2025

UNIDAD 1. PROGRAMACIÓN MULTIPROCESO

2ºDAM. PROGRAMACIÓN DE SERVICIOS Y PROCESOS

FRAN HUERTAS

Contenido

1. PROGRAMAS Y PROCESOS.

Un **programa** es un archivo que contiene instrucciones que se pueden ejecutar directamente en una máquina. La máquina puede ser física (hardware) o virtual, por ejemplo, Java. El programa se considera un objeto estático que se encuentra almacenado en memoria secundaria, como puede ser el disco duro.

Un **proceso** corresponde a una instancia de un programa en ejecución. Un proceso es un objeto dinámico.

La ejecución de un programa (creación de un proceso) se inicia con la carga de éste en memoria (cargador del SO). Esto implica el uso de diversos recursos:

Memoria principal. Espacio de memoria continua donde se almacenan las instrucciones de programa y los datos que requiera. Durante la ejecución el espacio de memoria puede variar de forma dinámica (reserva y/o liberación)

Procesador o CPU. Ejecuta el proceso una vez cargado en memoria. El uso de la CPU depende del planificador de procesos que decide cuándo un proceso toma la CPU. Para controlar la secuencia de ejecución del proceso se usa el PC (Contador de Programa) que es un registro del procesador donde se almacena la dirección de memoria de la instrucción que se está ejecutando. Los datos necesarios para la ejecución de cada instrucción deben almacenarse en los registros del procesador (se traen de la memoria a los registros del procesador). La ALU (Unidad Aritmético Lógica) ejecuta las operaciones indicadas en la instrucción. Los resultados obtenidos se almacenan en la memoria principal (en el espacio reservado al proceso).

La UC (unidad de control) es la encargada de secuenciar las instrucciones, es decir, decidir qué instrucción es la siguiente a ejecutar. Para ello, carga en el PC la dirección de memoria de la siguiente instrucción.

Dispositivos de E/S. Los procesos pueden compartir los dispositivos de E/S.

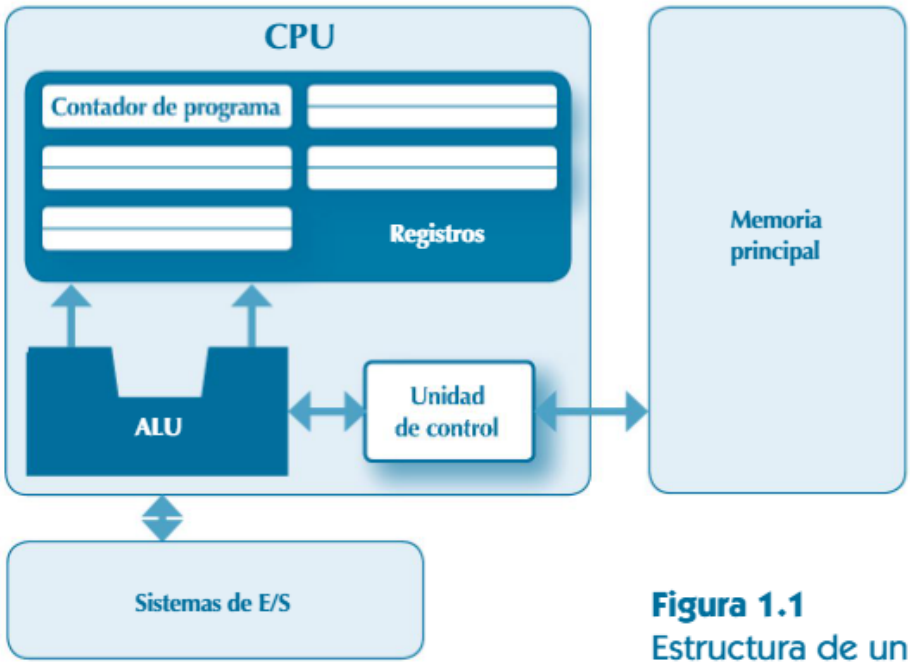


Figura 1.1
Estructura de un procesador.

Cada proceso lleva asociado un conjunto de datos que lo identifican de forma única llamado **PCB** (bloque de control de proceso)

Durante su ejecución, un proceso, puede crear otros procesos y éstos a su vez nuevos procesos. Creando así una jerarquía de procesos.

Para ver la jerarquía de procesos:

- Linux: Pstree
- Windows: Process Explorer.

2. MULTITAREA

Los sistemas operativos modernos pueden ejecutar varios procesos *a la vez*. Un usuario puede editar un fichero, imprimir un documento, escuchar música y copiar archivos en la memoria USB, de manera *simultánea*. A pesar de contar con un solo procesador. Todo esto se consigue gracias a que el SO es multitarea.

El procesador es el componente hardware encargado de ejecutar las instrucciones que conforman los programas. En un equipo con un único procesador ***mononúcleo*** y en un instante de tiempo determinado, **sólo se puede ejecutar una instrucción**, por lo que no existe posibilidad de que haya una ejecución paralela. No es posible para un núcleo del procesador ejecutar simultáneamente varias instrucciones de un programa.

Sin embargo, en este mismo sistema, es posible simular una ***ejecución concurrente***, de manera que el usuario tenga la impresión de que se ejecutan múltiples programas simultáneamente. Esta multitarea simulada la consigue el sistema operativo, aplicando un **algoritmo de planificación de tareas (Round-Robin y/o colas de prioridad, multitarea colaborativa)**, encargado de repartir el tiempo de procesador entre las distintas aplicaciones en ejecución.

En esta situación, el procesador sigue ejecutando una única instrucción en un momento dado, pero se reparte el tiempo de ejecución que dedica a cada programa. El componente del sistema operativo encargado de realizar la asignación de tiempos a cada programa es el **planificador de tareas** o ***task scheduler***.

Por otro lado, en equipos con **múltiples procesadores** (como suele ocurrir en los servidores) y en equipos con **procesadores multinúcleo**, sí existe la ejecución concurrente de aplicaciones o ***multitarea real***, puesto que en un mismo instante de tiempo cada procesador (o cada núcleo de procesador) puede estar ejecutando más de una instrucción distinta, de forma simultánea.

En los equipos actuales, que suelen ser multinúcleo, se da una **combinación de multitarea real y simulada**: aunque cada procesador o núcleo puede ejecutar una instrucción al mismo tiempo, normalmente hay un número superior de programas (procesos) en ejecución que el número de procesadores y/o núcleos presentes en el sistema. Por tanto, el sistema operativo debe repartir los programas a ejecutar entre los procesadores o núcleos, al tiempo que arbitra un algoritmo de asignación de tiempo de procesador a cada uno de los procesos.

CPU	A	A	A	A	A	B	B	B	B	C	C	C	C	C	C
Tiempo →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figura 1.2
Ejecución de procesos sin multitarea.

