uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Informática

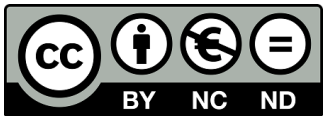
2019-2020

*Apuntes*

# Sistemas Operativos

---

Jorge Rodríguez Fraile[1]





[1]Universidad: 100405951@alumnos.uc3m.es | Personal: jrf1616@gmail.com

# ÍNDICE GENERAL

# 1. TEMA 1

## 1.1. Tema 1.1: Introducción a los Sistemas Operativos.

La función del sistema operativo es facilitar al usuario el uso del computador, antes solo se utilizaba para controlar y comunicarse con el hardware. Todas las funcionalidades y aplicaciones están sobre el sistema operativo.

**Porque saber sobre SS. OO**:

- Influye en el funcionamiento, seguridad y rendimiento del computador. Por ello es importante conocer cómo funciona.
- Elegir el que mejor se adapte a nuestras necesidades o de la empresa es fundamental.
  - Lo más importante, que sea rentable. Hay que proteger la inversión.
  - Encontrar administradores que sepan sobre el sistema.
  - Que ofrezca soporte y se actualice.
- Al desarrollar aplicaciones saber como se comportará y comprender los problemas.
  - Conocer los servicios que ofrece y como invocarlos. Y el desarrollo multi-hilo.

No tener ideas preconcebidas de los SS. OO, “ser agnóstico”, utilizar el que mejor se adapte a lo que necesito, hacer una checklist y ver cuál cumple más.

LECTURA CAPÍTULO 1, libro CARRETERO 2007. Conceptos arquitectónicos del computador.

**Sistema Operativo:** Programa intermediario entre el usuario y el hardware. Oculta el hardware, pero lo controla para ofrecer servicios al usuario, el usuario y hardware son muy distintos.

- Ejecuta programas.
- Hace uso eficiente de recursos.
- Proporciona una visión de máquina virtual.

- **Funciones del Sistema Operativo:**
  - **Gestor de recursos**:
    - Asigna y recupera recursos, como es la memoria.
    - Protege al usuario, controla el acceso y los permisos de los usuarios.
    - Contabiliza y monitoriza los recursos del sistema para ver como esta y que le pasa.
  - **Máquina extendida (servicios):**
    - Ejecución de programas.
    - Órdenes de E/S, leer/escribir, abrir/cerrar.
    - Operación sobre archivos.
    - Detección y tratamiento de errores, para que se reconozca y resuelvan.
  - **Interfaz de usuario:**
    - Shell, es una interfaz gráfica externa al sistema operativo.
- **Niveles del Sistema Operativo**, los 3 principales:
  - Usuario.
  - Shell.
  - Llamadas al sistema o Servicios.
  - Núcleo/Kernel. Se ha ido reduciendo el Kernel y ampliando los Servicios
  - Firmware/HardwareAbstractLogic, entre Kernel y Hardware.
  - Hardware.
- **Modos de ejecución**:
  - Modo **usuario**: Está restringido, no puede acceder a ciertas aplicaciones, E/S, etc. Para proporcionar más seguridad.
  - Modo **núcleo**: Puede ejecutar cualquier aplicación, sin restricciones. Más "peligro".
- Los procesos y el sistema operativo, tienen espacios memoria reservada separados, y el del sistema operativo no es accesible para el usuario.
- Cuando un proceso necesita un servicio, se hace con una llamada al sistema, y el sistema lleva a cabo la función de la llamada.
- **Modificar el sistema operativo:** Principalmente para añadir funcionalidades o nuevo **hardware**. Pero se deben seguir unos estándares para asegurar que las aplicaciones funcionen en sistemas que sigan el mismo estándar.

**Sistemas operativos monolíticos**: No hay estructura clara y bien definida, pero es más eficiente, aunque su mantenimiento es más difícil. Todos los módulos se pueden llamar entre ellos. Siempre modo núcleo y todo enlazado en un ejecutable.

**Sistemas operativos estructurados por capas:** Organizado por capas superpuestas, con una interfaz clara y bien definida. Una capa se apoya en la inferior, resulta menos eficiente, pero todo está más restringido y es más fácil el desarrollo y depuración.

**Sistemas operativos estructurados cliente/servidor:** Consiste en implementar la mayor parte del sistema operativo como procesos de usuario, y una pequeña parte corre en modo núcleo llamado microkernel. Muy flexible, ya que funciona como pequeños programas.

**Clasificación de Sistemas Operativos:**

- Número de procesos simultáneos:
  - Monotarea: Un solo proceso a la vez.
  - Multitarea: Varios procesos a la vez, necesita planificación.
- Modo de interacción:
  - Interactivo: Permite ejecutar procesos y obtenemos la solución en el momento.
  - Por lotes: Tú pides al sistema y no sabes cuando se hará.
- Número de usuarios simultáneos:
  - Monousuario.
  - Multiusuario.
- Numero de procesadores:
  - Monoprocesador.
  - Multiprocesador: Son actualmente casi todos. Multicore.
- Número de hilos:
  - Monothread.
  - Multithread.
- Tipo de uso:
  - Cliente: Pedir procesos.
  - Servidor: Encargarse de los procesos.
  - Empotrados: Dentro de un sistema cerrado, con unas funciones determinadas.
  - Tiempo real: Se usan otros sistemas operativos para gestionarlos, otra planificación.

**Arranque del sistema operativo**:

- Comienza con el envío de la interrupción de RESET (la 0), que carga el cargador (booter) y este carga la BIOS (Basic Input Output System) en memoria, la BIOS hace una comprobación del hardware (los periféricos, los discos, la memoria...) y cuando termina comienza a cargar en memoria el sistema operativo del disco que sabe que lo almacenaba. Una vez cargado el sistema operativo, se empieza a ejecutar, lo instala todo, y cuando ha terminado el sistema ya está ejecutado y se puede liberar la memoria que ocupaba para el usuario.

    - Reset, pone los valores predefinidos a los registros y carga el cargador/booter.
    - Se carga la BIOS, hace comprobaciones y cargar el sistema operativo en memoria.
    - Se ejecuta el cargador del sistema operativo, que empieza a instalar el sistema y cuando termina el sistema ya está ejecutado, y se libera la memoria del cargador.

**Parada del computador (Shutdown):** Procesos inversos al arranque.

- Al apagar, se vuelca la información a disco y se terminan todos los procesos.
- Si hay apagado brusco, se pierde información y se puede dañar, para evitarlo hay una pequeña batería que permite que se apague correctamente. Para sistemas grandes SAI.
- Otras alternativas son: Hibernación y Apagado en espera(Standby).

## 1.2. Tema 1.2: Servicios del SO.

- **Ejecución del sistema operativo:** Una vez finalizado el arranque del sistema, el sistema solo se ejecuta como respuesta a interrupciones. Como son:

    - Petición de servicio, están sobre el sistema operativo y no afectan a la máquina. lpd d=daemon=demonio
    - Interrupción de periféricos o reloj.
    - Excepción de hardware.

- **Fases en la activación del Sistema Operativo:** Lo primero que se ejecuta es el proceso init y a raíz de este van apareciendo más procesos que cuelgan de el, como si fuera un árbol del que van saliendo ramas. Al cambiar de proceso debe encontrarse tal y como estaba antes del cambio, eso se llama contexto. Antes de cambiar se almacena todo del proceso actual e inmediatamente después se cargan los datos

del otro. Todo proceso tiene su contexto. Se aprovechan los tiempos de espera(de cualquier proceso) para cambiar de proceso y ejecutar mientras tanto otras cosas, estos cambios los controla el planificador y es concurrencia.

