UNIDAD 1
PROGRAMACIÓN MULTIPROCESO

1 Programas y procesos

El resultado de la compilación de un programa se guarda en **ficheros ejecutables** junto con las estructuras de datos necesarias para que el sistema operativo pueda lanzar su ejecución. En los sistemas operativos de Microsoft se identifican fácilmente debido a que usan la extensión .exe. En sistemas operativos Unix y derivados, no suelen usar ninguna extensión. En ambos casos, se pueden usar programas desensambladores como IDA Freeware para obtener un listado del programa en código ensamblador. También es posible utilizar desensambladores en línea como el que se encuentra en la web disassembler.io.

Sin embargo, existen lenguajes de programación en los que el código fuente no se compila a código nativo. Es el caso de Java, que se compila a *byte codes* que se guardan en ficheros ejecutables con extensión .class, que a su vez se pueden agrupar en ficheros .jar ejecutables. En este caso, será la JVM la encargada de ejecutar los programas almacenados en estos ficheros. La JVM no es más que un programa compilado a código nativo, almacenado en un fichero ejecutable (p.ej. java.exe en Windows) que será ejecutado por el sistema operativo.

Un **proceso** es un programa en ejecución que se encuentra bajo el control del sistema operativo. En el caso de la multitarea, en la que los procesos han de compartir el uso de una CPU, el sistema operativo decide cuando suspender temporalmente la ejecución de un proceso para que otro pueda ejecutarse. Esto implica que cada vez que se reinicia un proceso, debe hacerlo en el mismo estado en que se encontraba cuando se paró. Para ello, es necesario que el sistema operativo mantenga una estructura de datos por cada proceso en ejecución denominada **BCP** (Bloque de Control de Proceso) donde se almacena la siguiente información:

- Identificador único del proceso.
- Estado del proceso.
- Contador de programa.
- Registros de CPU.
- Información de planificación de CPU como la prioridad del proceso.
- Información de gestión de memoria.
- Información contable como la cantidad de tiempo de CPU y tiempo real consumido.
- Información de estado de E/S como la lista de dispositivos asignados, archivos abiertos, etc.

Los estados en los que se pueden encontrar un proceso son los siguientes:

- En ejecución: el proceso está usando la CPU.
- Bloqueado: el proceso no puede hacer nada hasta que no ocurra un evento externo, como por ejemplo la finalización de una operación de E/S.
- Listo: el proceso está parado temporalmente y listo para ejecutarse cuando se le dé oportunidad.

La Figura 1. Estados de un proceso. muestra mediante un diagrama de estados, los estados en que se puede encontrar un proceso:

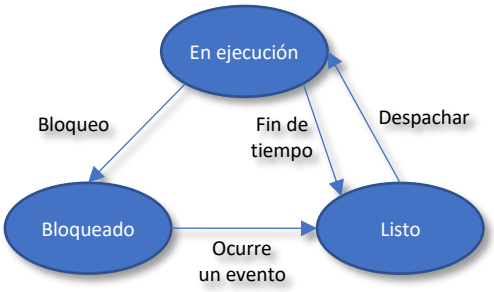


Figura 1. Estados de un proceso.

1.1 Creación de procesos con Java

Java incluye en el paquete *java.lang* varias clases para la gestión de procesos: *Process* y *Runtime*. Cada aplicación Java que se ejecuta dispone de una instancia de la clase *Runtime* que representa el entorno de ejecución de la aplicación y que se obtiene ejecutando:

Runtime.getRuntime()

La clase *Process* representa un proceso que se lanza ejecutando:

Runtime.getInstance().exec(comando)

El parámetro *comando*, de tipo *String*, representa el comando con el que se ejecutará el proceso tal y como se escribiría en una línea de comando del sistema operativo. El método *exec* lanza un proceso y retorna una referencia a un objeto *Process* que lo representa.

Los métodos *getInputStream*(), *getErrorInputStream*() y *getOutputStream*() del objeto *Process* retornan los streams de entrada y salida que servirán para leer lo que el proceso envía a la salida estándar o para enviarle datos que leerá de la entrada estándar.

