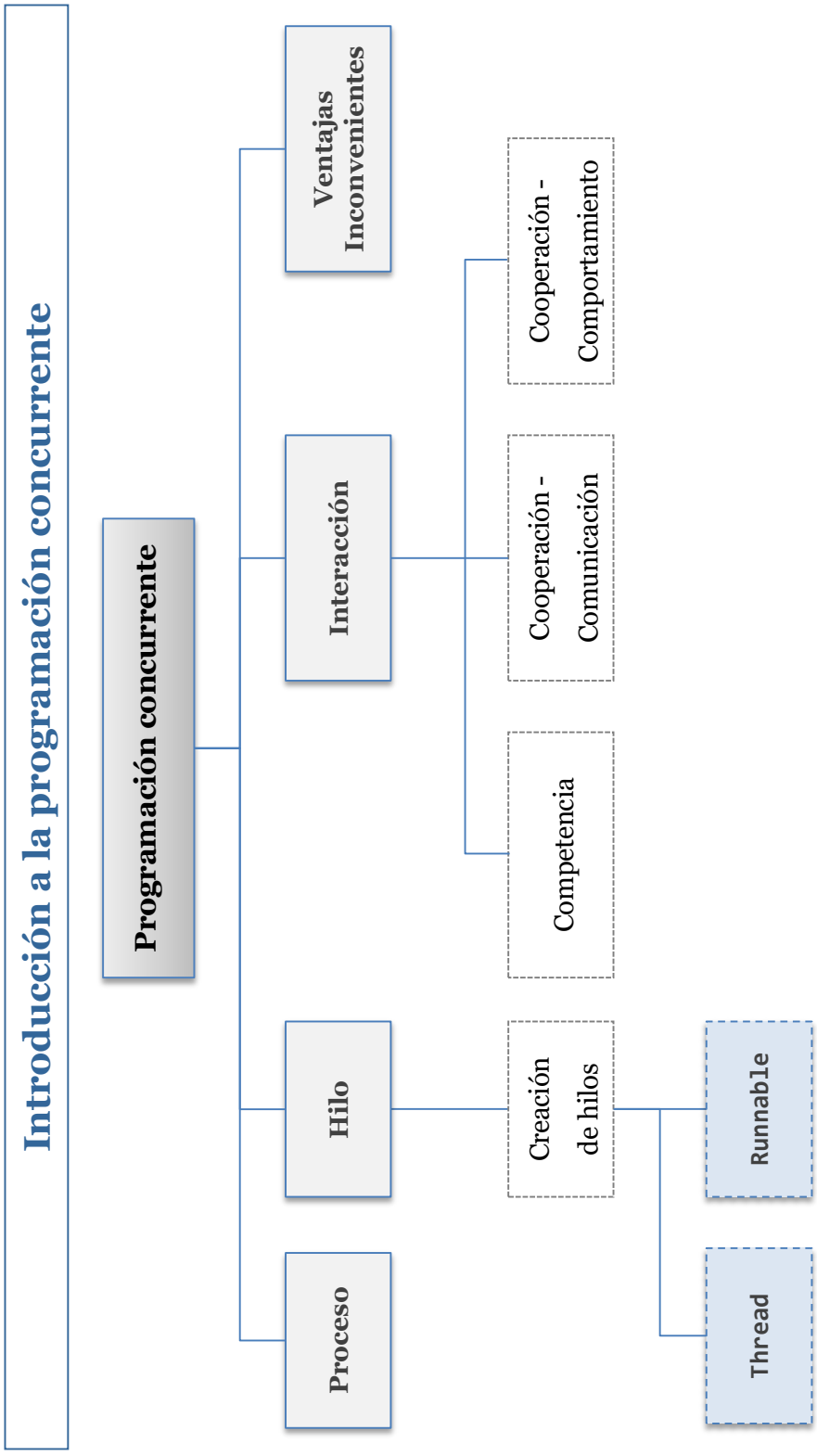
Introducción a la programación concurrente

TEMA 7

Esquema



Ideas clave

7.1. ¿Cómo estudiar este tema?

Para estudiar este tema lee el **capítulo 1 (páginas 1-8 y 14-18)** del siguiente libro, disponible en la Biblioteca Virtual de UNIR:

Espinosa, A. R., Argente, E. & Muñoz, F. D. (2013). *Concurrencia y sistemas distribuidos*. Valencia: Editorial de la Universidad Politécnica de Valencia.

