

SISTEMAS OPERATIVOS FOR DUMMIES®

*A Reference
for the
Rest of Us!*

FREE eTips at dummies.com®

QUIERO APROBAR
SO

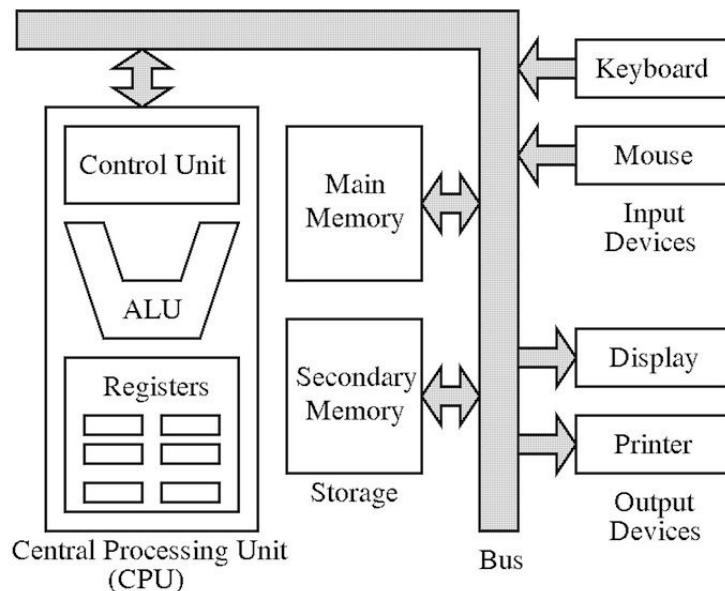


Índice

Repaso de arquitectura	3
Sistemas operativos	9
Procesos	14
Planificación	24
Hilos	38
Arquitectura de Kernel	44
Concurrencia y sincronización	50
Deadlock	65
Memoria Real/Principal/Física	72
Memoria virtual	86
Diseño del SO para memoria virtual	95
File System	99
FAT/EXT2	112
Entrada/Salida	122

Repaso de arquitectura

COMPONENTES BÁSICOS DE UNA COMPUTADORA



Procesador: está formado por un conjunto de registros que almacenan datos, una unidad aritmética (ALU) que realiza operaciones con los datos y una unidad de control que coordina a todos los componentes.

Registros: es una porción de memoria donde el SO almacena información. Se dividen en 2 categorías:

- Registros visibles por el usuario (de uso general): estos registros pueden ser modificados directa o indirectamente (leer o escribir). Por ejemplo, un registro donde guardo el resultado de una cuenta (AX, BX, etc).
- Registros de control y estado: generalmente no pueden ser modificados por un programador pero sí consultados. Los utiliza el HW o SO para funcionar y los modifica de acuerdo a la última instrucción que ejecutó.
 - PC: contiene la dirección de la próxima instrucción a ejecutarse.
 - IR: guarda la instrucción en sí que se está ejecutando en ese momento, pero no la dirección. Por ejemplo, ADD.
 - MAR: contiene la dirección del próximo operando a ser utilizado por la instrucción.
 - MBR: guarda la instrucción que está en la celda cuya dirección tiene MAR. Guarda la próxima instrucción que necesita.
 - PSW (palabra de estado del procesador): indica el estado luego de realizar una operación (overflow, carry, flags).

Memoria RAM: conjunto de direcciones lineales. Es un gran array de información.

Bus: dispositivo capaz de transferir datos entre los distintos componentes de una computadora. Podemos decir que permite la comunicación entre el procesador, dispositivos I/O y memoria principal y está formado por:

- Bus de datos.
- Bus de direcciones.
- Bus de control.

INSTRUCCIONES

Introducción: cada procesador tiene un set de instrucciones que son simplemente las funciones que realiza el procesador, como MOV, ADD, SUB, HLT, etc.

Por ejemplo: $i = i + 1$

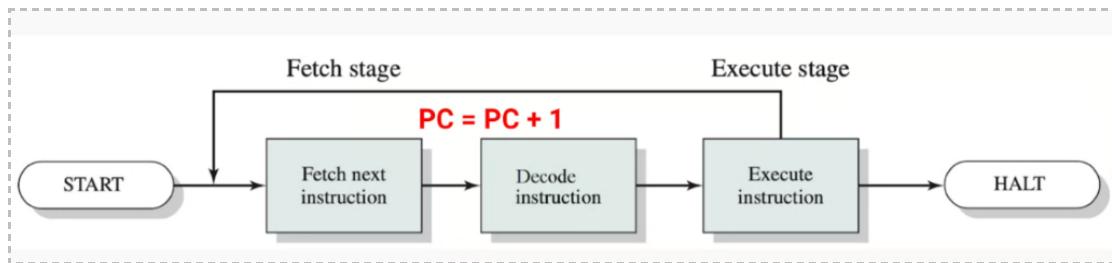
¿Es una instrucción? No. Es un conjunto de instrucciones. A esto se lo llama "sentencia". Esta sentencia lleva por detrás 3 instrucciones (que deberá realizar el procesador):

MOV AC, [100Ah] → Mueve al procesador (al accumulator) el valor de la variable i desde su lugar en memoria.
ADD AC, 1 → Una vez que el valor de i está en el acumulador le suma 1.
MOV [100Ah], AC → Devuelve el valor (ya incrementado) a su lugar en memoria.

Tipos de instrucciones: existe una extensa clasificación de las instrucciones pero lo único que nos importa saber es que hay instrucciones que sólo puede ejecutarlas el SO.

- Instrucciones no privilegiadas: cualquier programa puede ejecutarlas. Ej: MOV, ADD, SUB, JNZ, JZ, CALL.
- Instrucciones privilegiadas: sólo el SO puede ejecutarlas. Ej: CLI, STI, INT, HIT.

Ciclo de instrucción sin interrupciones: este es el ciclo básico de instrucción que no contempla la aparición de interrupciones durante la ejecución. Más adelante vamos a ver el ciclo de instrucción real que es el que sí contempla interrupciones.



Hay que tener en cuenta que sólo los programas que están en la memoria RAM pueden ejecutarse y que, en realidad, este ciclo está compuesto por 6 etapas pero en el gráfico y en la cursada sólo contemplamos 3:

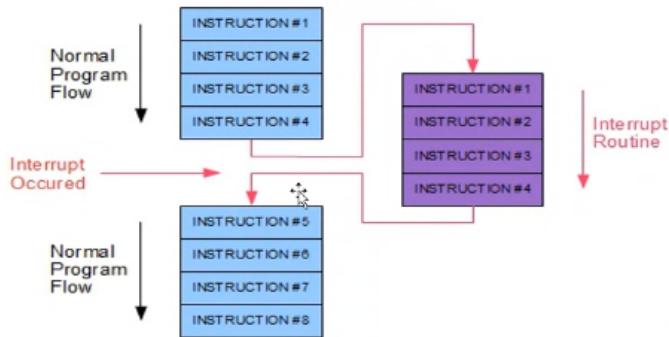
- Fetch: el procesador va a buscar a memoria la siguiente instrucción que tiene que ejecutar. Para saber cuál es debe consultar el PC.
- Decode: decodifica la instrucción, es decir, la traduce para encontrar qué operación es y (en otra etapa que no está en el gráfico) buscar los operandos. La guarda en el registro IR.
- Execute: el procesador ejecuta la instrucción.

Una vez que se completa este ciclo para una instrucción el ciclo vuelve a empezar pero ahora con la siguiente instrucción (a menos que haya habido una instrucción de salto). Si se sigue con la siguiente instrucción se hace PC++, y si hay un JMP el PC salta a la posición especificada.

INTERRUPCIONES

Definición: una interrupción es un mecanismo de HW mediante el cual se da aviso a la CPU de que ha ocurrido un evento. Habitualmente, la interrupción proviene de algún módulo de I/O, de la memoria o de la misma CPU. Cuando un programa es interrumpido se guardan todos los registros del procesador (a esto se lo llama "contexto de ejecución de un proceso", es decir, se guarda el contexto

de ejecución del proceso) antes de pasar a ejecutar la rutina de interrupción. Es decir que se interrumpe la ejecución normal de un programa para ejecutar las instrucciones que indique la rutina de interrupción (esto siempre implica un cambio a modo kernel, ya que este es el encargado de manejar las interrupciones). Una vez que finalice la rutina el SO va a decidir qué otro programa va a ejecutar. Todas las interrupciones deben ser atendidas en algún momento.



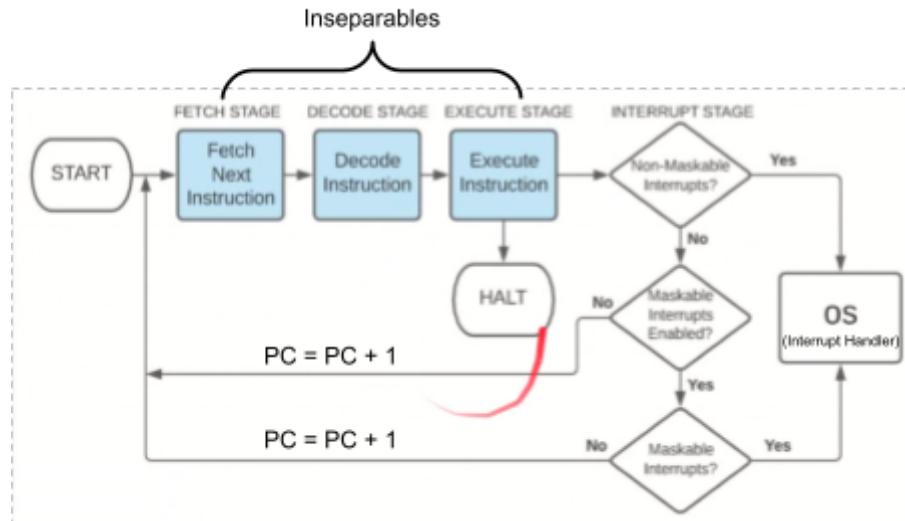
Clasificación:

- Hardware/Software:
 - Hardware: provienen de cualquier lado menos de la CPU. O sea, son *externas* al procesador.
 - Software: son generadas por la propia CPU (como consecuencia del ciclo de ejecución). O sea, son *internas*. Ejemplo: división por cero.
- Enmascarables/No enmascarables:
 - Enmascarables: pueden ser ignoradas temporalmente. Estas interrupciones no son atendidas inmediatamente cuando ocurren. No representan algo crítico dentro del sistema.
 - No enmascarables: tienen que ser atendidas inmediatamente. Representan algo crítico dentro del sistema. Están asociadas a elementos de hardware como la memoria o el disco y quien dispara esta interrupción es el dispositivo que está teniendo problemas.

En la arquitectura de la computadora vienen predefinidas cuáles interrupciones son enmascarables y cuales no enmascarables.
- Sincrónicas/Asincrónicas:
 - Sincrónicas: son las generadas por la CPU al ejecutar instrucciones. Son internas.
 - Asincrónicas: son generadas por dispositivos externos al procesador. Pueden producirse en cualquier momento, independientemente de lo que esté haciendo el procesador. Son externas.
- E/S: son interrupciones que indican que finalizó un evento asociado a un dispositivo de E/S, por ejemplo, una escritura en disco, la lectura de la placa de red, etc.
- Fallas de Hardware: son no enmascarables.
- De clock: son interrupciones que le indican a la CPU que debe desalojar al proceso que está ejecutando actualmente. Son usadas para manejar la planificación de la CPU y permitir la ejecución de varios programas de manera concurrente (multiprogramación).

- Excepciones: son generadas por errores en la programación o por condiciones anómalas (por ejemplo, un fallo de página). Son las de más alta prioridad.
 - Aborts: un error grave ocurrió como fallo de HW.
 - Fallos: pueden ser corregidos y retoman la ejecución.
 - Traps: son utilizadas para debugging.

Ciclo de instrucción con interrupciones: ahora que vimos las interrupciones podemos incorporarlas al ciclo de instrucción.



En INTERRUPT STAGE se pregunta si hubo una interrupción no enmascarable. Si la hubo, se interrumpe inmediatamente el ciclo de instrucción y el PC pasa a apuntar a la rutina de atención de la interrupción. Si no hubo interrupciones no enmascarables, se pregunta si las enmascarables están habilitadas. Si no lo están, se continúa con el ciclo normal de ejecución (o sea, PC++) y, sino, se pregunta si hubo alguna interrupción enmascarable.

Es importante remarcar que la interrupción pudo haberse mandado en cualquier instante (fetch, decode, execute), pero estas etapas son inseparables. Recién se va a preguntar si hubo una interrupción después de ejecutar la instrucción.

Pasos que se generan cuando ocurre la interrupción:

- Hardware (procesador):
 1. Se genera la interrupción en cualquier momento.
 2. Finaliza la instrucción actual normalmente.
 3. Se determina que, efectivamente, ha ocurrido una interrupción en la ejecución y se identifica de qué dispositivo provino.
 4. Se guarda el PC y PSW del programa que estaba ejecutando.
 5. Se carga en el PC la dirección del Manejador de Interrupciones y comienza a ejecutar el SO.
- SO (toma control el Manejador de Interrupciones):
 6. Guarda todo el resto de la información que estaba en el procesador (es decir, los registros AX, BX, registros de estado).
 7. Se inhabilitan las interrupciones. Esto vamos a discutirlo abajo.
 8. Se procesa la interrupción.
 9. Una vez que se hayan ejecutado todas las instrucciones del Manejador de Interrupciones, se restaura la información guardada del procesador.

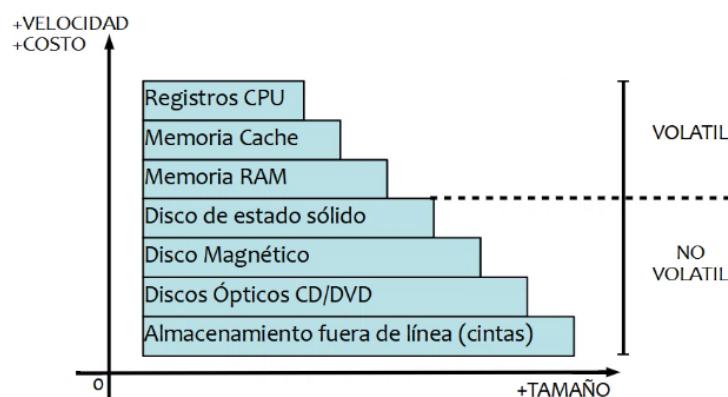
10. Se restaura el PC y el PSW (como resultado, la próxima instrucción que se ejecute va a ser la que le seguía al programa interrumpido. Siempre se restaura primero el PSW porque apenas se restaure el PC la CPU comenzará a ejecutar, por lo que antes debemos restaurar el PSW).
11. Se habilitan nuevamente las interrupciones.

Múltiples interrupciones: ocurre cuando surge una interrupción mientras se está atendiendo otra. Es algo muy común. Hay tres estrategias a seguir:

- Se atienden en orden secuencial.
- Se las ordena por prioridades. Para implementar esta alternativa se deberá definir una prioridad para cada interrupción. Se permitirá que una interrupción de mayor prioridad interrumpa el procesamiento de una interrupción de menor prioridad.
- Deshabilitar las interrupciones mientras una interrupción está siendo procesada.

JERARQUÍA DE MEMORIA

Definición: una computadora tiene mucho espacio de distinto tipo o de distintas características para almacenar información. En el gráfico podemos ver varios. Los de la base son más lentos, tienen más capacidad y son menos costosos. Los de la cima son más rápidos, más pequeños pero más costosos.



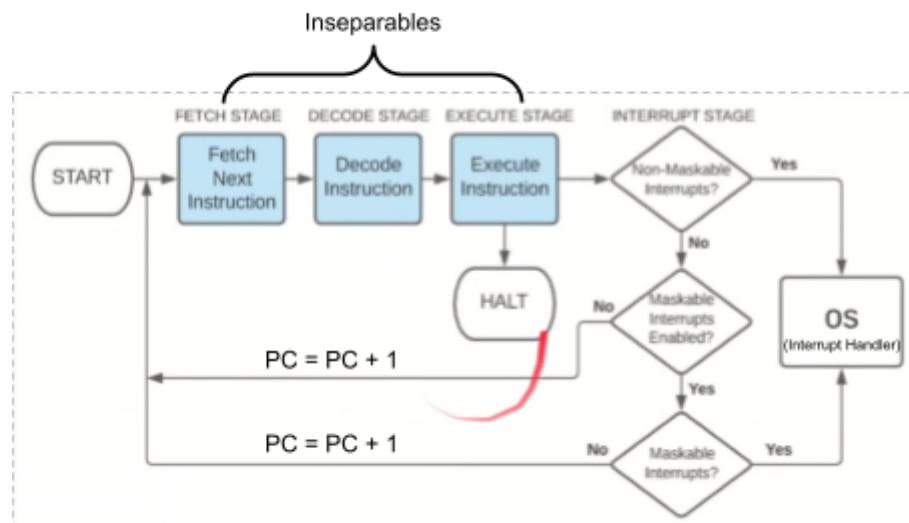
Volátil = apago la computadora y toda la información que estaba ahí se borra.

No volátil = apago la computadora y la vuelvo a prender y los archivos siguen estando.

¿Qué es suspender una computadora? es simplemente mantener un poco de energía en la memoria, mover la información de los registros volátiles a los registros no volátiles y cuando "despertamos" a la computadora los vuelve a cargar en la memoria volátil.

PREGUNTAS DE PARCIAL

1. ¿En qué consiste el ciclo de ejecución de una instrucción? ¿Podría una interrupción surgir como consecuencia de dicho ciclo? (en caso afirmativo brinde un ejemplo).



Sí, podría surgir una interrupción durante la ejecución de una instrucción, por ejemplo, una división por cero.

Sistemas operativos

INTRODUCCIÓN

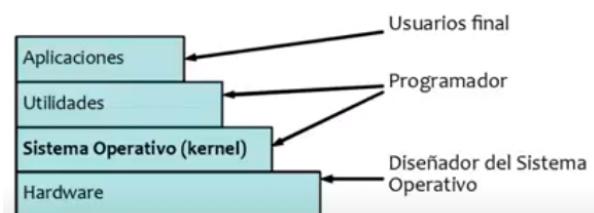
Definición: un SO es un programa o conjunto de programas que administran los recursos físicos (HW), el software y la interfaz de usuario. Además, se administra a sí mismo y se encarga de proveer seguridad en el sistema, asegurando que los programas que permite ejecutar hagan lo que deben hacer y no otras cosas que puedan dañar a la computadora. También permite la gestión de usuarios para que distintas entidades puedan trabajar con él.

Funciones de los SOs:

- Administrar la ejecución de programas.
- Proveer una interfaz para usuarios y programadores.
- Administrar los recursos de HW y SW.
- Administrar los dispositivos de E/S.
- Administrar los archivos.
- Administrar la comunicación entre programas.
- Asignar recursos.
- Brindar protección y seguridad.
- Administrarse a sí mismo.
- Ser interfaz de los dispositivos

Capas de una computadora: todo SO tiene un core (núcleo) donde están desarrolladas las funcionalidades básicas. Pero, el sistema no sería útil si no tuviera aplicaciones, una interfaz de usuario y demás funciones utilitarias (compilador, consola, debugger). Entonces tenemos muchas capas:

- Kernel: contiene las funciones principales de un SO. Es su núcleo. Gestiona los recursos hardware del sistema y suministra la funcionalidad básica del SO. A su vez, el kernel nos va a permitir la sincronización y comunicación entre procesos.
- Distribuciones: son sistemas operativos basados en el núcleo (por ejemplo, Ubuntu es un SO basado en el core Linux) que, además, incluyen determinados paquetes de software con aplicaciones para usos específicos, dando origen a ediciones domésticas, empresariales, educativas, etc.
 - Aplicaciones: destinadas al uso por parte del usuario final. Estas aplicaciones o programas, a la hora de usar el hardware, tienen que pasar por el SO. Los programas no hablan directamente con el hardware, sino que el SO está de intermediario.
 - Utilidades: recursos usados por los programadores para interactuar con el SO y el HW. Por ejemplo: terminales, debugger, compilador.

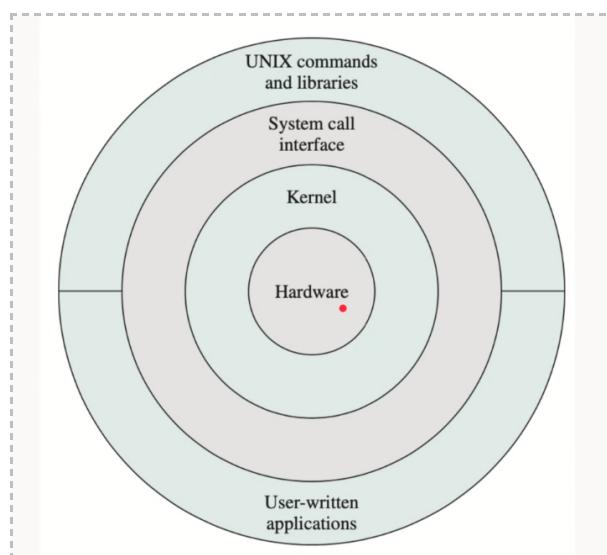


Evolución de los SOs:

- Monoprogramados: permite la ejecución de un sólo programa a la vez y dispone de todos los recursos disponibles para este programa.

- Procesamiento en serie:
- Sistemas en lotes sencillos: permite seleccionar una serie de programas y definir que se ejecute uno después del otro.
- **Multiprogramados:** permite a muchos programas ejecutar de manera concurrente. El SO debe asumir un rol de administrador ya que debe encargarse de la asignación de recursos a los programas que quieran ejecutar.
 - Sistemas en lotes multiprogramados: permiten que, mientras un programa está esperando un evento de un dispositivo de E/S, pueda haber otro programa ejecutando.
 - Sistemas de tiempo compartido: permite a muchos usuarios ejecutar de manera concurrente. La tarea de administración es mucho más compleja dado que no todos los usuarios disponen de los mismos recursos (archivos, programas, permisos).

LLAMADAS AL SISTEMA (SYSCALLS)



Syscalls: funciones incluidas en el kernel del SO mediante las cuales un programa puede solicitar servicios al SO. Estos servicios implican típicamente acceder al HW, o acceder a programas a los que sólo el SO tiene acceso. Es decir, el único modo que tienen las aplicaciones de hablar con el HW es a través de llamadas al sistema. Las mismas son accesibles a través de un lenguaje de programación porque son simplemente funciones, por ejemplo:

- Operaciones sobre archivos: open(), read(), write()
- Operaciones sobre procesos: fork(), exit(), kill()
- Manipulación de dispositivos: release(), eject().
- Otras: time(), sem_wait()

Cada SO tiene definidas sus propias llamadas al sistema. Es decir, los nombres de las funciones y los parámetros varían de sistema a sistema, por lo que no puedo usar las syscalls de un SO en otro. Para resolver este problema hacemos uso de los wrappers a través de una API o biblioteca.

Wrappers: son funciones que pretenden ser estándares a todos los sistemas operativos y que envuelven a las llamadas al sistema. Es decir, es el wrapper el que se encarga de hacer la llamada al sistema.

Por ejemplo, existe la librería estándar de C y cada SO tiene una implementación de esta librería. En mi programa puedo usar las funciones wrapper que ofrece esta biblioteca en vez de las específicas

de mi SO. Ambas hacen lo mismo pero usando los wrappers hago que mi programa sea compatible con cualquier SO que tenga una implementación de la librería estándar de C.

Los wrappers aportan:

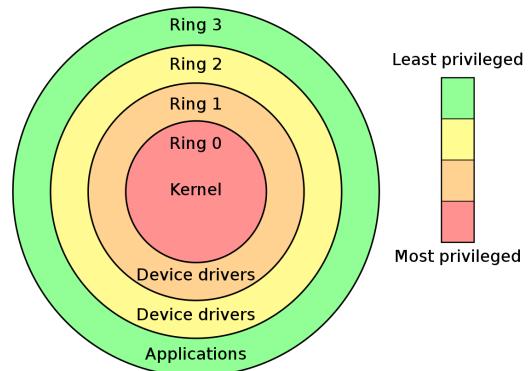
- Portabilidad: permite ejecutar mi programa en cualquier SO.
- Simplicidad: me ahorro detalles del SO que no me interesan.
- Eficiencia.

Hacer uso directo de las funciones de cada SO me otorga mayor precisión (porque suelen ser funciones con muchos parámetros) pero menor simplicidad y portabilidad.

MODOS DE EJECUCIÓN

Definición: los SOs manejan un “anillo de protección” (que habitualmente es entre 0 y 3) y, de acuerdo al nivel en el que estemos, podemos ejecutar ciertos tipos de instrucciones. Habitualmente se habla únicamente de dos modos de ejecución (el más privilegiado y el menos privilegiado):

- Kernel (0): usado por el SO. Puede ejecutar cualquier tipo de instrucciones. Accedemos al modo kernel mediante interrupciones o syscalls.
- Usuario (3): sólo puede ejecutar instrucciones no privilegiadas. Los programas ejecutan únicamente en modo usuario. Si un proceso quiere hacer uso de una instrucción privilegiada debe realizar una llamada al sistema.



CAMBIO DE MODO (MODE SWITCH)

Definición: cambio entre los modos de ejecución posibles.

- Las aplicaciones siempre ejecutan en modo usuario.
- El SO siempre ejecuta en modo kernel.



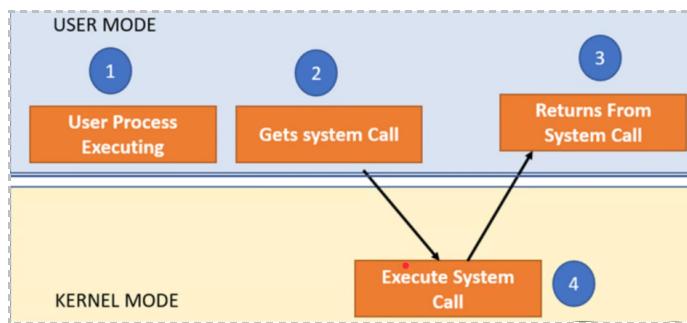
Al ejecutar distintos procesos se producen cambios de modo constantemente. Nunca puede ocurrir que ejecute una aplicación e inmediatamente otra porque es el SO quien decide cuál sigue. Debe haber siempre un cambio a modo kernel. Para que deje de ejecutar una aplicación pueden ocurrir dos cosas:

- Que aparezca una interrupción (suponiendo que se decida atenderla). Para atenderla deberá ejecutar el SO, entonces cambiará a modo kernel.
- Cuando la aplicación realice una llamada al sistema.

¿Cómo sé en qué modo estoy ejecutando? Le pregunto al PSW.

¿Cuándo se produce?: un usuario no puede cambiar el modo de ejecución. El cambio se produce cuando un programa realiza una llamada al sistema. Todos los programas comienzan en modo

usuario. Cuando se recibe la syscall empieza a ejecutar el sistema operativo y hace lo que tiene que hacer. Después se devuelve el resultado al proceso, que continúa ejecutando en modo usuario.



Ejemplo de cambio de modo luego de una syscall.

¿Puede un programa ejecutar una instrucción privilegiada?: no, el programa no puede ejecutar la instrucción. Se produce una interrupción y se llama al SO. El SO le manda una señal al programa para que finalice (un segmentation fault) o toma alguna acción. En otras palabras, puede intentar ejecutarlo pero nunca lo va a lograr. Se va a producir una interrupción.

PREGUNTAS DE PARCIAL

1. ¿Cuál es la diferencia entre una Syscall y una función Wrapper? ¿En qué caso utilizaría cada una y por qué?

Las syscalls son funciones definidas en el kernel de un SO mediante las cuales los procesos pueden solicitarle servicios (por ejemplo, un recurso). Las syscalls son propias del kernel de cada SO por lo que suelen tener nombres y parámetros distintos, por lo que carecen de portabilidad y simplicidad. Para resolver este problema existen los wrappers.

Los wrappers son funciones (ofrecidas por bibliotecas o APIs) que pretenden ser estándar a todos los SO y que envuelven a las syscalls. Es decir, un wrapper es un “intermediario” entre el proceso y la syscall. Permite que un programa sea compatible con cualquier SO que implemente la biblioteca en cuestión además de ser más simple.

2. En un sistema que tiene modos de ejecución para garantizar la protección, indique.

a. ¿Cómo sería la forma correcta de realizar una operación sobre el HW?

Los programas corren en modo usuario y si quieren realizar una operación sobre el HW no pueden hacerlo ellos mismos, sino que deben hacer una llamada al sistema para que el SO ejecute la operación en modo kernel.

b. Indique una forma “incorrecta” y explique por qué no sería permitida.

La forma incorrecta sería que el programa ejecutara la instrucción en modo usuario. Si es una instrucción que realiza una operación sobre el HW seguro es privilegiada y nunca podrá ser ejecutada en modo usuario porque no sería seguro.

3. Describa brevemente qué ocurre cuando se invoca una syscall desde un proceso en modo usuario.

- El proceso está ejecutando en la CPU y realiza una syscall.
- Se guardan los registros del procesador.
- Se cambia a modo kernel y se bloquea al proceso que hizo la syscall.
- El kernel identifica en la syscall table qué rutina utilizar para atender la misma.
- El kernel ejecuta la rutina.
- Se mueve el resultado de la syscall al stack del programa.
- Se produce una interrupción y el proceso vuelve a estado Ready. También se cambia a modo usuario para que continúe ejecutando algún proceso.

4. V o F.

a. Los wrappers de las syscalls permiten que se realice el cambio de modo de usuario a kernel.

Falso. Los wrappers son simplemente funciones que por detrás se encargan de hacer la correspondiente syscall. Entonces, es la syscall la que se encarga de realizar el cambio de modo usuario a modo kernel para que el SO pueda atenderla.

b. Nunca puede ocurrir un cambio de contexto sin realizar un cambio de proceso.

Falso. Se puede estar atendiendo una syscall y que surja una interrupción por lo que el contexto cambiaría de la rutina de la syscall a la del interrupt handler sin haber un cambio de proceso.

Puede darse también en el caso de cambio de KLTs, dado que se cambia de hilo pero no de proceso.

5. V o F. Si se estaba ejecutando una syscall y ocurre una interrupción se esperará a que finalice su ejecución para atenderla, por ser código del SO.

Falso. Luego de cada ciclo de instrucción se valida si hay interrupciones, por lo que por más que se esté ejecutando la rutina de la syscall, se atenderá la interrupción al finalizar de ejecutar la instrucción en curso. Luego de atender la interrupción se volverá a la syscall.

6. Compare los conceptos de wrapper y syscall en términos de simplicidad, flexibilidad y portabilidad.

	Simplicidad	Flexibilidad	Portabilidad
Syccalls	No son funciones sencillas dado que suelen tener muchos parámetros específicos a la operación que se quiera realizar.	Al estar parametrizadas son muy flexibles.	No son portables. Cada SO tiene sus propias syscalls con nombres y parámetros diferentes.
Wrappers	Son funciones más sencillas y con menos parámetros.	Son mucho menos flexibles que las syscalls.	Son portables dado que para utilizar los wrappers sólo es necesario implementar alguna API o biblioteca estándar.

7. Defina brevemente llamada al sistema (syscall). ¿Qué relación tiene con los modos de ejecución y las instrucciones privilegiadas?

Una syscall es una función definida en el kernel de un SO que puede ser llamada por cualquier proceso para solicitarle un servicio. Una syscall implica un cambio a modo kernel para atender la petición dado que el SO es el único que puede ejecutar instrucciones privilegiadas.

Procesos

DEFINICIONES PREVIAS

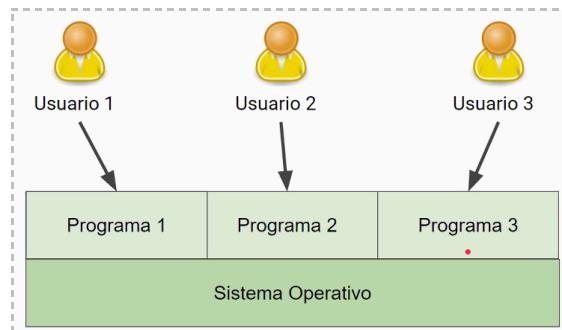
Programa: secuencia de instrucciones compiladas a código máquina. Es estático. A diferencia del proceso, que es dinámico.

Ejecución concurrente: dos o más programas ejecutando en el mismo intervalo de tiempo. No es en el mismo instante. En un instante hay sólo uno.

Sistema monoprogramado: el procesador puede ejecutar solo un programa en un período determinado.

Multiprogramación: modo de operación que permite que dos o más programas ejecuten de forma concurrente (mientras un programa ejecuta el otro permanece en espera). La multiprogramación tiene dos ventajas principales:

- Permite una reducción de tiempos.
- Permite ejecutar a múltiples usuarios de forma concurrente.



Multiprocesamiento: no es lo mismo que multiprogramación. Hace referencia a más de un programa ejecutando en el mismo instante. Nos obliga a tener más de un procesador ya que esta es la única forma de tener a dos programas ejecutando en el mismo instante.

PROCESO

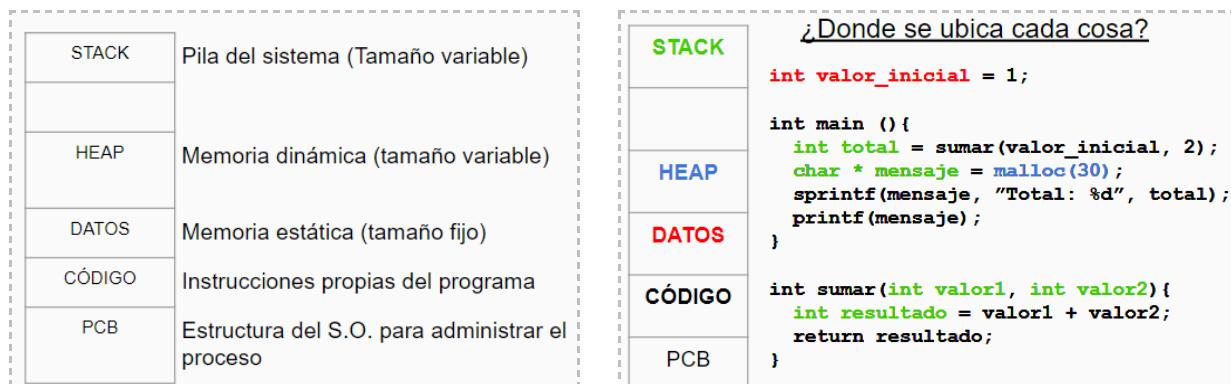
Definición: es un programa que está siendo ejecutado en un instante determinado y, además, se le asignó memoria o recursos para que pueda funcionar. También puede decirse que un proceso es una instancia de un programa. Los procesos son la unidad de trabajo de un SO. Están formados por:

- Secuencia de instrucciones que debe ejecutar el procesador.
- Conjunto de datos.
- Estado (nos da la información precisa de lo que está haciendo el proceso en un momento determinado).
- Atributos que lo identifican.
- Tiene los recursos que le fueron asignados (archivos que fue abriendo, memoria que fue reservando, semáforos, sockets, etc.).

Entonces, un proceso es toda la estructura que permite que un conjunto de instrucciones se pueda ejecutar. Todo proceso debe estar representado en algún lugar, este lugar es la memoria RAM.

Entorno de un proceso: conjunto de variables que se le pasan al proceso al momento de su ejecución.

Estructuras que contiene un proceso/Imagen de un proceso:

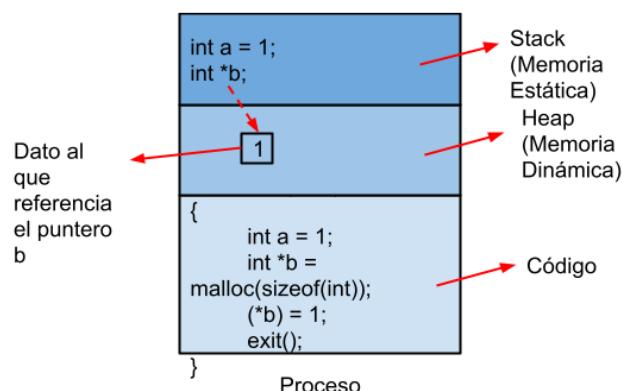


- Código: espacio asignado para almacenar la secuencia de instrucciones del programa. Las instrucciones (así como están, en lenguaje de máquina) se cargan del disco a la memoria. Los procesos no lo pueden modificar.
- Datos: espacio asignado para almacenar las variables globales. Todas las variables que estén definidas por fuera de la función main van acá. No lo puede modificar el proceso.
- Stack (memoria estática): es memoria que se reserva al declarar variables de cualquier tipo de dato (int, float, char, double, etc), así como también punteros a tipos de datos (hay que aclarar que se guarda la *declaración* del puntero, no el dato al que referencia. Ese dato se guarda en el HEAP). El stack es una estructura en forma de pila y el programador no deberá encargarse de liberarla.

Todos los datos que guarda son:

- Variables locales (locales a la función que se está ejecutando). Si la variable está inicializada (ver ejemplo) también se guarda el valor. El programador no puede modificar el espacio en memoria que ocupa una variable. Por ejemplo, si un int ocupa 4 bytes siempre ocupará eso.
- Llamadas a funciones que el programa va haciendo. Al producirse una llamada a una función, se almacena en el stack la dirección a la que debe retornar la ejecución tras finalizar la llamada. Una vez que se cambia el contexto de ejecución se borran las variables.
- Parámetros.
- Retorno de las funciones.

Los procesos pueden modificar el stack.



- Heap (memoria dinámica): memoria que se reserva en *tiempo de ejecución*. La vamos reservando a medida que necesitamos más espacio y liberamos cuando no necesitamos. Es decir, es responsabilidad del programador liberarla cuando no la utilice más. Varía dependiendo de las necesidades del proceso, es decir, el proceso lo puede modificar. Algunas ventajas que ofrece la memoria dinámica frente a la memoria estática es que podemos reservar espacio para variables de tamaño no conocido hasta el momento de la ejecución (por ejemplo, para listas o arrays de tamaños variables), o bloques de memoria que, mientras mantengamos alguna referencia a él, pueden sobrevivir al bloque de código que lo creó.
- PCB (process control block): es una estructura donde se guarda el contexto de ejecución de un proceso y es usada especialmente para la multiprogramación. Hay uno por cada proceso en el sistema. Es creado y gestionado por el SO. Contiene la información necesaria para que el SO administre al proceso y pueda guardar el contexto, en caso de un cambio de contexto solicitado por la CPU para ejecutar otra cosa. El procesador tiene el contexto del proceso que está ejecutando en ese instante, entonces, si se quiere ejecutar un proceso que ya venía ejecutando antes, primero hay que cargar su PCB en el procesador. Los procesos no pueden modificarlo.
El PCB está siempre cargado en RAM y está compuesto por:
 - PSW: el estado del proceso
 - Identificador:
 - PID: Process ID.
 - PPID: identificador del padre del proceso.
 - UID: identificador del usuario que inició el proceso.
 - IP/PC
 - Registros del procesador
 - Información de planificación de CPU (prioridad de ejecución de un proceso)
 - Información de manejo de memoria
 - Información de E/S
 - Información contable (por ejemplo, tiempo en CPU)

¿Qué ocurre si declaro un puntero y no hago free()? cuando declaro un puntero, en el stack se guarda la dirección de memoria al heap donde van a estar los datos. Cuando termine de ejecutar ese contexto de ejecución, el stack se va a borrar y se va a perder la dirección al heap. Si no hice un free() antes de esto, esos lugares en el heap van a quedar ocupados y perdemos la referencia a ellos.

Tiempo de vida de los datos: todos los datos tienen un tiempo de vida, nada persiste para siempre. En C, hay tres tipos de duración:

- Estática: son aquellas variables que se crean una única vez junto con la creación del proceso y se destruyen junto con la destrucción del mismo. Son únicas y generalmente pueden ser utilizadas desde cualquier parte del programa. Para generar una variable estática se la puede declarar por fuera de la función principal (arriba del main(), por ejemplo), o bien usando el calificador *static*.
- Automática: son aquellas variables locales que no son declaradas con el especificador *static*. Se crean al entrar al bloque en el que fueron declaradas y se destruyen al salir de ese bloque. Por ejemplo, el tiempo de vida de las variables internas de una función es lo que tome ejecutarla.
- Asignada: es la memoria que se reserva de forma dinámica (en el heap) y que se explicó más arriba.

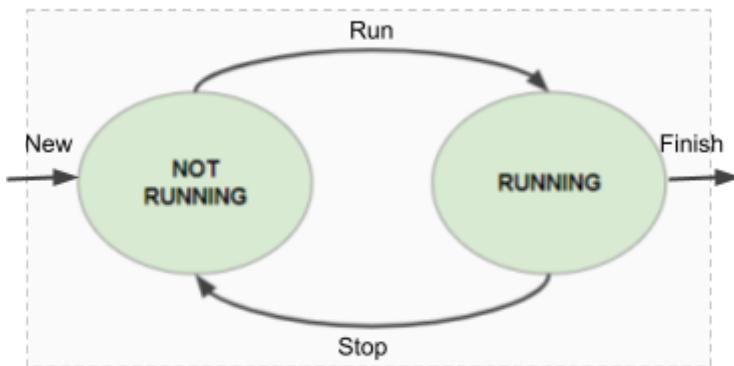
CICLO DE VIDA DE UN PROCESO

Definición: tiempo que transcurre entre la creación y finalización de un proceso. Durante este lapso, un proceso pasa por varios estados.

ESTADOS DE UN PROCESO

Definición: es el comportamiento de un proceso. Conocer su estado nos permite entender en qué condición se encuentra.

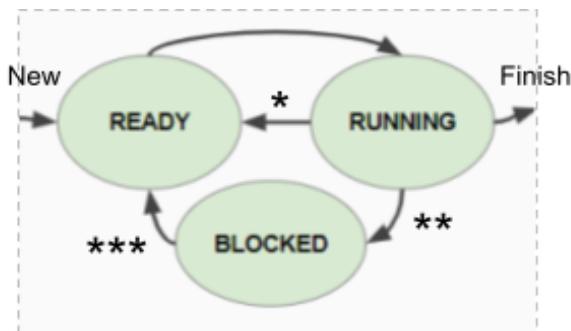
Diagrama de dos estados:



- Running/Ejecutando: un proceso que está usando la CPU.
- Not running/No ejecutando: un proceso (o muchos) que está esperando usar la CPU.

Diagrama de tres estados:

el estado not running se divide en 2:



- Ready/Listos: es una cola que contiene todos los procesos que están listos para ejecutar, es decir, que son elegibles por el procesador. Ya deben tener sus registros (PCB) en memoria. (ver más adelante la diferencia con el estado New/Nuevo). El dispatcher es quien se encarga de mandar los procesos a running.
*: puede ocurrir que un proceso vuelva de Running a Ready sin bloquearse si simplemente se le acabó todo el tiempo que tenía designado para ejecutar (interrupción de clock).
- Blocked: son procesos que ya ejecutaron en la CPU, pero que no pueden ser asignados al procesador porque están esperando que ocurra algún tipo de evento. Un proceso que está bloqueado nunca puede pasar a Running directamente, sino que antes debe pasar a Ready y ser elegido por el SO para ejecutar nuevamente.
**: cuando un proceso pasa de Running a Bloqueado es porque venía ejecutando e hizo una llamada al sistema bloqueante.

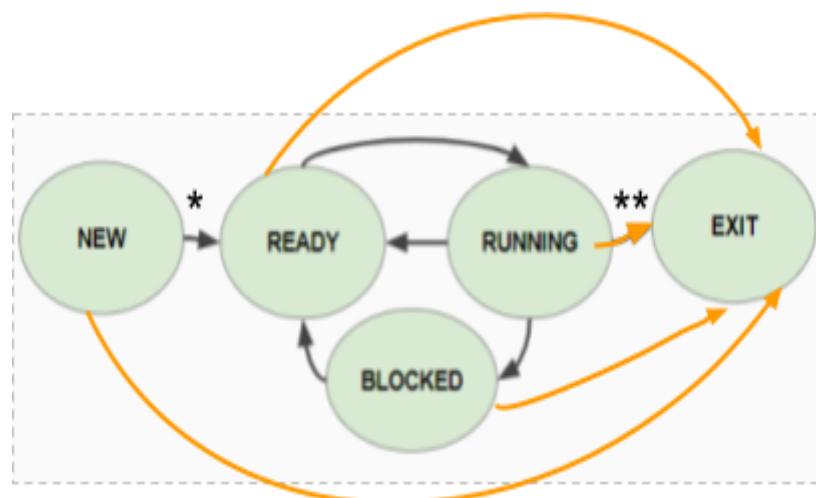
***: para que un proceso pase de Bloqueado a Ready, debe avisarle al SO que el evento que estaba esperando ya finalizó. Esto lo informa mediante una interrupción.

Llamadas al sistema bloqueantes o no bloqueantes:

- Bloqueante: se ejecutan en el momento en que se llaman. Hacen que un proceso se bloquee.
- No bloqueante: el proceso, una vez que ejecuta la syscall, no necesariamente pasa al estado de bloqueado. El proceso no sale de running, salvo cuando termina de ejecutar. En general, la syscall va a ser no bloqueante cuando un proceso no la necesite urgentemente para seguir ejecutando.

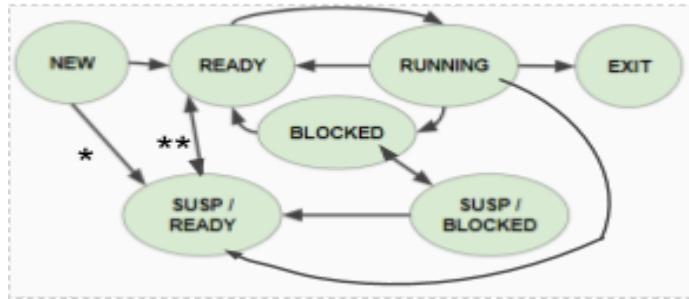
	<i>Bloqueante</i>	<i>No bloqueante</i>
Operación se puede realizar inmediatamente	Realiza la operación	Realiza la operación
Operación va a tardar mucho (indefinido)	Bloquea el proceso. Sigue con otros procesos que tardan menos	<ul style="list-style-type: none"> • No realiza la operación en <u>ese</u> momento. • Continúa ejecutando.
Valores de retorno	Ok/Error	Ok/Error/Reintentar

Diagrama de cinco estados:

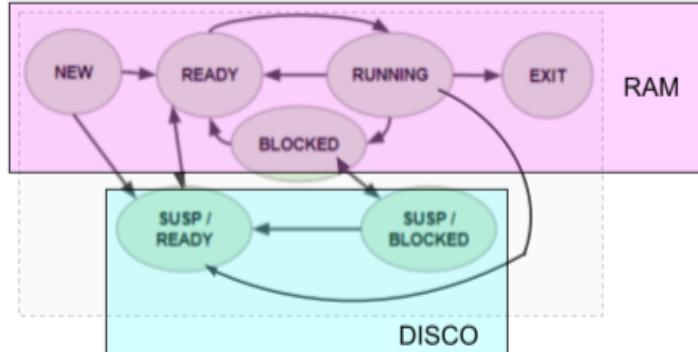


- New/Nuevo: cuando se quiere ejecutar un programa, pasa un tiempo entre que el CPU recibe el mensaje de ejecutar y cuando realmente se empieza a ejecutar. Esta es la etapa New. Acá ocurren varias cosas:
 - Se preparan las estructuras.
 - Se inicializa el PCB.
 - Se espera la admisión.
- *: una vez que terminan estas tareas el proceso pasa a Ready.
- **: puede pasar por tres motivos: porque ejecutó su última instrucción o por un error (que puede traducirse en una interrupción), o que el proceso se mate a sí mismo (kill).
- Exit/Finalizado: de cualquier estado puede pasarse al estado exit. Acá se libera la memoria, recursos, semáforos, etc.
 - Valor de retorno: es importante que cuando un proceso finaliza guarde su valor de retorno por si otro proceso quiere consultararlo.
 - Todas las estructuras menos el PCB se eliminan. El PCB se almacena para fines estadísticos/contables.