- **Activación de servicios:** Lo primero es pasar de modo usuario a modo núcleo, en la llamada el sistema tiene control total de la máquina. Ese cambio de privilegios es gracias a las bibliotecas que tienen acceso a superusuario. Y esos privilegios pueden ocasionar problemas de seguridad. Con la interrupción software trap se activa el modo seguro del sistema operativo.

  - La rutina de bibliotecas es: Preparar la llamada al SO, instrucción trap y procesar los datos recibidos de la llamada.

- **Llamadas al sistema**:

  - Interfaz entre aplicaciones y SO.
  - Servicios típicos
    - Gestión de procesos.
    - Gestión de procesos ligeros: Si se crea, que se pueda controlar y matar.
    - Gestión de señales y temporizador: Avisos y automatizaciones.
    - Gestión de memoria.
    - Gestión de ficheros y directorios.

- **Bibliotecas**: Agrupación de funciones completas

- **Invocación de la llamada:** Cada función de la interfaz de programación(API) se corresponde con algún servicio del sistema operativo. Incluye la ejecución de trap que transfiere el control al sistema operativo con una interrupción, el sistema operativo trata la interrupción y devuelve el control al programa de usuario.

  - Salta a la librería, comprueba y almacena los parámetros necesarios para la operación, se pueden pasar en registros, tabla de memoria o en pila. Pone el número de operación, el identificador numérico correspondiente y hace syscall (trap), para llamar al sistema a ejecutar esa operación.
  - El SO ejecuta internamente la operación, recibe el identificador que le indica la dirección de la rutina de tratamiento y la ejecuta, y devuelve 0 si ha funcionado o -1 si ha fallado.
  - Resuelta la operación, vuelve al código del usuario con el valor esperado.

- **Interfaz del programador:** Ofrece una visión que como máquina extendida tiene el usuario del sistema, cada sistema operativo puede ofrecer uno o varias.

- **Estándar POSIX (Portable Operative System Interface for X)**: Interfaz estándar de sistema operativo de IEEE. El objetivo es que todas la maquinas que compartan este estándar puedan ejecutar los mismos programas, portabilidad.

    - Nombres de funciones cortos y en letras minúsculas, representativos y en ingles.
    - Normalmente el éxito devuelve 0 y en caso de error -1. errno=error number y perror(errno) nos dice que error es y nos proporciona algo de información.

- **Ejecución de un mandato en Shell**: Lo que hace cada vez, hay una relación jerárquica de procesos shell, que permite que en caso de error el proceso shell padre no muera.

    - Lo primero el padre ejecuta fork(), que crea un duplicado del padre que funcionará a la vez que el padre y ese hijo se identifica por un pid(process id) de 0 y el padre tendrá otro cualquiera. Cuando se ejecuta fork el hijo siempre es pid=0.
    - Para que sea el hijo el que ejecute el mandato y corra el peligro de error, y no el padre. Lo que se hace es que el proceso con ID 0, el hijo, ejecute el mandato con execvp(,) y si da error ese proceso hijo muere y queda vivo el padre, pero si no falla ambos seguirán vivos hasta el final del programa. Se hace un switch, si pid del hijo es -1 ha habido un problema al crearlo y termina, si es 0 ejecuta el mandato y break y si no es ninguno de los anteriores es el padre.

- **Servicio fork()**: Devuelve un pid_t que es el ID del proceso, en el padre el valor es el identificador del hijo y en el hijo el valor es 0, y -1 en caso de fallo. Su función es duplicar el proceso que invoca la llamada, tanto el proceso hijo como el padre siguen ejecutando el mismo programa. El proceso hijo conservará una copia exactamente igual de todo lo que tenía el padre.

- **Servicio wait(&status):** Bloquea la ejecución del padre hasta que el proceso hijo muere, devuelve el identificador del proceso que ha finalizado, y status es el valor que devuelve el hijo cuando llama al exit().

- **Servicio exec:** Ejecuta un programa en ese proceso, el proceso recibe toda la información y cambia su imagen, desaparece el código que tiene y se sustituye por el del programa si ha ido bien, si no retorna -1. Cuando se hace exec la tabla de ficheros abiertos no se borra. Servicio único para múltiples funciones de biblioteca. Recibe como primer argumento el nombre de la función a ejecutar y el resto son los parámetros del main. execlp('ls','ls','-l')

- **Servicio exit (void exit(status)):** Finaliza la ejecución del proceso, se escribe al final de un programa o al detectar un error. Se cierran todos los descriptores de ficheros abiertos y se liberan todos lo recursos del proceso. Devolver negativo es indicativo de error y 0 de que la ejecución ha ido correctamente, pero también puede tomar otros valores. Todos lo valores que toma status sirven para dar información adicional de como ha ido la ejecución. return 0=exit(0)
- **Fichero**, es una manera de almacenar información de una manera más persistente, para reutilizarlos o simplemente para tenerlos.
- **Operaciones genéricas sobre ficheros**: AMPLIAR
  - Crear: creat(nombre, O_WRONLY|O_CREAT|O_TRUNC, mode);
  - Borrar: unlink(nombre);
  - Abrir: open(descriptor, permisos); O_RDWR/WRONLY/RDONLY/APPEND sigue al final del fichero.
  - Cerrar: close(descriptor);
  - Leer: read(descriptorLectura, buffer, tamañoALeer);
  - Escribir: write(descriptorEscritura, buffer, longitudAEscribir);
  - Posicionar: lseek(descriptor, desplazamiento, dondeComenzar); SEEK_SET/END/CUR
  - Control: fnctl para modificación de atributos, dup para duplicar descriptores de fichero, ftruncade asignación de espacio a un fichero, stat información sobre un fichero, utime para alterar los atributos de fecha.

# 2. TEMA 2

## 2.1. Tema 2.1: Introducción a la gestión de Procesos.

**Proceso:** Programa en ejecución, cada ejecución da lugar un proceso nuevo. Cada proceso tiene un identificador único llamado pid(process id), que nos permite obtener más información sobre el proceso, como el contexto, directorio, padre, etc.

- Está formado por: Código del programa(instrucciones) y los datos asociados a la propia ejecución del programa.
- **Partes**: Pila(se va ampliando hacia abajo), memoria dinámica, datos(se amplía hacia arriba) y texto(el código). Cada proceso tiene esta estructura y son independientes entre ellos.

**Ciclo de vida de un proceso:**

- **Listo**: Cuando se ejecuta un programa se hace el exec, lo primero que hace es comprobar sus propiedades y si se tiene recursos para ejecutar ese proceso nuevo. Si se tienen recursos se creará ese proceso nuevo y se le da un pid propio, el pid debe estar por debajo del límite máximo de procesos o no se podrá crear. No se reutilizan pid's, habría que reiniciar.
- **En ejecución:** Cuando está listo, se pasa a memoria y se ejecuta, si es para algo lento lo pasaremos ha bloqueado, para que espere a recibir los datos que necesite, y pasara a listo, si el proceso ha superado su tiempo asignado o si cede su CPU. Este último lo controla el planificador. El proceso puede terminar con exit, que liberará los recursos o si no, esperará un tiempo antes de liberarlo.
- **Bloqueado**: Cuando el proceso está esperando a recibir datos, si lo recibe pasa a listo, pero si no recibe los datos en ningún momento se quedará en bloqueado.

