

ISO-FINAL

03-Procesos 1

¿Qué es un proceso?

Un proceso es un **programa en ejecución**. Mientras que un programa es un conjunto de instrucciones, el proceso representa una instancia activa de ese programa.

Un proceso cobra vida cuando un usuario solicita ejecutar un programa. Desde ese momento, pasa a ser una entidad dinámica, con capacidad de cambiar su estado, su estructura, consumir recursos, interactuar con otros procesos y seguir una trayectoria particular de ejecución.

Diferencia entre programa y proceso

Característica	Programa	Proceso
Naturaleza	Es estático	Es dinámico
Contador de programa	No tiene	Tiene (almacena la dirección de la próxima instrucción a ejecutar)
Tiempo de vida	Desde que se crea/edita hasta que se borra	Desde que se ejecuta hasta que finaliza

El modelo de proceso

Dado que los sistemas operativos modernos están diseñados para ser **multiprogramados**, pueden mantener **múltiples procesos activos al mismo tiempo**.

Esto significa que en memoria pueden coexistir varios procesos, cada uno con su propio estado, historia de ejecución y contador de programa. Cada proceso es una **entidad independiente**.

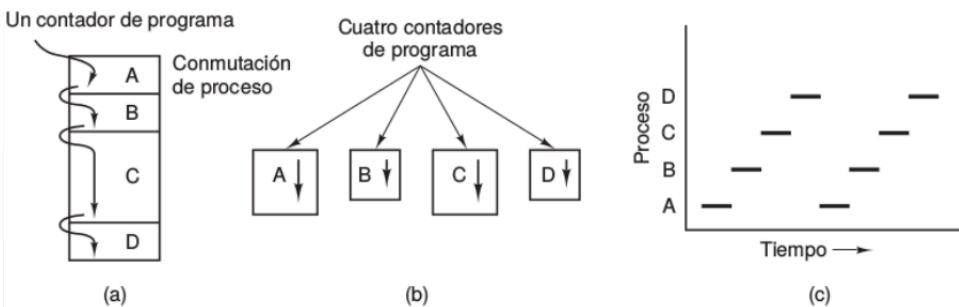


Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo hay un programa activo a la vez.

Importante: En esta materia se asume que hay un único procesador (CPU). Por lo tanto, en **cualquier instante de tiempo, solo un proceso se encontrará activo (ejecutándose)**. El resto de los procesos pueden estar en espera, bloqueados o listos para ejecutarse.

Componentes de un proceso

Un proceso es una **entidad de abstracción** que permite representar la ejecución de un programa.

Esta abstracción encapsula toda la información necesaria para que el programa pueda ejecutarse correctamente. Por lo tanto, un proceso para poder ejecutarse debe incluir como mínimo:

- **Sección de código (texto)**

Contiene el código del programa que se va a ejecutar. Es la parte estática, que no cambia durante la ejecución.

- **Sección de datos (variables globales)**

Incluye todas las variables globales del programa. Estas variables pueden cambiar a lo largo del tiempo de ejecución.

- **Stacks (datos temporarios: parámetros, variables temporales y direcciones de retorno)**

Cada proceso tiene uno o más stacks. En general, se utiliza **una pila para el modo usuario y otra para el modo kernel**.

- El stack se usa para **almacenar información temporal** como: parámetros de funciones, variables locales, direcciones de retorno (para volver al punto correcto después de ejecutar una función).

- Los stacks **se crean automáticamente y su tamaño se ajusta dinámicamente** en tiempo de ejecución (*run-time*).

- Están compuestos por **stack frames**, que son bloques de datos apilados cuando se llama a una rutina (**push**) y desapilados cuando esta finaliza (**pop**).

El stack frame tiene los parámetros de la rutina (variables locales) y datos necesarios para recuperar el stack frame anterior (el contador de programa y el valor del stack pointer en el momento del llamado)

Atributos de un proceso

Como toda entidad, un proceso posee un conjunto de **atributos** que lo identifican y caracterizan:

- Identificación del proceso (PID) y del proceso padre (PPID)
- Identificación del usuario que lo "disparo"/inició
- Si hay estructura de grupos, el grupo al que pertenece
- En ambientes multiusuario, desde qué terminal y quien lo ejecutó.

Toda esta información (y mucha más) se guarda en una estructura de datos especial llamada **Process Control Block (PCB)**.

Process Control Block (PCB)

La PCB es una **estructura de datos** asociada a cada proceso. Existe una por proceso.

Es lo **primero que se crea** cuando se crea un proceso y lo **último que se borra** cuando termina.

La PCB logra **abstracción** ya que encapsula y contiene toda la información asociada con cada proceso en una única estructura accesible para el SO.

Dentro del contenido típico del PCB se encuentra:

- Identificadores: PID, PPID, IDetc
- Valores de los registros de la CPU (PC, AC, etc)
- Planificación (estado, prioridad, tiempo consumido, etc)
- Ubicación en memoria (dirección de memoria asignada)
- Contabilidad (Accounting): cuanta memoria ocupó, cuantas operaciones E/S necesitó, tiempo de CPU, etc.). Todo esto sirve para saber qué recursos otorgarle al proceso.
- Información entrada salida (estado, operaciones E/S pendientes, etc).

¿Por qué es importante?

Cuando un proceso es interrumpido y deja de estar en un estado de ejecución (por ejemplo, porque el sistema decide ejecutar otro proceso) es necesario guardar **todo su contexto de ejecución** (registros, PC, etc.) en un lugar para luego poder volver a ejecutarlo sin problemas.

Esta información se almacena en su PCB, de manera que **pueda reanudarse más adelante desde el mismo punto** en el que fue interrumpido, sin pérdida de datos ni estado.

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

Campos Comunes

¿Qué es el espacio de direcciones de un proceso?

El espacio de direcciones de un proceso es el conjunto de direcciones de memoria que ocupa y puede utilizar el proceso durante su ejecución.

Representa cómo está organizada la memoria de un proceso.

El espacio de direcciones típicamente incluye:

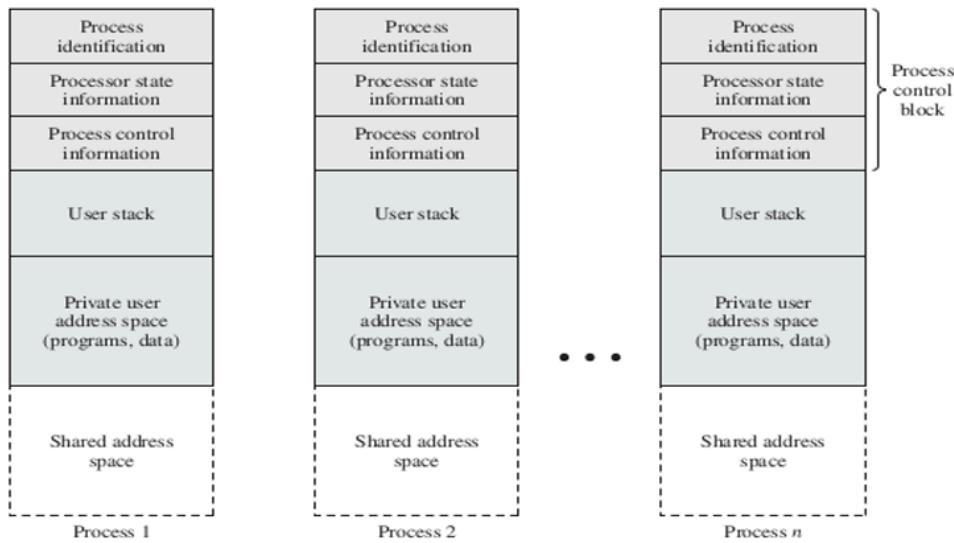
- stack, text y datos.

No incluye su PCB o tablas asociadas (ni otras estructuras del SO).

Modo usuario vs. modo kernel →

Mientras el proceso se ejecuta en modo usuario, puede acceder solo a su espacio de direcciones. El SO impone este límite como medida de seguridad y protección para evitar que el proceso interfiera con memoria de otros procesos.

Por otro lado, cuando el proceso se ejecuta en modo kernel, se tiene acceso completo a todas las estructuras internas (como la PCB de cualquier proceso) e inclusive a los espacios de direcciones de otros procesos, por lo tanto, ya no existe ese límite.



El contexto de un proceso

El contexto es toda la información que el SO necesita para administrar el proceso y la CPU para ejecutarlo correctamente.

Son parte del contexto los registros de cpu, inclusive el contador de programa, prioridad del proceso, si tiene E/S pendientes (los campos de la PCB, etc).

Cambio de Contexto (Contexto Switch)

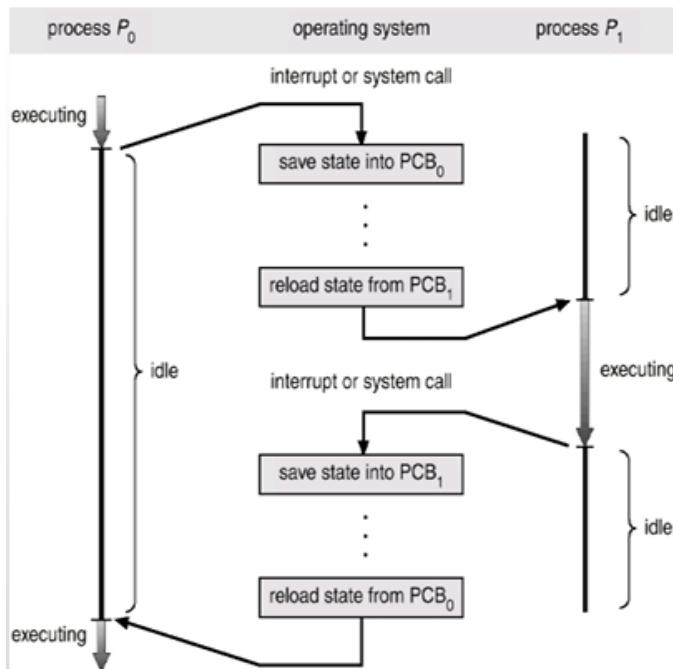
El cambio de contexto se produce cuando la CPU interrumpe la ejecución de un proceso y comienza a ejecutar otro. Es decir, se produce cuando la CPU cambia de un proceso a otro.

¿Qué ocurre durante un cambio de contexto?

- El proceso que estaba en ejecución es interrumpido. Cuando un proceso se saca, el SO debe guardar todo el contexto del proceso saliente, que pasa a estar en espera y se retornará después a la CPU.
- Luego, se debe cargar el contexto del nuevo proceso a ejecutar y continuar su ejecución desde la instrucción siguiente a la última ejecutada en dicho contexto (se posiciona PC).

El cambio de contexto consume tiempo de CPU, a este tiempo se lo considera **tiempo no productivo**. Esto no significa que no se utilice la CPU, si no que este tiempo **no** se utiliza para ejecutar instrucciones de ningún proceso.

El **tiempo total que tarda un cambio de contexto** depende de múltiples factores: La eficiencia del SO, el soporte que brinde el hardware, etc.



Kernel del sistema operativo

Es un **conjunto de módulos de software**. El kernel no es un único bloque monolítico de código, si no que está compuesto por varias partes, cada una con una función bien definida dentro del SO.

Se ejecuta en el procesador **como cualquier otro proceso**, pero NO es un proceso. El concepto "proceso" se reserva para programas de usuario.

Se ejecuta en modo privilegiado (o modo kernel) y tiene acceso completo.

Hay diferentes enfoques de diseño

Existen diferentes formas de integrar el kernel dentro del sistema operativo...

Enfoque 1: El kernel como entidad independiente

En este enfoque, el kernel se ejecuta como una **entidad independiente** (en modo privilegiado) separada de los procesos de usuario. Tiene su propia región de memoria, su propio stack y se ejecuta **frente al espacio de direcciones de los procesos**.

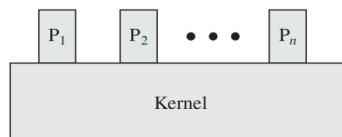
El kernel no es un proceso. El concepto de "proceso" se reserva para programas de usuario.

Es una arquitectura utilizada por los primeros SO.

¿Cómo funciona?

- Cada vez que ocurre una interrupción, se guarda el contexto completo del proceso saliente (el proceso que está en ejecución)
- El control se pasa al kernel del sistema.
- Finalizada su actividad, le devuelve el control al proceso interrumpido (o a otro diferente)

Desventaja → Esta arquitectura introduce una sobrecarga muy grande, ya que siempre es necesario realizar un cambio de contexto para ejecutar el código del kernel, incluso si se va a continuar con el mismo proceso después.



2- Enfoque 2: El kernel dentro del proceso

En este enfoque, el código del kernel está dentro de todos los procesos, es decir, está incluido dentro del espacio de direcciones de cada proceso. Sin embargo, no está duplicado: el código del kernel es compartido entre todos los procesos.

Dentro de un proceso se encuentra

- El código del programa (user)
- El código del kernel (el código de los módulos de SW del SO)
- Su propio stack (uno en modo usuario y otro en modo kernel) → se necesitan dos pilas ya que el proceso se ejecuta en modo usuario y el kernel se ejecuta en modo kernel

Importante: El **kernel se ejecuta en el MISMO contexto** que algún proceso de usuario.

Si ocurre una interrupcion mientras un proceso está ejecutándose, el cambio de contexto (de usuario a kernel) ocurre **sin necesidad de cambiar de proceso**.

Cada interrupcion (incluyendo las de system call) es atendida en el contexto del proceso que se encontraba en ejecución. **El kernel se ejecuta directamente en el contexto de ese proceso, usando su pila de kernel.**

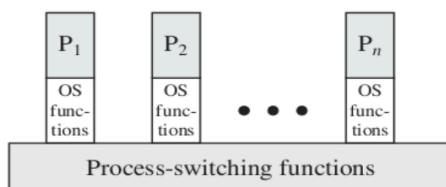
- Basicamente, se resuelve accediendo a las direcciones pertenecientes al código del kernel dentro de cada proceso.

Si el SO determina que el proceso debe seguir ejecutandose luego de atender la interrupcion, cambia a modo usuario y le devuelve el control

Ventajas:

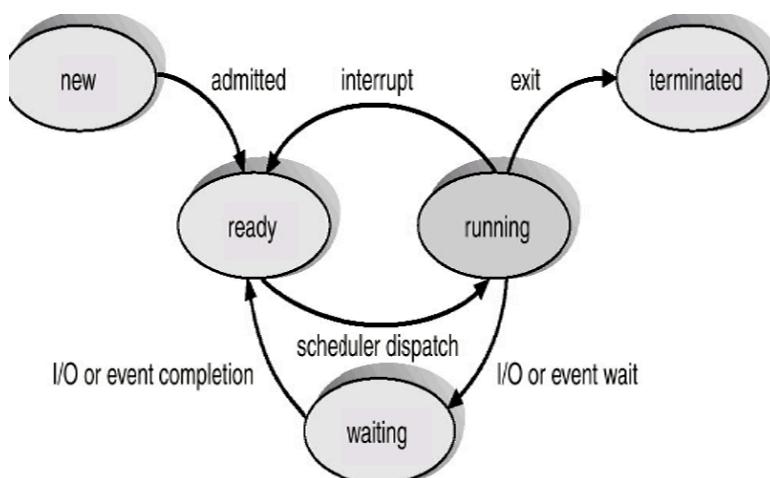
- No requiere un cambio de contexto completo, lo que reduce el overhead.
- Es más eficiente y economico en términos de rendimiento.
- Permite una **atención más rápida de interrupciones** y llamadas al sistema.

Observación: En este modelo, el kernel se puede ver como un **conjunto de rutinas** que son llamadas y utilizadas por los procesos cuando es necesario.



Estados de un proceso

Durante su ciclo de vida, un proceso puede pasar por distintos **estados**. Estos estados reflejan la situación actual del proceso en relación con la **CPU** y **los recursos del sistema**.



- **Nuevo (new)**
 - Es el estado de inicialización donde el proceso se está creando. Un usuario no crea directamente un proceso, lo “dispara”. Un proceso es creado por otro proceso, su proceso “padre”.
 - Se crean e inicializan sus estructuras asociadas (por ejemplo, su PCB)

- El proceso queda en la cola de procesos, normalmente en espera de ser cargado en memoria. Todavía no ha sido admitido en el sistema, es decir, no está listo para competir por la CPU.

Cuando el proceso está completo y se carga en memoria, el proceso es admitido y puede cambiar al estado →

- **Listo para ejecutar (ready)**

- Luego que el scheduler de largo plazo eligió al proceso para cargarlo en memoria, el proceso queda en estado de listo.
- El proceso tiene todo lo necesario para ejecutarse: tiene memoria, su PCB está inicializada, no necesita esperar ningún recurso adicional.
- Sin embargo, el proceso todavía no tiene asignada la CPU. Esta en la cola de procesos listos esperando su turno para ser seleccionado, por lo tanto, compite activamente por el uso de la CPU.

Cuando el proceso es elegido por el planificador, se realiza un cambio de contexto (context switch) y el proceso puede pasar al estado →

- **Ejecutándose (running)**

- El proceso está siendo ejecutado por la CPU. El scheduler de corto plazo lo eligió para asignarle la CPU.
- Tendrá la CPU hasta que se termine el período de tiempo asignado (quantum o time slice), termine o hasta que necesite realizar alguna operación de E/S.

Durante ese estado, se puede tomar más de un camino →

- **En espera (waiting)**

- Si el proceso, por algún motivo, se interrumpe y se saca de ejecución, pasa al estado de espera para volver a competir por la CPU (hay un cambio de contexto, context switch). Sigue en memoria, pero no tiene la CPU.
- El proceso necesita que se cumpla el evento para continuar. Al cumplirse el evento, pasará al estado de listo.
- Si un proceso necesita realizar una operación de E/S (por ejemplo, leer un archivo de un disco), pasa al estado de espera. Como depende de la finalización de la operación de E/S no puede continuar ejecutándose.

Una vez completada la operación, vuelve al estado Ready para competir por la CPU.

- Un proceso puede ser interrumpido por el sistema operativo (por ejemplo, por un **timer** que da lugar a una política de planificación por tiempo). En ese caso, el proceso cede la CPU, pero no está esperando recursos externos. Por eso, simplemente regresa al estado Ready para volver a competir por el procesador.
- **Terminado(terminated)**
 - El proceso ha finalizado su ejecución correctamente o fue abortado por el sistema.
 - Se liberan recursos. Se eliminan las estructuras asociadas, todo lo que estaba en memoria. La PCB es lo último que se elimina, una vez eliminada, el proceso deja de existir para el SO.
 - Este estado es lo contrario al estado new.

Posibles transiciones

- **New-Ready:** Por elección del scheduler de largo plazo (carga en memoria)
- **Ready-Running:** Por elección del scheduler de corto plazo (asignación de CPU)
- **Running-Waiting:** el proceso “se pone a dormir”, esperando por un evento.
- **Waiting-Ready:** Terminó la espera y compite nuevamente por la CPU.
- **Running-Ready (caso especial):** Cuando el proceso termina su quantum (franja de tiempo) sin haber necesitado ser interrumpido por un evento, pasa al estado de ready para competir por CPU, no necesita esperar por ningún evento. A diferencia de los demás casos, el proceso es expulsado de la CPU contra su voluntad. Esta situación se da en algoritmos apropiativos

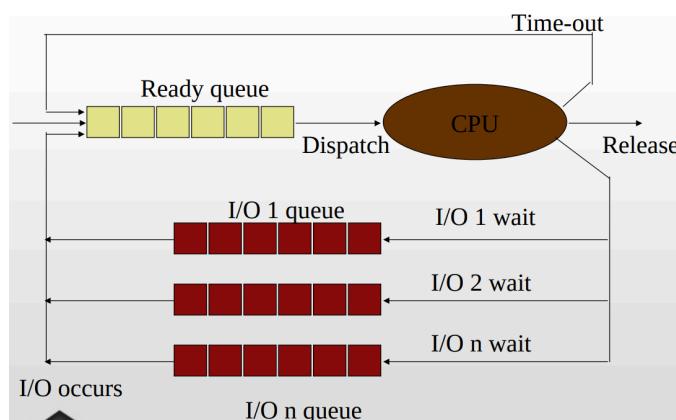
Colas en la planificación de procesos

En un sistema operativo, las colas **son estructuras de datos** que enlazan/relacionan las PCBs de los procesos según los estados en que se encuentran dichos procesos.

El sistema operativo utiliza la PCB como una representación o abstracción de cada proceso, y, en base al estado de cada proceso, la PCB se coloca en una cola correspondiente.

Por ejemplo:

- La cola de procesos listos: contiene las PCBs de procesos que están en memoria principal y listos para ejecutarse, esperando a que la CPU esté disponible.
- La cola de dispositivos: contiene las PCBs de procesos que están esperando completar una operación de E/S.
- La cola de trabajos o procesos: contiene todas las PCBs de procesos en el sistema (incluyendo los que aún no están en memoria principal).



Un aspecto importante es que las PCBs no se mueven físicamente en memoria, se crean en un lugar fijo dentro de la memoria. Lo que sí cambia, según el estado de los procesos, son los enlaces entre las PCBs.

Aunque se las llama "colas", **no siempre funcionan como una cola estricta (FIFO)**. Algunas pueden comportarse de otra manera, pueden estar organizadas por prioridad, tiempo de llegada, u otros criterios, dependiendo del algoritmo de planificación que utilice el sistema operativo.

Modulos de la planificación

La planificación de procesos es una **función** fundamental del SO que se encarga de **decidir qué proceso se ejecutará y cuándo**.

La planificación se realiza mediante **módulos del SO**, es decir, pedazos de software que realizan distintas tareas **llamados schedulers** (planificadores). Estos módulos se ejecutan ante determinados eventos que así lo requieren como:

- Creacion o finalización de procesos
- Eventos de sincronizacion o de E/S
- Finalización del lapso de tiempo asignado a un proceso

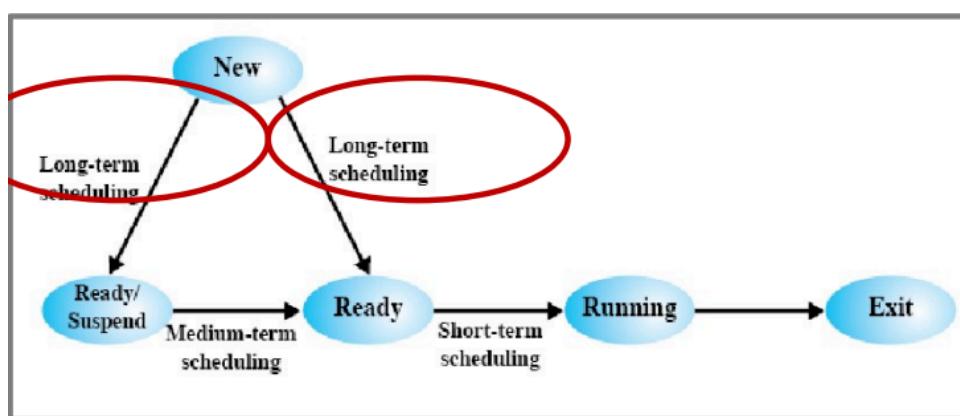
Los principales modulos de la planificación son

- **Scheduler de long term**

Función principal: Controla el *grado de multiprogramación*, es decir, la cantidad de procesos que pueden estar en memoria al mismo tiempo.

Es el que realiza la actividad de **admitir nuevos procesos en el sistema**, pasando a un proceso creado en estado "new" a estado "ready". Para esto, trabaja junto con el **loader**.

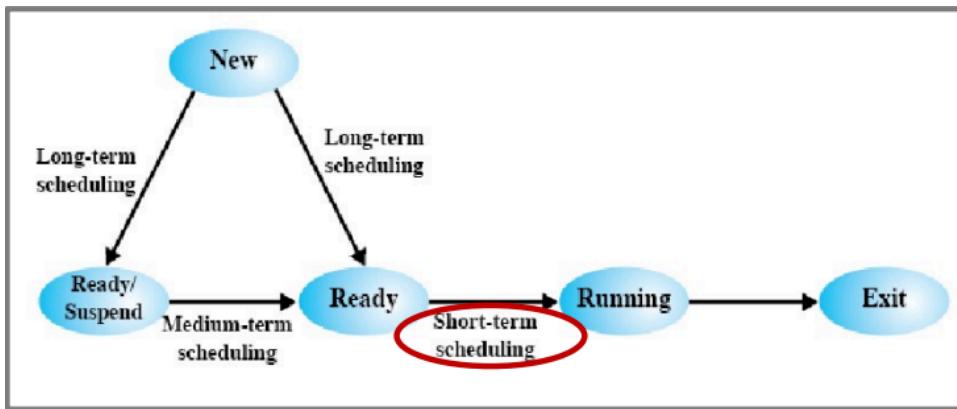
Observación: Puede no existir este scheduler y absorber esta tarea el de short term.



- **Scheduler de short term**

Función principal: Selecciona qué proceso de la cola de Ready usará la CPU. Trabaja junto con el **dispatcher**.

Términos asociados: planificación apropiativo (la CPU puede ser retirada de un proceso en ejecución), no apropiativo, algoritmo de scheduling (Round Robin, FIFO, SJF, etc.).



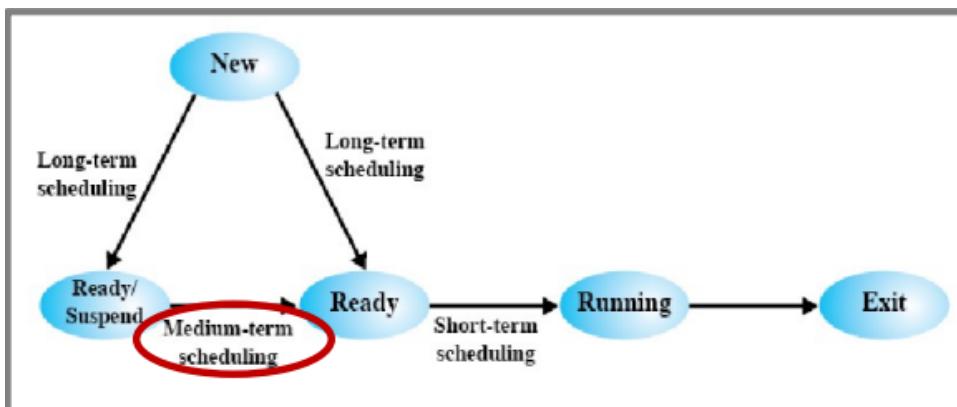
- **Scheduler de medium term**

Función principal: Ajustar dinámicamente el grado de multiprogramación, sacando temporalmente de memoria los procesos que sean necesarios para mantener el equilibrio y rendimiento del sistema.

En determinadas circunstancias, es necesario bajar el grado de multiprogramación. No siempre resulta favorable tener muchos procesos en memoria.

Las operaciones asociadas son:

- **Swap out:** cuando se saca un proceso de la memoria.
- **Swap in:** cuando se vuelve a cargar un proceso en memoria.



Los nombres **long**, **medium** y **short** hacen referencia a la **frecuencia con la que se ejecuta cada planificador**.

- El scheduler de **largo plazo** se ejecuta con poca frecuencia (solo cuando se crean procesos).
- El de **mediano plazo** actúa ocasionalmente, según la carga del sistema.

- El de **corto plazo** se ejecuta con mayor frecuencia que los otros schedulers, ya que interviene en cada decisión de asignación del procesador.

Otros módulos relacionados

- **Dispatcher**

Es el módulo que realiza el **cambio de contexto** (pasa del modo kernel al modo usuario) y pasa el control de la CPU al proceso **seleccionado por el scheduler de corto plazo (short term)**.

También se encarga del **cambio de modo de ejecución**, pasando del modo núcleo (kernel) al modo usuario para que el proceso retome su ejecución.

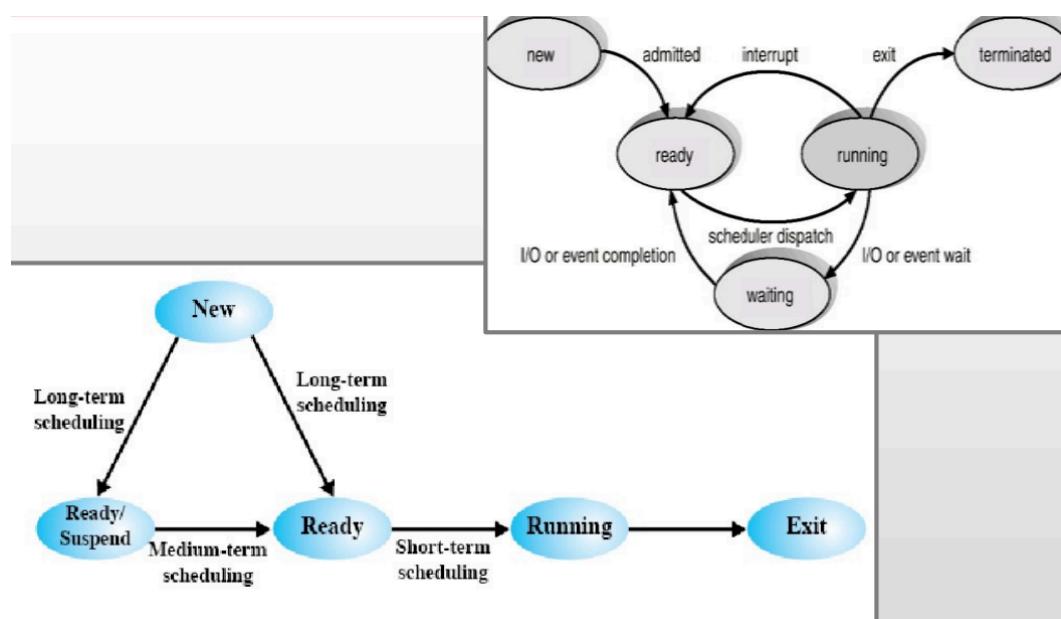
- **Loader.**

Encargado de **cargar en memoria** el proceso seleccionado por el scheduler de largo plazo (long term).

Arma el espacio de direcciones del proceso, cargando su código, datos y pila en memoria.

Nota: En algunos sistemas, los módulos **loader** y **dispatcher** no están separados formalmente, pero sus **funciones deben cumplirse**.

