PROGRAMACIÓN MULTIPROCESO

UT1

Programación Multiproceso

Conceptos previos.

Programa.

Es un conjunto de órdenes diseñadas y creadas a través del razonamiento lógico, almacenadas en ficheros respetando la sintaxis de un determinado lenguaje de programación, que una vez ejecutadas realizarán una o varias tareas en un ordenador.

Programación Multiproceso

Proceso.

Un proceso es un programa en ejecución. No se refiere únicamente al código y los datos, sino que incluye todo lo necesario para su ejecución. Esto es:

- **Un contador del programa**: que por dónde se está ejecutando.
- **Una imagen de memoria**: es el espacio de memoria que el proceso está utilizando.
- **Estado del procesador:** se define como el valor de los registros del procesador sobre los cuales se está ejecutando.

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

Programación Multiproceso

Ejecutable.

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

Demonio.

Proceso no interactivo que está ejecutándose continuamente en segundo plano. Proporcionan un servicio básico para el resto de procesos. Los demonios son los "***Servicios***" de Windows.

Programación Multiproceso

Sistema operativo.

El programa que hace de intermediario entre el usuario y las aplicaciones que utiliza y el hardware del ordenador.

Ejecuta los programas del usuario	• Crea procesos • Gestiona su ejecución
Hace sencillo el uso del ordenador	• Hace de interfaz entre el usuario y los recursos
Utiliza los recursos de forma eficiente	• Permite compartir los recursos entre diferentes programas y diferentes usuarios

Programación Multiproceso

Programación concurrente.

Permite tener en ejecución al mismo tiempo múltiples tareas Interactivas, de las siguientes maneras:

Programación Multiproceso

Un único procesador (multiprogramación o programación concurrente)

Sólo un proceso está en ejecución	El SO intercambia los procesos	No mejora el tiempo global de ejecución

Un único procesador con varios núcleos (multitarea o programación paralela)

Como los Dual Cores, Quad Cores…	El SO planifica los trabajos de cada núcleo y los intercambia	Los *cores* comparten la misma memoria y trabajan de forma coordinada	Mejora el rendimiento pues se ejecutan varias instrucciones a la vez del mismo programa	Cada ejecución en cada *core* será una tarea o **hilo de ejecución.**	Se puede utilizar conjuntamente con la programación concurrente

Programación distribuida

Cada ordenador con su procesador y su memoria y conectados en red.	Mejora el rendimiento pero no se comparte la memoria

Programación Multiproceso

Procesos.

El SO pone en ejecución y gestiona los procesos.

A medida que un proceso se ejecuta puede variar su estado.

El SO también puede variar el estado de un proceso.

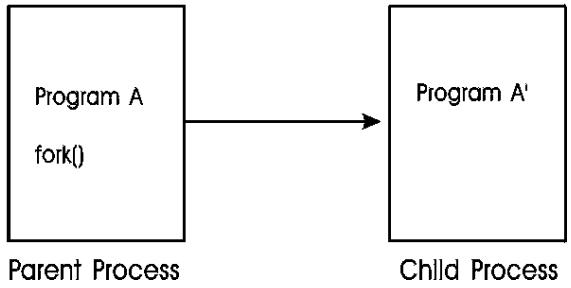
Estados de un proceso

Nuevo	Listo	En ejecución	Bloqueado	Terminado
• El fichero se crea a partir del fichero ejecutable	• El proceso no se encuentra en ejecución pero se encuentra listo para hacerlo • El SO no le ha asignado un procesador • El planificador del SO debe seleccionarlo	• El proceso se está ejecutando	• El proceso espera a que ocurra algún evento (E/S, sincronización…) • Cuando se desbloquea debe ser planificado de nuevo	• Cuando finaliza se libera su imagen de memoria • El mismo proceso llama al Sistema para informarle de su finalización o el Sistema finaliza el proceso con una ***EXCEPCIÓN***

