SISTEMAS OPERATIVOS - FINAL

Bibliografía útil:

La materia está basada completamente en los siguientes dos libros:

- *Silberschatz - Operating System Concepts, 10th edition*
- *Tanenbaum - Modern Operating Systems*

Para armar este resumen no me voy a basar únicamente en lo que dicen las slides de Baader, sino que también voy a leer los libros en gran medida.

1) Introducción

- **¿Qué es un Sistema Operativo?**

Los Sistemas Operativos (SSOO) funcionan como intermediarios entre el *Hardware* y *Software* de una computadora. Con ellos, el software no se tiene que preocupar de detalles de tan bajo nivel del hardware, y a su vez el usuario también puede hacer un buen uso del hardware.
Tiene que funcionar de manera inmediata. Debe darle una sensación de "instantaneidad" al usuario.
Los SSOO son extremadamente grandes y complejos, y deben ser armados "pieza por pieza"; tienen un input/output bien definido y ocupan una porción bien-delineada del sistema.

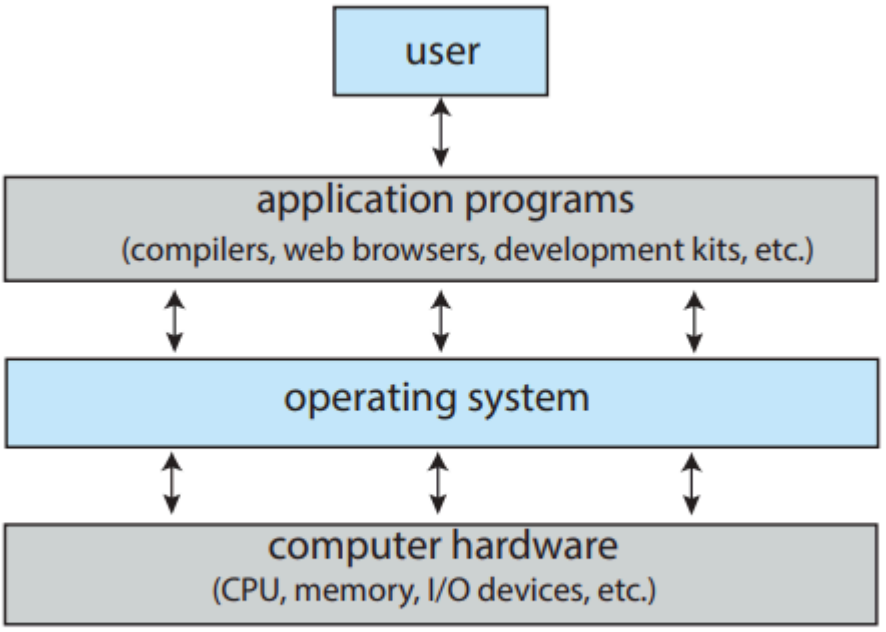


Figure 1.1 Abstract view of the components of a computer system.

Desde el punto de vista de la computadora, el sistema operativo (SO) es el programa más cercano e involucrado con el hardware.
Un sistema informático tiene muchos recursos que pueden necesitarse para resolver algún problema: tiempo de CPU, espacio en memoria, almacenamiento, dispositivos E/S, etc. El SO se encarga de administrar todos estos recursos. Actúa en nivel 0 dentro del sistema.

2) Procesos y API del SO (y Syscalls)

Un programa es una secuencia de pasos escrita en algún lenguaje. Cuando ese programa se pone a ejecutar, lo que obtenemos es un ***proceso***. A cada proceso se le asigna un identificador numérico ÚNICO: el *process-ID* (PID).
Un proceso puede:

- **Terminar**: Con la función *exit()*, el proceso le dice al SO que ya puede liberar sus recursos. En ese exit, indica su estatus de terminación: 0 si es bueno, 1 si hubo algún fallo. El código status se lo envía a su padre.
- **Lanzar un proceso hijo**: Un proceso puede usar la función *fork()*, y lanzar un proceso igual al actual. Luego de un fork, el proceso padre e hijo se ejecutan en paralelo.
- **Hacer una Syscall**: Para hacer una *Syscall*, se debe llamar al Kernel. Esto implica un cambio en el nivel de privilegio.
- **E/S sobre dispositivos**.

Decimos entonces que un sistema consiste en una colección de procesos donde algunos ejecutan código de usuario, y otros ejecutan código de SO. Potencialmente, estos procesos pueden ser ejecutados de forma concurrente.

En general, cuando un proceso crea un proceso hijo, este hijo necesita de ciertos recursos (tiempo de CPU, memoria, archivos, dispositivos de E/S) para completar su tarea. Un proceso hijo puede obtener sus recursos directamente del SO, o puede estar restringido a trabajar con ciertos recursos del proceso padre. (Aquí, el padre deberá de hacer una partición de sus recursos y dividirla entre sus hijos, o compartir algunos (memoria o archivos) entre varios de sus hijos). Restringir los conjuntos del hijo a un subconjunto de los recursos del padre evita que un proceso sobrecargue el sistema al crear muchos procesos hijos nuevos.

NOTA: *La creación de procesos consume tiempo y usa recursos de forma intensiva. Si el proceso nuevo que creamos va a realizar la misma tarea que ya estamos realizando, podemos optar por utilizar* ***threads*** *para evitar todo ese sobrecosto. Generalmente, es más eficiente usar un proceso que contiene múltiples threads.*
Los threads son más eficientes en este sentido ya que, todos los threads que pertenecen a un mismo proceso, comparten los recursos del mismo.

¿Qué es una *Syscall*?

Las syscalls proveen una interfaz a los servicios brindados por el SO: la API (*Application Programming Interface*) del SO. La mayoría de los programas hacen un uso intensivo de ellas. Por lo general, para pasar a modo kernel, se suelen utilizar interrupciones.

Las syscalls suelen usarse a través de *wrapper functions* en C. Esto se debe a que necesitamos que nuestro código sea **portable**, y C nos lo permite. Aún así, hay ciertas tareas de muy bajo nivel (por ejemplo, alguna tarea donde se debe acceder al hardware directamente) que DEBEN ser escritas en lenguaje ensamblador (*Assembly*).

El uso de una syscall implica el uso de otras más. Por ejemplo, si usamos el comando *cp* (copy) de Linux para copiar un archivo en un directorio, el uso de esa syscall genera otros:

- Ver si existe o no el archivo fuente, y si tiene los permisos para leerlos (syscalls)
- Buscar el directorio destino y crear el archivo (syscalls). Si un archivo con ese nombre ya existía, se debe usar una syscall para indicar un error
- Leer en loop el archivo fuente (más syscalls)
- Escribir en loop en el destino (más syscalls)
- Cerrar los 2 archivos, el copiado y el original (2 syscalls)
- Escribir por consola un mensaje de *success*
- Terminar la ejecución (una syscall)

En resumen, hasta la instrucción más pava hace un uso intensivo del SO, generando una cadena gigante de Syscalls.

Importancia de las API

Como ya mencionamos antes, una API nos brinda una serie de funciones bien definidas que se encuentran disponible para un desarrollador de aplicaciones, incluyendo los correspondientes parámetros para cada función, y su valor de retorno.
Un programador accede a una API a través de una biblioteca de código provista por el SO. El caso que más conocemos es la **libc**, usada para programas escritos en lenguaje C. Un desarrollador que está diseñando un programa usando una API, tiene garantizado que su programa va a compilar y correr en cualquier sistema que soporte esa misma API.
Ya vimos antes también que trabajar con *syscalls* puede resultar difícil si no lo hacemos con una API.

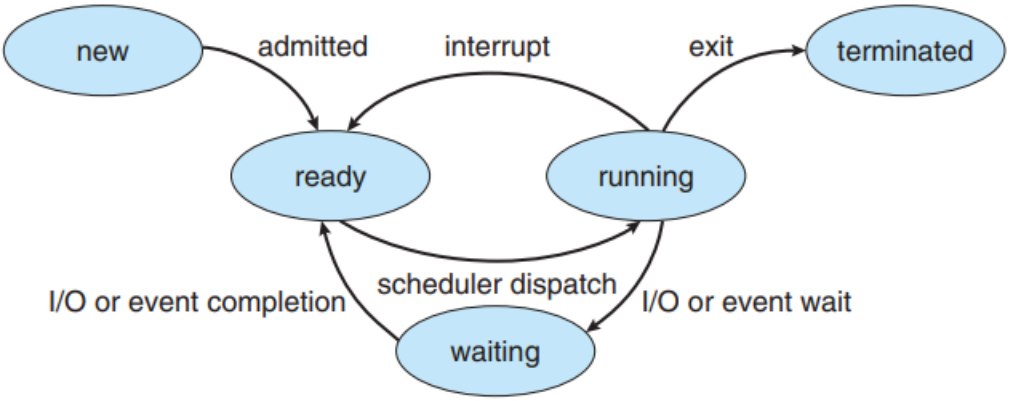


Figure 3.2 Diagram of process state.

Los estados de un proceso:

- **New:** Se está creando el proceso
- **Running:** Las instrucciones del proceso se están ejecutando
- **Waiting:** El proceso está esperando a que ocurra un evento.
- **Ready:** El proceso está listo para ser asignado al procesador. Recordemos que solo un proceso a la vez se encuentra en la CPU. El *scheduler* se encarga de asignar los tiempos que trabaja cada proceso.
- **Terminated:** El proceso terminó con su ejecución.

En un SO, cada proceso está representado por un *Process Control Block* (PCB). Este bloque incluye piezas de información como:

- *Process ID*
- Estado de los registros del CPU
- Estado del proceso
- *Program Counter*
- Prioridad del proceso
- Dispositivos de E/S asignados al proceso, lista de archivos abiertos, etc

En resumen, el PCB es un repositorio para toda la información que se necesita para iniciar, reiniciar o continuar un proceso.

3) Comunicación Inter-Procesos (IPC)

Los procesos en ejecución concurrente en un SO pueden definirse como un proceso *independiente* o como un proceso *cooperativo*. Decimos que un proceso es **independiente** si no comparte ningún tipo de data con los otros procesos en ejecución. Un proceso es **cooperativo** si puede afectar o ser afectado por otros procesos que hay en el SO.

Hay distintas razones para proveer un entorno que permita procesos cooperativos. Algunas de ellas son:

- **Compartir información:** Muchas aplicaciones pueden estar interesadas en la misma pieza de información. Debemos dar un entorno que permita un acceso concurrente a esa información.
- **Acelerar el computador:** Para que un proceso corra más rápido, se lo puede dividir en tareas más pequeñas que corran al mismo tiempo, paralelamente.
- **Modularidad:** Dividir las funciones del sistema en procesos (o *threads*) separados.

Es evidente que los procesos cooperativos necesitan de un mecanismo IPC que les permita enviar y recibir datos entre ellos.
Existen distintas formas de *Interprocess Communication* (IPC):

- Memoria compartida
- Recursos compartidos (archivos, base de datos, etc.)
- Pasaje de mensajes. Puede ser entre procesos o de equipos conectados en red.

Los fundamentales son **memoria compartida** y **pasaje de mensajes**. Vamos a centrarnos en ellos 2 y listar algunos pros y contras:
Muchos SSOO implementan ambos. *Pasaje de mensajes* es útil para intercambiar datos de pequeña medida y es más fácil de implementar en un sistema distribuido. Aún así, *memoria compartida* puede ser más rápido, ya que el pasaje de mensajes requiere del uso de syscalls, lo que deviene en un gran tiempo de intervención de tareas de kernel, como ya vimos antes. En *memoria compartida* alcanza con establecer las regiones de memoria compartida, valga la redundancia. Una vez instauradas, no es necesaria ninguna asistencia del kernel, puesto que los accesos a esas regiones de memoria son tratados como tal.

IPC en un sistema de Pasaje de Mensajes

El pasaje de mensajes provee un mecanismo que permite que los procesos se comuniquen y sincronicen sus acciones sin compartir el mismo espacio de direcciones. Como ya mencionamos, es útil para sistemas distribuidos, donde los procesos comunicados pueden residir en distintas computadoras que se encuentran conectadas por red.
Tenemos 2 tipos de pasaje de mensajes:

- ***Comunicación Directa:*** Cada proceso que quiera comunicarse, debe explicitar con quién quiere hacerlo. Debe nombrar quién es el receptor o el emisor a la hora de enviar/recibir mensajes.
- ***Comunicación Indirecta:*** Hay una serie de "buzones" (*mailbox)* que se enumeran de forma única. Cada proceso puede escribir y leer de la casilla que quiera, solo debe aclarar qué casilla quiere leer/escribir.
  En estos casos, un link puede estar establecido entre más de dos procesos.
  Una *mailbox* puede pertenecer al SO o a un proceso. Si le pertenece a un proceso, podemos distinguir entre el dueño de la *mailbox* (recibe mensajes en esa mailbox) y entre el usuario (manda mensajes a esa mailbox).
  Si el proceso dueño de una *mailbox* termina, esa caja desaparece. Si un proceso usuario intenta enviarle un mensaje a esa *mailbox*, se le debe notificar que ya no sigue existiendo.

FILE DESCRIPTORS:

Los *file descriptors* representan instancias de archivos abiertos. Son índices de una tabla que indica los archivos abiertos por un proceso. Cada entrada de la tabla apunta a un archivo.
La mayoría de los procesos esperan tener 3 *file descriptors* abiertos correspondientes a:

- 0 = *Standard Input*
- 1 = *Standard Output*
- 2 = *Standard Error*

Los *file descriptors* se heredan de un proceso padre a un proceso hijo. También se mantienen en la llamada a *execve*.
Si tenemos un *file descriptor* (fd), podemos utilizar las operaciones ordinarias de **read()** y **write()** para leer o escribir el archivo respectivamente.

PIPES:

Un pipe es un conducto que permite que dos procesos se comuniquen. Fueron uno de los primeros mecanismos IPC en los primeros sistemas UNIX.
Son un "pseudo" archivo que "esconde" un IPC. Se representa como un archivo **temporal** y **anónimo** que se aloja en memoria. Como los pipes son archivos, sus extremos (de escritura y lectura) son agregados a la tabla de *FILE DESCRIPTORS*. Es por esto mismo que se pueden utilizar las operaciones ordinarias para leer o escribir un extremo del pipe, según corresponda.

Al momento de implementar un pipe, 4 problemas deben ser considerados:

- *¿El pipe admite comunicación bidireccional o unidireccional?*
- *Si es bidireccional, ¿los datos pueden viajar en ambas direcciones al mismo tiempo?*
- *¿Debe existir una relación entre los procesos comunicados? (Padre-hijo, etc)*
- *¿Puede usarse el pipe en red o debe ser usado en una misma computadora?*

<u>Ordinary Pipe</u>: Permite una comunicación unidireccional entre procesos. Un proceso escribe sobre un borde (el borde de escritura), y el otro lee desde el borde contrario (el borde de lectura). Si se quiere lograr una comunicación bidireccional entre estos 2 procesos, otro pipe (ordinario) debe ser creado.

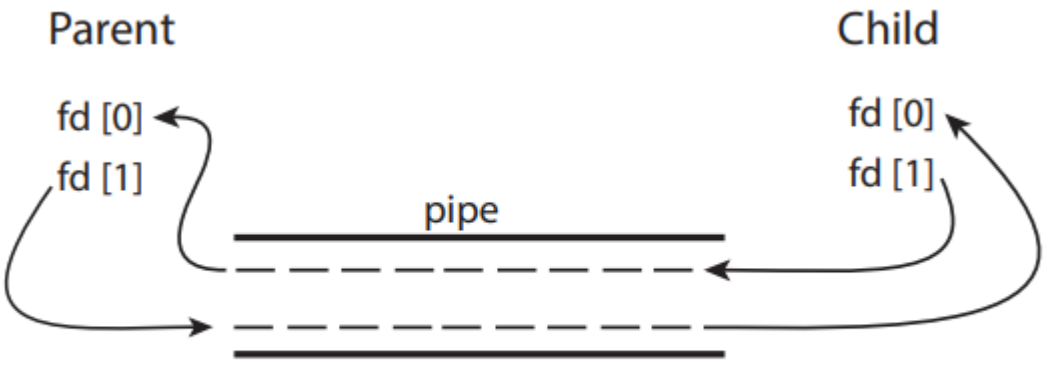


Figure 3.20 File descriptors for an ordinary pipe.

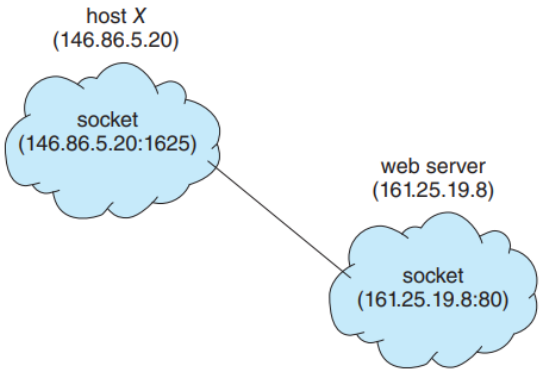
En una relación de padre-hijo, el hijo hereda los pipes del padre. Esto se debe a que un pipe es un tipo especial de archivo. Al heredarlo, puede usar ese pipe para enviarle mensajes al padre. Es importante notar que los pipes ordinarios requieren una relación de padre-hijo para ser utilizados. Esto significa que solo pueden ser usados entre procesos de una misma máquina.

Named Pipes: Acá la comunicación puede ser bidireccional, y no es requerida una relación de padre-hijo para ser utilizado. Una vez que un *named pipe* es inicializado, distintos procesos pueden usarlo. En un escenario típico, un *named pipe* tiene muchos escritores.
A diferencia de los pipes ordinarios, estos pipes siguen existiendo luego de que los procesos comunicados dejan de existir.

Ahora, hablemos sobre otro tema. Veamos como funciona la comunicación en sistemas ***cliente-servidor***.
Antes vimos dos métodos para que los procesos puedan comunicarse entre sí (*memoria compartida* y *pasaje de mensajes*). Ahora, vamos a ver otras dos estrategias que funcionan para estos sistemas: *sockets* y *remote procedure calls (RPCs)*.

Sockets:

Los sockets se definen como el punto final para la comunicación. Un par de procesos que se comunican a través de una red requieren de dos sockets–uno para cada proceso. Un socket se identifica como una dirección de IP concatenado con un número de "puerto". Veamos el siguiente ejemplo:



El número 146.86.5.20 se corresponde a la dirección de IP de un cliente con un host *X*.
Luego, el número 161.25.19.8 se corresponde a la dirección de un servidor web con la que se quiere conectar el cliente.
El servidor web está "escuchando" el puerto 80.

El número de puerto que un host le asigna siempre a un cliente es mayor a 1024 (los números ≤ 1024 son usados por servidores. Esos puertos son conocidos como ***well-known***). Es por eso que se le asigna el puerto 1625.
Luego, los paquetes de datos que viajan entre los *hosts* son entregados al proceso correcto ya que se basan en el número (de puerto) de destino.
Todas las conexiones deben ser únicas, por lo tanto, todas las conexiones consisten en un único par de sockets.

Remote Procedure Calls (RPC)

El paradigma RPC consiste en el pasaje de mensajes, pero fue diseñado de forma tal que funcione entre sistemas con conexiones de red.
Los mensajes en la comunicación RPC deben estar bien estructurados: cada mensaje es dirigido a un *daemon* (proceso que está corriendo en segundo plano) RPC que se encuentra conectado a un **port** en el sistema remoto. Además, cada mensaje indica qué función ejecutar bajo qué parámetros. Por último, la función es ejecutada y el output de la misma es enviada al emisor en forma de respuesta.
La semántica de RPC permite que un cliente invoque un proceso en un host remoto, de la misma forma que invocaría un proceso localmente.
Puede suceder que el cliente y las máquinas de los servidores representen los datos de forma diferente (por ejemplo, uno usa *big-endian (se almacenan bytes más significativos primero)*, mientras que el otro usa *little-endian (se almacenan bytes menos significativos primero)*).
Para evitar estos problemas, se usa una representación independiente de los datos. Un método conocido es **external data representation** (**XDR**). Por un lado, el cliente convierte sus datos en XDR antes de enviarlos, mientras que el servidor "descompone" los datos XDR primero.

4) Scheduling

La política de *scheduling* es de las huellas de identidad más importantes de un SO: Algunos sistemas incluyen más de una. Para optimizar un SO, una mayor parte del esfuerzo se va en organizar una buena política de scheduling.

Veamos los distintos objetivos de las políticas de scheduling:

- *Fairness*: Cada proceso recibe una dosis "justa" de CPU.
- *Eficiencia:* CPU ocupada todo el tiempo
- *Carga del Sistema:* Minimizar la cantidad de procesos que están esperando el CPU
- *Tiempo de Respuesta:* Minimizar el tiempo de respuesta percibido por los usuarios interactivos
- *Latency:* Minimizar tiempo requerido para que un proceso arroje resultados
- *Turnaround:* Minimizar el tiempo total que le toma a un proceso ejecutar completamente (tiempo en *ready* + tiempo en CPU + tiempo haciendo I/O)
- *Throughput:* Maximizar cant. procesos terminados por unidad de tiempo
- *Liberación de Recursos:* Que terminen antes los procesos que usan más recursos

Muchas de estas metas son contradictorias entre sí, es por eso que cada política de scheduling que se defina, intentará maximizar una función objetivo sin impactar negativamente sobre el resto.

