# wResumen 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 4 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).
* Pila

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

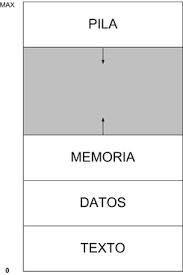
* Proceso es una entidad activa y dinámica (asignación dinámica de memoria: Es decir que puede solicitar y liberar memoria)
* Programa entidad pasiva y estática.

Un programa es el código ejecutable y sus datos asociados, mientras que un proceso es una instancia en ejecución de ese programa.

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 porque un programa en ejecución puede tener varios procesos. Dentro de un proceso pueden ejecutarse tanto un programa de usuario como uno del sistema operativo

# Proceso en Memoria



**Estas secciones trabajan juntas para permitir que un programa se ejecute correctamente**

* Pila del proceso: Datos temporales (Aquí se carga la dirección en memoria)
* Memoria (HEAP): Es la memoria que se le asigna dinámicamente al proceso en tiempo de ejecución.
* Sección de datos: Almacena Variables Globales.
* Sección de Texto: Almacena el 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:

# Identificación del proceso

Identificador del proceso, del proceso padre y del usuario.

# Información de estado del procesador (Contexto del programa)

**Registros de CPU:** **contador del programa**, **códigos de condición** (acarreo, signo, cero, desbordamiento, igualdad) **e Información de estado** (PSW). 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). Almacena información sobre el estado actual de un proceso.

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

# 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, hay un PCB por cada proceso**. El conjunto de PCBs define el estado del SO.

# 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.) Es tener los permisos de administrador prácticamente

La razón por la que se usan estos 2 modos, es para 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 a 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.

Debido a que los procesos están en memoria, un cambio de proceso implica inevitablemente un cambio de contexto ya que pasa de memoria a CPU. 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. Es decir que se puede realizar un proceso en modo usuario y si es necesario tener privilegios para algo no se cambia de proceso, solo cambia de modo pero en el mismo proceso. Cuando se produce una interrupción se salva el contexto del proceso y se puede que cambiar de modo usuario a modo kernel.

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:**

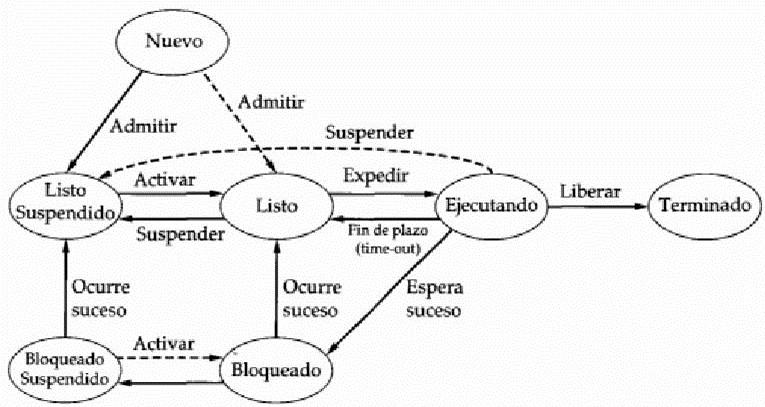
* **Interrupción Hardware externa:** originada por algún tipo de suceso que es externo e independiente del proceso que está ejecutándose. Interrupción de E/S, Interrupción de reloj (cada cierto tiempo, envía una señal al sistema operativo para que este pueda ocuparse de tareas importantes, como mantener la hora actualizada o gestionar los programas que se están ejecutando. Es una especie de "pausa programada" que mantiene todo en orden.).
* **Interrupción Hardware interna** (Excepciones): causada por una condición de error generada dentro del proceso que está ejecutándose (división por 0 (Evita que se produzcan resultados indefinidos o errores graves por operaciones matemáticas no permitidas), el desbordamiento (ocurre cuando estás tratando de almacenar un número más grande de lo que el sistema puede manejar, evita errores que no pueda manejar el sistema), 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: **Cada vez que se produce una interrupción: Se salva el 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. Hay que tener en cuenta que debido a una interrupción no siempre es necesario un cambio de proceso

Nota: **¿Qué 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 sacó 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 llamada al sistema (pasa a bloqueados), por lo que NO es posible que un proceso pase del estado “en ejecución” a listo.

Notas:

\*\* Fin de Quantum o fin de plazo: Se refiere al momento en que un proceso ha utilizado su tiempo asignado de CPU. El sistema operativo puede interrumpir ese proceso y dar tiempo de CPU a otro proceso.

Yield: es una instrucción o llamada que un proceso puede hacer al sistema operativo para indicar que está dispuesto a ceder la CPU antes de que su quantum termine. Al hacer un "yield", el proceso indica que no necesita utilizar todo su tiempo asignado y permite que otros procesos tengan la oportunidad de ejecutarse. \*\*

* + 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 - SWAP)

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 generar una situación en que todos los procesos estén en estado bloqueado y el procesador no tenga proceso para ejecutar. (Esto se llama “bloqueo por E/S", donde los procesos están temporalmente inactivos mientras esperan que se completen las operaciones de E/S como escritura o lectura hacia o desde un disco, y durante ese tiempo, el procesador no tiene nada más que hacer.)

Nota: \*La multiprogramación es una técnica en la que se cargan varios programas en la memoria principal simultáneamente. Pero solo un programa se ejecuta en la CPU en un momento dado. La multiprogramación permite que, cuando un proceso se bloquea por alguna razón (por ejemplo, espera una operación de entrada/salida), el sistema operativo cambie a otro proceso listo para ejecutarse. Esto optimiza el uso del tiempo de la CPU de modo que la CPU siempre tiene algo para procesar\*

Una solución a este problema, es el intercambio (swap), ayuda a evitar que el procesador quede inactivo al mover una parte del proceso o todo el proceso de la MP a M. De esta manera se baja el grado de multiprogramación, permitiendo el ingreso de un nuevo proceso listo para ejecutarse en memoria principal RAM (Se puede traer un nuevo proceso o uno listo suspendido). Por lo general se suspende aquel proceso bloqueado que hace tiempo está esperando un suceso.

Esto mejora la capacidad del sistema para manejar múltiples tareas simultáneamente.

1. Otros motivos de suspensión: cambios en los requisitos de memoria, espera de recursos compartidos, espera de datos, espera de sincronización, 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 en vez de mejorar (aunque la E/S de disco es la más rápida del sistema y por lo general suele mejorar el rendimiento).

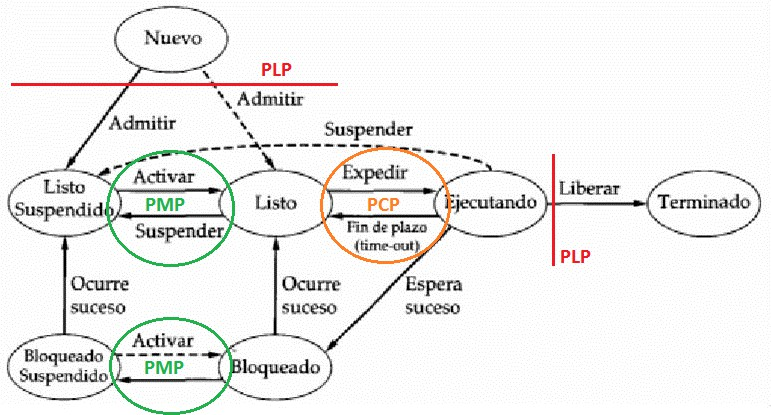
# Capítulo: Planificación de la CPU

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” con el objetivo que se minimice el tiempo de espera**, de respuesta y de ejecución y maximice el rendimiento del entorno (uso de CPU y tasa de procesamiento).