Otros métodos de la clase *Process* nos permitirán gestionar el estado del proceso:

- *isAlive*() permite conocer si el proceso sigue vivo
- *exitValue*() retorna el código de salida
- *waitFor*() espera a que el proceso finalice
- *destroy*() finaliza el proceso de forma abrupta.

El siguiente ejemplo Java muestra cómo se puede ejecutar el block de notas de Windows:

```
public class Ejemplo1 {
    public static void main(String[] args) {
       Runtime r = Runtime.getRuntime();
       String comando= "notepad";
       Process p;
       p = r.exec(comando);
    }
}
```

Para los comandos internos (los que están integrados en el intérprete de comandos, p. ej. *dir* o *attrib* en Windows) es necesario ejecutar el intérprete de comandos (p. ej. cmd.exe en Windows). A continuación, se

muestra cómo ejecutar desde un programa Java el comando *dir* de Windows y mostrar su salida en la consola, así como el código de salida:

```
import java.io.*;

public class Ejemplo2 {

    public static void main(String[] args) throws IOException,
            InterruptedException {
        Runtime r = Runtime.getRuntime();
        Process p =  r.exec( "cmd /C dir c:\\" );
        InputStream is = p.getInputStream();
        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        String linea;
        while((linea = br.readLine()) != null)
            System.out.println(linea);
        br.close();
        System.out.println("código de salida: " + p.waitFor());
    }
}
```

El parámetro /C se le pasa al intérprete de comandos para que ejecute el comando y después finalice. La salida estándar del comando se encuentra conectada mediante una tubería del sistema con el *InputStream* que retorna el método *getInputStream()* de la clase *Process*.

Después de compilar y ejecutar este ejemplo, se visualizará una salida similar a la siguiente:

```
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: BCC5-A7CC

 Directorio de c:\

07/12/2019  11:14    <DIR>          PerfLogs
12/09/2022  11:43    <DIR>          Program Files
11/09/2022  19:24    <DIR>          Program Files (x86)
07/09/2022  20:18    <DIR>          Users
22/09/2022  09:53    <DIR>          Windows
               0 archivos              0 bytes
               5 dirs  47.342.071.808 bytes libres
código de salida: 0
```

El código siguiente, añadido al ejemplo anterior, muestra cómo leer la salida estándar de error del proceso lanzado que se encuentra conectada mediante una tubería del sistema con el *InputStream* que retorna el método *getErrorStream()* de la clase *Process*. Será necesario también cambiar el comando para que haga referencia a un comando interno inexistente, p.ej. *"cmd /C dirr"*, provocando así que el intérprete de comandos envíe los mensajes de error correspondientes a la salida estándar de error:

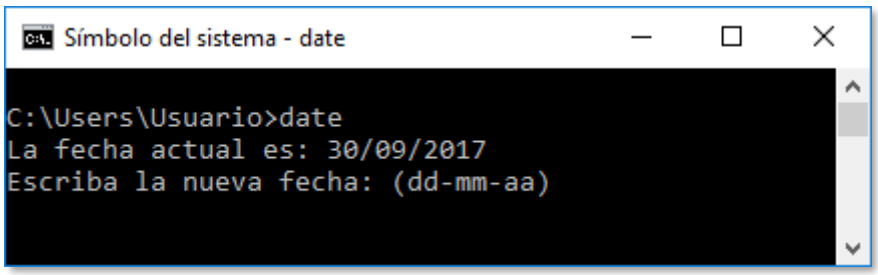
```
if (exitCode != 0)
    try (BufferedReader inErr = new BufferedReader(new InputStreamReader(p.getErrorStream()))) {
        String lineErr = null;
        while ((lineErr = inErr.readLine()) != null)
            System.out.println("ERROR >" + lineErr);
    }
```

El resultado de ejecutar el ejemplo genera la salida siguiente:

```
código de salida: 1
ERROR >"dirr" no se reconoce como un comando interno o externo,
ERROR >programa o archivo por lotes ejecutable.
```

El hecho de que el proceso lanzado use la salida estándar de error para mostrar mensajes de error no implica necesariamente que vaya a finalizar su ejecución retornando un código de error distinto de cero. Por tanto, el condicional en el código anterior no siempre estará justificado.

