

01. Programación multiproceso

Contenidos del tema

1. Procesos en el Sistema Operativo

- 1.1 Introducción
- 1.2 Estados de un proceso
- 1.3 Control de procesos en Linux
- 1.4 Creación de procesos en C/C++
- 1.5 Comunicación entre procesos en C/C++
- 1.6 Sincronización entre procesos en C/C++
- 1.7 Creación de procesos en Java

2. Programación concurrente

- 2.1 Programa-Proceso
- 2.2 Características
- 2.3 Programas concurrentes
- 2.4 Problemas de la programación concurrente
- 2.5 Programación concurrente en Java

3. Programación paralela y distribuida

- 3.1 Programación paralela
- 3.2 Programación distribuida
- 3.3 Framework de programación distribuida: PVM

Contenidos de la sección

1. Procesos en el Sistema Operativo

- 1.1 Introducción
- 1.2 Estados de un proceso
- 1.3 Control de procesos en Linux
- 1.4 Creación de procesos en C/C++
- 1.5 Comunicación entre procesos en C/C++
- 1.6 Sincronización entre procesos en C/C++
- 1.7 Creación de procesos en Java

1.1 Introducción

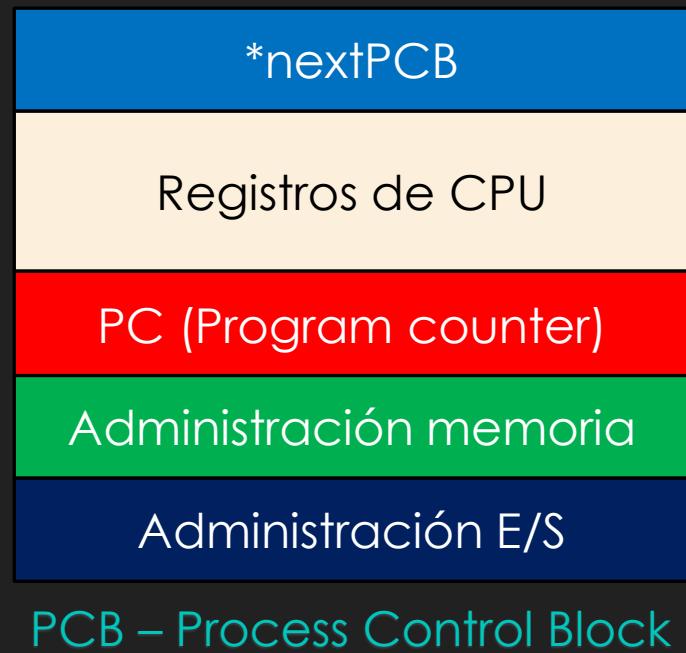
Sistema operativo multiproceso

- Un **sistema operativo multiproceso** puede ejecutar más de un proceso a la vez
- El usuario tiene la sensación de que cada proceso es el único en ejecución (sensación falsa)
- Los recursos (tiempo de **CPU** y la **RAM**) se reparten entre los procesos
- Diversos procesos pueden estar ejecutándose sobre el mismo código o programa
- Una aplicación puede realizar **tareas simultáneas**

1.1 Introducción

Algunas definiciones

- **Proceso:** programa en ejecución
 - **Contenido**
 - Código ejecutable del programa
 - Datos, pila, contador de programa (PC)
 - **Descriptor de un proceso (PCB - Process Control Block)**
 - Estructura que almacena datos del proceso
 - Se define en el momento de creación del proceso
 - Cada proceso tiene un PCB único
 - Se destruye cuando finaliza el proceso