Diagrama de siete estados: faltan las flechas naranjas que indican que de cualquier estado puede pasarse a exit. En este diagrama se agrega la idea de suspender un proceso. Hoy en día no tiene mucho sentido pero antes se usaba para pasar las estructuras de un proceso al disco para liberar memoria y poder ejecutar más procesos.



- Susp/Ready: la diferencia con Ready es que estos se encuentran en el disco y deberán ser pasados a memoria para ejecutar.
 - *: un proceso pasa de new a Susp/Ready cuando hay demasiados procesos en Ready. Esto es porque Ready está en memoria principal y Susp/Ready en disco. De todas formas, el pcb del proceso se pasa a Ready y si se decide ejecutarlo se irá a buscarlo a disco.
 - **: cuando el nivel de multiprogramación disminuye y se decide traer un proceso que estaba en disco a RAM o, caso contrario, aumenta el nivel de multiprogramación y se decide mandar a disco a un proceso.
- Susp/Blocked: se toman los procesos que están bloqueados (y que ocupan mucha memoria) y se pasan sus estructuras (menos el PCB) al disco. Si un proceso está siendo debuggeado también va a estar acá.



Estados en Linux:

- R (runnable/running): es la conjunción del estado running y ready.
- S (sleep): es un blocked “interrumpible”.
- Super bloqueado: no se puede interrumpir.
- T: stopped.
- ±: me indica que está en el foreground.
- Z (zombie): estado exit. El proceso queda en este estado hasta que alguien lo reclama. El proceso terminó de ejecutar pero el PCB sigue estando en memoria.
- D (uninterruptable sleep).
- X (dead).

CREACIÓN DE PROCESOS

Formas de creación de un proceso:

- Lo crea el SO para dar algún servicio.
- Lo crea otro proceso.
 - El proceso que solicita crearlo se llama padre y al proceso creado se lo llama hijo.
 - Proceso padre e hijo pueden ejecutar de forma concurrente o el padre puede esperar a que los hijos finalicen. El proceso padre y el proceso hijo pueden ser dos procesos completamente distintos.

Pasos para crear un proceso:

1. Asignar PID.
2. Reservar espacio para estructuras (código, datos, pila y heap).
3. Inicializar PCB.
4. Ubicar PCB en listas de planificación.

Creación de un proceso en Linux: los procesos hijos se crean a través de la syscall fork() y se les asigna el valor 0. Lo que hace fork es duplicar al proceso que lo llamó pero con un nuevo PID, agregándole el PPID (que es el PID del proceso padre) y los registros, pila, heap, se copian igual. Incluso el PC se copia.

Cuando un proceso padre crea un hijo, puede ocurrir que el padre siga ejecutando concurrentemente con su hijo o que espere a que su/s hijo/s termine/n o, incluso, pueden ejecutar distintas partes del programa (desviándolos con IFs). Puede ocurrir que el proceso sea la copia exacta del padre o que se le de una nueva imagen que reemplace la anterior (execv). En cuanto a los recursos (CPU, memoria, archivos y dispositivos), el hijo los puede pedir al SO o puede estar restringido al uso de un subset de los recursos del padre.

Tabla de procesos (TDP): cuando un proceso es creado, es agregado a la tabla de procesos del sistema. Es manejada por el SO y el grado de multiprogramación es definido a partir de la cantidad de procesos activos en la TDP.

FINALIZACIÓN DE PROCESOS

Formas de finalización de un proceso: hay varias maneras de que finalice un proceso

- Lo finaliza el SO (kill).
- Lo finaliza otro proceso (kill).
- Se finaliza el mismo:
 - Normal exit.
 - Abnormal exit.

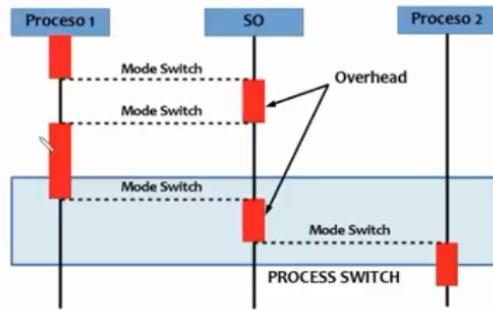
Finalización de un proceso en Linux: un proceso le indica al SO que finaliza con la syscall `exit()`. Un proceso hijo le indica al padre que finaliza y su resultado con la syscall `wait()`.

Un padre puede interrumpir la ejecución de sus hijos con la syscall `abort()`. Esto se puede dar si un hijo utilizó más recursos de los que debía, o la tarea que realizó el hijo ya no es necesaria, o el proceso padre debe terminar, o el proceso padre se lleva a cabo en el proceso abuelo.

En versiones anteriores de linux si un hijo se quedaba sin padre pasaba a ser hijo del proceso `init`.

CAMBIO DE PROCESOS

Definición: se produce cuando viene ejecutando un proceso en el procesador, por algún motivo deja de ejecutar y comienza a ejecutar otro. El cambio de contexto puede tener como objetivo: ejecutar otro proceso, atender una interrupción o ejecutar una syscall.



- Se debe guardar el contexto de ejecución del proceso 1 para luego poder reanudarlo desde que fue interrumpido. Por lo tanto, ese tiempo es considerado overhead y se debe minimizar.

PREGUNTAS DE PARCIAL

- 1. ¿Quiénes pueden crear o finalizar procesos en un sistema? ¿Cómo lo hacen? ¿Qué le sucede al proceso hijo si su padre finaliza inesperadamente?**

Los procesos pueden ser creados por otros procesos o por el SO. En cualquiera de los casos para crear un proceso es necesario hacer una llamada al sistema y el SO se encargará de:

- Asignarle un PID.
- Reservar espacio para estructuras.
- Inicializar su PCB.
- Mover el PCB a las listas de planificación.

En caso que el proceso padre finalice el hijo podrá seguir ejecutando normalmente.

- 2. ¿Qué comparten un proceso padre y un proceso hijo? ¿Y en el caso de dos hilos del mismo proceso?**

Un proceso padre y su hijo no comparten cosas. La única relación entre ellos es que el hijo conoce el ID de su padre (PPID)

Dos hilos del mismo proceso comparten PCB, código, datos y heap.

- 3. Proporcione ejemplos para las siguientes transiciones entre estados: Running -> Ready, Suspended/Ready -> Ready, Ready -> Exit, Ready -> Blocked.**

Running -> Ready: interrupción de clock (un proceso que agotó su quantum) o un proceso que fue desalojado porque llegó otro proceso a Ready con mayor prioridad.

Suspended/Ready -> Ready: cuando el nivel de multiprogramación disminuye y se decide traer un proceso que estaba en disco a RAM.

Ready -> Exit: cuando se decide finalizar un proceso y el mismo no estaba ejecutando.

Ready -> Blocked: esta transición no es posible.

Planificación

INTRODUCCIÓN

Problema que trae la multiprogramación: cuando hay muchos procesos queriendo ejecutar al mismo tiempo, se puede caer en alguno de los siguientes escenarios:

- Disparidad de tiempo en el uso de la CPU. Algunos procesos ejecutan más tiempo que otros en la CPU. ¿Qué pasa si tengo la CPU ocupada por un proceso muy largo que no le da lugar a otros?
- Procesos que no llegan a ejecutarse nunca.
- Degrado del tiempo de respuesta del sistema.
- Memoria RAM llena y CPU sin utilizarse¹.

El problema, entonces, es: ¿cómo logro una multiprogramación eficiente? Si no tengo cuidado puedo terminar en alguno de los escenarios descritos arriba. La clave es realizar una coordinación (planificación/scheduling).

Nivel de multiprogramación: cantidad de procesos activos en memoria principal (es decir, cuántos procesos o hasta cuántos tengo ejecutando en el sistema en ese momento). Toma en cuenta los procesos en estado Ready, Running y Blocked.

Tipos de procesos:

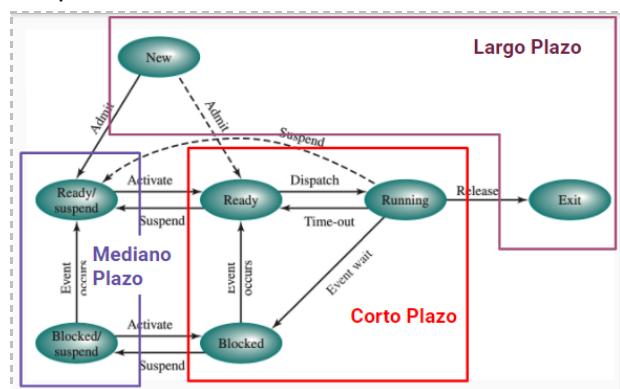
- Limitados por CPU (CPU bound): aquellos procesos que requieren más tiempo de CPU (porque realizan más cálculos) que de uso de los dispositivos de E/S.
- Limitados por E/S (IO bound): realizan cálculos muy básicos pero hacen uso de los dispositivos de E/S constantemente.

PLANIFICACIÓN

Objetivos de la planificación:

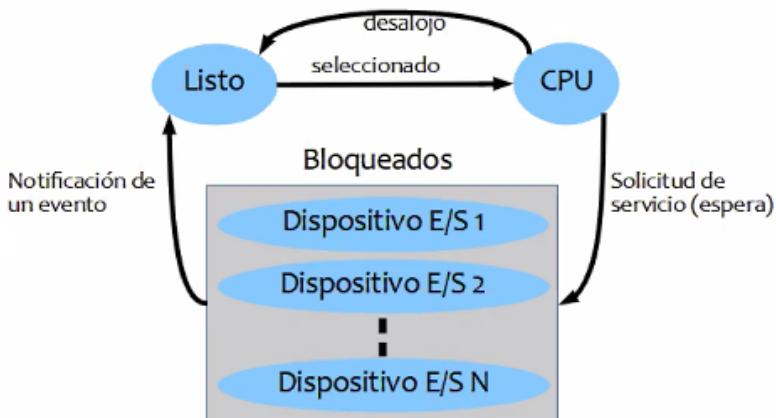
- Asignar procesos al procesador.
- Rendimiento/productividad.
- Optimizar algún aspecto del comportamiento del sistema.

Tipos de planificación: los nombres (largo, mediano y corto plazo) hacen referencia a la “frecuencia” con la cual intervienen estos planificadores.



¹ En realidad, la CPU nunca está sin utilizarse. Siempre está ejecutando. Si no ejecuta un proceso de usuario, se ejecuta el proceso idle del SO.

- Largo plazo: su objetivo es determinar qué procesos pueden o no ser admitidos en el sistema (es decir, controla quiénes pueden pasar a la cola de Ready y cuándo). Esto determina el grado de multiprogramación del sistema. Se incluye al estado Exit porque si está muy saturado puede mandar algún proceso a Exit.
 - Admitir un proceso (pasarlo a Ready) aumenta el grado de multiprogramación.
 - Finalizar un proceso (Exit) disminuye el grado de multiprogramación.
- Mediano plazo: también controla el grado de multiprogramación pero, ahora, manejando los estados Ready/Suspended y Blocked/Suspended, que envían a un proceso de memoria principal al disco. Es decir, este planificador realiza operaciones de swapping. Recordar que cuando hablamos de pasar un proceso a disco nos referimos a pasar todas sus estructuras menos el PCB.
 - Swap in: cargar nuevamente en RAM a un proceso suspendido. Aumenta el grado de multiprogramación.
 - Swap out: pasar un proceso de RAM a disco. Baja el grado de multiprogramación.
 - La arista entre Ready/Suspended y Blocked/Suspended no es manejada por este planificador. Simplemente finaliza el evento que el proceso estaba esperando y pasa de blocked a ready pero sigue estando suspendido.
- Corto plazo: decide cuál es el próximo proceso que se debe ejecutar (Running). Ejecuta constantemente (cada vez que ocurre un evento que libera la CPU o que da la oportunidad de elegir un proceso con mayor prioridad). Debe minimizar el overhead. Su trabajo se centra en las aristas de dispatch y timeout, pasando por el bloqueo de procesos cuando solicitan algún servicio. Para manejar a los procesos bloqueados el SO maneja listas/colas de bloqueados:



Estos algoritmos se clasifican en dos:

- Con desalojo: considerar todos los eventos en los que un proceso llega a Ready.
- Sin desalojo: sólo consideran eventos que liberan CPU. Pueden monopolizar la CPU.

Algunas características:

- Ejecuta con la mayor frecuencia.
- Es esencial para cualquier SO multiprogramado.
- Es más rápido que el planificador de largo plazo.
- Ejecuta cuando se involucra el SO:
 - Interrupciones.
 - Syscalls.
 - Señales.
- Funciones:

- Controla el tráfico de procesos.
- Dispatcher: encargado de darle a la CPU el proceso elegido.
- Context switch: se encarga de la permutación de procesos.



CRITERIOS DE PLANIFICACIÓN

Definición: para realizar la planificación necesitamos basarnos en ciertas métricas que nos permiten medir los datos acerca del funcionamiento del sistema.

	Prestaciones (cuantitativas). Son criterios medibles y comparables	Otros criterios (cualitativos). Tienen que ver con la percepción del usuario
Orientados al proceso/usuario	<ul style="list-style-type: none"> • Tiempo de ejecución. • Tiempo de espera. • Tiempo de respuesta. 	<ul style="list-style-type: none"> • Previsibilidad.
Orientados al sistema operativo	<ul style="list-style-type: none"> • Tasa de procesamiento. • Utilización de CPU (%). 	<ul style="list-style-type: none"> • Equidad. • Imposición de prioridades. • Equilibrado de recursos.

Cuantitativos:

- Tiempo de ejecución: mide el tiempo desde que se solicita la creación de un proceso hasta que finaliza. Mientras más bajo sea, más rápido va a ejecutar.
- Tiempo de espera: suma de todos los intervalos de tiempo que el proceso estuvo esperando en Ready. Los planificadores tienen este indicador en cuenta al momento de asignar prioridades.

$$\text{Tiempo de espera: } \text{Tiempo final} - \text{Tiempo Llegada} - \text{Tiempo de CPU}$$

- Tiempo de respuesta: tiempo que transcurre desde que el proceso es iniciado hasta que da la primera respuesta (consideramos como respuesta una IO).

$$\text{Tiempo de respuesta: } \text{Tiempo final} - \text{Tiempo de llegada}$$

- Tasa de procesamiento: mide la cantidad de procesos que finalizan en un intervalo de tiempo determinado. Este indicador tiene más sentido analizarlo en servidores que en computadoras de usuario.
- Utilización de la CPU: mide el porcentaje de tiempo en el cual la CPU estuvo ocupada en un intervalo de tiempo. Mientras más alto el número, mejor. Siempre se trata de que la CPU esté ocupada.

Cualitativos:

- Previsibilidad: orientado al usuario. Hace referencia a qué comportamiento espera el usuario del sistema.
- Equidad/Imposición de prioridades/Equilibrado de recursos: tratar de asignar prioridades de manera pareja para que todos los procesos puedan ejecutar y se asignen los recursos de forma equitativa.

ALGORITMOS DE PLANIFICACIÓN

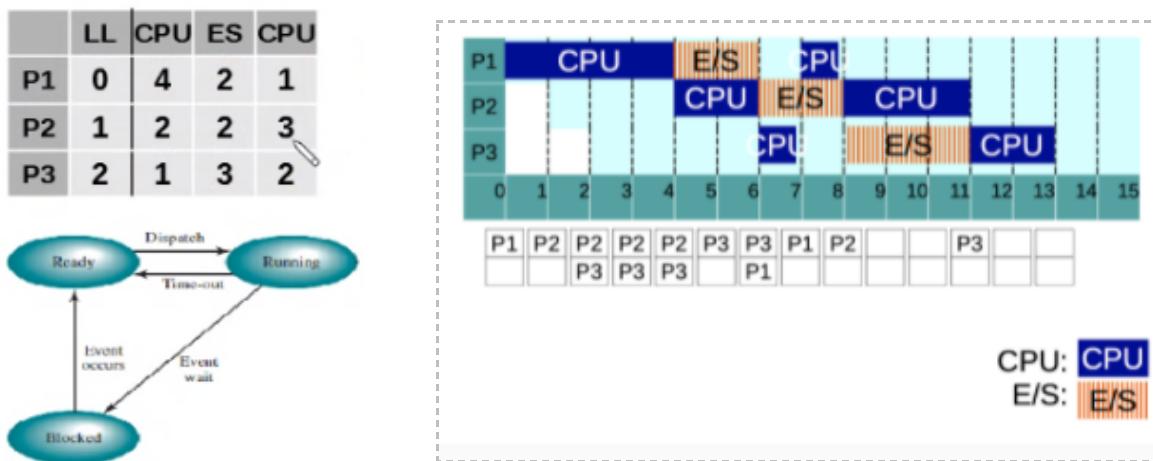
Definición:

- A cada proceso se le asigna una prioridad (se guarda en el PCB y va cambiando).
- La prioridad de un proceso puede variar en cada decisión.
- El planificador selecciona el proceso de prioridad más alta.

Consideraciones:

- Las IO no se planifican. Se atienden por FIFO.

FIFO (First In First Out):



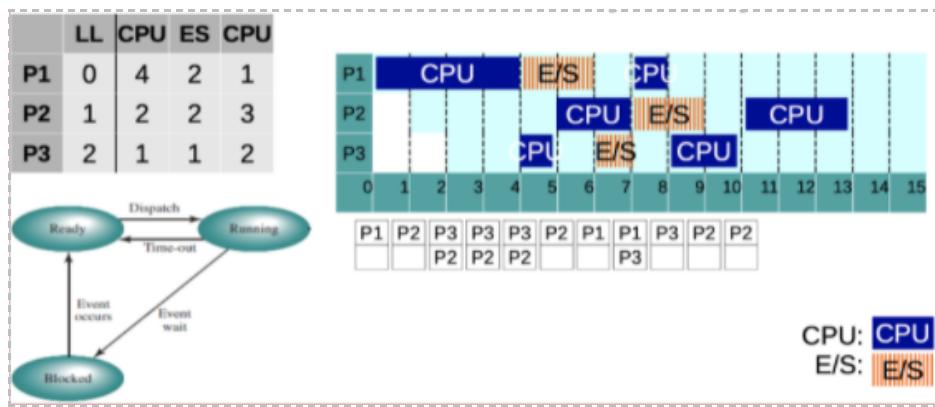
Nota: la flecha que dice "time out" no debería estar en este algoritmo.

Explicación del gráfico: arranca el proceso 1 ejecutando sus 4 unidades de CPU. Los cuadrados debajo del Gantt muestran los procesos que están en ready.

En el momento 4 el proceso 1 realiza una syscall. En este momento, la syscall bloquea al proceso y la CPU queda libre. ¿A quién le toca seguir ejecutando? Se fija en la lista ready y como ejecuta el primero que llega, ejecuta el proceso 2. En el instante 6 se bloquea y el proceso 1 se desbloquea. El proceso 1 que se desbloqueó, se posiciona al FINAL de ready. Por lo tanto, al que le toca ejecutar en el instante 6 es al proceso 3. Este proceso se bloquea, el proceso 2 también está bloqueado y el único que puede ejecutar es el proceso 1. Entonces ejecuta ese.

Problema que presenta este algoritmo: los procesos largos hacen que tarde mucho en ejecutarse los más cortos.

SJF (Shortest Job First) - sin desalojo: ejecuta el que tenga la ráfaga de CPU más corta.



Explicación del gráfico: arranca el proceso 1 con sus 4 instancias de tiempo. En ese instante ya están el proceso 2 y 3 en ready. Pero, como se ordenan según el que tenga el menor tiempo de ejecución de CPU, primero se ordena el proceso 3 y luego el 2 en la cola de prioridades. Por lo que en el instante 4, el proceso 3 ejecuta.

Convención: si hay dos procesos que quieren ejecutar y tienen la misma ráfaga, entonces seleccionamos por FIFO (es decir, va a ejecutar el que lleve más tiempo en Ready).

- **1º Problema (Starvation/Inanición):** ocurre cuando a un proceso se le niega la posibilidad de utilizar un recurso (en este caso la CPU) por la constante aparición de otros procesos con mayor prioridad.
- **2º Problema:** en la realidad es imposible saber cuánto tiempo va a usar la CPU un proceso, por lo que el algoritmo no se puede implementar en la realidad. Las ráfagas deben estimarse.

SJF con estimación de ráfaga: para la estimación de las ráfagas futuras usaremos la siguiente fórmula:

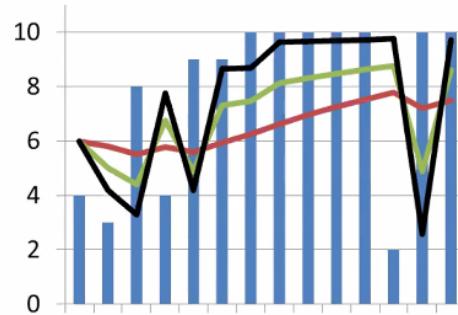
$$EST_{n+1} = \alpha * TE_n + (1 - \alpha) * EST_n$$

TE_n = Tiempo de ejecución de la ráfaga actual

EST_n = Tiempo estimado para la ráfaga actual

EST_{n+1} = Tiempo estimado para la próxima ráfaga

α = Constante entre 0 y 1

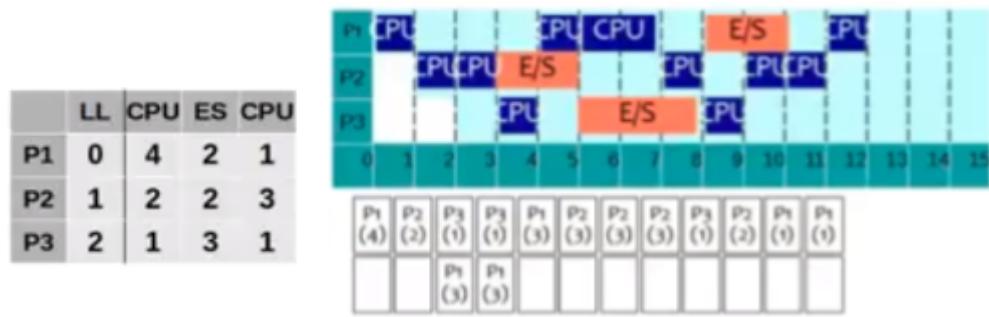


- Barras azules: ráfagas de CPU reales que ejecutó un proceso en ciertos momentos.
- Los demás colores son estimaciones a diferentes valores de alpha. Un alpha alto le da más peso al comportamiento más reciente. Alphas más pequeños le dan importancia al historial de estimaciones, mira toda la historia. Los cambios van a ser más lentos.
 - Rojo: alpha a 0.2
 - Verde: alpha a 0.5
 - Negro: alpha a 0.8

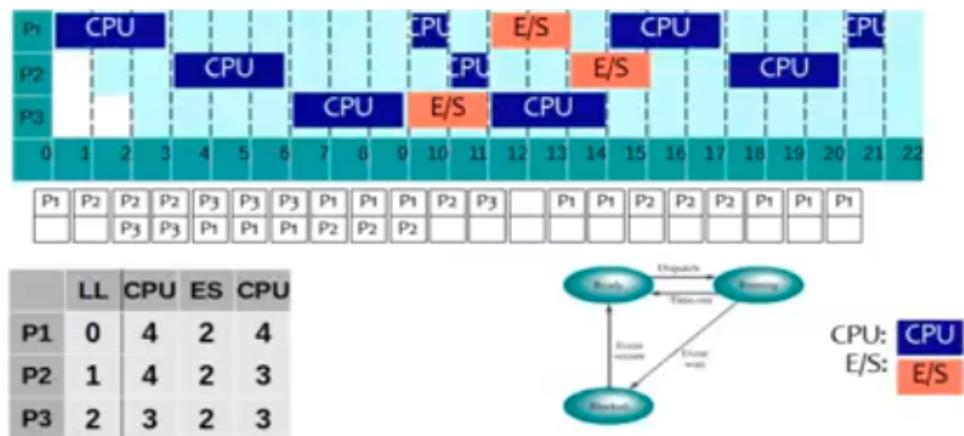
Entonces, ¿qué alpha me conviene? Depende de la estabilidad del proceso en cuestión. Si es estable conviene un alpha chico. Este algoritmo tiene mucho overhead. También genera inanición.

SJF con desalojo/SRT (Shortest Remaining Time): implica que el proceso que está siendo ejecutado puede ser expulsado de la CPU (es decir, pasar de Running a Ready). Cada vez que un nuevo proceso

llega a la cola Ready se pregunta si tiene mayor prioridad que el que está ejecutando. Puede tener inanición. Como usa estimadores también genera mucho overhead.



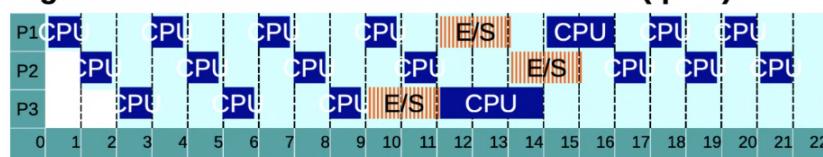
RR (Round Robin): la cola de Ready se maneja con FIFO pero el SO define un quantum (que dispara una interrupción de clock) para desalojar a un proceso cuando acaba su quantum.
Si un proceso se bloquea y no ha utilizado todo su quantum, este no se acumula.



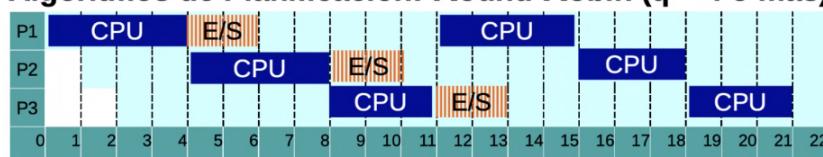
Quantum muy chico vs quantum mucho más grande: si bien parece que usando quantums distintos todos los procesos terminan de ejecutar en el mismo instante, el de quantum más chico tiene mucho más overhead, por lo que hay un montón de intervalos de tiempo en los que ejecuta el SO para administrar el cambio de proceso que no estamos graficando pero existen y hacen que termine siendo ineficiente.

En el de quantum más grande, es tan grande el quantum que no llega a cortar nunca al proceso, por lo que termina siendo un FIFO.

Algoritmos de Planificación: Round Robin (q = 1)



Algoritmos de Planificación: Round Robin (q = 4 ó más)

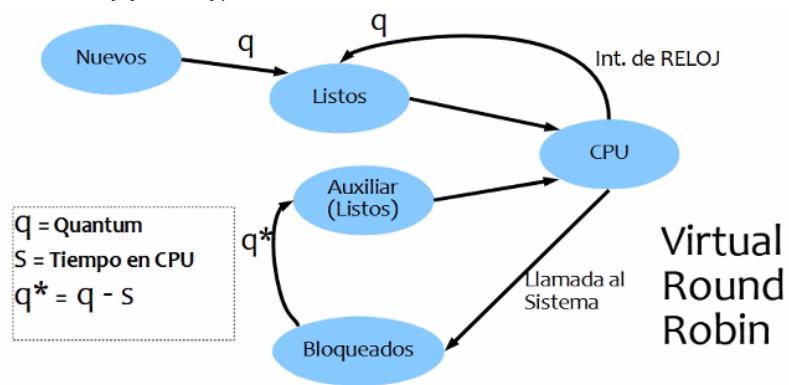


VRR (Virtual Round Robin): dado que el RR favorece a los procesos CPU Bound, el VRR surge para favorecer a los procesos IO Bound. Para esto, ahora hay dos colas, una con mayor prioridad que la otra. La idea es que, a la cola de mayor prioridad, vayan los procesos que terminen de hacer una E/S (que están en Blocked y, normalmente, tendrían que ir a la cola de Ready) para que no tengan que competir con los procesos CPU Bound.

El quantum se acumula en este algoritmo.

- Cola Aux Ready (+):
 - Vienen los procesos después de hacer IO.
 - Su quantum ahora es $q_{\text{Definido}} - \text{tiempoQueYaUsoCPU}$.
 - Cola de mayor prioridad.
- Cola Ready (-):
 - Vienen los procesos cuando se acabó su q . Salen con un q completo.
 - Obviamente esta cola no va a ejecutar hasta que la otra cola esté vacía.

En ambas colas (Aux Ready y Ready), se usa FIFO.



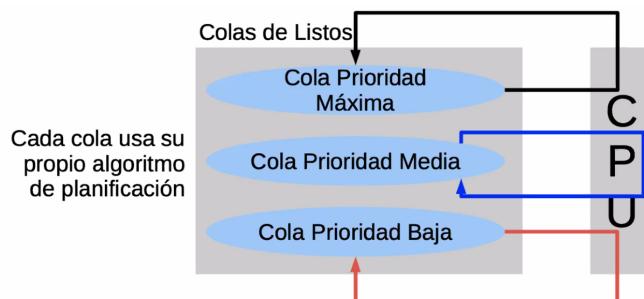
HRRN (Highest Response Ratio Next): contempla Aging, por lo que resuelve el problema de inanición del SJF. En este algoritmo va a ejecutar el que tenga mayor tasa de respuesta (R), que se calcula:

$$R = \frac{W + S}{S}$$

W: tiempo esperando en ready.
 S: lo que va a usar de CPU (se llama “tiempo de CPU esperado”).

Tener en cuenta que S (tiempo de CPU esperado), si bien lo sacamos de la tabla fijándonos cuánto dura la próxima ráfaga de CPU, en la realidad se estima con estimadores porque no se conoce.

Algoritmo de colas multinivel: se define una prioridad para cada proceso y cada prioridad tiene su propia cola. Puede haber muchas colas pero nosotros vamos a trabajar con 3: alta, media y baja. A su vez, cada cola puede usar su propio algoritmo. Es un algoritmo que sufre de starvation.



Algoritmo por prioridades: se define una única cola de Ready. A cada proceso se le define una prioridad y se ordenan todos los procesos en esta lista. Puede ser con desalojo o sin desalojo.

Multinivel retroalimentado: es parecido a las colas multinivel porque hay varias colas de Listos y cada cola puede tener su propio algoritmo. Cada cola tiene un quantum y, una vez que el proceso acaba el quantum va bajando de prioridad. Los procesos comienzan siempre en la cola de máxima prioridad. La prioridad es dinámica.

Puede usar mecanismos de aging (aumentarle la prioridad a un proceso que cayó en inanición) para evitar inanición. No está bien definido, puede tener distintas implementaciones y reglas.



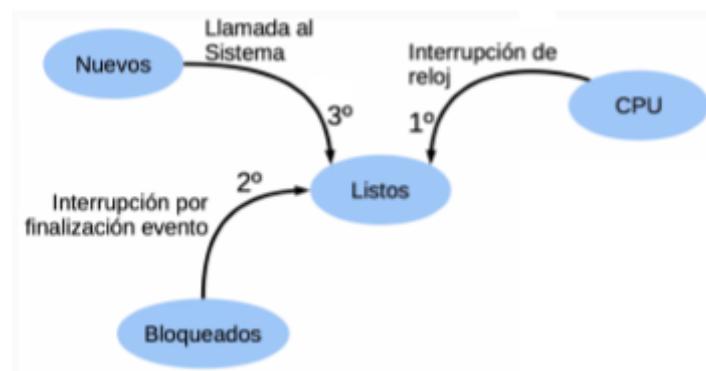
COSAS A TENER EN CUENTA

Simultaneidad de eventos en ready: supongamos que hay 3 procesos que quieren ir a Ready por diferentes razones:

- Porque es un nuevo proceso y realizó una syscall para entrar a Ready.
- Porque terminó de usar IO y realizó una interrupción para volver a Ready.
- Porque estaba en Running y acabó su quantum (entonces se disparó una interrupción de clock) y quiere volver a Ready.

¿Cómo establecemos el orden en la cola de Ready si llegan todos al mismo tiempo? Bueno, dado que las interrupciones se atienden antes que las syscalls, la syscall va a quedar al final. Después, entre la interrupción de clock y la interrupción por finalización de evento, la que tiene mayor prioridad es la interrupción de clock. Entonces, la pila quedará:

- Interrupción de clock.
- Interrupción por finalización de evento.
- Llamada al sistema por nuevo proceso.



¿En qué algoritmos me importa esto?

- En algoritmos FIFO y RR es importante cuál pongo primero porque la llegada define la prioridad.

- En SJF no me importa cuál pongo primero porque tengo que comparar las ráfagas.
- En VRR, como el proceso va a cola auxiliar después de finalizar un evento, esta simultaneidad de eventos en Ready no incluye a estos procesos.

Desalojo:

- Sin desalojo: el SO se involucra en dos casos
 - Syscall bloqueante.
 - Fin de proceso.
- Con desalojo: el SO se involucra en tres casos
 - Syscall bloqueante.
 - Fin de proceso.
 - Proceso con > prioridad llega a Ready.
 - New -> Ready
 - Blocked -> Ready
 - Timeout

RESUMEN DE ALGORITMOS

Nombre	Desalojo	Cómo determina prioridad	Q	Cant. colas	Aging/Inanición/ Favorece o perjudica a/Overhead/Monopolización de CPU
FIFO	No.	FIFO.	No.	1	<p><u>Starvation</u>: inicialmente no pero puede darse en caso de que se le asigne la CPU a un proceso que nunca termina de ejecutar y la monopoliza.</p> <p><u>Monopolización de CPU</u>: sí.</p>
SJF (Shortest Job First) SPN (Shortest Process Next)	No.	<p>Se comparan las ráfagas de tiempo. Se elige al que tenga la ráfaga más corta. En la realidad se usa la ecuación (1) para estimar las ráfagas. Sino, se usan las ráfagas dadas por la tabla.</p> <p><u>Convención</u>: si dos procesos con la misma ráfaga quieren ejecutar, se elige por FIFO (al que lleva más tiempo en Ready)</p>	No.	1	<u>Starvation</u> : hay inanición pues a un proceso se le negará el uso de CPU mientras aparezcan procesos con ráfagas más cortas.
SRT (Shortest Remaining Time) SJF con desalojo	Sí.	<p>Es igual que el SJF pero cada vez que llega un proceso a Ready se evalúa si tiene mayor prioridad al que está ejecutando.</p> <p><u>Convención</u>: si hay dos procesos con igual prioridad sigue el que ya estaba ejecutando (no se desaloja).</p>			
RR (Round Robin)	Por clock.	<p>Es FIFO pero usa las interrupciones de clock (quantum) para cortar procesos que usan mucho CPU. Si un proceso no utiliza todo su quantum no se acumula.</p> <p><u>Convención</u>: cuando dos procesos terminan al mismo tiempo y quieren CPU (uno viene de hacer IO y el otro se acabó su quantum), el que tiene mayor prioridad es el que venía usando CPU.</p>	Sí.	1	<p><u>Perjudica a</u>: los procesos IO Bound (porque no pueden aprovechar el tiempo de quantum).</p> <p><u>Favorece a</u>: los procesos CPU Bound porque pueden usar la CPU en forma equitativa.</p> <p><u>Starvation</u>: no. Al atender por FIFO todos los procesos serán atendidos en algún momento.</p> <p><u>Monopolización de CPU</u>: no porque hay quantum.</p> <p><u>Overhead</u>: medio.</p>
VRR (Virtual Round		Ambas colas (Aux Ready y Ready) usan FIFO. Si un proceso acaba todo su quantum y		2 colas. <u>Cola aux(+)</u> : - Vienen	<u>Favorece a</u> : mejora el rendimiento para procesos IO Bound ante los CPU Bound.

Robin)		justo va a hacer IO, cuando termine, va a ir a Ready.		después de hacer IO. <ul style="list-style-type: none"> - Su quantum ahora es q - tiempoQueYaHizoEnCPU. <u>Cola Ready(-):</u> <ul style="list-style-type: none"> - Vienen acá cuando se acabó su q. Salen con un q completo. 	Igual, no perjudica a los CPU Bound porque se supone que los IO Bound están bloqueados (usando IO) mucho tiempo y eso le da tiempo a los CPU Bound de ejecutar. <u>Starvation:</u> no. <u>Monopolización de CPU:</u> no porque hay quantum. <u>Overhead:</u> alto.
HRRN (Highest Response Ratio Next)	No.	A medida que un proceso espera, va aumentando su prioridad para poder ser elegido por el procesador. Para calcular la ráfaga se usa la ecuación (2). La prioridad es dinámica porque va cambiando a medida que pasa el tiempo.	No.	1.	<u>Favorece a:</u> los que tienen ráfaga corta y mucho tiempo esperando (aging). <u>Starvation:</u> no porque la prioridad de los procesos aumenta a medida que esperan. <u>Monopolización de CPU:</u> posible <u>Overhead:</u> alto.
Colas multinivel	Depende	Se define una prioridad para cada proceso y cada prioridad tiene su propia cola. Internamente, cada cola puede usar un algoritmo distinto. La prioridad de los procesos es estática. Siempre vuelven a su misma cola a menos que manualmente se cambie su prioridad.	No.	Depende. Nosotros usamos 3 (Alta, Media, Baja).	<u>Starvation:</u> sí.
Algoritmo por prioridades	Depende	Se define una única cola de Ready. A cada proceso se le define una prioridad y se ordenan todos los procesos en esta lista.	No.	1.	
Colas Feedback Multinivel		Eventos que pueden generar una replanificación: <ul style="list-style-type: none"> - Interrupción de quantum. - Llegada de un nuevo proceso. - Interrupción de fin de IO. - Bloqueo de un proceso. 			

PREGUNTAS DE PARCIAL

- 1. Los algoritmos de planificación con desalojo ¿Qué eventos tienen en cuenta para la re-planificación? ¿Por qué los algoritmos sin desalojo no?**

Los algoritmos con desalojo tienen en cuenta la llegada de procesos a Ready dado que si llega un proceso con mayor prioridad que el que esté ejecutando este deberá ser desalojado.

Los algoritmos sin desalojo no tienen en cuenta este evento dado que una vez que asignan la cpu a un proceso no lo desalojarán porque haya llegado otro proceso de mayor prioridad.

- 2. Compare los algoritmos HRRN, SJF con desalojo y sin desalojo en términos de criterio de selección, posible desalojo, penalización procesos cortos/largos, Overhead, y Starvation.**

	Criterio de selección	Desalojo	Overhead	Starvation	Penalización
HRRN	A medida que un proceso espera, va aumentando su prioridad para poder ser elegido por el procesador. Prioriza a los que tienen ráfaga corta y mucho tiempo esperando.	No	Alto.	No.	Prioriza procesos con ráfagas cortas de CPU.
SJF (desalojo) /SRT	Elige al proceso que le quede la ráfaga más corta para ejecutar.	Sí	Medio - alto porque usa estimadores.	Sí.	Prioriza procesos con ráfagas cortas de CPU y los que tienen ráfaga larga posiblemente entrarán en inanición.
SJF (sin desalojo)	Elige al proceso que tenga la ráfaga más corta para ejecutar.	No	Entre los SJF este tiene menor overhead porque no tiene desalojo		

- 3. Explique cómo el planificador de corto plazo podría llegar a generar una condición de carrera. Proponga una forma de solucionarla, sin soporte del SO, que pueda ser aplicado en entornos con múltiples procesadores.**

Puede ocurrir que estemos modificando una variable compartida entre dos procesos y que en medio de la operación (que a nivel instrucción se descompone en varias instrucciones) por causa de una interrupción el planificador de corto plazo decida ejecutar otro proceso. Esto da como resultado la condición de carrera. Una forma de solucionarlo es sincronizando la región crítica con instrucciones atómicas como test_and_set, que pueden ser utilizadas en entornos multiprocesador.

- 4. ¿Es consciente en algún momento el proceso del hecho de quedar bloqueado o continuar su ejecución? En caso afirmativo explique cómo, y en caso negativo indique por qué.**

No, el proceso no tiene noción de si es bloqueado o no dado que cuando se lo bloquea se guarda su contexto de ejecución y cuando vuelve a ejecutar se restaura.

- 5. Describa cómo afecta en el comportamiento de un sistema el tamaño del quantum. ¿Afecta de la misma manera en RR que en VRR?**

Un quantum menor implica mayor overhead dado que el planificador de corto plazo debe intervenir para seleccionar un proceso para ejecutar frecuentemente.

En RR un quantum grande hace que el algoritmo se convierta en una especie de FIFO mientras que un quantum más pequeño implica un alto grado de overhead pero los procesos ejecutarían de forma más equitativa. En VRR ocurre algo similar con el agregado de una cola auxiliar que puede llegar a crecer en gran tamaño.

6. Compare los algoritmos FIFO, Round Robin y SJF en términos de Equidad, Overhead y Starvation.

	Equidad	Overhead	Starvation
FIFO	No es equitativo. Hay starvation y monopolización de la CPU (si un proceso tiene una ráfaga de CPU muy grande)	Bajo.	Sí.
RR	Es equitativo porque todos los procesos ejecutan el mismo quantum de tiempo. Favorece a los procesos CPU bound.	Medio.	No.
SJF	No es equitativo.	Alto porque debe comparar las ráfagas de todos los procesos.	Sí.

7. Mencione todos los pasos que ocurren cuando al estar ejecutando un proceso se produce una interrupción de I/O finalizada (correspondiente a otro proceso que se encontraba bloqueado).

- Se produce la interrupción
- Al finalizar la instrucción actual se chequea si hay interrupciones pendientes
- Se guarda el contexto de ejecución del proceso actual
- Se produce un cambio de modo usuario a modo kernel
- Se pasa el control al Interrupt Handler
- El SO pasa el proceso bloqueado a la lista de ready
- Si el planificador de corto plazo tiene desalojo se evalúa si la prioridad de este proceso que acaba de llegar a ready es mayor a la del proceso que estaba ejecutando. Si lo es, el proceso que estaba ejecutando deberá volver a ready
- Se cambia el modo de ejecución de kernel a usuario
- Se restaura el contexto de ejecución del proceso que deba ejecutar

8. V o F. En el caso de utilizar jacketing en la biblioteca de ULTs, es lo mismo usar hilos a nivel de usuario que a nivel de kernel.

Falso. Si bien el jacketing permitiría que no se bloquee al proceso completamente los hilos de usuario seguirían siendo invisibles al SO y seguirían estando sujetos a una planificación distinta a la del SO.

9. ¿A qué se refiere el problema de starvation en planificación de procesos? Mencione dos algoritmos con desalojo que puedan sufrir del mismo e indique en cada caso cómo se podría solucionar.

Starvation hace referencia a cuando a un proceso se le niega el uso de la cpu (es decir que no puede ejecutar) porque constantemente llegan a ready otros procesos con mayor prioridad que él.

- SRT (SJF con desalojo): este algoritmo sufre de inanición dado que siempre tendrán mayor prioridad aquellos procesos que tengan las ráfagas de CPU más cortas, por lo que los

procesos que tengan ráfagas largas siempre quedarán al final de la lista de prioridades. Para solucionar este problema se podría ir aumentando la prioridad del proceso a medida que espera en ready (convirtiéndose en el algoritmo HRRN).

- Algoritmo por prioridades: en este algoritmo puede producirse starvation porque los procesos tienen definida una prioridad fija y no serán elegidos por el planificador mientras sigan apareciendo otros procesos con mayor prioridad. Una forma de resolverlo es hacer que las prioridades sean dinámicas, por ejemplo, si un proceso ha pasado más de x tiempo esperando en ready pueda aumentar su prioridad.

10. En caso de utilizar un algoritmo de planificación con desalojo, al llegar una interrupción de fin de IO de un KLT la misma será atendida sólo si dicho proceso tiene mayor prioridad (dependiendo el algoritmo que utilice) que el thread en ejecución y amerita el cambio.

Falso. La interrupción de fin de IO siempre será atendida al finalizar el ciclo de instrucción sin importar la prioridad del hilo/proceso que la genere.

11. Una transición entre el estado Running y el estado Ready sólo es posible en algoritmos con quantum.

Falso. También es posible en algoritmos con desalojo como el SRT, que además no tiene quantum.

12. En un sistema que sufre el fenómeno de inversión de prioridades, es posible solucionar este problema cambiando su algoritmo de planificación por Virtual Round Robin.

Inversión de prioridades: se da cuando un proceso con menor prioridad retiene un recurso que necesita uno de mayor prioridad. El de mayor prioridad no puede continuar y el otro no ejecuta por su prioridad baja.

Verdadero. En un VRR, en algún momento van a ejecutar todos un quantum, solucionando este problema.

13. Explique las implicancias de utilizar un planificador de corto plazo sin desalojo en sistemas operativos de tiempo compartido (multitarea).

Que no haya desalojo implica que un proceso que fue elegido para ejecutar solamente va a liberar la CPU cuando finalice o se bloquee, por lo que un proceso podría nunca ejecutar si otro proceso nunca libera el procesador, ya sea por un error o porque no se bloquea nunca.

14. ¿Qué diferencias existen entre los algoritmos de planificación de procesos con y sin desalojo? Indique brevemente en qué sistemas sugeriría utilizar cada uno.

Los algoritmos con desalojo deben evaluar la prioridad de cada proceso cuando llega a ready y si esta es mayor a la del proceso que esté ejecutando en ese momento tendrá que desalojarlo de la CPU para darle lugar a que ejecute el proceso con la mayor prioridad.

Es importante tener en cuenta que los algoritmos con desalojo generan más overhead que los que no tienen desalojo por lo que los algoritmos sin desalojo son recomendables para aquellos sistemas que quieran minimizar el overhead.

Los algoritmos con desalojo son recomendables para sistemas multitarea que quieren que sus procesos ejecuten con un poco más de equidad o que se prioricen a aquellos procesos más importantes (que serán en definitiva los que tengan mayor prioridad).

15. Todos los planificadores (largo, mediano, corto) modifican según sus decisiones el nivel de multiprogramación.

Falso. Solamente los planificadores de largo y mediano plazo lo modifican. El planificador de corto plazo solamente trata con procesos que están en RAM.

16. Explique cómo funciona el envejecimiento (aging) y para qué se utiliza. Ejemplifique con un algoritmo.

Aging hace referencia a aumentar la prioridad de un proceso a medida que espera a ser elegido. Se utiliza en la planificación de procesos para evitar que un proceso sea negado el uso de la CPU por la constante aparición de otros procesos con mayor prioridad que él (starvation).

Un ejemplo de un algoritmo que contempla aging es el algoritmo HRRN, el cual tiene en cuenta el tiempo que el proceso estuvo esperando en Ready.

