

# UT1: Programación Multiproceso

## Índice de Contenidos

UT1: Programación Multiproceso.....	1
1. Procesos: conceptos teóricos.....	2
1.1 Estados de un Proceso.....	2
1.2 Comunicación entre procesos (IPC Inter-Process Communication).....	3
1.2.1 E3:Comunicación de procesos Java y Python con lectura del flujo de salida.....	4
1.3 Sincronización entre procesos.....	6
1.3.1 E4:Sincronización de procesos Java mediante códigos de finalización.....	6
2. Programación de Aplicaciones Multiproceso.....	8
2.1 Creación de Procesos con ProcessBuilder.....	9
2.2 Creación de Procesos con Runtime.....	10
2.3 E5: Comparativa entre Runtime y ProcessBuilder. Sincronización con códigos de finalización y flujo de salida.....	11

# 1. Procesos: conceptos teóricos

**Un proceso es un programa en ejecución.**

Cada proceso está compuesto por:

- Las instrucciones que se van a ejecutar.
- El estado del propio proceso.
- El momento de la ejecución, recogido en los registros del procesador.
- El estado de la memoria.

El **contexto** es toda la información que determina el estado de un proceso en un instante determinado.

Los procesos están continuamente entrando y saliendo del procesador, esto se gestiona por el SO. Cuando se saca un proceso para meter otro, se realiza un **cambio de contexto**.

Pasos a seguir para realizar un cambio de contexto:

- 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.

El **planificador de procesos** es el elemento del SO que se encarga de gestionar los recursos que demandan los procesos. El objetivo principal es conseguir que todos los procesos terminen lo antes posible, aprovechando al máximo los recursos del sistema, atendiendo a los siguientes parámetros:

- Maximizar el rendimiento del sistema: que los recursos se aprovechen al máximo.
- Maximizar la equidad en el reparto de los recursos: todos los procesos tienen acceso a los recursos que necesitan, de forma equitativa **o estableciendo prioridades (por necesidades del sistema)**.
- Minimizar los tiempos de espera: todos los procesos conseguirán los recursos en algún momento. Ningún proceso se adueñará de los recursos. Ningún proceso estará eternamente esperando.
- Minimizar los tiempos de respuesta: que los procesos terminen lo antes posible.

## 1.1 Estados de un Proceso

El **estado del proceso**, es la situación en la que se encuentra el propio proceso. El planificador de procesos gestiona dichos estados, provocando su cambio.

**Nuevo:** Se crea el proceso, pasa inmediatamente a Listo.

**Listo:** El proceso está en memoria, listo para ejecutarse. Está a la espera de que el planificador le conceda tiempo de ejecución.

**En Ejecución:** El proceso se está ejecutando.

**Bloqueado:** Se encuentra a la espera de que ocurra un evento externo, ajeno al planificador. Por ejemplo, que termine otro proceso o se solicite un recurso que no está libre.

**Finalizado:** El proceso ha completado su ejecución.



Entre los distintos estados se producen transiciones.

En una ejecución normal, sin dependencias, el cambio de contexto se produce en la transición “Interrumpido”.

Investiga:

T1) ¿Qué es un lag? A la vista de los contenidos de esta unidad. ¿Cuándo crees que se produce?

El retardo es el tiempo de espera durante el cual un proceso se encuentra en estado de Listo o Bloqueado, a la espera de conseguir el recurso necesario (recurso necesario o tiempo de proceso).

Puede lagearse, en estado de Listo, porque otro proceso prioritario se ha apropiado del tiempo de proceso.

Puede lagearse, en estado de Bloqueado, cuando no consigue los recursos que necesita para su ejecución.

Los cambios de contexto, donde no se optimiza los tiempos de ejecución aumentan el lag.

## 1.2 Comunicación entre procesos (IPC Inter-Process Communication)

Por definición los procesos son independientes, pero puede surgir la necesidad de comunicación entre ellos, es decir, pueden surgir dependencias en lo referente a las entradas y salidas de datos.

Esta comunicación puede producirse de distintas formas:

- **Utilización de sockets:** se crean canales de comunicación bidireccionales a nivel de bytes. Se realiza entre procesos alojados en distintas máquinas y programados con diferentes lenguajes. (Ideal para procesamiento distribuido)
- **Utilización de flujos de entrada y salida:** los procesos pueden interceptar los flujos de entrada y salida estándar, por lo que pueden leer y escribir datos unos en otros. Para que esto sea posible los procesos deben estar relacionados, por ejemplo, uno debe haber iniciado a otro, de forma que tenga la referencia para acceder al mismo. (Veremos un ejemplo en el apartado 1.2.1) (Ideal para procesos relacionados ejecutándose en una misma máquina.)
- **RPC:** Llamada a procedimiento remoto. Se realizan llamadas a métodos de otros procesos en ejecución en otras máquinas. Esta llamada se realiza de forma transparente utilizando la tecnología RMI.
- **Utilizando medios persistentes:** consiste en realizar escrituras y lecturas en cualquier sistema de almacenamiento (ficheros o bd). La comunicación se realiza a través del acceso a estos recursos compartidos.
- **Utilizando servicios de Internet:** igual que en el caso anterior pero utilizando servicios ftp, servidores web, aplicaciones o cualquier tecnología cloud.

