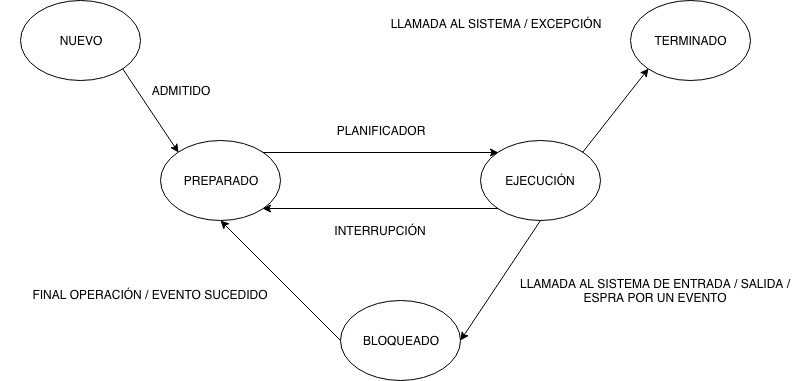
PAC 6. UF2. - Desarrollo

*Programación de servicios y procesos.*

Aníbal Santos Gómez

1. **Ejercicio 1.  Realiza un esquema de los estados de un proceso cualquiera, e identifica lo que ocurre en cada uno de ellos.**

* Nuevo: aquí el proceso se crea pero no ha sido admitido en el grupo de procesos ejecutables por el sistema operativo.
* Listo: el proceso se para temporalmente y está preparado para ejecutarse cuando tenga oportunidad de hacerlo.
* Ejecución: el proceso se está ejecutando, por lo tanto, usando el procesador.
* Bloqueado: el proceso no puede hacer nada hasta que no se de un evento externo.
* Terminado: el proceso sale del grupo de ejecutables en el sistema operativo. A partir de aquí se liberan los recursos utilizados por dicho proceso.

1. **Ejercicio 2.  A partir del siguiente conjunto de instrucciones, indica las que se pueden ejecutar concurrentemente y las que no. Justifica tus respuestas.**

Instrucción 1: a = x +y

Instrucción 2: c = z – 1

Instrucción 3: c = a – b

Instrucción 4: w = z + 1

Primero hacemos los conjuntos de lectura y escritura de las distintas instrucciones.

L(I1) = {x,y} E(I1) = {a}

L(I2) = {z} E(I2) = {c}

L(I3) = {a,b} E(I3) = {c}

L(I4) = {z} E(I4) = {w}

Para que dos conjuntos se puedan ejecutar concurrentemente se deben cumplir estas 3 condiciones:

- La inserción entre las variables leídas por un conjunto de instrucciones Ii y las variables escritas por otro conjunto Ij debe ser vacío, es decir, no debe haber variables comunes:

L(Ii) ∩ E(Ij) = Ø

- La inserción entre las variables de escritura de un conjunto de instrucciones Ii y las variables leídas por otro conjunto Ij debe ser nulo, es decir, no debe haber variables comunes:

E(Ii) ∩ L(Ij) = Ø

- Por último, la intersección entre las variables de escritura de un conjunto de instrucciones Ii y las variables de escritura de un conjunto Ij debe ser vacío, no debe haber variables comunes:

E(Ii) ∩ E(Ij) = Ø

Aplicando estas tres condiciones a cada pareja de instrucciones descubriremos cuales se pueden ejecutar concurrentemente y cuales no:

Conjunto de I1 e I2

L(I1) ∩ E(I2) = Ø

E(I1) ∩ L(I2) = Ø

E(I1) ∩ E(I2) = Ø

Conjunto de I1 e I3

L(I1) ∩ E(I3) = Ø

E(I1) ∩ L(I3) ≠ Ø

E(I1) ∩ E(I3) = Ø

Conjunto de I1 e I4

L(I1) ∩ E(I4) = Ø

E(I1) ∩ L(I4) = Ø

E(I1) ∩ E(I4) = Ø

Conjunto de I2 e I3

L(I2) ∩ E(I3) = Ø

E(I2) ∩ L(I3) ≠ Ø

E(I2) ∩ E(I3) = Ø

Conjunto de I2 e I4

L(I2) ∩ E(I4) = Ø

E(I2) ∩ L(I4) = Ø

E(I2) ∩ E(I4) = Ø

Conjunto de I3 e I4

L(I3) ∩ E(I4) = Ø

E(I3) ∩ L(I4) ≠ Ø

E(I3) ∩ E(I4) = Ø

Por lo tanto, las instrucciones que se pueden ejecutar concurrentemente son: **I1 e I2, I1 e I4, I2 e I4, I3 e I4.**

1. **Ejercicio 3.  A partir del siguiente conjunto de instrucciones, indica las que se pueden ejecutar concurrentemente y las que no. Justifica tus respuestas.**

Instrucción 1: x = z + y

Instrucción 2: z = y + 1

Instrucción 3: c = z + 1

Instrucción 4: x = c + 1

Primero hacemos los conjuntos de lectura y escritura de las distintas instrucciones.

