2. Comunicación entre procesos

2.1. Comunicación mediante Entrada/Salida

Una de las formas más sencillas que ofrece el sistema operativo para comunicar procesos es mediante la entrada y salida de estos.
Todo proceso tiene asociados tres flujos de información o streams:

- stdin (con descriptor de fichero 0), que representa la entrada estándar del proceso, por donde este recibe los datos. Esta entrada generalmente está asociada a la entrada de teclado de la terminal.
- stdout (con descriptor de fichero 1), que representa la salida estándar del proceso, por donde este muestra su salida. Generalmente esta salida está asociada a la pantalla de la terminal.
- stderr (con descriptor de fichero 2), que representa la salida de error del proceso, por donde este muestra los mensajes de error. Generalmente se trata de la pantalla de la terminal.

Vamos a ver diferentes mecanismos que aprovechan esta entrada y salida para realizar la comunicación entre procesos.

Redirecciones
Un mecanismo relativamente simple de comunicación de procesos es mediante redirecciones. Este mecanismo consiste en redirigir estos descriptores de ficheros desde o hacia otras fuentes diferentes a la estándar.
Veamos en la siguiente tabla algunas formas de hacer posible esta comunicación:

Tipo	Descripción	Ejemplo
Redirección de la entrada estándar (< y <<)	Redirige la entrada de datos de un proceso desde un fichero, en lugar de hacerlo desde el teclado. Con << obtenemos la entrada para el comando línea por línea mediante el teclado, hasta que se escriba la marca indicada.	$ cat << FIN > Hola a todos > Bienvenidos! > FIN Hola a todos ¡Bienvenidos!
Redirección de la salida estándar (> y >>)	Permite redirigir la salida del proceso a un fichero, eliminando su contenido (con >) o añadiendo el resultado al final (con >>).	$ echo Hola! > Saludo.txt $ echo "Bienvenidos" >> Saludo.txt

Redirección de la salida de error estándar (2> y 2>>)	Permite redirigir la salida de error del proceso a un fichero, del mismo modo que la salida estándar.	$ let a=3/0 2> /dev/null Redirige el error de división por 0 al periférico nulo, ignorando este error.

Tuberías (I)
Las tuberías (pipes) permiten conectar la salida estándar de un proceso con la entrada estándar de otro, estableciendo así una relación de productor-consumidor.
El uso de tuberías sigue la siguiente sintaxis:

comando_productor | comando_consumidor

Un ejemplo clásico es utilizar la salida del comando ps como entrada al comando grep, para filtrar la salida de ps y obtener, por ejemplo, información de un programa concreto.
Ejemplo:

```
$ ps aux | grep bash
user 13436 0.0 0.0 11872 5224 pts/ Ss+ Dic21 0:00 /bin/bash
user 53221 0.0 0.0 1160 6108 pts/2 Ss 16:33 0:00 /bin/bash
user 54581 0.0 0.0 11472 5808 pts/3 Ss 17:03 0:00 /bin/bash
user 57203 0.0 0.0 11604 5952 pts/4 Ss 17:57 0:00 /bin/bash
user 60416 0.0 0.0 11480 5868 pts/5 Ss+ 20:10 0:00 /usr/bin/bash
```

2.2. Comunicación mediante variables de entorno y de la shell

En bash podemos utilizar variables locales a la shell y variables de entorno, que definiremos mediante export. Estas variables de entorno pueden ser también utilizadas como mecanismo de comunicación entre procesos.
Veámoslo con un pequeño ejemplo. Si en un script (script1.sh) definimos la variable MYVAR del siguiente modo e invocamos a otro script (script2.sh), desde este segundo script tendremos acceso a la variable exportada del primero:

```
#!/bin/bash
# fichero script1.sh
export MYVAR="Valor"
./script2.sh
#!/bin/bash
# fichero script2.sh
echo "El contenido de la variable es $MYVAR"
Si lanzamos el script script1.sh, su salida será:
echo "El contenido de la variable es Valor"
```

2.3. Comunicación mediante señales

Las señales son mensajes que el sistema operativo envía a un proceso en tiempo de ejecución, para informarle del suceso de un evento. Estas interrupciones son enviadas

por el kernel al proceso, aunque puede ser otro proceso quien las genere. Algunas de estas señales podrán ser también interpretadas como órdenes que recibe el proceso.

