**Chapter 2 Introduction to OS**

OS= sistema operativo

Proceso = proc —> programa en ejecución—> ejecuta instrucciones

Ejecución: El procesador toma una instrucción de memoria, la decodifica(interpreta) y luego ejecuta.

El OS es el software que se encarga de que sea fácil correr programas de forma eficiente y correcta.

Para este cometido utiliza la virtualización del hardware y lo transforma en una forma virtual más simple y poderosa de usar. Debido a la virtualización los programas pueden correr en simultáneo compartiendo CPU, memoria y dispositivos.

Es decir es el OS el que maneja los recursos.

-Virtualización del CPU

El OS 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 políticas establecidas por el OS y diferentes mecanismos.

-Virtualización de la Memoria

La memoria física no es más que un array de bytes, para poder leer de memoria o escribir en ella se necesita la dirección de donde hacerlo.

Los programas acceden a memoria todo el tiempo, ya sea por información/archivos o su propio código que está en memoria

Cada programa tiene su propio espacio direccionable de memoria virtual, virtualizada por el OS el cual dsp se encarga de mapearla a memoria física. Y los accesos a memoria de un programa no afectan(no deben por protección) a otros programas o al OS mismo.

-Concurrencia

Los problemas de concurrencia surgen con la ejecución de programas en simultáneo o programas de hilos múltiples. Requiere de la ejecución atómica de instrucciones(toda la ejecución de una sin interferencia).

-Persistencia

La información en RAM es volátil, se borra al apagar el equipo, el OS debe encargarse de la persistencia de la información. A través de hardware como discos, etc. y software como archivos,etc.

Los dispositivos de almacenamiento no son virtualizados para cada programa, sino que estos pueden acceder compartiendo información entre ellos.

-Metas de diseño

1)Facilidad de uso del sistema

2)Alto rendimiento(minimizar overheads)---> equilibrar costo/desempeño

3)Protección de aplicaciones(entre ellas y al OS de ellas)---> se usa el aislamiento de los proc unos de otros

4)El OS debe ser confiable, si el OS se detiene el equipo con el, no debe pasar.

5)Eficiencia energética y seguridad(contra aplicaciones malignas)

6)Protección, el que pueden hacer los proc debe estar limitado y el OS debe mantener control sobre la máquina(los programas de usuario corren en User mode, un modo limitado y el OS en kernel mode, un modo con total privilegio).

**Chapter 4 : The abstraction: The process**

process→ running program (proceso = programa en ejecución)

Ilusión de que hay muchos CPU → Virtualización del CPU

→ Time sharing of the CPU allows concurrent process

-Virtualización —> OS usa low-level machinery(mechanisms(mecanismos): protocolos para implementar una funcionalidad Ej context switch(cambio de contexto) (cambia el proc en ejec.)) y high-level intelligence(políticas: algoritmos que toman decisiones en base a los recursos que tienen, otros factores y prioridades)

-Process —> machine state? Que puede leer o actualizar cuando está corriendo? qué “maquinaria” necesita para su ejecución?

Por un lado memoria. Las instrucciones están en memoria, la data que el proc lee está en mem. —> la memoria a la que un proceso puede acceder (address space) es parte del proceso

Por otro lado registros. Muchas instrucciones los leen o actualizan( hay algunos registros especiales como el program counter, dice que instrucción se ejecutará a continuación)

Por otro lado y finalmente un proc accede a almacenamiento para asegurar la persistencia

-PROCES API

cualquier interfaz de procesos debe poder

Crear nuevos procesos

Destruir procesos en caso de que no terminen por sí mismos

Esperar, poder esperar la finalización de un proc

Control misceláneo: otros tipos de control como suspensión

Status: mostrar el status/info de un proc

-Process creation

Para correr un programa el OS primero debe cargar su su código y memoria estática en memoria, en el address space del proc

Hoy en día los OS cargan el programa de forma vaga, cargan código conforme se va necesitando (de disco)

Se debe proporcionar memoria para el programa(Stack(variables locales, parámetros) y Heap(información dinámica y variable en tamaño, estructuras de datos como listas ))

Iniciar los file descriptors(std in/ std out / error)

Luego el OS pasa el control al proceso creado y lo deja ejecutarse

-Process States

Estados en los que puede estar un proc

Running (en ejecución)

Ready (listo para ejecutarse)

Blocked (realizó alguna operación que hace que no pueda correr hasta que otro evento ocurra,Ej un I/O- luego pasa a ready)

Si un proc pasa de ready a running significa que fue scheduled/ de running a ready fue descheduled.

-Data Structures

Ej process list (como el administrador de tareas) los proc listos para correr y corriendo

Cosas que el OS tiene en cuenta de los proc

Register context —> cuando cambia un proc para correr otro, guarda el contenido de los registros, cuando vuelva restaura los contenidos.

Otro estado en el que puede quedar un proc es Zombie, término pero no fue cerrado/shut down

**Chapter 5 : interlude : Process API**

Process creation with system calls : fork() y exec() tmb puede usarse wait() para esperar que un proc creado termine

Fork() system call

crea un nuevo proc —> el proc creado es casi una copia exacta del proc donde fue llamada fork() (parent) y el proc creado (child) tiene una copia del address space pero es su propia memoria privada, sus registros, su PC, etc, no es una copia exacta. Su ejecución no es determinista, son dos proc diferentes en el scheduler

El scheduler determina que proc se ejecuta en cada momento dado

Wait() system call

puede usarla un parent para esperar al child (puede ser muy útil), cuando el hijo termina el padre continúa su ejecución

Exec() system call

útil cuando se quiere correr un programa que es diferente al programa desde el que se la llama

usa el nombre de un ejecutable argumentos t carga el código del ejecutable y sobreescribe el segmento de código actual, luego el OS corre ese programa

Porque es así?

La separación de fork() y exec() da la oportunidad de ejecutar de diversas formas y con varias funcionalidades en el medio, como redireccionar la salida de un comando/proceso

Process Control and Users

Hay muchas system calls para el usuario como kill(), las system calls son usadas para enviar **signals** a un proc, cómo pausarlo, terminarlo u otras instrucciones, pero quién puede enviar signals a un proc? Con la noción de usuario se limita quién gana control sobre este recurso del sistema y puede controlar sus propios procesos

**Chapter 6 Mechanism : Limit Direct Execution**

Para virtualizar el CPU, el OS necesita compartir el CPU físico entre varios trabajos simultáneos, la idea es correr uno un poco y cambiar a otro rápidamente generando la idea de procesamiento simultáneo. Con **Time Sharing** se alcanza la virtualización.

Hay varios desafios al lograr esto, por un lado la *performance,* virtualizar sin sobrecargar el sistema y *control*, como correr proc mientras mantenemos control del CPU.

-Basic Technique: Limited Direct Execution

Simplemente correr el programa directamente en el CPU, el OS hace los preparativos, pone el proc en una list proc(lista de procesos) y lo ejecuta, pero como vemos que no haga nada que no queramos hasta que el OS retome el control? Como el OS lo detendría para correr otro proc y lograr time sharing?

-Problem 1: restricted operations

Ventaja, ejecución rápida. Qué pasa con las operaciones restringidas durante la ejecución?

**user mode**, el código que corre en user mode tiene restricciones(como los accesos a I/O, hacerlo lleva a una excepción lo que haría que el OS mate el proc).

**kernel mode,** el modo en el que funciona el OS, no tiene restricciones, puede hacer operaciones privilegiadas

El modo de usuario cuenta con las System calls para requisitar operaciones que tiene restringidas. Estas instrucciones ejecutan un **trap**  que salta a kernel mode elevando el privilegio y realiza la instrucción si está permitida (por el OS). Al terminar realiza un **return from trap** y baja el nivel de privilegio volviendo a user mode. Antes de un trap se guardan los registros del proc que llamó el trap, en un per process(uno por cada proc) **kernel stack**, para su posterior restablecimiento al volver a user mode. Kernel debe verificar bien que código ejecuta en una trap.

Para esto kernel setea una **trap table**  al momento del booteo(cuando se enciende) y establece que correr si ocurren excepciones, le dice al hardware informandole la localización de estos **trap handlers**, normalmente en una sección especial.

Para especificar la system call cada una tiene un **system-call number,** el código de usuario señala cual quiere ejecutar y el kernel en el **trap handler** la verifica, si es válida la ejecuta. Esto ofrece protección para no darle control total al usuario( que no salte a una dirección de memoria no autorizada)

**LDE**(limited direct execution) en el booteo inicia la trap table y el CPU recuerda su localización. Después , cuando corre un proc, el kernel setea algunas cosas( como un nodo en el process list, allocating memory) antes de return from trap para ejecutar el proc cambiando el CPU a user mode. Luego si el proc quiere llamar a una system call trapea de vuelta al OS que maneja la llamada y luego vuelve con un return from trap al user mode

-Problem 2: switching between processes

Si un proc corre en el CPU el OS no está corriendo, como podría hacer algo?

**cooperative approach: wait for system calls**

El OS confía en el proc en ejecución, se da por sentado que si corre mucho tiempo cada cierto periodo liberaría el CPU para que el OS decida que correr, como por ejemplo con una System call, algunas system call tienen una **yield** sistem call para transferir el control al OS, si se hace algo no debido tmb se da control al OS.

**a non-cooperative approach: the OS takes control**

Con ayuda adicional del hardware puede retomar control del CPU. **Timer interrupt**, Un dispositivo que cada varios milisegundos realiza una interrupción, cuando lo hace un pre- configurado **interrupt handler** en el OS se ejecuta y el OS toma el control.

En el booteo el OS deja explicitado que código correr en una interrupción, tmb comienza el timer en el booteo.

El OS se encarga de salvar el estado del proc actual al momento de la interrupción para su posterior return-from-trap.

Saving and restoring context

Si cuando el OS toma control decide cambiar a otro proc, el OS ejecuta un código de bajo nivel, el **context switch**, el OS debe salvar los valores de los registros del prog. en ejecución(en el kernel stack del proc) y restaurar otros para el proc que pasará a ejecutarse(de su kernel stack)

(MOSTRADO CON PRECISIÓN EN PAG 10 CAP 6)

Si hay un timer interrupt el hardware guarda los registros, si hay un switch por parte del OS el software guarda/restaura los registros del kernel

-worried about concurrency?

Para que no ocurran interrupts simultáneos, se suelen parar las interrupciones mientras se está manejando una interrupción.Tmb hay **lock**s

**Chapter 7 Scheduling: Introduction**

Debemos entender las políticas de alto nivel que utiliza el scheduler del OS.

**scheduling policies**( también llamadas **disciplines**)

-Workload Assumptions

Antes de ver cómo se desarrolla una política hacemos algunas suposiciones sobre los proc que corren en el sistema(workload), estas serán irrealistas.

Suposiciones sobre los proc/jobs:

1. Todos duran lo mismo hasta acabar
2. Todos llegan al mismo tiempo
3. Una vez empiezan corren hasta acabar
4. Solo usan CPU(no I/O)
5. Sabemos el tiempo que le toma ejecutarse

-Scheduling Metrics

Métrica para medir los proc, su desempeño. **turnaround time**, el tiempo al que se completa el job menos su llegada al sistema

**T-turnaround** **= Tcompletion - Tarrival**

Por 2 Tarrival = 0, Tturnaround = Tcompletion, es una métrica de performance, otra métrica es **fairness**(que tan justo es), suele contrastar con performance.

-First In, First Out(FIFO) (también llamado First Come, First Served)

Es simple y fácil de implementar y funciona bien con nuestras suposiciones, pero si relajamos 1) empeora el Tturnaround, puede sufrir de **convoy effect**, es decir puede llegar un job más largo que los demás y ralentizar todo,con jobs más cortos en espera, ya que corre hasta acabar.

-Shortest Job First(SJF)

Siempre corre el trabajo más corto primero(por suposición 5) .Con esto el Tturnaround.

*Aside: hoy en día hay schedulers que son preemptive, pueden dar preferencia a otro proc y realizar un context switch.*

Si relajamos 2) y pueden llegar jobs en cualquier momento, si llega primero uno muy largo y es el único, será ejecutado porque es el más corto disponible, y si llega uno más corto deberá esperar a que finalice el primero, empeorando el Tturnaround.

-Shortest Time-to-Completion First (STCF)

también llamado PSJF(preemptive shortest job first)

Relajamos 3), los jobs no corren hasta acabar. El scheduler puede **preempt** un trabajo(darle prioridad), en los momentos donde el OS retoma el control del CPU(interrupts, syscalls, etc) el scheduler puede realizar un context switch. Cada vez que llega un job nuevo analiza cual conviene correr.

-Nueva métrica: Response Time

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

**Tresponse = Tfirstrun - Tarrival**

Primera ejecución - Momento de llegada.

STCF es malo en esto.

-Round Robin(RR)

Corre los proc durante un periodo de tiempo fijo hasta que acaben(este **time slice**  puede ser llamado **quantum**), cambia de job cada cierto tiempo fijo de entre los trabajos que hay en una cola hasta que acaben. Este time slice debe ser múltiplo del timer interrupt del sistema para que el OS puede tomar control y hacer el context switch

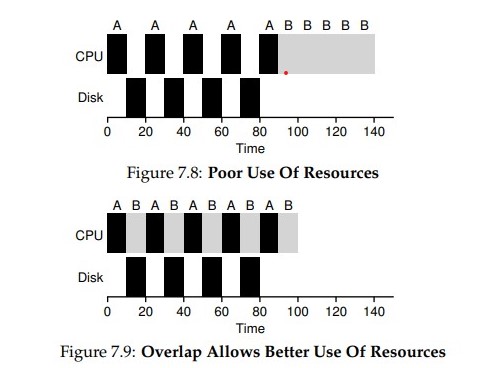
La duración del time slice es importante, si es muy corta es bueno para el Tresponse pero puede empeorar la performance al aumentar la cantidad de context switch mucho. Debe ser suficientemente larga para no empeorar la performance y corta como para un Tresponse aceptable.

RR es muy malo en Tturnaround. RR es **fair**(justa) esto hace que sea peor en Tturnaround, es un trade-off. Depende del objetivo que se tenga con el scheduler.

-Incorporando I/O

Relajamos 4), los proc piden I/O. Todos los programas usan I/O, un proc tampoco usa CPU mientras realiza un I/O y espera que este se complete( queda en estado Blocked), durante este tiempo el scheduler suele correr otro proc. Cuando el I/O termina el Scheduler también decide qué hacer, si lo deja es espera (Ready) o 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 entre programas se es más eficiente, tratando cada Pequeño tiempo de uso del CPU como un proc.



-No más oráculo

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

**Chapter 8 Scheduling: The Multi-Level Feedback Queue**

**MLFQ.** Por un lado optimiza *turnaround time* y tmb busca que sea interactivo, es decir minimizar el *response time*. Para lograr esto el scheduler debe “aprender” mientras el sistema corre las características de los proc.

-MLFQ: Basic Rules

Tiene diferentes colas(queues), cada una con un diferente nivel de prioridad, los proc en estado ready quedan en las diferentes colas y el MLFQ decide cual correr en base a la prioridad.

1)Si un proc A tiene más prioridad que proc B, corre A

2)Si A y B tienen la misma prioridad hace Round Robin entre los proc

La prioridad de los proc varía en base al *comportamiento observado* del proc. Si el proc normalmente suelta el CPU por I/O del teclado el MLFQ mantiene su prioridad alta viéndolo como un proc interactivo. En cambio frente a usos prolongados del CPU el MLFQ le baja prioridad. La MLFQ trata de aprender del comportamiento de los proc y predecir para el *futuro comportamiento.* Es decir la **prioridad cambia con el tiempo**

-Attempt 1: How to change Priority

Reglas:

3) Los nuevos proc en el sistema se les da max prioridad

4)a-Si un proc usa un time slice entero se le reduce la prioridad

4)b-Si suelta el cpu antes del time slice conserva prioridad

Ejemplos:



En estos casos el time slice es de 10 ms(milisegundos), luego baja la prioridad a menos que se libere el CPU

-Problemas con este MLFQ

Puede causar **starvation**(hambruna, el proc nunca llega a recibir CPU), si se juntan demasiado proc interactivos, combinados consumirian todo el CPU time y los log-running jobs starve (los proc largos no reciben CPU).

Otro problema es que un programa pensado para liberar el CPU justo antes de terminar el time slice se mantendría siempre en máxima prioridad, esto se llama **game the scheduler**(juega con el scheduler), podría monopolizar el CPU. Esto concierne a la seguridad, el scheduler debe ser a prueba de esto.

Además si un long-running job cambia en el tiempo de ejecución y se vuelve más interactivo igualmente se quedaría en la prioridad a la que haya bajado durante su largo uso del CPU.

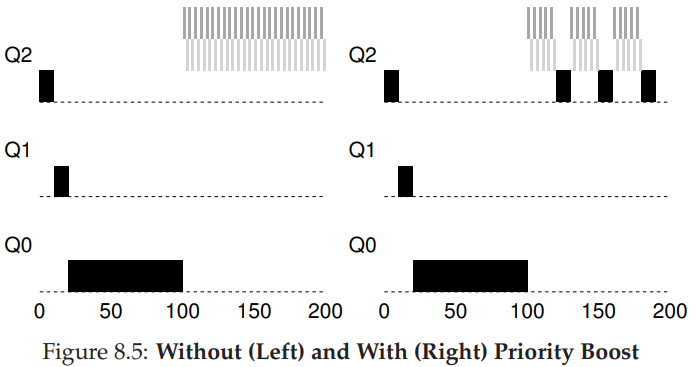
-Attempt 2: The Priority Boost

La idea sería periódicamente aumentar la prioridad de todos los proc (to **boost** the priority) poniéndolos en la cola más alta/ de mayor prioridad.

Regla 5) Cada cierto periodo S llevar todos los proc a la cola de mayor prioridad

Esto garantiza que los proc no se queden sin CPU time (they don’t **starve**). Y si un long-running job se vuelve interactivo esto ayudará a que al volver a la cola máxima el scheduler lo trate como tal.

Ejemplos:



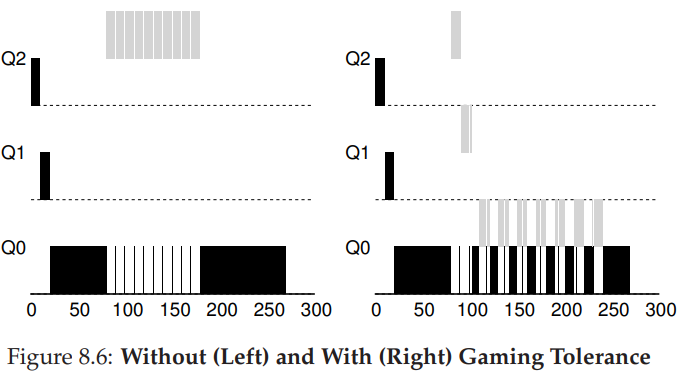
Hay un priority boost cada 50 ms, la duración de S debe ser bien elegida para que no haya starvation(no muy largo) y que los proc interactivos funcionen eficientemente(no muy corto).

-Attempt 3: Better Accounting

Cómo prevenir que el proc juegue con el scheduler? Solución: tener mejor contabilidad del uso del CPU en cada nivel de la MLFQ, se le da un tiempo a cada proc que no debe sobrepasar:

Regla 4 reescrita) Una vez que un proc uso su tiempo permitido a cierto nivel(sin importar cuantas veces libero el CPU), su prioridad baja.

Ejemplo:



-Arreglando otros problemas de la MLFQ

Cómo la parametrizamos? Cuántas colas debe haber? Time slice por cola? Duración de S(cada cuanto hay boost)?

Se suelen usar colas con diferentes time slices, más cortas cuanta mayor es la prioridad de la cola, entendiendo que a mayor prioridad proc más interactivos.

**Chapter 13 The abstraction: Address Spaces**

-Early Systems

En los primeros sistemas no se otorgaba mucha abstracción a los usuarios. El OS era un conjunto de rutinas(una librería) en memoria(en la dirección 0 por ejemplo), y un proc en memoria física.

-Multiprogramming and Time Sharing

Muchos proc corrían al mismo tiempo y el OS debía mediar(switchear entre los proc) para una mayor eficiencia en el uso del CPU por razones económicas, era muy caro había que usarlo bien.

La noción de **interactivity**(interactividad) se volvió importante al haber más usuarios de forma concurrente usando la máquina.

Una forma de time sharing era cambiar entre los procesos, en cada cambio guardar todo su estado y reg., hasta que todos terminen y luego almacenar todo de alguna forma(disco, etc.). Pero esto es muy lento y mal rendimiento cuanto más crece la memoria.

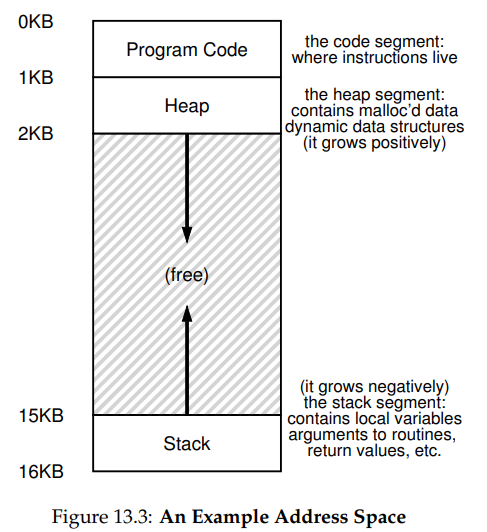
En mejor caso conviene dejar los proc en memoria y cambiar entre ellos sin guardar en disco cada vez. Al haber varios programas a la vez en memoria la **protección** se volvió importante. Los proc no debían leer o escribir en la memoria de otros proc.

-The address space

El OS debe crear una abstracción fácil de usar de la memoria física. Esta abstracción es el address space, y es lo que ve un proceso al funcionar sobre el sistema de memoria. Entender esto es clave para entender cómo se virtualiza la memoria.

El address space(espacio direccionable) de un proc contiene todo el estado de la memoria del programa en ejecución. El código mismo del programa reside en memoria, en este address space. Mientras el programa corre usa el **stack** para saber donde la llamada de la función y saber donde guardar variables locales, pasar parámetros y devolver valores de las rutinas. Finalmente el **heap** es usado para cosas guardadas dinámicamente(dynamically -allocated), memoria manejada por el usuario(como malloc en C). Hay otras cosas como variables estáticas, etc. pero por ahora pensemos en **code, stack** y **heap**.

Ejemplo:



El código es estático, no cambia => fácil de poner en memoria, luego las otras dos partes pueden crecer(y achicarse) mientras el programa corre. Al poner en forma opuesta el heap y stack podemos permitirles crecer, esto es una convención. (Luego al trabajar con múltiples hilos esta forma linda/simple de ponerlo no sirve más)

Esto es la abstracción que el OS le brinda al programa, en realidad el programa no está en la dirección 0 (el ejemplo) sino que está en una dirección arbitraria de la memoria física.

Cuando el OS hace esto virtualiza la memoria, cada proc cree que está en una dirección específica y que tiene una memoria potencialmente grande, lo que en realidad es diferente.

La dirección en la que el proc se ve a sí mismo es su **memoria virtual** en base a la cual hace requisitos al OS, el cual frente a estos debe traducir esta memoria a la memoria física real a la hora de responder los pedidos del programa, como cargar un archivo.

-Goals

El trabajo del OS es virtualizar la memoria. Para hacerlo bien hay algunos objetivos que cumplir, el más grande es **transparencia**(transparency), al contrario del sentido común se refiere a que el proc no se de cuenta de que corre en una memoria virtual(no el estado real de la memoria), cree que tiene su propia memoria física. El OS junto al Hardware crean esta ilusión.

Otra meta es la **eficiencia**, la virtualización debe ser lo más eficiente posible en términos de tiempo y espacio. Para esto el OS debe contar con/utilizar el hardware y características de este como la TLB(será visto dsp).

La tercera meta es **protección**, el OS debe proteger los procesos unos de otros así como proteger al OS de los procesos.

*ASIDE/de lado: Principio de aislamiento, si dos cosas están aisladas entre sí una puede fallar sin afectar la otra. Los OS tienden a aislar procs unos de los otros para que no interfieran entre ellos. Se puede aislar a través de la memoria.*

Un proc no debe acceder a cosas fuera de su address space, la protección nos permite dar la propiedad de aislamiento entre procesos, cada proc corre en su propio espacio. Esto se logra con varios mecanismos para virtualizar memoria y hardware y soporte del OS junto con importantes políticas sobre cómo manejar el espacio libre y que borrar de memoria cuando se necesita espacio.

**Chapter 14 Interludio: Memory API**

Este interludio es sobre interfaces de memoria (en sistemas UNIX).

Cómo asignar y manejar memoria?

-Tipos de memoria

Corriendo un programa de C hay dos tipos de memoria asignada. Primero el stack, las asignaciones y reasignaciones las maneja el compilador(por esto a veces se llama memoria automática). Luego de declarar por ej una función y sus variables, etc el compilador le asigna la mem necesaria y cuando no es más necesaria la desasigna/elimina.

Por esto para cosas que necesiten un poco más de permanencia se usa el heap donde las asignaciones y reasignaciones las realiza el usuario explícitamente, por esta naturaleza y usos varios esta memoria es un desafío para usuario y sistema.

