Procesos

Un proceso es un archivo ".exe" que está en ejecución y bajo el control del sistema operativo. Un proceso puede estar en diversos estados:

- En ejecución: está dentro del microprocesador.
- Pausado/detenido/en espera: el proceso necesita más ciclos de cpu para ejecutarse, pero en estos momentos el sistema operativo ha decidido dejar paso a otro.
- Interrumpido: el proceso necesita más ciclos de cpu para ejecutarse, pero en estos momentos el usuario ha decidido interrumpir la ejecución
- Otros estados que dependen del sistema operativo.

Servicios

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

Hilos

Un hilo es un concepto más avanzado que un proceso: al hablar de procesos cada uno tiene su propio espacio en memoria. Si abrimos 20 procesos cada uno de ellos consume 20x de memoria RAM. Un hilo es un proceso mucho más ligero, en el que el código y los datos se comparten de una forma distinta.

Un proceso no tiene acceso a los datos de otro procesos. Sin embargo un hilo sí accede a los datos de otro hilo. Esto complica algunas cuestiones a la hora de programar.

Creación de procesos en java

En java podemos utilizar la funcionalidad que nos ofrece la clase *ProcessBuilder* para crear procesos. Ver el ejemplo *"Ejemplo_1_ProcessBuilder".*

Para la realización de los próximos ejercicios se ofrece un pequeño programa *"Ejemplo_2_chuleta"*, con ese código aprendemos a obtener:

- Tiempo actual del sistema
- Información sobre el entorno en el que se ejecuta la aplicación
- Número de cores
- Memoria total de la JVM de la que disponen los futuros objetos y máxima disponible
- Propiedades del sistema
- Ejecución de un proceso del sistema.

Ejercicios

1. Crea un programa que nos muestre la siguiente información del sistema:
    - Número de núcleos (cores) del ordenador.
    - Memoria en Mb disponible y máxima en la Java Virtual Machine (jvm).
    - Listado de todas las propiedades del sistema que ofrezca la clase Properties.

2. Crea un programa que muestre los procesos que tiene el sistema y nos vaya preguntando por cada uno de ellos si lo queremos eliminar. Haz que funcione tanto en Windows como en Linux. *Solucion_Ejercicio_2_Matar_Procesos*

3. Calcula el tiempo, en segundos, que tarda el ordenador en ordenar dos arrays de enteros de 100.000 elementos con números aleatorios de entre 1 y 1000. Monitoriza la CPU mientras se ejecuta. *Solucion_Ejercicio_3_Ordenar_sin_hilos*

Programación secuencial vs programación paralela

Hasta ahora hemos hecho programación secuencial, cuando lanzamos cualquier tarea como ordenar un vector, ésta se ejecuta en un núcleo hasta que termina y luego comienza la siguiente tarea.
En programación paralela, podremos lanzar varias tareas a la vez y cada una se ejecutará en un núcleo diferente.
Si no ejecutamos dos cosas a la vez no tendremos nunca concurrencia o paralelismo y por tanto tendremos programación secuencial.
Además **NO** es lo mismo concurrencia que paralelismo (Lo explicaremos más adelante).
Con la concurrencia conseguiremos un incremento de rendimiento muy significativo en nuestro software.
Actualmente el hecho de que muchos ordenadores presentan varios núcleos en sus microprocesadores permite que nuestro software pueda ser mucho más potente.
La concurrencia permite que se puedan hacer tareas en segundo plano y programar tareas asíncronas.
El inconveniente es que los programas serán más complejos y sobretodo se presentarán problemas indeterminación.

En la programación concurrente necesitamos saber lanzar hilos.

Hilos

Un hilo es básicamente una forma que tenemos de planificar una serie de tareas. Anteriormente programabamos una tarea, después otra y luego la siguiente. Con la programación concurrente queremos que se ejecuten dos o más tareas a la vez. Un hilo al ser un planificador de tareas nos permitirá hacer esta ejecución de tareas según nos interese.
Además como se mencionó anteriormente mientras que un proceso tiene su propio espacio de memoria, un hilo es un proceso mucho más ligero en el que código y los datos se comparten de una forma distinta. Un proceso puede estar formado por muchos hilos que trabajan conjuntamente para conseguir algo.

