TEMA 1.- PROGRAMACIÓN MULTIPROCESO.

Objetivos

- Ejecutables. Procesos. Servicios
- Estados de un proceso.
- Hilos.
- Programación concurrente.
- Programación paralela y distribuida.
- Comunicación entre procesos.
- Gestión de procesos.
- Sincronización entre procesos.
- Programación de aplicaciones multiproceso.

Contenidos

1.- Ejecutables.

Un ejecutable es un archivo con la estructura necesaria para que el sistema operativo pueda poner en marcha el programa que hay dentro (En Windows ***.exe***).

Sin embargo, Java genera ficheros ***.jar*** o ***.class***. *Estos ficheros no son ejecutables sino que son archivos que el intérprete de JAVA (el archivo java.exe) leerá y ejecutará.*

El intérprete toma el programa y lo traduce a instrucciones del microprocesador en el que estemos, que puede ser x86 o un x64 o lo que sea. Ese proceso se hace «al instante» o JIT (Just-In-Time).

Un archivo ***.class*** puede desensamblarse utilizando el comando ***javap -c <archivo.class>***. Cuando se hace así, se obtiene un listado de «instrucciones» que no se corresponden con las instrucciones del microprocesador, sino con «instrucciones virtuales de Java». El intérprete Java traducirá en el momento del arranque dichas instrucciones virtuales Java a instrucciones reales del microprocesador.

Este último aspecto es el esgrimido por Java para defender que su ejecución puede ser más rápida que la de un EXE, ya que Java puede averiguar en qué microprocesador se está ejecutando y así generar el código más óptimo posible.

Un EXE puede que no contenga las instrucciones de los microprocesadores más modernos. Como todos son compatibles no es un gran problema, sin embargo, puede que no aprovechemos al 100% la capacidad de nuestro micro.

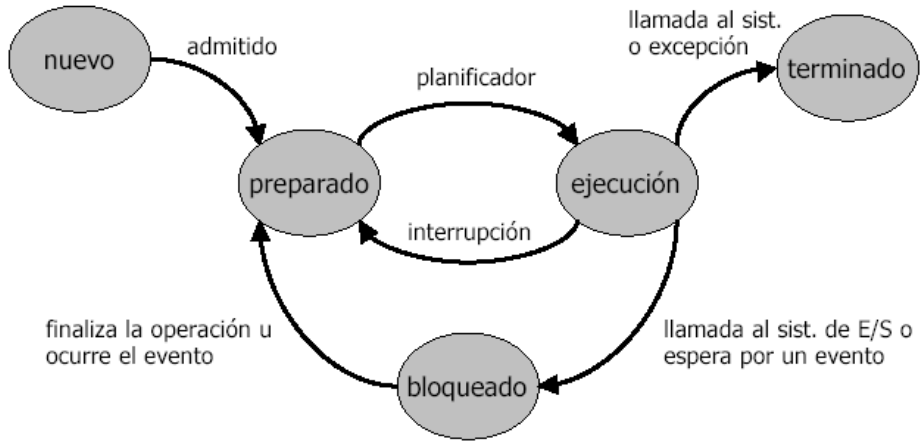
2.- Procesos.

Es un archivo que está en ejecución y bajo el control del sistema operativo.

Un proceso puede atravesar diversas etapas en su «ciclo de vida». Los estados en los que puede estar son:

- **Nuevo (new):** el proceso se está creando.
- **En ejecución (running):** el proceso está en la CPU ejecutando instrucciones.
- **Bloqueado (waiting, en espera):** proceso esperando a que ocurra un suceso (ej. terminación de E/S o recepción de una señal).
- **Preparado (ready, listo):** esperando que se le asigne a un procesador.
- **Terminado (terminated):** finalizó su ejecución, por tanto no ejecuta más instrucciones y el SO le retirará los recursos que consume.

Diagrama de estados de un proceso



Multiproceso

El multiproceso consiste en la ejecución de varios procesos diferentes de forma *simultánea* para la realización de una o varias tareas relacionadas o no entre sí.
Cada uno de estos procesos es una aplicación independiente.
El caso más conocido es aquel en el que nos referimos al Sistema Operativo (Windows, Linux, MacOS, . . .) y decimos que es *multitarea* puesto que es capaz de ejecutar varias tareas o procesos (o programas) al mismo tiempo.

