Tema 1

Programación multiproceso

2º Desarrollo Aplicaciones Multiplataforma

ÍNDICE

1. Introducción

- En la actualidad todos nuestros ordenadores son capaces de realizar diferentes tareas a la vez.

  Por ejemplo: ejecutar una IDE de programación y estar conectados a Teams para seguir una clase online.

- En S.O. **multiproceso o multitarea** podemos ejecutar más de un proceso (programa) a la vez pero nosotros como usuarios percibiremos cada proceso como si el único en ejecución.

- Tenemos dos soluciones:
  - Utilizar varias CPUs, cada una ejecute un proceso.
  - Compartir el tiempo de la CPU entre los diferentes procesos.

¿Qué es la programación multiproceso?

Capacidad de que una aplicación sea capaz de realizar varias tareas de forma simultánea.

1. Introducción

- **Programas, proceso, ejecutable y servicio** son términos que hacen referencia a elementos distintos, pero íntimamente relacionados.

- El sistema operativo es a quién se indica qué programas se desean ejecutar.

- Para llegar a poder ejecutar un programa primero hay que obtenerlo o, en el caso de los programadores, crearlo. El proceso es el siguiente:
  1. El programador escribe el **código fuente** con el editor de texto y lo almacena en un fichero.
  2. El programador compila el código fuente utilizando un compilador, generando **un programa ejecutable**. Este nuevo programa contiene instrucciones comprensibles por el sistema operativo.
  3. El usuario ejecuta el programa ejecutable, generando un **proceso**.

- Un **servicio** es también un programa cuya ejecución re realiza en segundo plano y que no requiere la interacción del usuario.

ÍNDICE

2. Proceso

- Se puede definir a un proceso como **un programa en ejecución**.
- En nuestros ordenadores todos los programas se organizan como un conjunto de procesos.
- Cada proceso está compuesto por:
    - Las instrucciones que se van a ejecutar.
    - El estado del propio proceso.
    - El estado de la ejecución.
    - El estado de la memoria.
- Se denomina **contexto** a toda la información que determina el estado de un proceso en un instante dado.

2. Proceso

Los pasos que se van a realizar para llevar a cabo un **cambio de contexto** son los siguientes:

- Guardar el estado del proceso actual.
- Determinar el siguiente proceso que se va a ejecutar.
- Recuperar y restaurar el estado del siguiente proceso.
- Continuar con la ejecución del siguiente proceso.

2. Proceso

- El S.O. el encargo de decidir su ejecución; si un proceso se suspende temporalmente, deberá rearrancarse posteriormente en el mismo estado que estaba al suspenderse lo que implica que será necesario guardar el estado del proceso al suspenderlo.
  - Dispondremos por tanto del **BCP(Bloque de Control de Proceso)** donde se almacenará información acerca de un proceso.

BCP	Identificación del proceso. Cada proceso que se inicia es referenciado por un identificador único.
	Estado del proceso
	Contador de programa
	Registros de CPU
	Información de planificación de CPU como unidad de proceso
	Información de gestión de memoria
	Información contable con la cantidad de tiempo de CPU y tiempo consumido real
	Información de estado de E/S como la lista de dispositivos asignados, archivos abiertos, etc…

2. BCP

- Un **BCP** contiene muchos elementos de información asociados con un proceso específico, entre los que se incluyen:

  • **Estado del proceso:** El estado puede ser nuevo, en ejecución, en espera, etc.

  • **Contador de programas:** Contiene la dirección de la siguiente instrucción a ejecutar por el proceso.

  • Registro de CPU: Varían en cuanto a número y tipo, dependiendo de la arquitectura de la computadora. Incluye acumuladores, registro de índices, punteros de pila y registros de propósito general.

  • **Información de planificación CPU:** Incluye prioridad del proceso.

  • **Información de gestión de memoria:** Incluye información acerca del mecanismo de gestión de memoria (las tablas de paginación, tablas de segmentos…)

  • **Información contable**: Contiene información acerca de la cantidad de CPU y tiempos empleados, los límites de tiempo asignados, el número de trabajo o de proceso.

  • **Información del estado de E/S**: Esta información incluye, solicitudes pendientes de E/S, dispositivos de E/S asignados al proceso, etc.

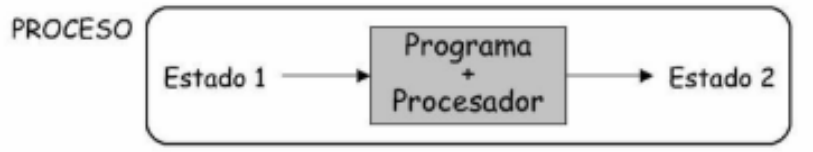
2. Proceso y programa

Un Programa Es estático. No varía nada
No tiene vector de estado → No requiere procesador
Variables sin valores
Rutinas sin dirección
Condiciones sin evaluar
Es el algoritmo a ejecutar

Un Proceso Es dinámico
Tiene vector de estado → Necesita procesador
Las variables tienen va ores
Las rutinas están en alguna dirección
Las condiciones se pueden evaluar
Es la ejecución del algoritmo

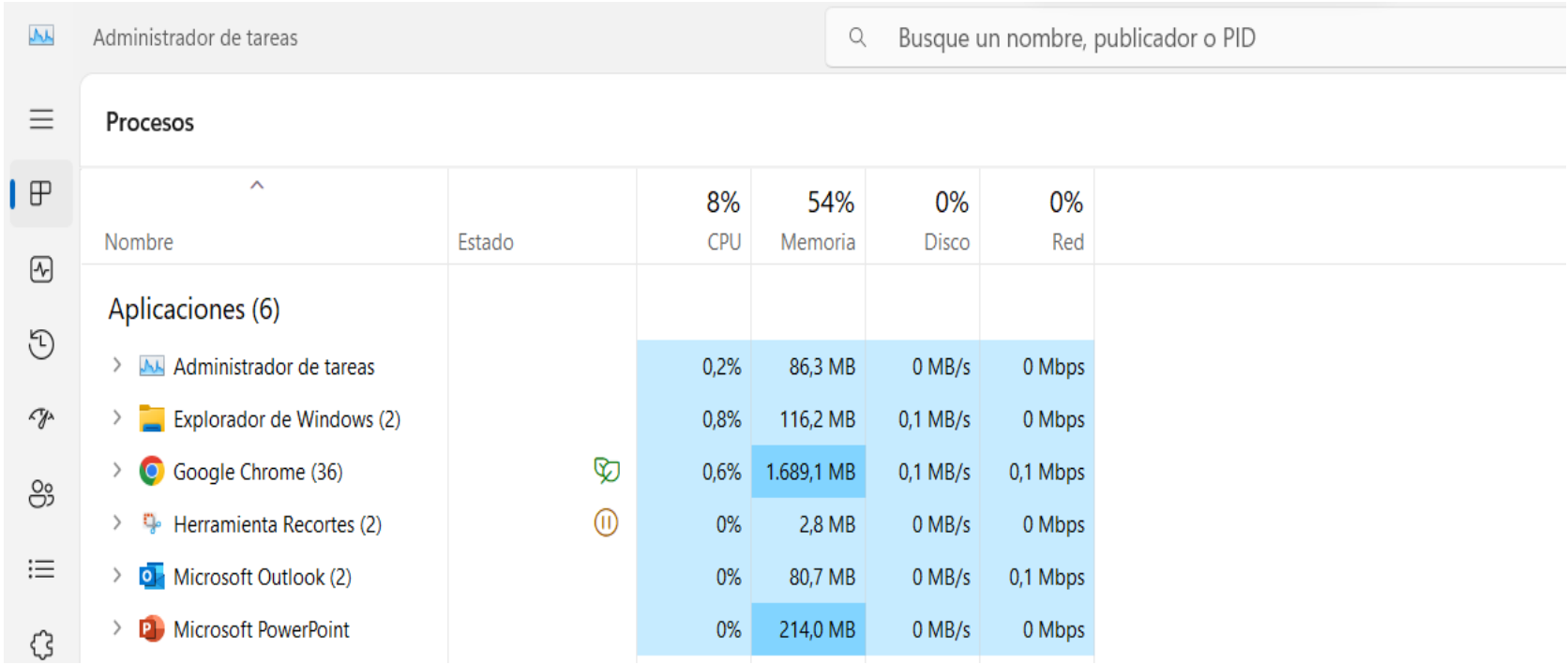
Un programa es una lista de instrucciones escritas en un papel, un fichero en disquete, disco duro, memoria RAM o cualquier otro soporte, pero el simple hecho de que estas instrucciones estén escritas no implica que se estén llevando a cabo.