# Nivel de planificación

* Planificación de **largo plazo** (Planificación de **Dispositivos y Recursos**): Controla el grado de multiprogramación (cuantos procesos pueden estar en MP) y asigna recursos de manera eficiente. A más procesos, menor cantidad de tiempo para ejecutar cada proceso. Se realiza al tomar la decisión de crear o terminar un proceso nuevo. Cuando termina, libera los recursos
* Planificación de **Mediano plazo** (Planificación de **Memoria**): Encargado de realizar el intercambio (swap) para bajar el grado de multiprogramación. Decide qué procesos deben estar en la memoria principal y cuáles pueden moverse a la memoria secundaria.
* Planificación de **corto plazo** o **Despachador** (Planificación de **CPU**): Un programa, parte del SO, que se encarga de elegir y cambiar el proceso que se está ejecutando en el procesador (ocupa procesador cada vez que se cambia de un proceso a otro). Determina qué proceso pasará del estado listo al estado en ejecución. Se ejecuta cuando se produce una llamada al sistema, una interrupción de reloj, una interrupción de E/S y terminación del proceso.



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

Nota:

Cuando el proceso se ejecuta **hasta que termina o hasta que tenga que hacer una llamada al sistema**, 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.

Cuando el proceso se ejecuta **hasta una interrupción de reloj o una interrupción de E/S**, 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 pasó 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** (↑): Maximizar la 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.
  + **Tiempo de Retorno** (turnaround time) (**↓**): Tiempo que tarda un proceso de pasar de listo a terminado (es el tiempo total que está en el sistema).

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

Los algoritmos de planificación a corto plazo son:

1. **FCFS (Sin desalojo o no Apropiativo)**
2. **RR (Turno Rotatorio) (Con desalojo o Apropiativo)**
3. **SJF (Sin desalojo es SPN o Con Desalojo es SRT)**
4. **Por Prioridades (Sin desalojo o Con Desalojo)**
5. **HRRN (Sin desalojo)**
6. **Retroalimentación (Con desalojo)**
7. **Planificación FCFS**

Primero en llegar, primero en ser atendido. “First-Come, First-Served"

**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 ejecutan primero los I/O bound. (Es que un proceso largo puede bloquear otros procesos detrás de él) Bound significa Límite.

Nota: Un proceso I/O Bound es aquel cuya ejecución está limitada principalmente por operaciones de entrada/salida (E/S)

Un proceso CPU Bound es aquel cuya ejecución está limitada principalmente por la capacidad de procesamiento de la CPU

1. **Planificación RR (Round Robin)**

Igual que FCFS pero con desalojo por fin de quantum. Si Q (El tiempo asignado) es muy grande: Termina siendo igual que FCFS Si Q es muy chico: Produce mucho OVERHEAD. (Uso adicional de recursos)

Nota: El "overhead" se refiere al costo adicional asociado con el cambio frecuente entre tareas debido al desalojo por fin de quantum.Un "overhead" alto significa que estamos perdiendo eficiencia en la ejecución de las tareas.

Importante:

Ante simultaneidad de evento se atiende primero:

1. Excepción
2. Interrupción Q
3. Interrupción E/S
4. Llamada al sistema
5. **Planificación SJF (SPN es sin desalojo, si es con desalojo se le llama SRT)**

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:

* + El 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.α + Rn (1 – α) siendo 0 ≤ α ≤ 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: sin desalojo o no preemptiva, SRT ("Shortest Remaining Time" o Tiempo Restante más Corto).

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.

1. **Planificación por Prioridades (Sin desalojo o Con Desalojo)**

Primero se atiende el de mayor prioridad. Se asume que 0 es la prioridad más alta. Las prioridades pueden ser asignadas por el sistema operativo, el usuario o de alguna manera dinámica basada en el comportamiento del proceso. Si hay varios procesos con la misma prioridad, se puede utilizar otro criterio, como el orden de llegada a la cola de listos.

Ventajas:

* El 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.

1. **Planificación HRRN (Sin desalojo)**

Highest Response Ratio Next →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. Cuanta mayor tasa de respuesta, mayor prioridad

Tiene en cuenta la edad del proceso (por W, tiempo de espera). Por lo tanto elimina el problema de inanición.

1. **Retroalimentación (Con desalojo)**

Penaliza los trabajos que han estado ejecutándose por más tiempo

La planificación por retroalimentación organiza procesos en colas. Cuando comienza, los nuevos procesos entran en la cola de mayor prioridad. A medida que se ejecutan, si un proceso no termina en su cola asignada, se mueve a la siguiente cola de menor prioridad. Lo especial es que las colas se adaptan según el comportamiento de los procesos. Si un proceso es rápido, ascienden a las colas de mayor prioridad; si consume mucho CPU desciende a colas de menor prioridad. Esto ayuda a equilibrar cómo se comparten los recursos de manera justa y eficiente.

Como realmente no se conoce el tiempo que le falta a los procesos, contra más tiempo se estuvo ejecutando menos prioridad tendrá.

**Preguntas interesantes**

**¿Cómo sabe el SO cuánto tiempo demora cada proceso?**

La duración se puede llegar a estimar según muchos criterios, pero como no se pueden prever eventos inesperados o cambios en el comportamiento del proceso, no hay un tiempo real definido.

**Si en multiprogramación hay un proceso en memoria principal que se está ejecutando y se bloquea por alguna razón entonces comienza a ejecutarse otro proceso que ya está listo, pero antes de que el segundo proceso termine el primer proceso vuelve a estar listo para ejecutarse. ¿Cómo actúa el SO en este ejemplo?**

La acción dependerá de la planificación y de si es Sin Desalojo o con Desalojo:

**Con Desalojo/Apropiativo/ preemptivo:** Si el sistema operativo es preemptivo, puede interrumpir el segundo proceso en algún punto y cambiar de nuevo al primer proceso que ha vuelto a estar listo. Esto se hace para maximizar la utilización de la CPU y garantizar que los procesos listos se ejecuten tan pronto como sea posible.

**Sin Desalojo/No apropiativo/No preemptivo/Cooperativo:** En un sistema no preemptivo, el segundo proceso puede continuar su ejecución hasta que termine voluntariamente o se bloquee. Solo después de que haya terminado o se haya bloqueado, el sistema operativo podría cambiar al primer proceso.

La planificación y las prioridades pueden influir en estas decisiones. Si el primer proceso tiene una prioridad más alta, el sistema operativo podría darle preferencia y cambiar a él tan pronto como vuelva a estar listo.

**Capítulo: Hilos**

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

# Proceso Monohilo vs Multihilo



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 son a partir el rendimiento:

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).**

Los hilos de un mismo proceso pueden comunicarse entre sí, sin invocar al núcleo, **por lo tanto esta comunicación es más rápida y eficiente ya que no se comunica con el SO.** (Núcleo == kernel: El kernel es la parte central y esencial del sistema operativo)

# Estado de un hilo

Un hilo puede estar en ejecución, listo y bloqueado. No tiene sentido asociar estados de suspensión ya que pertenece al concepto de proceso. Si se expulsa un proceso, todos sus hilos se deben expulsar porque comparten el espacio de direcciones del proceso.

Hay 4 operaciones básicas de los hilos:

* **Creación**: Cuando se crea un nuevo proceso, también se crea un hilo de dicho proceso. Posteriormente,

el proceso o un hilo del proceso puede crear otro hilo dentro del mismo proceso. Cada hilo tiene su propio registro de contexto y espacio de pila

* **Bloqueo**: Ocurre cuando necesita esperar por algún evento, como una operación de E/S o una señal, y no

puede continuar su ejecución hasta que dicho evento se complete. Durante el bloqueo, el sistema

operativo debe guardar el contexto del hilo para poder cambiar a otro hilo en la CPU en estado Listo.

* **Desbloqueo**: Cuando sucede el evento por el que el hilo está bloqueado, el hilo se pasa a la cola de

Listos.

* **Finalización**: Cuando se completa un hilo, el sistema operativo debe liberar los recursos asociados a ese

hilo.

