code
URJC

Programación Concurrente

Tema 2

Memoria Compartida

Micael Gallego
micael.gallego@urjc.es
@micael_gallego

Universidad Rey Juan Carlos

Memoria Compartida

- **SimpleConcurrent**
- Intercalación de instrucciones: Indeterminismo
- Variables Compartidas
- Sincronización con Espera Activa
- Propiedades de corrección
- Espera Activa vs Herramientas de sincronización
- Introducción a los Semáforos
- Sincronización con Semáforos
- Sincronización avanzada
- Conclusiones

SimpleConcurrent

- **SimpleConcurrent** es una librería **Java** diseñada para la **enseñanza de los conceptos básicos** a la programación concurrente
- Soporta los modelos de concurrencia:
  - Modelo de memoria compartida: **Hilos y cerrojos**
  - Paso de mensajes: **Actores**
- Se puede usar junto con cualquier otra característica básica de **Java** (Clases, métodos, arrays, etc…)

SimpleConcurrent

code URJC

- Se ha diseñado para que sea lo **más simple posible** y que los programas sean muy **compactos**
- Se ha evitado el paradigma **orientado a objetos** para hacer los programas sencillos
- **No se debe usar nunca** en un programa real porque no está optimizada ni diseñada para ello
- Se puede descargar de **github** como fuente o binario http://www.github.com/codeurjc/simpleconcurrent
- Todos los ejemplos y ejercicios de este tema están en github https://github.com/codeurjc/concurrencia-tema2

SimpleConcurrent

- **Para usar la librería vamos a usar Maven**
  - Es una **herramienta** para gestionar **proyectos Java**
  - Se **descarga automáticamente las librerías** (dependencias) de Internet
  - Se puede usar en cualquier entorno de desarrollo actualizado (**Eclipse Neon**, Netbeans, IntelliJ…)

SimpleConcurrent

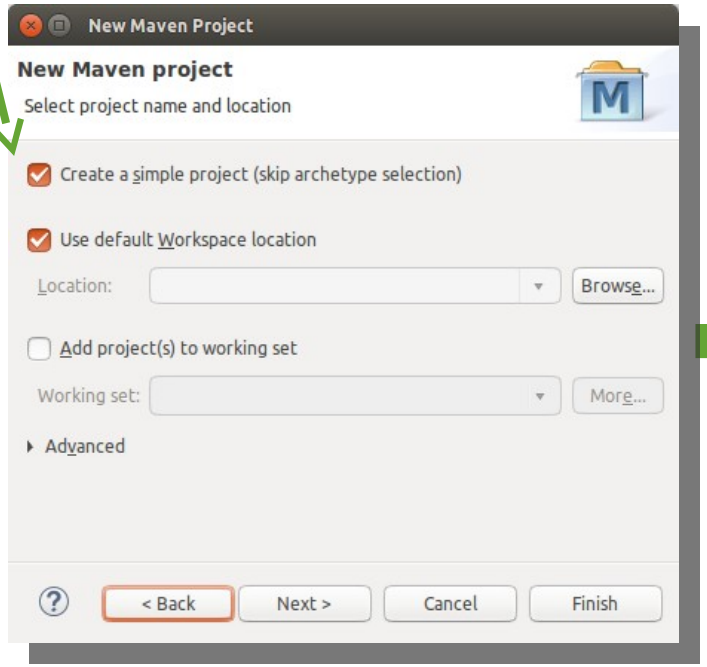
- **Cómo crear un proyecto Maven en Eclipse**
  - New project > Maven > Maven Project
  - Marcar "Create simple project (skip archetype…)"
  - Insertar los datos del proyecto
    - Group Id: es.urjc.pc
    - Artifact Id: ejercicios
  - Finish