Pues bien, cuando se leen estas instrucciones y se hacen ejecutar, entonces ya tenemos **programa+actividad**, es decir, un **proceso**.



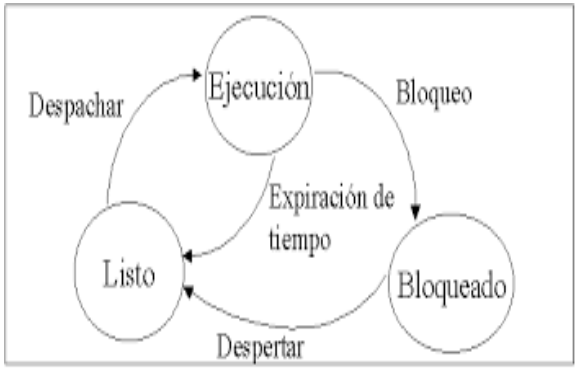
ACTIVIDAD

Ejemplo de procesos de un ordenador



2.2. Estados de un proceso

En ejecución	El proceso está actualmente ejecutándose, es decir, usando el procesador.
Bloqueado	A la espera de que ocurra un evento externo ajeno al planificador, como por ejemplo la finalización de una operación de E/S.
Listo	El proceso está en memoria, preparado para ejecutarse. A la espera de que el planificador le conceda tiempo de procesamiento.



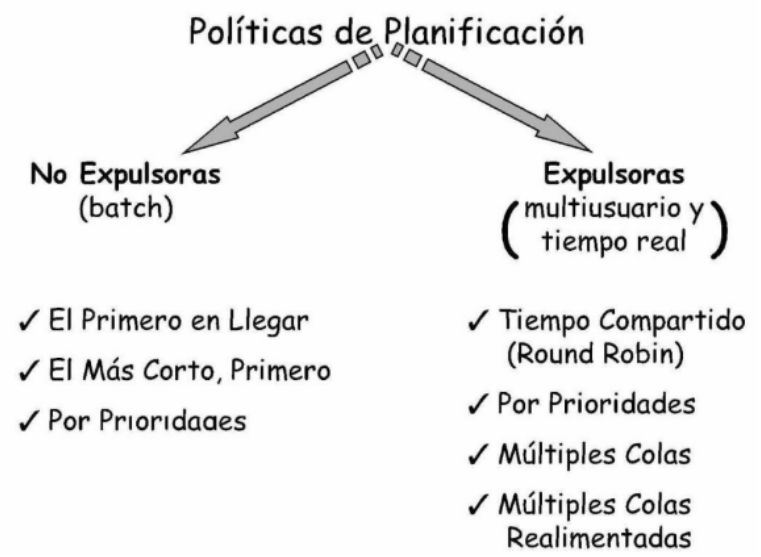
En ejecución - Bloqueado	Un proceso pasa de ejecución a bloqueado cuando espera la ocurrencia de un evento externo.
Bloqueado - Listo	Un proceso pasa de bloqueado a listo cuando ocurre el evento externo que se esperaba.
En ejecución - Listo	Un proceso pasa de ejecución a listo cuando se le acaba el tiempo asignado por el sistema operativo.

2.3. Planificador de procesos

El planificador se encarga de seleccionar el proceso que pasa a ejecución.

- Si los procesos de un sistema nunca dejan la CPU de forma involuntaria, se dice que la política de planificación de CPU es **no expulsora.**

- Si pueden perder la posesión del procesador sin solicitarlo, nos encontramos con una planificación **expulsora**.



2. Planificador de procesos

Objetivos del planificador:

- **Justicia (fairness):** que el proceso obtenga una porción de CPU "justa" o razonable.
- **Política:** que se satisfaga un determinado criterio establecido (ej. prioridades).
- **Equilibrio:** que todas las partes del sistema estén ocupadas haciendo algo.

2. Planificador de procesos

Existen una serie de **algoritmos de planificación** que le indican al planificador qué proceso se debe elegir para pasar a ejecutarse de los que están en espera. Estos algoritmos deben maximizar la utilización de la CPU, evitando que esté libre en algún momento, y minimizar el tiempo de respuesta de cada proceso.

En cada algoritmo sabremos para cada proceso:

- **Tiempo de entrada o de llegada al sistema (Ti)**: momento en el que el proceso entra en el sistema.

- **Tiempo de ejecución (Tx):** tiempo que el proceso necesita para su ejecución total.

