Sistemas Operativos - Apuntes para final

Gianfranco Zamboni

10 de octubre de 2021

Índice

1. Introducción

Un sistema informático tiene cuatro componentes:

- El hardware (CPU, memoría y dispositivos de entrada salida) proveen los recursos básicos del sistema.
- Las aplicaciones definen la forma en que estos recursos va a ser usados para resolver los problemas del usuario.
- El sistema operativo controla el hardware y coordina su uso entre las distintas aplicaciones de los usuarios.

1.1. El sistema operativo

Un sistema operativo provee un entorno de ejecución de programas. Para esto tiene ciertos componentes que difieren de sistema en sistema pero que entre lo más comunes se encuentran:

- **Drivers:** Programas manejan los detalles de bajo nivel relacionados con la operación de los distintos dispositivos.
- **Núcleo**: (o Kernel) Es el sistema operativo, propiamente dicho. Se encarga de las tareas fundamentales y contiene los diversos sub-sistemas que iremos viendo a lo largo de la materia.
- **Sistema de archivos**: Forma de organizar los datos en el disco para gestionar su acceso, permisos, etc.
  - **Archivo:** Secuencia de bits con un nombre y una serie de atributos que indican permisos.
    - **Binario del sistema**: Son archivos que no forman parte del kernel pero suelen llevar a cabo tareas muy importantes o proveer las utilidades básicas del sistema.
    - **Archivos de configuración:** Son archivos especiales con información que el sistema operativo necesita para funcionar.
  - **Directorio**: Colección de archivos y directorios que contiene un nombre y se organiza jerárquicamente.
    - **Directorios del sistema**: Son directorios donde el propio SO guarda archivos que necesita para su funcionamiento.
  - **Dispositivo virtual**: Abstracción de un dispositivo físico bajo la forma, en general, de un archivo de manera tal que se pueda abrir, leer, escribir, etc.
  - **Usuario**: La representación, dentro del propio Sistema Operativo, de las personas qo entidades que pueden usarlo. Sirve principalmente como una forma de aislar información entre distintos usuarios reales y de establecer limitaciones.
  - **Grupo**: Una colección de usuarios

Systems calls (Syscalls): Son rutinas que proveen una interfaz a los servicios disponibles en el sistema. Generalmente, son rutinas escritas en C/C++ y llamarlas implican un cambio de contexto y privilegios del proceso a ser ejecutado. Algunos de estos servicios son:

- Creación y control de procesos.
- Pipes.
- Señales
- Operaciones de archivos y directorios
- Excepciones
- Errores del bus
- Biblioteca C

Parte I
Manejo y comunicación entre procesos

2. Procesos y API

Proceso: Es un programa en ejecución y todas las estructuras que debe mantener el sistema para su correcto funcionamiento. Entre ellas, los valores de los registros, el program counter, el stack del proceso, el área de memoria en la que se cargan las instrucciónes a ejecutar (el programa propiamente dicho) y el área de datos (contiene variables globales) y un heap de memoria (que es donde reserva memoria dinámica).

2.1. Creación y control de un proceso

2.1.1. Creación de un proceso

Durante su ejecución, un proceso puede crear nuevos procesos. En este caso llamamos **proceso padre** al proceso que crea nuevos procesos y **procesos hijos** a los procesos creados. Además, cada proceso hijo podrá crear otros procesos, formando así un **árbol** de procesos.

En la mayoría de los sistemas operativos, se asigna a cada proceso un identificador numérico único llamado **process id** o **pid** que sirve como índice para poder acceder a varios de sus atributos.

El nuevo proceso hijo creado necesitara de ciertos recursos para poder llevar a cabo su objetivo. Estos recursos podrán ser obtenidos directamente del sistema operativo o estar restringidos a los recursos del proceso padre.

Cuando un proceso crea un nuevo proceso (`fork()`) hay dos posibilidades:

- El padre y su hijo continúan ejecutando concurrentemente.
- El padre espera a que alguno o todos sus hijos terminen para seguir ejecutando (`wait()`).

Además, el nuevo proceso puede ser un duplicado de su padre o cargar un nuevo programa (`exec()`)

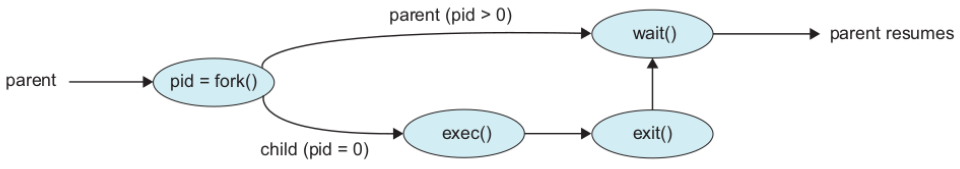

Figure 3.10 Process creation using the `fork()` system call.

2.1.2. Terminación

Un proceso termina cuando ejecuta su última instrucción y pide al sistema operativo que lo borre usando la syscall `exit()`. En este momento, el proceso envía el código de estado con el que terminó a su padre y el sistema operativo libera todo los recursos que le había asignado.

La terminación puede ocurrir por varios motivos. Un proceso puede causar la terminación de otro usando las syscalls adecuadas (usualmente, esto puede hacerlo solo el proceso padre).

2.1.3. Un poco más sobre las syscalls mencionadas

- `pid_t fork()`: Crea un nuevo proceso. En el caso del creador se retorna el PID del hijo. En el caso del hijo, retorna 0.
- `int execve(const char* filename, char* const argv[], char* const envp[] )`: Sustituye la imagen de memoria del programa por la del programa ubicado en filename.
- `pid_t forkv()`: Crea un hijo sin copiar la memoria del padre, el hijo tiene que hacer `exec`.

  Un proceso creado con esta syscall comienza con sus páginas de memoria apuntando a las mismas que las de su padre. Recién cuando alguno escriba en memoria, se hace la copia. Esto se llama **Copy-On-Write**.
- `pid_t wait(int* status)`: Bloquea al padre hasta que algún hijo termine o hasta que alcance el status indicado.
- `pid_t waitpid(pid_t pid, int* status)`: Igual al anterior pero espera a que el hijo con PID `pid` llegue a ese status.
- `void exit(int status)`: Finaliza el proceso actual.
- `clone(...)`: Crea un nuevo proceso. El hijo comparte parte del contexto con el padre. Es usado en la implementación de threads.

Fork Bomb: Un proceso crea infinitos hijos.

2.2. Estados de un proceso

Durante su ejecución, un proceso va modificando su estado acorde al siguiente diagrama:

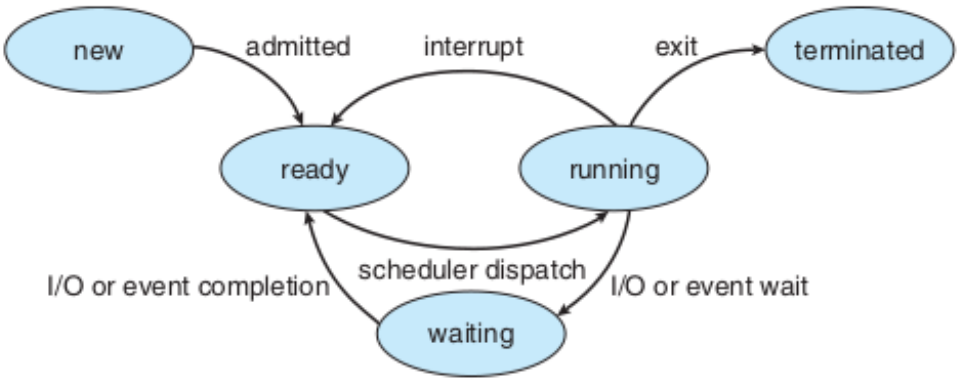

Figure 3.2 Diagram of process state.

- **Nuevo (New):** El proceso se está creando.
- **Listo (Ready):** El proceso está listo para ser ejecutado.
- **Bloqueado (Waiting):** El proceso está esperando a que ocurra algo (por ejemplo a que se complete una operación de entrada salida)
- **Ejecutando (Running):** El proceso se está ejecutando.
- **Terminado (Terminated):** El procesó termino de ejecutarse.

Es importante notar que solo un proceso puede estar **ejecutando** en un instante de tiempo de dado. Sin embargo, muchos procesos pueden estar bloqueados o listos simultáneamente.

2.3. Process Control Block (PCB)

Cada proceso está representado, en el sistema operativo, por un Process Control Block o Task Control Block. El mismo contiene información asociada a cada proceso, entre ellas:

- El estado del proceso
- El program counter (la dirección de memoría de la próxima instrucción a ser ejecutada)
- Los registros del CPU,
- Información de Scheduling, como puede ser prioridad de ejecución
- Información de manejo de memoria, como las direcciones de los directorios de paginas o las tablas de segmentación.
- Estadísticas de uso del sistema
- Información de E/S (que dispositivos está usando, lista de archivos abiertos, etc)

2.4. Scheduler

Multiprocesador: Un equipo con más de un procesador.

Multiprogramación: La capacidad de un SO de tener varios procesos en ejecución.

Multiprocesamiento: Se refiere al tipo de procesamiento que sucede en los multiprocesadores.

Multitarea: Es una forma especial de multiprogramación, donde la conmutación entre procesos se hace de manera tan rápida que da la sensación de que varios programas están corriendo en simultáneo.

Multithreaded: Son procesos en los cuales hay varios *mini procesos* corriendo en paralelo (de manera real o ficticia).

2.4.1. Context Switching

Una interrupción hace que el sistema operativo cambie la tarea de una CPU y corra un rutina del kernel. Cuando esto ocurre, el sistema, debe guardar el **contexto** del proceso que se está ejecutando para poder retornar a la ejecución del mismo cuando termine de atenderla. Este contexto es el PCB del proceso.

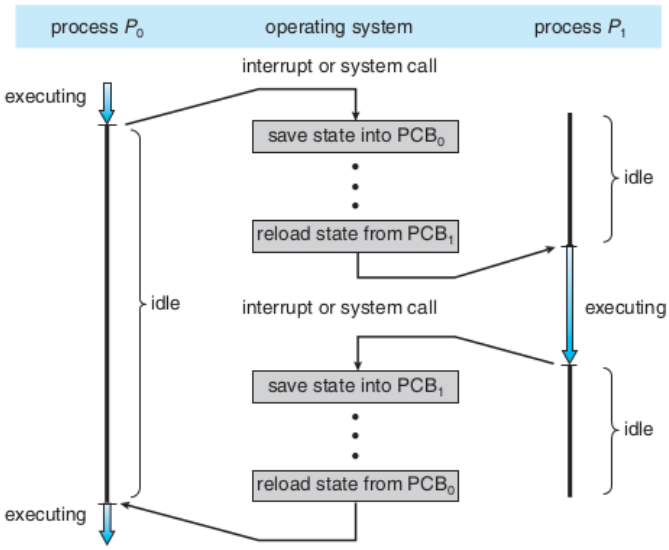

Figure 3.4 Diagram showing CPU switch from process to process.

2.4.2. Colas de procesos

Una de las características de los sistemas operativos actuales es que nos permiten correr varias aplicaciones a la vez, sin embargo solo un proceso puede estar ejecutando en la CPU. Para darnos la sensación de simultaneidad, el SO hace lo que se llama **preemption**.

Preemption: A cada proceso le asigna un **quantum** o una cantidad de tiempo durante la cual es ejecutado. Una vez que transcurrido ese tiempo, se guarda el contexto del proceso en memoria y se carga el contexto del próximo proceso a ejecutar.

El objetivo de la multiprogramación es intercambiar las tareas que se ejecutan en la CPU lo suficientemente rápido como para que el usuario pueda interactuar con dada programa como si estuviesen ejecutándose simultáneamente.Para esto existe un proceso especial llamado **Scheduler** que selecciona el próximo proceso que debe ser ejecutado.

El mismo dispone de varias colas de procesos en los que se encuentran los PCB de los procesos en ejecución. Los PCB se van encolando y desencolando de cada cola dependiendo del estado de cada proceso.

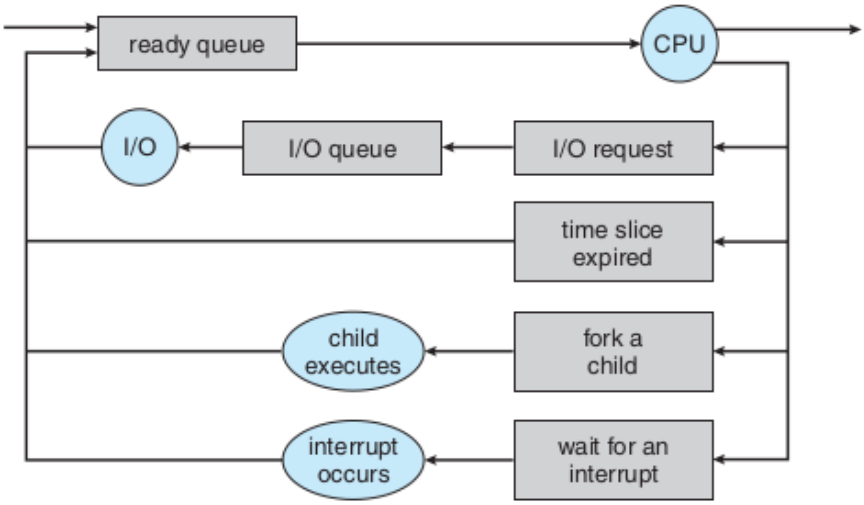

Figure 3.6 Queueing-diagram representation of process scheduling.

2.5. Inter Process Communication (IPC)

Los procesos que se están ejecutando concurrentemente en el sistema operativa pueden ser **procesos independientes** o **cooperativos**.

Procesos independiente: Es un proceso que no puede afectar o ser afectado por otros procesos ejecuntandose en el sistema operativo.

Procesos cooperativos: Son procesos que no son independientes (pueden afectar o ser afectados por otros procesos).

Los procesos cooperativos requieres de un mecanismo de comunicación que les permita intercambiar datos e información. Hay dos modelos fundamentales para hacer esto: **memoria compartida** y **Pasaje de mensajes**.

En el modelo de memoria compartida, se establece un área de memoría que es compartida entre procesos. Los procesos pueden intercambiar información leyendo y escribiendo en este área.

En el modelo de pasaje de mensajes, la comunicación se realiza a través de mensajes entre los procesos cooperativos.

2.5.1. Pasaje de mensaje

Los mensajes proveen a los procesos un mecanismo que permite que los procesos se comuniquen entre si y sincronicen sus acciones. Un sistema que permite el pasaje de mensajes provee al menos dos operaciones:

```
send(message)      receive(message)
```

Los mensajes pueden tener una longitud fija o variable. La primera opción es más fácil de implementar pero hace que la programación de tareas sea mas tediosa. Los mensajes de longitud variable necesitan un sistema más complejo pero permite más agilidad a la hora de programar tareas.

Si un proceso P y Q se quieren comunicar, entonces debe existir un **link de comunicación** entre ellos. Para crear este link, se deben tener en cuenta el tipo de comunicación que se desea ofrecer:

- **Direccionamiento**
  - **Conexión directa:** Cada proceso debe explicitar el nombre del proceso destinatario/receptor:
    - `send(P, message)` - Envía un mensaje al proceso P
    - `receive(Q, message)` - Recibe un mensaje del proceso Q

    En este caso, el link de comunicación se establece automáticamente entre cada par de procesos que quiere comunicarse. Y cada proceso sabe la identidad del otro.
  - **Comunicación indirecta:** Los mensajes son enviados a buzones (**mailboxes**) o puertos (**ports**). Cada buzón tiene un identificador único. Un proceso puede comunicarse con otro a través de varios buzones.
    - `send(A, message)` - Envía un mensaje al buzón A
    - `receive(A, message)` - Recibe un mensaje del buzón A

    En este esquema, un link se establece entre dos procesos solo si comparten un buzón. Además un link puede estar asociado a varios pares de procesos e incluso puede haber varios links entre dos mismos procesos.

- **Sincronización:** El pasaje de mensajes puede ser bloqueante (síncrono) o no bloqueante (asíncrono):
  - **Envío bloqueante:** El proceso que envía un mensaje espera a que el mismo sea recibido por su destinatario.
  - **Envío no bloqueante:** El proceso envía el mensaje y sigue su ejecución.
  - **Recepción bloqueante:** El proceso se bloquea hasta que recibe un mensaje.
  - **Recepción no bloqueante:** El receptor recibe un mensaje válido o null.
- **Buffering**: Los mensajes intercambiados entre procesos deben almacenarse en una cola temporal. Estas colas pueden ser de tres tipo:
  - **Capacidad Cero:** El link no puede tener mensajes en espera. El remitente debe bloquearse hasta que el mensaje sea recibido.
  - **Capacidad acotada:** La cola tiene una capacidad finita n por lo que puede haber a lo sumo n en espera. Si la cola no está llena se puede enviar un mensaje, sino el remitente debe bloquearse hasta que se que haya espacio disponible.
  - **Capacidad infinita:** El remitente nunca se bloquea.

2.5.2. Sockets

Los sockets son los extremos de una comunicación que usan dos procesos para comunicarse a través de una red. Cada socket está identificado por una dirección IP y un número de puerto.

En general, en este tipo de links, se usa una arquitecutra cliente-servidor: El servidor espera a que un cliente haga un pedido y, una vez que lo recibe, acepta la conexión del socket del cliente para completar la conexión. El número del socket del servidor, en general, va a ser menor o igual a 1024. Estos puerstos son los puertos conocidos (*well known ports*) e implementan distintos protocolos estandarizados.

Cuando el proceso cliente inicia la conexión, la computadora que lo está ejecutando le asigna un puerto arbitrario cuyo número es mayor a 1024.

2.5.3. Pipes

Un **pipe** es un canal que provee una de las formas más simples de comunicación entre dos procesos aunque tienen sus limitaciones. Al implementar un pipe, se debe tener ciertas consideraciones:

1. ¿El pipe permite comunicación bidireccional o unidireccional?
2. Si es bidireccional, es **half-duplex** (para mandar información se debe esperar a que el otro extremo del pipe termina de hacerlo) o **full-duplex** (la información puede viajar de un lado a otro y viceversa simultaneamente).

3. ¿Debe existir alguna relación entre los procesos que se están comunicando (por ejemplo, padre-hijo)?

4. ¿Los procesos se van a poder comunicar dentro de una red o tienen que estar en la misma maquina?

Ordinary pipes: Permiten que dos procesos se comuniquen en modo productor-consumidor: El productor escribe en un extremo del pipe (extremo de escritura) y el consumidor lee desde el otro extremo (extremo de lectura). Estos pipes son unidireccionales. Si se necesita una comunicación bidireccional se debe crear otro pipe que permita mandar datos en la otra dirección.

Este tipo de pipes requieren que los procesos que se comunican tengan una relación padre-hijo y dejan de existir una vez que la comunicación termina.

Named pipes: Proveen comunicación bidireccional y no necesitan que los procesos estén relacionados. Una vez que es establecido, varios procesos pueden usarlo para comunicarse y continuan existiendo incluso después de que las comunicaciones hayan finalizado.

2.6. E/S bloqueante / no bloqueante

Cuando un proceso necesita escribir o leer información de algún dispositivo necesita realizar operaciones de **entrada/salida**

Estas operaciones son muy lentas por lo que quedarse bloqueado es un desperdicio de tiempo.

Busy Waiting: El proceso no hace nada pero no libera el CPU. Se gastan ciclos de procesamiento en hacer nada.

