Apunte teórico de Sistemas Operativos

Resumen por Navier 🧃

Basado en el apunte de Juan Cruz González Moreno, en los capítulos de libro OSTEP vistos en clase, usando el proyecto de traducción a castellano de Lucía Martínez Gavier y colaboradorxs (OSTEP-castellano), junto a los podcasts y notas tomadas en clases del profesor Nicolas Wolovick en FaMAF - UNC.

Secciones vistas en la cursada del año 2024.
Capítulos incluidos: 2, 4-8 (virtualización de cpu), 13-15, 17-20 (virtualización de memoria)
26-28, 30-32 (concurrencia), 36, 37, 39-42 (persistencia)

02: Introducción al SO

Una computadora, según el pipeline del modelo Von Newmann, es una máquina de estados que ejecuta cíclicamente **fetch-decode-execute-writeback**. Es decir, el procesador constantemente lee una instrucción de la memoria, la decodifica (interpreta) y la ejecuta.

El SO (Sistema Operativo) es el software que se encarga de que sea fácil correr programas de forma eficiente y correcta. Para ello utiliza la **virtualización**; una técnica para representar al hardware de una forma simple, segura y eficiente, gracias a la cual los programas pueden correr en simultáneo compartiendo CPU (procesador), memoria (RAM) y dispositivos (discos).

La interacción entre el SO y los usuarios se da mediante una interfaz **API** (una **standard library**) que provee **system calls** a los programas.
Como muchos programas corren concurrentemente, el SO también se encarga de ser el **administrador** de los **recursos**, entregando el uso del hardware real a los programas de forma eficiente y siguiendo diferentes **mecanismos** y **políticas**.

Virtualización del CPU
El SO crea la ilusión de que hay muchos CPUs disponibles para que muchos programas puedan correr en simultáneo. La ejecución de estos es manejada por diferentes **mecanismos** y **políticas** establecidas por el SO.

Virtualización de la Memoria
La memoria física no es más que un array de bytes, y para poder leer o escribir en ella se necesita especificar una **dirección** en donde hacerlo. Al ser volátil, su contenido se pierde al interrumpir el suministro de electricidad.

Los programas acceden a memoria todo el tiempo, ya sea por información (archivos) o porque su propio código se encuentra allí (en cada instruction fetch se accede a memoria).
Al estar virtualizada, cada programa tiene su propio espacio direccionable de memoria virtual (**address space**). Esas direcciones no se corresponden con las reales de la memoria física (y por tanto, pueden repetirse entre programas), y es el SO quien se encarga de luego mapear (traducir) esas direcciones de **memoria virtual**.
Los accesos a memoria de un programa no afectan (o no deberían, por protección) a otros programas o al SO mismo (eso es, aislamiento de procesos).

Concurrencia
Los problemas de concurrencia surgen con la ejecución de programas en simultáneo (compartiendo memoria y CPU) o por programas que usan **múltiples hilos**. Requiere de la ejecución **atómica** de instrucciones (toda la ejecución de una sola vez, sin interferencias) para evitar comportamientos indefinidos y problemas de seguridad. Con el fin de evitar esas **race condiciones** (o posibles **deadlocks**), se utilizan **locks**, **semáforos**.

Persistencia
La información en RAM es **volátil**, por lo que el SO debe encargarse de la persistencia de la información. Lo hace a través de hardware (con dispositivos de **I/O**, como discos) y el software (**sistema de archivos**).

Los dispositivos de almacenamiento no son virtualizados para cada programa, sino que estos pueden compartir información entre ellos. Poseen una velocidad mucho menor a la de la memoria RAM, pero son más baratos y con una capacidad mayor.

Metas de diseño

La idea es desarrollar **abstracciones** que permitan **maximizar el uso del hardware**, asegurando la robustez y calidad del servicio (evitar sobrecargas, tolerar fallas, etc.):

1) Facilidad de uso del sistema.
2) **Minimizar gastos** (overheads) → Alto rendimiento equilibrando costo/desempeño.
3) **Protección**: entre aplicaciones y al SO de ellas → Aislamiento de procesos + modo de usuario (limitado) y modo kernel (privilegiado).
4) **Confiabilidad**: si el SO falla, todas las aplicaciones lo hacen.
5) **Eficiencia energética**.
6) **Seguridad**: contra aplicaciones malignas.
7) **Movilidad**: portabilidad a múltiples plataformas.

volatile int counter = 0; → volatile = variable no optimizada por compilador, que puede ser usada por más de un proceso

loop = atoi(argv[1]); → atoi = función que convierte string en int ("9" → 9) y devuelve 0 en caso de error

04: La abstracción de los procesos

Un **proceso** es la abstracción del SO de un programa en ejecución; el programa es un objeto estático (código), mientras que el proceso es un objeto dinámico (corriendo en memoria).

La técnica de ir intercalando varios procesos concurrentes, ejecutando uno a la vez, es conocida como **Time Sharing**. Su contrapartida, **Space Sharing**, consiste en dividir un recurso (en el espacio) entre procesos que deseen utilizarlo.

Virtualización

Para lograr la ilusión de que hay muchos CPU (y, a su vez, que los procesos no tengan el control directo de lo que se ejecuta en la CPU física), el SO usa **mecanismos** de bajo nivel: protocolos para implementar distintas funcionalidades. Algunas de ellas son el **context switch** (cambios de contexto), que permite cambiar un proceso en ejecución por otro, y las diferentes **políticas** de los **planificadores**: algoritmos que toman decisiones sobre la distribución de los recursos limitados en base a distintos factores y prioridades.

Proceso

Un proceso puede ser descrito (y por tanto guardado) por su **estado**:

1) **Memoria**: Las instrucciones y la información que el proceso lee están en memoria. La memoria a la que un proceso puede acceder (**address space**) es parte del proceso mismo.

2) **Registros** de la CPU: El programa puede leer o actualizar registros (como el **program counter** (PC), que indica qué instrucción se ejecutará a continuación, el **stack pointer**, o el **frame pointer**).
3) **Almacenamiento**: Un proceso puede acceder al almacenamiento (dispositivos I/O) para asegurar la persistencia.

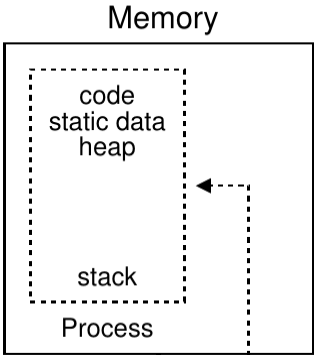
API de los procesos
Cualquier interfaz de procesos de un SO debe poder:

1) **Crear** nuevos procesos.
2) **Destruir** procesos en caso de que no terminen por sí mismos.
3) **Esperar** la finalización de un proceso.
4) Tener **control**: otros tipos de control, como la suspensión de un proceso.
5) Conocer su **estado**: poder mostrar información de un proceso.

Los programas utilizan estas funciones mediante **system calls** proporcionadas por la API.

Creación de procesos
Para correr un programa (convertirlo en proceso) el SO debe **cargar** su código y su información estática (con un formato ejecutable) del disco a la memoria, en el address space del proceso.

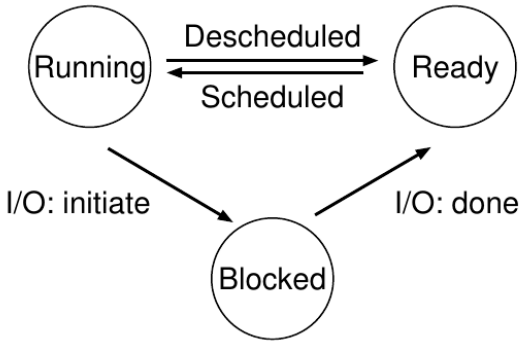
Se deben proporcionar memoria **Stack** (variables locales, parámetros de llamada, dirección de retorno) y **Heap** (información dinámica y variable en tamaño, estructuras de datos como listas; todo lo relacionado con malloc y free) para el programa. En el stack, además, el SO establece los parámetros 'argv' y 'argc count'.
Luego, se deben iniciar los **file descriptors** (**std in** / **std out** / **std error**) (0, 1, 2).
Por último, el SO setea todos los registros a 0 (menos el PC) y pasa el control al proceso creado, dejándolo ejecutarse.



Memoria asignada a un proceso al momento de su creación.

Estados de los procesos
Cada proceso está en alguno (solo uno) de los siguientes estados:

- **Running:** en ejecución
- **Ready**: en espera, listo para ejecutarse
- **Blocked**: no puede correr hasta que otro evento ocurra (por ej. un dispositivo I/O)
- **Zombie**: el proceso hijo finaliza, pero el padre aún no ha llamado a wait() para leerlo



Transiciones entre estados de un proceso.

El paso entre los distintos estados está dado por eventos del software o hardware, llamadas **interrupciones** (por ej., de reloj, del disco duro, etc.)

A lo sumo, puede haber n procesos en estado de running, siendo n = cantidad de cores (núcleos).

Estructuras de datos del SO
Lista de procesos (**process list**): conviene el listado de los procesos listos para correr o corriendo.
PCB (process control block): estructura de datos que, en cada entrada de sus tuplas, almacena el **contexto de los registros**, el mapa de la memoria, el estado del proceso, su process id (**pid**), los archivos abiertos, el directorio actual (**pwd**: process working directory), y el puntero al proceso padre. Además, almacena los datos de bajo nivel manejados por el procesador: el trap frame y el kernel stack.

Esta información sobre los procesos del sistema nos permite realizar un **context switch**: "congelarlos", guardar o colocar la información almacenada en los registros, y continuar la ejecución de un proceso. El SO almacena un array de PCBs.

- **User Time**: tiempo de cómputo del proceso en modo usuario.
- **CPU time**: tiempo total que un proceso ha usado la CPU. Si ejecuta más de un hilo, es la suma del tiempo de todos ellos.
- **Wall Time**: tiempo total real transcurrido, desde que comienza a ejecutarse un proceso hasta que termina.
- **S**: modo del proceso: sleeping, active, etc

WallTime ≥ *CPU Time*: cuando el proceso se ejecuta en un solo núcleo o en múltiples núcleos pero sin paralelismo (un núcleo alternando entre varios hilos).
WallTime < *CPU Time*: puede llegar a suceder si el proceso se ejecuta en múltiples núcleos simultáneamente (paralelismo), ya que el CPU Time se suma en cada núcleo que está en uso.

UserTime < *WallTime*: el proceso pasa parte del tiempo esperando: ya sea I/O, bloqueado, o al return del trap cuando hace una syscall y pasa a modo kernel.
UserTime = *WallTime*: un proceso sin hilos se ejecuta en modo usuario, sin interrupciones ni operaciones de I/O, sin llamadas al sistema, y no se bloquea ni o sufre context switches.
WallTime < *UserTime*: proceso multihilo, en el que cada hilo ejecuta en un core distinto y luego se suman todos sus tiempos de usuario.

05: La API de los procesos

La interfaz que permite interaccionar con la abstracción de los procesos;
La creación de procesos en Unix se da con dos system calls (llamadas al sistema): fork() y exec(). Puede usarse wait() para esperar que un proceso creado termine.
Cada proceso tiene un identificador único, un número llamado **PID**.

- argc = entero que representa el número de argumentos pasados a la línea de comandos. Incluye el nombre del programa como el primer argumento, por lo que argc siempre es al menos 1.
- argv[ ] = array de punteros a cadenas de caracteres que contiene el nombre del programa a ejecutar (en argv[0]) y los argumentos pasados al programa en el resto de posiciones.
Siempre está terminado por un puntero NULL (argv[argc] = NULL)

Fork() system call
Crea un nuevo proceso. El proceso creado es una **copia** casi idéntica del proceso donde fue llamada fork() (**parent**); el proceso creado (**child**) tiene una copia del **address space** (mapa de memoria) pero es su propia memoria privada, sus propios **registros** pero con el mismo contenido, el mismo **PC**, etc.
Tiene aridad 0 (no toma argumentos) y devuelve un entero; 0 para el proceso child, y el pid del child para el proceso padre (o un número negativo si falla). Sus ejecuciones no son deterministas; son dos procesos diferentes y el scheduler va a determinar cual se ejecuta en cada momento.

Exec() system call
Es una familia de system calls (por ej. execv) que se utilizan para correr un programa diferente al programa desde el cual se la llama. Usa el nombre de un programa como argumento (argv), carga el **ejecutable**, y sobreescribe el segmento de código actual. Luego, el SO corre ese programa.

- *execv()*: el primer argumento es el path al programa, el segundo argumento es un array de punteros a los argumentos, terminado en NULL.
- *execvp()*: el primer argumento es el nombre file del programa (se busca en el path actual), el segundo argumento es un array de punteros a los argumentos.
- *execl()*: se pasan los argumentos como una lista explícita en la llamada, uno por uno, terminando con NULL.

Wait() system call
Es usada por un proceso parent para **esperar** a que el proceso **child** termine de ejecutarse. Recién en ese momento el parent continúa su ejecución.
Si un parent tiene múltiples childs, puede usar la versión waitpid() para especificar el pid de un child específico al cual esperar (si no, el retorno del wait se vuelve no determinista)

La separación de fork() y exec() (diferenciando la creación de la ejecución) da la oportunidad de ejecutar procesos de diversas formas y con varias funcionalidades en el medio, ya que permite cambiar las **variables de entorno** y los **archivos abiertos** (file descriptors) en el espacio intermedio entre la ejecución de un proceso y otro.
Por ejemplo, permite utilizar una pipe (|) en shell para redireccionar (usando open, close y dup (duplicar el fd)) la salida de un comando/proceso hacia el input del siguiente.

Control de procesos y usuarios

kill(): familia de system calls usadas para enviar distintas señales (**signals**) asíncronas a un proceso, por ejemplo SIGSTOP, SIGKILL, SIGCONT, etc.
signal() es usada por los procesos para hacer catch de señales enviadas al mismo.
Esta es una forma primitiva de comunicación entre los procesos.

Con la noción de usuario se limita quién gana control sobre los recursos del sistema y quien puede controlar todo los procesos, o solo los propios (por razones de seguridad).

06: Ejecución directa limitada (LDE)

Para virtualizar el CPU el SO necesita compartir el CPU físico entre varios trabajos simultáneos. La idea principal es ejecutar uno un poco y cambiar a otro rápidamente generando la ilusión de procesamiento simultáneo. Con **Time Sharing** se alcanza la virtualización.
Además, debe lograrse asegurando la **performance** (virtualizar sin sobrecargar el sistema) y el **control** (como correr cada proceso mientras mantenemos control del CPU).

Ejecución directa limitada

Consiste en correr directamente el programa en la CPU; el SO hace los preparativos (crea una nueva entrada para el proceso en la tabla de procesos (crea un nuevo PCB), reserva memoria para el programa, carga desde el sistema de archivos el código ejecutable a la memoria, pone el argv y argv en la pila (stack), limpia los registros, y llama al main) y lo ejecuta.

Operaciones restringidas

La ventaja del método de ejecución directa limitada es su ejecución rápida, pero debe controlarse que el proceso no haga cosas que no queremos que haga sin el apoyo del SO. Para ello, se usan operaciones restringidas al modo en el que sean ejecutadas.

En **user mode** (restringido) el código corre con restricciones (como accesos a I/O o a memoria no permitida; hacerlo llevaría a una excepción, lo que haría que el SO termine el proceso).
En **kernel mode** (privilegiado), modo en el que funciona el SO, el proceso no tiene restricciones y puede hacer operaciones privilegiadas.

El modo de usuario cuenta con System calls para solicitar operaciones que tiene restringidas. Estas instrucciones ejecutan una **trap** que salta a kernel mode elevando el privilegio y realizando la instrucción (si el SO la permite). Al terminar, se realiza un **return from trap** y baja el nivel de privilegio volviendo a user mode.
Antes de ejecutar una trap se guardan los registros (contexto) del proceso que llamó a la trap, en un **kernel stack** (uno por proceso), para su posterior restablecimiento al volver a user mode. El kernel debe verificar que código ejecutar cuando ocurran determinadas excepciones, para lo cual el SO setea una **trap table** al momento del booteo que establece eso y además indica la localización de los **trap handlers**. Estos últimos serán ejecutados por el SO en modo kernel.

Esta indirección garantiza seguridad y genera una abstracción que permite cambiar de kernel mientras se mantenga el número id de las system calls.

Una trap puede ser ocasionada por:

- Un timer interrupt (evento asíncrono)
- Un hardware device interrupt (evento asíncrono)
- Una syscall (evento síncrono)
- Una exception (errores, acceso indebido a memoria; evento síncrono)

El SO reacciona a cualquiera de ellas de la misma forma (trap handler).

Para especificar la system call, se usa una **system-call number**; el código que el usuario señala que quiere ejecutar y que el kernel en el **trap handler** verifica y, si es válida, ejecuta. Esto ofrece protección para no darle control total al usuario (para que no salte a una dirección de memoria no autorizada), pero falla al no controlar el input que el usuario usa como argumento de las syscalls.

En **LDE** en el booteo inicia la trap table y el CPU recuerda su localización (operación privilegiada). Luego, cuando corre un proceso, el kernel configura algunas cosas (nodo en el process list, allocating memory) antes de hacer el return from trap, para ejecutar el proceso cambiando el CPU a user mode. Más adelante, si el proceso quiere llamar a una system call, trapea de vuelta al SO que maneja la llamada y luego vuelve con un return from trap a user mode.

Cada proceso tiene un kernel stack, donde los registros (incluyendo el PC) se guardan y restauran (por hardware) cuando se entra o sale del kernel.

SO @ boot (al iniciar) (modo kernel)	Hardware	
inicializar tabla de traps		
	guardar dir. del gestor de llamadas al sistema	
SO @ corriendo modo kernel	**Hardware**	**Programa modo usuario**
Crear entrada en la lista de procesos Asignar memoria al programa y cargarlo en memoria Poner argc/argv en el stack Llenar el kernel stack con reg/PC **retornar de la trap**		
	restaurar registros (del kernel stack) cambiar a modo usuario saltar a main	
		correr main() ... realizar la llamada al sistema **trap** al SO
	guardar registros (en el kernel stack) cambiar a modo kernel saltar al trap handler	
Encargarse de la trap Hacer el trabajo de la llamada al sistema retornar de la trap		
	restaurar registros (del kernel stack) cambiar a modo usuario saltar al PC	
		... retornar de main **trap** (por exit())
Liberar la memoria Borrarlo de la lista de procesos		

Línea del tiempo de las dos fases del protocolo de LDE ante una trap por instrucción privilegiada.

Intercambio entre procesos

Si un proceso corre en el CPU, el SO no está corriendo. Sin embargo, el mismo debe asegurarse poder recuperar el control para cambiar de proceso y lograr time sharing.

Enfoque colaborativo: esperar a nuevas system calls

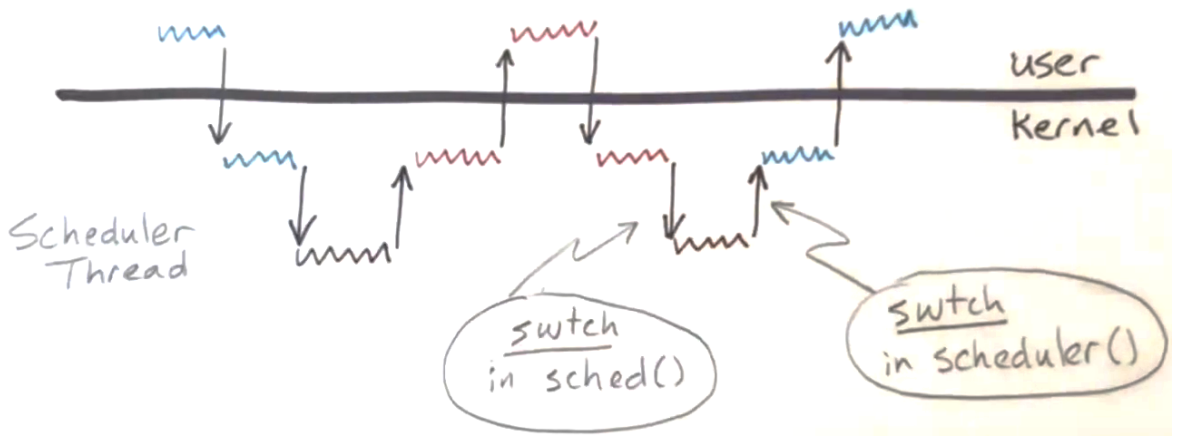
El SO confía en el proceso en ejecución y da por sentado que cada cierto periodo liberaría el CPU para que el SO decida que correr; al ejecutarse una trap o al llamar a una System call (incluso de manera explícita, con la llamada **yield**) se cedería y transferiría el control al SO.

Enfoque no colaborativo: el SO toma el control

Utilizando mecanismos del hardware, el SO puede retomar el control del CPU mediante el **Timer interrupt**; un dispositivo que cada varios milisegundos realiza una interrupción. Esta cuando ocurre se ejecuta un **interrupt handler** en el SO, que le permite retomar el control.

En el booteo el SO deja explicitado que código correr en una interrupción, momento en el que además comienza el timer.

El hardware se encarga de guardar el estado del proceso actual al momento de la interrupción para su posterior return-from-trap.



En cada proceso una porción se ejecuta en user mode y otra en kernel mode. Para pasar al scheduler, se da un segundo cambio de contexto; de kernel mode al scheduler.
En el cambio de contexto de user mode a kernel mode, debemos guardar todo. En el cambio de contexto de kernel mode al scheduler, en cambio, podemos hacer algunas asunciones.

Guardar y restaurar el contexto

Si cuando el SO toma control y decide cambiar a otro proceso (usando la rutina switch), ejecuta un código de bajo nivel, el **context switch**, que le permite guardar los valores de los registros del programa en ejecución (en el kernel stack del proceso) y restaurar otros para el proceso que pasará a ejecutarse (desde su kernel stack).

Si hay un **timer interrupt** (trap), es el **hardware** quien guarda los registros en el stack de kernel.
Si hay un **switch** por parte del SO, es el **software** quien guarda/restaura los registros del kernel en la estructura del proceso.

Cada vez que se da un context switch, por ej. ante una syscall exception, el proceso pierde performance, y tarda determinados ciclos en retomar la velocidad que tenía antes de la misma.

Concurrencia

Para que no ocurran interrupts simultáneas, se suelen deshabilitar las interrupciones mientras se está lidiando con interrupciones. También se usan **locks** para proteger las estructuras internas.

SO @ corriendo modo kernel	Hardware	Programa modo usuario
		Proceso A ...
	interrupción por tiempo guardar regs(A) → k-stack(A) cambiar a modo kernel saltar al gestor de traps	
encargarse de la trap Llamar a la rutina switch() guardar regs(A) → proc t(A) restaurar regs(B) ← proc t(B) cambiar al k-stack(B) retornar de la trap (a B)	(del kernel stack)	
	restaurar regs(B) ← k-stack(B) cambiar a modo usuario Saltar al PC de B	
		Proceso B ...

Línea del tiempo de la segunda fase del protocolo de LDE ante una interrupción por tiempo.

En resumen, en LDE se ejecuta **directamente** al programa en el CPU, habiendo configurado antes al hardware para poder **limitar** lo que cada proceso puede ejecutar, y habiendo el SO preparado a la CPU configurando al momento del boot al controlador de **traps** y al **timer** de interrupciones, y luego ejecutando procesos solo en **modo** restringido.

07: Planificación de la CPU

Las políticas de alto nivel (**scheduling policies**, o **disciplines**) son algoritmos utilizados el scheduler del SO para decidir qué procesos ejecutar y en qué orden.