Supongamos ahora que queremos ejecutar un comando que necesita información de entrada. Por ejemplo, si se ejecuta el comando *date* en la línea de comandos, mostrará la fecha actual y a continuación le pedirá al usuario que escriba una nueva fecha:



Para resolver este problema, se puede usar el método *getOutputStream()* de la clase *Process* para enviar los datos requeridos a la entrada estándar del proceso. El ejemplo siguiente ejecuta el comando *date* y le suministra de esta forma la fecha *01-01-22*:

```
import java.io.*;

public class Ejemplo3 {

    public static void main(String[] args) {
        Process process = Runtime.getRuntime().exec("cmd /C date");
        try (OutputStream os = process.getOutputStream()) {
            os.write(("01-01-22" + System.getProperty("line.separator")).getBytes());
            os.flush();
        }    }

}
```

ACTIVIDAD 1

Escribe un programa Java que lea 2 números de la entrada estándar y visualice su suma. Si los datos introducidos no son numéricos, muestra el mensaje correspondiente en la salida estándar de error. Crea otro programa Java para ejecutar el anterior y visualizar la salida del mismo.

A partir de la versión 1.5 del JDK se añadió una nueva forma de creación y ejecución de procesos del sistema operativo mediante la clase *ProcessBuilder*. Igual que *Process* y *Runtime* pertenece al paquete *java.lang*. Cada instancia *ProcessBuilder* gestiona una colección de atributos de proceso. El método

start() crea una nueva instancia de Process con esos atributos y puede ser invocado varias veces desde la misma instancia para crear nuevos subprocesos con atributos idénticos o relacionados.

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

- Un comando 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 padre.
- Una fuente de entrada estándar. Por defecto, la entrada estándar del subproceso está conectada mediante una tubería al stream que retorna el método *getOutputStream()* del proceso.
- Un destino para la salida estándar y la salida de error. Por defecto se conectan mediante una tubería a los streams de entrada retornados por los métodos *getInputStream* y *getErrorStream* del proceso respectivamente.

La clase *ProcessBuilder* suministra dos constructores para crear un objeto de este tipo:

- `ProcessBuilder(List<String> command)`
  Usa una lista de cadenas para especificar el ejecutable y los argumentos.
- `ProcessBuilder(String... command)`
  Un número variable parámetros de tipo cadena para especificar el ejecutable y los argumentos.

Una vez creado el objeto *ProcessBuilder*, se lanza el proceso invocando al método *start*. Por ejemplo, para lanzar el proceso visto en *Ejemplo2* usando el primer constructor, se pueden ejecutar las sentencias siguientes:

```
ProcessBuilder pb = new ProcessBuilder(Arrays.asList("cmd",  "/C", "dir"));
Process p = pb.start();
```

Con el segundo constructor:

```
ProcessBuilder pb = new ProcessBuilder("cmd", "/C","dir");
Process p = pb.start();
```

ACTIVIDAD 2

Crea un programa Java que use la clase *ProcessBuilder* para ejecutar el comando "*cmd /C dir*" y muestre en la consola lo siguiente:

- Las variables de entorno del subproceso.
- Los argumentos usados para ejecutar el comando.
- La salida estándar del subproceso.

La versión 1.7 del JDK incluye en la clase *ProcessBuilder* los métodos `redirectInput(File file)`, `redirectError(File file)` y `redirectOutput(File file)` para redirigir la entrada estándar, la salida estándar de error y la salida estándar respectivamente hacia un fichero.

ACTIVIDAD 3

a) El programa Java siguiente escribe 5 líneas repitiendo en cada una de ellas un saludo que recibe a través de un parámetro en la línea de comandos:

```
public class UnSaludo {
    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("SE NECESITA UN SALUDO...");
            System.exit(1);
        }
        for(int i=1; i<=5; i++)
            System.out.println(i + ". " + args[0]);
    }
}
```

Crea y ejecuta otro programa Java que ejecute el anterior redireccionando su salida estándar a un fichero. Realiza todo el proceso desde el entorno integrado de desarrollo.

b) Crea un programa Java que reciba en un parámetro de línea de comando la ruta a un fichero .bat. Tendrám que lanzar el intérprete de comandos de Windows y redireccionar su entrada estándar para que ejecute los comandos del fichero .bat. El intérprete de comandos se lanzará redireccionando su salida estándar y de error a los ficheros *salida.txt* y *error.txt* respectivamente. El contenido del fichero .bat será el siguiente:

```
mkdir nuevo
cd nuevo
echo prueba > MiFichero.txt
dir /W
dirr
echo FIN
```

2 PROGRAMACIÓN CONCURRENTE

Según el diccionario WordReference.com el término concurrencia en su tercera acepción significa "acaecimiento o coincidencia de varios sucesos o cosas a un mismo tiempo". Si sustituimos "sucesos o cosas" por "procesos" tendremos una idea aproximada del significado del término aplicado a la ejecución de programas en los sistemas informáticos.

