Sistemas Operativos y Distribuidos

Iren Lorenzo Fonseca
iren.fonseca@.ua.es

TEMA 1. Sistemas Operativos.

Sistema de Archivos, Concurrencia e IPC

Tema 1.3 Sistema de Archivos, Concurrencia e IPC

Contenidos

Sistema de archivos
Estructura, Interfaz

Concurrencia
Definición, problemas y sincronización

IPC
Señales, tuberías, memoria compartida

Sistema de Archivos

Contenidos

- Introducción
- Interfaz con el Sistema de Archivos
- Implementación del Sistema de Archivos

Introducción

Definiciones básicas

SA

- El ordenador para ejecutar programas necesita tenerlos en memoria principal
- Como la capacidad de memoria principal es reducida y además es volátil es necesario un Sistema de Almacenamiento Secundario (SAS) para contener los programas y datos que se vayan a utilizar
- El SO es el software encargado de la gestión del SAS
    - ✓ Abstrae las propiedades físicas de los dispositivos de almacenamiento
    - ✓ Proporciona una interfaz a los usuarios de acceso a la información
    - ✓ Administra el espacio libre
    - ✓ Realiza la asignación del almacenamiento

Introducción

Definiciones básicas

SA

El sistema de archivos está formado por:

- **Colección de archivos**. Cada uno de los cuales contiene datos relacionados.
- **Estructura de directorios**. Organiza todos los archivos del sistema y proporciona información sobre ellos.
- **Particiones**. Permite separar grandes colecciones de directorios.

Interfaz con el Sistema de Archivos

Interfaz con el Sistema de Archivos

Archivos

Una abstracción para la manipulación de memoria secundaria ofrecida por el sistema operativo

Una colección de información relacionada que reside en el almacenamiento secundario a la cual se le asigna un nombre.

Una secuencia de bits, bytes, líneas o registros con un significado definido por el creador y el usuario.

Interfaz con el Sistema de Archivos

SA

Archivos

Estructura lógica

La estructura lógica se refiere a cómo los datos de un archivo están organizados y presentados para los usuarios y las aplicaciones.

Desde esta perspectiva, se pueden distinguir diferentes formas de organización dependiendo del tipo de archivo

Estructura física

La estructura física se refiere a cómo se almacenan los archivos en el dispositivo de almacenamiento, como un disco duro o una unidad SSD.

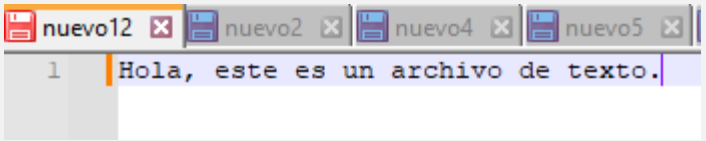
Esta estructura está más relacionada con la administración del almacenamiento por parte del sistema operativo.

Interfaz con el Sistema de Archivos

SA

Archivos

Secuencia de bytes. Por ejemplo, un archivo de texto es una secuencia de caracteres organizados en líneas.



Secuencia de registros. En algunos archivos, los datos están organizados en registros, que pueden ser de tamaño fijo o variable (ejemplo csv).

1, Juan, 25
2, María, 30
3, Pedro, 22

Estructura compleja. Están compuestos por diferentes tipos de datos organizados de manera estructurada y que requieren interpretaciones específicas para ser comprendidos por el sistema operativo o las aplicaciones. (ejemplo archivos ejecutables, archivos de formato gráfico)

Interfaz con el Sistema de Archivos

SA

Archivos

Estructura lógica

La estructura lógica se refiere a cómo los datos de un archivo están organizados y presentados para los usuarios y las aplicaciones.

Desde esta perspectiva, se pueden distinguir diferentes formas de organización dependiendo del tipo de archivo

Estructura física

La estructura física se refiere a cómo se almacenan los archivos en el dispositivo de almacenamiento, como un disco duro o una unidad SSD.

Esta estructura está más relacionada con la administración del almacenamiento por parte del sistema operativo.

Interfaz con el Sistema de Archivos

Archivos

Secuencia de bloques:

- A nivel físico, los archivos se almacenan como una secuencia de bloques en el dispositivo de almacenamiento.
- Un bloque es la unidad mínima de almacenamiento que el sistema operativo maneja.
- En los discos duros tradicionales, estos bloques corresponden a los sectores del disco.
- Los bloques tienen un tamaño fijo (por ejemplo, 4 KB) y un archivo puede ocupar uno o más bloques, dependiendo de su tamaño.

Interfaz con el Sistema de Archivos

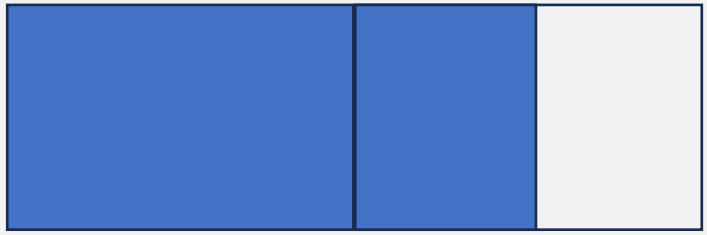
Conversión de bloques lógicos a bloques físicos:

- Los archivos son representados lógicamente como una secuencia de bytes o registros, pero físicamente están distribuidos en varios bloques dentro del disco.
- El sistema operativo se encarga de realizar la conversión entre los bloques lógicos (es decir, la visión secuencial de los datos por parte del usuario) y los bloques físicos (cómo se almacenan realmente los datos en el dispositivo).
- Esta conversión es transparente para el usuario, pero es fundamental para que el sistema operativo gestione correctamente el almacenamiento y acceso a los archivos.

Interfaz con el Sistema de Archivos

Fragmentación Interna

- La fragmentación interna ocurre cuando los archivos no utilizan completamente el espacio asignado en los bloques. Dado que los bloques tienen un tamaño fijo, si un archivo no llena completamente un bloque, se desperdicia espacio.



Interfaz con el Sistema de Archivos

SA

Archivos

Estructura lógica

La estructura lógica se refiere a cómo los datos de un archivo están organizados y presentados para los usuarios y las aplicaciones.

Desde esta perspectiva, se pueden distinguir diferentes formas de organización dependiendo del tipo de archivo

Estructura física

La estructura física se refiere a cómo se almacenan los archivos en el dispositivo de almacenamiento, como un disco duro o una unidad SSD.

Esta estructura está más relacionada con la administración del almacenamiento por parte del sistema operativo.

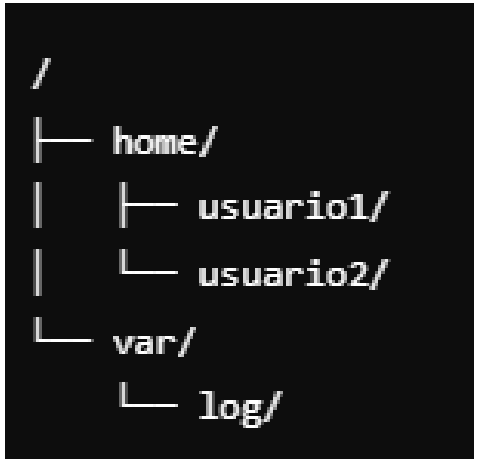
Interfaz con el Sistema de Archivos

Directorios

Directorio: Es una estructura que contiene una lista de archivos y otros directorios

Organización Jerárquica de Archivos:

- La organización de archivos y directorios es jerárquica.
- En sistemas operativos como Unix/Linux, esta jerarquía tiene un punto de partida llamado "directorio raíz" (/), desde el cual se organizan todos los archivos y directorios.



Interfaz con el Sistema de Archivos

SA

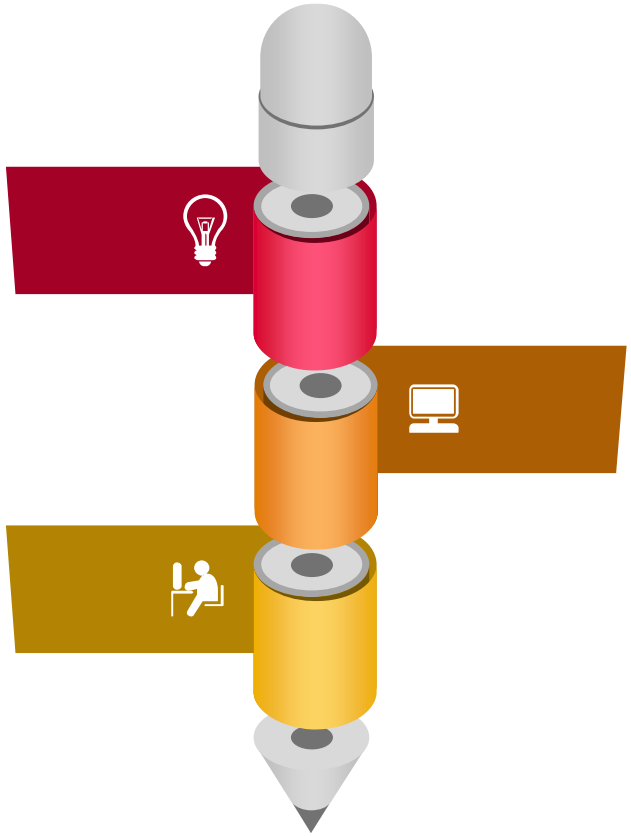
Datos

Metadatos

Información adicional sobre el archivo, como su tamaño, fecha de modificación, permisos de acceso, entre otros.

Nombres de archivos

Cada archivo y directorio tiene un nombre único dentro de su directorio. Algunos sistemas operativos permiten espacios y caracteres especiales en los nombres, pero otros tienen restricciones



Extensiones

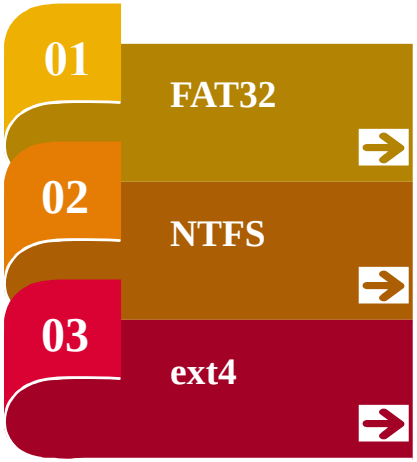
Son sufijos que indican el tipo de archivo, como .txt para un archivo de texto o .jpg para imágenes

Interfaz con el Sistema de Archivos

Estructura Interna

Aunque la base de los Sistemas de archivos son los directorios y los archivos, existen diferentes propuestas para estructurar los bloques que contienen la información y para almacenar y recuperar la información de gestión.

Sistemas de Archivos Comunes:



FAT32

Un sistema de archivos más antiguo utilizado en dispositivos portátiles, como memorias USB. Soporta archivos de hasta 4 GB

NTFS

Sistema de archivos de Windows, con soporte para archivos grandes, permisos avanzados y funciones como la compresión de archivos

Ext4

Común en distribuciones Linux, permite volúmenes de hasta 1 exabyte y archivos de hasta 16 TB. También tiene características de journaling, que previenen la corrupción de datos en caso de fallos.

Interfaz con el Sistema de Archivos

02 NTFS

Estructura Interna

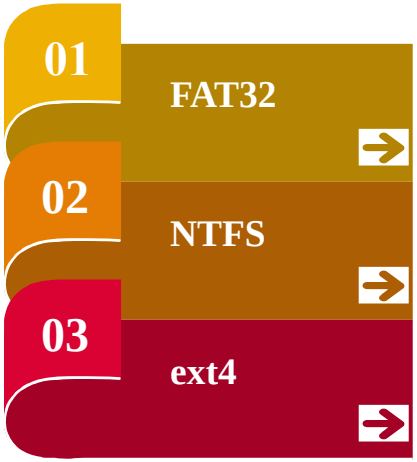
- NTFS organiza los datos en una estructura jerárquica de archivos y carpetas, utilizando una estructura llamada **Master File Table (MFT)**.
- Cada archivo tiene una entrada en el MFT, que contiene no solo metadatos sino también la ubicación física del archivo en el disco. En algunos casos, si un archivo es pequeño, los datos reales del archivo pueden almacenarse directamente en el MFT, lo que mejora el rendimiento.
- Si los datos son más grandes, el MFT contiene punteros que apuntan a los bloques donde los datos del archivo se almacenan físicamente en el disco.

Interfaz con el Sistema de Archivos

Estructura Interna

Aunque la base de los Sistemas de archivos son los directorios y los archivos, existen diferentes propuestas para estructurar los bloques que contienen la información y para almacenar y recuperar la información de gestión.

Sistemas de Archivos Comunes:



FAT32

Un sistema de archivos más antiguo utilizado en dispositivos portátiles, como memorias USB. Soporta archivos de hasta 4 GB

NTFS

Sistema de archivos de Windows, con soporte para archivos grandes, permisos avanzados y funciones como la compresión de archivos

Ext4

Común en distribuciones Linux, permite volúmenes de hasta 1 exabyte y archivos de hasta 16 TB. También tiene características de journaling, que previenen la corrupción de datos en caso de fallos.

Interfaz con el Sistema de Archivos

Estructura Interna

03 ext4

ext4 es un sistema de archivos moderno utilizado principalmente en distribuciones de **Linux**. Es la cuarta versión de la familia **ext** (Extended Filesystem) y ofrece características avanzadas como soporte para archivos grandes, eficiencia en la gestión de espacio y mecanismos para prevenir la corrupción de datos.

Estructura de ext4:

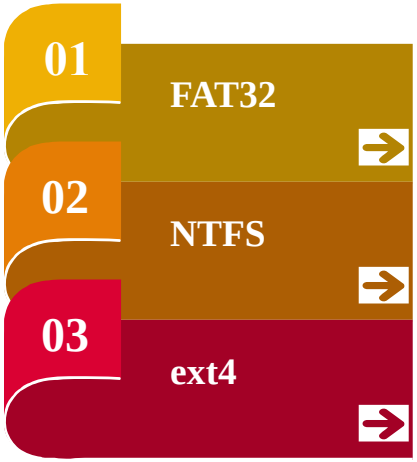
- **Bloques**: ext4 organiza el disco en **bloques**
- **Inodos**: utiliza una estructura llamada **inodo** para almacenar información sobre los archivos. Cada archivo tiene un inodo que contiene:
  - Propietario del archivo, permisos, tamaños y marcas de tiempo.
  - Punteros a los bloques de disco donde están almacenados los datos.

Interfaz con el Sistema de Archivos

Estructura Interna

Aunque la base de los Sistemas de archivos son los directorios y los archivos, existen diferentes propuestas para estructurar los bloques que contienen la información y para almacenar y recuperar la información de gestión.

Sistemas de Archivos Comunes:



FAT32

Un sistema de archivos más antiguo utilizado en dispositivos portátiles, como memorias USB. Soporta archivos de hasta 4 GB

NTFS

Sistema de archivos de Windows, con soporte para archivos grandes, permisos avanzados y funciones como la compresión de archivos

Ext4

Común en distribuciones Linux, permite volúmenes de hasta 1 exabyte y archivos de hasta 16 TB. También tiene características de journaling, que previenen la corrupción de datos en caso de fallos.

Interfaz con el Sistema de Archivos

Operaciones

Operación	Descripción	Ejemplo en UNIX
Crear archivo	El archivo se crea sin datos. Hay que encontrar espacio en el disco y anotar su existencia.	fd = creat(filename, mode)
Leer archivo	Lee datos desde un archivo abierto, copiándolos en un buffer.	read(fd, buffer, nbytes)
Escribir en archivo	Escribe datos desde un buffer hacia el archivo.	write(fd, buffer, nbytes)
Borrar archivo	Elimina un archivo del sistema de archivos.	unlink(filename)
Rebobinado de archivo	Cambia el puntero del archivo a una nueva posición, útil para leer o escribir en diferentes lugares.	lseek(fd, offset, where)
Truncar archivo	Se utiliza para reducir el tamaño de un archivo, generalmente vaciándolo.	fd = creat(filename, mode)
Obtener atributos	Obtiene la información del archivo, como tamaño, permisos, etc.	fd = fstat(filename, statbuf)

¿Que se repite en todas las llamadas?