Programación Multiproceso

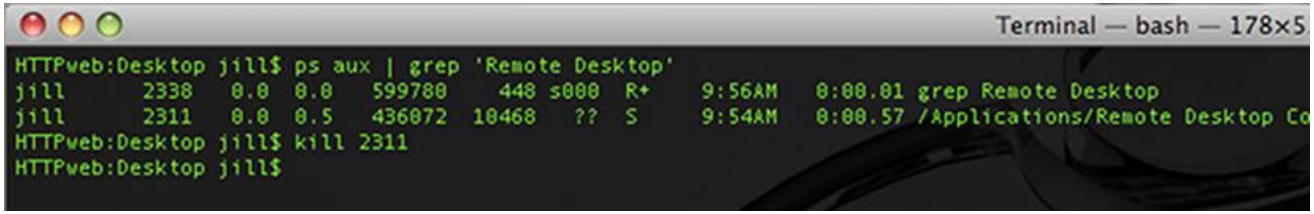
Cómo se crean los procesos:

- Arranque del sistema
- Una llamada al sistema
- Una petición deliberada del usuario
- El inicio de un trabajo por lotes



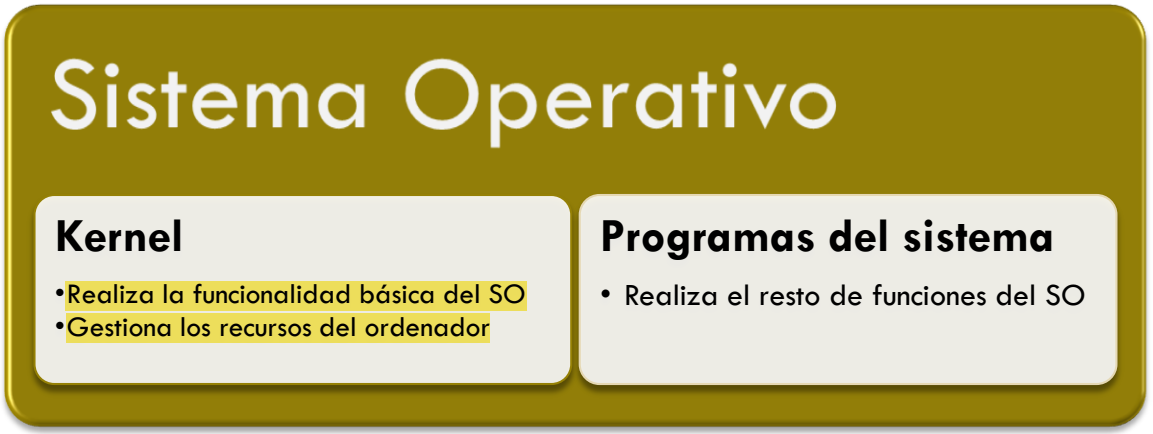
Cómo se terminan los procesos:

- Salida normal
- Salida por error
- Error fatal
- Eliminado por otro proceso



Programación Multiproceso

Los estados de un proceso y el SO:



- El SO utiliza el mecanismo de **interrupciones** para controlar la ejecución de los procesos.
  - Una **interrupción es una suspensión** temporal de la ejecución de un proceso. 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.
  - El SO no es un proceso demonio, sino que él solo se ejecuta respondiendo a interrupciones.

Programación Multiproceso

- Si el proceso necesita un recurso (E/S…) hace una **llamada al Sistema**.
  - Las llamadas al sistema son la interfaz que proporciona el kernel para que los programas de usuario puedan hacer uso de forma segura de determinadas partes del sistema.

El modo **dual** es una característica del hardware que permite al sistema operativo protegerse. El procesador tiene dos modos de funcionamiento indicados mediante un bit:

- **Modo usuario** (1). Utilizado para la ejecución de programas de usuario.
- **Modo *kernel*** *(0), también llamado "modo supervisor" o "modo privilegiado". Las instrucciones del procesador* más delicadas solo se pueden ejecutar si el procesador está en modo *kernel*.

- Si un proceso se ejecuta más tiempo del permitido salta un temporizador que lanza una interrupción

Programación Multiproceso

Colas de procesos.

Para que pueda existir la multiprogramación deben existir varios procesos en memoria y el SO intercambia el uso del procesador para la ejecución de los procesos de forma concurrente. Para ello, el sistema operativo organiza los procesos en varias colas:

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

Programación Multiproceso

Planificación de procesos.

