**Resumen de Sistemas Operativos**

*Basado en el Stallings 4ta y 5ta edición Español*

## Capítulo: Procesos

### Concepto de Proceso

Informalmente un proceso es un programa en ejecución. Es la unidad de ejecución más pequeña planificable. Un proceso está formado por 3 componentes:

- Un programa ejecutable.
- Datos asociados necesarios para el programa (variables, espacio de trabajo, buffers, etc.).
- Contexto de ejecución o estado del proceso (contenido en el PCB).

El contexto de ejecución o estado del proceso es el componente más esencial, ya que incluye toda la información que el SO necesita para administrar el proceso y que el procesador necesita para ejecutarlo correctamente.

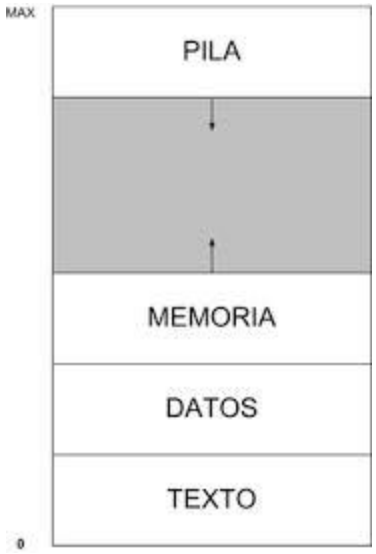
### Proceso vs Programa

- Proceso como entidad activa y dinámica (asignación dinámica de memoria)
- Programa entidad pasiva y estática.

Un programa se convierte en un proceso cuando se carga en memoria su archivo ejecutable.

La relación entre proceso y programa no es de uno a uno. Es decir dentro de un proceso pueden ejecutarse tanto un programa de usuario como uno del sistema operativo.

### Proceso en Memoria



- *Pila del proceso* → Datos temporales (Direcciones de retorno, parámetros de funciones, variables locales)
- *Memoria (HEAP)* → Cumulo de memoria. Es la memoria que se le asigna dinámicamente al proceso en tiempo de ejecución.
- *Sección de datos* → Variables Globales.
- *Sección de Texto* → Código del programa.

**Bloque de Control de Proceso (PCB)**

Cada proceso se representa en el SO mediante un bloque de control de proceso. Este contiene un conjunto de atributos utilizados por el SO para el control/administración del proceso.

La información contenida en el PCB se puede agrupar en 3 categorías:

1. **Identificación del proceso**
   Identificador del proceso. Identificador del proceso Padre. Identificador del usuario.

2. **Información de estado del procesador**
   Registros de CPU: contador del programa, códigos de condición (acarreo, signo, cero, desbordamiento, igualdad) e Información de estado (incluye habilitación/inhabilitación de interrupciones y modo de ejecución). Esta información de los registros debe guardarse, junto con el contador del programa, cada vez que se interrumpe el proceso para que pueda restaurarse cuando el proceso reanude su ejecución.

   *Nota*: PSW (Palabra de estado del programa). Conjunto de registros que contienen la información de estado.

   *Nota*: El contador del programa contiene la dirección de la siguiente instrucción que va a ejecutar el proceso.

3. **Información de control del proceso**
   *Información de planificación de CPU* (prioridad del proceso, etc.). *Información de gestión de memoria*. *Información contable* (uso de CPU, etc.). *Información del estado de E/S* (lista de archivos abiertos, dispositivos de E/S asignados al proceso, etc.).

El PCB es la estructura más importante del SO. El conjunto de PCBs define el estado del SO.
El PCB es parte de la imagen del proceso (Datos de usuarios, programa de usuario, pila del sistema y PCB). *La ubicación de la imagen del proceso depende del esquema de gestión de memoria utilizado.* En particular, en sistemas que utilizan memoria virtual, toda la imagen de un proceso activo está siempre en memoria secundaria. Cuando una imagen se carga en memoria principal, esta se copia en vez de moverse.

## Control de procesos

Se utilizan 2 modos de ejecución:

- Modo Usuario → No se permite la ejecución de instrucciones privilegiadas. El intento de ejecución de una instrucción privilegiada en este modo produce una excepción.

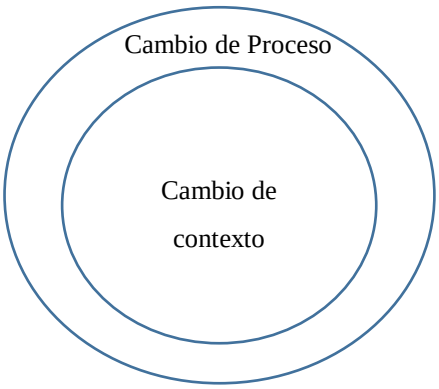
- Modo Kernel → Permite ejecución de instrucciones privilegiadas (llamadas al sistema, tratamiento de interrupciones, asignación de espacio de memoria a los procesos, etc.)

La razón por la que se usan estos 2 modos, se debe a que es necesario proteger al SO y a las estructuras del mismo, como los PCB, de las injerencias de los programas de usuarios.

*Nota:* ¿Cómo sabe el procesador en qué modo va a ejecutar? Hay un bit en el PSW (Palabra de estado del programa) que indica el modo de ejecución.
*Nota:* ¿Cómo se cambia de modo? El bit cambia como respuesta a ciertos sucesos. El control se encuentra inicialmente en manos del SO (modo Kernel). Cuando se da el control a una aplicación de usuario se pasa a modo usuario. El control finalmente se tiene que devolver al SO a través de una excepción o interrupción.

## Cambio de Proceso, Cambio de Contexto y Cambio de Modo

- Cambio de proceso → Cambio del uso de la CPU de un proceso de otro.
- Cambio de contexto → Cuando un proceso pasa de memoria a CPU o de CPU a memoria.
- Cambio de modo → Cuando se cambia de modo Kernel a modo usuario o viceversa.




Cada vez que se produce una interrupción o llamada al sistema, se debe guardar el estado del proceso que se está ejecutando para luego poder reanudar la ejecución. Se trata la interrupción o llamada al sistema y luego se carga nuevamente un PCB. Este puede ser el PCB del proceso que se estaba ejecutando u el PCB de otro proceso.

*Por lo tanto, un cambio de proceso, implica inevitablemente un cambio de contexto. En cambio, un cambio de contexto no implica siempre un cambio de proceso*, ya que se puede volver a cargar el proceso que se estaba ejecutando.

*Un cambio de contexto NO involucra siempre un cambio de modo.*

*Un cambio de modo NO involucra siempre un cambio de contexto.* Se puede cambiar el modo sin cambiar el estado del proceso (página 134 del Stallings).
Esto se debe a que se puede ejecutar casi todo el software del SO en el contexto de un programa de usuario (el SO se ejecuta dentro del proceso de usuario). Esto sirve para recordar que hay una distinción entre el concepto de programa y proceso y que la relación entre los 2 no es de uno a uno. Es decir dentro de un proceso pueden ejecutarse tanto un programa de usuario como uno del sistema operativo y los programas del SO que se ejecutan en los diferentes procesos de usuarios son idénticos.

- *Explicación de este enfoque*: Cuando se produce una interrupción, se salva el contexto de procesador y tiene lugar un cambio de modo hacia una rutina del SO. Sin embargo, la ejecución continúa dentro del proceso de usuario en curso. *De esta manera no se ha llevado a cabo un cambio de proceso (y por ende de contexto), sino un cambio de modo dentro del mismo proceso*. Debido al cambio de modo usuario a modo Kernel, el usuario no puede entrometerse ni estorbar en las rutinas del SO, aun cuando estas estén ejecutándose en el entorno de proceso de usuario. Cuando el SO termina su trabajo, determina que el proceso en curso debe continuar ejecutándose, entonces un cambio de modo continua el programa interrumpido del proceso en curso (aunque puede determinar cambiar el proceso).

  Ventaja de este enfoque: Un programa de usuario se interrumpe para emplear alguna rutina del SO, luego se reanuda y todo se produce SIN LA PENALIZACIÓN DE 2 CAMBIOS DE PROCESO *(sacar el proceso en ejecución, cargar la rutina del SO, sacar la rutina del SO, reanudar/ejecutar un proceso de usuario).*

Observación: Otro enfoque trata al SO como entidad separada que opera en modo privilegiado. El concepto de proceso se aplica solo a los programas de usuario.

*Nota:* ¿Cuándo se puede producir un cambio de proceso? En cualquier momento en el que el SO haya tomado el control a partir del proceso que está actualmente ejecutándose. Los sucesos que pueden darle el control al SO son:

- *Interrupción Hardware externa*: originada por algún tipo de suceso que es externo e independiente del proceso que está ejecutándose. Interrupción de E/S, Interrupción de reloj.

- *Interrupción Hardware interna (Excepciones)*: causada por una condición de error generada dentro del proceso que está ejecutándose (división por 0, el desbordamiento, el acceso a una posición de memoria no permitida, acceso ilegal a un archivo, etc.) o por condiciones anómalas (Fallo de página, etc.)

- *Interrupción Software (Llamadas al sistema):* Generadas por el programa en ejecución. Llamadas al

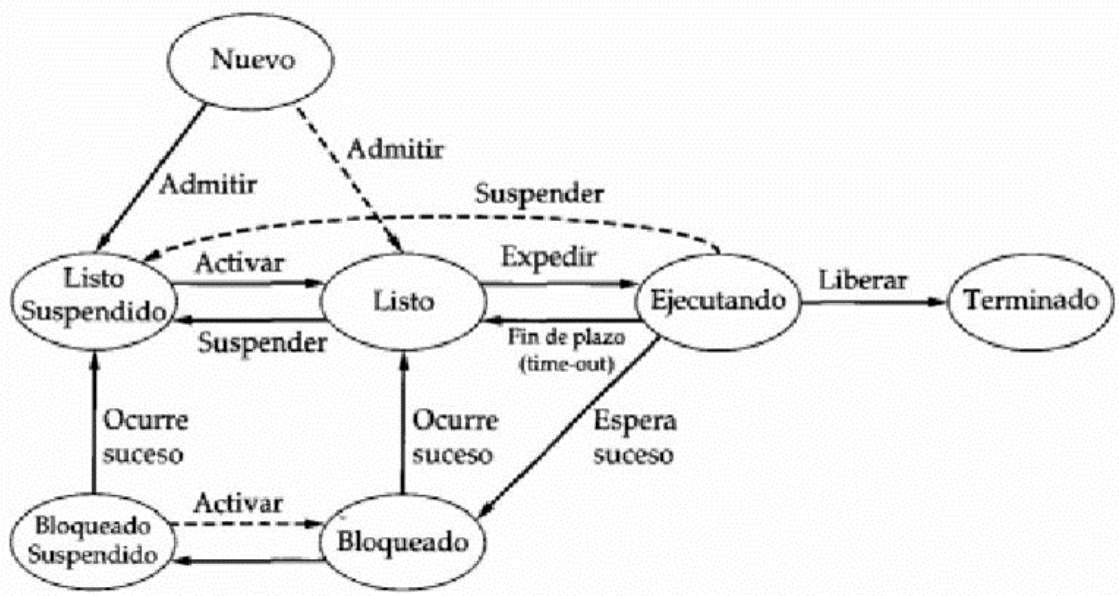
sistema para que el SO ejecute instrucciones privilegiadas en nombre del programa.

*Nota:* Sobre el cambio de modo. Cada vez que se produce una interrupción: Se salva e contexto del programa que está ejecutándose. Se asigna al contador del programa la dirección del programa de tratamiento de la interrupción. Finalmente cambia de modo de usuario a modo Kernel para poder ejecutar las instrucciones privilegiadas del programa de la interrupción.

*Nota:* ¿Que constituye salvar el contexto del programa? Se debe salvar cualquier información que pueda alterarse por la ejecución de la rutina de tratamiento de la interrupción y todo lo necesario para reanudar el programa interrumpido (*Debe salvarse la parte del PCB que se denomina Información de estado del procesador*).

**Modelo de 7 estados**

La principal tarea del SO es controlar la ejecución de los procesos




Cada uno de los procesos de los procesos puede estar en uno de los siguientes estados:

1. Nuevo → el proceso se acaba de crear, pero aún no es admitido por el SO como proceso ejecutable (aún no está cargado en la MP. Se mantiene en MS).

2. Listo → el proceso está preparado para poder ejecutarse (en MP y listo para ejecutarse).

3. Ejecutando → el proceso está actualmente en ejecución

4. Bloqueado → el proceso no se puede ejecutar hasta que se produzca cierto suceso (en MP esperando un suceso)

5. Terminado → por alguna razón el SO saco al proceso del grupo de procesos ejecutables.

6. Bloqueado y suspendido → el proceso está en MS esperando un suceso.

7. Listo y suspendido → el proceso está en MS pero está disponible para su ejecución tan pronto que se cargue en MP

*Nota: MP (Memoria Principal), MS (Memoria Secundaria).*

La figura indica los tipos de sucesos que producen cada transición de estados para un proceso. Algunas aclaraciones:

- Nulo → Nuevo: Las razones para la creación de un proceso son, Trabajo por lotes, Conexión interactiva (el usuario crea un proceso desde un terminal), el SO crea un proceso para dar un servicio y un proceso crea otro proceso.

- Ejecución → Varios: Por espera de un suceso, por fin de Quantum, Yield (proceso que sede voluntariamente el procesador y fin de proceso.

  *Importante:* Un proceso puede pasar de Ejecución → Listo, únicamente, si abandona involuntariamente la CPU (Preemptive /Apropiativo /Con desalojo), ya sea por una Interrupción de Reloj o porque se pasó a preparado un otro proceso con mayor prioridad.

  Si la planificación es Non-Preemptive/ No Apropiativo / sin desalojo) el proceso se ejecuta hasta que termina o hasta que tenga que hacer una E/S (pasa a bloqueados), por lo que NO es posible que un proceso pase del estado "en ejecución" a listo.

- Varios →Terminado: Un proceso puede terminar en cualquier momento. Por eso están las transiciones de Listo a Terminado, Bloqueado a Terminado, etc.

- Bloqueados: Es conveniente tener una cola de bloqueados por cada suceso, ya que si hay 1 sola cola de bloqueados, el SO debería recorrer toda la lista en busca de aquellos procesos que esperaban que ocurra el suceso. En cambio, si se tiene una cola por suceso, directamente la lista entera del suceso puede pasar al estado de listo.

**Procesos Suspendidos (Necesidad del intercambio)**

1. Por más que haya multiprogramación (varios procesos en MP), el procesador podría estar desocupado la mayor parte del tiempo. Esto se debe a que el procesador es mucho más rápido que cualquier operación de E/S, lo cual podría generarse una situación en que todos los procesos estén en estado bloqueado y el

procesador no tenga proceso para ejecutar.

Una solución a este problema, es el intercambio (swap), que significa mover una parte del proceso o todo el proceso de la MP a MS. De esta manera se baja el grado de multiprogramación, permitiendo el ingreso de un nuevo proceso listo para ejecutar (depende del diseño se puede traer un Nuevo proceso o uno Suspendido y listo). Por lo general se suspende aquel proceso bloqueado que hace tiempo esta esperado un suceso.

2. Otros motivos de suspensión: cambios en los requisitos de memoria, para mejorar la estabilidad del sistema, el proceso puede ser causante de un problema, por solicitud del proceso padre, por solicitud de un usuario, un proceso temporal que se ejecuta cada cierto tiempo, etc.

*Nota: El intercambio, en sí, es una operación de E/S a disco por lo que existe la posibilidad de que el problema empeore e vez de mejorar (aunque la E/S de disco es la más rápida del sistema y por lo general suele mejorar el rendimiento).*

***Importante:*** Incluso en un sistema de Memoria Virtual, el SO siempre tendrá que expulsar de vez en cuando algunos procesos, de forma explícita y completa (mediante intercambio), con el objetivo de mejorar el rendimiento, ya que el rendimiento de un sistema de memoria virtual puede desplomarse si hay un número suficientemente grande de procesos activos.

## Capitulo: Planificación de la CPU

Casi todos los recursos de la PC se planifican antes de usarlos. La CPU, es uno de los principales recursos de la PC, así que su correcta planificación resulta crucial en el diseño del SO.

La clave de la multiprogramación está en la planificación (varios procesos en MP para optimizar el uso de CPU), ya que en sí, la planificación es una "gestión de las colas" que minimice el tiempo de espera, de respuesta y de ejecución y maximice el rendimiento del entorno (uso de CPU y tasa de procesamiento).

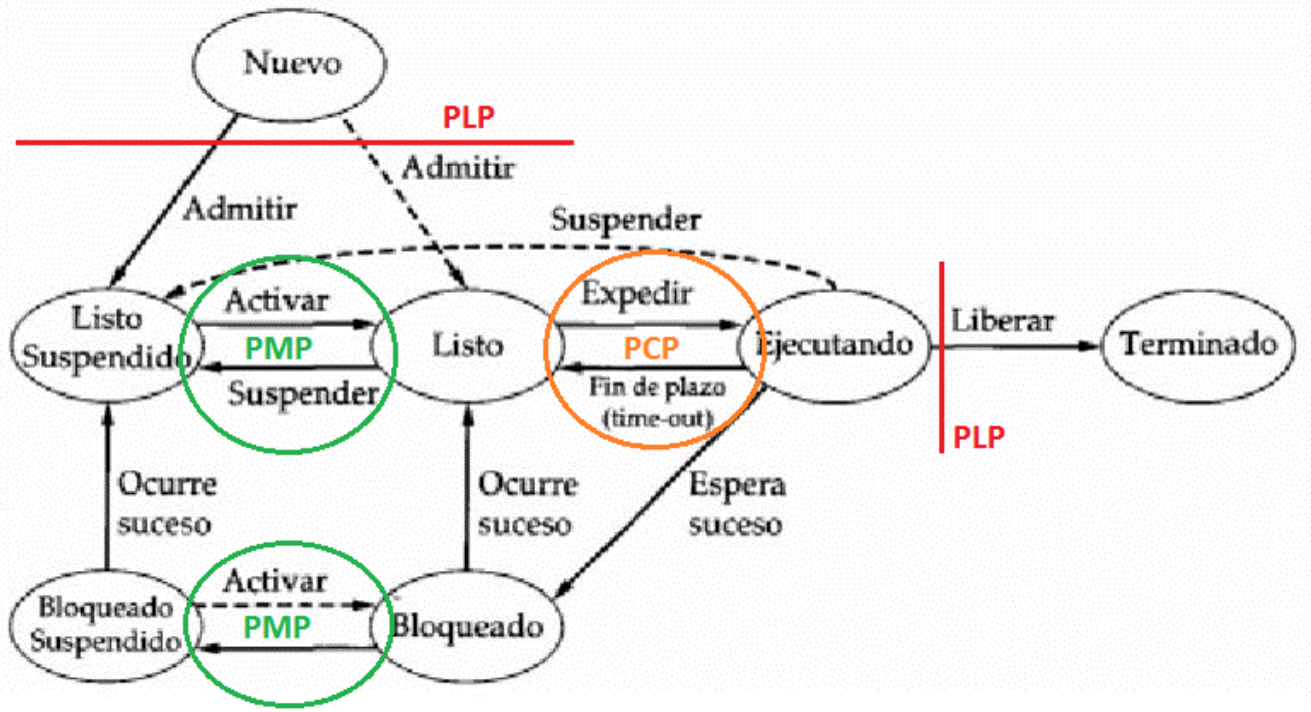
### Planificadores del Procesador

- PLP (Planificador de largo plazo): Determina el grado de multiprogramación (cuantos procesos pueden estar en MP). A su vez debe lograr la estabilidad del sistema (un equilibrio entre procesos CPU BOUND e IO BOUND).