Estados y schedulers



Procesos en espera y swapeados

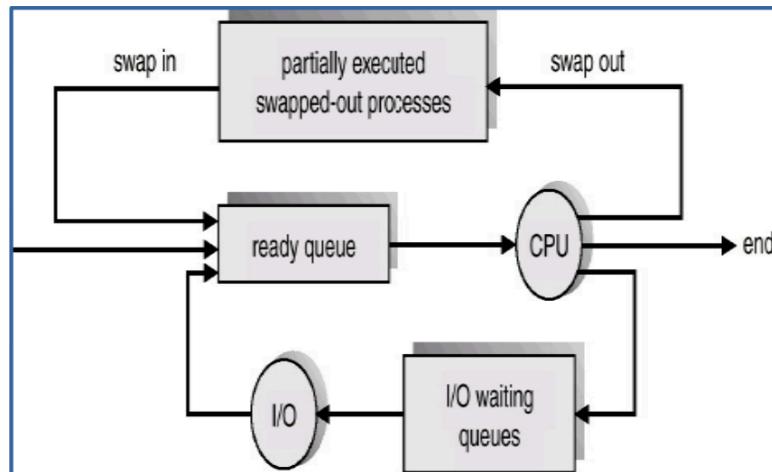
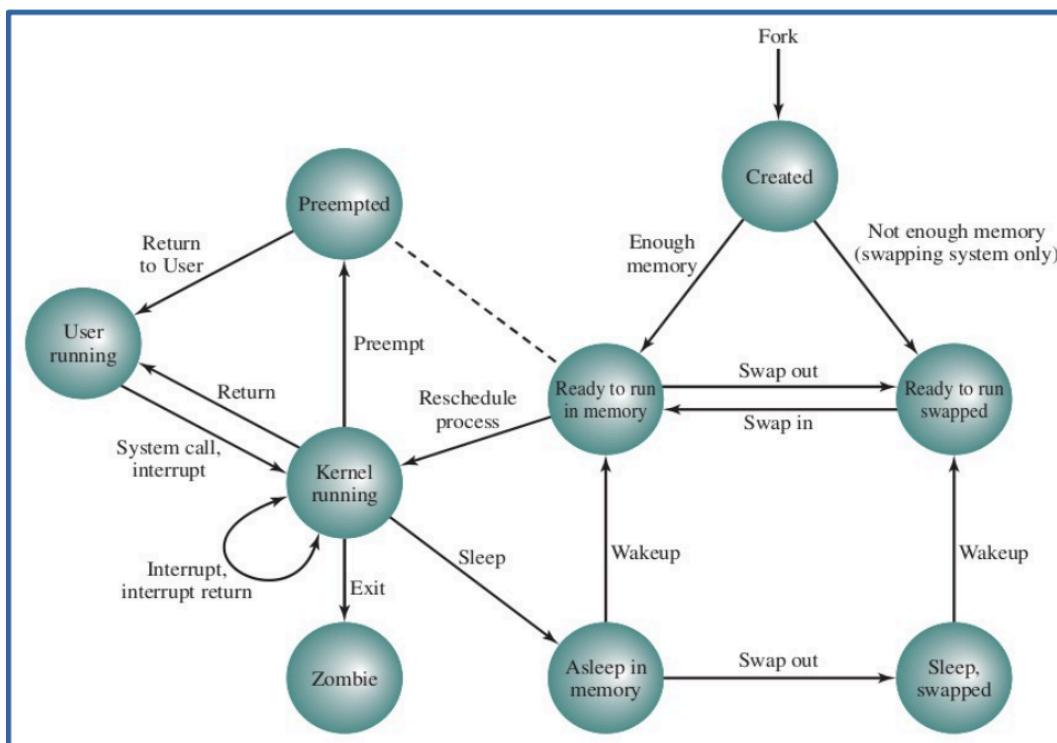


Diagrama de transiciones UNIX



Creación del proceso: Created

Un nuevo proceso se crea mediante la llamada al sistema `fork()`. Desde aquí, el proceso puede tomar dos caminos según la disponibilidad de memoria:

a. Si hay suficiente memoria, el proceso pasa al estado:

→ **Ready to run in memory** (Listo para ejecutarse en memoria)

b. Si NO hay suficiente memoria, se va a:

→ **Ready to run swapped** (Listo para ejecutarse pero está fuera de memoria). En este estado, el proceso está completamente inicializado, pero no puede

ejecutarse aún porque no está cargado en memoria principal.

Estado Ready: Esperando CPU

Cuando el planificador lo selecciona, el proceso pasa a ejecutarse en:

→  **Kernel running** (modo kernel): Se realiza el cambio de contexto. En este modo, pueden ocurrir interrupciones, pero no implican cambio de estado, ya que el kernel ya tiene privilegios.

Recién cuando todo está listo, el proceso puede cambiar a:

→  **User running** (modo usuario): En este modo, puede producirse una **interrupción o llamada al sistema**, que lo lleva de nuevo a modo kernel (**Kernel running**) para ser atendida.

Salida de ejecución

El proceso puede abandonar la CPU por varios motivos:

Apropiación (preempt): El sistema le saca la CPU al proceso antes de que termine, involuntariamente. El proceso pasa al estado preempted (apropiación).

Espera por un evento (Asleep in memory) 😴: El proceso espera que termine un evento mientras se encuentra cargado en memoria.

- También está el estado "**Sleep, swapped**". El proceso se encuentra en dicho estado si es "swappeado". Es decir, si se le hace un swap out para sacarlo de memoria y así, bajar el grado de multiprogramación.

Línea punteada

Si un proceso se encuentra en estado "**preempted**", **puede volver al estado "ready to run in memory"** cuando sea replanificado. Es decir, cuando el planificador vuelve a considerar al proceso y lo pone nuevamente en la cola de "Ready" donde espera que le asignen la CPU otra vez. (línea punteada)

Si un proceso se encuentra en estado "ready to run", puede volver al estado "preempted". Si bien, el término preempted suele referirse a cuando un proceso es interrumpido y le quitan la CPU. En algunos casos, indica una **etapa intermedia en que el planificador elige un proceso y está por transferirle el control**. De esta manera, el sistema puede realizar algunas tareas antes de habilitar al proceso (como verificar prioridades, contexto, etc.).

Finalización del proceso

- Cuando el proceso termina, pasa al estado:

→  **Zombie**

Aquí se comienza a destruir el proceso: liberar memoria, cerrar archivos, etc. **La PCB es lo último que se elimina.** Una vez eliminada, el proceso **deja de existir** por completo.

05-PROCESOS 3

Comportamiento de los procesos

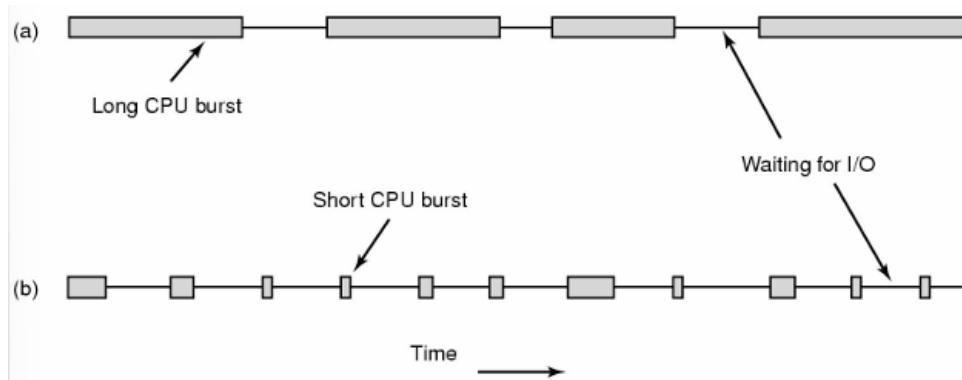
Cuando se necesita decidir **qué algoritmo de planificación de procesos usar**, debe **conocer el tipo de procesos** que se van a ejecutar:

Ejemplos de tipo de procesos:

- CPU-bound: Mayor tiempo utilizando la CPU
- I/O-bound: Mayor parte del tiempo esperando por I/O

Como los **procesos I/O-bound** pasan la mayoría del tiempo **esperando** que el disco, teclado u otro dispositivo termine su trabajo, es clave **atenderlos rápido cuando necesitan un poco de CPU**. Si no fuera así, quedan sin uso, desaprovechándose. **Mientras tanto, los procesos CPU-bound pueden aprovechar los momentos en que la CPU está libre.**

Importante: Los dispositivos de E/S son **más lentos** que la CPU, por eso **hay que evitar que estén inactivos**, lo cual se logra procesando rápido los pedidos de los procesos I/O-bound.



(a) Proceso **CPU-bound**:

- Los rectángulos grises grandes representan períodos largos en los que el proceso está usando la **CPU** intensivamente. Entre esos periodos, hay momentos de espera por **E/S (I/O)**, pero son escasos.

- Este tipo de proceso realiza mucho **cálculo o procesamiento**, y accede poco a dispositivos como discos o redes.

(b) Proceso **I/O-bound**:

- Este tipo de proceso depende mucho de dispositivos externos (disco, red, teclado, etc.), y utiliza la CPU solo por **cortos períodos**.

¿Qué es la planificación?

La **planificación de procesos** es una función del sistema operativo en un entorno **multiprogramado**. Consiste en decidir **cuál de todos los procesos que están listos (en estado Ready)** debe ejecutarse a continuación.

¿Qué es un algoritmo de planificación?

Es el algoritmo utilizado para realizar la planificación del sistema

Algoritmos apropiativos (preemptive)

En estos algoritmos existen situaciones que hacen que el proceso en ejecución sea expulsado de la CPU, incluso si este aún no ha terminado.

Esto sucede, por ejemplo, cuando:

- Llega un proceso con mayor prioridad.
- Se cumple con un **quantum de tiempo** en un sistema con **round robin**.
- Se produce una **interrupción de reloj** (timer interrupt) que obliga a revisar si hay otro proceso con mayor prioridad que debería ejecutarse o si ya pasó el tiempo asignado del proceso actual.

Algoritmos NO apropiativos (nonpreemptive)

En estos algoritmos los procesos se ejecutan hasta que ellos mismos (por su propia cuenta) abandonan la CPU, es decir, permanecen ejecutándose hasta que voluntariamente la liberan.

Esto sucede, por ejemplo, cuando:

- El proceso finaliza.
- El proceso se bloquea por una operación de entrada/salida (E/S).

En los algoritmos no apropiativos, **no hay decisiones de planificación durante las interrupciones de reloj** porque **el SO no interrumpe al proceso** que ya está

ejecutándose. Si un proceso se está ejecutando y ocurre una interrupción del reloj, no se hace nada y el proceso sigue ejecutándose.

Categorías de los algoritmos de planificación

Como los algoritmos de planificación tienen como fin decidir qué proceso de todos los que compiten por el procesador (CPU) usará la CPU y cuándo pero **no todos los sistemas tienen los mismos objetivos, por lo tanto, según el ambiente/contexto del sistema, se pueden priorizar distintas metas.**

Sin embargo, una meta en común muy importante:

- **Productividad de la CPU:** Que la CPU esté lo más ocupada posible. Es decir, minimizar los tiempos en que está "ociosa".

Otras posibles metas:

- **Equidad (fairness):** Que ningún proceso "retenga" la CPU por mucho tiempo, ni que otros queden indefinidamente esperando (*starvation*). Otorgar una parte justa de la CPU a cada proceso.
- Balance: Mantener ocupadas todas las partes del sistema
- **Balance de recursos:** No solo la CPU debe estar ocupada, sino también otros componentes (disco, memoria, red). Idealmente, se busca tener todas las partes ocupadas y que **todo el sistema esté equilibrado**.

Ejemplos →

- Procesos por lotes (batch)

Estos sistemas ejecutan trabajos sin intervención del usuario durante la ejecución.

El usuario carga un conjunto de tareas, y luego recibe los resultados. No hay interacción en tiempo real.

Características:

- No existen usuarios que esperen una respuesta en una terminal. No importa tanto la respuesta rápida, si no el rendimiento global.
- Se pueden utilizar algoritmos no apropiativos porque no hace falta interrumpir un proceso para ejecutar otro (nadie está esperando algo en pantalla)

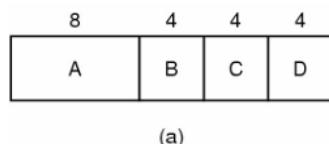
Metas de este tipo de algoritmos:

- Rendimiento: Maximizar la cantidad de trabajos completados por unidad de tiempo.
- Tiempo de retorno: Minimizar el tiempo de retorno, es decir, los tiempos entre el comienzo y la finalización.
- Tiempo de espera: Tiempo que un proceso pasa en la cola antes de que le asignen la CPU. Puede verse afectado.
- Uso de la CPU: Mantener la CPU ocupada la mayor cantidad de tiempo posible

Ejemplos de algoritmos:

- **FCFS - First Come First Served**

Se comporta como una **cola FIFO**: el primero que llega es el primero que se atiende. Simple de implementar.



Tiempos de retorno: A=8, B=12, C=16, D=20.

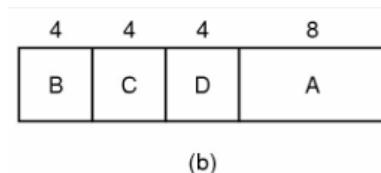
Pero puede tener el problema de que trabajos largos bloqueen a otros más cortos, lo que lleva a **tiempos de espera altos para procesos cortos**.

- **SJF - Shortest Job First**

Se elige **el proceso que tiene el menor tiempo de ejecución estimado**. Es decir, entre todos los procesos listos para ejecutarse, se selecciona **el que tarde menos** en completarse (menos CPU requiera).

No se basa en el orden de llegada como FCFS, sino en cuánto tiempo le queda al proceso para terminar.

Es difícil de implementar en la práctica ya que el sistema no siempre puede saber cuánto tiempo va a durar un proceso. Además, puede causar una posible starvation si siguen llegando procesos cortos y los largos pueden quedarse esperando indefinidamente.



Tiempos de retorno: B=4, C=8, D=12, A=20.

Observación: Sumando todos los tiempos, **SJF logra un menor tiempo promedio de retorno**, lo cual lo hace más eficiente en este tipo de entornos.

- Procesos interactivos

Hay interacción con los usuarios. Por lo tanto, este tipo de procesos requiere respuesta en tiempos breves.

Un servidor interactivo necesita ejecutar múltiples procesos (o hilos) concurrentemente para poder atender las múltiples solicitudes de los usuarios sin demoras.

Características:

- Procesos interactivos pueden ejecutarse de manera concurrente, y cada uno compite por usar la CPU.
- Son necesarios **algoritmos apropiativos** para evitar que un proceso acapare la CPU y **quitarle la CPU a un proceso** para dársela a otro si es necesario.
 - ! Si se usaran algoritmos no apropiativos (como FCFS), un proceso largo podría bloquear la ejecución de tareas que requieren inmediatez, como tipar en Word.

Metas propias de este tipo de algoritmos:

- **Tiempo de respuesta:** minimizar el tiempo entre que el usuario realiza una petición y el sistema responde.
- **Proporcionalidad:** Cumplir con expectativas de los usuarios. Por ejemplo, si un usuario le pone STOP a un reproductor de música, que la música deje de ser producida en un tiempo considerablemente corto.

Ejemplos de algoritmos:

- **Round Robin**

Se asigna a cada proceso un quantum para que utilice la CPU. El SO controla este tiempo a través de la interrupción por clock, es decir, si el proceso no termina en ese tiempo, se **interrumpe con una señal del clock** (interrupción por timer), se lo pone **al final de la cola**, y se ejecuta el siguiente.



La ventaja es que todos los procesos tienen su turno.

La desventaja es que si el quantum es muy corto, hay demasiadas interrupciones. Si es muy largo, la respuesta se vuelve lenta.

- **Prioridades**

Cada proceso tiene una prioridad asignada. Siempre se ejecuta el proceso con **mayor prioridad**.

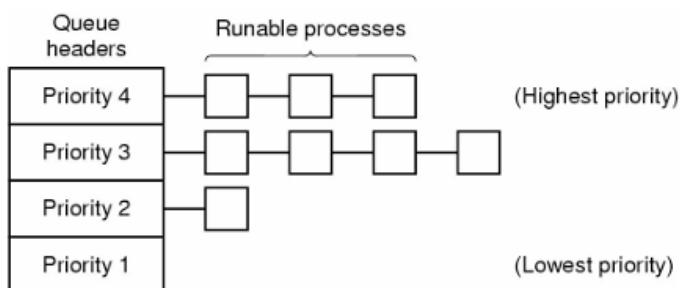
Permite priorizar procesos críticos.

Puede causar **inanición** de procesos con prioridad baja (quedan postergados indefinidamente o incluso no llegar a ejecutarse nunca).

- **Colas Multinivel**

Se empiezan a mezclar los algoritmos de planificación.

Se dividen los procesos en **varias colas** según su tipo o prioridad. Cada cola puede tener su **propio algoritmo** (por ejemplo, Round Robin en una, FCFS en otra).



Más flexibilidad. Puede combinar varios objetivos de planificación.

Requiere más configuración y control por parte del sistema.

- **SRTF - Shortest remaining time first**

Es la versión **apropiativa** de SJF. Siempre se ejecuta el proceso con **menor tiempo restante de ejecución**.

Si llega un proceso con tiempo restante de ejecución más corto mientras otro se ejecuta, el planificador **interrumpe** al actual y ejecuta el nuevo.

- ✖ Necesita estimar el tiempo restante (difícil de predecir). Se pierde tiempo de CPU en dicho cálculo.
- ✖ Puede causar inanición.

- Procesos en tiempo real

Estos procesos **deben cumplir con plazos estrictos**. No se trata solo de rapidez, sino de que ciertas tareas **se completen en un tiempo específico y garantizado**.

Controlan muy cuidadosamente cuándo y por qué se interrumpe un proceso.

Política vs mecanismo

Existen situaciones en las que es necesario que la planificación de uno o varios procesos se comporte de manera diferente, es decir, no siempre queremos que todos los procesos sean tratados de la misma manera por el SO.

El algoritmo de planificación debe estar parametrizado, debe permitir recibir parámetros, de manera que los procesos/usuarios pueden indicar los parámetros para modificar la planificación e indicarle cómo comportarse en cada caso.

¿Qué es un mecanismo? → Es la implementación general para gestionar la CPU. **Es implementado por el kernel**.

¿Qué es una política? → Es la decisión sobre qué hacer en cada situación, y puede variar según el contexto, el usuario, la configuración o el tipo de sistema. **El usuario/proceso/administrador utiliza los parámetros para determinar la política.**

Entonces, ¿por qué es importante separar ambos conceptos? porque da flexibilidad: el mismo kernel puede ser usado en contextos distintos cambiando solo los parámetros.

El kernel implementa el mecanismo pero el usuario/proceso/administrador utiliza los parámetros para determinar la política.

Ejemplo:

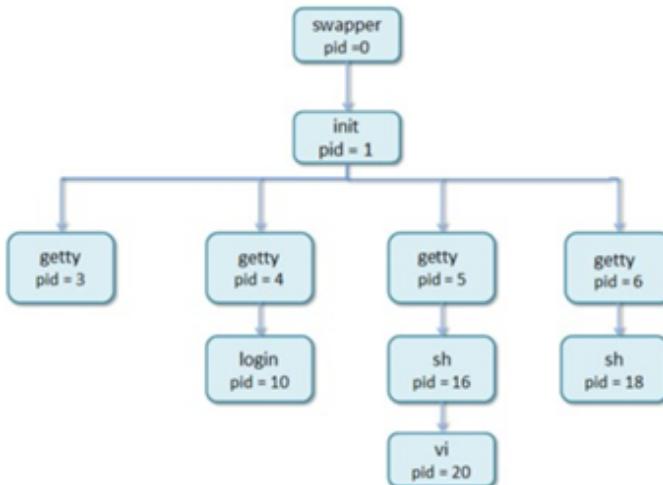
- Un algoritmo de planificación por prioridades y una System Call que permite modificar la prioridad de un proceso (el usuario puede usar la system call para cambiar la prioridad de un proceso con el comando “man nice”, cuanto más “nice” menos prioridad tiene)

- El mecanismo en la planificación por prioridades y el usuario puede modificar la **política de ejecución** de un proceso utilizando la system call `nice`, que cambia su prioridad.
- Un proceso puede determinar las prioridades de los procesos que el crea, según la importancia de los mismos
 - El **mecanismo** sigue siendo el mismo: planificación por prioridades. Pero la **política** ahora la define **el proceso padre**, asignándoles distintas prioridades según su importancia. Para ello puede usar system calls como `setpriority()`, sin modificar el mecanismo general del sistema operativo.

Creacion de procesos

Un proceso es creado por otro proceso. Todo proceso es creado por otro proceso que ya existe. Al proceso que crea se lo llama **padre**, y al que es creado, **hijo**.

Un proceso padre tiene uno o más procesos hijos. Se forma un arbol de procesos.



El proceso nº0 ya existe, no nace de ningún programa. El proceso raíz del árbol, por convención en Unix/Linux, es el **PID 1**, llamado `init`

¿Qué sucede al crear un proceso?

Cuando se crea un proceso:

- Crear la PCB (Process Control Block)
- Asignar PID (Process Identification) único

- Asignarle memoria: Se reserva espacio en memoria para stack, text, datos.
- Crear estructuras asociadas con fork(): Se hace una **copia exacta** del proceso padre. Esto incluye código, datos, contexto, etc.

En Unix: (2 System Calls)

- **System call fork()**

La llamada al sistema "fork()" crea un nuevo proceso hijo que es una copia del proceso padre.

Como el proceso hijo es una copia exacta del proceso padre: se copian tanto las regiones de memoria como los valores de los registros. Esto incluye el PC (Program counter), por lo que ambos procesos comienzan su ejecución en la misma instrucción.

Para distinguir cuál de los dos procesos está ejecutando el código tras el `fork()`, se utiliza el valor de retorno de la llamada `fork()`. Este valor se almacena en una variable, y permite identificar si el código se está ejecutando en el proceso padre o en el hijo.

¿Como funciona fork? (1)

```
Padre
//Instrucciones previas
PC → nue = fork()
if nue == 0
    // hijo
elsif nue > 0
    // padre
    // nue es el PID del hijo
else
    // error
end
```

¿Como funciona fork? (2)

Padre (el que invoca fork)	Hijo (el creado)
<pre><code>//Inst. previas nue = fork() if nue == 0 // hijo elsif nue > 0 // padre // nue es PID hijo else // error en el // llamado, el proceso // hijo nunca se creo end</code></pre>	<pre><code>//Inst. previas nue = fork() if nue == 0 // hijo elsif nue > 0 // padre // nue = PID hijo else // error end</code></pre>
PC = Program Counter	Fork retorna: 0 en el proceso hijo, valor mayor a cero en el padre (PID del hijo). valor negativo para error

Valor returnedo del fork	Significado
> 0 (positivo)	Estoy en el padre , y el número es el PID del hijo .
0	Estoy en el hijo .
< 0 (negativo)	Hubo un error y no se creó el proceso hijo.

- **System call execve()**

Después de crear un hijo con `fork()`, **el hijo suele querer ejecutar un programa diferente**. Para eso se usa la llamada `execve()`.

`execve()` borra el contenido actual del proceso y carga un nuevo programa

(recibido como parametro) en su lugar. Aunque el contenido cambia, el proceso sigue teniendo el mismo PID, es el mismo.

Ejemplo SysCall fork+execv

```
#  
# SHELL  
#  
import os, time  
print '''  
-----  
----- Esta es la ISO DIR/LS SHELL -----  
-----  
# exit (para salir)  
'''  
cmd = raw_input("iso:> ")  
while cmd >> 'exit':  
  
    if cmd == '':  
        cmd = raw_input("iso:> ")  
    else:  
        newpid = os.fork()  
        if newpid == 0:  
            # Sección del hijo  
            lista = cmd.split(' ')  
            os.execvp(lista[0], lista)  
            print "Imprimir AAAAAAAA"  
            exit(0)  
            print "Imprimir BBBBBBBBBB"  
        else:  
            # Sección del padre  
            #os.wait()  
            cmd = raw_input("iso:> ")  
  
exit(0)
```

¿Qué pasa con este código?

El código no se ejecuta nunca, luego de ejecutar execve() el resto del código se borra al cargar el nuevo programa pasado por parámetro.

En Windows: (1 System Call)

- **System call CreateProcess()**

La llamada al sistema “CreateProcess()” crea un nuevo proceso y carga el programa que se desea ejecutar.

A diferencia de Unix, donde se suele utilizar `fork()` seguido de `execve()` en dos pasos, en Windows todo se realiza en **una única llamada**: se crea el proceso y se carga directamente el programa especificado como parámetro.

Relación entre procesos padre e hijo

Con respecto a la ejecución:

- El proceso padre puede continuar ejecutándose **concurrentemente** con su(s) hijo(s).
- El padre puede esperar a que el proceso hijo (o los procesos hijos) terminen para continuar la ejecución (esta opción no suele utilizarse frecuentemente)

Con respecto al espacio de direcciones:

- En **Unix**, el proceso hijo es un **duplicado del proceso padre**.
 - Se crea un nuevo espacio de direcciones copiando el del padre.
- En **Windows**, al crear un proceso, se le **carga directamente el programa a ejecutar**
 - Se crea un nuevo espacio de direcciones vacío.

Terminación de procesos

Exit:

Cuando un proceso ejecuta exit, se retorna el control al sistema operativo.

- El padre no necesariamente espera a sus hijos. Si ejecuta `exit`, finaliza su ejecución aunque los procesos hijos aún estén en ejecución.
- Si se desea que el padre espere a sus hijos, puede utilizar la llamada `wait`, que permite recibir un **código de retorno** de los mismos.

Kill:

El proceso padre puede finalizar la ejecución de sus hijos usando la llamada `kill`

- Esto puede suceder si la tarea asignada al hijo ya se completó. También ocurre cuando el padre termina su ejecución; en algunos sistemas se permite que los hijos continúen, pero lo más habitual es que esto no esté permitido.
- Esta situación puede generar una **terminación en cascada**, donde la finalización de un proceso desencadena la terminación de varios.

Procesos cooperativos e independientes

- **Procesos Independientes** → Son aquellos que no afectan ni son afectados por la ejecución de otros procesos. No comparten datos ni estado.
- **Procesos cooperativos** → Son aquellos que afectan o son afectado por la ejecución de otros procesos. Son procesos que trabajan en conjunto con otros procesos para obtener una solución a un determinado problema, **interactuando o compartiendo información**.
 - *¿Para qué sirven los procesos cooperativos?*

Compartir información, por ejemplo, mediante archivos o memoria compartida.

Acelerar el cómputo, dividiendo una tarea en sub-tareas que se ejecutan en paralelo.

Planificar tareas para que puedan ejecutarse en paralelo de manera eficiente.

06- Administración de memoria

La **organización y administración** de la “memoria principal es uno de los factores más importantes en el diseño de los SO.

Para poder ser ejecutados y referenciados directamente, los **programas**, junto con sus **espacios de direcciones y datos**, deben estar ubicados en la memoria principal.

El **kernel** es el responsable de gestionar y organizar estos espacios de direcciones para permitir un alto grado de **multiprogramación**.

El SO debe cumplir con las siguientes funciones en relación con la memoria:

- **Llevar un registro** de las partes de memoria que se están utilizando y de aquellas que no.
- **Asignar espacio** en memoria principal a los procesos cuando estos la necesitan.
- **Liberar espacio** de memoria asignada a procesos que han terminado.
- **Abstraer al programador** de la alocación de los programas. El programador no necesita preocuparse por el lugar físico exacto de la memoria donde se carga su programa.
 - Desde su punto de vista, el programa ocupa un espacio de direcciones continuo y exclusivo, como si tuviera toda la memoria para sí. Esta visión es **una abstracción lógica** provista por el sistema operativo, que internamente traduce estas direcciones lógicas a direcciones físicas reales.
- Los procesos se abstraen de su ubicación física. Para los programadores los procesos pueden ocupar todas las direcciones posibles pero la realidad que en lo físico no sucede, es una cuestión lógica.

- **Brindar seguridad entre los procesos** para que unos no accedan a secciones privadas de otros.
- **Brindar la posibilidad de acceso compartido** a determinadas secciones de la memoria (librerías, código en común, etc.).
 - Por ejemplo, si dos procesos usan la misma librería, no es eficiente cargarla dos veces: el sistema puede cargarla una sola vez y permitir que ambos procesos accedan a ella.
- **Garantizar la performance del sistema**, haciendo un uso óptimo de la memoria disponible.

Se espera que el SO **utilice la memoria de forma eficiente**, permitiendo alojar la mayor cantidad posible de procesos sin comprometer la seguridad, etc.

División Lógica de la Memoria Física para alojar múltiples procesos

Para que múltiples procesos puedan coexistir en la memoria principal sin interferirse entre sí, es necesario establecer una **división lógica** de la memoria física.

Esta división lógica **depende del mecanismo provisto por el HW**.

- El SO usa los mecanismos que ofrece el hardware (como segmentación, paginación, MMU) para implementar esta división lógica. El SO no puede inventar cómo administrar la memoria; debe ajustarse a las capacidades del hardware.

Un objetivo es la **asignación eficiente**

- Contener el mayor número de procesos en memoria. Cuántos más procesos haya en memoria, mayor será la utilización de la CPU, ya que siempre habrá algo para ejecutar.

Esta división debe cumplir con ciertos requisitos:

- **Reubicación**

El programador **no necesita saber** en qué parte exacta de la memoria RAM será cargado su programa.

Mientras un proceso se ejecuta, puede ser sacado y traído a la memoria (swap) y, posiblemente, colocarse en diferentes direcciones.

Para lograr esto, todas las **referencias a memoria** que hace el proceso deben

traducirse a direcciones físicas correctas, según su ubicación actual.

Esto permite que cada proceso sea **independiente de su posición física** en la memoria.

- **Protección**

Se debe garantizar que un proceso **no pueda acceder/referenciar a las direcciones de memoria de otro proceso**, salvo que tenga un permiso explícito.

El chequeo se debe realizar durante la ejecución: el SO, con ayuda del hardware, **verifica cada acceso a memoria que quiera realizar el proceso durante su ejecución**, asegurándose de que no sobrepase sus límites asignados.

NO es posible anticipar/predecir todas las referencias a memoria que un proceso puede realizar de forma estática, ya que estas pueden depender de decisiones que el programa toma dinámicamente durante su ejecución.

- **Compartición**

Permitir que varios **procesos accedan a la misma porción de memoria**, como por ejemplo: Rutinas comunes, librerías, espacios explícitamente compartidos, etc.

Esto **optimiza el uso de la memoria RAM**, evitando copias innecesarias (repetidas) de instrucciones.

Abstracción- Espacio de direcciones

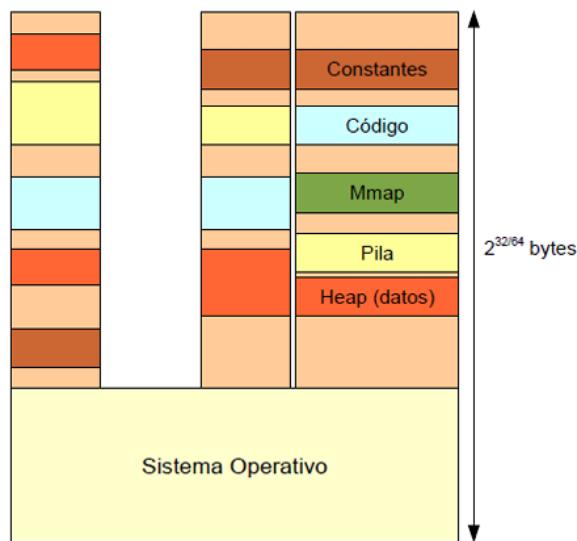
El espacio de direcciones es el rango de direcciones posibles que un proceso puede utilizar para acceder a sus instrucciones y datos.

Se dice que es una **abstracción** que permite al programador trabajar sin preocuparse por la ubicación real de los datos en la memoria física (RAM) ya que cuando un proceso se ejecuta, no ve ni accede a direcciones físicas reales, ve un espacio de direcciones lógico, que empieza desde 0 y llega hasta cierto límite. El proceso cree que está solo y que tiene toda la memoria para sí mismo, cuando en realidad está compartiendo la RAM con muchos otros procesos.

El tamaño del espacio de direcciones depende de la Arquitectura del Procesador

- Si la arquitectura es de 32 bits, se puede direccionar de **0 a $2^{32} - 1$**

- Si la arquitectura es de 64 bits, se puede direccionar de $0 \text{ a } 2^{64} - 1$



Cada proceso tiene su propio espacio de direcciones, **independiente del resto y de su ubicación real** en la memoria RAM.

Además del código, los datos, la pila, etc. Incluso el Sistema Operativo (SO) está presente en cada espacio de direcciones como parte del entorno de ejecución del proceso.

Tipos de Direcciones

Direcciones Logicas

Hacen referencia a ubicaciones/direcciones en el espacio de direcciones del proceso, sin importar dónde estén físicamente esos datos en la memoria RAM.

Direcciones físicas

Hacen referencia a ubicaciones/direcciones reales en la memoria física (RAM). Son las **direcciones absolutas**

Direcciones lógicas vs direcciones físicas

→ Los procesos no acceden directamente a direcciones físicas de memoria, utilizan direcciones lógicas (también llamadas virtuales) que hacen referencia a las direcciones dentro de su espacio de direcciones. Sin embargo, como la memoria principal utiliza direcciones físicas, es necesario traducirlas a **direcciones físicas** antes de acceder a la RAM.