-La llamada malloc()

Se le da un tamaño(en bytes) pidiendo dicho espacio en memoria heap y o lo logra y devuelve un puntero a dicha memoria o falla y devuelve NULL. Normalmente se utiliza sizeof() como operador para decir que tamaño requerimos en memoria.

-La llamada free()

Asignar memoria es lo fácil, por otro lado saber cuando, como e incluso si hace falta liberar memoria es lo difícil. Para liberarlo solo hace falta llamar a free() con un argumento, un puntero devuelto por malloc(), el tamaño de la memoria a liberar es buscada por la librería de asignación de memoria.

-Errores comunes

Hoy en día algunos lenguajes tienen un manejo de memoria automático al cual no hace falta señalarle que liberar, lo revisa solo. Pero en C no es así, errores:

-Olvidarse de asignar memoria

Muchas rutinas esperan ya tener memoria asignada al ser llamadas(ej: strcpy())

-NO asignar memoria suficiente

Algunas veces llamado **buffer overflow**. Normalmente se asigna la memoria justa, algunas veces malloc() deja un poco de margen para evitar errores pero cuando no es así un overflow de pequeños bytes puede ocurrir, algunas veces es inofensivo y otras puede ser muy dañino y crashear el programa.

-Olvidarse de inicializar la memoria signada

Si se llama a malloc() pero no se le asignan valores a la memoria asignada el programa eventualmente encuentra un **uninitialized read**(lee algo no inicializado) donde lee de heap alguna data de valor desconocido, causando comportamiento no previsto en el programa.

-Olvidarse de liberar memoria

Normalmente conocido como **memory leak**(pérdida de memoria), este problema es incluso mayor a largo plazo perdiendo memoria constantemente puede llevar a acabarse la memoria en aplicaciones continuas/long-running. Cuando ya no se necesita una memoria asignada debe ser liberada.

En programas cortos no es tanto problema, cuando este finalice toda la memoria del programa será liberada.

-Liberar memoria que todavía puede necesitarse

Si un programa libera memoria antes de usarla se queda con un puntero colgante, llegando subsecuentemente a crashear o sobreescribir memoria válida.

-Liberar memoria repetidamente

Los programas pueden liberar memoria más de una vez, conocido como **double free**. El resultado de esto es indefinido y puede generar crasheos.

-Llamar a free() incorrectamente

free() solo espera que se le pase como argumento un puntero devuelto por malloc(), con cualquier otro valor o argumento el free() es invalido.

Cabe añadir que malloc() y free() no son system calls, son parte de una librería de manejo de memoria.

**Chapter 15 Mecanismos: Address Translation(traducción de direcciones)**

En el desarrollo de la virtualización del CPU nos centramos en el mecanismo general **limited direct execution (LDE)**. La idea era que en mayor parte dejar el proc correr en el hardware y en ciertos puntos clave el Os se involucra y hace lo “correcto”, de esta forma el Os se asegura, con ayuda del hardware de mantener el control mientras trata de mantenerse fuera del camino del proc, creando así una buena virtualización del CPU.

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

Algo más que necesitaremos del sistema del sistema de memoria virtual(VM o MV) es flexibilidad, que los proc puedan usar su address space como quieran haciendo así el sistema más fácil para programar.

**¿Cómo virtualizar memoria eficiente y flexiblemente?**

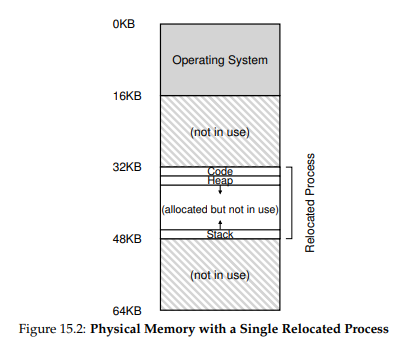
La idea general es algo llamado **hardware-based address translation**(traducción de direcciones en base a hardware) o solo **address translation** (básicamente la redirección que el OS hace entre la dirección virtual que el programa ve y la memoria física real), en cada referencia a memoria se hace una traducción de dirección por el hardware para redireccionar la referencia a memoria de la aplicación a su localización real en memoria.

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

De nuevo nuestro objetivo es crear la ilusión de que el programa tiene su propia memoria donde reside su código e información. Dejando atrás la fea verdad física de que muchos programas están compartiendo memoria al mismo tiempo mientras el OS cambia entre programas en ejecución.

-Suposiciones

Por ahora supondremos que el address space del usuario está localizada de forma continua en memoria física. Tmb , por simplicidad, asumimos que el tamaño del address space no es tan grande, específicamente es menor que la memoria física. Por último, todos los address space tienen el mismo tamaño( a medida que avancemos relajaremos estas suposiciones).



Para virtualizar la memoria el OS debe poner este programa en otro lugar, no en dirección 0, como puede hacerlo de forma transparente(que el proc no se de cuenta)?

En este ejemplo se puede ver un programa cargado en memoria, para dicho programa su dirección inicial es 0(en su address space) y no 32KB, y crece hasta un máximo de 16KB, cualquier referencia a memoria que el programa haga debe estar dentro de estos límites. El primer bloque de memoria es para el OS mismo.

-Reubicación dinámica(Basada en hardware)

Usaremos las ideas de **base and bounds** y **dynamic relocation**. Específicamente necesitaremos 2 registros de hardware en cada CPU, uno llamado **base** register y el otro **bounds**(algunas veces llamado registro límite). Este par de registros nos permite direccionar espacio en cualquier lugar de la memoria física y al mismo tiempo asegurar que el programa solo accede a su propio address space.

Cada programa cree estar en la dirección 0 pero el OS 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 de la siguiente forma:

**physical address = virtual address + base**

Cada referencia a memoria creada por el proc es una **dirección virtual**, el hardware amade los contenidos de la base a la dirección y el resultado es la **dirección física**  que puede ser usada por el sistema de memoria.

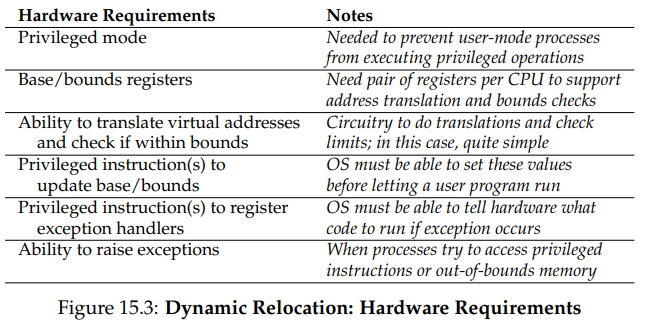
El transformar una dirección virtual en una física es a lo que nos referimos con **address translation**(traducción de dirección). El hardware toma la dirección virtual que el proc cree referenciar y la transforma en la memoria física donde está realmente la información. Debido a que esto ocurre durante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

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

Base y Bounds son estructuras de hardware mantenidas en el chip, un par por CPU. Algunas veces llamados **MMU(memory management unit)**, cuantas más técnicas agreguemos más contenido agregaremos a él MMU.

-Apoyo del Hardware: resumen

Aparte del kernel mode donde el OS tiene control y los registros Base y Bounds, el hardware debe proveer instrucción especiales para modificar Base y Bounds, para que el OS los pueda cambiar cuando otro proc corre. Estas instrucciones son privilegiadas, solo se pueden usar en kernel mode.



El CPU debe poder generar excepciones cuando un programa de usuario trata de acceder a memoria ilegal/ fuera de su address space, entonces debe parar el proc y el OS debe tomar control y correr el **exception handler**. Lo mismo si el proc trata de hacer una instrucción privilegiada.

-Problemas del OS

El hardware da herramientas, el OS debe manejarlas para lograr la virtualización de la memoria.

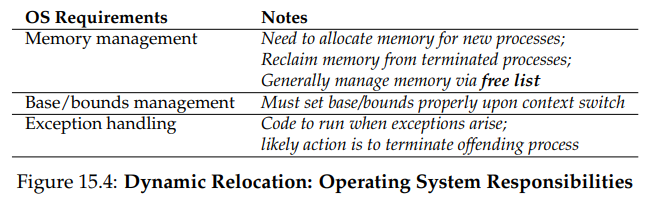
Primero, el OS debe encontrar espacio para el address space de un proc cuando este es creado(damos por hecho que entra en la memoria por nuestra suposición). El OS tiene que buscar la estructura de datos(**free list** normalmente), buscar espacio y asignarlo.

Luego, cuando un proc termina( por sí mismo/bien o la fuerza/ por el OS) el OS debe reclamar la memoria para su posterior uso por otros proc, agrega dicho espacio a la **free list**.

Tmb debe guardar y restaurar los registros Base y Bounds en cada context switch, guardar sus valores en memoria, en alguna estructura por cada proc, como la estructura del proc mismo(**process structure** o **process control block**), y cuando los restaura debe pasarle dichos valores al CPU.

Cuando un proc para el OS puede mover su address space en memoria, debe sacarlo del scheduler, actualizar la dirección de los registro Base y Bounds y listo.

El OS debe proveer **exception handlers**(manejo de excepciones), deben ser instalados en el momento de booteo, estos deben manejar acceso a memoria ilegal e intentos de uso de instrucciones privilegiadas.



El uso de Base y Bounds ofrece protección, un objetivo fundamental del OS, pero la **reubicación dinámica** tiene sus deficiencias. Por un lado en el address space de un proc, si el Heap y Stack no se usan con eficiencia el espacio entre ellos sin usar se desperdicia, esto normalmente se llama  **internal fragmentation**(fragmentación interna)---> el espacio dentro de memoria asignada no se usa=> se desperdicia.

**Chapter 16 Segmentation(segmentación)**

Con el modelo de manejo de mem del cal anterior se desperdicia el espacio libre entre Stack y Heap en cada proc. A su vez hace difícil el correr un programa cuyo address space no entra completo en memoria. Es decir, Base y Bounds no es tan flexible como necesitamos.

-Segmentation: Generalized Base/Bounds

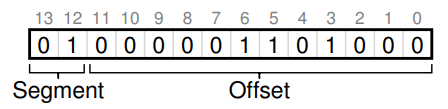
Para solucionar el problema señalado surge la segmentación, en vez de tener un par de registros Base/Bounds por MMU porque no usar un par por segmento lógico de address space( código, stack y heap). La segmentación le permite al OS poner cada bloque en mem física no necesariamente juntos y así evitar usar memoria física en address space sin usar.

Para su uso se requiere soporte del Hardware en forma de 3 pares de registros Base/Bounds por proc para los bloques señalados. Utiliza el mismo mecanismo que antes para verificar que la referencia a memoria es válida pero con cada bloque y sus reg.

-A que segmento nos referimos?

Un enfoque común, tmb llamado explícito, es cortar el address space en segmentos basándose en los bits más significativos de la dirección virtual. Para 3 segmentos necesitamos 2 bits para referenciarlos.

Ej:



Podemos decir que 00 es el código, 01 heap y 10 stack. Luego se utilizan los 12 bits de Offset, se les suma la Base y se consigue la memoria física a la cual se refiere. Además si el offset es mayor que los Bounds ya se identifica como acceso a mem ilegal/ fuera de su address space.

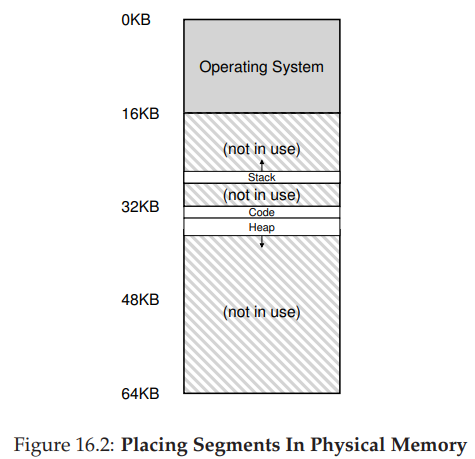
Con dos bits podriamos redireccionar hasta 4 segmentos, es decir estamos desperdiciando uno, para evitar esto algunos sistemas ponen el código en el mismo bloque que el heap redireccionando así solo dos bloques(code/heap y stack) y usando un solo bit.

Otro problema es que al tomar bits de la dirección se limita el tamaño máximo del address space del segmento, si el bloque quiere crecer más allá de su capacidad está sin suerte.

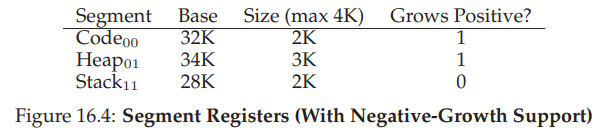
Hay otras maneras para que el Hardware identifique el bloque al que pertenece una dirección. Con un enfoque implícito, el hardware determina el segmento notando cómo se formó la dirección. Por ej, si la dire se formó desde el program counter está en el bloque de código, si la dirección se basa fuera del stack, debe estar en el bloque stack, cualquier otra es el heap.

-Qué pasa con el Stack?

Ej:



En el diagrama vemos que el stack crece hacía atrás, por lo tanto su traducción de dirección de mem es diferente. Hace falta soporte del hardware, aparte de los registros Base/Bound por bloque necesitamos saber en qué sentido crece. Por ejemplo señalandolo con un bit.



Luego de tomar los bits de bloque y de crecimiento, si crece de forma negativa el hardware deberá realizar una traducción en la que le resta al offset el valor total del address space y se queda con un valor negativo, sumamos este a la base y ya tenemos la mem física, pero al chequear los Bounds debe tomarle su valor absoluto al offset-address space.

-Support for Sharing

Los diseñadores de sistemas vieron nuevos mecanismo para hacer más eficiente el suo de la memoria, como compartir ciertos segmentos de memoria( con un poco más de soporte por parte del hardware). En particular el código es algo bastante común incluso hoy en día.

El soporte por parte del hardware es en forma de **bits de protección**. Añadir algunos bits por segmento indicando los permisos sobre esté como si un programa puede leer o escribir en un segmento, o ejecutar código que se encuentra en esté. Por ejemplo seteandolo en read-only(solo lectura) se puede compartir código con muchos proc. El proc cree que accede a su propia mem privada pero el OS secretamente le da acceso a memoria no modificable y así se preserva la transparencia.

Ahora el hardware aparte de si el acceso es en memoria legítima tmb debe chequear que el tipo de acceso está permitido(read,write, execute), sino trap handler.

-Fine-grained vs. Coarse-grained Segmentation

(Grano fino vs grueso)

La segmentación de la que venimos hablando es de pocos bloques, es decir de coarse-grained segmentation, corta el address space en pedazos relativamente grandes. En sistemas más tempranos se daba soporte a la creación de muchos segmentos más pequeños permitiendo un uso más flexible de la memoria, pero esto requiere más soporte del hardware como un **segment table** para seguir la pista de todos los segmentos. Fine grained segmentation le da más info al OS sobre el uso de la memoria, lo que le permite usarla más eficazmente.

-OS support

La segmentación le da nuevo trabajo al OS. Qué debe hacer frente a un context switch? Debe guardar los registros Base/Bounds de cada segmento, y debe restaurarlos correctamente antes de que el proc corra de nuevo.

Qué hace cuando los segmentos crecen(o se achican)? En caso de necesitar espacio debe pedirse con una syscall, luego el OS le asignará el espacio(y actualizará los reg) o se lo negara en caso de no haber más memoria física disponible.

Lo más importante al manejar el espacio libre en mem física para el OS es ser capaz de encontrar memoria física( para los segmentos requeridos cuando un proc se crea por ej). Un problema suele ser que la memoria física rápidamente se llena de pequeños agujeros de espacio libre y es difícil asignar nuevos segmentos o hacer crecer existentes. Este problema se llama **fragmentación externa**(external fragmentation).

Una solución sería compactar la memoria física ocupada reorganizando los segmentos existentes. Pero compactar es “caro” en cuanto a recursos, requiere frenar los proc en ejecución y un buen tiempo de CPU. Soluciona la asignación de nuevos bloques pero dificulta el crecimiento de los existentes. Es imposible que no haya **external fragmentation** pero hay algoritmos para minimizarla, como **best-fit**(guarda una lista de los espacios libres y asigna el más indicado), **worst-fit**, **first-fit** y más complejos como **buddy algorithm**.

**Chapter 17: Free-Space Management**

Antes de seguir con la virtualización de memoria veremos Manejo del espacio libre. Cuando veamos **paginación** veremos que el manejo del espacio libre es fácil cuando el espacio está dividido en unidades de espacio constante y solo hay que seguirle la pista a estas unidades, si se solicita espacio solo entregar la primera entrada libre. Pero se vuelve difícil cuando tratamos con bloques de espacio libre de diferentes tamaños.

-Suposiciones

Damos por hecho que contamos con una interfaz como la provista por malloc() y free(). específicamente malloc(size\_of size)) toma un solo parámetro size y devuelve un puntero a la región donde está la memoria asignada de tamaño size o mayor y free() puede tomar dicho puntero y liberar la reg a la que apunta. La librería debe interpretar por sí misma el tamaño del bloque a liberar(no es un parámetro).

El espacio que maneja esta librería es el heap y la estructura genérica que maneja el free space en esta es la **free list**, esta contiene una referencia a todos los bloques de espacio libre en la región que maneja de mem.

Asumimos que el principal problema a resolver es external fragmentation, tmb podría estar el problema de fragmentación interna con los allocs. Una vez se llama a malloc() con memoria dentro del heap esta no puede ser tocada por la librería hasta que se use free() ===> no es posible compactar el espacio libre(porque no podemos mover el bloque ya asignado).

Pero compactación puede ser usada por el OS para combatir la fragmentación de usar segmentación.

Por simplicidad asumimos que el allocator maneja una región de bytes continua( no va a crecer).

-Mecanismos de bajo nivel

Primero veremos el mecanismo de los allocators, que es splitting y coalescing, cómo seguir la pista del tamaño de las regiones asignadas rápido y fácil. Y cómo construir una free list.

**Splitting and Coalescing**

Una free list contiene un conjunto de elem. que describen el espacio libre que queda en la heap. Si se solicita un espacio mayor a los bloques libres que hay(puede que el espacio esté disponible pero no continuó en un bloque) el pedido falla.

En este caso el allocator hará **splitting:** encontrará un bloque de memoria que pueda satisfacer el pedido y lo partirá en dos. El primer bloque lo devolverá al que hizo el pedido y el segundo se mantendrá en la lista(se necesitan 3 bytes y tengo un bloque de 10, lo parte y usa uno solo de 5). Esto se suele usar cuando el pedido de espacio es menor que cualquier bloque de memoria libre.**Coalescing** es combinar los bloques de espacio libre continuos en uno solo, se hace cuando se libera espacio.

**Tracking the Size Of Allocated Regions**

A free() no se le especifica el tamaño de la reg a liberar, la librería es capaz de darse cuenta sola y recuperar la memoria. Para lograr esto la mayoría de allocators guardan un poco de info en un **header** block normalmente guardado justo antes de que comience el bloque de mem asignado.

El header contiene mínimamente el tamaño del bloque y tal vez punteros para acelerar la recuperación de memoria, un “número mágico” para verificar la integridad y otra info.

Con esto se puede calcular rápidamente el tamaño del espacio a liberar, la región que queda libre es del tamaño de el espacio solicitado + el del header (cuando se solicitan N bytes el allocator busca N bytes + tam del header que usará para rastrear el tam del bloque).

**Embedding a Free List**(incrustación en una lista de espacios libres)

La free list se encuentra dentro del espacio libre mismo. Cuando se solicita espacio, se asigna al comienzo del espacio disponible, con su header para el posterior uso de free() y el espacio disponible queda abajo, así sucesivamente, si se libera un bloque entre otros bloques en uso queda un hueco de mem libre y cuando se liberan todos queda el espacio libre fragmentado a menos que se haga coalescing.

**Growing the heap**

Los allocators tradicionales empiezan con una heap chica y piden espacio al OS a medida que se necesita(con una syscall), este, si es posible, le da el espacio y les devuelve la dirección del nuevo final de la heap.

-Estrategias básicas

Con los mecanismos listos solo falta ver cómo usarlos. El allocator ideal es ambos rápido y minimiza fragmentaciones, no hay un único enfoque mejor que los otros pero hay varias estrategias para lograr este objetivo.

1)**Best fit**

Se busca a través de la free list y se usa el más cercano al tamaño solicitado(del mismo tamaño o más grande, se selecciona al más pequeño de los candidatos). Pero una implementación ingenua puede costar mucho en performance al hacer una búsqueda extensiva en toda la free list.

2)**Worst fit**

Es el opuesto, se busca el bloque más grande, se usa el espacio necesario de ese bloque y se devuelve lo restante a la free list. Este enfoque trata de dejar muchos bloques grandes en vez de muchos muy pequeños. Este enfoque es malo, lleva a fragmentación y overheads y sufre de lo mismo que best fit, búsquedas intensivas en la free list.

3)**First fit**

Usa el primer bloque que es lo suficientemente grande para cumplir con lo solicitado, buscando desde el comienzo cada vez, su ventaja es la velocidad pero puede “contaminar” el comienzo de la free list con objetos pequeños. Con esto el allocator tiene más dificultades para manejar la free list. Un enfoque es usar address based ordering, manteniendo la lista ordenada por address del espacio libre hacer coalescing es más fácil y reduce la fragmentación.

4)**Next fit**

Parecido al anterior pero siempre busca el primero que sirva a partir de donde se quedó en la última búsqueda(señala dicha posición con un puntero). Esto evita el amontonamiento al comienzo de la free list.

-Otros enfoques

**Segregated Lists**

Idea: si una aplicación una(o varias) petición “popular”(recurrente petición de tamaño igual/similar), mantiene una lista separada para el manejo de objetos de ese tamaño, las demás peticiones van a un allocator más general.

Al tener un bloque para objetos de cierto tamaño la fragmentación es menor y los pedidos de dicho tamaño se satisfacen más rápido.

Esto trae nuevos problemas. De qué tamaño es el bloque que se guarda para estos pedidos genéricos? Un allocator en particular, el **slab allocator** lo maneja de la siguiente forma, cuando kernel bootea, le asigna un número de **object caches** para objetos del kernel que se solicitan seguido, el object caches son segregated lists de un tamaño dado y sirven para asignar memoria y liberarla rápido. Si a un cache le falta espacio pide algunos **slabs(**bloques pequeños) más de memoria. Cuando ya no se necesita/usa uno de estos caches el allocator recupera la memoria. Este allocator tmb mantiene los free object de las listas en un estado pre inicializado.

**Buddy Allocation**

Binary buddy allocator busca hacer simple el coalescing. La memoria libre es vista como un gran espacio de tamaño 2^N. Cuando se solicita memoria solo divide el espacio libre por 2 hasta encontrar un espacio que satisfaga el pedido, esto puede causar fragmentación interna, ya que solo se pueden dar espacios que son potencias de 2.

Cuando un bloque se libera, digamos de 8k, chequea que su “buddy” de 8k esté libre, si lo está los los combina en uno de 16k y así recursivamente hasta que hizo un coalescing de todo o encontró un “buddy” en uso.

-Otras ideas

Un problema general es la falta de escala, La búsqueda en una free list puede ser lenta y costosa. Frente a esto se hicieron varias cosas que veremos más adelante.

**Chapter 18 Paginación: Introducción**

Paging

Al cortar el espacio disponible en bloques de tam diverso como en segmentación acarrean dificultades como la fragmentación, lo que dificulta la asignación de espacio mientras más espacio se ocupa.

Otro enfoque sería cortar el espacio en partes iguales de un tamaño estándar. En memoria virtual esto se llama **paginación(paging)**, cada unidad es llamada una **página(page)**. Y vemos la memoria física como un array de de slots del mismo tamaño a los que llamamos **marco de página(page frame).**

Cómo virtualizar la memoria con páginas?

-Descripción general

La Paginación tiene varias ventajas comparado con los enfoques anteriores. La mejora más sustancial sería la flexibilidad. Con un sistema de paginación bien desarrollado el sistema sería capaz de apoyar la abstracción del address space eficientemente, más allá de cómo el proceso use el address space. Por ejemplo, no asumiremos la dirección de crecimiento del heap/stack o cómo son usados.

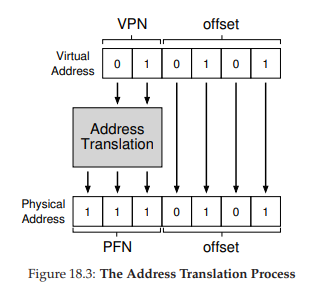
Otra ventaja es la simplicidad que permite la paginación al manejar espacio libre. Solo hace falta otorgar la cantidad de páginas necesaria.

Para mantener un registro de donde cada página virtual del address space está situada en memoria física, el OS normalmente guarda una información de cada proceso en una estructura conocida como **page table**(tabla de páginas). El rol más importante de la page table es almacenar las **address translations** de cada página virtual del address space.

El OS mantiene una page table por proc?

Para traducir una dirección virtual generada por un proc primero debemos dividir en 2 componentes la dirección: el **virtual page number**(VPN, número de pag virtual) y el **offset** de la página. Cuando un proc genera una dirección virtual el OS y el hardware combinados deben traducirlo a una dirección física.