Presunciones sobre la carga de trabajo

Antes de ver cómo se desarrolla una política, hacemos algunas suposiciones sobre los procesos que corren en el sistema (la carga de trabajo, o workload), las cuales no son realistas:

1) Todos duran lo mismo hasta acabar.
2) Todos llegan al mismo tiempo.
3) Una vez empiezan, se ejecutan hasta acabar.
4) Solo usan CPU (no I/O).
5) Sabemos de antemano el tiempo que toma su ejecución.

Métricas de la planificación

El **turnaround time** (T-turnaround) es la métrica que mide el tiempo total que tarda un proceso desde que llega al sistema hasta que finaliza:

T-turnaround = T-completion - T-arrival

Tiempo de entrega = Tiempo de finalización - Tiempo de llegada

Si se cumple la presunción 2), Tarrival = 0, lo que hace Tturnaround = Tcompletion

T-turnaround es una métrica de **performance** (rendimiento). T-response time era una métrica es **fairness** (que tan justo es), que se suele contrastar con la de performance.

Política: FIFO (First In, First Out) (también llamado FCFS: First Come First Serve)
Es simple y fácil de implementar pero con mal T-turnaround. Puede sufrir de **convoy effect**, es decir, puede llegar antes un proceso más largo que los demás y ralentizar al resto, teniendo a procesos más cortos en espera, ya que cada proceso corre hasta finalizar, aumentando así el tiempo promedio de entrega del sistema.

Política: Shortest Job First (SJF)
Siempre corre el proceso **más corto** primero, minimizando así el T-turnaround promedio.

Si llega primero un proceso largo será ejecutado, y si luego llega uno más corto deberá esperar a que finalice el primero, empeorando el T-turnaround (se genera el mismo **convoy effect**).

Política: Shortest Time-to-Completion First (STCF) (llamado PSJF: Preemptive Shortest Job First)
Los procesos no corren hasta acabar. El scheduler puede hacer **preempt** de un trabajo (darle prioridad) y realizar un context switch en los momentos en los que el SO retoma el control del CPU (interrupts, syscalls, etc). Tiene mal response time.

Métrica: Response Time
Las computadoras actuales deben tener una performance interactiva con el usuario, y esto se mide en Response Time:

T-response = T-firstrun - T-arrival

T de respuesta: Tiempo desde que el proceso llega hasta que es ejecutado por primera vez;
Primera ejecución - Momento de llegada.

Política: Round Robin (RR)
Corre los procesos durante un periodo de tiempo fijo o hasta que acaben (este segmento, **time slice,** es llamado **quantum**). Al ser cortado por un Q, el proceso va al final de una FIFO. Cambia de proceso cada cierto tiempo fijo de entre los trabajos que hay en una cola hasta que finalicen.
Este time slice debe ser **múltiplo** del **timer** interrupt del sistema para que el SO pueda tomar control y hacer el context switch en ese momento.

La duración del time slice es importante; si es muy corta es bueno para el T-response pero puede empeorar la performance al aumentar la cantidad de context switchs. A la vez, debe ser suficientemente larga para amortizar el costo del cambio y no empeorar la performance, y corta como para mantener un T-response aceptable.

RR es **fair** (justa), lo que hace que sea mala en su tiempo de entrega. Es un trade-off a considerar dependiendo del objetivo que se tenga con el scheduler.

Ante dos procesos que arriban a la vez se debe establecer una política que determine cual se ejecuta, pero si uno ya estaba ready antes que otro, se respeta el orden FIFO, no el de la política.

Incorporando I/O

Todos los programas que usan I/O no usan la CPU mientras realizan ese trabajo, ya que esperan a que este se complete (quedando el proceso en estado Blocked). Durante este tiempo, el scheduler puede correr otro proceso. Cuando el I/O termina, el Scheduler también decide qué hacer; si lo deja en espera (Ready) o si lo ejecuta nuevamente (Running, con un context switch).

De esta forma, se utiliza con más eficiencia el CPU, superponiendo cortos periodos de uso del CPU y el I/O a la vez entre programas, tratando cada pequeño tiempo de uso del CPU como un proceso y maximizar el uso de los recursos.

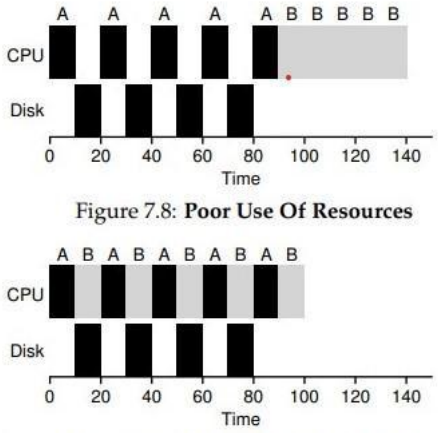


Figure 7.8: **Poor Use Of Resources**

Uso de los recursos con y sin superposición de procesos.

No más oráculo

Relajamos 5), normalmente no se sabe la longitud de un proceso. Sin esto SJF y STCF no funcionan bien.

08: Cola multinivel con retroalimentación (MLFQ)

Política: **MLFQ** (Multi-Level Feedback Queue)**.** Busca optimizar tanto el *turnaround time* como el *response time*, es decir, tener a la vez buena performance y ser interactivo, brindando una respuesta adaptativa a la carga.

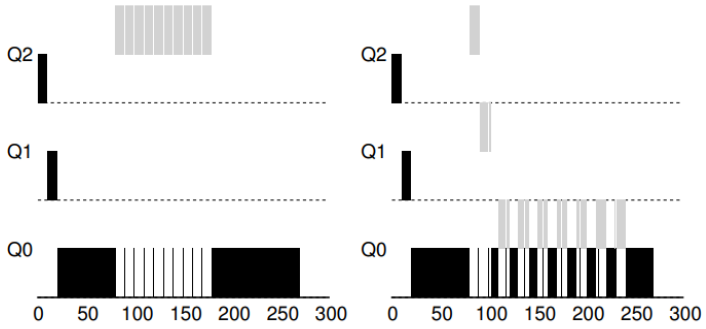
Reglas básicas

MLFQ cuenta con diferentes colas (queues) con un diferente nivel de prioridad cada una. Los procesos en estado *ready* quedan posicionados en ellas y el MLFQ decide cuál correr en base a la prioridad de cada proceso. Esta prioridad varía en base al *comportamiento observado* anteriormente; Si el mismo normalmente suelta el CPU para esperar un I/O (proceso I/O-bound), el MLFQ mantiene su prioridad alta viéndolo como un proceso interactivo. En cambio, frente a usos prolongados del CPU (proceso CPU-bound), el MLFQ le baja prioridad.

La MLFQ trata de aprender del comportamiento de los procesos para predecir el *futuro comportamiento.* Es decir, la **prioridad cambia** con el tiempo.

Además, periódicamente (cada un tiempo S) se aumenta la prioridad de todos los procesos (**priority boost**) poniéndolos en la cola de mayor prioridad. Esto garantiza que los procesos no se queden sin CPU time (evita **starvation** si se juntan demasiados procesos interactivos), que si un **long-running job** se vuelve **interactivo** el scheduler lo trate como tal (en vez de quedarse en las colas de baja prioridad), y que los procesos no puedan **jugar** con el **scheduler** (liberando el CPU justo antes de terminar su time slice para mantenerse en prioridad alta y monopolizar su uso).

La duración de S debe ser bien elegida para que no haya starvation (si es muy largo) y que los procesos interactivos funcionen eficientemente (no muy corto).
Por otra parte, para prevenir que un proceso juegue con el scheduler para evitar bajar de cola de prioridad, se le da a cada proceso un tiempo total (dependiendo de la cola; time slices más cortos a mayor prioridad) que no debe sobrepasar, independientemente de si fue un uso de la CPU ininterrumpido o si hubo cambios de contexto en el medio.



Ejemplo de ejecución sin (izq.) y con (der.) gaming tolerance

Las decisiones sobre la cantidad de colas, la duración de los slices, el quantum de cada cola, o cada cuánto hacer un boost, depende de los objetivos del planificador, y varían de un SO a otro. Son decisiones fundamentales, y suelen ser llamadas **constantes vudú**. En algunos SO, se permite al usuario establecer sugerencias sobre la prioridad de algunos programas.

En resumen, las reglas de la MLFQ consisten en:

1) Si Prioridad(A) > Prioridad(B), se ejecuta **A** (B no).
2) Si Prioridad(A) = Prioridad(B), se ejecutan A y B en **RR**.
3) Cuando un trabajo ingresa al sistema, se coloca en la cola de **prioridad** más **alta**.
4) Una vez que un trabajo utilice su **tiempo** asignado en un nivel dado (independientemente de cuántas veces haya renunciado a la CPU), su **prioridad** se **reduce**.
5) Después de un periodo de tiempo determinado (**S**), todos los trabajos del sistema se mueven a la **cola** de más **alta prioridad**.

Con esta última incorporación, las políticas de planificación estudiadas son:

- FIFO: First In First Out (starvation, trade off turnarund)
- SJF: Shortest Job First (starvation, trade off turnarund)
- STCF: Shortest Time to Completion First
- RR: Round Robin (quantum, response)
- MLFQ: Multi Level Feedback Queue (quantum, response)

13: El espacio de direcciones

Multiprogramación y Time Sharing

Cuando muchos procesos comenzaron a correr al mismo tiempo, el SO debió comenzar a mediar (switchear entre los procesos) para lograr una mayor eficiencia económica en el uso del CPU. Además, se volvió importante la noción de interactividad.

Una forma de resolver esos problemas fue el time sharing; ir intercambiando entre los procesos (en cada cambio guardar todo su estado y registro en el disco) hasta que todos terminen. Esto resulta lento y tiene mal rendimiento cuanto más crece la memoria.

En el mejor caso, conviene dejar los procesos en memoria y cambiar entre ellos sin guardar en disco cada vez. Al haber varios programas a la vez en memoria, la protección se volvió importante. Los procesos no debían leer o escribir en la memoria de otros procesos.

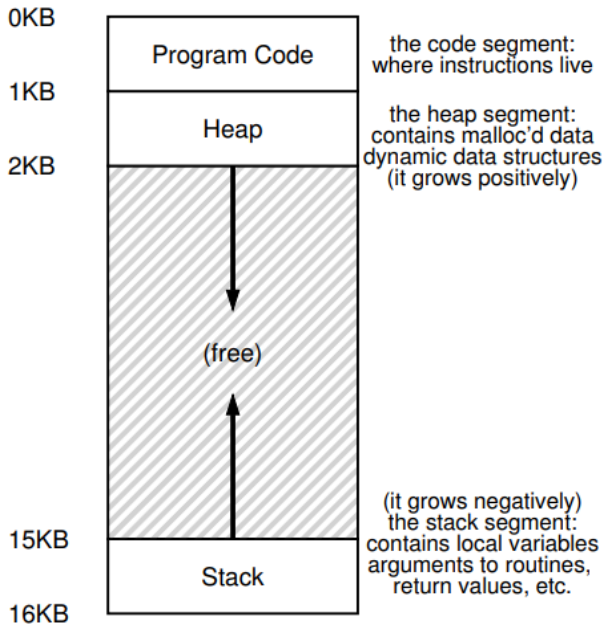
El espacio de direcciones

El **address space** es la abstracción de la memoria física que crea el SO, y es lo que ve un programa corriendo; la virtualización de memoria que le proporciona a los procesos la ilusión de un espacio de memoria amplio y privado.

El address space (espacio **direccionable**) de un proceso contiene todo el estado de la memoria del programa en ejecución; el **código** mismo del programa, el **stack** y el **heap** (por el momento ignoraremos otros elementos como variables estáticas):

El stack es usado para guardar la cadena de llamadas a función; dirección de retorno, variables locales y parámetros.

El heap se utiliza para almacenar elementos dinámicamente (dynamically allocated), es decir, es memoria manejada por el usuario (usando funciones como malloc en C).



Ejemplo de Address Space. La dirección 0x00 es virtual;
en la realidad el programa se encuentra en una dirección arbitraria de la memoria física.

El código tiene tamaño fijo (pero no es necesariamente estático; puede ser self-modifying code) lo cual lo hace fácil de poner en memoria. En cambio, el stack y el heap pueden crecer o decrecer mientras el programa corre. Al poner en forma opuesta el heap y stack (por convención) podemos permitirles **cambiar** su **tamaño** siguiendo direcciones opuestas (luego, al trabajar con múltiples hilos esta forma simple de ubicarlos ya no sirve).

La dirección en la que el proceso se ve a sí mismo es su **memoria virtual**, y en base a ella hace requisitos al SO, el cual debe traducir estas **direcciones virtuales** a memoria física real a la hora de responder los pedidos del programa (como, por ej., guardar un archivo), para lo cual utiliza la ayuda del hardware.

Objetivos
El trabajo del SO es virtualizar la memoria. Para hacerlo bien, debe cumplir 3 objetivos:

1) **Transparencia**: la implementación de la memoria virtual debe "ser invisible" para el programa, el cual debe creer que tiene su propia memoria física. El SO junto al hardware crean esta ilusión.
2) **Eficiencia**: la virtualización debe ser lo más eficiente posible en términos de tiempo y espacio. Para esto el SO utiliza distintas características del hardware, como la TLB.
3) **Protección**: el SO debe proteger los procesos unos de otros, así como proteger al SO de los procesos. Para eso aísla a la memoria de los procesos (solo pueden acceder a su address space) para que no puedan interferir entre sí, permitiendo por ejemplo, que uno falle sin que afecte al resto.

14: API de la memoria

Tipos de memoria
Corriendo un programa de C hay dos tipos de memoria:

1) El **stack**. Las asignaciones y reasignaciones las maneja el compilador **implícitamente**. El compilador asigna la memoria necesaria y cuando ya no es necesaria la desasigna.
2) El **heap**. Para cosas que requieren más permanencia se usa el heap, donde las asignaciones y reasignaciones las realiza el usuario **explícitamente**.

Malloc()
Se le da un tamaño (en bytes o un puntero a un int) pidiendo dicho espacio en memoria heap. Si lo logra, devuelve un puntero (sin tipo, para castear) a dicha memoria. Si falla, devuelve NULL.
Normalmente se utiliza sizeof() como operador para explicitar el tamaño requerido en memoria.

calloc() además inicializa la memoria asignada en cero.
realloc() copia una región de memoria y le asigna un espacio de tamaño diferente.

En caso de strlen(s) recordar agregar un +1 para el carácter end-of-string.

Free()

Para liberar memoria asignada en el heap se llama a free() con el puntero devuelto por malloc() como argumento; el tamaño de la memoria a liberar es buscado automáticamente por la librería de asignación de memoria.

Tanto malloc() y free() no son system calls; son parte de la librería de manejo de memoria stdlib.
Las systemcalls son brk() y sbrk(), que mueven el break; la memoria máxima a la que puede acceder el heap, y mmap(), que permite mapear un archivo en memoria. Usualmente no se usan.

Errores comunes

Algunos lenguajes tienen un manejo de memoria automático (**garbage collector**), lo que hace que no sea necesario liberar memoria explícitamente. En aquellos que no lo tienen (como C), algunos errores son comunes:

- Olvidarse de asignar memoria: **Segmentation fault**.
  Muchas rutinas esperan ya tener memoria asignada al ser llamadas (por ej. strcpy()).
- No asignar memoria suficiente: **Buffer overflow**.
  Normalmente se asigna la memoria justa, algunas veces malloc() deja un poco de margen para evitar errores. Cuando no es suficiente, un overflow de bytes puede ocurrir.
- Olvidarse de inicializar la memoria asignada: **Uninitialized read**.
  Si se llama a malloc() pero no se le asignan valores a la memoria asignada, el programa eventualmente puede acceder a ella y leer del heap información de valor desconocido, causando comportamientos no previstos.
- Olvidarse de liberar memoria: **Memory leak**.
  Cuando ya no se usa, la memoria debe ser liberada. De no hacerlo, puede llevar a acabar la memoria en aplicaciones long-running.
- Liberar memoria que todavía se puede necesitar: **Dangling pointer**.
  Si un programa libera memoria antes de usarla se queda con un puntero colgante, llegando a crashear o sobreescribir memoria válida.
- Liberar memoria repetidamente: **Double free**.
  Los programas pueden intentar liberar memoria más de una vez, lo cual genera comportamientos indefinidos o crasheos.
- Llamar a free() incorrectamente: **Free incorrectly**.
  free() solo espera que se le pase como argumento un puntero devuelto por malloc(). Con cualquier otro valor o argumento el free() es invalido.

15: Traducción de direcciones

En el desarrollo de la virtualización del CPU nos centramos en el mecanismo general Limited Direct Execution (LDE), cuya idea es dejar el proceso correr en el hardware la mayor parte del tiempo, y que en ciertos puntos clave el SO se involucre y tome decisiones que le permitan asegurarse, con ayuda del hardware, mantener el control mientras trata de mantenerse fuera del camino del proceso.

En la **virtualización** de la memoria se busca algo similar; obtener **control** y **eficiencia** mientras se provee la virtualización. La eficiencia es lo que dicta que se use el apoyo del hardware. Controlar implica que el SO asegure que ninguna aplicación tenga permitido acceder a otra memoria salvo la suya, y así proteger aplicaciones unas de otras y al SO de las aplicaciones (lo que también requiere ayuda del hardware).

Algo más que necesitaremos del sistema de memoria virtual (VM o MV) es **flexibilidad**; que los procesos puedan usar su address space como quieran, haciendo así el sistema más sencillo.

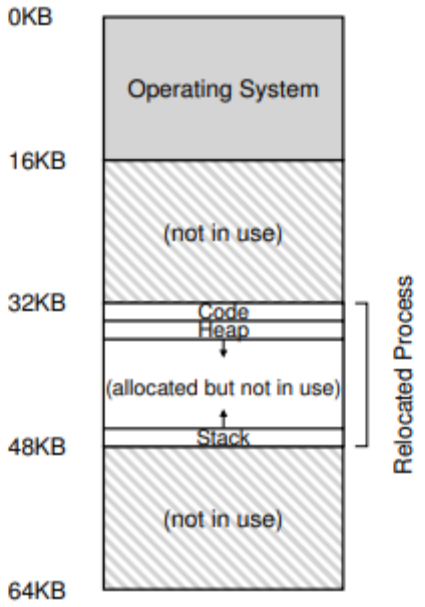
Virtualizar memoria eficiente y flexiblemente

La idea es realizar una **hardware-based address translation** (traducción de direcciones en base a hardware, o solo address translation); en cada referencia a memoria se hace una traducción de dirección por el hardware para redireccionar la referencia a memoria de la aplicación (dirección virtual) a su localización real en memoria.

El hardware no virtualiza la memoria, solo provee un mecanismo de bajo nivel para lograrlo eficientemente. Es el SO quien se involucra y maneja la memoria, sabiendo qué localizaciones están libres y cuales en uso, y manteniendo así control sobre cómo la memoria es usada.

Suposiciones

Por ahora supondremos, por simplicidad, tres cosas irreales: que el address space del usuario está localizado de forma *continua* en memoria física, que el tamaño del *address space es menor* que la memoria física, y que todos los address space tienen el *mismo tamaño*.



Ejemplo de programa cargado en memoria. Su dirección inicial es 0 (no 32KB) y crece hasta un máximo de 16KB (hasta los 48KB). Cualquier referencia a memoria que el programa haga debe estar dentro de estos límites. El primer bloque de memoria es para el SO mismo.
El valor de su registro Base es 32K, y el de Bound es de 16K.

Reubicación dinámica de memoria (apoyada en el hardware)

Para virtualizar la memoria, el SO debe poner a cada programa en un lugar diferente a la dirección 0. Para hacerlo de forma transparente (sin que el proceso se de cuenta), se usan las ideas de base and bounds y dynamic relocation (reubicación dinámica):

Se utilizan 2 **registros** de **hardware**; uno llamado **base** register y el otro **bounds** (*límite*). La base es el inicio del proceso en la memoria física, es decir, el desplazamiento (offset) desde la posición 0. El límite marca hasta donde se puede acceder a memoria, de manera continua desde la base. Este par de registros nos permiten direccionar espacio en cualquier lugar de la memoria física y al mismo tiempo asegurar que el programa solo accede a su propio address space (protección).

Como cada programa cree estar en la dirección 0 y es el SO quien al cargarlos decide dónde ponerlos en memoria física, y establece los registros base y bounds con ese valor. Cualquier referencia a memoria generada por el programa será traducida por el procesador usando:

physical address = virtual address + base

Cada referencia a memoria creada por el proceso es una **dirección virtual**, el hardware almacena los contenidos de la base a la dirección y el resultado es la **dirección física**.

El transformar una dirección virtual en una física es a lo que nos referimos con **address translation** (traducción de dirección). El **hardware** toma la dirección virtual que el proceso cree referenciar y la transforma en la memoria física donde está realmente la información. Debido a que esto ocurre **durante** la **ejecución**, y porque podemos mover el address space incluso después de que el programa comenzó a ejecutarse, la técnica se llama **dynamic relocation**.

En todo este proceso el SO **verifica** que la dirección a la cual quiere acceder el proceso esté dentro de los límites de su address space con el registro bounds. En caso de que el proceso acceda algo fuera de su address space (o una dirección negativa) el CPU levanta una excepción.

Base y Bounds son estructuras de datos del hardware mantenidas en el chip (un par por CPU) en la **MMU** (**memory management unit**). La MMU es el hardware encargado de traducir las direcciones de memoria virtuales en direcciones físicas (integrando paginación en el proceso).

Apoyo del Hardware

El hardware debe proveer al SO distintas mecanismos para lograr la virtualización de la memoria:

- La posibilidad de soportar dos modos de ejecución (usuario y kernel)
- Proporcionar los registros **Base** y **Bound**.
- La capacidad de **chequear** que la memoria a la que se intenta acceder se encuentre dentro de los límites de base/bound y, en ese caso, **traducir** la memoria virtual/física.
- Otorgar las **instrucciones privilegiadas** para que el SO pueda **modificar** los registros Base y Bounds (igual que poder modificar los exception handlers), mientras un proceso está en ejecución.
- Generar excepciones cuando un programa trata de acceder a memoria ilegal o fuera de su address space, parando el proceso y retornando el control al SO corriendo el **exception handler**; lo mismo que ocurre si un proceso trata de ejecutar una instrucción privilegiada.

Problemas del SO

Usando las herramientas proporcionadas por el hardware, el SO logra la virtualización de la memoria. Para ello, cuenta con tres responsabilidades:

1) Administración de la **memoria** (heap): Encontrar **espacio** para el address space de un proceso cuando este es creado, para lo cual el SO busca el espacio en la estructura de datos (**free list**) y la asigna.
   Luego, cuando un proceso termina (por sí mismo o la fuerza por el SO) debe quitarlo del scheduler, reclamar la memoria y agregar dicho espacio a la free list.

2) Manejo de los registros **base/bounds**: debe **guardar** y **restaurar** los registros Base y Bounds en cada **context switch**, guardar sus valores en memoria (en alguna estructura por cada proceso, como la estructura del proceso mismo (**process structure** o **process control block**; **PCB**)), y al restaurarlos debe pasarle dichos valores al CPU.

3) Definición de los **exception handlers** (manejo de excepciones) en el momento de booteo, para luego ser ejecutados en caso de accesos a memoria ilegal (errores **out of bounds**; fuera de rango) o intentos de uso de instrucciones privilegiadas.

OS @ boot (kernel mode)	Hardware	(No Program Yet)
initialize trap table		
	remember addresses of...	
	system call handler	
	timer handler	
	illegal mem-access handler	
	illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

Ejecuciones a tiempo de booteo para permitir un LDE con dynamic relocation.

17: Gestión de espacio libre

El manejo del espacio libre resulta sencillo cuando el espacio está dividido en unidades de espacio constante; si se solicita espacio solo entregar la primera entrada libre (**paginación**). Pero se vuelve difícil al tratar con bloques de espacio libre de diferentes tamaños.