Podemos lanzar hilos de dos formas:

- Heredando de la clase Thread

```
public class Main
{
	public static void main(String[] args)
	{
		MiThread miThread = new MiThread();
		miThread.start();
	}
}

public class MiThread extends Thread
{
	@Override
	public void run()
	{
		// Este método será invocado al llamar al método start.
	}
}
```

- Implementando el interfaz Runnable

```
public class Main
{
	public static void main(String[] args)
	{
		MiRunnable miRunnable = new MiRunnable();
		Thread miThread = new Thread(miRunnable);
		miThread.start();
	}
}

public class MiRunnable implements Runnable
{
	@Override
	public void run()
	{
		// Este método será invocado al llamar al método start.
	}
}
```

Ver *Ejemplo_3_Creacion_Hilo* y *Ejemplo_4_Crear_Hilos_Con_Parametros*

Hay que tener en cuenta que el *Main* se ejecuta en su propio hilo y éste puede terminar su función antes que los hilos lanzados, por tanto es posible que no se vea la terminación del resto de los hilos.
Si nos interesa que el hilo principal (el del *Main*) se ejecute después que algún otro hilo de los que fueron lanzados por el *Main* podemos utilizar el método *join()* de cada hilo. Ver *Ejemplo_5_Lanzar_Hilos_Desde_Vector_Join*

Ejercicios

4. Realiza el ejercicio anterior (Ejercicio 3) de nuevo, pero lanzando un hilo para cada proceso de ordenación. Monitoriza la CPU mientras se ejecuta. ¿Tarda menos?. Solución: *Solucion_Ejercicio_4_Ordenar_con_hilos*

5. Suponiendo que se dispone de una máquina con 4 cores, muestra de una manera eficiente los números primos que existen entre uno hasta un millón. Puedes crear una clase en la que al constructor se le indique el rango donde tiene que buscar los números primos. ¿Aparecerán ordenados?

```
public class RnBuscaPrimos implements Runnable
{
        private int _inicio;
        private int _fin;

        public RnBuscaPrimos(int inicio, int fin)
        {
                _inicio = inicio;
                _fin = fin;
        }
        @Override
        public void run()
        {
                …
        }
}
```

Indeterminismo

Si tenemos una variable común a todos los hilos que se la podemos pasar como un parámetro en el constructor de cada hilo e intentamos incrementarla en el método run de cada hilo. El resultado será indeterminado porque todos los hilos acceden

concurrentemente a la misma variable y si un hilo “h1” intenta acceder a la variable común para modificar su valor, es posible que otro hilo “h2” haya podido modificar su valor entre tanto y por lo que el hilo “h1” tendrá un valor erróneo En el ejemplo: *Ejemplo_6_Indeterminismo* se produce **indeterminismo,** prueba a ejecutar varias veces ese código y comprobarás que el resultado en cada ejecución.
La única forma de solucionar esto es mediante el concepto “**exclusión mutua**”, es decir que a esa variable solo pueda acceder un hilo a la vez. Por tanto es necesario controlar el acceso a la “**sección crítica**” que en este caso sería el método “*incrementaContador*” de la clase “*AlmacenContador*”. Una forma de evitar que dos hilos accedan a la vez a misma sección crítica es creando “Métodos sincronizados de instancia” y/o “Métodos sincronizados estáticos”.

Métodos sincronizados de instancia:

Los métodos sincronizados no permiten la ejecución simultánea de los mismos por varios hilos, obligando a los hilos que los invocan a esperar a que termine su ejecución por parte de otro hilo.
Un método sincronizado de instancia es sincronizado en la instancia (objeto) que contiene el método. Por tanto, cada instancia tiene sus propios métodos sincronizados e independientes de cada una de las instancias que pudiera haber.

```
public synchronized void incrementaContador()
{
    contador++;
}
```

Métodos sincronizados estáticos:

Los métodos sincronizados estáticos lo son a nivel de clase. Da igual que se invoquen desde diferentes objetos, guardarán una sincronización global.

```
public static synchronized void incrementaContador()
{
    contador++;
}
```

En el ejemplo: *Ejemplo_6_Indeterminismo* prueba a poner el modificador **synchronized** en el método *incrementaContador()* y ejecuta varias veces el código. Verás que siempre da el resultado esperado.