Conversión de direcciones

Para realizar la conversión de direcciones **se utilizan registros auxiliares que se fijan cuando el proceso** (su espacio de direcciones) **es cargado en memoria**

- **Registro base**

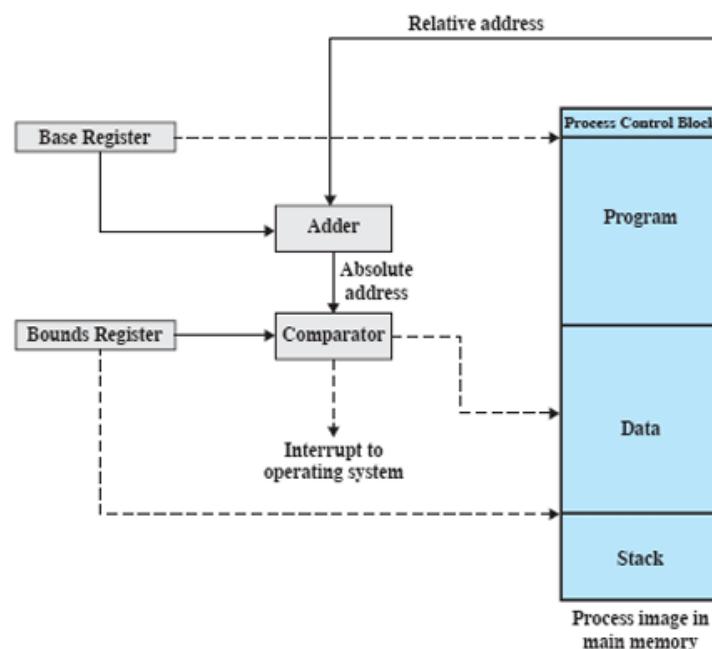
Contiene la dirección de comienzo del espacio de direcciones del proceso en la RAM

- **Registro límite**

Contiene la dirección final del proceso o medida del proceso (tamaño de su espacio de direcciones)

¿Como funciona?

1. A cada dirección lógica generada por el proceso se le suma el contenido del **registro base**, obteniendo la dirección física.
2. Luego, se verifica que la dirección lógica no exceda el **registro límite**.
* Si se supera el límite, se lanza una **interrupción de hardware**, lo que indica un intento de acceso inválido a memoria.



Ejemplo: Supongamos que el proceso tiene un espacio de direcciones lógicas de 0 a 50, y se lo carga en memoria física a partir de la dirección 100.

- La dirección lógica 45 se convierte en la dirección física 145 ($100 + 45$).

- Si el proceso intenta acceder a la dirección lógica 60, se detecta que esta supera el límite (50) y se genera una interrupción.

Esta traducción se denomina resolución de direcciones (address-binding)

La resolución de direcciones se puede realizar en distintos momentos:

- **En tiempo de compilación o en tiempo de carga**
 - Las direcciones lógicas y físicas son idénticas.
 - Como el código queda “pegado” a una posición fija en RAM, si se desea reubicar un proceso en otra parte de la memoria, es necesario recompilarlo o recargarlo
- **En tiempo de ejecución**
 - Las direcciones lógicas (también llamadas virtuales) y físicas son diferentes. La dirección física real en RAM se determina en el momento en que se accede a memoria.
 - Cuando el programa se está ejecutando, ante cada instrucción, se hace una resolucion de direcciones y se accede a la RAM.
 - La reubicación se puede realizar fácilmente ya que no hay direcciones físicas fijas, si no que traducciones dinámicas.
 - El mapeo entre “Virtuales” y “Físicas” es realizado por hardware (MMU). Es la CPU con ayuda de la MMU quein traduce las direcciones en tiempo real, sin intervención del SO.
 - **La resolución de direcciones debe acompañar a la velocidad del procesador.** Como el CPU es muy rápido, la traducción también debe serlo para no volverse un cuello de botella. **Por eso la MMU está integrada con los procesadores (CPU).**

Memory Management Unit (MMU)

Dispositivo de Hardware que **traduce direcciones lógicas (virtuales) a físicas**.

- Está **integrada al Procesador**
- Cada vez que hay un cambio de contexto (context switch), **se le dan los datos** del nuevo proceso (el registro base y límite) a la MMU para que pueda hacer correctamente la traducción de direcciones cada vez que el proceso intente acceder a memoria.

- Re-programar el MMU es una **operación privilegiada**, solo puede ser realizada en Modo Kernel.
- El valor del **registro de reallocación** se **suma** a cada dirección lógica generada por el proceso (al momento de acceder a la memoria) para obtener la dirección física.



Mecanismos de asignación de memoria

La asignación de memoria es la forma en la que el sistema operativo administra y partitiona la memoria principal (RAM) para alojar los procesos que se están ejecutando. La idea es decidir dónde colocar cada proceso dentro de la memoria.

Existen dos formas →

Particiones Fijas

Es el primer esquema implementado.

- La memoria RAM se divide en particiones o regiones de tamaño fijo
 - Pueden ser todas del mismo tamaño (uniforme) o de diferentes tamaños.
- Cada partición aloja un único proceso (su espacio de direcciones)
- Cuando llega un nuevo proceso, el SO lo coloca en alguna partición de acuerdo a algún criterio (First Fit, Best Fit, Worst Fit, Next Fit)

Particiones dinámicas

Es la evolución del esquema anterior.

- No existe una predivisión de la memoria principal (RAM).

- A medida que los procesos llegan, se crean particiones dinámicamente dentro de la memoria principal, con exactamente el tamaño que el proceso necesita.
- Cada partición aloja un único proceso (su espacio de direcciones)

¿Qué problemas se generan en cada caso? → La **necesidad de continuidad** en la memoria RAM genera un problema en ambos esquemas.

Fragmentación

La fragmentación se produce cuando una localidad de memoria no puede ser utilizada por no encontrarse en forma contigua.

Fragmentación interna

Se produce en el esquema de **particiones fijas**.

Sucede cuando **la partición es más grande que el proceso**, y la diferencia se desperdicia. Es la porción de la partición que queda sin utilizar.

No tiene solución.

Fragmentación externa

Se produce en el esquema de **particiones dinámicas**.

Son los **huecos libres** que van quedando en la memoria a medida que los procesos finalizan.

Si bien hay espacio libre, no es contiguo, y por eso puede no ser utilizable.

Para solucionar el problema, se acude a la **compactación**, pero es muy costosa.

Problemas del esquema

El esquema de Registro Base + Límite presenta problemas:

- Necesidad de almacenar el Espacio de Direcciones de forma continua en la Memoria Física
- Los primeros SO definían particiones fijas de memoria, luego evolucionaron a particiones dinámicas
- Fragmentación
- Mantener “partes” del proceso que no son necesarias

- Los esquemas de particiones fijas y dinámicas no se usan hoy en día

Solución → Nuevas técnicas de administración de la memoria

Paginación

Es una técnica de administración de memoria que busca resolver los problemas de fragmentación externa y aprovechar mejor el espacio libre en RAM. Busca **romper con la necesidad de continuidad** en la memoria principal.

La memoria física (RAM) es dividida lógicamente en pequeños trozos de igual tamaño llamados "**Marcos**".

La memoria lógica (el espacio de direcciones de un proceso) es dividida en trozos de igual tamaño que los marcos llamados "**Páginas**"



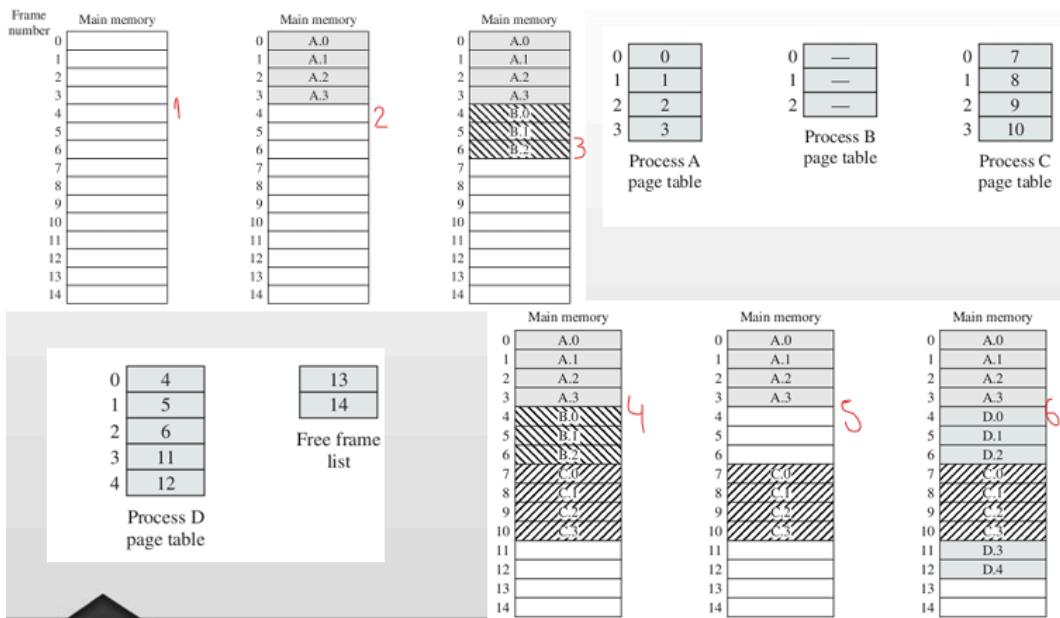
¿Qué logra esta división? No es necesario que todo el proceso esté cargado en bloques contiguos de memoria. **Cada página puede ser cargada en cualquier marco** disponible de la RAM.

Tabla de páginas

El SO debe mantener una tabla de páginas **por cada proceso para mantener el registro de qué página fue a qué marco**.

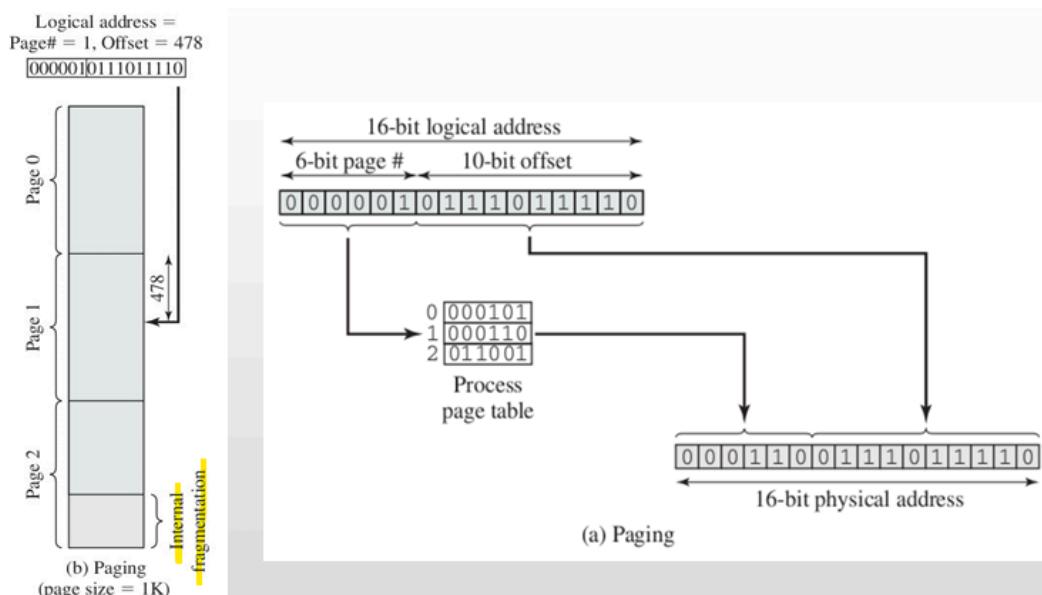
¿Qué contiene la tabla de páginas?

- Tiene una entrada por cada página del proceso, es decir, tantas entradas como páginas tenga el proceso.
- Cada entrada indica en qué marco de la RAM se encuentra esa página.



Por ejemplo, los procesos pueden ir liberandose de la memoria dejando marcos libres, que luego podrán ocupar otros procesos que lleguen. No es necesaria la continuidad.

La tabla de página **le sirve a la MMU** para traducir las direcciones logicas a físicas.



¿Como funciona?

La dirección logica se divide en dos campos

- Numero de página (representa la página dentro del proceso)

- Desplazamiento (representa la dirección dentro de la página).

¿Cómo traduce la MMU?

1. Usa el número de página para buscar en la tabla qué marco le corresponde.
2. Cambia el número de página por el número de marco.
3. El desplazamiento permanece igual (ya que las páginas y los marcos tienen el mismo tamaño).
4. Como resultado se obtiene la dirección física.

Entonces, basicamente, la MMU interpreta la dirección lógica, cambia los bits que representan a la página por los bits que representan al marco que se encuentran en la tabla de página (las páginas y los marcos tienen el mismo tamaño) para obtener la dirección física.

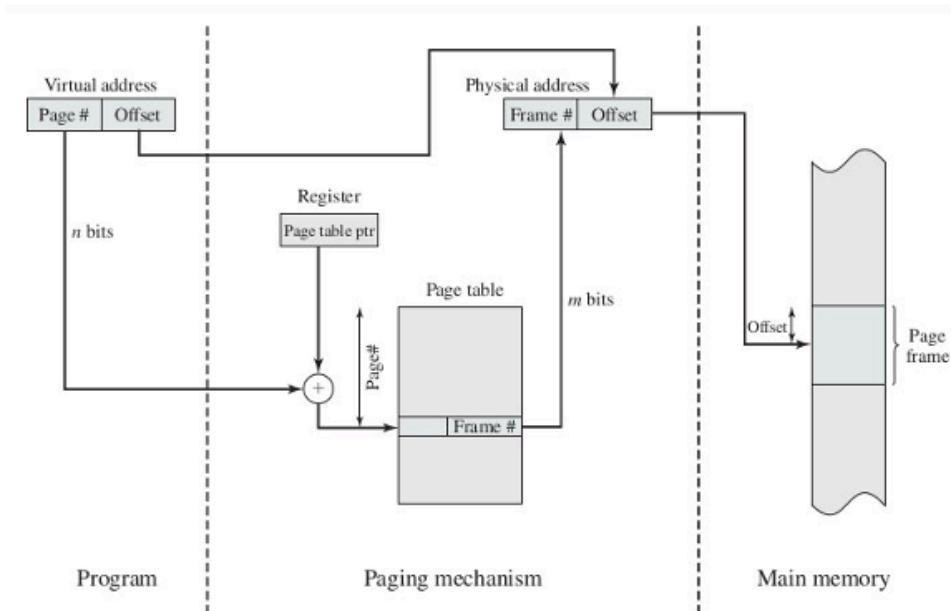


Figure 8.3 Address Translation in a Paging System

✓ Ventajas

- Elimina la fragmentación externa
- Permite usar la memoria más eficientemente.

⚠ Consideraciones

- La tabla de páginas debe mantenerse actualizada
- Puede ocurrir fragmentación interna en la última página.

Segmentación

Es una técnica de administración de memoria que organiza el espacio de direcciones de un proceso de manera lógica, tal como un programador la estructura o piensa, no como una secuencia de direcciones de memoria desde 0 hasta n, si no como partes diferenciadas con roles distintos (un segmento de código, de datos, etc.). Por este motivo, se dice que se asemeja a la "visión del usuario".

Cada proceso se divide en partes/secciones como: código, stack, datos, etc.

La segmentación piensa a cada una de estas partes como **segmentos**, cada segmento contiene datos de un mismo tipo.

Cada segmento **se carga en memoria RAM**. No necesitan estar ubicados de forma continua, solo tienen que estar en algún lugar donde entren.

Cabe aclarar que dentro del segmento si hay continuidad, es decir, los datos dentro de un segmento si están organizados uno tras otro.

¿Tienen todos el mismo tamaño? No. **Cada segmento tiene un tamaño propio**, dependiendo de lo que contiene.

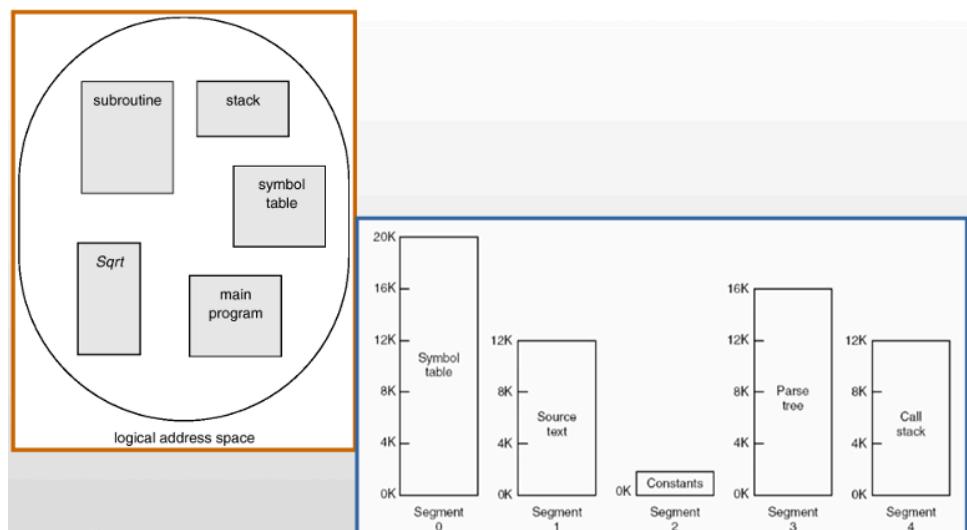


Tabla de segmentos

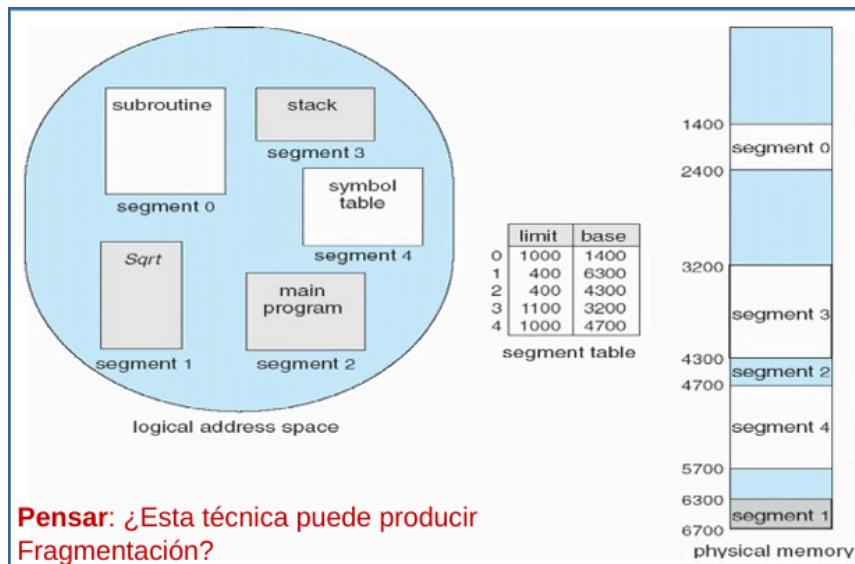
El SO debe mantener una tabla de segmentos **por cada proceso**.

¿Qué contiene la tabla de segmentos?

- Tiene **una entrada por cada segmento** del proceso, es decir, tantas entradas como segmentos tenga el proceso.
- Cada entrada contiene:

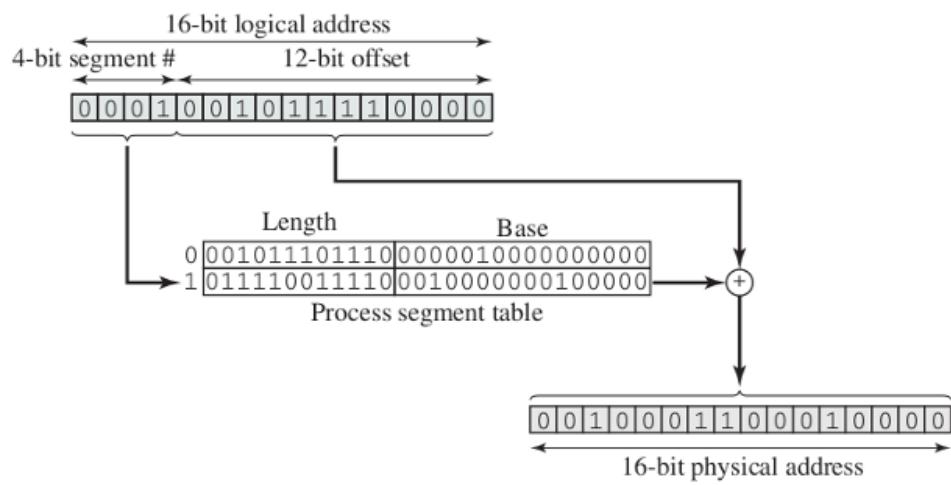
- Dirección base: dirección donde se cargó el segmento en la RAM.
- Dirección límite: tamaño del segmento (sirve para saber hasta dónde puede accederse).

Como los segmentos tienen distinto tamaño, cuando se traducen las direcciones lógicas a físicas, la dirección límite ayuda a controlar que una dirección no exceda el límite del segmento que le corresponde.



Si, puede producir **fragmentación externa**. Como cada segmento tiene tamaño variable, al ir cargándolos donde entran pueden quedar huecos que luego no se reutilicen. Esto reduce la utilización de esta técnica.

La tabla de página **le sirve a la MMU** para traducir las direcciones lógicas a físicas.



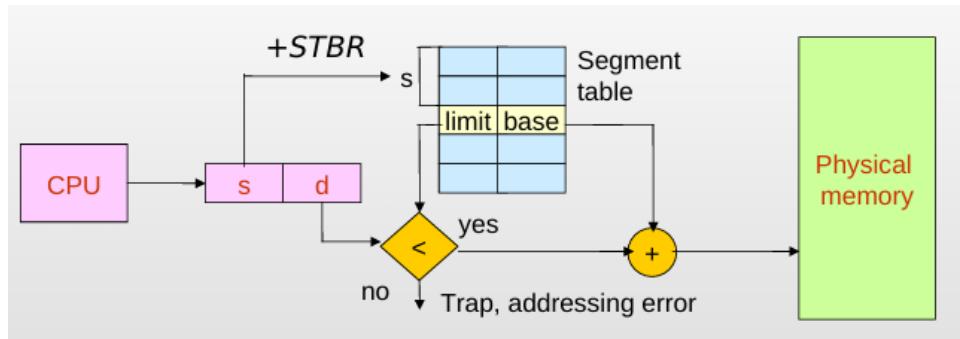
La dirección lógica que utiliza un proceso se divide en dos campos:

- **Selector de segmento**: Indica qué segmento se debe utilizar, es decir, cuál de los segmentos definidos en la **tabla de segmentos** es el que interesa.
- **Desplazamiento**: Es el valor que indica la posición exacta dentro del segmento seleccionado.

Para traducir una dirección lógica, se utilizan registros como:

STBR (Segment-Table Base Register) → Apunta a la dirección en memoria donde comienza la tabla de segmentos del proceso actual.

STLR (Segment-Table Length Register) → Indica cuántos segmentos tiene el proceso (es decir, el tamaño de la tabla de segmentos). Sirve para verificar que el selector de segmento sea válido.



¿Cómo funciona?

- Se verifica que el selector de segmento esté dentro del límite permitido, comparándolo con el valor del STLR.
 - Si es mayor que el STLR → es un acceso inválido.
- Se accede a la **tabla de segmentos**, usando el **selector** como índice. En esa entrada se encuentra la base y el límite.
- Se verifica que el desplazamiento sea menor al límite del segmento.
 - Si el desplazamiento es mayor al límite, la dirección lógica no es válida → se lanza una excepción
- Se suma el desplazamiento a la base del segmento, y se obtiene así la dirección física final.

✓ Ventajas

- Es visible al programador

- Facilita modularidad, estructuras de datos grandes y da mejor soporte a la compartición y protección

Ventajas de la segmentación sobre la paginación

Una de las principales ventajas de la segmentación frente a la paginación es la **facilidad para compartir y proteger segmentos de memoria**.

- **Compartir**

En la segmentación, si varios procesos deben compartir una misma parte de código o datos (por ejemplo, una biblioteca), basta con que todos apunten al **mismo segmento** en la tabla de segmentos. Como ese segmento tiene **una única entrada** con una dirección base compartida, todos acceden a la misma región de memoria física sin necesidad de duplicarla.

- **Proteger**

Al estar la información segmentada de forma lógica (por funciones o tipos de datos), es más fácil aplicar políticas de protección. Si se desea proteger un segmento (por ejemplo, hacerlo solo de lectura), basta con modificar **una única entrada** en la tabla de segmentos.

En cambio, en la paginación, como un segmento lógico se divide en múltiples páginas, habría que **modificar varias entradas** en la tabla de páginas para aplicar la misma política de protección.

Segmentación Paginada

Es una **combinación de segmentación y paginación**, utilizada por arquitecturas como **Intel x86** para aprovechar las ventajas de ambas técnicas.

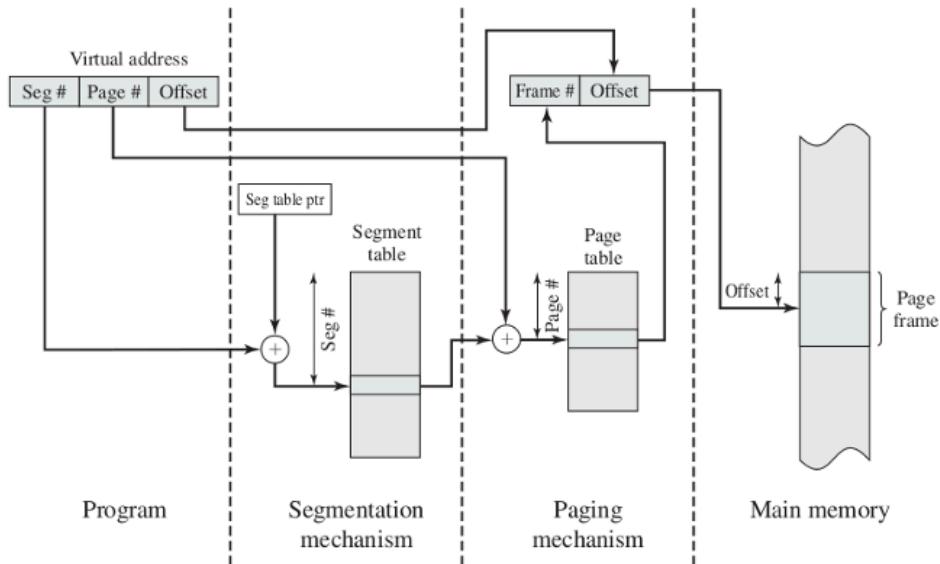


Figure 8.13 Address Translation in a Segmentation/Paging System

¿Cómo funciona?

Una **dirección lógica** (Logical Address) generada por un programa pasa por **dos niveles de resolución**:

1. Segmentación:

Se utiliza el **selector de segmento** y el **desplazamiento** para acceder a la **tabla de segmentos y obtener una dirección**. Intel la llama dirección lineal (Linear Address).

En esta etapa el sistema operativo puede aplicar mecanismos de **protección** y **compartición**.

2. Paginación:

La dirección se divide en **número de página + desplazamiento**. Se accede a la **tabla de páginas** para obtener la **dirección física (Physical Address)**.

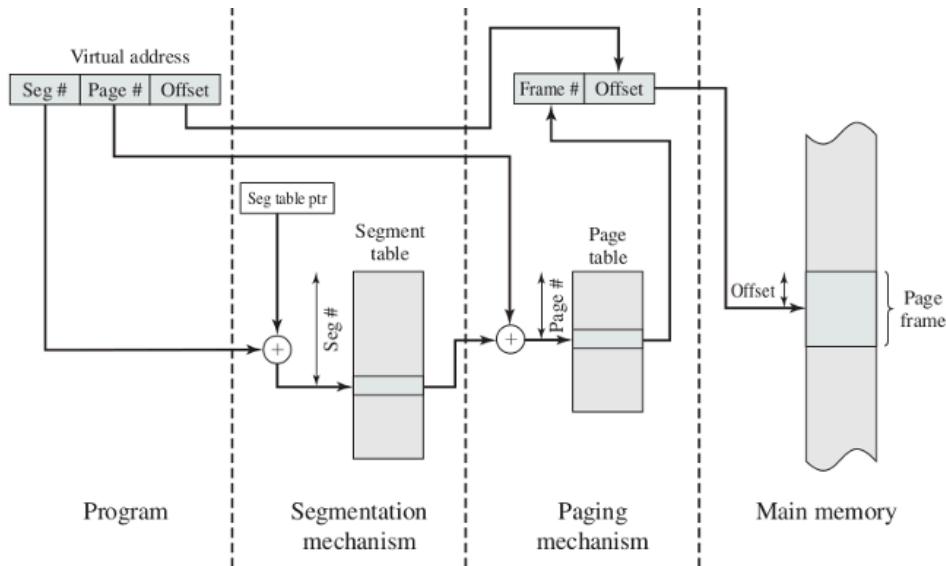
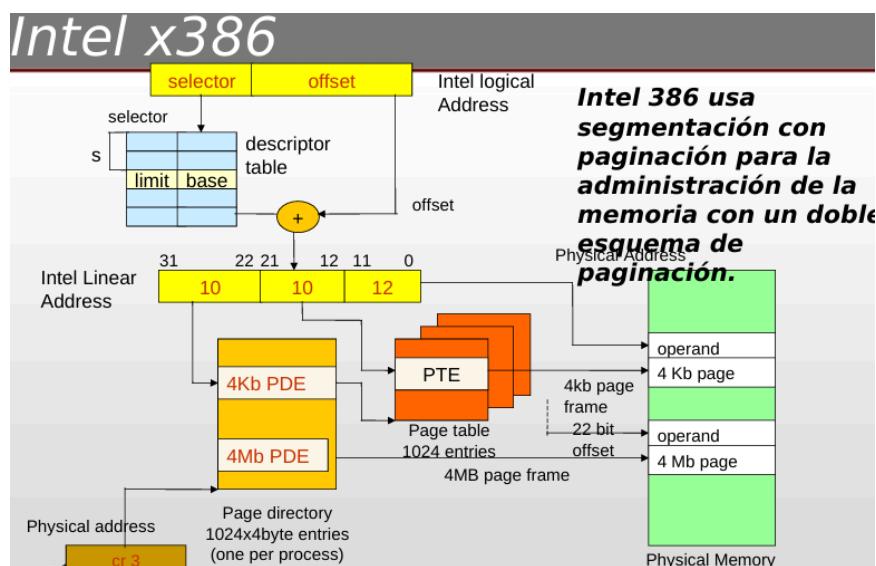


Figure 8.13 Address Translation in a Segmentation/Paging System

Cada segmento es dividido en páginas de tamaño fijo.



08-Memoria 2

Memoria virtual

Es una técnica que **permite que NO esté toda la imagen de un proceso en memoria principal** todo el tiempo.

¿Por qué es útil? Porque en la práctica suele suceder que

- Hay bibliotecas o rutinas que se ejecutan una única vez (o nunca)

- Algunas **partes del código** se ejecutan solo una vez y **no se vuelven a usar**.
- Regiones de memoria alocadas dinámicamente y luego liberadas, etc.

¿Qué propone la memoria virtual?

La memoria virtual propone que el SO pueda cargar a memoria solo las piezas/partes de un proceso a medida que las necesita.

Conjunto residente (o working set):

Se llama "**conjunto redidente**" (alguna bibliografía lo llama "Working set") a la **porción del espacio de direcciones del proceso que se encuentra en memoria principal (RAM)**.

¿Qué se necesita para que funcione la memoria virtual?