Hilos

INTRODUCCIÓN

Imagen de un proceso: habíamos dicho que el código es una porción de código de sólo lectura, los datos contienen a las variables globales y el heap es la memoria dinámica. El stack y el PCB tienen algo en común y es que se modifican constantemente y están muy relacionados con la ejecución del proceso (esto es importante remarcarlo porque en los procesos multihilos cada hilo tendrá su propia pila y su propio TCB). Hasta ahora sólo conocíamos procesos con un único camino de ejecución (en forma secuencial), que podía ser un sistema que no admitía hilos o un proceso multihilo con un sólo hilo.



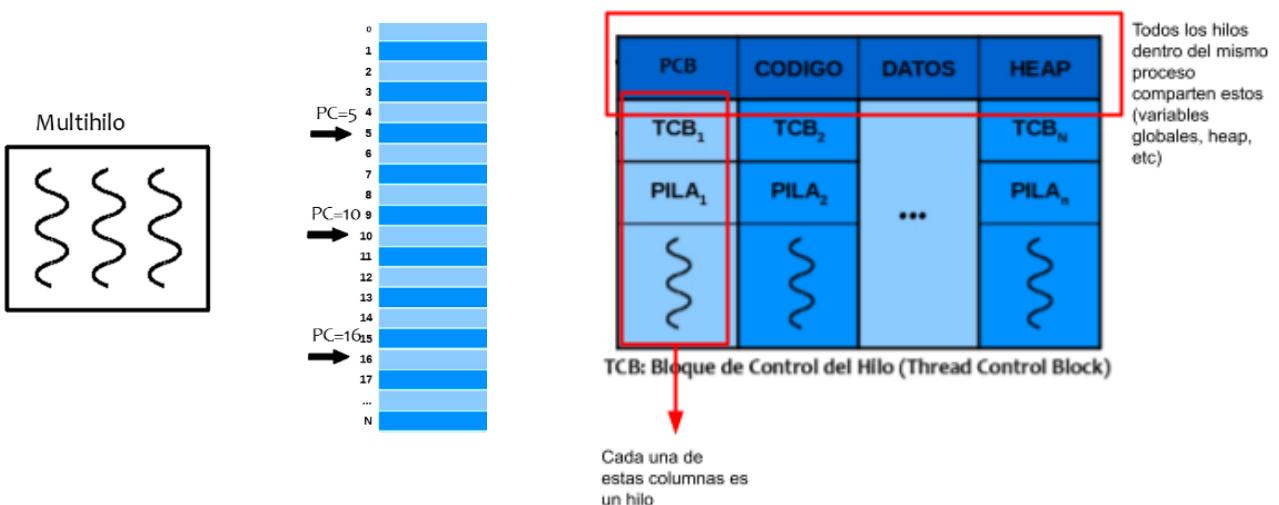
Hilo: es una línea de ejecución de un proceso. Los hilos son la unidad de trabajo del SO, dado que el SO se encarga de planificar a los hilos (de los procesos) para que todos puedan ejecutar (antes decíamos que el SO planificaba "procesos", ahora decimos que planifica "hilos"). El SO no es el único que puede planificar a los hilos, sino que existen bibliotecas que los procesos pueden implementar que se encargan de la planificación interna.

Un hilo comparte el código, datos y recursos con los otros hilos de su mismo proceso, pero cada hilo tiene su propio stack y su propio TCB donde guarda su contexto de ejecución: su propio PC, su estado de ejecución, su contexto de ejecución, etc.

MULTIHILOS

Definición: un programa multihilo contiene dos o más partes que se pueden ejecutar de manera concurrente. Es decir, agrega más trazas de ejecución.

Varios hilos = poder hacer varias cosas al mismo tiempo dentro de un mismo proceso



Estructuras involucradas:

- PCB: se sigue manteniendo como la estructura necesaria para administrar la ejecución de todo el proceso, pero ahora con una diferencia. Como ahora tenemos varios caminos de ejecución, es necesario que cada hilo posea su propio stack. ¿Por qué? porque si yo quisiera tener tres caminos concurrentes de ejecución, tal vez el camino 1 esté llamando a la función A y el camino 2 llamando a la función B. Entonces, el mismo stack no me sirve para ir encadenando lo que hace cada hilo.
- TCB (Thread Control Block): se contará con un TCB por cada hilo, el cual guardará cierto tipo de información administrativa propia de cada hilo de ejecución.
 - Prioridad
 - ID de hilo (TID)
 - Estado
 - Contexto de ejecución del hilo
 - PC
 - Flags

Nota respecto al cambio de hilos: dado que cada TCB tiene su propio contexto de ejecución, cuando hay un cambio de hilo hay un cambio de contexto.

Nota con respecto a la memoria: las variables locales de los hilos no son compartidas (porque están en el stack), las únicas que se comparten son las globales.

Ventajas de los hilos:

- Capacidad de respuesta (cada hilo sigue su propia traza de ejecución y no debe esperar a que finalicen los otros).
- Menos overhead para el SO (es preferible tener muchos hilos en un sólo proceso que tener muchos procesos separados ya que el thread switch es más rápido porque los hilos comparten memoria y al cambiar de hilo (debe ser ULT) no es necesario realizar un cambio de modo).
- Compartición de recursos entre hilos.
- Comunicación eficiente (a través de memoria compartida y semáforos). Si un hilo deja algo en la memoria compartida, los demás hilos lo pueden leer fácilmente.
- Permite multiprocesamiento (en el caso de los KLTs): puede ejecutarse cada hilo en un procesador distinto.
- Procesamiento asincrónico: no sigue un orden secuencial.

Desventajas de los hilos: dependen del contexto en que se los use o las funcionalidades del proceso. Por ejemplo, (tomando en cuenta la seguridad) al compartir recursos, a veces no es deseable que un hilo pueda consultar la memoria de otro hilo (sobre todo cuando se trata de datos sensibles).

HILOS DE KERNEL (KERNEL LEVEL THREADS)

Definición: la administración del hilo está hecha por una biblioteca que provee el SO. Entonces, el SO conoce a los hilos, los crea, los destruye y los administra.

Planificación: los hilos van a estar sujetos a la misma planificación que tiene el SO. El planificador de corto plazo del SO deja de planificar los procesos como entidades absolutas y empieza a planificar los KLT de cada proceso. Es decir, ahora, en vez de elegir qué proceso ejecuta, el planificador elige por hilo (sin importar a qué proceso pertenece).

Ventajas:

- Las syscalls bloqueantes sólo bloquean al hilo en cuestión.
- Multiprocesamiento de hilos del mismo proceso: los KLT pueden ejecutar cada uno en un procesador distinto.
- Menor overhead que si se usaran distintos procesos.

Desventajas:

- Mayor overhead que en los ULT (cualquier tarea de administración implica un mode switch, la creación de un hilo implica una syscall, etc.).
- Son menos estables y seguros que los procesos. Dado que los KLTs de un mismo proceso comparten heap, si un hilo fuera a generar memory leaks esta memoria alocada no se liberaría hasta la finalización del proceso en su totalidad.

HILOS DE USUARIO (USER LEVEL THREADS)

Definición: la administración del hilo está hecha por una biblioteca de usuario que el proceso incorpora y ejecuta. Es decir, esta biblioteca se encarga de crear a los hilos y de administrarlos sin recurrir a la ayuda del SO. Esto genera que la administración esté hecha dentro del espacio de usuario y que no haya involucración alguna del SO (quien ni siquiera se entera que el proceso por dentro tiene hilos).

Planificación: se puede decir que existe una doble planificación porque el SO planifica al proceso completo y la biblioteca planifica a los hilos.

Ventajas:

- Bajo overhead: no se necesita la intervención del SO. No hay syscalls ni cambios de modo.
- Planificación personalizada: internamente puedo tener un planificador (que no tiene porqué ser el mismo que el SO) para planificar la ejecución de los hilos.
- Portabilidad: como la planificación es a través de una biblioteca, puedo utilizar una biblioteca basada en funciones que pertenezcan a bibliotecas estándares, lo que hará que el programa sea portable.
- Su creación es más liviana y los cambios de hilo también.

Desventajas:

- No permite a los hilos ejecutar en procesadores distintos: dado que es el SO quien asigna el procesador y dado que los hilos no son conocidos por el SO, no hay manera que el SO pueda mandar a los hilos a diferentes procesadores.
- Si ocurriera una syscall bloqueante, todo el proceso se bloquearía, bloqueando todos los hilos. Esta problemática puede resolverse con una técnica llamada jacketing.

E/S bloqueante y ULT: la problemática es que las llamadas al sistema son bloqueantes. Supongamos que mientras estaba ejecutando un ULT llama a fwrite, que es un wrapper de WRITE. Esto sería una syscall bloqueante. Se bloquea todo el proceso, no sólo el ULT, lo que impide que los otros ULT sigan ejecutando. Para resolver este problema y que no se bloqueen todos los ULT, sino solamente el ULT que realizó la syscall, vamos a usar una técnica llamada jacketing (explicada abajo).

Pero acá tenemos otro problema a resolver: cada proceso tiene su PCB. Cuando un proceso se bloquea, en el PCB se guarda el contexto de ejecución y registros del procesador. Entre esos registros también está el PC. Este apunta a la siguiente instrucción a ejecutar. Cuando el proceso deje de estar bloqueado y de bloqueado pase a listo, ¿qué hilo va a ejecutar? El mismo que venía ejecutando porque ahí se quedó el PC cuando se produjo la interrupción. Entonces se pierde un poco esa

planificación interna de los hilos (porque no da lugar a que se realice, es decir, comienza a ejecutar el hilo directamente).

Para solucionar este problema la solución es ejecutar el WRAPPER de una biblioteca (fwrite_ULT podría ser), que antes de bloquearme me permite cambiarme de hilo. Es decir, si mi PC apuntaba a una instrucción de un hilo, antes de llamar a fwrite hace que PC apunte a otro lado (la planificación me dirá a dónde). O sea, la interrupción y bloqueo se siguen produciendo pero me permite mantener mi planificación interna de hilos. Entonces, cuando el enunciado me diga que se maneja una biblioteca de hilos de usuario con una planificación determinada, significa que se respeta esa planificación, porque la biblioteca usa sus propios wrappers.

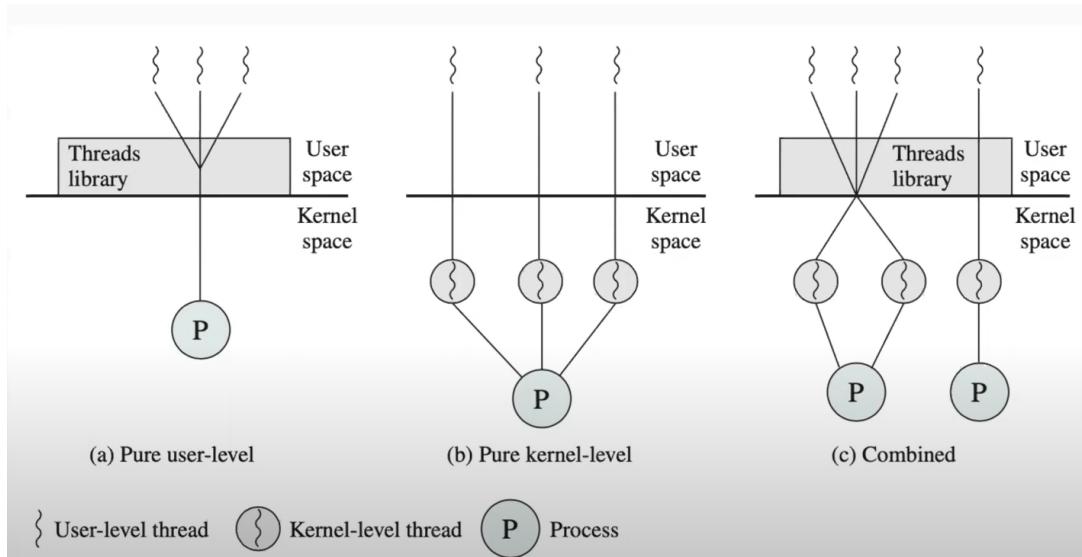
Jacketing/Revestimiento: jacketing es cuando en lugar de hacer la syscall bloqueante, se efectúa en modo no bloqueante. Hay que tener en cuenta que la mayoría de las operaciones bloqueantes tienen su contraparte no bloqueante. Entonces, en lugar de decirle al SO "escribíme esto en disco y bloqueame hasta que termines" le dice "escribíme esto en disco y yo sigo con la mía". Dentro de un rato te pregunta "¿Ya terminaste la escritura?", y el SO contesta.

Entonces, básicamente, la biblioteca hace esto: cuando el programa le pide realizar una IO bloqueante, la biblioteca se la pide al SO como no bloqueante, y la biblioteca "bloquea internamente" (o sea, no le da turno de ejecución) al hilo que pidió hacer la IO. Y, cada tanto, le va a preguntar al SO si esa operación terminó o no. Si terminó, lo "desbloquea".

Además podemos seguir con nuestra planificación interna. En los ejercicios asumimos que NO hay jacketing a menos que diga lo contrario.

COMBINACIÓN DE HILOS

¿Se pueden combinar KLTs y ULTs en un mismo proceso?: si. El propósito de hacer algo así es intentar obtener los beneficios de ambas estrategias.



PREGUNTAS DE PARCIAL

- 1. Utilizando semáforos en hilos ULT, no requieren realizar un cambio de modo para ejecutar las operaciones de wait/signal. Indicar verdadero o falso.**

Falso. No importa que sean hilos ULT. Wait y signal son syscalls por lo que implican un cambio a modo kernel.

- 2. Para compartir memoria entre procesos o entre KLTs se necesita intervención del SO. Entre ULTs no es necesario, debido a que se gestionan en espacio de usuario. Indicar verdadero o falso.**

Falso. Tanto los KLTs como los ULTs comparten memoria (variables globales) sin intervención del SO.

- 3. V o F. La utilización de KLTs en lugar de procesos, a pesar de ser más rápido su switcheo, puede generar problemas de memory leaks.**

Verdadero. Si bien los KLTs son más rápidos también son menos seguros y estables. Por ejemplo, los KLTs de un mismo proceso comparten recursos, por lo tanto, recién cuando finalice el proceso se liberarán los mismos. Entonces, si un usuario no libera correctamente la memoria que pide dinámicamente en sus KLTs, podrán existir problemas de secciones alocadas no referenciadas hasta que el proceso finalice y se liberen.

- 4. Explique ventajas y desventajas del uso de ULTs en lugar de KLTs.**

Ventajas:

- Velocidad en operaciones de hilos (thread create, thread switch).
- El proceso puede tener su propio planificador para la ejecución de sus hilos.
- Portabilidad.
- Bajo overhead. No se necesita la intervención del SO (no hay syscalls ni cambios de modo).

Desventajas:

- No permite a los hilos ejecutar en procesadores distintos.
- Si no se utiliza jacketing una syscall bloqueante bloquearía al proceso en su totalidad.

- 5. V o F. Los KLTs no pueden ocasionar memory leaks, dado que el TCB no tiene referencia al heap del proceso.**

Falso. Si bien no tienen una referencia directa, los KLTs comparten el heap. Por ende, si pidieran memoria y nunca la devolvieran, generarían memory leaks.

- 6. V o F. Los KLT de un mismo proceso pueden competir por el uso del procesador. En cambio, no es posible que los ULT de un mismo proceso compitan por el procesador.**

Falso. Si bien los KLTs pueden competir por ser elegidos por el planificador de corto plazo del SO, los ULTs compiten por ser elegidos por el planificador de hilos que implemente la biblioteca de hilos del proceso.

- 7. Explique las operaciones asociadas con el cambio de ejecución de un KLT a otro del mismo proceso.**

- Se guarda una parte inicial del contexto (PC, FLAGS) en el stack del sistema.
- El SO comienza su ejecución y guarda el resto de los registros en el stack del sistema.
- El SO determina que debe realizar un thread switch.
- Mueve el contexto de ejecución completo del stack del sistema al TCB del hilo.
- Elige el próximo hilo a ejecutar.
- Copia todo el contexto del TCB del nuevo hilo al procesador y realiza un cambio de modo (a usuario).

- Aclaración: no se actualizan registros relacionados con punteros a segmentos de memoria estáticos, heap o código, dado que pertenecen al proceso y son compartidos por todos los hilos

8. Compare utilizar Procesos vs KLTs vs ULTs respecto a overhead, multiprocesamiento y protección.

	Overhead	Multiprocesamiento	Protección
ULTs	Bajo. El SO no conoce a los hilos.	No es posible dado que el SO sólo conoce al proceso y no a sus hilos.	Los hilos son manejados por una biblioteca de usuario.
KLTs	Alto. Todas las operaciones con hilos (thread switch, thread create, etc) involucran al SO.	Es posible que los KLTs ejecuten en procesadores distintos.	Los hilos son manejados por el SO.

9. ¿En qué situaciones convendría usar hilos de usuario en lugar de hilos de kernel? Mencione al menos dos atributos propios del TCB (Thread Control Block)

Conviene usar hilos de usuario en varias situaciones:

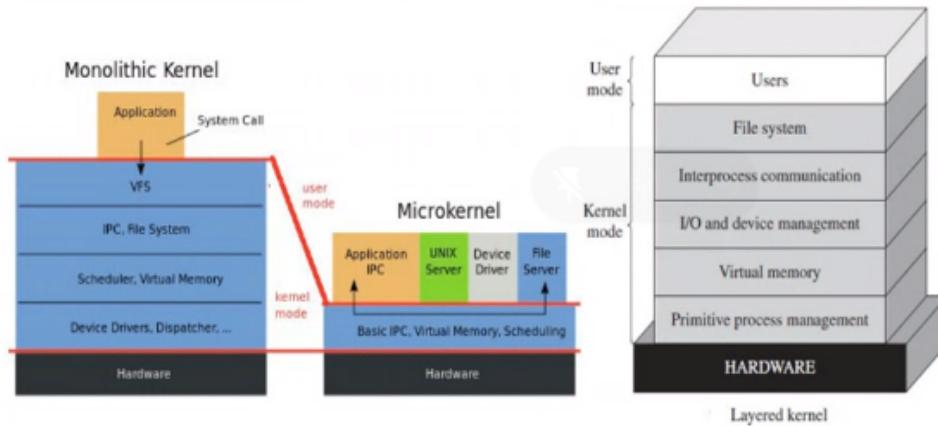
- Cuando el proceso quiere planificar a sus hilos con su propio planificador.
- Cuando se busca que el overhead sea bajo.
- Para que el proceso sea portable.
- Para que las operaciones con hilos (create, switch, etc) sean más rápidas.

Atributos del TCB:

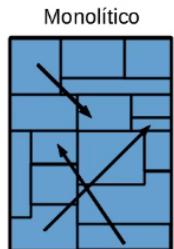
- TID.
- Estado del hilo.

Arquitectura de Kernel

TIPOS DE KERNEL



Kernel monolítico: hay un único módulo que hace todo. Todas las funcionalidades del SO están en un archivo (o varios, sin orden entre sí) y cada “módulo” puede tener acceso a los demás. Va creciendo a medida que se agregan nuevas funcionalidades.



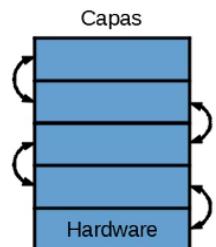
Problemáticas:

- Es un sistema muy difícil de mantener.
- El trackeo de errores es muy complejo.
- Al estar todo tan acoplado, un pequeño cambio puede afectar a todo el sistema.

Ventajas:

- Gran fluidez de la información y por lo tanto, mayor eficiencia.
- No hay ningún tipo de overhead.

Arquitectura de tipo capas (kernel multicapas): es una estrategia interesante aunque no muy aplicada donde se separan las funcionalidades del kernel en módulos y cada módulo constituye una capa. Cada capa, entonces, tiene una estructura bien definida y se comunica con las demás mediante interfaces.



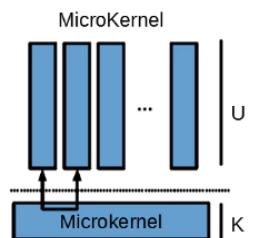
Problemáticas:

- Menor fluidez de la información. Si estoy en una capa y necesito comunicarme con otra, quizás tengo que pasar por muchas otras capas.

Ventajas:

- Es más simple de mantener.
- Los cambios no afectan a todo el sistema.

Microkernel: separa las funcionalidades que se ejecutan en modo kernel de las que se ejecutan en modo usuario. La idea es tener un módulo que corra en modo kernel bien chiquito (el microkernel), con un conjunto de funcionalidades mínimas, no todas las funciones del kernel, sino las más esenciales. El resto de las funcionalidades, que típicamente serían parte del SO, tenerlas pero corriendo como procesos en modo usuario. Igualmente, hay algunas cosas que no van a poder manejarse en modo usuario (todo lo relacionado a interrupciones, dispositivos de I/O, etc).



Ventajas:

- Provee más flexibilidad a la hora de poner o sacar módulos. Si se produce un problema en un componente no afectaría a los demás componentes.

Desventajas:

- Esta estrategia tiene un poco más de overhead, porque los módulos para comunicarse entre sí deben hacerlo a través del kernel. No pueden hablar directamente porque los procesos son entidades independientes. Por lo que podemos decir que tiene problemas de performance.

PREGUNTAS DE PARCIAL

1. V o F. Una de las mayores desventajas de los microkernels es su problema de performance.

Verdadero. La comunicación entre distintos módulos del SO ahora tiene que realizarse a través de paso de mensajes pasando por el kernel, lo cual provoca un overhead extra.

Concurrencia y sincronización

INTRODUCCIÓN

Distintas formas de concurrencia:

- Multiprogramación: implica múltiples procesos activos en memoria en un momento dado. Los procesos se van turnando en el uso de recursos. Si bien hay varios procesos en memoria, sólo uno puede ser ejecutado a la vez en una CPU.
- Multiprocesamiento: manejo de múltiples procesos en un sistema multiprocesador. Muchos procesos ejecutando al mismo tiempo (porque la computadora tiene muchos procesadores).
- Procesamiento distribuído: manejo de múltiples procesos en múltiples computadoras distribuídas para ejecutar.
- Compartición de recursos y competición por los recursos: está más adelante.

¿Por qué se usa la concurrencia?

- Mejora la performance. Debido a la multiprogramación, si un proceso se encuentra bloqueado, para no dejar a la CPU ociosa, necesito ejecutar otro proceso en ese instante.
- Aplicaciones estructuradas: por cuestiones de diseño, un programa puede ser definido como un conjunto de procesos/hilos concurrentes.

CONDICIÓN DE CARRERA

Definición: situación donde varios actores (hilos o procesos) modifican datos compartidos y se obtienen diferentes resultados finales dependiendo del orden en el que se ejecuten los procesos o hilos (se dice que depende de la “velocidad” de ejecución de ellos, de ahí el nombre).

Para garantizar la coherencia debemos asegurar que solo uno de los procesos pueda acceder a la manipulación de datos a la vez. Para esto hay que *sincronizarlos*. Solamente vamos a sincronizar dos procesos cuando acceden a los mismos datos (ya sea que los dos quieren escribir o uno de ellos quiere escribir y el otro leer). Pero, si los dos acceden en modo lectura, no hace falta sincronizarlos.

La sección donde puede ocurrir la condición de carrera se conoce como sección crítica.

FORMAS DE INTERACCIÓN ENTRE PROCESOS

- Comunicación entre procesos.
- Competencia de los procesos por los recursos.
- Cooperación de los procesos vía compartición.
- Cooperación de los procesos vía comunicación.

Si hay varios procesos, los mismos van a competir por el uso de memoria y demás recursos. En la competencia, el SO se va a encargar de decidir qué proceso recibe qué recurso y por cuánto tiempo. En la cooperación lo harán los hilos y procesos.

	Acoplamiento	Administración de recursos
Competencia	Desconocimiento de competidores.	SO.
Cooperación indirecta	Conocimiento de existencia.	Proceso/hilo.
Cooperación directa	Conocimiento de cuales son.	Proceso/hilo.

REQUISITOS QUE DEBEN CUMPLIRSE DENTRO DE LA SECCIÓN CRÍTICA

Mutua exclusión: sólo un proceso puede estar en la sección crítica usando un recurso. No puede haber ningún otro proceso o hilo en la misma sección para usar ese mismo recurso.

Progreso: cuando un proceso termina de usar el recurso de la sección crítica debe dar aviso así los demás hilos o procesos pueden acceder a él.

Espera limitada: la espera para entrar a la región crítica debe ser limitada. Se relaciona con el progreso. El proceso que está ejecutando en la región crítica debe avisar cuando termine de usarla porque sino los demás procesos que están esperando nunca se enterarían que está libre.

Velocidad relativa: qué tan rápido va a ejecutar un proceso sus instrucciones. Nunca puede ser predicho porque pueden ocurrir interrupciones en el medio. Por más mínima que sea la operación no podemos decir con certeza cuánto va a durar.

Consideraciones:

- La sección crítica debe ser lo más pequeña posible. Si la sección crítica fuera muy grande (al punto en que un proceso debería esperar a que finalice otro para poder ejecutar) entonces no se podrían tener procesos concurrentes porque la sección permanecería bloqueada.
- Un proceso que no está en una sección crítica no interfiere de ninguna manera con otros procesos, entonces puede ejecutar lo que quiera.
- La permanencia en la sección crítica debe ser por un tiempo finito y reducido.
- Un proceso puede tener muchas secciones críticas.

Condiciones de Bernstein: si se cumplen las siguientes condiciones entonces no existe la posibilidad de condición de carrera y no estamos en presencia de una sección crítica.

- R(A) intersección W(B) = {Ø}: las variables que va a leer el proceso A no deben ser las que va a escribir el proceso B.
- R(B) intersección W(A) = {Ø}: es igual a la anterior.
- W(A) intersección W(B) = {Ø}: el conjunto de escritura de A debe ser distinto al conjunto de escritura de B. No deben escribir sobre las mismas variables (variables, archivos o cualquier recurso).

Si alguna de las condiciones no se cumple estamos en presencia de una sección crítica.

POSIBLES SOLUCIONES PARA LA CONDICIÓN DE CARRERA/SECCIÓN CRÍTICA

Tipos de soluciones: existen 4 tipos.

- De software: implican que el programador modifique su código (ya sea usando flags, estructuras de condiciones, etc).
- De hardware: hacen uso de instrucciones provistas por el HW para resolver la condición de carrera.
- Provistas por el SO: semáforos.
- Provistas por los lenguajes de programación: se llaman "monitores". En general están provistas por los lenguajes orientados a objetos y están relacionadas con el concepto de encapsulamiento.

Ahora vamos a ver cada tipo más en detalle y varias soluciones propuestas con sus fallas.

Soluciones de software: todas las soluciones de software tienen el problema de espera activa, lo que produce un alto grado de overhead.

Solución 1:

int turno = 0;	
Proceso 0:	Proceso 1:
<pre>while(turno!=0) /*nada*/; ../* SC */.. turno = 1;</pre>	<pre>while(turno!=1) /*nada*/; ../* SC */.. turno = 0;</pre>

Se utiliza una sentencia while que realiza una especie de bloqueo hasta que cada proceso pueda ejecutar su sección. El proceso 0 va a ejecutar sus sentencias y cuando termine va a avisar que finalizó cambiando la variable *turno* a 1.

Ventajas:

- Cumple con la mutua exclusión porque me aseguro que en la sección crítica haya únicamente un proceso ejecutando a la vez.

Problemas:

- Esta solución contempla únicamente dos procesos. De tener más, debería modificar el código.
- Tiene espera activa: la espera activa es cuando un proceso/hilo evalúa repetidamente si una condición es cumplida o no (en este caso el *while*). Es una mala práctica dado que el proceso ocupa la CPU para realizar un procesamiento no útil.
- No hay progreso. En este caso no se nota tanto pero si tuviéramos otro proceso, P2, cuando termine de ejecutar podría poner *turno* = 0, por lo que no dejaría usar la sección al P1 por más que este pueda hacerlo.
- Alternancia: los procesos tienen un orden para acceder a la sección crítica.

Solución 2:

int estado[] = {falso , falso};	
Proceso 0:	Proceso 1:
<pre>while(estado[1]) /*nada*/; estado[0] = true; ../* SC */.. estado[0] = false;</pre>	<pre>while(estado[0]) /*nada*/; estado[1] = true; ../* SC */.. estado[1] = false;</pre>

Se tiene un array con el estado de todos los procesos. Cuando un proceso empieza a ejecutar debe poner su *estado* en true. Cuando termina lo pone en false. En el *while* cada proceso evalúa si el estado de los otros procesos es false (lo que indicaría que no están usando la sección crítica).

Ventajas:

- En parte soluciona el problema de la alternancia de la solución anterior porque ya no hay un orden para ejecutar. El proceso se fija si la sección crítica está vacía y si lo está la usa.
- Cumple con progreso.

Problemas:

- Hay espera activa.
- No cumple con la mutua exclusión. Por ejemplo, supongamos que se produce una interrupción por quantum en este instante:

int estado[] = {falso , falso};	
Proceso 0:	Proceso 1:
while(estado[1]) /*nada*/; estado[0] = true; ../* SC */.. estado[0] = false;	while(estado[0]) /*nada*/; estado[1] = true; ../* SC */.. estado[1] = false;

Cuando el otro proceso, el P1, comience a ejecutar y entre en el while, va a preguntar ¿alguien está usando la sección crítica? la respuesta es no (porque el P0 se interrumpió justo antes de avisar que iba a usar la sección crítica). Entonces comenzará a ejecutar la sección crítica el P1 y pondrá su *estado* en true. Cuando vuelva a ejecutar el P0 podrá acceder a la sección crítica (dado que ya pasó el while), habiendo dos procesos ejecutando en la sección crítica.

Solución 3:

int estado[] = {falso , falso};	
Proceso 0:	Proceso 1:
estado[0] = true; while(estado[1]) /*nada*/; ../* SC */.. estado[0] = false;	estado[1] = true; while(estado[0]) /*nada*/; ../* SC */.. estado[1] = false;

Esta solución surge a partir de la anterior. Si el problema de la anterior está cuando pregunto si puedo entrar y luego me declaro como que estoy usando la sección crítica, ¿por qué no lo hacemos al revés? primero me declaro en true y luego entro a la sección.

Ventajas:

- Contraria a la anterior, esta sí resuelve la mutua exclusión.

Problemas:

- No resuelve el progreso.
- Los procesos pueden entrar en deadlock. Esto quiere decir que se quedan los procesos bloqueados. Supongamos que se produce una interrupción en este instante:

int estado[] = {falso , falso};	
Proceso 0:	Proceso 1:
estado[0] = true; while(estado[1]) /*nada*/;	estado[1] = true; while(estado[0]) /*nada*/;
../* SC */..	../* SC */..
estado[0] = false;	estado[1] = false;

Entonces, supongamos que el P0 se declara como interesado y justo ahí ocurre una interrupción. Cuando empieza a ejecutar el P1, también se va a declarar como interesado, pero cuando entre al while le va a decir que el P0 está usando la sección crítica. Luego, cuando el P0 vuelve a ejecutar, le va a tocar entrar al while, donde le va a salir que el P1 está usando la sección crítica. Así, ambos procesos entrarán en deadlock y ninguno podrá acceder al recurso.

Solución 4:

int estado[] = {falso , falso};	
Proceso 0:	Proceso 1:
estado[0] = true; while(estado[1]) { estado[0] = false; sleep(); estado[0] = true; }	estado[1] = true; while(estado[0]) { estado[1] = false; sleep(); estado[1] = true; }
../* SC */..	../* SC */..
estado[0] = false;	estado[1] = false;

De nuevo, esta solución nace para intentar solucionar el problema de la solución anterior. Acá lo que ocurre es que ambos procesos ceden constantemente el paso a los otros procesos. Comienza cuando el proceso se declara como interesado. Entra al while y pregunta si la sección está ocupada. Si no lo está, entra y se vuelve a declarar como desinteresada y hace un sleep (para ceder al paso a algún otro proceso). Si ningún proceso la toma entonces se vuelve a declarar como interesada y sale del while para acceder a la sección.

Ventajas:

- Respeta la mutua exclusión.
- No tiene deadlock.

Problemas:

- Livelock: un livelock es similar a un deadlock, excepto que el estado de los dos procesos involucrados en el livelock cambia constantemente con respecto al otro mientras ningún proceso realiza un procesamiento útil. Livelock es una forma de inanición y la definición general sólo dice que un proceso específico no está procesando.

En un ejemplo del mundo real un livelock ocurre, por ejemplo, cuando dos personas que se encuentran en un pasillo angosto avanzando en sentidos opuestos tratan de ser amables con la otra, moviéndose a un lado para dejar pasar a la otra persona, pero terminan moviéndose

de lado a lado sin tener ningún progreso, pues ambos se mueven hacia el mismo lado, al mismo tiempo.

- Hay espera activa.

Soluciones de software que sí funcionan (pero tienen espera activa):

- Algoritmo de Dekker.
- Algoritmo de Peterson.
 - Soluciona el problema de la condición de carrera.
 - Tiene espera activa.
 - Considera una “sección de entrada” y una “sección de salida” para la mutua exclusión.
 - Cumple con progreso.

Soluciones de hardware: se llaman así porque utilizan las instrucciones que provee el procesador (set de instrucciones). Una de las soluciones tiene espera activa.

Solución 1:

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

Algunas de las instrucciones que posee el procesador sirven para deshabilitar o habilitar interrupciones. Si recordamos, justo todos los problemas de las soluciones de software se daban porque surgían interrupciones inesperadas. Entonces, deshabilitando las interrupciones nos aseguramos que un programa ejecute en la sección crítica por completo y, cuando sale de la sección crítica, volvemos a habilitar las interrupciones.

Lo que hacemos es deshabilitar las interrupciones (sólo se pueden deshabilitar las enmascarables), luego ejecutamos la sección crítica y finalmente volvemos a habilitarlas.

Ventajas:

- Cumple con el objetivo.

Problemas:

- Puede ocurrir que el proceso muera antes de poder volver a habilitar las interrupciones.
- Las interrupciones son importantes como para deshabilitarlas. No es bueno para la seguridad. Deshabilitarlas sería darle mucho poder a un proceso.
- El hecho de deshabilitar y habilitar las interrupciones para acceder a la sección crítica vuelve al acceso más lento.

Solución 2: consiste en utilizar instrucciones especiales del procesador.

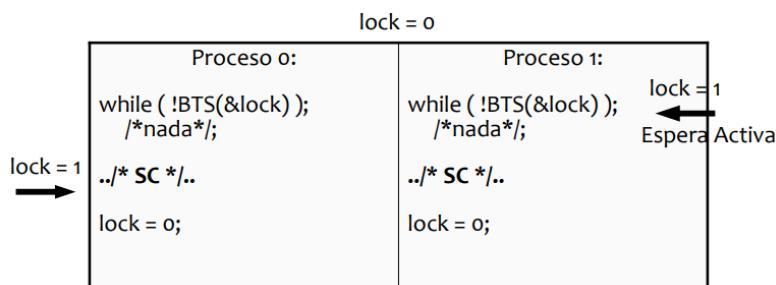
Test and Set (no ejecuta en modo kernel): lo que hace es preguntar y, de acuerdo a la pregunta que se hace (que es en realidad una condición) se define un valor o no. Entonces, sigue la misma lógica que las soluciones de software dado que realiza una pregunta para poder acceder a la sección crítica. Pero, tiene una ventaja y es que las instrucciones no se pueden interrumpir. Las instrucciones se ejecutan por completo y después se pregunta si hubo interrupciones.

```

BTS(*lock) { //Test_and_set
    if (*lock == 0) {
        *lock = 1;
        return TRUE;
    }
    else
        return FALSE;
}

```

Acá, pregunta por la variable *lock*. Si es 0 la define en 1 y devuelve *verdadero* y, si no es igual a cero, devuelve *falso*. Entonces, lo que hace test and set es preguntar y asignar un valor, todo en una misma instrucción. Es decir, este ejemplo que estamos viendo ahora en realidad es una instrucción sola que hace todo eso, nosotros no escribimos esas líneas de código en nuestro programa. Pero, el Test and Set no nos resuelve el problema completamente. Vamos a seguir teniendo que usar un while (lo que indica que esta solución tiene espera activa) pero por lo menos vamos a asegurar la mutua exclusión. Vamos a ver cómo implementarlo.



Comenzamos con una variable compartida *lock* que le va a indicar a cada proceso si puede o no entrar a la sección. Lock = 0 indica que mi sección crítica está disponible. Comienza el P0 y ve que lock es igual a 0, entonces BTS lo pone en 1 y devuelve true. Como el ! del while lo vuelve falso, salgo del while y accedo a la sección crítica. Hicimos todo eso en una única instrucción así que ese problema de que se interrumpa al programa en el medio no puede ocurrir más.

Características:

- No provoca la alternancia en la ejecución de los procesos.

Ventajas:

- Resuelve el problema de la mutua exclusión y el progreso.

Problemas:

- Tiene espera activa.

Soluciones provistas por el SO: los SOs proveen una herramienta, los semáforos, que sirven tanto para trabajar con la mutua exclusión como para sincronizar u ordenar la ejecución de distintos procesos. También permiten limitar o controlar la cantidad de accesos a un recurso.

Se implementan mediante dos llamadas al sistema: *wait()* y *signal()*² (además de las llamadas al sistema que se usan para crearlos y eliminarlos). Vamos a ver cómo está conformado un semáforo y qué hacen estas funciones:

```

struct semaphore {
    int count;
    queueType queue;
};

```

² Nota: signal no es una función bloqueante.

El struct está compuesto por un int que puede tomar cualquier valor (esto tiene algunas limitaciones, más adelante las explicamos) y por una lista de procesos bloqueados.

Vamos a ver lo que hace wait() y signal().

<pre>wait(s) { s->valor--; if (s->valor < 0); bloquar(pid, s->lista); }</pre>	<pre>signal(s) { s->valor++; if (s->valor <= 0); pid = despertar(s->lista);</pre>
---	---

- Wait(): decrementa el valor del semáforo en 1. Después pregunta si el valor del semáforo es menor que 0. Si lo es, este proceso que está haciendo el wait se va a bloquear.
- Signal(): incrementa el valor del semáforo en 1. Después pregunta si el valor del semáforo es menor o igual a 0. Si lo es, desbloquea a alguno de los procesos que estaban bloqueados.

Valores de un semáforo:

- Valor de inicialización: siempre es un valor positivo o cero. Nunca puede inicializarse en un valor negativo.
- Si es mayor a cero: representa la cantidad de instancias disponibles de un recurso.
- Si es menor a cero: representa la cantidad de procesos/hilos bloqueados en espera a ser llamados.
- Si es cero: quiere decir que no hay disponibilidad y tampoco nadie en espera.

Utilidades:

- Mutua exclusión: se utiliza un semáforo llamado *mutex*. Cuando un proceso quiere entrar a la sección crítica hace un *wait()* y cuando sale hace un *signal()*.

s->valor = 1	
Proceso 0: <pre>wait(s); /* SC */ signal(s);</pre>	Proceso 1: <pre>wait(s); /* SC */ signal(s);</pre>

- Sincronización de procesos: se utiliza un semáforo *binario*. Supongamos que queremos que una sección de código de un proceso se ejecute necesariamente después de que se ejecute otra sección de otro proceso. Como no sabemos en qué orden el SO va a designar a los procesos el uso de la CPU, necesito que los procesos estén sincronizados.

En este ejemplo tenemos dos procesos y queremos que ejecute primero P0 y luego P1. Entonces declaramos dos semáforos, s = 1 y q = 0. Ahora, para entender cómo funciona supongamos los dos casos:

- Comienza P0: se hace un wait de s y su valor pasa a 0. Se ejecuta la sección de código en cuestión y se señala al semáforo q, quedando su valor en 1. Cuando ejecute el P1, va a hacer un wait de q y su valor va a quedar en 0. Va a ejecutar su código normalmente y, cuando termine, va a señalar a s.
- Comienza P1: con q inicialmente en 0, al hacer un wait su valor quedará en -1. Al ser un valor menor a 0, se va a bloquear al proceso que hizo el wait. O sea, se va a bloquear a P1 y va a permanecer bloqueado hasta que el P0 haga un signal de q que lo despierte. Esto garantiza que siempre van a ejecutar en orden P0, P1, P0, ...

s->valor = 1 / q->valor = 0	
Proceso 0:	Proceso 1:
<pre>wait(s); /* código que no tiene por qué ser sección crítica */ signal(q);</pre>	<pre>wait(q); /* código que no tiene por qué ser sección crítica */ signal(s);</pre>

- Controlar accesos a recursos (n instancias): declaramos un semáforo con un valor igual a la cantidad de instancias de un recurso. Supongamos un recurso con 3 instancias. El primer recurso que se solicite dejará al semáforo en 2 (tiene sentido, quedan dos instancias disponibles). Luego, si otro proceso solicita una instancia quedará el semáforo en 1. Y cuando venga otro proceso tomará el último recurso y dejará el semáforo en 0. El tema es cuando llegue un nuevo proceso. Al hacer el wait, el semáforo bloqueará al proceso y lo agregaría a la cola de procesos bloqueados (o, procesos en espera para usar el recurso). El valor del semáforo será -1.

s->valor = N	
Proceso 0:	Proceso 1:
<pre>wait(s); .../* SC */.. signal(s);</pre>	<pre>wait(s); .../* SC */.. signal(s);</pre>

Tipos de semáforos:

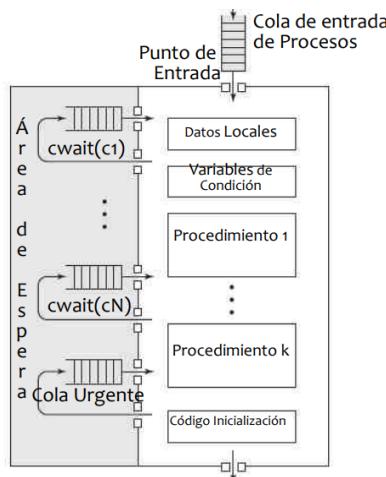
- Contador/General: permite controlar el acceso a una cantidad de recursos. Se inicializa en un valor mayor a 0, específicamente en n (siendo n la cantidad de instancias totales del recurso).
- Binario: permite garantizar un orden de ejecución. Representa libre u ocupado. Se inicializa en 0 o 1. También puede usarse para proteger recursos pero se usa de forma diferente (nunca vamos a saber ni cuantas instancias tenemos disponibles ni cuántos procesos están bloqueados, sólo vamos a saber cuándo hay recursos disponibles y cuándo no).
 - Mutex: es una variante del binario utilizada para garantizar la mutua exclusión sobre un recurso o sección crítica. Siempre se inicializan en 1 y resuelven el problema de condición de carrera.

¿Pueden ser interrumpidos los semáforos? sí y no. Wait y signal deben implementar alguno de los métodos que vimos (de software, hardware) para garantizar que, en caso de ser interrumpidos, no afecten el funcionamiento de los semáforos. En caso de que la solución sea deshabilitar las interrupciones (que tanto wait como signal pueden hacer dado que ambas ejecutan en modo kernel), se dirá que son atómicas, es decir, se ejecutan por completo en un ciclo de instrucción o no se ejecutan. No pueden ser interrumpidas.

¿wait() y signal() deben necesariamente deshabilitar las interrupciones? no. La implementación interna de las funciones wait()/signal() no necesita deshabilitar las interrupciones para su correcto funcionamiento.

Orden de la cola de bloqueados de un semáforo: el algoritmo FIFO parece ser el más justo y es el que vamos a usar en los ejercicios aunque puede usarse otro tipo de planificación.

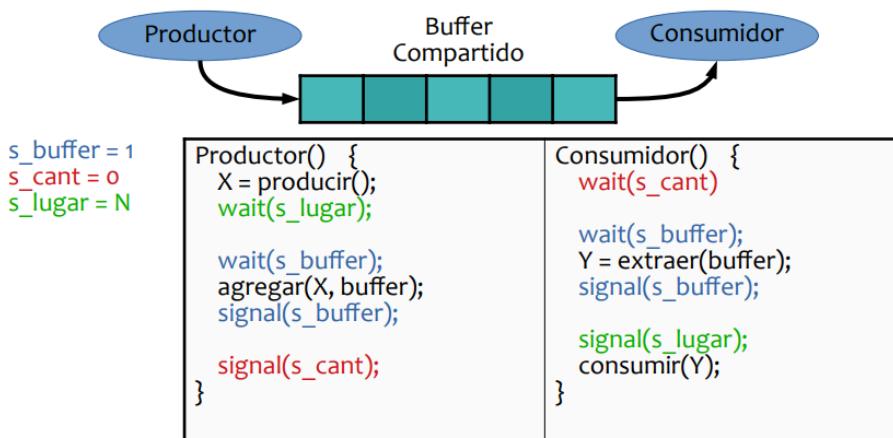
Soluciones provistas por los lenguajes de programación: son soluciones provistas por los lenguajes orientados a objetos haciendo uso del encapsulamiento. Un monitor va a ser una clase que va a permitir que únicamente un proceso ejecute en el monitor a la vez. Cuando ejecute, va a poder ejecutar cualquiera de los procedimientos definidos en el monitor. Pero, de nuevo, sólo un proceso puede estar activo en el monitor a la vez. Los demás procesos permanecerán en el área de espera. El monitor posee datos locales que sólo él puede modificar. Si un proceso quiere hacer uso de estos datos, deberá invocar a alguno de los procedimientos del monitor. De estar disponible, lo ejecutará, de otra forma, pasará al área de espera.



Ventajas:

- Provee la mutua exclusión.
- Permite sincronizar procesos

PRODUCTOR Y CONSUMIDOR



1. Identificamos que el buffer es un recurso compartido entre el productor y el consumidor. El productor agrega y el consumidor quita, por lo que los dos procesos modifican al recurso. Es entonces una sección crítica. Entonces tenemos que usar un semáforo mutex para garantizar la mutua exclusión sobre el buffer. Como es un mutex lo inicializamos en 1.

`s_buffer = 1`

2. Ahora nos damos cuenta de otra cosa: queremos que el productor ejecute primero (porque si el buffer está vacío al consumidor no le serviría). Es decir, queremos que cuando el consumidor quiera consumir haya algo en el buffer. Vamos a necesitar 2 semáforos. Un semáforo va a representar “**agregué elementos al buffer entonces hay cantidad suficiente para que puedas consumir**” y otro que represente “**hay x lugares disponibles en el buffer para que pongas más elementos**”. Pongámosle:

- **s_lugar = N** (porque al principio hay N lugares disponibles en el buffer). Este es, justamente, un semáforo de tipo contador que mide la cantidad de lugares (recursos) disponibles.
- **s_cant= 0** (porque al principio no hay elementos en el buffer).

PREGUNTAS DE PARCIAL

1. Definir el concepto de sección crítica. ¿Qué condiciones debe cumplir? Mostrar un ejemplo del uso de soluciones hardware dentro de las primitivas wait/signal.

La sección crítica es aquella donde se accede o modifica recursos que son compartidos entre dos o más procesos/hilos y donde puede producirse la condición de carrera.

Condiciones:

- Mutua exclusión: debe haber un único proceso/hilo ejecutando en la sección crítica a la vez.
- Progreso: cuando un proceso/hilo termina de ejecutar en la sección crítica debe dar aviso a los demás para que puedan acceder o permanecerán bloqueados. Un proceso que no está en la sección crítica no debe impedir el paso de los otros.
- Espera limitada: se relaciona con el progreso. Un proceso/hilo podrá acceder en algún momento a la sección crítica y no debería quedarse esperando indefinidamente.
- Velocidad relativa: nunca se sabe con certeza cuánto durará la ejecución de una sentencia por lo que no puede saberse de antemano cuánto tiempo estará un proceso en la sección crítica.