- PMP (Planificador de Mediano plazo): Controla el grado de multiprogramación. Encargado de realizar el intercambio (swap) para baja el grado de multiprogramación.

- PCP (Planificador de corto plazo): Controla el grado de multiprocesamiento (cantidad de procesos que se puede estar ejecutando a la vez). Encargado de determinar qué proceso se va a ejecutar. También conocido como distribuidor (dispatcher).

- PELP (Planificador extra largo plazo): En base a un histórico de procesos ejecutados puede cambiar el algoritmo de planificación o el grado de multiprogramación.




*Nota:* Los nombres de los planificadores hacen referencia a la frecuencia con la que estas funciones se ejecutan.

**Algoritmos de planificación a corto plazo**

El planificador a corto plazo se ejecuta cuando se produce:

1. Llamada al sistema (Ejecución → Espera).
2. Interrupción de Reloj (Ejecución → Preparado).
3. Interrupción de E/S (Espera → Preparado).
4. Terminación del proceso (Ejecución → Terminado).

*Nota:*
Cuando la planificación solo tiene lugar en las situaciones 1 y 4, se dice que el esquema de planificación es **Non-Preemptive / No Apropiativo / Sin Desalojo / Cooperativo** ya que el SO no tiene la capacidad de quitarle la CPU al proceso hasta que este termine o ceda voluntariamente la CPU con una llamada al sistema.

Si la planificación también tiene lugar en las situaciones 2 y 3, se dice que el esquema de planificación es **Preemptive / Apropiativo / Con Desalojo** ya que el SO tiene la capacidad de quitarle la CPU al proceso mediante un fin de quantum o debido a que paso a preparado un proceso con más prioridad que el proceso que se está ejecutando.

Los diferentes algoritmos de planificación de la CPU tienen distintas propiedades y la elección de un algoritmo en particular puede favorecer a una clase de procesos sobre otros. Los criterios son los siguientes:

- *Rendimiento / Tasa de procesamiento (↑):* Cantidad de procesos que se procesan por unidad de tiempo.
- *Utilización de la CPU (throughput) (↑):* No desperdiciar el uso de la CPU.
- *Tiempo de espera (waiting time)* (↓): Tiempo que un proceso tarda en ejecutarse (tiempo que está en la cola de listos).
- *Tiempo de respuesta (response time)* (↓)*:* Tiempo desde que el procesador se pone en la cola de listos hasta que genera una E/S.

  Minimizar la varianza del tiempo de respuesta, significa que si se vuelve a ejecutar el mismo proceso, su tiempo de respuesta debería ser el mismo. Esto es una diferencia con Minimizar el promedio del tiempo de respuesta. En sistemas interactivos conviene minimizar la varianza del tiempo de respuesta (ya que el tiempo de respuesta es predecible. Siempre tarda lo mismo).

- *Tiempo de Retorno (turnaround time)* (↓): Tiempo que tarda un proceso de pasar de listo → terminado (es el tiempo total que está en el sistema).

*Nota*: Maximizar *(↑) / Minimizar* (↓)

Los algoritmos de planificación a corto plazo son:

1. FCFS
2. SJF
3. Por Prioridades
4. Por Turnos (RR, round robin)
5. Colas Multinivel
6. Colas Multinivel Realimentadas

**Planificación FCFS** (Sin desalojo)

Primero en llegar, primero en ser atendido.

Ventajas:

- No se producen muertes por inanición.

Desventajas:

- Tiempo de espera medio largo y se modifica notablemente según el orden en que lleguen los procesos.

- Problemático para sistemas de tiempo compartido por no ser con desalojo.

- Rinde mejor con procesos CPU bound que con I/O bound. Puede provocar un uso ineficiente de la CPU y de los dispositivos de E/S.

  *Efecto convoy: muchos procesos I/O bound esperan mucho tiempo a que un proceso CPU bound se termine de ejecutar. Esto hace que la utilización de la CPU y los dispositivos sea menor que si se ejecutará primero los I/O bound.*

**Planificación SJF** (Sin desalojo o Con Desalojo)

Primero ejecuta el más corto (Ordena por la siguiente ráfaga de CPU más corta). Si 2 procesos tienen la misma ráfaga de CPU, se desempata por FCFS.

Ventajas:

- Tiempo de espera medio disminuye y se mantiene por más que cambie el orden de llegada de los procesos.

Desventajas:

- Se puede producir muerte por Inanición.
- Mucho OVERHEAD. No se puede saber cuál es la siguiente ráfaga de CPU. Solo se puede aproximar, lo que requiere de cálculos extras.

Se usa frecuentemente como mecanismo de planificación a largo plazo.

SJF en sí, es un algoritmo por prioridades → Cuanto más larga sea la ráfaga de CPU, menor prioridad

Tiempo estimado de la próxima Ráfaga → $Tn+1 = Tn.\alpha + Rn\ (1 - \alpha)$ siendo $0 \leq \alpha \leq 1$

- Tn tiempo estimado de la ráfaga anterior
- Rn tiempo real de la ráfaga anterior
- Si α tiende a 0 se le da poca bola al estimado anterior. Si α tiende a 1 poca bola al real anterior.

Observación: SPN → sin desalojo, SRT → con desalojo.
Observación: SRT a diferencia de RR no genera interrupciones adicionales. También debería devolver un mejor tiempo de retorno ya que atiende *inmediatamente* a los procesos de menor duración.

**Planificación por Prioridades** (Sin desalojo o Con Desalojo)
Primero se atiende el de mayor prioridad. Se asume que 0 es la prioridad más alta.

Ventajas:

- Tiempo de espera medio disminuye y se mantiene por más que cambie el orden de llegada de los procesos.

Desventajas:

- Se puede producir muerte por Inanición.

*Se puede solucionar la muerte por Inanición aplicando un mecanismo de* ***envejecimiento****. Se aumenta gradualmente la prioridad de los procesos que estén esperando en el sistema durante mucho tiempo.*

**Planificación HRRN** (Sin desalojo)
Primero el de mayor tasa de respuesta. (Basado en prioridades)

Tasa de Respuesta → $R = W + S / S$
Siendo W tiempo de espera y S tiempo de ráfaga esperado.
Cuanto mayor R, mayor prioridad
Tiene en cuenta la edad del proceso (por W, tiempo de espera). Por lo tanto *elimina el problema de inanición*.

**Planificación RR** (Con desalojo)
Igual que FCFS pero con desalojo por fin de quantum.
Si Q es muy grande: Termina siendo igual que FCFS
Si Q es muy chico: Produce mucho OVERHEAD.

Importante:

Ante simultaneidad de evento se atiende primero:

1. Excepción
2. Interrupción Q
3. Interrupción E/S
4. Llamada al sistema

## Anexo: Procesos y Planificación

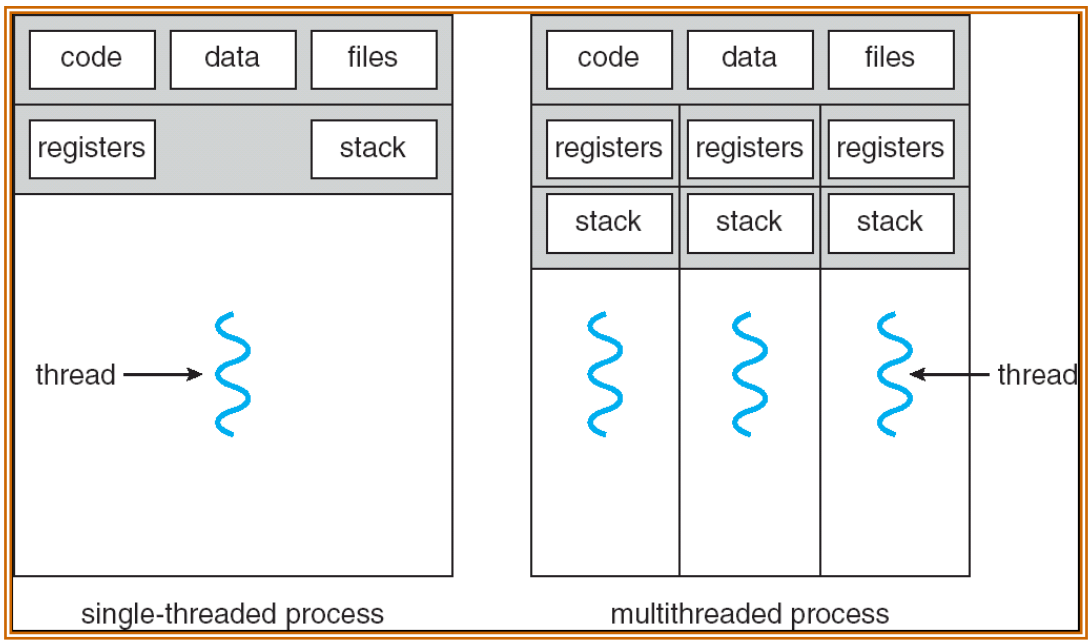
*Verdaderos y Falsos de Final*

1. *El algoritmo HRRN es más equitativo y justo que SJF.*

   **Verdadero.** En SJF los procesos pueden morir por inanición debido a que solamente le da prioridad al proceso que tenga la siguiente ráfaga de CPU más corta. En cambio, HRRN le da prioridad a los procesos más cortos, pero también tiene en cuenta la "edad" de los procesos según el tiempo que lleva en espera (por su fórmula)

Un hilo es un proceso ligero. Un proceso ahora está compuesto por uno o más hilos. Cada hilo puede realizar una tarea.

## Proceso Monohebra vs Miltihebra




En un modelo multihilo:

- Los hilos de un mismo proceso comparten el código, datos y los recursos del proceso (comparten el mismo espacio de direcciones).
- Cada hilo tiene su propia pila y su bloque de control de hilo (además del bloque de control de proceso).

## Ventajas de los hilos

Sus beneficios se derivan de las implicaciones de rendimiento:

1. Economía: Se consume mucho menos tiempo en crear/finalizar y realizar cambios entre hilos debido a que comparten recursos del proceso al que pertenecen. (PRINCIPAL VENTAJA DE LOS HILOS).

2. Compartir recursos: Permite que una aplicación tenga varios hilos que realizan una actividad diferente dentro del mismo espacio de memoria.

3. Capacidad de respuesta: Una aplicación puede realizar varias tareas a la vez e inclusive seguir ejecutándose aunque parte de ella este bloqueada.

4. Utilización de multiprocesador: Las hebras pueden ejecutarse en paralelo en cada procesador

incrementando el grado de concurrencia.

5. Los hilos de un mismo proceso pueden comunicarse ente si, sin invocar al núcleo.

**Estado de un proceso**

Ejecución, listo y bloqueado. No tiene sentido asociar estados de suspensión ya que pertenece al concepto de proceso.

**Hilos a nivel de usuario vs Hilos a nivel del núcleo**

En una aplicación ULT:

- La gestión de hilos la realiza la aplicación (biblioteca de hilos). El núcleo no es consciente de la existencia de hilos (el SO solamente ve un proceso con un hilo, no sabe de la existencia de hilos).
  Por defecto un proceso comienza su ejecución con un único hilo cuyo proceso es gestionado por el SO. El proceso puede crear un nuevo hilo que se ejecuta dentro del mismo proceso. *La creación se lleva a cabo por la biblioteca de hilos*. *La planificación de los hilos también está a cargo de la biblioteca de hilos.* Cuando el control pasa a la biblioteca, se salva el contexto del hilo actual, que se restaura cuando el control pasa de la biblioteca al hilo.
  Todas las operaciones descriptas, se llevan a cabo en el espacio de usuario dentro de un mismo proceso. El núcleo no tiene conocimientos de ellas. El núcleo continúa planificando el proceso como una única unidad y asignándole un único estado.

- Si un hilo realiza una llamada al sistema, se bloquea todo el proceso (la biblioteca el SO, el ULT sigue "ejecutando", por lo tanto cuando el proceso vuelve a ejecutar, sigue ejecutando el mismo ULT). Lo mismo pasa si al proceso se le acaba el quantum de tiempo, para la biblioteca, el hilo que estaba corriendo, sigue en estado ejecutando (aunque verdaderamente no esté haciendo uso de la CPU). Cuando el SO le devuelve a CPU al proceso, continua en ejecución el hijo que estaba ejecutando antes de que se bloqueara el proceso.

- Si un hilo A solicita a otro hilo B del mismo proceso que realice alguna acción. El hilo A se pone en bloqueado y el hilo B pasa a Ejecutando. Importante darse cuenta que el paso de un proceso a otro, lo realiza la biblioteca de hilos y que el proceso sigue en estado ejecutando.

*Ventajas de los ULT:*

- El intercambio de hilos no necesita de los privilegios del modo Kernel, por estos comparten el espacio de direcciones de usuario de un mismo proceso. NO se necesitan 2 cambios de modo: de usuario a Kernel y de Kernel a usuario.

- Planificación especifica → cada aplicación planifica sus hilos mediante la biblioteca de hilos según le convenga.

- Los ULT pueden ejecutarse en cualquier SO. No es necesario realizar cambios en el núcleo subyacente.

*Desventajas de los ULT*:

- La mayoría de las llamas al sistema son bloqueantes, por cual cada vez que un ULT realiza una llamada al sistema, bloquea todo el proceso.

- No aprovecha el multiprocesamiento. El SO puede asignar un solo proceso a un solo procesador cada vez. Por lo tanto se puede ejecutar un hilo de cada proceso en cada instante.

Nota: se puede solucionar mediante la técnica de jacketing (recubrimiento) que convierte una llamada bloqueadora en una llamada no bloqueadora.

En una aplicación KLT:

- El SO conoce la existencia de los hilos y se encarga de realizar la gestión de los mismos.

*Ventajas de los KLT*:

Resuelve los problemas de los ULT:

- Si se bloquea uno de los hilos, no bloquea a todo el proceso. El núcleo puede planificar otro hilo del mismo proceso.

- Aprovecha el multiprocesamiento. El SO puede planificar diferentes hilos de un mismo proceso en diferentes procesadores.

*Desventaja de los KLT*:

- El cambio de un hilo a otro requiere de cambio de modo.

**Tipo de relación entre los ULT y KLT**

- Muchos a uno: múltiples hilos de usuario a uno de Kernel. Desventaja que no hay concurrencia ya que 1 solo hilo ULT puede acceder al núcleo a la vez (no hay multiprocesamiento). Además una llamada bloqueante bloquea a todo el proceso.

- Uno a Uno: Soluciona los 2 problemas de Muchos a uno, aunque el problema es que por cada UTL hay que crear un KLT, por lo puede repercutir en el rendimiento. Se debe restringir el número de hilos soportados por el sistema.

- Muchos a Muchos: Soluciona el problema de uno a uno.

## Capitulo: Concurrencia (exclusión Mutua y Sincronización)

La concurrencia es fundamental en la:

- Multiprogramación
- Multiprocesamiento
- Procesamiento distribuido (gestión de varios procesos, ejecutándose en sistemas de computadores múltiples y distribuidos).

La concurrencia comprende un gran número de cuestiones de diseño del SO como es la *compartición de recursos*, comunicación entre procesos, asignación de recursos a los procesos, etc.

### Principios generales de la Concurrencia

En un sistema multiprogramado con un único procesador, los procesos se intercalan en el tiempo para dar la apariencia de ejecución simultánea, a diferencia de un sistema multiprocesamiento en el que consigue procesamiento paralelo real.
*Aunque intercalación y superposición representen formas de ejecución diferente, ambas técnicas pueden contemplarse como ejemplos de procesamiento concurrente y ambas plantean los mismos problemas.*

Las dificultades que presenta la concurrencia son:

1. **Compartir recursos globales está lleno de riesgos**. Si dos procesos hacen uso al mismo tiempo del mismo recurso, el orden en que se ejecuten las lecturas y escrituras, es crítico.
2. **Es difícil gestionar la asignación óptima de recursos**.
3. **Resulta difícil localizar un error de programación**.

El problema de concurrencia se presenta en la multiprogramación debido a que una interrupción puede detener la ejecución de instrucciones de un proceso en cualquier momento y dar lugar a otro sin que este allá finalizado.

En el caso de un sistema multiprocesador, se tiene la misma condición y además el problema puede ser causado por 2 procesos que se estén ejecutando simultáneamente y que intenten ambos acceder a la misma variable global (Recordemos que la velocidad relativa de ejecución de los procesos es impredecible.)

Sin embargo, la solución al problema de la concurrencia para ambos tipos de sistemas es la misma:
CONTROLAR EL ACCESO AL RECURSO COMPARTIDO.

*Moraleja:*
*La exigencia básica para soporta la concurrencia de procesos es la posibilidad de* ***hacer cumplir la exclusión mutua****, es decir la capacidad de prohibir a los demás procesos realizar una acción cuando un proceso haya obtenido el permiso (un solo proceso puede hacer uso de la región critica a la vez)*

*Nota:*
*Un sistema operativo no Apropiativo está esencialmente libre de condiciones de carrera en lo que respecta a las estructuras de datos del Kernel (ejemplo: lista de todos los archivos abiertos) ya que hay solo 1 proceso activo en el Kernel en cada momento.*

**Interacción ente procesos**

Los procesos que se ejecutan concurrentemente pueden ser procesos independientes o procesos cooperativos:

- *Procesos independientes* → Aquel que NO puede afectar o verse afectado por los restantes procesos que se encuentran en el sistema. Cualquier proceso que NO comparte datos con ningún otro proceso es un proceso independiente.

- *Procesos cooperativos* → Aquel que puede afectar o verse afectado por los restantes procesos que se encuentran en el sistema. Cualquier proceso que comparte datos con otros procesos es un proceso cooperativo. Los procesos cooperativos pueden compartir un espacio de memoria directamente (*cooperación por compartimiento*) o compartir datos solo a través de archivos o mensajes (*cooperación por comunicación).* El primer caso se consigue mediante el uso de hilos.

En el caso de los procesos independientes, aunque no compartan datos, el sistema operativo tiene que encargarse de la **competencia** por los recursos (CPU, impresora, memoria, ETC). En este caso, los resultados de un proceso son independiente de las acciones de los otros, pero la ejecución de un proceso puede influir en el comportamiento de los procesos que compiten (*los tiempos de los procesos pueden verse afectados*)

Cuando hay procesos en competencia por un mismo recurso, se deberán solucionar 3 problemas de control:

1. Exclusión Mutua (hacer cumplir la exclusión mutua crea 2 problemas de control adicionales, 2 y 3)
2. Interbloqueo
3. Inanición

*Nota:*
*Hacer cumplir la exclusión mutua hace que aparezcan los problemas 2 y 3. Uno es el* ***interbloqueo*** *(un proceso P1 está esperando que un proceso P2 libere un recurso y el P2 está esperado a que P1 libere un recurso. Ni P1 ni P2 liberan el recurso hasta que el otro libere el recurso que necesita) y* ***la inanición*** *(un proceso espera indefinidamente a que se le asigne el acceso al recurso).*

*Nota*:
*El* ***control de la competencia*** *siempre involucra al SO porque es quien asigna los recursos.*

*Nota:*
*Sección critica → Segmento de código que hace uso de un recurso crítico*

*Recurso critico → Cualquier recurso compartido (variables, archivos, impresora, lista, etc.)*
El caso de **cooperación por compartimiento** los procesos pueden tener acceso a variables compartidas, archivos o base de datos compartidos. No saben que procesos acceden a los datos, pero son conscientes de que estos otros pueden tener acceso a los mismos datos.

Como los datos se guardan en recursos (dispositivos, memoria, etc.) también presentan los problemas de exclusión mutua, interbloqueo e inanición por competir por recursos críticos. Además se debe introducir un nuevo requisito: **Integridad/Coherencia** de los datos.

