Sistemas Operativos (1)

Intro

¿Qué es un SO?

Un Sistema Operativo es una pieza de software que hace de intermediario entre el hardware y los programas de usuario.
Tiene que manejar la
contención y la **concurrencia** de manera tal de lograr:

- Hacerlo con buen rendimiento.
- Hacerlo correctamente.

La **contención** consiste en permitir que varios programas puedan acceder a un mismo recurso del SO a al vez, mientras que la **concurrencia** refiere al concepto de multiprogramación, varios trabajos realizándose a la vez.

Para lograr todo esto, corre en nivel de
privilegio 0, es decir, máximo privilegio

Elementos básicos de un SO

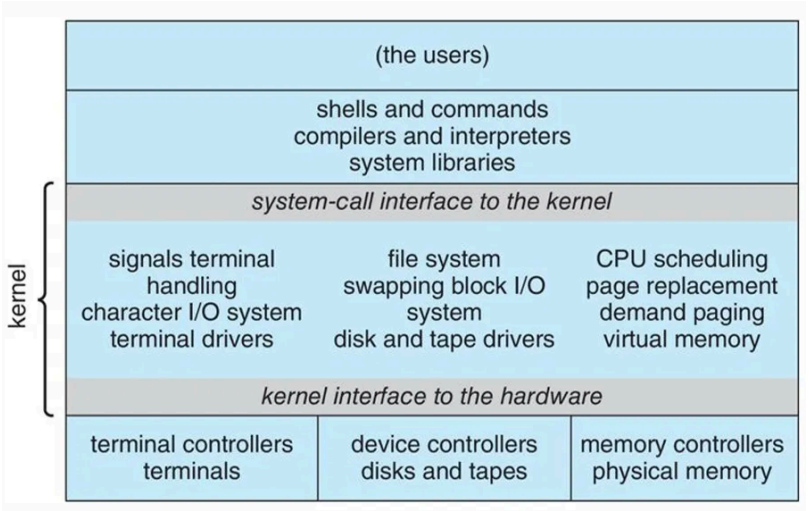
Primero y principal:

- **Drivers**: programas que manejan los detalles de bajo nivel relacionados con la operación de los distintos dispositivos.
- **Kernel**: es el SO propiamente dicho, su parte central. Se encarga de las tareas fundamentales y contiene los diversos subsistemas que iremos viendo en la materia.

Le siguen:

- Intérprete de comandos o **Shell**: un programa más, que muchas veces es ejecutado automáticamente al inicial la computadora, y le permite al usuario interactuar con el SO.
- Proceso: un programa en ejecución junto con su espacio de memoria asociado y otros atributos.
- Archivos
- Directorios
- File System
- Dispositivo virtual
- Directorios y binarios del sistema
- Archivos de configuración
- Usuario
- Grupos de usuarios

Estructura de un SO Unix tradicional



Procesos

Qué es un proceso?

Un programa es una secuencia de pasos escrita en algún lenguaje. Este programa eventualmente se compila en código objeto, que también es un programa escrito en lenguaje de máquina.

Cuando ese programa se ejecuta, se convierte en un **proceso**, el cual tendrá asignado un identificador numérico único, el PID (Process ID).

Visto desde la memoria, un proceso está compuesto por:

- Área de texto: donde se aloja el código de máquina del programa
- Área de datos: donde se aloja el heap
- Stack del proceso

Actividades de un proceso

Terminación

El proceso indica al SO que ya puede liberar todos sus recursos mediante `exit()`. Además, indica su estado de terminación. Este código de estado es reportado al proceso padre.

Lanzar un proceso hijo

El SO cuenta con una estructura jerárquica para la organización de procesos.

Existe un árbol de procesos en el que cada uno tiene un padre, y puede tener o no hijos. Se inicia con un proceso general llamado ***init*** o ***systemd***.

Un proceso puede entonces ejecutar las siguientes *syscalls* (entre otras) que pertenecen a la API de un Unix-like, POSIX:

- `fork()`: Es una llamada al sistema que crea un proceso exactamente igual al actual.
  - `pid_t fork(void);`
- `getpid()`: Retorna el PID del proceso llamante, que es una copia exacta del padre.
  - `pid_t getpid(void);`
- `wait()`: Permite al padre suspenderse hasta que termine el hijo.
  - `pid_t wait(int *_Nullable wstatus);`
- `exit()`: Cuando el hijo termina, el padre obtiene el código de status del hijo.
  - `void exit(int status);`
- `exec()`: Permite al proceso hijo reemplazar su código binario por otro, pudiendo así ejecutar algo distinto al padre.
  - `int execv(const char *pathname, char *const argv[]);`

Ejecutar en CPU

Una vez que el proceso está ejecutándose, se dedica a hacer operaciones entre registros y direcciones de memoria, E/S, llamadas al sistema (*Syscalls*)

Imaginemos el programa más elemental, que sólo hace operaciones entre registros.

Para cambiar el programa que se ejecuta en la CPU, debemos, guardar los registros, guardar el EIP, cargar nuevos regitros y programa, etc. Esto es el **context switch**.

El IP y demás registros se guardan en una estructura de datos llamada PCB (Process Control Block).

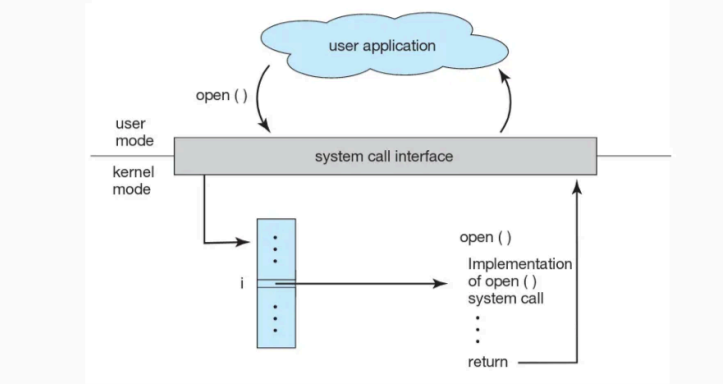
Es importante notar que el tiempo utilizado en cambios de contexto es tiempo muerto, donde no se realiza trabajo productivo.

Esto implica la necesidad de tener un ***Scheduler***, un software fundamental del sistema operativo. El *scheduling* supone un gran impacto en el rendimiento de un SO.

Syscalls

Un proceso puede hacer llamadas al sistema (*syscalls*) como `fork(), exec(), write()`. Estas llamadas requieren **cambiar el nivel de privilegio** y realizar un **cambio de contexto**, lo que consume tiempo.

Mediante *syscalls* se nos brinda la API del sistema operativo y **se implementan mediante interrupciones** (0x80 en Linux 32-bits). Los parámetros se pasan por registros o memoria, y normalmente se acceden mediante *wrapper functions* en C, que permiten interactuar con el sistema con mayor portabilidad y sencillez. Por ejemplo, `printf()` de la biblioteca estándar de C, que utiliza la syscall `write()` .



Invocación de la *syscall* `open()` desde una aplicación de usuario.
Imagen extraída de *Operating System Concepts* (Abraham Silberschatz et al.)

Context switch de procesos y syscalls

El **context switch** entre procesos es pesado porque implica el manejo completo de estados, cambios en la tabla de páginas, gestión de recursos y separación de memoria. En cambio, el context switch entre un proceso y una syscall es más ligero, ya que solo se cambia el estado del proceso y se opera en el mismo espacio de direcciones, lo que reduce significativamente la carga del sistema operativo.

Entrada/Salida

Trabajar con E/S es muy lento. Existen varios métodos

- Busy waiting: el proceso no libera la CPU. Solo puede ejecutarse un proceso a la vez.
- Polling: el proceso libera la CPU, pero aún recibe un quantum que desperdicia hasta que la E/S esté terminada.
- Interrupciones: este método permite la multiprogramación. El SO no otorga más quanta al proceso hasta que su E/S esté lista. El hardware comunica que la E/S terminó mediante una interrupción. Esta interrupción es atendida por el SO, que en ese momento "despierta" al proceso.

Multiprogramación

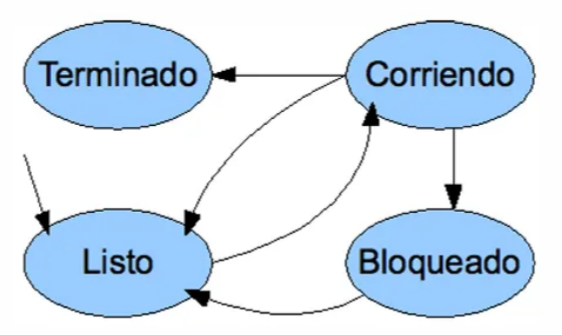
Hay dos formas de hacer esto desde el código:

- Bloqueante: hago la *syscall* y cuando recibo el control, la E/S ya terminó. Mientras tanto, me bloqueo.
- No bloqueante: hago la *syscall*, que retorna enseguida. Puedo seguir haciendo otras cosas, pero debo enterarme de alguna manera si mi E/S terminó.

Estados de un proceso

- **Corriendo**: está usando la CPU.
- **Bloqueado**: no puede correr hasta que algo externo suceda (típicamente E/S lista).

- **Listo**: el proceso no está bloqueado ni corriendo, es decir, no tiene CPU disponible como para correr.



Es responsabilidad del *scheduler* elegir entre los procesos listos cuál es el próximo a correr, que está determinado por su política de *scheduling*.

Sin embargo, queda claro que se necesita tener una lista de procesos. En realidad, es una **lista de PCBs**, llamada **tabla de procesos**.

En cada PCB se guarda la prioridad del proceso, su estado y aquellos recursos por los que está esperando. Los PCBs suelen también formar una lista enlazada que comienza en cada recurso por el que están esperando.

Señales

Las señales son un mecanismo que incorporan los sistemas operativos POSIX para notificar a un proceso la **ocurrencia de un evento**.

Cuando un proceso recibe una señal, su **ejecución se interrumpe y se ejecuta un *handler***.

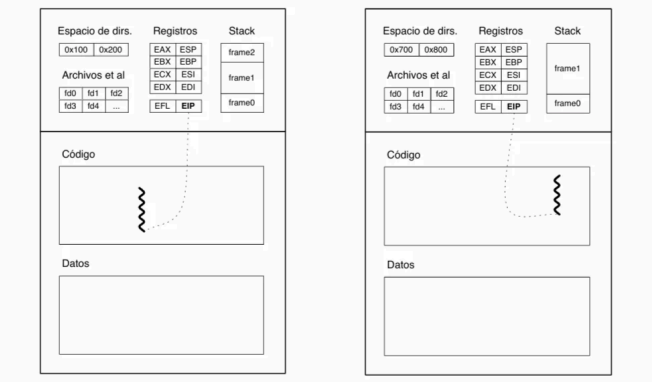
Cada tipo de señal tiene asociado un *handler* por defecto, que puede ser modificado mediante la *syscall* `signal()`.

Toda señal tiene asociado un número que identifica su tipo. Por ejemplo: SIGINT, SIGKILL, SIGSEGV.

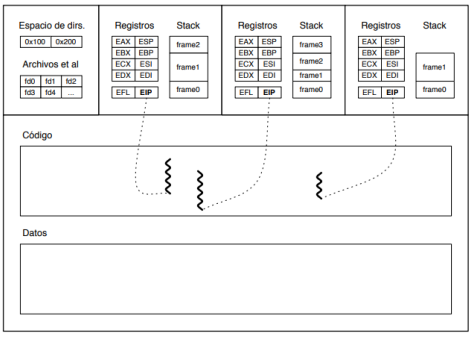
Las señales SIGKILL y SIGSTOP no pueden ser bloqueadas, ni se pueden reemplazar sus *handlers*.

Las señales pueden enviarse desde la terminal con `kill` o entre procesos usando la syscall `kill()`.

Procesos vs Hilos



Concurrencia de procesos:



Concurrencia de hilos

Diferencias entre el context switch de hilos y procesos

El cambio de contexto entre procesos es más costoso que entre threads por las siguientes razones:

1. **Estructuras del kernel**: Se necesita acceder al PCB (Process Control Block) y cambiar la tabla de páginas en la MMU para cada proceso, además de crear nuevas entradas en la tabla de procesos.
2. **Memoria separada**: Cada proceso tiene su propio segmento de datos, heap y pila. Aunque los threads también requieren una nueva pila, comparten el mismo espacio de datos y heap del proceso.
3. **Gestión de recursos**: Los procesos tienen su propia tabla de descriptores para archivos abiertos y manejan señales de forma independiente. Esto puede requerir flushing de cachés durante el cambio de contexto.
4. **Diferencias en el context switch**: El cambio entre procesos implica más operaciones en la MMU y gestión de memoria, mientras que el cambio entre hilos solo requiere actualizar el estado del hilo, sin afectar el espacio de memoria compartido.

En conclusión, el cambio de contexto entre procesos implica más sobrecarga debido a la gestión de memoria y estructuras del kernel, mientras que el de threads es más ligero porque comparten recursos dentro del mismo proceso.

---

1. Cambio de contexto entre procesos (proceso-proceso)

Cuando el sistema operativo cambia la ejecución de un proceso a otro. Implica:

- Guardar el estado del proceso saliente (valores de registros, stack pointer, program counter, puntero a File Descriptor Table, etc.) en la PCB
- Restaurar el estado del proceso entrante (PCB respectiva nuevamente). Todo esto es costoso debido a restauración de registros y memoria.
- Cambio de espacio de direcciones de memoria → gestión de MMU y TLB flush en muchos casos; espacios para datos, heap y stack.

> 💡 El cambio de espacio de direcciones al restaurar un proceso implica sólo actualizar el CR3 (hecho al restaurar registros con PCB), pero gestionar un nuevo espacio de memoria (lo cual, por lo dicho, implica trabajar con MMU y cache) es necesario en ciertas ocasiones: por ejemplo, al crear un proceso con `fork()`, donde, a largo plazo, requerirá un nuevo mapeo de memoria (al inicio puede compartirlo siguiendo la estrategia Copy on Write)

🔴 **Costo**: **ALTO**

Porque implica interacciones con la memoria y cambios en el TLB (lo que puede causar fallos de caché) hecho por el kernel.

2. Cambio de contexto por una llamada al sistema (proceso-syscall)

Cuando un proceso en modo usuario hace una llamada al sistema, (`read()`, `write()`, `open()`, etc). Esto requiere:

- Cambio de modo de usuario a modo kernel → cambio de contexto en términos de registros y punteros del proceso, pero sin cambiar el espacio de memoria (**el mismo proceso ejecuta la syscall** pero con otro nivel de privilegio →comparte direcciones de memoria)
- Ejecutar la llamada en el kernel → overhead del sistema (operaciones de SO como trabajar con dispositivos son costosas)
- Restaurar el contexto y regresar a modo usuario.