Ejemplo wait y signal con solución de hardware:

```
wait() {  
    deshabilitar_interrupciones();  
    semáforo --;  
    if (semáforo < 0) bloquear_proceso();  
    habilitar_interrupciones();  
}  
  
signal() {  
    deshabilitar_interrupciones();  
    semáforo ++;  
    if (semáforo <= 0) despertar_proceso();  
    habilitar_interrupciones();  
}
```

2. Describa brevemente los tipos de semáforos y qué problema resuelve cada uno. ¿Cómo solucionaría un problema de productor/consumidor con un buffer infinito utilizando semáforos?

Contador/General: es un semáforo inicializado en n (siendo n las instancias totales de un recurso) que permite controlar el acceso a una cantidad limitada de recursos.

Binario: puede tomar el valor de 0 o 1 ("libre" u "ocupado"). Sirve para sincronizar a dos o más procesos.

Mutex: es un semáforo binario utilizado para garantizar la mutua exclusión sobre un recurso o sección crítica.

3. ¿Cómo solucionaría un problema de productor/consumidor con un buffer infinito utilizando semáforos?

- Necesitaría 1 semáforo mutex para garantizar la mutua exclusión del recurso compartido (que es el buffer).
- No necesitaría un semáforo s_lugar porque al ser infinito siempre habría lugar para que el productor agregue nuevos elementos.

- Sí necesitaría un semáforo s_cant (de tipo general) para que el consumidor sepa si hay o no cantidad disponible para retirar.

4. V o F. Los semáforos, aún bien usados, pueden llegar a generar problemas en sistemas que utilicen planificadores de corto plazo basados en prioridades.

Verdadero. Por ejemplo, un proceso de mayor prioridad puede quedarse bloqueado por estar esperando un evento que genere un proceso de menor prioridad.

5. ¿Qué problema resuelve un semáforo mutex? ¿De qué otra forma podría resolver el mismo problema? Si en cierto momento el valor de dicho semáforo es negativo, ¿qué implicancias puede asumir?

Un semáforo mutex se utiliza para asegurar la mutua exclusión sobre un recurso o sección crítica por lo que resuelve el problema de condición de carrera. Este problema puede resolverse con alguna solución de software (por ejemplo Dekker) o hardware. El valor negativo de un mutex quiere decir que hay procesos bloqueados en espera del recurso o sección donde se quiere garantizar la mutua exclusión dado que está siendo utilizado por otro hilo/proceso.

6. Mencione los requisitos mínimos y deseables para la correcta aplicación de la mutua exclusión. Indique al menos dos ventajas del uso de semáforos por sobre soluciones de software

No entiendo la primera parte de la pregunta

Ventajas:

- Los semáforos no tienen espera activa.
- Son manejados por el SO.

7. ¿Qué problema principal busca solucionar la mutua exclusión de la sección crítica? Explique por qué ocurre, muéstrela en un ejemplo (en pseudocódigo) y sincronícelo con dos estrategias de sincronización distintas

La mutua exclusión establece que solamente puede haber un hilo/proceso en la sección crítica a la vez. Esto es así para evitar la condición de carrera, que hace referencia a la obtención de resultados diferentes de acuerdo al orden en que accedan los hilos/procesos al recurso en cuestión. Esto se debe a que acceden a datos compartidos y que acciones que para nosotros son atómicas en realidad no lo son para el procesador.

Ejemplo:

```
int a = 0; // variable global
```

Proceso 1	Proceso 2
<pre>var c = a; c++; a = c;</pre>	<pre>var b = a; b--; a = b;</pre>

Según el orden en que ejecuten pueden dar distintos resultados. Si se ejecutan todas las instrucciones de un proceso y luego las del otro, el resultado final de a será 0.

Sin embargo, si primero ejecutan la primera instrucción en ambos casos, el resultado final podrá ser 1 o -1 según cuál sea el último que setee el valor a a.

- Usando mutex: mutex = 1

Proceso 1	Proceso 2
<pre>wait(mutex); var c = a; c++; a = c; signal(mutex);</pre>	<pre>wait(mutex); var b = a; b--; a = b; signal(mutex);</pre>

- Desabilitando interrupciones:

Proceso 1	Proceso 2
<pre>deshabilitar_interrupciones(); var c = a; c++; a = c; habilitar_interrupciones();</pre>	<pre>deshabilitar_interrupciones(); var b = a; b--; a = b; habilitar_interrupciones();</pre>

8. ¿Sería eficiente el intentar detectar un bug ocasionado por una condición de carrera debugueando el programa?

No sería eficiente porque los bugs ocasionados por una condición de carrera dependen justamente de la velocidad relativa de ejecución de dos programas, y esta es explícitamente modificada mientras se debuguea (debido a que se frena la ejecución de alguno de ellos de forma temporal) pudiendo generar condiciones bajo las cuales nunca ocurriría el problema.

9. Si la ejecución de las syscalls wait y signal no se realizara en forma atómica generaría condición de carrera.

Verdadero. Las syscalls wait y signal manipulan un recurso común que es el semáforo tanto en forma lectura como de escritura, por lo que aplican las condiciones de Bernstein.

10. V o F. Tanto el uso de semáforos como la deshabilitación/habilitación de interrupciones, son técnicas que los usuarios pueden utilizar para resolver problemas de mutua exclusión sin producir espera activa.

Verdadero (?)

11. Explique por qué los semáforos cumplen con las condiciones para ser una buena solución a la sección crítica. ¿Cómo podría lograr el SO que sus syscalls wait y signal sean atómicas?

Los semáforos son una buena solución dado que garantizan mutua exclusión sobre la sección crítica (solucionando el problema de condición de carrera), cumplen con progreso y la espera es limitada.

Para que wait() y signal() sean atómicas podría utilizarse test and set. (?)

12. De un ejemplo de productor/consumidor. Indique qué problemas podrían ocurrir en caso de no sincronizar dicho código. ¿Qué tipo de semáforos debería utilizar? Indique qué problema resuelve cada uno.

Ejemplo: buffer de 5 posiciones.

s_lugar_disponible = 5; (general): el productor sólo va a agregar al buffer cuando haya lugar.

m_buffer = 1; (mutex): resuelve la mutua exclusión sobre el buffer.

elementos_para_retirar = 0; (general): el consumidor sólo va a extraer del buffer cuando el productor haya agregado elementos.

Productor (1 instancia)	Consumidor (1 instancia)
<pre>x = producir(); wait(s_lugar_disponible); wait(m_buffer); agregar(x, buffer); signal(m_buffer); signal(elementos_para_retirar);</pre>	<pre>wait(elementos_para_retirar); wait(m_buffer); Y = extraer(buffer); signal(m_buffer); signal(s_lugar_disponible); consumir(Y);</pre>

13. Explique cómo funcionan los semáforos con espera activa, indicando ventajas y desventajas.

Los semáforos con espera activa son aquellos que implementan alguna solución de software o hardware para garantizar la mutua exclusión.

Ventajas:

- Funcionan. Garantizan mutua exclusión, progreso, espera activa, etc.

Desventajas:

- Mal uso de la cpu (espera activa).

Deadlock

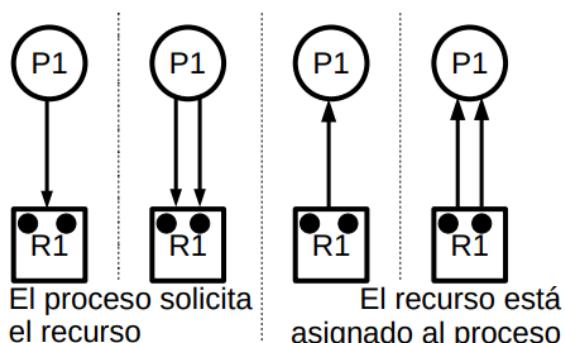
INTRODUCCIÓN

Definición: es el bloqueo permanente de un conjunto de procesos donde cada uno de estos procesos está esperando un evento que sólo puede generar un proceso del conjunto. Es una consecuencia de la mala sincronización o un mal uso de los recursos compartidos entre procesos.

- Recursos limitados: los procesos los solicitan, usan y finalmente liberan. Si el proceso necesita un recurso y no está disponible, se bloquea hasta que el recurso se encuentre disponible. Hay varios recursos:
 - Gestionados por el SO (generalmente no necesitan ningún tipo de gestión por parte de los desarrolladores).
 - No gestionados por el SO. En estos es donde se puede dar el deadlock.
- Tipos de recursos: el deadlock se da con los recursos reutilizables.
 - Reutilizables: un proceso lo solicita, lo utiliza y lo libera para que lo use otro proceso. Por lo general vamos a trabajar sobre estos.
 - Consumibles: puede ser utilizado una vez y luego “desaparece”, se consume. No hay “espera”, quien toma el recurso lo utiliza y listo.

GRAFO DE ASIGNACIÓN DE RECURSOS

Definición: es una de las herramientas que tenemos para empezar a analizar el problema de deadlock. El grafo me muestra, para un determinado instante, cómo es la asignación de recursos.



Cuadrados: recursos.

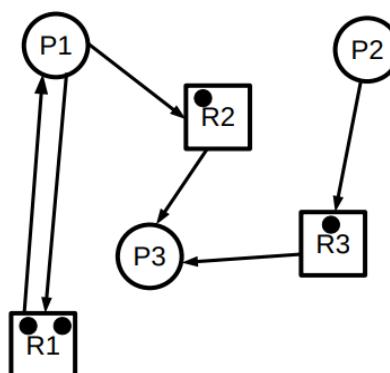
Círculos: procesos.

Aristas: cantidad de recursos solicitados o asignados.

Puntitos: cantidad de instancias de un recurso en particular.

Esto se puede representar en una matriz.

Ejemplo:



- Hay 3 procesos.
- Hay 1 recurso con dos instancias.
- Hay 2 recursos con una instancia.
- El P1 solicita 1 de R1 y 1 de R2.
- El P1 tiene asignado 1 de R1.
- El P2 solicita 1 de R3.
- El P2 no tiene recursos asignados.
- El P3 no solicita recursos.
- El P3 tiene asignado 1 de R2 y 1 de R3.

Cómo ver el deadlock en el grafo:

- Si no hay ciclos → no hay deadlock.
- Si hay un ciclo → podría o no haber deadlock.
- Si hay un ciclo y todos los recursos implicados en ese ciclo tienen una sola instancia → hay deadlock.

No se puede justificar la existencia de deadlock mediante el grafo, hay que usar un algoritmo.

Condiciones para la existencia de deadlock:

- Condiciones necesarias.
 - Que exista la mutua exclusión: es decir, que un recurso deba ser usado únicamente por un único proceso/hilo en todo instante.
 - Que haya retención y espera (un proceso toma ciertos recursos, los retiene y pide nuevos recursos. No va a liberar los viejos hasta que no le den los nuevos) GOD ENERGY.
 - Que no haya desalojo de recursos: si el SO pudiera sacarle a un proceso un recurso que está reteniendo, habría desalojo de recursos. Cuando no hay desalojo, es el mismo proceso el que retendrá el recurso hasta que decida liberarlo.
- Condiciones necesarias y suficientes:
 - Las tres mencionadas anteriormente.
 - Espera circular: cada proceso/hilo está bloqueado esperando un evento que lo dé otro proceso que está dentro de esta espera, esperando lo mismo.

CÓMO LIDIAR CON UN DEADLOCK

Estrategias: vamos a ver 4 estrategias para encarar los deadlocks.

- Alternativa 1 - prevención: garantiza que no ocurrirá deadlock.
- Alternativa 2 - evasión o predicción: garantiza que no ocurrirá deadlock.
- Alternativa 3 - detección y recuperación: ocurre deadlock.
- Alternativa 4 - no tratarlo: ocurre deadlock.

Alternativa 1 - prevenirlo:

- Garantiza que no ocurrirá deadlock.
- Se encarga de impedir que se produzca alguna de las cuatro condiciones necesarias y suficientes para que exista deadlock. Con que una no se cumpla es suficiente para que no ocurra deadlock en un sistema.
 - Mutua exclusión: no siempre puede evitarse. Si hay recursos que no se pueden compartir no puede evitarse.
 - Retención y espera: el proceso debe solicitar todos los recursos que va a utilizar de forma simultánea, y si alguno de los mismos no está disponible, la totalidad de la solicitud es denegada.
Lo malo de esta solución es que si un proceso que ejecuta durante mucho tiempo tiene asignados recursos que va a usar durante un periodo muy corto de tiempo y no los libera, hay una ineficiencia de esos recursos. Esto puede generar inanición en otros procesos.
 - Desalojo de recursos: "Si un proceso A solicita un recurso que está asignado a otro proceso B (que está a la espera de más recursos), el recurso asignado al proceso B puede asignarse al proceso A, dado que B también está esperando y no lo está usando".

Para implementar esto los procesos deben poder ser retrotraidos a un estado previo consistente y deben disponer de un mecanismo para ser notificados que les fue des-asignado un recurso.

- Espera circular: asignar un número de orden a los recursos. Los recursos sólo pueden solicitarse en orden creciente. La idea sería que un proceso sólo puede solicitar recursos que sean mayor (en número) al recurso que tenga asignado en ese momento.

Habiendo ocurrido un deadlock, si el SO expropia recursos a un proceso de manera que la espera circular desaparezca, podría de todas maneras volver a producirse la misma espera circular y generar deadlock.

Alternativa 2 - evasión o predicción de deadlock: se trata de 2 técnicas que garantizan que no ocurra deadlock.

- Denegar el inicio de un proceso: se le va a denegar el inicio a un proceso si está pidiendo más recursos de los totales.

$$M_{n+1} + \sum_{i=1}^n M_i \leq RT$$

M_i = Necesidades Máximas declaradas por el proceso i.

n = Cantidad actual de procesos

RT = Recursos Totales del sistema.

- Denegar la asignación de un recurso - algoritmo del banquero: se va a jugar con la cantidad de recursos que un proceso quiere y los que tiene asignados. Se evalúa la solicitud de recursos que hace un proceso, se simula esa asignación y se ve si se termina en un estado seguro o inseguro. Si el estado final es seguro, la solicitud es aceptada.
 - Estado seguro: no habrá deadlock. Se le asigna el recurso al proceso.
 - Estado inseguro: podría existir deadlock si se le asignara el recurso al proceso. No se le asigna el recurso al proceso.

Algoritmo para ver si estoy en estado seguro:

1. Me dan la matriz de Necesidades Máximas y la de Recursos Asignados. Las resto y obtengo la matriz de Necesidades Pendientes.
2. Calculo el vector de Recursos Disponibles.
3. Si hay algún proceso que no tenga necesidades pendientes lo finalizo para que libere los recursos que tiene asignados.
4. Me fijo qué otros procesos pueden finalizar y liberar recursos.
5. Si todos los procesos pueden finalizar y me quedo con todos los recursos totales entonces existe Secuencia Segura, está en Estado Seguro.

Algoritmo para simular asignación de recursos: es lo mismo que el anterior pero el enunciado va a decir, por ejemplo, "¿puede P2 solicitar 2 instancias de R2?. Lo que tengo que hacer es simular esa asignación y ver si se llega a un estado seguro. Para esto:

1. Me fijo si lo que está solicitando está dentro de las peticiones pendientes (si se pasa no sería coherente).
2. Me fijo si hay recursos disponibles para asignarle los que está pidiendo. Seguramente haya, entonces se los asigne. Ahora, con esa asignación el proceso probablemente no finalice (porque falta que se le asignen más recursos para cumplir todas sus necesidades pendientes) así que no me va a liberar ningún recurso.

Construyo la matriz de Necesidades Pendientes teniendo en cuenta la asignación que acabo de hacer.

3. Me fijo si todos los procesos pueden finalizar. Si pueden entonces se le asignarán los recursos al P2 y sino no.

Alternativa 3 - detección y recuperación de deadlock:

- Puede ocurrir deadlock. De hecho, dejamos que ocurra el deadlock y luego aplicamos un mecanismo de recuperación para arreglarlo.
- No hay restricciones para asignar recursos disponibles. Un proceso pide algo y se lo damos.
- Periódicamente se ejecuta el Algoritmo de Detección para determinar la existencia de deadlock.

Opciones de recuperación: estamos en deadlock, ¿qué podemos hacer para solucionarlo? alguna de las siguientes opciones.

- Matar a los procesos involucrados en el deadlock.
- Volver a un estado anterior. Es muy complejo de implementar.
- Identificar los procesos involucrados en el deadlock, matar uno a uno hasta que no haya más deadlock. No se matan a todos de una como en la primera alternativa, se mata a uno, se ve si hay deadlock; si hay, se sigue matando hasta que no haya más deadlock.
- Identificar a los procesos que son parte del deadlock y sacarles los recursos hasta que no exista deadlock.

Criterios de selección de procesos para terminar o expropiar:

- i) Menor tiempo de procesador consumido.
- ii) Menor cantidad de salida producida.
- iii) Mayor tiempo restante estimado.
- iv) Menor número total de recursos asignados.
- v) Menor prioridad.

Algoritmo de detección de deadlock: no necesita conocerse la matriz de necesidades máximas de los procesos.

1. Si no me dieron el vector de Recursos Disponibles lo calculo restando a los recursos totales los asignados.
2. Tacho de la matriz de Recursos Asignados a aquellos procesos que no tengan recursos asignados. También lo tachamos de la matriz de Peticiones.
3. Finalizo a los procesos que no estén peticionando recursos y sumo sus recursos a los Recursos Disponibles.
4. Con el array de Recursos Disponibles me fijo qué proceso puede finalizar (para esto veo la matriz de Peticiones Pendientes).
5. Finalizo a ese proceso incorporando a los Recursos Disponibles los recursos que tenía asignado ese proceso.
6. Hago esto hasta que no pueda finalizar ningún otro proceso. Esos van a ser los procesos que están en deadlock.

Desventajas de esta estrategia:

- Esta estrategia no puede aplicarse en aquellos sistemas donde no podemos permitir que ocurra un deadlock.
- Recuperar el deadlock (matando a los procesos involucrados en el deadlock) puede tener un impacto negativo para las víctimas.

Alternativa 4 - no tratarlo: ¿por qué tendríamos necesidad de recuperarnos del deadlock? La mayoría de los SO no tratan los deadlocks.

PREGUNTAS DE PARCIAL

- 1. ¿En qué se diferencian las técnicas de prevención y las de detección de deadlock? ¿En qué casos utilizaría cada estrategia?**

Se diferencian en que las técnicas de prevención garantizan que no ocurrirá deadlock mientras que las técnicas de detección de deadlock dejan que ocurra y luego intentan recuperar el estado. La estrategia de prevención la usaría en aquellos sistemas donde no podemos permitir que ocurra un deadlock (por ejemplo, el sistema detrás del funcionamiento de un avión o el sistema detrás de un dispositivo médico) y el de detección de deadlock lo usaría para el sistema operativo de una computadora de uso doméstico.

- 2. V o F. Justifique en ambos casos:**

- a. **En una situación de deadlock una buena solución suele ser matar a todos los procesos involucrados.**

Falso. Si bien matar a todos los procesos involucrados en el deadlock es una opción, no es una solución óptima en lo absoluto. Una mejor estrategia es ir eliminando uno a uno a los procesos involucrados hasta que se liberen los recursos necesarios hasta que no exista más deadlock.

- b. **Utilizando el algoritmo de evasión podría llegar a ocurrir un deadlock si es que todos los procesos piden lo máximo que podrían pedir.**

Falso. Este algoritmo asegura que no existirá deadlock.

- 3. Describa las condiciones necesarias y suficientes para la existencia de deadlock.**

Condiciones necesarias:

- Mutua exclusión
- Que no haya desalojo de recursos: el SO no puede quitarle recursos a un proceso una vez que le fueron asignados.
- Retención y espera: un proceso no va a entregar los recursos que ya le fueron otorgados hasta que no se le entreguen los nuevos que está solicitando.

Condiciones suficientes: las 3 anteriores +

- Espera circular: ocurre cuando un proceso está esperando la liberación de un recurso por parte de otro proceso que está en esta misma espera esperando lo mismo.

- 4. V o F. Tanto el uso de prevención como de detección y recuperación de deadlocks podría llegar a generar starvation.**

Verdadero. En la prevención puede producirse starvation en el caso de la retención y espera. En el caso de la recuperación puede generarse starvation si el algoritmo elige siempre al mismo proceso para ser finalizado o desalojado porque podría ocurrir que nunca termine su ejecución.

- 5. Indique las dos formas para prevenir el deadlock utilizando las siguientes condiciones:**

- a. **Retención y Espera.**

El proceso debe solicitar todos los recursos que va a utilizar de forma simultánea, y si alguno de los mismos no está disponible, la totalidad de la solicitud es denegada.

- b. **Sin Desalojo.**

Si un proceso A solicita un recurso que tiene asignado B y B está bloqueado entonces ese recurso podrá serle asignado a A dado que al B estar bloqueado no lo estaba utilizando.

- 6. Explique en detalle la estrategia de prevención del interbloqueo.**

La estrategia de prevención de deadlock garantiza que no ocurrirá deadlock y consiste en impedir que se cumpla alguna de las 4 condiciones necesarias y suficientes para la existencia de deadlock.

7. V o F. No puede ocurrir deadlock en un SO que ejecuta procesos sin concurrencia.

Verdadero. Al ejecutar de a uno los procesos, hasta que no termina no ejecuta otro. Teniendo en cuenta que cuando el proceso termina de ejecutar libera todos los recursos que le hubieran sido asignados, no va a haber retención y espera.

Caso especial: considerar entornos con hilos.

8. ¿En qué se parecen y en qué se diferencian las técnicas de Evasión y Prevención de deadlocks? Proponga una situación en la que sea mejor usar una que otra (justificando por qué)

Se parecen en el hecho de que ambas aseguran que no ocurrirá deadlock. La técnica de Prevención propone lograr que no se cumpla alguna de las 4 condiciones necesarias y suficientes para que exista deadlock mientras que la técnica de Evasión simula la asignación de recursos para ver si esa asignación dejaría al sistema en un estado seguro (en cuyo caso permite la asignación).

La técnica de prevención produce mucho overhead y requiere de muchas estructuras para poder llevarse a cabo por lo que no sería la mejor opción para, por ejemplo, una computadora de uso laboral.

9. V o F. En algunos casos, como por ejemplo semáforos con espera activa, puede darse un deadlock sin que ocurran las 4 condiciones.

Falso. El deadlock sólo se da si se cumplen las 4 condiciones (espera circular, retención y espera, no hay desalojo de recursos y mutua exclusión).

10. En un entorno con un procesador poco potente, muchos recursos y la necesidad de garantizar que no haya Deadlocks, conviene implementar una estrategia de Evasión.

Falso. Teniendo muchos recursos y un procesador poco potente, la Evasión generaría mucho overhead, además de necesitar estructuras de gran tamaño para administrar todos los recursos, con lo cual es recomendable utilizar Prevención.

11. Compare las técnicas de evasión y detección de deadlocks en función de los siguientes atributos: frecuencia de ejecución, overhead generado, criticidad del sistema y flexibilidad a la hora de asignar un recurso.

	Frecuencia de ejecución	Overhead	Criticidad del sistema	Flexibilidad a la hora de asignar un recurso
Evasión	Cada vez que un proceso solicita recursos	Muy alto	Baja	Baja. Si la solicitud fuera a dejar al sistema en un estado inseguro no permite la asignación.
Detección	Una vez que se produce el deadlock	Bajo	Puede llegar a ser muy alta si se mata a los procesos involucrados	Alta flexibilidad, no controla la asignación de recursos.

12. V o F. Cuando se utiliza el algoritmo del banquero, si se detecta un deadlock se puede desalojar recursos para solucionarlo.

Falso. Si se detecta un deadlock no se le asignarán los recursos solicitados al proceso.

13. En la estrategia de evasión de Deadlock mediante algoritmo del banquero, ¿Que significa estado seguro? ¿Podría el sistema quedar en estado inseguro ante alguna situación particular?

Estado seguro significa que el sistema no entrará en deadlock luego de asignar la cantidad peticionada de recursos (es decir, todos los procesos podrán finalizar correctamente, existiendo al menos una secuencia segura de finalización).

Con una correcta implementación de este algoritmo el sistema nunca podría quedar en un estado inseguro dado que rechazaría la petición.

14. Los ULTs de un mismo proceso pueden quedar en deadlock utilizando semáforos.

Falso. Si se bloquea un hilo se bloquean los otros y nunca llega a cumplirse la condición de espera circular. Supongamos que un hilo ULT pide usar un recurso, todos los demás ULTs se van a bloquear, entonces el primer ULT que solicitó el recurso va a poder usarlo tranquilamente dado que nadie va a intentar pedírselo. Nunca se forma una espera circular. Cuando el hilo termine de usar el recurso y se desbloquee, todos los hilos van a hacerlo y el recurso va a estar disponible.

Memoria Real/Principal/Física

INTRODUCCIÓN

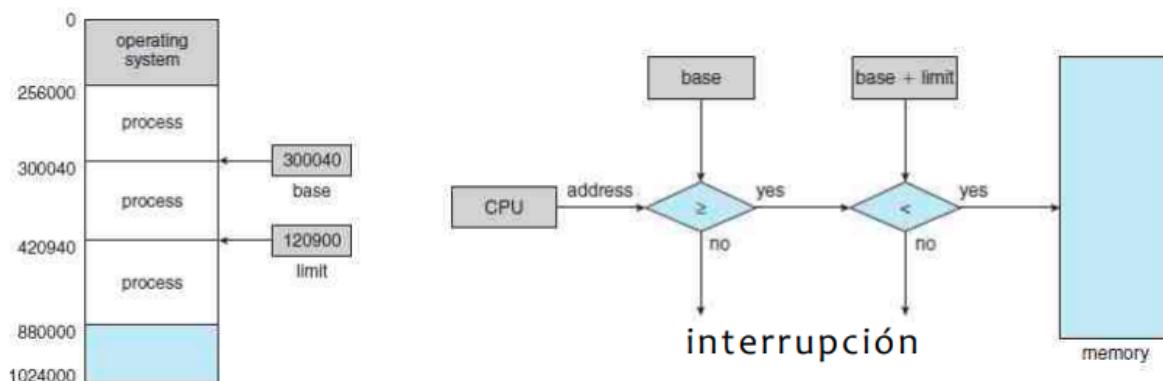
Ejecución de un programa: para poder ejecutarse, el programa debe cargarse completamente a la memoria (es decir, debemos cargar toda su imagen: datos, stack, heap, etc). Vamos a pensar a la memoria RAM como un conjunto de direcciones lineales, un gran array que maneja el SO. Es importante remarcar que la memoria no conoce su contenido.

Entonces, si cada programa que está ejecutando está en un lugar de memoria RAM, en algún lugar el SO debe tener guardado “el P1 está desde esta posición de memoria hasta esta otra, el P2...”. Esto es importantísimo saberlo dado que cada proceso (por cuestiones de seguridad) sólo debe poder acceder al espacio de memoria que le corresponde y no al de otro proceso.

Funciones del SO: en líneas generales, el SO cumple con estas 4 funciones en cuanto al manejo de programas en la memoria RAM:

- Reubicación: puede reubicar a un proceso de un lugar de la RAM a otro. Es necesario, por ejemplo, cuando suspendemos un proceso (es decir, lo mandamos al disco). Lo más probable es que, cuando se desuspenda, no vuelva a ejecutar en el mismo lugar de memoria que antes.
- Protección: permite a los procesos leer y escribir solamente en los espacios de memoria que le corresponden. En caso de que un proceso quiera acceder a la memoria de otro se dispara una *interrupción* y el SO decide qué hacer con ese proceso. Lo más común es que decida finalizarlo dado que comprometió la seguridad de otro proceso.
 - Registro base y límite: se marca dónde inicia y finaliza cada proceso. Para guardar estos datos vamos a tener un registro Base y otro Límite. En las siguientes fotos del algoritmo vemos que, por ejemplo, en un JMP se pregunta si la dirección a la que se quiere saltar está dentro del intervalo base-límite y si no está se produce una interrupción.

Pero, ¿quién seteo el registro base y el límite? El SO.

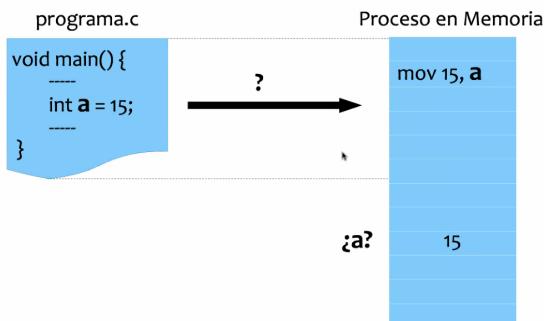


- Compartición: es lo contrario a la protección. Acá se permite que los procesos comparten memoria entre ellos siempre y cuando esté el SO de intermediario. Todos los procesos que comparten ese espacio podrán escribirlo o leerlo de acuerdo a sus permisos. Esto puede ser compartir datos o una biblioteca para ejecutar.
- Organización física y lógica:

- Organización física: conceptos inherentes a la RAM
 - Es de rápido acceso (en comparación a dispositivos de IO).
 - Es volátil.
- Organización lógica: tiene que ver con las distintas estrategias que existen para administrarla que vamos a ver en este capítulo.

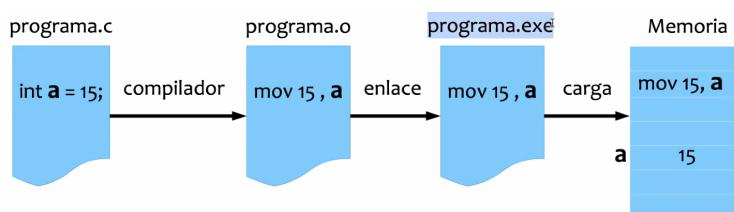
ASIGNACIÓN DE DIRECCIONES (ADDRESS BINDING)

¿Cómo se carga un proceso en memoria?:



En el programa hay un `a = 15`. En la memoria hay un `mov 15, a` pero, ¿cómo se asignó un espacio a `a`? Asociar variables con direcciones de memoria se llama justamente, address binding y requiere de varias etapas.

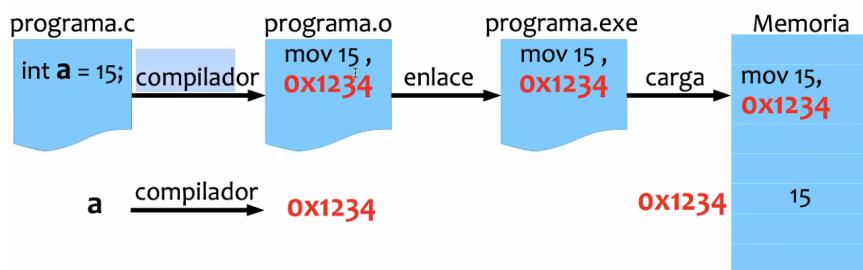
Etapas de la ejecución de un programa:



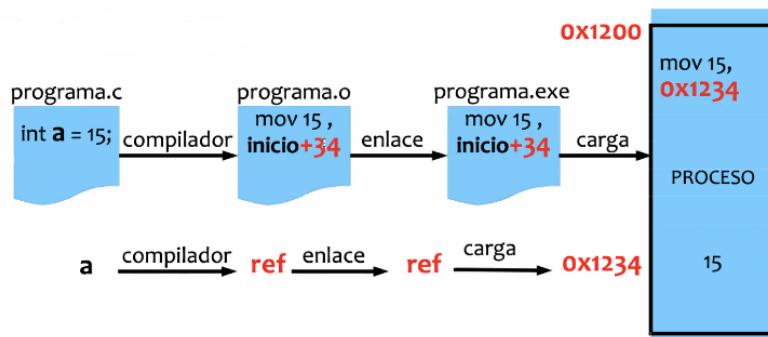
1. programa.c: es el código fuente que programamos en lenguaje C.
→ Compilación: se realiza un análisis léxico, sintáctico y semántico del código fuente y se traduce a código máquina.
2. programa.o: es un archivo en Assembler que la CPU puede entender.
→ Enlazado: se van a buscar todas las bibliotecas que el código necesite. Recién acá está listo para ser ejecutado.
3. programa.exe: este es un archivo ejecutable que se pasa a Memoria para ser ejecutado.
→ Address Binding: se asigna espacio en memoria RAM a nuestro programa para que pueda ejecutar. En otras palabras; cuándo se resuelve la traducción de la variable a una dirección de memoria.

Cuándo se hace el Address Binding: en la sección anterior puse que se hacía luego de generar el ejecutable pero, en realidad, puede hacerse en distintas etapas:

- En tiempo de compilación: si es el compilador quien se encarga de hacerlo, esto quiere decir que el programa ya va a saber la dirección donde va a estar cada variable. Así funcionaban los compiladores al principio.
 - Desventajas:
 - No se lleva bien con el concepto de reubicar un programa en la memoria dado que habría que “editar” todas las referencias y recompilar el programa.
 - Si quiero ejecutar una segunda instancia de un programa, al tener las mismas direcciones definidas, una instancia “pisaría” a la otra.
 - El compilador debe tener cierto conocimiento de las direcciones que maneja la memoria RAM de cada computadora para poder “generar” direcciones válidas. Aquí se ve claramente qué limitado es este método.

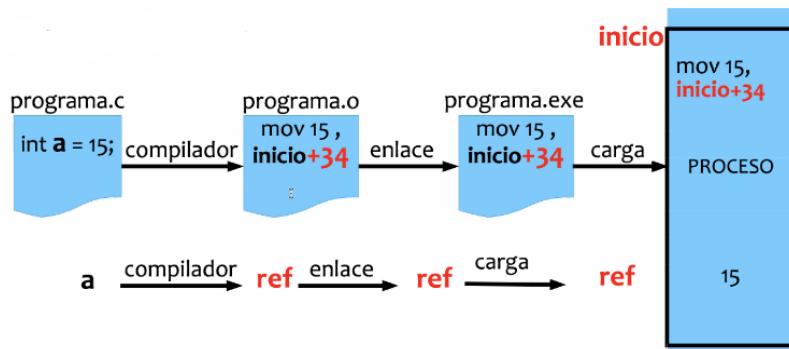


- En tiempo de carga: como vemos en la imagen, no se tiene una posición de memoria definida, sino que se tiene un desplazamiento desde la que sería la posición inicial (que se va a conocer una vez que se cargue el programa en la memoria). La variable a va a estar cargada al comienzo de donde sea que esté cargado el programa + un desplazamiento desde ese lugar.
 - Ventajas:
 - Ahora sí puede ejecutarse más de 1 instancia de un programa.
 - Desventajas
 - La posición inicial que se le define a un proceso va a ser siempre la misma durante todo su ciclo de vida. Entonces, si un proceso es suspendido, cuando quiera volver a ejecutar no va a poder ser reubicado y si su lugar ha sido ocupado por otro programa, no podrá ejecutar.



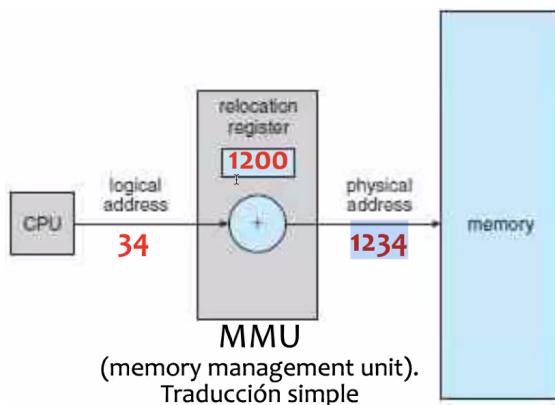
- En tiempo de ejecución: esto es lo que se hace hoy en día. La instrucción permanece del tipo “inicio + 34” y no se hardcodea la posición de memoria sino que el hardware la calcula en cada ejecución. Cada vez que una instrucción se ejecuta se toman esas direcciones relativas y se traducen (se traducen recién cuando quieren ser usadas).
 - Desventajas:
 - Produce mucho overhead dado que hay que volver a calcular las posiciones de memoria cada vez que se lo vuelva a cargar en memoria. En este caso

cuando hablamos de overhead no nos referimos al CPU, sino a un dispositivo llamado MMU que vamos a ver a continuación.



MMU (MEMORY MANAGEMENT UNIT)

Definición: módulo que se encarga de hacer la traducción de una dirección lógica a una física. Mejora la eficiencia de la memoria y su ventaja principal es que logra que las traducciones se realicen de una forma muy rápida.



Direcciones:

- Dirección lógica: referencia a una ubicación de memoria utilizada por los procesos. Son independientes de la ubicación real en memoria. Los procesos la conocen. La dirección lógica requiere una traducción para llegar a la dirección física.
 - Dirección relativa: es un tipo de dirección lógica en la que la dirección se expresa según un punto conocido (por ejemplo, "inicio + 34").
- Dirección física o absoluta: referencia a una verdadera ubicación en memoria. Los procesos normalmente no la conocen.

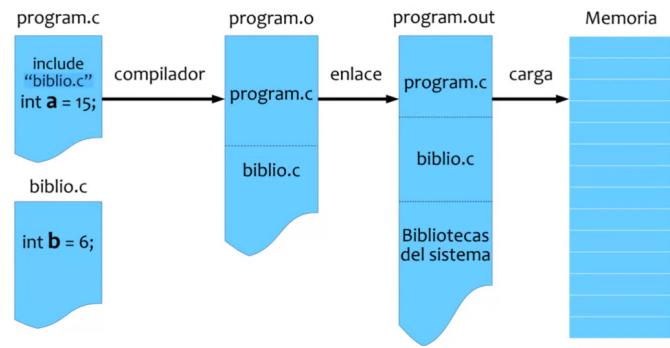
Traducciones: la traducción de dirección lógica a física tiene sentido solamente cuando se hace la asignación en tiempo de ejecución. Cuando la asignación se hace en tiempo de compilación se usa directamente la dirección física. Cuando la asignación es en tiempo de carga se usa la dirección relativa.

CARGA, ENLACE Y BIBLIOTECAS COMPARTIDAS

Introducción: con las bibliotecas pasa lo mismo. Se pueden enlazar en distintos momentos del proceso de construcción y carga del programa.

Enlace estático (en tiempo de compilación):

cuando escribimos nuestro código fuente escribimos sentencias que utilizan funciones de bibliotecas (por ejemplo, stdio.h). Este método propone que en la etapa de enlace se vayan a buscar esas bibliotecas y se "adjunten" a nuestro programa. Es decir, tenemos un único ejecutable con nuestro programa + las bibliotecas. Si se quiere ejecutar al programa se debe cargar por completo en la memoria.



Acá hay que entender el concepto de que nosotros lo que hacemos es "copiar" el código de la biblioteca y sumarlo a nuestro código. Una vez que se "ensambla" todo en el mismo ejecutable el SO no va a poder distinguir qué código es realmente nuestro programa y cuál es de la biblioteca porque maneja a todo como un bloque.

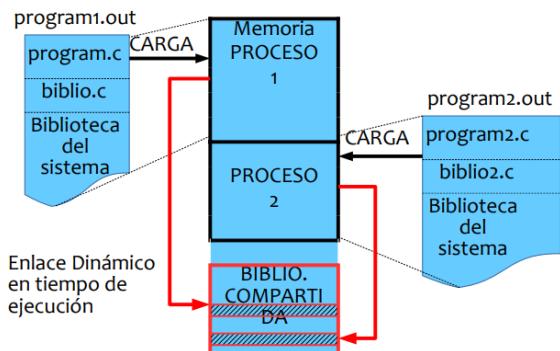
Ventajas:

- Se podría decir que el programa se vuelve un poco "stand alone" al tener las bibliotecas que necesita acopladas a su código.
- Por una parte su distribución es más sencilla dado que se distribuye un ejecutable que ya contiene absolutamente todo lo que necesita.

Desventajas:

- Se tiene un ejecutable mucho más grande. Mientras más bibliotecas use el programa, más grande será el ejecutable.

Enlace dinámico (bibliotecas compartidas): como alternativa al método anterior está este, en el que algunas partes del código pueden mantenerse en disco mientras no se necesiten y cargarse a la memoria ("adjuntarse" al código del programa) en tiempo de ejecución, cuando se necesiten. Por ejemplo, en el caso de una biblioteca, esta se cargará a memoria cuando un proceso la refiere.



Ventajas:

- El ejecutable es más chico.
- Surge la idea de "biblioteca compartida". O sea, siempre que un proceso necesite usar una biblioteca, ésta se va a cargar a la memoria RAM. Una vez que esté en la RAM podrá ser utilizada por varios procesos.

Esta es una ventaja crucial al compararlo con el enlace estático dado que, si tenemos 10 procesos usando a la biblioteca, con este método siempre usaríamos la misma biblioteca para todos ellos mientras que con el enlace estático tendríamos que cargar la biblioteca 10 veces embebida al ejecutable de cada proceso.

Desventajas:

- Su distribución es compleja en el sentido en que la computadora que quiera correr el programa deberá contar con las bibliotecas que este necesita.

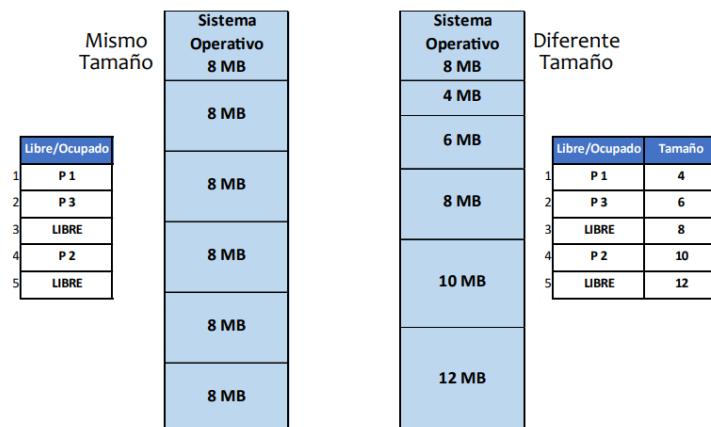
ASIGNACIÓN DE MEMORIA PARA PROCESOS

Definición: el SO debe administrar a todos los procesos que están en memoria (él mismo también está en la memoria pero la porción de memoria que nos interesa administrar es la porción donde están todos los demás procesos) para que no accedan a lugares que no les corresponden. Hay varias técnicas para esto:

- Particionamiento fijo.
- Particionamiento dinámico.
- Buddy system.
- Segmentación.
- Paginación.
- Segmentación paginada.

PARTICIONAMIENTO FIJO

Definición: se divide la memoria en particiones de tamaños fijos. Pueden ser todas las particiones del mismo tamaño o de distintos tamaños pero en ambos casos son tamaños fijos. Una vez que se define no se cambia.



Es importante remarcar que un proceso no puede estar dividido en dos particiones diferentes, es decir, su código es contiguo y debe entrar por completo en una partición.

¿Qué son esas tablas que dicen Libre/Ocupado? esas tablas las maneja el SO (están en su porción de memoria) y son lo que necesita el SO para administrar el particionamiento fijo. Es muy simple, me indica si la partición en cuestión está libre o si la está ocupando algún proceso y cuál.

Cuando las particiones son todas del mismo tamaño, los lugares LIBRES son iguales entre sí, es decir, da lo mismo cargar a un proceso en un lugar libre que en otro. Pero, cuando las particiones son de distinto tamaño, la tabla tiene que conocer el tamaño de cada partición para poder ubicar al proceso en el lugar más óptimo.

Importante: en ambas particiones (mismo tamaño y distinto) puedo ver que el grado de multiprogramación es fijo y está dado por la cantidad máxima de particiones. Mirando la imagen, es 5 en ambos casos.

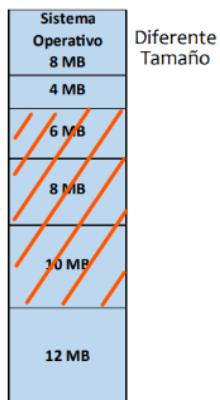
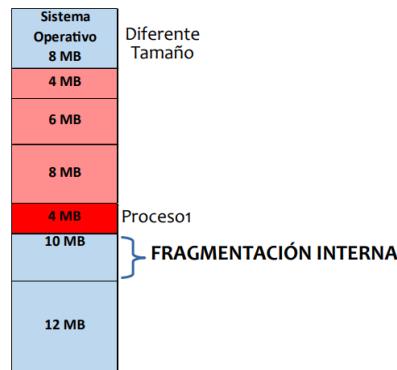
En qué partición ubicar a cada proceso: hay varias técnicas pero siempre se trata de cargar a cada proceso en la partición más chica donde entre.

- Una opción: tener una lista por cada partición y cada proceso nuevo espera en su lista más óptima.

Desventajas: puedo tener muchos procesos chiquitos esperando para usar la partición más pequeña mientras que las demás están libres.

- Otra opción: tener una única lista de procesos nuevos y que se vayan ordenando en las particiones que están libres.

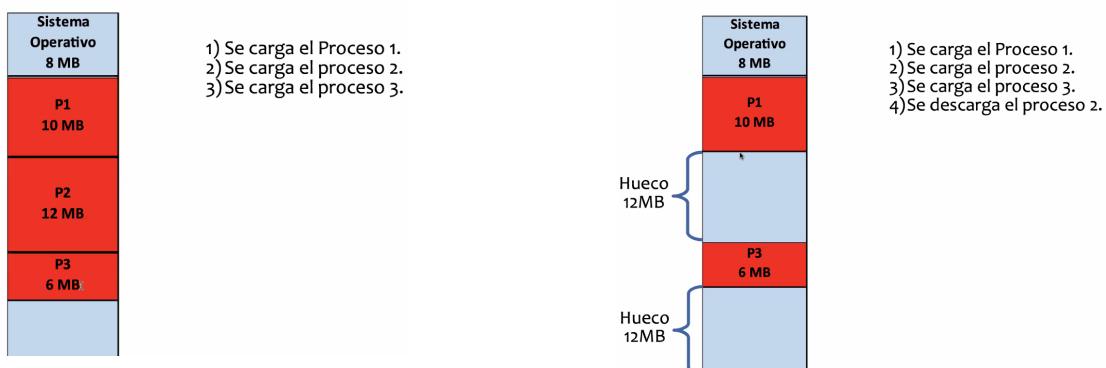
Desventajas: puedo terminar cargando programas muy pequeños en particiones muy grandes, no haciendo un buen uso del espacio. Esto se llama fragmentación interna.



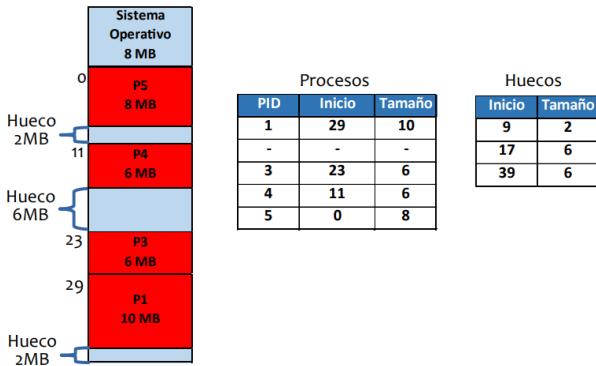
El particionamiento fijo también puede tener fragmentación externa. Supongamos el siguiente caso: quiero ubicar en la memoria un proceso de 16 MB y en memoria tengo disponible una partición de 12 MB y otra de 4 MB. Técnicamente tengo 16 MB libres para ubicar al proceso, pero, al no estar de forma contigua no vamos a poder traer el proceso a memoria.