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

En una aplicación ULT (User Level Threads):

* + 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 y . 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 descritas, 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.

Nota: La "biblioteca de hilos" es un conjunto de herramientas que facilita la creación, coordinación y control de hilos en un programa. En lugar de lidiar directamente con el sistema operativo, la aplicación utiliza funciones específicas de esta biblioteca. Proporciona una interfaz más sencilla y portable para trabajar con hilos, permitiendo que la aplicación gestione la concurrencia de manera más eficiente. Ejemplos incluyen "pthread" en sistemas UNIX y Threading en Python. Básicamente es una librería que facilita el uso de hilos.

* + **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, continúa 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 específica : 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 lo 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 (Kernel Level Threads):

* + 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.

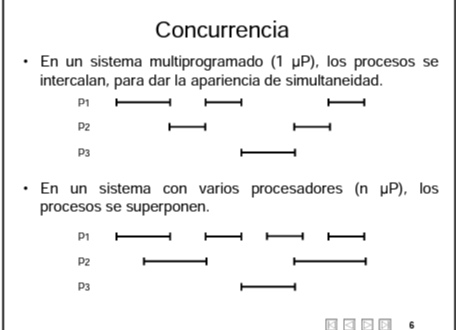
# Tipo de relación entre los ULT y KLT

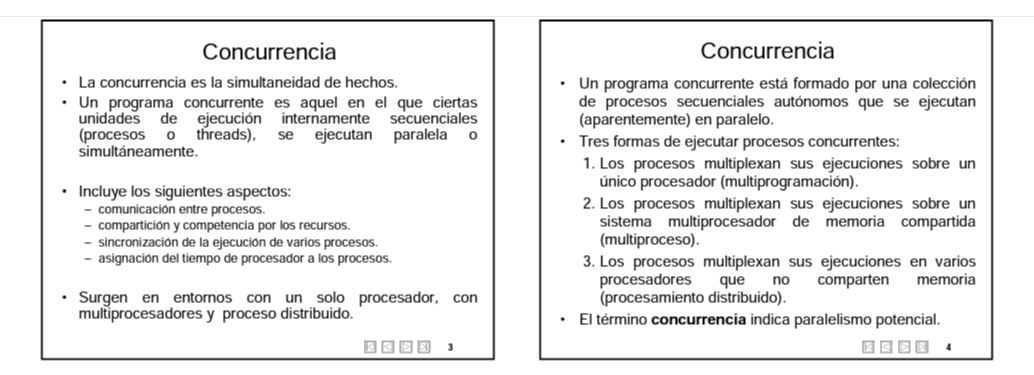
* + **Muchos a uno:** múltiples hilos de usuario a uno de Kernel. Desventaja que no hay concurrencia ya que 1 solo hilo ULT puede acceder al núcleo a la vez (no hay multiprocesamiento). Además una llamada bloqueante bloquea todo el proceso.
  + **Uno a Uno:** Soluciona los 2 problemas de Muchos a uno, 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.

# Capítulo: Concurrencia (exclusión Mutua y Sincronización)

**Concurrencia:** Ejecución simultánea de varias tareas o procesos. Comprende cuestiones del diseño del SO como compartición de recursos, interacción entre procesos, asignación de procesos a recursos, etc.

**Sección Crítica:** Es una **porción de un programa que manipula recursos compartidos**, como variables o datos, **de forma que si se ejecutan simultáneamente múltiples hilos o procesos podría generar resultados** **inesperados o incorrectos.**





**La concurrencia es fundamental en:**

* Multiprogramación (Varios procesos cargados en MP)
* Multiprocesamiento (El multiprocesamiento es una técnica en la que dos o más procesadores trabajan de manera conjunta en una única computadora o sistema)
* Procesamiento distribuido (Gestión de varios procesos ejecutándose en memorias de múltiples computadoras distribuidas).

**La concurrencia se presenta en 3 contextos:**

* Múltiples aplicaciones: Multiprogramación
* Aplicaciones estructuradas: Una aplicación puede ser un conjunto de procesos concurrentes
* Estructura del sistema operativo: Conjunto de procesos o hilos

**La concurrencia presenta dificultades como (3):**

* Compartir recursos globales (variables, disp. E/S).
* Para el sistema operativo es complicado gestionar la asignación de recursos de manera óptima.
* Dificultad de localizar errores de programación (Porque todo depende del orden en que se ejecuten los procesos).

**Exclusión mutua**

* **Sólo un proceso** de todos los demás, **puede estar en la sección crítica** de un mismo recurso compartido en un momento dado.
* Un proceso que se interrumpe en su sección NO crítica no debe interferir con otros procesos
* Un proceso no puede ser demorado indefinidamente (no puede haber interbloqueo e inanición).
* Si ningún proceso está en su sección crítica, cualquier proceso que solicite entrar debe poder sin demora.
* **No se deben hacer** **suposiciones** sobre las velocidades de los procesos ni sobre el número de procesadores.
* **Un proceso permanece** dentro de su sección crítica sólo **por un tiempo finito** (avisar que salió de la sección crítica).

Nota: \*\*Condición de carrera: Es un problema en la programación concurrente donde el resultado de un programa depende del orden en que se ejecutan las operaciones de hilos o procesos. Es como una "carrera" entre los hilos por acceder y modificar datos compartidos, y el resultado final puede variar según quién termine primero.\*\*

**Desventajas de Instrucciones de máquina especiales**

* Puede haber **inanición** ya que no hay una lista/fila/cola.
* Puede ocurrir **interbloqueo**.
* Emplea **espera activa,** es decir que el proceso que se queda esperando para acceder a una sección crítica consume procesador.

**Ventajas de Instrucciones de máquina especiales**

* Es aplicable a cualquier número de procesos sobre monoprocesador o multiprocesador.
* Es simple y, por tanto, fácil de verificar.

# **La inanición**, en términos técnicos, se refiere a una condición en la que un proceso o hilo queda bloqueado indefinidamente porque otros procesos o hilos siempre tienen prioridad sobre él. Es una situación en la que un proceso no puede avanzar, a pesar de estar listo para ejecutarse, debido a decisiones de planificación que favorecen constantemente a otros. La inanición puede afectar negativamente el rendimiento del sistema, ya que impide que ciertos procesos avancen y completen su trabajo.

**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.

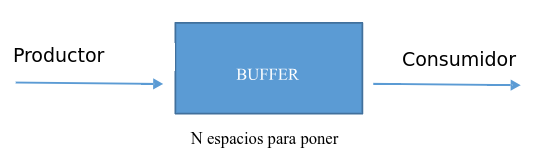
Nota:

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.

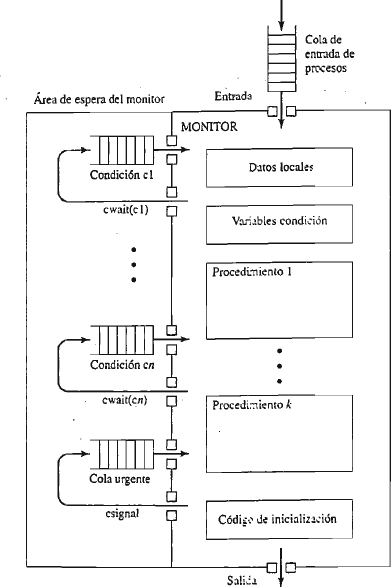
Los monitores ofrecen una funcionalidad equivalente a la de los semáforos y son más fáciles de controlar.

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).

# Capítulo: 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 en el que ambos procesos necesitan del otro:

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 ela 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 (Condiciones de coffman)

1. **Exclusión Mutua**: Sólo un proceso puede usar un recurso a la vez. (Puede generar interbloqueo si los procesos o hilos no liberan adecuadamente los recursos después de utilizarlos, por lo tanto se )
2. **Retención y espera**: un proceso puede retener unos recursos asignados mientras espera que se le asignen otros.
3. **No apropiación (No preemptivo- Sin desalojo)**: 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**:

1. **Espera Circular (Círculo 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).