Con el número de página virtual podemos indexar la page table y encontrar en cual marco físico(physical frame) reside la pag. Solo hay que reemplazar el VPN por el PFN(physical frame number) y está traducido.



El offset no se modifica, ya que solo señala que byte, dentro de la pag, nos interesa/queremos.

-Dónde se almacenan las Page Tables?

Las page table son terriblemente largas, tienen muchas **page table entry**(ies)(**PTE**) y hay muchas page table. Como las page table son tan grandes no tenemos ningún chip especial de hardware en el MMU para almacenar la page table de los procesos en ejecución. Guardamos la page table de cada proc en memoria en algún lugar, por ahora supongamos que están en memoria física.

-Qué hay en una page table?

Hablemos de su organización. Es una estructura de datos usada para mapear las direcciones virtuales a direcciones físicas. La más simple es llamada **linear page table**, la cual es solo un array donde se indexa con el VPN(núm. virtual de la pag.) y se busca el PTE(entrada en la page table) en dicha indexación para encontrar el PFN(frame físico de la pag.).

Luego cada PTE tiene algunos bits que vale la pena comprender hasta cierto nivel. Un **valid bit**, señala la validez de la traducción en esa entry, si se accede a una inválida una trap le dará control al OS. A su vez el valid bit es crucial para dar soporte a un address space escaso, simplemente marcando todas las páginas sin usar en el address space como inválidas, salvandonos de asignarles memoria y salvando espacio.

También hay **protection bits**, que indican si una página puede ser leída, escrita o ejecutada. Si se intenta acceder de una forma no permitida tmb saltara una trap que dará control al OS. Entre otros bits estan: **present bit** indica si se encuentra en mem física o en disco(veremos más esto más adelante), **dirty bit** indica si una pag a sido modificada desde que fue traída a mem, **reference bit** es usado para ver si una pag a sido accedida(útil para determinar páginas populares que deben ser mantenidas en mem).

-Paginación tmb es muy lento

Como son tan grandes la page table pueden ralentizar el sistema. Paginación requiere que se realice un referencia a memoria extra para primero buscar la traducción de la page table(de virtual a física, de la VPN -> PTE----------> PFN), esto es mucho trabajo y costoso, lo que ralentiza el proceso.

Sin un hardware y OS preparados/diseñados bien las page tables ralentizan demasiado el sistema.

**Chapter 19 Paginación: Traducciones más rápidas(TLBs)**

Paginación requiere un gran monto de mapeo de información. El mapeo de información normalmente está almacenado en memoria física, paginación requiere una búsqueda extra para cada dirección virtual generada por los proc. Ir a memoria por la traducción antes de cada instrucción es muy lento.

Cómo aumentar la velocidad de la Traducción de las direcciones?

Para aumentar la velocidad, el OS necesita ayuda del hardware, añadiremos lo que es llamado **translation-lookaside buffer** o **TLB**. Una TLB es parte del chip **memory-management unit** (**MMU**), es un **cache** de traducción de direcciones virtuales a físicas, por esto lo llamamos **address-translation cache**. Cada vez que hay una referencia virtual a memoria, el hardware primero mira la TLB para ver si la traducción está ahí, si está la traducción es realizada rápidamente sin necesidad de acceder a la Page Table. Debido a su enorme impacto TLBs hacen posible la memoria virtual.

-Algoritmo básico de la TLB

Supongamos una page table lineal y un TLB manejado por hardware. El algoritmo sigue de la siguiente manera:

Primero, extrae el VPN(virtual page number) de la virtual address y se fija si el TLB tiene la traducción para este VPN, si lo tiene, tenemos un **TLB hit**(tiene la traducción). Ahora podemos obtener el PFN(physical frame number) de la TLB entry, concatenarlo al offset de la dirección virtual original y ya tenemos la dirección física deseada para acceder a memoria(mientras que los chequeos de protección no fallen).

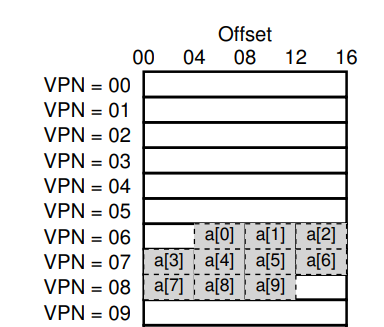
Si es **TLB miss**(no tiene la traducción), el hardware accede a la page table para encontrar la traducción y, asumiendo que la dire virtual es válida y accesible, la sube a la TLB. La prox que se busque esa traducción va a ser TLB hit y se ejecutará rápido.

El TLB está construido pensando en que el caso común es un hit, los miss ralentizan el proceso ya que requieren buscar en la page table. Por esto debemos intentar evitar los miss lo máximo posible.

-Accediendo a un Array

Digamos un array de números y un loop que solo le suma 1 a cada elem del array del primero al último, si en cada página de la page table tenemos varios elementos del array, el primer elemento al que queramos acceder de la página va a ser un miss, es decir el TLB va a tener que buscar la traducción, pero luego hará hit con todos los elem que estén en la misma página, mejorando drásticamente el desempeño que si se buscara la traducción de la referencia virtual para cada elem.

Ej: a(0) es miss pero de 1-2 es hit, a(3) es miss pero de 4-6 es hit, etc.



Es decir, la TLB se beneficia de la **spatial locality**, los elementos están cerca unos de otros. Si el programa se volviera ejecutar rápidamente, serían todos hits, la TLB en este caso se beneficiaría de **temporal locality**, la rápida re-referenciación a items en memoria en el tiempo.

-Quién maneja el TLB miss?

En sistemas más antiguos era manejado por el hardware, por un set de instrucciones complejas(**CISC**, complex-instruction set computers). El hardware debía saber dónde estaba la page table en memoria( con un **page table base register)** así como su formato exacto, en un miss, el hardware recorrería la page table, encontraría la PTE y extraería la traducción.

En sistemas modernos lo maneja el software, **RISC**(reduced-instruction set computers) tiene lo que se conoce como **software-managed TLB,** en un miss el hardware solo levanta una excepción que pausa la corriente de instrucciones, eleva el privilegio a modo kernel y salta al **trap handler**(un código dentro del OS), buscará la traducción en en la page table, y actualizará la TLB con instrucciones privilegiadas, luego volverá de la trap y el hardware reintenta la ejecución.

El return from trap luego de un TLB miss debe volver a ejecutar la instrucción que causó el miss para esta vez dar hit, no como el el return from trap normal el cual ejecuta la instrucción siguiente( el PC, program counter register, es diferente en cada caso).

El OS debe tener cuidado de no causar un loop de TLB misses, pero para esto tiene diversas estrategias que puede usar, esta es la principal ventaja del TLB miss manejado por software, su flexibilidad. Puede implementar cualquier estructura de datos que necesite sin necesitar un cambio en el hardware.

*ASIDE:*

*TLB valid bit es diferente a Page Table valid bit*

*En la page table, cuando una page table entry está marcada como inválida significa que no fue asignada por el proc, por lo que no debería ser usada por un programa que funcione correctamente.*

*En la TLB el valid bit señala si la TLB entry tiene una traducción válida o no.*

-Contenidos de la TLB, qué hay ahí?

Una TLB típica tiene 32,64 o 128 entradas y es lo que se llama **fully associative**. Básicamente significa que cualquier traducción puede estar en cualquier lado de la TLB, el hardware buscará la traducción. La TLB se ve algasí:

VPN / PFN / other bits

La TLB comúnmente tiene un **valid bit**, dice si una entrada tiene una traducción válida o no. Tmb **protection bits**, que determinan si se puede acceder a una pag(read, write, execute). También puede haber otros campos como **address space identifier**, **dirty bit** y más.

-TLB Issue: Context Switches

Las traducciones que hay en la TLB sólo sirven para el proceso en ejecución, por esto al hacer un context switch el hardware o el OS(o ambos) deben tener cuidado de no usar dichas traducciones con el nuevo proc que se ponga a ejecutarse.

Un enfoque frente a este problema es borrar el contenido de la TLB(**to flush the TLB**), ya sea por hardware(cuando cambia el page table register) o el OS(una instrucción privilegiada), simplemente setear todos los bits en 0. Lo malo es que si el cambio entre proc es frecuente el costo de esto será alto, teniendo que buscar las traducciones cada vez. Frente a esto algunos sistemas usan soporte del hardware añadiendo un **address space identifier(ASID)**, un campo en la TLB, con esto la TLB puede mantener traducciones de diferentes proc al mismo tiempo sin confusión sobre a qué proc pertenecen.

-Issue: Replacement Policy

Hay que tener en cuenta el **cache replacement**. Cuando estamos instalando una nueva entrada en la TLB, si hay que reemplazar una vieja, cual reemplazamos? Un enfoque es borrar la que más tiempo tiene sin usarse. Otro es borrar una random.

**Chapter 20 Paginación: Tablas más pequeñas**

Un problema antes mencionado es que las Page Tables ocupan mucho espacio en memoria, aparte de que cada proc tiene si page table, y en el sistema corren muchos procesos, por ende muchas page table. Ahora

Cómo achicar las page table?

-Solución simple: Páginas más grandes

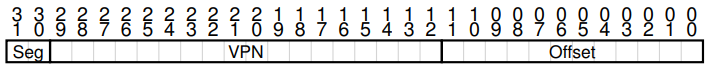
Un problema de esta solución es que el espacio que liberaron las page table, ahora lo ocupan las páginas. Por otro lado el mayor problema es el desperdicio de espacio dentro de las páginas llevando a **fragmentación interna**. Un proc puede usar una página pero si usa solo un pequeño pedazo de esta el resto se desperdicia.

-Enfoque híbrido: paginación y segmentos

Este enfoque plantea la idea de tener una page table por cada segmento del programa (code, heap, stack) en vez de una sola para todo el address space, lo que significaria páginas sin usar marcadas como inválidas.

Usaremos el registro Base de segmentación para señalar dónde está la page table de ese segmento en memoria física. Y el registro Bounds para señalar el final de la misma. En los bits de la dirección virtual debemos usar 2 bits para señalar a qué segmento nos referimos(01 para code por ejemplo), y el resto para el VPM y el Offset. Ahora cada proc tiene 3 page table y por lo tanto 3 pares de registros Base/Bounds que deben ser guardados y restaurados en un Context switch.

Ej:



En caso de TLB miss(asumimos hardware managed TLB miss), el hardware utiliza los bits de segmento(Seg) para determinar qué par de Base/Bounds usar, luego el hardware toma la memoria física (de la page table) y realiza la traducción. La principal diferencia de este enfoque híbrido es el uso de **Bounds** para determinar el final de la Page Table, en la cual solo estarán las páginas en uso y luego la pag límite, el cual limita que acceso es válido o no. Salvando espacio de una page table más grande con entradas sin usar marcadas como inválidas.

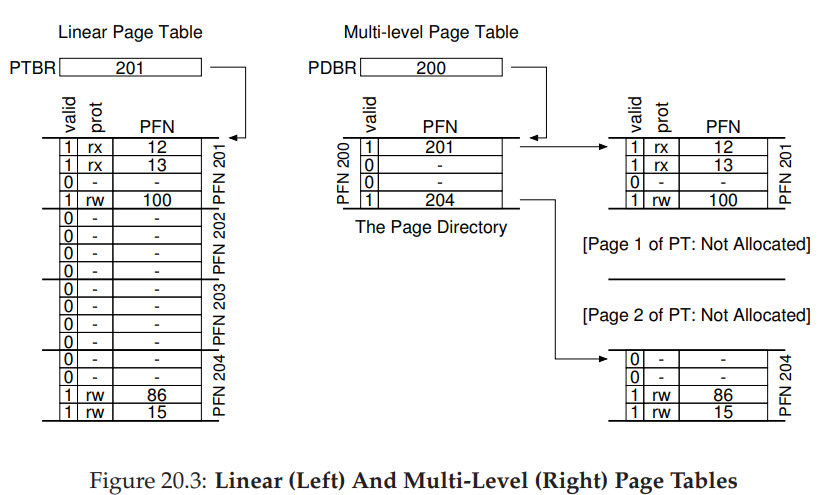
Pero no es perfecto, el uso de segmentación da por hecho cierto patrón del uso del address space, si tenemos un heap grande pero con pequeños bloques en uso separados terminamos desperdiciando espacio, a su vez este enfoque causa **fragmentación externa**. Las page table ahora son de tamaño arbitrario, dificultando la tarea de encontrar lugar para almacenarlas.

-Page Tables de nivel-múltiple

No usa segmentación pero busca solucionar el mismo problema, el espacio desperdiciado por páginas inválidas/sin asignar en la page table. Este enfoque convierte la Page Table lineal en algo como un árbol.

La idea básica es cortar la Page Table en unidades del tamaño de una página, si una PTE(page table entry) entera está sin usar/inválida, no allocar esa página para nada.

Para seguir la pista de qué página es válida( y si lo es donde está en mem) utiliza una estructura, el **page directory**(directorio de páginas). Esto puede decir donde una página de la page table está o si la página entera de la page table no contiene páginas válidas.



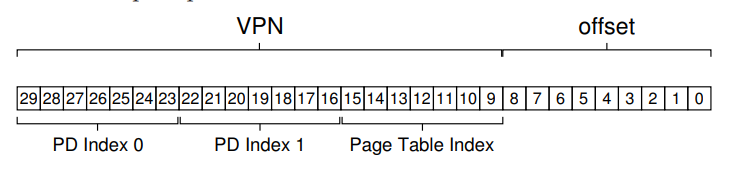
En la imagen se ve que solo la PTE 0 y 3 están en uso. La Multilevel page table lo que hace es eliminar esas regiones vacías de la page table sin usar que en una page table lineal ocupan memoria aunque esten vacias. Esta page table en dos niveles consiste de un número de **page directory entries(PDE)**(entradas del directorio de páginas). Cada PDE tiene al menos un **valid bit** y un **page frame number**(PNF), similar a PTE. El significado del 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.

Si está bien construido, cada porción de la page table entra bien en una página, haciendo fácil para el OS obtener la siguiente página libre cuando necesite hacer crecer una page table (Manejo de memoria más fácil).

Este enfoque añade un **nivel de indireccionalidad** a través del page directory, que señala a partes de la page table, esto permite poner las page tables pages donde queramos en mem física.Pero el costo es que frente a un TLB miss se deben hacer dos cargas antes de obtener la traducción, el page directory y luego la page table page. Esto es un ejemplo de trade off, ganamos espacio pero perdemos más tiempo en un TLB miss.

-Más de dos niveles

Cuando los bits de indexación del Page Directory son muchos, lo que no nos permite que cada parte de la multi level page table entre en una página se puede añadir otro nivel en la forma de otro page directory por encima del primero, partiendo los bits de indexación. 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.



En el ejemplo se usan pag de 512 bytes y PTEs de 4 bytes.

Cabe añadir que el aumentar los niveles también aumenta el costo de la búsqueda de la traducción en caso de un TLB miss.

-Swapping the Page Tables to Disk

Hasta ahora suponíamos que las page table estaban en memoria física del kernel. Incluso con todos estos “trucos” puede pasar que las page tables sean muy grandes para entrar en mem todas al mismo tiempo. Por esto algunos sistemas las colocan en **memoria virtual del kernel**, permitiendo al sistema el **pasar(swap)** algunas de estas page tables al disco cuando hay poca memoria.

**Chapter 26 Concurrencia: Una introducción**

Introduciremos 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, una forma de verlo es pensar en cada hilo como un proceso aparte, pero todos los hilos comparten el address space por lo que comparten la misma data(info).

El estado de cada hilo es similar al de un proceso, tiene su propio PC(program counter, registro que guarda a que instrucción está apuntando/ejecutando). Cada hilo tiene su propio set de registros privados, por lo que si dos hilos corren en un CPU hace un cambio de hilo1 a hilo2 se deben guardar/restaurar los registros respectivamente, es decir un context switch.

En un context switch entre procs se guardaba su estado en un **process control block(PCB)**, con hilos se necesita un **thread control block(TCB)** donde guardar el estado de cada hilo de un proc. La mayor diferencia es que no hace falta cambiar el address space en un context switch entre hilos ya que lo comparten, es decir no cambia la page table, las traducciones de la TLB todavía pueden servir.

Otra diferencia es que los los programas de hilos múltiples cada hilo tiene su propio stack, ya que cada hilo corre independientemente realizando sus propias rutinas con sus propias variables y cada resultado o variables de un hilo van a su propio **thread local storage**(almacenamiento local del hilo) o en otras palabras su stack. Ahora hay más bloques en el address space, pero no es un problema porque generalmente el stack no suele ser tan grande(salvo en proc muy recursivos).

-Porqué usar hilos?

Hay dos razones importantes para usar hilos. Primero **paralelismo**, se pueden encargar diferentes tareas a diferentes hilos dividiendo el trabajo y por lo atento haciéndolo más rápido y eficiente. La tarea de convertir un programa **single-threaded**(de un hilo) para que pueda hacer esta división del trabajo en varios PUs se llama **paralelización.**

La segunda es para evitar bloquear el progreso de un programa debido a un lento I/O. Usar hilos es normal para no detenerse, mientras un hilo espera, el CPU puede cambiar a otro y así no detenerse.

El “hilar” permite **overlap**(superponer) I/O con otras actividades dentro del mismo programa. La diferencia con **multiprogramming** es que al compartir address space es más fácil compartir data.

-Creación de hilos

Los hilos no necesariamente se ejecutan en el instante en el que se crean, llamemos a dos hilos Ta y Tb (thread a y b), el que creemos primero a Ta no significa que corra primero. Puede que sí, pero depende del scheduler, no se puede asumir de antemano el orden de la ejecución.

Una forma de pensar la creación de hilos es verla como hacer una llamada a función, pero en vez de ejecutar una función y devolver el resultado al que llamó la fun. el sistema crea un nuevo hilo que ejecuta la rutina que está siendo llamada, y corre de forma independiente al caller(el que llamó la creación del hilo), antes de volver.

La ejecución de los proc es administrada por el OS scheduler, por esto no se puede saber que correrá en cada momento.

Con la concurrencia la complejidad general del funcionamiento del sistema aumenta.

-Porque pasa esto? Shared data

Cómo interactúan los hilos al acceder a memoria? En el ejemplo del libro se pueden ver dos hilos que simplemente suman a un contador de a 1 en un loop la misma cantidad de veces cada uno, digamos 10.000, pero al final el contador no tiene 20.000 como valor como se esperaría si ambos hilos sumarán bien. Porque?

-Corazón del problema: Scheduling sin control

El problema yace en la variable contador, al cual se le suma en cada caso. El primer hilo puede haber sumado 1 pero antes de poder guardar el resultado un context switch guarda el estado actual del proc y corre el hilo 2, el cual le suma uno al contador y guarda este valor. Luego otro context switch ocurre, vuelve a correr el hilo 1 y realiza su última instrucción del ciclo guardando el valor del contador que el ténia, si el valor inicial del contador era, digamos, 50 luego de esto sera 51 en vez de 52 como sería en una ejecución correcta con un ciclo realizado por cada hilo, sumando 1 cada uno.

Esto es lo que se llama **race condition**( o más bien **data race**) (condición de carrera/ carrera de datos). El resultado depende del tiempo de ejecución, que con mala suerte, dará un resultado incorrecto. Y no es seguro que siempre el mismo resultado, esto es **indeterminista**( queremos algo **determinista**).

Cuando muchos hilos pueden ejecutar un código y causar una race condition, llamamos a este código **critical section**(sección crítica). Una pieza de código donde se accede a una variable compartida( o recurso compartido) y no debe ser ejecutada al mismo tiempo por más de un hilo.

Lo que necesitamos para este código se llama **mutual exclusion**( exclusión mutua). Esto garantiza que solo un hilo a la vez accede a la zona crítica.

-El deseo de atomicidad

Una solución sería tener instrucciones poderosas que , en un solo paso, hicieran todo lo que necesitamos y removieran así la posibilidad de un interrupt. Es decir, son atómicas, se pueden definir como “todo o nada”, si hay un interrupt o bien la instrucción no se ejecutó o ya ejecutó hasta completarse.

Pero en general no tendremos esto, lo que haremos será pedirle al hardware instrucciones con las que podemos construir lo llamado **synchronization primitives**(sincronización primitiva). Con soporte del hardware y ayuda del OS podemos construir código de hilo múltiple que acceda a secciones críticas de forma sincronizada y controlada.

-Un problema más: Como se esperan el uno al otro

Otro problema es cuando un hilo debe esperar que otro termine alguna acción para ejecutarse(como un I/O). Más adelante veremos mecanismos para dar soporte a esto del tipo sleeping/waking entre hilos, bastante común en programas múlti hilo.

-Palabras clave concurrencia:

**-sección crítica**

**-race condition**

**-indeterminate**(indeterministic, de resultados no deterministas, puede variar)

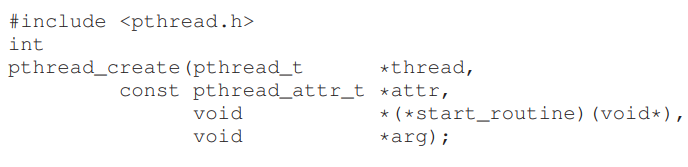
**-mutual exclusion**

**Chapter 27 Interludio: Thread API**

(Interfaz de hilos)Usamos POSIX threads library

Cómo crear y controlar hilos?

Primero hace falta un programa de hilos múltiples para poder crear hilos. Algo así



Hay 4 argumentos: thread, attr, start\_routine y arg. El primero es un puntero a una estructura de tipo pthread\_t; con esta estructura interactuaremos con el hilo, para esto hace falta pasarsela a pthread\_create() para inicializarla. El segundo argumento es attr, se usa para especificar atributos del hilo( como por ejemplo la prioridad de scheduling del hilo). Se inicializa con una llamada aparte a pthread\_attr\_init()(le vamos a pasar NULL nomas en el ejemplo). El tercero y más complejo simplemente pregunta: Qué función debería ejecutar en este hilo?(**function pointer** en C) espera algún nombre de función. Por último arg es el argumento que se le pasa a la fun que ejecute el hilo. (Se usan void pinterest para permitir cualquier tipo de argumento/resultado).

Una vez se ejecuta la creación de un hilo ya tenemos otra entidad en ejecución independiente que usa el mismo address space.

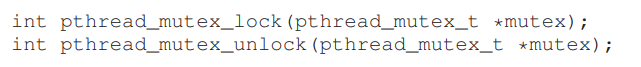
-Thread Completion

Si queremos esperar que un hilo termine de ejecutarse se llama a pthread\_join(). Esta rutina toma dos argumentos, el primero de tipo pthread\_t, especificando el hilo el cual esperamos. Y el segundo es un puntero al valor de retorno que se espera. Si creamos un thread sin argumentos podemos ponerle NULL en arg y lo mismo en pthread\_join(), si no nos interesa el valor podemos poner NULL.

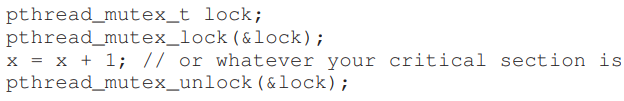
Hay que tener cuidado sobre qué valor retorna un hilo, nunca devolver un puntero a un valor en el stack del hilo, este se borra/reasigna la memoria cuando el hilo acaba. Por otro lado, no todos los proc usan join, ya que puede que se creen hilos “trabajadores” que se usen indefinidamente hasta que acabe el programa en un long-lived program.

-Locks

Para proveer **mutual exclusion** a una sección crítica usamos locks. El par de rutinas más básicas es de la siguiente forma:



Cuando se tiene una región crítica en el código y se necesita protegerla se usa un lock para bloquear el acceso antes de la sección crítica y se libera el lock al terminar la ejecución de esta. Algo así:



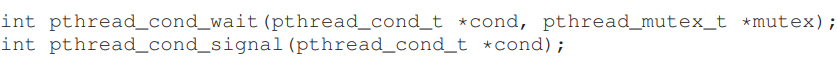
La idea es la siguiente, si ningún otro hilo posee el lock cuando pthread\_mutex\_lock() es llamado, el hilo lo adquiere y entra a la critical section. Si otro hilo tiene el lock, el que intentó agarrar el lock no volverá de la llamada hasta que lo consiga. Puede que haya muchos hilos en esta situación, y solo el hilo que tiene el lock puede llamar al unlock. Los locks se pueden inicializar así:



También está pthread\_mutex\_destroy() para cuando ya no usa más un lock.

-Condition Variables

Otro componente importante son las condition variables, son útiles cuando se debe mantener algún tipo de señales entre hilos. Como esperar que otro hilo haga algo antes de continuar. Hay dos rutinas principales para esto:

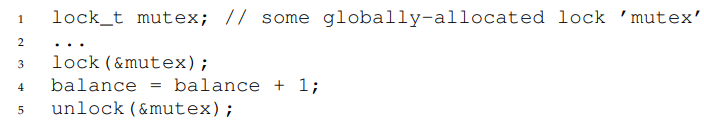


Para usar una condition variable se debe tener un lock asociado a esta y se debe tener control de este al llamar la rutina. La primera pone el hilo a dormir hasta recibir cierta señal, luego libera el lock que este tenga y se queda chequeando la condición en un loop. La segunda se usa para mandar una señal a la condición señalada.