Otros datos de interés que nos interesa obtener para cada proceso:

- **Tiempo de respuesta o de retorno (Tr):** tiempo que pasa desde que el proceso llega al sistema hasta que se obtienen los resultados

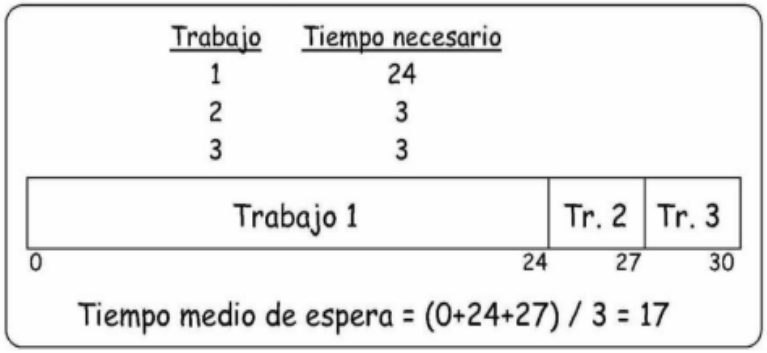
- **Tiempo de espera (Te):** tiempo que el proceso pasa dentro del sistema en espera

$$T_E = T_R - T_X$$

2.1. Planificación no expulsora

Algoritmos utilizados en las políticas no expulsoras:

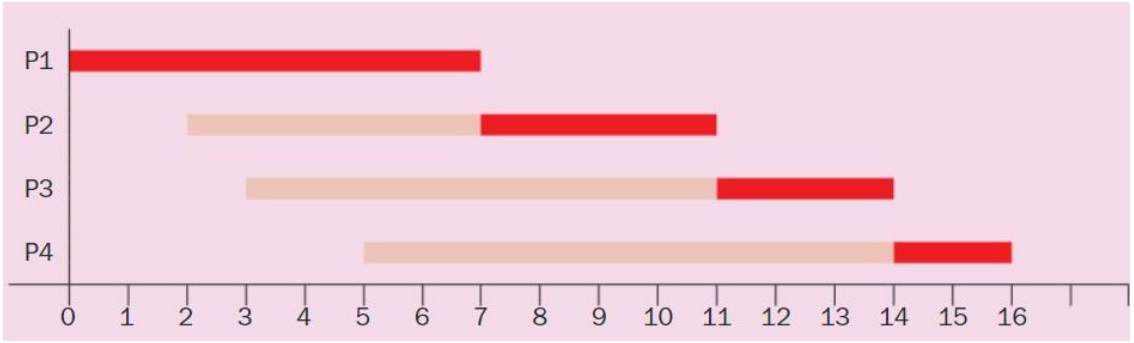
- **Primero en llegar - Primero en servir (FCFS: First Come - First Served ó FIFO: First Input Output):** el primer proceso que reclama la CPU, es el primero en conseguirla.



2.1. Planificación no expulsora

FCFS: First Come - First Served ó FIFO: First Input Output

PROCESO	TIEMPO DE LLEGADA	TIEMPO DE EJECUCIÓN
P1	0	7
P2	2	4
P3	3	3
P4	5	2

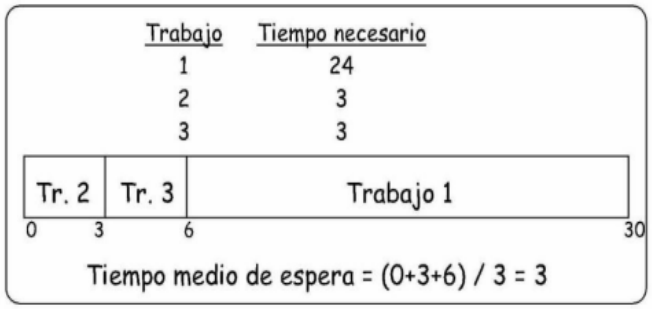


PROCESO	TIEMPO DE ESPERA	TIEMPO DE RESPUESTA
P1	0	7
P2	5	9
P3	8	11
P4	9	11
TIEMPOS MEDIOS	**5,5**	**9,5**

2.1. Planificación no expulsora

Algoritmos utilizados en las políticas no expulsoras:

- **El más corto primero (SJF: Short Job First):** concede la CPU al proceso que durante menos tiempo necesita la CPU de forma ininterrumpida en el momento de hacer la elección, no aquel cuyo tiempo total de CPU es el menor de todos .



El óptimo. Consigue el menor tiempo de espera posible.

Si dos procesos necesitan el procesador durante el mismo tiempo, se elige uno mediante FCFS

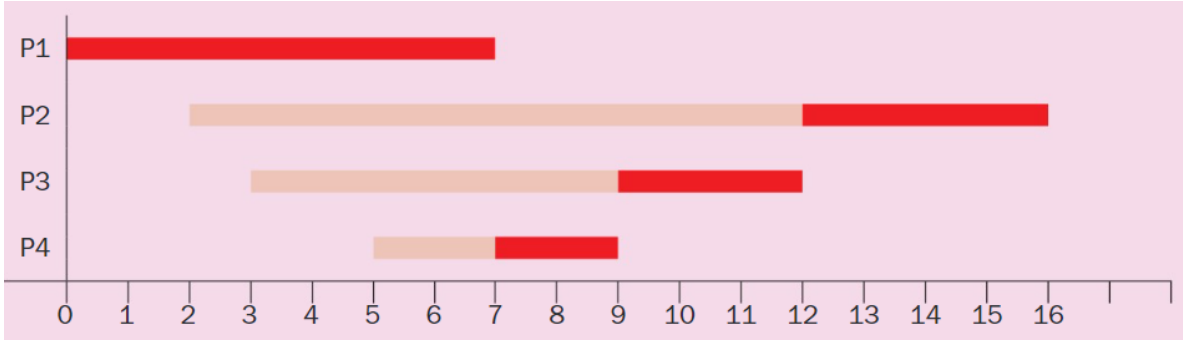
- **Por prioridades:** La prioridad de un proceso es un valor numérico que se usa como factor para determinar si debe entrar en CPU antes que otro(s).

Este algoritmo puede ser expulsor o no expulsor ya que se le puede ir cambiando la prioridad

2.1. Planificación no expulsora

El más corto primero (SJF: Short Job First):

PROCESO	TIEMPO DE LLEGADA	TIEMPO DE EJECUCIÓN
P1	0	7
P2	2	4
P3	3	3
P4	5	2

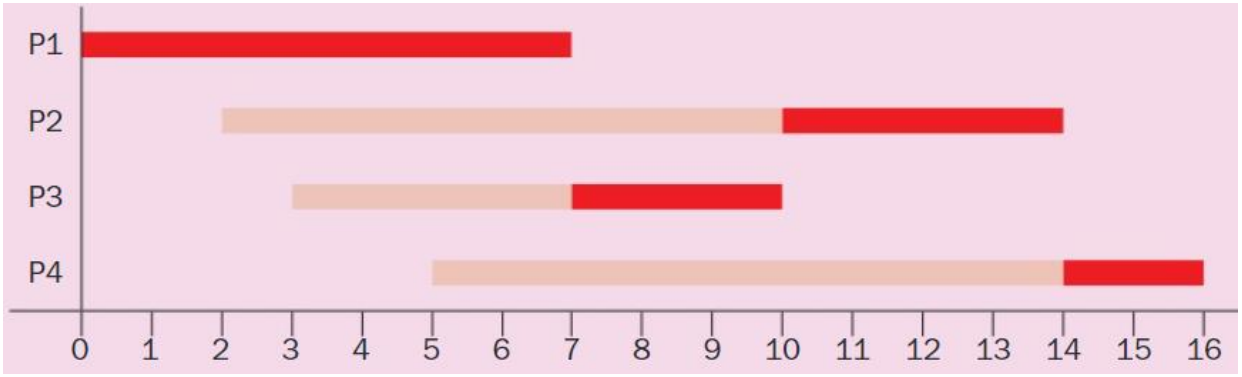


PROCESO	TIEMPO DE ESPERA	TIEMPO DE RESPUESTA
P1	0	7
P2	10	14
P3	6	9
P4	2	4
TIEMPOS MEDIOS	4,5	8,5

2.1. Planificación no expulsora

Por prioridades (no expulsivo)