**Modelo de colas simplificado: Un procesador**.

- Los procesos están en una cola, y según las unidades de procesado que necesita va a una parte u otra, hay una para procesos rápidos y otras para procesos más pesados. Si un proceso en su franja de tiempo no ha terminado, se vuelve a poner en la cola. El que elige donde va cada proceso es el dispatcher. (Función como la cola de un supermercado)

**Información de un proceso**: Toda la información que permite la correcta ejecución del proceso.

- **Tres categorías**:
  - Información almacenada en el procesador.
  - Información almacenada en memoria.
  - Información adicional gestionada por el sistema operativo.
- El pid se almacena en la tabla de procesos (TP). Cada proceso tiene su BCP, donde se almacena su entorno(no cambia), mapa de memoria(texto y datos) y el contexto/estado(puede cambiar).
- En el estado/contexto se incluye los valores de los registros accesibles y no accesibles para el usuario del procesador. Cuando se cambia de contexto, se salvaguarda el estado del procesador del saliente y se restaura el estado del proceso del entrante.
- **Memoria virtual**: Se engaña al sistema, ya que se necesitan recursos que no tiene la máquina y puede que tenga que matar a otro proceso para ir haciendo hueco. Mem. dinam.
- **Modelo de imagen de memoria con Regiones múltiples:**
  - El número de regiones es fijo, y cada región es de tamaño variable.
  - Las regiones son: Pila, Datos y Texto.
  - Con memoria virtual el hueco entre pila y datos no consume recursos físicos.
- Pero hay modelos de imagen de memoria con una sola región y también con regiones no fijas, que son opciones más avanzadas como Windows.
- **El sistema operativo** mantiene información adicional sobre los procesos, en la Tabla de Procesos. Cada una de las entradas de la TP es un BCP (Bloque de Control de Procesos), que mantiene casi toda la información sobre el proceso.
  - **Contenido del BCP:**
    - Información de identificación:
      - Estado del proceso.
      - Evento por el que espera, si es que está bloqueado.
      - Prioridad del proceso.
      - Información de planificación.

        - Estado del procesador:
            - Archivos abiertos.
            - Puertos de comunicación usado
            - Temporizadores
        - Información de control del proceso:
            - Punteros para estructurar los procesos en cola.
    - **Contenido fuera del BCP**:
        - El BCP no almacena toda la información del un proceso, se decide que no almacenar según la eficiencia, si se puede almacenar bien o no en la tabla, y si hay que compartir información, si se comparte algún dato no estará en el BCP.
        - Se usan punteros, que se almacenan en tablas de páginas, y desde el BCP se puede acceder a esa tabla. Se usa este método porque tienen tamaños variables y el compartir datos debe ser externo.
    - La tabla de punteros de posición de los ficheros, que almacena los ficheros abiertos por un proceso, se almacena fuera del BCP y si se almacena dentro no podrán ser compartidos.

- **El servicio fork,** hace que el proceso padre e hijo compartan el contexto, y solo se hará una copia cuando el hijo haga una modificación, mientras comparte el espacio con el padre.
- **El servicio exec,** lee un ejecutable, y cambia el mapa de memoria y el contexto, al proceso que lo está ejecutando.
- **El servicio exit,** finaliza la ejecución del proceso, cierra los descriptores, liberan todos los recursos y libera el BCP. Los 3 están explicados en el Tema 1.

**Tipos de sistemas operativos:**

- **Multiproceso**: Utiliza el paralelismo, varios procesos funcionan a la vez.
    - Multiusuario y Monousuario.
- **Monoproceso**: Concurrencia, funciona un proceso a la vez, pero se van compaginando y da la sensación de paralelismo
    - Monousuario.

**Multitarea**: Alternancia en los procesos de fases de E/S y de procesamiento, la memoria almacena varios procesos.

- **Ventajas**:
  - Facilita la programación, se divide un programa en procesos. Modularidad.
  - Permite el servicio interactivo simultáneo de varios usuarios.
  - Aprovecha los tiempos de espera de E/S de los procesos.
  - Aumenta el uso de CPU.
- **Grado de multiprogramación:** Numero de procesos activos, cuantos más procesos el rendimiento de la máquina disminuye. Los sistemas con memoria virtual dividen el espacio direccionable de los procesos en páginas y el de memoria física en marcos de página. El numero de páginas en memoria principal es el Conjunto Residente.
  - Cuando aumenta el grado de multiprogramación, disminuye el tamaño del conjunto residente de cada proceso, para que quepan todos.
    - **Con poca memoria física:** Se produce hiperpaginación cuando alcanzo alto porcentaje de uso, para solucionarlo hay que ampliar la memoria principal.
    - **Con mucha memoria física**: Se alcanza alto porcentaje de utilización de CPU con menos procesos de los que caben en memoria, para solucionarlo se puede mejorar el procesador o añadir más procesadores.

**Cambio de contexto**: Cuando se pasa de ejecutar un proceso a otro. Se produce cuando el sistema operativo asigna el procesador a un nuevo proceso.

- **Tipos de cambios de contexto:**
  - **Cambio de contexto voluntario(CCV):** El proceso realiza una llamada al sistema o produce una excepción que implica esperar por un evento, así deja funcionar a otro.
  - **Cambio de contexto involuntario(CCI)**: El SO quita de la CPU al proceso, puede darse porque el proceso haya agotado su rodaja de ejecución.
- **Acciones**:
  - Guarda el estado del procesador en el BCP del proceso en ejecución.
  - Restaurar el estado del nuevo proceso en el procesador.

Generación de ejecutables: MIRAR LIBRO Y DIAPOSITIVA.

## 2.2. Tema 2.2: Planificación de procesos.

**Creación de procesos en UNIX:** Se distingue entre crear procesos y ejecutar nuevos programas.

- Para crear un proceso se hace mediante fork(), que no cambia la imagen de memoria y mantiene los valores del padre, crea una copia, y los cambios en el hijo no le afectan. Copia el BCP del padre y su Mapa de memoria (incluyendo pilas).
    - Un inconveniente es que muchos datos podrían compartirse.
- Para ejecutar un programa se hace mediante exec(), se hace sobre procesos hijo, sustituye su imagen en memoria por la del programa. Mientras el padre puede esperar al hijo con wait o seguir creando hijos.
    - Un inconveniente es que fork ha copiado todo lo del padre, para que luego lo deshecha el exec.
- Para solucionar ambos inconvenientes se usa COW(Copy-on-Write), los datos se marcan de manera que si se intentan modificar se realiza una copia para cada proceso.

**Terminación de procesos:** Todos los recursos asignados son liberados, excepto el valor del exit que se mantiene en el BCP esperando el wait del padre y si el padre ha muerto (el hijo se queda zombi) lo hereda el init y este hace el wait. Entonces se elimina el BCP, tras el wait. Puede terminar de 2 maneras:

- **Voluntariamente**: Llamada al sistema exit().
- **Involuntariamente**: Excepciones, abortado por el usuario(ctrl+c) u otro proceso(kill).

**Expulsión a disco(swap):** Cuando hay demasiados procesos y baja el rendimiento, se pasan algunos procesos a disco con swap que libera memoria, y cuando se quieran seguir ejecutando se pasa de nuevo a memoria. Los nuevos estados son: Bloqueado y suspendido, y Listo y suspendido.

**Niveles de planificación:**

- Planificación a corto plazo: Selecciona el siguiente proceso a ejecutar.
- Planificación a medio plazo: Elegir que proceso se retira o añade a memoria principal.
- Planificación a largo plazo: Realiza el control de admisión de procesos a ejecutar.