Para comprobar si has comprendido los conceptos puedes realizar el test de autoevaluación del tema.

En este tema vamos a aprender los conceptos básicos de la programación concurrente:

- Diferencia entre hilo y proceso.
- Modos de interacción entre procesos o hilos.
- Creación de hilos en Java.
- Ventajas e inconvenientes de la programación orientada a objetos.

7.2. Introducción programación concurrente

Programación concurrente

Es la **simultaneidad en la ejecución de múltiples tareas interactivas**. Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa. Las tareas se pueden ejecutar en una sola unidad central de proceso (multiprogramación), en varios procesadores o en una red de computadores distribuidos. Fuente: Wikipedia.

7.3. El concepto de proceso e hilo

Proceso

Instancia de un programa ejecutándose en un ordenador.

Hilo

Flujo de control secuencial dentro de un programa.

Un único hilo es similar a un **programa secuencial**, es decir, tiene un comienzo, una secuencia y un final, además en cualquier momento durante la ejecución existe un sólo punto de ejecución. Sin embargo, un hilo no es un programa; no puede ejecutarse por sí mismo, se ejecuta dentro de un programa.

La máquina virtual de Java permite que una aplicación tenga **múltiples hilos** de ejecución ejecutándose concurrentemente.

Hay que diferenciar entre:

Concurrencia	Paralelismo
Ejecución excluyente en tiempo con un procesador. Sujeto a mecanismo de planificación con alternancia.	Ejecución simultánea en tiempo. En sistemas multiprocesador.

7.4. Interacción entre procesos/hilos

Los modos en que los procesos o hilos interactúan entre sí, se pueden clasificar en función del conocimiento que cada proceso o hilo tiene de los demás:

- **No tienen ningún conocimiento de los demás**: procesos/hilos independientes. Compiten por los recursos.
- **Tienen un conocimiento indirecto**: comparten recursos. Cooperan para compartir los recursos.
- **Tienen un conocimiento directo**: se comunican unos con otros. Cooperan para compartir los recursos.

Competencia por los recursos

Los hilos o procesos compiten por el uso de un recurso, sin tener conocimiento del resto.

Se deben solucionar tres problemas:

Exclusión mutua	Interbloqueo	Inanición
Solo uno puede acceder a la sección crítica que accede al recurso.	Bloqueo permanente de un conjunto de procesos o hilos por el acceso a recursos compartidos.	Negación indefinida a un proceso o hilo del acceso a un recurso.

Cooperación por compartimiento

Los hilos o procesos interactúan entre sí, pero no tienen un conocimiento explícito del resto. **Cooperan** para asegurar que los datos compartidos son correctos.

En la cooperación por compartimiento también **se deben resolver los problemas** de exclusión mutua, interbloqueo e inanición.

Cooperación por comunicación

Los hilos o procesos tienen un conocimiento explícito del resto y se comunican entre sí.

En la cooperación por comunicación no es necesario controlar la exclusión mutua pero se deben resolver los problemas de interbloqueo e inanición.

7.5. Los hilos en Java

Java proporciona dos métodos para la creación de hilos:

Herencia de la clase `Thread`	Implementación de la interfaz `Runnable`

Herencia de la clase `Thread`

El cuerpo del hilo va dentro del método `run()` por lo que es necesario realizar la redefinición de este método.

```
class Hilo extends Thread{
	//...
	public void run(){...}
}
```

- Para crear un hilo:

```
Thread miHilo= new Hilo();
```

- Para iniciar la ejecución del hilo:

```
miHilo.start();
```

Ejemplo:

```
class Hilo extends Thread{
	public void run(){
		for (int cont = 0; cont < 10; cont++){
			System.out.println("Contador: " + cont++);
		}
	}
}
class PruebaHilo{
	public static void main(String args[]){
		Thread hilo = new Hilo();
		hilo.start();
	}
}
```

La clase `Thread` tiene un atributo nombre, que puede ser inicializado al construir el hilo y consultado posteriormente:

- `Thread(String name)`
- `String getName()`

Ejemplo:

```
public class PruebaHilo{
	public static void main(String[] args){
		new Caballo("Novato").start();
		new Caballo("Imperioso").start();
		new Caballo("Avispado").start();
		new Caballo("Veloz").start();
		new Caballo("Cansino").start();
	}
}
class Caballo extends Thread{
	Caballo(String nombre){
		super(nombre); //nombre para el constructor de Thread
	}
	public void run(){
		for (int i = 0; i < 10; i++){
			System.out.println(i + " " + getName());
			try{
				sleep((long) (Math.random() * 1000));
			}catch (InterruptedException e){...}
		}
		System.out.println("Llega a Meta " + getName() + "!");
	}
}
```

Implementación de la interfaz `Runnable`

La interfaz `Runnable` tiene la siguiente estructura:

```
public interface Runnable{
	public void run();
}
```

Los pasos para la creación de hilos usando `Runnable` son:

1. Implementar interfaz en una clase
2. Crear objeto ejecutable
3. Pasarlo al constructor de `Thread`

Los constructores de `Thread` para objetos 'ejecutables' son los siguientes:

`Thread(Runnable target)`	`Thread(Runnable target, String name)`

```
class Hilo implements Runnable{
	public void run(){
		for (int cont=0; cont<10; cont++){
			System.out.println("Contador: " + cont++);
		}
	}
}
class PruebaHilo{
	public static void main(String args[]){
		Thread hilo = new Thread(new Hilo());
		hilo.start();
	}
}
```

7.6. Ventajas e inconvenientes de la programación concurrente

Ventajas

- **Eficiencia**: aplicaciones más rápidas.
- **Escalabilidad**: facilita la distribución de los componentes de una aplicación en diferentes ordenadores.
- **Gestión de comunicaciones**: uso eficiente de recursos de comunicación.
- **Flexibilidad**: facilita la modificación de las aplicaciones ante un cambio en los requisitos.
- **Menor hueco semántico**: facilita la coordinación de diferentes funcionalidades en una aplicación.

Inconvenientes

- **Programación delicada**: los problemas que pueden surgir son «condición de carrera» e «interbloqueos».
- **Depuración compleja**: un mismo escenario puede producir distintos resultados.

7.7. Computación de alto rendimiento

En la computación de alto rendimiento a nivel de *software* nos interesa mejorar diferentes aspectos del código que nos permitan obtener un **resultado óptimo** con respecto a los objetivos iniciales que el programador ha impuesto en el *software*.

De alguna manera, cuando queremos hacer un programa, la primera tendencia que queremos es que el «programa funcione», es decir, como si fuese una caja negra donde no nos importa cómo funcione el algoritmo, porque lo único que nos interesa es la salida del mismo.

Esta filosofía a veces conlleva a que tengamos partes del código con **bajo rendimiento o cuellos de botella**, que harán que el sistema rinda menos. Estos cuellos de botella o bajos rendimientos pueden detectarse si medimos el tiempo en las partes del código que queremos y de allí analizar alguna forma alternativa para que podamos aumentar el rendimiento del sistema.

Por otro lado, hay diferentes aplicaciones donde el cómputo de las operaciones tiene que ser el adecuado para poder predecir eventos, como usar modelos computacionales para predecir el tiempo o la trayectoria de un cometa.

Rendimiento en función del tiempo

En cualquier algoritmo el **tiempo de ejecución** será el tiempo que el algoritmo usará **desde que inicia hasta que termina**. Este tiempo de ejecución se verá afectado por partes del código que consuman más o menos porcentaje del tiempo total de ejecución.

Por ejemplo, en el siguiente código, se puede usar la clase `Tiempo` en Java para medir el tiempo en los dos bucles, habrá uno que use más tiempo que otro, dependiendo de qué operaciones realicen en cada bucle `for`. Una forma de iniciar un temporizador es haciendo uso de la función `System.currentTimeMillis()`.

