UT 2 : Programación multiproceso

PSP

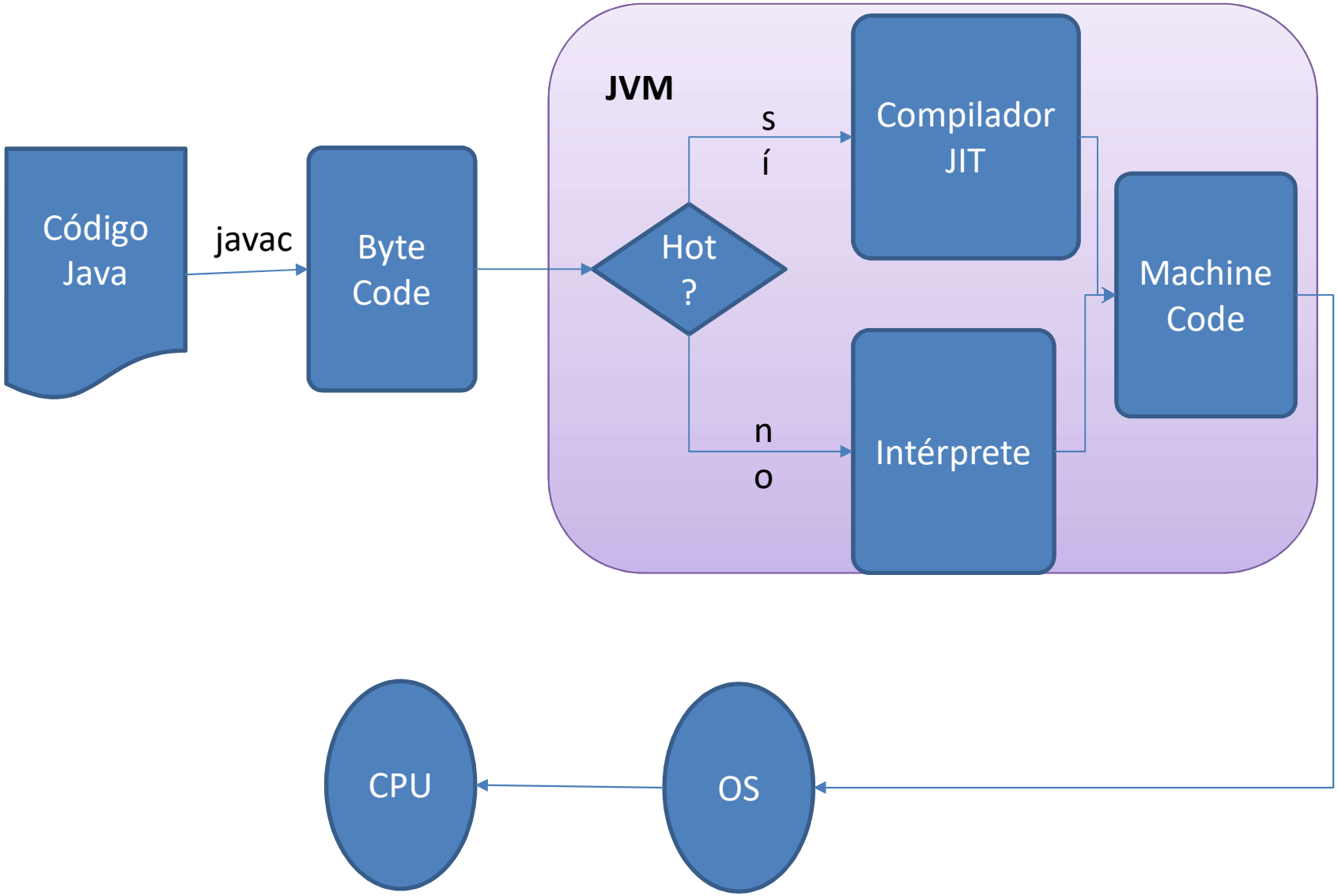
Curso 2024-2025

Ana Alonso

Ejecutables

- Archivo con la estructura necesaria para que el Sistema Operativo ejecute el programa que contiene:
  - Extensión .exe en Windows
- Un programa en Java: ¿Es un ejecutable?

