Sistemas Operativos: Teóricas

Introducción a los sistemas operativos

Que es un SO?

Una forma de dividir a los sistemas informáticos en hardware y software => requieren un intermediario

Maneja la contención y concurrencia para lograr buen rendimiento y hacerlo correctamente

Elementos Básicos

- **Drivers:** Programas parte del SO y manejan los detalles de bajo nivel relacionados con la operación de los distintos dispositivos.
- **Núcleo o kernel:** El SO propiamente dicho. Se encarga de las tareas fundamentales y contiene los subsistemas.
- **Interprete de comandos o shell:** Permite al usuario interactuar con el SO. Gráfico o command line.
- **Proceso:** Programa en ejecución + su espacio de memoria asociado
- **Archivo:** Secuencia de bits con un nombre y atributos que indican permisos
- **Directorio:** Colección de archivos y directorios que contiene un nombre y se organiza jerárquicamente
- **Dispositivo virtual:** Una abstracción de un dispositivo físico bajo la forma de un archivo
- **Sistema de archivos:** Forma de organizar datos en el disco
- **Directorios del sistema:** Directorios donde el propio SO guarda archivos que necesita para su funcionamiento
- **Binario del sistema:** Archivos que viven en los directorios del sistema. Proveen utilidades básicas
- **Archivo de configuración:**
- **Usuario:** Para aislar información entre si y establecer limitaciones
- **Grupo:** Colección de usuarios

Procesos y API del SO

Procesos

Un programa compilado en código objeto y ejecutándose es un proceso. Cada uno tiene un PID (Process ID)

Desde la memoria el proceso esta compuesto por:

- *Texto:* Código
- *Datos:* Donde se almacena el heap
- *Stack*

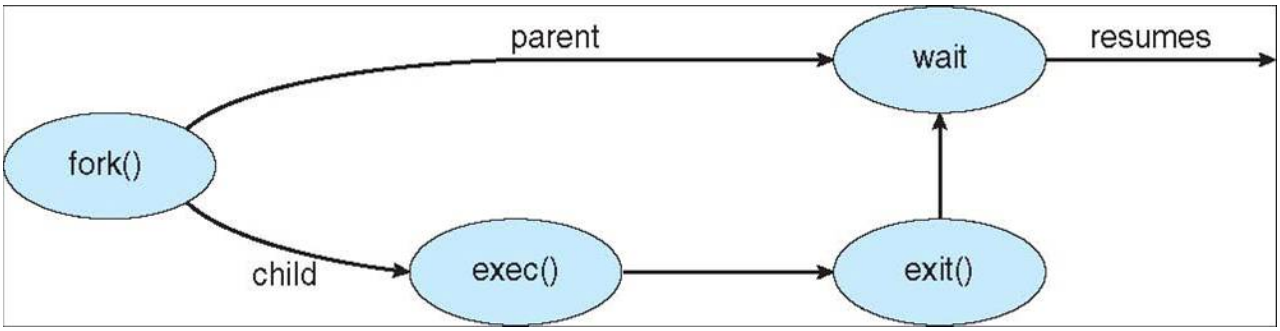
Y puede:

- *Terminar:* exit(). Indica al SO que puede liberar sus recursos y indica status de terminación.
- *Lanzar un hijo:* Los procesos se organizan como árbol. El primero se llama root o init
    - *fork():* crea un proceso igual al actual, devuelve el pid del hijo. El padre puede decidir suspenderse hasta que termine el hijo con *wait()*. Cuando termina el padre obtiene el código de status del hijo

- system() = fork() + wait()
- *exec():* El hijo hace algo distinto que el padre.

- *Ejecutar en CPU*
- *Hacer Syscall*
- *Hacer E/S*

Cuando lanzamos un programa desde el shell:



Scheduler

Decide a que proceso le corresponde ejecutar en cada momento.

Para cambiar de tarea hay que hacer un **cambio de contexto** y los datos necesarios para esto se guardan en el **PCB (Process Control Block)**

Políticas de scheduling

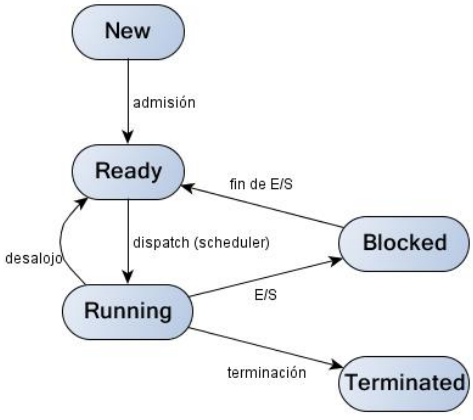
Que optimizar?

- Fairness
- Eficiencia
- Carga del sistema
- Tiempo de respuesta
- Latencia
- Tiempo de ejecución
- Rendimiento
- Liberación de recursos

Cuando actúa el Scheduler?

El Scheduler decide si el proceso debe seguir o no si...

1. Se bloquea.
2. Cuando un proceso nuevo se carga o algún proceso pasa cierto tiempo en ejecución (quantum)
3. Cuando un proceso se desbloquea.
4. Cuando un proceso termina.

• *Sin desalojo:* Cuando las decisiones solo ocurren en 1. y 4. • *Con desalojo:* con el reloj. Requiere clock con interrupciones y no da garantías de continuidad a los procesos. • Cooperativo: cuando el kernel toma control	

(syscalls).	*Con desalojo*

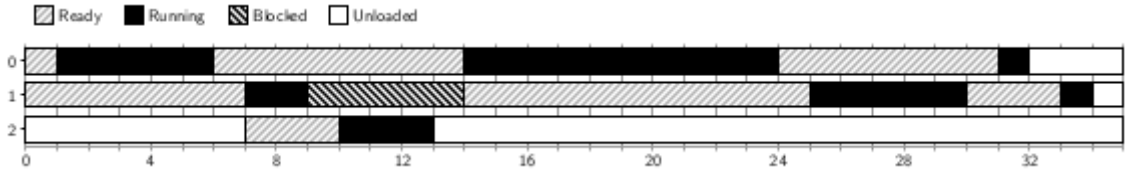
Como elijo que tarea hacer?

- *FCFS:* First Come First Serve. En orden sin interrumpir hasta que termine de ejecutarse.
- *SRTF:* Shortest Remaining Time First. (prioridades variables)
- *Sala de espera:* FIFO supone que todos los procesos son iguales => un proceso puede taponar
  => agregamos prioridades => puede generar *starvation*
  => aumento la prioridad a medida que los procesos envejecen (prioridades variables)
- *Round Robin:* Darle un **quantum** (tiempo fijo de ejecución) a cada proceso e ir alternando entre ellos
- *Múltiples colas:* La cola con mas prioridad es la que tiene menos quanta. Cuando a un proceso no le alcanza su cuota de CPU se pasa a la cola siguiente, disminuye su prioridad y y se le asigna mas tiempo de CPU en su próximo turno. Los procesos de máxima prioridad van a la cola de máxima prioridad.
- *Trabajo mas corto primero (SJF, Shortest Job First):* Intenta maximizar rendimiento. Sin desalojo
- *Scheduling para RT:* Se usan cuando hay deadlines estrictas. => EDF (Earliest Deadline First)
- *Scheduling en SMP:* Con el concepto de *afinidad al procesador* intentamos usar el mismo procesador para una tarea aunque tarde mas en liberarse para aprovechar la cache.
  Depende que tan estricto sea es *afinidad dura o afinidad blanda*

LEER DEL LIBRO

Diagrama de GANTT

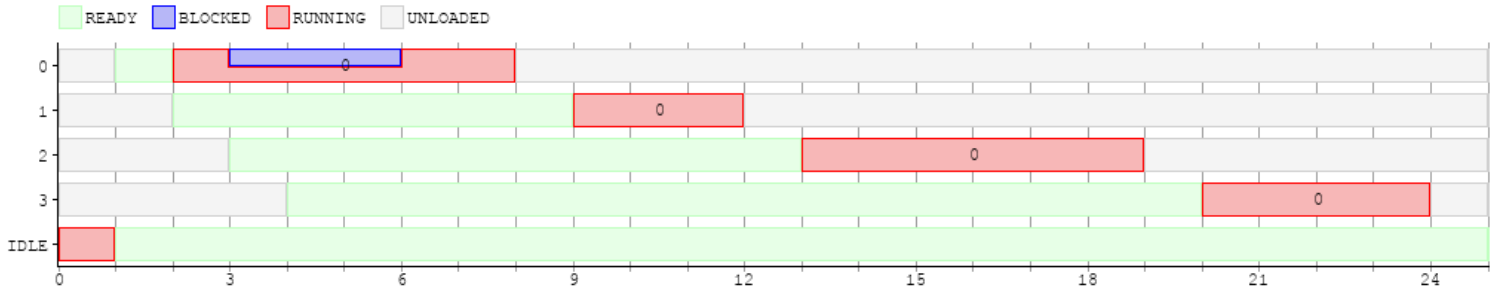
Muestra el estado de cada uno de los procesos existentes en un sistema durante un período de tiempo determinado.



