

# Intensivo, Teoría 1

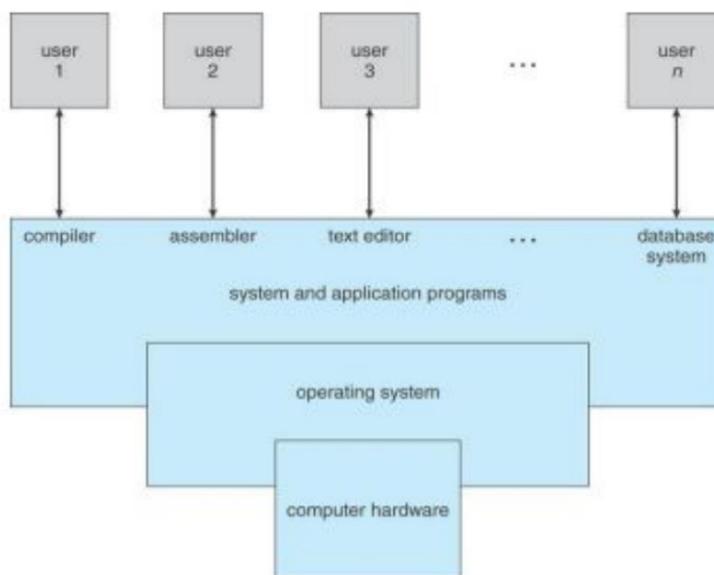
Introducción y Procesos

## 1. Introducción

### 1.1. Definición de Sistema Operativo (OS)

Un sistema operativo es un programa que actúa como intermediario entre el usuario de una computadora y el hardware de la computadora.

Tiene como objetivo ejecutar programas de usuario y hacer uso del hardware de manera eficiente. Internamente define estructuras de datos y algoritmos para administrar el hardware. Externamente: ofrece servicios y funciones para hacer uso del hardware.



### 1.2. Interfaces del OS

Disponemos de dos formas de comunicarnos con el sistema operativo a nivel de usuario, que son las siguientes:

#### 1.2.1. Shell

Es una interfaz de usuario (interfaz humana y de máquina) para acceder a un sistema operativo. Usualmente usan la interfaz gráfica de usuario (GUI) o la interfaz de línea de comandos (CLI). ↳ La consola negra de Linux

Se ha nombrado de esta forma porque es la capa más externa alrededor del núcleo del sistema operativo. Una línea de comandos donde podemos acceder desde el sistema operativo. Lee desde el teclado, crea un proceso y cambia el código del proceso por el que se quiere ejecutar.

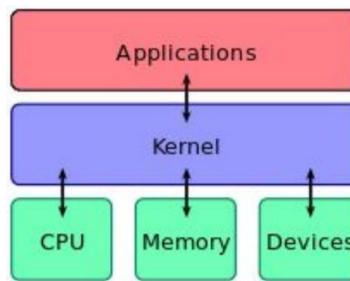
### 1.2.2. GUI (Graphics User Interface)

Es una forma que permite a los usuarios interactuar con dispositivos electrónicos a través de iconos gráficos, indicadores visuales e indicadores de sonido para facilitar la interacción con un sistema informático de una manera más intuitiva que el usuario basado en texto (programación).

Básicamente, todas aquellas aplicaciones basadas en ventanas, que tienen una interfaz gráfica más allá de una simple entrada de texto, usan el GUI.

### 1.3. Kernel

Es el módulo central de un sistema operativo, un programa que primero se carga en la memoria y es responsable de la memoria, el disco, los procesos y la gestión de tareas. Tiene un control completo sobre todo en el sistema.



#### 1.3.1. Interacción con el Kernel

Para poder interactuar con el kernel, debemos usar lo que se conoce como **system-call**.

Básicamente, una **system-call** una llamada al sistema es cómo un proceso solicita un servicio del kernel de un sistema operativo que normalmente no tiene permiso para ejecutar. Por lo tanto, la aplicación lo utiliza para solicitar el servicio al OS.

Principalmente se ejecuta en lo que se llama **modo privilegiado** (modo de administrador en Windows o super-usuario en UNIX). Un proceso debe poder acceder a los servicios proporcionados por el kernel. Esto se implementa de manera diferente para cada kernel, pero la mayoría proporciona una **API** (conjunto de definiciones de subrutinas y herramientas para crear software).

#### Extra:

- ▷ Compilador : De codi a codi màquina
- ▷ Interpret : Directament pel llegir el codi humà

### 1.3.2. Procesos del Kernel

- **Process manager:** Los procesos son la entidad principal existente en el sistema operativo. Un administrador de procesos realiza la gestión general diaria del proceso. Este subsistema asegura que todas las actividades del proceso se realicen correctamente y que cuenten con los recursos adecuados.
- **Storage manager – File manager:** Se refiere a soluciones de software que ayudan a facilitar el almacenamiento de datos, con una organización jerárquica de carpetas y subcarpetas (ruta de la carpeta). En el caso de Windows, tendríamos el *Explorador de archivos*.
- **Device driver management:** Para realizar funciones útiles, los procesos necesitan acceso al periférico conectado al ordenador, que es controlado por el kernel a través de los controladores del dispositivo (drivers).

Un controlador de dispositivo o *driver* es un programa informático que permite que el sistema operativo interactúe con un dispositivo de hardware. El objetivo de diseño de un controlador es la abstracción; la función del controlador es traducir las llamadas de función ordenadas por el sistema operativo (llamadas de programación) en llamadas específicas del dispositivo.

- **Memory management:** El kernel tiene acceso completo a la memoria del sistema y debe permitir que los procesos accedan de manera segura según lo requieran. Para hacerlo, utilizan el direccionamiento de memoria virtual. Esto permite que el kernel, mediante traducción de dirección virtual a física y viceversa, gestionar todo el direccionamiento de forma abstracta.