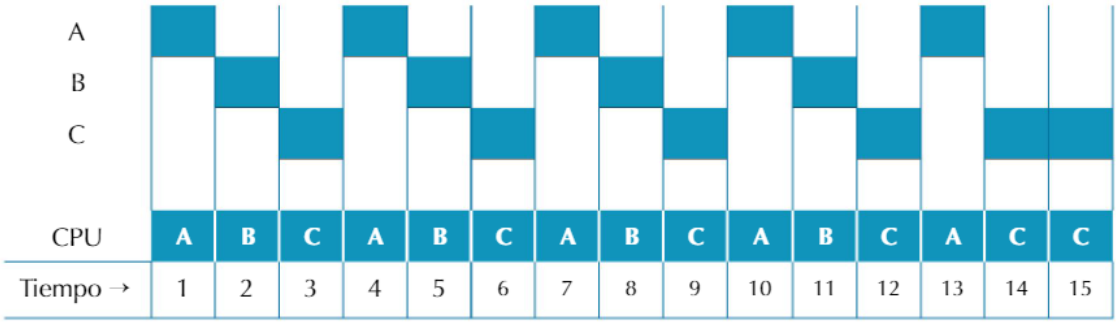
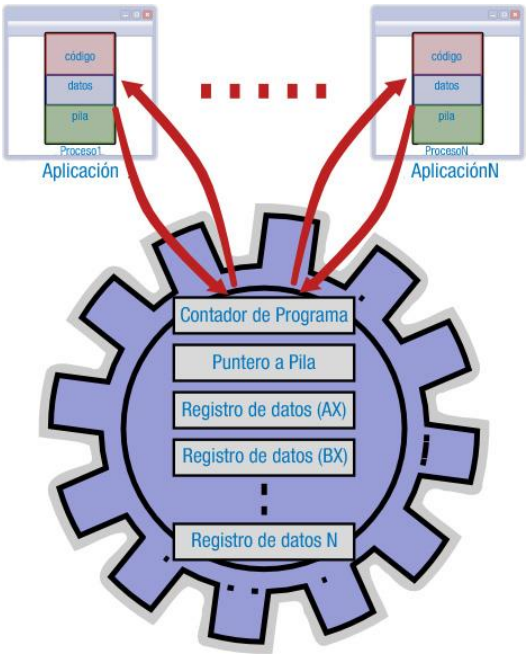


Figura 1.3
Ejecución de procesos con multitarea.

En la siguiente figura se muestra un ejemplo de multitarea colaborativa, donde se aprovechan los tiempos de bloqueo de un proceso para que otro proceso haga uso de la CPU. Para compartir la CPU entre dos o más procesos se requiere realizar una operación llamada **cambio de contexto** que consiste en guardar el estado del proceso actual (PCB) y cargar el del proceso entrante.



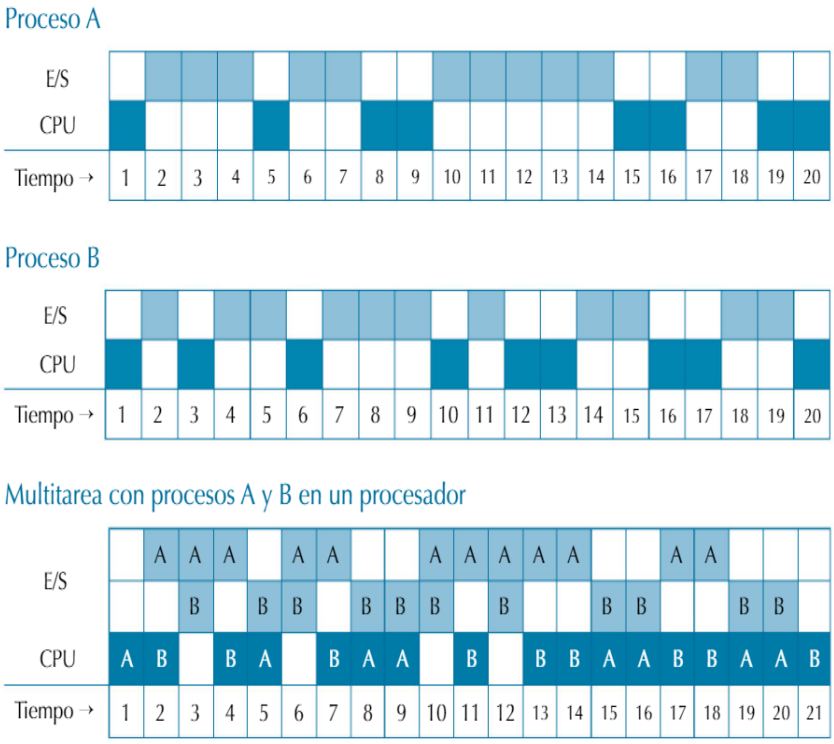


Figura 1.5
Ejecución de procesos con multitarea, con operaciones de E/S.

3. PROCESOS Y SISTEMAS MONOPROCESADORES Y MULTIPROCESADORES.

Un sistema monoprocesador tiene un solo procesador y un multiprocesador tiene varios.

Se denominan **procesos concurrentes** a los que se ejecutan simultáneamente durante un intervalo de tiempo de forma real o simulada.

Se conoce como **programación concurrente** cuando se ejecutan varios procesos concurrentes en un sistema y también cuando se aplican técnicas de programación para desarrollar programas que se van a ejecutar de forma concurrente y se van a comunicar entre sí de manera sincronizada y coordinada.

3.1. SISTEMAS MONOPROCESADOR

La ejecución concurrente de varios procesos en un sistema monoprocesador se conoce como **multiprogramación**.

Los procesadores actuales son multinúcleo desde la aparición del *dual-core* en adelante. A estos sistemas se les puede considerar multiprocesadores aunque solo tengan un solo microprocesador.

3.2. SISTEMAS MULTIPROCESADOR

Se trata de sistemas con más de un procesador. Pueden ser fuertemente acoplados o débilmente acoplados.

La **programación paralela** consiste en la ejecución de varios procesos concurrentes en un sistema multiprocesador (varios procesadores o núcleos), de manera que se pueden ejecutar de una forma simultánea real.

Sistemas fuertemente acoplados. Los procesadores comparten memoria, dispositivos ES y se comunican mediante un bus. Pueden ser simétricos (todos iguales) o asimétricos (uno hace de maestro y el resto de esclavos)

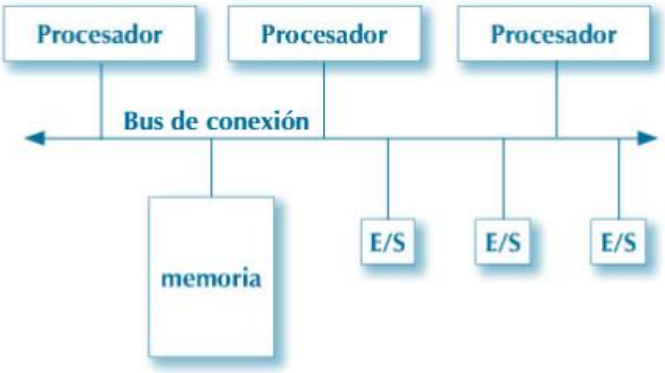


Figura 1.6
Sistema multiprocesador fuertemente acoplado.

Sistemas débilmente acoplados. En estos sistemas no hay memoria compartida ni dispositivos de ES. La comunicación se hace a través de una red de comunicaciones.