Suposiciones

Contamos con una interfaz para administrar la memoria que provee las funciones **malloc**(size_of size) y **free**(), tal y como las describimos en capítulos anteriores. Esta librería maneja la memoria heap, y la estructura genérica que maneja el free space es la **free list**, la cual contiene una referencia a todos los bloques de espacio libre en la región que maneja la memoria.
Esta abstracción se usa tanto en espacio de usuario como en espacio de kernel.

Una vez se llama a malloc() con memoria dentro del heap, esta no puede ser tocada por la librería hasta que se use free(). Esto implica que hasta ese momento no es posible compactar el espacio libre (porque no podemos mover el bloque ya asignado). La compactación puede ser usada por el SO para combatir la fragmentación de usar segmentación.
También asumimos que el allocator maneja una región de bytes continua (no va a crecer).

Mecanismos de bajo nivel: División y fusión (Splitting and Coalescing)
Una free list contiene un conjunto de elementos que describen los espacios libres que quedan en la memoria heap. Consiste en una serie de listas ligadas con la dirección de los bloques, su longitud, y la dirección del siguiente espacio libre. Si se solicita un lugar mayor al disponible (puede que el espacio esté desocupado pero no en un bloque continuo) el pedido falla.

Los allocators cuentan con dos mecanismos usados para administrar el espacio libre:
Splitting: Cuando se solicita memoria, el allocator busca un bloque de memoria que satisfaga el pedido. Si el mismo tiene un tamaño mayor al requerido, lo partirá en dos y devolverá un puntero al primero (del tamaño justo necesario), manteniendo al segundo en la free list.
Coalescing: Consiste en combinar los bloques de espacio libre continuos en uno solo. Este procedimiento se realiza cuando se libera espacio.

Seguimiento del tamaño de las regiones asignadas
Al llamar a free() no se especifica el tamaño de la región a liberar; la librería es capaz de darse cuenta sola y recuperar la memoria. Para lograr esto, la mayoría de allocators guardan información en un **header** block al inicio de cada bloque de memoria asignado.
El header contiene el **tamaño** de memoria asignada a ese bloque. Además, puede contener **punteros** para acelerar la recuperación de memoria, un "número mágico" para verificar la **integridad**, y otra información extra.
Notar entonces que el tamaño del bloque de memoria, es decir el espacio a liberar, ahora puede saberse rápidamente y consiste del tamaño del espacio solicitado más el tamaño de la estructura de su header *size* y *magic* (mismo al hacer alloc). Esto es, tamaño solicitado + **8 bytes**.

Encastración en una lista de espacios libres
La free list se encuentra en un bloque dentro del espacio libre mismo.
Cuando se solicita espacio, se asigna al comienzo de la free list (header + bloque). Cada vez que se recibe un nuevo pedido, es asignado al final del bloque anterior (siempre y cuando el espacio disponible sea el suficiente).
Al liberarse un bloque, la celda "magic" pasa a ser un puntero "next" al siguiente chunk. De haber dos bloques contiguos que cumplan esa condición, se los une y se modifica el puntero next.

Aumentando el tamaño del heap
Los allocators suelen comenzar con una memoria heap pequeña e ir pidiendo más espacio al SO a medida que lo necesitan. Este, si tiene éxito, les devuelve la dirección del nuevo final del heap.

Estrategias básicas
El allocator ideal es rápido, eficiente en el uso del espacio (minimizando fragmentaciones) y escalable. No hay un único enfoque mejor que el resto, pero existen varias **políticas** de manejo de espacio libre que persiguen ese objetivo:

1) **Best fit**

Se busca a través de la free list al bloque libre más **pequeño** de los espacios iguales o superiores al solicitado. Al hacer de una búsqueda **exhaustiva** en la free list, genera una penalización de performance, y una fragmentación en espacios libres pequeños.

2) **Worst fit**

Se busca al bloque más **grande** disponible, se usa el espacio necesario, y se devuelve lo restante a la free list. Genera los mismos **overheads** al realizar también una búsqueda **exhaustiva** (fragmentando esta vez en bloques libres grandes).

3) **First fit**

Usa el **primer** bloque de la lista lo suficientemente grande para cumplir con lo solicitado. Su ventaja es la **velocidad** ya que evita realizar una búsqueda exhaustiva, pero "**contamina**" el **comienzo** de la free list al concentrar allí la fragmentación en bloques pequeños.

4) **Next fit**

Igual a First Fit pero utiliza un puntero que le permite comenzar la búsqueda desde la **última posición** revisada la vez anterior. Desparrama la fragmentación a lo largo de la free-list y mantiene la **velocidad** del enfoque anterior, pero requiere un puntero extra en la implementación.

5) **Listas segregadas**

Ante aplicaciones que tengan peticiones **recurrentes** de tamaño similar, se crea una nueva lista para el manejo de objetos de ese tipo, y se envían las demás peticiones al allocator general. La fragmentación es menor y los pedidos de dicho tamaño se satisfacen más rápido.

Por ejemplo, el **slab allocator** asigna un número de **object caches** para objetos del kernel que se solicitan seguido. Si le falta espacio pide más slabs (bloques pequeños) de memoria. Este allocator también mantiene los free objects de las listas en un estado pre inicializado.

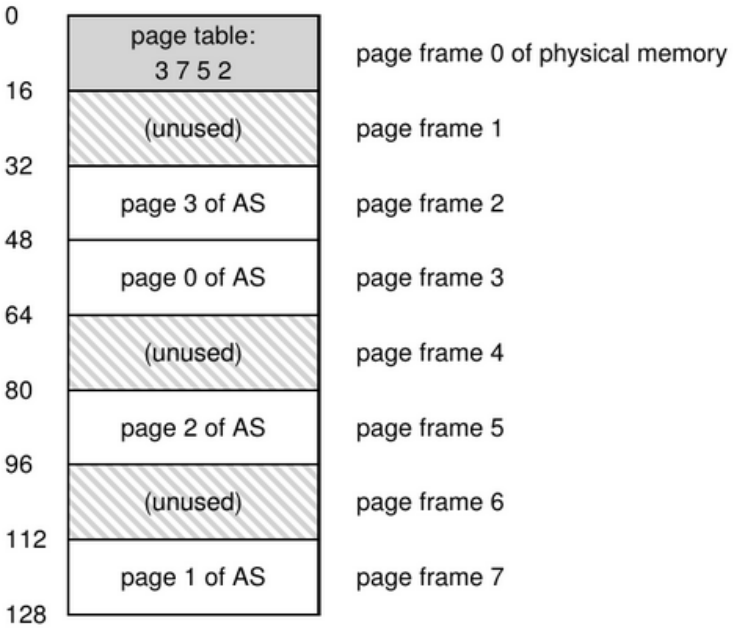
6) **Buddy Allocation**

Tanto la memoria libre como la memoria que se asigna son espacios de tamaño $\mathbf{2^N}$. Cuando se solicita un **bloque**, se divide el espacio libre por 2 hasta encontrar uno que satisfaga el pedido.

Cuando un bloque se libera se chequea que su "buddy" del mismo tamaño esté libre, y si lo está los combina, y así recursivamente hasta hacer coalescing de todo o encontrar un "buddy" en uso; simplifica el **coalescing** pero genera **fragmentación** interna.

18: Introducción a la paginación

Si se corta el espacio disponible en bloques de tamaños diversos se genera fragmentación, lo que complica la asignación de espacio mientras más espacio se ocupa.
Un enfoque diferente consiste en cortar el espacio en partes iguales de un tamaño estándar. Esto en memoria virtual se llama **paginación** (**paging**) y cada unidad es una **página** (**page**). A la memoria física se la ve como un array de slots iguales llamados **marco de página** (**page frame**).



Ejemplo de Page Table en memoria física.

Descripción general

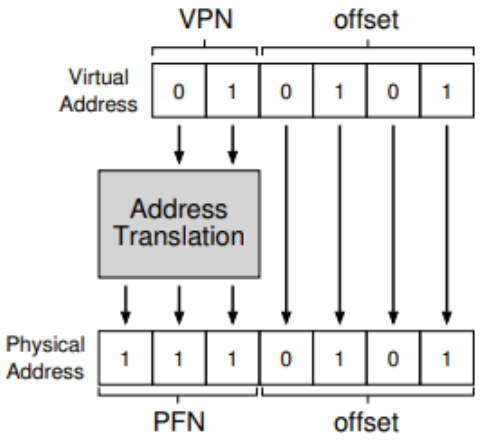
La paginación tiene varias ventajas comparado con los enfoques anteriores:

Flexibilidad: es capaz de apoyar la abstracción del address space eficientemente, más allá de cómo el proceso use el address space (por ej., no se asume la dirección de crecimiento del heap/stack o cómo son usados).

Simplicidad: permite manejar el espacio libre (la free list) tan solo otorgando la cantidad de páginas necesarias (y evitando fragmentación externa).

Para mantener un registro de donde cada página virtual del address space está situada en memoria física, el SO guarda información de *cada proceso* en una estructura conocida como **page table** (tabla de páginas), la cual almacena las **address translations** de cada página virtual.

Para que el hardware y el SO traduzcan una dirección virtual debemos dividir en 2 componentes la dirección: el **virtual page number** (**VPN**, número de página virtual) y el **offset** de la página:
Con el número de página virtual se indexa la **page table** para encontrar el marco físico (**physical frame number**; **FPN**) en el cual reside la página; luego solo se reemplazar el VPN por el PFN y se mantiene el mismo offset (el cual señala el byte, dentro de la página, que estamos solicitando).



Traducción de una virtual page number (VPN) a una physical frame number (FPN).

Page Tables

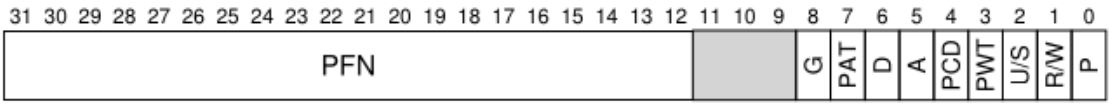
Son las estructuras de datos usadas para mapear las direcciones virtuales a direcciones físicas. Al ser largas y muchas (cada una tiene muchas **page table entry**(ies) (**PTE**); una VPN de 20 bits, por ej., implica 2^{20} traducciones) no se almacenan en la MMU, sino directamente en memoria.

Organización

La más simple es **linear page table**; la page table es vista como un array el cual se indexa con el VPN y se busca el PTE (entrada en la page table) de dicha indexación para encontrar el PFN.

Cada PTE tiene algunos bits importantes:

- **Valid bit**, permite reservar adress space al marcar páginas como inválidas, evitando tener que asignarles memoria (por ej, el espacio aún no solicitado entre el code/heap y el stack de un proceso al momento de crearlo).
- **Protection bits**, indican si una página puede ser leída, escrita o ejecutada.
- **Present bit**, indica si se encuentra en memoria física o en disco.
- **Dirty bit**, indica si una página ha sido modificada desde que fue traída a memoria.
- **Accessed bit** (“*reference”*), indica si la página ha sido accedida (útil para determinar páginas populares que deben ser mantenidas en memoria).



Ejemplo de PTE, donde P = present bit, R/W = allowed read/write, U/S = allowed access to user-mode processes, PWT/PCD/PAT/G = hardware catching, A = accessed, D = dirty, PFN.

Notar que diferentes procesos (y diferentes VPN) pueden apuntar a una misma página (dirección) física (lo que reduce el número de páginas físicas en uso). Para esos casos, puede usarse un bit **G** que indica si la página es compartida globalmente entre más de un proceso.

Tanto el valid bit como los protection bits pueden lanzar una trap en caso de intentar acceder a la página cuando no se debería poder hacerlo.

Rapidez y paginación

La paginación requiere que se realice un referencia a memoria extra para buscar la traducción de la page table (de virtual a física, de la VPN a PTE y luego a PFN), lo cual es costoso, y dado el tamaño de las page tables, se ralentiza demasiado el sistema.

Cuando corre, cada instrucción fetcheada genera dos referencias de memoria; una a la page table para encontrar el physical frame en la que la instrucción reside, y otra a la instrucción en sí misma para poder fetchearla hacia la CPU.

19: Traducciones más rápidas: TLB

La paginación requiere un gran mapeo de información, el cual normalmente está almacenado en memoria física, y la paginación requiere una búsqueda extra para cada dirección virtual generada por los procesos. Ir a memoria por la traducción antes de cada instrucción hace perder al CPU demasiados ciclos de clock en espera.

Para aumentar la velocidad de la traducción de las direcciones, el SO se apoya en el hardware; usa una parte del chip MMU llamada **translation-lookaside buffer** o **TLB**, el cual es un **caché** de traducción de direcciones virtuales a físicas, y por esto lo llamamos **address-translation caché**.

Cada vez que hay una referencia virtual a memoria, el hardware primero revisa la TLB para ver si la traducción está ahí y, de estarlo, la traducción es realizada rápidamente sin necesidad de acceder a la Page Table y causar un cuello de botella en el pipeline del CPU.

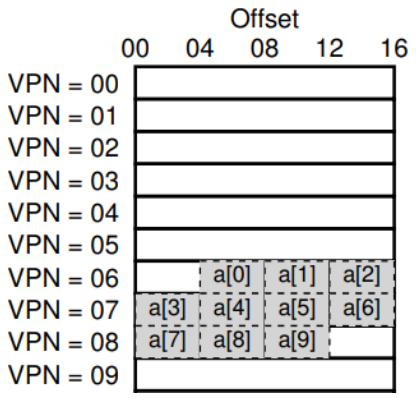
Algoritmo básico de la TLB

Supongamos una page table lineal y un TLB manejado por hardware. El algoritmo consiste en:

1) Extraer el VPN de la virtual address y revisar si el TLB tiene la traducción para este VPN.
2) Si lo tiene (**TLB hit**), obtiene el PFN de la TLB entry, lo concatena al offset de la dirección virtual original, y consigue la dirección física deseada para acceder a memoria (mientras que los chequeos de protección no fallen).
3) Si es **TLB miss** (no tiene la traducción), el hardware accede a la page table para encontrar la traducción y, asumiendo que la dirección virtual es válida y accesible, la sube a la TLB. La próxima vez que se busque esa traducción, va a ser TLB hit y se ejecutará rápido.

Localidad espacial y temporal

Supongamos un array y un loop que lo recorre linealmente. Si en cada página de la page table tenemos varios elementos del array, el primero al que queramos acceder de la página va a ser un miss (por lo que el TLB va a tener que buscar la traducción) pero luego hará hit con todos los elementos que estén en la misma página, mejorando drásticamente el desempeño en comparación a tener que buscar la traducción de la referencia virtual para cada uno de ellos.



a(0) sería miss pero a(1) y a(2) serían hit, a(3) sería miss pero … (etc.)

Esto se da porque, ante un TLB miss, se guarda **toda la page** en la TLB, y no solo la dirección cuya traducción fue requerida.
Es decir, la TLB (la memoria caché) se beneficia de la **spatial locality** (que los elementos están cerca unos de otros) y, si el programa se vuelve a ejecutar rápidamente (por ej. un loop) se beneficia de **temporal locality** (la rápida re-referenciación a items en memoria en el tiempo) ya que todavía formarán parte de la TLB y volverían a ser hit.
Claramente, ante mayor tamaño de las páginas o de la TLB, mayor porcentaje de TLB hit se genera y mayor se aprovechan estas localidades.

Manejo del TLB miss

En sistemas antiguos CISC, los TLB miss eran manejados por hardware. En sistemas modernos **RISC** son manejados por el software, usando un **software-managed TLB**; en un miss el hardware solo levanta una excepción que pausa el flujo de instrucciones, eleva el privilegio a modo kernel, salta al **trap handler** (un código dentro del SO), busca la traducción en en la page table, y actualiza la TLB con instrucciones privilegiadas, para luego volver de la trap y que el hardware re-ejecuta la ejecución.

El return from trap luego de un TLB miss debe volver a ejecutar la **misma instrucción** que causó el miss para esta vez dar hit, a diferencia del return from trap normal que ejecuta la siguiente instrucción (el PC, program counter register, es diferente en cada caso).

El SO debe tener cuidado de no causar un loop de TLB misses (si, por ej., el trap handler se encuentra en una memoria virtual no cacheada en la TLB), y para esto tiene diversas estrategias que puede usar. Las principales ventajas de que el TLB miss sea manejado por software son su **simplicidad** y **flexibilidad**; se puede implementar cualquier estructura de datos que se necesite sin requerir un cambio en el hardware.

Contenidos de la TLB

Una TLB típica tiene 32, 64 o 128 entradas y es **fully associative**, es decir, el hardware buscará la traducción en paralelo (rapidez constante) en todas las entradas.
La TLB contiene **VPN | PFN | other bits**.

Entre esos otros bits suelen incluirse un **valid bit** que indica si una entrada tiene una traducción válida para esa dirección de memoria virtual o no, **protection bits R/W/X**, que determinan si se puede acceder a una página (read, write, execute), **address space identifier**, **dirty bit** y otros.

Notar que el TLB (PTE) valid bit es diferente al valid bit de la Page Table, el cual marcaba que una page table entry no ha sido asignada (allocated) por el proceso y no debe ser usada por ningún programa.

Context Switches

Las traducciones que hay en la TLB sólo sirven para el proceso en ejecución; al hacer un context switch dichas traducciones no deben ser usadas con el nuevo proceso.

Un enfoque frente a este problema es borrar el contenido de la TLB (**flush**), seteando todos los valid bit en 0. Sin embargo, ante cambios de contexto frecuentes, se genera un alto costo al tener que buscar las traducciones cada vez.

Frente a esto, algunos sistemas usan se apoyan en el hardware al añadir un **address space identifier** (**ASID**); un campo en la TLB que permite identificar a qué proceso pertenecen las traducciones, pudiendo almacenar a la vez las de diferentes procesos.

Política de reemplazos
Las memorias caché son veloces pero pequeñas. Si para insertar una nueva entrada en la TLB es necesario reemplazar una vieja, se debe elegir una política para realizar ese **caché replacement**. La idea siempre es bajar el porcentaje de TLB miss; puede elegirse borrar la que más tiempo lleva sin usarse (**LRU**: least recently used) para tratar de mantener la localidad temporal, o simplemente borrar una **aleatoria**, que no presenta casos borde como LRU (por ej. ante un recorrido en loop de un array que no entre en la TLB).

20: Tablas de paginación más pequeñas

Un problema ya mencionado es que las page tables ocupan mucho espacio en memoria, y cada proceso tiene su page table. La idea es administrar la memoria de manera más eficiente, reduciendo el overhead de mantener una sola gran page table.

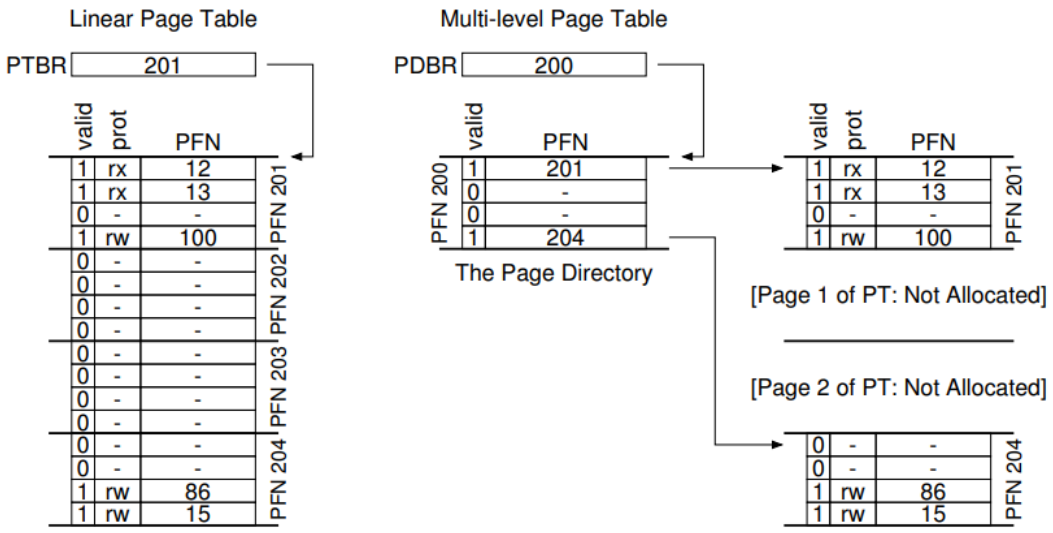
Intentar resolver el problema usando páginas más grandes generaría un gran desperdicio de espacio dentro de las páginas cuando un proceso requiere de poca memoria, llevando a **fragmentación interna**.

Por otro lado, intentar combinar paginación y segmentación para tener una page table por cada segmento del programa (code, heap, stack) en vez de una sola para todo el address space (lo que llevaba a marcar espacios de páginas sin usar como inválidas), es decir usar segmentación externa con paginación en cada segmento, falla ya que da por hecho cierto patrón del uso del address space; si tenemos poco uso en un segmento determinado, se sigue desperdiciando espacio. A su vez, causa **fragmentación externa**; las page table ahora son de tamaño arbitrario, dificultando la tarea de encontrar lugar para almacenarlas.

Page Tables de nivel-múltiple
La solución adoptada. No usa segmentación pero busca solucionar el espacio desperdiciado por páginas inválidas/sin asignar en la page table. Este enfoque convierte la Page Table lineal en algo parecido a un árbol.

La idea es cortar la Page Table en unidades del tamaño de una página, y si una PTE (page table entry) entera está sin usar/inválida, directamente no allocar esa página (a diferencia de las page table lineales, en donde ocuparía memoria)

Para saber qué página es válida (y, de serlo, donde está en memoria) se utiliza una estructura llamada **page directory** (directorio de páginas, una “meta page table”), la cual indica donde está una página de la page table, o si la página entera de la page table no contiene páginas válidas.



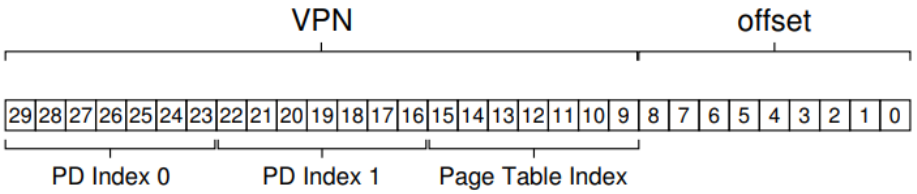
Page tables lineales (izq.) y Page tables multi-nivel (derecha).
Se puede observar que solo la PTE 0 y 3 están en uso.

Esta page table en dos niveles consta de un número de **page directory entries** (**PDE**) (entradas del directorio de páginas).
Cada PDE tiene al menos un **valid bit** y un **page frame number** (PNF), similar a PTE. El significado de este valid bit es que si la PDE es válida, al menos una de las páginas de la page table a la que señala la entry es válida, con que una sea válida el valid bit ya será 1. Además tiene un bit **present** que indica si la page table que le sigue está presente o no.
Cada porción de la page table entra en una página, haciendo fácil para el SO obtener la siguiente página libre cuando necesite hacer crecer una page table (manejo de memoria más fácil).
El **page directory base register** (**PDBR**) es un registro del procesador que apunta a la base del page directory.

Este enfoque permite no tener que almacenar page tables vacías, pero añade un **nivel de indirección** a través del page directory, que señala a partes de la page table, lo que permite poner las page tables pages donde queramos en memoria física. El costo de este nivel extra es que es ante un TLB miss ahora se deben hacer **dos cargas** antes de obtener la traducción; primero al page directory y luego la page table page. Esto es un ejemplo de trade off, ganamos memoria pero perdemos mucho más tiempo en un TLB miss.

Más de dos niveles
Cuando los bits de indexación del Page Directory son muchos, no es posible que cada parte de la multi level page table entre en una página. Para solucionarlo, se puede añadir otro nivel en la forma de otro page directory por encima del primero, partiendo los bits de indexación (PAE). Esto permite evitar o un page directory grande o una page table grande.
El primer page directory chequea validez, luego la entrada válida señala al segundo page directory, en el cual si la entrada de este es válida finalmente se llega a la traducción de la VPN a la PFN.