PARTICIONAMIENTO DINÁMICO

Definición: las particiones no están predefinidas. Se van creando a demanda (si hay memoria disponible a cada proceso se le asigna exactamente la memoria que necesita). Tampoco hay un número fijo de particiones sino que hay tantas según se necesiten en un momento determinado. Cabe recalcar que este método no se lleva bien con procesos que crezcan en tiempo de ejecución.



Podemos ver que al abandonar la memoria los procesos van dejando huecos que luego los ocuparán uno o más procesos que quepan ahí. Esto se llama fragmentación externa (tener espacios libres, pero no juntos. No es espacio continuo por lo que quizás sí hay el suficiente lugar para agregar a un nuevo proceso pero, al no ser contiguo, no se podrá agregar). El SO también manejará una tabla de huecos para saber cuáles son los lugares disponibles.



Esto se soluciona mediante compactación, que compacta todos esos espacios vacíos de memoria en uno. Tiene un costo asociado y es el overhead: el SO necesita mover a todos los procesos de un lugar a otro, por lo que los procesos no podrán estar ejecutando en ese momento.

Ventajas:

- No hay un grado de multiprogramación limitado.
- No sufre fragmentación interna.

Desventajas:

- Sufre fragmentación externa.

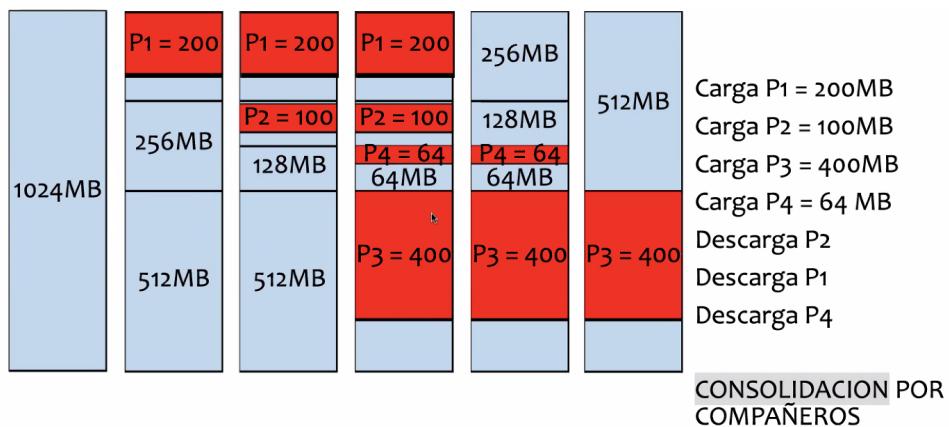
Algoritmos de ubicación: ¿dónde ubico al proceso si tengo muchas opciones de distinto tamaño? Vamos a ver varios algoritmos.

- Primer ajuste: busca el primer hueco disponible desde el comienzo de la memoria.
- Siguiente ajuste: busca el primer hueco disponible desde la posición de la última asignación.
- Mejor ajuste: busca el hueco más chico donde entre el proceso. Es el que me crea peor fragmentación externa.
- Peor ajuste: busca el hueco más grande donde entre el proceso. Es el que menos fragmentación externa genera.

BUDDY SYSTEM (DESCOMPOSICIÓN BINARIA)

Definición: es una combinación de los dos anteriores y compensa las desventajas de cada uno. Se asigna a los procesos tamaños de memoria que son potencias de dos (2^n). La memoria asignada es según el tamaño del proceso y se redondea a la siguiente potencia de 2.

Es decir, se generan particiones dinámicas dividiendo por dos y se asigna la más chica posible en la cual entra el proceso.



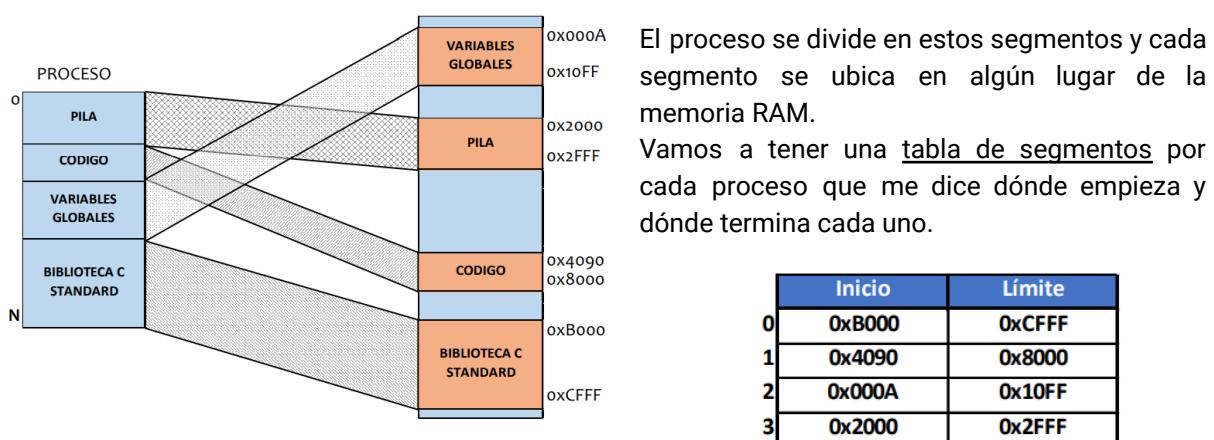
Cuando se van descargando los procesos, la memoria "vuelve a su estado inicial". Los bloques que eran compañeros (es decir que se habían generado de la misma división) y se habían dividido ahora vuelven a unirse.

Hay fragmentación interna y externa.

SEGMENTACIÓN

Definición: en los métodos anteriores era necesario que el proceso estuviera en la memoria en forma contigua. En la segmentación la idea es dividir al proceso en varios segmentos. Es decir, se puede guardar un proceso en diferentes lugares de la memoria. Si no hay lugar para cargar alguno de los segmentos el programa no podrá ejecutar. Deben estar todos sus segmentos en RAM. En resumen:

- El proceso no necesita estar contiguo en memoria.
- El proceso se divide en segmentos de tamaño variable.
- Típicamente cada segmento representa una parte del proceso desde la visión del programador:
 - Código.
 - Pila.
 - Datos.
 - Biblioteca.
 - Heap.



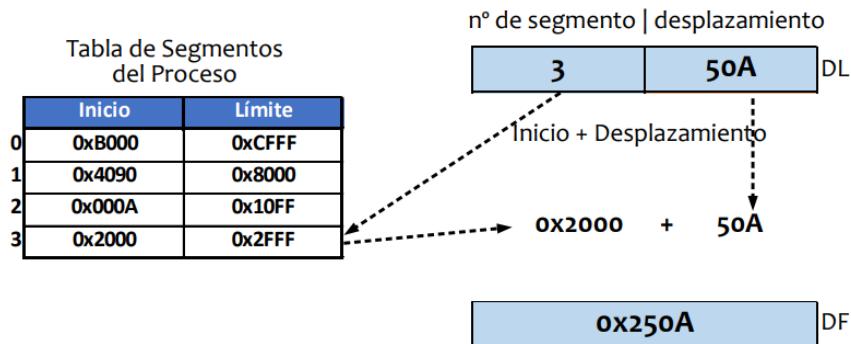
Ventajas:

- No hay fragmentación interna porque cada segmento va a tener asignado el espacio que realmente necesita.

Desventajas:

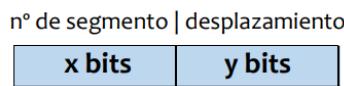
- Hay fragmentación externa porque se generan huecos pero la “ventaja” es que hay menos fragmentación externa que en el particionamiento dinámico porque ahora no tengo que ubicar procesos enteros.

Dirección lógica a física: las direcciones lógicas están formadas por una tupla: el número de segmento y el desplazamiento.



Con el número de segmento (en este caso 3), el hw va a la tabla de segmentos del proceso (que se encuentra en la memoria ram, apuntada por un registro) y busca al segmento. Ve que la posición de inicio del segmento 3 es 0x2000. A eso le sumo el desplazamiento (50A) y obtengo la dirección física. Obviamente se realiza la validación para ver si la dirección obtenida es menor al límite del segmento y de no ser así se produce una interrupción.

Estas traducciones se hacen en tiempo de ejecución. Siempre y cuando las tablas de segmentos estén bien actualizadas, todo va a funcionar bien (si muevo un segmento de lugar, tengo que actualizar las tablas).



Cantidad máxima de segmentos por proceso = 2^x .

Tamaño máximo de segmento = 2^y .

Medidas de protección:

- Hay una validación, que es que la dirección física que nos da debe ser menor que el límite que está en la tabla de segmentos, sino se produce una interrupción (Segmentation Fault).
- A cada segmento se le asignan permisos: puede ser leído (R), escrito (W) o ejecutado (X). En general:
 - Pila: RW
 - Código: X
 - Datos: RW
 - Bibliotecas: X
 - Heap: RW

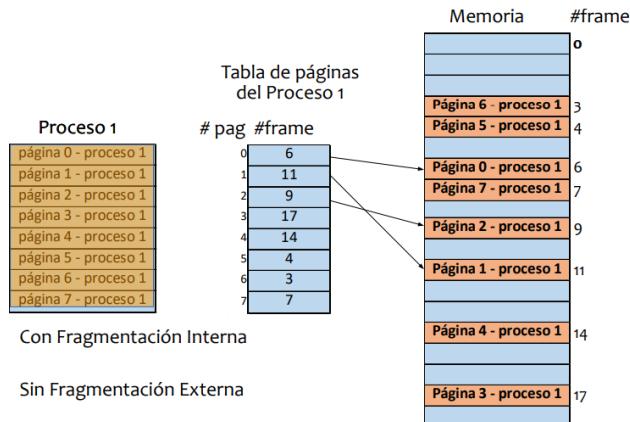
	Inicio	Límite	Protección
0	0x8000	0xCFFF	R
1	0x4090	0x8000	RX
2	0x000A	0x10FF	RW
3	0x2000	0x2FFF	RW

PAGINACIÓN

Definición: el proceso se divide en pequeñas porciones (páginas). Estas páginas no se dividen según el código, datos, heap, etc. Se divide todo en tamaños iguales, sin criterio alguno. Por su parte, la memoria RAM se encuentra dividida en marcos o frames, de igual tamaño que las páginas.

Entonces:

- El proceso no necesita estar contiguo en memoria.
- Los procesos se dividen en páginas y la memoria en frames.
- Esos frames tienen el mismo tamaño que las páginas. Esto permite que cualquier página pueda ubicarse en cualquier lado de la memoria RAM.



Se necesita tener una Tabla de Páginas (una por cada proceso) que me indica en qué posición real de la memoria ubique a cada página. Las páginas no tienen que estar en orden o continuas.

Ventajas:

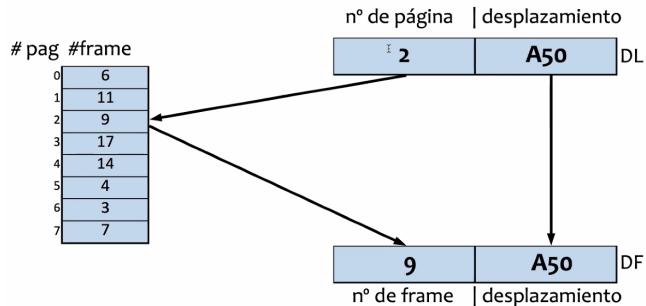
- No hay fragmentación externa porque cualquier página puede caber en cualquier frame.

Desventajas:

- Tiene un poco de fragmentación interna, porque asigna páginas de tamaño fijo (pero poco, porque los tamaños son chicos). Como mucho tendríamos fragmentación interna en la última página del proceso. La fragmentación interna máxima que tendremos es el tamaño de la página - 1 byte.

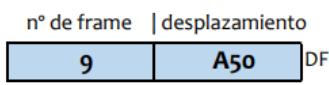
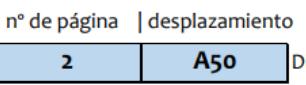
Dirección lógica a física: es muy similar al caso anterior. Parto de una dirección que tiene el nº de página y el desplazamiento. En este caso el nº de página es 2. Me fijo en la Tabla de Páginas a qué frame corresponde. Corresponde al 9. Entonces, sé que mi página está en el frame 9 + el desplazamiento (A50). Pero, el frame 9+A50, ¿qué posición de la memoria es? Bueno, tengo que hacer 9 * el tamaño de la página + el desplazamiento (A50).

Esta traducción requiere de 2 accesos a memoria (uno para consultar la Tabla de Páginas y el otro para consultar el dato que quería buscar).



Ejemplo: la dirección en binario está compuesta por el número de página y el desplazamiento en binario. Se pueden juntar todos esos números y pasarlo a decimal.

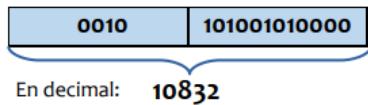
- Dirección Lógica a Física:



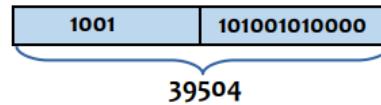
- Ejemplo:

- 16 páginas por proceso.
- Páginas de 4096 bytes.

Binario:



En decimal: 10832



39504

Fórmula: es importante saber que cuando me dan la DL en decimal puedo hacer la división entera entre DL/tam_pagina (en bytes) y el resultado me da:

- La parte entera es el número de página
- El resto es el offset

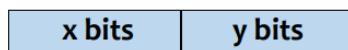
Decimal:

$$(\#página * tamaño de página) + desplazamiento$$

$$(2 * 4096) + 2640 = 10832$$

$$(9 * 4096) + 2640 = 39504$$

nº de página | desplazamiento



Cantidad máxima de páginas por proceso = 2^x .

Tamaño de la página = 2^y .

Medidas de protección:

- Dado que el programa fue dividido en páginas casi en forma aleatoria, es difícil asignar permisos dado que no está bien separado lo que es código, pila, heap. Pero igual pueden definirse estos permisos.
- Cada página tiene un bit que indica si es válida o no (que básicamente quiere decir si está en memoria o no). Vamos a verlo más adelante en memoria virtual.

# frame	Protección	Válido/Inválido
0	R	1
1	RX	1
2	RW	1
3	RW	0

Compartición (sharing): la paginación hace muy sencilla la compartición de recursos entre procesos. Solamente necesitan apuntar al mismo frame (en el ejemplo es el frame 8).

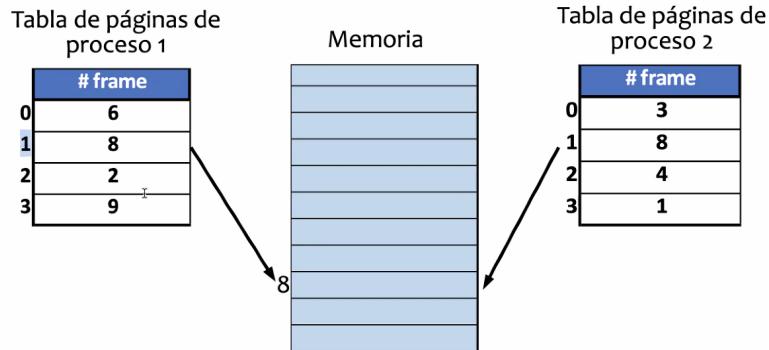
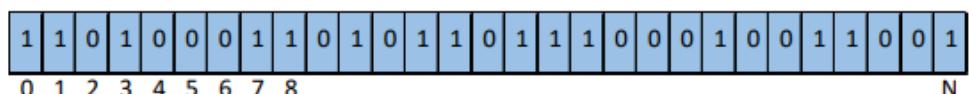
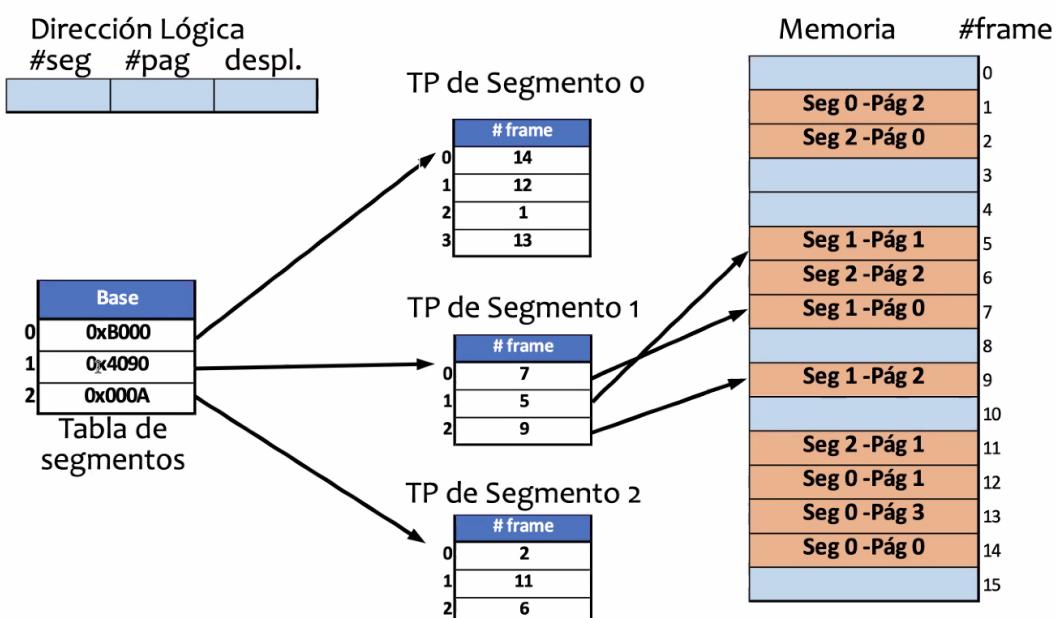


Tabla de marcos/frames libres: es necesario tener una estructura extra para poder saber qué frames de la memoria están libres y cuáles están ocupados. Lo más cómodo es utilizar un bitmap.



SEGMENTACIÓN PAGINADA

Definición: es la combinación de los dos esquemas. Combina las ventajas de ambos. Los segmentos se dividen en páginas. Tiene un poco más de overhead.



El proceso primero se divide en segmentos y después cada segmento se divide en páginas. Terminamos teniendo una tabla de páginas para cada segmento del proceso. La memoria está dividida en frames entonces puedo guardar cada página en cualquier posición disponible. Como vemos arriba a la izquierda, la dirección lógica ahora es más compleja.

Ventajas:

- No hay fragmentación externa.
- Se pueden asignar buenos permisos para garantizar la protección dado que está bien separar lo que es código, pila, heap.

Desventajas:

- Hay más fragmentación interna.

Memoria virtual

INTRODUCCIÓN

Características necesarias para memoria virtual:

- La traducción de direcciones debe realizarse en tiempo de ejecución porque en tiempo de ejecución se permite la reubicación de los procesos.
- El proceso debe estar dividido en partes (páginas o segmentos).

Motivaciones: hasta el momento teníamos la idea de que, para que un proceso pueda ejecutar, era necesario que estuviera cargado completamente en la RAM. Si fuera suspendido se lo pasaría completamente a disco y si quisiera volver a ejecutar debería volver a cargarse completamente en memoria. La realidad es que durante la ejecución de un proceso no se usan todas sus partes a la vez. Las partes que no se están ejecutando están ocupando memoria. ¿Es necesario ocupar la memoria con partes que no se usan? En la memoria real, entonces, sólo vamos a tener las páginas del proceso que necesitamos y el resto se van a mover a la memoria virtual.

MEMORIA VIRTUAL

Aclaración: todos los ejemplos que vamos a ver de memoria virtual son con paginación pero puede hacerse con segmentación pura.

Definición: dispositivo de almacenamiento secundario que puede ser direccionado como si fuese memoria real. No es la memoria RAM. En la mayoría de las computadoras es el disco. Es decir, parte del disco se va a utilizar como memoria. Nosotros podemos definir cuál es el tamaño de la memoria virtual en nuestras computadoras y también el tamaño puede ir actualizándose dinámicamente.

Por ejemplo: si tengo un proceso de 6 páginas, lo que voy a hacer es cargar esas 6 páginas en la memoria virtual (disco) en lugar de en la memoria real. A medida que necesito páginas las voy a cargar en la memoria real, pero voy a cargar solamente lo que el proceso realmente necesite para ejecutar.

Ventajas:

- Me permite correr procesos más grandes que la memoria real porque nunca lo carga completamente en la memoria real.
- Permite aumentar ampliamente el grado de multiprogramación del sistema.
- Existen menos restricciones para el programador a la hora de ponerse a pensar si tiene memoria suficiente o no. Se asume que la memoria virtual es lo suficientemente grande como para contener a la totalidad del proceso.

Memoria Virtual	
P1-pag0	0
P1-pag1	1
P1-pag2	2
P1-pag3	3
P1-pag4	4
P1-pag5	5
P3-pag0	6
P3-pag1	7
P4-pag0	8
P4-pag1	9
P2-pag0	10
P2-pag1	11
P2-pag2	12
P2-pag3	13
P6-pag0	14
P6-pag1	15
P6-pag2	16
P5-pag0	17
P5-pag1	18

Memoria Real	
P2-pag0	0
P2-pag1	1
P6-pag2	2
P1-pag2	3
P6-pag1	4
P1-pag0	5
P4-pag1	6
P3-pag0	7
P5-pag1	8
P1-pag5	9
Libre	10

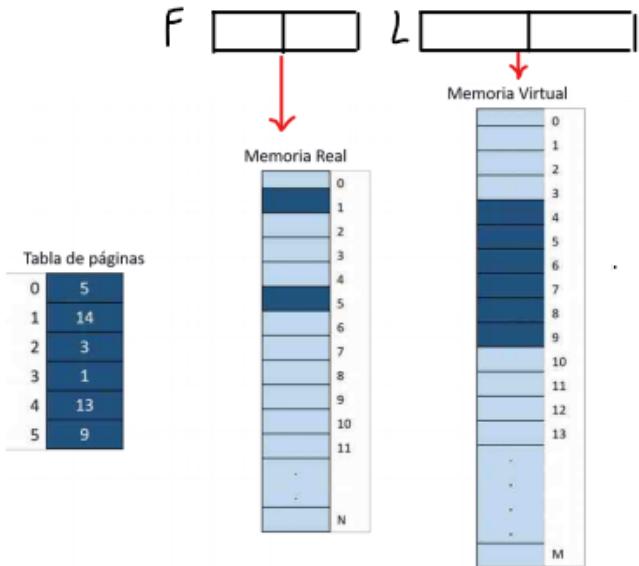
Direcciones: más arriba dijimos que la memoria virtual podía ser direccionada como si fuese memoria real, ¿qué quiere decir esto?

El espacio virtual abarca toda la dimensión de la memoria virtual (disco) y es el espacio que maneja el programador de aplicaciones. Es decir que los programas, en sus instrucciones (por ejemplo MOV

5. Oxdirección el 0xdirección va a hacer referencia a la memoria virtual), utilizan este rango de direcciones pero no la CPU que tiene acceso a memoria real, entonces es necesario traducir estas direcciones lógicas a físicas con la MMU.

Para resumir, las direcciones lógicas hacen referencia a la memoria virtual y las direcciones físicas a la memoria real. De pasar la dirección lógica a la física se encarga la MMU.

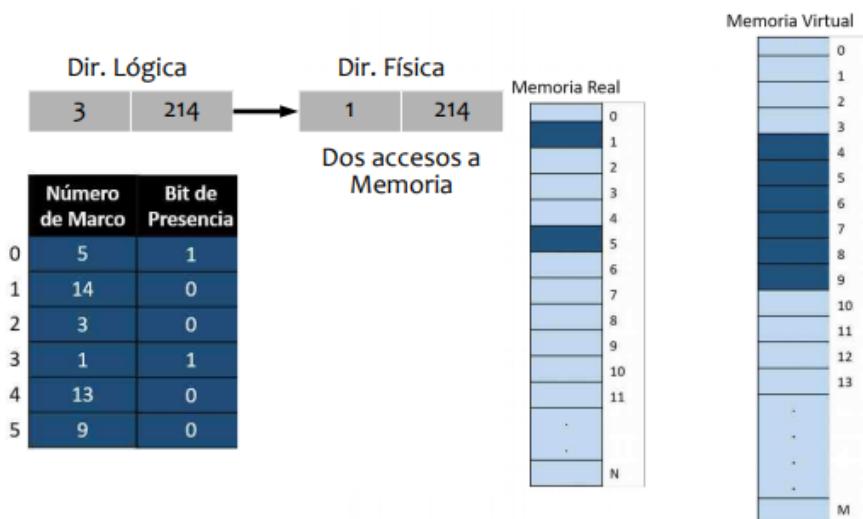
Las direcciones lógicas van a ser mucho más grandes porque tengo que poder referenciar a todos los frames de la memoria virtual (que tiene un gran tamaño) mientras que las direcciones físicas apuntan a la memoria real y son más cortas porque hay que referenciar a menos frames.



ESTRUCTURA DE TABLA DE PÁGINAS Y TRADUCCIÓN DE DIRECCIONES

Cuando el bit de presencia está en 1: supongamos que tenemos la siguiente dirección lógica: página 3, desplazamiento 214. Como siempre, primero vamos a la tabla de páginas y ahí me dice que la página 3 se encuentra en el frame 1 de la memoria real. Vemos que hay una nueva columna "Bit de presencia" que es un booleano que indica si la página está en memoria real. En este caso lo está, entonces la dirección física sería "frame 1 de la memoria real, offset 214".

Esta traducción conlleva dos accesos a memoria real: uno para consultar la tabla de páginas y otro para acceder al valor deseado. Aquí ya vemos un impacto en la performance.

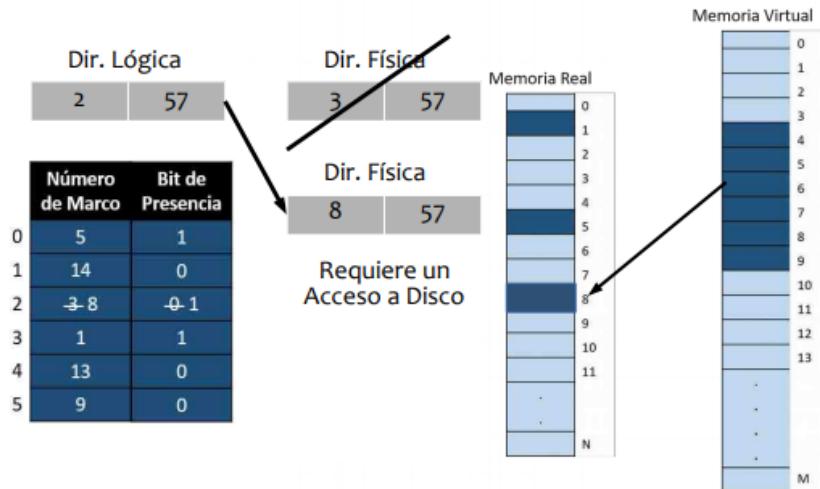


Cuando el bit de presencia está en 0: ahora supongamos la dirección lógica "página 2, offset 57". Vamos a la tabla de páginas y encontramos que la página 2 está en el frame 3 pero el bit de presencia es 0. Esto quiere decir que la página no está en la memoria real sino que está en la memoria virtual. Por lo tanto, el frame 3 que nos indica la tabla de páginas no nos interesa porque no es cierto. Vamos a tener que traer la página a memoria real y actualizar la tabla de páginas. Pero antes de ver eso se dispara una excepción especial: Page Fault o Fallo de Página.

Page Fault: es una excepción que arroja la MMU cuando un programa quiere acceder a una dirección que no se encuentra en la memoria principal actualmente (bit de presencia en 0) porque ha sido puesta en el disco (memoria virtual).

Pasos:

1. Se realiza una referencia a memoria:
 - a. Se verifica la presencia de la página en memoria. Si el bit de presencia es 0 se produce un Page Fault (acceso a memoria para leer la tabla de páginas).
2. Se dispara el Page Fault.
 - a. El SO bloquea al proceso.
 - b. Mientras tanto otro proceso puede utilizar la CPU.
3. Mientras el proceso está bloqueado hay que traer la página que solicitó a memoria:
 - a. Se va a buscar la página solicitada a memoria virtual (acceso a disco).
 - b. Se ubica a la página en cualquier marco libre de la memoria real (acceso a memoria para escribirla).
4. Se produce una interrupción para que el SO tome el control
 - a. El SO actualiza la tabla de páginas con el nuevo frame y pone el bit de presencia en 1 (acceso a memoria)
 - b. Desbloquea al proceso (cambia su estado a Ready).
5. Se ejecuta nuevamente la instrucción que provocó el Page Fault (esto provocará los 2 accesos a memoria que conlleva cuando el bit de presencia está en 1).

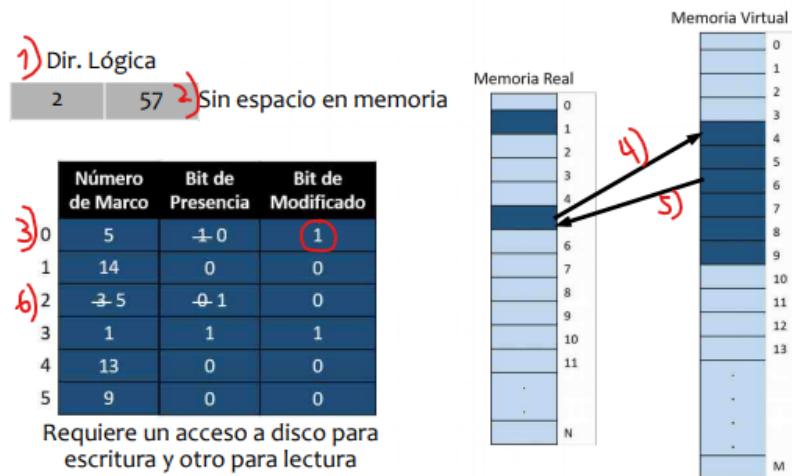


Cuando el bit de presencia está en 0 y no hay más espacio en memoria: en el paso 3b dice que hay que ubicar a la página en algún frame libre de la memoria. ¿Qué pasa si no hay ninguno?

1. Hay que mover alguna página del proceso que esté en la memoria real a la memoria virtual. (en realidad se "actualiza" la que está en disco si su bit de modificado está en 1. Si no fue modificada solamente se cambia su bit de presencia a 0).
2. Se trae a memoria real la página solicitada (pero no se borra de la memoria virtual) y se coloca en el frame que quedó libre en el paso 1. Se actualiza la Tabla de Páginas cambiando su frame y setando su bit de presencia en 1.
3. La página pudo haber sido solicitada para ser leída o escrita. Es importante saber para qué porque si la página es modificada mientras está en la memoria real, después vamos a tener que actualizar la que está en disco. Si la página únicamente es leída entonces no hará falta. Para registrar si una página es modificada o no se agrega una nueva columna a la Tabla de Páginas llamada "bit de modificado". Por ejemplo:

Número de Marco	Bit de Presencia	Bit de Modificado
0 5	1 0	1
1 14	0	0
2 3 5	0 1	0
3 1	1	1
4 13	0	0
5 9	0	0

Ejemplo: tenemos un proceso que ya tiene 2 marcos en memoria y quiere traer otro. Entonces el marco 5 lo swappeamos y en su lugar colocamos el nuevo marco.



Eficiencia: vemos que el uso de memoria virtual requiere de muchos accesos a disco y a memoria (más en este último caso mencionado). Comparemos las ventajas (ya mencionadas antes) y las desventajas:

Ventajas:

- Me permite correr procesos más grandes que la memoria real.
- Permite aumentar ampliamente el grado de multiprogramación del sistema.
- Existen menos restricciones para el programador a la hora de ponerse a pensar si tiene memoria suficiente o no. Se asume que la memoria virtual es lo suficientemente grande.

Desventajas:

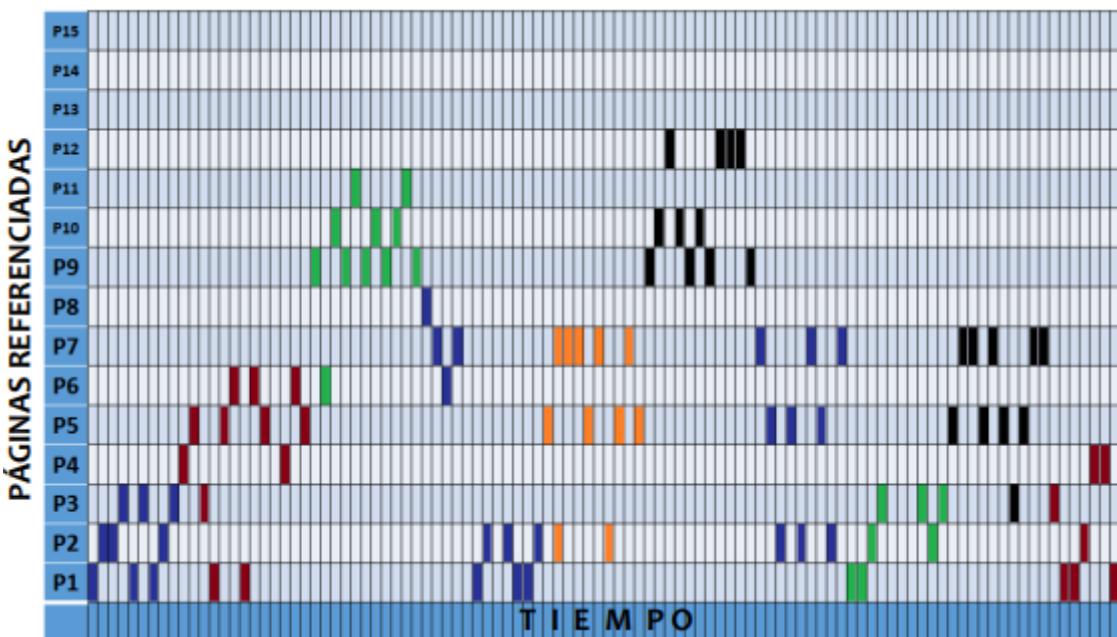
- No es eficiente.
- Requiere más accesos a memoria.
- Requiere más accesos a disco.
- No mejora el rendimiento para la ejecución del proceso por dos razones:
 - Porque el mecanismo de búsqueda de páginas demora.
 - Porque puede producirse Page Fault y bloquearse al proceso.

Conclusión: entonces, con todas esas desventajas, ¿la memoria virtual es una porquería? No, no lo es porque existe el principio de localidad que hace que este bajo rendimiento "no se note tanto".

Principio de localidad o proximidad: establece que en un intervalo de tiempo sólo se usan unas páginas de forma activa. A este conjunto se lo denomina "localidad".

Por ejemplo, supongamos que estamos corriendo un juego que tiene 10 niveles y actualmente estamos en el nivel 2. Durante el transcurso de ese nivel sólo vamos a usar las páginas de ese nivel. Si todas esas páginas están cargadas en la memoria no se van a producir Fallos de Página.

Pero, una vez que pasemos al nivel 3 lo más probable es que necesitemos cargar todas las páginas del nivel 3 a memoria real y pasar las del nivel 2 a memoria virtual. Entonces sólo en ese momento se producirían los Fallos de Página pero no durante el transcurso del nivel.



ESTRUCTURAS DE TABLAS DE PÁGINAS

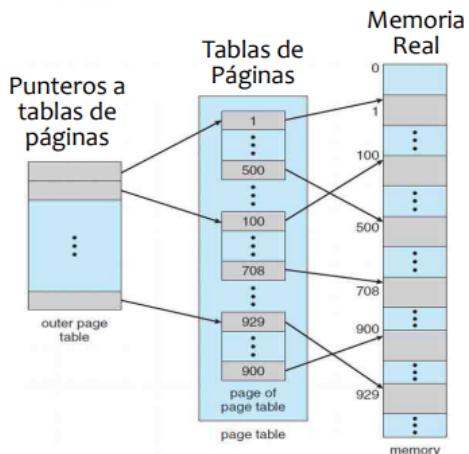
Introducción: quien define el tamaño de las páginas es el SO (suelen tener tamaño 2^n). Actualmente los procesos suelen ser muy grandes y las páginas muy chicas, por lo que cada proceso tiene asociada una Tabla de Páginas bastante extensa. Lo que termina pasando es que ocupa mucho lugar de la memoria tener una tabla de estas características por cada proceso.

Paginación jerárquica o por niveles: lo que se hace ahora es paginar la Tabla de Páginas, es decir, se separa a la Tabla de Páginas en páginas del mismo tamaño. Se agrega una nueva tabla que contiene punteros a las tablas. Como pasaba con los procesos paginados, en memoria real sólo van a permanecer las páginas que necesitamos.

Ahora la dirección lógica no me sirve más que sea "número de página, offset", sino que tiene que ser:

10 bits	42 bits	12 bits
Puntero a Tabla	Número de Página	Desplazamiento

En este ejemplo la paginación es de 2 niveles pero podría ser de más. La cantidad de accesos a memoria necesarios es la cantidad de niveles + 1.



La tabla de páginas, al estar partida, no está toda en memoria. Entonces, podría llegar a tener un page fault porque una parte de la tabla de páginas no está en memoria. Ni siquiera pude saber si la página estaba en memoria o no, porque cuando quise ir a la tabla, el pedazo de tabla que quería leer no estaba.

El peor caso que puede darse es que la página no esté en memoria real y la tabla tampoco. En este caso se dispararán dos page faults. Uno para indicar que la tabla no está en memoria y el otro (una vez que la tabla fue cargada en memoria y consultada) cuando, al consultar la tabla nos damos cuenta que la página tampoco está.

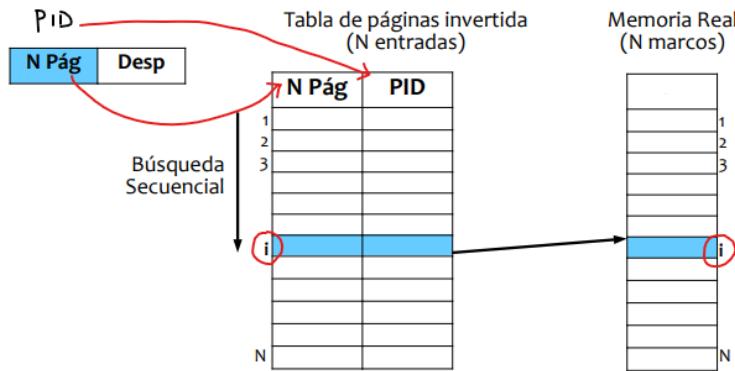
La primera tabla (outer page table) tiene que estar siempre en RAM.

Tabla de páginas invertida: otra estructura que puede usarse es la tabla de páginas invertida y consiste en tener una única tabla de páginas para todos los procesos.

- Esta tabla tendrá tantas entradas como frames en la memoria.
 - Se dice “invertida” porque está indexada por frame, no por número de página. La tabla indica la página de qué proceso se encuentra en cada frame.

N Pág	PID
1	
2	
3	
i	
N	

Ahora, ¿cómo se pasa de la dirección lógica a la física? Tengo que buscar la página que quiero en la tabla y obtener en qué frame se encuentra. La búsqueda es secuencial y el índice en el que se encuentra a la página es el #frame.



Ventajas:

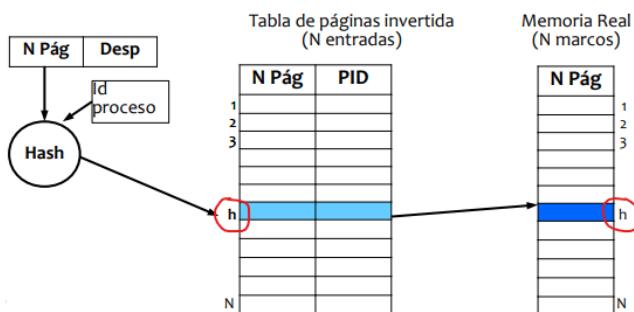
- Tengo una tabla de tamaño fijo.

Desventajas:

- Es muy lento por la búsqueda secuencial que genera mucho overhead. Además, cada vez que encuentra una página con el número correcto tiene que preguntar por el PID.
- En la paginación simple era muy sencillo lograr que varios procesos compartieran recursos (ambos procesos simplemente apuntaban al mismo frame). Ahora, al tener una única tabla y estar indexada por frame, esto no es posible. Tampoco es imposible pero para lograrlo será necesario incorporar nuevas estructuras además de la tabla invertida.
- Si vemos, la tabla no tiene un bit de presencia. Por lo que si estamos buscando una página que no está en memoria, tendríamos que iterar por toda la tabla. El page fault se produce cuando recorrió toda la estructura y no encontró la página que estábamos buscando.

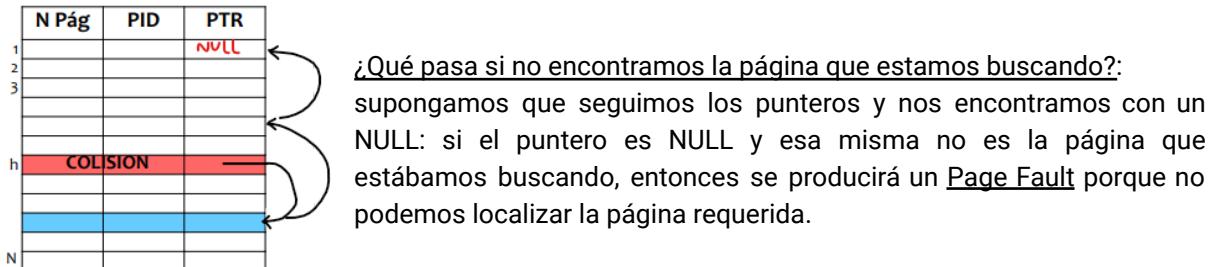
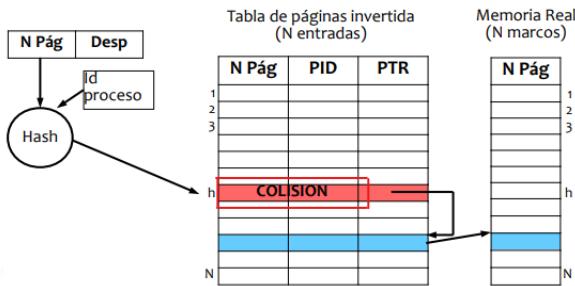
Para solucionar este problema de la lentitud se utiliza el hash.

Hash: es una función que devuelve un número a partir del número de página y el PID. Ese número va a ser directamente el número de frame. Es decir que el uso de la función de hash se va a utilizar en reemplazo de la búsqueda secuencial dado que es mucho más rápido.



Problema de las funciones de hash: puede retornar el mismo valor a partir de distintas entradas. A esto se lo conoce como "colisión". Entonces, la primera entrada guardará un puntero (PTR) hacia la próxima. Supongamos que "P1, página 3" y "P5, página 2" generan el mismo hash "h": si h ya está ocupado por "P1, página 3", guardará un puntero a "P5, página 2".

Cuando nosotros estamos buscando una página, primero vamos a la entrada h que retorne el hash. Vamos a consultar los primeros dos valores (el número de página y el PID) para ver si realmente es la página que necesitamos. Si lo es entonces estará en el frame h de la memoria real pero si no lo es deberemos seguir buscando. Para esto usaremos la columna PTR.



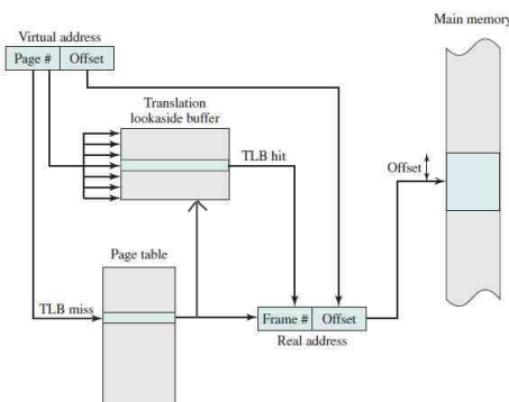
TRANSLATION LOOKASIDE BUFFER - BUFFER DE TRADUCCIÓN ANTICIPADA (TLB)

Definición: es una memoria caché de acceso por contenido administrada por la MMU cuyo objetivo es mejorar el rendimiento de la traducción de direcciones. Lo que hace que las traducciones sean lentas son los accesos a memoria. Entonces, lo que la TLB quiere hacer es reducir la cantidad de accesos a memoria. Para esto, la TLB lo que hace es guardar etiquetas con un dato asociado. Como etiqueta actúa el número de página y como dato asociado el número de frame correspondiente. Entonces, cada vez que se realiza una traducción se guarda el resultado en la TLB. Luego, cada vez que se requiere traducir una dirección lógica, primero se consulta la TLB para ver si ya sabe el número de frame, para no tener que pedírselo a memoria. En caso contrario, se inicia la traducción y se guarda el resultado en la TLB, siendo la única desventaja el tiempo que se perdió en consultar la TLB sólo para descubrir que el resultado no se encontraba allí.

Cuando decimos que la TLB es una caché de acceso por contenido, estamos diciendo que se compone de una etiqueta y un dato asociado. No está indexada. Cuando se quiere obtener una información, se suministra un valor, el cual es comparado con el campo etiqueta de todas las entradas. En el caso en que el valor suministrado coincide con alguna etiqueta, el dato asociado a la misma será suministrado como salida/respuesta.

Caso contrario, cuando no se encuentra el valor, hay 2 posibilidades:

1. La página está presente en memoria y solamente se necesita crear la entrada de la TLB.
2. La página no está presente en memoria y se necesita generar un Page Fault.



Ventajas:

- Es muy rápida.
- Reduce la cantidad de accesos a memoria.

Desventajas:

- Suele tener muy pocas entradas.

ASID: la TLB tradicional contiene la información del proceso que está ejecutando actualmente y se borra cuando el proceso deja de ejecutar. Pero, hay otro tipo de estructura de la TLB donde se le agrega un ASID (identificador de espacio de direcciones), que sería un PID y que evita tener que borrarla.

Aclaración final sobre memoria virtual: en la realidad, no se suspenden procesos cuando tenemos memoria virtual. Los estados de suspendido/listo y suspendido/bloqueado no se usan, no existen. Ya no hay necesidad de suspender procesos. El proceso siempre está activo, con una parte en ram y otra en disco. No puede ocurrir que todas las páginas de un proceso estén en memoria virtual, por lo menos una está en memoria real

PREGUNTAS DE PARCIAL

- Explique al menos dos maneras que tiene el hardware para mejorar la eficiencia del sistema de gestión de memoria. ¿Qué ventajas implica cada una?**

MMU, TLB, disco de swap.

- La corrupción total de la tabla de páginas de un proceso no sería un impedimento para que el mismo siga ejecutando, siempre y cuando el sistema operativo utilice memoria virtual, el proceso no haya escrito páginas y se disponga de espacio libre en memoria RAM.**

Verdadero. El SO podría frenar la ejecución del proceso, crear una nueva tabla de páginas y a partir de ahí comenzar a cargar las páginas que solicite en nuevos frames. Es necesario el sobrante de memoria RAM porque las páginas cargadas previamente serían imposibles de detectar y por ende imposibles de liberar. También es necesario que no hayan sido modificadas previamente, con lo que la página guardada en memoria virtual estaría actualizada.