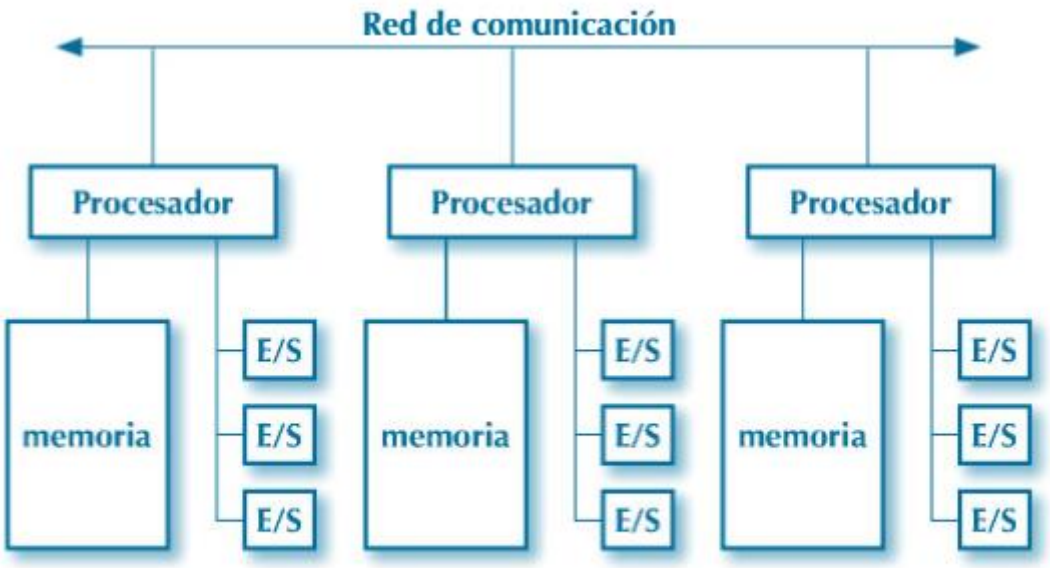


Figura 1.8
Sistema multiprocesador débilmente acoplado.

Otra posibilidad, es que los procesadores estén en ordenadores independientes comunicados mediante una red LAN o WAN. Estos **sistemas** se denominan **distribuidos** y se caracterizan por la heterogeneidad del hardware y la relativa independencia entre sí.

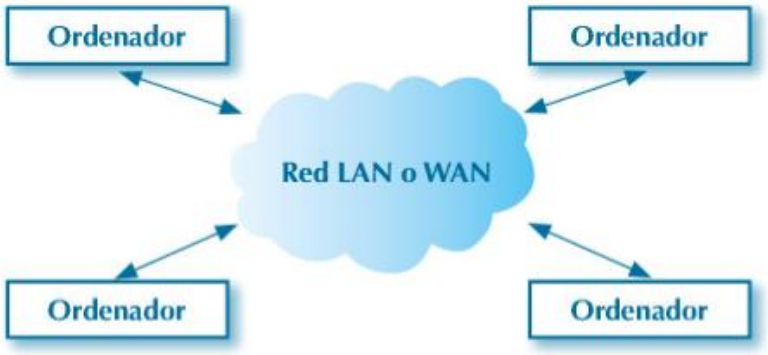


Figura 1.10
Sistema distribuido.

La programación distribuida consiste en la ejecución de varios procesos concurrentes en un sistema distribuido.

Los sistemas distribuidos son altamente escalables y configurables. Por otro lado, son más costosos de mantener y de sincronizar.

4. KERNEL DEL SISTEMA OPERATIVO Y LLAMADAS AL SISTEMA.

La parte central del SO se denomina *kernel* o núcleo. Es una parte pequeña y muy optimizada del SO que da respuesta a multitud de eventos mediante la gestión de interrupciones. Las interrupciones se pueden producir por:

- **Eventos del hardware.** Por ejemplo: pulsación de una tecla, movimiento del ratón, final de lectura o escritura en disco, etc.
- **Llamadas al sistema**. Petición al *kernel* por parte una aplicación para el uso de un servicio o funcionalidad del SO.

- **Interrupciones periódicas**. Las provoca el planificador de procesos a corto plazo del SO (*Scheduler*) para decidir si un proceso continúa usando la CPU o se produce un cambio de contexto.

Al producirse la interrupción el procesador deja de ejecutar el proceso en curso y pasa a ejecutar la rutina de tratamiento de la interrupción. Cuando finaliza dicha rutina, el procesador reanuda la ejecución.

Las rutinas de interrupciones se ejecutan de forma prioritaria, en un modo conocido como supervisor o *kernel*. El resto de procesos se ejecutan en modo usuario.

5. ESTADOS DE EJECUCIÓN DE UN PROCESO

Desde que se inicia un proceso hasta que finaliza puede pasar por varios estados. El SO gestiona los diferentes estados de un proceso durante su ciclo de vida.

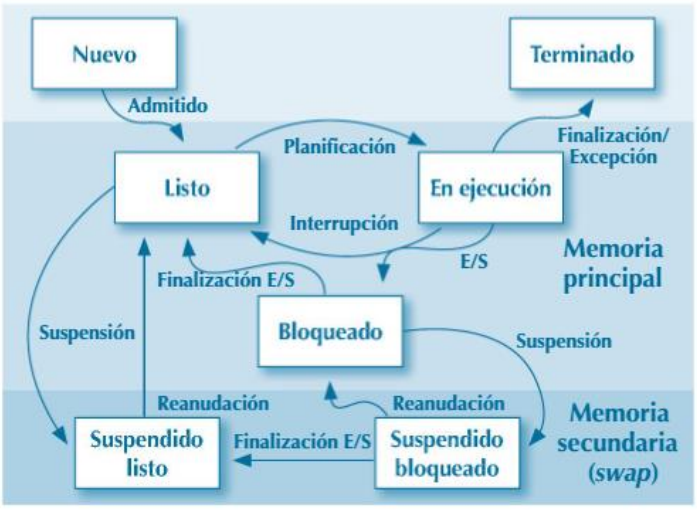


Figura 1.11 Planificación de procesos.

Los cambios de estado de un proceso los determina el SO para optimizar el uso de los recursos del sistema (memoria, CPU y dispositivos de ES). Para ello usa diferentes algoritmos de planificación:

Planificación a largo plazo. Decide qué procesos son admitidos para su ejecución. Estos procesos son cargados en memoria principal y pasan a estado listo.

Planificación a medio plazo. Gestiona el paso de procesos de la memoria principal a secundaria (suspensión) y viceversa (reanudación)

Planificación a corto plazo. Actúa sobre procesos cargados en memoria principal. Su objetivo es repartir el tiempo de CPU para optimizar el uso de la misma. Un proceso en estado de listo puede pasar a estado de ejecución y empezar a usar la CPU. El cambio de estados de procesos funciona mediante interrupciones que se generan periódicamente.

Para la gestión de procesos, es SO utiliza el PCB de cada proceso. El SO mantiene colas de procesos para cada uno de los estados. También colas para cada dispositivo de E/S que tienen operaciones pendientes.

6. HILOS Y PROCESOS

Un proceso (programa en ejecución) tiene asignado un espacio de memoria. Esta operación de reserva de memoria tiene un coste importante para el SO.

Los hilos son conocidos como procesos ligeros. Un proceso en ejecución siempre tiene un hilo inicialmente, pero puede generar más hilos de ejecución. La creación de hilos no requiere reserva de memoria, sino que comparten la memoria del proceso que los inicia. Esto hace más sencillo la comunicación entre ellos, aunque requiere de mecanismos de sincronización para evitar los problemas que pueden darse.

Al finalizar un proceso se finalizan todos los hilos que éste haya creado.

El planificador a corto plazo, gestión de manera independiente los distintos hilos de un mismo proceso.

7. SERVICIOS

Los servicios son un tipo de procesos que ofrecen alguna funcionalidad (servicio) a otros procesos. Se suelen ejecutar en segundo plano (background) y no interactúan con el usuario.