Creación de procesos con Java

El método start() de la clase ProcessBuilder de Java permite crear instancias de procesos Process. La instancia de ProcessBuilder recibe como parámetros el comando y los parámetros de ejecución.

Por ejemplo, podemos ejecutar en la consola CMD el comando DIR para obtener el contenido del directorio activo. El parámetro /C ejecuta el comando y cierra automáticamente el interprete de comandos CMD.

```
ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");
```

Ejemplo1:

```
import java.io.IOException;
public class Ejemplo1 {
   public static void main(String[] args) throws IOException  {
         ProcessBuilder pb = new ProcessBuilder("NOTEPAD");
         Process p = pb.start();
   }
}
```

Ejemplo1b:

```
import java.io.IOException;
public class Ejemplo1b {
   public static void main(String[] args) throws IOException  {
         Process p = new ProcessBuilder("NOTEPAD").start();
   }
}
```

Ejemplo1c:

```
import java.io.IOException;
public class Ejemplo1c {
   public static void main(String[] args) throws IOException  {
         ProcessBuilder pb = new ProcessBuilder("NOTEPAD");
         Process p1 = pb.start();
         Process p2 = pb.start();
   }
}
```

Entrada y Salida de procesos con Java

La clase Process dispone de métodos que permiten realizar operaciones con el proceso como obtener la salida del proceso, realizar una entrada al proceso, etc... Los métodos getOutputStream(), getInputStream() y getErrorStream() permiten operaciones de E/S sobre el proceso.

Ejemplo2. El método getInputStream() lee la salida del proceso, en este caso la salida de la ejecución del comando DIR en el interprete de comandos. El método read() lee carácter a carácter la salida del proceso. Por último, el método waitFor() espera hasta que termina el proceso y devuelve 0 si todo ha ido bien o 1 si se ha producido algún error.

```
import java.io.*;
public class Ejemplo2 {
    public static void main(String[] args) throws IOException {
      Process p = new ProcessBuilder("CMD", "/C", "DIR").start();
      try {
```

```
            // Recoge la salida del proceso
            InputStream salida = p.getInputStream();
            // muestra caracter a caracter la salida del proceso
            int c;
            while ((c = salida.read()) != -1){
                System.out.print((char) c);
            }
            // cierra el flujo de salida del proceso
            salida.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        // COMPROBACION DE ERROR - 0 bien - 1 mal
        int exitVal;
        try {
            // Obtiene y muestra el valor de finalización del proceso
            exitVal = p.waitFor();
            System.out.println("Valor de Salida: " + exitVal);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Ejemplo2Error. El método getInputStream() lee la salida del proceso, en este caso la salida de la ejecución del comando DIRR en el interprete de comandos. Dicho comando no existe lo que devolverá un error. El método read() lee carácter a carácter la salida del proceso. Por último, el método waitFor() espera hasta que termina el proceso y devuelve 0 si todo ha ido bien o 1 si se ha producido algún error.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

public class Ejemplo2Error {
    public static void main(String[] args) throws IOException {
      Process p = new ProcessBuilder("CMD", "/C", "DIRR").start();
      try {
            // Recoge la salida del proceso
            InputStream salida = p.getInputStream();
            // mostramos en pantalla caracter a caracter la salida del
proceso
            int c;
            while ((c = salida.read()) != -1){
```

```
                System.out.print((char) c);
            }
            // cierra el flujo de salida del proceso
            salida.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        // COMPROBACION DE ERROR - 0 bien - 1 mal
        int exitVal;
        try {
            // Obtiene y muestra el valor de finalización del proceso
            exitVal = p.waitFor();
            System.out.println("Valor de Salida: " + exitVal);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            // Creamos flujo de datos para obtener el mensaje de error
            InputStream er = p.getErrorStream();

            BufferedReader brer = new BufferedReader(new
InputStreamReader(er));
            // Cadena que almacenará el error
            String mensajeError = null;
            System.out.print("ERROR: ");
            // Mostramos el mensaje de error leyendolo línea a línea
            while ((mensajeError = brer.readLine()) != null){
                System.out.println(mensajeError);
            }
        } catch (IOException eErr) {
            eErr.printStackTrace();
        }
    }
}
```

