Resumen sistema operativos

Certamen 2

Deadlocks

Definiciones

en resumen un deadlock es cuando un grupo de hebras espera una acción que no va a ocurrir, donde cada hebra retiene el recurso y espera obtener un recurso obtenido por otra hebra

¿Qué debe ocurrir para que ocurra el DeadLock?

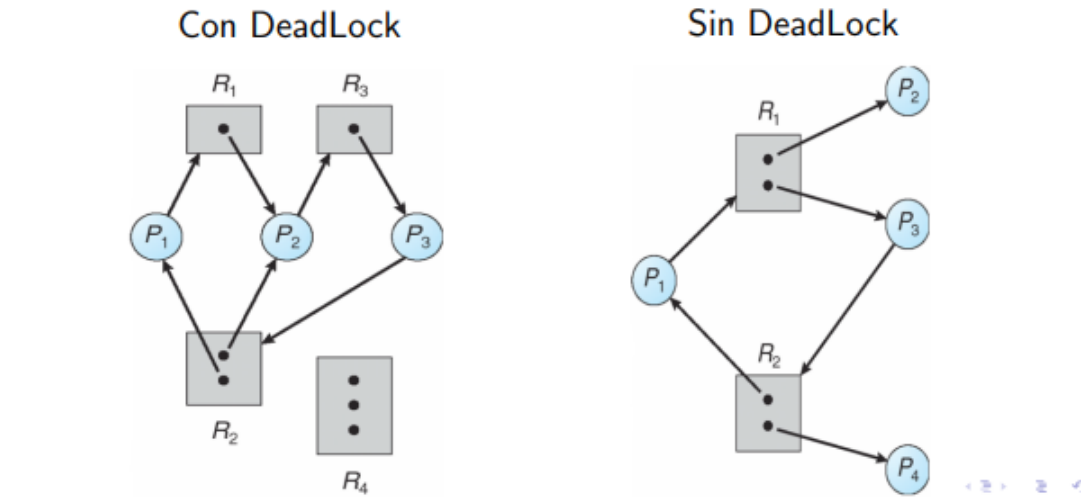
Hebra 1	Hebra 2
`lock1.aquire();`	`lock2.aquire();`
`lock2.aquire();`	`lock1.aquire();`
`lock2.release();`	`lock1.release();`
`lock1.release();`	`lock2.release();`

Deadlocks y starvation(inanición)

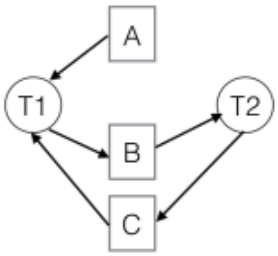
En la inanición una hebra queda detenida por un tiempo indefinido.
Un deadlock se considera una inanición especial ya que es imposible continuar.
En el caso de los RWLocks, puede producirse inanición de un lector, si llegan muchos escritores.
Existe la posibilidad de que a la hora de probar un programa no se presente ningún síntoma por eso es necesario tratar de diseñar el software aprueba de estos errores.

Condiciones necesarias para el deadlock

1. Recursos limitados: número finito de hebras que comparten el acceso a recursos finitos.
2. No se pueden quitar: cuando una hebra adquiere el recurso no se le puede quitar externamente.
3. Espera y retención: Una hebra retiene un recurso, mientras que otro se desocupe.
4. Espera circular: Existe un conjunto de hebras que esperan y cada hebra espera que se libere un recurso que tiene otra hebra del grupo.



En resumen hay que ver que hay dos tipos distintos de flechas, de recurso a proceso es que este proceso tiene ese recurso, y de proceso a recurso es que este proceso espera este recurso, el deadlock ocurre cuando se forma un ciclo de espera, puede haber ciclos sin que sea deadlock siempre que no sean de espera.



	Hebra 1	Hebra 2
1	Obtiene A	
2		Obtiene B
3	Obtiene C	
4		Espera C
5	Espera B	
	...	...

- Si bien el DeadLock ocurre en el paso 5, podemos ver que a partir del paso 2 comienzan a configurarse las circunstancias que nos llevan al problema.

Prevención de deadlocks

1. limitar el comportamiento

- Podemos proporcionar mas recursos
- Eliminar la retención y espera
- Eliminar la espera circular

2. Pronosticar el futuro

- Se utiliza el algoritmo del banquero
  - Se determinan los valores máximos disponibles en el sistema para cada recurso.
  - se realiza una comparación entre los recursos solicitados y los disponibles.
  - En el caso que se dispongan todos los recursos solicitados, se otorgan(se marcan como utilizados) para garantizar que la tarea se lleve a cabo
- No se presta mas dinero del que se dispone.

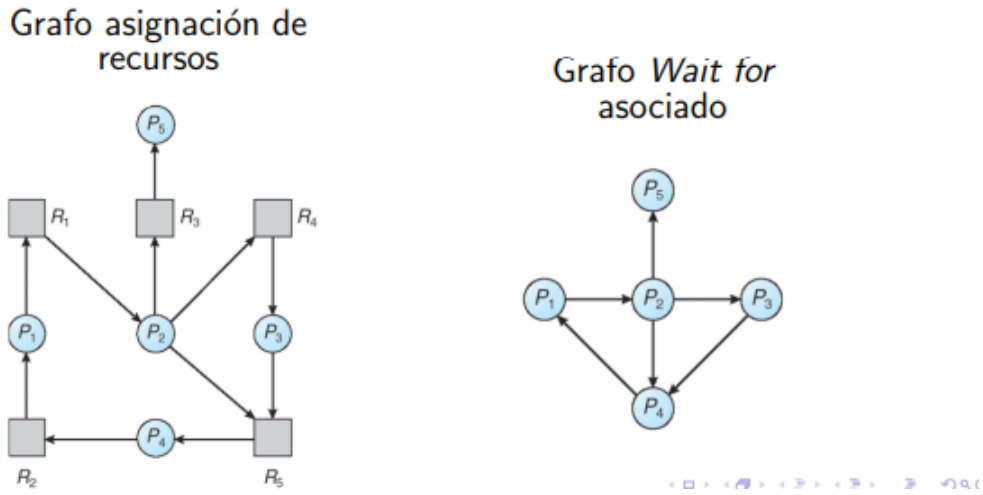
3. Detección y reparación

- Podemos usar grafos para detectar deadlocks y repararlos

4. Suponer que no van a ocurrir

Detectar y recuperar: algoritmo Wait-for

Invocar un algoritmo que busca ciclos en el grafo, si detecta un ciclo, entonces hay un deadlock



Algoritmo del banquero para evitar deadlocks

Una opción es esperar a que existan disponibles todos los recursos solicitados y cuando esta condición se cumpla asignarlos todos atómicamente.
Una hebra que comienza a ejecutarse establece los requerimientos máximos de recursos, pero los adquiere y libera de manera incremental.
La idea es intercalando recursos.

Estados de un proceso

1. Estado seguro: Existe al menos una secuencia segura de procesar el requerimiento, puede requerir espera.
2. Estado inseguro: Existe al menos una secuencia que lleven al deadlock
3. Deadlock: El sistema tiene al menos un deadlock

- El Algoritmo del Banquero **mantiene** al sistema en un estado seguro. Lo hace al entregar el recurso si y solo si hacerlo mantiene al sistema en un estado seguro.
- La suma de los "Requerimientos máximos" de las Hebras del sistema puede ser mayor al total los recursos. Siempre que exista una manera de que las Hebras puedan terminar sin caer en un deadlock.
- Procede con la asignación de recursos si y solo si:
  - Max_{hebra} es la cantidad de recursos que requiere la Hebra para terminar.
  - $R_{disponibles} - R_{asignados} \geq Max_{hebra}$

Ejemplo

Un sistema tiene 8 páginas de memoria disponibles y 3 procesos: A, B, C que necesitan 4, 5, 5 páginas respectivamente para terminar.

Proc	Asignación																		
A	0	**1**	1	1	**2**	2	2	**3**	3	3	**4**	**0**	0	0	0	0	0	0	0
B	0	0	**1**	1	1	**2**	2	2	W^2	W	W	W	**3**	**4**	4	**5**	**0**	0	0
C	0	0	0	**1**	1	1	**2**	2	2	W	W	W	**3**	3	W	W	**4**	**5**	**0**
Total	0	1	2	3	4	5	6	7	7	7	8	4	6	7	7	8	4	5	0

cuando un proceso se pone en wait, guarda los recursos que se le asignaron antes de ponerse en ese estado.
y si vemos las paginas crecen en diagonal.

