Sistemas Operativos

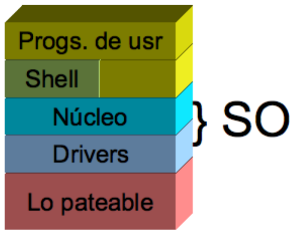
Índice

Introducción a SO

Un sistema operativo podemos definirlo como un **intermediario entre HW y SW.** De esta forma genera una abstracción para el usuario y sus programas, para que este no deba interactuar con cuestiones de HW de bajo nivel, y también puede controlar que el uso del HW sea el correcto. Es decir, que **el SO le provee servicios al usuario, pero a la vez lo controla y limita** en lo que puede hacer.
Históricamente, el SO no era SW como hoy en día, sino que era un operador físico que debía gestionar adecuadamente los inputs/outputs de los usuarios (dar outputs solo al dueño del input y a nadie más), asi como también operar el HW de manera de asegurarse que este no se dañara.
Entre otros problemas a gestionar por el SO, está la contención, que nace cuando se quiere optimizar el tiempo ocioso del procesador tomando partes de tareas de otros programas. Que haya contención significa que es muy probable que varios programas quieran acceder a la vez al mismo recurso, por ejemplo, el input del teclado, o el output de pantalla.

Hoy en día entonces, el SO es una pieza de software que corre con privilegios máximos dentro de un HW, que gestiona los recursos de manera tal de manejar la contención y concurrencia de los distintos programas de usuario de manera correcta y con buen rendimiento, y que a la vez le brinda al usuario una interfaz "amigable" para interactuar con los distintos componentes de HW.

Actualmente, las distribuciones de los Sistemas Operativos vienen con muchas más utilidades que las propias de un SO, esas utilidades son programas que forman parte de la Suite del SO, pero no del Kernel (ej: calculadora, terminal de comandos, admin de tareas). **El SO está formado por el núcleo (kernel) y los drivers que permiten controlar el HW.**



Proceso

Un proceso se puede pensar como la **instancia de ejecución de un programa**, que incluye su estado actual (valores de variables, registros, IPC). El SO será el encargado de administrar los recursos que cada proceso necesite, así como también gestionar y controlar que cada proceso sólo acceda a sus porciones de memoria y variables. Además, deberá poder administrar de manera "eficiente" (no habiendo una mejor política de eficiencia que otra, y siempre dependiendo de cada contexto de uso) el hecho de que varios procesos se puedan ejecutar en simultáneo. Las políticas que un SO utiliza para determinar esto se conoce como Scheduling.
Lo importante es que, visto desde el lado del proceso, este debe desconocer y debe ser totalmente transparente el hecho de que haya más o menos procesos distintos ejecutándose en simultaneo con él. A sus ojos, es el único proceso (o él y sus hijos) en el CPU.

El SO le asigna a cada proceso un *id* o *pid*, que será único entre todos los procesos en ejecución.

Un proceso puede realizar las siguientes tareas:

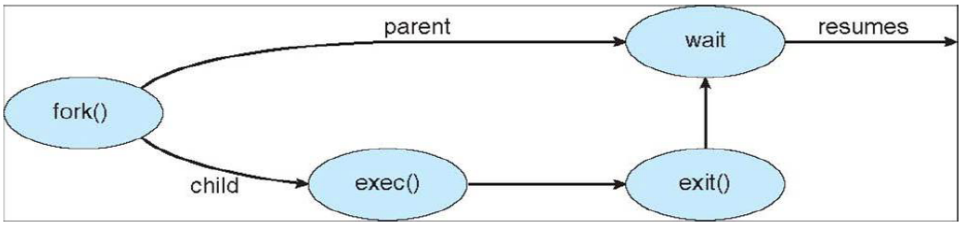
- **Ejecutarse**. Es decir, realizar operaciones entre sus registros, variables, accesos a su espacio de memoria, etc
- **Hacer una syscall**. Acceder a operaciones que la interfaz del SO provee para interactuar con el Hardware.
    - **Crear otro proceso hijo**. Mediante las syscalls fork(), execve()

- **Realizar E/S**. Mediante las syscalls de lectura y escritura en las direcciones válidas según el sistema de archivos
- **Terminar**. Libera todos sus recursos pedidos y el SO lo quita de las colas de ejecución del scheduler. El código de status de terminación se reporta al padre.

Cabe aclarar que realizar una llamada al sistema requiere cambiar el nivel de privilegio, cambiar el contexto, poner el programa en espera, generar una interrupción, **y eso toma tiempo**.

Árbol de Procesos

Todos los procesos están jerarquizados. Cuando el SO arranca, el primer proceso, padre de todos es *init.*
A partir de ese proceso, mediante fork se van creando nuevos procesos.
Usando la syscall fork, se crea un proceso hijo igual al actual. Copia páginas de memoria y valores (puede utilizar copy on write). La única diferencia es el pid, que a su vez es el valor de retorno de la función fork (si el retorno es 0, entonces soy el hijo ejecutando, sino es el padre conociendo el pid del hijo que acaba de crear). El proceso hijo entonces va a empezar a ejecutar desde donde había dejado el padre. Aquí puede seguir el mismo código, o reemplazar su código binario mediante exec. El padre puede esperar a que el hijo termine de ejecutar mediante la syscall wait.



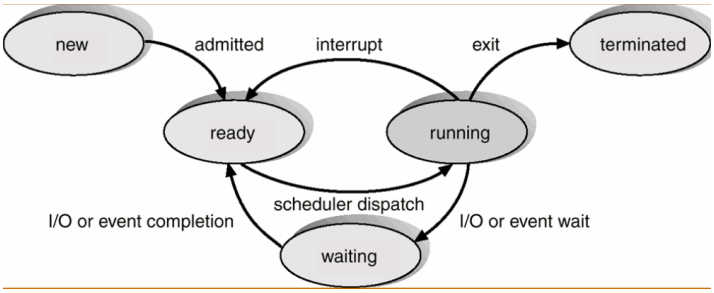
1 Ejemplo de ejecución de un comando desde el shell

```
while (TRUE) {                          /* repeat forever */
    type_prompt();                      /* display prompt on the screen */
    read_command(command, parameters);  /* read input from terminal */

    if (fork() != 0) {                  /* fork off child process */
        /* Parent code. */
        waitpid(-1, &status, 0);        /* wait for child to exit */
    } else {
        /* Child code. */
        execve(command, parameters, 0); /* execute command */
    }
```

Estado de un proceso

Para que el SO pueda administrar sus recursos de la mejor manera, es conveniente saber en qué estado se encuentra un proceso. Por ejemplo, si un proceso hizo una syscall de E/S y está esperando una interrupción que lo notifique que la operación terminó, no tiene sentido que el SO le asigne tiempo de ejecución de CPU, ya que no lo utilizará. De igual forma, si un proceso está listo para correr, el SO debe saberlo para poder decidir a quién de todos ellos darle CPU para que continúe su ejecución.
Elementalmente, los estados de un proceso son:



Los procesos en estado listo, son aquellos entre los cuales el SO puede “elegir” para poner a correr y se registran en una lista de PCB (*Process Control Block*), que contiene la prioridad del proceso, el estado, y la información necesaria

para realizar el context switch. La cantidad de procesos listos se conoce como la **carga del sistema**. Cuál de todos será el que corra, dependerá de la prioridad que tenga asociada cada proceso, y fundamentalmente de la política de scheduling que el SO implemente.

Syscalls

Sabemos que un SO debe poder manejar los recursos del sistema cuando un programa quiere crear, escribir, leer modificar archivos (o cualquier recurso de HW, que se manejarán básicamente como archivos).
Entonces, un proceso que necesita interactuar con el Hardware debe hacer una llamada al sistema. Estas llamadas son provistas por el SO a modo de API para que tanto el programa como el usuario detrás de él puedan abstraerse de las particularidades propias del HW del equipo y le sea totalmente transparente.

Detalles técnicos

Si tomamos el ejemplo de la syscall Read (Tanenbaum – 48), una vez que se ejecuta el código assembler con el número de syscall asociado a Read, se ejecuta la instrucción TRAP para cambiar a modo Kernel y empezar la ejecución en una dirección fija dentro de él. Luego, el kernel examina el número de syscall y hace un dispatch al handler asociado a ese número. Se ejecuta ese handler, luego se vuelve a cambiar a modo usuario, se setea el IPC luego de la instrucción TRAP, se incrementa el valor del stack para limpiar los parámetros del Read y sigue la ejecución normal. Notar que este tipo de syscall es **bloqueante**. Por ejemplo, si el read era hacia el teclado y el usuario aun no tipeó nada, se detiene la ejecución normal del programa.

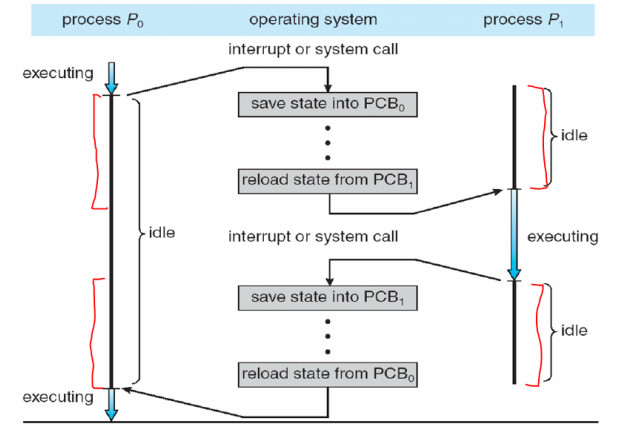
Ejemplos

fork, exec, write, read, wait, getpid, pipe, open, close, lseek, kill, exit.

La particularidad de write y read es que son lo suficientemente genéricas como para funcionar bien según el HW sobre el que se esté haciendo la operación. Es decir, que se puede escribir o leer tanto en la pantalla, como en una impresora, un disco externo, el SSD, etc.
En todas las syscalls es necesario llamar al kernel, lo que implica **un cambio de privilegio**. Para esto entonces, también hay que hacer un cambio de contexto.

Context Switch

Es necesario hacerlo para pasar de ejecutar un proceso, a ejecutar otro.
Como generalmente hay muchos procesos esperando ejecutarse en la CPU, el SO debe poder cambiar entre un proceso y otro. Además, también puede darse que un proceso haga un llamado a una syscall, o que él mismo, por ejemplo por estar esperando un resultado de una E/S, no necesite procesar más.
Para realizarlo, se deben guardar los valores de los registros, el IPC, el descriptor de archivos en una tabla llamada PCB, junto con el estado actual del proceso, límites de memoria asignados y la prioridad que tenga para volver a correr cuando vuelva a estar listo. Luego, según la política de scheduling, se carga el nuevo proceso que va a correr del PCB de ese proceso, se carga el IPC nuevo y el proceso se ejecuta.
El PCB es el representante de cada proceso para el SO.
Realizar el cambio de contexto genera tiempo muerto, algunos procesadores tienen dos sets de registros para minimizar este tiempo, pero aun así sucede. Por eso, para aprovechar al máximo el procesador y minimizar el tiempo administrativo del mismo, es necesario tener una buena política de Scheduling, que tendrá un impacto muy determinante en el rendimiento del sistema.



2Los tiempos idle del sistema se busca minimizar. Ejemplo de CS por Interrupcion.

Comunicación entre Procesos

Ya sea de forma remota o dentro de un mismo equipo, es necesario que existan mecanismos de comunicación entre los procesos para (Silberschatz – 122)

- **Compartir Información**: Por ejemplo, un archivo compartido entre usuarios.
- **Mejorar la velocidad de procesamiento**: Por ejemplo, al romper una tarea en subtareas más pequeñas mejora la velocidad, pero estas deben poder comunicarse.
- **Modularizar**: Ya que querremos construir sistemas modulares, separando las funcionalidades en procesos o threads separados.
- **Conveniencia**: Ya que, de esta forma, se aprovecha al máximo el HW y se da una sensación al usuario de real paralelismo y genera una buena respuesta.

Ejemplo

Por cuestiones de rendimiento y seguridad, Chrome utiliza 3 tipos de procesos. Cada tab se ejecuta en un proceso distinto, de esta forma si una página se cuelga, el navegador está disponible para seguir ejecutando. Además, las tabs corren en el tipo de proceso que genera un sandbox, restringiendo el acceso a disco y E/S, con lo que se minimiza el riesgo de un ataque.

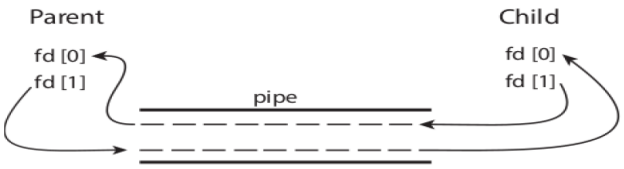
La comunicación entre procesos puede ser sincrónica o asincrónica. En la primera, el emisor se queda bloqueado hasta que el receptor termina de recibir el mensaje, es más fácil asegurarse que el receptor recibió el mensaje.
En la segunda, el emisor envía algo y el receptor lo recibirá en algún momento. El emisor no se bloquea esperando que el receptor termine de recibir, aunque es necesario algún mecanismo para saber que el mensaje se recibió.

Formas de IPC

Algunas formas para hacer IPC incluyen el pasaje de mensajes entre procesos a través de una cola de mensajes, o bien la memoria o algún otro recurso compartido (un archivo, bdd, etc).