Ejemplo3. Obtenemos la salida de la ejecución de un programa java. Previamente hemos debido compilar ambos programas y ejecutar Ejemplo3 desde la consola de comandos.

```
import java.io.*;
public class Ejemplo3 {
  public static void main(String[] args) throws IOException {
      //El proceso a ejecutar es Ejemplo2
      ProcessBuilder pb = new ProcessBuilder("java", "Ejemplo2");
        // Se ejecuta el proceso
      Process p = pb.start();
        //obtener la salida devuelta por el proceso
```

```
      try {
            InputStream is = p.getInputStream();
            int c;
            while ((c = is.read()) != -1)
                  System.out.print((char) c);
            is.close();
      } catch (Exception e) {
            e.printStackTrace();
      }
  }
```

Salida de la ejecución:

```
C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP\src\tema1_psp>javac Ejemplo2.java
C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP\src\tema1_psp>javac Ejemplo3.java
C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP\src\tema1_psp>java Ejemplo3

 El volumen de la unidad C es OS
 El número de serie del volumen es: 00C3-EC8A

 Directorio de C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP\src\tema1_psp

08/08/2019  01:59    <DIR>          .
08/08/2019  01:59    <DIR>          ..
08/08/2019  01:56               456 Ejemplo1.class
08/08/2019  01:43               233 Ejemplo1.java
08/08/2019  02:02             1.330 Ejemplo2.class
08/08/2019  02:02               676 Ejemplo2.java
08/08/2019  01:12             1.538 Ejemplo2Error.java
08/08/2019  02:02               952 Ejemplo3.class
08/08/2019  02:01               818 Ejemplo3.java
08/08/2019  01:07               446 Tema1_PSP.java
               8 archivos          6.449 bytes
               2 dirs  156.135.395.328 bytes libres
Valor de Salida: 0

C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP\src\tema1_psp>
```

Ejercicio examen

Ejemplo5. El método getOutputStream() nos permite escribir en el stream de entrada de un proceso y así enviarle datos. El método write() envía los datos al stream y el método getBytes() codifica la cadena en una secuencia de bytes.

El proceso destino, recoge los bytes enviados mediante InputStreamReader() o la clase Scanner y los procesa.

```
import java.io.*;
public class Ejemplo5 {
    public static void main(String[] args) throws IOException {
       Process p = new ProcessBuilder("java", "EjemploLectura").start();
        // Envia entrada al fichero Ejemplo Lectura utilizando getOutputStream()
       OutputStream entrada = p.getOutputStream();
       entrada.write("Curso DAM\n".getBytes());
```

```
        entrada.flush(); // vacia el buffer de salida
        try {
            // Recoge la salida del proceso
            InputStream salida = p.getInputStream();
            // mostramos en pantalla caracter a caracter la salida del proceso
            int c;
            while ((c = salida.read()) != -1){
                System.out.print((char) c);
            }
            // cierra el flujo de salida del proceso
            salida.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        // COMPROBACION DE ERROR - 0 bien - 1 mal
        int exitVal;
        try {
            // Obtiene y muestra el valor de finalización del proceso
            exitVal = p.waitFor();
            System.out.println("Valor de Salida: " + exitVal);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            // Creamos flujo de datos para obtener el mensaje de error
            InputStream er = p.getErrorStream();
            BufferedReader brer = new BufferedReader(new InputStreamReader(er));
            // Cadena que almacenará el error
            String mensajeError = null;
            System.out.print("ERROR: ");
            // Mostramos el mensaje de error leyendolo línea a línea
            while ((mensajeError = brer.readLine()) != null){
                System.out.println(mensajeError);
            }
        } catch (IOException eErr) {
            eErr.printStackTrace();
        }
    }
}
```

```
import java.io.*;
public class EjemploLectura{
    public static void main (String [] args) {
        // Instancia para recoger los datos enviados con getOutputStream()
        // creando un flujo de datos y almacenarlos en la variable texto
        InputStreamReader entrada = new InputStreamReader(System.in);
        BufferedReader flujoEntrada = new BufferedReader (entrada);
        String texto;
        try {
            // Recoge y muestra datos a través del flujo de datos de entrada
            // Cierra el flujo de datos de entrada
            texto= flujoEntrada.readLine();
            System.out.println("Cadena escrita: "+texto);
            entrada.close();
        }catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }
}
```

