1 Hilos

En este tema se va a estudiar la concurrencia mediante hilos. Debido a que comparten memoria, su programación es más rica y compleja que los procesos.

Sumario

2 Procesos vs Hilos

Los procesos se pueden definir como programas en ejecución. Supongamos que dentro de un mismo proceso se pudiesen ejecutar varias partes del código de forma concurrente. A cada una de esas ejecuciones se las denomina **hilo** de ejecución.

Importante: Dentro de un proceso se van a tener funcionando uno o varios hilos.

Para entender el concepto de hilo se puede hacer la siguiente analogía:

Supongamos que un proceso es una persona haciendo una tarea usando solo una mano. Se pueden tener varias personas, por ejemplo, encerando un coche. Cada persona puede primero dar la cera y después pulir la cera usando una sola mano. Pero una persona puede coger a la vez la cera con una mano y el trapo de pulir con la otra y..., mientras da la cera, puede pulirla. Evidentemente esta persona iría más rápido que el resto pues está usando las dos manos. Se podría decir que las personas son los procesos y sus manos son los hilos, por lo que un proceso puede hacer varias cosas a la vez. La ventaja de los procesos al usar los hilos es que pueden lanzar todos los hilos que

necesiten. Es como decir que una persona puede tener 2, 3, 4, 5, 6,... brazos sin ningún tipo de esfuerzo, sólo con desearlo.

Proceso con dos hilos

Proceso con 4 hilos

Proceso con muchos hilos

Cada persona tiene su propia memoria que no comparte con otras personas (habitualmente). Si una persona está pensando en un número, otra persona no puede saber qué número es. Un proceso tiene sus propias variables que no son accesibles desde otro proceso.

En el caso de los hilos, cada hilo tiene acceso a todas las variables y recursos del proceso. Si lo aplicamos al caso de las personas, cada brazo comparte el mismo cerebro, por lo tanto tiene acceso a la misma memoria.

Los procesos que se han hecho en el tema anterior sólo tienen un hilo funcionando, **el hilo principal**. En este tema a partir del hilo principal, se van a generar otros hilos.

Los hilos comparten *todos* los recursos del proceso en el que se ejecutan. Es decir, todos los hilos podrán acceder a las mismas variables, archivos, streams,...

Por ejemplo:

```
int n = 0;

....

// Hilo funcionando
n++;

...

// Otro hilo funcionando posteriormente
n++;

// n = 2
```

Como los hilos comparten la memoria habrá que tener muy claras las secciones críticas dentro del código, para prohibir, de alguna manera, que varios hilos las ejecuten a la vez.
Puede suceder que dos hilos accedan a la vez a la misma variable que está en la RAM y la copien a la caché del procesador. Cada hilo modificará el valor de la variable que se encuentra en la caché y se subirá un valor diferente a la RAM, y generalmente erróneo, que si cada hilo hubiese accedido primero uno y luego el otro. Se denominará **condición de carrera** a la salida errónea de un programa concurrente cuando los procesos o hilos no acceden a los recursos compartidos de la manera que el programador esperaba.

Cuando el sistema operativo tiene que desalojar un proceso que se está ejecutando en la CPU por otro, se produce lo que se denomina un **cambio de contexto**. El cambio de contexto es muy complejo de realizar, por lo que cambiar entre procesos en ejecución es mucho más lento que cambiar entre hilos, por ello el cambio entre hilos de ejecución o la creación de nuevos hilos es mucho más rápido que el cambio o creación de procesos.

Por otro lado los procesos suelen ser más seguros que los hilos. Por ejemplo, si se tiene un servidor de páginas web que lanza un proceso cada vez que sirve una página web con el fin de servir esa página y luego lo destruye (es un comportamiento muy común en servidores web), será más lento que un servidor que haga la misma tarea usando hilos. Ahora bien, en cuanto seguridad, si un se produce un fallo en un proceso, por lo general, sólo afectará a ese proceso y el servidor seguirá funcionando. Si el mismo fallo se produce en un hilo, este puede afectar a todo el servidor, pues todos los hilos comparten la misma memoria y recursos.

Curiosidad: El servidor web Apache tiene dos variantes Apache Worker y Apache Prefork. Apache Worker se basa en hilos y Apache Prefork de basa en procesos. El servidor Prefork se suele usar para ejecutar programas en el servidor, por ejemplo, PHP. El servidor Worker tiene más rendimiento y se suele usar para servir HTML estático. Para hacerse una idea, sirviendo contenido estático, un Prefork suele tardar de media unos 200 ms por petición. Worker tarda del orden de 4 ms por petición.