### 1.3.3. Ejemplos de llamadas al Kernel

Tipo de llamada	Windows	UNIX
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

## 2. Procesos

### 2.1. Definición

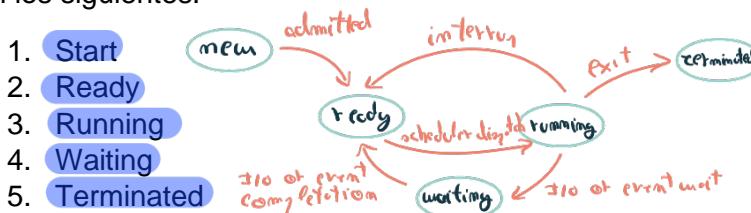
Un proceso (o JOB) es la entidad activa que representa un programa en ejecución, en otras palabras, representa la unidad básica de trabajo que se implementará en el sistema. Un sistema consiste en una colección de procesos (procesos del usuario y del sistema). Pero no son un programa (ente pasivo).

Un proceso se puede dividir en cuatro secciones:

- **Stack:** Contiene los datos temporales, como parámetros de método / función, dirección de retorno y variables locales.
- **Heap:** Memoria asignada dinámicamente a un proceso durante su tiempo de ejecución
- **Text:** Esto incluye la actividad actual representada por el valor del contador del programa y el contenido de los registros del procesador.
- **Data:** contiene las variables globales y estáticas.

### 2.2. Ciclo de vida de un Proceso

Cuando un proceso es ejecutado, pasa por diferentes fases y comportamientos, que son los siguientes:



### 2.3. Modos de gestión de procesos

#### 2.3.1. Multiprogramación

Uno o más programas cargados en la memoria principal que están listos para ejecutarse. Solo un programa a la vez puede obtener la CPU (Unidades de procesamiento central) para ejecutar sus instrucciones. La idea principal es maximizar el uso del tiempo de CPU (mantenerlo ocupado siempre que haya procesos listos para ejecutarse).

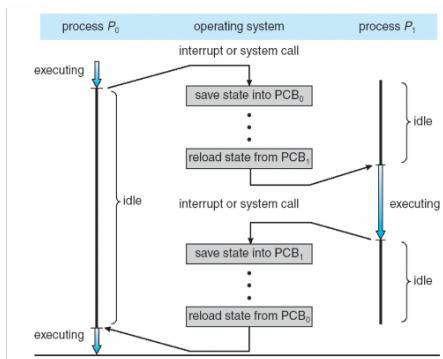
#### 2.3.2. Tiempo compartido (multitarea)

Tener múltiples (programas, procesos, tareas, subprocessos) ejecutándose al mismo tiempo. (Paralelismo logrado cuando la CPU se re-asigna a otra tarea). El cambio de contexto es una parte esencial de las funciones de un sistema operativo multitarea.

Tanto el sistema operativo multiprogramación como el multitarea son sistemas de tiempo compartido (CPU). Sin embargo, mientras que en la multiprogramación (sistemas operativos más antiguos) un programa en su conjunto sigue ejecutándose hasta que se bloquea, en la multitarea (sistemas operativos modernos) el tiempo compartido se manifiesta mejor porque cada proceso en ejecución toma solo una cantidad justa del tiempo de CPU. (Menos de 1 segundo).

Concurrencia Ferir correr diversos procesos simultàniament  
Paralelisme Fer correr diversos processos al mateix moment

Context Switching  
Quan un process acaba, s'ha d'actualitzar i carregar el nou process



### 2.3.3. Sistemas en tiempo real

Describe los sistemas de hardware y software que están sujetos a una "restricción en tiempo real". Los programas en tiempo real deben garantizar la respuesta dentro de las limitaciones de tiempo especificadas, a menudo denominadas "fechas límite". Es un sistema reactivo que debe actuar rápidamente en respuesta a eventos externos.

## 2.4. Bloque de control de procesos

Es una estructura de datos mantenida por el sistema operativo para cada proceso. Cada proceso dentro de esta estructura es identificado por un ID, un número entero llamado PID (Process ID). Es la información asociada a cada proceso.

Otros datos importantes guardados en dicha estructura son:

- **Process State:** El estado actual del proceso. Uno de los vistos en el apartado 2.2.
- **Program Counter:** Es un puntero a la siguiente instrucción a ejecutar del programa actual.
- **CPU registers:** Información sobre los registros de la CPU que el proceso necesita.
- **CPU Scheduling Info:** Guarda la prioridad del proceso y otros datos requeridos por el proceso de **Scheduling**.
- **Memory management info:** Incluye la información de la tabla de paginación, límites de memoria, tabla de segmentos... dependiendo de la memoria utilizada por el sistema operativo.
- **Información del estado de I/O:** Incluye una lista de los dispositivos de I/O alojados en dicho proceso.