Interfaz con el Sistema de Archivos

Descriptores de Archivos (fd)

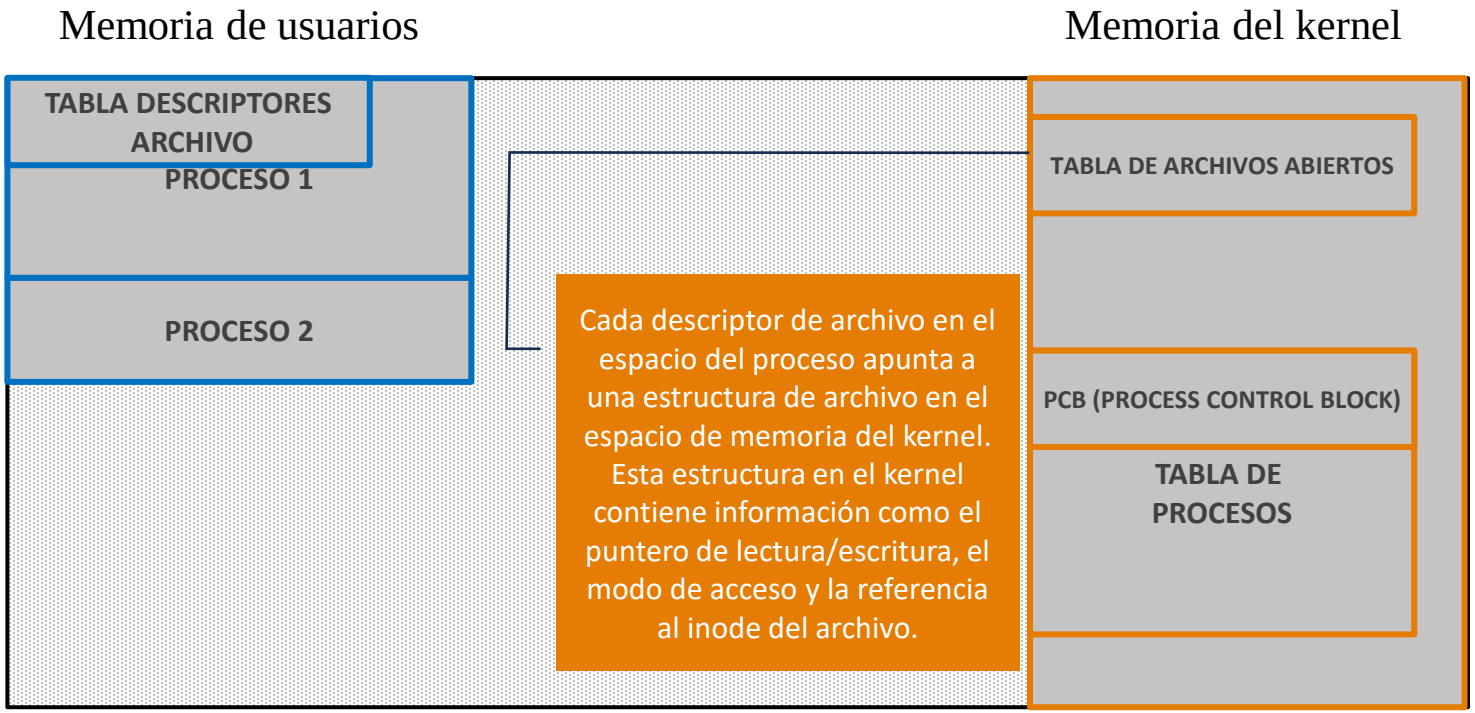
- Un descriptor de archivo es un número entero que el sistema operativo asigna a un archivo o recurso cuando se abre. Este número se usa en las llamadas al sistema como read(), write(), close(), etc., para referirse al archivo o recurso en cuestión.

- Por ejemplo, cuando se abre un archivo en un programa en C con open(), el SO devuelve un descriptor de archivo, que es un número único dentro del contexto del proceso. El programa usa este número para interactuar con el archivo.

Interfaz con el Sistema de Archivos

Tabla de descriptores de archivos

- La tabla de descriptores de archivo es una estructura clave gestionada por el SO para facilitar la interacción entre los programas y los archivos o dispositivos de almacenamiento.
- Cada vez que un programa abre un archivo, el SO asigna un descriptor de archivo (file descriptor), que actúa como un identificador que el programa usa para realizar operaciones de lectura, escritura o cierre sobre ese archivo.



Interfaz con el Sistema de Archivos

Ejemplos

Descriptores 0, 1, 2: Son los descriptores estándar (stdin, stdout, stderr), presentes por defecto en cada proceso. Apuntan a sus respectivas entradas en la tabla de archivos abiertos del kernel.

TABLA DESCRIPTORES ARCHIVO

FD	Archivo	Referencia en la Tabla de Archivos Abiertos (kernel)
0	stdin (entrada estándar)	Entrada 0 en la tabla del kernel
1	stdout (salida estándar)	Entrada 1 en la tabla del kernel
2	stderr (error estándar)	Entrada 2 en la tabla del kernel
3	/home/user/file.txt	Entrada 3 en la tabla del kernel

Entrada en el kernel	Archivo	Posición de Lectura/Escritura	Modo de Apertura	Referencia al inode	Contador de Referencias
0	stdin	0 (lectura en espera)	O_RDONLY (solo lectura)	Inode 1	1
1	stdout	0 (para escribir)	O_WRONLY (solo escritura)	Inode 2	1
2	stderr	0 (para escribir)	O_WRONLY (solo escritura)	Inode 3	1
3	/home/user/file.txt	120 (posición de lectura)	O_RDWR (lectura y escritura)	Inode 454	2

TABLA DE ARCHIVOS ABIERTOS

Interfaz con el Sistema de Archivos

Operaciones

Operación	Descripción	Ejemplo en UNIX
Crear archivo	El archivo se crea sin datos. Hay que encontrar espacio en el disco y anotar su existencia.	fd = creat(filename, mode)
Leer archivo	Lee datos desde un archivo abierto, copiándolos en un buffer.	read(fd, buffer, nbytes)
Escribir en archivo	Escribe datos desde un buffer hacia el archivo.	write(fd, buffer, nbytes)
Borrar archivo	Elimina un archivo del sistema de archivos.	unlink(filename)
Rebobinado de archivo	Cambia el puntero del archivo a una nueva posición, útil para leer o escribir en diferentes lugares.	lseek(fd, offset, where)
Truncar archivo	Se utiliza para reducir el tamaño de un archivo, generalmente vaciándolo.	fd = creat(filename, mode)
Obtener atributos	Obtiene la información del archivo, como tamaño, permisos, etc.	fd = fstat(filename, statbuf)

¿Que se repite en todas las llamadas?

Interfaz con el Sistema de Archivos

SA

Sesiones

Para utilizar un archivo hay que definir una sesión con las llamadas al sistema open (abrir) y close (cerrar).

```
void main(){

        int fd = open("archivo.txt", O_RDWR);

        // uso del archivo
        …

        close(fd)
}
```

1. El proceso llama a open(filename, mode) para abrir el archivo "filename.txt" y el modo de apertura O_RDWR (lectura y escritura)
2. El kernel toma el control, localiza el archivo en el sistema de archivos y verifica permisos.
3. Si todo es correcto, el kernel:
    1. Crea una entrada en la tabla de archivos abiertos del kernel.
    2. Asigna un descriptor de archivo en la tabla de descriptores del proceso.
4. El descriptor de archivo se devuelve al proceso, que ahora puede usarlo para operar sobre el archivo (si falla open devuelve -1)

Interfaz con el Sistema de Archivos

SA

Sesiones

Para utilizar un archivo hay que definir una sesión con las llamadas al sistema open (abrir) y close (cerrar).

```
void main(){

        int fd = open("archivo.txt", O_RDWR);

        // uso del archivo
        …

        close(fd)
}
```

1. El proceso llama a close(fd) para cerrar el descriptor de archivo
2. El kernel verifica que fd sea un descriptor válido.
3. El kernel decrementa el contador de referencias del archivo asociado.
4. Si el contador llega a cero, el kernel libera los recursos asociados.
5. close() devuelve 0 en caso de éxito o -1 en caso de error.

Interfaz con el Sistema de Archivos

SA

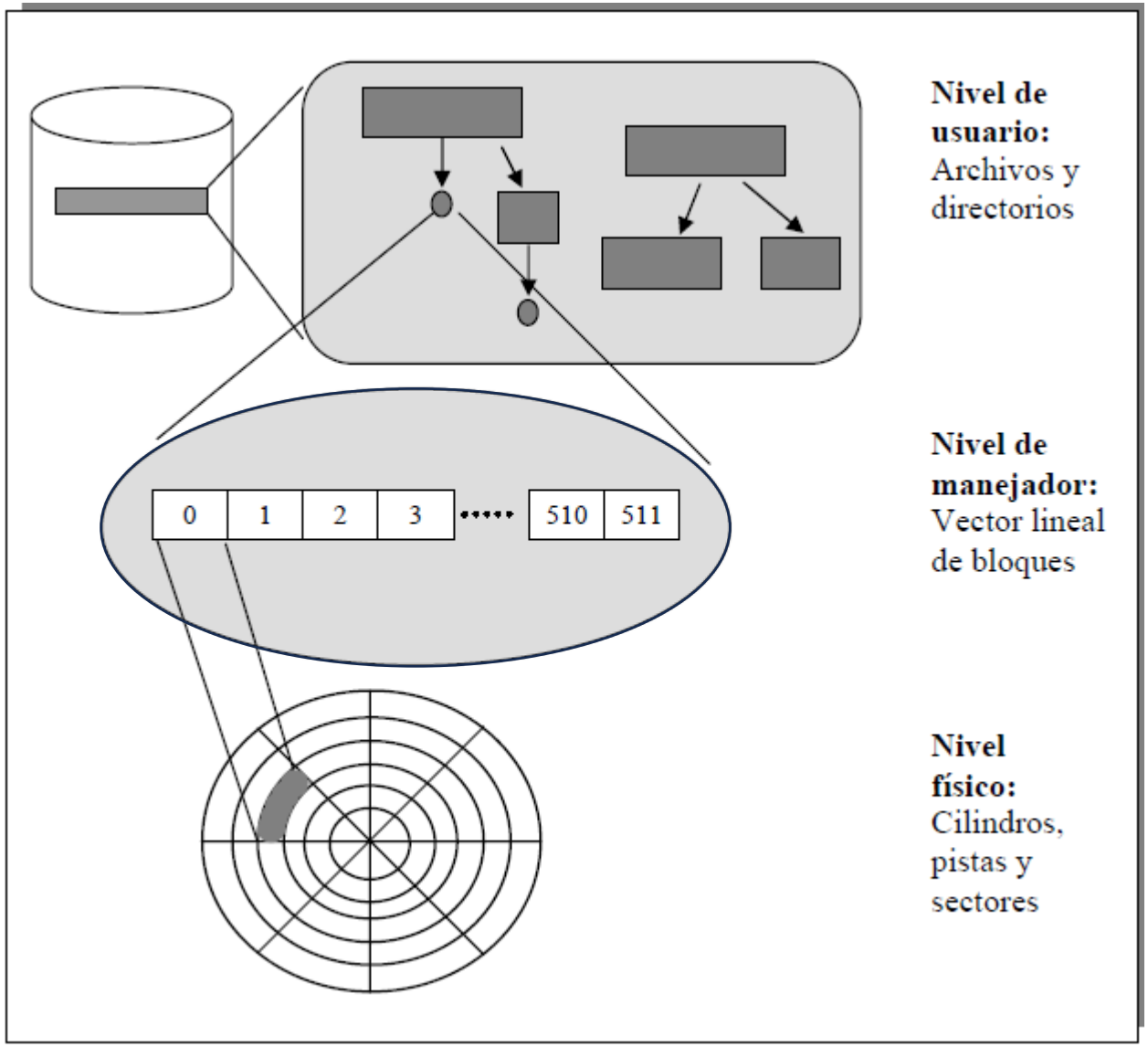
Ejemplo

```c
int main() {
    int fd = open("archivo.txt", O_RDONLY);  // Abrir el archivo en modo solo lectura
    if (fd == -1) {
        perror("Error al abrir el archivo");
        return 1;
    }

    char buffer[100];  // Crear un buffer para almacenar los datos leídos
    ssize_t bytes_leidos = read(fd, buffer, sizeof(buffer) - 1);  // Leer del archivo
    if (bytes_leidos == -1) {
        perror("Error al leer el archivo");
        close(fd);
        return 1;
    }

    buffer[bytes_leidos] = '\0';  // Agregar un terminador nulo al final del buffer
    printf("Datos leídos: %s\n", buffer);  // Mostrar los datos leídos

    close(fd);  // Cerrar el archivo
    return 0;
}
```

Implementación del Sistema de Archivos

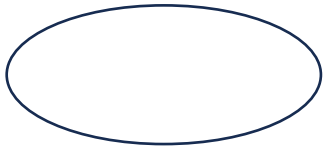
Implementación del Sistema de Archivos

Niveles de la memoria secundaria



Implementación del Sistema de Archivos

Métodos de asignación



Desde el punto de vista de la implementación un archivo es una colección de bloques.
Existen varias técnicas que permiten implementar esa colección de bloques de modo que el espacio se aproveche de forma eficaz y se pueda acceder rápidamente a ellos.

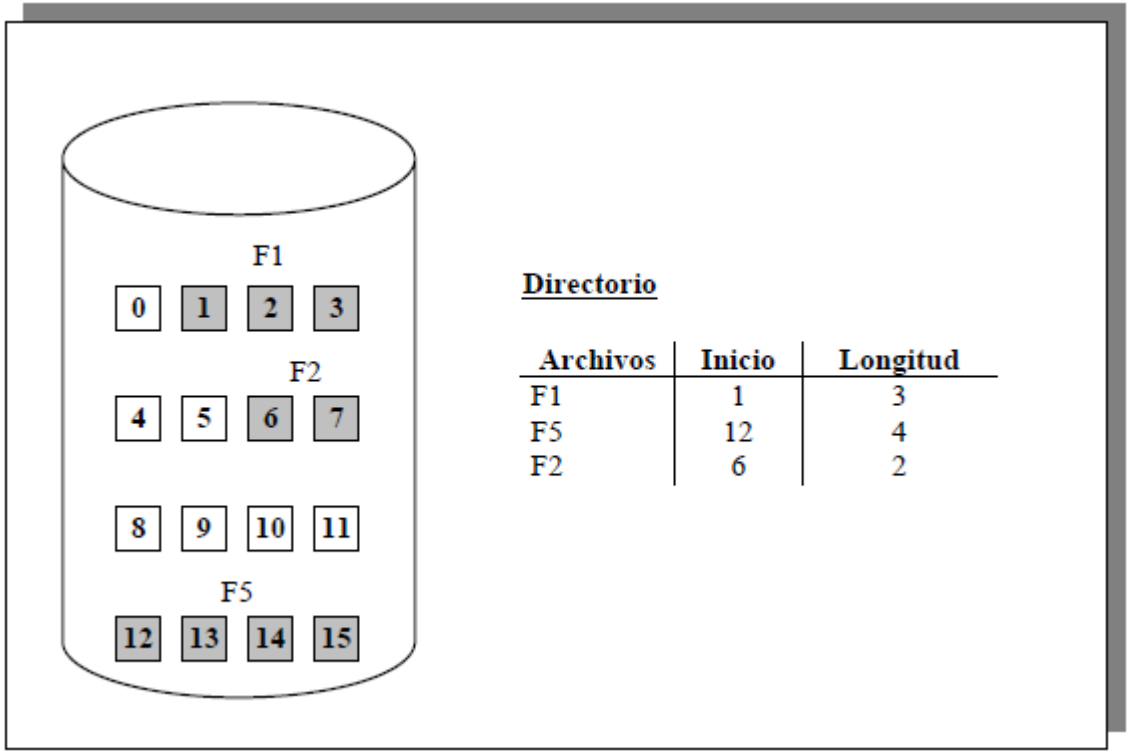
- Asignación contigua
- Asignación enlazada
- Asignación por tabla
- Asignación indexada

Implementación del Sistema de Archivos

Métodos de asignación

Asignación contigua

- Cada archivo ocupa un conjunto de bloques consecutivos en el disco.
- Un archivo queda definido por el primer bloque y su tamaño en bloques



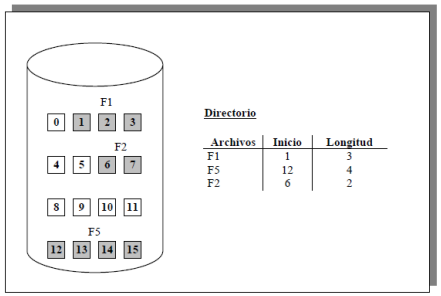
Directorio

Archivos	Inicio	Longitud
F1	1	3
F5	12	4
F2	6	2

Implementación del Sistema de Archivos

Métodos de asignación

- Asignación contigua



Ventaja: Rapidez de acceso ya que para acceder a bloques consecutivos no hace falta mover el cabezal del disco si fuese HDD.

Métodos de gestión: primer, mejor y siguiente hueco.