Ejecución y Salida:

```
C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP\src\tema1_psp> javac Ejemplo5.java
C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP\src\tema1_psp>                javac
EjemploLectura.java
C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP\src\tema1_psp> java Ejemplo5
Cadena escrita: Curso DAM
Valor de Salida: 0
ERROR:
```

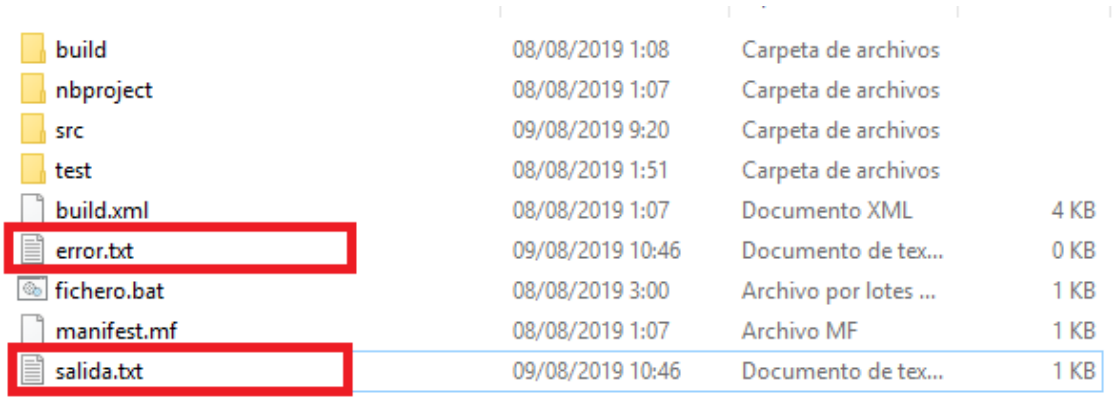
Redirección de la entrada y salida de procesos con Java

Los métodos redirectOutput y redirectError permiten la salida estandar y de error a un fichero. El método redirectInput() permite recoger la entrada a un proceso del contenido de un fichero.

Ejemplo6. Ejecuta un proceso que lista los ficheros del directorio actual y redirige la salida y el error a los ficheros *salida.txt* y *error.txt*.

```
import java.io.File;
import java.io.IOException;

public class Ejemplo6 {
   public static void main(String args[]) throws IOException {
      // Crea el proceso que ejecuta el comando dir en el interprete de comandos
      ProcessBuilder pb = new ProcessBuilder("CMD","/C" ,"DIR");
      // Establece los ficheros a los que redigirá la salida y el error
      File fOut = new File("salida.txt");
      File fErr = new File("error.txt");
      // Indica que la redirección del proceso se realizará sobre los ficheros
      pb.redirectOutput(fOut);
      pb.redirectError(fErr);
      // Ejecuta el proceso
      pb.start();
   }
}
```



Fichero salida.txt

```
El volumen de la unidad C es OS
```

```
 El número de serie del volumen es: 00C3-EC8A

 Directorio de C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP

09/08/2019  10:46    <DIR>          .
09/08/2019  10:46    <DIR>          ..
08/08/2019  01:08    <DIR>          build
08/08/2019  01:07             3.612 build.xml
09/08/2019  10:46                 0 error.txt
08/08/2019  03:00                31 fichero.bat
08/08/2019  01:07                85 manifest.mf
08/08/2019  01:07    <DIR>          nbproject
09/08/2019  10:46                 0 salida.txt
09/08/2019  09:20    <DIR>          src
08/08/2019  01:51    <DIR>          test
               5 archivos          3.728 bytes
               6 dirs  156.119.896.064 bytes libres
```

Ejemplo7. Lanza un proceso que envía el comando DIRR en la consola de comandos y redirige la salida y el error a los ficheros *salida.txt* y *error.txt*.

```
import java.io.File;
import java.io.IOException;

public class Ejemplo6 {
   public static void main(String args[]) throws IOException {
      // Crea el proceso que ejecuta el comando dir en el interprete de comandos
      ProcessBuilder pb = new ProcessBuilder("CMD","/C" ,"DIRR");
      // Establece los ficheros a los que redigirá la salida y el error
      File fOut = new File("salida.txt");
      File fErr = new File("error.txt");
      // Indica que la redirección del proceso se realizará sobre los ficheros
      pb.redirectOutput(fOut);
      pb.redirectError(fErr);
      // Ejecuta el proceso
      pb.start();
   }
}
```