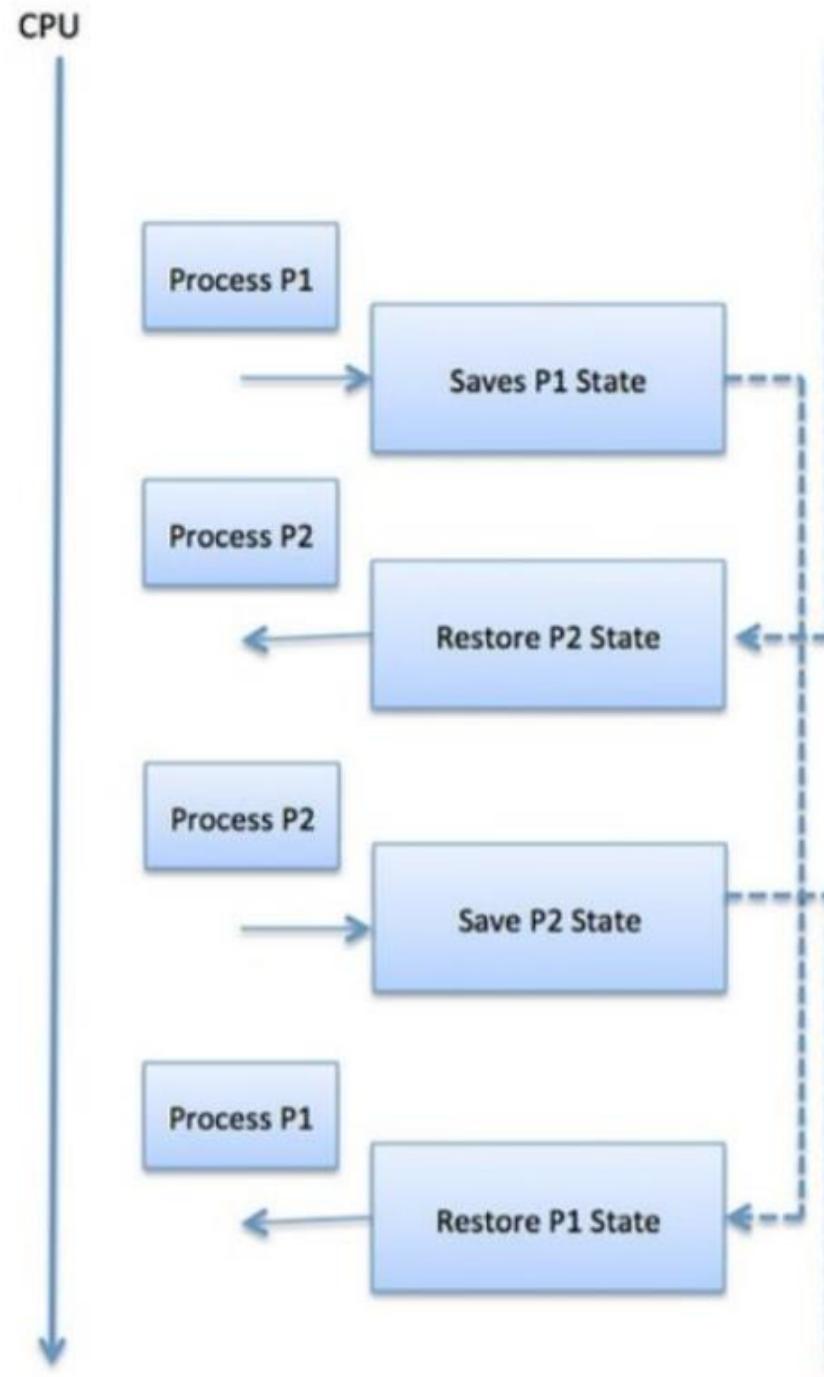
## 2.5. Colas de planificación de procesos

Es el subsistema encargado de mantener todos los Bloques de control de procesos. Consta de 3 colas:

- **Job queue:** Contiene todos los procesos activos del sistema.
- **Ready queue:** Mantiene el conjunto de todos los procesos que residen en la memoria principal, listos y en espera de ejecución. Al crear un proceso, siempre se coloca en esta cola.
- **Device queues:** Contiene los procesos que están bloqueados debido a la falta de disponibilidad de un dispositivo de I/O.

## 2.6. Cambio de contexto

Cuando la CPU cambia a otro proceso en PCB, el sistema debe guardar el estado del proceso anterior y cargar el estado guardado para el nuevo.



## 2.7. Creación de un proceso

La creación de un nuevo proceso es el resultado de una *call-system* de tipo “crear proceso”, que debe ser invocada por otro proceso existente.

De este modo, el nuevo proceso resultante será hijo del proceso que invocó la llamada al sistema, creando así una jerarquía de procesos padre-hijo. Un proceso puede tener de 0 a N hijos, pero solo puede (y debe) tener un proceso padre.

Como ya vimos anteriormente, cada proceso es identificado por un PID. Esto se mantiene en el árbol de procesos.

Existen tres formas de que los procesos comparten recursos entre sí:

- Padre e hijos comparten todos los recursos.
- El padre comparte un subconjunto de sus recursos con sus hijos.
- No comparten recursos de ningún tipo.

Existen dos formas de ejecución:

- El padre y sus hijos se ejecutan de forma concurrida, paralela, asíncrona.
- El padre espera a que sus hijos terminen para continuar su ejecución.

### 2.7.1. Función fork()

La función *fork* nos permite crear un proceso nuevo a partir del proceso actual, creando de esta forma un proceso hijo. Ambos procesos continuarán su ejecución independientemente a partir de la llamada a *fork()*.

Dicha función retorna dos valores diferentes dependiendo de si estamos en el proceso hijo o en el padre:

- Si estamos en el proceso hijo, retornará siempre 0.
- Si estamos en el padre, retornará el PID del proceso hijo recién creado.

### 2.7.2. Función exec()

Función que permite que un proceso ejecute un nuevo programa. Reemplaza el código que el proceso actual está ejecutando, por uno nuevo. Todo cambia (código, pila, contador de programa,...). Se usa después de *fork* para reemplazar la memoria del proceso con un nuevo programa.

### 2.7.3. Funciones write() y read()

Implican llamadas al sistema que nos permiten escribir y leer bytes tanto de archivos como de la propia entrada y salida estándar del sistema. Pueden ser empleadas para intercambiar información entre diferentes procesos.

## 2.8. Finalización de un proceso

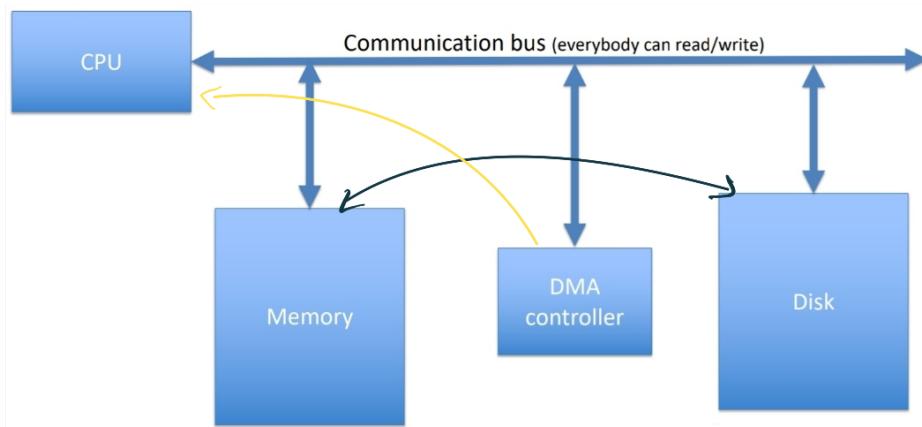
Un proceso puede terminar de forma voluntaria (empleando algunas funciones preparadas para ello) o de forma involuntaria (por algún error ocurrido o una interrupción).