Primer Hueco: asigna el primer bloque de espacio libre que encuentra y que es lo suficientemente grande para satisfacer la solicitud de almacenamiento. Esto se hace escaneando la lista de bloques libres desde el principio hasta encontrar uno adecuado

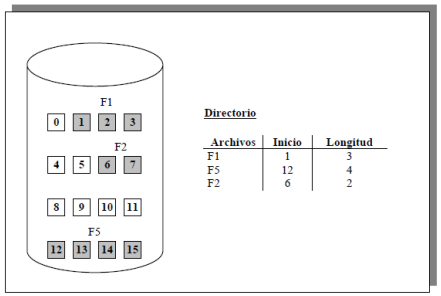
Mejor Hueco: busca el bloque de espacio libre que se ajusta de manera más óptima a la solicitud, es decir, el bloque más pequeño que es capaz de satisfacer la demanda de espacio.

Siguiente Hueco: similar al de primer hueco, pero en lugar de comenzar la búsqueda desde el principio de la lista de bloques libres, empieza desde el último bloque asignado.

Implementación del Sistema de Archivos

Métodos de asignación

Asignación contigua



Ventaja: Rapidez de acceso ya que para acceder a bloques consecutivos no hace falta mover el cabezal del disco si fuese HDD.

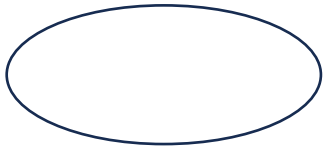
Métodos de gestión: primer, mejor y siguiente hueco.

Desventajas:

- El crecimiento de los archivos está limitado
  - Posible solución: Reservar un número de bloques superior al requerido inicialmente (Producirá fragmentación interna y un escaso aprovechamiento del disco)
- Fragmentación externa
  - Posible solución: Compactación

Implementación del Sistema de Archivos

Métodos de asignación



Desde el punto de vista de la implementación un archivo es una colección de bloques.
Existen varias técnicas que permiten implementar esa colección de bloques de modo que el espacio se aproveche de forma eficaz y se pueda acceder rápidamente a ellos.

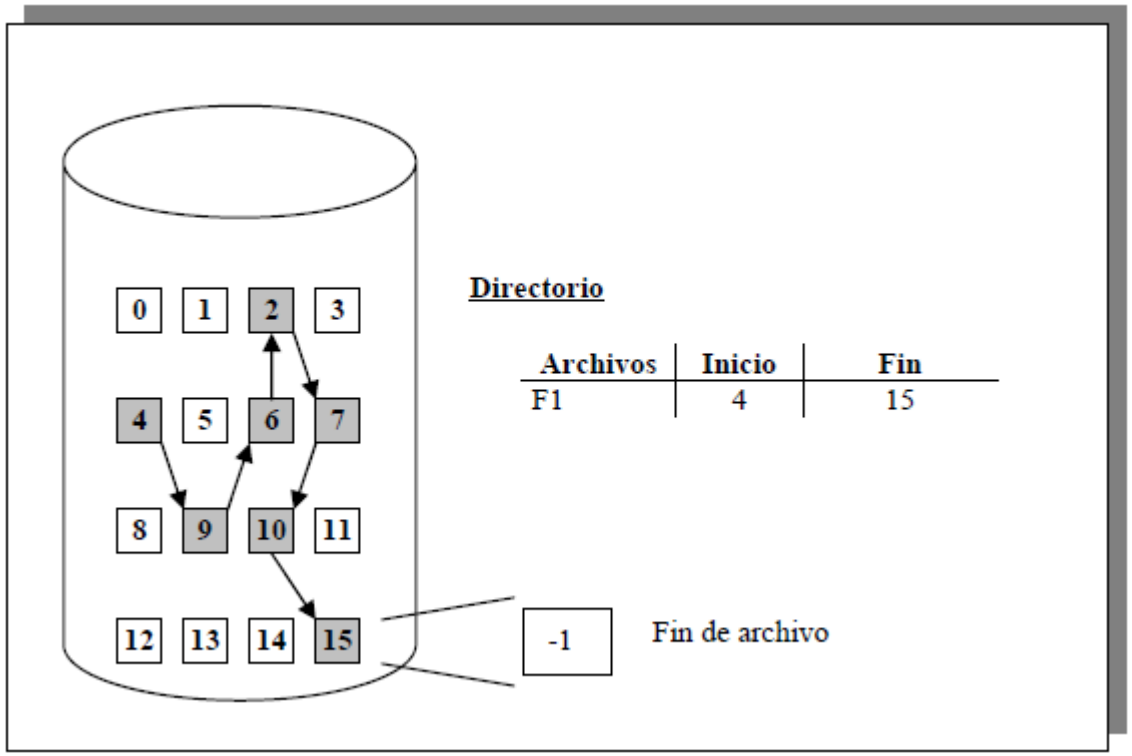
- Asignación contigua
- Asignación enlazada
- Asignación por tabla
- Asignación indexada

Implementación del Sistema de Archivos

Métodos de asignación

Asignación enlazada

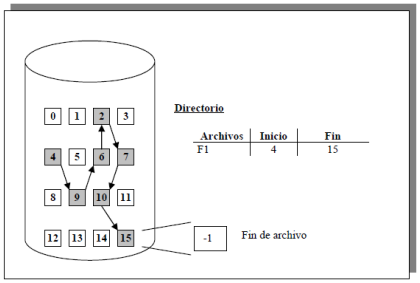
- Cada archivo es una lista enlazada de bloques.
- La entrada de directorio da el número del primer bloque.
- Cada bloque tiene un puntero al siguiente.



Implementación del Sistema de Archivos

Métodos de asignación

Asignación enlazada



Ventajas:

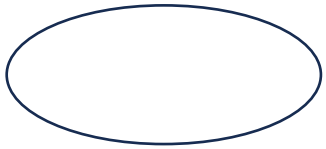
- No limita el crecimiento de los archivos.
- Es fácil el aumento y reducción del tamaño de los archivos
- No existe fragmentación externa por lo que no existe necesidad de compactación.

Desventajas:

- Puede ser ineficiente en términos de espacio si los archivos son pequeños.

Implementación del Sistema de Archivos

Métodos de asignación



Desde el punto de vista de la implementación un archivo es una colección de bloques.
Existen varias técnicas que permiten implementar esa colección de bloques de modo que el espacio se aproveche de forma eficaz y se pueda acceder rápidamente a ellos.

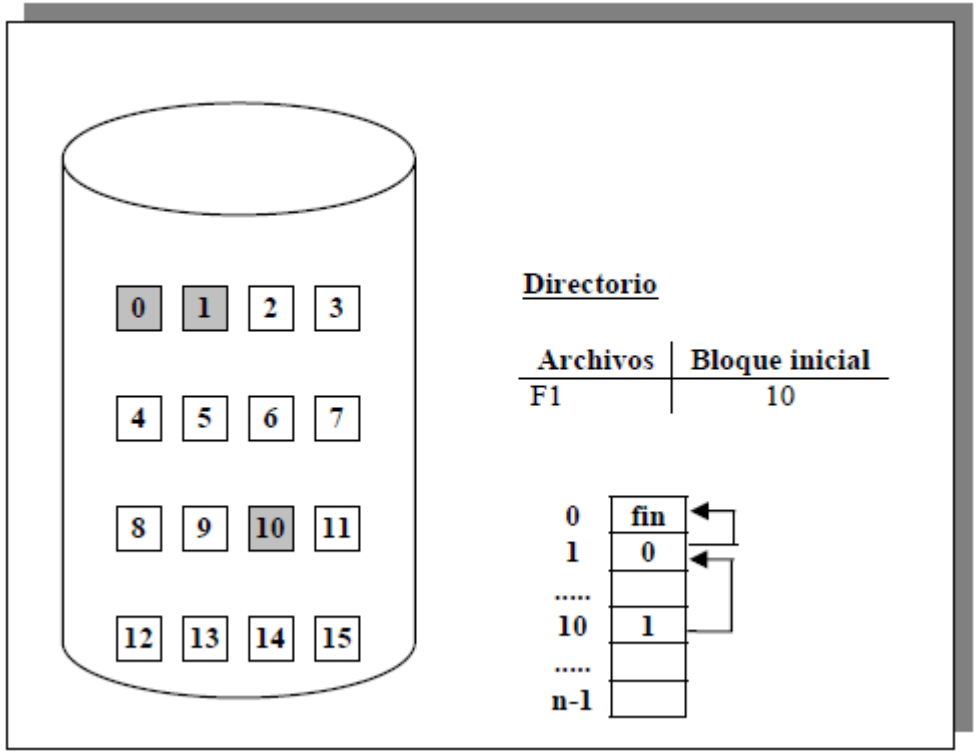
- Asignación contigua
- Asignación enlazada
- Asignación por tabla
- Asignación indexada

Implementación del Sistema de Archivos

Métodos de asignación

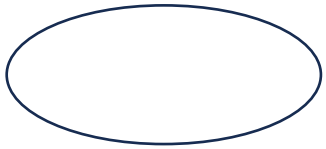
Asignación por tabla

Es una variante de la asignación enlazada en la que los punteros no se encuentran en el bloque, sino en una estructura de datos separada (File Allocation Table - FAT) localizada en los primeros bloques del disco.



Implementación del Sistema de Archivos

Métodos de asignación



Desde el punto de vista de la implementación un archivo es una colección de bloques.
Existen varias técnicas que permiten implementar esa colección de bloques de modo que el espacio se aproveche de forma eficaz y se pueda acceder rápidamente a ellos.

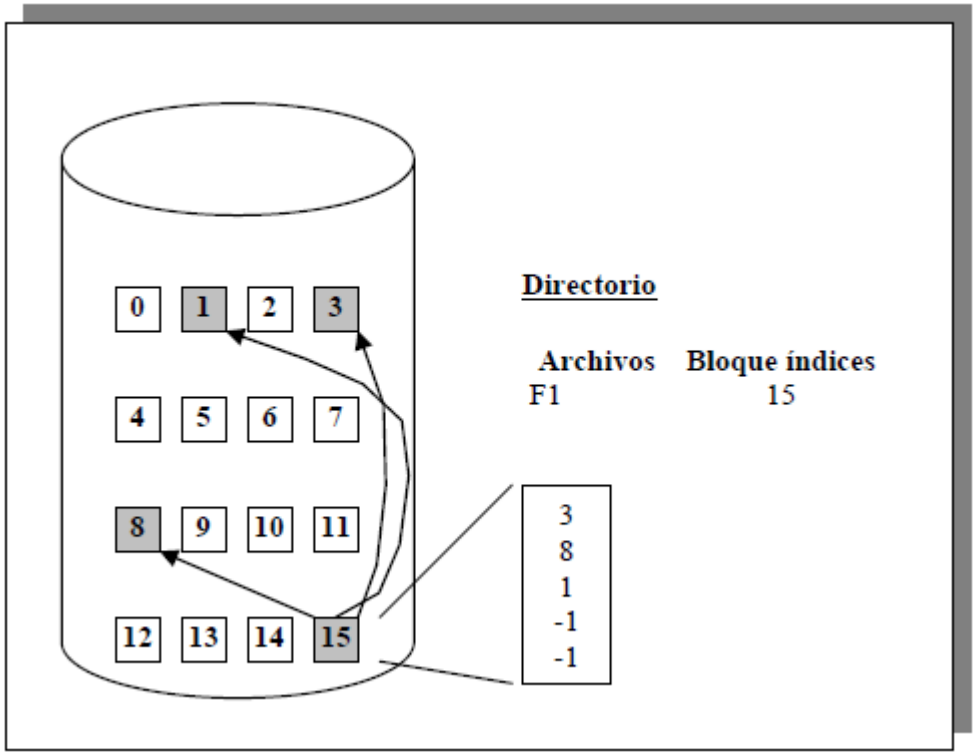
- Asignación contigua
- Asignación enlazada
- Asignación por tabla
- Asignación indexada

Implementación del Sistema de Archivos

Métodos de asignación

Asignación indexada

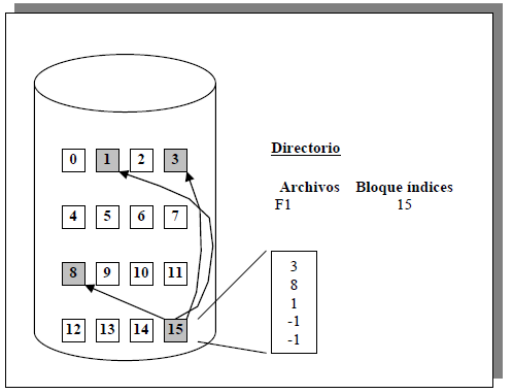
- Cada archivo tiene un bloque de índice donde hay ubicado un vector con todos los punteros a los bloques del archivo.
- La i-ésima entrada del vector contiene el puntero al i-ésimo bloque.



Implementación del Sistema de Archivos

Métodos de asignación

Asignación indexada



Ventajas

- El acceso aleatorio (a un bloque concreto del archivo) se implementa eficientemente.
- No hay fragmentación externa.

Desventajas

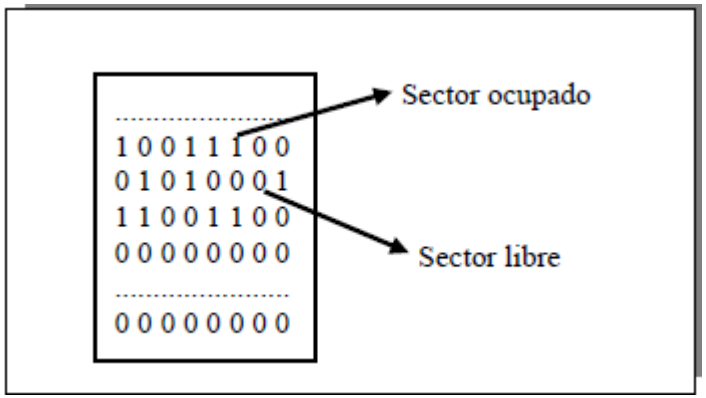
- Con pocos bloques, el bloque de índice supone un desperdicio importante de espacio.
- El tamaño máximo del archivo está limitado por el número de punteros que cabe en un bloque.
- Fragmentación interna en los bloques índice.

Implementación del Sistema de Archivos

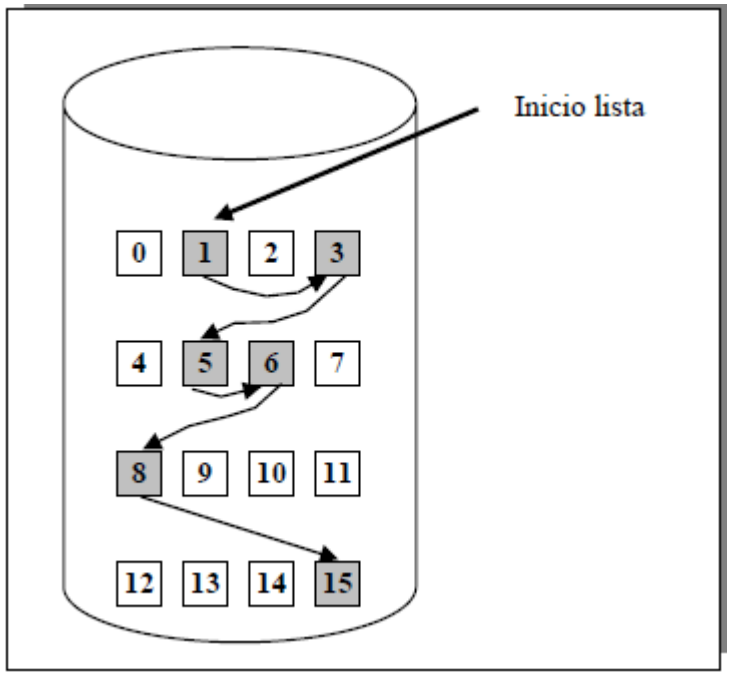
Gestión de espacios

Para llevar a cabo cualquiera de las técnicas de asignación anteriores, es necesario saber qué bloques del disco están disponibles. Para ello el SO debe mantener una estructura de datos con todos los bloques no asignados a archivos.