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

Existen dos tipos de planificación:

A corto plazo: selecciona qué proceso de la cola de procesos preparados pasará a ejecución. Se invoca muy frecuentemente por lo que los algoritmos deben ser muy sencillos:

I. **Planificación sin desalojo o cooperativa.** Únicamente se cambia el proceso en ejecución si dicho proceso se bloquea o termina.

II. **Planificación apropiativa.** Además de los casos de la planificación cooperativa, 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.

III. **Tiempo compartido:** cada cierto tiempo (llamado *cuanto)* se desaloja el proceso que estaba en ejecución y se selecciona otro proceso para ejecutarse. En este caso, todas las prioridades de los hilos se consideran iguales.

A largo plazo: selecciona qué procesos nuevos deben pasar a la cola de procesos preparados. Se invoca con poca frecuencia, por lo que puede tomarse más tiempo en tomar la decisión. Controla el grado de multiprogramación (número de procesos en memoria).

Programación Multiproceso

Cambio de contexto.

Nunca se justifica un cambio de contexto.

Cuando el procesador pasa a ejecutar otro proceso, el SO debe guardar el contexto del proceso actual y restaurar el contexto del proceso que el planificador a corto plazo ha elegido ejecutar.

Se conoce como **contexto** a:

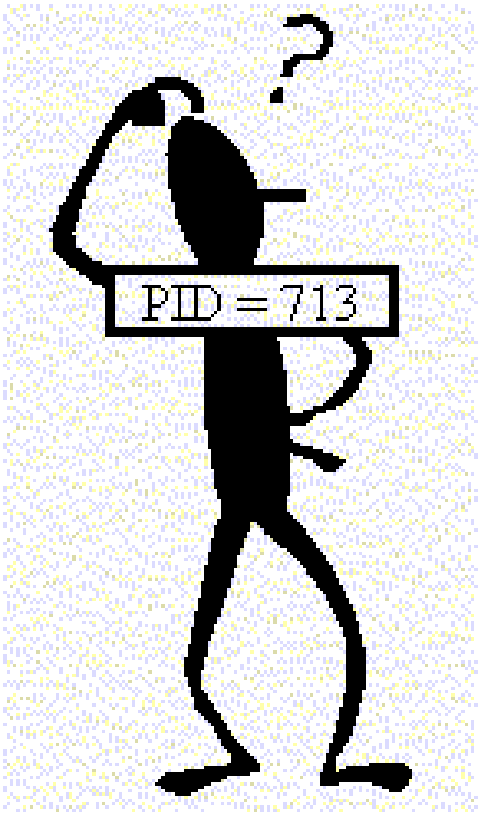
- **Estado del proceso.**
- **Estado del procesador:** valores de los diferentes registros del procesador.
- **Información de gestión de memoria:** espacio de memoria reservada para el proceso.

El cambio de contexto es tiempo perdido, ya que el procesador no hace trabajo útil durante ese tiempo.

Programación Multiproceso

Árbol de procesos.

Cualquier proceso en ejecución siempre depende del proceso que lo creó. A su vez, el nuevo proceso puede crear nuevos procesos, formándose lo que se denomina un **árbol de procesos.** Cuando se arranca el ordenador, y se carga en memoria el *kernel del sistema a partir de su imagen en* disco, se crea el proceso inicial del sistema. A partir de este proceso, se crea el resto de procesos de forma jerárquica, estableciendo padres, hijos, abuelos, etc.



Para identificar a los procesos, los sistemas operativos suelen utilizar un **identificador de proceso** (*process identifier [**PID**]*) *unívoco para cada proceso*.

Programación Multiproceso

Operaciones básicas con procesos.

El proceso creador se llama padre, el proceso creado se llama hijo y la operación de creación, ***create***.

Ambos procesos, padre e hijo, se ejecutan concurrentemente, compartiendo la CPU. Si el padre necesita esperar por el hijo, puede utilizar la operación ***wait***.

Cada proceso tendrá su **imagen de memoria**, aunque sean padre e hijo. Esto permite que puedan compartir recursos para intercambiar información, utilizando ficheros o **memoria compartida**: región de memoria a la que pueden acceder varios procesos cooperativos para compartir información.