**Tipos de planificación:**

- **No apropiativo:** El proceso conserva el uso de CPU, no se puede expulsar hasta que termine.
- **Apropiativo:** El sistema puede expulsar a un proceso de CPU, se puede expulsar, aunque no haya terminado para ejecutar otro proceso.

**Momentos de decidir la planificación:**

- Cuando un proceso se bloquea, mientras espera el evento se pasa a ejecutar otro proceso.
- Cuando se produce una interrupción, como la del reloj o de E/S.
- Cuando termina un proceso.

**Cola de procesos:** Los procesos listos para ejecutar se mantiene en una cola, que puede ser única, por tipo de proceso o por prioridad. Cuando se saca un proceso que no ha terminado se pone al final de la cola. Si entra a ejecutar uno nuevo se pone antes que el que está terminando.

**Algoritmos de planificación:**

- Se pueden orientar a:
  - **Utilización de CPU:** maximizar el tiempo de uso.
  - **Productividad:** maximizar el número de procesos que terminan por unidad de tiempo.
  - **Tiempo de retorno(Tq=Tf-Ti):** Minimizar el tiempo que un proceso pasa desde que entra hasta que termina.
  - **Tiempo de servicio(Ts):** Tiempo que se ha dedicado a tareas productivas.
  - **Tiempo de espera(Te):** Tiempo que un proceso ha estado en la cola de espera.
  - **Tiempo de retorno normalizado(Tn=Tq/Ts):** Tanto por uno que un proceso ha estado en espera.
- **Asignación First to Come First to Serve (FCFS):** Es FIFO, primero en llegar primero en salir. Es no apropiativo, no se puede expulsar un proceso. Penaliza a los cortos, que tienen que esperar a que terminen todos los previos.
- **Asignación Shortest Job First:** Primero los procesos más cortos, menor tiempo de servicio. Es no apropiativo, no se puede expulsar un proceso. Se debe conocer la duración de antemano. El problema es que si solo llegan procesos cortos los largos no se ejecutan.

- **Cíclico o Round-Robin:** Mantiene una cola FIFO con los procesos listos para ejecutar. A cada proceso se le asigna una rodaja de tiempo de procesador, cuando la agota pasa a la cola y se ejecuta el siguiente en la cola. Se ejecutan un poco todos, hasta que terminan. Un proceso puede volver a lista porque acabe su tiempo o pase ha bloqueado, esperando un evento. Es apropiativo, por eso se puede expulsar un proceso para poner otro a ejecutar.

    - Hay que intentar que tampoco haya demasiados cambios de contexto, ya que estos añaden tiempo.

- **Asignación por prioridades:** Cada proceso tiene una prioridad asignada, y se seleccionan primero los procesos con mayor prioridad.

### 2.3. Tema 2.4: Hilos y Procesos.

Hilos de ejecución de un proceso de forma concurrente. Los threads de un proceso se guardan en su BCP en la tabla de threads, Threads Table, donde se almacena su información y contenido. Se considera la unidad básica de utilización de la CPU.

**Crear hilos permite que:**

- Compartan memoria entre ellos, pero cada uno necesita una pila para almacenar sus datos y un minicontexto propio.
- Aprovechen mejor la rodaja de tiempo que tienen asignada, al hacer menos cambios de contexto, que los que implica crear y terminar un proceso hijo.
- Un thread está ligado a un proceso, no es independiente por lo que si muere el proceso mueren los hilos, no como pasa con los procesos hijos.

**Contenido de un hilo:**

- Identificador del thread.
- Contador de programa.
- Conjunto de registros.
- Pila.

**Comparten entre hilos:**

- Mapa de memoria.
- Ficheros abiertos.
- Señales, semáforos y temporizadores.

**Beneficios:**

- Mayor interactividad al separar las tareas en hilos.
- Comparten la mayor parte de los recursos.
- Cuesta menos tiempo crear un hilo que un proceso.
- En arquitecturas multiproceso permite hacer ejecución en paralelo asignando hilos distintos a distintos procesadores.

**Soporte de hilos:**

- **Espacio de usuario, ULT**, User Level Threads.
  - Están implementados en forma de biblioteca de funciones.
  - El kernel no tiene conocimiento sobre los threads, solo sobre el proceso creador.
  - Es más rápido, pero un bloqueo bloquea todo el proceso.
  - Un hilo de usuario se puede pasar al kernel para que esté lo gestione, y deja de depender del proceso y aunque muera el proceso no muere el hilo. Cambia su id.
- **Espacio de núcleo, KLT**, Kernel Level Threads.
  - Son de los que se ocupa el kernel, de crearlos, planificarlo y destruirlos.
  - Un poco más lentos al hacerlo por medio del kernel y no directamente.
  - Los bloqueos solo bloquean el thread implicado, no a todo.
  - Se pueden ejecutar varios procesos a la vez, pero hay un límite de threads.

**Modelos de múltiples hilos:**

- **Muchos a uno:** Corresponden muchos hilos de usuario con un único hilo del núcleo. Una llamada bloqueante bloquea todos los hilos.
- **Uno a uno:** Hace corresponder un hilo del kernel a cada hilo de usuario. La mayoría de las implementaciones restringen al numero de hilos que se pueden crear.
- **Muchos a muchos:** Multiplexa los threads de usuario en un numero determinado de threads en el kernel. El núcleo se complica mucho.

**Aspectos de diseño:**

- Cuando se hace fork de un proceso con hilos, duplica el proceso con todos sus hilos.
- exec no es adecuado, sustituirá la imagen del proceso, que quitará todo incluido los hilos.
- **Otra versión:** Cuando hace fork, duplica el proceso solo con el hilo que hace fork.
- Esta versión es más eficiente si va a hacer un exec.
- **Cancelación de hilos:** Un hilo notifica a otro de que deben terminar.
  - **Cancelación asíncrona:** Se fuerza la terminación instantánea del hilo.
    - Puede ocasionar problemas con los recursos asignados.
  - **Cancelación diferida:** El hilo comprueba periódicamente si debe terminar. Es preferible.
- Para aplicaciones que reciben peticiones y las procesan se pueden usar hilos.
  - Se hace **Thread Pool (Conjunto de Hilos)** que consiste en crear con anterioridad los hilos para evitar el tiempo de creación (retardos) y se dejan en espera, y se establece un límite, para intentar evitar que una avalancha de peticiones agote los recursos del sistema. Cuando se reciben más peticiones de las que se pueden tratar se ponen en cola y cuando la cola se ha llegado se hace denegación de servicios, que no la deja entrar mientras no vayan acabando las peticiones.

**Hilos en pthreads:**

- **Creación de hilos:**
  - int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*func)(void *), void *arg)
    - Devuelve un entero que indica como ha ido la ejecución.
    - *thread es el puntero donde se almacenará el ThreadId, el manejador del hilo.
    - *attr Estructura de atributos.
    - *(*func)(void *) función con el código a ejecutar.
    - *arg parámetro del hilo.