- **Soporte de hardware** que permita:
 - Detectar cuando se necesita una porción del proceso que no está en su conjunto de residente
 - Generar una **fallo de página (page fault)** cuando eso sucede.
 - Cargar en memoria dicha porción (página o segmento) desde el almacenamiento secundario para continuar con la ejecución.
- El hardware debe **soportar paginación por demanda (y/o segmentación por demanda)**, es decir, poder traer solo la página o el segmento necesario, y no todo el proceso.
- Un **dispositivo de almacenamiento secundario** (como un disco) para almacenar las secciones del proceso que no están en memoria principal.
- El **SO debe ser capaz de mover las páginas** (y/o segmentos) entre la memoria principal y la secundaria. De esta manera, debe llevar un registro de qué partes del proceso están en RAM y cuáles no.

Ventajas

- **Mayor concurrencia:** Más procesos pueden ser mantenidos en memoria ya que solo son cargadas algunas secciones de cada proceso
- **Más procesos listos:** Con más procesos en memoria principal es más probable que existan más procesos Ready

- **Procesos más grandes que la RAM:** Un proceso puede tener un tamaño mayor al de la memoria principal. El programador **no necesita preocuparse** por eso; el límite lo impone el hardware (principalmente el tamaño del bus de direcciones).

Memoria virtual con paginación

Cuando se usa memoria virtual con paginación, **cada proceso tiene su tabla de páginas** donde cada entrada referencia al frame o marco en el que se encuentra la página en la memoria principal.

Como ahora **no todas las páginas están siempre en memoria**, se necesita agregar más información a cada entrada de la tabla de páginas: **bits de control** para saber si una página está presente o no, y si fue modificada o no.

📌 Bit V (bit de validez)

- Indica si la página está en memoria principal (RAM)
 - Si el bit V es **1** → la página **está en RAM** y puede accederse normalmente.
 - Si el bit V es **0** → la página **no está en memoria**: se produce una **falla de página** (page fault).
- Lo activa/desactiva el SO, lo consulta el HW.
 - El SO es el encargado de subir o bajar páginas, el HW (la MMU) lo utiliza para saber si puede traducir o no la dirección.

📌 Bit M(bit de modificación)

- Indica si la página fue modificada mientras estuvo en memoria.
 - Si **M = 1**, entonces, en algún momento, hay que volver a **escribirla en el almacenamiento secundario (disco)** porque tiene cambios que no están guardados.
 - Si **M = 0**, no fue modificada → **se puede descartar** sin guardar.
- Lo activa/desactiva el HW, lo consulta el SO.
 - El HW es el que sabe si se realiza o no una escritura en una página, el SO lo utiliza al bajar una página a almacenamiento secundario (disco).

Entrada en la Tabla de páginas de x86 (32 bits)

El HW define el formato de la tabla de páginas y el SO se adapta a él

Una entrada válida tiene:

- ✓ Bit V = 1
- ✓ Page Frame Number (PFN) – Marco de memoria asociado
- ✓ Flags que describen su estado y protección



Fallo de página (Page Fault)

Un fallo de página ocurre cuando:

- El proceso quiere acceder a una página que no está cargada en la memoria principal (RAM).
- La MMU intenta traducir una dirección logica y al revisar la tabla de páginas encuentra el bit de validez en 0 (V=0), lo que significa que la página está en disco, no en RAM.

El HW lanza un trap o interrupción por software al SO para notificarle que necesita traer esa página desde el disco.

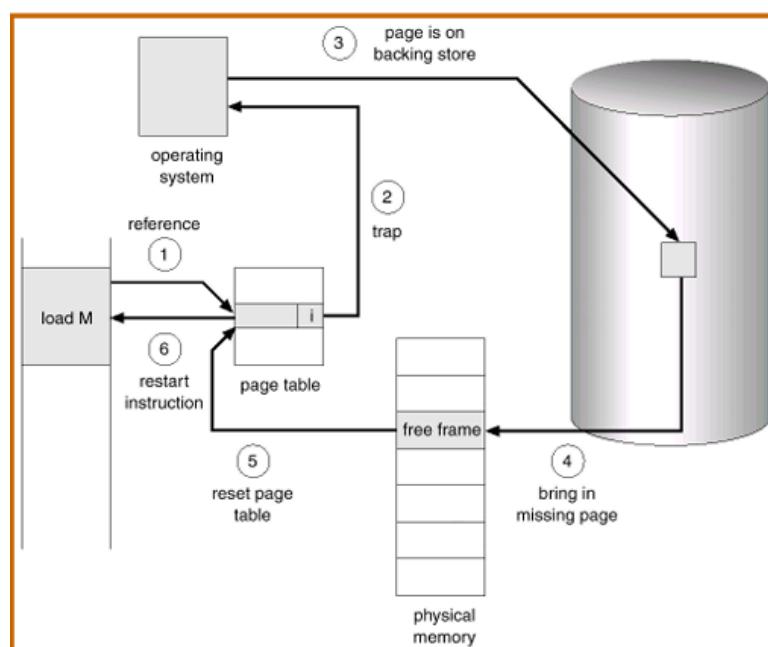
→ Una **buenas prácticas** en esta situación es que el SO coloque al proceso en estado de "blocked" (espera) mientras gestiona que la página que se necesite se cargue.

¿Qué hace el SO ante un fallo de página?

- El SO debe buscar un "Frame o Marco libre" en la memoria principal.
 - Podría ocurrir que no haya marcos disponibles, con lo cual habrá que descargar uno para lograr espacio para la nueva página entrante.
- El SO debe generar una operación de E/S al disco para copiar en dicho Frame la página del proceso que se necesita utilizar.
 - Mientras se completa la operación de E/S, el SO puede asignarle la CPU a otro proceso. La operación E/S se realizará y avisará mediante interrupción su finalización.

- Cuando la operación de E/S finaliza, se notifica al SO y este:
 - Actualiza la tabla de páginas del proceso
 - Coloca el Bit V en 1 en la página en cuestión
 - Coloca la dirección base del Frame donde se colocó la página
 - El proceso que generó el fallo de página vuelve al estado de ready (listo)

→ Cuando el planificador le dé CPU al proceso otra vez, se volverá a ejecutar la instrucción que antes generó el fallo de página



Performance

Si ocurren muchos fallos de página, la performance del sistema decae.

Tasa de fallos de página (probabilidad de que ocurra un fallo de página) → 0 ≤ p ≥ 1

- Si $p=0$, no hay page faults
- Si $p=1$, todo acceso a memoria principal genera un fallo de página.

Observación: p jamás es 0 ya que es una técnica basada en los fallos de página.

Efective Access Time (EAT)

Esta fórmula refleja como la tasa de fallos (p) impacta en el tiempo que realmente lleva acceder a memoria

$$EAT = (1-p) * \text{memory access} + p * (\text{page_fault_overhead} + [\text{swap_page_out}] + \text{swap_page_in} + \text{restart_overhead})$$

Se calcula considerando tanto el tiempo de acceso a memoria cuando no hay fallo como el tiempo de gestión cuando ocurre un fallo de página.

Tabla de Páginas

Cada proceso tiene su tabla de páginas que permite traducir direcciones virtuales a direcciones físicas.

El **tamaño de la tabla de páginas** depende del espacio de direcciones del proceso y de el tamaño de cada página.

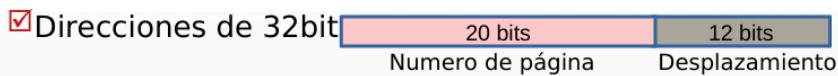
 **Problema** → Cuando el espacio de direcciones es grande, como en sistemas de 64 bits, **la tabla puede alcanzar un tamaño considerable (muy grande)**.

Como las tablas deben almacenarse en memoria principal, aparecen nuevas **formas de organizarlas**: La forma de organizarla depende del HW subyacente (hay SO que solo soportan tablas de 1 nivel, otros usan tablas invertidas, etc.)

Tabla de 1 nivel: Tabla única lineal

Es una **estructura lineal** que almacena una entrada por **cada página lógica posible** del proceso.

Tabla de 1 nivel - 32 bits



Ejemplo

- ✓ Cantidad de Page Table Entries (PTEs) máximas que puede tener un proceso = 2^{20} (1.048.576)
- ✓ El tamaño de cada página es de 4KB (2^{12})
- ✓ El tamaño de cada PTE es de 4 bytes
 - ✓ Cantidad de PTEs que entran en un marco: $4KB/4B = 2^{10}$

Tamaño de tabla de páginas

- Cantidad de marcos necesarios para todas las PTEs de la tabla de páginas de un proceso = $2^{20}/2^{10} = 2^{10}$
- Tamaño tabla de páginas del proceso: $2^{10} * 4\text{bytes} = 4\text{MB por proceso}$

Páginas de 4 KB → 12 bits de desplazamiento
 Espacio de direcciones de 32 bits → 20 bits para el número de página
 $\rightarrow 2^{20} = 1$ millón de entradas por tabla por proceso
 Si cada entrada ocupa 4 bytes, tenemos: $1 \text{ millón} \times 4 \text{ B} = 4 \text{ MB}$ por proceso solo para la tabla**
➡ Esto no escala bien.

Tabla de 1 nivel - 64 bits

<input checked="" type="checkbox"/> Direcciones de 64bits	52 bits	12 bits
	Numero de página	Desplazamiento
<input checked="" type="checkbox"/> Ejemplo		
<ul style="list-style-type: none"> ✓ Cantidad de Page Table Entries (PTEs) máximas que puede tener un proceso = 2^{52} ✓ El tamaño de cada página es de 4KB ✓ El tamaño de cada PTE es de 4 bytes <ul style="list-style-type: none"> • Cantidad de PTEs que entran en un marco: $4\text{KB}/4\text{B} = 2^{10}$ ✓ Tamaño de tabla de páginas <ul style="list-style-type: none"> • Cantidad de marcos necesarios para todas las PTEs de la tabla de páginas de un proceso = $2^{52}/2^{10} = 2^{42}$ • Tamaño tabla de páginas del proceso = $2^{42} * 4\text{bytes} = 2^{54}$ 		
Más de 16.000GB por proceso!!!		

→ Son fáciles de implementar y hay acceso muy rápido (1 acceso) a memoria pero **consumen una cantidad masiva de memoria** cuando el espacio de direcciones es grande.

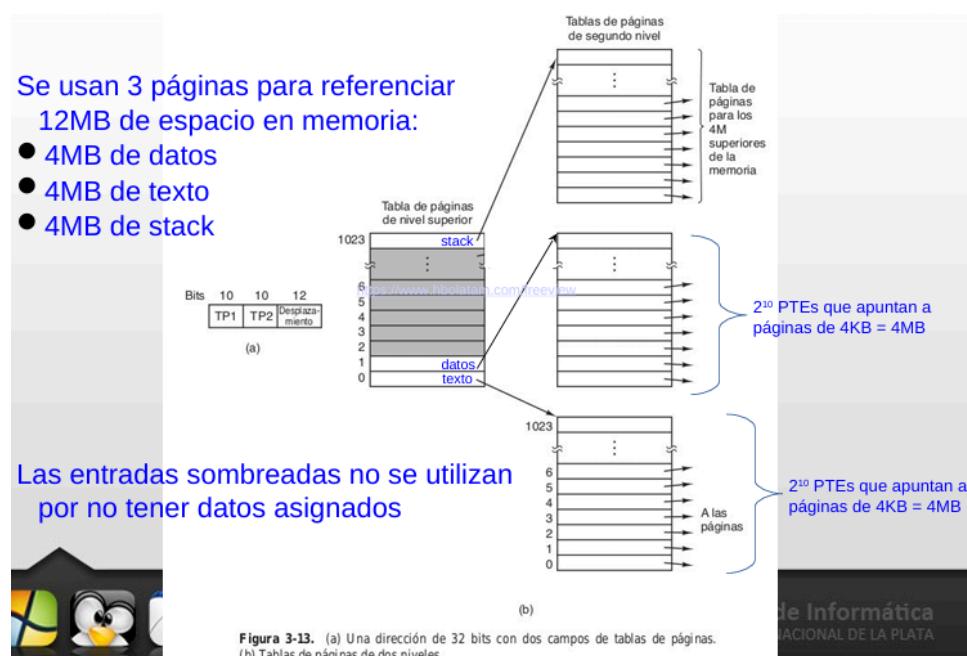
Tabla de 2 niveles (o más, multinivel)

El propósito de la tabla de páginas multinivel es **dividir la tabla de páginas lineal en múltiples tablas de páginas**.

- Cada tabla de páginas suele tener el mismo tamaño pero se busca que tengan **un menor número de páginas por tabla**
- La idea general es que cada tabla sea más pequeña
- Se busca que **la tabla de páginas no ocupe demasiada memoria RAM**
- Solo **se carga una parcialidad de la tabla de páginas** (solo lo que se necesite resolver y está en uso)
 - No es necesario **cargar toda la tabla de páginas** completa en memoria principal.

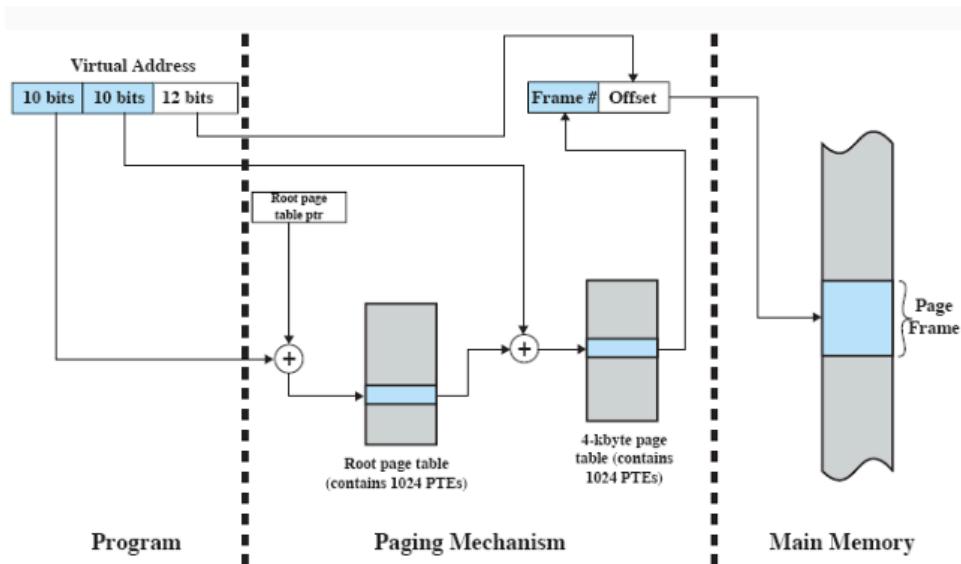
- Existe un **esquema de direccionamientos indirectos**

- En lugar de acceder directamente a una sola tabla plana, la dirección virtual se divide en varias partes donde cada parte se usa para indexar un nivel distinto de la tabla y así llegar indirectamente a la entrada final que contiene el marco/frame.



Se evita tener una tabla completa, en su lugar, por ejemplo, con 2 niveles:

- Tabla de primer nivel → Contiene punteros (direcciones) base a la tabla de segundo nivel
- Tabla de segundo nivel → Contiene los marcos reales (frames) para luego realizar la suma con desplazamiento y llegar a la dirección física.



La **MMU** interpreta la dirección virtual de otra manera.

- **Primeros 10 bits:** Índice para acceder a la **tabla de primer nivel** → esta entrada contiene la dirección base de una **tabla de segundo nivel**.
- **Siguientes 10 bits:** Índice para acceder a una entrada específica en la **tabla de segundo nivel**, donde está el **número de marco físico (frame)**.
- **Últimos 12 bits:** Desplazamiento dentro del marco.

Ventajas:

- **Menor uso de RAM:** Solo se crean las tablas de segundo nivel necesarias. Si el proceso no usa ciertas áreas de memoria, **no se crea** la estructura correspondiente.
- **Paginables:** Las tablas de segundo nivel pueden paginarse, es decir, estar también en disco y traerse solo si se necesitan. **Ahorra más memoria.**

Desventaja:

- **Accesos múltiples a memoria:** La MMU debe acceder dos veces a memoria principal. Una vez para acceder a la tabla de primer nivel y otra vez para acceder a la tabla de segunda nivel.
 - Como se puede generalizar a n niveles, si tengo n niveles, implicaría n accesos a memoria, es costoso.

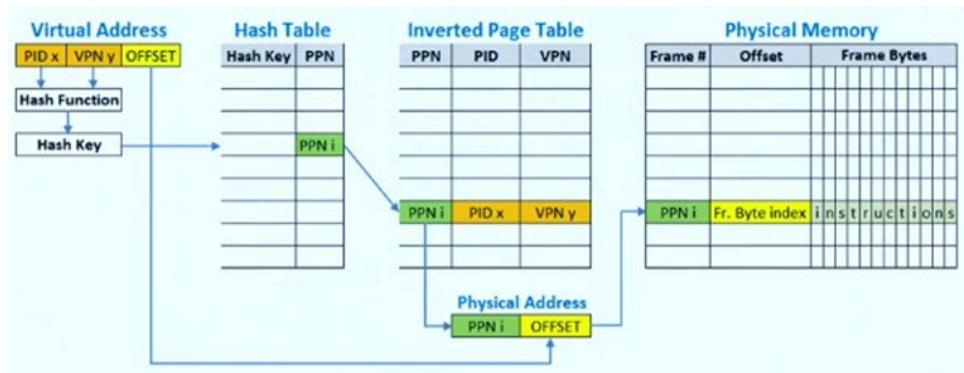
Tabla invertida: Hashing

Esta técnica es utilizada en Arquitecturas **donde el espacio de direcciones es muy grande debido al tamaño de las tablas de páginas requeridas** (que

ocuparían muchos niveles) y, por lo tanto, donde la traducción es muy costosa.

Se la llama “**tabla de páginas invertida**” ya que es la visión inversa a las anteriores técnicas, donde habían tantas entradas como páginas tenía un proceso, estén o no cargadas en memoria principal. Ahora:

- Hay **una sola tabla** para todo el sistema.
- Hay **una entrada por cada marco de página en la memoria real (RAM)**. Sólo se mantienen los PTEs (page table entries) de páginas **presentes en memoria física**.
- El **espacio de direcciones de la tabla se refiera al espacio físico de la RAM**, en vez del espacio de direcciones virtuales de un proceso.
 - A diferencia de las tablas tradicionales que tienen una entrada por cada página virtual, acá se invierte el concepto: la tabla representa la memoria física, no virtual.
- Esta técnica es usada en PowerPC, UltraSPARC, y IA-64. Sin embargo, no todas las arquitecturas tienen la capacidad para manejar la tabla invertida debido a que es costoso tener una memoria del estilo (por ejemplo, x86 **no lo implementa** porque la sobrecarga de búsqueda con hashing lo hace ineficiente en ese entorno).



¿Como funciona?

Dado que solo se mantiene información para las páginas que están cargadas en memoria principal y no se puede buscar directamente por número de página virtual, se utiliza una función **hash**.

- El número de página es transformado en un valor de HASH
 - Cuando un proceso accede a una dirección virtual, se extrae su **número de página virtual**, y se aplica una **función hash** sobre ese

número

- **El HASH se usa como índice de la tabla invertida** para encontrar el marco asociado
 - El valor del hash obtenido se utiliza para **acceder directamente a una entrada en la tabla invertida**, que podría contener la asociación entre esa página virtual y un marco físico.

Esta búsqueda permite saber si esa página virtual está cargada en memoria y, en caso de estar presente en la tabla, extraer el marco de página para luego obtener la **traducción a dirección física** correspondiente.

Si no está presente en tabla, corresponde a un **fallo de página**.

Este enfoque **reduce el uso de memoria** cuando el **espacio de direcciones virtuales es muy grande**, ya que no es necesario mantener toda la tabla de páginas completa, sino solo las entradas activas (las páginas actualmente cargadas en RAM).

→ ¿Qué sucede cuando el hash da igual para 2 direcciones virtuales?

Se define un **mecanismo de encadenamiento** para solucionar colisiones, es decir, una forma de almacenar varias entradas que caen en la misma posición (por ejemplo, una lista enlazada) para buscar cuál es la correcta.

Tamaño de la Pagina

El **tamaño de una página** depende de la **cantidad de bits asignados al desplazamiento (offset)** dentro de la dirección virtual.

Este número de bits está determinado por el **hardware**, por lo tanto, no es una decisión del sistema operativo.

→ Tamaño de página chico

- **Menor fragmentación interna:** Hay menos probabilidades de que queden grandes espacios sin usar dentro de una página, ya que cada página es más ajustada.
- **Más páginas por proceso:** Con menos bits para el desplazamiento, hay más bits disponibles para identificar páginas. Esto significa más páginas para representar un proceso completo → tablas de páginas más grandes.
- **Más páginas pueden residir en memoria:** Si una página es grande y solo se usa una parte, se desperdicia espacio. Con páginas más pequeñas, se

puede cargar exactamente lo que el proceso necesita y permitir que **más procesos tengan sus páginas cargadas al mismo tiempo**, lo que mejora la concurrencia y permite tener información más concisa.

→ Tamaño de página grande

- **Mayor fragmentación interna:** Es más probable que queden partes sin usar dentro de una página grande.
- **Mas rápido mover páginas hacia la memoria principal.**
 - Las memorias secundarias (como los discos) están diseñadas para transferir grandes bloques de datos de forma más eficiente. Así, mover una sola página grande es más rápido que mover varias pequeñas.

Tamaño de página → Relación con la E/S

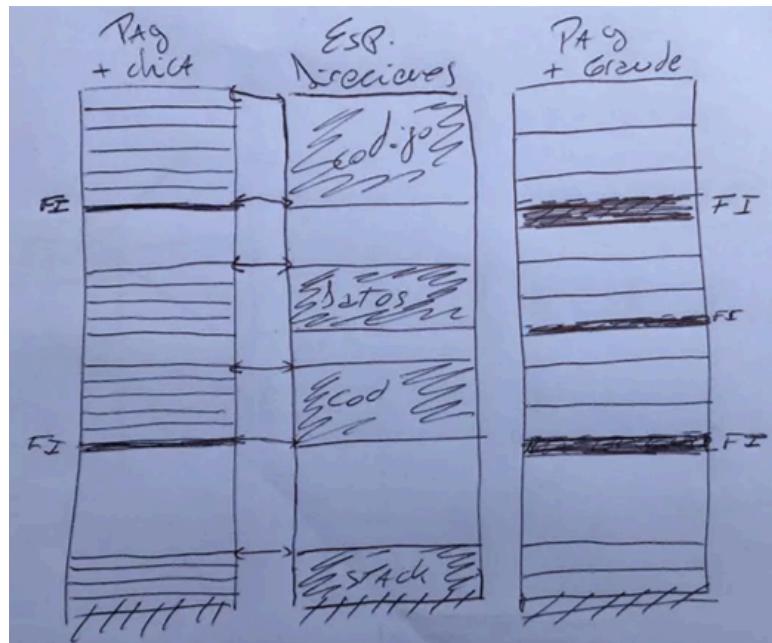
Cuando se mueve información entre la memoria principal y el disco, hay que tener en cuenta 3 factores clave que afectan el tiempo total:

Factor	Tiempo Aproximado
Latencia	8 ms
Tiempo de búsqueda	20 ms
Velocidad de transferencia	2 MB/s

- ✓ Pagina de 512 bytes
 - 1 pagina → total: 28,2 ms
 - Solo 0,2 ms de transferencia (1%)
 - 2 paginas → 56,4 ms
- ✓ Pagina de 1024 bytes
 - total: 28,4 ms
 - ▲ Solo 0.4 ms de transferencia

El mayor costo está en la **latencia y búsqueda**, no en la transferencia en sí. Por eso, si se puede hacer **una sola operación con una página grande (en lugar de realizar, por ejemplo, dos operaciones con páginas más pequeñas)**, se evita pagar dos veces los costos fijos que implican latencia y búsqueda, lo que mejora el rendimiento.

Tamaño de página → Relación con la fragmentación interna



Dado que, por razones de protección, no se suele compartir información de distinta naturaleza dentro de una misma página (por ejemplo, código y datos), es común que se asignen páginas separadas para cada sección del espacio de direcciones (como código, datos, pila, etc.)

Si cada sección del espacio de direcciones tiene un tamaño que no coincide con el tamaño de página correspondiente a esa sección en la memoria principal, es probable que el final de la página asignada a esa sección no esté completamente llena. El “fragmento que queda” sin usar dentro de la página, debido a que el contenido no se ajusta exactamente al tamaño de la página, es lo que se llama fragmentación interna.

- Cuando el tamaño de página es pequeño, la fragmentación interna tiende a ser menor, ya que los espacios desperdiciados por sección también serán más pequeños.
- Cuando el tamaño de página es grande, la fragmentación interna tiende a aumentar, dado que el desperdicio potencial en cada sección también crece.

TLB (Translation Lookaside Buffer)

→ Cuando la CPU accede a una **dirección virtual**, necesita traducirla a una **dirección física** mediante una **tabla de páginas**. Esto implica varios accesos a memoria:

- Uno (o más) para obtener la entrada en tabla de páginas
- Uno para obtener los datos

 Para solucionar este problema, se usa la TLB (Translation Lookaside Buffer).

La TLB es una **memoria caché de alta velocidad** que se encuentra en la CPU ligada a la MMU y **almacena entradas** de tabla de páginas.

Como la TLB es muy pequeña en comparación con la memoria principal, almacena sólo las entradas de la tabla de páginas que fueron **usadas más recientemente**.

Cuando la TLB se llena, se utiliza una **política de reemplazo** para decidir qué entrada descartar. Una política común es **LRU (Least Recently Used)**, que reemplaza la entrada que **hace más tiempo que no se utiliza**.

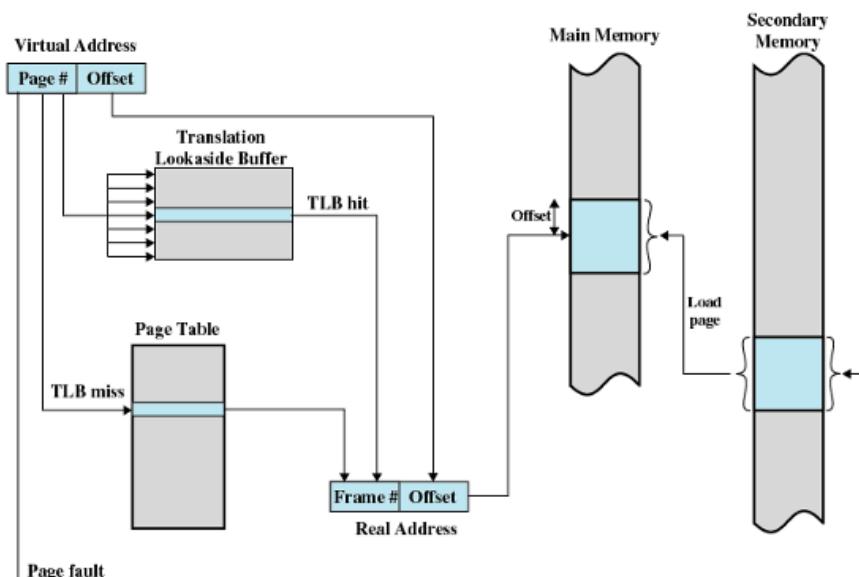


Figure 8.7 Use of a Translation Lookaside Buffer

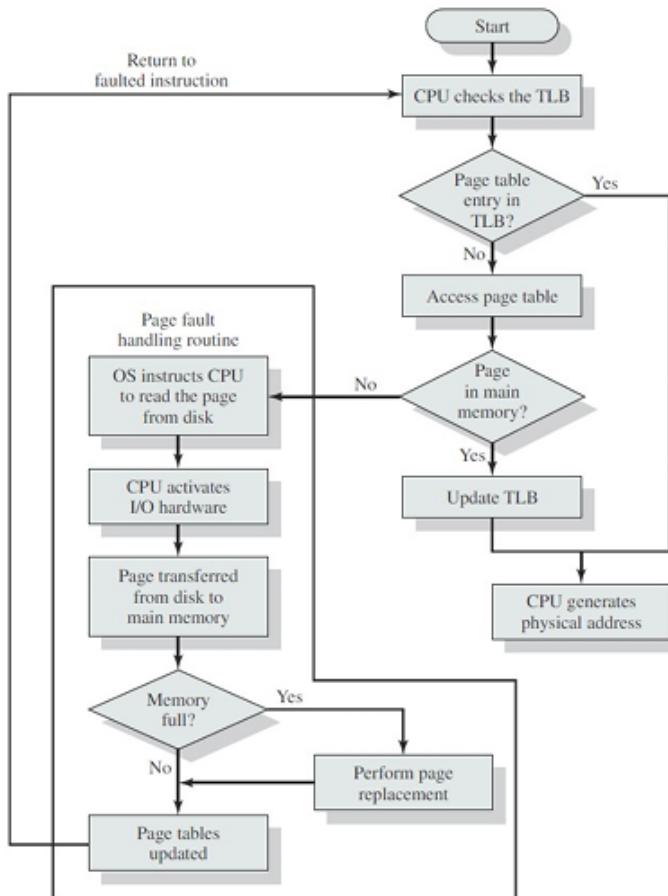
¿Cómo funciona?

Cada vez que un programa accede a una dirección de memoria (por ejemplo, al leer una variable), la CPU necesita traducir la **dirección virtual** que el programa usa en su espacio de direcciones, a la **dirección física** real de la RAM.

Normalmente se hace consultado la tabla de páginas, que está en memoria, pero ahora entra en juego la TLB:

1. El procesador (la MMU) consulta primero la TLB para ver si ya se encuentra la tabla de página buscada cargada.

2. TLB hit (acierto): Si la entrada de la tabla de páginas se encuentra en la TLB, se obtiene el frame (marco) y se arma la dirección física, evitando tener que acceder a la tabla de páginas en memoria principal.
3. TLB miss (fallo): Si la entrada de la tabla de páginas no se encuentra en la TLB:
 - a. El número de página es usado como índice en la tabla de páginas del proceso, es decir, se accede a la tabla de páginas de la memoria principal.
 - i. Se controla si la página está en la memoria principal. Si no lo está, se genera un Page fault.
 - b. Una vez que se obtuvo la dirección, se actualiza la TLB para incluir la nueva entrada (para que futuras traducciones de esa misma página sean más rápidas)



📌 Context switch y la TLB

Cuando ocurre un cambio de contexto (context switch), la TLB puede quedar con entradas de otro proceso que ya no son válidas para el nuevo proceso. Por lo tanto, **se invalida** la TLB durante un context switch, **borrando todas sus entradas**.

Consecuencia: si el **quantum de CPU es muy pequeño**, habrá muchos cambios de contexto por segundo, y por lo tanto muchas invalidaciones de la TLB, lo cual **afecta negativamente el rendimiento de la memoria**, ya que se pierde la ventaja de tener traducciones en caché.

Entonces, ¿Por qué es útil la TLB?

La TLB permite **evitar o minimizar los accesos adicionales a la tabla de páginas en memoria principal** al traducir direcciones virtuales a físicas, manteniendo una copia caché de las traducciones más recientes.

Esto **mejora mucho el rendimiento** (ahorando tiempo) en sistemas con administración de memoria virtual.

Políticas en el manejo de MV (Virtual Memory)

Fetch Policy Demand paging Prepaging	<i>Cuando una página debe ser llevada a la memoria</i>
Placement Policy <i>Donde ubicarla (best-fit, first-fit, etc...)</i>	
Replacement Policy Basic Algorithms Optimal <i>Elección de víctima</i> Least recently used (LRU) First-in-first-out (FIFO) Clock Page Buffering	Resident Set Management Resident set size Fixed Variable Replacement Scope Global Local
	<i>Cuántas páginas se traen a memoria</i>
	Cleaning Policy Demand Precleaning
	<i>Cuando una página modificada debe llevarse a disco</i>
	Load Control <i># de procesos en memoria</i> Degree of multiprogramming

Asignación de Marcos

Esta sección busca responder:

- ¿Cuántas páginas de un proceso están en memoria? ¿Cuál es el tamaño del conjunto residente?
- ¿Cuánta memoria real ocupa un proceso?

