# UNIDAD 1

# PROGRAMACIÓN MULTIPROCESO

## PROGRAMA

Toda la información (tanto código como datos) almacenada en disco de una aplicación que resuelve una necesidad concreta para los usuarios.

## PROCESO

Programa en ejecución. Este concepto no se refiere únicamente al código y a los datos, sino que incluye todo lo necesario para su ejecución:

- Contador de programa: registro interno en donde se almacena la dirección de la última instrucción leída. Con ello, el procesador puede identificar la siguiente instrucción que debe ejecutar.
- Imagen de memoria: memoria que el proceso está usando
- Estado del procesador: valor de los registros del procesador sobre los cuales se está ejecutando.

Los procesos son entidades independientes, aunque ejecuten el mismo programa. De tal forma, pueden coexistir dos procesos que ejecuten el mismo programa, pero con diferentes datos (con distintas imágenes de memoria) y en distintos momentos de su ejecución (con diferentes contadores de programa).

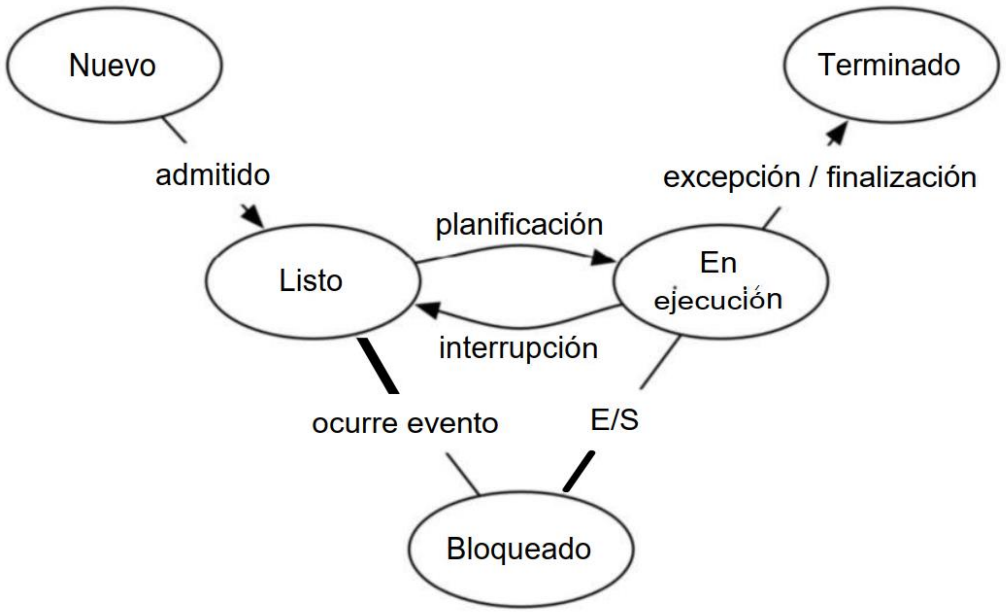
## PID

Process Identifier.
Unívoco para cada proceso.
Su utilización es básica a la hora de gestionar procesos.

## ESTADO DE UN PROCESO

Los procesos pueden cambiar de estado a lo largo de su ejecución.

- Nuevo: el proceso está siendo creado a partir del fichero ejecutable.
- Listo: el proceso no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse.
- En ejecución: el proceso se está ejecutando. El sistema operativo utiliza el mecanismo de interrupciones para controlar su ejecución.
- Bloqueado: el proceso está bloqueado esperando que ocurra algún suceso. Cuando ocurre el evento que lo desbloquea, el proceso no pasa directamente a ejecución, sino que tiene que ser planificado de nuevo por el sistema.
- Terminado: el proceso ha finalizado su ejecución y libera su imagen de memoria.



## EJECUTABLE

Fichero que contiene la información necesaria para crear un proceso a partir de los datos almacenados de un programa. Fichero que permite poner el programa en ejecución como proceso.