1.1 Introducción

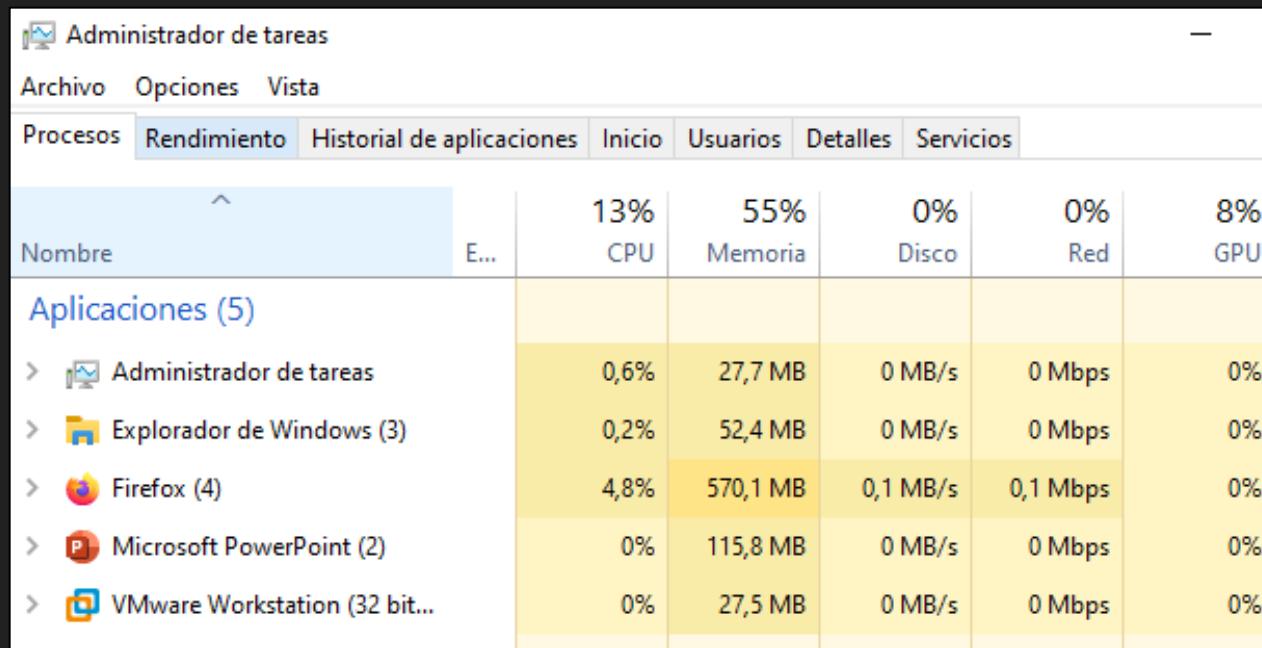
Obteniendo información sobre los procesos

- En **linux**:
 - El comando **ps** proporciona información sobre los procesos
 - Para mostrar más información podemos usar **ps -f** o **ps -Af**

1.1 Introducción

Obteniendo información sobre los procesos

- En Windows:
 - Podemos usar el comando **tasklist** o el **administrador de tareas**