Cada vez que el CPU se "vuelve" *idle*, entra en juego el scheduler para definir qué proceso será el próximo en entrar a la CPU. El scheduler selecciona un proceso entre todos los procesos que haya en memoria en estado *ready*, y le asigna la CPU a ese proceso.
Todos los procesos que se encuentran esperando en la *ready queue* están "representados" por sus *process control blocks* (PCBs).

El scheduling puede ser ***preemptive*** (con desalojo) o ***nonpreemptive*** (cooperativo/sin desalojo). En el caso de preemptive, el scheduler se vale de las interrupciones de clock para determinar si el proceso que está ocupando el CPU debe seguir ejecutando o no. En el caso de un scheduler nonpreemptive, espera a que el proceso finalice, o a que se ponga en estado *waiting* para asignarle la CPU a otra tarea.
Un kernel preemptive requiere el uso de mecanismos como *lock* y *mutex* para evitar race conditions cuando se acceden a estructuras de datos compartidos de kernel.
Asimismo, como las interrupciones pueden ocurrir en cualquier momento, las secciones de código afectadas por la interrupción deben admitir un uso simultáneo.

Dispatcher: Es un componente involucrado en las funciones de CPU-Scheduling. Es el encargado de darle el control del CPU al proceso seleccionado por el scheduler. Realiza los siguientes pasos:

- *Context Switch* de un proceso a otro
- Cambia a modo usuario
- Salta a la ubicación adecuada para resumir el programa de turno

El *despachador* debe ser lo más rápido posible, ya que es llamado en cada cambio de contexto que se realiza en la máquina. El tiempo que le toma a un dispatcher cambiar de un proceso a otro, y hacerlo arrancar se conoce como **dispatcher latency**.

Algoritmos de Scheduling:

El problema con el que lidia el scheduler de una CPU es el de decidir qué proceso que se encuentra en la cola *ready* será el que tome control de la CPU.

NOTA: De ahora en adelante, cada vez que digamos que un proceso "hace" un pedido de I/O, nos referimos a que está esperando a que un dispositivo externo finalice su trabajo.

Veamos algunos algoritmos.

- First Come, First Served (FCFS)

Es el más simple de todos los algoritmos. Como su nombre lo indica, el primer proceso en llegar a la cola de procesos *listos*, es el primero en tomar la CPU. Luego, esa cola se irá

llenando, y se irá "vaciando" respetando el orden de llegada de cada proceso. Básicamente, la política de scheduling de FCFS está implementada con una *FIFO queue*.

Desventaja de este algoritmo: por lo general, el **waiting time** (el tiempo que espera cada proceso en la *ready queue*) es muy grande. Esto se debe a que un proceso libera completamente la CPU cuando finalizó su ejecución o cuando haya creado un pedido de I/O.

- ## Shortest-Job First (SJF)

En este algoritmo, cada proceso es asociado con la longitud que tiene su ráfaga de CPU, es decir, su tiempo total de ejecución. Los procesos con una ráfaga más corta, son los primeros en tomar posesión de la CPU. Este algoritmo está orientado a maximizar el **throughput**.

Aunque el algoritmo SJF es óptimo, no puede ser implementado en el nivel de CPU-Scheduling. Esto se debe a que NO podemos saber de antemano cuál es la ráfaga de CPU de un proceso. Pueden predecirse los tiempos futuros a partir de los tiempos de ejecución de los procesos pasados, pero esta alternativa nos puede salir mal si los procesos tienen un comportamiento irregular.

- ## Round-Robin

El algoritmo de scheduling Round-Robin (RR) es similar a FCFS, pero se le agrega la propiedad de *preemptive* al scheduler, permitiendo así la posibilidad de cambiar entre procesos.

A cada proceso se le asigna una misma cantidad de tiempo de CPU. A estos tiempos les llamamos *quantum*. Un quantum no puede ser muy largo puesto que, para SO interactivos, podría parecer que el sistema no responde. Tampoco un quantum puede ser muy corto, ya que el tiempo de *scheduling + context switch* nos consumirían una gran parte del mismo. La performance de un algoritmo RR depende fuertemente del tiempo asignado al quantum. Por lo general, un quantum dura entre 10 y 100 milisegundos (un *context switch* demora menos de 10 microsegundos).
El scheduler de la CPU va iterando sobre la cola de procesos *ready*, y le asigna a cada proceso el equivalente a 1 quantum. Para implementar RR, se trata a la *ready queue* como una cola FIFO. El scheduler toma el primer proceso de la cola, le asigna 1 quantum de tiempo, y lo despacha.

- ## Multilevel Queue

Es una variación que surge de **Priority Scheduling** y **RR**. En esos 2 modelos, tenemos una cola de procesos en estado *ready*. Si se quiere correr primero el proceso con mayor prioridad, esto implica una búsqueda en la *queue* de O(n) procesos. Es por eso que se crearon las colas de prioridad.
En Multilevel Queue, tenemos distintas colas con distintas prioridades, donde cada proceso es colocado en la cola que le corresponda (según su prioridad). Es un *mix* de RR y Priority Scheduling. Cuando se selecciona una cola llena de procesos (la que tenga procesos

esperando, y sea la de más alta prioridad en el momento), esta es ejecutada con RR, hasta despachar a todos los procesos que hayan.

- **Multilevel Feedback Queue**

Es como la anterior, pero ahora cada proceso tiene *aging*. En criollo, si hace mucho que se atienden procesos de más alta prioridad, los procesos más bajos comienzan a aumentar su prioridad para ser movidos de *queue*, y ser atendidos lo antes posible. Este modelo soluciona el problema de *starvation* que trae la política de *Multilevel Queue*.

Scheduling en Sistemas SMP (Symmetric Multiprocessing):

A continuación vamos a hablar sobre un sistema que tiene **más de un procesador** (es decir, más de un chip). El approach más común en estos sistemas es que cada procesador es auto-programado (*self-scheduling* // No sé como traducirlo). Cada proceso tiene al scheduler revisando su *ready queue*, y eligiendo qué *thread* es el siguiente en correr. Existen 2 formas de organizar los threads que van a ser seleccionados para ejecutar:

1. *Todos los threads en una misma ready queue*
2. *Cada procesador tiene su PROPIA ready queue*

El primer método genera Race Conditions si no somos cuidadosos a la hora de implementarlo. El segundo es el método más usado para sistemas SMP. Además, que cada procesador tenga su propia cola de procesos en espera, nos lleva a un mejor manejo de memoria cache.

Si nuestro sistema utiliza el segundo enfoque, el de *queues* privadas para cada procesador, puede suceder que un procesador agote todas sus tareas y se quede en estado *idle*. Nosotros siempre queremos evitar esta situación, así que veamos una manera de solucionarlo:
Load Balancing es un mecanismo que se encarga de mantener la carga de trabajo de cada procesador de manera equivalente. Existen dos enfoques diferentes para implementar este mecanismo:

- ***Push Migration:*** Una tarea específica chequea periódicamente la carga de cada procesador. Si encuentra un desbalance, comienza a mover los *threads* de los procesadores más cargados a los menos.
- ***Pull Migration:*** Un procesador en estado *idle* toma una tarea que está en espera en el otro procesador.

Ahora, veamos cómo funciona el scheduler en procesadores ***multicore***.
Se sabe que, cuando un procesador accede a memoria, gasta mucho tiempo esperando a que los datos estén “disponibles”. Estas situaciones ocurren porque los procesadores trabajan a una velocidad mucho más rápida que la memoria (aunque se hayan creado las memorias *cache*, tenemos el mismo problema ante un *cache miss*).
Para solucionar estos problemas, se diseñaron hardwares que le asignan dos (o más) *threads* a cada núcleo del procesador. De esta manera, si un *hardware-thread* se clavó esperando la memoria, el núcleo puede cambiar al otro *thread*. El *switch* entre *hardware-threads* es caro,

ya que se debe vaciar todo el pipeline de instrucciones antes de que el nuevo *thread* comience su ejecución.
En estos sistemas tenemos dos niveles de scheduling:

1) Scheduling del SO, que decide qué *software-thread* es asignado en cada *hardware-thread*. Esto lo hace con algún algoritmo de scheduling que vimos antes.
2) Scheduling de cada núcleo, en el que se define qué *hardware-thread* corre en cada núcleo. (Existen distintos enfoques para este scheduling. Uno es Round-Robin entre los *hardware-threads*. Otro es la aparición de un evento con un alto nivel de urgencia. Aquí se cambia de *hardware-thread* si sucede algún evento importante que implique el *switch* entre *threads*.)

Real-Time Scheduling:

Podemos distinguir un sistema entre **Soft real-time systems** y entre **Hard real-time systems**. Veamos la diferencia entre ambos:

- **Soft real-time:** En este tipo de sistemas no se provee ningún tipo de garantía de cuándo una tarea será seleccionada para ejecutarse.
- **Hard real-time:** En este sistema hay unos requisitos más estrictos. Las tareas tienen un tiempo definido, y deben ser atendidas antes de que se termine su *deadline*. No atender estos procesos antes de sus *deadlines* equivale a no atenderlos por completo.

Decimos que un sistema es de tiempo real cuando sus tareas tienen fechas de finalización (deadlines) estrictas. Los sistemas de tiempo real suelen usarse en entornos críticos: "Si un *deadline* no se cumple, algo malo pasa".
Una posible política de scheduling en estos sistemas es *Earliest-Deadline-First* (EDF).

Scheduling en Linux:

En Linux, los procesos se ejecutan por orden de prioridad. Fusionado con un RR y Multilevel Queue, Linux tiene un árbol binario llamado "***red-black tree***". En este árbol, los valores con menor **vruntime** (tiempo de procesamiento) son colocados en los nodos que están más a la izquierda. Para el sentido que tiene el *Completely Fair Scheduler* (CFS), estos nodos (procesos) son los de más alta prioridad.
Tenemos garantizado que los **red-black tree** están balanceados (como los AVLs), por lo que buscar una tarea dentro del mismo toma un tiempo de O(log N). De igual forma, por cuestiones de eficiencia, Linux se guarda en la cache el valor del nodo que está más a la izquierda. Así, el scheduler ya sabe que proceso es el próximo a ejecutar tan solo leyendo la memoria cache.

Scheduling en Windows:

En Windows se usa una política de prioridad basada en un scheduler *preemptive*. El scheduler de Windows asegura que el *thread* de máxima prioridad siempre correrá. El *thread* seleccionado para correr lo hará hasta que: aparezca un *thread* de mayor prioridad, se acabe su quantum, termine, o hasta que llame a una syscall bloqueante. Si una tarea de prioridad

más alta se “pone” en estado *ready* mientras hay otro proceso de menor nivel en ejecución, el proceso que está siendo ejecutado termina siendo reemplazado.
La porción del kernel de Windows que se encarga de manejar al scheduler se llama **dispatcher**. El dispatcher utiliza un esquema de 32 niveles de prioridad para determinar cuál es la próxima tarea a ser ejecutada. Los procesos se dividen en 2 clases:

- **Variable class:** Threads con prioridades del 1 al 15
- **Real-time class:** Threads con prioridades del 16 al 31.

Está de más decir que, cuando un proceso está más cerca del 31, debe ser atendido inmediatamente, mientras que si un proceso tiene una prioridad cercana al 1, es porque no es “tan importante”.

LO QUE VIENE A CONTINUACIÓN LO SAQUÉ DEL *TANENBAUM* Y ME PARECIÓ CURIOSO:

Cuando el kernel manjea *threads*, el scheduling se realiza por *thread*, sin darle importancia al proceso al que esos *threads* estén asociados.
Para tener un scheduling *preemptive*, requerimos que ocurra una interrupción de reloj al final de cada intervalo de tiempo (quantum) para darle de vuelta el control de la CPU al scheduler. Si no hay un clock disponible, un scheduling *nonpreemptive* es la única opción.

Mencionamos los procesos *interactivos* y los *real-time*. Veamos los procesos *batch*:
Los sistemas Batch son de uso cotidiano en el mundo para hacer depósitos en el banco, inventarios, cuentas pagables, etc. En los sistemas batch, no hay ningún usuario esperando impacientemente a que la terminal responda de forma veloz, es por eso que un scheduler *nonpreemptive* es una opción, o un scheduler *preemptive* que le dedique mucho tiempo a cada tarea.

5) Sincronización

Vimos anteriormente que un proceso *cooperativo* era aquel que podía afectar o ser afectado por otros procesos cooperativos en ejecución. Estos procesos pueden compartir un espacio de direcciones lógicas (datos y código), o solo compartir datos entre sí. El acceso concurrente a datos puede resultar en una inconsistencia de datos.

En un mundo donde cada vez se tiende más a la programación paralela, los dos problemas fundamentales que se nos presentan son la *contención* y la *concurrencia*. Dos eventos son concurrentes si no podemos decir a simple vista en qué orden se ejecutan.

Un problema que nos trae la concurrencia:

- **Race Conditions:** Defecto de un programa concurrente por el cual su correctitud depende del orden de ejecución de ciertos eventos. En otras palabras, el resultado de un programa varía sustancialmente dependiendo de en qué orden se ejecuten las cosas

Una forma de solucionar las *race conditions* es con **secciones críticas**, secciones de código donde solo se puede ingresar de a un proceso a la vez. En una sección crítica:

1) Solo hay un proceso a la vez
2) Todo proceso que esté esperando para entrar a CRIT va a entrar
3) Ningún proceso fuera de CRIT puede bloquear a otro

Para implementar secciones críticas, es necesaria la ayuda del hardware. Los *locks* (hacemos comparaciones: lock = 0 si PUEDO entrar, lock = 1 si NO PUEDO entrar) no los podemos usar sin ningún tipo de cuidado. Existe la posibilidad de que un proceso vea el 0 y se le acabe el quantum, por lo que otro proceso ingresará luego a CRIT, pero cuando volvamos al primer proceso, este había visto un 0, por lo que entrará también a CRIT.

- **Busy Waiting:** Decimos que un proceso hace *busy waiting* cuando se encuentra dentro de un ciclo *while* sin hacer nada, osea, esperando a que ocurra un evento (espera a que la guarda del *while* sea false para salir del mismo).

Cuando un proceso hace *busy waiting* consume MUCHA CPU, ya que el proceso sigue activo sin hacer nada. La espera activa es una forma muy agresiva de obtener un recurso. Llamamos **spinlock** a los locks que no "duermen" a los *threads* mientras esperan a que el lock esté disponible. Esto es por lo que explicamos antes, el proceso gira (spins) mientras espera al lock.

Muchos sistemas modernos proveen instrucciones atómicas con ayuda del hardware. Esto significa, que se puede ejecutar toda la instrucción de forma ininterrumpida. Dos de estas instrucciones atómicas son:

- *test_and_set() – TAS*
- *compare_and_swap() – CAS*

En **TAS**, se toma el puntero pasado por parámetro, se guarda su valor en una variable local y luego el puntero es seteado a *true*. Por último, se retorna la variable local que representa el valor "anterior" que tenía el puntero.
En **CAS**, se trabaja sobre 3 operandos: un puntero con cierto valor, un valor esperado y un nuevo valor. Si sucede que *(valor == valor_esperado)*, entonces se cambia el valor del puntero al nuevo valor. Por último, la instrucción retorna el valor original del puntero en el momento que se invocó la instrucción. CAS puede ser usado para ver el valor de un *lock*: ***compare_and_swap(&lock, 0, 1)*** – aquí el valor esperado es 0, por lo que el *lock* es seteado a 1 si su valor es el esperado.

Es importante destacar que, si dos instrucciones atómicas son invocadas al mismo tiempo, entonces se ejecutarán secuencialmente en algún orden arbitrario.

Dijkstra inventó los **semáforos**. Un semáforo es una variable entera que puede inicializarse con cualquier valor, y que se puede manipular con las siguientes dos operaciones atómicas:

- **wait()**
- **signal()**

También existen los *mutex* (*mutual exclusion*), que es un semáforo binario. Los *mutex* se implementan con la operación atómica *Compare and Swap* (CAS).
Estos softwares de más alto nivel fueron diseñados para que los programadores no se compliquen la vida usando TAS y CAS al desarrollar una aplicación.

En un semáforo, si un proceso hace un wait(), esperando a que la sección crítica esté habilitada para pasar, ese proceso se duerme. De esta forma, no gasta CPU mientras espera. En particular, la función *wait()* agrega a un proceso P a una lista de procesos en espera y lo "duerme" con la operación *sleep()*. Luego, la función *signal()* se encarga de despertar a alguno de los procesos que se encuentran en esa lista de procesos en espera. Lo hace utilizando la operación *wakeup(P)*.
sleep() y *wakeup()* son syscalls básicas que provee el SO.
La "lista de espera" puede implementarse con los *Process Control Block* (PCB) de cada proceso. Cada semáforo lleva entonces un valor entero y un puntero a una lista de PCBs.

NUNCA nos podemos olvidar de un **signal()** si hay un wait(). Podemos llegar a generar un *deadlock* con los semáforos. Puede que nuestro código sea inconsistente, y que algunos eventos estén esperando un signal(), cuando ningún otro proceso vaya a hacerlo.

Condiciones de Coffman:

Coffman nos dice que, si las siguientes 4 condiciones se cumplen en simultáneo, entonces estamos ante la presencia de un *deadlock*.

- **Exclusión mutua:** Hay un recurso asignado a un *thread*, y ese recurso no puede ser compartido con otro *thread* hasta que sea "liberado" por el primero.
- **Hold and wait:** Hay un *thread* que tiene un recurso y está esperando a adquirir otro recurso que se encuentra en "manos" de otro *thread*.
- **No preemption:** No hay mecanismo compulsivo para quitarle los recursos a un *thread*.
- **Espera circular:** Tiene que haber un ciclo de N >= 2 tal que T_i espera un recurso que tiene T_i+1.

Decimos que un conjunto de procesos está *deadlockeado* (? cuando cada proceso del conjunto está esperando a un evento que solo puede ser generado por otro proceso del conjunto. (El evento en cuestión es la ejecución de un signal())

Monitores:

Por más de que los semáforos estén bien implementados, existe la posibilidad de que generen errores si no les damos un buen uso. Por ejemplo, para cada sección crítica, debe existir un wait() para ingresar a ella, y un signal() al salir de la misma. Si no seguimos ese protocolo, lo más probable es que tengamos fallas de concurrencia en nuestro código.
Una estrategia para evitar estos errores es utilizar estructuras de sincronización de alto nivel. Una de ellas, es el "tipo" *monitor*, un tipo abstracto de datos.

El TAD *monitor* tiene un conjunto de variables compartidas, un código de inicialización, y un conjunto de N funciones/operaciones. Como cualquier TAD, las funciones de *monitor* pueden operar sobre esa data "global" definida dentro de su estructura. En criollo, una función definida adentro de *monitor* solo puede acceder a las variables declaradas en el struct y a las que recibe por parámetro. Asimismo, los datos que definimos adentro de *monitor* solo pueden ser accedidos a través de sus operaciones ya definidas.
Entonces, *monitor* es como un TAD cualquiera, con la diferencia de que:

- La estructura de *monitor* se asegura de que solo un proceso esté activo adentro de *monitor*.
- Esperar a que cambie el estado de una variable de *monitor* no bloquea el struct.

Entonces, la ventaja que nos traen los *monitores* es que son más fáciles de implementar que los semáforos, y la exclusión mutua en ellos es automática. Además, no hay errores de *timing* como con semáforos. Aún así, los *monitores* no permiten que más de un *thread* acceda a la sección crítica (mientras que los semáforos sí lo permiten).

Variables de Condición:

La estructura de *monitor* no es muy poderosa para diseñar esquemas de sincronización muy grandes, entonces es necesario definir otros mecanismos de sincronización para poder hacerlo.
Al momento de codear un esquema sincronizado, se pueden definir una o más variables del tipo *condition*:

```
condition x, y
```

Las únicas operaciones que pueden usarse con estos tipos de datos son wait() y signal(), pero con una ligera diferencia.

- **x.wait():** El proceso que invoque esta operación será suspendido hasta que otro proceso invoque a x.signal().
- **x.signal():** Enciende al proceso que estaba dormido desde el punto exacto en el que se había quedado. Si no había ningún proceso esperando, entonces signal() no tiene ningún tipo de efecto. Esto significa que el estado de x sigue siendo el mismo, es como si nunca se hubiera ejecutado esa operación.

Notar que en semáforos, la función signal() SIEMPRE afecta al semáforo, aunque no haya ningún proceso esperando.

Si tenemos dos procesos P y Q ejecutándose, y una variable de condición, existe un compromiso entre estas 2 tareas. Supongamos que Q está esperando por una condición o a

que P abandone el monitor. Si P llama a la función signal(), abandona el monitor inmediatamente puesto que Q comienza a ejecutar (hubo un cambio de condición).

Priority Inversion:

Un desafío de scheduling se nos presenta cuando un proceso de mayor prioridad quiere leer o escribir datos de kernel que están siendo ocupados actualmente por un proceso de menor prioridad. Obviamente, los datos de kernel son bloqueantes, usan un *lock*, por lo que el proceso de mayor prioridad debe esperar tiene que esperar a que el más chico termine de usar el recurso. La situación se complica más si el proceso más "chico" es reemplazado por el scheduler por OTRA tarea de mayor prioridad que el más "chico", pero de menor prioridad que el proceso más "grande". Indirectamente, un proceso de menor prioridad alarga el tiempo de espera del proceso de mayor prioridad. Este problema de *liveness* se conoce como **inversión de prioridad**.