La programación concurrente es la disciplina que se ocupa del estudio y desarrollo de programas que ejecuten sus acciones de forma concurrente.

Esta disciplina surge en los años 60, cuando las mejoras en el hardware permiten que uno de los objetivos de los sistemas operativos se centre en el mejor aprovechamiento de las capacidades de los procesadores ejecutando varios procesos de forma concurrente.

2.1 Programa y proceso

Al principio del tema se definió un proceso como un programa en ejecución. Y ¿qué es un programa?, podemos definir programa como un conjunto de instrucciones que se aplican a un conjunto de datos de entrada para obtener una salida. Un proceso es algo activo que cuenta con una serie de recursos asociados, en cambio un programa es algo pasivo, para que pueda hacer algo hay que ejecutarlo.

Pero un programa al ponerse en ejecución puede dar lugar a más de un proceso, cada uno ejecutando una parte del programa. Por ejemplo, el navegador web, por un lado, está controlando las acciones del usuario con la interfaz, por otro hace las peticiones al servidor web. Entonces cada vez que se ejecuta este programa crea 2 procesos.

En la Figura 2. Programa con 3 instancias en ejecución. se representa un programa almacenado en disco y 3 instancias del mismo ejecutándose, por ejemplo, por 3 usuarios diferentes. Cada instancia del programa es un proceso, por tanto, existen 3 procesos independientes ejecutándose al mismo tiempo sobre el sistema operativo, tenemos entonces 3 procesos concurrentes.

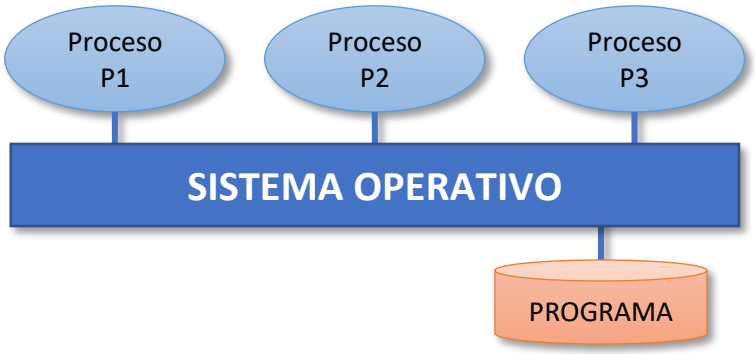


Figura 2. Programa con 3 instancias en ejecución.

Dos procesos serán concurrentes cuando la primera instrucción de uno de ellos se ejecuta después de la primera instrucción del otro y antes de la última. Es decir, existe un solapamiento o intercalado en la ejecución de sus instrucciones. No hay que confundir el solapamiento con la ejecución simultánea de las instrucciones, en este caso estaríamos en una situación de programación paralela, aunque a veces el hardware subyacente (más de un procesador) si permitirá la ejecución simultánea.

Supongamos ahora que el programa anterior al ejecutarse da lugar a 2 procesos más, cada uno ejecutando una parte del programa, como se muestra en la Figura 3. Un programa dando lugar a más de un proceso.. Ya que un programa puede estar compuesto por diversos procesos, una definición más acertada de proceso es la de una actividad asíncrona susceptible de ser asignada a un procesador.