2.1 Sincronía y asincronía

En algunos lenguajes y entornos, como NodeJS, se tiene el concepto de sincronía y asincronía al realizar ciertas operaciones:

En una operación **síncrona** el proceso o hilo que invoca la operación se queda bloqueado hasta que la operación termina.

En una operación **asíncrona** el proceso o hilo que invoca la operación puede continuar realizando su ejecución hasta que la operación finalice. Cuando la operación finaliza, se notifica al hilo solicitante de la operación asíncrona el resultado de la operación, o el fin de la misma.

Por ejemplo, en NodeJS es posible leer los contenidos de un archivo de forma síncrona y asíncrona. Si se decide escribir el archivo de forma síncrona, el proceso se detendrá hasta que el archivo haya sido escrito completamente. Es una forma similar a como lo hace Java. Esta es una porción de código en NodeJS mostrando la escritua síncrona:

```
const fs = require('fs');
var texto="Este es el texto que se va a guardar en el archivo.";


try {
   fs.writeFileSync('salida2.txt', texto, 'utf-8');
   console.log("El archivo salida2.txt se ha guardado correctamente");
} catch(err) {
   console.log("Error al escribir el archivo salida2.txt");
}
```

Si en NodeJS se decide escribir un archivo de forma asíncrona, el proceso pasará una función que se ejecutará cuando el fin de la escritura del archivo se alcance y, el proceso, continuará su ejecución de forma habitual. Este es un ejemplo de código en NodeJS mostrando la escritura asíncrona:

```
const fs = require('fs');
```

```
var texto="Este es el texto que se va a guardar en el archivo.";
fs.writeFile('salida1.txt', texto, 'utf-8' ,
 // Método que se ejecutará cuando se finalice la escritura del archivo:
 (err) => {
   // Si hay un error en la lectura se lanza una excepción
   if (err) throw err;
   console.log("El archivo salida1.txt se ha guardado correctamente");
});
// El programa continua su ejecución
console.log('¿Fin de la escritura del archivo?');
```

Para implementar la asincronía se pueden usar hilos que hagan el trabajo o crear mecanismos que se encarguen de gestionar los eventos.

El uso adecuado de la sincronía y asincronía, puede aumentar mucho el rendimiento de un código.

2.2 Hilos de usuario e hilos al nivel del núcleo

En los **hilos a nivel de usuario**, la implementación de los hilos se hacen mediante bibliotecas que se ejecutan en el espacio del usuario. No es necesario hacer llamadas al sistema operativo para que lo hilos funcionen. Por ejemplo, algunas implementaciones del estándar POSIX de UNIX son de este tipo.

En los **hilos al nivel del núcleo** se necesitan llamadas al sistema operativo para que los hilos puedan funcionar. En sistemas operativos basados en Win32 (Windows) o el antiguo OS/2, la implementación de los hilos era de esta forma.

2.3 El planificador de hilos

En algunos sistemas operativos se define el concepto de procesador lógico, donde se ejecutan los hilos será en un procesador lógico. Se tendrán dos niveles el de los procesadores lógicos y el de los procesadores reales. Los procesadores lógicos son una abstracción de un procesador, no existen realmente, pero el hilo que se está ejecutando va a creer que es un procesador convencional. Se puede suponer que varios procesadores lógicos se van a ejecutar sobre varios procesadores reales:

Bajo este esquema se tienen varias configuraciones:

- Un hilo por procesador lógico. Es el caso representado en la situación 1.
- Muchos hilos por procesador lógico. Es el caso representado en la situación 2.
- Muchos hilos en muchos procesadores lógicos estricto. Es el caso representado en la situación 3. Varios hilos se van a ejecutar en varios procesadores lógicos compartiéndolos.
- Muchos hilos en muchos procesadores lógicos no estricto. Es el caso representado en la situación 4. Es similar al anterior, pero se puede asignar un único hilo a uno de los procesadores lógicos.

