Sistemas Operativos

https://github.com/AdriCri22/Sistemas-Operativos-SO-FIB

Tema 1 – Introducción

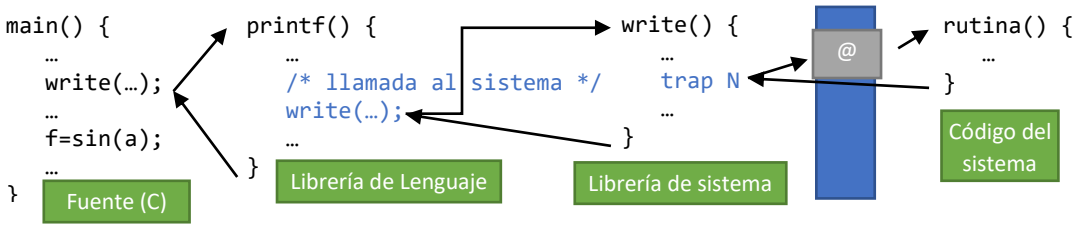
El SO hace de intermediario entre el usuario y el hardware, utilizando eficientemente los recursos disponibles y evitando que el usuario dañe estos mismos, de este modo el SO hace que sea para el usuario un entorno usable, seguro y eficiente.

2 modos (pueden haber más) de ejecución del SO:

- *User Mode* (NO-privilegiado)
- *Kernel mode* (privilegiado): donde dejan ejecutar instrucciones privilegiadas, instrucciones que pueden llegar a dañar la máquina. Formas de acceder al código *kernel*:
  - **Interrupciones** (asíncronas) generadas por el hardware.
  - Los errores de software generan **excepciones** (síncronas).
  - Peticiones de servicio de programas: **Llamada a sistema** (síncronas).

Llamadas al sistema: se hacen desde el código (C, C++, …) se hacen desde el *User Mode*, se envían al *kernel* para su ejecución.

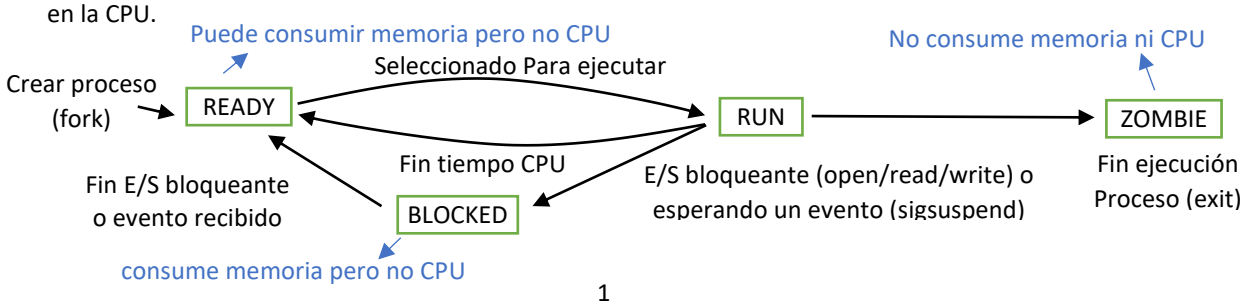
- Librerías "de sistema": contienen la parte compleja de la invocación, sirven para facilitar. (*User Mode*).
- Librerías "de lenguaje": Ofrecidas por los lenguajes de programación, ofrecen funciones de uso frecuente. (*User Mode*).



Tema 2 – Procesos

Conceptos:

- **Proceso**: representación del S.O. de un programa en ejecución. Este es nuevo cada vez que ejecutamos.
- **Programa ejecutable**: tiene código, datos y pila, inicializa los registros de la CPU, da acceso a dispositivos que necesitan acceso al modo *kernel*.
- **PCB** (*Process Control Bank*): gestiona la información de cada proceso. La PCB contiene 3 partes:
  - Espacio de direcciones: (1) de código, pila y datos
  - Contexto:
    - (2) Software: PID (*Process ID*, identificador único para cada proceso generado por el *kernel*), planificación, información sobre el uso de dispositivos, estadística, …
    - (3) Hardware: tabla de páginas, *program counter*, …
- **Paralelismo**: cuando realmente se ejecutan varios procesos a la vez (gracias a una arquitectura multiprocesador o *multi-core*).
- **Concurrencia** (paralelismo virtual): es la capacidad de ejecutar varios procesos de forma simultánea.
- **Procesos secuenciales**: independientemente de la arquitectura, los procesos se ejecutan uno detrás de otro.
- **Hilos de ejecución** (*Threads*): subproceso de ejecución y es la unidad mínima de planificación del SO, un proceso puede tener varios *threads*. Sirven para explotar el paralelismo, encapsular tareas… Son más versátiles que los procesos y al usar la misma memoria en un proceso pueden intercambiar información sin hacer llamadas al sistema.
- **Estados de un proceso**: para garantizar que todos los procesos se ejecutan y ninguno acapare todo el tiempo en la CPU.



Servicios básicos (UNIX)

`fork();`	Crea un proceso hijo en el punto en el que estaba el padre, se ejecutan de forma concurrente.
`waitpid(-1, NULL, 0);` (**Bloqueante**)	Espera a un hijo cualquiera.
`waitpid(pid_hijo, NULL, 0);` (**Bloqueante**)	Espera a un hijo a partir del PID.
`exit(0);`	Termina un proceso sin errores.
`exit(1);`	Termina el proceso con algún error
`execlp(const char *file, const char *arg, ...);`	Sirve para mutar = cambiar de ejecutable.
`getpid();`	Devuelve el PID del proceso hijo.
`getppid();`	Devuelve el PID del padre del proceso.

Bloqueante: es una llamada al sistema que puede forzar que deje el estado RUN (abandone la CPU) y pase a un estado en que no puede ejecutarse (WAITING)

Aspectos que el hijo **HEREDA (fork)**	Aspectos que el hijo **NO HEREDA (fork)**	**CAMBIA (execlp)**	**NO CAMBIA (execlp)**
Código, datos, pila, … La memoria física es nueva y contiene una copia de la del padre	PID, PPID	Se define por defecto el tratamiento de los signals	PID
Programación de *signals*	Contadores internos	Código, datos, pila…	Signals pendientes
Máscara de *signals*	**Alarmas y *signals* pendientes**	direcciones	Contadores internos
Dispositivos virtuales			
userID y *groupID*			
variables de entorno			

Mientras el padre no haga `waitpid` no se libera el espacio que ocupa el PCB del hijo muerto (ESTADO ZOMBIE). Si el padre muere sin liberar los PCB's de sus hijos el sistema los libera (proceso init).