Pipes

Es para envío de mensajes de forma unidireccional para procesos que se pueden acceder entre sí. Con los pipes ordinarios (‘|’ en Linux), solo puede acceder al mensaje el destinatario explicito, y el mismo muere una vez que se envió el mensaje.
En cambio, con los named pipes se crea una cola FIFO y cualquiera puede acceder al pipe (que cuente con los permisos otorgados al mismo).
Para generar un pipe se crean dos file descriptors, uno en cada dirección entre los procesos involucrados (read only y write only)



Los pipes anónimos requieren que los procesos que se comunican sean descendientes del proceso que crea el pipe. El padre también puede usarlos para comunicarse con sus hijos.

Redes WAN y LAN

Para una red local o mediante internet por ejemplo con protocolo TCP/IP.
El tipo de comunicación LAN generalmente genera poca latencia y es para equipos que se encuentren dentro de una misma red local. Ejemplo, dispositivos conectados a un WAP, o cableados entre si como una PC y una impresora, etc.

Sockets

Si los procesos no se conocen entre sí y se encuentran en máquinas distintas, se puede generar una comunicación mediante sockets. Es el concepto de enchufe. Ambos extremos deben crear un socket. Solo deben conocer de antemano la dirección IP y los puertos utilizados en cada extremo para poder establecer la conexión. Generalmente, varios procesos socket clientes se conectan a un proceso socket servidor y mediante función de enviar y recibir, se establece la comunicación.
Una vez establecida la comunicación, se puede leer y escribir en el como si fuese un dispositivo más, un archivo.

Scheduling

(Silberschatz 261)
En un SO que soporta multiprogramación (capacidad de un SO de tener varios procesos corriendo), con frecuencia se tienen varios procesos o threads que necesitan compartir la CPU. Sin embargo, cada procesador que tengamos solo podrá correr de a un proceso a la vez, por lo que si tenemos más procesos en ejecución es necesario realizar una **planificación de ejecución de los procesos para poder lograr efectivamente la multiprogramación y la sensación de multitarea.** El módulo del SO que se encarga de eso es el Scheduler.

La política de Scheduling es una de las principales marcas registradas de un SO, gran parte del esfuerzo por optimizar el rendimiento de un SO se centra en su scheduler.

Para tener una buena política de scheduling, resulta fundamental conocer el propósito del sistema en el que estará corriendo el SO. Casi todos los procesos alternan ráfagas de cálculos con peticiones de E/S, pero la distribución y cantidad de estas ráfagas puede variar. A los procesos que invierten la mayor parte de su tiempo realizando cálculos se los llama limitados a cálculos (CPU-bound), mientras que a los que están la mayor parte del tiempo esperando E/S (por ser programas interactivos, por ejemplo, un procesador de texto) se los llama limitados a E/S. De todas formas, a medida que las CPUs se vuelven más rápidas, los procesos tienden cada vez más a ser limitados a E/S, por lo que es importante encontrar un buen balance para que los mismos puedan realizar sus cálculos rápidamente hasta llegar a la petición E/S que los haga esperar, y que no estén bloqueados porque están esperando para ejecutarse.

Objetivos de la política de Scheduling

- **Fairness**: Que cada proceso reciba una dosis justa de CPU, definiendo que implica ser justo.
- **Eficiencia**: Que la CPU esté ocupada la mayor cantidad de tiempo posible. (Recordar que el context switch lleva tiempo)
- **Carga del sistema**: Minimizar esto, es decir la cantidad de procesos en estado listo.
- **Tiempo de respuesta**: Minimizar esto para mejorar la experiencia de usuario. Esto tiene en cuenta en la prioridad los procesos interactivos versus los que no lo son.
- **Latencia**: Minimizar el tiempo que le toma a un proceso dar resultados

- **Tiempo de ejecución (turnaround)**: minimizar el tiempo total que le toma a un proceso finalizar.
- **Throughput**: Maximizar el número de procesos terminados por unidad de tiempo. Esta política es muy usada. Busca terminar la mayor cantidad de procesos posibles. Tiene en cuenta para la prioridad aquellos procesos que estén cercanos a su finalización.
- **Liberación de recursos**: Hacer finalizar cuanto antes los procesos que tienen reservados más recursos.

No es posible garantizar todos estos objetivos a la vez. Por ejemplo, mejorar el throughput atenta contra el tiempo de respuesta. Es necesario **buscar un compromiso** entre ellos para tener un buen funcionamiento buscando una combinación de objetivos y teniendo el menor impacto negativo en el resto.

Tipo de Scheduler

Puede ser **cooperativo (non preemptive) o con desalojo (preemptive).** En el primero, el proceso, voluntariamente o por syscall devuelve el CPU. Mientras eso no pase, el proceso seguirá ejecutando.
En el segundo caso, que es la opción que hoy en día usan todos los SO modernos, el scheduler usa la interrupción del clock (y también las llamadas a syscall) para decidir si el proceso actual debe seguir ejecutando o no. Para este estilo de scheduler, la decisión puede estar basada en que un nuevo proceso en estado ready tiene más prioridad y debe ejecutar de inmediato, o si es un scheduler más basado en fairness, puede desalojarlo simplemente porque agotó su tiempo máximo de uso del CPU (quantum).

Como un ejemplo útil de scheduler cooperativo, podemos tomar sistemas de procesamientos únicamente **batch** o por lotes. No hay interacción con usuarios, ni necesidad de resultados intermedios, entonces puede ser aceptable un algoritmo no apropiativo, ya que minimiza el turnaround y maximiza la eficiencia por no tener intervalos de Context Switch.
Los sistemas de **tiempo real** también pueden tener en ocasiones schedulers no apropiativos ya que los procesos saben de antemano que no pueden ejecutar por períodos largos, entonces se liberan voluntariamente. Además, usar esta política da garantía de continuidad.
Por otro lado, para los sistemas **interactivos**, resulta fundamental que el scheduler sea preemptive. Aunque esto si bien maximiza el tiempo de respuesta, puede derivar en problemas de sincronización si varios procesos comparten recursos, y es necesario implementar mecanismos para lidiar con ellos (Ver Sincronización).

Planificación en sistemas Non Preemptive

Generalmente, como sea que esté implementada la cola o línea de espera de los procesos listos, los registros en estas colas suelen ser los PCBs de los procesos, que tienen toda la información necesaria del proceso en si para que el scheduler pueda decidir y asignarle CPU cuando corresponda.

First Come, First Served

Es el algoritmo más simple de implementar, está basado en una cola FIFO, tal que el primer proceso en llegar va a ser el primero en ponerse a procesar. Todos los procesos tienen la misma importancia, por ende, no hay distinción si llega un proceso que consume más recursos, o más CPU. Al ser non preemptive, en este caso el trhoughput se verá afectado según el orden de los procesos y, en cualquier caso, el tiempo de respuesta no es bueno.

Shortest Job First (SJF)

Ejecuta aquellos procesos que sean más cortos, para eso debe poder conocer o estimar los tiempos de ejecución de antemano. De forma óptima, **maximiza el thrughput**, pero para eso todos los trabajos deben estar disponibles al mismo tiempo, ya que, si primero ejecuta uno muy largo porque era el único, a pesar de que llegue uno más corto, al ser non preemptive, no cambiará la ejecución.
Es un caso particular de uso de prioridades fijas, por lo que puede causar inanición.

Puede en lugar de predecir cuánto tarda en ejecutar, estimar cuanto tardara en solicitar la próxima E/S, combinado con información del pasado para predecir. Puede salir mal si los procesos tienen comportamiento irregular.
Puede ser preemptive o non preemptive. En el primer caso, si hay un nuevo proceso con un tiempo de ejecución más corto que lo que le queda al actual, se procede al desalojo.

Planificación en sistemas Preemptive

Prioridades

Una solución clásica para poder resolver el problema de procesos que van llegando a destiempo para ejecutarse, es el de asignar prioridades. La asignación de prioridades puede estar dada por requerimientos de memoria de los procesos, cantidad de archivos abiertos, promedio de ráfagas de E/S, etc.
Ahora bien, si hay procesos con prioridades muy bajas, corren el riesgo de nunca llegar a ejecutarse, lo que puede derivar en starvation.
Una posible solución a esto, es la de establecer prioridades no fijas y, por ejemplo, lograr que la prioridad de un proceso vaya aumentando, a medida que éste envejece o sigue esperando en la cola de ready; o bien la inversa: ir bajando la prioridad a los procesos que van corriendo con mayor frecuencia.
Esta planificación, también puede combinarse con quantums. Es decir, que a pesar de que por prioridad podría seguir corriendo, su quantum se agotó y será desalojado para darle paso al siguiente. *(OJO con esto, porque si ejecuta uno de prioridad más baja, al toque podría desalojarse por el que acaba de pasar a ready, como solución se puede implementar que la prioridad vaya bajando a medida que pasa su quantum).*

Round Robin

A cada proceso se le asigna un quantum. Cuando este se agota, se desaloja y se pasa al siguiente proceso a ejecutar. La implementación más simple de esto también podría hacerse con una cola FIFO y se optimiza el fairness, y el tiempo de respuesta, aunque el throughput puede verse afectado. Si el proceso se bloquea antes de que termine su quantum, también es desalojado.
El desafío de esta planificación radica en definir el tiempo del quantum, ya que muchos cambios de contexto atentan contra la eficiencia del CPU, pero quantums muy largos pueden atentar contra el tiempo de respuesta, además si los quantums son suficientemente largos, transformamos la política de scheduling en FCFS.

Múltiples Colas

Se tienen varias colas que representan diversas prioridades. Cada cola puede tener asignados valores de quantums específicos, y los procesos a medida que se ejecutan, van moviéndose de colas.
A la hora de elegir un proceso, la prioridad la tiene siempre la cola con menos quantum. Si al proceso no le alcanzo ese quantum, se pasa a la cola siguiente, con menos prioridad, pero más quantum cuando vuelva a ejecutar.
Se ven beneficiados aquellos procesos que usan menos CPU, ya que tendrán mayor prioridad.
Una división clásica de prioridades podría ser entre procesos de foreground y procesos batch o de background. Estos últimos tienen prioridad baja, pero se espera que tengan quantums un poco más largos para poder realizar su trabajo.
Cada cola puede implementar su propio scheduling. Y un scheduling general controla las colas.
Ejemplo: La de Foreground puede ser RR, la de background puede ser SJF, la general de prioridades fijas, o con un RR.

Scheduling para RT

Los procesos tienen fechas de finalización estrictas, no puede **no** cumplirse un deadline. Una política posible consiste en correr el proceso más cercano a perder su deadline.

Scheduling para SMP (múltiples cores de procesamiento simétrico)

Esto es un claro ejemplo de cómo impacta la arquitectura de HW en el SO.
Todos los cores tienen el mismo bus de memoria y tienen el mismo tiempo de acceso a todas las partes de la memoria.

Sin embargo, cada procesador tiene su cache, y lo que está almacenado ahí es muy importante para los programas. Por ende, si a un proceso lo cambio de Core, podría implicar que deba llenar la cache nuevamente y se empeore el tiempo de respuesta, versus que haya esperado que el core en el que había corrido previamente se libere y luego tenga hits en su cache. En arquitecturas NUMA, esto es aún más deseado, ya que el acceso a memoria no es uniforme.

Afinidad del procesador es justamente eso, tratar de usar el mismo procesador, aunque tome un poco más de tiempo para ejecutar nuevamente. Esta información se almacena también en el PCB. La afinidad puede ser dura o bien puede ser algo deseable, afinidad blanda. Linux provee una syscall sched_setaffinity(), que soporta afinidad dura.

La afinidad puede generar desbalanceos en los cores. Técnicas como pull o push migration apuntan a eventualmente rebalancear la carga.

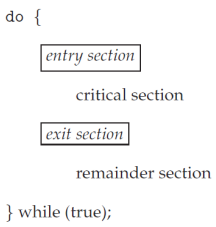
Sincronización

--73 pdf tanen; 203 silber

Los SO tienen que manejar la contención (varios programas pueden querer acceder a un mismo recurso a la vez) y concurrencia de manera de tal de hacerlo correctamente y **hacerlo con buen rendimiento.** El acceso concurrente a datos compartidos puede resultar en inconsistencias, necesitamos mecanismos para evitar eso.

Con multiprogramación, toda ejecución paralela debería dar un resultado equivalente **a alguna ejecución secuencial** de los mismos procesos. *(Condición necesaria para ejecución correcta, pero no suficiente)*

Una condición de carrera, o **race condition**, se da cuando varios procesos acceden y manipulan los mismos datos de manera concurrente, y **el resultado de la ejecución varía de acuerdo al orden en que ellos hayan accedido a esa información**. Para evitar esto, es necesario implementar mecanismos, como por ejemplo asegurarse que solo un proceso a la vez pueda acceder a esas *secciones críticas* donde se da una manipulación concurrente de datos. En otras palabras, si el resultado de las ejecuciones depende del scheduling, hay una race condition. Recordar que cada sección crítica es deseable que tenga una sola responsabilidad.



En general, para resolver los problemas de sincronización y poder implementar accesos y salidas a las secciones críticas, se necesita obtener un poco de ayuda del HW (Ejemplo que intenta solucionar con booleanos, donde ambos pasan el if es un ejemplo de eso).