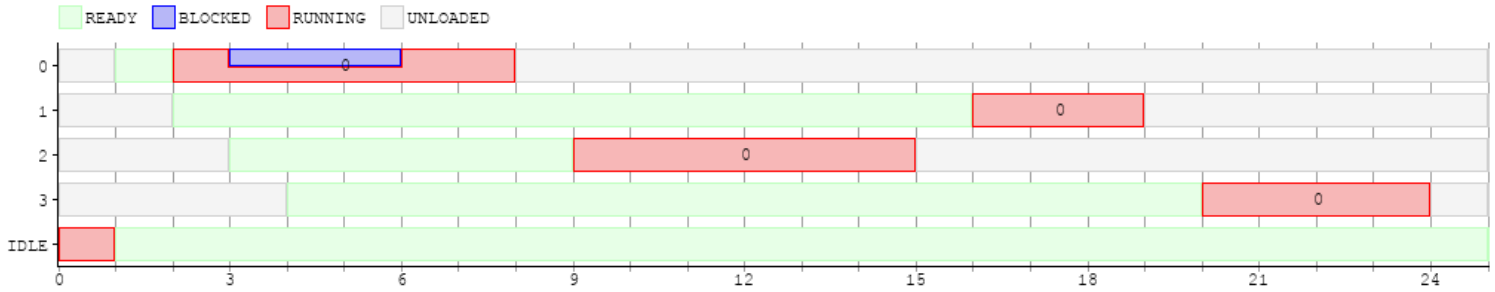
Procesos vs. Diagramas

Proceso	Tiempo de llegada	Tiempo de ejecución	Tiempo de bloqueo (incl.)	Prioridad
P0	1	6	2-4	3
P1	2	3	-	2
P2	3	6	-	1
P3	4	4	-	4

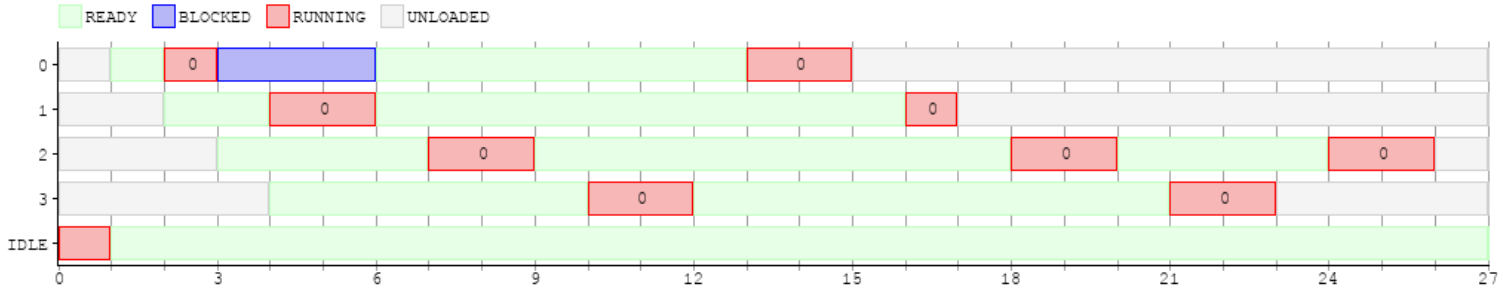
Cambio de contexto = 1
FCFS



Prioridades sin desalojo



Round Robin (q=2)



Evaluaciones de rendimiento

- *Fairness:* “Justicia” en la asignación del CPU.
- *Tiempo de* respuesta: Tiempo que el proceso tarda en empezar a ejecutarse.
- *Throughput:* Cantidad de procesos que terminan por unidad de tiempo.
- *Turnaround:* Tiempo total que le toma a un proceso ejecutar completamente.
- *Waiting time:* Tiempo que un proceso pasa en estado ready.

Sincronizacion

Tenemos dos problemas que generan la programación paralela y estructuras compartidas: contención y concurrencia.

Condición de carrera: El resultado obtenido baria dependiendo en que momento u orden se ejecuten las cosas. Toda ejecución debería dar un resultado equivalente a alguna ejecución secuencial de los mismos procesos

Veamos posibles soluciones

- Secciones criticas (CRIT)
  1. Sólo hay un proceso a la vez en CRIT.
  2. Todo proceso que esté esperando entrar a CRIT va a entrar.
  3. Ningún proceso fuera de CRIT puede bloquear a otro.

Implementaciones *Con dos llamados:* uno para salir, otro para entrar.

- *Suspender todas las interrupciones en la sección critica*
- *Locks:* Variable booleanas que cuando entro en la sección critica pongo en 1 y cuando termino en 0. Si esta en 1 espero que este en 0. Para que funcione bien me conviene usar TAS.
- *Test and set (TAS):* Se pone una variable en 1 y devuelve el valor anterior de manera atómica (indivisible). Para esperar que termine me conviene ponerla en un ***while (TestAndSet (& lock))*** para esperar e intentar obtener un lock y cuando salgo de ahí y termino el código critico ***lock = FALSE.*** Esto se llama **busy waiting** y consume mucha CPU.
- *Sleep:* Si es mucho, perdemos tiempo, si es poco desperdiciamos CPU. También desde otro lado podemos llamar a *wakeup* para terminar con el sleep cuando ya no es necesario.
- *Semáforos:* Variable entera con las siguientes características:
  - *sem(int value):* Se inicializa en cualquier valor
  - Tiene dos operaciones que se ejecutan sin interrupciones:
    - *wait(): while (s<=0) dormir(); s--;*
    - *signal(): s++; if (alguien espera por s) despertar a alguno;*
  - La versión binaria se llama *mutex()*
  - Si me olvido un signal o invierto el orden, genero deadlock
- *Atomic bools:* tienen *getAndSet(bool b)* (guarda b y devuelve lo que había en el registro) y *testAndSet()* (= a GAS pero con b = true)
- *Atomic ints:* tienen *getAndInc() / getAndDec(), getAndAdd(int) y compareAndSwap(int,int)* => no generan busy waiting (wait free)
- *Atomic queue:* tienen *enqueue(elem) y dequeue(*elem)*
- *Atomic<T> objeto:* tienen *get(), set(), compareAndSwap(T test, T value)*
- *TAS lock (Spin lock):* Tiene *create(), reg.lock()* y *reg.unlock()*. No se puede usar recursivamente (deadlock) y hace busy waiting.

```
atomic < bool > reg ;
void create () { reg . set ( false ); }
```

```
void lock () { while ( reg . TestAndSet ()) {} }
void unlock () { reg . set ( false ); }
```

- *TTAS lock (local spinning):* Testear antes de TAS. c*reate(), lock() y unlock()*

```
void create () { mtx . set ( false ); }

void lock () {
        while ( true ) {
                while ( mtx . get ()) {}
                if (! mtx . testAndSet ()) return ;
        }
}

void unlock () { mutex . set ( false ); }
```

Es mas eficiente porque:
- while hace *get()* en vez de *testAndSet()*
- cache hit mientras true
- Cuando *unlock()* hay cache miss

Tenemos el problema de garantizar exclusión mutua:

- Si la sección critica es una función => menor concurrencia
- Si la sección critica es un bloque => mayor concurrencia

Condiciones para la existencia de deadlock

- **Exclusión mutua:** Un recurso no puede estar asignado a más de un proceso.
- **Hold and wait:** Los procesos que ya tienen algún recurso pueden solicitar otro.
- **No preemption:** No hay mecanismo compulsivo para quitarle los recursos a un proceso.
- **Espera circular:** Tiene que haber un ciclo de $N \geq 2$ procesos, tal que P_i espera un recurso que tiene P_{i+1}.

