

Programación de servicios y procesos

UNIDAD 1

Programación multiproceso

CONTENIDOS

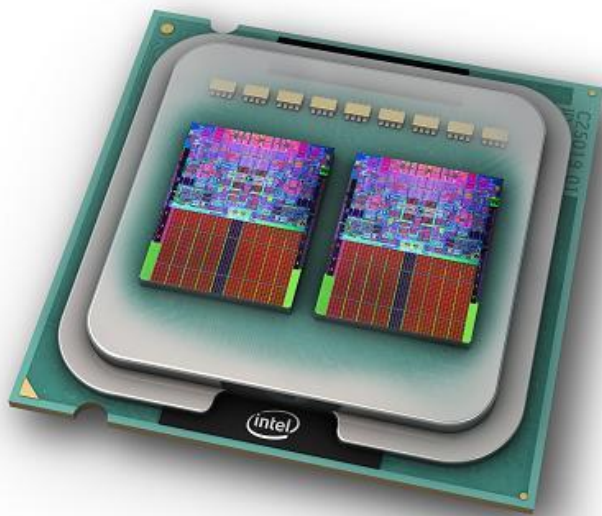
- 1.1.** Introducción a los sistemas multitarea
- 1.2.** Procesos: conceptos teóricos
- 1.3.** Programación de aplicaciones multiplataforma en Java

Introducción

Los ordenadores son capaces de realizar muchas tareas de manera simultánea, pese a que el número de procesadores que tienen no es muy alto habitualmente. En parte es un engaño, ya que las tareas se van ejecutando por turnos, pero a tal velocidad que el ojo humano no es capaz de apreciar las ínfimas discontinuidades en la ejecución.

En otras ocasiones, en cambio, el procesamiento es realmente simultáneo. Los procesadores modernos tienen varios núcleos de ejecución que funcionan como procesadores de facto. Cada núcleo es capaz de ejecutar una instrucción en un instante dado, por lo que se podrán ejecutar a la vez tantas instrucciones como núcleos de manera real.

También existe la posibilidad de utilizar varios ordenadores y construir una red en la que los elementos se distribuyan el trabajo para hacer bueno aquello de «la unión hace la fuerza». Varios procesadores con distintos núcleos trabajan a la vez resolviendo diversos procesos de manera simultánea.



<https://infocuatro.com.ar/como-funciona-un-procesador-de-varios-nucleos-2/>

Multitarea y monotarea

- Los ordenadores ejecutan cientos de tareas que se escapan a ojos del usuario, ya que va tan rápido que parece que se ejecutan a la vez, aunque se ejecuten de forma secuencial.
- En otras ocasiones, en cambio, el procesamiento es realmente simultáneo. Los procesadores modernos tienen varios núcleos de ejecución que funcionan como procesadores. Cada **núcleo** es capaz de ejecutar una instrucción en un instante dado, por lo que se pueden ejecutar tantas instrucciones a la vez cómo núcleos existan.

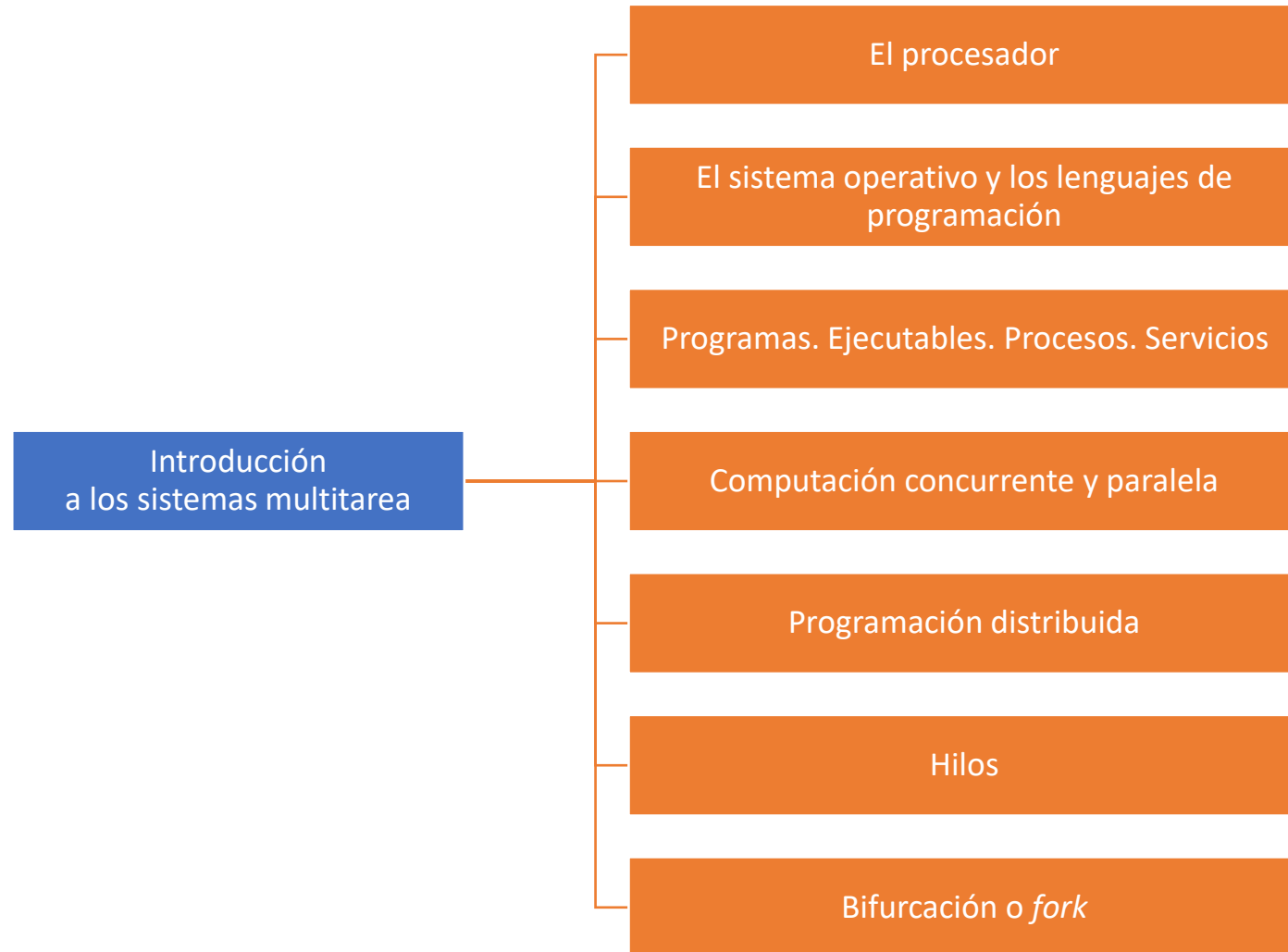
1.1. Introducción a los sistemas multitarea

Un ordenador actual de potencia media con un procesador con cuatro núcleos y correctamente configurado es capaz de realizar simultáneamente varias tareas.



Figura 1.1. Los antiguos sistemas monotarea no permitían ejecutar más de un programa a la vez.

UNIDAD 1. Programación multiproceso



Cualquier procesador puede, potencialmente, ejecutar varias tareas y que parezca que las realiza todas simultáneamente.



Figura 1.2. Los antiguos procesadores solo disponían de un núcleo, pero permitían la multitarea.



Actividad propuesta 1.1

Comparación de procesadores

Compara algún procesador de la familia Intel 486 o i486 y alguno de la familia Intel Core i3, i5 o i7. Examina parámetros como la velocidad, el número de bits y el número de núcleos. Intenta comparar los precios actualizados de unos y otros.

Recuerde utilizar las normas y plantillas para la realización de informes





Figura 1.3. El sistema operativo es el director de orquesta de un computador.

El **sistema operativo** hace de intermediario entre los programas y el hardware del ordenador.

Cuando se pulsa una tecla del teclado o se mueve el ratón, es el sistema operativo quien detecta el evento y hace posible su gestión. Cuando se accede a una variable en un programa, es el sistema operativo quien va a la memoria RAM a buscar su valor.

Cuando se almacena un fichero en un disco duro mecánico, es el sistema operativo quien mueve el motor y desplaza las cabezas lectoras.

Sabías que...



Las clasificaciones de los lenguajes de programación no son estancas, sino que conviven entre sí. Por ejemplo, Java es un lenguaje de alto nivel, de propósito general y orientado a objetos.