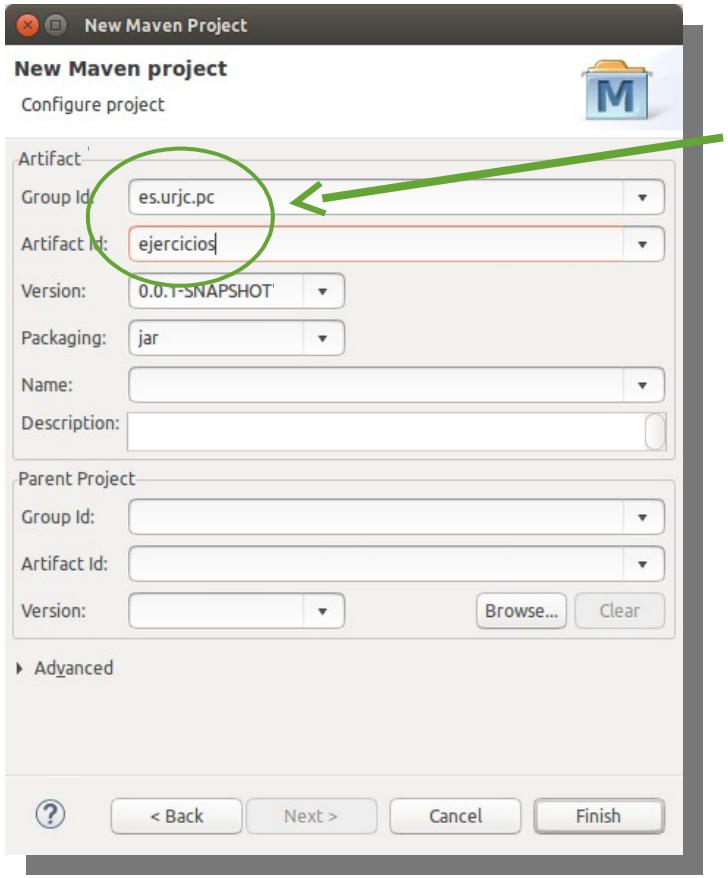
SimpleConcurrent

code URJC

- **Cómo crear un proyecto Maven en Eclipse**

Marcar

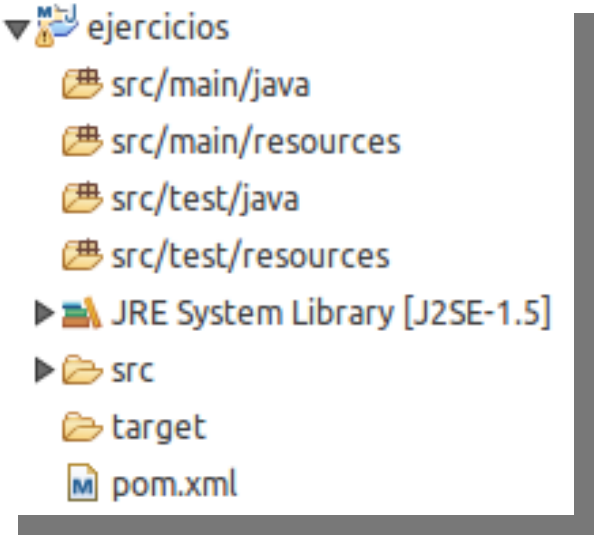




Nombre del proyecto

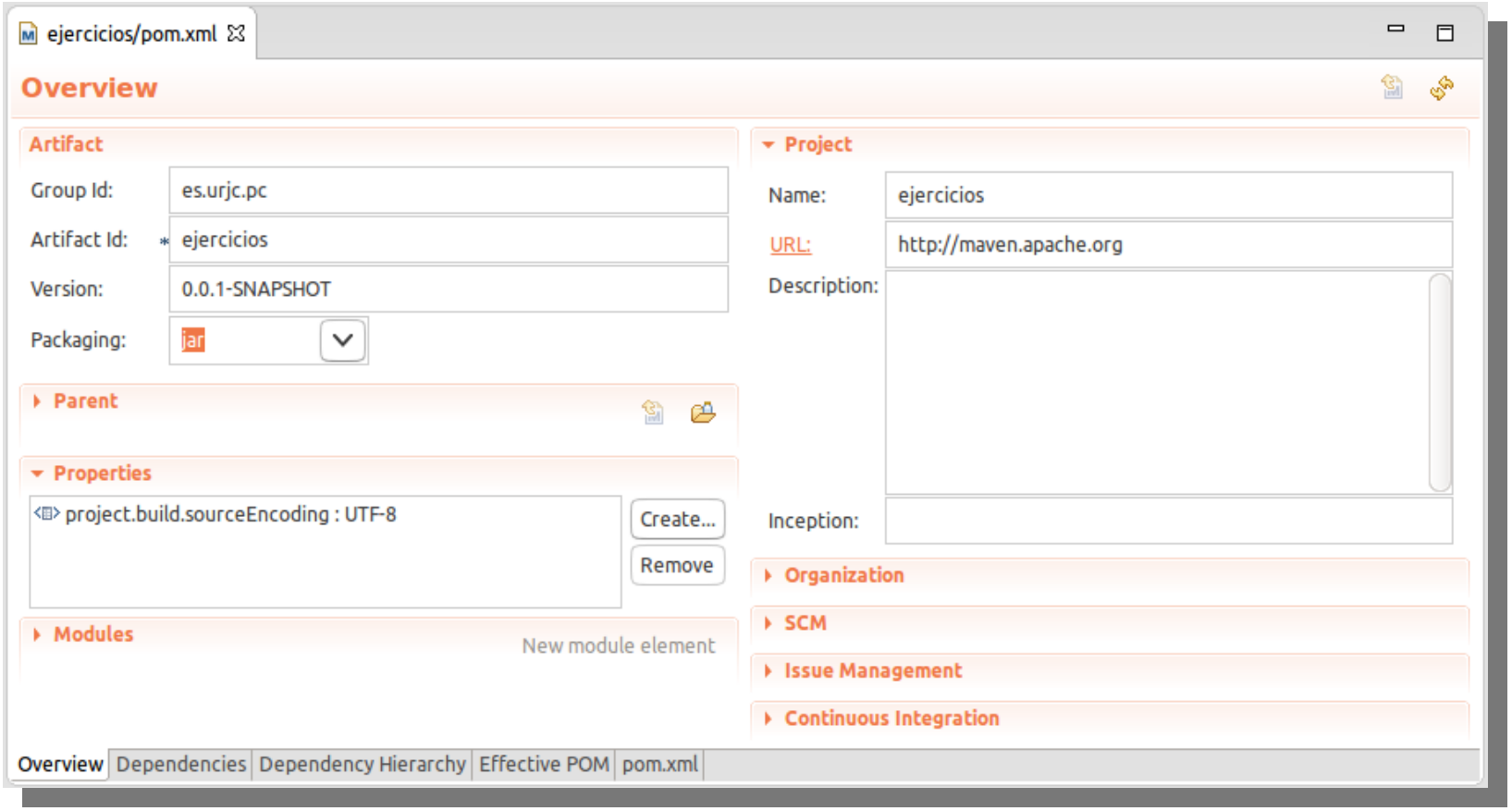
SimpleConcurrent

- **El nuevo proyecto tiene la siguiente estructura**
  - **src/main/java:** Código de la aplicación
  - **src/test/java:** Código de los tests
  - **pom.xml:** Fichero de descripción del proyecto (nombre, dependencias, configuraciones, etc…)