La orden Kill

Para enviar señales a un proceso utilizamos la orden kill, que responde a la siguiente sintaxis:

kill -señal [Lista de PIDs]

Si no especificamos nada, la señal que se envía es la número 15 (SIGTERM).
Los números de señal más comunes son:

Número	Señal	Descripción
1	SIGHUP	Permite parar y reiniciar un servicio.
2	SIGINT	Interrupción (equivalente a Control+C).
9	SIGKILL	Terminación abrupta del proceso.
15	SIGTERM	Terminación controlada del proceso.
18	SIGCONT	Reanuda un proceso suspendido.
19	SIGSTOP	Suspensión temporal del proceso.

Para ver una lista completa de estas señales podemos utilizar la orden kill -l.

La orden killall
Con killall podemos enviar una señal concreta a todos los procesos con el mismo nombre, en lugar de su PID.

Captura de señales. La orden trap
Mediante la orden trap podemos capturar señales y realizar ciertas acciones en respuesta a esta señal. La sintaxis de trap es:

trap 'órdenes' [lista de señales]

Donde órdenes serán las instrucciones a realizar cuando se reciba la señal. En caso de que no especifiquemos nada, se inhibirá la señal.
Hay que tener en cuenta que, por diseño del sistema operativo, la señal SIGKILL (9) no se puede capturar, ya que haría a un proceso indestructible.

3. Programación multiproceso. Procesos en Java

En este apartado vamos a centrarnos en la creación de procesos en Java. Mediante las

clases ProcessBuilder y Runtime, Java nos permitirá ejecutar comandos en la terminal, generando procesos que serán gestionados mediante la clase Process.

3.1. La clase ProcessBuilder

La clase java.lang.ProcessBuilder nos permite lanzar cualquier ejecutable desde Java. Para ello, una vez creada una instancia de ProcessBuilder, establecemos ciertos atributos del proceso y creamos este mediante el método start.
En otras palabras, podríamos decir que ProcessBuilder, como su nombre indica, es un Constructor o Generador de procesos.
Para crear una instancia de un ProcessBuilder podemos utilizar el constructor del siguiente modo:

- Un constructor vacío, sin argumentos:
  ProcessBuilder pb = new ProcessBuilder();
- Un constructor al que le proporcionamos un String con la orden a ejecutar:
  ProcessBuilder pb = new ProcessBuilder("firefox");
- Un constructor al que le proporcionamos una lista de Stings con el comando a ejecutar y sus argumentos:
  ProcessBuilder pb = new ProcessBuilder("firefox", "https://www.google.es");

IMPORTANTE
Debemos tener en cuenta que el comando que vayamos a lanzar se encuentre en alguna carpeta incluida en el PATH, del mismo modo que cuando lo lanzamos a través de la terminal. En caso contrario, deberemos especificar la ruta completa al ejecutable.

Una instancia de ProcessBuilder puede, además, gestionar los siguientes atributos del proceso a través de los métodos indicados para ello:

Atributo	Descripción
command	La lista con el comando que deseamos lanzar y sus argumentos. Una vez creado un ProcessBuilder, podemos modificar esta lista mediante el atributo command. Para acceder a él utilizaremos el método command(), sin argumentos para obtener su valor y con argumentos para establecerlos.
environment	Contiene un mapa de las diferentes variables de entorno. Inicialmente será una copia de las variables de entorno del proceso actual. Para acceder a él utilizaremos el método environment().

working directory	El directorio de trabajo en el que se va a ejecutar el comando. Si no se especifica, por defecto utiliza el directorio actual del proceso.
InputStream, OutputStream ErrorStream	Representan la entrada, la salida y la salida de error estándar del proceso, y pueden ser modificadas mediante los métodos redirectInput, redirectOutput y RedirectError. Trataremos con la E/S más adelante.