Mapa de Bits



Lista enlazada de espacios libres



Concurrencia

Contenidos

- Definición
- Problemas de la concurrencia
- Mecanismos de sincronización

Introducción

Definiciones básicas

Concurrencia

Proceso ≠ programa

Proceso: *Programa en ejecución*

Servicios del SO

- **Ejecución concurrente**
- **Sincronización de procesos**
- **Comunicación entre procesos**

Introducción

Definiciones básicas

Proceso ≠ programa

Proceso: *Programa en ejecución*

Servicios del SO

- ☑ Ejecución concurrente
- ☑ **Sincronización de procesos**
- ☑ **Comunicación entre procesos**

Concurrencia
Definición, problemas y sincronización

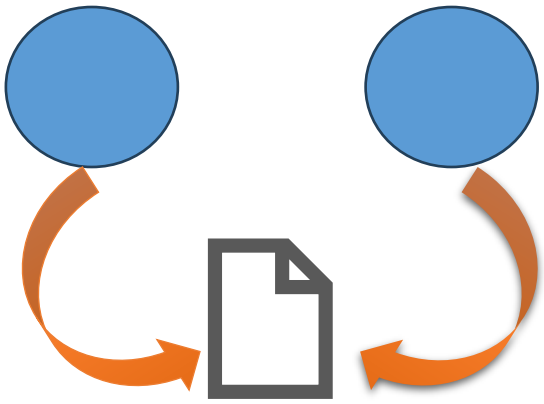
IPC
Señales, tuberías, memoria compartida

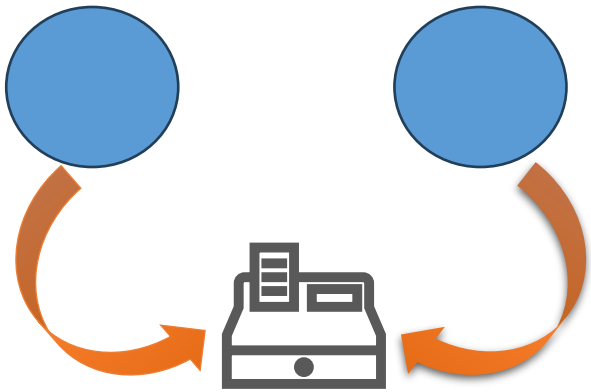
Introducción

Definiciones básicas

- **Ejecución concurrente**

Multiprogramación **Multiprocesamiento** **Procesamiento Distribuido**

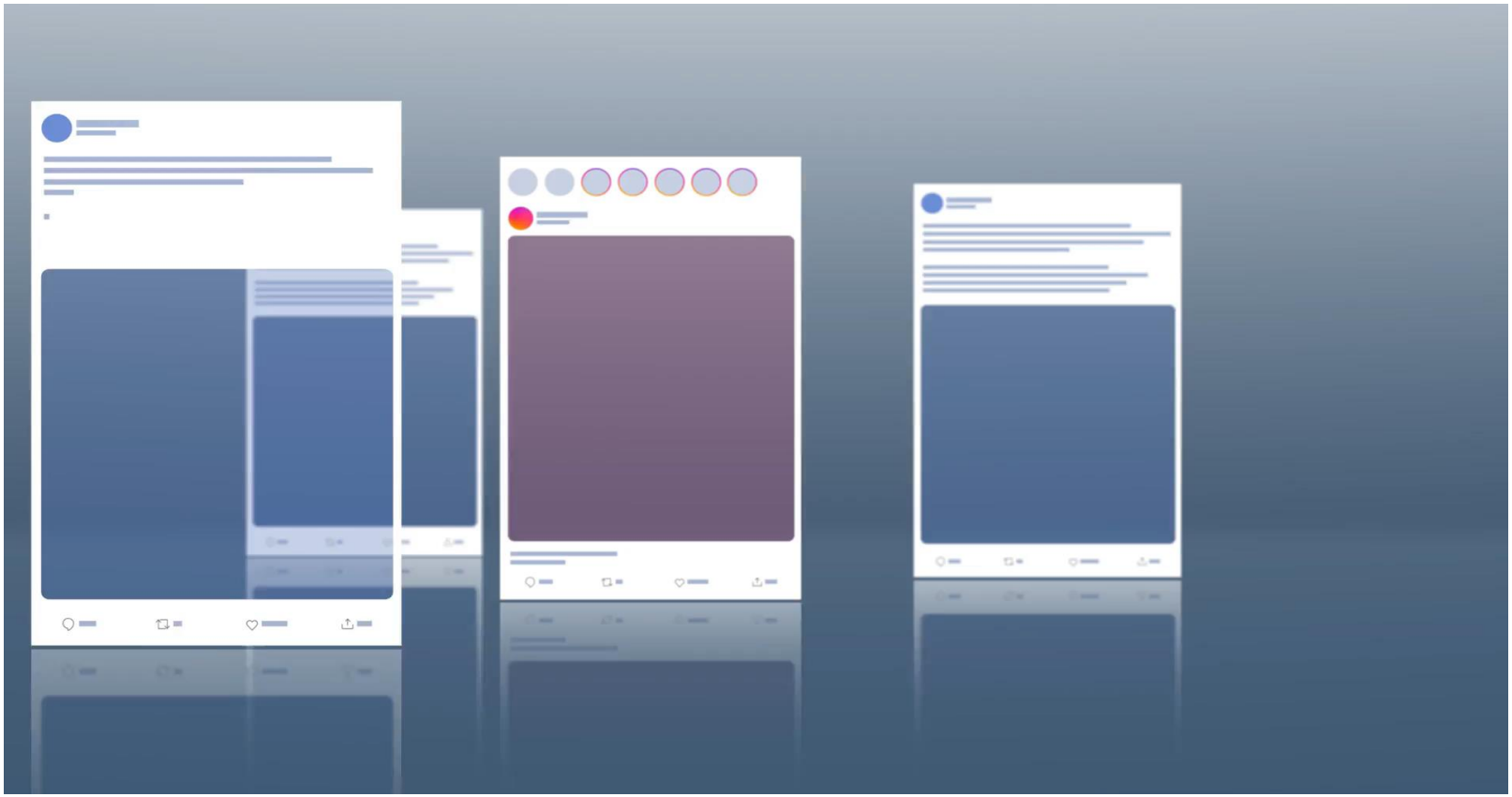




Definición

Concurrencia

La **concurrencia** es el concepto en SO y programación que se refiere a la capacidad de ejecutar múltiples tareas o procesos de manera que parezca que se están ejecutando al mismo tiempo. Estas tareas pueden ser programas, procesos o hilos que compiten por los recursos de un sistema.



Definición

Concurrencia

A **diferencia** del **paralelismo**, donde las tareas se ejecutan simultáneamente en diferentes procesadores o núcleos

En la **concurrencia**, las **tareas** pueden estar **intercaladas** en su ejecución en un solo procesador

Un **SO gestiona** la **concurrencia** al **alternar** entre los procesos o hilos de manera rápida para que parezca que se ejecutan a la vez

Aunque realmente se están **ejecutando** en pequeñas **porciones** de **tiempo**

Definición

Concurrencia

Que hay en un sistema multinúcleo o multiprocesador: ¿concurrencia o paralelismo?

Definición

Concurrencia y Paralelismo

En sistemas con múltiples procesadores o núcleos, **puede haber concurrencia y paralelismo al mismo tiempo**

Concurrencia

En concurrencia, el sistema operativo puede estar alternando entre varias tareas rápidamente, compartiendo los recursos de un solo procesador o de múltiples núcleos

Paralelismo

Paralelismo ocurre cuando varias tareas se ejecutan simultáneamente en diferentes núcleos o procesadores físicos. Es decir, varias tareas se ejecutan literalmente al mismo tiempo, cada una en su propio núcleo

- Un sistema puede tener **varias tareas concurrentes**, donde algunas se ejecutan en paralelo (cada una en un núcleo diferente), y otras se intercalan en el mismo núcleo.
- Por ejemplo, en un sistema de cuatro núcleos, el sistema operativo puede **asignar varias tareas a diferentes núcleos** (paralelismo), mientras que, dentro de cada núcleo, el sistema puede estar **alternando entre varios hilos o procesos concurrentes** (concurrencia)

Definición

Concurrencia y Paralelismo

Concurrencia sin paralelismo

En un sistema con un solo procesador, el SO intercalaría la ejecución de varios procesos, cambiando de uno a otro rápidamente. Ningún proceso se ejecuta al mismo tiempo que otro, pero todos parecen avanzar simultáneamente

Paralelismo sin concurrencia

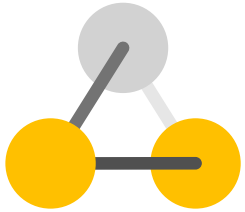
En un sistema con varios procesadores, podrías asignar una tarea única y muy pesada a cada procesador, sin que tengan que intercalar la ejecución con otras tareas. Aquí los procesadores ejecutan tareas en paralelo, pero no hay alternancia de tareas

Concurrencia y paralelismo juntos

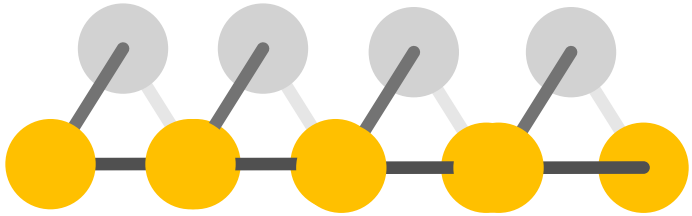
En un sistema con varios núcleos, puedes tener varios procesos concurrentes. Algunos de esos procesos pueden ejecutarse en paralelo (cada uno en un núcleo distinto), mientras que, dentro de cada núcleo, puede haber concurrencia, con varios hilos alternando en el mismo núcleo.

Definición

Concurrencia y Paralelismo



Concurrencia puede ocurrir tanto en sistemas de un solo procesador como en sistemas con múltiples procesadores.



Paralelismo solo es posible en sistemas con múltiples procesadores o núcleos

Problemas de la concurrencia

Problemas de la concurrencia

problemas

- **Condiciones de Carrera** — 01
- **Sección crítica** — 02
- **Exclusión mutua** — 03

Problemas de la concurrencia

Condición de Carrera

01

Condiciones de Carrera

La **condición de carrera** (**race condition**) es un problema que ocurre en sistemas concurrentes o paralelos cuando el resultado de la ejecución de un programa depende del **orden de ejecución no controlado** de varios hilos o procesos que acceden y modifican recursos compartidos

- ✓ En una condición de carrera, si varios procesos o hilos acceden simultáneamente a un recurso (como una variable compartida o una base de datos) **sin una adecuada sincronización**, el resultado final puede ser impredecible o incorrecto.
- ✓ Esto sucede porque los hilos o procesos "compiten" por acceder a los recursos, y el orden en que lo hacen influye en el resultado.

Problemas de la concurrencia

Ejemplo

Condiciones de Carrera

```
contador = 0

def incrementar():
    global contador
    for _ in range(1000):
        contador += 1

# Creamos dos hilos que ejecutarán la misma función
hilo1 = threading.Thread(target=incrementar)
hilo2 = threading.Thread(target=incrementar)

# Iniciamos ambos hilos al mismo tiempo
hilo1.start()
hilo2.start()

# Esperamos que ambos terminen
hilo1.join()
hilo2.join()

print(contador)
```

El valor esperado de contador es 2000 (1000 incrementos por cada hilo)

Sin embargo, debido a la condición de carrera, el valor final podría ser incorrecto, como 1987, 1994, o cualquier otro número menor a 2000

Esto se debe a que ambos hilos están accediendo y modificando el valor de contador sin tener en cuenta el estado en el que se encuentra el otro hilo.

Problemas de la concurrencia

Ejemplo

Concurrencia

01

Condiciones de Carrera

```python
contador = 0

def incrementar():
    global contador
    for _ in range(1000):
        contador += 1

# Creamos dos hilos que ejecutarán la misma función
hilo1 = threading.Thread(target=incrementar)
hilo2 = threading.Thread(target=incrementar)

# Iniciamos ambos hilos al mismo tiempo
hilo1.start()
hilo2.start()

# Esperamos que ambos terminen
hilo1.join()
hilo2.join()

print(contador)
```

El problema surge porque el proceso de lectura y escritura en contador no es atómico. Es decir, el incremento contador += 1 involucra múltiples pasos:

1. Leer el valor de contador.
2. Incrementar el valor.
3. Guardar el nuevo valor en contador.

Si dos hilos hacen esto al mismo tiempo, pueden leer el mismo valor antes de que cualquiera de ellos lo actualice, lo que genera resultados incorrectos.

Problemas de la concurrencia

Ejemplo

Condiciones de Carrera

```
contador = 0

def incrementar():
    global contador
    for _ in range(1000):
        contador += 1

# Creamos dos hilos que ejecutarán la misma función
hilo1 = threading.Thread(target=incrementar)
hilo2 = threading.Thread(target=incrementar)

# Iniciamos ambos hilos al mismo tiempo
hilo1.start()
hilo2.start()

# Esperamos que ambos terminen
hilo1.join()
hilo2.join()

print(contador)
```

Solución: Sincronización

- Para evitar una condición de carrera, necesitamos **sincronizar** los **hilos** y asegurar que solo **uno acceda** a la **variable compartida** a la vez.
- Un mecanismo común es el uso de locks (**mutex**), que garantizan que solo un hilo pueda entrar en la **sección crítica** (el código que accede a la variable compartida) a la vez.

Problemas de la concurrencia

Condición de Carrera

01

Condiciones de Carrera

La **condición de carrera** (**race condition**) es un problema que ocurre en sistemas concurrentes o paralelos cuando el resultado de la ejecución de un programa depende del **orden de ejecución no controlado** de varios hilos o procesos que acceden y modifican recursos compartidos

- ✓ En una condición de carrera, si varios procesos o hilos acceden simultáneamente a un recurso (como una variable compartida o una base de datos) **sin una adecuada sincronización**, el resultado final puede ser impredecible o incorrecto.
- ✓ Esto sucede porque los hilos o procesos "compiten" por acceder a los recursos, y el orden en que lo hacen influye en el resultado.

Problemas de la concurrencia

problemas

- **Condiciones de Carrera** — 01
- **Sección crítica** — 02
- **Exclusión mutua** — 03

Problemas de la concurrencia

Sección crítica

Se denomina **Sección Crítica (SC)** de un proceso o hilo a aquellas partes del código que no pueden ejecutarse de forma concurrente.

Problema:

Si dos procesos están dentro de la misma sección crítica al mismo tiempo, pueden modificar los recursos compartidos de manera no coordinada, causando problemas en los datos o el funcionamiento del sistema.

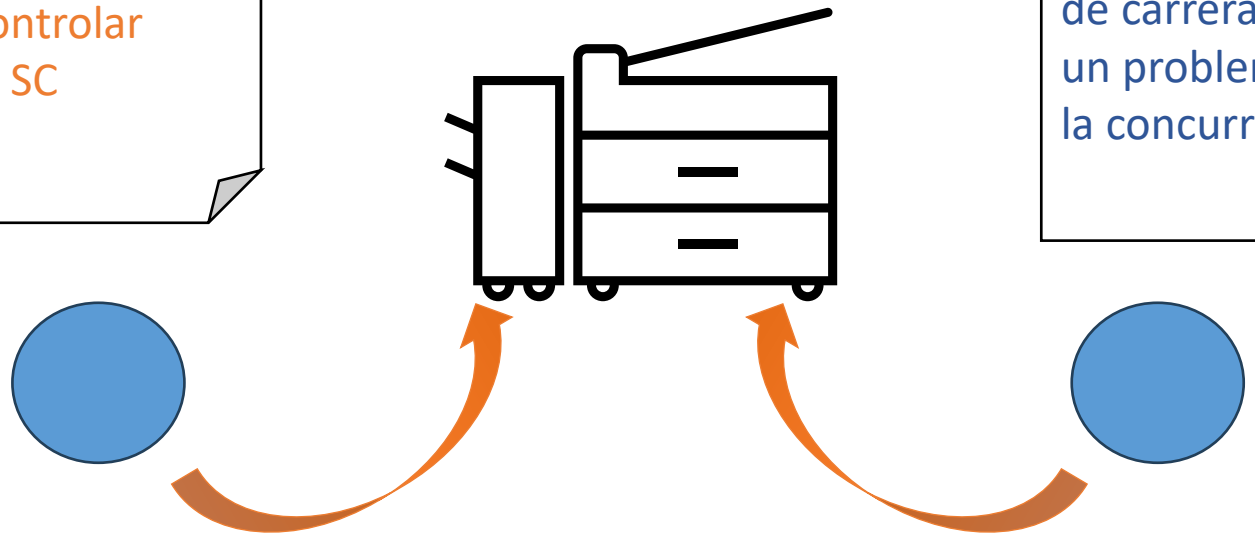
Problemas de la concurrencia

- **Sección crítica**

Es importante Las condiciones de carrera son controlar un problema de la SC la concurrencia

Es importante controlar la SC

Las condiciones de carrera son un problema de la concurrencia



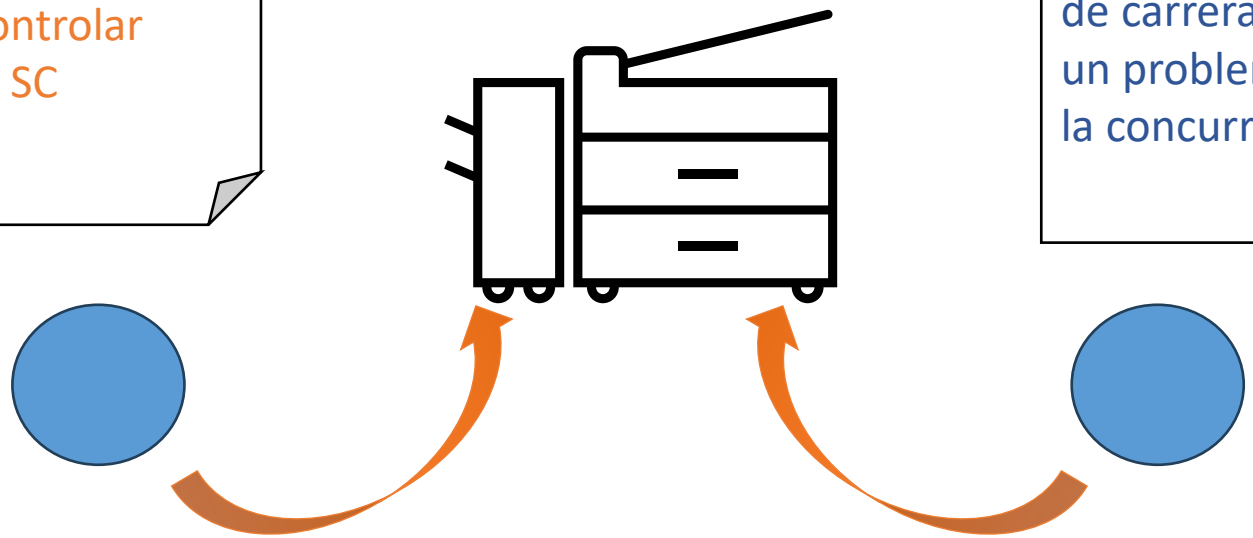
Problemas de la concurrencia

Sección crítica

La gestión de concurrencia en este caso la implementa el SO (con colas de impresión) junto con los drivers

Es importante controlar la SC

Las condiciones de carrera son un problema de la concurrencia



En el diseño de sistemas concurrentes es necesario que identifiquemos las secciones críticas para tratarlas correctamente y así evitar condiciones de carrera.