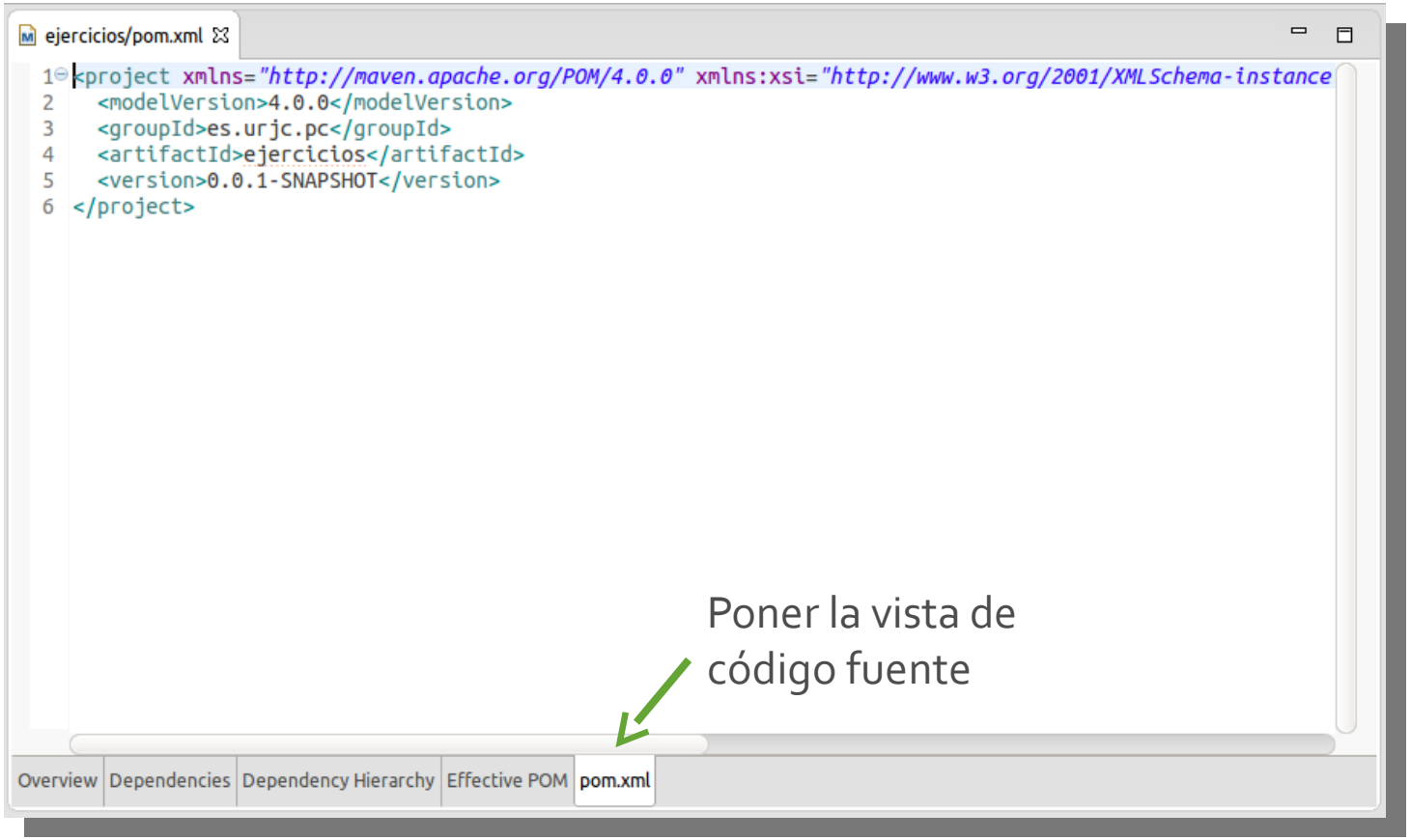
SimpleConcurrent

- **pom.xml:** Configuración del proyecto



SimpleConcurrent

- **pom.xml:** Configuración del proyecto



SimpleConcurrent

- **pom.xml:** Configuración del proyecto
  - Todas las opciones del proyecto se configuran en el fichero pom.xml
    - Configuración de **Java 8**
    - Ficheros compatibles **linux y windows (UTF-8):**

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

SimpleConcurrent

- **pom.xml:** Configuración del proyecto
  - **Librerías (dependencias):**
    - Se indica el **groupId**, **artifactId** y **versión** de cada librería que use el proyecto
    - Las librerías se descargan automáticamente
    - **Librería SimpleConcurrent versión 0.6**

```
<dependencies>
   <dependency>
      <groupId>com.github.codeurjc</groupId>
      <artifactId>simpleconcurrent</artifactId>
      <version>0.6</version>
   </dependency>
</dependencies>
```

SimpleConcurrent

- **pom.xml:** Configuración del proyecto
  - **Repositorios de librerías:**
    - La mayoría de las librerías se descargan del repositorio por defecto (Maven Central)
    - **Otras librerías están en su propio repositorio**
    - **Repositorio de SimpleConcurrent:**

```
<repositories>
   <repository>
      <id>jitpack.io</id>
      <url>https://jitpack.io</url>
   </repository>
</repositories>
```

SimpleConcurrent

code URJC

pom.xml para usar SimpleConcurrent

Datos del proyecto

Java 8

Librería SimpleConcurrent versión 0.5

Repositorio de la librería

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>es.urjc.pc</groupId>
    <artifactId>ejercicios</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.github.codeurjc</groupId>
            <artifactId>simpleconcurrent</artifactId>
            <version>0.6</version>
        </dependency>
    </dependencies>

    <repositories>
        <repository>
            <id>jitpack.io</id>
            <url>https://jitpack.io</url>
        </repository>
    </repositories>

</project>
```

SimpleConcurrent

- **Cuidado!** Algunos cambios en el fichero pom.xml no se reflejan el eclipse de forma **automática**
- Cuando se hace un cambio y eclipse no se actualiza con esos cambios, se tiene que **actualizar manualmente**
  - Botón derecho proyecto > Maven > Update Project…

SimpleConcurrent

- **Memoria Compartida** en SimpleConcurrent
  - Modelo de **Hilos y cerrojos**
  - La implementación de un **programa** concurrente con memoria compartida se divide en **dos partes**:
    - **1)** Implementar el **código** que podrá ser **ejecutado concurrentemente** (programa secuencial)
    - **2) Iniciar la ejecución** de ese código (crear el **proceso**)

SimpleConcurrent

Programa Concurrente

- Se importan los miembros estáticos de la clase **SimpleConcurrent**
- De esta forma, los métodos estáticos de **SimpleConcurrent** se pueden usar directamente, **sin** indicar el nombre de la **clase**

