espacio desperdiciado se podría utilizar un tamaño de bloque de 64 KB

### Resumiendo:

Un tamaño de bloque grande --> **Desperdiciar espacio**

- Los archivos pequeños desperdician una gran cantidad de espacio en disco.
- El desperdicio de espacio al final de cada pequeño archivo no es muy importante, debido a que el disco se llena por una cantidad de archivos grandes y la cantidad de espacio ocupado por los pequeños archivos es insignificante.

Un tamaño de bloque pequeño --> **Desperdiciar tiempo**

- La mayoría de archivos abarcarán varios bloques y por ende, necesitan varias búsquedas y retrasos rotacionales para leerlos, lo cual reduce rendimiento
- La acción de leer un archivo que consistía en de muchos bloques pequeños será lenta.

**Ejemplo:** Disco con 1 MB por pista, un tiempo de rotación de 8.33 mseg y un tiempo de búsqueda promedio de 5 ms. El tiempo en milisegundos para leer un bloque de k bytes es entonces la suma de los tiempos de búsqueda, el retraso rotacional y de transferencia.

$$5 + 4.165 + (k/1000000) \times 8.33$$

### 4.4.2 Registro de Bloques Libres

Existen dos métodos utilizados ampliamente para llevar registro de los bloques libres del disco:

**Lista enlazada** de bloques libres del disco, en la que cada entrada es el identificador de un bloque de disco que no está ocupado.

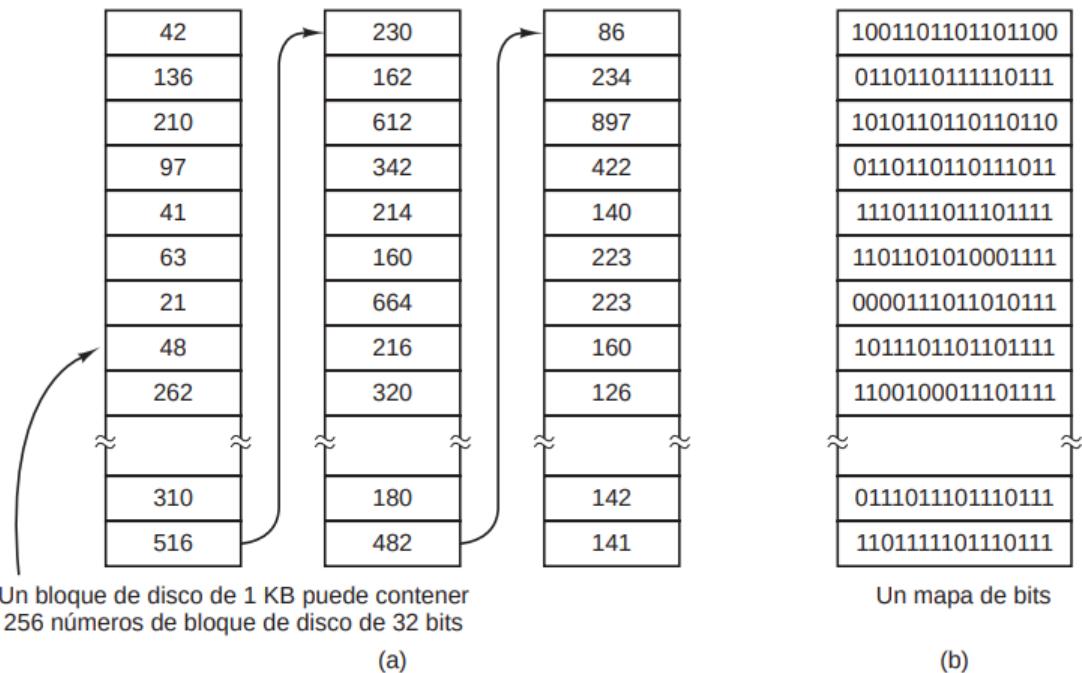
- Las entradas **no** están **ordenadas**.
- Se divide en **secciones** del **tamaño de un bloque**, de manera que la última entrada de cada una de ellas es un apuntador a la siguiente. Sólo la primera sección de la lista estará en memoria principal, para más rápido acceso. El resto se almacenan en el disco. Al crear un archivo, se tomarán los bloques libres necesarios de la sección que está en memoria. Si no hay suficientes, se lee una nueva desde el disco. Al borrar un archivo, se agregarán los bloques que quedaron libres a la sección que está en memoria. Si no tiene espacio suficiente, se escribirán en una del disco.
- Su **tamaño cambia** con el tiempo en función de la cantidad de bloques libres que haya en el disco en un momento dado.

