Tema: Introducción a los sistemas operativos

¿Qué es un sistema operativo?

- Es el **software intermediario** entre el hardware y las aplicaciones.
- Los **programas de usuario** acceden a estos recursos a través de **llamadas al sistema (system calls)**.
- Permite a los programas acceder al hardware de forma **segura y controlada**.
- Se encarga de **gestionar recursos**: CPU, memoria, dispositivos de entrada/salida, almacenamiento.

Objetivos:

1. **Abstracción - Virtualizacion**:
    - Presenta una **visión simplificada y uniforme** del hardware (por ejemplo, el sistema de archivos abstrae los discos).
    - Permite que los programas no tengan que preocuparse por los detalles de bajo nivel del hardware.
    - Permite que los programas se ejecuten **independientemente de la máquina física**.
2. **Multiplexación**:
    - Permite **compartir recursos** entre múltiples procesos o usuarios (ejemplo: tiempo de CPU, memoria, disco).
    - Puede hacerse en el **tiempo** (ej. turnos de CPU) o en el **espacio** (ej. dividir memoria).
3. **Aislamiento y protección**:
    - Impide que los procesos interfieran entre sí o con el SO.
    - Fundamental para la **seguridad y estabilidad** del sistema.
    - Decide **qué programa usa qué recursos y cuándo** (CPU, RAM, disco, etc.).
4. **Eficiencia y rendimiento**:
    - El sistema operativo debe ser **rápido y liviano**, usando los recursos con eficacia.
    - Controla el acceso para asegurar un **uso eficiente y justo**.

Partes del sistema operativo

- **Kernel**: el "núcleo" del SO, que ejecuta con privilegios elevados.
- **Librerías del sistema**: funciones de alto nivel usadas por aplicaciones (ej. `malloc`, `printf`).
- **Servicios del sistema**: como el programador de tareas, el manejador de memoria, etc.

Transformación de programas fuente en ejecutables

- Se explica el proceso:
  - **Código fuente** (C) → **Preprocesador**
  - → **Compilador** → **Ensamblador** → **Enlazador (Linker)** → **Ejecutable**
- Cada etapa transforma el programa a un nivel más cercano al hardware.

La CPU ejecuta instrucciones secuencialmente

- Utiliza el **ciclo de instrucción**: **fetch** → **decode** → **execute**.
- Opera sobre datos almacenados en registros o memoria.

Jerarquía de memoria

- La memoria no es plana: hay **niveles con diferentes velocidades y tamaños** (registro, caché, RAM, disco).
- El sistema debe mover datos entre esos niveles eficientemente.

Sistemas multitarea y concurrencia

- Varios programas pueden parecer estar corriendo a la vez mediante **multiprogramación**.
- Aparece el concepto de **context switching**, donde el SO cambia qué proceso está en ejecución.

Redes y sistemas distribuidos

- El sistema maneja **protocolo TCP/IP**, **sockets**, etc., para comunicarte con otros sistemas.

Modo usuario vs. modo kernel

- El sistema corre en dos modos:
  - **Modo usuario**: donde se ejecutan los programas normales.
  - **Modo kernel**: donde se ejecuta el núcleo del SO, con acceso total al hardware.
- Las **System Calls** hacen que el programa **pase de modo usuario a modo kernel** temporalmente.

System Calls

- Son la interfaz entre el programa y el SO (ej. `read()`, `write()`, `fork()`, `exec()`).

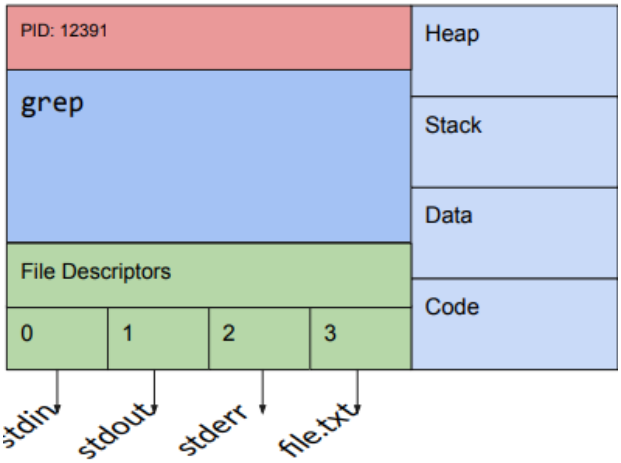
Archivos y descriptores de archivo

- En UNIX, **todo es un archivo**: archivos reales, dispositivos, sockets, pipes.
- Cada archivo abierto se gestiona con un **file descriptor** (un número entero que lo representa).

- Los descriptores estándar son:
  - 0: entrada estándar (stdin)
  - 1: salida estándar (stdout)
  - 2: salida de error estándar (stderr)

Procesos

- Un **proceso** es una instancia en ejecución de un programa.
- Cada proceso tiene:
  - Su **propio espacio de direcciones**
  - Su propia **memoria**
  - Sus propios **archivos abiertos**
  - Su propio **PID**



Fork y Exec

- `fork()` crea un **nuevo proceso hijo** copiando al padre.
- `exec()` reemplaza el contenido de un proceso con un nuevo programa.

Tema: Kernel: userland vs kernelspace

¿Qué es el kernel?

- El **kernel** es el núcleo del sistema operativo: el software que **corre con privilegios totales** y **controla directamente el hardware**.
- A través del kernel, el SO provee **servicios clave** a los programas de usuario.

Abstracción del hardware

- El kernel **transforma hardware complejo y diverso** en **interfaces simples y consistentes**.
- Por ejemplo:

    - En vez de interactuar con sectores del disco → usamos **archivos**.
    - En vez de interactuar con registros de la CPU → usamos **procesos e hilos**.
    - En vez de manejar interrupciones manualmente → usamos **llamadas al sistema**.

Aislamiento

- El kernel **protege a los programas entre sí**:
    - Un programa no puede corromper la memoria de otro.
    - Los recursos se asignan de forma controlada y segura.
- Se implementa mediante:
    - **Modos de ejecución** (usuario/kernel)
    - **Memoria virtual**
    - **Control de acceso a recursos**

Multiplexación

- Permite que múltiples programas **usen el mismo recurso** sin conflictos.
- Ejemplo: varios procesos pueden compartir la CPU → el kernel se encarga de alternarlos (scheduler).
- El kernel **decide quién accede a qué y cuándo**.

Comunicación controlada

- El kernel permite que los procesos **se comuniquen de forma segura**, por ejemplo:
    - Pipes
    - Sockets
    - Memoria compartida

Herramientas del kernel

- El kernel provee herramientas como:
    - **System Calls**
    - **Manejadores de interrupciones**
    - **Scheduler**
    - **Manejo de memoria**

Componentes principales del kernel

- **Procesos**: ejecución de programas con estado propio (registro, memoria, archivos).
- **Scheduler**: elige qué proceso corre.
- **Memoria virtual**: cada proceso tiene su propio espacio de direcciones.
- **File System**: organiza y accede a archivos.
- **Dispositivos**: entrada/salida (I/O), acceso mediante drivers.
- **Traps e interrupciones**: permiten responder a eventos internos y externos.

Beneficios de la abstracción del kernel

- **Simplifica el desarrollo de software**
- **Aumenta la portabilidad** (el mismo programa puede correr en distintos hardwares)
- **Asegura estabilidad y seguridad** del sistema

Memoria virtual

- Cada proceso cree que tiene acceso a toda la memoria.
- El kernel usa la **memoria virtual** para mapear direcciones virtuales a físicas.
- Protege a los procesos entre sí y al kernel.

Qué es una System Call

- Es la **forma principal en que un programa en modo usuario solicita servicios al kernel**.
- Ejemplos de servicios: leer o escribir archivos, crear procesos, asignar memoria, etc.
- Permiten el acceso controlado a recursos que **solo el kernel puede gestionar directamente**.

Transición de modo usuario a modo kernel

- Una llamada al sistema implica pasar de **modo usuario a modo kernel**.
- Esto se hace mediante una **instrucción especial de la CPU** (como `int 0x80`, `syscall`, o `sysenter`).
- El kernel ejecuta la operación y **retorna el control al programa**.

Manejo de errores

- Las llamadas al sistema **retornan -1 en caso de error**.
- El detalle del error se guarda en la variable global `errno`.
- Puedes imprimir el error con `perror()` o `strerror(errno)`.

Lista de llamadas comunes

Algunas de las llamadas al sistema más comunes que se mencionan:

- `read()` – Leer desde un descriptor de archivo
- `write()` – Escribir a un descriptor de archivo
- `open()` / `close()` – Abrir o cerrar un archivo
- `fork()` – Crear un nuevo proceso
- `exec()` – Ejecutar un nuevo programa
- `wait()` – Esperar que termine un proceso hijo
- `exit()` – Terminar el proceso actual

xv6

- **xv6** es una reimplementación de **UNIX Version 6**, escrita en **C y ensamblador** para la arquitectura **x86**.
- Se usa como **sistema de enseñanza**: es más simple que Linux, pero funcionalmente completo.

Estructura modular del SO

- xv6 separa las funcionalidades del kernel en archivos fuente bien definidos:
  - `proc.c`: manejo de procesos.
  - `fs.c`: sistema de archivos.
  - `trap.c`: manejo de traps e interrupciones.
  - `exec.c`: ejecución de nuevos programas.
  - `syscall.c`: implementación de llamadas al sistema.

Ciclo de vida de un proceso

- Un programa es cargado y ejecutado como un **proceso**.
- El proceso:
  1. Se crea (`fork`)
  2. Ejecuta un programa (`exec`)
  3. Puede esperar hijos (`wait`)
  4. Termina (`exit`)
- El kernel guarda el estado de cada proceso y **cambia entre ellos** usando interrupciones de reloj.

