1 Programación de procesos

OBJETIVOS DEL CAPÍTULO

- ✓ Comprender de los conceptos básicos del funcionamiento de los sistemas en lo relativo a la ejecución de diferentes programas.
- ✓ Comprender el concepto de concurrencia y cómo el sistema puede proporcionar multiprogramación al usuario.
- ✓ Entender las políticas de planificación del sistema para proporcionar multiprogramación y multitarea.
- ✓ Familiarizarse con la programación de procesos entendiendo sus principios y formas de aplicación.

1.1 CONCEPTOS BÁSICOS

Para poder empezar a entender cómo se ejecutan varios programas a la vez, es imprescindible adquirir ciertos conceptos.

- **Programa**: se puede considerar un programa a toda la información (tanto código como datos) almacenada en disco de una aplicación que resuelve una necesidad concreta para los usuarios.
- **Proceso**: cuando un programa se ejecuta, podremos decir de manera muy simplificada que 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. Esto incluye tres cosas:
  - Un contador del programa: algo que indique por dónde se está ejecutando.
  - Una imagen de memoria: es el espacio de memoria que el proceso está utilizando.
  - Estado del procesador: se define como el valor de los registros del procesador sobre los cuales se está ejecutando.

¿SABÍAS QUE?

La memoria principal almacena toda la información de un programa (instrucciones y datos) mientras se está procesando en la CPU. Todos los programas y datos antes de ser procesados por el procesador han de ser almacenados en esta memoria. Para que un proceso se pueda ejecutar tiene que estar en memoria toda la información que necesita. En este sentido, el tipo y cantidad de memoria, pueden influir decisivamente en la velocidad del ordenador. Si el tamaño de memoria es pequeño, se deberán traer de donde estén almacenados los programas (por ejemplo, el disco duro) más veces los datos del programa necesarios, ralentizando el ordenador.

Es importante destacar que los procesos son entidades independientes, aunque ejecuten el mismo programa. De tal forma, pueden coexistir dos procesos que ejecuten el mismo programa, pero con diferentes datos (es decir, con distintas imágenes de memoria) y en distintos momentos de su ejecución (con diferentes contadores de programa).

EJEMPLO 1.1

Se pueden tener, por ejemplo, dos instancias del programa Microsoft 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.

- **Ejecutable**: un fichero ejecutable contiene la información necesaria para crear un proceso a partir de los datos almacenados de un programa. Es decir, llamaremos "ejecutable" al fichero que permite poner el programa en ejecución como proceso.
- **Sistema operativo**: programa que hace de intermediario entre el usuario y las aplicaciones que utiliza y el hardware del ordenador. Entre sus objetivos, se pueden destacar:
  - Ejecutar los programas del usuario. Es el encargado de crear los procesos a partir de los ejecutables de los programas y de gestionar su ejecución para evitar errores y mejorar el uso del computador.
  - Hacer que el computador sea cómodo de usar. Hace de interfaz entre el usuario y los recursos del ordenador, permitiendo el acceso tanto a ficheros y memoria como a dispositivos hardware. Esta serie de abstracciones permiten al programador acceder a los recursos hardware de forma sencilla
  - Utilizar los recursos del computador de forma eficiente. Los recursos del ordenador son compartidos tanto por los programas como por los diferentes usuarios. El sistema operativo es el encargado de repartir los recursos en función de sus políticas a aplicar.

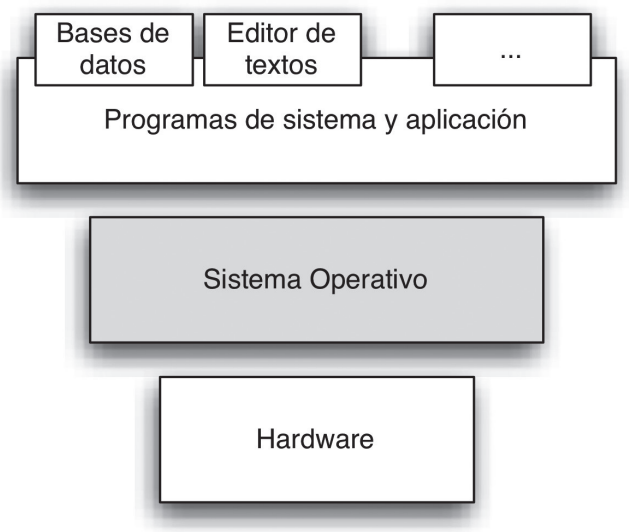


Figura 1.1. *Sistema operativo como intermediario entre aplicaciones y hardware*

- **Demonio**: proceso no interactivo que está ejecutándose continuamente en segundo plano, es decir, es un proceso controlado por el sistema sin ninguna intermediación del usuario. Suelen proporcionar un servicio básico para el resto de procesos.

¿SABÍAS QUE?

La palabra "demonio" fue usada en 1963 por primera vez en el área de la informática, para denominar a un proceso que realizaba en segundo plano *backups* en unas cintas. El nombre proviene de Fernando J. Corbató, director del proyecto MAC del MIT basándose en el famoso demonio de James Maxwell. Este demonio era un elemento biológico que residía en medio de un recipiente dividido en dos, lleno de moléculas. El demonio se encargaba de permitir, dependiendo de la velocidad de la molécula, que estas pasaran de un lado al otro. Los demonios actúan de forma similar ya que están continuamente vigilando y realizando acciones en función de las necesidades del sistema. En Windows, los demonios se denominan "servicios", estableciendo una clara relación entre los procesos no interactivos y los servicios que se estudiarán en el Capítulo 4

EJEMPLO 1.2

El proceso nulo del sistema o hasta el propio recolector de basura (*garbage collector*) de Java pueden ser considerados como demonios.

Una vez llegados a este punto, ya han sido explicados los conceptos claves para poder avanzar con la programación de procesos.

1.2 PROGRAMACIÓN CONCURRENTE

La **computación concurrente** permite la posibilidad de tener en ejecución al mismo tiempo múltiples tareas interactivas. Es decir, permite realizar varias cosas al mismo tiempo, como escuchar música, visualizar la pantalla del ordenador, imprimir documentos, etc. Pensad en todo el tiempo que perderíamos si todas esas tareas se tuvieran que realizar una tras otra. Dichas tareas se pueden ejecutar en:

- Un único procesador (multiprogramación). En este caso, 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 (del orden de milisegundos). Esto permite que en un segundo se ejecuten múltiples procesos, creando en el usuario la percepción de que múltiples programas se están ejecutando al mismo tiempo.

EJEMPLO 1.3

A la vez que se modifica un documento en Microsoft Word, se puede estar escuchando música en iTunes y navegando a través de la red con Google Chrome. Si los cambios entre los procesos se producen lo suficientemente rápido, parece que todo se ejecuta al mismo tiempo, y así la música se escucha sin cortes.

Este concepto se denomina **programación concurrente**. La programación concurrente no mejora el tiempo de ejecución global de los programas ya que se ejecutan intercambiando unos por otros en el procesador. Sin embargo, permite que varios programas parezca que se ejecuten al mismo tiempo.