Aumentar los niveles aumenta, nuevamente, el costo de la búsqueda de la traducción en caso de un TLB miss.



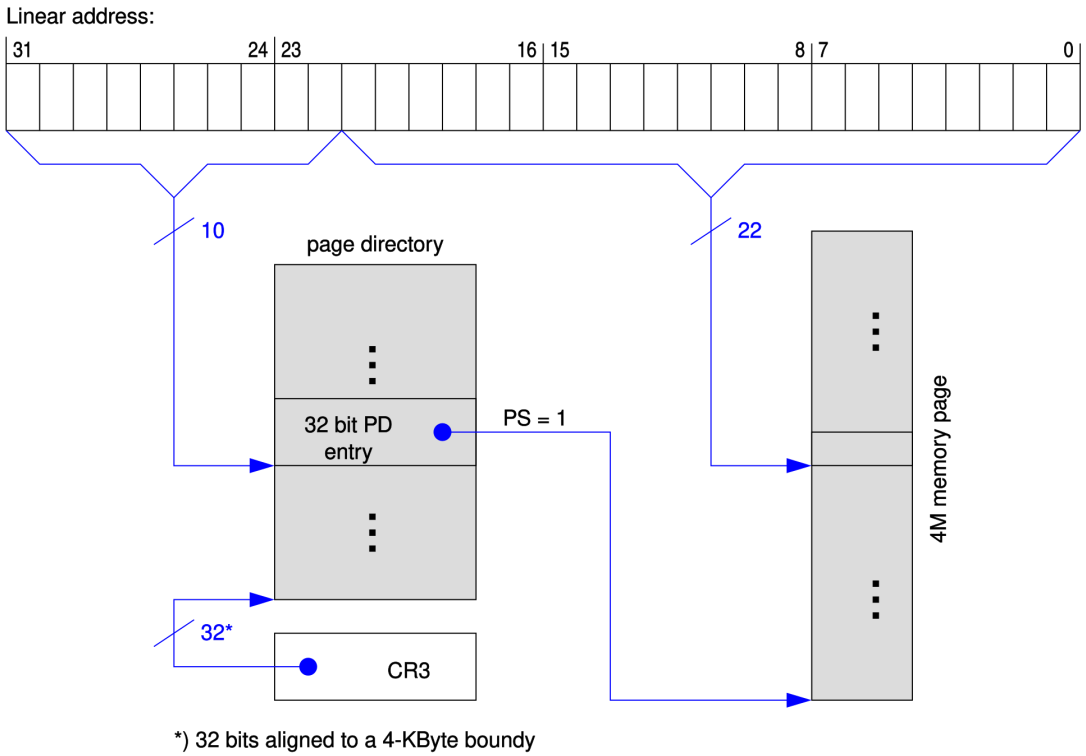
Ejemplo para páginas de 512 bytes y PTEs de 4 bytes.

Swapear Page tables al disco

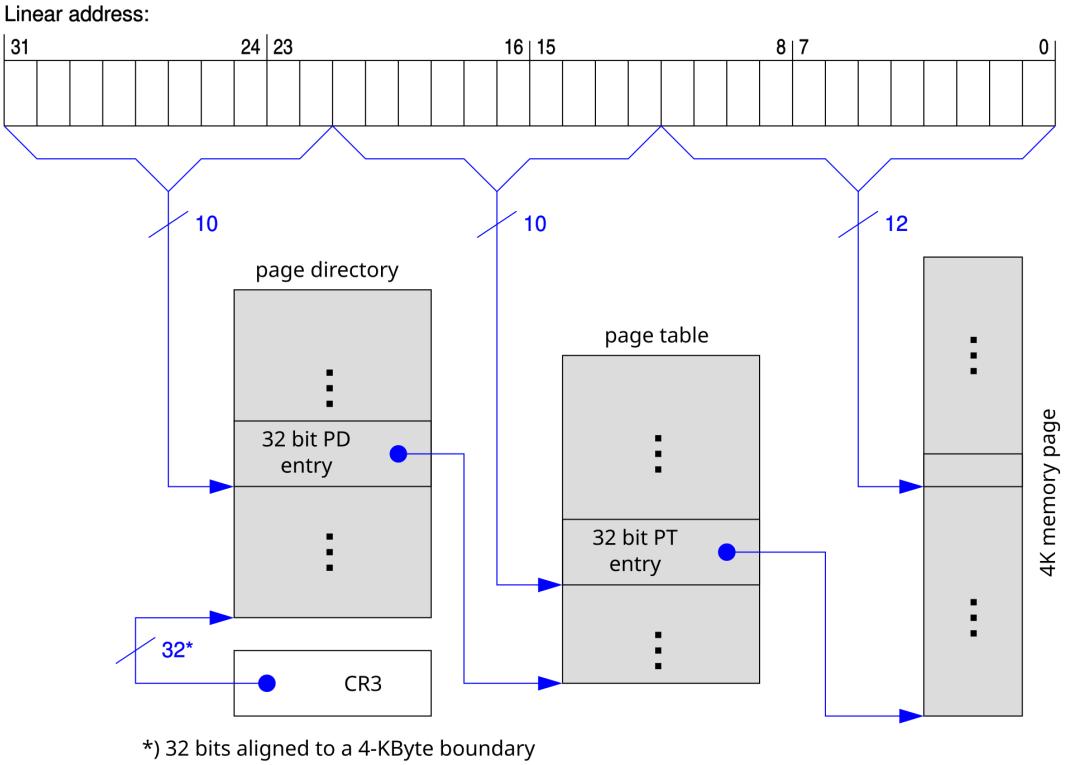
Hasta ahora suponíamos que las page table estaban en memoria del kernel, pero incluso con todos estos “trucos”, las page tables pueden ser demasiado grandes para entrar en memoria todas al mismo tiempo. Cuando esto pasa, algunas se colocan en **memoria virtual del kernel**, permitiendo **pasar** (**swapear**) algunas de estas page tables al disco cuando hay poca memoria. Claramente, este swap introduce una penalización de tiempo grande, ya que si un proceso intenta leer memoria virtual de su address space y esta no está en la RAM, se genera un page fault y la misma debe ser traída desde el disco.

Ejemplos

Recordar que son estructuras de datos que, a partir de una dirección virtual de 32 bits (arriba), permiten generar una dirección física de 32 bits (64 si se usa PAE).



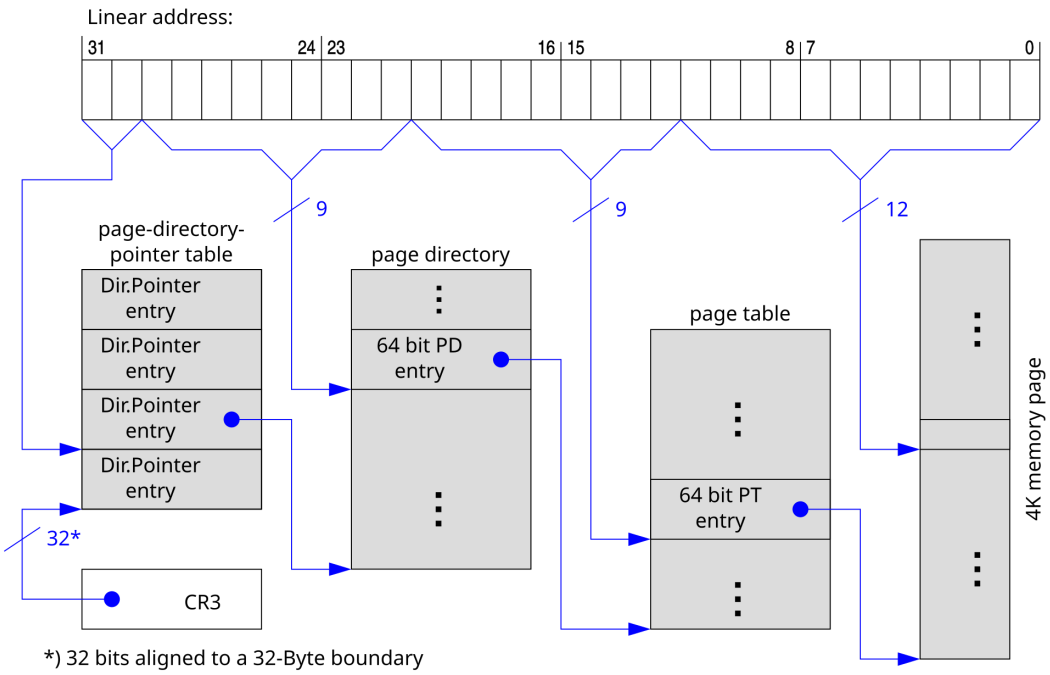
Page table 10/22 (de un nivel) para páginas de 4MiB.



Page table 10/10/12 (de dos niveles) para páginas de 4K, de un i386 (x86).

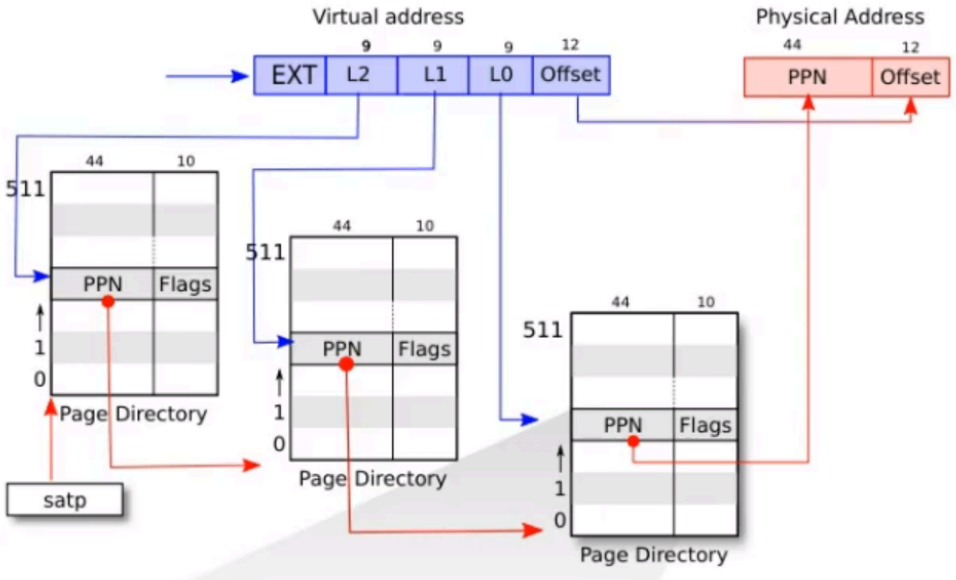
- **CR3**: registro especial de 32 bits solo modificable en modo kernel, que apunta a la base de la tabla Page Directory. Los últimos 12 bits (3 hexa) del CR3 siempre son 0 (en hexa) porque siempre debe estar alineado a saltos de 4K.
- **Page Directory**: tabla que tiene 1024 entradas, cada una de 32 bits (4 bytes). 1024 x 32 bits = 4K lo cual es el tamaño de la tabla. Para direccionar esas 1024 entradas son necesarios los 10 bits más significativos de la dirección virtual (2^{10} = 1024).
  La entrada seleccionada es un puntero que define la base de la Page Table (su posición 0 ya que crece de forma negativa, es decir, la posición desde la cual se comienza a contar (descontar) para usar el próximo indice)
- **Page Table**: tabla de 1024 entradas de 32 bits, cuya base está dada por el puntero de la page directory, y cuyo índice está dado por los 10 bits que le siguen a los usados por la Page Directory. La entrada seleccionada es un puntero (20 bits) que indica la base de la página física que se va a usar.
- **Memory page** es la página física que se va a leer, cuya base está dada por la Page Table. Tiene 4K entradas, para lo cual son necesarios los 12 bits menos significativos (llamados offset) de la dirección virtual (2^{12} = 4K) para seleccionar la página.

Recordar que una página no puede contener segmentos (heap, stack, code) de distintos tipos para proteger a cada uno de manera adecuada.
En un sistema con páginas lineales puede que una page table corresponda a un proceso. En uno de páginas multinivel hay todo un directorio por cada proceso, el cual puede tener 1 o más tablas.



Page table 9/9/12 (de tres niveles) de un i386 (x86) con PAE (Physical Address Extension).

Physical Address Extension permite direccionar más de 4GB de memoria física (2^{32} bytes, límite del diseño anterior) al pasar las table entries de 32 bits a 64 bits (sumando más espacio en el page frame number de cada una de ellas). Para ello, añade un nivel más en la jerarquía de las page tables. El virtual address space todavía es de hasta 4GB (32 bits).



Traducción de direcciones en la arquitectura RISC-V (9/9/9).

Page Directory ahora con un ancho de 64 bits (al igual que la dirección virtual) y con direcciones físicas de 56 bits.

Notar que al ser 9/9/9/12 usa los bits menos significativos de los 64 ahora disponibles (los restantes son bits EXT de extensión).

26: Introducción a la concurrencia

Se introduce una nueva abstracción para un proceso en ejecución: el **hilo** (**thread**). Un programa de **hilos-múltiples** (**multi-threaded**) tiene más de un punto de ejecución; cada hilo es un sub-proceso en sí mismo que avanza de manera asíncrona, pero compartiendo todos un mismo address space (comparten heap y code).

Cada hilo mantiene su propio PC, registros y memoria stack (**thread local storage**), por lo que ante un cambio de contexto estos deben ser guardados y restaurados. Los diferentes bloques stack se almacenan en el mismo address space del proceso. Por ello, si el context switch es entre dos hilos del mismo proceso, se usa un **thread control block** (**TCB**) en lugar de un PCB, manteniendo address space y por tanto page table (las traducciones de la TLB todavía podrían servir), lo cual mejora el desempeño.

Ventajas de usar hilos

- **Paralelismo**; se puede dividir al trabajo en hilos, haciéndolo más rápido y eficiente (en caso de que el overhead no supere la ganancia). Convertir un programa **single-thread** (de un hilo) **multi-thread** es llamado **paralelización**.

- Evitar **bloquear** un programa en **espera** de un **I/O** (lento); mientras un hilo espera, el CPU hace un **overlap** (superpone) de tareas y puede ejecutar otra del mismo programa.

  A diferencia de hacer **multiprogramming** (dividir una tarea en muchos procesos), al compartir address space es más fácil compartir información.

Scheduling y Race conditions

La ejecución de los procesos es administrada por el scheduler del SO, por lo que no se puede asumir que hilo se ejecutará en cada momento (incluso si uno es creado antes que otro); una vez llamado, cada uno se ejecuta de forma independiente a su caller hasta retornar. Esto aumenta la complejidad general del funcionamiento del sistema, por lo que es preferible intentar minimizar las interacciones entre ellos.

Que diferentes hilos puedan compartir información puede dar lugar a errores. Por ejemplo, en el caso de dos hilos incrementando un contador común; el primer hilo puede modificado la variable, pero si antes de poder guardar el resultado un context switch guarda el estado actual del proceso y corre el hilo 2 (el cual modifica la variable y guarda en memori el nuevo valor) cuando vuelva a correr el hilo 1 este realizara la instrucción que anteriormente no pudo, guardando el valor del contador que él tenía, pisando el contenido y haciendo que el resultado sea diferente al esperado.

Esto es llamado una **race condition** (o más bien **data race**, condición de carrera) y genera resultados **indeterministas**.

Cuando más de un hilo ejecuta una misma parte del código que contiene un recurso compartido, (por ej. una variable o estructura de datos) y se genera una race condition, esa parte es llamada **sección crítica**. Como esas zonas no deberían ser ejecutadas al mismo tiempo por más de un hilo, se busca asegurar una **exclusión mutua** que garantice el acceso de solo un hilo a la vez.

Atomicidad

Una solución que garantiza la exclusión mutua es tener instrucciones que, en un solo paso, se ejecuten por completo y remuevan así la posibilidad de un interrupt intermedio. Es decir, que sean **atómicas**; que o bien la instrucción se ejecute hasta completarse, o bien no se ejecute.

En general esto no es posible, por lo que, usando soporte del hardware y del SO, se construyen diferentes **primitivas de sincronización**; código multi-thread en el que cada hilo accede a las secciones críticas de forma sincronizada y controlada, evitando las condiciones de carrera y asegurando la exclusión mutua.

27: API de los hilos

Para crear y controlar threads se utiliza la POSIX library:

pthread_t: tipo thread.
pthread_attr_t: tipo de los atributos de un hilo.
pthread_attr_init: inicializa un objeto tipo pthread_attr_t con los atributos correspondientes.
Pthread_create: crea un hilo.
Pthread_join: ejecuta el hilo hasta que se complete.
pthread_mutex_destroy

Pthread_create toma 4 argumentos: *thread, attr, start_routine y arg:*
**thread* es un puntero a una estructura de tipo pthread_t, y es la estructura con que permite interactuar con el hilo una vez inicializada.
**attr* tiene tipo pthread_attr_t y es usado para especificar atributos del hilo (por ej. la prioridad de scheduling del hilo). Se inicializa anteriormente, usando *pthread_attr_init()*.
start_routine* indica la función que debería ejecutar el hilo (function pointer** en C) y espera algún nombre de función.
**arg* es la lista de argumentos que se le pasa a la función que ejecuta el hilo (se usan void pointers para permitir cualquier tipo de argumento/resultado).

Pthread_join permite esperar a que un hilo termine de ejecutarse antes de continuar la ejecución secuencial del programa. Esta rutina toma dos argumentos: *thread*, y *value_ptr*:
thread tiene tipo pthread_t y especifica el hilo a esperar
***value_ptr* es un puntero al valor de retorno que se espera.

Una vez se ejecuta la creación de un hilo, ya se tiene otra entidad en ejecución independiente de la original pero que usa el mismo address space.
Tanto en Pthread_create como en Pthread_join se puede usar NULL como argumentos.

Al terminar la ejecución de un thread su memoria stack se borra (como en cualquier proceso) por lo que un hilo nunca debe devolver un puntero a un valor en su stack.
Por otro lado, no todos los procesos usan join, ya que se pueden crear hilos “trabajadores” que se usen indefinidamente hasta que finalice la ejecución del programa más general.

Locks

También provistos por la POSIX library, los locks permiten generar **mutual exclusion** en una **sección crítica** mediante sincronicidad, bloqueando el acceso antes de la misma, y liberando el lock al finalizar la ejecución de esta.

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Inicialización de un lock.

Si ningún otro hilo posee el lock cuando pthread_mutex_lock() es llamado, el hilo lo adquiere y entra a la sección crítica. Si otro thread tiene el lock, el que intentó agarrarlo no volverá de la llamada hasta que lo consiga. Solo el hilo que tiene el lock puede llamar al unlock.

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

Ejemplo de código que utiliza un lock para garantizar exclusión mutua.

La función pthread_mutex_destroy() es llamada para destruir el lock cuando ya no se lo usa más.

Variables de condición

Las condition variables son útiles para mantener algún tipo de **comunicación** entre hilos (por ej., esperar que otro hilo haga algo antes de continuar). Para usarlas se debe tener un lock asociado a esta, y se debe tener control de ese lock al momento de llamar la rutina (lock→rutina→unlock).

Hay dos rutinas principales:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

pthread_cond_wait: libera el lock y pone el hilo (quien la llama) a dormir hasta recibir la señal (cond) esperada, momento en el que vuelve a adquirir el lock. Suele ser llamada en un loop.

pthread_cond_signal: se ejecuta cuando un un hilo cambió algo en el programa, y se manda una señal a la condición señalada para despertar a los hilos que esperaban por la misma.

wait asume que el hilo tiene un lock en su poder que debe ser liberado al dormir el proceso para que otros hilos/procesos lo puedan adquirir. El hilo cuando se despierte debe readquirir el lock para poder ejecutarse.
wait debe liberar el lock y dormir el proceso de forma atómica, para evitar race conditions.

28: Locks

Un **lock** es una **primitiva de sincronización** abstracta que cuenta con solo una variable y, debido a la naturaleza secuencial del programa, permite proteger una zona de código. Se declara la variable lock de algún tipo y la misma tiene el estado del lock en cualquier momento dado.
Puede estar **available**/**disponible** (o **unlocked**, **free**), es decir ningún hilo tiene el lock en su posesión, o estar **acquired**/**adquirido** (o **locked**, **held**), lo que significa que un hilo (y solo uno) tiene el lock.
Se pueden guardar otros **datos** en el lock, como qué hilo tiene el lock en su poder (si lo hay), o una lista de los hilos en espera de adquirir el lock. Esta información está oculta al usuario.

La rutina *lock()* trata de adquirir el lock y, si ningún hilo lo tiene en su control (is unlocked), va a hacerlo y entrar a la zona crítica. Este hilo es el llamado **dueño**/**owner** del lock. Si otro hilo llama lock() con la misma variable lock, no va a volver y se va a quedar en un bucle intentando conseguirlo, previniendo así que acceda a la zona crítica.

Los locks no otorgan control sobre el scheduler. Los hilos son entidades creadas por el programador y administradas por el SO; con los locks el usuario se asegura que solo un hilo pueda acceder a una sección de código al mismo tiempo.

Pthread locks
La POSIX library llama **mutex** a los locks, y los usa para proveer **mutual exclusion** entre hilos.

Las rutinas pasan variables a lock() y a unlock() porque pueden usarse diferentes locks para proteger diferentes zonas críticas. Esto aumenta la **concurrencia** ya que en vez de un solo bloque grande que se lockea cada vez que se accede a cualquier zona crítica en su interior (una estrategía de lock de grano grueso/**coarse-grained** locking), se pueden proteger diferentes datos y estructuras de datos con diferentes locks, permitiendo que más hilos accedan al código a la vez (**fine-grained** locking/de grano fino). Estas hacen uso de instrucciones provistas por el hardware para evitar race conditions que permitan que dos hilos obtengan un mismo lock.

Un lock puede implementarse de distintas maneras, tanto con ayuda del hardware como solo mediante software, pero siempre deben cumplir con ser **correctas** (proteger la región crítica) **justas** (evitar la starvation de hilos esperando entrar en una sección) y tener buen **desempeño**. A continuación se presentan algunas implementaciones:

- Desactivar Interrupts

Una solución para asegurar la atomicidad en las secciones críticas es desactivar las interrupciones. Esto es efectivo pero solo funciona como solución para sistemas mono core y además significa confiar en que los programas no abusen de esto, llamando a lock para monopolizar el CPU y haciendo que el SO pierda el control del sistema. Se usa solo de manera interna y en regiones pequeñas y puntuales del sistema operativo.

- Usar solo Loads/Stores

Se puede intentar construir un lock sin desactivar las interrupciones usando una variable flag para indicar si algún hilo tiene un lock en su posesión, y en caso de que así sea, hacer **spin wait** (un bucle “;” chequeando la condición del lock) hasta que el lock sea liberado.

Esto tiene problemas tanto de correctness (exactitud) ya que por concurrencia dos hilos podrían adquirir el lock, y de performance (desempeño) debido al alto coste de hacer spin waiting.

Estos dos intentos fallidos muestran que es necesario el **soporte del hardware** para construir soluciones que garanticen la exclusión mutua:

- Spinlocks con Test-And-Set

El hardware provee soporte para instrucciones **test-and-set** (**atomic exchange**) atómicas.

Test and set devuelve el valor viejo apuntado por *old_per*, y actualiza el valor con *new*. Con esta instrucción se puede construir un **spin lock** (usando test-and-set para chequear la condición del lock, y solo saliendo del spin-wait si el valor viejo es 0).

El que se use spin lock para esperar a que el **owner** libere el lock hace que sea necesario un **preemptive scheduler**. Sin embargo, (y por más que se garantice correctness y performance para sistemas multi core) se pueden generar problemas de **fairness** con hilos en espera que pueden quedar en spin indefinidamente.