En el caso de **cooperación entre procesos por comunicación**, la comunicación es una manera de sincronizar o coordinar las distintas actividades. *Por lo que no es necesario el control de la mutua exclusión para este tipo de cooperación*. Sin embargo, los problemas de inanición e interbloqueo siguen presentes.

**Condición de carrera**
Cuando varios procesos manipulan y acceden concurrentemente a los mismos datos y el resultado de la ejecución depende del orden en que se produzcan los accesos.

Para protegerse frente a este tipo de condiciones, se necesita garantizar que solo un proceso cada vez puede acceder a la sección crítica.

**Protocolo de la región critica**

1. Exclusión mutua: cuando un proceso usa la región crítica, ningún otro proceso, que tenga región crítica por el mismo recurso u objeto compartido, puede entrar en ella.

2. Progreso: Cuando un proceso está fuera de la región crítica, no pueden impedir a otros procesos que la usen.

3. Espera limitada: Cualquier proceso que quiere entrar a la región critica lo debe poder hacer luego de un número finito de intentos.

4. Velocidad de los procesos: todos los procesos pueden usar la región crítica las veces que quieran y el tiempo que quieran.

*Nota*:

- *Espera activa → Cuando un proceso que no obtuvo permiso para entrar en su sección critica, se queda comprobando periódicamente la variable hasta que pueda entrar. Consume tiempo de procesador (está activo) mientras espera su oportunidad.*

- *Espera NO activa → se bloquea al proceso y se lo desbloquea cuando puede entrar.*

**Soluciones para satisfacer los requisitos del protocolo de región crítica**

1. Solución por software.
2. Solución por hardware.
3. Soporte del SO / lenguaje de programación.

**Soluciones por software**

*Algoritmo de Dekker (1° intento)*

Cualquier intento de exclusión mutua debe depender de algunos mecanismos básicos de exclusión en el hardware. El más habitual es que sólo se puede acceder a una posición de memoria en cada instante, teniendo en cuenta esto se reserva una posición de memoria global llamada turno. Un proceso que desea ejecutar su sección crítica primero evalúa el contenido de turno. Si el valor de turno es igual al número del proceso, el proceso puede continuar con su sección crítica. En otro caso el proceso debe esperar. El proceso en espera, lee repetitivamente el valor de turno hasta que puede entrar en su sección crítica. Este procedimiento se llama espera activa.

Después de que un proceso accede a su sección crítica y termina con ella, debe actualizar el valor de turno para el otro proceso.

| Proceso 0 | Proceso 1 |
|---|---|
| ... ... | ... |
| /*esperar*/ | /*esperar*/ |
| while (turno!=0); | while (turno!=1); |
| /*sección crítica*/ | /*sección crítica*/ |
| ... | ... |
| turno=1; | turno=0; |
| ... | ... |

Esta solución garantiza el problema de la exclusión mutua pero tiene 2 inconvenientes:

- Deben alternarse de forma estricta en el uso de sus secciones críticas, por lo que el ritmo de ejecución depende del proceso más lento.

- *No cumple la condición de progreso* → Si un proceso falla fuera o dentro de la sección critica, el otro proceso se bloquea permanentemente.

*Algoritmo de Dekker (2do intento)*

Cada proceso debe tener su propia llave de la sección crítica para que, si uno de ellos falla, pueda seguir

accediendo a su sección crítica; para esto se define un vector booleano señal. Cada proceso puede evaluar el valor de señal del otro, pero no modificarlo. Cuando un proceso desea entrar en su sección crítica, comprueba la variable señal del otro hasta que tiene el valor falso (indica que el otro proceso no está en su sección crítica). Asigna a su propia señal el valor cierto y entra en su sección crítica. Cuando deja su sección crítica asigna falso a su señal.

| Proceso 0 | Proceso 1 |
|---|---|
| ... | ... |
| /*esperar*/ | /*esperar*/ |
| While (señal[1]); | While (señal[0]); |
| señal[0] = cierto; | señal[1]=cierto; |
| /*sección crítica*/ | /*sección crítica*/ |
| ... | ... |
| señal[0] = falso; | señal[1] = falso; |

Se solucionan los problemas anteriores, sin embargo, ahora surgen otros nuevos:

- Si uno de los procesos falla dentro de su sección crítica el otro quedará bloqueado permanentemente.
- No se garantiza la Exclusión Mutua como vemos en la secuencia:
  1. P0 ejecuta el While y encuentra señal [1] a falso.
  2. P1 ejecuta el While y encuentra señal [0] a falso.
  3. P0 pone señal [0] a cierto y entra en su sección crítica.
  4. P1 pone señal [1] a cierto y entra en su sección crítica.

Ambos procesos están en su sección crítica, esto se debe a que esta solución no es independiente de la velocidad de ejecución relativa de los procesos.

*Algoritmo de Dekker (3° intento)*

Una vez que un proceso ha puesto su señal en cierto, el otro no puede entrar a su sección crítica hasta que el primero salga de ella. Se garantiza la EM, sin embargo, se generará interbloqueo si ambos procesos ponen su señal a cierto antes de ambos hayan ejecutado el While.

Además, si un proceso falla en su sección crítica, el otro queda bloqueado permanentemente.

| Proceso 0 | Proceso 1 |
|---|---|
| ... | ... |
| señal[0] = cierto; | señal[1]=cierto; |
| /*esperar*/ | /*esperar*/ |
| While (señal[1]); | While (señal[0]); |
| /*sección crítica*/ | sección crítica*/ |

| | |
|---|---|
| ... ... | ... ... |
| señal[0] = falso; | señal[1] = falso; |

*Algoritmo de Dekker (4° intento)*

En el tercer intento, un proceso fijaba su estado sin conocer el estado del otro. Se puede arreglar esto haciendo que los procesos activen su señal para indicar que desean entrar en la sección crítica pero deben estar listos para desactivar la variable señal y ceder la preferencia al otro proceso.

Existe una situación llamada **bloqueo vital**, *esto no es un interbloqueo, porque cualquier cambio en la velocidad relativa de los procesos rompería este ciclo y permitiría a uno entrar en la sección crítica*. Recordando que el interbloqueo se produce cuando un conjunto de procesos desean entrar en sus secciones críticas, pero ninguno lo consigue. Con el bloqueo vital hay posibles secuencias de ejecución con éxito.

*Algoritmo de Dekker (Solución Correcta)*

Combinación entre intento 1 y 4:

```
while (true)
{
        interesado[0] = TRUE;
        while (interesado[1])
        {
                if (turno ≠ 0)
                {
                        interesado[0] = FALSE;
                        while (turno ≠ 0);
                        interesado[0] = TRUE;
                }
        }
                SECCIÓN CRÍTICA
        turno = 1;
        interesado[0] = FALSE;
                SECCIÓN RESTANTE
}
```

El algoritmo de Dekker resuelve el problema de la exclusión mutua.

*Algoritmo de Peterson*

Resuelve también el problema de la exclusión mutua, pero a diferencia del algoritmo de Dekker, mediante una solución simple y elegante.

**Soluciones por hardware**

- *Inhabilitación de interrupciones*
  En un sistema Apropiativo, un proceso continuará ejecutándose hasta que solicite una llamada al sistema o hasta que sea interrumpido. ***Por lo tanto, para garantizar la exclusión mutua, es suficiente con impedir que un proceso sea interrumpido.***

  *Problemas 1:* Valido solamente para sistemas monoprocesador. Para sistemas multiprocesador, es posible que haya más de un proceso ejecutándose al mismo tiempo.

  *Problema 2:* Pierde eficiencia el sistema ya que se limita la capacidad del procesador para intercalar procesos.

- *Instrucciones de maquina*
  - *Test and set (TLS)*
    Instrucción Hardware que lee y modifica atómicamente una variable (por ser atómica esta instrucción no puede ser interrumpida). Además, si nos encontramos en un sistema multiprocesador, ninguno de los demás procesadores tendrá acceso a la variable hasta que termine de ejecutarse la instrucción.

  - *Swap (intercambio)*
    De forma atómica sobre 2 palabras.

  Ventajas de la solución con instrucciones de máquina
  - Valido para sistema monoprocesador y multiprocesador.

  Desventajas
  - Emplea espera activa.
  - Pude producir inanición.
  - Puede producir interbloqueo.

En las soluciones software, los procesos deben coordinarse unos con otros para para cumplir la exclusión mutua, si ayuda por parte del lenguaje de programación o del sistema operativo.

Las soluciones por software son propensas a errores y a una fuerte carga de proceso.
Las soluciones por hardware reducen la sobrecarga, pero no son interesantes como solución de carácter general.

Debido a estos inconvenientes, tanto de las soluciones por software como por hardware, es necesario buscar otros mecanismos.

**Soluciones con soporte del lenguaje de programación o del SO**

*Semáforos*

Son *variables especiales* que tienen un valor entero sobre el que se definen 3 operaciones:

1) Un semáforo puede iniciarse con un valor no negativo
2) La operación wait disminuye el valor del semáforo. Si el valor se hace negativo, el proceso que ejecuta el wait se bloquea.
3) La operación signal incrementa el valor del semáforo. Si el valor no es positivo, se desbloquea un proceso bloqueado por una operación wait.

Son las únicas 3 operaciones posibles para manipular semáforos.

| WAIT(S) | SIGNAL(S) |
|---|---|
| Semáforo-- | Semáforo++ |
| If (semáforo<0) | If (semáforo<=0) |
| { | { |
| Poner este proceso en la cola del semáforo; | Quitar un proceso de la cola del semáforo; |
| Bloquear este proceso | Poner el proceso en la cola de listos; |
| } | } |

En cualquier instante el valor del semáforo puede interpretarse de la siguiente forma:

- Semáforo >= 0: Es el número de procesos que puede ejecutar un wait(s) sin bloquearse.
- Semáforo < 0: Cantidad de procesos bloqueados en la cola del semáforo

Las primitivas wait(s) y signal(s) son atómicas. Son relativamente cortas, por lo que la cantidad de espera activa que se obtiene será menor.

Los semáforos emplean una cola para mantener los procesos que están esperando en el semáforo. Si el semáforo incluye la estrategia de cómo se retiran los procesos de la cola, se denomina **semáforo robusto** (Ejemplo: se retira por FIFO). En caso contrario, **semáforo débil**.
Los semáforos robustos garantizan la inexistencia de inanición, pero no así los semáforos débiles.

*Nota*:
Semáforos binarios

| WAITB(S) | SIGNALB(S) |
|---|---|
| If (semáforo = 1) | If (Cola.semáforo.esvacia ) |
| Semáforo = 0; | Semáforo = 1; |
| Else | Else |

| | |
|---|---|
| { | { |
| Poner este proceso en la cola del semáforo; | Quitar un proceso de la cola del semáforo; |
| Bloquear este proceso; | Poner el proceso en la cola de listos; |
| } | } |

**Problema del productor y consumidor**


Uno o más productores generan cierto tipo de datos y los sitúan en un buffer. Un único consumidor saca elementos del buffer de uno en uno. El sistema debe impedir la superposición de operaciones sobre el buffer. Es decir un solo agente consumidor o productor puede acceder al buffer en un instante dado (se logra con un semáforo binario mutex)

Si el buffer está lleno, el productor debe ser demorado (en el caso que el buffer sea limitado) → Semáforo Lugar
Si el buffer está vacío, el consumidor debe ser demorado → Semáforo Lleno

*Buffer ilimitado (color rojo) y buffer limitado (se agrega el semáforo lugar en verde)*

```
Lleno = 0
Mutex = 1
Lugar = N
Productor ( )
{
        While (1)
        {
        Msg = producir ( )
        Wait (Mutex)
        Wait (Lugar)
        Depositar (msg)
        Signal (Lleno)
        Signal (Mutex)
        }

Consumidor ( )
{
```

```
While (1)
{
Wait (Lleno)
Wait (Mutex)
Msg = retirar ( )
Signal (Mutex)
Signal (Lugar)
Consumir ( )
```

**Monitores**

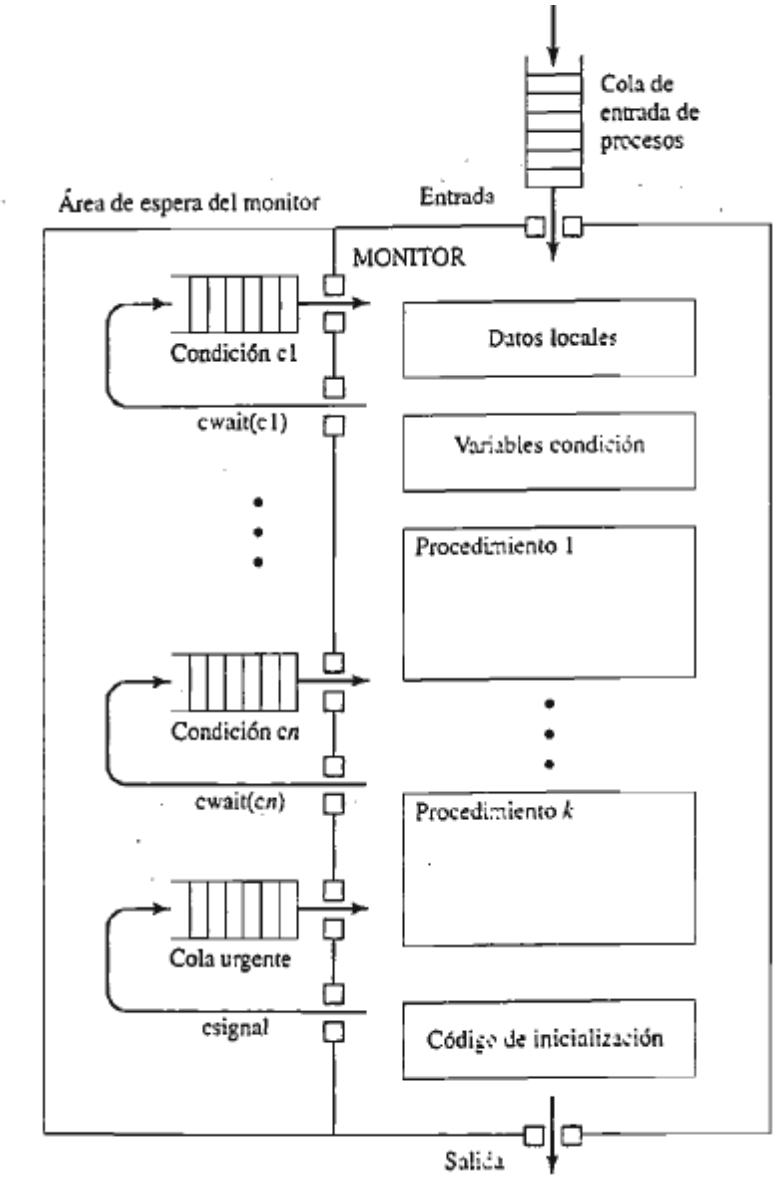
Los semáforos son flexibles y potentes pero puede resultar difícil construir un programa correcto por medio de semáforos, ya que las operaciones wait y signal deben distribuirse por todo el programa y no es fácil advertir el efecto global de estas operaciones sobre los semáforos a los que afectan.

Los monitores ofrecen una funcionalidad equivalente a la de los semáforos y que son más fáciles de controlar.

Un monitor es un módulo de software que consta de uno o más procedimientos, una secuencia de inicio y variables locales. Solo un proceso puede estar ejecutando el monitor a la vez, por lo que ofrece un servicio de exclusión mutua fácilmente.

La ventaja que brinda, es que al almacenar los procedimientos dentro del módulo monitor automáticamente se garantiza la exclusión mutua, ya que solamente un proceso a la vez puede acceder al monitor y por ende a los procedimientos. De esta manera se desliga al programador de hacer cumplir la mutua exclusión. En cuanto a la sincronización de procesos, al igual que en semáforos, es responsabilidad del programador mediante las herramientas de sincronización que brindan los monitores (csignal y cwait).
En cambio en los semáforos, la exclusión mutua como la sincronización son responsabilidades del programador.

Otra de las ventajas de los monitores sobre los semáforos, es que es sencillo de verificar que la sincronización se ha realizado correctamente y detectar los fallos, ya que todas las funciones de sincronización están confinadas dentro del monitor.




*Nota*:
Cwait y csignal son diferentes de las de los semáforos. Si un proceso de un monitor ejecuta un csignal y no hay procesos esperando en la variable condición, el csignal se pierde (como si nunca se hubiera ejecutado).

*Nota*:
Una vez que un proceso está dentro del monitor, puede suspenderse a sí mismo temporalmente bajo la condición x ejecutando cwait(x); entonces se sitúa en una cola de procesos que esperan volver a entrar al monitor cuando la condición cambio (es decir, el proceso no sale fuera del monitor, sino que pasa a la cola de bloqueados de la condición, que se ubica en la zona de espera del monitor. Ver imagen).

## Capitulo: Interbloqueo

El interbloqueo se puede definir como el bloqueo permanente de un conjunto de procesos que compiten por los recursos del sistema o bien se comunican unos con otros.

Todos los interbloqueos suponen necesidades contradictorias de recursos por parte de 2 o más procesos.
Ejemplo:

Proceso P: Obtener A → Obtener B → Liberar A → Liberar B…
Proceso Q: Obtener B → Obtener A → Liberar B → Liberar A…

Podría pasar por ejemplo:

- Q obtiene B y después A y a continuación libera B y A. Cuando P se ejecute podrá obtener ambos recursos.
- O que, Q obtiene B y después P obtiene A. Es inevitable el interbloqueo porque al continuar la ejecución Q se bloqueara por A y P se bloqueara por B.

Por lo tanto, que se produzca o no el interbloqueo depende tanto de la *dinámica de la ejecución* como de los *detalles de la aplicación*. Por ejemplo si P tuviese la siguiente forma:

Proceso P: Obtener A → Liberar A → Obtener B → Liberar B

Independientemente de la ejecución de cada proceso, no puede darse interbloqueo.

### Recursos reutilizables

Un recurso reutilizable es aquel puede ser usado con seguridad por un proceso y no se agota con el uso. Como ejemplo de recursos reutilizables se tienen los procesadores, canales de E/S, memoria principal y secundaria, dispositivos y estructuras de datos tales como *archivos*, base de datos, *semáforos*, etc.

### Recurso Consumible

Un recurso consumible es aquél que puede ser creado (producido) y destruido (consumido). Normalmente no hay límites en el número de recursos consumibles de un tipo en particular. Cuando un proceso adquiere un recurso, este deja de existir. Como ejemplo de recursos consumibles están las interrupciones, señales, mensajes e información de buffers de E/S.

### Condiciones de interbloqueo

Para producirse un interbloqueo deben darse 3 condiciones:

1. **Exclusión Mutua**: solo un proceso puede usar un recurso a la vez.

2. **Retención y espera**: un proceso puede retener unos recursos asignados mientras espera que se le asignen otros.

3. **No apropiación**: ningún proceso puede ser forzado a abandonar un recurso que tenga.

   *Puede existir interbloqueo con estas 3 condiciones, pero puede NO existir con solo estas 3 condiciones. Es decir son condiciones necesarias pero NO suficientes para que exista interbloqueo. Se necesita una cuarta condición para que se produzca interbloqueo:*

4. **Espera Circular (circulo vicioso de espera)**: existe una cadena cerrada de procesos, cada uno de los cuales retiene, al menos un recurso que necesita el siguiente proceso de la cadena.

La cuarta condición es, en realidad, una consecuencia potencial de las 3 primeras. Es decir, dado que se producen las 3 primeras condiciones, puede ocurrir una secuencia de eventos que desemboque en un círculo vicioso de espera irresoluble. Un círculo de espera irresoluble es, de hecho, la definición de interbloqueo.

En resumen, las 4 condiciones en conjunto constituyen una condición necesaria y suficiente para el interbloqueo. (1 a 3 son decisiones estratégicas, mientras que la condición 4 depende de la secuencia de solicitudes y liberación de los proceso).