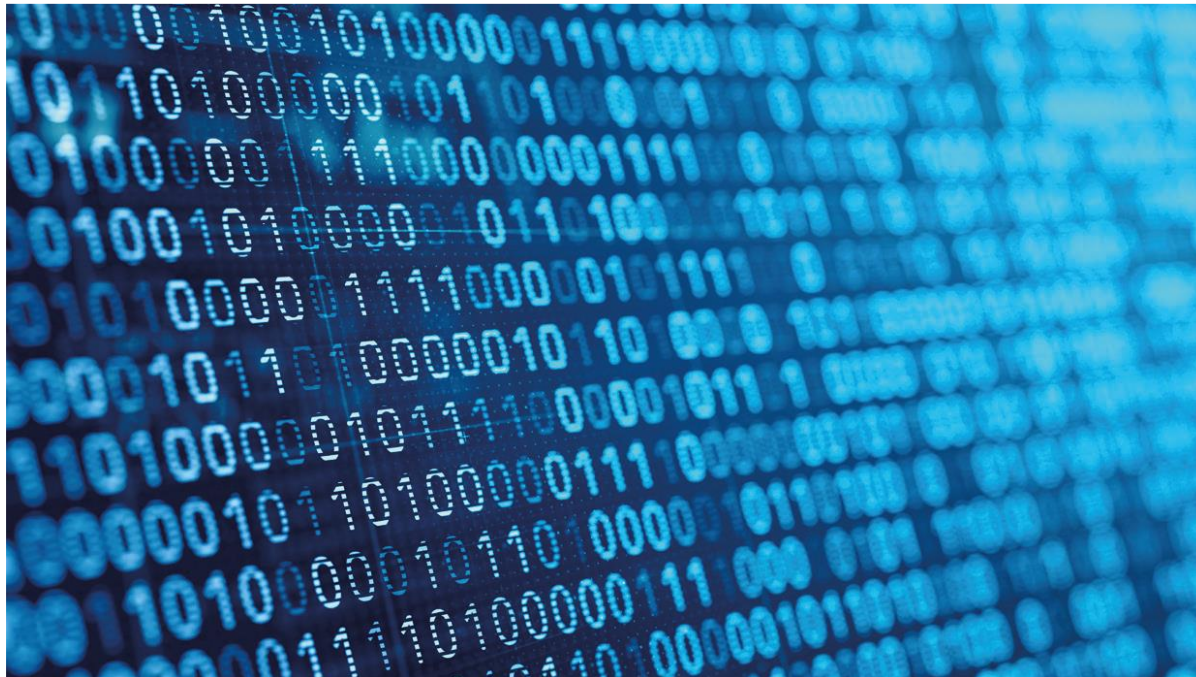


Figura 1.4. El código binario es comprensible solo para la plataforma para la que se compiló.

UNIDAD 1. Programación multiproceso

Ventaja

Al no tener que realizar proceso de compilación, el mismo código fuente se puede llevar a cualquier ordenador que tenga instalado el intérprete, independientemente del sistema operativo instalado.

Desventaja

Necesidad de disponer del intérprete instalado en el sistema de destino y que el proceso de «traducción» del programa a las instrucciones que entiende el sistema operativo supondrán una pequeña pérdida de velocidad, que en algunas ocasiones podrá no ser asumible.

UNIDAD 1. Programación multiproceso

El **sistema operativo** es el elemento del ordenador que coordina el funcionamiento del resto de componentes de este, tanto software como hardware. Es a él a quien se indica qué se quiere hacer o, siendo más precisos, qué programas se desean ejecutar.

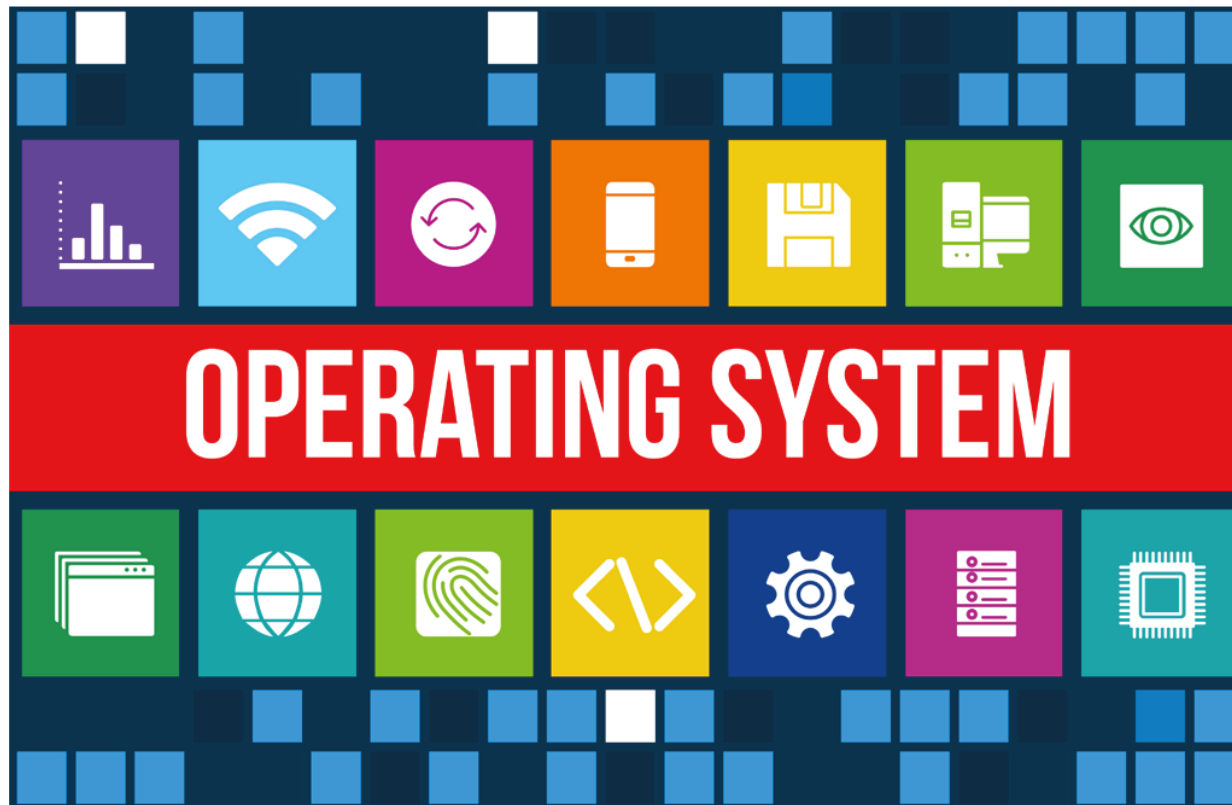
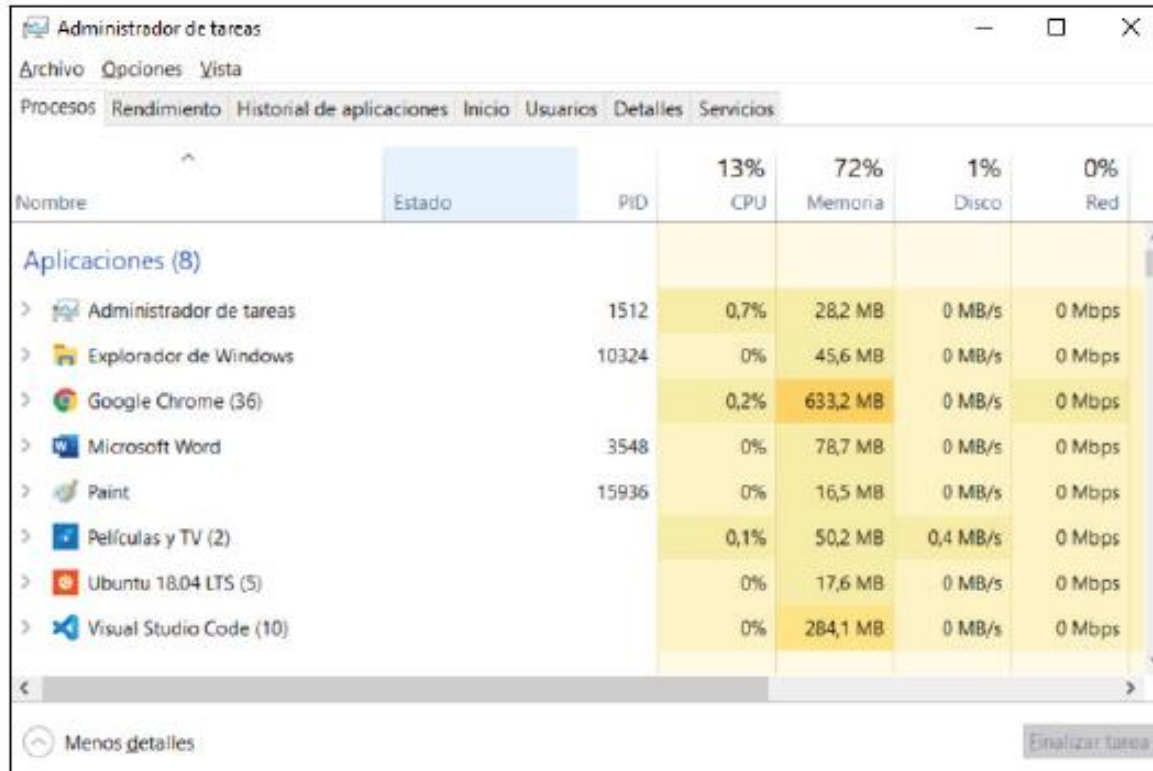


Figura 1.5. El sistema operativo coordina tanto el hardware como el software.