Para evitar esto se utilizan algunas técnicas que permiten al SO seguir ejecutando mientras espera la respuesta de los dispositivos:

- **Polling**: El proceso libera la CPU pero todavía recibe un quantum cada tanto que desperdicia hasta que la E/S esté terminada.
- **Interrupciones:** Esto permite la multiprogramación. El SO no le otorgá más quantum al proceso hasta que su E/S esté lista. El hardware comunica esto mediante una interrupción que hace que el proceso se despierte.

2.7. Manejo básico de un shell Unix

2.7.1. File descriptors:

En Unix, cada proceso se crea con una tabla que le permite identificar cuales son los archivos que tiene abiertos. Cada índice es un **file descriptors** que usa el Kernel para saber como leer/escribir datos en los distintos archivos (en Unix, el teclado y la pantalla se modelan como archivos).

Además, cada proceso hereda de su proceso padre tres archivos abiertos que ocupan los file descriptors 0, 1 y 2 y representan la entrada estándar (**stdin**), la salida estandar (**stdout**) y el error estandar (**stderr**), respectivamente.

Linux provee de dos llamadas al sistema que nos permiten leer/escribir a un archivo usando su file descriptor `fd`:

```
ssize_t read(int fd, void *buf, size_t count)
ssize_t write(int fd, const void *buf, size_t count);
```

Aquí `buf` es un puntero a donde se almacenan los datos a leer o escribir y `count` la cantidad de bytes que hay escribir/leer.

2.7.2. Comandos de consola

- `echo` escribe lo que le pasemos como parámetro en su **stdout**.

  ```
  echo ‘‘Esto es un mensaj’’
  ```

- `>` Se le indica a la consola que el **stdout** se redirija a un archivo:

  ```
  echo ‘‘Esto es un mensaje’’ >mensaje.txt
  ```

- `|` Redirige el **stdout** de un proceso hacia el **stdin** de otro:

  ```
  echo ‘‘Esto es un mensaje’’ | wc -c
  ```

  En este caso, el primer proceso ejecuta el comando `echo` que imprime en **stdout** el mensaje `‘‘Esto es un mensaje’’`. El segundo proceso recibe por **stdin** lo que se escribió en el **stdout** del proceso que ejecutó `echo`

3. Scheduling

En sistemas con un único procesador, solo se puede correr de a un proceso por vez. Uno de los objetivos de la multiprogramación es que todo el tiempo se esté ejecutando un proceso para maximizar el uso de CPU.

Cada vez que el CPU entra en estado IDLE, el sistema debe seleccionar alguno de los procesos que estén listos para ser ejecutados. El proceso de selección es llevado a cabo por el **scheduler** usando la que se llama **cola de ejecución**.

3.1. Objetivos de la política de scheduling

Diferentes algoritmos de schedulling tienen diferentes propiedades y la elección de los mismos depende de las situación particular del sistema. Por lo general, un algoritmo de schedulling busca optimizar alguna combinación de las siguientes propiedades:

- **Eficiencia**: Maximizar la cantidad de tiempo que el CPU esté ocupado.
- **Rendimiento (Throughput)**: Maximizar el número de procesos terminados por unidad de tiempo.
- **Tiempo de ejecución (Turnaround time)**: Minimizar el tiempo total que le toma a un proceso ejecutar completamente (el intervalo de tiempo desde que el proceso se crea hasta que temina, incluye tiempo de esperas en la cola de procesos).
- **Ecuanimidad (Fairness)**: Que Cada proceso reciba una dosis "justa"de CPU (para alguna definición de justicia).
- **Carga del sistema (Waiting time)**: Minimizar la cantidad de tiempo que un proceso esté en la cola de espera.
- **Tiempo de respuesta (Response time)**: Minimizar el tiempo de respuesta percibido por los usuarios interactivos.
- **Latencia**: Minimizar el tiempo requerido para que un proceso comienze a dar resultados.
- **Liberación de recursos:** Hacer que terminen cuanto antes los procesos que tiene reservados más recursos.

Muchos de estos objetivos son contradictorios. Si los usuarios del sistema son heterogéneos, pueden tener distintos intereses por lo que cada política de scheduling debe buscar maximizar una función objetivo que es una combinación de estas metas tratando de impactar lo menos posible en el resto.

3.2. Scheduling con y sin desalojo

El sistema puede tomar decisiones de scheduling en alguna de las siguientes situaciones:

1. Cuando un proceso pasa de estado *Ejecutando* al estado de espera (por ejemplo cuando hace un request de entrada salida)
2. Cuando pasa de *Ejecutando* a *Listo*
3. Cuando pasa de *Esperando* a Listo
4. Cuando termina.

Starvation (Bloqueo indefinido) : Un proceso sufre de starvation cuando está listo para ser ejecutado pero la CPU nunca le asigna clocks de reloj en lo que ejecutar.

Scheduling sin desalojo: Tambien conocido como **coperativo** o **nonpremptive**, se da cuando las decisiones de scheduling solo toman lugar en las situaciones 1 y 4, es decir se espera a que el proceso haya terminado o esté inactivo. Tiene como desventaja que si un proceso muy largo toma control del procesador se puede generar un cuello de botella y otros procesos mas cortos tardarían demasiado en ser ejecutados.

Scheduling con desalojo: Tambien llamado scheduling *apropiativo* o *preempitve*, se vale de la interrupción del clock para decidir si el proceso actual debe seguir ejecutando o le toca a otro. No da garantías de continuidad a los procesos.

Por lo general se usa una combinación de los dos tipos de scheduling para decidir las políticas adecuadas.

3.3. Políticas de scheduling

3.3.1. First In/First Out (FIFO)

El algoritmo FIFO (también conocido como First Come, First Served) es una de las políticas de scheduling más simples. Los process control block (PCB) se ubican en una cola, el próximo proceso a ejecutar es el que está en la cabeza. Cuando un proceso está listo para ser ejecutado se lo encola al final.

Por un lado, es simple de implementar. Por otro, el tiempo de espera promedio de un proceso es bastante largo y hay que tener en cuenta que es un algoritmo sin delajo. Si llega un proceso que requiera mucho tiempo de CPU, tapona todos los demás, esto se llama **convoy effect**.

3.3.2. Shortest Job First (SJF)

Este algoritmo asocia cada proceso con su duración y ejecuta primero aquellos que duran menos. Está ideado para sistemas donde predominan los trabajos batch y está orientado a maximizar el throughput.

Si los procesos ejecutados tienen un comportamiento regular, se puede usa el historial de ejecución para predecir los tiempos ejecución de los procesos actuales. Sin embargo, en sistemas con procesos heterogéneos no es posible saber cuanto tiempo de ejecución va a necesitar cada uno por lo que no es posible implementar este algoritmo.

3.3.3. Round Robin

Este algoritmo está especialmente diseñado para sistemas de tiempo compartido (varios procesos deben usar el cpu al mismo tiempo). Se comporta de manera similar al FIFO, solo que se agrega desalojo para permitir al sistema ejecutar otros procesos.

Para esto se define una pequeña unidad de tiempo llamada **quantum** durante la cual puede correr cada proceso. Si la ráfaga de procesamiento (CPU Burst) de un proceso es mas chico que el quantum, entonces el mismo la liberará y el scheduler eligirá el siguiente proceso a ejecutar.

Si el CPU Burst toma más de un quantum, entonces se enviará una interrupción al sistema. Éste desalojará el proceso, cambiará el contexto y el proceso que se estaba ejecuntando se pondrá al final de la cola de ejecución.

El rendimiento de este tipo de algoritmos depende del tamaño del quantum. Por un lado, si el quantum es demasiado largo, la política de Round Robind termina siendo una FIFO. Por el otro, si el quantum es extremadamente pequeño se pueden producir una gran cantidad de cambios de contexto, por lo que una gran parte del tiempo del CPU sería gastado solo en esto.

En general, se debe elegir el quantum de tal manera que la mayoria de los procesos termine su CPU Burst durante el mismo pero no tan largo como para que sea un FIFO.

3.3.4. Multiples colas

Otra idea, es serparar los procesos en distintas colas de acuerdo a la duración de su CPU Burst. Cada cola tendrá mas prioridad sobre la otra. Si un proceso toma demasiado tiempo, entonces se lo mueve a una cola con una prioridad menor.

Este esquema, da mayor prioridad a los procesos interactivos y aquellos procesos que estén esperando demasiado tiempo pueden ser movidos a colas de mayor prioridad para evitar starvation.

4. Sincronización de procesos (Memoria compartida)

Normalmente, los sistemas operativos tratan de prevenir que un proceso acceda a la memoria de otro. Sin embargo, si dos o mas procesos deciden remover esta restricción pueden definir una región de memoria mediante la cual podrán intercambiar información. En este caso, los mismos procesos son los responsables de asegurar que no escriben simultáneamente en esta región.

Contención y concurrencia: Ocurre cuando uno o más procesos tratan de acceder al mismo recurso de manera simultánea. Esto puede causar que algunas secuencias de ejecución terminen con resultados erróneos.

4.1. Modelo Productor-Consumidor

Uno de los paradigmas clásicos de procesos cooperativos con este tipo de intercomunicación es el de **productor-consumidor**. En este esquema, un proceso **productor** debe producir información que va a ser consumida por un proceso **consumidor**.

Una posible implementación de este problema, es definir un buffer en una región de memoria compartida entre ambos procesos en la cual el productor pueda encolar elementos mientras que el consumidor los desencola. Ambos procesos deben estar sincronizados de tal manera que el consumidor no trate de desencolar elementos si la cola está vacía.

Se pueden consierar dos tipos de buffers en esta solución:

- **Unbounded buffer** (buffer infinito): No posee límites de espacio. El consumidor tiene que esperar a que el buffer tenga algo y el productro siempre puede agregarle elementos.
- **Bounded buffer** (buffer limitado): Tiene un tamaño fijo. Si está vacío, el consumidor debe esperar a que haya algo. Si está lleno, el productor debe esperar a que se haya consumido por lo menos un elemento antes de agregar otro.

Veamos una solución con buffer limitado:

Zona de memoria compartida

```
buffer in[BUFFER_SIZE];
int cant = 0;
```

Proceso productor

```
while(true){
  while(counter == BUFFER_SIZE) {};
  in.push(item);
  counter++;
}
```

Proceso consumidor

```
while(true){
  while(counter == 0) {};
  item = pop(in);
  counter--;
}
```

4.1.1. Condiciones de carrera (race conditions)

En el ejemplo anterior, si bien el código para cada proceso puede parecer correcto, puede haber errores si ambos se ejecutan de manera concurrente. Supongamos que las lineas `counter++` y `counter--` tienen la siguiente forma en lenguaje maquina:

`counter++`:

```
register1 = counter;
register1 = register+1;
counter = register1
```

`counter--`:

```
register2 = counter;
register2 = register2-1;
counter = register2;
```

Supongamos que el valor inicial de `counter` es 5, el proceso productor agrega un elemento al buffer, el consumidor quita el primer elemento encolado y ambos procesos ejecutan `counter++` y `counter--` de manera concurrente, entonces una posible secuencia de ejecución de estas instrucciones podria ser:

Productor	**Consumidor**	`counter`
		5
`register1 = counter`		5
`register1 = register + 1`		5
	`register2 = counter`	5
	`register2 = register - 1`	5
`counter = register1`		6
	`counter = register2`	4

En este caso, el consumidor comienza a modificar la variable `counter` antes de que el productor guarde el valor correspondiente en memoria y no se "entera"que un nuevo elemento fue agregado a la cola (por lo que todavía habría 5 elementos en ella). Entonces se genera una inconsistencia entre el valor de `counter` y la cantidad elementos en el buffer.

Este tipo de situaciones se llama **race condition**: Se da cuando varios procesos pueden acceder y modificar la misma variable de manera concurrente y el valor final de ésta depende del orden particular en el que se hayan realizado los accesos a la misma.

4.2. Secciones críticas

La idea es que cada proceso que se esté ejecutando en el sistema tenga un segmento de código, llamado **sección crítica**, que solo se puede ejecutar cuando ningún otro proceso esté en ella. Es decir, no puede haber mas de un proceso ejecutando su sección crítica al mismo tiempo. Para poder implementar esto, por lo general, se divide el código del segmento en tres partes:

1. **Entry section:** En donde el proceso pide permiso para acceder a la sección crítica.
2. **Critical section:** La zona crítica per se, en la que se puede manejar la memoria compartida
3. **Exit section:** Donde el proceso avisa al sistema que dejó de realizar operaciones críticas.

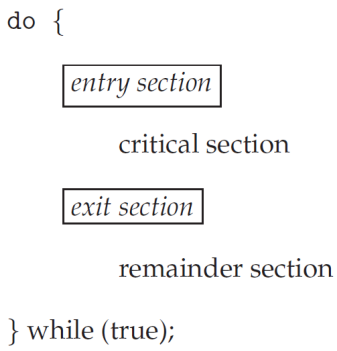

Figura 1: Estructura general de un proceso que implementa sección crítica

Una buena implementación de esta método debe sastifacer los siguientes requerimientos:

1. **Exclusión mutua**: Si un proceso está ejecutando su sección crítica, entonces ningún otro debe estar ejecutando la suya.
2. **Progreso**: Si un proceso necesita entrar a su sección crítica, entonces se le dará permiso para hacerlo en algún momento.
3. **Bounded waiting (espera acotada/no bloqueante)**: Si un proceso P pide entrar a la sección crítica, entonces hay un límite en la cantidad de veces que se le da mayor prioridad a otros procesos sobre P.

4.2.1. Instrucción TestAndSet

Los sistemas opertivos modernos proveen una instrucción de hardware especial que nos permite testear y/o modificar una palabra de manera **atómica**, es decir como unidad no interrumpible de ejecución. Cuando se utiliza, el sistema no ejecuta ninguna otra instrucción hasta que ésta haya terminado. Para abstraernos, de sistemas operativos espécificos, llamemos esta instrucción `testAndSet()`, la misma toma como paramétro una variable booleana que se va a setear en `true` y va devolver el valor anterior:

```
bool testAndSet(bool* source) {
  bool result = *source;
  *source = true;
  return result;
}
```

En este caso, un sistema de procesos que utilice esta instrucción podría tener una variable booleana compartida llamada **lock** que controlaria el acceso a la secciónes criticas de cada proceso, si `lock == false` entonces es posible entrar, si es `true` el proceso debe esperar:

```
bool lock;

void main() {
  while(true) {
    ...
    while(testAndSet(&lock)) {};

    /* Sección crítica */

    lock = false;
    ...
  }
}
```

4.2.2. Instrucción CompareAndSwap

Es una instrucción primitiva ofrecida por varios porcesadores que toma 3 argumentos: una dirección de memoria, un valor esperado y un valor que se debe escribir en esa posición si se encuentra el valor esperado:

```
bool testAndSet(int* value, int expected, int new_value) {
  int temp = *value;
  if(*value == expected) *value = new_value
  return temp;
}
```

4.2.3. Mutex Lock o SpinLock (busy waiting)

La solución basada en hardware presentada en la sección anterior, generalmente es inaccesible a los programadores. Por esta razón, los sistemas operativos diseñan herramientas básicas para implementar secciones críticas: La más simple de ellas es el **mutex lock**. El mismo contiene de una variable booleana `avaliable` que indica si el lock está disponible o no y provee dos funciones:

- `acquire()`: Permite a un proceso adquirir el lock y bloquear otros procesos el acceso a su sección crítica. Si un proceso llama a esta función y el lock ya estaba tomado, el proceso queda en espera hasta que el lock se libere.

  ```
  void aquire() {
    while(!avaliable) {}; // Busy wait
    avaliable = false;
  }
  ```

- `release()`: Libera el lock para que los procesos bloqueados puedan continuar con su ejecución.

```
void release() {
  avaliable = true;
}
```

Ambas funciones deben ejecutarse de manera atómica por lo que generalmente son implementadas usando la instrucción de hardware `testAndSet()`. La principal desventaja es que requiere de **busy waiting** (mientras un proceso está en su sección crítica, cualquier proceso que llame a `aquire()`) debe ciclar continuamente hasta que el lock se libere. Esto es un problema en los sistemas de multiprogramación, donde los ciclos de CPU son compartidos entre varios procesos.

4.3. Semáforos

Un semáforo `S` es una estructura que contiene una variable entera `value` y una lista `list` de procesos a las que se puede acceder mediante dos operaciones atómicas:

- `wait()`: Cuando un proceso llama a esta operación, si `S.value` es negativo entonces debe esperar. Sin embargo, en vez de hacer busy waiting, el proceso se bloquea (entra en estado `waiting`, ver sección 2.2) y se encola en `S.list`. Luego, el control es transferido al scheduler que selecciona otro proceso para ejecutar.

  ```
  void wait(S) {
    S.value--;
    if(S.value < 0) {
      agregar este proceso a S.list;
      block();
    }
  }
  ```

- `signal()`: Cuando un proceso termina de ejecutar su sección crítica, llama a esta operación que indica al semáforo que puede despertar alguno de los procesos en espera. Para esto, se quita algun proceso de la lista y se lo pasa a estado `ready` para que que el scheduler lo vuelva a tener en cuenta.

  ```
  void signal() {
    S.value++
    if(S.value <= 0) {
      quitar proceso P de S.list;
      wakeup(P);
    }
  }
  ```

Este tipo de semáforos está pensado para administrar la asignación de recursos (de los cuales se tiene una o más instancias) del sistema.

La lista de procesos en espera puede ser implmentada por un campo `link` en cada process control block (sección 2.3) y la lista de procesos del semáforo en realidad es una lista punteros a PCBs.

4.3.1. Monitores y variables de condición

Aunque los semáforos proveen un mecanismo conveniente y efectivo para realizar la sincronización de proceso, usarlos incorrectamente puede conllevar a errores dificiles de detectar, ya que estos pueden ocurrir en una secuencia de ejecución particular.

Para resolver estos errores, se dasorrollo un tipo abstracto de dato llamado **monitor** que contiene un conjunto de operaciones que ya aseguran mutual exclusion. El monitor declara variables locales que definen su estado junto con las funciones que operan esas variables (y son la única forma de accederlas).

```
Monitor M {
/** Declaración de variables compartidas **/
var x;
var y;
var z;

/** operaciones sobre esas variables **/
function op1(...) { ... }
function op2(...) { ... }
function op3(...) { ... }
function constructor(...) { ... }
}
```

La implementación del monitor debe asegurar que nunca hay más de un proceso activo dentro del mismo. De esta forma, el programador no debe preocuparse por los requerimientos de sincronización. Sin embargo, la construcción presentada no alcanza para modelar algunos mecanismos de sincronización, por lo que se agrega los mismos lo que se llama **variables de condición**.

La únicas operaciones invocadas en una variable de condición son `signal()` y `wait()`. `wait()` suspende el proceso hasta que otro proceso llame al `signal()`. Si no hay procesos esperando, entonces `signal()` no tiene efecto. Este tipo de variables se puede implementar con semáforos.

Supongamos que definimos una variable de condición $c1$ sobre el monitor M y que un proceso P invoca $c1.signal()$. Si existe un proceso Q suspendido asociado a $c1$ entonces Q debería poder ingresar al monitor, sin embargo, P sigue estando dentro del mismo. Tenemos dos posibilidades:

1. **Signal and wait:** P se pausa y espera a que Q salga del monitor o espera a que se cumpla otra condición.

2. **Signal and continue:** Q espera a que P deje el monitor o espera a que se cumpla otra condición.

Por un lado, tiene sentido dejar que P siga ejecutandose dentro del monitor. Por otro, si permitimos esto, puede llegar a pasar que para el momento en el que Q sea activado, la condición lógica que estaba esperando deje de valer. En muchos casos, lo que se hace es hacer que la ultima instrucción que se ejecuta dentro del monitor sea un `signal()`. De esta manera, P deja la sección crítica, puede seguir su ejecución y Q puede entrar en su propia sección.