Veamos un ejemplo de cómo lanzar el navegador web Firefox mediante un ProcessBuilder.

IMPORTANTE
Descarga desde la sección Ejemplos para trabajar, el archivo PSP_U01_A03_01.zip. Ahí encontrarás los archivos necesarios para todos los ejemplos que veremos a continuación.
En este caso, hablamos de Lanzador.java, que puedes encontrar dentro de PSP_U01_A03_01.zip.

Debemos destacar que el proceso no se crea hasta que se invoca al método start del ProcessBuilder. De este modo, si invocamos, por ejemplo, dos veces el método start, crearemos dos instancias de la clase Process y, por lo tanto, dos procesos diferentes.
Por otra parte, debemos también tener en cuenta que el método start() puede lanzar una excepción de Entrada y Salida (IOException), por lo que debemos tratar esta adecuadamente, bien mediante captura o mediante propagación.

3.2. La clase Process

La clase java.lang.Process es una clase abstracta que encapsula la información entorno a la ejecución de un proceso.
Cuando invocamos al método start de un ProcessBuilder, este nos devuelve una instancia de Process, con la información sobre el proceso lanzado. Esta clase puede utilizarse para realizar operaciones de entrada y salida de los procesos, comprobar si este ha finalizado, su estado de finalización, o bien enviarle una señal para cerrarlo forzosamente.
Veamos cómo podemos, por ejemplo, lanzar dos instancias del mismo proceso y obtener información asociada a los procesos. Para ello utilizaremos el ejemplo Lanzador2.java. En él hemos creado dos objetos de tipo Process, donde guardamos los valores devueltos por sendas invocaciones al método start del ProcessBuilder:

```
Process p1=pb.start();
Process p2=pb.start();
```

Y mediante el método .pid() (incorporado en la versión 9 de Java) podemos obtener el PID de cada proceso y comprobar que cada invocación a start ha generado un proceso diferente.

```
System.out.println("PID del proceso 1: "+p1.pid());
System.out.println("PID del proceso 2: "+p2.pid());
```

IMPORTANTE
Observa que no hemos instanciado directamente ninguna de las instancias de Process. Esto se debe a que la clase Process es una clase abstracta y no puede ser instanciada. Cuando invocamos al método start, es este quien se encarga de instanciar el proceso mediante java.lang.ProcessImpl, que es una clase final.

Hay que tener en cuenta que, a diferencia de Bash, los procesos creados de esta forma no tienen una consola asociada (tty), por lo que no podemos redirigir las entradas o salidas estándar (stdin, stdout y stderr). Para ello podemos acceder a través de streams ofrecidos por varios métodos de la clase Process. Profundizaremos en esto más adelante.
Algunos de los métodos más interesantes de la clase Process son los siguientes:

Método	Descripción
int exitValue()	Devuelve el código de salida del proceso ejecutado.
Boolean isAlive()	Comprueba si el proceso hijo sigue en ejecución.
int waitFor()	Espera que el proceso hijo finalice. El valor entero que se obtiene es el código de salida del proceso.
Boolean waitFor (long timeOut, TimeUnit unidad)	Se trata de una sobrecarga del método anterior, donde podemos especificar el tiempo de espera. Devolverá cierto si el proceso sigue en ejecución después del tiempo indicado y falso en caso contrario.
void destroy()	Fuerzan la terminación del proceso, ya sea de manera controlada (SIGTERM) o forzosa (SIGKILL).
void destroyForcibily()	

En el ejemplo Lanzador3.java se ilustran algunos de estos métodos de la clase Process. Veamos algunos fragmentos clave del ejemplo.