Diseño del SO para memoria virtual

INTRODUCCIÓN

Introducción: vamos a ver cuáles son las estrategias, algoritmos, métodos, etc que utiliza el SO para tratar de mejorar el rendimiento del uso de la memoria virtual. Cuando vimos memoria virtual llegamos a la conclusión de que no es demasiado eficiente pero que gracias al "principio de localidad" puede llegar a tener un mejor rendimiento. Luego, con el uso de la MMU puede reducirse el overhead del SO al realizar las traducciones.

Lo que vamos a ver ahora son ciertas "políticas" que utiliza el SO para mejorar el rendimiento de la memoria virtual. Concretamente, todas estas políticas tienen el objetivo de reducir la cantidad de Page Faults porque es lo que más demora genera (además de que bloquea a los procesos para ir a buscar la página faltante).

Políticas de:

- a. Recuperación.
- b. Ubicación.
- c. Reemplazo o sustitución.
- d. Conjunto residente.
- e. Limpieza.

POLÍTICAS

Recuperación: está asociada al principio de localidad. Este principio indica que "lo que acabo de usar es probable que lo vuelva a usar". Esto implica que una vez que terminemos de usar ciertos recursos dejaremos de usarlos y pasaremos a usar otros, los cuales debemos cargar a la memoria. Y, ¿en qué momento tengo que traer las páginas de memoria virtual a memoria real? Hay 2 opciones:

- Paginación bajo demanda: las traigo a medida que las necesito. Si la necesito y no está, espero a que se cargue.
- Paginación adelantada (prepaging): básicamente lo que hace es: "si me pediste la página 0 te traigo la 0, 1, 2 y 3 porque ya estoy por ahí". Esto conlleva un gran riesgo o una gran ventaja, dependiendo de si uso o no esas páginas que trajo de más.

Estas dos técnicas suelen usarse en conjunto. Puede ser que apenas comience a ejecutar un proceso se use prepaging y que luego sea bajo demanda.

Ubicación: tiene que ver solamente cuando usamos segmentación pura/simple con memoria virtual. En los ejemplos que vimos de memoria virtual usábamos páginas pero pueden usarse segmentos. Los segmentos tienen diferentes tamaños y para ubicarlos en la memoria de la forma más eficiente posible usamos los siguientes algoritmos:

- Primer ajuste
- Siguiente ajuste
- Mejor ajuste
- Peor ajuste

Cuando se usa paginación no tiene sentido analizar esto dado que la memoria está dividida en frames que son todos iguales. Entonces un frame libre es igual a otro frame libre, no vale la pena analizar en qué frame ubicar una página.

Reemplazo o sustitución: ya habíamos mencionado la situación en la que necesito traer una página a memoria y no es posible dado que ya no tengo más memoria disponible. Si fue el P1 el que hizo la

solicitud hay una de sus páginas que debe pasar de memoria real a virtual. La página a reemplazar no se elige de forma aleatoria. Para hacer este tipo de reemplazo hay 2 técnicas principales:

- Bloqueo de marcos: se refiere a agregar un bit de bloqueo a uno de los marcos de la memoria. Ese bit indica que esa página no la puedo reemplazar. No tendría sentido que todas las páginas tuvieran este bit en 1 ya que no estaríamos haciendo uso de la memoria virtual.
- Algoritmos/políticas de reemplazo o sustitución: permiten elegir cuál es la siguiente página a reemplazar (llamada "víctima"). Todos menos el FIFO tratan de reducir el page fault. El criterio para medir/comparar estos algoritmos es la cantidad de fallos de página que generan.

Todos estos algoritmos se ejecutan SOLAMENTE cuando se cumplen las siguientes condiciones:

- Ocurrió un page fault
- Es necesario seleccionar una víctima. Es decir, si hay lugar en memoria entonces no.

Dicho esto, los algoritmos son:

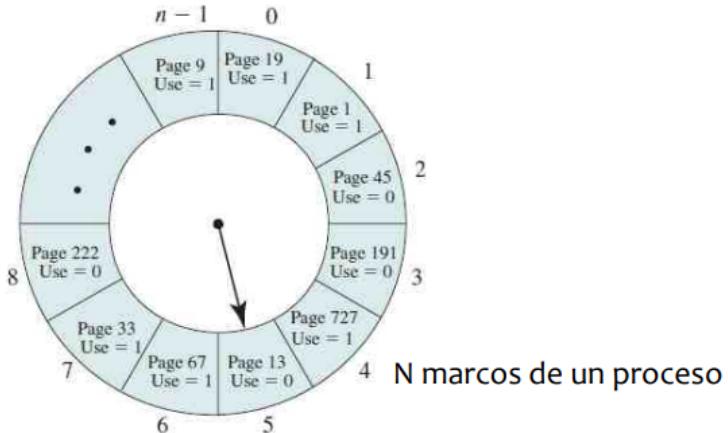
- Óptimo: se elige la página a la que se realizará una referencia en el futuro más lejano. Es decir, veo el futuro y me fijo cuál de las páginas actuales voy a usar por último y la reemplazo. No puede implementarse. Sólo "existe" a modo de comparación para medir a los demás algoritmos.
- FIFO: se elige la página que hace más tiempo está en memoria. No requiere un formato distinto de la tabla de páginas. Solamente necesita un puntero a la próxima página que será reemplazada.
- LRU (least recently used): se elige a la página que hace más tiempo no es referenciada. Para implementar este algoritmo es necesario que en la Tabla de Páginas esté registrado el momento exacto en el que fue usada cada página por última vez. Cada vez que se corra el algoritmo se van a tomar estos tiempos y compararse. Termina siendo el algoritmo que más overhead genera. La tabla de páginas necesita los siguientes datos:

#Marco	Bit de Presencia	Bit de Modificado	Instante de referencia
--------	------------------	-------------------	------------------------

- Algoritmo del reloj o Segunda oportunidad (clock): es una combinación entre FIFO y LRU pero se agrega un bit: "bit de uso". Este bit me indica si la página se usó hace mucho tiempo o no. Cada vez que se carga una página en la memoria real o cada vez que se referencia se pone el bit en 1. Debido a ciertas circunstancias ese bit puede cambiarse a 0, lo que indica que puede ser una posible víctima.

Implementación: la Tabla de Páginas va a tener una nueva columna para el bit de uso. Además, vamos a tener un puntero apuntando a un frame de la memoria. Cuando quiera encontrar una víctima el puntero va a mirar el bit de uso. Si está en 0 va a seleccionar a esa página como víctima y si está en 1 van a suceder 2 cosas:

- Se va a poner su bit de uso en 0.
- El puntero va a moverse a la siguiente posición.



Es importante aclarar que esta estructura circular que estamos viendo no es la tabla de páginas. Es una estructura independiente.

- Clock Modificado: este algoritmo en sí no disminuye la cantidad de page faults, solamente trata de ver qué página debe ser la víctima (que será una página que no fue modificada) para que no haya que escribir en disco.

Recordando lo básico: un page fault sin lugar en memoria real implica que una página de memoria virtual debe ir a memoria real y viceversa, una de memoria real debe ir a virtual. Acá es donde entra en juego el bit de modificado porque siempre nos será conveniente elegir como víctima a una página que no haya sido modificada (así no hay que ni siquiera llevarla a la memoria virtual para actualizar los cambios).

Este algoritmo requiere un puntero al siguiente marco a analizar, el bit de uso y el bit de modificado. Se pueden dar las siguientes situaciones con los bit de uso y modificado:

Bit de uso	Bit de modificado	Significado
0	0	No accedido recientemente y no modificado
1	0	Accedido recientemente y no modificado
0	1	No accedido recientemente pero modificado
1	1	Accedido recientemente y modificado

Orden de filas de mejor a peor caso: 1, 3, 2, 4

Algoritmo:

1. Recorre los marcos y selecciona el primero con bits 0 - 0
2. Si no encuentra, recorre los marcos y selecciona el primero con 0 - 1. A medida que recorre modifica el bit de uso, cambiándolo de 1 a 0.
3. Repite el 1er paso y, si es necesario, el 2do.

Como mucho son cuatro pasos: hace 1, 2 y luego vuelve a hacer 1, 2. No se queda iterando infinitamente.

Conjunto residente: tengo mi proceso por completo en la memoria virtual y en la memoria real tengo el espacio de direcciones que le fue asignado al proceso. Mi conjunto residente es aquel conjunto de páginas que se encuentran en la memoria real.

Hay diferentes maneras de gestionar ese conjunto residente porque a un proceso se le pueden asignar marcos fijos o variables. Si de entrada la asignación es fija (se le asigna una cantidad máxima de páginas a cada proceso: "todo proceso va a tener hasta 10 páginas en memoria RAM, no más") entonces el conjunto residente va a tener siempre el mismo tamaño máximo. Si es variable, lo contrario.

La cantidad es variable cuando se busca asignar a un proceso la cantidad de marcos óptimas para que produzca la menor cantidad de page faults.

El SO tiene dos criterios para decidir sacar una página o no de memoria RAM y sustituirla por otra:

- Cuando no tengo más frames libres y necesito cargar una nueva página.
- Cuando la memoria RAM no está llena pero de igual manera decide reemplazar un frame por una página que trae de disco. Esto ocurre en la asignación fija, porque evita que un proceso tenga más de una cierta cantidad de páginas en memoria.

Con respecto al reemplazo, si es local, al momento de elegir una víctima se elige una página del proceso en cuestión. Si es global se puede seleccionar una página de cualquier proceso.

	Reemplazo Local	Reemplazo Global
Asignación Fija	<ul style="list-style-type: none"> • Número de marcos asignados a un proceso no varía. • Páginas a cambiar son del mismo proceso. 	<ul style="list-style-type: none"> • No es posible.
Asignación Variable	<ul style="list-style-type: none"> • Número de marcos asignados a un proceso puede cambiar. • Páginas a cambiar son del mismo proceso. 	<ul style="list-style-type: none"> • Las páginas a reemplazar se eligen entre todos los marcos. • Cada reemplazo puede afectar a otro proceso.

¿Por qué no se puede implementar reemplazo global en asignación fija? En el caso que el proceso llegue al número de páginas máximas en memoria y solicite una nueva, no va a poder reemplazarla por una perteneciente a otro proceso, porque el proceso pasaría a tener más páginas en memoria que las permitidas. Si el proceso todavía no llegó a su máximo de páginas en memoria, ahí sí sería posible aplicar reemplazo global en asignación fija.

En resumen, no es posible cuando los procesos llegan al máximo de páginas permitidas cargadas en memoria.

Se tiende a pensar que mientras más grande es el conjunto residente, menor va a ser la existencia de page faults. Pero existe la anomalía de Belady que dice que cuando se usa FIFO ocurre lo contrario.

Anomalía de Belady: se da por la ampliación de memoria, solamente en FIFO. La anomalía es que ocurren más page faults cuando se aumenta la memoria.

Limpieza: liberar páginas de los procesos en memoria real.

- Limpieza bajo demanda: no limpia, las páginas se van cuando las reemplaza.
- Limpieza adelantada: el SO decide sacar páginas que los procesos no están usando para liberar la memoria RAM, para que quede libre para futuras peticiones.

CONSIDERACIONES

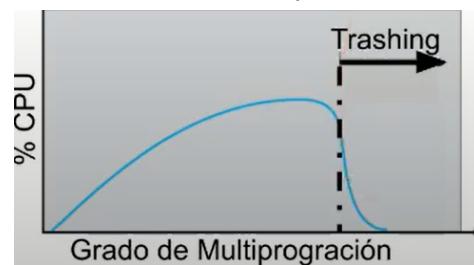
Trashing/sobrepaginación: situación donde se producen demasiados fallos de página. Se da cuando

fracasa la memoria virtual, porque todo el tiempo se piden páginas que no están.

El trashing puede ser causado por muchas cuestiones:

- Un algoritmo de reemplazo que hace mal su trabajo (saca páginas que son necesitadas próximamente).
- Cargo muchos procesos. Para ejecutar, estos procesos van a necesitar tener páginas cargadas en memoria real, y como cargué muchos procesos en RAM, cada proceso tiene asignados pocos marcos en memoria. Si los procesos tienen muy pocos marcos asignados, eso también va a influir a la hora de que se produzcan fallos de página. Esto es provocado por el aumento del grado de multiprogramación.

El trashing genera una baja tasa de uso de la CPU. Los procesos se bloquean y no pueden ejecutar.



File System

INTRODUCCIÓN

¿Qué es un file system?: es un conjunto de estructuras y funciones (syscalls) que nos van a permitir poder guardar, modificar, borrar y realizar más operaciones con archivos. Pero, ¿qué es un archivo? es simplemente un conjunto de bytes que utilizamos para alojar información. El file system y sus funciones son los que nos permiten agrupar a estos bytes para así poder identificarlos como un archivo y manipularlo.

Objetivos:

- Almacenar datos y operar con ellos.
- Soporte para varios usuarios. Implica protección.
- Minimizar la posibilidad de pérdida de datos.
- Maximizar el desempeño del sistema
 - SO: administrar espacio en disco y aprovecharlo.
 - Usuario: tiempo de respuesta.
- Soporte para distintos tipos de dispositivos.
- Garantizar la integridad o coherencia de los datos.

ARCHIVOS

Definición: conjunto de datos relacionados e identificados. Tienen ciertos atributos y, por lo general, permanecen en memoria no volátil.

Atributos de un archivo:

- Nombre: incluye su extensión. En la gran mayoría de los sistemas, el nombre y la ubicación son sinónimos.
- Identificador.
- Tipo: no tiene nada que ver con la extensión del archivo. Solamente hace referencia a si es de tipo directorio o del sistema. Lo vamos a ver más adelante.
- Ubicación: no hace referencia al path donde reside el archivo sino a dónde/cómo está guardado ese archivo internamente. Si está guardado en forma contigua se podrá acceder a todas las partes del archivo conociendo su base y tamaño. Pero, si no está guardado de forma contigua debemos tener una forma de poder hallar todos los bytes que forman a ese archivo.
- Tamaño: incluye el tamaño del contenido del archivo + el tamaño de las estructuras necesarias para administrarlo.
- Permisos: qué usuarios pueden acceder a qué archivo y con qué permisos.
- Fechas: se guardan fechas importantes, por ejemplo, cuándo se creó, cuando se modificó por última vez, etc.
- Propietario: el creador del archivo.

Operaciones básicas sobre archivos: todas son llamadas al sistema.

- Crear
- Abrir
- Leer
- Borrar
- Cerrar

- Escribir
- Reposicionar: es simplemente cambiar el valor de fseek (que es un puntero que indica dónde se va a producir la próxima lectura o escritura). Es simplemente saltar a una posición específica de un archivo.
- Truncar: tiene muchas definiciones. La más sencilla es “cambiarle el tamaño a un archivo”, puede ser para achicarlo o agrandarlo.
- Renombrar (mover): puede ser una operación básica o combinada. Es una operación simple cuando no se tuvo que realizar una copia de los bytes del archivo, sino que permanecen iguales. Es decir, lo único que se cambió fue el nombre del archivo pero el contenido del archivo nunca tuvo que ser modificado. Por eso mover entre carpetas es mucho más rápido que copiar un archivo.

El mover un archivo (dentro del mismo disco) implica sólo cambiar su nombre, es decir, su ruta en la entrada de directorio pero no implica mover los datos en disco. Distinto sería el caso si lo estuviésemos moviendo a otro disco, ya que acá sí se mueven los datos.

Crear y borrar tienen una particularidad con respecto a las demás. Ninguna de las dos requiere que abramos el archivo (todas las demás sí lo requieren). ¿Por qué tengo que abrir un archivo para leerlo? porque cada vez que tengo que hacer una operación sobre un archivo necesito conocer sus permisos. La idea de abrir archivos es que el SO tenga en memoria la información de sus atributos porque se van a leer constantemente. Por eso no está en el disco. También debe conocer cuáles archivos están abiertos (porque capaz si un archivo está abierto otro proceso no puede acceder a él). En el tema siguiente vamos a explicar esto mejor.

Operaciones combinadas sobre archivos: son combinaciones de las operaciones básicas.

- Copiar: abrir el archivo que quiero copiar, leerlo, crear un archivo en otro lugar, escribirlo y después cerrar los dos archivos. Es una combinación de operaciones básicas.
- Renombrar (mover): ¿por qué es una operación combinada? es una operación combinada cuando tengo que moverlo a otro file system. En este caso se necesitan copiar los archivos en el nuevo file system y luego eliminarlos del viejo.

Apertura de archivos: es una serie de tareas que se hace para poder acceder a los atributos de un archivo de forma más sencilla y rápida.

- Modo de apertura: cuando abrimos un archivo tenemos varias formas de hacerlo: abrirlo para escribir, abrirlo para leer, abrirlo para escribir al final, etc. En esta etapa también se evalúa si cuento con los permisos necesarios para abrir el archivo en ese modo.

A medida que empiezo a abrir archivos puedo ver que se forman 2 listas:

- Tabla global de archivos abiertos: indica todos los archivos abiertos en el sistema. Esta tabla contiene información importante sobre los archivos en cuestión. Esta información es global a todos los procesos porque pueden existir dos o más procesos que hagan referencia al mismo archivo. En general se guardan los siguientes datos:
 - Nombre
 - Tamaño
 - Contador de cantidad de aperturas en ese momento
 - Permisos
- Tabla de archivos abiertos por proceso: en esta tabla voy a tener información importante del proceso que está usando el archivo (por ej: modo de apertura, el puntero donde está leyendo o escribiendo, etc). Esta tabla va a tener un puntero a la tabla anterior porque necesita conocer los atributos de los archivos.

Bloqueos/locks: son a los archivos lo que los semáforos son a los procesos. Permiten al usuario bloquear alguna porción de un archivo para que no pueda ser accedida por otro proceso en un determinado momento. No se usan semáforos ya que los locks permiten bloquear desde un byte hasta otro. Por lo que dos o más procesos podrían acceder al archivo si todos acceden a bytes diferentes. Los semáforos bloquearían al archivo en su totalidad. La finalidad de esto es lograr la consistencia. Hay distintos tipos de locks:

- Compartido/Exclusivo:
 - Exclusivo: sólo puede acceder un proceso a la vez (se puede aplicar a un archivo completo o a una sección de un archivo). Es como un mutex.
 - Compartido: es lo contrario. Se permite que otros procesos accedan al archivo para leerlo.
- Obligatorio/Sugerido:
 - Obligatorio: el SO siempre aplica algún tipo de bloqueo cuando se abre.
 - Sugerido: el SO no garantiza nada. La responsabilidad es del programador para que las cosas salgan bien.
¿Cuál sería la ventaja de los lock sugeridos por sobre los obligatorios? Menos overhead. El hecho de tener un lock obligatorio implica que el sistema tenga que hacer validaciones extras para garantizar el bloqueo.

Tipos de archivos:

- Archivos del SO: archivos que puede reconocer el SO. Incluye a los archivos regulares y ejecutables, también a los archivos de tipo Directorio, los de tipo Dispositivo (/dev/sda o /dev/hda si es más viejo), los sockets, las terminales.
- Archivos regulares: archivos que guardan información para los usuarios. Pueden ser archivos word, excel, etc.
- Archivos ejecutables: bat = sh, exe, com

Métodos de acceso: existen diferentes maneras de acceder a un archivo y tienen que ver con cómo está armado el archivo internamente y en qué dispositivo está guardado (¿en un registro o en varios? ¿son todos del mismo tamaño?, etc). Hay dos métodos:

- Secuencial: parto de un inicio y leo hasta cierto punto. Leo el primer byte, luego el segundo, etc.
- Acceso directo: por ejemplo, si todos mis registros son del mismo tamaño y quiero acceder al tercero hago 3 * tam_registro y listo, no es necesario leer todo lo anterior (cosa que sí sucede con las cintas, que funcionan de forma secuencial). Un ejemplo de dispositivo con acceso directo es un disco de estado sólido.

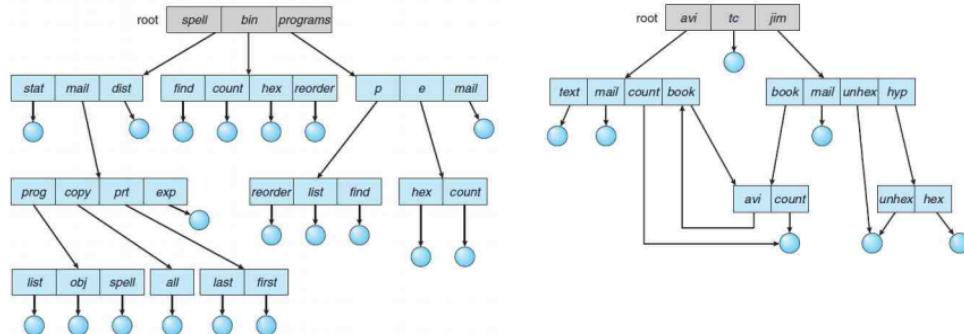
Ruta de un archivo: el nombre de un archivo no está compuesto solamente por el nombre que le dimos al archivo en sí, sino que encadena los nombres de los directorios donde está contenido. Es decir, el nombre de un archivo termina siendo su ruta. Hay dos tipos de rutas:

- Ruta absoluta: es el nombre completo del archivo, desde el punto de montaje(/ en linux o C: en Windows). No puede haber dos archivos con la misma ruta absoluta.
- Ruta relativa: es relativa a mi directorio de trabajo.
 - Working Directory: indica en qué directorio estoy trabajando en un momento determinado. Se cambia con cd y cuando quiero acceder a un archivo de mi working directory lo hago con ./ (el punto concatena los nombres de los directorios hasta el directorio actual).

Directorios: un directorio es un tipo de archivo, por lo que tiene operaciones que se hacen sobre él (que no son las mismas que se hacen sobre archivos regulares). Estas operaciones son:

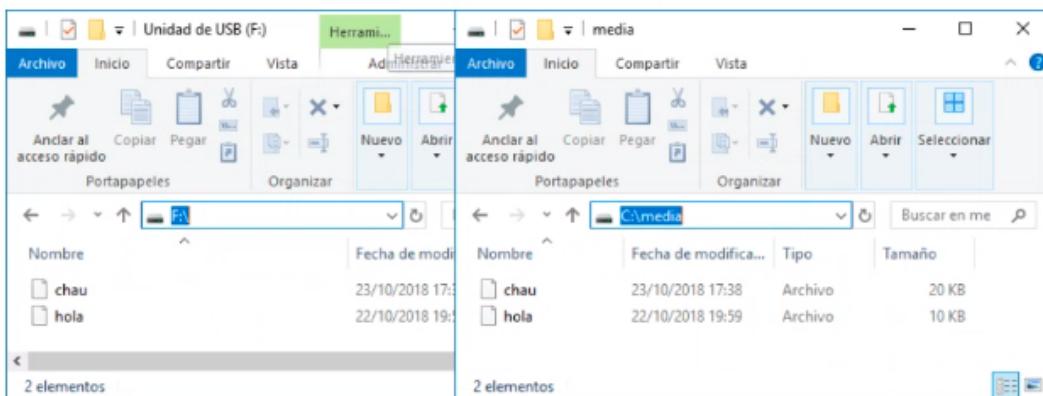
- Búsqueda de un archivo
- Crear un archivo nuevo
- Renombrar un archivo
- Borrar un archivo
- Listar un directorio
- Recorrer el Filesystem

Directorio raíz: desde donde se desprenden otros directorios. Inicialmente eran en forma de árbol apuntando hacia abajo, es decir, no se podía retroceder (figura de la izquierda). Hoy en día se usan árboles cíclicos que permiten tener “accesos directos”.



Punto de montaje: es el directorio o el archivo en el que se hacen accesibles un nuevo sistema de archivos, un directorio o un archivo. Para montar un sistema de archivos o un directorio, el punto de montaje debe ser un directorio y para montar un archivo, el punto de montaje debe ser un archivo. En linux el punto de montaje es un directorio dentro de la estructura de directorios y en Windows, habitualmente es una unidad representada por una letra C:, E:, F:, etc, pero también se puede hacer a través de una carpeta.

Ejemplo:



F:\
C:\media

En ese ejemplo, conecto un pendrive y F se convierte en el punto de montaje para empezar a leer el contenido.

Protección de archivos:

- Acceso total: sin protección. Los archivos pueden ser accedidos por cualquiera. Se usaba antes cuando los SO manejaban un único usuario.

- Acceso restringido: sólo el propietario tiene acceso y no puede darle acceso a otro usuario.
- Acceso controlado (permisos de acceso): para permitir o denegar. Específica qué operaciones se pueden hacer sobre los archivos.

Los tipos de permisos son 3: lectura (r), escritura (w) y ejecución (x) y se los puede combinar, por ejemplo rwx, rw, etc.

Permisos de acceso:

- Tipo Unix (rwx): a cada archivo creado se le asigna un propietario (que es quien lo creó), a qué grupo pertenece el archivo y en base a eso una serie de permisos. Esta estrategia tiene un problema y es que no me permite asignar distintos permisos para distintos grupos. Solamente ocupa 9 bits.

Por ejemplo:

```
-rw-rw---- 1 dan utnso 74841 oct 23 16:21 1er_llamado.doc
```

Donde el primer rw- pertenece al propietario (dan), el segundo permiso, rw-, pertenece al grupo (utnso) y el otro (--, es decir, ningún permiso) corresponde a todos los demás usuarios.

En Unix el comando chmod sirve para cambiar los permisos. chmod 777, por ejemplo, otorga todos los permisos a todos los usuarios. El primer 7 es para el usuario, el otro para el grupo y el otro para el resto (7 en binario es 111, o sea pone el permiso de lectura, escritura y ejecución en 1).

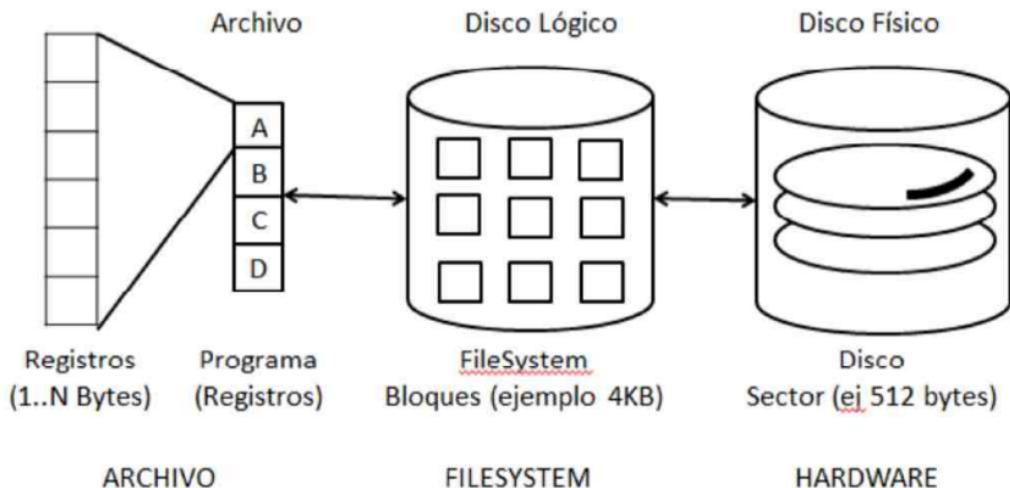
- Matriz de acceso: es una matriz que contiene archivos relacionados con distintos usuarios y grupos y qué permisos tiene cada uno. La desventaja de esta estrategia es que es una matriz demasiado grande y, además, cada vez que se agrega un nuevo usuario o archivo deben agregarse filas y columnas a esta tabla.
- Listas de control de acceso (ACL): es como la primera estrategia pero va agregando usuarios y grupos con punteros si es que necesita dar más permisos.
- Contraseñas: implica proteger al archivo con una contraseña. Hoy en día no se usa tanto. Lo que se hace actualmente es validar al usuario (seguramente con una contraseña) y con esa validación podrá acceder a todos sus archivos, pero no se suelen usar contraseñas individualmente para cada archivo.

DISCO LÓGICO

Concepto: un usuario, en un archivo, tiene cierta información. Esa información puede estar definida en registros (que son simplemente un conjunto de bytes). ¿Cómo se guardan estos archivos en el filesystem? El filesystem está separado en bloques de tamaño fijo e idéntico para todos. En un bloque no puede haber dos archivos dado que la unidad mínima de lectura es 1 bloque. Es decir, leen y escriben de a bloques.

Supongamos que tengo un archivo de 3 KB y quiero modificar un valor (un byte). Se va a leer el bloque completo, se va a traer a memoria y recién en memoria voy a poder modificar ese byte que quería cambiar. Una vez que guarde el archivo se va a "pisar" el archivo que estaba en el filesystem con el modificado que tengo en memoria.

Con respecto al disco físico, este también agrupa a los bytes en un conjunto, esta vez llamado sector, que no es lo mismo que un bloque. Un bloque del filesystem está compuesto por un grupo de sectores del disco físico. Para leer el disco se lee de a sectores.

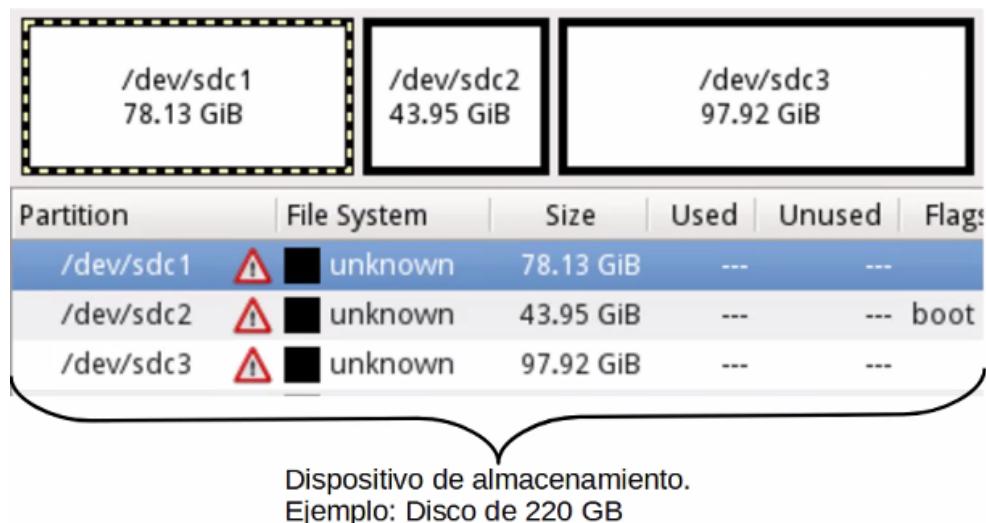


PARTICIÓN

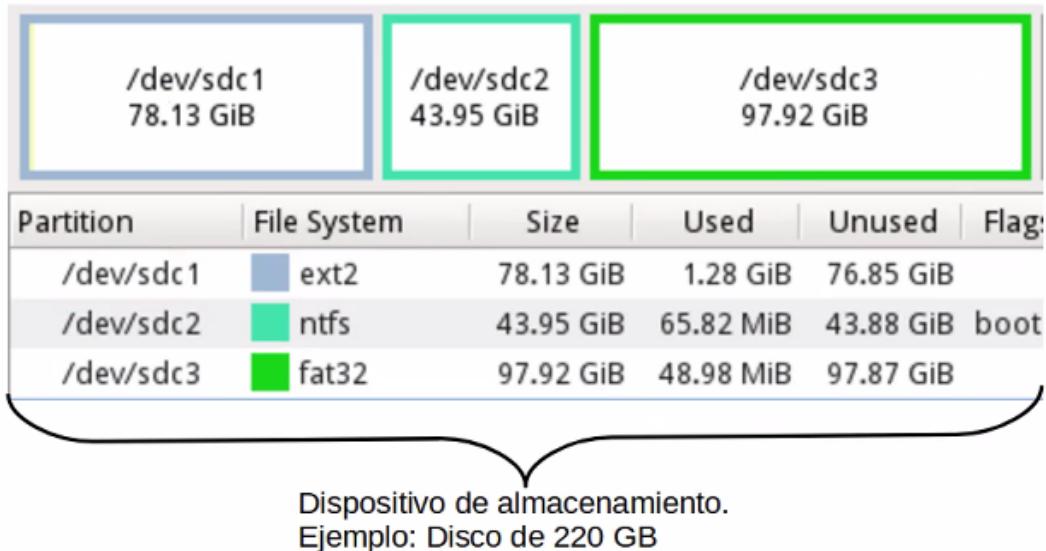
Definición: lo que estamos viendo es un disco y los discos se pueden separar en partes. En este caso, está separado en 3 particiones.

A la primera parte linux la reconoció como /dev/sdc1, a la otra /dev/sdc2 y a la otra como /dev/sdc3. Son simplemente archivos que mapean contra el disco. En los tres casos vemos que no tienen file system, que es "desconocido".

Esto no es más que separar al disco en partes y decir "la primera parte arranca acá, después arranca la segunda" y así, es simplemente un conjunto de bytes donde se podrán guardar archivos y directorios. Pero como está ahora (con file system en "desconocido" todavía no se puede guardar nada). Primero hay que "formatearlo".



Formatearlo hace referencia a "dar formato". ¿Formato de qué? de file system. Dar un formato de file system a una partición que está inicialmente vacía no es más que agregar las estructuras, tablas y espacios reservados para que el SO después sepa dónde tiene que guardar o buscar los archivos que están contenidos en el disco.



Una vez hecho esto puedo montar los filesystem. Esto se llama instalar un “volumen” (ext2, ntfs, fat32). Ahora tenemos que empezar a manejar un montón de estructuras internas al filesystem.

ESTRUCTURAS DE UN FILESYSTEM (EN DISCO)

Bloque de arranque o booteo: contiene la información necesaria para poder iniciar el SO.

Bloque de control del archivo (FCB): es al archivo lo que el PCB es al proceso. Es básicamente una estructura (hay una por cada archivo del filesystem) que contiene todos los atributos de un archivo (puede contener TODOS sus atributos o solamente parte de ellos, después lo vamos a ver), sean:

- ID
- Tamaño
- Fechas
- Nombre

También contiene la información necesaria para ir a buscar a todos sus bloques. Es almacenado en el disco. Se cachea a memoria cuando se está operando con el archivo, pero luego se escribe de nuevo en el disco al final.

Bloque de control de volumen: contiene detalles sobre el volumen, por ejemplo, la cantidad de bloques que hay en el filesystem, el tamaño de los bloques, dónde se encuentran las demás estructuras del filesystem, etc.

Estructura de directorios: es una estructura que debe indicar en qué bloque se encuentra el archivo de tipo directorio que representa al directorio raíz. Este sería el punto de entrada al filesystem. Por ejemplo: C:\

- Entradas de directorio: una entrada de directorio es un archivo. Las estructuras usadas para crear un archivo son las mismas, tanto para los archivos comunes, como para los directorios.

ESTRUCTURAS DE UN FILESYSTEM (EN MEMORIA)

Definición: hay estructuras que conviene que no estén siempre solamente guardadas en el filesystem y que estén en memoria porque es más eficiente tenerlas en memoria que en disco. Estas son:

- Estructura de directorios: es muy conveniente que esté en memoria dado que voy a usarla cada vez que reciba una llamada al sistema relacionada a un archivo. Tener que leerlo del disco haría que fuera todo más lento.
 - Tabla o lista global de archivos abiertos: lo mismo ocurre con esta tabla y la siguiente.
 - Tabla o lista de archivos abiertos por proceso:
 - Tabla de montaje:

IMPLEMENTACIÓN DE DIRECTORIOS

Definición: no hace referencia a la estructura de directorios que vimos antes, sino a cómo cargamos las entradas de un directorio (dentro de un directorio hay muchas entradas, que pueden ser directorios o archivos). La idea de esto es poder presentar los archivos de una forma ordenada y rápida.

- Lista lineal: una debajo de la otra.
 - Lista ordenada: se puede guardar en forma ordenada por un criterio.
 - Árbol:
 - Tabla de hash: similar a la tabla de páginas invertida.

MÉTODOS DE ASIGNACIÓN

Definición: básicamente los archivos se parten de acuerdo al tamaño del bloque y se guardan así. Si, por ejemplo, guardáramos el archivo en un momento (ocupando 4 bloques) y luego, en otro momento, quisiéramos agrandarlo se agregarían más bloques. Caso contrario, si nosotros borráramos parte del archivo entonces liberaríamos esos bloques que ocupaba. Ahora, esto no es tan sencillo. Todos los bloques menos el último siempre tienen que estar completos, entonces, si nosotros modificáramos la cantidad de caracteres del bloque 2, por ejemplo, esto impactaría en todos los bloques siguientes, dado que se debería mover la cantidad de bytes modificados al bloque anterior o al siguiente (dependiendo de si se borró o escribió).

También hay que aclarar que el último bloque puede sufrir fragmentación interna (máximo: $\text{tam_bloque} - 1$).

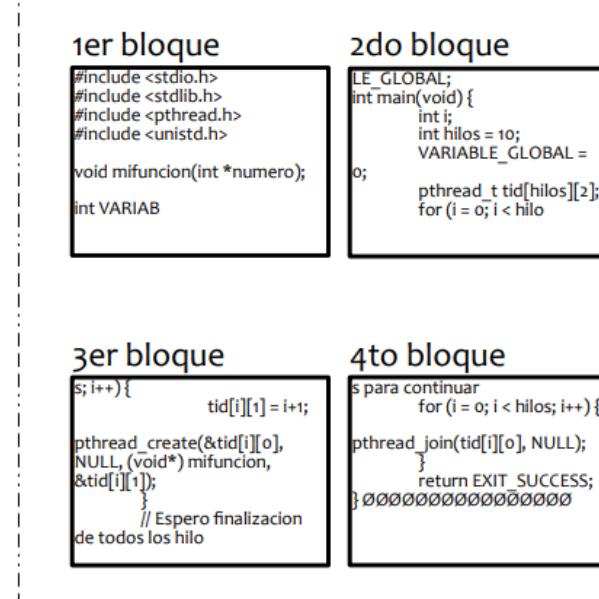
```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void mifuncion(int *numero);

int VARIABLE_GLOBAL;
int main(void){
    int i;
    int hilos = 10;
    VARIABLE_GLOBAL = 0;
    pthread_t tid[hilos][2];
    for (i = 0; i < hilos; i++) {
        tid[i][1] = i+1;
        pthread_create(&tid[i][0], NULL, (void*)
mifuncion, &tid[i][1]);
    }
    // Espero finalizacion de todos los hilos para
    continuar
    for (i = 0; i < hilos; i++) {
        pthread_join(tid[i][0], NULL);
    }
    return EXIT_SUCCESS;
}

```



Los directorios son un tipo de archivo, entonces, pasa exactamente lo mismo:

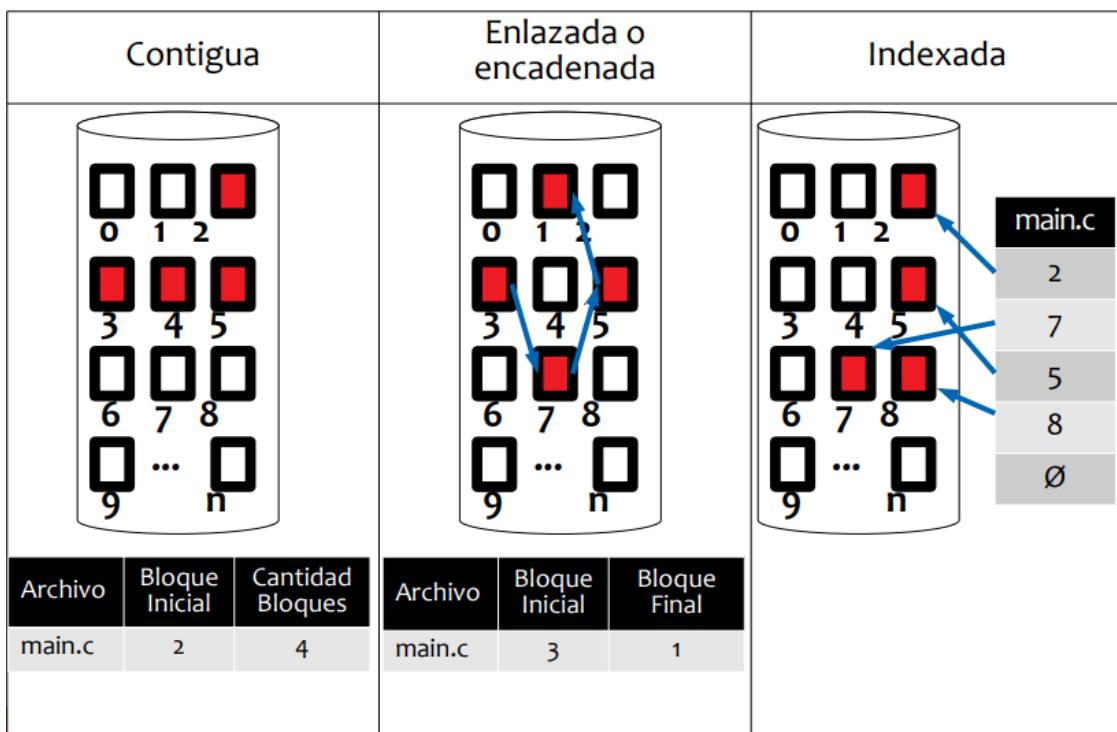
Documentos	directorio	1325	
Escritorio	directorio	7894	
Favoritos	directorio	1389	
final_febrero	regular	1367	
Imágenes	directorio	1684	
chrome	link	1328	
Contactos	directorio	9852	
Música	directorio	1289	
1er_parcial.doc	regular	1254	
1er_recup.doc	regular	1980	

1er bloque		
Documentos	directorio	1325
Escritorio	directorio	7894
Favoritos	directorio	1389
final_febrero	regular	1367

2do bloque		
Imágenes	directorio	1684
chrome	link	1328
Contactos	directorio	9852
Música	directorio	1289

3er bloque		
1er_parcial.doc	regular	1254
1er_recup.doc	regular	1980
00000000000000000000		
00000000000000000000		

Ahora la pregunta es, ¿dónde están estos bloques? Para guardarlos hay 3 estrategias:



Contigua: se establece un bloque inicial. Sabiendo la cantidad de bloques y que se van a guardar de forma contigua entonces se sabe que el archivo está en los bloques 2-3-4-5 y en ese orden.

- Estructuras necesarias: por cada archivo necesito saber el bloque inicial y la cantidad de bloques (o el tamaño del archivo). Esta información está en el FCB. Actualmente casi ni se usa, solamente para casos muy específicos.
- Ventajas:
 - Si quiero leer un bloque en específico simplemente hago la cuenta para saber en dónde está y leo únicamente ese bloque.
 - Es bueno para el acceso secuencial y el acceso directo.
- Problemas:

- Fragmentación externa (tengo bloques libres pero no los puedo usar porque no están contiguos).
- No permite agrandar los archivos. Si en el ejemplo el bloque 6 estuviera ocupado por otro proceso, no le podría asignar más bloques a mi proceso.

Enlazada/encadenada: es muy distinto al anterior. Para empezar, los bloques pueden estar en cualquier lugar del disco (no tienen que estar guardados en forma contigua). Se debe guardar únicamente el bloque inicial y el bloque final (pudiendo haber x cantidad de bloques entre medio). Los últimos bytes de cada bloque se utilizarán como un puntero al siguiente bloque.

- Estructuras necesarias: con conocer el bloque inicial y final de un archivo se puede obtener la totalidad del archivo.
- Ventajas:
 - Ya no existe la fragmentación externa (a un archivo puedo asignarle cualquier bloque que esté libre).
 - Se resuelve el tema de agrandar archivos.
- Problemas:
 - Si quiero leer el quinto bloque, por ejemplo, sí o sí tengo que pasar por todos los bloques anteriores (o sea, hacer 5 lecturas) y ver a dónde me llevan los punteros. No hay forma de saber de antemano qué bloque es.
 - En relación a lo anterior, se dice que no es adecuado para "accesos directos" porque si quisiera leer el último bloque de mi archivo debería leer todos los anteriores.
 - Si por un problema se pierde un bloque no se pierde solamente ese bloque, sino que se pierde la referencia a todos los bloques que le seguían.

Indexada: por cada archivo se tiene una tabla que indica cuáles son los bloques que contiene. Por cada bloque se tiene un puntero.

- Estructuras necesarias: una tabla por cada archivo.
- Ventajas:
 - No existe la fragmentación externa.
 - Se puede agrandar el tamaño de los archivos.
 - Se resuelve el problema del acceso directo. Si ahora quiero acceder al bloque 5 no es necesario leer los bloques anteriores, sino que simplemente accedo a la entrada 5 de la tabla.
 - ¿Qué pasa si pierdo un bloque? En principio se perdería solamente ese bloque. Al estar las referencias en la tabla se seguirá teniendo acceso a los demás.
- Problemas:
 - Si se pierde la tabla se pierde la totalidad del archivo.
 - ¿Qué tamaño le damos a la tabla? las tablas tienen una restricción de filas.
 - Es el que más espacio ocupa en el disco y el que más overhead genera.

Todas las implementaciones sufren de fragmentación interna en el último bloque.

GESTIÓN DE ESPACIO LIBRE

Definición: hay diferentes estrategias

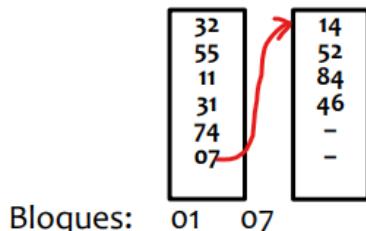
- Lista de bloques libres/Lista de porciones libres: se puede armar una lista con bloques libres y, cada vez que se asigna un bloque quitarlo de la lista.



- Bitmap: es una estructura de tamaño fijo y es muy sencilla y rápida de consultar.
 $\{0101001010001001010001010\}$ (libre/ocupado)
- Indexada (como archivo indexado): la idea es tener una tabla que indique cuáles son los bloques libres.

free
32
55
11
...
21

- Agrupamiento en bloques: es una combinación de lista con indexado. La idea es agarrar un bloque del file system y llenarlo de bloques que están libres (siendo los últimos bytes un puntero al siguiente bloque).



EJEMPLO DE ESCRITURA DE UN ARCHIVO NUEVO

Concepto: vamos a ver qué pasos hay que seguir para crear un archivo y escribirlo.

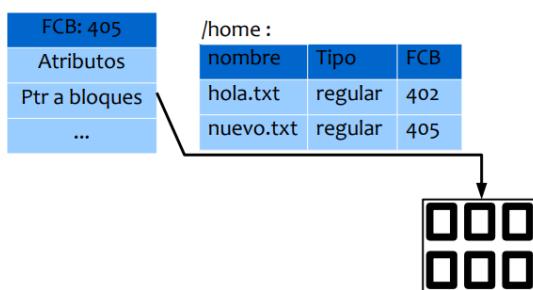
1. Crear el archivo:

- Verificar que exista FCB disponible. En este caso estaba libre el 405.
- Crear Entrada de Directorio.

FCB: 405	/home :
Atributos	nombre Tipo FCB
Ptr a bloques	hola.txt regular 402
...	nuevo.txt regular 405

2. Asignarle bloques requeridos:

- Abrir el archivo y agregarlo en las listas de archivos abiertos (a las dos listas, la global y la que es por proceso).
- Obtener bloques libres.
- Asignar los bloques al archivo.