🟠 **Costo**: **MODERADO**

Menor que el cambio entre procesos porque **no cambia el espacio de direcciones**, pero sigue habiendo costo por el cambio de privilegios y la ejecución del código en el kernel.

3. Cambio de contexto entre hilos (thread-thread)

Si los hilos son del **mismo proceso**, comparten el espacio de direcciones, por lo que un cambio de contexto entre ellos implica:

- Guardar y restaurar registros de CPU.
- Cambio de stack → el resto de segmentos (como el de datos, donde se almacenan variables globales, el de código y el de heap) son compartidos en el proceso.

🟢 **Costo**: **BAJO**

Porque no hay cambio de espacio de direcciones (se evita gestión de memoria con MMU e invalidación de TLB), como sí entre procesos.

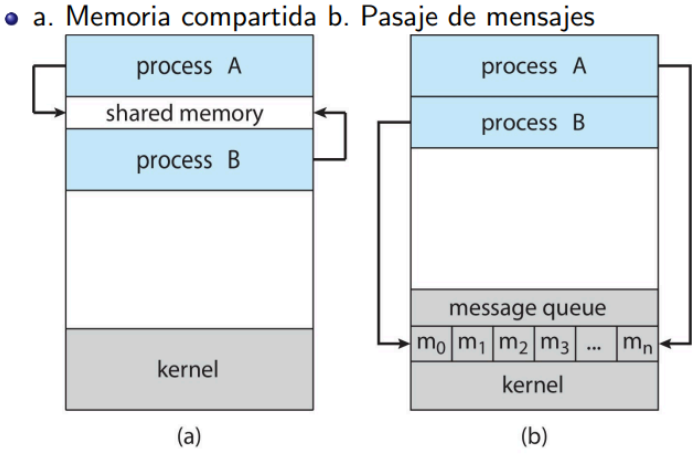
IPC

La comunicación entre procesos, ya sea entre procesos en un mismo equipo o entre procesos remotos, sirve para:

- Compartir información
- Mejorar la velocidad de procesamiento
- Modularizar

Hay varias formas de IPC:

- **Memoria o recursos compartidos**
- **Pasaje de mensajes**

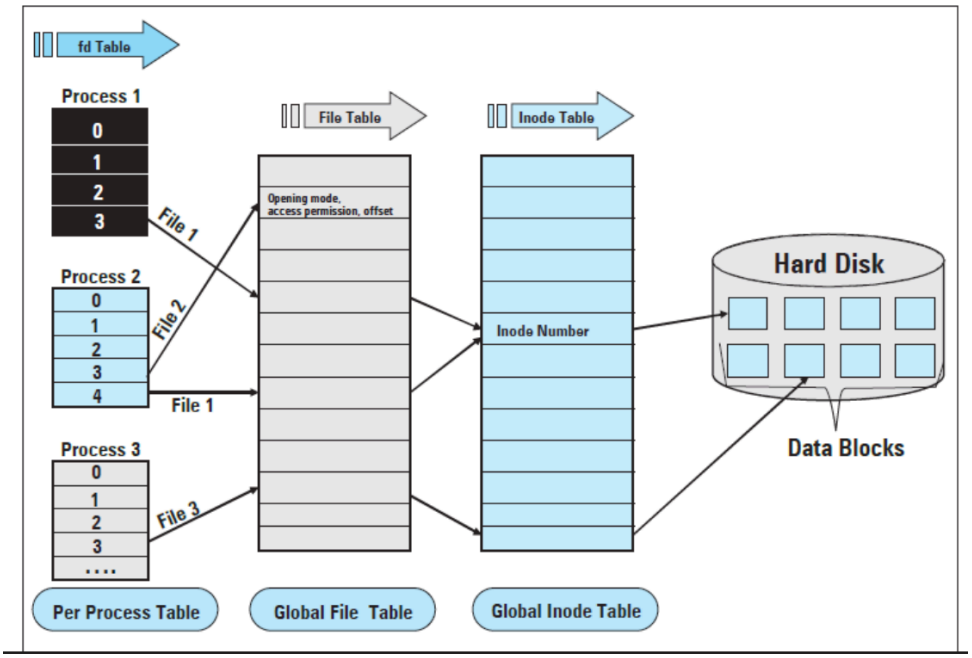


File Descriptors

Intuitivamente, representan **instancias de archivos abiertos**. En términos concretos, son **índices de una tabla** que **almacena los archivos abiertos** por el **proceso**.

Cada proceso en UNIX posee su propia tabla (en su PCB) desde el momento de su creación. El Kernel utiliza esta tabla para referenciar los archivos abiertos del proceso, donde cada entrada apunta a un archivo específico.

La mayoría de los procesos esperan tener abiertos 3 file descriptors (las entradas 0, 1 y 2 de la tabla) correspondientes a: 0 = standard input, 1 = standard output y 2 = standard error.



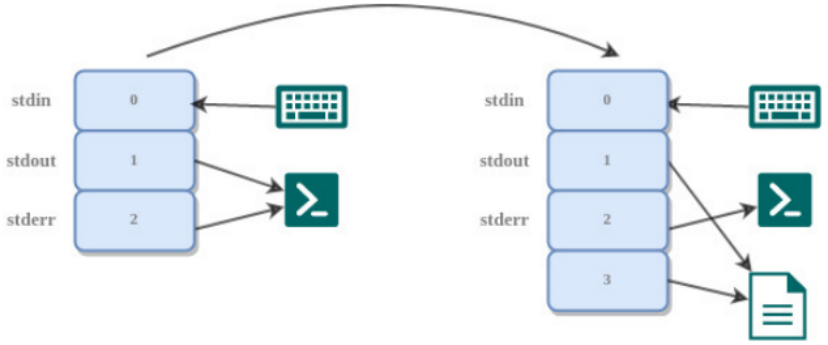
Estos file descriptors se heredan de un proceso padre a un proceso hijo al usar la llamada a `fork()`, y se mantienen en la llamada a `execve`.

Se trabaja con ellos mediante:

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

Función dup2: pisa en el file descriptor `newfd` el contenido que está en `oldfd`:

```
int dup2(int oldfd, int newfd);
```



Ejemplo: redireccionando stdout a un archivo nuestro

Pipes

Un pseudo-archivo que encapsula una forma de IPC. Se los llama en consola con el caracter "|".

- **Ordinary pipes**: ls -l | grep so
- **Named pipes**: mkfifo -m 0640 /tmp/mituberia

Ejemplo:

`echo "sistemas" | wc -c`
Se llama a
`echo`, que escribe por `stdout`.
Se llama a
`wc -c`, que cuenta cuántos caracteres entran por `stdin`.
Se conecta el
`stdout` de `echo` con el `stdin` de `wc -c`.

Cómo funcionan?

Un pipe se representa como un **archivo temporal** y **anónimo** que se aloja en memoria y actúa como un ***buffer* para leer y escribir de manera secuencial**.

> Nota: Los pipes son un canal que debe ser interpretado como un ***byte stream***. No hay noción de separación por mensajes.

Uso de Pipes en C

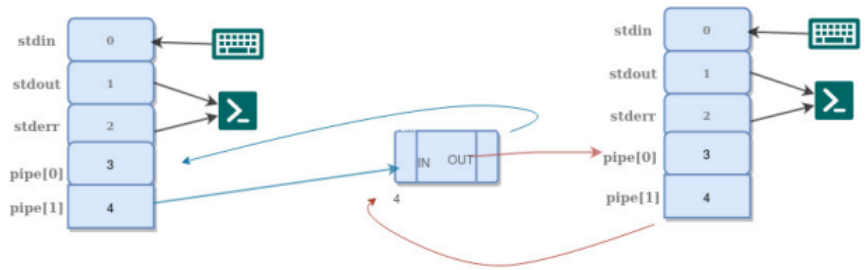
Se crea mediante la syscall:

`int pipe(int pipefd[2]);`

Luego de ejecutar pipe, obtenemos:

- En `pipefd[0]`: un file descriptor que apunta al extremo de lectura del pipe
- En `pipefd[1]`: un file descriptor que apunta al extremo de escritura del pipe

Recordemos que los *file descriptors* del padre se copian al hijo, y siguen apuntando a los mismo extremos del pipe.



Comunicación vía señales

En **C**, la comunicación entre señales se maneja con la biblioteca `<signal.h>`, permitiendo que un proceso capture y responda a señales del sistema, como `SIGINT` (Ctrl+C) o `SIGTERM`.

Uso de señales

- Definir handlers y asociar señal con ellos → `signal(signum, handler)`
- Enviar señales con `kill` desde otro proceso o desde el mismo programa → `kill(pid, signum)`.
- Esperar y manejar la señal en el proceso receptor → `pause()` o `sleep()`.

Ejemplo:

```
void handler(int sig){
	printf("Recibida señal %d\n", sig);
}

int main() {
    signal(SIGUSR1, handler);  // Configurar el manejador para señal SIGUSR1
    printf("PID: %d\n", getpid());

    while (1) pause();  // Esperar señales
}
```

```
int main() {
    pid_t pid = 12345;  // PID del receptor
    kill(pid, SIGUSR1); // Enviar señal SIGUSR1
}
```

Scheduling

La política de scheduling es una de las principales características de un sistema operativo.

Su importancia es tal que algunos sistemas operativos ofrecen más de una opción. Gran parte del esfuerzo para optimizar el rendimiento de un sistema operativo se centra en mejorar su política de scheduling.

¿Qué se debe optimizar?

- **Ecuanimidad (*fairness*):** Garantizar que cada proceso reciba una cantidad "justa" de tiempo de CPU (según alguna definición de justicia xD).
- **Eficiencia:** Mantener la CPU ocupada en todo momento.
- **Carga del sistema:** Minimizar la cantidad de procesos listos que están en espera de la CPU.
- **Tiempo de respuesta:** Reducir el tiempo de respuesta percibido por los usuarios interactivos.
- **Latencia:** Disminuir el tiempo que tarda un proceso en comenzar a generar resultados.
- **Tiempo de ejecución:** Minimizar el tiempo total que un proceso necesita para completarse.

- **Rendimiento (throughput):** Maximizar la cantidad de procesos finalizados por unidad de tiempo.
- **Liberación de recursos:** Asegurar que los procesos que tienen reservados más recursos finalicen lo antes posible.
- **Turn-around**: Maximizar el tiempo que le toma a un proceso desde que inicia hasta que finaliza toda su ejecución.

Tipos

El scheduling puede ser **cooperativo** o **con desalojo**.

1. Scheduling con desalojo (preemptive)

Permite al scheduler utilizar interrupciones del **clock** para decidir si el proceso en ejecución debe continuar o ceder el control a otro.

Si bien el scheduling con desalojo es generalmente deseable, presenta algunas consideraciones:

- **Requiere un clock con interrupciones**, el cual podría no estar disponible en procesadores embebidos.
- **No garantiza continuidad a los procesos**, lo que puede ser un problema en sistemas operativos de tiempo real.

2. Scheduling cooperativo (non-preemptive)

En este enfoque, el scheduler toma decisiones únicamente cuando el **kernel** obtiene el control, como en las llamadas al sistema (**syscalls**), especialmente en operaciones de **entrada/salida (E/S)**.

Algunos sistemas proporcionan llamadas explícitas para que un proceso pueda ceder voluntariamente el control y permitir la ejecución de otros.

> 💡 En la práctica, **los schedulers con desalojo combinan ambos enfoques**, ya que también consideran eventos como las llamadas al sistema para optimizar la planificación de procesos.

Políticas de Scheduling

1. FIFO (First-Come, First-Served - FCFS)

Los procesos se ejecutan en el orden en que llegan.

✅ **Pros**:

- Simple de implementar.
- **Justo** en términos de orden de llegada (**fairness**).

⚠️ **Contras**:

- Puede causar el **efecto convoy**: un proceso largo al principio puede hacer que todos los procesos siguientes se retrasen significativamente.
- No tiene en cuenta la duración de los procesos, lo que puede llevar a una **latencia alta** (alto tiempo de respuesta) para procesos cortos.

---

2. Round Robin (RR)

Cada proceso recibe un ***quantum*** de tiempo para ejecutarse. Después de ese tiempo, el scheduler pasa al siguiente proceso.

✅ **Pros**:

- Adecuado para sistemas **interactivos**.
- Asegura que todos los procesos reciban tiempo de CPU de manera justa.

⚠️ **Contras**:

- El **quantum** debe ser elegido con cuidado. Si es demasiado largo, puede parecer que el sistema no responde; si es demasiado corto, el **context switch** y el mantenimiento del sistema ocupan más tiempo que el trabajo real.
- Los **procesos en I/O** ceden su quantum, pero en **multilevel feedback queues** se les premia por esto y "castigan" si consumen todo su quantum sin hacer I/O.

---

3. Prioridades

Se asigna un número de prioridad a cada proceso. Los procesos de mayor prioridad son ejecutados primero.

✅ **Pros**:

- Útil para asignar diferentes **importancias** a distintos procesos.
- **Flexible**: se pueden ajustar las prioridades para mejorar la equidad.

⚠️ **Contras**:

- Puede llevar a **inanición** si siempre llegan procesos de alta prioridad y los de baja prioridad nunca se ejecutan.
  - Se puede mitigar con **aging** (aumentar la prioridad de los procesos a medida que pasan el tiempo).
- Generalmente se **combina con Round Robin** para equilibrar la equidad y el rendimiento.

---

4. Shortest Job First (SJF)

Es un tipo de política de **prioridades**. Se ejecuta primero el proceso que tiene el menor tiempo de ejecución.

✅ **Pros**:

- **Optimiza el throughput** (número de procesos terminados por unidad de tiempo).

- Ideal para sistemas **batch** con trabajos de duración predecible.
- **Óptimo en cuanto a latencia promedio** si se conocen los tiempos de ejecución de antemano.

⚠️ **Contras**:

- Puede llevar a **inanición** de los procesos largos si siempre se ejecutan los cortos.
- No siempre se puede conocer con anticipación la duración de los procesos.

4.1. Shortest Remaining Job First (SRJF)

Variante preemptiva de SJF que considera el tiempo restante de cada proceso.

✅ **Pros**:

- **Preemptivo**: Si llega un proceso con menos tiempo restante, puede interrumpir a otro. Luego, se tiene en cuenta el tiempo restante del proceso que fue interrumpido.

⚠️ **Contras**:

- Al igual que SJF, puede causar **inanición** de procesos más largos.