EJEMPLOS

FORK

```c
int ret = fork();
if (ret == 0) {
    // HIJO
} else if (ret < 0) {
    // ERROR
} else {
    // PADRE
    // ret == pid del hijo
}
```

ESQUEMA SECUENCIAL

```c
#define NUM_PROCESOS 2
int ret;
for (int i = 0; i < NUM_PROCESOS; i++) {
    ret = fork();

    if (ret < 0) control_error();

    else if (ret == 0) { // HIJO
        exit(0);
    }
    else { // PADRE
        waitpid(-1, NULL, 0);
    }
}
```

CREAR 2^n PROCESOS

```c
#define N 10

for(int i = 0; i < N; i++) {
    fork();
    printf("Hello World. I'm %d\n", getpid());
}
```

ESQUEMA CONCURRENTE

```c
#define NUM_PROCESOS 2
int ret;
for (int i = 0; i < NUM_PROCESOS; i++) {
    ret = fork();

    if (ret < 0) control_error();

     else if (ret == 0) { // HIJO
        exit(0);
    }
}

while (waitpid(-1, NULL, 0) > 0);
```

MIRAR EL EXIT_STATUS

```c
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

void tratar_exit_code(int status) {
    if (WIFEXITED(status)) {
        // Ha terminado por culpa de un exit
        int exitcode = WEXITSTATUS(status);
        printf("Ha terminado por un exit con exit_code: %d\n", exitcode);
    }

    else {
        // Ha terminado por un signal
        int signalcode = WTERMSIG(status);
        printf("Ha terminado con un signal con signal_code: %d\n", signalcode);
    }
}

int main(int argc, char const *argv[]) {
    for (int i = 0; i < 10; ++i) {

        int ret = fork();// FORK

        if (ret == 0) { // HIJO
            exit(i);
        }

        // PADRE
        int status;
        waitpid(-1, &status, 0);
        tratar_exit_code(status);
    }
    return 0;
}
```

Signals → Notificaciones que puede recibir un proceso (por el usuario o por el *kernel*) para informarle que ha sucedido con un evento (un evento es un *signal* asociado).

Hay dos *signals* que no están asociados a ningún evento SIGUSR1 y SIGUSR2.

Cada proceso tiene un **tratamiento** (que hacer cuando llegue la señal) por defecto asociado a cada *signal*, un tratamiento puede ser modificado excepto SIGKILL y SIGSTOP.

SIGNAL	TRATAMIENTO por defecto	EVENTO	Se pueden Bloquear/Desbloquear
SIGCHLD	IGNORAR	Un proceso hijo ha terminado o ha sido parado	SI
SIGCONT	-	Continua si estaba parado	SI
SIGSTOP	STOP	Parar proceso	No
SIGINT	TERMINAR	Interrumpido desde teclado (Ctrl + C)	SI
SIGALRM	TERMINAR	El contador definido por la llamada ha terminado	SI
SIGKILL	TERMINAR	Terminar proceso	No
SIGSEGV	CORE	Referencia inválida a memoria	No si son provocados por una excepción
SIGUSR1	TERMINAR	Definido por el usuario (proceso)	SI
SIGUSR2	TERMINAR	Definido por el usuario (proceso)	SI

Al recibir un *signal* el proceso interrumpe la ejecución del código y pasa a ejecutar el tratamiento que ese tipo de *signal* tenga asociado y al acabar (si sobrevive) continua donde estaba.

- Una **máscara** es una estructura de datos que permite determinar qué *signals* (solo uno de cada tipo) puede recibir un proceso en un momento determinado de la ejecución.

- Cuando un proceso **bloquea** un *signal* el sistema lo marca como pendiente de tratar
- Cuando el proceso **desbloquea** el *signal* recibe el tratamiento

`kill(int pid, int signal);` → Enviar *signal*

```
// Se usa para cambiar la acción de un signal
sigaction(int signum, struct sigaction *tratamiento,
        struct sigaction *tratamiento_antiguo); →  Reprogramar un signal concreto
```

`struct sigaction`:
- sa_handler
  - SIG_IGN → Ignorar signal recibido
  - SIG_DFL → Tratamiento por defecto
  - my_func → función de usuario con una cabecera predefinida: `void nombre_funcion(int s);`
- sa_mask
  - Vacía → solo se añade el *signal* que se está capturando
  - Al salir se restaura la máscara anterior
- sa_flags
  - 0 → Configuración por defecto
  - SA_RESETHAND → después de tratar el signal se restaura el tratamiento por defecto del *signal*.
  - SA_RESTART → si un proceso bloqueado en una llamada a sistema recibe el *signal* se reinicia la llamada que lo ha bloqueado.

`int sigprocmask(int operacion, sigset_t *mascara, sigset_t *vieja_mascara);` → Bloquear/Desbloquear *signals*.
- SIG_BLOCK → bloquea los *signals* de la máscara que le pases.
- SIG_UNBLOCK → desbloquea los *signals* de la máscara que le pases.
- SIG_SETMASK → intercambia las máscaras.

`int sigsuspend(sigset_t *mascara);` → Esperar hasta que llega un evento cualquiera (Bloqueante).

`int alarm(int num);` → Programa una alarma y devuelve el número de segundos restantes

Manipulación de máscaras con *signals*:

```
sigset_t mask;     // inicia máscara
sigemptyset(&mask);     // vacía
sigfillset(&mask);      // llena
sigaddset(&mask, SIGNUM)    // añade el signal a la máscara
sigdelset(&mask, SIGNUM)    // elimina el signal de la máscara
sigismember(&mask, SIGNUM)  // devuelve true si el signal está en la mascará, false si no.
```

Sincronización de procesos:

- Espera activa → Consume CPU ideal si vas a tardar poco.

```
void configurar_esperar_alarma() {
    alarma = 0;
}

void esperar_alarma() {  // Técnica del pesado
    while (alarma != 1); // Pregunta sin parar si ya se ha recibido el signal
}
```

- Espera pasiva (Bloqueo) → El proceso libera CPU

```
void configurar_esperar_alarma() {
    sigemptyset(&mask);
    sigaddset(&mask, SIGALRM);
    sigprocmask(SIG_BLOCK, &mask, NULL);
}

void esperar_alarma() {
    sigfillset(&mask);
    sigdelset(&mask, SIGALRM);
    sigsuspend(&mask);
}
```

Gestión interna de procesos → para gestionar procesos se necesita:

Estructuras de datos

- **PCB** → Para representar los datos de un proceso
  - PID
  - userID y groupID
  - Estado: RUN, READY, …
  - Espacio para salvar los registros de la CPU
  - Datos para gestionar *signals*
  - Información sobre la planificación
  - Información sobre la gestión de memoria
  - Información sobre la gestión de la E/S
  - Información sobre los recursos consumidos (*Accounting*)
- Gestionar y representar *threads* (depende del SO)

Estructuras de gestión → Que organicen los PCB's en función de su estado o de necesidades de organización del sistema.

- Cola de procesos → Todos los procesos creados en el sistema
- Cola de procesos en READY
- Cola esperando datos de algún dispositivo E/S

} El sistema mueve los procesos de una cola a otra según corresponda.

Algoritmo/s de planificación → Indica como gestionar las estructuras

- Política de planificación → Indica quien entra, durante cuánto tiempo, cuando se evalúa si hay que cambiar el proceso en la CPU y que pasa con el proceso que se estaba ejecutando, esto se ejecuta periódicamente. Las políticas de planificación son:
  - Preemptiva → La política le quita la CPU al proceso aunque este pudiera seguir ejecutándose.
  - No premptiva → La política no le quita la CPU él la "libera".

