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:

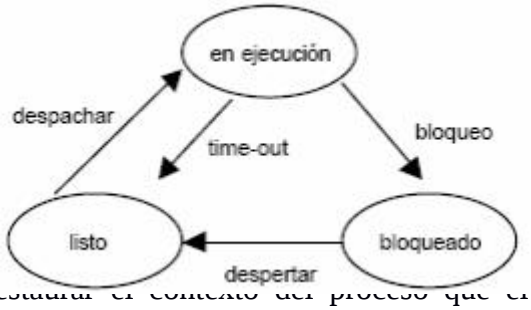
- **Programación Concurrente:** Un único procesador (multiprogramación). En este caso, aunque para el usuario parezca que varios procesos se ejecutan al mismo tiempo, si solamente existe un único procesador, solamente un proceso puede estar en un momento determinado en ejecución. Para poder ir cambiando 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. 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.
- **Programación multitarea:** 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. Se puede utilizar conjuntamente con la programación concurrente, permitiendo al mismo tiempo multiprogramación.
- **Programación distribuida**: 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.

La parte central que realiza la funcionalidad básica del sistema operativo se denomina kernel. 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.
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.

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. Los procesos se pueden encontrar en distintos estados:

- En ejecución-Bloqueado: se pasa de un estado a otro cu
  evento externo
- Bloquedo-Listo: pasa de bloqueado a listo cuando pasa
- Listo-en Ejecución: cambiará de estado cuando el siste
- En Ejecución-Listo: cuando se acaba el tiempo asignad



Cuando el procesador pasa a ejecutar otro proceso, lo cual
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.

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. 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.

Cuando se crea un nuevo proceso, se utiliza la operación **create**. Hay 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**.
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.
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. Al terminar la ejecución de un proceso mediante la operación **exit**, 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. El padre puede terminar la ejecución de un proceso hijo cuando crea conveniente, para ello puede utilizar la operación **destroy**.

Por simplificación y portabilidad, evitando así depender del sistema operativo sobre el cual se esté ejecutando vamos a explicar la gestión de procesos para la máquina virtual de Java (Java Virtual Machine, que 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. 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.

Los hilos son secuencias de codigo dentro del contexto de un proceso. Los hilos siempre van a tener la supervisión de un proceso padre. Por eso los hilos comparten el espacio de memoria, o lo que es lo mismo, corren dentro del contexto del programa, al contrario que los procesos, que cada uno tendrá su contexto específico.
Los hilos se suelen utilizar por procesos que tengan que realizar tareas de manera simultanea. Por ejemplo, en un coche todos los sensores deben estar trabajando al mismo tiempo y por tanto cada sensor puede tener un hilo corriendo. En un servidor web, se podría tener un hilo para atender peticiones e ir generando un hilo nuevo por cada cliente al que haya que servir.

Ejercicios sobre procesos

En el curso vamos a utilizar Eclipse como entorno integrado de desarrollo (IDE). En esta primera práctica, sin embargo, vamos a utilizarlo para analizarlo como aplicación escrita en Java que es.

1. Lanza Eclipse.
2. Utilizando las herramientas del sistema, mira cuántas hebras tiene lanzadas.
   - En GNU/Linux, **usa ps -eO nlwp** (la O es una o mayúscula)
   - En Windows, lanza el administrador de tareas, y activa la columna #Subprocesos#

```
7640    1 S pts/2    00:00:00 bash
7686    1 S ?        00:00:00 [kworker/u8:2]
7688    1 S ?        00:00:00 [kworker/1:0]
7697    1 S ?        00:00:00 [kworker/0:1]
7701    1 S ?        00:00:00 [kworker/u8:0]
7714    1 S pts/3    00:00:00 bash
7732    1 S ?        00:00:00 [kworker/1:3]
7739    1 S pts/3    00:00:00 sudo ps -eO nlw
7740    1 R pts/3    00:00:00 ps -eO nlwp
elkin@PC-Elkin:/$ sudo ps -eO nlwp
```

3.

Observa que el número de hebras lanzadas es significativo.