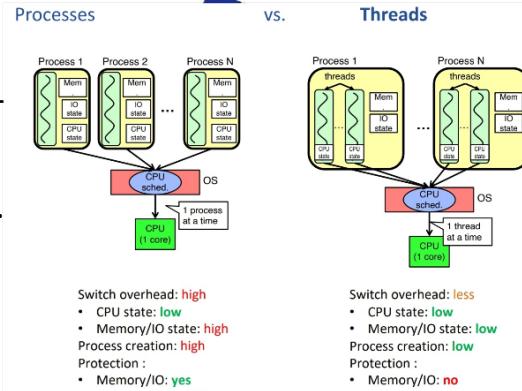
Para finalizar un proceso de forma voluntaria, debe llamarse a la función `exit()`. De esta forma el sistema libera los recursos asociados a dicho proceso, y este termina con normalidad.

La función `exit` tiene un parámetro de tipo entero, cuyo valor puede ser enviado desde un proceso hijo a su proceso padre, y este recuperarlo mediante las funciones `wait()` y `waitpid()`. Dichas funciones congelan el proceso padre hasta que uno de los procesos hijos termine, en el caso de la primera, o hasta que un proceso hijo con PID concreto termine, en el caso de la segunda.

### Accés a la memòria directe

- ① La CPU pregunta a lo controlador del DMA
- ② El controlador DMA copia entre disk i memòria
- ③ El controlador notifica a la CPU.





## 1. Threads

### 1.1. Definición

Un **Thread** (o hilo) es una unidad básica de utilización de la CPU, con su propio contador de programa que realiza un seguimiento de qué instrucción ejecutar a continuación, registros del sistema que contienen sus variables de trabajo actuales y una pila que contiene el historial de ejecución. Es una versión ligera de un proceso y comparten memoria, pero con CPU distintas.

La relación entre hilos y procesos, es la implementación. Un proceso realiza un solo hilo de control. Si un proceso tiene múltiples hilos de control, puede realizar más de una tarea a la vez.

Pueden existir múltiples hilos dentro de un proceso, ejecutándose simultáneamente y compartiendo recursos como la memoria, mientras que diferentes procesos no comparten estos recursos. Un proceso pesado = tarea con un hilo.

### 1.2. Multithreading

Capacidad de una CPU para ejecutar múltiples procesos o hilos simultáneamente, compatible con el sistema operativo.

### 1.3. Threads a nivel de Usuario

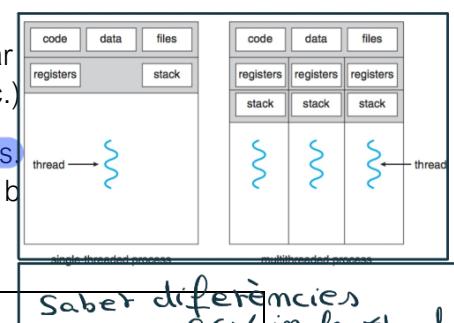
Existen numerosas librerías a disposición del usuario para poder crear función de en que arquitectura nos encontramos (Windows, UNIX, etc.)

En UNIX, concretamente disponemos de la librería: **POSIX PThreads**, muy desarrollada, documentada y cuenta con un set de herramientas b

Veamos como crear un hilo usando PThreads:

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg
);
```

*Li passem la funció que el thread que volem que executi*



Saber diferències entre multi-/single-thread

Esta función nos pide un puntero hacia una estructura de tipo `pthread_t`, la cual contendrá información sobre el thread que queremos crear, y que será rellenada por esta función.

El parámetro “attr” nos es una estructura de atributos, que si no queremos darla, podemos pasar un NULL.

El tercer parámetro ha de ser una función, la cual será la que ejecute el thread. El thread estará vivo mientras le queden instrucciones de dicha función por ejecutar.

Por último, “arg” nos permite enviarle parámetros a la función del thread.

La función `pthread_create` no solo crea un thread sino que lo ejecuta, es decir, en el momento de invocarla, si no ha habido ningún error, se crea el nuevo thread y se ejecuta.

```
int pthread_join(
    pthread_t thread,
    void **retval
);
```

La función `pthread_join` tiene un comportamiento parecido al system-call `wait`. Espera hasta que un hilo finalice.

El primer parámetro nos pide el thread en concreto al que queremos esperar.

El segundo parámetro nos pide un doble puntero, que no es más que un lugar de memoria en el que poder almacenar punteros, y guardar todos los datos que vaya a retornar la función que esté ejecutando el thread.

```
void pthread_exit(
    void *retval
);
```

`Pthread_exit` es similar a la system-call `exit`. Al ser llamada dentro de la función que un thread esté ejecutando, lo finaliza de forma voluntaria.

Se puede insertar un valor de retorno, que será capturado por la función `pthread_join`.

### 1.3.1 Ejemplos con PThreads

```
pthread_t * tids[10]; //creo 10 threads
for (int i=0; i<10; i++) {
    pthread_create(tids[i], NULL, code, NULL);
}
for (int i=0; i<10; i++) {
    pthread_join(tids[i], NULL)
}
```

El código anterior crea un array con 10 threads, después los inicializa todos, enviándoles una función llamada `code`, que contiene el código que deben ejecutar. Por último, espera a que todos ellos finalicen antes de continuar con su ejecución.