```
package ejemplo;

import static
    es.urjc.etsii.code.
      concurrency.SimpleConcurrent.*;

public class Ejemplo1 {

    public static void repeat(String text) {
       for(int i=0; i<5; i++){println(text);}
    }

    public static void printText() {
       println("B1");
       println("B2");
       println("B3");
    }

    public static void main(String[] args) {

       createThread("repeat","XXXXX");
       createThread("repeat","-----");
       createThread("printText");

       startThreadsAndWait();
    }
}
```

Download this code

SimpleConcurrent

Programa Concurrente

- Se escriben los **códigos secuenciales** como métodos **estáticos** de la clase principal
- Como **cualquier método estático**, pueden tener parámetros
- Usarán el método *println(...)* para imprimir por **pantalla**

```
package ejemplo;

import static
    es.urjc.etsii.code.
      concurrency.SimpleConcurrent.*;

public class Ejemplo1 {

    public static void repeat(String text) {
       for(int i=0; i<5; i++){println(text);}
    }

    public static void printText() {
       println("B1");
       println("B2");
       println("B3");
    }

    public static void main(String[] args) {

       createThread("repeat","XXXXX");
       createThread("repeat","-----");
       createThread("printText");

       startThreadsAndWait();
    }
}
```

Download this code

SimpleConcurrent

Programa Concurrente

- En el método *main* se crea un **nuevo hilo** con el método ***createThread(…)***
- Se indica el **nombre** del método que será ejecutado en ese nuevo hilo
- Se pueden pasar **parámetros**
- Es posible crear **varios hilos** que ejecutan el **mismo método,** con distintos **parámetros**
- El método ***startThreadsAndWait ()*** crea los hilos y se bloquea hasta que han terminado su ejecución

```
package ejemplo;

import static
    es.urjc.etsii.code.
      concurrency.SimpleConcurrent.*;

public class Ejemplo1 {

    public static void repeat(String text) {
       for(int i=0; i<5; i++){println(text);}
    }

    public static void printText() {
       println("B1");
       println("B2");
       println("B3");
    }

    public static void main(String[] args) {

       createThread("repeat","XXXXX");
       createThread("repeat","-----");
       createThread("printText");

       startThreadsAndWait();
    }
}
```

Download this code

SimpleConcurrent

code URJC

Programa Concurrente

- Al ejecutar el programa varias veces se obtienen las siguientes salidas por pantalla
- Como se puede ver, se obtienen **diferentes resultados** en diferentes **ejecuciones**

```
-----
XXXXX
B1
-----
XXXXX
-----
B2
-----
XXXXX
B3
XXXXX
-----
XXXXX
```

```
B1
-----
XXXXX
XXXXX
B2
-----
XXXXX
-----
B3
XXXXX
-----
XXXXX
-----
```

```
-----
XXXXX
B1
XXXXX
B2
XXXXX
-----
XXXXX
B3
-----
-----
XXXXX
-----
```

```
B1
B2
-----
XXXXX
B3
-----
-----
XXXXX
-----
XXXXX
-----
XXXXX
XXXXX
```

Memoria Compartida

- SimpleConcurrent
- **Intercalación de instrucciones: Indeterminismo**
- Variables Compartidas
- Sincronización con Espera Activa
- Propiedades de corrección
- Espera Activa vs Herramientas de sincronización
- Introducción a los Semáforos
- Sincronización con Semáforos
- Sincronización avanzada
- Conclusiones

Intercalación de instrucciones: Indeterminismo

- **El orden de las instrucciones**
  - En un programa **secuencial**, todas las instrucciones están ordenadas
  - Está definido el **orden** en el que se van ejecutando las instrucciones y la forma en la que van cambiando los valores de las **variables**
  - En la programación **concurrente**, diferentes **ejecuciones** del mismo programa pueden ejecutar las sentencias en **orden diferente**

El orden de las instrucciones

```
package ejemplo;

import static es.urjc.etsii.code.
     concurrency.SimpleConcurrent.*;

public class MaxMin {

   static volatile double n1, n2, min, max;

   public static void min(){
      min = n1 < n2? n1 : n2;
   }

   public static void max(){
      max = n1 > n2? n1 : n2;
   }

   public static void main(String[] args) {

      n1=3; n2=5; // I0
      min();      // I1
      max();      // I2
      println("m:"+min+" M:"+max); // I3
   }
}
```

Download this code

Programación Secuencial

↓

Orden Total

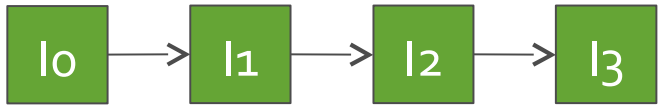


Diagrama de Precedencia

La **Relación de Precedencia** (->) entre las instrucciones define una relación de orden

I0 -> I1 -> I2 -> I3

El orden de las instrucciones

Programación Secuencial

↓

Orden Total

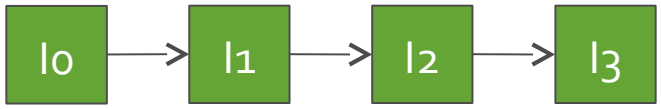


Diagrama de Precedencia

- Existe **determinismo**
- Al ejecutar el programa con los **mismos datos de entrada** se obtienen los **mismos resultados**
- Hay veces que no es necesario que una sentencia sea ejecutada antes que otra, se podrían ejecutar en **cualquier orden** ¿Cómo lo podríamos especificar en el código?

El orden de las instrucciones

```
package ejemplo;

import static es.urjc.etsii.code.
     concurrency.SimpleConcurrent.*;

public class MaxMinCon {

   static volatile double n1, n2, min, max;