Llamadas al sistema en xv6

- Implementa una API UNIX-like: `read()`, `write()`, `fork()`, `exec()`, etc.
- Las llamadas al sistema se **despachan** desde una tabla (`syscall.c`) que asocia números con funciones del kernel.

Tema: La abstracción de proceso

¿Qué es un proceso?

- Es la **instancia en ejecución de un programa**.
- El sistema operativo hace que parezca que el proceso tiene:
  - Su propio **CPU** (aunque se comparta entre muchos).
  - Su propio **espacio de memoria** (aunque físicamente no sea así).

Transiciones de estado del proceso

El sistema operativo maneja procesos en distintos **estados**:

- **New**: recién creado.
- **Ready**: listo para ejecutar.
- **Running**: actualmente ejecutando.
- **Waiting (blocked)**: esperando I/O u otro evento.
- **Terminated**: finalizado.

Cambio de contexto (context switch)

- Cuando el kernel cambia de un proceso a otro:
  - Guarda el estado del proceso actual.
  - Restaura el del siguiente.
  - Esto **permite multitarea**, pero tiene un costo (tiempo).

Ilusión poderosa

- Aunque solo hay un CPU, el sistema operativo usa planificación y cambio de contexto para crear la **ilusión de múltiples CPUs**.
- Esta abstracción permite escribir programas como si **fueran los únicos en ejecución**.

API del proceso

- Es el conjunto de **funciones y llamadas al sistema** que permiten a los programas de usuario interactuar con el sistema operativo para gestionar procesos.
- A través de estas funciones, los programas pueden **crear**, **terminar**, y **comunicar** entre procesos.

Crear un proceso: `fork()`

- La **llamada al sistema `fork()`** crea un nuevo proceso.
- El nuevo proceso es una **copia del proceso padre**, pero con un **identificador de proceso (PID)** diferente.
- Después de `fork()`, el proceso padre y el hijo continúan su ejecución de manera **independiente**.
- El valor de retorno de `fork()` es **0 en el hijo** y **el PID del hijo en el padre**.
- El hijo **hereda** los **File Descriptors** del proceso padre.

```
int pid = fork();
if(pid == 0){
    printf("child: exiting\n");
} else if(pid > 0){
    printf("parent: child=%d\n", pid);
} else {
    printf("fork error\n");
}
```

Solo lo ejecuta el hijo

Solo lo ejecuta el padre

Ejecutar un programa: `exec()`

- El sistema operativo ofrece la llamada `exec()` para reemplazar el **contenido del proceso** por un **nuevo programa**.
- Después de ejecutar `exec()`, el código, la memoria, variables, data segments cambian, pero su **ID de proceso y sus File Decriptors** siguen siendo los mismos.

Terminar un proceso: `exit()`

- La llamada `exit()` finaliza la ejecución de un proceso.
- La **recogida de estado del proceso** se realiza después de la finalización, lo que es gestionado por el sistema operativo.

Esperar la terminación de un proceso: `wait()`

- Los procesos pueden usar `wait()` para **esperar** a que un **proceso hijo termine** antes de continuar su ejecución.
- Evita que el proceso hijo se quede como **zombie**, es decir, sin ser limpiado del sistema tras su terminación.

```
int pid = fork();
 if(pid == 0){
    printf("child: exiting\n");
    exit(0);
 } elseif(pid > 0){
    printf("parent: child=%d\n", pid);
    pid = wait((int *) 0);
    printf("child %d is done\n", pid);
 } else {
    printf("fork error\n");
}
```

getpid():
Devuelve el ID del proceso actual, permitiendo su identificación dentro del sistema.

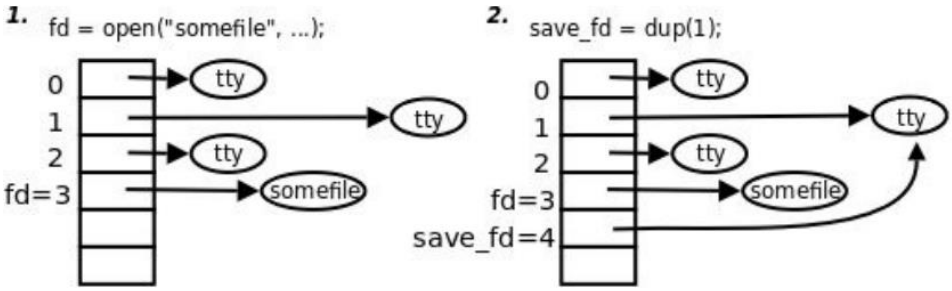
execve():

- Ejecuta el programa al que hace referencia el nombre de pathname, con los argumentos que se le envían por argv[].
- No cambia el pid ni modifica los File Descriptors
- Ej: execve("/bin/echo", argv, NULL);

kill(pid):
Envía una señal para terminar el proceso especificado por su ID (pid), permitiendo la terminación controlada de procesos.

pipe():
Crea un canal de comunicación entre procesos, permitiendo que uno lea datos por un extremo y el otro escriba datos por el otro extremo.

dup():
Duplica un descriptor de archivo, creando una copia exacta de él. Esto es útil para redirigir la entrada o salida de un proceso.



Comunicación entre procesos (Inter-process Communication, IPC)

- Aunque no se profundiza en este capítulo, se menciona que los procesos pueden **comunicarse** entre sí a través de diversos mecanismos de **IPC** como **pipes**.
- Un **pipe** es un pequeño buffer en el kernel expuesto a los procesos como un par de files descriptors, uno para escritura y otro para lectura. Al escribir datos en el extremo el pipe hace que estos estén disponibles en el otro extremo para ser leídos.
- Ej: int pipe(int pipefd[2]);

El ciclo de vida de un proceso:

- El ciclo de vida de un proceso generalmente sigue estos pasos:
    1. **Creación**: `fork()`
    2. **Ejecución**: ejecuta código propio o un nuevo programa con `exec()`
    3. **Terminación**: `exit()`
    4. **Espera**: el padre espera con `wait()` a que los hijos terminen.

Zombie processes

- Un **proceso zombie** es un proceso que ha terminado, pero cuyo **estado de salida** no ha sido recogido por su proceso padre.
- El sistema operativo mantiene el proceso en la tabla de procesos hasta que el padre lee el código de salida usando `wait()`.

Procesos huérfanos (Orphan Processes)

- Un **proceso huérfano** es un proceso cuyo **padre ha terminado** antes que él.
- Estos procesos son **adoptados por el proceso `init`** (PID 1), que se convierte en su nuevo padre y recoge su estado de salida.

Control de señales

- Los procesos pueden enviar **señales** a otros procesos para **notificarles eventos** o **controlar su comportamiento**.
    - Ejemplo: `SIGTERM` para terminar un proceso.
    - Las señales pueden ser **enviadas**, **ignoradas**, o **manejadas** por los procesos a través de manejadores de señales.

Ejercicio de parcial:

Ping pong:

```c
#define READ 0
#define WRITE 1
#define INT_MAX

int main(void) {
    int pelota = 0;

    int padre_a_hijo[2], hijo_a_padre[2];
    pipe(padre_a_hijo);
    pipe(hijo_a_padre);

    pid_t pid = fork();
    if (pid) { // PADRE
        close(padre_a_hijo[READ]);
        close(hijo_a_padre[WRITE]);
        while(pelota >= 0 && pelota <= INT_MAX) {
            pelota++;
            write(padre_a_hijo[WRITE], &pelota, sizeof(pelota));
            read(hijo_a_padre[READ], &pelota, sizeof(pelota));
        }
        close(padre_a_hijo[WRITE]);
        close(hijo_a_padre[READ]);
    }
    else { // HIJO
        close(padre_a_hijo[WRITE]);
        close(hijo_a_padre[READ]);
        while(pelota > 0 && pelota <= INT_MAX) {
            read(padre_a_hijo[READ], &pelota, sizeof(pelota));
            pelota++;
            write(hijo_a_padre[WRITE], &pelota, sizeof(pelota));
        }
        close(padre_a_hijo[READ]);
        close(hijo_a_padre[WRITE]);
    }
    return 0;
}
```

Tema: Virtualización de memoria

Memoria virtual

- Es una **abstracción** que permite a cada proceso actuar como si tuviera su propio espacio de direcciones de memoria continuo, aunque físicamente esté distribuido o compartido.
- Permite **aislar procesos**, proteger la memoria y usar más memoria que la física disponible.

Traducción de direcciones

- Cada dirección virtual generada por un proceso debe ser **traducida a una dirección física** real por el sistema.
- Esta traducción se realiza mediante una **unidad de gestión de memoria (MMU)**, controlada por el sistema operativo.

Time Sharing:

Tiempo compartido se refiere **a compartir de forma concurrente** un **recurso computacional** entre muchos usuarios por medio de las tecnologías de multiprogramación y la inclusión de interrupciones de reloj por parte del sistema operativo.

Páginas y marcos (pages and frames)

- La memoria se divide en **bloques fijos** llamados:
  - **Páginas** (en memoria virtual).
  - **Marcos (frames)** (en memoria física).
- La traducción consiste en mapear una página virtual a un marco físico.

Tablas de páginas (page tables)

- El sistema usa una **tabla de páginas** para **cada proceso** que indica **a qué marco físico** corresponde cada página virtual.
- La MMU consulta esta tabla para realizar la traducción.
- Cada proceso tiene su propia tabla de páginas.

Estructura de una dirección virtual

- Una dirección virtual se divide en dos partes:
  - **Número de página** (índice en la tabla de páginas).
  - **Desplazamiento** (offset dentro de la página).