3 Hilos en Java

3.1 Creación de hilos en Java

En Java existen dos formas de crear hilos:

- Heredando de la clase Thread
- Implementando el método Runnable

3.1.1 Clase Thread

Para crear un hilo heredando de la clase Thread, se debe sobrescribir el método run(). Por ejemplo:

```
class Ej1 extends Thread   {
        private int n;
        private String nombre;

        public Ej1(String nombre) {
                n = 0;
                this.nombre = nombre;
        }

        public void run() {
                // El método run contiene el código que va a ejecutar el hilo
                for(int i = 0; i < 100; i++) {
                        n += i;
                        System.out.println(nombre + " " + n);
                }
        }

 public static void main(String args[]) {
   Ej1 e1 = new Ej1("Hilo 1");
   Ej1 e2 = new Ej1("Hilo 2");
   e1.start(); // El hilo se ejecuta con start()
   e2.start();
 }
}
```

En la salida se van mezclando los dos hilos de forma aleatoria:

```
Hilo 2 1326
Hilo 2 1378
Hilo 2 1431
Hilo 2 1485
Hilo 1 4753
Hilo 1 4851
Hilo 1 4950
Hilo 2 1540
Hilo 2 1596
Hilo 2 1653
Hilo 2 1711
```

3.1.2 Implementando el método Runnable

Java no soporta herencia múltiple, por lo que se dispone también de la interfaz Runnable para la creación de hilos. Al igual que con la clase Thread, hay que implementar el método run():

```
class Ej2 implements Runnable      {
        private int n;
```

```
        private String nombre;

        public Ej2(String nombre) {
                n = 0;
                this.nombre = nombre;
        }

        public void run() {
                for(int i = 0; i < 100; i++) {
                        n += i;
                        System.out.println(nombre + " " + n);
                }
        }

 public static void main(String args[]) {
  Ej2 e1 = new Ej2("Hilo 1");
  Ej2 e2 = new Ej2("Hilo 2");
  Thread h1 = new Thread(e1);
  Thread h2 = new Thread(e2);
  h1.start();
  h2.start();
 }
}
```

En este casa para ejecutar el hilo, después de implementar el método run(), se creará un objeto Thread al que se le pasará como argumento el objeto que implementa Runnable y se iniciará el hilo con el método start().

3.2 Esperando por un hilo

Hay situaciones en las que es necesario saber si un hilo continua ejecutándose o si ha terminado. Para ello son interesantes los siguientes métodos de la clase Thread:

- join()
- join(long milisegundos)
- join(long milisegundos, int nano segundos)

Los tres métodos bloquean al hilo actual hasta que el hilo sobre el que se haya invocado el método join haya terminado. Cuando se le pasa un tiempo como argumento, el hilo actual sólo se bloquea durante el tiempo indicado.

Con el método:

- isAlive()

Se puede comprobar si el hilo sigue en ejecución.

Por ejemplo:

```
class Ej6 implements Runnable      {
        private String nombre;
        private int inicio, fin;
        public int suma;

        public Ej6(String nombre, int inicio, int fin) {
```

```
                this.nombre = nombre;
                this.inicio = inicio;
                this.fin = fin;
                suma = 0;
        }

        public void run() {
                for(int i = inicio; i <= fin; i++) {
                        suma += i;
                }
        }

 public static void main(String args[]) {
        Contador contador = new Contador();
  Ej6 e1 = new Ej6("Hilo 1", 1, 50);
  Ej6 e2 = new Ej6("Hilo 2", 51, 100);
  Thread h1 = new Thread(e1);
  Thread h2 = new Thread(e2);

  h1.start();
  h2.start();

  try {
                h1.join(); // Se espera a que finalice el primer hilo
                if(h2.isAlive())
                        h2.join(); // Si el segundo hilo está vivo, se espera por él
        } catch(InterruptedException e) {
                System.err.println("Error " + e);
        }
        System.out.println("La suma de 1 a 100 es: " + (e1.suma + e2.suma));
 }
}
```

3.3 Grupos de hilos y pool de hilos

A través de la clase ThreadGroup se pueden crear grupos de hilos e incluso jerarquías de hilos. Esto permite gestionar de forma más ordenada los hilos, de forma que se puede, por ejemplo interrumpir la ejecución de todos los hilos del conjunto de hilos.

Un **pool de hilos** permite crear una cola de hilos que se van procesando conforme se van completando los anteriores. Se pueden asignar 100 trabajos a una cola de 10 de modo que cada vez que se libera uno va procesando el siguiente. Java proveé un mecanismo para implementar los pool de hilos pero, por su generalidad, en este texto se usarán los semáforos para realizar dicha función.

3.4 Estados de un hilo en Java

Los estados de un hilo en Java se pueden ver reflejados en el siguiente diagrama:

Existe un planificador que va tomando los hilos que se encuentran en el estado de “listo” y los ejecuta durante un cuanto de tiempo, cuando su tiempo en la CPU se termina lo devuelve al estado listo y va en busca de otro hilo.

Los hilos pueden salir del estado de ejecución por diversas causas. Pueden bloquearse por una operación de entrada / salida; por la ejecución del método sleep() que duerme un hilo durante el

tiempo que se le indique; por la ejecución del método wait(), en este caso sólo se despertará el hilo si otro hilo ejecuta el método notify() o nofifyAll(); por intentar entrar a un método synchronized.