**Mapa de bits**, vector de tantos bits como bloques hay en el disco de manera que en cada bit se almacena un 0 o un 1 en función de si el bloque correspondiente está ocupado o no.

- Se almacena en su totalidad en memoria principal.
- Su tamaño es constante, ocupa tantos bits como bloques tenga el disco.

Por lo general, el **mapa de bits ocupa menos espacio que la lista**. Sólo si el disco está casi lleno (es decir, hay pocos bloques libres) la lista ocupará menos espacio. Para que la lista sea más corta, las entradas pueden representar **series de bloques libres** consecutivos en lugar de bloques individuales. En cada entrada se almacenaría el identificador del primer bloque de la serie y el número de bloques libres consecutivos que la forman. Si el disco está muy fragmentado, esta alternativa sería menos eficiente en términos de espacio, pues las entradas serían más anchas y no se reduciría mucho su cantidad.

Bloques de disco libres: 16, 17, 18



**FIGURA 4.13.** (a) Almacenamiento de la lista de bloques libres en una lista enlazada.

(b) Un mapa de bits.

## 4.5 Cosas importantes de este Tema

Saber bien lo del IS\_UID y seteuid. Saber de qué depende el tamaño de las tablas FAT y los inodos y como calcularlos.

# 5

## Entrada/Salida - Métodos de Tortura Medieval PT2

Una de las principales tareas del SO es **controlar todos los dispositivos de E/S de la computadora**. Para poder hacer esto, el SO debe de ser capaz de:

- Emitir **comandos** para los dispositivos.
- Captar las **interrupciones que** produzcan los dispositivos.
- Manejar los **errores** que puedan provocar los dispositivos.
- Proporcionar una abstracción al usuario en forma de **interfaz simple entre los dispositivos y el resto del sistema**. Hasta donde sea posible, esta interfaz debe ser igual para todos los dispositivos.

El **software** de E/S forma una parte significante del código del SO. Está estructurado en **niveles**, cada uno de los cuales tienen una **tarea bien definida**. Sobre el **hardware** de E/S nos centraremos únicamente en su **interfaz** con el software.

### 5.1. Principios del Hardware de E/S

#### 5.1.1. Dispositivos de E/S

Los dispositivos de E/S se pueden dividir básicamente en dos categorías: **dispositivos de bloque** y **dispositivos de carácter**. Un dispositivo de bloque almacena información en bloques de tamaño fijo, **cada uno con su propia dirección**. Los tamaños de bloque comunes varían desde 512 bytes hasta 32,768 bytes. Todas las **lecturas y escrituras** se realizan en unidades de **uno o más bloques enteros consecutivos**. La propiedad esencial de un dispositivo de bloque es que es posible leer o escribir cada bloque de manera **independiente** de los demás. Los discos duros, CDs y memorias USBs son dispositivos de bloque comunes.

El otro tipo de dispositivo de E/S es el **dispositivo de carácter**.

Un dispositivo de carácter envía o acepta un flujo de caracteres, sin importar la estructura del bloque. Los caracteres **no son direccionables** y **no tienen ninguna operación de búsqueda**.

Las impresoras, las interfaces de red, los ratones y la mayoría de los demás dispositivos que no son parecidos al disco se pueden considerar como dispositivos de carácter.

**Este esquema de clasificación no es perfecto.** Algunos dispositivos simplemente no se adaptan. Por ejemplo, los **relojes** no son direccionables por bloques. Tampoco generan ni aceptan flujos de caracteres. Todo lo que hacen es producir interrupciones a intervalos bien definidos. Aún así, el modelo de dispositivos de bloque y de carácter es lo bastante general como para poder utilizarlo como base para hacer que parte del sistema operativo que lida con los dispositivos de E/S sea independiente.

Los dispositivos de E/S cubren un amplio rango de velocidades. La figura 5.1. muestra las velocidades de transferencia de datos de algunos dispositivos comunes. La mayoría de estos dispositivos tienden a hacerse más rápidos a medida que pasa el tiempo.