- Estructura pthead_attr_t: Son los atributos asociados a cada hilo.
  - Controlan:
    - Si el hilo es independiente o dependiente, de kernel o de biblioteca.
    - Tamaño de la pila privada.
    - Localización de la pila.
    - Política de planificación, solo posible si es de kernel.
  - Inicializar: pthread_attr_init(pthread_attr_t *attr)
  - Destruir: pthread_attr_destroy(pthread_attr_t *attr)
  - Por defecto el atributo de independencia está en PTHREAD_CREATE_JOINABLE, lo que quiere decir que espera un join y no se liberan los recursos. La otra opción cuando acaba el sistema operativo libera los recursos. PT..._DETACHED.
  - El resto en diapositiva 28 de Tema 2.3.
- pthread_t pthread_self(void) Devuelve el identificador del thread.
- int pthread_join(pthread_t thread, void **value) Es el wait de los hilos.
  - thread manejador del hilo a esperar.
  - **value Valor de terminación del hilo.
- int pthread_exit(void *value) Finaliza su ejecución y devuelve value que puede ser de cualquier tipo. Aunque no hagan join, no se quedan hilos zombis.
- **EVITAR usar variables globales, fork y exec.**

**Planificación de hilos:** Se basa en el modelo de prioridades y no utiliza el modelo de segmentación por segmentos de tiempo. Un thread continuará ejecutándose en la CPU hasta pasar a un estado que no le permita seguir en ejecución. Para alternar entre thread, se debe asegurar que el thread permita la ejecución de otros threads.

- PTHREAD_SCOPE_PROCESS Planificación PCS, del proceso del user, de biblioteca.
- PTHREAD_SCOPE_SYSTEM Planificación SCS, del kernel.

### 2.4. Tema 2.4 Señales

**Señales**: Son un mecanismo para comunicar eventos a los procesos, para que estos puedan reaccionar. Son excepciones asíncronas, se procesan de inmediato. Posibles reacciones a una respuesta:

- Ignorar la señal, SIG_IGN.
- Invocar la rutina de tratamiento por defecto, si el proceso no esperaba ninguna lo general es que sea muerte.

- Invocar a una rutina de tratamiento propia.
    - Por ejemplo, un proceso padre recibe la señal SGCHLD cuando su proceso hijo termina, esa señal la envía el SO, no el hijo directamente.

- Son interrupciones al proceso, si está ejecutando normal la rutina se lleva a cabo en el siguiente ciclo de reloj y si se captura cuando termine continuara por donde estaba. Pero si está en un bucle se ejecuta en ese momento, ya que no se sabe cuando terminara el bucle.
- **Tratamiento**: El SO las transmite al proceso, pero el proceso debe estar preparado para recibirlo o en general muere. El proceso debe especificar con sigaction que señal espera y que rutina de tratamiento seguir.
- **Enviar señal a un proceso**: **int kill(pid_t pid, int sig);** No es para matar el proceso, aunque si el proceso no la espera le matará.
    - Para enviar una señal a sí mismo **int raise(int sig);**
    - CTRL-C SIGINT
    - CTRL-Z SIGSTOP
- **Esperar a una señal:** Con el servicio **int pause(void)** el proceso espera hasta que le llegue una señal cualquiera que le reanuda, se utiliza para ejecutar en orden.
    - No se puede especificar un plazo de desbloqueo.
    - No se puede indicar el tipo de señal que se espera.
    - No se desbloquea si recibe una señal ignorada.
- **Sleep(unsigned int sec)** Suspende un proceso hasta que vence un plazo o se recibe una señal.
- **Especificar acción:** Se utiliza sigaction para especificar la acción a realizar como tratamiento de la señal sig. OJO, cuando se ha completado la rutina rearmar el sigaction.
    - **int sigaction(int sig, struct sigaction *act, struct sigaction *oact);**
        - Preferible ante signal(,)
        - Sig es la señal para la que actúa.
        - En la estructura de act, en sa_handler se pone que rutina seguir.
        - En oact está la configuración.

- **Estructura sigaction{void (*sa_handler)(); sigset_t sa_mask; int sa_flags;};**
  - Sa_handler Es el manejador, lo que hace, puede ser SIG_DFL por defecto, SIG_IGN ignora la señal o una dirección de una función.
  - Sa_mask Mascara de señales a ignorar durante el manejador.
  - Sa_flags Opciones.

- **Lista de señales importantes**: Están incluidas en signal.h, que son propias del sistema operativo.
  - SIGILL Instrucción ilegal.
  - SIGALRM Cuando termina un temporizador.
  - SIGKILL Mata al proceso.
  - SIGSGEV Violación de segmento de memoria.
  - SIGUSR1 y SIGUSR2 Reservadas par el uso del programador.

- **Conjunto de señales:**

**Temporizadores:** El sistema operativo mantiene un temporizador por proceso, en su BCP.

- Es el sistema operativo quien actualiza todos los temporizadores.
- Cuando llega a cero el SO pasa al proceso la señal SIGALRM, para que se ejecute la rutina de tratamiento.

- **Establecer un temporizador:** int alarm(unsigned int sec) puede cambiarse sec por otra unidad de tiempo. Si el parámetro es 0, desactiva el temporizador.
- **Diferencia de sleep y alarm:**
  - **alarm**, es una llamada al sistema y del proceso se duerme exactamente el tiempo indicado, lo tiene medido.
  - **sleep**, no es una llamada al sistema y como mínimo duerme el tiempo indicado.

**Excepciones**: Cuando el hardware detecta condiciones especiales; fallo de página, escritura a página de solo lectura, desbordamiento de pila, violación de segmento, syscall

- Transfiere control al SO para su tratamiento, que:
  - Salva el contexto del proceso.
  - Ejecuta la rutina si es necesario.
  - Envía una señal al proceso indicando la excepción.

- Sirven para optimizar el rendimiento, ya que ahorran código de comprobaciones.
- Muchos lenguajes usan mecanismos llamados Exceptions para controlar errores.

**Entorno de un proceso**: Se hereda del padre y contiene los siguientes datos:

- Vector de argumentos del comando del programa
- Vector de entorno, una lista de variables que el padre pasa al hijo. <nombre, valor>
    - Es una forma flexible de comunicar ambos procesos y determinar aspectos de la ejecución del hijo en modo usuario. Partícula a aspectos a nivel de cada proceso.
- **Variables de entorno:** Mecanismo de paso de información a un proceso.
- **Localización:** El entorno se coloca en la pila del proceso al iniciarlo. Y se recibe como tercer parámetro de main la dirección de la tabla de variables de entorno.

# 3. TEMA 3

**Tema 3.1: Procesos concurrentes y problema en la comunicación y la sincronización.**

**Procesos concurrentes:** Dos procesos ejecutándose en el mismo momento, es decir, se ejecutan de manera que sus intervalos de ejecución se solapan. No que uno se pasó a otro.

- **Tipos de concurrencias:**
  - **Concurrencia aparente:** Hay más procesos que procesadores, los procesadores se multiplexan en el tiempo. Pseudoparalelismo.
  - **Concurrencia real:** Cada proceso se ejecuta en un procesador, se produce una ejecución en paralelo. Paralelismo real.
- **Modelos de programación concurrente:**
  - **Único procesador:** El sistema operativo se encarga de repartir el tiempo entre los procesos, incluso expulsándolos. Es inherente.
  - **Multiprocesador**: Combinan paralelismo real con pseudoparalelismo. Ya que normalmente hay más procesos que procesadores.
  - **Sistema distribuido:** Varios computadores conectados por red.
- **Ventajas de ejecución concurrente:**
  - Facilita la programación, se puede estructurar en procesos separados.
  - Acelera la ejecución de cálculos, división de cálculos en procesos ejecutados en paralelo.
  - Mejora la interactividad de las aplicaciones, separar las tareas de procesamientos de las tareas de atención de usuarios.
  - Mejora el aprovechamiento de la CPU, aprovechan las fases de E/S de una aplicación.
- **Tipos de procesos concurrentes:**
  - **Independientes**: Se ejecutan concurrentemente, pero sin ninguna relación.
    - No necesitan comunicarse, ni sincronizarse.
  - **Cooperantes**: Se ejecutan concurrentemente con alguna interacción entre ellos.
    - Pueden comunicarse entre sí y pueden sincronizarse.