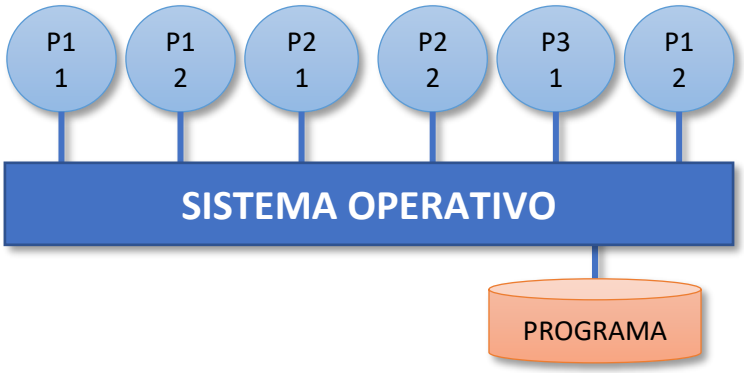


Figura 3. Un programa dando lugar a más de un proceso.

Cuando varios procesos se ejecutan concurrentemente puede haber procesos que colaboren para un determinado fin (por ejemplo, P1.1 y P1.2), y otros que compitan por los recursos del sistema (por ejemplo, P2.1 y P3.1). Estas tareas de colaboración y competencia por los recursos exigen mecanismos de comunicación y sincronización entre procesos.

2.2 Características

La **programación concurrente** es la disciplina que se encarga del estudio de las notaciones que permiten especificar la ejecución concurrente de las acciones de un programa, así como las técnicas para resolver los problemas inherentes a la ejecución concurrente (comunicación y sincronización).

BENEFICIOS

La programación concurrente aporta una serie de beneficios:

Mejor aprovechamiento de la CPU. Un proceso puede aprovechar ciclos de CPU mientras otro realiza una operación de entrada/salida.

Velocidad de ejecución. Al subdividir un programa en procesos, estos se pueden "repartir" entre procesadores o gestionar en un único procesador según importancia.

Solución a problemas de naturaleza concurrente. Existen algunos problemas cuya solución es más fácil utilizando esta metodología:

- Sistemas de control: son sistemas en los que hay captura de datos, normalmente a través de sensores, análisis y actuación en función del análisis. Un ejemplo son los sistemas de tiempo real.
- Tecnologías web: los servidores web son capaces de atender múltiples peticiones de usuarios concurrentemente, también los servidores de chat, correo, los propios navegadores web, etc.
- Aplicaciones basabas en GUI: el usuario puede interactuar con la aplicación mientras la aplicación está realizando otra tarea. Por ejemplo, el navegador web puede estar descargando un archivo mientras el usuario navega por las páginas.
- Simulación: programas que modelan sistemas físicos con autonomía.
- Sistemas Gestores de Bases de Datos: Los usuarios interactúan con el sistema, cada usuario puede ser visto como un proceso.

CONCURRENCIA Y HARDWARE

En un sistema monoprocesador se puede tener una ejecución concurrente gestionando el tiempo de procesador para cada proceso. El S.O. va alternando el tiempo entre los distintos procesos, cuando uno necesita realizar una operación de entrada salida, lo abandona y otro lo ocupa; de esta forma se aprovechan los ciclos del procesador. En la Figura 4. Concurrencia. se muestra cómo el tiempo de procesador es repartido entre 3 procesos, en cada momento solo hay un proceso. Esta forma de gestionar los procesos en un sistema monoprocesador recibe el nombre de **multiprogramación**.

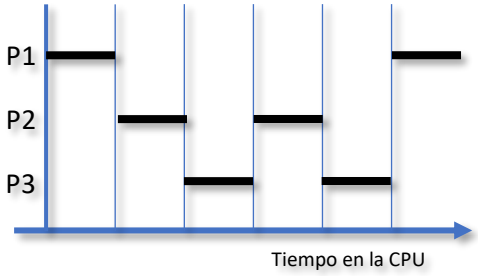


Figura 4. Concurrencia.

En un sistema monoprocesador todos los procesos comparten la misma memoria. La forma de comunicar y sincronizar procesos se realiza mediante variables compartidas.

En un sistema multiprocesador (existe más de un procesador) podemos tener un proceso en cada procesador. Esto permite que exista paralelismo real entre los procesos, véase la Figura 5. Paralelismo.. Estos pueden ser de memoria compartida (fuertemente acoplados) o con memoria local a cada procesador (débilmente acoplados). Se denomina multiproceso a la gestión de varios procesos dentro de un sistema multiprocesador, donde cada procesador puede acceder a una memoria común.