Un ejemplo sería tener un recurso S, y tres procesos L < M < H. El proceso L tiene el recurso S, y H está esperando por el semáforo S. Cuando L termina de ejecutar, el proceso M cambia su estado a *ready*, y es elegido como el próximo a entrar a S. En este caso, H sigue esperando mientras un proceso de menor prioridad empezó a usar S.

Estos problemas surgen típicamente en sistemas que tienen más de 2 prioridades. Una solución al mismo es implementar un protocolo de herencia de prioridad (***priority-inheritance protocol***). De acuerdo a este protocolo, un proceso que está utilizando un recurso que necesita un proceso de mayor prioridad, hereda la prioridad más alta del proceso que está esperando ese recurso. Luego, su prioridad vuelve a la normalidad. Siguiendo el ejemplo anterior, aquí L hereda la prioridad de H, y H es el próximo que obtiene el recurso S, dado el orden de jerarquía.

6) Administración de memoria

En el libro de Silberschatz, *Operating System Concepts*, hay dos capítulos dedicados a la administración de memoria: uno es **main memory**, y el otro **virtual memory**. Voy a hablar de cada uno por separado.

Main Memory

La memoria es la central de operaciones de una computadora. Consiste en un arreglo extremadamente largo de bytes, donde cada uno tiene su dirección propia. La CPU busca instrucciones en la memoria, de acuerdo al número indicado por el *program counter*.
Un típico ciclo de ejecución de instrucciones es *fetch - decode - (buscar operandos) - execute*

El único tipo de almacenamiento al que puede acceder el CPU es a la memoria principal y a los registros de cada núcleo. Los datos deben estar en memoria para que la CPU pueda operar sobre ellos (si no lo están, son movidos a memoria).

Uno de los temas centrales de esta sección es la velocidad de acceso a direcciones de memoria física, pero también nos interesa garantizar su uso correcto. Debemos proteger al SO de los procesos del usuario, y a los procesos del usuario entre sí. Esta protección debe ser dada por el hardware, puesto que el SO no interviene entre la CPU y sus accesos a memoria (porque la *performance* de la computadora sería bajísima).
En resumen, para lograr proteger el espacio de memoria se usa el hardware de la CPU. Este se encarga de usar registros para comparar las direcciones generadas por el usuario. Si un programa en modo usuario intenta acceder a direcciones de memoria que le pertenecen al SO, se recibe un error (*fatal error*).

Espacio lógico vs espacio físico
Nos referimos a las direcciones generadas por el CPU como *direcciones lógicas*, mientras que las direcciones que son vistas por la unidad de memoria (las que se cargan al registro de direcciones de memoria) las llamamos *direcciones físicas*.

Asignación contigua de memoria
Es uno de los métodos más sencillos. La memoria estará dividida en distintas particiones de distintos tamaños, en los cuales ubicaremos exactamente un proceso en cada una. En este esquema de partición, el SO tiene una tabla que le informa qué partes de la memoria están ocupadas y cuáles no.

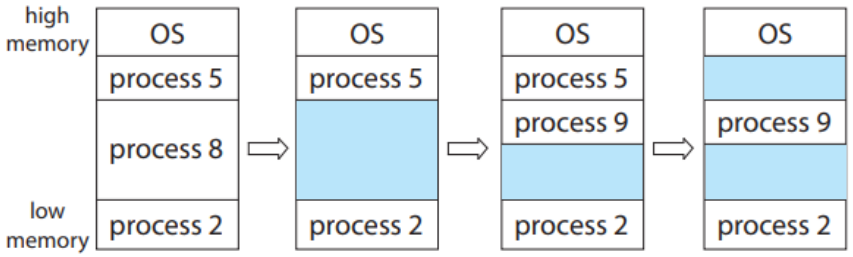


Figure 9.7 Variable partition.

En general, los bloques libres terminan conformando un conjunto de agujeros disjuntos ubicados a lo largo de la memoria.
Cuando un proceso llega y pide memoria, el SO busca un agujero que sea lo suficientemente grande para que entre ahí. Si el agujero que encuentra es muy grande, lo divide en 2; en una parte ubica el proceso, y el otro lo devuelve al set de agujeros disjuntos.

Cuando un proceso termina su ejecución, libera su memoria y devuelve su espacio al set de agujeros.

Existen 3 estrategias a la hora de determinar qué agujero darle a un nuevo proceso:

- **First fit:** Busca el primer agujero que sea lo suficientemente grande para que el proceso entre en ese espacio. Ni bien encontramos uno, termina la búsqueda.
- **Best fit:** Busca el agujero más chico y suficientemente grande para que el proceso entre en ese espacio.
- **Worst fit:** Ubica el proceso en el agujero MÁS GRANDE que esté disponible.

Obviamente, *first fit* y *best fit* son mejores que *worst fit*, pero entre *first fit* y *best fit* ninguno es mejor que el otro. La única diferencia que podemos destacar es que *first fit* es más rápido.

Fragmentación

Todas las estrategias que mencionamos antes sufren de *fragmentación*. En particular, *first fit* y *best fit* sufren de **fragmentación externa** (hay bloques libres en memoria, pero son muy chicos y están todos dispersos), mientras que *worst fit* sufre de **fragmentación interna** (se desperdicia espacio dentro del bloque mismo).

Una posible solución a la fragmentación externa es la **compactación**. Este esquema consiste en reubicar de manera contigua todos los agujeros en memoria para que queden todos pegados. No siempre es posible llevarlo a cabo.
Otra posible solución es permitir que las direcciones lógicas de los procesos no sean contiguas, permitiendo así que un proceso sea ubicado en memoria física en donde sea que haya espacio. Esta estrategia es la que se utiliza en **paginación**.

Paginación

La implementación básica de paginación consiste en "romper" la memoria física en bloques de tamaño fijo llamados *frames* (marcos), y en dividir la memoria lógica en bloques del mismo tamaño, llamados *páginas*. Cuando un proceso va a ser ejecutado, sus páginas son cargadas en cualquier *page frame* que se encuentre disponible.

Cada dirección generada por el CPU está dividida en dos partes: *page number* - *page offset*. El número de página es usado para indexar en una **page table** dedicada a un proceso. La page table contiene las direcciones bases de los *frames* que estamos buscando en la memoria física. Luego, con el *page offset*, nos movemos "dentro" de la página que buscamos.
En criollo, usamos el *page number* para elegir la página que queremos de la page table, y una vez seleccionado y encontrado el *frame* de la página que queríamos, indexamos sobre ella con el *page offset*.
La MMU (*Memory Management Unit*) es la unidad encargada de traducir direcciones lógicas en direcciones físicas. En el caso de arquitectura IA-32, la unidad de *segmentación* y *paginación* formaban el equivalente de la MMU (CPU produce dirección lógica → Se la

entrega a la unidad de *segmentación* → Se produce dirección linear → Se le entrega la dirección a la unidad de *paginación* → Se forma la dirección física).

El tamaño de las páginas (y de los marcos) varía según el hardware que se utilice. Siempre el tamaño de una página es una potencia de 2, y el peso varía entre los 4KB y 1GB por página, dependiendo de la arquitectura que usemos.

Asistencia del Hardware:

Cuando el scheduler selecciona un proceso para ser ejecutado, se deben recargar los registros de usuario y los valores de la page-table apropiada del hardware con la page-table almacenada del usuario.
Aunque guardar la page-table en la memoria principal nos garantiza un *context switch* más rápido, nos genera un tiempo de acceso a memoria más lento.
La solución a este problema es el uso de una especial y pequeña cache llamada **translation look-aside buffer** (TLB). La TLB contiene unas pocas entradas de page-table. Cuando el CPU genera una dirección de memoria lógica, la MMU se fija inicialmente si esa dirección se encuentra en la TLB. Si está, retorna inmediatamente el *frame* ligado a esa dirección. Caso contrario, se debe realizar una traducción de direcciones como ya sabemos (Lógica → Virtual → Física). Adicionalmente, ese *frame* nuevo que buscamos se agrega a la TLB (hay que reemplazar una dirección cualquiera si la tabla está llena).
Hay ciertas CPUs que tienen distintos niveles de TLB, con memorias cada vez más rápidas. Por ejemplo, el Intel Core i7 tiene un primer nivel de caché L1 con 64 entradas, y luego un segundo nivel de caché L2 con 512 entradas.

Swapping:

Para ser ejecutado, las instrucciones y datos de un proceso se deben encontrar en memoria. Sin embargo, una porción del proceso puede ser *swappeado* temporalmente afuera de la memoria a un almacén de respaldo.

El *swapping* hace posible que el espacio físico total de todos los procesos exceda el espacio total de memoria física real que hay en el sistema. Esto nos permite incrementar el nivel de multiprogramación en el sistema. Existen distintos tipos de *swapping*.

Standard Swapping

Consiste en mover procesos completos entre la memoria principal y el almacén de respaldo. Este almacén es, por lo general, un rápido almacenamiento secundario. Cuando un proceso (o una parte de él) es movido a este almacén, los datos asociados a la tarea deben ser escritos también en el almacenamiento secundario. Para un proceso *multithread*, todas las estructuras de datos de cada *thread* debe ser swappeada al almacén también.
La ventaja del *swapping* estándar es que permite que la memoria física sea sobreescrita.

Swapping con paginación

El *swapping* estándar quedó obsoleto por el tiempo que demandaba mover un proceso entero entre el almacén de respaldo y la memoria principal.
Una nueva estrategia que se optó utilizar es el swappear algunas páginas de un proceso, en vez de todas. Cabe destacar que no necesitamos que todas las páginas de un proceso se encuentren en memoria para que se pueda ejecutar.

Virtual Memory

La memoria virtual es una técnica que permite la ejecución de procesos que no están completamente en memoria. Una de las grandes ventajas de este esquema es que los programas pueden ser más grandes que la memoria física. Funciona como un separador entre las direcciones lógicas y direcciones físicas. Esta separación le brinda a los programadores una enorme cantidad de memoria virtual para escribir sus programas, cuando solo una pequeña porción de memoria física está disponible.

Demand Paging

Una implementación de la memoria virtual se denomina **demand paging**. Como sabemos, no todas las páginas de un programa deben estar cargadas en memoria para que este funcione. Teniendo esto en cuenta, la estrategia que se tomó es SOLO cargar en memoria aquellas páginas que son **demandadas** durante la ejecución de un programa. Entonces, las páginas que nunca son accedidas, nunca son cargadas en memoria.
Un sistema con **demand paging** es similar a un sistema de paginación con *swapping*.

La idea del *demand paging* es que, cada vez que un proceso en ejecución demande una página, se genere un *page fault* en el sistema, logrando así que el handler de la excepción se encargue de traer la página a memoria.
Una versión alternativa de *demand paging* se llama **pure demand paging**, que consiste en iniciar a ejecutar un proceso sin que NINGUNA página esté cargada en memoria, generando constantemente *page faults*, pero cargando de igual forma las páginas en memoria.

El hardware necesario para poder implementar *demand paging* es el mismo que se requiere para paginación y *swapping*:

1) **Page table.** Una tabla con las páginas, en donde se pueda marcar el atributo de *válida-inválida* para garantizar la protección de la memoria.
2) **Memoria secundaria.** Memoria de almacén para guardar las páginas que no están siendo usadas en el momento.

Un requerimiento crucial para implementar *demand paging* es la habilidad de poder reiniciar una instrucción luego de una *page fault*. Veamos un ejemplo de suma entre A y B, guardando el resultado en C.

- Fetch de la instrucción (ADD)

- Fetch A
- Fetch B
- add A B
- Guardar la suma en C

Pero si fallamos cuando intentamos guardar el resultado en C (porque C es una página que no estaba cargada en memoria), hay que traer la página a memoria, corregir la page-table, y volver a ejecutar la instrucción.

Este problema recién mencionado no trae muchas dificultades, puesto que solo debemos ejecutar nuevamente una única instrucción. La dificultad más grande viene cuando una instrucción puede modificar distintas direcciones. Por ejemplo, si intentamos mover una cierta cantidad de bytes de una página a otra, si la página fuente o destino se extiende a ambos lados de los límites de una página, vamos a tener una *page fault* luego de haber escrito parcialmente en una dirección.
Una solución al problema es que el microcódigo intente acceder a los bordes de ambos bloques. Si vamos a tener una *page fault*, la tendremos en este momento. Existe otra solución que hace uso de registros temporales, donde se guardan las direcciones que fueron escritas y sus valores previos. Si surge una *page fault*, todos los valores anteriores son escritos a las direcciones correspondientes. De esta forma, estaríamos "reseteando" las direcciones, llevándolas a su estado inicial antes de ser escritas.

Copy-on-Write

Recordemos que la syscall *fork()* crea a un proceso hijo a partir del proceso que llamó a la función. Ese proceso hijo es una copia del padre, pues tiene una copia del espacio de direcciones del proceso padre, duplicando así la cantidad de páginas en posesión del creador.

Hay una técnica denominada **copy-on-write** la cual, ante un llamado de *fork()*, permite que tanto el hijo como el padre compartan las mismas páginas. Esas páginas están marcadas como *copy-on-write*, lo que significa que si algún proceso cualquiera escribe una (o varias) de ellas, se crea una copia de esa(s) página(s).

Notas sobre vfork():
Hay muchas versiones de UNIX que brindan una syscall llamada **vfork()** (*Virtual Memory Fork*), la cual es una variación de fork(). vfork() trabaja sin *copy-on-write*. A diferencia de fork(), cuando invocamos esta "nueva" syscall, el proceso padre es suspendido mientras que el proceso hijo usa el mismo espacio de direcciones que tiene el padre. Si el hijo modifica alguna página del *address space* del padre, esto será visible para el padre cuando resuma su actividad.
Como vfork() no crea ninguna copia de páginas, es un método extremadamente eficiente a la hora de crear procesos, aunque puede ser peligroso si el proceso hijo modifica el espacio de direcciones del padre. La idea del uso de vfork() es que el proceso hijo llame inmediatamente a *exec()* ni bien es creado.

Reemplazo de Páginas

En paginación se tiene una lista enlazada en donde tenemos las direcciones de los *page frames* libres en memoria física. Si en algún momento sucede que esta lista está **vacía**, significa que toda la memoria está ocupada, por lo que debemos reemplazar alguna página que se encuentre ubicada en la memoria física.

La idea de *page replacement* es la siguiente: Si ningún *frame* está libre, buscamos uno que no se esté usando en el momento y lo liberamos. Notemos que si ningún *frame* está disponible, entonces ocurren **dos** transferencias de páginas: una de la página que sale, y otro para la página que entra en memoria.
Para reducir este *overhead*, se utiliza un atributo de las páginas: el bit **dirty**. Este bit nos indica si la página fue escrita cuando estaba en memoria. En el caso de que el bit sea 1, significa que se escribió, por lo que debemos bajar la página a disco con los datos actualizados. En el otro caso, alcanza con solo reemplazar la página en memoria.

FIFO Replacement

Es como el nombre lo indica. Cuando ya no hay *frames* disponibles, la página que se elige para desalojar es la más antigua (la primera que se guardó en memoria). Podemos crear una FIFO queue que contenga todas las páginas que se encuentran en memoria y reemplazamos la página que se encuentra en la cabeza de la cola.
El algoritmo de reemplazo FIFO es fácil de entender y programar, aunque no tiene una gran *performance*. Por un lado, la página que desalojemos puede ser un módulo de inicialización que usamos hace mucho tiempo y no volveremos a usar. Pero por el otro, la página puede contener variables que están en constante uso.

Optimal Page Replacement

El objetivo es sencillo, ser el algoritmo con el menor *page fault rate* de todos. La idea del mismo es reemplazar la página que no será usada por el período de tiempo más largo. Sin embargo, este algoritmo es difícil (o imposible) de implementar, puesto que necesitamos saber lo que sucederá en el futuro.

Least Recently Used (LRU) Page Replacement

El algoritmo de FIFO usa el tiempo de cuando una página fue traída a memoria, mientras que el algoritmo de OPR usa el tiempo de cuando una página será usada. Acá, LRU usa el pasado reciente como una aproximación del futuro, para reemplazar la página que **no fue usada** por la cantidad de tiempo más larga.
En la implementación, cada página tiene asignado un tiempo de uso, el cual es cambiado cada vez que la página es usada. Cuando se usa, su tiempo es actualizado al tiempo más reciente, dejando en claro en el sistema que esa página fue usada hace poco. Al momento de reemplazar una página, el algoritmo de LRU selecciona la página que no fue usada por el período de tiempo más largo. Notar que es necesario un reloj lógico para poder armar este sistema. Hay que tener en cuenta el *overflow* del clock.
Es obligatoria la asistencia del hardware a la hora de implementar LRU.

LRU-Approximation Page Replacement

Si no tenemos la ayuda del hardware, podemos optar por una ligera ayuda del software. Para ello, podemos usar el **bit de referencia** para cada página. Este bit estará seteado en 1 si la página en memoria volvió a ser usada desde que fue cargada, sino, estará en 0. Acá podemos determinar si las páginas fueron o no usadas, aunque no podemos decidir el orden.

Second-Chance Algorithm

El reemplazo de páginas del algoritmo de *second-chance* está implementado como FIFO. Aquí, hacemos uso de una FIFO *queue* con la diferencia de que, cada vez que una página es usada, se la "refuerza". Cuando se intenta desalojar una página reforzada, esta es enviada al final de la FIFO *queue*. Si la página no está reforzada, es desalojada normalmente.
Para reforzar una página, se le marca el **bit de referencia**. Si la página es enviada al final de la cola, el bit se vuelve a setear en 0.

Existen también otros algoritmos irrelevantes que se basan en contar la cantidad de veces que una página fue referenciada, por ejemplo:

- **Least Frequently Used (LFU):** Se desaloja la página que tenga el contador más chico.
- **Most Frequently Used (MFU):** Se desaloja la página con el contador más alto con el argumento de que las páginas con el contador más bajo todavía no se usaron, puesto que se trajeron hace poco a memoria.

Ninguno de esos 2 es útil en la práctica y son poco usados.

Frames Allocation

Un problema con el que nos enfrentamos con paginación es el de ver cuántos *page frames* le asignamos a cada proceso. Un método es asignarle a todos una cantidad equitativa. A este algoritmo le llamamos "**equal allocation**". Otro esquema es el de darle *frames* a un proceso dependiendo de su tamaño. A este otro algoritmo le llamamos "**proportional allocation**".
En ambos métodos, la asignación varía dependiendo del nivel de multiprogramación que haya. También, en ningún esquema se distinguen a los procesos de menor y mayor prioridad. Todos reciben *frames* de igual manera.

Veamos una forma de *swappear* páginas (o agregar nuevas) una vez que ya se hayan asignado todos los *frames* a cada proceso:

- ### Global Allocation
  Permite que un proceso seleccione un *frame* de la *pool* general de *frames*, es decir, puede incluso elegir *frames* que están siendo usados por otros procesos.

- **Local Allocation**
  El proceso se maneja únicamente con el set de *frames* que se le fue dado inicialmente. Solo puede *swappear* páginas entre esos *frames* dados.

Thrashing

Cuando no alcanza la memoria y los procesos compiten por usarla, recibimos constantes fallos de página, por lo que el SO cambia todo el tiempo páginas de memoria a disco, en ida y vuelta. A esta alta actividad de cambio de páginas se la denomina **thrashing**.
Un programa está *dando vueltas* cuando gasta más tiempo intercambiando páginas que ejecutando.

Una forma de limitar los efectos de *thrashing* es utilizando **local replacement algorithm**. Vimos antes que, si la asignación de *frames* es local, entonces un proceso solo puede intercambiar páginas con el set de *frames* que se le otorgó inicialmente. De esta manera, evitaríamos que un proceso le robe *frames* a otros, y así evitaríamos que luego esos otros procesos recurran a tomar *frames* de otros procesos, y así. Sin embargo, esta solución no resuelve todo nuestro problema.

Para prevenir completamente el *thrashing*, tenemos que otorgarle a cada proceso la cantidad de *frames* que necesita. Para saber cuántos *frames* necesita, tenemos que ver cuántos está usando actualmente. Este *approach* define el **modelo de localidad** de ejecución de procesos. Definimos como una localidad a un conjunto de páginas que están siendo usadas activamente, juntas.

El **modelo de conjunto de trabajo** (*working-set model*) es un conjunto de páginas que se encuentran constantemente activas. El modelo se basa en la suposición de la localidad. Si una página tiene un uso activo, se encuentra en el *working-set*. Si no se sigue usando, luego de un tiempo dejará de pertenecer al conjunto. Entonces, el *working-set* es una aproximación de la localidad del programa.
Si para cada proceso que hay en el sistema computamos el tamaño de su *working-set*, entonces podemos saber cuántos *frames* son demandados en TOTAL. Si la demanda total de *page frames* es mayor a la cantidad de *frames* que tenemos disponibles, vamos a tener **thrashing**.

