Programación de procesos

Tabla de contenido

1. Conceptos básicos

Programa: conjunto comprendido por código y datos que se encuentra en el disco que resuelven una necesidad concreta de los usuarios.

Proceso: simplificando se podría decir que es cuando un programa se ejecuta o un programa en ejecución, aunque no es una definición muy acertada. Concretando más, un proceso es el estado de un programa en un momento concreto de su ejecución, lo que incluye:

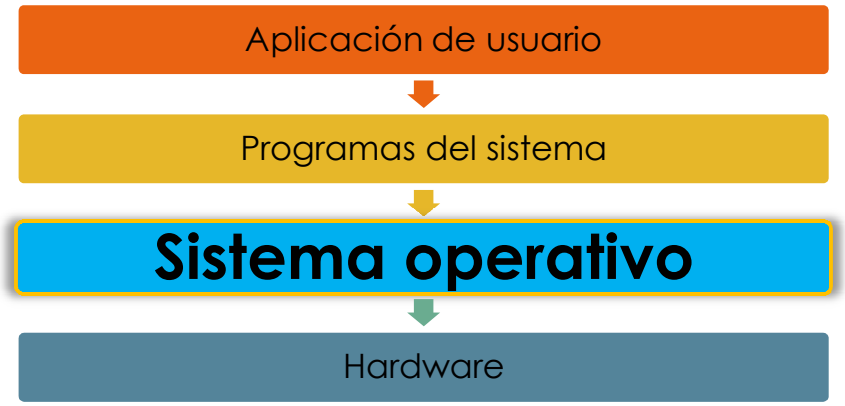
- **Contador del programa**: indica el punto de ejecución.
- **Imagen de memoria**: todo el espacio en memoria que está utilizando.
- **Estado del procesador**: valor de los registros del procesador que se están utilizando.

Cuando se interrumpe la ejecución de un proceso ppor alguna causa, se verán más adelante, se guarda una copia de los tres elementos nombrados arriba para poder restaurar la ejecución del proceso en el mismo punto en el que se interrumpió.

Los procesos son entidades independientes aunque ejecuten el mismo programa de manera que pueden coexistir en ejecución dos procesos que ejecutan el mismo programa. Por ejemplo se pueden tener abiertos dos archivos pdf a la vez o si se utiliza el navegador chrome cada pestaña que se tiene abierta es un proceso diferente.

Ejecutable: es el fichero que contiene la información necesaria para poder crear el proceso correspondiente.

Sistema operativo: programa encargado de gestionar de manera eficiente todos los recursos hardware y software del equipo proporcionando un entorno amigable y fácil de usar al usuario.



Demonio: proceso que se ejecuta en segundo plano no accesible al usuario, su función es proporcionar un servicio al resto de procesos que se ejecutan. Un ejemplo de demonio sería el *Garbage collector* de Java que se encarga de liberar memoria cuando un objeto ya no tiene ninguna referencia a memoria activa.

2. Programación concurrente

La **computación concurrente** permite que múltiples tareas se estén ejecutando a la vez, por ejemplo se puede estar escuchando música mientras se descargan archivos p2p y se redacta un documento de texto, además el antivirus está también en ejecución.

Si todas esas tareas se tuvieran que ejecutar una tras otra se perdería mucho tiempo a nivel de usuario.

Con sistemas operativos antiguos como **MS-DOS** los programas se ejecutaban de manera exclusiva ya que no existía la programación concurrente, solo se podía realizar una tarea a la vez.

Para poder implementar la **programación concurrente** se dispone de diferentes escenarios posibles:

- **Un único procesador** (multiprogramación) sin diferentes núcleos ni hilos. Mediante este caso solo un proceso se esta ejecutando cada vez y el sistema operativo va cambiando el proceso en ejecución después de un periodo corto de tiempo (milisegundos), así en un segundo pueden ejecutarse múltiples procesos.
  No mejora el tiempo global de ejecución de los programas pero el usuario tiene la percepción de que varios procesos se ejecutan a la vez aunque no es así.

- **Varios núcleos en un mismo procesador** (multitarea) sin diferentes hilos. Joy en día es bastante normal que los procesadores contengan varios núcleos, y no solo los procesadores de los ordenadores si no que esto también ocurre en los procesadores de los teléfonos móviles.
  Cada núcleo ejecuta una instrucción diferente ya sea del mismo programa o de un programa diferente. Como todos los núcleos comparten memoria la comunicación entre procesos es muy rápida y sencilla de implementar. Por ejemplo al escribir un documento de texto un núcleo se encarga de gestionar la escritura y otro de ejecutar la corrección de gramáica y ortografía.
  A esto se le conoce como **programación paralela** y sí que mejora el rendimiento de un programa al permitir que diferentes instrucciones se ejecuten a la vez.
  Se puede utilizar conjuntamente con la multiprogramación.