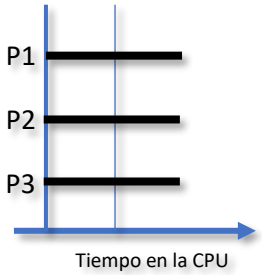


Figura 5. Paralelismo.

2.3 Programas concurrentes

Un programa concurrente define un conjunto de acciones que pueden ser ejecutadas simultáneamente. Supongamos que tenemos estas dos instrucciones en un programa, está claro que el orden de la ejecución de las mismas influirá en el resultado final:

x = x + 1; y = x + 1;	La primera instrucción se debe ejecutar antes de la segunda.

En cambio, si tenemos estas otras, el orden de ejecución es indiferente:

x = 1;
y = 2;
z = 3;
	El orden no interviene en el resultado final.

CONDICIONES DE BERNSTEIN

Bernstein definió unas condiciones para que dos conjuntos de instrucciones se puedan ejecutar concurrentemente. En primer lugar, es necesario formar para cada instrucción 2 conjuntos de variables:

- **Conjunto de lectura**: formado por variables a las que se accede en modo lectura durante la ejecución de la instrucción.
- **Conjunto de escritura**: formado por variables a las que se accede en modo escritura durante la ejecución de la instrucción.

Por ejemplo, sean las siguientes instrucciones:

Instrucción 1 ($I1$):	x = y + 1;
Instrucción 2 ($I2$):	y = x + 2;
Instrucción 3 ($I3$):	z = a + b;

Los conjuntos de lectura y escritura estarían formados por las variables siguientes:

	Conjunto lectura (L)	Conjunto escritura €
I_1:	y	x
I_2:	x	y
I_3:	a, b	z

Se pueden expresar de la siguiente manera:

$L(I_1) = \{y\}$	$E(I_1) = \{x\}$
$L(I_2) = \{x\}$	$E(I_2) = \{y\}$
$L(I_3) = \{a, b\}$	$E(I_3) = \{z\}$

Para que dos instrucciones, I_i e I_j se puedan ejecutar concurrentemente se deben cumplir estas 3 condiciones:

- La intersección entre el conjunto $L(I_i)$ el conjunto $E(I_j)$ debe ser vacío, es decir, no debe haber variables comunes:

$$L(I_i) \cap E(I_j) = \varnothing$$

- La intersección entre el conjunto $E(I_i)$ el conjunto $L(I_j)$ debe ser vacío, es decir, no debe haber variables comunes:

$$E(I_i) \cap L(I_j) = \varnothing$$

- Por último, la intersección entre el conjunto $E(I_i)$ el conjunto $E(I_j)$ debe ser vacío, es decir, no debe haber variables comunes:

$$E(I_i) \cap E(I_j) = \varnothing$$

En el ejemplo anterior tenemos las siguientes condiciones, donde se observa que las instrucciones I_1 e I_2 no se pueden ejecutar concurrentemente porque no cumplen las 3 condiciones:

Conjunto I1 e I2	Conjunto I2 e I3	Conjunto I1 e I3
$L(I_1) \cap E(I_2) \neq \varnothing$	$L(I_2) \cap E(I_3) = \varnothing$	$L(I_1) \cap E(I_3) = \varnothing$
$E(I_1) \cap L(I_2) \neq \varnothing$	$E(I_2) \cap L(I_3) = \varnothing$	$E(I_1) \cap L(I_3) = \varnothing$

$E(I_1) \cap E(I_2) = \varnothing$ $E(I_2) \cap E(I_3) = \varnothing$ $E(I_1) \cap E(I_3) = \varnothing$

En los programas secuenciales hay un orden fijo de ejecución de las instrucciones, siempre se sabe por dónde va a ir el programa. En cambio, en los programas concurrentes hay un orden parcial. Al haber solapamiento de instrucciones no se sabe cuál va a ser el orden de ejecución, puede ocurrir que ante unos mismos datos de entrada el flujo de ejecución no sea el mismo. Esto da lugar a que los programas concurrentes tengan un comportamiento indeterminista donde repetidas ejecuciones sobre un mismo conjunto de datos puedan dar diferentes resultados.