Primitivas de Concurrencia

Test and Set (TAS)

Es una instrucción provista por el HW para poder establecer **atómicamente** (se deshabilitan las interrupciones al momento de ejecutarse la instrucción) el valor de una variable en 1 y devolver el valor anterior.

```
bool TestAndSet(bool *destino)
{
 bool resultado = *destino;
 *destino = TRUE;
 return resultado;
}
```

3 Esto es solo pseudocódigo, en realidad se resuelve a nivel HW

Si más de un proceso ejecuta esta instrucción, si bien todos escribirán un 1, pueden saber si fueron o no los primeros. Para salir, sin necesidad de atomicidad, se puede reestablecer el 0, ya que eso se hace dentro de CRIT.

```
boolean lock; // Compartida.

void main(void)
{
 while (TRUE)
        {
         while (TestAndSet(&lock))
          // Si da true es que ya estaba lockeado.
          // No hago nada.
          ;

         // Estoy en la sección crítica,
         // hago lo que haga falta.

         // Salgo de la sección crítica.
         lock = FALSE;

         // Si hay algo no crítico lo puedo hacer acá.
         }
}
```

La desventaja de esta primitiva es que todos los procesos que están esperando entrar a la sección crítica, durante todo su quantum se quedarán colgados en el while… **busy waiting.** Si bien, es una forma válida de sincronización, es una forma muy agresiva de intentar obtener un recurso (aunque a veces es la mejor).

Productor-Consumidor

Mecanismo propuesto por Djikstra, que atacó el problema de forma más general. Se tiene un buffer compartido de tamaño limitado. Hay procesos que insertan en el buffer (productores), y procesos que sacan de ahí (consumidores).
El problema radica en ver cómo resolver este buffer acotado. Las race conditions que pueden darse en este approach están claras: productores y consumidores concurrentemente querrán actualizar el mismo buffer y sus variables.
Adicionalmente, si el productor quiere producir cuando el buffer está lleno, o el consumidor consumir cuando está vacío, deben esperar.
La solución a esto: **Semáforos.**
Un semáforo (TAD) tiene las siguientes características:

- Es una variable entera que se puede inicializar con **cualquier** valor.
- Solo tiene dos operaciones: **wait()** y **signal()**. Notar que no es posible leer el valor en el que se encuentra.

```
wait(s): while (s<=0) dormir(); s- -;
```

4 Una vez que s>0, consume, es decir que sale del while y decrementa s

```
signal(s): s++; if (alguien espera por s) despertar
a alguno;
```

5 La especificación no define a que proceso hay que despertar

La implementación de estas dos operaciones es sin interrupciones.
Un semáforo inicializado en 1, que siempre varíe entre estos valores se conoce como mutex, ya que otorga acceso exclusivo a un único proceso.

Un mal uso de semáforos puede que hacer que se genere una espere circular en la que un proceso espera a otro que está esperando…. que está esperando al primero. **Deadlock.**

Puede ser que tengamos un SO muy rudimentario o light y no tenga disponible las instrucciones para usar semáforos y debamos recurrir a spin locks. Puede ser también que no contemos con el soporte de HW para los exclusive lock del spin lock.

Exclusión mutua

Llamamos un objeto atómico básico a un registro atómico que nos provee:

- getAndSet(bool b): Devuelve el valor anterior y setea el valor de b.
- testAndSet(): Devuelve el valor anterior y setea true.
- Además tiene las operaciones de get y set.

De esta forma, se pone en interfaces lo visto al principio.

Spin Lock (TAS Lock)

Objeto que provee una interfaz para crear un registro atómico tal que posee una función lock y otra unlock para poder tomar acceso exclusivo a una sección de código. **Lock no es atómico**, lo que es atómico es la llamada a testAndSet dentro de la implementación de lock. **Spin lock hace busy waiting, pero tiene un overhead más bajo que el uso de semáforos,** el wait y el signal tienen implementaciones más pesadas en cuanto a uso de recursos. Por ende amerita un análisis de cual vale la pena. Si las secciones críticas son chicas y/o simples, entonces el busy waiting no es una mala alternativa, ya que la espera será realmente poca.

Local Spinning (TTAS Lock)

La idea es testear antes de hacer testAndSet, de esta forma, como el get obtiene el valor directamente accediendo a cache, no es necesario hacer escrituras en memoria innecesarias que consumen tiempo. Una vez que mediante el get obtiene permiso para lockear, hace un nuevo check mediante testAndSet por si el scheduler cambió y perdió nuevamente el lock. De ser asi, el ciclo se repite, sino consigue el lock.

```
3  void lock() {
4    while (true) {
5      while (mtx.get()) {}
6      if (!mtx.testAndSet()) return;
7    }
8  }
9
```

Registros Read-Modify-Write atómicos

Estas primitivas permiten, sin interrupciones, devolver el valor anterior del registro e incrementar la variable en 1, sumar un valor arbitrario, o swappear valores.

```
1  atomic int getAndInc() {
2    int res = reg;
3    reg++;
4    return res;
5  }
6
7  atomic int getAndAdd(int v) {
8    int res = reg;
9    reg = reg + v;
10   return res;
11 }
12
13 atomic T compareAndSwap(T  u, T  v) {
14   T res = reg;
15   if (u == res) reg = v;
16   return res;
17 }
```

Los tipos T pueden ser cualquiera siempre y cuando tengan implementada la igualdad.

Colas atómicas

Utilizan un mutex para garantizar acceso exclusivo a las operaciones de la cola, el desencolado también puede informar mediante un bool si la cola estaba vacía o no.

Mutex recursivo

Utiliza un contador para chequear la cantidad de llamadas de un mismo proceso al mutex. Para que otro proceso pueda acceder, el que lo tiene debe desbloquear la misma cantidad de veces que lo bloqueó. Puede ser una estrategia peligrosa si no se maneja con cuidado. Para bloquear, espera hasta que el proceso bloqueante sea igual al que había bloqueado previamente, o bien que no haya nadie. Incrementa el contador y obtiene el lock. Para desbloquear decrementa el contador, si llega a 0, setea el owner en ‘ninguno’.

Condiciones de Coffman

Son incorrectas. Pero se aproximan mucho a las condiciones necesarias para garantizar la existencia de deadlocks. No son ciertas en todos los contextos, pero brindan una aproximación.

- Un recurso tiene que ser de asignación exclusiva. Es decir, debe existir una sección crítica con mecanismos de exclusión mutua.
- Los procesos que tienen un recurso, pueden solicitar otro.
- No hay mecanismo de timeout para quitar un recurso. (No preemption)
- Se pueden armar ciclos de espera circular.

Si se puede garantizar que alguna de estas condiciones en un programa particular no se cumplan, entonces podremos garantizar que es deadlock free.

Problemas Clásicos

1: Turnos

Procesos todos corriendo en simultáneo.

Problema: Que se ordenen en algún momento. Es decir, se tienen N procesos numerados, se quiere que se ejecuten en orden.

Solución: Inicializar N semáforos en 0 a los que cada proceso pueda acceder. Lanzar los N procesos. Hacer un signal al primer semáforo.

Luego, cada proceso lo que hace es un wait sobre su propio semáforo, cuando lo habilitan a correr, ejecuta su código y termina haciendo un signal al semáforo siguiente.

2: Barrera

No es un problema de exclusión mutua. La sección crítica es un poco más laxa, ya que pueden estar N procesos a la vez, pero debe darse una condición previa.

Problema: Cada proceso tiene dos partes de ejecución: una parte a y una parte b. Queremos que cada b empiece a ejecutarse una vez que **todos los a de todos los procesos** se hayan ejecutado, el orden de ejecución de a y b no importa.

Solución: Se tiene un semáforo inicializado en 0 (barrera baja) y un entero atómico contador. Cada proceso ejecuta su parte a, luego incrementa atómicamente el contador y el valor de retorno de ese contador es menor a la cantidad de procesos, el mismo hace un wait sobre la barrera. Una vez que el último proceso llega, salta el if, hace un signal y ejecuta b. Ese signal despierta a algún proceso que hace signal y ejecuta b, y asi hasta finalizar todos los signals.

3: Barbero (Peluquería de Lamport)

Problema: Servidor tiene un único procesador para atender de a una petición a la vez (peluquero). Un buffer de espera de procesos limitado con N posiciones, se rechazan pedidos si el buffer está lleno. Si no hay pedidos, no se hace busy waiting, el servidor se duerme.

Solución: El buffer es un semáforo inicializado en 0, cuando llega un cliente, consulta los clientes en fila de manera segura y si es mayor a N se va, sino se hace un signal al semáforo de clientes (buffer) y se hace un wait a un semáforo siguientes, que es el que tomara el peluquero cuando va a cortar a alguien. **Esta solución no garantiza que se preserve el orden de llegada, y podría causar inanición en los clientes.**

4: El problema del consenso

Nace como motivación de ver que tan buenos son los registros atómicos, que tan bueno es TAS.

Problema: Decidir si pueden n procesos acordar sobre un estado booleano.

Solución: No se puede garantizar consenso para un n arbitrario **con registros atómicos**.

Número de consenso es el n fijo para el cual las primitivas de sincro resuelven el problema.

Los registros RW tienen valor 1 (1 solo proceso, es decir, no sirven); colas y pilas atómicas tienen valor 2; TAS 2 (le falta WAIT FREEDOM); **compare and swap infinito** (si nadie decidió la decisión es de ese proceso, sino el proceso decide lo que decidió el proceso que decidió)**.**

Correctitud de Programas en paralelo

La definición clásica de que un programa es correcto si dado un estado que cumple con la precondición de la especificación del mismo, llega a la poscondición en una cantidad finita de pasos, se queda corto. Existen en el mundo de los programas paralelos, muchas más aristas y matices que evaluar, ya que para empezar los programas paralelos tienen muchas **ejecuciones** posibles.
La pregunta cambia entonces a considerar **todas las ejecuciones posibles (para todos los schedulings (todos los schedulings que cumplan cierta condición de fairness)).** Además, interesa saber si los procesos abortan, se mueren, se bloquean, tienen inanición, etc.
Se plantearán entonces un conjunto de propiedades que predican sobre toda ejecución posible.

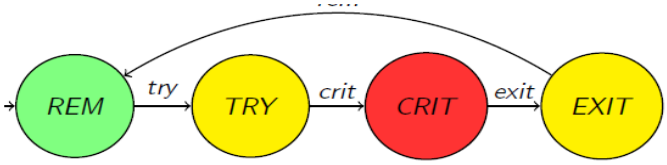
Tipos de propiedades:

- **Safety**: Cosas malas no suceden. Es decir, hay ausencia de deadlocks, exclusión mutua, no pérdida de mensajes, no muerte repentina, etc. Más formalmente podemos decir que **tienen un contraejemplo finito** (si construyo un contraejemplo, en una cantidad finita de pasos demuestro que no se cumple alguna característica de safety).
- **Liveness** o Progreso: En algún momento algo bueno sucede. Es decir, cada vez que se recibe un estímulo el sistema responde, siempre en el futuro el sistema avanza, no inanición, etc. Más formalmente podemos decir que **tienen un contraejemplo infinito** (si construyo un contraejemplo, podrá el sistema quedar ejecutando o con CPU disponible para ejecutar instrucciones infinitamente y el sistema no progresará. Eventualmente se vuelve a un estado anterior que termina generando un loop entre los mismos estados, y no alcanza nunca el/los estados deseados).
- **Fairness:** Es en realidad una propiedad de liveness. Significa que los procesos reciben su turno con infinita frecuencia. Muchas veces, para argumentar correctitud, asumimos que contamos con un scheduling que garantiza esta propiedad.

Para poder trabajar con estas propiedades se inventaron lógicas temporales, que permiten demostrar por la positiva (que no hay contraejemplos). Para hacer demostraciones formales, se utilizan automátas y se deben formalizar las propiedades de nuestros programas en alguna de esas lógicas. Existen herramientas como Model Checkers y Theorem Provers.

Modelo de Procesos

En general, en el mundo paralelo, los algoritmos se pueden analizar mediante la siguiente máquina de estados:



- Estado: $\sigma : [0 \dots N-1] \mapsto \{REM, TRY, CRIT, EXIT\}$
- Transición: $\sigma \xrightarrow{\ell} \sigma'$, $\ell \in \{rem, try, crit, exit\}$
- Ejecución: $\tau = \tau_0 \xrightarrow{\ell} \tau_1 \dots$

Esto genera ciertas propiedades de las cuales nos interesará saber o demostrar de nuestros algoritmos:

Wait Freedom

Intuitivamente, “todo proceso que intenta acceder a la sección crítica, en algún momento lo logra, **cada vez que lo intenta”.** Es decir que el programa no va a esperar para siempre.

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

En toda ejecución, para todo estado k, para todo proceso i; Si el proceso en el estado k está en TRY entonces existe un estado k' posterior a k tal que ese mismo proceso en ese estado k' está en la sección crítica.

$$WAIT\text{-}FREEDOM \equiv \forall i.\Box\ IN(i)$$

Exclusión mutua

$$EXCL \equiv \Box\ \#CRIT \leq 1$$

***Para toda ejecución** y todo estado, no puede haber más de un proceso i tal que esos procesos en esos estados están en CRIT.*

Progreso del sistema

$$LOCK\text{-}FREEDOM \equiv$$
$$\Box(\#TRY \geq 1 \wedge \#CRIT = 0 \implies \Diamond \#CRIT > 0)$$

Siempre sucede que, si al menos un proceso está en TRY y no hay procesos en la sección crítica, entonces en algún estado posterior, la sección crítica dejará de estar vacía.

Notar que no asegura que proceso entra en CRIT. Es una versión light, solo dice que efectivamente algún proceso entrará en CRIT.

La intuición es que, si la sección crítica está libre, algún proceso podrá entrar eventualmente.

WAIT FREEDOM ES MAS FUERTE QUE LOCK FREEDOM

Progreso global Dependiente

$$STARVATION\text{-}FREEDOM$$
$$\equiv \forall i.\Box\ OUT(i) \implies \forall i.\Box\ IN(i)$$

***Si** todo proceso en cualquier momento siempre que está en la sección crítica sale, entonces **todo proceso** que esté intentando entrar, en algún momento lo logrará.*

Notar que no garantiza que los procesos no se cuelguen en CRIT y salgan.