Dispositivo	Velocidad de transferencia de datos
Teclado	10 bytes/seg
Ratón	100 bytes/seg
Módem de 56K	7 KB/seg
Escáner	400 KB/seg
Cámara de video digital	3.5 MB/seg
802.11g inalámbrico	6.75 MB/seg
CD-ROM de 52X	7.8 MB/seg
Fast Ethernet	12.5 MB/seg
Tarjeta Compact Flash	40 MB/seg
FireWire (IEEE 1394)	50 MB/seg
USB 2.0	60 MB/seg
Red SONET OC-12	78 MB/seg
Disco SCSI Ultra 2	80 MB/seg
Gigabit Ethernet	125 MB/seg
Unidad de disco SATA	300 MB/seg
Cinta de Ultrium	320 MB/seg
Bus PCI	528 MB/seg

FIGURA 5.1. Velocidad de transferencia de datos comunes de algunos dispositivos, redes y buses.

## 5.1.2. Controladores de Dispositivos

Por lo general, las unidades de E/S consisten en un componente mecánico (dispositivo) y un componente electrónico (dispositivo controlador). A menudo es posible separar las dos porciones para proveer un diseño más modular y general. El componente electrónico se llama **controlador de dispositivo** o adaptador. En las computadoras personales, comúnmente tiene la forma de un **chip** en la **tarjeta principal** o una **tarjeta de circuito integrado** que se puede insertar en una ranura de expansión (PCI). El componente mecánico es el dispositivo en sí.

El controlador tendrá un conector para un cable que lleva al dispositivo en sí. Muchas veces, a una sola tarjeta controladora se le pueden conectar **varios dispositivos**. Si la **interfaz** entre la controladore y el dispositivo es **estándar**, las empresas fabricarán controladores y dispositivos que se adapten a ella.

## 5.1.3. Comunicación Controladora – CPU

Cada **controlador** tiene unos cuantos **registros** que se utilizan para **comunicarse con la CPU**. Al escribir en los **registros**, el sistema operativo puede hacer que el dispositivo envíe o acepte datos, se encienda o se apague, o realice cualquier otra acción. Al leer de estos **registros**, el sistema operativo puede **conocer el estado del dispositivo**, si está **preparado o no para aceptar un nuevo comando**, y sigue procediendo de esa manera. Además de los registros de control, muchos dispositivos tienen un **búfer de datos** que el sistema operativo puede leer y escribir.

De todo esto surge la cuestión acerca de cómo se comunica **la CPU** con los registros de control y los búferes de datos de los dispositivos. Existen dos alternativas. Mediante un **puerto de E/S** o una **dirección de memoria**.

### 5.1.3.1 Puertos de E/S

Consiste en agujarle a cada registro de control un **número de puertos de E/S** (entero de 8 o 16 bits). El conjunto de todos los puertos de E/S forma el **espacio de puertos de E/S**. **Sólo el SO** puede acceder a él. Es **independiente del espacio de direcciones de la memoria**: una dirección x en el espacio de direcciones de memoria es diferente a una dirección x en el de puertos. Las operaciones de lectura y escritura en puertos no se pueden realizar con las instrucciones para memoria **lw** y **sw**, se usan **instrucciones especiales IN** y **OUT**.

IN REG,PUERTO, o OUT PUERTO, REG

En este esquema, los espacios de direcciones para la memoria y la E/S son distintos, como se muestra en la figura 5.2.(a). Las instrucciones **IN R0,4** y **MOV R0,4** son completamente distintas en este diseño. La primera lee el contenido del puerto 4 de E/S y lo coloca en R0, mientras que la segunda lee el contenido de la palabra de memoria 4 y lo coloca en R0. Los 4s en estos ejemplos se refieren a espacios de direcciones distintos y que no están relacionados.

#### Importante

Con el método de E/S por **asignación de memoria**, no usa las instrucciones **IN** y **OUT**, ya que se puede acceder a todos los registros de control mediante **lw** y **sw**, puesto que se encuentran en el espacio de memoria.

E **IN** y **OUT** son necesarias cuando queremos escribir en los **puertos** teniendo espacios de direcciones separados, esto se debe a que nos permiten **diferenciar** a qué espacio se está accediendo y la naturaleza de la operación.

### 5.1.3.2 Asignación de Memoria.

Consiste en asignar los registros de control al **espacio de direcciones de memoria**. A cada registro de control se le asigna una dirección de memoria **única** para la que **no hay memoria asignada**.