**Chapter 28 Locks**

-Locks, la idea básica

Pensemos en la sección crítica como la típica variable compartida( más tipos de zonas críticas posibles y más complejas pero para el ej con esto basta). Para usar lock se agrega un poco de código alrededor de la zona a proteger. Por ejemplo:



Un **lock** es solo una variable, por ende debe ser declarada la **variable lock** de algún tipo. Esta variable tiene el estado del lock en cualquier momento dado. Si está **available**/**disponible**( o **unlocked**, **free**), es decir ningún hilo tiene el lock en su posesión, o está **acquired**/**adquirido**( o **locked**, **held**), lo que significa que un hilo(y solo uno) tiene el lock. Podemos guardar otros datos en el lock así como que hilo tiene el lock en su poder(si lo hay), o una lista de los hilos en espera de adquirir el lock, esta info está oculta al usuario.

Las rutinas lock() y unlock() son simples. lock() trata de adquirir el lock, si ningún hilo lo tiene en su control el lock( is unlocked), el hilo va a adquirir el lock y entrar a la zona crítica, algunas veces se llama a este hilo el **dueño**/**owner** del lock. Si otro hilo llama lock() con la misma variable lock, no va a volver(se queda en un bucle intentando conseguir el lock) mientras otro hilo tenga el lock, previniendo así que otros hilos accedan a la zona crítica(al no poder tomar el lock).

Los lock otorgan un mínimo de control sobre el scheduler al usuario. Los hilos son entidades creadas por el programador y administradas por el OS scheduler, con los locks el usuario solo se asegura que solo un hilo accedera a una sección de código al mismo tiempo.

-Pthread locks

La POSIX library usa **mutex** como nombre general de las locks, los usa para proveer de **mutual exclusion** entre hilos.

Las rutinas de POSIX pasan variables a lock() y a unlock(), porque pueden usarse diferentes locks para proteger diferentes variables(zonas críticas). Hacerlo aumenta la concurrencia, en vez de un solo bloque grande que se bloquea 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).

-Controlando Interrupts

Una de las primeras soluciones fue desactivar las interrupciones en las secciones críticas asegurando la atomicidad en estas. La ventaja es la simplicidad. Lo negativo: Primero significa hacer que un hilo llame una operación privilegiada y confiar en que no abuse de esto. Esto es malo, un programa codiciosos puede llamar a lock() al comienzo de su ejecución y monopolizar el procesador, o un programa malicioso llamar a lock() e ir a un loop infinito. En cualquier caso el OS pierde control del sistema, deshabilitar las interrupciones significa confiar mucho en las aplicaciones. Otra opción es trabajar en un solo proc para evitar accesos paralelos a la zona crítica, mala performance.

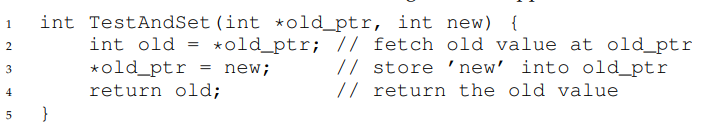
-Intento fallido: Usar solo Loads/Stores

Sin desactivar las interrupciones tendremos que usar hardware e instrucciones provistas por este para construir un lock. Probemos usando una variable flag para indicar si algún hilo tiene un lock en su posesión. El primer hilo en entrar a la sección crítica chequea si la flag es 1(como no se ha usado el lock no lo es), luego adquiere el lock si flag=0 y actualiza el valor a 1, mostrando el lock como adquirido, cuando termine de ejecutar la zona crítica llama a unlock que setea la flag en 0 nuevamente. Si un hilo intenta adquirir el lock mientras esté en posesión de otro hilo va a hacer **spin wait**(un bucle chequeando la condición del lock) hasta que el lock sea liberado.

Este código tiene dos problemas: uno es correctness(exactitud) y performance(desempeño), la exactitud falla porque con las interrupciones del OS puede darse el caso de que dos hilos seteen la flag en 1, accediendo simultáneamente a la zona crítica, que es lo que se busca evitar(mutual exclusion). Luego hacer el spin waiting tiene un costo muy alto, he ahí el problema de performance.

-Construyendo spinlock que funcionen con Test-And-Set

El hardware más simple de entender es conocido como **test-and-set**(o **atomic exchange**) instruction. En código de C se ve así:



Test and set devuelve el valor viejo apuntado por old\_per, y simultáneamente actualiza el valor con new. La clave de esto es que esta secuencia es atómica. Se llama test-and-set porque permite testear el valor viejo y setear la memoria con el nuevo valor. Con esta instrucción se puede construir un **spin lock**(usando test-and-set para chequear la condición del lock). Al hacer el **test** del valor viejo del lock y **set** del nuevo atómicamente conseguimos que solo un hilo acceda al lock. Ej: si ningún hilo tiene el lock y uno lo intenta conseguir test-and-set chequea el valor ,es 0, y lo actualiza a 1 adquiriendo el lock. Luego si otro proc quiere el lock y llama a test-and-set este chequea el valor, es 1, y lo vuelve a actualizar a 1, no consiguiendo el lock, y así hasta que el **owner** llame a unlock, setee el valor en 0 y el hilo que quedo en **spin-wait**(loop esperando el lock) chequea el valor con test-and-set, es 0, lo actualiza a 1 y adquiere el lock.

Se llama **spin lock** por como cuando espera el lock se queda haciendo “girando” usando ciclos del CPU hasta poder obtener el lock. Para funcionar bien en un único procesador hace falta un

**preemptive scheduler**. Sin un interrupt que cambie de proc cada cierto tiempo, un proc que quede haciendo spining nunca liberaria el CPU.

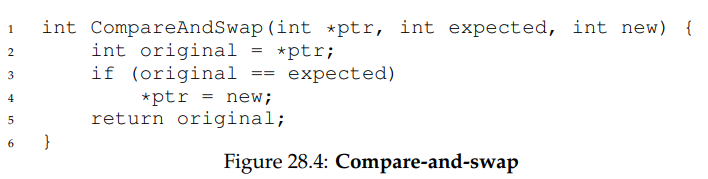
-Evaluando Spin Locks

El principal aspecto que debe cumplir es **correctness**: provee mutual exclusion? Si, cumple su función. Después, el siguiente es **fairness**: es justo con los hilos en espera? No, no está garantizado, incluso puede pasar que un hilo quede en spin indefinidamente. Puede producir starvation.

El último aspecto es la **performance**: cuál es el costo de usar spin locks? Depende del caso, pero en general hacer spin no gasta muchos ciclos del procesador y las zonas críticas no suelen ser muy largas, por lo que en cuanto a performance es efectivo.

-Compare-And-Swap

Otro primitive hardware es conocido como **compare-and-swap**, o compare-and-exchange. Se ve algo así:



La idea es comprar e intercambiar para testear si el valor de la dirección especificada por ptr con el nuevo valor y ver si es igual a expected(esperado), si lo es, actualizar la memoria señalada por ptr con el nuevo valor. Si no, no hacer nada. En cualquier caso devuelve el valor original, permitiendo al código llamar a compare-and-swap para saber si tuvo éxito o no.

Se puede generar un lock similar al de **test-and-set**, simplemente usando **compare-and-swap** en el cheque de condición. Compare and swap es una instrucción más poderosa que test and set, en spinlock no varia mucho pero más adelante usaremos este poder en **lock-free synchronization**.

-Load-linked and Store-Conditional

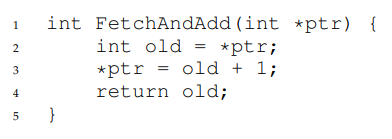
Algunas plataformas proveen instrucciones para ayudar en la construcción de secciones críticas. En MIPS se provee **load-linked** y **store-conditional** como instrucciones para construir locks y otras estructuras de concurrencia. load-linked es parecida a la típica instrucción que toma un valor de memoria y lo coloca en un registro. La diferencia está en store-conditional, que solo tiene éxito (y actualiza el valor almacenado en la dirección donde se usó load-linked) si no hubo intervención de almacenamiento en esa dirección. Si lo logra, devuelve 1, si no no actualiza el valor y devuelve 0.

Veamos el código de lock(). Primero un hilo hace spin wait esperando que flag=0(indicando que el lock está libre). Cuando esto pasa el hilo intenta adquirir el lock a través de store-conditional, si lo logra, el hilo a cambiado atómicamente el valor de la flag a 1 y entra en la sección crítica.

Puede que más de un hilo realize 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 de hardware primitivo, atómicamente incrementa un valor mientras devuelve el valor viejo a una dirección particular. El código es algo así:



Con esto construimos un **ticket lock**, en vez de un solo valor está solución usa una variable ticket y turno combinados para construir el lock. La idea es: cuando un hilo quiere el lock, hace un fetch-and-add atómico en el valor del ticket, 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, cuando myturn==turn. Si es el turno del hilo este entra a la sección crítica. Unlock se logra al aumentar el valor del turno, así el siguiente hilo en espera(si lo hay) puede entrar a la sección crítica.

La ventaja de este método es que asegura que todos los hilos progresen, algo que los otros métodos no. Un hilo podía quedar haciendo spin wait indefinidamente.

-Demasiado Spinning?

Estos métodos son simples y funcionan, dos cosas muy importantes, pero puede ser muy ineficiente, desperdiciando procesador. Entonces, cómo evitamos el spinning?

-Un enfoque simple: Just yield

Qué hacer cuando ocurre un context switch en una zona crítica e hilos empiezan a hacer spinning sin fin. Primer enfoque, si vas a hacer spin solo ceder el CPU a otro hilo. Asumimos que tenemos yield() provisto por el OS con lo que un hilo puede liberar el CPU y dejar que otro hilo corra.

Un hilo puede estar en 3 estados(running, ready, blocked), yield es una system call que mueve al hilo de running a ready, es decir se saca del scheduler solo/desencola. Aun así este enfoque puede ser costoso, el costo del context switch para correr el siguiente hilo luego de que el actual llame a yield() puede ser alto, menor que hacer spinning pero alto. Y no soluciona el problema de starvation.

-Usar colas: sleeping en vez de spinning

Los enfoques anteriores tienen un potencial desperdicio del CPU y no previenen starvation. Entonces debemos explícitamente ejercer control sobre qué hilo es el siguiente para adquirir el lock dsp de que el actual lo libere. Con ayuda del OS en forma de una cola que siga la pista de qué hilos esperan el lock.

Para ejemplificar usaremos el soporte que da Solaris con las llamadas park() para dormir un hilo y unpark(ThreadID) para despertar el hilo especificado por su ID. Se pueden usar para construir un lock que duerma al hilo que pide el lock si este ya tiene dueño y que lo despierte cuando el lock es liberado. Se pueden combinar los enfoques anteriores, con la cola de procesos en espera del lock y el uso de dormir/despertar hilos para construir un lock que no cause starvation y que minimice el spinning haciendo de este lock un lock razonable en performance. Debe ser hecho con cuidado de que un mal timing no cause un **wakeup/waiting race**, con un mal timing de context switch un hilo puede terminar durmiendo para siempre.

-Two-phase Locks

Es un enfoque híbrido que se da cuenta que el spinning puede ser útil, en la primera fase el lock() spins por un pequeño periodo de tiempo con la esperanza de obtener el lock. Si no lo consigue rápido entra en segunda fase y el hilo es puesto a dormir hasta que se libere el lock. Esto es muy útil si el lock estaba justo por ser liberado.

**Chapter 30 : Condition Variables**

Entre los **primitives** que se necesitan para construir programas concurrentes hay más que locks. En particular hay muchos casos en los que un hilo desea chequear si una **condición(condition)** es true antes de continuar su ejecución(por ejemplo si un hilo “hijo” terminó su ejecución).

El problema es que hacer mientras se chequea la condición, es decir mientras se espera a que esta cambie al valor que el hilo espera para proceder? Spinning?

Cómo esperar una condición?

-Definición y rutinas

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

Para declarar una variable de condición se escribe algo como esto : *pthread\_cond\_t* *c;* lo que declara *c* como una variable de condición( hace falta inicializarla tmb). Una **condition variable**  tiene dos operaciones: **wait()** y **signal()**. **Wait()** se ejecuta cuando desea ponerse a sí mismo a dormir; **signal()** se ejecuta cuando un hilo cambió algo en el programa y , por ende, quiere despertar un hilo durmiente que espera en esta condición. Se ven algo así:

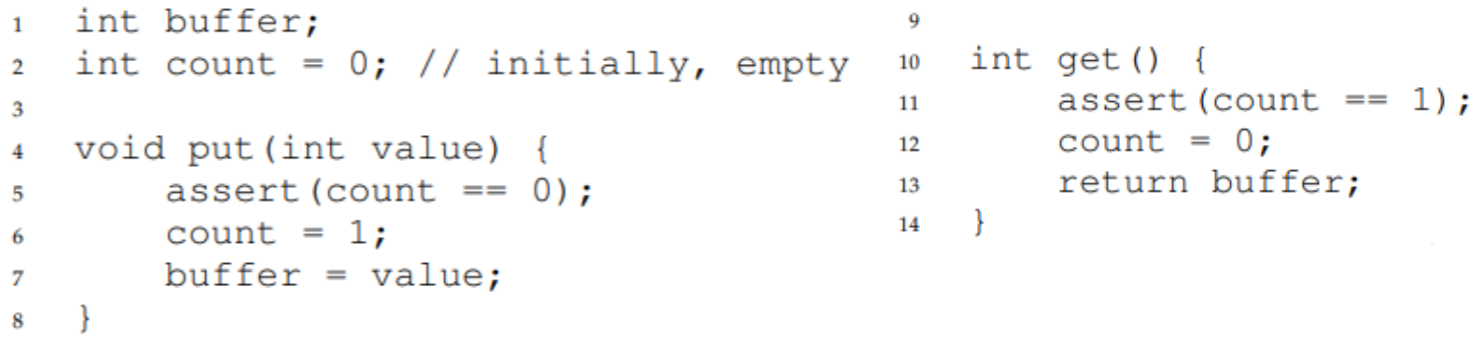


wait() toma un mutex/lock al ser llamada, asume que el hilo tiene un lock en su poder que debe ser liberado al dormir el proceso para que otros hilos/procs lo puedan adquirir. A su vez el hilo cuando se despierte debe readquirir el lock para poder ejecutarse. wait() debe liberar el lock y dormir el proceso de forma atómica. Esto es para evitar race conditions cuando un hilo está tratando de dormirse.

-The Producer/Consumer (Bounded Buffer) Problem

El próximo problema de sincronización a tratar es llamado problema de **productor/consumidor** o **bounded buffer** problem. Imaginemos uno o más *hilos productores*  y uno o más *hilos consumidores.* Los productores general data items(elementos de datos) y los colocan en un buffer; los consumidores obtienen dichos elementos y los consumen de alguna forma.

Como este buffer es un recurso compartido se necesita alguna clase de sincronización para acceder a este, para evitar una condición de carrera(race condition). usando un entero(llamemoslo contador) como buffer para ejemplificar y usando las siguientes rutinas:



Put actualiza el buffer a 1 si está vacío(count ==0) y get lo opuesto(lo pone en 0 si count==1). Se necesitan rutinas que chequeen cuando se puede acceder al buffer(count), ya que un productor solo debería añadir contenido a este si está vacío(count==0), y un consumidor sólo debe obtener un elemento cuando haya alguno disponible(count/=0). Esto va a ser realizado por dos tipos de hilos, hilos productores (**producer threads**) e hilos consumidores(**consumer threads**).

-Primer intento(condition variable)

Tanto put() como get() tienen secciones críticas que debemos proteger, pero sólo con locks no funciona, hace falta algo más, *condition variables*. Cuando un productor quiere llenar el buffer debe esperar a que este vacío(condition), lo mismo el consumidor, para consumir del buffer debe esperar a que este tenga contenido que consumir(condition). Esta solución no funciona para más de dos hilos(un consumidor y un productor), se puede dar el caso en el que un consumidor intente consumir y no haya buffers disponibles de los que consumir(imaginando un productor y dos consumidores).

Con más consumidores puede pasar que un consumidor al no poder consumir(no hay buffer con contenido) se duerma esperando consumir, luego un productor llena el buffer, da la señal de que está lleno, el consumidor 1 se despierta y trata de adquirir el lock para poder consumir, antes de esto mientras espera conseguir el lock un consumidor 2 ya lo obtuvo y consumió el buffer, luego suelta el lock, consumidor 1 lo adquiere e intenta consumir del buffer, pero está vacío, se levanta una excepción y el programa seguramente será terminado.

No hay garantía de que desde cuando un hilo se despierte hasta que corra la condición por la que se despertó se mantenga, suele referirse a esta interpretación de señales 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), si no está en true se duerme. Gracias a Mesa Semantics esta regla que siempre debemos recordar **siempre usar *while* loops**. Aunque no sea totalmente necesario es más seguro. Pero todavía hay un error.

Puede pasar que dos consumidores quieran consumir, el buffer está vacío, se duermen, luego un productor llena el buffer, después, como está lleno el buffer duerme a la vez que da la señal que despierta algún consumidor. Digamos consumidor 1, este consume el buffer, da la señal de que está vacío y duerme, pero que hilo se despierta(estan durmiendo productor 1 y consumidor 2 en la mismo condición) si se despierta productor 1 funciona, pero si despierta consumidor 2 se encontraría el buffer vació y dormiría. Ahora todos duermen indefinidamente. El error es que las señales deberían ser más directas, un consumidor no debería poder despertar a otros consumidores y viceversa.

-The Single Buffer Producer/Consumer Solution

La solución es usar dos variables de condición en vez de una para dar bien la señal de que tipo de hilo debería despertar cuando el estado del sistema cambia. Los consumidores pueden esperar en una condición **empty**(vacío) y dan la señal de **fill**(llenar); los productores esperan en la señal **fill**(llenar) y dan la señal **empty**(vacío). Así un consumidor nunca puede despertar a otro consumidor y viceversa.

-The Producer/Consumer Solution

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. Principalmente un productor solo debería dormir si todos los buffers están llenos y un consumidor sólo debería dormir si todos los buffers están vacíos.

-Covering Conditions

Otro problema de concurrencia a tratar es cuando un hilo llama a memoria para 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, a qué hilo(o hilos) despierta?

Una supongamos que hay 0 bytes disponibles, luego un hilo T1 pide 100 bytes, como no hay => duerme, otro hilo T2 pide 10 bytes, no hay =>duerme. Luego T0 libera 50 bytes, como elijo que hilo despertar? Una solución que evita problemas es despertarlos a todos asegurando que los que tenían que ser despertados lo son, el costo son todos los hilos que son despertados innecesariamente, los cuales rechequean la condición y como no se cumple duermen nuevamente. Esto se llama **covering condition**, asegura que cubre los casos que deben ser cubiertos.

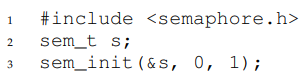
**Chapter 31 Semaphores**

Vamos a ver una primitiva de sincronización llamada **semáforos**, se pueden usar tanto como locks así como condition variables.

Cómo usar semáforos?

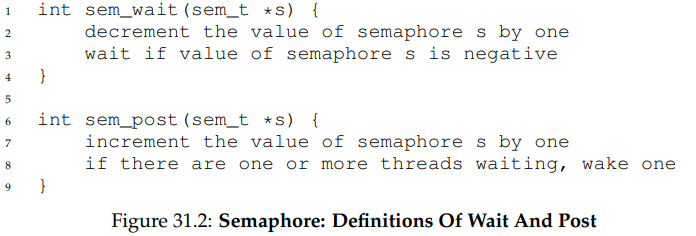
-Semáforos: Una definición

Un semáforo es un objeto con un valor entero que podemos manipular con dos rutinas; en el POSIX estándar estas rutinas son sem\_wait() y sem\_post(). Ya que el valor inicial del semáforo determina su comportamiento, antes de llamar otra rutina para interactuar con el semáforo debemos inicializar algún valor. Por ej:



El valor que lo inicializa es 1(el 0 indica que está compartido por los hilos del mismo proc). Luego de que un semáforo es inicializado podemos llamar a *sem\_wait()* o *sem\_post()* para interactuar con el. Más allá de la implementación de estas está claro que hay que tener cuidado, ya que con muchos hilos llamandolas hay que manejar estas secciones críticas.

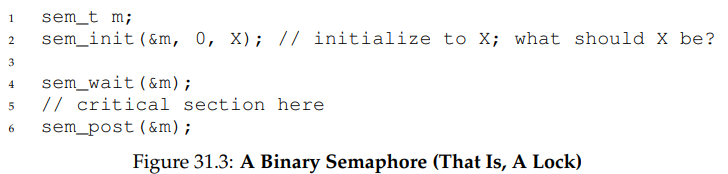
*sem\_wait()* puede volver inmediatamente( el valor del sem era 1 o mayor) o causará que el hilo duerma, con muchos hilos llamando a wait se hará una cola de hilos que esperan ser despertados. *sem\_post()* no espera por ninguna condición como wait, solo incrementa el valor del sem, y si hay algún hilo esperando ser despertado, despierta uno de ellos. Si el valor del sem es negativo es igual al núm. de hilos esperando despertar(el valor suele estar escondido al usuario).



Por ahora no nos preocupemos por race conditions, demos por hecho que las operaciones de los sem son atómicas.

-Semáforos Binarios(Locks)

Usaremos el semáforo como un lock.



Simplemente rodeamos la sección crítica de interés con un par de *sem\_wait()*/*sem\_post().* Para que esto funcione es crítico que el valor inicial sea 1. Si viene más de un hilo, el primero dejará el valor en 0 luego de entrar en la zona crítica, el siguiente hilo que quiera entrar será suspendido/dormirá y dejará el sem en -1, termina el hilo 1, deja el sem en 0(le suma 1 a -1), el hilo 2 despierta(queda en ready, el scheduler debe ponerlo a correr), adquiere el lock, entra en la sección crítica y para cuando termina el sem queda de nuevo en 1. Como los locks solo tienen dos estados(held or not held), el uso de un semáforo como lock se suele llamar **binary semaphore**(**semáforo binario**).

-Semaphores For Ordening

Los sem también son útiles para ordenar eventos en un programa concurrente(con concurrencia). Por ejemplo un hilo está esperando que una lista este **no-vacía** para borrar algo de ella. Este patrón de uso suele haber un hilo *esperando* que algo pase y otro *dando una señal* de que paso, y así despertando al primero. Por ende estamos usando al sem para ordenar(similar a cuando usamos **condition variables**). Esto se puede usar por ejemplo para que un *parent* espere a un *child*(padre a hijo), simplemente el parent debe llamar a sem\_wait() y dormir(valor inicial del sem==0) y el hijo 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→1 llama el padre y pasa de 1→0, sin necesidad de dormir).

-El problema de Productor/Consumidor( Bounded Buffer)

Si usamos dos semáforos, llamémoslos **empty** y **full** y un solo buffer, usando los semáforos para comunicar el estado del buffer, cuyo valor máximo es 1 (empty inicializado en 1, y full inicializado en 0). Esto funciona correctamente, ya sea con 2 hilos o más, pero si aumentamos el valor máximo que puede tomar el buffer(digamos 10) se produce una race condition(suponiendo muchos hilos).

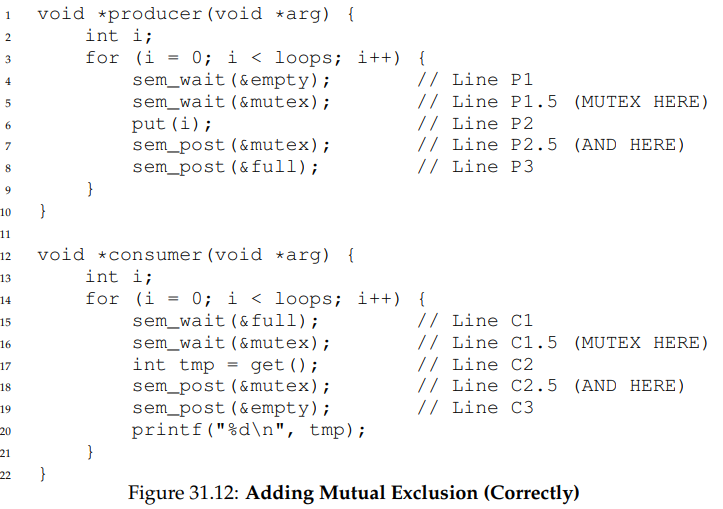
**Una solución: Añadir mutual exclusion(mal hecho)**

El llenar un buffer e incrementar el índice dentro del buffer es una sección crítica, por ende debe ser protegida. Con el uso de binary semaphore lock podemos intentar, pero si ponemos locks de este tipo rodeando las instrucciones *put()*/*get()*, aunque parece correcto se da un **Deadlock**(tmb llamado bloqueo mutuo, abrazo mortal, etc. dos claves/locks se llaman de manera opuesta llegando a un punto donde nadie puede conseguirlas nuevamente).

En este caso lo que ocurre es simple, un consumidor obtuvo el lock, luego el buffer está vacío, se bloquea, el productor podría llenar el buffer y despertar luego al consumidor pero no puede adquirir el lock, el cual quedó con el consumidor. Deadlock.