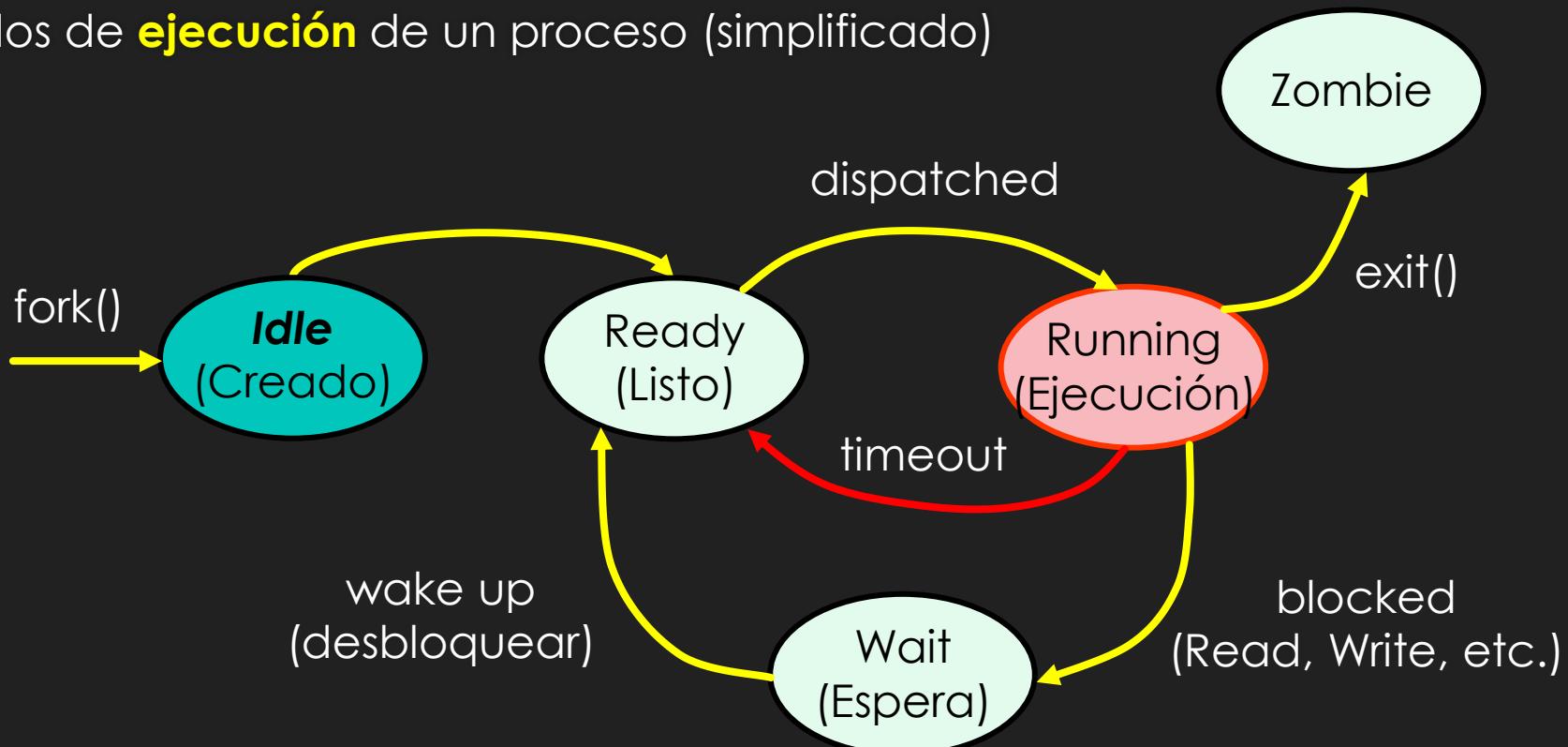


Nombre	E...	13%	55%	0%	0%	8%
		CPU	Memoria	Disco	Red	GPU
Aplicaciones (5)						
>  Administrador de tareas		0,6%	27,7 MB	0 MB/s	0 Mbps	0%
>  Explorador de Windows (3)		0,2%	52,4 MB	0 MB/s	0 Mbps	0%
>  Firefox (4)		4,8%	570,1 MB	0,1 MB/s	0,1 Mbps	0%
>  Microsoft PowerPoint (2)		0%	115,8 MB	0 MB/s	0 Mbps	0%
>  VMware Workstation (32 bit...)		0%	27,5 MB	0 MB/s	0 Mbps	0%

1.2 Estados de un proceso

Esquema de los estados de un proceso

- Estados de ejecución de un proceso (simplificado)



1.3 Control de procesos en Linux

Funciones para gestionar procesos en Linux

- Linux ofrece las siguientes funciones para gestionar procesos:
 - **system()**
 - Incluida en **stdlib.h** (funciona en cualquier sistema operativo con un compilador C/C++)
 - Permite la ejecución de comandos del sistema operativo
 - **exec()**
 - Permite la ejecución de comandos del sistema con privilegios de administrador
 - **fork()**
 - Se utiliza para crear procesos