Los servicios están ejecutándose de forma continua. Se suelen iniciar al cargar el SO y disponen de algún mecanismo de información tipo fichero log para registrar los diferentes eventos que se van produciendo.

Los servicios pueden ser usados por procesos del mismo ordenador o por procesos de otros ordenadores. En último caso se suelen comunicar mediante una red de comunicaciones usando los protocolos de red TCP o UDP de la pila TCP/IP. Este tipo de servicios representan un parte fundamental en los sistemas distribuidos.

Para optimizar los tiempos de respuesta a peticiones de un servicio se suele crear un hilo por cada petición o mantener un pool de hilos para asignar de forma inmediata a cada petición. A los servidores (ofrecen servicios) de este tipo se les conoce como multihilo.

Los servicios en Windows los podemos ver ejecutando *services*

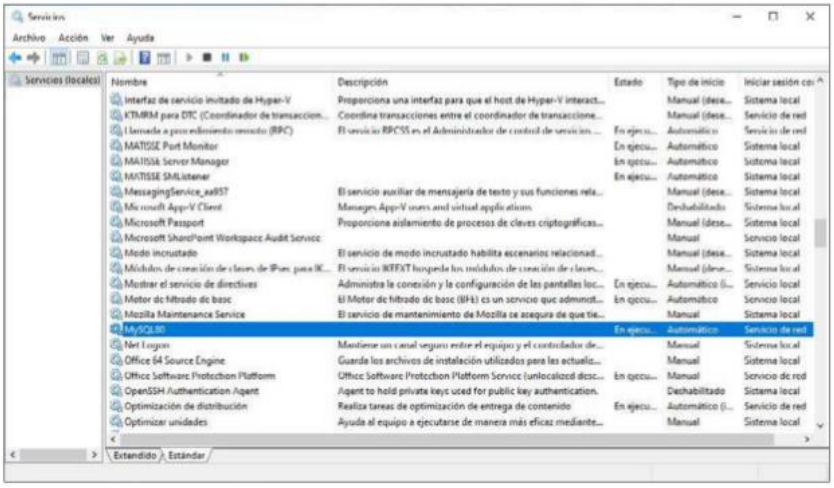


Figura 1.12
Administración de servicios de Windows.

En Linux podemos usar el comando *systemctl –type=service*

8. GESTIÓN DE PROCESOS EN JAVA

Las principales clases para trabajar con procesos en Java son:

- `Process.` Es una clase abstracta, y por tanto, no se puede instanciar directamente. Para crear objetos de tipo **Process** tendremos que usar clases derivadas que implementan procesos nativos dependientes del sistema operativo. Tiene métodos para lanzar procesos, obtener información de su estado y controlar su ejecución.

  Referencia completa: Process

- `ProcessBuilder` permite crear procesos además de permitir la configuración del entorno de ejecución de los mismos. Por otro lado, permite redirigir la entrada y salida del proceso.

  Referencia completa: ProcessBuilder

- `Runtime.` Cada programa Java tiene asociado un objeto de esta clase que le permite obtener información del entorno de ejecución e interactuar con él.

  Referencia completa: Runtime

- `ProcessHandle.` Se trata de una interface que representa un manejador de procesos que ya se encuentran en ejecución, facilitando la comunicación con el mismo y la obtención de información.

  Referencia completa: ProcessHandle

En los siguientes apartados se muestran algunos de ejemplos de su utilización.

8.1. INICIAR UN PROCESO

USANDO LA CLASE RUNTIME.

El método **getRuntime()** devuelve la referencia al objeto ligado a la aplicación java. Con este objeto podemos obtener información del proceso en ejecución.

Obtener información del entorno.

```
public class EjemploRuntime {
    public static void main(String[] args) {
        Runtime rt = Runtime.getRuntime();
        System.out.println("Procesadores:" + rt.availableProcessors()
            + " Memoria: " + rt.freeMemory() );
    }
}
```

El método **exec()** permite iniciar un proceso nuevo.

```
import java.io.IOException;

public class Proceso1 {
    public static void main(String[] args) throws IOException {
        Process ps = Runtime.getRuntime().exec("notepad.exe");
        if (ps.isAlive()) System.out.println("Está vivo");
    }
}
```

USANDO LA CLASE PROCESSBUILDER

Para iniciar un proceso, conocida la ruta del ejecutable, es necesario construir un objeto de la clase `ProcessBuilder`. Este objeto se puede configurar con los parámetros con los que se quiere iniciar el ejecutable, como el nombre del ejecutable o la lista de argumentos a pasar al programa. Una vez instanciado, se puede invocar al método `start`, que inicia el ejecutable y devuelve una instancia `Process` que representa a ese ejecutable, tal y como se muestra a continuación:

```
public class Main {
    public static void main(String[] args) {

        try { // La instancia ProcessBuilder puede ser configurada con parámetros de
              // inicio del programa, como la ruta del ejecutable, argumentos y
              // opciones adicionales.
            ProcessBuilder builder = new ProcessBuilder("notepad.exe");
            Process process = builder.start();
            // Bloquea la ejecución del programa en este punto hasta que el proceso
            // finalice
            process.waitFor();
        } catch (Exception ex) {
            System.err.println(ex.getMessage());
        }
    }
}
```

Además de especificar la ruta del ejecutable, las instancias de la clase `ProcessBuilder` permiten especificar opciones adicionales para iniciar el proceso:

- Para especificar argumentos del proceso, se debe utilizar la sobrecarga del constructor que recibe la ruta del ejecutable y una lista de parámetros `varargs`.

- Para especificar el directorio de trabajo, se utiliza el método *directory*.

- Para especificar variables de entorno, se utiliza el método *environment*.

```
import java.io.File;

public class Main2 {
    public static void main(String[] args) {
        try {
            // Especifica el ejecutable (cmd.exe) y los argumentos que se pasan al
            // ejecutable (/C y dir):
            ProcessBuilder builder = new ProcessBuilder("cmd.exe","/C", "dir");
            builder.directory(new File("C:\\TEMP\\"));
            // Especifica una variable de entorno y su valor:
            builder.environment().put("VAR_ENTORNO", "VALOR");
            Process process = builder.start();
        } catch (Exception ex) {
            System.err.println(ex.getMessage());
        }
```

```
    }
}
```

Comparación entre Runtime Y ProcessBuilder

Característica	Runtime.getRuntime().exec()	ProcessBuilder
Manejo de Argumentos	Propenso a errores con espacios. Requiere "parseo".	**Seguro y explícito.** Cada argumento es un String separado.
Directorio de Trabajo	Heredado del proceso padre. No se puede cambiar.	**Totalmente configurable** con el método directory().
Variables de Entorno	Heredadas del proceso padre. No se pueden modificar.	**Totalmente configurables** a través del método environment().
Redirección de E/S	Manual y compleja (obteniendo streams).	**Simple y potente** con métodos como redirectOutput() e inheritIO().
Modelo de Uso	Llamada a un método estático (funcional).	Creación y configuración de un objeto (orientado a objetos).
Recomendación	**Obsoleto y desaconsejado.** Usar solo si es estrictamente necesario por compatibilidad con código muy antiguo.	**El estándar moderno.** Siempre debe ser la primera opción para crear procesos.