---

5. Earliest Deadline First (EDF)

Otro tipo de política con prioridad. Utilizado en sistemas **de tiempo real**. Los procesos con el **deadline más cercano** son ejecutados primero.

✅ **Pros**:

- **Ideal para sistemas en tiempo real**, donde los procesos deben cumplir con fechas de finalización estrictas.

⚠️ **Contras**:

- No siempre puede garantizar la ejecución correcta si hay demasiados procesos.

---

6. Multilevel Queue

Los procesos se dividen en colas con diferentes niveles de prioridad. Los procesos de mayor prioridad, como los **interactivos**, se colocan en la cola de máxima prioridad.

✅ **Pros**:

- **Optimiza el tiempo de respuesta** para procesos interactivos.
- Se puede ajustar la prioridad de los procesos según su tipo (1. Tareas Real-Time, 2. Tareas interactivas, 3. Tareas de batches.).

⚠️ **Contras**:

- Los procesos largos pueden ser menos sensibles a las demoras si están en colas de baja prioridad.
- No siempre garantiza que todos los procesos reciban su debido tiempo de CPU si las colas están desbalanceadas.

---

7. Multilevel Feedback Queue

Los procesos pueden cambiar de cola según su uso de CPU. Si no terminan su quantum, pasan a una cola de menor prioridad. Si hacen I/O, pueden volver a la cola de mayor prioridad.

✅ **Pros**:

- **Flexible**: Los procesos pueden moverse entre colas según su comportamiento (por ejemplo, los procesos I/O intensivos reciben crédito extra).
- Minimiza la **inanición** de los procesos interactivos, premiándolos por sus ráfagas cortas de CPU.

⚠️ **Contras**:

- Requiere un manejo cuidadoso de las **colisiones de prioridad** y las reglas de transición de las colas.
- El sistema puede volverse **complejo** de gestionar con múltiples colas y políticas de prioridad.

💡

Sobre procesos...

- **Procesos RT**: Necesitan cumplir con fechas de finalización estrictas (ej. control industrial, imágenes médicas).
- **Procesos interactivos**: Son programas que esperan respuestas rápidas del usuario (ej. procesadores de texto).
- **Procesos batch**: Tareas periódicas o repetitivas que no dependen de la interacción del usuario (ej. copias de seguridad, generación de informes).

La complejidad a la hora de elegir una política o algoritmo de scheduling se incrementa cuando se manejan **threads**, **virtualización**, y escenarios específicos como **bases de datos**, **cómputo científico**, o **benchmarking**, lo que añade capas adicionales a la tarea de asignar recursos eficientemente.

Sincronizacion

Volvemos al principio: los SO tienen que manejar la **contención** y la **concurrencia** de manera correcta y con buen rendimiento.

Toda ejecución debería dar como resultado uno equivalente a alguna ejecución secuencial de los mismos procesos.

Ante esto, surge un problema:

Race condition

Error que ocurre en sistemas concurrentes cuando varios procesos o hilos acceden y modifican recursos compartidos al mismo tiempo, donde el resultado variará según el orden en que se ejecuten las tareas. Esto puede generar **resultados inconsistentes** o incorrectos.

Secciones críticas

Una sección crítica es una parte del código donde **solo un proceso puede ejecutarla a la vez** para evitar interferencias. Se debe asegurar que un proceso en espera pueda entrar, y ningún proceso fuera de la sección crítica pueda bloquear a otro.

Posibles soluciones:

1. Deshabilitar interrupciones

Desactivar temporalmente las interrupciones en una sección haría en un principio que fuera crítica, pero esta solución tienen limitaciones. Si bien garantiza el correcto uso de los recursos compartidos, desactivar interrupciones **elimina la multiprogramación**, lo que genera otros problemas.

2. Locks

Los locks usan variables booleanas para indicar si la sección está ocupada, pero volvemos al problema inicial: pueden también generar race conditions. Es decir, para usar un lock al entrar a una sección crítica, deberíamos tener otra sección crítica!

Una solución más general es aprovechar el **hardware** para garantizar la exclusión mutua sin los problemas anteriores.

2.1. TestAndSet (TAS)

En esta solución mediante **hardware,** se ofrecer una instrucción atómica llamada **TestAndSet**, que permite modificar una variable de forma indivisible, incluso en sistemas con múltiples CPUs. TAS pone la variable en 1 y devuelve su valor anterior, lo que permite implementar exclusión mutua.

Para usarlo en la sincronización de procesos, se implementa un **lock.**

El problema de este enfoque es la **busy waiting**, donde un proceso consume mucha CPU intentando obtener el lock sin realizar trabajo útil. Esto afecta el rendimiento del sistema y perjudica a otros procesos.

Una posible mejora es introducir **sleep**, pero elegir la duración adecuada es complicado: si es demasiado largo, se pierde tiempo; si es muy corto, se sigue desperdiciando CPU. La solución ideal es que el sistema operativo notifique al proceso cuando el lock esté disponible, algo que será resuelto con semáforos y otros mecanismos más avanzados.

⚠️

Deadlock

Ocurre cuando dos o más procesos se bloquean mutuamente, esperando cada uno a que el otro libere un recurso necesario para continuar. Esto resulta en un estancamiento donde ninguno de los procesos puede avanzar.

Para que ocurra un deadlock, deben cumplirse cuatro condiciones simultáneamente (**condiciones de Coffman**):

1. **Exclusión mutua**: Solo un proceso puede usar un recurso a la vez.
2. **Retención y espera**: Un proceso retiene un recurso mientras espera por otro.
3. **No apropiación**: Un recurso no puede ser quitado a un proceso, debe ser liberado voluntariamente.
4. **Espera circular**: Existe una cadena circular de procesos donde cada uno espera un recurso retenido por el siguiente.

Ejemplo: Si el Proceso A tiene el Recurso 1 y espera el Recurso 2, y el Proceso B tiene el Recurso 2 y espera el Recurso 1, ambos quedan en deadlock.

Para evitarlo, se usan técnicas como la prevención, detección o recuperación.

Sincronización con objetos

1. MutexLocks / SpinLocks (mediante Hardware)

Herramienta básica para proteger secciones críticas y evitar condiciones de carrera. Un proceso **debe adquirir el lock antes de entrar** a la **sección crítica** y **liberarlo al salir**.

1.1 TASLock

Implementación de un spin lock que utiliza TAS para garantizar la exclusión mutua, haciendo que los procesos esperen en un bucle (busy waiting) hasta que el lock esté disponible. Bloquea el bus compartido, lo que puede afectar el rendimiento en sistemas multiprocesador.

```
// Uso del TestAndSet para implementar un lock
bool lock = false; // Inicialmente, el recurso está libre

void criticalSection() {
   while (TestAndSet(&lock)) {
      // Espera activa (busy waiting)
   }
   // Código en la sección crítica
   lock = false; // Libera el recurso
```

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

1.2 TTASLock

Mejora sobre TASLock, donde el proceso primero verifica el estado del lock localmente antes de intentar adquirirlo. Reduce el tráfico en el bus compartido, mejorando el rendimiento.

Comparativa TASLock vs TTASLock:

- **TASLock**: Bajo rendimiento debido al bloqueo constante del bus compartido.
- **TTASLock**: Mejor rendimiento al reducir el tráfico en el bus, pero aún no es óptimo.
- **Busy Waiting**:
  - **Problemas**: Desperdicio de ciclos de CPU, especialmente en sistemas con scheduler preemptive. Puede llevar a inversión de prioridades y deadlock.
  - **Smart Scheduling**: Técnica para evitar el desperdicio de ciclos de CPU al detectar cuándo un proceso tiene un spin lock.

2. Sleep y Wakeup (mediante Software)

Mecanismo para evitar el busy waiting. Un proceso puede dormir (`sleep`) hasta que otro proceso lo despierte (`wakeup`).

- **Problema del Productor-Consumidor**: Ejemplo clásico de sincronización donde el productor y el consumidor deben coordinarse para evitar condiciones de carrera.
- **Lost Wake-Up Problem**: Ocurre cuando una señal de `wakeup` se pierde porque el proceso destinatario no estaba dormido aún.

2.1 Semáforos

Herramienta de sincronización que permite contar señales (contar la cantidad de wakeups) y evitar el problema del wakeup perdido. Su caso particular en el que se usa con valor máximo 1 es un mutex.

- **Operaciones**: `wait` (decrementa el semáforo y bloquea si es negativo) y `signal` (incrementa el semáforo y despierta un proceso si es necesario).
- **Uso en Productor-Consumidor**:
  - **mutex**: Garantiza exclusión mutua en el acceso al buffer.
  - **full_slots**: Cuenta los slots ocupados en el buffer.
  - **empty_slots**: Cuenta los slots vacíos en el buffer.

Métodos de Spinning vs Blocking:

- **Spinning (Busy Waiting)**: Útil cuando se espera que el lock esté disponible en un corto período de tiempo.

- **Blocking**: Más eficiente cuando el tiempo de espera es largo, ya que evita el desperdicio de ciclos de CPU.

Comparación

Aspecto	Spinning (Busy Waiting)	Blocking
Consumo de CPU	Alto (el proceso sigue ejecutándose).	Bajo (el proceso se suspende).
Overhead	Bajo (no hay cambio de contexto).	Alto (cambio de contexto necesario).
Eficiencia	Mejor para esperas cortas.	Mejor para esperas largas.
Uso en sistemas	Multiprocesador.	Monoprocesador o multiprocesador.
Ejemplo de uso	TASLock, TTASLock, Mutex	Sleep y Wakeup, Semáforos

Conclusión

- **Mutex Locks**: Útiles para proteger secciones críticas, pero el *busy waiting* puede ser costoso en términos de rendimiento.
- **Sleep y Wakeup**: Permiten evitar el *busy waiting*, pero requieren mecanismos adicionales como semáforos para evitar problemas como el Lost Wake-Up.
- **Semáforos**: Herramienta poderosa para sincronización, especialmente en problemas como el del Productor-Consumidor.
- **Spinning vs Blocking**: La elección entre spinning y *blocking* depende del tiempo de espera esperado y del contexto del sistema.

Correctitud de programas concurrentes

La corrección de estos sistemas se aborda así:

- Definir propiedades de **safety**: evitar eventos indeseados (como *deadlocks*).
- Definir propiedades de **liveness**: garantizar que ocurran eventos deseables.
- Demostrar que su combinación asegura el comportamiento esperado.

Método Rendezvous

Una vez que N-1 procesos realizan `a(i)`, el proceso número N abre la barrera a uno, y el resto se van levantando la barrera entre ellos, y así sólo pueden iniciar a realizar `b(i)` una vez todos hayan realizado `a(i)`.

```
atomic<int> cant = 0;  // Procesos que han terminado a
semaphore barrera = 0;  // Barrera inicialmente baja

proc P(i) {
   a(i);
   // TRY: ¿Se puede ejecutar b?
   if (cant.getAndInc() < N - 1) {
      // No, esperar en la barrera.
```

```
        barrera.wait();
    }

    // Sí, entrar y avisar al resto.
    barrera.signal();

    // CRIT: Ejecutar b.
    b(i);
}
```

Monitores

Un **monitor** es una estructura que encapsula datos compartidos junto con las operaciones que se pueden realizar sobre ellos, garantizando que estas operaciones sean **mutuamente excluyentes**. Esto significa que en un momento dado, solo un hilo puede ejecutar un método del monitor, evitando condiciones de carrera y garantizando la consistencia del **objeto compartido**.

El monitor se basa en dos mecanismos clave:

1. **Exclusión mutua**: Se usa un mutex (o mecanismo similar) para asegurarse de que solo un hilo pueda acceder a los métodos del monitor a la vez.
2. **Variables de condición**: Permiten que los hilos esperen a que se cumpla una condición específica antes de continuar, facilitando la sincronización entre hilos.

Memoria

La memoria se comparte no solo para comunicación entre procesos, sino también para la multiprogramación. El sistema operativo debe:

- Manejar el espacio libre y ocupado.
- Asignar y liberar memoria.
- Controlar el swapping.

Swapping

Si solo hay un proceso en memoria, no hay conflictos. En multiprogramación, cuando un proceso se bloquea y otro toma su lugar, se usa **swapping**: mover procesos inactivos al disco y traer otros a memoria. Es una solución simple pero lenta. Mantener procesos en memoria mientras sea posible mejora el rendimiento, pero puede generar problemas de ubicación al reactivarlos.

Problemas comunes

- **Reubicación**: cambio de procesos sin modificar direcciones, swapping.
- **Protección**: evitar accesos indebidos entre procesos.

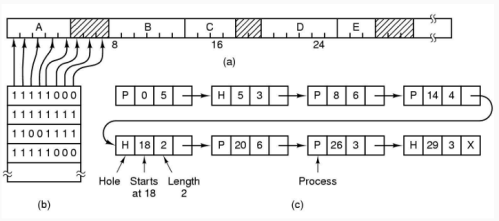
- **Manejo del espacio libre**: asignación eficiente de memoria, evitar la **fragmentación,** que desperdicia memoria.

Fragmentación

La fragmentación de memoria impide su uso eficiente al no ser continua. La compactación es costosa e inviable en sistemas en tiempo real, por lo que es mejor prevenirla mediante una organización adecuada.

Organización de memoria

- **Bitmap**: Divide la memoria en bloques iguales (digamos 4KB), marcando cada uno como libre (0) u ocupado (1). Su asignación es simple, pero buscar bloques consecutivos es costoso. Poco utilizado.
- **Lista enlazada**: Cada nodo representa un proceso o un bloque libre con su tamaño y límites. Liberar es O(1) y asignar depende de la estrategia de ubicación.

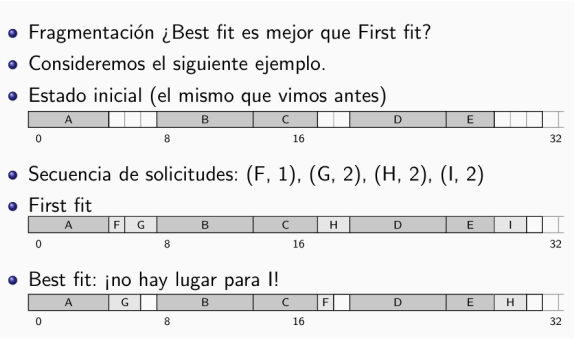


Pedazo de memoria con 5 procesos y 3 espacios libres

Bitmaps y lista enlazada

Asignación de bloques

- Estrategias:
  - *First fit*: Asigna en el primer bloque donde cabe. Es rápido, pero fragmenta la memoria.
  - *Best fit*: Busca el ajuste más exacto, pero genera pequeños bloques inútiles.
  - *Quick fit*: Mantiene listas de bloques libres con tamaños comunes.
  - *Buddy system*: Divide bloques en tamaños específicos.
- **Problemas comunes**: Estas estrategias suelen ser ineficientes, causando *fragmentación externa* (bloques pequeños dispersos) o *fragmentación*



En Best Fit, cuando llega (F,1), la coloca en donde hay 2 espacios libres, ya que así sobra 1 (minimiza el espacio libre restante donde se ubique), pero esto evita luego que G, H o I se ubiquen allí.

interna (espacio desperdiciado dentro de los bloques).