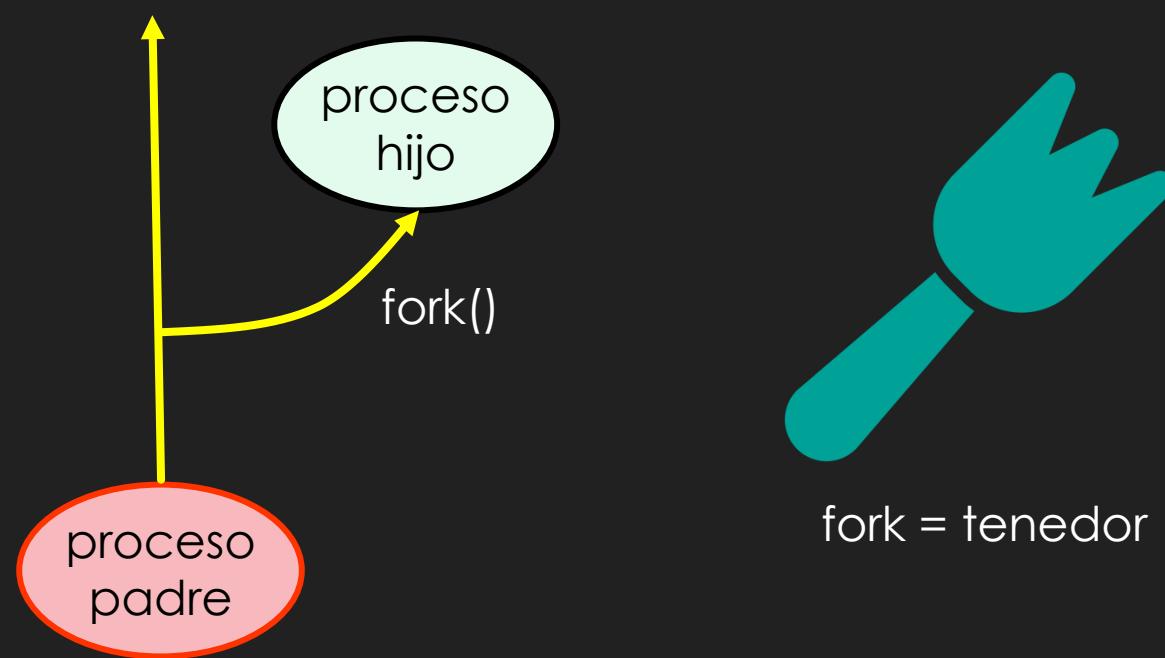
1.4 Creación de procesos en C/C++

¿Cómo crear un nuevo proceso?

- La función **fork()** permite crear procesos desde C/C++
 - El nuevo proceso (hijo) es una copia exacta del proceso que ejecuta la llamada fork()
 - El proceso recibe una copia idéntica de las variables del padre (¡pero no son punteros!)
 - El proceso hijo tiene su propio **PID**, su **espacio de memoria** y un **PCB** independiente
 - Valor devuelto por **fork()**:
 - -1 si se produce un error
 - 0 si no hay error y nos encontramos en el proceso hijo
 - El PID asociado al hijo si no se produce ningún error y nos encontramos en el proceso padre

1.4 Creación de procesos en C/C++

¿Cómo crear un nuevo proceso?



fork = tenedor

1.5 Comunicación entre procesos en C/C++

Si son independientes, ¿cómo los comunicamos entre sí?

- Existen diversos mecanismos: pipes, colas de mensajes, semáforos y segmentos de memoria
- En este tema usaremos el mecanismo más sencillo, las **tuberías** (pipe)
- Una **tubería** actúa como un **fichero compartido** que permite intercambiar información
- Permite **intercambiar** información entre uno o más procesos
- Si el pipe está **vacío**, un proceso se **bloquea** hasta que tenga datos que pueda consumir

1.5 Comunicación entre procesos en C/C++

Si son independientes, ¿cómo los comunicamos entre sí?

- Funciones importantes para gestionar tuberías:
 - **pipe(int fd[2])**: crea la tubería (recibe dos descriptores, uno para lectura y el otro para escritura)
 - **write(int fd, void *buf, int count)**: graba del buffer buf en el descriptor indicado N bytes
 - **read(int fd, void *buf, int count)**: lee del descriptor indicado N bytes y lo guarda en el buffer buf
 - **open(const char *fichero, int modo)**: abre el fichero indicado en la cadena **fichero**
 - **close()**: para cerrar el fichero

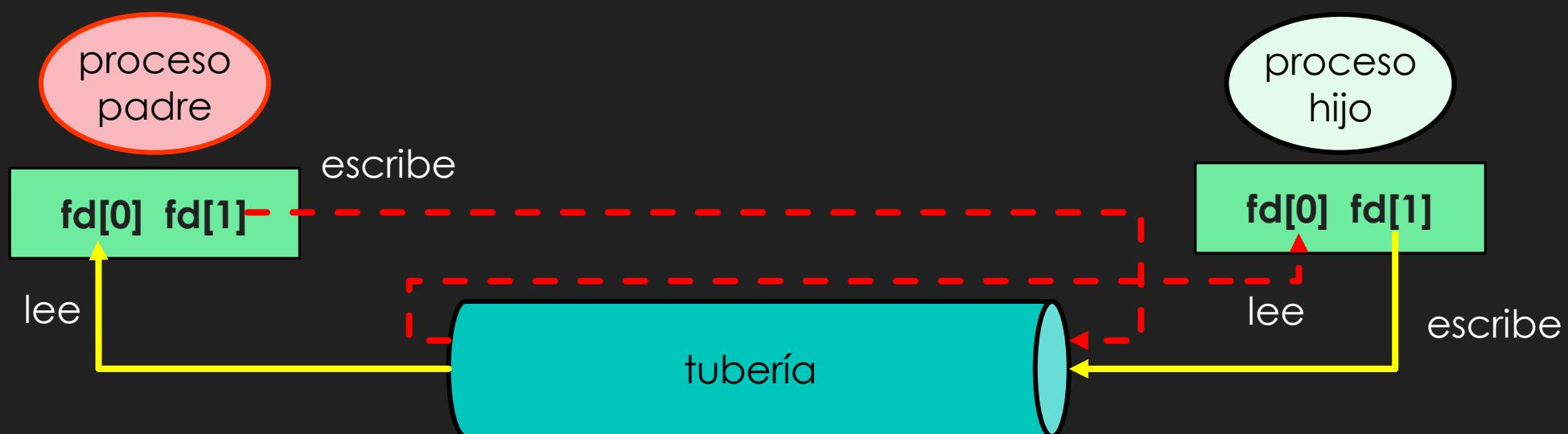
1.5 Comunicación entre procesos en C/C++