**Estrategias para tratar el interbloqueo**

* Prevención
* Predicción
* Detección

# 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 círculo 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.

Ejemplo:

Sabiendo que un solo un proceso puede acceder a recursos a la vez, supongamos que tenemos 3 procesos en un sistema multiprogramado concurrente, cada uno de ellos tiene 10 segundos de procesamiento (Quantum). También suponemos que hay 4 recursos: A, B, C yD.

El proceso 1 utiliza el recurso A y B durante sus primeros 10 segundos, pero no termina con las tareas, por lo tanto no libera los recursos.

El proceso 2 utiliza el recurso C y D durante sus primeros 10 segundos, pero no termina con las tareas, por lo tanto no libera los recursos.

Cuando llegan los 10 segundos del proceso 3, no puede acceder a los recursos que necesita, (por ejemplo si necesitara del recurso A y C debe esperar a que se liberen los recursos), por lo tanto este se bloquea hasta que terminen los otros procesos y liberen los recursos que necesita. Esto es muy ineficiente por lo mencionado arriba.

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.

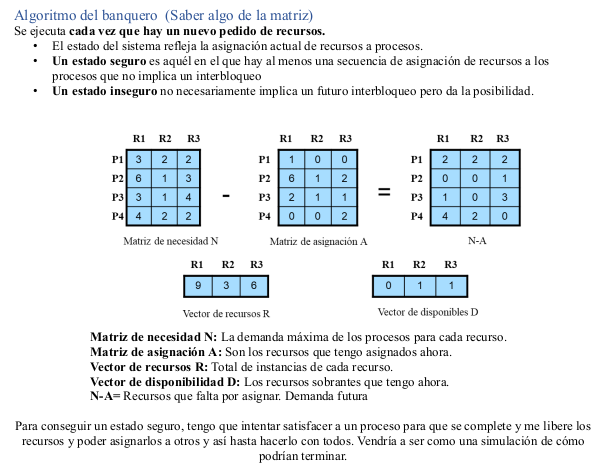
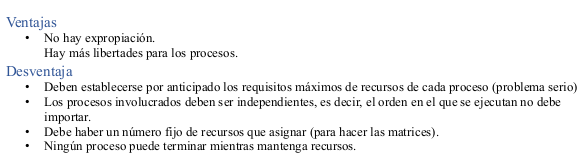
# Predicción del interbloqueo

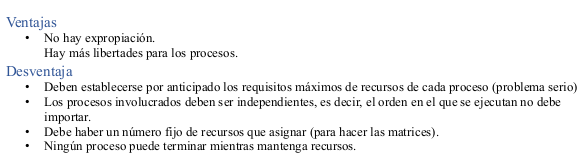
Con la predicción del interbloqueo, se puede alcanzar las 3 condiciones necesarias, pero se realizan elecciones acertadas para asegurar que nunca se llegue al puto de interbloqueo.

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 interbloqueo 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 procesos 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 (Qué estrategia usar en distintas situaciones)- Al truco con tuco

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

La memoria principal ofrece un acceso rápido con un coste relativamente alto. Además, la memoria principal

es volátil; esto es, no proporciona almacenamiento permanente. La memoria secundaria es más lenta y barata

que la memoria principal y, normalmente, no es volátil. De este modo, una memoria secundaria de gran

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

principal pequeña mantiene los programas y datos de uso actual. Leer esto y BORRARLO

En un **sistema monoprogramado, la memoria principal (RAM)** **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 esté 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

**Cuando un programa se carga en la memoria se determinan las direcciones, con el paso del tiempo pueden ir cambiando:**

**– Un proceso puede ocupar diferentes particiones durante la ejecución (por el swap, se puede**

**descargar a disco y volver a cargar en una dirección de memoria diferente).**

**– La compactación también causa que un programa tenga que ocupar diferentes particiones (se va**

**trasladando para no dejar huecos)**

**-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. (Ej: Si en un juego se guarda tu puntuación en la ubicación de memoria real 0x7F3A92B4, la dirección lógica podría ser 001, y hace referencia a la posición a la posición real, pero sin preocuparse por esta.)
    - **Dirección relativa**: posición **relativa a algún punto conocido, normalmente el principio del programa.**
    - **Dirección física**, es una posición real en la MP.

# Protección

* **El código de un proceso no puede hacer referencia a posiciones de memoria de otros procesos, sin permiso**.
* **Es imposible chequear direcciones dentro de los programas puesto que se pueden reubicar.**
* Por lo tanto, **todas las referencias a memoria generadas por un proceso deben comprobarse durante la ejecución**

# Compartimiento

Se debe **permitir a varios procesos acceder a la misma porción de memoria**. Los mecanismos para respaldar la reubicación forman parte básica de las capacidades de compartimiento.

Ejemplos:

• Acceder a la misma copia del programa en lugar de tener su propia copia.

• Acceder a una estructura de datos compartida (procesos del mismo programa o hilos).

# Organización lógica

Los programas se escriben en módulos. Existen diferentes grados de protección para los distintos módulos de un programa (solo lectura, solo ejecución). Compartición de módulos (Sería dividir un programa en partes más pequeñas y manejables llamadas módulos.)

# 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 operativo**.La responsabilidad de este flujo NO puede ser asignada al programador, por dos razones:

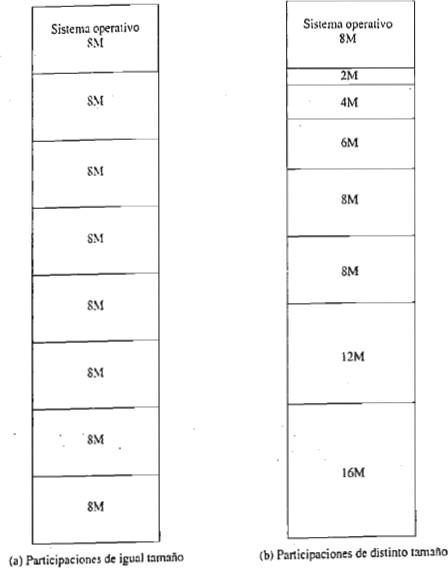
* + En un sistema multiprogramado, el programador no conoce durante la codificación cuanto espacio de memoria habrá disponible o donde estará este espacio.
  + La memoria principal para un programa puede ser insuficiente. En este caso el programador debe emplear una práctica conocida como superposición (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:

1. Emplear particiones estáticas de igual tamaño.
2. Emplear particiones estáticas 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, ocupará 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:

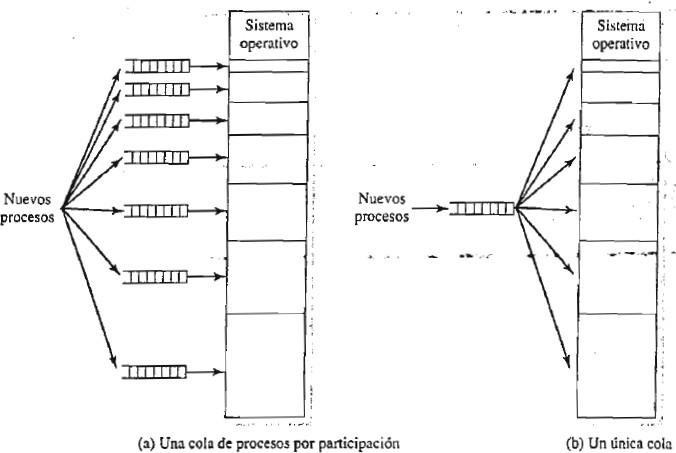
* **Fragmentación interna:** Es el **desperdicio de memoria** que se produce **cuando un programa ocupa una partición de memoria más grande de la que necesita**. Este espacio adicional no utilizado dentro de la partición asignada es la fragmentación interna. Por lo tanto, la memoria no utilizada "sobra" dentro de la partición. (**Cuando en una partición sobra memoria debido a que el programa ocupa menos memoria**)
* **Fragmentación externa:** Es el desperdicio de memoria que se produce cuando hay memoria disponible (sumando la memoria de varias particiones), pero esta no se puede usar, porque el programa no cabe dentro de UNA partición. Ej: Si quiero instalar windows que pesa 10 gb pero tengo 7 particiones de 6gb, el programa no cabe dentro de ninguna partición por más de que haya mucha memoria disponible.