> 💡 En la práctica, se usan esquemas más sofisticados que consideran la distribución de los pedidos. Asumimos que ya está solucionado

---

Memoria virtual

Permite ejecutar programas más grandes que la memoria física combinando *swapping* y virtualización de direcciones.

Requiere la **MMU (Memory Management Unit)** para gestionar las traducciones.

Funcionamiento

- Sin memoria virtual, las direcciones corresponden directamente a la memoria física.
- Con memoria virtual, las direcciones se traducen antes de acceder a la memoria, permitiendo el uso de *swap* y optimizando recursos, ya que:
  1. Permite ejecutar programas más grandes que la memoria física, cargando solo las partes necesarias en cada momento (*paginación por demanda*).
  2. Mejora la utilización de la memoria al compartir páginas entre procesos sin duplicar datos.
  3. Reduce la fragmentación al usar direcciones virtuales, evitando los problemas de memoria contigua.
  4. Facilita la multitarea al aislar procesos y proteger su espacio de memoria.
  5. Usa *swap* en disco, liberando memoria RAM para tareas prioritarias.

Paginación y traducción de direcciones

La **memoria virtual se divide en *páginas*** y la **física en *marcos*** *(page frames)*.

La **MMU** traduce direcciones virtuales en físicas usando una T**abla de Páginas** (P*age Table*).

Si una página no está en memoria, ocurre un *page fault* y el SO la carga desde el disco, reemplazando otra si es necesario.

Gestión de la *Page Table*

Queremos que la búsqueda sea rápida y que la tabla no ocupe mucho espacio en memoria.

Para optimizar espacio y velocidad, se usa una **tabla de páginas multinivel**, permitiendo cargar solo las partes necesarias en memoria, gracias a su estructura jerárquica. Ejemplo: Si un proceso solo usa unas pocas páginas, no hace falta cargar entradas en memoria para las que nunca se usan.

- **Entradas de la tabla de páginas**: Contienen el *page frame*, bit de presencia, bits de protección, bit *dirty* (modificación desde el disco) y bit de referencia (indica si fue usada recientemente).

Memoria Asociativa (Translation Lookaside Buffer)

Como la tabla de páginas está en memoria, acceder a ella en cada referencia puede hacer que el rendimiento caiga drásticamente.

Para evitarlo, se usa un **caché especializado** llamado **TLB (Translation Lookaside Buffer)**, que almacena las traducciones recientes de direcciones virtuales a físicas.

- Si la dirección consultada está en la TLB, la traducción es inmediata.
- Si no, se busca en la tabla de páginas y se carga en la TLB para agilizar futuros accesos.

Reemplazo de Páginas

Cuando la memoria está llena y se necesita cargar una nueva página, es necesario **desalojar** otra. La estrategia de reemplazo afecta el rendimiento.

- **Algoritmo Óptimo (teórico)**: Reemplaza la página que tardará más en ser utilizada nuevamente. Es ideal pero imposible de implementar en tiempo real.
- **FIFO (First-In, First-Out)**: Saca la página más antigua. Es simple, pero puede eliminar páginas aún en uso (*Anomalía de Belady*).
- **Segunda Oportunidad**: Variante de FIFO que revisa si la página ha sido referenciada antes de eliminarla. Si lo fue, se le da una "segunda oportunidad" y se pasa a la siguiente.
- **NRU (Not Recently Used)**: Clasifica páginas según si han sido referenciadas y/o modificadas. Prioriza la eliminación de páginas menos usadas y no modificadas.
- **LRU (Least Recently Used)**: Saca la página que no se ha usado en más tiempo, basándose en marcas de tiempo o estructuras como pilas. Es muy eficiente pero costoso en hardware.

Page Faults

Un **page fault** ocurre cuando un programa intenta acceder a una página que no está actualmente en la memoria RAM. El proceso que sigue el sistema operativo (SO) cuando se produce un page fault es el siguiente:

1. **Interrupción**: Ocurre un page fault y el kernel guarda el contador de programa (IP) y otros registros en la pila.
2. **Identificación**: El kernel identifica que es un page fault y verifica la dirección virtual solicitada.
3. **Validación**: Se chequea la validez de la dirección y los permisos del proceso. Si no son válidos, el proceso se termina.
4. **Asignación de Page Frame**: Se busca un page frame libre. Si no hay, se aplica un algoritmo de reemplazo para liberar espacio.

5. **Manejo de Páginas Dirty**: Si la página a reemplazar está modificada (bit dirty activado), se escribe en disco antes de continuar.
6. **Cargar la Nueva Página**: Se realiza la operación de E/S para cargar la nueva página en memoria.
7. **Actualización de la Tabla de Páginas**: Se actualiza la tabla de páginas para indicar que la nueva página está cargada.
8. **Reintento de la Instrucción**: Se reanuda la instrucción que causó el page fault usando los registros restaurados.
9. **Control de Vuelta al Proceso**: Se devuelve el control al proceso de usuario para que continúe su ejecución con la página ya en memoria.

Thrashing

Ocurre cuando la memoria es insuficiente y hay mucha competencia entre procesos, lo que provoca un intercambio constante de páginas entre memoria y disco.

Resulta en un alto costo en términos de tiempo, ya que la mayoría del tiempo se dedica a la **gestión de memoria y mantenimiento** en lugar de realizar trabajo productivo.

Protección y Reubicación

- **Protección**: Cada proceso tiene su propia tabla de páginas, lo que impide el acceso a las páginas de otros procesos, garantizando la protección de memoria.
- **Segmentación**: Se asigna un espacio de memoria único a cada proceso llamado segmento. Esto facilita la protección y permite que los segmentos crezcan sin modificar el programa.
- **Bibliotecas compartidas**: La segmentación también facilita el uso de bibliotecas compartidas, que pueden estar ubicadas en sus propios segmentos.

Segmentación

- **Visibilidad**: A diferencia de las páginas, los segmentos son visibles para el programador (especialmente en assembler).
- **Tamaño variable**: No tienen un tamaño fijo, lo que puede causar fragmentación y necesidad de swapping.
- **Combinación con paginación**: Se usa comúnmente para mitigar problemas de fragmentación.

Segmentación en Intel (Pentium)

- **Tablas de descriptores**: definen los segmentos accesibles por los procesos y SO
  - **LDT (Local Descriptor Table)**: Cada proceso tiene la suya.
  - **GDT (Global Descriptor Table)**: Compartida entre procesos, con segmentos del sistema.

- **Registros (DS y CS, de 16 bits)**:
  - Controlan el segmento en uso.
  - Si se modifican incorrectamente, generan un *trap* manejado por el SO.

Segmentación vs. Paginación

- **Segmentación**: Espacios de memoria separados para el mismo proceso, con mejor protección.
- **Paginación**: Transparente para el programador y evita fragmentación externa.
- **Mejor enfoque**: La combinación de ambos.

Copy-on-Write (COW)

Estrategia para procesos creados con `fork()`.

- **En lugar de copiar toda la memoria**, se comparten páginas hasta que uno de los procesos las modifica.
- **Optimiza el uso de memoria** y reduce copias innecesarias.

Administración de memoria en la API

- **Explícita**:
  - Uso de bibliotecas como libc (`malloc()`, `free()`) o primitivas (`new`, `delete` en C++).
- **Implícita**:
  - Recolección automática de memoria (*Haskell*).
  - Control parcial (*Swift*).
- **Híbrida**:
  - Creación manual, liberación automática, *garbage collector* (*Java*).
  - Uso de bibliotecas y análisis en compilación (*RTSJ*).
  - Administración integrada en sistemas móviles (*Android, iOS*).

Entrada/salida

Tradicionalmente, los sistemas operativos han debido gestionar distintos tipos de almacenamiento, entre ellos:

- **Discos duros (HDD y SSD)**: Siguen siendo la principal preocupación en la administración del almacenamiento, ya que contienen el sistema operativo, aplicaciones y datos del usuario.
- **Unidades de cinta**: Aunque su uso ha disminuido, todavía se emplean principalmente para copias de seguridad y almacenamiento a largo plazo debido a su alta capacidad y bajo

costo por terabyte.

- **Dispositivos de almacenamiento removibles**: Incluyen disquetes (ya en desuso), CDs, DVDs y unidades USB, que permiten transportar datos fácilmente.

Almacenamiento en red

Además del almacenamiento local, existen soluciones de almacenamiento remoto, que permiten acceder a datos a través de la red:

- **Discos virtuales**: Se encuentran en servidores remotos y son accesibles mediante la red, funcionando como si fueran unidades locales.
- **Sistemas de archivos en red (NFS, CIFS, DFS, AFS, Coda)**: Permiten compartir archivos entre múltiples dispositivos y usuarios en una red. Este tipo de almacenamiento se agrupa bajo el término **Network Attached Storage (NAS)**, donde los datos se almacenan en servidores dedicados accesibles a través de protocolos estándar de red.

Alternativas avanzadas

- **Storage Area Network (SAN)**: A diferencia de NAS, una SAN no se basa en protocolos de red convencionales, sino en una red especializada de alta velocidad diseñada específicamente para el acceso a datos. Utiliza protocolos de bajo nivel optimizados para almacenamiento, como Fibre Channel o iSCSI, ofreciendo mayor rendimiento y menor latencia en comparación con NAS.

Nos vamos a concentrar principalmente en los **dispositivos de almacenamiento**.

Un dispositivo de **E/S** se conceptualiza en dos partes:

1. **Dispositivo físico**: La parte tangible del hardware, que realiza la acción física (por ejemplo, un disco, una impresora, etc.).
2. **Controlador del dispositivo**: El componente que interactúa con el sistema operativo a través de un bus o registros, gestionando la comunicación y control del dispositivo.

Drivers

Los **drivers** son componentes de software altamente especializados que conocen las características del hardware con el que interactúan. Incluso dentro de un mismo fabricante, distintos modelos de dispositivos pueden requerir drivers diferentes.
Por ejemplo, para indicar el fin de una operación, puede ser necesario leer el **segundo** o el **cuarto bit**; esta información solo la conoce el driver.

Características clave de los drivers

- Tienen **máximo privilegio** en el sistema, lo que significa que pueden afectar el funcionamiento de todo el sistema si algo sale mal.
- Son fundamentales para el **rendimiento de E/S**, que a su vez impacta directamente en el rendimiento global del sistema.

Interacción con los dispositivos

1. **Polling:** El driver verifica periódicamente si el dispositivo se ha comunicado.
   - **Ventajas**: Sencillo, cambios de contexto controlados.
   - **Desventajas**: Consume CPU innecesariamente.
2. **Interrupciones:** El dispositivo genera una interrupción para avisar que ha ocurrido un evento.
   - **Ventajas**: Eventos asincrónicos poco frecuentes, ahorra CPU.
   - **Desventajas**: Cambios de contexto impredecibles, puede afectar el rendimiento.
3. **DMA (Acceso Directo a Memoria):** Permite transferir grandes volúmenes de datos sin intervención de la CPU, utilizando un controlador de DMA.
   - **Ventajas**: Eficiente para grandes transferencias de datos.
   - **Desventajas**: Requiere hardware adicional y puede generar interrupciones cuando termina la transferencia.
4. **Spooling:** técnica para manejar dispositivos que requieren acceso dedicado en sistemas multiprogramados. Un ejemplo típico es la **impresora**. El trabajo se coloca en una **cola** y un proceso se encarga de desencolarlo a medida que el dispositivo se libera.
   - El **kernel** no detecta el spooling, pero **el usuario sí**.

El
subsistema de E/S proporciona al programador una API sencilla con funciones como `open()`, `close()`, `read()`, `write()`, y `seek()`.
Sin embargo, hay aspectos que no deben ser ocultados, como la necesidad de ciertas aplicaciones de conocer si no obtuvieron acceso exclusivo a un dispositivo. La misión del SO es gestionar esto de manera correcta y eficiente, siendo responsabilidad compartida entre el **manejador de E/S** y los **drivers**.

Clasificación

Los dispositivos se pueden clasificar en dos grupos:

1. **Char device:** Son dispositivos donde la información se transmite **byte a byte**.
   - **Ejemplos**: Mouse, teclado, terminales, puerto serie.
   - **Características**: Acceso secuencial, no soportan acceso aleatorio y no utilizan caché.
2. **Block device:** Son dispositivos donde la información se transmite en **bloques**.
   - **Ejemplos**: Disco rígido, memoria flash, CD-ROM.
   - **Características**: Permiten acceso aleatorio y generalmente utilizan un buffer (caché).

Linux/dev

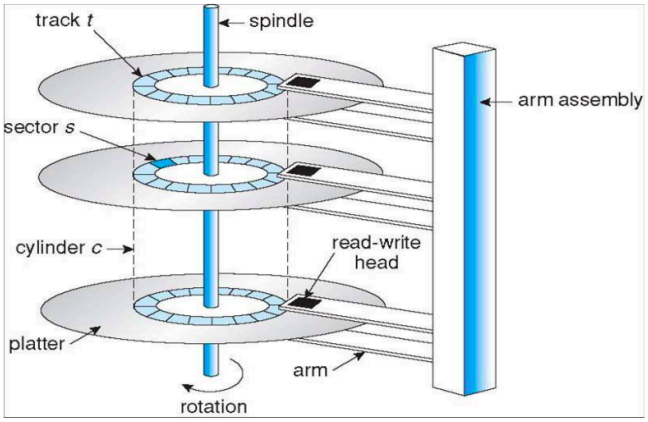
Permisos	Usuario	Grupo	Mayor, Menor	Fecha y Hora	Dispositivo
crw-rw-rw-	root	wheel	17, 1	Apr 27 07:41	cu.Bluetooth
brw-r---	root	operator	1, 0	Apr 27 07:41	disk0
crw-rw-rw-	root	wheel	24, 2	Apr 27 07:41	dtrace
crw-r---	root	operator	1, 0	Apr 27 07:41	rdisk0

La **interacción con dispositivos** tiene características como ser de lectura, escritura o lecto-escritura, con acceso secuencial o aleatorio, y puede ser compartida o dedicada. La comunicación puede ser de caracteres o bloques y ser sincrónica o asincrónica, además de tener diferentes velocidades de respuesta. El sistema operativo se encarga de proporcionar un acceso consistente a todos los dispositivos, ocultando sus particularidades.