4.3.2. Deadlock

La implementación de semáforos puede resultar en una situación donde dos o más procesos se pueden quedar esperando por un evento que solo puede ser causado por otro proceso en espera. Cuando esto sucede, se dice que estos procesos están en **deadlock**.

Decimos que un conjunto de procesos está en estado de deadlock cuando cada proceso del conjunto está esperando por un evento que solo puede causado por otro proceso de ese conjunto.

Ejemplo: Supongamos que tenemos dos procesos P_0 y P_1 que hacen uso de los semáforos binarios S y Q y se produce la siguiente secuencia de comandos:

P_0	P_1	Efecto
		Ambos semáforos comienzan habilitados
`wait(S)`		P_0 continua ejecución y reserva S
	`wait(Q)`	P_1 continua ejecución y reserva Q
`wait(Q)`		P_0 se suspende hasta que P_1 libere Q
	`wait(S)`	P_1 se suspende hasta que P_0 libere S

Al final de la secuencia, ambos procesos están suspendidos porque uno necesita un recurso que tiene en el otro y el sistema entra en deadlock.

4.3.3. Condiciones de Coffman

Un sistema puede entrar en deadlock si se cumplen las siguientes condiciones de manera simultanea:

1. **Mutual exclusion**: Hay al menos un recurso que no puede ser compartido. Es decir, es recurso no puede estar asignado a mas de un proceso al mismo tiempo.
2. **Hold and Wait**: Un proceso debe tener asigado al menos un recurso y estar esperando por otro que está asignado a otro proceso.
3. **No preemption**: El sistema no implementa un mecanismo que le permita quitarle los recursos a los procesos.
4. **Circular wait**: Existe un conjunto de procesos $\{P_0, \ldots, P_n\}$ tal que P_i espera un recurso que está asignado a P_{i+1} y P_n espera un recurso que está asignado a P_0.

Livelock: Se da cuando dos o más procesos no pueden progresar porque hay otros procesos esperando para conseguir un recurso. En este caso, todos los procesos involucrados esperan para poder obtener el recurso y ninguno lo acepta.

4.4. Correctitud de sistemas concurrentes

4.4.1. Modelo del proceso

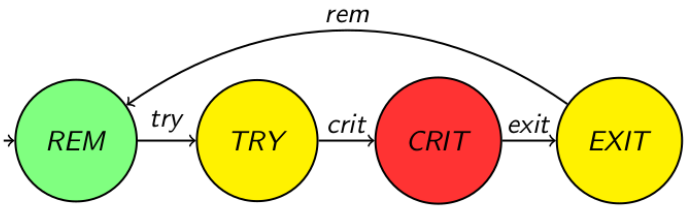

Figura 2: Modelo de un proceso según Lynch

Un proceso está definido por un aútomata finito con cuatro estados:

- **REM** es el estado en el que se encuentra cuando no está ejecutando en su sección crítica
- **TRY** es el estado en el que se encuentra cuando está ejecutando los chequeos necesarios para saber si puede entrar en su sección
- **CRIT** es cuando está ejecuntando su sección crítica
- **EXIT** cuando está ejecutando los pasos necesarios para salir correctamente de la sección crítica.

Ejecución: Es una secuencia de estados $\tau = \tau_0 \xrightarrow{l_1} \tau_1 \xrightarrow{l_2} \dots$ donde cada τ_i es un posible estado del sistema y l_i el nombre de la transición utilizada para hacer el pasaje de estados.

Una de las dificultades para demostrar correctitud sobre programas concurrentes es que tiene infinitas posibles ejecuciones. Por lo que la noción de *correcto* deja de ser únivoca y pasa a transformarse en comprobar que el sistema cumple ciertas propiedades que en conjunto deben asegurar el comportamiento deseado. Hay tres tipos de propiedades que podemos plantear:

- **Propiedades de Safety:** Aseguran que no ocurren cosas malas. Son propiedades tales que si no se cumplen entonces existe una ejecución finita del sistema en las que ocurre el evento no deseado. Por ejemplo:
  - No hay deadlocks
  - La función f nunca va devolver null

- **Propiedades de Liveness (Progreso):** Aseguran que, en algún momento, van a ocurrir cosas buenas. Por ejemplo:
  - Si se presiona el botón de stop, el tren frena.
  - Cada vez que el sistema recibe un estímulo, el sistema responde en X tiempo.
- **Propiedades de Fairness:** Los procesos ejecutandose en el sistema reciben su turno con infinita frecuencia. Es decir, los procesos que componen el sistema se ejecuntan regularmente y no son postergados para siempre.

  En general, se asume que el sistema estudiado cumple este tipo de propiedades para poder demostrar las propiedades de liveness.

4.4.2. Formalización de algunas propiedades

Fairness: Para toda ejecución τ y todo proceso i, si i puede hacer una transición l_i en una cantidad infinita de estados de τ entonces existe un k tal que $\tau(i) \xrightarrow{l_i} \tau_{k+1}$. Es decir, que si un proceso puede pasar de un estado a otro, entonces en algún momento lo va a hacer.

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

Progreso: Para toda ejecución τ, si en τ_k hay un proceso i en TRY y ningún otro proceso se encuentra en $CRIT$ entonces $\exists$ un momento $k' > k$ tal que $\tau_{k'}(i) = CRIT$.

Progreso global dependiente (deadlock-free): Para toda ejecución τ, si para todo proceso que esté en estado $CRIT$ en el momento k, en algun momento k' pasa a REM entonces, va a valer que todo proceso i' tal que $\tau_{k'}(i) = TRY$ va entrar en su sección crítica en algun momento $k'' > k'$.

Progreso global absoluto (WAIT-FREEDOM): Para toda ejecución τ, estado τ_k y proceso i, si $\tau_k(i) = TRY$ entones existe $k' > k$ tal que $\tau_{k'} = CRIT$

5. Programación concurrente (no bloqueante)

Todos los métodos de sincronización vistos hasta ahora son métodos **bloquantes** porque un delay inesperado en alguno de los threads puede bloquear el progreso de otros threads. Este tipo de delays es común en multiprocesadores en los que suelene ocurrir cache misses, page faults, cambios de contexto, etc.

5.1. Algoritmos wait-free y lock-free

Wait-Free Algorithm: Un método es *wait-free* si garantiza que termina en una cantidad finita de pasos cada vez que es llamado. Si la cantidad de pasos es acotada, entonces se lo llama *bounded wait-free*. Este tipo de algoritmos aseguran la condición de progreso no bloqueante (el delay en un thread, no neceseriamente bloquea la ejecución de otros threads.)

Population-oblivious Algorithm: Un algoritmo *wait-free* cuyo rendimiento no depende de la cantidad de threads activos.

Wait-Free Object: Un objeto tal que todos sus métodos son *wait-free*

Lock-Free Algorithms: Algoritmos que garantizan que, infinitamente amenudo, alguna llamada a un método termina en un número finito de pasos.

Esta condición es más débil que la condición de *wait-free*, por lo que cualquier algoritmo *wait-free* es un algoritmo *lock-free* pero no vale la vice versa. Los últimos admiten la posibilidad de que algunos threads sufran de inanición.

Obstruction-Free Algorithm: Son métodos que terminan en una cantidad finita de pasos si se ejecutan de manera aislada. Este tipo de métodos, obliga a pausar todos los threads que comparten el objeto/región de memoria que va a ser modificado.

Todos los tipos de algoritmos mencionados en esta sección anterior garantizan que la computación realiza progreso como un todo, independiente de como el sistema maneja los threads.

5.2. Problema ABA

Supongamos que tenemos tres threads (A, B y C) que operan sobre una cola implementada con algoritmos *wait-free*. Y se produce la siguiente situación:

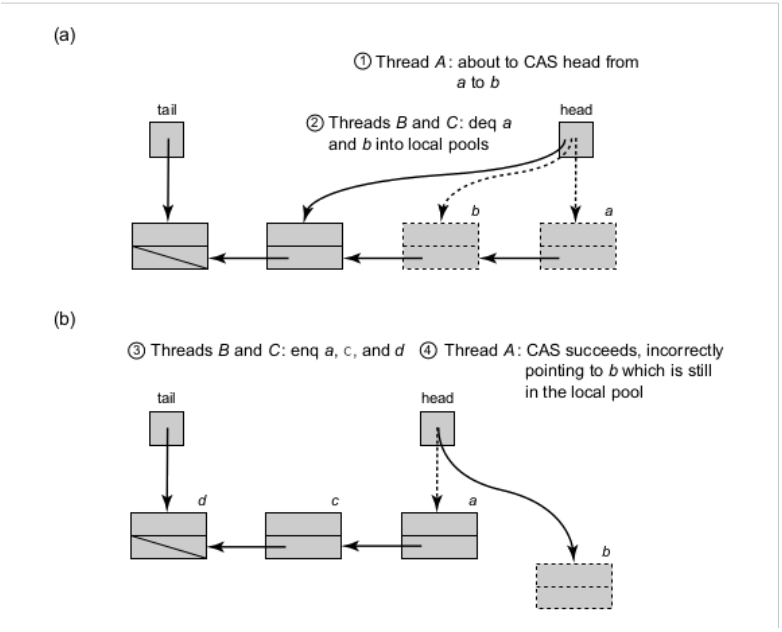

Figura 3: Problema ABA

1. El thread A necesita deseconlar el primer elemento de la cola. Lee el valor almacenado en `head` y observa que la cabeza es el nodo a y que el siguiente nodo es b. Entonces se prepara para hacer la llamada `compareAndSwap(&head, a, b)` (ver sección 4.2.2) pero es desalojado.

2. Los thread B y C desencolan los nodos a y b. y vuelven a encolar el nodo a, entonces en la cabeza queda el nodo a y c como el nodo siguiente.

3. El thread A ejecuta la llamada `compareAndSwap(&head, a, b)`. En este caso, $head = a$ por lo que la condición de la instrucción se cumple y A termina remplazando a por b. Dejando incosistente la cola.

Este tipo de situaciones se da a menudo en algoritmos que usan memoria dinámica y operaciones de sincronización condicionales. Usualmente, una referencia que está por ser modificada por `compareAndSwap` cambia de a a b y vuelve a ser a por lo que la instrucción termina exitosamente a pesar que la estructura cambió y ya no tiene el efecto deseado.

Una forma facil de resolver este problema es agregar a cada referencia atómica una estampa única que permite definir si el valor fue cambiado o no en el intervalo de tiempo entre que se llamó a la función y efectivamente se ejecuta.

Parte II
Administración de memoria

6. Espacio de direcciones

La parte del sistema operativo que maneja la jerarquía de memoria se llama **Memory Managment Unit (MMU)**. Su trabajo es mantener un registro de las partes de la memoria que están en uso, reservarla para los procesos que la necesitan y liberarla cuando ya no.

Uno de los métodos más simple de manejo memoria es permitir al programador usar la memoria física directamente.En este caso, se le presenta con un conjunto de direcciones que van desde 0 hasta algún máximo y el proceso ocupa toda la memoria disponible al momento de ejecutarse por lo que se debe esperar a que finalice antes de ejecutar otro.

En sistemas multi-usuarios, n tener varios procesos ejecutando de manera simultanea. Dado que esta situación es difícil de conseguir sin ningún tipo de abstracción, se agregó lo que se conoce como **espacio de memoria** de un proceso que es el conjunto de direcciones a las que tiene permitido acceder.

Para separar los espacios de memoria entre distintos procesos, la implementación más simple consiste en agregar dos registros que indiquen cual es la primer dirección de memoria que tiene disponible (**base register**) y cual es el rango de memoria que puede usar (**limit register**).

De esta forma, cuando el programador escribe su programa, lo hace como si la dirección de memoria más baja disponible fuese la 0000000. Cuando el proceso haga alguna referencia a memoria, el valor descrito por el programador es sumado al valor del registro base y se comprueba que la dirección resultante no supere el rango definido por el límite.

6.1. Swapping

Los métodos vistos hasta ahora permiten asignar a cada proceso un área de memoria. Si queremos ejecutar uno nuevo pero la misma está llena, debemos esperar a que uno o más procesos terminen hasta que se libere el espacio necesario para almacenar el nuevo proceso.

La técnica de **swapping** trata de resolver este problema guardando los estados de los procesos en ejecución en disco para luego poder cargar el estado de un nuevo proceso y ejecutarlo. Entonces, mientras haya espacio, se cargan y ejecutan como veníamos haciendo hasta ahora. Cuando se llena, se toma alguno de los procesos que se estén ejecutando, se guarda su estado en disco y se lo remplaza por el nuevo. A partir de aquí, cada proceso ejecuta por un tiempo determinado y luego se lo remplaza por otro que esté en la lista de espera.

En este caso no se asegura que los procesos sean cargados en el mismo área de memoria. Cada vez que se recarga uno, se deben reubicar las direcciones a las que hace referencia.

Además, como no todos los procesos ocupan el mismo espacio en memoria, cuando se realiza un *swap* pueden quedar bloques de memoria vacíos entre dos de ellos. Este efecto se llama **fragmentación externa** y es un problema porque tras varios swaps pueden quedar varios

bloques pequeños vacíos dispersos por toda la memoria. Estos bloques son inutilizables pero si estuviesen todos aglomerados en algún sector específico podrían usarse para correr otro proceso por lo que estaríamos desperdiciando memoria.

Una forma de solucionar este problema es **compactar la memoria** cada vez que se realiza un swap. Esto es, mover todo los procesos cargados al principio de la memoria, dejando todo el espacio libre al final. Sin embargo, realizar esto cada vez que se realiza un intercambio es demasiado costoso por lo que no es una técnica utilizada.

Por otro lado, si el sistema operativo ofrece la posibilidad de reservar memoria de manera dinámica entonces el área de datos de un proceso debe tener la posibilidad de crecer:

- Si la ubicación del mismo es adyacente a un bloque vacío entonces se puede agrandar su área de memoria sin problemas.
- Si no es adyante a ningún bloque, entonces el proceso se tiene que mover a un bloque lo suficientemente grande para alojarlo y, si este bloque no existe, se deberá swappear uno o más procesos para crear dicho bloque.

6.2. Manejo de memoria libre

Cuando la memoria es asignada de manera dinámica, el sistema operativo de mantener un registro preciso de que partes están ocupadas y cuales no. En general, para realizar esto, se puede utilizar alguna de las siguientes estructuras: un **bitmap** o una **lista enlazada**.

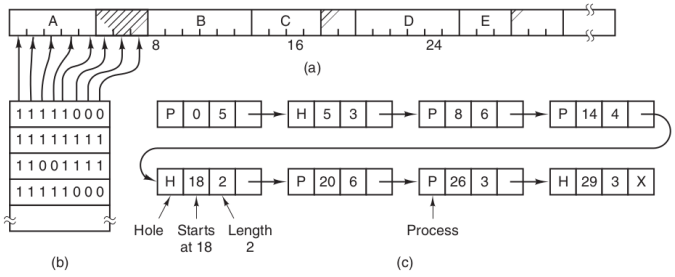

Figura 4: (a) Pedazo de memoria con cinco procesos y 3 bloques vacios. (b) Bitmap que representa el estado de la memoria. (c) Lista enlazada que representa el mismo estado.

6.2.1. Bitmaps

La memoria se divide en bloques de igual tamaño llamados **unidades de reserva** a las que se les asigna un bit en un array que tiene tantos elementos como bloques haya. Si el bit correspondiente a una unidad se encuentra en cero (0) entonces está libre. Si es uno (1), la unidad está asignada a un proceso.

El tamaño de la unidad de reserva es una decisión de diseño. Mientras más chico sea, más unidades de reservas habrá pero el bitmap se hará más grande. En cambio, si se elige una tamaños

demasiado grande, el bitmap se hará más pequeño pero podría desperdiciarse mucha memoria si los tamaños de los procesos no son múltiplos de ese tamaño (muchas unidades reservadas no serán usadas en su totalidad).

El principal problema que tiene esta estructura es que cuando se debe cargar un proceso de k unidades de tamaño, el Memory Managment Unit debe buscar de manera lineal una secuencia de k ceros consecutivos, lo que puede llegar a ser muy lento.

6.2.2. Listas enlazadas

Otra forma de representar la memoria es mantener una lista enlazada de bloques reservados y espacios libres. Cada entrada de la lista especifica si el bloque pertenece a algún proceso o si está vacío, la dirección de memoria en la que empieza, su longitud y un puntero al próximo bloque. De esta forma es facil encontrar el bloque correspondiente a cada proceso (podemos agregar un puntero en el PCB que apunte al mismo) y liberar su memoria (y combinar bloques vacios) solo implica modificar un par de valores.

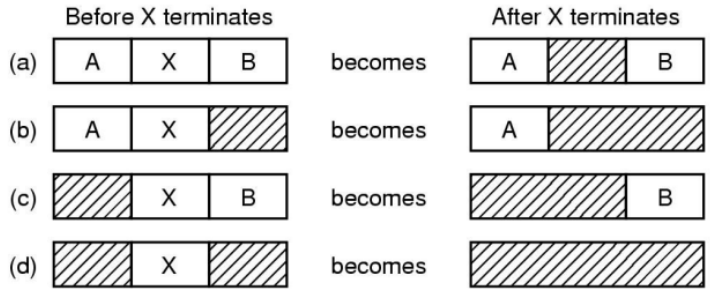

Figura 5: Cuatro combinaciones de vecinos para el proceso X

6.2.3. Algortimos para reservar memoria

Dado un proceso de tamaño K debemos asignarle un espacio de la memoria para que pueda ser ejecutado. Sabemos que el área asignada tiene ser lo suficientemente grande como almacenarlo, el problema es decidir cuál de todos los espacios válidos elegimos. Para esto contamos con los siguientes algoritmos, entre otros:

- **First Fit:** El MMU escanea la meoria hasta que encuentra un bloque lo suficientemente grande como para almacenar el proceso y ocupa la parte necesaria dejando la parte sobrante como un nuevo bloque vacío más pequeño.
- **Best Fit:** El MMU escanea toda la lista (de principio a fin) en busca de todos los bloques en los que podría caber el proceso. Luego le asigna el bloque más pequeño encontrado. En vez de partir un bloque grande, trata de que el bloque extra resultante sea lo más pequeño posible.

Este algoritmo es un poco más lento que el anterior pero tampoco soluciona el problema de fragmentación.

- **Quick Fit:** Es una variación de los algoritmos anterior en la que se mantiene una lista de los bloques disponibles ordenado por tamaño. Aquí es facil encontrar un bloque adecuado pero cuando proceso termina o se swappea es más complejo combinar bloques vacíos.

7. Memoria virtual

Si bien los registros de base y límite nos permiten abstraernos un poco de la memoria, tienen sus limitaciones: Debemos cargar el proceso completo en memoria para poder ejecutarlo. Con los tamaños del softare de hoy en dia, surge la necesidad de correr programas que no entran en memoria o que si bien entran de manera aislada cuando se lo corre en un sistema multi-usuario con otros procesos, no entran completamente en memoria.

El método diseñado para resolver el problema se conoce como **memoria virtual**. La idea es que cada proceso tiene definido un espacio de direcciones contiguo a los que puede hacer referencia. Cada una de estas direcciones está mapeada a una dirección física pero dos direcciones virtuales y deben ser traducidas para poder acceder a la información que se está solicitando.