*Nota*:
Condiciones Coffman.

**Prevención del interbloqueo**

Consiste en diseñar un sistema de manera que esté excluida la posibilidad de interbloqueo.
Los métodos para prevenir el interbloqueo son de 2 tipos:

- Métodos indirectos: impedir la aparición de alguna de las 3 condiciones necesarias.
- Métodos directos: evitar la aparición del cirulo vicioso de espera.

*Exclusión mutua*

Esta condición NO puede anularse.

*Retención y espera*

Esta condición puede prevenirse exigiendo que todos los procesos soliciten todos los recursos que necesiten a un mismo tiempo y bloqueando el proceso hasta que todos los recursos puedan concederse simultáneamente.
Ineficiente desde cualquier punto de vista:

- Un proceso puede permanecer mucho tiempo suspendido hasta que obtenga todos los recursos que necesita.

- Los recursos asignados pueden permanecer mucho tiempo sin ser utilizados por el proceso que los obtuvo.
- Un proceso puede NO conocer por adelantado todos los recursos que necesitará.

*No apropiación*

Esta condición puede prevenirse de varias formas:

- Si a un proceso retiene varios recursos se le deniega una nueva solicitud, dicho proceso deberá liberar sus recursos anteriores y solicitarlos de nuevo.
- Si un proceso solicita un recurso que es retenido por otro, el SO puede solicitarle al segundo que lo libere (si los 2 procesos no tienen la misma prioridad).

Esta técnica solo sirve cuando se aplica a recursos cuyo estado puede salvarse y restaurase más tarde de forma fácil, como el procesador.

*Espera circular*

Esta condición puede prevenirse definiendo una ordenación lineal de los tipos de recursos.
Ineficiente como en retención y espera.

**Predicción del interbloqueo**

Con la predicción del interbloqueo, se puede alcanzar las 3 condiciones necesarias, pero se realizan elecciones acertadas para asegurar que nunca se llegue al puto de interbloqueo.

Se decide dinámicamente si la petición actual de asignación de un recurso podría llevar potencialmente a un interbloqueo. Se necesita conocer las peticiones futuras de recursos.

Dos enfoques de predicción:

- *No iniciar un proceso si sus demandas pueden llevar a interbloqueo*: un proceso comenzará solo si puede asegurarse la demanda máxima de recursos de todos los procesos actuales más la del nuevo proceso.

- *No conceder una solicitud de recurso si esta asignación puede llevar a un interbloqueo (**algoritmo del banquero**):* El algoritmo decide si le asigna recursos a los procesos que los solicitan para mantener el estado seguro (estado en el que existe al menos una secuencia que no lleva a interbloqueo, es decir todos los procesos pueden ejecutarse hasta el final)

*Nota*:
La predicción permite más concurrencia y es menos restrictiva que la prevención (se pueden alcanzar las 3 condiciones necesarias, pero se hace la asignación de manera a que nunca se llegue al interbloqueo).
*Nota*:

No predice el interbloqueo exactamente, sino que anticipa la *posibilidad* de interbloque y asegura que nunca exista esa posibilidad.

Para poder predecir el interbloqueo hay una serie de restricciones:

- Los procesos a considerar deben ser independientes.
- No debe haber un número variable de procesos y recursos, sino un número fijo.
- Los procesos no pueden finalizar mientras retengan recursos.
- Se debe presentar la máxima demanda de recursos por anticipado.

**Detección del interbloqueo**

El algoritmo de detección de interbloqueo:

1. Marcar los procesos que no tengan ningún recurso asignado (fijándose en la matriz de asignación que la fila sea todo cero)
2. Crear un vector W con los recursos disponibles
3. Marcar los procesos en los que las solicitudes sean menor o igual a W y sumarlos a W.

Si al finalizar el algoritmo de detección hay algún proceso NO marcado, indica que hay interbloqueo.

Luego de detectar el interbloqueo, hace falta alguna técnica de recuperación del interbloqueo. Algunas de ellas son:

- Abortar todos los proceso interbloqueados. Es la estrategia más común adoptada por el SO.
- Retroceder cada proceso interbloqueado hasta un punto en que no lo haya.

**Estrategia integrada de interbloqueo**

Es eficiente usar diferentes estrategias en diferentes situaciones:

- Espacio intercambiable → Prevención de retención y espera.
- Recursos de procesos → Predicción
- Memoria Principal → Prevención por apropiación.
- Recursos internos → Prevención por ordenación de recursos.

## Capitulo: Gestión de memoria

En un sistema monoprogramado, la memoria principal se divide en 2 partes: una para el sistema operativo (normalmente en posiciones bajas de la memoria) y otra para el programa que se ejecuta en ese instante (normalmente en posiciones altas de la memoria).

En cambio, en un sistema multiprogramado, la parte de usuario de la memoria debe subdividirse aún más para hacer lugar a varios procesos. *La tarea de subdivisión la lleva a cabo dinámicamente el SO y se conoce como* ***gestión de memoria***.

En un sistema multiprogramado es vital repartir eficientemente la memoria para poder introducir tantos procesos como sea posible y que el procesador este la mayor parte del tiempo ocupado.

### Requisitos de la gestión de memoria

Se proponen 5 requisitos:

1. Reubicación
2. Protección
3. Compartición
4. Organización lógica
5. Organización física

### Reubicación

El sistema busca cargar y descargar los procesos activos en la memoria principal para maximizar el uso del procesador, manteniendo una gran reserva de procesos listos para ejecutarse. Una vez que se descargó el programa a disco, cuando vuelva a ser cargado se puede necesitar **reubicar** el proceso en un área distinta de la memoria.

El *SO* debe conocer la ubicación del PCB, de la pila y el punto de partida del programa del proceso (parte de la imagen el proceso).

El *procesador y el SO* deberán ocuparse de las referencias a la memoria dentro del programa:

- *Instrucciones de bifurcación* → Referencia a la instrucción que se va a ejecutar a continuación.
- *Referencias a los datos* → Deben contener la dirección del byte o de la palabra de datos referenciada.

Nota:
"Ocuparse" hace referencia a que deberán traducir las referencias a memoria en el código del programa a las direcciones físicas reales que reflejen la posición actual del programa en MP.

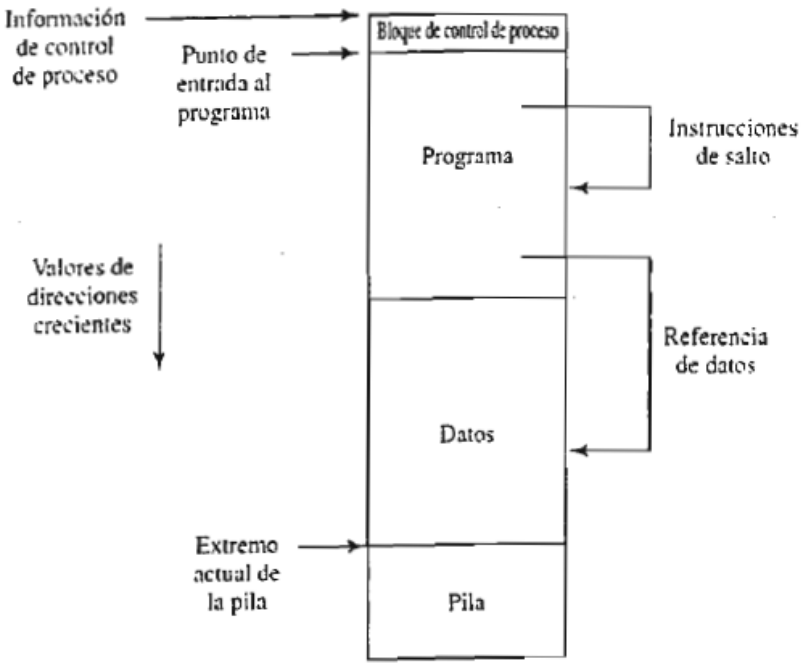



Figura 7.1. Requisitos de direccionamiento para un proceso.

**Protección**

El código de un proceso no puede hacer referencia a posiciones de memoria de otros procesos, sin permiso. Al desconocerse la ubicación de un programa en memoria principal, es imposible comprobar las direcciones absolutas durante la compilación para asegurar la protección. Por lo tanto, todas las referencias a memoria generadas por un proceso deben comprobarse durante la ejecución para asegurar que las instrucciones solo hagan referencia al espacio de memoria destinado a dicho proceso. *El procesador es el que debe satisfacer las exigencias de protección de memoria*, ya que el SO no puede anticiparse a todas las referencia de memoria que hará el programa (y si la anticipación fuera posible, consumiría mucho tiempo proteger por adelantado a cada programa de posibles violaciones de referencia a la memoria).

**Compartimiento**

La protección debe permitir el acceso de varios procesos a la misma zona de la MP. Los mecanismos para respaldar la reubicación forman parte básica de las capacidades de compartimiento.

**Organización lógica**

Si el SO y el hardware pueden tratar a los programas de usuario y los datos en forma de módulos, se conseguirá una serie de ventajas. *La herramienta que satisface esta necesidad es la segmentación*.

**Organización Física**

La memoria secundaria puede permitir un almacenamiento a largo plazo de programas y datos, al tiempo que

una memoria principal mantiene los programas y datos de uso actual.
*En este esquema de 2 niveles, la organización del flujo de información entre la memoria principal y la memoria secundaria debe ser* ***responsabilidad del sistema.*** La responsabilidad de este flujo NO puede ser asignada al programador, por dos razones:

- En un sistema multiprogramado, el programador no conoce durante la codificación cuanto espacio de memoria habrá disponible o donde estará este espacio.

- La memoria principal para un programa puede ser insuficiente. En este caso el programador debe emplear una práctica conocida como superposición (varios módulos asignados a la misma región de memoria, intercalándose entre sí según se necesite). La programación superpuesta malgasta el tiempo del programador.

**Técnicas de gestión de memoria**

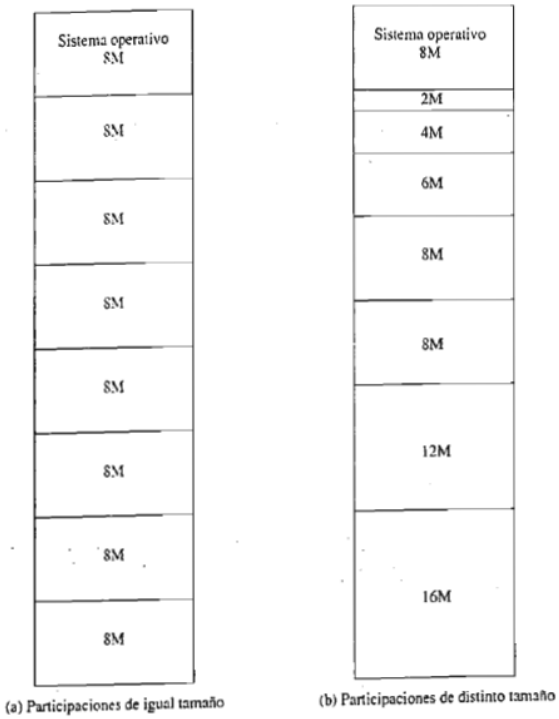
1. Partición estática
2. Partición dinámica
3. Paginación simple
4. Segmentación simple
5. Memoria virtual paginada
6. Memoria virtual segmentada

**Partición estática**

La MP se divide en un conjunto de particiones estáticas durante la generación del sistema (en el momento de arranque del SO). Un proceso se puede cargar en una partición de menor o igual tamaño. Hay dos alternativas de partición estática:

A. Emplear particiones estáticas de igual tamaño.

B. Emplear particiones estáticas de distinto tamaño.



(a) Participaciones de igual tamaño (b) Participaciones de distinto tamaño

Las particiones estáticas de igual tamaño plantean dos dificultades:

- El programa puede ser demasiado grande para caber en la partición. El programador deberá crear el programa mediante superposiciones.

- Uso de la MP extremadamente ineficiente. Cualquier programa, sin importar lo pequeño que sea, ocupara una partición completa. Esto genera **fragmentación interna**.

Ambos problemas pueden reducirse, aunque no eliminarse, por medio del empleo de particiones estáticas de distinto tamaño.

*Nota:*
La *fragmentación* es el desperdicio de memoria. Hay 2 tipos de fragmentación:

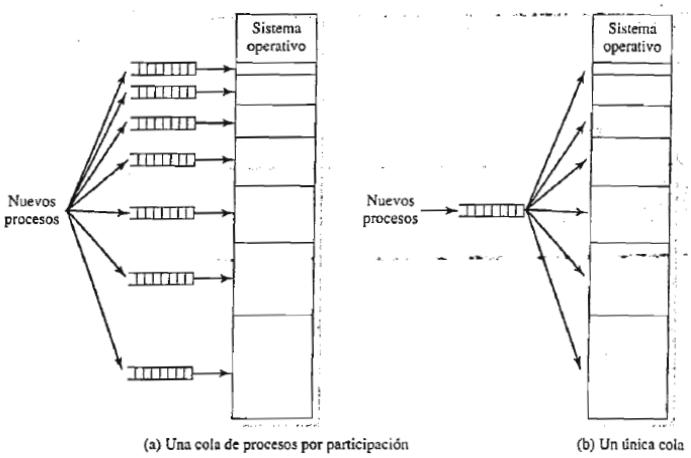
- *Fragmentación interna*: Es el desperdicio de memoria que se produce cuando un programa ocupa una partición de memoria más grande de la que necesita.
- *Fragmentación externa*: Es el desperdicio de memoria que se produce cuando hay memoria disponible, pero esta no se puede usar, porque es menor de la que requiere el programa.

*Algoritmos de ubicación*
Con particiones del mismo tamaño, la ubicación de un proceso en la memoria es trivial. Mientras haya alguna partición libre, puede cargarse un proceso en esa partición y no importa la partición que se use (ya que todas son del mismo tamaño). Si están todas las particiones ocupadas, sacar a un proceso de ella es una cuestión de planificación de procesos.

Con particiones de distinto tamaño hay 2 formas de asignar los procesos a las particiones:

A. La más fácil, es asignar cada proceso a la partición más pequeña que quepa, de forma que los procesos siempre están asignados de forma que se minimiza la memoria desaprovechada dentro de cada partición. Sin embargo, esto no es conveniente porque pueden haber particiones sin usar, y que dicha partición no sea asignada al proceso porque esta no es la partición más pequeña en la que quepa.

B. Asignar a cada proceso la partición más pequeña *disponible* que quepa.



(a) Una cola de procesos por participación (b) Un única cola

Resumen:

| **Partición Estática** | **<u>Ventajas</u>** | **<u>Desventajas</u>** |
|---|---|---|
| La MP se divide en un conjunto de particiones estáticas durante la generación del sistema. Un proceso se puede cargar en una partición de menor o igual tamaño | Sencillo de implementar.<br><br>Poca sobrecarga del SO. | Fragmentación interna<br><br>El número de procesos activos es fijo. |

**<u>Particiones dinámicas</u>**

Las particiones se crean dinámicamente, de forma que cada proceso se carga en una partición de exactamente el mismo tamaño que el proceso. Es decir se le asigna tanta memoria como necesita y no más.

Este método comienza bien, pero finalmente, desemboca en una situación en la que hay un gran número de huecos pequeños de memoria. Conforme pasa el tiempo, la memoria comienza a estar más fragmentada y su rendimiento decae. Se produce **fragmentación externa**.

Una técnica para vencer a la fragmentación externa, es **la compactación**. De vez en cuando, el sistema operativo desplaza los procesos para que estén contiguos, de forma que toda la memoria libre quede juta en un bloque. El problema es que es un procedimiento que consume tiempo, por lo que desperdicia tiempo del procesador.

*Algoritmo de ubicación*

En el esquema de particiones dinámicas se pueden considerar 3 algoritmos. Los 3 se limitan a elegir entre los bloques de memoria libres que son mayores o iguales que el proceso a cargar.

- *Mejor ajuste (best-fit)*
  Elige el bloque de tamaño más próximo al solicitado.

- *Primer ajuste (first-fit)*
  Recorre la memoria desde el principio y escoge el primer bloque disponible suficientemente grande.

- *Siguiente ajuste (next-fit)*
  Recorre la memoria desde la última ubicación y elige el siguiente bloque disponible que sea suficientemente grande.

El mejor método aplicable dependerá de la secuencia exacta de *intercambios de procesos que ocurran y del tamaño de estos procesos*.

| ***Primer ajuste*** | ***Siguiente Ajuste*** | ***Mejor ajuste*** |
|---|---|---|
| Sencillo.<br>El mejor y más rápido. | *Genera peores resultados que el primer ajuste.*<br><br>*Necesita una compactación más frecuente que el primer ajuste* ya que el bloque de memoria libe más grande, que suele aparecer al final del espacio de memoria, se divide rápidamente en fragmentos pequeños. | *Proporciona en general los peores resultados*<br><br>Al elegir el bloque de tamaño más aproximado al tamaño del proceso, genera rápidamente huecos muy pequeños. Así pues debe compactar con más frecuencia que los otros 2 algoritmos. |

**Sistema de colegas**

Los bloques de memoria disponibles son de tamaño $2^K$, para valor de K tal que $L \leq K \leq U$ y donde:

- $2^L$ = tamaño de bloque más pequeño asignable.

- $2^U$ = tamaño de bloque más grande asignable (generalmente es el tamaño de la memoria entera disponible para asignar).

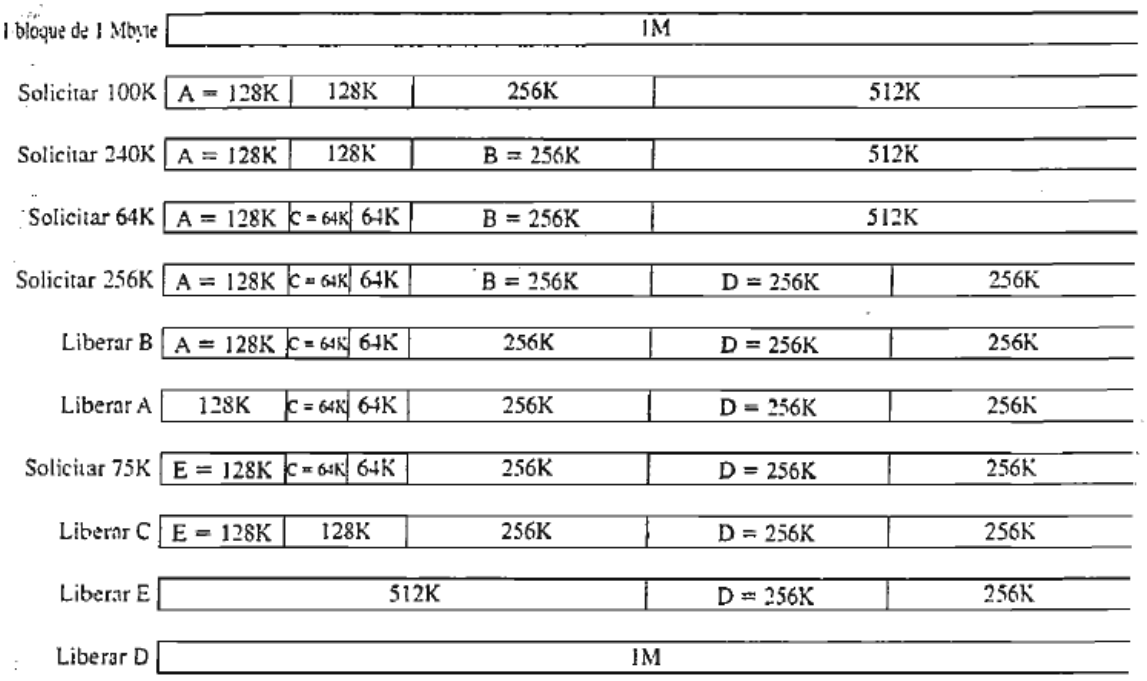



**Figura 7.6.** Ejemplo de sistema *Buddy*.

La figura anterior muestra un ejemplo utilizando un bloque inicial de 1MB. La primera solicitud A de 100KB necesitara un boque de 128KB. El bloque inicial se divide en 2 colegas de 512KB. El primero de estos se divide en 2 de 256KB y el primero de estos se divide en 2 de 128K.