2.4 Problemas inherentes a la programación concurrente

A la hora de crear un programa concurrente podemos encontrarnos con dos problemas:

- **Exclusión mutua**. En programación concurrente es muy típico que varios procesos accedan a la vez a una variable compartida para actualizarla. Esto se debe evitar, ya que puede producir inconsistencia de datos: uno puede estar actualizando la variable a la vez que otro la puede estar leyendo. Por ello es necesario conseguir la exclusión mutua de los procesos respecto a la variable compartida. Para ello se propuso la región crítica. Cuando dos o más procesos comparten una variable, el acceso a dicha variable debe efectuarse siempre dentro de la región crítica asociada a la variable. Solo uno de los procesos podrá acceder para actualizarla y los demás deberán esperar, el tiempo de estancia es finito.
- **Condición de sincronización**. Hace referencia a la necesidad de coordinar los procesos con el fin de sincronizar sus actividades. Puede ocurrir que un proceso P1 llegue a un estado X que no pueda continuar su ejecución hasta que otro proceso P2 haya llegado a un estado Y de su ejecución. La programación concurrente proporciona mecanismos para bloquear procesos a la espera de que ocurra un evento y para desbloquearlos cuando este ocurra.

Algunas herramientas para manejar la concurrencia son: la región crítica, los semáforos, región critica condicional, buzones, sucesos, monitores y sincronización por rendez-vous.

ACTIVIDAD 4

Responde a las siguientes cuestiones:

a) Escribe alguna característica de un programa concurrente.
b) ¿Cuál es la ventaja de la concurrencia en los sistemas monoprocesador?
c) ¿Cuáles son las diferencias entre multiprogramación y multiproceso?
d) ¿Cuáles son los dos problemas principales inherentes a la programación concurrente?

3 PROGRAMACIÓN PARALELA Y DISTRIBUIDA

3.1 Programación paralela

Un programa paralelo es un tipo de programa concurrente diseñado para ejecutarse en un sistema multiprocesador. El procesamiento paralelo permite que muchos elementos de proceso independientes trabajen simultáneamente para resolver un problema. Estos elementos pueden ser un número arbitrario de

equipos conectados por una red. un único equipo con varios procesadores o una combinación de ambos. El problema a resolver se divide en partes independientes de tal forma que cada elemento pueda ejecutar la parte de programa que le corresponda a la vez que los demás.

Recordemos que en un sistema multiprocesador, donde existe más de un procesador, podemos tener un proceso en cada procesador y todos juntos trabajan para resolver un problema. Cada procesador realiza una parte del problema y necesita intercambiar información con el resto. Según cómo se realice este intercambio podemos tener modelos distintos de programación paralela:

- **Modelo de memoria compartida**: los procesadores comparten físicamente la memoria, es decir, todos acceden al mismo espacio de direcciones. Un valor escrito en memoria por un procesador puede ser leído directamente por cualquier otro.
- **Modelo de paso de mensajes**: cada procesador dispone de su propia memoria independiente del resto y accesible solo por él. Para realizar el intercambio de información es necesario que cada procesador realice la petición de datos al procesador que los tiene, y este haga el envío. El entorno de programación PVM que veremos más adelante utiliza este modelo.

Tradicionalmente, el paralelismo se ha utilizado en centros de supercomputación para resolver problemas de elevado coste computacional en un tiempo razonable, pero en la última década su interés se ha extendido por la difusión de los procesadores con múltiples núcleos.

NOTA: El intercambio de información entre procesadores depende del sistema de almacenamiento que se disponga. Según este criterio las arquitecturas paralelas se clasifican en: *Sistemas de memoria compartida* o *multiprocesadores*: los procesadores comparten físicamente la memoria; y *sistemas de memoria distribuida* o *multicomputadores*: cada procesador dispone de su propia memoria.

VENTAJAS

Ventajas del procesamiento paralelo:

- Proporciona ejecución simultánea de tareas.
- Disminuye el tiempo total de ejecución de una aplicación.
- Resolución de problemas complejos y de grandes dimensiones.
- Utilización de recursos no locales, por ejemplo, los recursos que están en una red distribuida, una WAN o la propia red internet.
- Diminución de costos, en vez de gastar en un supercomputador muy caro se pueden utilizar otros recursos más baratos disponibles remotamente.