- Compare-And-Swap

Otra primitiva soportada por hardware es **compare-and-swap** (**CaS**), o compare-and-exchange. Esta se basa en comparar el valor de la dirección especificada por *ptr* con el esperado y, si lo es, actualizar la memoria señalada por ptr con el nuevo valor. Luego, devuelve el valor original, permitiendo a quien llamó a compare-and-swap saber si se actualizó o no algún valor.

Se puede generar un lock similar al de test-and-set usando compare-and-swap en el chequeo de condición del loop. Compare and swap es una instrucción más poderosa que test and set, especialmente para algoritmos concurrentes que no usan locks (**lock-free synchronization**).

- Load-linked y Store-Conditional

Algunas plataformas **desacoplan** las instrucciones anteriores (complejas) en dos más sencillas que ayudan en la construcción de estructuras de concurrencia (como locks) para secciones críticas, **load-linked** (ll) y **store-conditional** (sc).

Load-linked toma un valor de memoria y lo coloca en un registro. Store-conditional almacena en memoria el contenido de un registro, pero solo tiene éxito si no hubo **intervención** de almacenamiento en esa dirección (un store) luego de haber usado load-link. Si lo logra, devuelve 1, y si no, no actualiza el valor y devuelve 0 (detecta race conditions).

Con lock(), un hilo puede hacer spin wait esperando que flag=0 (que el lock está libre) y cuando esto pasa intentar adquirir el lock a través de store-conditional. Si lo logra, el hilo cambia atómicamente el valor de la flag a 1 y entra en la sección crítica. Puede que más de un hilo realice un load-linked y luego intente un store-conditional, debido a un interrupt y context switch, pero solo un store conditional lograra adquirir el lock, preservando la zona crítica.

- Fetch-And-Add

Otra instrucción provista por el hardware es **Fetch-And-Add**, la cual atómicamente incrementa un valor mientras devuelve el valor viejo a una dirección particular.

Con ella se puede construir un **ticket lock** que usa una variable ticket y turno combinados para construir el lock. Así, cuando un hilo quiere el lock hace un fetch-and-add atómico en el valor del ticket y ese valor se considera el turno (myturn) de dicho hilo. La variable global lock→turn es usada para determinar el turno de que hilo es, y cuando myturn==turn el hilo entra a la sección crítica. El unlock aumenta el valor del turno, y así el siguiente hilo en espera (si lo hay) puede entrar a la sección crítica. Este método asegura que todos los hilos progresen.

Demasiado Spinning
Estos métodos son simples y funcionan, pero pueden ser ineficientes si la cantidad de hilos esperando por un lock es grande, desperdiciando tiempo de procesador en mucho spinning.

Enfoque simple: Just yield (“ceda el paso”)
Consiste en que el hilo haciendo spin ceda el CPU, con la esperanza de que se ejecute el hilo que libere el lock (o la condición) que espera. Asumimos que tenemos yield() provisto por el SO.
Este enfoque puede ser malo en **performance**, ya que el costo del context switch para correr el siguiente hilo es alto (aunque menor que hacer spinning) y no soluciona el problema de starvation.

Enfoque con colas: sleeping en vez de spinning
Se puede ejercer control explícito sobre qué hilo es el siguiente para adquirir el lock después de que el actual lo libere, usando una cola con el **orden** de los hilos esperan el lock en modo sleep.
Debe ser hecho con cuidado, ya que un mal timing de context switch puede causar un **wakeup/waiting race** y que un hilo termine durmiendo para siempre.

Enfoque híbrido: Two-phase Locks
En la primera fase el lock() hace **spin** por un pequeño periodo de tiempo con la esperanza de obtener el lock (por si justo estaba por ser liberado, si la sección crítica era pequeña). Si no lo consigue rápido, entra en segunda fase y el hilo es puesto a **dormir** hasta que se libere el lock.

30: Variables de condición

El sistema lock-unlock no es *synchronization complete*, por lo que para construir programas concurrentes es necesario utilizar otras **primitivas de sincronización**. En particular, es necesario para cuando un hilo desea chequear una **condition** antes de continuar su ejecución (si un hilo “hijo” terminó su ejecución por ej.) sin usar una variable global (chequeos spin = ineficiente).

Definición y rutinas
Para esperar que una condición se vuelva true, un hilo puede usar una **condition variable**; una **lista** explícita en la que hilos en algún estado de ejecución (**condition**) que no es el deseado (**esperando** a la **condición**, chequeando si puede obtener el lock) se insertan. Luego, algún otro hilo, puede despertar uno o varios de esos hilos en espera cuando cambie dicho estado, y así permitirles que continúen (al darle una **señal**/**signaling** a la condición) su ejecución.

Se utilizan las rutinas wait() y signal() definidas en el capítulo 27.
Es recomendable, para evitar errores, mantener el lock mientras se llama a signal o wait.

El problema del productor-consumidor (Bounded Buffer)

El problema de sincronización **productor/consumidor** se da cuando tenemos uno o más *hilos productores* y uno o más *hilos consumidores.* Los productores generan data items y los colocan en un buffer, mientras que los consumidores obtienen y consumen dichos elementos.
Como este buffer es un recurso compartido, se necesita alguna clase de sincronización para acceder al mismo y evitar una condición de carrera (race condition).

```
1  int buffer;
2  int count = 0; // initially, empty
3
4  void put(int value) {
5      assert(count == 0);
6      count = 1;
7      buffer = value;
8  }
9
10 int get() {
11     assert(count == 1);
12     count = 0;
13     return buffer;
14 }
```

Ejemplo de buffer int, en el que el productor solo coloca un elemento si el buffer está vacío, y el consumidor solo consume un elemento si el contador indica un elemento.

Primer intento (condition variable)

Tanto put() como get() tienen secciones críticas que debemos proteger, pero no es suficiente solo con locks y hacen falta condition variables; la solución actual no funciona para más de dos hilos (un consumidor y un productor) ya que se puede dar el caso en el que un consumidor intente consumir y no haya buffers disponibles de los que consumir.
En ese caso se dormiría, un productor llenaría el buffer y daría la señal de que está lleno, el consumidor se podría despertar y tratar de adquirir el lock, pero antes de esto otro consumidor podría obtenerlo y consumir el buffer para luego soltar el lock. Si ese lock lo obtiene el primer consumidor e intenta consumir del buffer, al estar este vacío se levantaría una excepción.

No hay garantía de que desde cuando un hilo se despierte hasta que adquiera el lock y se ejecute la condición por la que se despertó se mantenga. Esta interpretación de señales es conocida como **Mesa semantics**.

Segundo intento (while, not if)

Si cambiamos el cómo se chequea la condición y en vez de un *if* ponemos un *while,* cuando un hilo consumidor ejecute la rutina va a rechequear siempre la condición (que lo despertó en un comienzo), y si no está en true se duerme. Gracias a Mesa Semantics la regla que siempre debemos recordar es **siempre usar *while* loops**. Aunque no sea totalmente necesario es más seguro. Pero todavía hay un error:
Si luego de consumir un buffer y antes de dormir un consumidor despierta a otro consumidor en vez de a un productor, podría suceder que todos los hilos duerman indefinidamente. Por ello, las señales deben ser directas y un consumidor no debería poder despertar a otros consumidores (solo a productores) y viceversa.

Solución para un único buffer de productor-consumidor

La solución es usar dos variables de condición para dar la señal de que tipo de hilo debería despertar cuando el estado del sistema cambia. Los consumidores esperan en una **condición empty** (“hay al menos un lugar vacío”) y dan la **señal** de **fill** (“hay al menos un lugar ocupado”); los productores esperan en la señal **fill** y dan la señal **empty**. Así, un consumidor nunca puede despertar a otro consumidor y viceversa.

La solución correcta del productor-consumidor

Se puede crear una solución más general modificando las rutinas y añadiendo más buffers, consumidores y productores, aumentando así la concurrencia y optimizando el desempeño, permitiendo que un productor duerma solo si todos los buffers están llenos y que un consumidor duerma solo si todos los buffers están vacíos.

Condiciones de cobertura

Otro problema de concurrencia ocurre cuando un hilo intenta pedir/asignar memoria, y si no hay memoria disponible se duerme. Luego, si un hilo libera memoria da una señal de que hay memoria, pero debe decidir a qué hilo (o hilos) despertar.

Una solución que evita problemas es despertarlos a todos, con el costo de despertar hilos innecesariamente. Esto se llama **covering condition** y asegura que cubre los casos que deben ser cubiertos, sacrificando la eficiencia.

31: Semáforos

Un semáforo es una primitiva de sincronización, un objeto con un valor entero que puede usarse como lock y como condition variable, y puede ser manipulado con dos rutinas; *sem_wait()* y *sem_post()*. El valor inicial del semáforo determina su comportamiento, por lo que debe ser inicializado antes de llamar otra rutina que interactúe con él.

```
1  #include <semaphore.h>
2  sem_t s;
3  sem_init(&s, 0, 1);
```

Ejemplo de inicialización.
El valor inicial es 1, ya que 0 indicaria que está siendo compartido por hilos del mismo proceso.

sem_wait() puede volver inmediatamente (si el valor del sem era 1 o mayor) o causar que el hilo se duerma. Con muchos hilos llamando a wait, se hará una cola en espera a ser despertados.
sem_post() no espera por ninguna condición; solo incrementa el valor del sem y, si hay algún hilo esperando ser despertado, despierta uno de ellos.
Cuando el valor del semáforo es negativo, su valor (absoluto) es igual al número de hilos esperando despertar (este valor suele estar escondido al usuario).

Semáforos binarios (uso como locks)

Los semáforos pueden ser usados como un **lock** si se rodea la sección crítica con un par de *sem_wait()*/*sem_post()*. Como los locks solo tienen dos estados (held or not held), el uso de un semáforo como lock es llamado **binary semaphore**.

Para que funcione correctamente, el valor inicial del sem debe ser 1; el primer hilo que lo obtenga dejará el valor en 0 y entrará a la zona crítica, el siguiente hilo dejará el sem en -1 y será suspendido/dormido, etc. Al terminar el primer hilo, dejará el sem en 0 y despertará al siguiente hilo (que quedará en ready, esperando al scheduler).

Semáforos para determinar orden (uso como condition variables)

Los semáforos también son útiles para ordenar eventos en un programa concurrente. En este patrón de uso suele haber un hilo *esperando* que algo pase y otro *dando una señal* de que paso y despertando al primero, lo que da un orden (similar a usar **condition variables**).

Esto se puede usar, por ejemplo, si un hilo está esperando que una lista este **no-vacía** para borrar algo de ella, o para que un *parent* espere a un *child*; el parent debe llamar a sem_wait() y dormir (valor inicial del sem==0), y el hijo debe llamar a sem_post() al finalizar (en caso de que el hijo corra antes de que el padre llame a wait el sem pasa de 0 a 1 llama el padre y pasa de 1 a 0, sin necesidad de dormir).

El problema de Productor/Consumidor (Bounded Buffer)

Si se usan solo dos semáforos y un buffer, se pueden usar los semáforos para comunicar el estado del buffer. Esto funciona para cualquier cantidad de hilos, pero solo si el valor máximo del buffer es 1. Si se aumenta el tamaño, se genera una race condition.

El llenar un buffer e incrementar el índice del mismo es una sección crítica y por ende debe ser protegida. Usar para esto un binary semaphore lock genera un deadlock; si un consumidor obtiene el lock y el buffer está vacío, se bloquea. Luego, el productor podría llenar el buffer y despertar al consumidor, pero este no podría adquirir el lock ya que el mismo quedó con el consumidor.

Una solución: añadiendo exclusión mutua

Se busca reducir el alcance del lock; se los pone exactamente alrededor de la sección crítica:

```
1  void *producer(void *arg) {
2      int i;
3      for (i = 0; i < loops; i++) {
4          sem_wait(&empty);       // Line P1
5          sem_wait(&mutex);       // Line P1.5 (MUTEX HERE)
6          put(i);                 // Line P2
7          sem_post(&mutex);       // Line P2.5 (AND HERE)
8          sem_post(&full);        // Line P3
9      }
10 }
11
12 void *consumer(void *arg) {
13     int i;
14     for (i = 0; i < loops; i++) {
15         sem_wait(&full);        // Line C1
16         sem_wait(&mutex);       // Line C1.5 (MUTEX HERE)
17         int tmp = get();        // Line C2
18         sem_post(&mutex);       // Line C2.5 (AND HERE)
19         sem_post(&empty);       // Line C3
20         printf("%d\n", tmp);
21     }
22 }
```

Lock de lector-escritor

Otro problema es el del lector-escritor, que requiere un lock más flexible que permita que accesos de diferentes estructuras de datos puedan pedir diferentes tipos de locking. Por ejemplo, operaciones en listas, con rutinas como insertar o buscar; mientras que insertar cambia el estado de la lista y por tanto debe ser protegida la sección crítica, buscar solo lee la estructura de datos sin realizar cambios, por lo que podrían hacerse tantas búsquedas concurrentes como se necesite.

Para dar soporte a este tipo de operaciones se usa un **reader-writer lock** (lock de lector/escritor): si se intenta insertar datos se adquiere el lock convencional de escritura, y cuando se desea buscar datos se adquiere el **lock** de **lectura**, el cual aumenta la variable *lectores* que sigue la pista de cuantos hilos están leyendo la estructura y el **primer lector** también adquiere el **lock** de **escritura**, para asegurar que no se modifique la estructura mientras se la lee.
Para escribir los escritores deben esperar que todos los lectores terminen. Esto puede resultar injusto para los hilos escritores, sumado a que los lectores pueden monopolizar su lock (writers starving) y que la implementación agrega un overhead al sistema.

```
1  typedef struct _rwlock_t {
2    sem_t lock;      // binary semaphore (basic lock)
3    sem_t writelock; // allow ONE writer/MANY readers
4    int   readers;   // #readers in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8    rw->readers = 0;
9    sem_init(&rw->lock, 0, 1);
10   sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14   sem_wait(&rw->lock);
15   rw->readers++;
16   if (rw->readers == 1) // first reader gets writelock
17     sem_wait(&rw->writelock);
18   sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22   sem_wait(&rw->lock);
23   rw->readers--;
24   if (rw->readers == 0) // last reader lets it go
25     sem_post(&rw->writelock);
26   sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30   sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34   sem_post(&rw->writelock);
35 }
```

Implementación de un lock de tipo lector-escritor.

Estrangulación de hilos

Para prevenir que demasiados hilos hagan algo a la vez y atasquen el sistema, se puede poner un límite a la cantidad de hilos que concurrentemente acceden a una sección crítica, y usar un semáforo para organizarlos. Este enfoque se llama **throttling** (estrangulación) y es una forma de **control de admisión**.