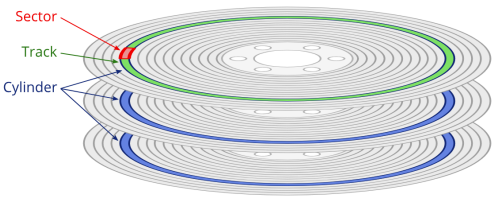
En cuanto a la **API del subsistema de E/S**, todo se trata como un archivo, y se ofrecen funciones como `fopen`, `fclose`, `fread`, `fwrite`, entre otras, para facilitar la manipulación de archivos.

Planificación de E/S

El objetivo es **optimizar el rendimiento de E/S** minimizando los movimientos de la cabeza del disco.



- Factores clave:
  -
  Tiempo de búsqueda (seek time): Tiempo para mover la cabeza al cilindro correcto.
  -
  Latencia rotacional: Tiempo para que el disco gire al sector deseado.
  -
  Ancho de banda: Cantidad de bytes transferidos por unidad de tiempo.



Políticas de Scheduling de E/S

1. **FIFO/FCFS**:
   - Atiende las solicitudes en orden de llegada.

- Problema: Movimientos innecesarios de la cabeza (ineficiente). Ejemplo : la cola tiene pedidos para los cilindros: 20, 200, 10

2. **SSTF (Shortest Seek Time First)**:
   - Atiende la solicitud más cercana a la posición actual de la cabeza.
   - Mejora los tiempos de respuesta, pero puede causar **inanición** (algunas solicitudes nunca se atienden).
3. **SCAN (Ascensor)**:
   - La cabeza se mueve en una dirección, **atendiendo solicitudes en el camino**, y **luego cambia de dirección**. Ejemplo: llega una solicitud para el cilindro inmediato anterior, pero hay que esperar a que cambie de dirección.
   - Evita la inanición, pero puede generar tiempos de espera desiguales.

En la práctica, se usan **combinaciones de algoritmos** con prioridades (ejemplo: swapping de procesos o gestión de caché).

Tecnología de Discos

Característica	HDD (Hard Disk Drive)	SSD (Solid State Drive)
Tecnología	Usa **componentes mecánicos**: discos giratorios y cabezas móviles. La lectura y escritura es más compleja y lenta.	No tiene partes móviles, utiliza **memoria flash** para almacenar datos, lo que proporciona mayor velocidad, resistencia y eficiencia energética.
Escritura	La escritura debe minimizar la distancia entre bloques escritos para mayor eficiencia, además de manejar problemas como la fragmentación.	Acceso a cualquier celda de memoria flash, la ubicación no importa tanto, pero requiere escribir uniformemente para evitar desgaste (*wear leveling*). Además, no se puede sobrescribir directamente, se debe borrar primero el bloque (comando TRIM). Permite el acceso concurrente mediante NVMe.
Lectura	Requiere movimientos físicos de disco y cabezal, lo que aumenta la latencia debido a los algoritmos de movimiento para una lectura eficiente.	No tiene partes móviles, por lo que no hay movimiento físico, resultando en una latencia mucho menor. Permite el acceso concurrente mediante NVMe.
Fragmentación	La fragmentación es un problema importante en su sentido físico y debe ser gestionado para evitar la pérdida de rendimiento.	La fragmentación no afecta tanto el rendimiento debido al acceso directo a celdas de memoria flash, aunque sí en la gestión virtual de la misma.
Duración	Vida útil de 3-5 años, debido al desgaste mecánico (componentes móviles).	Vida útil de 5-10 años, dependiendo del uso y la tecnología de la memoria flash.
Protocolos	Usualmente usa **SATA** o **SAS**; la velocidad está limitada por la mecánica del dispositivo.	Usa **SATA** (limitado) o **NVMe** (más rápido y eficiente). Para un rendimiento óptimo, **NVMe** es la mejor opción.

Rendimiento	Más lento en comparación con SSD debido a las limitaciones mecánicas (movimiento de los discos y cabezales).	Mucho más rápido que HDD, especialmente con **NVMe**. Ofrece tiempos de acceso rápidos y mayor velocidad de lectura/escritura.

Gestión del Disco

1. **Formateo**: Es el proceso de preparar el disco para almacenar datos. Durante el formateo, se agregan códigos (pre y post fijo) de detección y corrección de errores en cada sector para asegurar la integridad de los datos.. Si los códigos no coinciden, el sector está dañado.
2. **Booteo**: Un programa en ROM carga sectores iniciales del disco y ejecuta un **cargador del SO**.
3. **Bloques Dañados**: Se manejan por software (ejemplo: FAT) o hardware (discos SCSI). Los discos SCSI tienen sectores extra para reemplazar bloques defectuosos.

Protección de la Información

La protección se puede basar en políticas de respaldo y resguardo.

Copias de Seguridad

Esencial para proteger datos importantes. Se pueden realizar copias **completas** de forma periódica y copias **incrementales** (sólo de archivos modificados desde la última copia) o **diferenciales** (sólo desde la última copia total).

Redundancia

La redundancia, como **RAID (Redundant Array of Inexpensive Disks)**, busca evitar pérdidas de datos y mejorar la disponibilidad del sistema. RAID permite realizar copias en tiempo real, lo que permite que, si un disco falla, los datos sigan disponibles.

RAID no protege contra la eliminación accidental de archivos ni la corrupción de datos. Para eso, se deben combinar con **copias de seguridad** y usar sistemas de archivos con protección adicional.

Tipo de RAID	Ventajas	Desventajas	Uso principal	Término clave
RAID 0	Alto rendimiento, mejora el **ancho de banda**, permite **lecturas en paralelo**	No ofrece redundancia	Mejora el rendimiento (sin redundancia)	**Stripping**
RAID 1	Alta **disponibilidad**, mejora las **lecturas**	Alto costo, duplicación de datos	Alta disponibilidad y protección de datos	**Mirroring**
RAID 0+1	Mejor rendimiento de **lecturas**, ofrece **redundancia**	Requiere al menos 4 discos, si falla un disco en un par de espejos, se pierde todo el volumen	Combina **RAID 0 (stripping)** y **RAID 1 (mirroring)**	**Stripping** y **Mirroring**

RAID 2 y 3	Usan **información de paridad** para detectar y corregir errores	Todos los discos participan en cada operación de E/S, lo que puede ser lento	Utilizan **información de paridad** para recuperación de errores	**Paridad**
RAID 4	Mejora el rendimiento de lectura, permite **striping a nivel de bloque**	El disco dedicado a la paridad es un **cuello de botella** para las escrituras	Similar a RAID 3, pero con **striping a nivel de bloque**	**Striping**
RAID 5	Buena eficiencia en espacio, no hay un disco dedicado a la paridad	La **escritura** es más lenta, el rendimiento se degrada al reconstruir un disco perdido	**Distribución de paridad** entre los discos	**Paridad distribuida**
RAID 6	Soporta el fallo de hasta **dos discos**, alta redundancia	**Espacio desperdiciado** por la doble paridad, más lento que RAID 5	Similar a RAID 5, pero con **doble paridad**	**Paridad doble**

File Systems

Archivo: Secuencia de bytes sin estructura específica, identificada por un nombre. Puede incluir una extensión para indicar su contenido (ejemplo: `.txt`, `.c`, `.tex`).

Un **File System** (FS) es un módulo dentro del kernel que organiza la información en el disco. Los sistemas Unix modernos pueden soportar múltiples FS mediante módulos dinámicos.

- Ejemplos de FS populares: ext2, ext3, ext4, XFS, ZFS, ISO-9660.
- También existen **file systems distribuidos**, que almacenan datos en varias máquinas en red (ejemplo: NFS, DFS, SMBFS).

Responsabilidades del FS

- **Organización interna**: Cómo se estructura la información dentro del archivo. En Windows y Unix es una secuencia de bytes, quedando la responsabilidad al usuario.
- **Organización externa**: Cómo se ordenan los archivos. Se usan directorios jerárquicos en forma de árbol.
- **Links**: Alias o nombres alternativos para un archivo, transformando la estructura en un grafo dirigido (con ciclos posibles).
- **Nombrado de archivos**: extensión, mayúscula o minúscula, contrabarra o barra, etc

Representación de archivos en un FS

El FS debe gestionar la representación de archivos, formados por:
- bloques de datos,
- metadatos,

La disposición de los bloques de un archivo de manera contigua en un disco supone dos grandes problemas:

- Si un archivo crece y no hay mas espacio, cómo lo represento?
- Qué hago con la fragmentación

Para evitar esto, se puede imaginar como una **lista enlazada de bloques**, donde cada bloque apunta al siguiente. Esto permite archivos dinámicos pero ralentiza las lecturas aleatorias y desperdicia espacio.

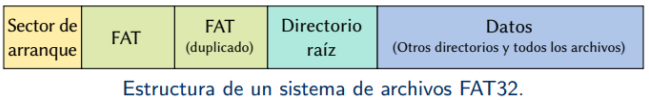
FAT (File Allocation Table)

FAT usa una tabla donde cada bloque indica el siguiente.
Soluciona los problemas mencionados, porque no desperdicia espacio del bloque y porque al tener en memoria todos los bloques que necesita, pueden leerse fuera de orden y es eficiente para lecturas no secuenciales.

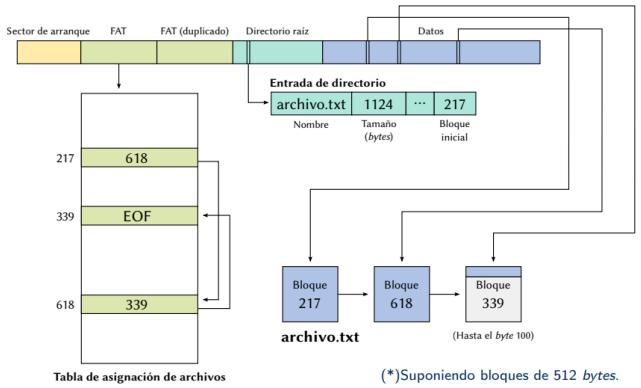
Directorios en FAT

Consiste en una lista de entradas de tamaño fijo. Cada entrada de directorio indica el índice del primer bloque de cada archivo.

La entrada de directorio almacena también todos los metadatos: nombre (tamaño fijo), tamaño, fecha de último acceso, etc.
El bloque del directorio root es distinguido. De esta forma, podemos encontrar cualquier archivo a partir de su ruta.



Estructura de un sistema de archivos FAT32.



(*)Suponiendo bloques de 512 *bytes*.

Desventajas

- Sin embargo, **requiere mantener la tabla en memoria**, lo que puede ser un problema en discos grandes; mucha contención.
- Es poco robusto: si el sistema falla, la tabla en memoria puede perderse.
- No maneja seguridad.

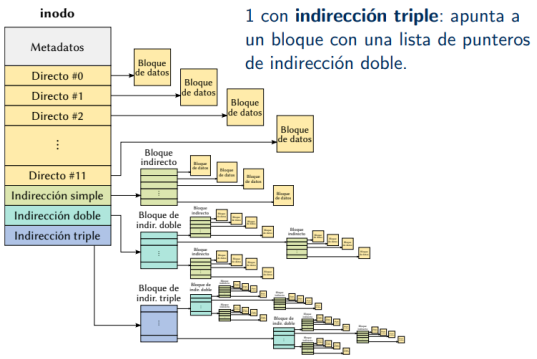
Inodos

Solución utilizara en los Unix, donde cada archivo tiene un inodo que almacena sus atributos y punteros a bloques de datos.

- Optimiza el acceso a archivos pequeños.
- Usa punteros directos e indirectos para gestionar archivos grandes:
  - *Single indirect block*: hasta 16 MB.
  - *Double indirect block*: hasta 32 GB.
  - *Triple indirect block*: hasta 70 TB.
- Solo mantiene en memoria las tablas de archivos abiertos, reduciendo la contención y mejorando la consistencia.

Directorios

Se reserva un inodo como entrada al root directory.
Por cada archivo o directorio dentro del directorio
hay una entrada. Dentro del bloque se guarda una lista de (inodos, nombre de archivo/directorio). El directorio root tiene un inodo reservado.



Es decir,

- si estamos viendo el inodo de un archivo, sus bloques de datos son los datos del archivo,
- si estamos viendo el inodo de un directorio, sus bloques de datos contienen entradas de directorio, que mapean a otros inodos (los inodos de archivos o directorios dentro del mismo)

Entradas de directorio

Tienen longitud variable para poder almacenar nombre no acotado y metadatos.
Pueden ocupar más de un bloque!

Links y atributos

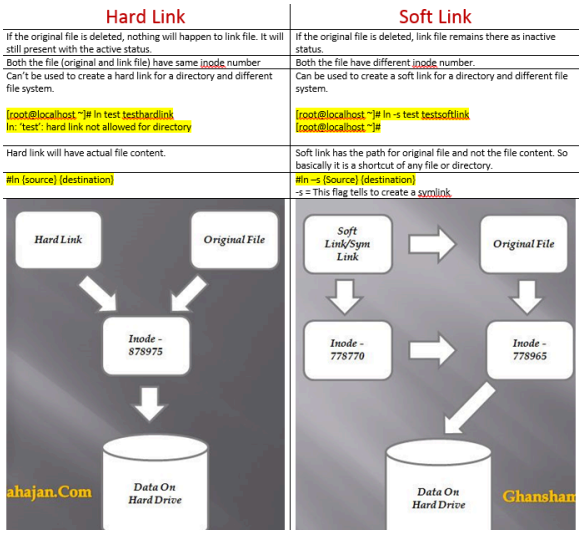
En los FS con inodos, **el nombre de los archivos no aparece en los inodos**. Así, **podemos referenciar el mismo inodo con diferentes nombres**, y desde más de un directorio. Esto se conoce como hard link.

- **Hard links (`ln`)** crean múltiples nombres para un mismo archivo.

El nombre de archivo "." en un directorio es un enlace duro al mismo directorio. El nombre de archivo ".." es un enlace duro al directorio padre.

- **Links simbólicos (`ln -s`)** apuntan a otro archivo o directorio. Estos se implementan mediante un inodo adicional, donde se almacena el destino del enlace.

Los **atributos** incluyen permisos, propietarios, fechas, tipo de archivo, flags y conteo de referencias.



Manejo del Espacio Libre

Existen varias técnicas para abordar este problema:

1. **Mapa de bits empaquetado**:

   Se utiliza un mapa de bits donde cada bit representa un bloque de disco. Un bit en `1` indica que el bloque está libre. Esta técnica permite saltar rápidamente bloques ocupados con una sola comparación, pero requiere mantener el vector en memoria, lo que puede ser ineficiente.