Aceleración con TLB (Translation Lookaside Buffer)

- La **TLB** es una pequeña caché que almacena traducciones recientes de direcciones para acelerar el acceso.
- Si la traducción no está en la TLB (**TLB miss**), se consulta la tabla de páginas completa.

Protección de memoria

- Cada entrada en la tabla de páginas puede tener **permisos** (lectura, escritura, ejecución).
- Esto previene accesos indebidos entre procesos y mejora la seguridad del sistema.

Ciclo de traducción

1. El proceso genera una **dirección virtual**.
2. La **MMU** busca la traducción en la **TLB**.
3. Si hay fallo en la TLB, se consulta la **tabla de páginas**.
4. Se obtiene la **dirección física**.
5. Se accede a la **memoria física**.

¿Qué es un espacio de direcciones?

- Es la **visión que tiene un proceso** sobre su memoria.
- Cada proceso cree tener acceso exclusivo a un conjunto contiguo de memoria, empezando en dirección 0.
- Esta es una **abstracción proporcionada por el sistema operativo** para ocultar los detalles físicos de la memoria.

¿Por qué usar espacios de direcciones?

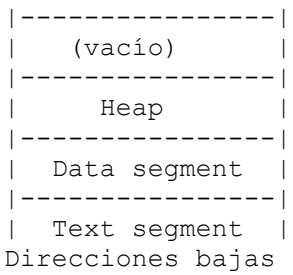
- **Aislamiento:** evita que un proceso interfiera con otro.
- **Seguridad:** protege la memoria de cada proceso.
- **Facilidad de uso:** los programadores pueden trabajar como si tuvieran acceso total a una gran memoria continua.

Partes de un espacio de direcciones típico:

- **Texto (Text segment):** código ejecutable del programa.
- **Datos (Data segment):** variables globales y estáticas.
- **Heap:** memoria dinámica (malloc, new...).
- **Stack:** memoria para llamadas a funciones, parámetros, variables locales.
- **Espacio libre:** puede crecer el heap o el stack si es necesario.

4. □ Ejemplo gráfico típico del espacio de direcciones:

```
Direcciones altas
------------------
|    Stack       |
```



¿Cómo se implementa esta abstracción?

- El sistema operativo y la **MMU (unidad de gestión de memoria)** traducen las **direcciones virtuales** a **físicas**.
- Esto permite que cada proceso tenga su **propio espacio de direcciones privado**, aunque todos compartan la misma memoria física.

Proceso de traducción

1. El proceso genera una dirección virtual (por ejemplo, 0x0040).
2. El **hardware consulta la tabla de páginas** usando el número de página.
3. Se obtiene el marco físico correspondiente.
4. Se suma el offset para obtener la dirección física completa.

Ejemplo:

Si la dirección virtual es 0x1234 y el sistema usa páginas de 1 KB (1024 bytes):

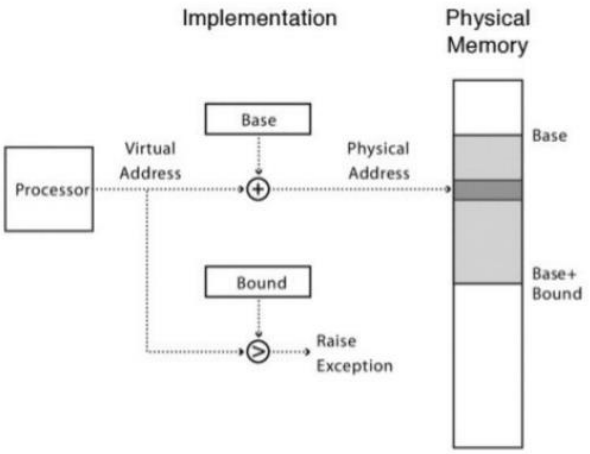
- Número de página = 0x1234 / 1024 = 0x1
- Offset = 0x1234 % 1024 = 0x234

Supongamos que la tabla de páginas dice que la página virtual 0x1 está en el marco físico 0x8.
Entonces, la dirección física es:
0x800 + 0x234 = 0xA34

Base and Bound:

Técnica clásica utilizada por los sistemas operativos para implementar protección de memoria y traducción de direcciones en sistemas con segmentación simple.

Supongamos que un proceso quiere acceder a la dirección 20 (dirección lógica):
El hardware verifica si 20 < bound:
Si no lo es → Violación de memoria (segmentation fault).
Si pasa el control, el sistema suma la dirección lógica a la base:
Dirección física = base + 20



Segmentación

- La **segmentación** divide la memoria en **segmentos con significado lógico**, como:
  - Código (text)
  - Datos (data)
  - Heap
  - Stack
- Cada segmento tiene:
  - Un **base address** (inicio del segmento en memoria física).
  - Un **límite (limit)** que indica el tamaño del segmento.

Estructura de una dirección virtual segmentada:

- Una dirección virtual tiene dos partes:
  - **Número de segmento**
  - **Offset** dentro del segmento

Ejemplo:
Una dirección virtual (2, 100) significa:
Segmento 2, desplazamiento 100 dentro de ese segmento.

Ventajas:

- Espacios de direcciones más organizados.
- Soporta segmentos de distintos tamaños (a diferencia de páginas fijas).
- Facilita el crecimiento independiente del stack y heap.

Desventajas:

- **Fragmentación externa:** como los segmentos son de distintos tamaños, es difícil encontrar espacio contiguo para nuevos segmentos.
- Menos flexible que la paginación en términos de gestión de memoria.

Paginación

- Es una técnica de gestión de memoria que divide:
  - El **espacio de direcciones virtuales** en **páginas** (virtual pages).
  - La **memoria física** en **marcos de página** (page frames).
- Las páginas y marcos tienen **tamaño fijo** (por ejemplo, 4 KB).

Ventaja frente a segmentación:

- **Evita la fragmentación externa**, ya que cualquier página puede ser colocada en cualquier marco disponible.
- Es más simple para el sistema operativo administrar bloques de tamaño fijo.

Protección y aislamiento:

- Cada entrada en la tabla de páginas puede tener:
  - Permisos de acceso (lectura, escritura, ejecución).
  - Bit de presencia (para saber si la página está en memoria).
- Se puede proteger regiones de memoria fácilmente.

Costos:

- Requiere almacenar y consultar la **tabla de páginas**, que puede ser grande si el espacio de direcciones virtuales es amplio.
- Necesita soporte de hardware (unidad MMU) para hacer eficiente la traducción.

Ejemplo:

Con páginas de 4 KB (2^{12} bytes), una dirección virtual de 32 bits se divide así:

- 20 bits para el número de página virtual.
- 12 bits para el offset.

Si la tabla de páginas dice que la VPN 0x123 está en el marco 0x8A, la dirección física es: `0x8A000 + offset`.

Los registros importantes para la paginación son: CR0 y CR3.

El bit más a la izquierda del registro **CR0** es el **PG** (paging), si vale **0** determina que la **dirección virtual** se convierte **directamente** en **memoria física** para acceder a la memoria. Si vale **1**, es que la **paginación** esta **habilitada** por lo que se utiliza la **MMU** para **traducir** la **dirección virtual** a la **física** con las **page tables**. Para que el **PG funcione**, el bit **PE** (bit 0, modo **Protegido**) **también** debe estar **activado**.

El **CR3** se **utiliza cuando** la **paginación** está **habilitada**. Este **contiene** la **dirección física** de la **Page Directory** que contiene **1024 entradas** de **4 bytes** cada una, las cuales **direccionan** a una **Page Table** con **1024 entradas** también.

MMAP:

Permite que una parte de tu **memoria virtual** esté "conectada" directamente a un archivo o dispositivo, **sin** tener que **cargar todo** de golpe. El **kernel** hace el trabajo pesado por vos.

Es útil porque **no** tenés que copiar el contenido de un archivo a **memoria** manualmente. El sistema operativo se encarga de traerlo a **RAM** cuando lo **necesitás** (¡gracias a la **paginación**!). Esto ahorra tiempo y memoria.

¿Por qué la traducción con paginación puede ser lenta?

- Cada acceso a memoria requiere:
    1. **Buscar en la tabla de páginas** (para traducir dirección virtual a física).
    2. Luego, **acceder a la dirección física**.
- Esto **duplica** el número de accesos a memoria.

Solución: TLB (Translation Lookaside Buffer)

- La **TLB** es una **caché de hardware** que almacena **traducciones recientes** de páginas virtuales a marcos físicos.
- Si la dirección virtual está en la TLB → acceso rápido (TLB hit).
- Si no está → TLB miss, y se consulta la tabla de páginas en memoria.

Implementación:

- La TLB se suele implementar como una caché **asociativa por conjunto o totalmente asociativa**.
- Puede tener **entradas protegidas** con bits de permisos.
- Cada entrada TLB almacena:
    - Número de página virtual.
    - Número de marco físico.
    - Permisos y bits de control.

Desafíos:

- Si hay **cambio de proceso**, se deben invalidar las entradas TLB (context switch).
- Algunos sistemas usan un **ASID (Address Space Identifier)** para identificar qué proceso generó cada entrada TLB → reduce invalidaciones.

Tema: Scheduling

Scheduling

- Es la **decisión del sistema operativo** sobre **qué proceso ejecutar** en un momento dado.
- El objetivo es **maximizar el rendimiento**, **minimizar la latencia** y **garantizar la equidad**.
- Es **fundamental en sistemas multitarea**, donde hay más procesos listos para ejecutarse que CPUs disponibles.

Contexto: proceso vs. CPU

- La CPU puede ejecutar **un solo proceso** a la vez (en un núcleo).
- El planificador debe decidir **cuál de los procesos listos** pasa a ejecutarse.