JVM


JVM
Código Java
javac
Byte Code
Hot ?
s í
n o
Compilador JIT
Intérprete
Machine Code
CPU
OS

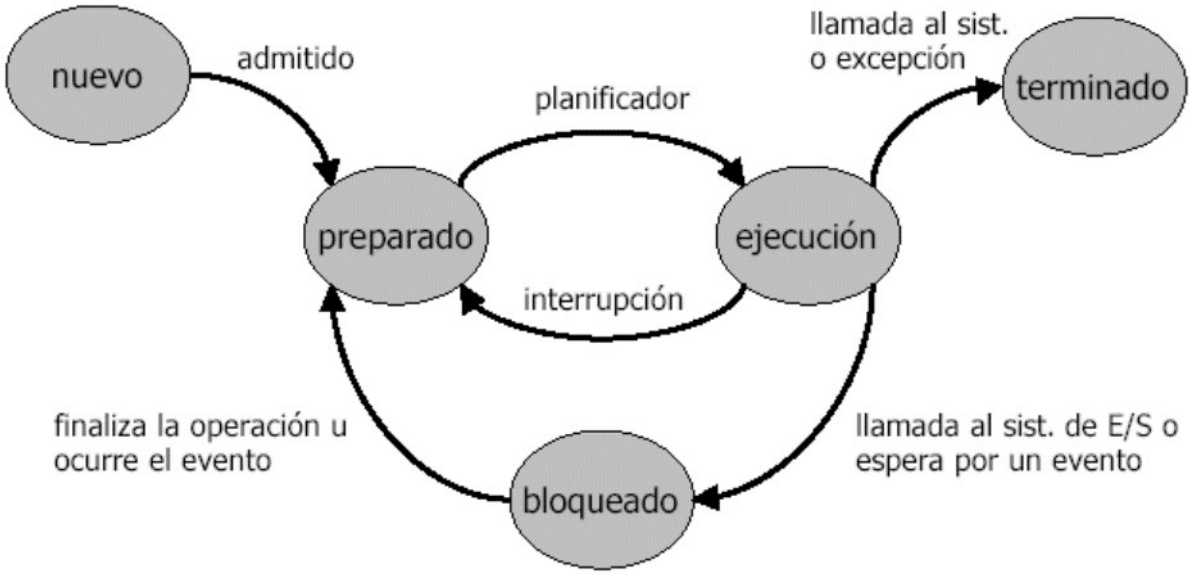
Procesos

- Cuando el Sistema Operativo ejecuta un programa lo hace dentro de un **PROCESO** que necesita de:
  - Tiempo de CPU
  - Memoria
  - Archivos
  - Dispositivos de E/S

Procesos

- El proceso consta de:
  - El código del programa
  - La actividad actual
    - Contador de programa
    - Registros de CPU
  - Pila
    - Parámetros
    - Variables locales
    - Direcciones de retorno
  - Sección de datos
    - Variables globales
    - Memoria dinámica

Estados


nuevo
admitido
planificador
llamada al sist.
o excepción
terminado
preparado
ejecución
interrupción
finaliza la operación u
ocurre el evento
bloqueado
llamada al sist. de E/S o
espera por un evento

Estados

- El sistema operativo controla los estados de un proceso:
  - **Nuevo**: el proceso se está creando.
  - **Preparado**: esperando que se le asigne a un procesador.
  - **En ejecución**: el proceso está en la CPU ejecutando instrucciones.
  - **Bloqueado**: proceso esperando a que ocurra un suceso (ej. terminación de E/S o recepción de una señal).
  - **Terminado**: finalizó su ejecución o falló, por tanto no ejecuta más instrucciones y el SO le retirará los recursos que consume.
- Solo un proceso puede estar ejecutándose en cualquier procesador en un instante dado, pero muchos procesos pueden estar listos y esperando.

Servicio

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

Creación de procesos en java

ProcessBuilder

- **ProcessBuilder** permite crear procesos en el sistema operativo
- **Constructores:**
  - ProcessBuilder(String... command)
  - ProcessBuilder(List<String> command)
  - command es lo mismo que escribiríamos en la consola del sistema: el nombre del programa seguido por los argumentos del programa
- **Métodos:**
  - start(), crea y devuelve un **Process** con los parámetros pasados al constructor del ProcessBuilder

VARARGS

Argumentos de longitud variable

- Solo en la última posición de los parámetros de una función
- Acepta array, lista, o secuencia de argumentos separados por comas

Process

- Process representa a un proceso del sistema operativo
- No se construye directamente, siempre a través de ProcessBuilder
- Métodos:
  - **isAlive**: nos dice si el proceso está vivo
  - **pid**: el identificador de proceso
  - **waitFor**: bloquea el hilo actual esperando a que termine el proceso (admite timeout)
  - **exitCode**: el código de salida del proceso

Crear un proceso

- Utilizando la clase ProcessBuilder lanza un programa que esté instalado en tu máquina:
  1. Localiza la ruta al programa a ejecutar (por ejemplo Acrobat Reader)
  2. Crea el ProcessBuilder pasándole la ruta
  3. Crea un Process llamando a start en el ProcessBuilder
  4. Imprime el PID del proceso por consola
  5. Espera a que termine el proceso
  6. Imprime el código de salida del proceso por pantalla

Crear un proceso con argumentos

- Debes añadir la ruta al fichero como un segundo argumento del constructor de ProcessBuilder que, recuerda, recibe varargs.