Detección y recuperación de deadlocks

Algunos sistemas permiten los deadlocks ya que si son raros, no vale la pena gastar recursos en detectarlos.
Existen dos aproximaciones:

1. Proseguir sin el recurso: ya estará disponible.
2. Transacciones rollback/retry: revoca los recursos previamente dado y luego se vuelve a ejecutar, es mas lento.

Scheduling

Introducción

- Scheduling: Decide qué tarea se ejecuta en el procesador durante un cambio de contexto.
- Se aplica en recursos escasos: CPU, disco, red, energía.
- Impacto real: 100ms extra en respuesta → pérdida del 5%-10% de clientes (Google/Amazon).

Definiciones básicas

- Tarea (task/job): Requerimiento del usuario.
- Tiempo de respuesta: Tiempo percibido por el usuario para completar la tarea.
- Predictibilidad: Baja varianza en tiempos de respuesta.
- Throughput: Tasa de ejecución de tareas.
- Overhead: Costo de cambiar de tarea.
- Equidad: Igualdad en recursos.
- Starvation: Tarea no avanza por falta de recursos.
- Workload: Conjunto de tareas con tiempos de llegada y duración.
- Preemptive: Recursos pueden ser quitados.
- Conservador de trabajo: Nunca deja CPU ociosa si hay tareas.

Criterios de optimización

- Maximizar utilización CPU.
- Maximizar throughput.
- Minimizar tiempo de espera.
- Minimizar tiempo de respuesta.

Perfiles de carga

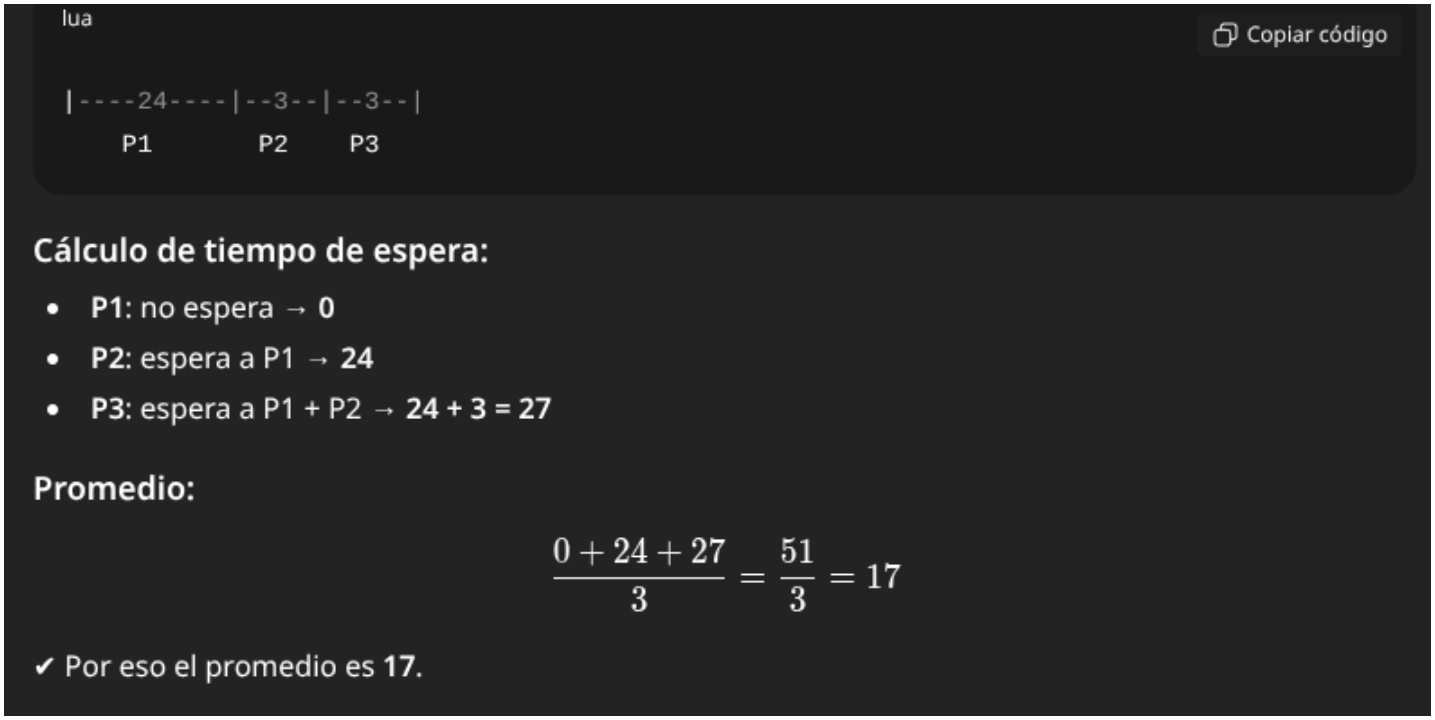
- CPU bound: Uso intensivo de CPU (ej: cálculos numéricos).
- I/O bound: Espera E/S (ej: descarga de archivo).
- Mixto: Compilador, navegador, juegos.

Algoritmos de planificación en uniprocesador

FIFO (First In First Out / FCFS)

- Procesos se ejecutan en orden de llegada.
- Ventajas: Bajo overhead, buen throughput, equidad.
- Desventaja: Problema con tareas largas seguidas de cortas → alto tiempo de espera promedio.
- Ejemplo:

- Orden P1(24), P2(3), P3(3) → Espera promedio = 17.



- Reordenado P2, P3, P1 → Espera promedio = 3.

Línea de tiempo:

lua — Copiar código

```
|--3--|--3--|---------24---------|
   P2    P3           P1
```

Cálculo:

- **P2**: espera 0 → **0**
- **P3**: espera a P2 → **3**
- **P1**: espera a P2 + P3 → **3 + 3 = 6**

Promedio:

$$\frac{0+3+6}{3} = \frac{9}{3} = 3$$

✔ Por eso el promedio es **3**.

SJF (Shortest Job First)

- Elige la tarea con menor tiempo restante.
- Ideal para minimizar tiempo de respuesta.
- Problemas:

- Starvation para tareas largas.

- Difícil saber tiempo restante.

- Más cambios de contexto si llegan muchas tareas cortas.

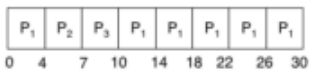
Round Robin (RR)

- Cada tarea recibe un quantum (q).
- Si no termina en q → se interrumpe y pasa a la siguiente.

Supongamos que se tienen las siguientes tareas P_1, P_2 y P_3 y que nuestro q=4:

Proceso	Tiempo de CPU
P_1	24
P_2	3
P_3	3

Será ejecutada según el siguiente orden:



- Ventajas: Sin inanición.
- Elección de q:

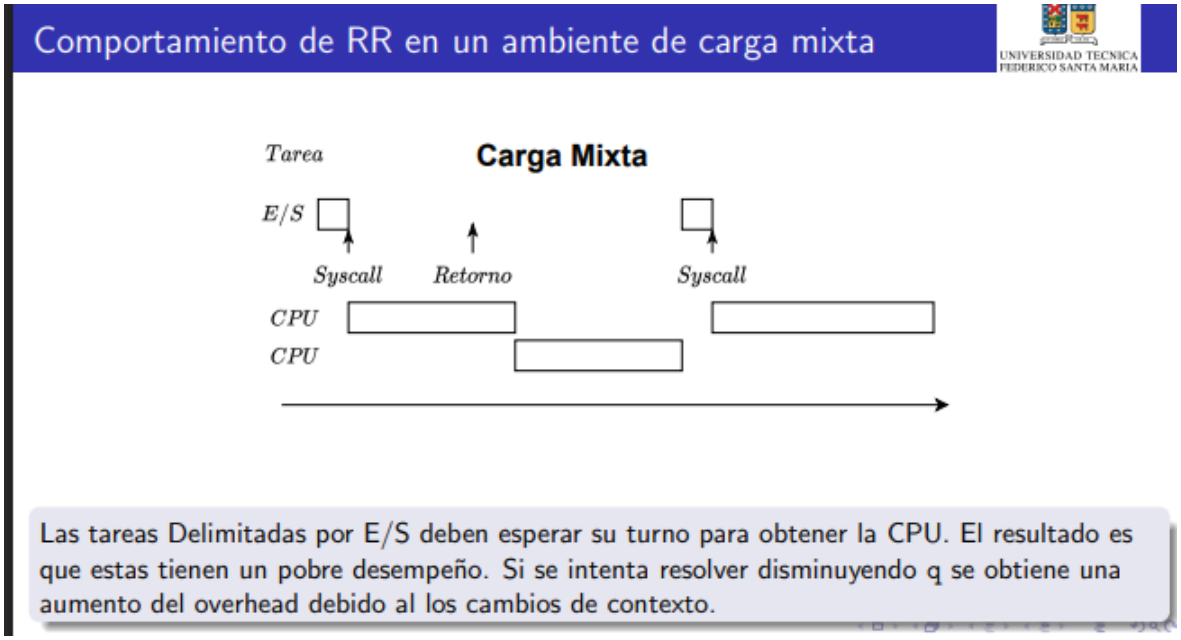
- q → 0 → alto overhead.