Resumen de problemas de sincronizacion

- Problemas
  - Race condition
  - Deadlock
  - Starvation
- Prevención
  - Patrones de diseño
  - Reglas de programación
  - Prioridades
  - Protocolo (e.g., Priority Inheritance)
- Detección
  - Análisis de programas
    - Análisis estático
    - Análisis dinámico
  - En tiempo de ejecución
    - Preventivo (antes que ocurra)
    - Recuperación (deadlock recovery)

Razonamiento en paralelo

Correctitud de programas distribuidos

- Plantear propiedades de **safety:** cosas malas no suceden (por ejemplo, deadlock).
- Plantear propiedades de **progreso** (o **liveness**): en algún momento algo bueno sucede.
- Demostrar (o argumentar) que la combinación de esas propiedades implica el comportamiento deseado.

Tipos de Propiedades:

- *Contra-ejemplo:* Sucesión de pasos que muestra una ejecución del sistema que viola cierta propiedad.

- *Safety:* “nada malo sucede”.
  - Ejemplos: exclusión mutua, ausencia de deadlock, no pérdida de mensajes, “los relojes nunca están más de δ unidades desincronizados”.
- *Liveness:* “en algún momento algo bueno sí va a suceder”
  - Ejemplos: “si se presiona el botón de stop, el tren frena”, “cada vez que se recibe un estímulo, el sistema responde”, “siempre en el futuro el sistema avanza”, no inanición.
- *Fairness:* “Los procesos reciben su turno con infinita frecuencia”.
  - Incondicional: El proceso es ejecutado “regularmente” si está habilitado siempre.
  - Fuerte: El proceso es ejecutado “regularmente” si está habilitado con infinita frecuencia.
  - Débil: El proceso es ejecutado “regularmente” si está continuamente habilitado a partir de determinado momento.

  “No se van a dar escenarios poco realistas donde alguien es postergado para siempre”

  En general, fairness se suele suponer como válida para probar otras propiedades (liveness en general).

Propiedades

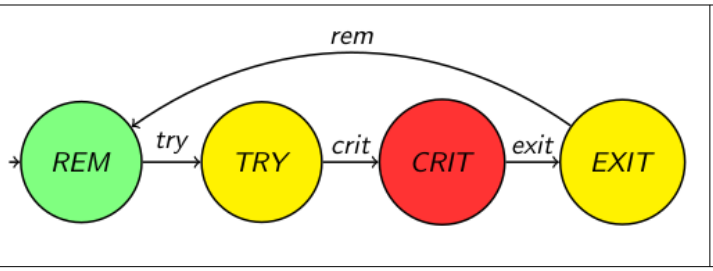
Propiedad barrera o rendezvous

Cada $P_i, i \in [0..N-1]$, tiene que ejecutar a(i); b(i).

Propiedad BARRERA a garantizar: b(j) se ejecuta después de todos los a(i)

Pero, no hay que restringir de más: no hay que imponer ningún orden entre los a(i) ni los b(i)

Modelo de proceso



- Estado:
  $\sigma : [0..N-1] \rightarrow \{REM, TRY, CRIT, EXIT\}$
- Transición:
  $\sigma \rightarrow^{l} \sigma', l \in \{rem, try, crit, exit\}$
- Ejecución:
  $\tau = \tau_0 \rightarrow^{l} \tau_1 \ldots$

Wait-Freedom

“Todo proceso que intenta acceder a la sección critica, en algún momento lo logra, cada vez que lo intenta”.

Esta propiedad se llama WAIT-FREEDOM.

$$\forall \tau, \forall k, \forall i . \tau_k(i) = TRY \Rightarrow \exists k' > k . \tau_{k'}(i) = CRIT$$

Intuición: “libre de procesos que esperan (para siempre)”. Es una garantía muy fuerte.

Fairness

Para toda ejecución τ y todo proceso i,

si i puede hacer una transición l_i en una cantidad infinita de estados de τ

entonces existe un k tal que $\tau_k \rightarrow^{l_i} \tau_{k+1}$

Exclusión mutua

Para toda ejecución τ y estado τ_k , no puede haber más de **un** proceso i tal que $\tau_k(i)=CRIT$

$EXCL \equiv cant(CRIT) \leq 1$

Progreso del sistema

Para toda ejecución τ y estado τ_k ,

si en τ_k hay un proceso i en TRY y ningún i' en CRIT

entonces $\exists j>k$, t. q. en el estado τ_j algún proceso i' está en CRIT .

$LOCK-FREEDOM \equiv (cant(TRY) \leq 1 \wedge cant(CRIT)=0 \Rightarrow cant(CRIT)>0)$

Predicados auxiliares:

- Lograr entrar: $IN(i) \equiv i \in TRY \Rightarrow i \in CRIT$
- Salir: $OUT(i) \equiv i \in CRIT \Rightarrow i \in REM$

Progreso global dependiente (deadlock-, lockout-, o starvation-freedom)

Para toda ejecución τ ,

si para todo estado τ_k y proceso i tal que $\tau_k(i)=CRIT$,

$\exists j>k$, tal que $\tau_j(i)=REM$

entonces para todo estado $\tau_{k'}$ y todo proceso i',

si $\tau_{k'}(i')=TRY$

entonces $\exists j'>k'$, tal que $\tau_{j'}(i')=CRIT$

$STARVATION-FREEDOM \equiv \forall i . OUT(i) \Rightarrow \forall i . IN(i)$

Progreso global absoluto

Para toda ejecución τ , estado τ_k y todo proceso i,

si $\tau_k(i)=TRY$

entonces $\exists j>k$, tal que $\tau_j(i)=CRIT$.

$WAIT-FREEDOM \equiv \forall i . IN(i)$

Livelock

Un conjunto de procesos está en livelock si estos continuamente cambian su estado en respuesta a los cambios de estado de los otros.

Sección critica de M <= N

Propiedad a garantizar

$\forall \tau, \forall k$

1. $cant(i|\tau_k(i)=CRIT) \leq M$
2. $\forall i . \tau_k(i)=TRY \wedge cant(j|\tau_k(j)=CRIT)<M \Rightarrow \exists k'>k . \tau_{k'}(i)=CRIT$

SWMR (Single-Writer/Multiple-Readers)

Una variable compartida que los escritores necesitan acceso exclusivo pero los lectores pueden leer simultáneamente

Registros RW

Características:

- Si read() y write() NO se solapan => read() devuelve el último valor escrito.
- Si read() y write() se solapan
  - "Safe": read() devuelve cualquier valor
  - Regular: read() devuelve algún valor escrito.
  - Atomic : read() devuelve un valor consistente con una serialización.

Ejemplos de algoritmos EXCL con registros RW

	Dijkstra	*Panadería de lamport*
Registros	• flag[i]: atomic single-writer / multi-reader • turn: atomic multi-writer / multi-reader	• choosing[i], number[i]: atomic single-writer / multi-reader

Proceso i	/* TRY */
	L : flag [i] = 1;
	while (turn 6 = i)
	  if (flag [turn] == 0) turn = i ;
	flag [i] = 2;
	foreach j 6 = i
	  if (flag [j] == 2) goto L ;
	/* CRIT */
	...
	/* EXIT */
	flag [i] = 0;
		/* TRY */
		choosing [i] = 1;
		number [i] = 1 + max j6 = i number [j];
		choosing [i] = 0;
		foreach j 6 = i {
		  waitfor choosing [j]==0;
		  waitfor number [j]==0 ‖
		  (number [i] , i) < (number [j] , j);
		}
		/* CRIT */
		...
		/* EXIT */
		number [i] = 0;

Caract.	• Garantiza EXCL. • Suponiendo FAIRNESS, garantiza LOCK-FREEDOM, pero no WAIT-FREEDOM.	• Garantiza EXCL, LOCK-FREEDOM y WAIT-FREEDOM.
Complej.	Los algoritmos vistos requieren O(n) registros RW.	

Teorema: No se puede garantizar EXCL y LOCK-FREEDOM con menos de n registros RW

	Fischer
Registros	• turn: multi-writer / multi-reader
Proceso i	/* TRY */ L : waitfor turn = 0; turn = i ; *tarda a lo sumo δ* pause Δ; if turn 6 = i goto L ; /* CRIT */ ... /* EXIT */ turn = 0;
Caract.	• Garantiza EXCL.

Problema del consenso

Dados:

- *Valores:* V = {0, 1}
- *Inicio:* Todo proceso *i* empieza c/ *init(i)* $\in V$.
- Decisión: Todo proceso i decide un valor decide(i) $\in$ V .