8.2. OBTENER INFORMACIÓN DE UN PROCESO EN EJECUCIÓN

Para capturar un proceso que ya está en ejecución y obtener toda la información disponible sobre él, es necesario recurrir al empleo de la clase `ProcessHandle`, que representa un manejador de procesos actualmente en ejecución en el sistema. En esta clase:

- El método estático `allProcesses` devuelve una lista de todos los procesos en ejecución.

- El método estático `current` devuelve una instancia `ProcessHandle` correspondiente al propio programa.

- El método estático `Of` devuelve una instancia `ProcessHandle` que se corresponde con el proceso en ejecución cuyo `PID` coincide con el que se pasa como parámetro del método.

El siguiente ejemplo muestra cómo obtener la colección de procesos actualmente en ejecución:

```
import java.util.stream.Stream;

public class Main2 {
    /**
     * @param args
     */
    public static void main(String[] args) {
        Stream<ProcessHandle> ps =ProcessHandle.allProcesses();
        for (ProcessHandle p: ps.toList()){
            p.
            System.out.println("ID: "+ p.pid());
            System.out.println("Ejecutable: "+p.info().command());
            System.out.println("------------------------");
        }
    }
}
```

Observa que el método **allProcesses** devuelve un tipo de retorno *Stream<T>*. Esta clase *Stream* NO representa un *stream* convencional, como los *streams* del paquete java.io. Por el contrario, la clase **java.util.Stream** representa una colección de elementos de un determinado tipo, que puede ser consultada mediante métodos que representan predicados de consulta sobre esa colección.

En el ejemplo anterior, el método *allProcesses*, devuelve un tipo de retorno *Stream<ProcessHandle>,* lo que quiere decir que devuelve una colección de objetos **ProcessHandle**.

Por otro lado, si queremos capturar un proceso a partir de su nombre, tendríamos que realizar las siguientes operaciones:

```
import java.util.Optional;
import java.util.stream.Stream;

public class Main3 {
public static void main(String[] args)
{
    String procesoBuscado = "notepad.exe";
    // Obtiene una colección de instancias ProcessHandle representando todos los
    // procesos actualmente en ejecución en el sistema, y filtra, sobre esa
    // colección, aquellos procesos cuyo nombre coincide con el almacenado en la
    // variable procesoBuscado.
    Stream<ProcessHandle> procesos = ProcessHandle.allProcesses();
    Optional<ProcessHandle> procesoEnEjecucion = procesos.filter(proceso->pro-
ceso.info().command().isPresent()).filter(proceso->proceso.info().com-
mand().get().contains(procesoBuscado)).findFirst();
    if (procesoEnEjecucion.isPresent())
```

```
    {
        System.out.println("ID del proceso: " + procesoEnEjecucion.get().pid());
        System.out.println("Ruta del ejecutable: " +
        procesoEnEjecucion.get().info().command().orElse("Desconocido"));
    }
    else
    {
        System.out.println("El proceso " + procesoBuscado + " no está en ejecu-
ción.");
    }
    }
}
```

Al igual que antes, hemos obtenido la colección de procesos en ejecución, mediante el método **allProcesses**. Una vez obtenida esta colección, se han aplicado predicados de consulta **filter** para seleccionar sólo los procesos que tienen una ruta de ejecutable definida y que, además, contienen el proceso buscado en esa ruta.

Cada vez que se aplica un predicado *filter*, se obtiene una nueva colección de procesos en la forma de *Stream<ProcessHandle>,* en la que se han filtrado los elementos de acuerdo a la expresión de filtrado. **Cada llamada a estos predicados de consulta devuelve la misma instancia, pero con los cambios aplicados**. De este modo, se puede encadenar la llamada a múltiples predicados, como se muestra en el ejemplo.

Finalmente, de entre todos los posibles procesos que hayan llegado hasta esta etapa del filtrado, se selecciona el primero de ellos, haciendo uso para ello del predicado *findFirst*. Este predicado devuelve un tipo de retorno Optional<T>, que es similar a devolver un tipo T, pero al emplear la utilidad *Optional*, se puede trabajar con más facilidad en los casos en los que la instancia T es null.

Una vez capturada la instancia **ProcessHandle** correspondiente al proceso que nos interesa, basta con consultar sus métodos y atributos para obtener la información disponible. Entre la información que obtendremos de esa instancia, podremos encontrar aspectos tan variados como el nombre y PID, su tiempo de ejecución, el usuario que lo ha iniciado o la ruta del ejecutable, entre otros aspectos.

8.3. FINALIZACIÓN DE PROCESOS Y CÓDIGOS DE ESTADO

Si hemos iniciado un proceso con la clase **Process**, tenemos la opción de esperar a su finalización a través del método *waitFor* de esa clase.

Sin embargo, si capturamos un proceso en ejecución con *ProcessHandle*, no dispondremos de un mecanismo directo para esperar a la finalización del proceso. Por el contrario, será necesario implementar manualmente alguna técnica de espera.

Para forzar la finalización de un proceso iniciado con la clase Process, sin esperar a que éste finalice normalmente, se dispone de los métodos **destroy** y **destroyforcibly**. El segundo realiza una terminación forzosa e inmediata del proceso.

Por otra parte, todos los procesos finalizan con un código de estado que indica las condiciones en las que ha finalizado el proceso. El método **exitValue** de la clase **Process** devuelve el código de estado con el que finaliza el proceso. También puede obtenerse este valor a partir del método waitFor. Sin embargo, si hemos capturado un proceso con ProcessHandle, no disponemos de una forma directa de obtener el código de estado del proceso, teniendo que implementar manualmente una solución.

Por convenio, se asume que un código de estado 0 indica una finalización correcta del proceso, mientras que un valor distinto de cero indica una finalización anormal (forzada) o con errores.