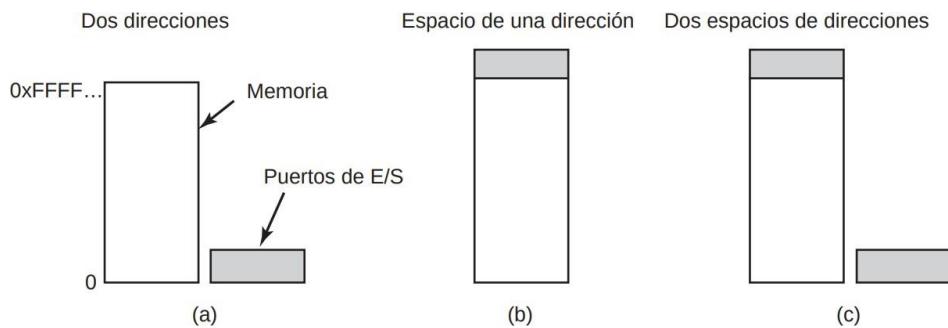
### 5.1.3.3 Esquema Híbrido

Consiste en asignar los **búferes de datos al espacio de direcciones de memoria** y los **registros de control en un espacio de puertos de E/S** separados.

#### 5.1.3.4 Funcionamiento de estos Esquemas.

Cuando la CPU desea una palabra (ya sea de memoria o de un puerto de E/S):

- La CPU coloca la dirección que necesita (ya sea de memoria o de un puerto de E/S) en el bus de direcciones.
- La CPU activa la señal *read* en el bus de control.
- En función del esquema usado:
  - o Usando **puertos de E/S**:
    - Se añade un bit en el bus de control para indicar si se necesita una palabra de E/S o de memoria.
    - Responde la memoria o el dispositivo de E/S.
  - o Usando **asignación de memoria**:
    - Todos los módulos de memoria y los dispositivos de E/S comprueban si la dirección está en su rango.
    - Si la dirección está en su rango, responde a la petición. Como ninguna dirección se asigna tanto a la memoria como a un dispositivo de E/S, **no hay ambigüedad ni conflicto**.



**FIGURA 5.2.** (a) Espacio separado de E/S y memoria. (b) E/S por asignación de memoria. (c) Híbrido.

En la imagen *a* se encuentran en el dispositivo controlador. En la imagen *b* se encuentran en la RAM en una región protegida en el kernel.

#### 5.1.3.5 Ventajas de la E/S por asignación de Memoria.

Los dos esquemas para direccionar los controladores tienen distintos puntos fuertes y débiles. Vamos a empezar con las ventajas de la E/S por **asignación de memoria**. En primer lugar, si usamos el **sistema de puertos de E/S** si se necesitan instrucciones de E/S especiales para leer y escribir en los registros de control, para acceder a ellos se requiere el uso de código **ensamblador**, ya que no hay forma de ejecutar una instrucción IN o OUT en C o C++. En contraste, con la E/S por **asignación de memoria** los registros de control de dispositivos son sólo variables en memoria y se pueden direccionar en C de la misma forma que cualquier otra variable. Así, con la E/S por asignación de memoria, un controlador de dispositivo de E/S puede escribirse completamente en C.

En segundo lugar, con la E/S por **asignación de memoria** no se requiere un mecanismo de **protección** especial para evitar que los procesos realicen operaciones de E/S. Todo lo que el sistema operativo tiene que hacer es abstenerse de colocar esa porción del espacio de direcciones que contiene los registros de control en el espacio de direcciones virtuales de cualquier usuario. Mejor aún, si cada dispositivo tiene sus registros de control en una página distinta del espacio de direcciones, el sistema operativo puede proporcionar a un usuario el control sobre dispositivos específicos, pero no el de los demás, con sólo incluir las páginas deseadas en su tabla de páginas.

#### 5.1.3.6 Desventajas de la E/S por asignación de Memoria.

La E/S por asignación de memoria también tiene sus **desventajas**. En primer lugar, la mayoría de las computadoras actuales tienen alguna forma de colocar en caché las palabras de memoria. **Sería desastroso**

**colocar en caché un registro de control de dispositivos.** Considere el ciclo de código en lenguaje ensamblador mostrado a continuación en presencia de caché. La