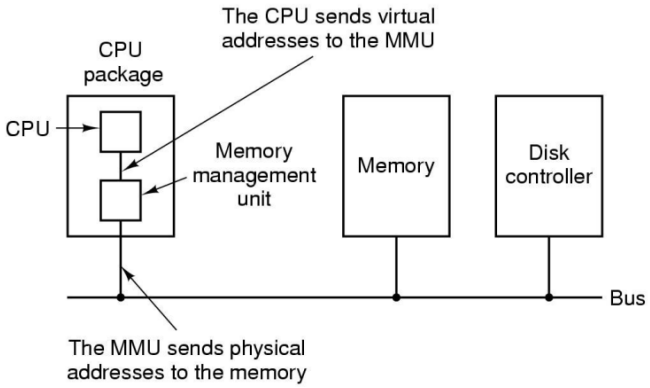

Figura 6: Funcionamiento del Memory Managment Unit

7.1. Paginación

La mayoría de los sistemas que usan memoria virtual implementan una técnica conocida como **paginación**: Cada programa tiene asignado un espacio de memoria virtual dividido en pedazos llamados **páginas**. Cada página es un rango continuo de direcciones y está mapeada a un pedazo de memoría física llamado **page frame**.

Entonces un proceso puede comenzar a ejecutar sin necesidad que todas sus páginas estén cargados en memoria. Cuando hace referencia a una **dirección virtual**, la misma se envía al MMU que se fija si el page frame correspondiente está cargado en memoria. Si lo está entonces recupera la información pedida. Si no, emite un **page fault** que es atrapado por el sistema operativo que se encarga de traer el page frame necesario para poder continuar con la ejecución.

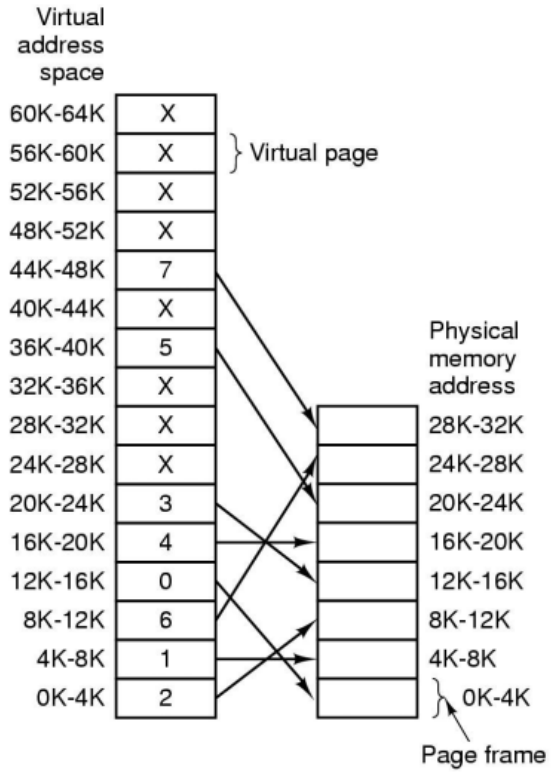

Figura 7: Ejemplo de mapeo de direcciones virtuales a direcciones físicas

7.1.1. Page tables

En esta implementación usamos una tabla para mapear direcciones virtuales a físicas. Dada una dirección virtual, la partimos en dos:

- **Número de página:** Son los bits más significativos. Nos indicara cuál es la posición de la tabla que apunta al page frame que contiene la dirección física deseada.
- **Offset:** Son los bits menos significativos. Se suman a la dirección del page frame encontrado para poder conseguir la dirección de memoria física deseada.

7.1.2. Estructura de una entrada de la tabla de páginas

Cada entrada de esta tabla tendrá la siguiente información, en la mayoría de los sistemas:

- **Número de page frame:** El page frame al que mapea una página virtual.
- **Bit de presente/ausente:** Este bit es el que usa la MMU para saber si la entrada es válida (el page frame está cargado en memoria) o no.

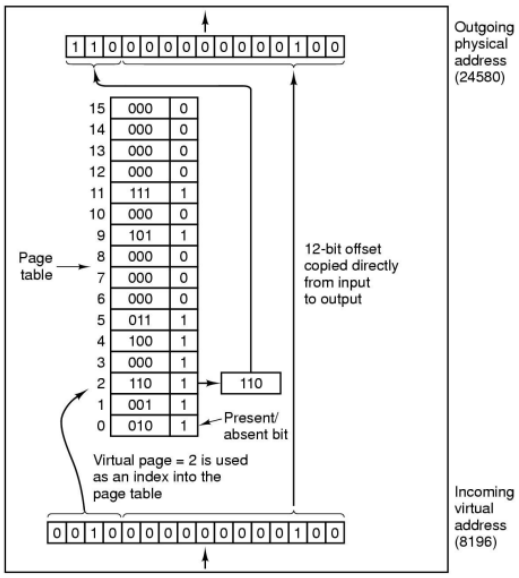

Figura 8: Conversión de una dirección virtual a una física

- **Bit de protección:** Indica que tipo de accesos son permitidos. En su versión más simple, si está activo, el proceso solo puede leer. Si no lo está entonces puede escribir y leer.
- **Bit dirty:** Indica si la página fue modificada o no por el proceso. Este bit lo usa el sistema operativo cuando necesita desalojar el page frame. Si fue modificado, debe copiarse a disco sino la podemos sobrescribir ya que la copia en disco sigue siendo valida.
- **Bit de referencia:** Indica si la página fue referenciada por el proceso (tanto para escribir como para leer). Este bit tambien es usado por el sistema operativo para ayudarlo a decidir que page frame desalojar en caso de ser necesario. Las páginas que no están siendo usadas son mejores candidatos que aquellas que fueron referenciadas últimamente.

7.2. Optimizaciones para paging

Un sitema con paginación tiene que tener en cuenta dos cosas:

- El mapeo de direcciones virtuales a físicas debe ser rápido.
- Si el espacio virtual es grande, entonces la tabla de páginas tambien lo será.

7.2.1. Translation Lookaside Buffer (TLB)

Para acelerar el mapeo de direcciones, se agregó al procesador una caché llamada **Translation Lookaside Buffer** o **Memoria asociativa** que permite realizar el mapeo sin tener que acceder a la tabla de páginas.

Cada entrada de esta cache contiene información sobre una página (su índice en la tabla de páginas, bit de protección, dirty y el page frame que le corresponde) además de un bit que indica si la entrada es válida.

Cuando la MMU recibe una dirección virtual, el hardware primero se fija si el número de página está presente en la TLB comparandolo simultáneamente con todas las entradas de la misma. Si lo encuetnra y el acceso no viola los bits de protección entonces el page frame se obtiene directamente. Si lo encuentra pero el proceso no tiene los permisos necesarios, entonces se genera un page fault.

Cuando ninguna de las entradas contiene el número de página buscado, el MMU detecta el *miss* y realiza la búsqueda en la tabla de páginas de manera normal. Una vez encontrada, borra una de las entradas de la TLB y la remplaza con la entrada de la página encontrada.

Cuando una entrada se borra de la TLB, el bit de dirty se copia a la entrada de la page table correspondiente.

7.2.2. Multilevel Page Tables

El segundo problema que teniamos que resolver era como manejar tablas de páginas muy grandes. Esto se hace usando **tablas de página multinivel**.

La idea es evitar que la tabla de páginas se mantenga completamente en memoria. En particular, aquellas entradas que no se necesitan no deberían mantenerse cargadas. Para conseguir esto, creamos un **directorio de páginas** y particionamos la tabla de páginas en partes iguales.

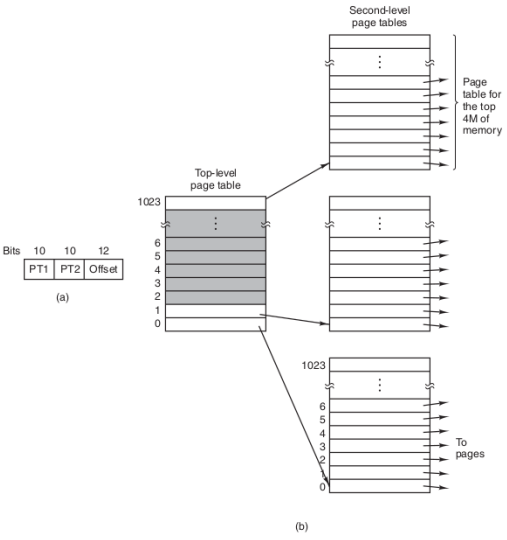

Figura 9: Descomposición de una dirección virtual en paginación multinivel

El directorio de páginas contiene información sobre como ubicar la tabla de páginas que necesitamos para mapear la dirección virtual. De esta forma, cuando la MMU recibe un dirección virtual, la divide en tres partes:

- Los primeros bits son usados como índice para ubicar la entrada en el directorio de páginas que nos permite conseguir la tabla correspondiente a esa dirección.
- La segunda parte se usa como número de página dentro de esa tabla para encontrar el page frame deseado.
- Y la última parte es el offset dentro del page frame encontrado.

La entrada del directorio de páginas es igual que la de una tabla de páginas normal. Si el bit de presente no está activado cuando buscamos una tabla determinada, se genera un page faul y el sistema operativo debe cargar en memoria la tabla correspondiente.

7.3. Algoritmos de reemplazo de páginas

El mejor algoritmo de remplazo de páginas es imposible de implementar: Remplazar la página que menos se va a utilizar en el futuro.

7.3.1. Not Recently Used (NRU)

Como dijimos, las entradas de las tablas de páginas tienen dos bits (bit dirty y referenced) que nos dan información sobre como se estuvo usando cada una de ellas. Gracias a ellos, se pueden crear algoritmos de paginación con distintos criterios que nos permitan saber como remplazar page frames a medida que se van ejecutando los procesos.

Cuando un proceso comienza su ejecución, ambos bits están desactivados para todas las entradas de la tabla. Además, el bit de referencia se desactiva periódicamente para distinguir las entradas que fueron accedidas recientemente de aquellas que no.

Cuando ocurre un page fault, el sistema operativo inspecciona todas las páginas y las divide en 4 categorías:

- Clase 0: No referenciadas, no modificadas
- Clase 1: No referenciadas, modificadas
- Clase 2: Referenciadas, no modificadas
- Clase 3: Referenciadas y modificadas

El algoritmo NRU remueve una página random de la clase más baja no vacía. En este algoritmo está implicita la ideae de que es mejor swapear una página modificad que hace rato que no se referencia, a una página no modificada que se está usando.

7.3.2. First In, Firt Out (FIFO)

El sistema operativo mantiene una lista de todas las páginas cargadas en memoria. La última página de esta lista es la ultima que fue cargada y la primer es la que más tiempo estuvo en memoria.

Cuando se realiza un page faul, se remueve el primer elemento y se encola la nueva página.

Second Chance Algorithm: Se modifica el FIFO para que se inspecciones el bit de referencia de la página más vieja. Si el bit está en cero, entonces la página es vieja y no está en uso por lo que es remplazada.

Si el bit está seteado, entonces el bit se limpia y la página se encola al final de la lista. Luego, la búsqueda continua. Si todas las páginas fueron referenciadas, entonces, se revisan todas las páginas cargadas hasta volver a empezar y se termina eliminando la que había sido la primer página analizada.

7.3.3. Least Recently Used

Una buena aproximación al algoritmo optimo se basa en la observación que las páginas que han sido más usadas en las últimas instrucciones probablemente vuelvan a ser usadas. En cambio, las páginas que no son usadas hace rato probablemente sigan sin usarse.

En este algoritmo, cuando sucede un page fault, se descarga la página que no ha sido usada durante el mayor rango de tiempo.

7.3.4. Working Set

En su forma más pura, los procesos comienzan sin ninguna de páginas en memoria. Tan pronto como el CPU trata de conseguir la primer instrucción se produce un page fault, causando que el sistema operativo traiga la página que contiene la primer instrucción. Por lo general, se producen varios page fault para traer memoria variables globales y el stack.

Después de un tiempo, el proceso tiene la mayoría de las páginas que necesita para poder ejecutar con relativamente pocos page fault. Esta estrategia se llama **demand paging** porque ninguna página se carga a no ser que sea necesaria.

La mayoría de los procesos exhiben **localidad de referencia**, esto significa que durante cualquier etapa de ejecución el proceso referencia solo una pequeña fracción de sus páginas disponible.

El conjunto de páginas que un proceso está usando se llama conjunto de trabajo (**working set**). Si este conjunto entra complentamente en memoria, entonces el proceso podrá correr realizando poco page faults. Si la memoria disponible no alcanza para almacenarlo entonces el proceso generara page faults constantemente y se alentizará su ejecución. Este efecto se llama **trashing**.

Por esta razón, muchos sistemas tratan hacer un seguimiento del working set de cada proceso y asegurarse que esté completamente en memoria antes de dejar que se ejecute. Este método se llama **working set model** y está diseñado para disminuir el ratio de page faults provocados por proceso. Cargar páginas antes de que los procesos la necesiten es una técnica llamada **prepaging**

7.4. Page fault

Estos son los pasos que realiza un sistema opertivo una vez que el MMU emite un page fault:

1. El hardware emite el page fault que es atrapado por el sistema operativo.
2. Se guarda el program counter de la instrucción fallida y el estado del proceso.

3. El sistema operativo se fija cual es la página que necesita el proceso.
4. El sistema operativo controla que la dirección pedida sea válida y que la protección de la misma sea consistente con el acceso. Si no lo es, se manda una señal al proceso o se lo mata. Si la dirección es válida y no hay un protection fault, el sistema se fija si hay page frames libres. Si no los hay, se usa alguno de los algoritmos de remplazo de páginas.
5. Si el page frame a desalojar está modificado, la página se marca para transferir a disco. Se realiza un context switch suspendiendo la rutina de interrupción y dejando que otro proceso se ejecute mientras se completa la transferencia.
6. Una vez completada la transferencia, el sistema operativo, busca la dirección del disco en la que está almacenada la página y la marca para cargarla en memoria. Mientras sucede esta transferencia se vuelve a suspender la rutina y se deja ejecutar a otro proceso.
7. Despues de que la página está cargada en memoria, se actualizan las tablas de página para reflejar los cambios y la entrada se marca como válida.
8. El proceso vuelve a entrar en la cola del scheduler y el estado se reseta a como estaba antes de ejecutar la instrucción que generó el page fault.

8. Segmentación

La memoria virtual discutida hasta ahora es unidimensional: A cada proceso se le asigna un conjunto de direcciones que van desde 0 hasta una dirección máxima. Cuando el compilador genera un proceso divide este espacio en partes de uso espécifico (por ejemplo, stack, tabla de constantes, tabla de símbolos, código fuente, etc).

Sin embargo, esta division puede llegar a traer problemas. En la figura **??**, por ejemplo, la tabla de símbolos se llenó y no puede seguir creciendo porque las próximas direcciones contienen el código del programa.

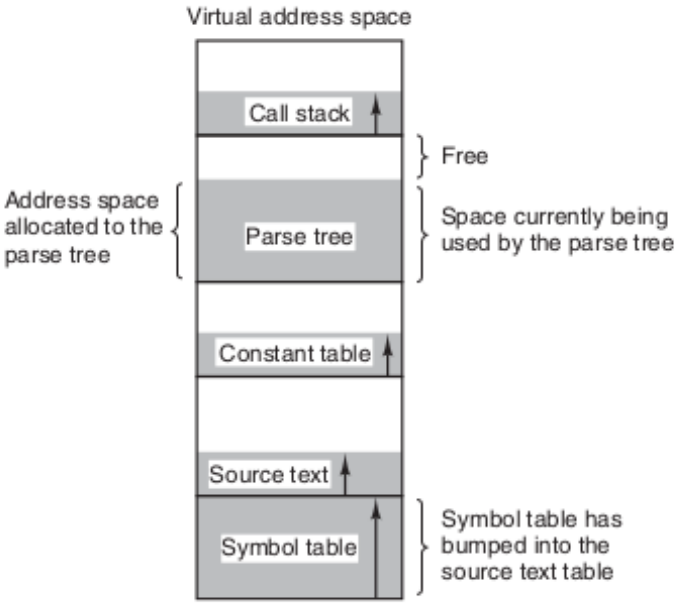

Figura 10: Organización de un proceso con memoria virtual

Una solución a este problema es proveer al proceso de varios espacios de memoria virtuales completamente independientes llamados **segmentos**. Cada segmentos consiste en una secuencia lineal de direcciones que van desde cero hasta algún valor máximo y como son independiente cada uno puede crecer o contraer sin afectarse entre si.

Para especificar una dirección ubicada en un segmento, el programa debe enviar una dirección que se divide en dos partes: Un número de segmento y la dirección virtual dentro de ese segmento.

Esta técnica además facilita la implementación de librerias compartidas. Dado que cada fragmento es una entidad lógica que el programador sabe que contiene, cada uno de ellos puede tener distinto tipos de protección.

En muchos sitemas es común encontrar la segmentación implementada con paginación para segmento.

8.1. Shared Libraries

En sistemas multiprogramables, es comun tener varios usuarios corriendo el mismo programa de manera simultánea. Incluso un único usuario puede estar corriendo varios programas que hagan uso de las mismas rutinas. Cuando esto sucede, los segmentos de código de cada proceso apuntan a la misma tabla de páginas (como el código es solo lectura, ambos procesos pueden accederlas sin problemas).

Sin embargo, hay que tener en cuenta que cuando dos o mas procesos comparten las páginas y uno termina, no debemos desalojar las páginas compartidas ya que cuando se vuelva a ejecutar el otro proceso, se generarian varios page faults hasta que todas las paǵinas desalojadas vuelvan a estar en memoria. Por esta razón, debemos mantener una estructura que nos permita identificar rápidamente cuales son las páginas compartidas para evitar esta situación.

8.1.1. Copy On Write

Tambien se puede compartir páginas que contenga datos aunque hay que tener algunas consideraciones más. En los sistemas UNIX, en particular, cuando se realiza una llamada a `fork()`, el proceso padre comparte todas sus páginas con el proceso hijo pero cada uno tiene su propio conjunto de tablas de páginas. Sin embargo, en ambos procesos se marca a todas las páginas como *Read Only*.

Mientras ambos procesos solo hagan lecturas, la situación se mantiene. Tan pronto como algunos de los dos necesita realizar una modificación, se genera un protection fault que es interceptado por el sistema operativo. El mismo realiza una copia de la página que se quiere modificar y ahora cada proceso tiene su propia copia de la información con acceso de escritura/lectura. Así, las próximas modificaciones a esa página suceden sin interrupciones. Esto implica que la información que las páginas que no son modificadas nunca no necesitan ser copiadas.

Esta técnica es conocida como **copy on write**.

Parte III
Administración de entrada/salida

9. Subsistema de I/O

Los elementos básicos de hardware, como puertos, buses y controladores se conectan una gran variedad de disposivitos de entrada/salida que varian en sus funciones y velocidades por lo que se necesitan varios métodos para controlarlos. Éstos forman parte del **subsistema de I/O** y son encapsulados en módulos llamados **drivers**.

Los drivers presentan una interfaz de acceso uniforme a los subsitemas de entrada/salida, eliminando la necesidad de que el sistema operativo sepa como manejar las particularidades de cada dispositivo. Los mismos corren con máximo privilegio y de ellos depende el rendimiento de todo el subsistema de entrada/salida

La mayoría de los dispositivos que se usan en una computadora pueden ser clasificado como de almacenamiento (discos, memorias flash), de transmisión (conexiones al internet, bluetooth) e interface de usuarios (pantallas, teclados, mouse).

Puerto (Port): Es un cable mediante el cual se comunican los dispositivos de entrada/salida con el sistema

Bus: Es un conjunto de cables compartidos por varios dispositivos junto con un protocolo específico que define un conjunto de mensajes que pueden ser enviados a los mismos.