**Una solución que anda**

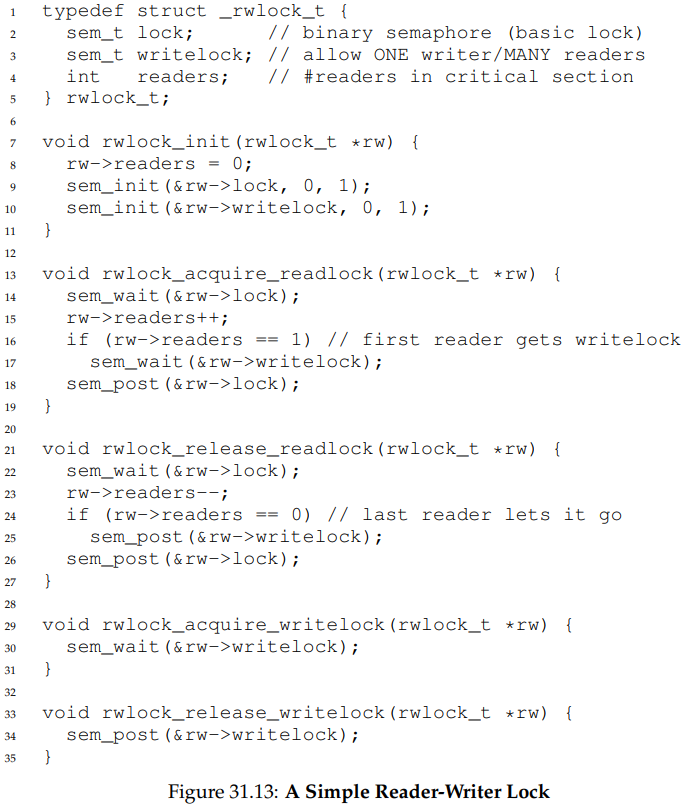
Solo hay que reducir el alcance del lock, lo ponemos exactamente alrededor de la sección crítica, el código sería así:(las versiones anteriores del code estan en el apunte)

****

**-**Reader-Writer Locks

Este es otro problema clásico que necesita de 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 búsqueda. Insertar si es una sección crítica a ser protegida, se podría insertar al mismo tiempo en un mismo lugar dos cosas, mal! Por otro lado mientras las búsquedas solo necesitan leer sin cambiar nada, mientras no haya una inserción en el medio(ya que modifica lo que se está leyendo) puede haber tantas búsquedas como se necesite. Para dar soporte a este tipo de operación está a lo que se refiere como **reader-writer lock**(lock de lector/escritor).

Si se intenta subir datos a la estructura se adquiere el lock de escritura, y con otra operación se libera, esto para permitir que solo un escritor a la vez pueda modificar la estructura. Por otro lado, el lock de lectura cuando es adquirido también aumenta la variable *lectores* que sigue la pista de cuantos hilos están leyendo la estructura. Cuando el primer lector adquiere el lock de lectura, tmb adquiere el lock de escritura, así segura que no se modifique la estructura mientras la lee. Para escribir los escritores deben esperar que todos los lectores terminen.



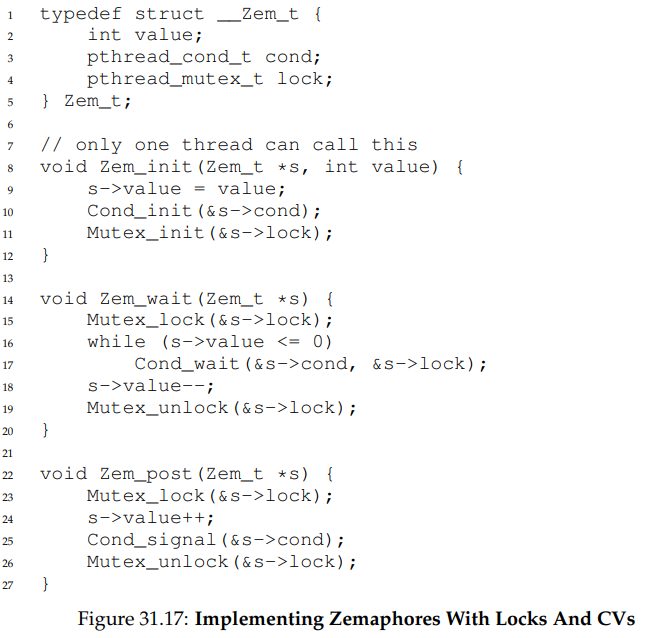
Lo malo de esto es que es injusto, es muy fácil que los lectores monopolicen el lock(writers starve).

-Thread Throttling(estrangulación de hilos)

Cómo puede un programador prevenir que “demasiados” hilos hagan algo a la vez atascando el sistema? Solución, decidir un límite a cuánto es “demasiados”, luego usar un semáforo para limitar el número de hilos en concurrencia. Este enfoque se llama **throttling**(estrangulación), es considerado una forma de **control de admisión**.

-Cómo implementar semáforos

Lo haremos con locks y variables de condición:



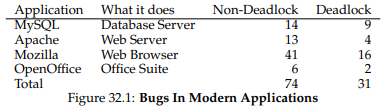
La diferencia entre estos semáforos y los de Dijkstra(los que venimos viendo) es que el valor del semáforo no puede bajar de 0, sin reflejar en negativo la cantidad de hilos durmientes(La implementación de Linux tmb tiene esta característica).

**Chapter 32 Problemas Comunes de Concurrencia**

Veremos otros errores comunes además del deadlock.

-Qué tipos de Bugs/errores existen

Resultados de un estudio sobre errores de concurrencia resueltos:



Qué tipo de bugs son los Non-Deadlock bug?

-Non-Deadlock bugs

Veremos los dos tipos más comunes de estos: **atomicity violation**(violación de la atómicidad) y **order violation**(violación del orden).

**Atomicity-Violation Bugs**

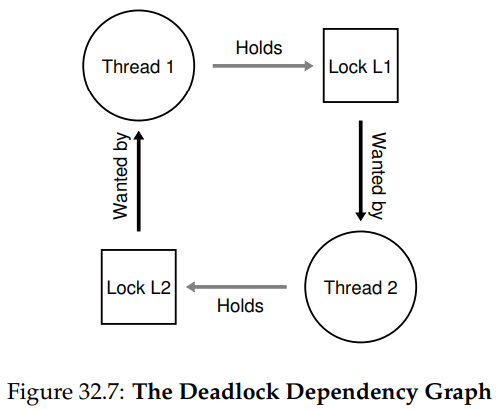
La serializabilidad deseada entre accesos a memoria múltiples es violada(se espera una ejecución atómica pero no es así durante la ejecución). Normalmente (pero no siempre) la solución es directa, poner locks alrededor de la zona a proteger.

**Order-Violation Bugs**

El orden deseado entre dos (o más) accesos a memoria son volteados/dados vuelta( Por Ej A debe ejecutarse siempre antes que B, pero no pasa en la ejecución). La solución que generalmente arregla este problema es el uso de variables de condición( el uso de semáforos también puede servir).

-Deadlock Bugs

Generalmente ocurre cuando hay una dependencia de locks entre hilos, digamos el hilo T1 tiene el Lock L1 y está esperando por otro lock, L2, pero el hilo T2 tiene en su poder L2, y está esperando el lock L1. En el siguiente gráfico se puede ver un **ciclo**, un indicativo de deadlock.



**Por qué ocurren Deadlocks?**

En un caso simple como el ejemplo anterior con asegurar el orden de adquisición de los locks bastaría para evitar el deadlock pero en código más complejo no es tan simple. En grandes proyectos se debe construir un sistema de locking que evite el deadlock potencial en las dependencias circulares que ocurren naturalmente en el código.

Otra razón es debido a la naturaleza de la **encapsulación**. Los desarrolladores tienden a construir su software sin mostrar los detalles de la implementación, construyendolo en forma modular. Esto no se lleva bien con los locks.

**Condiciones para un Deadlock**

Se deben cumplir 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 (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.

Si no se da ninguna de estas condiciones, no puede haber un deadlock. Veremos técnicas para prevenir esto.

**Prevention**(prevenir dichas condiciones antes de llegar a un Deadlock)

**Circular Wait**

Una buena forma es escribir el código en forma de que no caiga en un circular wait. La forma más directa de lograrlo es proveyendo **orden total** en la adquisición de los locks. En sistemas complejos con muchos locks esto es complicado, por ende se usa un **orden parcial** para estructurar la adquisición de locks, para evitar un deadlock antes de que suceda.

Ambas, orden total y parcial requieren un cuidadoso diseño de estrategias de locking, que deben ser construidas con cuidado. El orden es sólo una convención, un programador descuidado que ignore el protocolo de locking puede fácilmente causar un Deadlock.

**Hold-and-wait**

Puede ser evitado adquiriendo todos los locks a la vez, atómicamente. Esto requiere un lock global de prevención a ser adquirido antes de adquirir los demás para asegurar que solo un hilo los adquiera atómicamente. Esto tiene algunos problemas, la encapsulación nos juega en contra, se necesitaria saber de antemano qué lock requiere el hilo para obtenerlos de antemano. Esto tmb disminuye la concurrencia, ya que los locks son controlados de antemano en vez que solo cuando son necesarios para el hilo.

**No Preemption**

Generalmente las lock se ven en estado held(alguien las tiene) hasta la llamada de unlock(), las llamadas múltiples de adquisición de locks seguido causan el que un hilo esté esperando un lock mientras tiene otro en su poder. Muchas librerías proveen algunas rutinas para evitar esto, como una rutina que trate de obtener el lock(trylock) o devuelva un código de error si está en control de otro hilo(se puede probar de nuevo dsp para ver si se consigue el lock). Pero surge un nuevo problema: **livelock.** Es posible que dos hilos intenten esto y fallen(ambos) en conseguir el lock, y esto sucede continuamente, no hay deadlock pero el programa no progresa, de ahí el nombre **livelock**. Una solución sería poner un retraso antes de reintentar conseguir el lock, disminuyendo las posibilidades de competencia entre hilos.

Surge nuevamente el problema por encapsulación, si un lock está enterrado en alguna rutina que está siendo llamada, volver al principio se vuelve complejo de implementar.

Aunque este enfoque no añade la capacidad de retomar un lock por la fuerza, sino que usa trylock para permitirle al desarrollador solar el lock.

**Mutual Exclusion**

La última técnica sería evitar del todo la necesidad de exclusión mutua, esto es difícil, qué se puede hacer? Se pueden construir estructuras de datos que no requieran locks(**lock-free**): usando poderosas instrucciones de hardware, se pueden construir estructuras de datos en forma que no necesiten lock explícitos.

**Evitar Deadlock a través del Scheduler**

En vez de prevenir deadlocks, en algunos casos **evitarlos** es preferible. Requiere conocimiento global de que lock puede ser que soliciten los hilos durante su ejecución y así el scheduler los ejecuta de una forma que garantice que no ocurra un deadlock. Lo malo de este enfoque es que puede disminuir la concurrencia, por esto no es un enfoque muy usado.

**Detectar y Recuperarse**

La última estrategia general es dejar que algún deadlock ocurra generalmente y tomar acción una vez este ha sido detectado. Si los deadlock son raros, un enfoque de no-solución como este puede valer la pena. Muchas bases de datos emplean detección y recuperación de deadlocks. Un detector corre periódicamente, construyendo un gráfico de los recursos y chequeando por ciclos. Si hay un deadlock, el sistema debe ser reiniciado.

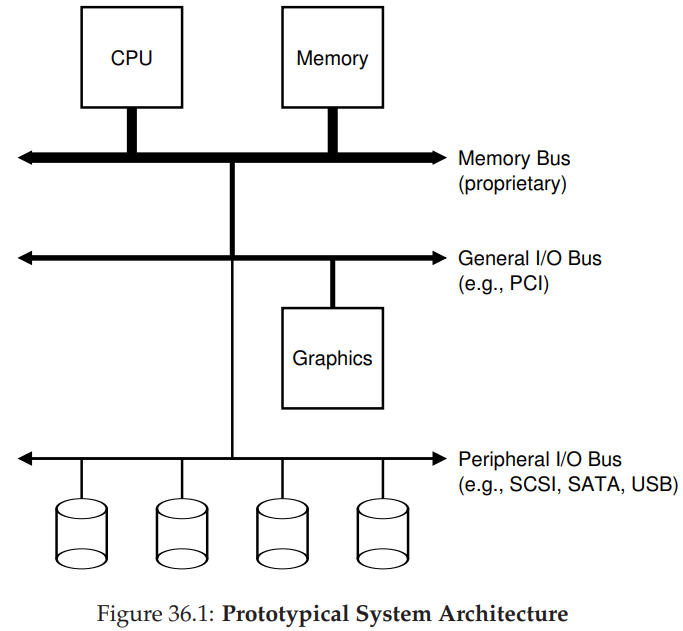
**Chapter 36 I/O Devices**

Introduciremos el concepto de **input/output(I/O) device** (dispositivo de I/O) y como el OS interactúa con esta entidad. I/O es crítico en los sistemas de computación: un programa sin input siempre produciria el mismo resultado y cuál es el sentido de correr un programa sin output, nunca devuelve nada. Ahora:

Cómo integrar I/O en los sistemas?

-System Architecture

Veamos primero la arquitectura del sistema:

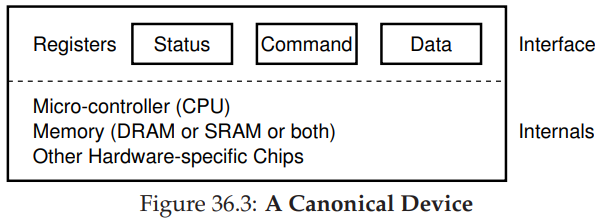


Se ve un sólo CPU conectado a la memoria principal por algún tipo de **memory bus**(cable de memoria). Algunos dispositivos están conectados al sistema por un **I/O bus**(en los sistemas modernos suele ser **PCI**); gráficos y otros dispositivos de alta performance de I/O pueden encontrarse en esté nivel. Finalmente en la parte más baja un **bus periférico** que conecta dispositivos lentos al sistema como **discos, mouse, teclado**.

Por qué se necesita esta jerarquía? Básicamente: Física y costos. Mientras más rápido el cable/bus más corto debe ser, por ende la memoria de alto desempeño no tiene mucho espacio para la conexión de dispositivos. Los dispositivos de menor desempeño se alejan del CPU. Las ventajas de poner dispositivos como los discos en un bus periférico son muchas, en particular podes poner varios.

-Un dispositivo canónico

Ahora veamos un dispositivo canónico(no es real) y usémoslo para entender qué maquinaria se necesita para que la interacción de los dispositivos sea eficiente. En la siguiente imagen …

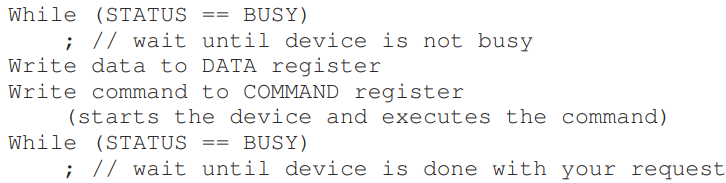


…se pueden ver dos componentes importantes. Primero la **interfaz** del hardware que se le presenta al resto del sistema, así como el software el hardware debe presentar alguna interfaz que permita al sistema de software controlar su operación.

La segunda essu **estructura interna**. Esta parte es específica de la implementación y responsable de implementar la abstracción que el dispositivo presenta al sistema.

-El protocolo canónico

En la imagen el dispositivo está compuesto por 3 registros: **status** register(muestra el estado del dispositivo); **command** register(le dice al dispositivo que haga cierta tarea); y un **data** register(le pasa datos al dispositivo u obtiene datos de este). El OS controla el dispositivo con estos registros. Veamos ahora una interacción típica del OS:



El protocolo tiene 4 pasos. Primero el OS espere hasta que el dispositivo esté listo para recibir un comando(repetidamente chequea el reg status). Llamamos a esto **polling** the device (polling=votación)(básicamente preguntar qué está pasando). Segundo, el OS envía datos al data register, si esto fuera un disco el OS necesitaría hacer muchas escrituras hasta que transfiera un disk block(un bloque de datos entero). Cuando el CPU principal está involucrado en el movimiento de los datos nos referimos a nos referimos a esto como **programmed I/O(PIO)**. Tercero, el OS escribe un comando en el command register, le deja saber al dispositivo implícitamente que ambos, los datos están y que debe empezar a ejecutar el comando. Finalmente el OS espera en un loop haciendo polling(puede tener éxito o dar un código de error si falla).

Este protocolo básico tiene la ventaja de ser simple y funciona, pero tiene problemas. Es ineficiente, primero hacer polling desperdicia mucho CPU time, solo esperando que el dispositivo, que es potencialmente lento, complete su actividad, en vez de cambiar a otro proc que pueda usar el CPU mientras el OS espera.

Cómo evitar el costo del Polling?

-Disminuir los CPU overheads con interrupts

El OS puede, en un interrupt, dormir el proceso que espera el I/O y hacer context switch con otro proc para aprovechar CPU. Luego cuando el dispositivo terminó su operación, se inicia un hardware interrupt, causando que el CPU le de control al OS (**interrupt handler**), el hander es solo es código del OS que terminara el pedido del programa que hizo I/O, despertandolo para que pueda proseguir su ejecución(o operando en caso de un fallo del I/O solicitado).

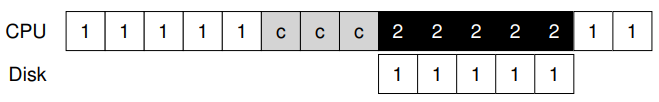
Los interrupts permiten hacer **overlap**, superponer procesos, cuando uno se bloquea esperando un I/O, otro puede ejecutarse en ese tiempo. Pero usar interrupts no siempre es la mejor solución, en el caso de uns dispositivo I/O muy rápido, el uso de un interrupt ralentizaria el proceso. Básicamente si el dispositivo es rápido, hacer **polling**, si es lento usar **interrupts**. Si no se conoce del todo la velocidad del dispositivo o esta puede variar se puede usar un enfoque **híbrido**, hacer polling un corto periodo y si el dispositivo no terminó un interrupt, este enfoque de **dos-fases** puede alcanzar lo mejor de ambos métodos.

Otra razón para no usar interrupt surge en las networks, si hay muchos paquetes en camino, y cada uno genera un interrupt, puede llevar a un **livelock**, el OS termina sin correr ningún user program con todos los interrupt.

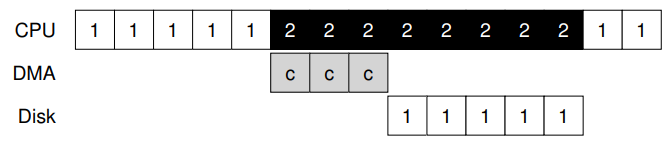
Otro optimización basada en interrupts es **coalescing**. En dicha configuración, un dispositivo que requiere un interrupt suceda primero espera un poco antes de entregar el interrupt al CPU. Mientras espera otros pedidos de I/O terminarán, y así múltiples interrupts pueden juntarse(coalescing) en uno solo, disminuyendo el overhead del manejo de interrupts(aunque esperar mucho es ineficiente).

-Movimiento de Datos más eficiente con DMA

Hay otro aspecto del protocolo canónico que necesitamos ver. Cuando ocurre un PIO(I/O con el CPU involucrado) para transferir un bloque de datos a un dispositivo, el CPU se sobrecarga con una tarea trivial, es decir se desperdicia CPU time que podría usar otro proc. Esto sucede si antes de que el dispositivo I/O haga lo que el proc necesita que haga hay que pasarle muchos datos, esto es lo que es lento y desperdicia tiempo de CPU, como se ve en el diagrama marcado con la letra *c* :



La solución a esto es un **Direct Memory Access(DMA)**, un DMA es un dispositivo muy específico dentro de un sistema, que puede hacer transferencias entre dispositivos y la memoria principal sin mucha intervención del CPU. Para usarlo el OS programa el DMA diciéndole dónde están los datos en memoria, cuantos copiar, y a que dispositivo mandarlo. Luego el OS queda libre y puede seguir con otra tarea. Cuando el DMA termina, hace un interrupt que le da a conocer al OS que la transferencia termino:



El CPU puede correr otro proc mientras el DMA se encarga de la transferencia de datos al dispositivo, luego el dispositivo realiza su tarea, y luego el proc 1 reanuda su ejecución en CPU.

-Métodos de interacción con dispositivos

Falta hablar sobre cómo el OS se comunica con los dispositivos. A lo largo del tiempo se desarrollaron dos principales métodos para dicha comunicación.

El primero, y más viejo, es tener **instrucciones de I/O** explícitas. Estas instrucciones especifican una manera en la que el OS puede mandar datos a los registros de un dispositivo específico, y así permite la construcción de protocolos como el antes descripto. Dichas instrucciones son privilegiadas, el OS es la única entidad con permiso para manejar los dispositivos.

El segundo método es conocido como **memory-mapped I/O**(memoria mapeada). Con este enfoque, el hardware hace que los registros de los dispositivos se vean como direcciones de memoria. Para acceder a un determinado registro el OS realiza un load(to read(lee)) o un store(to write(escribe)) en la dirección. Luego el hardware direcciona dicha acción del OS al dispositivo en vez de a la memoria principal. No hay gran ventaja de un enfoque sobre el otro, ambos se usan hoy en día.

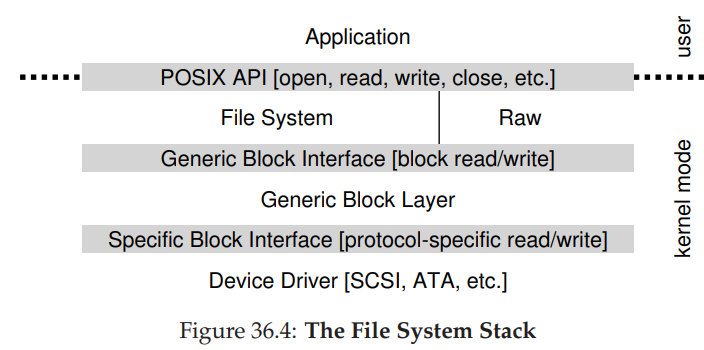
-Fitting into the OS: The device driver

Nuestro problema final es cómo encajan los dispositivos, cada uno con su específica interfaz, en el OS, al que queremos mantener lo más general posible.

Cómo construir un OS neutro frente a los dispositivos?

Este problema se resuelve a través de la **abstracción**. En el nivel más bajo, una pieza de software en el OS debe saber en detalle como un dispositivo funciona. Llamamos a eso un **device driver**(o driver del dispositivo), y cualquier forma específica de interacción con el dispositivo está encapsulada en su interior.

Para entender cómo ayuda la abstracción usaremos de ejemplo el sistema de archivos en pila de Linux:



Se puede ver que a un sistema de archivos(y una aplicación como se ve por encima) se le pasa inadvertida la especificación de qué clase de disco 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 tmb muestra la interfaz **raw** a los dispositivos, la cual permite aplicaciones especiales( como un **file-system checker**-lo veremos dsp- o una herramienta de **desfragmentación de discos**) realizar read/write blocks sin la abstracción de archivos.

La encapsulación antes señalada tiene su lado negativo tmb. Si un dispositivo tiene muchas capacidades especiales pero debe mostrar una interfaz genérica dichas capacidades se desperdician.

Los drivers de cada dispositivo son necesarios para cualquier dispositivo que se conecte al sistema, y se han convertido en la mayoría del código del kernel.

**Chapter 37 Hard Disk Drives**

Estos drives han sido la principal forma de persistencia de almacenamiento de datos en los sistemas de computadoras desde hace décadas y la mayor parte del desarrollo de tecnología de sistemas de archivos se basa en su comportamiento. Ahora la principal pregunta:

Cómo almacenar y acceder a datos/información en/de un disco?

-La interfaz

Primero entendamos la interfaz del drive de un disco moderno. La interfaz básica es bastante directa. El driver consiste en un gran número de sectores(bloques de 512-byte), donde cada uno puede ser leído o escrito. Los sectores están numerados de 0 a *n-1* en un disco de *n* bloques, podemos ver el disco como un array de sectores. Este es el **address space** del drive.

Operaciones con múltiples sectores son posibles y muchos file systems leerán o escribirán 4KB a la vez (o más). Pero cuando se escribe en el disco el driver solo asegura que la escritura de un bloque de 512-byte es **atómica**. Por ende si la computadora se apaga/pérdida de energía solo una parte de la larga escritura se guarda(algunas veces llamado **torn write/ lectura rasgada**). No hay una interfaz directa que lo diga pero hay algunas suposiciones comunes, como suponer que el acceso a dos bloques cercanos entre ellos dentro del driver address space será más rápido que si estuvieran lejos uno del otro. También se suele asumir que acceder a los bloque de forma secuencial, uno detrás de otro, es la forma más rápida de hacerlo, en especial que un acceso aleatorio.