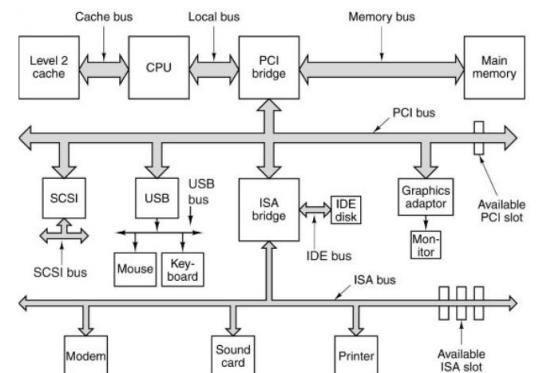
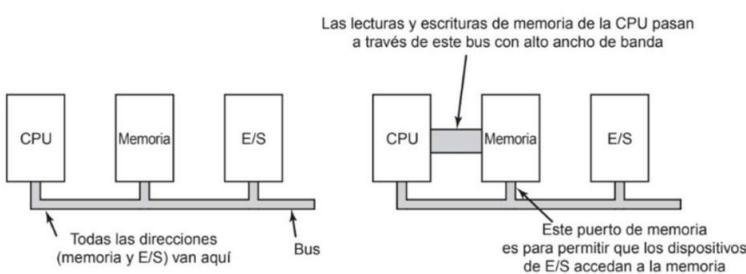
```
CICLO: TEST PUERTO_4 // comprueba si el puerto 4 es 0
       BEQ LISTO // si es 0, ir a lista
       BRANCH CICLO // en caso contrario continúa la prueba
LISTO:
```

primera referencia a PUERTO\_4 haría que se colocara en la caché. Las referencias subsiguientes sólo tomarían el valor de la caché sin siquiera preguntar al dispositivo. Después, cuando el dispositivo por fin estuviera listo, el software no tendría manera de averiguarlo. En vez de ello, el ciclo continuaría para siempre.

Para evitar esto, las páginas del espacio de direcciones asignadas a los registros de control deben marcarse como **no cacheables**. **Coherencia caché:** si guardamos el registro de control en caché, los datos que la CPU lee o escribe pueden estar desfasados. La CPU puede leer de la cache en vez del dispositivo.

**Todos los módulos de memoria y los dispositivos de E/S deben examinar todas las referencias a memoria** para saber a cuáles tienen que responder. Esto es sencillo si el ordenador tiene un solo bus, pero normalmente existirá un **bus separado** de alta velocidad dedicado únicamente a la **memoria**. Como consecuencia, **los dispositivos de E/S no tienen manera de ver las direcciones de memoria que van por el bus de memoria**, así que no las pueden responder. Hay dos posibles soluciones a esto:

- Se envían todas las referencias a la memoria principal, y si esta no puede responder, la CPU prueba con otros buses. Requiere hardware adicional.
- Se filtran las direcciones que salen de la CPU con un **ponte PCI** que contiene registros de rango que se cargan durante el arranque del sistema, de manera que las direcciones ubicadas dentro de un rango que no pertenece a la memoria se envían al bus PCI en lugar de al bus de memoria.



#### 5.1.4. Acceso directo a memoria (DMA)

El **acceso directo a memoria** (DMA) permite liberar a la CPU de labores poco sofisticadas relacionadas con la transferencia de datos de operaciones de E/S. Para usar este esquema, el hardware debe tener alguna **controladora de DMA**, que accede al bus de forma independiente de la CPU. Normalmente hay **una única controladora DMA**, que accede al bus de forma independiente de la CPU. Contiene **varios registros** en los que escribir y leer:

- Registro de **dirección de memoria**.
- Registro de **contador de bytes**.
- Registros de **control**, que especifican:
  - El **puerto E/S** a utilizar
  - El **tipo** de transferencia (R/W)
  - **Unidad** de transferencia (byte o palabra)
  - Número de bytes a transferir por **ráfaga**.

Puede gestionar **varias transferencias** a la vez, de manera que cada una de ellas usar una **controladora de E/S distinta** y tiene un **canal con sus propios registros**. Usa **direcciones físicas** para realizar las

transferencias, por lo que las direcciones virtuales se tendrán que traducir antes de pasárselas a la controladora DMA.

Las **lecturas de disco sin DMA** siguen el siguiente esquema:

- La controladora del disco lee el bloque solicitado, lo coloca bit a bit en su buffer interno y comprueba que no hay errores en él.
- La controladora del disco produce una **interrupción**.
- La CPU lee el bloque del buffer palabra a palabra (pues para poder acceder al buffer se tienen que cargar sus datos en un registro) y las va almacenando en memoria.