Wait Freedom es más fuerte que todas. Starvation Freedom es un poco más fuerte que Lock Freedom.

Sección Crítica de a M

- Propiedad *SCM* a garantizar:

$$\forall \tau.\ \forall k.$$
$$1)\quad \#\{i \mid \tau_k(i) = CRIT\} \leq M$$
$$2)\quad \forall i.\ \tau_k(i) = TRY \wedge \#\{j \mid \tau_k(j) = CRIT\} < M \implies \exists k' > k.\ \tau_{k'}(i) = CRIT\}$$

```
1  semaphore sem = M;
2  proc P(i) {
3   // TRY
4   sem.wait();
5   // CRIT
6   sc(i);
7   // EXIT
8   sem.signal();
9  }
```

La sección crítica permite a lo sumo M procesos a la vez. Para todo proceso que esté en TRY, si en la sección crítica hay menos de M procesos en ese momento, entonces en algún estado posterior, ese proceso entrará a CRIT.
Notar que es una propiedad fuerte, no dice que algún proceso entrará, sino que **ese proceso entrará.**
Con semáforos la implementación es muy simple. Se inicializa el semáforo con el valor M.

--Off topic--
POSIX provee apertura de archivos de manera atómica, donde al abrir un archivo si no existe lo crea y si ya existe falla. Entonces se tiene un mecanismo sencillo, pero no eficiente de exclusión mutua.

Registros RW

Son registros o variables que cumplen ciertas propiedades y permiten la lectura y escritura de sus valores. Son una alternativa al uso de TAS y, eventualmente, de exclusión mutua.
Los registros pueden ser de simple escritura y simple lectura (de a una a la vez), de múltiples lecturas y simple escritura, o ambos múltiples. También pueden manejar solo valores booleanos o múltiples, y los registros puede ser **safe, regular o atomic.** Lógicamente, cuanto mayor es lo que pedimos, mayor será la complejidad de los mismos.

Si se solapa una lectura con una escritura entonces:

- Safe: Lectura puede devolver cualquier valor
- Regular: Lectura puede devolver valor consistente con los que hubieron
- Atomic: Las lecturas sucesivas devuelven resultados consistentes con una serializacion (es decir que una lectura previa no puede devolver un valor más nuevo que una lectura posterior)

Si tengo n procesos y voy a usar registros atómicos, para garantizar EXCL y LOCK-FREEDOM necesito **al menos** n registros RW (Burns & Lynch)

Administración de Memoria

--tanen 186; silber 348; silber 397
La memoria es compartida por todos los procesos. Sin embargo, procesos que no deben compartirla explícitamente, tienen la “ilusión” de que toda la memoria está disponible para ellos (al igual que el CPU). La primera pregunta que

surge es “Cuando tenemos multiprogramación y un proceso se bloquea para pasar a ejecutar otro, ¿Qué se hace con la memoria de esos procesos?”.

Swapping

Mecanismo “sencillo” que consiste en pasar a disco todo el espacio de memoria de procesos que no se estén ejecutando, y traer de disco el proceso que va a ejecutar.
Claramente esto funciona. pero es extremadamente lento. Además, podría pasar que la memoria no esté llena y haya espacio para mantenerlos en memoria a ambos

Ahora bien, si más de un proceso estará en memoria, debemos pensar en las direcciones de memoria que pueden acceder, y también pensar en el problema de que, si efectivamente un programa va a disco y vuelve, garantizar que siga accediendo a su porción de memoria, y no a otras direcciones.
Una solución para esto es que las direcciones del programa sean relativas (offset) y haya un registro que marque la base real en memoria.

Todo esto sigue trayendo problemas, que pueden ubicarse dentro de estas categorías de problemas a resolver con el manejo de memoria:

- Reubicación: Swapping y cambio de contexto de manera rápida y segura.
- Protección: Que los programas solo puedan acceder a sus porciones de memoria
- Manejo del espacio libre: Donde ubicar nuevos programas, sobre todo evitando fragmentación.

Fragmentación

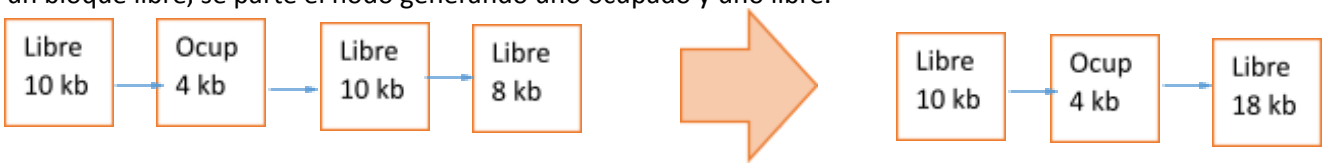
Cada vez que un proceso pide memoria para crear variables, por ejemplo, la memoria que se le asigna es memoria contigua (desde su punto de vista). Entonces resulta fundamental tener la memoria lo menos fragmentada posible.
La fragmentación es un problema tal que se posee memoria suficiente para atender el pedido de memoria de un proceso, pero esos espacios no son contiguos.
Existe la **fragmentación interna**, que es el espacio desperdiciado dentro de los bloques (por ejemplo, un programa solicito 4kb y le di 32kb); **y la externa** que es la de bloques libres pequeños y dispersos que tienen poca chance de ser tomados.
Para evitar la fragmentación, los SO organizan la memoria de forma muy específica (el stack arriba creciendo hacia abajo, el heap creciendo hacia arriba, el código de programa debajo de todo, etc)

Bitmap

Es un array de bits (0 y 1). El uso de bitmaps permite tener la memoria dividida en bloques (supongamos de igual tamaño, 4kb), y mediante el bitmap saber que bloque está ocupado (1) y cuál vacío (0).
Se programa relativamente de forma sencilla, pero se genera una tensión entre granularidad y tamaño del bitmap (a menor granularidad más fácil el recorrido, pero más fragmentación interna; a mayor más complejo el recorrido de la memoria). Además, buscar bloques consecutivos requiere una barrida lineal.

Lista Enlazada

Cada nodo de la lista representa un bloque libre u ocupado por un proceso, y además cada nodo tiene su tamaño y sus límites. Cuando se liberan recursos contiguos a otros libres, se mergean los nodos. Si se pide un bloque más chico a un bloque libre, se parte el nodo generando uno ocupado y uno libre.



Es más eficiente que el bitmap, la liberación y merge son O(1), el problema sigue siendo **donde** asignar, es decir, qué bloque libre elegir.

Estrategias de elección de bloque libre

- **First Fit**: Primer bloque que entra, asigna. Es rápido, pero genera más fragmentación partiendo bloques muy grandes.
- **Best Fit**: Busca donde entra de manera más acotada. Es más lento porque debe recorrer toda la lista y tampoco es mejor ya que genera muchos pequeños bloques que luego no pueden ser reutilizados.
- **Quick Fit**: Mantiene una lista de los bloques libres con los tamaños que más frecuentemente se usan. Consume más tiempo administrativo y es una estructura más grande.

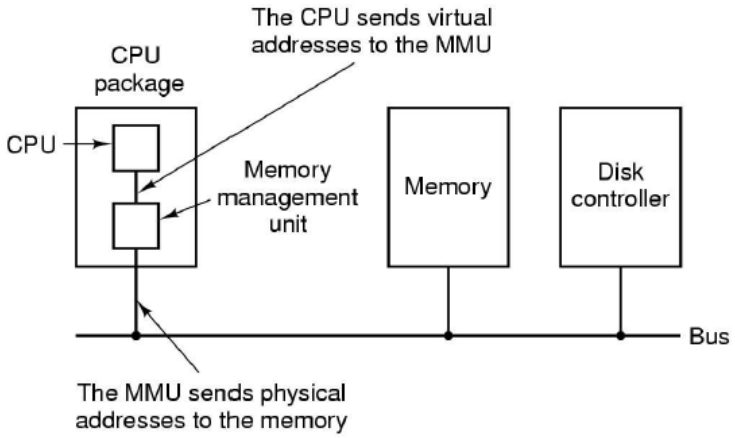
Ninguna de estas técnicas "directas" terminan resolviendo el problema de la fragmentación. Se necesitan algoritmos más complejos, pero genera una tensión ya que deben resolverse rápido.

Memoria Virtual

--397 silber

La motivación surge para resolver el problema de la reubicación y de otro que postula que, si tengo N bytes de memoria libre, y un programa que ocupa M>N bytes pero que solo utiliza de a K<N bytes a la vez, caramente se debería poder correr de alguna manera.

La idea entonces, es combinar tanto swapping junto a la virtualización del espacio de direcciones. **Eso es memoria virtual. Se requiere ayuda del HW,** mediante la unidad de manejo de memoria (MMU), tal que el programa y el CPU envían direcciones virtuales a la MMU, y esta las convierte en direcciones físicas que pone en el BUS.



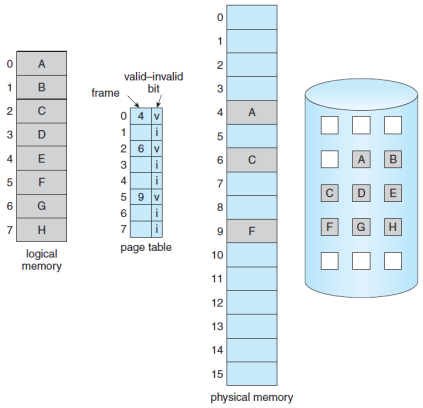
Con la memoria virtual ya no se depende del tamaño de la memoria para decidir el tamaño de direcciones (puedo procesar una imagen de 4gb con una memoria de 1gb), sino que queda determinada por cuanto es posible swappear (mandar a disco y traer de disco páginas de memoria).

La MMU posee varias tablas para realizar las conversiones, y entre otras posee una con un bit que indica si la dirección pedida está cargada o no. Si no está, se bloquea el proceso para realizar el swapping de memoria y disponibilizar esa dirección.

- El espacio de memoria virtual está dividido en bloques de tamaño fijo: **páginas.**
- El espacio de memoria física está dividido en bloques del mismo tamaño: **Frames.**

En los marcos, se puede ir poniendo en distintos momentos, distintas páginas.

La MMU traduce páginas a frames. Con los bits más significativos de la dirección virtual obtiene la página, que apunta a un frame, y con los menos tiene el offset de ese frame.



El problema que sigue teniendo el SO es qué página mandar a disco una vez que se solicita una nueva página.
Además, las páginas están en memoria, por lo que estas consultas consumen tiempo y ponen en jaque este Sistema de paginado. Para eso se creó la TLB que es un bloque en cache que tiene las páginas de más frecuente acceso.

Reemplazo de páginas

--silber 409
Existen varios algoritmos:

- FIFO. El viejo y conocido. Muy ingenuo.
- Second Chance. Como FIFO, pero con un bit de referenciado que le da una segunda oportunidad y la manda nuevamente atrás de la cola si es que se la referenció.
- Not Recently Used. Elimina las páginas que no fueron modificadas ni referenciadas, luego en orden de prioridad las que fueron referenciadas (solo se le hicieron lecturas) pues son baratas, y en última instancia las que se modificaron.
- Least Recently used. Es el que mejor tiende a funcionar, aunque es caro. La que se usó menos recientemente, es decir, hace más tiempo, es la que menos chance tiene de volver a referenciarse.

Copy on Write

--408 silber
Cuando se hace fork, es ingenuo duplicar la memoria del proceso. Entonces, se utiliza copy on write, que hace que compartan la memoria hasta que alguno de los procesos escriba. En ese momento, la página que fue escrita se duplica (se actualizan las tablas consecuentemente) y el resto sigue igual. De esta forma se ahorra tiempo y memoria.

Entrada / Salida

Los dispositivos de E/S pueden ser de almacenamiento, comunicaciones, interfaz de usuario.
Hoy en día, la preocupación primaria en cuanto al almacenamiento sigue siendo el disco rígido, ya que tiene un tiempo de respuesta mucho más lento que la memoria. También las unidades de cinta, menos usadas hoy en día pero que siguen vigentes para copias de seguridad. Otros dispositivos son los discos virtuales, que se encuentran en algún punto de la red y mediante ella se puede acceder (ejemplo, la facu). Hay distintos protocolos para acceder a discos virtuales: NFS (571 silber), CIFS, DFS, AFS. Genéricamente, estos protocolos se llaman Network Attahed Storage (NAS), que permiten disponibilizar un disco a través de la red, incluyendo restricciones de seguridad por usuario, por IP, etc. Además, usa TCP/IP para establecer la comunicación.

Un dispositivo de E/S tiene conceptualmente dos partes: El dispositivo físico y su controlador, que interactúa con el SO mediante BUS o registros. Con el dispositivo físico solo interactua su controlador.
Además, en cuanto a SW, es necesario contar con un Driver que conozca las particularidades del software y exponga una interfaz al Kernel para atender las peticiones.

Estos dispositivos además pueden ser:

- De lectura (mouse), escritura (impresora), o ambos (discos)
- Brindar acceso secuencial o aleatorio.
- Ser compartidos (discos), o dedicados (impresoras: hasta que no finalizan un trabajo o proceso, no pueden tomar otro).
- Comunicarse de a bloques o caracteres
- Ser sincrónicos o asincrónicos.

Además, todos tienen distinta velocidad de respuesta.