- **Varios ordenadores distribuidos en red**. Cada equipo tiene su propio procesador y memoria, la gestión conjunta de todos los equipos se llama **programación distribuida**. Mejora el rendimiento considerablemente pero imposibilita la comunicación directa entre procesos por lo que hay que utilizar unos esquemas de comunicación más complejos y costosos a través de la red.

3. Funcionamiento básico del sistema operativo

El sistema operativo (SO) como cualquier otro programa necesita estar en memoria para poder ser ejecutado. Existe una parte del sistema operativo que ha de estar siempre en memoria, es el **núcleo o kernel**, el kernel es una parte muy pequeña del SO sobre todo si se compara con todo lo necesario para implementar la interfaz gráfica. Al resto de SO se le llama **programas del sistema**.

El kernel gestiona los recursos del ordenador permitiendo el acceso a estos mediante las **llamadas al sistema**.

El kernel del sistema generalmente trabaja en base a **interrupciones** (IRQ – InterruptReQuest). Una interrupción es la suspensión temporal de un proceso para ejecutar la rutina que gestiona dicha interrupción. Mientras se atiende una interrupción se deshabilita la llegada de otras interrupciones. Las interrupciones suelen estar generadas por llamadas al sistema.

Las llamadas al sistema se programan con **lenguajes de bajo nivel** como C o C++ debido a que permiten el acceso a niveles más bajos del hardware. En **lenguajes de alto nivel** los programadores hacen uso de **API**s, las más comunes son Win32 (Windows), API POSIX (para sistemas Unix en los que se incluye GNU Linux y Mac OS).

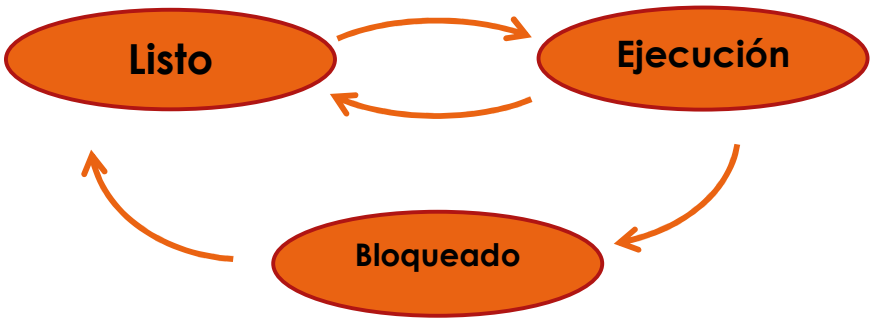
4. Procesos

Como se ha dicho anteriormente un proceso es el estado de un programa en un momento dado de su ejecución con todo lo que ello significa: contador, memoria y registros de la CPU.

Es el SO el encargado de poner en ejecución y gestionar los procesos. Los procesos para poder cumplir con la multiprogramación y programación paralela pasan por diferentes estados a lo largo de su ciclo de vida. Los cambios de estado también son gestionados por el SO.Existen varios modelos para implementar el ciclo de vida de un proceso.

4.1. Estados de un proceso

- Modelo de tres estados.



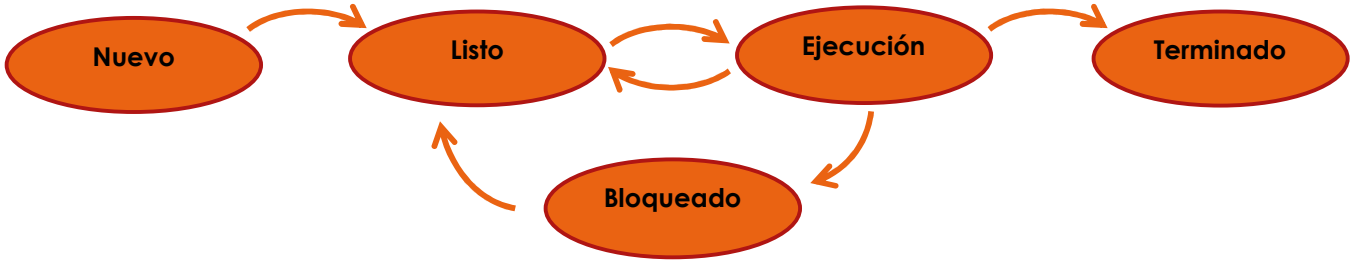
Listo: con posibilidad de entrar a la CPU, ya tiene asignada su zona en memoria.