1. En primer lugar, creamos un ProcessBuilder para lanzar el navegador Firefox y lanzamos este mediante start:
   String app[] = {"firefox",
   "https://docs.oracle.com/en/java/javase/17/core/process-apil.html"};
   ProcessBuilder pb = new ProcessBuilder(app);
   Process p = pb.start();

En el programa tienes otros ejemplos para lanzar, por ejemplo un editor de texto. Ten en cuenta que, si ya tienes Firefox abierto, el ejemplo puede no funcionar del todo.

2. En segundo lugar, utilizamos el método waitFor para esperar 3 segundos. Transcurrido este tiempo, si el usuario no ha cerrado la aplicación, la destruiremos mediante el método destroy().
   Boolean isProcessDead = p.waitFor(3, TimeUnit.SECONDS);
   if (!isProcessDead) {
   System.out.println("Destruyendo la aplicación");
   p.destroy();
   }
   Dado que la destrucción del proceso no es inmediata, esperaremos a que este finalice completamente. Para ello utilizamos el método isAlive, que devuelve si el proceso sigue vivo o no. Si el proceso sigue vivo, esperaremos un milisegundo y mostraremos un mensaje.
   while (p.isAlive()) {
   System.out.println("El proceso sigue vivo. Espero un milisegundo.");
   p.waitFor(1, TimeUnit.MILLISECONDS);
   }
3. Y finalmente, mostramos un mensaje indicando que este ha terminado y la salida del proceso (obtenida mediante exitValue):
   System.out.println("El proceso ha finalizado con la salida:" +p.exitValue());
   Como puedes comprobar, el valor devuelto por exitValue es 143, que se corresponde a la señal SIGTERM (Código 15). Si modificamos el código y en lugar de destroy utilizamos destroyForcibly, comprobaremos que el valor devuelto es el 137, que se corresponde con la señal SIGKILL (Código 9).

3.3. La interfaz ProcessHandle.Info

La interfaz ProcessHandle.Info nos permite obtener información adicional sobre un proceso.
Para obtener dicha información, únicamente deberemos acceder al método info() de la clase Process:
ProcessHandle.Info informacion = p.info();
Esta interfaz define los siguientes métodos:

Método	Descripción
String[] arguments()	Devuelve un vector de Strings con la lista de argumentos.
String command()	Devuelve un String con el path ejecutable del proceso.

String commandLine()	Devuelve un String con la línea del comando introducida, concatenando el comando y los argumentos.
Instant startInstant()	Devuelve la hora de inicio del proceso en un objeto de tipo Instant.
Duration totalCpuDuration()	Devuelve un objeto de tipo Duration con el tiempo de CPU acumulado por el proceso.
String user()	Devuelve una cadena de texto con el nombre del usuario que lanzó el proceso.

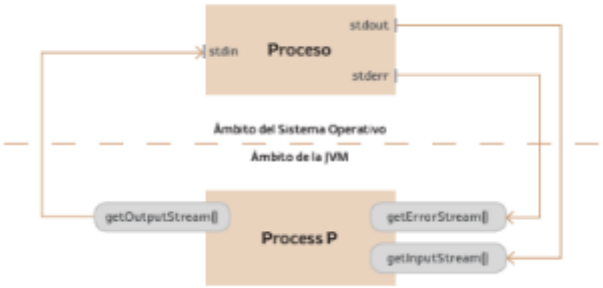
En el ejemplo Lanzador4.java podemos ver un ejemplo de uso de los métodos anteriores, así como el uso de esta interfaz.

4. Gestión de la entrada y salida de procesos en Java

4.1. Gestión de la entrada y salida de procesos mediante Streams
Como comentamos, los procesos creados con ProcessBuilder o con Runtime.exec no disponen de una terminal (tty) asociada, por lo que no podemos trabajar directamente con las entradas y salidas estándar del proceso. Para ello, la clase Process nos ofrece varios Streams o flujos de bytes asociados a la entrada y salida, a los que podemos acceder mediante los siguientes métodos:

- OutputStream getOutputStream(): Devuelve el flujo de salida conectado a la entrada estándar del proceso.
- InputStream getInputStream(): Devuelve el flujo de entrada conectado a la salida estándar del proceso.
- InputStream getErrorStream(): Devuelve el flujo de entrada conectado a la salida de error del proceso.