En este ejemplo se fuerza la finalización del proceso. Una finalización forzada puede generar una excepción en el camino de ejecución del método, por lo que siempre devuelve un código de estado distinto de cero:

```
import java.io.IOException;


public class Program {
    public static void main(String[] args) throws InterruptedException {
        try {
            Process proceso = new ProcessBuilder("notepad.exe").start();
            // Invoca la finalización de un proceso. El proceso debe haber sido
            // programado para responder a esta petición de finalización
            //proceso.waitFor();
            proceso.destroy();

            System.out.println("Proceso detenido con código de salida: " + proceso.exitValue());
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

En la ejecución de este programa observamos que el código de finalización del proceso da un valor distinto de 0 al ser forzada la destrucción del mismo. Sin embargo, si esperamos a que el proceso finalice de una forma ordenada (por la acción del usuario), mediante el método **waitFor**() observaremos un código de finalización 0.

Existe una versión más moderna y segura de waitFor, que permite esperar durante un tiempo y si el proceso no acaba, continúa el hilo de ejecución.

waitFor(long timeout, TimeUnit unit)

8.4. COMUNICACIÓN ENTRE PROCESOS

Los procesos tienen básicamente dos formas de comunicarse:

- **Intercambio de mensajes.** Consiste en enviar y recibir mensajes por un canal.
- **Recursos (o memoria) compartidos.** Se trata de escribir y leer datos en un recurso.

El intercambio de mensajes dentro de una máquina se puede realizar mediante buffer de memoria o mediante socket.

En esta unidad veremos como realizar la **comunicación mediante buffer a través de los flujos de entrada y salida de cada proceso**.

ENTRADA/SALIDA ESTÁNDAR

Todos los procesos disponen, de manera predeterminada, de un flujo (un *stream*) de entrada de datos, otro flujo de salida de resultados y un flujo de salida de mensajes de error y depuración. El sistema operativo dota de estos 3 flujos a cada proceso, sin que el programador tenga que implementarlos:

- **La entrada estándar de datos** de un proceso hace referencia a un *stream* de entrada de datos que puede utilizarse para proporcionar datos al proceso. La entrada estándar es, conceptualmente, el flujo de entrada de datos predeterminado, de entre todos los disponibles en la máquina.
- De igual modo, **la salida estándar de datos** es un *stream* de salida de resultados. Conceptualmente, se considera el *stream* de salida predeterminado del programa.
- Finalmente, **la salida de error estándar** es el flujo de información de depuración, trazas y mensajes de error que emite el proceso a través de un *stream* específico

Estos flujos son un **mecanismo adicional** para transferir datos al programa o para obtener información del mismo. El programador decide si el programa leerá o escribirá en estos flujos, o bien si empleará otros mecanismos de comunicación con el programa, como un entorno gráfico.

En la actualidad es poco frecuente trabajar con estos conceptos al desarrollar aplicaciones sobre entorno gráfico, puesto que disponemos de múltiples entradas y salidas, todas ellas prioritarias para la aplicación (teclado, ratón, datos procedentes de la red, ficheros, etc.), al tiempo que los entornos gráficos concentran la mayor parte de la interacción con el sistema, actuando como entrada y salida simultáneamente.

No obstante, al desarrollar aplicaciones de tipo consola o servicios, los conceptos de entrada/salida estándar aún resultan de utilidad, permitiendo proporcionar datos al proceso o recoger resultados de éste.

Los lenguajes de programación actuales tratan la entrada/salida mediante *streams*, de manera que cada proceso tiene un *stream* de datos de entrada, un *stream* de datos de salida y un *stream* de datos de salida de errores.

Por defecto, todas las entradas/salidas estándar están conectadas a la consola, es decir, los datos de entrada del proceso se leen de lo que se ha escrito en la consola, y los resultados, al igual que los errores, se muestran por la consola.

Sin embargo, los lenguajes de programación permiten **redirigir** estas entradas/salidas a prácticamente cualquier dispositivo, lógico o físico. Por ejemplo, la entrada estándar se puede redirigir para que el programa interprete como entrada estándar un fichero con los datos a leer por el programa, en vez de leer de la consola.

En Java, los tres flujos se manipulan desde los atributos **in, out y err** de la clase ***System***. En los siguientes ejemplos se muestra cómo trabajar con la entrada/salida estándar:

Programa que redirige la salida estándar a un fichero.

```
import java.io.*;

/**
 *
 * @author Fran
 */
public class Salida {

    public static void main(String[] args) {
// Especifica el nombre del archivo de destino
        String archivoSalida = "salida.txt";
// Crea una referencia al stream de salida original (conectado a la consola)
// para poder restituir más tarde la salida otra vez a la consola
        PrintStream originalStdOut = System.out;
        try (
                PrintStream stdOut = new PrintStream(new FileOutputStream(archivoSa-
lida))) {
// Redirige la salida estándar al stream conectado al fichero
            System.setOut(stdOut);
// Ahora todo lo que se escriba en System.out se guardará en el
// archivo
            System.out.println("La salida se ha redirigido al archivo: "
                    + archivoSalida);
// Redirige de nuevo la salida estándar a la consola
            System.setOut(originalStdOut);
            System.out.println("La salida se ha redirigido de nuevo a la "
                    + "consola");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Programa que redirige la entrada estándar a un fichero.

```
import java.io.*;

/**
```

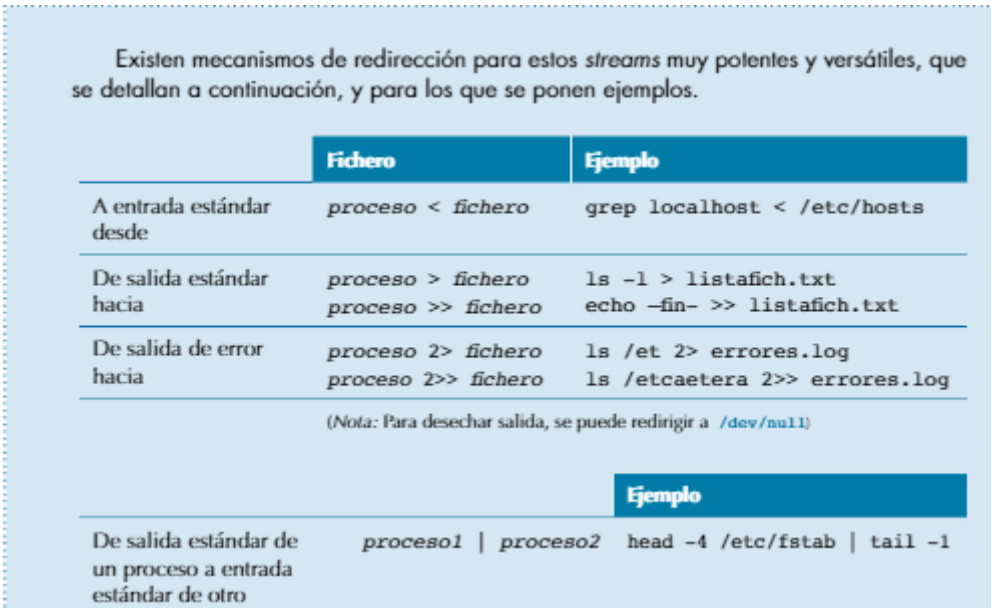
```
 *
 * @author Fran
 */
public class Entrada {

    public static void main(String[] args) {

// Especifica el nombre del archivo de entrada de datos
        String archivoEntrada = "entrada.txt";
// Crea una referencia al stream de entrada original (conectado a la
// consola) para poder restablecer más tarde la entrada otra vez a la
// consola
        InputStream originalStdIn = System.in;
        try (
                FileInputStream fileInputStream = new FileInputStream(archivoEn-
trada)) {
// Redirige la entrada estándar al stream conectado al archivo
            System.setIn(fileInputStream);
// Ahora, al leer a través de System.in se está leyendo del archivo
            System.out.println("La entrada se ha redirigido al archivo: "
                    + archivoEntrada);
// Leee y muestra datos desde la entrada estándar actual (fichero)
            BufferedReader reader = new BufferedReader(new InputStreamReader(Sys-
tem.in));
            String linea;
            while ((linea = reader.readLine()) != null) {
                System.out.println("Línea leída desde el archivo: " + linea);
            }
// Redirige de nuevo la entrada estándar a la consola
            System.setIn(originalStdIn);
            System.out.println("La entrada se ha redirigido de nuevo a la consola");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

ENCADENAMIENTO DE PROCESOS

Los sistemas operativos proporcionan mecanismos de redirección de E/S a través de operadores como tuberías (|) y redirectores (<,<<,>,>>).



Existen mecanismos de redirección para estos *streams* muy potentes y versátiles, que se detallan a continuación, y para los que se ponen ejemplos.

	Fichero	Ejemplo
A entrada estándar desde	*proceso* < *fichero*	`grep localhost < /etc/hosts`
De salida estándar hacia	*proceso* > *fichero* *proceso* >> *fichero*	`ls -l > listafich.txt` `echo –fin– >> listafich.txt`
De salida de error hacia	*proceso* 2> *fichero* *proceso* 2>> *fichero*	`ls /et 2> errores.log` `ls /etcaetera 2>> errores.log`

(*Nota:* Para desechar salida, se puede redirigir a `/dev/null`)

	Ejemplo
De salida estándar de un proceso a entrada estándar de otro	*proceso1* | *proceso2*	`head -4 /etc/fstab | tail -1`

Ejemplo de dos procesos encadenados: Productor.java envía un mensaje a la salida estándar que Consumidor.java recibe y muestra.