Funcionamiento de las tuberías

- Pasos a seguir
 - Creación de la tubería
 - Creación del proceso hijo (comparte la tubería con el padre)
 - Se establece la comunicación (**unidireccional**, pero hay un **truco**)

1.5 Comunicación entre procesos en C/C++

Funcionamiento de las tuberías



Las tuberías tienen un único sentido

1.6 Sincronización de procesos en C/C++

Si son independientes, ¿cómo los comunicamos entre sí?

- Para que los procesos interactúen entre sí necesitan sincronizarse
- Es decir, tienen que operar de forma coordinada
- La mejor forma de coordinarlos es usando señales
- La función **signal()** especifica la acción a realizar cuando un proceso recibe una señal

1.6 Sincronización de procesos en C/C++

Algunas funciones útiles

- **signal()**: permite capturar una señal recibida en un proceso
- **kill()**: envía una señal a un proceso, especificando el PID y la señal a enviar
- **pause()**: hace que un proceso se bloquee a la espera de una señal
- **sleep()**: suspende el proceso una cantidad de segundos o hasta que se reciba una señal

1.6 Sincronización de procesos en C/C++

Analicemos ahora algunos ejemplos

- **Estos ejemplos deben ser utilizados en una máquina Linux**
- **Requisitos**
 - Tener instalado el compilador GCC: sudo apt-get install gcc
- **Compilación de los ejemplos**
 - `gcc -o ejemplo.out ejemplo.c`
- **Ejecución de los ejemplos**
 - `chmod u+x ejemplo.out`

1.7 Creación de procesos en Java

¿Cómo crear un proceso en Java?

- **Clase ProcessBuilder**

- Cada instancia gestiona una serie de atributos del proceso
- El método **start()** crea una nueva instancia de tipo **Process** con dichos atributos
- El proceso puede ser invocado varias veces para crear nuevos subprocessos
- El proceso creado redirige su terminal a la del proceso padre
- **Métodos de la clase Process:** [enlace](#)
- **Métodos de la clase ProcessBuilder:** [enlace](#)

Contenidos de la sección

2. Programación concurrente

- 2.1 Programa-Proceso
- 2.2 Características
- 2.3 Programas concurrentes
- 2.4 Problemas de la programación concurrente
- 2.5 Programación concurrente en Java

2.1 Programa-Proceso

¿Qué es la programación concurrente?

- La **concurrentia** permite que varios sucesos ocurran en el mismo tiempo
- Esta característica es posible gracias a la existencia de **múltiples unidades de proceso**
- En cada unidad de proceso se puede ejecutar un **hilo** al mismo tiempo
- Un programa es un conjunto de instrucciones
- Las instrucciones de un programa se aplican a unos datos de entrada para obtener resultados
- Un programa en ejecución puede dar lugar a más de un proceso