En Linux, podemos ver el estado de cada hebra (si se está o no ejecutando) usando: **ps aux -L**

```
elkin@PC-Elkin:/$ sudo ps aux -L
USER         PID   LWP %CPU NLWP %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1     1  0.0    1  0.0 139192  7060 ?        Ss   00:49   0:01 /sbi
root           2     2  0.0    1  0.0      0     0 ?        S    00:49   0:00 [kth
root           3     3  0.0    1  0.0      0     0 ?        S    00:49   0:00 [kso
root           5     5  0.0    1  0.0      0     0 ?        S<   00:49   0:00 [kwo
root           7     7  0.0    1  0.0      0     0 ?        S    00:49   0:01 [rcu
root           8     8  0.0    1  0.0      0     0 ?        S    00:49   0:00 [rcu
root           9     9  0.0    1  0.0      0     0 ?        S    00:49   0:00 [mig
root          10    10  0.0    1  0.0      0     0 ?        S<   00:49   0:00 [lru
   ...
elkin      7865  7867  0.0    3  0.0 371792  7164 ?         Sl   02:08   0:00 /usr
root       7886  7886  0.0    1  0.0      0     0 ?        S    02:09   0:00 [kwo
root       7887  7887  0.1    1  0.0      0     0 ?        S    02:09   0:00 [kwo
root       7899  7899  0.0    1  0.0      0     0 ?        S    02:10   0:00 [kwo
root       7909  7909  0.0    1  0.0  55480  3792 pts/3    S+   02:11   0:00 sudo
root       7910  7910  0.0    1  0.0  38308  3392 pts/3    R+   02:11   0:00 ps a
elkin@PC-Elkin:/$ sudo ps aux -L
```

ax : muestra los procesos de todos los usuarios, incluso los que no estén asociados a ningún terminal.

u : muestra columnas adicionales, como el usuario al que pertenece el proceso, tiempo de ejecución, etcétera.

-L : muestra las hebras de manera independiente. También se puede usar **-m**

En la columna STAT puedes ver el estado de cada hebra. Algunas de las letras que pueden aparecer para indicar el estado son:

- D : uninterruptible sleep
- R : (normalmente por E/S) en ejecución, o preparado a la espera de procesador.
- S : interruptible sleep (esperando a que ocurra algún evento).
- Z : zombie

También pueden aparecer otras con información extra:

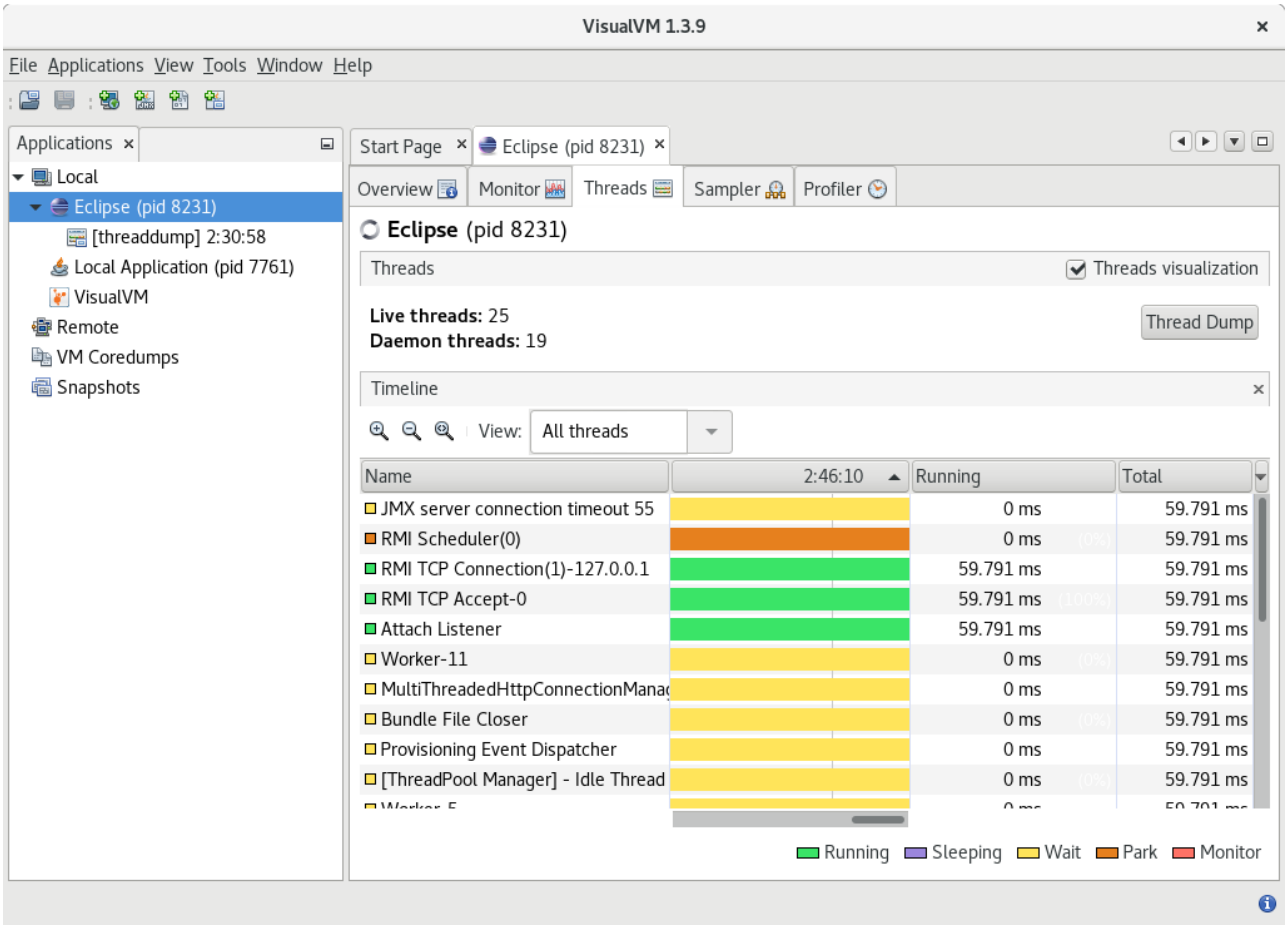
- s : lider de sesión.
- l : multihebra.
- + : proceso en primer plano (en su terminal).
- < : proceso con prioridad alta.
- N : proceso con prioridad baja.
- L : proceso con páginas bloqueadas en memoria.

La ejecución de Eclipse hace uso de multitud de hebras.

La JVM de Sun/Oracle, soporta herramientas de monitorización y auditoría externa para analizar su funcionamiento. Vamos a utilizarlas para ver esas hebras.