Runqueue:

Una **runqueue** (cola de ejecución) es una estructura de datos utilizada por el **scheduler** para gestionar y hacer un seguimiento de todos los procesos que están listos para ejecutarse en un núcleo de CPU.

Cada núcleo de CPU tiene su **propia runqueue**, que contiene las tareas que están **listas para ejecutar** pero que no están ejecutándose en ese momento.

Los procesos ejecutables se colocan en la **runqueue**, y se extrae de la misma el que tiene **menor vruntime**. Es esencialmente una **priority-queue**, donde la prioridad es el menor vruntime.

VRUNTIME:

Virtual Runtime (vruntime): Cada proceso tiene asociado un contador llamado **vruntime**, que representa la **cantidad de tiempo** que el proceso ha estado ejecutándose en el **CPU**. El vruntime se **incrementa** con cada **tick** del **reloj** mientras el proceso está en ejecución.

Los procesos que consumen vruntime mas **lentamente** son los que tienen **mayor** prioridad para ser ejecutados.

$$\text{vruntime} += \frac{\text{delta\_exec}}{\text{weight}}$$

- **delta_exec** es la cantidad de tiempo que el proceso ha estado ejecutando desde la última actualización.
- **weight** es el peso asociado a la prioridad del proceso.

Nice:

El valor **nice** permite a los usuarios manipular la **prioridad** de un proceso de manera que los procesos con una **mayor** prioridad obtengan **más** tiempo de CPU comparado con los que tienen una **menor** prioridad.

Se puede configurar con la system call **nice()**

Asignación de CPU: Un valor **nice bajo** (más cercano a -20) hace que el proceso sea **más** "**egoísta**", obteniendo más tiempo de CPU. Por el contrario, un valor **nice alto** (más cercano a 19) hace que el proceso sea **más** "**generoso**", cediendo el tiempo de CPU a otros procesos.

Latencia objetivo (sched_latency):

Este es un valor que define el período de tiempo durante el cual el scheduler intenta que todos los procesos ejecutables se ejecuten al menos una vez.

Mínimo tiempo de ejecución garantizado (min_granularity):

Este valor define el mínimo tiempo de ejecución que un proceso debería tener antes de ser desalojado del CPU.

Objetivos típicos de un scheduler:

- **Maximizar el throughput** (cantidad de trabajos completados por unidad de tiempo).
- **Minimizar la latencia** (tiempo de respuesta).
- **Minimizar el tiempo de espera** en la cola.
- **Equidad** entre procesos.
- **Prioridades**: algunos procesos pueden tener prioridad sobre otros.

Estados de un proceso:

- **Ejecutando (Running)**.
- **Listo (Ready)**.
- **Esperando (Waiting o Blocked)**.

El scheduler decide entre los procesos en el estado **Ready**.

Trampoline:

- Código assembler para **transicionar** de modo **usuario** a modo **kernel**.
- Conceptualmente es la única página del kernel que está todo el tiempo cargada en el user-space del proceso. Es la única página compartida por el address space del usuario y el address space del kernel.

Trapframe:

- Es el lugar donde el kernel **guarda** “la foto de los **registros**”, para poder **restaurarlos luego**.
- Es un espacio donde el Kernel va a **guardar** el **estado** de todos los **registros** del

procesador, previo a pasar al modo kernel, en el cual no hay garantías de que estos registros se hubieran conservado.

Clasificación:

- Interactivo: necesita tiempos de **respuesta bajos**. Los procesos deben recibir atención frecuentemente. Ejemplo: **Round Robin**.
- Por lotes (batch): los procesos pueden **tardar más** en completarse. Se optimiza el tiempo total de ejecución más que la respuesta inmediata. Ejemplo: FIFO, SJF.

Para poder hacer algún tipo de análisis se debe tener algún tipo de métrica estandarizada para comparar las distintas políticas de planificación o scheduling. Bajo estas premisas, por ahora, para que todo sea simple se utilizará una única métrica llamada ***turnaround time.*** Que se define como ***el tiempo en el cual el proceso se completa menos el tiempo de arribo al sistema***:

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Tipos de scheduling:

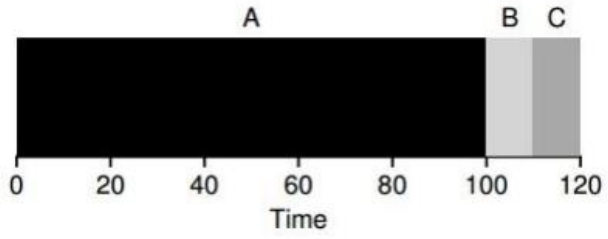
- **Preemptivo**: el sistema puede **interrumpir** un **proceso** en **ejecución** para **darle tiempo** a **otro**. Esto permite más equidad y respuesta rápida (como en **RR** o STCF).
- **No Preemptivo**: una vez que un proceso empieza, **no se interrumpe hasta que termina** (como FIFO y SJF).

Rol del sistema operativo:

- Usar una **estrategia de planificación** definida.
- Implementar estructuras de datos eficientes para manejar la cola de procesos listos.
- Decidir el **momento y la lógica** de la planificación (por ejemplo, tras una interrupción, una E/S o un evento del sistema).

Evaluación de algoritmos de planificación:

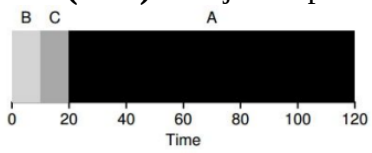
- **First In, First Out (FIFO):** Se ejecuta primero el proceso de mayor duracion



¿Cuánto es el Turnaround?

$$(100+110+120)/3=110$$

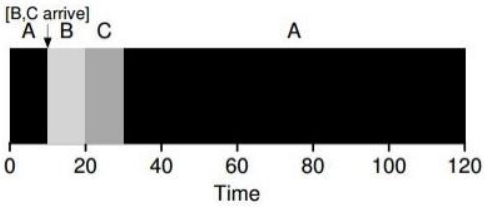
- **Shortest Job First (SJF)**: Se ejecuta primero el proceso de menor duracion



En el mismo caso de arriba, se mejora el turnaround time con el sencillo hecho de ejecutar B, C y A en ese orden:

$$(10+20+120)/3=50$$

- **Shortest Time-to-Completion (STCF):** Cuando un nuevo job llega al sistema, el scheduler determina cuál de los jobs en el sistema (incluyendo al recientemente arribado) le falta menos tiempo para terminar, y elige a dicho proceso, desalojando al proceso actual.

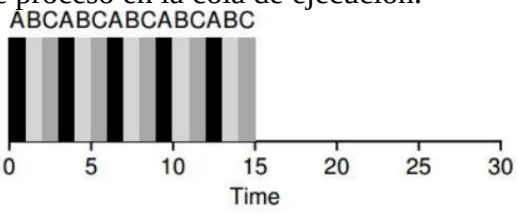


El cálculo para el turnaround time sería

$$(120-0+20-10+30-10)/3=50$$

- **Round Robin**: Se ejecuta un proceso por un período determinado de tiempo (time slice) y transcurrido el período se pasa a otro proceso, y así sucesivamente

cambiando de proceso en la cola de ejecución.



Response time:

(0 + 1 + 2) / 3 = 1

Codigo Round Robin:

```
int ultimo_proceso = -1;

struct p* elegir() {
    int i;
    int total_procesos = 64;

    // Comenzar a buscar desde el siguiente proceso después del último seleccionado
    for (i = 1; i <= total_procesos; i++) {
        int indice = (ultimo_proceso + i) % total_procesos; // Circular
        if (procesos[indice].status == RUNNABLE) {
            ultimo_proceso = indice; // Actualizamos el último proceso elegido
            return &procesos[indice];
        }
    }

    // Si no hay ningún proceso RUNNABLE, retornar NULL
    return NULL;
}
```

Resumen:

Algoritmo	Tipo de sistema	¿Preemptivo?	Tiempo de respuesta	Turnaround Time
FIFO	Batch	❌ No	❌ Malo (puede ser alto)	⚠️ Variable (puede ser malo)
SJF	Batch	❌ No	❌ Malo (espera inicial larga)	✅ Excelente (mínimo promedio)
STCF	Batch	✅ Sí	✅ Bueno (procesos cortos responden rápido)	✅ Muy bueno (casi óptimo)
RR	Interactivo	✅ Sí	✅ Excelente (respuesta rápida inicial)	❌ Regular (turnaround más alto)

¿Qué es MLFQ?

- Es un algoritmo de planificación basado en **múltiples colas de prioridad**.
- Cada cola representa un **nivel de prioridad diferente**.
- Los procesos **pueden moverse entre colas** según su comportamiento.
- **Feedback**: el sistema observa cómo actúan los procesos y **ajusta dinámicamente sus prioridades**.

Reglas típicas del MLFQ:

1. **Muchas colas de prioridad**, donde la 0 es la de mayor prioridad.
2. Siempre se elige un proceso de la **cola de mayor prioridad disponible**.
3. Dentro de una cola, puede usarse **Round-Robin** u otra política.
4. Si un proceso **usa toda su porción de CPU**, baja de nivel (se le considera CPU-bound).
5. Si un proceso **cede la CPU antes de terminar su porción**, puede mantenerse o subir (se le considera I/O-bound o interactivo).
6. Después de cierto tiempo, se puede aplicar **"boosting"**: todos los procesos se devuelven a la cola más alta para evitar hambre (starvation).

¿Por qué MLFQ es útil?

- Distingue entre procesos **interactivos** y **pesados en CPU**.
- Intenta **favorecer la respuesta rápida** para procesos interactivos.
- Se adapta al comportamiento dinámico de los procesos.
- Evita el **starvation** con mecanismos de **recuperación de prioridad**.