La respuesta depende del tipo de asignación de marcos (frames) que utilice el sistema operativo.

Asignación Dinámica

En este enfoque, el número de marcos asignados a cada proceso varía dinámicamente. El sistema operativo va otorgando marcos **a medida que el proceso los necesita.**

Possible problema:

Si se utiliza, por ejemplo, un algoritmo de planificación con prioridades, puede suceder que un **proceso de alta prioridad** comience a solicitar muchas páginas, ocupando gran parte de la memoria. Esto puede perjudicar a otros procesos de **menor prioridad**, que recibirán menos marcos y podrían quedar limitados en su ejecución.

Asignación Fija

En este enfoque, se le asigna a cada proceso una **cantidad fija de marcos**, que no cambia durante su ejecución. Existen dos variantes:

- **Asignación equitativa:** La cantidad total de frames o marcos disponibles se distribuyen de manera igualitaria entre todos los procesos, sin importar el tamaño de cada uno.
 - Por ejemplo, si hay 100 marcos y 5 procesos, se asignan 20 marcos a cada uno.
- **Asignación proporcional:** La cantidad total de frames o marcos disponibles se distribuyen acorde al tamaño del proceso. Procesos más grandes reciben más marcos.

Desventaja común:

Ninguna de las dos variantes es completamente eficiente.

La **cantidad de procesos en memoria cambia constantemente**, por lo que habría que estar reasignando marcos todo el tiempo, lo cual genera **sobrecarga en el sistema operativo** y un **mayor uso de CPU**.

¿Cómo se soluciona?

El sistema operativo juega con ambas asignaciones: asigna una cantidad de marcos fija a cada proceso pero si se detecta que el proceso necesita más marcos (porque genera muchos fallos de página), el SO **puede asignarle más**. Lo mismo para el caso contrario, si al proceso le sobran marcos, el SO **puede reducirle la cantidad** para dárselos a otro proceso que los necesite.

Reemplazo de páginas

Cuando ocurre un **fallo de página** y **todos los marcos están ocupados**, el sistema operativo debe **seleccionar una "página víctima"**, es decir, una página que será **sacada de la memoria principal** para hacer lugar a la nueva.

¿Cuál sería reemplazo óptimo?

- El reemplazo **óptimo** sería aquel que **elimina la página que no va a ser usada en el futuro cercano**, es decir, que no será referenciada nuevamente. Esto minimiza la cantidad de futuros fallos de página.

Problema: Este método no es viable en la práctica, porque **no se puede predecir el futuro** ni saber con certeza qué hará el proceso.

 **Solución práctica:** La mayoría de los algoritmos reales **predicen el futuro mirando el comportamiento pasado**, es decir, basándose en el historial de uso de las páginas.

Algoritmos de Reemplazo

→ Optimo (ideal, téórico)

Es solo teórico, reemplaza la página que no se utilizará en el futuro más cercano.

→ FIFO (First In, First Out)

Las páginas se sacan en el orden que ingresaron, no se tiene en cuenta ni el uso pasado ni futuro.

Es simple de implementar, pero puede ser ineficiente (sacar páginas muy usadas).

→ LRU (Least Recently Used)

Reemplaza la página que **hace más tiempo no fue utilizada**. Por ende, asume que las páginas usadas recientemente seguirán siendo utilizadas.

Requiere **soporte de hardware** (timestamp por página o bit de acceso) para mantener la información de uso.

→ Segunda oportunidad (Second Chance)

Es un avance del FIFO tradicional que beneficia a las páginas **que fueron referenciadas recientemente**.

- Si la página que está por salir tiene el **bit de referencia (R)** en 1: No se reemplaza, se pone R=0 y se reubica al final de la cola (se le da una "segunda oportunidad").
- Si la página que está por salir tiene el **bit de referencia (R)** en 0: Se reemplaza normalmente.

→ **NRU (Non Recently Used)**

Utiliza los **bits R (referenciado)** y **M (modificado)**.

Clasifica las páginas en 4 categorías según esos bits, priorizando:

1. No referenciada y no modificada → **mejor candidata a ser reemplazada**.
2. No referenciada pero modificada.
3. Referenciada y no modificada.
4. Referenciada y modificada → **última opción a reemplazar**.

Se favorece sacar páginas **no referenciadas recientemente** y que **no fueron modificadas** (así se evitan escrituras al disco).

Alcance del Reemplazo de páginas

Cuando ocurre un fallo de página y es necesario seleccionar una página víctima para liberar un marco de memoria, existen **dos enfoques posibles** según **de dónde se permite elegir esa víctima**:

Reemplazo Global

El fallo de página de **un proceso puede reemplazar la página de cualquier proceso**.

- El SO **no controla la tasa de page-faults de cada proceso** por separado.
- Un proceso puede terminar **tomando frames de otro proceso**, aumentando así la cantidad de marcos asignados a él dinámicamente.
 - **✗ Esto** puede hacer que un **proceso de alta prioridad** le quite páginas a un proceso de **menor prioridad**, quitándole recursos.

Reemplazo Local

El fallo de página de un proceso solo puede reemplazar **páginas de su propio conjunto residente**, es decir, sus propias páginas.

- No cambia la cantidad de frames o marcos asignados al proceso.

- El SO puede determinar y controlar cuál es la tasa de page-faults de cada proceso por separado.
- ~~• Sin embargo, un proceso puede tener frames o marcos asignados que no usa, y no pueden ser usados por otros procesos.~~

Observaciones:

- **Generalmente, se utiliza más el reemplazo local**, ya que permite un mayor control sobre los recursos asignados a cada proceso y evita que un proceso invada la memoria de otro.
- Lo ideal es mantener la **tasa de fallos de página (p)** lo más baja posible.
 - Si p es muy alto y se usa reemplazo global, es difícil saber cuál proceso está generando los problemas de rendimiento.

Table 8.5 Resident Set Management

	Local Replacement	Global Replacement
Fixed Allocation	<ul style="list-style-type: none"> • Number of frames allocated to a process is fixed. • Page to be replaced is chosen from among the frames allocated to that process. 	<ul style="list-style-type: none"> • Not possible.
Variable Allocation	<ul style="list-style-type: none"> • The number of frames allocated to a process may be changed from time to time to maintain the working set of the process. • Page to be replaced is chosen from among the frames allocated to that process. 	<ul style="list-style-type: none"> • Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.

Consideraciones prácticas:

- El reemplazo global **no es compatible con asignación fija**, porque no se podrían "robar" páginas a otros procesos si cada uno tiene un número fijo de marcos asignado.
- Cuando se utilizan los algoritmos de reemplazo, lo más común es que solo busque una víctima entre las páginas pertenecientes al proceso que generó el fallo de página. Si tuviera que revisar **todas las páginas de todos los procesos**, la búsqueda sería **muy costosa** y poco eficiente.

09- Memoria 3.a

Thrashing (hiperpaginación)

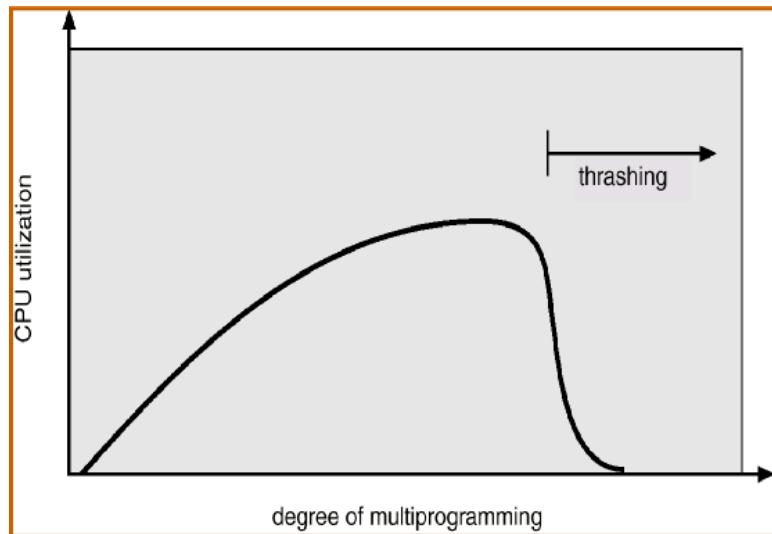
Thrashing es un problema que se presenta en la paginación por demanda cuando **el sistema pasa más tiempo paginando que ejecutando procesos**.

Como consecuencia, hay una **baja importante en el rendimiento (performance) del sistema** ya que se desperdicia gran parte del tiempo de la CPU esperando operaciones de E/S de páginas en lugar de ejecutar instrucciones de procesos.

El *thrashing* ocurre cuando los procesos que están en memoria **no tienen suficientes páginas cargadas** para continuar su ejecución, lo que provoca una sucesión de fallos de página (*page faults*) y un ciclo de degradación...

Ciclo del thrashing

1. El kernel **monitorea el uso de la CPU** para evaluar el rendimiento.
2. Si hay baja utilización de la CPU → el SO **aumenta el grado de multiprogramación** (el número de procesos en memoria) para intentar mejorar el uso de la CPU.
3. Como aparecen nuevos procesos que requieren marcos de página, si el **algoritmo de reemplazo es global**, los nuevos procesos pueden sacarle frames a otros procesos, reduciendo el número de páginas disponibles para ellos.
4. Si un proceso necesita más frames, **comienzan los fallos de página (page-faults)** y se producen robos de frames a otros procesos.
5. El tiempo se consume en intercambio de páginas (swapping de páginas) y en la espera en colas de dispositivos de E/S para leer o escribir páginas (encolamiento en dispositivos), lo que reduce aún más el uso de la CPU.
6. Vuelve a 1).



El scheduler de CPU y el thrashing

El *scheduler* de CPU y el **thrashing** están estrechamente relacionados.

A medida que el *scheduler* de largo plazo (*long-term scheduler*) incrementa el grado de multiprogramación, la utilización de la CPU aumenta. Si este grado es demasiado alto, puede empezar a ocasionar *thrashing*: los procesos empiezan a generar una gran cantidad de *page faults* porque no tienen en memoria todas las páginas que necesitan.

Por lo tanto, un alto de grado de multiprogramación →

- Puede impactar en la técnica de administración de memoria.
 - Para poder decir que se tiene una buena administración de memoria, los procesos que se están ejecutando deberían tener todas las páginas que necesiten cargadas.
- Incrementa la actividad de paginado y satura la E/S, ya que las colas de espera de los dispositivos se llenan con peticiones de lectura de páginas.

Como consecuencia, la utilización de la CPU disminuye, lo que lleva nuevamente al *scheduler* a aumentar el grado de multiprogramación, repitiendo el ciclo:

1. Cuando se disminuye el uso de la CPU, el *scheduler long term* aumenta el grado de multiprogramación.
2. Los nuevos procesos provocan más *page faults* y, por lo tanto, más actividad de paginado.
3. El uso de la CPU disminuye aún más.

4. Se vuelve al paso 1.

Control del thrashing

Possible solución → Se puede limitar el thrashing usando **algoritmos de reemplazo local**, donde si un proceso entra en *thrashing* no puede robarle *frames* a otros procesos.

→ Aunque esta técnica no considera el grado de multiprogramación y puede afectar el rendimiento general, mantiene el problema más controlado.

Conclusión sobre thrashing

Si el sistema operativo puede garantizar que cada proceso disponga de todos los *frames* que necesita, el *thrashing* no se produciría.

Como abordar el thrashing

→ Una manera de abordar esta problemática es utilizando la estrategia de **Working Set** basada en el modelo de localidad.
→ Otra estrategia similar es el **algoritmo PFF (Frecuencia de Fallos de Página)**.

Pero primero, ¿Qué es el modelo de localidad?

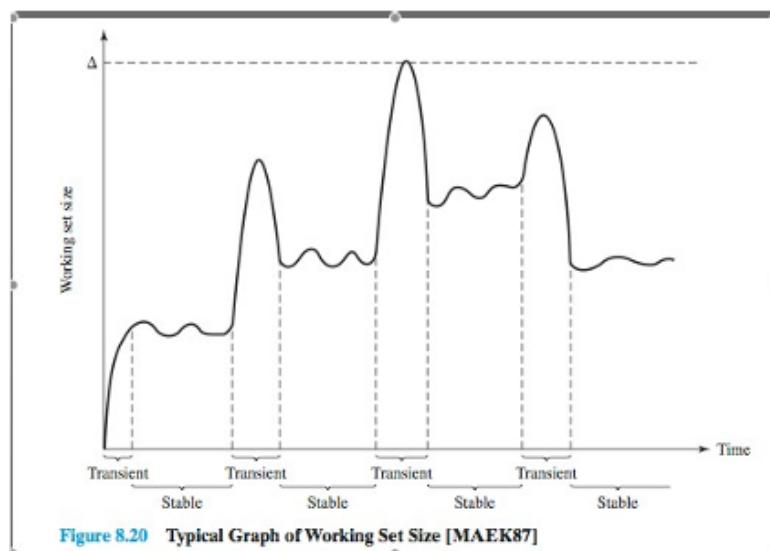
- Un programa se compone o puede dividirse en varias localidades.
- La localidad de un proceso, en un momento dado, está formada por el conjunto de páginas que tiene en memoria en ese instante.
 - Por ejemplo: cada rutina será una nueva localidad, se refieren sus direcciones (cercanas) cuando se está ejecutando.
- Las referencias (accesos) a datos e instrucciones dentro de un proceso tienden a agruparse, es decir, cuando un proceso se ejecuta, las direcciones de memoria que utiliza **tienden a estar cercanas unas tras otras, formando grupos**.
- En cortos períodos de tiempo, el proceso necesitará pocas "piezas" del proceso (por ejemplo, una página de instrucciones y otra de datos...)

Entonces, ¿**como** funciona?

Dicho todo esto, el **modelo de localidad**, también llamado **cercanía de referencias o principio de cercanía**, plantea la idea de que cuando un proceso se ejecuta, **las direcciones de memoria que accede** -ya sean instrucciones o datos- **durante su ejecución tienden a agruparse y estar próximas entre sí**.

Durante su ciclo de vida, un proceso **atraviesa distintas localidades**. Mientras permanece una localidad, las direcciones de memoria accedidas son cercanas y, por lo tanto, las páginas necesarias suelen estar cargadas en memoria.

Cuando cambia de localidad, se produce un pico en la tasa de fallos de página, ya que comienza a necesitar páginas que no están en memoria. Una vez que se cargan dichas páginas en el working set, la tasa se estabiliza hasta el próximo cambio de localidad.



Si el sistema operativo logra mantener en el conjunto residente (working set) las páginas correspondientes a la localidad actual de un proceso, puede reducir la frecuencia de fallos de página. Sin embargo, predecir con precisión la próxima localidad que recorrerá un proceso es difícil, debido a la presencia de estructuras de control como bucles, condicionales o saltos.

Para prevenir la hiperactividad (trashing), un proceso debe tener en memoria RAM sus páginas más activas para seguir ejecutándose sin interrumpirse para traer páginas desde disco (menos page fault).

Técnica - El modelo de working set

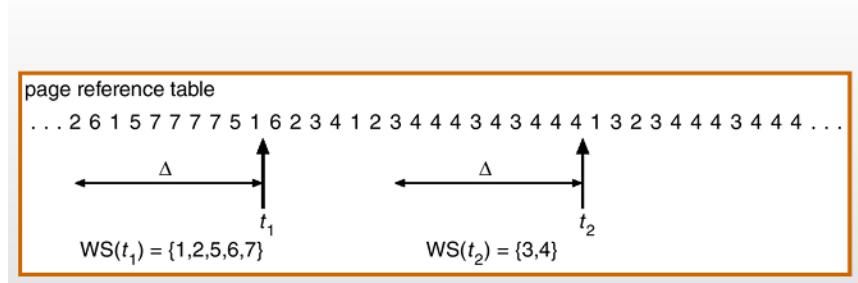
Se basa en el modelo de localidad: Un proceso tiende a acceder a un grupo reducido de páginas cercanas durante un período de tiempo (su localidad actual).

Esta técnica plantea **definir una ventana del working set (Δ)** → Es la cantidad de referencias recientes a páginas que se toman en cuenta para estimar la localidad actual.

Working set → Es el conjunto de páginas que han sido referenciadas en las últimas Δ referencias.

Este conjunto es una aproximación a la localidad que está usando el proceso en determinado momento.

$\Delta=10$



Ejemplo Se plantea una ventana de tamaño 10 (se miran las ultimas 10 referencias: 2,6,1,5,7,7,7,7,5,1). Se arma el conjunto con las páginas que fueron referenciadas en dicha ventana (1,2,5,6,7) → El WSSi=5.

Selección de víctima:

Si en el instante de tiempo t_1 se debe seleccionar una víctima:

- No se elige ninguna página que esté en el working set actual.
 - Se evita sacar páginas que probablemente se seguirán usando en la localidad activa.

Problemas con la elección de Δ :

¿Que valor se le da al Δ para obtener el mejor conjunto de trabajo (working set) que representa la definición teórica de localidad?

- Δ muy chico: Puede suceder que el working set no cubra realmente toda la localidad del proceso (se expulsan páginas que se necesitarán pronto).
- Δ muy grande: Puede suceder que el working set mantenga páginas innecesarias, mezclando localidades distintas.

¿Como el SO obtiene la información sobre qué páginas se fueron referenciando recientemente?

- Necesita **soporte de hardware** (por ejemplo, se puede utilizar el **bit de referencia** en cada entrada de la tabla de páginas para registrar referencia).

- Mecanismo típico:
 - Cada cierto intervalo, el SO inspecciona la tabla de páginas del proceso en ejecución.
 - Registra qué páginas se usaron.
 - Limpia los bits de referencia (los pone en 0) para la próxima medición.

Medida del working set

El *working set* mide el conjunto de páginas activas que un proceso necesita en un momento dado para ejecutar sin generar fallos de página excesivos (thrashing) y terminar gastando más tiempo de CPU en administración de memoria.

Sea:

- m = cantidad frames disponibles.
- WSS_i = tamaño del working set del proceso p_i .
- $\sum WSS_i = D \rightarrow$ Sumando todos los tamaños del working set de todos los procesos se obtiene la demanda total de frames.
- D = demanda total de frames para todos los procesos activos (todos los frames que necesito para mantener todos los conjuntos de trabajo de todos los procesos).

| Si $D > m$, habrá thrashing.

Esto ocurre porque la cantidad de *frames* que se necesitan para mantener todos los conjuntos de trabajo es mayor a la cantidad de frames disponibles.

Se entra en thrashing ya que los procesos empezarán a causar fallos de páginas para traer las páginas que les faltan a memoria RAM.

Observación: Independientemente de si el reemplazo de páginas es global o local, el problema persiste:

- En reemplazo global, un proceso podría “robar” páginas de otro, afectando su rendimiento.
- En reemplazo local, un proceso se ve obligado a reemplazar sus propias páginas, pero igualmente sufrirá continuos *page faults*.

Prevención del thrashing

- El SO monitoriza a cada proceso y le asigna *frames* suficientes para cubrir su **WSS_i**.
- Si quedan *frames* libres, puede iniciarse otro proceso
- Si D crece, excediendo m, se suspende un proceso y se reasignan sus *frames*.

De esta forma, se mantiene un alto grado de multiprogramación y se optimiza el uso de CPU.

Problema del working set

- Mantener un registro de los WSS_i (medida del working set del proceso p_i)
 - Es muy costoso, se utiliza mucho tiempo de CPU para lograr revisar todas las páginas, actualizar estructuras, y resetear bits.
- La ventana es móvil
 - “**Se desplaza**” con cada nueva referencia de página que hace el proceso. Por ejemplo, si Δ = 10 referencias, cada vez que llega una referencia nueva, se descarta la más antigua y se añade la nueva.
 - Esto significa que el SO debe estar **recalcular constante**mente qué páginas están dentro del working set.

Técnica - Frecuencia de fallo de páginas

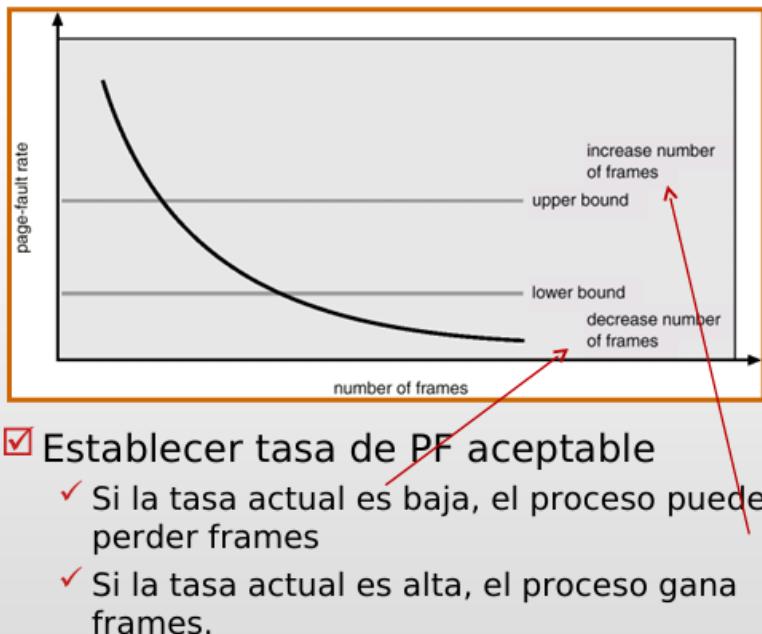
La técnica PFF (Page Fault Frequency o Frecuencia de Fallo de Páginas) **utiliza la tasa de fallos de página** para estimar si el conjunto residente de un proceso representa adecuadamente su WS.

El objetivo es mantener **la frecuencia de tasa de fallos de página en un numero manejable**, es decir, un valor que no afecte la **performance** del sistema.

Consiste en aceptar determinada PFF por proceso y poder determinar la situación de un proceso con dicha PFF.

Principios básicos

- PFF: Frecuencia de page faults.
- PFF alta → Los procesos necesitan más marcos
- PFF baja → Los procesos tienen marcos asignados que le sobran.



Se deben **establecer límites** superior e inferior de las PFF's deseadas:

- **Si se excede PFF máx.** → Se le asignan más frames al proceso, ya que el mismo genera muchos PF y probablemente los esté necesitando.
- **Si la PFF está por debajo del mínimo** → Se le pueden quitar frames al proceso para ser utilizados en los que necesitan más.

Reasignación y suspensión:

- Si un proceso con PFF alta convive con uno con PFF baja, el primero con PFF alta le puede "robar" marcos al segundo.
- **Si no hay marcos libres y todos los procesos superan el PFF máx.**, el sistema puede suspender uno o más procesos para liberar memoria.

Implementación práctica

Es una técnica implementable, de hecho los SO la utilizan para **definir el tamaño del conjunto residente de un proceso**.

Una vez asignados los marcos, sobre ese conjunto residente se aplican **algoritmos de reemplazo de página** como **LRU, FIFO**, etc., para decidir qué página expulsar ante un fallo.

Demonio de paginación

En todos los sistemas operativos existe un proceso llamado **demonio** creado por el SO durante el arranque y ejecutado en segundo plano (background). Su

función es **apoyar a la administración de la memoria** realizando actividades/tareas que no dependen de llamadas explícitas al sistema.

Este dominio se ejecuta cuando el sistema tiene una baja utilización o cuando algún parámetro de la memoria lo indica, por ejemplo:

- Poca memoria libre.
- Mucha memoria modificada.

Tareas:

- **Limpieza de páginas modificadas:** Limpia las páginas modificadas, es decir, escribe su contenido actualizado en el área de intercambio (swap) y la marca como "no modificada" (dirty bit=0). Esto no la saca de RAM todavía, solo la deja lista para ser reemplazada si es necesario, lo que:
 - Reduce el tiempo de swap posterior, ya que las páginas quedan "limpias" (en un futuro fallo de página, su resolución será más rápida ya que no hay que realizar un swap out previo, el SO puede descartarla inmediatamente y hacer solo el swap in).
 - Escribe varias páginas contiguas que fueron modificadas al área de intercambio en una sola operación de E/S, esto reduce el tiempo de transferencia.
- **Análisis del estado de la RAM:** Analiza el estado general de la RAM y la frecuencia de fallo de página de los procesos y mantiene un cierto número de marcos libres en el sistema.
- **Liberación diferida de páginas:** Demora la liberación de una página hasta que haga falta realmente

Basicamente, el demonio hace tareas en nombre del SO, en modo kernel, sin depender que ocurran llamadas al sistema.

Ejemplos en sistemas reales

- En Linux → Proceso "**kswapd**"
- En Windows → Proceso "**system**"

Memoria compartida

- Gracias al uso de la tabla de páginas, varios procesos pueden **compartir un mismo marco de memoria**.

Para lograrlo, ese marco debe estar **asociado a una página en la tabla de páginas de cada proceso**.

- El número de página que apunta a ese marco puede ser diferente en cada proceso, ya que cada uno maneja su propio espacio de direcciones lógicas.

Código compartido:

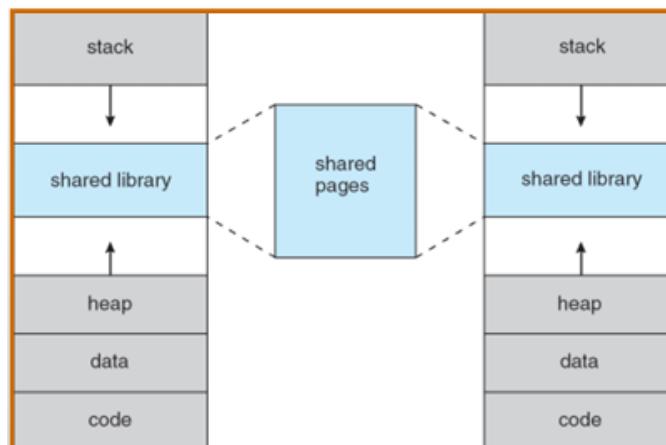
- Los datos son privados a cada proceso y se almacenan en páginas **no compartidas**.
- Los procesos pueden compartir una sola copia de código si está marcada como solo lectura, por ejemplo, editores de texto, compiladores, etc.

Motivos para compartir memoria:

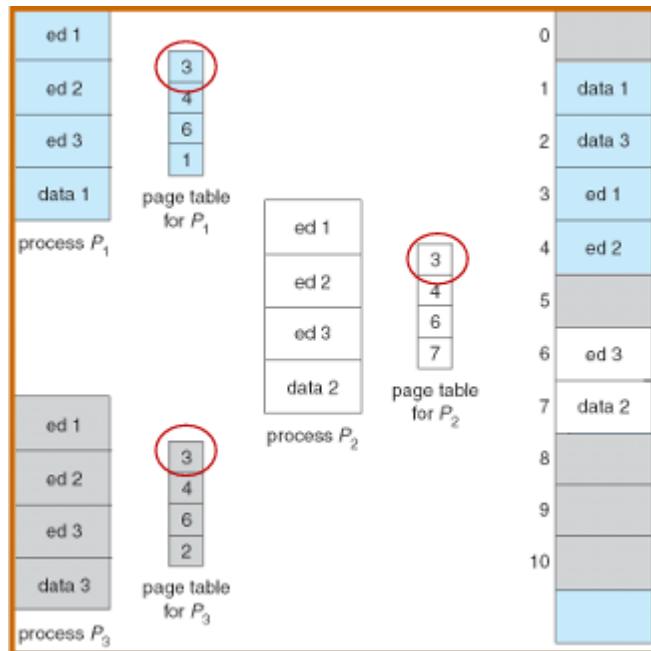
- **Eficiencia:** Permite hacer un uso óptimo de la memoria física. Por ejemplo, dos procesos ejecutando el mismo programa no necesitan tener copias duplicadas del código en RAM.
- **Cooperación explícita entre procesos:** Algunos procesos deciden compartir memoria para intercambiar información rápidamente, por ejemplo, dos procesos que trabajan sobre una misma estructura de datos para comunicarse.

En ambos casos, el **SO puede decidir compartir código automáticamente**, sin que los procesos lo pidan explícitamente.

Analiza las tablas de páginas y, si detecta que más de un proceso necesita el mismo contenido (por ejemplo, una librería o ejecutable), ajusta las entradas para que ambos apunten al **mismo marco en RAM**.



Ejemplo: dos procesos que usan la misma librería apuntan a los mismos marcos de código en RAM.



Ejemplo: varios procesos ejecutan el mismo editor. Cada uno tiene 3 páginas que corresponden al editor (compartidas) y 1 página a los datos (no compartida). En las tablas de páginas, las entradas de código apuntan al mismo lugar en RAM, pero la entrada de datos apunta a marcos distintos para cada proceso.

Mapeo de archivo a memoria

Es otra técnica que ofrecen los sistemas operativos que permite que un proceso **asocie el contenido de un archivo a una región de su espacio de direcciones virtuales**.

Esto significa que **acceder a esa región de memoria es equivalente a acceder al archivo**, sin necesidad de usar funciones tradicionales de entrada/salida.

Características:

- El contenido del archivo no se sube a memoria inmediatamente, si no que se trae cuando se generan page faults.
 - El contenido de la página que genera el PF es obtenido desde el archivo asociado (no del area de intercambio)
 - Cuando el proceso finaliza o el archivo se libera, si se modificaron páginas deben escribirse de vuelta en el archivo correspondiente.

- Esta técnica permite **entrada/salida más eficiente**, ya que:
 - No requiere llamadas explícitas como open, read o write para manipular el archivo.
 - Se trabaja **directamente sobre la memoria** y el sistema operativo se encarga de **reflejar los cambios en el archivo**.

Usos comunes:

- Cargar **librerías compartidas** (por ejemplo, DLLs en Windows o **.so** en Linux).
- Manipular archivos grandes sin necesidad de leerlos completos en memoria de forma manual.

Copia en escritura (Copy-on-Write, COW)

La copia en escritura (Copy-on-Write, COW) es una técnica que permite que **varios procesos compartir inicialmente las mismas páginas de memoria**.

Funcionamiento básico:

- Al inicio, todos los procesos que comparten esas páginas **acceden a la misma copia en memoria**.
- **Si uno de los procesos intenta modificar una página compartida**, el sistema operativo crea **una copia independiente** de esa página solo para ese proceso.
- Esto evita que los cambios de un proceso afecten a los demás.

COW permite **crear procesos de forma más eficiente debido a que sólo las páginas modificadas son duplicadas**. Esto reduce el tiempo y el consumo de memoria en la creación de procesos.

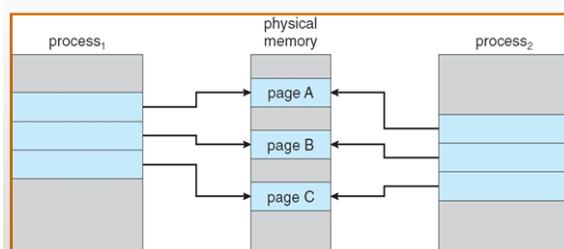
Ejemplo de uso: Esta técnica la usa comúnmente linux, por ejemplo, al crear un proceso hijo con fork.

→ **¿Que hace el sistema operativo con las páginas que contienen código?**

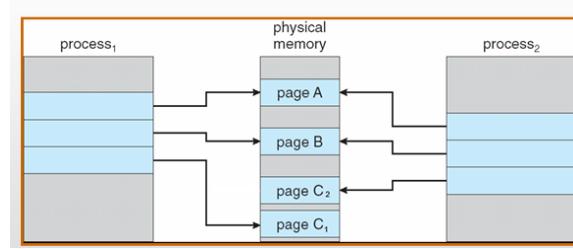
- Las **marca como de solo lectura** en la tabla de páginas.
- Si un proceso intenta escribir en ellas, se produce una **interrupción** por acceso inválido.
- El sistema operativo detecta que no es un error real, sino un intento de modificación de una página compartida, y entonces:

- **Crea una copia** de la página.
- **Actualiza la tabla de páginas** de ambos procesos:
 - La nueva copia queda accesible en modo lectura/escritura para el proceso que la modificó.
 - La página original sigue siendo de solo lectura para el resto.
- Reactiva el proceso que provocó la interrupción para que continúe su ejecución normalmente.