PROCESO	TIEMPO DE LLEGADA	PRIORIDAD	TIEMPO DE EJECUCIÓN
P1	0	4	7
P2	2	2	4
P3	3	1	3
P4	5	3	2

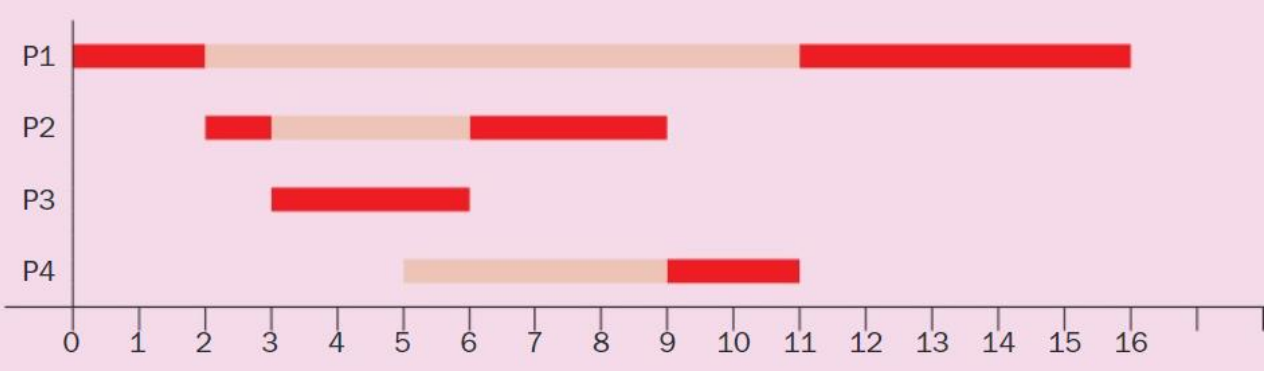


PROCESO	TIEMPO DE ESPERA	TIEMPO DE RESPUESTA
P1	0	7
P2	8	12
P3	4	7
P4	9	11
TIEMPOS MEDIOS	5,25	9,25

2.2. Planificación expulsora

Por prioridades (expulsivo)

PROCESO	TIEMPO DE LLEGADA	PRIORIDAD	TIEMPO DE EJECUCIÓN
P1	0	4	7
P2	2	2	4
P3	3	1	3
P4	5	3	2



PROCESO	TIEMPO DE ESPERA	TIEMPO DE RESPUESTA
P1	9	16
P2	3	7
P3	0	3
P4	4	6
TIEMPOS MEDIOS	4	8

2.2. Planificación expulsora

Algoritmos utilizados en las políticas expulsoras:

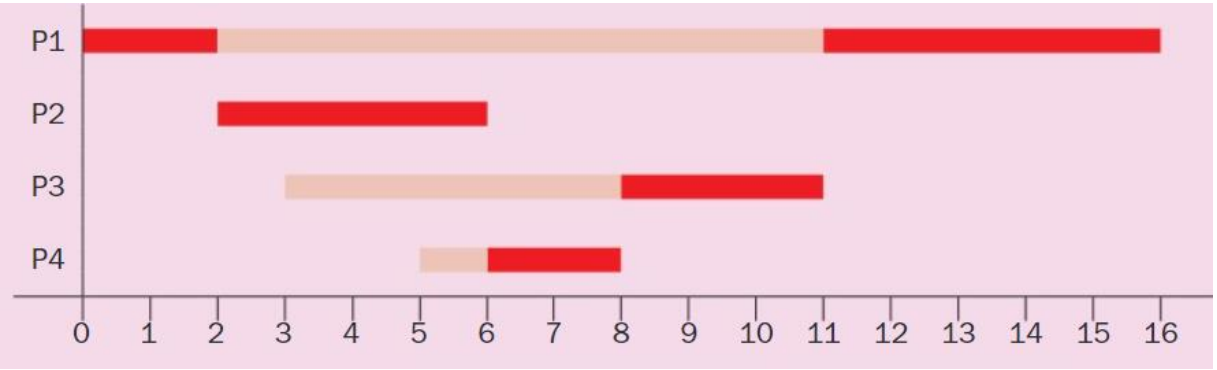
- **Algoritmo primero el tiempo restante más corto (SRTF: Short Remaining Time First):** este algoritmo va seleccionando de los procesos que están en espera al que le quede menor tiempo para terminar. En caso de empate se utiliza FIFO o FCFS.

Este algoritmo es expulsivo, ya que si mientras se está ejecutando un proceso llega otro que le quede menos tiempo para acabar que el que está utilizando la CPU en este momento lo desplaza, es decir, se produce un cambio de contexto

2.2. Planificación expulsora

SRTF: Short Remaining Time First

PROCESO	TIEMPO DE LLEGADA	TIEMPO DE EJECUCIÓN
P1	0	7
P2	2	4
P3	3	3
P4	5	2



PROCESO	TIEMPO DE ESPERA	TIEMPO DE RESPUESTA
P1	9	16
P2	0	4
P3	5	8
P4	1	3
TIEMPOS MEDIOS	3,75	7,75

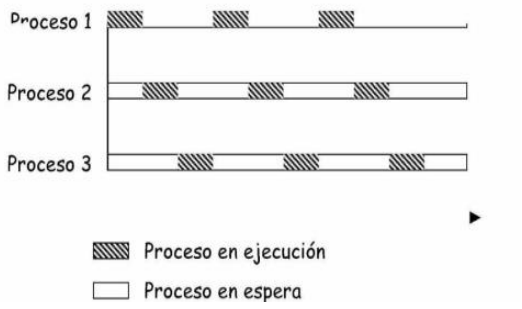
2.2. Planificación expulsora

Los algoritmos utilizados en las políticas de planificación expulsora:

- **Round – Robin (RR):** cuando el planificador tiene que seleccionar un proceso, elige el primero de la cola, establece una temporización correspondiente a la porción de tiempo, y le cede el control:

El proceso termina o realiza una operación de E/S antes de que se acabe su porción de tiempo, cede la CPU voluntariamente, pasa a espera y el planificador selecciona el siguiente proceso de la cola.