- Varios núcleos en un mismo procesador (multitarea). La existencia de varios núcleos o *cores* en un ordenador es cada vez mayor, apareciendo en *Dual Cores, Quad Cores*, en muchos de los modelos *i3, i5* e *i7*, etc. Cada núcleo podría estar ejecutando una instrucción diferente al mismo tiempo. El sistema operativo, al igual que para un único procesador, se debe encargar de planificar los trabajos que se ejecutan en cada núcleo y cambiar unos por otros para generar multitarea. En este caso todos los *cores* comparten la misma memoria por lo que es posible utilizarlos de forma coordinada mediante lo que se conoce por **programación paralela**.
- La programación paralela permite mejorar el rendimiento de un programa si este se ejecuta de forma paralela en diferentes núcleos ya que permite que se ejecuten varias instrucciones a la vez. Cada ejecución en cada *core* será una tarea del mismo programa pudiendo cooperar entre sí. El concepto de "tarea" (o "hilo de ejecución") se explicará en más profundidad a lo largo del Capítulo 2. Y, por supuesto, se puede utilizar conjuntamente con la programación concurrente, permitiendo al mismo tiempo multiprogramación.

EJEMPLO 1.4

Mientras se está escribiendo un documento en Microsoft Word, al mismo tiempo se puede estar ejecutando una tarea del mismo programa que comprueba la ortografía.

- Varios ordenadores distribuidos en red. Cada uno de los ordenadores tendrá sus propios procesadores y su propia memoria. La gestión de los mismos forma parte de lo que se denomina **programación distribuida**.

  La programación distribuida posibilita la utilización de un gran número de dispositivos (ordenadores) de forma paralela, lo que permite alcanzar elevadas mejoras en el rendimiento de la ejecución de programas distribuidos. Sin embargo, como cada ordenador posee su propia memoria, imposibilita que los procesos puedan comunicarse compartiendo memoria, teniendo que utilizar otros esquemas de comunicación más complejos y costosos a través de la red que los interconecte. Se explicará en más profundidad en el Capítulo 3.

1.3 FUNCIONAMIENTO BÁSICO DEL SISTEMA OPERATIVO

La parte central que realiza la funcionalidad básica del sistema operativo se denomina *kernel*. Es una parte software pequeña del sistema operativo, si la comparamos con lo necesario para implementar su interfaz (y más hoy en día que es muy visual). A todo lo demás del sistema se le denomina **programas del sistema**. El *kernel* es el responsable de gestionar los recursos del ordenador, permitiendo su uso a través de llamadas al sistema.

En general, el *kernel* del sistema funciona en base a interrupciones. Una **interrupción** es una suspensión temporal de la ejecución de un proceso, para pasar a ejecutar una rutina que trate dicha interrupción. Esta rutina será dependiente del sistema operativo. Es importante destacar que mientras se está atendiendo una interrupción, se deshabilita la llegada de nuevas interrupciones. Cuando finaliza la rutina, se reanuda la ejecución del proceso en el mismo lugar donde se quedó cuando fue interrumpido.

Es decir, el sistema operativo no es un proceso demonio propiamente dicho que proporcione funcionalidad al resto de procesos, sino que él solo se ejecuta respondiendo a interrupciones. Cuando salta una interrupción se transfiere el control a la rutina de tratamiento de la interrupción. Así, las rutinas de tratamiento de interrupción pueden ser vistas como el código propiamente dicho del *kernel*.

Las **llamadas al sistema** son la interfaz que proporciona el *kernel* para que los programas de usuario puedan hacer uso de forma segura de determinadas partes del sistema. Los errores de un programa podrían afectar a otros programas o al propio sistema operativo, por lo que para asegurar su ejecución de la forma correcta, el sistema implementa una interfaz de llamadas para evitar que ciertas instrucciones peligrosas sean ejecutadas directamente por programas de usuario.

El **modo dual** es una característica del hardware que permite al sistema operativo protegerse. El procesador tiene dos modos de funcionamiento indicados mediante un bit:

- Modo usuario (1). Utilizado para la ejecución de programas de usuario.
- Modo *kernel* (0), también llamado "modo supervisor" o "modo privilegiado". Las instrucciones del procesador más delicadas solo se pueden ejecutar si el procesador está en modo *kernel*.

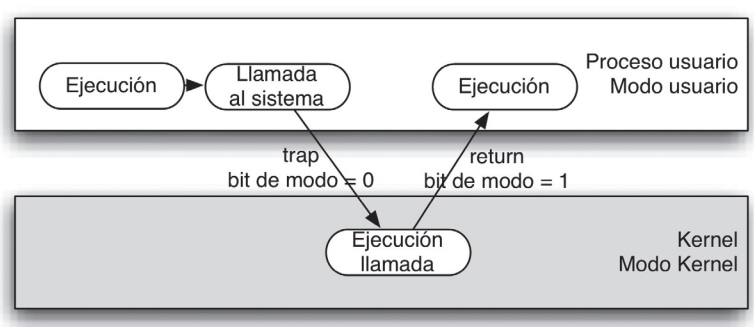


Figura 1.2. *Ejecución de llamadas al sistema*

Una llamada al sistema en un programa de usuario genera una interrupción (en este caso denominada *trap*). La interfaz de llamadas al sistema lanza la interrupción correspondiente que trata el sistema de la forma adecuada.

¿SABÍAS QUE?

Las llamadas al sistema suelen estar escritas en un lenguaje de bajo nivel (C o C++), el cual es más cercano al funcionamiento de la arquitectura del ordenador. Aun así, el programador no suele utilizar las llamadas al sistema directamente, sino que utiliza una API de más alto nivel. Las API más comunes son Win32, API para sistemas Microsoft Windows, y la API POSIX, para sistemas tipo UNIX, incluyendo GNU Linux y Mac OS.

1.4 PROCESOS

Como se ha dicho anteriormente, el sistema operativo es el encargado de poner en ejecución y gestionar los procesos. Para su correcto funcionamiento, a lo largo de su ciclo de vida, los procesos pueden cambiar de estado. Es decir, a medida que se ejecuta un proceso, dicho proceso pasará por varios estados. El cambio de estado también se producirá por la intervención del sistema operativo.

¿SABÍAS QUE?

Los primeros sistemas operativos como MS-DOS (anterior a Microsoft Windows) funcionaban con un único proceso (proceso residente). Los programas, o funcionaban en forma exclusiva en la máquina o no funcionaban. Con la creciente sofisticación de las aplicaciones y la creciente demanda de ordenadores personales, especialmente en lo relativo a aplicaciones gráficas y de red, los sistemas operativos multiproceso y multitarea se volvieron algo común.

1.4.1 ESTADO DE UN PROCESO

Los estados de un proceso son:

- **Nuevo.** El proceso está siendo creado a partir del fichero ejecutable.
- **Listo:** el proceso no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse. El planificador del sistema operativo es el responsable de seleccionar que proceso está en ejecución, por lo que es el que indica cuando el proceso pasa a ejecución.
- **En ejecución**: el proceso se está ejecutando. El sistema operativo utiliza el mecanismo de interrupciones para controlar su ejecución. Si el proceso necesita un recurso, incluyendo la realización de operaciones de entrada/

salida (E/S), llamará a la llamada del sistema correspondiente. Si un proceso en ejecución se ejecuta durante el tiempo máximo permitido por la política del sistema, salta un temporizador que lanza una interrupción. En este último caso, si el sistema es de tiempo compartido, lo para y lo pasa al estado Listo, seleccionando otro proceso para que continúe su ejecución.