  Los procesos presentan ráfagas de computación y ráfagas de acceso a dispositivos E/S que lo bloquean
  - Procesos de cálculo → Consumen más tiempo haciendo cálculo que E/S.
  - Procesos E/S: consumen más tiempo haciendo E/S.

Mecanismos → Aplican las decisiones tomadas por el planificador

- Cambios de contexto (Context Switch) → Cuando un proceso deja la CPU y se pone otro proceso.
  - El sistema tiene que salvar el estado del proceso que deja la CPU y restaurar el estado del proceso que pasa a ejecutarse.
  - El cambio de contexto no es tiempo útil de la aplicación, así que hay que hacerlo rápido.

Ejecutando código usuario Proceso A	Modo usuario	int reloj
Salvar contexto Proceso A en PCB[A]	Modo *kernel*	
Planificador decide cambiar a proceso B		
Restaurar contexto Proceso B de PCB[B]		
Ejecutando código usuario Proceso B	Modo usuario	int reloj
Salvar contexto Proceso B en PCB[B]	Modo *kernel*	
Planificador decide cambiar a proceso A		
Restaurar contexto Proceso A de PCB[A]		
Ejecutando código usuario Proceso A	Modo usuario	int reloj

- Tiempo de ejecución de un proceso → Tiempo que pasa desde que llega al sistema hasta que termina.
- Tiempo de espera de un proceso → Tiempo que pasa en READY.

Round Robin (RR) → Los procesos se organizan según su estado. Están encolados por orden de llegada. El proceso recibe la CPU durante un quantum (10ms ó 100ms). El planificador hace una interrupción de reloj para que ningún proceso monopolice la CPU.

Eventos que activan la política Round Robin:

- Cuando el proceso se bloquea (no premptivo) → Pasa a la cola de bloqueados hasta que termina el acceso al dispositivo.
- Cuando el proceso termina (no premptivo) → El proceso pasa a zombie en el caso de Linux o terminaría.
- Cuando termina el quantum (premptivo) → El proceso se añade al final de la cola de READY.

Ningún proceso espera más de (N - 1) * Q milisegundos. Donde N es el número de procesos y Q el tiempo de quantum.

- Q grande → es como si fuesen orden secuencial
- Q pequeño → produce overhead si no es muy grande comparado con el cambio de contexto.

¿Qué hace el kernel cuando se ejecuta un…?

- Fork
  - Busca PCB libre y lo reserva.
  - Inicializar datos (PID, ...).
  - Se aplica política de **Gestión de memoria.**
  - Se actualizan las estructuras de **Gestión de E/S.**
  - En el caso de RR → Se añade a la cola de READY.
- Exec
  - Se substituye el espacio de direcciones por el código/datos/pila del nuevo ejecutable
  - Se inicializan las tablas de *signals*, contexto, …
  - Se actualizan las variables de entorno, argv, registros, …
- Exit
  - Se liberan todos los recursos del proceso.
  - En Linux se guarda el estado de finalización en el PCB y se elimina de la cola de READY.
  - Se aplica la política de planificación.
- Waitpid
  - Se busca el proceso en la lista de PCB's para conseguir su estado de finalización.
  - Si el proceso estaba zombie, el PCB se libera y se devuelve el estado de finalización a su padre.
  - Si no estaba zombie, el proceso padre se elimina pasa de estado run a bloqued hasta que el proceso hijo termine y se aplicaría la política de planificación.

Protección y seguridad

- Físico → Poner las máquinas en habitaciones / edificios seguros.
- Humanos → Controlar quien accede al sistema
- SO
  - Evitar que un proceso sature el sistema
  - Asegurar que determinados servicios funcionen
  - Asegurar que ciertos puertos de acceso no están operativos
  - Controlar que los procesos no se salgan de su espacio de direcciones.
- RED → Es el más atacado

Tema 3 – Memoria

Gestión de memoria → La CPU sólo puede acceder a memoria y registros → Las instrucciones y datos deben cargarse en memoria para poder referenciarse, para ello se reserva memoria, se escribe en ella el programa y se ejecuta desde el principio.

- @ procesador → Conjunto de direcciones que el procesador puede emitir
- Memoria lógica → **@ lógicas** → Conjunto de @ lógicas que un proceso puede referenciar (el kernel las elige cuales son válidas).
- Memoria física → **@ físicas** → Conjunto de @ físicas asociadas al espacio de direcciones lógicas del proceso (el kernel decide cuales se referencian).

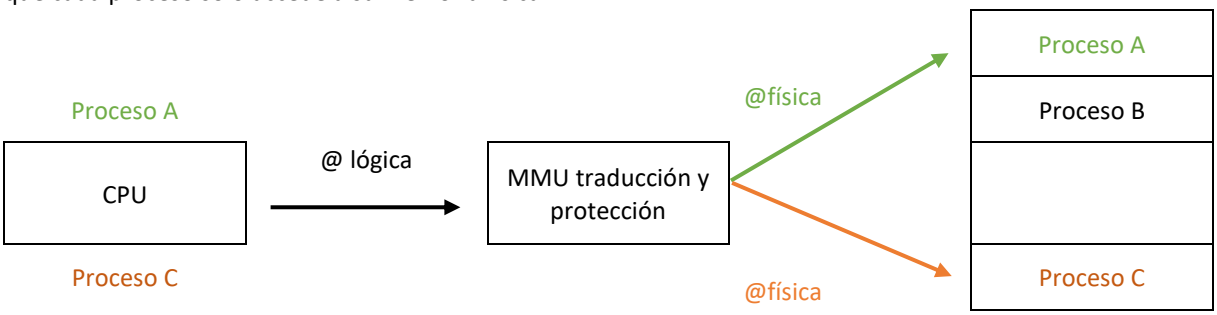
Espacio de @ lógicas = Espacio de @ físicas, ya que las lógicas referencian a las físicas

MMU (Memory Management Unit) → es el mecanismo de traducción y protección de acceso en memoria del HW, el kernel lo configura para cargar el programa en memoria.

1- Si las @ lógicas son válidas → Usa su estructura de datos para traducir y asignar la @ física que le corresponde y se asegura de que el proceso solo acceda a sus @ físicas asociadas.
2- Si la @ lógica no es válida o no tiene @ física asociada → Genera una excepción para avisar al SO.

El SO tiene que **modificar la traducción** de direcciones cuando se asigna una nueva memoria (execlp), cuando aumenta/disminuye el espacio o al pedir/liberar memoria dinámica, y tiene que **modificar la protección** cuando las @ lógicas son invalidas, con acceso incorrecto y con acceso incorrecto porque el SO hace alguna optimización.

Sistemas multiprogramados → Facilita la ejecución concurrente al haber 1 proceso en la CPU y N procesos en la memoria física esperando para hacer el cambio de contexto y poder ejecutarse, simplificándolo ya que no hay que cargar de nuevo la memoria, pero al hacer el cambio el SO debe actualizar la MMU (que detecta los accesos ilegales) con la información del proceso (aumento del espacio, etc), para que el SO pueda garantizar que cada proceso solo accede a su memoria física.