Otra forma de proteger la sección crítica es mediante "*Bloques sincronizados en métodos de instancia*"

Bloques sincronizados en métodos de instancia

El objeto que va entre paréntesis hace de monitor, es decir, es el recurso potencialmente compartido con otros hilos (puede ser cualquier objeto). Con los bloques sincronizados se permite por tanto sincronizar a nivel de bloque, y no de método completo, aunque su uso no está recomendado. Si ponemos this como monitor, hace que la sincronización tenga efecto en toda la instancia del objeto donde se encuentra el bloque. Prueba a modificar el código del ejemplo *Ejemplo_6_Indeterminismo.* Modifica la clase *"AlmacenContador*" con el código mostrado abajo, comprobarás que funciona correctamente mostrando el resultado esperado.

```
public class AlmacenContador {

    private int contador;
    private Object obj = new Object();

    public AlmacenContador()
    {
        contador = 0;
    }

    public void incrementaContador()
    {
        synchronized (obj){
            contador++;
        }
    }

    public int getContador()
    {
        return contador;
    }

}
```

A veces no es necesario sincronizar los hilos mediante **synchronized** puesto que los hilos pueden trabajar de forma paralela accediendo a secciones distintas del elemento en común. Por ejemplo supongamos necesitamos multiplicar por 10 todos los elementos de un vector, en este caso podemos hacer que cada hilo acceda a secciones distintas del elemento común que queremos modificar. Ver *Ejemplo_7_Seccion_Critica_Sin_Synchronized*

Ejercicio

6. Realiza el ejercicio anterior, pero esta vez los números deben ordenarse. Utiliza un almacén común “AlmacenOrdenadoSincronizado” donde cada hilo vaya depositando los números primos que vaya consiguiendo. Puedes utilizar un objeto *TreeSet* para que se vayan ordenando y sería recomendable utilizar el parámetro **synchronized**. Solución *Solucion_Ejercicio_6_Buscar_Primos_Ordenados*

Señales

A veces es necesario sincronizar dos o más hilos, que se envíen señales para coordinarse a la hora de realizar determinadas operaciones. De esta manera se evita la espera activa y se gana eficiencia.
Un hilo que ejecuta *wait*() en un objeto pasa a estar inactivo hasta que otro hilo ejecuta *notify*() o *notifyAll*() para ese mismo objeto. ***Estos métodos se han de ejecutar en un método o bloque sincronizado synchronized.***

```
public synchronized void metodoDurmiente()
{

        try
        {
                wait(); // El hilo pasa a estar inactivo
        }
        catch (InterruptedException e)
        {

        }
        System.out.println("Alguien me despertó con un notify()");

        notify(); // Despierto otro hilo (cualquiera) que pudiera estar esperando a que termine
}
```

Cuando un hilo ejecuta *wait*(), libera el bloqueo sobre el método o bloque sincronizado en el que está, haciendo así posible que desde otro método o bloque sincronizado se pueda ejecutar *notify*(). Con *notifyAll*() se libera la espera de todos los hilos.

Demonios
Cuando un hilo se lanza como demonio, al terminar el hilo desde el cual ha sido lanzado, éste termina también. Se ha de configurar como demonio antes de *start*.

```
th.setDaemon(true);
th.start();
```

Supongamos que se desea controlar el orden en el que se ejecuta un *println* por parte de cada hilo, de tal forma que se escriban los identificadores de los hilos por orden de una serie de hilos lanzados por el *main*. Sol. *Ejemplo_8_Coordinacion_Hilos_Wait_Notify_1*

En el siguiente ejemplo se realiza una suma total de unas sumas calculadas previamente por cada hilo de forma independiente, de forma que al final la suma tiene sentido y concuerda con el resultado esperado. En este ejemplo no se utilizan señales como *wait()* o *notify()*, pero se utiliza un ***synchronized*** a nivel de bloque mediante una variable estática. Sol. *Ejemplo_9_Suma_Total*

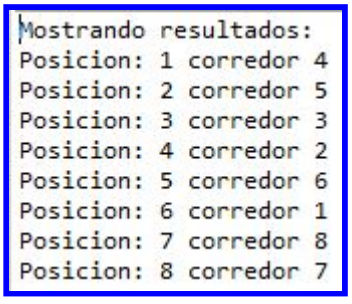
En el siguiente ejemplo se implementa un conocido patrón de diseño *"Singlenton"* en el que varios hilos acceden a la información de una única instancia. Sol. *Ejemplo_10_Patron_Singlenton_Hilos*

Ejercicios

7. Codifica un programa que lance una carrera de 8 corredores, representados cada uno de ellos por un hilo:

    - Cada corredor tendrá un número de dorsal desde el 1 hasta el 8 y tendrá que incrementar una variable de uno en uno desde el 1 hasta el 10000 (en un bucle for por ejemplo) para llegar a la meta.
    - Los corredores saldrán en el mismo momento en el que ejecute la aplicación.
    - Cuando hayan llegado los 8 corredores se mostrará un listado de resultados.