L(I1) = {z,y} E(I1) = {x}

L(I2) = {y} E(I2) = {z}

L(I3) = {z} E(I3) = {c}

L(I4) = {c} E(I4) = {x}

Para que dos conjuntos se puedan ejecutar concurrentemente se deben cumplir estas 3 condiciones:

- La inserción entre las variables leídas por un conjunto de instrucciones Ii y las variables escritas por otro conjunto Ij debe ser vacío, es decir, no debe haber variables comunes:

L(Ii) ∩ E(Ij) = Ø

- La inserción entre las variables de escritura de un conjunto de instrucciones Ii y las variables leídas por otro conjunto Ij debe ser nulo, es decir, no debe haber variables comunes:

E(Ii) ∩ L(Ij) = Ø

- Por último, la intersección entre las variables de escritura de un conjunto de instrucciones Ii y las variables de escritura de un conjunto Ij debe ser vacío, no debe haber variables comunes:

E(Ii) ∩ E(Ij) = Ø

Aplicando estas tres condiciones a cada pareja de instrucciones descubriremos cuales se pueden ejecutar concurrentemente y cuales no:

Conjunto de I1 e I2

L(I1) ∩ E(I2) = Ø

E(I1) ∩ L(I2) = Ø

E(I1) ∩ E(I2) = Ø

Conjunto de I1 e I3

L(I1) ∩ E(I3) = Ø

E(I1) ∩ L(I3) ≠ Ø

E(I1) ∩ E(I3) = Ø

Conjunto de I1 e I4

L(I1) ∩ E(I4) = Ø

E(I1) ∩ L(I4) = Ø

E(I1) ∩ E(I4) = Ø

Conjunto de I2 e I3

L(I2) ∩ E(I3) = Ø

E(I2) ∩ L(I3) ≠ Ø

E(I2) ∩ E(I3) = Ø

Conjunto de I2 e I4

L(I2) ∩ E(I4) = Ø

E(I2) ∩ L(I4) = Ø

E(I2) ∩ E(I4) = Ø

Conjunto de I3 e I4

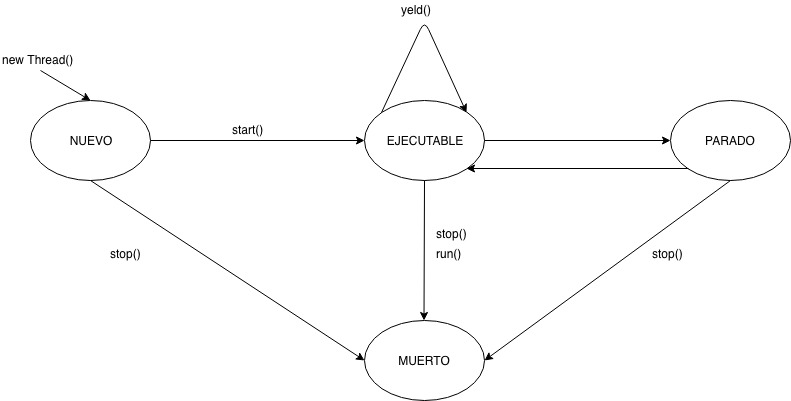
L(I3) ∩ E(I4) = Ø

E(I3) ∩ L(I4) ≠ Ø

E(I3) ∩ E(I4) = Ø

Por lo tanto, las instrucciones que se pueden ejecutar concurrentemente son: **I1 e I3, I2 e I4.**

1. **Ejercicio 4.  Realiza un esquema de los estados de un hilo, e identifica lo que ocurre en cada uno de ellos**.



* Nuevo: se creará un nuevo hilo sin arrancar, que será un objeto Thread vacío.
* Ejecutable: el hilo está disponible para ponerse en ejecución, aunque no tiene porque estar en ejecución.
* Parado: el hilo no está ejecutándose aunque podría hacerlo pero existe un evento que no lo permite.
* Muerto: la ejecución ha finalizado, ya sea natural o no.