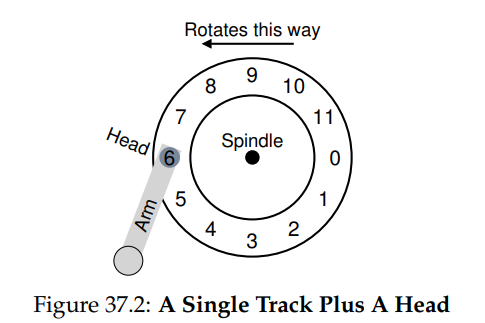
-Geometría básica

Entendamos algunos componentes del disco. Empecemos con un **platter**(plato), una superficie circular dura donde datos son almacenados de forma persistente al inducir cambios magnéticos en este. Un disco puede tener uno o más(platters), cada uno tiene dos lados, cada uno es llamado una **surface**(superficie). Generalmente está hecho de un material duro, como aluminio, y recubierto con una capa magnética que le permite al drive almacenar bits de forma persistente incluso cuando el drive está apagado.

Los platos están unidos alrededor del **spindle**(eje), el cual está conectado a un motor que gorra los platos(mientras el drive tiene poder) a un constante ritmo(fijo y preestablecido). El ritmo es normalmente medido en **rotaciones por minuto(RPM)**, y los calores modernos típicos tienen un rango de de 7200 RPM a 15000 RPM(un ritmo de 10000 RPM significa que cada rotación dura 6 milisegundos/6ms).

Los datos son codificados en cada superficie en sectores circulares con el mismo centro; llamamos un círculo de estos(con mismo centro) un **track**(pista). Una sola superficie contiene varios cientos y cientos de pistas, todas muy juntas, cientos de pistas entran en el grosor de un pelo humano.

Para leer y escribir en la superficie, necesitamos un mecanismo que permita sentir(leer) los patrones magnéticos o introducir un cambio en ellos(escribir). Este proceso es logrado por el **disk head**(cabezal del disco), hay uno por superficie, y está unido a un solo **disk arm**(brazo del disco), el cual se mueve a través de la superficie para poner el cabezal encima de la pista deseada.



-A Simple Disk Drive

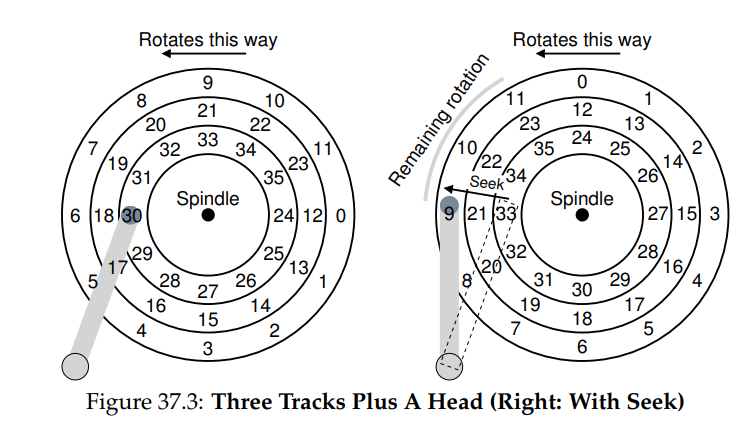
Ahora para entender cómo funciona un disco construyamos un modelo, imaginemos el de la imagen de arriba. Un disco simple con una sola pista con 12 sectores, cada uno de 512 bytes, y por ende su address space es de 0 a 11.

**Latencia de una sola pista: el retraso rotacional**

Cómo procesar un pedido este disco, imaginemos un pedido de leer el bloque 0. En nuestro disco, el disco no debe hacer mucho, solo esperar que el sector deseado rote hasta estar bajo el cabezal, esto es común en discos modernos y tiene un nombre: **rotational delat/rotation delay**(retraso rotacional). En el ejemplo si el retraso de rotación total es R, el disco debe hacer un retraso de alrededor de R/2 para esperar que el 0 este bajo el cabezal(si empezamos en el sector 6), el peor caso posible es tener que esperar un R(que de toda la vuelta) completo.

**Múltiples pistas: Tiempo de búsqueda(Seek Time)**

Nuestro disco tiene una pista, uno moderno tiene varios millones, veamos otro ejemplo de disco.



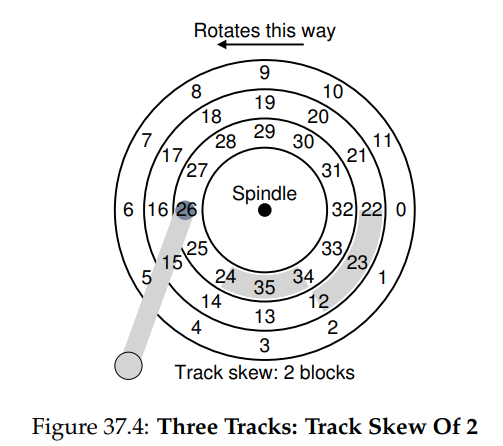
En la imagen el cabezal está en la pista más interna(sec 24 a 35), la siguiente tiene otro set de sectores(sec 12 a 23) y la última otro(sec 0 a 11). Imaginemos un pedido de lectura al sector 11, para esto primero el brazo debe mover el cabezal a la pista correcta, este proceso se llama **seek** (búsqueda), está, junto con las rotaciones, es una de las operaciones más costosas del disco.

La búsqueda tiene varias fases: primero *aceleración*, el brazo se comienza a mover, dsp *coasting* mientras el brazo se mueve a máxima velocidad, dsp *desaceleración* mientras el brazo frena; finalmente *settling*(asentamiento) mientras el cabezal cuidadosamente se posiciona sobre la pista, El tiempo de la última fase suele ser significativo(0.5 a 2 ms) ya que debe hacerse con cuidado. En lo que el brazo se movía el disco siguió rotando, el sector 11 casi está bajo el cabezal por lo que el retraso rotacional es bajo.

Cuando el sector 11 este bajo el cabezal inicia la fase final del I/O, conocida como **transfer** (transferir), donde los datos son leídos o escritos de/en la superficie. El tiempo de la operación sería el seek time + retraso de rotación.

**Otros detalles**

Muchos drives emplean algún tipo de **track skew**(desviación de la pista) para asegurar que las lecturas secuenciales se hagan bien cuando se cruza un límite/cambio de pista. Esto para ser más eficiente al mover el brazo.



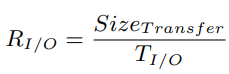
Otra realidad es que los sectores exteriores tienden a tener más sectores que los interiores. Estas pistas generalmente son llamadas **multi-zoned** disk drives (multi zonas del disco). Finalmente, una parte importante de los discos modernos es el **cache**, algunas veces llamado **track buffer**. Este cache es solo un pequeño monto de memoria( de 8 a 16 MB normalmente) que el disco puede usar para mantener datos que fueron leídos del disco o escritos en este. 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 pronto, si lo hay se hará rápido. En escritura el drive puede elegir, debería reconocer que la escritura se completa cuando los datos están en su memoria o cuando terminaron de escribirse en disco? La primera forma se llama **write back** caching(es rápida pero riesgosa), y la última **write through.**

-I/O Time: Haciendo la matemática

Ya podemos pasar a entender la performance del disco. En particular podemos representar I/O time de la siguiente manera:



Notemos que el ritmo del input/output(R\_i/o), es fácil de calcular(por ahora) y se usa para comparar discos. Es el tamaño transferido dividido por el tiempo que tomó:



Imaginemos dos tipos de cargas de trabajo/modos de trabajo, accesos **random**(aleatorio) y **sequential**(secuenciales). Para entender la diferencia en el desempeño debemos suponer algunas cosas sobre el disco. Los discos tienen varias características, pero podemos centrarnos en dos importantes: “high performance” (alto desempeño, giran lo más rápido posible, bajo seek time y transfiere rápido), el segundo es “capacity”(capacidad). Pensemos en dos discos diferentes basándonos en esto, por un lado un Cheetah, disco de alta performance y un Barracuda, centrado en la capacidad.

Primero analicemos el acceso random. Asumamos un 4KB read, en un lugar aleatorio del disco, cuánto tardaría esto, tomamos los datos del fabricante del disco(un Cheetah, alta performance):

****

Tseek=4 es un promedio, no la búsqueda total(una total puede ser el doble o más). El delay se calcula directamente de los RPM, 15000 RPM son 250 RPS(rotaciones por segundo), por ende cada rotación son 4 ms, el disco tendrá un promedio de media rotación, es decir 2 ms. El T\_i/o es de 6 ms para el Cheetah) y un R\_i/o de 0,66 MB/s. Luego un Barracuda(capacity disk) tiene un T\_i/o de 13.2 ms y un R\_i/o de 0,31 MB/s. Ahora acceso secuencial, asumimos un solo seek y delay de rotación antes de un largo transfer(digamos de 100 MB). El Cheetah tiene un T\_i/o de 800 ms y el Barracuda de 950 ms y un R\_i/o de 125 MB/s y 105 MB/s respectivamente.

Se ve que un workload random tiene un desempeño mucho mejor y que hay mucha diferencia entre un disco centrado en la velocidad/desempeño y uno centrado en la capacidad/acomodar mejor los bytes.

-Disk Scheduling

Debido a su costo el OS maneja los I/O decidiendo el orden de estos. Frente a varios pedidos de I/O el **disk scheduler** examina los pedidos y decide cual correr a continuación. A diferencia que el scheduler de procesos el del disco puede hacer una aproximación bastante acertada de cuanto va a tardar un I/O. Calculando según el seek time y el posible retraso de rotación, con esta información elige el pedido que menos va a tardar primero. El scheduler tratará de seguir el principio de **SJF**(shortest job first).

**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. Pero no es lo mejor, primero porque la geometría del drive no está disponible al Os, solo es un array de bloques(no lo ve de forma circular en el plato del disco). Esto se puede arreglar implementando **nearest-block-first(NBF)**(el bloque más cercano primero). El segundo problema que tiene es **starvation**, si hay muchos request en la misma pista, los de pistas exteriores pueden padecer starvation.

Cómo manejar Starvation en los discos?

**Elevator(a.k.a SCAN o C-SCAN)**

Esta solución es un algoritmo que se mueve 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 varias variantes de este algoritmo. **F-SCAN**, que congela la cola de pedidos cuando hace sweep, los pedidos que lleguen durante el Sweep van a otra cola para dsp.

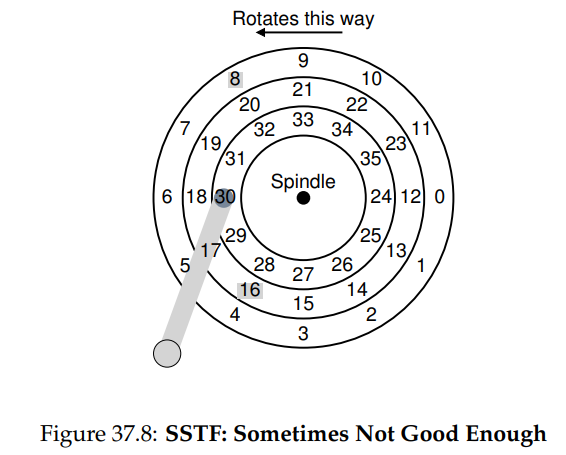
**C-SCAN**, circular scan. En vez de hacer sweep en ambas direcciones a través del disco, solo cambia desde el exterior al interior, y va de vuelta a la exterior directamente para empezar de nuevo. Esto es más justo para las pistas exteriores e interiores, SCAN solo favorecía las interiores visitandolas dos veces. Por sus características suele ser llamado **elevator algorithm**(algoritmo del ascensor).

Pero este no es mejor scheduler, no siguen el principio de SJF del todo, ignoran el tiempo de rotación. Ahora:

Cómo tener en cuenta el costo de la rotación?

**SPTF:Shortest Positionen Time First**

Al elegir el siguiente request a cumplir, cuál elegir? Depende del tiempo relativo del Seek comparado a la rotación. Si el seek time es más alto entonces SSTF sieve. Pero si el seek es un poco más rápido que la rotación, en el ejemplo convendría responder al pedido bloque 8 que al 16:



Frente a las variaciones entre seek time y rotation delay, SPTF sirve para mejorar la performance.

**Otros problemas de scheduling**

Dónde se hace el scheduling? En sistemas modernos los discos pueden acomodar múltiples pedidos, y tener sofisticados schedulers internos ellos mismos. Por ende el scheduler del OS solo le pasa varios pedidos que ve como las mejores opciones al scheduler del disco el cual, con el conocimiento de la posición del cabezal, etc. realiza el scheduling.

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

Por otro lado, cuánto debería esperar el OS antes de pasar un pedido de I/O a disco? El pasarlo tan pronto tenemos un pedido se llama **work-conserving**(el disco nunca para mientras haya pedidos). Sin embargo, la investigación sobre **anticipatory disk scheduling** ha demostrado que algunas veces es mejor esperar un poco, llamado **non-work-conserving**, al esperar puede que un nuevo y mejor pedido de I/O llegue, mejorando la eficiencia general.

**Chapter 39 Interludio: Archivos y Directorios**

En esta sección veremos una pieza al enigma de la virtualización: **persistent storage**(almacenamiento persistente). Un dispositivo de almacenamiento persistente como un **disco duro** o un **disco de estado sólido**(más moderno), almacena información de forma permanente(o al menos un largo tiempo), al contrario que la memoria. El Os debe tener mucho cuidado con dichos dispositivos. En los siguientes capítulos veremos técnicas críticas para manejar datos persistentes, centrándonos en métodos que aumenten la eficiencia/performance y la fiabilidad.

-Archivos y Directorios

Para la virtualización del almacenamiento se desarrollaron dos abstracciones clave. La primera, el **archivo/file**. Simplemente un array de bytes, donde cada uno puede ser escrito/leído. Cada archivo tiene algún tipo de **low-level name**(el nombre por el que lo identifica el OS), normalmente algún tipo de número. El usuario no suele conocer este low-level name, dicho nombre suele ser llamado **inode number**.

En la mayoría de sistemas, el OS no sabe la estructura del archivo(imagen, archivo de texto, etc.), la responsabilidad del **file system**(sistema de archivos) es almacenar dichos datos de forma persistente en disco para que estén disponibles al ser requeridos.

La segunda abstracción es el **directorio/directory**. Al igual que los archivos cada uno tiene un low-level name(nombre de bajo nivel/inode number), pero sus contenidos son específicos: contiene una lista de (nombre que ve el usuario, low-level name) pares. Al poner directorios en otros directorios el usuario puede crear un **árbol de directorios**( o **jerarquía de directorios**/ directory hierarchy), bajo la cual directorios y archivos son almacenados.

Dicha jerarquía comienza en un **directorio raíz/root directory** y usa algún tipo de **separador/**separatorpara nombrar los subsecuentes **sub-directorios** hasta el archivo o directorio deseado es nombrado. Siguiendo dicho nombre se puede obtener el **absolute pathname/ la ruta absoluta del archivo**. Por ejemplo (root)/Documentos(un directorio)/cuentas.txt(archivo). 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 suele ser el nombre del mismo y la segunda normalmente indica el **tipo** de archivo. Esto es solo una **convención**, no hay nada que asegure que un file.c contenga código de C. En esto se puede ver que el file system nos da una forma conveniente de **nombrar** todos los archivos que nos interesen.

-La interfaz del File System

Discutamos en detalle la interfaz. Empezaremos con lo básico de crear, acceder y borrar archivos.

-Creando Archivos

Lo más básico: crear un archivo. Esto se puede lograr con la sys call *open* y pasarle la bandera *O\_CREAT.* Con esto un programa puede crear un archivo, veamos un ejemplo:



Las banderas en este caso indican que si no existe lo cree(O\_CREAT), se asegura de que solo puede ser escrito(O\_WRONLY), y que, si el archivo ya existe, lo trunca a 0 bytes, borrando su contenido (O\_TRUNC). El tercer parámetro especifica los permisos que tendrá el archivo, haciendo que se pueda leer y escribir en el mismo por el creador.

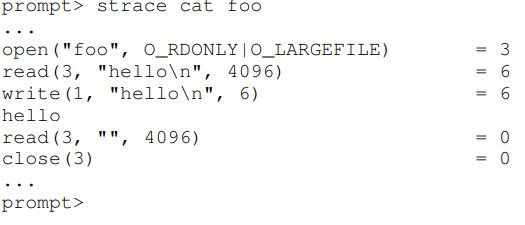
Un aspecto importante de *open* es el **file descriptor** que devuelve. Esto es solo un entero, hay uno privado por cada proc, y es usado en sistemas UNIX para acceder a archivos, por ende una vez un archivo es creado, se utiliza el file descriptor para leer/escribir en el archivo(si se tienen los permisos). De esta forma el fd(file descriptor) es una **capability/capacidad**, un poder de realizar ciertas operaciones. Otra forma de verlo es como un puntero que señala un objeto o tipo de archivo, una vez tenes dicho objeto se hacen llamadas a otros “métodos” para acceder al mismo(read(), write()).

Los file descriptors son manejados por el OS, uno por proc(al menos) en algún tipo de estructura simple que se mantiene en la estructura del proceso(en sistemas UNIX). Un simple array (con un máximo de archivos que pueden estar abiertos) mantiene la pista de cuantos archivos tiene abierto el proceso, así con cada proc. Cada entrada del array es un puntero a la *estructura del archivo* para ver info del mismo, si es escrito, leído, etc.

-Leyendo y Escribiendo en/de Archivos

Empecemos por la lectura de un archivo ya existente. Por ejemplo en una línea de comandos podemos usar el comando *cat*  para mostrar los contenidos del archivo. Cómo accede *cat* al archivo?

Podemos usar **strace,** herramienta de Linux para rastrear todas las system calls hechas por un programa durante su ejecución, e imprime dicho trazo en pantalla Veamos un ejemplo: el archivo “foo” tiene escrito “hello” en su interior, veamos que hace cat:



El archivo se abrirá sólo para lectura. Primero abre el archivo abre el archivo y se le devuelve un file descriptor => 3, porque 3? Porque cada proc ya tiene abiertos 3 fd, el 0(standard input, donde el proc recibe inputs), 1(standard output/ por donde el porc escribe para devolver información /resultados) y el 2(standard error, donde escribe mensajes de error). Con el fd 3 el OS sabe a qué archivo se refiere el proc. El primer *read* lee “hello”, luego usa el stdout(standard output) fd = 1 para escribir por pantalla lo que leyó. Al intentar leer nuevamente ve que ya no queda nada por leer, por ende cierra el archivo(*close(3)*, cierra el archivo indicando cual con el fd).

-Leyendo y escribiendo pero no secuencialmente

En el ejemplo anterior, todos los accesos fueron **secuenciales**. O bien se leyó un file de principio a fin o se escribió un file de principio a fin. Algunas veces es útil escribir o leer de un lugar determinado de un archivo, con un offset. Por ejemplo, si tenemos un índice a lo largo de un documento de texto y buscamos una palabra en específico, puede ser que terminemos leyendo de algunos offsets **random**. Para ejemplificar usaremos *lseek()* sys call:



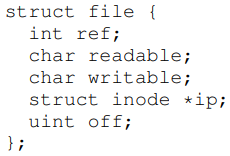
EL primer arg es el fd, el segundo es la posición en el archivo/el offset y el tercero determina cómo se realiza la búsqueda, de la siguiente manera:

-Si whence==SEEK\_SET, el offset se setea a offset bytes

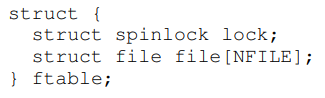
-Si whence==SEEK\_CUR, el offset se setea a su posición actual + offset bytes

-Si whence==SEEK\_END, el offset se setea al tamaño del archivo + offset bytes

Para cada archivo que se abre el OS rastrea su offset ”actual”, el cual determina donde se hará la siguiente lectura o escritura en el archivo. Parte de la abstracción de un archivo abierto es que tiene un offset actual, en cada momento dado se sat en un lugar del file. Dicho offset se actualiza de dos formas, cuando un read/write de N bytes ocurre, el offset pasa a ser N + current offset(current=actual), así cada read/write actualizan el offset implícitamente. La segunda es explicitamente con lseek() cambiando el offset como se explicó arriba. EL offset se guarda en las estructura del archivo,que puede ser algo así(versión simplificada de xv6):

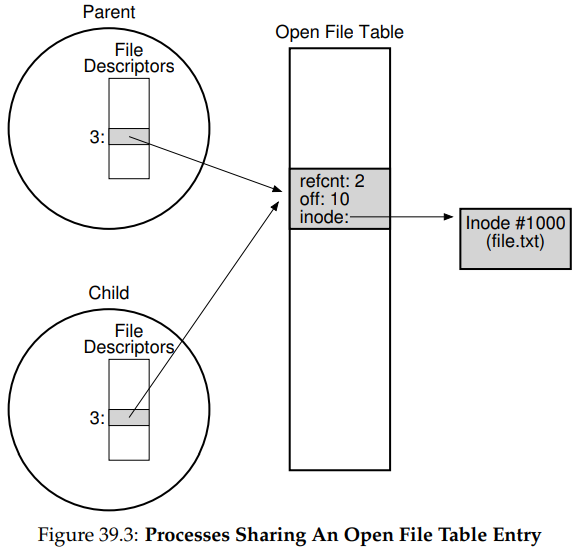


Con esto el OS determina si el archivo abierto se puede leer/escribir(uno o ambos), a que archivo se refiere/apunta(inode) y el current offset. Dichas estructuras representan todos los archivos abiertos, juntas algunas veces se les refiere como **open file table**(lista de archivos abiertos), esto se puede ver en una estructura:



-Entradas compartidas de la File Table: fork() y dup()

En muchos casos el mapeo de un fd a una entrada en la file table es en una relación 1-a-1. Incluso si más de un proc lee el mismo archivo al mismo tiempo, cada uno tendrá su propia entrada en la open file table. Lo que es lógico, cada lectura/escritura en un archivo es independiente y tiene su propio offset.

Hay algunos ejemplos interesantes de cuando un entry en la open file table es compartida, uno de esos es cuando un parent crea un child con **fork()**.El child ajusta su offset con lseek() y luego se crea. El cómo se comparte se puede ver en la sig imagen:

En esta se hace uso de la **reference count**(contador de referencias). Cuando una entry es compartida, sus referencias aumentan, sólo cuando ambos procesos cierren el archivo(o terminen) dicha entry será removida.

El compartir entry entre parent y child es útil ocasionalmente, por ejemplo si están trabajando cooperativamente en una misma tarea y requieren escribir en el mismo file.

Otro ejemplo interesante puede ser la **dup()** sys call(y sus primos dup2(), dup3() ), dup() le permite a un proceso crear un nuevo fd que se refiera al mismo archivo que ya está abierto. Esta syscall es útil cuando se escriben operaciones de redirección, como redirección de output, creando un fd que apunte al archivo deseado e intercambiando el stdout que tiene un proc por dicho fd.

-Escribiendo inmediatamente con *fsync()*

Cuando un proc llama a write() le pide al OS que escriba ciertos datos en almacenamiento persistente en algún momento del futuro(cercano en lo posible). El file system, por razones de desempeño, **amortiguara**(will **buffer**) dichos writes en memoria por algún tiempo, luego esos writes van a ser ejecutados por el dispositivo de almacenamiento. Aunque para la aplicación que llamó a write esta concluyó rápido, pero hay casos donde un crasheo puede llevar a la pérdida de dicha memoria si esta no llegó a escribirse en disco. Algunas aplicaciones no pueden permitirse esto por lo que hay protocolos de recuperación de datos que necesitan la habilidad de forzar escrituras a disco.

Para dichas aplicaciones la mayoría de file systems proveen alguna API(interfaz). En UNIX dicha interfaz es fsync(int fd). Cuando un proc llama fsync con un fd particular, el file system responde forzando todos los datos **sucios(dirty)**(todavía no escritos) al disco, al archivo especificado por el fd. fsync() retorna cuando los writes terminan.

-Renombrar archivos

Algunas veces es útil poder cambiarle el nombre a un archivo existente. En una linea de comandos esto se puede lograr con el comando *mv*, por ejemplo renombrar el archivo que usamos antes “foo” con el nombre “bar”: prompt> mv foo bar

Si usaramos *strace*, podríamos ver que se llama a la syscall *rename(char \*old, char \*new)*, que toma el nombre viejo y el nuevo que le queremos poner al archivo. Lo interesante es que rename es ejecutada, usualmente, de forma atómica(en cuanto a crasheos del sistema, si crashea el archivo quedará o con el nombre viejo o nuevo, nara entre medio). La syscall rename es crítica en ciertas aplicaciones que requieren actualizar el estado de un file de forma atómica.

-Obteniendo información sobre Archivos

Se espera que el sistema mantenga cierta información sobre cada archivo almacenado, se suele llamar a esta información sobre archivos **metadata**. Para ver esta info se pueden usar las syscalls stat() o fstat(). Las cuales toman un pathname o fd a un archivo y rellenan una estructura stat, en esta se puede ver información sobre el archivo como: tamaño/size; low-level nave/inode; ownership info e información sobre cuando este fue accedido modificado. La mayoría de los sistemas guardan esta info en una estructura llamada **inode**. Por ahora solo digamos que es una estructura de datos que mantiene el file system con la información antes dicha. Todos los inodes están en disco, y normalmente una copia en memoria de los que están en uso se mantiene para su rápido acceso.