En el ejemplo se puede observar que en el segundo bucle `for` están comentados las instrucciones `System.out.println("El resultado 1 es hip1")` y `System.out.println("El resultado 1 es hip2").`

Si activamos estas instrucciones veremos cómo el rendimiento del sistema estará afectado por consumir más tiempo al imprimir cada resultado en el ciclo `for`. Si eliminamos el comando de imprimir veremos cómo el rendimiento en función de tiempo aumentará.

```
package time;
public class Tiempo {
        public static void main(String[] args) {
             long Tiempoinicio = System.currentTimeMillis();
             long contador = 0;
             for (int k = 0; k < 8000000; k++) {
                contador = k;
             }

             long Tiempoparada = System.currentTimeMillis();
             long Tiempoinicio2 = System.currentTimeMillis();
             long contador2 = 0;
             for (int k2 = 0; k2 < 200000000; k2++) {
                contador2 = k2;
                if(!(contador==7999999) || !(contador2<200000000))
                {
               // System.out.println("El resultado 1 es hip1");
                }
                else
                {
               // System.out.println("El resultado 1 es hip2");
                }
             }
             long Tiempoparada2 = System.currentTimeMillis();
                   long Tiempotranscurrido = Tiempoparada - Tiempoinicio;
             long Tiempotranscurrido2 = Tiempoparada2 - Tiempoinicio2;
                   System.out.println("El contador 1 es " +contador);
             System.out.println("El tiempo transcurrido es "
+Tiempotranscurrido);
             System.out.println("El contador 2 es " +contador2);
             System.out.println("El tiempo transcurrido es "
+Tiempotranscurrido2);
          }

}
```


En este caso, se tendría que plantear si es necesario o no que se imprima en cada iteración un mensaje o si un reporte de mensajes es suficiente. Dependiendo de la aplicación se tendrá que analizar cuál es la mejor manera para poder aumentar el rendimiento del código en función del tiempo de ejecución.

Otro aspecto importante es que, a la hora de analizar el rendimiento de un algoritmo, debemos tener en cuenta que **el análisis se debe de ser realizado en la misma arquitectura del sistema**. Por ejemplo, no es lo mismo realizar una instrucción en una GPU, o realizar instrucciones de tipo RISC (del inglés Reduced Instruction Set Computing) o CISC (del inglés Complex Instruction Set Computing).

Es decir, que para mejorar el código de un algoritmo es necesario que se aumente el rendimiento en la misma arquitectura.

Una manera de aumentar el rendimiento de un algoritmo consistirá en identificar las partes del código que tienen bajo rendimiento, tal cual analizamos en el ejemplo anterior al identificar el «cuello de botella» que puede generar el imprimir mensajes en cada iteración del bucle `for`.

Para aumentar la velocidad de ejecución se puede usar la **Ley de Amdhal**, que establece que el rendimiento del sistema puede mejorar si consideramos que la modificación de uno de sus componentes está limitada por la facción de tiempo que utiliza dicho componente.

La Ley de Amdahl es la siguiente:

$$T_{em} = T_{an} \cdot \left((1 - F_{ms}) + \frac{F_{ms}}{A_{ms}} \right)$$

Donde,

F_{ms} es la Fracción de tiempo que el sistema utilizará para que el subsistema sea mejorado.
A_{ms} es el Factor de mejora para el subsistema que se quiere mejorar.
T_{an} es el Tiempo de ejecución antiguo.
T_{em} es el Tiempo de ejecución mejorado.

El incremento de la velocidad (IV) está relacionado con los tiempos de ejecución antiguo y mejorado, es decir, IV = T_{an}/T_{em}. Usando esta expresión, la ley de Amdahl se puede reescribir de la siguiente manera:

$$I_v = \frac{1}{(1 - F_{ms}) + \frac{F_{ms}}{A_{ms}}}$$

Por ejemplo, se requiere reducir el tiempo de ejecución de un programa informático a la mitad de tiempo (A_{ms}=*2*), considerando que uno de sus componentes utiliza 20% (F_{ms}=*0,2*) de tiempo de la ejecución total.

Usando la ley de Amdahl establece que el incremento de velocidad (I_V) o la aceleración general de la aplicación de la mejora será:

$$I_v = \frac{1}{(1 - .2) + \frac{.2}{2}} \cong 1.11$$

El incremento de velocidad por medio de la ley de Amdahl no tiene unidades y con ello podremos saber cuánto podemos aumentar la velocidad si incrementamos la velocidad de un elemento.

Cabe aquí explicar que a partir de valores menores de 20% el incremento de velocidad es muy poco y conviene tener incrementos mayores del 20% para poder aumentar más el rendimiento del sistema.