```
void *runner(void *param) //the thread
pthread_t id; //the thread identifier
pthread_attr_t attr; //set of attributes for the thread
pthread_attr_init(&attr); //get the default attributes
pthread_create(&tid, &attr, runner, argv[1]); //create the thread
pthread_join(tid, NULL); //wait for the thread to exit
```

En el código anterior podeis observar varias llamadas a las funciones que hemos visto anteriormente, con el añadido de las que controlan los atributos especiales que queráis insertar en el nuevo thread a crear.

## 2. Bloqueo de Threads

### 2.1. Mutex lock

Los **mutexes** (bloqueos de exclusión mutua) se usan para implementar secciones críticas y proteger las estructuras de datos mutables compartidas contra accesos concurrentes.

El uso típico es Mutex.lock y Mutex.unlock, veamos su significado:

- **Mutex.lock:** El mutex queda bloqueado. Solo un hilo puede tener el mutex bloqueado en cualquier momento. Un hilo que intente bloquear un mutex ya bloqueado por otro hilo, se suspenderá hasta que el otro hilo lo desbloquee.
- **Mutex.unlock:** Libera el mutex bloqueado. Si existen otros threads intentando bloquearlo, uno de ellos lo hará, repitiendo de nuevo este ciclo.

Una **race condition** (condición de carrera) en el software es un evento indeseable que ocurre cuando dos o más subprocesos pueden acceder a datos compartidos e intentan cambiarlos al mismo tiempo, en otras palabras, es como si los subprocesos están haciendo una carrera por la CPU para acceder / cambiar los datos. El orden de ejecución afecta el resultado.

En el siguiente ejemplo podemos ver como sincronizar dos threads mediante el uso de un mutex:



Tal y como podemos observar en el ejemplo, si el primer hilo entra, el segundo debe esperar. Si se interrumpe el primer hilo, todavía tendrá derecho a terminar el código, y luego entrará el segundo hilo.

Recordemos que estamos hablando de hilos; los hilos comparten memoria, a diferencia de los procesos. Este ejemplo no sería posible realizarlo de esta forma.

### 2.2. Operaciones atómicas

Los **bloqueos Mutex** se implementan mediante operaciones atómicas o secciones críticas.

**Operaciones atómicas:** Son aquellas que siempre se ejecuta hasta su finalización y no se puede interrumpir. Es indivisible. Si las operaciones atómicas no existieran, entonces no hay forma de hacer que dos hilos funcionen juntos.

**Sección crítica:** Se puede interrumpir hasta que N procesos. (cuando está lleno, lo siguiente tiene que esperar).

Tenemos varias funciones en la librería **PThreads** que nos permiten trabajar con mutex.

**Creación e iniciación de mutex:**

```
int pthread_mutex_init(
    pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr
);
```

**Destrucción de un mutex:**

```
int pthread_mutex_destroy(
    pthread_mutex_t *mutex
);
```

Si el mutex lo tenía otro hilo y éste es destruido, POSIX no define el comportamiento de mutex en esta situación.

Una lock no visible y libre  
de otra variable global

**Solicitud y liberación de un mutex:**

```
int pthread_mutex_lock(pthread_mutex_t * mutex);
int pthread_mutex_trylock(pthread_mutex_t * mutex);
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

Con **pthread\_mutex\_trylock** el hilo siempre retorna, si la función es exitosa, se retorna 0 - como en los otros casos; si no se retornará EBUSY indicando que otro hilo tiene el mutex.

Ejemplo: Para proteger una zona crítica, usar:

```
pthread_mutex_t mylock;
pthread_mutex_init(& mylock, NULL);

pthread_mutex_lock(&mylock);
/* Sección crítica */
pthread_mutex_unlock(&mylock);
```

### 3. Sincronización de Threads

Un Thread puede mantenerse en estado ocupado, esperando, suspendido. Esto puede ser un problema de pérdida de tiempo, durante el cual, con una buena sincronización entre ellos, podría agilizarse todo. *Um semáforo controla um enter no mecanismo*

#### 3.1. Semáforos

Un semáforo es una estructura empleada para la sincronización con dos operaciones atómicas:

- **Wait():** revisar el valor que guarda el semáforo. Si el valor es diferente de 0, puede pasar y el valor lo disminuye en 1.
- **Signal():** es la operación contraria. Cuando se sale de la sección crítica, el contador se incrementa en 1, indicando que se puede pasar de nuevo.

Tipos de semáforos:

- **Semáforos binarios:** Fijan la cantidad de recursos disponibles. (equivalente a una cerradura). Inicializado a 1.
- **Restricciones de programación (Scheduling constraints):** Inicializado a 0. Permite que el hilo 1 espere una señal del hilo 2. El hilo 2 programa el hilo 1 cuando se cumple una restricción dada.

#### 3.2. Monitores

Como los semáforos, pero sin restricciones ni condiciones. Tiene lo siguiente:

- **Lock:** para administrar la variable que guarda el monitor.
- **Variable de condición:** cola de subprocessos en espera de ingresar a la sección crítica.

De esta forma, nunca hay threads en estado de espera.

Operaciones:

- **wait(&lock):** Apartas el lock pero garantizando de que lo tendrás cuando quieras seguir.
- **signal():** Si hay alguien en cola, lo despierta para que pueda seguir.
- **notifyall() / broadcast():** Como una señal, pero en lugar de despertar a uno, despierta toda la cola.

### Implementación de MESA:

#### Locks:

```
pthread_mutex_lock  
pthread_mutex_unlock
```

#### Condition:

```
pthread_cond_wait(cond, lock); //Takes the thread and puts it in  
queue  
pthread_cond_signal(cond); Implementation of the variables  
locks, unlocks and condition variables (queue).
```

Implementación de las variables bloqueos, desbloqueos y variables de condición (cola).