   public static void min(){
      min = n1 < n2? n1 : n2;
   }

   public static void max(){
      max = n1 > n2? n1 : n2;
   }

   public static void main(String[] args) {

      n1 =3; n2 =5;          // I0
      createThread("min"); // I1
      createThread("max"); // I2
      startThreadsAndWait();
      println("m:"+min+" M:"+max); // I3
   }
}
```

Download the code

Programación Concurrente

↓

Orden Parcial

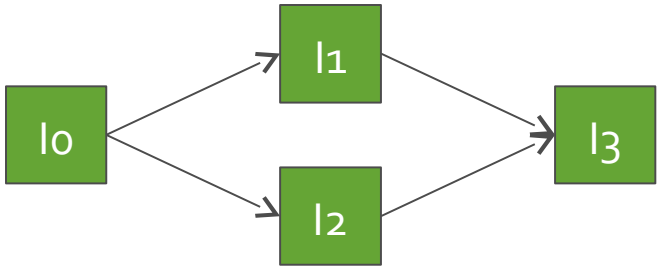


Diagrama de Precedencia

I0->I1, I0–>I2, I1->I3,I2->I3

I1 || I2

El orden de las instrucciones

Programación Concurrente

Orden Parcial

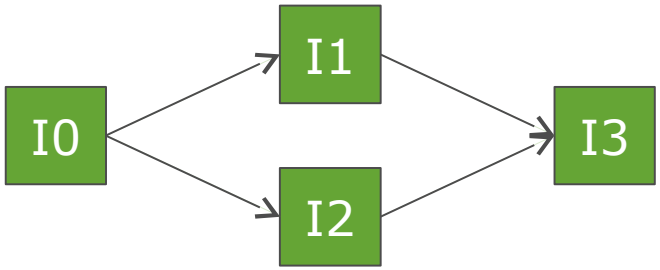


Diagrama de Precedencia

- No existe determinismo
- No se restringe el orden de ejecución de I1 e I2. Podrían ejecutarse en cualquier ordena o de forma concurrentemente

$$I1 \parallel I2$$

Instrucciones atómicas

1ª Abstracción de la Programación Concurrente

Se considera que cada proceso se ejecuta en su propio procesador

2ª Abstracción de la Programación Concurrente

Se ignoran las velocidades relativas de cada proceso, lo que posibilita considerar sólo las secuencias de instrucciones que se ejecutan

Instrucciones atómicas

- La 1ª y 2ª abstracción permiten **abstraernos** de detalles como el **número de procesadores y su velocidad**
- Nos permiten centrarnos en las diferentes posibles **secuencias de instrucciones** que ejecuta cada proceso
- Pero… ¿Exactamente **qué instrucciones** ejecuta un proceso?

Instrucciones atómicas

- Una **instrucción atómica** es aquella cuya ejecución es indivisible
- Independientemente del número de procesadores, una instrucción atómica de un proceso se **ejecuta completamente** sin interferencias
- Durante la ejecución de una sentencia atómica, otros procesos **no pueden interferir** en su ejecución

Instrucciones atómicas

- Las instrucciones atómicas se usan mucho **aplicaciones empresariales**
- Ejemplo: Reserva de vuelo con escala
  - Un viajero quiere ir de Madrid a Los Ángeles
  - Tiene que hacer escala en New York
  - Se debe reservar el billete Madrid-New York y también de New York-Los Ángeles
  - La reserva no se puede quedar a medias, reservando sólo un trayecto
  - O se reservan ambos trayectos o no se reserva ninguno

Instrucciones atómicas

- En programación concurrente, es muy importante conocer las **instrucciones atómicas** que **ejecuta el procesador**
- Esas instrucciones atómicas serán las que se ejecuten **completamente sin interferencias** de otros procesos
- Para desarrollar aplicaciones concurrentes en un lenguaje de programación, hay que conocer **qué sentencias** del lenguaje se **corresponden a sentencias atómicas**
- Las sentencias **no atómicas** ejecutadas por un proceso, podrán verse **interferidas** por la ejecución de otros procesos

Instrucciones atómicas

Sentencia Java

```
x = x+1
```

- Incrementar una variable en 1 no es una sentencia atómica en Java

Corresponde a las instrucciones atómicas

Instrucciones Atómicas

```
LOAD R,x
ADD R,#1
STR R,x
```

- Carga la variable x en el registro R del procesador
- Suma 1 al registro R del procesador
- Guarda el valor del registro R en la variable x

Intercalación (*Interleaving*)

- La 1ª y 2ª abstracción nos permiten pensar en la **secuencia de instrucciones** que ejecuta cada proceso en su procesador
- Pero es bastante **complicado** pensar en la ejecución en paralelo de **múltiples secuencias** de instrucciones (una secuencia por cada proceso)
- Para que sea más **sencillo estudiar** el comportamiento de un programa concurrente, se considera que todas las sentencias de todos los procesos se **intercalan en una única secuencia**

Intercalación (*Interleaving*)

3ª Abstracción de la Programación Concurrente
Se considera que las secuencias de ejecución de las acciones atómicas de todos los procesos se intercalan en una única secuencia

- **No hay solapamientos**
- La ejecución de dos instrucciones atómicas en **paralelo** tiene los **mismos resultados** que una después de otra (de forma **secuencial**)

Intercalación (*Interleaving*)

- Hay que estudiar lo que ocurre en **todas las posibles intercalaciones** de instrucciones atómicas (y son muchas)
- Así comprenderemos todas las posibles ejecuciones del programa (y comprobaremos si son **correctas**)

Intercalación (*Interleaving*)

- **Multiprogramación**
  - Realmente las instrucciones se ejecutan de forma intercalada porque sólo hay un procesador
- **Multiproceso**
  - Si dos instrucciones compiten por un mismo recurso, el hardware se encarga de que se ejecuten de forma secuencial
  - Si dos instrucciones no compiten, son independientes, el resultado es el mismo en paralelo que de forma secuencial

Intercalación (*Interleaving*)

- Todas las **abstracciones** se pueden resumir en una sola

Abstracción de la Programación Concurrente
Es el estudio de las secuencias de ejecución intercalada de las instrucciones atómicas de los procesos secuenciales

Principles of Concurrent and Distributed Programming, 2nd ed. 2006 Addison-Wesley. Ben-Ari.

Indeterminismo

- Al **ejecutar** un programa concurrente, se ejecutarán las sentencias con una **intercalación** determinada y se obtendrá un **resultado**
- Puesto que todas las intercalaciones de instrucciones atómicas son posibles, un mismo programa puede obtener **resultados diferentes** en **diferentes ejecuciones**
- Cuando un mismo programa obtiene resultados diferentes dependiendo de la ejecución concreta, se dice que es **indeterminista**

Indeterminismo