System.getproperty

- Utilidad para obtener información del entorno de ejecución:
- “user.dir” es el directorio de trabajo
    - El directorio desde donde se llama al comando java en una consola

E/S en java

Streams

- En java, para **leer** datos desde teclado, desde un fichero o desde un socket, y para **escribir** datos por pantalla, a un fichero o a un socket utilizamos Streams (flujos).

- Por defecto un proceso tiene tres Streams disponibles (flujo estándar de java de la clase java.lang.system) :

  - System.out: la salida estándar (generalmente la consola)
    - Instancia de PrintStream →print(),println()
  - System.in: la entrada estándar (generalmente la consola)
    - Instancia de InputStream →read(), devuelve byte
  - System.err: la salida de error (generalmente la consola)

Readers

- Los métodos read de un InputStream (como System.in) devuelven bytes
- Si queremos leer caracteres utilizamos **InputStreamReader**

```
InputStreamReader inputStreamReader = new
InputStreamReader(System.in);
char[] array = new char[100];
int read = inputStreamReader.read(array, 0, 100);
```

Buffered Readers

- Leer utilizando buffers de caracteres es incómodo.
- Podemos utilizar **BufferedStreamReader** creado a partir de un **InputStreamReader** para leer líneas completas.

```
InputStreamReader inputStreamReader = new
InputStreamReader(System.in);
BufferedReader bufferedReader = new
BufferedReader(inputStreamReader);
bufferedReader.readLine();
```

Writers

- System.out es fácil de usar, pero no me sirve si quiero escribir por un Stream que no sea la salida estándar, por ejemplo a un fichero o a un socket, o en el caso de comunicación entre procesos.
- Si tenemos un OutputStream (trabaja con bytes), podemos utilizar un **OutputStreamWriter** para escribir caracteres.

```
OutputStream outputStream;
OutputStreamWriter outputStreamWriter = new
OutputStreamWriter(outputStream);
```

Liberar recursos

- Debemos liberar correctamente los recursos del sistema, especialmente ficheros y sockets, cuando ya no los necesitemos.

- Los Streams, Readers, Writers y derivados tienen un método **close** para liberar los recursos utilizados.

Leer datos de un subproceso

- El proceso padre puede obtener un InputStream para leer datos desde un proceso hijo llamando a **Process.getInputStream**.
- A partir de ese InputStream creamos InputStreamReader y BufferedReader según nuestras necesidades.
- El proceso hijo utilizará System.out para escribir datos, pero estos no llegan a la salida estándar a la consola, sino al InputStream del padre.

1. Ejecutar comando DIR

- Ejecutar el comando DIR mediante el método getInputStream() de la clase Process para leer el Stream de salida del proceso, es decir lo que el comando DIR envía a la consola:

1. Ejecutar proceso DIR
2. Mostramos carácter a carácter la salida generada por DIR
3. Comprobación del error : 0 bien , -1 mal. El método waitFor() hace que el proceso actual espere hasta que el proceso representado por el objeto Process finalice.

   Este método recoge lo que System.exit() devuelve.

2. Ejecutar comando DIR desde Eclipse

- Ejecutar el comando DIR anterior. En este caso lo ejecutamos desde Eclipse (carpeta **bin**) :

1. Creamos objeto File al directorio donde está ComandoDIR.
2. El proceso a ejecutar es ComandoDIR.
3. Establecemos el directorio donde se encuentra el ejecutable mediante el método directory().
4. Ejecutamos el proceso.
5. Obtenemos la salida devuelta por el proceso.

3. Ejecutar comando DIR con ERROR

- Leemos errores del proceso. La clase Process tiene el método getErrorStream() que nos va a permitir obtener un stream para leer los posibles errores que se producen al lanzar un proceso. En este caso lanzamos el comando DIR con ERROR para comprobarlo:

1. Ejecutar proceso DIR erróneo.
2. Mostramos carácter a carácter la salida generada por DIR.
3. Comprobación del error : 0 bien , -1 mal. El método waitFor() hace que el proceso actual espere hasta que el proceso representado por el objeto Process finalice.
4. Finalmente leemos los errores.