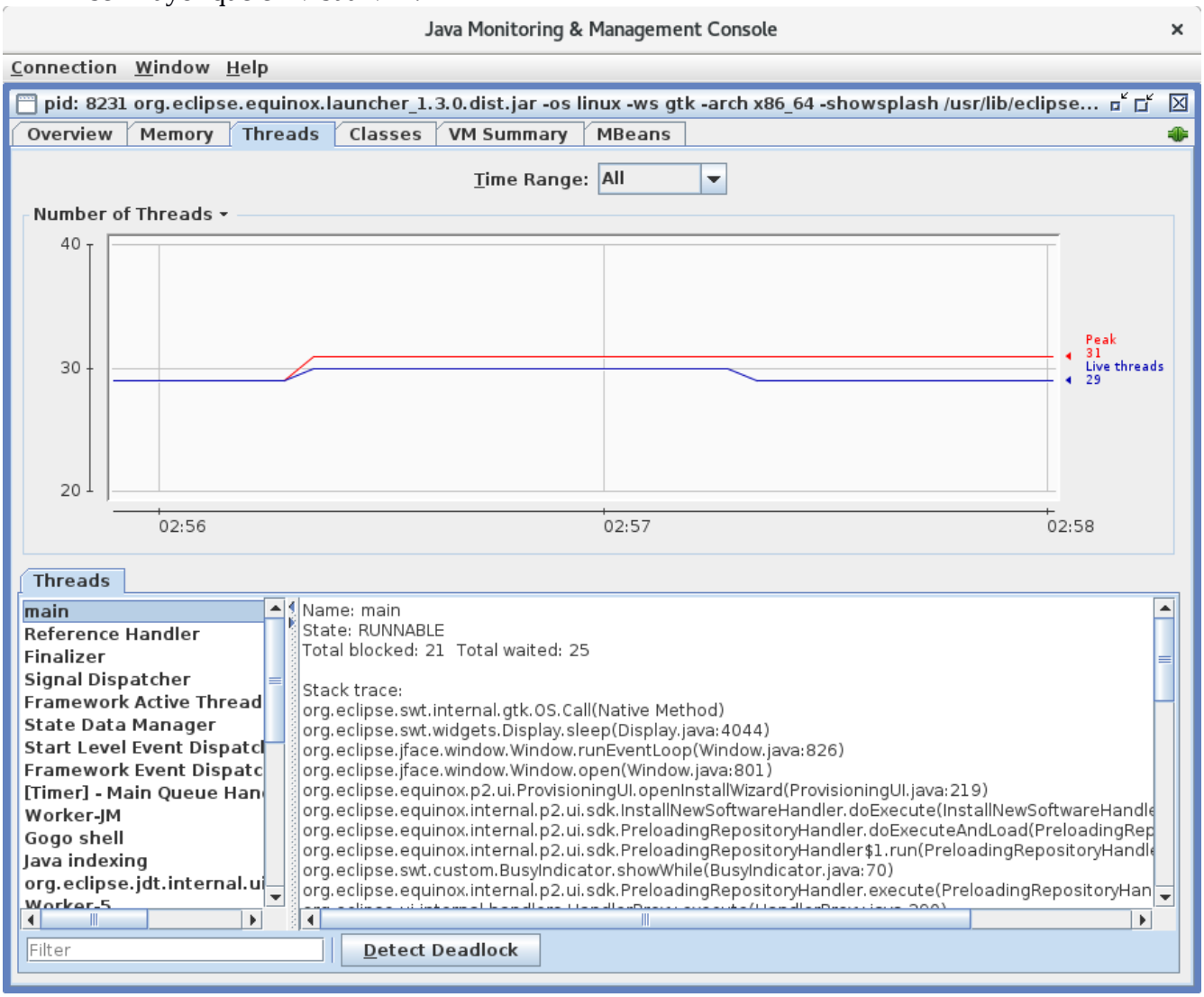
1. Con Eclipse aún lanzado, ejecuta VisualVM (visualvm en Linux).
2. En el lado izquierdo de la ventana verás los #procesos Java# lanzados localmente. También podríamos conectarnos a máquinas remotas y analizar los procesos Java que tenga lanzados que admitan monitorización remota.

3. Verás dos procesos: eclipse, y el propio VisualVM. Haz doble clic sobre el de Eclipse, y selecciona la pestaña Threads

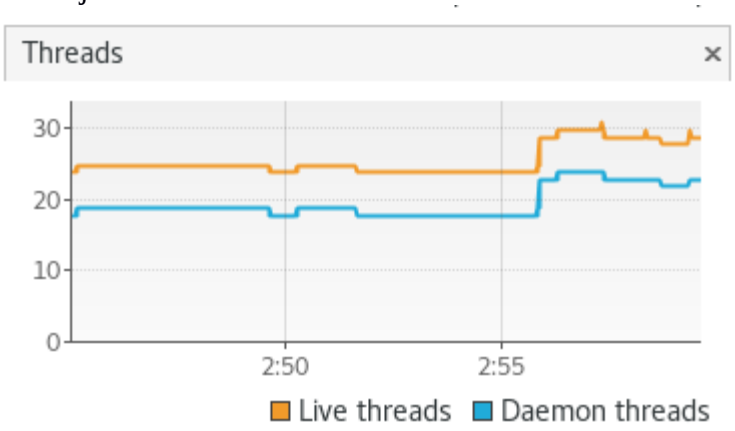


4. VisualVM nos muestra información interesante sobre las hebras, incluyendo su nombre (Java).
5. Observa los posibles estados de las hebras: running , sleeping , wait , monitor y, dependiendo de la versión, park.
6. Compara el número de hebras mostradas por VisualVM y por el sistema. Verás que no es igual. En este ejemplo 36 frente a 25.