- **Bloqueado:** el proceso está bloqueado esperando que ocurra algún suceso (esperando por una operación de E/S, bloqueado para sincronizarse con otros procesos, etc.). Cuando ocurre el evento que lo desbloquea, el proceso no pasa directamente a ejecución sino que tiene que ser planificado de nuevo por el sistema.
- **Terminado:** el proceso ha finalizado su ejecución y libera su imagen de memoria. Para terminar un proceso, el mismo debe llamar al sistema para indicárselo o puede ser el propio sistema el que finalice el proceso mediante una excepción (una interrupción especial).

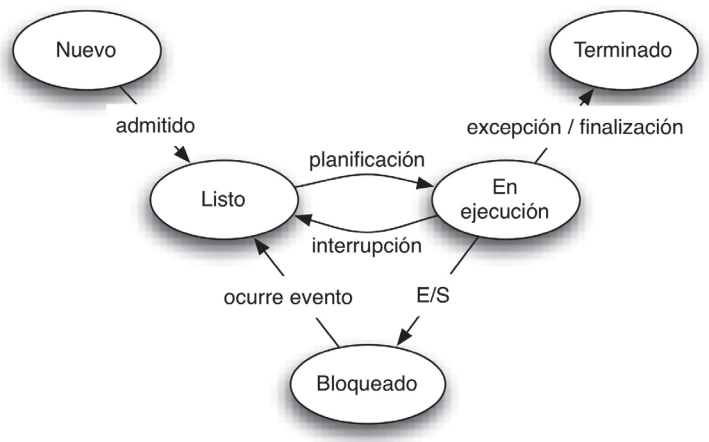


Figura 1.3. *Estados de un proceso*

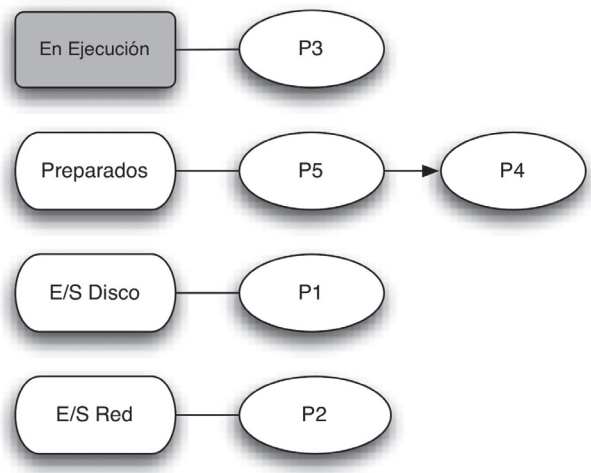
1.4.2 COLAS DE PROCESOS

Uno de los objetivos del sistema operativo es la multiprogramación, es decir, admitir varios procesos en memoria para maximizar el uso del procesador. Esto funciona ya que los procesos se irán intercambiando el uso del procesador para su ejecución de forma concurrente. Para ello, el sistema operativo organiza los procesos en varias colas, migrándolos de unas colas a otras:

- Una **cola de procesos** que contiene todos los procesos del sistema.
- Una **cola de procesos preparados** que contiene todos los procesos listos esperando para ejecutarse.
- Varias **colas de dispositivo** que contienen los procesos que están a la espera de alguna operación de E/S.

ACTIVIDADES 1.1

- Supongamos que varios procesos (1, 2, 3, 4 y 5) se están ejecutando concurrentemente en un ordenador. El proceso 1 se está ejecutando cuando realiza una petición de E/S al disco. En ese momento, el proceso 2 pasa a ejecución, haciendo otra operación de E/S, en este caso por la tarjeta de red. El proceso 3 pasa a ejecutarse. En ese momento, las colas de procesos serán las siguientes:



- En ese momento supongamos que el proceso 3 realiza una petición de E/S teniendo que esperar por el disco duro. Dibuja el estado de las colas correspondientes e identifica cuál sería el proceso que se ejecutará utilizando la filosofía FIFO para sacar los procesos de las colas.

¿SABÍAS QUE?

Una pila es una estructura de datos en la que sus elementos se almacenan de forma que el último que entra es el primero que sale. Esta forma de almacenamiento se denomina LIFO (del inglés *Last In First Out*) y se basa en dos operaciones: *push*, que introduce un elemento en la pila, y *pop,* que desapila el último elemento introducido. En este sentido, una pila puede verse como un saco. El primer elemento que se introduce va al fondo del saco y, por tanto, es el último que sale.

En cambio, una cola es una estructura de datos que sigue un almacenamiento de tipo FIFO (del inglés *First In First Out*), donde el primer elemento que se almacena es el primero que sale. La operación *push* se realiza por un extremo de la cola mientras *pop* se realiza por el otro. Esto se puede comparar con la típica cola de espera en la caja de un supermercado. El primero que llega es al primero que atienden y mientras se le atiende, el resto que va llegando se coloca al final de la cola.

1.4.3 PLANIFICACIÓN DE PROCESOS

Para gestionar las colas de procesos, es necesario un planificador de procesos. El planificador es el encargado de seleccionar los movimientos de procesos entre las diferentes colas. Existen dos tipos de planificación:

- A corto plazo: selecciona qué proceso de la cola de procesos preparados pasará a ejecución. Se invoca muy frecuentemente (del orden de milisegundos, cuando se produce un cambio de estado del proceso en ejecución) por lo que debe ser muy rápido en la decisión. Esto implica que los algoritmos sean muy sencillos:
  - Planificación sin desalojo o cooperativa. Únicamente se cambia el proceso en ejecución si dicho proceso se bloquea o termina.
  - Planificación apropiativa. Además de los casos de la planificación cooperativa, se cambia el proceso en ejecución si en cualquier momento en que un proceso se está ejecutando, otro proceso con mayor prioridad se puede ejecutar. La aparición de un proceso más prioritario se puede deber tanto al desbloqueo del mismo como a la creación de un nuevo proceso.
  - Tiempo compartido: cada cierto tiempo (llamado *cuanto*) se desaloja el proceso que estaba en ejecución y se selecciona otro proceso para ejecutarse. En este caso, todas las prioridades de los hilos se consideran iguales.
- A largo plazo: selecciona qué procesos nuevos deben pasar a la cola de procesos preparados. Se invoca con poca frecuencia, por lo que puede tomarse más tiempo en tomar la decisión. Controla el grado de multiprogramación (número de procesos en memoria).

1.4.4 CAMBIO DE CONTEXTO

Cuando el procesador pasa a ejecutar otro proceso, lo cual ocurre muy frecuentemente, el sistema operativo debe guardar el contexto del proceso actual y restaurar el contexto del proceso que el planificador a corto plazo ha elegido ejecutar. La salvaguarda de la información del proceso en ejecución se produce cuando hay una interrupción.

Se conoce como **contexto** a:

- Estado del proceso.
- Estado del procesador: valores de los diferentes registros del procesador.
- Información de gestión de memoria: espacio de memoria reservada para el proceso.

El cambio de contexto es tiempo perdido, ya que el procesador no hace trabajo útil durante ese tiempo. Únicamente es tiempo necesario para permitir la multiprogramación y su duración depende de la arquitectura en concreto del procesador.