INCONVENIENTES

- Los compiladores y entornos de programación para sistemas paralelos son más difíciles de desarrollar.
- Los programas paralelos son más difíciles de escribir.
- El consumo de energía de los elementos que forman el sistema.
- Mayor complejidad en el acceso a los datos.
- La comunicación y la sincronización entre las diferentes subtareas.

La computación paralela resuelve problemas como: predicciones y estudios meteorológicos, estudio del genoma humano, modelado de la biosfera, predicciones sísmicas, simulación de moléculas... En algunos

casos se dispone de tal cantidad de datos que sería muy lento o imposible tratarlos con máquinas convencionales.

3.2 Programación distribuida

Uno de los motivos principales para construir un sistema distribuido es compartir recursos. Probablemente, el sistema distribuido más conocido por todos es Internet que permite a los usuarios donde quiera que estén hacer uso de la World Wide Web, el correo electrónico y la transferencia de ficheros. Entre las aplicaciones más recientes de la computación distribuida se encuentra el Cloud Computing que es la computación en la nube o servicios en la nube, que ofrece servicios de computación a través de Internet.

Se define un sistema distribuido como aquel en el que los componentes hardware o software, localizados en computadores unidos mediante una red, comunican y coordinan sus acciones mediante el paso de mensajes. Esta definición tiene las siguientes consecuencias:

- **Concurrencia**: lo normal en una red de ordenadores es la ejecución de programas concurrentes.
- **Inexistencia de reloj global**: cuando los programas necesitan cooperar coordinan sus acciones mediante el paso de mensajes.
- **Fallos independientes**: cada componente del sistema puede fallar independientemente, permitiendo que los demás continúen su ejecución.

La programación distribuida es un paradigma de programación enfocado en desarrollar sistemas distribuidos, abiertos, escalables, transparentes y tolerantes a fallos. Este paradigma es el resultado natural del uso de las computadoras y las redes. Casi cualquier lenguaje de programación que tenga acceso al máximo al hardware del sistema puede manejar la programación distribuida, considerando una buena cantidad de tiempo y código.

Una arquitectura típica para el desarrollo de sistemas distribuidos es la arquitectura **cliente-servidor**. Los clientes son elementos activos que demandan servicios a los servidores realizando peticiones y esperando la respuesta, los servidores son elementos pasivos que realizan las tareas bajo requerimientos de los clientes.

Por ejemplo, un cliente web solicita una página, el servidor web envía al cliente la página solicitada. La comunicación entre servidores y clientes se realiza a través de la red.

Existen varios modelos de programación para la comunicación entre los procesos de un sistema distribuido:

Sockets. Proporcionan los puntos extremos para la comunicación entre procesos. Es actualmente la base de la comunicación. Pero al ser de muy bajo nivel de abstracción, no son adecuados a nivel de aplicación. En el Capítulo 3 se tratarán los sockets en Java.

Llamada de procedimientos remotos o RPC (Remote Procedure Call). Permite a un programa cliente llamar a un procedimiento de otro programa en ejecución en un proceso servidor. El proceso servidor define en su interfaz de servicio los procedimientos disponibles para ser llamados remotamente.

Invocación remota de objetos. El modelo de programación basado en objetos ha sido extendido para permitir que los objetos de diferentes procesos se comuniquen uno con otro por medio de una invocación a un método remoto o RMI (Remote Method Invocation). Un objeto que vive en un proceso puede invocar métodos de un objeto que reside en otro proceso. Java RMI extiende el modelo de objetos de Java para proporcionar soporte de objetos distribuidos en lenguaje Java.

VENTAJAS

- Ventajas que aportan los sistemas distribuidos:
- Se pueden compartir recursos y datos.
- Capacidad de crecimiento incremental.
- Mayor flexibilidad al poderse distribuir la carga de trabajo entre diferentes ordenadores.
- Alta disponibilidad.
- Soporte de aplicaciones inherentemente distribuidas.
- Carácter abierto y heterogéneo.

INCONVENIENTES

- Aumento de la complejidad, se necesita nuevo tipo de software.
- Problemas con las redes de comunicación: pérdida de mensajes, saturación del tráfico.
- Problemas de seguridad, como por ejemplo, ataques de denegación de servicio en la que se "bombardea" un servicio con peticiones inútiles de forma que un usuario interesado en usar el servicio no pueda usarlo.