2.1 Programa-Proceso

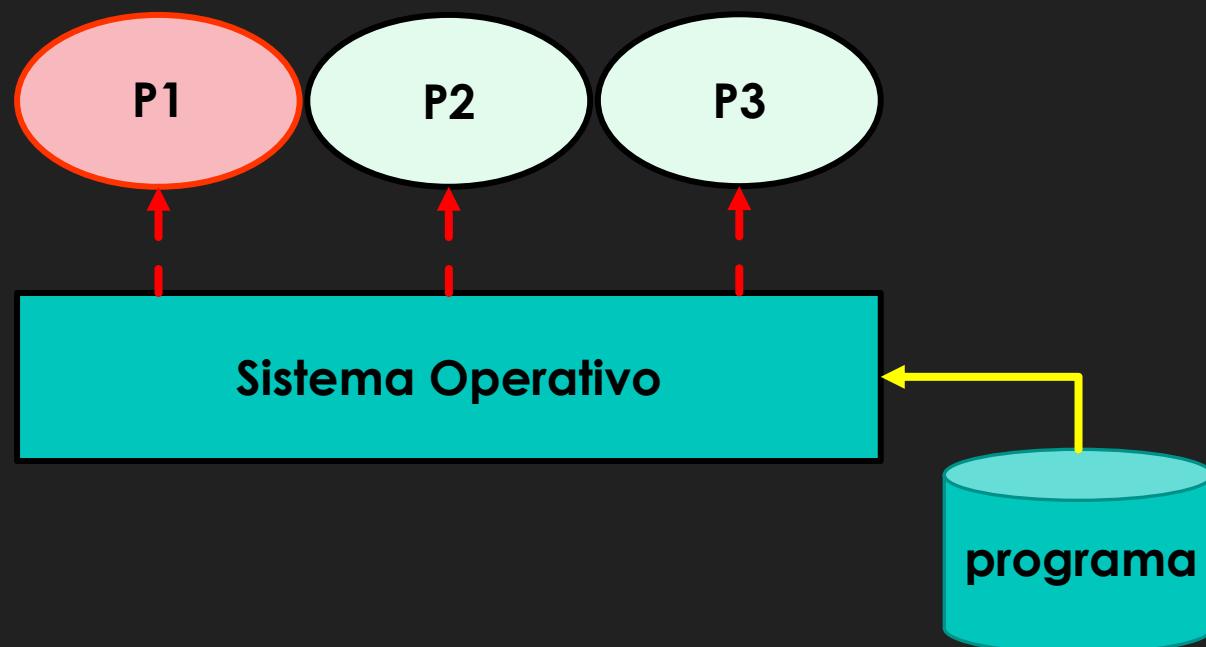
¿Qué es la programación concurrente?

- Dos **procesos** serán **concurrentes** cuando sus instrucciones se **intercalan**
- **No confundir con la programación paralela**
- Dicho esto, podemos considerar que un proceso es:
 - Toda actividad asíncrona que puede ser asignada a un procesador

2.1 Programa-Proceso

¿Qué es la programación concurrente?

- Ejemplo de un programa que produce más de un proceso



2.1 Programa-Proceso

¿Qué es la programación concurrente?

- Cuando varios procesos se ejecutan de forma **concurrente** pueden **colaborar** o **competir**
 - **Colaborar**: los procesos colaboran entre sí para lograr un fin común
 - **Competir**: los procesos compiten por los recursos del sistema
- Debido a esta naturaleza el **Sistema Operativo** incorpora los siguientes mecanismos:
 - **Mecanismos de comunicación**: permiten la comunicación entre los procesos
 - **Mecanismos de sincronización**: permiten sincronizar los procesos y el acceso a los recursos
- Los programas concurrentes tienen un **flujo parcial** (comportamiento **indeterminista**)

2.2 Características

Beneficios de la programación concurrente

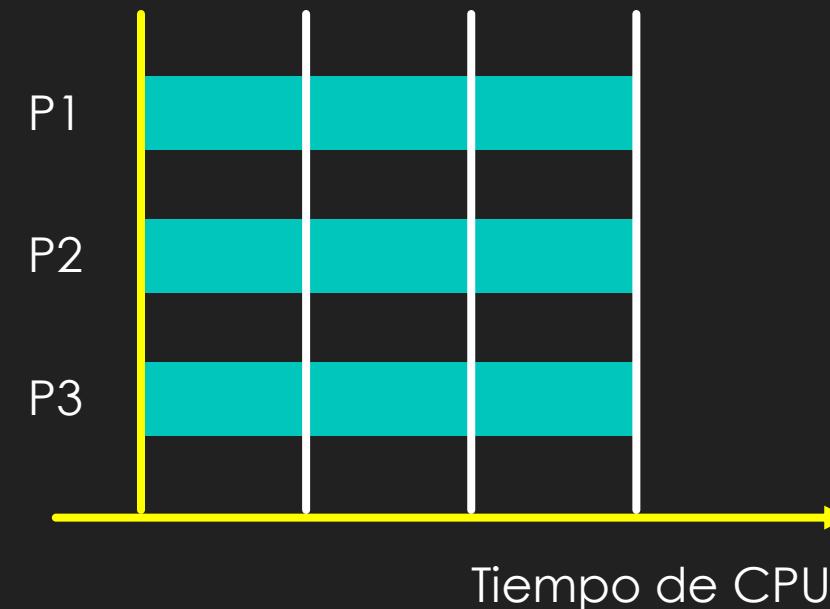
- Mejor aprovechamiento de la CPU
- Incremento de la velocidad de ejecución
- Solución de problemas cuya naturaleza es concurrente:
 - Sistemas de control
 - Tecnologías web
 - Aplicaciones con GUI
 - Simulación
 - Sistemas Gestores de Bases de Datos