El Proceso 1 Modifica la Página C (Antes)



El Proceso 1 Modifica la Página C (Después)



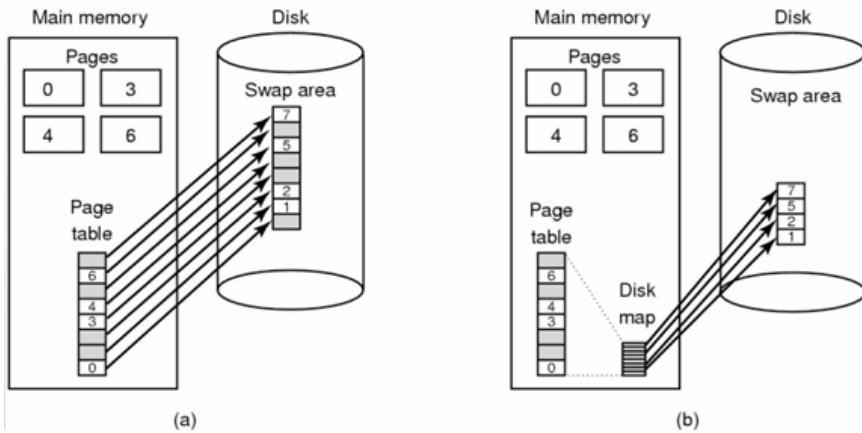
Área de Intercambio

El **área de intercambio** es una zona de almacenamiento secundario (disco) que utiliza el sistema operativo para **guardar temporalmente páginas que no están en uso en la memoria RAM**, liberando así espacio para otras páginas más activas.

Cuando un proceso necesita descargar datos que ya no está utilizando, el sistema operativo los **traslada** desde la RAM al área de intercambio.

Ubicación del área de intercambio:

- En linux: Puede ser una partición o un archivo separado del resto del sistema de archivos.
- En windows: Generalmente es un archivo dentro del sistema de archivos.



Técnicas para la administración del área de intercambio:

a) Reserva completa al crear el proceso →

- Cada vez que se crea un proceso, **se reserva en el área de intercambio un espacio igual al tamaño total de la imagen del proceso.**
 - Por ejemplo, si hay un proceso con 10 páginas, se guarda lugar para 10 páginas en el área de intercambio.
- A cada proceso se le asigna la dirección de comienzo dentro del área de intercambio.
- Cuando se necesita leer una página, se suma el número de página virtual a la dirección de comienzo del área asignada al proceso para ubicarla.
- **Desventaja:** Se desperdicia espacio. Algunas páginas (como código de librerías o ejecutables) pueden estar ya en otro lugar permanente y no es necesario duplicarlas en el área de intercambio.

b) Asignación bajo demanda (más eficiente) →

- No se asigna nada inicialmente. No se asigna espacio en el área de intercambio al crear el proceso.
- Se le asigna espacio en disco (en el área de intercambio) a cada página recién cuando debe intercambiarse (swap out).
- Cuando la página regresa a memoria RAM (swap in), se libera ese espacio en disco.
- El sistema mantiene una estructura de datos en memoria llamada "disk map" que indica en qué parte del disco está cada página realmente almacenada. Generalmente, son páginas de datos.
- **Ventaja:** Optimiza el uso del espacio en el área de intercambio.

-  Problema: Se debe llevar un registro, página por página, de la localización de las páginas en disco.

→ **Cuando una página no está en memoria, sino en el área de intercambio, como podríamos saber en que parte del swap está?**

Primero, en la entrada de la tabla de páginas (PTE) de esa página, el bit de validez (V) está en  (indicando que no está en RAM).

Los demás bits del PTE que normalmente contendrían el número de marco **quedan libres (sin usar)**, se utilizan para almacenar información sobre la ubicación de la página en el área de intercambio.

12-Subsistema de E/S

¿Qué responsabilidades tiene el sistema operativo?

- Controlar dispositivos de E/S
 - Emitir comandos de E/S, gestionar las interrupciones que generan los dispositivos, manejar posibles errores.
- Proporcionar una interfaz de utilización

Problemas

- Heterogeneidad de dispositivos: Hay mucha cantidad de dispositivos, no todos son iguales.
- Características de los dispositivos: No todos los dispositivos tienen las mismas características.
- Velocidad
- Nuevos tipos de dispositivos: Con el tiempo aparecen nuevos dispositivos de E/S y el SO debe poder manejarlos sin realizar cambios bruscos en el diseño (si no sería inmanejable mantenerse al día).
- Diferentes formas de realizar E/S

Lo ideal sería que los programas se abstraigan de todos estos detalles.

Aspectos de los dispositivos de I/O (cont)

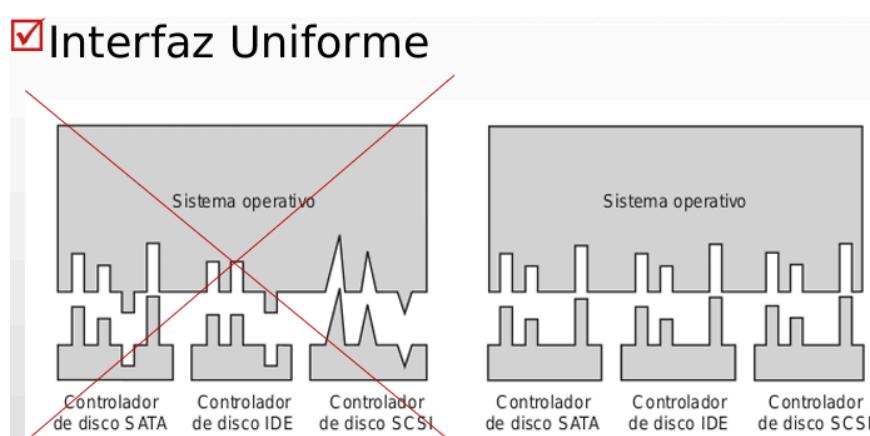
- Unidad de Transferencia
 - Dispositivos por bloques (discos).

- Operaciones: Read, Write, Seek.
- Dispositivos por Caracter (keyboards, mouse, serial ports).
 - Operaciones: get, put
- Formas de Acceso
 - Secuencial o Aleatorio
 - Aspectos de los dispositivos de I/O (cont)
- Tipo de acceso
 - Acceso Compartido: Disco Rígido
 - Acceso Exclusivo: Impresora
- Tipo de acceso:
 - Read only: CDROM
 - Write only: Pantalla
 - Read/Write: Disco

Metas, objetivos y servicios

En general:

- Se desea manejar todos los dispositivos de E/S de manera uniforme, estandarizada
- Ocultar la mayoría de los detalles del dispositivo en las rutinas de niveles más bajos para que los procesos vean a los dispositivos en términos de operaciones comunes como: read, write, open, close, etc.
- Brindar una interfaz uniforme



La forma de utilizar e interactuar con los dispositivos es la misma sin importar el tipo del dispositivo.

Con respecto a la eficiencia:

- Los dispositivos de E/S pueden resultar extremadamente lentos respecto a la memoria y la CPU.
- Se aprovecha el uso de la multiprogramación que permite que un proceso espere por la finalización de su E/S **mientras que otro proceso se ejecuta**

Con respecto a la planificación:

- El sistema operativo debería ser capaz de organizar los requerimientos que hay de los dispositivos para que se utilicen eficientemente.
 - Ej: Planificación de requerimientos a disco para minimizar tiempo

Con respecto al buffering – Almacenamiento de los datos en memoria mientras se transfieren

- Solucionar problemas de velocidad entre los dispositivos
- Solucionar problemas de tamaño y/o forma de los datos entre los dispositivos

Con respecto al caching

- Mantener en memoria copia de los datos de reciente acceso para mejorar performance
 - Por ejemplo, almacenar en memoria los bloques de disco más utilizados para reducir los tiempos de acceso a disco

Con respecto al spooling – Administrar la cola de requerimientos de un dispositivo

- Algunos dispositivos de acceso exclusivo, no pueden atender distintos requerimientos al mismo tiempo: Por ej. Impresora
- Spooling es un mecanismo para **coordinar el acceso concurrente al dispositivo que no puede ser utilizado por varios dispositivos a la vez.**
 - Implementa una cola de requerimientos para ir tomando uno por uno

Con respecto a la reserva de dispositivos:

- Acceso exclusivo: Permitir reservar un dispositivo para acceso y uso exclusivo.

Con respecto al manejo de errores:

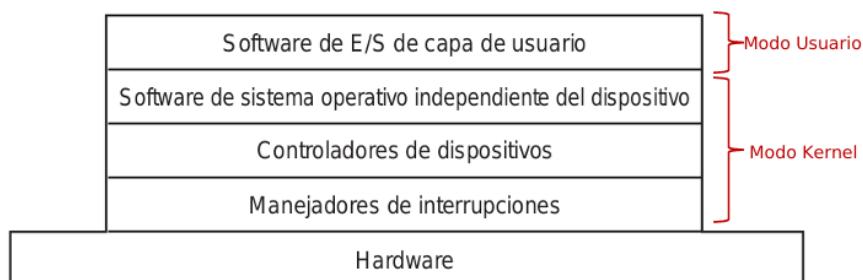
- El S.O. debe administrar errores ocurridos (lectura de un disco, dispositivo no disponible, errores de escritura)
- La mayoría retorna un número de error o código cuando la E/S falla.
- Logs de errores

Con respecto a las formas de realizar I/O

- Bloqueante: El proceso se suspende hasta que el requerimiento de I/O se completa
 - Fácil de usar y entender
 - No es suficiente bajo algunas necesidades??
- No Bloqueante: El requerimiento de I/O retorna en cuanto es posible. El proceso tiene la capacidad de seguir ejecutandose más allá de la finalización o no del requerimiento.
 - Generalmente, es la utilizada. Dificulta la programación al programador.
 - Ejemplo: Interfaz de usuario que recibe input desde el teclado/mouse y se muestra screen.
 - Ejemplo: Aplicación de video que lee frames desde un archivo mientras va mostrandolo en pantalla.

Diseño del software → ¿Como pensaron los diseñadores del SO el software para satisfacer todas estas cuestiones?

Un diseño de programación en capas donde cada capa tiene su objetivo específico y cada capa oculta su información a las demás.



Software de E/S de capa de usuario

Esta capa es modo usuario.

En el modo usuario, los sistemas operativos suelen dejar un conjunto de librerías de funciones para que los procesos utilicen que no requieren la ejecución en modo kernel.

- Librerías de funciones
 - Permiten acceso a SysCalls: Un proceso no realiza llamadas al sistema por su cuenta, utiliza librerías.
 - Implementan servicios que no dependen del Kernel
- Procesos de apoyo
 - Demonio de Impresión (realiza el spooling): Proceso de apoyo a E/S que mantiene una cola de los requerimientos de impresión y le va diciendo al SO imprime esto, luego esto. Esta cola la maneja un proceso en modo usuario

Software de sistema operativo independiente del dispositivo

Es la capa controladora de la E/S en modo kernel.

Brinda los principales servicios de E/S antes vistos a la capa superior:

- Interfaz uniforme
- Manejo de errores
- Buffer
- Asignación de Recursos
- Planificación

Esta capa es independiente de los dispositivos. Se puede dividir en subcapas para lograr un diseño más modular y lograr que sea más fácil aplicar cambios, mejoras, etc.

- El Kernel mantiene la información de estado de cada dispositivo o componente
 - Archivos abiertos
 - Conexiones de red
 - Etc.
- Hay varias estructuras complejas que representan buffers, utilización de la memoria, disco, etc.

Cada capa tiene un conjunto de estructuras de datos que permiten modelar y abstraer.

Controladores de dispositivos (Drivers)

Esta capa esta más cercana al hardware ya que los drivers realmente saben como usar los dispositivos. Cuando la capa superior que es independiente quiere leer un disco no sabe como hacerlo, lo delega a la capa mas inferior que son los drivers.

- Contienen el código dependiente del dispositivo
- Manejan un tipo dispositivo
- Traducen los requerimientos abstractos en los comandos para el dispositivo. Por ejemplo, al driver le llega un read sabe que comandos emitir de E/S
 - Escribe sobre los registros del controlador
 - Acceso a la memoria mapeada
 - Encola requerimientos
- Comúnmente las interrupciones generadas por los dispositivos son atendidas por funciones provistas por el drive.
 - El driver contiene una rutina que se ejecuta cuando un dispositivo genera una interrupcion, delega la resolucion de dicha interrupcion en un metodo/codigo que esté implementado en el driver ya que es el que sabe como atender dicho dispositivo
 - La resolucion de la interrupcion no esta dada por el sistema operativo porque si asi fuera el SO deberia saber responder a todas las interrupciones de cientos de dispositivos, impracticable.
- Interfaz entre el SO y el HARDW. Sabe entender lo que el sistema operativo le pide y sabe hablar con el hardware.
- Forman parte del espacio de memoria del Kernel.
 - Se ejecutan dentro del kernel. Debe estar bien programado ya que es un software programado por terceros, no por los que crearon el SO.
 - En general se cargan como módulos. Recien cuando se utiliza el dispositivo se carga/levanta el modulo correspondiente, no estan todos cargados, se evita un megakernel. Se cargan dinámicamente.

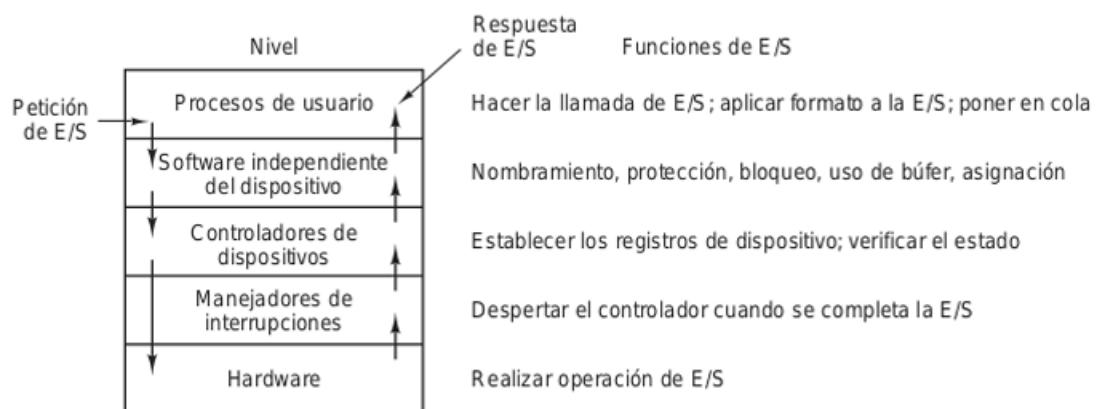
- Los fabricantes de HW implementan el driver en función de una API especificada por el SO
 - open(), close(), read(), write(), etc
 - Por ejemplo, implementar un driver para linux habrá que leer y averiguar que funciones requiere e implementarlas. Lo mismo para windows.
- Para agregar nuevo HW sólo basta indicar el driver correspondiente sin necesidad de cambios en el Kernel

Gestor de interrupciones

Es parte de la programación del kernel, ante una interrupción la capa la capta y resguarda información de registros, etc. y llama a la rutina de atención de interrupciones correspondiente al tipo de dispositivo que la genero que basicamente está en el driver.

- Atiende todas las interrupciones del HW
- Deriva al driver correspondiente según interrupción
- Resguarda información
- Independiente del Driver

Ciclo de atención de un Requerimiento

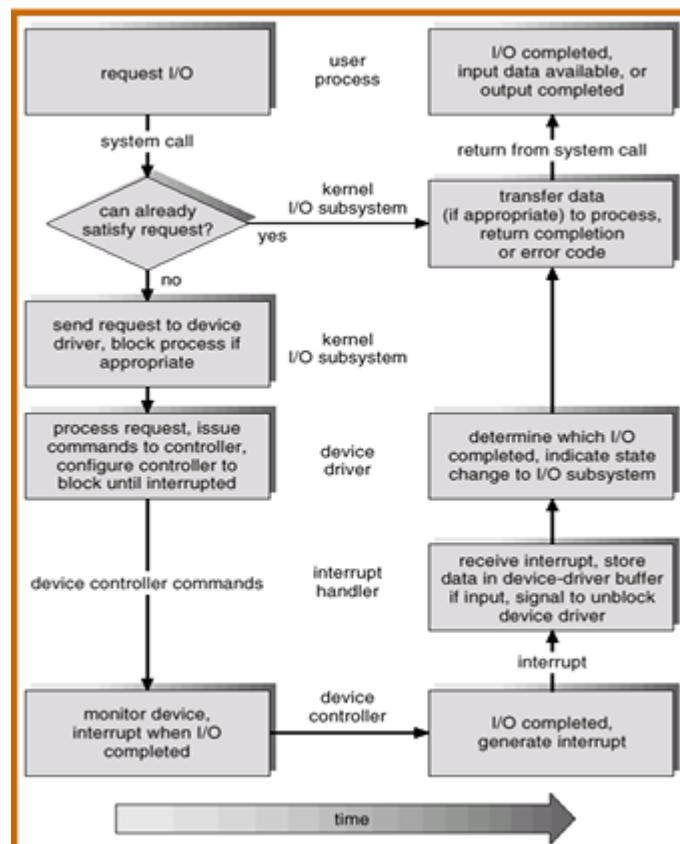


A partir de una petición de E/S, el requerimiento va bajando por la estructura de capas, a medida que baja es menos abstracto, hasta que se llega al hardware (dispositivo de E/S).

Cuando el hardware termina de realizar la operación de E/S, genera nuevamente una interrupcion que sera derivada al driver para luego devolverle el control a la capa de software independiente que luego le dará el control al usuario.

Ejemplo: Consideremos la lectura sobre un archivo en un disco:

- Determinar el dispositivo que almacena los datos
 - Traducir el nombre del archivo en la representación del dispositivo.
- Traducir requerimiento abstracto en bloques de disco (Filesystem)
- Realizar la lectura física de los datos (bloques) en la memoria
- Marcar los datos como disponibles al proceso que realizo el requerimiento
 - Desbloquearlo
- Retornar el control al proceso



Performance

Hay que tener en cuenta que I/O es uno de los factores que mas afectan a la performance del sistema:

- Utiliza mucho la CPU para ejecutar los drivers y el codigo del subsistema de I/O
- Provoca Context switches ante las interrupciones y bloqueos de los procesos
- Utiliza el bus de mem. en copia de datos:
 - Aplicaciones (espacio usuario) – Kernel
 - Kernel (memoria fisica) - Controladora

Para mejorar dicha performance:

- Reducir el número de context switches, es decir, tratar de resolver la E/S en el mismo proceso que está en ejecución (sin importar a quien corresponda)
 - Por ejemplo, si se está ejecutando el proceso 2 y se ocurre una interrupcion que representa el fin de E/S del proceso 1, el proceso 2 puede hacer lo necesario para no tener que volver al proceso 1. En cualquier proceso se puede resolver cualquier interrupcion sin importar el origen.
- Reducir la cantidad de copias de los datos mientras se pasan del dispositivo a la aplicación
 - A través que se pasa por las etapas tratar de no estar copiando datos de memoria de un lado a otro.
- Reducir la frecuencia de las interrupciones, utilizando:
 - Transferencias de gran cantidad de datos. Por ejemplo, si el dispositivo cuenta con una cache, se puede pasar una gran cantidad de datos para evitar que produzca interrupciones pidiendo datos varias veces.
 - Controladoras mas inteligentes
 - Polling, si se minimiza la espera activa.
- Utilizar DMA
 - De esta manera, no se usa la CPU para manejar las interrupciones y el DMA resuelve.

13-Arquivos 1

¿Por qué necesitamos archivos?

Los archivos existen para resolver tres necesidades clave:

- **Almacenar grandes cantidades de datos**
- Tener almacenamiento a largo plazo (**persistente**), es decir, que los datos permanezcan guardados incluso después de apagar el sistema.
- Permitir a **distintos procesos acceder al mismo conjunto** de información

¿Qué es un archivo?

Un archivo es una **entidad abstracta con nombre** que nos permite almacenar contenido dentro de ella.

Características:

- Se ve como un **espacio lógico continuo y direccionable**: desde el punto de vista lógico, un archivo se puede ver como una sucesión de bytes que se pueden dirigir, aunque físicamente puedan estar en distintos lugares del disco.
- Provee datos a los programas (entrada)
- Permite a los programas guardar datos (salida)
- El propio programa ejecutable es en sí información (un archivo) que debe guardarse

Punto de vista del usuario

El usuario interactúa con el archivo sin preocuparse de cómo se almacena físicamente en el disco y cómo se traduce su espacio lógico en posiciones reales de almacenamiento. Lo que le importa es:

- Qué operaciones se pueden llevar a cabo (abrir, leer, escribir, etc.)
- Cómo nombrar a un archivo (nombre, extensiones, etc.)
- Protección y permisos: quién puede leer, escribir o ejecutar el archivo.
- Cómo compartir archivos (cómo otros procesos o usuarios pueden acceder a él), etc.

Punto de vista del diseño

Aquí el interés está en **cómo implementar internamente el sistema de archivos** para que sea eficiente y seguro. Esto incluye:

- **Implementación de archivos:** estructuras de datos para representarlos y asociarlos a su contenido físico en disco.
- **Implementación de directorios.**
- **Manejo del espacio en disco:** cómo se distribuye y se organiza la información en bloques o sectores.
- **Manejo del espacio libre:** cómo saber qué bloques están libres y cuáles ocupados.

El diseño busca:

- **Eficiencia:** acceso rápido y sin desperdiciar recursos.
- **Mantenimiento sencillo:** que las estructuras de datos no sean costosas de actualizar.
- **Persistencia:** que la información y sus metadatos sobrevivan a reinicios o apagados.

Sistema de Manejo de Archivos

El **sistema de manejo de archivos** es el **conjunto de unidades de software** del sistema operativo que proveen los servicios necesarios **para la utilización de archivos**, para su creación, eliminación, búsqueda, copia, lectura, escritura y manipulación de archivos.

- Estas unidades **consumen CPU**, por lo que deben ser **eficientes**: las operaciones sobre archivos no deberían tardar demasiado, ya que la CPU está pensada principalmente para ejecutar procesos y no para quedarse bloqueada en tareas de administración de archivos.
- Facilitan el acceso a los archivos por parte de aplicaciones y procesos a través de **operaciones de alto nivel** como `open` , `read` , `write` , `close` , etc., **abstrayendo al programador** de las complejidades del acceso físico y de bajo nivel al almacenamiento.
- El programador **no necesita** implementar el software que administra archivos; esa tarea la realiza el **sistema de archivos** del SO.



Estructuralmente, el sistema de archivos se ubica:

- **Por encima del driver del dispositivo**, que gestiona el hardware (el dispositivo).
- **Por debajo del coordinador de E/S**, que organiza las operaciones de entrada/salida de manera general.

Objetivos del SO en cuanto a archivos

El sistema operativo, respecto al manejo de archivos, busca:

- Cumplir con la gestión de datos: organizar, guardar y recuperar información de forma eficiente.
- Atender las solicitudes del usuario
- Minimizar o eliminar la posibilidad de perder o destruir datos
 - Garantizar la integridad del contenido de los archivos, es decir, asegurarse de que los datos guardados no se pierdan, corrompan o queden inconsistentes.
 - Por ejemplo, ante un error como corte de energía, las estructuras de datos del sistema de archivos sigan siendo consistentes.
- Dar soporte de E/S a distintos dispositivos: discos duros, SSD, memorias USB, etc.
- Brindar un conjunto de interfaces de E/S para el tratamiento de archivos.

Tipos de Archivos

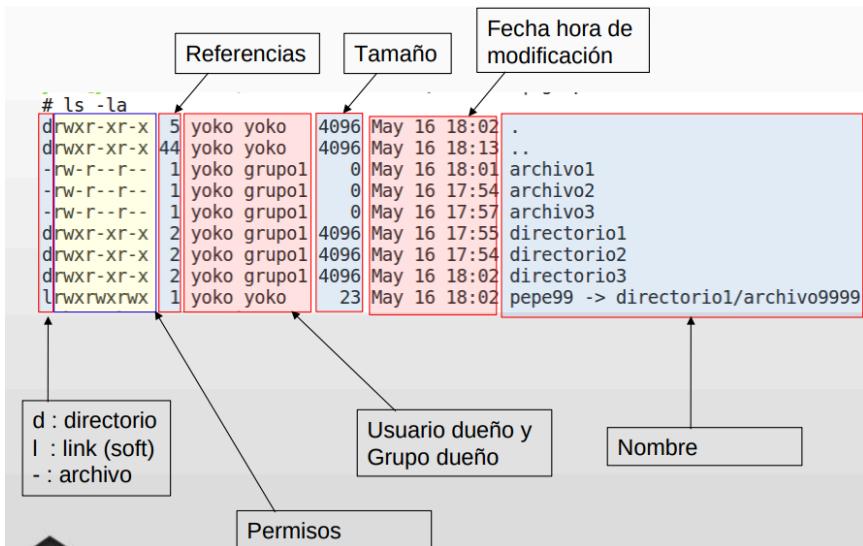
- **Archivos Regulares:** Son los archivos más comunes, contienen datos o programas.
 - Texto Plano: Contienen únicamente caracteres, por ejemplo, archivos de código fuente (source File)

- Binarios: Contienen datos codificados en formato no legible para humanos, por ejemplo, Object File o Executable File.
- **Directorios:** Archivos que mantienen la estructura en el FileSystem, cuyo contenido son relaciones (referencias) a otros archivos o, incluso, a otros directorios.

Atributos de un Archivo

Cada archivo tiene una serie de atributos que permiten identificarlo:

- Nombre
- Identificador
- Tipo: define la naturaleza del archivo (regular, directorio, enlace/link, etc.)
- Localización: es toda la información necesaria para ubicar el contenido del archivo en el medio de almacenamiento.
- Tamaño
 - Lógico: cantidad real de bytes que contiene
 - Físico: espacio que ocupa en el medio de almacenamiento (puede ser mayor al lógico)
- Protección, Seguridad y Monitoreo
 - Owner (propietario)
 - Permisos: quien puede leer, escribir o ejecutar el archivo.
 - Password
 - Fechas: Momento en que el usuario lo modificó, creó, accedió por última vez
 - ACLs (Access Control Lists): listas que especifican permisos para distintos usuarios o grupos.



Directarios

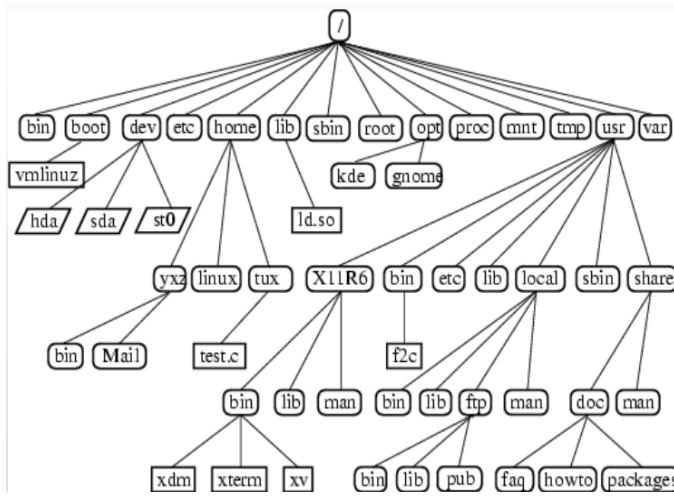
El directorio es, en sí mismo, un archivo que no contiene datos de usuario, si no que contiene información sobre otros archivos y directorios que están dentro de él.

- Los directorios intervienen y cumplen un rol en la resolución de nombres:
 - Cuando un proceso intenta acceder a un archivo, el sistema operativo busca en la estructura de directorios la correspondencia entre el nombre del archivo y su ubicación real en disco.
 - Restricción: dentro de un mismo directorio no puede haber dos archivos con el mismo nombre. Pero sí pueden existir en directorios distintos.
- Operaciones que el File System debe implementar sobre directorios:
 - Buscar un archivo, crear un archivo (agregar una nueva entrada en el directorio), borrar un archivo, listar el contenido, renombrar archivos, Etc.

Ventajas del uso de directarios

- Eficiencia: permiten una localización rápida de archivos.
- Uso del mismo nombre de archivo: diferentes usuarios pueden tener archivos con el mismo nombre, ya que su path completo los diferencia.
- Agrupación lógica: permiten organizar archivos según su función o características, por ejemplo, agrupar por un lado programas Java, Juegos, Librerías, etc.

Estructura de directorios



Los sistemas operativos suelen usar una estructura **jerárquica de árbol**, donde:

- Los archivos pueden ubicarse a través de un camino completo (**pathname o path absoluto**) que parte desde el directorio raíz (/) y recorre los directorios intermedios hasta llegar al archivo.
- Distintos archivos pueden tener el mismo nombre pero el **full pathname es único**

Directorio de trabajo (working directory)

Cada proceso tiene un **directorio de trabajo** que es el **directorio actual** (lo que en linux/unix se consulta con pwd)

Dentro del directorio de trabajo, se pueden referenciar archivos de dos formas:

- **Path absoluto:** indicando toda la ruta desde la raíz.
- **Path relativo:** indicando la ruta desde el directorio actual.

Identificación absoluta y relativa

Tanto los archivos como los directorios se pueden identificar dentro del sistema de archivos mediante la:

Identificación absoluta: Indica todo el camino completo del archivo desde la raíz del sistema de archivos.

- Ejemplo: /var/www/index.html o C:\windows\winhelp.exe

Identificación relativa: Depende del directorio en el que se esté trabajando actualmente.

- Por ejemplo, si estoy en el directorio /var/spool/mail/ entonces es:
..../www/index.html

Compartir archivos

En un ambiente **multiusuario** se necesita que varios usuarios puedan compartir archivos. Esto debe ser realizado bajo un **esquema de protección**, que garantice tanto la seguridad como la integridad del contenido.

- **Derechos de acceso:** Determinan a quién quiero compartir el archivo y qué permisos va a tener.
 - Execution: el usuario puede ejecutar
 - Reading: el usuario puede leer el archivo
 - Appending: el usuario puede agregar datos pero no modificar o borrar el contenido del archivo
 - Updating: el usuario puede modificar, borrar y agregar datos. Incluye la creación de archivos, sobreescibirlo y remover datos.
 - Changing protection: el usuario puede modificar los derechos de acceso.
 - Deletion: el usuario puede borrar el archivo.
 - Owners (propietarios): Tiene todos los derechos y puede dar derechos a otros usuarios.
- **Manejo de accesos simultáneos:** Es necesario **controlar el acceso concurrente** para evitar inconsistencias (por ejemplo, que dos usuarios editen al mismo tiempo y se pierdan cambios).

Protección

El **propietario/administrador** debe ser capaz de controlar:

- Qué se puede hacer
 - Definiendo los derechos de acceso.
 - Cabe aclarar que **los directorios también tienen permisos**, y estos determinan si el usuario puede o no acceder al contenido que hay dentro. Por ejemplo, un archivo puede ser legible, pero si el usuario no tiene permiso de acceso al directorio donde está, no podrá abrirlo.
- Quien lo puede hacer

- Especificando qué usuarios o grupos tienen acceso y qué tipo de acciones pueden realizar sobre los archivos y directorios.
- Esto asegura que cada recurso solo sea utilizado por los usuarios autorizados.

14-Archivos 02

Metas del Sistema de Archivos

- Brindar espacio en disco a los archivos de usuario y del sistema.
- Mantener un registro del espacio libre. Cantidad y ubicación del mismo dentro del disco.