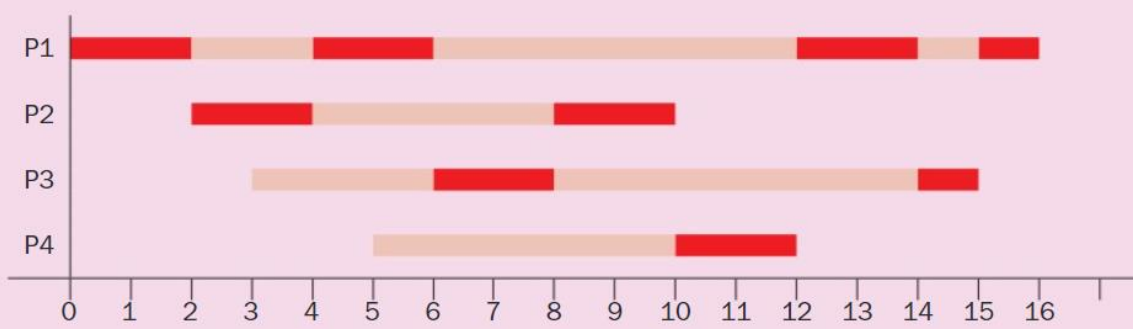
El proceso se ejecuta hasta agotar la temporización establecida, se le expropia la CPU al proceso en ejecución, que pasa al final de la cola y el planificador seleccione al primer proceso de la cola para pasar a ejecución.



2.2. Planificación expulsora

Round – Robin (RR)

PROCESO	TIEMPO DE LLEGADA	TIEMPO DE EJECUCIÓN
P1	0	7
P2	2	4
P3	3	3
P4	5	2



PROCESO	TIEMPO DE ESPERA	TIEMPO DE RESPUESTA
P1	9	16
P2	4	8
P3	9	12
P4	5	7
TIEMPOS MEDIOS	6,75	10,75

2.2. Planificación expulsora

Los algoritmos utilizados en las políticas de planificación expulsora:

- **Múltiples colas:** , la cola podría estar dividida en varias colas separadas, una por cada tipo de proceso. Cada proceso siempre se encola en su cola correspondiente.

- **Múltiples colas realimentadas:** Cuando un proceso lleva mucho tiempo en una cola de poca prioridad, se le pasa a otra cola de mayor prioridad, consiguiendo así mayores porciones de tiempo.

ACTIVIDAD – GESTIÓN DE PROCESOS CON POWERSHELL

Abrir Actividad en campus virtual – Gestión de procesos con Powershell

ÍNDICE

3. Clase ProcessBuilder

En java disponemos de varias clases para la gestión de procesos, una de ellas es la clase **ProcessBuilder**; cada una de sus instancias es capaz de gestionar una colección de atributos del proceso**.**

Cuando llamamos al método **start()** creamos una nueva instancia de **Process** con los atributos descritos, de esta forma si lo **invocamos varias veces** desde la misma instancia **crearemos varios subprocesos**.

ACTIVIDAD

Crear un proceso que nos abra NotePad en Java.

```java
package main;

import java.io.IOException;

public class Ejemplo1 {

    public static void main(String[] args) {

        ProcessBuilder pb = new ProcessBuilder("notepad.exe");
        try {
            pb.start();
        } catch (IOException e) {

            e.printStackTrace();
        }

    }

}
```

3. Clase ProcessBuilder

Métodos de la clase ProcessBuilder

Método	Descripción
start	Inicia un nuevo proceso usando los atributos especificados.
command	Permite obtener o asignar el programa y los argumentos de la instancia de ProcessBuilder.
directory	Permite obtener o asignar el directorio de trabajo del proceso.
environment	Proporciona información sobre el entorno de ejecución del proceso.
redirectError	Permite determinar el destino de la salida de errores.
redirectInput	Permite determinar el origen de la entrada estándar.
redirectOutput	Permite determinar el destino de la salida estándar.

3.1. Clase ProcessBuilder – método directory()

El siguiente código crea un objeto *ProcessBuilder* y determina el directorio de trabajo del proceso:

```
ProcessBuilder pBuilder = new
     ProcessBuilder("Notepad.exe","datos.txt");
pBuilder.directory(new File("c:/directorio_salida/"));
```

3.2. Clase ProcessBuilder – método environment()

Para acceder a la información del entorno de ejecución, el método *environment* devuelve un objeto Map con la información proporcionada por el SO. El siguiente ejemplo muestra por pantalla el número de procesadores disponibles en el sistema:

```
ProcessBuilder pBuilder = new
     ProcessBuilder("Notepad.exe","datos.txt");
java.util.Map<String, String> env = pBuilder.environment();
System.out.println("Número de procesadores:" +
     env.get("NUMBER_OF_PROCESSORS"));
```

```
USERDOMAIN_ROAMINGPROFILE=CLAUDIA
PROCESSOR_LEVEL=6
CHROME_RESTART=Google Chrome|¡Vaya! Se ha producido un fallo en Google Chrome. ¿Quieres reiniciar el navegador ahora?|LEFT_TO_RIGHT
RegionCode=EMEA
SESSIONNAME=Console
ALLUSERSPROFILE=C:\ProgramData
PROCESSOR_ARCHITECTURE=AMD64
__COMPAT_LAYER=DetectorsAppHealth
PSModulePath=C:\Program Files\WindowsPowerShell\Modules;C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
SystemDrive=C:
USERNAME=claud
ProgramFiles(x86)=C:\Program Files (x86)
CHROME_CRASHPAD_PIPE_NAME=\\.\pipe\crashpad_8200_RQYOOFATKNDZAKNH
FPS_BROWSER_USER_PROFILE_STRING=Default
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
DriverData=C:\Windows\System32\Drivers\DriverData
OneDriveConsumer=C:\Users\claud\OneDrive
platformcode=KV
ProgramData=C:\ProgramData
ProgramW6432=C:\Program Files
```

3.3. Clase ProcessBuilder – método command()

ACTIVIDAD

Demostrar que el método command() que devuelve los argumentos del objeto ProcessBuilder definido.

```
17    public static void main(String[] args) {
18
19        try {
20            // Crear un ProcessBuilder para un comando
21            ProcessBuilder builder = new ProcessBuilder("notepad.exe","datos.txt");
22
23            // Iniciar el proceso
24            Process process = builder.start();
25
26            List<String> list = builder.command();
27            Iterator<String> iterator = list.iterator();
28            while(iterator.hasNext()) {
29                System.out.println(iterator.next());
30            }
31
32        } catch (Exception e) {
33            e.printStackTrace();
34        }
35
36    }
```

Problems Javadoc Declaration Console ×

```
<terminated> Ejemplo1 [Java Application] C:\Users\claud\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32
notepad.exe
datos.txt
```

3.4. Clase Process

En la clase **Process** nos encontramos con una serie de métodos para realizar tareas:

Método	Descripción
destroy()	Destruye el proceso sobre el que se ejecuta.
exitValue()	Devuelve el valor de retorno del proceso cuando este finaliza. Sirve para controlar el estado de la ejecución.
getErrorStream()	Proporciona un InputStream conectado a la salida de error del proceso.
getInputStream()	Proporciona un InputStream conectado a la salida normal del proceso.
getOutputStream()	Proporciona un OutputStream conectado a la entrada normal del proceso.
isAlive()	Determina si el proceso está o no en ejecución.
waitFor()	Detiene la ejecución del programa que lanza el proceso a la espera de que este último termine.