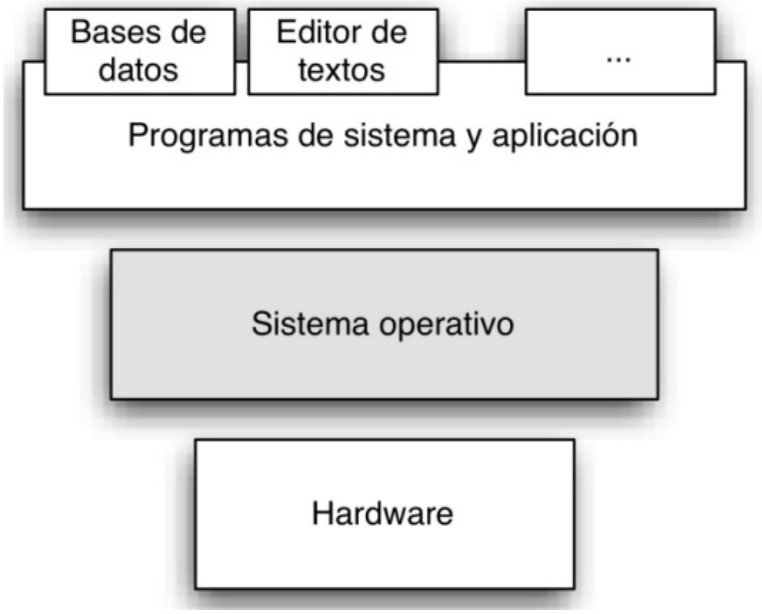
## DEMONIO

Proceso no interactivo que está ejecutándose continuamente en segundo plano. Suele proporcionar un servicio básico para el resto de procesos.

## SISTEMA OPERATIVO

Programa que hace de intermediario entre el usuario y las aplicaciones que utiliza y el hardware del ordenador. Entre sus funcionalidades:

- Hacer de interfaz entre usuario y los recursos del ordenador.
- Permite utilizar los recursos del computador de forma eficiente.
- Ejecuta los programas de usuario.



## FUNCIONAMIENTO BÁSICO DE UN SISTEMA OPERATIVO

### KERNEL

Parte central del sistema operativo responsable de gestionar los recursos del ordenador, permitiendo su uso a través de llamadas al sistema.

Funciona en base a interrupciones. Una interrupción es una suspensión temporal de la ejecución de un proceso, para pasar a ejecutar una rutina que trate dicha interrupción.

- Cuando salta una interrupción se transfiere el control a la rutina de tratamiento de la interrupción.

- Mientras se está atendiendo una interrupción, se deshabilita la llegada de nuevas interrupciones.

- Cuando finaliza la rutina, se reanuda la ejecución del proceso en el mismo lugar donde se quedó cuando fue interrumpido.

## COLAS DE PROCESOS

Los procesos se van intercambiando el uso del procesador para su ejecución de forma concurrente. Para ello, el sistema operativo organiza los procesos en varias colas, migrando los procesos de unas a otras:

- Cola de procesos: contiene todos los procesos del sistema.

- Cola de procesos preparados: contiene todos los procesos listos que esperan ser ejecutados

- Cola de dispositivo: contiene los procesos que están a la espera de alguna operación de E/S.

## PLANIFICACIÓN DE PROCESOS

El planificador es el encargado de seleccionar los movimientos de procesos entre las diferentes colas.

Existen dos tipos de planificación:

- A corto plazo:
    - Selecciona que proceso de la cola de procesos preparados pasará a ejecución.
    - Se invoca del orden de milisegundos. Cuando se produce un cambio de estado del proceso en ejecución
    - Decisiones rápidas. Algoritmos de planificación sencillos
- A largo plazo:
    - Selecciona que procesos nuevos deben pasar a la cola de procesos preparados.
    - Se invoca con poca frecuencia. Puede tardar más tiempo en la toma de decisión.
    - Controla el grado de multiprogramación (número de procesos en memoria)

## PLANIFICACIÓN DE PROCESOS A CORTO PLAZO

- Planificación cooperativa:
    - Únicamente se cambia el proceso en ejecución si dicho proceso se bloquea o termina.