Reglas:

REGLA 1: si la prioridad(A) > prioridad(B), (A) se ejecuta y (B) no.

REGLA 2: si la prioridad(A) = prioridad(B), (A) y (B)se ejecutan en Round-Robin.

REGLA 3: Cuando una tarea llega al sistema se la coloca en la más alta prioridad.

REGLA 4: Una vez que una tarea consume su time-slice en un nivel dado (independientemente de cuántas veces haya renunciado al uso de la CPU) su prioridad se reduce; baja un nivel en la cola de prioridad.

REGLA 5: Después de un cierto periodo de tiempo S, se mueven todas las tareas a la cola con mas prioridad

Problemas y soluciones:

- Algunos procesos podrían **"engañar" al scheduler** actuando como interactivos.
- Para resolver esto, se introducen mecanismos de:
    - **Control del uso de CPU**.
    - **Reajuste periódico de colas**.
    - **Tiempos de espera máximos**.

¿Por qué es distinto planificar en multiprocesadores?

- En sistemas con múltiples CPUs o núcleos, hay **varias unidades de ejecución**.

- Se busca **utilizar eficientemente todos los núcleos** sin generar conflictos ni ineficiencias.

Tipos de planificación:

◆ **Planificación Global (Global Scheduling)**

- Hay **una única cola de procesos listos** compartida entre todas las CPUs.
- Cualquier CPU puede tomar un proceso de la cola.
- ✅ Balancea mejor la carga.
- ❌ Puede causar problemas de **sincronización** y **cache misses** si los procesos se mueven entre núcleos.

◆ **Planificación por CPU (Per-CPU Scheduling)**

- Cada CPU tiene su **cola local** de procesos.
- ✅ Mejora la **localidad de caché**.
- ❌ Puede causar **desequilibrio de carga** (una CPU con muchos procesos, otra sin nada que hacer).

◆ **Híbrida**

- Uso de colas por CPU con un **mecanismo de "load balancing"** ocasional para redistribuir procesos.

Problemas comunes:

- **False sharing**: múltiples CPUs accediendo a datos cercanos en memoria que causan invalidaciones de caché.
- **Contention**: varias CPUs queriendo acceder a estructuras compartidas (colas, contadores, etc.).
- **Overhead**: el costo extra de coordinar múltiples CPUs.

Interrupciones y Excepciones

- **Interrupciones**: Son **señales externas** que le indican al procesador que debe dejar de ejecutar el proceso actual para atender otro evento, como una solicitud de hardware (entrada/salida, temporizadores).
    - Ejemplo: El teclado o la tarjeta de red generan una interrupción para que el procesador atienda una acción.
- **Excepciones**: Son **errores internos** o situaciones especiales generadas por el propio procesador o el sistema operativo, como errores de ejecución (división por cero, acceso a memoria no válida).
    - Ejemplo: Intentar dividir un número entre cero genera una excepción que debe ser manejada.

Cómo funciona el manejo de interrupciones y excepciones:

- **Proceso general**:
    1. El procesador recibe una interrupción o excepción.
    2. **Suspende** la ejecución del proceso actual.
    3. Guarda el **estado del proceso** (el contexto).
    4. Ejecuta el **manejador de interrupción/excepción** (un código especial en el sistema operativo).
    5. Después de manejar la interrupción, el procesador **restaurará el estado del proceso** y continuará su ejecución.

Tipos de interrupciones:

- **Interrupciones de hardware**: Generadas por dispositivos externos.
- **Interrupciones de software**: Generadas por el software, generalmente por una System Call.
- **Interrupciones de temporizador**: Generadas por un temporizador para permitir la **multiprogramación** (darle tiempo de CPU a diferentes procesos).

Excepciones:

- Las excepciones generalmente se dividen en dos tipos:
    - **Excepciones sin recuperación**: Errores graves que no pueden recuperarse, como una violación de acceso a memoria.
    - **Excepciones con recuperación**: Situaciones en las que el sistema puede intentar continuar con la ejecución, como el desbordamiento de un número.

Manejadores de interrupciones:

- El **manejador de interrupciones** es un código específico encargado de gestionar la interrupción o excepción.
- Cuando ocurre una interrupción, el sistema operativo se **encarga de determinar qué hacer** con ella (por ejemplo, detener un proceso, gestionar un recurso de hardware, mostrar un mensaje de error, etc.).

Tema: Concurrencia

Concurrencia

- Es la capacidad de un sistema de **ejecutar múltiples tareas aparentemente al mismo tiempo**.
- No implica necesariamente paralelismo (varias tareas al mismo instante), sino que puede ser una **intercalación rápida de tareas** en un solo CPU.

Hilo (thread)

- Es una **unidad de ejecución dentro de un proceso**.
- Un **thread** es una secuencia de ejecución atómica que representa una tarea planificable de ejecución.
- Todos los hilos de un mismo proceso comparten:
    - Código
    - Datos globales
    - Archivos abiertos
- Pero cada hilo tiene su **propio contador de programa, pila y registros**.

Thread Scheduler

En la actualidad hay dos formas de que los threads se relacionan entre sí:

- **Multi-threading Cooperativo**: no hay interrupción a menos que se solicite.
- **Multi-threading Preemptivo**: Es el más usado en la actualidad. Consiste en que un threads en estado de running puede ser movido en cualquier momento.

Ventajas de los hilos

- Permiten dividir una tarea en partes concurrentes (por ejemplo, leer datos, procesarlos y mostrarlos en pantalla al mismo tiempo).
- Son **más ligeros** que los procesos (menos costosos de crear y cambiar de contexto).
- Útiles en programas que realizan **entrada/salida y cómputo al mismo tiempo**.

Desafíos de la concurrencia:

- **Condiciones de carrera (race conditions)**: cuando dos hilos acceden y modifican datos compartidos al mismo tiempo sin coordinación, pueden producir resultados impredecibles.
- **Interbloqueos (deadlocks)**: cuando dos o más hilos esperan indefinidamente por recursos que se bloquean mutuamente.
- **Inanición (starvation)**: cuando un hilo nunca consigue acceder a un recurso porque otros siempre lo acaparan.
- **No determinismo**: El orden de ejecución de los hilos **puede variar en cada ejecución**, lo que hace que los errores sean difíciles de reproducir

Sincronización:

- Para evitar errores, se deben usar mecanismos de sincronización:
    - **Mutexes**: bloqueos mutuos para asegurar que solo un hilo acceda a una sección crítica a la vez.
    - **Semáforos**: variables de conteo que controlan el acceso a recursos compartidos.
    - **Barreras**: puntos donde varios hilos deben llegar antes de continuar.

Motivación:

- Aumentar el **desempeño** en máquinas multiprocesador.
- Mejorar la **capacidad de respuesta** de las aplicaciones.
- Permitir que un programa siga ejecutándose mientras otra parte está esperando (por ejemplo, un programa que descarga archivos mientras muestra el progreso al usuario).

Ejemplo clásico:

Dos hilos incrementan una misma variable compartida sin sincronización. Cada uno lee el valor, lo incrementa y lo escribe, pero si se intercalan mal, se puede perder un incremento.

Interleavings:

- El sistema operativo puede **interrumpir y cambiar de hilo** en cualquier momento.
- Esto crea muchas **posibles interleavings (intercalaciones de instrucciones)**, lo que hace que depurar errores de concurrencia sea muy difícil.

Funciones básicas de la API de hilos (basada en `pthread`):

Crear un hilo:

```
pthread_create(&tid, NULL, thread_function, arg);
```

- `tid`: identificador del hilo.
- `NULL`: atributos del hilo (opcional).
- `thread_function`: función que ejecutará el hilo.
- `arg`: argumento que se pasa a la función del hilo.

Terminar un hilo:

```
pthread_exit(NULL);
```

- Finaliza el hilo actual. También puede devolver un valor.

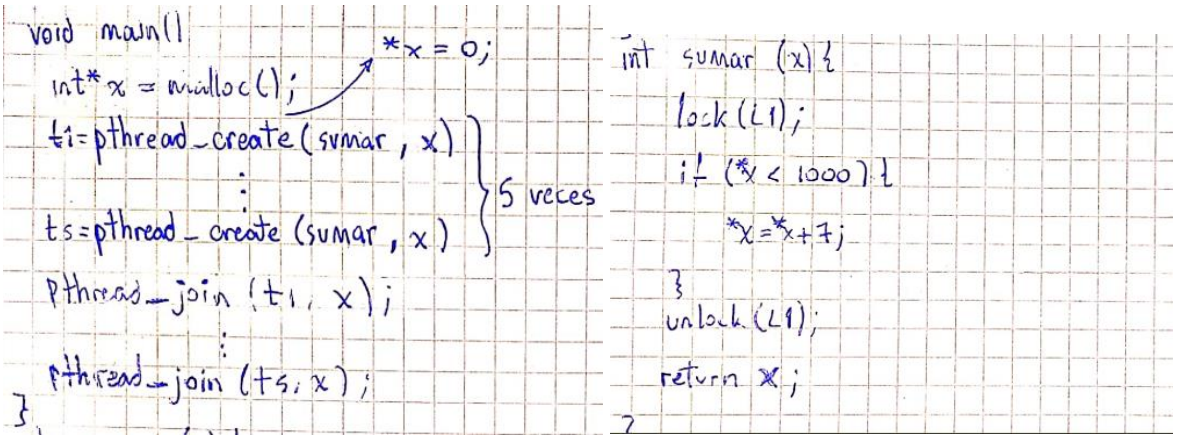
Esperar a que un hilo termine (join):

```
pthread_join(tid, NULL);
```

- Espera a que el hilo `tid` finalice antes de continuar.