3.5. Clase Process – método getInputStream()

Por defecto el proceso se crea (o subproceso) no tiene su propia terminal o consola. Todas las operaciones de E/S serán redirigidas al proceso padre. Podremos acceder a ellas a través de los métodos: **getOutputStream()**, **getInputStream()** y **getErrorStream()**

ACTIVIDAD

Ejecutar el comando dir de cmd para listar los archivos de directorio actual del proyecto y mostrar el resultado impreso por pantalla:

```
// Comando para listar archivos en el directorio
builder.command("cmd", "/c", "DIR", "C:\\Users\\claud\\Documents");
```

```
public class Ejemplo1 {

    public static void main(String[] args) {

        ProcessBuilder pb = new  ProcessBuilder("cmd", "/c", "DIR");

        try {
            Process p = pb.start();

            InputStream st = p.getInputStream();
             BufferedReader reader = new BufferedReader(new InputStreamReader(p.getInputStream()));
                String linea;
                while ((linea = reader.readLine()) != null) {
                    System.out.println(linea); // Imprimir la salida
                }

        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}
```

3.6. Clase Process – método waitFor()

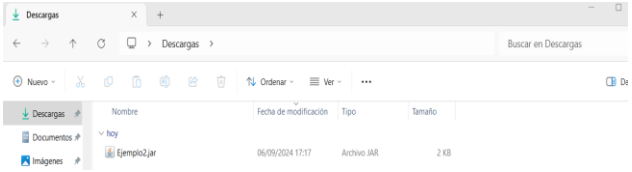
ACTIVIDAD

En el siguiente ejemplo imprimiremos el código de salida del proceso una vez que finalice

```java
public class Ejemplo1 {

    public static void LanzarProceso(String comando) {

    }

    public static void main(String[] args) {

        try {
            // Crear un ProcessBuilder para un comando
            ProcessBuilder builder = new ProcessBuilder("notepad.exe");

            // Iniciar el proceso
            Process process = builder.start();

            // Esperar a que el proceso termine
            int exitCode = process.waitFor();
            System.out.println("Proceso finalizado con código: " + exitCode);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

}
```

ACTIVIDAD

En el siguiente ejemplo llamaremos a otro programa desde nuestro proceso usando para ello el método directory().




```java
package main;

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

public class main {

    public static void main(String[] args) {
        File directorio = new File("C:\\Users\\claud\\Downloads");
        ProcessBuilder pb = new ProcessBuilder("CMD", "/c", "java -jar","Ejemplo2.jar");
        pb.directory(directorio);

        try {
            pb.start();
        } catch (IOException e) {

            e.printStackTrace();
        }

    }

}
```

```java
package main;

import java.io.IOException;

public class main {

    public static void main(String[] args) {
        ProcessBuilder pb = new ProcessBuilder("CMD", "/c", "java -jar",
                "C:\\Users\\claud\\Downloads\\Ejemplo2.jar");

        try {
            pb.start();
        } catch (IOException e) {

            e.printStackTrace();
        }

    }

}
```

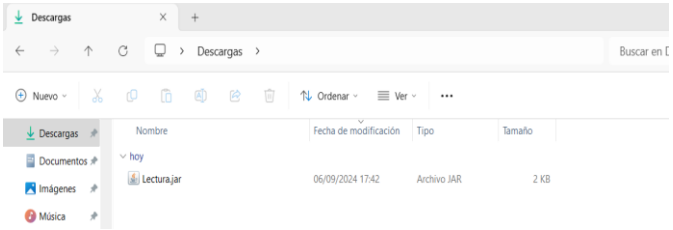
3.7. Clase Process – método getOutputStream()

ACTIVIDAD

En el siguiente ejemplo necesitaremos un pequeño programa que lea una cadena por teclado, nuestro programa utilizará ProcessBuilder para lanzar La lectura de cadena y con el método OutputStream sobrescribiremos la entrada del programa.

1. Crear el programa de lectura, crear el archivo .jar de ejecución.

```
package Lectura;

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

public class main {

    public static void main(String[] args) {

        try {
            InputStreamReader in = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(in);
            System.out.println("Introduce una cadena de texto: ");
            String texto = br.readLine();
            in.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}
```



3.7. Clase Process – método getOutputStream()

ACTIVIDAD

En el siguiente ejemplo necesitaremos un pequeño programa que lea una cadena por teclado, nuestro programa utilizará ProcessBuilder para lanzar el programa de lectura de cadena y con el método OutputStream sobrescribiremos la entrada del programa.

2. Creamos el programa de escritura

```
public class main {

    public static void main(String[] args) {

        try {
            File directorio = new File("C:\\Users\\claud\\Downloads");
            ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "java -jar",
                    "Lectura.jar");
            pb.directory(directorio);

            Process p = pb.start();

            //Sobreescribimos la entrada del proceso Lectura
            OutputStream out = p.getOutputStream();
            out.write("hola mundo".getBytes());
            out.flush();

            //Escribimos el resultado
            InputStream in = p.getInputStream();
            int texto;
            while(( texto = in.read())!=-1) {
                System.out.print((char)texto);
            }
            in.close();

        } catch (IOException e) {

            e.printStackTrace();
        }

    }

}
```

3.8. Clase Process – método getErrorStream()

ACTIVIDAD

En el siguiente ejemplo capturaremos los errores del proceso y los mostraremos por pantalla:

```
// Crear el ProcessBuilder para ejecutar un comando
ProcessBuilder builder = new ProcessBuilder();

// Comando para listar archivos en el directorio
builder.command("cmd", "/c", "java -jar", "C:\\Users\\claud\\Documents\\Ejemplo5.jar");

// Iniciar el proceso
Process process = builder.start();

// Capturar los errores
BufferedReader errorOutput = new BufferedReader(new InputStreamReader(process.getErrorStream())

String line;

// Mostrar los errores si los hay
System.out.println("Salida de errores:");
while ((line = errorOutput.readLine()) != null) {
    System.err.println(line);
}
```

3.9. Clase Runtime

Otra de las clases que disponemos en java para la gestión de procesos, una de ellas es la clase **Runtime.**

Los parámetros se reciben separados por espacios

```
Runtime.getRuntime().exec("Notepad.exe notas.txt");
```

```
String[] infoProceso = {"Notepad.exe","notas.txt"};
Runtime.getRuntime().exec(infoProceso);
```