El problema de consenso requiere:

- *Acuerdo:* Para todo i ≠ j, decide(i) = decide(j).
- *Validez:* Existe i, tal que init(i) = decide(i).
- *Terminación:* Todo i decide en un número finito de transiciones (WAIT-FREEDOM).

Teorema: No se puede garantizar consenso para un n arbitrario con registros RW atómicos.

	• Suponiendo FAIRNESS, garantiza LOCK-FREEDOM si Δ > δ.	

Consensus number

Es la cantidad de procesos para los que resuelve consenso

- Registros RW atómicos = 1
- Colas, pilas = 2
- (TAS) getAndSet() = 2
- compareAndSwap() tiene consensus number infinito.

Administración de memoria

El manejador de memoria se encarga de:

1. Asegurar la disponibilidad de memoria.

   Memoria virtual: Hacerle creer al proceso que dispone de más memoria de la que realmente tiene.

   - Cantidad de bytes de memoria física = MEM SIZE.
   - Cantidad de unidades de direccionamiento de memoria física = MEM SIZE / DIR UNIT.
   - Cantidad de unidades de direccionamiento de memoria virtual = 2^DIR_BITS .
   - Cantidad de bytes de memoria virtual = 2^DIR_BITS * DIR UNIT.

2. Asignar y liberar memoria.
   - Asignar = reservar una porción de memoria para un proceso.
     - La porción de memoria pasa a estar ocupada por el proceso que la solicitó.
     - Tenemos que saber quién es el dueño de esa porción de memoria.
   - Liberar = un porción de la memoria vuelve a estar disponible para cualquier proceso.

3. Organizar la memoria disponible.

   El MM debe elegir que porción de memoria asignar

   - ¿Cómo agrupamos la memoria?
     - *En unidades de direccionamiento.*
     - *En Bloques de tamaño fijo.*
     - *En Bloques de tamaño variable.*
   - ¿Cómo organizamos la memoria libre?
     - *Con un mapa de bits.*
     - *Con una lista enlazada.*
   - Mecanismos más sofisticados:
     - *Segmentación:* Separar la memoria en segmentos.
       - Cada segmento tiene una base y un límite
       - Se acceden mediante **direcciones lógicas.**

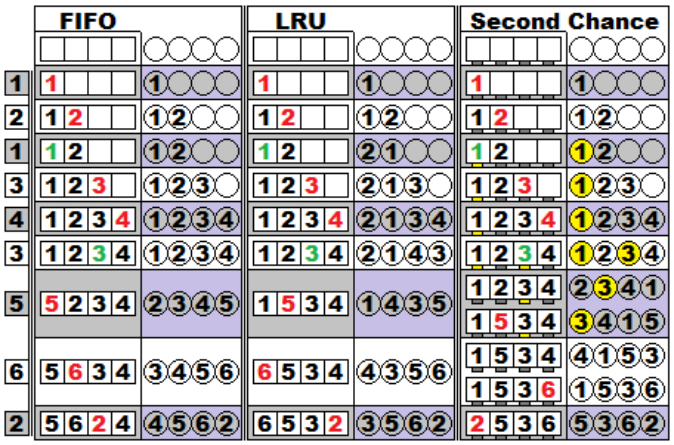
- *Paginación:* Dividir la memoria en paginas.
  - Nos permite mapear mucha más memoria de la que realmente tiene el sistema, Si una página no está cargada en ningún marco de página, el MM se encarga de ir a buscarla al disco y cargarla en memoria.
  - Algoritmos de remoción:
    - FIFO, LRU
    - Segunda oportunidad: si fue referenciada le doy otra chance
    - Not Recently Used: Primero desalojo las ni referenciadas ni modificadas. Después las referenciadas y por último las modificadas.

Tengo un sistema con 6 páginas y sólo 4 marcos de página. La memoria comienza vacía.

Llegan los siguientes pedidos de memoria (número de página) en ese orden:

1, 2, 1, 3, 4, 3, 5, 6, 2

Hit-Rate= Páginas qué pedí y ya estaban cargadas en memoria/páginas totales pedidas.



- *Segmentación + paginación.*

<u>Direcciones lógicas vs. Físicas</u>

- *<u>Memoria física:</u>* Una celda dentro de la memoria del sistema. El tamaño está determinado por el hardware.
  - Un **marco de página** es una porción de memoria física.
- *<u>Memoria virtual:</u>* Una representación de la información almacenada. El tamaño depende de la unidad de direccionamiento y la cantidad de bits de direccionamiento.
  - Una **página** es una porción de memoria virtual.

4. Asegurar la protección de la memoria.

Un proceso no debería poder usar memoria que no reservó. Soluciones:

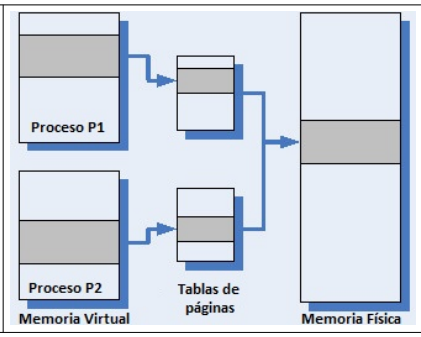
- *Paginación:* Cada proceso tiene su propia tabla de páginas.
- *Segmentación:* Cada proceso tiene su propia tabla de segmentos.

5. Permitir acceso a memoria compartida.

Queremos que dos procesos lean y escriban sobre una misma variable.

- *Paginación:* Podemos mapear dos páginas al mismo marco de página.

***malloc* no es syscall** sino un “memory manager” provisto por la lib-C. Ergo, *malloc* vive y se ejecuta en el espacio de usuario, al igual que *calloc, realloc y free*



En linux malloc pide memoria al kernel con:

- *brk() y sbrk():* cambian el límite del heap del proceso
- *mmap():* mapea una página en el espacio de direcciones del proceso.

Como pide mas memoria?

1. busca entre las porciones libres de memoria del Heap uno o más buckets contiguos que puedan satisfacer el tamaño pedido.
2. Si eso falla, se intentará extender el Heap del proceso a través brk()/sbrk().
3. La llamadas brk()/sbrk() al Kernel ajustarán, dentro del mm struct del proceso, el tope del Heap para que éste sea más extenso. Esto no aumenta la memoria directamente, ya que no se está mapeando memoria física al proceso, sino que lo que se hace es ampliar el espacio de direcciones reservado para el Heap.
4. Finalmente, cuando alguna de esas posiciones de memorias no mapeadas es accedida (generalmente por una lectura/escritura de la implementación de malloc) se produce una excepción de Page Fault. Dicha excepción será atrapada por el Kernel y producirá una invocación al administrador de páginas para obtener una nueva página de memoria física para el frame que generó la excepción.

6. Controlar el swapping.

Si solo hay un proceso, no hace falta compartir. Si hay mas de uno hay que hacer **swapping,** es decir, pasar a disco el espacio de memoria de los procesos que no se están ejecutando. Esto nos puede causar problemas como:

- *Reubicacion:* Puede ser que cuando vuelva a la memoria, el espacio que les corresponda no sea el mismo, lo que se puede solucionar teniendo un registro que haga de base y que las direcciones del programa sean relativas.
- *Protección:* Como nos aseguramos que un proceso no lea los datos de otro (memoria privada)?
- *Manejo de espacio libre:* Como sabemos que pedazos de memoria tenemos libres y donde conviene ubicar un programa? (Evitar fragmentacion)

Fragmentacion

Ocurre cuando tenemos suficiente memoria para atender una solicitud pero no es continua.

Puede ser:

- *Externa:* Bloques pequeños de memoria no contiguos.
- *Interna:* Memoria desperdiciada dentro de una partición (un bloque o página).