2. **Lista enlazada de bloques libres**:

   En este enfoque, los bloques libres se organizan en una lista enlazada. Cada bloque contiene punteros a otros bloques libres. Por ejemplo, si un bloque puede almacenar `n` punteros, los primeros `n-1` apuntan a bloques libres, y el último apunta al siguiente nodo de la lista.

   2.1. **Refinamiento con contadores**:

   Una mejora a la lista enlazada es incluir un contador en cada nodo que indique cuántos bloques libres consecutivos hay a partir de ese punto. Esto optimiza la gestión de espacios contiguos.

Caché

Para mejorar el rendimiento, los sistemas operativos utilizan un **caché** en memoria que **almacena copias de bloques de disco**. Este caché funciona de manera similar al manejo de páginas en memoria. Los sistemas operativos modernos emplean un caché unificado para archivos y páginas, evitando duplicaciones cuando los archivos se mapean en memoria.

- **Ventajas del caché**:
  - Permite escrituras ordenadas, lo que facilita una planificación más eficiente de las operaciones de E/S.
  - Las aplicaciones pueden configurarse para usar escritura sincrónica (escribir en disco inmediatamente), aunque esto es más lento.

Consistencia en FS

La **consistencia** en un sistema de archivos significa que los datos y la estructura del sistema de archivos (como las carpetas, archivos e inodos) están en un estado válido y correcto.

El problema ocurre cuando hay un **fallo inesperado**, como un corte de energía o un bloqueo del sistema, antes de que los cambios en los archivos se guarden correctamente en el disco. Esto puede hacer que:

- Un archivo recién creado desaparezca porque nunca se escribió en el disco.
- Un archivo quede en un estado corrupto porque se guardó solo una parte de la información.
- La estructura del sistema de archivos tenga errores, como que haya bloques de datos marcados como "libres" cuando en realidad estaban en uso.

Para evitar esto, los sistemas operativos usan **herramientas y técnicas** que garantizan que los datos sean consistentes incluso después de un fallo.

Herramientas de recuperación y `fsync()`

Cuando escribes un archivo, los datos **primero se guardan en caché (RAM)** para hacer el sistema más rápido. El problema es que si el sistema falla antes de que la caché se escriba en el disco, se pierden los cambios.

- `fsync()`: Es una llamada del sistema que **fuerza a que los datos en caché se escriban en el disco inmediatamente** cuando se dispara el corte de energía, asegurando que los cambios no se pierdan.
- `fsck` (File System Consistency Check): Es una herramienta que **verifica la integridad del sistema de archivos** y **corrige inconsistencias** si el sistema no se apagó correctamente.

Ejemplo:

1. Guardas un documento en un editor de texto.
2. El sistema operativo lo guarda en caché (memoria RAM) para optimizar el rendimiento.
3. Justo antes de que la caché se escriba en el disco, hay un corte de energía.

4. Cuando el sistema reinicia, `fsck` revisa si hay archivos o metadatos dañados y trata de repararlos.

Técnicas para mejorar la consistencia

Algunos sistemas han desarrollado mecanismos para mejorar la consistencia sin afectar mucho el rendimiento:

1. **Soft Updates**
   - Controlan **el orden en el que se escriben los metadatos**, asegurándose de que el sistema de archivos siempre esté en un estado seguro.
   - Permiten que algunas verificaciones se hagan en segundo plano, lo que **reduce el tiempo de recuperación** tras un fallo.
2. **Journaling (Registro de cambios)**
   - En lugar de escribir directamente en el disco, primero **se guardan los cambios en un "diario" o “journal” (log)**.
   - Si hay un fallo, el sistema usa este diario para recuperar los cambios pendientes en lugar de escanear todo el disco.
   - Ejemplos de sistemas de archivos que usan journaling:
     - **ext3 / ext4** en Linux
     - **NTFS** en Windows
     - **ZFS** en sistemas avanzados

¿Cómo funciona el Journaling?

Imagina que quieres modificar un archivo en un sistema con journaling:

1. El sistema **primero escribe los cambios en un "diario"** (un log en el disco).
2. Luego, los datos reales se escriben en el sistema de archivos.
3. Una vez que los datos están correctamente guardados, se borra la entrada del diario.

Si hay un **fallo inesperado**, al reiniciar el sistema simplemente **lee el diario y aplica los cambios que quedaron pendientes**, en lugar de hacer un escaneo completo del disco.

✅ **Menos riesgo de corrupción de datos.**

✅ **Recuperación rápida después de un fallo.**

✅ **Las escrituras secuenciales en el diario son más rápidas** que escribir datos en diferentes partes del disco.

❌ **Puede afectar ligeramente el rendimiento**, porque primero se escriben los cambios en el diario antes de aplicarlos.

El journaling asegura que, incluso si hay un fallo, los metadatos del sistema de archivos puedan ser restaurados a un estado consistente al aplicar las operaciones registradas en el journal. Este mecanismo previene la corrupción de datos y mantiene la integridad del sistema de archivos.

Rendimiento

El rendimiento depende de factores como:

- Tecnología del disco: HDD vs SSD
- Políticas de E/S.
- Tamaños de bloque.
- Uso de cachés.
- Elección entre journaling y soft updates.

NFS (Network File System)

El **NFS** es un protocolo que permite acceder a sistemas de archivos remotos como si fueran locales, utilizando RPC.

- **Funcionamiento**:
  - El sistema remoto se monta en un punto local.
  - La capa **VFS (Virtual File System)** gestiona archivos locales y remotos.
- **Limitaciones**:
  - No es completamente distribuido (los datos de un directorio deben estar en el mismo servidor).
  - Alternativas como **AFS** y **Coda** ofrecen sistemas completamente distribuidos.

Estructura de Ext2

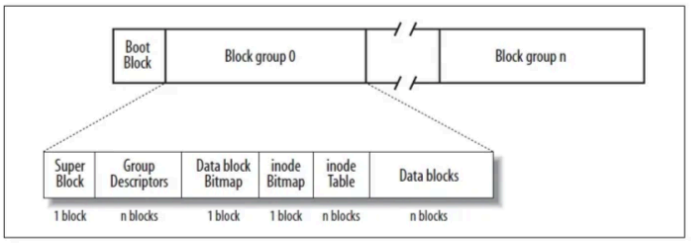
El sistema de archivos **Ext2** se organiza en:

1. **Superbloque**: Contiene metadatos críticos.
2. **Group Descriptor**: Describe grupos de bloques, cantidad de bloques libres, etc.
3. **Inodo**: Almacena metadatos de archivos y directorios.

Creación de archivos en ext2

- Se busca un nuevo inodo dentro del **block group** más adecuado, revisando su **bitmap de inodos** para encontrar un inodo libre. Allí se lo marca como ocupado, y

Estructura de FS Ext2



- El superbloque (superblock) contiene metadatos críticos del sistema de archivos tales como información acerca del tamaño, cantidad de espacio libre y donde se encuentra los datos. Si el superbloque es dañado, y su información se pierde, no podría determinar que partes del sistema de archivos contiene información.

se elige el nuevo inodo de la **tabla de inodos** del block group.

- Luego, se crea el **struct del inodo**, inicializando sus metadatos como permisos, usuario, grupo, timestamps y tamaño, además de asignar los punteros a los bloques de datos (directos, indirectos, dobles indirectos).
- También se **incrementa el contador de enlaces** (i_links_count), ya que el inodo ahora tiene al menos una referencia.
- Como se utilizó espacio libre para crear el archivo, se deben **marcar los bloques como ya no disponibles**. Se buscan bloques de disco libres en el **bitmap de bloques**, se marcan como ocupados o se eliminan de la lista enlazada de bloques libres, dependiendo del método utilizado.
- Se asigna el **nuevo Dir Entry en el directorio** en el que fue creado el archivo, para que mapee al inodo del mismo. Si no entra en el espacio de los bloques de datos que está usando, se debe utilizar un nuevo bloque para guardar el nuevo Dir Entry, repitiendo el proceso anterior de asignación de bloques.
- Finalmente, se actualizan las estructuras del sistema de archivos, como los **contadores de inodos** y **bloques libres en el superbloque** y en el **descriptor de grupo**. También se actualizan los timestamps en el inodo del directorio padre.

Seguridad

Preservar:

- Confidencialidad
- Integridad
- Disponibilidad

Protagonistas

Los sistemas de seguridad suelen tener:

- Sujetos.
- Objetos.
- Acciones.

La idea es decir **qué sujetos** pueden realizar **qué acciones** sobre **qué objetos**.
Importante: los roles de sujeto y objeto no son excluyentes.
Ejemplo: los procesos

El **usuario** es la abstracción más común en los sistemas operativos. Es un sujeto que puede ejecutar acciones y ser dueño de objetos como archivos, procesos, memoria y puertos.

Acciones de un usuario

Un usuario puede:

- Leer, escribir, copiar, abrir, borrar, imprimir.
- Ejecutar programas.
- Matar procesos.

Grupos y roles

- **Grupos:** conjunto de usuarios que comparten permisos.
- **Roles:** asignan permisos según funciones, como operador, usuario común o administrador.

Esta estructura permite gestionar permisos de forma más organizada y eficiente.

Criptografía

Rama de las matemáticas y la informática que permite cifrar información para que solo el destinatario la pueda leer.

El **criptoanálisis** es el análisis de técnicas para descifrar información sin la clave.

- **Cifrado simétrico:** usa la misma clave para cifrar y descifrar (Caesar, DES, AES).
- **Cifrado asimétrico:** usa claves distintas, pública y privada (RSA).

Funciones Hash

Una **función de hash** es un algoritmo que toma un dato de entrada y genera un valor de longitud fija, llamado **hash** o **resumen**. Siempre que se ingrese el mismo dato, se obtendrá el mismo hash.

En criptografía, las funciones de hash deben cumplir ciertas propiedades para ser seguras:

1. **Resistencia a la preimagen**: Dado un hash **h**, debe ser difícil encontrar un dato **m** tal que **hash(m) = h**.
2. **Resistencia a la segunda preimagen**: Dado un dato **m1**, debe ser difícil encontrar otro **m2 ≠ m1** con el mismo hash (**hash(m1) = hash(m2)**).

Uso en almacenamiento de contraseñas

Las funciones de hash son útiles para **almacenar contraseñas de forma segura** porque convierten la contraseña en un valor irreconocible. Sin embargo, como son rápidas, los atacantes pueden usar ataques de fuerza bruta o tablas precalculadas (**rainbow tables**) para intentar adivinar contraseñas.

Para mejorar la seguridad:

- **Usar SALT**: Agregar un valor aleatorio único a cada contraseña antes de aplicar el hash, evitando que dos contraseñas iguales tengan el mismo hash.
- **Iterar varias veces**: Aplicar el hash repetidamente (con algoritmos como PBKDF2, bcrypt o Argon2) para hacer más difícil un ataque.

Estas técnicas dificultan que un atacante recupere contraseñas, incluso si obtiene los hashes.

Autenticación y Contraseñas

Las contraseñas nunca deben almacenarse en texto plano ni siquiera cifradas, sino en forma de hash.

- En Unix, se lamacenan en `/etc/passwd` (`/etc/shadow` hoy en día).

Como alternativa a las contraseñas, se pueden utilizar pares de claves pública y privada. En este caso, el usuario demuestra su identidad sin necesidad de enviar información sensible.

RSA: Cifrado Asimétrico

RSA permite cifrar información sin necesidad de compartir una clave secreta entre emisor y receptor.

Se basa en la **dificultad de factorizar números grandes**.

Cada usuario tiene dos claves:

- **Clave pública:** Se puede compartir libremente.
- **Clave privada:** Debe mantenerse en secreto.

Para enviar un mensaje seguro, se cifra con la clave pública del destinatario. Solo quien tenga la clave privada correspondiente podrá descifrarlo. La seguridad de RSA radica en que, aunque cualquiera pueda ver la clave pública, obtener la privada a partir de ella es prácticamente imposible con la tecnología actual.

Firma Digital con RSA

RSA también se usa para verificar la autenticidad de un documento mediante firmas digitales:

1. Se genera un **hash** del documento.
2. Se cifra con la clave privada del remitente.
3. El receptor usa la clave pública del remitente para descifrarlo y comparar el hash con el del documento recibido.

Si los valores coinciden, significa que el documento es auténtico y no ha sido alterado.
Además de la parte técnica, las firmas digitales tienen respaldo legal, como en la **Ley 25.506**.

Autenticación remota con hash

El **replay-attack** es una técnica donde un atacante intercepta y reutiliza credenciales para acceder sin autorización. Las funciones de hash por sí solas no evitan este problema.

Para mitigar esto, se usan métodos de **Challenge-Response**:

- El servidor genera un número aleatorio y lo envía al cliente.
- El cliente usa ese número como semilla para cifrar su contraseña y envía el resultado.
- El servidor repite el proceso y verifica si ambos valores coinciden.

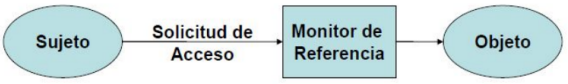
Aunque este método mejora la seguridad, también puede ser atacado con estrategias más avanzadas.

Sobre Permisos

Monitor de referencias

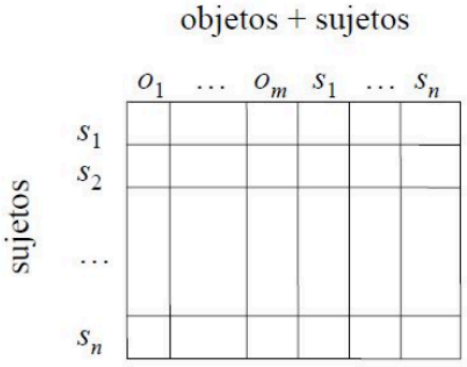
Es el mecanismo que controla los intentos de acceso a objetos según una política definida.

Actúa como **intermediario entre usuarios y recursos**, permitiendo o denegando acciones según reglas establecidas.



Representación de permisos

- Matriz de control de accesos:



- Sujetos $S = \{ s_1, \ldots, s_n \}$
- Objetos $O = \{ o_1, \ldots, o_m \}$
- Permisos $R = \{ r_1, \ldots, r_k \}$
- Entradas $A[s_i, o_j] \subseteq R$

$A[s_i, o_j] = \{ r_x, \ldots, r_y \}$
Es decir el sujeto s_i tiene permisos $r_x, \ldots, r_y$ sobre el objeto o_j

DAC vs. MAC

Existen dos esquemas principales de control de acceso:

- **DAC (Discretionary Access Control)**:
  - Los atributos de seguridad deben definirse explícitamente.
  - El dueño del recurso decide los permisos.
- **MAC (Mandatory Access Control)**:
  - Se usa para información altamente sensible.
  - Cada usuario tiene un nivel de acceso y los archivos creados heredan el nivel del último usuario que los modificó.
  - Un usuario solo puede acceder a archivos de su mismo nivel o inferior.