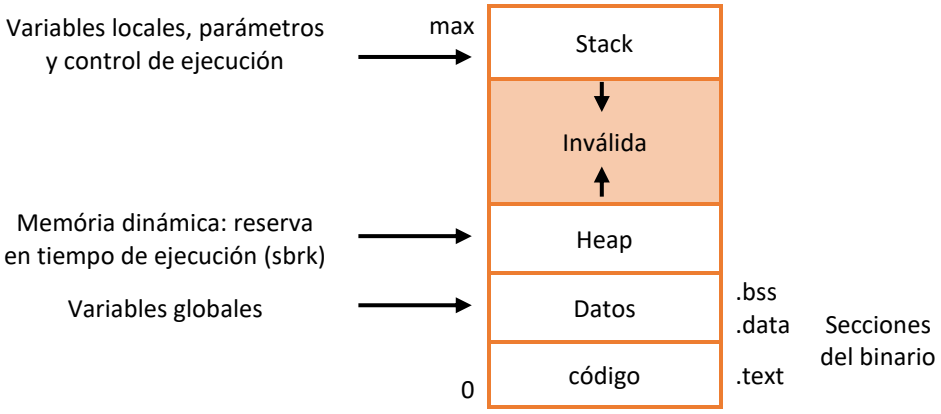


Carga de programas

1- Lee e interpreta el ejecutable.

Formato del ejecutable	
.text	**Código**
.data	**Datos globales inicializados**
.bss	**Datos globales sin valor inicial**
.debug	Información del debug
.comment	Información de control
.dynamic	Información para enlace dinamico
.init	@ de la 1ª instrucción

2- Prepara el esquema del proceso en memoria lógica y asigna memoria física:
   a. Escoge las @ lógicas válidas y el tipo de acceso.
   b. Prepara la información para configurar la MMU para el cambio de contexto y la inicializa.



3- Lee las secciones del programa.
4- Carga el PC con el punto de entrada del programa.
5- Optimizaciones aplicadas a la carga de programas

a. Carga bajo demanda
   I. Una rutina no se carga hasta que se llama
   II. Se aprovecha mejor la memoria al no cargarse funciones que no se usan
   III. Para saber si una rutina se ha cargado:
      i. SO registra en sus estructuras de datos que una zona de memoria es válida y dónde leer su contenido.
      ii. Cuando el proceso accede a la @ lógica, la MMU genera una excepción avisando al SO de una acceso a @ que no sabe traducir → El SO comprueba que el acceso es válido, carga la rutina y reanuda la ejecución

b. Librerías compartidas y enlace dinámico
   I. Los binarios (en disco) solo contienen un enlace a las librerías dinámicas, ahorrando espacio retrasando el enlace hasta el momento de ejecución.
   II. Los procesos (en memoria) pueden compartir la zona de memoria que contiene el código (de solo lectura) de las librerías comunes, ahorrando espacio en memoria.
   III. Facilita la actualización de librerías , sin tener que recompilar.
   IV. El binario contiene el código de una rutina de enlace (stub):
      i. Comprueba si una librería ya ha sido cargada y si no la carga.
      ii. Sustituye la llamada a sí misma por la llamada a la rutina de la librería compartida.

Memoria dinámica → Permite solicitar memoria en tiempo de ejecución, por lo que va solicitando memoria al SO según la necesite haciendo una llamada al sistema, reservando nuevas regiones en memoria (se almacena en la zona heap del espacio lógico de @).

```
// Permite modificar el límite del heap, el SO no sabe que variables están ubicadas en que
// zonas -> el programador ha de controlar la posición de cada variable en el heap
anterior = sbrk(nuevo * sizeof(int));
```

```
// Reserva memoria
//    Si hay espacio consecutivo lo reserva y no hace llamada al sistema ocupando lo que
//    se había reservado anteriormente
//    Si no hay espacio consecutivo aumenta el heap, reservando más de lo necesario para
//    reducir el número de llamadas al sistema
p = malloc(nuevo * sizeof(int));
```

```
// Libera memoria pasando a formar parte de zona libre o reduciendo el tamaño del heap
free(p);
```

La **diferencia** es que con sbrk siempre haces la llamada al sistema pidiendo el número exacto de espacio en memoria y con el malloc no siempre hace la llamada al sistema, depende de si anteriormente ya se ha hecho un malloc y hay espacio suficiente, entonces lo reserva, si no hace la llamada al sistema y aumenta el heap.

Errores comunes

```
...
for (i = 0; i < 10; i++)
ptr = malloc(SIZE);
// uso de la memoria
// ...
for (i = 0; i < 10; i++)
free(ptr);
...
```

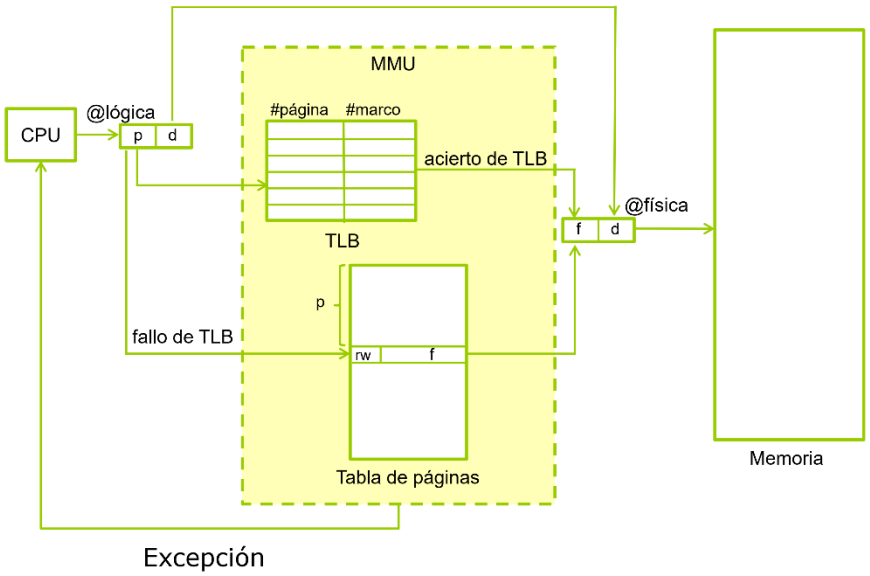
En la segunda iteración del bucle fallará porque hemos vaciado el `ptr` anteriormente, ptr será `NULL` y no podemos hacer `free(NULL)`

```
int *x, *ptr;
...
ptr = malloc(SIZE);
...
x = ptr;
...
free(ptr);
sprintf(buffer,"...%d", *x);
```

Hay 2 punteros apuntando al mismo sitio, por lo tanto, si liberas la memoria de uno, el otro apuntara a una posición de memoria no válida y producirá error.

Asignación de memoria

- **Asignación contigua**: espacio de @ físicas contiguo → todo el proceso ocupa una partición seleccionada en el momento de la carga, esto es poco flexible y dificulta aplicar optimizaciones.
- **Asignación no contigua**: espacio de @ físicas no contiguo → aumenta la flexibilidad, la granularidad de gestión de memoria de un proceso y la complejidad del SO y la MMU.
- **Paginación**: particiones fijas.



TLB (Translation Lookaside Buffer): Memoria asociativa (cache) de acceso más rápido en la que se almacena la información de traducción para las páginas activas.