3. Escribir en el archivo:

- Escribir los bloques.
- Actualizar atributos (fechas, tamaño, permisos). Si escribí en el archivo, el tamaño cambió, por lo que tengo que actualizarlo. Lo mismo con la fecha de última modificación, etc.

4. Cerrar el archivo:

- Actualizar listas de archivos abiertos.

Internamente el filesystem tuvo que hacer muchas tareas. ¿Qué pasa si por alguna razón se corta la ejecución en el medio? Para tratar con las incoherencias existen varias estrategias de recuperación.

ESTRATEGIAS DE RECUPERACIÓN

Comprobación de coherencia: existe, por ejemplo, un comando llamado chkdsk (check disk) en Windows que sirve para revisar que todas las estructuras estén en buen estado y sino se trata de recuperar.

Backups: copias de seguridad.

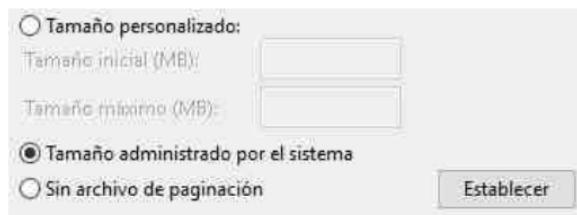
Journaling: tiene que ver con asegurarse que la operación (por ejemplo la escritura de un archivo) o no se ejecute o se execute por completo (parecido a las transacciones en bases de datos).

La idea es crear una lista con las tareas que se tienen que realizar para completar la operación. Una vez que se escriba la lista se comenzará a ejecutar las tareas. Si fuera a ocurrir un problema en el medio de la ejecución no habría problema porque luego se podría acceder a la lista y seguir ejecutando las tareas.

ÁREA DE SWAPPING

Definición: la memoria virtual está en el disco. Hay dos estrategias para usar memoria virtual.

- Una es la que usa Windows y es usar un archivo llamado "memoria de páginas" como memoria virtual. Simplemente maneja a la memoria virtual como si fuera un archivo más del filesystem. Otra estrategia es tener un archivo por cada proceso (en vez de tener todo en un único archivo). Vemos que en Windows se puede configurar el tamaño de la memoria virtual o incluso no usarla.



- Lo que hace Linux, en cambio, es asignar una partición (de tipo swap) para usar como memoria virtual. En Linux en principio no se podría cambiar el tamaño de la memoria virtual, dado que es una partición fija.

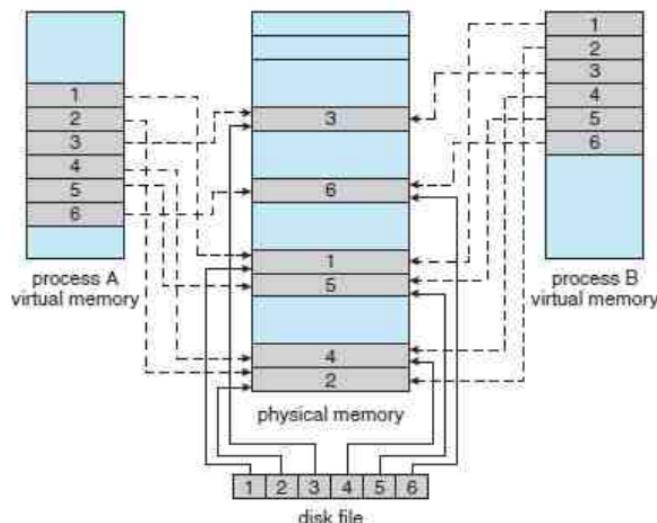
		New Partition #1 82.03 GiB				New Partition #2 17.97 GiB
Partition	File System	Size	Used	Unused	Flags	
New Partition #1	ext2	82.03 GiB	---	---	---	
New Partition #2	linux-swap	17.97 GiB	---	---	---	

MAPEO DE ARCHIVOS EN MEMORIA

Definición: mapear un archivo en memoria es agarrar un archivo muy grande y, en lugar de recorrerlo con fwrite para leerlo se le asigna un puntero a un void* y se empieza a usar ese archivo como si fuera memoria.

La idea es abrir un archivo mapeado a memoria y lo opero como si lo hubiese cargado todo a la memoria RAM. Es más eficiente en archivos grandes.

- Msync
- Munmap



FAT/EXT2

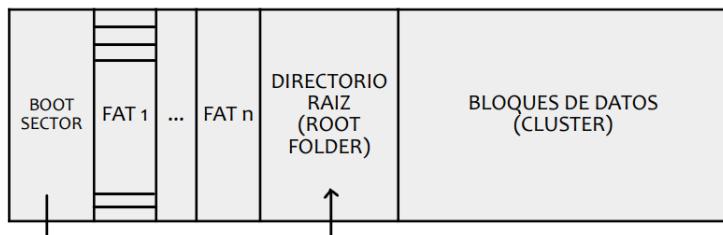
INTRODUCCIÓN

Filesystem: la mirada que tienen los filesystem sobre los archivos es de a bloques. Lo que hace un filesystem es dividir el espacio del dispositivo de almacenamiento en bloques lógicos (de tamaño fijo aunque existen de tamaño variable). Todos los bloques van a tener el mismo tamaño y la mínima unidad de lectura o escritura es un bloque. A su vez los bloques del filesystem se relacionan con alguna característica propia que tenga el dispositivo en donde se están guardando. Por ejemplo, si estamos hablando de un disco magnético (que tiene sectores), entonces podemos decir que un bloque del filesystem corresponde a varios sectores del disco. En este caso, la unidad mínima de lectura y escritura del disco es un sector.

FAT (FILE ALLOCATION TABLE)

FAT: es un filesystem que fue desarrollado en 1977 y utilizado es diskettes. Actualmente se utiliza en algunas memorias flash. Algunas de sus versiones son: 12/16/32/VFAT/ExFAT.

Instalación de volumen: no podemos simplemente tomar una partición del disco y guardar ahí nuestro archivo. Necesitamos contar con ciertas estructuras que luego nos permitan poder encontrarlo. Al conjunto de esas estructuras se lo conoce como "volumen". A esto también se lo llama "formatear", que quiere decir agregarle estructuras a algo para poder comenzar a utilizarlo. También se lo llama "instalar un filesystem".



Estructuras:

- **Sector de booteo:** en este sector vamos a encontrar información general del filesystem. Por ejemplo: el nombre del filesystem, el tipo de filesystem, el tamaño de los bloques. En algún lugar del boot sector nos va a decir cuál es el directorio raíz, es decir, en qué bloque de todo este sistema se encuentra (C; a fin de cuentas es un directorio en donde encontramos archivos y carpetas y es el punto a partir del cual podemos acceder a todas las demás carpetas del sistema).
- **FAT (1...N):** la FAT es básicamente una tabla que tiene una entrada por cada bloque del filesystem y cada entrada contiene un puntero al bloque siguiente. Estas entradas tienen un tamaño determinado según el tipo de filesystem. Esto lo vamos a ver más adelante.
Puede haber muchas tablas FAT (hasta N) pero estas son simplemente copias de la tabla original FAT 1. De forma estándar solamente se tiene FAT 1 y FAT 2 que sería su backup pero se puede pasarle un parámetro al filesystem cuando se está formateando para que cree más FAT.
- **Directorio raíz:** dentro del directorio raíz lo que tenemos es una lista de los archivos y directorios que contiene. También se tienen los atributos propios de ese directorio y los datos que necesito para poder acceder a los distintos bloques que pueden formar parte de ese directorio. A esto se lo conoce como "entrada de directorio" y en FAT tiene 32 bytes (1 byte

para el tipo de archivo, 11 bytes para el nombre, etc). Este sistema es tan sencillo que no tiene una estructura específica para el FCB, entonces los pocos atributos que tienen los archivos en este filesystem están guardados en la entrada de directorio:

Tipo de archivo	Nombre y extensión	Primer Cluster Archivo	Tamaño	...
-----------------	--------------------	------------------------	--------	-----

No hay un ID para los archivos, sino que tenemos un dato muy importante que es el "primer cluster archivo". Cluster es lo mismo que bloque. Es decir, esta entrada me dice cuál es el primer bloque del archivo. No me dice cuáles son todos sus bloques, sólo el primero.

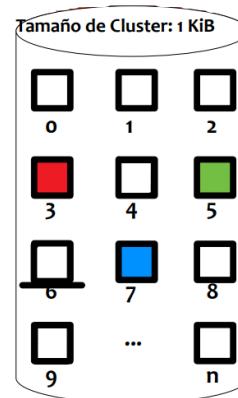
Por ejemplo:

Cluster con archivo tipo directorio	a arc.txt 7 2000b a tp.c 5 3000b d utnso 3 ob
-------------------------------------	---

Lo que estamos viendo es un archivo de tipo directorio. Adentro podemos ver listados los archivos que contiene. El primero es un archivo de tipo a (archivo) que se llama arc.txt, su bloque inicial es el 7 y su tamaño es de 2000 bytes.

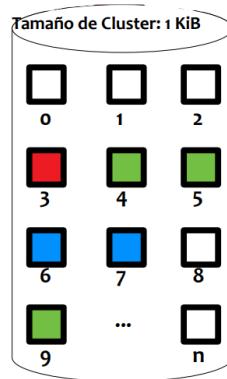
Entonces, en FAT, por cada archivo lo que tenemos es su bloque inicial. Por lo tanto, con la tabla anterior tenemos los siguientes datos:

A	arc.txt	7	2000 bytes	...
A	tp.c	5	3000 bytes	...
D	utnso	3	0 bytes	...



Pero me falta saber dónde está el resto de cada archivo. Para eso lo que vamos a usar es la tabla FAT. Esta tabla está indexada por bloque (es decir, T[0] corresponde al bloque 0, T[1] al bloque 1 y así, por lo que esta tabla tendrá tantas entradas como bloques haya en el filesystem). Esta tabla lo que guarda es un puntero al bloque siguiente. Por ejemplo, A empieza en el bloque 7. Por su tamaño sabemos que va a tener otro bloque. ¿Cuál es ese bloque? Lo vamos a encontrar en FAT[7], que tendrá un puntero al bloque siguiente.

0	libre
1	libre
2	libre
3	fin
4	9
5	4
6	fin
7	6
8	error
9	fin
...	
n	libre



En la tabla FAT estamos viendo que hay varios indicadores además de punteros: error, fin, libre. Fin quiere decir que es el último bloque de un archivo. Libre quiere decir que ese bloque está libre y error que ha habido un error en ese bloque y que se recomienda no usar.

VERSIONES DE FAT

Versiones que vamos a ver:

- FAT12: punteros de 12 bits. Puedo direccionar 2^{12} bloques
- FAT16: punteros de 16 bits. Puedo direccionar 2^{16} bloques
- FAT32: punteros de 32 bits. Puedo direccionar 2^{28} bloques
 - Sólo se usan los 28 bits menos significativos para el puntero.

Los bits hacen referencia al tamaño de las entradas de la tabla FAT.

Ejemplo FAT32:

- Puntero de 28 bits.
- Tamaño de cluster 2 KiB.

Tengo un puntero de 28 bits, entonces puedo referenciar hasta 2^{28} bloques. Si cada bloque es de 2 KiB entonces voy a poder referenciar hasta: $2^{28} * 2^{11} = 512 \text{ GiB}$ Este es el tamaño máximo teórico del filesystem.

Caso contrario tenemos el tamaño máximo real que es el tamaño que permite el disco. Con el ejemplo anterior, si tengo un disco de 500 GiB, el tamaño máximo real va a ser 500 GiB. Pero si tengo un disco de 1000 GiB, el tamaño máximo real va a ser 512 GiB.

¿Cuál es el tamaño máximo que puede tener un archivo? hay una limitación con respecto al tamaño del archivo en FAT.

En realidad hay 2 respuestas a esta pregunta. Una es la real y la otra es la que usa la cátedra. Según la cátedra, el tamaño máximo de un archivo es tan grande como sea el filesystem porque el archivo ocuparía todos esos bloques. Pero, en la realidad esto no es así porque en la Entrada de Directorio de FAT (que la vimos más arriba), el campo "tamaño del archivo" está limitado a 4 bytes. Entonces, en la realidad, en FAT 12, 16 y 32, los archivos no pueden tener un tamaño mayor a 4 GB.

Pasos para crear un archivo en FAT: se simplifica en crear una entrada de directorio, ubicar el primer bloque, si quiero más bloques necesitaré buscar más bloques libres, y los voy encadenando en la FAT.

FAT es un filesystem sencillo que no dispone de ciertas características avanzadas:

- No tiene seguridad.
- No hay permisos en los archivos.
- No hay registros de propietarios.
- No hay estructuras para administrar el espacio libre (no hay bitmap). Para saber si hay un bloque libre tiene que recorrer la tabla FAT.
- No hay esquema de journaling para manejar la integridad.
- Tolerancia a fallos: la tabla FAT suele estar copiada varias veces en disco.
- No tiene FCB.
- No tiene accesos directos de ningún tipo.

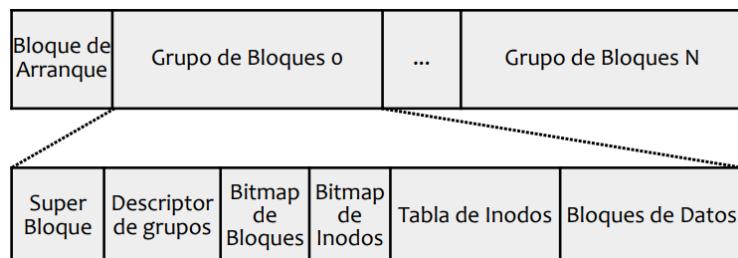
UFS (UNIX FILESYSTEM)/EXT2/EXT3

UFS: fue desarrollado en el año 1992 y actualmente se utiliza en sistemas Linux/MAC. Sus versiones son: EXTFS/EXT2/EXT3/EXT4.

Instalación de volumen: en el caso de UFS el volumen queda de la siguiente manera:



Inicialmente tenemos un bloque de arranque y después vemos que tenemos N grupos de bloques. Estos grupos de bloques internamente están diseñados de la misma manera y tienen las mismas estructuras que son las siguientes:



Estructuras:

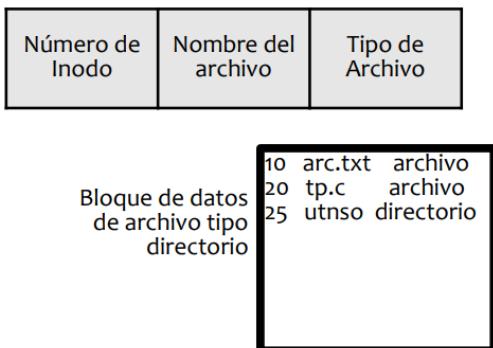
- Super bloque: en este sector vamos a encontrar información general del filesystem. Por ejemplo: el nombre del filesystem, el tipo de filesystem, el tamaño de los bloques, cantidad de bloques libres que puede llegar a haber, cantidad de FCB libres (o también llamados “inodos”).
- Descriptor de grupos: tiene información específica del grupo, por ejemplo, dónde empieza y dónde termina, cuántos bloques tiene libres, etc. Esta información y la información del super bloque se repite en todos los grupos (análogo a la creación de réplicas de la tabla FAT) por cuestiones de seguridad.
- Bitmap de bloques: indica cuáles bloques de este grupo están libres. Hay tantos bits como bloques haya.
- Tabla de inodos: un inodo es el FCB. En FAT no teníamos FCB porque todos los atributos del archivo estaban en la entrada de directorio. La particularidad de los inodos es que por cada grupo tenemos cierta cantidad de ellos que ya están creados de antemano (esta cantidad es igual para todos los grupos). Esto quiere decir que la cantidad de archivos que puedo tener es limitada.
- Bitmap de inodos: bitmap que indica qué FCB están libres.
- Bloques de datos: estos bloques son los que finalmente van a guardar los datos del archivo. Es el equivalente a lo que en FAT llamábamos Cluster con la diferencia de que lo tenemos separado en grupos (grupo 0...N).

Las últimas cuatro estructuras son distintas en su contenido en cada grupo, las dos primeras son iguales en todos los grupos.

DIRECTORIOS EN UFS

Comparación con FAT: los directorios en UFS tienen mucha menos información que en FAT. Recordemos que en FAT se guardaba toda la información del archivo en la entrada de directorio mientras que en UFS esa información se guarda en el inodo.

Estructura: lo único que se guarda es lo siguiente:



El número de inodo sería el identificador del archivo (porque la relación entre inodo y archivo es única). Estos son todos los datos básicos que se guardan aunque podría haber más (por ejemplo, el tamaño del nombre del archivo).

INODOS

Definición: por cada archivo se tiene un FCB/inodo en donde se guardan todos sus atributos. Algunos de ellos son: id, propietario, grupo, permisos, tamaño y debajo de eso vemos que hay una serie de punteros. Estos punteros son punteros a los bloques donde se encuentra el archivo. Entonces, el primer puntero va a apuntar al primer bloque del archivo, el segundo al segundo y así.

Un inodo tiene un tamaño fijo de 128 bytes (en todas las versiones de EXT menos en las más nuevas). Es decir que en esos 128 bytes se deben guardar tanto los atributos del archivo como los punteros a los bloques. En general siempre terminan siendo más o menos 15 punteros, lo que limita en gran manera el tamaño del archivo. Pero en la realidad esto no es así porque hay una forma de asignar más punteros.

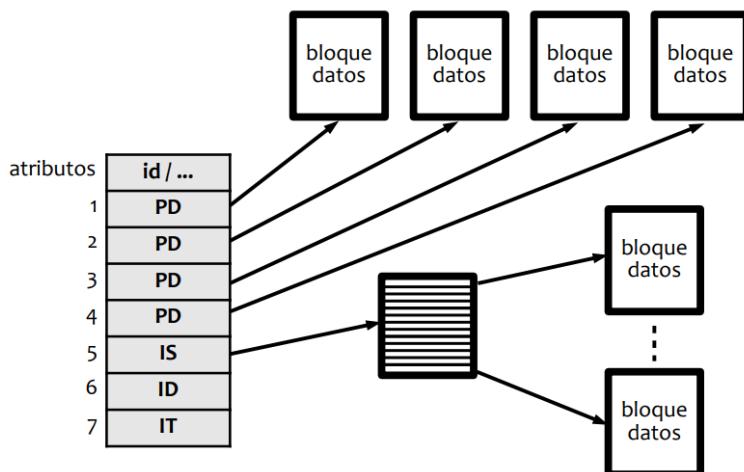
inodo 20	
	id
	propietario
	grupo
	permisos
	tamaño
	...
1	puntero
2	puntero
3	puntero
4	puntero
	...
P	puntero

El FCB se guarda dentro de la metadata del filesystem (por ejemplo, en la tabla de inodos en el caso de UFS), salvo casos particulares como FAT, que la distribuyen en las entradas de directorio o bien en la metadata de cada archivo.

Punteros: los punteros se crean a demanda utilizando bloques de datos. Una vez que se usaron todos los punteros del inodo lo que se hace es tomar un bloque de datos (estos bloques son los que guardan la información del archivo) pero en lugar de guardar la información del archivo lo voy a usar para guardar punteros. Ahí nacen dos tipos de punteros:

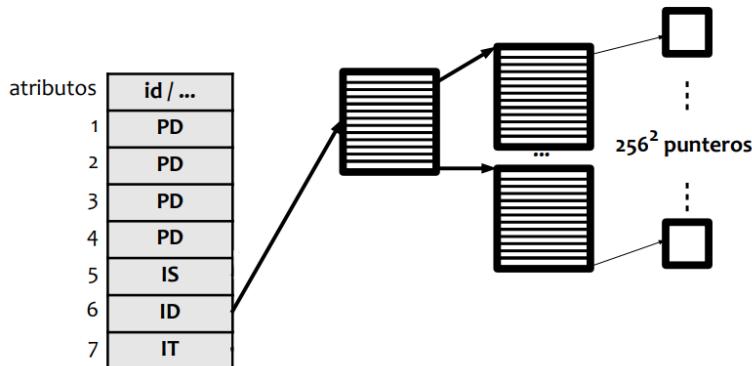
- Punteros directos: apuntan a un bloque de datos.
- Punteros indirectos: apuntan a un bloque de punteros.
 - Simples: apunta a un bloque de punteros.
 - Dobles: punteros que apuntan a bloques de punteros, donde sus punteros apuntan a otros bloques de punteros y recién esos punteros apuntan a bloques de datos.
 - Triples: un nivel más que los dobles.

Ejemplo indirección simple: en este ejemplo vemos que el inodo tiene sus atributos y abajo tiene 7 punteros. De esos 7 punteros los primeros 4 son punteros directos a bloques y los siguientes punteros son punteros indirectos (uno simple, uno doble y uno triple).

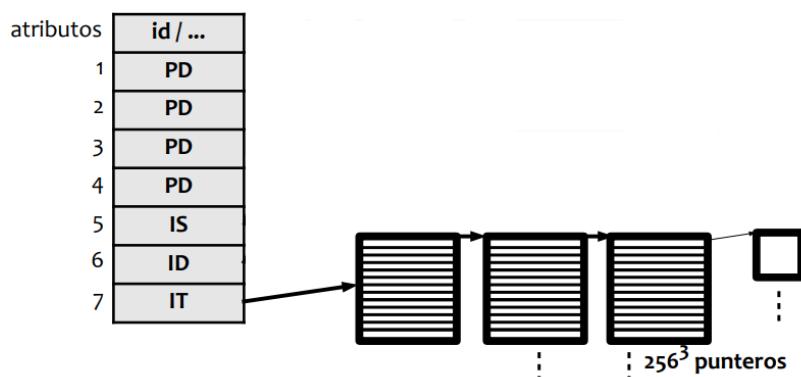


¿Qué cantidad de punteros hay en un bloque? $\text{Tam_Bloque} / \text{Tam_Puntero}$

Ejemplo indirección doble: en una indirección doble se tiene un puntero que apunta a un bloque y este bloque, en vez de estar formado por punteros que apuntan a bloques de datos, está formado por punteros que apuntan a bloques de punteros.



Ejemplo indirección triple:



Cómo hallar el tamaño máximo de un archivo: para este ejemplo tener en cuenta que el inodo tiene 4 PD (punteros directos), 1 IS (puntero indirecto simple), 1 ID y 1 IT. En los ejercicios de la práctica o de parciales muchas veces no nos va a decir cómo está conformado el inodo. En ese caso tenemos que tomar la configuración estándar que es: 12 PD, 1 IS, 1 ID y 1 IT.

- Tamaño Máximo Teórico de un archivo:

- Tamaño de bloque = 1 KiB.
- Tamaño de puntero = 4 bytes.
- Punteros por bloque = $1 \text{ KiB} / 4 \text{ bytes} = 256$ punteros por bloque.

$$(4 \text{ PD} + 256 \text{ IS} + 256^2 \text{ ID} + 256^3 \text{ IT}) * 1 \text{ KiB bloque} = \underline{\underline{16 \text{ GiB}}}$$

En el paréntesis lo que hago es sumar la cantidad de punteros que tengo. Tengo 4 PD, después tengo un IS, que me representa 256 punteros (porque por bloque entran 256), después sumo 256^2 por el ID y después 256^3 por el IT. Finalmente multiplico esos punteros por el tamaño de bloque y obtengo el tamaño máximo teórico que puede tener un archivo de ese filesystem.

- Tamaño Máximo Real de un archivo:

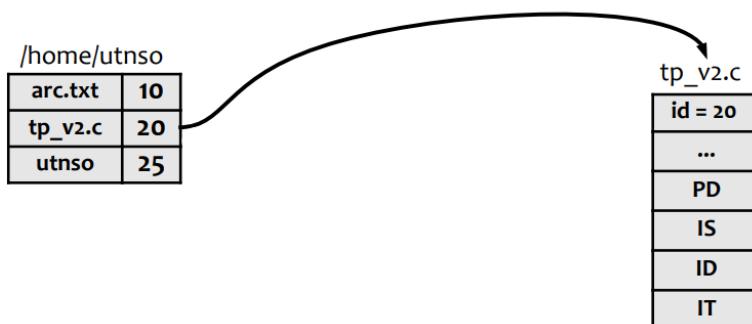
- Disco 8 GiB: Tamaño máximo real 8 GiB.
- Disco 30 GiB: Tamaño máximo real 16 GiB.

Tamaño máximo teórico del FS: acá no nos interesa la configuración del inodo en sí. Eso solamente nos sirve para saber el tamaño de un archivo. Para saber el tamaño máximo teórico que puede referenciar nuestro FS sólo nos interesa conocer 2 cosas: el tamaño del puntero y el tamaño de cada bloque. Supongamos que tenemos un puntero de 32 bits. En total existirán 2^{32} combinaciones posibles, entonces tendremos un total de 2^{32} punteros, cada uno apuntando a un bloque diferente. Si cada puntero apunta a un bloque, entonces el tamaño máximo que se puede referenciar es:

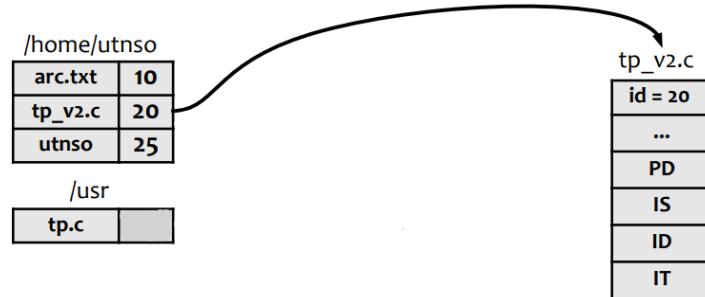
$$\text{Tam máximo teórico} = \text{tam puntero (en bytes)} * \text{tam bloque (en bytes)}$$

ACCESOS DIRECTOS (LINKS) EN UFS

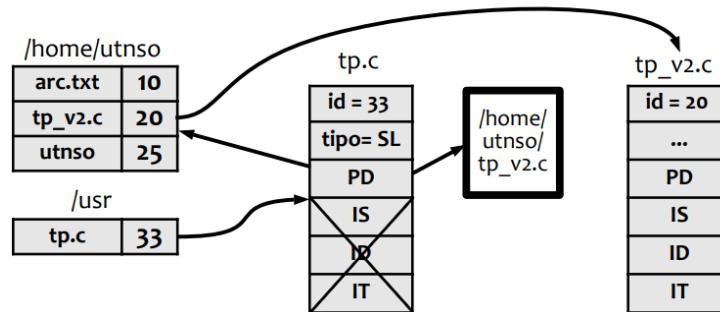
Soft link/Symbolic link: supongamos el siguiente ejemplo: en /home/utnso tengo los siguientes archivos:



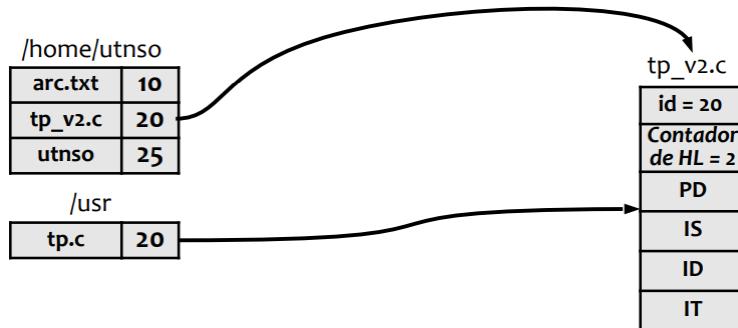
El que importa particularmente para el ejemplo es el archivo tp_v2.c, que corresponde al inodo 20. Supongamos que quiero crear un acceso directo a este archivo en otra carpeta, por ejemplo, /usr.



¿Qué inodo debería tener? Uno podría pensar que va a apuntar al 20, pero en realidad se va a crear un nuevo archivo (es decir, otro FCB) y este FCB va a ser de tipo SL (soft link). Su primer puntero directo apuntará a un bloque donde va a estar la ruta donde se encuentra el archivo original y los demás punteros no serán utilizados.



Hard link: el hard link es más directo aún. La entrada de directorio del acceso directo apuntará directamente al inodo del archivo original, de esta manera:



Entonces este tipo de acceso directo se ahorra la lectura del bloque que el soft link requiere. Ahora, al hacer esto se pierde un poco la noción de cuál es la entrada de directorio original. En realidad está mal dicho así, ya que el archivo original es el FCB 20 y allí están los bloques, después tp_v2.c o tp.c o cualquier otra que agreguemos son simplemente entradas de directorio, ninguna es el archivo original. Pero, entonces, si borramos tp.c, ¿tenemos que borrar el inodo 20? Como no sabemos cuál era la "entrada de directorio original", no tenemos forma de saberlo.

Para solucionar esto es que se agrega en el inodo un contador de hard links que indica cuántas entradas de directorio hay relacionadas a un inodo. El inodo no sabe cuáles son, solamente sabe cuántas hay. Entonces el inodo sólo se borrará cuando el contador quede en 0 y no haya nadie apuntando a él.

Limitaciones del hard link:

- Puramente por una cuestión de diseño, el hard link no puede apuntar a archivos de tipo directorio.
- El inodo de la entrada de directorio (en el ejemplo es el 20) es exclusivo de este file system. Si quiero que apunte a un inodo de otro filesystem no va a poder.

Diferencias con soft link:

- Cuando creo un hard link no creo un archivo nuevo.
- El soft link ocupa más espacio en disco porque tengo un archivo nuevo.
- En soft link tengo claro cuál es el archivo original. En hard link no.

¿Si quiero trabajar con archivos que van a estar en distintos discos, o en distintas particiones y quiero implementar accesos directos, cuál de las dos me conviene usar? Soft link porque es una referencia de alto nivel. Si tengo hard links a un archivo que está en otra partición, en otro disco, el número de inodo al que apunta va a ser otra cosa. Esto es porque los inodos son únicos dentro de un volumen.

Ejemplo:

Punteros de 32 bits (4 bytes).

Tamaño de Cluster 1 KiB (2^{10} bytes).

- Tamaño máximo teórico del filesystem:
 $2^{32} * 2^{10} = 2^{42} = \textbf{4 TiB}$
- Tamaño Máximo Real (Disco de 500 GiB)
 $4 \text{ TiB} > 500 \text{ GiB}$ entonces es 500 GiB
- Tamaño Máximo Real (Disco de 8 TiB)
 $4 \text{ TiB} < 8 \text{ TiB}$ entonces es 4 TiB

Por qué no hay accesos directos en FAT: hablamos de accesos directos solamente en UFS y no en FAT porque en FAT los atributos de un archivo se encuentran en la entrada de directorio. Usando accesos directos y haciendo que estos apunten al mismo bloque inicial, uno pensaría que en FAT funciona, pero no, porque al tener atributos del archivo en la entrada de directorio, cualquier acceso directo que creemos podría modificar estos atributos y esto generaría inconsistencias.

Pasos para crear un archivo en UFS: primero hay que buscar un inodo libre en cualquiera de los grupos. Despues hay que completar el inodo y buscar con el bitmap de bloques qué bloques están libres y despues empezar a escribir el archivo. Si el archivo es muy grande tendré que empezar a usar los bloques de punteros. Vemos que son muchos más pasos que en FAT, por eso se dice que EXT es mucho más complejo y tiene más overhead.

FAT VS. UFS

	FAT	UFS
Complejidad	Baja	Alta
FCB	No	Inodo
Espacio Libre	No	Bitmap
Journaling	No	Si
Links	No	Si
Seguridad	No	Si

Entrada/Salida

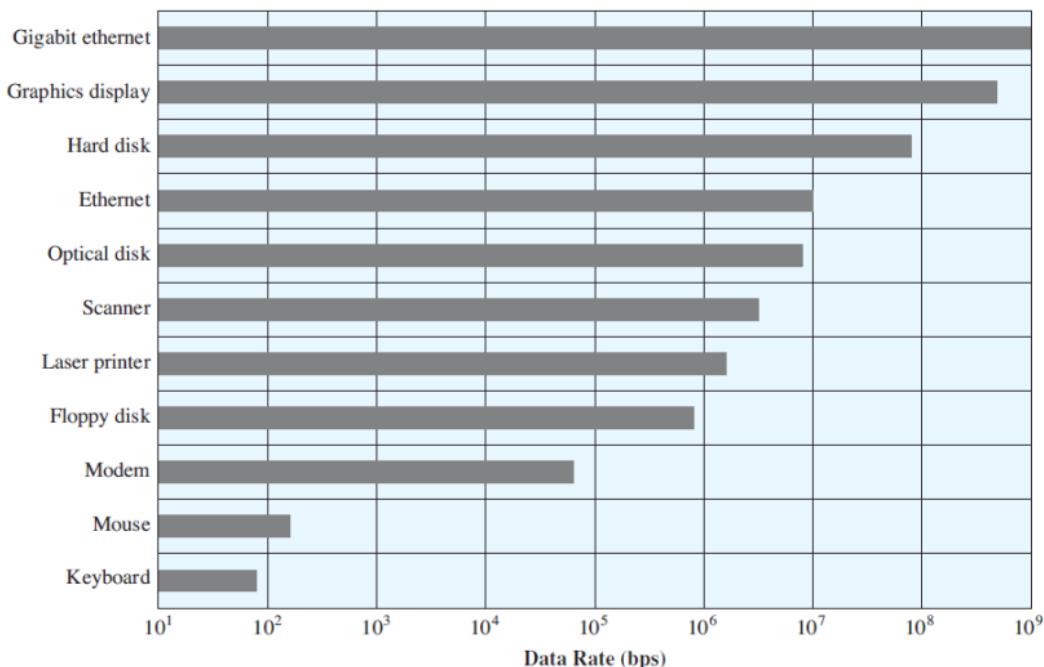
DISPOSITIVOS DE E/S

Tipos:

- Comprensibles para el usuario: el usuario interactúa directamente con ellos. Por ejemplo, un mouse, un teclado, una impresora.
- Comprensibles para el sistema: el usuario no suele interactuar con ellos directamente. Por ejemplo, un disco.
- De comunicación: una placa de red, un módem.

Diferencias entre los distintos dispositivos de E/S:

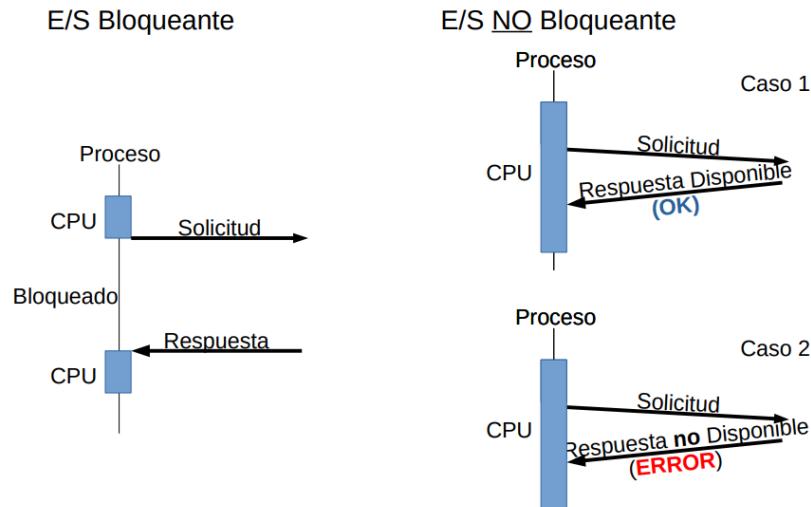
- Velocidad de transferencia: mientras más cerca están del usuario más lentos son y mientras más cerca están del sistema más rápidos son. Se dice que son “más rápidos” porque pueden procesar más bits por segundo.



- Uso: el uso de los dispositivos es muy variado pero este ítem hace referencia a que un mismo dispositivo se puede usar para cosas diferentes. Por ejemplo, un disco rígido puede usarse para guardar los archivos del filesystem pero también puede usarse para extender la memoria (que es lo que vimos como memoria virtual).
- Complejidad: este es otro ítem que varía altamente de dispositivo a dispositivo. Tomemos, por ejemplo, un mouse y un disco rígido.
- Unidad de transferencia: un byte o un bit no siempre son la unidad de transferencia. Acá hay que diferenciar dos tipos de unidades de transferencia.
 - Por carácter: este es el caso del teclado.
 - Por bloque: por bloque significa que la unidad de transferencia mínima deja de ser un byte y pasa a ser un valor más grande. Por ejemplo, como vimos en filesystem, en el disco la unidad mínima es un sector.
- Condiciones de error: ¿qué sucede si falla el dispositivo de E/S? de nuevo, esto es algo que varía ampliamente dependiendo del dispositivo. Si se rompe un mouse no pasa nada, se

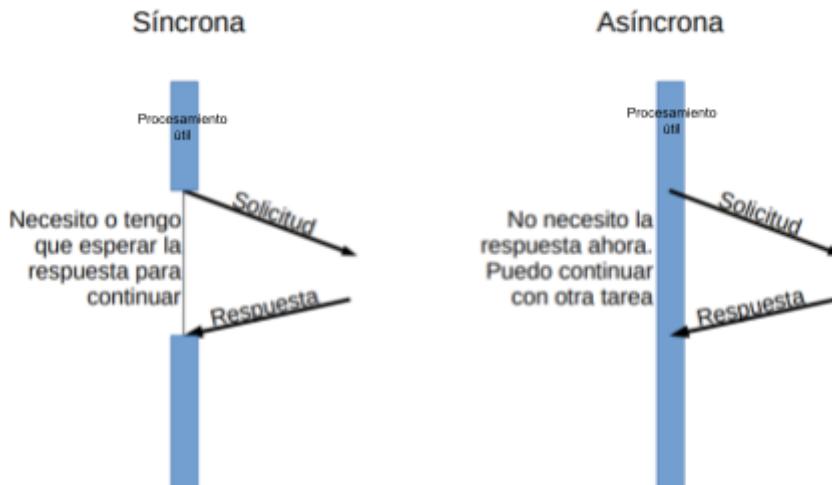
puede reemplazar por cualquier otro mouse. En cambio, si se rompe el disco de nuestra computadora probablemente perderemos la mayoría de los archivos.

- Bloqueante/No bloqueante: las E/S bloqueantes bloquean al proceso que realiza la solicitud del recurso. El proceso se desbloqueará una vez que reciba una interrupción de finalización de evento. En cambio, en las no bloqueantes el proceso no se bloquea sino que puede continuar su ejecución pudiendo producirse dos casos:
 - Caso 1: cuando el proceso necesita la respuesta ya la ha recibido.
 - Caso 2: el proceso necesita que haya ocurrido el evento de E/S y este no ha ocurrido aún o no se ha obtenido una respuesta, por lo que al intentar obtener el resultado va a obtener un mensaje de error.. El programa deberá implementar un mecanismo de validación para manejar esto.



El hecho de que sean bloqueantes o no bloqueantes tiene que ver más que nada con el SO y el tipo de syscalls que utilice para solicitar el uso de cada recurso, no es algo que normalmente pueda modificar el programador.

- Síncrona/Asíncrona: está muy relacionado a lo anterior.
 - Síncrona: el proceso sí o sí necesita la respuesta para continuar con su ejecución.
 - Asíncrona: no necesita la respuesta de forma inmediata sino que mientras espera la respuesta puede realizar otras tareas.



- Acceso aleatorio o secuencial: el acceso secuencial es un acceso que se tiene que dar en orden (por ejemplo en las cintas, para llegar a un punto determinado hay que leer todo lo

anterior). El acceso aleatorio implica que se puede acceder directamente a la sección deseada, sin necesidad de tener que leer lo anterior.

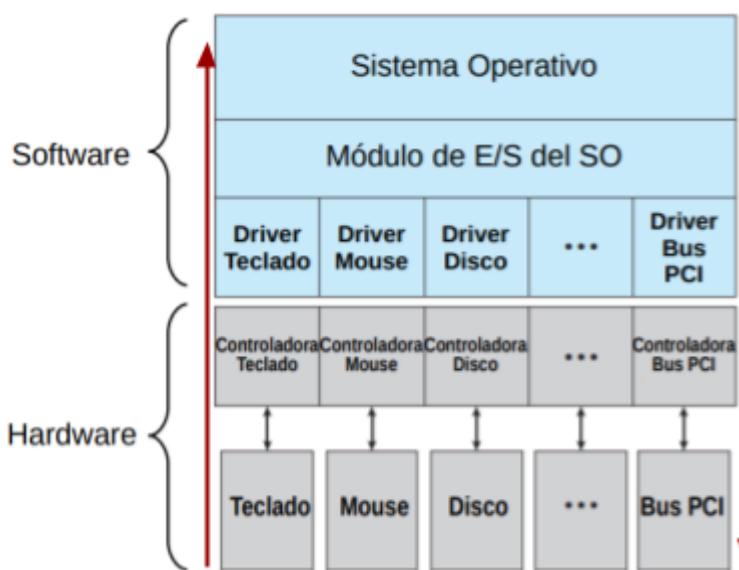
- Compartible o dedicado: puede ser usado por muchos usuarios o por un único usuario a la vez.
- Lectura, escritura o ambos:

Combinación entre Síncrona/Asíncrona y Bloqueante/No bloqueante: los casos están explicados en el cuadro pero quizás el más fácil de ver es el Síncrona-Bloqueante, que es cuando me bloqueo a la espera de una respuesta. El otro caso más común es Asíncrona-No bloqueante.

	Síncrona	Asíncrona
E/S Bloqueante	Luego de la solicitud el proceso se bloquea a la espera de la respuesta. Ej: read() bloqueante	El proceso se bloquea a la espera de algún evento no específico. Ej: select()
E/S NO Bloqueante	Luego de la solicitud el proceso no se bloquea, pero como necesita la respuesta para continuar, pregunta constantemente si la respuesta está disponible. Ej: while(noListo) {/esperar*{}}	Luego de la solicitud el proceso sigue realizando tareas no relacionadas a la respuesta esperada. Ej: read() no bloqueante

ESTRUCTURA PARA MÓDULOS DE E/S

Concepto: el gráfico muestra los componentes que intervienen en el uso de dispositivos de E/S desde el SO hasta el dispositivo en cuestión.



Como indican las flechas, la información puede viajar de arriba hacia abajo o de abajo hacia arriba. Por ejemplo, la lectura de disco comenzaría en la capa de SO donde se realiza una llamada a "READ" y, por los parámetros, vamos a identificar a qué dispositivo se refiere. Con esta información el módulo de E/S del SO va a saber a través de qué driver va a poder comunicarse con el dispositivo en cuestión. Un driver no es más que un SW que suele realizar el fabricante de cada dispositivo (y por cada SO suele haber uno diferente) que le permite al SW de la computadora comunicarse con el HW del dispositivo.

Ahora, este dispositivo (y todos los dispositivos) va a estar relacionado con alguna controladora (placa, chip) que tiene que cumplir una función de acuerdo a lo que el driver indica. Siguiendo el ejemplo de la lectura de disco, supongamos que al driver del disco se le indica que se quieren leer 700 bytes, esto el driver lo va a traducir en un idioma que el disco lo pueda entender y después le va a dar la orden a la controladora, quien le dará la orden exacta al disco para hacer que sus discos se muevan y demás para efectuar la lectura.

¿Esto funciona así para todos los dispositivos?: sí, la idea es que sí. El objetivo es lograr que se trate a todos los dispositivos de la misma manera sin importar su complejidad dado que sería imposible implementar una estrategia distinta para cada dispositivo de E/S que se quiera conectar.

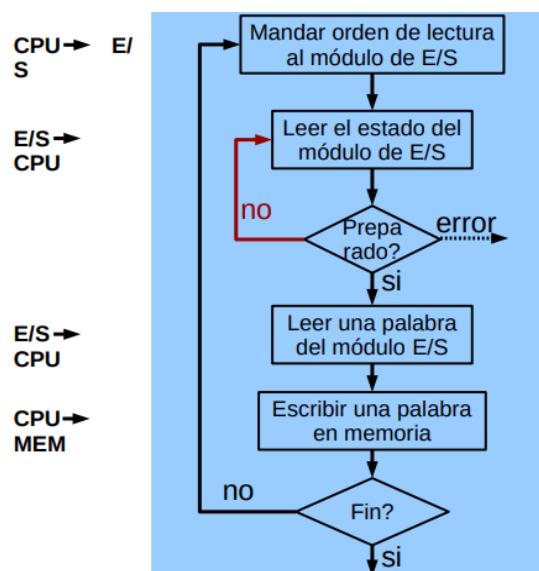
TÉCNICAS PARA TRANSFERIR INFORMACIÓN

Concepto: existen 3 técnicas para transferir información desde o hacia la E/S.

- E/S programada.
- E/S por interrupciones.
- Acceso directo a memoria (DMA).

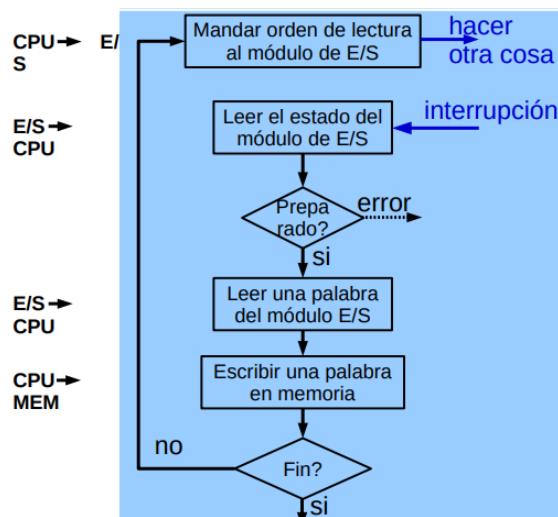
E/S programada: el procesador es el encargado de pasar la información que tenga que leer o escribir del dispositivo de E/S al espacio de memoria del proceso que solicitó la operación. Para hacer esto la CPU tiene que realizar 4 operaciones:

- Activación: verificar que el dispositivo de E/S esté listo.
- Tarea a realizar: determinar qué tipo de operación va a hacer.
- Comprobación de estado: verificar el estado del módulo de E/S.
- Transferencia: realizar la operación.



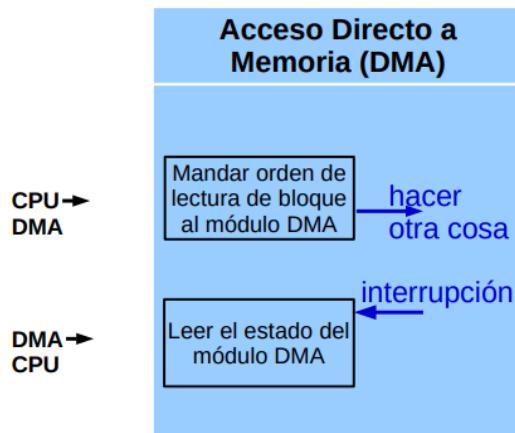
¿Qué vemos en el diagrama? primero, que todo lo realiza la CPU. Si vemos el IF de ¿Preparado?, vemos que verifica si el módulo está preparado o no y si no lo está entra en un loop que verifica esta condición.

Este es un procesamiento no útil y un desperdicio del recurso (CPU). Esto genera que el rendimiento general del sistema decaiga. Para evitar esto es que surge la siguiente técnica.