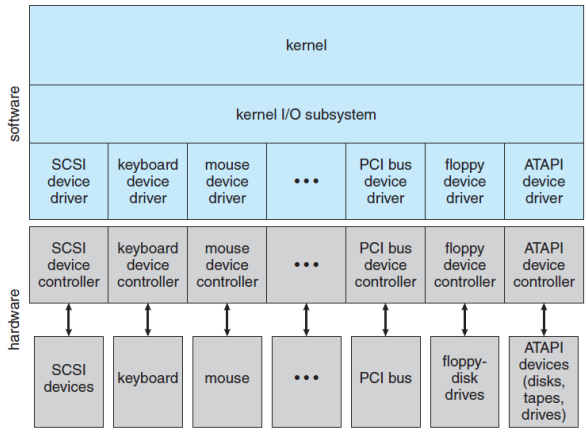
La principal función del SO es brindar un acceso consistente a toda esta variedad, ocultando lo máximo posible las particularidades de cada dispositivo (Subsistema de E/S).

Drivers

Como ya vimos, corren en el kernel con máximo privilegio para poder acceder a los dispositivos. Son los que terminan mandando los comandos al dispositivo a través de su controlador.
Existen drivers genéricos y específicos, pero siempre conocen las particularidades del HW con el que hablan.
Debido al alto privilegio con el que corren, hoy en día los drivers están aprobados y firmados por la compañía del SO (Microsoft, por ejemplo).

Subsistema de E/S

--- 597 silber
A nivel kernel, se cuenta con el Device Independent I/O que busca unificar la comunicación entre drivers y usuario, y con los drivers. Tanto el controlador de los dispositivos, como el dispositivo en sí mismo forma parte del HW.
Tanto el Device Independent I/O como los drivers corren con máximo privilegio, por lo que, entre otras cosas podrían colgar el sistema.



Es el intermediario entre el usuario y los drivers, y provee la interfaz necesaria para abstraer al usuario de las particularidades y cuestiones más técnicas que necesitan los dispositivos.
La idea es proveer entonces un set de syscalls bien sencillos:

- Open/close
- Read/Write
- Seek

Además, genera un filtro. Quiere decir que se encarga de chequear que los usuarios que lanzaron los procesos de accesos a dispositivos, cuenten con los permisos necesarios para hacerlo, y solo en ese caso proceder con la petición.
En Linux, los dispositivos se dividen en dos grupos:

Char Device

Permite transmitir (leer o escribir) la información byte a byte de forma secuencial. Ejemplo, mouse, teclado. No utilizan cache.

Block Device

Dispositivos como discos, transmiten la cantidad de a bloques. Los tamaños de los bloques están definidos. Además, permite el acceso aleatorio, es decir poder acceder a cualquier bloque sin acceder a los previos. Por lo general utilizan algún tipo de cache para por ejemplo almacenar bloques que podrían ser pedidos en el futuro.

En Linux, para saber el archivo a que tipo pertenece, es necesario mirar la primera letra de su definición: - es archivo común; d es directorio; b es un block device; c es un char device.

Para llevar a cabo estas acciones, el subsistema de E/S maneja todo como si fuese un archivo, es decir, que esa es la interfaz que provee al usuario. Dependiendo el dispositivo algunas funciones no estarán disponibles, pero en general será siempre el mismo conjunto de operaciones para todos los dispositivos.

Interacción con los dispositivos

Uno de los mecanismos más sencillos para interactuar es el **polling.** En este caso, regularmente el driver consulta un registro de status del dispositivo a fin de saber si la operación se finalizó. Con esta técnica los cambios de contexto están controlados, pero consume tiempo de CPU innecesario y en ese sentido no es eficiente.

Otro mecanismo es el de **interrupciones.** Aquí, el dispositivo genera una interrupción (poniendo una señal en el BUS) cuando tiene el resultado listo y se termina ejecutando un handler desde el lado del usuario para atender esa respuesta. La ventaja de esto, es que es eficiente en cuanto a consumo de CPU y permite el asincronismo para que el mismo proceso pueda seguir ejecutando otras partes del programa. En su contra, los cambios de contexto son impredecibles.

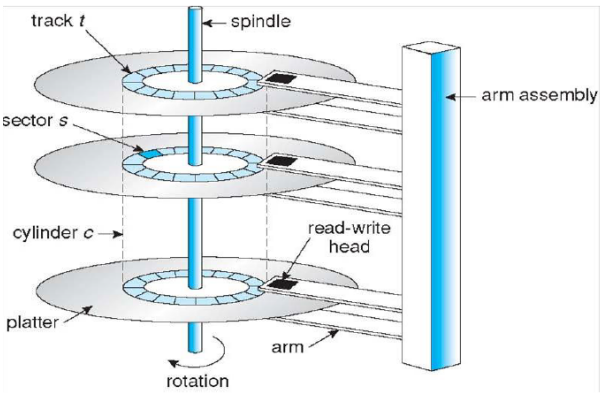
Algunos dispositivos permiten hacer **DMA (Direct Access Memory)**, en general se utiliza para transferir grandes volúmenes de datos sin que intervenga la CPU, que va directo a la memoria. Se requiere un controlador específico. La interrupción se genera una vez que todo el volumen fue transmitido y el controlador de DMA le avisa al CPU.

Rendimiento de E/S

--- 467 silber

Una de las claves del rendimiento de E/S es el manejo eficiente de los discos magnéticos. Estos tienen un cabezal en movimiento y moverla toma tiempo. También hay varios platos dentro del disco.

El disco rota constantemente, el desafío consiste en minimizar el movimiento del cabezal de acuerdo a los pedidos que van llegando. Para eso, el planificador de los pedidos va reordenando esos pedidos.



Disco con 3 platos, cada uno tiene dos lados y se tienen entonces 6 cabezas. Lo que se lee en el disco como unidad mínima es un sector. La pista son todos los sectores que le dan vuelta al plato. El cilindro son todas las pistas que están en esa posición del cabezal.

La idea es leer primero todo lo que está en el mismo cilindro para ser más eficientes, pero hay otros esquemas de planificación.

- FIFO. No funciona bien, la cabeza se mueve de acá para allá.

- Shortest Seek Time First. Ver dónde está el cabezal y obtener el pedido más cercano a la posición actual. Puede producir inanición, por quedarse siempre en un mismo rango.
- Elevator. El cabezal avanza en algún sentido (creciendo en el número de cilindros, por ejemplo) y atiende los pedidos que van en esa dirección. Si llega un pedido tal que el cabezal ya pasó, se demora hasta que el cabezal cambie el sentido.

De todas formas, todo esto se combina con prioridades. Por ejemplo, si es necesario el acceso a disco para hacer swapping, es una operación que tiene que ser rápida, por lo que se le da prioridad.

Discos de estado sólido

El módulo del SO que maneja las prioridades de E/S para el acceso a discos magnéticos es completamente inútil con estos discos, el acceso a los pedidos es uniforme, aunque tienen otras particularidades. Por ende, es necesario contar con un SO actualizado que pueda contemplar esta tecnología.
Estos discos, de todas formas, tienen un pero. La escritura es más compleja. No están permitidas las sobreescrituras, es necesario primer borrar y luego escribir. Además, no se pueden escribir infinitas veces y cuando se borra se debe borrar un espacio más grande del que se quiere escribir.
Pensar que hay archivos que son fijos (propios del SO, archivos como videos descargados, etc), y también hay otros archivos de cache, o de registros que se modifican con gran frecuencia.
Entonces estos discos deben ir reorganizando la información. Mover datos que están fijos para dejar lugar a nuevas escrituras y así mantenerse por más tiempo. Sin embargo, la escritura con el tiempo tiende a degradarse.

Spooling

Es una forma para manejar dispositivos que requieren acceso **dedicado** como, por ejemplo, una impresora de red. Con esta técnica, el proceso no se bloquea hasta que termina.
Para evitar los bloqueos, todos los procesos quedan en una cola de trabajo pendiente, y se van desencolando. De esta forma, el kernel no se entera del spooling, y para él, la E/S finalizó.

Protección de la información

¿Que implica proteger? Que la información sea accedida y modificada solo por quien/es esta autorizado, y que la información esté disponible cuando el usuario autorizado la necesite.
Debe tomar en cuenta el valor de la información que se protege, qué pasa si se pierde, etc.
Luego, se debe tomar una política de resguardo (backup), que consiste en resguardar lo importante en otro lado. Por lo general es una tarea que toma tiempo y se la suele en el menor período de actividad del sistema.

Consistencia de Backup

--- 570 silber
Es importante tener en cuenta este tema, ya que muchas veces el estado de los archivos que estoy backupeando al inicio del proceso, es distinto al del final. La forma más sencilla es poner offline el dispositivo hasta que termine. Hay otras formas que implican que mientras se hace backup se guardan unos archivos de cambios y lo que se backupea es el estado al momento que se inició el backup.
Otro problema es que copiar *todos* los datos puede ser muy costoso, y muchas veces no se modifican el 100% de los datos entre backup y backup. Entonces surge una estrategia que consiste solo en backupear la diferencia entre cada backup, y cada cierto período de tiempo más largo efectivamente hago un backup completo. Se puede hacer de forma incremental, es decir la diferencia desde la última copia incremental. O diferencial, que siempre copia las diferencias que hay con la copia original.

Redundancia

--- 484 silber
Todo lo que no se backupea se pierde, y los backups no pueden hacerse literalmente todo el tiempo. Además, restaurar desde un backup puede llevar mucho tiempo. Entonces se implementa siempre algún tipo de redundancia de los datos.

RAID 0

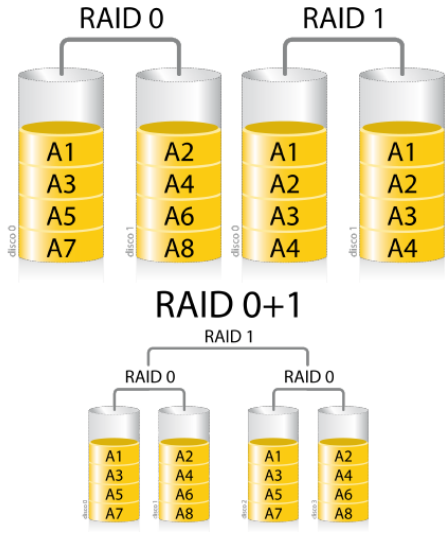
No aporta redundancia, pero distribuye los bloques de un archivo en varios discos entonces aumenta el rendimiento general de ambos, distribuyendo la carga.

RAID 1

La forma más sencilla para esto, es tener discos espejados y al escribir, se escribe en ambos. Es costosa porque se necesita el doble de almacenamiento, pero es más eficiente para lecturas porque se pueden distribuir las lecturas entre los dos discos.

RAID 0+1

Combina las técnicas anteriores. Tiene dos sistemas RAID 0 espejados. Es más costoso, pero el rendimiento no es malo.

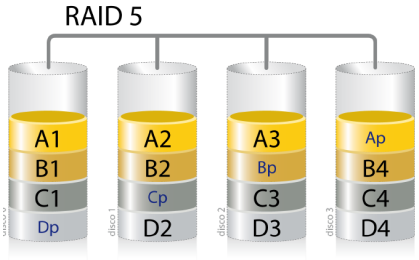


RAID 2 y 3

Se combinan con RAID 0 y también distribuyen la información. Además, por cada bloque, guarda información adicional para determinar si se dañó y en ese caso, intentar regenerar el bloque. RAID 2 necesita más discos de paridad que RAID 3. Todos los discos participan de todas las E/S asi que es más lento que RAID 1. No se suele usar esto.

RAID 5

Este suele ser utilizado junto con 0 y 0+1. Usa datos redundantes, pero los distribuye entre todos los discos, no hay un único disco con redundancia. Es costoso el mantenimiento de la paridad distribuida, **pero no importa que disco se rompa, es posible gracias a los otros restablecer la información.** Además, es en caliente, con una merma en la performance, pero sin ponerse offline.



Raid 6 es similar, pero tiene un segundo bloque de paridad, por lo que soporta que se rompan hasta dos discos.

Sistemas de Archivos

--- 543 de silber; 253 tanen

Un archivo es un conjunto de datos, una secuencia de bytes que comparten ciertas características, y que **no tienen estructura.** El SO no sabe si es un video, un archivo de texto, una imagen. Además, tienen un nombre para

identificarlo, y opcionalmente pueden incluir una extensión que el SO no valida, pero que se utiliza para determinar cómo se debe interpretar el contenido.

Existen múltiples tipos de FS, algunos SO solo soportan uno (DOS solo soporta FAT), otros soportan más de uno, y otros como UNIX mediante módulos de kernel pueden soportar casi cualquiera. En la metadata del HW está el tipo de FS que tiene.

Lo que hace a un FS mejor a otro es la eficiencia, la protección, capacidad de recuperación ante una caída inesperada. También cuestiones de tamaño. **FAT32 por diseño no soporta archivos de más de 4GB. FAT NO maneja permisos.**
Los FS también difieren y regulan el límite de los nombres, que no es solo el nombre del archivo en sí, sino que incluye el path al mismo.

Responsabilidades del FS

Deben resolver como una de las formas más elementales, cómo se organizan de manera lógica los archivos, para poder acceder a ellos. Todos los FS soportan el concepto de directorio, que hace que la organización sea jerárquica, con forma de árbol (Windows agrega el concepto de unidades, lo que hace que tenga varios árboles).
Además, casi todos los FS soportan alguna versión de *link* (shortcut). Un link es un alias, otro nombre para el mismo archivo. Con links, el árbol deja de ser tal y pasa a ser un grafo dirigido, con la posibilidad de que se generen ciclos si un archivo tiene un link de otros dos archivos distintos.