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:

1. 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.
2. Asignar a cada proceso la partición más pequeña disponible que quepa.



Resumen:

| **Partición Estática**  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 | **Ventajas**  Sencillo de implementar. Poca sobrecarga del SO. | **Desventajas**  Fragmentación interna  El número de procesos activos es fijo. |
| --- | --- | --- |

# Particiones dinámicas

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 justa 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.

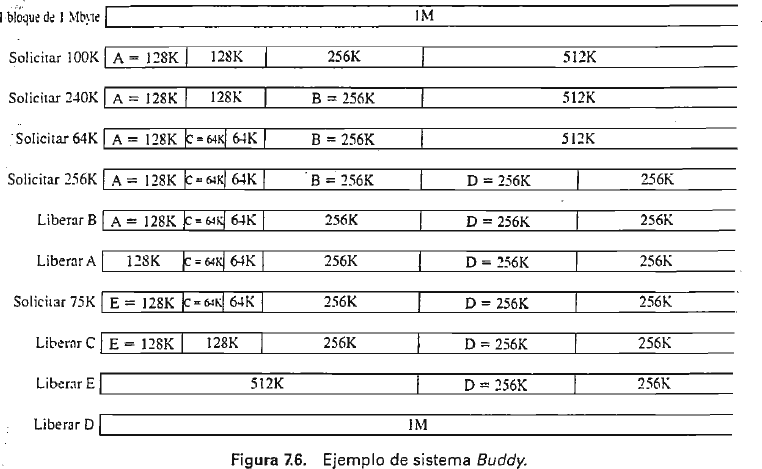
El mejor método aplicable dependerá de la secuencia exacta de intercambios de procesos que ocurran y del tamaño de estos procesos.

| **Primer ajuste** | **Siguiente Ajuste** | **Mejor ajuste** |
| --- | --- | --- |
| Sencillo.  El mejor y más rápido. | Genera peores resultados que el primer ajuste.  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  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 NO LO VI BIEN

Los bloques de memoria disponibles son de tamaño 2K, para valor de K tal que L ≤ K ≤ U y donde:

* 2L = tamaño de bloque más pequeño asignable.
* 2U = tamaño de bloque más grande asignable (generalmente es el tamaño de la memoria entera disponible para asignar).



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.

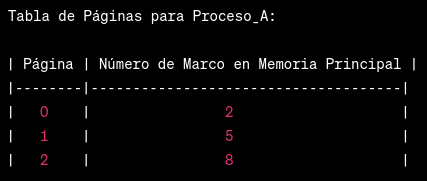
* **Dirección lógica**: dirección generada por la CPU que hace referencia a una posición de memoria independiente de la asignación actual de datos a la memoria. Es de 16 bits?
* **Dirección relativa**: posición relativa a algún punto conocido, normalmente el principio del programa.
* **Dirección física**, es una posición real en la MP.

# Paginación sencilla

Mediante la paginación simple, la memoria principal se encuentra dividida en **trozos iguales de tamaño fijo**, denominados **marcos**. A su vez, cada proceso está dividido también en trozos de tamaño fijo y del mismo tamaño que los de memoria, denominados **páginas**. El término **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. Es decir, un marco tiene el mismo tamaño que una página, el marco almacena la página, entonces cuando un programa se carga en la memoria, todas las páginas del programa se asignan a marcos libres. (Un proceso puede dividirse en varias páginas)

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 (que esté al lado o cerca) 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. Ejemplo: Un proceso A consta de 3 páginas.



(El número de marco sería por ejemplo en la página 0: A2)

Al igual que la partición estática, el procesador también realiza la traducción de direcciones lógicas a físicas. En el caso de la partición estática 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 l**a 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 no tienen por qué ser contiguas.

Ejemplo que proporcioné (dirección lógica: Página 2, Desplazamiento 1500), significa que el programa está accediendo a un dato o instrucción que se encuentra en la Página 2, y más precisamente, está situado 1500 bytes desde el inicio de esa página.

# Segmentación sencilla

Aquí la MP no se divide. El proceso es el que se divide. En este caso, el proceso 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. (En vez de dividirse en páginas se divide en segmentos)

Nota: \*\*Desplazamiento en Segmentación: El desplazamiento es similar al concepto en paginación. Indica cuántos bytes desde el inicio del segmento se encuentra la dirección lógica. Es decir, **se refiere a la ubicación específica del dato al que se quiere acceder dentro de un segment**o. Ej: El dato está a 1239 bytes desde el comienzo del segmento \*\*

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 porqué 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. Entonces, la "visibilidad" en la segmentación significa que la estructura de segmentación del programa es evidente para el programador, mientras que en la paginación esta estructura se mantiene oculta y gestionada por el sistema operativo. La visibilidad puede ofrecer más flexibilidad y control al programador, pero también implica una mayor complejidad en el manejo de las direcciones y segmentos.

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. La tabla de segmentos para cada proceso está formada por: número de segmento (índice), longitud

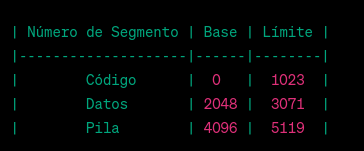
(tamaño del segmento) y la base (la dirección en donde empieza el segmento en la memoria física).

**Número de Segmento:** Identifica cada segmento del programa (por ej: Segmento1, Segmento2, Segmento3).

**Base:** Indica el inicio de cada segmento en la memoria física. Ej: Seg1 comienza en el byte 0, Seg2 comienza en el byte 2048, y Seg3 en el 4096.

**Límite:** Indica la longitud máxima del segmento.

Ejemplo de tabla de segmentos del **Segmento1:**



# Capítulo**: 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.

Cuando un nuevo proceso se carga en memoria, se suelen traer dos porciones inicialmente (La porción inicial del programa que es la parte inicial del código ejecutable y la porción inicial de datos que acceden las primeras instrucciones ), formando el **conjunto** **residente** del proceso **en la memoria principal**. Durante la ejecución, si una referencia de memoria está fuera de este conjunto, el procesador genera una interrupción por fallo de acceso.

El sistema operativo bloquea el proceso, realiza una solicitud de E/S para cargar la porción faltante desde el disco a la memoria y permite que otro proceso se ejecute durante la operación de E/S. Después de cargar la porción solicitada, se genera otra interrupción de E/S, devolviendo el control al sistema operativo, que coloca al proceso original en el estado Listo para reanudar su ejecución.

Nota: Conjunto residente se refiere a la parte de un proceso que se encuentra actualmente en la memoria principal. Es la porción del programa y datos accesibles en un momento dado durante la ejecución.

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

La memoria virtual funciona como una extensión de la memoria RAM. **Cuando un programa necesita más memoria de la que la RAM física puede proporcionar, el sistema operativo utiliza parte del disco duro como un espacio adicional**, conocido como archivo de intercambio. (Intercambio Swap)

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.

**Cuando el sistema operativo traiga una porción** (parte del proceso que estaba en disco) **a la memoria principal, debe expulsar otra**.

**Demasiados intercambios de fragmentos o porciones lleva a** una condición denominada **hiperpaginación** (**thrashing**): el procesador consume más tiempo intercambiando fragmentos que ejecutando instrucciones de usuario. (Es que el sistema operativo está constantemente moviendo datos entre la memoria principal (RAM) y el almacenamiento secundario (Disco)) 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 en un periodo de tiempo corto. (Cuando un programa accede a una dirección de memoria específica, es probable que acceda también a direcciones cercanas en un futuro cercano.)