1.5 GESTIÓN DE PROCESOS

1.5.1 ÁRBOL DE PROCESOS

El sistema operativo es el encargado de crear y gestionar los nuevos procesos siguiendo las directrices del usuario. Así, cuando un usuario quiere abrir un programa, el sistema operativo es el responsable de crear y poner en ejecución el proceso correspondiente que lo ejecutará. Aunque el responsable del proceso de creación es el sistema operativo, ya que es el único que puede acceder a los recursos del ordenador, el nuevo proceso se crea siempre por petición de otro proceso. La puesta en ejecución de un nuevo proceso se produce debido a que hay un proceso en concreto que está pidiendo su creación en su nombre o en nombre del usuario.

EJEMPLO 1.5

Si un usuario pincha en el icono de Adobe Photoshop para arrancar una instancia del mismo, la interfaz gráfica es la que hace la petición para crear el nuevo proceso de Photoshop.

En este sentido, cualquier proceso en ejecución siempre depende del proceso que lo creó, estableciéndose un vínculo entre ambos. A su vez, el nuevo proceso puede crear nuevos procesos, formándose lo que se denomina un **árbol de procesos**. Cuando se arranca el ordenador, y se carga en memoria el *kernel* del sistema a partir de su imagen en disco, se crea el proceso inicial del sistema. A partir de este proceso, se crea el resto de procesos de forma jerárquica, estableciendo padres, hijos, abuelos, etc.

Para identificar a los procesos, los sistemas operativos suelen utilizar un **identificador de proceso** (*process identifier* [PID]) unívoco para cada proceso. La utilización del PID es básica a la hora de gestionar procesos, ya que es la forma que tiene el sistema de referirse a los procesos que gestiona.

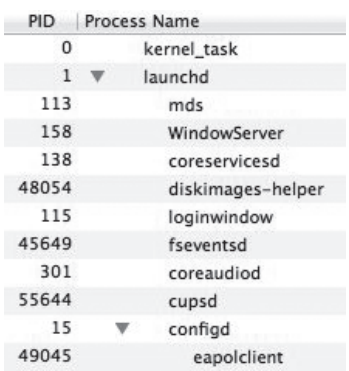


Figura 1.4. *Árbol de procesos de Mac OS X. Se puede observar cómo el proceso principal del cual cuelgan el resto de procesos es el proceso con PID 0*

ACTIVIDADES 1.2

- El Administrador de tareas en Microsoft Windows —llamado "Monitor de actividad" en Mac OS y "Monitor del sistema" en Ubuntu Linux— permite gestionar los procesos del sistema operativo. En él se puede ver qué procesos se están ejecutando con su uso de procesador y memoria.
- Utiliza el Administrador de tareas para obtener los procesos del sistema.
- Dibuja el árbol de procesos correspondiente para tu propio ordenador en ese momento.

1.5.2 OPERACIONES BÁSICAS CON PROCESOS

Siguiendo el vínculo entre procesos establecido en el árbol de procesos, el proceso creador se denomina **padre** y el proceso creado se denomina **hijo**. A su vez, los hijos pueden crear nuevos hijos. A la operación de creación de un nuevo proceso la denominaremos *create*.

Cuando se crea un nuevo proceso tenemos que saber que padre e hijo se ejecutan concurrentemente. Ambos procesos comparten la CPU y se irán intercambiando siguiendo la política de planificación del sistema operativo para proporcionar multiprogramación. Si el proceso padre necesita esperar hasta que el hijo termine su ejecución para poder continuar la suya con los resultados obtenidos por el hijo, puede hacerlo mediante la operación *wait*.

Como se ha visto al inicio del capítulo, los procesos son independientes y tienen su propio espacio de memoria asignado, llamado **imagen de memoria**. Padres e hijos son procesos y, aunque tengan un vínculo especial, mantienen esta restricción. Ambos usan espacios de memoria independientes. En general, parece que el hijo ejecuta un programa diferente al padre, pero en algunos sistemas operativos esto no tiene por qué ser así. Por ejemplo, mientras que en sistemas tipo Windows existe una función *createProcess()* que crea un nuevo proceso a partir de un programa distinto al que está en ejecución, en sistemas tipo UNIX, la operación a utilizar es *fork()*, que crea un proceso hijo con un duplicado del espacio de direcciones del padre, es decir, un duplicado del programa que se ejecuta desde la misma posición. Sin embargo, en ambos casos, los padres e hijos (aunque sean un duplicado en el momento de la creación en sistemas tipos UNIX) son independientes y las modificaciones que uno haga en su espacio de memoria, como escritura de variables, no afectarán al otro.

Como padre e hijo tienen espacios de memoria independientes, pueden compartir recursos para intercambiarse información. Estos recursos pueden ir desde ficheros abiertos hasta zonas de memoria compartida. La **memoria compartida** es una región de memoria a la que pueden acceder varios procesos cooperativos para compartir información. Los procesos se comunican escribiendo y leyendo datos en dicha región. El sistema operativo solamente interviene a la hora de crear y establecer los permisos de qué procesos pueden acceder a dicha zona. Los procesos son los responsables del formato de los datos compartidos y de su ubicación.

Al terminar la ejecución de un proceso, es necesario avisar al sistema operativo de su terminación para que de esta forma el sistema libere si es posible los recursos que tenga asignados. En general, es el propio proceso el que le indica al sistema operativo mediante una operación denominada *exit* que quiere terminar, pudiendo aprovechar para mandar información respecto a su finalización al proceso padre en ese momento.

El proceso hijo depende tanto del sistema operativo como del proceso padre que lo creó. Así, el padre puede terminar la ejecución de un proceso hijo cuando crea conveniente. Entre estos motivos podría darse que el hijo excediera el uso de algunos recursos o que la funcionalidad asignada al hijo ya no sea necesaria por algún motivo.

Para ello puede utilizar la operación *destroy*. Esta relación de dependencia entre padre e hijo, lleva a casos como que si el padre termina, en algunos sistemas operativos no se permita que sus hijos continúen la ejecución, produciéndose lo que se denomina "terminación en cascada".

En definitiva, cada sistema operativo tiene unas características únicas, y la gestión de los procesos es diferente en cada uno de ellos. Por simplificación y portabilidad, evitando así depender del sistema operativo sobre el cual se esté ejecutando hemos decidido explicar la gestión de procesos para la **máquina virtual de Java** (*Java Virtual Machine* [JVM]). JVM es un entorno de ejecución ligero y gratuito multiplataforma que permite la ejecución de binarios o *bytecode* del lenguaje de programación Java sobre cualquier sistema operativo, salvando las diferencias entre ellos. La idea que hay detrás de ello se conoce como *Write Once, Run Anywhere* ("escribe una vez y ejecuta en cualquier lugar", entendiendo "cualquier lugar" como plataforma o sistema operativo). Las próximas secciones del capítulo estarán basadas por tanto en la utilización de Java.

¿SABÍAS QUE?