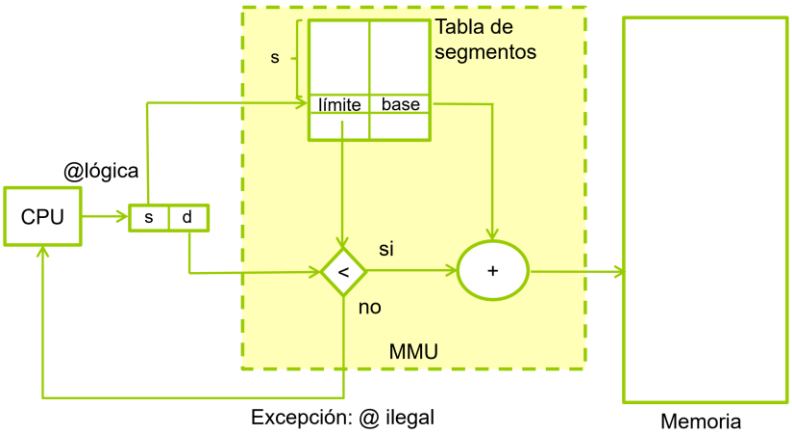
- Hay que actualizar/invalidar la TLB cuando hay un cambio en la MMU

TP (tabla de páginas)

El problema de las tablas de páginas es su tamaño, ya que están guardadas en memoria física. Tiene que ser una potencia de 2 (normalmente 4 Kb (2^12)). La TP está dividida en secciones, estas se añaden a medida que crece el espacio lógico de direcciones.

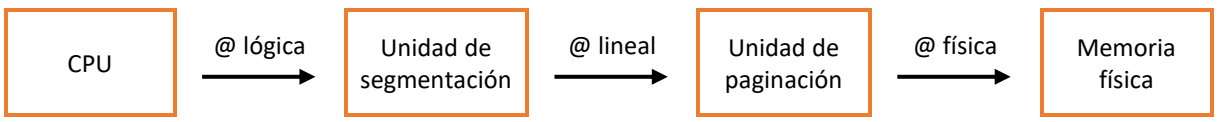
	Espacio lógico de procesador	Número de páginas	Tamaño TP
Bus de 32 bits	2^32	2^20	4MB
Bus de 64 bits	2^64	2^52	4PB

- **Segmentación**: particiones variables.



El problema es que puede haber fragmentación externa, no todos los trozos libres son igual de buenos.

- **Esquema mixto**



  - Espacio lógico del proceso dividido en segmentos
  - Segmentos divididos en páginas $\left(tamaño_{segmento} = n * tamaño_{página}\right)$
- Cuando hay problemas al asignar una cantidad X en memoria en una zona más grande → **Fragmentación:**
  - **Fragmentación interna**: memoria reservada a un proceso pero que no usa
  - **Fragmentación externa**: memoria libre pero que no se puede asignar por no estar contigua.

Compartición de memoria entre procesos: se puede especificar a nivel de página o de segmento:

- Librerías compartidas (implícito) → Procesos que ejecutan el mismo código no ocupan copias en MF.
- Memoria compartida como mecanismo de comunicación (explícito) → El SO proporciona llamadas al sistema para que un proceso cree zonas de memoria en su espacio lógico que sean compartibles y para que otro proceso la pueda mapear en su espacio de memoria.

Optimización del uso de memoria

COW (Copy on Write): intenta reduce la reserva/inicialización de memoria física hasta que sea necesario.

- Si no se accede a una zona nueva → no necesitamos reservarla realmente.
- Si no modificamos una zona que es una copia → no es necesario duplicarla en memoria.

Al hacer el fork

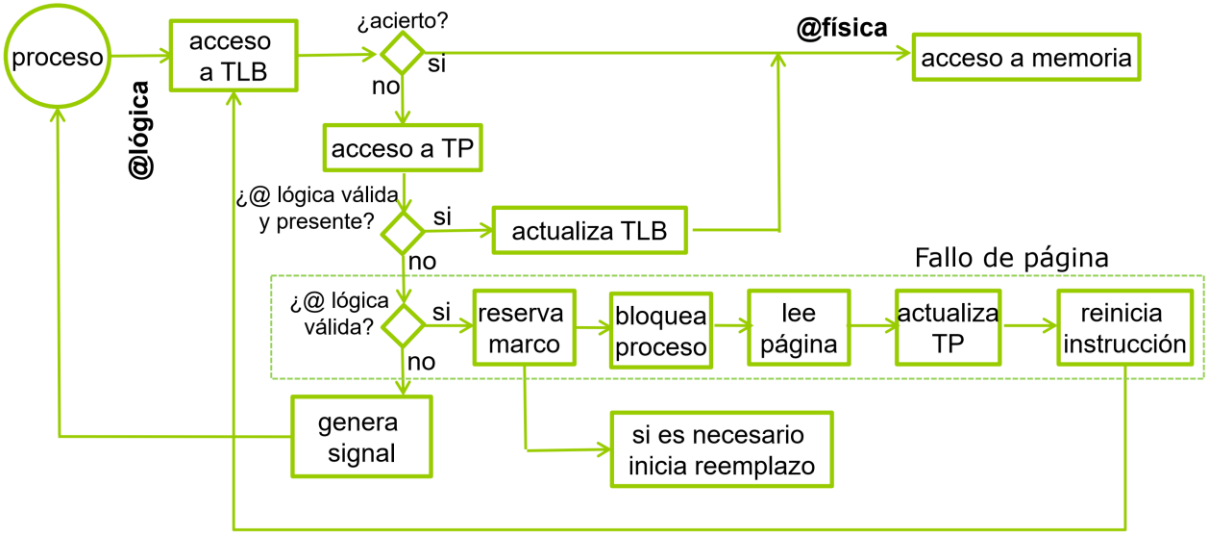
- Retrasa el momento de copia de código, datos, etc mientras sólo se acceda en modo lectura.
- Se gestiona a nivel de página lógica: se van reservando/copiando páginas a medida que las necesita.

Implementación: el kernel asume que se podrá ahorrar la reserva de la memoria física, pero de no ser así realiza la asignación:

- En la estructura de datos del kernel: se guardan los permisos reales.
- En la MMU se marca las regiones destino y fuente como permiso de solo lectura.
- En la MMU asocia la @ destino a las @ físicas: si es un fork a las regiones del padre, si es de memoria dinámica unas páginas que actúan como comodín.
- Si un proceso escribe en la zona nueva → la MMU genera excepción y el SO hace una reserva real reiniciando el acceso.

Memoria virtual: intenta reducir la cantidad de memoria física asignada al proceso de ejecución, un proceso realmente solo necesita memoria física para la instrucción actual y los datos que esa instrucción referencia.

La idea es que si el proceso activo necesita más memoria física que la disponible en el sistema → expulsar temporalmente de memoria alguno de los otros procesos cargados (swap out) → estos pasan a un almacen secundario, normalmente en disco (swap area), dónde están los procesos no residentes (swapped out) a la espera de que poder volverse a ejecutar (hay que volver a cargarlos en memoria) esto ralentiza la ejecución, por lo que se intenta evitar expulsar de memoria procesos enteros.