Veamos posibles soluciones:

- *Compactar:* requiere reubicacion, es muy costoso en tiempo
- *Organizar la memoria:*
  - *Bitmap:* Dividir en bloques de igual tam la memoria. Cada posición del bitmap representa un bloque (0 = libre, 1 = ocupado). Encontrar bloques consecutivos requiere barrida lineal.
  - *Lista enlazada:* Cada nodo representa un proceso o bloque libre, y dicen el tam y sus limites. Liberar es simple. Asignar puede ser (ninguna se usa en la vida real):
    - First fit: La primera sección de memoria contigua del tamaño necesario.
      Rápido, tiende a fragmentar la memoria
    - Best fit: De todas las secciones de tamaño mayor o igual al tamaño necesario, tomo la más chica.
      Lento, no es mejor (llena la memoria de bloquecitos inservibles)
    - Worst fit: Toma la mas grande
    - Quick fit: Mantengo una lista de bloques libres de los tamaños mas frecuentemente solicitados.
    - Buddy system: usa splitting de bloques
  - *Memoria virtual:* Requiere un MMU. Espacio de direcciones: tamaños de la mem física + swap.
    Los programas usan direcciones virtuales. La memoria virtual se divide en **paginas** de tam fijo y la física en **page frames.**
    La MMU traduce paginas a frames y se swappean las paginas. Si la pagina no esta en memoria, la MMU hace page fault y la atrapa el SO.
    La MMU es una tabla de paginas, y para hacerla mas eficiente conviene hacerla multinivel: Los primeros bits nos llevan hacia la tabla que tenemos que consultar. Hace que no haga falta tener toda la tabla en memoria y se pueden swappear sus partes.
  - *Memoria asociativa:* Para mejorar la performance, podemos agregar un cache a la memoria virtual, lo que nos permite mapear directamente páginas a frames sin consultar las tablas de páginas (que pueden tener varios niveles).

Page Fault

¿Qué pasa cuando una página no está cargada en la memoria?

1. Un proceso accede a una dirección (virtual) de memoria.
2. La MMU traduce la dirección virtual a dirección física (accede a la entrada en la última tabla de páginas).
3. Lee el atributo correspondiente a presencia en la memoria.
4. Si es negativo, se produce la interrupción Page Fault.
5. Se ejecuta la RAI correspondiente.
6. Si la memoria está llena, se ejecuta el algoritmo de remoción.
7. Si la página que se va a desalojar fue modificada, hay que bajarla al disco.
8. Se carga en el lugar liberado la página solicitada.
9. Se vuelve a ejecutar la instrucción del proceso que accede a la dirección solicitada.

Thrashing

Situación en la que el SO pasa más tiempo cargando páginas que ejecutando procesos.

Supongan que tenemos 2 procesos y un solo marco de página disponible.

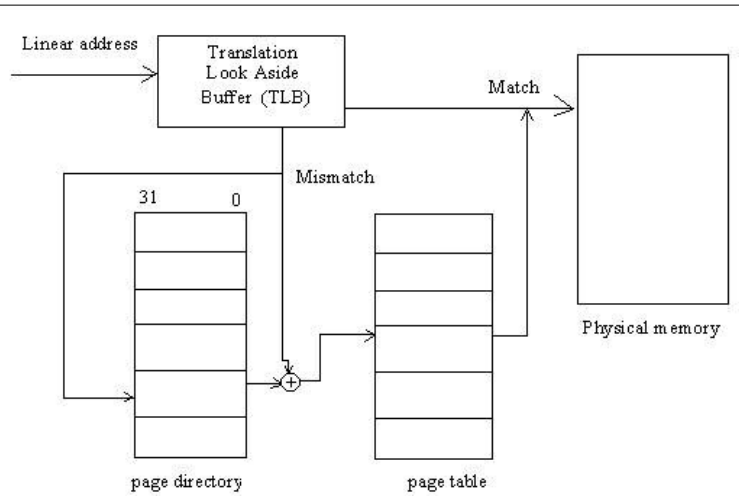
Cada proceso usa una sola página, pero cada vez que ejecute, su página no va a estar cargada.

Siempre va a estar cargada la página del otro.

TLB (Translation Lookaside Buffer)

Buffer de Traducción Adelantada.

- Es una caché que guarda ’traducciones’.
- Paginación de 4 niveles: Cuatro accesos a memoria (1 por cada tabla) más uno para leer la página.



Copy-on-write

Al crear un nuevo proceso se duplica toda su memoria.

Sabemos que, en general después de un fork() viene un exec().

exec() inutiliza todas las páginas de memoria, entonces ¿para qué nos gastamos en duplicar todo?

Copy-on-Write: Sólo duplico (copy) cuando alguno de los procesos escribe (write).

2do parcial

Administración de E/S

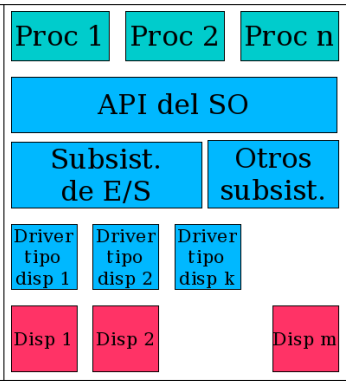
Esquema de E/S

Cada dispositivo de E/S tiene dos partes: el dispositivo físico y un controlador que interactua con el SO

Drivers

Los drivers son componentes de software muy específicos que conocen las particularidades del HW contra el que hablan

- Corren con máximo privilegio: pueden hacer colgarse a todo el sistema.
- De ellos depende el rendimiento de E/S, que es fundamental para el rendimiento combinado del sistema.



Interacción con los dispositivos

Polling	*Interrupciones o push*	*DMA*
Ventajas: sencillo, cambios de contexto controlados. *Desventajas:* Consume CPU.	*Ventajas:* eventos asincrónicos poco frecuentes. *Desventajas:* cambios de contexto impredecibles.	Requiere de un componente de HW, el controlador de DMA. Cuando el controlador de DMA finaliza, interrumpe a la CPU.

Subsistema de E/S

El manejador de E/S y los drivers se ocupan de proveerle al programador una API sencilla: open(), close(), read(), write(), seek(). Pero sin ocultar cosas que necesitan saber las aplicaciones.

Los dispositivos pueden separarse en:

- *Char device:* Dispositivos en los cuales se transmite la información byte a byte.
  Ejemplos: mouse, teclado, terminales o puerto serie.
  Debido a su acceso secuencial (byte a byte) no soportan acceso aleatorio y no utilizan cache.
- *Block device:* Dispositivos en los cuales se transmite la información en bloque.
  Ejemplos: disco rígido, flash memory o CD-ROM.
  Permite el acceso aleatorio y por lo general utilizan un buffer (cache).

El dialogo con los dispositivos tiene las siguientes características:

- Son de lectura, escritura o lecto-escritura.
- Brindan acceso secuencial o aleatorio (sería mejor decir arbitrario).
- Son compartidos o dedicados.
- Permiten una comunicación de a caracteres o de a bloques.
- La comunicación con ellos es sincrónica o asincrónica.

- Tienen distinta velocidad de respuesta.

API del subsistema de E/S

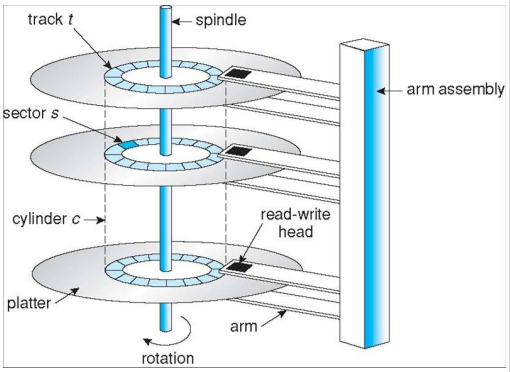
- Todo es un archivo
- Se proveen funciones de *alto nivel* para acceso a archivos: *fopen, fclose, fread, fwrite,* etc.

Planificación de E/S

Para tener buen rendimiento hay que manejar apropiadamente el disco.
La planificación de disco se trata de cómo manejar la cola de pedidos de E/S para lograr el mejor rendimiento posible.

Hay que manejar:

- *Ancho de Banda:* Cantidad de bytes que se pueden transmitir a la vez
- *Latencia Rotacional:* Tiempo necesario para que el disco rote y la cabeza quede en el sector deseado
- *Tiempo de Búsqueda/Seek Time:* tiempo necesario para que la cabeza se ubique sobre el cilindro que tiene el sector buscado.



Políticas de Scheduling de E/S a disco

- *FIFO o FCFS*
- *SSTF* (Shortest Seek Time First): Goloso, no optimo. Mejora tiempo de rta. Puede producir inanicion.
- *Algoritmo Scan o del ascensor:* Ir primero en un sentido, atendiendo los pedidos que encuentro en el camino, luego en el otro. El tiempo de espera no es tan uniforme.

SSD – Solid State Drive

- Son más livianos, resistentes y silenciosos. Consumen menos energía.
- Tienen mejor performance en la lectura que los HDD, al no tener componentes mecánicos. La escritura es más compleja.
- Problema de durabilidad y write amplification

Gestión del disco

Formateo:

- Poner en cada sector unos códigos que luego sirven a la controladora de disco para efectuar detección y corrección de errores.
- Funcionan como un prefijo y un postfijo a la parte donde efectivamente van los datos en cada sector.
- Si al leer un sector, el prefijo y postfijo no tienen el valor que deberían, el sector está dañado.