Java es un lenguaje de programación orientado a objetos concurrente, de propósito general y multiplataforma creado por Sun Microsystems. Es un lenguaje de alto nivel, es decir, sus instrucciones son cercanas al lenguaje natural, facilitando la comprensión del código al programador. Sin embargo, sus instrucciones son lejanas al hardware no pudiendo aprovecharse de la arquitectura física específica subyacente por lo que ofrece peor rendimiento y tiene menores capacidades que lenguajes de bajo nivel como C. Las aplicaciones de Java son traducidas a *bytecode*, el cual se puede ejecutar en cualquier JVM sin importar la arquitectura del ordenador donde está corriendo. Esto permite su ejecución sin tener que volver a compilar en diferentes plataformas. Se puede encontrar más información sobre Java en la página: *http://www.java.com/es/about/*

Respecto a la utilización de Java, tenemos que saber que los procesos padre e hijo en la JVM no tienen por qué ejecutarse de forma concurrente. Además, no se produce terminación en cascada, pudiendo sobrevivir los hijos a su padre ejecutándose de forma asíncrona. Por último, hay que tener en cuenta que puede no funcionar bien para procesos especiales en ciertas plataformas nativas (por ejemplo, utilización de ventanas nativas en MS-DOS/Windows, *shell scripts* en GNU Linux/UNIX/Mac OS, etc.).

1.5.2.1 Creación de procesos (operación create)

La clase que representa un proceso en Java es la clase *Process*. Los métodos de *ProcessBuilder.start()* y *Runtime.exec()* crean un proceso nativo en el sistema operativo subyacente donde se está ejecutando la JVM y devuelven una objeto Java de la clase *Process* que puede ser utilizado para controlar dicho proceso.

- **Process ProcessBuilder.start**(): inicia un nuevo proceso utilizando los atributos indicados en el objeto. El nuevo proceso ejecuta el comando y los argumentos indicados en el método **command**(), ejecutándose en el directorio de trabajo especificado por **directory**(), utilizando las variables de entono definidas en **environment**().
- **Process Runtime.exec**(**String**[] **cmdarray**, **String**[] **envp**, **File dir**): ejecuta el comando especificado y argumentos en *cmdarray* en un proceso hijo independiente con el entorno *envp* y el directorio de trabajo especificado en *dir*.

Ambos métodos comprueban que el comando a ejecutar es un comando o ejecutable válido en el sistema operativo subyacente sobre el que ejecuta la JVM. El ejecutable se ha podido obtener mediante la compilación de código en cualquier lenguaje de programación. Al final, crear un nuevo proceso depende del sistema operativo en concreto que esté ejecutando por debajo de la JVM. En este sentido, pueden ocurrir múltiples problemas, como:

- ✓ No encuentra el ejecutable debido a la ruta indicada.
- ✓ No tener permisos de ejecución.
- ✓ No ser un ejecutable válido en el sistema.
- ✓ etc.

En la mayoría de los casos, se lanza una excepción dependiente del sistema en concreto, pero siempre será una subclase de *IOException*.

EJEMPLO 1.6

Creación de un proceso utilizando *ProcessBuilder*

```
import java.io.IOException;
import java.util.Arrays;

public class RunProcess {

    public static void main(String[] args) throws IOException {

     if (args.length <= 0) {
          System.err.println("Se necesita un programa a ejecutar");
               System.exit(-1);
     }

    ProcessBuilder pb = new ProcessBuilder(args);
    try {
          Process process = pb.start();
          int retorno = process.waitFor();
          System.out.println("La ejecución de " +
                    Arrays.toString(args) + " devuelve " +  retorno);
     }catch(IOException ex){
           System.err.println("Excepción de E/S!!");
                  System.exit(-1);
    }catch(InterruptedException ex){
          System.err.println("El proceso hijo finalizó
                    de forma incorrecta");
          System.exit(-1);
    }
    }
}
```

1.5.2.2 Terminación de procesos (operación destroy)

Un proceso puede terminar de forma abrupta un proceso hijo que creó. Para ello el proceso padre puede ejecutar la operación *destroy*. Esta operación elimina el proceso hijo indicado liberando sus recursos en el sistema operativo subyacente. En caso de Java, los recursos correspondientes los eliminará el *garbage collector* cuando considere.

¿SABÍAS QUE?

El Administrador de tareas también permite eliminar procesos, siendo el propio sistema operativo en nombre del usuario (el cual puede convertirse en administrador) el encargado de ejecutar la operación *destroy*.

Si no se fuerza la finalización de la ejecución del proceso hijo de forma anómala, el proceso hijo realizará su ejecución completa terminando y liberando sus recursos al finalizar. Esto se produce cuando el hijo realiza la operación *exit* para finalizar su ejecución.

EJEMPLO 1.7

Creación de un proceso mediante *Runtime* para después destruirlo.

```
import java.io.IOException;

public class RuntimeProcess {

    public static void main(String[] args) {

        if (args.length <= 0) {
                System.err.println("Se necesita un programa a ejecutar");
                System.exit(-1);
        }

        Runtime runtime = Runtime.getRuntime();
        try {
             Process process = runtime.exec(args);
             process.destroy();
        }catch(IOException ex){
                System.err.println("Excepción de E/S!!");
                System.exit(-1);
        }
        }
}
```

1.6 COMUNICACIÓN DE PROCESOS

Es importante recordar que un proceso es un programa en ejecución y, como cualquier programa, recibe información, la transforma y produce resultados. Esta acción se gestiona a través de:

- La **entrada estándar** (*stdin*): lugar de donde el proceso lee los datos de entrada que requiere para su ejecución. No se refiere a los parámetros de ejecución del programa. Normalmente suele ser el teclado, pero podría recibirlos de un fichero, de la tarjeta de red o hasta de otro proceso, entre otros sitios. La lectura de datos a lo largo de un programa (por ejemplo mediante *scanf* en C) leerá los datos de su entrada estándar.
- La **salida estándar** (*stdout*): sitio donde el proceso escribe los resultados que obtiene. Normalmente es la pantalla, aunque podría ser, entre otros, la impresora o hasta otro proceso que necesite esos datos como entrada. La escritura de datos que se realice en un programa (por ejemplo mediante *printf* en C o *System.out.println* en Java) se produce por la salida estándar.
- La **salida de error** (*stderr*): sitio donde el proceso envía los mensajes de error. Habitualmente es el mismo que la salida estándar, pero puede especificarse que sea otro lugar, por ejemplo un fichero para identificar más fácilmente los errores que ocurren durante la ejecución.

La utilización de *System.out* y *System.err* en Java se puede ver como un ejemplo de utilización de estas salidas.

En la mayoría de los sistemas operativos, estas entradas y salidas en proceso hijo son una copia de las mismas entradas y salidas que tuviera su padre. De tal forma que si se llama a la operación *create* dentro de un proceso que lee de un fichero y muestra la salida estándar por pantalla, su hijo correspondiente leerá del mismo fichero y escribirá en pantalla. En Java, en cambio, el proceso hijo creado de la clase *Process* no tiene su propia interfaz de comunicación, por lo que el usuario no puede comunicarse con él directamente. Todas sus salidas y entradas de información (*stdin*, *stdout* y *stderr*) se redirigen al proceso padre a través de los siguientes flujos de datos o *streams:*

- ***OutputStream*:** flujo de salida del proceso hijo. El *stream* está conectado por un *pipe* a la entrada estándar (*stdin*) del proceso hijo.
- ***InputStream*:** flujo de entrada del proceso hijo. El *stream* está conectado por un *pipe* a la salida estándar (*stdout*) del proceso hijo.
- ***ErrorStream*:** flujo de error del proceso hijo. El *stream* está conectado por un *pipe* a la salida estándar (*stderr*) del proceso hijo. Sin embargo, hay que saber que, por defecto, para la JVM, *stderr* está conectado al mismo sitio que *stdout*.