UNIDAD 1. Programación multiproceso



The screenshot shows the Windows Task Manager window with the 'Procesos' (Processes) tab selected. The window title is 'Administrador de tareas'. The menu bar includes 'Archivo', 'Opciones', and 'Vista'. The tabs at the top are 'Procesos', 'Rendimiento', 'Historial de aplicaciones', 'Inicio', 'Usuarios', 'Detalles', and 'Servicios'. The main area displays a table of running processes. The columns are: 'Nombre' (Name), 'Estado' (Status), 'PID', 'CPU' (13% total), 'Memoria' (72% total), 'Disco' (1% total), and 'Red' (0% total). The processes listed are: 'Administrador de tareas' (PID 1512), 'Explorador de Windows' (PID 10324), 'Google Chrome (36)' (PID 6332), 'Microsoft Word' (PID 3548), 'Paint' (PID 15936), 'Películas y TV (2)', 'Ubuntu 18.04 LTS (5)', and 'Visual Studio Code (10)'. The 'Microsoft Word' process is highlighted in yellow, indicating it is the selected process.

Nombre	Estado	PID	CPU	Memoria	Disco	Red
Aplicaciones (8)						
> Administrador de tareas		1512	0,7%	28,2 MB	0 MB/s	0 Mbps
> Explorador de Windows		10324	0%	45,6 MB	0 MB/s	0 Mbps
> Google Chrome (36)			0,2%	633,2 MB	0 MB/s	0 Mbps
> Microsoft Word		3548	0%	78,7 MB	0 MB/s	0 Mbps
> Paint		15936	0%	16,5 MB	0 MB/s	0 Mbps
> Películas y TV (2)			0,1%	50,2 MB	0,4 MB/s	0 Mbps
> Ubuntu 18.04 LTS (5)			0%	17,6 MB	0 MB/s	0 Mbps
> Visual Studio Code (10)			0%	284,1 MB	0 MB/s	0 Mbps

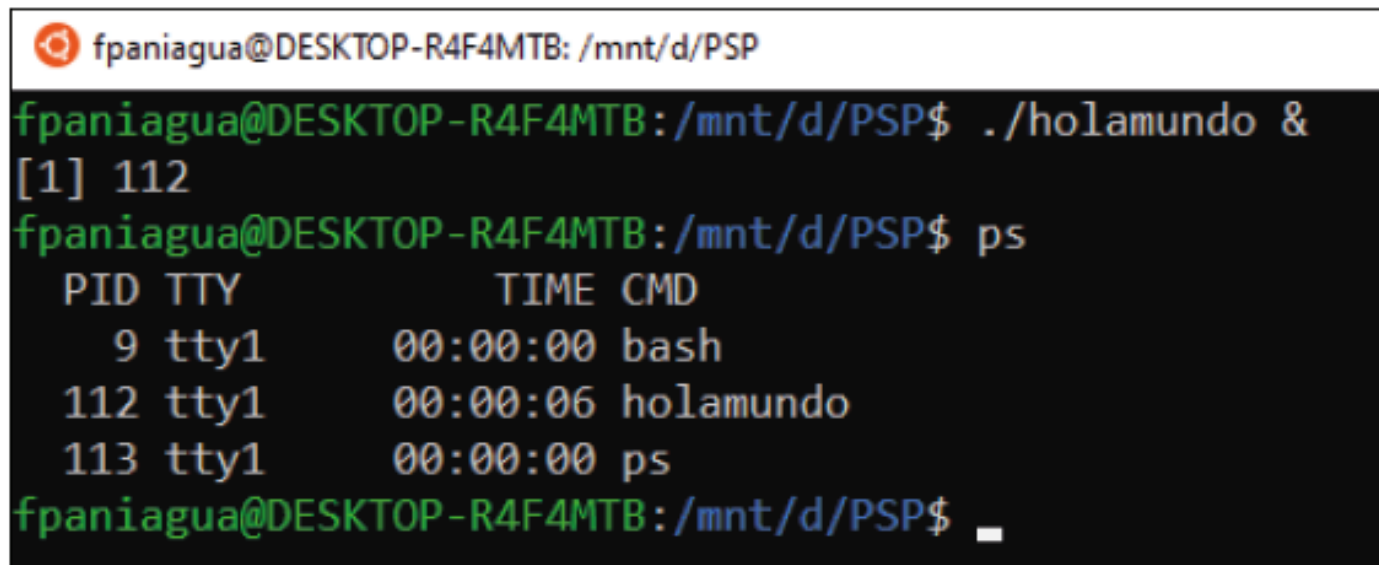
At the bottom left, there is a 'Menos detalles' button. At the bottom right, there is a 'Finalizar tareas' button.

En la **pestaña «Procesos»** se muestran las aplicaciones que están en ejecución, junto con su PID (identificador de proceso) y datos relacionados con el consumo de recursos. El proceso identificado con el PID 3548 se corresponde con una instancia del programa Microsoft Word.

Figura 1.6. El Administrador de tareas de Windows permite examinar los procesos en ejecución.

UNIDAD 1. Programación multiproceso

En la Figura 1.7 se muestra el listado de procesos de un sistema Linux. El proceso identificado con el PID 112 se corresponde con un programa escrito en C con el nombre «holamundo».

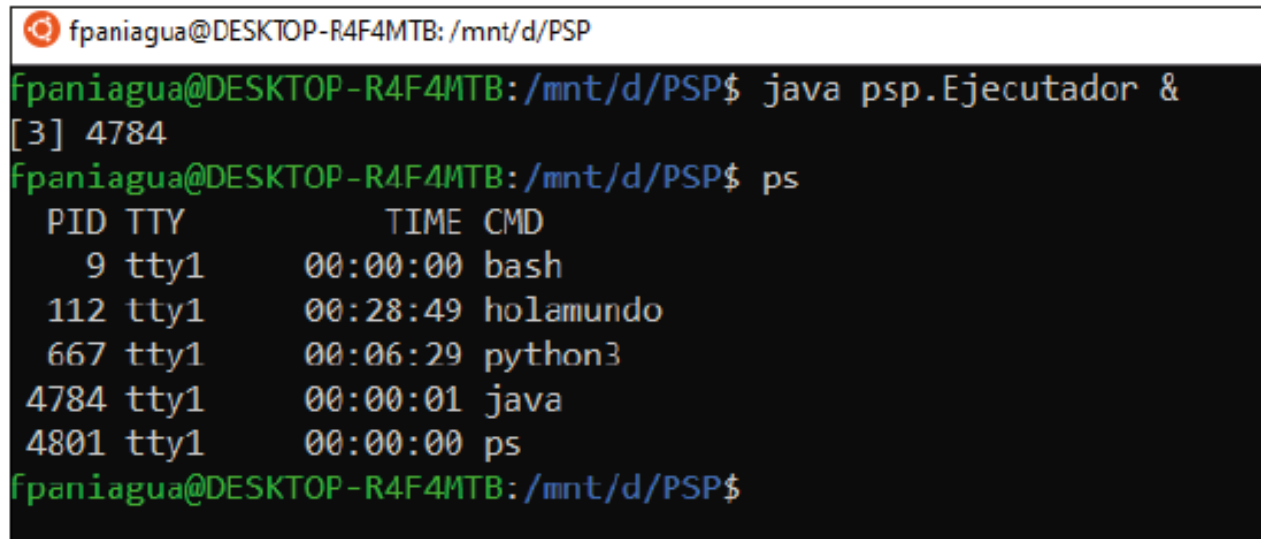
A terminal window with a black background and white text. The prompt is 'fpaniagua@DESKTOP-R4F4MTB: /mnt/d/PSP'. The user enters './holamundo &', which returns '[1] 112'. Then the user enters 'ps', which displays a table of processes.

```
fpaniagua@DESKTOP-R4F4MTB: /mnt/d/PSP$ ./holamundo &
[1] 112
fpaniagua@DESKTOP-R4F4MTB: /mnt/d/PSP$ ps
  PID TTY          TIME CMD
    9 tty1      00:00:00 bash
   112 tty1      00:00:06 holamundo
   113 tty1      00:00:00 ps
fpaniagua@DESKTOP-R4F4MTB: /mnt/d/PSP$
```

Figura 1.7. El comando ps de Linux muestra los procesos.

UNIDAD 1. Programación multiproceso

En la Figura 1.8 se puede observar que están conviviendo tres procesos correspondientes a tres programas de distintas naturalezas: un programa escrito en C con PID 112, otro escrito en Python con PID 667 y un último escrito en Java con PID 4784. En estos dos últimos no se observa el nombre del programa que se está ejecutando, sino el nombre del intérprete en el caso de Python y el de la máquina virtual en el caso de Java, ya que dentro de estos procesos es donde se ejecutan los programas correspondientes.



```
fpaniagua@DESKTOP-R4F4MTB: /mnt/d/PSP
fpaniagua@DESKTOP-R4F4MTB:/mnt/d/PSP$ java psp.Ejecutador &
[3] 4784
fpaniagua@DESKTOP-R4F4MTB:/mnt/d/PSP$ ps
  PID TTY          TIME CMD
    9 tty1        00:00:00 bash
   112 tty1        00:28:49 holamundo
   667 tty1        00:06:29 python3
  4784 tty1        00:00:01 java
  4801 tty1        00:00:00 ps
fpaniagua@DESKTOP-R4F4MTB:/mnt/d/PSP$
```

Figura 1.8. Los procesos tienen como nombre el del programa ejecutado.