Problemas de la concurrencia

Concurrencia

Ejemplo

02

Sección crítica

```python
contador = 0

def incrementar():
    global contador
    for _ in range(1000):
        contador += 1

# Creamos dos hilos que ejecutarán la misma función
hilo1 = threading.Thread(target=incrementar)
hilo2 = threading.Thread(target=incrementar)

# Iniciamos ambos hilos al mismo tiempo
hilo1.start()
hilo2.start()

# Esperamos que ambos terminen
hilo1.join()
hilo2.join()

print(contador)
```

SC

Identificar la sección crítica

El objetivo es garantizar que, en cualquier momento, solo un proceso o hilo pueda ejecutar la sección crítica. Esto se logra mediante la **exclusión mutua**.

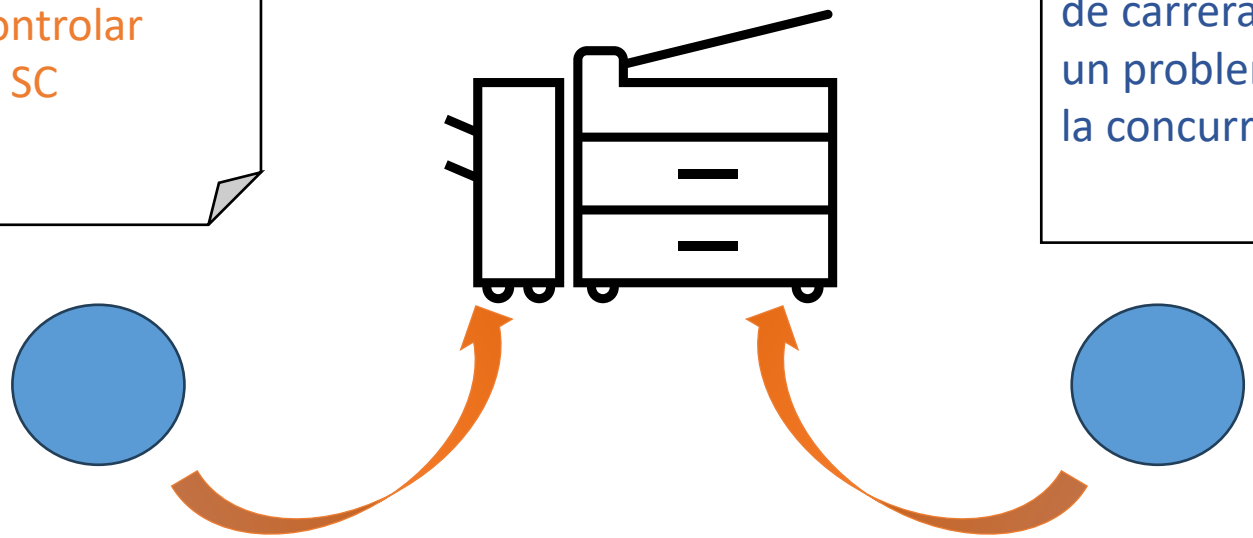
Problemas de la concurrencia

Sección crítica

La gestión de concurrencia en este caso la implementa el SO (con colas de impresión) junto con los drivers

Es importante controlar la SC

Las condiciones de carrera son un problema de la concurrencia



En el diseño de sistemas concurrentes es necesario que identifiquemos las secciones críticas para tratarlas correctamente y así evitar condiciones de carrera.

Problemas de la concurrencia

problemas

- **Condiciones de Carrera** — 01
- **Sección crítica** — 02
- **Exclusión mutua** — 03

Problemas de la concurrencia

Exclusión mutua

La **exclusión mutua** asegura que una sección crítica sea ejecutada por un solo proceso o hilo a la vez. Esto evita la interferencia entre procesos.

La exclusión mutua es una propiedad que asegura que, cuando un proceso está ejecutando su sección crítica (SC) —es decir, la parte del código que accede a recursos compartidos—, ningún otro proceso puede acceder a esa SC hasta que el primer proceso haya terminado.

El **protocolo** es un conjunto de reglas o mecanismos implementados en el código para asegurar que la SC se ejecute de forma exclusiva.

Problemas de la concurrencia

Exclusión mutua

03

Para que un mecanismo de exclusión mutua sea efectivo, debe cumplir con los siguientes requisitos

01

Acceso Exclusivo

Solo un proceso debe tener permiso para entrar en la SC en un momento dado.

02

Interrupción

Cuando un proceso es interrumpido mientras está en una región no crítica, no debe interferir con el resto de los procesos.

03

Progreso

No puede demorarse un proceso indefinidamente en una sección crítica

04

No Bloqueo

Cuando ningún proceso está en su SC, cualquier proceso que solicite entrar debe hacerlo sin dilación

05

Independencia de Velocidad

No se deben hacer suposiciones sobre la velocidad relativa de los procesos ni sobre el número de procesadores. Esto asegura que el mecanismo funcione en sistemas heterogéneos y con diferentes arquitecturas.

06

Tiempo Finito

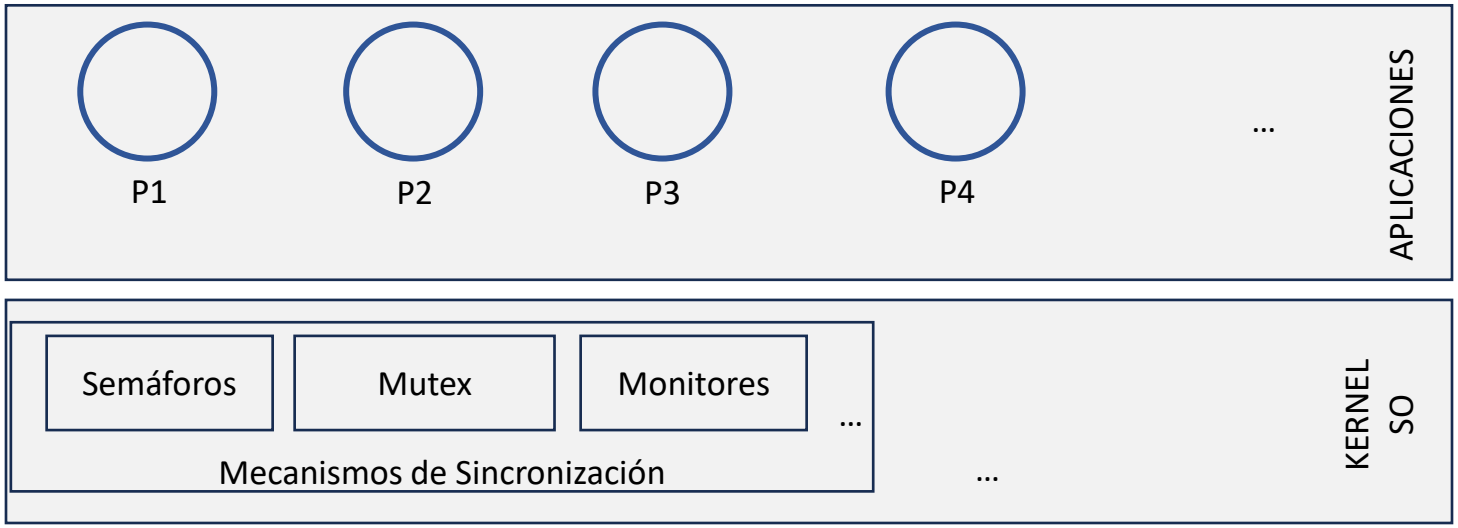
Un proceso debe permanecer en su SC solo por un tiempo finito

Problemas de la concurrencia

Exclusión mutua

La responsabilidad de implementar y mantener la exclusión mutua recae sobre los procesos y sus programadores

- Es decir, los programadores deben utilizar mecanismos adecuados para asegurar que solo un proceso pueda acceder a la sección crítica en un momento dado.
- Los SO modernos implementan de manera nativa mecanismos de sincronización que permiten la implementación de la exclusión muta



Mecanismos de Sincronización

Mecanismos de Sincronización

Contenido

- **Semáforos** 01
- **Mutex** 02
- **Monitores** 03

Concurrencia

Mecanismos de Sincronización

01 Semáforos

Un **semáforo** es una estructura de datos utilizada para controlar el acceso a recursos compartidos en un sistema operativo mediante operaciones atómicas.

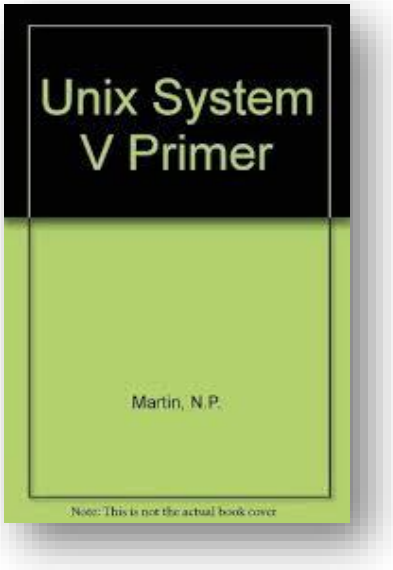
60

Fue introducido por **Edsger Dijkstra** en los años 60 para resolver problemas de sincronización y se implementan de forma experimental en sistemas académicos como el **THE Operating System**.

Mecanismos de Sincronización

01 Semáforos

Un **semáforo** es una estructura de datos utilizada para controlar el acceso a recursos compartidos en un sistema operativo mediante operaciones atómicas.



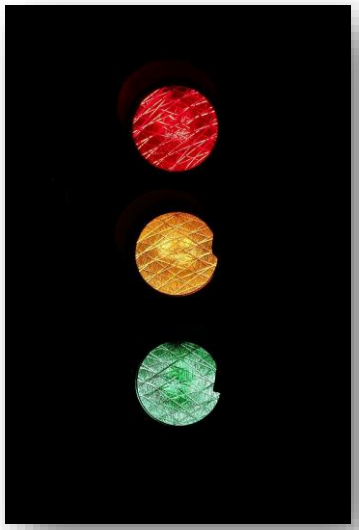
70 - 80

Las primeras implementaciones comerciales en sistemas operativos como **UNIX System V** usan semáforos para IPC (comunicación entre procesos)

Mecanismos de Sincronización

01 Semáforos

Un **semáforo** es una estructura de datos utilizada para controlar el acceso a recursos compartidos en un sistema operativo mediante operaciones atómicas.



90

Los semáforos se implementan de forma nativa y estandarizada en muchos sistemas operativos a través del estándar POSIX (POSIX.1b), haciendo que semáforos, mutexes y otros mecanismos de sincronización formen parte del kernel en sistemas como Linux, BSD, Windows NT, entre otros.

Mecanismos de Sincronización

01 Semáforos

Tipos de semáforos

Un semáforo mantiene un **contador** que refleja el estado de disponibilidad de un recurso.

1. **Semáforo Binario**
   - Solo permite que un único proceso acceda a la sección crítica a la vez.
2. **Semáforo Contador**:
   - Permite un número limitado de procesos en la sección crítica.
   - Se usa para gestionar acceso concurrente cuando múltiples instancias de un recurso están disponibles.

Mecanismos de Sincronización

Operaciones en los semáforos

1. wait() (también conocida como **P** o **down**, **acquire**() en Python):

Problemas de la concurrencia

Concurrencia

01 Semáforos

WAIT

```
wait(semaforo) {
    while (semaforo <= 0);
    semaforo = semaforo - 1;
}
```

Si el valor del semáforo es mayor que 0, decrementa el contador y permite que el proceso entre en la sección crítica.

Si el valor es 0, el proceso se bloquea y espera hasta que otro proceso incremente el semáforo.

Mecanismos de Sincronización

Operaciones en los semáforos

2. signal() (también conocida como **V** o **up**, **release**() en Python):

Problemas de la concurrencia

01 Semáforos

SIGNAL

```
signal(semaforo) {
    semaforo = semaforo + 1;
}
```

Incrementa el valor del semáforo y despierta a cualquier proceso que esté esperando para entrar en la sección crítica

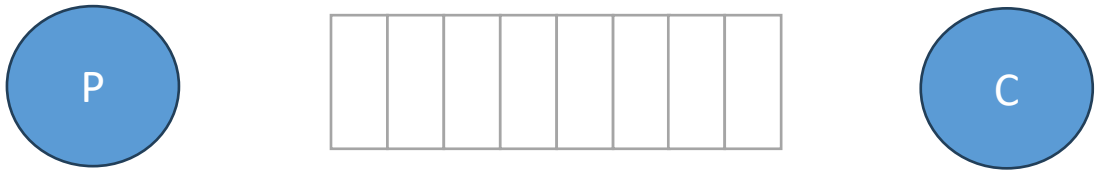
Mecanismos de Sincronización

Ejemplo

01 ▶ **Semáforos**

Problema del Productor-Consumidor tamaño ilimitado

- Dos procesos (un productor y un consumidor) que comparten un buffer de tamaño ilimitado.