El uso del *working-set* es simple. Se monitorea el *working-set* de cada proceso, y se le otorga a cada uno la cantidad de *frames* que necesite. Si nos sobran *page frames* luego de asignarle a todos los procesos, entonces podemos inicializar otra tarea. Si la cantidad de *frames* que se piden en total excede la cantidad disponibles, entonces el SO elige una tarea y la suspende. La tarea suspendida puede ser resumida más tarde.
Esta tarea "suspendida" se *swappea* de memoria a disco, y su ejecución puede ser retomada desde el punto en el que se dejó antes de ser *swappeada*. Luego, se puede llevar el proceso de vuelta a memoria para seguir ejecutándolo.

7) Administración de E/S

Los dispositivos de E/S (o de I/O) pueden ser categorizados como de almacenamiento, comunicaciones, interfaz de usuario, etc. Una de las mayores preocupaciones de los diseñadores de los sistemas operativos es el control de los dispositivos conectados a la computadora. Para encapsular los detalles y rarezas de los distintos dispositivos de I/O, el kernel del SO está armado para trabajar con los controladores de cada aparato. Estos *device-drivers* brindan una interfaz para acceder al subsistema de E/S.

Un dispositivo se comunica con el sistema informático mandando señales por un cable (o incluso por el aire). Tienen un punto de conexión con la computadora, o un **port**. Si los distintos dispositivos comparten un conjunto de cables, a esa conexión le llamamos **bus**.
Un *bus* es un conjunto de cables con un rígido protocolo predefinido que brinda un set de mensajes que pueden ser enviados a través de los cables.

Un *controlador* es un conjunto de electrónica que puede controlar un *port* o un *bus*. Algunos dispositivos tienen un controlador interno, por ejemplo, los discos duros tienen un circuito eléctrico integrado. Cabe destacar que *device-drivers* y *device-controllers* son distintos. Los controladores son un tipo de programación de hardware que actúan como un puente entre el SO y los dispositivos.
Los controladores tienen más de un registro para recibir datos y señales de control. De esta manera, el procesador puede enviar comandos y datos a los controladores para lograr una transferencia de E/S.

Polling

El dispositivo tiene dos registros por los que se "comunica": el **status register**, y el **command register**. En el **status register**, el dispositivo utiliza el **busy bit**, el cual estará prendido si el dispositivo está laburando. Si se apaga, está listo para recibir otra instrucción. Luego, cuando un comando está listo para ser ejecutado por el dispositivo, el driver setea en 1 el **command bit** del **command register**.
En resumen, la CPU está chequeando constantemente si se comunicó o no el dispositivo de E/S. El *polling* es poco práctico porque otros procesos de la CPU no son realizados cuando se verifica el dispositivo de E/S. Es por eso que se opta por otra alternativa, en donde el dispositivo se encarga de comunicarle a la CPU que está lista para obtener otra instrucción. (**interrupciones**).

Interrupciones

El hardware de la CPU tiene un cable llamado *interrupt-request line*, el cual chequea luego de ejecutar cualquier instrucción. Cuando se recibe una interrupción de un controlador, la CPU guarda su estado y salta a la rutina de atención de interrupciones correspondiente. Aquí, el controlador del dispositivo "levanta" una interrupción que es atrapada y despachada por el CPU, para que el handler de interrupción se encargue de atender al dispositivo.

Es importante notar que para utilizar este método, es necesario aplazar la atención de interrupciones durante la ejecución de un proceso crítico, y también se deben diferenciar las interrupciones con niveles de prioridad, para poder así responderlas con el apropiado grado de urgencia a cada una (puesto que pueden llegarnos múltiples interrupciones a la vez).

Un SO moderno, al ser encendido, sondea los buses del hardware para determinar qué dispositivos de I/O están conectados, y así saber qué *handlers* de interrupciones debe instalar. Cada interrupción tiene su propio *handler*, permitiendo que el CPU sepa cuál es la rutina que debe seguir para solucionar la misma. También, durante la etapa de I/O, varios *device-controllers* levantan interrupciones cuando están listos para ser servidos. (ya sea porque se encontraron con un error, hay *data* para que sea leída por el SO, etc)

Veamos entonces las ventajas y desventajas que nos encontramos con estos dos mecanismos

- **Polling**
  - PROS: Sencillo, *context switch* controlados
  - CONTRAS: Consume CPU

- **Interrupciones**
  - PROS: Eventos asincrónicos poco frecuentes
  - CONTRAS: *Context switches* impredecibles

Direct Memory Access (DMA)

Para un servicio que realiza largas transferencias de datos (como los discos rígidos, por ejemplo), es excesivo utilizar un procesador de propósito general que se encargue de mirar su *status-bit* para guardar datos en un registro (de a un byte a la vez). Para evitar usar constantemente el CPU principal, las computadoras suelen enviar parte de su trabajo a un procesador de propósito especial, llamado controlador de **DMA**.

Entonces, la ventaja de DMA es que nos presenta la posibilidad de transferir datos de la memoria del sistema sin la intervención de la CPU. Inicialmente, para comenzar una transferencia DMA, se debe escribir un comando DMA en memoria. Dicho bloque debe contener un puntero a la *fuente* a transferir, un puntero al *destino*, y la cantidad de bytes a ser transferidos. Luego, el CPU le escribe al *DMA controller* la dirección de memoria de dicho bloque, y este controlador se encarga del resto; el CPU puede seguir con otro trabajo.
El *DMA controller* opera sobre el bus de la memoria directamente, escribiendo direcciones sobre el bus para realizar las transacciones de datos (sin la asistencia del CPU).

Application I/O Interface

Existen distintas técnicas e interfaces para el SO que permite que trate a los dispositivos de E/S de forma estándar. Una de ellas son los ***drivers***. Son componentes de software muy específicos; un conjunto de funciones que están encapsuladas en módulos del kernel. Corren

en máximo privilegio y conocen las particularidades del hardware con el que están conectados.

Los dispositivos pueden dividirse en dos grupos:

- **Char device:** Dispositivos en los cuales se transmite la información byte por byte (teclado, terminales, mouse, etc).
- **Block device:** Dispositivos en los cuales se transmite la información en bloques (discos rígidos, memorias flash, CD-ROM, etc).

Scheduling de E/S

Programar un conjunto de llamados de E/S significa que debemos determinar el mejor orden para ejecutarlos. En un disco rígido tenemos una cabeza lectora que se mueve. Girarla lleva tiempo. Supongamos que la cabeza se encuentra cerca del comienzo del disco, y hay tres llamados de lectura pendientes sobre este mismo disco.

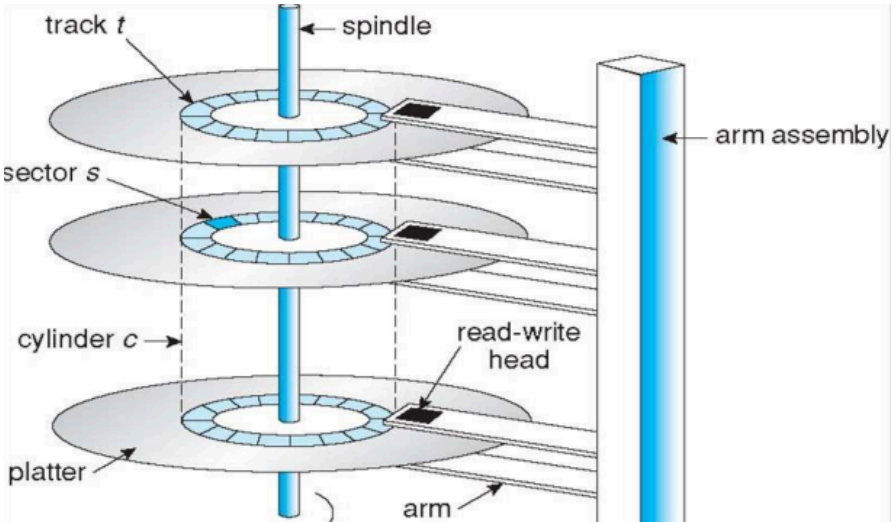
1) Aplicación 1 pide un bloque de datos que está cerca del **final** del disco.
2) Aplicación 2 pide un bloque de datos que está cerca del **comienzo** del disco.
3) Aplicación 3 pide un bloque de datos que está en el **medio** del disco.

El SO puede reducir la distancia recorrida por la cabeza atendiendo los llamados en el orden 2–3–1. Reorganizar el orden de los servicios es la esencia del scheduling de E/S.

Por lo general, cada dispositivo tiene su propia *queue* de llamados. Cada vez que una aplicación emite una *syscall* de E/S bloqueante, ese pedido es colocado en la *queue*. Luego, el scheduler de I/O reorganiza la *queue* para mejorar la eficiencia del sistema, y el tiempo de respuesta de cada aplicación. Además, el SO *intenta* ser justo, para que ninguna aplicación reciba un mal servicio.

Una de las responsabilidades del SO es usar el hardware de forma eficiente. Para los discos rígidos, esta eficiencia significa *minimizar* el tiempo de acceso al mismo, y *maximizar* el ancho de banda de transferencia de datos.
Para los HDDs, el tiempo de acceso de los mismos tiene dos componentes importantes:

- **Seek time:** Lo que tarda el brazo del disco en mover las cabezas lecto-escritoras hacia la pista deseada.
- **Rotational latency:** El tiempo que tarda en girar el plato (disco) para que la cabeza se ubique sobre el sector deseado.



La existencia de una *queue* de pedidos de E/S para cada dispositivo permite que los *drivers* mejoren la *performance* del sistema ordenando la *queue*, evitando la menor cantidad de movimientos posibles de la cabeza lecto-escritora.

Vamos a ver entonces las distintas políticas de ***scheduling en disco***, ya que la clave para obtener un buen rendimiento de E/S es manejar apropiadamente el disco. (Además tenemos que verlo porque **Lord Baader** lo toma en el final)
(ACLARACIÓN: hablo de *scheduling* en discos HDDs, no en NVM)

FCFS Scheduling

Por supuesto, la forma más sencilla de *scheduling* en disco es *First-Come, First-Served* (o FIFO). Este algoritmo es esencialmente **justo**, pero no provee el servicio más rápido.
Es fácil de implementar, y ya sabemos como funciona: se atienden los pedidos de lectura/escritura a disco en el orden de llegada.

SCAN Scheduling

En el algoritmo SCAN, el brazo del disco comienza en una punta del disco, y se mueve hacia la otra punta. A medida que se va moviendo y alcanzando los distintos cilindros, atiende y cumple los distintos llamados pendientes hasta llegar a la otra punta del disco. Una vez que alcanza el final, realiza el mismo camino en reversa. La cabeza se mueve constantemente de adelante para atrás y de atrás para adelante.

El algoritmo SCAN a veces es llamado **elevator algorithm**, ya que el brazo del disco se comporta como un ascensor en un edificio: primero cumple todos los pedidos yendo hacia arriba, y luego cumple los restantes yendo en reversa.
Podría suceder que llegue una solicitud para el cilindro inmediato anterior, pero tenga que esperar a que se cambie de dirección. Además, el tiempo de espera no es uniforme.

C-SCAN Scheduling

Circular SCAN (C-SCAN) *scheduling* es una variante de SCAN, diseñado para que los tiempos de espera sean uniformes. Funciona de la misma manera, con la diferencia de que una vez que la cabeza lecto-escritora alcanza la otra punta del disco, inmediatamente regresa al comienzo del disco, sin atender ningún pedido de E/S en el “viaje” de vuelta.
El C-SCAN *scheduling* trata los cilindros como una lista circular.

Selección de un Algoritmo de Scheduling de Disco

Hay muchos algoritmos de *scheduling* que no son mencionados en el libro ya que suelen ser usados muy raramente. Uno no mencionado es el esquema **SSTF** (*Shortest Seek Time First*), cuya idea es atender como próximo el pedido más cercano a donde esté la cabeza en el momento. Aunque este algoritmo mejore los tiempos de respuesta, puede causar inanición.
Entonces, ¿cómo hacen los SO para seleccionar e implementar uno? Para cada tipo de *scheduling*, la performance depende fuertemente en la cantidad y en el tipo de pedidos que recibe.

En la práctica, ningún algoritmo mencionado es usado de manera pura. Se deben establecer prioridades para cada pedido de lectura/escritura sobre el disco (ya sea para bajar páginas de cachés, hacer un *swap* de procesos, etc).

Linux diseñó el ***deadline scheduler***. Este *scheduler* tiene dos *queues* separadas: una para pedidos de lectura, y otra para pedidos de escritura. Además, le da prioridad a los pedidos de lectura, ya que los procesos tienden más a bloquearse mientras esperan leer un archivo. Cada *queue* está ordenada por LBA (*Logical Block Address*). Esencialmente implementa C-SCAN.

NOTA: *Para aquel lector de este resumen que lo esté matando la curiosidad, el scheduling que suelen usar los dispositivos NVM es, generalmente, FCFS. Esto se debe a que los NVM son mucho más rápidos porque no tienen un disco giratorio, por lo tanto, no tienen ningún tipo de* ***seek time*** *ni* ***rotational latency****.*
Los dispositivos NVM pueden ser "discos" SSD, un pendrive, o una tarjeta DRAM. En cualquiera de sus formas, actúa y puede ser tratado de igual manera.

Gestión de Disco

- ## Formateo

Antes de que un dispositivo de almacenamiento pueda guardar datos, tiene que ser dividido en sectores que el controlador pueda leer y escribir. Este proceso es llamado **formateo físico** (o *formateo de bajo nivel*). Este proceso llena el dispositivo de datos especiales en cada dirección de almacenamiento. La estructura de estos datos para una página consiste en información que puede ser usada por el controlador, ya sea un número de página/sector, la detección de un error o código de corrección.
Los discos son formateados físicamente por el manufacturador de los mismos. Así, los fabricantes pueden testear y *mapear* (bloques de direcciones lógicas a páginas) los discos antes de ser vendidos.

Antes de que pueda guardar archivos en el disco, el SO debe registrar sus estructuras de datos en el dispositivo.
Primero, se debe realizar una **partición** del disco, en donde se divide el disco en uno o más grupos de bloques o páginas. El SO puede entonces tratar a cada bloque como si fuera un dispositivo distinto. Por ejemplo, una partición puede contener un *file system* que contenga una copia de todo el código ejecutable del sistema operativo, otra partición que consista en espacio de *swap*, y otra que contenga un *file system* con todos los archivos de usuario.
Segundo, es la creación y gestión del **volumen** de cada partición. Definimos el tamaño que tendrá cada partición.
Tercero, se crea un *sistema de archivos*. El SO almacena las estructuras de datos iniciales del *file system* en el dispositivo. Estos datos incluyen información sobre un directorio vacío inicial, y también memoria libre y ya asignada.
// En *Windows*, cada partición viene dada por una letra (C:, D:, E:).

- ## Booteo

Cuando una computadora es prendida, tiene que tener un programa inicial para correr. En la mayoría de las computadoras, el ***bootstrap*** inicial está guardado en una memoria *flash*, y se encarga de cargar en memoria ciertos sectores del comienzo del disco, y los comienza a ejecutar. Es decir, a partir de un *bootstrap* muy pequeño, se puede traer uno más complejo del almacenamiento secundario. El *bootstrap* completo se encuentra guardado en "***boot blocks***" en una dirección conocida del dispositivo.
En resumen, el programa cargado es muy pequeño. No llega a ser el SO, sino más bien un cargador del SO.

- ## Bloques Dañados

Como los discos tienen partes en constante movimiento y pequeñas tolerancias, son propensos a fallar. En algunos casos, falla completamente el disco, por lo que la solución es reemplazar el disco en su totalidad (y pasar el *backup* que teníamos del disco al nuevo). En otros casos, ciertas regiones del disco pueden volverse defectuosas (a veces, algunos discos vienen con **bloques dañados** de fábrica). Dependiendo del disco y su controlador, estos bloques son manejados de distintas maneras.

En discos viejos, los bloques dañados se "solucionan" de forma manual. Mientras el disco está siendo formateado, se escanea el disco en busca de **bad blocks**. Cualquier bloque que sea descubierto, es marcado como *inutilizable*, así el sistema de archivos no tiene en cuenta esa región.

En discos más sofisticados, el controlador del dispositivo lleva una lista de los bloques dañados que tiene el disco. Esta lista es inicializada en el **formateo físico** del disco, en la fábrica, y es actualizada durante toda la vida del disco. Además, durante el formateo físico, se guardan de forma apartada *sectores de repuesto* que no son visibles por el SO. Luego, se le puede pedir al controlador que reemplace estos bloques dañados con los sectores de repuesto. Este esquema es conocido como **forwarding** o **spare sectors**.
Veamos como funciona, paso por paso:

- El SO intenta leer el bloque *lógico* 87.
- El controlador realiza el cálculo (traducción) correspondiente y encuentra que el bloque 87 está dañado. Se lo reporta al SO como un error de I/O.
- El controlador del dispositivo reemplaza el bloque dañado con un sector de repuesto.
- Ahora, cuando el SO quiera leer la dirección lógica 87, el pedido es traducido al sector de reemplazo por el controlador del dispositivo.

Estas redirecciones realizadas por el controlador pueden afectar el algoritmo de *scheduling* de disco. Es por eso que, la mayoría de los discos, están formateados de forma tal que puedan ofrecer *sectores de repuesto* en cada cilindro, y también tienen un *cilindro de repuesto*. Cuando se *remappea* un bloque dañado, el controlador usa un sector de repuesto de su mismo cilindro si es posible.

Una alternativa a **spare sectors** es ***sector slipping***. Veamos un ejemplo: supongamos que el bloque 17 está dañado, y que el primer *sector de repuesto* es el 202. Todos los bloques, desde el 17 hasta el 202, son *remapeados* por el controlador. El bloque 202 es copiado en el repuesto, el 201 es copiado sobre el 202, luego el 200 se copia sobre el 201, y así hasta que el 18 sea movido al 19. Así, el bloque 17, que está dañado, puede ser *mapeado* al 18.

Puede suceder también que un bloque que esté escrito se dañe. Esto resulta en datos perdidos. El archivo que estaba usando ese bloque debe ser reparado (puede restaurarse con un *backup*), pero requiere de intervención manual.

Al igual que antes, los dispositivos NVM son superiores a los HDDs. Los dispositivos NVM tienen bits, bytes y hasta páginas que funcionan como sectores de reemplazo. La gestión de las áreas dañadas en estos dispositivos es más simple que en los discos, ya que, como sabemos, no se tiene ningún tipo de **seek time** ni de **rotational latency**.

RAID Structure

Los dispositivos de almacenamiento se hicieron más chicos y más baratos con el paso del tiempo, es por eso que hoy en día es más accesible agregarle varios discos a una misma computadora. Tener muchas unidades en un sistema brinda oportunidades para mejorar el ritmo con el que los datos son escritos y leídos, si los dispositivos son manejados en paralelo. Este tipo de organización nos permite tener una mayor fiabilidad de los datos; esto se debe a que podemos tener información redundante almacenada en distintos discos, por lo tanto, si un disco falla, este problema no nos conduce a la pérdida de esa información. Hay una amplia variedad de técnicas para organizar un disco, llamadas ***Redundant Arrays of Independent Disks*** (**RAID**).

Una forma, pero muy cara, de mejorar la *fiabilidad* de la información en nuestro sistema, es tener dos discos. En el primero guardamos toda nuestra información, y en el segundo se guarda una copia de toda la información que posee el primero. Así, si el primero falla, tenemos el segundo disco para que nos "cubra". La única forma de que falle este método es si el segundo disco falla antes de que falle el primero. (Puede pasar también que el primer disco nunca sea arreglado, por lo que el segundo eventualmente falle y nos genere una pérdida de datos.)
Luego, si tenemos dos (o más) discos en el sistema, se puede mejorar la *performance* del mismo al hacer lecturas o escrituras en paralelo. Se puede buscar información en los distintos discos a la vez, mejorando el *throughput* y *response time* del sistema.

RAID Levels

- **RAID 0 *(striping)***

No aporta redundancia. Consiste en tener diversos discos en nuestro sistema, pero no se clona la información de ninguno en ningún otro. Los bloques de un mismo archivo son distribuidos

entre uno (o más) discos, pero evitando la redundancia. Lo único que aporta este nivel de RAID es mejorar la *performance* del sistema al poder leer o escribir paralelamente.

- ## RAID 1 (*mirroring)*

Es completamente redundante. Se espejan los discos, es decir, si tenemos 4 discos, agregamos 4 discos extra que contengan la misma información que los otros cuatro, para evitar la pérdida de la información. Es muy costoso, e incluso las escrituras pueden costarnos el doble (al tener que escribirlas en dos o más discos).

- ## RAID 4

Antes de explicar RAID 4, es importante que sepamos qué es y cómo funciona *Error-Correcting Code* (**ECC**).

ECC es un método que se usa en los sistemas de almacenamiento y memoria de las computadoras. Sirve para detectar (y a veces arreglar) si un bloque de datos fue dañado durante una transmisión o almacenamiento del mismo. Para detectar los errores, ECC tiene un bit extra (*parity bit*) que se basa en los datos que están siendo almacenados. Los *parity bits* son generados de forma tal que el sistema pueda reconocer si un error de un **único** bit ocurrió durante la transmisión o almacenamiento de un bloque.