Actividad propuesta 1.2

Destrucción de proceso de terminal

En Windows, abre la calculadora, y con PowerShell ciérrala

Pregunta existencial:

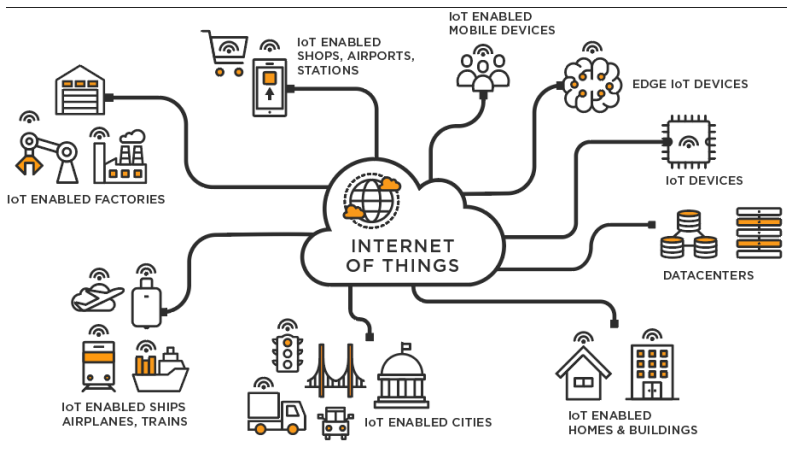
¿... y si sólo tenemos un núcleo?

Aunque para el usuario parezca que varios procesos se ejecutan al mismo tiempo, si solamente existe un único procesador, solamente un proceso puede estar en un momento determinado en ejecución. Para poder ir cambiando entre los diferentes procesos, el sistema operativo se encarga de cambiar el proceso en ejecución después de un período corto de tiempo (milisegundos).

La velocidad en la asignación de la CPU a los distintos procesos logra que no se perciba el cambio, pero, aunque este se apreciase, seguiría siendo multitarea, ya que todas las tareas avanzan sin tener que esperar unas a que las otras terminen. Este tipo de computación se denomina **conurrencia**.

Tenemos varios núcleos

Los sistemas basados en varios procesadores o en procesadores de varios núcleos aportan una mejora sustancial: permiten ejecutar varias instrucciones en un único ciclo de reloj. Esta capacidad hace posible ejecutar en paralelo varias instrucciones, lo que da origen al término ***procesamiento paralelo***.



Los sistemas basados en varios procesadores o en procesadores de varios núcleos aportan una mejora sustancial: permiten ejecutar varias instrucciones en un único ciclo de reloj. Esta capacidad hace posible ejecutar en paralelo varias instrucciones, lo que da origen al término ***procesamiento paralelo***.

Procesamiento Concurrente

Procesamiento Paralelo

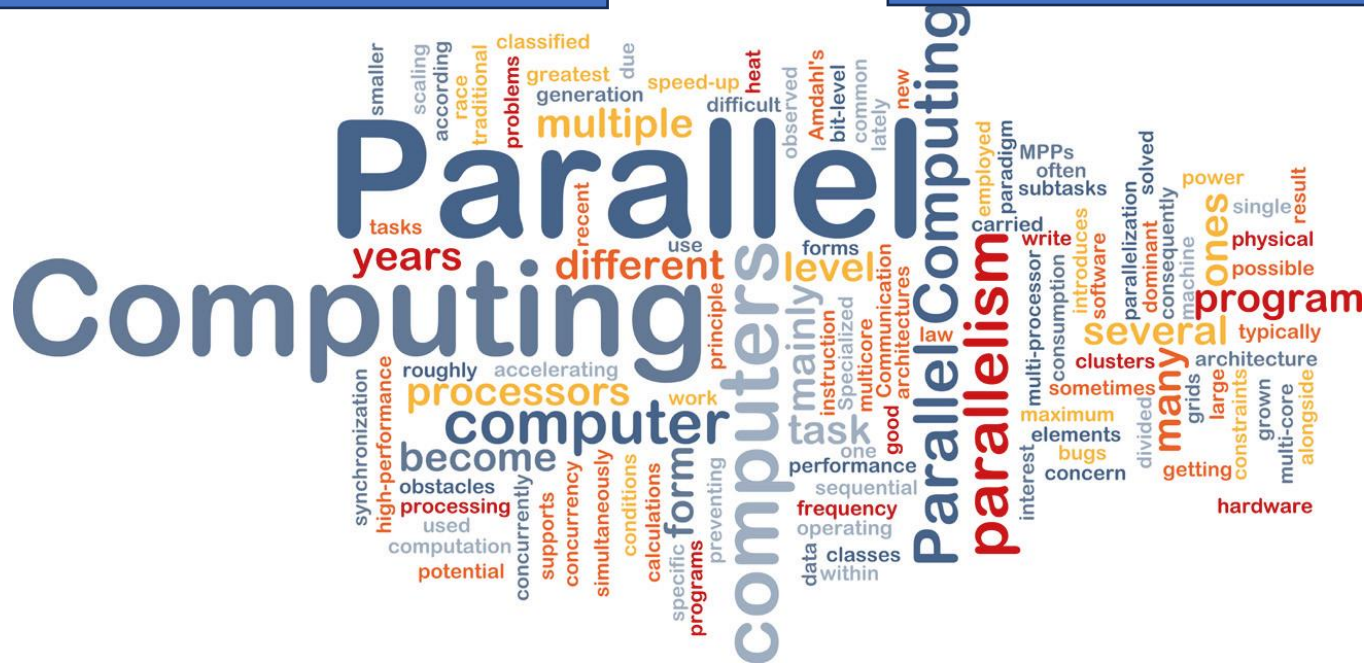
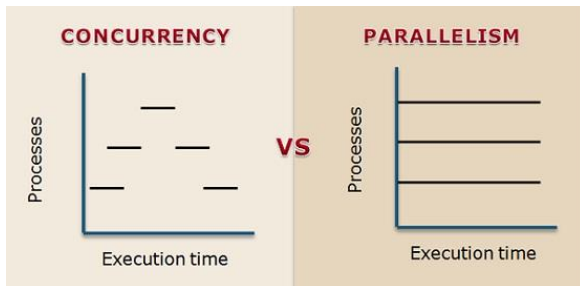


Figura 1.9. El paralelismo optimiza el uso de los recursos de computación.

Concurrencia vs Paralelismo

Concurrencia

Cuando el trabajo lo realiza una única CPU, podemos dividir el tiempo de procesamiento de esta CPU de forma que los diferentes procesos parezcan que se ejecutan de forma concurrente, cuando en realidad está haciendo una cosa cada vez. En este caso, la CPU está priorizando tiempos de ejecución de cada proceso o tarea, intercalando una fracción mínima de tiempo a cada proceso. La palabra clave en este caso, es la **coordinación** de los tiempos de ejecución de cada tarea.



Paralelismo

Los ordenadores de hoy en día tienen procesadores multi-core, por lo que disponen de varias CPUs (Unidades Centrales de Procesamiento), capaces de ejecutar procesos diferentes en paralelo y al mismo tiempo.

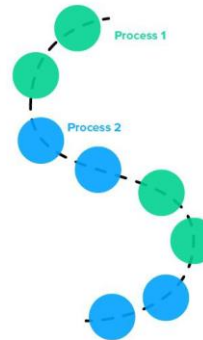
Diferencia entre concurrencia y paralelismo

La principal diferencia entre concurrencia y paralelismo es que **la concurrencia está sincronizando o coordinando** varias cosas a la vez, mientras que **el paralelismo está haciendo** varias cosas a la vez.

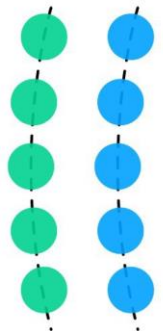
Resumen Concurrencia y paralelismo

- Como resumen:
- **Procesamiento concurrente.** Es aquel en el que varios procesos se ejecutan en una misma unidad de proceso de **manera alterna**, provocando el avance simultáneo de los mismos y evitando la secuencialidad.
- **Procesamiento paralelo.** Es aquel en el que divisiones de un proceso se ejecutan de **manera simultánea** en los diversos núcleos de ejecución de un procesador o en diversos procesadores.