También un hilo puede decidir volver de forma voluntaria su cuanto de ejecución y volver al estado "listo". Para ello debe ejecutar el método "yield()". Esto puede deberse, por ejemplo, a que un hilo está esperando un determinado evento y no se desea que se bloquee, se puede comprobar si está libre el recurso y si no lo está ejecutar yield() para volver al estado "listo" para que otro hilo gane la CPU. Cuando el hilo vuelva a la CPU, volverá a comprobar si tiene disponible el recurso y si no volverá a ejecutar yield(),…

3.5 Prioridad de los hilos en Java

Java está pensado para que los hilos tengan prioridades, de esta forma el planificador preferirá que los hilos que tengan más prioridad estén más tiempo en la CPU. Al se Java multiplataforma, puede ser que en una determinada plataforma esto no se pueda garantizar. En nuestro caso supondremos que nuestra plataforma puede ejecutar los hilos sin problemas.

Las prioridades de los hilos van de 1 (MIN_PRIORITY) a 10 (MAX_PRIORITY). La prioridad de un hilo es la misma que la del hilo que lo creó. Por defecto, todo hilo tiene una prioridad de 5 (NORM_PRIORITY).

Para cambiar la prioridad se usan los métodos setPriority(nueva_prioridad). Con getPriority() se puede consultar la prioridad de un hilo:

```
class Ej2 implements Runnable      {
            private int n;
            private String nombre;

            public Ej2(String nombre) {
                        n = 0;
                        this.nombre = nombre;
            }

            public void run() {
                        for(int i = 0; i < 100; i++) {
                                    n += i;
                                    System.out.println(nombre + " " + n);
                        }
            }

 public static void main(String args[]) {
   Ej2 e1 = new Ej2("Hilo 1");
   Ej2 e2 = new Ej2("Hilo 2");
   Thread h1 = new Thread(e1);
   Thread h2 = new Thread(e2);

   h1.start();
   h2.start();

   h1.setPriority(1);
   h2.setPriority(10);

   System.out.println("Hilo 1 prioridad " + h1.getPriority());
   System.out.println("Hilo 2 prioridad " + h2.getPriority());
 }
}
```

3.6 Hilos daemon

Un hilo daemon es un hilo que representa una tarea que se va a ejecutar de forma continua. La máquina virtual no se cerrará hasta que todos los hilos daemon hayan finalizado. Antes de lanzar un hilo daemon, se deberá marcar como tal con el método “setDaemon(true)”. Con el método “getDaemon()”, se puede comprobar si un hilo es daemon.

Los hilos daemon tienen la prioridad más baja. La cualidad de hilo daemon se hereda desde un hilo cuando éste crea otro hilo. No puede cambiarse después de haber iniciado un hilo.

La máquina virtual Java lanza también hilos daemon como el recolector de basura, que libera la memoria ocupada por los objetos que no tienen referencias.

3.7 Synchronized

Para marcar secciones críticas de código en Java y evitar que varios hilos puedan ejecutar dichos código a la vez, se marcarán los procesos como synchronized. Cuando varios hilos intentan ejecutar un método marcado como synchronized, se bloquean y sólo uno de ellos puede ejecutar el método. Cuando termina su ejecución, el planificador de Java despertará otro hilo, entre los hilos que han sido bloqueados, para ejecutar el método.

Por ejemplo, en el siguiente código la variable “n” es incrementada por 2 hilos a través del método “sumar()”:

```
class Ej4 implements Runnable      {
        private static int n = 0;
        private String nombre;

        public Ej4(String nombre) {
                this.nombre = nombre;
        }

        private synchronized void sumar() {
                n++;
        }

        public void run() {
                for(int i = 0; i < 100; i++) {
                        sumar();
                        System.out.println(nombre + " " + n);
                }
        }

 public static void main(String args[]) {
   Ej4 e1 = new Ej4("Hilo 1");
   Ej4 e2 = new Ej4("Hilo 2");
   Thread h1 = new Thread(e1);
   Thread h2 = new Thread(e2);

   h1.start();
   h2.start();
 }
}
```

Una posible salida de este código puede ser:

```
Hilo 1 1
Hilo 2 2
Hilo 2 4  ← Falta el 3
Hilo 2 5
Hilo 2 6
Hilo 2 7
Hilo 2 8
Hilo 2 9
Hilo 2 10
Hilo 2 11
Hilo 2 12
Hilo 2 13
Hilo 2 14
Hilo 2 15
Hilo 1 3  ← El "Hilo 1" fue el que sumó 3
Hilo 2 16          ← Falta el 17
Hilo 2 18
Hilo 2 19
Hilo 1 17          ← El "Hilo 1" fue el que sumó 17
```

En esta caso se puede ver que el "Hilo 2" hace la mayoría de las sumas, pero el "Hilo 1" de vez en cuando es capaz de hacer alguna de las sumas. También se puede ver que los resultados de la sumar 1 a n no salen ordenados al cambiar de hilo, esto se debe a la forma que ha tenido Java de planificar los hilos.