- q → ∞ → se comporta como FIFO.

- RR es punto medio entre FIFO y SJF.
- Problemas:

- Mal desempeño con cargas mixtas (CPU + I/O).

- Tareas I/O bound pueden esperar demasiado.



Equidad Min-Max

- Garantiza mínimo de recursos a cada proceso.
- Si todos son CPU bound → RR.
- Si hay I/O bound → se les da lo que necesitan, resto se reparte.
- Implementación: Elegir tarea con menor tiempo acumulado.

Colas Multinivel con Retroalimentación (MFQ)

- Usado en sistemas modernos (Windows, Linux, MacOS).
- Objetivos:

- Buen tiempo de respuesta (similar a SJF).

- Bajo overhead (similar a FIFO).

- Sin inanición (similar a RR).

- Características:

- Varias colas con distinta prioridad y quantum.

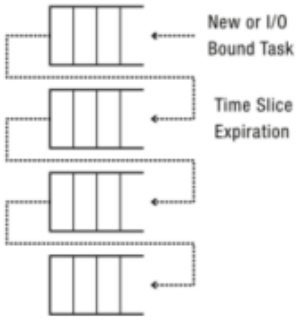
- Alta prioridad → quantum pequeño.

- Tareas bajan prioridad si no terminan en su quantum.

- Nivel más bajo puede ser FIFO o RR.

- Operaciones de E/S pueden subir prioridad.

Priority	Time Slice (ms)
1	10
2	20
3	40
4	80



- Problemas:

- No garantiza equidad Max-Min.

- Puede haber inanición si muchas tareas I/O bound.

- Solución: Ajuste dinámico de prioridades (ej: Linux).

Planificación en multiprocesadores

- Desafíos:

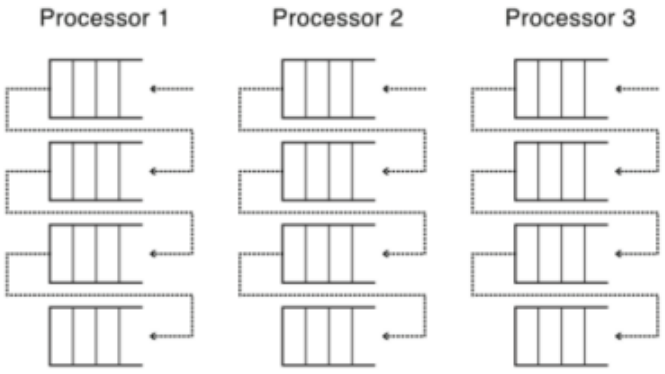
- ¿Cómo usar múltiples CPUs para tareas secuenciales?

- ¿Cómo adaptar algoritmos para tareas paralelas?

- Estrategias:

- MFQ por procesador (evita contención).

Los Sistemas Operativos modernos tienen una cola MFQ por cada procesador.



- Afinidad de procesador: Mejora uso de caché.

- Robo de tareas: CPUs ocupados pueden tomar tareas de otros (penalización posible).

Casos prácticos

- Servidor Web:

- Muchas hebras E/S bound → favorecer tareas cortas.

- MFQ centralizado puede causar contención → mejor MFQ por CPU.

Caso: Servidor Web

UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

- Los servidores web normalmente tienen una gran cantidad de solicitudes por parte de los clientes:
  - Una Hebra por cada conexión de cliente
  - Cada Hebra accede a un objeto compartido para verificar si el elemento solicitado ya se encuentra en memoria (cache).
  - La Hebra solicita que se lea el elemento desde el disco en caso que no se encuentre.
  - La Hebra responde al cliente lo que solicita.
  - ¿Que algoritmo de planificación se debe utilizar?
- Cada Hebra es E/S bound. Esto significa que para tener buenos tiempos de respuesta se deben favorecer las tareas cortas.
- Una posibilidad es manejar una cola MFQ centralizada, pero esto puede traer problemas de contención (locks) y altos overheads para mantener un cache coherente.

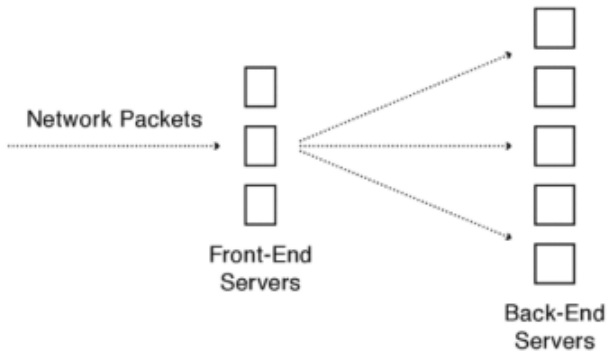
- Datacenter:

- Balanceo de carga entre back-ends.

- Afinidad para mantener coherencia.

- Predicción para asignar recursos.

Suponga un servicio realmente grande en un datacenter. Ej: Facebook, Google, Amazon.



Traducción de Direcciones (Memoria Virtual)

Introducción

- Objetivo: Crear una realidad virtual para los programas.
- El programa no debe preocuparse por dónde está almacenado.

- Conversión de direcciones virtuales → físicas: concepto simple pero poderoso.
- Separa espacio de direcciones físicas del espacio virtual.

¿Qué puede hacer el SO con memoria virtual?

- Aislar procesos: Protección de espacios de memoria.
- Comunicación entre procesos: Compartir memoria de forma transparente.
- Compartir código: Reduce huella de memoria.
- Inicialización parcial: Ejecutar sin cargar todo en memoria.
- Asignación dinámica: Expandir espacios de forma transparente.
- Manejo de caché: Coloreo de páginas.
- E/S eficiente: Transferencias seguras.
- Archivos memory-mapped: Mapear archivos al espacio de direcciones.
- Checkpoint & restart: Reanudar tras falla.
- Migración de procesos: Balanceo de carga.
- Control de flujo: Seguridad adicional.
- Memoria compartida distribuida: Sistemas paralelos.

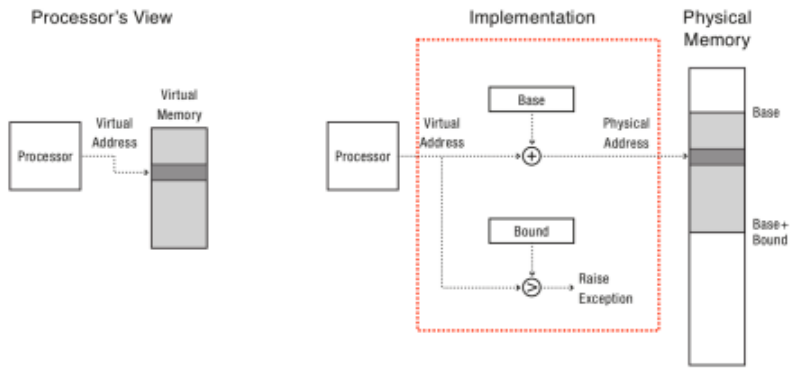
Objetivos de la traducción de direcciones

- Protección: No toda memoria accesible por todos.
- Compartir memoria: Reduce presión del recurso.
- Ubicación flexible: Procesos en cualquier parte.
- Dispersión: Regiones dinámicas.
- Eficiencia: Conversión rápida (cada instrucción).
- Tablas compactas: Bajo overhead.
- Portabilidad: Independencia de arquitectura.

Métodos de traducción

Base y Límite

- Dos registros: `base` y `bound`.
- Direcciones virtuales comienzan en 0; físicas = base + offset.
- Ventajas: Protección general.
- Desventajas: Difícil compartir memoria.