Fichero error.txt

```
"DIRR" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.
```

Ejemplo8. Lanza un proceso que toma la entrada del fichero fichero.bay y redirige la salida y el error a los ficheros salida.txt y error.txt.

```
import java.io.File;
import java.io.IOException;
public class Ejemplo8 {
    public static void main(String args[]) throws IOException {
        // Crea el proceso
        ProcessBuilder pb = new ProcessBuilder("CMD");
        // Establece los ficheros a los que redigirá la salida y el error
        // Establece la entrada al proceso desde el fichero fichero.bat
```

```
        File fBat = new File("fichero.bat");
        File fOut = new File("salida.txt");
        File fErr = new File("error.txt");
        // Indica que la redirección del proceso se realizará sobre los ficheros
        pb.redirectInput(fBat);
        pb.redirectOutput(fOut);
        pb.redirectError(fErr);
        // Ejecuta el proceso
        pb.start();
    }
}
```

Fichero fichero.bat

```
DIR
DIRR
```

Fichero salida.txt

```
Microsoft Windows [Versi¢n 10.0.17763.615]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Mario G\Documents\NetBeansProjects\Tema1_PSP>DIR
 El volumen de la unidad C es OS
 El numero de serie del volumen es: 00C3-EC8A

 Directorio de C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP

09/08/2019  11:17    <DIR>          .
09/08/2019  11:17    <DIR>          ..
08/08/2019  01:08    <DIR>          build
08/08/2019  01:07             3.612 build.xml
09/08/2019  11:17                 0 error.txt
09/08/2019  11:17                11 fichero.bat
08/08/2019  01:07                85 manifest.mf
08/08/2019  01:07    <DIR>          nbproject
09/08/2019  11:17                 0 salida.txt
09/08/2019  09:20    <DIR>          src
08/08/2019  01:51    <DIR>          test
               5 archivos          3.708 bytes
               6 dirs  156.115.238.912 bytes libres

C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP>DIRR

C:\Users\Aula\Documents\NetBeansProjects\Tema1_PSP>
```

Fichero error.txt

```
"DIRR" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.
```

Redirección por defecto.

La redirección especial Redirect.INHERIT indica que la fuente de entrada y salida del proceso será la misma que la del proceso actual.

Ejemplo9. Redirect.INHERIT muestra por consola la salida del comando DIR.

```
import java.io.IOException;
```

```
public class Ejemplo9 {
    public static void main(String args[]) throws IOException {
        // Crea el proceso.
        ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");
   // Establece salida del proceso a la misma que el proceso actual (consola)
   pb.redirectOutput(ProcessBuilder.Redirect.INHERIT);
   // Ejecuta el proceso
        Process proceso = pb.start();
    }
}
```

La salida del proceso sea en la salida por defecto que es la pantalla, así nos ahorramos el while

3.- Gestión de Procesos.

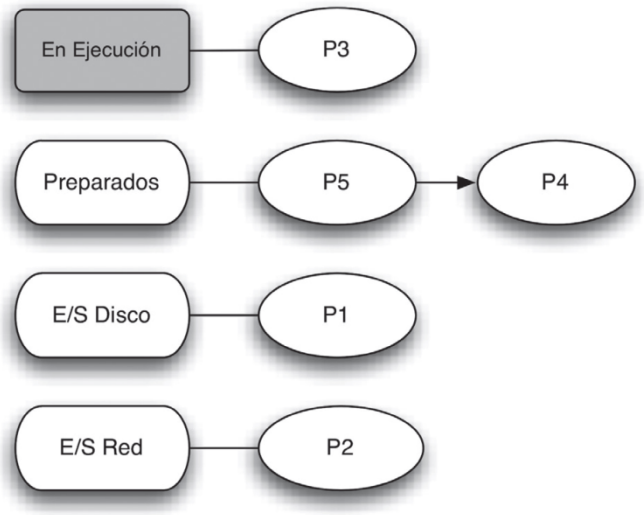
Uno de los objetivos del sistema operativo es la ***multiprogramación***, es decir, admitir varios procesos en memoria para maximizar el uso del procesador. Esto funciona ya que los procesos se irán intercambiando el uso del procesador para su ejecución de forma concurrente.