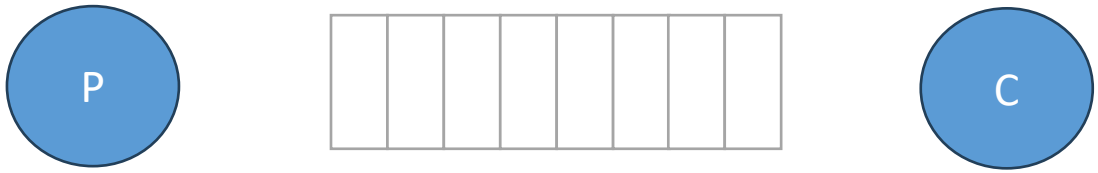


Mecanismos de Sincronización

Ejemplo

Problema del Productor-Consumidor

- Dos procesos (un productor y un consumidor) que comparten un buffer de tamaño fijo.
- El productor añade elementos al buffer y el consumidor los consume



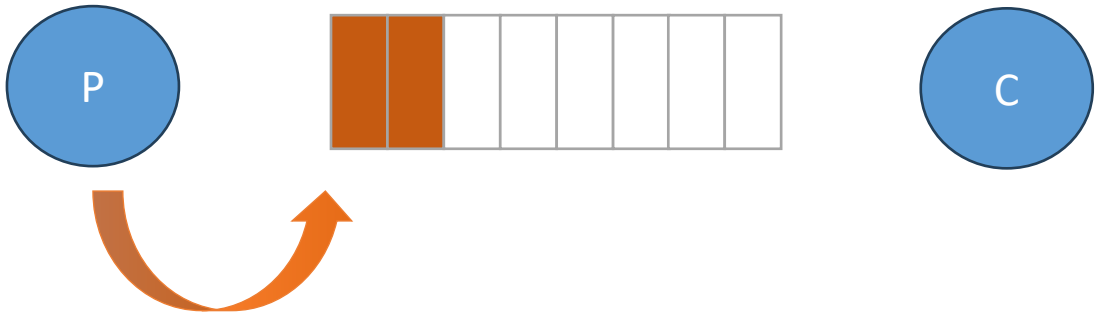
Mecanismos de Sincronización

Ejemplo

01 ▶ Semáforos

Problema del Productor-Consumidor

- Dos procesos (un productor y un consumidor) que comparten un buffer de tamaño fijo.
- El productor añade elementos al buffer y el consumidor los consume



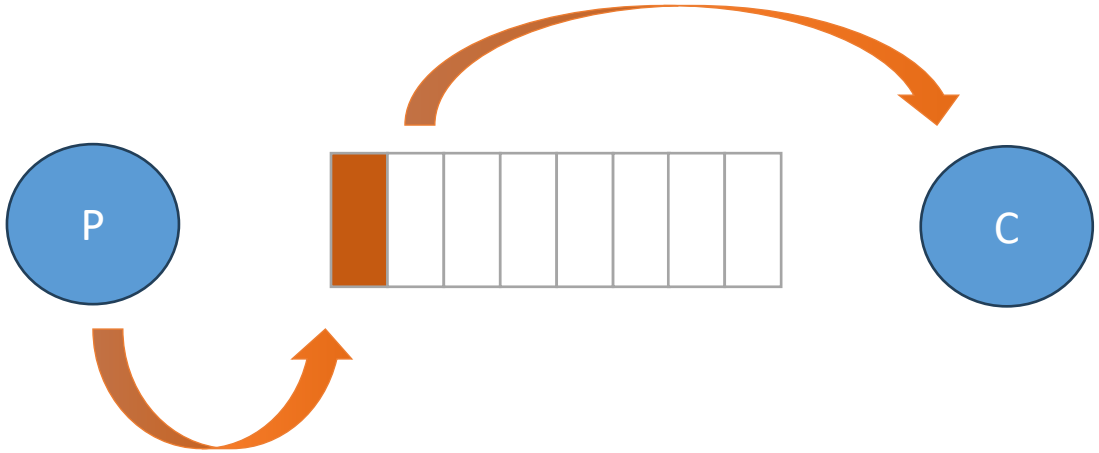
Mecanismos de Sincronización

Ejemplo

01 ▶ Semáforos

Problema del Productor-Consumidor

- Dos procesos (un productor y un consumidor) que comparten un buffer de tamaño fijo.
- El productor añade elementos al buffer y el consumidor los consume



Problemas de la concurrencia

01 ▶ **Semáforos**

Productor-Consumidor buffer ilimitado

```
void productor() {
  while (true) {
    producir();

    añadir_buffer();


  }
}
```

```
void consumidor() {
  while (true) {


    coger_ buffer();

    consumir();
  }
}
```

Lanzar los procesos en paralelo

Solo uno de los dos procesos puede tener acceso al buffer, de modo que hay que identificar la sección crítica (exclusión mutua)

```
void main() {


  cobegin
      productor();
      consumidor();
  coend;
}
```

Problemas de la concurrencia

01 ▶ **Semáforos**

Productor-Consumidor buffer ilimitado

```
void productor() {
  while (true) {
    producir();

    añadir_buffer();


  }
}
```

```
void consumidor() {
  while (true) {

    coger_ buffer();

    consumir();
  }
}
```

Lanzar los procesos en paralelo

Solo uno de los dos procesos puede tener acceso al buffer, de modo que hay que identificar la sección crítica (exclusión mutua)

```
void main() {


  cobegin
     productor();
     consumidor();
  coend;
}
```

Problemas de la concurrencia

Concurrencia

01 ▶ Semáforos

Productor-Consumidor buffer ilimitado

```
TSemáforo s , n;
```

```
void productor() {
  while (true) {
    producir();
    wait(s);
    añadir_buffer();
    signal(s);
    signal(n);
  }
}
```

```
void consumidor() {
  while (true) {
    wait(n);
    wait(s);
    coger_ buffer();
    signal(s);
    consumir();
  }
}
```

```
void main() {
  inicializar(s, 1);
  inicializar(n, 0);
  cobegin
      productor();
      consumidor();
  coend;
}
```

Lanzar los procesos en paralelo

Solo uno de los dos procesos puede tener acceso al buffer, de modo que hay que identificar la sección crítica (exclusión mutua)

Creamos un semáforo s para la sección crítica y lo inicializamos en 1, así el primer proceso que acceda podrá entrar.

El consumidor solo puede consumir si hay algo en el buffer (sincronización de procesos)

Creamos un semáforo n para la sincronización y lo inicializamos en 0, así si el consumidor llega antes tiene que esperar

WAIT:

- Si el valor del semáforo es mayor que 0, decrementa el contador y permite que el proceso entre en la sección crítica.
- Si el valor es 0, el proceso se bloquea y espera hasta que otro proceso incremente el semáforo.

SIGNAL:

- Incrementa el valor del semáforo y despierta a cualquier proceso que esté esperando para entrar en la sección crítica

Problemas de la concurrencia

Concurrencia

01 ▶ Semáforos

Productor-Consumidor

Buffer Limitado

```
TSemáforo s , n, b;
#define tamaño_buffer N
```

```
void productor() {
  while (true) {
    producir();
    wait(b);
    wait(s);
    añadir_buffer();
    signal(s);
    signal(n);
  }
}
```

```
void consumidor() {
  while (true) {
    wait(n);
    wait(s);
    coger_ buffer();
    signal(s);
    signal(b);
    consumir();
  }
}
```

Lanzar los procesos en paralelo

Solo uno de los dos procesos puede tener acceso al buffer, de modo que hay que identificar la sección crítica (exclusión mutua)

Creamos un semáforo s para la sección crítica y lo inicializamos en 1, así el primer proceso que acceda podrá entrar.

El consumidor solo puede consumir si hay algo en el buffer (sincronización de procesos)

Creamos un semáforo n para la sincronización y lo inicializamos en 0, así si el consumidor llega antes tiene que esperar

Añadir un semáforo para que el productor no produzca si está lleno el buffer(sincronización de procesos)

```
void main() {
  inicializar(s, 1);
  inicializar(n, 0);
  inicializar(b, tamaño_buffer);
  cobegin
      productor();
      consumidor();
  coend;
}
```

WAIT:

- Si el valor del semáforo es mayor que 0, decrementa el contador y permite que el proceso entre en la sección crítica.
- Si el valor es 0, el proceso se bloquea y espera hasta que otro proceso incremente el semáforo.

SIGNAL:

- Incrementa el valor del semáforo y despierta a cualquier proceso que esté esperando para entrar en la sección crítica

Problemas de la concurrencia

Concurrencia

01 ▶ Semáforos

Contador en Python

```python
contador = 0

def incrementar():
    global contador
    for _ in range(1000):
        contador += 1

# Creamos dos hilos que ejecutarán la misma función
hilo1 = threading.Thread(target=incrementar)
hilo2 = threading.Thread(target=incrementar)

# Iniciamos ambos hilos al mismo tiempo
hilo1.start()
hilo2.start()

# Esperamos que ambos terminen
hilo1.join()
hilo2.join()

print(contador)
```

```python
contador = 0
semaforo = threading.Semaphore(1)  # Creación del semáforo

def incrementar():
    global contador
    for _ in range(1000):
        semaforo.acquire()  # Adquirir el semáforo
        try:
            contador += 1  # Sección crítica
        finally:
            semaforo.release()  # Asegurarse de liberar el semáforo

# Creación de hilos
hilo1 = threading.Thread(target=incrementar)
hilo2 = threading.Thread(target=incrementar)

# Iniciar hilos
hilo1.start()
hilo2.start()

# Esperar a que terminen
hilo1.join()
hilo2.join()

print(contador)  # Valor correcto esperado: 2000
```

Problemas de la concurrencia

Concurrencia

Contador en Python

01 Semáforos

```python
contador = 0
semaforo = threading.Semaphore(1)  # Creación del semáforo

def incrementar():
    global contador
    for _ in range(1000):
        with semaforo:  # Sección crítica
            contador += 1  # Solo un hilo puede ejecutar esta línea a la vez

# Creación de hilos
hilo1 = threading.Thread(target=incrementar)
hilo2 = threading.Thread(target=incrementar)

# Iniciar hilos
hilo1.start()
hilo2.start()

# Esperar a que terminen
hilo1.join()
hilo2.join()

print(contador)  # Valor correcto esperado: 2000
```

Cuando se usa **with** semaforo::

Adquisición: Al entrar en el bloque with, el semáforo se adquiere automáticamente. Esto significa que el hilo que entra en este bloque puede proceder a ejecutar el código dentro de él si el semáforo está disponible.

Liberación: Al salir del bloque with (ya sea porque el código se ejecutó con éxito o porque ocurrió una excepción), el semáforo se libera automáticamente. Esto asegura que otros hilos puedan adquirirlo.

Mecanismos de Sincronización

Contenido

- **Semáforos** 01
- **Mutex** 02
- **Monitores** 03

Concurrencia

Mecanismos de Sincronización

Mutex

02 ▶ **Mutex**

Un **mutex** es un objeto de sincronización que se utiliza para garantizar que solo un hilo tenga acceso a un recurso compartido a la vez.

Características de un Mutex:

- **Exclusividad**: Solo un hilo puede poseer un mutex a la vez. Si un hilo ha adquirido el mutex, otros hilos que intenten adquirirlo se bloquearán hasta que el mutex sea liberado.
- **Sección Crítica**: Se utiliza para proteger secciones críticas en el código donde se accede a recursos compartidos

Un **semáforo,** a diferencia de un mutex, puede permitir que varios hilos accedan a un recurso compartido simultáneamente

Problemas de la concurrencia

Concurrencia

02 ▶ Mutex

Contadores

```python
contador = 0
lock = threading.Lock()  # Crear un lock (mutex)

def incrementar():
    global contador
    for _ in range(1000):
        with lock:  # Adquirir el lock
            contador += 1  # Sección crítica

# Creación de hilos
hilo1 = threading.Thread(target=incrementar)
hilo2 = threading.Thread(target=incrementar)

# Iniciar hilos
hilo1.start()
hilo2.start()

# Esperar a que terminen
hilo1.join()
hilo2.join()

print(contador)  # Valor correcto esperado: 2000
```

Mecanismos de Sincronización

Contenido

- **Semáforos** 01
- **Mutex** 02
- **Monitores** 03

Concurrencia

Mecanismos de Sincronización

03 ► Monitores

Los monitores no son un tipo de sincronización a nivel de sistema operativo, sino que son una abstracción de alto nivel creada por los programadores o lenguajes de programación para facilitar la sincronización y la exclusión mutua.
En esencia, los monitores encapsulan mecanismos más básicos de sincronización, como mutexes o semaforos, que sí son gestionados por el sistema operativo o el runtime de la aplicación.

Mecanismos de Sincronización

Contenido

- **Semáforos** 01
- **Mutex** 02
- **Monitores** 03

Concurrencia

IPC

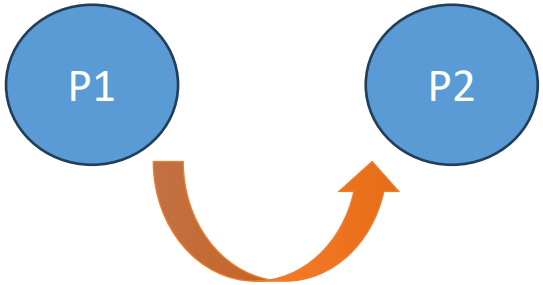
Contenidos

- [x] Señales
- [x] Tuberías
- [x] Memoria compartida

IPC

Inter Process Comunication

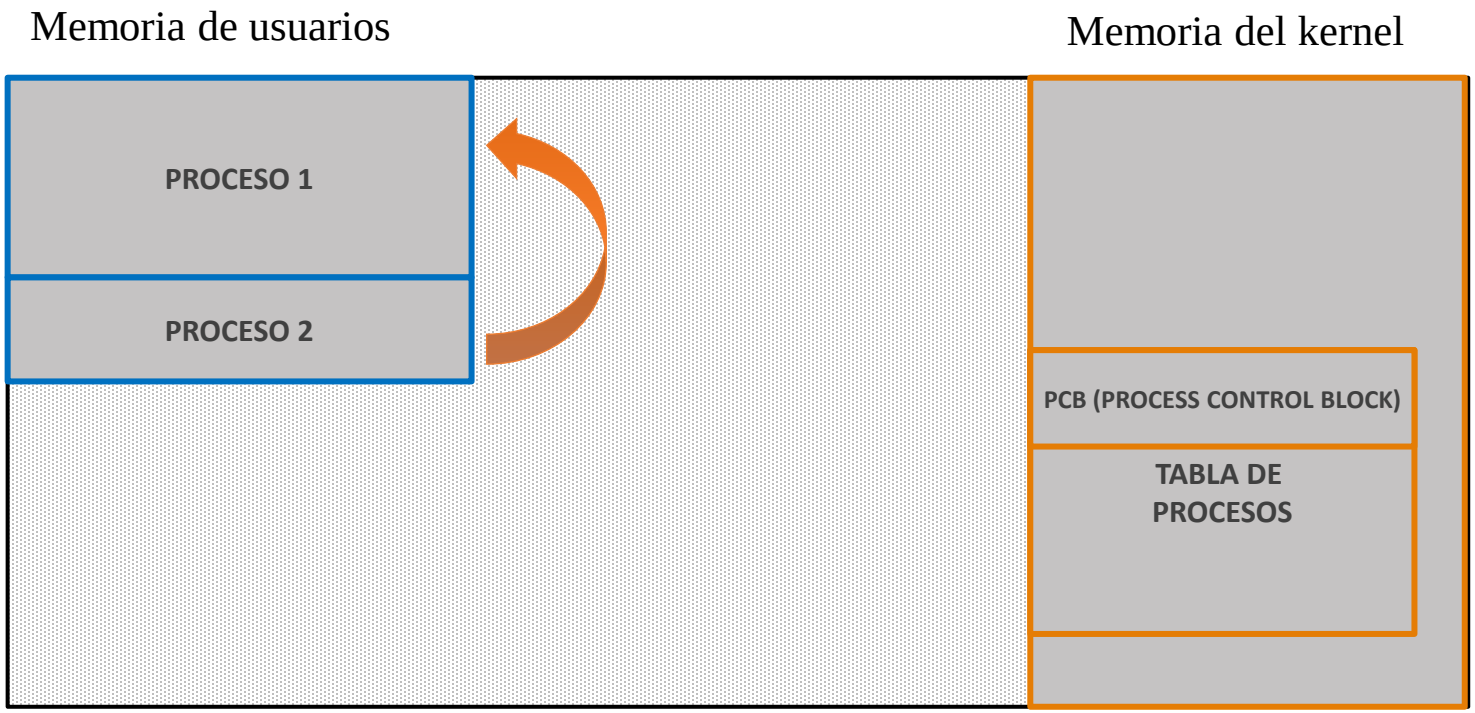
IPC (Inter-Process Communication) se refiere a los mecanismos y técnicas utilizados para que los procesos que se ejecutan en un sistema operativo puedan comunicarse y compartir datos entre sí. Los procesos son generalmente independientes y están aislados unos de otros, pero en muchas aplicaciones es crucial que estos procesos trabajen en conjunto y colaboren de manera coordinada



IPC

Inter Process Comunication

IPC (Inter-Process Communication) se refiere a los mecanismos y técnicas utilizados para que los procesos que se ejecutan en un sistema operativo puedan comunicarse y compartir datos entre sí. Los procesos son generalmente independientes y están aislados unos de otros, pero en muchas aplicaciones es crucial que estos procesos trabajen en conjunto y colaboren de manera coordinada



IPC

Inter Process Comunication

- Los sistemas operativos modernos suelen ejecutar cada proceso en su propio espacio de memoria aislado por razones de seguridad y estabilidad.
- Este aislamiento evita que un proceso interfiera directamente con los datos de otro.
- Sin embargo, en aplicaciones complejas, múltiples procesos deben colaborar para realizar tareas más grandes.
- IPC proporciona un puente seguro entre estos espacios de memoria aislados para que los procesos puedan intercambiar información.