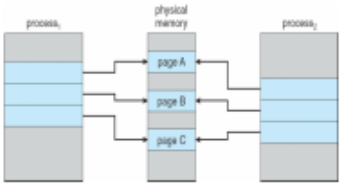


Segmentación

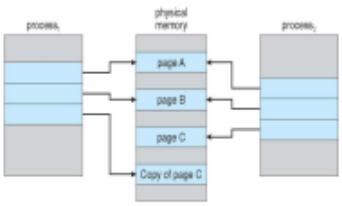
- Vector de pares (base, bound) → múltiples segmentos.
- Cada segmento contiguo en espacio virtual, disperso en físico.
- Permisos por segmento.
- Falla de segmentación: Referencia fuera de segmento.
- Compartir segmentos entre procesos.
- Copy on Write (CoW):

- Fork copia tabla de segmentos.

- Se copia la tabla de segmentos al hijo.
- Se marcan los segmentos del padre y el hijo como readonly (solo lectura). -
- Se activa el proceso hijo recién creado. Se retorna desde fork en el padre.
- Si el padre o el hijo intentan modificar datos en sus segmentos, se genera un trap en el kernel. En este caso, señaliza al kernel que haga una copia del segmento para el proceso que desea escribir, al retornar el control.
  - Marca readonly; si se escribe → trap → copia real.



Antes que el Proceso 1 modifique la página o segmento C



Después que el Proceso 1 modifique la página o segmento C

- Zero on Reference:

- Limpia memoria bajo demanda.

- Permite iniciar procesos con heap/stack en 0.

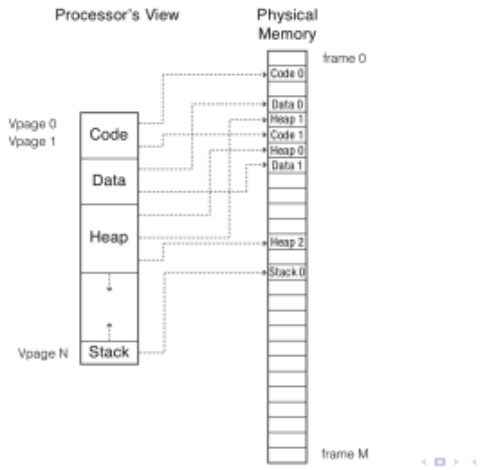
- Limitaciones de la segmentación:

- Overhead por muchos segmentos.

- Fragmentación externa → compactación.

Paginación

- Memoria en frames fijos.
- Conversión mediante tabla de páginas.
- Espacio virtual continuo, físico disperso.
- Mapa de bits para frames libres.



- Carga bajo demanda:

- Páginas inválidas → trap → carga → marcar válida.

- Core Map: Registra qué procesos usan cada frame.
- Limitaciones:

- Tabla proporcional a memoria virtual.

- Fragmentación interna si frames grandes.

Traducciones multinivel

- Reemplaza tablas por árboles.
- Ventajas:

- Asignación eficiente (mapa de bits).

- Transferencias disco eficientes.

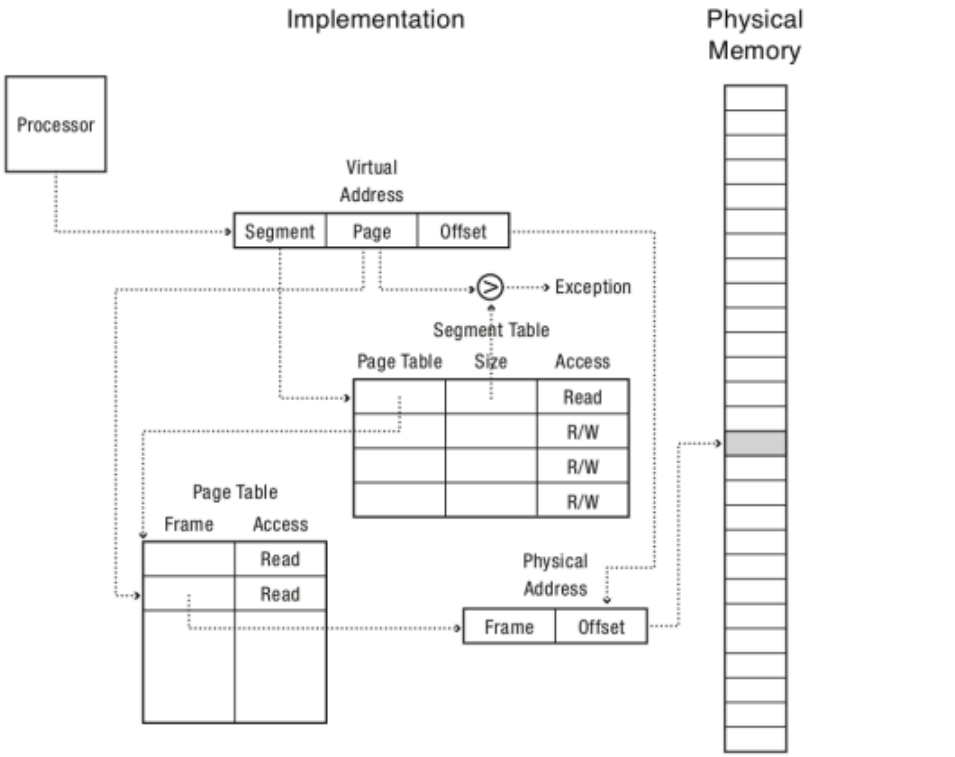
- Conversión rápida.

- Búsqueda inversa eficiente (Core Map).

- Granularidad en permisos.

- Implementaciones:

- Segmentación paginada.

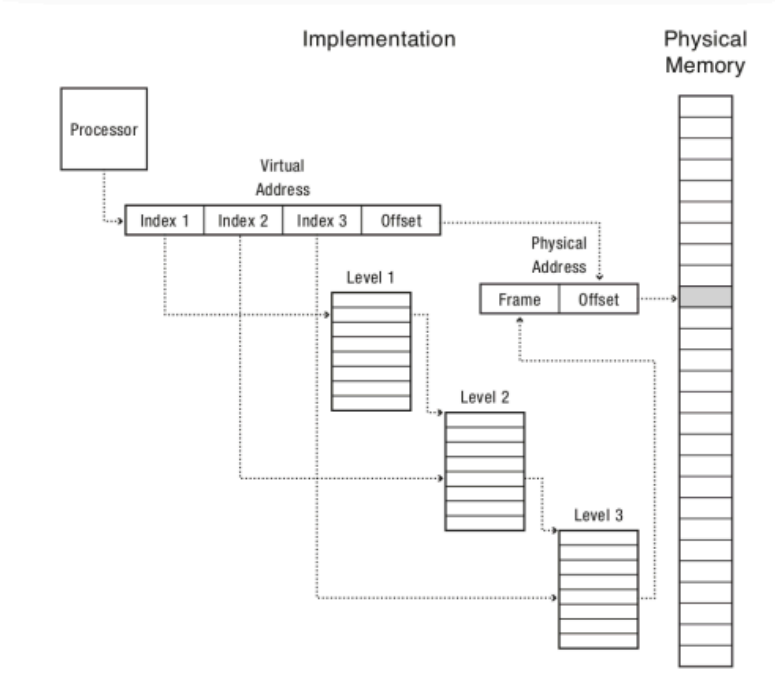


- Paginación multinivel.

- Segmentación paginada multinivel (x86).

Implementación de Paginación Multinivel

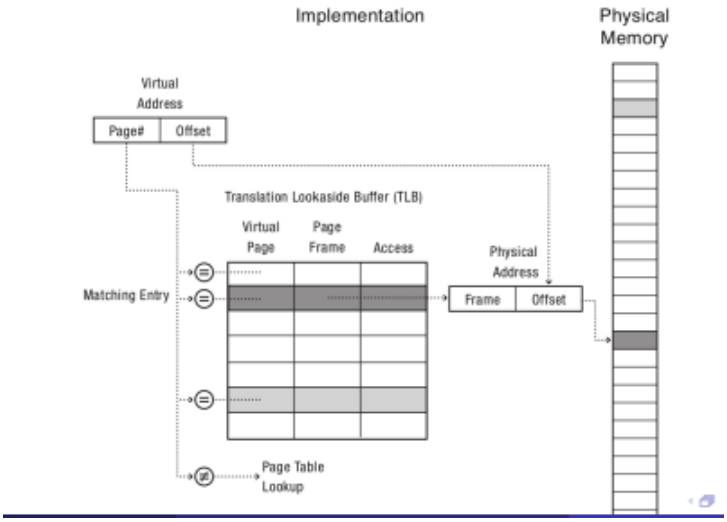
UNIVERSIDAD TEC
FEDERICO SANTA M



Eficiencia: TLB (Translation Lookaside Buffer)

```
TLB entry = {
N° de página virtual,
N° de frame físico,
permisos de acceso
}
```

- Caché en hardware para traducciones recientes.
- Entrada TLB: {Página virtual, Frame físico, Permisos}.
- TLB hit: Traducción rápida.
- TLB miss: Conversión completa.