3.8 Bloques sincronizados

Hay otra forma de usar synchronized sin necesidad de definir un método, si no directamente dentro de cualquier código:

```
synchronized(objeto) {
...
}
```

Si se usa de esta forma cualquier hilo que trate de ejecutar esa sección de código tratará primero de bloquear el objeto que se le pasa como argumento a synchronized. Si no lo consigue, esperará a que el objeto sea liberado. Por ejemplo:

```
class Contador {
        public int n = 0;
}

class Ej5 implements Runnable      {
        private static int n = 0;
        private String nombre;
        private Contador contador;

        public Ej5(String nombre, Contador contador) {
                this.nombre = nombre;
                this.contador = contador;
        }

        public void run() {
                for(int i = 0; i < 100; i++) {
```

```
                        // Sección crítica
                        synchronized(contador) {
                                contador.n++;
                                System.out.println(nombre + " " + contador.n);
                        }
                }
        }

 public static void main(String args[]) {
        Contador contador = new Contador();
  Ej5 e1 = new Ej5("Hilo 1", contador);
  Ej5 e2 = new Ej5("Hilo 2", contador);
  Thread h1 = new Thread(e1);
  Thread h2 = new Thread(e2);

  h1.start();
  h2.start();
 }
}
```

En el ejemplo se puede ver que se crea un objeto que servirá para compartir información entre los hilos, el objeto “contador”.

Ahora la sección crítica se marcará con syncronized:

```
                        synchronized(contador) {
                                contador.n++;
                                System.out.println(nombre + " " + contador.n);
                        }
```

Cuando un hilo llegue a esta porción de código, tratará de bloquear el objeto contador. Si otro hilo ya lo tiene bloqueado, esperará hasta que el bloqueo se libere.

4 Comunicación y sincronización entre hilos

Los hilos y los procesos para comunicarse y sincronizarse pueden usar:

- Semáforos
- Monitores
- Paso de mensajes

4.1 Semáforos

Un semáforo se podría definir como un contador (que será un número entero que puede tener los valores 0, 1, 2, 3,...) en el que se pueden hacer 3 operaciones:

- Asignar un valor inicial (0, 1, 2, 3,…)
- Incrementar: Suma 1 al valor del contador
- Decrementar: Resta 1 al valor del contador. Si el contador vale 0, el proceso se bloquea a la espera de que el contador vuelva a ser mayor que cero.

Se puede hacer la siguiente analogía:

Supongamos que se tiene una local con aforo limitado. En la puerta de la tienda hay un vigilante, el semáforo, encarada de que no haya más personas, que serán los procesos, en el local que las que indica el aforo. Cuando el local se llena, no deja pasar a otra persona dentro.

Si alguien quiere entrar cuando el local está lleno, tendrá que hacer cola hasta que alguien salga y el vigilante, semáforo, le permita entrar.

Su funcionamiento va a ser el siguiente, se asigna un valor inicial al semáforo, que puede ser cero.

El proceso que quiera entrar, deberá decrementar el semáforo. Si el valor contador del semáforo es cero, deberá esperar. Si el valor del semáforo es mayor que 1, continuará su ejecución.

Cuando el proceso termine con la tarea y desee abandonar el semáforo, incrementará el contador del semáforo. Si hay algún otro proceso esperando, podrá usar el semáforo.

Como los procesos son los encargados de realizar las operaciones de incrementar y decrementar, un proceso podría incrementar el semáforo varias veces, sin decrementarlo, aumentando así el número de procesos que pueden usar a la vez el semáforo.

Si las operaciones de incrementar o decrementar no se usan correctamente, podría suceder que usen un recurso más de los procesos deseados o que se decremente el semáforo y que ningún proceso lo incremente provocando que los procesos que quieran usar el semáforo queden esperando eternamente.

Semáforo binario, es aquel en el que el contador sólo puede tomar los valores 0 ó 1.