Ejecución: se encuentra dentro de la CPU ejecutando sus instrucciones.

Bloqueado: sin posibilidad de entrar a la CPU, por ejemplo, está esperando una operación de E/S.

Este modelo es muy simple y no contempla que un proceso se esté creando o que ya haya terminado.

- Modelo de cinco estados.

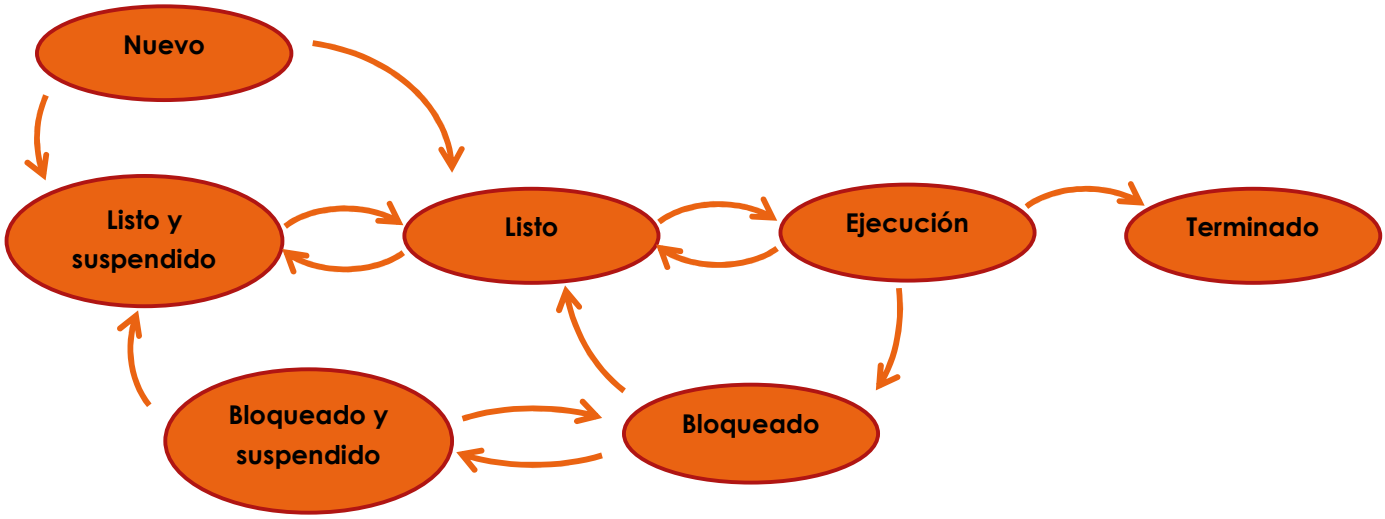


Nuevo: se está creando o acaba de crearse, pero no está listo para la ejecución, por ejemplo, si ya se ha alcanzado el límite de la memoria.

Terminado: ha finalizado y libera su imagen de memoria, también pasa a terminado si se produce un error y deja de ejecutarse.

- Modelo de siete estados.

Hay ocasiones en las que se encuentran muchos procesos bloqueados ocupando memoria y no dejan que otros procesos entren en ejecución. En estos casos es importante que se permita un **intercambio** y que esos procesos bloqueados pasen a disco (memoria virtual).



Bloqueado y suspendido: procesos bloqueados que se encuentran esperando en la memoria secundaria al no caber más procesos en memoria.

Listo y suspendido: procesos preparados para entrar en ejecución pero que se encuentran en memoria secundaria al no caber más procesos en memoria.

4.2. Colas de procesos

Como uno de los objetivos del SO es la multiprogramación, el SO necesita saber qué procesos están en cada estado para saber cuál de ellos puede entrar en la CPU, para esto hace uso de colas de procesos.

- Cola general de procesos: contiene todos los procesos del sistema.
- Cola de procesos preparados: contiene aquellos procesos listos para ejecutarse.
- Colas de dispositivo: contiene los procesos que están esperando alguna operación E/S.