Ejercicio de Parcial:

Programa con 5 threads que incrementa variable compartida de a 7 hasta llegar a 1000:



Pasar argumentos:

- Se pueden pasar punteros a estructuras como argumentos para que el hilo tenga más de un dato de entrada.

Compartir datos:

Como los hilos comparten el mismo espacio de direcciones, **pueden acceder a:**
- El Código
- Variables Globales
- Variables del Heap

Process primitive	Thread primitive	Description
fork	pthread_create	crea un nuevo flujo de control
exit	pthread_exit	sale de un flujo de control existente
waitpid	pthread_join	obtiene el estado de salida de un flujo de control
atexit	pthread_cleanup	función a ser llamada en el momento de salida de un flujo de co
getpid	pthread_self	obtiene el id de un determinado flujo de control
abort	pthread_cancel	terminación anormal de un flujo de control

El Estado Per-thread y Threads Control Block (TCB):

Thread Control Block (TCB): Estructura que representa el **estado** del **thread**, en el cual se crea **una entrada** por cada thread. Este **almacena** el estado **per-thread**, osea el **estado** de **Computo** que debe ser realizado por el thread.

Metadata del thread:

Comando para mostrarla: **ls –l <ruta>**

Por cada thread se debe guardar determinada información sobre el mismo:
- ID
- Prioridad de scheduling
- Status

Estados:

- **Init**: Un thread se encuentra en estado INIT mientras se está inicializando el estado per-thread y se está reservando el espacio de memoria necesario para estas estructuras.
- **Ready**: Un thread en este estado está listo para ser ejecutado, pero no está siendo ejecutado en ese instante.
- **Running**: Un thread en este estado está siendo ejecutado en este mismo instante por el procesador.
- **Waiting**: En este estado el Thread está esperando que algún determinado evento suceda.
- **Finished**: Un thread que se encuentra en estado FINISHED nunca más podrá volver a ser ejecutado.

Diferencias Proceso/Thread

Los threads

Por defecto comparten memoria
Por defecto comparten los descriptores de archivos
Por defecto comparten el contexto del filesystem
Por defecto comparten el manejo de señales

Los Procesos

Por defecto no comparten memoria
Por defecto no comparten los descriptores de archivos
Por defecto no comparten el contexto del filesystem
Por defecto no comparten el manejo de señales

Comparación directa

Característica	Proceso	Thread
Espacio de memoria	Independiente	Compartido entre threads del mismo proceso
Stack	Propio	Propio
Registros de CPU	Propios	Propios
Comunicación	Más lenta (IPC)	Rápida (memoria compartida)
Costo de creación	Alto (fork)	Bajo
Robustez	Aislado, más seguro	Menos aislado, más riesgo

la

Lock

- Un **lock** es un mecanismo que **asegura acceso exclusivo a una sección crítica** del código: una parte donde se accede o modifica un recurso compartido.
- Permite que **solo un hilo a la vez** entre en esa sección.

Sección crítica:

- Es la parte del código donde hay acceso a datos compartidos que no deben ser modificados por más de un hilo simultáneamente.

Protocolo básico:

```
lock();
  // sección crítica
unlock();
```

Requisitos para un buen lock:

- **Mutua exclusión**: solo un hilo accede a la sección crítica a la vez.
- **Progreso**: si varios hilos quieren entrar, alguno debe poder avanzar.
- **Equidad**: no debe haber hilos que se queden esperando para siempre (no **starvation**).
- **Eficiencia**: debe ser rápido en ausencia de contención.

Implementación:

API de Locks

```
pthread_mutex_t lock;
int pthread_mutex_init(&lock,NULL);
int pthread_mutex_lock (pthread_mutex_t * mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t * mutex);
int pthread_mutex_timedlock(pthread_mutex_t * mutex, struct timespec
*abb_timeout);
```

Implementación con instrucciones atómicas:

- Se necesita usar **instrucciones atómicas** como `test-and-set` o `compare-and-swap` (que proporcionan los procesadores).

Ejemplo con `test_and_set`:

```
while (test_and_set(&lock)) ;  // spinlock

Realiza todas las siguientes operaciones de manera atómica
1. Lee el valor actual de la variable.
2. Asigna un nuevo valor a la variable.
```

```
3. Devuelve el valor original de la variable.
```

Spinlock:

- Tipo de lock que **gasta CPU esperando activamente** (busy waiting).
- Es útil porque en lugar de bloquear o suspender el hilo, el CPU da vueltas (spin) en un bucle corto.
- Esto es útil cuando se espera que el lock se libere muy pronto.
- Se evita el costo de suspender y reactivar un hilo/proceso.

Sleep lock:

- En vez de usar CPU activamente, un hilo puede **dormir y ser despertado**.
- Esto requiere soporte del sistema operativo (bloqueo basado en semáforos o mutexes reales).
- Util en sistemas con espera larga, con un solo procesador.

Principales problemas:

1. Race Conditions (Condiciones de carrera)

- Ocurren cuando **el resultado del programa depende del orden en que se ejecutan los hilos**.
- Especialmente peligrosas si varios hilos **leen y escriben** la misma variable sin protección.
- Ejemplo típico: incrementos no atómicos de una variable global.

2. Deadlock (Interbloqueo)

- Se produce cuando **dos o más hilos se bloquean entre sí esperando recursos** que el otro posee.
- Los cuatro requisitos para que ocurra un deadlock son:
    - **Mutua exclusión** (un recurso solo puede ser usado por un hilo a la vez),
    - **Retención y espera** (un hilo mantiene recursos mientras espera otros),
    - **No expropiación** (los recursos no pueden ser quitados),
    - **Espera circular** (hay un ciclo de hilos esperando unos por otros).

3. Livelock

- Similar al deadlock, pero los hilos **siguen ejecutando código** sin progresar porque ceden el recurso constantemente al otro.

4. Starvation (Inanición)

- Un hilo **nunca obtiene acceso a un recurso** porque otros hilos siempre lo ganan antes.

- Puede deberse a prioridades mal asignadas o a políticas injustas de scheduling.

5. Atomicidad rota

- Supone que una operación es atómica (indivisible), pero en realidad no lo es.
- Ejemplo clásico: leer-modificar-escribir una variable sin protección.

6. Orden de instrucciones inesperado

- Los compiladores o procesadores pueden **reordenar instrucciones** para optimizar.
- Esto puede romper la lógica concurrente si no se usan barreras de memoria adecuadas (fences).

7. Problemas de prueba y depuración

- Los errores de concurrencia son **no deterministas**, es decir, **pueden no aparecer siempre**.
- Esto los hace difíciles de detectar, reproducir y corregir.

Tema: Filesystems

File System

- Es una **abstracción del sistema operativo** que permite almacenar y acceder datos de manera organizada y persistente.
- Transforma el almacenamiento físico (como discos) en una estructura lógica de **archivos y directorios**.

Funciones clave:

- **Nombramiento**: asociar nombres simbólicos a bloques de datos (por ejemplo, `/home/usuario/documento.txt`).
- **Persistencia**: los datos sobreviven reinicios.
- **Compartición y protección**: múltiples usuarios/procesos pueden compartir archivos con control de acceso.
- **Rendimiento**: acceso eficiente a datos, incluso para archivos grandes.

Abstracciones fundamentales:

- **Archivo**: secuencia de bytes con nombre.
- **Directorio**: colección de entradas que mapean nombres a archivos o directorios.
- **Path**: forma de identificar archivos (absoluto o relativo).

Metadatos:

- Información sobre el archivo, como:

    - Tamaño
    - Fecha de modificación
    - Permisos de acceso
    - Ubicación de los bloques de datos

Organización en disco:

- El sistema de archivos gestiona el **espacio del disco**: decide **dónde colocar cada archivo y sus bloques**.
- También maneja problemas como:
    - **Fragmentación**
    - **Bloques libres**
    - **Eficiencia en la lectura/escritura**

Retos principales:

- **Crash consistency**: garantizar que el sistema se mantenga coherente tras un corte de energía o fallo.
- **Escalabilidad**: soportar muchos archivos y acceso simultáneo.
- **Seguridad**: asegurar que solo usuarios autorizados accedan a los datos.

Archivos: la abstracción de almacenamiento

- Un **archivo** es simplemente una **secuencia de bytes**.
- El sistema operativo no impone estructura: el significado de los datos depende del programa que los utiliza.

Directorios: estructura jerárquica

- Los directorios permiten organizar archivos en una **estructura de árbol**.
- Un directorio **contiene entradas con nombres** que apuntan a archivos o a otros directorios.

Nombres de archivos y rutas

- Dos formas de identificar archivos:
    - **Ruta absoluta** (ej. `/home/ana/archivo.txt`)
    - **Ruta relativa** (desde el directorio actual)
- Un archivo puede tener múltiples **nombres (links)** apuntando a él (con hard links).

APIs

- `open()`: abre un archivo.
- `read()`, `write()`: leer y escribir datos.
- `close()`: cierra el archivo.
- `mkdir()`, `rmdir()`: crear y eliminar directorios.

- link(): crea un nuevo nombre para un archivo. Esto también se conoce como un link (hard link).
- symlink(): crea un soft link para un archivo.
- `unlink()`: borra un nombre de archivo (si no quedan nombres, el archivo se elimina).
- `rename()`: cambia el nombre de un archivo o lo mueve de ubicación.

Internamente…

- Cada archivo tiene un **inode** que guarda metadatos: tamaño, permisos, timestamps, ubicación en disco.
- El sistema de archivos mantiene una tabla que mapea **nombres** a **inodos**.