Controlador: Es un dispositivo que puede operar un puerto, un bus u otros dispositivos. Algunos controladores pueden llegar a tener un pequeño procesador, microcodigo y memoria privada integradas para implementar protocolos complejos. Un ejemplo de esto son los controladores **Small Computer System Interface (SCSI)** que se encargan de comunicar los discos con el sistema

9.1. Interacción con los dispositivos

La interacción entre el sistema y un controlador puede tomar varias formas. La más simple es conocida como **polling** o **busy-waiting**. Con este método, el sistema está constantemente verificando el estado de un dispositivo hasta que está listo para ser usado. Una vez que el dispositivo está listo, le manda un pedido y vuelve al ciclo de verificación hasta que el dispositivo marca que el pedido fue atendido.

Esté método tiene sentido en dispositivos que deben ser atendidos rápidamente. Por ejemplo,el buffer de los controladores de teclados es muy pequeño y podríamos perder información si el sistema tarda demasiado en leer los bytes del mismo.

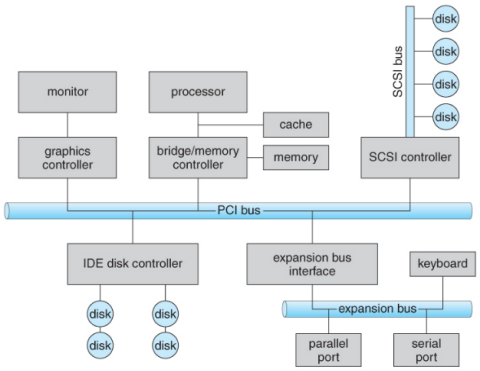

Figura 11: Estructura típica de un Bus

Sin embargo es ineficiente cuando el dispositivo se usa poco y hay otros procesos que deben terminarse. En estos casos es más eficiente configurar los controladores para que notifiquen al CPU cuando el dispositivo está listo para atender un pedido. El mecanismo que permite implementar esto es llamado **interrupciones**.

9.1.1. Interrupciones

El CPU tiene un cable llamado linea de pedido de interrupciones (**interrupt-request line**) que se chequea cada vez que se termina de procesar una instrucción. Cuando se detecta una interrupción en este cable, el sistema determina cuál fue la causa, realiza el procesamiento necesario para atenderla y luego sigue ejecutando los procesos del usuario.

9.1.2. Direct Memory Access

Para dispositivos que realizan transferencias de grandes cantidades de datos, como discos rígidos, se usa un procesador especial llamado controlador de acceso directo a memoria (**Direct Memory Access Controller**).

El sistem debe avisarle al controlador DMA que desea realizar una transferencia y le pasa un puntero a la data que quiere transferir, un puntero a donde quiere hacerlo y la cantidad de bytes que deben ser transferidos. El controlador procede a realizar la transferencia mientras el CPU pasa a otra tareas por lo que no es necesario esperar a que termine la transferencia para seguir trabajando.

9.2. Drivers

Podemos abstraer las diferencias especificas entre los dispositivos de entrada/salida clasificandolos de una manera más generalizada. Cada clase es accedida a través de una conjunto estandarizado de funciones (una **interfaz**) que debe ser implementada por los drivers de cada dispositivo.

Las interfaces presentadas por el sistema deben en tener en cuenta las siguientes características de los dispositivos:

- **Tamaño de transferencia** (character-stream or block-steram): Los dispositivos **character-stream** transfieren byte a byte, mientras que los dispositivos de transferencia bloques transmiten una unidad de información compuesta de varios bytes.
- **Acceso secuencial o aleatorio**: Un dispositivo secuencial transfiere información en un orden determinado por el dispositivo, mientras que los usuarios de un dispositivo de acceso aleatorio pueden pedir que se busque cualquier porción de la memoria disponible.
- **Sincróno o asincróno:** Un dispositivo síncrono realiza las transferencia con un tiempo de respuesta predecible y coordina con otros aspectos del sistema. Un dispositivo asíncrono exhibe tiempos de respuesta irregulares que no se pueden coordinar con otros eventos del sistema.
- **Compartido o dedicado:** Un dispositivo compartido puede ser usado por varios procesos o thread de manera concurrente, mientras que uno dedicado no. **Velocidad de operación:** Puede variar de uno pocos bytes por segundo a varios gigabytes por segundo, dependiendo el dispositivo. **Tipo de comunicación:** Puede ser dispositivos de solo escritura, de solo lectura o de escritura/lectura.

9.2.1. Block devices

La interfaz de los dispositivos de bloque capturan todos los aspectos necesarios para acceder a dispositivos de almacenamiento. Los comandos básicos que se espera que respondan son: `read()` y `write()`. Además de la operación `seek()`si el dispositivo permite acceso aleatorio.

Sin embargo, hay aplicaciones especiales (como los motores de bases datos) que pueden preferir acceder a los dispositivos como si fuesen un arreglo lineal de bloques (**raw I/O**) debido a que implementan sus propios buffers o técnicas de escrituras. Por estar razón, se permite a esos programas realizar los accesos directos y evitar el comportamiento default del sistema operativo.

9.3. Spooling

Su nombre proviene de **Simultaneous Peripheral Operation On-Line** y consiste en un buffer que almacena información que se debe enviar a un dispositivo, como una impresora, que no puede aceptar información de manera intercalada. Aunque una impresora puede ejecutar un solo trabajo por vez, varias aplicaciones pueden requerir su uso de manera concurrente y es necesario que sus salidas no se mezclen. El sistema operativo resuelve este problema interceptando todos los pedidos y enconlandolos en este buffer. Cuando la impresora está lista para responder un nuevo pedido, el sistema copia el siguiente archivo de la cola.

Hay que tener en cuenta que este proceso es visible a nivel usuario y es manejado por el sistema operativo. El kernel y los controladores actuan de manera normal.

10. Almacenamiento Secundario

10.1. Tipos de discos

10.1.1. Discos Magnéticos

Son la mayor parte del sistema de memoria secundario en las computadoras modernas. Un disco rígido, consiste en un conjunto de platos (**platters**) con una forma circular y varios cabezales de lectura-escritura (**read-write head**) que "vuelan"sobre su superficie:

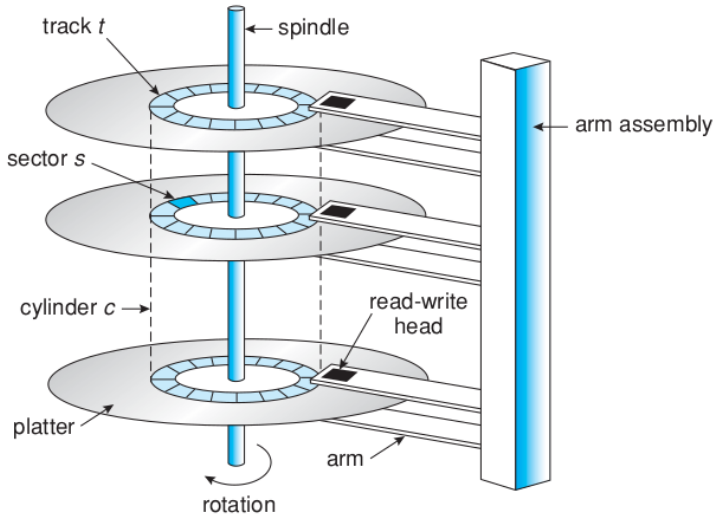

Figura 12: Mecanismo de lectura/escritura de un disco

Todos los cabezales están unidos a un brazo que los mueve de manera unísona. Cuando un disco está en uso, un motor gira sus platos a alta velocidad y el brazo mueve los cabezales de manera radial (desde el centro del plato hasta el perímetro del mismo y viceversa).

La superficie de cada plato está divida en sectores lógicos llamados **tracks** que están subdivido en **sectores**. El conjunto de **tracks** que estan a la misma distancia del brazo se llama **cilindro**. Cada disco puede tener miles de cilindros y cada track cientos de sectores por lo que en general la capacidad de estos dispositivos se mide en gigabytes.

La velocidad de un disco se divide en dos partes:

- **Velocidad de transferencia (Transference Rate):** La velocidad a la que la infromación viaja entre el disco y la computadora.
- **Tiempo de posicionamiento (random-acces time):** Que es el tiempo que tarda el disco en encontrar un sector determinado. Este tiempo se puede dividir en dos partes:
  - **Tiempo de búsqueda (seek time):** El tiempo necesario para mover el cabezal al cilindro adecuado.

- **Latencia rotacional (Rotational Latency):** El tiempo necesario para que el sector llegue al cabezal ya posicionado

Generalmente, los discos pueden transferir varios megabytes de datos por segundo y los tiempos de búsqueda y de giro toman varios milisegundos.

Un disco puede ser removible permitiendo que varios discos se vayan montando y desmontando a medida que sea necesario. Algunas formas de discos removibles incluyen: CDs, DVDs y Blu Rays y memorias **flash**.

10.1.2. Discos de estado solido (SSD)

Un disco de estado sólido es una memoria no volátil que es usada como disco rígido. Tienen las mimas características que los discos clásicos pero son más seguros porque no tienen partes mobiles y son más rápidos por queno tienen latencia y ni tiempo de búsqueda. Además, consumen menos energía. Sin embargo, son más caros que los discos tradicionales, tienen menos capacidad y tiempos de vida más cortos.

Otro problema que tiene este tipo de discos es llamado **write amplification**, en el cual la cantidad de información escrita en el disco es un múltiplo de la cantidad lógica que se intentó escribir. En las memorias flash, es necesario borrar la memoria antes de poder escribirla pero las operaciones de borrado tienen una granularidad bastante mayor a la de escritura. Entonces, el proceso de escribir a memoria, muchas veces implica mover la sección usada de un bloque a alguna ubicación no usada del disco y luego borrar todo el bloque para luego poder escribir en la ubicación deseada.

Osea que una escritura puede requerir que se lea, actualize y reescriba alguna parte de la memoria ya utilizada en una nueva ubicación. Borrar la ubicación inicial y luego escribir los datos que se desean guardar. Este efecto multiplicador aumenta la cantidad de escrituras necesarias en el disco, lo que acorta el tiempo que puede funcionar de manera confiable.

10.1.3. Cintas Magneticas

Fueron de los primeros medios de almacenamiento utilizados. Aunque tienen una larga vida útil y tienen una gran capacidad de almacenamiento, sus tiempos de acceso son muchos mas lentos que la de los discos magnéticos.

10.1.4. Almacenamiento virtual (Network-Attached Storage - NAS)

Un dispositivo de almacenamiento virtual es sistema de almacenamiento que se accede remotamente desde una red de datos. Los dispositivos y sus usuarios están conectados a una red y se comunican mediantes paquetes TCP o UDP. Los clientes envian las las llamadas a los procedimientos que se deben realizar en algún dispositivo y luego los dispositivos le responden con la información pedida.

Este tipo de almacenamiento es una forma conveniente de compartir archivos y datos entre todas las computadoras de una red LAN. Sin embargo, tiende a ser menos eficiente que los

dispositivos conectados fisicamente a cada computadora.

10.1.5. Storage-Area Network (SAN)

Son redes privadas que implementan protocolos específicos de almacenamiento y sirven para conectar servidores con unidades de almacenamiento. Estos protocolos, incluyen como repartir la memoria disponible entre multiples hosts y las políticas de uso de cada dispositivo de almacenamiento, etc.

10.2. Politicas de Scheduling de E/S a disco

Una de las respobilidades del sistema operativo es usar el hardware de manera eficiente. Para los discos magnéticos, esto es tener la responsabilidad de tener un tiempo de acceso rápido y conseguir un gran ancho de banda.

Ancho de banda: Es el número total de bytes transferidos divido el tiempo total transcurrido desde el primer pedido del servicio hasta que se termina de completar la transferencia.

Cuando un proceso necesita realizar una operación de I/O en el disco, realiza una llamada al sistema operativo que especifica:

- Si la operación es de entrada/salida.
- La dirección del disco desde con la que se debe realizar la transferencia
- La dirección de memoria principal a usar para la transferencia
- La cantidad de sectores que se deben transferir

Si el disco deseado y el controlador están disponibles, entonces el pedido se puede atender inmediatamente. Si alguno de los dos están ocupados, entonces se debe encolar en una cola de pedidos pendientes para ese disco.

Una vez que el disco y el controlador están libres, el sistema operativo debe elegir cuál pedido atender:

10.2.1. First-come, first-served (FCFS)

El algoritmo más simple. Es intrisicamente justo pero no provee el servicio más rápido. Ya que dependiendo el orden en el que lleguen los pedidos puede ser que sea necesario mover el cabezal a través de todo el disco:

10.2.2. Shortest Seek Time First (SSTF)

Se atienden primero los pedidos más cercanos al cabezal de escritura/lectura. Este algoritmo provee una sustancial mejora en el rendimiento del disco pero puede llegar a causar inanición de algunos pedidos, un escenario que se vuelve cada vez más probable a medida que la cola de espera crece.

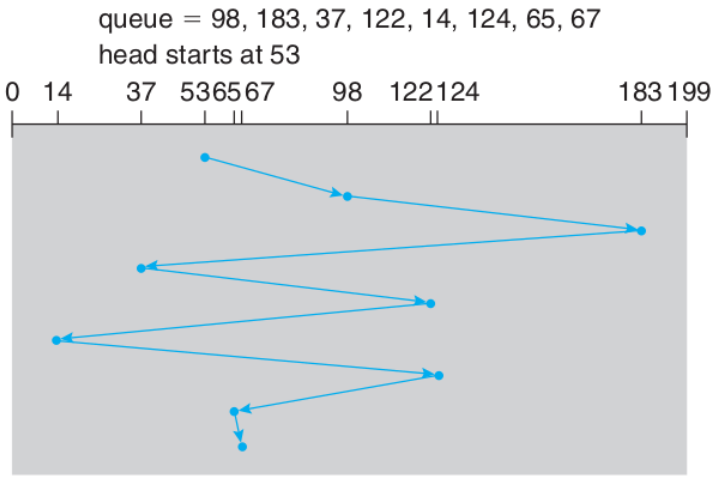

Figura 13: Recorrido del cabezal para algoritmo FCFS: El cabezal va desde el cilindro 53 hasta el 98, luego al 183, 37, 122, en orden hasta atender el último pedido en el cilindro 67. Se puede ver que si, en vez de haberlos hechos de esta manera, se hubiesen atendido primero el 37 y el 14 y después el 122 y el 124 también se hubiese reducido la distancia recorrida por el cabezal

10.2.3. SCAN

El brazo del disco comienza en un extremo del disco y se mueve hacia el final respondiendo pedidos a medida que va alcanzando cada cilindro. Una vez que termina de recorrer el disco, el brazo vuelve a repetir el proceso pero en la dirección inversa.

Si un pedido que esté justo enfrente del cabezal llega a la cola, entonces es resuelto inmediatamente. Si el cabezal ya pasó por ese sector, se deberá esperar a que el mismo llegue hasta el final del disco y vuelva hasta el sector correspondiente.

10.3. Gestión del disco

El sistema operativo también es responsable de manejar otros aspectos del disco:

10.3.1. Formateo

Antes de porder guardar información en un disco, se lo debe dividir en sectores que el controlador del disco pueda leer y escribir. Este proceso es llamado **formateo de bajo nivel** o **formateo físico**. El formateo llena el disco con una estructura de datos especial para cada sector que consiste en un header, un área de datos y un *posfijo*.

El header y el posfijo contienen información como el número del sectory codigo de corrección de errores. El controlador del disco actualiza esta información cada vez que realiza una escritura. Y la usa para comprobar que no haya fallas en ese sector cuando necesita leer del mismo. Si hay

alguna falla, entonces puede usar el código de corrección de errores para recuperar la información corrupta. Sin embargo, esto solo es posible si solo fueron corrompidos uno pocos bits.

10.3.2. Booteo

Cuando una computadora se enciende, necesita que se ejecute un programa llamado **bootstrap** que inicializa todos los aspectos del sistema (desde los registros de la CPU, los controladores de los dispositivos. el contenido de la memoria principal hasta el sistema operativo).

En la mayoria de las computadoras, se guarda un cargador de este programa (**bootsrap loader**) en una área de memoria de solo lectura (**read only memory - ROM**). El bootstrap loader se encarga de leer el programa de bootstrap desde disco. El mismo está ubicado en los bloques de inicialización (**boot blocks**) que están en una posición fija.

10.3.3. Bloques dañados

Como los discos tienen partes movibles y tolerancias bajas, son propensos a fallas. En caso de que el fallo sea completo, el disco debe ser remplazado y no será posible recuperar su contenido a menos que se cuente con un disco de backup.

En el caso de que solo haya sectores defectuosos, se pueden utilizar varias estrategias para manejarlos:

- La más simple de todas, es escanear el disco para encontrar aquellos bloques defectuosos y marcarlos como inservibles para que el sistema operativo no los reserve.
- Otra técnica más sofisticada, es hacer que el controlador del disco mantenga una lista de los bloques defectuosos. Esta lista se inicializa durante el formateo de bajo nivel durante la fabriación y es actualizada constatemente durante la vida útil del disco.

  En este caso, el formateo genera sectores extra que no son visibles al sistema. De esta manera, cuando un bloque se daña, puede ser remplazado por uno de estos sectores sin mayores problemas.

10.4. RAID

Gracias a que los discos son cada vez mas baratos, es cada vez más factible tener varios discos en un sistema, lo que nos da la oportunidad de mejorar la velocidad de transferencia del sistema, y la seguridad de la información.

La solución para disminuir la probabilidad de perdida de datos, es agregar **redundancia**. Es decir, guardamos información que no es necesaria pero que puede usar en caso de que un disco falle para reconstruir la informción perdida.

La forma más simple (y más cara) de lograr esto es utilizar la técnica de **mirroring** que consiste en duplicar los datos de cada disco. En este caso, cada vez que se escribe algo a memoria, la operación se realiza en ambos discos por igual. Si algún disco falla, los datos pueden ser leídos del otro disco hasta que el disco dañado sea remplazado por uno nuevo.

Para mejorar la velocidad de transferencia se puede distribuir los bloques de un mismo archivo entre los discos. Está técnica es conocida como ***data stripping***. Cada disco participa en cada acceso a memoria lo que permite multiplicar la cantidad de información enviada por cada acceso de memoria.

Entonces **mirroring** provee alta seguridad de los datos pero es caro y **stripping** nos permite alcanzar altas velocidades de transferencia pero no mejora la seguridad. Por esta razón, se diseñaron varios esquemas que combinan estás dos técnicas de distintas maneras para lograr buenos trade-offs entre seguridad y ancho de banda.

Este conjunto de esquemas son conocidos como **Redundant Arrays of Independent Disks (RAID)** y están classificados en niveles:

- **RAID 0 (Stripping):** Los bloques de un mismo archivo se distribuyen en dos (o más discos). No aporta ninguna redundancia, pero permite escrituras en paralelo lo que mejora el ancho de banda.
- **RAID 1 (Mirroring)**: Las escrituras se realizan en dos (o más) discos de manera simultanea. En el mejor de los casos tardan lo mismo que si tuviesemos un solo disco, en el peor tarda el doble. Mejora el rendimiento de las lecturas pero es demasiado caro de implementar.
- **RAID 2:** Tambien conocido como **memory-style error-correcting-code organizatio n(ECC)**. Se realiza un stripping a nivel bit de los datos que se guardan en discos distintos. Los códigos de corrección de error se guardan en discos aparte. Si un disco falla, entonces se puede usar el resto de los bits del archivo desde los otros discos y junto con su código de corrección correspondiente para recuperar la información perdida.
- **RAID 3:** Mejora el RAID 2 teniendo en cuenta que los controladores de disco pueden detectar si un sector fue leído correctamente por lo que un único bit de paridad puede ser usado tanto como para corregir como para detectar daños. Si uno de los sectores está dañado, sabemos que sector es y podemos calcular el valor del bit perdido a partir de la paridad de los sectores en los otros discos. Este tipo de RAID es igual de efectivo que el RAID 2 pero es más barato de implementar.

  Este nivel, es mejor que el nivel 1 porque se reduce la cantidad de discos necesarios para conseguir redundancia. Sin embargo, soporta menos operaciones de entrada/salida por segundo ya que todos los discos deben participar en cada transferencia. Además, puede llegar a ser muy caro computar la paridad de cada bloque.
- **RAID 4:** O **block-interleaved parity orgnaization** usa stripping a nivel de bloque, como raid 0, pero mantiene un disco a parte en el que se guarda bloques de paridad por cada bloque de los otros discos. Si un disco falla, se puede recuperar el bloque dañado usando los bloques del resto de los discos junto con el bloque de paridad.

  Un lectura accede solo un dico, por lo que es posible atender varios pedidos de manera simultanea por lo que el ancho de banda para lecturas es alto. Sin embargo, cuando se

desean realizar escrituras de tamaños pequeños, es necesario cargar en memoria todo el bloque, modificarlo y luego volver a escribirlo en disco, además de actualizar el bloque de paridad. Esto puede alentizar el sistema.

- **RAID 5:** O **block-interleaved distributed parity** es el RAID 4 con la diferencia que los datos y los bloques de paridad se reparten entre todos los discos. Esto aligera la carga de los discos que hubiesen sido usados como discos de paridad, alargando su vida útil.
- **RAID 6:** Es como el RAID 5 pero almacena más información redudante para proteger de fallas de múltiples discos.
- **RAID 0 + 1:** Es una combinación de los RAID 0 y RAID 1. Se realiza un stripping de la data entre varios discos y luego se duplican esa escritura en otro conjunto de discos. Generalmente, tiene mejor performance que el RAID 5 pero duplica la cantidad de discos necesarios para implementarlo.

Por lo general, junto con estos esquemas se agrega un disco de respuesto que no es usado hasta que un disco falle completamente. Una vez que esto sucede, se reconstruye en el mismo la data perdida. Además RAID solo protege de errores físicos en los discos, no de errores de otros hardware o software por lo que es necesario tener copias de seguridad y sistemas de archivos que brinden protección extra.

11. Sistemas de archivos

El sistema de archivos (filesystem) es el módulo del kernel que se encarga de organizar la información de manera lógica. Es decir, se encarga de estructurar la información guardad en cada archivo y de como ordenarlos en su conjunto.

11.1. Archivos

Un archivo es un mecanismo de abstracción que provee una forma de guardar información en el disco. Nos permiten abstraer los detalles del método usado guardar la información deseada y como funciona el disco.

Cada archivo es identificado por un nombre que puede consistir en dos partes separadas por un punto. La segunda parte se conoce como la **extensión** del archivo y usualmente indica como está estructurado internamente. Sin embargo, el nombre (junto con la extensión) solo das a los usuarios una forma de distinguir fácilmente el tipo de contenido guardado en el mismo. Para el sistema operativo, el archivo es simplemente una secuencia de bytes sin sentido.

Esto nos da la máxima flexibilidad posible, ya que un usuario puede poner lo que quiera en sus archivos, de la forma que mejor le parezca. El sistema operativo no ofrece ninguna ayuda pero tampoco estorba.

Por lo general, los sistemas operativos soportan varios tipos de archivos. En los sistemas UNIX podemos distinguir los siguientes:

- Los **archivos regulares** son los archivos que contienen la información del usuario.
- Los archivos de **directorios** son los archivos que usa el sistema para mantener la estructura del filesystem.
- Los archivos de carácter especial (**character special files**) son usados para modelar dispositivos de entradas/salida como terminales, impresoras y redes.
- Los archivos especiales de bloques **Block Special Files** son usados para modelrar discos.

11.1.1. Atributos

Cuando un usuario guarda un archivo, el filesystem, además de asociar el nombre con los datos correspondientes almacena varios atributos que le permitirán operar al sistema en el futuro. Estos atributos son conocidos como los **metadatos** del archivo y son almacenados en las estructuras de directorio.

Algunos ejemplos de metadatos son:

- **Flags de protección:** Indican que operaciones podrá realizar cada usuario o grupo de usuarios sobre el archivo.
- **Tamaños**: El tamaño actual del achivo y, si está definido, el tamaño máximo permitido.
- **Propietario**: Un identificador del usuario que creó el archivo

- **Fechas de creación, modificación y acceso**
- **Bits de archivado**: Indica si hay alguna copia e seguridad del archivo.
- **Tipo de archivo**: Si es regular, directorio, representa algun dispositivo, etc.
- **Conteo de referencias**
- **CRC**

11.2. Estructuras de directorios

Volumen: Una partición de un dispositivo, un dispositivo entero o un varios dispositivos que participan en un RAID que contienen un sistema de archivos. Cada volumen puede ser pensado como un disco virtual y es capaz de contener uno o más sistemas operativos.

Cada volumen, además de contener un filesystem debe almacenar la información sobre los archivos que contiene el sistema. Una de las formas más simples de mantener esta información es en una tabla llamada **directorio del dispositivo** o **tabla de contenidos del volumen**. La misma puede ser vista como una tabla que traduce nombres de archivos en sus entradas de directorio correspondientes (es decir, a punteros que indican donde están los metadatos del mismo).

Hoy en día, la mayoría de los sistemas operativos utiliza directorios estructurados en forma de árbol. Los mismos establecen una jerarquía de directorios que permiten al usuario crear sus propios subdirectorios y organizar sus archivos como mejor le parezca.

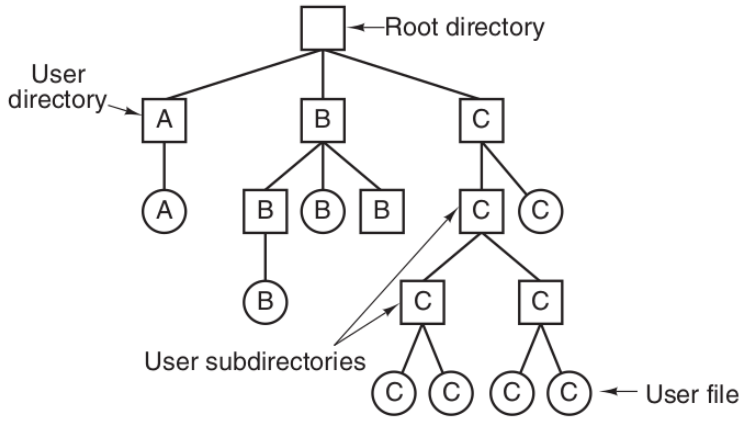

Figure 4-7. A hierarchical directory system.

11.2.1. Directorios de grafos áciclico (DAG)

Consideremos dos porgramadores que están trabajando en el mismo projecto. Los archivos asociados con ese proyecto son guardados en un subidirectorio, separandolos así de otros projectos

y archivos de cada uno de los programadores.

Un directorio con estructura de gráfo acíclico es una generalización a uso de un árbol que permite que se compartan archivos y directorios.

En este esquema, el mismo archivo o subdirectorio puede estar en dos directorios distintos de manera simultánea. Entonces se dice que ese archivo/subdirectorio es **compartido**. Es importante notar, que un archivo compartido no es lo mismo que tener dos copias del mismo archivo. Con dos copias, si un programador cambia su copia, el otro no verá los cambio en la suya. Con un archivo compartido, los cambios hechos por una persona son inmediatamente visibles por el otro.

En sistemas UNIX, este tipo los archivos y subdirectorios compartidos se crean agregando una nueva entrada de directorio al sistema llamado **link simbólico** que apunta al archivo o subdirectorio en cuestión.

Cuando se hace una referencia a un archivo, buscamos en el directorio. Si la entrada del directorio está marcada como link, entonces debemos resolverlo usando el nombre real del archivo al que apunta.

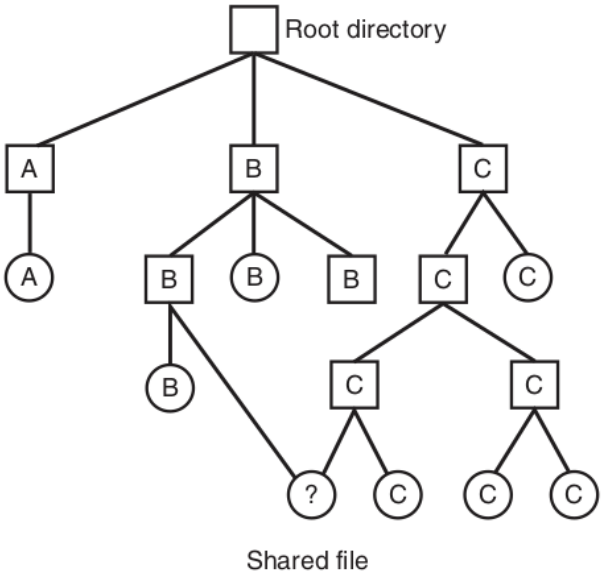

Figure 4-16. File system containing a shared file.

11.3. Punto de montaje

Así como hay que abrir un archivo antes de usarlo, un sistema de archivo debe ser **montado**. Mas específicamente, la estructura de directorio del sistema operativo se construye a partir de múltiples volumenes que deben ser montados para que estén disponibles al usuario.

El proceso de montaje es simple. Primero, se envia al sistema operativo el nombre del dispositivo y un **punto de montaje** (la ubicación dentro de la estructura de archivos a través de la cual el usuario podrá acceder al sistema de archivos). Algunos sistemas operativos tambien requieren el tipo de filesystem que va a ser montado, mientras que otros analizan la estructura del dispositivo para determinarlo. Por lo general, el punto de montaje es una directorio vacío.

Luego, el sistema operativo verifica que el dispositivo contiene un sistema de archivos válido y luego marca el punto de montaje como forma de acceso al mismo. Este esquema permite al usuario recorrer de manera uniforme diferentes sistemas de archivos.

11.4. Representación de archivos

El sistema de archivos maneja toda la metadata (atributos - ver 11.1.1) relacionada con los archivos y las estructuras de los directorios. Esta información está almacenada en bloques de control de archivos (file-control block).

11.4.1. Representacion continua

Una de las formas más fáciles de almacenar los archivos es guardandolos en bloques continuos en el disco. En este caso, los file-systems tienen que almacenar su file-control block la dirección del primer bloque y la cantidad de bloques ocupados por el archivo. Sin embargo, es un problema encontrar el espacio necesario para cada uno de ellos, es difícil saber cuanto espacio necesitara en el futuro atributos-archivos y produce fragmentación externa.

11.4.2. Representacion como lista enlazada

Soluciona todos los problemas de la representación continua. En este esquema, cada archivo es una lista enlazada de bloques de disco. El directorio contiene un puntero al primer y último bloque del archivo.

Para crear un archivo nuevo, simplemente creamos una entrada nueva en el directorio con sus punteros en `null`. Una escritura en el disco, hace que el sistema de manejo de memoria libre encuentre un bloque vacío.

Sin embargo, el mayor problema con esta representación es que el acceso aleatorio es ineficiente.

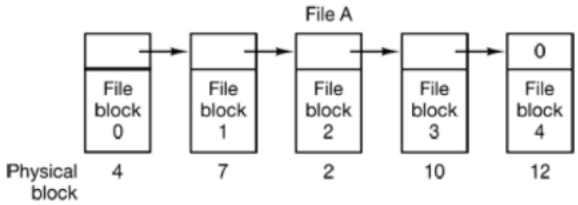

Figura 14: Archivo como lista enlazada

Una variación importante de este que esquema, es el uso de **file-allocation-table (FAT)**. En esta variación, se reserva una sección del disco para contener una tabla que indica, para cada bloque, cual es su siguiente y los bloques no usados son marcados con un símbolo especial.

Entonces el problema de encontrar un bloque vacío se reduce a buscar en esta tabla el valor especial, sin embargo no es un esquema escalable ya que la misma puede llegar a ocupar mucho espacios y requerir varios movimientos de la cabeza del disco para leer un solo archivo (en el peor de los casos, un movimiento para leer la FAT y otro para leer el bloque por cada bloque del archivo).

11.4.3. Indexación (inodos)

Cada archivo tiene su propio bloque de indices. La i-ésima entrada en este bloque apunta al i-ésimo bloque del archivo. Cuando el archivo es creado, todos los punteros del bloque son inicializados en *null* y se van modificando a medida que el archivo va creciendo.

Para permitir archivos que necesiten mas bloques que los que puede apuntar el bloque indice se pueden usar distintos esquemas:

Lista enlazada: Los bloques de una archivo están definidos por una lista enlazada de bloques de índices.

Indice multinivel: Se comienza con un bloque de indices cuyos punteros apuntan a un segundo bloque de índices que apuntan a los bloques del archivo.

Combinación de los ultimos dos: Es lo que hace linux. Los file-control blocks (inodos), contienen los atributos, una lista de punteros que apuntan directamente a bloques del archivo (**direct blocks**)(permiten acceder a archivos pequeños), luego un puntero a un bloque llamado **single indirect block** que apunta a una lista de **direct blocks**, luego otro puntero a un **double indirect block** y por último un puntero a un **triple indirect block**. Cada uno agregando un nivel más de indirección a los bloques del archivo

De esta manera solo es necesario mantener en memoria las tablas de los archivos abiertos (y solo las partes que se van usando de la misma).

En este caso, se reserva un inodo como entrada al directorio raíz y por cada archivo o subdirectorio hay una entrada que apunta a al inodo correspondiente.

11.5. Manejo del espacio libre

Dado que el espacio del disco es limitado y los archivos se van creando, modificando y borrando con el tiempo, el sistema mantiene una **lista de espacio libre** que indica cuales son los bloques libres del disco. De esta forma, para crear un archivo, se debe buscar en esta lista el espacio requerido. Luego, ese espacio se borra de dicha lista.

Hay varias formas de implementar esta lista:

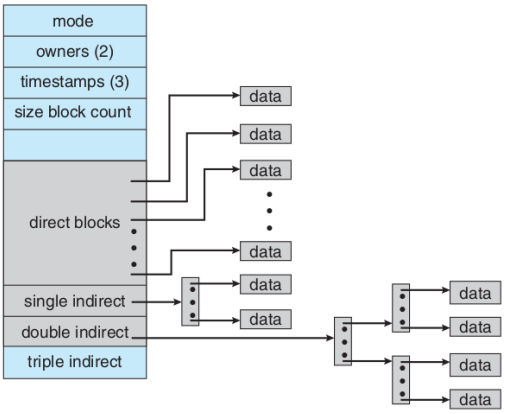

Figure 12.9 The UNIX inode.

Figura 15

11.5.1. Bit Vector

La forma más simple es usar un **bit map** o **vector bit**. Cada bloque esta representado por 1 bit. Si el bloque está libre, el bit es 1, si está siendo usado es un 0.

La principal ventaja de esta implementación es su relativa simplicidad y la eficiencia que tiene para encontrar el primer bloque libre en el disco. Sin embargo, son ineficientes salvo que se mantengantotalmente en memoria, lo cual es un problema si el disco es demasiado grande ya que puede llegar ocupar demasiado espacio.

11.5.2. Listas enlazadas

Otra forma es mantener un puntero al primer bloque libre del disco y cachearlo en memoria. El bloque contiene un puntero al siguiente bloque libre y así. Entonces la lista enlazada está repartida en los bloques libres del disco. Este esquema no eficiente: Para recorrer la lista debemos leer cada bloque lo que requiere un gran tiempo de I/O. Afortunadamente, esto no es algo que se haga usualemente. Por lo general, el sistema operativo, solo necesita un bloque libre para asignar un archivo.

Se puede mejorar este método haciendo que cada bloque tenga n punteros a bloques libres y un último puntero al siguiente bloque de la lista. De esta manera, se acelera el acceso a las direcciones de varios bloques.

Además, también se puede mantener punteros solo al primer bloque de una lista contigua de bloques y la cantidad de bloques contiguos. De esta manera, también nos ahorramos una cantidad gran cantidad de punteros.

11.6. Perfomance

Algunos sistemas separan una parte de la memoria principal para usarla como caché usando el sistema de paginación similar al visto en la sección 7.1. Este tipo de caché trata la información de los archivos como páginas en vez de bloques, esto nos permite unificar los accesos de los procesos con la memoria virtual en vez de utilizar el sitema de archivos: Este esquema se llama **memoria virtual unificada**.

Otro problema que puede afectar la perfomance del sistema es si las operaciones de I/O son síncronas o asíncronas. Las **escrituras síncronas** ocurren en el el orden en el que el subsistema de disco recibe los pedidos y las escrituras no se bufferean. En las **escrituras asíncroncas**, los datos son almacenados en la caché y se devuelve el control al proceso. Luego la caché es bajada a memoria en otro momento. Por lo general, las escrituras se hacen de manera asincroca, sin embargo los sistemas operativos incluyen un flag en la system `open` que permiten a un proceso requerir escrituras sincrónicas.

11.7. Recuperación

Los archivos y directorios deben ser mantenidos tanto en memoria principal como en disco y se debe asegurar que una falla en sistema no resulte en perdida de información o incosistencia en las estructuras de datos del filesystem.

11.7.1. Consistencia

Cualquiera sea la causa de corrupción, un sistema de archivos debe detectar el problema y corregirlo. Para detectarlo, se puede escanear la metadata de cada archivo para confirmar su estado. Sin embargo, esto puede llegar a tomar varias horas si el sistema es muy grande y debería hacerse cada vez que el sistema bootea.

Alternativamente, el sistema de archivos puede almacenar su estado en la meta-data del sistema: Un bit se setea cuando se están produciendo cambios en los datos del sistema. Si todas las actualizaciones son completadas correctamente, entonces se limpia el bit. Si esto no sucede, se debe correr un programa llamado **consistency checker** que realiza el escaneo mencionado y trata de corregir las inconsistencias que encuentre.

11.7.2. Journaling

Con el verificador de incosistencias, permitimos que las estucturas se rompan y las reparamos en la fase de recuperación. Sim embargo, este método tiene varios problemas: Puede haber inconsistencias irreparables, puede requerir intervención humana y tomar mucho tiempo.

La solución a este problema es aplicar técnicas de recuperación basadas en logging: Cada conjunto de operaciones para realizar una tarea específica en el disco es llamada **transacción**. Primero se escriben en un log todos los cambios, se consideran como listos para *commitear* (escribir en disco) y se devuelve el control al proceso. Mientras tanto, las entradas escritas en el log se ejecutan en el sistema de archivos.

A medida que estas acciones se van completando, se actualiza un puntero que indica cual es la próxima acción a realizar. Una vez que se terminó de commitear una transacción, la misma se remueve del log (que suele ser un **buffer circular**).

Si el sistema falla, el log tendrá cero o más transacciones que no fueron completadas, por lo que deben terminarse. El sistema puede retomar la ejecución desde el puntero guardado y se puede recuperar la consistencia del sistema. Si la transacción fue abortada, entonces sus cambios deben revertirse.

Este registro se escrbie en bloques consecutivos, por lo que para modificarlo, se realiza una escritura secuencial y el impacto en performance es bajo.

11.8. Network File System

Con la evolución de la red y la tecnología de archivos, los métodos de compartir archivos han cambiado. Los primeros métodos implementados consistian en transferir manualmente los archivos usando programos como **ftp**. Luego, se comenzaron a usar sistemas de archivos distribuidos (distributed file system - DFS) en la cual los directorios remotos son visibles desde una computadora local. El tercer: **World Wide Web** es, en cierto modo es una reversión al primer método, ya que es necesario un navegador web para poder acceder a los archivos remotos y distintas operaciones son usadas para transferir archivos.

Los sistemas de archivos de red son protocolos que permiten acceder a sistemas de archivos remotos como si fuesen locales usando llamadas a procedimientos remotos. Para esto se define un protocolo de montaje que permite montar el file system remoto en el sistema local como si fuese una parte integral del de sistema de archivos locales. Para poder soportar esto, los SO implementan una capa llamada **Virtual File System** que contiene *vnodes* por cada archivo abierto. Así, los pedidos de E/S que llegan a este sistema son despachados al sistema de archivos locales o al sistema de archivos de red de manera transparente.

Si bien es necesario un módulo de kernel especial del lado del cliente, del lado del server alcanza con un programa común y corriente.

Parte IV
Protección y seguridad

Decimos que un sistema es seguro si sus recursos son usados y accedidos como es debido bajo toda circunstancia. A pesar de que es imposible implementar un sistema totalmente seguro, existen mecanismos para hacer que las infracciones de seguridad sean poco probables. Por lo general, se descompone la seguridad de un sistema en tres partes:

- **Confidencialidad**: Se debe garantizar que solo los usuarios autorizados a ver cierta información pueden hacerlo.
- **Integridad**: Ningún usuario debería poder modificar los datos del sistema salvo que tenga expresa autorización de su dueño.
- **Disponibilidad**: Nadie puede alterar el sistema y hacer que sea inutilizable.

Para esto, debemos tener formas de cerciorarnos que un usuario es quien dice ser (**autenticar**), de controlar a que archivos/procesos puede acceder, usar y modifcar (**autorizar**) y, por último, mantener un registro de cuales son su acciones (**auditar**) para poder identificar quien son los responsables de las modificaciones al sistema.

12. Autenticación: Encriptación de valores/mensajes

La autenticación de un usuario es uno de los mayores problemas de seguridad en un sistema operativo. Por lo general, los usuarios se identifican a si mismos sin embargo debemos tener una forma de verificar la autenticidad de esa identificación.

Criptografía: Rama de las matemáticas y de la informática que se ocupa de cifrar/descifrar información utilizando métodos y técnicas que permitan el intercambio de mensajes de manera que sólo puedan ser leídas por las personas a quienes van dirigidos.

Criptoanálisis: Estudio de los métodos que se utilizan para quebrar textos cifrados con objeto de recuperar la información original en ausencia de la clave.

12.1. Funciones de hash de una vía

La forma más común de autenticación es el uso de contraseñas: Cuando el usuario se identifica con su n nombre de cuenta y una contraseña. Si la contraseña que provee mateha con la almacenada en el sistema, entonces asume que el usuario es quien dice ser.

Desafortunadamente, las contraseñas pueden ser adivinadas, expuestas accidentalmente o robadas de manera muy fácil. Por esta razón, los sistemas operativos las guardan hasheadas en

un archivo del sistema: Dada una clave x, guardan el valor $f(x)$ donde f es una función no inversible (dado $f(x)$ es imposible o es muy difícil saber quien es x).

Si bien, no es posible revertir un hash a su valor original, este método puede ser vulnerado si se toma un diccionario de contraseñas y se hashea cada uno de sus valores. Si el usuario había elegido una palabra del diccionario como contraseña, entonces es descubierta. Por esta razón, el sistema le agrega "sal.ª la misma: Es un valor random que se agrega al final de la contraseña antes de hashearla.

En el sistema se guarda tanto el hash generado por esta concatenación como el valor usado para salar la contraseña original. De esta manera, se complica al atacante descubrir una contraseña porque debe probar cada palabra de su diccionario con cada posible sal existente.

Además, si dos usuarios habían elegido la misma contraseña como se sala con distintos valores, el valor guardado en el sistema es distinto y si la palabra es vulnerada para uno de esos usuarios, el otro no es afectado.

12.2. Encriptación de mensajes

12.2.1. Algoritmos de encriptación simétricos

El remitente codifica su mensaje M usando una clave secreta K que debe ser conocida por el destinatario. Cuando el mensaje llega a destino, el de debe usar la misma clave para decodificarlo.

El problema con este tipo de algoritmo, es que se debe mantener secreta la clave K por lo que se deben buscar medios alternativos para comunicarla (no puede ser a través de una red).

12.2.2. Algoritmos de encriptación asimetricos

En este tipo de algoritmos, el remitente y el destinatario usan claves distintas para codificar y decoficar el mismo mensaje, respectivamente.

Un entidad que desee recibir una comunicación encriptada crea dos claves: Una clave pública P y una clave privada K tales que cualquier mensaje encriptado con P solo puede ser desencriptado con K.

De esta manera, cualquier remitente que desee entablar una comunicación segura con esta entidad podrá encriptar su mensaje con la clave P y enviarla sin preocuparse por que el mensaje sea iterceptado pues solo la entidad correspondiente podrá descifrarlo.

Uno de los algoritmos más famosos de este tipo es el algoritmo RSA.

Otra forma de usar este tipo de algoritmos, es para firmar documentos de manera digital. En estos casos, la entidad que debe firmarlo computa un hash del documento usando un hash de ua via y lo "encriptaçon su clave privada obteniendo lo que se conoce como **bloque de firma (signature block)**.

Por su parte el destinatario computa el hash del documento usando la misma función de hash y cuando recibe el signature block, lo "desencriptaüsando la clave pública del usuario que lo firmó. Si el hash que obtuvo al desencriptar es distinto al que calculó él, entonces el documento o la firma (o ambos) fueron manipulados.

13. Autorización: Privilegio de procesos/usuarios

Se trata de los mecanismos que implementa el sistema para prevenir que los usuarios modifiquen o vean información/archivos a los que no deberían tener acceso.

Uno de los principios más usados durante el desarrollo de un sistema operativo es el **principio del mínimo privilegio**: Se le da a los programas, usuarios e incluso el sistema los privilegios mínimos y necesarios como para que puedan cumplir con su trabajo.

Un sistema de computadora es una colección de **sujetos** (usuarios, procesos e incluso el mismo sistema), **objetos** y recursos (procesos, semáforos, bloques memoria, impresoras, discos). Cada objeto tiene un nombre único que lo diferencia del resto de los objetos del sistema y tiene asociadas **acciones** (operaciones) bien definidas.

Un sujeto solo debe poder acceder a los objetos para los cuales tiene autorización. Lo que es más, solo debe poder acceder a los objetos en el momento en el que los necesita. Este segundo requerimiento es conocido como **need-to-know principle** (principio de necesidad de conocer?[1]) y útil para limitar la cantidad de daño que puede provocar un proceso defectuoso.

Por lo general, como los sujetos en un sistema pueden ser muchos, se los agrupa en roles o dominios según ciertas características comunes: Por ejemplo, puede haber usuarios administradora que pueden hacer cualquier operación, usuarios *normales* que pueden abrir, leer y modificar ciertos archivos y usuarios *visitantes* que solo pueden leer un subconjunto mínimo de archivos públicos.

13.1. Matrices de permisos

Una de las formás más fáciles de implementar la autorizaciones es como una matriz de control de accesos o permisos. Cada columna de la matriz representa un objeto y cada fila un sujeto. En cada celda (i,j) se almacena las operaciones que tiene permitido realizar el sujeto i sobre el objeto j. Algunos ejemplos de estas operaciones son: Leer, escribir, copiar, abrir, borrar, imprimir, ejecutar, matar (a un proceso), etc.

Por lo general, el administrador del sistema define cuales son las operaciones por defecto que se les permite a hacer a cada usuario cuando se crea un objeto. Pero además, cada usuario puede decidir agregar o quitar permisos en cada celda a medida que vaya siendo necesario.

La matriz de acceso nos provee de un mecanismo adecuado para definir e implementar los controles estrictos que siguen los principios mencionados. Dado que cada entrada en la matriz de acceso se modifica individualmente, cada una de ellas debe ser considerada como un objeto a ser protegido.

Cuando un proceso es creado, por lo general, hereda los permisos del usuario que los crea. Algunos sistemas operativos, ofrecen formas de crear procesos con permisos de administrador en casos especiales (por ejemlo, el comando `sudo`, en linux).

[1] No se como traducir esto, si se les ocuree alguna traduccion mejor, propongan

13.1.1. Implementación

Dado que la mayoría de los dominios no van a necesitar acceso a la mayoría de los objetos, la matriz de permisos termina siendo sparsa (muy grande y con la mayoría de sus celdas vacías). Por esta razón, se suele almacenar solo las celdas no vacías.

Hay dos técnicas para lograr esto:

- **Access Control List:** El sistema asocia a cada objeto una lista enlazada con los dominios que pueden tener acceso a dicho objeto.
- **Capabilities List:** Se asocia a cada sujeto/dominio una lista enlazada de los objetos a los que puede acceder.

13.1.2. MAC vs. DAC

En la mayoría de los sistemas operativos, se permite a cada usuario determina quien puede leer y escribir cada uno de sus archivos y objetos, este esquema se denomina **Discretionary Access Control (DAC)**. Sin embargo, hay sistemas que requieren un modelo de protección más estricto: **Mandatory Access Control (MAC)**.

En MAC, se regula el flujo de la información, para asegurar que no se filtre a nadie que no tenga los permisos necesarios. El modelo más conocido es el modelo de **Beill-LaPadula** que tiene las siguientes reglas:

1. **Propiedad de seguridad simple:** Un proceso corriendo a nivel de seguridad k puede leer objetos que estén al mismo nivel o más abajo.
2. **Propiedad *:** Un proceso corriendo a nivel de seguridad k solo puede escribir objetos con el mismo nivel de seguridad o más alto.

El modelo descripto se asegura que toda la información de nivel de seguridad k se mantenga secreta a los usuarios con menor prioridad. Sin embargo, no garantiza la integridad de los datos (alguien de nivel k puede escribir objetos con mayor nivel de seguridad). Por esta razón, tambien existe el modelo **Biba** que es el inverso del anterior: Asegura integridad de la información pero no así mantenerla secreta.

1. **Propiedad de integridad simple:** Un proceso corriendo a nivel de seguridad k solo puede escribir objetos que estén al mismo nivel o más abajo.
2. **Propiedad *:** Un proceso corriendo a nivel de seguridad k solo puede leer objetos con el mismo nivel de seguridad o más alto.

14. Algunos ataques y formas de evitarlos

14.1. Replay Attack

Es una ataque de red en el cual la información de una comunicación es repetida o demorada con fines malicisios. El objetivo del atacante es personificar por algún usuario. Supongamos que hay una comunicación encriptada entre un servidor y un usario, el atacante solo necesita interceptar alguno de los mensajes encriptados y reenviarlo al destinatario original. En este caso, el mensaje será aceptado y el destinatario confundirá al atacante con el remitente original.

Por lo general, para evitar este tipo de ataque, se hace algo parecido al "salt", el servidor envía un token (número random) al cliente por cada mensaje que desea enviar. De esta forma, si el cliente encripta su mensaje con este token de único uso y un atacante lo intercepta, cuando el mismo intente reenviar ese mensaje al servidor, el servidor lo rechazará porque el token ya fue usado.

14.2. Buffer Overflow

Puede suceder en los sistemas escritos con lenguajes que no realizan comprobación de límites de array (por ejemplo C o C++). Esta falta de chequeo, permite que un usuario pueda modificar direcciones de memoria a las que normalmente no debería tener acceso. Veamos un ejemplo:

```
void A(){
  char buffer[128];
  gets(B);
  writeLog(B);
}
```

En este caso, si el programador no implementó por su cuenta que el mensaje ingresado por el usuario tenga, efectivamente, menos de 128 carácteres, se producirá un buffer overflow:

El sistema escribirá en las direcciones consecutivas a la última posición válida del array dentro del stack del proceso. Un usuario malicioso podría usar esta vulnerabilidad y escribir un mensaje B que remplaze todo el stack de la función modificando la dirección de retorno en el proceso. Suponiendo que el atacante, había logrado infiltrar código malicioso dentro de la memoria previamente, podría redigir la dirección de retorno hacia su propio código, logrando tomar el control del sistema.

14.3. Inyección de parámetros

Algunos programas permiten ejecutar comandos a través de sus parámetros sin saberlo. Este tipo dde bug se da cuando un proceso realiza un sysyem call para ejecutar una tarea pero no valida que los parámetros ingresados por el usuario sean correctos. Por ejemplo, supongamos que tenemos un programa que copia un archivo en otro y toma dos parametros: `copy original destino` y que para lograr su objetivo hace una syscall del estilo: `syscall(‘‘cp ’’ + original + ‘‘ ’’ + destino)`.

Si el programado no valida que `original` y `destino` son nombres válidos, entonces un usuario podría llamar al programa con los parametros .ºriginal.txtz "destino.txt; rm -r /". En este caso, la syscall ejecutaria el comando de copiar y el segundo comando que borraría todos los datos del sistema si tiene los permisos necesarios.

14.4. Condiciones de carrera

Existen ataque que aprovechan las race condition de un sistema para vulnerarlo. Por ejemplo, supongamos que un proceso intenta crear un archivo. En este caso, el sistema debe comprobar que el proceso tiene los permisos necesarios para escribir en la locación fijada y luego se le permite escribir.

El problema es que esto ocurre en momentos distintos y en el intervalo entre que uno ocurre y comienza lo segundo, un atacante podría crear un link símbolico que rediriga el path del nuevo archivo a alguna parte del sistema (por ejemplo, el archivo de contraseñas). Entonces, cuando el proceso escriba, lo hará sobreescribiendo el archivo apuntado dejando al sistema operativo incosistente.

14.5. Malware

Se denomina **malware** al software específicamente diseñado para llevar a cabo acciones no deseadas y sin el consentimiento explicito del usuario. En este tipo de software se incluyen los troyanos, worms, bots, adware, keyloggers, dialer, ransomwware, rogueware, etc.

Por lo general, estos programas infectan el sistema del usuario cuando el mismo realiza descargas pocos confiables, conecta dispositivos que no conoece, abre mails de remitentes que podrían contener codigo ejecutable.

Para prevenir este tipo de infecciones, se usa una gran variedad de mecanismos:

- **Firewall:** Es un sistema que permite, al sistema controlar el flujo de datos desde y hacia la red. Gracias a este, puede decidir rechazar información que provienen de direcciones ip desconocidas y evitar que se envie información a las mismas.
- **Antivirus:** Son programas que analizan el código de los procesos a ejecutar en busca de código malicioso.
- **Actualización de software:** Estas actualizaciones tinen como objetivo resolver problemas de seguridad que se van descubriendo con el correr del tiempo.
- **Protecciones y permisos acotados:** La implementación de alguno de los esquemas descriptos en la sección 13 para acotar los permisos de los proceso y así evitar modificaciones no deseadas.
- **Sandboxes:** Es una técnica en el que el çarcelero", un sistema monitorea el comportamiento su prisionero: Cuando el prisionero realiza un system call, el carcelero toma el control del kernel y decide si se debe le debe permitir o no. Si se acepta, el carcelero le devuelve el control. Sino se mata al prisionero.

Parte V
Conceptos avanzados

15. Sistemas distribuidos

Un sistema distribuido es un conjunto de nodos conectados a través de un red de comunicación. Desde el punto de vista específico de cada nodo, el resto de los nodos y sus respectivos recursos son remotos mientras que los suyos son locales.

Hay 4 grandes razones para construir sistemas distribuidos:

- **Compartir recursos:** Si hay varios sitios conectados entre si, entonces un usario puede usar los recursos disponibles en alguno de los nodos de la red. En general, los mecanismos para realizar esto son dispositivos de hardware especializados.
- **Aceleración de procesamiento:** Si hay un proceso que puede ser particionado en varias subprocesos que se pueden correr de manera concurrente, entonces un sistema distribuido puede usar distintos nodos para correrlos simultáneamente para ahorrar tiempo. Adicionalmente, si un nodo está demasiado sobrecargado, el sistema puede repartir el trabajo a otros nodos que no estén en uso (esto se llama **load sharing** o **job migration**).
- **Redundancia:** Los sistemas distribuidos permiten mantener copias redundantes de sus datos en distintos nodos de la red. De esta manera, si un nodo cae, un usuario puede seguir accediendo a la información que necesita sin ningún problema.

Si bien estas son buenas razones para hacer sistemas distribuidos hay que tener en cuenta que su implementación conlleva resolver ciertos problemas como:

- **Sincronización de eventos:** Como ordenarlos eventos cronológicamente. En general, cada nodo de una red tiene su propio clock y es imposible que todos los clock estén sincronizados a la perfección por lo que se deben implementar mecanismos que permitan decidir si un evento en un nodo a es anterior o posterior a un evento en un nodo b.
- **Coherencia de datos:** Se debe asegurar que los datos sean consistentes entre los distintos nodos de la red. Si se realizan ciertas acciones sobre el sistema, se debe asegurar que todos los nodos afectados reciban la información necesaria para actuar acorde a los cambios que se producen. Por ejemplo, si en un nodo se borra alguna entrada de una bases de datos, todos los nodos que estén haciendo uso de esa entrada deben poder darse cuenta que ya no es válida.
- **Información parcial:** Los datos del sistema están repartidos en toda la red: Ningún nodo tiene toda la información, por esta razón debe saber a quien a recurrir para conseguir los datos específicos que necesita en cada momento.´

15.1. Arquitecturas de sistemas distribuidos con memoria compartida

Los sistemas distribuidos pueden tener tres tipos de arquitectura de hardware:

- **Acceso de memoria uniforme (UMA):** Son sistemas en los que se usa un único controlador de memoria. El mismo se encarga de administrar la memoria asiganada a cada proceso/nodo de la red.
- **Acceso de memoria no uniforme (NUMA):** En este sistema se le asigna a cada nodo su propia memoria local para su propio uso. Esto permite que cada nodo acceda de manera más rápida a sus datos y solo se comunica con otros nodos de la red si es necesario.

En cuanto a la administración de memoria del sistema, tenemos distintos tipos de asignaciones:

- **Estructurada:**
  - **Memoria asociativa:** Son memorias que están optimizadas para realizar búsquedas a través de todos los datos (a diferencia de las memorias normales que proveen acceso directo a un dato en base a su dirección).
  - **Arrays distribuidos:** Son arrays cuyos datos se almacenan en distintos nodos de una red. Las operaciones sobre el mismo son enviadas a un nodo *master* que las mapea a operaciones que se distribuyen a los nodos correspondientes.
- **No estructurada:**
  - **Memoria virtual global:** Asigna un namespace a la memoria distribuida en la red. De esta manera, todos los nodos pueden acceder a la misma sin necesitar saber donde se encuentra ubicado el dato que necesita.
  - **Memoria virtual particionada por localidad:** ????

15.2. Clusters

Los clusters son un conjunto de computadoras conectadas por una red de alta velocidad sincronizados para realizar un trabajo en común. Estas computadoras trabajan cooperativamente para proveer servicios de manera cooperativa.

Muchas veces, los sistemas están compuestos por un conjunto de clusters (*grid*). Muchas de estas grillas están conectadas a una red, son conocidas como *clouds* y, por lo general, se alquilan bajo demanda.

Para coordinar la ejecución de un proceso, un scheduler necesita asignar que procesadores de la red lo harán (*asignación estática*) y cada nodo debe esperar que el proceso esté listo y asignarle el tiempo de procesador necesario.