En sistemas tipo Windows se utiliza ***createProcess()***, en sistemas UNIX se utiliza ***fork()***.

Programación Multiproceso

Para la terminación de los procesos, si es el propio proceso el que avisa al SO, utiliza la operación ***exit***.

El proceso hijo depende tanto del SO como del padre, que si cree que debe darle fin utiliza la operación ***destroy***.

En cada SO la gestión de procesos es diferente y es posible que si un padre acaba, el SO no permita que los hijos sigan en ejecución y se produce la "terminación en cascada".

Para evitar depender del SO sobre el cual se esté ejecutando utilizaremos la **máquina virtual de Java** [JVM]). JVM es un entorno de ejecución ligero y gratuito multiplataforma que permite la ejecución de binarios o *bytecode del lenguaje de programación Java sobre cualquier SO*.

> Ya refiriéndonos a **JAVA**, los procesos padre e hijo en la JVM no tienen por qué ejecutarse de forma concurrente y no se produce terminación en cascada, pudiendo sobrevivir los hijos a su padre ejecutándose de forma asíncrona.

Programación Multiproceso

Creación de procesos.

La clase que representa un proceso en Java es la clase ***Process***. Los métodos de ***ProcessBuilder.start()*** y ***Runtime.exec()*** crean un proceso nativo en el sistema operativo subyacente donde se está ejecutando la JVM y devuelven un objeto Java de la clase ***Process*** que puede ser utilizado para controlar dicho proceso.

- Process ***ProcessBuilder.start():*** inicia un nuevo proceso utilizando los atributos indicados en el objeto. El nuevo proceso ejecuta el comando y los argumentos indicados en el método *command()*, ejecutándose en el directorio de trabajo especificado por *directory()*, utilizando las variables de entono definidas en *environment()*.
- Process ***Runtime.exec(String[] cmdarray, String[] envp, File dir):*** ejecuta el comando especificado y argumentos en *cmdarray* en un proceso hijo independiente con el entorno *envp* y el directorio de trabajo especificado en *dir*

Programación Multiproceso

El proceso a crear depende del SO en concreto sobre el que se está ejecutando. Y… ¿qué problemas pueden aparecer?

A. No encuentra el ejecutable en esa ruta.
B. No tiene permisos de ejecución.
C. No es un ejecutable válido.
D. etc…

Se lanzará una excepción dependiente del sistema en concreto, pero siempre será una subclase de **IOException**.

Programación Multiproceso

Ejemplo: Creación de un proceso utilizando *ProcessBuilder*

```
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);
    }
```

Programación Multiproceso

```
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);
    }
}
}
```

Programación Multiproceso

Ejemplo: Creación de un proceso mediante Runtime para después destruirlo

```
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);
        }
    }
}
```

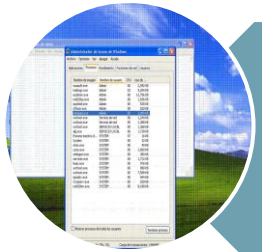
Programación Multiproceso

Comunicación de procesos.

El proceso, como programa en ejecución, recibe información, la transforma y recibe resultados.

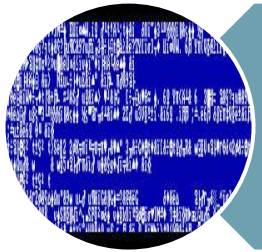
Entrada estándar (stdin)

- Lugar de donde el proceso lee los datos de entrada que requiere para su ejecución.
- Pueden ser teclado, un fichero, de la tarjeta de red, de otro proceso…



Salida estándar (stdout)

- Sitio donde el proceso escribe los resultados que obtiene
- Pueden ser la pantalla, la impresora, otro proceso…



Salida de error (stderr)

- Sitio donde el proceso envía los mensajes de error.
- Habitualmente el mismo que la salida estándar, pero puede especificarse otro, por ejemplo un fichero de errores…

Programación Multiproceso

La utilización de *System.out y System.err en Java se puede ver como un ejemplo de utilización de estas salidas.*