Los sistemas operativos suelen implementar los semáforos binarios a través del bloqueo de archivos, de forma que, cuando un archivo es bloqueado por un proceso, sólo ese proceso puede usar el archivo. Si otro proceso trata de bloquear el archivo, suspenderá su ejecución hasta que el primer proceso libere el archivo.

En Java se implementa usando los métodos lock() y release() de la clase FileChannel:

```
File file = new File("/path/to/file");
FileChannel channel = new RandomAccessFile(file, "rw").getChannel();
FileLock lock = channel.lock();

// Operaciones con el recurso

lock.release();
System.out.println("released file\n");
```

4.1.1 La clase Semaphore

Java implemente los semáforos a través de la clase Semaphore. Esta clase tiene el siguiente constructor:

```
Semaphore sem = new Semaphore(int valor_inicial_contador);
```

Donde valor_inicial_contador, es el valor inicial que tendrá el contador del semáforo.

El semáforo se puede incrementar o decrementar usando los métodos:

```
sem.cdquire() // Decrementa el semáforo
...
sem.release() // Incrementa el semáfono
```

Los hilos que deseen usar el recurso que esté protegiendo el semáfono, deberán primero decrementar (acquire). Si el contador del semáforo es cero, el proceso quedará bloqueado.

Cuando el proceso debe de dejar de usar el recurso, deberá liberarlo con un incrementar (release). Si hay algún hilo a la espera de usar el recurso será despertado.

Si se usa el método acquire(), se deberá captura la excepción InterruptedException.

El siguiente ejemplo, muestra un ejemplo de cómo dos procesos pueden usar un semáforo binario para conseguir acceder a una variable n:

```
import java.util.concurrent.Semaphore;

class EjSemaforo implements Runnable {
	// Variable a la que los dos métodos desean entrar:
	public static int n;
	private String nombre;
	private Semaphore sem;

	public EjSemaforo(String nombre, Semaphore sem) {
		this.sem = sem;
		this.nombre = nombre;
	}

	public void run() {
		for(int i = 0; i < 100; i++) {
			try {
				// Se decrementa el semáforo
				sem.acquire();
					// Sección crítica
					n ++;
					System.out.println(nombre + " " + n);
				// Se incrementa el semáforo
				sem.release();
			} catch(InterruptedException e) {
				System.err.println(e);
			}
		}
	}

 public static void main(String args[]) {
	// Se crea un semáforo binario
	Semaphore sem = new Semaphore(1);

  EjSemaforo e1 = new EjSemaforo("Hilo 1", sem);
  EjSemaforo e2 = new EjSemaforo("Hilo 2", sem);
  Thread h1 = new Thread(e1);
  Thread h2 = new Thread(e2);

  h1.start();
  h2.start();
 }
}
```

Se deja como ejercicio probar a comentar la línea que incrementa el semáforo para ver como ambos procesos de bloquean.

El hilo que decrementa el semáforo no tiene que coincidir con el que lo incrementa y viceversa. Por ejemplo, un hilo puede decrementar el semáforo y otro incrementarlo:

```
import java.util.concurrent.Semaphore;

class EjSemaforo implements Runnable {
	// Variable a la que los dos métodos desean entrar:
	public static int n;
	private String nombre;
	private Semaphore sem;

	public EjSemaforo(String nombre, Semaphore sem) {
		this.sem = sem;
		this.nombre = nombre;
	}

	public void run() {
		for(int i = 0; i < 100; i++) {
			try {
				// Se decrementa el semáforo
				sem.acquire();
					// Sección crítica
					n ++;
					System.out.println(nombre + " " + n);
				// No se incrementa el semáforo
			} catch(InterruptedException e) {
				System.err.println(e);
			}
		}
	}

 public static void main(String args[]) {
	// Se crea un semáforo
	Semaphore sem = new Semaphore(2);

  EjSemaforo e1 = new EjSemaforo("Hilo 1", sem);
  EjSemaforo e2 = new EjSemaforo("Hilo 2", sem);
  Thread h1 = new Thread(e1);
  Thread h2 = new Thread(e2);
	try {
		sem.acquire();
		sem.acquire();
		h1.start();
		h2.start();
		System.out.println("Se interrumpen los hilos");
		Thread.sleep(1000);
		while(e1.n < 200) {
			sem.release();
			sem.release();
			sem.release();
			sem.release();
			System.out.println("Permisos del semáforo " + sem.availablePermits());
			Thread.sleep(100);
		}
	} catch(InterruptedException e) {
		System.err.println(e);
```

```
        }
    }
}
```

En el ejemplo se puede ver también que aunque el número de permisos del semáforo es 2 al inicio, se puede incrementar tanto como se desee usando release().