- Planificación apropiativa:
    - Se cambia el proceso en ejecución si en cualquier momento en que un proceso se está ejecutando, otro proceso con mayor prioridad se puede ejecutar.
    - La aparición de un proceso más prioritario se puede deber tanto al desbloqueo del mismo como a la creación de un nuevo proceso.

- Tiempo compartido:

    - Cada cierto tiempo se desaloja el proceso que estaba en ejecución y se selecciona otro proceso para ejecutarse.

    - Todas las prioridades de los procesos se consideran iguales.

## GESTIÓN DE PROCESOS

- Creación de procesos

    - Cuando se crea un nuevo proceso hijo, ambos hijos, padre e hijo se ejecutan concurrentemente.

    - Ambos procesos comparten CPU y se irán intercambiando siguiendo la política de planificación del sistema operativo.

    - Si el proceso padre necesita esperar hasta que el hijo termine su ejecución, puede hacerlo mediante la operación *wait.*

    - Los procesos son independientes y tienen su propio espacio de memoria asignado, llamado imagen de memoria.

- Terminación de procesos

    - Al terminar la ejecución de un proceso, es necesario avisar al sistema operativo de su terminación para liberar los recursos que tenga asignados.

    - En general, es el proceso el que le indica al sistema mediante la operación *exit* que quiere terminar, pudiendo aprovechar para mandar información de su finalización al padre

## CREACIÓN DE PROCESOS

La clase que representa un proceso en Java en la clase *Process*.

- Los métodos de *ProcessBuilder.start()* y *Runtime.exec()* crean un proceso nativo en el sistema operativo y devuelven una clase *Process,* la cual usaremos para controlar dicho proceso.

```
import java.io.IOException;
import java.util.Arrays;

public class RunProcess {

        public static void main(String[] args) throws IOException {

         if (args.length <= 0) {
               System.err.println("Se necesita un programa a ejecutar");
               System.exit(-1);
         }

         ProcessBuilder pb = new ProcessBuilder(args);
         try {
                                          Process process = pb.start();
                                          int retorno = process.waitFor();
                                          System.out.println("La ejecución de " +
                  Arrays.toString(args) + " devuelve " +  retorno);
          }catch(IOException ex){
                        System.err.println("Excepción de E/S!!");
                                System.exit(-1);
                        }catch(InterruptedException ex){
                                          System.err.println("El proceso hijo finalizó
                     de forma incorrecta");
                                          System.exit(-1);
                        }
   }
```

## TERMINACIÓN DE PROCESOS

- El proceso hijo realizará su ejecución completa terminando y liberarndo sus recursos al finalizar.

- Esto se producto cuando el hijo realiza la operación *exit* para finalizar la ejecución.

- Un proceso padre puede terminar de formar abrupta un proceso hijo creado por él mediante la operación *destoy*.

```
import java.io.IOException;

public class RuntimeProcess {

   public static void main(String[] args) {

 if (args.length <= 0) {
        System.err.println("Se necesita un programa a ejecutar");
        System.exit(-1);
 }

                Runtime runtime = Runtime.getRuntime();
                try {
                        Process process = runtime.exec(args);
                        process.destroy();
                }catch(IOException ex){
                        System.err.println("Excepción de E/S!!");
                        System.exit(-1);
                }
   }
```

## COMUNICACIÓN DE PROCESOS

En Java, el proceso hijo no tiene su propia interfaz de comunicación, por lo que el usuario no puede comunicarse con él directamente.

- *OutputStream:* Flujo de salida del proceso hijo. El stream está conectado por un pipe a la entrada estándar del proceso hijo

- *InputStream:* Flujo de entrada del proceso hijo. El stream está conectado por un pipe a la salida estándar del proceso hijo.