1. **Ejercicio 5.  Cuando varios hilos comparten el mismo espacio de memoria es posible que aparezcan algunos problemas de sincronización, explícalos con ejemplos**.

Lor procesos se caracterizan por no compartir la memoria con la que trabajan y ser independientes entre ellos. Los hilos por el contrario suelen compartir la memoria con la que trabajan, puesto que pertenecen al mismo proceso.

El uso de hilos en lugar de procesos tiene ventajas como la no reserva de memoria para cada hilo puesto que es compartida y que además más rápido cambiar de un hilo a otro que de un proceso a otro.

Uno de los principales problemas es la medida del consumo de recursos utilizados, ya que cuando existe un solo proceso existe la posibilidad de que el consumo de memoria y los recursos del procesador se incrementen por la ejecución y se sobrecargue el sistema.

Cuando varios hilos comparten el mismo espacio de memoria es posible que aparezcan algunos problemas de sincronización:

* Condición de carrera: que se dará cuando en la ejecución de un programa en la que su salida depende de la secuencia de eventos que se produzcan.
* Inconsistencia de memoria: los hilos ven diferentes valores para el mismo elemento.
* Inanición: se deniega siempre el acceso a un recurso compartido al mismo hilo, de forma que quede bloqueado a la espera del mismo.
* Interbloqueo: un hilo está esperando por un recurso compartido que está asociado a un hilo cuyo estado es bloqueado.

1. **Ejercicio 6.  Implemente el problema de los filósofos en Java. Ayúdate de las siguientes indicaciones. Necesitamos tres clases Java**:

* **Filósofo: donde se van a realizar todas las operaciones relacionadas con el filósofo.**
* **Tenedor: donde se van a realizar todas las operaciones relacionadas con el tenedor.**
* **Principal: el programa principal que lo lanzará todo.**

**Delante de los métodos de la clase tenedor deberemos poner synchronized, con esto nos aseguramos de que solo va a existir un hilo ejecutándose a la vez en ese método.**

**La clase filósofo debe extender de Thread, para poder usar el método run dentro de esta clase cuando se llame al método start() desde la clase filósofos.**

package Philosophers;

public class Main {

public static void main(String[] args)

{

Fork fork[] = new Fork[5];

fork[0] = new Fork(0);

fork[1] = new Fork(1);

fork[2] = new Fork(2);

fork[3] = new Fork(3);

fork[4] = new Fork(4);

Philosopher philosopher[] = new Philosopher[5];

philosopher[0] = new Philosopher(0, fork[0], fork[1]);

philosopher[1] = new Philosopher(1, fork[1], fork[2]);

philosopher[2] = new Philosopher(2, fork[2], fork[3]);

philosopher[3] = new Philosopher(3, fork[3], fork[4]);

philosopher[4] = new Philosopher(4, fork[4], fork[0]);

philosopher[0].start();

philosopher[1].start();

philosopher[2].start();

philosopher[3].start();

philosopher[4].start();

}

} package Philosophers;

public class Fork

{

private boolean busy = false;

Fork(int id)

{

}

synchronized void leave()

{

this.busy = false;

notify();

}

synchronized boolean isBusy()

{

if (this.busy)

{

return true;

}

else

{

return false;

}

}

synchronized void take()

{

this.busy = true;

}

void drop()

{

this.busy = false;

}

}

package Philosophers;

import java.util.Random;;

public class Philosopher extends Thread

{

private int id;

private int choosen;

private Fork left;

private Fork right;

private boolean takeLeft;

private boolean takeRight;

private boolean flag;

private Random random;

public Philosopher(int id\_, Fork left\_, Fork right\_)

{

id = id\_;

left = left\_;

right = right\_;

random = new Random();

}

public void think()

{

try

{

Thread.sleep(5000);

}

catch (InterruptedException iE)

{

}

}

public void eat()

{

try

{

System.out.println("El filósofo " + id + " se encuentra comiendo.");

Thread.sleep(5000);

System.out.println("está terminando de comer [...]");

System.out.println(id);

}

catch (InterruptedException iE)

{

}

}

public void run()

{

while(true) // Iniciamos un bucle infinito

{

this.think(); // cada filosofo piensa 5 minutos.

flag = false;

choosen = random.nextInt(2); //generamos un numero aleatorio entre 0 y 1 que determina las acciones

if(choosen == 0)

{

if(!left.isBusy())

{

left.take();

takeLeft = true;

}

else if(!right.isBusy())