Permisos y usuarios

- Archivos tienen:
  - **Usuario dueño**
  - **Grupo**
  - **Permisos**: lectura (r), escritura (w), ejecución (x) para usuario, grupo y otros.

Componentes principales de un sistema de archivos

- **Superbloque**: contiene metadatos globales del sistema de archivos (como cuántos bloques, cuántos inodos, etc.).
- **Inodos (i-nodes)**: estructuras que describen cada archivo: tamaño, ubicación de datos, permisos, timestamps, etc.
- **Bloques de datos**: contienen el contenido real de los archivos.
- **Bloques de directorios**: contienen pares nombre→número de inodo.
- **Bitmap**: para gestionar qué bloques o inodos están libres u ocupados.

Inodo vs nombre

- El **inodo** representa un archivo concreto.
- El **nombre** es solo una referencia que apunta a ese inodo.
- Esto permite, por ejemplo, tener **hard links** (múltiples nombres para el mismo archivo).

Almacenamiento del contenido de archivos

- Los inodos tienen **punteros a bloques de datos**.
- Como no todos los archivos son del mismo tamaño, se usan niveles:
  - **Punteros directos**: apuntan directamente a bloques de datos.
  - **Punteros indirectos**: apuntan a bloques que contienen punteros.
  - **Doble/triple indirectos**: para archivos grandes.

Asignación de espacio

- Es fundamental decidir **cómo y dónde** poner los bloques de cada archivo en disco:
  - **Contiguo**: rápido, pero poco flexible.
  - **Enlazado**: más flexible, pero lento para acceder aleatoriamente.
  - **Indexado (como con inodos)**: combina lo mejor de ambos.

Consistencia

- Un fallo (como un apagón) puede dejar el sistema de archivos en un estado corrupto.
- Estrategias para mantener la **consistencia**:
  - **Journaling (registro)**: anotar los cambios antes de hacerlos.
  - **Copy-on-write**: hacer una copia de lo modificado antes de sobrescribir.

La estructura de directories

- **Linux utiliza una jerarquía de directorios en forma de árbol**, con `/` como raíz.
- Ejemplo: `/home/usuario/documentos/archivo.txt` representa un camino desde la raíz hasta el archivo.
- Los nombres de archivos pueden contener casi cualquier carácter, excepto `/` y el nulo (`\0`).
- Cada directorio es en realidad **un archivo especial** que contiene una lista de entradas (nombre → inodo).

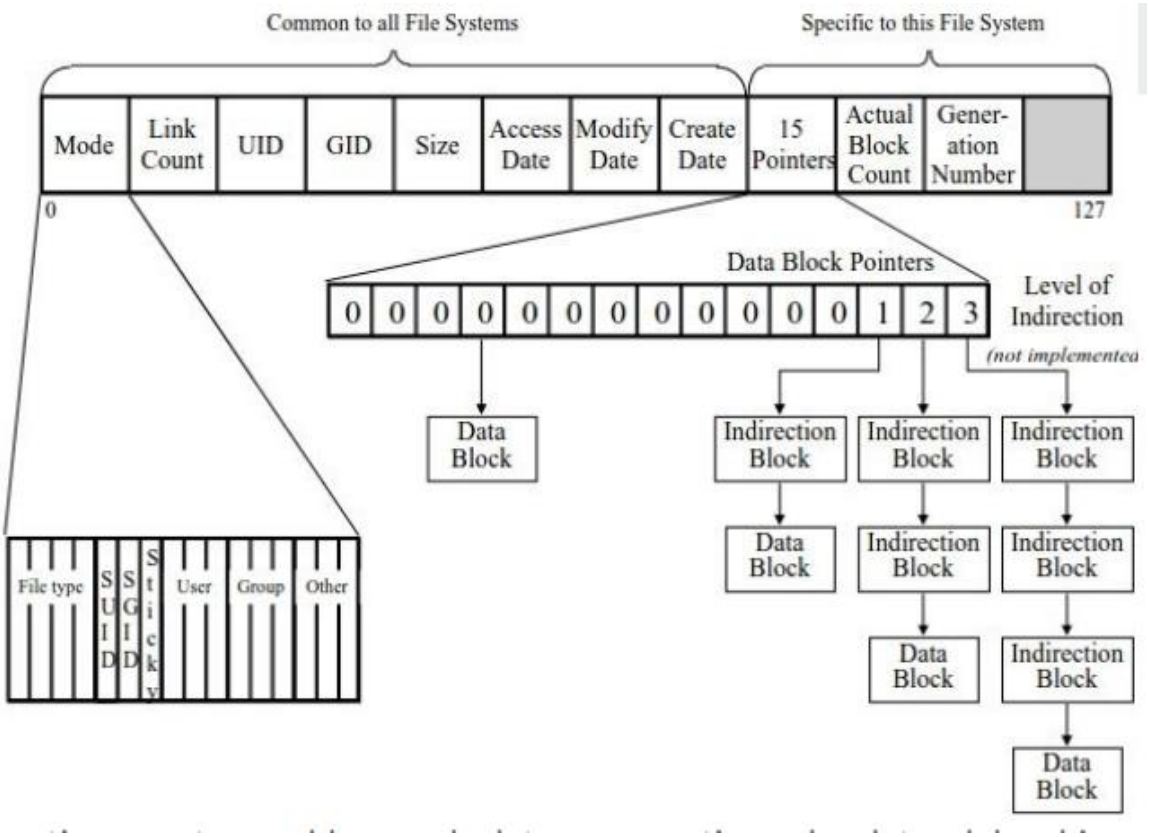
Dentry:

Es una **estructura** de **datos** que el kernel utiliza para representar las **entradas** de un **directorio** en el sistema de archivos, y que se usa para resolver rutas. Cada **dentry** asocia un **número** de **inodo** (que identifica el archivo o directorio) con un **nombre** de **archivo**.

Inodos y enlaces (links)

Inodos

- Un **inodo** contiene la **información del archivo**, como:
  - Tipo (regular, directorio, etc.)
  - Permisos
  - Tamaño
  - Dueño (UID, GID)
  - Timestamps (acceso, modificación, etc.)
  - Ubicación física de los datos



Los **punteros** permiten ubicar los **bloques de datos**, los primeros 12 son **directos**, el 13 es **indirecto simple** (apunta a una tabla con mas bloques), el 14 **indirecto doble** (apunta a una tabla que tiene tablas de bloques) y el 15 **indirecto triple** (apunta a una tabla de tablas de tablas de bloques)

Hard links

- Un **hard link** es **otro nombre** que apunta al mismo inodo.
- Todos los nombres tienen el mismo estatus: no hay "original".
- Se puede borrar un nombre sin eliminar el archivo, mientras otros nombres sigan existiendo.

Soft links (symlinks)

- Un **symlink** es un archivo especial que contiene **una ruta hacia otro archivo**.
- A diferencia del hard link, puede cruzar sistemas de archivos y apuntar a directorios.

Differencias entre hard y softlinks

Característica	Hard link	Soft Link (Symlink)
El inodo apunta a:	Los datos del archivo	Un bloque con la ruta del archivo
Relacion con el archivo	Mismo inodo	Archivo separado
Funcionamiento tras borrar el destino	Sigue funcionando	Se rompe
Directorios	No se permite	Si se permite
Diferentes particiones	No se permite	Si se permite
Tamaño	No ocupa espacio extra	Ocupa como un archivo pequeño

Ejemplo de Parcial:

Si preguntan cuántos inodos y bloques tiene una ruta, se va ingresando a cada inodo de / y luego si busca dentro del bloque de cada uno. Si me piden mostrar los datos (como un ls), se accede al inodo del ultimo archivo de la ruta; si solo me pide mostrar lo que contiene (como un cat) el archivo no hace falta acceder al inodo.

Si pasa lo mismo pero se creo un hardlink y después se hizo un rm del directorio original, no pasa nada porque era un hardlink.

En lo mismo pero tengo un softlink, tengo que acceder primero al original y después volver a contar los inodos y bloques accediendo al path del softlink.

Montaje de sistemas de archivos

- Linux puede **usar múltiples sistemas de archivos simultáneamente**, pero los presenta como **una única jerarquía unificada**.
- El **montaje (mounting)** es el proceso de **adjuntar un nuevo sistema de archivos** a un punto del árbol (p. ej., montar un USB en `/media/usb`).
- Comando típico:

  ```
  mount /dev/sdb1 /mnt/usb
  ```

- También se puede usar `umount` para desmontar.
- Se puede consultar el estado de los sistemas montados con `mount`, `df`, o mirando `/proc/mounts`.

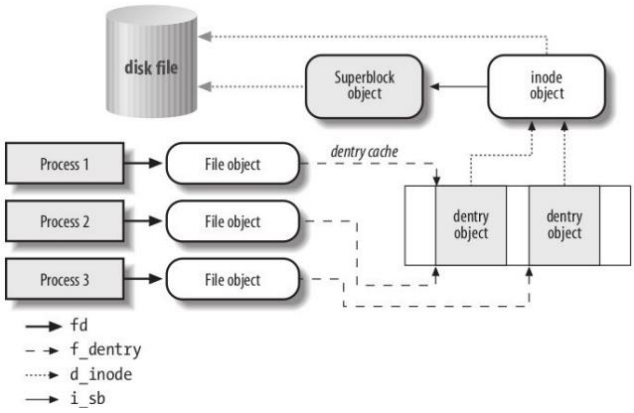
Virtual Files System (VFS):

El **VFS** actúa como un **puente** entre las **llamadas** al **sistema** de **archivos** hechas por los **programas** y las **implementaciones** reales de los **sistemas** de **archivos**.