- Costo:$Costo = Costo(TLBlookup) + Costo(conversionreal) * (1 - P(Hit))$
- Alta P(Hit) = eficiencia.

Sistemas de Archivos

Introducción

- Objetivos del sistema de archivos:

- Confiabilidad: Datos intactos incluso ante fallas.

- Alta capacidad y bajo costo.

- Alto desempeño: Acceso rápido.

- Espacio de nombres: Organización clara.

- Seguridad: Control de acceso.

- Persistencia:

- RAM es volátil; almacenamiento no-volátil (HDD/SSD) más lento pero persistente.

- Acceso por bloques (512B a 4KiB).

- HDD: ~10ms vs RAM: ~10ns.

Problemas de desempeño

- Archivos grandes con auto-guardado:

- Desempeño bajo: Reescribir completo por cambios pequeños.

- Corrupción: Fallo durante escritura → inconsistencia.

- Pérdida total: Si se sobrescribe original y falla.

Abstracción del sistema de archivos

- Definición: Conjunto de datos (archivo) + nombres (directorios).
- Ejemplo: `/home/javier/Documentos/Filesystem.pdf`.
- Metadata: Tamaño, fecha, permisos.
- Datos = arreglo de bytes sin tipo (interpretación por el proceso).

Directorios y volúmenes

- POSIX:

- `/` raíz.

- `.` directorio actual.

- `..` padre.

- `~` home usuario.

- Volumen:

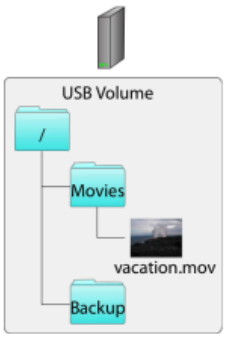
- Recurso lógico que agrupa almacenamiento físico.

- Puede ser un disco, partición o conjunto de discos.

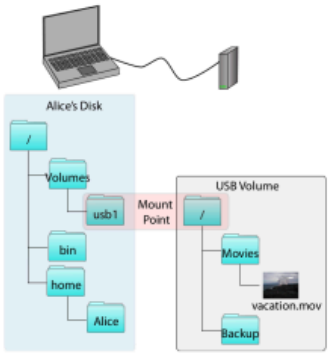
- Montaje:

- Volumen debe montarse en un punto del sistema (`/media/disk-1`).

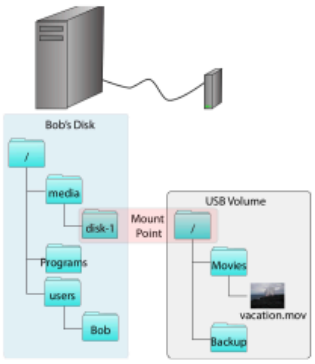
- Los volúmenes deben ser *montados* en alguna ubicación dentro de un sistema de archivos para poder tener acceso a su contenido.



Contenido del USB

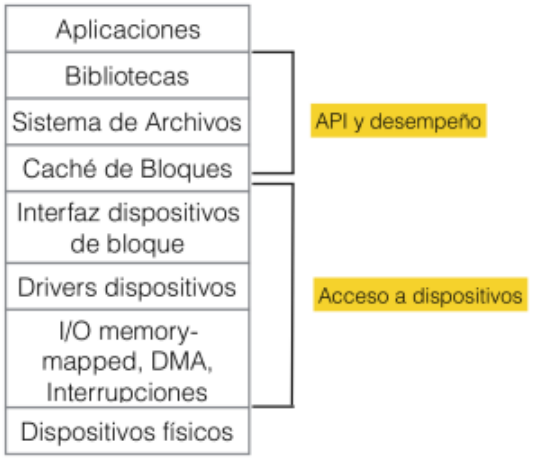


Volúmen montado en `/Volumes/usb1`



Volúmen montado en `/media/disk-1`

Arquitectura del sistema de archivos



- Cache de bloques:

- Mantiene bloques en RAM para optimizar lecturas/escrituras.

- Prefetching: Leer anticipadamente bloques próximos.

- Dispositivos físicos:

- HDD: Sectores 512B, seek ~12ms, rotación 7200RPM.

- SSD: Celdas 128KB-512KB, borrado antes de escribir, desgaste limitado.

- Comparativa HDD vs SSD:

- HDD: Mejor costo/capacidad, peor velocidad aleatoria.

- SSD: Mejor velocidad, menor tolerancia a apagado prolongado.

Archivos y directorios: desafíos

Comparativa

UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

- El dispositivo de almacenamiento, dependerá de la aplicación.
- Ej: Un dispositivo de respaldo tendrá necesidades diferentes que el almacenamiento de una base de datos.
- También se debe encontrar un balance entre los requerimientos de performance vs. los costos asociados.

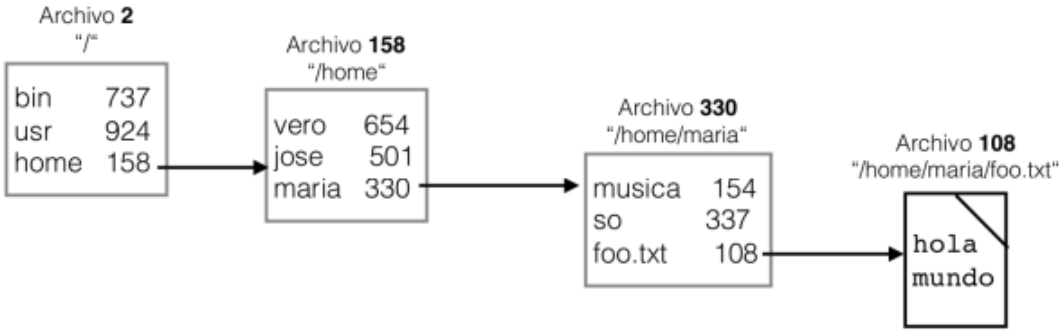
Métrica	Disco	Flash
Capacidad / Costo	Excelente	Buena
Velocidad de lectura secuencial / Costo	Buena	Buena
Velocidad de lectura aleatoria / Costo	Mala	Buena
Consumo de energía eléctrica	Regular	Buena
Tamaño físico	Bueno	Excelente
Tolerancia a vibraciones o caídas	Mala	Excelente
Almacenamiento prolongado desenergizado	Buena	Regular

- Rendimiento: Independiente del medio físico.
- Flexibilidad: Diferentes tamaños, tipos, dueños.
- Persistencia: Datos y metadatos consistentes.
- Confiabilidad: Mantener datos ante fallas.

Implementación

- Directorios: Mapean nombres a bloques.
- Índices: Árboles que relacionan nombres con bloques físicos.
- Mapas de espacio libre: Bitmap para bloques disponibles.
- Heurísticas de localidad: Minimizar seek entre datos y metadatos.

POSIX: inodos



- Bloques especiales para metadatos.
- En POSIX, los bloques que contienen metadatos de archivos son llamados inodes, estos son bloques convencionales que el sistema de archivo reserva para esta función.
- Funcionalidades:

- Lectura secuencial y aleatoria.

- Soporte para archivos grandes.

- Metadata: permisos, dueño, fecha.

Comparativa de sistemas

	FAT (16/32)[1]	NTFS[2]	EXT3/EXT4
Estructura de índice	Lista enlazada	Árbol dinámico	Árbol estático
Granularidad	Cluster	Extensión	Bloque
Asignación de espacio libre	Matriz FAT	Archivo de Mapa de Bits	Mapa de bits fijo
Localidad	Desfragmentación	Extensiones Desfragmentación	Grupo de bloques Reserva de espacio

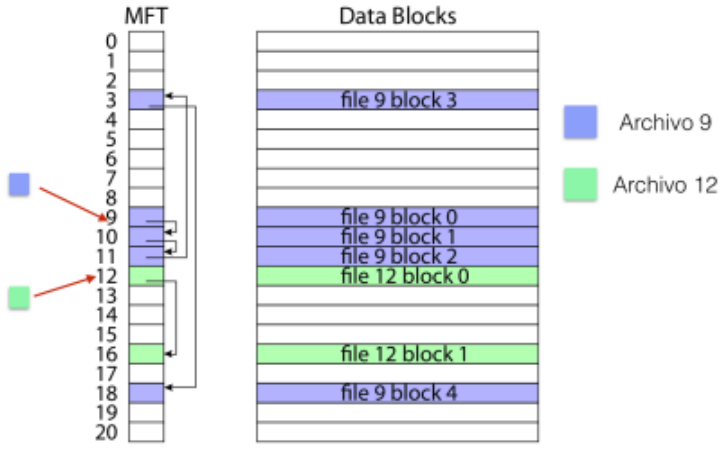
- FAT:

- Lista enlazada de clusters.