En este módulo nos centraremos en los tres primeros sistemas de comunicación.

### 1.2.1 E3: Comunicación de procesos Java y Python con lectura del flujo de salida

Tendremos dos programas. El primero es Pspud1e3.java que lanza un proceso Python, captura la salida, espera a que termine y muestra dicha salida. Asegúrate de poner la ruta correcta, tanto para el intérprete Python como para el propio programa escrito en este lenguaje. Posibles valores de salida:

0: Todo OK

1: Error de sintaxis en el programa pasada por parámetro al intérprete Python

2 o 9009: Error en la ruta del interprete Python o el fichero de entrada

#### pspud1e3.java

```
package pspud1e3.ProgramaciónMultiproceso;

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

public class Pspud1e3 {

    public static void main (String[] args) {
        try {
            Process proceso = new ProcessBuilder(
                "C:\\Users\\Lopema\\AppData\\Local\\Programs\\Python\\Python310\\python.exe",
                "D:\\PSP\\pspud1e3\\src\\pspud1e3\\proceso_python.py").start();

            BufferedReader br = new BufferedReader(
                new InputStreamReader(proceso.getInputStream()));

            proceso.waitFor();

            int exitStatus = proceso.exitValue();
            System.out.println("Retorno:" + br.readLine());
            System.out.println ("Valor de la salida:" + exitStatus);

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

Por otro lado tendremos un programa en Python, que crea una estructura y la muestra por la salida standard.

### proceso\_python.py

```
import json
import sys

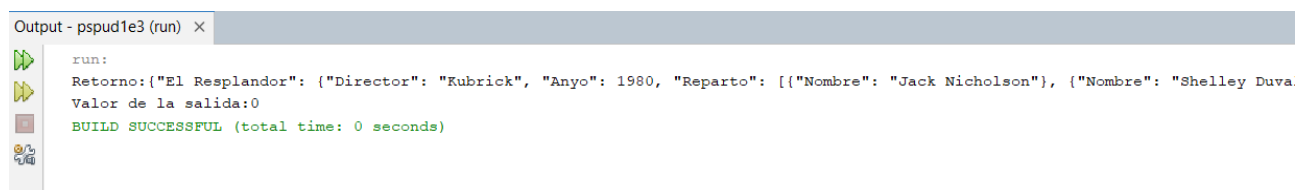
pelicula = { "El Resplandor":
    { "Director": "Kubrick",
      "Anyo": 1980,
      "Reparto": [
        { "Nombre": "Jack Nicholson"},
        { "Nombre": "Shelley Duvall"},
        { "Nombre": "Danny Lloyd"},
        { "Nombre": "Scatman Crothers"}
      ]
    }
  }

print(json.dumps(pelicula))
sys.exit(0)
```

Si ejecutamos el programa Python directamente en la consola cmd, simplemente mostrará la salida:

```
C:\Users\Lopema>C:\\Users\\Lopema\\AppData\\Local\\Programs\\Python\\Python310\\python.exe D:\\PSP\\pspud1e3\\src\\pspud1e3\\proceso_python.py
{"El Resplandor": {"Director": "Kubrick", "Anyo": 1980, "Reparto": [{"Nombre": "Jack Nicholson"}, {"Nombre": "Shelley Duvall"}, {"Nombre": "Danny Lloyd"}, {"Nombre": "Scatman Crothers"}]}}
```

Si ejecutamos el programa en Java, a través de NetBeans, tendremos la salida:



```
Output - pspud1e3 (run) x
run:
Retorno:{"El Resplandor": {"Director": "Kubrick", "Anyo": 1980, "Reparto": [{"Nombre": "Jack Nicholson"}, {"Nombre": "Shelley Duva.
Valor de la salida:0
BUILD SUCCESSFUL (total time: 0 seconds)
```

## 1.3 Sincronización entre procesos

Para posibilitar la sincronización de procesos es necesario que dichos procesos se comuniquen entre si. Una forma de hacerlo es utilizando los códigos de finalización. Estos códigos se deben describir suficientemente en el Manual de Explotación de cualquier sistema software.

El planificador de procesos es quien decide en qué momento tendrá un proceso acceso a los recursos que necesita. Pero es el programador quien decide el funcionamiento de dichos procesos y como se comportarán ante determinadas situaciones: que procesos se lanzarán, cuando y que consecuencias se derivan.

Para que esto sea posible son necesarios algunos mecanismos:

Mecanismo	Clase	Método
<b>Ejecución:</b> Un mecanismo para ejecutar procesos desde un proceso.	Runtime	exec()
	ProcessBuilder	start()
<b>Espera:</b> Un mecanismo para bloquear la ejecución de un proceso a la espera de que otro proceso termine.	Process	waitFor()
<b>Generación de código de terminación:</b> Un mecanismo que permite a un proceso indicar cómo finalizó su ejecución, mediante un código	System	exit(N)
<b>Obtención de código de terminación:</b> Un mecanismo que permite a un proceso obtener el código de terminación de otro proceso. (Capturamos el código en una variable)	Process	waitFor() exitValue()

### 1.3.1 E4:Sincronización de procesos Java mediante códigos de finalización

Primero definimos un proceso secundario, que va a generar un código de finalización aleatorio entre 0 y 9999

#### ProcesoSecundario.java

```
package pspud1e4.SincronizacionProcesos;

public class ProcesoSecundario {

    public static void main(String[] args) {

        System.out.println("Ejecutando proceso secundario...");
        var codigo = (int) Math.floor(Math.random()*101);
        System.out.println("Terminado proceso secundario..." + codigo);
        System.exit(codigo);

    }
}
```

## ProcesoPrincipal.java

```
package pspud1e4.SincronizacionProcesos;

public class ProcesoPrincipal {

    public static void main(String[] args) {

        try {
            String[] infoProceso = {"Java","D:\\PSP\\PspUd1E4\\src\\pspud1e4\\ProcesoSecundario.java"};
            //String[] infoProceso = {"Java","pspud1e4.ProcesoSecundario"};

            Process proceso = Runtime.getRuntime().exec(infoProceso);

            int valorRetorno = proceso.waitFor();

            if (valorRetorno<50) {
                System.out.println("El proceso secundario terminó correctamente. Código: "+valorRetorno);
            } else {
                System.out.println("El proceso secundario generó el Error: "+valorRetorno);
            }

        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

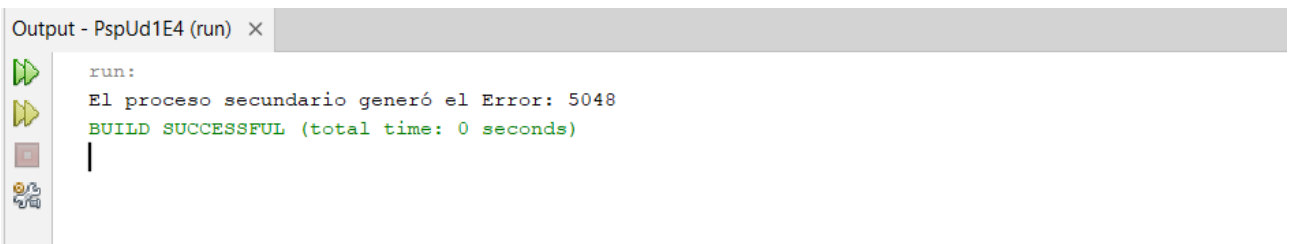
Error inesperado, ¿por qué no puede invocar un objeto ejecutable desde otro objeto ejecutable del mismo paquete? ¿Es problema del NetBeans o del Path de Java?

Si invocamos utilizando rutas absolutas funciona correctamente.

Si ejecutamos el proceso secundario obtendremos la salida:



Si ejecutamos el proceso principal, que invoca al proceso secundario y captura el error, obtendremos la salida:



No es el mismo código de finalización por la propia naturaleza aleatoria del ejercicio.

## 2. Programación de Aplicaciones Multiproceso

Cada instancia de una aplicación en ejecución es un proceso. La programación de aplicaciones multiproceso consiste en coordinar la ejecución de un conjunto de aplicaciones para lograr un conjunto común.

Como hemos visto en el punto anterior los mecanismos necesarios para realizar una programación multiproceso son:

- Poder arrancar un proceso y hacerle llegar los parámetros de ejecución.
- Poder quedar a la espera de que el proceso termine.
- Poder recoger el código de finalización de ejecución para determinar si el proceso se ha ejecutado correctamente o no.
- Poder leer los datos generados por el proceso para su tratamiento.

En Java la creación de un proceso se puede hacer de dos maneras:

- Utilizando la clase **java.lang.Runtime**
- Utilizando la clase **java.lang.ProcessBuilder**

Investiga:

T2) ¿Qué diferencias existen entre Runtime y ProcessBuilder?

T3) Diferencia entre programación paralela y distribuida.