{

right.take();

takeRight = true;

}

}

else if(choosen == 1)

{

if(!right.isBusy())

{

right.take();

takeRight = true;

}

else if(!left.isBusy())

{

left.take();

takeLeft = true;

}

if(takeLeft == true)

{

if(!right.isBusy())

{

right.take();

takeRight = true;

flag = true;

}

else

{

left.drop();

takeLeft = false;

}

}

if(takeRight == true && !flag)

{

if(!left.isBusy())

{

left.take();

takeLeft = true;

}

else

{

right.drop();

takeRight = false;

}

}

}

if(takeRight && takeLeft)

{

this.eat();

choosen = random.nextInt(2);

if(choosen == 0)

{

left.drop();

takeLeft = false;

right.drop();

takeRight = false;

}

else

{

right.drop();

takeRight = false;

left.drop();

takeLeft = false;

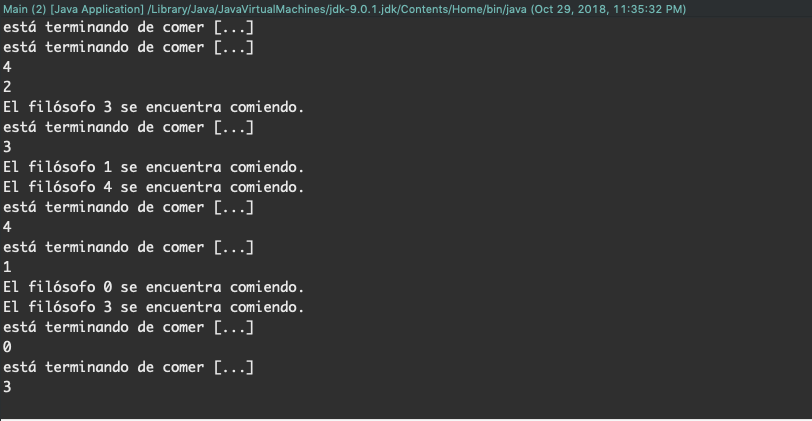
}

}

}

}

}



1. **Ejercicio 7.  Contesta el siguiente test**.

**1. Señala la respuesta correcta:**

a. Se tarda mucho menos tiempo en crear un nuevo hilo en un proceso existente que en crear un nuevo proceso.

b. Se tarda mucho menos tiempo en terminar un hilo que un proceso.

c. Se tarda mucho menos tiempo en conmutar entre hilos de un mismo proceso que entre procesos.

**d. Todas las respuestas son correctas**

**2. Señala la respuesta correcta:**

a. Un hilo dentro de un proceso se ejecuta secuencialmente.

b. Cada hilo tiene su propia pila y contador de programa.

c. Pueden crear sus propios hilos hijos.

**d. Todas las respuestas son correctas.**

**3.¿En qué se diferencian los hilos de los procesos?**

a. No comparten la CPU.

**b. Los hilos no son independientes entre sí, mientras que los procesos son independientes.**

c. Los procesos no son independientes entre sí, mientras que los hilos son independientes.

d. Los hilos no crean sus propios hijos mientras que los procesos sí.

**4. Los hilos no comparten:**

**a. Contador del programa.**

b. Instrucciones.

c. Variables globales.

d. Ficheros abiertos.

**5. Con la función fork():**

a. Creamos un hilo.

**b. Creamos un proceso.**

c. Ejecutamos un proceso.

d. Ejecutamos un hilo.

**6. ¿Qué es un proceso zombie?**

1. Aquel que está en ejecución.

b. Es lo mismo que un proceso huérfano.

**c. Un proceso que ha terminado pero que sus recursos no han sido liberados.**

d. Un proceso que hemos revivido.

**7. ¿Cuál es la sentencia para que el proceso padre espere a que finalice el hijo?**

**a. Wait(NULL)**

b. Wait(PID\_HIJO)

c. Pipe(NULL)

d. Pipe(PID\_HIJO);

**8. ¿Para qué utilizamos la función kill()?**

a. Para matar un proceso.

b. Para pausar un proceso.

c. Para esperar por un proceso.

**d. Para enviar una señal.**

**9. ¿Cuál de las siguientes sentencias utilizarías para que un proceso se suspenda 1 segundo?**

a. Sleep(1)

**b. Sleep(1000)**

c. Pause(void)

d. Sleep(void)

RESPUESTAS:

1 – D

2 – D

3 – B

4 – A

5 – B

6 – C

7 – A

8 – D

9 – B