**Deadlocks:** Cuando necesitamos administrar varios bloqueos al mismo tiempo y más de un hilo necesita acceder al mismo bloqueo en un orden diferente y se bloquean entre sí.

```
if ( from.id < to.id ){ //Solution to deadlocks  
    pthread_mutex_lock(from->mutex);  
    pthread_mutex_lock(to->mutex);  
}  
else{  
    pthread_mutex_lock(to->mutex);  
    pthread_mutex_lock(from->mutex);
```

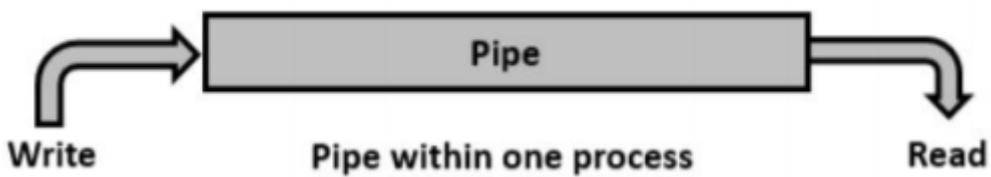
# Intensivo, Teoría 3

Comunicación y control de memoria

## 1. Comunicación

### 1.1 Pipes

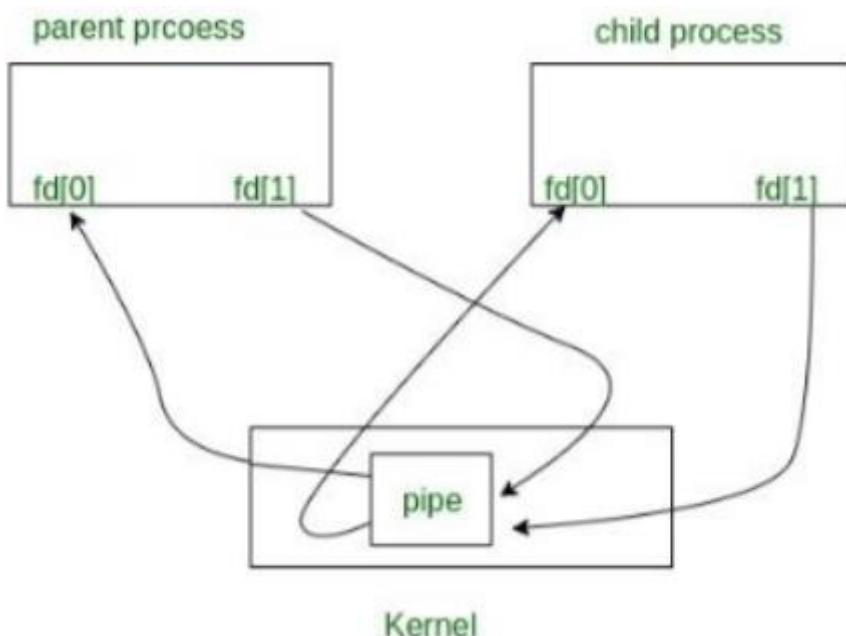
Una **Pipe** (tubería de comunicación) es un medio entre dos o más procesos relacionados (o no). Puede estar dentro de un proceso o en una comunicación entre los procesos hijo y padre. La comunicación se logra mediante un proceso de escritura en la tubería y otra lectura de la tubería.



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

Esta *system-call* crearía una tubería para la comunicación unidireccional, es decir, crea dos descriptores, el primero está conectado para leer desde la tubería (fd [0]) y el otro está conectado para escribir en la tubería (fd [1]).

La función retornaría cero en caso de éxito y -1 en caso de falla. Un ejemplo de cómo se establecería dicha comunicación es el siguiente:



## 1.2. Sockets

Los **Sockets** proporcionan comunicación bidireccional punto a punto entre dos procesos. Se puede asociar con uno o más procesos. Un socket se identifica mediante una dirección IP concatenada con un número de puerto.

Usan una arquitectura cliente-servidor:

- El servidor espera las solicitudes de clientes entrantes escuchando un puerto específico.
- Una vez que se recibe una solicitud, el servidor acepta una conexión del socket del cliente para completar la conexión.
- Todas las conexiones deben ser únicas.

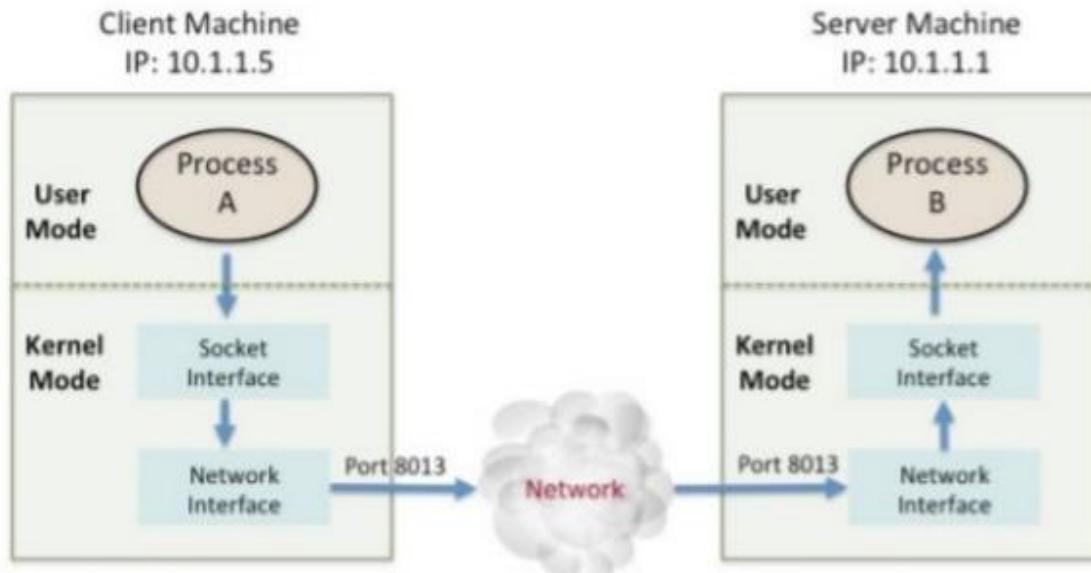
**Unix domain sockets:** Punto final para la comunicación, más similar a las tuberías y utilizado en la misma máquina.