Para ello, el sistema operativo organiza los procesos en varias colas, migrándolos de unas colas a otras:

- Una cola de procesos: contiene todos los procesos del sistema.
- Una cola de procesos preparados: contiene todos los procesos listos esperando para ejecutarse.
- Varias colas de dispositivo: contienen los procesos que están a la espera de alguna operación de E/S.

Supuesto práctico:

Supongamos que varios procesos (1, 2, 3, 4 y 5) se están ejecutando concurrentemente en un ordenador. El proceso 1 se está ejecutando cuando realiza una petición de E/S al disco. En ese momento, el proceso 2 pasa a ejecución, haciendo otra operación de E/S, en este caso por la tarjeta de red. El proceso 3 pasa a ejecutarse. En ese momento, las colas de procesos serán las siguientes:



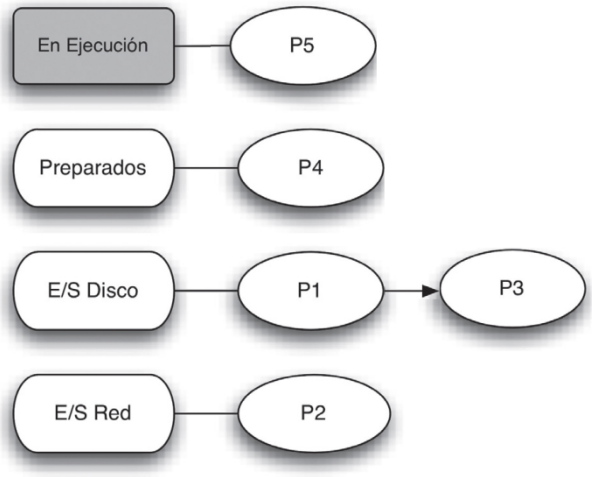
Si en ese momento el proceso 3 realizará una petición de E/S teniendo que esperar por el disco duro:

1. ¿Cómo quedaría el estado de las colas correspondientes?
2. Identifica cuál sería el proceso que se ejecutaría utilizando la filosofía FIFO para sacar los procesos de las colas.

Supuesto práctico: (*Solución*)

Si en ese momento el proceso 3 realizará una petición de E/S teniendo que esperar por el disco duro:

1. **¿Cómo quedaría el estado de las colas correspondientes?**



2. **Identifica cuál sería el proceso que se ejecutaría utilizando la filosofía FIFO para sacar los procesos de las colas.**

   P5, ya que es el primero en la cola de preparados.

4.- Servicios.

Un servicio es un proceso que no muestra ninguna ventana ni gráfico en pantalla porque no está pensado para que el usuario lo maneje directamente. Habitualmente, un servicio es un programa que atiende a otro programa.

5.- Hilos.

Un proceso tiene su propio espacio en memoria. Si abrimos 20 procesos cada uno de ellos consume 20x de memoria RAM. Además, un proceso no tiene acceso a los datos de otro proceso.

Sin embargo, un hilo es un concepto más avanzado que un proceso. Un hilo es un proceso mucho más ligero, en el que el código y los datos se comparten de una forma distinta. Además, un hilo sí accede a los datos de otro hilo. *Esto complicará algunas cuestiones a la hora de programar.*

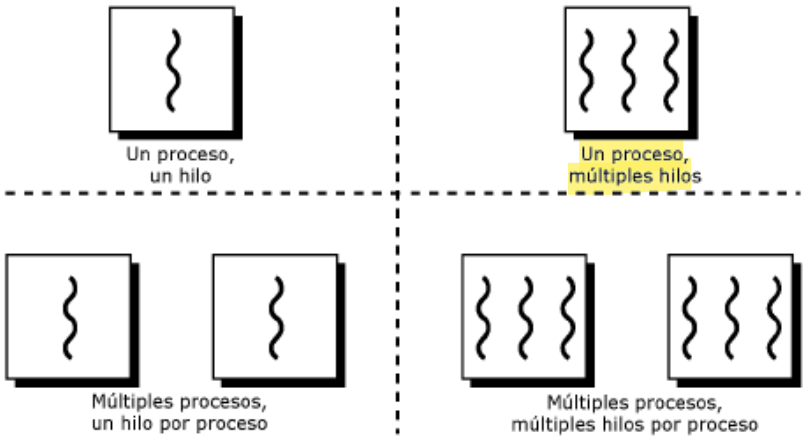
Los principales estados de un hilo son ***ejecución, preparado y bloqueado***; y hay cuatro operaciones básicas relacionadas con el cambio de estado de los hilos:

- *Creación:* Cuando se crea un proceso se crea también un hilo para ese proceso. Posteriormente, ese hilo puede crear nuevos hilos dándoles un puntero de instrucción y algunos argumentos.
- *Bloqueo:* Cuando un hilo debe esperar por un suceso, se le bloquea guardando sus registros. Así el procesador pasará a ejecutar otro hilo preparado.
- *Desbloqueo:* Cuando se produce el suceso por el que un hilo se bloqueó pasa a la cola de listos.
- *Terminación:* Cuando un hilo finaliza, se liberan su contexto y sus pilas.

Multihilo

Hablamos de multihilo cuando se ejecutan varias tareas relacionadas o no entre sí dentro de una misma aplicación. En este caso no son procesos diferentes sino que dichas tareas se ejecutan dentro del mismo proceso del Sistema Operativo. A cada una de estas tareas se le conoce como hilo o thread. Esto se conoce como **Programación Concurrente**.

Procesos vs Hilos.



Semejanzas: Los hilos operan, en muchos sentidos, igual que los procesos.

- Pueden estar en uno o varios estados: listo, bloqueado, en ejecución o terminado.
- También comparten la CPU.
- Sólo hay un hilo activo (en ejecución) en un instante dado.
- Un hilo dentro de un proceso se ejecuta secuencialmente.
- Cada hilo tiene su propia pila y contador de programa.
- Pueden crear sus propios hilos hijos.

Diferencias: Los hilos, a diferencia de los procesos, no son independientes entre sí.

- Como todos los hilos pueden acceder a todas las direcciones de la tarea, un hilo puede leer la pila de cualquier otro hilo o escribir sobre ella. Aunque pueda parecer lo contrario la protección no es necesaria ya que el diseño de una tarea con múltiples hilos tiene que ser un usuario único.

Ventajas: de los hilos sobre los procesos.

- Se tarda mucho menos tiempo en crear un nuevo hilo en un proceso existente que en crear un nuevo proceso.
- Se tarda mucho menos tiempo en terminar un hilo que un proceso.
- Se tarda mucho menos tiempo en conmutar entre hilos de un mismo proceso que entre procesos.
- Los hilos hacen más rápida la comunicación entre procesos, ya que al compartir memoria y recursos, se pueden comunicar entre sí sin invocar el núcleo del SO.

Ejemplos de uso de los hilos:

- *Trabajo en segundo plano:* En una aplicación móvil, una petición a internet, ejecución de alguna tarea costosa mientras sigues utilizando la app, para así no bloquearla…
- *Procesamiento asíncrono:* Se podría implementar, con el fin de protegerse de cortes de energía, un hilo que se encargará de salvaguardar el buffer de un procesador de textos una vez por minuto.
- *Estructuración modular de los programas:* Los programas que realizan una variedad de actividades que no dependan unas de otras, se pueden diseñar e implementar mediante hilos.

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:

- Un único procesador con un único núcleo.
- Varios núcleos en un mismo procesador.
- Varios ordenadores distribuidos en red.

6.- Programación Concurrente.

Las tareas se ejecutan en un solo procesador con un único núcleo.
Aunque para el usuario parezca que varios procesos se ejecutan al mismo tiempo, solamente un proceso puede estar en un momento determinado en ejecución.
El SO se encarga de cambiar el proceso en ejecución después de un período corto de tiempo (ms).
No mejora el tiempo de ejecución global de los programas.

7.- Programación Paralela.

Las tareas se ejecutan en varios núcleos de un solo procesador.
Cada núcleo puede ejecutar 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.
Todos los núcleos 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.

8.- 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 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.

9.- Programación Concurrente en Java.