Por ejemplo:

```
Semaphore sem = new Semaphore(2);
…
sem.release();
sem.release();
sem.release();
sem.release();
```

¿Cuántos hilos podrían ejecutar un "sem.acquire()" sin bloquearse? Inicialmente el contador del semáforo esta inicializado a 2, pero después se incrementa el semáforo 4 veces más, por lo que, finalmente 2+4 = 6 hilos podrán decrementar el semáforo sin bloquearse.

Por lo tanto, el contador de un semáforo puede alcanzar valores superiores al que tuviese cuando se inicializa el semáforo.

4.1.2 El problema de los semáforos

El problema que presentan los semáforos es que deben ser los propios hilos los que soliciten decrementar e incrementar el semáforo. En una aplicación grande con un gran número de hilos de diferentes clase es fácil que se produzca un error y algún hilo use el recurso sin haber usado antes el semáforo.

4.2 Monitores

Un **monitor** es una abstracción en la cual se tienen una serie de recursos privados que pertenecen al monitor, una serie de métodos para acceder a dichos recursos y sólo un hilo podrá ejecutar alguno de dichos métodos cada vez.

Es decir, si en Java se quisiera implementar un monitor, sería una clase en la que los recursos son miembros private de la clase. Dispondría de una serie de métodos para acceder a dichos recursos, pero si dos hilos tratasen de acceder a alguno de esos métodos, uno de los se bloquearía hasta que el primero hubiese terminado.

Para implementar el monitor, se puede usa un semáforo binario. La implementación sería similar a la siguiente:

```
class Monitor {
        // Se definen los recursos
        private Tipo1 recurso1;
        private Tipo2 recurso2;
        ...

        // Semáforo binario para controlar el acceso
        private Semaphore sem = new Semaphore(1);
        ...

        // Métodos para acceder a los recursos
        public ... metodo1(Argumentos) {
                sem.acquire();
```

```
                ...
                // Se accede a los recursos
                ...
                sem.release();
        }

        public ... metodo2(Argumentos) {
                sem.acquire();
                ...
                // Se accede a los recursos
                ...
                sem.release();
        }
}
```

Como se puede ver:

- Los recursos se definen dentro de la clase como private
- Se tiene un semáforo binario para controlar el acceso a los métodos
- Dentro de cada método primero se decrementa el semáforo y después se incrementa

Gracias a este semáforo sólo un hilo podrá ejecutar cualquiera de los métodos. El resto de hilos deberá esperar a que el primer hilo termine para poder usar los recursos.

4.3 Paso de mensajes

Hay una forma de comunicación entre procesos que se puede usar tanto en sistemas fuertemente acoplados como débilmente acoplados, el **paso de mensajes**: los sistemas pueden intercambiar mensajes mediante primitivas básicas como **enviar** y **recibir**.

4.3.1 Clasificación de los tipos de paso de mensajes

El paso de mensajes se puede clasificar según:

a) Identificación en el proceso de comunicación

b) Sincronización

c) Características del canal

Según la **identificación en el proceso de comunicación**, el paso de mensajes se pueden clasificar:

a) Comunicación directa

b) Comunicación indirecta

Cuando se tiene **comunicación directa**, el emisor identifica claramente al receptor del mensaje. Por ejemplo, se envía un mensaje al proceso con PID 1234.

Las primitivas para comunicaciones serán del tipo:

```
enviar(ProcesoReceptor, mensaje)
```

Donde ProcesoReceptor es el proceso que va a recibir al mensaje.

```
recibir(ProcesoEmisor, mensaje)
```

El ProcesoEmisor sirve para que el proceso que recibe el mensaje pueda identificar al emisor.

Cuando se tiene **comunicación indirecta**, no se indica claramente a los procesos emisor y receptor. Los mensajes se envían colocándolos en un depósito intermedio llamado **buzón**. Los buzones pueden ser usados por varios procesos para enviar o recibir mensajes. Esto permite comunicación uno a uno, uno a muchos, muchos a uno o muchos a muchos. En las comunicaciones muchos a uno, los buzones se suelen llamar **puertos**.

Las primitivas de comunicación serán del tipo:

```
enviar(buzón, mensaje)
```

Donde buzón es el buzón que va a recibir al mensaje.

```
recibir(buzón, mensaje)
```

Según la **sincronización**, el paso de mensajes se puede clasificar en:

a) Comunicación síncrona

b) Comunicación asíncrona

En la **comunicación síncrona** el mensaje es bloqueante, es decir, el emisor o el receptor quedan bloqueados cuando invocan las primitivas de enviar o recibir, hasta que el mensaje sea recibido o enviado, respectivamente.