En Java, el proceso hijo creado de la clase **Process** no tiene su propia interfaz de comunicación, por lo que el usuario no puede comunicarse con él directamente. Todas sus salidas y entradas de información (*stdin, stdout y stderr*) se redirigen al proceso padre a través de los siguientes flujos de datos o streams:

- ***OutputStream***: flujo de salida al proceso hijo. El stream está conectado por un pipe a la entrada estándar (stdin) del proceso hijo.
- ***InputStream***: flujo de entrada desde el proceso hijo. El stream está conectado por un pipe a la salida estándar (stdout) del proceso hijo.
- ***ErrorStream:*** flujo de error del proceso hijo. El stream está conectado por un pipe a la salida estándar (stderr) del proceso hijo. Sin embargo, hay que saber que, por defecto, para la JVM, stderr está conectado al mismo sitio que stdout. Si se desea tenerlos separados, lo que permite identificar errores de forma más sencilla, se puede utilizar el método redirectErrorStream(boolean) de la clase ProcessBuilder.

Utilizando estos *streams, el proceso padre puede enviarle datos al proceso hijo y recibir los resultados de salida* que éste genere comprobando los errores.

En algunos sistemas operativos, el tamaño de los buffers de entrada y salida que corresponde a *stdin y stdout* está limitado. En este sentido, un fallo al leer o escribir en los flujos de entrada o salida del proceso hijo puede provocar que el proceso hijo se bloquee.

Programación Multiproceso

Ejemplo de comunicación de procesos utilizando un buffer:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Arrays;
public class ComunicationBetweenProcess {
        public static void main(String args[]) throws IOException {
                Process process = new ProcessBuilder(args).start();
                InputStream is = process.getInputStream();
                InputStreamReader isr = new InputStreamReader(is);
                BufferedReader br = new BufferedReader(isr);
                String line;

                System.out.println("Salida del proceso " +Arrays.toString(args) + ":");
                while ((line = br.readLine()) != null) {
                        System.out.println(line);
                }
        }
}
```

Programación Multiproceso

Además de comunicarse mediante flujo de datos, existen otras alternativas para la comunicación de procesos:

- Sockets
- JNI (*Java Native Interface*)
- Librerías de comunicación no estándares

Programación Multiproceso

Sincronización de procesos.

1) Los envíos y recepciones por los flujos de datos permiten a un proceso hijo comunicarse con su padre a través de un canal de comunicación unidireccional bloqueante, por lo que se puede considerar un método de sincronización.

2) Se puede esperar por la finalización del hijo mediante ***wait()***. Dicha operación bloquea al proceso padre hasta que el hijo finaliza su ejecución mediante ***exit***. El código de finalización que devuelve el hijo no tiene nada que ver con la información que puedan compartir mediante los *streams*, es sólo un entero que indica el estado de la finalización del proceso hijo.

Programación Multiproceso

PROGRAMACIÓN MULTIPROCESO.

- La multiprogramación puede producirse entre procesos totalmente independientes o entre procesos que pueden cooperar entre sí para realizar una tarea común.
- Si se pretende realizar procesos que cooperen entre sí, deberá ser el propio desarrollador quien lo implemente utilizando la comunicación y sincronización.

A la hora de realizar un programa *multiproceso cooperativo*, se siguen las siguientes **fases**:

1. **Descomposición funcional**. Se identifican las diferentes funciones que debe realizar la aplicación y las relaciones existentes entre ellas.

2. **Partición**. Distribución de las diferentes funciones en procesos. Se establece el esquema de comunicación.

3. **Implementación**. En Java se permite únicamente métodos sencillos de comunicación y sincronización de procesos para realizar la cooperación.

Programación Multiproceso

Multiproceso en Java => clase ***Process***.

Método	Retorno	Descripción
getOutputStream()	OutputStream	Obtiene el flujo de salida del proceso hijo conectado al stdin
getInputStream()	InputStream	Obtiene el flujo de entrada del proceso hijo conectado al stdout
getErrorStream()	InputStream	Obtiene el flujo de entrada del proceso hijo conectado al stderr
destroy()	void	Implementa la operación destroy
waitFor()	int	Implementa la operación wait
exitValue()	int	Obtiene el valor de retorno del proceso hijo