Entonces, vamos a hablar ahora sobre RAID 4.
RAID 4 consiste en N+1 dispositivos de almacenamiento. N dispositivos son usados de forma común y corriente, como en RAID 0, mientras que el disco restante es un disco dedicado a almacenar información de *paridad* (*parity information*). Entonces, el "disco de paridad" es usado para poder detectar errores de información entre los otros discos. Además, este nivel de RAID puede corregir errores siguiendo cierto mecanismo, el cuál puede determinar, para cada bit de un bloque dañado, si le corresponde un 0 o un 1 (basándose en el bit de paridad).

Hay cierto problema de *performance* con RAID 4, ya que resulta costoso computar y escribir el *XOR parity*.

- ## RAID 5

A diferencia de RAID 4, este nivel de RAID distribuye todos los bloques de datos e información de paridad entre los N+1 dispositivos, en vez de usar N dispositivos para datos y el otro para información de paridad. Para cada conjunto de N BLOQUES, uno de los discos guarda información de paridad, mientras que los otros almacenan datos. Al distribuir la información de paridad entre todos los discos, RAID 5 evita el constante uso de un mismo disco de paridad.
RAID 5 usa datos redundantes, pero los distribuye entre los N+1 discos, es decir, no sucede que un disco es utilizado únicamente para almacenar información redundante. RAID 5 puede soportar la pérdida de un disco cualquiera. Lo difícil de este método es mantener la paridad distribuida.

- **RAID 6**

Funciona como RAID 5, con la diferencia de que guarda información redundante extra para poder combatir fallos de múltiples discos. Usa un método distinto al XOR de paridad que usaban RAID 4 y RAID 5; se basa en otros ECC, como **Galois field math**. Esto se debe a que, al guardar más información redundante, dos bloques de paridad resultarían idénticos.

- **RAID 0+1**

Es una combinación de RAID 0 con RAID 1. RAID 0 nos ofrece *performance*, mientras que RAID 1 nos ofrece *fiabilidad*. Generalmente, este nivel nos brinda una mejor *performance* que RAID 5, pero, como ya sabemos, hay que duplicar la cantidad de discos de almacenamiento que tenemos, lo que resulta sumamente caro.

- **RAID 2 y 3**

Estos niveles de RAID no son mencionados en el *Silberchatz* (o al menos yo no los encontré leyendo el capítulo 11), así que procedo a listar lo que ví en la teórica de Baader:

La idea es tener, por cada bloque, información adicional que nos diga si se dañó o no. Además, cierto tipos de errores pueden ser corregidos automáticamente, recomputando el bloque dañado a partir de la información redundante. Adicionalmente, cada bloque lógico se distribuye entre todos los discos participantes.
RAID 2 requiere 3 discos de paridad por cada 4 discos de datos, mientras que RAID 3 requiere tan solo 1. Puede requerir mucho procesamiento para computar las redundancias.

Backup

// En esta teórica (y en el programa de la materia) se habla de copias de seguridad. En el libro no encontré nada entre los capítulos 11 y 12, sino que lo encontré en el 14 ("File System Implementation"). Lo que voy a escribir a continuación es un resumen de ese apartado del capítulo

A veces, los dispositivos de almacenamiento fallan, y se debe buscar una forma de asegurar que los datos perdidos no se queden perdidos para siempre. Para lograr esto, se puede optar por realizar un *backup* de los datos de un dispositivo, copiando la información de un disco a otro (si el disco que estamos usando para copiarle los archivos es removible, mejor).

Copiar todos los datos puede ser muy costoso (y llevar mucho tiempo). Es por eso que, para minimizar la cantidad de archivos que copiamos, podemos usar la información del directorio de cada archivo. Por ejemplo, si hicimos *backup* de un archivo, y cuando estamos por realizar otra vemos que ese archivo no fue modificado, no es necesario copiarlo nuevamente.
Un ciclo típico de *backup* es como el siguiente:

1) Día 1: Copia de seguridad de TODOS los archivos del sistema de archivos. (**full backup**)

2) Día 2: Copia de seguridad de los archivos que fueron modificados desde el día 1. (**incremental backup**).
3) Día N: Copia de seguridad de todos los archivos que fueron modificados desde el día N-1. Regresar al día 1.

Alternativamente, se puede realizar una **copia diferencial**, la cual consiste en realizar una copia de seguridad de todos los archivos que fueron modificados desde el último **full backup**.

Luego, para restaurar la información:
- Si solo hago **copias totales**, alcanza con tomar la última realizada.
- Si hago **copias diferenciales**, necesitamos la última copia total y la última diferencial.
- Si lo que tenemos son **copias incrementales**, necesitamos la última copia total y todas las copias incrementales entre la última total y la última incremental.

Spooling

Un **spool** es un *buffer* que mantiene el output de un dispositivo, por ejemplo una impresora, que no puede aceptar datos intercalados. Aunque una impresora pueda atender solo un pedido a la vez, diversos programas pueden querer imprimir su output concurrentemente, sin que sus outputs se mezclen entre sí. El SO resuelve este problema interceptando todos los pedidos y colocándolos en una *queue*. La idea es designar un proceso que desencole los pedidos a medida que el dispositivo se libere. En algunos SO, el *spooling* es gestionado por un proceso en segundo plano. En otros, es controlado por un *thread* del kernel.

El nombre **SPOOL** viene de **Simultaneous Peripheral Operation On-Line**.

8) Sistemas de Archivos

Las computadoras pueden guardar información en distintos dispositivos de almacenamiento, como dispositivos NVM, HDDs, cintas magnéticas, unidades ópticas, etc. Los *archivos* son mapeados por el SO dentro de estos dispositivos físicos.
Un archivo es una colección de información relacionada, que se guarda en el almacenamiento secundario. Pueden representar un programa y datos, ya sean datos numéricos, alfabéticos, binarios, entre otros. Dependiendo del SO, los archivos tienen ciertos atributos. Algunos de ellos son:
- **Nombre.** Nombre con el que se identifica a un archivo, normalmente un *string*.
- **Identificador.** Un tag único, usualmente es un número. Sirve para identificar el archivo dentro de un *file system*. Es el nombre del archivo en forma no-legible para humanos.
- **Tipo.** Indica el tipo de archivo. (**.txt** indica que el archivo contiene texto, **.c** indica que el archivo contiene código fuente en lenguaje C)

- **Locación.** Puntero al dispositivo y lugar en el que se encuentra almacenado el archivo en ese dispositivo.
- **Tamaño.** Indica el tamaño del archivo.
- **Protección.** Indica quién puede leer/escribir/ejecutar el archivo.

Podemos ejecutar ciertas operaciones sobre los archivos (crearlos, abrirlos, escribirlos, leerlos, eliminarlos, etc). Estas operaciones incluyen la búsqueda del archivo en el directorio, hasta encontrar el archivo que se corresponda con el nombre. Para evitar esta búsqueda constante, el SO tiene una tabla llamada **open-files table**, que contiene información sobre todos los archivos abiertos. Cuando una operación es requerida, se realiza con un índice de la tabla que indica qué archivo debe ser manipulado.

Directorios

El directorio puede ser visto como una tabla de símbolos que traduce nombres de archivos en sus respectivos **file control blocks**. Los directorios pueden ser organizados de distintas formas, y esas organizaciones deben permitirnos crear, eliminar, editar, buscar (entre otras cosas) entradas dentro del directorio. Veamos cómo pueden estar organizados.

Single-Level Directory

Es la estructura más simple. Consiste en un único nivel, es decir, todos los archivos son guardados en un mismo lugar. Tiene ciertas limitaciones si la cantidad de archivos en el directorio aumenta, ya que los nombres de los archivos deben ser únicos. Puede haber distintos usuarios utilizando el mismo directorio, lo que restringe los nombres que puede elegir cada uno para sus respectivos archivos.

Two-Level Directory

Para evitar la confusión de nombres de archivos en un mismo directorio entre los distintos usuarios, se implementó una solución estándar en la que cada usuario tiene su propio directorio.
Ahora, cada usuario tiene su propio **user file directory** (**UFD**), que consiste en un directorio que lista únicamente los archivos de un solo usuario. Aquí, distintos usuarios pueden tener archivos cuyos nombres coincidan, siempre y cuando los nombres de todos los archivos en sus UFD sean únicos.
La desventaja de esta estructura es que aísla a los usuarios entre sí. Si los usuarios son independientes, nos viene bárbaro. El problema surge si los usuarios deben cooperar en alguna tarea y acceder a los archivos de otros (hay veces que no se permite el acceso a archivos de otros usuarios).
Si el acceso está permitido, un usuario debería poder nombrar el archivo de otro. El nombre de un usuario + el nombre de un archivo definen un **path name**.

Tree-Structured Directory

Como ya vimos los niveles uno y dos de un directorio, podemos generalizar su estructura. Podemos pensar un directorio como un árbol de una altura arbitraria, que permite que los

usuarios creen sus propios subdirectorios y organicen sus archivos acorde. El árbol es la estructura más común para un directorio. Entonces, aquí, un directorio contiene un conjunto de archivos y/o un conjunto de subdirectorios.

En el uso normal, cada proceso debe tener su directorio *actual*. El *directorio actual* debe contener la mayor cantidad de archivos que sean de importancia para el proceso en ejecución.

Acyclic-Graph Directory

Supongamos que dos programadores están laburando en un mismo proyecto. Los archivos de ese proyecto son guardados en un subdirectorio, que los aísla del resto de los archivos del sistema. Como los dos programadores están en el proyecto, ambos son responsables de los archivos y quieren tener acceso a ese subdirectorio. En esta situación, el subdirectorio en común debe ser **compartido**. Un directorio con estructura de árbol no permite que un subdirectorio sea compartido.
Un **grafo acíclico** permite que los directorios tengan subdirectorios y archivos compartidos. El mismo archivo o subdirectorio puede estar en dos o más directorios. Es importante notar que un archivo compartido no es lo mismo que una copia de un mismo archivo.

Existen diversas formas para implementar archivos o subdirectorios compartidos. Una de ellas, usada en los sistemas UNIX, es la creación de un nuevo archivo o subdirectorio, llamado **link**. Un link es un puntero a otro archivo o subdirectorio, y puede ser implementado como un *path name*. El SO ignora estos links al navegar un directorio, para así preservar la condición acíclica del sistema.

General Graph Directory

Un problema de usar una estructura acíclica es asegurar la no existencia de ciclos. La ventaja principal de un grafo acíclico es lo simples que son los algoritmos para determinar que ya no hay más referencias a un archivo. Si permitimos que los ciclos existan en nuestra estructura, queremos evitar que cualquier componente sea “visitado” dos veces; sería una pérdida de tiempo. Un algoritmo pobremente hecho podría encontrarse *loopeando* infinitamente. Una vaga solución a este problema es limitar la cantidad de directorios que serán visitados durante la búsqueda.

// Leí el libro y básicamente habla poco y nada de este esquema. Nos dice que si agregamos muchos links en un directorio de tipo árbol, obviamente va a resultar en un grafo con ciclos. Habla muchas pavadas, pero básicamente dice que hay algoritmos simples para evitar esos ciclos. (NO se hace uso de DFS porque es computacionalmente caro para detectar ciclos, ya que el grafo está en almacenamiento secundario)

Algunas formas de **implementar** un directorio es con una **lista enlazada** (lentísimo, ni hace falta explicar porqué), o con una **hash table**. La idea de la *hash table* es que tome un valor computado a partir del nombre del archivo, y devuelva un puntero al archivo correspondiente en la lista enlazada.

Allocation Methods

Ya estudiamos los directorios y los archivos. Ahora nos resta ver cómo hacemos para distribuir estos dispositivos en el espacio que tiene nuestro disco. Existen tres métodos importantes al momento de asignar espacio a los archivos: **contiguous**, **linked**, **indexed**.

Contiguous Allocation

Como sabemos, al hablar de almacenamiento secundario, el espacio está dividido en bloques. La asignación contigua consiste en asignarle bloques contiguos a un archivo al guardarlo en memoria, es decir, si un archivo es inicialmente guardado en el bloque 1 y ocupa otros 3 en tamaño, entonces ocupará los bloques 1, 2, 3 y 4.

Es sencillo, pero como vimos en la sección de Memoria Principal y Virtual, estos métodos generan fragmentación externa. Es muy pobretón. Una solución (pero extremadamente lenta) es tener un segundo disco en donde copiamos enteramente nuestro sistema de archivos. Así, el dispositivo principal solo tendrá espacio libre (contiguo) y podemos copiar, de forma contigua, todos nuestros archivos en este bloque gigante de almacenamiento.

Linked Allocation

La asignación enlazada soluciona todos los problemas que trae la asignación contigua. Consiste en una lista enlazada de bloques de almacenamiento; aquí, los bloques pueden estar distribuidos en cualquier parte del dispositivo. Un archivo puede tener su primer bloque en el bloque 9, continuar en el bloque 25, luego en el 16, etc.

Una desventaja de la asignación enlazada es que, si queremos leer el i-ésimo bloque de datos de un archivo, tenemos que recorrerlo secuencialmente, desde el inicio del mismo. Otra desventaja es la cantidad de espacio que se gasta en punteros, para mantener las conexiones entre cada bloque.
Una posible solución a este problema sería juntar bloques y transformarlos en **clusters**, para así usar menos punteros. Por ejemplo, podemos definir un cluster como cuatro bloques, y operar sobre el dispositivo de almacenamiento usando esa medida. El costo de esta solución trae *fragmentación interna*, obviamente.

Otro problema de la asignación enlazada es la *fiabilidad*. Como los archivos están enlazados a través de punteros, si un puntero se daña o se pierde, nos encontraríamos con un *bug* en el SO. Una solución parcial es usar listas doblemente enlazadas.

Una importante variación de *linked allocation* es el uso de una **file-allocation table** (**FAT**). Una porción del almacenamiento de cada *volumen* (partición) que tenemos en el disco se aparta para que contenga la tabla. La tabla contiene una entrada para cada bloque y es indexada por número de bloque. El directorio contiene el número de bloque del primer bloque del archivo. La tabla es indexada con el número de bloque que tenemos para saber

cuál es el próximo número. Esta cadena sigue hasta llegar al último bloque, el cual contiene un valor especial que dicta el final del archivo (**end-of-file** / **EOF**).
El esquema FAT puede resultar en un número significante de *head seeks* en el disco, a menos que la FAT se coloque en cache.

Indexed Allocation

Con la ausencia de la FAT, la *linked allocation* falla con los accesos directos, ya que un archivo contiene todos sus bloques distribuidos por todo el disco. La **asignación indexada** resuelve este problema al llevar todos los punteros a una misma locación: el **bloque de índices**.

Cada archivo tiene su propio bloque de índices, el cual consiste en un arreglo de direcciones de bloques de almacenamiento. La i-ésima entrada del bloque de índices nos dice en dónde se encuentra el i-ésimo bloque de un archivo. Para acceder el i-ésimo bloque, utilizamos el i-ésimo puntero de la tabla de índices.
Cuando un archivo es creado, todos los punteros de su tabla están seteados en NULL.

Sin embargo, una desventaja que nos encontramos con la asignación indexada es que, los gastos generales de los punteros en el bloque de índices es, generalmente, mayor a los gastos generales de la asignación enlazada. Gastamos bloques para colocarles punteros, para poder así indexar bloques más rápido. Aquí surge la pregunta de *qué tan grandes deberían ser los bloques de índices*. Queremos que sean pequeños, pero no tanto, ya que sino no podrían controlar archivos grandes.
Veamos entonces distintos esquemas posibles para responder esta pregunta:

- **Linked Scheme:** Podemos linkear juntos muchos bloques. Un bloque de índices podría contener un pequeño apartado con el nombre del archivo con el que se corresponde, seguido por las 100 direcciones de bloques del archivo. Luego, la siguiente dirección del bloque es NULL (para un archivo pequeño) o es un puntero a otro bloque de índices (para un archivo grande).
- **Multilevel Index:** Podemos usar el primer-nivel de un bloque de índices para que apunte a un segundo-nivel (otros bloques de índices). Luego, este segundo nivel apuntan a los bloques de los archivos.
  Para acceder a un bloque, el SO debe acceder al bloque de primer nivel para así acceder al segundo, y luego acceder al bloque deseado del archivo deseado.
  *En esencia, esto es el **double indirect block*** de los inodos.
- **Combined Scheme:** Una alternativa en los sistemas UNIX es mantener los primeros 15 punteros del bloque de índices en el **inodo** del archivo. (Un inodo es una estructura usada en *file systems* para almacenar información sobre un archivo o un directorio en un dispositivo de almacenamiento. Cada archivo o directorio está asociado a un inodo). Los primeros 12 de esos punteros apuntan a **bloques directos**; esto significa que contienen direcciones de los bloques del archivo. Los siguientes 3 punteros consisten en **bloques indirectos**.

El primero, apunta a un ***single indirect block***, el cual es un bloque de índices que contiene direcciones de bloques que sí tienen datos de archivos.
El segundo, apunta a un ***double indirect block***, que es una tabla que apunta a ***single indirect blocks***.
El tercero, apunta a un ***triple indirect block***, que es una tabla que apunta a ***double indirect blocks***.

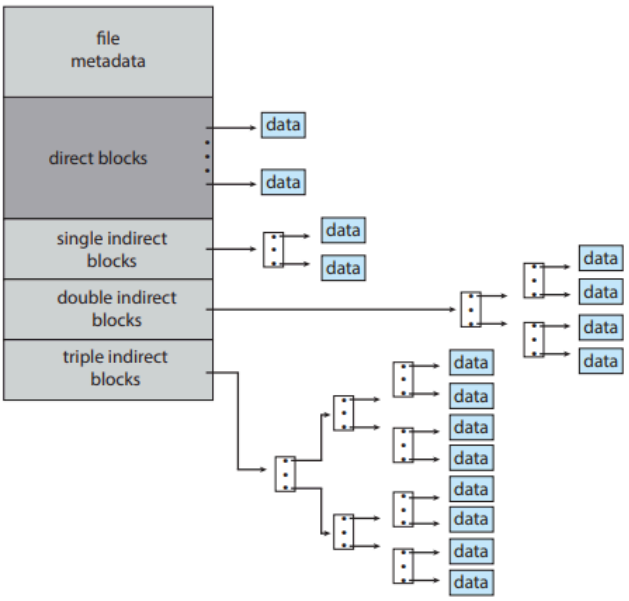


Figure 14.8 The UNIX inode.

Manejo del Espacio Libre

Como el espacio de almacenamiento es limitado, necesitamos reusar el espacio de los archivos eliminados para guardar archivos nuevos (si es posible). Vimos en la sección de *administración de memoria* que, para llevar un *tracking* del espacio libre, el sistema tiene una **lista de espacio libre**, la cual indica qué bloques del dispositivo se encuentran libres. La *lista de espacio libre* puede no ser representada como una lista, como veremos ahora.

- Bit Vector

Frecuentemente, la *lista de espacio libre* es implementada como un **bitmap** o un **bit vector**. Cada bloque es representado por un 0 o un 1. El 1 indica que el bloque está libre, listo para ser usado, mientras que el 0 indica que está ocupado. Si en un sistema tenemos que los bloques 2, 3 y 5 se encuentran libres, el *bit vector* se vería así:

001101…

La ventaja principal de este esquema es que es sencillo encontrar el primer bloque disponible, o los primeros *n* bloques consecutivos libres. Aún así, los *bit vector* son ineficientes a menos que la totalidad del vector se encuentre en memoria.

- Linked List

Otro esquema para la gestión del espacio libre son las listas enlazadas. La idea es enlazar a todos los bloques libres en memoria (con punteros), manteniendo un puntero al primer bloque

libre en una locación especial del *file system*. Este primer bloque libre contiene un puntero al siguiente, el siguiente tiene un puntero al que le sigue, y así.
Este método no es eficiente, ya que para atravesar la lista necesitamos leer cada bloque, lo cual requiere un tiempo substancial de E/S en discos HDD. Afortunadamente, atravesar la lista no es una acción frecuente. El método FAT incorpora un contador de bloques libres a su estructura de asignación.

- ## Grouping

Es una modificación de la *linked list*. Acá, el primer bloque contiene las direcciones de *n* bloques libres. Los primeros *n-1* bloques de estas direcciones están libres. La última dirección contiene las direcciones de otros *n* bloques libres, y así sucesivamente. Ahora, las direcciones de muchos bloques pueden ser encontradas más rápidamente.

- ## Counting

En vez de conservar una lista de *n* bloques libres, podemos conservar un puntero al primer bloque libre y un número (*n*) de los bloques contiguos que suceden al primer bloque. Cada entrada de la *lista de espacio libre* consiste entonces de una dirección y un contador. Estas entradas podrían estar almacenadas en un árbol balanceado (AVL), en vez de una lista; así, las operaciones de búsqueda, inserción y borrar son más eficientes.

- ## Space Maps

El *file system* ZFS (*Zettabyte File System*) fue diseñado para controlar grandes cantidades de archivos y directorios.
En la gestión de espacio libre, ZFS usa una combinación de diversas técnicas para controlar el tamaño de los datos, y minimizar la cantidad de E/S necesitada para manejar esas estructuras.
Primero, ZFS crea unos **metaslabs** para dividir el espacio del dispositivo en chunks de tamaños manejables. Una partición puede contener cientos de *metaslabs*. Cada *metaslab* tiene un **space map** asociado, es decir, un "historial" de la actividad de todos los bloques (asignación y liberación) en orden cronológico. Cuando ZFS decide asignar o liberar espacio de un *metaslab*, carga su *space map* asociado en memoria en forma de AVL. Además, ZFS condensa el mapa lo máximo posible, convirtiendo bloques contiguos en una única entrada. Finalmente, el *space map* es actualizado según corresponda, al igual que la *lista de espacio libre*. En esencia, el "historial" y el AVL son la lista libre.