4.3. Planificación de procesos

El SO necesita un **planificador de procesos** que se encargue de gestionar las colas de procesos. Existen dos tipos de planificación:

- **Corto plazo**: se encarga de elegir cual es el siguiente proceso en entrar a la CPU, esta operación se ejecuta muchas veces y cada poco tiempo (milisegundos) por lo que el algoritmo ha de ser sencillo. Existen tres tipos de algoritmos para esta planificación.
  - Sin desalojo: solo se cambia de proceso en ejecución si este se termina o bloquea.
  - Con desalojo: si llega un proceso con más prioridad se desaloja al que está ejecutándose para que entre ese proceso prioritario.
  - Tiempo compartido: cada cierto tiempo se cambia el proceso en ejecución.
- **Largo plazo**: se encarga de gestionar los procesos que pasan a la cola de preparados, se invoca con poca frecuencia.

Cada proceso es único e impredecible, los hay orientados a CPU y orientados a E/S.

CPU	E/S	CPU	E/S	CPU

CPU	E/S	CPU	E/S	CPU	E/S	CPU

Los algoritmos de programación a corto plazo deben de cumplir los siguientes objetivos:

- **Justicia/equidad**: todos los procesos han de tener su turno de CPU.
- **Eficiencia/utilización**: deben mantener la CPU ocupada el mayor tiempo posible.

4.4. Cambio de contexto

Se llama contexto al conjunto de datos relacionados con el proceso en ejecución, dentro de esos datos se encuentra el estado del proceso, el estado del procesador (valores de los registros) y el espacio de memoria que ocupa el proceso.

Cuando se cambia el proceso en ejecución se produce un **cambio de contexto** mediante el cual se guarda una copia del contexto del proceso que se interrumpe para poder más tarde volver al mismo punto de ejecución del mismo.

Los cambios de contexto son tiempos perdidos ya que el procesador no realiza ninguna tarea útil durante ese tiempo. La duración depende de la arquitectura del procesador.

4.5. Algoritmos de planificación

- Orden de llegada – FCFS (**F**irst**C**ome **F**irst**S**erved).

  Se atendrá a los procesos por orden de llegada y el proceso que entra a la CPU no la abandona hasta que termine su ejecución.

  Ejemplo: P1 → 5 ciclos de CPU, P2 →1 ciclo de CPU, P3 → 2 ciclos de CPU

Ciclo CPU	1	2	3	4	5	6	7	8
P1	CPU	CPU	CPU	CPU	CPU			
P2	E	E	E	E	E	CPU		
P3	E	E	E	E	E	E	CPU	CPU

  P1: Tiempo de vida = 5 Tiempo de espera = 0

  P2: Tiempo de vida = 6 Tiempo de espera = 5

  P3: Tiempo de vida = 8 Tiempo de espera = 6

  Tiempo medio de vida: (5+6+8)/3 = 6,33

  Tiempo de espera medio: (0+5+6)/3 = 3,66

- Prioridad al más corto – SJF (**S**hortest**J**ob **F**irst).

  Cuando tiene que entrar un proceso, entrará aquel que tenga menos tiempo de ejecución. No se expulsará a ningún proceso de la CPU.

  Ejemplo: P1 → 3 ciclos de CPU, P2 →4 ciclos de CPU, P3 → 2 ciclos de CPU

Ciclo CPU	1	2	3	4	5	6	7	8	9
P1	E	E	CPU	CPU	CPU				
P2	E	E	E	E	E	CPU	CPU	CPU	CPU
P3	CPU	CPU							

  P1: Tiempo de vida = 5 Tiempo de espera = 2

  P2: Tiempo de vida = 9 Tiempo de espera = 5

  P3: Tiempo de vida = 2 Tiempo de espera = 0

  Tiempo medio de vida: (5+9+2)/3 = 5,33

Tiempo de espera medio: (0+5+2)/3 = 2,66

- Prioridad al que menos tiempo le queda – SRTF (**S**hortest**R**emaining**T**ime **F**irst).
  Cada vez que llega un proceso se comprobará cuál de todos los procesos tiene menos tiempo de ejecución y ese entrará en la CPU, este algoritmo conlleva la expulsión del proceso de la CPU si llega un proceso más corto.