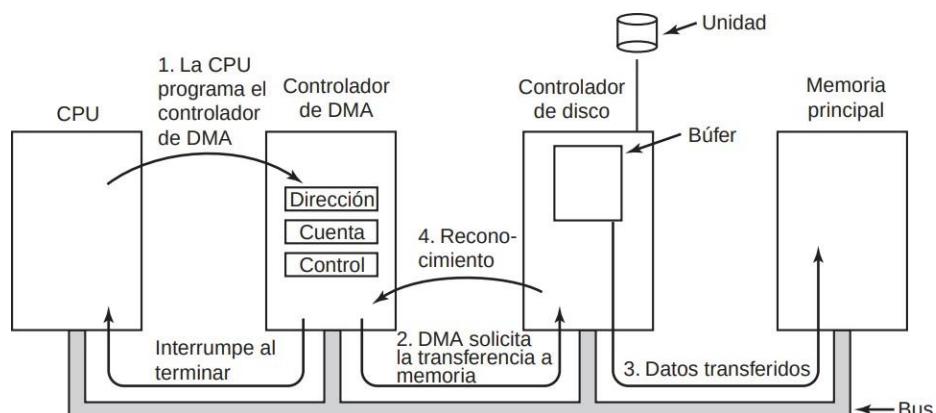
Las **lecturas de disco con DMA** siguen el siguiente esquema:

- El SO programa la controladora del DMA escribiendo en sus registros y le indica al disco que debe leer datos en su buffer y verificarlos.
- Cuando ya hay datos válidos en el búfer del disco, la controladora de DMA envía una petición de lectura al controlador del disco que incluirá la dirección de memoria principal donde se escribirán los datos.
- La controladora del disco va escribiendo los datos de su buffer en memoria principal en ciclos de bus estándar, es decir, palabra por palabra.
- Cuando acaba una escritura, le envía un reconocimiento a la controladora de DMA. Esta incrementa la dirección de memoria a escribir y disminuye la cuenta de bytes.
- Cuando la cuenta de bytes llega a 0, la controladora de DMA envía una señal a la CPU para informarle de que ya acabó la transferencia de datos.

Las controladoras DMA pueden operar en el bus de dos maneras:

- En **modo palabra**: la controladora DMA compite con la CPU por el acceso al bus, **robando ciclos** de bus a la CPU para hacer las transferencias de datos.
- En **modo bloque**: la controladora DMA indica al dispositivo que debe adquirir el bus, relajarse una **ráfaga** de transferencias y después liberarlo. El **modo de ráfaga** es mucho **más eficiente** que el robo de ciclo, pero **puede bloquear la CPU** si las ráfagas son muy largas.

**No todos los sistemas usan DMA**, pues si no hay más trabajo que realizar, es más rápido que la CPU se ocupe de las transferencias.



**FIGURA 5.4.** Operación de una transferencia de DMA.

### Concluyendo:

El **robo de ciclo** consiste en la apropiación del bus por parte de la controladora DMA de forma intermitente en momentos de actividad baja de la CPU, "robando" un ciclo de reloj para realizar sus operaciones de acceso a memoria e intercambio de datos. Dado que el proceso de apropiación del bus lleva un tiempo, la CPU puede verse ralentizada ligeramente debido a dichos "robos", ya que es considerablemente más rápida que la DMA.

Por otro lado, el modo ráfaga es un procedimiento más eficiente en general ya que consiste en la coordinación entre la DMA y el dispositivo periférico para realizar la transmisión de secuencias de datos completos, ocupando un menor número de veces el bus y tratando un mayor número de datos de cada vez. No obstante, si la secuencia de datos es extensa puede bloquear el acceso de la CPU al bus durante más tiempo.

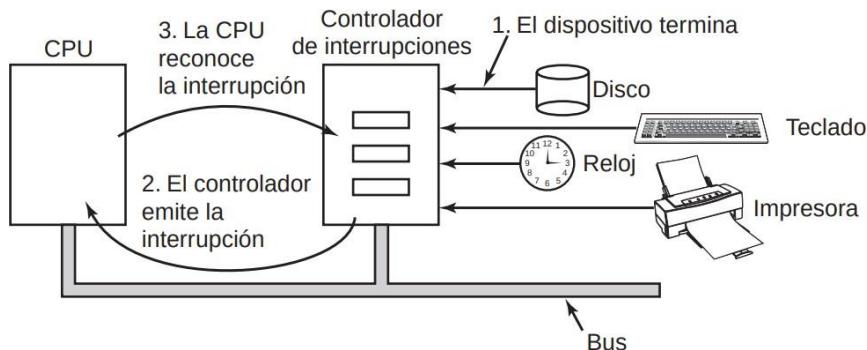
No compensa usar DMA en procesos donde las operaciones de E/S son rápidas, dado que la CPU es más veloz que la DMA. La existencia de la DMA sirve para que los demás procesos se beneficien de que la CPU no se encuentre ocupada gestionando operaciones de E/S, permitiendo así un mayor acceso a la misma.