Conceptos

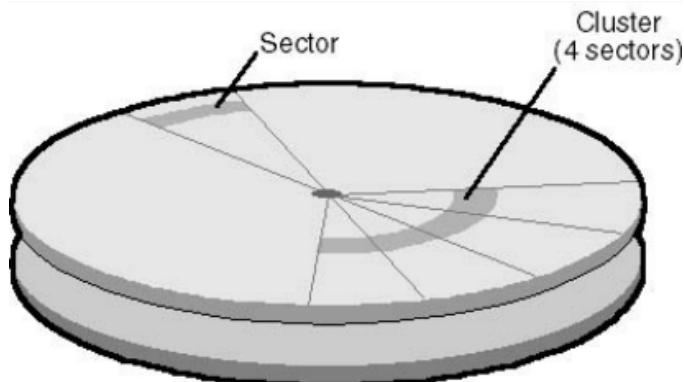
File System → Define la forma en que los datos son almacenados, es decir, como se representan los archivos para ser almacenados.

Sector → Unidad de almacenamiento utilizada en los Discos Rígidos

Bloque/Cluster → Conjuntos de sectores consecutivos que el sistema de archivos maneja como unidad.

FAT(File Allocation Table) → Contiene información sobre en que lugar están alojados los distintos archivos.

- Es como una tabla de punteros que indica qué clusters pertenecen a un archivo.



Se verán **formas (y estructuras)** en las que el archivo lógico se representan en el medio de almacenamiento, es decir, como el espacio continuo de un archivo

se distribuye en el almacenamiento.

Estas estructuras tendrán referencias hacia sectores o clusters. La diferencia es que si, por ejemplo, se quiere almacenar un archivo más pesado, conviene buscar un conjunto de sectores (un cluster) en lugar de ir almacenandolos en varios sectores.

Para asignar espacio a un archivo, existen dos técnicas →

Pre-asignación

Es una técnica que se utiliza para planificar **cúanto espacio ocupa un archivo**, si sabemos que los archivos no cambian en tamaño.

- Se necesita saber cuantos sectores va a ocupar el archivo en el momento de su creación.
- Se tiende a **definir espacios mucho más grandes que lo necesario** → desperdicio potencial de espacio.
- Posibilidad de utilizar **sectores contiguos** para almacenar los datos de un archivo → acceso más rápido.
- ¿Qué pasa cuando el archivo supera el espacio asignado? Puede que no haya espacio contiguo para ampliarlo y deba reubiscese o encadenarse con bloques adicionales → pérdida de eficiencia.

Asignación Dinámica

El espacio se pide a medida que se necesita, es decir, a medida que un archivo crece, se van obteniendo nuevos bloques/cluster que puede quedar no contiguos.

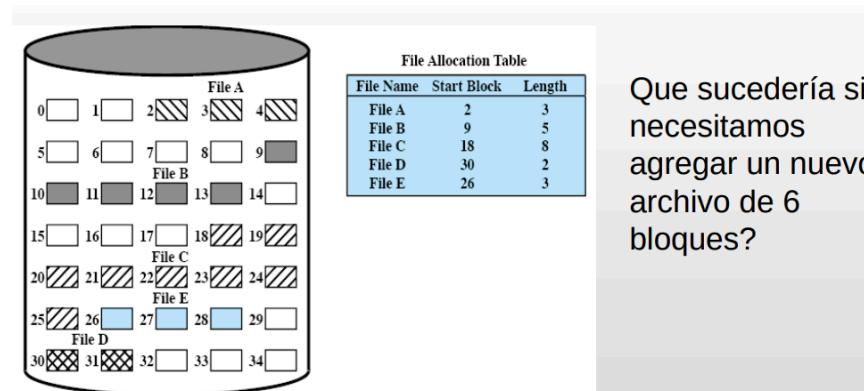
- Los bloques de datos no necesariamente son contiguos → mayor flexibilidad.
- Ventaja: se ajusta al crecimiento del archivo, no se desperdicia espacio.

Formas de asignación

Continua →

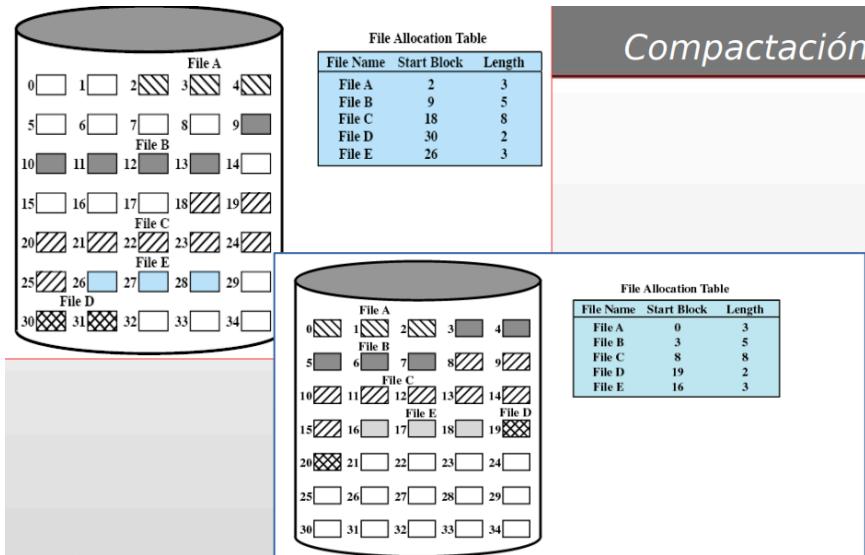
Los sectores o clusters que representan al contenido de un archivo, se almacenan de forma continua en el medio de almacenamiento.

- Hay un conjunto **continuo** de bloques que son utilizados
- Se requiere una **pre-asignación**: Se debe conocer el tamaño del archivo durante su creación



Rta: Si hay que guardar un archivo de 6 bloques pero no hay 6 sectores libres consecutivos, no se podria almacenar porque no están juntos si no se encuentran dispersos.

- La **File Allocation Table (FAT)** es simple con respecto a la info. que mantiene → Cada archivo tiene una sola entrada que incluye **bloque de inicio y longitud**
 - Por ejemplo: Archivo A → comienza en cluster 10 y ocupa 5 clusters → la FAT guarda solo "inicio=10, longitud=5"..
- El archivo puede ser leído con **una única operación**
- Puede existir el problema de la **fragmentación externa**. Ocurre cuando hay bloques libres, pero no de manera contigua.



Desventajas:

- Encontrar bloques libres continuos en el disco: **fragmentación externa**.
- Incremento del **tamaño de un archivo**: Si el archivo es muy grande requiere mas espacios contiguos

Solución → Compactación: lograr que los bloques libres queden todos en un mismo lugar. Es muy costoso ya que implica reubicar muchos archivos en el disco (super costoso mover sectores/clusters de un lugar a otro).

Encadenada →

Es una técnica que **quita la necesidad de continuidad**. En lugar de exigir que un archivo se guarde en bloques continuos, permite que un archivo esté disperso en diferentes lugares siempre que se mantenga un enlace (puntero) entre los bloques que lo componen.

- El archivo se divide en bloques. La asignación se realiza en base a bloques individuales
- **Enlaza/encadena los bloques** que pertenecen a un archivo. Cada bloque, además de tener sus datos, tiene un puntero al próximo bloque del archivo.

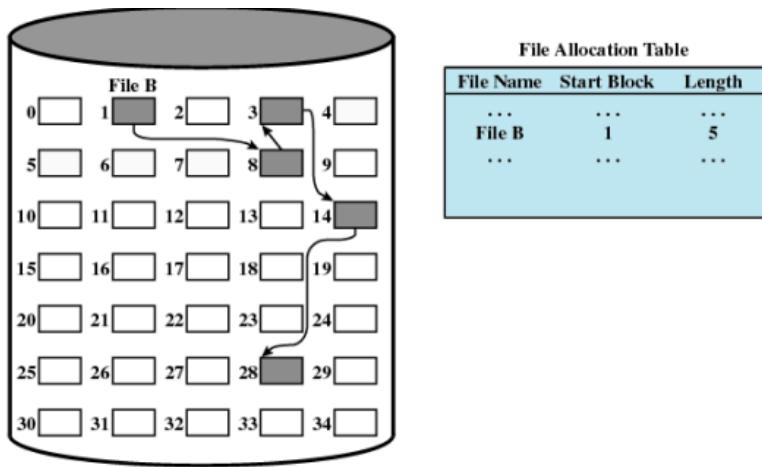


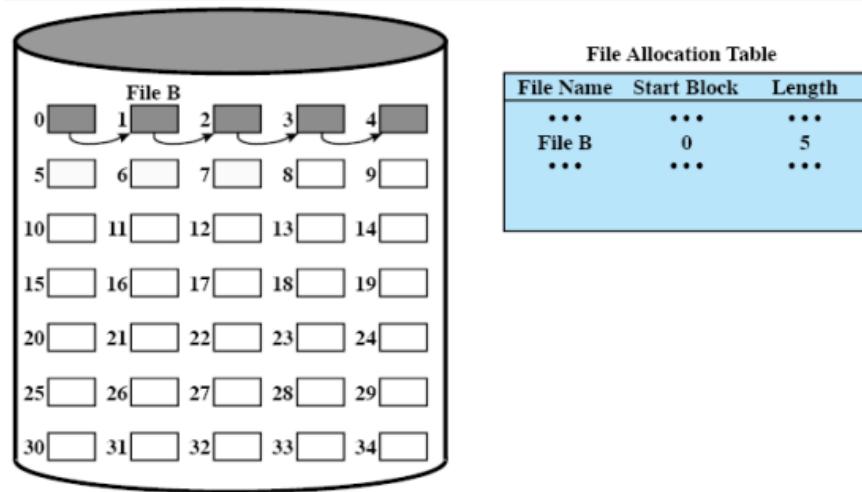
Figure 12.9 Chained Allocation

- La **File allocation table (FAT)** → Cada archivo tiene una sola entrada que incluye **bloque de inicio y tamaño del archivo**
- **No hay fragmentación externa**
- **Útil para acceso secuencial** (no random): Leer de principio a fin es fácil porque se siguen los enlaces.
 - **Desventaja:** Acceso aleatorio lento. Si se quisiera ir a una dirección concreta, se debe recorrer los bloques anteriores (si es que los hay).
- Los **archivos pueden crecer** bajo demanda: Alcanza con asignar un nuevo bloque y enlazarlo.
- No se requieren bloques contiguos.

Optimización:

Se pueden **consolidar los bloques de un mismo archivo (defragmentación)** para garantizar cercanía de los bloques de un mismo archivo.

Basicamente, se mueven para que estén cerca físicamente en el disco. Esto reduce el movimiento del cabezal y hace que leer el archivo sea más rápido, logrando una mejor performance del disco.



Observación: La estructura FAT se almacena en el disco, sin embargo, esa información que almacena no se usa directamente desde el disco, si no que se carga en memoria RAM para poder ser procesada más rápido.

Indexada →

En lugar de encadenar bloques o reservarlos contiguos, existe un bloque especial de índices que guarda los punteros a todos los bloques de datos que forma el archivo.

- La **File allocation table (FAT)** → Contiene una única entrada con la dirección del bloque índice (no contiene datos del archivo, contiene las referencias a los bloques donde sí están los datos). Es decir, la entrada almacena el numero del bloque que contiene las direcciones que apuntan a los datos que contiene el archivo.

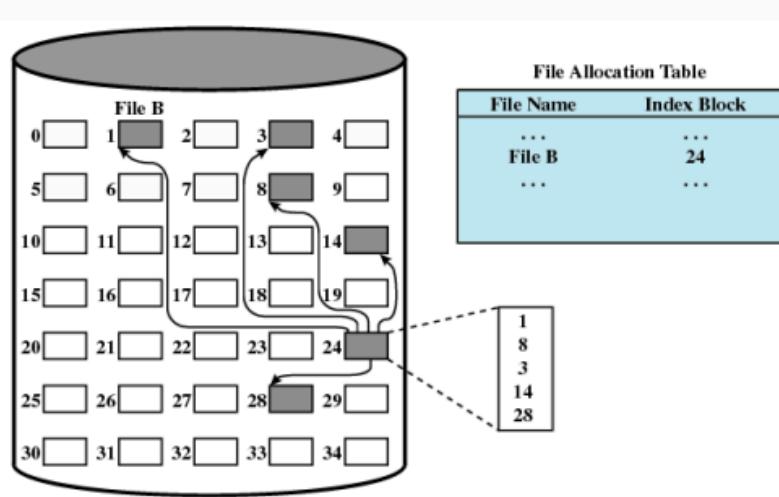


Figure 12.11 Indexed Allocation with Block Portions

- Asignación en base a bloques individuales (no hace falta continuidad)
- **No se produce Fragmentación Externa** (ya que cualquier bloque libre puede usarse)
- El acceso “random” a un archivo es eficiente: Para llegar a un bloque específico se calcula directamente el índice en la tabla (se puede calcular con la regla de 3 simples).

Desventaja:

- El tamaño del archivo está limitado por la cantidad de entradas que pueda almacenar la estructura de índices.

Variantes de la forma de asignacion indexada:

Asignacion por secciones

Esta variante **busca achicar la tabla de indices**. Se aprovecha que muchas veces los bloques de un archivo se guardan contiguos en el disco y, en lugar de tener una entrada para cada bloque, hay una sola entrada que indica a partir desde donde y hasta cuando se puede leer los bloques que pertenecen a ese mismo archivo.

Así, **cada entrada en el bloque índice** tiene:

- Dirección del primer bloque de un conjunto almacenado de manera contigua

- Longitud

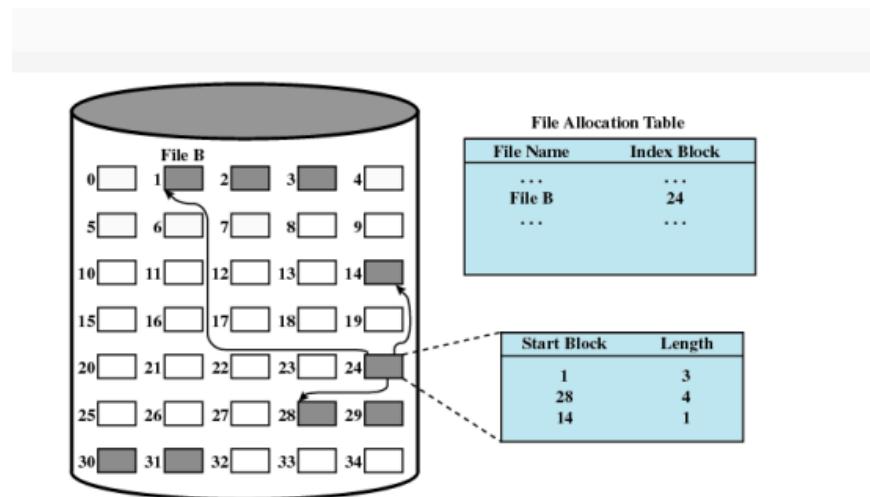


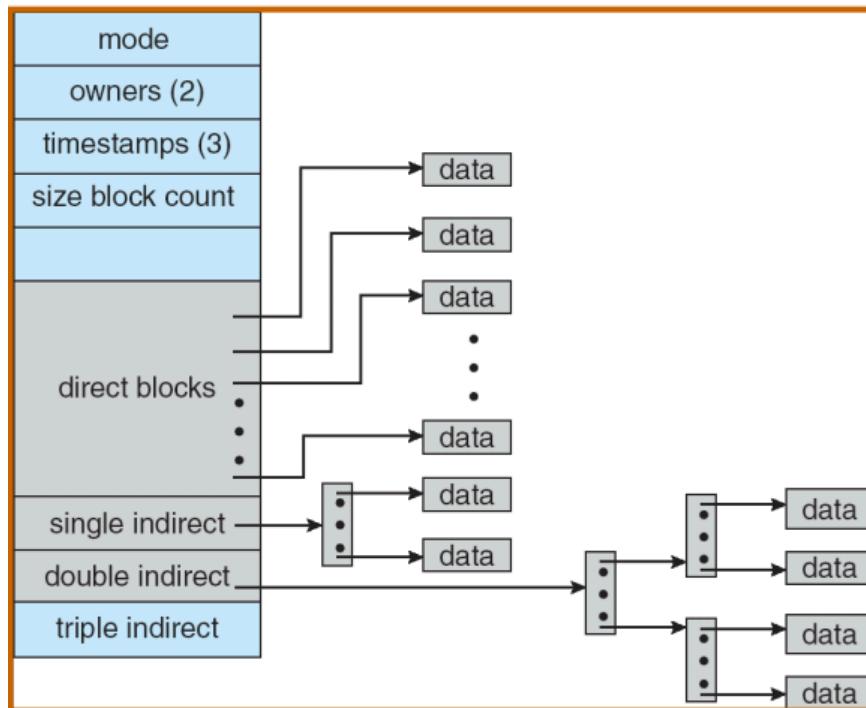
Figure 12.12 Indexed Allocation with Variable-Length Portions

Niveles de indirección

Existen **bloques que apuntan directamente a datos** y también **hay bloques índices que apuntan a varios bloques de datos**.

Puede haber **varios niveles de indirección**:

- **Directos** → apuntan a datos.
- **Indirectos simples** → apuntan a un bloque que contiene direcciones de datos.
- **Indirectos dobles** → apuntan a un bloque que contiene direcciones de bloques índice, que a su vez apuntan a datos.
- **Indirectos triples** → un nivel más... (muy raro, pero existe en algunos FS).



Cada I-NODO contiene 9 direcciones a los bloques de datos, organizadas de la siguiente manera:

- ◆ 7 de direccionamiento directo.
- ◆ 1 de direccionamiento indirecto simple
- ◆ 1 de direccionamiento indirecto doble

Si cada bloque es de 1KB y cada dirección usada para referenciar un bloque es de 32 bits:

- ✓ ¿Cuántas referencias (direcciones) a bloque pueden contener un bloque de disco?

$$1 \text{ KB} / 32 \text{ bits} = 256 \text{ direcciones}$$

- ✓ ¿Cuál sería el tamaño máximo de un archivo?

$$(7 + 256 + 256^2) * 1 \text{ KB} = 65799 \text{ KB} = 64,25 \text{ MB}$$

Gestión de espacio libre

El sistema de archivos necesita tener un control sobre cuáles de los bloques de disco están disponibles y cuáles están ocupados.

Alternativas:

Tablas de bits

Es una tabla (vector) donde **cada bloque de disco está representado por 1 bit**.

Cada entrada:

- 0 → bloque libre
- 1 → bloque en uso

Ejemplo

00111

00001

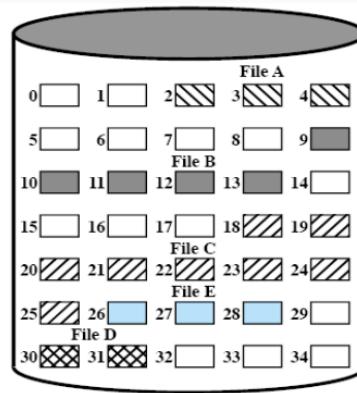
11110

00011

11111

11110

11000



Ventaja

- Facil encontrar un bloque o grupo de bloques libres: Como los bits están almacenados **de manera consecutiva**, se puede recorrer el vector para encontrar fácilmente un bloque libre o un grupo de bloques contiguos.

Desventaja

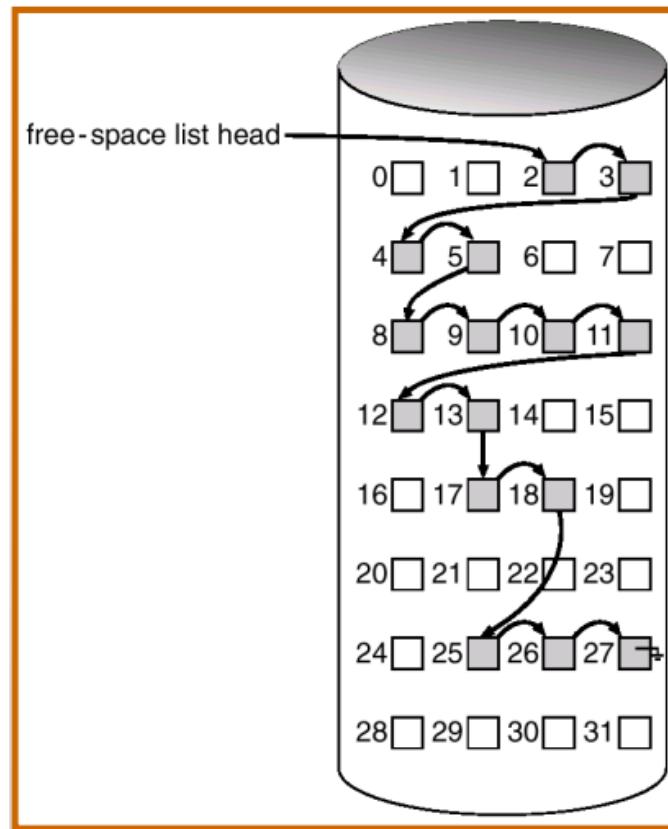
- La tabla debe estar almacenada en memoria para poder usarse. Si su tamaño es muy grande, debe cargarse entera en memoria para buscar bloques libres, aunque uno solo sea necesario.
 - Ejemplo: Un disco de 16 Gb con bloques de 512 bytes → la tabla sola ocupa unos 23MB.
- Si se corta la luz mientras se está actualizando, puede quedar inconsistente (un bloque marcado como libre aunque está ocupado, o al revés).

Bloques libres encadenados

Se tiene **un puntero al primer bloque libre**. Cada bloque libre tiene un puntero al siguiente, formando una **lista enlazada**.

- Ineficiente para la búsqueda de bloques libres → Hay que realizar varias operaciones de E/S para obtener un grupo libre y implica mucho consumo de CPU.
- Problemas con la pérdida de un enlace: Si se pierde un enlace no se sabe después en donde continua el proximo bloque libre.

- Difícil encontrar bloques libres consecutivos: Como están encadenados, los libres pueden quedar dispersos en disco, entonces casi nunca hay un grupo seguido de bloques libres.



Es simple de implementar y ahorra espacio en memoria, pero es lento y muy vulnerable a errores de enlace.

Indexación (o agrupamiento)

Es una mejora sobre “bloques libres encadenados”.

La idea es que, en lugar de que cada bloque libre apunte solamente al siguiente, **el primer bloque libre almacena muchas direcciones de bloques libres**.

- El primer bloque libre contiene las direcciones de N bloques libres
- Los primeros **N-1 son bloques libres utilizables**.
- La **última dirección (N)** apunta a otro bloque índice con más direcciones de bloques libres.

Ventajas:

- Se encuentra más rápido un conjunto de bloques libres (se busca en el índice en memoria, no recorriendo uno por uno).
 - Por ejemplo, si se necesitan 30 bloques libres, en lugar de seguir 30 punteros uno por uno generando muchas operaciones E/S, solo se acceden a los bloques índice. Si hay dos bloques índice, uno con N=10 y otro con N=25, se toman los primeros 9 índices del primero y los restantes del segundo bloque.
- Reduce la cantidad de operaciones de E/S

5-Filesystem 3

Manejo de archivos

En UNIX, están los siguientes tipos de archivos:

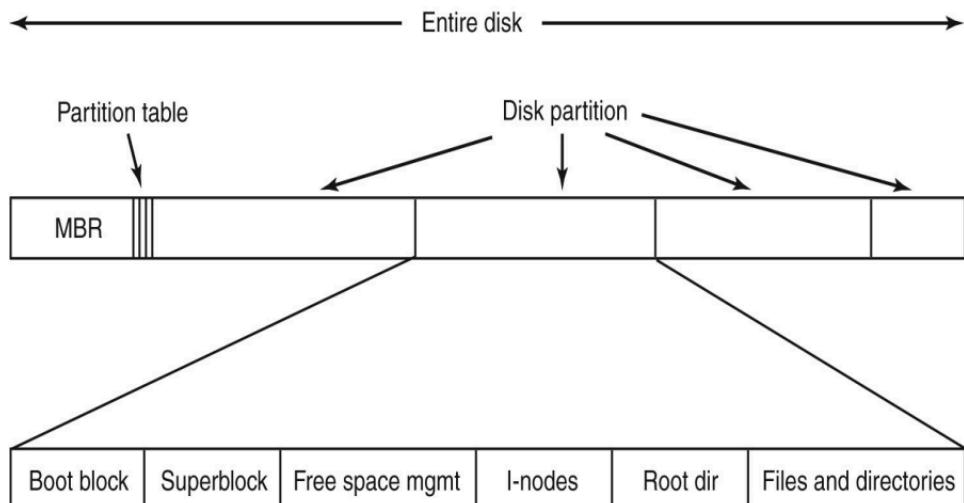
- Archivo común
- Directorio
- Archivos especiales (dispositivos /dev/sda)
- Named pipes (comunicación entre procesos)
- Links (comparten el i-nodo, solo dentro del mismo filesystem)
- Links simbólicos (tiene i-nodo propio, para filesystems diferentes)

Estructura del volumen

¿Como es un sistema de archivos en Unix?

Un **disco físico** puede ser dividido en uno o mas volúmenes (**particiones**).

Cada volumen o partición contiene su propio sistema de archivos.



Un **sistema de archivos en unix es una división lógica donde hay:**

Boot Block → código para iniciar el sistema operativo, hay información de arranque.

Superblock → describe al filesystem, contiene sus atributos (tamaño, bloques libres, ubicación de estructuras).

Tabla de i-nodos → tabla que contiene todos los i-nodos.

- **I-NODO:** Estructura de control que contiene la información clave de un archivo

Data Blocks → bloques de datos reales de los archivos.

Directorio raíz → primer directorio del filesystem, donde empiezan todas las rutas.

Archivos y directorios →

I-nodo

Es una **estructura de datos** que **mantiene los atributos** (como tamaño, propietario, permisos, etc. todos menos el nombre) **de un archivo**.

Información del i-nodo:

File Mode	16-bit flag that stores access and execution permissions associated with the file.
	12-14 File type (regular, directory, character or block special, FIFO pipe)
	9-11 Execution flags
	8 Owner read permission
	7 Owner write permission
	6 Owner execute permission
	5 Group read permission
	4 Group write permission
	3 Group execute permission
	2 Other read permission
	1 Other write permission
	0 Other execute permission
Link Count	Number of directory references to this inode
Owner ID	Individual owner of file
Group ID	Group owner associated with this file
File Size	Number of bytes in file
File Addresses	39 bytes of address information
Last Accessed	Time of last file access
Last Modified	Time of last file modification
Inode Modified	Time of last inode modification

¿Qué pasa al formatear?

Cuando se formatea un disco con un sistema de archivos tipo UNIX:

- Se crea de entrada una **tabla fija de i-nodos**.
- Eso significa que el número de i-nodos es limitado y se reserva desde el principio. Los i-nodos no aparecen y desaparecen dinámicamente.
- Por eso, aunque tengas espacio libre en disco, podés quedarte sin i-nodos y no poder crear más archivos.

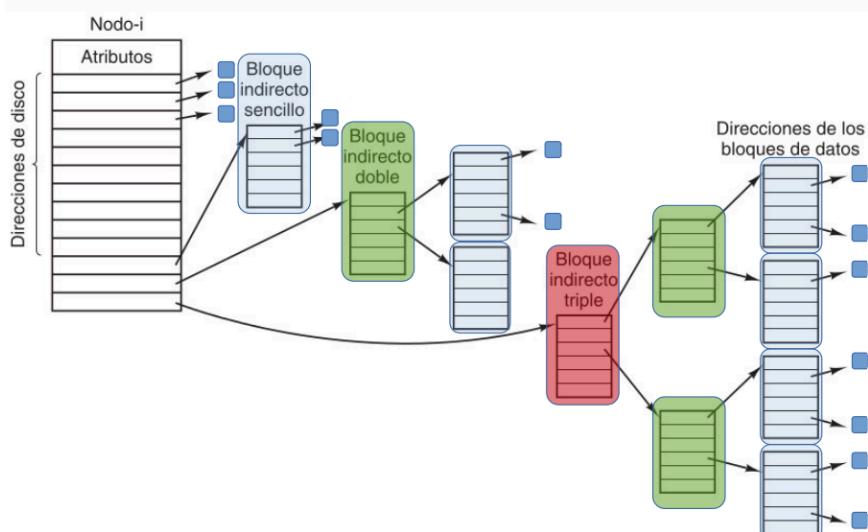
¿Como se ubican los datos de un archivo?

El i-nodo tiene una serie de punteros:

- Punteros directos: señalan directamente un bloque de datos.
- Puntero de indirección simple: el puntero no va a un bloque de datos, si no a un bloque de punteros, ese bloque extra contiene muchas direcciones de bloques de datos → permite ampliar el tamaño de archivo.
- Puntero de doble indirección: El puntero señala un bloque, que a su vez apunta a otros bloques de punteros, y recién ahí se llega a los datos →

sirve para manejar archivos aún más grandes.

- Puntero de triple indirección → con un nivel más que el anterior.



Explicación:

- Cuando el archivo se crea, se reserva un i-nodo (sin datos) libre de la tabla de i-nodos.
 - Ese i-nodo contiene metadatos iniciales (dueño, permisos, fechas, etc.). Todavía no tiene bloques de datos asignados (porque el archivo está vacío).
- A medida que se empiezan a cargar datos en el archivo y comienza a crecer, se asigna el primer bloque de datos de disco.
 - El i-nodo se guarda un puntero al primer bloque de datos.
- El archivo crece y los bloques directos que tiene el i-nodo ya no alcanzan, se reserva un bloque especial para guardar direcciones de otros bloques (bloque de indirección simple).
 - Si aún no alcanza, se usan dobles y triples niveles de indirección.

Observación → El i-nodo se encuentra en el sistema de archivos en disco, la memoria RAM entra en juego solo cuando se accede al archivo y el SO debe leer los bloques del disco (usando las direcciones del i-nodo) y los copia a memoria

UNIX-Directorios

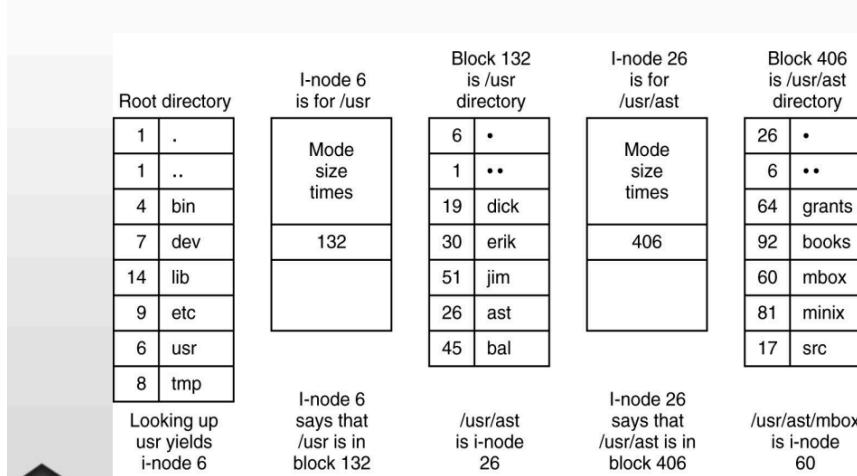
¿Qué es un directorio en UNIX?

Es una tabla que **relaciona un nombre (string) y un número de i-nodo**.

Recordar que el nombre del archivo no se encuentra almacenado en el i-nodo, si no en esta tabla (directorio).

- Se puede tener dos archivos en el filesystem que apunten al mismo i-nodo.
 - Ejemplo: `/usr/ast/mbox` y `/tmp/copia_mbox` podrían apuntar ambos al i-nodo 60. Son el mismo archivo físico, solo con nombres distintos en directorios.

Buscar el i-nodo del archivo `/usr/ast/mbox`



¿Cómo se resuelve el path?

- El directorio raíz (`/`) tiene entradas que contiene un numero que representa un i-nodo y un string que representa el nombre.
 - En la tabla del root se busca el nombre `"usr"` que corresponde al **i-nodo 6**.
- Una vez que se tiene el i-nodo que representa a la primera "parte" de nuestro path, se buca dentro de él el siguiente bloque.
 - Vamos al **i-nodo 6**, que representa el directorio `/usr`. Ese i-nodo dice que su contenido está en el **bloque 132**. Ahí se busca `"ast"` y se ve que corresponde al **i-nodo 26**.
 - Ahora se abre el **i-nodo 26** (que es el de `ast`). Ese i-nodo apunta al **bloque 406**, que contiene las entradas de ese directorio. En ese bloque se busca `"mbox"` y se ve que corresponde al **i-nodo 60**. El i-nodo 60 ya no es un directorio, sino un **archivo común**.

15 buffer cache

Disk Cache

Es un **mecanismo que implementa el SO** que tiene como objetivo **reducir el acceso al disco**.

Son un **conjunto de buffers en memoria principal que mantienen temporalmente bloques de disco**.

- 📌 Idea: Si un proceso pide un bloque de disco, primero se busca en la caché
 - Si está, se evita ir al disco (trabaja mucho más lento que la RAM).
 - Si no está, se lee del disco, se guarda en la caché y después se da el bloque al proceso

¿Cómo acceden los procesos a un bloque de caché? Hay dos alternativas.

- **Se copia el bloque desde la caché del SO hacia el espacio de direcciones del usuario**
 - Problema: No se puede compartir el bloque entre procesos porque se copió en la memoria privada del proceso que lo solicitó.
- Se trabaja como memoria compartida
 - El bloque se mantiene en un área común de caché (compartida)
 - Varios procesos pueden acceder al mismo bloque sin duplicarlo.

📌 Observación clave

- La **caché es limitada** (no se puede guardar todo el disco en memoria).
- Por eso, el SO necesita un **algoritmo de reemplazo** para decidir **qué bloque sacar cuando la caché está llena**.

Estrategia de reemplazo

Cuando la **caché de buffers** (espacio en memoria donde se guardan temporalmente bloques de disco) está llena y se necesita cargar un nuevo bloque, hay que decidir **qué bloque sacar**. Esa decisión la define la **estrategia de reemplazo**.

Entre las estrategias de reemplazo más utilizadas se encuentran:

Alternativa: (LRU, Least Recently Used) → Se elige el que hace más tiempo que no es referenciado.

- Es una lista de bloques, donde el último es el más recientemente usado. El primero es el más viejo (el menos usado recientemente).
- Cuando un bloque se referencia o usa, queda al final de la lista.

Otra alternativa: Least Frequently Used → Se reemplaza el que tenga menor número de referencias. En lugar de mirar **cuándo fue la última vez que se usó**, se mira **cuántas veces se usó en total**.

Importante: **No se mueven los bloques en la memoria**: No se copian los datos en memoria cada vez que se referencia a un bloque de disco, **lo que cambia es la estructura de punteros que mantiene la lista**.

👉 Después de explicar las políticas de reemplazo en general, lo siguiente es mostrar **cómo Unix System V** implementa en detalle este manejo de la caché de buffers.

UNIX SYSTEM V- BUFFER CACHÉ

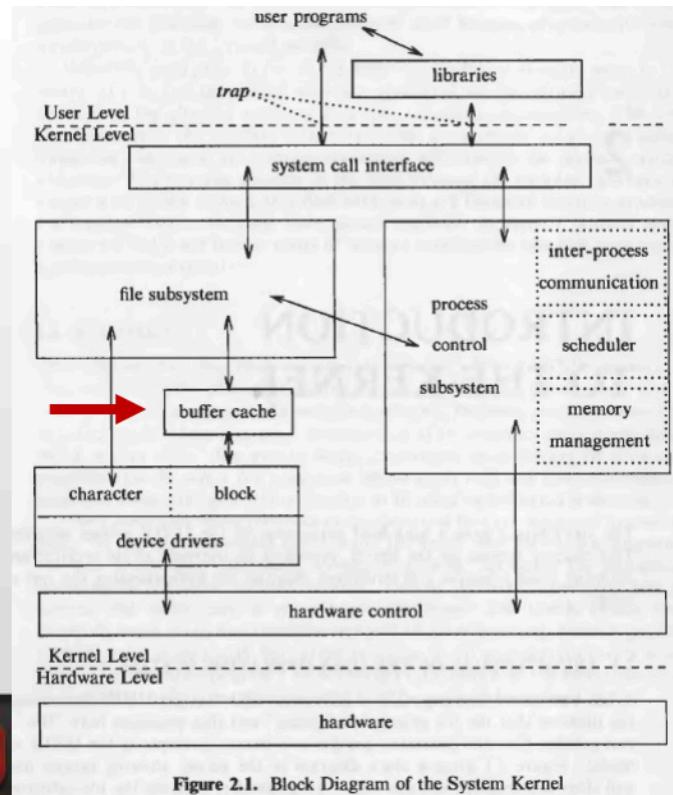
Objetivo → Minimizar la frecuencia de acceso a disco.

Es una **estructura formada por buffers**. El kernel reserva espacio en la memoria durante la inicialización para esta estructura.

Cada **buffer tiene dos partes**:

- **Header**: Es la metadata del buffer. Contiene información del bloque, numero del bloque, estado, relación con otros buffers, etc.
- **El buffer en sí**: el espacio real en RAM donde se almacena el contenido de un bloque de disco.

- El módulo de buffer cache es independiente del sistema de archivos y de los dispositivos de hardware
- Es un servicio del SO



¿Puede funcionar un sistema de archivos sin buffer cache? Si.

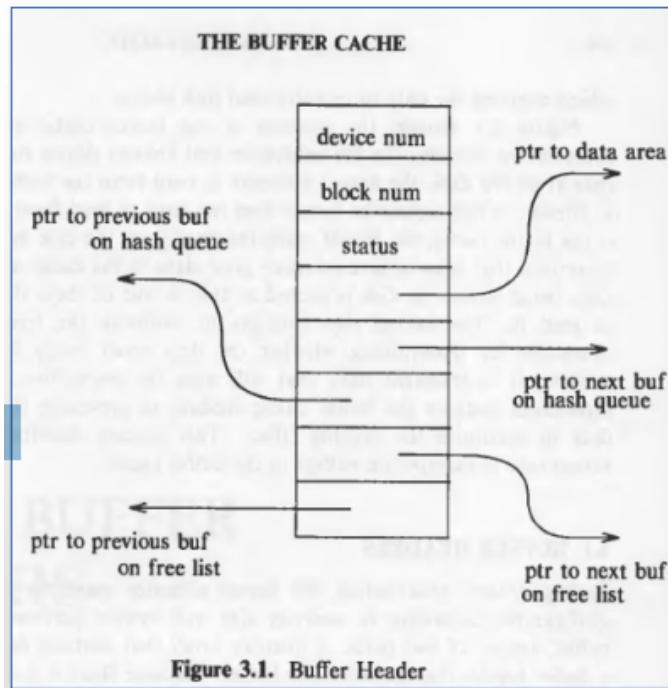
El header

El header es fundamental, porque el kernel no trabaja directamente con los datos del bloque, si no que gestiona todo a través del header.

Cada header indica:

- **El numero de dispositivo:** Como en unix todo dispositivo (disco, una partición, etc) se representa con un device ID, permite diferenciar de qué dispositivo físico o partición viene el bloque
- **El numero de bloque dentro de ese dispositivo:** Una vez que sabemos de qué disco/partición viene, hay que saber qué bloque exacto es el que está en cache.
- **Un estado.**
- Contiene **punteros** que permiten organizar todos los buffers de la caché y permitirle al kernel gestionar la caché como un sistema de listas.
 - 2 punteros para la hash queue
 - 2 punteros para la free list

- 1 puntero al bloque en memoria RAM: apunta al espacio en RAM donde está almacenado el contenido del bloque de disco.



Estados de los buffers

- **Free o disponible:** El buffer no está siendo usado por ningún proceso.
 - Puede significar que está vacío ya que todavía no se utilizó o puede ser que un proceso lo utilizó y lo libero.
- **Busy o no disponible:** El buffer si está en uso por algun proceso.
- **Se está escribiendo o leyendo del disco:** Como las operaciones de disco son **asíncronas** (se disparan y tardan un tiempo en completarse), se marca el buffer para indicar que **la operación aún no terminó**.
- **Delayed write (DW):** El buffer fue modificado en memoria, pero los cambios no han sido reflejados en el bloque original en disco.

Free List

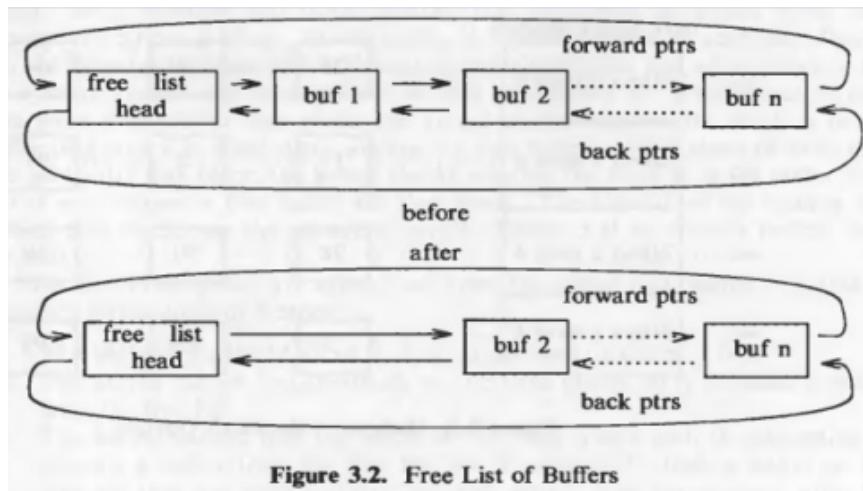
Es una **lista enlazada** que mantiene los **headers** de los buffers que están en **estado “disponible”**. Esta lista organiza los buffers disponibles para ser utilizados para cargar nuevos bloques de disco.

Aclaración: disponible puede significar que el buffer nunca fue usado (está vacío) o que fue utilizado, liberado por un proceso, pero ya no está en uso. Ojo: puede contener información vieja que quizás siga necesitando, o incluso estar en estado **Delayed Write** (datos modificados en memoria pero aún no escritos en disco).

¿Cómo se organiza?

La lista enlazada se organiza según **LRU (least recent used)**

- Cada vez que se usa un buffer (se lee/escribe un bloque), su **header se mueve al final** de la lista → "es el más recientemente usado".
- Cuando se necesita un buffer nuevo y no hay libres, la **victima** será uno del **principio de la lista** (el "least recently used").



Problema: Si tenés un **montón de bloques cargados en memoria**, y cada uno tiene un *buffer header*, la búsqueda de "¿está el bloque X del disco Y en memoria?" sería lenta si tuvieras que recorrer **toda la lista de headers** uno por uno (como una lista enlazada).

Hash Queues

Como puede haber un montón de bloques cargados en memoria, se utiliza una **estructura mas óptima para la búsqueda de un buffer en particular**.

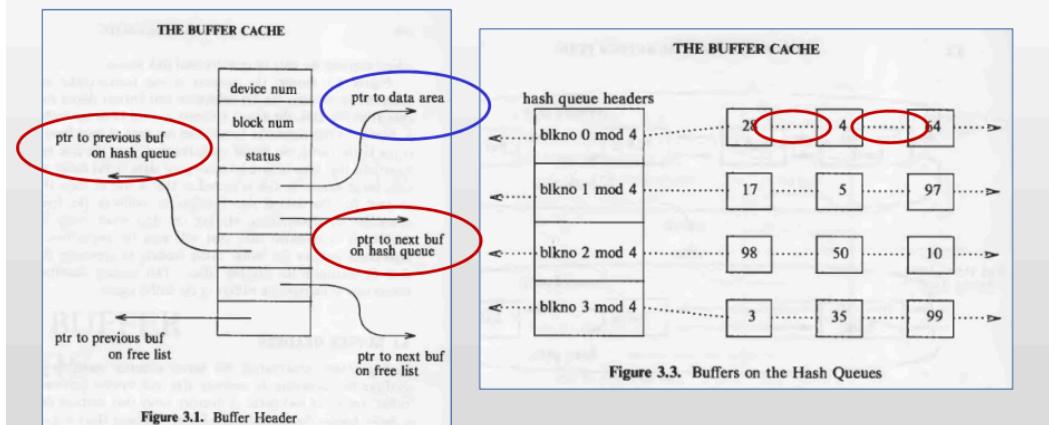
En lugar de una sola lista, **se usa un conjunto de listas (colas)**.

- Los headers de los buffers **se organizan según una función de hash** usando como clave el par **(dispositivo, #bloque)**

- A ese par se le aplica una función de hash, devuelve un número entre 0 y N-1 (siendo N la cantidad de hash queues disponibles). Ese resultado te dice en qué cola tenés que buscar.
- Se agrupan los buffers cuyo resultado, luego de aplicar la función hash, dio igual para hacer las busquedas más eficientes.
- Se busca que la función de hash provea alta dispersión para lograr que las colas de bloques no sean tan extensas, es decir, que los valores (dispositivo,#bloque) se repartan lo más uniformemente posible en las diferentes colas.

¿Cuantas hash queues? Habrá tantas como posibles valores que da la función hash. Más hash queues → más eficiente podría resultar la búsqueda para saber si un bloque está o no en buffer cache ya que cada cola es más pequeña.

Para agrupar los bloques se utilizan los punteros que anteriormente habíamos visto que se almacenaban en el header



Ejemplo concreto: Tenés 4 hash queues ($N=4$ → índices 0,1,2,3). La función hash: $h(\text{dev}, \text{bloque}) = (\text{dev} + \text{bloque}) \bmod 4$.

Si se busca el bloque ($\text{dev}=2$, $\text{bloque}=10$), al aplicar la función hash $h(2,10) = (2+10) \bmod 4 = 12 \bmod 4 = 0$, se guarda/busca en la cola 0.

Ventaja: En vez de recorrer toda la lista global de headers, solo buscás en una cola mucho más corta (las que tengan el mismo hash), lo que termina en una búsqueda más rápida

Observaciones:

- Free List sigue el mismo esquema de la Hash queue (también es una lista enlazada de headers) pero contiene los headers de los buffers de aquellos procesos que ya han terminado.
 - La hash queue organiza todos los buffers que están en uso o ya han sido cargados y los agrupa según una función hash.
- El header de un buffer siempre está en la Hash Queue
 - Aunque el buffer esté libre o ocupado, **siempre aparece en alguna Hash Queue**, porque la Hash Queue es la estructura que permite encontrar rápidamente a qué bloque de disco corresponde.
- Si ya ningún proceso utiliza un bloque que estaba siendo referenciado, va a estar en la Hash Queue y en la Free List
 - el header nunca deja la Hash Queue (allí indica qué bloque contiene o contenía ese buffer) pero se **agrega además a la Free List**, para marcarlo como "candidato a reutilizar".

Funcionamiento del buffer cache

→ **El proceso pide un archivo**

El sistema de archivos **identifica el archivo por su inodo** (recordar que el inodo guarda su metadata, incluyendo el tamaño, permisos, punteros a bloques de datos en disco) **ya que ayuda a localizar los bloques de datos donde se encuentra éste.**

→ Antes de leer o escribir los bloques de disco para satisfacer la operación del proceso, **se consulta al buffer caché**.

El kernel no accede al disco directamente, primero pregunta si el bloque que necesita está en el buffer caché. Entonces, el requerimiento llega al buffer cache quien evalúa si puede satisfacer el requerimiento o si debe realizar la E/S.

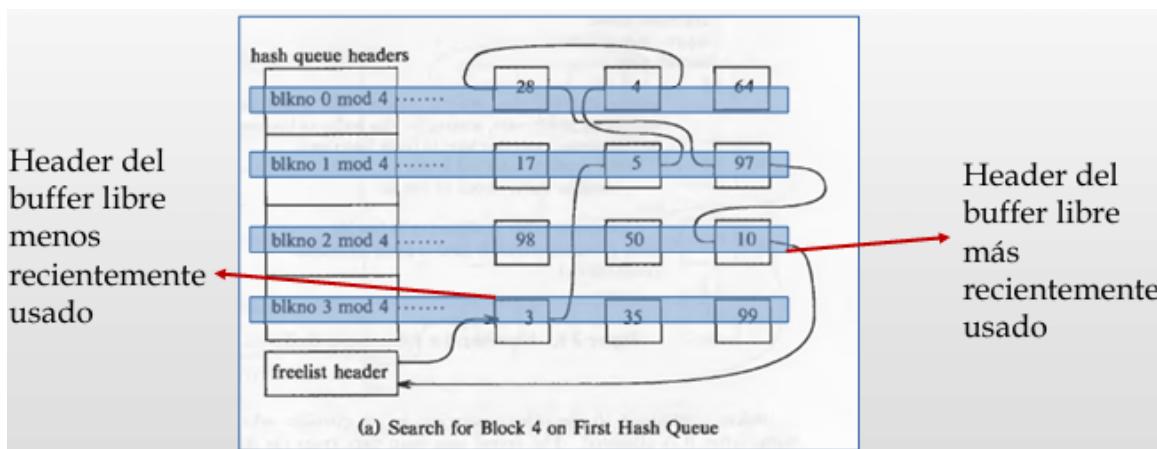
- Si está en el buffer caché, se usan los datos directamente
- Si no está en el buffer caché, hay que traer los bloques de datos desde el disco (E/S)

Observación: Lo importante es entender que el inodo no apunta al buffer caché, apunta a los bloques físicos en disco, el buffer caché actua como intermediario (evita acceder a disco si los bloques están cargados en memoria)

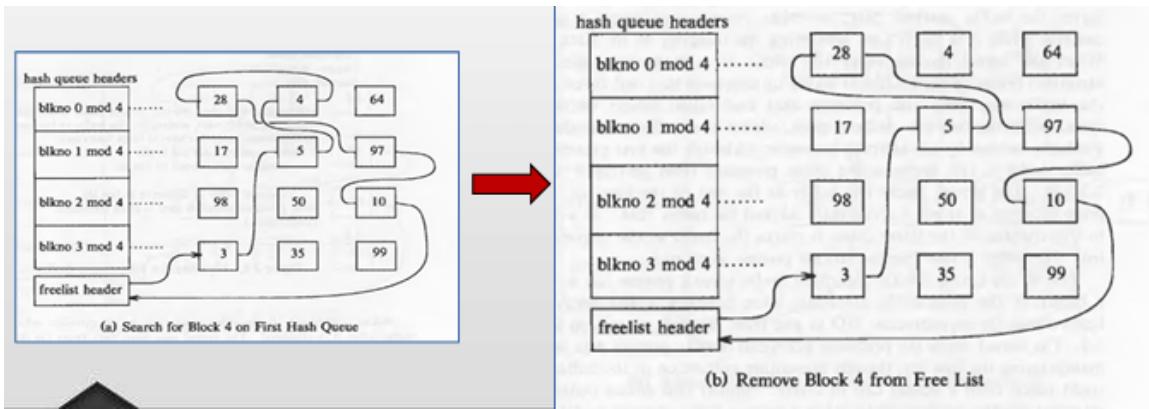
Se pueden dar 5 escenarios en la búsqueda/recuperación de un buffer:

1. El kernel encuentra el bloque en la hash queue y el buffer está libre (en un la free list).

- El proceso pide un bloque de disco, por ejemplo, el bloque 4.
- El kernel consulta el buffer cache antes de ir al disco
 - el kernel necesita saber en qué hash queue buscarlo dentro del buffer, para esto, aplica una función hash (número de bloque MOD cantidad de hash queues=índice)
 - En nuestro ejemplo, $4 \text{ MOD } 4 = 0$, el kernel busca en la hash queue 0, empieza a recorrer los enlaces.
- Se encuentra el bloque en memoria dentro de esa hash queue
- El header del buffer indica que está enlazado en la free list, eso significa que ningún proceso lo está usando, tiene estado disponible.

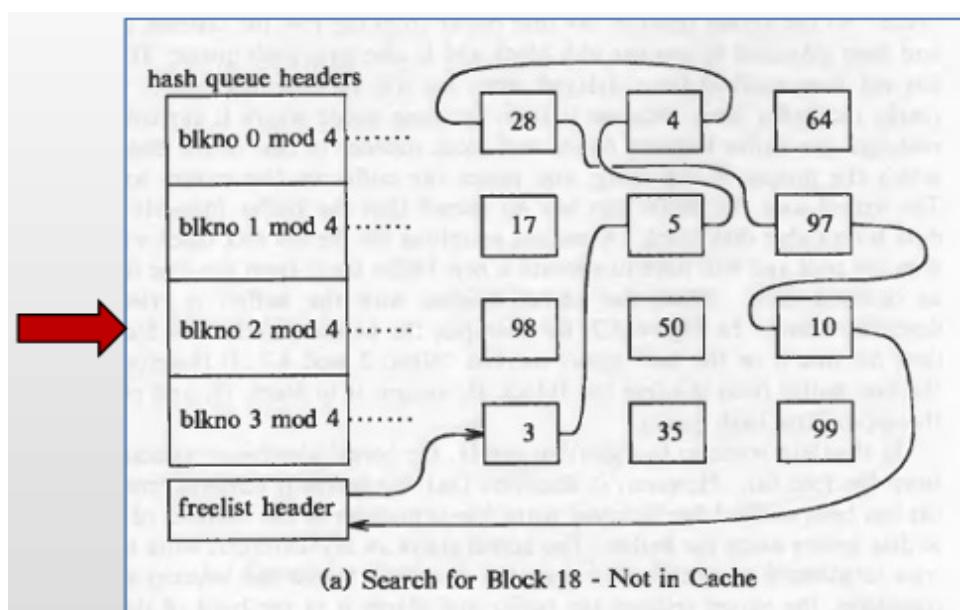


- el kernel remueve el bloque disponible de la free list y lo marca con estado ocupado (BUSY)
- El proceso ya puede usar el bloque 4 desde memoria, mucho más rápido que leer o escribir del disco.
- Una vez que lo termina de usar, podrá volver a la free list, por lo tanto, se deben reacomodar los punteros de la FreeList

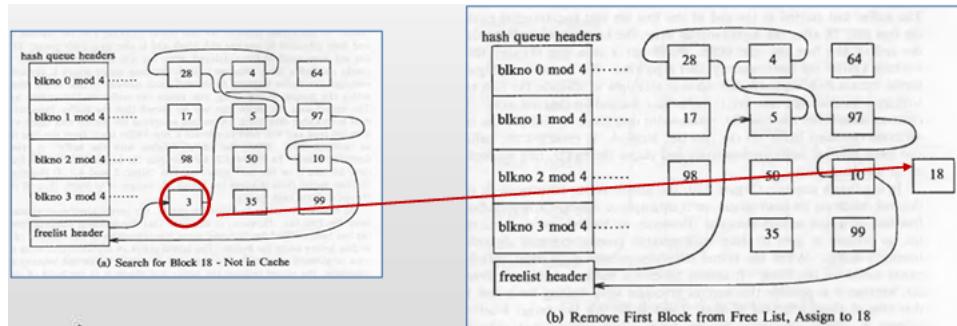


2. El kernel no encuentra el bloque en la hash queue y utiliza un buffer libre.

- El proceso pide un bloque, por ejemplo, el bloque 18.
- El kernel consulta el buffer cache antes de ir al disco.
 - Se aplica la función hash, se hace $18 \text{ MOD } 4$ da 2, se mete en la hash queue 2 y empieza a recorrer sus enlaces.
- El bloque buscado no se encuentra en la hash queue, el kernel necesita un espacio para cargar el bloque 18 desde el disco.
- El kernel busca un buffer libre en la free list, va a la free list, que está ordenada por política LRU, saca el primer buffer disponible (que será el menos utilizado)
- Se lee del disco el bloque deseado y se ubica en el buffer obtenido de la free list.

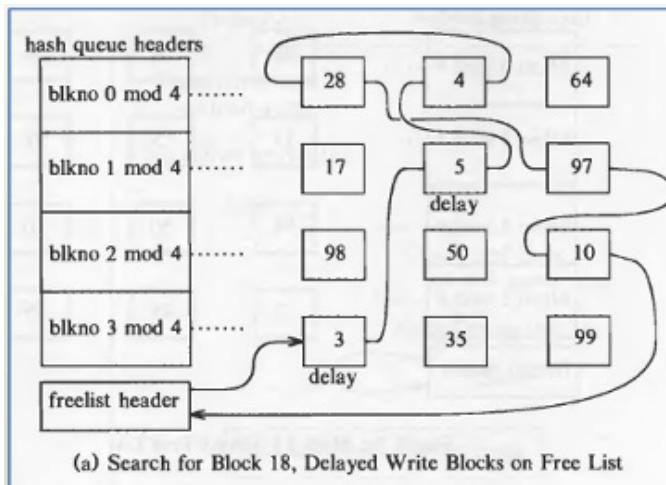


- Se actualizan las estructuras
 - El buffer obtenido deja de estar en la free list, pasa a estar ocupado (BUSY) y se inserta en la hash queue 2.
 - Importante: no se mueve el buffer en memoria, solo se actualizan los punteros de las listas para reflejar la nueva ubicación y estado.

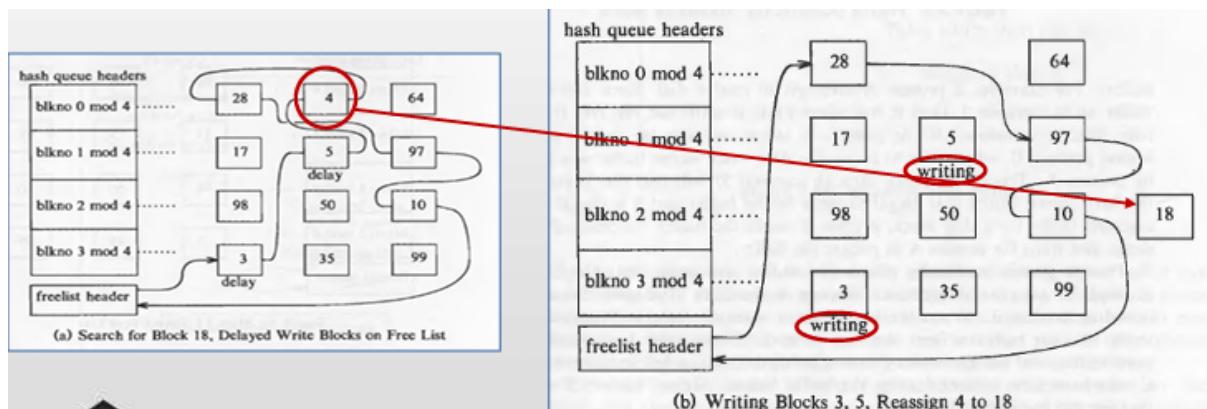


3. Idem 2, pero el bloque libre esta marcado como DW.

- El proceso pide un bloque, por ejemplo, el bloque 18.
- El kernel consulta el buffer cache antes de ir al disco.
 - Se aplica la función hash, se hace $18 \text{ MOD } 4$ da 2, se mete en la hash queue 2 y empieza a recorrer sus enlaces.
- El bloque buscado no se encuentra en la hash queue, el kernel necesita un espacio para cargar el bloque 18 desde el disco.
- El kernel busca un buffer libre en la free list, debe tomar el primero **pero esta marcado DW** (fue modificado pero todavía no se reflejaron los cambios en disco).
 - El kernel debe mandar a escribir a disco dicho bloque (en nuestro caso, el 3) y **tomar el siguiente buffer de la free list**
 - Si también está DW, sigue con el mismo proceso **hasta encontrar uno que no esté marcado como DW**.



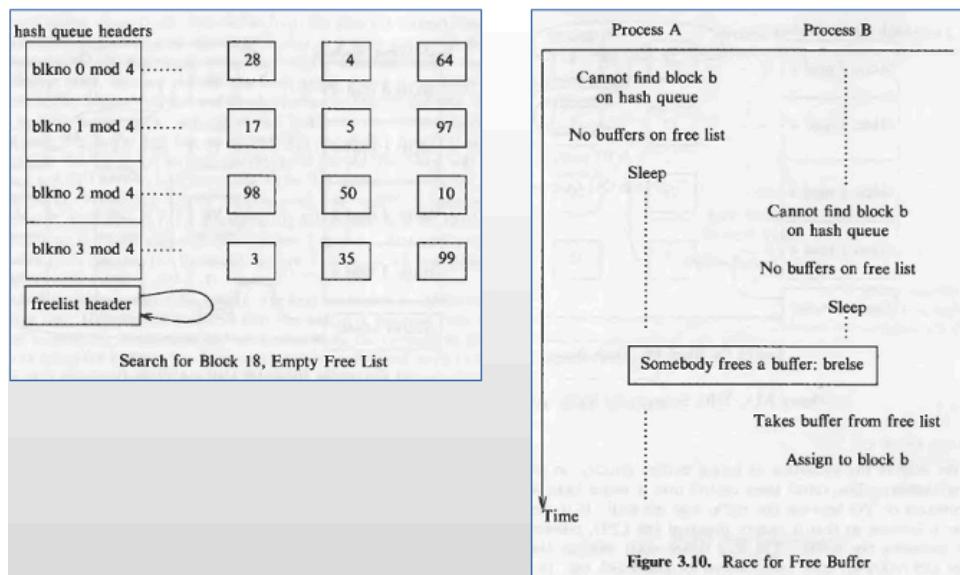
- Mientras los DW se escriben en disco, se asigna el siguiente buffer free al proceso → Una vez escritos a disco los bloques DW, se ubican nuevamente al principio de la free list.



4. El kernel no encuentra el bloque en la hash queue y la free list está vacía.

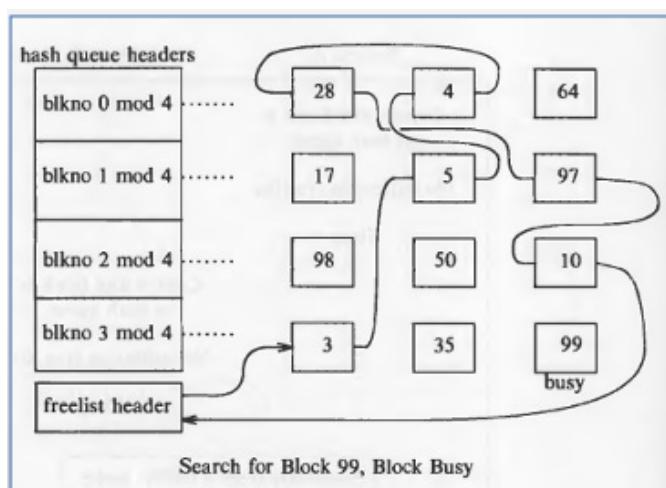
- El proceso pide un bloque.
- El kernel no encuentra el bloque en la hash queue y la free list está vacía
- El proceso **quedó bloqueado en espera** a que se libere algún buffer.
- Cuando el proceso despierta se debe verificar nuevamente que el bloque no esté en la hash queue (algún proceso pudo haberlo pedido mientras dormía)
 - es decir, mientras dormía pudo pasar que otro proceso también pidio el mismo bloque y que si haya conseguido un buffer libre, por ende, leyó desde disco e inserto el bloque en dicha hash queue.

- si no se verificará, puede pasar que el proceso termine leyendo de nuevo el bloque desde disco innecesariamente.



5. El kernel encuentra el bloque en la hash queue pero está BUSY

- El proceso pide un bloque, por ejemplo, el bloque 99
- El kernel busca el bloque pero el buffer que lo contiene está marcado como BUSY.
- El proceso se bloquea a la espera de que el buffer se desbloquee



- Eventualmente el proceso que tenia el buffer 99 lo libera
 - Se despiertan todos los procesos que estaban en espera de algún buffer.

- Cada uno (incluyendo al que pidió el bloque 99), al despertar, debe volver a chequear la hash queue y la free list para ver si el bloque que quiere ya está libre, o todavía está ocupado, o si necesita tomar otro buffer.

¿Por qué se despiertan todos? Porque el kernel no sabe de antemano **qué bloque quiere cada proceso** ni cuál se liberó.