Debemos tener presente que la clase Process representa un proceso, pero no es el proceso en sí. Con el uso de estos métodos podemos proporcionar información al proceso a través del flujo de salida que tenemos conectado a él, y obtener así información del proceso mediante los flujos de entrada conectados.
Gráficamente, podríamos esquematizar esta relación del siguiente modo:



Además, cuando utilizamos estos tres métodos, deberemos tener en cuenta que los buffers de entrada y salida tienen una longitud limitada, por lo que si no se escriben y leen a continuación pueden llegar a bloquear el subproceso.

Ejemplo 1. Mostrando la información por pantalla
Vamos a ver un ejemplo de cómo trabajar con estos Streams de entrada y salida de los procesos. El ejemplo completo lo tienes disponible en el fichero PSP_U01_A04_01.zip, y en él vemos cómo podemos obtener la salida estándar de un proceso lanzado por nosotros y volcarla en nuestra propia terminal.
En primer lugar, creamos la instancia de ProcessBuilder con el comando y ejecutamos este mediante el método start, obteniendo el objeto Process correspondiente:

```
ProcessBuilder pb = new ProcessBuilder("cal", "2022");
Process p = pb.start();
```

Una vez hayamos lanzado el proceso, obtenemos el flujo de entrada conectado a la salida estándar del proceso:

```
InputStream is=p.getInputStream();
```

Para poder leer cómodamente de este Stream, el paquete java.io nos ofrece los decoradores InputStreamReader y BufferedReader, de modo que InputStreamReader recubre a InputStream, codificando los bytes en caracteres, y BufferedReader recubre a InputStreamReader, ofreciendo métodos para leer líneas directamente:

```
InputStreamReader isr=new InputStreamReader(is);
BufferedReader br=new BufferedReader(isr);
```

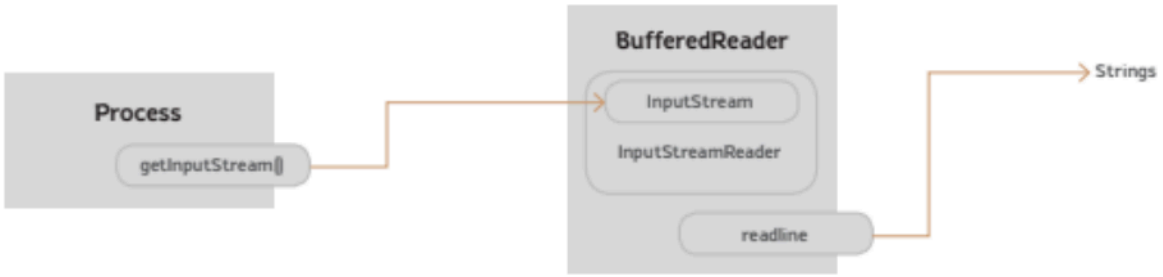
También podemos definir directamente la variable br sin necesidad de definir variables intermedias, del siguiente modo:

```
BufferedReader br = new BufferedReader(
new InputStreamReader(
p.getInputStream()));
```

Con estos decoradores podremos leer directamente líneas a partir de la salida estándar del proceso:

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

Gráficamente, podemos ver este proceso del siguiente modo:



IMPORTANTE
Sobre la codificación de caracteres
Un aspecto a tener en cuenta cuando trabajamos con Streams es la codificación de la información que se intercambia entre los procesos, que depende del sistema operativo en que estemos trabajando. La mayoría de sistemas (GNU/Linux, Mac OS, Android, iOS..) utilizan la codificación UTF-8, basada en el estándar Unicode. Por su parte, MS Windows utiliza sus propios formatos, incompatibles con el resto, como Windows-1252. Así que para tratar los datos correctamente en Java, cuando hacemos uso de mecanismos de comunicación entre procesos más avanzados habrá que tener en cuenta el tipo de codificación que el propio sistema utiliza.