Otra responsabilidad del FS es definir los caracteres de separación de directorios, decidir si deben o no tener extensión, restringir la longitud, los caracteres permitidos para la ruta, distinguir mayus de minus.

```
/usr/local/etc/apache.conf
C:\Program Files\Antivirus\Antivirus.exe
\\SERVIDOR3\Parciales\parcial1.doc
```

Punto de Montaje

Si tengo más de un disco, o uno con varias particiones, debo poder referenciarlos, y el SO debe saber de alguna forma que los discos están colocados, conocer el punto de inicio del grafo del mismo y a partir de que ruta local disponibilizarlo. Eso es montaje. El nombre hace referencia a la operación que se hacía con las cintas.
Un SO puede tener montados varios discos, tanto locales como remotos (NFS), y para el usuario es algo totalmente transparente. Linux hoy ofrece la función automount, que resuelve estas decisiones automáticamente.

Representación de un archivo

Para el FS un archivo es una lista de bloques de datos + metadata, la cuestión está en cómo almacenarlos.

Bloques Contiguos

La forma más sencilla de representarlos sería poner todos los bloques de cada archivo contiguos en el disco. Si bien las lecturas serían óptimas, los archivos pueden crecer indeterminadamente y además generaría fragmentación.
Esta idea solo es buena en FS de solo lectura (CDS: ISO 9660)**.**

Lista enlazada

La alternativa es la lista enlazada de bloques, donde cada bloque además de los datos tiene un puntero al bloque siguiente, tal que físicamente los bloques puedan guardarse no consecutivamente. Las escrituras son un poco más lentas, el disco podría tener que moverse para todos lados, pero la principal desventaja es que **siempre es necesario recorrerlo todo.**

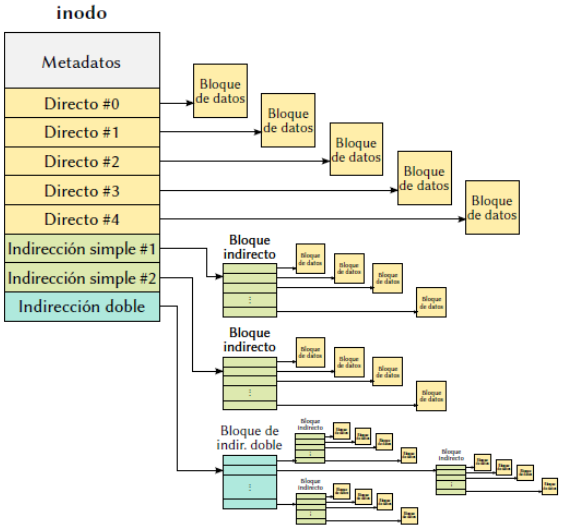
FAT (File Allocation Table: Dónde están los archivos)

La idea para mejorar la lista enlazada es tener una tabla donde, por cada bloque se indique dónde está el bloque siguiente. Lógicamente también se necesita una tabla adicional para indicar que determinado archivo empieza en

determinado bloque. Entonces en vez de recorrer todo el archivo, alcanza con recorrer solamente la tabla para tener un acceso aleatorio a cualquier parte de cualquier archivo, además la tabla está en memoria, por lo que es más eficiente.
La desventaja es justamente esa, que **toda la tabla del disco debe almacenarse en memoria,** y puede pasar que la tabla actualizada por algún imprevisto no pueda bajarse a disco y se corrompa. Otro problema es la contención que genera esta única tabla en memoria.
FAT no maneja permisos, tiene fecha de modificación, nombre y tamaño de archivos como metadata, no más que eso.

Inodos (Unix)

Cada archivo tiene uno. Dentro del inodo se tienen atributos como tamaños, permisos, y metadata (no el nombre).
Además, contiene las direcciones de **algunos** bloques de datos, lo que permite acceder muy rápidamente a archivos pequeños.
Para archivos de tamaños más grandes, también agrega una entrada a un bloque de acceso indirecto simple. Ese bloque tiene punteros a bloques de datos del archivo, y gracias a eso puede referenciar más bloques que antes, permitiendo tener archivos más grandes.
Luego se tiene un segundo nivel de indirección que soporta tamaños más grandes, y hasta un tercer nivel de indirección también.
Esto permite tener en memoria solo las tablas de los archivos abiertos, además se tiene una tabla por archivo lo que reduce la contención.
A medida que el archivo crece y debo acceder a indirecciones las lecturas son sensiblemente más lentas.



Árbol de directorios con Inodos

Los directorios también son archivos, entonces también tienen un inodo. Existe un inodo como entrada al directorio root. Por cada archivo dentro del directorio hay una entrada que son inodos con el nombre de archivo o de otros directorios.

Atributos

En la metadata de los archivos (dentro de su inodo) se almacenan los permisos, el tamaño, los propietarios, fechas de modificación, creación, tipo de archivo, flags. **El nombre no es parte de la metada del archivo, está en el directorio.**

Manejo del espacio libre

Para saber que bloques están libres, no es útil tener que recorrer todos los inodos. Se necesita una estructura de datos para saber que bloques están libres y cuales ocupados.

Una posible solución es el mapa de bits, pero requiere tener el vector en memoria, también se puede tener una lista enlazada.

Caché

Es decir, tener bloques del disco en memoria para consultar rápidamente. Otra ventaja que genera el uso de cache es que le permite al planificador de E/S armar una forma más eficiente de escribir cuando lo necesita, ya que cuenta con esta memoria que mantiene los archivos.
Esto puede traer un problema, ya que para el usuario el archivo ya fue escrito cuando en realidad, todavía el módulo de E/S lo tiene en memoria y si hay una caída repentina del sistema los datos podrían perderse. Existe una syscall llamada fsync que fuerza a grabación a disco, sin importar la degradación de performance.

Consistencia

A pesar de contar con fsync, el sistema podría caerse en cualquier momento. En UNIX existe un programa fsck que restaura la consistencia del FS. Además, el FS cuenta con un bit que indica si el apagado fue normal o es necesario correr ese programa.

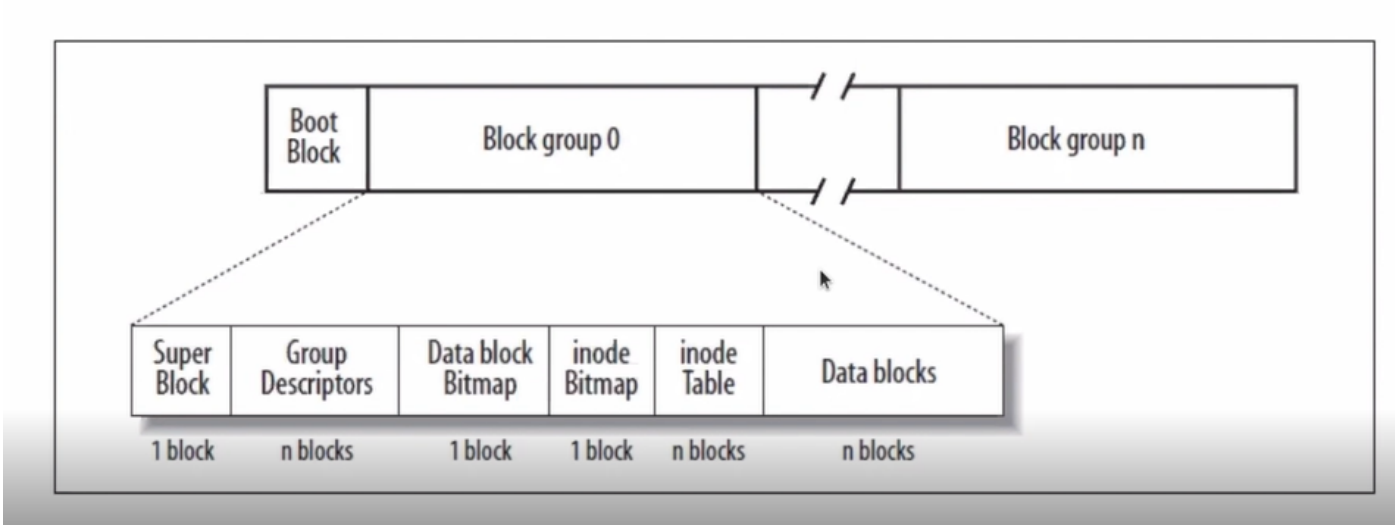
Journaling

La mayoría de los FS modernos llevan un log o un journal, que básicamente es un registro de los cambios en la metadata que hay que hacer (cambios en los inodos), no todos los cambios de todos los datos. Cuando se baja el caché a disco, se actualiza una marca indicando qué cambios ya se reflejaron.
Si bien el journaling impacta en la performance, es bajo.
Una vez que el log quedó efectivamente escrito, es que empieza la operación en la estructura del FS. Por eso, si el sistema se cae abruptamente, en el log quedó reflejado que cambios aún no se habían aplicado, haciendo que no se pierda consistencia en el FS (datos si puede perder, pero no estructura del FS).

NFS

--- 792 tanen;
El Network File System es el que permite acceder a discos remotos o virtuales como si fueran locales, haciéndolo transparente al usuario (pero sin transparencia de ubicación ni independencia). Para esto, los SO incorporan un Virtual File System (VFS), que tiene vnodes por cada archivo abierto. Los vnodes (796 tanen) son la correspondencia a los inodos en el mundo local.
Sabemos que los procesos acceden a los archivos de manera transparente, entonces el kernel a través del VFS determina si el archivo (gracias al punto de montaje del inicio del path del archivo) es local o virtual y lo despacha correspondientemente. NFS no resuelve bien la transparencia ya que hay que indicarle la locación del servidor en el que está el archivo.
La contra de esto es que tiene un único punto de falla si el server se cae, y además es un cuello de botella para todos los pedidos.

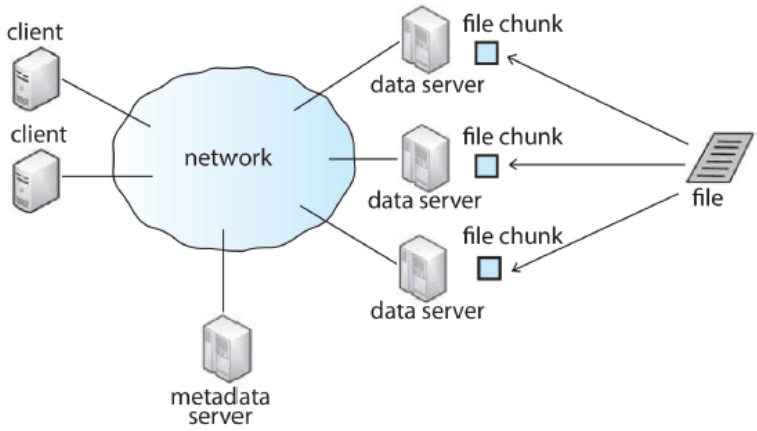
Ext2



Tiene un bloque de booteo que indica la dirección para la carga del SO. Luego tiene n Block Groups con la misma estructura.
El super bloque tiene la información necesaria de como acceder al resto de la información, como el tamaño del bloque. Además, sabe en donde comienza el resto de los bloques, etc
Luego posee un bitmap con los inodos libres y ocupados, y un bitmap con los bloques de datos libres o no, los descriptores de los grupos y finalmente los bloques de datos y los datos de los inodos.

FS Distribuidos

Es un sistema de archivos cuyos clientes, servidores y dispositivos de almacenamiento están distribuidos entre las máquinas de un sistema distribuido. Los usuarios/procesos (clientes) que consumen están en distintas máquinas.
Como principal característica, el cliente tiene que poder acceder a toda esta información de manera transparente.
El principal desafío es la **administración** de los dispositivos de almacenamiento dispersos.



Modelo Cliente-servidor

La manera que esto funciona es mediante un módulo de kernel que implementa un modelo Cliente-Servidor.
Los archivos pueden estar en cada server o por pedazos como en la imagen. Lo importante es que se cuenta con servidores de datos, servidores de metadatos y storage (racks con muchos discos).

Los clientes contactan al servidor para pedirle archivos. El servidor es responsable de la autenticación, chequeo de permisos y envío de archivos, por partes, encriptado, etc.

Si el cliente hace cambios en el archivo, esto debe ser propagado al servidor y replicado correctamente en los lugares donde el archivo esté almacenado. Ejemplo típico: NFS.

Modelo basado en Cluster

Son más resistentes a fallas y más escalable que el modelo cliente servidor convencional. Ejemplos son Google File System y Hadoop File System, para este modelo, **el SO debe tener un módulo de Kernel que soporte esos FS.**
Como vimos anteriormente en la imagen, los clientes se conectan al servidor de metadatos y hay varios servidores que contienen los archivos o porciones de un archivo.
El servidor de metadatos tiene un mapeo de los servidores en los que está el archivo. Además, las porciones de archivos pueden estar replicadas n veces por cuestiones de redundancia y de obtener más puntos de acceso para reducir la latencia y evitar cuellos de botella.
El problema de esto es que, al momento de escribir archivos, tendremos problemas de coherencia y sincronización.

Nomenclatura y transparencia

El problema de la nomenclatura tiene que ver con, dado un nombre de un archivo, resolver donde está el objeto físico de ese archivo. Además de en qué servidor, hay que resolver en que disco está en ese servidor. Más aún se deben devolver el conjunto de todas las ubicaciones de las réplicas del archivo si las tuviera, aunque al usuario se le oculta esta información y se le devuelve un único punto para leer.
Un File System Distribuido transparente debe ocultar por completo esta información, el nombre del archivo no revela la ubicación física del mismo, además el nombre tampoco debe cambiar cuando cambie su ubicación física. En la práctica para resolver esto, los DFS tienen una especie de DB en el servidor de metadatos donde mapean el nombre que ve el usuario con información más concreta sobre la ubicación física de cada chunk de cada archivo.