  Si se desea tenerlos separados, lo que permite identificar errores de forma más sencilla, se puede utilizar el método *redirectErrorStream(boolean)* de la clase *ProcessBuilder*. Si se pasa un valor *true* como parámetro, los flujos de datos correspondientes a *stderr* y *stdout* en la JVM serán diferentes y representarán la salida estándar y la salida de error del proceso de forma correspondiente.

Utilizando estos *streams*, el proceso padre puede enviarle datos al proceso hijo y recibir los resultados de salida que este genere comprobando los errores.

Hay que tener en cuenta que en algunos sistemas operativos, el tamaño de los *buffers* de entrada y salida que corresponde a *stdin* y *stdout* está limitado. En este sentido, un fallo al leer o escribir en los flujos de entrada o salida del proceso hijo puede provocar que el proceso hijo se bloquee. Por eso, en Java se suele realizar la comunicación padre-hijo a través de un *buffer* utilizando los *streams* vistos.

EJEMPLO 1.8

Comunicación de procesos utilizando un *buffer*

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Arrays;

public class ComunicationBetweenProcess {

    public static void main(String args[]) throws IOException {

            Process process = new ProcessBuilder(args).start();
        InputStream is = process.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String line;

        System.out.println("Salida del proceso " +
                  Arrays.toString(args) + ":");

        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }

    }
}
```

Por último, hay que tener en cuenta cómo se está codificando la información que se envía y se recibe entre los diferentes procesos. La codificación de la información depende del sistema operativo subyacente donde se está ejecutando el proceso hijo.

¿SABÍAS QUE?

GNU Linux, Mac OS, Android, UNIX, iOS, etc. utilizan el formato de codificación de caracteres UTF-8, que utiliza por debajo el estándar Unicode. Unicode es un estándar de codificación de caracteres diseñado para facilitar el tratamiento informático, transmisión y visualización de textos de múltiples lenguajes y disciplinas técnicas. El término "Unicode" proviene de los tres objetivos perseguidos: universalidad, uniformidad y unicidad.

Sin embargo, diferentes versiones de Microsoft Windows no utilizan UTF-8, sino sus propios formatos de codificación no compatibles con el resto, como Windows-Western.

Para mostrar los datos correctamente codificados en Java puede ser necesario especificar cómo se reciben los datos en el sistema operativo subyacente, ya que es en el que se está ejecutando los procesos hijos, con los que hay que comunicarse.

ACTIVIDADES 1.3

Un proceso puede esperar recibir por su entrada estándar los datos con los que operar en un formato específico. Por ejemplo, si el proceso se crea a partir de un ejecutable en Unix, la comunicación de datos con el mismo debería producirse en UTF-8. Si los datos de entrada no contienen caracteres extraños (saltos de línea, tildes, *ñ*, etc.), esto no suele ser necesario, pero aun así veremos cómo puede hacerse.

```
//Importamos todos los paquetes necesarios
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Arrays;

public class UnixInteractor {

   public static void main(String[] command) {

          String line;

                  //Preparamos el commando a ejecutar
        ProcessBuilder pb = new ProcessBuilder(command);
        pb.redirectErrorStream(true);
        try {
                        //Se crea el Nuevo proceso hijo
     Process shell = pb.start();
                        //Se obtiene stdout del proceso hijo
     InputStream is = shell.getInputStream();
                        //Se convierte el formato de UTF-8 al de un String de Java
     BufferedReader br = new BufferedReader (new
                            InputStreamReader(is,"UTF-8"));

              System.out.println("La salida del proceso
                                  hijo" + Arrays.toString(command) + ":" );
                        //Se muestra la salida del hijo por pantalla
              while ((line = br.readLine()) != null) {
                 System.out.println(line);
              }
                        //Cuando finaliza se cierra el descriptor
                        //de salida del hijo
                 is.close();
                    } catch (IOException e) {
                 System.out.println("Error ocurrió ejecutando el comando.
Descripción:" + e.getMessage());
      }
   }
}
```

» Modifica el ejemplo visto para que la comunicación se realice de forma efectiva con un ejecutable de Microsoft Windows. Justifica cualquier cambio realizado incluyendo los paquetes a utilizar.

Además de la posibilidad de comunicarse mediante flujos de datos, existen otras alternativas para la comunicación de procesos:

- Usando *sockets* para la comunicación entre procesos. Este concepto se estudiará en detalle en el Capítulo 3.
- Utilizando JNI (*Java Native Interface*). La utilización de Java permite abstraerse del comportamiento final de los procesos en los diferentes sistemas operativos. Sin embargo, al ser un lenguaje de alto nivel oculta funcionalidad de procesos de bajo nivel que puede depender del sistema subyacente. Por ejemplo, un *pipe* es un canal de comunicación unidireccional de bajo nivel que permite comunicarse a los procesos de forma sencilla. La comunicación de procesos en Java solo se puede hacer mediante los *streams* explicados, perdiendo parte de la funcionalidad ofertada por los *pipes* de bajo nivel. Sin embargo, se puede utilizar JNI para acceder desde Java a aplicaciones desarrolladas en otros lenguajes de programación de más bajo nivel, como C, que pueden sacar partido al sistema operativo subyacente.

¿SABÍAS QUE?

La mayoría de los sistemas operativos actuales están desarrollados en C. C es un lenguaje de bajo nivel cercano a la arquitectura que obtiene un elevado rendimiento. Al ser más cercano a la arquitectura del sistema, permite comunicar diferentes procesos utilizando métodos más cercanos al sistema operativo.

- Librerías de comunicación no estándares entre procesos en Java que permiten aumentar las capacidades del estándar Java para comunicarlos. Por ejemplo, CLIPC (*http://clipc.sourceforge.net/*) es una librería Java de código abierto que ofrece la posibilidad de utilizar los siguientes mecanismos que no están incluidos directamente en el lenguaje gracias a que utiliza por debajo llamadas a JNI para poder utilizar métodos más cercanos al sistema operativo:
  - Memoria compartida: se establece una región de memoria compartida entre varios procesos a la cual pueden acceder todos ellos. Los procesos se comunican escribiendo y leyendo datos en ese espacio de memoria compartida.
  - *Pipes*: permite a un proceso hijo comunicarse con su padre a través de un canal de comunicación sencillo.
  - Semáforos: mecanismo de bloqueo de un proceso hasta que ocurra un evento.

  Sin embargo, aunque dichas librerías pueden aumentar los métodos de comunicación entre procesos, hay que tener en cuenta que se encuentran actualmente en fase de investigación y desarrollo.

ACTIVIDADES 1.4

- Busca en Internet qué es JNI (*Java Native Interface*) y descubre cómo se puede programar una aplicación Java que llame a código desarrollado en otro lenguaje de programación. Gracias a JNI, se le puede sacar una mayor funcionalidad a los procesos al poder utilizar funcionalidad dependiente del sistema subyacente.

1.7 SINCRONIZACIÓN DE PROCESOS

Los métodos de comunicación de procesos se pueden considerar como métodos de sincronización ya que permiten al proceso padre llevar el ritmo de envío y recepción de mensajes.

Concretamente, los envíos y recepciones por los flujos de datos permiten a un proceso hijo comunicarse con su padre a través de un canal de comunicación unidireccional bloqueante.

EJEMPLO 1.9

Si el padre pide leer de la salida del hijo *stdout* a través de su *InputStream* se bloquea hasta que el hijo le devuelve los datos requeridos. En este sentido, padre e hijo pueden sincronizarse de la forma adecuada.

1.7.1 ESPERA DE PROCESOS (OPERACIÓN WAIT)

Además de la utilización de los flujos de datos se puede esperar por la finalización del proceso hijo y obtener su información de finalización mediante la operación *wait*. Dicha operación bloquea al proceso padre hasta que el hijo finaliza su ejecución mediante *exit*. Como resultado se recibe la información de finalización del proceso hijo. Dicho valor de retorno se especifica mediante un número entero. El valor de retorno significa cómo resulto la ejecución. No tiene nada que ver con los mensajes que se pasan padre e hijo a través de los *streams*. Por convención se utiliza 0 para indicar que el hijo ha acabado de forma correcta.

Mediante *waitFor*() de la clase *Process* el padre espera bloqueado hasta que el hijo finalice su ejecución, volviendo inmediatamente si el hijo ha finalizado con anterioridad o si alguien le interrumpe (en este caso se lanza la interrupción *InterruptedException*). Además se puede utilizar *exitValue*() para obtener el valor de retorno que devolvió un proceso hijo. El proceso hijo debe haber finalizado, si no, se lanza la excepción *IllegalThreadStateException*

EJEMPLO 1.10

Implementación de sincronización de procesos