```java
public class Productor {
    public static void main(String[] args) {
        System.out.println("Hola soy el productor de mensajes");
    }
}
```

```java
import java.util.Scanner;

public class Consumidor {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String linea = sc.nextLine();
        System.out.println("Consumidor: "+linea);
    }
}
```

Para ejecutarlos: ***java Productor.java | java Consumidor.java***

REDIRECCIONES CON LA CLASE PROCESSBUILDER

Los métodos que proporciona **ProcessBuilder** para la redirección de flujos de E/S, son:

Método **inheritIO()**. Este método enlaza todos los flujos de ES del proceso creado (mediante ProcessBuilder) con la del proceso actual.

Métodos **redirectInput(), redirectOutput() y redirectError()** se utilizan para cambiar los flujos de E/S del proceso creado.

Métodos de redirección para clase `ProcessBuilder` de Java

Métodos	Línea de comandos de Linux
`redirectInput(new File (fichero))`	*comando* < *fichero*
`redirectOutput(new File(fichero))`	*comando* > *fichero*
`redirectOutput(Redirect.appendTo(new File(fichero)))`	*comando* >> *fichero*
`redirectOutput(Redirect.DISCARD)`	*comando* > /dev/null
`redirectError(new File(fichero))`	*comando* 2> *fichero*
`RedirectError(Redirect.appendTo(new File(fichero)))`	*comando* 2>> *fichero*
`redirectError(Redirect.DISCARD))`	*comando* 2> /dev/null

Métodos de la clase `ProcessBuilder` para redirección de entrada y salida

Método	Funcionalidad
`ProcessBuilder inheritIO()`	Redirige la salida estándar y de error de los subprocesos creados hacia las del proceso padre, y su entrada estándar desde la del proceso padre.

`ProcessBuilder redirectInput(File f)`
`ProcessBuilder redirectInput(`
`ProcessBuilder.Redirect fuente)`
`ProcessBuilder redirectOutput(File f)`
`redirectOutput(`
`ProcessBuilder.Redirect destino)`
`ProcessBuilder redirectError(File f)`
`redirectError(`
`ProcessBuilder.Redirect destino)`
	Redirige las entrada y salida estándares y de error, respectivamente, desde o hacia:
	– Un fichero (con `File f`).
	– La correspondiente del proceso padre (con `Redirect.INHERIT`).
	– La salida estándar y de error se pueden descartar (con `Redirect.DISCARD`).

`ProcessBuilder redirectErrorStream(` `boolean redir)`	Con valor `true` para `redir`, dirige la salida de error hacia la estándar. Esto hace innecesario gestionar por separado la salida de error. Tiene el inconveniente de que no se puede distinguir entre ambas. Pero tiene la ventaja de que evita que ambas aparezcan entremezcladas, lo que dificulta su lectura e interpretación.
`static List<Process> startPipeline(` `List<ProcessBuilder> builders)`	Inicia un proceso para cada `ProcessBuilder`, y crea una secuencia de procesos cuyas salida y entradas están enlazadas mediante tuberías o *pipes*, de manera que la salida de un proceso se dirige a la entrada del siguiente en la secuencia. Solo se pueden redirigir la entrada del primer proceso y la salida del último. El resto de entradas y salidas están conectadas mediante *pipes* de un proceso al siguiente y, obviamente, no se pueden redirigir.

REDIRECCIONES DE STREAMS DE LA CLASE PROCESS

La clase **Process** tiene métodos que devuelven los flujos de ES (streams) asociados al proceso. Esto permite controlar las redirecciones de los mismos.

Métodos de `Process` para obtener *streams* asociados a entrada estándar y a salidas estándar y de error

Método	Funcionalidad
`InputStream getInputStream()`	Devuelve un *stream* de entrada conectado con la salida estándar del proceso.
`OutputStream getOutputStream()`	Devuelve un *stream* de salida conectado con la entrada estándar del proceso.
`InputStream getErrorStream()`	Devuelve un *stream* de entrada conectado con la salida de error del proceso.

Ejemplo de redirección de la salida.

```java
import java.io.*;
/**
 *
 * @author Fran
 */
public class Comando {
    public static void main(String[] args) throws IOException, InterruptedException {
        ProcessBuilder pb = new ProcessBuilder("ipconfig");
        Process ps = pb.start();
        InputStream is = ps.getInputStream();
        InputStreamReader isr = new InputStreamReader(is,"UTF-8");
        BufferedReader br = new BufferedReader(isr);
        int codRet = ps.waitFor();
        System.out.println("El comando ha terminado " +
                (codRet==0?"Correctamente":"Con algún error"));
        String line;
        System.out.println("Salida del proceso ipconfig: "  );
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    }
}
```

Ejemplo de redirección de entrada a un proceso.

```java
import java.io.*;

public class RedireccionEntrada {
    public static void main(String[] args) {
        try {
            ProcessBuilder pb = new ProcessBuilder("nslookup");
            pb.redirectInput(ProcessBuilder.Redirect.PIPE);
            pb.redirectOutput(ProcessBuilder.Redirect.INHERIT);

            Process subproceso = pb.start();

            OutputStream outputStream = subproceso.getOutputStream();
            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
```

```
            BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(outputStream));

            System.out.println("Introduce líneas de texto (presiona Ctrl+Z para finalizar):");
            String linea;
            while ((linea = reader.readLine()) != null) {
                writer.write(linea);
                writer.newLine();
                writer.flush();
            }

            writer.close();
            subproceso.waitFor();

        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Ejemplo de comunicación entre dos procesos coordinados por un tercero (el principal)

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


public class Intercomunicador {
    public static void main(String[] args) throws Exception {
        // Crear ProcessBuilder para Emisor y Receptor
        ProcessBuilder pbEmisor = new ProcessBuilder("java", "Emisor.java");
        ProcessBuilder pbReceptor = new ProcessBuilder("java", "Receptor.java");

        // Configurar redirecciones
        pbEmisor.redirectOutput(ProcessBuilder.Redirect.PIPE);
        pbReceptor.redirectInput(ProcessBuilder.Redirect.PIPE);
        pbReceptor.redirectOutput(ProcessBuilder.Redirect.INHERIT);

        // Iniciar procesos
        Process pEmisor = pbEmisor.start();
        Process pReceptor = pbReceptor.start();

        // Conectar la salida del Emisor con la entrada del Receptor
        InputStream emisorOutput = pEmisor.getInputStream();
        OutputStream receptorInput = pReceptor.getOutputStream();

        emisorOutput.transferTo(receptorInput);
        receptorInput.close();

        // Esperar a que los procesos terminen
        int exitCodeEmisor = pEmisor.waitFor();
```

```
        int exitCodeReceptor = pReceptor.waitFor();

        System.out.println("Proceso Emisor terminó con código: " + exitCodeEmisor);
        System.out.println("Proceso Receptor terminó con código: " + exitCodeReceptor);
    }
}
```

```
public class Emisor {
    public static void main(String[] args) throws Exception {
        // Simplemente envía un mensaje
        System.out.println("Hola desde el proceso 1");
    }
}
```

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class Receptor {
    public static void main(String[] args) throws Exception {
        // Lee el mensaje y responde
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String mensaje = br.readLine();
        System.out.println("Proceso 2 recibió: " + mensaje);
        System.out.println("Hola desde el proceso 2");
    }
}
```

ACCESO A RECURSO COMPARTIDO

Otra forma de comunicarse los procesos es mediante el acceso a un recurso compartido, por ejemplo, un fichero. Para habrá procesos que escriban y otros que lean mediante las correspondientes operaciones de escritura y lectura.