Esquema de nombres

Existen tres enfoques:

- Combinando el nombre del host con el nombre local. Como en NFS, funciona, pero no es independiente ni transparente de ubicación
- Montar directorios remotos en directorios locales. Se tiene un árbol más grande de directorios, o varios árboles en Windows. Solo los directorios montados previamente pueden accederse de manera transparente.
- Única estructura global de nombre que abarque a todos los archivos del sistema. Garantiza transparencia e independencia, pero si un servidor no está disponible, arbitrariamente un conjunto de directorios tampoco lo estará.

Acceso a un archivo remoto

A qué servidor hay que acceder fue resuelto por el servidor de metadata y la resolución de nombres. El problema ahora es transferir efectivamente los datos.
El módulo de kernel establece una conexión TCP hacia los servidores remotos pidiendo el archivo una vez que habló con el server de metadatos. Es decir, se comporta como un cliente de alto nivel para una operación que en un entorno local sería de bajo nivel.
Para mejorar la eficiencia de estos archivos se implementan cachés locales, donde el server informa el time to live de la información que brinda. Entonces, durante ese tiempo se puede acceder al cache sin problemas y manejar la información localmente.
También existen cachés de escritura para mejorar el tiempo de respuesta y eventualmente se escribirá físicamente. Con un sync, igual que en un entorno local, se puede lograr que la escritura sea en el momento.
Existen alternativas de escritura como escribir apenas el cache se modifique (poco performante), o retrasar las escrituras, aunque es menos confiable. Si se retrasan se puede escribir en intervalos regulares o una vez que el archivo se cierra.
Tener cachés de escritura es eficiente en tiempo, pero genera un problema de consistencia frente al master del archivo.

Consistencia

El cliente puede indicar que es importante que se mantenga la consistencia, entonces al escribir o leer, se demora la operación para previamente realizar un chequeo de consistencia.
En otro enfoque, el servidor tiene un registro de todos los que están cacheando información. Cuando detecta que alguien tiene una copia inconsistente, hace un push de un update para actualizar el archivo.
Si los datos existen en más de un servidor, es necesario utilizar protocolos de sincronización de sistemas distribuidos, pero algunos FS para mitigar el problema, ponen limitaciones a las escrituras del archivo, como que sean solo append-only y un solo escritor por archivo. Entonces mantener la consistencia es más simple.

Sistemas Distribuidos

Es un conjunto de recursos conectados que interactúan entre sí, como ser varias máquinas conectadas a la red, un procesador con varias memorias, varios procesadores que comparten varias memorias, etc
La gran ventaja de estos sistemas son la replicación, el paralelismo y la **descentralización** de los datos.
A su vez, los grandes desafíos consisten en sincronizar datos, mantener coherencia y algo muy importante: **No comparten clock.**

Conjetura de Brewer: En un entorno distribuido no se puede tener a la vez consistencia, disponibilidad y tolerancia a fallas. Solo dos de tres.

Memoria No Compartida

Existen arquitecturas de software para manejar procesos distribuidos cuando no se tiene memoria compartida.

RPC

Mecanismo para que los programas (en C en un ppio) puedan hacer llamadas a procedimientos de forma remota.
Un programa puede necesitar ejecutar una parte de su código en otra máquina ya sea por alta necesidad de computo o por ejemplo por no contar con la base de datos necesaria localmente para hacer lo que debe hacer.
Entonces, el compilador que entiende RPC, crea dos programas distintos: el que necesita ejecutarse remotamente y el original local. El compilador agrega la interfaz interna necesaria para poder generar la comunicación remota entre los procesos remotos. Luego cuando se tiene que ejecutar el programa, los parámetros se envían mediante la interfaz creada por la red remotamente, y se retorna el resultado de la función.
Para este mecanismo necesitamos ayuda del kernel y se crea un server que se queda escuchando en algún puerto, el server solo responde a los pedidos, la comunicación es asimétrica. **Es sincrónico.**

RPC asincrónico

Los Future vienen de acá. Una variable Future todavía no sabemos cuál es su valor, pero si sabemos su tipo. Luego, definimos un handler para invocar una rutina una vez que el pedido está resuelto. Se interrumpe el flujo normal del programa para invocar los handlers, pero no es una interrupción a bajo nivel, sino a nivel comunicación entre procesos.
La idea general detrás de esto es el la de **pasaje de mensajes,** donde no suponemos que nada se comparte, solo se tiene entre los procesos un canal de comunicación.
Para la comunicación asincrónica, el mensaje se manda y se sigue la ejecución del programa. Mientras tanto, se implementa un mecanismo de polling a un socket o file descriptor para chequear si la respuesta llegó.

La complejidad en algoritmos distribuidos va a estar dada en contar la cantidad de mensajes que se mandan.

Modelo de Fallas

Cuando se trabaja con algoritmos distribuidos es importante determinar esto, ya que las fallas en los sistemas distribuidos suelen ser más la regla que la excepción.

- Nadie falla. Es decir que se puede esperar que los resultados sean correctos si no hay fallas. Si bien parece ingenuo, estos modelos se combinan con monitoreo de red.
- Los nodos mueren, pero no reviven. Resulta anti intuitivo, pero que un nodo se recupere puede ser un problema grave porque puede estar manejando información inconsistente.
- Los nodos mueren y pueden revivir, pero en determinados momentos.
- La red se parte. Los cómputos avanzan independientemente sin comunicación.
- Los nodos se comportan de manera impredecible.

Locks Distribuidos

No contamos con registros atómicos como TestAndSet, ya que no tenemos un único HW que nos pueda proveer esto. Una de las soluciones más elementales de esto son los proxies (procesos que representan a los nodos remotos) en un nodo único que hace de árbitro. Cuando un nodo remoto pide un recurso, le manda un mensaje al proceso remoto que lo representa en el nodo único, y dentro de este nodo se resuelve con los mecanismos conocidos. La clara desventaja de esto es el **único punto de falla**, en el nodo árbitro. Además, se genera un cuello de botella no solo de procesamiento del nodo sino de capacidad de red.

Como alternativa hay que resolver el problema de que un nodo pueda "avisarle" a los demás que es el primero. No es trivial descifrar que nodo envía primero un mensaje, u ordenar los mensajes cronológicamente.
Sincronizar relojes con mucha precisión es caro y difícil.

Orden Parcial entre eventos

Lamport propone que lo único que necesitamos es ponernos de acuerdo en un orden de suceso de eventos, y no es importante saber cuándo sucedieron. Por eso sugiere un **orden parcial no reflexivo** entre eventos.

- Dentro de un mismo proceso, si A pasa antes que B (a modo tradicional), A->B
- Si E es el envio de un mensaje y R su recepción, E->R, no importa si sucede en procesos distintos.
- Si A->B y B->C, A->C (transitividad)
- Si no vale ninguna de estas reglas, es decir ni A->B ni B->A, entonces A y B son concurrentes. Por eso el orden es parcial, "sucedieron al mismo tiempo"

Para la implementación se usa un reloj o una función monótona creciente. Cada mensaje que se envía toma un valor de este reloj. Cuando se recibe un mensaje se chequea el valor t que tenía el mensaje y el reloj interno del receptor se actualiza a t+1 (segunda condición de Lamport).
Para generar un orden total, se deben romper los empates por algún otro criterio arbitrario, por ejemplo, el pid de cada proceso en el empate.

Problemas Distribuidos

Las grandes familias de los problemas en entorno distribuidos, tiene que ver con el orden de ocurrencia de los eventos, exclusión mutua y consenso.

Acuerdo Bizantino

El problema consiste en que todos los procesos distribuidos se pongan de acuerdo en tomar la misma decisión. Este problema formalmente no tiene solución (salvo para situaciones donde no se pierden mensajes, y solo se caen procesos), entonces se buscan dar aproximaciones de probabilidad de falla y minimizar esa probabilidad para distintos escenarios.

Exclusión mutua distribuida

Una técnica es la del **token passing**. La idea es generar un anillo entre los nodos y poner a circular un token. Aquel proceso que tiene el token es aquel que puede entrar a la sección crítica.
Lo difícil de este algoritmo es que se deben conocer los procesos involucrados de antemano, y cada nodo además debe saber quién es su anterior y cuál es su siguiente. Por eso no es muy resistente a fallas. Una particularidad es que no produce inanición si no hay fallas, aunque todos acceden a la sección crítica aún de forma innecesaria.

Otra alternativa para lograr mutex es que cada nodo que quiere entrar manda un mensaje *Solicitud(Pi, ts)* al resto de los nodos y a el mismo, donde ts es el timestamp (o un número monótono creciente de Lamport). Solo se puede acceder a la sección crítica cuando todos los demás aceptan.
Cada nodo responde *sí* inmediatamente si no le interesa entrar a la sección crítica o si bien quiere entrar, aún no lo hizo y el ts del pedido ajeno que tiene es menor al del ts del pedido propio, ya que el otro pidió antes.
Este enfoque también requiere conocer todos los nodos de la red y su estado y que no se pierdan mensajes.

Locks Distribuidos (cont)

El **protocolo de mayoría** consiste en obtener un lock de un objeto que esta copiado en n lugares de la red. Además, cada copia del objeto tiene un número de versión. El pedido de lock se manda a los n lugares del objeto, sin embargo, se considera obtenido el lock si al menos la mitad más uno dio el OK. El proceso que escribe el objeto incrementa la versión con la versión mayor de los procesos obtenidos más uno. Al obtener la mitad más uno del consenso entonces se asegura que no podría estar leyendo una versión desactualizada de los valores.
Puede predecir deadlock por ejemplo si hay nodos pares y se toman mitad y mitad.
Lo lindo de este protocolo, es que soporta que parte de los recursos compartidos se caigan, ya que, aun así, con la mitad más uno se puede seguir escribiendo, y cuando la parte caída se recupere, eventualmente se sincronizará.

Elección de líder

Una serie de nodos debe elegir a un único proceso como líder para tomar una tarea. Un caso real de esto pueden ser distintos servidores atendiendo pedidos.
En una red sin fallas, todos los nodos tienen un status que indica que nadie es el líder. La red está organizada en anillo, y los nodos hacen circular un Id o valor que indique quién lo envía. Cuando un nodo recibe un mensaje con el valor, pasa a su compañero el mayor entre el que recibió y su propio valor. Al completar la vuelta se tiene al líder, y se hace una nueva vuelta de notificación.
Este protocolo puede complicarse si se realizan varias elecciones en simultáneo.

Instantánea Global Consistente

Se desea conocer el **estado total** de la red, un snapshot del estado global consistente. Lo único que modifica los estados son los mensajes que se mandan los procesos entre sí.
Cuando un proceso quiere el snapshot se envía a si mismo *marca.*
Cuando un proceso recibe *marca* (no importa de dónde vino), guarda una copia de su estado y les envía *marca a todos los otros procesos.*
Además, el proceso que inició empieza a registrar los mensajes *marca* del resto de todos los procesos, hasta tanto haya recibido el mensaje de todos.
En ese momento cada proceso tiene una secuencia de estados recibidos antes de que el primer proceso tome la instantánea.
Ejemplo con banco y dos nodos: A le envía $50 a B. Snapshots validos: O bien no se hizo aun la transferencia o bien B ya los recibió. Casos posibles:
-) A manda $50 y luego marca. B recibe $50 y luego el mensaje marca, entonces envía a A marca y su estado con los $50.
-) A manda marca y luego los $50. B recibe marca y reenvia, y luego los $50.
Se ve claramente que se requiere que los mensajes lleguen en orden y que los pedidos también se atiendan en orden.
Sirve para debugging y detección de deadlocks.

Two Phase Commit

Muy usado en DB Distribuídas para ponerse de acuerdo en si un valor se puede escribir.
Se desea hacer una operación atómica. Todos los nodos deben estar de acuerdo en si se puede o no se puede.
1 Fase: Un nodo pregunta al resto si están de acuerdo en que se haga la operación. Si se recibe un no, se aborta. De lo contrario se anotan todos los que aceptaron. Si pasa un tiempo máximo y no se recibieron todos los sí, se aborta.
2 Fase: Con todos los sí recibidos, se les avisa a todos que la operación quedó confirmada porque todos aceptan.

Protege contra muchas fallas, pero no contra el hecho de que se caiga un nodo en la segunda fase. No garantiza que todo proceso que no falla decida, ante eso se implementa Three Phase Commit.

Seguridad (de la información)

Se basa en la preservación de tres características

- **Confidencialidad**: Que solo puedan acceder a ciertos datos las personas que están autorizadas a acceder
- **Integridad**: Que solamente pueda modificar un dato quien esté autorizado, y también poder determinar quién originó un documento.
- **Disponibilidad**: Que la información esté disponible para los usuarios autorizados cada vez que ellos quieran ingresar.

La idea general dentro de seguridad es poder decir qué sujetos pueden realizar qué acciones sobre qué objetos.

Las 3 A

- Autenticación: Como demostrar a un sistema la identidad de un usuario. Contraseñas, datos biométricos. Se usan fuertemente conceptos de criptografía. Se puede utilizar más de un factor de autenticación.
- Autorización: Incluye las acciones que pueden realizarse en base a qué permisos se tienen dentro del sistema.
- Accounting: Es el logging, el registro de las operaciones que se hicieron para generar trazabilidad y rastreabilidad.

Criptografía

Se encarga del cifrado y descifrado de información. La idea es generar un registro ilegible que solo pueda entender quien conozca el mecanismo de descifrado y tenga las claves pertinentes para eso. El criptoanálisis estudia los métodos para quebrar textos sin tener la clave y estudiar la robustez de los algoritmos.