3.10. Clase Runtime – método waitFor()

Para gestionar el proceso lanzado, primero es necesario obtener la referencia a la instancia de la clase Process proporcionada por el método exec.

```
String[] infoProceso = {"Notepad.exe","notas.txt"};
Process proceso = Runtime.getRuntime().exec(infoProceso);
```

Si se necesita esperar a que el proceso ejecutado termine y conocer el estado en que ha finalizado dicha ejecución, se puede utilizar el método waitFor. Este método suspende la ejecución del programa que ha arrancado el proceso hasta que este termine.

```
String[] infoProceso = {"Notepad.exe","notas.txt"};
Process proceso = Runtime.getRuntime().exec(infoProceso);
int codigoRetorno = proceso.waitFor();
System.out.println("Fin de la ejecución:" + codigoRetorno);
```

ACTIVIDAD

Ejecutar el comando dir de cmd para listar los archivos de directorio actual del proyecto y mostrar el resultado impreso por pantalla:

```
18    public static void main(String[] args) {
19
20        try {
21             // Comando a ejecutar (cambia el comando si usas Windows o Mac)
22            String comando = "cmd /c dir";
23            Process proceso = Runtime.getRuntime().exec(comando);
24
25            // Leer la salida del comando
26            BufferedReader reader = new BufferedReader(new InputStreamReader(proceso.getInputStream()));
27            String linea;
28            while ((linea = reader.readLine()) != null) {
29                System.out.println(linea);
30            }
31
32            // Esperar a que el proceso termine
33            proceso.waitFor();
34
35        } catch (Exception e) {
36            e.printStackTrace();
37        }
38
39    }
40
41 }
```

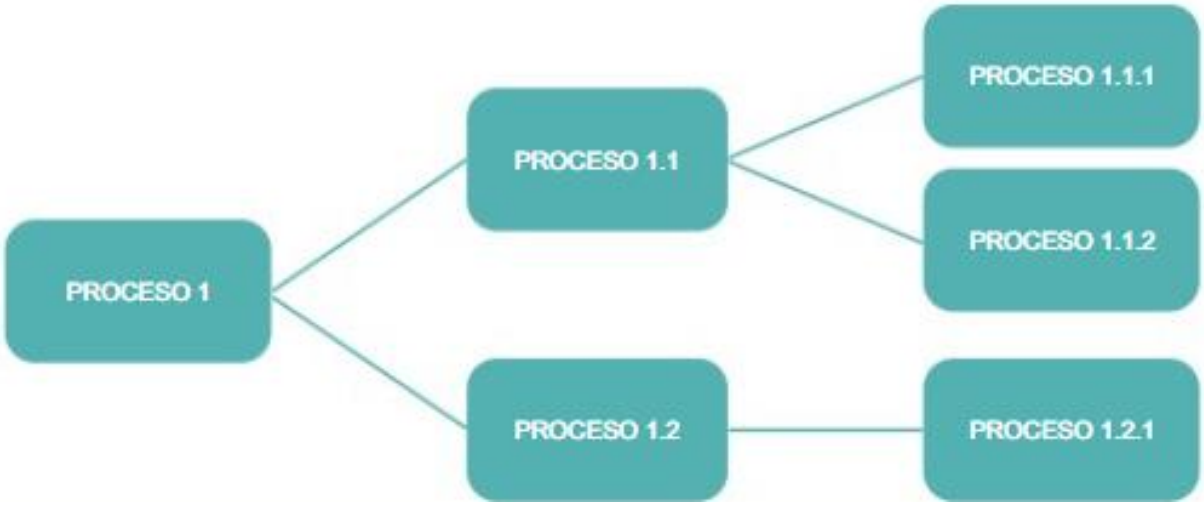
Problems Javadoc Declaration Console

<terminated> Ejemplo1 [Java Application] C:\Users\claud\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v20240802-1626

```
14/09/2024  14:33    <DIR>          .
06/09/2024  17:44    <DIR>          ..
06/09/2024  15:33               396 .classpath
06/09/2024  15:33               391 .project
06/09/2024  15:33    <DIR>          .settings
14/09/2024  14:33    <DIR>          bin
14/09/2024  14:33                 0 datos.txt
07/09/2024  13:02               433 output.txt
06/09/2024  15:34    <DIR>          src
               4 archivos          1.220 bytes
               5 dirs  419.758.743.552 bytes libres
```

3.11. Sincronización entre procesos

Todos los sistemas en los que participan múltiples actores de manera concurrente están sometidos a ciertas condiciones que exigen sincronización entre ellos.



3.11. Sincronización entre procesos

Para gestionar un flujo de trabajo con el del ejemplo anterior se necesita disponer de los siguientes mecanismos:

Ejecución. Un mecanismo para ejecutar procesos desde un proceso.

Espera. Un mecanismo para bloquear la ejecución de un proceso a la espera de que otro proceso termine.

Generación de código de terminación. Un mecanismo de comunicación que permita indicar a un proceso cómo ha terminado la ejecución mediante un código.

Obtención de código de terminación. Un mecanismo que permita a un proceso obtener el código de terminación de otro proceso.

Mecanismo	Clase	Método
Ejecución	Runtime	exec()
Ejecución	ProcessBuilder	start()
Espera	Process	waitFor()
Generación de código de terminación	System	exit(valor_del_retorno)
Obtención de código de terminación	Process	waitFor()

ACTIVIDAD

Crear dos procesos sincronizados en Java

1. **El proceso secundario devolverá el código de retorno 301 que significa que ha ocurrido un error de un tipo determinado.**

2. **Creamos el jar del proceso para ejecutarlo en el proceso principal**

```
public class main {

    public static void main(String[] args) {

        try {

            System.out.println("Proceso secundario");
            System.exit(103);

        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}
```

ACTIVIDAD

Crear dos procesos sincronizados en Java

3. Creamos el proceso principal que ejecutará el proceso anterior, quedará a la espera de que termine y evaluará el valor de finalización del proceso secundario para tomar una decisión

```
10    public static void main(String[] args) {
11
12    try {
13         // Comando a ejecutar (cambia el comando si usas Windows o Mac)
14        String comando = "cmd /c java -jar C:\\Users\\claud\\Downloads\\ProcesoSecundario.jar";
15        Process proceso;
16        proceso = Runtime.getRuntime().exec(comando);
17
18        // Esperar el valor de retorno del proceso secundario
19        int valorRetorno = proceso.waitFor();
20
21        if(valorRetorno ==0) {
22            System.out.println("El proceso termino correctamente");
23        }else {
24             System.out.println("El proceso termino con código de error " + valorRetorno);
25        }
26
27    } catch (IOException | InterruptedException e) {
28        // TODO Auto-generated catch block
29        e.printStackTrace();
30    }
31
32  }
33
34 }
```