Rendimiento en función de cambiar el algoritmo

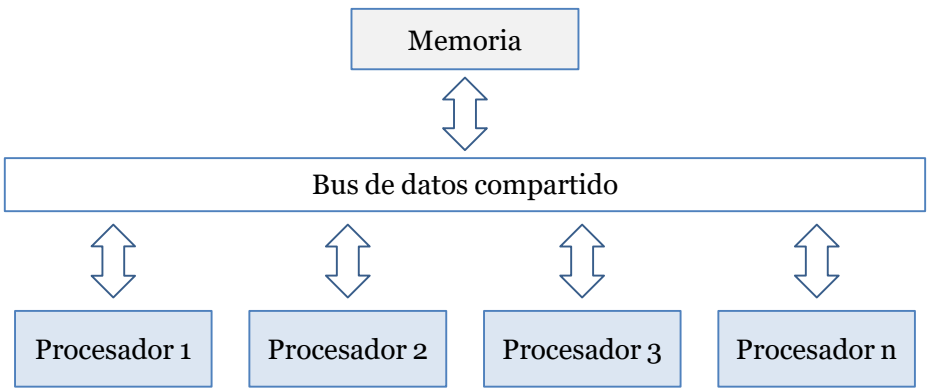
Otra forma de mejorar el rendimiento es **remplazando funciones que hagan lo mismo por funciones que requieran menos tiempo** y, por consiguiente, sean más rápidos. Por ejemplo, si se desea buscar un elemento en un vector ordenado, puede usarse una búsqueda secuencial que es fácil de implementar y encontrará el vector buscado. Pero cuando se requiere mayor rendimiento, se podría sustituir la búsqueda secuencial por una búsqueda binaria que es más compleja pero más rápida al buscar vectores grandes.

El siguiente código muestra cómo hacer una búsqueda de un arreglo ordenado:

```
package Mibusquedabinaria;
public class Busqueda_Binaria {
	public static void main(String[] args){
		int arreglo[]={1,2,3,4,5,6,6,8,9,10,1,12,13,14,15};
	int posicioninicial=0;
		int posicionfinal=arreglo.length-1;
			int numero_buscado= 3;
				int posicioncentro;
while (posicioninicial <=posicionfinal){
posicioncentro =(posicionfinal+posicioninicial)/2;
if(arreglo[posicioncentro]==numero_buscadSystem.out.println("El numero
buscado se encuentra en la posición: "+posicioninicial+ "y" +posicioncentro);
					break;
					}else if (numero_buscado <
arreglo[posicioncentro]){
						posicionfinal=posicioncentro - 1;
					}else{
						posicioninicial =posicioncentro + 1;
					}
			}

	}
}
```

Por último, otra forma de aumentar el rendimiento de un algoritmo es utilizando código multihilo en sistemas que dispongan de multiprocesadores o multicore (como se muestra en la siguiente figura). También recurriendo a cómputo paralelo en donde se aumentará la velocidad de proceso al procesar varios subprocesos de manera paralela.



Lo + recomendado

No dejes de leer…

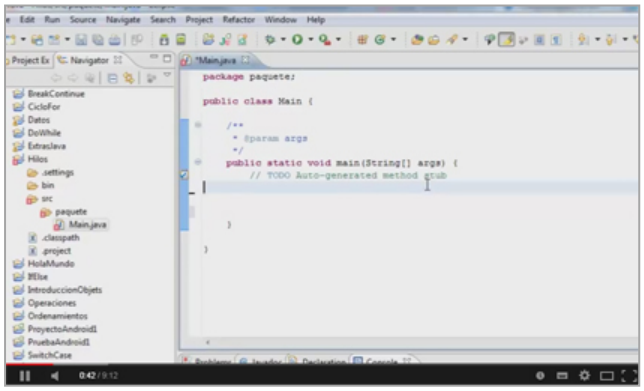
Concurrencia

Consulta la página de Oracle en la que podrás encontrar información detallada sobre concurrencia.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web:
http://docs.oracle.com/javase/tutorial/essential/concurrency/

No dejes de ver…

Hilos/`Threads` en Java

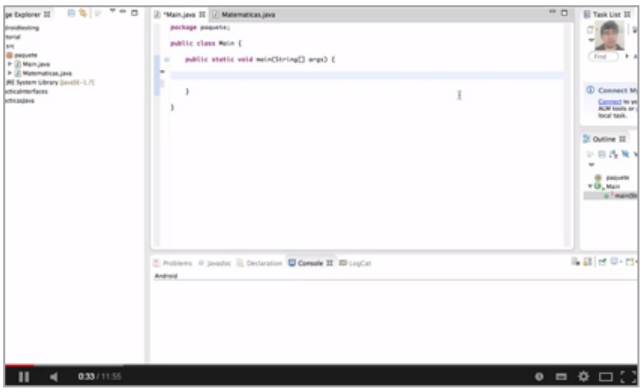


El siguiente vídeo explica los hilos en Java.

Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
https://www.youtube.com/watch?v=ZoCk8w86JlU

Java-`Threads` (hilos) implementados con `Runnable`

El siguiente vídeo explica la implementación de hilos con `Runnable`.



Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
https://www.youtube.com/watch?v=nE4Cdt5MebA

+ Información

A fondo

Programación concurrente

Palma, J. T. (2003). *Programación concurrente*. Madrid: Thomson.

Libro en el que se hace un seguimiento de SWING en profundidad.

Accede a una parte del libro desde el aula virtual o a través de la siguiente dirección web:
http://books.google.es/books?id=LGMZodtyKXMC&printsec=frontcover

Bibliografía

Oakx, S. & Wong, H. (2004). *Java Threads*. California: Ed. O'Reilly.

Test

1. Indica cuál de las siguientes afirmaciones es correcta:

A. Un hilo es un programa.

B. Un hilo se ejecuta dentro de un programa.

C. Un hilo se puede ejecutar por sí mismo.

D. La máquina virtual de Java no permite múltiples hilos de ejecución.

2. Para iniciar la ejecución de un hilo se utiliza el método:

A. `run()`

B. `new()`

C. `start()`

D. `runnable()`

3. En la creación de hilos mediante la interfaz `Runnable` es necesario:

A. Implementar la interfaz en una clase.

B. Crear una clase que herede de la clase `Thread()`.

C. Instanciar la interfaz `Runnable`.

D. Implementar el método `start()`.

4. Indica cuál de las afirmaciones no es correcta:

A. Al constructor de la clase `Thread` siempre hay que pasarle un argumento.

B. Al crear un objeto `Thread` podemos indicar el nombre del hilo.

C. Al crear un objeto `Thread` podemos indicar la interfaz `Runnable`.

D. Para crear un objeto `Thread` debemos redefinir el método `start()`.

5. A una sección crítica podrá acceder:

A. Todos los procesos o hilos que compartan el recurso crítico.

B. Solo un proceso o hilo.

C. Cualquier proceso o hilo.

D. Solo un proceso que tenga conocimiento explícito del resto.

6. Considera el siguiente escenario: dos hilos H1 y H2, y dos recursos R1 y R2. Cada hilo necesita acceder simultáneamente a los dos recursos. El hilo H1 posee el recurso P1 y el hilo H2 posee el recurso R2. Indica qué problema se produce:

A. Inanición.

B. Exclusión mutua.

C. Interbloqueo.

D. Ninguno.

7. Considera el siguiente escenario: tres hilos H1, H2 y H3, que acceden al recurso compartido R1. Si el recurso R1 se concede a los hilos H1 y H2 alternativamente y se niega el acceso indefinidamente al hilo H3, ¿qué problema se produce?

A. Inanición.

B. Exclusión mutua.

C. Interbloqueo.

D. Ninguno.

8. Indica cuál de los siguientes problemas no es necesario controlar cuando los procesos o hilos cooperan por comunicación:

A. Inanición.

B. Exclusión mutua.

C. Interbloqueo.

D. Paralelismo.

9. Indica cuál de las afirmaciones es correcta:

A. Los hilos compiten por los recursos cuando no tienen conocimiento del resto de hilos.

B. Los hilos cooperan para compartir recursos cuando tienen un conocimiento indirecto del resto.

C. Los hilos cooperan para compartir recursos cuanto tienen un conocimiento directo del resto.

D. Todas las anteriores son correctas.

10. El paralelismo es:

A. La ejecución excluyente con un solo procesador.

B. La ejecución simultánea con varios procesadores.

C. La ejecución simultánea con un solo procesador.

D. La ejecución excluyente en tiempo.