Para usar un encoding concreto, la clase InputStreamReader tiene un constructor que nos permite especificar el tipo de codificación usado en el Stream de bytes que leemos.

Ejemplo 2. Redirección a ficheros
En el fichero PSP_U01_A04_02.zip tenemos un ejemplo de cómo redirigir esta salida hacia un fichero, utilizando para ello los decoradores FileWriter y BufferedWriter.

Para ello, una vez tengamos preparado el BufferedReader br para la lectura del proceso, en lugar de mostrarlo por pantalla lo volcamos al disco del siguiente modo:

```
BufferedWriter bw = new BufferedWriter(
new FileWriter(
new File("calendario2022.txt")));
String line;
while ((line = br.readLine()) != null) {
bw.write(line + "\n");
}
bw.close();
```

Con ello generaremos el fichero calendario2022.txt, cuyo contenido será la salida que antes mostrábamos por pantalla.

4.2. Redirección con ProcessBuilder

La clase ProcessBuilder nos ofrece una forma más simple de redirigir la entrada y salida de un proceso, ya sea hacia la salida estándar, hacia un fichero o hacia cualquier otra fuente o destino de datos, a través de métodos como redirectOutput, redirectInput o redirectError. Mediante estos métodos, únicamente deberemos indicar hacia dónde redirigir la salida.
Para indicar esta redirección podemos utilizar la clase ProcessBuilder.Redirect, que puede tomar algunos de los siguientes valores:

- ProcessBuilder.Redirect.PIPE: para indicar que el subproceso de E/S se conectará al proceso Java actual a través de una tubería. Este es el comportamiento por defecto.
- ProcessBuilder.Redirect.INHERIT: para indicar que el subproceso de E/S tendrá los mismos flujos de E/S que el proceso actual.
- ProcessBuilder.Redirect.DISCARD: descarta la salida del subproceso.
- Una redirección para leer de un fichero, obtenida mediante Redirect.from(fichero).
- Una redirección para escribir en un fichero, obtenida mediante Redirect.to(fichero).
- Una redirección para escribir al final de un fichero, obtenida mediante el método Redirect.appendTo(fichero).

En los ejemplos dispones de los ficheros incluidos en PSP_U01_A04_03.zip, que muestran cómo utilizar este mecanismo. El primer ejemplo utiliza ProcessBuilder.Redirect.Inherit para mostrar la salida del proceso por la misma salida que el proceso Java, esto es, la pantalla. Para ello, definimos nuestro ProcessBuilder, redirigimos la salida e iniciamos el proceso:

```
ProcessBuilder pb = new ProcessBuilder("cal", "2022");
pb.redirectOutput(ProcessBuilder.Redirect.INHERIT);
pb.start();
```

Por su parte, en el segundo ejemplo se muestra cómo redirigir esta salida a un fichero:

```
ProcessBuilder pb = new ProcessBuilder("cal", "2022");
File calendario=new File("calendario2022.txt");
pb.redirectOutput(ProcessBuilder.Redirect.to(calendario));
pb.start();
```

Tengamos en cuenta que con Redirect.to vamos a sobreescribir el contenido del fichero. Si lo que deseamos es mantener el contenido, utilizaremos el método Redirect.appendTo. Por ejemplo:

```
ProcessBuilder pb = new ProcessBuilder("cal", "2022");
File calendario=new File("calendario2022.txt");
BufferedWriter bw = new BufferedWriter(new FileWriter(calendario));
bw.write("------------ Calendario del año 2022 ------------\n");
```

```
bw.close();
pb.redirectOutput(ProcessBuilder.Redirect.appendTo(calendario));
```