Otra cosa a tener en cuenta, es que puede si un nodo se sobrecarga de trabajo, puede ser necesario redistribuir la carga (*asingación dinámica*) de cada nodo de la red para lograr un

balance de uso equitativo entre los nodos de la red. En este caso, un nodo sobrecargado o un nodo libre, inicia una *migración* que es atendida por el scheduler.

Cuando el scheduler es notificado de que hay que realizar una migración, debe tomar las siguientes decisiones:

- **Transferencia:** Cuándo hay que migrar un proceso.
- **Selección:** Qué proceso hay que migrar.
- **Ubicación:** A donde hay que enviar el proceso.
- **Información:** Cómo difundir el nuevo estado de la red a todos los nodos.

15.3. Acuerdo bizantino

Supongamos que un proceso debe realizar una acción y notificar al resto de la red que dicha acción se realizó exitosamente. En este caso, debemos propagar la información a través de toda la red, sin embargo puede haber nodos de la misma que no funcionen correctamente o intenten sabotear la operación comunicando cosas que no son ciertas. Si es así, el estado de la operación puede parecer exitosa para algunos nodos y fallida para otros.

El acuerdo bizantino es un método usado para encontrar un consenso sobre el estado de la red entre los nodos de la misma. Dada una red de n nodos en la cual a lo sumo f pueden fallar, este consenso se debe lograr entre todos los nodos correctos $(n-f)$ y debe satisfacer tres propiedades:

- **Acuerdo:** Todos los nodos correctos deben decidir el mismo valor (0 si se considera a la operación fallida o 1 si no).
- **Terminación:** Se debe llegar al consenso en una cantidad finita de pasos
- **Validez:** La decisión tomada debe ser un valor que haya sido propuesto por alguno de los nodos.

Teorema: Si el canal por el que se comunican los nodos no estable (hay fallas en la comunicación de los mensajes), no existe algoritmo que nos permita conseguir concenso.

Teorema: Si de los procesos n involucrados en el concenso, llegasen a dejar de funcionar por alguna razon $k < n$, entonces el consenso se puede resolver en $O((k+1) \cdot n^2)$ mensajes.

Teorema: Si los procesos no son confiables (intentan boicotear el sistema), se puede resolver consenso bizantino para n procesos y k fallas si y solo si $n > 3 \cdot k$ y la conectividad es mayor que $2 \cdot k$. Donde "conectividad"‘ es el mínimo número de nodos que tenemos que sacar a la red para que deje de ser conexa.

15.4. Protocolos de comunicación

15.4.1. Comunicación sincrónica

Los protocolos más comunes soportan una arquitectura *cliente/servidor* en la que un nodo (*cliente*) ejecutando un proceso solicita los servicios de otro (*servidor*):

Telnet: Es un protocolo que permite que un usuario se conecte de manera remota a un dispositivo de la red y utilizar sus recursos a través de una terminal de comandos.

RPC: Es un mecanismo que les permite a los programas realizar procedure calls remotas. Involucra una serie de bibliotecas que oculta del programador los detalles de comunicación y le permiten además enviar los datos de un lugar a otro de la red.

15.4.2. Pasaje de mensajes asincrónico

Este mecanismo es uno de lo más generales porque no supone que haya nada compartido, excepto un canal de comunicación a través del cual envían datos. El remitente del mensaje, lo codificado antes de enviarlo por el canal y luego el destinatario lo decodifica. Si la comunicaciones asíncrona, se debe tener en cuenta que el sistema debe atender el traspaso de mensajes, la comunicación es lenta y eventualmente se podrían perder paquetes.

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

15.5. Locks en entornos distribuidos

15.5.1. Lock centralizado

En sistemas distribuidos no tenemos la posibilidad de ejecutar operaciones atómicas por lo que es necesario crear mecanismos que permitan controlar el flujo de los eventosy la modificación de los datos para que no se genere ninguna inconsistencia.

Las opcion más simple es hacer que un único nodo se encarge de coordinar el uso de todos los recursos de la red. En el mismo se ejecutan procesos llamados *proxies* que representan a cada proceso remoto y negocia con el resto de los proxies la asignación de recursos al proceso que representa.

Uno de los principales problemas de esta metodología, es que si el nodo coordinador se desconecta de la red, los recursos quedan inutilizables. Además, dependiendo de la capacidad de la red, se pueden llegar a generar cuellos de botella si hay demasiados procesos haciendo pedidos simultáneamente ya que cada interacción entre un proceso y el coordinador requiere de mensajes que viajen por la red.

Otro problema es la sincronización de los eventos. Cada proceso tiene su propio clock que no está sincronizado con el del resto por lo que el coordinador debe decidir que pasó antes y

qué paso después. Cuando la precisión importa, Lamport, propone definir un *orden parcial no reflexivo* entre los eventos de la siguiente manera:

- Si dentro de un proceso, A sucede antes que B, entonces $A \to B$.
- Si E es el envío de un mensaje y R su recepción, $E \to R$. Aunque E y R sucedan en procesos distintos.
- Si $A \to B$ y $B \to C$, entonces $A \to C$.
- Si no vale ni $A \to B$ y $B \to C$, entonces $A \to C$.
- Si no vale ni $A \to B$, ni $B \to A$, entonces A y B son *concurrentes*.

Entonces, cada proceso usa su reloj para estampar con un valor monótonamente creciente en cada mensaje que envía. Como la recepción del mensaje siempre es posterior al envío, cuando un proceso se recibe un mensaje con una marca de tiempo t que es mayor al valor actual del reloj, actualiza su reloj interno a $t + 1$.

Ahora, supongamos que un proceso recibe dos mensajes de distintos remitentes al mismo tiempo (ambos con el mismo t) y se desea saber cual fue realmente primero. En este caso, deberemos desempatar el t con algún otro valor arbitrario (por ejemplo, usando el pid).

15.5.2. Locks distribuidos

Esta alternativa utiliza el protocolo de mayoría para obtener un lock sobre un recurso copiado en n lugares. Para lograr esto, el proceso que necesita utilizar el recurso envía un pedido al resto de los nodos de la red. Si $n/2 + 1$ nodos le responden que pueden usarlo, entonces consigue el lock.

Cada copia del objeto tiene un número de versión que es actualizado cada vez que el recurso se modifica.

15.5.3. Exclusión mutua

Token passing: Se arma un anillo lógico entre los procesos y se pone a circular un token. Cuando un proceso quiere entrar a su sección crítica debe esperar a que le llegue dicho token.

Si un proceso recibe el token pero no necesita usarlo, lo reenvia al proxima nodo inmediatamente. Si necesita entrar en su sección crítica, lo retiene hasta que termina de ejecutarla.

Si no hay fallas en la red, no hay inanición, sin embargo usar este método implica que haya mensaje circulando aún cuando no son necesarios.

Solicitudes: En este esquema, cuando proceso quiere entrar en su sección crítica debe enviar una solicitud a todos los nodos de la red y esperar la respuesta de cada uno de ellos. Una vez recibidas, el proceso podrá acceder a su sección crítica.

Si hay algún nodo de la red que ya esté ejecutando una sección crítica, entonces ese nodo no devolverá una respuesta al finalizar su ejecución. Además, si hay otro nodo esperando (que haya

pedido para entrar a su sección crítica antes), entonces se le da prioridad a ese nodo (y ese nodo tampoco dará respuesta hasta haber entrado y salido de su sección crítica).

El resto de los nodos (aquellos que no necesitan entrar en su sección crítica o que hayan hecho un pedido después que el proceso actual) deberán responder inmediatamente.

15.5.4. Elección de Lider

Es parecido a token passing. Se organizan los proceso en un anillo y cada uno hace circular su ID. Cuando un proceso recibe un id, lo compara con el suyo y envía al siguiente proceso el mayor de los dos.

El proceso termina cuando a un proceso le llega su propio ID en el mensaje, esto quiere decir que su ID recorrió todo el anillo sin encontrar a alguien mayor por lo que él puede considerarse líder. En este momento, el proceso auto-identificado como líder pone en circulación un mensaje notificando el hecho.

15.6. Instantánea global consistente

Una instantánea global de una red es un estado que consiste en el estado local de cada proceso del sistema junto con los mensajes en tránsito en sus canales de comuncación en determinado momento.

Cuando un proceso pide una instantanea, un proceso envía un mensaje a todos los nodos de la red (incluso a si mismo) llamado *marca*. En el momento que un proceso recibe este mensaje, guarda una copia de su estado y envía otro mensaje de *marca* a todos los procesos.

El proceso que inicio la instantanea, comienza a registrar todos los mensajes que recibe hasta que consigue todos los mensajes de marca del resto de los nodos. En este momento, la instantánea estará completa.

Esta puede ser usada para determinar propiedades estables, determinar si hubo procesos que terminaron, realizad debugging del sistema y detectar deadlocks.

15.7. 2PC

Two Phase Commit (2PC) es un protocolo de transacciones atómica que coordina a los procesos que participan de una transacción distribuida para commitearla o abortarla. Una vez que se comienza a ejecutar una transacción, el protocol consa de dos fases:

1. La fase de request de commit (o de votación) en la que el coordinador de los procesos intenta prepararlos para que tomen todos los pasos necesarios para commitear o abortar la transacción y a votar que se debe hacer.

2. La fase de commit: Basandose en la votación de los procesos participantes, el coordinador decide si debe hacer commit de la transacción o abortarla (si al menos uno decidió hacer lo último entonces todos deben hacerlo). Una vez tomada la decisión, notifica a todos los participantes que deben proceder a realizar las acciones necesarias para cumplir lo notificado.

15.8. File System Distribuidos (DFS)

Un sistema de archivos distribuidos es un sistema que provee servicios de archivos (abrir, leer, escribir, cerrar, etc) a sus clientes. Con la particularidad, de que los archivos están repartidos entre todas las máquinas de una red.

La interfaz del cliente no debería distinguir entre archivos locales y remotos. Es deber del DFS localizar el archivo y gestionar el transporte de datos.

15.8.1. Estructuras de nombrado

- **Transparencia de ubicación:** El nombre de un archivo no da ninguna pista de cual es la ubicación física en la que está almacenado.
- **Independencia de ubicación:** No es necesario cambiar el nombre del archivo cuando este es movido entre distintos dispositivos físicos.

La mayoría de los DFSs proveen mapeo estático, independiente de la ubicación para nombre a nivel de usuario.

Hay tres enfoques básicos para nombrar los archivos en estos sistemas:

1. El más simple, es identificar a los archivos con alguna combinación del nombre de su host y su nombre local, lo que garantiza un nombre único en todo el sistema distribuido. Sin embargo, este esquema no transparente ni independiente de la ubicación física del archivo.

2. La segunda forma consiste en montar los directorios remotos en directorios locales, dando la apariencia de un árbol de directorios coherente. Este método permite la integración transparente de archivos, sin embargo cada máquina debe montar cada directorio remoto a su árbol, lo que puede resultar en diferencias entre los directorios de cada una.

3. La última opción es declarar una estructura única global de nombres que abarque todos los archivos del sistema. Sin embargo, este método es difícil de implementar debido a los archivos con nombres especiales y si algún servidor se cae, todos los archivos con nombres almacenados en ese servidor dejarán de ser accesibles.

15.8.2. Acceso a archivos remotos

Cuando un usuario acceder a un archivo de forma remota, el sistema manda un pedido al servidor, el servidor realiza las accesos necesarios y devuelve los resultados al usuario. Por lo general, esto se realiza mediante algún protocolo de RPC (Remote Procedure Call).

Para asegurar un rendimiento adecuado del mecanismo, reducir operaciones de disco y disminuir el tráfico dentro de la red, se utilizan cachés en las máquinas de usuario que mantienen una copia local de los accesos realizados recientemente.

De esta forma, si la información está cacheada (y es válida), se usa directemente sin hacer ningún pedido al server, si no hace el request correspondiente. Cuando la copia cacheada sufre alguna modificación, ésta se debe ver reflejada en el servidor para mantener consistencia.

Políticas de actualización de Cachés

Una de las políticas de actualización más seguras es la **write-through** que envia las modificaciones al servirdor apenas se realizan. Sin embargo, requiere que cada escritura deba esperar a que la información sea recibida por el server por lo que tiene bajo rendimiento.

Una alternativa es la política**delayed-write** o **write-back**, en la que las modificaciones son escritas en caché y luego son enviadas al server. Esto soluciona el problema de **write-through** y además, las modificaciones sobre-escritas por otros nodos de la red pueden ser obviadas. Sin embargo, es más difícil mantener el estado del archivo consistente.

Las variantes de esta política se diferencian por el momento en que un bloque de datos es enviado al servidor. Esto puede realizarse en el momento en que un bloque va a ser borrado de la caché del host, o a intervalos regulares en los que se envian los bloques modificados. Otras opcion, en sistemas en los que los archivos están abiertos durante un largo período de tiempo y sufren cambios frecuentes es enviar los bloques modificados una vez que el archivo en cuestión es cerrado.

Consistencia de archivos

Al momento de utilizar un archivo, el host debe decidir si la copia de un archivo en su caché es consistente con el archivo guardado en el servidor. Si no lo es, entonces debe actualizar su copia antes de accederlo. Hay dos formas de verificar la validez de la información cacheada:

1. **Validación iniciada por el cliente:** El cliente contacta al servidor, quien compara las dos copias y responde con una respuesta. La frecuencia de la validación puede ser una vez antes de cada acceso o solo una única vez en el primer acceso.

   Hay que tener en cuenta que cada verificación demora el acceso y hacerlos de manera muy frecuente puede conllevar a una sobrecarga de la red.

2. **Validación iniciadad por el servidor:** El servidor mantiene un registro de los archivos cacheados en cada cliente. Cuando detecta que puede llegar a haber una inconsistencia en algún archivo, notifica al cliente afectado y deshabilita el cacheo para ese archivo.

16. Virtualización

16.1. Microkernels

Los microkernels son kernels que solo hacen tareas básicas (manejo de memoria básico, IPC liviano y manejo básico de Entrada/Salida). La idea era minimizar el tamaño del kernel y que las funcionalidades que se le deseen agregar sean provistas por servicios que el usuario debería instalar. Esto permite mayo flexibiidad, extensibilidad y facilidad al momento de actualizar un sistema. Además, una baja de alguno de los servicios no implica una caída total del sistema.

Aunque esta idea, derivó en una comunicación mas rápida entre procesos y módulos de kernel, este enfoque resultó ser mucho más lentos que los kernels monolíticos y nunca se terminaron de instalar en la industria.

16.2. Máquinas Virtuales

La idea de una máquina virtual es abstraer el hardware de una computadora en diferentes ambientes de ejecución, creando la ilusión de que cada ambiente está corriendo en su propia computadora.

La implementación de una máquina virtual involucra varios componentes:

- En la base está el **host**, que es el sistema físico que corre todos las máquinas virtuales.
- Un **administrador de máquinas virtuales** (*virtual machine manager - VMM*) que crea y corre las máquinas virtuales ofreciendoles una interfaz para comunicarse con el hardware.
- Y por último, los procesos *huesped* que reciben una copia virtual del host.

Debido a que el host y cada sistema huésped están casi totalmente aislados entre sí, si un virus afecta alguno de ellos, el mismo no podrá afectar al resto de la máquinas virtuales.

En centros de datos, las máquinas virtuales se usan para **consolidar** sistemas en un una única máquina virtual, por lo que es más fácil optimizar el uso recursos. Además, algunas maquinas virtuales incluyen *migración sin perdida de servicios (live migration)*, lo que permite mover huéspedes en ejecución desde un servidor físico a otro sin interrrumpir sus operaciones.

16.2.1. Implementación

Simulación: Con este método, se crea una variable de estado artificial que representa la sistema huésped que es modificada cada ve que se ejecuta una instrucción. Sin embargo, el mecanismo puede llegar a ser muy lento.

Emulación : Consiste en mantener un modo usuario y un modo kernel virtuales que interactuan directamente con el hardware de la máquina real. Cuando el kernel virtual intenta ejecutar una instrucción privilegiada, se genera una *trap* que redirige la instrucción hacia la máquina virtual y emula la acción pedida. Una vez realizado esto, el sistema real vuelve a tomar control de los procesos.

Este método soluciona, tiempos de ejecución de las instrucciones ejecutadas en modo usuario ya que se corren de manera nativa en el hardware. Sin embargo, ejecutar instrucciones privilegiadas puede afectar el rendimiento del sistema.

Translación binaria: En los microprocesadores en los que no hay una separación clara entre instrucciones privilegiadas y no privilegiadas (por ej, en el x86 hay instrucciones que en modo usuario cambian algunos flags y en modo kernel cambian otros), se usa este método. La idea es que si la máquina virtual quiere ejecutar una de estas instrucciones especiales, primero las traduce a un conjunto de instrucciones equivalente que luego son ejecutadas por el kernel o emuladas dependiendo el efecto que tengan.

Virtualización asistida por Hardware: Para facilitar la imlementación de los sistemas de emulación (y resolver varios problemas), las fabricantes agregaron soporte para virtualización a los microporocesadores. En el caso de intel, la extensión se llama *VT-x* y provee dos modos: **VMX root** y **VMX non-root** que permiten ejecutar las instrucciones del procesador de manera segura.

Además, se agrega la **Virtual Machine Control Structure** que contiene el estado del huésped, el del anfitrión y distintos campos de control (interrupciones que recibio el huésped, puertos de E/S, etc). De esta manera, cuando se ejecutan una instrucción privilegiada, el hardware pasa automaticamente a modo **VMX root** y deja que el controlador de la máquina virtual la emule, ignore o termina la acción pedida.

16.2.2. Contenedores

Durante el flujo de trabajo tradicional, el código de un programa se instala y prueban en un sistema particular. Cuando es transferido a un nueva máquina, las diferencias con el sistema operativo sobre la cual se corre la nueva copia puede resultar en errores.

Los contenedores eliminan este problema al encapsular un único paquete ejecutables con el el código de la aplicación, los archivos de configuración, librerías y dependencias necesarias para que pueda correr. Las aplicaciones dentro de los contenedores están aisladas ya que son ejecutadas a través de un motor de ejecución instalado en el sistema operativo del *host*, mediante el cual varios contenedores pueden compartir los recursos de la máquina. Además, gracias a esto, ocupan menos espacio y usan menos recursos que las máquinas virtuales.

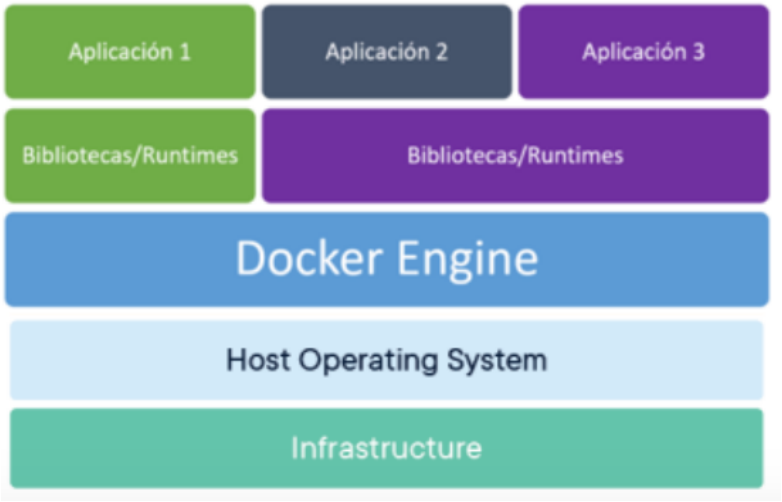

Figura 16: Estructura de un sistema con contenedores