-Removiendo Archivos

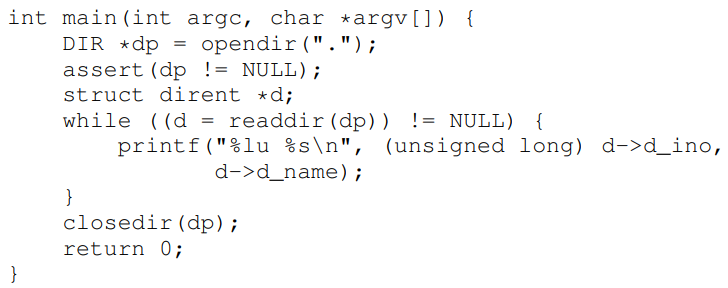
Cómo se borran los archivos? En UNIX podemos llamar el comando rm, pero que llama rm por detrás? Si usamos strace para verlo vemos , entre otras llamadas varias, que llama a *unlick()*, que toma el archivo a ser removido y devuelve 0 si tiene éxito. Luego comprenderemos porqué se llama unlink y no delete o remove.

-Creando directorios

Hay varias librerías relacionadas a los directorios que proveen syscalls que permiten crear, leer y borrar directorios. Por otro lado no se puede escribir directamente en un directorio, usar write() directamente escribiría en la metadata del directorio, la cual está protegida por el file system, el cual es responsable de la integridad de esta. Sólo se puede crear archivos, directorios u otro tipo de objetos dentro del directorio, “escribiendo” en él indirectamente. De esta forma el sistema se asegura de proteger la integridad de los directorios.

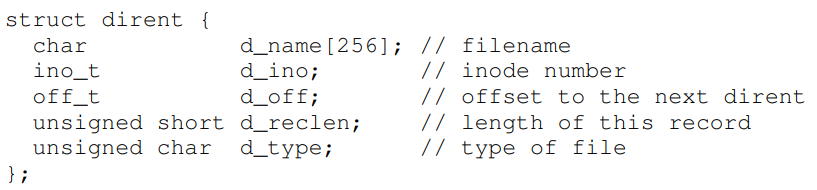
Con la syscall *mkdir()* podemos crear un directorio, una vez creado este se considera vacío, aunque tiene un mínimo de contenidos, específicamente dos entradas, si mismo y su parent(“padre” en el árbol de directorios).

-Leyendo directorios



Esto es exactamente lo que hace *ls*. Veamos cómo, en vez de abrir un directorio como si fuera un archivo se usan un grupo de llamadas, por ejemplo algo como la imagen de la izquierda.

El programa tiene 3 llamadas: opendir(), readdir() y closedir(). Se usan en un simple ciclo que lee entradas del directorio de a una hasta que esté vacío imprimiendo el nombre e inode de cada archivo que lee.



Veamos ahora la estructura de un directorio(struct dirent, una estructura de datos), la misma solo tiene un poco de información, básicamente mapea el nombre con el inode number de cada archivo que el directorio contiene y algunos otros detalles.

-Borrando directorios

Se puede borrar un directorio con *rmdir()*. Pero remover directorios es peligroso, al menos más que remover un archivo. Básicamente porque podemos borrar una gran cantidad de datos de una sola vez al borrar todo el contenido de este. Por esto rmdir requiere que previamente el directorio este vacío.

-Hard Links

Volviendo a la llamada unlink() para borrar archivos, para entenderla primero entendamos una nueva manera de crear una entry en el árbol del file system a través de la syscall link(). Esta toma dos argumentos, un pathname viejo y uno nuevo. Cuando hacemos link de un archivo viejo con uno nuevo básicamente creamos una nueva forma de referirnos al mismo archivo.

Link solo crea un nuevo nombre en el directorio al cual se está creando el link, y se refiere al mismo inode number que el archivo original. El archivo no se copia de ninguna forma, ahora hay dos nombres que el usuario puede leer que se refieren al mismo archivo.

Ahora se puede entender mejor a unlink(). Cuando se crea un file, se hacen dos cosas, primero se crea una estructura(el inode) que contiene la información relevante sobre el archivo. Segundo, se “linkea” (link(), ln()) un nombre que el usuario puede ver al archivo y se pone ese link en el directorio.

Para el sistema no hay diferencia entre un link hecho al archivo u otro, ambos referirán al mismo inode number. Para eliminar un archivo se llama a unlink(), esto funciona porque cuando el sistema “deslinkea” el archivo, chequea la **reference count** dentro del inode number. Este contador de referencias al archivo(algunas veces llamado **link count**) le permite al file system saber cuantos nombres están linkeado al este inode en particular. Luego el sistema elimina todas estas referencias que el usuario puede ver, finalmente libera el inode y todos los bloques de memoria asignados a este.

-Links Simbólicos

Hay otro tipo de link llamado **symbolic link**  o **soft link**, los hard links son un poco limitados: No se puede crear uno a un directorio(para evitar un ciclo en el árbol de directorios), no se puede hacer un hard link a archivos en otras particiones del disco(porque los inodes solo son únicos en cada file system, no a través de varios file systems). En estos casos se usa el symbolic link, su creación es muy parecida a la de un hard link y funciona de la misma forma, ahora podemos acceder a un archivo a través de dos nombres diferentes.

La diferencia es más notoria cuando no miramos la superficie. Un soft link es en realidad un archivo en sí mismo, de un tipo diferente. Ya hablamos de files y directorios, soft link es un tercer tipo del file system. La forma en la que un soft link está formado es por el pathname del archivo al que está linkeado en forma de datos dentro del archivo dentro del archivo soft link. Por la forma en que los soft links están creados se abre la posibilidad de una **dangling reference**(referencia colgante). Si se borra el archivo original el soft link no se borra, por lo tanto intentar abrirlo solo dirá que el archivo no existe.

-Bits de permiso y Control de Acceso a Listas

Igual que la virtualización del CPU y la memoria, el file system también presenta una forma virtual de ver el disco. Una más amigable que un array de bloques. Esta abstracción difiere de la del CPU y la memoria, en el hecho de que los archivos suelen ser compartidos por diferentes procesos y no siempre son privados(al contrario que los procesos que creen tener todos los CPUs y memoria para ellos). Por esto varios mecanismos más comprensivos son usados para que los file systems permitan compartir archivos.



El primero de estos mecanismos son los clásico de UNIX **permission bits**(bits de permiso). Para ejemplificar veamos:

Nos interesa la parte de -rw-r–r—. Estos bits dan información, el primer - muestra que es un archivo regular(d seria un directory, 1 un soft link). Pero los que nos interesan son los bits de permiso, los últimos 9 caracteres. Estos determinan quién puede acceder al archivo/directorio/softlink. Estos se dividen en tres grupos: lo que puede hacer el **duelo** del archivo, lo que alguien en un **grupo** puede hacer, y lo que cualquiera( normalmente llamado **other/otro**) puede hacer/ tiene permitido. En el ejemplo el file el dueño puede leer y escribir(rw- / read write -) y los otros dos grupos solo pueden leer.

El dueño del archivo puede cambiar estos permisos, con el comando chmod para cambiar el **file mode**/ modo del archivo. Puede remover los permisos de lectura a los demás por ejemplo. El último bit que está vacío en todos los casos es el de ejecución, indicando quién puede ejecutar el archivo( por ejemplo si este es código). En directorios el bit de ejecución se comporta diferente, le permite al usuario(o group, o other) hacer cosa como cambiar directorios dentro del directorio, y en combinación del bit de write el poder de crear archivos dentro del directorio.

Otros file systems tienen otros controles más sofisticados, como un **ACL(access control list)** por cada directorio. Son una forma más poderosa y general de representar exactamente quien accede a un determinado recurso. Por ejemplo, le da al usuario la capacidad de determinar quién puede o no leer un conjunto de archivos.

-Creando y montando un File System

Cómo ensamblar un árbol de directorios completo a partir de muchos sistemas de archivos subyacentes? Esta tarea se completa creando File Systems, y luego montandolos para hacer sus contenidos accesibles. Muchos file systems proveen una herramienta, a la que normalmente se refieren como mkfs, que hace exactamente esta tarea. La idea es la siguiente: darle a la herramienta, como input, un dispositivo(como la partición de un disco) y un tipo de file system, y simplemente escribe un file system vacío, empezando en el directorio **root**, en esa particion de disco. Una vez dicho file system es creado debe ser accesible dentro del file system tree más general. Esto se logra con el programa *mount*. El mismo toma el **mount point**/ punto de montura, un directorio ya existente, y pega un nuevo file system, nuestro nuevo file system, en el árbol de directorios en ese punto.

**Chapter 40 Implementación del File System**

Introduciremos la implementación de un simple file system, conocido como **vsfs**(the **Very Simple File System**), una versión simplificada del sistema de archivos de UNIX, por que que sirve para introducir algunas de las estructuras de tos más básicas del disco, métodos de acceso y varias políticas que están presentes en muchos file systems actuales.

El File System es puro software, no añadiremos ninguna pieza de hardware para hacer que algún aspecto funcione mejor(pero hay que tener en cuenta las características del dispositivo para asegurar que el file system funcione bien).

Cómo implementar un file system simple?

-The Way To Think

Para pensar sobre file systems conviene pensar en dos aspectos de estos, si los entendemos, entenderemos básicamente como funciona el file system.

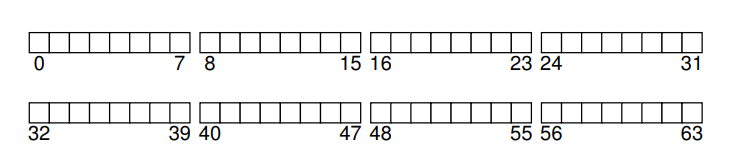
Primero las **estructuras de datos** en el file system. Qué tipos de estructuras on-disk(en disco) son utilizadas por el file system para organizar sus datos y su metadata. El primer file system que veremos emplea estructuras simples, como arrays the bloques u otros objetos(file systems más complejos usan estructuras del tipo árbol).

El segundo aspecto son los **métodos de acceso** del file system. Cómo mapea las llamadas hechas por los proc( open(),read(),write()) en las estructuras? Qué estructuras se leen/escriben en la ejecución de una syscall particular? Qué tan eficientemente se ejecutan estos pasos?

Entendiendo ambos aspectos nos damos una buena idea mental de cómo funciona el file system.

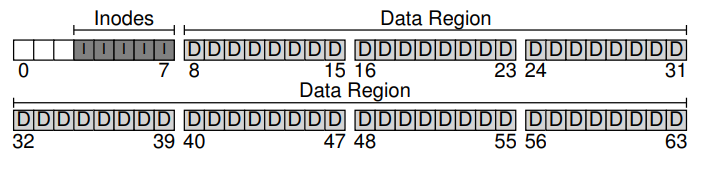
-Organización general

Ahora desarrollaremos la organización general de las estructuras de datos en disco del vsfs file system. Primero necesitamos dividir el disco en **bloques**, como los sistemas simples hacen usaremos un solo tamaño de bloque(4 KB). Dichos bloques están en una serie de 0 a N-1, en particiones del tamaño de N bloques de 4 KB. Asumamos un tamaño chico, N=64:



Qué necesitamos guardar en estos bloques? User data/datos del usuario, la mayoría de espacio ocupado en cualquier file system es user data. Llamemos a la región con estos datos **data region**/región de datos, y reservemos una cantidad de bloques para estos datos, digamos 56 de 64.

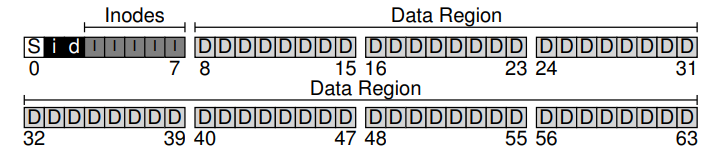
De las regiones en la imagen anterior serían todas menos de 0-7. La **metadata**, info que el file system guarda sobre cara file, será guardada como generalmente se hace en una estructura llamada **inode**. Esta estructura tmb necesita espacio reservado en disco. Digamos 5 bloques:



Los inodes no suelen ser muy grandes, por ejemplo 128 o 256 bytes. Asumiendo 256 bytes por inode, en un bloque de 4 KB entran 16, con 5 bloques serían 80 inodes. Este es el número máximo de files que podemos tener en nuestro file system(notar que el mismo file system en un disco más grande podría acomodar más inodes, por ende más archivos).

Nos falta algo con lo que ver si bloques de inodes o de user data están ocupados o libres. Algo como **allocation structures**(estructuras de asignación), necesarias en cualquier file system. Muchos métodos para seguir la pista de las asignaciones de bloques son posibles. Por ejemplo una **free list** que primero apunte al primer bloque libre, dsp al siguiente y así. En este caso usaremos una estructura conocida como **bitmap,** uno por región de datos, data region —> **data bitmap** y uno para la inode table—> **inode bitmap**.

Un bitmap es una estructura simple: cada bit se usa para indicar si el correspondiente objeto/bloque está libre(0) u ocupado(1). Usamos un bloque para cada bitmap dejando solo libre el bloque 0(pensando en la imagen de más arriba). Un bitmap de este tamaño podría ver hasta 32 K objetos asignados y solo tenemos 80 inodes y 56 data blocks, sin embargo usamos un bloque por bitmap por simplicidad. Guardamos el último bloque libre para el **superblock**(superbloque), este contiene la información sobre este file system particular, como la cantidad de inodes y data blocks, donde empieza la inode table, etc. También puede incluir un número mágico y alguna clase de identificador del tipo del file system. El diagrama queda algo asi:

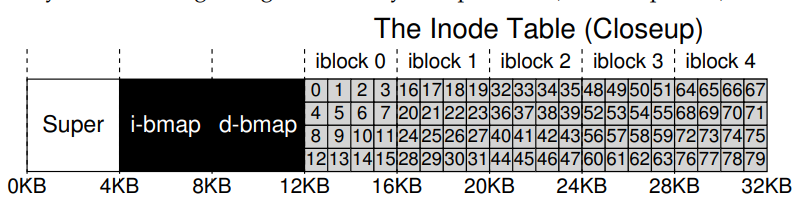


Cuando se monte el file system, el OS va a leer el superblock primero para inicializar varios parámetros y luego añadir el volumen entero al file-system tree.

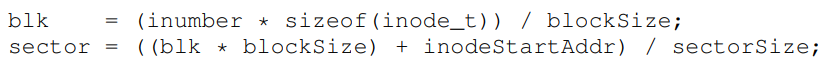
-Organización de Archivos: El Inode

**Inode**, una de las estructuras en disco más importante de un file system, virtualmente todos los file system tienen una estructura similar a esta. El nombre viene 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 cap anterior. En nuestro vsfs, dado un i-number, podemos directamente calcular donde en el disco está el inode al que se refiere. La tabla de inodes son 20KB( 5 bloques de 4KB), 8n inodos, dicha tabla empieza en los 12KB(4 bloque, el primer bloque es el superblock, y los siguientes dos los bitmap), sería algo así:



Para leer el inode 32, el file system primero calcula el offset en la región de inodos(32 . inode\_size(256) = 8192), lo suma a la dirección inicial de la inode table(12KB) y llega al bloque deseado de inodes (20KB block). Los discos no son direccionables por byte, sino por sectores, normalmente de 512 bytes. Por ende para señalar en el bloque de inodes que contiene el inode 32, el file system leerá el sector ((20x1024)/512) o 40, luego de tener el sector de 512 bytes donde esta el inode se calcula de la siguiente manera:



Dentro del inode tenemos toda la información del file(permisos, tamaño, dueño, cuando se creo, cantidad de bloques asignados, etc), dicha información es la **metadata**. En realidad cualquier info en el file system que no es user data es llamada metadata generalmente.

Una de las decisiones más importantes en el diseño del inode es como referiste a donde están los bloques de datos. Un enfoque simple sería tener uno o más **direct pointers**(disk addresses) (punteros director(direcciones de disco)) dentro del inodo; cada puntero se refiere a un bloque que pertenece al file. Dicho enfoque es limitado, un archivo muy grande(más grande que el tamaño del bloque multiplicado por el número de punteros directos en el inode) no funciona.

**La indexación de múltiple-nivel**

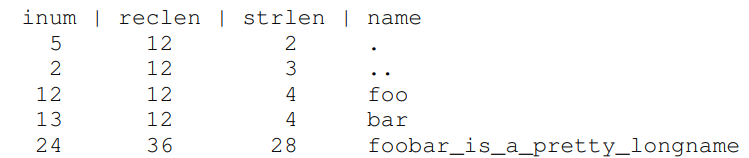
Para dar soporte a archivos más grandes se introdujeron diferentes estructuras en los inodos. Una común es tener un puntero especial, un **indirect pointer**. En vez de apuntar a un bloque con user data apunta a uno con más punteros, donde cada uno apunta a user data. Si un archivo se pasa del límite se usa un bloque de la data region como bloque indirecto y un indirect pointer del inode apunta a este. Con bloques de 4 KB y 4 byte disk addresses, eso añade 1024 punteros, el archivo puede crecer (12+1024). 4KB o 4114KB. En este enfoque puede pasar que se necesite dar soporte a archivos incluso más grandes, al introducir otro puntero al inode surge el **double indirect pointer**. Este puntero apunta a un bloque que tiene punteros a bloques indirectos, los cuales contienen punteros a data blocks. Puede direccionar archivos de más de 4 GB. Si se quiere más se puede hacer un **triple indirect pointer**. Este árbol imbalanceado es a lo que nos referimos con enfoque **multi-level index**(indexación de múltiple-nivel).

Este enfoque se usa mucho en muchos sistemas de archivos, por qué? Con lo desbalanceado que esta. Con el tiempo y las investigaciones se vio que la gran mayoría de archivos son pequeños, con optimizar este enfoque alcanza, no hace falta otro diferente o no vale la pena.

-Organización de Directorios

En vsfs(y en muchos file systems), los directorios tienen una organización simple; un directorio básicamente solo contiene una lista de(entry name, inode number) pares. Por cada file o directorio en un directorio, hay un string(de hasta cierto tamaño) y un número en el data block(s) del directorio.

Por ejemplo veamos un directorio *dir*  con 3 archivos, sus datos en disco se verían así:



Se ve el inode number(inum), la longitud del nombre en bytes y el espacio sobrante(reclen), la longitud del string(strlen) y el nombre(name). Las otras dos entradas “ . “ y “. . “ son el directorio actual y el parent(root en este caso) respectivamente.

Dónde se guardan los directorios? Normalmente, los file systems tratan a los directorios como un tipo especial de archivos. El inode de un directorio, en algún campo del inode que especifique el tipo de archivo, está marcado como “directory” en vez de “regular file”. El dir(directorio) tiene data blocks señalados por el inode(y tal vez bloques indirectos); estos bloques de datos están en la data region(en nuestro sistema). Notar que aunque usemos una lista de entryes en este caso, no es la única estructura de datos posible.

-Manejo del espacio libre

En nuestro sistema tenemos dos bitmaps para el manejo del espacio libre, es decir siguen la pista de que inodos y bloques de datos están libres y cuales asignados, así cuando se crea un file o dir le puede encontrar espacio. Al crear un archivo, se le asigna un inode, el sistema debe buscar un inode libre, asignarlo, marcarlo como usado(1) y luego actualizar el estado de los inodes en disco, la creación de un dir es similar.

-Access Paths: Reading and Writing

Cómo es el flujo de operaciones al leer/escribir en un archivo. Entender que pasa en estos **access path**(ruta de acceso) es importante para entender cómo funciona un file system. Para los siguientes ejemplos asumamos que el file system ya se montó y por ende el superblock ya está en memoria(lo leyó el OS). Todo lo demás(inodes,directorios,etc.) sigue en disco.

**Leyendo un archivo del/desde el disco**

Queremos abrir un file(ej /foo/bar), leerlo, y cerrarlo, el file es de 12 KB( 3 bloques).

Cuando se llama a open(“/foo/bar”, O\_RDONLY), el file system primero necesita encontrar el inode de bar, para obtener información del archivo(permisos,tamaño,etc.). Pero solo tiene el path name, para encontrar el inode debe **atravesar** el path name.

El atravesarlo siempre comienza en root, en **root directory**(/.), primero se lee el inode de root. Para esto necesita el i-number del inode, el del root debe ser conocido por el file system cuando es montado. En la mayoría de sistemas UNIX el i-number del root es 2. El FS(file system) lee el inode 2.

Luego el FS mira el inode de root, busca punteros a data blocks, los cuales contienen los contenidos de root dir. Con estos el FS lee el directorio y busca el siguiente componente del path name, “foo”. Luego de encontrarlo debe buscar el inode de foo(digamos 44). Luego recursivamente de atraviesa el path name de esta forma hasta que se llegue al inode deseado. En foo están sus entradas, entre ellas finalmente bar. El paso final de open() es leer el inodo de bar en memoria; el FS hace el check de permisos, asigna un file descriptor para este proceso y retorna al usuario. Luego se llama a read() para leer el archivo, a lo largo de la lectura se va actualizando el offset y el last time access en el inode, el cual empieza en 0(leyendo desde el comienzo del archivo, salvo que se use lseek() ) leyendo el primer bloque del archivo, consultando el inode para ver dónde está dicho bloque, y así con cada bloque del file.

En algún punto se cierra el archivo, el FS solo debe desasignar el espacio que le dio al file descriptor y nada más, no toma lugar ningún I/O. Y listo. Notar que la cantidad de I/O que hace el open() depende de lo largo del path name, mientras más largo, más inodes debe buscar y leer.

**Escribiendo a un Archivo en Disco**

Es similar a leer, primero se debe realizar un open() como del de antes. Luego la aplicación puede realizar una write() call para actualizar los contenidos del archivo con los nuevos. Escribir en un archivo puede requerir **asignar** un nuevo bloque(a menos que se sobreescriba en uno). Cuando se escribe en un nuevo archivo, cada write debe no solo escribir datos en disco sino primero decidir qué bloque asignar al archivo y por ende actualizar otras estructuras del disco de forma acorde. Cada write genera I/Os: uno para leer el data bitmap(para encontrar un bloque libre),otro para escribir en el bitmap(actualizarlo para marcar el bloque que uso como asignado), dos más para leer y luego escribir el inode(en el cual se actualiza la localización del bloque), y finalmente uno más para escribir los datos en el bloque en sí. El costo es todavía más alto cuando se crea un archivo, aparte hay que buscar espacio en el directorio para asignar una entry, etc. El resto del proceso queda en el apunte.

Cómo reducir el costo de I/O del FS?

-Caching and Buffering

(almacenamiento en cache y en buffers)

El alto costo de I/O afecta el desempeño enormemente. Sistemas primitivos implementaron un **fixed-size cache**(un cache de un tamaño fijo) para mantener bloques populares, generalmente a este cache se le asignaria espacio en tiempo de booteo.

Este **particionamiento estático** de memoria puede causar un desperdicio, y si se guarda el 10% de memoria para el cache y no se necesita el 10% todo el tiempo. Los sistemas modernos al contrario emplean **particionamiento dinámico**. Específicamente muchos sistemas modernos integran páginas de memoria virtual en un **cache de páginas unificadas**. De esta forma la memoria puede ser asignada más flexiblemente. El uso de este cache es más útil en la lectura que en la escritura, ya que la lectura no necesita un manejo individual para pasar datos al disco de forma persistente y los efectuar un write si. Frente a esto se usa **write buffering**, que posee ciertas ventajas. Primero, al retrasar writes(), el FS puede juntar en un lote varias actualizaciones disminuyendo los I/Os. Segundo al almacenar en un buffer en memoria un número de writes(), el **scheduler** del sistema puede encargarse de los subsecuentes I/Os e incrementar el desempeño. Finalmente el retrasar algunos writes() puede evitarlos completamente, por ejemplo si un proc crea un file, le escribe algo y lo borra.

**Chapter 41 Locality and The Fast File System**

Cuando el UNIX OS se introdujo por primera vez tmb tenía un file system, el “old UNIX file system” llamémoslo, el cual era muy simple. Sus estructuras de datos en el disco eran algo así:



El superblock(S) contenía la información de todo el file system: tamaño del volumen, cuantos inodes hay, un puntero a la cabeza de una free list(de los bloques), etc. Luego la región con todos los inodes del sistema y finalmente la parte más grande, los data blocks. Lo bueno de este sistema es que era simple y proveía las abstracciones necesarias: archivos y una jerarquía de directorios.

-El problema: Pobre desempeño

La performance era terrible, llegó al punto en que sólo entregaba el 2% del ancho de banda del disco. El problema es que trataba al disco como si fuera una random-access memory(memoria de acceso aleatorio/ RAM), los datos estaban esparcidos por todos lados sin tener en cuenta que el medio que almacenaba estos datos era un disco, y por ende el posicionamiento de los datos costaba mucho.

Este file system terminaba **fragmentado**, ya que el espacio libre no se manejaba con cuidado. Teniendo los bloques de un file esparcidos por el disco implicaba que la lectura de estos sea una búsqueda constante disminuyendo la performance(la lectura de un mismo archivo no era secuencial). La herramienta de **desfragmentación** lidia con este problema, reorganiza los bloques de los archivos de forma continua al igual que con los bloques libres.