7. Vamos a utilizar una herramienta de monitorización distinta para ver si nos muestra todas las hebras. Sin cerrar VisualVM, lanza jconsole. Como antes, selecciona el proceso asociado a Eclipse, y ve a la pestaña Threads . De nuevo, el número total no es el mismo, aunque parece ser mayor que en VisualVM.



8. Vuelve a VisualVM y mira el número de hebras. Verás que ha crecido y que coincide con el indicado por jConsole. Ve a la pestaña Monitor para ver el número de hebras en una gráfica temporal. Fíjate que el número ha subido, justo en el instante en el que comenzamos la monitorización desde jConsole.



9. Cierra jConsole. En VisualVM observa que el número de hebras no baja, al menos inmediatamente. Si eres paciente, verás que naturalmente las hebras adicionales desaparecen.
10. jConsole tampoco muestra todas las hebras. Ejecuta: **jstack <*pid*>** sustituyendo pid por el identificador del proceso de Eclipse.
11. Para cada hebra, nos muestra información adicional, como la pila de llamadas actual. Cuenta el número de hebras y comprueba que hay hebras adicionales.

PROCESOS CON JAVA

Java dispone en el paquete **java.lang** de varias clases para la gestión de procesos.

- **ProcessBuilder:** Cada instancia **ProcessBuilder** gestiona una colección de atributos del proceso. El método **ProcessBuilder.start()** crea una nueva instancia. 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(). Puede ser invocado varias veces desde la misma instancia para crear subprocesos con atributos idénticos o relacionados.
- **Runtime:** 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. Se pueden dar problemas porque no encuentra el ejecutable en la ruta indicada, no tener permisos de ejecución, no ser un ejecutable válido en el sistema…

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

```
import java.io.IOException;
public static void main(String[] args) throws IOException {…
catch(InterruptedException e){…
```

El proceso hijo realizará su ejecución completa terminando y liberando sus recursos al finalizar, cuando el hijo realiza la operación exit para finalizar su ejecución. Un proceso puede terminar de forma abrupta un proceso hijo que creó mediante el método destroy.

Por ejemplo, para ejecutar el comando DIR de DOS usando estas dos clases escribimos lo siguiente, indicando en el constructor de ProcessBuilder los argumentos del proceso que se quiere ejecutar como una lista de cadenas separadas por comas, y después usamos el método strart() para iniciar el proceso.

```
ProcessBuilder proceso = new ProcessBuilder ("CMD" , "/C " ,"DIR " );
Process hijo = proceso.start();
```

METODOS CLASE PROCESS

InputStream getInputStream ( )

Devuelve el flujo de entrada conectado a la salida normal del subproceso. Nos permite leer el stream de salida del subproceso, es decir, podemos leer lo que el comando que cjecutamos escribió en la consola.

int waitFor( )

Provoca que el proceso actual espere hasta que el subproceso representado por el objeto Process finalice. Devuelve 0 si ha finalizado correctamente.

InputStream getErrorStream()

Devuelve el flujo de entrada conectado a la salida de error del subproceso. Nos va a permitir poder leer los posibles errores que se produzcan al lanzar el subproceso.

OutputStream getOutputStream()

Devuelve el flujo de salida conectado a la entrada normal del subproceso. Va a permitir escribir en el stream de entrada del subproceso, así podemos enviar datos al subproceso que se ejecute.

void destroy( )

Elimina el subproceso.

int exitValue ( )

Devuelve el valor de salida del subproceso.

boolean isAlive ()

Comprucba si el subproceso representado por Process está vivo

Por defecto cl proceso que se crea (o subproceso) no tiene su propia terminal o consola. Todas las operaciones de E/S serán redirigidas al proceso padre, y se puede acceder a ellas usando getOutputStream( ), getlnputStream( ) y getErrorStream( ). El proceso padre utiliza estos flujos para alimentar la entrada y obtener la salida del subproceso.En algunas plataformas se pueden producir bloqueos en el subproceso debido al tamaño de búfer limitado para los flujos de entrada y salida estándar.