Booteo:

- Las computadoras suelen tener un minúsculo programa en ROM que carga a memoria ciertos sectores del comienzo del disco, y los comienza a ejecutar.
- El programa cargado es muy pequeño. No llega a ser el SO, sino más bien un cargador del SO.

Bloques dañados

- A veces se manejan por software, se encarga de estos el sistema de archivo.
- Los discos SCSI vienen con sectores extra para reemplazar los defectuosos.
- Cuando la controladora detecta un bloque dañado actualiza una tabla interna de remapeo y utiliza otro sector.
- Para no interferir con las optimizaciones del scheduler de E/S, los discos a veces traen sectores extra en todos los cilindros

Spooling (Simultaneous Peripheral Operation On-Line)

Forma de manejar a los dispositivos que requieren acceso dedicado en sistemas multiprogramados.

La idea es poner el trabajo en una cola, y designar un proceso que los desencole a medida que el dispositivo se libere. Esto permite que el dispositivo no se bloquee hasta que termine de realizar un proceso.

El kernel no se entera de que se está haciendo spooling, el usuario sí.

Locking

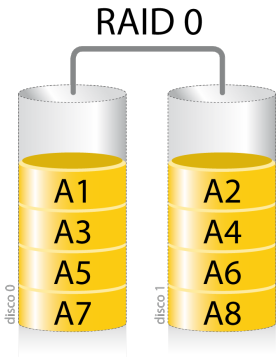
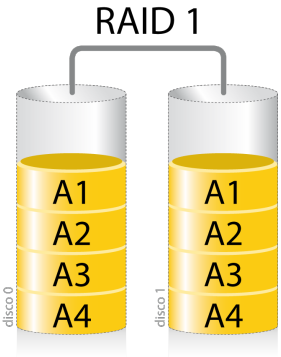
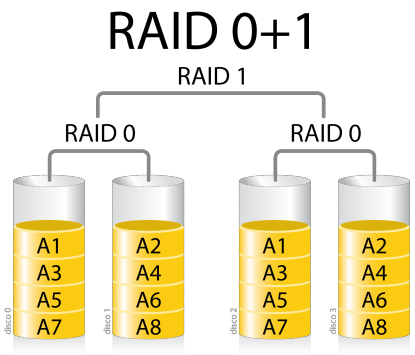
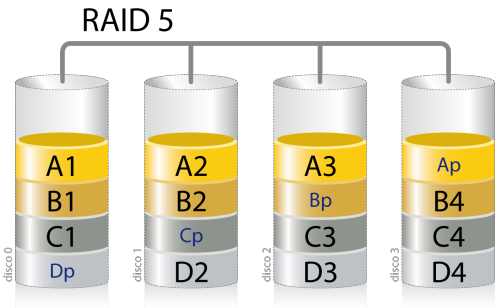
POSIX garantiza que open(..., O CREAT | O EXCL) es atómico y crea el archivo si no existe o falla si ya existe.

Eso es un mecanismo sencillo de exclusión mutua para implementar locks.

Proteccion de informacion

A partir del valor que tenga esta informacion debo tomar una politica de resgualdo de costo acorde. Algunos ejemplos de estas:

- **Copias de seguridad (backup):** Puede ser muy costoso.
  - *Copia total.* Para restaurar tomo la del dia correspondiente y listo.
  - *Copia incremental:* Solo los archivos modificados desde la ultima copia incremental. Para restaurar necesito la última copia total y todas las incrementales entre ésa copia total y la fecha requerida.
  - *Copia diferencial:* Sólo los archivos modificados desde la última copia total. Para restaurar necesito la última copia total más la última diferencial.
- **Redundancia:** Cuando el costo de que el sistema salga de linea es muy alto se usa un RAID: Redundant Array of Inexpensive Disks.
  - Es usar dos discos: cada escritura se hace en los dos. Si uno se rompe, tengo el otro: **mirror** y si bien es
  - Conveniente, pero muy costosa.
  - Ventaja: Puedo hacer dos lecturas a la vez, una en cada disco.
  - RAID no protege contra borrar (o modificar) un archivo accidentalmente. Se combina con copias de seguridad.
  - Si la aplicación corrompe los datos, ningún mecanismo sirve.
  - Si se corrompe la estructura interna de los archivos, RAID tampoco ayuda.
  - Para eso hay sistemas de archivos que brindan algo de protección
  - Tiene distintos posibles niveles:

Nombre	*Caracteristicas*	*Imagen*
RAID 0 (stripping)	No aporta redundancia, pero mejora el rendimiento: los bloques de un mismo archivo se distribuyen en dos (o más) discos. Mejora el ancho de banda, permite escrituras en paralelo (si los discos están en diferentes controladoras).	
RAID 1 (mirroring)	Espejado de los discos. Mejor rendimiento de lecturas. Las escrituras, en mejor caso, tardan lo mismo, en peor, el doble. *Es muy caro.*	
RAID 0+1	Combina los dos anteriores: espejado y stripping. Cada archivo está espejado, pero al leerlo leo un bloque de cada disco. Lo leo más rápido que en mirroring simple. Como si fuera stripping. Pero al escribir, tengo que escribir cada de bloque en ambos	
RAID 2 y 3	• Tener, por cada bloque, información adicional para determinar si se dañó o no. • Algunos errores se pueden corregir automáticamente a partir de la información redundante. • Cada bloque lógico se distribuye entre todos los discos participantes. • RAID 2 requiere 3 discos de paridad por cada 4 de datos mientras que RAID 3 requiere sólo 1. • Todos los discos participan de todas las E/S, lo cual lo hace más lento que RAID 1. • Puede requerir mucho procesamiento para computar las redundancias. • Por eso se suele implementar por HW en una controladora dedicada, al igual que todos los niveles siguientes.	
RAID 4	• Como RAID 3, pero stripping a nivel de bloque (ie, cada bloque en un solo disco). • El disco dedicado a paridad sigue siendo un cuello de botella para el rendimiento, porque todas las escrituras lo necesitan.	
RAID 5	También usa datos redundantes, pero los distribuye en N+1 discos. Es decir, no hay un disco que sólo contenga redundancia. Cada bloque de cada archivo va a un disco distinto. Para cada bloque, uno de los discos tiene los datos y otro tiene la información de paridad. Si bien ya no hay cuello de botellas para las escrituras hay que mantener la paridad distribuida, lo que no es sencillo. Puede soportar la pérdida de un disco cualquiera. Cuando se reemplaza y comienza la reconstrucción, el	

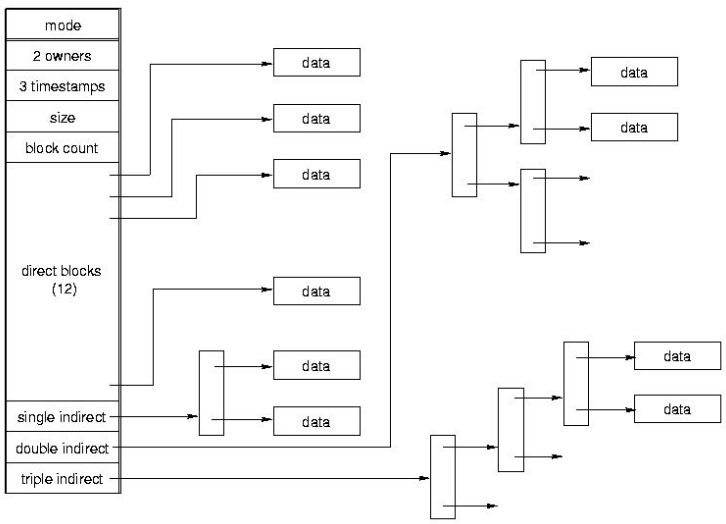
	rendimiento se degrada notablemente.	
RAID 6	• Es como RAID 5, pero agrega un segundo bloque de paridad, • también distribuido entre todos los discos. • Las implementaciones varían, pero el objetivo principal es soportar la rotura de hasta dos discos. • Considerando que RAID 5 se suele usar con un hot spare, no hay diferencia sustancial en el espacio “desperdiciado” (a grandes rasgos).	