- Ventajas: Simple, compatible.

- Desventajas: Fragmentación, acceso aleatorio lento, sin journaling.

Ejemplo de una FAT con dos archivos



- Ventajas:
  - Es fácil de encontrar un clúster libre.
  - Es fácil agregar un nuevo archivo.
  - Es fácil borrar un archivo.
- Desventajas:
  - FAT y Datos en lugares diferentes. Hace lenta la lectura y escritura de archivos pequeños.
  - Acceso aleatorio del contenido de un archivo lento. Acceso secuencial.
  - Fragmentación: Los clústers disponibles pueden estar muy dispersos. Un archivo puede estar muy lejos del directorio que lo contiene, más difícil cuando queda poco espacio libre.
  - No soporta enlaces simbólicos.
  - No tiene *journaling*.

- EXT3/EXT4:

- Tabla de inodos.

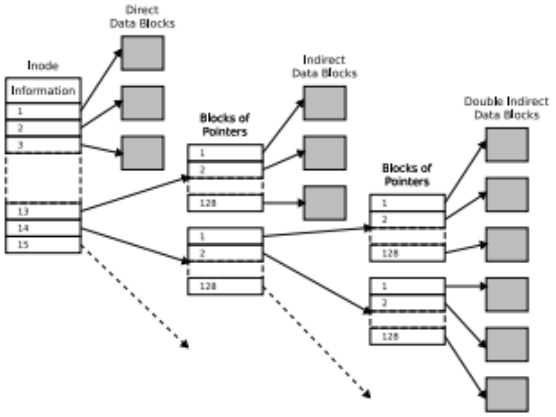
- Punteros directos (12), indirectos simples/dobles/triples.

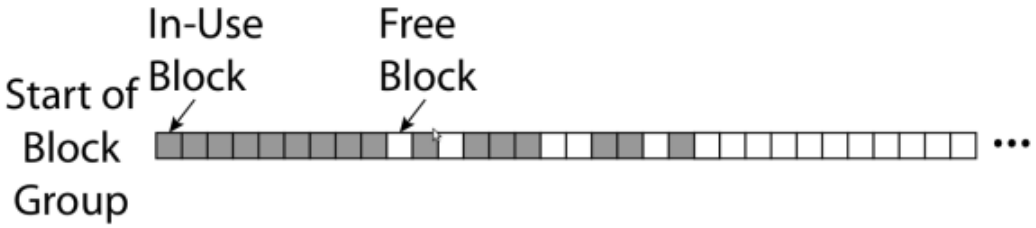
- Soporta archivos sparse.

- Bitmap para espacio libre.

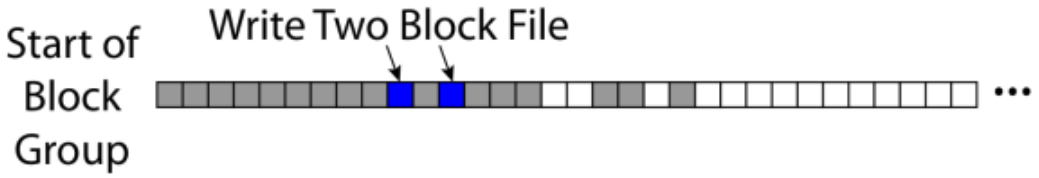
- Heurísticas: grupos de bloques para optimizar localidad.

- Cada archivo está representado en el Sistema de Archivos como un árbol, lo que permite una búsqueda eficiente de bloques.
- Cada nodo en el árbol tiene un alto grado: Minimiza las búsquedas. Con un tamaño de 4KB, un bloque indirecto apunta a 1024 bloques.
- Estructura fija: Los primeros 12 punteros siempre apuntan a los primeros 12 bloques.
- Soporta archivos poco densos (sparse), es decir, archivos que contienen espacios vacíos sin datos. Por ejemplo un archivo puede contener solo bloques directos y un bloque indirecto triple al final sin solicitar bloques para sus posiciones intermedias.

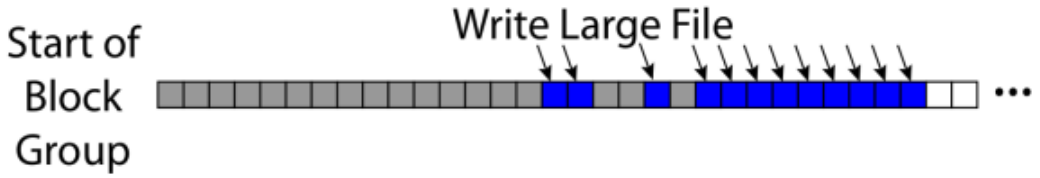




El `bitmap` del grupo, permite mantener un registro de los bloques disponibles para utilizar, de ésta manera se tiene un puntero al primer bloque libre del grupo.



En ese ejemplo se muestra como se asignan los bloques para la escritura de un archivo pequeño (2 bloques 8KB), por lo que se intenta escribir los archivos dentro del mismo grupo del directorio en el que se encuentran.



En caso de que se necesiten más bloques para almacenar el archivo, estos son tomados de manera consecutiva, intentando de minimizar el número de lecturas y *seek* necesarios para leer un archivo de manera secuencial.

- Ventajas:
  - Almacenamiento eficiente para archivos grandes y pequeños.
  - Localidad para archivos grandes y pequeños.
  - Localidad para datos y metadatos.
- Desventajas:
  - Ineficiente para archivos diminutos. Un archivo de 10 Bytes, requiere un `inodo` y un bloque.
  - En escenarios patológicos puede mostrar serios problemas de fragmentación. Se requiere de un 10% a un 20% del espacio reservado para evitar este tipo de problemas.

Disponibilidad y redundancia

- Problemas:

- Fallas eléctricas → inconsistencia.

- Fallas físicas → pérdida de datos.

- Soluciones:

- Transacciones: Propiedades ACID (Atomicity, Consistency, Isolation, Durability).

- Redundancia: Replicación en múltiples dispositivos.

Se refiere ciertas propiedades que deben tener las transacciones de un sistema de almacenamiento:

- **A**tomicity: Los cambios se hacen todos o ninguno.
- **C**onsistency: Las transacciones llevan el sistema a un estado válido.
- **I**solation: Cada transacción se ejecuta aislada.
- **D**urability: Una transacción hecha, debe sobrevivir a una falla.

- Técnicas:

- Códigos de corrección de errores:Estrategia útil para detectar y corregir errores menores en la transferencia y almacenamiento de datos. Al transferir datos para almacenarlos en el discos se agregan bits de redundancia adicional. Estos bits permiten corregir en forma transparente un error, o en casos graves detectar que se produjo un error. Normalmente los errores varían de un sector por cada 10^{14} a 10^{16} bits

- Re-mapeo de sectores dañados:Tanto en discos mecánicos como en los de estado sólido se consideran sectores de repuesto. Al detectar fallas en un sector, el controlador intenta re-asignar el sector dañado. En discos de estado sólido puede realizarse de manera transparente al incluir celdas adicionales

- MTTF y CAFR: Métricas de confiabilidad.

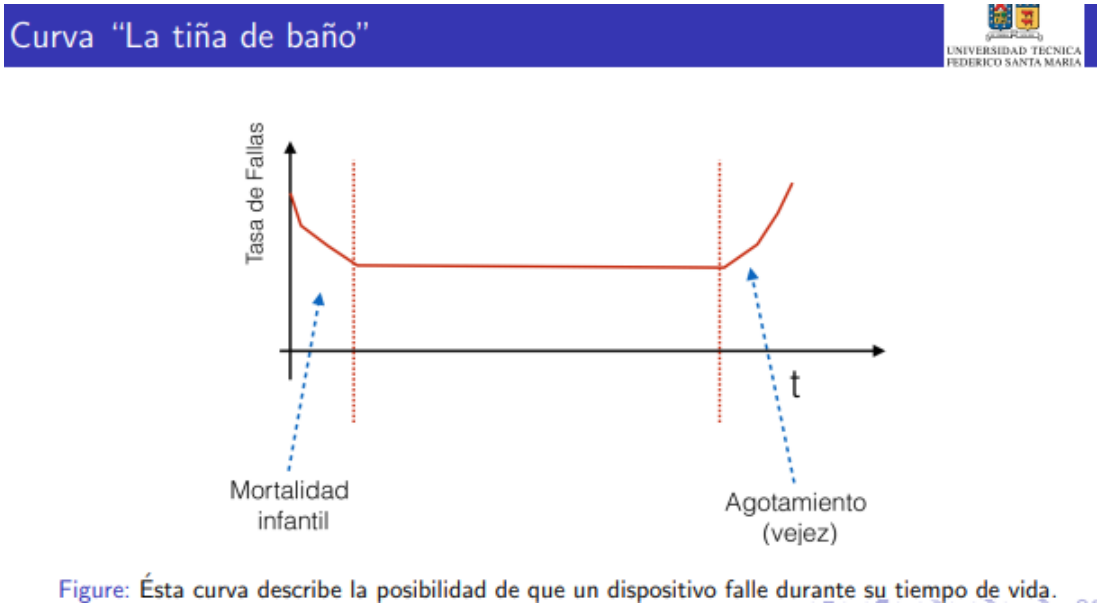


Figure: Ésta curva describe la posibilidad de que un dispositivo falle durante su tiempo de vida.