  Ejemplo: P1 → 6 ciclos de CPU (llega en el ciclo 1)
  P2 → 2 ciclos de CPU (llega en el ciclo 2)
  P3 → 3 ciclos de CPU (llega en el ciclo 3)

Ciclo CPU	1	2	3	4	5	6	7	8	9	10	11
P1	CPU	E	E	E	E	E	CPU	CPU	CPU	CPU	CPU
P2		CPU	CPU								
P3			E	CPU	CPU	CPU					

P1: Tiempo de vida = 11 Tiempo de espera = 5

P2: Tiempo de vida = 2 Tiempo de espera = 0

P3: Tiempo de vida = 4 Tiempo de espera = 1

Tiempo medio de vida: (11+2+4)/3 = 5,66

Tiempo de espera medio: (5+0+1)/3 = 2

Los anteriores algoritmos no son implementables debido a que es imposible calcular el tiempo que va a estar un proceso en CPU.

- Por prioridades.

  Se tendrán en cuenta diversos factores para ampliar la prioridad de los procesos, por ejemplo el tiempo que llevan el sistema.

- Múltiples colas.

  Este algoritmo permite dividir la cola de procesos "listos" en diferentes colas (categorías), cada una de estas subcolas tendrá su propio algoritmo de planificación,

se debe de poder cambiar a los procesos de cola. Y, por supuesto, se debe elegir un algoritmo que elija de qué cola se elegirá al siguiente proceso en entrar a la CPU.

- Circular – Round-Robin.

Es el más usado en los SO modernos. Tiene como base el algoritmo FCFS, además se establece un tiempo máximo de uso de la CPU por turno llamado **quantum**. Al acabar el tiempo establecido por el quantum el proceso en ejecución pasará al estado "listo".

Es un algoritmo que conlleva la expulsión de los procesos en la CPU.

Ejemplo: P1 → 6 ciclos de CPU (llega en el ciclo 1)
P2 → 2 ciclos de CPU (llega en el ciclo 5)
P3 → 3 ciclos de CPU (llega en el ciclo 3)
Quantum = 2

Ciclo CPU	1	2	3	4	5	6	7	8	9	10	11
P1	CPU	CPU	CPU	CPU	E	E	CPU	CPU			
P2					E	E	E	E	CPU	CPU	
P3			E	E	CPU	CPU	E	E	E	E	CPU

Cola ciclo 1:

Listos	P1(entra)				

Cola fin ciclo 2:

Listos	P1				

Cola ciclo 3:

Listos	P1(entra)	P3			

Cola fin ciclo 4:

Listos	P3	P1			

Cola ciclo 5:

Listos	P1	P2			

Cola fin ciclo 6:

Listos	P1	P2	P3		

Cola ciclo 7:

Listos	P1(entra)	P2	P3		

Cola ciclo 9:

Listos	P2(entra)	P3			

Cola ciclo 10:

Listos	P3(entra)				

P1: Tiempo de vida = 8 Tiempo de espera = 2

P2: Tiempo de vida = 6 Tiempo de espera = 4

P3: Tiempo de vida = 9 Tiempo de espera = 6

Tiempo medio de vida: (8+6+9)/3 = 4,33

Tiempo de espera medio: (2+4+6)/3 = 4

5. Gestión de procesos

5.1. Árbol de procesos

Cuando un usuario procede a abrir un programa es el SO el responsable de crear y poner en ejecución el proceso correspondiente. El responsable del proceso de creación es el SO ya que es el que gestiona los recursos del ordenador. Por lo que la creación de un nuevo proceso siempre viene dada por un proceso ya existente. La creación de procesos nuevos siempre es en nombre del usuario o de otro proceso existente.

Por ejemplo, si se quiere abrir el navegador y se hace doble clic en su icono, es la interfaz gráfica la encargada de hacer la petición de nuevo proceso del navegador.

Como cualquier proceso en ejecución siempre depende del proceso que lo creó se establece un **vínculo** entre ambos procesos. El proceso creador es el **padre** y el creado el **hijo**.

Cuando se arranca el ordenador el kernel es cargado en memoria a partir de su imagen en disco de manera que se crea el proceso principal. A partir de ese proceso se crearán el resto de procesos de manera jerárquica.

Los SO utilizan un **identificador de proceso** (PID – **P**rocess**ID**entifier) único y unívoco para cada proceso. El uso del PID es una herramienta básica para la gestión de procesos ya que es la única forma con la que el sistema puede referirse a ellos.

5.2. Operaciones básicas con procesos

Como se ha dicho anteriormente los procesos se crean de manera jerárquica, así el creador será el padre y el creado el hijo. Esta operación de creación de un nuevo proceso se denomina **create**.

Nada más crear un nuevo proceso, padre e hijo se ejecutan concurrentemente, intercambiándose en la CPU según el algoritmo de planificación establecido por el SO. Si el padre tuviera que esperar a que el hijo finalice su ejecución para continuar lo hace mediante la operación **wait**.