    Un ejemplo de ejecución podría ser el siguiente:

    
    ```
    Mostrando resultados:
    Posicion: 1 corredor 4
    Posicion: 2 corredor 5
    Posicion: 3 corredor 3
    Posicion: 4 corredor 2
    Posicion: 5 corredor 6
    Posicion: 6 corredor 1
    Posicion: 7 corredor 8
    Posicion: 8 corredor 7
    ```

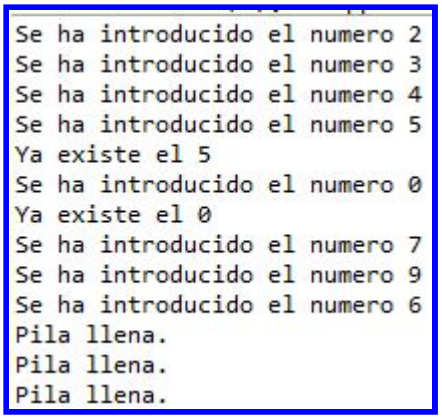
    Sol: *Solucion_Ejercicio_7_Corredores*

8. Codifica una clase Elementos que permita almacenar en forma de pila y sin repetir un determinado número de elementos positivos de tipo int. El número de elementos máximo a almacenar será especificado en el constructor.

    - El método sacarElemento extrae un elemento y lo devuelve. Devolverá -1 si la estructura está vacía.

- El método ponerElemento añade un elemento si hay espacio, no existe previamente, y es positivo o 0. Devuelve false si no lo ha podido añadir, true en caso contrario.

Ejemplo de ejecución:



```
Se ha introducido el numero 2
Se ha introducido el numero 3
Se ha introducido el numero 4
Se ha introducido el numero 5
Ya existe el 5
Se ha introducido el numero 0
Ya existe el 0
Se ha introducido el numero 7
Se ha introducido el numero 9
Se ha introducido el numero 6
Pila llena.
Pila llena.
Pila llena.
```

Sol: *Solucion_Ejercicio_8_Productor_Consumidor*

Una vez comprobado el correcto funcionamiento de la estructura, crea dos hilos dedicados a hacer cada uno de ellos una tarea distinta sobre una instancia de la clase Elementos:

- thPonerElementos: Añadirá, indefinidamente, enteros aleatorios.
- thSacarElementos: Extraerá, indefinidamente, los enteros almacenados.

Para poder hacer un seguimiento de lo que está sucediendo, añade mensajes por consola en el código:

- Extraído el elemento X.
- Sin elementos.
- Añadido el elemento X.
- Elemento X ya existe.
- No hay sitio.

9. Optimiza la eficiencia del ejercicio 8 usando las señales wait() y notify() en los hilos productor y consumidor para evitar que hagan peticiones repetitivamente que no serán satisfechas como extraer elemento si la estructura está vacía o meter elemento si la estructura está llena.
   Sol: *Solucion_Ejercicio_9_Productor_Consumidor_Seniales*

10. Sincroniza dos hilos, uno encargado de escribir ping y otro pong, para que el resultado de la ejecución sea la secuencia infinita ping, pong, ping, pong, ping, pong.

```
public class Main
{
        public static void main(String[] args)
        {
                EscribePingPong escribePingPong = new EscribePingPong();
                Thread thPing = new Thread(new EscribePing(escribePingPong));
                Thread thPong = new Thread(new EscribePong(escribePingPong));
                thPong.start();
                thPing.start();
        }
}
```

```
public class EscribePingPong
{
        public synchronized void escribePing()
        {
            ...
                System.out.println("ping");
            ...
        }
        public synchronized void escribePong()
        {
            ...
                System.out.println("pong");
            ...
        }
}
```

Sol: *Solucion_Ejercicio_10_Ping_Pong*