El sistema de colegas es un equilibrio razonable para superar las desventajas de los esquemas de partición fija y variable.

**Reubicación**

Un proceso puede ocupar diferentes particiones de memoria a lo largo de su vida. De esta manera las ubicaciones (de las instrucciones y datos) referenciadas por un proceso no son fijas, sino que cambian cada vez que se intercambia o desplaza un proceso. Para resolver este problema, se realiza una distinción entre varios tipos de direcciones.

- Dirección lógica: dirección generada por la CPU que hace referencia a una posición de memoria independiente de la asignación actual de datos a la memoria.

- Dirección relativa: posición relativa a algún punto conocido, normalmente el principio del programa.

- Dirección física, es una posición real en la MP.

Cuando un proceso pasa a estado ejecutando, se carga un registro especial del procesador, denominado registro

base, con la dirección más baja en la memoria principal del proceso. Existe también un registro límite que indica la posición final del programa. A lo largo de la ejecución del proceso se encuentran direcciones relativas. Cada una de estas direcciones relativas pasa por 2 etapas de manipulación en el procesador. En primer lugar, se suma el valor del registro base a la dirección relativa para obtener una dirección absoluta. Esta dirección absoluta obtenida se compara con el valor del registro límite. Si la dirección está dentro de los límites, se puede proceder a la ejecución de la instrucción. En otro caso, se genera una interrupción del SO, que debe responder al error de algún modo

Este procedimiento se conoce como **carga dinámica en tiempo de ejecución** y *permite a los programas cargarse y descargarse de memoria a lo largo de su ejecución*. También proporciona una medida de protección: cada imagen de proceso está aislada por el contenido de los registros base y limite y es segura contra accesos no deseados por parte de otros procesos.

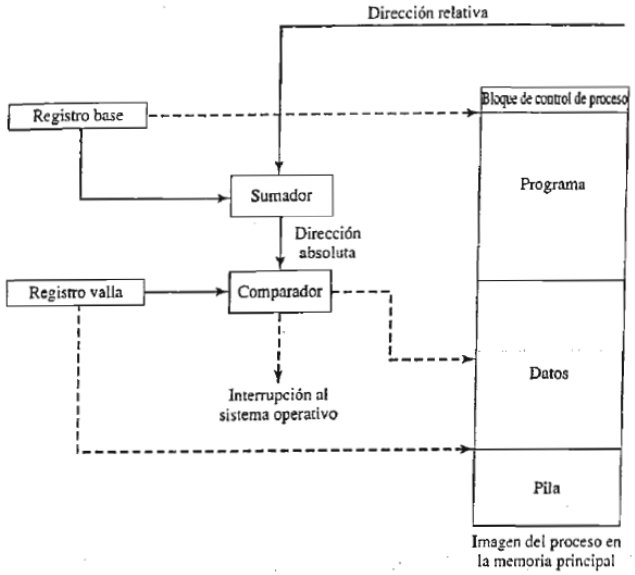



**Figura 7.8.** Soporte hardware para la reubicación.

*Nota*:

La dirección absoluta no se calcula hasta a se ejecute una instrucción. Para asegurar que esta función no degrade el rendimiento, debe realizarse por medio de un hardware especial del procesador, en vez de por software.

Este hardware se denomina **unidad de gestión de memoria** (*MMU, memory management unit*).

El programa de usuario maneja direcciones lógicas y el hardware de conversión de memoria convierte esas direcciones lógicas en direcciones físicas.

*Nota*:

La vinculación de instrucciones y datos a direcciones de memoria puede realizarse en tres etapas diferentes:

- *Compilación*: Si se conoce a priori la posición que va a ocupar un proceso en la memoria se puede generar código absoluto con referencias absolutas a memoria; si cambia la posición del proceso hay que recompilar el código.

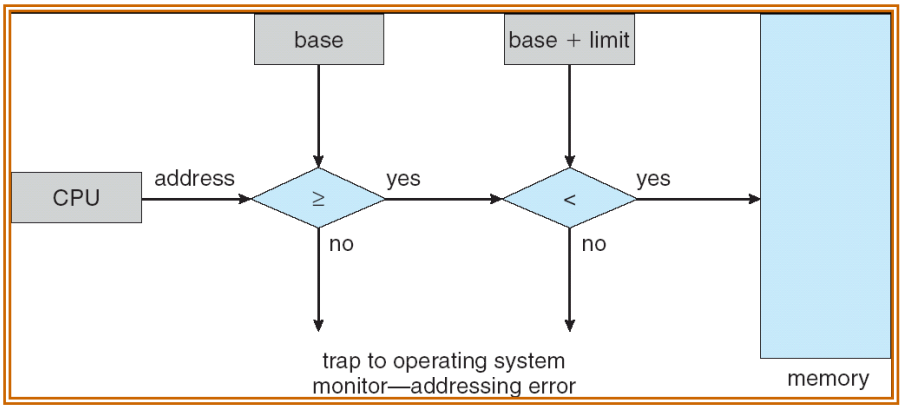
- *Carga*: Si no se conoce la posición del proceso en memoria en tiempo de compilación se debe generar

***código reubicable;*** el compilador no generará direcciones reales de memoria principal sino direcciones relativas a algún punto conocido, como el comienzo del programa. Al momento de la carga del proceso a memoria principal, se convierten todas las direcciones relativas a direcciones absolutas. El problema surge que en un sistema multiprogramado, el proceso puede ser suspendido luego de la carga en memoria, y vuelto a ser cargado en otra posición diferente.

- *Ejecución*: Si no se conoce la posición del proceso en memoria en tiempo de compilación se debe generar código reubicable. Es lo que se conoce como **carga dinámica en tiempo de ejecución.**

*Nota*:
Las direcciones lógicas y físicas son iguales en los esquemas de vinculación en tiempo de compilación y de carga; pero difieren en el esquema de vinculación en tiempo de ejecución.

*Nota*:
Un par de registros **base** y **límite** definen el espacio de direcciones lógicas. También se llaman registros de reubicación y se usan para proteger los procesos de usuario unos de otros.




**Paginación**
Mediante la paginación simple, la memoria principal se encuentra dividida en trozos iguales de tamaño fijo, denominados **marcos**. A su vez, cada proceso está divido también en trozos de tamaño fijo y del mismo tamaño que los de memoria, denominados **páginas**. El termino marco se utiliza porque un marco puede mantener o encuadrar una página de datos. Es decir, cuando se introduce un proceso en la memoria, se cargan todas sus páginas en los marcos libres.

*El espacio desaprovechado en la memoria para cada proceso por fragmentación consta solo de una fracción de la última página del proceso. Además, no hay fragmentación externa*.

El sistema operativo mantiene una lista de los marcos libres. Cuando llega el momento de cargar un proceso en memoria, el SO busca los marcos libres necesarios para cargar todas las páginas del proceso. No necesariamente los marcos deben ocupar una posición contigua de memoria. Para lograr esto, el SO operativo mantiene una **tabla de páginas** para cada proceso. Cada tabla de página contiene una entrada por cada página del proceso. Y

en cada entrada se encuentra el número de marco de la memoria principal que alberga la página correspondiente.

Al igual que la partición simple, el procesador también realiza la traducción de direcciones lógicas a físicas. En el caso de la partición simple el programa contenía direcciones relativas al comienzo del programa. En la paginación, el programa contiene direcciones lógicas que constan de un número de página y de un desplazamiento dentro de la página. Para hacer la traducción, el procesador debe acceder a la tabla de páginas del proceso actual. Dada una dirección lógica (número de página, desplazamiento), el procesador emplea la tabla de páginas para obtener una dirección física (número de marco, desplazamiento).

La diferencia entre la paginación simple y la partición estática, es que con la paginación, las particiones son algo más pequeñas (por lo que la fragmentación interna será menor) y que un programa puede ocupar más de una partición y estas o tienen por qué ser contiguas.

*Aplicación correcta del esquema de paginación*

El tamaño de la página y por lo tanto, el tamaño del marco, debe haber una potencia de 2.
Un ejemplo donde se emplean direcciones de 16 bits y el tamaño de página es de 1K = 1024 bytes. Con un tamaño de página de 1K, se necesitan 10 bits para el campo desplazamiento ($2^{10}$ = 1024 bytes). De este modo, un programa puede estar formado por un máximo de $2^6$ = páginas de 1KB cada una.
Si se tiene una dirección lógica = 0000010111011110, se corresponde con el número de página 1 (000001) y desplazamiento 478 (0111011110). Suponiendo se fija en la entra 1 de la tabla de página y dicha entrada contiene el marco 6 = 000110 en binario. Entonces la dirección física es el marco 6 y el desplazamiento 478 = 0001100111011110.

**Segmentación**

Otro modo de subdividir el programa es la segmentación. En este caso, el programa se divide en un conjunto de **segmentos**. No es necesario que todos los segmentos de todos los programas tengan la misma longitud, aunque existe una longitud máxima de segmento. En este esquema, una dirección lógica consta de 2 partes, un número de segmento y un desplazamiento.

Este esquema resulta similar a la partición dinámica. La diferencia, radica en que, con segmentación un programa puede ocupar más de una partición y estas no tienen por qué estar contiguas.

La segmentación no sufre de fragmentación interna, pero, como en la partición dinámica, sufre de fragmentación externa. Aunque esta será menor.

Mientras que la paginación es transparente al programador, la segmentación es visible.

Para la traducción de direcciones lógicas a físicas, al igual que en la paginación, el SO hará uso de una **tabla de**

**segmentos** para cada proceso y una lista de bloques libres en la memoria principal. Cada entrada a la tabla de segmentos tendrá que contener la dirección de comienzo del segmento correspondiente de la memoria principal.

## Capitulo: Memoria Virtual

No es necesario que todas las páginas o todos los segmentos de un proceso se encuentren en la memoria principal durante la ejecución.

Supongamos que se tiene que traer un nuevo proceso de memoria. El sistema operativo comienza trayendo únicamente una o dos porciones, que incluye la porción inicial del programa y la porción inicial de datos sobre la cual acceden las primeras instrucciones acceden. *Esta parte del proceso que se encuentra realmente en la memoria principal para, cualquier instante de tiempo, se denomina* **conjunto residente** *del proceso*. Cuando el proceso está ejecutándose, las cosas irán perfectamente mientras que todas las referencias a la memoria se encuentren dentro del conjunto residente. Usando una tabla de segmentos o páginas, el procesador siempre es capaz de determinar si esto es así o no. Si el procesador encuentra una dirección lógica que no se encuentra en la memoria principal, generará una interrupción indicando un fallo de acceso a la memoria. El sistema operativo coloca al proceso interrumpido en un estado de bloqueado y toma el control. Para que la ejecución de este proceso pueda reanudarse más adelante, el sistema operativo necesita traer a la memoria principal la porción del proceso que contiene la dirección lógica que ha causado el fallo de acceso. Con este fin, el sistema operativo realiza una petición de E/S, una lectura a disco. Después de realizar la petición de E/S, el sistema operativo puede activar otro proceso que se ejecute mientras el disco realiza la operación de E/S. Una vez que la porción solicitada se ha traído a la memoria principal, una nueva interrupción de E/S se lanza, dando control de nuevo al sistema operativo, que coloca al proceso afectado de nuevo en el estado Listo.

Esta nueva estrategia de no cargar todo el proceso en memoria conduce a mejorar la utilización del sistema ya que:

1. **Pueden mantenerse un mayor número de procesos en memoria principal.**

2. **Un proceso puede ser mayor que toda la memoria principal.** Con la memoria virtual basada en paginación o segmentación, el sistema operativo automáticamente carga porciones de un proceso en la memoria principal cuando éstas se necesitan.

La memoria virtual permite una multiprogramación muy efectiva que libera al usuario de las restricciones excesivamente fuertes de la memoria principal.

### Memoria virtual y Proximidad

Se puede hacer un mejor uso de la memoria cargando únicamente unos pocos fragmentos. Entonces, si el programa hace referencia a un dato que se encuentra en una porción de memoria que no está en la memoria principal, entonces se dispara una interrupción. Éste indica al sistema operativo que debe conseguir la porción deseada.

Así, en cualquier momento, sólo unas pocas porciones de cada proceso se encuentran en memoria, y por tanto se pueden mantener más procesos alojados en la misma. Además, se ahorra tiempo porque las porciones del proceso no usadas no se cargan ni se descargan de la memoria.

Sin embargo, el sistema operativo debe saber cómo gestionar este esquema. Cuando el sistema operativo traiga una porción a la memoria, debe expulsar otra. Si elimina una porción justo antes de que vaya a ser utilizada, deberá recuperar dicha porción de nuevo casi de forma inmediata. Demasiados intercambios de fragmentos lleva a una condición denominada **hiperpaginación** (*thrashing*): el procesador consume más tiempo intercambiando fragmentos que ejecutando instrucciones de usuario. Para evitar esto, el SO trata de adivinar, en base a la historia reciente, qué porciones son menos probables de ser utilizadas en un futuro cercano. Esto se basa en el **principio de cercanía**. Este principio indica que las referencias al programa y a los datos dentro de un proceso tienden a agruparse. Por tanto, se resume que sólo unas pocas porciones del proceso se necesitarán a lo largo de un periodo de tiempo corto.

**Paginación**

Para la memoria virtual basada en el esquema de paginación también se necesita una tabla de páginas por proceso. En este caso, debido a que sólo algunas de las páginas de proceso se encuentran en la memoria principal, se necesita un bit en cada entrada de la tabla de páginas para indicar si la correspondiente página está presente (P) en memoria principal o no lo está. Si el bit indica que la página está en memoria, la entrada también debe indicar el número de marco de dicha página.

Otro bit de control necesario en la entrada de la tabla de páginas es el bit de modificado (M), que indica si los contenidos de la correspondiente página han sido alterados desde que la página se cargó por última vez en la memoria principal. Si no ha habido cambios, no es necesario escribir la página cuando llegue el momento de reemplazarla por otra página en el marco de página que actualmente ocupa. Pueden existir también otros bits de control en estas entradas.

*Estructura de la tabla de páginas*

Debido a que la tabla de páginas es de longitud variable dependiendo del tamaño del proceso, la cantidad de memoria demandada por las tablas de página únicamente puede ser inaceptablemente grande.

Por esto, la mayoría de los esquemas de memoria virtual almacenan las tablas de páginas también en la memoria virtual, en lugar de en la memoria real. Esto representa que las tablas de páginas están sujetas a paginación igual que cualquier otra página. Cuando un proceso está en ejecución, al menos *parte* de su tabla de páginas debe encontrarse en memoria, incluyendo la entrada de tabla de páginas de la página actualmente en ejecución.

Algunos procesadores utilizan un ***esquema de dos niveles*** para organizar las tablas de páginas de gran tamaño.

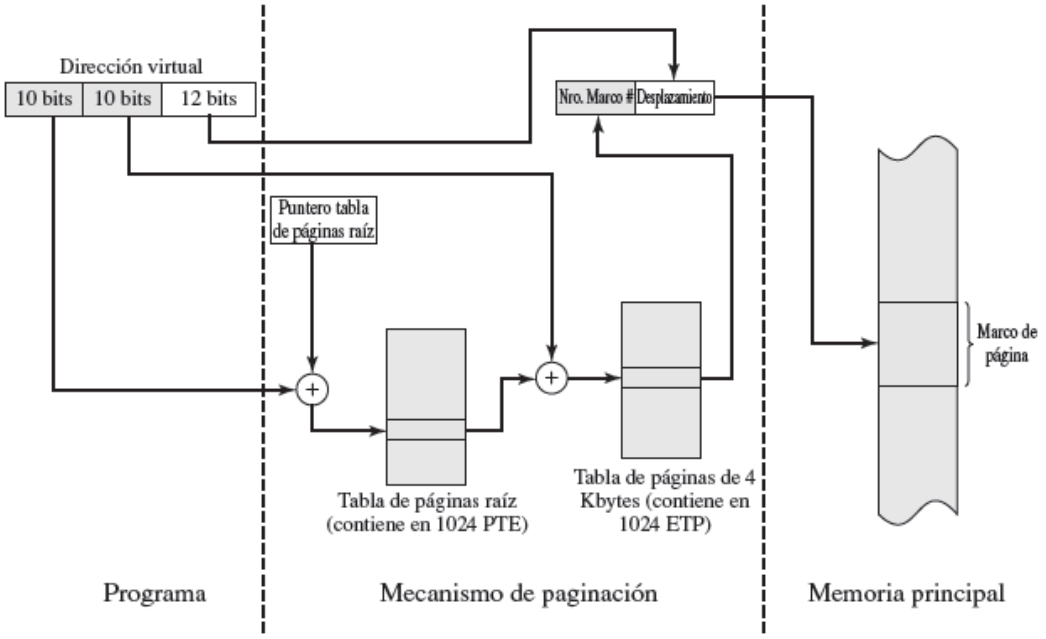



**Figura 8.5.** Traducción de direcciones en un sistema de paginación de dos niveles.

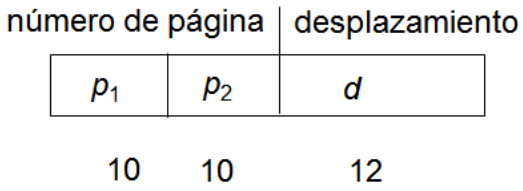
Un ejemplo de un esquema típico de dos niveles que usa 32 bits ($2^{32}$) para la dirección con tamaño de páginas de 4K, tiene una dirección lógica formada por:

- Un desplazamiento de $2^{12}$ (4K)
- Un número de páginas de 20 bits.

Ya que la tabla de páginas está paginada y cada entrada de la tabla de páginas ocupa 4 bytes, el número de página es de nuevo dividido en:

- Un número de página de 10 bits
- Un desplazamiento de 10 bits

Por tanto, una dirección lógica tiene el siguiente aspecto:




Donde $P_1$ es un índice a la tabla de páginas raíz y $P_2$ es un desplazamiento en la segunda tabla de páginas.

Una estrategia alternativa al uso de tablas de páginas de uno o varios niveles es el uso de la estructura de tabla de **páginas invertida**.