Aun siendo padre e hijo los procesos son independientes y cada uno dispone de su espacio de memoria propio. Puede parecer que el proceso hijo ejecute un programa diferente que el padre pero es no es así, depende del SO.

En sistemas operativos Windows existe la función **createProcess()** que crea un proceso nuevo a partir de un programa distinto al que está en ejecución. Sin embargo en sistemas UNIX se utiliza la función **fork()**, esta función crea un proceso hijo con un duplicado del padre y continúa la ejecución en ese punto. Aun así, en los dos casos los procesos son independientes y los cambios en memoria solo afectarán a su espacio de memoria reservado.

Para poder compartir información, los procesos pueden hacer uso de **recursos compartidos** (ficheros abiertos, memoria compartida…). La **memoria compartida** es una región de la memoria a la que pueden acceder varios procesos que colaboran. El SO solo se encarga de crear y establecer los permisos de los procesos que pueden acceder a ese espacio de memoria compartido. Los procesos se encargarán del formato de los datos compartidos y su ubicación.

Al terminar un proceso, mediante la operación **exit**, el proceso indica al SO que quiere terminar, pudiendo mandar información al SO en esa misma operación. El SO liberará si es posible los recursos asignados.

El proceso hijo depende tanto del SO como del proceso padre que lo creó. De esta manera un padre puede finalizar la ejecución de un proceso creado por él. Para esto existe la operación **destroy**. Hay sistemas operativos en los que no se permite que un hijo continúe ejecutándose si su padre termina, a esto se le denomina **terminación en cascada**.

Como se ha podido ver hasta ahora cada SO tiene sus peculiaridades y características por lo que gestionan los procesos de manera diferente. Para intentar evitar depender mucho del sistema operativo durante el curso se trabajará con Java. En Java los padres e hijos no tienen por qué ejecutarse de manera concurrente y no se produce la terminación en cascada. Aunque hay que tener en cuenta que algunos procesos especiales nativos pueden no funcionar correctamente (ventanas en MS-Dos/Windows o Shell scripts en UNIX/Linux/Mac).

5.2.1. Creación de procesos - create

En Java la clase **Process** representa un proceso. Los métodos **ProcessBuilder.start()** y **Runtime.exec()**sirven para crear un proceso nativo en el SO devolviendo un objeto de la clase Process. Dicho objeto será el que se utilice para controlar al proceso creado.

- **ProcessProcessBuilder.start()**: Se ejecutará el comando y argumentos que se indiquen en **command()**, se ejecutará en el directorio que se indique con **directory()** y se usarán las variables del entorno que se indiquen en **environment()**.

- **ProcessRuntime.exec(String[] cmdarray, String[] envp, File dir)**: Se ejecutará el proceso y argumentos especificados en cmdarray en un proceso hijo independiente, usando el entorno envp y en el directorio de trabajo dir.

Los dos métodos anteriores se encargan de comprobar que el comando a ejecutar es válido en el SO en el que se ejecuta la Máquina Virtual Java (JVM). Por lo que pueden surgir problemas como que no se encuentre el ejecutable por ruta errónea, que no se disponga de los permisos correctos, que no sea un comando/ejecutable válido… Cuando esto ocurre se lanzará una excepción dependiente del SO que será una subclase de **IOException**.

Ejemplo – Creación de un proceso mediante ProcessBuilder.

```
importjava.io.IOException;
importjava.util.Arrays;

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

ProcessBuilderpb = new ProcessBuilder(args);
      try {
Processprocess = 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");
      }
   }
}
```

5.2.2. Terminación de procesos - destroy

Un proceso hijo finalizará cuando acabe de realizar su función ejecutando la operación exit o bien si su padre lo termina mediante la operación destroy, de las dos maneras se liberarán los recursos usados por el proceso hijo.