Problemas:

- La suma de espacios lógicos de los procesos en ejecución o el espacio lógico de un proceso puede ser mayor que la cantidad de memoria física.
- Acceder a una página no residente es más lento que acceder a una página residente:
  - Porque tiene que hacer excepción + carga de la página.
  - Es importante minimizar el número de fallos de página.

Modificaciones en el SO:

- Añadir estructuras de datos y algoritmos para gestionar el área de swap.
- Algoritmo de remplazo: intenta minimizar los fallos de página y acelerar su gestión liberando marcos.

Proceo en thrashing (sobrepaginación): invierte más tiempo en el intercambio de memoria que avanzando su ejecución, por lo que no consigue mantener simultáneamente en memoria el conjunto mínimo de páginas que necesita avanzar.

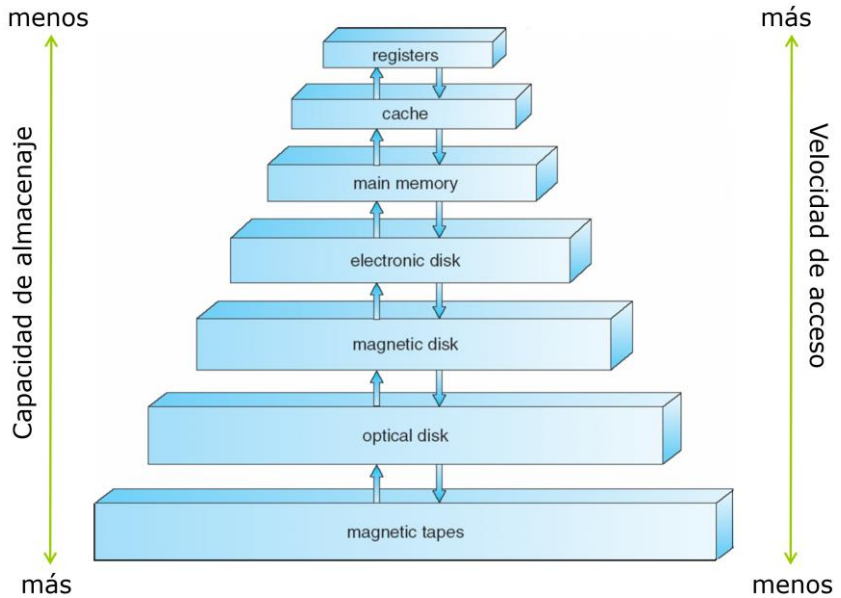
- Detección: controlar tasa de fallos de página por proceso.
- Tratamiento: controlar el número de procesos que se permiten cargar y parar procesos temporalmente (swap out).

Memoria prefetch: intenta minimizar el número de fallos de página, anticipando que páginas va a necesitar el proceso en el futuro inmediato y cargarlas con anticipación.

- Parámetros en cuenta:
  - Distancia de prefetch: con que antelación hay que cargar las páginas.
  - Número de páginas a cargar.
- Algoritmos sencillos de predicción de páginas:
  - Secuencial
  - Strided

Resumen (Linux sobre Pentinum)

- **exec** → inicialización del PCB con la descripción del nuevo espacio de direcciones, asignación de memoria…
- **fork** → espacio de direcciones y COW del padre y inicializa la Tabla de Páginas del nuevo proceso (del hijo).
- **Planificación** → se actualiza la tabla de páginas en la MMU en el cambio de contexto y se invalida la TLB
- **exit** → elimina la tabla de páginas del proceso y libera los marcos del proceso (si nadie más los usa).
- **Segmentación Paginada**
  - Tabla de páginas multinivel (2 niveles)
    - Una por proceso, guardadas en memoria y la CPU contiene la @ base de la TP del proceso.
  - Algoritmo de reemplazo: aproximación de LRU (Least Recently Used)
    - Se ejecuta cada cierto tiempo y cuando el número de marcos libres es menor a un umbral.
- Implementa **COW** a nivel de página.
- **Carga bajo demanda.**
- **Soporte para librerías compartidas.**
- **Prefetch simple** (secuencial).



Tema 4 – Entrada / Salida

La E/S es una transferencia de información entre un proceso y el exterior, de echo los procesos realizan cálculo y E/S, muchas veces la E/S es la principal tarea de un proceso.

El objetivo de los procesos es que sean **independientes** del dispositivo al que se esta accediendo, para ello tenemos un diseño de 3 niveles:

- **Nivel virtual**: aísla al usuario de la complejidad del dispositivo
  - Establece un vínculo entre un **nombre simbólico** (`dev/dispX` o `.../dispX nombre fichero`) y la aplicación de usuario, mediante un dispositivo virtual. Dispositivo virtual = canal = descripción del fichero (es un número entero), los procesos tienen 3 canales estándar:
    - Entrada estándar al canal 0 (stdin).
    - Salida estándar al canal 1 (stdout).
    - Salida error estándar canal 2 (stderr).
  - Las llamadas a sistema de transferencia de datos utilizan como identificador el dispositivo virtual.
- **Nivel lógico**: establece la correspondencia entre el dispositivo virtual y el físico (si es que tiene representación física).
  - Manipula bloques de datos de tamaño independiente.
  - Proporciona una interface uniforme a nivel físico.
  - Ofrece compartición (acceso concurrente) a dispositivos físicos que representan (si hay).
  - Este nivel tiene en cuenta los permisos.
  - **En Linux se identifican con un nombre de fichero**
- **Nivel físico**: Implementan a bajo nivel las operaciones de nivel lógico.
  - Traduce parámetros de nivel lógico a parámetros concretos.
  - Inicializa los dispositivos, comprueba si está libre y si no se pone en cola.
  - Realiza la programación de la operación deseada (Ej: hacer funcionar los motores de un disco).
  - Espera o no a la finalización de la operación.
  - Devuelve los resultados o informa sobre errores.
  - **En Linux se identifican con 3 parámetros:**
    - Tipo: **Block/Character.**
    - Con 2 nombres major/minor:
      - **Major:** indica el tipo de dispositivo.
      - **Minor:** instancia concreta respecto al major.

Device Drivers (DD): ofrecen independencia por parte del fabricante, ya que es un superconjunto de todas las operaciones que se pueden ofrecer para acceder a un dispositivo físico (no todos los dispositivos la ofrecen), de esta manera cuando se traduce de dispositivo virtual a lógico y físico están definidas que operaciones corresponden. Incluye:

- El código + datos para acceder a un dispositivo.
  - **Información general sobre el DD**: nombre, autor, licencia, descripción…
  - **Implementación de las funciones genéricas de acceso al dispositivo** (Respeta la interface definida por el SO): open, read, write… (dependiendo del SO)
  - **Implementación de las funciones especificas de acceso a los dispositivos** (Habitualmente contienen código de bajo nivel): acceso a puertos E/S, interrupciones…
  - **Estructura de datos con lista de apuntadores a las funciones específicas**
  - **Función de inicialización:**
  - Se ejecuta al instalar el DD
  - Registra el DD en el sistema, asociándolo a un major
  - Asocia las funciones genéricas al DD registrado.
  - **Funciones de desinstalación:** desregistra el DD del sistema y las funciones asociadas
  - Quedan encapsuladas en un archivo binario.
- Siguen las especificaciones de interface de acceso a las operaciones E/S definidas por el SO. Hay 2 maneras de añadir un nuevo dispositivo:
  - Recompilar el kernel, esto puede tardar horas.