DAC en UNIX

Unix implementa DAC con un esquema de permisos básicos para archivos y directorios.

- **SETUID y SETGID**:
  - Permiten ejecutar binarios con privilegios elevados temporalmente.
  - Si un archivo tiene el bit **SETUID**, se ve con una "s" en los permisos:

```
- rws  r-x  r-x     1 root shadow 27920 ago 15 22:45 /usr/bin/passwd
| |||  |||  └└└──── Permisos de "otros" (r-x)
| |||  └└└──────── Permisos del "grupo" (r-x)
| └└└────────────── Permisos del "owner" (rws), s de SETUID
└──────────────── Archivo regular (-)

s = setuid: cuando se activa, otros usuarios que no son el owner del archivo, puede
n utilizarlo con los mismos
      privilegios que el (no se heredan los flags de owner, simplemente privilegios)
x = execute
w = write
r = read
```

  - Esto permite que un usuario ejecute el archivo con los privilegios del propietario del archivo. Por ejemplo, para ejecutar binarios que requieren mayores privilegios.

- Ejemplo SETUID en C

El siguiente código muestra cómo SETUID cambia el UID efectivo de un proceso:

```
c
#include <stdio.h>
#include <unistd.h>
int main () {
   int real = getuid();
   int euid = geteuid();
   printf("The REAL UID =: %d\n", real);
   printf("The EFFECTIVE UID =: %d\n", euid);
}
```

Si se ejecuta con permisos elevados, el **EFFECTIVE UID** reflejará el del propietario del archivo:

```
-rwsr-xr-x 1 root root 8392 oct 12 00:07 setuid

output al ejecutarlo con usuario no privilegiado:
```

```
The REAL UID =: 1000
The EFFECTIVE UID =: 0
```

SUDO

- Permite ejecutar comandos como otro usuario de manera controlada.
- **Ventajas**:
  - Registra los comandos ejecutados, mejorando la auditabilidad.
  - Se pueden revocar permisos de forma centralizada.

Puntos sensibles en los permisos

Hay varios aspectos clave en la administración de permisos:

- **Propagación de permisos:** ¿Quién puede otorgar permisos a otros?
- **Revocación:** Puede ser inmediata o diferida, dependiendo de cómo esté implementado el sistema.
- **Seguridad en archivos:** Un usuario que puede modificar un archivo no debería necesariamente poder cambiar sus permisos.

Procesos y permisos

Los procesos heredan los permisos del usuario que los ejecuta. Sin embargo, hay situaciones donde necesitan permisos distintos, como en el cambio de contraseña.

Para esto, en Unix se usa el **bit setuid**, permitiendo que un programa se ejecute con los permisos de su propietario en lugar de los del usuario que lo ejecuta.

> ⚠️ Esto puede ser peligroso si no se maneja bien, ya que un atacante podría explotar el sistema para obtener privilegios elevados.

Windows Integrity Control

Este sistema implementa **MAC.**

Define cuatro niveles de integridad para objetos y usuarios:

- **System:** Máxima prioridad, usado por procesos críticos.
- **High:** Para administradores o procesos privilegiados.
- **Medium:** Nivel por defecto para usuarios estándar.
- **Low:** Para procesos menos confiables, como navegadores web.

Un usuario no puede asignar un nivel de integridad mayor al suyo, lo que evita escaladas de privilegios no autorizadas.

Seguridad en el software

La seguridad debe considerarse en todas las fases del desarrollo:

- **Arquitectura/Diseño:** Durante la planificación de la aplicación.
- **Implementación:** Al escribir el código.
- **Operación:** Cuando la aplicación está en producción.

Errores de implementación

Son más fáciles de detectar y corregir que los errores de diseño. Un problema común es hacer suposiciones incorrectas sobre el entorno, como **asumir que el usuario ingresará datos válidos y dentro de un límite**.

Fuentes de entrada insegura:

- Variables de entorno (ejemplo: `PATH`).
- Entradas del programa (locales o en red).

Exploits de vulnerabilidades

1. Buffer overflows

Cuando una función en C usa más memoria de la reservada, puede sobrescribir datos críticos, permitiendo ejecución de código malicioso.

Ejemplo clásico en **stack-based buffer overflow**:

```
void f(char *origen) {
    char buffer[16];
    strcpy(buffer, origen); // No hay validación del tamaño de origen
}
```

Si `origen` contiene más de 16 caracteres, sobrescribe otras partes de la pila, incluyendo la dirección de retorno.

Otro ejemplo:

```
struct User {
    char name[16];   // Offset 0
    char pass[16];   // Offset 16
};

void login() {
    struct User user;
    char realpass[16] = "secret123";  // Offset 32

    printf("Ingrese su nombre de usuario: ");
```

```
    gets(user.name); // VULNERABILIDAD: No verifica el tamaño

    printf("Ingrese su contraseña: ");
    gets(user.pass); // VULNERABILIDAD: No verifica el tamaño

    if (strcmp(user.pass, realpass) == 0) {
        printf("Acceso concedido\n");
    } else {
        printf("Acceso denegado\n");
    }
}
```

```
0x0000 - las vars locales crecen hacia direcciones MÁS ALTAS del stack (en sentido contrari
El STACK crece hacia direcciones más bajas de memoria.
 _______________ -
|_______________|
|____user.name____| [0,...,7]
|____user.name____| [8,...,15]
|____user.pass____| [0,...,7]
|____user.pass____| [8,...,15]
|____realpass_____| [0,...,7]
|____realpass_____| [8,...,15]
|_______EBP_______|
|_EIP (ret addr)__|

0xFFFF          +
```

Variaciones:

- **Stack-based:** Ocurre en la pila de ejecución.
- **Heap-based:** Similar, pero en memoria dinámica.

Prevención

- Chequear largo de entradas u otras restricciones
- Uso de funciones seguras:
  - `strncpy(buffer, src, size)` en vez de `strcpy(buffer, src)`
  - `fgets(buffer, sizeof(buffer), stdin)` en vez de `gets(buffer)`
- **Stack Canaries:** Se coloca un valor en el stack después de crear el stack frame. Antes de retornar de la función, se valida que el valor sea el correcto, **protegiendo el valor de retorno de la función** de posibles buffer overflows.
- Compiladores con protecciones:
  - **DEP** (Data Execution Prevention): Ninguna región de memoria debería ser al mismo tiempo WRITEABLE y EXECUTABLE. Ejemplo: heap y stack

- **ASLR** (Address Space Layout Randomization): Modifica de manera aleatoria la dirección base de regiones importantes de memoria entre las diferentes ejecuciones de un proceso. Ejemplo: heap, stack, LibC, etc...

2. Control de parámetros

Problema de **Format String**. Ejemplo de una aplicación que envía correos:

```
$direccion=argv[1]
echo "Feliz cumple" | mail -subject='Felicitación' $direccion
```

Si el usuario modifica la URL:

```
http://www.example.com/cgi-bin/felicitar.cgi?fulano@dc.uba.ar; rm -rf /
```

El código ejecuta:

```
echo "Feliz cumple" | mail -subject='Felicitación' fulano@dc.uba.ar; rm -rf /
```

lo que permite ejecución de comandos arbitrarios.

Prevención

- Aplicar **mínimo privilegio** al programa en cuestión
- **Validar parámetros** antes de usarlos.
  - Usar el concepto de **"tainted data"**, marcando y limpiando datos de entrada.

3. SQL Injection

Ocurre cuando un sistema no sanitiza las entradas antes de ejecutar una consulta SQL.

Ejemplo de código vulnerable:

```
db.execute("UPDATE alumnos SET aprobado=true WHERE lu='" + input.getString("lu") + "'")
```

Si un usuario malicioso ingresa:

```
307/08'; DROP alumnos; SELECT '
```

La consulta resultante elimina toda la tabla.

Prevención

- Usar **consultas preparadas** (`Prepared Statements`).
- Escapar correctamente los valores de entrada.

4. Condiciones de carrera

Ocurren cuando el resultado de un programa depende del **orden o timing** de ejecución de múltiples procesos o hilos.

Ejemplo típico: un programa verifica si un archivo existe y luego lo crea si no está.

El problema es que, entre la verificación y la creación, otro proceso podría haberlo creado o modificado. Esto puede explotarse para **sobreescribir archivos críticos** mediante enlaces simbólicos.

Prevención

La solución es hacer la operación atómica, es decir, que no pueda ser interrumpida por otro proceso.

Ejemplo en Unix:

```
fd = open("archivo", O_CREAT | O_EXCL, 0600);
```

La opción `O_EXCL` asegura que el archivo solo se cree si **no existía previamente**, evitando ataques de enlaces simbólicos.

5. Variables de entorno

Cuando un programa ejecuta un binario sin especificar su **ruta absoluta**, el sistema busca ese binario en los directorios listados en la variable de entorno **$PATH**.

Esto puede ser un problema de seguridad si un atacante logra modificar el **$PATH** e insertar un binario malicioso antes que el binario legítimo.

Escenario vulnerable

Un script de administración ejecuta `echo` sin especificar su ruta absoluta:

```
#!/bin/bash
mi_usuario=$(whoami)
echo "Usuario actual: $mi_usuario"
```

Si este script se ejecuta con `sudo`, un atacante podría interceptar la llamada a `echo` y ejecutar código malicioso.

El atacante crea un **falso** `echo` y modifica `$PATH`:

```
echo '#!/bin/bash' > echo
echo 'echo "¡Privilegios escalados!"' >> echo
chmod +x echo

export PATH=".:$PATH"  # Agrega la carpeta actual al inicio del PATH
sudo ./script.sh  # Si el script ejecuta "echo", se ejecuta el falso echo
```

Prevención

- Usar rutas absolutas (`/bin/echo` en vez de `echo`).

- Resetear `$PATH` en scripts con privilegios (`unset PATH; export PATH="/usr/bin:/bin"`).
- No ejecutar scripts desconocidos con `sudo`.

Código malicioso

Malware es cualquier software diseñado para ejecutar acciones no autorizadas en un sistema. Puede tomar diferentes formas: virus, troyanos, gusanos, ransomware, rootkits.

Métodos de infección

- Descargas de internet (incluso sin intervención del usuario).
- Archivos adjuntos en correos electrónicos.
- Exploit de vulnerabilidades en programas.
- Dispositivos USB infectados.
- Ingeniería social y phishing.

Prevención de infecciones

Tener **antivirus actualizado** ayuda, pero no es suficiente.

Se recomienda:

✅ Mantener actualizado el sistema operativo y software.

✅ No ejecutar archivos de origen desconocido.

✅ Evitar privilegios de administrador innecesarios.

✅ No hacer clic en enlaces sospechosos o adjuntos no verificados.

Aislamiento de usuarios y procesos

Muchos sistemas operativos modernos incluyen **mecanismos de aislamiento** para reducir el impacto de ataques:

- `chroot()`: restringe el acceso de un proceso a un directorio específico.
- `jail()`: similar a `chroot`, pero con más restricciones.
- **Virtualización**: crear entornos completamente separados para ejecutar aplicaciones de riesgo.

Principios básicos de seguridad

Para diseñar sistemas más seguros, es fundamental seguir ciertas reglas:

- **Mínimo privilegio**: los procesos deben ejecutarse con los permisos más bajos posibles.
- **Simplicidad**: sistemas complejos son más difíciles de proteger.
- **Validar entradas y accesos**: nunca asumir que los datos recibidos son confiables.
- **Separación de privilegios**: dividir funciones críticas entre diferentes procesos o usuarios.
- **Múltiples capas de seguridad**: no depender de un solo mecanismo de protección.

- **Usabilidad**: las medidas de seguridad deben ser fáciles de usar para evitar que los usuarios las ignoren.

La confianza en la seguridad

Un sistema seguro no solo depende de la tecnología, sino también de la confianza en los mecanismos implementados.

Ejemplo:

- Se descubre una vulnerabilidad en un software.
- El fabricante libera un parche de seguridad.
- Los usuarios confían en que el parche **fue probado correctamente** y que realmente soluciona el problema.

El problema es que **si alguna de estas suposiciones es incorrecta, la seguridad se rompe**.

Sistemas Distribuidos

Un conjunto de recursos conectados que interactúan.

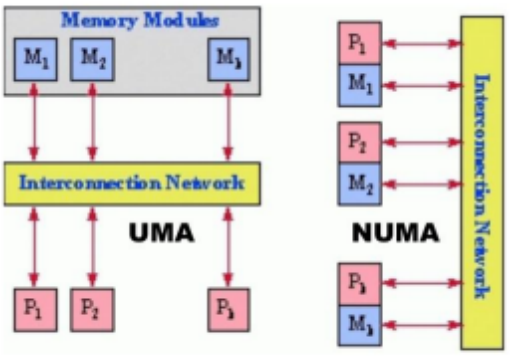
Ejemplos:

- Varias máquinas en red.
- Un procesador con múltiples memorias.
- Varios procesadores compartiendo memoria.

Fortalezas	Debilidades
✔️ Paralelismo. ✔️ Replicación. ✔️ Descentralización.	⚠️ Difícil sincronización. ⚠️ Mantener coherencia es complejo. ⚠️ No suelen compartir reloj. ⚠️ Información parcial.

Memoria Compartida en Sistemas Distribuidos

- **Hardware:**
  - **UMA (Uniform Memory Access):** Todos los procesadores acceden a la memoria con la misma latencia.
  - **NUMA (Non-Uniform Memory Access):** La latencia varía según la proximidad del procesador a la memoria.
  - **Híbrida:** Combinación de ambas arquitecturas.



- **Software:**
  - **Estructurada:**
    - Memoria asociativa (*Linda*, *JavaSpaces*).
    - *Distributed Arrays* en lenguajes como Fortran, X10, Chapel.
  - **No estructurada:**
    - Memoria virtual global.
    - Memoria virtual particionada por localidad.

Comunicación en sin memoria compartida

Arquitectura Cliente/Servidor

La arquitectura **cliente/servidor** es un modelo en el que un cliente solicita servicios a un servidor, el cual responde a las peticiones.

- El cliente **inicia la comunicación** y solicita recursos o información.
- El servidor **no actúa de forma autónoma**, solo responde cuando recibe una solicitud del cliente.