Ejemplo – Creación de un proceso mediante Runtime y posterior destrucción del mismo.

```
importjava.io.IOException;

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

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

6. Comunicación de procesos

Reader: lector de datos.

Stream: flujo de datos.

Buffer: almacén de datos.

Pipe: tubería, canalizar/verter.

Como ya se ha dicho anteriormente los procesos son programas que están en ejecución y cualquier programa recibe información, la transforma y muestra los resultados. Estas acciones se gestionan mediante:

- **Entrada estándar** (stdin): es el lugar del que el proceso lee los datos que necesita para su ejecución. Puede ser el teclado, un fichero, la tarjeta de red, otro proceso…

- **Salida estándar** (stdout): es el lugar donde el proceso escribe los resultados que genera debido a su ejecución. Entre los destinos posibles se encuentra la impresora, la pantalla, un fichero, otro proceso…

- **Salida de error** (stderr): lugar donde se envían los mensajes de error que se generen durante la ejecución. Suele ser el mismo que la salida estándar pudiéndose indicar otra salida diferente. Generalmente se suelen generar ficheros **log** con los errores.

En la gran mayoría de SO la entrada y salida en un proceso hijo suelen ser copia del proceso padre. Por ejemplo, si un proceso con un fichero como entrada estándar y la pantalla como salida estándar hace una operación **create**, el hijo podrá leer del mismo fichero y su salida será la pantalla.

En Java, sin embargo, al crear un proceso de la clase Process, este hijo creado no tiene su misma interfaz de comunicación, o lo que es lo mismo no hay comunicación directa entre el padre y e hijo.

En el caso de Java todas las salidas y entradas de información (stdin, stdout y stderr) se redirigen al padre mediante los siguientes flujos de datos (**streams**):

- **OutputStream**: flujo de salida del hijo. El stream está conectado mediante un **pipe** a la entrada estándar del proceso hijo.
- **InputStream**: flujo de entrada del proceso hijo. El stream está conectado mediante un **pipe** a la salida estándar del proceso hijo.

- **ErrorStream**: flujo de error del hijo, El stream está conectado mediante un **pipe** a la salida estándar de error del proceso hijo, que por defecto en JVM es la salida estándar.

Si se necesita tener separada la salida estándar de la salida de error la clase **ProcessBuilder** tiene un método llamado **redirectErrorStream(boolean)**, si se le pasa un valor

Usando los streams vistos anteriormente un proceso puede enviar y recibir información a su hijo y viceversa.

En algunos SO el tamaño de los buffers de entrada y salida correspondientes a la entrada y salida estándar tienen un tamaño limitado, así un fallo al leer o escribir en los flujos de entrada o salida provocarían que el proceso hijo se pudiera bloquear. Pasa solucionar esto en Java la comunicación entre procesos padre e hijo se realiza con buffers propios.

Es importante conocer cómo codifica la información que se envía entre procesos. Esto depende del SO donde se ejecuta el proceso hijo. En Java puede ser necesario especificar cómo se reciben los datos en el SO donde se ejecute.

Ejemplo – Comunicación entre procesos usando un buffer.

```
importjava.io.BufferedReader ;
importjava.io.IOException;
importjava.io.InputStream;
importjava.io.InputStreamReader;
importjava.util.Arrays;

publicclassComunicationBetweenProcess {
publicstaticvoidmain (Stringargs[]) throwsIOException {
Processprocess = new ProcessBuilder(args).start() ;
InputStreamis = process.getInputStrearn() ;
InputStreamReaderisr = new InputStreamReader(is);
BufferedReaderbr = new BufferedReader(isr);
String line;

System.out.println("Salida del proceso " + Arrays.toString(args) + " :" ) ;
while((line=br.readLine())!= null ) {
System.out.println(line);
}
}
}
```

Generalmente y salvo alguna excepción sobre todo con algunas versiones de Windows, todos los SO utilizan UTF-8 como formato de codificación de caracteres.

Existen otras alternativas de comunicación a parte de los streams:

- Sockets: se verá en la unidad 3.
- JNI (Java Native Interface): como Java es un lenguaje de alto nivel, no es posible a acceder a elementos de bajo nivel del SO donde se ejecutan los programas. Por ejemplo, no se puede acceder a un **pipe** al ser un elemento de bajo nivel. Usando JNI se puede acceder desde Java a aplicaciones desarrolladas en lenguajes de programación de bajo nivel como por ejemplo C.
- Librerías de comunicación no estándares. Se encuentran en fase de investigación y desarrollo. CLIPC, librería de código abierto para Java que utiliza llamadas a JNI para utilizar elementos de bajo nivel del SO:
  - Memoria compartida: zona de memoria a la que pueden acceder varios procesos. Se utiliza para que se puedan comunicar los procesos.
  - Pipes: canal de comunicación sencillo
  - Semáforos: mecanismo que permite bloquear un proceso hasta que ocurra un evento.

Actividad – Busca como desarrollar una aplicación en Java que llame a un código escrito en otro lenguaje de programación usando JNI.

7. Sincronización de procesos

Mediante los métodos con lo que se comunican los procesos se pueden sincronizar estos porque permiten al proceso padre controlar el ritmo de envío y recepción de mensajes. Mediante los streams es posible tener un canal de comunicación unidireccional bloqueante.

Por ejemplo, si un proceso solicita una lectura de la salida estándar del hijo a través de su InputStream, el proceso padre se bloqueará hasta que el hijo le envíe los datos.

7.1. Espera de procesos – operación **wait**

Se ha visto cómo se puede hacer que un proceso espere hasta que un hijo le envíe información, esto también se puede hacer utilizando la operación wait. Con esta operación el proceso padre se bloqueará hasta que el hijo finalice su ejecución mediante una operación exit.

Al finalizar el hijo enviará un valor de retorno que convencionalmente es 0 para indicar que todo ha acabado de forma correcta.

La clase Process dispone del método **waitFor()** el padre esperará bloqueado a que el hijo termine. El padre también volverá a su ejecución si el hijo termina antes de tiempo u otro proceso lo interrumpe (se lanzará una interrupción InterruptedException en este caso). Se dispone también del método **exitValue()** para obtener el valor de retorno que envió el proceso hijo al acabar. Si el proceso hijo no acabara se lanzará la excepción IllegaThreadStateException.

Ejemplo – Sincronización de procesos.