Sistemas de archivos o File Sistem (FS)

- **Que es?:** módulo dentro del kernel encargado de organizar la información en disco.
- **Responsabilidades:**
  - Ver cómo se organizan, de manera lógica, los archivos:
    - Interna: como se estructura la información dentro del archivo.
    - Externa: cómo se ordenan los archivos. Directorios: organización sea jerárquica, con forma de árbol.
  - Soportan *links:* un alias, otro nombre para el mismo archivo. Teniendo links la estructura deja de ser arbórea y se vuelve un grafo dirigido propiamente dicho, con ciclos y todo.
  - Determina como se nombrara a los archivos
    - Punto de montaje: indicaba al SO que la cinta ya estaba colocada, y qué punto del grafo de nombres de archivos íbamos a considerar como la raíz de la cinta.
  - Cómo se representa un archivo? ¿Cómo gestiono el espacio libre? ¿Qué hago con los metadatos?
- **Representacion de archivos:** para el FS un archivo es una lista de bloques + metadata.
  - *Contiguos:* no es bueno porque los archivos pueden crecer y no tener lugar / fragementacion
  - *Lista enlazada:* Lecturas consecutivas rapidas, lecturas aleatorias lentas. Desperdicio de espacio.
  - *FAT:* Tabla que por cada bloque me dice en que bloque esta el siguiente elemento de la lista.
    + Permite leer en desorden y no desperdicio espacio del bloque
    - Necesito toda la tabla en memoria, inmanejable para discos grandes, unica tabla → mucha contencion, poco robusto, no maneja seguridad.
  - *Nodos:*

Estructura:
- Atributos
- Direcciones de bloques directas
- Indireccion single
- Indireccion doble → apunta a tabla de singles
- Indireccion triple → apunta a bloque de double

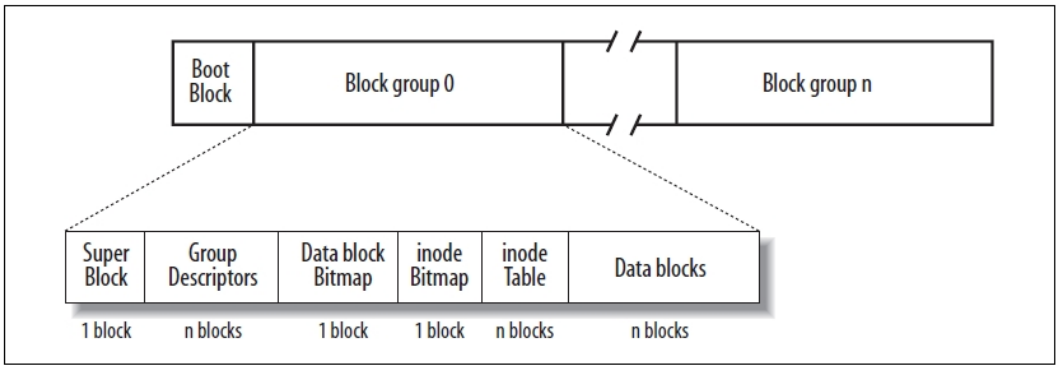
+ Tengo en memoria solo las tablas correspondientes a los archivos abiertos
+ Una tabla por archivo → menos contencion
+ Consistencia: sólo están en memoria las listas correspondientes a los archivos abiertos.



- **Implementacion de directorios:**
  - Un inodo se reserva como entrada al root directory
  - Por cada archivo o directorio dentro del directorio hay una entrada.
  - Dentro del bloque se guarda una lista de (inodos, nombre de archivo/directorio).
  - En algunos casos, cuando los directorios son grandes, conviene pensarlos como una tabla de hash más que como una lista lineal de nombres, para facilitar las búsquedas.
  - A veces esto se hace a mano, en la capa de software de aplicación.

- **Atributos:** Parte del metadata, junto con los inodos. Incluye permisos, tamanio, propietarios, etc
- **Manejo del espacio libre:** Dos metodos:
  - *Mapa de bits empaquetado:* Bits en 1 significan libre. Si una palabra tiene todos 0, la puedo saltear con una unica comparacion.
    - Requiere tener el vector en memoria.
  - *Lista enlazada de bloques libres:* En general clusterizada. Un bloque de disco puede contener n punteros a otros bloques, los primeros n − 1 indican bloques libres y el último es el puntero al siguiente nodo de la lista.
- **Cache:** Mejora el rendimiento.
  + puede grabar las páginas de manera ordenada, de manera tal que el administrador de E/S pueda planificar más eficientemente la escritura.
  - A veces, las aplicaciones pueden configurarse para hacer escritura sincrónica, es decir, escribiendo en disco inmediatamente. (mas lento)
  - *Consistencia:* Un bit del FS indica apagado normal para saber cuando se copio a disco todos los datos de la cache. Si ese bit no esta prendido, el sistema corre *fsck*, hace *soft updates*, o *journaling*
  - *Journaling:* Registro de los cambios que habria que hacer en un buffer circular. Cuando se baja el caché a disco, se actualiza una marca indicando qué cambios ya se reflejaron. Si el buffer se llena, se baja el caché a disco.
    + Poco impacto en performance.
    + Cuando el sistema levanta, se aplican los cambios aun no aplicados, que es mucho mas rapido que recorrer todo el disco.
- **Cuotas de disco, encripción, snapshots, manejo de RAID por software, compresión**
- **Performance:** La impactan muchos factores
- **Network File System (NFS):** Protocolo que permite acceder a FS remotos como si fueran locales usando RPC
  Usa una capa llamada Virtual FS que tiene vnodes por cada archivo abierto que se correponden con inodos si es local o algo mas si es remoto.
- **Estructura Ext2:** El superbloque (superblock) contiene metadatos críticos del sistema de archivos tales como información acerca del tamaño, cantidad de espacio libre y donde se encuentra los datos. Si el superbloque es

dañado, y su información se pierde, no podría determinar que partes del sistema de archivos contiene información.



Protección y seguridad

Protección: Mecanismos para asegurarse de que nadie pueda meter los garfios en los datos del otro.

Seguridad: Se trata de asegurarse que quien dice ser cierto usuario, lo sea. También se trata de impedir la destrucción o adulteración de los datos.

Seguridad de la información

Mantiene...	***Elementos / Protagonistas***
• Confidencialidad • Integridad • Disponibilidad	• Sujetos. • Objetos. • Acciones. Qué sujetos pueden realizar qué acciones sobre qué objetos.
Acciones	***Algoritmos de encriptación***
• *Autenticación* • *Autorización:* que podes hacer? • *Auditoria / Accounting:* Deja registrado que hiciste	• *Simétricos:* utilizan la misma clave para encriptar y para desencriptar. • *Asimétricos:* usan claves distintas. • *Funciones de hash one-way*

Funciones de hash one-way

Se suele pedir que cumplan con:

- *Resistencia a la preimagen.* Dado h debería ser difícil encontrar un m tal que h = hash(m).
- *Resistencia a la segunda preimagen.* Dado m_1 debería ser difícil encontrar un $m_2 \neq m_1$ tal que $hash(m_1) = hash(m_2)$
- Etc.

Método RSA

p, q primos de muchos digitos, $n = pq$, $n' = (p-1)(q-1)$

Elijo un entero $e \in [2, (n'-1)]$ coprimo con n' => e y n son clave publica

Calculo $d \rightarrow d.e \mod n' = 1$ => d y n son clave privada

Para encriptar m: $m^e \mod n$ / Para desencriptar c: $c^d \mod n$

Representación de permisos

La manera mas sencilla es una matriz de Sujetos x Objetos de acciones permitidas.

- **Discretionary Access Control (DAC):** Los atributos de seguridad se tienen que definir explícitamente por el dueño
- **Mandatory Access Control (MAC):** Para data altamente sensible. Cada sujeto tiene un grado, los obketos creados heredan el grado del ultimo sujeto que lo modifico. Un sujeto solo puede acceder a objetos de grado menor o igual que el.

Sistemas Distribuidos

Conjunto de recursos conectados que interactúan.

- Varias máquinas conectadas en red.
- Un procesador con varias memorias.
- Varios procesadores que comparten una (o más) memoria(s).

Fortalezas:

- Paralelismo.
- Replicación.
- Descentralización.

Debilidades:

- Dificultad para la sincronización.
- Dificultad para mantener coherencia.
- No suelen compartir clock.
- Información parcial .

Sistemas distribuidos de memoria compartida

Hardware

- Uniform Memory Access (UMA)
- Non-Uniform Memory Access (NUMA)
- Hibrida

Software

- Estructurada
  - Memoria asociativa: Linda (tuple-spaces), JavaSpaces.
  - Distributed arrays: Fortran, X10, Chapel.
- No estructurada
  - Memoria virtual global.
  - Memoria virtual particionada por localidad.