Señales

IPC

Señales

Definiciones

- Las señales (signals) son un mecanismo de comunicación asíncrona en sistemas operativos tipo Unix (como Linux), que permiten a los procesos notificarse sobre ciertos eventos.
- Las señales pueden ser enviadas por el sistema operativo, por otros procesos o incluso por el mismo proceso para notificar la ocurrencia de eventos como interrupciones, errores, finalización de procesos, etc.
- Cuando un proceso recibe una señal, puede reaccionar a ella de diferentes maneras, como ejecutar una función específica (llamada manejador de señales) o simplemente ignorar la señal.

Señales

Cómo funcionan

¿Cómo funcionan las señales?

Envío de señales:
Las señales pueden ser enviadas a un proceso para indicarle que algo sucedió. Por ejemplo, la señal SIGINT (señal de interrupción) se envía cuando el usuario presiona Ctrl + C en la terminal.

Manejadores de señales:
Un proceso puede definir una función especial llamada manejador de señales (signal handler) que se ejecuta cuando el proceso recibe una señal específica.
Si no se define un manejador, el proceso puede usar la acción por defecto para esa señal, que en algunos casos podría ser terminar el proceso.

Señales

Ejemplo de señales

Señal	Descripción	Acción Predeterminada	Código Asociado
SIGINT	Interrupción de teclado (generalmente activada por `Ctrl + C`).	Termina el proceso.	2
SIGKILL	Mata el proceso de forma inmediata, no se puede manejar ni ignorar.	Mata el proceso inmediatamente.	9
SIGTERM	Solicita la terminación de un proceso de manera "amable". Puede ser manejada o ignorada.	Termina el proceso, a menos que se maneje.	15
SIGSEGV	Error de segmentación, ocurre cuando se intenta acceder a una región de memoria no permitida.	Mata el proceso.	11
SIGCHLD	Notifica al proceso padre que uno de sus hijos ha terminado o ha cambiado de estado.	Ignorar o esperar al hijo.	17 (Linux/Unix)
SIGALRM	Se envía cuando una alarma (configurada con `alarm()`) expira.	Termina el proceso, a menos que se maneje.	14

Señales

Funciones en C

signal(int sig, void (*handler)(int))

Se utiliza para registrar un manejador de señales. Permite que el proceso asocie una función a una señal específica.

kill(pid_t pid, int sig)

Se utiliza para enviar una señal a un proceso.

int alarm(int seconds)

Configura una alarma para que se envíe la señal SIGALRM al proceso después de un tiempo específico (en segundos).

Señales

Ejemplo de código


```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

// Manejador de la señal SIGINT
void manejador_SIGINT(int sig_num) {
    printf("Se recibió la señal SIGINT (Ctrl+C). ¡No me puedes detener tan fácilmente!\n");
}

int main() {
    // Registrar el manejador de la señal SIGINT
    signal(SIGINT, manejador_SIGINT);


    while (1) {
        printf("Ejecutando... Presiona Ctrl+C para enviar SIGINT.\n");
        sleep(1);
    }

    return 0;
}
```

Concurrencia

```c
void manejador_SIGTERM(int sig_num) {
    printf("Proceso hijo: Se recibió la señal SIGTERM. Terminando...\n");
    exit(0);  // Termina el proceso hijo
}

int main() {
    pid_t pid = fork();  // Crear un proceso hijo

    if (pid < 0) {
        perror("Error al crear el proceso hijo");
        exit(1);
    }

    if (pid == 0) {
        // HIJO
        signal(SIGTERM, manejador_SIGTERM);

        printf("Proceso hijo: PID = %d. Esperando la señal SIGTERM...\n", getpid());

        pause();  // Espera a que llegue una señal
    } else {
        // PADRE
        printf("Proceso padre: PID = %d. Enviando SIGTERM al hijo...\n", getpid());
        sleep(5);  // Esperar 5 segundos antes de enviar la señal

        kill(pid, SIGTERM);  // Enviar la señal SIGTERM al proceso hijo

        wait(NULL);
        printf("Proceso padre: El proceso hijo ha terminado.\n");
    }

    exit(0);
}
```

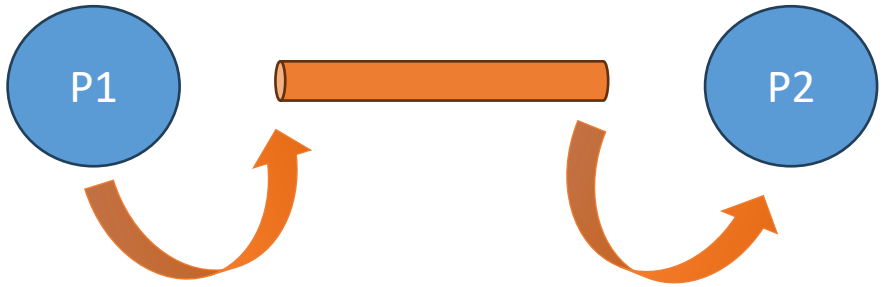
Tuberías

IPC

Tuberías

Definiciones

- Las **tuberías** o **pipes** son un mecanismo de **comunicación entre procesos (IPC)** que permite que dos procesos se comuniquen entre sí de manera eficiente, utilizando un **canal unidireccional**.
- Las tuberías permiten enviar datos de un proceso a otro, funcionando de manera similar a cómo funciona una tubería física: los datos entran por un extremo y salen por el otro.



Tuberías

Definiciones

Características principales:

•**Unidireccionales**: Las tuberías permiten que los datos fluyan en una sola dirección (de un proceso escritor a un proceso lector). Si se necesita comunicación bidireccional, deben crearse dos tuberías.

•**Anónimas o con nombre**:

- **Tuberías anónimas**: Se utilizan principalmente entre procesos relacionados (por ejemplo, procesos padre-hijo). Son temporales y no tienen un nombre en el sistema de archivos.
- **Tuberías con nombre (named pipes o FIFOs)**: Son identificadas por un nombre en el sistema de archivos y pueden ser usadas por procesos no relacionados.

•**Bloqueo**: Cuando un proceso escribe en la tubería, si no hay otro proceso leyendo, el proceso escritor puede quedar bloqueado hasta que los datos sean leídos (y viceversa).

Tuberías

Definiciones

Ejemplo de uso de una tubería en Linux:

Una tubería permite conectar la **salida estándar (stdout)** de un comando con la **entrada estándar (stdin)** de otro comando. Esto se puede hacer fácilmente en la terminal usando el carácter |

ls -l | grep ".txt"

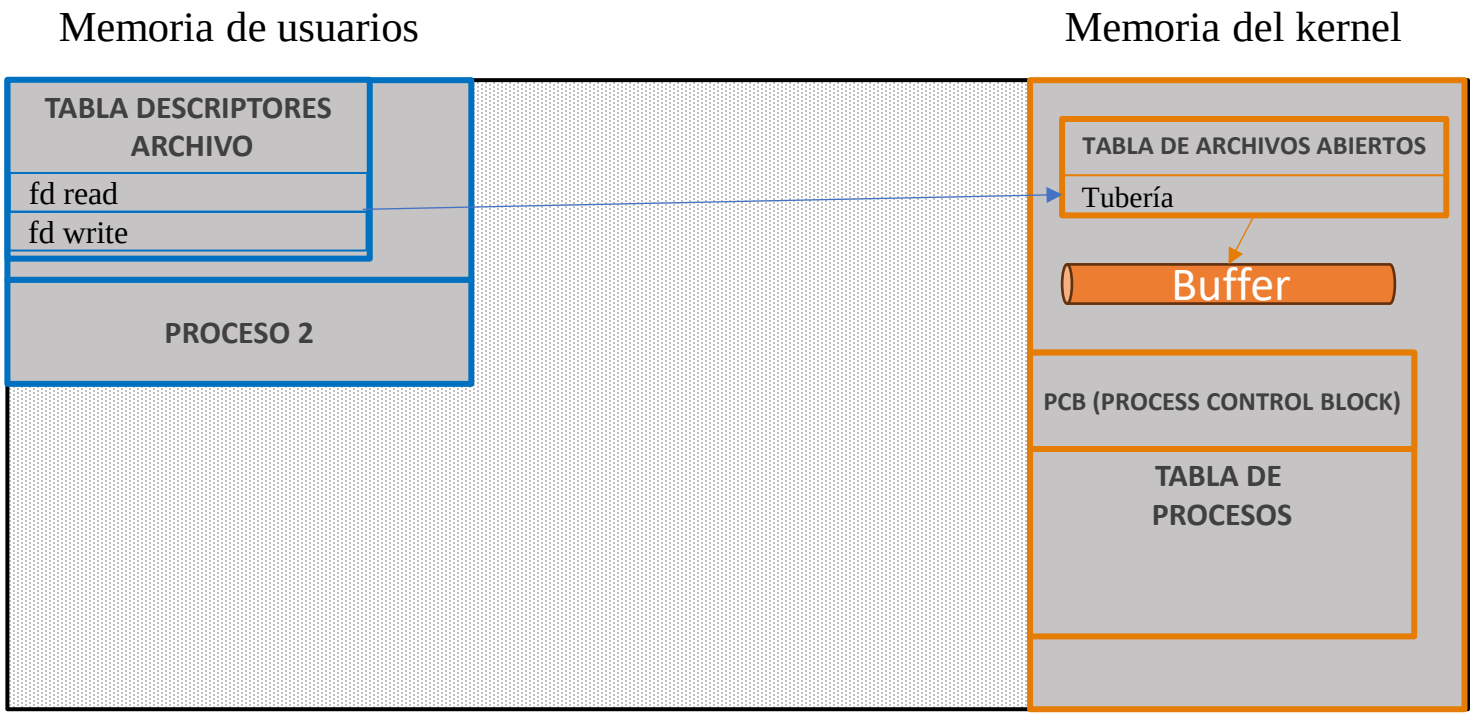
```
ls -l | grep ".txt"
```

1. El comando ls -l genera una lista de archivos, y su salida es enviada a una tubería.
2. El comando grep ".txt" recibe la salida a través de la tubería y filtra los archivos que terminan en .txt

Tuberías

Memoria

- Para la gestión de las tuberías, el SO utiliza la abstracción de archivos.
- De esta forma, cuando se crea una tubería se crean dos descriptores de archivo (uno de lectura y otro de escritura).



Tuberías

Operaciones

Operación	Descripción	Ejemplo de Uso
Crear tubería	Crea una nueva tubería, que permite la comunicación entre procesos.	pipe(fd) donde fd[0] es lectura y fd[1] es escritura.
Escribir en tubería	Envía datos a la tubería a través del descriptor de escritura.	write(fd[1], buffer, size)
Leer de tubería	Lee datos de la tubería a través del descriptor de lectura.	read(fd[0], buffer, size)
Cerrar descriptor	Cierra el extremo de lectura o escritura de la tubería.	close(fd[0]) o close(fd[1])

Tuberías

Código de ejemplo

```c
int main() {
    int fd[2]; // Descriptores de la tubería: fd[0] para lectura, fd[1] para escritura
    pid_t pid;
    char buffer[100]; // Buffer para almacenar el mensaje
    // Crear la tubería
    if (pipe(fd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    // Crear un proceso hijo
    pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) { // Proceso hijo
        close(fd[1]); // Cerrar el extremo de escritura

        // Leer el mensaje del padre
        read(fd[0], buffer, sizeof(buffer));
        printf("Mensaje recibido en el hijo: %s\n", buffer);

        close(fd[0]); // Cerrar el extremo de lectura
        exit(EXIT_SUCCESS);
    } else { // Proceso padre
        close(fd[0]); // Cerrar el extremo de lectura

        const char *mensaje = "Hola desde el proceso padre!";

        // Escribir el mensaje en la tubería
        write(fd[1], mensaje, strlen(mensaje) + 1); // +1 para incluir el carácter nulo

        close(fd[1]); // Cerrar el extremo de escritura
        wait(NULL); // Esperar a que el hijo termine
    }

    exit(EXIT_SUCCESS);
}
```

Memoria compartida

Memoria compartida

Definiciones

- La memoria compartida es un mecanismo de comunicación entre procesos que permite que varios procesos accedan a una misma región de memoria.
- Este método se utiliza para intercambiar datos de manera eficiente entre procesos que se están ejecutando en el mismo sistema operativo.
- La memoria compartida es particularmente útil cuando se requiere un alto rendimiento y baja latencia en la comunicación entre procesos.

Memoria Compartida

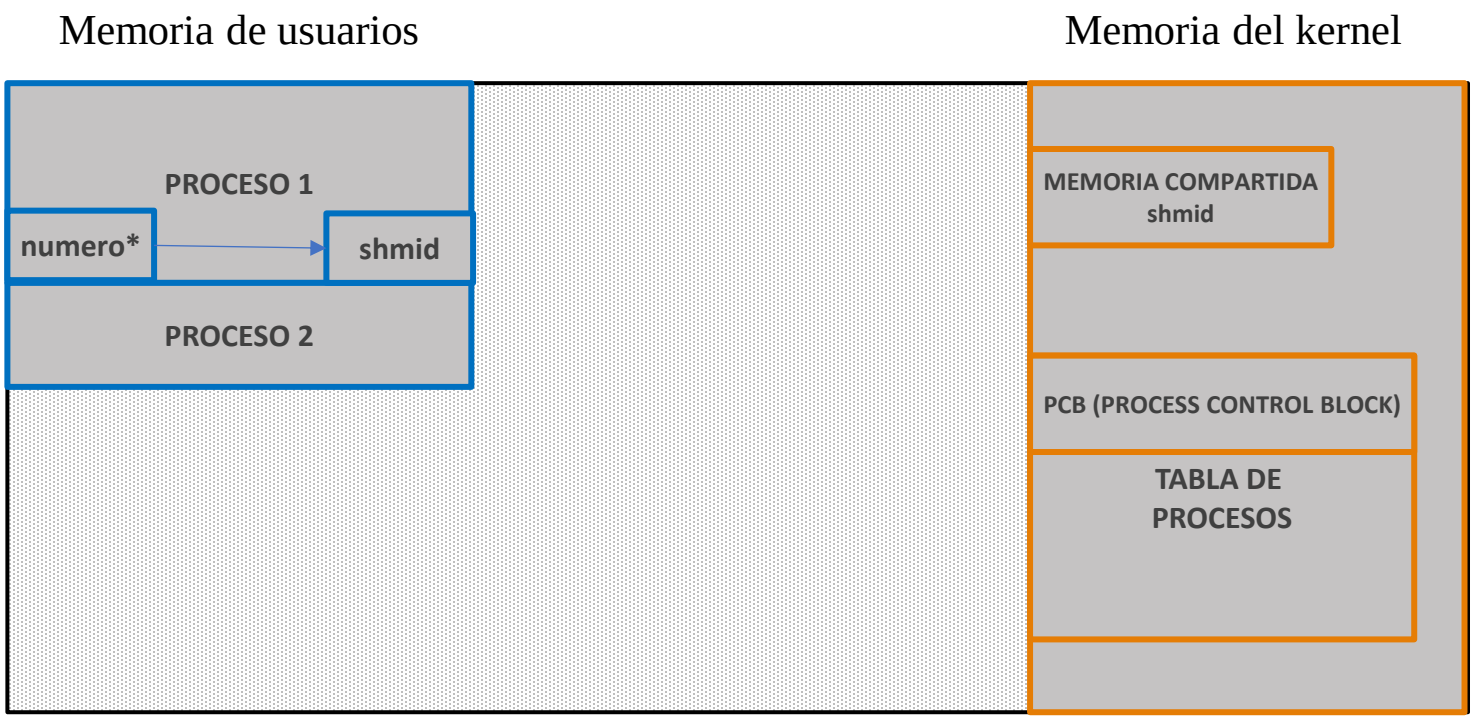
Operaciones

```
shmid=shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT|0666)
numero=(int *) shmat (shmid,0,0);
```

Adjunta la MC al proceso

```
shmdt(numero)
```

Desadjunta la memoria del proceso



Memoria Compartida

Operaciones

```
shmid=shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT|0666)
numero=(int *) shmat (shmid,0,0);
```