4.3. ProcessBuilder y tuberías: StartPipeline

La API de ProcessBuilder incorpora el método estático startPipeline, que permite crear una tubería de procesos en los que la salida de un proceso se envía a la entrada del siguiente.
Este método se define del siguiente modo:

```
static List<Process> startPipeline(List<ProcessBuilder> builders)
```

Como vemos, este método recibe una lista de objetos de tipo ProcessBuilder, instanciados ya con sus propiedades correspondientes (comandos, entorno...), y devuelve una lista de objetos de tipo Process, como resultado de la puesta en marcha de cada proceso.
Así pues, si deseamos crear una tubería, por ejemplo, con la salida de:

```
ps aux | grep root
```

definiríamos la lista de builders con los dos ProcessBuilder:

5. Programación multiproceso

La paralelización de un proceso o aplicación tiene como principal objetivo mejorar su rendimiento. La idea es bastante sencilla. Si disponemos de un ordenador con varios procesadores y una tarea que requiera de cálculos intensivos, podemos dividir esta tarea entre los diferentes procesadores y ejecutarlos de forma paralela, para posteriormente combinar los resultados.

Durante este proceso nos encontraremos con algunas dificultades relacionadas con el intercambio de información o la sincronización entre tareas.

5.1. Diseño e implementación de algoritmos concurrentes

En líneas generales, los pasos a seguir cuando abordamos el diseño de algoritmos concurrentes son los siguientes:

1. **Identificar la concurrencia en el algoritmo**, analizando las tareas que se realizan en él, así como las diferentes estructuras de datos, con la finalidad de determinar qué partes son susceptibles de paralelizar.
2. **Diseñar el algoritmo de manera que podamos explotar la paralelización**. Para ello, distribuimos las tareas en subtareas o subprocesos y establecemos los mecanismos de comunicación y sincronización necesarios. Hay que tener en cuenta también que esta comunicación implica cierta pérdida de tiempo, por lo que es conveniente que sea mínima.
3. **Implementar el algoritmo en un entorno de programación** que soporte la creación de subtareas.

4. **Ejecutar el programa y analizar las mejoras en el rendimiento**, mejorando detalles de su funcionamiento si fuese necesario.

A. PASO 1: IDENTIFICANDO LA CONCURRENCIA

El primero de los pasos para diseñar un algoritmo concurrente es identificar la concurrencia. Esta puede encontrarse en las tareas, los datos o los flujos de datos y, según el caso, determinará la estrategia de paralelización y los patrones de diseño a seguir. Distinguimos entonces:

- **Distribución por tareas**, cuando diferentes partes del código se identifican como tareas, lo que implica una distribución de los datos.
- **Distribución por datos**, cuando las estructuras de datos se dividen de forma que cada tarea se encarga de una parte de estos datos.
- **Distribución por flujo de datos**, cuando es el flujo de datos el que determina cuándo se activan unas tareas para procesarlos.

Esta clasificación no es excluyente, de manera que puede que un problema presente los tres tipos de distribuciones. En todo caso, el resultado será una lista de tareas que podemos lanzar de forma paralela y que posiblemente presente ciertas dependencias de control y datos entre ellas.

En cualquier caso, es conveniente seguir ciertas buenas prácticas o criterios en la distribución de tareas. Estos criterios se centran en tres pilares: flexibilidad, eficiencia y simplicidad.

En el caso de la distribución por tareas, estas características se concretan en:

- **Flexibilidad.** El diseño del programa debe aportar flexibilidad en el número y tamaño de las tareas generadas, no vinculándose a una arquitectura concreta. Además, en la medida de lo posible deberíamos dar preferencia a las tareas parametrizadas respecto a las tareas fijas.
- **Eficiencia.** Las tareas deben tener suficiente trabajo como para compensar el coste de creación y gestión de estas, así como ser lo suficientemente independientes, minimizando con ello las dependencias entre ellas.
- **Simplicidad.** El código debe ser legible, fácil de entender y fácil de depurar.