Implementando semáforos

Para utilizar semáforos con locks y variables de condición, se utiliza la siguiente implementación:

```c
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

Esta forma de programar semáforos más sencilla y diferente a la de Dijkstra (la que venía usando) ya que el valor del mismo no puede bajar de 0, por lo que no refleja (con su valor negativo) la cantidad de hilos durmiendo esperando para poder adquirir el lock. Es usada, por ej., por Linux.

32: Problemas comunes de concurrencia

Non-Deadlock bugs

Atomicity-Violation Bugs:

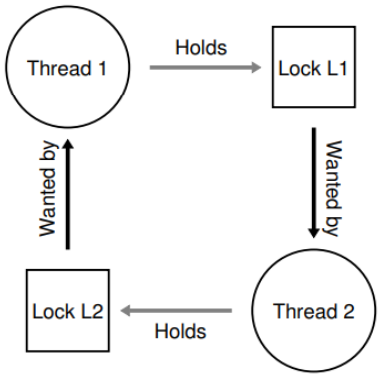
La serializabilidad deseada entre accesos a memoria múltiples es violada (se espera una ejecución atómica pero no es así durante la ejecución). Normalmente (pero no siempre) la solución es directa; poner locks alrededor de la zona a proteger.

Order-Violation Bugs:

El orden deseado entre dos (o más) accesos a memoria no es respetado. La solución usual es el uso de variables de condición, pero el uso de semáforos también puede servir.

Deadlock Bugs

Ocurre cuando hay una dependencia de locks entre hilos. Por ejemplo, si un hilo T1 tiene el Lock L1 y está esperando por L2, pero el hilo T2 tiene en su poder L2, y está esperando por L1.



***Ciclo** indicativo de un deadlock.*

En un caso simple con asegurar el orden de adquisición de los locks bastaría para evitar el deadlock, pero en código más complejo es necesario construir un sistema de locking que evite el deadlock potencial en las dependencias circulares que ocurren naturalmente en el código.

Otra razón por la que ocurren deadlocks es debido a la **encapsulación** de las implementaciones y las construcciones en forma modular, las cuales pueden interaccionar con los locks.

Condiciones para un Deadlock

Se deben cumplir todas las cuatro condiciones para que ocurra un deadlock:

- **Mutual exclusion:** Hilos reclaman control exclusivo sobre recursos que necesitan.
- **Hold-and-wait:** Hilos mantienen recursos asignados a ellos (por ej: lock) mientras esperan por recursos adicionales (otro lock).
- **No preemption:** Los recursos no pueden ser removidos “a la fuerza” de los hilos que los tienen.
- **Circular wait:** Existe una cadena circular de hilos de forma que cada hilo mantiene control de uno o más recursos que están siendo solicitados por el siguiente hilo en la cadena.

Prevenir Circular Wait

Una forma de prevenirlo es escribir código proveyendo un **orden total** en la adquisición de los locks. En sistemas complejos con muchos locks esto es complicado y se usa un **orden parcial** para estructurar la adquisición.

Tanto orden total como parcial requieren un cuidadoso diseño de estrategias de locking. Al ser solo una convención, un programador que ignore el protocolo puede causar deadlocks.

Prevenir Hold-and-wait

Puede ser evitado adquiriendo atómicamente todos los locks a la vez. Esto requiere un **lock global** de prevención que debe ser adquirido antes que los demás, lo cual genera algunos problemas; se necesita saber con anterioridad qué locks requiere el hilo para obtenerlos de **antemano**, lo cual se complica con la encapsulación. Esto disminuye drásticamente la concurrencia ya que los locks son controlados antes, en vez de serlo solo cuando son necesarios para el hilo, disminuyendo así también el rendimiento.

Prevenir No Preemption

Generalmente los lock se encuentran en estado held (alguien las tiene) hasta la llamada de unlock(), por lo que los hilos se mantienen esperando. Algunas librerías proveen rutinas para que tratan de obtener el lock (trylock) y devuelven un código de error si está en control de otro hilo.

Esto genera un nuevo problema; es posible que dos hilos intenten esto y fallen (ambos) en conseguir el lock. Si esto sucede continuamente, no se produce un deadlock pero el programa no progresa; un **livelock**. Una solución es poner un retraso (generalmente aleatorio) antes de reintentar conseguir el lock, disminuyendo las posibilidades de competencia entre hilos.
Este enfoque no añade la capacidad de retomar un lock por la fuerza, pero permite al desarrollador tratar de obtener el lock.

Prevenir Mutual Exclusion

La técnica es evitar la necesidad de exclusión mutua. Para ello se pueden construir estructuras de datos que no requieran locks explícitos (**lock-free**) usando instrucciones de hardware.

Evitar deadlocks a través del Scheduler

En vez de prevenir deadlocks, en algunos casos **evitarlos** es preferible. Requiere conocimiento global de qué locks pueden ser solicitados por los hilos durante su ejecución, para que el **scheduler** los ejecute de forma tal que se garantice que no ocurra un deadlock. Este enfoque disminuye la concurrencia, por lo que no es muy usado.

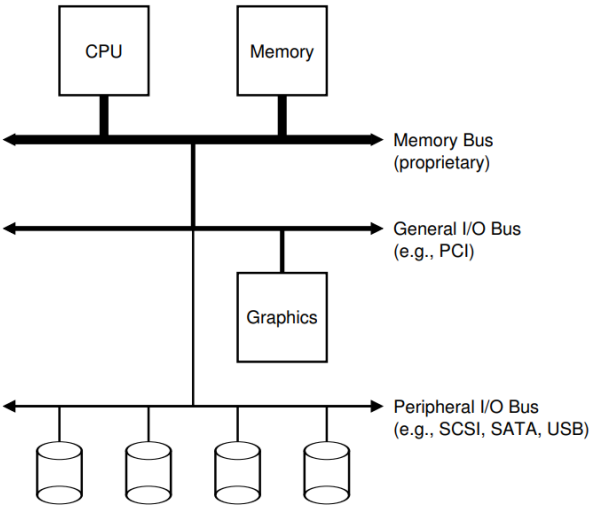
Detectar y Recuperarse

Ante la ocurrencia de un deadlock, emplear un sistema de detección y recuperación. Un detector corre periódicamente, construyendo un gráfico de los recursos y chequeando por ciclos. Si hay un deadlock se elimina uno de los procesos o, en casos graves, se reinicia todo el sistema.

36: Dispositivos I/O

Los dispositivos **input/output (I/O) device** son críticos en los sistemas de computación; un programa sin input siempre produciría el mismo resultado, y no tiene sentido correr un programa sin output ya que nunca devuelve nada, por lo cual es necesario integrarlos en las computadoras.

Arquitectura del sistema



Jerarquía de conexiones; sistema de árboles.
Las interfaces (conectores) usadas por los dispositivos son estándar para todos los fabricantes y deben corresponderse en compatibilidad física, eléctrica y lógica (protocolo de comunicación).

Los CPU están conectados a la memoria principal por un **memory bus** (cable de memoria). Algunos dispositivos I/O de alta performance (por ej. una gráfica) usan el mismo sistema y se conectan directamente por un **I/O bus** (suele ser **PCI**). En una capa más baja, un **bus periférico** conecta al sistema otros dispositivos de I/O más lentos, como discos, mouse, teclado.

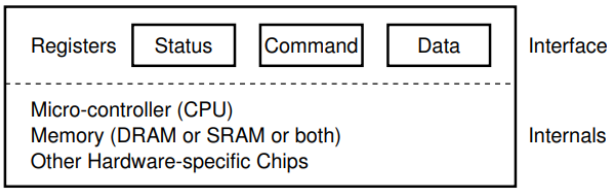
Esta jerarquía se utiliza ya que el bus (cable) debe ser lo más corto posible para alcanzar la máxima **velocidad**, por lo cual la memoria de alto desempeño no tiene mucho espacio para la conexión de dispositivos. Los dispositivos de menor desempeño se alejan del CPU en un bus periférico, permitiendo poner muchos de ellos a una menor rapidez.

Dispositivo canónico

En el siguiente ejemplo de dispositivo canónico (no real) se distinguen los dos componentes esenciales del hardware:

- La **interfaz** que el hardware le presenta al resto del sistema y permite al software controlar su operación.
- La **estructura interna**, parte responsable de implementar la abstracción que el dispositivo presenta al sistema a través de la interface.

Estos dispositivos *son computadoras* en sí mismas (poseen CPU, registros, RAM, ROM, protocolo de comunicación) que corren un software interno llamado firmware. Así, una computadora (la principal, en el uso común de la palabra) es en realidad una *federación* de computadoras.



Ejemplo de dispositivo de hardware canónico y sus partes principales.

Los dispositivos también pueden diferenciarse por cómo comparten datos entre sí; los de **bloque** permiten que se pueda volver a leer información que ya se pasó, y de **carácter** no.

Protocolo

El dispositivo de ejemplo está compuesto por 3 registros, a través de los cuales el SO lo controla:

- **Status** register, el cual muestra el estado del dispositivo
- **Command** register, el cual le indica al dispositivo que realice cierta tarea.
- **Data** register, el cual envía o recibe datos.

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

Interacción típica del SO con un dispositivo.

El protocolo de la imagen anterior consta de 4 pasos:

1) El SO espera hasta que el dispositivo esté listo para recibir un comando (chequeando repetidamente el registro de **status**). Esto es llamado "**polling** the device".
2) El SO envía información al **data** register. Para algunos dispositivos (por ej. un disco) se realizan repetidas escrituras hasta que se transfiera un disk block (un bloque de datos entero). Cuando el CPU principal está involucrado en el movimiento de los datos este paso es llamado **programmed I/O (PIO)**.
3) El SO escribe un comando en el **command** register, haciendo saber al dispositivo que el paso anterior se ha completado y debe empezar a ejecutar el comando. El SO espera a que se complete haciendo **polling** en loop (puede ser exitoso o dar un código de error).

Disminuir el CPU overhead con interrupts

El protocolo anterior resulta ineficiente ya que esperar haciendo polling de un dispositivo lento desperdicia demasiado CPU time. Esto puede evitarse, por ejemplo, cambiando en un context switch al proceso que espera por otro que pueda aprovechar el CPU. Luego, cuando el dispositivo termine su operación, puede enviar un hardware interrupt, causando un **interrupt handler** que despierte al proceso que hizo I/O.
De esa forma, los interrupts permiten hacer **overlap**; superponer procesos para que cuando uno se bloquee esperando un I/O, otro pueda ejecutarse en ese tiempo.

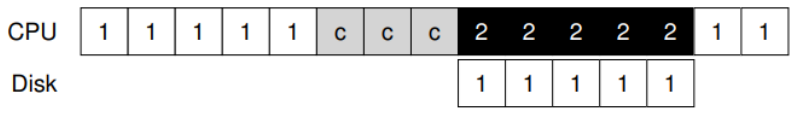
Este enfoque no resulta eficiente para dispositivos rápidos, los cuales se ven ralentizados por los cambios de contexto. Tampoco para networks, donde muchos paquetes pueden generar cada uno un interrupt y llevar a un **livelock** donde el SO no pueda ejecutar ningún user program.

Por ello, si el dispositivo es rápido se hace **polling**, y si es lento se usan **interrupts**. Si no se conoce la velocidad del mismo (o esta puede variar) se usa un enfoque **híbrido** de **dos-fases**, haciendo polling por un corto periodo y lanzando una interrupción si el dispositivo no terminó.

Otro optimización basada en interrupts es el **coalescing**, en la cual un dispositivo que requiere un interrupt espera un poco antes de entregarlo al CPU. Mientras tanto, otros pedidos de I/O podrían terminar, pudiendo juntar múltiples interrupts en uno solo, disminuyendo así el overhead.

Movimiento de datos más eficiente con DMA

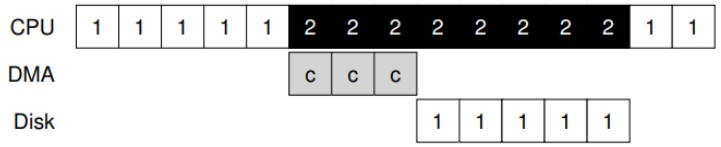
Cuando ocurre un PIO (I/O con el CPU involucrado) encargado de transferir un bloque de datos a un dispositivo, el CPU se sobrecarga con una tarea trivial, ya que enviar datos es lento.



Tiempos de CPU en ejecución y en el copiado de memoria principal hacia el dispositivo (c).

La solución a ello es usar un **Direct Memory Access (DMA)**; un dispositivo específico encargado de hacer **transferencias** de datos entre otros dispositivos y la memoria principal sin intervención del CPU. Para usarlo, el SO programa al DMA informandole dónde se encuentran los datos en memoria, cuántos debe copiar, y a que dispositivo debe mandarlos. Luego, el CPU queda libre y puede realizar otra tarea.

Cuando el DMA termina, hace un interrupt que le da a conocer al SO que la transferencia ya terminó, pudiendo el proceso que realizó el I/O retomar su ejecución.



Tiempo de copiado de memoria al disco desacoplado al de CPU, gracias al uso de un DMA.

Métodos de interacción con dispositivos

Para la comunicación entre el SO y los dispositivos se pueden utilizar dos métodos diferentes:

- Tener **instrucciones de I/O** explícitas, las cuales especifican una manera en la que el SO puede mandar datos a los registros de un dispositivo específico, permitiendo la construcción de protocolos como el ya visto. Dichas instrucciones son privilegiadas y el SO es la única entidad con permiso para manejar los dispositivos.
- Usar una **memory-mapped I/O** (memoria mapeada) con la cual el hardware hace que los registros de los dispositivos se vean como direcciones de memoria, las cuales pueden ser accedidas por SO mediante un load (to read) o un store (to write) en la dirección. Es el hardware quien direcciona dicha acción al dispositivo.

Drivers de dispositivos

Para que los dispositivos, cada uno con su interfaz específica, puedan encajar en los distintos SO generales, se utiliza una **abstracción** en el nivel más bajo llamada el **device driver**; la pieza de software en el SO que sabe en detalle cómo funciona el dispositivo. Cualquier forma de interacción con el mismo está encapsulada en su interior. Son necesarios para cualquier dispositivo que se conecte al sistema, los cuales son descubiertos y cargados por el kernel.

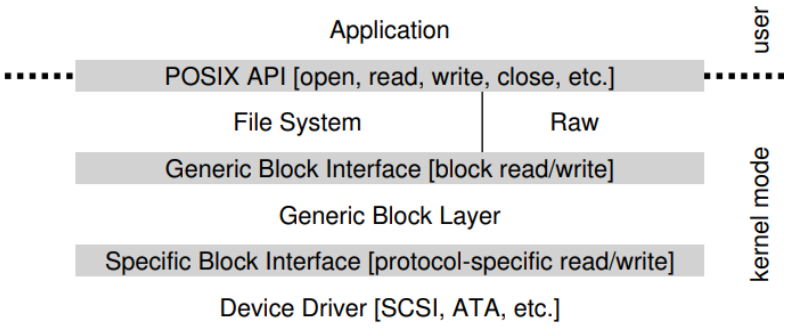


Figure 36.4: **The File System Stack**

Stack del sistema de dispositivos usado por Linux.

En la imagen anterior se puede ver que a un **sistema** de archivos (o una aplicación por encima de este) le pasa inadvertida la especificación de qué clase de disco se está usando; simplemente realiza un pedido de **block** read/write al bloque genérico, el cual redirecciona al **driver** de dispositivo apropiado, el cual maneja los detalles de ejecutar el pedido específico.

El diagrama también muestra la interfaz **raw** a los dispositivos, la cual permite aplicaciones especiales (como un **file-system checker** o una herramienta de **desfragmentación de discos**).

37: Discos duros

Los drives son la principal forma de persistencia de almacenamiento de datos en los sistemas de computadoras y la mayor parte de los sistemas de archivos se basa en su comportamiento.

La interfaz

El driver de un disco consiste en un número de **sectores** (bloques de 512 bytes) los cuales pueden ser leídos o escritos y están numerados de 0 a *n-1*, para un disco de *n* bloques. Eso es llamado el **address space** del drive. Un disco puede ser visto como un array de sectores.

Las operaciones con múltiples sectores son posibles y muchos file systems pueden leer o escribir 4KB a la vez (o más), pero cuando se escribe en el disco el driver solo asegura la **atomicidad** en la escritura de bloques de 512 bytes. Por ende, si la computadora se apaga solo una parte de la escritura se guarda (esto es llamado **torn write** o **lectura rasgada**).
Se asume que el acceso a dos bloques cercanos entre ellos dentro del driver address space es más rápido que si estuvieran lejos uno del otro, y que acceder a los bloque de forma secuencial, uno detrás de otro, es la forma más rápida de hacerlo.

Geometría

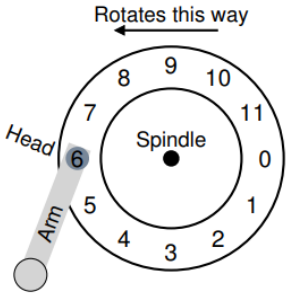
El primer componente de un disco es el **platter** (plato); una superficie circular dura donde los datos son almacenados de forma persistente al inducir cambios magnéticos. Un disco puede tener uno o más platos y cada uno de ellos tiene dos lados llamados **surfaces** (superficies).

Los platos están unidos alrededor del **spindle** (eje) el cual está conectado a un motor que gira los platos a un ritmo constante (fijo y preestablecido) medido en **rotaciones por minuto** (**RPM**). Por ejemplo, un disco moderno con ritmo de 10.000 RPM implica que cada rotación dura 6ms.

Los datos son codificados en sectores circulares concéntricos llamados **track** (pistas). Una sola superficie contiene cientos de pistas, cada una con cientos de sectores los cuales son la unidad mínima de información que puede ser leída o escrita. El conjunto de las pistas con igual radio (misma distancia al eje), de ambas caras de todos los platos, es llamado un **cilindro**.

Para leer y escribir en la superficie se utiliza un **disk head** (cabezal del disco); mecanismo que permite sentir (no toca al plato) los patrones magnéticos o introducir un cambio en ellos. Hay uno por superficie (dos por plato) y está unido a un **disk arm** (brazo del disco), el cual se mueve a través de la superficie para poner el cabezal encima de la pista deseada.
La rapidez de movimiento de estas dos partes es el principal limitante en la velocidad de estos dispositivos. Este cuello de botella hace que sea muy importante colocar los datos de forma lo más continua posible (desfragmentada).

Así, cualquier información en disco se lee/escribe siguiendo un sistema de cuatro “coordenadas”; plato, superficie, pista y sector.

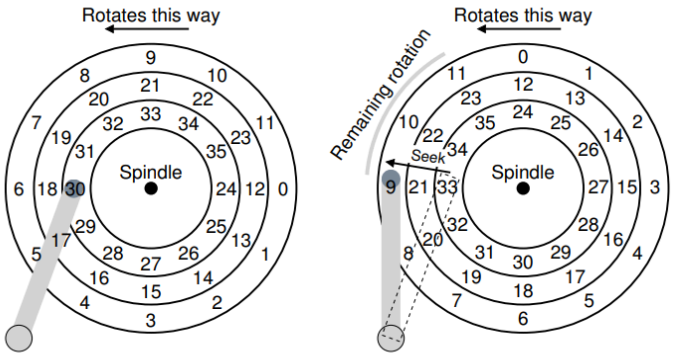


Una pista de un plato, con su correspondiente eje, brazo y cabezal sobre el sector seis.

Latencia de una sola pista: Retraso Rotacional

Para procesar un pedido, estando ya dentro de la pista deseada, el disco debe esperar que el **sector** deseado rote hasta estar bajo el cabezal. Esto es el **rotation delay** (retraso rotacional).
Si el retraso de rotación total del plato es R, el disco deberá esperar en promedio R/2 para que el bloque deseado esté bajo el cabezal (o, en el peor caso posible, esperar R)

Tiempo de búsqueda: Seek Time



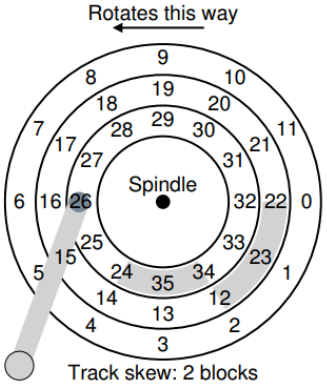
Ejemplo de disco con múltiples pistas con un set de sectores cada una.

Los discos modernos tienen múltiples pistas. Un pedido de lectura (en el ejemplo, al sector 11) implica que, antes de poder esperar a la rotación del sector deseado, el brazo debe mover el **cabezal** a la pista correcta. El tiempo promedio de ese proceso es llamado **seek time** (búsqueda).

Recién una vez el sector se encuentra bajo el cabezal se puede iniciar la fase de transfer de datos. El tiempo de la operación es, entonces, seek time + retraso de rotación.

Otros detalles

Mientras el brazo se mueve para cambiar de pista, el disco sigue rotando. Por ello, muchos drives emplean algún tipo de **track skew** (desviación de la pista), haciendo que el primer sector de una pista no esté alineado con el último de la pista anterior, para asegurar que las lecturas **secuenciales** sean eficientes cuando se cruza ese límite (cambio de pista).



Tack skew de datos secuenciales (sectores 23 y 24) en pistas diferentes.

Las pistas exteriores tienen mayor longitud y por tanto más sectores que los interiores, por lo que son llamadas **multi-zoned** disk drives (multi zonas del disco). A igual cantidad de sectores (como en la imagen) las pistas interiores tendrán mayor densidad de información.

Una parte importante de los discos es el **cache**, algunas veces llamado **track buffer**. Es una pequeña memoria que el disco usa para mantener datos que fueron leídos del disco o escritos.
Por ejemplo, si el drive está leyendo un sector de una pista puede decidir leerlos todos y guardarlos en el cache por si hay un uso posterior en el corto plazo, ahorrando así tiempo. En escritura, el drive puede elegir informar que la escritura se ha completado cuando los datos están en su memoria caché (**write back** caching, rápido pero riesgoso), o cuando se han terminado de escribir en disco (**write through**).

La interfaz física de comunicación del disco es lo que limita la velocidad de transferencia del mismo con la computadora. Los estándares de interfaz son, por ejemplo, SATA, USB, etc.

Cálculo de tiempo de I/O
La performance de un disco, en particular su **I/O time**, puede ser representado como:

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

Esto es así porque leer/escribir un dato consta de esos tres momentos (posicionar el cabezal en la pista indicada (tiempo de **búsqueda**), esperar a que el disco rote para que el bloque se posicione bajo el cabezal (tiempo de **rotación**), y por último leer/escribir efectivamente el sector (tiempo de **transferencia máxima** (sin mover cabezal, no la promedio), dado por el ancho de banda máximo)

El ritmo del input/output ($\mathbf{R_{I/O}}$ o **tasa de transferencia de lectura al azar**) para un tamaño de datos se usa para comparar discos y es el tamaño transferido dividido por el tiempo que tomó:

$$R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}}$$

No confundir con la velocidad de transferencia dada en las tablas de descripción de los discos, la cual hace referencia a la velocidad para datos secuenciales.

Para analizar la diferencia en desempeño, los tipos de cargas de trabajo pueden dividirse en **accesos random** (aleatorio) o **sequential** (secuenciales), y los discos pueden ser separados en los centrados en high **performance** (alto desempeño, altas rotaciones, bajo seek time, transferencias rápidas) o en **capacity** (capacidad de datos).
La mayor diferencia de desempeño entre los discos centrados en performance o capacidad se ve ante cargas de trabajo de accesos aleatorios, pero son “casi” iguales para accesos secuenciales.

Por ejemplo, para calcular cuanto tardaria un acceso random de 4KB read en un disco high performance, usamos los datos del fabricante:

$$T_{seek} = 4\ ms,\ T_{rotation} = 2\ ms,\ T_{transfer} = 30\ microsecs$$

- T_{seek} = 4ms es un promedio, no la búsqueda total (la cual puede ser de más del doble).
- El delay ($T_{rotation}$) se calcula directamente de los RPM; 15.000 RPM son 250 RPS, por que cada rotación son 4 ms, y el disco tendrá un promedio de media rotación de 2 ms.
- El $T_{transfer}$ es el tiempo que toma la operación, dada la tasa de transferencia (velocidad, dato) y el tamaño del archivo. No es para averiguar cuánto tiempo demora en comenzar.

Así, el el $T_{I/O}$ es de 4ms + 2ms + 0.03ms = 6.03ms, por lo que el $R_{I/O}$ para 4KB es de 4KB / 6.03ms = 0,66MB/s.

Planificación en el disco

Frente a varios pedidos de I/O, el SO los administra mediante el **disk scheduler** el cual examina los pedidos y decide cual correr a continuación (orden). A diferencia del scheduler de procesos, el del disco puede hacer una aproximación bastante acertada de cuanto va a tardar un I/O (al calcular seek time y el posible retraso de rotación) por lo que puede elegir ejecutar primero al pedido que menos va a tardar en completarse (seguir el principio de shortest job first **SJF**).

SSFT: Shortest Seek Time First

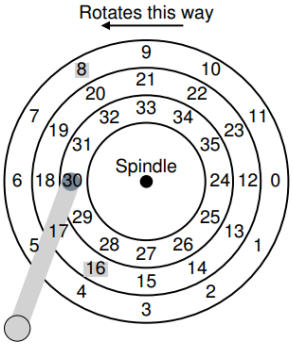
Ordena la lista de request (pedidos) de I/O por pista y elige el request más cercano a completarse en esa pista, y así en cada pista. No es ideal porque la geometría del drive no está disponible para el SO; solo es un array de bloques (no ve su forma circular en el plato del disco).
Esto se puede arreglar implementando **nearest-block-first (NBF)** (el bloque más cercano primero), pero aun así mantiene el problema de la **starvation** para las pistas exteriores cuando hay muchos request en la misma pista interior.

Elevator (SCAN)

Este algoritmo hace que el cabezal se mueva atrás y adelante a través del disco respondiendo a request en orden a través de las pistas. Un solo paso a través del disco es un *sweep*. Si hay un request en una pista que ya se visitó en este sweep deberá esperar al siguiente en una cola.

Hay variantes de este algoritmo, como **F-SCAN** que congela la cola de pedidos cuando hace sweep y los pedidos que lleguen durante el Sweep van a otra cola para ser atendidos luego, o **C-SCAN** (circular scan) que en vez de hacer sweep en ambas direcciones a través del disco, solo cambia desde el exterior al interior, y va de vuelta a la exterior directamente para empezar de nuevo.
C-SCAN es más justo para las pistas exteriores e interiores, ya que SCAN favorecía las interiores visitandolas dos veces. Por sus características, suele ser llamado elevator algorithm**.**

SPTF: Shortest Positionen Time First
Las dos políticas de scheduler anteriores seguían el principio de SJF pero ignoran el tiempo de **rotación**. SPTF elige el siguiente request a cumplir dependiendo del tiempo relativo del seek comparado a la rotación. Si el seek time es más alto se usa SSTF, pero si el seek es más rápido que la rotación, SPTF sirve para mejorar la performance.



En este ejemplo, convendría responder al pedido del bloque 8 antes que al 16.

Otros problemas de scheduling
Los discos pueden acomodar múltiples pedidos y tener sofisticados schedulers internos ellos mismos, por lo que el **scheduler** del **SO** solo le pasa los pedidos que ve como las mejores opciones al scheduler del disco el cual, con el conocimiento de la posición del cabezal, y de otros parámetros, realiza el scheduling final.

Otra tarea hecha por los schedulers del disco es el **I/O merging**; ante un pedido de, por ejemplo, los bloques 33 y 34, el scheduler los combina en uno solo del bloque 33, reduciendo overheads.

El SO puede seguir una política de enviar los pedidos de I/O al disco tan pronto como los recibe, lo cual es llamado **work-conserving** (el disco nunca para mientras haya pedidos), o esperar un poco por si llega un nuevo y mejor pedido de I/O, lo cual es denominado **non-work-conserving** y mejora la eficiencia general (según investigaciones sobre *anticipatory disk scheduling*).

39: Archivos y directorios

Un dispositivo de almacenamiento persistente (**persistent storage**), como un **disco duro** o un **disco de estado sólido**, almacena información de forma permanente (al menos un largo tiempo).

Archivos y Directorios
Para la virtualización del almacenamiento se utilizan dos abstracciones clave:

El **archivo** (**file**) es un array de bytes, donde cada uno puede ser escrito o leído. Cada archivo tiene un **low-level name** (el nombre por el cual el SO lo identifica), normalmente un número llamado **inode number**.
La mayoría de los SO no conoce la estructura del archivo (imagen, archivo de texto, etc.); la única responsabilidad del **file system** (sistema de archivos) es almacenar dichos datos de forma persistente en el disco para que estén disponibles al ser requeridos.

El **directorio** (directory) tiene un low-level name y su contenido es una lista de pares "**nombre** que ve el usuario" y "**low-level name**" (un diccionario que asocia nombres e inode numbers).
Los directorios son almacenados en otros directorios, creando así un **árbol de directorios** (o **jerarquía de directorios**; directory hierarchy). Dicha jerarquía comienza en un **directorio raíz** (**root**) y se usa un **separador** para nombrar los subsecuentes **sub-directorios** hasta el archivo o directorio deseado. Siguiendo dicho nombre se puede obtener el **absolute pathname** (**ruta absoluta**). Los directorios y archivos pueden tener el mismo nombre mientras no estén en la misma localización del árbol del sistema de archivos.

Los archivos suelen tener dos partes en su nombre separadas por un punto; la primera es el nombre del mismo y la segunda indica el **tipo** de archivo. Esto es solo una **convención**, no hay nada que asegure que un archivo contenga lo que declara contener.
Si un archivo pesa menos de 12 bits, puede ser guardado directamente en el **inode** (en lugar de en un bloque de data) para ahorrar espacio.

La relación entre el tamaño de los archivos (file size, FS) y el espacio que verdaderamente usa en el disco (disk usage, DU) puede ser:

- **FS > DU**: El sistema "miente" indicando espacio libre que en realidad no está disponible, apuntando a un bloque de disco que no existe.
- **FS = DU**: Ocurre en sistemas como Ext4.
- **FS < DU**: Sucede al guardar archivos menores al tamaño mínimo de bloque, o al guardar una traza de inodes de forma secuencial.

Interfaz del File System
Las siguientes son diferentes formas en las que un File System permite interactuar con el almacenamiento persistente y las abstracciones creadas sobre este:

Crear Archivos
Puede hacerse con la system call ***open*** con la bandera ***O_CREAT***:

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
                     S_IRUSR|S_IWUSR);
```

Las **banderas** del ejemplo indican que si el archivo no existe debe ser creado (**O_CREAT**), que solo puede ser escrito (**O_WRONLY**), y que, si el archivo ya existe, debe ser truncado a 0 bytes borrando su contenido (**O_TRUNC**). El tercer parámetro especifica los **permisos** que tendrá el archivo, haciendo que pueda ser leído y escrito por el creador.

La función *open* devuelve un **file descriptor** (fd); un entero privado por cada proceso usado en sistemas UNIX para acceder a archivos; una vez un archivo es creado se utiliza el fd para leer/escribir en el mismo (si se tienen los permisos), por lo que se considera que el fd es una "**capability**" (**capacidad**); un poder para realizar ciertas operaciones.

Los file descriptors son manejados por el SO y almacenados en la estructura del proceso. Un array (con un número máximo) mantiene un registro de los archivos que tiene abierto a la vez cada proceso. Cada entrada del array es un puntero a la *estructura del archivo*, y permite ver la información del mismo.

Leer y escribir en archivos

Para acceder y leer un archivo ya existente se debe abrir (***open***) el archivo y luego usar la syscall ***read***. Debe recordarse luego cerrar el archivo.