Nota: **Fallo de página**: ocurre **cuando un programa** **necesita acceder a una dirección de memoria que no se encuentra en la memoria principal (RAM) actualmente.**

# Paginación en memoria virtual o memoria virtual paginada

Sólo algunas de las páginas de un proceso pueden estar en memoria principal

Cuando se está ejecutando un proceso en particular, la dirección de comienzo de la tabla de páginas para este proceso se mantiene en un registro. Se almacena en el BCP.

Las tablas, si son muy grandes, también se pueden almacenar una parte en memoria virtual.

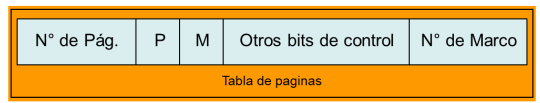
Cuando un proceso se ejecuta, parte de su tabla está en la memoria principal.

Cada proceso tiene su propia tabla de páginas:

* Incluye el número de marco para esa página.
* Numero de pagina
* Bit de Presencia (P): Se necesita un bit para indicar si la página está (1) o no (0) en memoria principal. Si es 0, provoca una interrupción de fallo de página.
* Bit de Modificación (M): Se necesita un bit para indicar si la página ha sido modificada desde la última vez que fue cargada en memoria. Si no ha habido cambio, la página no necesita volver a escribirse en el disco cuando se quita de memoria principal, se utiliza la que ya está escrita y evitó hacer una escritura en disco que es muchísimo más lenta.

•Puede haber también otros bits de control. Por ejemplo, si la protección o la compartición se gestiona a nivel de página, se necesitarán más bits con tal propósito.





Cada referencia a una dirección virtual puede causar dos accesos a memoria física.

• Uno para leer la tabla.

• Uno para leer el dato.

Esto doblaría el tiempo de acceso a memoria, entonces para mejorar esto se usa una caché especial para las

entradas de las tablas llamada TLB (Translation Lookaside Buffer).

Buffer de traducción adelantada (almacenamiento intermedio)

**Es una pequeña memoria** **más cercana al procesador.**

**Contiene** **las traducciones específicas de direcciones lógicas a direcciones físicas que han sido recientemente utilizadas¸ al momento de**

**reemplazar una referencia de pág., se elige la menos usada recientemente.**

**Dada una dirección virtual**, **el procesador examinará primero la TLB**:

1. **Si la traducción de la dirección virtual está presente (un acierto en la TLB)**:

**Se obtiene el número de marco y se forma la dirección real**.

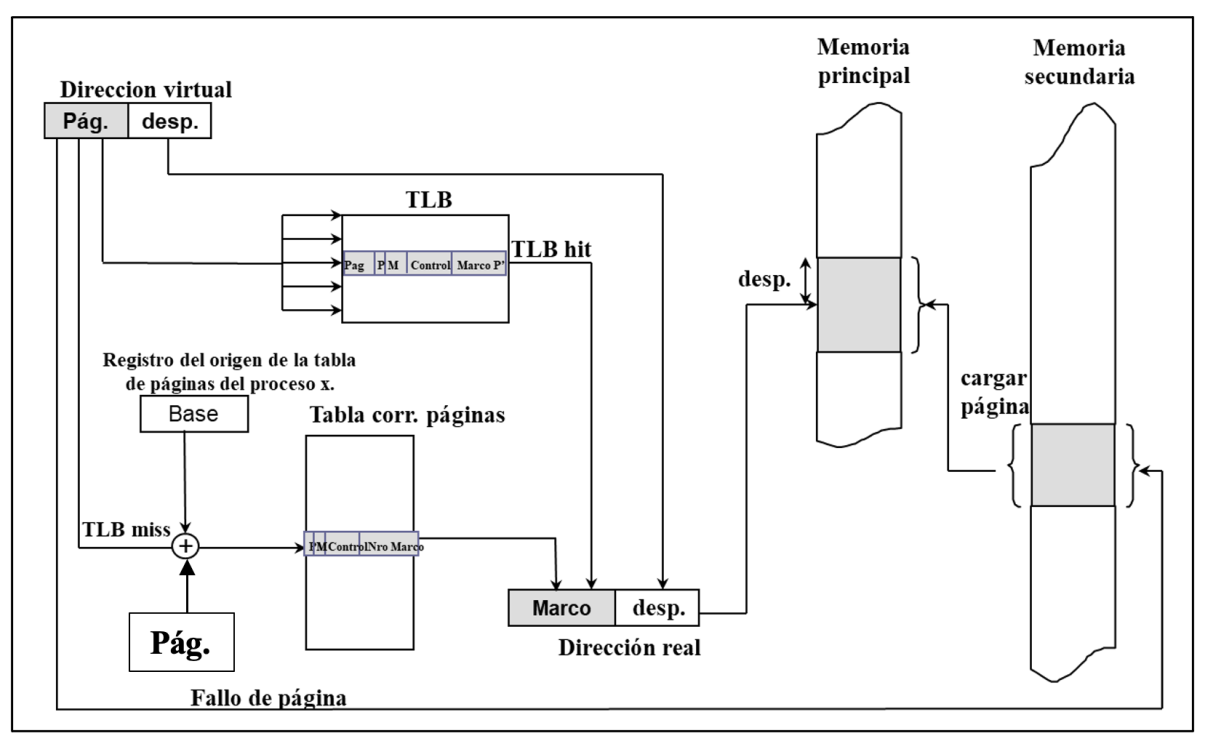
1. **Si no se encuentra la entrada de la tabla de páginas buscada (un fallo de TLB)**:

**El procesador busca la coincidencia con el número de página en la tabla de procesos:**

Si se encuentra activo el bit de presencia, es que la página está en memoria principal y el procesador puede obtener el número de marco de la entrada de la tabla de páginas para formar la dirección real. El procesador, además, actualiza la TLB para incluir esta nueva entrada de la tabla de páginas. Por último, si el bit de presencia no está activo, es que la página buscada no está en memoria principal y se produce un fallo en el acceso a memoria, llamado fallo de página.

En este punto, se abandona el ámbito del hardware y se invoca al sistema operativo, que carga la página necesaria y actualiza la tabla de páginas.

La lectura de pág. es en paralelo(múltiples páginas pueden ser leídas y escritas en la memoria secundaria (como el disco duro) o en la memoria RAM al mismo tiempo, en lugar de tener que hacerlo de forma secuencial y no indexada (no se leen o escriben una por una de manera secuencial, sino que se accede a ellas mediante la utilización de estructuras de datos que permiten acceder directamente a la página deseada.).



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 visibles 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 cuándo se debe cargar una página 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 de 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** selección 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

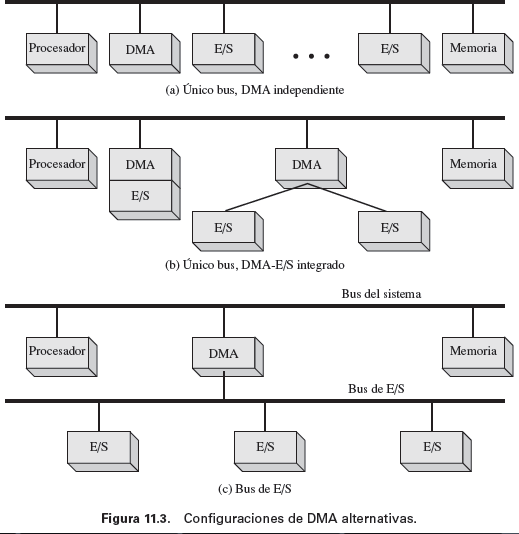
Determinar el grado de multiprogramación: pocos procesos en la memoria principal, entonces menor aprovechamiento de la CPU. Si hay demasiados procesos, el tamaño medio del conjunto residente de cada proceso no será el adecuado y se producirán frecuentes fallos de páginas. El resultado es la hiperpaginación.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

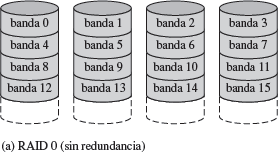
Con múltiples discos que funcionan en forma independiente y en paralelo, las distintas solicitudes de E/S se pueden gestionar en paralelo, siempre que los datos solicitados residan en discos separados. Es más, una única petición de E/S se puede ejecutar en paralelo si el bloque de datos que se pretende 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.