```
import java.io.IOException;
import java.util.Arrays;

public class ProcessSincronization {

    public static void main(String[] args)
          throws IOException, InterruptedException {

    try{
          Process process = new ProcessBuilder(args).start();
          int retorno = process.waitFor();
          System.out.println("Comando " +  Arrays.toString(args)
               +  " devolvió: " + retorno);
    } catch (IOException e) {
          System.out.println("Error ocurrió ejecutando el
                              comando. Descripción: " + e.getMessage());
    } catch (InterruptedException e) {
          System.out.println("El comando fue interrumpido.
                              Descripción del error: " + e.getMessage());
    }
    }
}
```

1.8 PROGRAMACIÓN MULTIPROCESO

La programación concurrente es una forma eficaz de procesar la información al permitir que diferentes sucesos o procesos se vayan alternando en la CPU para proporcionar multiprogramación. La multiprogramación puede producirse entre procesos totalmente independientes, como podrían ser los correspondientes al procesador de textos, navegador, reproductor de música, etc., o entre procesos que pueden cooperar entre sí para realizar una tarea común.

El sistema operativo se encarga de proporcionar multiprogramación entre todos los procesos del sistema, ocultando esta complejidad tanto a los usuarios como a los desarrolladores. Sin embargo, si se pretende realizar procesos que cooperen entre sí, debe ser el propio desarrollador quien lo implemente utilizando la comunicación y sincronización de procesos vistas hasta ahora.

A la hora de realizar un programa multiproceso cooperativo, se deben seguir las siguientes fases:

1 Descomposición funcional. Es necesario identificar previamente las diferentes funciones que debe realizar la aplicación y las relaciones existentes entre ellas.

2 Partición. Distribución de las diferentes funciones en procesos estableciendo el esquema de comunicación entre los mismos. Al ser procesos cooperativos necesitarán información unos de otros por lo que deben comunicarse. Hay que tener en cuenta que la comunicación entre procesos requiere una pérdida de tiempo tanto de comunicación como de sincronización. Por tanto, el objetivo debe ser maximizar la independencia entre los procesos minimizando la comunicación entre los mismos.

3 Implementación. Una vez realizada la descomposición y la partición se realiza la implementación utilizando las herramientas disponibles por la plataforma a utilizar. En este caso, Java permite únicamente métodos sencillos de comunicación y sincronización de procesos para realizar la cooperación.

1.8.1 CLASE PROCESS

A la hora de realizar un algoritmo multiproceso en Java se utiliza la clase *Process*. Se presenta por simplicidad una tabla con los métodos recogidos más importantes explicados a lo largo del capítulo:

Método	Tipo de retorno	Descripción
getOutputStream()	OutputStream	Obtiene el flujo de salida del proceso hijo conectado al *stdin*
getInputStream()	InputStream	Obtiene el flujo de entrada del proceso hijo conectado al *stdout* del proceso hijo
getErrorStream()	InputStream	Obtiene el flujo de entrada del proceso hijo conectado al *stderr* del proceso hijo
destroy()	void	Implementa la operación *destroy*
waitFor()	int	Implementa la operación *wait*
exitValue()	int	Obtiene el valor de retorno del proceso hijo

1.9 CASO PRÁCTICO

En este caso práctico se va a desarrollar una solución multiproceso al problema de sincronizar y comunicar dos procesos hijos creados a partir de un proceso padre. La idea es escribir una clase Java que ejecute dos comandos (cada hijo creado ejecutará uno de ellos) con sus respectivos argumentos y redireccione la salida estándar del primero a la entrada estándar del segundo. Por sencillez, los comandos y sus argumentos irán directamente escritos en el código del programa para no complicar demasiado el problema.

El siguiente ejemplo muestra la ejecución de los comandos **ls –la** y **tr “d” “D”** en Unix (el resultado debería ser el mismo que el de ejecutar en la *shell* de Linux o Mac **ls –la | tr “d” “D”**):