```
importjava.io.IOException;
importjava.util.Arrays;

publicclassProcessSincronization {
publicstaticvoidmain(String[ ] args)throwsIOException, InterruptedException {
try{
Processprocess = 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 elcomando. Descripción: " + e.getMessage());
      } catch(InterruptedException e) {
System.out.println("E l comando fue interrumpido. Descripción del error: " + e.getMessage()) ;
      }
   }
}
```

8. Programación multiproceso

Como se ha visto anteriormente la programación concurrente es una manera de procesar la información eficaz al permitir que diferentes procesos compartan el procesador alternándose en él. Estos procesos pueden ser de diferentes programas independientes o bien procesos que realizan una tarea común o que comparten información.

El SO operativo es el que se encarga de proveer la multiprogramación abstrayendo tanto a los usuarios como a los desarrolladores, pero si se pretende que los procesos cooperen entre sí entonces el desarrollador deberá implementar esto usando la comunicación y la sincronización.

Los desarrolladores deben de seguir unas fases para poder realizar un programa multiproceso cooperativo:

1 Descomposición funcional: consiste en identificar las diferentes funciones que realiza la aplicación y cómo se relacionan entre sí.

2 Partición: consiste en distribuir las funciones en diferentes procesos teniendo en cuenta cómo se comunicarán estos. Como estos procesos colaborarán tendrán que comunicarse y sincronizarse, hay que tener en cuenta el tiempo que se pierde en esas tareas.

3 Implementación: una vez realizada la descomposición y la partición toca desarrollar con las herramientas que ofrece el lenguaje de programación.

8.1. Clase **Process**

Para implementar algoritmos en los que entren técnicas de multiproceso en Java se utiliza la clase **Process**. A continuación, se puede observar una tabla con los métodos que se han visto en la unidad.

Método	Tipo de retorno	Descripción
getOutputStream()	OutputStream	Obtiene el flujo de salida del hijo conectado al stdin.
getInputStream()	InputStream	Obtiene el flujo de entrada del hijo conectado al stdout.
getErrorStream()	InputStream	Obtiene el flujo de entrada del hijo conectado al stderr.
destroy()	void	Termina el proceso.
waitFor()	int	Realiza una espera hasta que el hijo responda.
exitValue()	int	Obtiene el valor de retorno del hijo.

9. Caso práctico

Desarrollar una clase en Java que ejecute dos comandos, cada uno en un hijo, con sus respectivos argumentos y redireccione la salida estándar del primero a la entrada estándar del segundo.

Los comandos se pondrán directamente en el código del programa por sencillez.

Ejemplo: ejecución de los comandos **ls –la** y **tr "d" "D"**. El resultado es el mismo que si se por línea de comandos en Unix la siguiente instrucción: **ls –la | tr "d" "D"**.

```
total 6
Drwxr-xr-x5userusers40962011-02-2210:59 .
Drwxr-xr-x 8userusers 4096 2011-02-0709:26..
-rw-r--r-- 1 userusers 302011-02-22 10:59 a.txt
-rw-r--r-- 1 userusers 27 2011-02-22 10:59 b.txt
Drwxr-xr-x 2 userusers 4096 2011-01-24 17:49 Dir3
Drwxr-xr-x 2 userusers 4096 2011-01-24 11:48 Dir4
```