- ## TRIMing Unused Blocks

Los dispositivos HDD y otros, permiten que sus bloques sean reescritos para actualizaciones. Lo único que necesitan es la *lista de espacio libre* para gestionar ese espacio.
Existen dispositivos de almacenamiento que no permiten reescrituras, tal es el caso de los dispositivos NVM. En estos aparatos, al borrar un archivo, su bloque no es liberado, sino que es colocado en una sección llamada **garbage collection**. Por ciertas características que tienen, la liberación de bloques debe ser en grandes cantidades (chunks). Entonces, se necesita de un

mecanismo que permita al *file system* informarle al dispositivo que una página está libre y puede ser considerada para ser borrada. Para eso es usado TRIM.

El libro no especifica mucho (al menos en el capítulo 14) sobre cómo funciona TRIM. Sólo nos comenta que, con este mecanismo, la *garbage collection* y los borrados pueden suceder antes de que el dispositivo NVM esté casi lleno.

Consistencia

Sea cual sea la causa de corrupción de archivos, un *file system* debe poder detectar los problemas y corregirlos. Para detectarlos, escanear toda la *metadata* de cada archivo del sistema puede confirmarnos o denegarnos la consistencia del mismo. Como este proceso puede llevar minutos u horas, alternativamente un *file system* puede guardar su estado entre la *metadata* del propio *file system*.
Se le puede agregar al *file system* un bit adicional. Al comienzo de cualquier cambio sobre la *metadata*, este bit de status es prendido, indicando que la *metadata* está en flujo. Al finalizar, el sistema apaga ese bit. Si ese bit se sigue manteniendo prendido, significa que algo sucedió, por lo que se debe correr un **consistency checker**. Para evitar que el sistema se clave mucho tiempo, se pueden realizar *soft updates*, es decir, se trata de rastrear las dependencias en los cambios de la *metadata* para grabar sólo cuando haga falta. Aún así, hace falta recorrer la lista de bloques libres. Una solución a esto, es **journaling**.

Journaling

Llamamos **journaling** (o **log-based transaction-oriented**) a los algoritmos que utilizan la idea de llevar un registro ("historial") para resolver el problema de la consistencia de los archivos.
Puede suceder el problema de que la inconsistencia de archivos sea irreparable. El chequeo de la consistencia puede no recuperar las estructuras dañadas, lo que resulta en pérdida de archivos, o incluso directorios.
La solución a este problema es aplicar técnicas basadas en registros a las actualizaciones de la *metadata* del *file system*. La idea es que todos los cambios sobre la *metadata* son escritos, secuencialmente, a un registro. A medida que el registro se llena, las actividades son reproducidas a través de todas las estructuras del *file system*. Mientras los cambios son realizados, un puntero es actualizado para indicar cuáles fueron terminadas y cuáles siguen incompletas. Cuando una *transacción* (una **transacción** es un conjunto de operaciones usadas para realizar cierta tarea) es completada, se genera una entrada en el registro indicandolo.
El registro de actividad es un **buffer circular**, es decir, una vez que llega a su fin, comienza a sobrescribir los datos que se encuentran al comienzo del registro. Son evitados los casos en donde se sobreescriben datos que aún no fueron guardados.

En resumen, cuando se realizan cambios sobre el *file system* (ya sea crear, borrar, o modificar archivos), estos cambios son escritos en un registro ANTES de *committearlos* a la estructura del *file system*. El *file system* lleva un *trackeo* de las operaciones que le quedan por realizar. Usualmente este registro es almacenado en una parte separada del disco principal.

Si el sistema se crashea, el registro puede contener cero o más *transacciones*. Cualquier *transacción* que ya haya sido *committeada* por el SO deben ser completadas. Si una *transacción* fue abortada (es decir, no fue *committeada* antes de que el sistema se crashee), cualquier cambio sobre el *file system* que esa *transacción* haya generado debe ser deshecho.

Snapshots

Un **snapshot** es una vista de todo el *file system* en un momento específico (antes de que cualquier actualización sobre el mismo tome lugar). En otras palabras, es una "foto" del disco. Los *snapshots* son muy útiles para realizar copias de seguridad y testing.

No confundir con los snapshots de Lógica y Computabilidad ahre. En LyC, se hacen sobre el estado de un programa, no sobre todo un FS. Las instantáneas presentadas en LyC son una especie de debugging.

El ***WAFL file system*** (**write-anywhere file layout**) utiliza snapshots para optimizar las escrituras aleatorias sobre el disco (WAFL es exclusivamente usado en *network file servers* producidos por NetApp). Entonces, el *WAFL file system* es un árbol de bloques de datos, en donde el inodo *root* es su base. Para tomar un *snapshot*, WAFL crea una copia del inodo *root*. Cualquier actualización que tengan los archivos o metadata del sistema irán a los nuevos bloques en vez de sobreescribir los bloques existentes. El nuevo inodo *root* apunta a la metadata y data cambiada como resultado de estas actualizaciones. Mientras tanto, el *snapshot* sigue apuntando a los bloques viejos, que aún no han sido modificados. Permite entonces así acceso al *file system* en la forma que tenía al instante de tomar la "foto".
En esencia, el espacio extra ocupado por un *snapshot* consiste únicamente en los bloques que fueron modificados luego de que el *snapshot* fuera tomado.
Versiones más actuales de WAFL permiten *snapshots* de lectura-escritura, conocidas como **clones**. Los *clones* usan la misma técnica que los *snapshots*. En este caso, un *snapshot* de solo-lectura captura todo el *file system*, y el clon se refiere de nuevo a esa captura. Cualquier escritura realizada sobre el clon es almacenada en nuevos bloques, y los punteros del clon son actualizados para que se refieran a estos nuevos bloques; los otros siguen intactos. Por el otro lado, el *snapshot* original sigue sin modificarse.
Los *clones* son útiles para testing y para actualizaciones: la versión original del sistema no se toca, y el clon es eliminado si el test finalizó, o si la actualización falló.

Montaje de File Systems

Al igual que los archivos tienen que ser abiertos antes de ser usados, un *file system* debe ser **montado** antes de encontrarse disponible para los procesos del sistema.
El proceso de montaje es directo. El SO recibe el nombre del dispositivo y un **punto de montaje**, esto es, la locación en la estructura de archivos en donde el *file system* debe ser adjunto. Típicamente, un *punto de montaje* es un directorio vacío. Luego, el SO verifica que el dispositivo tenga un *file system* válido. Esto lo hace al leer el directorio del dispositivo, chequeando que este tenga el formato esperado. Finalmente, el SO anota en la estructura de

su directorio que un *file system* fue montado en un punto específico de montaje. Así, el SO puede navegar el directorio.

Virtual File System

Los SO más modernos pueden soportar más de un tipo de *file systems* simultáneamente. Una forma de hacerlo es escribir rutinas de directorios y archivos para cada tipo, pero para poder simplificarlo, se usan técnicas de programación orientada a objetos.
Las estructuras de datos y procedimientos son usados para aislar el llamado de las funciones de sus implementaciones. Entonces, la implementación del *file system* consiste en 3 capas.

1) Interfaz del *file system*. Funciones como **open()**, **read()**, **write()** y **close()**.

2) La segunda capa se llama **Virtual File System** (**VFS**). Esta capa nos otorga dos importantes funciones:
   - Separa las operaciones genéricas del *file system* de sus implementaciones.
   - Provee un mecanismo para representar de forma única un archivo en una red. VFS está basado en una estructura de archivos, llamada **vnode**, que contiene un designador numérico para un archivo único en toda una red. Esta unicidad otorgada en toda una red es requerida para la mantención de los *network file systems*.

   Así, VFS distingue los archivos locales de los archivos remotos. Para los archivos locales, VFS activa operaciones específicas del *file system*, mientras que llama a los procedimientos del protocolo NFS (*network file system*) para los archivos remotos. En resumen, VFS es una capa de abstracción de los SSOO que provee una interfaz unificada para distintos tipos de *file systems* y dispositivos de almacenamiento.

3) La tercera capa de la arquitectura es la implementación del tipo de *file system* o el protocolo del *file system remoto*.

NFS

NFS es una implementación de un software de sistema para conectar archivos remotos a través de LANs (*Local-Area Network*–conecta computadoras entre cuartos, en una construcción o en un campus), e incluso de WANs (*Wide-Area Network*–conecta ciudades, construcciones o países).

Si se quiere acceder a un directorio remoto desde una máquina particular, un *cliente* de la máquina deberá primero llevar a cabo una operación de montaje. La idea es montar el *file system* remoto en algún punto del sistema local. Una vez que el montado haya terminado, los usuarios de la máquina pueden acceder al directorio remoto de una forma completamente transparente.

Un *file system* puede ser montado sobre otro *file system* que fue remotamente montado (que no es local).

Protocolo de Montaje

El protocolo de montaje establece la conexión lógica inicial entre un servidor y un cliente. Una operación de montaje incluye el nombre del directorio remoto que va a ser montado, y el nombre de la máquina servidor que lo almacenará. El servidor tiene una **export list** que especifica los archivos locales que exportará con el montaje, junto a los nombres de máquinas que tienen permitido montarlo.
Además, el servidor mantiene una lista de todos los clientes y directorios correspondientemente montados. Esto se hace por fines administrativos, por ejemplo, avisar que el servidor será apagado por algún motivo.

Protocolo NFS

NFS se encuentra integrado en el SO vía VFS. El cliente inicia la operación sobre un archivo con una *syscall* regular. El SO *mapea* esta llamada a una operación VFS en el *vnode* apropiado. Luego, la capa VFS identifica el archivo como remoto o local. Vimos anteriormente que, si un archivo es local, VFS procede de cierta manera, mientras que si lo identifica como remoto, lo hace de otra.

NFS accede a *file systems* remotos a través de RPC (*remote procedure calls*). Como NFS utiliza RPC, las operaciones remotas que se hacen sobre los archivos remotos pueden ser traducidas directamente a su correspondiente RPC.

9) Sistemas Distribuidos

NOTA: Antes de comenzar, me parece importante aclarar que nada de lo que viene a continuación lo leí del Silberschatz. La información es puramente de la clase teórica de Baader. Además, parte de los temas fueron expandidos con el libro de Nancy A. Lynch – "Distributed Algorithms". Las slides teóricas tienen algunas secciones mencionadas, únicamente me centré en leer esas secciones del libro.

Parte 1

Los **sistemas distribuidos** son un conjunto de recursos conectados que interactúan entre sí. Pueden ser varias máquinas conectadas en red, un procesador con varias memorias, varios procesadores que comparten varias memorias, entre otros.

Fortalezas:

- Paralelismo
- Replicación
- Descentralización

Debilidades

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

Hay sistemas distribuidos con y sin memoria compartida. Para los casos de sistemas distribuidos sin memoria compartida, tenemos como ejemplos *telnet* y *RPC* (*remote procedure calls*). *Telnet* es un protocolo que permite que una máquina se conecte con otra. Esa otra máquina tiene los recursos necesarios para cierta parte de un procesamiento, entonces allí (en la máquina con los recursos) un equipo realiza el trabajo, mientras que el otro solamente corre un programa interactivo de comunicaciones. Luego, sobre RPC, no hay mucho para explicar, ya se habló y mencionó en el capítulo anterior.

Vemos entonces que lo que tienen en común estos mecanismos **sincrónicos** es que la cooperación tiene la forma de solicitarle servicios a otros. Esos otros equipos no tienen un rol activo. A este tipo de arquitecturas de software las llamamos *cliente/servidor*: el programa principal "hace" de cliente, mientras que el servidor da los servicios cuando el cliente los pide.

Un mecanismo **asincrónico** es el *pasaje de mensajes* (alguna vez en un capítulo hablamos sobre *buzones* y *pipes*). Este es el mecanismo más general, no hay nada compartido, únicamente un canal de comunicación.
Como la comunicación es asincrónica, tenemos que dejar de procesar para atender el traspaso de un mensaje. Además, hay que manejar la codificación y decodificación de los datos.

- **Teorema CAP**
  También conocido como *conjetura de Brewer*: en un entorno distribuido no se puede tener a la vez *consistencia*, *disponibilidad* y *tolerancia a fallas*. Solo dos de esas tres.

Locks en entornos distribuidos

En entornos distribuidos no hay *TestAndSet* atómico, entonces se debe buscar una alternativa para la implementación de *locks*.
Una de las más sencillas consiste en poner el control de los recursos bajo un único nodo, que hace de coordinador. Se "piensa" que dentro de ese nodo hay procesos que ofician de representantes (o *proxies*) de los procesos remotos. Cuando un proceso necesita de un recurso, se lo pide a su *proxy*, el cual lo "negocia" con el resto de los *proxies* usando los mecanismos que vimos antes.
Aún así, este enfoque tiene muchos problemas. Tenemos un único nodo del que todo depende; si este falla, cagamos la fruta. Además, tenemos un cuello de botella en procesamiento y capacidad de red. Por si fuera poco, la interacción con el coordinador requiere de mensajes que viajan por la red, que son lentos. Por último, se requiere consultar al coordinador, que puede estar lejos, incluso para acceder a recursos cercanos.

Una alternativa a esta implementación es hacer que el recurso lo posea el equipo que primero lo pide. Pero estamos en un entorno distribuido, entonces… ¿qué significa pedirlo primero? Puede significar que se refiere al que pidió primero el mensaje, o al que logró que su mensaje llegue antes a todos los equipos. En caso de un empate, habría que ver los *timestamps* de los relojes de cada equipo.
En sistemas distribuidos, si la precisión no importa, cada uno puede consultar su reloj y decidir qué pasó antes de qué. El problema aparece si necesitamos de una precisión de milésimas de segundos, y para los eventos que generan las computadoras sí la necesitamos. (Sabemos que por complicaciones de representación decimal, la precisión que requerimos es jodida). Se puede intentar sincronizar los relojes, pero es muy difícil y no es necesario.

Leslie Lamport (el que inventó LaTex) se dio cuenta de que lo único importante era saber si algo había ocurrido antes o después de otra cosa, pero no exactamente cuándo. Su propuesta es definir un ***orden parcial no reflexivo*** entre los eventos de la siguiente manera:

- Si dentro de un proceso, A sucede antes que B, entonces $A \rightarrow B$
- Si E es el envío de mensajes y R su recepción, $E \rightarrow R$
- Si $A \rightarrow B$ y $B \rightarrow C$, entonces $A \rightarrow C$
- Si no vale ni $A \rightarrow B$ ni $B \rightarrow A$, entonces A y B son concurrentes

La implementación es la siguiente: cada procesador tiene un reloj propio (*puede ser real, pero alcanza con que tenga un valor monótonamente creciente con cada lectura*). Cada mensaje lleva adosada la lectura del reloj. Como la recepción siempre es posterior al envío, cuando se recibe un mensaje con una marca de tiempo t que es **mayor** al valor actual del reloj, se actualiza el reloj interno a $t + 1$. En caso contrario, tomamos el valor de nuestro reloj y le sumamos 1.
Esta implementación genera un orden parcial, ya que no indicamos ningún mecanismo en el caso de tener un empate de tiempos. En el caso de que exista, estaríamos teniendo un orden total. Los empates vienen dados por eventos concurrentes, entonces se pueden ordenar arbitrariamente (por el PID, por ejemplo).
Este algoritmo es denominado como el **algoritmo de Lamport** (o **tiempos lógicos de Lamport**).

Acuerdo Bizantino

Supongamos que las distintas divisiones del ejército bizantino rodean una ciudad, desde distintas comarcas. Solo pueden ganar si atacan todos juntos, así que deben coordinar su ataque. La única forma de comunicarse es con mensajeros, que corren de un lugar a otro, y pueden ser interceptados.

Imaginemos sólo A y B. A decide la hora de ataque y envía un mensajero a B. B recibe este mensaje, y envía otro mensajero a A proponiendo otra hora. Supongamos que el mensajero no llega… ¿qué hace A? Supongamos que sí llega… ¿cómo sabe B que A sabe?

Teorema. Se puede resolver el consenso bizantino para *n* procesos y *k* fallas si y solo si *n > 3k* y la conectividad es mayor a *2k.*

Clusters

En el sentido científico, un **cluster** es un conjunto de computadoras conectadas por una red de alta velocidad, con un *scheduler* de trabajos en común.
En el resto, es un conjunto de computadoras que trabajan cooperativamente desde alguna perspectiva. A veces para proveer servicios relacionados, o a veces para proveer el mismo de manera redundante.

Grids: Conjunto de *clusters*, cada uno bajo un dominio administrativo distinto.
Clouds: *Clusters* donde uno puede alquilar una capacidad fija o bajo demanda.

Scheduling en Sistemas Distribuidos

Tenemos dos niveles:

1) *Local*: darle el procesador a un proceso listo
2) *Global:* asignarle un proceso a un procesador (*mapping*)

En el nivel **global**, se debe compartir la carga entre los procesadores.

- *Global Static Scheduling*: Se asignan los procesos en el momento de la creación de los mismos. (***affinity***)
- *Global Dynamic Scheduling:* La asignación de los procesos varía durante la ejecución. (***migration***)

Política de scheduling:

- *Transferencia:* **cuándo** hay que migrar un proceso
- *Selección:* **qué** proceso hay que migrar
- *Ubicación:* **a dónde** hay que enviar el proceso
- *Información:* **cómo** se difunde el estado

// *No se entiende muy bien lo que explican las slides de Baader, son una poronga. No pienso leer el libro si no me lo pide.*

Parte 2

Algoritmos sobre Sistemas Distribuidos

Cuando se trabaja con algoritmos distribuidos, es importante determinar el modelo de fallas. Algunas alternativas combinables son:

- Nadie falla
- Los procesos caen y no se levantan
- Los procesos caen y pueden levantar
- Los procesos caen y pueden levantar únicamente en determinados momentos
- La red se particiona

- Los procesos pueden comportarse de manera impredecible (fallas bizantinas)

Nosotros casi exclusivamente nos vamos a centrar en los modelos sin fallas.

Para medir la complejidad en este tipo de algoritmos, tiene sentido tener en cuenta la cantidad de mensajes enviados a través de la red. Aún así, hay redes de altísima velocidad que tienen otros cuellos de botella. Otra métrica entonces a tener en cuenta es qué tipos de falla soportan.
Otra forma de evaluarlos es cuánta información necesitan: el tamaño de la red, la cantidad de subprocesos, o cómo ubicar cada uno de ellos.

Exclusión Mutua Distribuida

La forma más sencilla se llama ***token passing***. Consiste en armar un anillo lógico entre los procesos y poner a circular el *token*. Cuando un proceso quiere entrar a la sección crítica, espera a que le llegue el *token*. Notemos que si no tenemos fallas, no hay inanición.
El problema es que siempre que un proceso quiere entrar a CRIT, envía mensajes a todos los procesos, entonces, tenemos mensajes innecesarios circulando en la red.

Veamos otro enfoque.
Cuando quiero entrar a la sección crítica, envío a todos (incluyéndome) un mensaje solicitud(P_i, ts) siendo *ts* el *timestamp*. Cada proceso puede responder este mensaje inmediatamente o encolar la respuesta. Podemos entrar a CRIT cuando hayamos recibido todas las respuestas. Si entramos, al salir, respondemos todos los pedidos demorados.
Únicamente respondemos inmediatamente sí:

- No quiero entrar a CRIT
- Quiero entrar, aún no lo hice, y el *timestamp* que recibo es menor al mío. Como es menor, el otro tiene prioridad.

Lo bueno de este enfoque es que no circulan mensajes si no se quiere entrar a la sección crítica. Este algoritmo requiere que todos los procesos sepan de la existencia de todos.

Eso sí, lo que necesitamos para poder implementar cualquiera de estos dos algoritmos es:

1) Que no se pierda ningún mensaje
2) Que ningún proceso falle

Locks Distribuidos

Ya hablamos de esto en la parte anterior, sobre una versión con un coordinador de *locks* centralizado. Vamos a hablar ahora sobre una alternativa completamente distribuida: el **protocolo de mayoría**.

La idea es que queremos obtener un *lock* sobre un objeto del cual hay copia en *n* lugares. Para obtener un *lock*, debemos pedirlo a, por lo menos, (n/2) + 1 sitios. (osea, digamos, la mitad más uno (no soy bostero ni de derecha)). Cada sitio puede responder si puede o no dárnoslo.

Cada copia del objeto tiene un número de versión. Si lo escribimos, tomamos el más alto y lo incrementamos en uno. No puede suceder que se otorguen dos *locks* a la vez, ya que para que eso suceda, dos procesos deberían tener más de la mitad de los *locks*, lo cual es imposible.
En resumen, le estamos pidiendo literalmente la copia del objeto a cada sitio.
Es imposible también que leamos una copia desactualizada, puesto que cualquier proceso escribe sobre más de la mitad de los *locks*, y para que nosotros leamos una versión desactualizada, tuvo que haber habido un proceso que escriba sobre menos de la mitad de las copias.