- **Interacciones entre procesos:**
  - **Accesos a recursos compartidos**: Procesos que comparten recursos y que compiten por un recurso.
  - **Comunicación**: Procesos que intercambian información.
  - **Sincronización**: Un proceso debe esperar a un evento en otro proceso.

**Condiciones de carrera**: Cuando varios procesos trabajan sobre el mismo recurso a la vez y el resultado que obtenemos es cada vez distinto, y nunca se obtiene el resultado correcto. Este problema se debe a que dependen de la velocidad de cada proceso.

- El funcionamiento y resultado de un proceso debe ser independiente de su velocidad relativa de ejecución con respecto a otros procesos. Para garantizar que el orden de ejecución no afecte al resultado, se utilizan mecanismos como la Exclusión Mutua.
- **Exclusión Mutua:** Cuando un proceso usa un recurso compartido nadie más lo puede usar, de esta manera nos aseguramos de que no acceden dos procesos a los mismos datos de forma simultánea. Solamente un proceso puede estar simultáneamente en la sección crítica de un recurso.
  - **Sección crítica**: Es el segmento de código que manipula un recurso.
  - **Problemas de la sección crítica**:
    - **Interbloqueos**: Se produce cuando admitimos exclusión mutua para más de un recurso, de esta manera los dos pueden quedarse bloqueados y no avanzan. Cuando está en una sección crítica y solicita entrar en otra, y el proceso que está en esa otra esta esperando a entrar en la sección crítica del primero.
    - **Inanición**: Un proceso queda indefinidamente bloqueado, a la espera de entrar. No puede acceder a la sección crítica porque llegan otros.
  - **Condiciones para la exclusión mutua:**
    - Solamente se permite que un proceso esté en la sección crítica de un recurso.
    - Tiene que ser justo y dejar a todos entrar, no postergarlo indefinidamente.
    - Si una sección crítica no está siendo accedido, se podrá acceder sin demora.
    - No puede depender de la velocidad relativa de los procesos.
    - El tiempo dentro de una sección crítica es finito.
  - **Mecanismos de sincronización**: Cualquier mecanismo que solucione el problema de la sección crítica debe proporcionar sincronización entre procesos. Primero se debe solicitar permiso para entrar en la sección crítica y cuando termine lo debe indicar. **Alternativas**:

- **Desactivar interrupciones**, en el caso de los procesadores monoprocesador, de esta manera no ejecuta el otro proceso para acceder a la sección crítica.
- **Instrucción máquina**, con una variable, test and set o swap. Pero implican espera activa, en un bucle o similar, y además es posible inanición e interbloqueos.
- **Solución de Peterson:** Solo válida para 2 procesos. Asume que Load y Stores son atómicas y no interrumpibles. Ambos procesos comparten 2 variables, una que es el turno y otra que es flag un array de 2 booleanos. Turno indica quien entra y flag controla si un proceso está listo para entrar.
- **Semáforos (Dijkstra):** Sincronización de procesos mediante un mecanismo de señalización, un semáforo. Un semáforo se asocia a una sección crítica.
- **Las posibles operaciones son:**
  - Primero, iniciación a un valor, que no debe ser negativo.
  - semWait, que reduce el contador del semáforo. Espera a que se desbloquee, y cuando entra reduce el contador para bloquearlo.
    Se bloquea cuando el contador<0.
  - semSignal, que incrementa el contador del semáforo. Se hace al salir, para que otro pueda entrar.
    Se desbloquea cuando el contador<=0
- **Código**:
  - **Declarar**: sem_t semáforo;
  - **Inicio**: sem_init(&semáforo, 0,1);
  - **Entrada/Pedir acceso**: sem_wait(&semáforo)
  - **Salida/Indicar abandono**: sem_post(&semáforo);
  - **Fin**: sem_destroy(&semáforo);

- **El problema del productor-consumidor:** Proceso/Thread que va escribiendo algo en un buffer y otro lo va leyendo. Se puede hace con semáforos, primero uno se encarga de controlar el acceso a ese espacio intermedio y otro se encarga de dar paso a cada uno u otro. El segundo lo empieza en 0, para que uno lo primero que haga es esperar y mientras otro hace un camino hasta aumentar ese numero y dar paso al otro. Los dos modifican la zona de datos compartida.

  - El proceso productor produce elementos de información.
  - El proceso consumidor consume elementos de información.
  - Hay un espacio de almacenamiento intermedio, que es el buffer.

- **El problema de los lectores-escritores:** Se plantea cuando se tiene un área de almacenamiento compartida.
    - Múltiples procesos leen información.
    - Múltiples procesos escriben información.
    - **Condiciones**: Se le da prioridad una de las dos funciones, escritura o lectura.
        - Cualquier numero de lectores pueden leer de la zona de datos concurrentes, pero si se está escribiendo no puede leer nadie y solo escribir uno.
        - Solamente un escritor puede modificar la información a la vez.
        - Durante una escritura ningún lector puede realizar una consulta.
    - **Lectores con prioridad:** Si hay alguien leyendo pueden entrar más, pero solo se podrá escribir si no hay nadie leyendo. Este puede provocar inanición para escritores.
    - **Escritores con prioridad:** Si se está escribiendo no se admiten lectores.

## 3.1. Tema 3.2: Hilos y mecanismo de comunicación y sincronización.

- **Mecanismos de comunicación**: Permiten la transferencia de información entre dos procesos.
    - Archivos, Tuberías, variables en memoria compartida y Paso de mensajes.
- **Mecanismos de sincronización**: Permiten forzar a un proceso a detener su ejecución hasta que ocurra un evento. Al recibir una señal se desbloquea. La operación debe ser atómica.
    - Señales, Tuberías, Semáforos, Mutex y variables condicionales y Paso de mensajes.
- **Semáforos POSIX:** Mecanismo de sincronización para procesos y threads.
    - **Semáforos con nombre**: Se utiliza para distintos procesos, deben conocer el nombre.
    - **Semáforos sin nombre**: Solo puede ser utilizado por el proceso que lo crea, por lo tanto sirve para threads de un proceso, o procesos que tengan memoria compartida.
    - **Estructura (ya creado el semáforo):**
        - sem_wait(s); //Entrada, resta 1
        - sección crítica, código que usa la sección.
        - sem_post(s); //Salida, suma 1

- **Diferencias con otros problemas:**
  - **Lectores-escritores:** Pueden leer y escribir múltiples procesos, leer simultáneamente cuantos quieran, pero escribir solo puede hacerlo uno a la vez y sin leer otros.
  - **Exclusión mutua:** Solo permite a un proceso acceder a la vez a la información.
  - **Productor-consumidor:** Los dos procesos modifican la zona de datos compartida.
- **Mutex (Mutual Exclusion) y variables condicionales**: Mecanismo de sincronización indicado para procesos ligeros, threads. Es un semáforo binario, 1 o 0, con dos operaciones:
  - **lock(m)** Bloquea el mutex, y si la lo esta suspende el proceso.
  - **unlock(m)** Desbloquea el mutex, pero solo uno de ellos.
- Variables condicionales: Variables de sincronización asociadas a un mutex, tiene dos operaciones, que conviene ejecutarlas entre lock y unlock:
  - **wait** Bloquea el thread y le expulsa del mutex, lo libera para el otro.
  - **signal** Desbloquea a uno o varios procesos suspendidos en la variable condicional.
  - **Uso de mutex y variables condicionales:**

## 3.2. Tema 3.3: Desarrollo de servidores concurrentes.

- **Servidores de peticiones:** Se aplican en muchos contextos. Un servidor recibe peticiones que debe procesar.
  - **Estructura**:
    - **Recepción de petición**: Cada petición requiere un cierto tiempo en operaciones de entrada/salida.
    - **Procesamiento de las peticiones**: Tiempo de procesamiento en CPU.
    - **Envío de respuesta**: Tiempo de entrada/salida para contestar.
  - Una solución es ejecutar la recepción, procesamiento y envío, en un único proceso. Es muy lento, si llegan dos peticiones al mismo tiempo o mientras una se procesa se pierden.