2.2 Características

Concurrencia en Sistemas monoprocesador y multiprocesador



Sistemas monoprocesador



Sistemas multiprocesador

2.3 Programa concurrente

Ejecución simultánea

- Conjunto de acciones que pueden ser ejecutadas de forma simultánea

x=x+1;
y=x+1; El orden de ejecución **importa**

x=1;
y=1; El orden de ejecución **no importa**
z=1;

2.3 Programa concurrente

Condiciones de Bernstein

- Estas condiciones permiten que dos conjuntos de instrucciones se ejecuten a la vez
- Para esto hay que formar dos conjuntos de instrucciones:
 - **Instrucciones de lectura**: formado por instrucciones que tienen variables que son leídas en ejecución
 - **Instrucciones de escritura**: formado por instrucciones que tienen variables que son escritas en ejecución
- Ejemplo:

x=y+1; Instrucción 1

y=x+2; Instrucción 2

z=a+b; Instrucción 3

2.3 Programa concurrente

Condiciones de Bernstein (ejemplo)

- Conjuntos de lectura y escritura

	Lectura (L)	Escritura (E)	
Instrucción 1 I1	y	x	$x=y+1;$
Instrucción 2 I2	x	y	$y=x+2;$
Instrucción 3 I3	a,b	z	$z=a+b;$
	$L(I1)=\{y\}$	$E(I1)=\{x\}$	
	$L(I2)=\{x\}$	$E(I2)=\{y\}$	
	$L(I3)=\{a,b\}$	$E(I3)=\{z\}$	

2.3 Programa concurrente

Condiciones de Bernstein (ejemplo)

- **Para que dos conjuntos se puedan ejecutar tienen que cumplir 3 condiciones**
 - La intersección de las variables de lectura de un conjunto i y las de escritura de un conjunto j debe ser vacía
$$L(I_i) \cap E(I_j) = \emptyset$$
 - La intersección de las variables de escritura de un conjunto i y las de lectura de un conjunto j debe ser vacía
$$E(I_i) \cap L(I_j) = \emptyset$$
 - La intersección de las variables de escritura de un conjunto i y las de escritura de un conjunto j debe ser vacía
$$E(I_i) \cap E(I_j) = \emptyset$$

2.3 Programa concurrente

Condiciones de Bernstein (ejemplo)

- Para que dos conjuntos se puedan ejecutar tienen que cumplir 3 condiciones

Conjunto I1 e I2	Conjunto I2 e I3	Conjunto I1 e I3
$L(I_1) \cap E(I_2) \neq \emptyset$	$L(I_2) \cap E(I_3) = \emptyset$	$L(I_1) \cap E(I_3) = \emptyset$
$E(I_1) \cap L(I_2) \neq \emptyset$	$E(I_2) \cap L(I_3) = \emptyset$	$E(I_1) \cap L(I_3) = \emptyset$
$E(I_1) \cap E(I_2) = \emptyset$	$E(I_2) \cap E(I_3) = \emptyset$	$E(I_1) \cap E(I_3) = \emptyset$

2.3 Programa concurrente

Condiciones de Bernstein (ejemplo)

- Para que dos conjuntos se puedan ejecutar tienen que cumplir 3 condiciones

Conjunto I1 e I2	Conjunto I2 e I3	Conjunto I1 e I3
$L(I_1) \cap E(I_2) \neq \emptyset$	$L(I_2) \cap E(I_3) = \emptyset$	$L(I_1) \cap E(I_3) = \emptyset$
$E(I_1) \cap L(I_2) \neq \emptyset$	$E(I_2) \cap L(I_3) = \emptyset$	$E(I_1) \cap L(I_3) = \emptyset$
$E(I_1) \cap E(I_2) = \emptyset$	$E(I_2) \cap E(I_3) = \emptyset$	$E(I_1) \cap E(I_3) = \emptyset$