- SMART: Monitoreo preventivo.

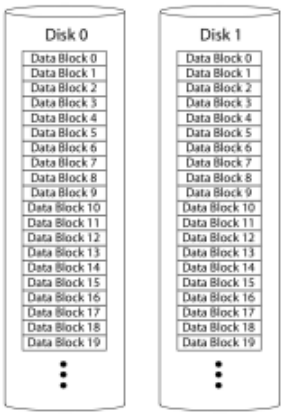
RAID

- Objetivo: Redundancia y disponibilidad.
- Niveles:

- RAID 0: Sin redundancia, solo mejora rendimiento.

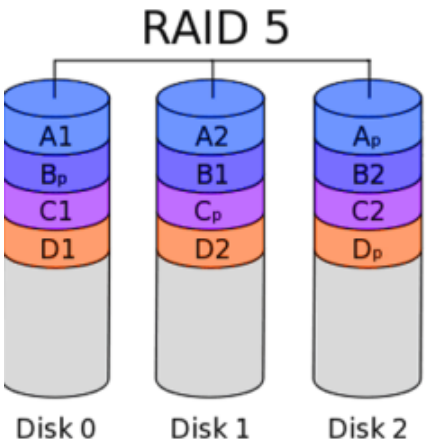
- Técnicamente no es un RAID, porque no tiene la Redundancia. Por eso se le llama también JBOD. Los bloques son repartidos entre los discos en forma secuencial.
- Permite incrementar el ancho de banda de escritura y lectura.
- Si cualquier disco del arreglo falla, se tendrá una pérdida catastrófica de datos.

- RAID 1: Espejo, redundancia total, capacidad = 50%.

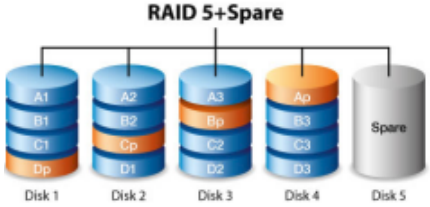


- RAID 1 corresponde a discos espejo. Se replica la escritura en dos discos. La lectura puede ser a cualquiera de ellos.
- Genera redundancia y mejora los tiempos de búsqueda (seek time), ya que pueden realizarse en paralelo en diferentes discos.
- El precio: el arreglo solo tiene la mitad del tamaño de los discos que lo componen.
- En caso de falla, el arreglo puede seguir operando con un disco menos en un estado *degradado*.

- RAID 5: Paridad rotada, requiere ≥3 discos, capacidad = (n-1).

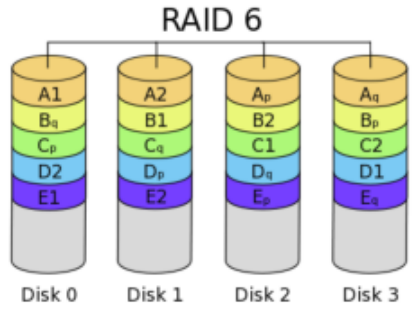


- Utiliza un esquema que mantiene la *paridad* de los datos en forma intercalada. De manera que la capacidad de un disco se utiliza como redundancia.
- La paridad de calcula realizando la operación `xor` con todos entre los bloques el mismo *stripe*.
- Requiere **almenos** 3 discos para poder realizarse.



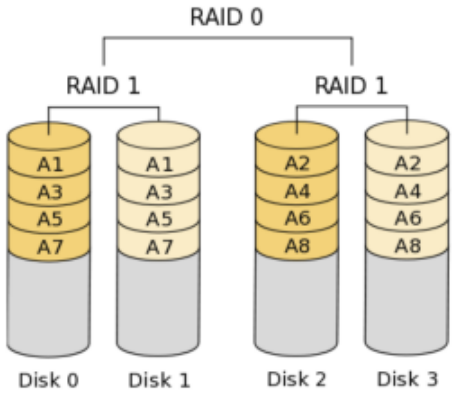
- Todas las versiones de RAID, consideran la posibilidad de tener un disco de repuesto listo y conectado en caso de falla (spare).
- El sistema quita del arreglo el disco defectuoso y regenera el arreglo utilizando el disco conectado como repuesto.
- Esto disminuye el riesgo de fallas dobles, ya que no es necesario esperar para recuperar la redundancia.

- RAID 6: Doble paridad, tolera 2 fallas, ≥4 discos.



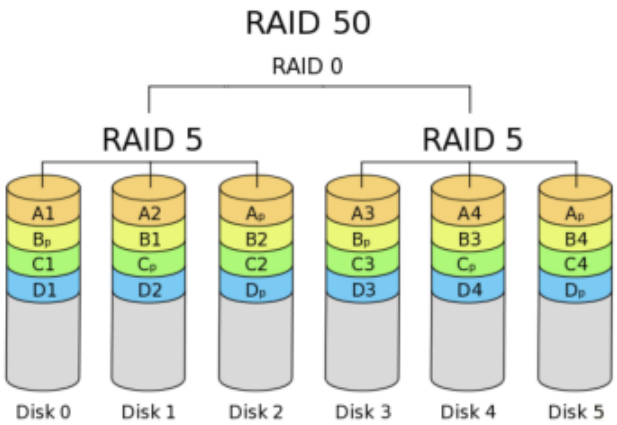
- Considera una doble redundancia dentro del propio arreglo. Tolera hasta dos fallas.
- Esto permite mejorar la confiabilidad del arreglo al tener dos copias de la paridad en todo momento.
- Se requiere al menos 4 discos para conformar éste arreglo.

- RAID 10: Combina RAID 0 + RAID 1.



- Combina los beneficios de RAID 0 y RAID 1. Es posible implementarlo como RAID 0+1 y como RAID 1+0.
- Al contener un RAID 1, la capacidad del arreglo es la mitad de los discos que lo conforman.
- Dependiendo los discos que fallen, el arreglo puede soportar múltiples fallas de discos sin perder la información.

- RAID 50: Combina RAID 5 + RAID 0.



- Una extensión del caso anterior, done se reemplaza el RAID 1 por RAID 5.
- También permite las múltiple fallas de discos, dependiendo en que grupo ocurran.

- RAID + Spare: Disco de repuesto para reconstrucción automática.

Caché y Memoria Virtual

Introducción a las memorias caché

- Objetivo: Mejorar el desempeño mediante jerarquías de memoria.
- Basado en localidad temporal (datos usados recientemente) y localidad espacial (datos cercanos).
- Ejemplos:

- TLB: Traducción rápida de páginas.

- Caché de direcciones virtuales y físicas.

- Desafíos del diseño:

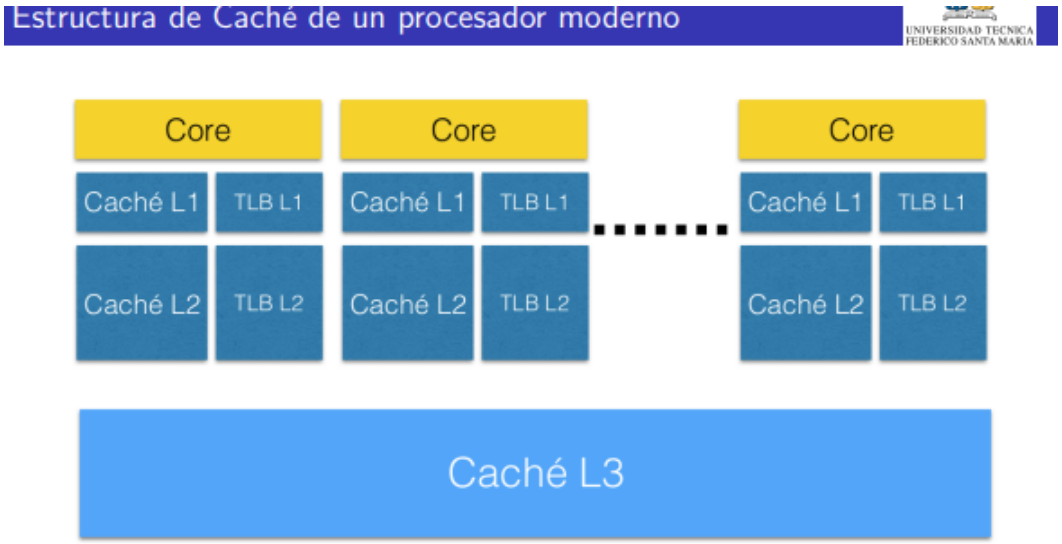
- Localizar copia cacheada.