## 2.1 Creación de Procesos con ProcessBuilder

La clase ProcessBuilder, al igual que Runtime, permite crear procesos. Su funcionamiento sería el siguiente:

```
// Preparamos el proceso  
String[] infoProceso = {"Notepad.exe", "notas.txt"};  
ProcessBuilder pBuilder = new ProcessBuilder(infoProceso);  
// Lo lanzamos y esperamos a que termine  
Process proceso = pBuilder.start();  
int valorRetorno = proceso.waitFor();  
//Mostramos el valor de retorno  
System.out.println("Valor de Retorno: " + valorRetorno);
```

Métodos de la clase **java.lang.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.

Bibliografía:

Clase ProcessBuilder: <https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html>

## 2.2 Creación de Procesos con Runtime

Veamos el funcionamiento de la **clase java.lang.Runtime**

Toda aplicación java tiene una única instancia de la clase Runtime que le permite interactuar con su entorno de ejecución, a través del método estático `getRuntime`.

La instrucción: **`Runtime rt = Runtime.getRuntime();`**

**Nos da acceso al entorno de ejecución del propio proceso.** Construye el objeto entorno desde el que podemos pedir al SO que lance otro proceso. Utilizamos el método `exec`.

Con las instrucciones: **`String[] infoproceso = {"Notepad.exe", "notas.txt"};`**

**`Process proceso = rt.exec(infoproceso);`**

Creamos un vector de cadenas con el programa y sus parámetros, que pasamos como entrada al método `exec` para que lo ejecute en el entorno capturado anteriormente. Para tener acceso al proceso creado, lo enlazamos con un proceso `Process`. Esto nos permitirá comunicarnos con el proceso creado.

Con las instrucciones: **`int codigoRetorno = proceso.waitFor();`**

**`System.out.println("Cod. Fin: "+codigoRetorno);`**

Forzamos que el proceso original que ejecutó el segundo proceso espere a que este segundo termine, capturando el código de finalización del segundo proceso.

Métodos de la clase **java.lang.Process**

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

Bibliografía:

Clase Runtime: <https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html>

Clase Process: <https://docs.oracle.com/javase/8/docs/api/java/lang/Process.html>

## 2.3 E5: Comparativa entre Runtime y ProcessBuilder.

### Sincronización con códigos de finalización y flujo de salida.

**Ejercicio Práctico:** Se proporciona el código de un programa que acepta dos números por parámetro que definen un intervalo. Este programa genera números aleatorios dentro del intervalo y verifica si el número es primo. El programa termina cuando encuentra un número primo. Mostrará por pantalla todos los números generados. Además contará la cantidad de números generados, que devolverá como código de salida.

```
package pspud1e5.RuntimeYProcessBuilder;

public class Primo {

    static int Numero = 0;
    static int Contador = 0;

    public static void main(String[] args) {

        var N = Integer.parseInt(args[0]);
        var M = Integer.parseInt(args[1]);

        System.out.println("  Generando Número Primo entre "+N+" y "+M+" ");

        boolean Bandera;

        do {
            Numero = (int) Math.floor(Math.random()*(N-M+1)+M);
            Bandera=!esPrimo(Numero);
            if (Bandera) {
                System.out.println("  Numero Generado: "+Numero+" no es primo.");
            } else {
                System.out.println("  Numero Primo Generado: "+Numero);
            }
            Contador++;
        } while (Bandera);

        System.exit(Contador);
    }

    public static boolean esPrimo(int numero){

        int contador = 2;
        boolean primo=true;

        while ((primo) && (contador!=numero/2)){
            if (numero % contador == 0)
                primo = false;
            contador++;
        }

        return primo;
    }
}
```

Si ejecutamos este programa como clase principal del proyecto y estableciendo los parámetros del Run en las propiedades del proyecto como “500 700” obtendremos la siguiente salida o similar:

```
run:
Generando Número Primo entre 500 y 700
Numero Generado: 590 no es primo.
Numero Generado: 654 no es primo.
Numero Generado: 566 no es primo.
Numero Generado: 549 no es primo.
Numero Generado: 594 no es primo.
Numero Generado: 655 no es primo.
Numero Primo Generado: 659
C:\Users\Lopema\AppData\Local\NetBeans\Cache\15\executor-snippets\run.xml:111: The following error occurred while executing this line:
C:\Users\Lopema\AppData\Local\NetBeans\Cache\15\executor-snippets\run.xml:68: Java returned: 7
BUILD FAILED (total time: 0 seconds)
```

**Se pide:** Crea dos programas, uno utilizando ProcessBuilder y otro con Runtime que lancen el programa anterior y capturen su resultado. Por pantalla debe salir un mensaje del estilo:

“Se encontró el primo 659 al intento 7.”