Existen algoritmos de encriptación **simétricos** que son aquellos que utilizan la misma clave para cifrar y descifrar. Ej: Caesar, DES, AES. Un ejemplo básico de esto son las claves que desfasan en n posiciones las letras del abecedario, entonces por ejemplo la *a* se transforma en la *e*, etc.
AES se usa hoy en día e Intel tiene primitivas para usarlo rápidamente.
En este mecanismo de información el problema radica en cómo comunicar la clave. Entonces para eso existen los algoritmos **asimétricos**, que usan claves distintas, y generalmente se usa el concepto de clave pública y privada. Ej: RSA.
Clave pública y privada funciona así: El sujeto A posee una clave privada que solo él tiene y una pública que puede tenerla todo el mundo. Cuando alguien le quiere mandar un mensaje, lo encripta con esa clave pública, y el único que puede desencriptarlo será A ya que se hace con la clave privada.

Hashing

Otro concepto importante son las funciones de Hash one-way *criptográficamente seguras* como ~~MD5, SHA1~~, SHA256.
En criptografía se suele pedir que se cumplan ciertas reglas:

- Resistencia a la preimagen. Dado un hashing debería ser difícil encontrar un mensaje que devuelva el mismo hashing.
- Resistencia a la segunda preimagen. Dado un mensaje m1, debería ser difícil encontrar un mensaje m2 distinto con hashes iguales.

Las funciones de hash son determinísticas y siempre tienen largo fijo. Con la resistencia a la segunda, se garantiza la integridad, ya que nadie puede adulterar un archivo sin que también cambie el hashing del mismo.

Observación: No es una buena recomendación usar SHA256 para almacenar claves. Si obtuviéramos el hash de una clave e ir probando hashes de distintos mensajes hasta obtener el hash original. Y hacer eso con SHA256 se podría hacer muy rápido, algo bueno para el hashing en sí, pero malo porque me permite probar muchas claves en una unidad de tiempo.
Para solucionar esto y evitar un ataque con tablas previamente computadas, una alternativa es que la función de Hash no de siempre lo mismo. Para eso se utiliza un SALT. Cuando se genera una clave nueva, el sistema elige al azar un SALT y lo utiliza para el hashing. El resultado es SALT+Hashing. De todas formas, sigue funcionando el ataque de fuerza bruta hasheando claves con el SALT del usuario.

Para que el mecanismo de prueba sea más lento para el atacante, se suele para cada clave, aplicar muchas veces la función de hash.

RSA

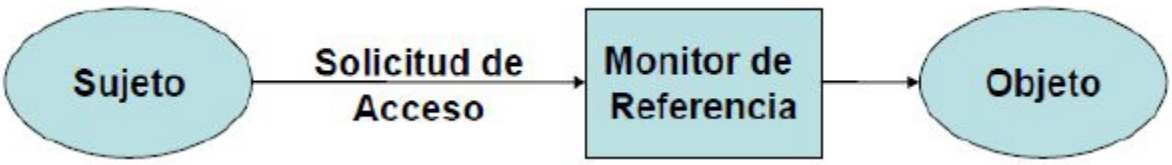
Es un método asimétrico de clave pública y privada. Se toman dos números muy grandes (primos), cada uno hará de cada clave. La privada no se difunde.
Para encriptar un mensaje, se interpreta cada letra como un número y se encripta con la clave pública del receptor. El receptor con la clave privada hace la cuenta inversa y desencripta.
La seguridad de este método se basa en la dificultad de factorizar números grandes (problema NP).
Con RSA puedo generar una firma digital de un documento. Se hashea el documento y se encripta **con la clave privada**. Se envía el documento + hash encriptado.
El receptor del documento como tiene la clave pública desencripta (y valida autenticidad del usuario) y obtiene el hash del documento sin cifrar. Luego genera el hash del documento y compara. De esta forma se sabe tanto que el usuario era el que decía ser, y por otro lado que el documento no se modificó.

Replay attack

Cuando viaja una password por la red, a pesar de estar cifrada, si la red no es segura se puede sniffear y luego poder acceder en cualquier otro momento, ya que el servidor cuando des-hashee le va a otorgar acceso.
Para combatir eso se utiliza un mecanismo de Challenge Response. Aquí el servidor elige un número al azar y se lo manda al cliente. El cliente encripta la password usando ese valor como semilla y envía eso por la red.
De esa forma el atacante puede obtener el hash basado en el número de ese momento, pero no puede usarla nuevamente en otro ataque ya que el servidor habrá cambiado la semilla.
No es infalible ya que con el challenge y el hasheado se podría volver a obtener la password.
Los métodos de hoy en día se centran en cifrar todo el canal de comunicación.

Permisos

Una vez autenticado dentro de un sistema, es necesario tener en cuenta qué permisos se tienen dentro de ese sistema. Los SO tienen un componente que es el **monitor de referencias** y es el encargado de mediar cuando los sujetos intentan realizar operaciones sobre los objetos.



Todos los pedidos de acceso deben pasar por ese monitor para chequear siempre los permisos necesarios.

DAC (Control de acceso discrecional)

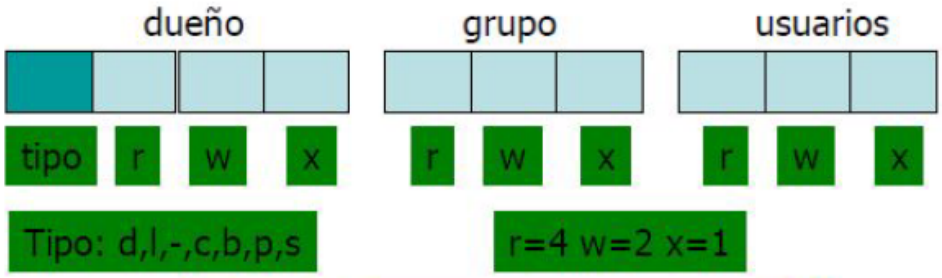
De manera más sencilla se puede pensar como una matriz de sujetos por objetos, tal que en cada celda se tengan los permisos que cada sujeto tiene sobre cada objeto. Es una matriz muy rala, y generalmente, se almacena por filas o columnas y por cuestiones implementativas son los archivos los que suelen guardar qué puede hacer cada usuario con él (podría ser al revés).

Por lo general a la hora de otorgar permisos, el principio más común es el de **mínimo privilegio** para cada usuario sobre cada archivo. También existen permisos por defecto para ciertos objetos que varían según el tipo de objeto.
Se llama discrecional porque el dueño de un objeto, discrecionalmente puede dar permisos a otros usuarios, incluso dar permisos de más por error.

MAC (Control de acceso mandatorio)

A diferencia de DAC, contiene reglas que no se pueden romper que están definidas a nivel de sistema. Cada usuario tiene un grado, y los objetos creados heredan el grado del último sujeto que los modificó.
Cada sujeto puede acceder a objetos de grados menores o iguales que su grado. Nadie de mayor grado puede otorgar permisos altos a alguien de menor grado.
Este esquema se usa para manejar información sensible.
Ej: BELL-LAPADULA.

DAC en UNIX



Además de esos permisos básicos existen el SETUID y SETGID. Son permisos de acceso especial.
Por ejemplo, los usuarios no pueden acceder al archivo de contraseñas etc/shadow ni siquiera para lectura. Sin embargo, un usuario tiene que poder cambiar su propia contraseña. El comando passwd utiliza SETUID, que se visualiza como una s cuando se visualizan permisos de archivo.
Cuando cualquier usuario ejecuta passwd, como tiene SETUID prendido, hereda los permisos del owner de passwd, entonces podrá modificar contraseñas. Si hubiera un bug de seguridad en passwd, todo el código ejecutaría en modo root, lo que es muy peligroso.
Es decir que con SETUID se heredan permisos del owner y cambia el EFFECTIVE USER ID.

Además, Linux tiene una lista de permisos extendidos (**chattr)** que permiten entre otras cosas que los archivos sean inmutables y nadie, ni siquiera el dueño, pueda borrarlos.
También se pueden agregar permisos extendidos, llamadas **POSIX ACL** que permite a usuarios y grupos específicos explícitamente cambiarle los permisos.

MAC en Windows

Se basa en el modelo Biba, y define cuatro niveles de integridad: Sistema, Alto, Medio y Bajo. Por defecto, todos los recursos corren con nivel medio, los ejecutados como Administrador en nivel alto, y no es posible asignar a un proceso o usuario un nivel de integridad mayor al propio. Sí es posible asignar privilegios más bajos. Un ejemplo de esto es Chrome, que crea algunos procesos con nivel bajo para evitar que softwares maliciosos puedan modificar archivos del usuario.

Seguridad de Software

Existen varios mecanismos en distintos momentos de vida de la aplicación.

- Arquitectura / Diseño: En el momento de pensar y armar la aplicación. Por ejemplo, si se va a decidir cifrar la red o no. Son los problemas más difíciles de solucionar.
- Implementación: Son problemas en el código, que permiten a los atacantes aprovechar esas vulnerabilidades y atacarlo.
- Operación: Problemas de configuración cuando la aplicación está productiva.

Errores de implementación

Generalmente son errores más fáciles de solucionar que los de diseño. Un error de diseño puede implicar cambios estructurales y de compatibilidad muy costosos.
El error de implementación se da en hacer suposiciones sobre el ambiente del programa, por ejemplo, asumir que una entrada se va a dar en un formato particular.

Control de Parámetros

Buffer overflow es una particularidad de un problema de control de parámetros. En general, también puede darse que un mal chequeo de parámetros permita a un atacante correr código malicioso. Es importante poder chequear la estructura del parámetro, el formato, validarlo, etc. Además, es importante que los programas corran con el mínimo privilegio posible.
Un ejemplo clásico es el de SQL Injection, o bien el de agregar comandos a un parámetro que sabemos que va a terminar siendo parámetro de un comando de Linux.

Llamadas a programas del SO

Otro problema es el de llamar dentro de un programa, comandos de SO. Si no especifica correctamente el PATH del binario a ejecutar por asumir que se debería usar el que es por defecto (por ejemplo, una simple llamada a echo, sin especificar /bin/echo). La máquina en la que corre el programa puede tener maliciosamente cambiadas las rutas de variables de ambiente y hacer ejecutar literalmente cualquier código al programa.

Sandbox

Es una forma de aislar a los procesos. En Linux, chroot, le cambia a un proceso el árbol de directorios que ve. La virtualización y el uso de contenedores es otro ejemplo de sandbox.

Principios Generales

- Mínimos Privilegios.
- Simplicidad. KISS. Generar configuraciones de más puede generar problemas de seguridad.
- Validar todos los accesos a datos.
- Minimizar la cantidad de mecanismos compartidos. Por ejemplo, evitar el uso de /tmp/
- Seguridad multicapa. Combinar todos los mecanismos para generar diversos grados de seguridad.
- Facilidad de uso. Poner restricciones muy complejas puede generar problemas. Por ejemplo, obligar a contraseñas muy complejas, o que se cambien muy seguido, puede que un usuario lo anote en un papel y se generó un problema de seguridad de todas formas.

Ejemplos de Ataques

Race Conditions

Se llama así porque el atacante busca generar una condición de carrera que le permita acceder a lugares donde de otra manera no podría. Se aprovecha de la vulnerabilidad TOCTOU, donde se chequea sobre el permiso sobre un archivo y si es correcto se le concede el open, **pero las operaciones no se realizan atómicamente.** Así, la idea es que los permisos se chequeen para un archivo que, si tenga permisos, y en el medio cambiar el valor del puntero a uno que no tenga. Además, se combina con que un programa corra con SETUID y que sea de root para acceder a archivos de máximo privilegio.

Format String

Es otro problema de implementación y es una idea similar a buffer overflow, donde el usuario programador para escribir mediante funciones printf o de log, manda directamente el parámetro del usuario, sin poner los formateadores de string. De esta forma, el mensaje del usuario sí podría tener maliciosamente formateadores que le permiten ejecutar código que el desee, o poder sobreescribir la pila hasta llegar a la dirección de retorno y retornar a una función propia.

La entrada de usuario además, podría ser arbitrariamente grande si se usa un gets para leer, ya que el gets no valida el tamaño.

Buffer Overflow

Cuando se llama a una función en C (no afecta a Java, pero su máquina virtual sí), primero se hace push de los parámetros y luego del IP a la pila. A su vez, las variables locales se guardan en la pila.
El ataque consiste en aprovechar vulnerabilidades que no validen los tamaños de los buffers implicados en alguna operación (por ejemplo, strcpy), y de esa manera **se sobreescribe la pila con código malicioso**, asegurándose que la dirección de retorno luego apunte a ese código, o bien se produce un error en el programa que lo hace caer, o también puede escribir tanto demás hasta pisar la dirección de retorno.

Return to libc

Hoy en día, se puede configurar al compilador para que los buffers sean no ejecutables (NX bit), aunque necesita ayuda del HW.
Sin embargo, las syscall de Linux son ampliamente accesibles por los programas. En este tipo de ataque, también se aprovecha el buffer overflow, pero para generar un return a una syscall como system, donde solo es necesario agregar en la pila los parámetros de esa función y nada más.
De esta forma, no se está ejecutando código en el stack.

Canario

Es un mecanismo que realiza el compilador para chequear la integridad de la pila antes de restaurar registros importantes, como el IP.
Cuando se invoca la función, entre las variables locales y la dirección de retorno se guarda un valor que se llama Canario, y la idea es que no sea predecible.
De esta manera, cuando la función termina la ejecución, antes de saltar a la dirección de retorno, se chequea que el valor del canario sea el mismo.
Claramente, esto agrega overhead en cuanto a performance.