**Dominio de Internet:** Comunicación de máquinas a través de la red.

- **Datagram Socket**
- **Socket de flujo:** Los sockets de flujo funcionan como una conversación telefónica y proporcionan un flujo de datos bidireccional y confiable sin límites de registro. Este flujo de datos también es secuenciado y no duplicado.

Un socket se identifica mediante una dirección IP con un número de puerto. El servidor con una IP y escucha los mensajes en un puerto de la máquina. El cliente envía un mensaje a una IP específica (o un broadcast) y a un puerto.

En el siguiente ejemplo se muestra el funcionamiento de un socket:



### 1.2.1 Ejemplo de sockets

#### Cliente:

1. Crear el socket:

```
int socket(int domain, int type, int protocol)
```

2. Conectamos el socket a la dirección del servidor. El cliente inicia una conexión al socket del servidor:

```
int connect(int s, struct sockaddr *name, int namelen)
```

3. Enviamos y recibimos datos usando:

```
int send(int s, const char *msg, int len, int flags)
int recv(int s, char *buf, int len, int flags)
```

#### Servidor:

1. Creamos el socket:

```
int socket(int domain, int type, int protocol)
```

2. Enlazamos el socket a una dirección (Número de puerto y el host de la máquina):

```
int bind(int s, const struct sockaddr *name, int namelen)
```

3. Escuchamos peticiones de conexión:

```
int listen(int s, int backlog)
```

4. Aceptamos una conexión y quedamos suspendidos hasta que el cliente se conecte:

```
int accept(int s, struct sockaddr *addr, int *addrlen)
```

5. Enviamos y recibimos datos usando las mismas funciones que en el cliente.

### 1.3. Archivos

Un archivo es una colección de datos, que pueden ser binarios o no, y que están alojados en el disco físico (HHD, SSD, M.2, etc.)

Podemos acceder a su contenido de forma directa o de forma secuencial.

#### File lseek():

```
off_t lseek(int fildes, off_t offset, int whence)
lseek(file, 10, SEEK_SET)
```

1. **int fildes:** El descriptor de archivo del puntero que se va a mover
2. **off\_t offset:** El desplazamiento del puntero (en bytes).
3. **wence:** El método en el que se debe interpretar el desplazamiento.
  - a. **SEEK\_SET:** La compensación se debe medir en términos absolutos. (solo usa esto).
  - b. **SEEK\_CUR:** El desplazamiento se medirá en relación con la ubicación actual del puntero.
  - c. **SEEK\_END:** El desplazamiento se medirá en relación con el final del archivo.

Esta *system-call* nos permite cambiar la localización del puntero de escritura/lectura que tenemos sobre un archivo, sobre su **File Descriptor**.

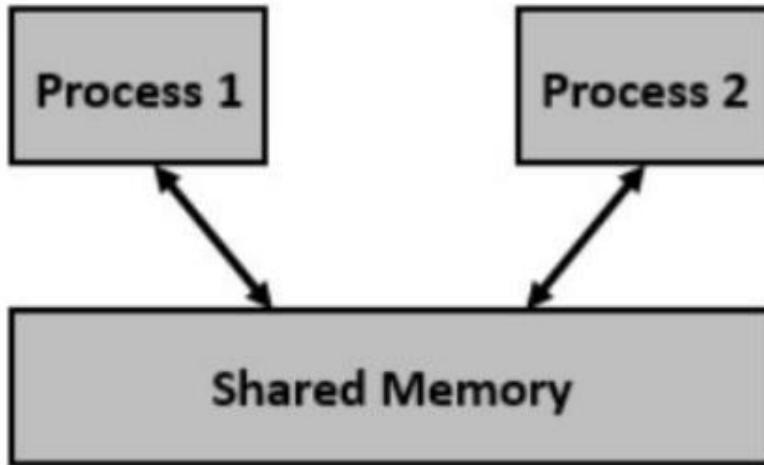
Podemos abrir un archivo en diferentes modos, dependiendo de que queremos hacer con él:

- **O\_RDONLY:** Abrir en modo lectura únicamente.
- **O\_WRONLY:** Abrir en modo escritura únicamente.
- **O\_RDWR** = Abrir en modo escritura y lectura.
- **O\_APPEND** = Abrir en modo escritura sin descartar el contenido previo.
- **O\_CREAT** = Crear un archivo si este previamente no existía.

## 1.4. Memoria compartida (Shared Memory)

Memoria compartida entre dos o más procesos.

Cada proceso tiene su propio espacio de direcciones. Si algún proceso desea comunicarse con cierta información desde su propio espacio de direcciones a otros procesos, entonces solo es posible con IPC (comunicación entre procesos).



**Crear un segmento de memoria compartida o usar uno ya creado:**

```
int shmget(key_t key, size_t size, int shmflg)
```

- **key:** Identificador del segmento de memoria compartida.
- **size:** Tamaño del segmento de memoria compartida redondeado a múltiplos
- **shmflg:** Especifica los indicadores de memoria compartida necesarios, como crear un nuevo segmento (IPC\_CREAT) o crear un nuevo segmento y la llamada falla si el segmento ya existe (IPC\_EXCL).

**Adjunte el proceso al segmento de memoria compartida ya creado:**

```
void* shmat(int shmid, const void *shmaddr, int shmflg)
```

- **shmid:** Identificador del segmento de memoria compartida. Valor de retorno de shmget().
- **shmaddr:** Especifica la dirección adjunta. Si es NULL, elije la dirección más adecuada para adjuntar el segmento.
- **shmflg:** Especifica los marcadores de memoria compartida requeridos.