- No recompilar el kernel, el SO incluye un mecanismo para añadir dinámicamente código/datos al kernel. **Dynamic Kernel Module Support (Linux) o Plug&Play (windows)**. Para añadir un nuevo dispositivo:
  - **Compilar** el DD, si hace falta, en un formato determinado: .ko (kernel object)
  - **Instalar** (insertar) en tiempo de ejecución las rutinas del driver
  - **Crear un dispositivo lógico** (nombre archivo) y vincularlo con el dispositivo físico: **comando** `mknod /dev/mydisp c major minor`
  - **Crear el dispositivo virtual (llamada al sistema)**: `open("/dev/mydisp", …)`

Ejemplo de dispositivos:

- **Terminal:** objeto a nivel lógico que representa el conjunto teclado + pantalla.
- **Fichero de datos:** Objeto a nivel lógico que representa información almacenada en disco. Se interpreta como una secuencia de bytes y el sistema gestiona la posición dónde encontramos la secuencia.
- **Pipe**: Objeto lógico que implementa un buffer temporal con funcionamiento FIFO. Los datos de la pipe se eliminan a medida que se van leyendo. Sirven para intercambiar información entre procesos.
  - **Pipe sin nombre**: conecta solo los procesos con parentesco ya que es accesible sólo vía herencia.
  - **Pipe con nombre**: permite conectar cualquier proceso que tenga permiso para acceder al dispositivo.
- **Socket**: objeto a nivel lógico que implementa un buffer temporal con funcionamiento FIFO. Sirve para intercambiar información entre procesos que se encuentran en diferentes computadores conectados por una red. Su funcionamiento es similar a los pipes, pero internamente más complejo ya que usa la red.

Tipos de ficheros:

- **Ordinary files**: ficheros en el sistema que contienen datos, texto o instrucciones de un programa.
- **Special files**: ficheros que permiten el acceso a dispositivos hardware o los ficheros parecidos a accesos directos que permiten el acceso a un mismo fichero usando diferentes nombres.

`mknod nombre_fichero TIPO major minor:`

- TIPO = c → Dispositivo de caracteres
- TIPO = b → Dispositivo de bloques
- TIPO = p → PIPE (no hace falta poner major/minor)

Estructuras de datos del kernel: Inodo: contiene toda la información de un archivo excepto el nombre que esta por separado. (Tamaño, tipo, protecciones, propietario, fechas, número de enlaces al inodo, etc)

- Por proceso:
  - Tabla de canales (TC): para cada proceso (se guarda en la PCB): indica a que archivos estamos accediendo mediante el canal (un índice a la TC), este es un dispositivo virtual, que referencia a una entrada de la TFO. Para ello usaremos el campo: `num_entrada_TFO`.
- Global:
  - Tabla de archivos abiertos (TFO): Gestiona los ficheros en uso en todo el sistema (pueden haber entradas compartidas entre más de un proceso y para varias entradas del mismo proceso), una entrada de la TFO referencia a una entrada de la TI. Se usan los campos: `num_referencias, modo, puntero, num_entrada_TI.`
  - Tabla de inodos (TI): contiene información sobre cada objeto físico abierto, incluyendo las rutinas del DD. Es una copia en memoria de los datos en disco, para mayor eficiencia. Los campos asumidos son: num_referencias, datos_inodo.

Operaciones E/S bloqueantes: el proceso realiza una transferencia de N bytes esperando a que esta acabe y al finalizar devuelve el número de bytes transferidos.

- Si los datos están disponibles en el momento, aunque no sean todos → se realiza la transferencia.
- Si no están disponibles → El proceso se bloquea
  1- Cambia de estado RUN a BLOCKED (deja la CPU y pasa a los procesos en espera)
  2- Se pone en ejecución el primer proceso de la cola de preparados (Si Round Robin)
  3- Cuando disponemos de los datos → Se produce una interrupción recogiendo el dato y poniendo el proceso a la cola de preparados (SI Round Robin)
  4- Cuando le toque el turno, el proceso se pondrá de nuevo en ejecución.

Operaciones E/S no bloqueantes: El proceso solicita una transferencia, envía/recibe los datos de los que disponga en ese momento y devuelve el número de datos transferidos, tanto si hay como si no.

Operaciones básicas E/S:

```
// Pasa el nombre del archivo y devuelve dispositivo virtual (fd: file descriptor o canal)
// Estructuras de datos:
//       Ocupa el canal libre de la TC (el primero disponible)
//       Ocupa una nueva entrada de la TFO - Posición L/E = 0
//       Asocia estas estructuras al DD correspondiente (major del nombre simbólico)
int fd = open(name, access_mode[,permision_flags]);
access_mode:
- O_RDONLY        // Lectura
- O_WRONLY        // Escritura
- O_RDWR          // Lectura y escritura
- O_CREAT         // Crear el archivo de datos (OR bits)
- O_TRUNC         // Para borrar los datos completamente (no se pueden borrar parcialmente)
permision_flags:  // (OR bits)
- S_IRWXU  // Permisos de escritura y lectura de bits para el propietario del archivo
- S_IRUSR  // Permisos de lectura de bits para el propietario del archivo
- S_IWUSR  // Permisos de escritura de bits para el propietario del archivo

// Ejemplo - Crear el archivo "X" si no existía, si existía no afecta
open(“X”, O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
// Ejemplo - Si existía se liberan sus datos y cambia el tamaño a 0 bytes, si no crea el
archivo "X"
open(“X”, O_RDWR|O_CREAT|O_TRUNC, S_IRWXU);

// Lee un número de bytes de un canal y los guarda en memoria.
int n = read(fd, buffer, count);

// Lee un número de bytes de memoria y escribe en el dispositivo indicado
int n = write(fd, buffer, count);
// n = número de bytes leídos
// fd = canal
// buffer = puntero de memoria dónde dejar los bytes leídos
// count = número de bytes que leer:
//       Si hay >= count bytes disponibles -> leerá/escribirá count bytes
//       Si hay < count bytes disponibles -> leerá/escribirá los que haya
//       Si no hay datos disponibles / no hay espacio disponible ->
//                Devuelve EOF (no ha leído/escrito ningún byte/carácter)
//                Se bloquea a la espera a que haya (Depende del dispositivo)
// La posición de L/E avanza automáticamente tantas posiciones como bytes leídos/escritos

// Duplica un canal, ocupa primer canal libre y copia la información del canal fd
int newfd = dup(fd);

// Duplica el canal, pero especificando el canal en la TC (newdf)
int newfd = dup2(fd, newfd);

// Libera el canal fd y las estructuras asociadas a niveles inferiores
close(fd);

// Crea pipe con nombre
mknod p nombre;
open(nombre, R);   // Crear lectura y escritura por separado
open(nombre, W);
// Estructuras de datos:
//       Se crean dos entradas a la TC (R/W)
```

```
//       Se crean dos entradas a la TFO (R/W)
//       Se crea una entrada temporal a la TI
// Crea una pipe sin nombre
int fd_vector[2];  // Creamos un vector de tamaño 2
pipe(fd_vector);

// Se bloquea el proceso hasta que haya datos en la pipe (aunque no sea la cantidad pedida)
// Si no hay proceso que escriba y la pipe está vacía -> provoca EOF
int canal_lectura = fd_vector[0];
// El proceso escribe a no ser que la pipe esté llena -> se bloquea hasta que se vacíe
// Si no hay lector -> provoca la excepción SIGPIPE
int canal_escritura = fd_vector[1];

// Permite modificar la posición de lectura/escritura manualmente
int nueva_posicion = lseek(fd, desplazamiento, relativo_a);
relativo_a:       // (el desplazamiento puede ser un valor negativo)
- SEEK_SET: posicion_LE = desplazamiento
- SEEK_CUR: posicion_LE = posicion_LE + desplazamiento
- SEEK_END: posicion_LE = posicion_LE – desplazamiento
```