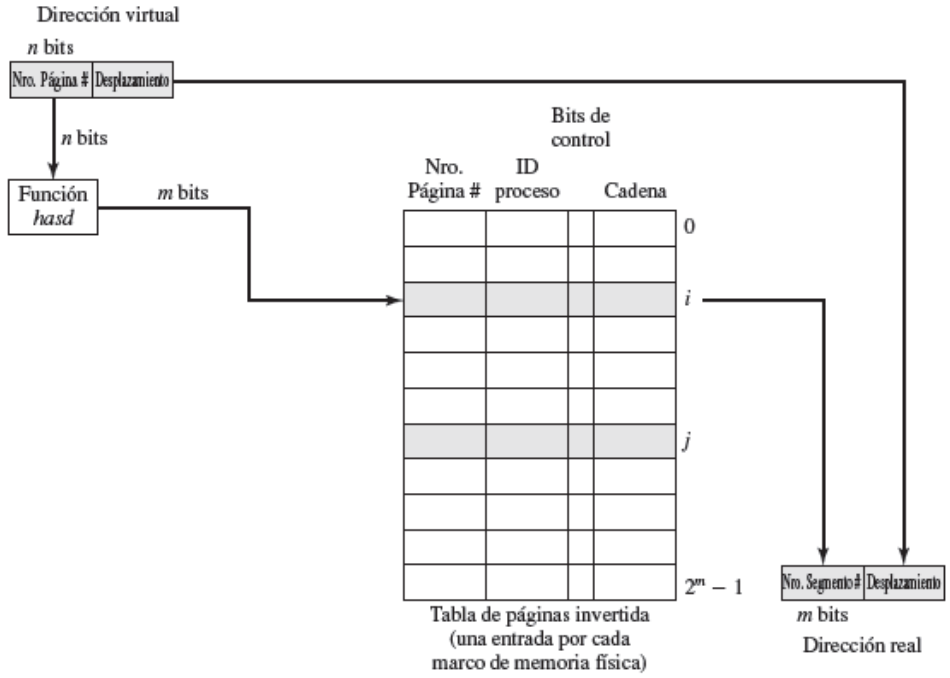



**Figura 8.6.** Estructura de tabla de páginas invertida.

La tabla de páginas invertida contiene un registro por cada marco de página real en lugar de un registro por cada página virtual. La estructura de la tabla de páginas se denomina invertida debido a que se indexan sus entradas de la tabla de páginas por el número de marco en lugar de por el número de página virtual. Para un tamaño de memoria física de $2^m$ marcos, la tabla de páginas invertida contiene $2^m$ entradas, de forma que la entrada en la posición i-esima se refiere al marco i.

Las entradas contienen la dirección virtual de la página almacenada en el marco *con información sobre el proceso* que la posee.

Disminuye la memoria necesaria para almacenar cada tabla de páginas.

*Buffer de traducción adelantada (almacenamiento intermedio)*

En principio, toda referencia a la memoria virtual puede causar dos accesos a memoria física: uno para buscar la entrada la tabla de páginas apropiada y otro para buscar los datos solicitados.

Por esta situación, la mayoría de esquemas de la memoria virtual utilizan una *cache especial* hardware de alta velocidad para las entradas de la tabla de página **TLB** (translation lookaside buffer). Esta contiene aquellas entradas de la tabla de páginas que han sido usadas de forma más reciente. Dada una dirección virtual, el procesador primero examina la TLB, si la entrada de la tabla de páginas solicitada está presente (acierto en TLB), entonces se recupera el número de marco y se construye la dirección real. Los estudios sobre la TLB han demostrado que este esquema significa una importante mejora del rendimiento.

*Tamaño de página*

Cuanto menor es el tamaño de la página, menor cantidad de fragmentación interna. Por otro lado, cuanto menor es la página, mayor número de páginas son necesarias para cada proceso. Un mayor número de páginas por proceso significa también que el tamaño de las tablas de páginas será mayor. Esto puede significar que una gran parte de las tablas de páginas de los procesos activos debe estar en memoria virtual y no en memoria principal. Esto puede provocar 2 fallos de página para una única referencia a memoria:

- Para traer la parte necesaria de la tabla de páginas.
- Para traer la página del proceso.

Por principio de proximidad, si el tamaño de página es muy pequeño habrá gran cantidad de páginas disponibles en la memoria principal para cada proceso. Después de un tiempo, las páginas en memoria contendrán las partes de los procesos a las que se ha hecho referencia de forma reciente. De esta forma, la tasa de fallos de página debería ser baja. A medida que el tamaño de páginas se incrementa, la página en particular contendrá información más lejos de la última referencia realizada. Así pues, el efecto del principio de proximidad se debilita y la tasa de fallos de página comienza a crecer.

Para un tamaño determinado de una TLB, a medida que el tamaño del proceso crece y la proximidad de referencias decrece, el índice de aciertos en TLB se va reduciendo. Bajo estas circunstancias, la TLB se puede convertir en el cuello de botella del rendimiento.

**Paginación y Segmentación combinadas**

La paginación es transparente al programador y elimina la fragmentación externa, y por tanto proporciona un uso eficiente de la memoria principal.

La segmentación sí es visible al programador, incluyendo la posibilidad de manejar estructuras de datos que crecen, modularidad, y dar soporte a la compartición y a la protección. A su vez, elimina la fragmentación interna.

En un sistema con paginación y segmentación combinadas, el espacio de direcciones de un usuario se divide en varios segmentos según el criterio del programador. Cada segmento se vuelve a dividir en varias páginas de tamaño fijo, que tienen la misma longitud que un marco de memoria principal.
*Asociada a cada proceso existe una tabla de segmentos y varias tablas de páginas, una por cada uno de los segmentos.*

**Protección y seguridad**

*La segmentación proporciona protección y compartición*.

Cada entrada en la tabla de segmentos incluye la longitud así como la dirección base, por lo que un programa NO puede acceder a una posición de memoria más allá de los límites del segmento. Para conseguir compartición, es posible que un segmento se encuentre referenciado desde las tablas de segmentos de más de un

proceso. Los mecanismos están disponibles en los sistemas de paginación. Sin embargo, en este caso la estructura de páginas de un programa y los datos no son visible para el programador, haciendo que la especificación de la protección y los requisitos de compartición sean más difíciles.

**Políticas de lectura**

La política de lectura (fetch) está relacionada con la decisión de cuando se debe cargar una agina en la memoria principal:

- *Paginación por demanda*
  Una página se trae a memoria sólo cuando se hace referencia a una posición en dicha página. Al principio se producirá un aluvión de fallos de página, pero luego al estar cargadas varias páginas utilizadas recientemente, por el principio de proximidad, la situación se estabilizará y el número de fallos será muy bajo.

- *Paginación previa*
  Se traen a memoria también otras páginas, diferentes de la que ha causado el fallo de página. Es mucho más eficiente traer a la memoria un número de páginas contiguas de una vez, en lugar de traerlas una a una a lo largo de un periodo de tiempo más amplio (por la operación de E/S a disco). Por supuesto, esta política es ineficiente si la mayoría de las páginas que se han traído no se referencian a posteriori.

**Políticas de ubicación**

La política de ubicación determina en qué parte de la memoria real van a residir las porciones de un proceso.

En los *sistema de segmentación* puros, la política de ubicación es un aspecto de diseño muy importante (políticas del estilo mejor ajuste, primer ajuste).

Para sistemas que usan *paginación pura o paginación combinada con segmentación*, la ubicación es irrelevante debido a que el hardware de traducción de direcciones y el hardware de acceso a la memoria principal pueden realizar sus funciones en cualquier combinación de página-marco con la misma eficiencia.

**Políticas de reemplazo**

La política de reemplazo determina entre el conjunto de páginas consideradas, cuál es la página especifica que debe elegirse para el reemplazo. Todas las políticas reemplazo tienen como objetivo que la página que va a eliminarse sea aquella que tiene menos posibilidades de volver de ser referencia en un futuro cercano.
Cuanto más elaborada y sofisticada es una política de reemplazo, mayor va a ser la sobrecarga a nivel software y hardware para implementarla.

*Bloqueo de marcos*

Cuando un marco está bloqueado, la página actualmente almacenada en dicho marco no puede reemplazarse. Gran parte del núcleo del sistema operativo se almacena en marcos que están bloqueados,

así como otras estructuras de control claves.

*Algoritmos*

Existen ciertos algoritmos básicos que se emplean para la selección de una página a reemplazar:

1. Óptima
2. Usada menos recientemente (LRU)
3. Primero entrar, primero en salir (FIFO)
4. Reloj

**Óptima** selecciona para reemplazar la página que tiene que esperar una mayor cantidad de tiempo hasta que se produzca la referencia siguiente. Genera el menor número de fallos de página pero es *imposible de implementar* ya que requiere que el SO tenga un conocimiento exacto de los sucesos futuros.

**LRU** reemplaza la página de memoria que ha sido referenciada desde hace más tiempo. LRU proporciona unos resultados casi tan buenos como la política óptima. El problema con esta alternativa es la dificultad en su implementación. Es una opción costosa en cuanto a la sobrecarga del sistema.

**FIFO** trata los marcos de página ocupados como si se tratase de un buffer circular, y las páginas se remplazan mediante una estrategia cíclica de tipo round-robin. Sencillo de implementar pero su rendimiento es relativamente pobre ya que reemplaza la página que lleva en memoria más tiempo y por lo general que estos sean las zonas utilizadas de forma intensiva durante todo el tiempo de vida del proceso.

**Reloj** asocia un bit a cada marco, denominado bit de uso. Cuando una página se trae por primera vez a la memoria, el bit de usado de dicho marco se pone a 0. Cuando se hace referencia a la página posteriormente, el bit se pone en 1. El conjunto de marcos a ser reemplazados se considera como un buffer circular con un puntero asociado. Cuando llega el momento de reemplazar una página, el sistema operativo recorre el buffer para encontrar un marco con su bit de usado a 0. Cada vez que encuentra un marco con el bit de usado a 1, se reinicia este bit a 0 y se continúa. Si alguno de los marcos del buffer tiene el bit de usado a 0 al comienzo de este proceso, el primero de estos marcos que se encuentre se seleccionará para reemplazo. Si todos los marcos tienen el bit a 1, el puntero va a completar un ciclo completo a lo largo del buffer, poniendo todo los bits de usado a 0, parándose en la posición original, reemplazando la página en dicho marco. *Véase que esta política es similar a FIFO, excepto que, en la política del reloj, el algoritmo saltará todo marco con el bit de usado a 1. Se aproxima al rendimiento LRU sin introducir mucha sobrecarga.*

*El algoritmo del reloj puede hacerse más potente incrementando el número de bits que utiliza (y cuantos menos bits se utilicen se degenera en FIFO).*Se asocia un bit de modificado que combinado con el bit de usado, en resumen, el algoritmo de reemplazo de páginas da vueltas a través de todas las páginas del

buffer buscando una que no se haya modificado desde que se ha traído y que no haya sido accedida recientemente. Esta página es una buena opción para reemplazo y tiene la ventaja que, debido a que no se ha modificado, no necesita escribirse de nuevo en la memoria secundaria.

**Almacenamiento intermedio de páginas**

Para mejorar el rendimiento, una página remplazada no se pierde sino que se asigna a una de las dos siguientes listas: la lista de páginas libres si la página no se ha modificado, o la lista de páginas modificadas si lo ha sido. La página no se mueve físicamente de la memoria, en su lugar, se suprime su entrada en la tabla de páginas y se pone en la lista de páginas libres o modificadas.

.

En efecto, la lista de páginas modificadas y libres actúa como una cache de páginas (la página que se va a reemplazar se mantiene en la memoria).

**Gestión de conjunto residente**

*Asignación Fija* → proporciona un número fijo de marcos de memoria principal disponibles para ejecución. Siempre que se produzca un fallo de página del proceso en ejecución, la página que se necesite reemplazará una de las páginas del proceso.

*Asignación variable* → permite que se reserven un número de marcos por proceso que puede variar a lo largo del tiempo de vida del mismo. Muchos fallos de páginas, se asignan marcos adicionales al proceso. Pocos fallos de página, se saca marcos que no utiliza. La dificultad de esta estrategia se deben a que el SO debe saber cuál es el comportamiento del proceso activo por lo que produce una sobrecarga en el sistema.

**Alcance del reemplazo**

La política de reemplazo surge cuando se produce un fallo de página. Esta puede ser:

- *Reemplazo de Alcance local*
  Se escoge entre la paginas residentes del proceso que origino el fallo.

- *Reemplazo de Alcance Global*
  Se escoge entre todas las páginas de la memoria para reemplazo.

NO es posible aplicar *asignación fija* y una política de *reemplazo global*.

**Políticas de vaciado**

Determina el momento en el que hay que escribir en la memoria secundaria una página modificada.

- *Vaciado por demanda*

Una página se escribirá en la memoria secundaria solo cuando haya sido elegida para reemplazarse.

- *Vaciado previo*
  Escribe las páginas modificadas antes de que se necesiten sus marcos, de forma que las páginas puedan escribirse por lotes.

Las 2 políticas son ineficientes, ya que el vaciado por demanda la escritura de una página es anterior a la lectura de una nueva página, por lo que un proceso que sufra una falla de página pueda tener que esperar 2 transferencias de páginas antes de desbloquearse. Con el vaciado previo, las páginas pueden ser nuevamente modificadas luego de ser escritas en memoria secundaria.

Solución: incorporar *almacenamiento intermedio de páginas* → Se desactivan las políticas de reemplazo y vaciado para dar lugar al almacenamiento intermedio.

**Control de carga**

Determinar el grado de multiprogramación: pocos procesos en la memoria principal, entonces menor aprovechamiento de la CPU. Si hay demasiados procesos, el tamaño medio del conjunto residente de cada proceso no será el adecuado y se producirán frecuentes fallos de páginas. El resultado es la *hiperpaginación*.

# Capitulo: Entrada y Salida

Hay 3 técnicas para realizar la E/S:

1. **E/S programada**: el procesador emite una orden de E/S de parte de un proceso a un módulo de E/S; el proceso espera hasta que se complete la operación antes de continuar (espera activa, se queda preguntando al dispositivo de E/S si ya está listo para realizar la E/S - Bit de ocupado/desocupado de la controladora).

2. **E/S dirigida por interrupción**: el procesador emite una orden de E/S de parte de un proceso a un módulo de E/S, el procesador continua la ejecución de las instrucciones siguientes mientras el modulo E/S se prepara para la E/S. Cuando está listo para realizar la operación de E/S manda una interrupción y el procesador la capta (entre medio de cada instrucción verifica la línea de interrupciones). El procesador interrumpe lo que estaba haciendo y comienza la transferencia.

3. **Acceso directo a memoria:** un módulo de DMA, que es un procesador de propósito especial, controla el intercambio de datos entre la memoria principal y un módulo de E/S. El procesador envía una petición de transferencia al DMA. El DMA interrumpe al procesador solo para indicarle que la transferencia ya termino.

## Acceso directo a la memoria

EL DMA transfiere datos desde y hacia la memoria a través del bus del sistema. Normalmente el DMA debe usar el bus solamente cuando el procesador no lo necesite o debe obligar al procesador a que se suspenda temporalmente su operación. Esta última técnica es más común y se denomina *robo de ciclos de bus* donde el procesador debe esperar un ciclo de bus mientras la unidad de DMA transfiere una palabra. *El efecto final es que el procesador ejecuta más lentamente*. *Sin embargo, para una transferencia de varias palabras, el DMA es*

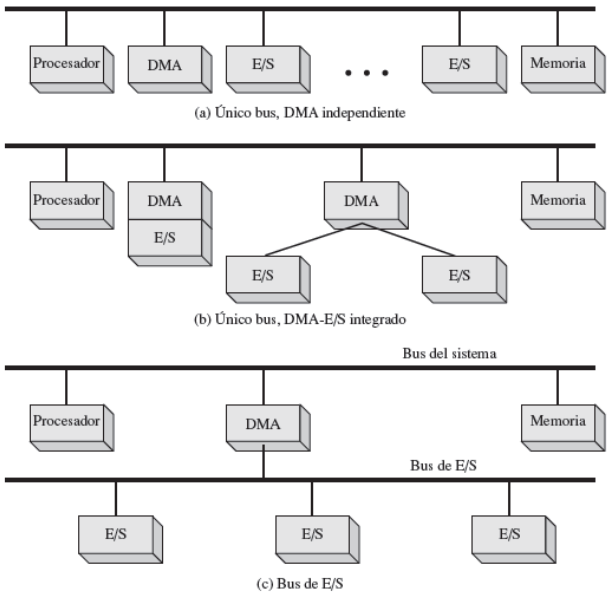



**Figura 11.3.** Configuraciones de DMA alternativas.

*mucho más eficiente que la E/S programada o la dirigida*.
En las alternativas de configuraciones de DMA (imagen de arriba) se ve que:

- En el caso A, EL DMA y los dispositivos de E/S utilizan el bus del sistema por lo que la transferencia de una palabra consume 2 ciclos de bus: solicitud de transferencia + la transferencia.

- En el caso B y C, se conectan los módulos de E/S al módulo de DMA mediante un bus de E/S, por lo que el bus del sistema solo es utilizado por el DMA para intercambiar datos con la memoria y señales de control con la CPU. El intercambio de datos entre el módulo de DMA y el de E/S tiene lugar fuera del bus del sistema, achicando así el número de ciclos de bus del sistema requeridos.

**Aspectos de diseño en los sistemas operativos**

Las operaciones de E/S constituyen un cuello de botella en los sistemas informáticos ya que la mayoría de los dispositivos de E/S son extremadamente lentos en comparación con la memoria principal y el procesador. Una manera de abordar este problema es con el uso de la multiprogramación, que permite que algunos procesos esperen la E/S mientras otro proceso está ejecutado.

**Almacenamiento intermedio (buffering)**

Un buffer es una memoria intermedia que almacena datos mientras se están transmitiendo entre 2 dispositivos o entre un dispositivo y una aplicación. *Es un amortiguador de velocidades*.

*Si un proceso realiza una E/S sin buffer, este debe quedar residente en la memoria principal, no pudiendo ser expulsado a disco*. Esta condición reduce la oportunidad de usar el intercambio al fijar como residente parte de la memoria principal, lo que conlleva una disminución del rendimiento global del sistema. Asimismo, el dispositivo de E/S queda asociado al proceso durante la duración de la transferencia, no estando disponible mientras tanto para otros procesos.

*Por lo tanto, en un entorno multiprogramado, donde hay múltiples operaciones de E/S, el almacenamiento intermedio es una herramienta que puede incrementar la eficiencia del SO y el rendimiento de los procesos individuales.*

**Doble buffer**

Un proceso puede transferir datos hacia o desde una memoria intermedia mientras el SO vacía o rellena el otro.

**Planificación de discos**

*Tiempo de búsqueda* → tiempo que se tarda en ubicar la cabeza en la pista.

*Retardo de giro* → tiempo que tarda el inicio del sector deseado en llagar hasta la cabeza.

*Tiempo de acceso* → suma del tiempo de búsqueda + el retardo de giro.

El orden en que se leen los sectores del disco tiene un efecto inmenso en el rendimiento de la E/S. Acá entra en juego las políticas de planificación del disco.

*Nota*: Si los pedidos de lectura a disco no llegan en simultáneo, no hay nada que planificar ya que hay un solo pedido a la vez.

Las políticas de planificación son:

- *FIFO (Primero en entrar primero en salir)*: Buen rendimiento si hay pocos procesos que requieren acceso y si muchas de las solicitudes son a sectores agrupados de un archivo. Sino mal rendimiento (como si fuese aleatorio).

- *Prioridad*: *la planificación queda fuera del control del software de gestión de disco. Esta* estrategia no está diseñada para optimizar la utilización del disco sino satisfacer otros objetivos del SO. Favorece a los trabajos cortos, por los que los trabajos mayores pueden tener que esperar excesivamente. Poco favorable para sistemas de bases de datos.

- *LIFO (Ultimo en entrar, primero en salir)*: Puede producir inanición. Buen rendimiento en los sistemas de transacciones ya que conceder el dispositivo al último usuario acarrea pocos movimientos de brazos al recorrer un archivo secuencial.

*Nota*: las siguientes políticas tienen en cuenta la posición actual de la pista.

- SCAN (subo hasta la última pista del disco y bajo): Favorece a los trabajos con solicitudes de pistas cercanas a los cilindros más interiores y exteriores; así como a los últimos trabajos en llegar.
  El primer problema se puede evitar con C-SCAN, mientras el 2do con SCAN N pasos.

*Nota*:
Entre otras políticas (LOOK, CLOOK, FSCAN, SCAN N PASOS, SSTF. Ver practica).