Conexión Remota (Telnet)

- **Telnet** es un protocolo y una herramienta que permite a un usuario conectarse a otro equipo de forma remota.
- El usuario accede a la terminal del equipo remoto como si estuviera físicamente presente en él.
- En este modelo, **solo uno de los equipos realiza el trabajo real** (el equipo remoto), mientras que el otro simplemente **actúa como una interfaz** de comunicación para enviar comandos y recibir respuestas.

Llamadas a Procedimiento Remoto (RPC)

- **RPC (Remote Procedure Call)** permite a un programa ejecutar funciones en otra máquina como si fueran locales.
- Se basa en la idea de que una aplicación distribuida puede dividirse en partes que se ejecutan en diferentes computadoras, comunicándose entre sí mediante llamadas a procedimientos remotos.
- **Es un mecanismo síncrono**, lo que significa que el cliente espera la respuesta del servidor antes de continuar su ejecución.

Mecanismos Asíncronos

Los mecanismos asíncronos permiten que los procesos continúen ejecutándose sin esperar respuestas inmediatas, mejorando la eficiencia en sistemas distribuidos.

- **RPC Asíncrono:** Variante de RPC que no bloquea al cliente mientras espera la respuesta del servidor.
- **Windows Asynchronous RPC:** Implementación de RPC asíncrono en sistemas Windows.
- **Pasaje de Mensajes (send/receive):** Es un método de comunicación en sistemas distribuidos en el que los procesos intercambian mensajes.
- Ej. Mailbox, Pipe, MPI (Message Passing Interface) para C/C++

Pasaje de Mensajes

El pasaje de mensajes permite la comunicación entre procesos sin necesidad de memoria compartida, utilizando un canal de comunicación.

Problemas del Pasaje de Mensajes

A pesar de sus ventajas, este mecanismo presenta ciertos desafíos:

- **Codificación y decodificación de datos:** Es necesario transformar los datos en un formato entendible para ambos extremos.
- **Interrupciones en el procesamiento:** Los procesos pueden verse obligados a detenerse temporalmente para gestionar el intercambio de mensajes.
- **Latencia:** La comunicación a través de la red introduce retardos en la ejecución.
- **Posible pérdida de mensajes:** Aunque TCP/IP proporciona confiabilidad, algunos protocolos pueden perder paquetes de datos.

Para abordar estos problemas, existen bibliotecas especializadas como **MPI**, que optimizan el pasaje de mensajes en entornos de alto rendimiento.

Teorema CAP (Brewer's Conjecture)

Este teorema establece que, en un sistema distribuido, **no es posible garantizar simultáneamente** las siguientes tres propiedades, sino hasta dos de ellas:

1. **Consistencia (Consistency):** Todos los nodos ven la misma información en el mismo momento.
2. **Disponibilidad (Availability):** El sistema sigue funcionando y respondiendo a solicitudes incluso en presencia de fallos.
3. **Tolerancia a particiones (Partition Tolerance):** El sistema sigue operando incluso si la red se divide en partes incomunicadas.

Sincronización en Sistemas Distribuidos

Problema de los Locks

En entornos distribuidos no existe una operación **TestAndSet atómica**, lo que dificulta la implementación de locks.

Se requieren estrategias alternativas para **gestionar la concurrencia sin compartir memoria** de manera directa.

Enfoque Centralizado

Una solución básica consiste en designar un **nodo coordinador** que **controle el acceso a los recursos**.

- Los procesos remotos cuentan con proxies dentro de este nodo, que negocian entre sí el acceso a los recursos.

Este enfoque presenta varias limitaciones:

- Al depender de un único nodo, se introduce un punto de falla crítico.
- Genera un cuello de botella tanto en el procesamiento como en la capacidad de red. Cada interacción con el coordinador requiere comunicación a través de la red, lo que introduce latencias adicionales.

Alternativas

- Se busca determinar **quién tiene prioridad** en un entorno distribuido.
- ¿Quién pidió primero el lock? → Depende del orden de eventos.

Razonamiento Distribuido

La **sincronización precisa de relojes en un sistema distribuido es costosa y difícil de mantener**.

- Alternativa: **ordenar eventos sin depender del tiempo exacto.**

Orden Parcial de Eventos (Lamport)

Lamport propuso un esquema para establecer un orden parcial entre eventos sin necesidad de sincronizar relojes de forma precisa. Se definen las siguientes reglas:

- Si un evento A ocurre antes que B dentro de un mismo proceso, se establece A → B.

- Si un evento E es el envío de un mensaje y R su recepción, entonces E → R.
- Si A → B y B → C, entonces A → C.
- Si no se puede establecer A → B ni B → A, los eventos se consideran concurrentes.

Implementación del Orden Parcial

- **Cada procesador tiene un reloj** (monótonamente creciente).
- **Los mensajes llevan una marca de tiempo**.
- **Actualización del reloj**:
  - Si se recibe un mensaje con una marca mayor, se ajusta a **t + 1**.
- **Orden Total**: Para resolver empates, se pueden ordenar los eventos de forma arbitraria (ej. por **PID**).

Modelo de Fallas

En sistemas distribuidos, es fundamental definir cómo pueden fallar los procesos o la red. Algunos modelos comunes incluyen:

- Sistemas sin fallas.
- Procesos que caen y no se recuperan.
- Procesos que caen y se recuperan.
- Particiones de red.
- Fallas bizantinas (comportamiento impredecible).

En este curso, el enfoque principal está en sistemas sin fallas, aunque es importante conocer estos modelos para entender cómo los algoritmos se adaptan a diferentes escenarios.

Métricas de Complejidad

Para evaluar algoritmos distribuidos, se utilizan métricas como:

- **Cantidad de mensajes**: clave en sistemas distribuidos, aunque en redes de alta velocidad puede no ser el único cuello de botella.
- **Tolerancia a fallas**: qué tipos de fallas soporta el algoritmo.
- **Información necesaria**: tamaño de la red, cantidad de procesos, ubicación, etc.

Estas métricas ayudan a comparar y elegir algoritmos según el contexto.

Problemas Principales

Los problemas más comunes en sistemas distribuidos se dividen en tres grandes familias:

1. **Orden de ocurrencia de eventos**.
2. **Exclusión mutua**.
3. **Consenso**.

Cada uno de estos problemas tiene sus propios algoritmos y desafíos, que se exploraron en detalle durante la teórica.

Exclusión Mutua Distribuida

La exclusión mutua en sistemas distribuidos busca garantizar que solo un proceso acceda a una sección crítica a la vez. Dos enfoques principales son:

1. **Token Passing**:
   - Un token circula en un anillo lógico entre los procesos.
   - Ventaja: sin inanición (si no hay fallas).
   - Desventaja: mensajes innecesarios.
   - Ejemplos: FDDI, TDMA, TTA.
2. **Algoritmo basado en solicitudes**:
   - Cada proceso envía un mensaje con un timestamp a todos los demás.
   - Se entra a la sección crítica cuando se reciben todas las respuestas.
   - No se envían mensajes si no se necesita entrar a la sección crítica.

Locks Distribuidos

En lugar de usar un coordinador centralizado, se puede implementar un **protocolo de mayoría** para locks distribuidos. La idea es que, para obtener un lock sobre un objeto replicado en \(n \) sitios, se debe obtener la mayoría (\(n/2 + 1 \)) de los locks. Cada copia del objeto tiene un número de versión, lo que evita lecturas desactualizadas. Sin embargo, este enfoque puede generar deadlocks, por lo que se necesitan algoritmos de detección.

Elección de Líder

En sistemas distribuidos, a veces es necesario elegir un líder para coordinar tareas. Un algoritmo clásico es el de **Le Lann, Chang y Roberts**, donde los procesos se organizan en un anillo y circulan sus IDs. El proceso con el ID más alto es elegido líder. Este algoritmo tiene una complejidad de tiempo de \(O(n) \) sin fase de stop y \(O(2n) \) con fase de stop, y una complejidad de comunicación de \(O(n^2) \).

Instantánea Global Consistente

Una instantánea global consistente permite obtener una "foto" del estado de un sistema distribuido en un momento dado. Para lograrlo:

1. Un proceso envía un mensaje de marca a todos los demás.
2. Cada proceso guarda su estado local y registra los mensajes en tránsito.
3. Cuando se reciben todas las marcas, se conforma el estado global.

Esta técnica es útil para la detección de propiedades estables, debugging distribuido y detección de deadlocks.

Algoritmo Two-Phase Commit (2PC)

Es el protocolo más simple y ampliamente utilizado para garantizar consistencia en transacciones distribuidas. Se compone de **dos fases**:

Fase 1: PREPARACIÓN (Votación)

1. **El coordinador envía un mensaje** `PREPARE` **a todos los participantes**, preguntando si están listos para confirmar la transacción.
2. **Cada participante responde con** `YES` **(listo para commit) o** `NO` **(abort)**. Si responde `YES`, guarda un estado provisional en su log de transacciones para asegurar que puede completar la transacción más tarde si es necesario.

Fase 2: COMMIT o ABORT

1. **Si todos los participantes responden** `YES`, el coordinador envía un mensaje `COMMIT`, y todos los nodos ejecutan la transacción y la confirman en su base de datos.
2. **Si algún participante responde** `NO` o no responde, el coordinador envía `ABORT`, y todos los nodos deshacen cualquier cambio provisional.

📌 **Problema de 2PC:**

- Es **bloqueante**: si el coordinador falla después de la fase 1, los participantes pueden quedar indefinidamente en espera, sin saber si deben hacer commit o abort.

---

2. Algoritmo Three-Phase Commit (3PC)

3PC extiende 2PC agregando una fase intermedia para reducir bloqueos y mejorar la tolerancia a fallos. Consiste en **tres fases**:

Fase 1: PREPARE (Votación)

1. **El coordinador envía** `PREPARE` **a todos los participantes**.
2. **Los participantes responden con** `YES` **o** `NO`, igual que en 2PC.

Fase 2: PRE-COMMIT (Confirmación tentativa)

1. **Si todos responden** `YES`, el coordinador envía un mensaje `PRE-COMMIT`, indicando que deben prepararse para la confirmación final.
2. **Cada participante responde** `ACK` **cuando está listo** y guarda la transacción en un estado temporal.

Fase 3: COMMIT o ABORT

1. **Si el coordinador recibe todos los** `ACKs`, envía `COMMIT`, y los participantes hacen commit definitivo.
2. **Si alguno falla o responde negativamente, se envía** `ABORT` **y todos los participantes deshacen la transacción.**

📌 **Ventajas de 3PC sobre 2PC:**

✅ **Menos bloqueante**: si el coordinador falla en la fase 2, los participantes pueden decidir por sí mismos si abortar o continuar.

✅ **Mayor tolerancia a fallos**: los participantes no quedan atrapados indefinidamente esperando una decisión.

📌 **Problema de 3PC:**

- **No garantiza consistencia en presencia de fallos de red**: Si los mensajes de commit o abort se pierden, algunos nodos pueden hacer commit mientras otros abortan, generando inconsistencia.

Consenso y Aplicaciones

El consenso en sistemas distribuidos puede tomar varias formas, como:

- **k-agreement**: se eligen k valores.
- **Acuerdos aproximados**: los valores deciden dentro de un rango ϵ.
- **Acuerdos probabilísticos**: baja probabilidad de desacuerdo.

Estas técnicas se aplican en la sincronización de relojes (NTP) y en sistemas críticos con tolerancia a fallas.

Algoritmos y Tecnologías Adicionales

Además de los algoritmos clásicos, existen otros como:

- **Paxos**: algoritmo de consenso con tolerancia a fallas.
- **RAFT**: alternativa a Paxos, más simple.
- **Blockchain**: registro distribuido y descentralizado.

Estas tecnologías son fundamentales en sistemas distribuidos modernos.

Virtualización

La virtualización permite que **un conjunto de recursos físicos se comporten como múltiples recursos lógicos**, permitiendo que una computadora realice el trabajo de varias.

- **Historia**: La idea de máquinas virtuales existe desde la década de 1960, con objetivos como la portabilidad, simulación, aislamiento, aprovechamiento de hardware, y protección ante fallas.

Simulación y Emulación

- **Simulación**: Se crea un estado artificial en el sistema anfitrión para representar al sistema huésped, pero puede ser **lento y complicado para manejar interrupciones y concurrencia**.
- **Emulación**: El sistema emulado se ejecuta en la CPU del anfitrión, emulando componentes de hardware, pero presenta **problemas de separación de privilegios y velocidad de acceso a dispositivos**.

Virtualización Asistida por Hardware

Técnica que utiliza extensiones del procesador para facilitar la creación y gestión de VMs, resolviendo problemas que surgían con la virtualización basada solo en software.

Problemas que resuelve:

1. **Ring Aliasing**: Programas que deberían correr en modo kernel (privilegiado) se ejecutan en modo usuario, causando problemas de permisos.
2. **Address-Space Compression**: La VM y el hipervisor comparten memoria, lo que puede llevar a conflictos.
3. **Non-Faulting Access**: Algunas instrucciones privilegiadas no generan excepciones, dificultando su interceptación.
4. **Interrupt Virtualization**: Simular interrupciones para la VM es complicado sin soporte de hardware.

Soluciones con hardware:

Intel introdujo las extensiones **VT-x**, que añaden dos modos de ejecución:

- **VMX root**: Para el hipervisor (anfitrión).
- **VMX non-root**: Para la máquina virtual (huésped).

Además, se usa la **Virtual Machine Control Structure (VMCS)**, una estructura en memoria que almacena el estado del huésped y el anfitrión. Cuando la VM intenta ejecutar una acción "prohibida", el hardware sale automáticamente del modo non-root, y el hipervisor toma el control para manejar la situación.

En resumen, la virtualización asistida por hardware mejora el rendimiento y la seguridad al permitir que el hipervisor y las VMs operen de manera más eficiente, evitando los cuellos de botella y problemas de permisos que surgían con la virtualización basada solo en software.

Contenedores

Los contenedores permiten ejecutar aplicaciones en un entorno aislado, compartiendo el kernel del sistema operativo anfitrión.

- **Tecnologías del Kernel**:
  - **Cgroups**: Limitan el uso de recursos como memoria, CPU, I/O y red.
  - **Namespaces**: Proporcionan a los procesos una visión aislada del sistema.
  - **Netfilter, Netlink, Capabilities**: Filtrado de paquetes, comunicación kernel-usuario, y asignación de privilegios.
  - **SELinux y AppArmor**: Módulos de seguridad para control de acceso obligatorio.
- **Orquestación**: Herramientas como Kubernetes y OpenShift facilitan la automatización de la implementación, escalado y administración de aplicaciones en contenedores.