La mayoría de los controladores de DMA utilizan direcciones físicas de memoria para sus transferencias. No todas las computadoras utilizan DMA. El argumento en contra es que la CPU principal es a menudo más rápida que el controlador de DMA y puede realizar el trabajo con mucha mayor facilidad (cuando el factor limitante no es la velocidad del dispositivo de E/S). Si no hay otro trabajo que realizar, hacer que la CPU (rápida) tenga que esperar al controlador de DMA (lento) para terminar no tiene caso.

### 5.1.5. Repaso de interrupciones

A nivel de hardware, las interrupciones funcionan de la siguiente manera:

- Cuando un dispositivo de E/S (o DMA) termina su trabajo, produce una interrupción imponiendo una señal en la línea de bus que tenga asignada.
- Esta señal es detectada por la controladora de interrupciones, que está dentro de la placa base.
- La controladora de interrupciones emite una señal para interrumpir a la CPU, pasándole un número que indica qué dispositivo de E/S (o DMA) provocó la interrupción.
- La CPU deja lo que está haciendo y comienza a ejecutar la rutina de manejo de la interrupción, cuya dirección se encuentra en la entrada del vector de interrupciones correspondiente al número del dispositivo que provocó la interrupción.
- La CPU ejecuta la rutina de interrupción, que siempre comenzará por guardar la información en la pila.



**FIGURA 5.5.** Cómo ocurre una interrupción. Las conexiones entre los dispositivos y el controlador de interrupciones en realidad utilizan líneas de interrupción en el bus, en vez de cables dedicados.

Cabe destacar que el **vector de interrupciones** es una estructura que no está relacionada con ningún proceso en particular. Es una estructura encargada de indicar al sistema operativo las rutinas asociadas a determinadas interrupciones para su posterior gestión, y **se almacena en el kernel**.

## 5.2. Fundamentos del software de E/S

### 5.2.1. Conceptos clave

Existen **2 tipos de transferencias E/S**:

- Transferencias **síncronas** → implican el bloqueo del proceso involucrado.
- Transferencias **asíncronas** → son controladas por interrupciones, la CPU inicia la transferencia y hace otras cosas hasta que llega la interrupción.

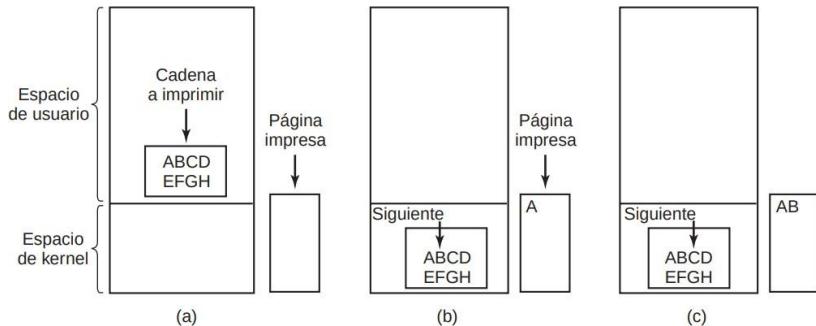
La mayoría de las operaciones de E/S son **asíncronas**, el SO es el responsable de hacer que en los programas de usuario parezcan de bloqueo. Todas las transferencias usan un **buffer en la zona de kernel de la memoria principal** como almacenamiento temporal hasta que se llevan los datos a su destino final, que en ocasiones no se conoce hasta que se examina el contenido del búfer.

- Permite realizar un chequeo de errores sobre los datos y desacoplar el llenado del vaciado del búfer.
- Su uso implica una cantidad considerable de copiado, lo que afecta al rendimiento de las operaciones de E/S.

Hay 3 maneras distintas para llevar a cabo la E/S: programada, controlada por interrupciones o mediante DMA

### 5.2.2. E/S programada

La CPU **sondea constantemente el dispositivo de E/S** para ver si está lista para continuar con la transferencia. Es la forma más simple de realizar operaciones de E/S. El problema es que se **ocupa la CPU hasta que se finaliza la operación**.