Subsistema del kernel que implementa la interfaz que tiene que ver con los archivos y el sistema de archivos provistos a los programas corriendo en modo usuario.

Objetos:

El **super bloque**, que representa a un sistema de archivos.
El **inodo**, que representa a un determinado archivo.
El **dentry**, que representa una entrada de directorio, que es un componente simple de un path.
El **file** que representa a un archivo asociado a un determinado proceso



Tipos de dispositivos:

Los **dispositivos** de **carácter** permiten la transferencia de datos byte a byte. Estos dispositivos transmiten la información carácter por carácter, lo que los hace adecuados para dispositivos que **no requieren** procesamiento de **grandes** cantidades de **datos** de una sola vez. Ej: Terminales.

Los **dispositivos** de **bloque** permiten la transferencia de datos en bloques de tamaño fijo. Este tipo de dispositivo es adecuado para hardware que **requiere** el acceso a **grandes** cantidades de **datos** de forma eficiente, como los sistemas de almacenamiento. Ej: Discos Duros.

Cómo se vinculan los dispositivos al filesystem:

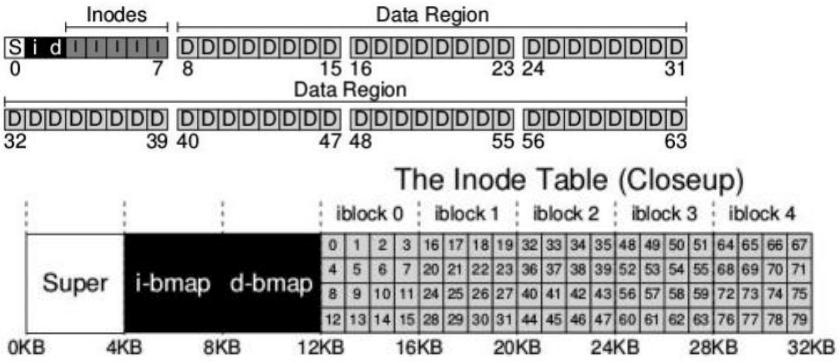
Major Number. El número mayor identifica driver que maneja un grupo específico de dispositivos. El kernel utiliza este número para determinar qué controlador debe gestionar las solicitudes de entrada/salida (I/O) dirigidas a un dispositivo específico.

Minor Number. El número menor identifica un dispositivo específico dentro del grupo de dispositivos gestionado por un mismo controlador.

```
sudo mknod /dev/ttyS1 c 4 65
```

/dev/ttyS1: es el nombre del archivo especial que vamos a crear
C: indica que este es un dispositivo de caracteres, no bloques (sino seria b)
4: es el major
65: es el minor

Very Simple File System



Ejemplo de parcial

Diseñe un vsfs para un sistema de 1024 bloques, cada bloque 4kb, cada inodo 256 bytes.

Criterio de la catedra. Asumir que en un disco de N bloques entran N archivos*

Independientemente de los bloques especiales (además cuando N -> infinito, la suposicion se aproxima)

Resolucion:

- Cantidad de inodos necesarios: 1024
- Cantidad de inodos por bloque: 4096/256 = 16
- Cantidad de bloques de inodos: 1024/16 = 64
- Cantidad de los bitmap: 4096*8=32768 (sobra! El disco tiene 1024 bloques, y 1024 inodos)

** se puede calcular un optimo de la cantidad máxima de inodos, que será generalmente menor a la cantidad de bloques. Hacerlo como ejercicio. Pero a menos que lo pidamos explícitamente, el criterio en rojo es lo que vale en los exámenes (además es más fácil)*

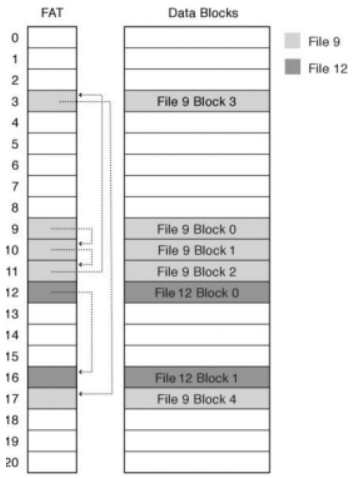
Solucion:

- 1 superbloque
- 1 bloque de bitmap de inodos
- 1 bloque de bitmap de bloques
- 64 bloques de inodos
- 1024-64-1-1-1=957 bloques de datos

FAT:

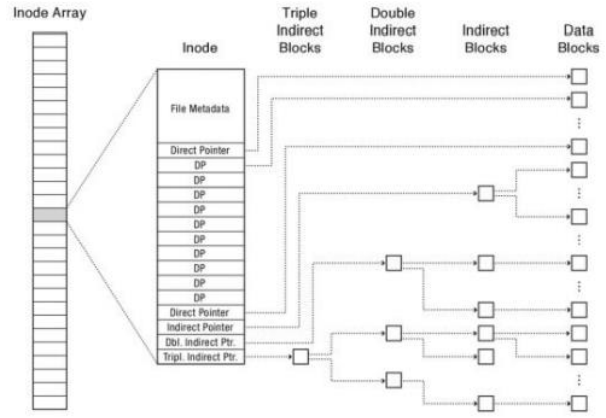
File Allocation Table, un arreglo de entradas de 32 bits, en un área reservada del volumen. Cada archivo en el sistema corresponde a una lista enlazada de entradas en la FAT, en la

que cada entrada en la FAT contiene un puntero a la siguiente entrada. La FAT contiene una entrada por cada bloque de la unidad de disco o volumen



FFS:

Fast File System es un árbol cuidadosamente estructurado que permite a FFS localizar cualquier bloque de un archivo y que es eficiente tanto para grandes como para pequeños archivos



PageCache:

El **buffer cache** tiene dos **funciones**:
1. Sincronizar el acceso a los bloques de disco para asegurar que solo haya una copia de un bloque en la memoria y que solo un hilo del kernel use esa copia a la vez;
2. Almacenar en caché los bloques populares para que no sea necesario volver a leerlos desde el disco, que es más lento.

El **Page Cache** en **Linux** es una parte del sistema de administración de memoria que se utiliza para almacenar en memoria RAM los datos que se han leído o escrito en el disco, con el fin de acelerar el acceso a los archivos.

Para que los datos escritos sean **persistentes inmediatamente** (es decir, que estén almacenados de manera segura en el disco), se debe usar **fsync**() o **fdatasync**() después de la llamada a **write**(). Estas funciones **fuerzan** al **sistema operativo** a escribir cualquier dato pendiente del **page cache** asociado al **archivo** en el **disco**.

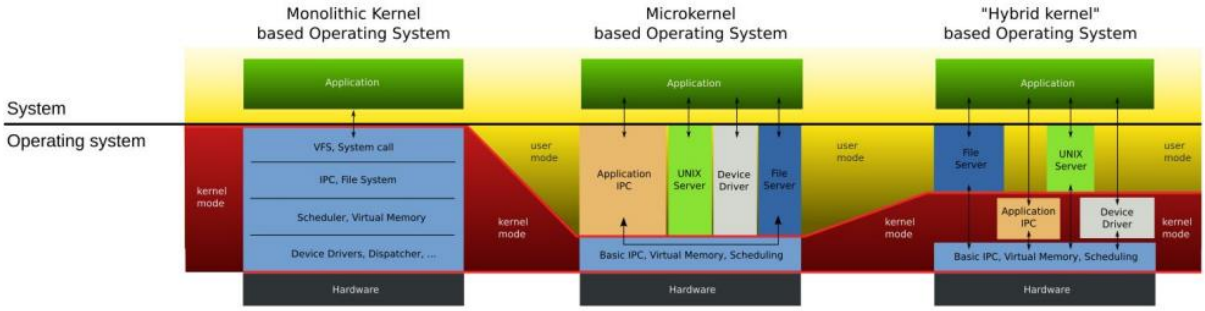
MMAP():

mmap() es una **llamada** al **sistema** en Linux que permite **mapear** un **archivo** o dispositivo directamente en el **espacio** de **direcciones** de un **proceso**. Esto es útil para acceder a grandes cantidades de datos de manera eficiente.
Si se hacen **cambios** a una **región** de memoria mapeada usando **mmap**(), esos cambios pueden **no** ser **persistentes** hasta que el sistema decida **escribir** el **contenido** del **page cache** en el disco. Para **garantizar** que los **cambios realizados** en una **memoria mapeada** se **escriban** de manera persistente en el **disco**, se puede utilizar una llamada explícita a **msync**().

Tipos de Kernel:

En un **kernel monolítico**, el núcleo y la mayor parte del sistema operativo (incluidos drivers, gestión de archivos, red, etc.) se ejecutan en **el mismo espacio de memoria privilegiado** (espacio de kernel). Este modelo es muy **rápido** en cuanto a acceso y rendimiento, ya que todo se ejecuta sin necesidad de pasar mensajes entre componentes separados. Sin embargo, **si un driver tiene un error (bug), puede colapsar todo el sistema**, ya que todo corre con privilegios elevados.

Un **microkernel** es una evolución del kernel monolítico que **minimiza la funcionalidad del núcleo**, limitándolo a tareas esenciales: comunicación entre procesos (IPC), planificación de procesos (scheduling), gestión de interrupciones y memoria básica.
Todo lo **demás** (drivers, sistema de archivos, red, GUI) corre como **servicios en espacio de usuario**, comunicándose mediante mensajes. Esto **mejora** la **estabilidad** y **seguridad**, pero puede tener un **costo en rendimiento** por la sobrecarga de comunicación entre componentes


Monolithic Kernel
based Operating System
Microkernel
based Operating System
"Hybrid kernel"
based Operating System
System
Operating system
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
kernel mode
user mode
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling