

# Programación multiproceso

## Objetivos

- Descubrir los sistemas multitarea y sus múltiples variantes.
- Enumerar los elementos participantes en los sistemas multitarea.
- Conocer el funcionamiento básico del sistema operativo en lo referente a la gestión de las tareas.
- Profundizar en el conocimiento de los procesos, incluidos sus mecanismos de sincronización y de comunicación.
- Aprender a realizar aplicaciones multitarea en varias de sus alternativas.

## 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.

Son muchos los términos, las posibilidades, las dificultades y sus correspondientes soluciones que orbitan alrededor de toda la tecnología relacionada con el procesamiento y la programación multiproceso. En esta unidad se presentan las bases de la programación multiproceso para comprender el funcionamiento de esta área de la computación.

### 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. Por ejemplo, puede reproducir un fichero de sonido, imprimir un documento, descargar un programa de internet, recibir un correo electrónico, actualizar el sistema operativo y monitorizar la temperatura de la CPU. Este hecho sugiere una pregunta, ¿cómo puede un ordenador ejecutar a la vez más tareas que el número de unidades de proceso que tiene disponibles? La respuesta es gracias a la **multitarea**, que es la capacidad de ejecutar varias tareas simultáneamente por parte de un computador.

Los primeros ordenadores domésticos eran muy limitados y algunos sistemas operativos reflejaban de forma evidente esa limitación: o se estaba editando un documento de texto, o ejecutando un programa que realizaba un cálculo o jugando a un videojuego. Solo permitían hacer una tarea en un momento determinado. Estos sistemas se denominan **monotarea** y el sistema operativo MS-DOS, publicado en 1981, sería uno de sus principales ejemplos. No obstante, esa fecha no debe ser una referencia del paso de la monotarea a la multitarea, ya que sistemas como UNIX, desarrollado varios años antes, ya contemplaban la multitarea.



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

La **multitarea** es, por lo tanto, la capacidad que tienen los ordenadores de realizar varias tareas (ejecutar varios programas) al mismo tiempo, independientemente del número de elementos de procesamiento. Esta multitarea puede ser real (hay tantas unidades de proceso como procesos a ejecutar) o simulada (hay menos unidades de proceso que procesos a ejecutar).

La capacidad de realizar multitarea de un sistema depende principalmente del procesador y del sistema operativo. El primero tiene la capacidad física y el segundo, la capacidad lógica. Curiosamente, unas determinadas características del sistema operativo son necesarias para que exista multitarea mientras que las necesidades del procesador son menos restrictivas: un procesador simple con un sistema operativo adecuado puede disponer de multitarea; un procesador con múltiples núcleos con un sistema operativo inadecuado no aprovechará los recursos disponibles.

Alrededor de la multitarea existe una serie de conceptos que, en ocasiones, provocan confusión, ya que la frontera que los delimita es difusa. Concurrencia, paralelismo, proceso o hilo son algunos de los términos cuyo significado es fundamental explicar para comprender esta interesante disciplina de la computación.

### ■■■ 1.1.1. El procesador

Uno de los elementos más importante de un ordenador (o de un teléfono móvil, o de una videoconsola...) es el procesador. Este componente es el que proporciona la capacidad de ejecutar las instrucciones de los programas. Es «el cerebro» del dispositivo.

Existen muchos modelos de procesador, tantos como necesidades de capacidad, tamaño y coste. Desde el legendario Zilog Z80A de los ordenadores Sinclair ZX Spectrum de los años ochenta del siglo xx hasta los modernos Intel Core i9, por citar solo dos

ejemplos representativos, ha habido un salto en potencia, gestión energética y reducción de tamaño inimaginable hace años. También es llamativo el incremento en el número de núcleos y su impacto en la multitarea.

Un **núcleo** es una unidad con capacidad de ejecución dentro de un procesador. En un sistema con un procesador y cuatro núcleos se pueden ejecutar cuatro instrucciones simultáneamente. Esto no significa que se disponga de cuatro procesadores, pero sí que se tiene mucha más capacidad de procesamiento simultáneo.

Obviamente, no todos los procesadores son iguales. Sin ir a los extremos más distantes, se pueden comparar los procesadores Intel i486 SX de 33 MHz y un solo núcleo con los Intel Core i7 de más de 2 GHz en sus modelos más sencillos y cuatro núcleos. Los resultados de dicha comparación evidencian las diferencias entre distintos procesadores.

## 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.

¿Significa todo esto que un procesador con un único núcleo no puede realizar multitarea? La respuesta es no. De hecho, el procesamiento multitarea existe en sistemas operativos que se ejecutan sobre arquitecturas que disponen de un solo procesador con un único núcleo.



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

Cualquier procesador puede, potencialmente, ejecutar varias tareas y que parezca que las realiza todas simultáneamente. El «truco» consiste en dedicar una pequeña porción de tiempo de ejecución a cada una, de tal manera que no se aprecie el cambio entre tarea (denominado cambio de contexto). Si el ordenador es suficientemente rápido se

producirá una «ilusión» de ejecución simultánea. La realidad es que todas las tareas avanzan sin tener que esperar unas a que las otras terminen, pero en un instante dado solo alguna de ellas está realmente en ejecución.

Un procesador con varios núcleos, o un sistema basado en varios procesadores, podrá realizar una ejecución simultánea real, pero nunca será capaz de ejecutar más operaciones concurrentes que el número de unidades de procesamiento disponible.

## ■■■ 1.1.2. El sistema operativo y los lenguajes de programación

Un ordenador, para funcionar y ser útil, además del propio hardware necesita disponer de un sistema operativo y de programas. Se puede crear una analogía entre un sistema computacional y una orquesta: el hardware es el auditorio y los instrumentos; el sistema operativo, el director y los programas, los músicos. Todo es necesario, ya que si falta alguna de las partes el resto no tiene utilidad.



**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.

Un programador medianamente experimentado sabe capturar las pulsaciones del teclado, utilizar variables (de lo contrario no sería un programador) y guardar un fichero en un disco duro, pero puede no saber cómo funciona internamente ninguno de los elementos de hardware involucrados. ¿Se puede programar algo que no se sabe muy bien cómo funciona? En la mayoría de los casos la respuesta es afirmativa.

Cuando se va a crear un programa informático, una de las decisiones más importantes que se debe tomar consiste en elegir el lenguaje de programación en el que se va a escribir.

Existen muchos lenguajes de programación, cada uno con sus particularidades. Hay lenguajes adaptados a un tipo de temática concreta como puede ser Fortran a las matemáticas o de vocación más general como C++. También hay lenguajes orientados a objetos como Java y otros que siguen el paradigma de la programación estructurada como C. Otra clasificación divide a los lenguajes de programación en lenguajes de bajo nivel, como ensamblador, y de alto nivel, como Kotlin.

### 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.

Hay varias clasificaciones entre las que los lenguajes se distribuyen según corresponde, pero en lo que compete a esta unidad, la clasificación que resulta de interés es la que divide a los lenguajes de programación en compilados e interpretados, debido a que su pertenencia a uno o a otro grupo determina cómo son enviados al sistema operativo para su ejecución.

En los lenguajes de programación compilados, el código fuente se «traduce» a código binario que es el que entiende el sistema operativo para el que se está realizando la compilación. El resultado del proceso de compilación es un programa ejecutable solamente en el sistema operativo de destino y en la versión correspondiente (y quizás en distintas versiones si estas son compatibles a nivel binario). Una compilación realizada para Windows 11 no será ejecutable en un ordenador con sistema operativo Ubuntu 21, y viceversa. A cambio de esta restricción, el programa ejecutable, al estar optimizado para la plataforma de destino, se ejecutará de manera más rápida. A pesar de todo, si el lenguaje es estándar, en muchas ocasiones el mismo código fuente puede ser compilado para distintos sistemas sin necesitar modificaciones. C y C++ son dos claros ejemplos de lenguajes compilados.



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

En el otro lado de la balanza se encuentran los lenguajes interpretados. Los programas escritos en lenguajes interpretados no se compilan para generar un programa binario ejecutable directamente sobre el sistema operativo. Este tipo de programa se ejecuta por parte de otro programa llamado intérprete, que sí está programado para un sistema operativo concreto.

La principal ventaja es que, 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.

Las desventajas más evidentes son la 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 supondrá una pequeña pérdida de velocidad, que en algunas ocasiones podrá no ser asumible. PHP o Python son dos buenos ejemplos de este tipo de lenguaje.

Existe, además, un punto intermedio entre los lenguajes compilados e interpretados y su máximo exponente es Java. Java es un lenguaje de programación compilado, pero en lugar de realizarse la compilación para un sistema operativo real se efectúa para un «sistema operativo virtual», la conocida como JVM o *Java Virtual Machine*. Es la JVM la encargada de traducir el código compilado en Java al lenguaje del sistema operativo durante la ejecución de los programas. Por lo tanto, el escenario es parecido al de los lenguajes de programación interpretados, ya que se requiere tener la máquina virtual de Java instalada y la traducción en ejecución ralentizará algo los programas. En cambio, se dispone de programas multiplataforma que se ejecutan en un entorno seguro y controlado como es la JVM.

Dadas, por lo tanto, estas dos opciones (más la intermedia abanderada por la máquina virtual de Java) a la hora de escribir un programa, se debe saber cómo se relacionan estos con el sistema operativo. Todos ellos tienen un punto de partida común, el **código fuente**, que está formado por las líneas de código escritas por el programador, siendo a partir de este donde comienza la relación con el sistema operativo.

En los lenguajes compilados, el proceso de compilación transforma el código fuente en un **código ejecutable**. El código obtenido es directamente ejecutable sobre el sistema operativo de destino.

En los lenguajes interpretados, el código fuente se procesa en el momento de la ejecución, por lo que no existe un código ejecutable como tal.

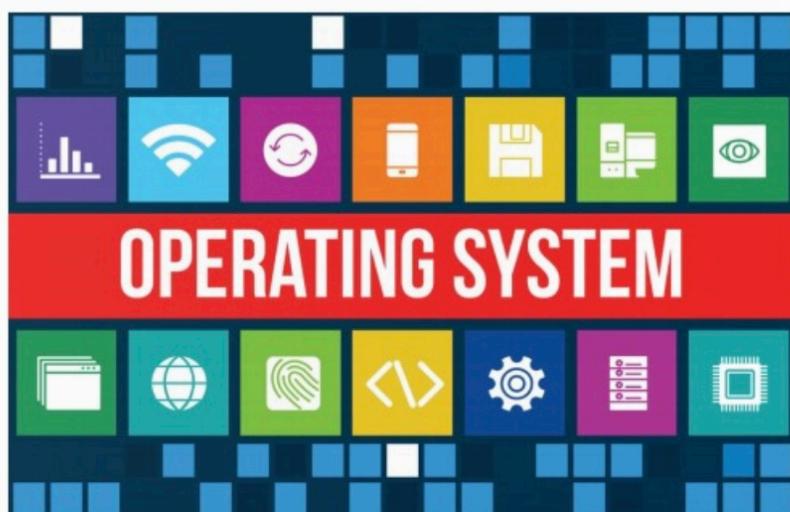
### Sabías que...

Aunque los lenguajes interpretados no generan un código ejecutable, disponen de técnicas de optimización que permiten que en sucesivas ejecuciones algunas de las fases del análisis del código no se repitan, agilizando dichas ejecuciones.

### 1.1.3. Programas. Ejecutables. Procesos. Servicios

Programa, proceso, ejecutable y servicio son términos que hacen referencia a elementos distintos, pero íntimamente relacionados. Es necesario comprender las particularidades de cada uno de ellos para poder avanzar en el conocimiento de la 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.

Para llegar a poder ejecutar un programa primero hay que obtenerlo o, en el caso de los programadores, crearlo, para posteriormente proceder a su ejecución. El proceso de creación por parte del programador y de ejecución por parte del usuario final de un programa compilado es el siguiente:

1. El programador escribe el **código fuente** con un editor de texto y lo almacena en un fichero.
2. El programador compila el código fuente utilizando un compilador, generando un **programa ejecutable**. Este programa contiene instrucciones comprensibles por el sistema operativo para el cual se realizó la compilación.
3. El usuario ejecuta el programa ejecutable, generando un **proceso**.

Por lo tanto, se puede afirmar que un programa, al ser ejecutado por un usuario, genera un proceso en el sistema operativo: **un proceso es un programa mientras se encuentra en ejecución**.

Por su parte, un **servicio** es también un programa cuya ejecución se realiza en segundo plano y que no requiere la interacción del usuario. Normalmente, se arranca de manera automática por el sistema operativo y está en constante ejecución.

Todos los sistemas operativos modernos distinguen entre procesos y servicios, aunque en esta unidad se pondrá el foco en los primeros.

En la Figura 1.6 se puede observar el Administrador de tareas de Windows 10. 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.

Nombre	Estado	PID	13% CPU	72% Memoria	1% Disco	0% Red
<b>Aplicaciones (8)</b>						
> 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

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

En Unix y sus derivados, el comando ps (process status) muestra, con distinto nivel de detalle, información sobre los procesos que están en ejecución en un momento determinado, incluyendo también su identificador de proceso (PID). En Windows se puede ejecutar este comando desde la consola PowerShell.

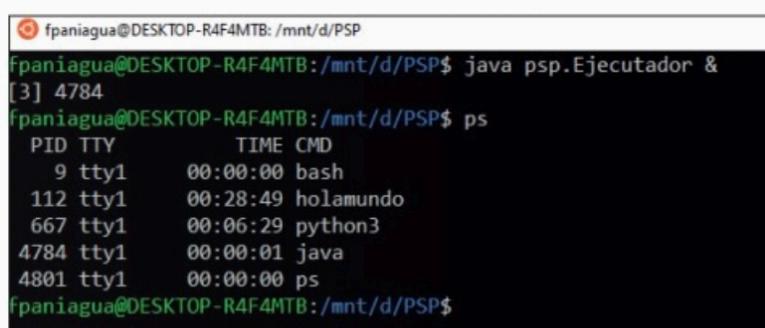
En la Figura 1.7 se muestra el listado de procesos de un sistema Linux. El proceso identificado con el PID 112 se corresponde con un programa escrito en C con el nombre «holamundo».

```
fpanagua@DESKTOP-R4F4MTB:/mnt/d/PSP
fpanagua@DESKTOP-R4F4MTB:/mnt/d/PSP$ ./holamundo &
[1] 112
fpanagua@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
fpanagua@DESKTOP-R4F4MTB:/mnt/d/PSP$
```

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

Es interesante reseñar que cuando el programa que se ejecuta no es compilado, el proceso que se arranca no es el propio programa, sino el del intérprete (como en Python) o el de la máquina virtual (como en Java). En ambos casos el nombre del proceso no coincidirá con el nombre del programa.

En la captura del listado de procesos de Ubuntu (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.

A terminal window titled 'fpaniagua@DESKTOP-R4F4MTB' showing the output of the 'ps' command. The output lists several processes: bash (PID 9), holamundo (PID 112), python3 (PID 667), java (PID 4784), and ps (PID 4801).

```
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.

Como se ha comentado anteriormente, el número de procesos que puede estar conviviendo en un sistema es alto y muy superior al número de núcleos de ejecución. Es el sistema operativo el encargado de conseguir que sea posible repartir los limitados recursos del ordenador de forma ordenada, justa y eficiente.

## Actividad propuesta 1.2

### Destrucción de proceso de terminal

Abre un editor de texto y destruye el proceso desde un terminal con el comando *kill* a partir de su PID. En Windows es posible ejecutar dicho comando desde la consola PowerShell.

## 1.1.4. Computación concurrente y paralela

Salvo los sistemas más antiguos, sencillos o limitados, como MS-DOS o los embebidos en las placas de desarrollo Arduino, todos los sistemas operativos modernos disponen de la capacidad de realizar multitarea o multiproceso.

La **multitarea**, como indica su nombre, es la **capacidad de realizar varias tareas simultáneamente**, frente a la restricción de la monotarea, en donde las tareas se ejecutan una detrás de otra. Cuando se trabaja en un sistema multitarea, varias tareas avanzan a la vez, aunque pueden hacerlo de diferentes maneras.

Un sistema basado en un único procesador con un único núcleo es capaz de realizar multitarea mediante el uso de la **concurrency**. En la computación concurrente, los tiempos de CPU se reparten entre los distintos procesos según una planificación dirigida por el sistema operativo. Si la CPU es suficientemente rápida, el número de procesos limitado y el planificador tienen un buen diseño, todas las tareas avanzarán a la vez (aparentemente) pese a que en un ciclo de CPU en un momento dado solo se puede ejecutar una instrucción. En una unidad de tiempo de computación solo avanza un proceso. 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 *concurrente*.

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*. En este tipo de procesamiento, los procesos se dividen en pequeñas subtareas (hilos) que se ejecutan en los diferentes núcleos, consiguiendo una reducción en los tiempos de ejecución de los procesos.



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

Como resumen, se pueden definir los conceptos de procesamiento concurrente y paralelo de la siguiente manera:

- **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.

Se puede, para finalizar, concluir que el procesamiento concurrente es responsabilidad del sistema operativo mientras que el procesamiento paralelo es una responsabilidad compartida entre el sistema operativo y el programa.

## ■■■ 1.1.5. Programación distribuida

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.

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

## ■■■ 1.1.6. Hilos

Un programa básico está compuesto por una serie de sentencias que se ejecutan de manera secuencial y síncrona: hasta que no se completa la ejecución de la primera de las sentencias no se comienza con la ejecución de la segunda, y así sucesivamente hasta terminar la ejecución del programa completo.

En muchos casos, es necesaria esta secuencialidad y sincronía, ya que los diferentes pasos del algoritmo programado son dependientes entre sí y no existe la posibilidad de invertir el orden de ejecución sin generar un resultado del proceso erróneo. En otros casos, en cambio, un algoritmo podría trocearse en varias unidades más pequeñas, ejecutar cada una de ellas por separado y en paralelo, juntar los resultados sin que importe el orden en que se obtengan y generar el resultado final. Esta técnica se conoce como **programación multihilo**.

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 continúa

con el estado del proceso original (padre), pero a partir de la creación cada proceso mantiene su propio estado de memoria.

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. }
```

La salida generada por la ejecución es la siguiente:

```
1 Comenzando la ejecución
2 Antes de bifurcar
3 Después de bifurcar
4 Después de bifurcar
5 Id. padre:143 Id. hijo:143 Contador:2
6 Id. hijo:144 Id. padre:143 Contador:2
```

Sin entrar en detalles sintácticos sobre el lenguaje C, los aspectos relevantes de la creación de la bifurcación son los siguientes:

- La variable *contador* se declara e inicializa a 0 en la línea 5.
- En la línea 6 se muestra el texto «Comenzando la ejecución» una única vez, ya que en este punto el proceso es único.
- En la línea 10 se muestra el texto «Antes de bifurcar» una única vez, ya que en este punto el proceso es único.
- En la línea 11 el valor de *contador* se incrementa en 1, tomando el valor 1.
- En la línea 12 se crea la bifurcación.

- En la línea 13 el valor de *contador* se incrementa en 1, una vez por cada proceso. No obstante, cada proceso dispone de su propio espacio de memoria por lo que la variable de ambos procesos es distinta.
- En la línea 14 se muestra el texto «Después de bifurcar» dos veces, una por cada copia del proceso.
- En la línea 16 se muestra la información referente al proceso hijo. Se puede observar que el PID es 144 y el valor de la variable *contador* es 2.
- En la línea 18 se muestra la información referente al proceso padre. Se puede observar que el PID es 143 y el valor de la variable *contador* es 2.

### Sabías que...



En Java existe el *framework Fork/Join* desde la versión 7. Proporciona herramientas para aprovechar los núcleos del sistema operativo y realizar procesamiento paralelo.

## 1.2. Procesos: conceptos teóricos

De manera simplificada, se puede definir un proceso como un programa en ejecución. Algunos ejemplos de proceso podrían ser las instancias de un navegador web, de un procesador de textos, de un entorno de desarrollo o de una máquina virtual de Java.

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.

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.

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. Lógicamente, el cambio de contexto implica un consumo de recursos y en circunstancias extremas y con un planificador mal diseñado, el sistema podría estar constantemente realizando cambios de contexto sin que los procesos implicados avanzasen en la ejecución.

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.

Es el sistema operativo el encargado de la gestión de los procesos, quedando en la responsabilidad del programador el crear los programas que van a dar lugar a los procesos y en manos de los usuarios ejecutarlos.

### ■ ■ ■ 1.2.1. Gestión y estados de los procesos

En la actualidad, prácticamente todos los sistemas de computación (ordenadores, teléfonos móviles inteligentes —smartphones—, tabletas, relojes inteligentes —smartwatches—, videoconsolas, etc.) son multiproceso, por lo que tienen la capacidad de mantener en ejecución simultánea varios programas o procesos. Normalmente, el número de estos es mucho mayor que el número de núcleos de ejecución que hay disponibles en el procesador o procesadores que tenga el dispositivo.

Los procesos necesitan recursos y estos son limitados. El procesador, la memoria, el acceso a los sistemas de almacenamiento o a los diferentes dispositivos son algunos de ellos. La pregunta que surge como consecuencia de esta afirmación es la siguiente: ¿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.

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.

Se puede sintetizar que el objetivo del planificador es conseguir que todos los procesos terminen lo antes posible aprovechando al máximo los recursos del sistema. La tarea, como se puede suponer, es compleja.

Es posible imaginar al planificador como el responsable del almacén de las herramientas de un taller. El número de herramientas es limitado y mucho menor que el número de empleados que las necesitan para realizar su trabajo. El responsable del almacén debe satisfacer los siguientes requisitos:

- Todos los empleados tienen acceso a las herramientas para poder realizar su trabajo.
- Todos los empleados deben tener acceso a las herramientas durante aproximadamente la misma cantidad de tiempo.
- Ningún empleado puede quedarse la herramienta de forma permanente.
- Ningún empleado puede estar eternamente esperando a que le llegue el turno de usar la herramienta.
- Si hay una emergencia, el responsable debe tener la capacidad de recuperar las herramientas del empleado que las esté utilizando para dicha emergencia.

Si este proceso de gestión de recursos se realiza a la suficiente velocidad y los empleados fuesen muy rápidos, la percepción de un observador externo sería que todos los empleados avanzan en la ejecución de sus respectivas tareas a la vez. Eso es, en cierto modo, lo que ocurre en un sistema de computación gracias al planificador de procesos.

Existen muchos algoritmos para la planificación de los procesos, pero su enumeración y explicación están fuera del alcance de este libro. No obstante, hay que considerar que cada sistema operativo utiliza sus propias estrategias de gestión de recursos a distintos niveles y que dichas estrategias influyen de manera directa en el funcionamiento del sistema.

Para realizar la gestión de los procesos, el planificador necesita conocer el estado en que se encuentran. En un momento dado, un proceso se encuentra en un estado de un conjunto de estados posibles. Los estados básicos en los que puede estar un proceso son los siguientes:

- **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.

Las posibles transiciones entre estos estados se recogen en el diagrama de transición de estados que representa su ciclo de vida y que se muestra en la Figura 1.11.

Este es un diagrama simplificado, ya que no recoge estados intermedios relacionados con la suspensión (listo y suspendido o bloqueado y suspendido, por citar algunos ejemplos clásicos), pero permite tener una visión panorámica del flujo de cambios de estado.



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

No es infrecuente que un sistema se detenga durante un breve instante de tiempo cuando está sometido a una excesiva carga de trabajo (fenómeno conocido como *lag*, también asociado a las fluctuaciones de las redes de datos). Esto puede deberse, entre otras razones, a que, en el establecimiento de prioridades del sistema operativo, alguna tarea ha acaparado los recursos del sistema durante una breve fracción de tiempo, dejando «pausados» al resto de procesos. Estas pausas se perciben especialmente durante la reproducción de contenidos multimedia o en la ejecución de videojuegos y son inevitables si la carga de trabajo del sistema excede a su capacidad.

### Recuerda



Ni el programador ni el usuario tienen control directo sobre los estados de un proceso, ya que el ciclo de vida es responsabilidad exclusiva del planificador del sistema operativo.

## ■ ■ ■ 1.2.2. Comunicación entre procesos

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.

Algunas de estas alternativas son las siguientes:

- **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. Se tratan en detalle en la Unidad 3 de este libro.

- **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). En la Actividad resuelta 1.1 de esta unidad se puede observar el funcionamiento de este tipo de técnica.
- **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. En Java, este tipo de llamada se realiza mediante la tecnología conocida como RMI (*Remote Method Invocation*), equivalente a las RPC, pero orientada a objetos. La tecnología RMI se presenta en la Unidad 4 de este libro.
- **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.

#### Sabías que...

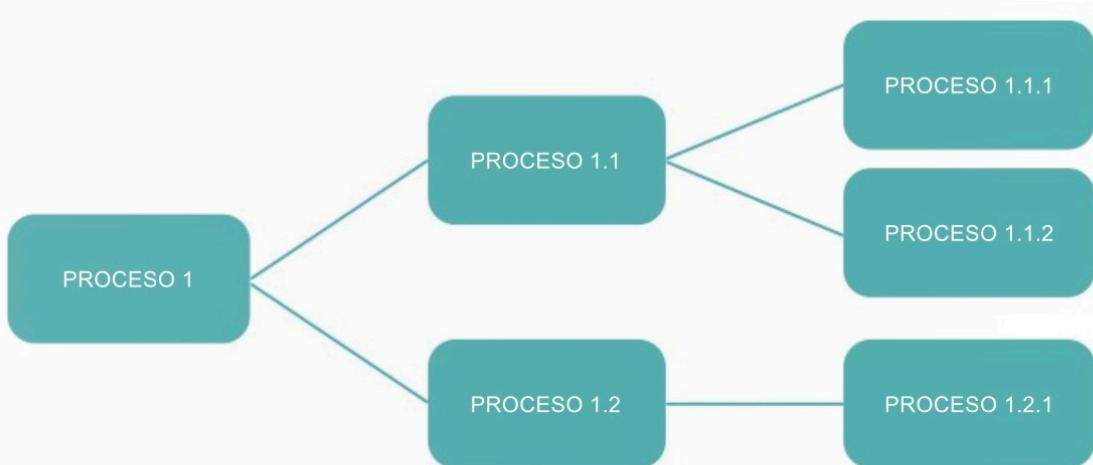
En ocasiones, el documento conocido como «Manual de Explotación» o «Manual de Administración» de un sistema software debe recoger los posibles códigos de finalización de los procesos junto con su descripción. Esta información es necesaria para poder construir flujos de trabajo que incluyan a varios procesos, permitiendo establecer las rutas de ejecución en función de los códigos de finalización.



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.

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.

En Java, estas necesidades se satisfacen con las clases y métodos que se muestran en la Tabla 1.1.

**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()

Pese a que en el Apartado 1.3 se estudian las clases *Runtime* y *ProcessBuilder*, es interesante presentar un ejemplo de sincronización entre procesos realizada en Java.

La siguiente clase escrita en Java representa un proceso que, tras intentar realizar aquello para lo que está programado, devuelve un código de retorno 103 utilizando el método *exit* de la clase *System*, que significa que ha ocurrido un error de un tipo determinado.

```

1 package es.paraninfo.sincronizacion;
2
3 public class ProcesoSecundario {
4     public static void main(String[] args) {
5         System.out.println("Proceso secundario...");
6         System.exit(103);
7     }
8 }
```

Por otra parte, la clase siguiente representa un proceso que ejecuta el proceso anterior (método *exec*), queda a la espera de que termine (método *waitFor*), recoge el valor de finalización del proceso y lo evalúa para tomar una decisión.

```

1 package es.paraninfo.sincronizacion;
2
3 public class ProcesoPrincipal {
4     public static void main(String[] args) {
5         try {
6             String[] infoProceso =
7                 {"java", "es.paraninfo.sincronizacion.ProcesoSecundario"};
8             Process proceso = Runtime.getRuntime().exec(infoProceso);
9             int valorRetorno = proceso.waitFor();
10            if (valorRetorno==0) {
11                System.out.println("El proceso se ha
completado satisfactoriamente");
12            } else {
13                System.out.println("El proceso ha fallado.
Código de error:" + valorRetorno);
14            }
15        } catch (Exception e) {
16            e.printStackTrace();
17        }
18    }
}
```

El resultado de la ejecución del programa *ProcesoPrincipal* es el siguiente:

```
El proceso ha fallado. Código de error:103
```

El proceso principal recoge el código de finalización del proceso secundario, lo evalúa y en función del valor que toma elige uno u otro camino.

En este ejemplo se muestra uno u otro mensaje de texto por la salida estándar en función del código de finalización del proceso secundario. Si en lugar de mostrar mensajes se hubiesen ejecutado procesos, se habría creado un flujo de ejecución de trabajos.

### Recuerda



Este tipo de sincronización se puede realizar entre procesos escritos en distintos lenguajes de programación.

## Actividad propuesta 1.3

### Redacción de descripciones asociadas a códigos de finalización

Un programa recalcula la tarifa de los precios de los productos de una empresa almacenados en una base de datos creada según el índice de precios al consumo (IPC) obtenido de un servicio web. Una vez recalculados se deben actualizar en la base de datos y enviar por correo electrónico al responsable correspondiente.

Completa la siguiente tabla, redactando la descripción de cuatro posibles finalizaciones incorrectas.

Código de finalización	Descripción
0	El proceso ha terminado correctamente.
1	
2	
3	
4	

## ■ 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?

La respuesta es sí, siempre y cuando se acote el significado del concepto «aplicación multiproceso», ya que se trata de un término cuyo alcance es difuso. En esta unidad se aborda la programación de aplicaciones multiproceso como la capacidad de coordinar la ejecución de un conjunto de aplicaciones para lograr un objetivo común.

Por ejemplo, si se dispone de un sistema compuesto por un conjunto de procesos que deben ejecutarse de manera individual, pero que tienen dependencias entre sí, se necesita disponer de un mecanismo de gestión y coordinación.

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**.

A continuación, se presentan estas dos alternativas y sus diferencias a la hora de crear y ejecutar los procesos.

### 1.3.1. Creación de procesos con *Runtime*

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");
```

En muchos casos, los procesos necesitan parámetros para iniciarse. El método *exec* puede recibir una cadena de caracteres (un objeto de la clase *String*) y en dicha cadena, separados por espacios, se indicarán, además del programa que se desea ejecutar, los diferentes parámetros.

En el siguiente código se está ejecutando el bloc de notas indicando que «notas.txt» es el fichero que debe abrir o crear si no existe.

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

Alternativamente, se puede crear el proceso proporcionando un array de objetos *String* con el nombre del programa y los parámetros.

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

El siguiente nivel consiste en gestionar el proceso lanzado. Para ello, se debe obtener la referencia a la instancia de la clase *Process* proporcionada por el método *exec*. Es este objeto el que proporciona los métodos para conocer el estado de la ejecución del proceso.

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

Si se necesita esperar a que el proceso ejecutado termine y conocer el estado en que ha finalizado dicha ejecución, se puede utilizar el método *waitFor*. Este método suspende la ejecución del programa que ha arrancado el proceso quedando a la espera de que este termine, proporcionando además el código de finalización.

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

La clase *Process* representa al proceso en ejecución y permite obtener información sobre este. Los principales métodos que proporciona dicha clase son los que se recogen en la Tabla 1.2.

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

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

### Recuerda

Se puede ampliar información sobre las clases y métodos en la documentación de Java.

Por ejemplo, se puede consultar la documentación completa del API de la clase *Process* en la siguiente dirección:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Process.html>



## Actividad resuelta 1.1

### Conexión de procesos Java y Python con lectura de salida

Desde un programa escrito en Java, ejecutar un programa escrito en Python y leer los datos que genera como salida.

#### Solución

##### Proceso iniciador (Java). Fichero App.java

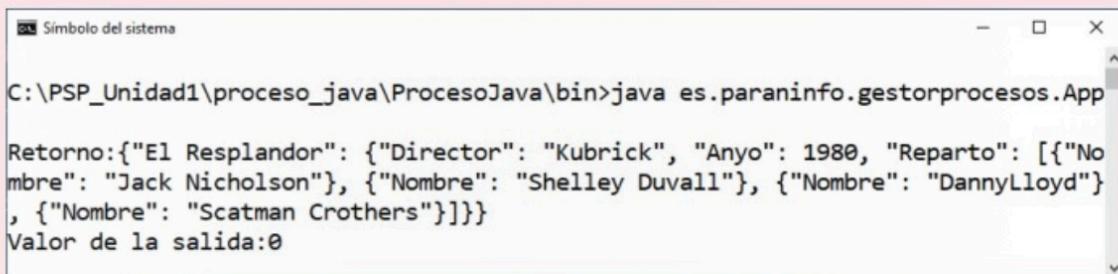
```

1 package es.paraninfo.gestorprocesos;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6
7 public class App {
8
9     public static void main(String[] args) {
10         try {
11
12             Process proceso = new
13                 ProcessBuilder("Python",
14                 "C:\\\\PSP_Unidad1\\\\proceso_python\\\\proceso_python.py").start();
15
16             BufferedReader br = new
17                 BufferedReader(new
18                 InputStreamReader(proceso.getInputStream()));
19             proceso.waitFor();
20             int exitStatus = proceso.exitValue();
21             System.out.println("Retorno:" + br.readLine());
22             System.out.println("Valor de la salida:" + exitStatus);
23         } catch (IOException | InterruptedException e) {
24             e.printStackTrace();
25         }
26     }
27 }
```

##### Proceso iniciado (Python). Fichero proceso\_python.py

```

1 import json
2 import sys
3
4 pelicula = {"El Resplandor":
5             {
6                 "Director": "Kubrick",
7                 "Anyo": 1980,
8                 "Reparto": [
9                     {"Nombre": "Jack Nicholson"},
10                    {"Nombre": "Shelley Duvall"},
11                    {"Nombre": "Danny Lloyd"},
12                    {"Nombre": "Scatman Crothers"}
13                ]
14            }
15        }
16
17 print(json.dumps(pelicula))
18 sys.exit(0)
```

**Ejecución y salida:**


```

Símbolo del sistema
C:\PSP_Unidad1\proceso_java\ProcesoJava\bin>java es.paraninfo.gestorprocesos.App
Retorno:{"El Resplandor": {"Director": "Kubrick", "Anyo": 1980, "Reparto": [{"Nombre": "Jack Nicholson"}, {"Nombre": "Shelley Duvall"}, {"Nombre": "Danny Lloyd"}, {"Nombre": "Scatman Crothers"}]}}

Valor de la salida:0
  
```

Como se puede observar, desde el programa escrito en Java (proceso iniciador) se está ejecutando el intérprete de Python (proceso iniciado) que recibe un parámetro (el nombre del programa Python a ejecutar). El proceso iniciador queda a la espera de que el proceso iniciado termine para leer la salida que este proporciona a través del stream de salida, en este caso una cadena de caracteres que contiene un fichero JSON.

### 1.3.2. Creación de procesos con *ProcessBuilder*

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();
```

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 subprocessos 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 }
```

Además del método `start`, la clase `ProcessBuilder` dispone de métodos para consultar y gestionar algunos parámetros relativos a la ejecución del proceso. Los métodos más relevantes de `ProcessBuilder` se muestran en la Tabla 1.3.

**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.

A continuación, se muestran algunos ejemplos relacionados con los métodos expuestos.

El siguiente código crea un objeto `ProcessBuilder` y determina el directorio de trabajo del proceso:

```

ProcessBuilder pBuilder = new
    ProcessBuilder("Notepad.exe","datos.txt");
pBuilder.directory(new File("c:/directorio_salida/"));
```

Para acceder a la información del entorno de ejecución, el método `environment` devuelve un objeto `Map` con la información proporcionada por el sistema operativo. El siguiente ejemplo muestra por pantalla el número de procesadores disponibles en el sistema:

```

1 ProcessBuilder pBuilder = new
2     ProcessBuilder("Notepad.exe","datos.txt");
3 java.util.Map<String, String> env = pBuilder.environment();
4 System.out.println("Número de procesadores:" +
5     env.get("NUMBER_OF_PROCESSORS"));
```

## Actividad resuelta 1.2

### Conexión de procesos Java y C con configuración de directorio de trabajo

Desde un programa escrito en Java ejecutar un programa compilado escrito en C indicando el directorio de trabajo. El programa en C debe escribir un texto en un fichero. Dicho fichero debe guardarse en el directorio de trabajo indicado.

#### Solución

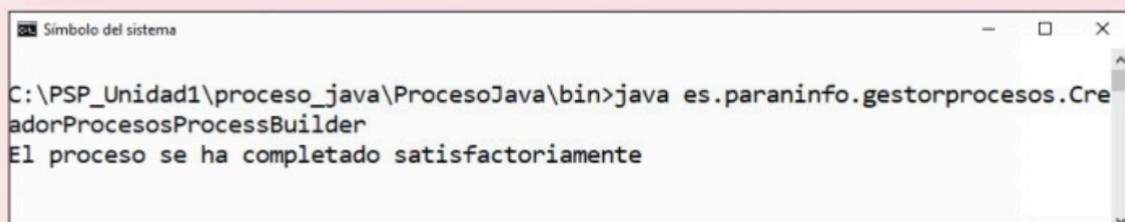
##### Proceso iniciador (Java). Fichero CreadorProcesosProcessBuilder.java

```
1 package es.paraninfo.gestorprocesos;
2
3 import java.io.File;
4 import java.io.IOException;
5
6 public class CreadorProcesosProcessBuilder {
7
8     public static void main(String[] args) {
9
10         try {
11             ProcessBuilder pBuilder = new
12             ProcessBuilder("C://PSP_Unidad1//proceso_c//proceso_c.exe");
13             pBuilder.directory(new
14             File("C://directorio_salida"));
15             Process proceso = pBuilder.start();
16             int valorRetorno = proceso.waitFor();
17             if (valorRetorno==0) {
18                 System.out.println("El proceso se ha
19                 completado satisfactoriamente");
20             } else {
21                 System.out.println("El proceso ha fallado.
22                 Código de error:" + valorRetorno);
23             }
24         }
25     }
26 }
```

##### Proceso iniciado (C). Fichero proceso\_c.c.

Nota: este fichero debe ser compilado.

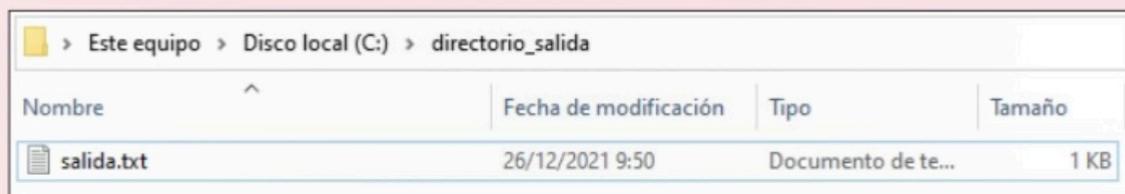
```
1 #include <stdio.h>
2 int main() {
3     FILE* fichero;
4     fichero = fopen("salida.txt", "wt");
5     fputs("Este texto está generado por el proceso escrito en
6     C", fichero);
7     fclose(fichero);
8 }
```

**Ejecución y salida:**

```
C:\PSP_Unidad1\proceso_java\ProcesoJava\bin>java es.paraninfo.gestorprocesos.CreadorProcesosProcessBuilder
El proceso se ha completado satisfactoriamente
```

El proceso escrito en C escribe el fichero en el directorio de trabajo indicado a través del objeto *ProcessBuilder*.

Es importante recordar que el programa ejecutable escrito en C se encuentra en una ubicación diferente al directorio de salida y que en el código de dicho programa no se hace ninguna referencia a la ubicación del archivo generado, lo que implica que el proceso se ha ejecutado utilizando como directorio de trabajo el indicado en el objeto *ProcessBuilder*.



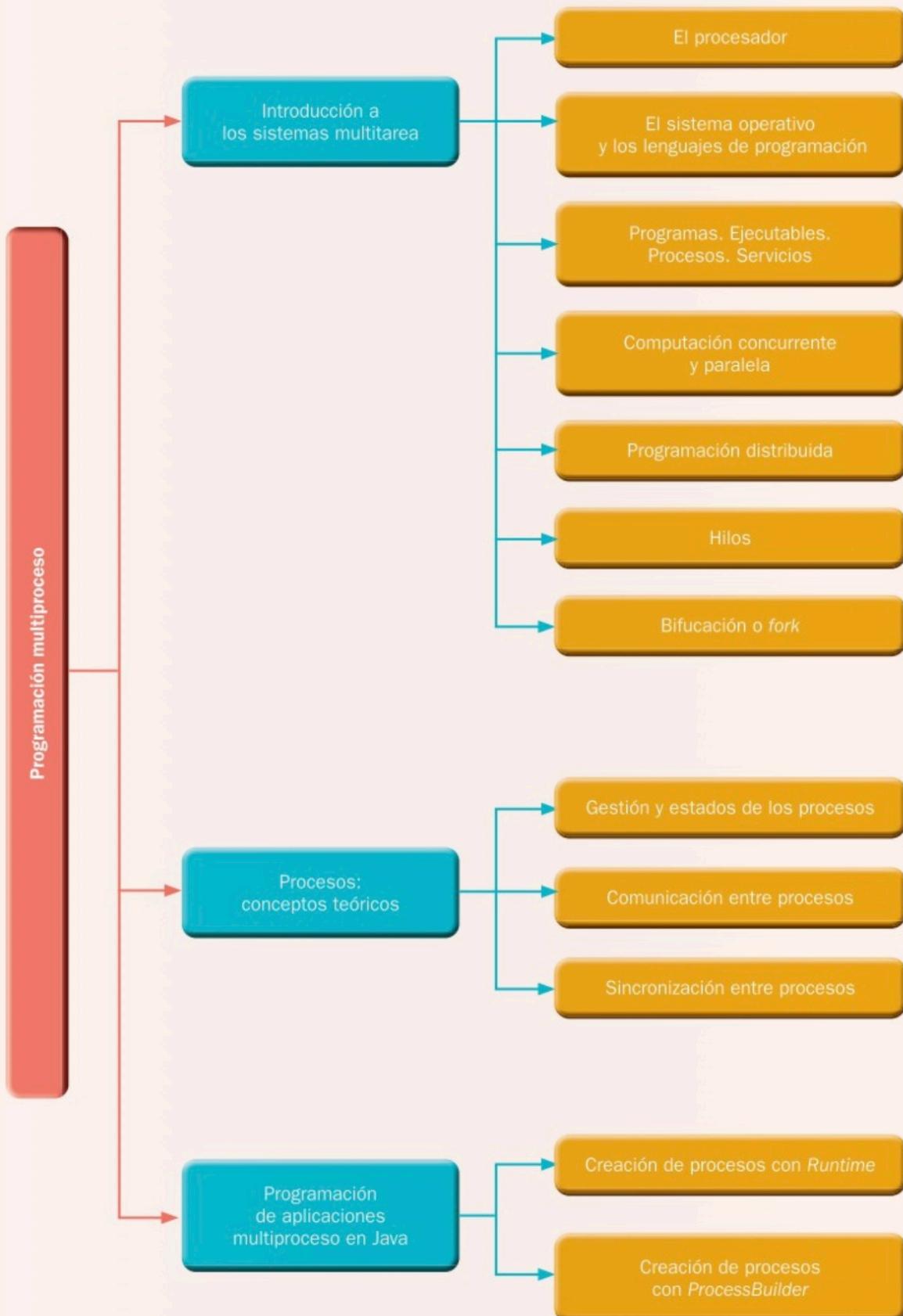
Este equipo > Disco local (C:) > directorio_salida			
Nombre	Fecha de modificación	Tipo	Tamaño
salida.txt	26/12/2021 9:50	Documento de te...	1 KB

El fichero tiene el contenido siguiente:

Este texto está generado por el proceso escrito en C

# MAPA CONCEPTUAL

## 1. PROGRAMACIÓN MULTIPROCESO



## Actividades de comprobación

- 1.1. ¿Cuál de los siguientes sistemas operativos no es multitarea?**
  - a) Unix.
  - b) Linux.
  - c) Windows 10.
  - d) MS-DOS.
  
- 1.2. En computación, se entiende por multitarea:**
  - a) La capacidad que tiene un ordenador de realizar tareas complejas.
  - b) La capacidad que tiene un ordenador de ejecutar un programa detrás de otro.
  - c) La capacidad que tiene un ordenador de realizar tareas sincronizadas entre sí.
  - d) La capacidad que tiene un ordenador de ejecutar varios programas al mismo tiempo.
  
- 1.3. ¿Qué elemento del ordenador se encarga de gestionar la multitarea?**
  - a) El lenguaje de programación elegido para crear los programas.
  - b) El sistema operativo.
  - c) El sistema de archivos.
  - d) Ninguna de las respuestas anteriores es correcta.
  
- 1.4. ¿Qué afirmación referente a los lenguajes de programación compilados no es correcta?**
  - a) En condiciones normales son más rápidos que los interpretados.
  - b) Requieren una compilación específica para cada sistema operativo.
  - c) Si el lenguaje es estándar, la misma compilación sirve para todos los sistemas operativos.
  - d) El código fuente debe compilarse para obtener el código binario para ser ejecutado.
  
- 1.5. El identificador de proceso se suele identificar por las siglas:**
  - a) CPU.
  - b) PID.
  - c) IPD.
  - d) TTY.
  
- 1.6. El procesamiento que se ejecuta en diferentes ordenadores independientes, pero conectados y sincronizados se denomina:**
  - a) Distribuido.
  - b) Concurrente.
  - c) Paralelo.
  - d) Monohilo.
  
- 1.7. ¿Cuál de los siguientes no es un objetivo del planificador de procesos del sistema operativo?**
  - a) Maximizar el rendimiento del sistema.
  - b) Maximizar los tiempos de respuesta.
  - c) Maximizar la equidad en el reparto de los recursos.
  - d) Minimizar los tiempos de espera.

- 1.8.** ¿Cuál es el estado al que puede pasar un proceso que está en estado Listo?
  - a) Nuevo.
  - b) Bloqueado.
  - c) En ejecución.
  - d) Finalizado.
  
- 1.9.** ¿Qué afirmación referente a la clase *Runtime* de Java es errónea?
  - a) Permite lanzar un proceso indicando los parámetros de ejecución.
  - b) Permite quedar a la espera de la terminación del proceso lanzado.
  - c) Permite conocer el valor de las variables internas del proceso lanzado.
  - d) Permite obtener el estado de la finalización el proceso lanzado.
  
- 1.10.** ¿Cómo se llama el método de la clase *Process* de Java que hace esperar a que termine el proceso lanzado?
  - a) sleep().
  - b) finish().
  - c) waitFor().
  - d) waitEnd().

## Actividades de aplicación

- 1.11.** Explica las diferencias entre los lenguajes de programación interpretados y compilados.
- 1.12.** Describe las diferencias entre programa y proceso.
- 1.13.** Explica en qué consiste la programación distribuida.
- 1.14.** Encuentra las diferencias entre ejecutar dos procesos o realizar una bifurcación o *fork*.
- 1.15.** Diseña un sistema basado en bases de datos para intercambiar información entre dos procesos.
- 1.16.** Diseña un sistema basado en ficheros de texto para sincronizar procesos.
- 1.17.** Decide qué sistema de sincronización podría servir para sincronizar procesos que se ejecutan en ordenadores independientes conectados a través de internet.
- 1.18.** Explica por qué los hilos (*threads*) consumen menos recursos que los procesos.
- 1.19.** Explica qué tarea cumple el planificador de procesos dentro del sistema operativo.
- 1.20.** Diseña un planificador de proceso sencillo. Determina cómo se asigna el tiempo de CPU a los distintos procesos que gestiona para que puedan avanzar todos más o menos simultáneamente.
- 1.21.** Elabora el listado de los valores y sus descripciones que podrá devolver la ejecución de un proceso que tenga que acceder a una base de datos, a un ordenador a través de la red y a internet.
- 1.22.** Desarrolla un programa en Java que ejecute el navegador web Firefox. Utiliza la clase *Runtime*. Intenta que el navegador abra directamente una página web.

### ■ Actividades de ampliación

- 1.23. Suponiendo que se necesita una capacidad de cálculo de 40 núcleos de procesamiento, elabora un presupuesto de los ordenadores que se necesitarían para montar un sistema distribuido capaz de proporcionar dicha capacidad.
- 1.24. Busca información referente a la programación de hilos (*threads*) y a los lenguajes que los aceptan de forma nativa.
- 1.25. Se conoce como «renderización» al proceso de generar una imagen bidimensional a partir de un modelo tridimensional. Busca información referente al renderizado distribuido entre varias computadoras. Descubre si existen programas comerciales capaces de realizar esta actividad.
- 1.26. Busca información referente al primer sistema operativo de Microsoft multiproceso. Averigua su nombre, en qué año se creó y en qué tipo de procesadores se ejecutaba.
- 1.27. Elige tres procesadores de distintos precios que estén a la venta para el público general. Averigua cuántos núcleos de ejecución tiene cada uno de ellos.
- 1.28. Existen muchos algoritmos de planificación de procesos. Busca tres de ellos y estudíalos, analizando su funcionamiento y diferencias.

### Enlaces web de interés

-  **Intel** - <https://www.intel.es>  
(Fabricante de procesadores)
-  **AMD** - <https://www.amd.com/es>  
(Fabricante de procesadores)
-  **Windows** - <https://www.microsoft.com/es-es/windows>  
(Sistema operativo de Microsoft)
-  **Ubuntu** - <https://ubuntu.com>  
(Distribución del sistema operativo Linux)
-  **RMI** - <https://www.ibm.com/docs/es/sdk-java-technology/8?topic=orb-rmi-rmi-iiop>  
(Información sobre RMI de IBM)
-  **RPC** - <https://www.ionos.es/digitalguide/servidores/know-how/que-es-rpc/>  
(Artículo sobre RPC de la web de IONOS)

# ACTIVIDADES FINALES

## 1. PROGRAMACIÓN MULTIPROCESO

-  **ProcessBuilder** - <https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html>  
*(Información y API sobre la clase ProcessBuilder de Java)*
-  **RunTime** - <https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html>  
*(Información y API sobre la clase RunTime de Java)*
-  **Process** - <https://docs.oracle.com/javase/8/docs/api/java/lang/Process.html>  
*(Información y API sobre la clase Process de Java)*