Ejemplo: syscall para imprimir “ABCDEFGH” en una impresora.

- El SO copia el búfer del espacio de usuario en un array en el espacio de kernel para usarlo con más facilidad.
- El SO comprueba si la impresora está disponible. Si no lo está, espera hasta que lo esté.
- Tan pronto como la impresora esté disponible, el SO copia un carácter al registro de datos de la impresora.
- Se repiten los pasos 2 y 3 hasta finalizar la cadena.

Las acciones realizadas por el sistema operativo se sintetizan en la figura 5.7. El aspecto esencial de la E/S programada, que se ilustra con claridad en esta figura, es que después de imprimir un carácter, la CPU sondea en forma continua el dispositivo para ver si está lista para aceptar otro. Este comportamiento se conoce comúnmente como **sondeo u ocupado en espera**. La E/S programada es simple, pero tiene la desventaja de ocupar la CPU tiempo completo hasta que se completen todas las operaciones de E/S.

```
copiar_del_usuario(bufer, p, cuenta);           /* p es el búfer del kernel */
for (i=0; i<cuenta; i++) {                      /* itera en cada carácter */
    while (*reg_estado_impresora != READY);      /* itera hasta que esté lista */
    *registro_datos_impresora = p[i];            /* imprime un carácter */
}
regresar_al_usuario();
```

### 5.2.3. E/S controlada por interrupciones

La CPU comienza la transferencia y **se planifica otro proceso**, de manera que el que comenzó la transferencia queda **bloqueado** hasta que esta finalice. Sin embargo, el dispositivo de E/S **generará muchas interrupciones** para poder realizar la operación, que tendrá que resolver la CPU, **desperdimando tiempo de CPU**.

Ejemplo: syscall y manejador de interrupciones para imprimir “ABCDEFGH” en una impresora.

1. El SO copia el búfer del espacio de usuario en un arreglo en el espacio de kernel para usarlo con más facilidad.
2. Tan pronto como la impresora esté disponible se envía el primer carácter.
3. La CPU llama al planificador y se bloquea el proceso.
4. Cuando la impresora ha impreso el carácter y puede recibir otro, genera una interrupción.
5. La interrupción bloquea el proceso actual y ejecuta el procedimiento de manejo de interrupciones de la impresora.
6. Si no hay más caracteres en la cadena, se desbloquea el proceso.
7. Si hay más caracteres por imprimir, se le envía a la impresora el siguiente y la CPU vuelve a hacer lo que estaba haciendo antes de la interrupción

```
copiar_del_usuario(bufer, p, cuenta);
habilitar_interrupciones();
while (*reg_estado_impresora != READY);
*registro_datos_impresora = p[0];
planificador();

if (cuenta==0) {
    desbloquear_usuario();
} else {
    *registro_datos_impresora = p[i];
    cuenta=cuenta - 1;
    i = i + 1;
}
reconocer_interrupcion();
regresar_de_interrupcion();
```

(a)

(b)

**Figura 5-9.** Cómo escribir una cadena a la impresora, usando E/S controlada por interrupciones. (a) Código que se ejecuta al momento en que se hace una llamada al sistema para imprimir. (b) Procedimiento de servicio de interrupciones para la impresora.

### 5.2.4. E/S mediante el uso de DMA

La **controladora DMA realiza la transferencia sin intervención de la CPU**. Es como una E/S programada pero el trabajo lo realiza la DMA en lugar de la CPU.

**Libera la CPU** durante la operación de E/S para realizar otro trabajo. Sin embargo, **requiere hardware especial**.

Ejemplo: syscall y manejador de interrupciones para imprimir “ABCDEFGH” en una impresora.

```
copiar_del_usuario(bufer, p, cuenta);
establecer_controlador_DMA();
planificador();
reconocer_interrupcion();
desbloquear_usuario();
regresar_de_interrupcion();
```

**FIGURA 5.9.** Cómo imprimir una cadena mediante el uso de DMA.

(a) Código que se ejecuta cuando se hace la llamada al sistema para imprimir.

(b) Procedimiento de servicio de interrupciones.

## 5.3. Cosas Importantes de este Tema

Saber muy bien cómo funcionan los distintos tipos de acceso a memoria o chaparlos porque siempre pregunta uno y cómo funciona.