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)       = 3
read(3, "hello\n", 4096)                = 6
write(1, "hello\n", 6)                  = 6
hello
read(3, "", 4096)                       = 0
close(3)                                = 0
...
prompt>
```

Ejemplo de traza de system calls llamadas por 'cat' sobre el archivo 'foo'.
*Para rastrear las syscalls hechas por un proceso durante su ejecución se usa **strace**.*

En el programa de ejemplo se abre un archivo sólo para lectura. Open devuelve el file descriptor 3 (3 porque cada proceso ya tiene abiertos 3 fd), por lo que a partir de ahora el SO ya sabe a qué archivo se refiere el proceso con fd 3. El primer *read* lee "hello", luego usa stdout fd = 1 para escribir (*write*) por pantalla lo que leyó. Al intentar leer nuevamente ve que ya no queda nada por leer, por lo que cierra el archivo con ***close(3)*** (refiriéndose al fd 3).

Leer y escribir en archivos de forma no secuencial

Para escribir o leer de un archivo de forma no secuencial, sino desde un lugar determinado, se usa un **offset** para indicar el lugar de comienzo (a diferencia del caso anterior que usaba el principio del archivo). Para ello se usa la system call ***lseek()***:

```
off_t lseek(int fildes, off_t offset, int whence);
```

El primer argumento usado es el **fd**, el segundo es la **posición** en el archivo, y el tercero determina cómo se realiza la **búsqueda** (absoluta, relativa o dependiente del tamaño del archivo):

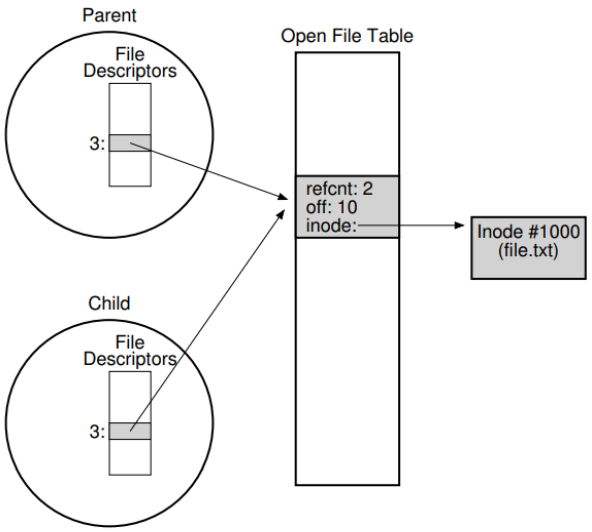
- Si whence==SEEK_SET, el offset se setea a offset bytes
- Si whence==SEEK_CUR, el offset se setea a su posición actual + offset bytes
- Si whence==SEEK_END, el offset se setea al tamaño del archivo + offset bytes

Parte de la abstracción de un archivo abierto es que siempre se tiene un **offset actual** (lugar donde 'se está') que determina dónde hacer el siguiente lectura o escritura, y se actualiza implícitamente cuando ocurre un read/write de N bytes (offset = offset + N) o explícitamente con *lseek()*. El offset se guarda en la estructura propia del archivo, la cual también permite al SO determinar los permisos del mismo a qué archivo se apunta (inode) y el current offset. Dichas estructuras representan a todos los archivos abiertos, y el conjunto de ellas es la **open files table**.

Entradas compartidas de la file table

Generalmente la relación entre fd y una entrada en la file table es 1-a-1. Incluso si más de un proceso lee el mismo archivo al mismo tiempo, cada uno tendrá su propia entrada en la open file table ya que cada lectura/escritura es independiente y tiene su propio offset.

Sin embargo, una **entry** en la open file table puede ser **compartida**. Esto sucede cuando un parent crea un child con ***fork()***, donde el child ajusta su offset con *lseek()* y luego se crea. Esto resulta útil cuando se está trabajando de forma cooperativa en el mismo archivo. Otro caso de entry compartida es el de la ***dup()*** syscall, la cual permite al nuevo proceso crear un otro fd que refiera al mismo archivo que ya está abierto. Esto es útil cuando se escriben operaciones de **redirección** (por ej. de output), creando un fd que apunte al archivo deseado e intercambiando el stdout que tiene un proceso por dicho fd.



Entrada de la Open File Table compartida por dos procesos; parent y child.

El campo refcnt (**reference count**, contador de referencias) indica la cantidad de procesos que han abierto el archivo de la entry, la cual solo será removida cuando tal contador llegue a cero.

Escribir de forma inmediata

Cuando un proceso llama a *write()* le solicita al SO que escriba ciertos datos en **almacenamiento** persistente en algún momento del futuro cercano. El file system, por razones de desempeño, **amortigua** (**buffer**) dichos writes en memoria por algún tiempo antes de ser ejecutados por el dispositivo de almacenamiento. Aunque este proceso puede ser rápido, hay casos en los que un crasheo puede llevar a la pérdida de dicha memoria si esta aún no llegó a escribirse en disco.

Para aplicaciones **críticas**, existen protocolos de recuperación de datos que necesitan la habilidad de **forzar** escrituras a disco. El file system de UNIX provee una API llamada ***fsync****(int fd)*, la cual fuerza a escribir todos los datos **sucios** (**dirty**, aun en el buffer) al disco, en el archivo especificado por el fd, y retorna cuando todos los writes se hayan completado.

Renombrar archivos

En una línea de comandos puede lograrse con el comando *mv*, el cual llama a la syscall ***rename****(char *old, char *new)*, que toma el nombre viejo y el nuevo. La syscall rename es crítica, por lo que normalmente es ejecutada de forma **atómica**.

Obtener información sobre archivos

El sistema mantiene cierta información sobre cada archivo, la cual es llamada **metadata**. Para leerla se usan las syscalls ***stat()*** o ***fstat()***, las cuales toman un pathname o fd a un archivo y leen una estructura stat en la cual se encuentran datos como size, inode number, ownership, última vez que fue accedido o modificado, etc.

La mayoría de los sistemas guardan esta información en una estructura llamada **inode**, los cuales están en disco pero normalmente se copian a memoria aquellos que están en uso para lograr un acceso más rápido (ya que seguramente deban ser actualizados).

Eliminar archivos
El comando *rm* usa, entre otras llamadas varias, la syscall ***unlink()***, la cual toma el archivo a ser removido y devuelve 0 si tiene éxito. Notar que su nombre es unlink, no delete ni remove.

Crear directorios
Se utiliza la syscall ***mkdir().*** Una vez creado el directorio, este se considera vacío aunque se genera con dos entradas dentro; **sí mismo** '.' y su **parent** '..' ("padre" en el árbol de directorios).

No se puede escribir directamente en un directorio; usar *write()* escribiría en la metadata del mismo, la cual está protegida por el file system. Debe escribirse en el directorio de forma **indirecta**, ya que es la forma en la que el sistema protege la integridad de los directorios.

Leer directorios
En vez de abrir un directorio como si fuera un archivo, se usan un grupo de 3 syscalls: ***opendir()***, ***readdir()*** y ***closedir()***. Estas se llaman en un ciclo que lee las **entradas** del directorio de a una hasta que esté vacío, imprimiendo cada vez el **nombre** e **inode** del archivo que lee.

La estructura interna de un directorio (struct **dirent**) mapea el nombre de cada archivo del directorio con su inode number y otros datos (offset al siguiente dirent, tamaño y su tipo).

Eliminar directorios
Los directorios se borran llamando a ***rmdir()***, la cual requiere que previamente el directorio esté vacío, con el fin de evitar eliminar archivos involuntariamente.

Hard Links
Una forma de crear una **entry** en el árbol del file system (además de crear un nuevo archivo) es a través de la syscall ***link()***, la cual toma como argumentos un pathname viejo y uno nuevo. Al hacer *link* de un archivo viejo con uno nuevo se crea una nueva forma de referirse al mismo archivo viejo; ahora dos nombres refieren al **mismo archivo** (al mismo inode number que el archivo original) pero el archivo en sí no se copia de ninguna forma.

Al generar un file se crea una estructura (inode) que contiene la información sobre el archivo, y luego se "linkea" un nombre que el usuario puede ver a ese inode y se pone ese link en el directorio. Al usar *unlink()* para borrar un archivo, se reduce el **reference count** (**link count**) del inode number y, si ya no hay referencias, se libera el inode y los bloques de memoria asignados

Links simbólicos
Los hard links son limitados ya que no se puede crear uno para un **directorio** (para evitar un ciclo en el árbol de directorios) y no se puede hacer un hard link a archivos almacenados en otras **particiones** del disco (porque los inodes solo son únicos en cada file system, no se comparten entre file systems, por lo que se podría repetir el mismo número). En estos casos se usa un **symbolic link** (o **soft link**). Su creación es similar a la de un hard link y funciona de la misma forma, pudiendo acceder a un mismo archivo a través de dos nombres diferentes.

La diferencia reside en que un soft link es un **archivo** en sí mismo, de un tipo diferente; no es un file ni un directorio, sino que es de un ***tercer tipo***. Un soft link está formado por el **pathname** del archivo al que está linkeado en forma de un **datos** dentro del archivo soft link.
Por la forma en la que están creados se abre la posibilidad de una **dangling reference** (referencia colgante) si se borra el archivo original (ya que eso no implica que el soft link se borre).

Bits de permiso
El file system presenta una virtualización amigable del disco. A diferencia de las abstracciones del CPU y la memoria, los archivos suelen ser compartidos entre procesos y no ser privados.

El mecanismo de **permission bits** (bits de permiso) consiste en asignar a cada archivo/directorio/softlink una serie de bits que determinan quién puede acceder a los mismos. Se dividen en tres grupos: lo que puede hacer el **dueño** del archivo, lo que puede hacer alguien que pertenezca a un **grupo**, y lo puede hacer alguien cualquiera (normalmente llamado **other**).

```
prompt> ls -l foo.txt
-rw-r--r--  1 remzi wheel  0 Aug 24 16:29 foo.txt
```

Ejemplo de Bits de permiso para un archivo; lectoescritura para el dueño, y solo permiso de lectura tanto para el grupo como para "cualquiera".

Consisten en 10 bits. El primero indica el **tipo**; '-' para un archivo, 'd' para un directorio, y '1' para un soft link. Los siguientes 9 indican los **permisos**; 3 para el owner, los siguientes 3 para el grupo y los últimos 3 para cualquiera. Estos paquetes de tres consisten en un bit de **read**, uno de **write** y uno de **ejecución**, los cuales valen 'r', 'w' y 'x' si están permitidos, o '-' si no lo están.

El dueño del archivo puede cambiar estos permisos con el comando ***chmod***, el cual modifica el **file mode** del archivo. En los directorios el bit de ejecución otorga permiso al usuario para hacer cosas como cambiar directorios dentro del mismo, lo que en combinación del bit de write permite crear archivos dentro del directorio.

Otros filesystems usan controles diferentes, como por ejemplo un **ACL** (**access control list**) por cada directorio; una forma más poderosa de representar quién puede acceder a cada recurso.

Crear y montar un File System
Para ensamblar el árbol de directorios a partir de muchos sistemas de archivos subyacentes, se crea un nuevo file system y se lo monta para hacer sus contenidos accesibles. Para ello se usa la herramienta ***mfks***, la cual recibe como input un dispositivo (la partición de un disco) y un tipo de file system, y escribe un file system vacío empezando en el directorio **root** de esa partición.

Una vez dicho file system es creado, debe ser accesible dentro del file system tree más general, para lo cual se usa el programa ***mount***. Esta toma el **mount point** (punto de montaje) y un directorio ya existente, y pega un el file system a montar en el árbol de directorios en ese punto.

40: Implementación del File System

El File system es la estructura de datos en disco que mantiene los archivos de forma consistente. El kernel mantiene diferentes Virtual File System para todos los formatos que sabe leer; UFS, FAT, exFAT, NTFS, ISO, EXT, etc. y cada partición puede tener un formato diferente (cada uno de los cuales tiene sus features y limitaciones).

Syscall → Virtual File System → Device Driver → Device

El **VSFS** (the **Very Simple File System**) es una versión simplificada de un sistema de archivos de Unix (**UFS**: Unix File System) usada para introducir diferentes estructuras del disco, métodos de acceso y políticas.

Cómo implementar un file system
El funcionamiento del modelo se basa en dos aspectos clave:

Las **estructuras de datos** en el file system: Los tipos de estructuras on-disk (en disco) utilizadas por el file system para organizar sus datos y su metadata. Los sistemas más sencillos usan estructuras simples (arrays de bloques), los más complejos emplean estructuras de tipo árbol.

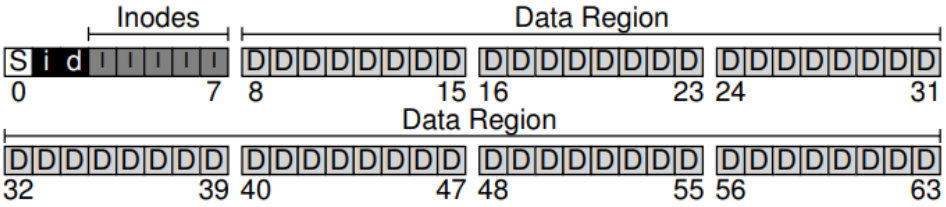
Los **métodos de acceso** del file system: Cómo se mapean las llamadas hechas por los procesos (open, read, write) en las estructuras de datos, qué estructuras se leen/escriben en la ejecución de cada syscal en particular, y qué tan eficientemente se ejecutan esos procesos.

Organización general
En VSFS el disco es dividido en **bloques**, numerados de 0 a N-1, usando un tamaño único de bloque de 4KB. La mayor parte de estos bloques es usada para almacenar datos de usuario en lo que es llamada la **data region**.

La metadata que el file system guarda para cada archivo (permisos, tamaño, dueño, cuando se creó, cantidad de bloques asignados, etc.) es guardada en una estructura llamada **inode**, para la cual se reserva un espacio en determinados bloques del disco llamado **inode table**. La cantidad total de inodes almacenados (en cada bloque, por la cantidad de bloques reservados para ello) es el número **máximo de files** que el file system va a poder almacenar.
Generalmente, cualquier información en el file system que no sea user data es llamada metadata.

Las **allocation structures** (estructuras de asignación) son usadas para determinar si los bloques están ocupados o libres. Para ello VSFS emplea un **bitmap**, estructura la cual usa un bit para cada bloque asignado (0 = libre, 1 = en uso). Se necesita un bitmap por cada tipo de bloque a referenciar (uno para los de **inode**, otro para los de user **data**) y por simplicidad se usa un bloque para almacenar cada inode (por más que un bitmap de dos bloques de 4KB cada uno podría ver hasta 32K objetos asignados).

El bloque 0 es usado para el **superblock** (superbloque), el cual contiene la información sobre el file system (cantidad de inodes y data blocks, inicio de la inode table, etc.) y es leído por el SO al momento de montaje para inicializar diferentes parámetros y añadir el volumen al file-system tree.



Disco separado en N = 64 bloques, con una región de datos, una de inodes (inode table de 5 bloques), dos bloques de bitmap y un bloque de superblock.

El inode

El nombre de **inode** proviene de **index node**, ya que originalmente los nodos se organizaban en un array. Cada inode está implícitamente referenciado por un número (el **i-number**); el **low-level name** mencionado en el capítulo anterior. En VSFS dado un i-number podemos calcular donde está almacenado en el disco el inode al que se refiere.

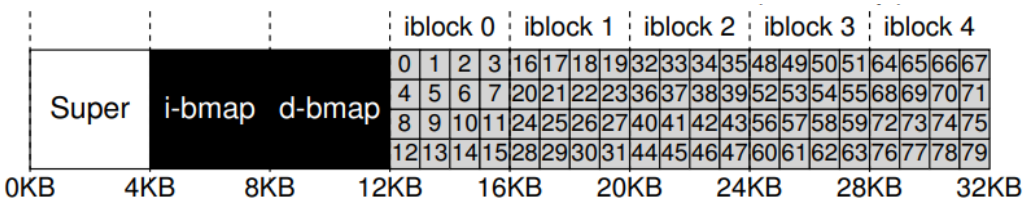


Tabla de inodos de 20KB (cinco bloques de 4KB); 80 nodos de 256 bytes cada uno.

Por ejemplo, para leer el inode 32 en la tabla anterior, el file system primero calcula el **offset** de la región de inodos (inumber · sizeof(inode_t) = 32 * 256B = 8192B = 8KB), lo suma a la **dirección inicial** de la inode table (12KB), y obtiene la dirección del bloque de inodes deseado (20KB block).

Los discos no son direccionables por byte sino por sectores, normalmente de 512B cada uno. Por ende, para señalar el bloque de inodes que contiene al inode 32, el file system deberá primero calcular el sector en el cual se aloja el bloque. (20KB/512B = (20Bx1024)/512B = 40).

```
blk    = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

Ecuaciones para calcular bloque y sector.

Una forma en la que el inode puede referirse a donde están los bloques de datos es tener dentro uno o más **direct pointers** (disk addresses, punteros directos) en donde cada puntero se refiere a un bloque que pertenece al file. Dicho enfoque es limitado y no funciona para archivos más grandes que el tamaño del bloque multiplicado por el número de punteros directos en el inode.

Indexación de múltiple nivel

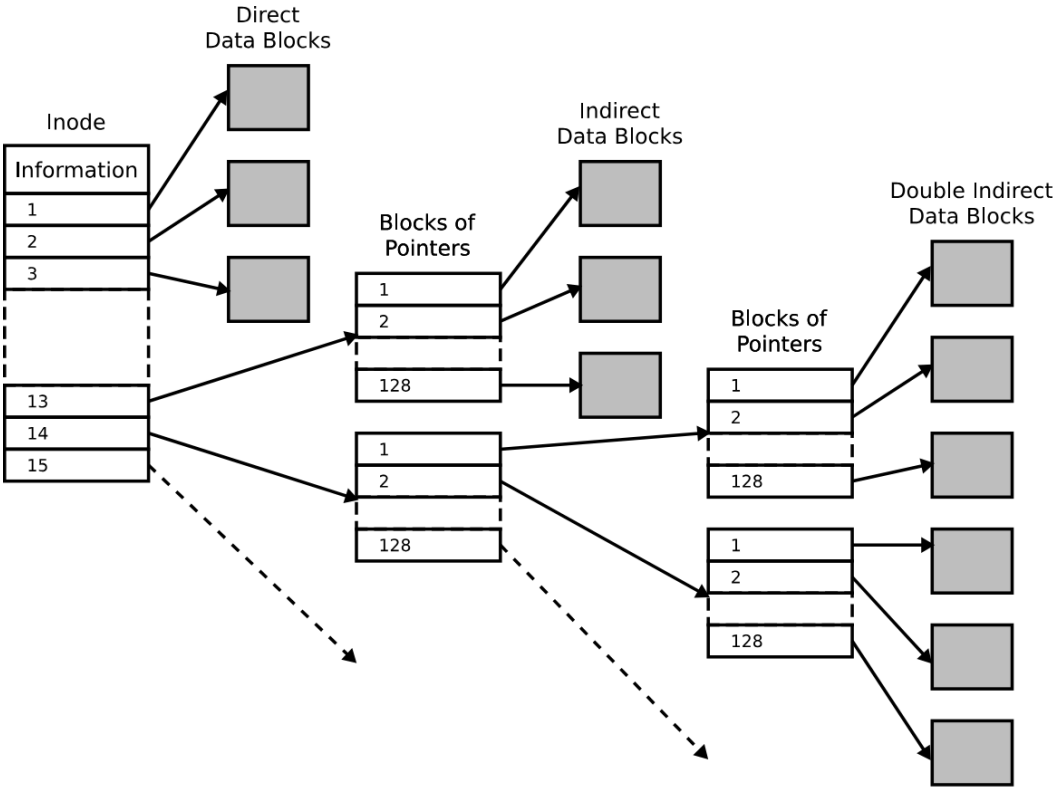
Una forma de dar soporte a archivos más grandes es tener un puntero especial llamado **indirect pointer** que en vez de apuntar a un bloque con user data apunta a un **bloque** con **punteros** que apuntan a **user data**. Si un archivo se pasa del límite de tamaño, se usa un bloque de data region como bloque indirecto y un indirect pointer del inode pasa a apuntar a este bloque de punteros.

Con bloques de 4KB y direcciones de disco de 4B, un bloque añade 1024 punteros y un archivo (que, por ejemplo, ya tenía 12 punteros directos) puede crecer hasta (12+1024) · 4KB = 4114KB.

Para dar soporte a archivos todavía más grandes se puede agregar un segundo puntero; el **double indirect pointer**. Este puntero apunta a un bloque que tiene punteros a bloques indirectos, los cuales contienen punteros a data blocks. De esta forma se pueden direccionar archivos de hasta 4GB. Si se requiere almacenar archivos más grandes, se puede hacer un **triple indirect pointer**.

Este enfoque **multi-level index** genera un árbol imbalanceado, pero como la mayoría de archivos almacenados en los sistemas son pequeños, es eficiente.

… → New double indirect pointer → New indirect pointer → New pointers block → User data

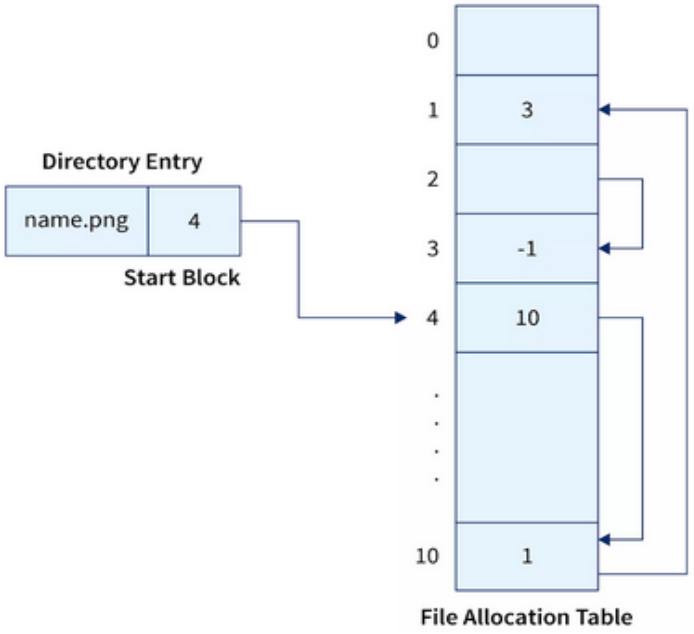


Estructura de inodes pointers con hasta doble indirección.

FAT

Otros filesystem, como FAT (file allocation table), usan enfoques diferentes al anterior; en lugar de punteros indirectos usan una linked list (lista ligadas) para cada archivo. Cada entrada del arreglo Directory Entry apunta al primer bloque del archivo, el cual apunta al siguiente bloque (no necesariamente contiguo), etc. El último bloque del archivo apunta a EOF (por ejemplo '-1').
Al no almacenar en el directory entry otras entradas (como el size del archivo), este proceso secuencial hace que FAT sea malo para búsquedas al azar, ya que las lecturas son de orden N (depende del tamaño del archivo).

Si dos archivos apuntan a un mismo bloque ("cadenas cruzadas"), o si se produce un ciclo, se produce una **corrupción** del filesystem. El programa para detectarlo es chkdsk.



Implementación del FAT File System; forma de almacenar bloques de archivos.

FAT12 = 12 bits para índices de bloques → tengo hasta 2^{12} = 4.096 bloques → si cada uno es de 4K, el tamaño máximo del área de datos para ese FS es $2^{12} * 2^{12}B = 2^{24}B$ = 16MB.

Organización de directorios

En VSFS un directorio solo contiene una lista de pares (**entry name**, **inode number**); por cada archivo o directorio en un directorio se tiene un string y un número en el data block(s) del mismo.

```
inum | reclen | strlen | name
   5      12        2     .
   2      12        3     ..
  12      12        4     foo
  13      12        4     bar
  24      36       28     foobar_is_a_pretty_longname
```

Ejemplo de directorio 'dir' (inode number 5) con 3 archivos en disco (con inodes 12, 13 y 24). Además, se indica la longitud del nombre (incluyendo '\0') y el espacio sobrante (reclen). El directorio actual es la entrada '.', mientras que el directorio padre es la entrada '..'

Los file systems diferencian directorios de archivos normales especificando su tipo en un campo de sus inodes. El dir (directorio) tiene **data blocks** señalados por el inode (y tal vez bloques indirectos) que se almacenan en la data region.
Cada tabla de directorios puede apuntar a archivos y a otros directorios, pero por consistencia un mismo nombre no puede apuntar a dos inodes diferentes. Notar que nada impide que se usen ciclos de directorios.

Manejo del espacio libre

En VSFS se usan dos **bitmaps** (para **inodes** y para **data**) para el manejo del espacio libre que siguen la pista de que inodos y bloques de datos están libres y cuales asignados, permitiendo encontrar espacio al crear un file o directorio (y actualizar su estado a usado).
Algunas políticas de pre-allocation pueden ser usadas en este proceso, como elegir una secuencia de bloques libres de un tamaño mínimo, para buscar un mejor rendimiento al usar una porción contigua del disco.