La Progrmación Concurrente o multiprogramación, puede producirse entre procesos totalmente independientes, como podrían ser los correspondientes al procesador de textos, navegador, reproductor de música, etc., o entre procesos que pueden cooperar entre sí para realizar una tarea común. El SO se encarga de gestionar todo esto.

Sin embargo, *si se pretende realizar procesos que cooperen entre sí debe ser el propio desarrollador quien lo implemente utilizando la comunicación y sincronización de procesos e hilos.*

A la hora de realizar un programa multiproceso cooperativo, se deben seguir las siguientes fases:

1.- **Descomposición funcional**: Es necesario identificar previamente las diferentes funciones que debe realizar la aplicación y las relaciones existentes entre ellas.
2.- Partición: Distribución de las diferentes funciones en procesos estableciendo el esquema de comunicación entre los mismos. Al ser procesos cooperativos necesitarán información unos de otros por lo que deben comunicarse.
3.- **Implementación:** Una vez realizada la descomposición y la partición, se realiza la implementación utilizando las herramientas disponibles por la plataforma (java).

IMPORTANTE: Hay que tener en cuenta que la comunicación entre procesos requiere una pérdida de tiempo tanto de comunicación como de sincronización. Por tanto, el objetivo debe ser maximizar la independencia entre los procesos minimizando la comunicación entre los mismos.

Para programar concurrentemente debemos dividir nuestro programa en hilos. Java proporciona la construcción de programas concurrentes mediante la clase Thread (hilo o hebra). Esta clase permite la ejecución de código en un hijo de manera independiente.

En Java existen dos formas de crear o utilizar hilos:

a) Creando un clase que herede de Thread y sobrecargando el método run().
b) Implementando la interface Runnable y declarando el método run(). Se utiliza cuando una clase ya deriva de otra.

Ejemplo10. Crear un hilo con nombre HiloSimple heredado de Thread. En el método run() se incluyen las líneas de código que se ejecutarán simultáneamente con las otras partes del programa. Cuando termina la ejecución del método, también termina el hilo de ejecución. Se ha añadido la sentencia Thread().sleep(x) para ralentizar en x milisegundos la ejecución del código en cada vuelta del bucle.

```
import java.io.IOException;
public class Ejemplo10 {
    public static void main(String[] args) throws IOException  {
        HiloSimple hs = new HiloSimple();
        hs.start();
        for(int i=0; i<5; i++){
            try {
                System.out.println("Fuera del Hilo...");
                Thread.sleep(150);
            } catch (InterruptedException ex) {
                System.out.println(ex.getMessage());
            }
        }
    }

}
class HiloSimple extends Thread{
    @Override
    public void run(){
        for(int i=0; i<5; i++){
            try {
                System.out.println("En el Hilo...");
                Thread.sleep(100);
            } catch (InterruptedException ex) {
                System.out.println(ex.getMessage());
            }
        }
    }
}
```

Salida de la ejecución:

```
run:
Fuera del Hilo...
En el Hilo...
En el Hilo...
Fuera del Hilo...
En el Hilo...
En el Hilo...
Fuera del Hilo...
En el Hilo...
Fuera del Hilo...
Fuera del Hilo...
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejemplo11. Implementación del Ejemplo10 con Interface Runnable.

```
package tema1_psp;
import java.io.IOException;
public class Ejemplo11 {
    public static void main(String[] args) throws IOException  {
        HiloSimple hs = new HiloSimple();
        Thread hilo = new Thread(hs);
        hilo.start();
        for(int i=0; i<5; i++){
            try {
                System.out.println("Fuera del Hilo...");
                Thread.sleep(150);
            } catch (InterruptedException ex) {
                System.out.println(ex.getMessage());
            }
        }
    }

}
class HiloSimple implements Runnable{
    @Override
    public void run(){
        for(int i=0; i<5; i++){
            try {
                System.out.println("En el Hilo...");
                Thread.sleep(100);
            } catch (InterruptedException ex) {
                System.out.println(ex.getMessage());
            }
        }
    }
}
```

Salida de la ejecución:

```
run:
Fuera del Hilo...
En el Hilo...
En el Hilo...
Fuera del Hilo...
En el Hilo...
Fuera del Hilo...
En el Hilo...
En el Hilo...
Fuera del Hilo...
Fuera del Hilo...
BUILD SUCCESSFUL (total time: 0 seconds)
```