E/S y fork:

- El hijo copia la tabla TC, pero esta apunta los mismos sitios en la TFO
- Permite compartir los accesos a los dispositivos abiertos antes de hacer el fork

E/S y exec:

- Se mantienen las mismas estructuras internas de E/S del proceso
- Al hacer fork+exec permite hacer redirecciones antes de cambiar de imagen

Si mientras se está ejecutando una operación E/S se interrumpe por un signal:
- Se trata el signal y la operación E/S sigue (si `struct sig_action sa; sa.sa_flags = SA_RESTART;`).
- Se trata el signal y la operación E/S devuelve error → `errno == EINTR` (si `sa.sa_flags != SA_RESTART;`).

Sistema de ficheros: gestiona los ficheros y almacena los datos

- Organiza los ficheros del sistema
- Garantiza el acceso correcto a los ficheros (permisos de acceso)
- Gestiona (asigna/libera) espacio libre/ocupado del fichero de datos.
- Encuentra/almacena los datos de los ficheros de datos.

Directorio: es una estructura lógica que organiza los ficheros. Es un fichero especial gestionado por el SO que permite enlazar los atributos de los ficheros (tipo de fichero, tamaño, propietario, permisos…). Estos se organizan jerárquicamente (formando un grafo) que permite al usuario clasificar sus datos. Todo directorio tiene como mínimo dos ficheros especiales:

- **.** Referencia el actual directorio
- **..** Referencia el directorio padre

Cada fichero se puede referenciar de dos maneras:

- Nombre absoluto (único): Camino des de la raíz + nombre (Ej: /home/usr1/file)
- Nombre relativo: comino des del directorio de trabajo + nombre (Ej: file)

Como el nombre de fichero está separado de la información (inodo), Linux permite referenciar un inodo por más de un nombre de fichero. Hay 2 tipos de vínculos:

- **Hard-link**: La forma más habitual de nombre de fichero que referencia directamente al nombre de inodo dónde están los atributos + la información de datos. El inodo tiene un número de referencias que indica cuantos nombres de ficheros apuntan a él, pero es diferente al que hay en la tabla de inodos.

  No se permiten ciclos con hard-links en la estructura del directorio (el grafo).

- **Soft-link**: el nombre de fichero no apunta directamente a la información, si no que apunta al inodo que contiene el nombre del fichero destino

  Se permiten ciclos con hard-links en la estructura del directorio (el grafo).

Dado que el sistema va haciendo Backups y no hace copias de seguridad del mismo fichero dos veces → el Al eliminar un fichero del sistema no comprueba si han soft-links de un fichero y los hard-links se eliminan si el contador de referencias llega a 0.

Servicio	**Llamada al sistema**
Crear/eliminar enlace a fichero / softlink	Link/unlink symlink
Cambia permisos de un fichero	Chmod
Cambiar propietario/grupo de un fichero	chown / chgrp
Obtener información del estado del inodo	stat, lstat, fstat

Unidad de trabajo del disco

Un dispositivo de almacenamiento está dividido en partes llamadas sectores y la unidad de asignamiento del SO es el bloque → 1 bloque = 1 o más sectores

Una **partición** o volumen o sistema de ficheros es un conjunto de sectores consecutivos con identificador único y identidad propia y se gestionan por el SO como entidad lógica independiente. Cada partición tiene su propia estructura de directorios y ficheros independiente al resto de particiones.

El SO para saber que bloques son los suyos existe una tabla de índices: bloques asignados a un fichero.

El **espacio libre** se sabe mediante una lista de bloques libres y una lista de inodos libres, cuando se necesita un nuevo inodo (creación de un nuevo fichero) o un nuevo bloque (para aumentar el tamaño de un fichero) se coge el primero de la lista correspondiente.

Los **metadatos** persistentes se guardan en disco (inodos y lista de bloques de un fichero, directorios, lista de bloques libres, lista de inodos libres y la información necesaria para el sistema de ficheros), pero cuando estos metadatos están en uso se guardan en memoria (por eficiencia).

Superbloque: bloque de la partición que contiene los metadatos del sistema de ficheros (tolerante con fallos)

Metadatos y memoria:

- Zona de memoria para guardar los últimos inodos leídos.
- Zona de memoria para guardar los últimos directorios leídos.
- Buffer cache: zona de memoria para guardar los últimos bloques leídos.
- Superbloque: se guarda también en memoria.

Relación entre llamadas a sistema y estructuras de datos

- Open:
    - Localiza el inodo del fichero con el que se quiere trabajar y lo deja en memoria.
        - El directorio está en la cache de directorios → Se accede a el.
        - El directorio está en disco → Se lee en disco y se deja en memoria.
    - Si localiza el inodo:
        - Comprueba si el acceso es correcto (permisos) → modifica la tabla de canales, tabla de ficheros abiertos y la tabla de inodos.
        - Si no es correcto → devuelve error.
        - Si es un soft-link lee el path del fichero al que apunta y localiza el inodo correspondiente.
    - Si no localiza el inodo → Error al intentar acceder a un archivo que no existe.
    - Si utilizamos el open para crear un nuevo fichero → se ha de reservar e inicializar un nuevo nodo → hay que actualizar la lista de nodos libres en el superbloque y actualizar el directorio dónde se crea el fichero.
- Read:
    - Tenemos el inodo correspondiente de la TI.
    - Tenemos el puntero lectura/escritura de la TFO.
    - Calculamos los bloques involucrados:
        - Comprobamos no estar al final del fichero (valor puntero == tamaño fichero).
        - Obtenemos el primero bloque que leemos (valor puntero / tamaño bloque) y bloques que hay que leer de más (parámetro del tamaño syscall).
    - Si los bloques ya se habían usado antes → Los encontramos en memoria, si no en disco (y se dejan en memoria).
- Write: igual que el read, excepto que si escribimos al final del fichero → Es posible tener que añadir más bloques a éste → Modificamos la lista de bloques libres del superbloque y actualizamos el inodo del fichero con los nuevos bloques y el nuevo tamaño.
- Close: provoca la actualización del inodo con los cambios que se han añadido (como mínimo la fecha del último acceso).