## RAID

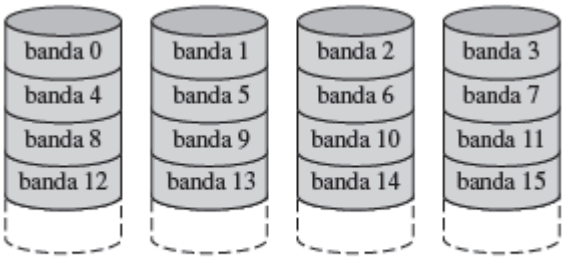
RAID (Vector Redundante de Discos Independientes) es un conjunto de unidades de disco físico vistas por el SO como una sola unidad lógica.

Con múltiples discos que funcionan en forma independiente y en paralelo, las distintas solicitudes de E/S se pueden gestionar en paralelo, siempre que los datos solicitados residan en discos separados. Es más, una única petición de E/S se puede ejecutar en paralelo si el bloque de datos que se pretende acceder está distribuido a lo largo de múltiples discos. *Por lo que RAID tiende a mejora significativamente el rendimiento y fiabilidad del*

*sistema de E/S a disco*.

El sistema de RAID consta de 6 niveles posibles, de 0 a 6. Los niveles de 2 a 4 no se comercializan.

- *RAID 0 (no redundante)*
  No incluye redundancia. Es un RAID "trucho". Todos los datos están distribuidos a lo largo de todo el vector de discos.




(a) RAID 0 (sin redundancia)

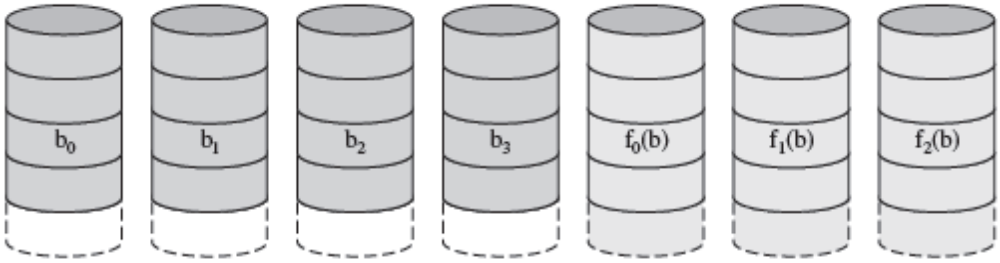
  Los datos están distribuidos en bandas a lo largo de los discos disponibles. El RAID 0 se usa normalmente para aplicaciones que requieren alto rendimiento para datos no críticos.

- *RAID 1 (espejo)*
  Se consigue redundancia por el simple medio de duplicar todos los datos. *Mucha fiabilidad*. *Permite lecturas más rápidas* ya que los datos los puede servir cualquiera de los 2 discos que contienen los datos solicitados. La velocidad de escritura depende de la escritura más lenta de las 2, pero *NO hay penalización de escritura* (escribir información adicional como bit de paridad).
  La principal desventaja es el elevado costo por tener que duplicar toda la información en otro disco.

| Discos = 2N |
|---|

- *RAID 2 (redundancia mediante código Hamming – Acceso paralelo)*
  RAID 2 usa división a nivel de bits. Se utilizan un par de discos para datos y otros para paridad. El número de discos redundantes es proporcional al logaritmo del número de discos de datos.

| Discos = N + M |
|---|




(c) RAID 2 (redundancia mediante código Hamming)

  Uno de sus efectos secundarios es que normalmente *no puede atender varias peticiones simultáneas*,

debido a que por definición cualquier simple bloque de datos se dividirá por todos los miembros del conjunto, residiendo la misma dirección dentro de cada uno de ellos. Así, cualquier operación de lectura o escritura exige activar todos los discos del conjunto.

*Puede conseguir una tasa de transferencia muy alta*.

*Sirve solamente para un entorno donde se produjeran muchos errores de disco*.

- *RAID 3 (paridad por intercalación de bits – Acceso paralelo)*
  Idéntico a RAID 2 pero solamente requiere de 1 disco redundante, sin importar la cantidad de discos. En lugar de un código de corrección de errores, se calcula un bit de paridad simple para el conjunto de bits almacenados en la misma posición en todos los discos de datos.

Diagrama de una configuración RAID 3. Cada número representa un byte de datos; cada columna, un disco.

Discos = N + 1

En el ejemplo del gráfico, una petición del bloque «A» formado por los bytes A1 a A6 requeriría que los tres discos de datos buscaran el comienzo (A1) y devolvieran su contenido. Una petición simultánea del bloque «B» tendría que esperar a que la anterior concluyese.
*Puede conseguir una tasa de transferencia muy alta*. *No puede atender varias peticiones simultáneas*.

- *RAID 4 (paridad por intercalación de bloques – Acceso Independiente)*
  Usa división a nivel de bloques con un disco de paridad dedicado. Permite acceso independiente.

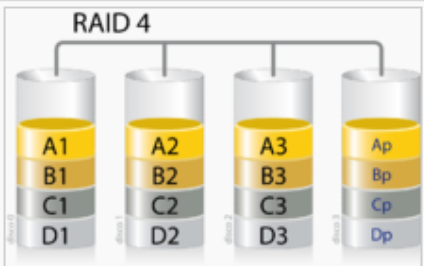


Diagrama de una configuración RAID 4. Cada número representa un bloque un disco.

Discos = N + 1

En ejemplo del gráfico, una petición del bloque «A1» sería servida por el disco 0. Una petición simultánea del bloque «B1» tendría que esperar, pero una petición de «B2» podría atenderse concurrentemente.

- *RAID 5 (paridad por intercalación distribuida de bloques – Acceso independiente)*
  El esquema RAID 5 se organiza de manera similar al RAID 4. La diferencia estriba en que el esquema RAID 5 distribuye las bandas de paridad a través de todos los discos.

  El RAID 5 ha logrado popularidad gracias a su bajo coste de redundancia. Además a diferencia de RAID 4 no tiene cuello de botella al no tener un disco dedicado a redundancia.

Diagrama de una configuración RAID 5.

Discos = N + 1

- *RAID 6 (paridad dual por intercalacion distribuida de bloques – Acceso independiente)*
  En el esquema RAID 6, se realizan dos cálculos de paridad diferentes, almacenándose en bloques separados de distintos discos. Por tanto, un vector RAID 6, cuyos datos de usuario requieran N discos, necesitará:

Discos = N + 2

La ventaja del esquema RAID 6 es que proporciona una extremadamente alta disponibilidad de datos.

*Nota*:
*Acceso paralelo* → todos los discos participan en la ejecución de cada solicitud de E/S. Más apropiados para aplicaciones que requieran tasas altas de transferencia de datos

*Acceso independiente* → cada disco opera independientemente, por lo que se pueden satisfacer en paralelo solicitudes de E/S. Más apropiados para aplicaciones que requieran tasas altas de solicitudes de E/S.

**Memoria cache**

*Memoria pequeña y de alta velocidad, más rápida que la memoria principal, y que se sitúa entre esta y el procesador.* Dicha memoria cache reduce el tiempo medio de acceso a memoria aprovechándose del principio de la proximidad.

*La cache de disco contiene una copia de algunos de los sectores del disco.* Cuando se hace una petición de E/S solicitando un determinado sector, se comprueba si el sector está en la cache del disco. En caso afirmativo, se sirve la petición desde la cache. Debido al fenómeno de la proximidad de referencias, cuando se lee un bloque de datos en la cache para satisfacer una única petición de E/S, es probable que haya referencias a ese mismo bloque en el futuro.

La *diferencia entre cache y buffer* en que el buffer puede almacenar la única copia existen de un elemento de datos, mientras que la cache almacena una copia de un elemento que reside en otro lugar.

**Tipos de E/S**

- *Bloqueante y No bloqueante*
  Bloqueante → Se suspende la ejecución de la aplicación que la solicito. Cuando se completa la operación vuelve a la cola de listos.

  No bloqueante → No detiene la ejecución. En lugar de ello, la llamada vuelve rápidamente, con un valor de retorno que indica cuantos bytes se han trasferidos.

- *Asincrónicas / Sincrónicas*
  Asincrónico → La llamada vuelve rápidamente, sin ningún valor de retorno. Cuando la E/S termina avisa (consume una cantidad impredecible de tiempo).

  La diferencia con una no bloqueante, es que una operación read () no bloqueante vuelve inmediatamente con los datos que haya disponibles, mientras que una llamada read () asincrónica solicita una transferencia que se realizara de modo completo pero que completara en algún instante futuro.

  Sincrónico → Como voy a seguir con el proceso depende del resultado de E/S (consume una cantidad predecible de tiempo)

# Capitulo: Interfaz del Sistemas de Archivos (Parte A)

## Concepto de Archivo

Un archivo es una colección de información relacionada, con un nombre, que se graba en almacenamiento relacionado. Es la unidad lógica más pequeña de almacenamiento secundario (no pueden escribirse datos en el almacenamiento secundario a menos que estos se encuentren dentro de un archivo).

Los atributos de un archivo generalmente son:

- Nombre → identifica el archivo para el usuario.
- Identificador → identifica el archivo dentro del sistema de archivos
- Tipo
- Ubicación
- Tamaño
- Protección → Información de control de acceso al archivo (quien puede leer, escribir, etc.)
- Fecha, hora e identificación del usuario

La información de los archivos se almacena en *la estructura de directorios*, que también reside en almacenamiento secundario.

## Sistema de gestión de archivos

Un sistema de gestión de archivos es aquel conjunto de software del sistema que proporciona servicios a los usuarios y aplicaciones para el uso de archivos. Los objetivos de un sistema de gestión de archivos son:

- Almacenar datos y operar con ellos
- Protección de los datos
- Optimizar la performance del sistema
- Minimizar la posibilidad de pérdida de datos.
- Brindar soporte para distintos tipos de almacenamiento
- Interface personalizada para aplicaciones de usuarios
- Garantizar que sea posible la coherencia de los datos

**Arquitectura de los sistemas de archivos**

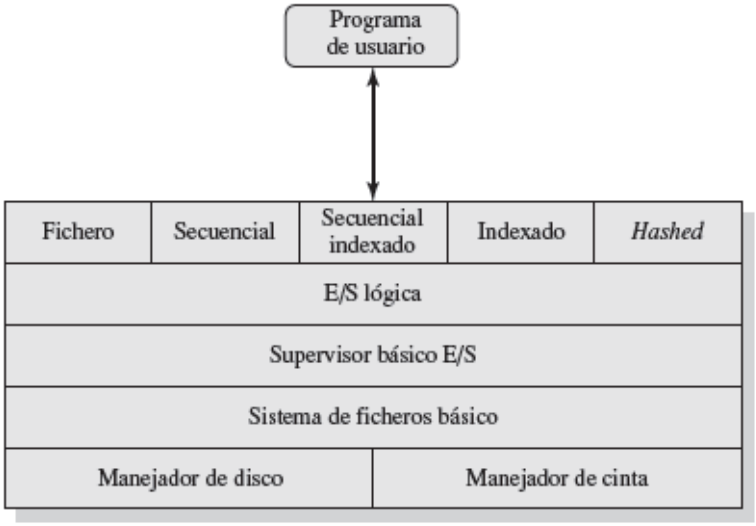



**Figura 12.1.** Arquitectura software de un sistema de ficheros.

La figura muestra una representación de una organización típica del software (puede variar dependiendo del sistema). Por lo tanto, el sistema de archivos está compuesto, generalmente, de muchos niveles. Cada nivel del diseño utiliza funciones de los niveles inferiores. Desde el nivel más bajo al más alto:

- *Controladores de dispositivo* → Un controlador de dispositivo es el responsable de iniciar las operaciones de E/S de un dispositivo y procesar la finalización de una petición de E/S. Los controladores de dispositivos se consideran normalmente parte del sistema operativo.

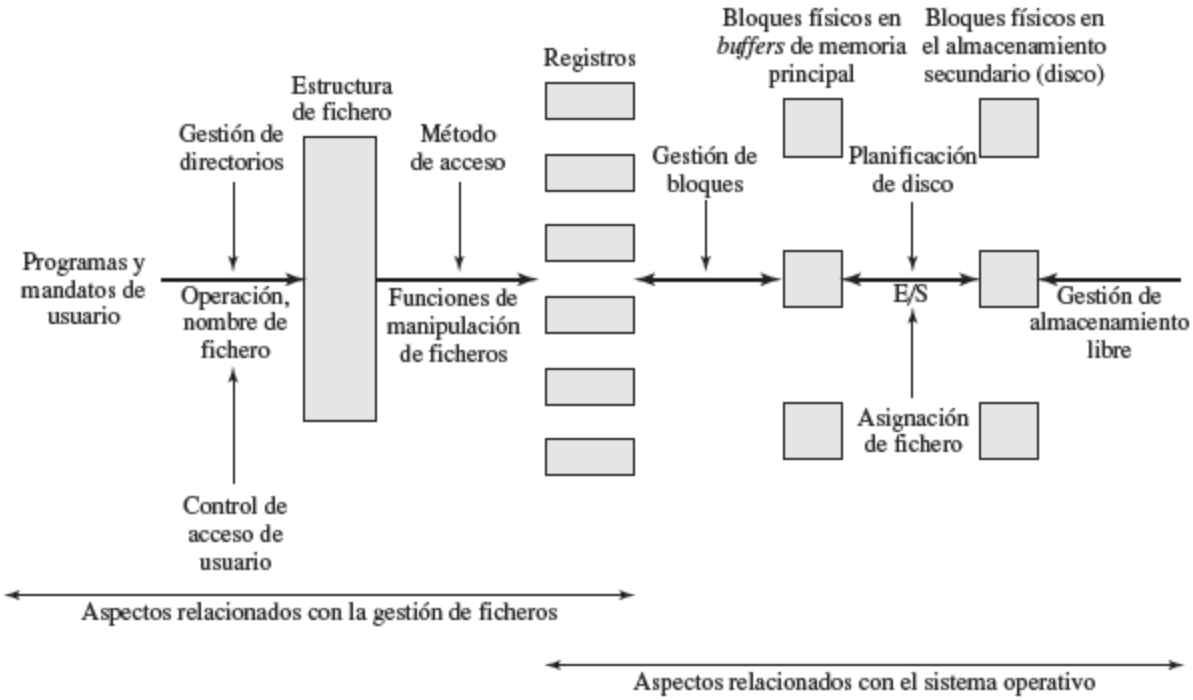
- *Sistema de archivos básico* → envía comandos genéricos al controlador de dispositivo apropiado, con el fin de leer y escribir bloques físicos en el disco. El sistema de archivos básico se considera normalmente parte del sistema operativo.

- *Supervisor básico de E/S* → El supervisor de E/S básico selecciona el dispositivo en el cual se van a llevar a cabo la operación de E/S. También se encarga de la planificación de disco y cinta para optimizar el rendimiento. A este nivel, se asignan los buffers de E/S y se reserva la memoria secundaria. El supervisor de E/S básico es parte del sistema operativo.

- *E/S Lógica* → Permite a los usuarios y a las aplicaciones acceder a los registros.

- *Método de acceso* → Diferentes métodos de acceso reflejan diferentes estructuras de archivos y diferentes formas de acceder y procesar los datos.

Otra forma de ver las funciones de un sistema de archivos se muestra en la siguiente figura:




Sigamos este diagrama de izquierda a derecha:

- Los usuarios y programas de aplicaciones interaccionan con el sistema de archivos mediante órdenes/operaciones sobre archivos.

- El sistema de archivos debe identificar y localizar el archivo seleccionado. Esto requiere el uso de algún tipo de *directorio* que se utilice para describir la ubicación de todos los archivos. Adicionalmente se aplica un control de acceso (sólo se permite determinado acceso particular a los usuarios autorizados.)

- Las operaciones básicas que puede realizar un usuario o aplicación se realizan a nivel de registro. El usuario o aplicación ve el archivo como una estructura que organiza los registros. Por tanto, para traducir los órdenes de usuario en órdenes de manipulación de archivos, debe emplearse el *método de acceso* apropiado para esta estructura de archivos.

- La E/S se realiza a nivel de bloque. Por tanto, los registros de un archivo se deben traducirse en bloques para la salida y los bloques traducirse a registros después de la entrada.

- Se debe gestionar el almacenamiento secundario Asignación de archivos a bloques libres de almacenamiento secundario, gestión del espacio libre, planificación de las peticiones de E/S.

En sí, la figura sugiere una división entre las funciones del sistema de gestión de archivos y las funciones del

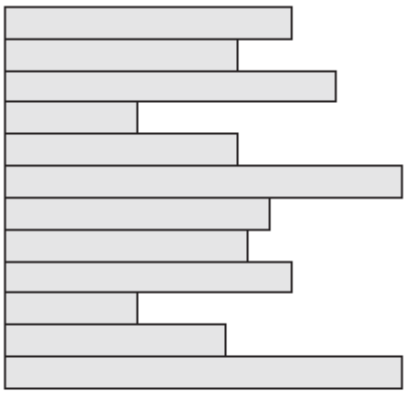
sistema operativo, siendo el punto de intersección el procesamiento de registros.

**Organización y acceso a archivos**

Las organizaciones son: *archivo de pila, archivo secuencial, archivo secuencial indexado, archivo indexado, archivos hash*.

- *Pilas*

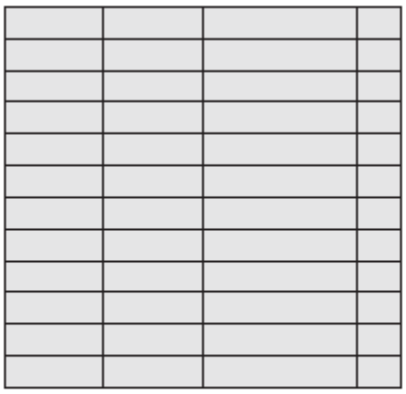
  Forma más simple de organización de archivos. Los datos se recolectan en el orden en que llegan. Cada registro consiste en una ráfaga de datos. El propósito de la pila es simplemente acumular la masa de datos y guardarlo. El acceso a los datos se hace por *búsqueda exhaustiva* por no tener una estructura. Fáciles de actualizar. Aprovecha bien el espacio cuando los datos almacenados varían en tamaño y estructura.



Registros de longitud variable
Conjunto variable de campos
Orden cronológico

- *Archivos secuenciales*

  Se emplea un formato fijo para los registros. Todos los registros son de igual tamaño y están compuestos por el mismo número de campos de longitud fija en un orden específico. Las búsquedas se realizan en forma secuencial.



Registros de longitud fija
Conjunto fijo de campos en orden fijo
Orden secuencial basado en el campo clave

- *Archivos secuenciales indexados*
  Solventa las desventajas de los archivos secuenciales. Mantienen las características básicas de los archivos secuenciales. Los registros se organizan en una secuencia basada en un campo clave, pero se añaden 2 características nuevas: un índice del archivo para soportar los *accesos aleatorios* y un archivo de desbordamiento (overflow). Reducen enormemente el tiempo necesario para acceder a un solo registro sin sacrificar la naturaleza secuencial del archivo.

- *Archivos indexados*
  Cuando es necesario buscar por algún otro atributo que no sea el campo clave, ambas formas de archivos secuenciales son inadecuadas. En algunas aplicaciones, esta flexibilidad es deseable.
  Para lograr esta flexibilidad, se necesita una estructura que emplea *múltiples índices*, uno por cada tipo de campo que puede estar sujeto a una búsqueda (índice exhaustivo y parcial)

- *Archivos directos o hash*
  Los archivos hasheados o directos, hacen uso de la capacidad del disco para acceder directamente a un bloque. Se requiere una clave para cada registro. Sin embargo, en este tipo de archivos no existe el concepto de ordenación secuencial.

  El archivo directo hace uso de una función hash sobre un valor clave. Los archivos directos se utilizan frecuentemente cuando se requiere un acceso muy rápido (ejemplo: tabla de precios).