```
public class IncDec {

   static volatile double x;

   public static void inc(){ x = x + 1; }

   public static void dec(){ x = x - 1; }

   public static void main(String[] args){

      x = 0;

      createThread("inc");
      createThread("dec");
      startThreadsAndWait();

      println("x:"+x);
   }
}
```

Download the code

Instrucciones atómicas del tipo de proceso **inc**

```
LOAD R,x
ADD R,#1
STR R,x
```

Instrucciones atómicas del tipo de proceso **dec**

```
LOAD R2,x
SUB R2,#1
STR R2,x
```

Indeterminismo

- Una posible intercalación de instrucciones

	inc	dec	x	R	R2
1		`LOAD R2,x`	0		0
2	`LOAD R,x`		0	0	0
3		`SUB R2,#1`	0	0	-1
4	`ADD R,#1`		0	1	-1
5		`STR R2,x`	-1	1	-1
6	`STR R,x`		1	1	-1

`inc`

```
LOAD R,x
ADD R,#1
STR R,x
```

`dec`

```
LOAD R2,x
SUB R2,#1
STR R2,x
```

Resultado Final: 1

Indeterminismo

- Otra posible intercalación de instrucciones

	inc	dec	x	R	R2
1		`LOAD R2,x`	0		0
2	`LOAD R,x`		0	0	0
3	`ADD R,#1`		0	1	0
4		`SUB R2,#1`	0	1	-1
5	`STR R,x`		1	1	-1
6		`STR R2,x`	-1	1	-1

`inc`

```
LOAD R,x
ADD R,#1
STR R,x
```

`dec`

```
LOAD R2,x
SUB R2,#1
STR R2,x
```

Resultado Final: -1

Indeterminismo

Como se deben considerar todas las posibles intercalaciones, también hay que considerar que un proceso se **ejecute completamente** antes que el otro

- Otra más…

	inc	dec	x	R	R2
1		`LOAD R2,x`	0		0
2		`SUB R2,#1`	0		-1
3		`STR R2,x`	-1		-1
4	`LOAD R,x`		-1	-1	-1
5	`ADD R,#1`		-1	0	-1
6	`STR R,x`		0	0	-1

`inc`

```
LOAD R,x
ADD R,#1
STR R,x
```

`dec`

```
LOAD R2,x
SUB R2,#1
STR R2,x
```

Resultado Final: 0

Indeterminismo

- Se han encontrado **tres intercalaciones** posibles de las instrucciones atómicas de un programa concurrente en las que se obtiene un **resultado diferente**
- Hay 20 posibles intercalaciones
  - En sólo 2 de ellas se obtiene un 0
  - En 9 intercalaciones se obtiene un 1
  - En 9 intercalaciones se obtiene un -1

INTERCALACIÓN DE INST
Indeterminismo

- Los compiladores y máquinas virtuales pueden hacer ciertas **optimizaciones al intercalar** las instrucciones de los procesos
- Para **observar los efectos** de la intercalación de instrucciones en el ejemplo anterior se pueden ejecutar **múltiples incrementos** y decrementos en cada proceso

```
public class IncDecFor {

   static volatile double x;

   public static void inc(){
     for (int i = 0; i < 10000000; i++){
         x = x + 1;
     }
   }

   public static void dec(){
     for (int i = 0; i < 10000000; i++) {
         x = x - 1;
     }
   }