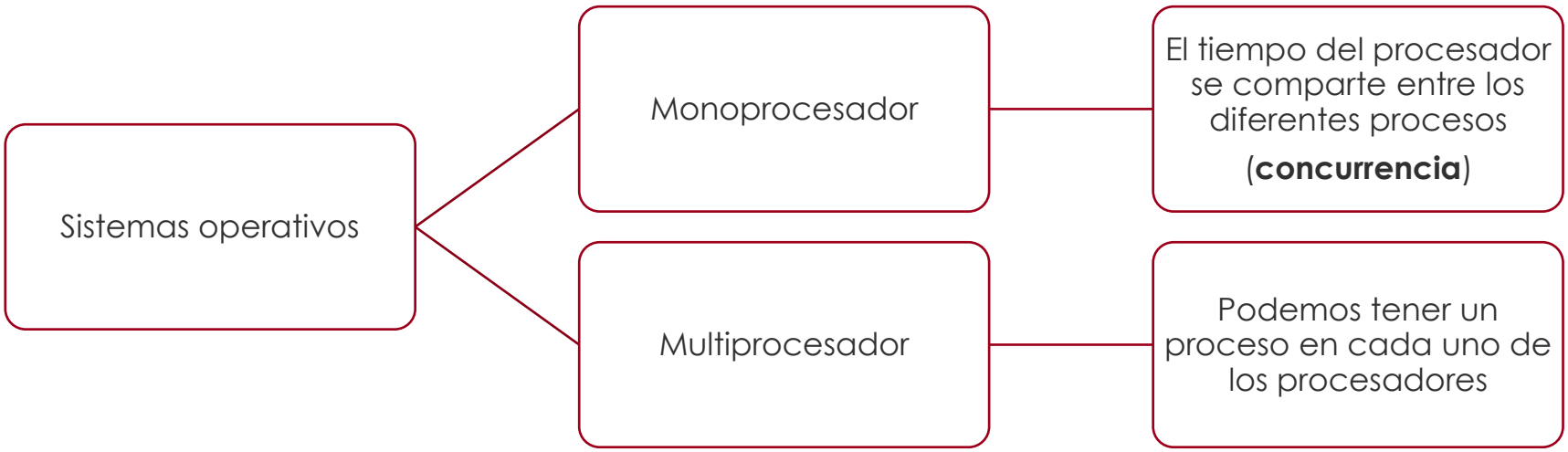
Problems Javadoc Declaration Console ×

<terminated> main (2) [Java Application] C:\Users\claud\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v20240802

El proceso termino con código de error 103

ÍNDICE

4. Programación concurrente



4. Programación concurrente

- La programación concurrente se refiere a la **existencia simultánea de varios procesos en ejecución en una misma unidad de proceso de manera alterna.**

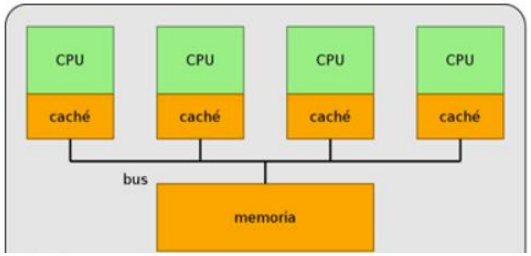
- Habrá procesos en los que compartan recursos y otros que compitan por ellos.

BENEFICIOS	DESCRIPCIÓN
Mejor aprovechamiento de la CPU	Un proceso puede aprovechas ciclos de la CPU mientras otro está realizando operaciones de I/O
Velocidad de ejecución	Al estar constituido un programa por pequeños procesos, esto se pueden "repartir" entre los diferentes procesadores o en uno únicamente en función de su importancia

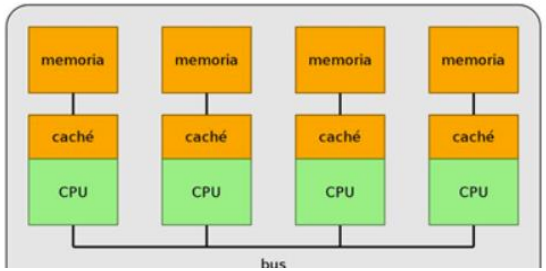
4.1. Sistema multiprocesador

Los sistemas multiprocesadores los podemos clasificarlos en dos tipos:

- **Fuertemente acoplados:** existe una memoria compartida para todos los procesadores.



- **Débilmente acoplados:** los procesadores poseen su propia memoria local e independiente, de manera que no comparten memoria.



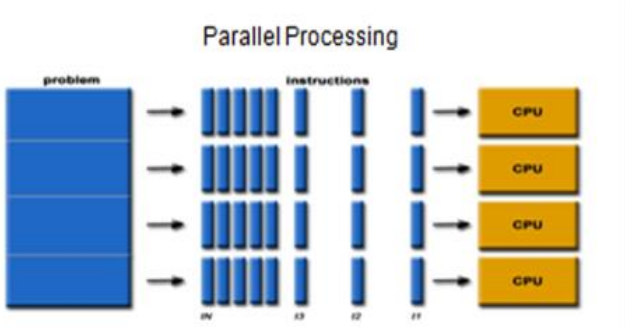
4.1. Multiproceso

Denominaremos por lo tanto **multiproceso** a la gestión de varios procesos dentro de un sistema multiprocesador, donde capa procesador puede acceder a una memoria común.

ÍNDICE

5. Programación paralela

- Los programas paralelos se ejecutan sólo en los sistemas multiprocesadores.
- Permite ejecutar un proceso en cada procesador (como mínimo) y que todos trabajen juntos para resolver un problema. Normalmente esto se consigue dividiendo el problema en pequeñas partes e intercambiando información entre ellas.

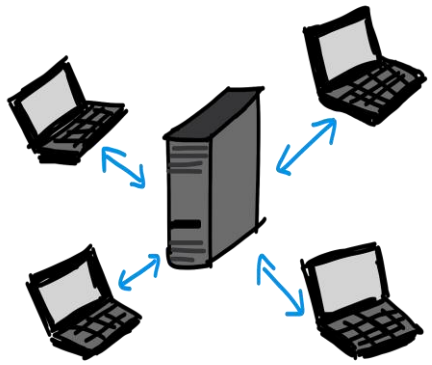


MODELO	EXPLICACIÓN
Memoria Compartida ⇓ Sistemas de Memoria Compartida	Los procesadores comparten físicamente la memoria, es decir, todos acceden al mismo espacio de direcciones. Un valor escrito en memoria por un procesador puede ser leído directamente por cualquier otros
Paso de mensajes ⇓ Sistemas de memoria distribuida	Cada procesador dispone de su propia memoria independiente del resto y accesible solo para él. Para realizar el intercambio de información es necesario que cada procesador realice la petición de datos al procesador que los tiene, y éste haga el envío.

ÍNDICE

6. Programación distribuida

- Se define un **sistema distribuido** como aquel en el que la ejecución de software se distribuye entre varios ordenadores



CARACTERÍSTICAS	EXPLICACIÓN
Concurrencia	Lo normal es 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 arquitectura por excelencia es la de Cliente/Servidor.