Rutas de acceso: lectura y escritura

El flujo de operaciones necesarias para leer o escribir un archivo requiere involucrar una serie de operaciones de I/O y **access path** (rutas de accesos). En los siguientes casos se asume que el file system ya se montó (por tanto el superblock fue leído por el SO y cargado en memoria) pero que todo el resto (inodes, directorios, etc.) sigue en disco.

Leer un archivo del disco

El primer paso es llamar a *open*, para lo cual el file system necesita encontrar el inode del archivo para obtener la información del mismo (permisos, tamaño, etc.). Al recibir un pathname, primero debe atravesarlo para poder encontrar el **inode** del file. Para ello comienza en el **root directory** (/), leyendo el inode del mismo (para lo cual usa el **i-number** del inode root, el cual es conocido por el file system al ser montado y suele ser el número 2).

Luego, el FS busca **punteros** a **data blocks** en el inode de root, los cuales señalan a los contenidos de root dir y permiten al FS leer el directorio y buscar el inode del siguiente componente del path name. Este proceso se repite hasta atravesar todo el path name y llegar al inode deseado. El paso final de *open* es leer/escribir el inode de bar en **memoria**; el FS hace el check de **permisos**, asigna un **file descriptor** para este proceso y retorna al usuario.

Una vez abierto el archivo, se llama a *read*. A lo largo de la lectura se va actualizando el **last time access** y el **offset** del inode (el cual empieza en 0 si se lee desde el comienzo del archivo, o en otro valor si se usa lseek). Se lee primer bloque del archivo, consultando al **inode** para ver dónde se almacena dicho **bloque**, y se repite el mismo procedimiento con cada bloque del file.

En algún punto se cierra el archivo, para lo cual el FS solo desasigna el espacio que le dio al file descriptor, sin involucrar ninguna operación I/O. Notar que la cantidad de I/O que hace *open* depende de lo largo del path name; a mayor **longitud**, más inodes deben ser buscados y leídos.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**	**read**	**read**	**read**	**read**			
read()					**read** **write**			**read**		
read()					**read** **write**				**read**	
read()					**read** **write**					**read**

Operaciones de I/O en el proceso de lectura de un archivo con datapath de largo 2 ("/foo/bar"). Línea de tiempo descendiente.

Escribir en un archivo del disco
Se realiza un *open* que sigue el mismo proceso de atravesar el datapath que en el caso de las lecturas. Luego, la aplicación llama a *write* para actualizar los contenidos del archivo.

Escribir en un archivo puede requerir **asignar** (allocar) un nuevo bloque (a menos que se sobreescriba en uno), por lo que antes de escribir los datos en disco se debe decidir qué nuevo **bloque** asignarle, y por ende actualizar otras **estructuras** del disco de forma acorde (como el data bitmap y el inode).
Por este motivo, cada write genera **5** operaciones de **I/O**: una para **leer** el data **bitmap** (encontrar un bloque libre), otra para **escribir** en el **bitmap** (actualizarlo para marcar el nuevo bloque como asignado), dos para leer y luego **escribir** el **inode** (en el cual se actualiza la localización del bloque), y finalmente una más para **escribir** los **datos** en el **bloque** en sí.

El costo es aún más alto cuando se crea un archivo, ya que además de allocar el inode el sistema debe asignar espacio en el directorio que va contener el nuevo archivo (una entry).

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read	read	read write	read	read	write		
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

Operaciones de I/O en el proceso de creación de un archivo usando un datapath de largo 2 ("/foo/bar") y escribiendo en él 3 bloques. Línea de tiempo descendiente.
Notar que requiere de 10 I/Os para crear el archivo y 5 I/Os por cada alloc de memoria.

Caché y buffers
El alto costo de cada operación de I/O afecta el desempeño del filesystem. Por ello los primeros sistemas usaban una **fixed-size caché** (caché de un tamaño fijo) para mantener allí a los bloques más utilizados. Sin embargo, este **particionamiento estático** de memoria puede causar un desperdicio si se guarda más espacio de memoria para el caché de la que se usa realmente.

Los sistemas actuales emplean **particionamiento dinámico**, en donde se integran páginas de memoria virtual en un **caché de páginas unificadas**. De esta forma, la memoria puede ser asignada de forma flexible.
Esta caché es más útil en la lectura que en la escritura, ya que la primera no necesita un manejo individual para pasar los datos al disco de forma persistente, cosa que la escritura si. Frente a esto se usa un **write buffering**, el cual disminuye los I/Os al **retrasar** los *writes* y permitir al FS juntar varias actualizaciones en un solo **lote** (o directamente **evitarlos** si, por ejemplo, un proceso crea o escribe un archivo pero luego lo borra), y además incrementa el desempeño al almacenar en un buffer en memoria un número de *writes*, permitiendo al **scheduler** del sistema encargarse de los siguientes I/Os.

Como se vio en el capítulo anterior, existe un trade-off al usar buffers de escrituras, ya que un crasheo del sistema podría llevar a que se pierdan los datos que aún no han sido escritos en el disco, lo que hace que suelan ser usados por aplicaciones críticas.

41: Localidad y Fast File System

Los primeros filesystem eran simples y proveían las abstracciones necesarias (archivos y una jerarquía de directorios) mediante el uso de un superblock (S) que contenía la información de todo el file system (tamaño del volumen, cantidad de inodes, un puntero a la cabeza de una free list, etc.), una región con los inodes del sistema, y una gran región de data blocks.
Toda estructura necesaria por el filesystem que no sea datos de usuario, es **overhead**.

Estructura de un filesystem sencillo.

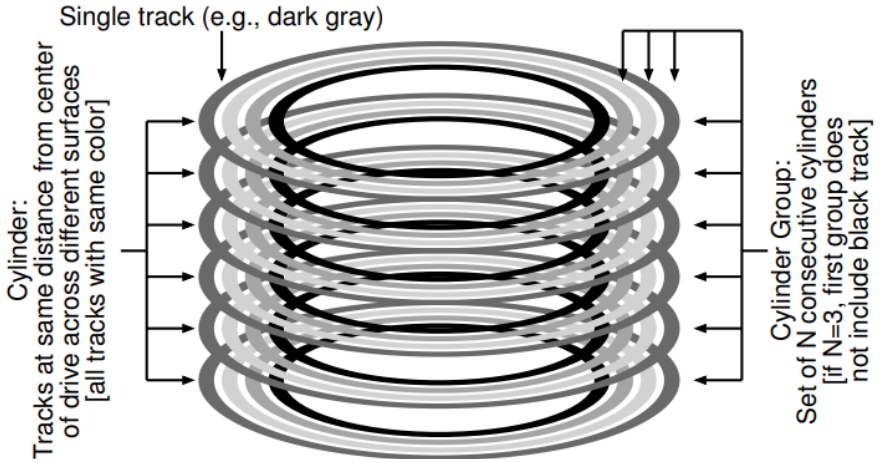
Sin embargo estos modelos tenían un mal desempeño ya que estaban diseñados sin tener en cuenta las características del hardware de almacenamiento (discos rotacionales con pistas y cabezales que se benefician de la **localidad** de los datos) por lo que esparcían los datos (bloques) en diferentes ubicaciones (como si de una memoria RAM se tratase) generando gran **fragmentación**, lo que llevaba a tener grandes tiempos de búsqueda para cada bloque ya que la lectura no se podía hacer de forma secuencial (incluso para bloques de un mismo archivo).
Frente a esto, eran muy usadas las herramientas de desfragmentación para reorganizar los bloques de los archivos (y los bloques libres) de posiciones contiguas.

Otro problema de estos modelos era el tamaño de bloque; al ser muy pequeños (512 bytes) se disminuía la **fragmentación interna** (minimizando el desperdicio de espacio dentro de cada bloque) pero hacía la transferencia de datos todavía más ineficiente al requerir más búsquedas.

FFS: Fast File System
Para solucionar esos problemas se creó el **Fast File System** (**FFS**) cuyo diseño de las estructuras del FS y de las políticas de asignación de espacio son “disk aware” (conscientes del disco). Si bien la implementación interna cambió, se mantuvo la misma interfaz (API).

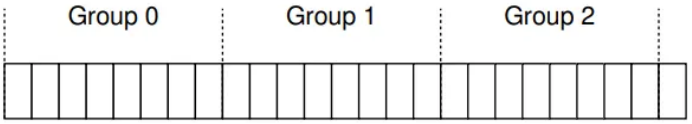
Grupo de cilindros

FFS organiza al disco en una colección de **cylinder groups** (grupos de cilindros). Un **cilindro** es un conjunto de pistas (tracks) en diferentes superficies del disco que están a la misma distancia del centro. Un número de cilindros consecutivos forman un **grupo**.



Tracks (pistas) en cilindros, formando grupos de cilindros en un disco de 6 platos.

Los discos no aportan información al file system como para que éste sepa si un cilindro particular está en uso; solo informan el address space de los bloques y esconden los detalles de su geometría. Es por eso que los sistemas modernos organizan el disco en **grupos de bloques**, cada uno de una **porción** consecutiva del address space del disco.



Sección de memoria vista como grupos de bloques.

Ya sea con cilindros o grupos de bloques, FFS asegura que si se colocan dos archivos en el mismo grupo y se accede a uno después de otro, no se van a necesitar largas búsquedas a través del disco. Para almacenar archivos y directorios en estos grupos, el FFS necesita tener la habilidad de ponerlos en un bloque y seguir la pista de toda la información necesaria sobre ellos. Para esto, incluye en cada grupo de cilindros a las **estructuras** de un file system; espacio para data blocks, inodes, estructuras para localizar los lugares libres y asignados (un bitmap por región inodes/data) Además, por razones de fiabilidad se coloca una copia de respaldo del superblock.

Contenido de un grupo de cilindros usando FFS;
Superblock, inode bitmap, data bitmap, inodes, datos.

Política de asignación

El FFS debe decidir cómo **posicionar** archivos y directorios (y la metadata asociada), buscando mantener *"cosas **relacionadas** juntas, y no relacionadas separadas"* para mejorar la performance. Para esto sigue dos políticas que permiten que los archivos estén cerca de sus inodes y que los files en un directorio estén cerca unos de otros:

Para el posicionamiento de **directorios** busca un grupo de cilindros con un bajo número de directorios (para mantener un **balance**) y un alto número de inodes libres, y pone los datos del directorio y su inodo en ese grupo.
Para posicionar los **archivos** trata de asignar los bloques de cada file en el mismo **grupo** de **bloques** que su inode (para prevenir largas búsquedas), y además posiciona a todos los archivos que están en un directorio en el mismo **grupo** de **cilindros** en el que está dicho directorio.

Para los **archivos grandes** se aplica una excepción en la política de posicionamiento. Para evitar que estos llenen un grupo de bloques por completo, cada cierto número de estos asignados en el primer grupo de bloques, FFS posiciona el siguiente "gran" pedazo del archivo en otro grupo (tal vez elegido por su bajo uso), y así subsecuentemente hasta almacenar todo el archivo.
Para evitar que esta separación en partes afecte la performance, se elige el tamaño de esas partes con cuidado para lograr que el tiempo usado en buscar la siguiente parte sea pequeña en comparación al usado para transferir los datos de esa parte. Esta técnica de reducir el overhead haciendo más trabajo por cada "paid overhead" se llama **amortization**.

Otros aspectos de FFS
Para disminuir la fragmentación interna en los bloques de 4KB FFS utiliza **sub-bloques** de 512 bytes los cuales el file system asigna mientras el archivo crece, y al llegar a los **4 KB** libera los sub-bloques luego de copiarlos todos a un bloque de 4 KB.
Para evitar los I/Os extra que esto supe, se suele utilizar un **buffer** de escrituras para hacerlas directamente en bloques de 4KB, evitando la creación de sub-bloques cuando son innecesarios.

Otro aspecto de FFS es la técnica de **parametrización**. Durante las lecturas secuenciales es lo suficientemente rápido para requisitar los siguientes bloques a leer antes de que estos pasen por el cabezal del disco, evitando rotaciones extra.
Además, a nivel usuario, FFS permitió **nombres más largos en los archivos** (más de 8 char) e introdujo los conceptos de **symbolic link** y *rename* atómico.

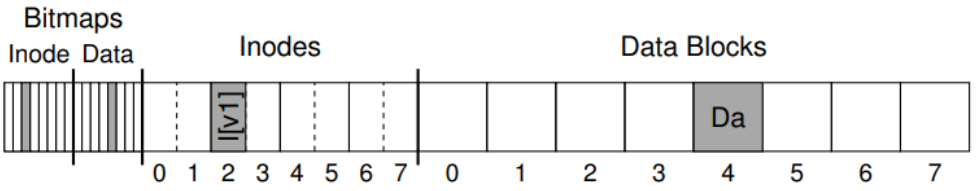
42: Consistencia ante errores: FSCK y Journaling

Las estructuras de datos del file system, a diferencia de las de la memoria, deben ser **persistentes**. Por ello es necesario poder actualizarlas manteniendo consistencia incluso ante **pérdidas de energía** o **crasheos del sistema** mientras se realiza ese proceso.

Solo una operación de lectura puede ser ejecutada a la vez, por lo que la posibilidad de dejar una estructura de disco en estado **inconsistente** no sólo está presente mientras se realiza una escritura, sino para las operaciones que se encuentran esperando en el buffer.

El siguiente es un ejemplo de una **carga de trabajo** (**workload**) que actualiza estructuras de disco adjuntando un data block a un archivo existente abriendo el file, llamando a lseek() para mover el file offset al final del file, y luego realizando un write de 4 KB al file antes de cerrarlo.
Se asume una estructura de datos estándar, un inode bitmap (8 bits, 1 por inode), un data bitmap (8 bits, uno por bloque de datos), 8 inodes en 4 bloques y 8 data blocks:

```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```



Primera versión del inode; I(v1).
Estructura de disco con el bloque 4 y el inode 2 asignados y marcados en sus bitmaps.

Al agregar un nuevo bloque se deben **actualizar** 3 estructuras en disco: **inode** (*pointer* hacia el nuevo bloque y actualizar el tamaño del file), **data block** (en la data region structure), y el **bitmap** de los data block.

Es decir, el file system debe realizar 3 writes en disco, lo cual no puede suceder inmediatamente ni a la vez; los nuevos datos esperan en un **page/buffer cache** de memoria hasta que el FS decida escribirlo y el filesystem realice la *write* call. Si un crasheo ocurre una vez comenzados a escribirse en disco pero antes de que se completen los 3, el file system queda en un “funny state”.

Escenarios de crash
Casos en los que solo un write se completa:

- Solo el **data block** se actualizó: los datos están en disco pero ningún inode apunta a ellos y el bitmap no indica que el bloque esté asignado, por lo que es como si la escritura nunca hubiera sucedido. No es un problema desde el punto de problema de consistencia.
- Solo el **inode** se actualizó: inode apunta a donde debería estar el Db, pero allí sólo se encuentra **basura**. El bitmap muestra al bloque como libre/no asignado pero el inode indica que hay información, por lo que se produce una inconsistencia en las estructuras de datos.
- Solo el **bitmap** se actualizó: se muestra al bloque como asignado pero no hay inode que apunte a este. Esta inconsistencia generará un **space leak** (pérdida de espacio) ya que el bloque nunca podrá ser usado por el file system.

Casos en los que solo dos writes se ejecutan:

- **Inode** y **bitmap** actualizados: la metadata es consistente ya que el inode apunta al bloque correcto y el bitmap lo marca como asignado, pero el mismo contiene **datos basura**.
- **Inode** y **data block** actualizados: el inode apunta al bloque correcto y los datos están ahí, pero el bitmap no marca el bloque como asignado, lo que genera una inconsistencia.
- **Bitmap** y **data block** actualizados: Se genera una inconsistencia entre el inode y el bitmap. El bloque fue escrito y se indica que está en uso, pero ningún inode lo señala por lo que no se sabe a qué file pertenece.

Ante la posibilidad de que se generen estructuras de datos inconsistentes, lectura de datos basura al usuario y pérdidas de espacio, se busca modificar al file system de forma **atómica** entre un estado consistente y otro nuevo. Sin embargo el disco solo puede una escritura al mismo tiempo, lo cual genera un **crash consistency problem**.

Solución 1: Verificar el sistema de archivos (FSCK)

El enfoque consiste en dejar que sucedan inconsistencias y, antes de que el filesystem se monte, ejecutar **FSCK** (**file system checker**) para verificar su estado, encontrar esas inconsistencias y arreglarlas. Para garantizar el buen estado del filesystem FSCK analiza:

- **Superblock**: chequeos de sensatez, principalmente que el tamaño del file system sea mayor al de los bloques asignados. Si está corrupto, se puede usar una copia alternativa.
- **Free blocks**: se escanean inodes e indirect blocks para saber qué bloques están asignados al filesystem y generar una versión correcta de los bitmaps e inodes asignados.
- **Inode state**: cada inode es chequeado por corrupción y otros problemas (por ej. que su campo type sea válido). Si se detectan inconsistencias que no puedan arreglarse, el inode es eliminado y el bitmap correspondiente es actualizado acorde a ello.
- **Inode links**: se comprueba la cuenta de links de cada inode (cantidad de referencias al archivo) comparándolo con el resultado de escanear el árbol de directorios desde root. Si hay discrepancias se arreglan actualizando el valor del inode. Si se encuentra un inode asignado pero sin referencias, se lo mueve al directorio *lost+found*.
- **Duplicados**: se chequean punteros duplicados (dos inodos apuntando al mismo lugar) y puede decidirse copiar el bloque para que cada inode tenga el propio (y actualizar uno de esos punteros). Si un inode se detecta como corrupto es eliminado.
- **Bad blocks**: se chequean punteros a bloques corruptos escaneando la lista de punteros. Un puntero es considerado corrupto si apunta fuera de su rango permitido (por ej. a un bloque por fuera de su partición) y en este caso es eliminado.
- **Directory checks**: se verifica que las primeras dos entradas de cada directorio sean '.' y '..', que cada referencia a un inode esté asignada, y que ningún directorio esté conectado a más de un 'padre' en la jerarquía de directorios.

Al realizar tantas comprobaciones exhaustivas, fsck resulta costoso y especialmente **lento** para discos de gran capacidad, por lo que en la actualidad otras soluciones son más utilizadas.

Solución 2: Journaling (Write-Ahead Logging; registro de escrituras)

Al momento de actualizar el disco, antes de sobreescribir las estructuras en su lugar, se escribe en una estructura organizada como un **log** (registro) una descripción de la modificación que se está por hacer, procedimiento llamado **write ahead**.

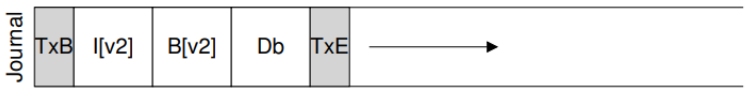
El log garantiza que, después de un eventual crasheo en medio de la operación de actualización del disco, su información va a permitir reintentar la escritura sabiendo exactamente qué parte del disco debe repararse y cómo hacerlo (en vez de tener que escanear el disco entero). Journaling añade algo de overhead en cada actualización, pero reduce mucho el tiempo de **recuperación**.

ext3

Un filesystem con journaling usado por linux es **ext3**. Este divide al disco en grupos de bloques, cada uno con un inode bitmap, data bitmap, inodes y data blocks. La estructura del journaling ocupa un pequeño espacio en la partición.

Estructura del Journal en el disco.

Data journaling
Antes de realizar un update a disco, debe escribirse el log. Este consta de un bloque inicial **TxB** con información sobre el update (como la dirección de destino y un **transaction identifier** (**TID**); un ID de la operación), una serie de bloques con el contenido de los bloques a escribir llamados **physical logging** (la información física exacta de la update en el journal), y un bloque **TxE** que indican el final de la operación y contiene una copia del TID,



Log de escritura de inode I(v2), bitmap B(v2) y data block (Db).

Una vez dicha operación está en disco, se puede comenzar a ejecutar el pedido de write en sí y actualizar las data structures en disco (escribir el contenido de los bloques de physical logging), proceso llamado **checkpointing**.

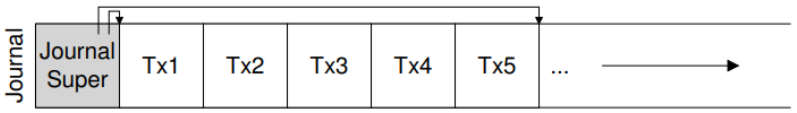
Para evitar inconsistencias provenientes de posibles crasheos durante las escrituras del journal, el filesystem primero hace el transaccional write de todos los bloques menos el TxE, y al finalizar escribe el TxE. La escritura de este último logra ser **atómica** gracias a que el disco garantiza la atomicidad de writes de 512 bytes.

Recovery
Para recuperarse luego de un crasheo (en cualquier parte de la secuencia de escritura), el filesystem usa los contenidos del journal:

- Si el crash es antes de que la actualización pendiente se escriba en el log (journal commit), se saltea/ignora y el write no ocurre.
- Si el crash ocurre después del commit pero antes de que el checkpoint se complete, el filesystem **recupera** la update escaneando el log y buscando las transacciones encomendadas en el disco en el booteo del sistema. Luego, dichas transacciones vuelven a ejecutarse (son **repetidas** en orden). Es por esto que esta forma de logging es llamada **redo logging**. Un crasheo durante esta secuencia sólo llevaría a repetir el proceso nuevamente.

Agrupar actualizaciones del log
Debido a que hacer un commit en cada update del disco puede añadir bastante tráfico al mismo, algunos sistemas agrupan los updates en un buffer para evitar writes excesivos.

Hacer el log finito
Para evitar que se llene la estructura finita del Log y se inutilice el sistema (al no poder commitear más transactions al disco), los filesystems con journaling tratan el log como una estructura de datos circular, reutilizándola una y otra vez (por esto journal es llamado **circular log**).
El sistema libera el espacio de la transacción asignada del log una vez completado el checkpoint, marcando la misma como **libre** en el **superblock** del Journal. En este último se guardan los datos de las transacciones todavía no llegaron al checkpoint.



Superblock y contenidos del journal.

De esta forma, el protocolo de journaling consta de cuatro partes:

1. **Journal write**: escribir la transacción en el log (TxB y contenidos a actualizar).
2. **Journal commit**: escribir el transaction commit block (TxE) en el log; lo que pone a la transacción en estado de **committed** (encomendada).
3. **Checkpoint**: escribir los contenidos de la update (metadata y data) en su destino en disco.
4. **Free**: marcar la transacción como free en el journal al actualizar el journal superblock.

Se espera a que cada paso finalice antes de comenzar con el siguiente.

Metadata journaling

Notar que, si bien data journaling permite una recuperación rápida ante crasheos, cada bloque se está escribiendo en disco dos veces, lo que genera un gran **overhead** debido al alto costo de I/O.

Una forma de aumentar la performance es usar modelos como **ordered journaling** (también llamado **metadata journaling**), en los cuales la data no es escrita en el journal; esta es directamente escrita en disco y solo la metadata es copiada en el journal.

El protocolo de metadata journaling consiste en:

1. **Data write**: escribir los datos en su dirección final.
2. **Journal metadata write**: escribir la transacción en el log (TxB y contenidos a actualizar)
3. **Journal commit**: escribir el transaction commit block (TxE) en el log.
4. **Checkpoint metadata:** escribir el contenido de la nueva metadata en su destino en disco.
5. **Free**: marcar la transacción como free en el journal al actualizar el journal superblock.

Notar que con el paso 1 se evita que ante un crasheo un puntero quede apuntando a datos basura, ya que la información es escrita en disco antes de crear al puntero mismo.