"Hot Spare" en un sistema RAID (Redundant Array of Independent Disks) se refiere a un disco duro adicional que se mantiene en espera y no se utiliza activamente en la matriz RAID a menos que otro disco en la matriz falle. Su propósito principal es actuar como un reemplazo inmediato y automático en caso de que se produzca una falla en uno de los discos activos. Este disco que está en espera copia los datos de otro cuando falla, ejemplo: Si tengo un raid 1, y falla uno de los discos, este entra en accion y copia los datos del disco que sigue funcionando para no perder redundancia. En cambio en un raid 0 si se pierden los datos de un disco no tiene de donde recuperarlos, por lo tanto no funcionaría.

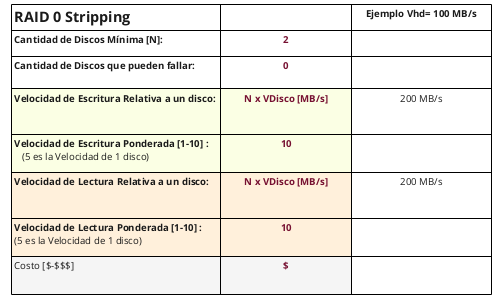
El sistema de RAID consta de 6 niveles posibles, de 0 a 6. Los niveles de 2 a 4 no se comercializan.

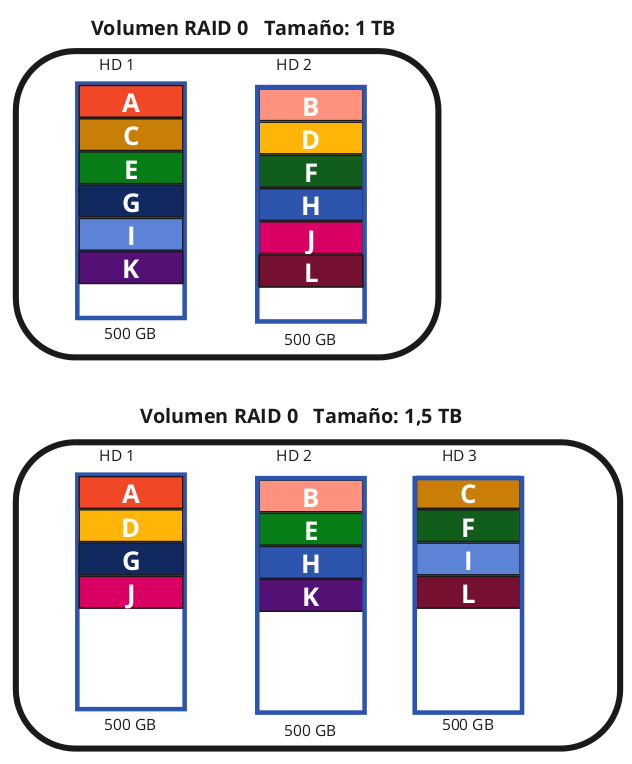
* + **RAID 0 (no redundante)**

No incluye redundancia. Es un RAID “trucho”. Todos los datos están distribuidos a lo largo de todo el vector de discos.



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.



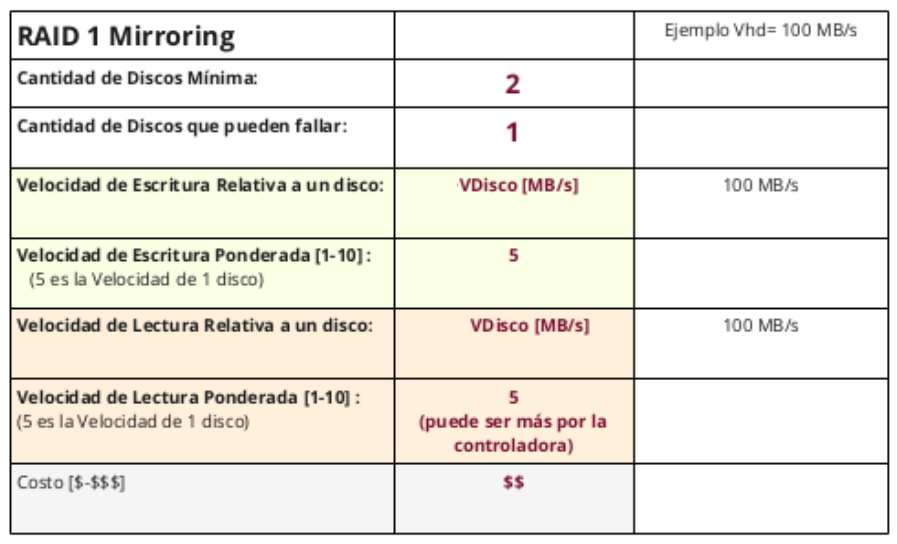


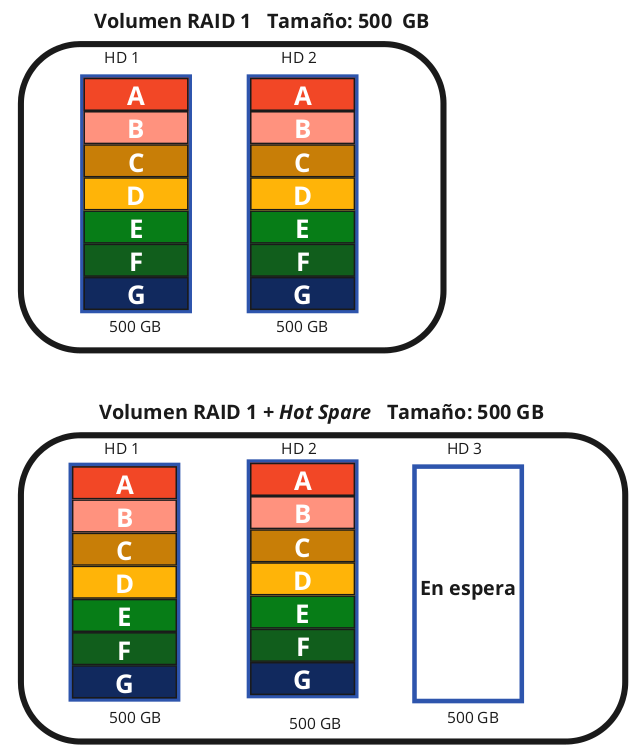
* + RAID 1 (espejo)

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).

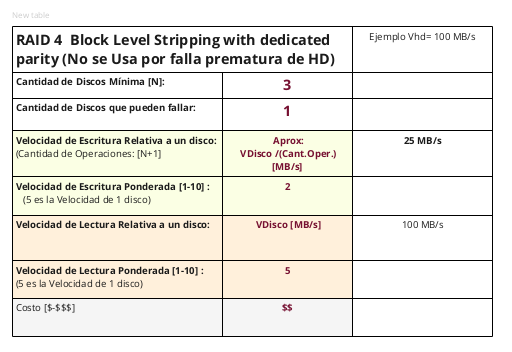
La principal desventaja es el elevado costo por tener que duplicar toda la información en otro disco.







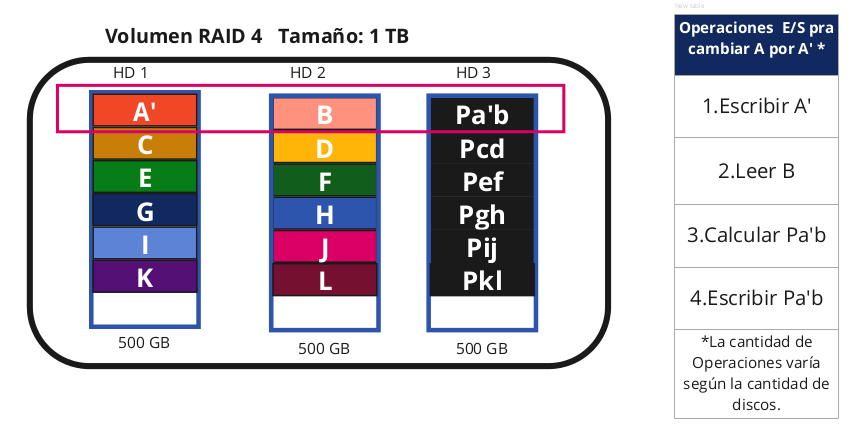
* + RAID 4 (paridad por intercalación de bloques – Acceso Independiente)



La cantidad de operaciones es N+1, siendo N la cantidad de discos utilizados (Se incluye el de paridad)

Es decir que con 3 discos son 4 operaciones, con 5 discos son 6 operaciones

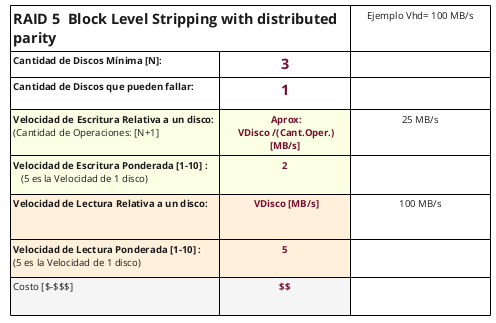


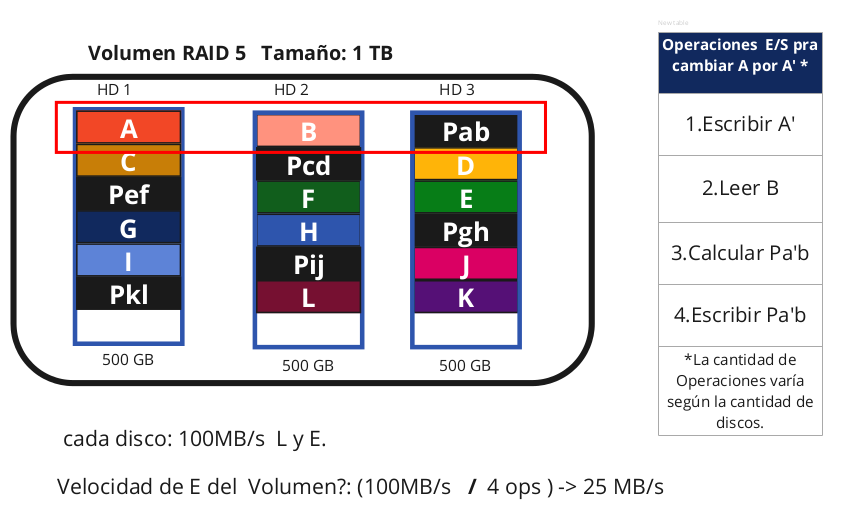


* + RAID 5 (paridad por intercalación distribuida de bloques – Acceso independiente)

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.

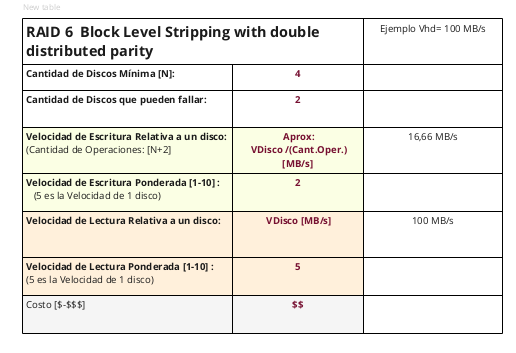


La cantidad de operaciones es N+1, siendo N la cantidad de discos 

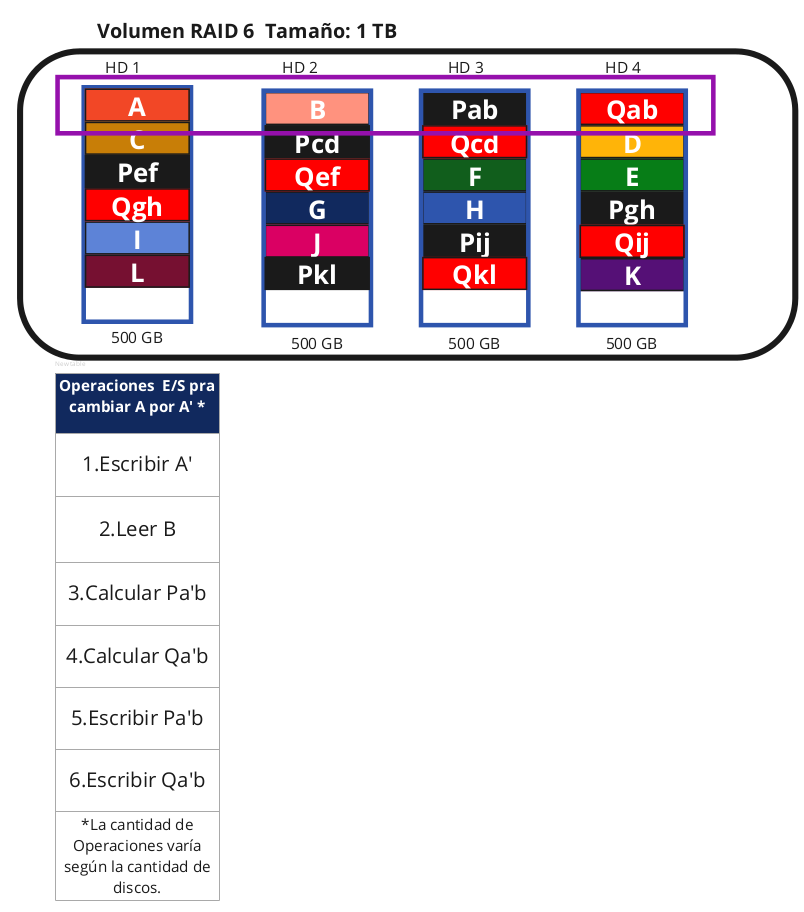
* + RAID 6 (paridad dual por intercalacion distribuida de bloques – Acceso independiente)

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:





La cantidad de operaciones es N+2, siendo N la cantidad de discos



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 caché

Memoria pequeña y de alta velocidad, más rápida que la memoria principal, y que se sitúa entre esta y el procesador. Dicha memoria caché 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.

**FALTA VIRTUALIZACIÓN**

**FINALES DE LA MATERIA**

**BRUNO**:

Desarrollar un mapa del tema que te toca y después ir explicando

RAID:

**Implementar un almacenamiento de 6Tb** con **discos de 1Tb** tiene que tener **fiabilidad**, **soportar fallas** pero **ser barato y tener x velocidad de E/L**

**FACU**:

Me tomaron paginación con memoria virtual, su traducción, que es la caché TLB y el registro de dirección de base de tabla de proceso.

RAID:

Para una base de datos de 10TB con discos de 1TB a 100Mb/s de lectura y escritura. Con los requisitos de lectura (del 0 al 5) un 5 y de escritura un 2.

Ahí hice un raid 50 con 5 raid 5 de 3 discos cada uno, con una velocidad total de lectura de 600Mb/s y de escritura de 125Mb/s

**LUISMA:**

Interbloqueo

Hipervisor 1 , Docker y Containers.