Cada constructor de ProcessBuilder gestiona los siguientes atributos de un proceso:

- Un comando. Es una lista de cadenas que representa el programa que se invoca y sus argumentos si los hay.
- Un entorno (environment) con sus variables.
- Un directorio de trabajo. El valor por defecto es el directorio de trabajo del proceso en curso (System.getenv( ))
- Una fuente de entrada estándar. Por defecto, el subproceso lee la entrada de tubería. El código Java puede acceder a esta tubería a través de la secuencia de salida devuelta por Process.getOutputStream( ). Sin embargo, la entrada estándar puede ser redirigida a otra fuente con redirectInput( ). En este caso Process.getOutputStream( ) devolverá una secuencia de salida nulo. Lo mismo pasará con el resto de flujos.

METODOS CLASE PROCESSBUILDER

ProcessBuilder command (List<String>)

Devuelve un objeto con los parámetros para ejecutar el programa. Se puede pasar como parámetro también un String

List<String> command ( )

Devuelve los argumentos que tiene el ProcessBuilder

Map<String, String> enviroment ( )

Devuelve en una estructura de tipo Map las variables de entorno del objeto ProcessBuilder

ProcessBuilder redirectError (File)

Redirige la salida de error estándar a un fichero

ProcessBuilder redirectOutput (File)

Redirige la salida estándar a un fichero

ProcessBuilder redirectInput (File)

Establece la entrada estándar en un fichero

File directory ( )

Devuelve el directorio de trabajo del objeto ProcessBuilder

File directory ( )

Devuelve el directorio de trabajo del objeto ProcessBuilder

ProcessBuilder directory (File)

Establece el directorio de trabajo del objeto ProcessBuilder

Process start ( )

Devuelve un proceso en base a los atributos del objeto ProcessBuilder

Ejemplo: Ejecutar un programa que muestre el editor de texto

```
import java.io.*;

public class Clase1{
	public static void main(String[] args) throws IOException {
		// se crea el objeto processBuilder usando los comandos
		ProcessBuilder paramProceso = new ProcessBuilder("gedit");
		// se crea el proceso en base al objeto processBuilder
		Process proceso = paramProceso.start();
		//Process proceso = new ProcessBuilder("gedit").start();
	}
}
```

Modifica el programa anterior para que pasando un argumento ejecute el programa que nos interese