   public static void main(String[] args){
      x = 0;
      createThread("inc");
      createThread("dec");
      startThreadsAndWait();
      println("x:"+x);
   }
}
```

Download the code

Memoria Compartida

- SimpleConcurrent
- Intercalación de instrucciones: Indeterminismo
- **Variables Compartidas**
- Sincronización con Espera Activa
- Propiedades de corrección
- Espera Activa vs Herramientas de sincronización
- Introducción a los Semáforos
- Sincronización con Semáforos
- Sincronización avanzada
- Conclusiones

Variables Compartidas

- **Modelo de memoria compartida** Los procesos pueden acceder a una **memoria común**
- Existen variables compartidas que **varios procesos** pueden **leer** y **escribir**
- Un valor **escrito por un proceso** puede ser leído por **otro proceso**

¿Qué ocurre si dos procesos **leen** o **escriben** de forma **simultánea en la misma variable**?

Variables Compartidas

- La **abstracción** de la programación concurrente nos indica que todas las **instrucciones atómicas** de cada proceso se intercalan en una única secuencia
- Con **SimpleConcurrent**, la lectura y la escritura de **atributos compartidos** de tipo simple (**`boolean`**, **`int`**, **`char`**…) son instrucciones **atómicas** en Java

Variables Compartidas

- **Lectura simultáneas**
  - Ambos procesos leen el valor de la variable
  - No interfieren entre sí
- **Escrituras simultáneas**
  - El resultado de la escritura simultánea sobre una variable compartida será el valor escrito por uno u otro proceso
  - Nunca una mezcla de las dos escrituras

Variables Compartidas

- **Lectura y Escritura simultáneas**
  - Si un proceso lee sobre una variable compartida y simultáneamente otro escribe sobre ella, el resultado de la lectura será:
    - O el valor de la variable antes de la escritura
    - O el valor de la variable después de la escritura
  - **Nunca un valor intermedio**

Memoria Compartida

- SimpleConcurrent
- Intercalación de instrucciones: Indeterminismo
- Variables Compartidas
- **Sincronización con Espera Activa**
- Propiedades de corrección
- Espera Activa vs Herramientas de sincronización
- Introducción a los Semáforos
- Sincronización con Semáforos
- Sincronización avanzada
- Conclusiones

Memoria Compartida

- **Sincronización con Espera Activa**
  - **Relaciones entre procesos**
  - Sincronización Condicional
  - Exclusión Mutua

Relaciones entre procesos

- **Relaciones entre procesos concurrentes**
  - **Independientes**
    - Son gestionados por el Sistema
    - No nos tenemos que preocupar como programadores.
  - **Interactúan entre sí**
    - Compiten por los recursos o se comunican entre sí para obtener un resultado conjunto

Relaciones entre procesos

- **Interacción**
  - **Competencia:** Varios procesos deben compartir recursos comunes del sistema (procesador, memoria, disco, impresoras,…) por lo que compiten entre ellos para conseguirlo
  - **Cooperación:** Varios procesos deben trabajar sobre distintas partes de un problema para resolverlo conjuntamente

Relaciones entre procesos

- Dos procesos que **interactúan** entre sí lo hacen a través de las siguientes **actividades**
  - **Comunicación:** Es el intercambio de información entre procesos. Dos procesos cooperan entre sí intercambian información de algún modo
  - **Sincronización:** La sincronización impone restricciones a la ejecución de las sentencias de los procesos (**espera**)
    - Sincronización condicional
    - Exclusión mutua

Relaciones entre procesos

- **Sincronización Condicional**
  - Uno o más procesos no pueden continuar su ejecución (**tienen que esperarse**) a que se **cumpla cierta condición**
  - **Otro proceso** es el que establece esa **condición**

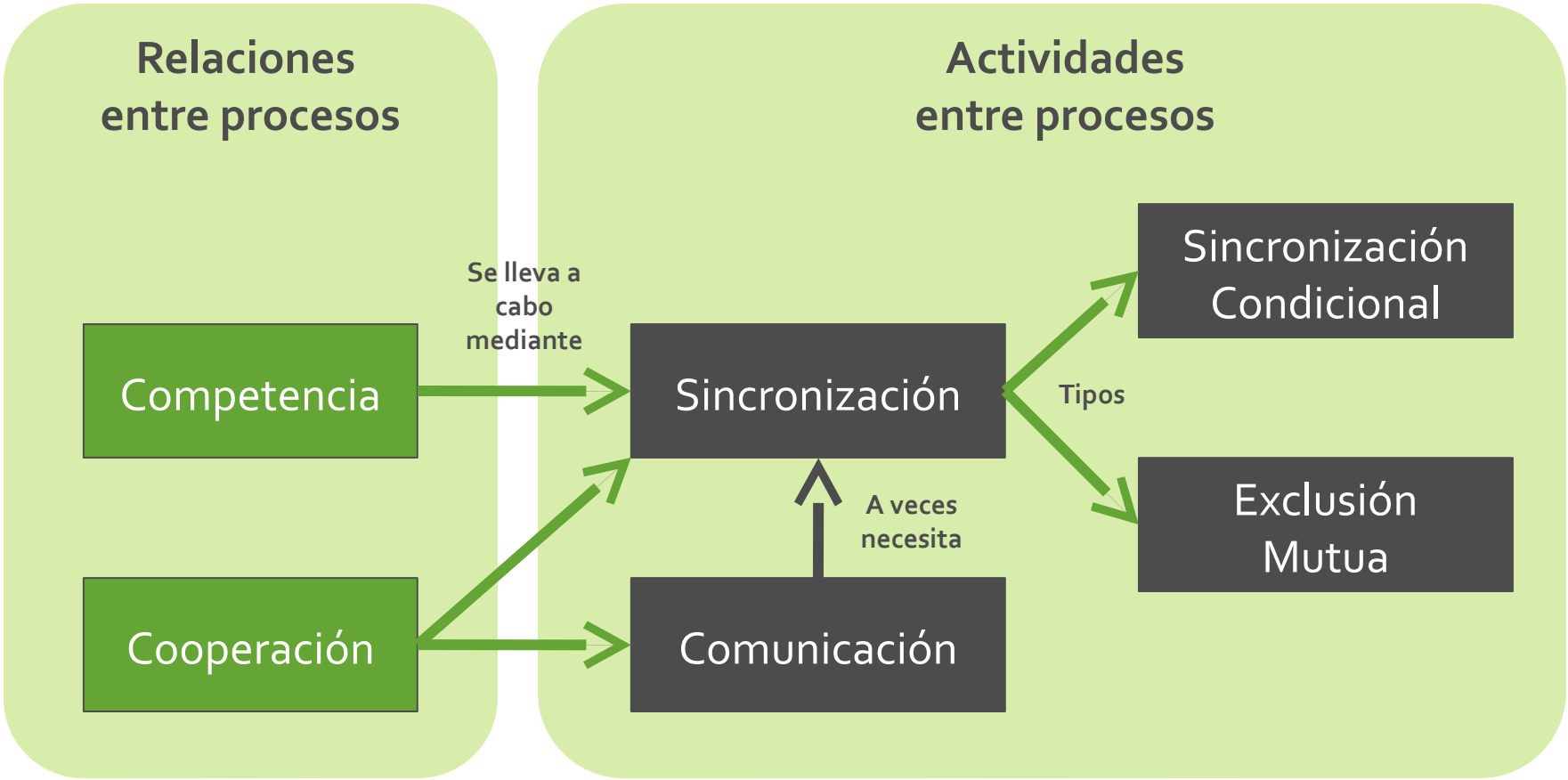
Un proceso **servidor web** está esperando a que otro proceso **navegador web** realice una **petición** mediante http

Relaciones entre procesos

- **Exclusión Mutua**
  - Varios procesos compiten por un recurso común de **acceso exclusivo**
  - Sólo uno de los procesos puede estar accediendo al **recurso a la vez**
  - Si varios procesos quieren usar el recurso, **tienen que esperarse a que quede libre**
  - Los procesos se **excluyen mutuamente**

Sólo un proceso puede acceder a la vez a un recurso como un **lector de huellas digitales**

Relaciones entre procesos



Relaciones entre procesos

- **Comunicación** en el modelo de memoria compartida
  - Se utilizan las **variables o atributos compartidos** para compartir información
  - Un proceso **escribe un valor** en la variable y otro proceso **lee ese valor**

Relaciones entre procesos

- **Sincronización** en el modelo de memoria compartida
  - **Espera activa:** Utiliza únicamente variables compartidas para sincronizarse
  - **Herramientas de sincronización:** Cuando se usan, además de variables compartidas, herramientas de sincronización entre procesos (Semáforos, monitores, etc…)

Memoria Compartida

- Sincronización con Espera Activa
  - Relaciones entre procesos
  - **Sincronización Condicional**
  - Exclusión Mutua

Sincronización Condicional

- La **Sincronización Condicional** se produce cuando un proceso debe esperar a que se cumpla una cierta condición para proseguir su ejecución
- Esta condición sólo puede ser activada por otro proceso

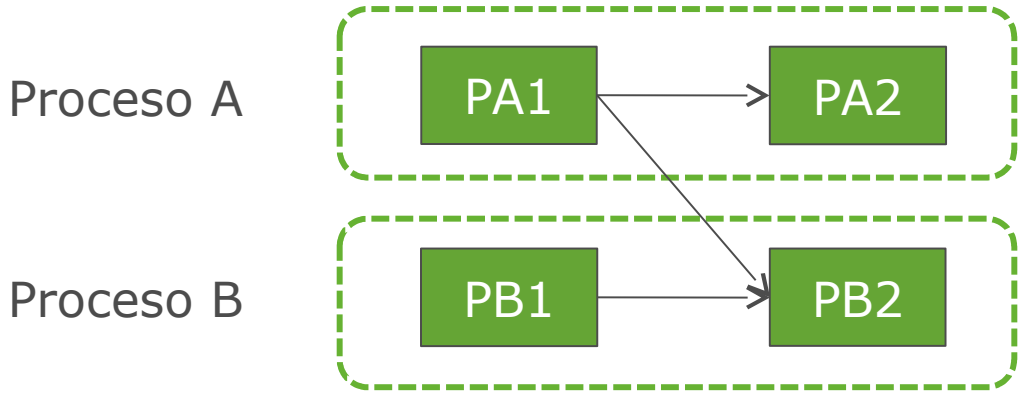


Diagrama de Precedencia

61

Sincronización Condicional

- Si los procesos **muestran por pantalla*** el nombre de la acción que han realizado, el resultado de ejecutar el programa concurrente del diagrama podría ser cualquiera de los siguientes:

Salidas Posibles:

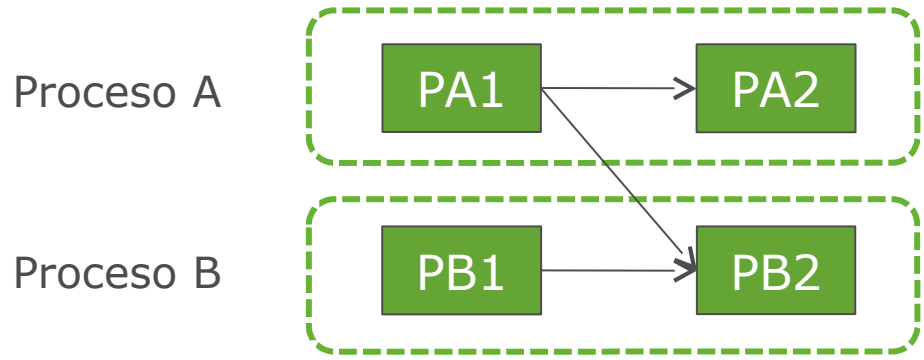
```
PA1 PA2 PB1 PB2
```