Hay que tener en cuenta que una descomposición de tareas implica frecuentemente una descomposición de los datos, de modo que debamos abordar la descomposición de ambos.

En aquellos problemas en los que identifiquemos que el proceso principal se organiza en torno a la manipulación de grandes estructuras de datos o se realizan operaciones similares sobre diferentes partes de las estructuras de datos será conveniente empezar por una descomposición de los datos.

Respecto a esta distribución en datos, las características antes mencionadas se concretan en:

- **Flexibilidad.** El tamaño y la cantidad de fragmentos deben ser flexibles.
- **Eficiencia.** Los fragmentos deben generar volúmenes de datos que equilibren la carga entre las diferentes tareas.
- **Simplicidad.** Para facilitar la administración y depuración de los datos, la distribución de estos debe ser lo más simple posible.

B. PASO 2. DISEÑO DEL ALGORITMO

Existen diferentes patrones de diseño según el tipo de descomposición realizada y según también cómo se organicen las estructuras de datos.

La organización de las estructuras de datos puede ser:

- **Geométrica** (vectores y matrices), en cuyo caso podemos realizar una descomposición basada en filas, columnas o bloques.
- **Recursiva**, cuando tratamos con listas, árboles o grafos.

Así pues, la distribución que hayamos hecho y la organización de las estructuras de datos determinarán el patrón de diseño a seguir, para lo cual podemos aplicar el siguiente árbol de decisión:

Veamos los diferentes patrones mencionados aquí:

- **Patrón Task Parallelism.** Las tareas en que dividimos el problema se pueden realizar de forma concurrente.
- **Patrón Divide and Conquer.** Se aplica cuando el problema original se resuelve de forma recursiva, de modo que podemos o paralelizar las hojas del árbol de recursión o el árbol de recursión completo.
- **Patrón Geometric Decomposition.** Se aplica cuando tenemos estructuras de datos lineales y distribuimos las tareas de modo que se encarguen de diferentes subpartes de esta descomposición.
- **Patrón Recursive Data.** Se aplica con estructuras de datos recursivas, para el tratamiento de las cuales se aplica también el patrón de divide y vencerás.
- **Patrón Pipeline.** Consiste en el procesamiento por etapas de los datos a tratar, de modo que las salidas de una etapa se toman como entrada en la siguiente.
- **Patrón Event Based Coordinator.** No se trata de una secuencia de etapas, como en el caso del Pipeline, sino que puede haber ciclos entre ellas.

Estructuras de soporte

Para el diseño del algoritmo concurrente se pueden aplicar varios esquemas de programación, que no tienen por qué ser excluyentes entre ellos:

- **SPMD (Single Program, Multiple Data)**, donde las unidades de ejecución (de momento, en nuestro caso, procesos) ejecutan el mismo programa en paralelo, pero sobre diferentes datos.
- **Master/Workers**, donde un proceso maestro crea un conjunto de subprocesos para que realicen las diferentes subtareas.
- **Loop Parallelism.** Se aplica cuando tenemos bucles de cómputo intensivo, de manera que cada iteración se puede realizar en paralelo.
- **Fork/Join.** Se aplica cuando un proceso principal crea procesos hijos y debe esperar a que estos finalicen.

C. PASO 3. IMPLEMENTACIÓN DEL ALGORITMO

Para la implementación del algoritmo debemos tener en cuenta la tecnología con la que contamos: por una parte, el lenguaje de programación que dé soporte a la programación concurrente y, por otra, el hardware. En líneas generales, deberemos determinar:

- Qué unidades de ejecución (UE) tenemos disponibles: procesos o hilos.
- Qué mecanismos de sincronización ofrecen: memoria compartida, exclusión mutua, etc.
- Cómo se realiza la comunicación entre los procesos: paso de mensajes, E/S, etc.