```
total 6
Drwxr-xr-x 5 user users 4096 2011-02-22 10:59 .
Drwxr-xr-x 8 user users 4096 2011-02-07 09:26 ..
-rw-r--r-- 1 user users 30 2011-02-22 10:59 a.txt
-rw-r--r-- 1 user users 27 2011-02-22 10:59 b.txt
Drwxr-xr-x 2 user users 4096 2011-01-24 17:49 Dir3
Drwxr-xr-x 2 user users 4096 2011-01-24 11:48 Dir4
```

RESUMEN DEL CAPÍTULO

El capítulo ha mostrado el funcionamiento de los sistemas operativos en lo relativo a la ejecución de diferentes programas. Para ello, se ha definido el concepto de proceso, viéndolo como un programa en ejecución con toda la estructura necesaria que necesite para ello. El sistema operativo tiene que gestionar múltiples procesos. Aunque solo haya un único procesador, permitiendo la ejecución de un único proceso, el sistema operativo se encarga de aparentar que varios procesos se ejecutan a la vez (concepto conocido como "concurrencia" y en este caso en concreto, "multiprogramación") cambiando la ejecución del proceso lo suficientemente rápido. Estos cambios generan los estados de un proceso. Un proceso a lo largo de su ciclo de vida, puede pasar por diferentes estados —listo, en ejecución o bloqueado, entre otros—, saltando de una cola de espera a otra cuando no esté en ejecución. En función de cómo se gestionen las colas de espera de procesos, el funcionamiento del sistema variará. Esto se conoce como la "planificación del sistema", pudiendo ser apropiativa (desalojando los procesos más prioritarios a los menos prioritarios), cooperativa (sin desalojo, o de tiempo compartido [cambiando de proceso cada cierto tiempo]). Cuando un proceso pasa de estar en ejecución a otro estado se produce lo que se denomina un "cambio de contexto". En ese momento hay que salvar toda la información del proceso para poder ponerlo a ejecutarse desde el mismo punto donde lo dejó cuando fue desalojado.

El sistema operativo es el encargado de gestionar los procesos. Para ello se comunica con los procesos utilizando su identificador o PID, número unívoco que identifica a un proceso. Un proceso a su vez puede crear otros procesos. El proceso creador se denomina "padre", y el nuevo proceso, "hijo". Esto produce que se cree un árbol de procesos del sistema ya que al arrancar el ordenador solo existe un proceso, y el resto de procesos se van creando en forma arbórea partiendo de él. La creación de procesos se realiza mediante la operación *create,* permitiendo al proceso padre controlar a sus hijos. Puede destruirlos mediante la operación *destroy* o esperar por su finalización mediante *wait*. La comunicación de procesos se realiza a través de su interfaz con el mundo real. Dicha interfaz está formada por su entrada estándar (*stdin*) de donde recibe datos de entrada; la salida estándar (*stdout*), donde el proceso escribe los resultados; y la salida de error (*stderr*), donde envía los mensajes de error. La utilización de la interfaz, especialmente la recepción de datos por *stdin* es bloqueante, lo que permite junto a las otras operaciones sincronizar procesos para realizar operaciones multiproceso.

Cada sistema operativo tiene unas características únicas, y la gestión de los procesos es diferente en cada uno de ellos. Por interoperabilidad se ha visto cómo se gestionan procesos utilizando la JVM de Java, aunque su utilización oculta algunas de las posibilidades existentes para sincronizar y comunicar procesos a bajo nivel.

EJERCICIOS PROPUESTOS

1. Escribe una clase llamada *Ejecuta* que reciba como argumentos el comando y las opciones del comando que se quiere ejecutar. El programa debe crear un proceso hijo que ejecute el comando con las opciones correspondientes mostrando un mensaje de error en el caso de que no se realizase correctamente la ejecución. El padre debe esperar a que el hijo termine de informar si se produjo alguna anomalía en la ejecución del hijo.

2. Escribe un programa *Aleatorios* que haga lo siguiente:

- Cree un proceso hijo que está encargado de generar números aleatorios. Para su creación puede utilizarse cualquier lenguaje de programación generando el ejecutable correspondiente. Este proceso hijo escribirá en su salida estándar un número aleatorio del 0 al 10 cada vez que reciba una petición de ejecución por parte del padre. *Nota*: no es necesario utilizar JNI, solamente crear un ejecutable y llamar correctamente al mismo desde Java.
- El proceso padre lee líneas de la entrada estándar y por cada línea que lea solicitará al hijo que le envíe un número aleatorio, lo leerá y lo imprimirá en pantalla.
- Cuando el proceso padre reciba la palabra "fin", finalizará la ejecución del hijo y procederá a finalizar su ejecución.

Ejemplo de ejecución:

```
ab (enter)
7
abcdef (enter)
1
Pepe (enter)
6
fin (enter)
```

3. Escribe una clase llamada *Mayusculas* que haga lo siguiente:

- Cree un proceso hijo.
- El proceso padre y el proceso hijo se comunicarán de forma bidireccional utilizando *streams*.
- El proceso padre leerá líneas de su entrada estándar y las enviará a la entrada estándar del hijo (utilizando el *OutputStream* del hijo).
- El proceso hijo leerá el texto por su entrada estándar, lo transformará todo a letras mayúsculas y lo imprimirá por su salida estándar. Para realizar el programa hijo se puede utilizar cualquier lenguaje de programación generando un ejecutable.
- El padre imprimirá en pantalla lo que recibe del hijo a través del *InputStream* del mismo.

Ejemplo de ejecución:

```
hola (enter)
HOLA
mundo (enter)
MUNDO
```

TEST DE CONOCIMIENTOS

1 ¿Qué proporciona el modo dual?

a) Un mecanismo para ejecutar dos instrucciones en paralelo si se tienen varios núcleos.
b) Un mecanismo de protección que evita que determinadas instrucciones puedan ser ejecutadas directamente por código de usuario.
c) Un mecanismo que permite la ejecución del *kernel* del sistema operativo únicamente como respuesta a un *trap*.
d) Un mecanismo que permite la ejecución del *kernel* del sistema operativo únicamente como respuesta a una interrupción hardware.

2 Indica cuál de las siguientes respuestas es falsa:

a) Los procesos son independientes y tienen su propio espacio de memoria asignado.
b) El sistema operativo se refiere a los procesos que gestiona mediante su PID.
c) Dos procesos diferentes pueden tener el mismo PID.
d) La puesta en ejecución de un nuevo proceso se produce bajo la responsabilidad de otro proceso.

3 El *kernel* del sistema operativo se ejecuta:

a) Como un proceso más dentro del sistema atendiendo las peticiones que los procesos y el hardware le hacen.
b) En base a interrupciones.
c) En base a rutina de tratamiento de señales.
d) Utilizando *pipes* para comunicarse con los diferentes procesos.

4 Una interrupción software causada por una petición del usuario es:

a) Una excepción.
b) Una interrupción.
c) Un *trap*.
d) Un *fork*.

5 Si un proceso está en el estado de "En ejecución", y se produce una interrupción:

a) Pasará el estado "Listo".
b) Pasará al estado "Bloqueado".
c) Seguirá en ejecución.
d) Terminará.

6 Si un proceso está en el estado de "En ejecución" y solicita una operación de entrada/salida:

a) Pasará el estado "Listo".
b) Pasará al estado "Bloqueado".
c) Seguirá en ejecución.
d) Terminará.

7 Indica cuál de las siguientes afirmaciones es FALSA:

a) El cambio de contexto es tiempo perdido, durante el cual el procesador no ejecuta instrucciones de los procesos.
b) No importa que el tiempo de cambio de contexto sea grande, ya que es una operación que no se realiza muy frecuentemente.
c) El tiempo de cambio de contexto depende de la arquitectura del procesador.
d) El contexto de un proceso incluye el estado del proceso, el estado del procesador y la información de gestión de memoria.

8 Un planificador que solo toma decisiones cuando un proceso pasa de ejecución a espera, o cuando un proceso termina, es un planificador:

a) Apropiativo.
b) Cooperativo.
c) Perezoso.
d) Automático.

9 En la planificación por tiempo compartido:

a) De forma secuencial cada proceso preparado pasa a ejecución durante una cota de tiempo llamada "cuanto".
b) El proceso preparado que pasa a ejecución corresponde al de tiempo de ejecución restante más corto.
c) De acuerdo a su prioridad, cada proceso preparado pasa a ejecución durante una cota de tiempo llamada "cuanto".
d) El proceso preparado que pasa a ejecución corresponde al de mayor prioridad asignada.

10 Dado el siguiente fragmento de código:

```
Process p1 = new ProcessBuilder(args).start();
Process p2 = new ProcessBuilder(args1).start();
BufferedReader br1 = new BufferedReader(new
  InputStreamReader(p1.getInputStream()));
BufferedReader br2 = new BufferedReader(new
  InputStreamReader(p2.getInputStream()));
```

Indica el máximo número de procesos que pueden comunicarse entre sí utilizando br1 y br2.

a) 3
b) 2
c) 4
d) 0