- **Solución basada en procesos**: Cada vez que llega una petición se crea un proceso hijo.
    - El proceso hijo realiza el procesamiento de la petición.
    - El proceso padre pasa a esperar la siguiente petición.
    - El problema es tener que arrancar un proceso por cada petición que llega, y terminarlo por cada petición. Consume demasiados recursos.
- **Solución basada en hilos bajo demanda**: Cada vez que se recibe una petición se crea un hilo.
    - Un hilo receptor encargado de recibir las peticiones.
    - Cada vez que llega una petición se crea un hilo y se le pasa una copia la petición al hilo recién creado.
    - La creación y terminación de hilos tiene un coste menor que la de procesos, pero sigue siendo un coste.
    - Hay que controlar la condición de carrera.
- **Solución basada en pool de hilos**: Se tiene un numero fijo de hilos creados desde el principio para ejecutar un servicio.
    - Cada vez que llega una petición se pone en una cola de peticiones pendientes.
    - Todos los hilos esperan a que haya alguna petición en la cola y la retiran para procesarla.

# 4. TEMA 4

## 4.1. Tema 4.1: Ficheros

- **Fichero**:
  - **Almacenamiento**:
    - **Memoria principal:** Volátil, no persistente. Desaparecen al apagar el sistema y son accedidos por el procesador.
    - **Memoria secundaria:** No volátil, persistente. Organizada en bloques de datos, se necesita una abstracción para simplificar las aplicaciones, fichero.
  - Cuando se lee o escribe se opera a nivel de bloque, no de byte. Se cogen bloques completos.
  - **Sistema de ficheros:** Ofrece al usuario una visión lógica simplificada del manejo de los dispositivos periféricos en forma de ficheros. Capa de software entre dispositivos y usuarios.
    - Proporciona un mecanismo de abstracción que oculta los detalles relacionados con el almacenamiento y distribución de la información.
    - **Funciones:**
      - Organización.
      - Almacenamiento.
      - Recuperación.
      - Gestión de nombres.
      - Implementación de la semántica de Coutilizacion.
      - Protección.
    - **Función principal:** Establece una correspondencia entre los ficheros y los dispositivos lógicos.
    - **Visión del usuario:**
      - **Visión lógica:** Ficheros, Directorios, Sistemas de Ficheros y particiones (simula discos a partir de un dispositivo. La tabla de particiones almacena los nombres, bloques de inicio y fin de cada partición).
      - **Visión física:** Bloques o bytes.

    - **Características para el usuario**: Almacenamiento permanente de información, no desaparece, aunque se apague el computador.
        - Información estructurada de forma lógica.
        - Nombres lógicos y estructurados.
        - Se acceden a través de llamadas al sistema operativo o de biblioteca de utilidades.
    - **El acceso a los dispositivos es:** Incómodo y no seguro, si se accede no hay restricciones.

- **Atributos**:

    - **Nombre**: Identificador legible por una persona.
        - Es característico, para que el usuario lo recuerde, ya que el identificador número es muy complicado.
        - **Longitud**: fija en MS DOS o variable en UNIX, Windows.
        - **Extensión**: Fija para cada tipo de ficheros.
        - Sensible a tipografía.
    - **Identificador:** Etiqueta unívoca del archivo.
    - **Tipo de fichero:** Formatos de Ficheros.
    - **Ubicación**: Identificador del dispositivo del almacenamiento y posición dentro del fichero.
    - **Tamaño de fichero:** Numero de bytes en el fichero.
    - **Protección:** Control de accesos y de las operaciones sobre el fichero.
    - **Información temporal:** Creación, acceso, modificación, etc.

- **Los directorios** relacionan nombres lógicos y descriptores internos de ficheros.

- **Operaciones:**

    - **Creación:** Asignación de espacio inicial y metadatos.
    - **Borrado**: Liberación de recursos.
    - **Escritura**: Almacena información.
    - **Lectura**: Recupera información.

- **Vista lógica:**
  - **Estructura del fichero**:
    - **Ninguna**, secuencia de palabras o bytes.
    - **Estructura sencilla de registros**, en líneas de tamaño fijo o variable.
    - **Estructuras complejas**, documentos con formato y fichero de carga reubicable.
  - Conjunto de información relacionada que ha sido definida por su creador.
  - Secuencia o tira de bytes.
  - **Métodos de acceso:**
    - **Acceso secuencia**: Basado en el modelo de acceso a datos en una cinta magnética. Operaciones orientadas a bytes, se puede avanzar hacia adelante o atrás, pero no dar un salto.
    - **Acceso directo**: Basado en el método de acceso a dispositivos de disco. Fichero dividido en registros de longitud fija. Puede dar saltos, utilizando un puntero de posición para evitar tener que especificar la posición en todas las operaciones.
- **Semántica de compartición**:
  - **Compartición de ficheros:**
    - Varios procesos pueden acceder simultáneamente a un fichero. Ambos pueden leer a la vez, pero no escribir simultáneamente.
    - Es necesario definir una semántica de coherencia.
  - **Opciones**:
    - **Semántica UNIX:** Las escrituras en un archivo son inmediatamente visibles
      - Un archivo abierto tiene asociado un puntero de posición, el puntero puede ser compartido o uno por proceso.
    - **Semántica de sesión:** Las escrituras sobres archivo abierto no son visibles por otros procesos con el archivo abierto. Cuando se cierra un fichero los cambios son visibles. Será visible para los que habrá el fichero después de la modificación, pero no antes.
    - **Semántica de archivos inmutables**: Un archivo puede ser declarado como compartido. No se puede modificar, no admite modificación de nombre y contenido.
    - **Semántica de versiones**: Las actualizaciones se hacen sobre copias con numero de versión. Visibles cuando se consolidan versiones. Sincronización explicita si se requiere actualización inmediata.

- **Control de acceso:**
  - **Lista de control de acceso:** Lista de usuarios que pueden acceder a un fichero.
    - Si hay diferentes tipos de acceso una lista por tipo de control de acceso.
  - **Permisos**:
    - Versión condensada.
    - Tres tipos de acceso (rwx)
    - Permisos para tres categorías (usuario, grupo, otros)

- **Llamadas al sistema:**
  - **Abrir**: int open(const char * path, int flags, [mode_t mode])
    - Abre o crea un fichero especificado por path.
    - **Flags**: O_RDONLY, O_WRONLY, or O_RDWR. Se pone solo 1
    - **Mode**: O_CREAT, O_APPEND, O_TRUNC, ... . Se separan con |.
    - Devuelve un descriptor de fichero (o -1 si error).
  - **Crear un fichero:** int creat(rutaDelFichero, permisos) La ruta con el nombre del fichero.
    - OJO, creat siempre crea un fichero para solo lectura.
  - **Cerrar**: int close(int fildes)
    - Cierra un archivo abierto anteriormente asociado al descriptor fildes (o -1 si error).
    - Cierra, pero no borra el fichero, para ello se usa el unlink.
  - **Leer**: ssize_t read(int fildes, void * buf, size_t nbyte)
    - Intenta leer de un archivo (fildes) tantos bytes como indica nbyte, colocando la información leída a partir de la dirección de memoria buf.
    - Devuelve el número de bytes leídos (que puede ser menor o igual a nbyte). 0 → Fin de fichero (EOF).
  - **Escritura**: ssize_t write(int fildes, const void * buf, size_t nbyte)
    - Intenta escribir en un archivo (cuyo descriptor de fichero fildes se obtuvo de abrirlo) tantos bytes como indica nbyte, tomándolos de la dirección de memoria indicada buf.
    - Devuelve en n el número de bytes que realmente se han escrito (que puede ser menor o igual a nbyte). Si retorno = -1 → Error de escritura.