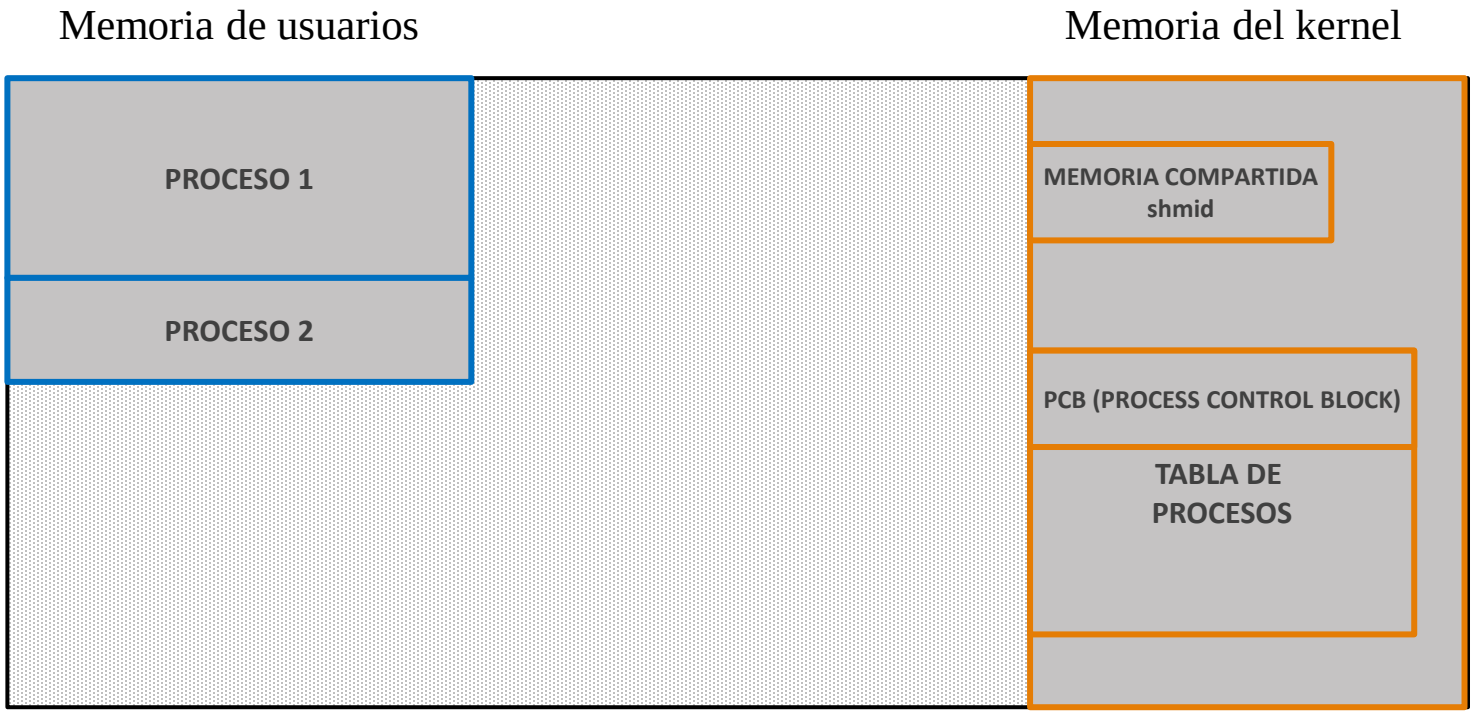
Adjunta la MC al proceso

```
shmdt(numero)
```

Desadjunta la memoria del proceso

```
shmctl(shmid,IPC_RMID,0)
```

Marca la MC para ser borrada cuando no haya procesos que la tengan adjunta.



Memoria Compartida

Operaciones

```
shmid=shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT|0666)
numero=(int *) shmat (shmid,0,0);
```

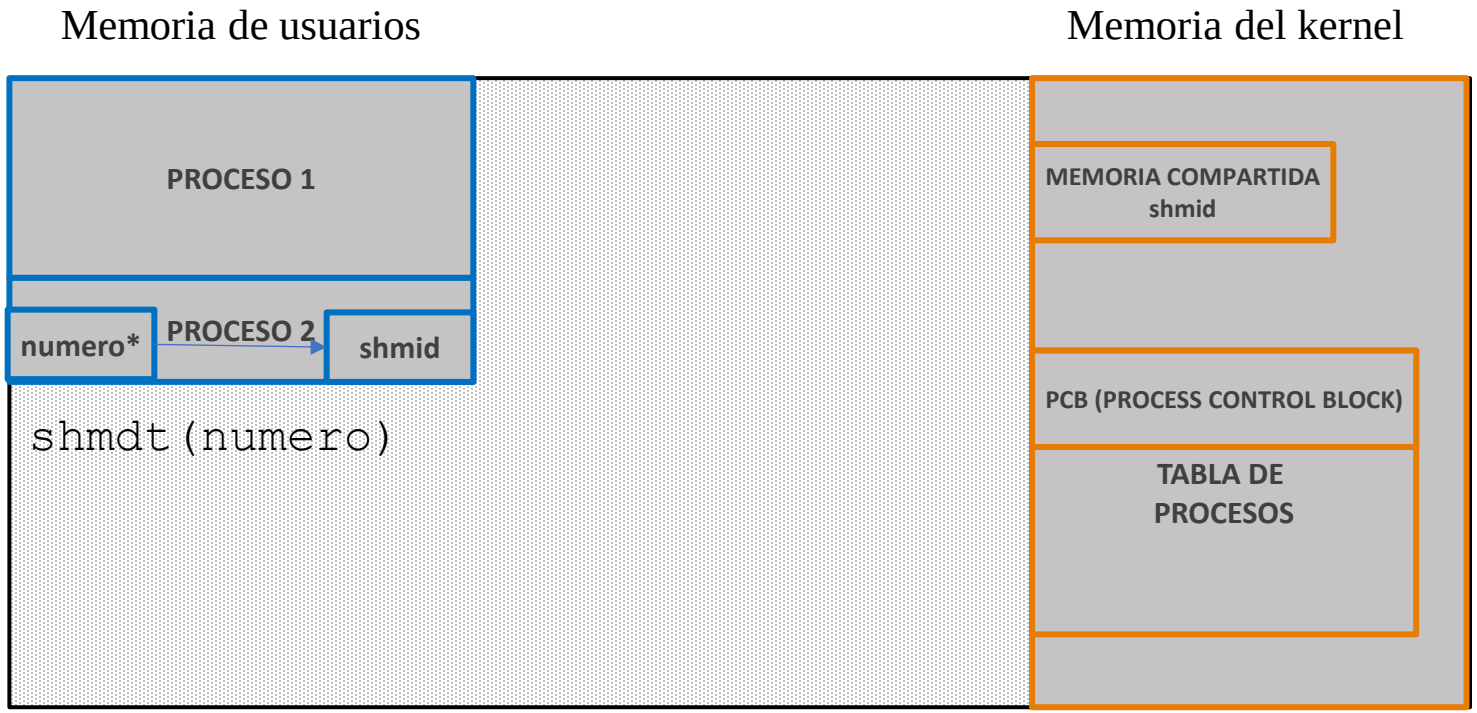
Adjunta la MC al proceso

```
shmdt(numero)
```

Desadjunta la memoria del proceso

```
shmctl(shmid,IPC_RMID,0)
```

Marca la MC para ser borrada cuando no haya procesos que la tengan adjunta.



Memoria Compartida

Operaciones

```
shmid=shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT|0666)
numero=(int *) shmat (shmid,0,0);
```

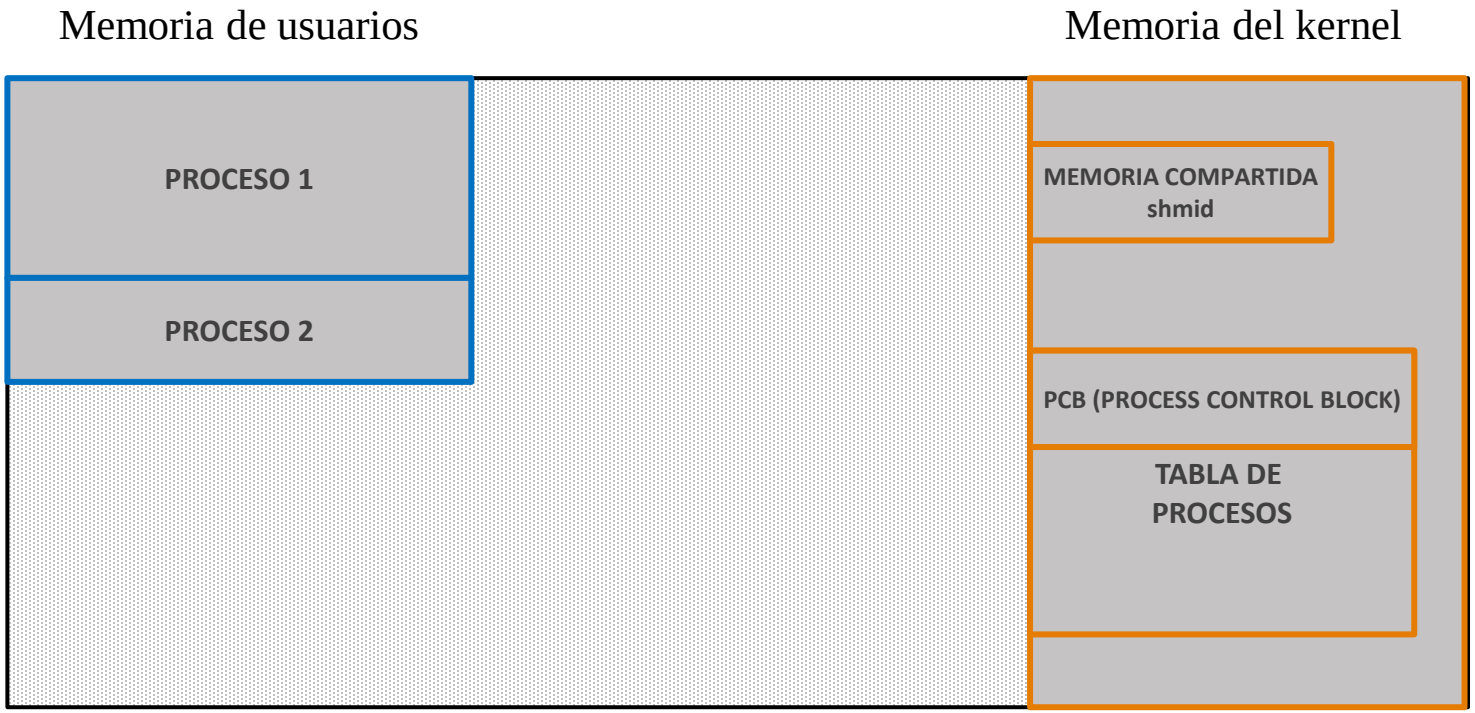
Adjunta la MC al proceso

```
shmdt(numero)
```

Desadjunta la memoria del proceso

```
shmctl(shmid,IPC_RMID,0)
```

Marca la MC para ser borrada cuando no haya procesos que la tengan adjunta.



Memoria Compartida

Operaciones

```
shmid=shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT|0666)
numero=(int *) shmat (shmid,0,0);
```

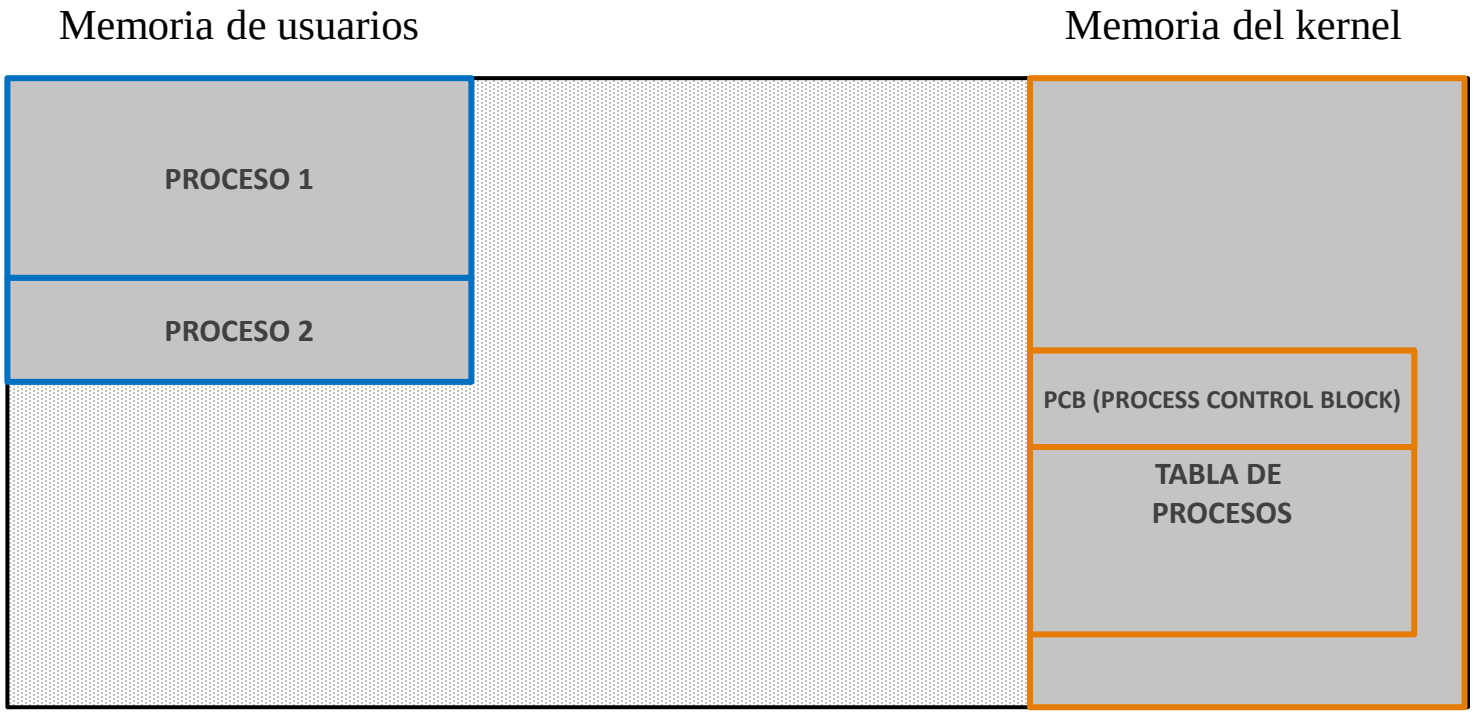
Adjunta la MC al proceso

```
shmdt(numero)
```

Desadjunta la memoria del proceso

```
shmctl(shmid,IPC_RMID,0)
```

Marca la MC para ser borrada cuando no haya procesos que la tengan adjunta.



Memoria Compartida

Código de ejemplo sin control de errores exhaustivo

```
int main() {
    int shmid = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | 0666);

    // Crear un proceso hijo
    pid_t pid = fork();

    if (pid == 0) { // Proceso hijo

        int *data = (int *) shmat(shmid, NULL, 0);// Adjuntar el segmento de memoria compartida
        if (data == (int *) -1) {
            perror("Error al adjuntar el segmento en el hijo");
            exit(1);
        }

        printf("Proceso hijo: El número en memoria compartida es: %d\n", *data);
        if (shmdt(data) < 0) {
            perror("Error al desadjuntar el segmento en el hijo");
            exit(1);
        }
    } else {
        // Proceso padre
        int *data = (int *) shmat(shmid, NULL, 0);
        *data = 42;  // Almacena el número 42
        if (shmdt(data) < 0) {
            perror("Error al desadjuntar el segmento en el padre");
            exit(1);
        }
        wait(NULL);
        if (shmctl(shmid, IPC_RMID, NULL) < 0) {
            perror("Error al eliminar el segmento");
            exit(1);
        }
    }

    exit (0)
}
```

¿Problema de sincronización?

Memoria Compartida

Código de ejemplo sin control de errores exhaustivo

```
int *data;  // Puntero a la memoria compartida

void manejador(int signum) {
    printf("Proceso hijo: El número en memoria compartida es: %d\n", *data);
    shmdt(data) //Falta controlar errores
    exit(0);
}

int main() {
    int shmid = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | 0666);

    pid_t pid = fork();

    if (pid == 0) {  // Proceso hijo

        data = (int *) shmat(shmid, NULL, 0);
        signal(SIGUSR1, manejador);
        pause();  // El proceso hijo queda en pausa hasta recibir la señal

    } else {  // Proceso padre

        data = (int *) shmat(shmid, NULL, 0);
        *data = 42;
        printf("Proceso padre: Número 42 escrito en memoria compartida\n");

        kill(pid, SIGUSR1);

        wait(NULL);

        shmdt(data) //Falta controlar errores

        if (shmctl(shmid, IPC_RMID, NULL) == -1) {
            perror("Error al eliminar el segmento de memoria compartida");
            exit(1);
        }
        printf("Proceso padre: Memoria compartida eliminada\n");
    }
    exit(0);
}
```

Tema 1.3 Sistema de Archivos, Concurrencia e IPC

Contenidos

Sistema de archivos
Estructura, Interfaz

Concurrencia
Definición, problemas y sincronización

IPC
Señales, tuberías, memoria compartida y sockets

Sistemas Operativos y Distribuidos

Iren Lorenzo Fonseca
iren.fonseca@.ua.es

TEMA 1. Sistemas Operativos.

Sistema de Archivos, Concurrencia e IPC