Otro problema: el tamaño original de los bloques era muy pequeño(512 bytes) por ende, la transferencia de datos era muy ineficiente. Aunque los bloques pequeños tenían el lado bueno de disminuir la **fragmentación interna**(desperdicio de espacio dentro de cada bloque). Entonces:

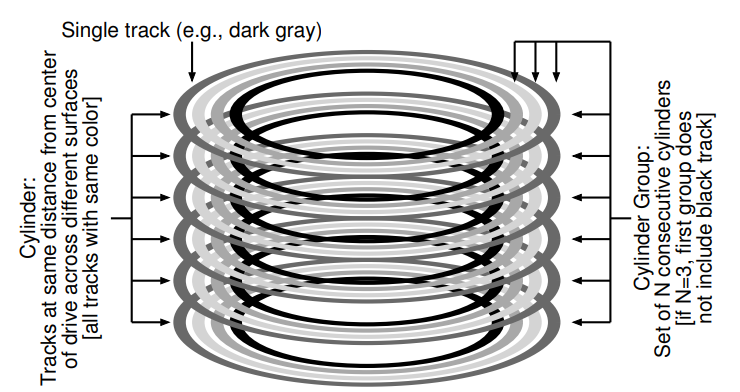
Cómo organizar los datos en el disco para mejorar la performance?

-FFS: Disk Awareness Is the Solution

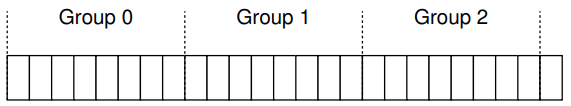
Un grupo en Berkeley decidió hacer otro file system que llamaron **Fast File System(FFS)**(el file system rapido), La idea era diseñar las estructuras del file system y las políticas de asignación de espacio de forma que sean “disk aware”(conscientes del disco) y por ende mejoran la performance. Esto inició una nueva era en la investigación de los file systems: al mantener la misma interfaz en el file system(La misma API con open(),read(),write(), close(), etc.) pero cambiando la implementación interna, esto sigue hasta hoy en día. Virtualmente hoy todos los file system se adhieren a la interfaz ya existente pero cambian su implementación interna por performance, fiabilidad, y otras razones.

-Estructura organizativa: The Cylinder Group

El primer paso fue cambiar las estructuras en disco. FFS divide el disco en un número de **cylinder groups**(grupos de cilindros). Un solo **cilindro** es un conjunto de pistas en diferentes superficies de un disco duro que están a la misma distancia del centro. FFS agrega N cilindros consecutivos en un grupo, de esta forma todo el disco puede ser visto como una colección de cilindros. Veamos un ejemplo que muestra las cuatro pistas de la parte más exterior del disco, un disco de 6 platos, y un grupo de cilindros que consiste de 3 cilindros:



Notas que los discos modernos no exportan suficiente información al file system como para que esté realmente entienda si un cilindro particular está en uso, los discos exportan address space lógico de los bloques y esconden los detalles de su geometría a los clients/clientes. Por eso sistemas modernos organizan el disco en **grupos de bloques**, cada uno de una porción consecutiva del address space del disco, por ejemplo:



Ya sea con cilindros o grupos de bloques se asegura que si se colocan dos archivos en el mismo grupo, FFS se asegura que si accede a uno después de otro no va a haber largas búsquedas a través del disco. Para almacenar files y dir en estos grupos, el FFS necesita tener la habilidad de poner dir en un bloque y seguir la pista de toda la info necesaria sobre ellos. Para esto incluye las típicas estructuras de un file system en cada grupo. Espacio para inodes, data blocks y estructuras para seguirles la pista a los lugares libres y asignados, esto es lo que FFS mantiene dentro de un grupo de cilindros:



Examinemos los componentes en más detalle. Mantiene una copia del superblock en cada grupo por razones de fiabilidad. El S es necesario para montar el sistema, si una copia se corrompe hay respaldos. Para el manejo del espacio libre/asignado usa un **bitmap** por región(inodes/data). Finalmente las regiones **inode** y **data block** son como las del sistema simple descrito en el cap anterior. Cada grupo de cilindros está compuesto en su mayoría de data blocks.

-Políticas: Como Asignar Archivos y Directorios

Ahora el FFS debe decidir cómo posicionar files y dirs y la metadata asociada para mejorar la performance. La idea es la siguiente: *mantener cosas relacionadas juntas(y cosas no relacionadas separadas).*

El FFS debe decidir que está relacionado y ponerlo en el mismo grupo de bloques, para esto usa algunas disciplinas. Primero el posicionamiento de directorios, usa un simple enfoque:encontrar el grupo de cilindros con un bajo número de directorios(para mantener un balance) y un alto número de inodes libres, y poner los datos del directorio y su inodo en ese grupo.

Para los archivos se hacen dos cosas. Primero, el FFS se asegura(en general) de asignar los bloques de un archivo en el mismo grupo que su inode(para prevenir largas búsquedas). Segundo, posiciona todos los archivos que están en un directorio en el mismo grupo de cilindros en el que está dicho directorio.

Estas son dos ventajas del FFS policies, un file est cerca de su inode y los files en un dir están cerca unos de otros. Estos enfoques que toma el FFS más que en estudio sobre el flujo de los file systems están basados en **sentido común**.

-La excepción del Archivo-Grande

En FFS hay una excepción importante en la política general de posicionamiento de archivos, esta surge con los archivos muy grandes. Sin una regla que lo diferencie tal archivo llenaría un grupo de bloques completo, esto no es deseable. Por ende para archivos grandes FFS hace lo siguiente. Después de que cierto número de bloques son asignados en el primer grupo de bloques, FFS posiciona el siguiente “gran” pedazo del archivo en otro grupo de bloques(tal vez elegido por su bajo uso), y así subsecuentemente hasta almacenar todo el archivo.

Se puede pensar que esta separación de las partes puede afectar a la performance, pero este problema se puede abordar eligiendo el tamaño de esas arte con cuidado. Si el tamaño es suficientemente grande(de cada parte) el file system demorara la mayor parte del tiempo transfiriendo los datos y solo un pequeño(relativamente) tiempo en buscar la siguiente parte del archivo. Este proceso de reducir el overhead haciendo más trabajo por cada “paid overhead” se llama **amortization** y es una técnica común.

Otros aspectos de FFS

Introdujo varias innovaciones, en particular los diseñadores se preocupan de la fragmentación interna al almacenar pequeños archivos (digamos 2KB o así) en bloques de 4 KB. Esto podía llevar a desperdiciar mucho disco. Los diseñadores de FFS introdujeron **sub-bloques** solucionando el problema, los cuales eran de 512 bytes, en los que el file system podía almacenar archivos. Mientras el archivo crecía le asignaban más sub-bloques, cuando ya ocupaba 4 KB, el sistema liberaba los sub bloques, luego de copiarlos juntos en un bloque de 4 KB.

Para evitar el trabajo extra que esto suponía(extra I/Os), FFS generalmente eludia este comportamiento modificando la librería *libc*, la librería almacenaría en un buffer escrituras/writes y después los haría en bloques de 4 KB, en varios casos evitando la creación de sub-bloques innecesarios.

Otra cosa que introdujeron fue un limpio diseño del disco(disk layout) que fue optimizado para aumentar su desempeño, en ese tiempo los discos eran mucho menos sofisticados y requieren del CPU para controlar sus operaciones. Principalmente durante las lecturas secuenciales, FFS introdujo un disk layout, que al saltarse bloques era lo suficientemente rápido para requisitar secuencialmente los bloques a leer antes de que pasen por el cabezal del disco, evitando rotaciones extra. FFS logró calcular cuántos bloques se deben saltar en el layout para evitar estas rotaciones, esta técnica se llamó **parametrización.** Los sistemas actuales ya no se preocupan de estos detalles de bajo nivel, las abstracciones de interfaces de alto nivel hacen su trabajo(si están diseñadas bien).

Otra mejora al nivel usuario es el FFS permitió **nombres más largos en los archivos**(más de 8 char como podría tener un enfoque de nombres de tamaño fijo). También se introdujo un nuevo concepto llamado **symbolic link**, el ya explicado en el capítulo anterior. FFS también introdujo el *rename()* atómico.

**Chapter 42 Crash Consistency: FSCK and Journaling**

Ya vimos las estructuras de datos que maneja un file system para implementar sus abstracciones, es decir archivos, directorios y otra metadata. Al contrario que las demás estructuras de datos, las del file system deben ser **persistentes**. Deben sobrevivir a largo plazo, almacenadas en dispositivos que retienen datos a pesar de la pérdida de energía/poder(que se apaguen).

Un desafío para los file system es cómo actualizar dichas estructuras y que sean persistentes a pesar de una **pérdida de energía** o **crasheo del sistema**. Específicamente si justo cuando se estaban actualizando las estructuras en disco, por ejemplo, se apaga la máquina/crashea el sistema por un bug. Este problema en la implementación de los file system es conocido como **crash-consistency problem**(problema de consistencia en los crasheos).

El problema es simple de entender, por ejemplo si dos requisitos A y B llegan y quieren escribir en disco en orden, para cumplir cierta operación, ya que solo puede ser una a la vez, el sistema tomará alguna, digamos A, y dejará B para después. Si en el momento en el que termina A y está por responder al pedido B hay un crasheo la estructura en disco quedaría en un estado **inconsistente**.

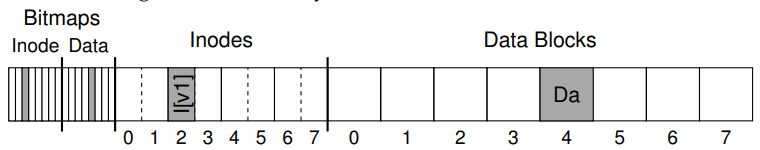
Cómo subir los datos al disco a pesar de los crasheos?

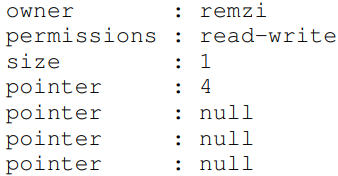
Veremos este problema en detalle y algunos métodos que usaron los FD para solucionarlo. Empezaremos por un enfoque de los sistemas más viejos, conocido como **fsck** o **file system checker**. Luego veremos otro, **journaling**(tmb conocido como **write-ahead logging**), una técnica que añade un poco de overhead/ gastos generales a cada escritura pero se recupera más rápido de un crasheo/pérdida de energía. Veremos la maquinaria básica de journaling.

-Un ejemplo detallado

Usaremos una **carga de trabajo/workload** que actualiza estructuras de disco de alguna forma. Esta será simple, adjuntar un data block a un archivo existente. Se logra abriendo el file, llamando a lseek() para mover el file offset al final del file, luego un 4 KB write al file antes de cerrarlo.

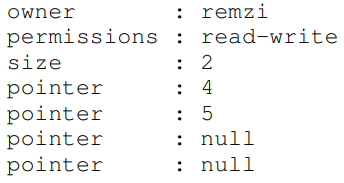
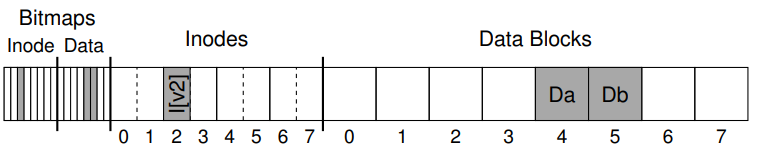
Asumimos estructuras de datos estándar. Este ejemplo incluye un **inode bitmap**(8 bits, 1 por inode) y un **data bitmap**(8 bits, uno por bloque de datos). Los indodes(son 8, van de 0-7, esparcidos en 4 bloques) y data blocks (son 8, van de 0-7), nuestro file system:



Solo está asignado el inode 2, y un solo bloque, el 4(marcados cada uno en su bitmap). I(v1) es la primera versión de este inode, pronto se actualizara. Veamos el inode: —--->

Su tam es 1(1 bloque), su puntero apunta al bloque 4(primer data block del file). Cuando le adjuntamos un bloque nuevo debemos actualizar 3 estructuras

en disco: inode(debe apuntar al nuevo bloque y actualizar el tamaño del file), el nuevo data block(data region structure), y el bitmap de los data block. El resultado es algo así:



Para este proceso completo el file system debe realizar 3 writes en disco, uno a cada estructura. Estos no suelen suceder inmediatamente, generalmente el dirty inode, bitmap y nuevos datos esperaran en la memoria principal(**page cache** o **buffer cache**), luego cuando el FS decide escribirlos en disco(dsp de 5 o 30 seg), el FD realiza el call write(). Pero un crasheo puede tomar lugar. Específicamente después de que 1 o 2 de estos writes ocurran, pero no los 3, el FD puede quedar en un “funny state”.

**Crash Scenarios**

Imaginemos casos, primero, solo un write se completa:

* **Solo el data block(Db) se actualizó**. En este caso los datos están en disco, pero ningún inode apunta a ellos y el bitmap no dice que el bloque esté asignado. Es como si nunca hubiera sucedido este write. Desde el punto de problema de consistencia en los crasheos, esto no es problema(aunque puede ser molesto para el usuario).
* **Solo el inode(I(v2)) se actualizó**. El inode apunta a 5 donde el Db iba a estar, pero no está todavía. Si confiamos en este puntero leeremos datos **basura** del disco. Pero este caso si es un problema de consistencia de crasheos. El bitmap dice que el block 5 está libre/no asignado y el inode dice que si, esta inconsistencia en las estructuras de datos debe ser resuelta.
* **Solo el bitmap se actualizó**. Ahora indica que el bloque 5 está asignado, pero no hay inode que apunte a este. Una inconsistencia de nuevo, este write resultará en un **space leak**(pérdida de espacio) ya que el bloque 5 nunca será usado por el FD.

Hay otros tres casos resultantes de plantear que solo dos writes se ejecutaron bien:

* **El inode y bitmap actualizados**. La metadata es consistente: el inode apunta a 5 y bitmap lo marca como asignado(al bloque 5). El FD ve todo bien, pero el bloque 5 son **datos basura**.
* **El inode y data block(Db) actualizados**. El inode apunta a 5 y los datos están ahí, pero el bitmap no marca el bloque como asignado, inconsistencia de nuevo.
* **El bitmap y data block(Db) actualizados**. De nuevo inconsistencia, entre el inode y bitmap. De todas formas, aunque el bloque fue escrito y se indica que está en uso, no sabemos a qué file pertenece, ningún inode lo señala.

**The crash consistency problem**

Se pueden ver los problemas que pueden ocurrir, estructuras de datos inconsistentes, darle datos basura al usuario, etc. Lo que queremos hacer es mover el Fd de un estado consistente a otro **atómicamente**. Pero el disco solo hace un write al mismo tiempo, así que no podemos si un crasheo ocurre entre los writes, este es el **crash consistency problem**.

-Solución 1: The File System Checker

FD primitivos tomaron un enfoque simple, dejaron que sucedan inconsistencias y luego arreglarlas. Un ejemplo de este enfoque vago se encuentra en una herramienta que hace esto: **fsck**. Es una herramienta de UNIX para encontrar inconsistencias y repararlas. Notar que este enfoque no soluciona problemas, solo trata de conseguir un estado consistente en el FD. Esta herramienta opera en fases. Se ejecuta antes de que el FD se monte y esté disponible, una vez termina, el estado del FD debería ser consistente. Veamos básicamente que hace:

* **Superblock**: chequea que se vea razonable, básicamente con chequeos de sensatez, como que el tamaño del FD sea mayor a los bloques asignados. Si determina que el superblock está corrupto el sistema(o administrador)puede que decida usar una copia alternativa de este.
* **Free blocks**: escanea los inodes, indirect blocks, double indirect blocks, etc. para construir un entendimiento de que bloques están asignados en el Fd. Usa este conocimiento para producir una versión correcta de la asignación de los bitmaps, si hay alguna inconsistencia confía en la info de los inodes.
* **Inode state**: cada inode es chequeado por corrupción u otro problema. Como que su campo type sea válido. Si hay problemas con los campos del inode que no se arreglan fácil es visto como sospechoso y despejado por el fsck, tmb actualiza el bitmap acorde.
* **Inode links**: tmb se verifica la cuenta de links de cada inode. Esto indica la cantidad de directorios que tienen una referencia a este archivo. Para verificar la cuenta escaneo el árbol de directorios desde root. Si hay discrepancias generalmente se arreglan actualizando la cuenta del inode. Si hay un inode asignado pero no hay referencias a él en los directorios se mueve este al directorio *lost+found*.
* **Duplicados**. Tmb chequea punteros duplicados, dos inodos apuntando al mismo lugar. Si un inode se ve que es malicioso es despejado. Alternativamente puede que se de una copia del bloque apuntado a cada inode(actualizando los punteros).
* **Bad blocks**: se chequean punteros a bloques malos escaneando la lista de todos los punteros. Un puntero es considerado malo si apunta fuera de su rango permitido(un bloque más grande que la partición). En este caso solo se remueve el puntero del inode o indirect block.
* **Directory checks**: fsck no comprende el contenido de los user files, sin embargo los directorios contienen información creada por el FD. Por ende se chequea la integridad del contenido de cada directorio viendo que “.” y “..” sean las primeras dos entradas, que cada referencia a un inode esté asignada, y verificando que ningún directorio esté conectado a más de uno en la jerarquía de directorios.

Uno de los principales problemas de este método es que es muy lento. Con un disco muy grande esto puede tomar mucho tiempo, muchos minutos u horas.

-Solución 2: Journaling(or Write-Ahead Logging)

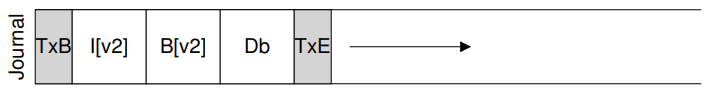
Posiblemente la solución más popular. La idea es la siguiente: Cuando se actualiza el disco, antes de sobreescribir las estructuras en su lugar, primero escribir una pequeña nota(en algún lugar del disco ya prefijado), describiendo que se está por hacer, esta es la parte llamada write ahead, se la escribimos a una estructura que organizamos como un “log”(registro). Al escribir esta nota garantizamos que dsp de un crasheo(en medio de nuestras operaciones actualizando el disco) se puede leer la nota e intentar de nuevo, se sabrá exactamente qué debe arreglarse(y cómo), en vez de escanear todo el disco. Por su diseño journaling añade un poco de trabajo durante las actualizaciones pero reduce mucho el tiempo de una recuperación.

Describiremos ahora como **Linux ext3,** un FD con journaling popular, incorpora esto en el FD. El disco está dividido en grupos de bloques, cada grupo con un inode bitmap, data bitmap, inodes y data blocks. La estructura nueva es el journaling en sí mismo que ocupa un pequeño espacio en la partición o en otro dispositivo.Algo así:



**Data Journaling**

Veamos un ejemplo de cómo funciona. Digamos que tenemos la misma update a disco que dijimos antes, deseamos escribir el inode(I(v2)), bitmap(B(v2)) y el data block(Db) en disco. Antes de escribir esto en disco los escribimos en el log/registro. El log se ve algo así:



Tenemos 5 bloques. La comienzo (TxB) nos cuenta sobre el update, incluyendo info como dirección final de lo que será escrito y algún tipo de **transaction identifier(TID)**(ID de la operación). El contenido de los 3 bloques de en medio solo contiene el contenido mismo de los bloques, esto es conocido como **physical logging**, ya que ponemos el contenido físico exacto de la update en el journal. EL TxE marca el final de esta operación y tmb contiene el TID.

Una vez dicha operación está en disco, podemos seguir con el pedido de write en sí, actualizar las data structures en disco. Este proceso se llama **checkpointing**. Por ende para hacerle un **checkpoint** al FD(ponerlo al día con los datos escritos en la operación en el log), hacemos los writes de I(v2), B(v2), Db a sus localización en el disco, si terminamos con éxito, hicimos checkpoint exitosamente.

Las cosas se ponen más complicadas con un crasheo en medio de las escrituras al journal. En esta parte queremos escribir el set de bloques ya señalado en la última imagen. Una simple forma de hacerlo es uno a la vez, esperando que termine uno antes del siguiente. Esto es lento, lo ideal sería escribir los cinco a la vez, esto convertiría cinco writes en un write secuencial y más rápido. Pero es menos seguro: por ejemplo el disco puede dividir el big write en partes y con su scheduler ejecutarlas en el orden que le parezca dejando una operación/transacción inválida que podría causar que en un reboot el FD no se pueda montar.

Para evitar este problema el FD hace el transaccional write en dos pasos. Primero todos los bloques menos el TxE en el journal, cuando este termine se escribe el TxE. Una parte importante es garantizar que sea atómico, sabemos que el disco garantiza que cualquier 512 bytes write es atómico, asegurándonos que el TxE es atómico. Nuestro protocolo tiene tres partes

1. **Journal write**: escribir los contenidos de la transacción en el log, esperar que terminen.
2. **Journal commit**: escribir el transaction commit block en el log, esperar que termine, se dice que la transacción fue **committed(encomendada)**.
3. **Checkpoint**: escribir los contenidos de la update(metadata y data) a su lugar final en disco.

**Recovery**

Entendamos como el FD usa los contenidos del journal para recuperarse de un crasheo, que puede suceder en cualquier parte de la secuencia. Si el crash es antes de que la actualización pendiente se escriba en el log(antes del paso dos), se saltea/ignora. Si pasa después pero antes de que el checkpoint se complete, el FD debe **recuperar** la update de la siguiente forma. Cuando el sistema bootea, el proceso FD recovery va a escanear el log y buscar transacciones encomendadas en el disco; dichas transacciones son **repetidas**(en orden), con el FD tratando de escribir los bloques de la transacción en su lugar final en el disco. Es una de las formas más simples de logging llamada **redo logging**. De esta forma el FD se asegura de la consistencia de sus estructuras. Notar que un crasheo puede ocurrir durante esta secuencia, lo cual solo repetiría todo el proceso nuevamente con estos writes(que en realidad son los mismos).

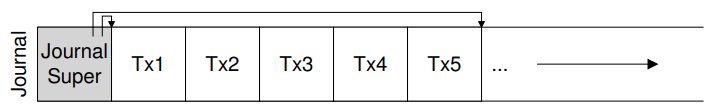
**Batching Log Updates**

Se puede pensar que esto le añade mucho tráfico al disco. Frente a esto varios sistemas no hacen un commit de cada update al disco, sino que se puede juntar en un buffer para hacer una global evitando writes excesivos.

**Making The Log Finite**

El log tiene una estructura finita, si le seguimos añadiendo transactions se llenará. Cuando esto pasa surgen dos problemas. El primero es simple y menos crítico: mientras más grande el log más lenta la recuperación post crasheo. El segundo es más problemático: cuando un log está lleno(o casi lleno) no más transactions puede ser committed al disco, haciendo que el sistema no sirva.

Para enfrentar estos problemas, journaling FDs tratan el log como una estructura de datos circular, reutilizándola una y otra vez(por esto journal algunas veces es llamado **circular log**). Para esto el sistema toma accion despues de un checkpoint, básicamente liberar el espacio de la transaction asignada del log:



En este superblock del Journal, se guardan datos para saber qué transacciones todavía no llegaron al checkpoint, lo que sirve para liberar espacio del log. Se añade un paso al protocolo:

1. **Journal write**: escribir los contenidos de la transacción en el log, esperar que terminen.
2. **Journal commit**: escribir el transaction commit block en el log, esperar que termine, se dice que la transacción fue **committed(encomendada)**.
3. **Checkpoint**: escribir los contenidos de la update(metadata y data) a su lugar final en disco.
4. **Free**: algún tiempo dsp, marcar la transacción como free en el journal a través de actualizar el journal superblock.

El protocolo está completo, pero estamos escribiendo cada bloque en disco dos veces, un costo muy alto.

**Metadata Journaling**

Aunque la recuperación ahora es rápida, las operaciones normales del FD son más lentas, específicamente los write, se dobla el trabajo, esto es peor en un write secuencial. Debido a esto se han buscado formas de aumentar la performance, una de estas es el modelo **data journaling**, ya que tmb registra todos los datos del usuario, no solo metadata. Una forma más simple de journaling es llamado **ordered journaling**(o solo **metadata journaling**), casi igual pero sin la user data. Para evitar el caso de que se escriba en el journal todo pero se pierda la user data se pensó en escribirla directamente a disco, con el siguiente protocolo:

1. **Data write**: escribir los datos en su dirección final, esperar que se complete
2. **Journal write**: escribir los contenidos de la transacción en el log, esperar que terminen.
3. **Journal commit**: escribir el transaction commit block en el log, esperar que termine, se dice que la transacción fue **committed(encomendada)**.
4. **Checkpoint**: escribir los contenidos de la update(metadata y data) a su lugar final en disco.
5. **Free**: algún tiempo dsp, marcar la transacción como free en el journal a través de actualizar el journal superblock.

Con el paso uno se evita que se apunte a datos basura.

**CHapter 43 Log-structured File Systems**

Al comienzo de los 90 se desarrolló un nuevo file system conocido como log-structured file system, la motivación de sus creadores para hacerlo fueron las siguientes observaciones:

* **La memoria de los Sistemas está creciendo**: Con el crecimiento de la memoria, más datos pueden almacenarse en el cache de la memoria. Mientras más datos en cache, el tráfico del disco consiste cada vez más en writes, los reads son servidos por el cache por lo que la performance del FD está dada por la performance de los writes.