E/S por interrupciones: es parecido a lo que ya vimos antes. La idea es que se genere una interrupción cuando el recurso esté disponible. Este método sigue teniendo el problema de que es la CPU la encargada de copiar los datos que quieren transmitirse cosa que es no deseable dado que la principal función de la CPU debería ser ejecutar las instrucciones de los programas, no ocuparse de las entradas y salidas. Para mejorar estas dos primeras técnicas surge la última, el DMA.

Acceso directo a memoria (DMA): esta técnica permite desligar al procesador de ir transmitiendo información porque agrega un procesador extra (llamado "DMA") cuya única función es llevar y traer datos entre la memoria y los dispositivos de E/S. Su funcionamiento es mucho más sencillo que el de la CPU.



La idea es que la CPU le indique al DMA qué operación debe realizar y que éste dispare una interrupción una vez que haya finalizado.

La CPU debe indicarle al módulo DMA:

1. Tipo de operación (Lectura o Escritura).
2. Dirección del dispositivo de E/S.
3. Cantidad de bytes a leer (o escribir).
4. La ubicación de la memoria..

Igualmente sigue habiendo un problema y es que el DMA y la CPU utilizan el mismo bus de datos. Esto quiere decir que mientras uno lo esté usando el otro no podrá hacerlo. De todas maneras, considerando que la CPU tiene sus propias caches y otras formas de optimización, este inconveniente termina siendo mínimo y hoy en día todas las computadoras usan DMA dado que es la técnica más óptima para transferir gran cantidad de datos. En general se dice que la DMA "roba ciclos". Un "ciclo" hace referencia a la operación básica de un bus de datos (ciclo de bus). Es decir, un ciclo permite realizar una transferencia elemental de un dato entre dos dispositivos. En esta transferencia, la información se lleva de un elemento que se denomina fuente a otro que se denomina destino. Entonces, como la DMA utiliza el mismo bus de datos que la CPU, se dice que la DMA le "roba" ciclos de bus.

Resumiendo:

	Sin Interrupciones	Con Interrupciones
Transferencia de E/S a Memoria a través de la CPU	E/S Programada	E/S por Interrupciones
Transferencia directa de E/S a Memoria (mínima intervención del CPU)		D M A

BUFFERING

Buffering: cuando hablamos de "buffering" hablamos del uso de buffers.

Buffer: son espacios de memoria dentro del kernel del SO que están reservados para almacenar datos mientras se transfieren entre dispositivos o entre un dispositivo y una aplicación. Sus funciones más importantes son:

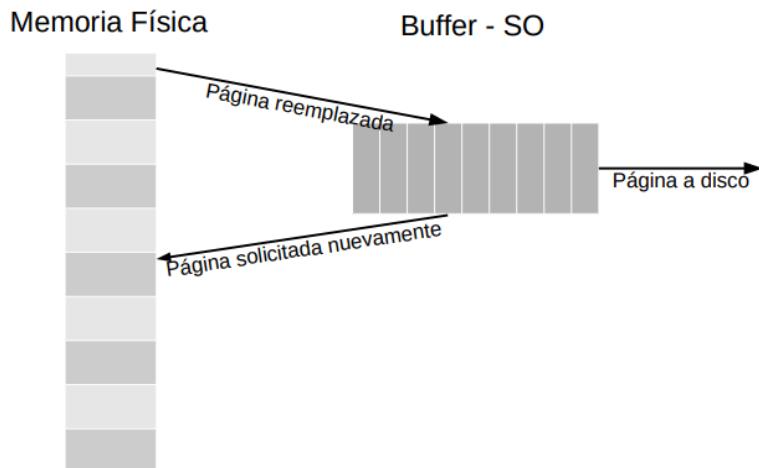
- Adaptar la unidad de transferencia: supongamos que quiero leer 25 bytes de un archivo pero al leer el bloque obtengo 50 bytes. El buffer se encargará de devolverle al usuario (es decir, al proceso) los 25 bytes que pidió.

- Adaptar diferencias de velocidades: tiene que ver con la diferencia de velocidad de transferencia de los dispositivos de E/S.

Buffering de páginas: un ejemplo del uso de un buffer es cuando hacemos el reemplazo de una página en memoria virtual. Habíamos dicho que cuando se seleccionaba a una página como víctima, esta se escribía en el disco. Como una escritura en disco tarda, lo que se hace es copiar la página en uno de estos buffers del SO.

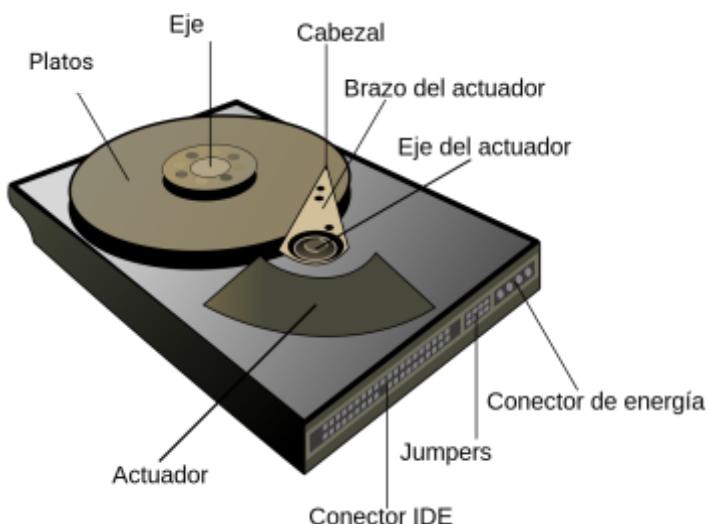
Esto trae dos ventajas:

- Páginas modificadas se escriben en grupos: supongamos que guardamos en los buffers varias páginas víctimas, cuando se quieran escribir a disco sólo se tendrá que acceder una vez y allí se copiarán todas las páginas.
- Una página reemplazada puede estar disponible en memoria: si esta página vuelve a ser solicitada antes de que se escriba en disco, nos ahorraríamos 2 entradas a disco porque la página siempre estuvo en el buffer.
- Evita el uso del bit de bloqueo de páginas:



PARTES DE UN DISCO

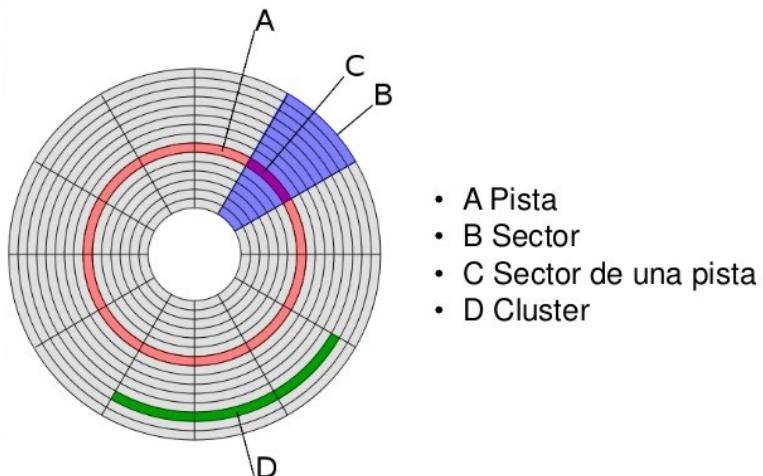
Estructura interna de un disco:



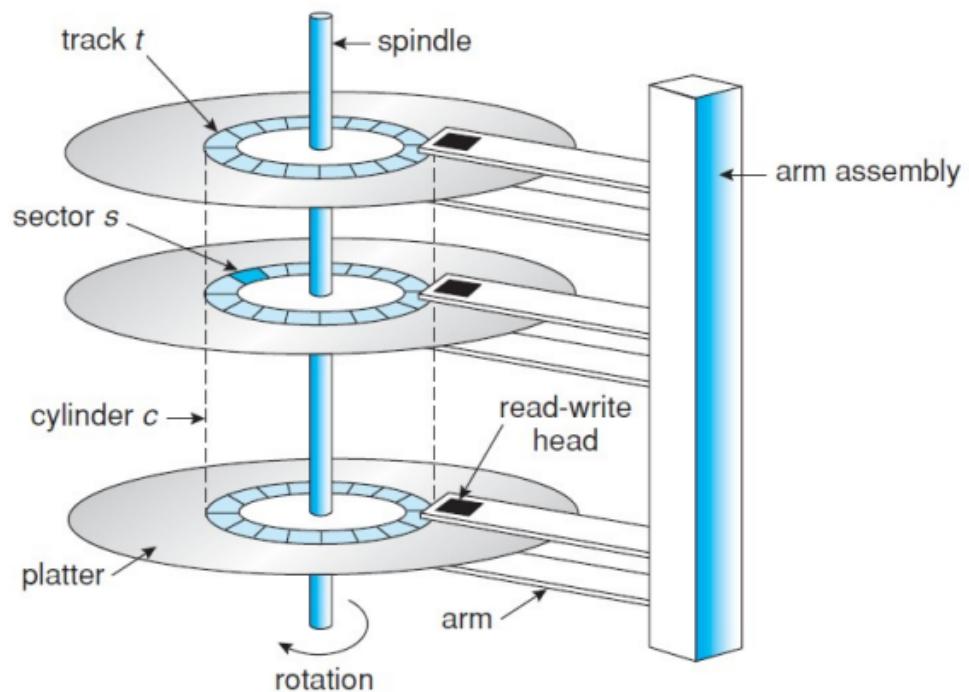
Plato o disco: es el componente principal de un disco duro dado que es donde se almacenan los datos. Los discos duros modernos normalmente emplean uno o más platos, fijados en un mismo eje

y perfectamente calibrados. Un plato puede tener una cara o dos, por lo que puede almacenar información en una de sus caras o en ambas, requiriendo en ese caso un cabezal de lectura/escritura para cada cara.

Visto desde una cara, un disco tiene las siguientes partes:



Visto en otra perspectiva, la estructura es de la siguiente forma:



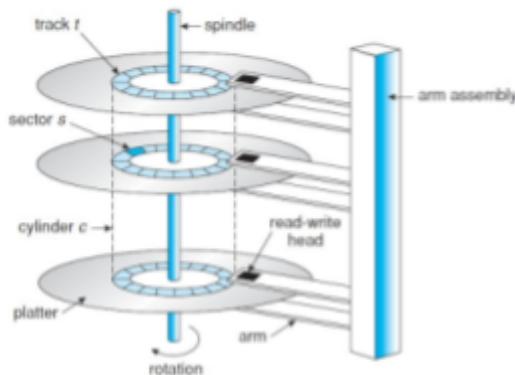
Donde los platos tienen 2 caras (por eso hay un cabezal por cada cara del disco). Este cabezal es el que va a terminar realizando la operación de lectura o escritura. Supongamos que quiero leer el sector s señalado, para referenciarlo vamos a decir "está en la pista n° x del sector x de tal cabezal". En realidad, en vez de hablar de pistas vamos a hablar de "cilindros". ¿Qué es un cilindro? el cilindro no es más que un conjunto de pistas (es decir, un conjunto de las pistas identificadas con el mismo número de todos los discos). Entonces, la referencia a un sector quedaría de la siguiente manera:

#C
nro. de cilindro

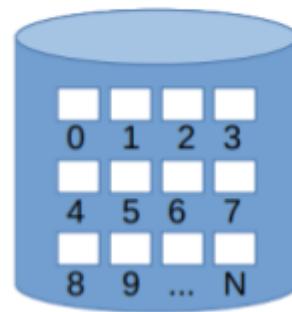
H
cabezal

S
nro. de sector

Todo esto que hablamos hasta ahora es la estructura física del disco pero nosotros ya habíamos visto su estructura lógica cuando vimos filesystem.



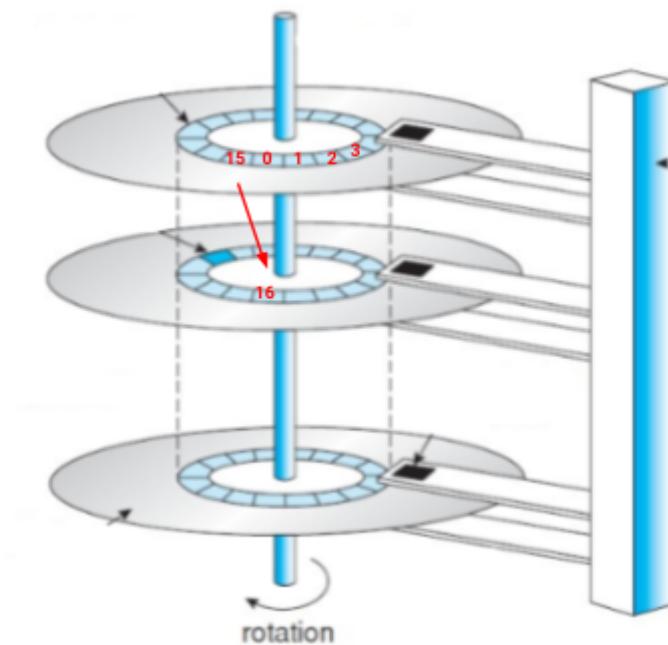
Estructura física



Estructura lógica

Como vemos, la estructura lógica (que es la que maneja el SO) es muy sencilla. Simplemente se numeran los sectores de 0 a N pero no se manejan platos, cabezales, cilindros ni nada de eso.

¿Cuál es la relación entre la numeración de los sectores en la estructura física y lógica?:



Los sectores se empiezan a numerar por cilindro. Por ejemplo, el primer plato en su primera pista tiene los sectores 0 - 15, y los sectores siguientes comienzan en el mismo cilindro pero del plato siguiente.

Es decir que se va completando de a cilindros y sólo se puede pasar al cilindro siguiente si se ha completado el anterior.

TIEMPO DE ACCESO A UN SECTOR

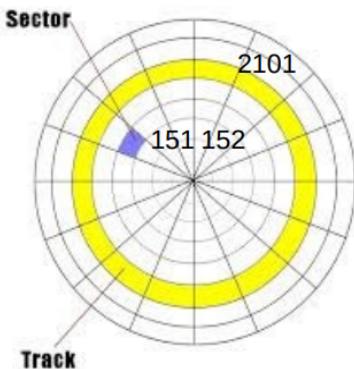
Concepto: ¿por qué tardan las operaciones de disco? ¿Cuánto tardan estas operaciones? ¿Tardan todas las operaciones el mismo tiempo? no, no tardan todas lo mismo. Depende de la operación, no es como la memoria. El tiempo total de acceso (TA) a un sector es la suma de:

- Tiempo de búsqueda (TB): tiempo para posicionar el brazo en la pista deseada.
- Latencia rotacional (LR): tiempo en que tarda el sector en posicionarse debajo de la cabeza.
- Tiempo de transferencia (TT): cuánto tarda el cabezal en leer la información que está en el sector. Este sería el único de los tiempos que se puede decir que es "constante".

Entonces: $TA = TB + LR + TT$

Pero entonces, la planificación de disco, ¿qué tiempo trata de optimizar? el tiempo de búsqueda. La idea de planificar el disco es ver qué pistas y sectores leer primero o, mejor dicho, en qué orden leer los sectores pedidos para lograr el menor tiempo posible.

Ejemplo:



Tiempo promedio de búsqueda: 4ms
Latencia Rotacional promedio: 4ms

¿Qué es lo más óptimo? en este caso va a ser leer los sectores 151 y 152 juntos dado que están uno al lado del otro pero esto no siempre es así.

Posicionarse en Sector 151 y 152 aprox 8ms. Posicionarse en Sector 151 y 2101 aprox 16ms.

ALGORITMOS DE PLANIFICACIÓN DE DISCO

Definición: ¿por qué es importante planificar el disco? básicamente porque si los planificamos podemos tener mejor rendimiento que si no los planificáramos. El objetivo de la planificación es reducir el tiempo de búsqueda, que es el tiempo que tarda el brazo en posicionarse en la pista deseada.

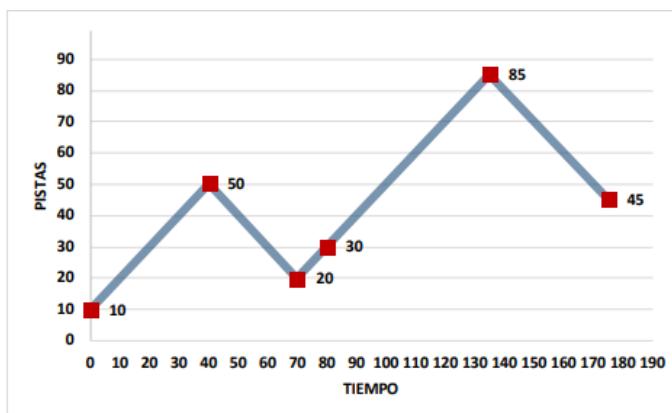
Estos algoritmos son:

- FCFS (FIFO).
- SSTF.
- SCAN y C-SCAN.
- LOOK y C-LOOK.
- FSCAN.
- N step SCAN.

FCFS (FIFO)

Ejemplo:

- Cantidad de pistas: 100. Tiempo entre pistas: 1ms.
- Pedidos de pista: 50, 20, 30, 85, 45.
- Posición inicial: pista 10



Tiempo Total:
175 ms

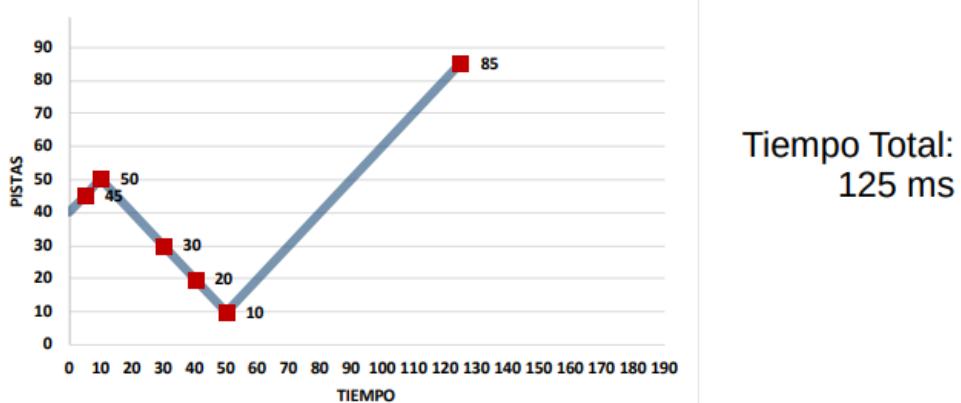
Para el ejemplo solamente tomamos en cuenta el tiempo de movimiento del brazo (TB).

SSTF (PRIMERO EL DE MENOR TIEMPO DE BÚSQUEDA)

Funcionamiento: va a buscar el pedido más cerca de donde está el cabezal sin importar el orden en que fueron hechos los pedidos. En este ejemplo no lo tenemos en cuenta pero este algoritmo también tiene en cuenta si el brazo venía subiendo o bajando (es decir, si hay dos pedidos que están igualmente cercanos se elige al que esté en la dirección en que esté yendo el brazo). Mejora el tiempo de espera promedio de los pedidos.

Ejemplo:

- Cantidad de pistas: 100. Tiempo entre pistas: 1ms.
- Pedidos de pista: 10, 50, 20, 30, 85, 45.
- Posición inicial: pista 40



Problemas:

- Starvation.

SCAN

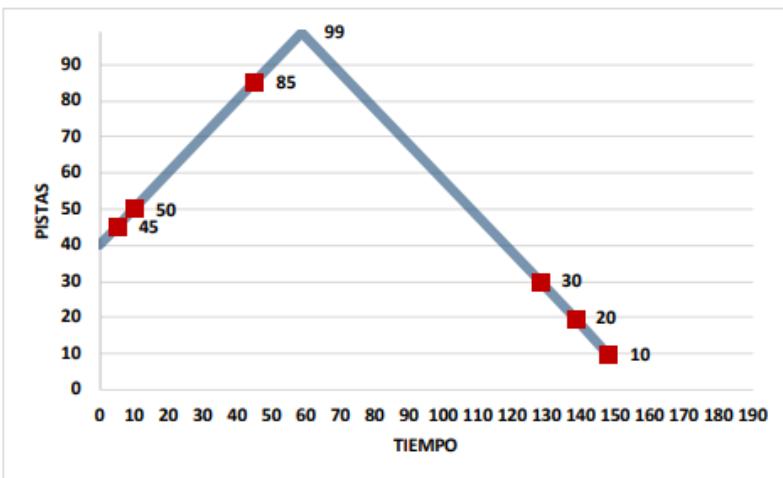
Funcionamiento: también llamado el algoritmo del “ascensor”. En este algoritmo el cabezal se mueve hacia arriba o abajo desde donde parte y va atendiendo los pedidos en orden de cercanía hasta llegar al tope máximo o mínimo y ahí cambia de dirección.

Es importante remarcar dos cosas:

- El brazo sube o baja dependiendo de si ya venía ascendiendo o descendiendo. Es decir, se debe contar con este dato. Si el cabezal está en la pista 20 y el brazo está ascendiendo, si llega el pedido 10, el brazo no va a poder bajar inmediatamente a atenderlo. Va a tener que subir hasta el tope y luego lo atenderá en la bajada.
- Es importante remarcar que el cabezal siempre llega hasta el tope máximo (o mínimo) por más que el pedido mayor o mínimo no sea el tope.

Ejemplo:

- Cantidad de pistas: 100. Tiempo entre pistas: 1ms.
- Pedidos de pista: 10, 50, 20, 30, 85, 45.
- Posición inicial: pista 40 y cabezal ascendiendo.



Tiempo Total:
148 ms

Problemas:

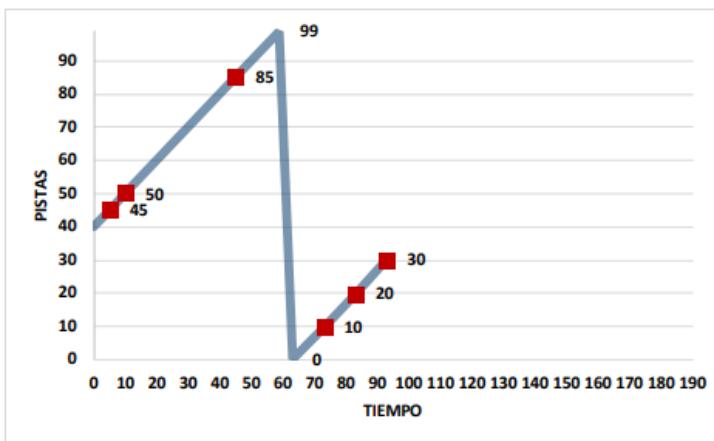
- Starvation: el starvation se da en un caso específico y es cuando estamos en una pista y se empiezan a recibir pedidos en esa misma pista. Supongamos que el cabezal está leyendo la pista 85 y le empiezan a llegar más pedidos de lectura de esa pista. El cabezal va a comenzar a atender esos pedidos y dejará de lado los otros.
Una forma de "salvar" esta situación sería que el algoritmo decidiera no atender pedidos nuevos en la pista donde esté atendiendo.

C-SCAN (CIRCULAR SCAN)

Funcionamiento: atiende pedidos solamente en un sentido, subida o bajada. En el ejemplo el brazo atiende pedidos únicamente en forma ascendente y cuando llega a la cima cae drásticamente hacia la primera pista (en un tiempo despreciable) y ahí vuelve a iniciar su recorrido en forma ascendente para atender pedidos.

Ejemplo:

- Cantidad de pistas: 100. Tiempo entre pistas: 1ms.
- Pedidos de pista: 10, 50, 20, 30, 85, 45.
- Posición inicial: pista 40 y cabezal ascendiendo.



Tiempo Total:
93 ms

Problemas:

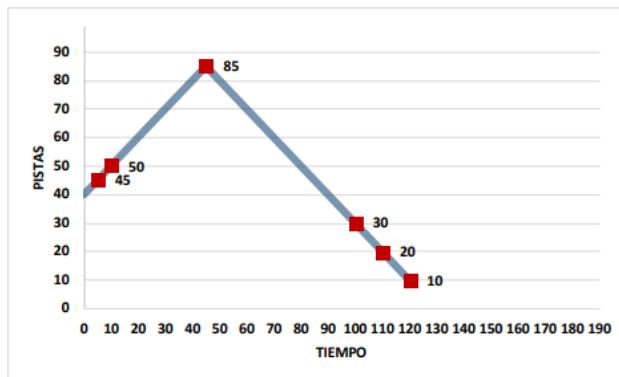
- Starvation.

LOOK

Funcionamiento: es muy similar al SCAN pero está optimizado porque no sube (o baja) hasta los topes, sino que va hasta la pista máxima solicitada.

Ejemplo:

- Cantidad de pistas: 100. Tiempo entre pistas: 1ms.
- Pedidos de pista: 10, 50, 20, 30, 85, 45.
- Posición inicial: pista 40 y cabezal ascendiendo.



Tiempo Total:
120 ms

Problemas:

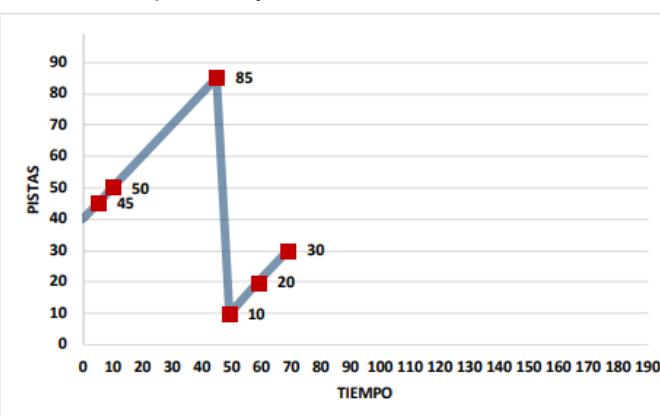
- Starvation.

C-LOOK (CIRCULAR LOOK)

Funcionamiento: misma lógica que el LOOK pero sólo se mueve en una dirección hasta el valor máximo solicitado y luego cae al valor mínimo solicitado (o sube al máximo).

Ejemplo:

- Cantidad de pistas: 100. Tiempo entre pistas: 1ms.
- Pedidos de pista: 10, 50, 20, 30, 85, 45.
- Posición inicial: pista 40 y cabezal ascendiendo.



Tiempo Total:
69 ms

Problemas:

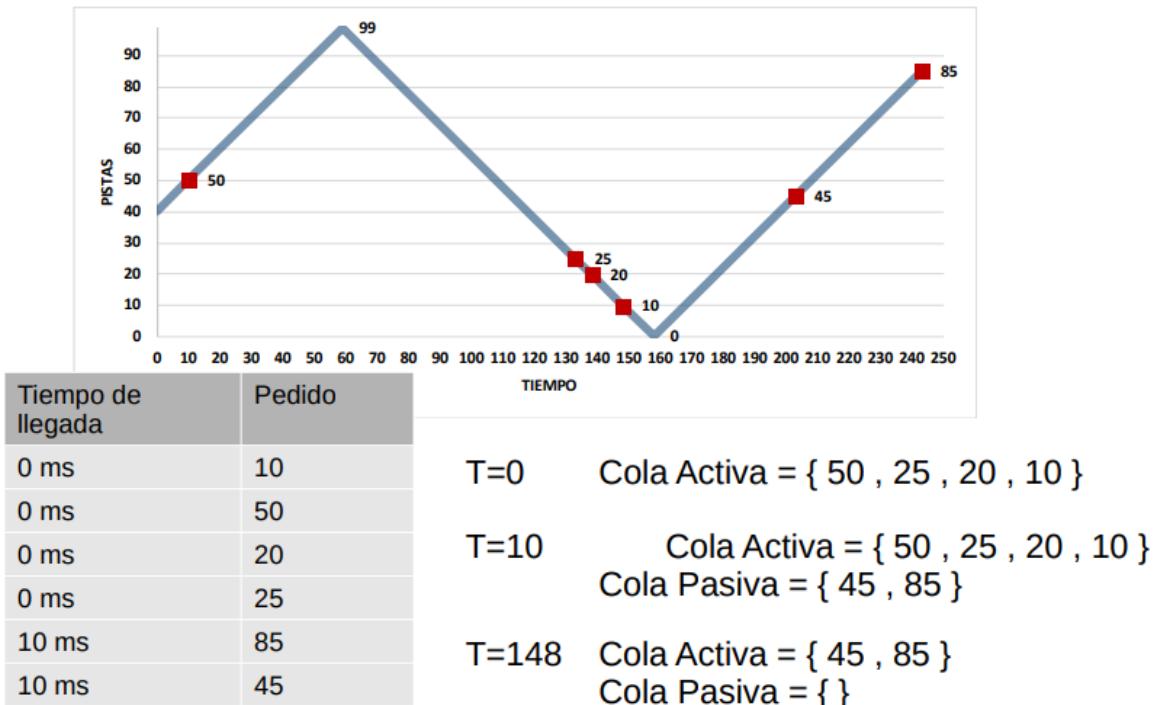
- Starvation:

FSCAN

Funcionamiento:

- En vez de manejar una lista de pedidos maneja dos listas de pedidos: Activa/Pasiva.
 - Activa: contiene los pedidos que está atendiendo en este momento.
 - Pasiva: están los pedidos que va a recibir mientras esté atendiendo los pedidos de la lista activa. Es como que se guardan estos pedidos y no los va a atender hasta que no termine con la otra cola.
- Se atienden los pedidos de la cola Activa utilizando el algoritmo SCAN.
- Los pedidos nuevos se agregan a la cola Pasiva.
- Cuando se atienden todos los pedidos de la cola Activa, la cola Pasiva pasa a ser Activa.

Ejemplo: si vemos el gráfico, este algoritmo tarda mucho más que los otros (250 ms). Pero tiene una ventaja: no tiene inanición porque los pedidos se dividen en diferentes listas.

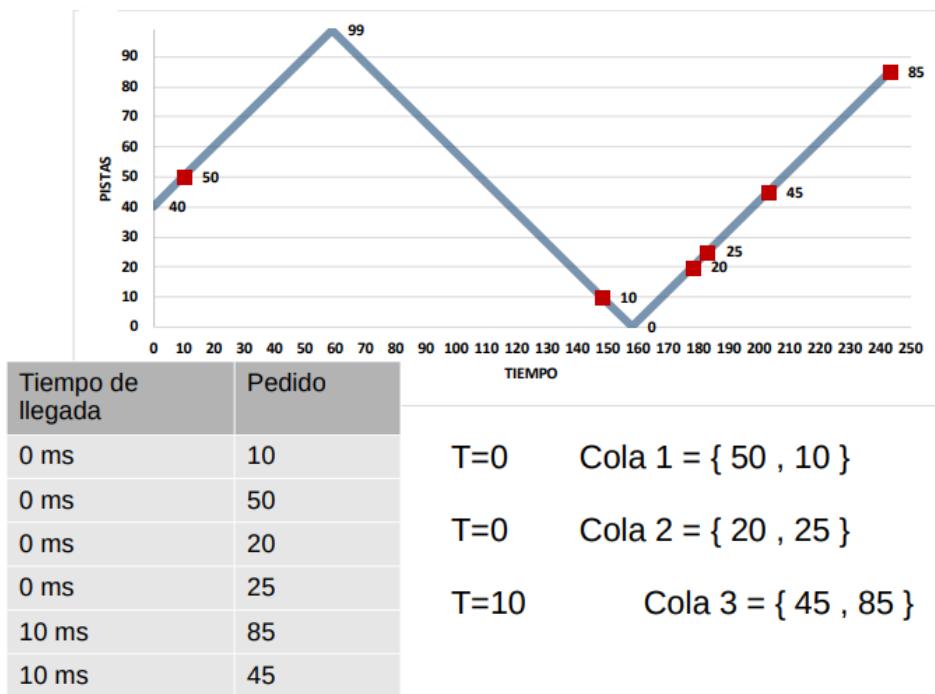


N step SCAN

Funcionamiento: es igual al anterior pero se pueden usar tantas colas como se necesiten.

- Utiliza colas de N pedidos o menos para acumular los pedidos.
- Los pedidos llegan y se los coloca en una cola. Si esa cola se llena se los comienza a ubicar en la siguiente.
- Cada cola se atiende utilizando el algoritmo SCAN.

Ejemplo: cabe aclarar que tardó lo mismo que el anterior de casualidad.



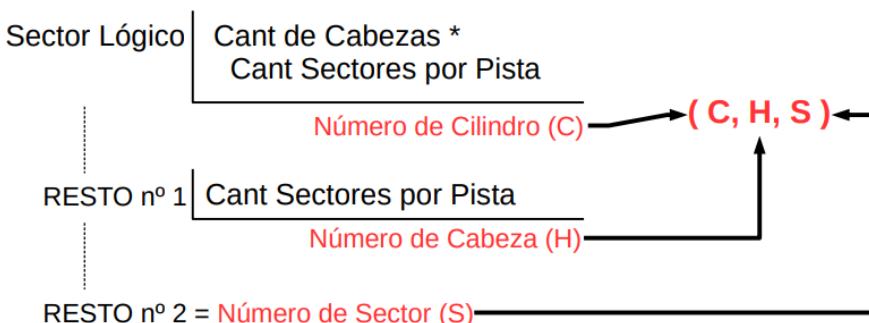
SECTOR LÓGICO A CHS

Concepto: en los ejercicios anteriores nos daban como dato la C (es decir, el cilindro, la pista en este caso) pero en vez de eso nos pueden dar el sector lógico (que obviamente va a ser un número mucho mayor).

¿Cómo convertir este sector lógico en CHS?: hay que hacer la siguiente cuenta:

- nro cilindro = sector lógico / (cant cabezas * cant sectores por pista)
= sector lógico / cant. sectores por cilindro
- nro cabezal = resto / cant sectores por pista
- nro sector = resto2

*cant. sectores por cilindro = caras (2 o 1) * sectores por pista * cant. discos*



Tamaño total de un disco:

Tamaño total de un disco = $C * H * S * \text{tam sector}$

- C: cant cilindros
- H: cant cabezas
- S: cant sectores por pista

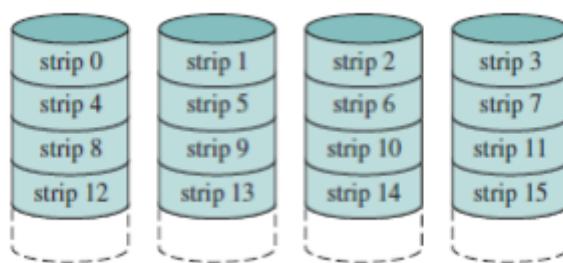
RAID (REDUNDANT ARRAY OF INDEPENDENT DISKS)

Definición: la idea básicamente es tener un conjunto de discos (un array de discos) en paralelo y guardar la información distribuida entre ellos así cuando se quiere hacer una lectura se puede leer cada disco en paralelo y así mejorar la performance.

El RAID puede estar implementado por el software o por el hardware. Si se implementa a nivel software lo maneja el SO y sino lo hace la controladora de RAID.

Niveles de RAID: ¿cómo se pueden distribuir los discos para obtener distintos resultados?

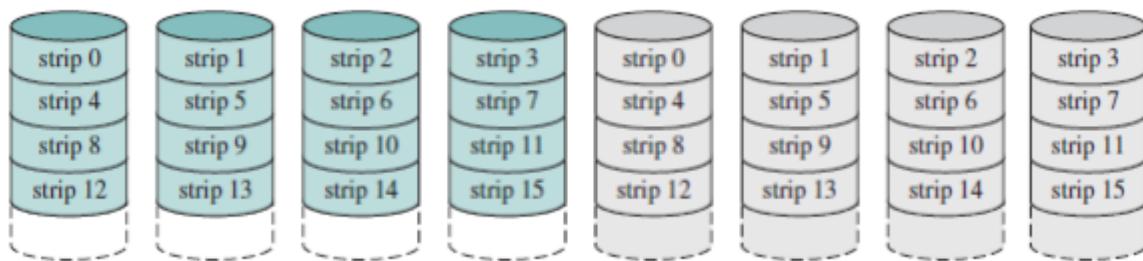
RAID 0: la idea es dividir la información de un único disco en bandas (strips) a lo largo de distintos discos. En el ejemplo vemos que hay 4 discos. Estos discos se leen o se escriben secuencialmente hacia la derecha (por eso la numeración strip 0, strip 1, etc). ¿Cuál es la idea? si tengo que leer, supongamos, el strip 0 y el strip 1, los voy a leer en paralelo.



Características:

- Redundancia: no tiene.
- Overhead: bajo, porque divide los archivos en bandas
- Tolerancia a fallos: no admite fallos.

RAID 0 + 1: al igual que el RAID 0, divide los discos en bandas pero además hace una copia de cada disco. Cada disco tiene otro que es una copia exacta de él (llamado "redundancia"). ¿Cuál es el objetivo de la redundancia? tener mayor tolerancia a fallos. Esto es así porque cuando uno tiene más discos, la probabilidad de fallo es mayor.

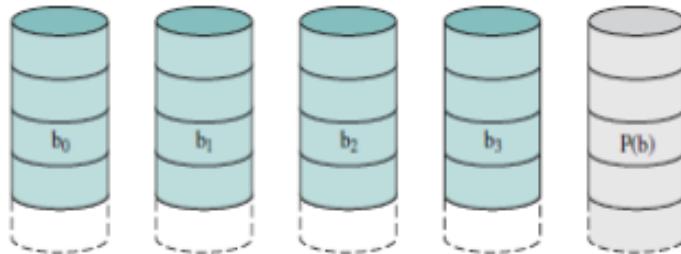


Características:

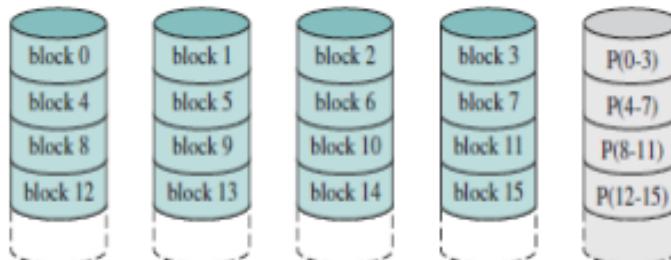
- Redundancia: tiene un disco redundante por cada disco (discos espejados).
- Overhead: bajo, divide los archivos en bandas y las espeja.
- Tolerancia a fallos: soporta la caída de una copia de cada disco.

RAID 3: este es distinto. Si bien los discos están divididos en strips, no se tiene una redundancia por cada disco. En vez de tener una copia por cada disco se guardan bits de paridad por cada conjunto de franjas de un mismo nivel. Es una forma de tener redundancia más compacta para que el sistema sea

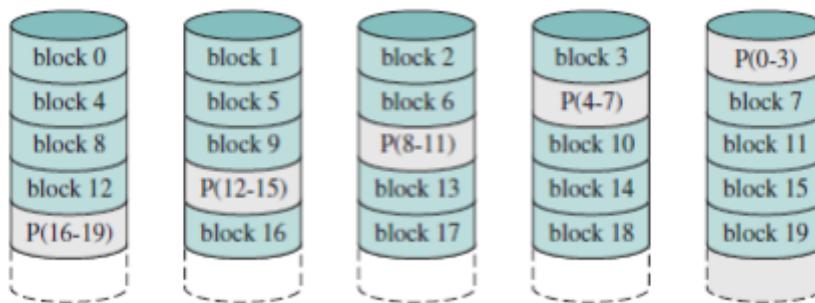
más económico. La limitación es que solamente un disco se puede romper (y reconstruir). Si se rompen dos discos ya no se pueden recuperar.



RAID 4:



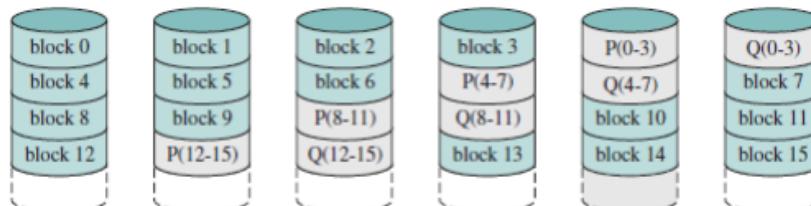
RAID 5: si bien tiene un único disco de redundancia (de paridad), la diferencia respecto a los demás es que este disco está distribuido en los demás. No tiene un disco aparte dedicado a la redundancia. El objetivo de esto es eliminar ese “cuello de botella” que se generaba en los otros esquemas dado que siempre que se realizaba una operación se debía replicar en el disco redundante. Se puede dañar hasta un disco.



Características:

- Redundancia: Incluye el equivalente a un disco de paridad, distribuido en todos los discos.
- Overhead: medio, porque divide a los archivos en bandas y calcula la paridad de cada banda.
- Tolerancia a fallos: soporta la caída de un sólo disco (cuálquiera).

RAID 6: es igual al RAID 5 pero cuenta con dos discos de redundancia distribuidos en los demás. En este caso se puede reconstruir hasta dos discos.



RESUMEN DE ALGORITMOS

Nombre	Cómo determina prioridad	¿Tiene en cuenta la dirección del brazo?	Topes	Cant. colas	Aging/Inanición/Starvation/
FIFO	Se atienden los pedidos de pista en el orden en que llegan.	No.		1	<u>Starvation</u> : no.
SSTF (primero el de menor tiempo de búsqueda)	En cada momento va a buscar el pedido más cerca de donde está el cabezal sin importar el orden en que fueron hechos los pedidos.	No. Solamente la tiene en cuenta en caso de haber dos pedidos igualmente cercanos. Se elige al que esté en la dirección en que esté yendo el brazo.		1	<u>Starvation</u> : sí. Si continuamente llegan pedidos cercanos a donde está el brazo, los pedidos de pistas más lejanas serán ignorados.
SCAN (ascensor)	El cabezal se mueve hacia arriba o abajo desde donde parte y va atendiendo los pedidos en orden de cercanía (no de llegada) hasta llegar al tope máx o mín y ahí cambia de dirección.	Sí. El cabezal sube o baja dependiendo de si ya venía ascendiendo o descendiendo.	Llega hasta el tope máximo o mínimo y ahí cambia de dirección para atender los pedidos.	1	<u>Starvation</u> : sí.
C-SCAN	Atiende pedidos solamente en un sentido (subida o bajada) cayendo o subiendo drásticamente al tope opuesto al completar una corrida.	No. Siempre va en un sentido.	Llega hasta el tope máx antes de caer al mín o al revés.		
LOOK	El cabezal se mueve hacia arriba o abajo desde donde parte y va atendiendo los pedidos en orden de cercanía (no de llegada) hasta llegar al pedido máx o mín y ahí cambia de dirección.	Idem SCAN.	No. Va hasta las pistas máximas o mínimas solicitadas.	1	<u>Starvation</u> : sí.
C-LOOK	Misma lógica que el LOOK pero sólo se mueve en una dirección hasta el valor máximo solicitado y luego cae al valor mínimo solicitado (o sube al máximo).	Idem C-SCAN.	Llega hasta el pedido máx antes de caer al pedido mín o al revés.		

FSCAN	<p>Se manejan dos listas de pedidos: Activa/Pasiva.</p> <p><u>Activa</u>: están los pedidos que está atendiendo en este momento.</p> <p><u>Pasiva</u>: están los pedidos que va a recibir mientras esté atendiendo los pedidos de la lista activa.</p> <p>1) Se atienden los pedidos de la cola Activa utilizando el algoritmo SCAN.</p> <p>2) Los pedidos nuevos se agregan a la cola Pasiva.</p> <p>3) Cuando se atienden todos los pedidos de la cola Activa, la cola Pasiva pasa a ser Activa.</p>	Idem SCAN.	Idem SCAN.	2 (Activa y Pasiva)	<u>Starvation</u> : no.
N step SCAN	<p>Es igual al anterior pero se pueden usar tantas colas como se necesiten.</p> <p>1) Los pedidos llegan y se los coloca en una cola. Si esa cola se llena se los comienza a ubicar en la siguiente.</p> <p>2) Cada cola se atiende utilizando el algoritmo SCAN.</p>			x colas de N elementos	

PREGUNTAS DE PARCIAL

1. Explique las diferencias entre E/S síncronas y asíncronas. ¿Cuál es la función de los buffers en cada caso? Justifique.

En las E/S síncronas el proceso necesita la respuesta para continuar con su ejecución, mientras que en las E/S asíncronas el proceso puede continuar con su ejecución mientras espera la respuesta de la E/S.

Para el primer caso, el buffer simplemente será llenado y vaciado cuando se realice la operación. Para el segundo, el buffer puede llenarse durante una cantidad de tiempo, y el SO puede vaciarlo cuando lo considere oportuno, mejorando la performance.

2. ¿Qué métrica buscan optimizar los algoritmos de planificación de disco? Elija uno de los algoritmos y explique por qué lo optimiza. Elija otro algoritmo (no FIFO) que priorice un poco más el orden de llegada que el anterior y explique por qué.

Los algoritmos de planificación de disco buscan optimizar el tiempo de búsqueda, es decir, el tiempo entre pistas. El SSTF lo optimiza porque mejora el tiempo de espera promedio de los pedidos. NSTEP SCAN privilegia el orden de llegada, porque arma colas de hasta N pedidos, perdiendo un poco de performance pero evitando inanición.

3. Compare los algoritmos SSTF, FSCAN y N-step-SCAN en términos de tiempo de búsqueda y equidad. ¿Cuál de estos algoritmos y de qué manera podrían sufrir inanición? Ejemplifíquelo.

Algoritmo	Tiempo de búsqueda	Equidad	Inanición
SSTF	El tiempo de búsqueda promedio tiende a ser menor dado que siempre se atiende la petición de la pista que esté más cercana.	Baja	Sí. Por ejemplo, si el cabezal está en la pista 85 y recibe más pedidos para esa pista, los atenderá e ignorará a las demás.
FSCAN	Alto.	Media	No.
N STEP SCAN	El tiempo de búsqueda es mayor al de los demás algoritmos pero asegura que todos los pedidos serán atendidos.	Alta	No.

4. V o F

- a. Si un proceso requiere la respuesta de una E/S para continuar probablemente quien lo programe la realice en forma asíncrona.

Falso. La realizará de forma síncrona.

- b. Tanto en FSCAN como en N-STEP-SCAN evitan el problema de la inanición.

Verdadero. Como los pedidos son agregados a listas y estas listas se atienden en orden, garantiza que todos los pedidos van a ser atendidos.

5. Todos los algoritmos de disco pueden sufrir inanición.

Falso. Los algoritmos FIFO, FSCAN y N-STEP-SCAN no sufren inanición.

6. Diferencias y similitudes entre los algoritmos N-STEP-SCAN y F-SCAN. ¿Qué tienen en particular estos algoritmos respecto al resto?

Diferencias: N step usa x colas de N elementos mientras que FSCAN usa 2 colas.

Similitudes: no tienen inanición. Atienden las colas con el algoritmo SCAN.

Se diferencian de los demás algoritmos (menos FIFO) por no generar inanición y garantizar que todos los pedidos van a ser atendidos.

7. Indique un caso en el que sería útil realizar una E/S asíncrona. ¿Cómo implementaría una E/S síncrona no bloqueante?

Las E/S asíncronas son útiles cuando el proceso no necesita la respuesta inmediatamente y puede realizar otras tareas mientras la espera. Para implementar una E/S síncrona no bloqueante se podría usar una espera activa que pregunte constantemente si la respuesta está disponible.