- *ErrorStream:* Flujo de errores del proceso hijo. El stream está conectado por un pipe a la salida estándar del proceso hijo

Consideramos un pipe como un canal de comunicación sencillo entre padre e hijo. Se podría asociar como si fuera una especia de falso fichero que sirve para conectar dos procesos con sus respectivas funciones *write()* y *read().*

## SINCRONIZACIÓN DE PROCESOS

- Los métodos de comunicación de procesos se pueden considerar como métodos de sincronización ya que permiten al proceso padre llevar el ritmo de envío y recepción de mensajes.

- Además de la utilización de los flujos de datos, se puede esperar por la finalización del proceso hijo mediante la operación *wait.*

    - Bloquea al proceso padre hasta que el hijo finaliza su ejecución.

    - Como resultado, el padre recibe la información de finalización del proceso hijo.

    - El valor de retorno especifica mediante un número entero el resultado de la ejecución. Se utiliza el 0 para indicar que el hijo ha acabado de forma correcta.

```
import java.io.IOException;
import java.util.Arrays;

public class ProcessSincronization {

        public static void main(String[] args)
          throws IOException, InterruptedException {

        try{
            Process process = new ProcessBuilder(args).start();
            int retorno = process.waitFor();

           System.out.println("Comando " +  Arrays.toString(args) +  " devolvió: " + retorno);
        } catch (IOException e) {
                System.out.println("Error ocurrió ejecutando el comando:" + e.getMessage());
        } catch (InterruptedException e) {
                System.out.println("El comando fue interrumpido. Descripción del error: " +
                e.getMessage());
        }
    }
}
```

Si se pretende realizar procesos que cooperen entre sí, debe ser el propio desarrollador quien lo implemente utilizando comunicación y sincronización de procesos.

# CLASE PROCESS

| MÉTODOS | MISIÓN |
|---|---|
| **InputStream getInputStream ()** | Devuelve el flujo de entrada conectado a la salida normal del subproceso. Nos permite leer el stream de salida del subproceso, es decir, podemos leer lo que el comando que ejecutamos escribió en la consola. |
| **int waitFor ()** | Provoca que el proceso actual espere hasta que el subproceso representado por el objeto **Process** finalice. Devuelve 0 si ha finalizado correctamente. |
| **InputStream getErrorStream()** | Devuelve el flujo de entrada conectado a la salida de error del subproceso. Nos va a permitir poder leer los posibles errores que se produzcan al lanzar el subproceso. |
| **OutputStream getOutputStream()** | Devuelve el flujo de salida conectado a la entrada normal del subproceso. Nos va a permitir escribir en el stream de entrada del subproceso, así podemos enviar datos al subproceso que se ejecute. |
| **void destroy ()** | Elimina el subproceso. |
| **int exitValue ()** | Devuelve el valor de salida del subproceso. |
| **boolean isAlive ()** | Comprueba si el subproceso representado por **Process** está vivo |

# CLASE PROCESSBUILDER

| MÉTODOS | MISIÓN |
|---|---|
| **ProcessBuilder command ( String argumentos ...)** | Define el programa que se quiere ejecutar indicando sus argumentos como una lista de cadenas separadas por comas. |
| **List < String > command ()** | Devuelve todos los argumentos del objeto **ProcessBuilder**. |
| **Map < String , String > environment ()** | Devuelve en una estructura Map las variables de entorno del objeto **ProcessBuilder**. |
| **ProcessBuilder redirectError (File file)** | Redirige la salida de error estándar a un fichero. |
| **ProcessBuilder redirectInput (File file)** | Establece la fuente de entrada estándar en un fichero. |
| **ProcessBuilder redirectOutput ( File file)** | Redirige la salida estándar a un fichero. |
| **File directory()** | Devuelve el directorio de trabajo del objeto **ProcessBuilder**. |
| **ProcessBuilder directory(File directorio)** | Establece el directorio de trabajo del objeto **ProcessBuilder**. |
| **Process start ()** | Inicia un nuevo proceso utilizando los atributos del objeto **ProcessBuilder**. |