Ejemplo: listar el contenido del directorio \home\elkin haciendo que el directorio de trabajo del proceso sea \etc

```
import java.io.*;

public class Clase1{
	public static void main(String[] args) throws IOException {
		String strLinea;//para leer del buffer de salida del proceso
		int intDevuelveProceso; //valor que devuelve al terminar el proceso
		// estos son los comandos a ejecutar, seria:
	  // usuario@usuario-pc:~$ ls /home/elkin
		final String strComandos[] = {"ls", "/home/elkin"};
```

```
            // se crea el objeto processBuilder usando los comandos
            ProcessBuilder paramProceso = new ProcessBuilder(strComandos);
            // donde ruta = la carpeta del ejecutable
            paramProceso.directory(new File("/etc"));

            // se crea el proceso en base al objeto processBuilder
            Process proceso = paramProceso.start();

            // Se lee la salida
            InputStream is = proceso.getInputStream();
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);

            while ((strLinea = br.readLine()) != null)
              System.out.println(strLinea);

            // Esperamos que el proceso termine
            try {
                  intDevuelveProceso = proceso.waitFor();
                  System.out.println("\nCódigo de salida: "+ intDevuelveProceso);
            } catch (InterruptedException e) {
                  e.printStackTrace(System.err);
            }
      }
}

Mismo ejemplo con la clase Runtime
import java.io.*;

public class Clase1{
      public static void main(String[] args) throws IOException {
            String strLinea;//para leer del buffer de salida del proceso
            String[] arrStr={"/home/elkin"};
            try
        {
            // Se lanza el ejecutable.
            Process proceso = Runtime.getRuntime().exec ("ls", arrStr, new
File("/etc"));

            // Se obtiene el stream de salida del programa
            InputStream is = proceso.getInputStream();

            // Se prepara un bufferedReader para poder leer la salida más
            // comodamente.
            BufferedReader br = new BufferedReader (new InputStreamReader (is));


            // Mientras se haya leido alguna linea
            while ((strLinea = br.readLine())!=null)
                // Se escribe la linea en pantalla
                System.out.println (strLinea = br.readLine());
        }
        catch (Exception e)
        {
```

```
            // Excepciones si hay algún problema al arrancar el ejecutable o al
            // leer su salida.
            e.printStackTrace();
        }
      }
}

mismo ejemplo con captura de los errores del programa
import java.io.*;

public class Clase1{
      public static void main(String[] args) throws IOException {
            String strLinea;//para leer del buffer de salida del proceso
            int intDevuelveProceso; //valor que devuelve al terminar el proceso
            // estos son los comandos a ejecutar, seria:
          // usuario@usuario-pc:~$ ls /home/elkin
            final String strComandos[] = {"ls", "/home/elkign"};


            // se crea el objeto processBuilder usando los comandos
            ProcessBuilder paramProceso = new ProcessBuilder(strComandos);
            // donde ruta = la carpeta del ejecutable
            //paramProceso.directory(new File("/etc"));
            try{
                  // se crea el proceso en base al objeto processBuilder
                  Process proceso = paramProceso.start();

                  // Se lee la salida
                  InputStream is = proceso.getInputStream();
                  InputStreamReader isr = new InputStreamReader(is);
                  BufferedReader br = new BufferedReader(isr);

                  while ((strLinea = br.readLine()) != null)
                    System.out.println(strLinea);

                  //si ha ocurrido un error se muestra el error
                  is = proceso.getErrorStream();
                  isr = new InputStreamReader(is);
                  br = new BufferedReader(isr);

                  while ((strLinea = br.readLine()) != null)
                    System.out.println(strLinea);

                  // Esperamos que el proceso termine
                  intDevuelveProceso = proceso.waitFor();
                  System.out.println("\nCódigo de salida: "+ intDevuelveProceso);
            }
            catch (InterruptedException e) {
                  e.printStackTrace(System.err);
            }
      }
}
```

12. Vamos a pasarle parámetros al proceso con el método getOutputStream y escribiendo luego en el stream

```
import java.io.IOException;
import java.io.OutputStream;
import java.io.InputStream;

public class Flujos {

	public static void main(String[] args) throws IOException{
		int intAux;//lo uso para leer del stream de salida y para ver la
salida de la consola
		//en este caso he abierto el procedimiento indicando el proceso
		Process proceso = new
ProcessBuilder().command("/bin/bash").start();//en windows con cmd

		//le paso el resto de parámetros al ftp, es importante no olvidar
los \n
		OutputStream os = proceso.getOutputStream();
		os.write("ftp ftp.rediris.es\n".getBytes());//servidor
		os.write("anonymous \n".getBytes());		//usuario anónimo
		os.write("passive \n".getBytes());		//modo pasivo
		os.write("ls \n".getBytes());			//ver los ficheros
		os.write("bye\n".getBytes());			//salirse de ftp
		os.flush();
		os.close();							//cierro el
stream de salida

		InputStream is = proceso.getInputStream();

		while ((intAux = is.read()) != -1)
		  System.out.print((char)intAux);
		is.close();

		try{
			intAux=proceso.waitFor();
			System.out.print("valor de salida " + intAux);

		}
		catch (InterruptedException ie) {
			ie.printStackTrace();
		}
	}
}
```

13. En base al ejercicio anterior, crea un programa que me permita ejecutar comandos de consola

14. Crea una clase llamada sumar 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 dos números por su entrada estándar, e imprimirá por su salida estándar la suma de los mismos. 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: 2ENTER 4ENTER **9** 12ENTER 16ENTER **70** 125ENTER 130ENTER **765**