Concurrency



Parallelism



vs

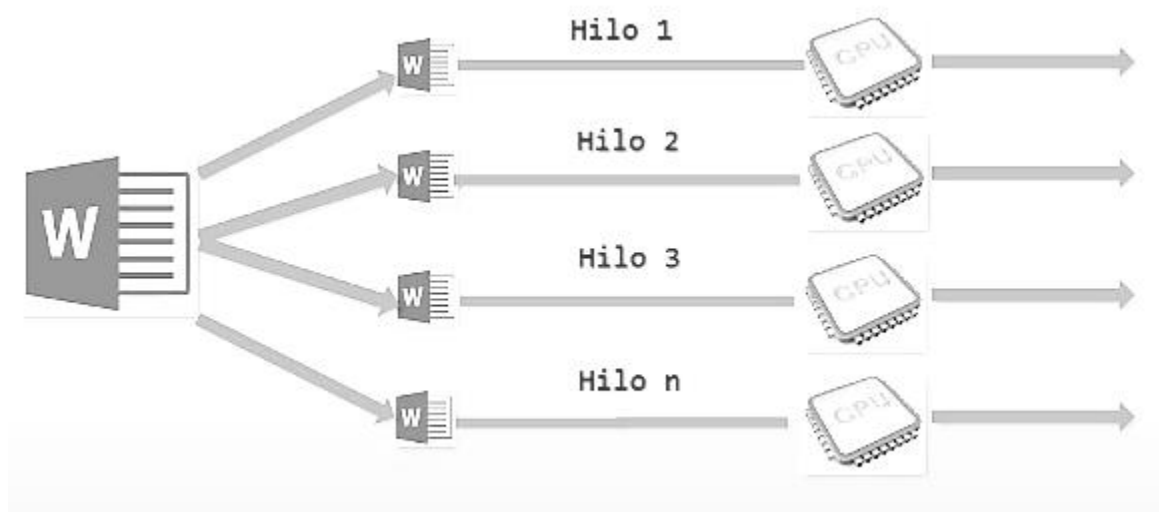
■ ■ 1.1.5. Programación distribuida

Se puede definir el **procesamiento distribuido** como aquel en el que un proceso se ejecuta en unidades de computación independientes conectadas y sincronizadas.

La programación distribuida es otro de los paradigmas de la programación multiproceso. En este tipo de arquitectura, la ejecución del software se distribuye entre varios ordenadores, consiguiendo así disponer de una potencia de procesamiento mucho más elevada, escalable y, normalmente, económica. Si en un único ordenador el número de núcleos viene determinado por el procesador que se está utilizando y es inmutable, en un sistema distribuido se elimina dicha restricción.

Para construir un sistema distribuido se requiere una red de ordenadores entre los que distribuir el trabajo. No todas las tareas son susceptibles de distribuirse ni en todos los casos se obtendrá beneficio respecto de una ejecución convencional, pero en el caso en el que la ventaja es posible, estos tipos de sistemas son altamente eficientes y no requieren de una inversión tan elevada como la resultante de conseguir la misma potencia de cálculo con un único ordenador.

Los hilos de ejecución son fracciones de programa que, si cumplen con determinadas características, pueden ejecutarse simultáneamente gracias al procesamiento paralelo. Al formar parte del mismo proceso son extremadamente económicos en referencia a los recursos que utilizan.



Los programas que se ejecutan en un único hilo se denominan *programas monohilo*, mientras que los que se ejecutan en varios hilos se conocen como *programas multihilo*.

1.1.7. Bifurcación o *fork*

Una bifurcación, referenciada habitualmente por su término en inglés **fork**, es una copia idéntica de un proceso. El proceso original se denomina padre y sus copias, hijos, teniendo todos ellos diferentes identificadores de proceso (PID). La copia creada continua con el estado del proceso original (padre), pero a partir de la creación cada proceso mantiene su propio estado de memoria.



Actividad propuesta 1.2.1

En el siguiente código se realiza la creación de un *fork* de un proceso escrito en lenguaje C y ejecutándose en un sistema Linux.

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <sys/types.h>
4.  int main(void) {
5.      int contador=0;
6.      printf("Comenzando la ejecución\n");
7.      pid_t idHijo;
8.      pid_t idPadre;
9.      idPadre = getpid();
10.     printf("Antes de bifurcar\n");
11.     contador++;
12.     idHijo = fork();
13.     contador++;
14.     printf("Después de bifurcar\n");
15.     if (idHijo == 0) {
16.         printf("Id. hijo:%ld Id. padre:%ld Contador:%d \n ", (long)getpid(),
17.             (long)idPadre, contador);
18.     } else {
19.         printf("Id. padre:%ld Id. hijo:%ld Contador:%d \n", (long)getpid(),
20.             (long)idPadre, contador);
21.     }
22.     return 0;
23. }
```

Realizar ejercicios Fork en el aula

Intenta compilar estos ejemplos tanto para Windows como para Linux, y explica los resultados obtenidos.



`crearHijo.c`



`holaMundo.c`



`primerFork.c`

■ ■ 1.2.1. Gestión y estados de los procesos

■ ■ 1.2.2. Comunicación entre procesos

■ ■ 1.2.3. Sincronización entre procesos

¿qué es un proceso?

En definitiva, puede definirse “proceso” como *un programa en ejecución*. Este concepto no se refiere únicamente al código y a los datos, sino que incluye todo lo necesario para su ejecución.

Pueden coexistir dos procesos que ejecuten el mismo programa, pero con diferentes datos (distintas direcciones de memoria) y en distintos momentos de su ejecución (con diferentes contadores de programa). Por ejemplo, podemos tener dos instancias de Word ejecutándose a la vez, modificando cada una un fichero diferente. Para que los datos de uno no interfieran con los del otro, cada proceso se ejecuta en su propio espacio de direcciones en memoria permitiendo independencia entre los procesos.

Cada proceso está compuesto por:

- Las instrucciones que se van a ejecutar.
- El estado del propio proceso.
- El estado de la ejecución, principalmente recogido en los registros del procesador.
- El estado de la memoria.

• Contexto

Los procesos están constantemente entrando y saliendo del procesador. Se denomina **contexto** a toda la información que determina el estado de un proceso en un instante dado. Es una especie de fotografía que permite quitar al proceso del procesador y restaurarlo en otro momento en el mismo estado en el que se encontraba.

• Cambio de Contexto

Sacar a un proceso del procesador para meter a otro se conoce como **cambio de contexto**. Un cambio de contexto implica capturar el estado de la CPU y de sus registros, de la memoria y de la propia ejecución del proceso saliente para restaurar la información equivalente del proceso entrante y poder continuar en el punto en el que este último abandonó el procesador en el cambio de contexto anterior.

Los pasos que se han de realizar para llevar a cabo un cambio de contexto se pueden resumir en los siguientes:

- Guardar el estado del proceso actual.
- Determinar el siguiente proceso que se va a ejecutar.
- Recuperar y restaurar el estado del siguiente proceso.
- Continuar con la ejecución del siguiente proceso.

¿Cómo se consigue que la convivencia entre los procesos, que compiten entre sí por los limitados recursos del sistema de computación, sea posible?

La respuesta está en el sistema operativo y, más concretamente, en el planificador de procesos.



Figura 1.10. El sistema operativo se encarga de gestionar la convivencia de los procesos.

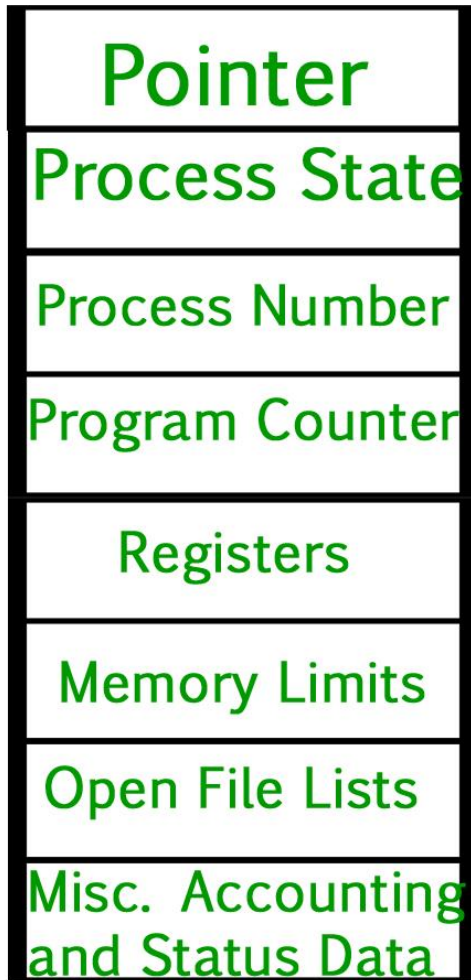
■ ■ 1.2.1. Gestión y estados de los procesos

El planificador de procesos

El planificador de procesos es el elemento del sistema operativo que se encarga de repartir los recursos del sistema entre los procesos que los demandan. De hecho, es uno de sus componentes fundamentales, ya que determina la calidad del multiproceso del sistema y, como consecuencia, la eficiencia en el aprovechamiento de los recursos.

Los objetivos del planificador son los siguientes:

- Maximizar el rendimiento del sistema.
- Maximizar la equidad en el reparto de los recursos.
- Minimizar los tiempos de espera.
- Minimizar los tiempos de respuesta.



Process Control Block

Procesos y BCP

- Todos los programas que se ejecutan en el ordenador se organizan como un conjunto de procesos. El SO se encarga de parar/reanudar la ejecución de los mismos.
- A cada proceso que se ejecuta le pertenece el ***Bloque de Control de Proceso (BCP)***, donde almacena, en memoria:
 - El ID del proceso, único.
 - El estado del proceso.
 - El contador del programa (la instrucción que se está ejecutando)
 - La prioridad del proceso

Algoritmos de planificación

¿...cómo gestiona todo esto el SO?

Todos los SO multitarea tienen un componente software conocido como **Planificador**, el cual se encarga de repartir el tiempo de CPU entre los procesos en espera.

La Planificación de procesos tiene como principales objetivos:

- **Equidad**: Todos los procesos deben ser atendidos.
- **Eficacia**: El procesador debe estar ocupado el 100% del tiempo.
- **Tiempo de respuesta**: El tiempo empleado en dar respuesta a las solicitudes del usuario debe ser el menor posible.
- **Tiempo de regreso**: Reducir al mínimo el tiempo de espera de los resultados esperados por los usuarios por lotes.
- **Rendimiento**: Maximizar el número de tareas que se procesan por cada hora.

Algoritmos de planificación

La planificación no apropiativa (en inglés, no preemptive) es aquella en la que, cuando a un proceso le toca su turno de ejecución, ya no puede ser suspendido; es decir, no se le puede arrebatar el uso de la CPU, ***hasta que el proceso no lo determina no se podrá ejecutar otro proceso.***

Este esquema tiene sus problemas, puesto que, si el proceso contiene ciclos infinitos, el resto de los procesos pueden quedar aplazados indefinidamente. Otro caso puede ser el de los procesos largos que penalizarían a los cortos si entran en primer lugar. Los más conocidos son el algoritmo **FIFO y SJF**.

La planificación apropiativa (en inglés, preemptive) supone que el sistema operativo puede arrebatar el uso de la CPU a un proceso que esté ejecutándose.

En la planificación apropiativa existe un reloj que lanza interrupciones periódicas en las cuales el planificador toma el control y se decide si el mismo proceso seguirá ejecutándose o se le da su turno a otro proceso.

En ambos enfoques de planificación se pueden establecer distintos algoritmos de planificación de ejecución de procesos.

Los más conocidos son el algoritmo **SRT y Round Robin**.

Métricas

¿En qué nos basamos para elegir un algoritmo u otro?

Hay varias medidas que se utilizan para evaluarlos.

- **Rendimiento** (throughput) = número de ráfagas por unidad de tiempo. Se define una ráfaga como el período de tiempo en que un proceso necesita la CPU; un proceso, durante su vida, alterna ráfagas con bloqueos. Por extensión, también se define como el no de trabajos por unidad de tiempo.
- **Tiempo de espera** (E) = tiempo que un proceso ha permanecido en estado preparado. Es decir, es el tiempo que pasa desde que un proceso llega a la cola de procesos, hasta que se empieza a ejecutar.
- **Tiempo de finalización** (F) = tiempo transcurrido desde que un proceso comienza a existir hasta que finaliza. $F = E + t$ (t = tiempo de CPU de la ráfaga).

FIFO (First IN First OUT)

Este algoritmo es muy sencillo y simple, pero también el que menos rendimiento ofrece, básicamente en este algoritmo *el primer proceso que llega se ejecuta y una vez terminado se ejecuta el siguiente*.

Cuando se tiene que elegir a qué proceso asignar la CPU se escoge al que llevara más tiempo listo.

IN

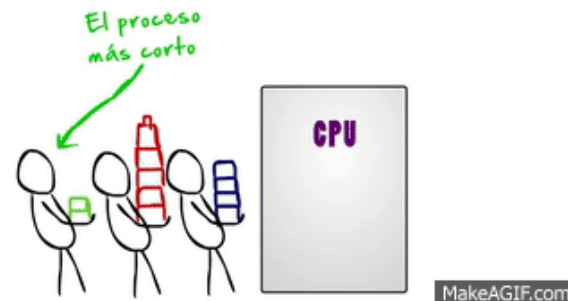


OUT

First Come First Served (FCFS)

SJF (Shortest Job First)

Este algoritmo siempre prioriza los procesos más cortos primero independientemente de su llegada y en caso de que los procesos sean iguales utilizara el método FIFO anterior, es decir, el orden según entrada. Este sistema tiene el riesgo de poner siempre al final de la cola los procesos más largos por lo que nunca se ejecutarán, esto se conoce como inanición (starving).



SRT Shortest - Remaining –Time

(El tiempo restante más corto)

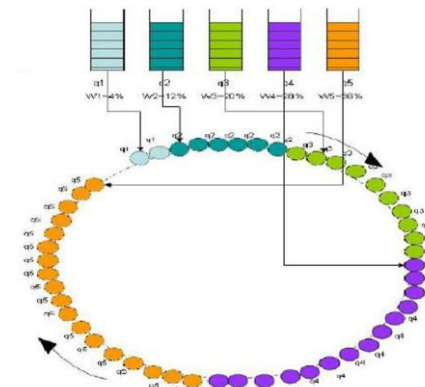
Es similar al anterior (SJF), con la diferencia de que si un nuevo proceso pasa a listo se activa el planificador para ver si es más corto que lo que queda por ejecutar del proceso en ejecución.

Si es así el proceso en ejecución pasa a listo y su tiempo de estimación se decrementa con el tiempo que ha estado ejecutándose.

Es decir, actúa igual que el SJF, pero de forma APROPIATIVA

RR (Round-robin)

Asigna a cada proceso un tiempo equitativo tratando a todos los procesos por igual y con la misma prioridad. Este algoritmo es circular, volviendo siempre al primer proceso una vez terminado con el último, para controlar este método a cada proceso se le asigna un intervalo de tiempo llamado quantum.



- **Nuevo.** Técnicamente no es un estado, sino el reflejo del instante en el que se crea el proceso.
- **Listo.** El proceso está en memoria, preparado para ejecutarse. Está a la espera de que el planificador le conceda tiempo de procesamiento.
- **En ejecución.** El proceso se encuentra en ejecución.
- **Bloqueado.** Se encuentra a la espera de que ocurra un evento externo ajeno al planificador.
- **Finalizado.** Al igual que el «estado» *Nuevo*, no es técnicamente un estado. Refleja el instante posterior a la finalización del proceso.

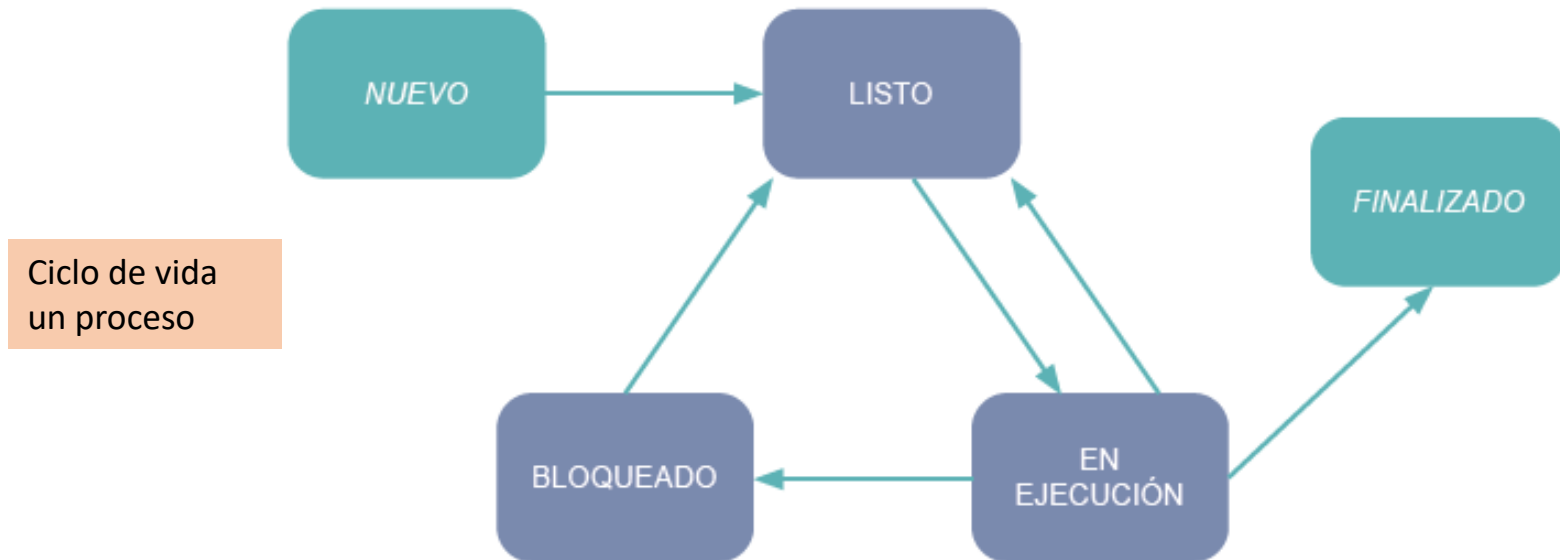


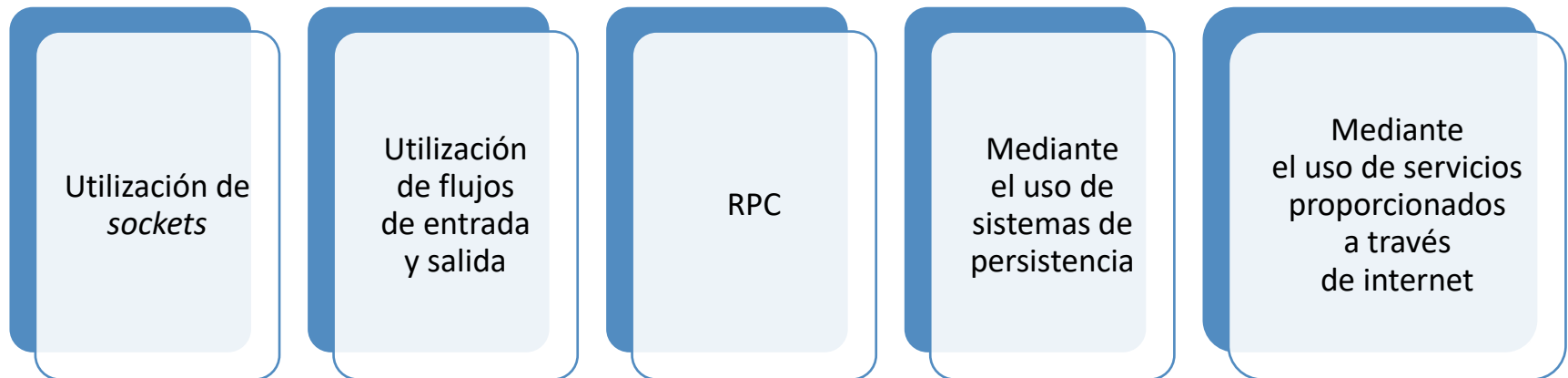
Figura 1.11. Los procesos tienen un ciclo de vida controlador por el planificador.

■ ■ 1.2.2. Comunicación entre procesos

UNIDAD 1. Programación multiproceso

Por definición, los procesos de un sistema son elementos estancos. Cada uno tiene su espacio de memoria, su tiempo de CPU asignado por el planificador y su estado de los registros. No obstante, los procesos deben poder comunicarse entre sí, ya que es natural que surjan dependencias entre ellos en lo referente a las entradas y salidas de datos.

La comunicación entre procesos se denomina IPC (*Inter-Process Communication*) y existen diversas alternativas para llevarla a cabo.



1.2.2. Comunicación entre procesos

- **Utilización de sockets.** Los sockets son mecanismos de comunicación de bajo nivel. Permiten establecer canales de comunicación de *bytes* bidireccionales entre procesos alojados en distintas máquinas y programados con diferentes lenguajes. Gracias a los sockets dos procesos pueden intercambiar cualquier tipo de información.
- **Utilización de flujos de entrada y salida.** Los procesos pueden interceptar los flujos de entrada y salida estándar, por lo que pueden leer y escribir información unos en otros. En este caso, los procesos deben estar relacionados previamente (uno de ellos debe haber arrancado al otro obteniendo una referencia al mismo).
- **RPC.** Llamada a procedimiento remoto (*Remote Process Call*, en inglés). Consiste en realizar llamadas a métodos de otros procesos que, potencialmente, pueden estar ejecutándose en otras máquinas. Desde el punto de vista del proceso que realiza la llamada, la ubicación de los procesos llamados es transparente.

- **Mediante el uso de sistemas de persistencia.** Consiste en realizar escrituras y lecturas desde los distintos procesos en cualquier tipo de sistema de persistencia, como los ficheros o las bases de datos. Pese a su sencillez, no se puede ignorar esta alternativa, ya que puede ser suficiente en múltiples ocasiones.
- **Mediante el uso de servicios proporcionados a través de internet.** Los procesos pueden utilizar servicios de transferencia de ficheros FTP, aplicaciones o servicios web, así como la tecnología *cloud* como mecanismos de conexión entre procesos que permiten el intercambio de información.

■ ■ 1.2.3. Sincronización entre procesos

Todos los sistemas en los que participan múltiples actores de manera concurrente están sometidos a ciertas condiciones que exigen que exista sincronización entre ellos. Por ejemplo, puede que sea necesario saber si un proceso ha terminado satisfactoriamente para ejecutar el siguiente que se encuentra en un flujo de procesos o, en caso de que haya ocurrido un determinado error, ejecutar otro proceso alternativo.

Es el planificador del sistema operativo el encargado de decidir en qué momento tiene acceso a los recursos un proceso, pero a nivel general, la decisión de crear y lanzar un proceso es humana y expresada a través de un algoritmo.

En la Figura 1.12 se muestra un posible ejemplo de flujo de ejecución de un conjunto de procesos. Las condiciones que determinan dicho flujo son las siguientes:

- El proceso «Proceso 1» se ejecuta inicialmente.
- Si el código de finalización de «Proceso 1» es 0, se ejecuta el proceso «Proceso 1.1».
- ✓ Si el código de finalización de «Proceso 1.1» es 0, se ejecuta el proceso «Proceso 1.1.1».
- ✓ Si el código de finalización de «Proceso 1.1» es 1, se ejecuta el proceso «Proceso 1.1.2».
- Si el código de finalización de «Proceso 1» es 1, se ejecuta el proceso «Proceso 1.2».
- ✓ Independientemente del código de finalización del proceso «Proceso 1.2», pero únicamente cuando haya finalizado, se ejecuta el proceso «Proceso 1.2.1».

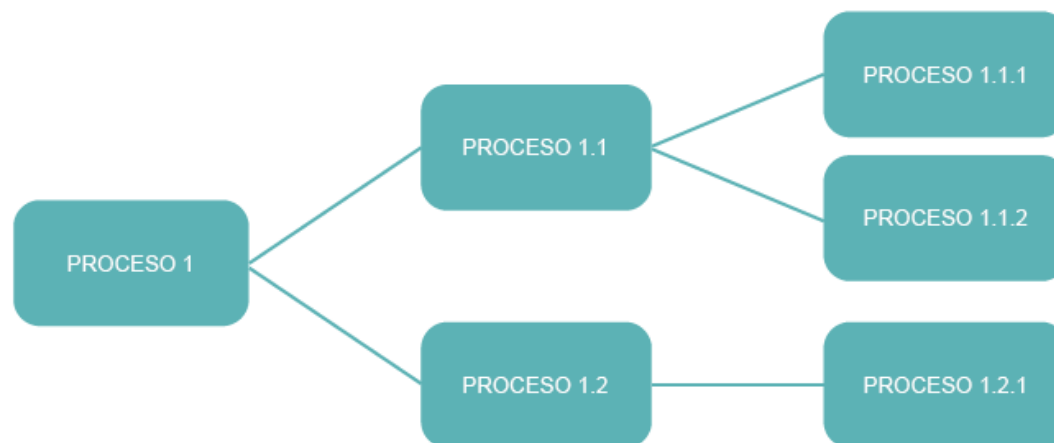
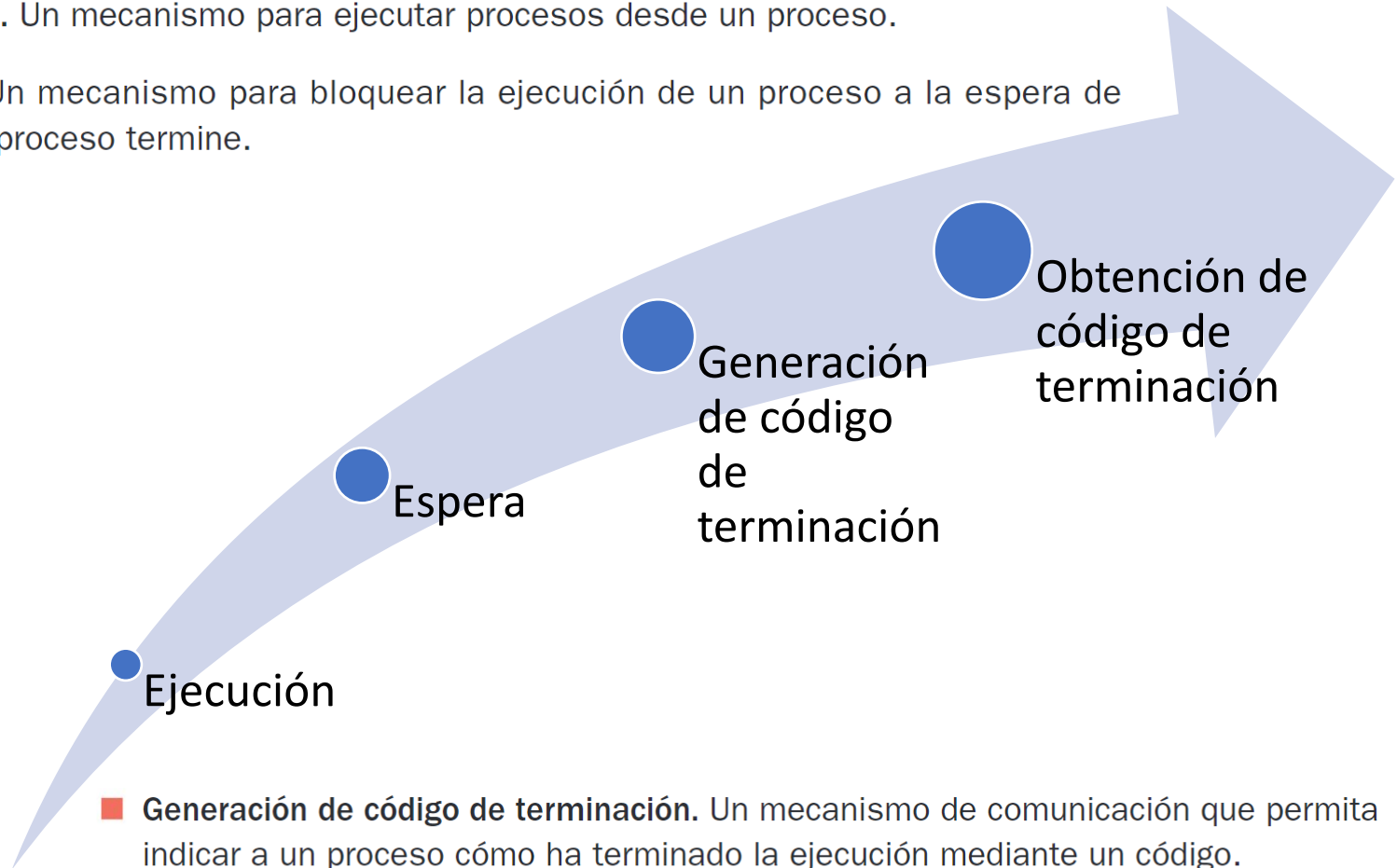


Figura 1.12. El flujo de ejecución de un sistema de procesos puede depender del resultado de las ejecuciones de sus integrantes.

UNIDAD 1. Programación multiproceso

Para gestionar un flujo de trabajo como el presentado en el ejemplo se necesita disponer de los siguientes **mecanismos**:

- **Ejecución.** Un mecanismo para ejecutar procesos desde un proceso.
- **Espera.** Un mecanismo para bloquear la ejecución de un proceso a la espera de que otro proceso termine.



- **Generación de código de terminación.** Un mecanismo de comunicación que permita indicar a un proceso cómo ha terminado la ejecución mediante un código.
- **Obtención de código de terminación.** Un mecanismo que permita a un proceso obtener el código de terminación de otro proceso.

Tabla 1.1. Clases y métodos de Java que proporcionan soporte a la gestión de procesos

Mecanismo	Clase	Método
Ejecución	Runtime	exec()
Ejecución	ProcessBuilder	start()
Espera	Process	waitFor()
Generación de código de terminación	System	exit(valor_del_retorno)
Obtención de código de terminación	Process	waitFor()

■ 1.3. Programación de aplicaciones multiproceso en Java

Cada instancia de una aplicación en ejecución es un proceso. Cada proceso dispone de un conjunto de instrucciones, un estado de los registros del procesador, un espacio de memoria y un estado en lo referente a la gestión que hace de él el planificador del sistema operativo. ¿Tiene sentido, por lo tanto, hablar de programación de aplicaciones multiprocesos si una aplicación cuando se ejecuta constituye uno único?

**Creación de procesos con
*Runtime***

**Creación de procesos con
*ProcessBuilder***



UNIDAD 1. Programación multiproceso

Las necesidades que se deben satisfacer para poder programar un sistema basado en la ejecución de múltiples procesos son las siguientes:



Poder arrancar un proceso y hacerle llegar los parámetros de ejecución.



Poder quedar a la espera de que el proceso termine.



Poder recoger el código de finalización de ejecución para determinar si el proceso se ha ejecutado correctamente o no.



Poder leer los datos generados por el proceso para su tratamiento.

En Java, la creación de un proceso se puede realizar de dos maneras diferentes:

- Utilizando la clase `java.lang.Runtime`.
- Utilizando la clase `java.lang.ProcessBuilder`.

Toda aplicación Java tiene una única instancia de la clase *Runtime* que permite que la propia aplicación interactúe con su entorno de ejecución a través del método estático *getRuntime*. Este método proporciona un «canal» de comunicación entre la aplicación y su entorno, posibilitando la interacción con el sistema operativo a través del método *exec*.

El siguiente código Java genera un proceso en Windows indicando al entorno de ejecución (al sistema operativo) que ejecute el bloc de notas a través del programa «Notepad.exe». En este caso, la llamada se realiza sin parámetros y sin gestionar de ninguna manera el proceso generado.