- Política de reemplazo.

- Coherencia (mantener datos actualizados).

Jerarquía de memoria (tiempos aproximados)

- L1 / TLB: 1 ns, 64 KB.
- L2: 4 ns, 256 KB.
- L3: 12 ns, 2 MB.
- DRAM local: 100 ns, 10 GB.
- DRAM datacenter: 100 µs, 100 TB.
- Disco local: 10 ms, 1 TB.
- Disco remoto: 200 ms, 1 EB.



Operación de caché

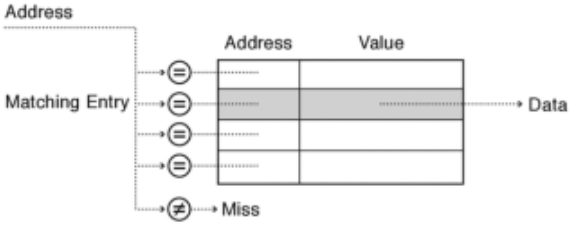
- Cache hit: Dato encontrado en caché.
- Cache miss: Buscar en nivel inferior.
- Condición: `costo(hit) < costo(miss)`.

Estrategias de búsqueda en caché

- Completamente asociativa:

- Máxima flexibilidad.

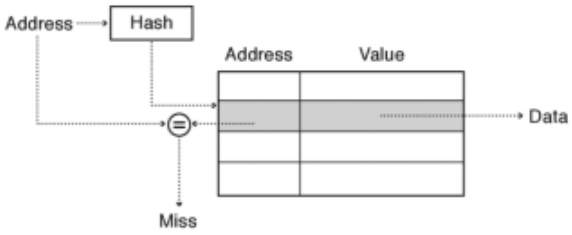
- Verifica toda la tabla (costoso).



- Directamente mapeada:

- Cada dirección en un único lugar (hash).

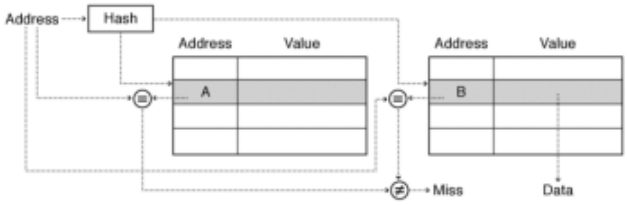
- Rápida pero menos flexible.



- Set associative:

- Mezcla de ambas.

- Búsquedas en paralelo.



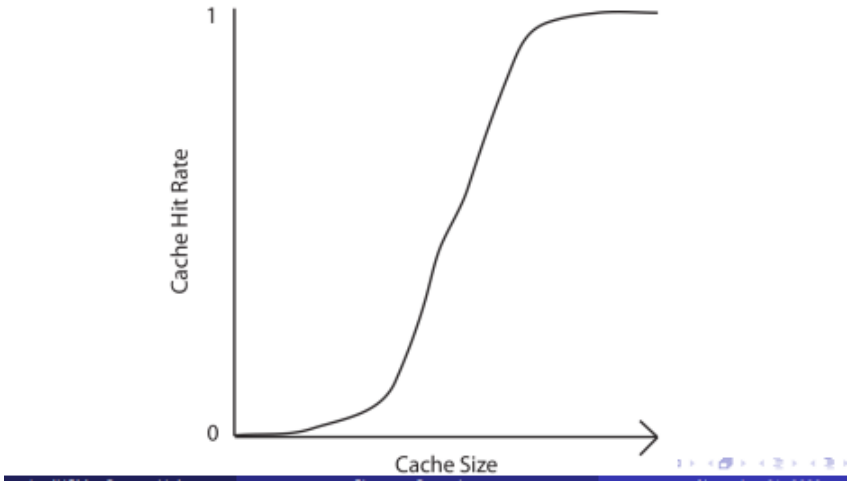
Políticas de reemplazo

- Random: Entrada aleatoria (rápido, impredecible).
- FIFO: Reemplaza la más antigua (mal rendimiento en ciclos).
- MIN (óptimo): Bloque usado más tarde (no implementable).
- LRU: Menos recientemente usado (considera localidad temporal).
- LFU: Menos frecuentemente usado (mantiene páginas populares).

Efectividad y Working Set

- Working Set: Conjunto mínimo de páginas para buena tasa de hits.
- Thrashing: Caché muy pequeña → muchos misses.

- Tasa de hits aumenta con tamaño de caché hasta punto de inflexión.



Paginación bajo demanda

- Permite usar más memoria que la física disponible.
- Fallo de página: SO carga página desde disco.
- Usado en:

- Archivos mapeados en memoria:

  - Archivo tratado como segmento de memoria.

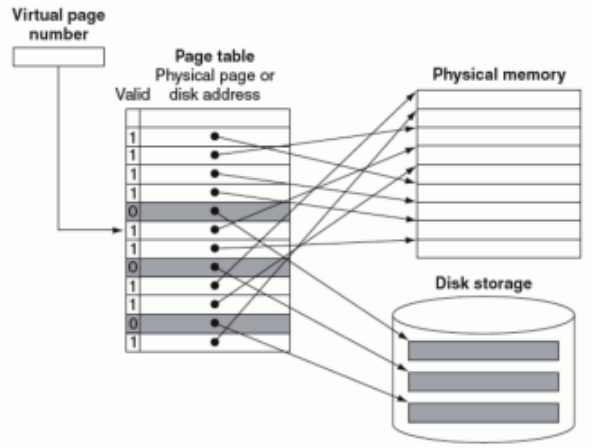
  - Ventajas:

    - Transparencia.

    - Zero copy I/O.

    - Pipelining.

    - IPC (compartir memoria entre procesos).



- Memoria virtual:

  - Frames del proceso pueden copiarse al disco.

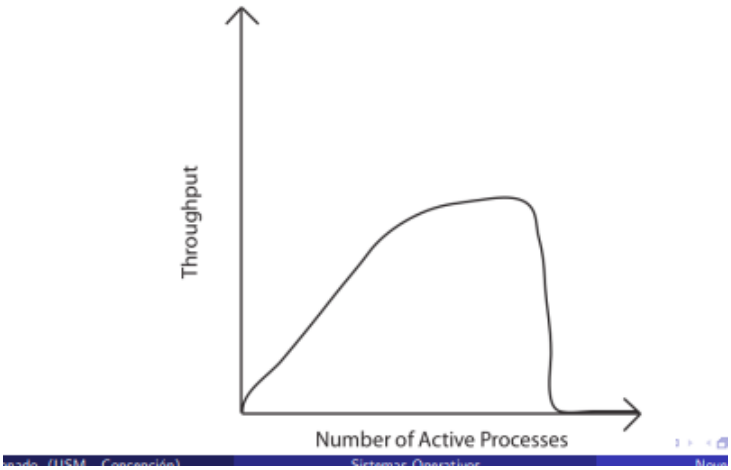
- Todas los frames del proceso (codigo, heap, stack, bibliotecas, etc) pueden ser copiados al disco, como estrategia para liberar memoria física del sistema. Aquí es donde se pone lento el pc.
- Flexibilidad: SO sigue funcionando aunque memoria física esté llena.

Problemas y soluciones

- Thrashing:

- Si suma de Working Sets ≥ memoria física.

- Disco maneja ~100 fallos/s vs CPU 10^10 instrucciones/s.



- Auto-paginación:

- Distribuye marcos equitativamente.

- Evita que procesos monopolicen memoria.

- Swapping:

- Desaloja procesos completos al disco para liberar memoria.

- Impacta rendimiento global.

Herramientas

- vmstat: Muestra estado de memoria virtual en sistemas POSIX.