**Organización de directorios**

El directorio de archivos contiene información sobre los archivos, incluyendo atributos, ubicación y propietario. Gran parte de esta información la gestiona el SO. El directorio es propiamente un archivo, poseído por el sistema operativo.

La información que almacena difiere mucho entre los distintos sistemas, pero algunos elementos claves siempre deben permanecer en el directorio, como el nombre, dirección, tamaño y organización.

Los tipos de operaciones que pueden realizarse con un directorio:

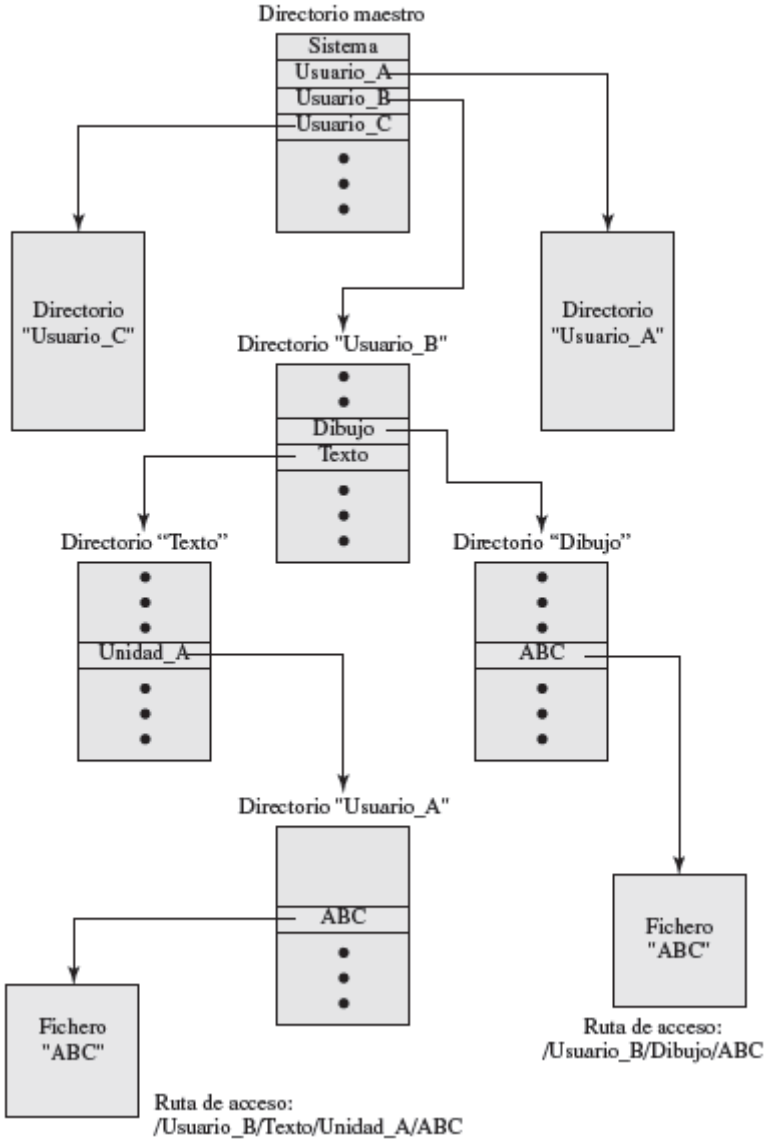
- Buscar Archivo → cuando un usuario hace referencia a un archivo debe buscarse en el directorio la entrada correspondiente.
- Crear Archivo → se añade una entra al directorio
- Borrar Archivo → se elimina una entrada del directorio
- Enumerar directorio → una lista de todos los archivos poseídos por dicho usuario.
- Actualizar directorio → cambio en la entrada del directorio correspondiente por un cambio de algún atributo del archivo

Una única lista no es adecuada como estructura del directorio para soportar estas operaciones por cuestiones de organización (por ejemplo tener los archivos separados por tipo, por usuario, etc.).
Una técnica más potente y flexible, que es casi universalmente adoptada, es utilizar una estructura jerárquica en forma de árbol.

Como en la técnica anterior, hay un directorio maestro, que tiene bajo dicho directorio varios directorios de usuario. Cada uno de estos directorios de usuario, a su vez, podría tener subdirectorios y archivos como entradas. Esto se cumple para todos los niveles. Es decir, en cada nivel, un directorio podría estar formado por subdirectorios y/o archivos.

Cada directorio se almacena como un archivo secuencial. Cuando los directorios tengan un número muy grande de entradas es preferible una estructura hash.

El conjunto de nombres de directorios, finalizando en el nombre del archivo, constituye un **nombre de camino**



para el archivo.

**Agrupación de registros**

Como se vio los registros so la unidad lógica de acceso a los archivos, mientras que los bloques son la unidad de E/S para almacenamiento secundario. Para que la E/S se pueda realizar, los registros se deben organizar como bloques.

- Los bloques en la mayoría de los sistemas son *de longitud fija.* Esto simplifica la E/S.
- Cuanto *mayor* sea el bloque, más registros se transferirán en una operación de E/S. La preocupación viene por el hecho de que bloques más grandes requieren buffers de E/S mayores, haciendo la gestión de buffers más difícil.
- Los bloques pueden ser de:
    - *Tamaño fijo*: Se utilizan registros de longitud fija, guardándose en cada bloque un número entero de registros. Podría haber espacio no utilizado al final de cada bloque. Esto se denomina *fragmentación interna*.

    - *Bloques de longitud variable con tramos:* Se utilizan registros de longitud variable y se agrupan en bloques sin dejar espacio sin usar. Por tanto, algunos registros deben abarcar dos bloques, con su continuación indicada por un puntero al bloque sucesor.

    - *Bloques de longitud variable sin tramos*. Se utilizan registros de longitud variable, pero no se dividen en tramos. En la mayoría de los bloques habrá espacio desperdiciado si el siguiente registro es mayor que el espacio sin usar.

    *Utilizar bloques fijos es el modo común para archivos secuenciales con registros de longitud fija.*

    Los *Bloques de longitud variable con tramos* son difíciles de implementar ya que los registros que ocupan 2 bloques requieren dos operaciones de E/S y los archivos son difíciles de actualizar.

    Los *Bloques de longitud variable sin tramos* implican espacio malgastado y limitan el tamaño del registro al tamaño de un bloque.

## Capitulo: Implementación de sistemas de archivos (Parte B)

En esta segunda parte, se verá los temas relativos al almacenamiento de archivos y al acceso a archivos en el medio más común de almacenamiento secundario, que es el disco.

Las transferencias de E/S entre la memoria y el disco se realizan en unidades de *bloques* para mejorar la eficiencia de E/S. *Cada bloque tiene uno o más sectores*. Los sectores varían entre 32 bytes y 4096 bytes; generalmente su tamaño de 512 bytes.

*Bloque de control de archivo (FCB)* → Estructura que contiene información acerca de un archivo especifico, incluyendo su propietario, los permisos y la ubicación del contenido del archivo. Hay un FCB por archivo.

*Nota:* Cuando se crea un archivo se asigna un nuevo FCB.

| Permisos del archivo |
|---|
| Fechas del archivo (creación, accesos, escritura) |
| Propietario del archivo |
| Tamaño del archivo |
| Punteros a los bloques de datos del archivo |

*Partición* → división lógica de un dispositivo físico.
*Volumen* → Partición formateada con un sistema de archivos

*Tabla de asignación de archivos (FAT)* → Estructura de datos que se usa para guardar constancia de las secciones asignadas a un archivo.

### Asignación del espacio de memoria secundaria a los archivos

El principal problema de un sistema de archivos es como asignar el espacio a los archivos de modo que el espacio de disco se utilice de forma eficaz y que se pueda acceder a los archivos de forma rápida. Hay 3 métodos: *asignación contigua, asignación enlazada y asignación indexada*.

*Tipos de asignación*

- *Asignación previa* → Requiere que el tamaño máximo del archivo sea declarado en el momento crearlo. Es difícil, a veces imposible, estimar el tamaño máximo del archivo. Por lo que habría que sobrestimar el tamaño de lo que supone un desperdicio de la asignación de espacio de almacenamiento.

- *Asignación dinámica* → Asigna espacio a un archivo en secciones a medida que se necesitan.

*Tamaño de la sección*

- *Secciones contiguas variables y grandes* → El mejor rendimiento. Evita malgastar espacio, y las tablas de asignación de archivos serán pequeñas. Espacio difícil de reutilizar.

- Bloques → Secciones fijas y pequeñas ofrecen mayor flexibilidad. Tablas de asignación serán grandes y/o complejas. Los bloques se asignan a medida que se necesiten (pueden no ser contiguos).

En el caso de secciones contiguas variables y grandes se asigna previamente un grupo de bloques contiguos. Esto elimina la necesidad de una tabla de asignación de archivos; todo lo que se requiere es un puntero al primer bloque y el número de bloques asignados.

En el caso de los bloques, todas las secciones requeridas se asignan a la vez. Esto significa que la tabla de asignación de archivos para el archivo es de tamaño fijo.

Con secciones de tamaño variable, es necesario preocuparse de la fragmentación del espacio libre. Las siguientes estrategias son posibles:

• *Primer ajuste*: Elegir el primer grupo de bloques contiguo sin usar de tamaño suficiente.

• *Siguiente ajuste*: Elegir el grupo más pequeño sin que sea de tamaño suficiente.

• *Ajuste más cercano*: Elegir el grupo sin usar de tamaño suficiente que está más cercano al asignado previamente al archivo para aumentar la cercanía.

*Métodos de asignación de archivos*

| | Contiguos | Encadenado | Indexado | |
|---|---|---|---|---|
| **¿Preasignación?** | Necesaria | Posible | Posible | |
| **¿Porciones de tamaño fijo o variable?** | Variable | Bloques fijos | Bloques fijos | Variable |
| **Tamaño de porción** | Grande | Pequeño | Pequeño | Medio |
| **Frecuencia de asignación** | Una vez | Pequeña a alta | Alta | Baja |
| **Tiempo a asignar** | Medio | Largo | Corto | Medio |
| **Tamaño de tabla de asignación de ficheros** | Una entrada | Una entrada | Grande | Medio |

*Asignación contigua*

Cuando se crea el archivo se asigna un único conjunto contiguo de bloques. Es una estrategia de *asignación previa*. La FAT contiene sólo una entrada por cada archivo y que muestra el bloque de comienzo y la cantidad de bloques que ocupa.

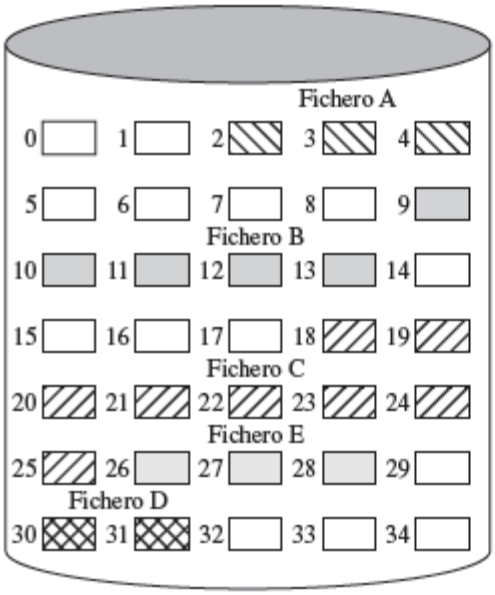



Tabla de asignación de ficheros

| Nombre de fichero | Bloque inicial | Longitud |
|---|---|---|
| Fichero A | 2 | 3 |
| Fichero B | 9 | 5 |
| Fichero C | 18 | 8 |
| Fichero D | 30 | 2 |
| Fichero E | 26 | 3 |

Ventajas:

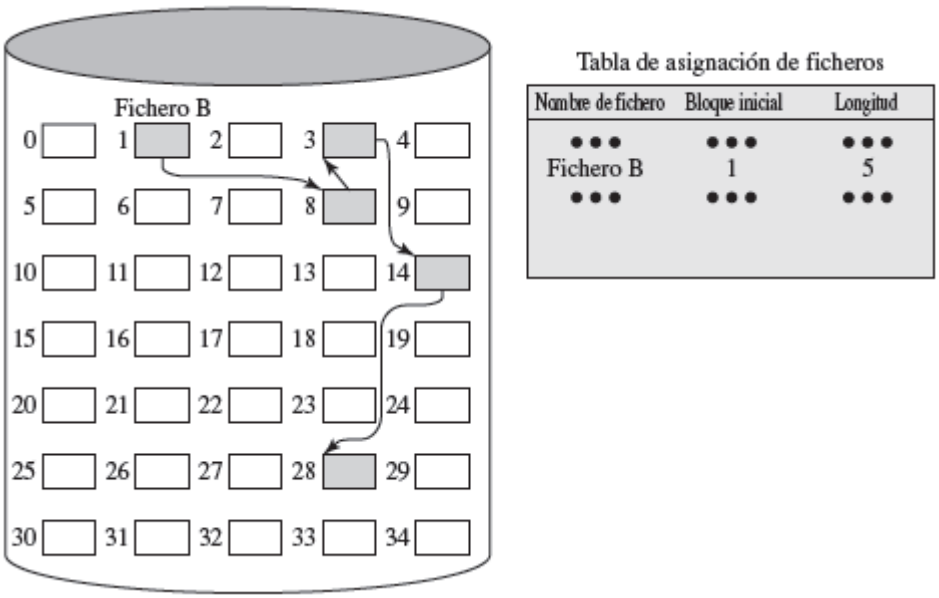
- Requiere menos reposicionamientos del cabezal para escritura y lectura al ser posiciones contiguas
- Eficiente para acceso directo y secuencial
- Si se daña un bloque NO pierdo al resto.

Desventajas:

- Fragmentación externa a causa de la asignación previa. A medida que se asignan y borran archivos, el espacio libre del disco se descompone en pequeños fragmentos.
- Sufre demasiada *fragmentación interna* a causa de la asignación previa.
- Poco óptimo para expandir archivos a causa de la asignación previa.

*Asignación enlazada*

Asigna bloques dinámicamente a medida que se necesiten (aunque es posible la asignación previa). Para poder seguir la cadena de bloques de un archivo, cada bloque contendrá un puntero al siguiente bloque de la cadena. La tabla de asignación de archivos necesita una sola entrada por cada archivo que muestre el bloque de comienzo y la longitud del archivo.




Ventajas:

- *No tiene fragmentación externa* al asignar un bloque cada vez y pudiendo ser de forma no continua (asignación dinámica).

- Sufre en menor medida la fragmentación internada en comparación a la asignación contigua

- Óptimo para expandir el archivo

Desventajas:

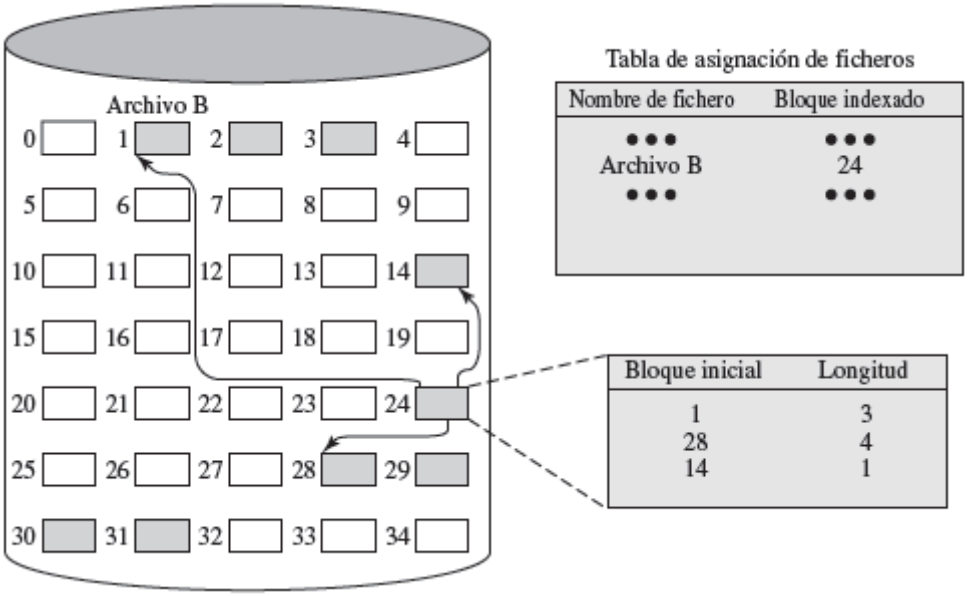
- *Requiere muchos reposicionamientos del cabezal* para escritura y lectura al ser posiciones no contiguas (muy dispersas en el disco).

- *No es eficiente para acceso directo*. Se debe recorrer la cadena hasta el bloque deseado.

- Si se daña un bloque pierdo al resto.

*Asignación indexada*

La tabla de asignación de archivos contiene un índice separado de un nivel para cada archivo. El índice posee una entrada para cada sección asignada al archivo. El índice del archivo se guardará en un bloque aparte y la entrada del archivo en la tabla de asignación apuntará a dicho bloque.




Ventajas:

- *No tiene fragmentación externa* al asignar un bloque cada vez y pudiendo ser de forma no continua (asignación dinámica).
- Sufre en menor medida la fragmentación internada en comparación a la asignación contigua

- Eficiente para acceso directo y secuencial

- Si se daña un bloque NO pierdo al resto.

Desventajas:

- *Requiere muchos reposicionamientos del cabezal* para escritura y lectura al ser posiciones no contiguas (muy dispersas en el disco).

- Requiere de espacio para almacenar el índice

- Requiere mantener actualizado el índice.

**Gestión del espacio libre**

Para llevar a cabo cualquiera de las técnicas de asignación de archivos descritas previamente, es necesario saber qué bloques del disco están disponibles. Por tanto se necesita una **tabla de asignación de disco**. Tres técnicas son de uso común: las *tablas de bits, las secciones libres encadenadas y la indexación*.

*Tablas de bits*

Utiliza un vector que está formado por un bit por cada bloque en el disco. Cada entrada 0 corresponde a un bloque libre y cada 1 corresponde a un bloque en uso.
La cantidad de memoria (en bytes) requerida para un mapa de bits en bloques se puede calcular de la siguiente manera: *Tamaño del disco en bytes / (8 x tamaño del bloque en el sistema de archivos)*

*Ventajas*:

- Fácil de encontrar un bloque libre o un grupo contiguo de bloques libres.
- Trabaja bien con cualquier método de asignación de archivos

*Desventajas*:

- Rendimiento inaceptable cuando la tabla de bits es grande. La tabla de bits puede ser grande como para almacenarla en memoria principal y una tabla en disco no es eficiente porque requiere E/S cada vez que se necesite un bloque (parte de la ineficiencia se puede reducir manteniendo estructuras auxiliares que resumen el contenido de subrangos de la tabla de bits)

*Secciones libres encadenadas*

Las secciones libres pueden encadenarse mediante un puntero y un valor de longitud en cada sección libre.

*Ventajas*:

- Sobrecarga de espacio insignificante. No se necesita una tabla de asignación de disco, sino simplemente un puntero al comienzo de la cadena y la longitud de la primera sección.

- Trabaja bien con cualquier método de asignación de archivos

*Desventajas*:

- Disco fragmentado después de un tiempo de uso
- Se ralentiza la creación del archivo al tener que actualizar los punteros del bloque.

*Indexación*

La técnica de indexación trata el espacio libre como un archivo y utiliza una tabla de índices tal y como se describió en la asignación de archivos. Por motivos de eficiencia, el índice se debería utilizar en base a *secciones de tamaño variable* en lugar de bloques. Por tanto, hay una entrada en la tabla por cada sección libre en el disco. Esta técnica proporciona soporte eficiente a todos los métodos de asignación de archivos.