Sistemas distribuidos sin memoria compartida

Si **no hay memoria compartida** => Arquitectura del software

Tienen en comun que la cooperación tiene la forma de solicitarle servicios a otros. (cliente/servidor)

El programa “principal” hace de cliente de los distintos servicios que va necesitando para completar la tarea.

Ejemplos:

- *Telnet:* Protocolo y programa para conectarse remotamente a otro equipo.
- *RPC:* Permite a programas hacer *procedure calls* de manera remota. Es un mecanismo sincrónico

Mecanismos asincronicos: De comunicacion asincronica.. Ej: RPC asincronico, pasaje de mensajes

Pasaje de mensajes

Es un mecanismo asincrónico mas general porque no supone que haya nada compartido mas que el canal de comunicacion.

Problemas a considerar:

- Tengo que manejar la codificación/decodificación de los datos.
- Si hago una comunicación asincrónica, tengo que dejar de procesar para atender el traspaso de mensajes.
- La comunicación es lenta.
- Eventualmente el canal puede perder mensajes (hoy en día con TCP/IP se puede pensar que el canal es confiable).
- Eventualmente podría haber un costo económico por cada mensaje que se transite por el canal.

Problemas a ignorar:

- Los nodos pueden morir (también en los otros esquemas, pero acá es más probable/de mayor impacto).
- La red se puede partir (partes incomunicadas).

Locks en entornos distribuidos

No hay TAS atomico. Para lockear podemos....

- Poner el control de los recursos bajo un único nodo, que hace de coordinador.
- Dentro de él hay procesos que ofician de representantes (o proxies) de los procesos remotos.
- Cuando un proceso necesita un recurso se lo pide a su proxy, que lo “negocia” con el resto de los proxies.

=> *problemas:*

- Unico punto de falla
- Cuello de botella
- El coordinador puede estar lejos
- Las interacciones con el coordinador requiere mensajes (lento)

=> *alternativas:*

- Decidir el orden de los mensajes a partir del reloj? Los relojes pueden no estar sincronizados.
- No hace falta saber el momento de emision de mensajes, solo el orden
  => **orden parcial no reflexivo** entre los eventos:
  - Si dentro de un proceso, A sucede antes que B, $A \rightarrow B$.
  - Si E es el envío de un mensaje y R su recepción, $E \rightarrow R$. Aunque E y R sucedan en procesos distintos.
  - Si $A \rightarrow B$ y $B \rightarrow C$, entonces $A \rightarrow C$.
  - Si no vale ni $A \rightarrow B$, ni $B \rightarrow A$, entonces A y B son concurrentes.

  Implementacion orden parcial

- Cada procesador tiene un reloj
- Cada mensaje tiene la lectura del reloj
- La recepcion es siempre posterior a envio. Si recibo un mensaje con una marca de tiempo *t* mayor al valor actual del reloj, actualizo mi reloj a *t+1*

Si queremos *orden total* hay que romper empates arbitrariamente, por ejemplo, por pid.

Scheduling en sistemas distribuidos

- Dos niveles
  - *Local*: dar el procesador a un proceso listo
  - *Global*: asignar un proceso a un procesador (**mapping**)
- Global: **compartir** la carga entre los procesadores
  - *Estática*: en el momento de la creación del proceso (**affinity**)
  - *Dinámica*: la asignación varía durante la ejecución (**migration**)
- Compartir vs **balancear**
  - *Balancear*: repartir equitativamente
  - *Evaluar* costo-beneficio
- Migración
  - Iniciada por el procesador sobrecargado (**sender initiated**)
  - Iniciada por el procesador libre (r**eceiver initiated / work stealing**)
- Política de scheduling
  - *Transferencia*: **cuándo** hay que migrar un proceso.
  - *Selección*: **qué** proceso hay que migrar.
  - *Ubicación*: a **dónde** hay que enviar el proceso.
  - *Información*: **cómo** se difunde el estado.

Problemas de sincronizacion

Modelo de fallas

Los posibles problemas (combinables) que pueden pasar son:

- Nadie falla (ie, los resultados son correctos si no hay fallas).
- Los procesos caen pero no levantan.
- Los procesos caen y pueden levantar.
- Los procesos caen y pueden levantar pero sólo en determinados momentos.
- La red se particiona (ojo con los algoritmos de acuerdo).
- Los procesos pueden comportarse de manera impredecible (fallas bizantinas).

Pero veremos versiones sin fallas

Metricas de complejidad

- Cantidad de mensajes
- Tipos de fallas que soportan
- Cuanta info necesitan (tam red, # procesos, como ubicarlos)

Problemas

- *Exclusión mutua distribuida:*

  Usar **token passing** => armar un anillo lógico entre los procesos y poner a circular un token. Cuando quiero entrar a la sección crítica espero a que me llegue el token.

  + No hay inanición si no hay fallas
  - Hay mensajes circulando incluso cuando no son necesarios

  Otra opcion es **solicitud** => Cuando quiero entrar a la sección crítica envío a todos (incluyéndome) $solicitud(P_i, ts)$, siendo *ts* el timestamp.

  - Cada proceso puede responder inmediatamente o encolar la respuesta.
  - Puedo entrar cuando recibí todas las respuestas.
  - Si entro, al salir, respondo a todos los pedidos demorados.
  - Respondo inmediatamente si:
    - No me interesa entrar en la sección crítica.
    - Si quiero entrar, aún no lo hice y el *ts* del pedido que recibo es menor que el mío, porque el otro tiene prioridad.

  - Este algoritmo exige que todos conozcan la existencia de todos.
  + No circulan mensajes si no se quiere entrar a la sección crítica.

- *Locks distribuidos:*

  Protocolo de la mayoria =>

  - Queremos lock sobre un objeto del cual hay copia en n lugares.
  - Debemos pedirlo a por lo menos n/2 + 1 sitios.
  - Cada sitio responde si puede o no dárnoslo.
  - Cada copia del objeto tiene un número de versión. Si lo escribimos, tomamos el más alto y lo incrementamos en uno.

  - Puede producir deadlock y no pueden haber dos locks a la vez.
  + No puedo leer copias desactualizadas

- *Elección de lider:*

  Para elegir un lider, mantengo un status que dice que no soy el lider, a traves de un anillo de procesos hago circular mi ID. Cuando recibo un mensaje, comparo el ID con el mio y circulo el mas alto.
  Cuando da toda la vuelta es porque encontre al lider y circula un mensaje de notificacion para que todos lo sepan

  Tiempo:

  - Sin fase de stop O(n).
  - Con fase de stop $O(2 \cdot n)$.

  Comunicacion

  - $O(n^2)$
  - Cota inferior $\Omega(n \log n)$.

- *Instantánea global consistente:*
  - Cuando se quiere una instantánea, un proceso se envía a sí mismo un mensaje de **marca.**
  - Cuando P_i recibe un mensaje de marca por primera vez guarda una copia C_i de E_i y envía un mensaje de marca a todos los otros procesos.
  - En ese momento, P_i empieza a registrar todos los mensajes que recibe de cada vecino P_k hasta que recibe marca de todos ellos.
  - En ese momento queda conformada la secuencia $Recibidos_{i,k}$ de todos los mensajes que recibió P_i de P_k antes de que éste tomara la instantánea.
  - El estado global es que cada proceso está en el estado C_i y los mensajes que están en Recibidos están circulando por la red.

  Usos:
  - Detección de propiedades estables (una vez que son verdaderas, lo siguen siendo).
  - Detección de terminación.
  - Debugging distribuido.
  - Detección de deadlocks.
- *2PC:*

  Two Phase Commit => transacción de manera atómica. Todos de acuerdo en que se hizo o no se hizo.

  En una primera fase le preguntamos a todos si están de acuerdo en que se haga la transacción.
  - Recibo *no*: aborto
  - Recibo *si*: tomo nota de los que dicen si

  Si no recibo todos los *si* despues de un tiempo, aborto. Si recibo todos los si, en la segunda fase aviso todos que se confirmo.
- *Acuerdos:*
  - *k-agreement (o k-set agreement):* $decide(i) \in W, tal\ que |W| = k$
  - *Aproximado:* $\forall i \neq j. |decide(i) - decide(j)| \leq \epsilon$
  - *Probabilístico:* $Pr(\exists i \neq j.\ decide(i) \neq decide(j)) \leq \epsilon$