Con respecto a las lecturas y escrituras, debemos recordar, que serán bloqueantes. Es decir, un proceso quedará bloqueado hasta que los datos estén listos para poder ser leídos. Una escritura, bloqueará al proceso que intenta escribir, hasta que el recurso no esté preparado para poder escribir.

Con el fin de establecer mecanismos de seguridad que eviten la pérdida de información o problemas de bloqueos entre procesos vamos a ver algunos conceptos de sincronización.

En programación concurrente, siempre que accedamos a algún **recurso compartido** (eso incluye a los **ficheros**), deberemos tener en cuenta las **condiciones** en las que **nuestro proceso debe hacer uso de ese recurso**: ¿será de forma exclusiva o no?

En el caso de lecturas y escrituras en un fichero, debemos determinar si queremos acceder al fichero como sólo lectura; escritura; o lectura-escritura; y utilizar los objetos que nos permitan establecer los mecanismos de sincronización necesarios para que un proceso pueda bloquear el uso del fichero por otros procesos cuando él lo esté utilizando.

Esto se conoce como el **problema de los procesos lectores-escritores**. El **sistema operativo, nos ayudará** a resolver los problemas que se plantean; ya que:

- Si el acceso es de **sólo lectura**. Permitirá que todos los procesos lectores, que sólo quieren leer información del fichero, **puedan acceder simultáneamente** a él.
- En el caso de **escritura**, o lectura-escritura. El SO nos permitirá pedir un tipo de **acceso** de **forma exclusiva** al fichero. Esto significará que el proceso deberá esperar a que otros procesos lectores terminen sus accesos. Y otros procesos (lectores o escritores), esperarán a que ese proceso escritor haya finalizado su escritura.

Debemos tener en cuenta que, nosotros, **nos comunicamos con el SO a través de los objetos y métodos proporcionados por un lenguaje de programación**; y, por lo tanto, tendremos que **consultar cuidadosamente la documentación de las clases** que estamos utilizando para **conocer** todas las **peculiaridades** de su comportamiento.

Sección crítica o región crítica. El conjunto de instrucciones en las que un proceso accede a un recurso compartido.

Exclusión mutua. Las instrucciones que forman la región crítica, **se ejecutarán de forma indivisible o atómica y de forma exclusiva con respecto a otros procesos que accedan al mismo recurso** compartido al que se está accediendo.

Ejemplo sin exclusión mutua.

```
public class Incrementador {

    private static final Path RUTA_FICHERO = Paths.get("valor.txt");
    private static final int MILISEGUNDOS_ESPERA = 100; // Pequeña pausa para forzar la concu-
rrencia

    public static void main(String[] args) {
        int valorActual = 0;

        try {
            // 1. LEER EL VALOR ACTUAL
            if (Files.exists(RUTA_FICHERO)) {
                String contenido = Files.readString(RUTA_FICHERO).trim();
                if (!contenido.isEmpty()) {
                    valorActual = Integer.parseInt(contenido);
                }
            }

            System.out.println("Proceso " + ProcessHandle.current().pid() + " ha leído el valor:
" + valorActual);

            // Introducimos una pequeña pausa para hacer más probable la condición de carrera
```

```
            Thread.sleep(MILISEGUNDOS_ESPERA);

            // 2. INCREMENTAR EL VALOR
            int nuevoValor = valorActual + 1;

            // 3. ESCRIBIR EL NUEVO VALOR
            Files.writeString(RUTA_FICHERO, String.valueOf(nuevoValor), StandardOpenOp-
tion.CREATE, StandardOpenOption.TRUNCATE_EXISTING);
            System.out.println("Proceso " + ProcessHandle.current().pid() + " ha escrito el va-
lor: " + nuevoValor);

        } catch (IOException e) {
            System.err.println("Error de E/S: " + e.getMessage());
        } catch (NumberFormatException e) {
            System.err.println("El fichero no contiene un número válido. Se reiniciará a 0.");
            // Si el fichero está corrupto, podemos decidir empezar de 0.
            try {
                Files.writeString(RUTA_FICHERO, "1", StandardOpenOption.CREATE, StandardOpenOp-
tion.TRUNCATE_EXISTING);
            } catch (IOException ioException) {
                System.err.println("No se pudo reiniciar el fichero: " + ioException.getMes-
sage());
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("El proceso fue interrumpido.");
        }
    }
}
```

Ejemplo con exclusión mutua

```
public class IncrementadorSincronizado {

    private static final Path RUTA_FICHERO = Paths.get("valor.txt");
    private static final int MILISEGUNDOS_ESPERA = 100;

    public static void main(String[] args) {

        // Usamos try-with-resources para asegurar que el fichero y el canal se cierran
        try (RandomAccessFile file = new RandomAccessFile(RUTA_FICHERO.toFile(), "rw");
             FileChannel channel = file.getChannel()) {

            // 1. ADQUIRIR EL BLOQUEO EXCLUSIVO
            // El método lock() es bloqueante: espera hasta que el bloqueo esté disponible.
            FileLock lock = channel.lock();
            System.out.println("Proceso " + ProcessHandle.current().pid() + " ha adquirido el
bloqueo.");
```

```
            try {
                int valorActual = 0;

                // 2. LEER (dentro de la sección crítica)
                if (file.length() > 0) {
                    file.seek(0); // Posicionarse al inicio del fichero
                    String contenido = file.readLine();
                    if (contenido != null && !contenido.trim().isEmpty()) {
                       valorActual = Integer.parseInt(contenido.trim());
                    }
                }
                System.out.println("Proceso " + ProcessHandle.current().pid() + " ha leído el
valor: " + valorActual);

                Thread.sleep(MILISEGUNDOS_ESPERA);

                // 3. INCREMENTAR (dentro de la sección crítica)
                int nuevoValor = valorActual + 1;

                // 4. ESCRIBIR (dentro de la sección crítica)
                file.seek(0); // Volver al inicio para sobrescribir
                file.setLength(0); // Truncar el fichero por si el nuevo número ocupa menos
                file.writeBytes(String.valueOf(nuevoValor));
                System.out.println("Proceso " + ProcessHandle.current().pid() + " ha escrito el
valor: " + nuevoValor);

            } finally {
                // 5. LIBERAR EL BLOQUEO
                // Es crucial liberar el bloqueo para que otros procesos puedan acceder.
                lock.release();
                System.out.println("Proceso " + ProcessHandle.current().pid() + " ha liberado el
bloqueo.");
            }

        } catch (IOException e) {
            System.err.println("Error de E/S: " + e.getMessage());
        } catch (NumberFormatException e) {
            System.err.println("Fichero corrupto. No se pudo leer un número.");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("El proceso fue interrumpido.");
        }
    }
}
```