**Separé el proceso del segmento de memoria compartida ya conectado:**

```
int shmdt(const void *shmaddr)
```

**Operaciones de control en el segmento de memoria compartida:**

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

## 2. Control de memoria (Memory management)

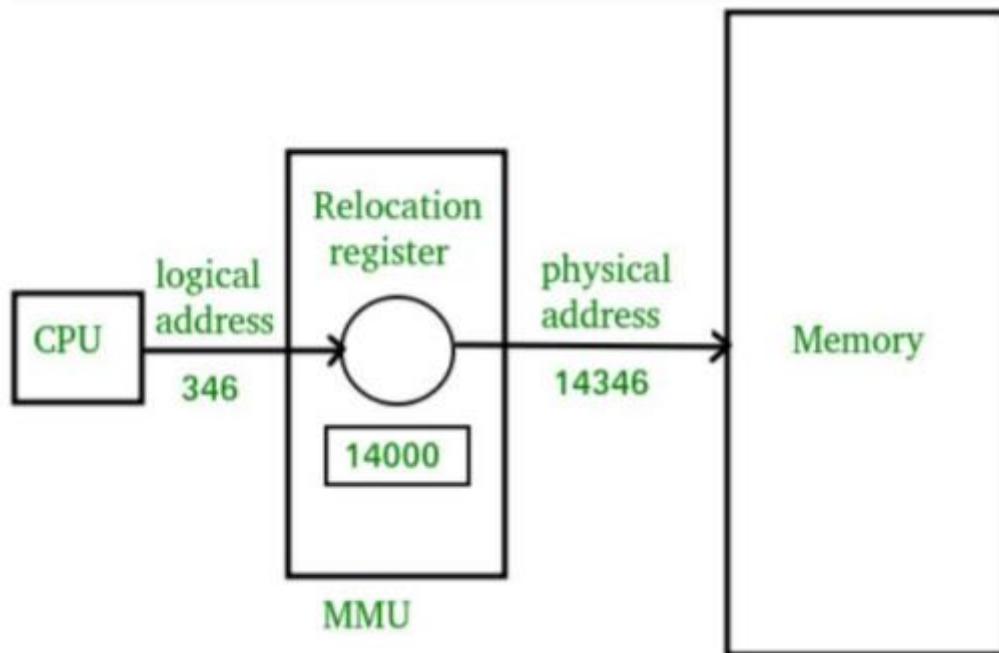
### 2.1. Definición y conceptos:

La unidad de control de memoria (o Memory-Management-Unit, MMU) Es una unidad o sistema encargado de mapear y redireccionar todas las direcciones virtuales hacia sus direcciones reales/físicas.

**Dirección virtual o lógica:** Es generada por la CPU mientras se ejecuta un programa. Esta dirección se utiliza como referencia para acceder a la ubicación de la memoria física. No pueden acceder directamente a la memoria.

**Dirección física:** Identifica una ubicación física de los datos requeridos en la memoria. El usuario nunca trata directamente con la dirección física, pero puede acceder por su dirección lógica (virtual) correspondiente.

El siguiente esquema muestra como funcionaría el direccionamiento empleando la MMU:



## 2.2. Espacio discontinuo

### 2.2.1 Esquemas de paginación

La paginación es un esquema de administración de memoria. Permite que un proceso se almacene en una memoria de manera no contigua. El proceso de almacenamiento de manera no contigua resuelve el problema de la fragmentación externa.

Para implementar la paginación, los espacios de memoria física y lógica se dividen en los mismos bloques de tamaño fijo. Estos bloques de memoria física de tamaño fijo se denominan **frames**, y los bloques de memoria lógica de tamaño fijo se denominan **pages**.

Cuando es necesario ejecutar un proceso, las páginas del proceso desde el espacio de memoria lógica se cargan en los marcos del espacio de direcciones de la memoria física. Ahora la dirección generada por la CPU para acceder al marco se divide en dos partes, es decir, número de página y desplazamiento de página.

### 2.2.2. Segmentación

La segmentación también es un esquema de gestión de memoria. Es compatible con la vista del usuario de la memoria. El proceso se divide en segmentos de tamaño variable y se carga en el espacio de direcciones de la memoria lógica.

El espacio de direcciones lógicas es la colección de segmentos de tamaño variable. Cada segmento tiene su nombre y longitud. Para la ejecución, los segmentos del espacio de memoria lógica se cargan en el espacio de memoria física.

## 2.3. Memoria virtual

**Estrategias de gestión de memoria:** Mantener muchos procesos en la memoria simultáneamente para permitir la multiprogramación y los sistemas concurrentes.

La memoria virtual es una técnica que permite la ejecución de procesos que no están completamente en la memoria.

Memoria virtual de página de demanda (las páginas solo se cargan cuando se exigen durante la ejecución del programa)

**Page fault:** Acceso a una página que no es válida (está marcada como invalida).

**Page replacement:** Si no hay ningún marco libre, buscamos uno que no se esté utilizando actualmente y lo liberamos. Algoritmos utilizados:

- **FIFO:** Reemplaza la página con el tiempo más antiguo en que se guardó en la memoria.
- **OPT:** Reemplaza la página que no se utilizará durante el período de tiempo más largo.
- **LRU:** Reemplaza la página que no se ha utilizado durante el período de tiempo más largo.

## 2.4. Sistema de archivos (File-system)

El **sistema de archivos** o **sistema de ficheros** es el componente del sistema operativo encargado de administrar y facilitar el uso de las memorias periféricas, ya sean secundarias o terciarias.

Sus principales funciones son la asignación de espacio a los archivos, la administración del espacio libre y del acceso a los datos resguardados. Estructuran la información guardada en un dispositivo de almacenamiento de datos o unidad de almacenamiento (normalmente un disco duro de una computadora), que luego será representada ya sea textual o gráficamente utilizando un gestor de archivos.

La mayoría de los sistemas operativos manejan su propio sistema de archivos.

Lo habitual es utilizar dispositivos de almacenamiento de datos que permiten el acceso a los datos como una cadena de bloques de un mismo tamaño, a veces llamados sectores, usualmente de 512 bytes de longitud (también denominados **clusters**).

El software del sistema de archivos es responsable de la organización de estos sectores en archivos y directorios y mantiene un registro de qué sectores pertenecen a qué archivos y cuáles no han sido utilizados.

En la práctica, un sistema de archivos también puede ser utilizado para acceder a datos generados dinámicamente, como los recibidos a través de una conexión de red de computadoras (sin la intervención de un dispositivo de almacenamiento).