Hay que tener mucho ojo con este algoritmo, ya que pueden producirse *deadlocks*, y hay que usar algoritmos de detección.

Elección de Líder

Una serie de procesos debe elegir a uno como **líder** para algún tipo de tarea. En una red sin fallas, es sencillo. Podemos mantener un *status* que dice que no soy el líder. Luego, organizamos los procesos en forma de anillo y hacemos circular los IDs de cada uno. Cuando recibo un mensaje, comparo mi ID con el que circula. Si el ID recibido es más grande que el mío, lo hago circular; si no, descartamos el ID recibido. Cuando un mensaje dio toda la vuelta, ya sabemos quién es el líder. Por último, se pone a girar un mensaje en forma de notificación para que todos sepan quién es.
Este algoritmo es conocido como ***LCR algorithm*** (*Le Lann, Chang, Roberts*).

La notificación que se envía al final es para que los procesos ***halteen*** (se detengan). En la versión original del algoritmo, el *status* inicial de cada proceso es *unknown*. Cuando un nodo con ID *u* recibe, de otro proceso, el ID *u*, puede cambiar su estado a *líder*, y luego enviar una notificación al nodo siguiente. La notificación hará que el proceso siguiente cambie su *status* a *non-leader*, enviará la misma notificación al proceso siguiente, y luego se **detendrá**.

La complejidad de este algoritmo, sin fase de *halt* es de O(*n*), mientras que si el algoritmo tiene fase de *halt*, su complejidad es O(2*n*).
Para ambas versiones, la complejidad de la comunicación es O(*n*^2).

Existe una versión de este algoritmo con complejidad O(*n* log *n*). En esta versión, se permite la comunicación bidireccional. La idea es la siguiente: cada proceso *i* envía su ID en ambas direcciones en cierto rango, el cual incrementará con cada iteración. Sea *l* el número de ronda (*l* es 0, 1, 2…), en cada "iteración" se enviará el ID de *i* en un rango de 2^*l*. En cada fase, lo ideal es que el ID viaje a través de cada nodo, y luego vuelva a su proceso inicial. Si ambos *token* IDs enviados (se envían 2 porque la comunicación es bidireccional) vuelven a salvo, se repite el proceso con un rango mayor. Caso contrario, no lo vuelve a hacer. La única forma de que un *token* ID no regrese a su proceso inicial, es que en el camino se haya encontrado con un proceso con un ID mayor. Recordemos que, en el algoritmo LCR, si un nodo con ID *j* recibía un *token* ID *i* tal que *j* > *i*, entonces el nodo *j* descartaba el ID *i* recibido y no lo propagaba por el resto de la red.

Este algoritmo es conocido como ***HS algorithm*** *(Hirschberg* y *Sinclair*).

La complejidad de este algoritmo (a nivel comunicación, es decir, cantidad de mensajes) es O(*n* log *n*). Luego, la complejidad temporal es O(*n*).

Existe una cota inferior para este problema, la cual es Ω(*n* log *n*). Aún así, esta cota sólo es válida para los algoritmos de elección de líder que utilizan comparaciones. Hay algoritmos de complejidad (comunicacional) O(*n*), que no usan comparaciones. Sus nombres son *TimeSlice algorithm* y *VariableSpeeds algorithm*. Estos algoritmos permiten que los UIDs de cada proceso sean enteros positivos, y que puedan ser manipulados por operaciones aritméticas.

- ***TimeSlice Algorithm***
  Este algoritmo considera que todos los UIDs son positivos, y que el tamaño *n* del anillo es conocido por todos los nodos pertenecientes al mismo. *TimeSlice* consiste en fases (1, 2, 3…), en donde cada fase tiene *n* rondas consecutivas.
  Sea *v* la fase actual, hay un único *token* circulando el anillo con UID *v*. Si existe un proceso *i* con UID *v*, entonces ese proceso se elige a sí mismo como lider y envía un *token* con su UID al resto de los procesos.
  Este algoritmo nos da una complejidad comunicacional de O(*n*), pero desafortunadamente nos da una complejidad temporal de O(*n* u_{min}), ya que hay que esperar u_{min} fases hasta elegir un líder, lo cual es un número incierto.

- ***VariableSpeeds Algorithm***
  Este algoritmo no sirve en la práctica, ya que tiene una complejidad temporal peor que la del algoritmo anterior, la cual es O(*n* 2^u_{min}). La "ventaja" de este algoritmo es que podemos aplicarlo en un entorno en el que NO todos los procesos conozcan el tamaño *n* del anillo.
  Otra vez, el algoritmo consiste en rondas. Cada proceso *i* inicia un *token* que viaja alrededor del anillo, llevando el UID del proceso *i*. Distintas *tokens* viajan a distintos ritmos. En particular, un *token* con UID *v* transmite un mensaje cada 2^*v* rondas. Cada proceso se queda con el UID más chico que haya encontrado. Si un *token* vuelve a su creador, entonces el creador es el líder.

Snapshot global consistente

Sea $E = \Sigma E_i$, siendo E_i la parte del estado que le corresponde a P_i. Lo único que modifica al estado son los mensajes que se envían los procesos entre sí. Si queremos tomar un *snapshot* consistente de E, significa que, en un momento dado (a partir de que hacemos el pedido), queremos ver cuánto valían los E_i y qué mensajes estaban circulando en la red.

Cuando se quiere un *snapshot*, un proceso se envía a sí mismo un mensaje de **marca**.
Cuando P_i recibe un mensaje de marca por primera vez, guarda una copia C_i de E_i, y envía

un mensaje de marca a todos los otros procesos. En ese momento, P_i empieza a registrar todos los mensajes que recibe de cada vecino P_j, hasta que recibe marca de todos ellos. Luego, queda conformada la secuencia $Recibidos_{i,j}$ de todos los mensajes que recibió P_i de P_j antes de que éste tomara la instantánea.
El **estado global** es que cada proceso está en el estado C_i, y los mensajes que están en *Recibidos* están circulando en la red.

Estos *snapshots* globales pueden ser usados para detectar *deadlocks*, para un *debugging* distribuido, detección de terminación y detección de propiedades estables (es decir, propiedades que, una vez son verdaderas, luego lo siguen siendo).

2PC (*Two-Phase Commit*)

La idea es realizar una transacción de manera atómica. Todos deben estar de acuerdo en que se hizo o no se hizo.

El espíritu es que, en una primera fase, se le pregunta a todos los nodos si estamos de acuerdo en que se haga la transacción. Si recibimos un *no*, abortamos. Si recibimos un *sí*, vamos anotando de quienes provienen. Si pasado un tiempo máximo no todos responden, también se aborta. Si recibimos todos los *sí*, se le avisa a todos los nodos que se aceptó la transacción (segunda fase).
Veamos lo que realmente pasa en el algoritmo. Todos los procesos, exceptuando el proceso 1, le envían sus valores iniciales al proceso 1, y cualquiera cuyo valor inicial fuera 0, decide 0. El proceso 1 colecta todos estos valores (y el suyo también) y los coloca en un vector. Si el vector está lleno de 1s, entonces el proceso 1 decide 1. Caso contrario, el proceso 1 decide 0. En la segunda fase, el proceso 1 propaga su decisión por todo el sistema.

2PC resuelve **COMMIT** con *terminación débil*. (*Terminación débil es que, si no hay fallas, todo proceso decide*). El problema es que no resuelve **COMMIT** con *terminación fuerte*. (Esto es, que todo proceso que no falla, decide).
Veamos porqué 2PC no resuelve COMMIT con *terminación fuerte*: si el proceso 1 falla antes de propagar la decisión al comienzo de la fase 2, entonces ningún proceso que NO falló y decidió 1 forma parte de la decisión final. Usualmente, si el proceso 1 falla, los procesos llevan a cabo un protocolo de terminación y a veces se las arreglan para decidir.

3PC (*Three-Phase Commit*)

Este es un algoritmo que resuelve COMMIT con *terminación fuerte*. Estos tipo de algoritmos también son conocidos como *non-blocking algorithms*.

La clave es simple: el proceso 1 no decide 1 hasta que todos los procesos que aún no fallaron están “listos” para decidir 1. Para asegurarnos de que los procesos están “listos”, requerimos de una fase extra.
Analicemos cada fase del algoritmo.

- **Ronda 1:** Igual a la de 2PC, con la diferencia de que, si el vector del proceso 1 está lleno de 1s, entonces el proceso 1 se vuelve *listo (ready)*, pero todavía no decide. Caso contrario, si hay al menos un 0 en el vector, el proceso 1 decide 0.
- **Ronda 2:** Si el proceso 1 decidió 0, entonces propaga **abortar** (*decide(0)*) por toda la red. Cualquier proceso que recibe *decide(0)* decide 0. Si no, entonces el proceso 1 propaga ***ready***. Cualquier proceso que recibe *ready* se vuelve *ready*. El proceso 1 decide 1 si aún no decidió.
- **Ronda 3:** Si el proceso 1 decide 1, propaga **commit** (*decide(1)*) por toda la red. Cualquier proceso que reciba *decide(1)* decide 1.

Estas 3 rondas no alcanzan para asegurar que 3PC es un algoritmo no-bloqueante. Si el proceso 1 falla, puede dejar en un estado de *indecisión* al resto de los procesos. Para asegurar que el algoritmo sea no-bloqueante, los procesos restantes deben ejecutar un *protocolo de terminación* luego de las primeras 3 rondas.

- **Ronda 4:** Los procesos que no hayan fallado le envían su status al proceso 2, ya sea que decidieron 0, 1, *ready* o *indeciso*. El proceso 2 toma todos los status recibidos (y el suyo también) y lo coloca en un vector. Si en el vector hay un 0 (y el 2 no decidió nada), entonces el proceso 2 decide 0. Lo mismo sucede si el vector está lleno de 1s. Si todos los estados son *indeciso*, entonces el proceso 2 decide 0. Si los estados son *indeciso*, pero hay al menos un estado *ready*, entonces el proceso 2 decide *ready*.
- **Ronda 5:** Si el proceso 2 decidió, anuncia su decisión por toda la red. Caso contrario, comunica *ready*. Si el proceso 2 aún no decidió (está en estado *ready*), decide 1.
- **Ronda 6:** Si el proceso 2 decidió 1, lo comunica por toda la red. Cualquier proceso que reciba *decide(1)*, decide 1.

Los algoritmos que resuelven COMMIT tienen una cota inferior en la cantidad de mensajes. No importa que el algoritmo sea bloqueante (*terminación débil*), si resolvemos COMMIT en una red sin ninguna falla, la cantidad mínima de mensajes que deben ser enviados es $2n - 2$.

Parte 3

File Systems Distribuidos

En esta parte nos centramos en estudiar *Distributed File Systems* (DFS). DFS es un sistema de archivos cuyos clientes, servidores y dispositivos de almacenamiento están distribuidos entre las máquinas de un sistema distribuido. Al cliente se le debe presentar como un sistema de archivos centralizado convencional.
La característica principal distintiva es la administración de dispositivos de almacenamiento dispersos.

Modelo cliente-servidor

El servidor almacena tanto los archivos como su metadata en almacenamiento conectado al servidor. Los clientes contactan al servidor para pedirle archivos, mientras que el servidor es el responsable de la autenticación, el chequeo de permisos, y el envío del archivo. Además,

los cambios que el cliente realiza sobre el archivo deben ser propagados al servidor. Un ejemplo de este modelo que vimos antes es NFS.

El único punto de falla de este diseño es el servidor, puesto que puede caerse.
El servidor es un cuello de botella para todos los pedidos de datos y metadatos. Esto puede traer problemas de escalabilidad y ancho de banda.

Modelo basado en cluster

Está diseñado para ser más resistente a fallas y escalable que el modelo cliente-servidor.
Los clientes se conectan al servidor de metadata, y hay varios servidores de datos que contienen *chunks* de archivos. El servidor de metadata mantiene un mapeo de qué servidores de datos tienen *chunks* de cada archivo.
Los *chunks* de cada archivo se replican *n* veces.

Un ejemplo de este modelo es *Google File System* (***GFS***), y *Hadoop File System* (*HDFS*)
El diseño de GFS fue influenciado por las siguientes observaciones:

- La falla en componentes de hardware es la norma más que la excepción. Debe ser esperada en forma rutinaria.
- Los archivos en ciertos sistemas son muy grandes.
- Muchos archivos son modificados mediante el agregado de nueva información al final del mismo, en lugar de sobreescribir datos existentes.
- Se pueden rediseñar las aplicaciones y API de *file system* para incrementar la flexibilidad del sistema.

Los desafíos de DFS incluyen:

- Nomenclatura y transparencia.
- Acceso a archivos remotos.
- Cachés y consistencia.

Nomenclatura: Mapear entre objetos lógicos y físicos.
Mapeo multinivel: Abstracción de un archivo que oculta los detalles de cómo y dónde en el disco está almacenado el archivo.

Un DFS transparente oculta la ubicación en la que se almacena el archivo en la red.
Para un archivo replicado en varios sitios, el mapeo devuelve un conjunto de las ubicaciones de las réplicas del mismo. Tanto la existencia como la ubicación de múltiples copias se encuentran ocultas.

Transparencia de la ubicación: El nombre del archivo no revela la ubicación física del mismo.
Independencia de la ubicación: El nombre del archivo no cambia cuando su ubicación física cambia.

Esquema de Nombres

Tenemos tres enfoques.

- Los archivos se nombran combinando el nombre del *host* con el nombre local. Garantiza un nombre único en todo el sistema, pero no es independiente ni transparente de ubicación.
- Montar directorios remotos en directorios locales. Sólo los directorios remotos previamente montados pueden ser accedidos de manera transparente.
- Única estructura global de nombre abarca a todos los archivos del sistema. Si un servidor no está disponible, un conjunto arbitrario de directorios en distintas máquinas también está indisponible.

Caché

Si un usuario necesita acceder a un archivo remoto, el servidor que aloja el archivo fue ubicado por el esquema de nombres, y debe hacer la transferencia de datos. Como ya explicamos, los pedidos de acceso son enviados al servidor, el servidor realiza el acceso, y el resultado es devuelto al usuario. Uno de los mecanismos más comunes para implementar el servicio remoto es con el paradigma RPC.

Para reducir el tráfico de red, se pueden retener los bloques de disco recientemente accedidos en caché, para que repetidos accesos a la misma información puedan ser realizados localmente.
Si la información necesaria no está en caché, una copia es traída del servidor al usuario. La copia maestra del archivo reside en el servidor, pero copias (o partes) del archivo están distribuidas en distintos cachés.

Tenemos un problema, puesto que se deben mantener las copias en caché consistentes con el master del archivo. Es por eso que se deben considerar las políticas de actualización de caché:

- **Write-through:** Se escribe la información a disco ni bien se modifica el caché.
  Este mecanismo es confiable, pero poco performante.
- **Delayed-write (write-back):** Las modificaciones se guardan en caché, y son escritas más tarde al servidor.
  La escritura termina rápido. Nos ahorramos escrituras al servidor. Es menos confiable, ya que los datos no escritos se perderán si la máquina del usuario se cuelga. Una variante es recorrer el caché en intervalos regulares, y actualizar la información del servidor con los bloques que se modificaron desde la última recorrida. Otra opción es **write-on-close**, la cual escribe los datos en el servidor cuando el archivo es cerrado.

Veamos cómo se hace para mantener la consistencia en modelos cliente-servidor.
Enfoque iniciado por el cliente:

- El cliente inicia un chequeo de validez.
- El servidor chequea si los datos locales coinciden con la copia maestra.

Enfoque iniciado por el servidor:

- El servidor registra, para cada cliente, los archivos que cachea.
- Cuando el servidor detecta una potencial inconsistencia, reacciona.

Las diapositivas de Baader son bastante pobretonas. No aclara cómo reacciona el servidor, ni que hace luego de eso.

10) Seguridad

La seguridad y la protección son vitales para las computadoras. La seguridad es una medida de confianza en que la integridad del sistema y sus datos van a ser preservados. La protección es el conjunto de mecanismos que controlan el acceso de los procesos y usuarios a los recursos de la computadora.
Decimos que un sistema es **seguro** si todos sus recursos son usados y accedidos de forma correcta y deseada, en cualquier circunstancia. Desafortunadamente, la seguridad total no puede ser lograda.

- **Infracción de confidencialidad:** Este tipo de violación incluye la lectura no autorizada de datos (o robo de información).
- **Infracción de integridad:** Este tipo de violación incluye la modificación no autorizada de datos.
- **Infracción de disponibilidad:** Este tipo de violación incluye la destrucción/eliminación no autorizada de datos.
- **Robo de servicio:** Este tipo de violación incluye el uso no autorizado de recursos.
- **Negación de servicio:** Conocido en inglés como *Denial of Service* (DoS). Este tipo de violación no permite el uso correcto del sistema a usuarios legítimos.

Malware

Los *malwares* (*malicious softwares*) son softwares diseñados para dañar, deshabilitar o aprovechar sistemas informáticos.

Muchos sistemas tienen mecanismos que permiten que programas, escritos por un usuario, sean ejecutados por otros usuarios. Si estos programas son ejecutados en un dominio que le permite al usuario externo acceder a datos no autorizados, estamos en problemas. A los programas que actúan de forma maliciosa, y no de la manera que dicen hacerlo, los llamamos **troyanos** (**trojan horse**, en inglés).
Una variación de los troyanos son las **trojan mules**, programas que simulan ser programas de *logueo / inicios de sesión*.

Otra variación de los troyanos son los **spyware**. Los *spywares* a veces vienen acompañados con programas que el usuario instala en su computadora. Se dedican a colocar publicidades

en el sistema; abren ventanas cuando ciertos sitios son visitados, o a veces recopilan información del usuario.

También existen los **ransomwares**. Son *malwares* que se dedican a encriptar toda la información del sistema en el que se encuentra. La información no le es útil a la persona que la encripta, sino al usuario de la computadora. La idea es forzar al usuario a pagar un rescate (de ahí, *ransom*) para recuperar su información.

Otra forma de *malware* son las **trap doors** (o **back doors**). Son agujeros en el software que deja una persona que diseñó cierto programa o sistema; solo esa persona puede acceder a esa "puerta". Son usadas, en parte, para **bombas lógicas**. Hay casos también en donde fueron usadas para robarle dinero a bancos, colocando errores de redondeo en códigos de transferencia.
Las *trap doors* suponen un problema, ya que para detectarlas, habría que leer todo el código fuente de todos los componentes del sistema. Obviamente esa actividad no la hace nadie (o casi nadie). Una metodología al desarrollar software que se emplea es **revisar código**. En estas revisiones, la persona que escribió el programa lo envía a una base, en donde una o más personas leen el código para ver si es aprobado o no.

Code Injection

La mayoría de los softwares no son maliciosos, pero pueden poseer ciertas amenazas hacia el sistema debido a un ataque de inyección de código. Estos ataques se basan en modificar (o agregar) código ejecutable al sistema. Casi siempre, este tipo de ataques son un resultado de pobres o inseguros paradigmas de programación. Comúnmente sucede en lenguajes de bajo nivel, como C o C++, en los cuales podemos acceder a memoria de manera sencilla a través de punteros.

La forma más zonza de sufrir una inyección de código es con un *buffer overflow*. Se sobreescriben el *stack frame*, y el curso del programa que estaba corriendo cambia.
Si sucede que el *overflow* es muy pequeño, puede pasar desapercibido. Esto se debe a que la asignación de BUFFER_SIZE bytes van a tener *padding*, dependiendo de la arquitectura del sistema. Si el *overflow* excede el *padding*, la siguiente (o siguientes) variables del *stack* van a ser sobreescritas con el contenido del *overflow*. Si el *overflow* excede exageradamente el *padding*, entonces sonamos la flauta.
Aún así, hay muchas formas sencillas de prevenir *buffers overflows*.

Virus y Gusanos

Otra forma que toman los programas amenazantes son los **virus**. Son fragmentos de código embebidos en programas legítimos. Los virus son autorreplicantes, y están diseñados para infectar otros programas. Están limitados a las arquitecturas, sistemas operativos, y aplicaciones. UNIX y otros SSOO multiusuarios no son susceptibles a los virus, ya que los programas ejecutables están protegidos de ser escritos por el SO.

Todo muy lindo (o feo) pero… ¿cómo funcionan los virus? Una vez que un virus llega a una computadora, un programa, conocido como **virus dropper**, inserta el virus en el sistema. Los **virus dropper** son, usualmente, *troyanos*.
Hay millones de virus, pero todos recaen en diversas categorías: **file virus**, **boot virus**, **macro virus** (virus escritos en lenguajes de alto nivel), **rootkit virus** (comprometen todo el sistema, tienen fácil acceso a *root*), **source code virus**, **polymorphic virus** (el virus cambia constantemente para no ser detectado por el antivirus. No cambia la funcionalidad del virus), **encrypted virus**, **stealth virus**, **multipartite virus**, y **armored virus**.