- Las instrucciones I1 e I2 no se pueden ejecutar concurrentemente

```
x=y+1;  
y=x+2;  
z=a+b;
```

2.4 Problemas de la programación concurrente

Problemas que podemos encontrarnos

- **Exclusión mutua**
 - Ocurre cuando varios procesos acceden a la misma variable a modificarla (Ejemplo: BBDD)
 - Uno actualiza a la vez que otro podría estar leyendo
 - La variable se **protege** creando una **sección crítica**
- **Condición de sincronización**
 - Necesidad de coordinar las actividades de dos procesos
 - Puede suceder que un proceso P1 necesite que P2 finalice para poder ejecutarse

2.5 Programación concurrente en Java

Programación concurrente en Java: Hilos

- En Java podemos crear **varios hilos de ejecución** dentro de un mismo **proceso**
- Los hilos comparten el **contexto de ejecución** del proceso (datos compartidos)
- Los hilos son llamados **procesos ligeros**
- Se tarda menos tiempo en crear un **hilo** que un proceso
- La comunicación entre procesos requiere el **núcleo** del SO, en los hilos no hace falta
- Generalmente, los hilos **cooperan** para resolver un problema

2.5 Programación concurrente en Java

Programación concurrente en Java: Hilos

- Existen dos formas:
 - Crear una clase que herede de la clase **Thread** y sobrecargar el método **run()**
 - Implementar la interfaz **Runnable** y declarar el método **run()**
 - Este método se usa cuando una clase ya hereda de otra (herencia múltiple en Java)

Contenidos de la sección

3. Programación paralela y distribuida

- 3.1 Programación paralela
- 3.2 Programación distribuida
- 3.3 Framework de programación distribuida: PVM

3.1 Programación paralela

Descripción

- Un programa paralelo está pensado para ejecutarse en un sistema multiprocesador
- Varias CPU pueden trabajar a la vez para resolver el problema
- El problema se divide en partes
- **Más información:** [video](#)

3.1 Programación paralela

Ventajas e inconvenientes

- **Ventajas**
 - Ejecución simultánea
 - Disminuye el tiempo de ejecución
 - Abordaje y resolución de problemas complejos

3.1 Programación paralela

Ventajas e inconvenientes

■ Inconvenientes

- Los compiladores y entornos son más complejos
- Los programas son más difíciles de escribir
- Mayor consumo de energía
- Mayor complejidad en el acceso a los datos
- Comunicación y sincronización más compleja

3.2 Programación distribuida

Descripción

- Los componentes hardware o software están distribuidos
- El ejemplo más conocido es Internet (cloud computing)
- Los componentes se interconectan mediante una red
- **Consecuencias:**
 - Concurrencia
 - Inexistencia de un reloj global
 - Fallos independientes (Sería como un RAID a nivel de programa)

3.2 Programación distribuida

Modelos de programación para la comunicación en entornos distribuidos

- Sockets
- Llamada a procedimientos remotos (RPC)
- Invocación remota de objetos (RMI)

3.2 Programación distribuida

Ventajas e inconvenientes

■ Ventajas

- Se pueden compartir recursos y datos
- Capacidad de crecimiento incremental
- Mayor flexibilidad
- Alta disponibilidad
- Carácter abierto y heterogéneo

3.2 Programación distribuida

Ventajas e inconvenientes

■ Inconvenientes

- Aumento de complejidad
- Problemas con las redes de comunicación (pérdida de mensajes, saturación...)
- Problemas de seguridad (DDoS)

3.3 Framework de programación distribuida: PVM

PVM (Parallel Virtual Machine)

- Permiten crear un marco de computación concurrente, distribuido y de propósito general
- **Más información:** [enlace](#)
 - **AVISO:** El video tiene audio a un volumen muy estridente

Créditos de las imágenes y figuras

Cliparts e iconos

- **Obtenidos mediante la herramienta web [IconFinder](#)** (según sus disposiciones):
 - Diapositiva 1
 - Según la plataforma IconFinder, dicho material puede usarse libremente (free commercial use)
 - A fecha de edición de este material, todos los cliparts son free for commercial use (sin restricciones)

Diagramas, gráficas e imágenes

- Se han desarrollado en PowerPoint y se han incrustado en esta presentación
- Todos estos materiales se han desarrollado por el autor
- Para el resto de recursos se han especificado sus fabricantes, propietarios o enlaces
- Si no se especifica copyright con la imagen, entonces es de desarrollo propio o CC0