- **Desplazamiento en un fichero:** off_t lseek(int fildes, off_t offset, int whence) l de long
  - Modifica el valor del apuntador del descriptor fildes en el archivo, a la posición explícita en desplazamiento (offset) a partir de la referencia impuesta en origen (whence). Si retorno = -1 → Error de posicionamiento.
  - **whence** indica desde dónde se salta:
    - SEEK_SET → desde el principio del fichero.
    - SEEK_CUR → desde la posición actual.
    - SEEK_END → desde el final del fichero.
  - offset se expresa en un numero positivo o negativo de bytes.
  - En C se puede saltar más lejos que el final del fichero, pero estaremos fuera, el sistema operativo asume que sabemos lo que hacemos.
- **Crear un enlace a un fichero**: int link(const char *nombre, const char *nuevo);
  - Se crea un enlace al nodo i del fichero indicado, si se ha creado no se borra hasta eliminar todos los hardlink.
  - Se crea entre ficheros de la misma partición.
  - Comparte el i nodo del enlazado, no tiene uno propio
  - Crea un hard link con nombre nuevo al archivo nombre.
  - Incrementa contador de enlaces del fichero nombre.
- **Crear un enlace simbólico:** int symlink(const char *nombre, const char *nombreenlace);
  - Tiene su propio i nodo, no lo comparte, ya que es un fichero que almacena el nombre/path del fichero al que apunta. Por lo que no suma al contador de enlaces, por lo que si se quiere borrar el fichero este enlace no afecta y se borrará.
  - Crea un enlace simbólico hacia nombre desde nombre enlace.
  - Crea un nuevo archivo nombreenlace que incluye ”nombre” como únicos datos.
- **Borrar un fichero de disco:** int unlink(const char *nombre);
  - Borra el archivo nombre siempre que NO tenga enlaces hard pendientes (contador enlaces = 0) y nadie lo tenga abierto.
  - Si hay enlaces duros (contador enlaces > 0), se decrementa el contador de enlaces.
  - Si algún proceso lo tiene abierto, se espera a que lo cierren todos.
  - El fichero debe estar cerrado, con close(,), si no lo está tardara en borrarlo

- **Representación del fichero**: El sistema debe mantener información sobre el fichero, metadatos, que son dependiente del sistema operativo.
    - **Asignación de espacio en disco:** Gestión del espacio libre y ocupado del disco.
        - **Preasignación**: Asigna en creación del tamaño máximo posible del fichero, todo el que podría necesitar.
        - **Asignación dinámica:** Asignación de espacio según se va necesitando, va tomando unidades de asignación según le hagan falta.
        - **Cuestiones a considerar** en cuanto al tamaño de asignación:
            - **Tamaño de asignación grande:** Permite tener toda la información contigua en disco, mayor rendimiento.
            - **Tamaño de asignación pequeño:** Se necesitan más metadatos, aumenta el tamaño de los metadatos.
            - **Tamaño de asignación fijo**: Reasignación de espacio simple.
            - **Tamaño de asignación fijo y grande**: Incrementa el malgasto de espacio.
            - **Tamaño de asignación variable y grande**: Incrementa el rendimiento, pero aumenta la fragmentación externa.
        - **Fragmentación**: Para solucionarlo es necesaria la compactación, elimina los espacios.
            - **Interna**: Cuando queda espacio libre en el último bloque de la secuencia.
            - **Externa**: Cuando hay bloques vacíos que no podemos usar al no haber suficientes contiguos como para introducir información.
    - **Asignación encadenada**: Cada bloque contiene un puntero al bloque siguiente, se asignan bloques de uno en uno. No hay fragmentación externa, al poder haber uno suelto mientras apunte a otro. **La consolidación** mejora las prestaciones colocándolos secuencialmente.
        - Se indica la dirección del primer bloque y cuantos bloques hay encadenados.
        - La tabla que lo almacena es FAT.
    - **Asignación indexada:** Se mantiene una tabla con los identificadores de las unidades de asignación que forman el fichero.
        - Puede ser **por bloques**, apunta a los bloques en orden, o **por porciones**, indica el inicio de las secuencias de bloques consecutivos.
    - **FAT (File Allocation Table):** Tabla que almacena todos los punteros a bloque enlazados de un fichero, apunta a todos lo bloques. El problema es que para llegar a un bloque lejano debe ir avanzando bloque a bloque, muy costoso. Para que no se pierda si falta un enlace, se almacena en varios sitios.

    - **Representación genérica:**
        - Nombre.
        - Atributos.
        - Fecha de creación.
        - Nombre del dueño.
        - Puntero FAT, a la tabla de todos los punteros.
        - Tamaño.

- **UNIX:** Utiliza un nodo i para guardar la información. Permite acceder de una manera mucho más rápida a un determinado bloque gracias a los punteros que actúan como un árbol, ya que se van recorriendo los más pequeños al principio (directos), pero luego los punteros a bloques de punteros permiten avanzar más rápido. Almacena punteros a bloques de punteros, por lo que comenzando desde el nivel más alto puede acceder rápidamente a uno específico recorriendo los niveles del árbol. No tiene fragmentación interna.
- **Nodo i:** Almacena la información de un fichero, pero NO almacena ni el nombre ni el puntero de la posición de lectura/escritura (este último, se almacena en la tabla de ficheros abiertos y puede ser apuntada por varios procesos o cada uno crea su entrada propia).
    - **FORMA DE ÁRBOL.**
    - Tipo de fichero y protección.
    - Usuario propietario del fichero.
    - Grupo propietario del fichero.
    - Tamaño del fichero.
    - Hora y fecha de creación.
    - Hora y fecha del último acceso.
    - Hora y fecha de la última modificación.
    - **Número de enlaces.**
    - **Punteros directos a bloques (10)**. Guarda los 10 primeros punteros a bloque, de esta manera puede acceder rápido a esos específicos, y si necesita ampliar el almacenamiento se usan los punteros a bloques de punteros.
    - **Puntero indirecto simple**. Puntero a bloque de punteros directos
    - **Puntero indirecto doble**. Puntero a bloque de punteros indirectos simples.
    - **Puntero indirecto triple**. Puntero a bloque de punteros indirectos dobles.
- **Estructura NTFS (Node Tree File System)**: Es un árbol B+ de bloques, tiene punteros a bloques de punteros y las hojas son punteros directos a bloque. Para que no perder enlaces se hacen copias de NTFS. Además almacena información sobre el fichero.

### 4.2. Tema 4.2: Directorios

Acceso en Drive a las diapositivas

### 4.3. Tema 4.3: Sistemas de ficheros

Acceso en Drive a las diapositivas