Una distinción que puede realizarse de los virus son los **gusanos** (**worms**), los cuales usan una red para replicarse, sin usar ningún tipo de ayuda de los humanos. (*Lo último se debe a que, para que un virus entre a nuestra PC, si o si tenemos que descargar un programa, o usar un disco infectado, entre otras muchas posibilidades.*)

Criptografía como herramienta de seguridad

Es generalmente considerado inviable construir una red, de cualquier escala, en donde las direcciones fuente y de destino puedan ser completamente **confiables**. Entonces, la única alternativa es, de alguna manera, eliminar la necesidad de confiar en la red. Este es el trabajo de la criptografía. La **criptografía** es usada para restringir a los emisores y/o receptores de un mensaje.
La criptografía moderna está basada en **claves** que están selectivamente distribuídas en distintas computadoras en red, y son usadas para procesar mensajes. Al recibir un mensaje, la criptografía nos permite verificar si el mensaje fue creado por alguna computadora con cierta clave. Similarmente, un emisor puede encriptar un mensaje, de forma tal que solo una computadora con cierta clave pueda desencriptarlo.

Encriptación simétrica

En el algoritmo de encriptación simétrica, la misma **clave** es usada para encriptar y desencriptar mensajes. Las personas que se envían y reciben mensajes deben **realizar un intercambio de claves**, para poder así leer los mensajes que reciben.
El algoritmo de encriptación simétrico más usado es **data-encryption standard** (**DES**). DES toma un valor de 64 bits, y una clave de 56 bits, y realiza una serie de transformaciones basadas en permutaciones y sustituciones para cifrar mensajes. Aún así, DES es considerado inseguro para muchas aplicaciones, ya que sus claves pueden ser exhaustivamente buscadas (y encontradas) con medianos recursos computacionales.
Hay una variación de DES llamada **triple DES**, en la que el algoritmo DES se repite 3 veces en el mismo texto, usando dos o tres claves. Usando tres claves, la longitud de la clave es de 168 bits.

Un "nuevo" cifrado de bloques que reemplazó a DES es **advanced encryption standard** (**AED**).Usa claves con una longitud de 128, 192 o 256 bits. Puede trabajar con bloques de datos de hasta 128 bits.

Generalmente, los cifrados de bloques no son esquemas de encriptación muy segura. En particular, no pueden manejar mensajes más largos que sus tamaños de bloques requeridos.

Encriptación asimétrica

Como era de esperar, en un algoritmo de encriptación asimétrica, se usan distintas claves para encriptar y desencriptar mensajes. Una entidad, preparada para recibir mensajes encriptados, crea dos claves y hace que una de ellas (llamada **public key**, o **clave pública**) esté disponible para cualquier usuario que la desee. Cualquier emisor puede usar esa clave para encriptar un mensaje, pero solo el creador puede desencriptarla.
En la encriptación simétrica, habíamos dicho que las personas que se comunicaban debían compartir entre ellos su clave, para poder comunicarse (y esa clave, obviamente, debía ser compartida por un medio seguro, para que nadie más la tenga). Con la encriptación asimétrica ya no existe más ese problema.

El algoritmo más famoso de encriptación asimétrica es **RSA** (inventores *Rivest*, *Shamir*, *Adleman*). Veamos (o recordemos) cómo funciona:
En RSA, $\mathbf{k_e}$ es la **clave pública**, y $\mathbf{k_d}$ es la **clave privada**. N es el producto de dos números primos (p y q) enormes. Luego, tenemos N' = (p-1)(q-1). El valor de $\mathbf{k_e}$ se obtiene de calcular un número que sea *coprimo* (que no tienen factores comunes) con N' (el número *coprimo* debe estar entre 2 y N'-1). Por último, podemos calcular $\mathbf{k_d}$ al hacer que $\mathbf{k_e k_d \bmod N' = 1}$.
Veamos entonces cómo encriptar y desencriptar:

- **Encriptar:** Para cada letra m, calculamos el resto de dividir $\boldsymbol{m}^{\mathbf{k_e}}$ por N.
- **Desencriptar:** Para cada letra encriptada c, calculamos el resto de dividir $\boldsymbol{c}^{\mathbf{k_d}}$ por N.

El método funciona ya que factorizar números es difícil (particularmente, la operación pertenece a la clase NP–*Nondeterministic Polinomial*), incluso para las computadoras más potentes.

// Un sharau a la gente que hizo en el Taller de Álgebra el TP de RSA, solo para los reales.

Autenticación

La encriptación ofrece formas de limitar un conjunto de posibles receptores de un mensaje. Llamamos **autenticación** a limitar el conjunto de potenciales emisores de un mensaje; la autenticación es entonces complementaria a la encriptación. Es muy potente para probar que un mensaje no fue modificado.

Como también hay dos tipos principales de encriptación, también hay dos tipos principales de autenticación. Ambos algoritmos hacen uso de funciones de **hash**. Veamos primero como funcionan estas.
Una **función de hash** H(m) crea un bloque de datos de un tamaño predeterminado, conocido como ***hash value***, a partir de un mensaje m. Las funciones de hash toman un mensaje y lo

dividen en bloques, y luego procesan cada bloque para producir un **hash** de n-bits. H debe ser resistente a colisiones, es decir, NO debe existir un mensaje *m'* tal que $\boldsymbol{m \neq m'}$ y $\mathbf{H(}\boldsymbol{m}\mathbf{)=H(}\boldsymbol{m'}\mathbf{)}$. Con esto en mente, si H(*m*) = H(*m'*) sabemos que *m = m'*, por lo tanto el mensaje no fue modificado.

El primer tipo principal de autenticación utiliza encriptación simétrica, y es conocido como **message-authentication code** (**MAC**).
El segundo tipo principal es un **digital-signature algorithm** (algoritmo de firma digital), y los autenticadores producen, de este modo, firmas digitales. Las firmas digitales son muy útiles, y permiten que **cualquiera** verifique la autenticidad de un mensaje.
Existe un algoritmo de firma digital con RSA. La idea es la siguiente: consideramos como firma digital un hash de nuestro documento. Luego, encriptamos con nuestra clave privada el hash y entregamos el documento con el hash encriptado. El receptor puede desencriptar el documento con la clave pública; si lo logra, se asegura de que nosotros seamos el autor. Luego, verifica que el hash obtenido se corresponda con el del documento (para chequear que no fue modificado ni nada del estilo).

Transport Layer Security (TLS)

TLS es un protocolo criptográfico que permite que dos computadoras se comuniquen de forma segura. Es el protocolo estándar que usan los navegadores web para comunicarse con los servidores web.

Es un protocolo complejo. Consiste en una mezcolanza de encriptación asimétrica y simétrica. En pocas palabras, se utiliza encriptación asimétrica entre cliente y servidor para establecer una ***session key***, que puede ser utilizada para encriptación simétrica entre ambos. Estas claves son olvidadas una vez que la sesión termina.

Autenticación de usuario

Cuando hablamos antes de autenticación, hablábamos de mensajes y sesiones. Si un sistema no puede identificar a un usuario, entonces autenticar un mensaje que vino de ese usuario no tiene sentido. Los usuarios, normalmente, se identifican a sí mismos, pero el problema es determinar si la identidad de un usuario es auténtica. Generalmente, la autenticación de usuario está basada en: la posesión del usuario de un objeto (una clave, o tarjeta), el conocimiento del usuario de algo (una contraseña), o sobre un atributo del usuario (una firma, la retina, o huellas dactilares).

La forma más común de autenticar la identidad de un usuario es con una **contraseña**. Si el ID del usuario y la contraseña ingresada coinciden con los datos almacenados en el sistema, el sistema ASUME que la persona ingresando a la cuenta es el dueño de la misma.
El sistema de autenticación con contraseñas no es el más seguro, ya que los usuarios tienden a usar contraseñas sencillas, son descuidados, alguien puede ver nuestro teclado al escribir nuestra contraseña, etcétera.

Un problema que surge es cómo se hace para mantener secreta la contraseña dentro del sistema. El sistema UNIX utiliza un *hashing* seguro para evitar el uso de una lista secreta de contraseñas. Como la contraseña está *hasheada*, es imposible para el sistema desencriptar el valor almacenado y determinar la contraseña original. Las funciones de *hash* son fáciles de computar, pero difíciles (o imposibles) de invertir.

Prevención de intrusiones

Un **sistema de detección de intrusos** (**Intrusion-Detection System–IPS**) es un programa de detección de accesos no autorizados a un computador o una red.
Es difícil entender cómo está constituída una intrusión, por lo que las IPSs de hoy en día usan uno de dos posibles enfoques.

- Uno es **signature-based detection**, en donde se examinan los *inputs* del sistema y el tráfico de la red, buscando comportamientos específicos (un patrón, una **firma**) conocidos para indicar ataques. Un ejemplo sencillo es escanear los paquetes de la red, buscando el *string /etc/passwd* en los sistemas UNIX. Otro ejemplo, es usar un software de detección de virus, los cuales escanean binarios o paquetes de red en busca de algún virus conocido.
- El segundo enfoque es **anomaly detection**, intenta, mediante diversas técnicas, detectar un comportamiento anómalo en el sistema. Un ejemplo es monitorear las *syscalls* que realiza un proceso en segundo plano, observando si el comportamiento de las *syscalls* se desvió de su patrón común y corriente.

Address Space Layout Randomization (ASLR)

ASLR es una técnica utilizada para evitar los ataques de inyección de código / inyección de parámetros. Para montar uno de estos ataques, el *hacker* deben de poder deducir exactamente en qué parte de la memoria se encuentra su objetivo. Normalmente, esto es sencillo, ya que el diseño de la memoria tiende a ser predecible. Es por esto que ASLR intenta resolver este problema, al randomizar los espacios de direcciones (esto es, colocar espacios de direcciones, como las direcciones del *stack* y del *heap*) en locaciones impredecibles. Esta solución hace que el *exploit* sea más difícil de lograr, pero no imposible.

Protección

Ya hablamos antes sobre **seguridad**, ahora nos toca hablar sobre **protección**, que se encarga de controlar el acceso de los procesos y otros usuarios a los recursos de nuestro sistema computacional.
Los procesos en un SO deben estar protegidos de las actividades de otros. Para proveer esta protección, se pueden implementar varios mecanismos para asegurar que, solo los procesos que hayan ganado la debida autorización del SO, puedan operar sobre archivos, segmentos de memoria, la CPU, redes, y otros recursos del sistema.

Un principio general (y clave) de la protección es el **principio del privilegio mínimo**, el cual dicta que a los programas, usuarios, e incluso sistemas, se les debe otorgar sólo los privilegios necesarios para llevar a cabo sus tareas.
Uno de los principios de UNIX es que un usuario no debería tener privilegios de root. Consideremos que sufrimos un ataque malicioso, un virus que se lanzó por clickear en un link indebido, o tuvimos un buffer overflow, o un ataque de inyección de código. Si uno de esos ataques es realizado contra un usuario que tiene privilegios root, cagamos fuego, sería catastrófico para el sistema.
– Por eso en las compus de la UBA nos REEE cagan y no temenos root
– Usted no aprende, ¿verdad?

Otro principio importante es **compartimentación**. La compartimentación es el proceso de proteger individualmente a cada componente del sistema, a través del uso de permisos específicos y restricciones de acceso.

Un sistema computacional puede ser tratado como una colección de procesos y *objetos*. Con objetos nos referimos a tanto objetos de *hardware* y de *software*. Los objetos son, esencialmente, tipos abstractos de datos. Las operaciones que podemos realizar sobre cada uno varía según el objeto. Por ejemplo, en la CPU solo podemos ejecutar, en la memoria podemos leer y escribir, mientras que en un DVD-ROM solo podemos leer. Los archivos de datos pueden ser creados, abiertos, leídos, escritos, cerrados y eliminados.
A un proceso se le debe dar acceso únicamente a los objetos a los que tiene autorización. Además, un proceso solo debería tener acceso a los objetos que necesita **en ese momento** para completar su tarea. Este segundo requisito, el **need-to-know principle**, es útil para limitar el daño que un proceso fallado o un atacante pueda causar en el sistema.
*Por ejemplo, si el proceso **p** invoca el procedimiento **A()**, el procedimiento únicamente debería poder acceder a sus propias variables y parámetros que se le fueron dados; no debe tener acceso a todas las variables del proceso **p**.*

Un proceso opera con un **dominio de protección**, el cual especifica los recursos a los que ese proceso tiene acceso. Cada dominio define un conjunto de objetos y el tipo de operaciones que el proceso puede realizar sobre cada objeto, dentro de ese dominio. La habilidad de ejecutar una operación sobre un objeto es un **derecho de acceso**.
Pueden existir dominios que comparten derechos de acceso.
La asociación entre un proceso y su dominio puede ser **estática**, si el conjunto de recursos disponibles para el proceso es fijo durante la vida del proceso, o **dinámica**. Si la asociación es *estática*, y queremos apegarnos al *need-to-know principle*, entonces un mecanismo debe estar disponible para cambiar los contenidos de un dominio.

Ejemplo: UNIX

Sabemos que en UNIX un usuario *root* puede ejecutar comandos con privilegios, mientras que los usuarios normales no. Consideremos, por ejemplo, que un usuario normal quiere

cambiar su contraseña. Inevitablemente, esto requiere acceder a la *database* de contraseñas (comúnmente, encontrada en */etc/shadow*), que solo puede ser accedida por un usuario *root*.
La solución a estos tipos de problemas es el **setuid bit**. En UNIX, un identificador de propietario y un bit de dominio, conocido como el **setuid bit**, están asociados con cada archivo. El *setuid* bit puede o no estar prendido. Cuando el bit está prendido, quienquiera que ejecute ese archivo, temporalmente asume la identidad del dueño.
Suena peligroso (de hecho, peligrosísimo), pero los ejecutables binarios de *setuid* son **estériles** (afecta solo los archivos necesarios bajo restricciones específicas) y **herméticos** (a prueba de manipulaciones, e imposibles de subvertir). Los programas *setuid* deben ser escritos de manera extremadamente cuidadosa.

Access Matrix

El modelo general de protección puede ser visto abstractamente como una matriz, llamada **matriz de control de accesos** (o **access matrix**). Cada fila representa un *dominio*, y cada columna representa un objeto. Cada entrada de la matriz representa un conjunto de *derechos de acceso* (justo sobre lo que estábamos hablando antes). La entrada **access(i, j)** define el conjunto de operaciones que un proceso, ejecutando en el dominio D_i, puede invocar sobre el objeto O_j.
La matriz de accesos nos da un mecanismo para definir una variedad de políticas en el sistema. Más específicamente, se debe asegurar que un proceso ejecutando en el dominio D_i SOLO pueda acceder a aquellos objetos especificados en la fila *i*, y que solo pueda realizar las operaciones dictadas por la matriz.
La matriz de accesos puede implementar decisiones políticas relativas a la protección. Una de ellas es decidir el dominio sobre el que cada proceso se ejecutará. Esto suele ser decidido por el SO.
Los usuarios normalmente definen el contenido de la matriz. Cuando un usuario crea un nuevo objeto O_j, la columna O_j es agregada a la matriz, inicializada con sus entradas apropiadas. El usuario puede agregar nuevos derechos para los dominios, tanto en la columna del objeto O_j como en las otras.
Existe un "permiso" que permite que los procesos se muevan entre dominios. Esto lo hacen a través de la operación *switch*. Un proceso en el dominio D_i puede *switchear* al dominio D_j si y solo si tiene el derecho de acceso a hacerlo, es decir, ***switch*** $\in$ **access(i, j)**.

La matriz de accesos tiene ciertos permisos para algunos dominios, como **owner** o **copy**. Como su nombre lo indica, **copy** nos permite copiar o transferir ciertos permisos de un dominio a otro, mientras que **owner** permite borrar, crear y copiar permisos de un dominio a otro.

DAC vs MAC

Los sistemas operativos tradicionalmente han usado ***discretionary access control*** (**DAC**) como método para restringir el acceso a archivos y otros objetos de un sistema computacional. Con DAC, el acceso está basado en las identidades individuales de cada usuario, o en grupos. En sistemas UNIX, DAC toma la forma de permisos-de-archivos (que

se pueden cambiar con los comandos *chmod*, *chown*, y *chgrp*), mientras que en Windows se permite una granularidad más fina por medio de **ACLs** (*access-control lists*).
Sin embargo, se probó que DAC resultó insuficiente con el paso de los años. Una debilidad clave reside en su naturalidad, que permite que el dueño de un recurso pueda establecer o modificar los permisos del mismo. Otra debilidad es el acceso ilimitado que tienen los usuarios *root*.

Una forma más fuerte de protección surgió bajo el nombre de ***mandatory access control*** (**MAC**). MAC se aplica como una política del sistema que, ni siquiera, el usuario *root* puede cambiar (a menos que la política explícitamente lo permita, o se *rebootee* el sistema a una configuración alternativa). Las restricciones en MAC son muy fuertes, y se pueden usar para que ciertos recursos resulten inaccesibles para todos, menos para sus dueños.
MAC utiliza el concepto de **labels** (**etiquetas** o **grados**). Un **label** es un identificador (usualmente, un *string*) asignado a un objeto. También pueden ser asignados a sujetos (procesos, usuarios, etc). Cuando un sujeto quiere realizar una operación sobre cierto objeto, el SO inicia un chequeo (definido en la política de MAC), que determina si un sujeto con cierta **etiqueta** puede realizar su operación sobre el objeto **etiquetado**.
Los usuarios con cierto **label** pueden crear procesos con un *label* similar, los cuales tendrán acceso a los recursos que tengan su mismo *grado* o menor. Ni el usuario, ni los procesos que este crea, saben de la existencia de los recursos que tienen un *grado* más grande que el suyo.

Access Control List (ACL)

Las Access Control Lists (ACLs) son mecanismos de seguridad utilizados en SSOO para definir y controlar el acceso a recursos como archivos, directorios, y otros objetos del SO. Las ACLs permiten que los administradores coloquen permisos sobre estos recursos, especificando qué usuarios o grupos se les puede dar o denegar el acceso, al igual que el tipo de acceso permitido (read, write, execute, etc).

La ACL de **Windows** consiste en Access Control Entries (ACEs). Cada entrada contiene información sobre un usuario específico o sobre un grupo, junto a los permisos que se le son otorgados y/o denegados. Además, Windows utiliza DAC, lo que significa que los dueños de un objeto pueden cambiar sus permisos.
La ACL de **Linux** funciona casi al igual que Windows, con la ligera diferencia de que utiliza DAC y MAC. También, Linux permite que un archivo y/o directorio tenga más de un dueño, es decir, que un grupo sea dueño de un objeto; Windows no nos deja hacerlo.

11) Virtualización, contenedores y nubes

Virtualización

Es la posibilidad de que un conjunto de recursos físicos se vean como varias copias de recursos lógicos. La aceptación más común es pensar en una computadora realizando el trabajo de varias.

Un tentador ejemplo de la *virtualización* son las **máquinas virtuales**. Las razones por las que querríamos tenerlas son variados:

- Portabilidad
- Simulación / Testing
- Aislamiento
- Particionamiento del hardware
- Protección ante fallas de hardware
- Agrupamiento de funciones
- Migración entre hardware sin pérdida de servicio

Generalmente, en una máquina virtual (VM), los SSOO invitados y las aplicaciones corren en un ambiente que, para ellos, es el hardware nativo de la computadora (y funciona como el hardware nativo de la computadora), pero también protege, administra y limita a estas aplicaciones.

Las implementaciones de las VMs pueden variar. Algunas de ellas son:

- **Simulación:** El sistema *anfitrión* construye una variable de estado que representa al sistema *huésped*. Se lee cada instrucción y se modifica el estado como si ésta se estuviera ejecutando realmente. Aún así, el mecanismo puede resultar muy lento.
- **Emulación:** El sistema emulado se ejecuta realmente en la CPU del anfitrión. Se emulan componentes de hardware. Aplicaciones escritas para un entorno de hardware corren en uno completamente diferente.
  Cuando la VM cree que está haciendo E/S de un dispositivo, en realidad lo está haciendo contra el controlador de VMs.

Con estos métodos (y otros) se tienen muchos problemas ya que en las VMs corre un SO, el cual tiene programas escritos para modo kernel, pero en realidad se están ejecutando en modo usuario. Con esto surgen problemas de permisos para ejecutar ciertas instrucciones, y no hay protección entre kernel y programas de usuario. Además, no queremos que la máquina pise la memoria del propio emulador. También, debemos simularle interrupciones al SO de la VM.
Para solucionar estos problemas, los fabricantes agregaron soporte para la virtualización en el hardware.
En el caso de Intel, le agregaron al procesador extensiones que poseen dos modos:

- **VMX root**: Las instrucciones se comportan de manera similar, pero hay algunas extensiones. (*anfitrión*)
- **VMX non-root:** Mismo set de instrucciones, pero con comportamiento restringido. (*huésped*)

Contenedores

En el contexto de la virtualización, un *contenedor* es una unidad portable y autosuficiente que encapsula y corre una aplicación y sus dependencias. En otras palabras, son una tecnología

que se usa para agrupar una aplicación con todos sus archivos necesarios en un entorno de ejecución.

Docker es la plataforma de contenerización más usada para implementarlos. Provee un conjunto de herramientas y una plataforma para desarrollar, enviar y correr aplicaciones en contenedores.
Otro método es **Kubernetes**. Es una plataforma de código abierto para automatizar la implementación, el escalado, y la administración de aplicaciones en contenedores.
También existe **OpenShift** (y **OKD**). OpenShift usa kubernetes de base, pero agrega restricciones de seguridad por defecto, una interfaz web más completa, manejo de roles, y facilidades para el desarrollador.
Estos tres métodos mencionados son una de las tantas formas que existen para implementar contenedores.