```
Runtime.getRuntime().exec("Notepad.exe");
```

```
Runtime.getRuntime().exec("Notepad.exe notas.txt");
```

UNIDAD 1. Programación multiproceso

Se puede crear un proceso con un array de objetos del tipo String
Agregando algunos parámetros a la ejecución del programa

```
String[] infoProceso = { "Notepad.exe", "notas.txt" };  
Runtime.getRuntime().exec(infoProceso);
```

Gestionar el proceso lanzado:

waitFor(): Pone en pausa la ejecución del programa ejecutor (quien lanza el proceso), y queda a la espera de que termine el proceso lanzado.

```
String[] infoProceso = { "Notepad.exe", "notas.txt" };  
Process proceso = Runtime.getRuntime().exec(infoProceso);  
int codigoRetorno = proceso.waitFor();  
System.out.println("Fin de la ejecución:" + codigoRetorno);
```

La clase ***Process*** representa al proceso en ejecución y permite obtener información sobre este.

1.3.1. Creación de procesos con *Runtime*

Método	Descripción
<code>destroy()</code>	Destruye el proceso sobre el que se ejecuta.
<code>exitValue()</code>	Devuelve el valor de retorno del proceso cuando este finaliza. Sirve para controlar el estado de la ejecución.
<code>getErrorStream()</code>	Proporciona un <code>InputStream</code> conectado a la salida de error del proceso.
<code>getInputStream()</code>	Proporciona un <code>InputStream</code> conectado a la salida normal del proceso.
<code>getOutputStream()</code>	Proporciona un <code>OutputStream</code> conectado a la entrada normal del proceso.
<code>isAlive()</code>	Determina si el proceso está o no en ejecución.
<code>waitFor()</code>	Detiene la ejecución del programa que lanza el proceso a la espera de que este último termine.

Tabla 1.2. Métodos de la clase `java.lang.Process`

Realizar

Actividad resuelta 1.1



1.3.2. Creación de procesos con *ProcessBuilder*

UNIDAD 1. Programación multiproceso

La clase *ProcessBuilder* permite, al igual que *Runtime*, crear procesos.

La creación más sencilla de un proceso se realiza con un único parámetro en el que se indica el programa a ejecutar. Es importante saber que esta construcción no supone la ejecución del proceso.

```
new ProcessBuilder("Notepad.exe")
```

La ejecución del proceso se realiza a partir de la invocación al método *start*:

```
new ProcessBuilder("Notepad.exe").start();
```

El constructor de *ProcessBuilder* admite parámetros que serán entregados al proceso que se crea.

```
new ProcessBuilder("Notepad.exe", "datos.txt").start();
```

UNIDAD 1. Programación multiproceso

Al igual que ocurre con el método `exec` de la clase *Runtime*, el método `start` de *ProcessBuilder* proporciona un proceso como retorno, lo que posibilita la sincronización y gestión de este.

```
1 Process proceso = new
2     ProcessBuilder("Notepad.exe","datos.txt").start();
3 int valorRetorno = proceso.waitFor();
4 System.out.println("Valor retorno:" + valorRetorno);
```

El método `start` permite crear múltiples subprocesos a partir de una única instancia de *ProcessBuilder*. El siguiente código crea diez instancias del bloc de notas de Windows.

```
1 ProcessBuilder pBuilder = new ProcessBuilder("Notepad.exe");
2 for (int i=0;i<10;i++) {
3     pBuilder.start();
4 }
```

Realizar esta actividad

Tabla 1.3. Métodos de la clase `java.lang.ProcessBuilder`

Método	Descripción
<code>start</code>	Inicia un nuevo proceso usando los atributos especificados.
<code>command</code>	Permite obtener o asignar el programa y los argumentos de la instancia de <code>ProcessBuilder</code> .
<code>directory</code>	Permite obtener o asignar el directorio de trabajo del proceso.
<code>environment</code>	Proporciona información sobre el entorno de ejecución del proceso.
<code>redirectError</code>	Permite determinar el destino de la salida de errores.
<code>redirectInput</code>	Permite determinar el origen de la entrada estándar.
<code>redirectOutput</code>	Permite determinar el destino de la salida estándar.

Realizar: **Actividad resuelta 1.2**

Actividad de aplicación 1.22 (navegadores)

UNIDAD 1. Programación multiproceso

Realizar test

Fin Unidad 1