En la **comunicación asíncrona** el mensaje no es bloqueante, es decir, ni el emisor ni el receptor quedan bloqueados cuando se envía a recibe el mensaje.

Se podría comparar la comunicación síncrona con una llamada telefónica, en la que emisor y receptor quedan bloqueados mientras dura la llamada.

La comunicación asíncrona se puede comparar con el envío de una carta por correo. Ni el emisor ni el receptor quedan bloqueados mientras se produce la trasferencia del mensaje.

Según las **características del canal**, el paso de mensajes se puede clasificar en:

1. Flujo de datos. Según el flujo de datos la comunicación puede ser unidireccional (el canal sólo se puede usar para enviar o recibir) y bidireccional (el canal se puede usar para enviar y recibir a la vez).
2. Capacidad del canal. Este concepto hace referencia al tamaño del buffer que se usar para enviar y recibir el mensaje. Si el **canal es de capacidad cero**, el buffer tiene un tamaño 0 y la comunicación es bloqueante. Según el tamaño del buffer sea finita o infinita, también existen **canales de capacidad finita e infinita**.
3. Tamaño del mensaje. Los mensajes pueden tener un tamaño fijo o variable.
4. Canales con tipo o sin tipo. Algunos canales exigen definir algún tipo de datos en concreto para poder realizar la comunicación, por ejemplo, sólo se pueden enviar datos tipo String.
5. Paso por valor o por referencia. En algunos casos sólo se pueden pasar por el canal copias del mensaje, a esto se le llama **paso por valor**. En otros casos se le puede pasar al receptor del mensaje la dirección de memoria donde se encuentra el mensaje, **paso por referencia**.

4.3.2 Invocación remota y llamada a procedimiento remoto RPC

En los contenidos vistos hasta ahora sobre el paso de mensajes son bastante básicos. Al igual que los semáforos de pueden mejorar mediante el uso de monitores, el paso de mensajes se puede mejorar mediante la invocación remota. En el paso de mensajes simplemente se envía información de un proceso a otro o de un hilo a otro.

La **invocación remota** consiste en invocar una función de otro proceso o hilo distinto al actual. El proceso que invoca la función queda bloqueado en espera de la respuesta.

La invocación remota es muy útil para crear aplicaciones cliente servidor.

Por ejemplo, supongamos que se tiene una aplicación Java que desea hacer una consulta a una base de datos instalada en el mismo ordenador en la que se está ejecutando. Si la base de datos posee una API en Java, se podrán hacer llamadas desde nuestra aplicación a la base de datos para realizar búsquedas. Nuestro proceso se quedará esperando hasta que la base de datos devuelva el resultado de la consulta.

En la **llamada a procedimiento remoto** (RPC, Remote Procedure Call) se hace una llamada a un procedimiento de un proceso que estará en otro equipo.

Este mecanismo también es muy propicio para realizar aplicaciones cliente-servidor.

En teoría la llamada podría ser asíncrona de forma que el proceso cliente pueda estar realizando otras tareas hasta que reciba la respuesta.

Por ejemplo, supongamos que se quiere realizar una consulta a una base de datos que no se encuentra en nuestro equipo, si no en un equipo remoto.

Para realizar la llamada a procedimiento remoto, se disponen actualmente de las siguientes herramientas:

- RMI (Remote Method Invocation) de Java: Es un mecanismo que va a permitir invocar métodos que no se encuentran el la misma máquina virtual que la que está ejecutando el programa. Sólo funciona para Java.
- CORBA (Common Object Request Broker Architecture): Es un protocolo que permite la llamada a procedimiento remoto desde diversos lenguajes de programación.
- DCOM (Distributed Component Object Model): Similar a CORBA pero es una tecnología propiedad de Microsoft.

Estos protocolos se basan en paso de mensajes en binario, usando sus propios protocolos. Actualmente el uso del protocolo HTTP está muy extendido, de hecho en algunas redes en las que hay proxies y firewalls, puede que sea el único protocolo disponible. Por ello son interesantes los protocolos:

- SOAP (Simple Object Access Protocol): Se usa XML para pasar mensajes y realizar la llamada a procedimiento remoto. Los mensajes se pueden enviar por HTTP, FTP, TCP o incluso por correo electrónico.

- gRPC: Sistema de RPC desarrollado por Google que permite la llamada a procedimiento remoto usando diversos lenguajes y plataformas. Se usa HTTP/2 para el envío de la información.