```
PA1 PB1 PA2 PB2
```

```
PB1 PA1 PA2 PB2
```

Salidas No Posibles:

~~`PB1 PB2 PA1 PA2`~~



* La escritura en pantalla con `println` es una instrucción atómica en `SimpleConcurrent`

Sincronización Condicional

Se declara un atributo compartido **continuar**

El proceso **a** la pondrá a **true** cuando haya ejecutado PA1 y continuará ejecutando PA2

El proceso **b** ejecutará PB1 y comprobará repetidamente en un bucle el valor de **continuar**
Saldrá del bucle cuando valga **true** y continuará ejecutando PB2

```
public class SincCond {

   static volatile boolean continuar;

   public static void a() {
      print("PA1 ");
      continuar = true;
      print("PA2 ");
   }

   public static void b() {
      print("PB1 ");
      while (!continuar);
      print("PB2 ");
   }

   public static void main(String[] args){

      continuar = false;

      createThread("a");
      createThread("b");

      startThreadsAndWait();
   }
}
```

Download this code

Sincronización Condicional

code URJC

Posible intercalación de instrucciones

	a	b	continuar
1	`print("PA1 ");`		`false`
2		`print("PB1 ");`	`false`
3	`continuar = true;`		`true`
4	`print("PA2 ");`		`true`
5		`while (!continuar);`	`true`
6		`print("PB2 ");`	`true`

Salida: `PA1 PB1 PA2 PB2`

Sincronización Condicional

code URJC

Posible intercalación de instrucciones

	a	b	continuar
1		`print("PB1 ");`	`false`
2		`while (!continuar);`	`false`
3		`while (!continuar);`	`false`
4		`while (!continuar);`	`false`
5	`print("PA1 ");`		`false`
6	`continuar = true;`		`true`
7	`print("PA2 ");`		`true`
8		`while (!continuar);`	`true`
9		`print("PB2 ");`	`true`

Salida: `PB1 PA1 PA2 PB2`

Sincronización Condicional

code URJC

Posible intercalación de instrucciones

	a	b	continuar
1	`print("PA1 ");`		`false`
2	`continuar = true;`		`true`
3	`print("PA2 ");`		`true`
4		`print("PB1 ");`	`true`
5		`while (!continuar);`	`true`
6		`print("PB2 ");`	`true`

Salida: `PA1 PA2 PB1 PB2`

Ejercicio 1 – Productor Consumidor

code URJC

- Se desea implementar un programa concurrente con un proceso que produce información (**productor**) y otro proceso que hace uso de esa información (**consumidor**)
- El proceso **productor** genera un número aleatorio y termina
- El proceso **consumidor** muestra por pantalla el número generado y termina

Ejercicio 2 – Productor Consumidor N

- Se desea ampliar el programa anterior de forma que el proceso **productor** genere 5 números (consecutivos)
- El proceso **consumidor** consumirá esos 5 números
- No se puede quedar ningún producto sin consumir
- No se puede consumir dos veces el mismo producto

Ejercicio 3 – Cliente Servidor 1 Petición

- Programa formado por un **proceso servidor** y otro **proceso cliente**
- El **proceso cliente** hace una petición al **proceso servidor** (con un número entero) y espera su respuesta. Cuando la recibe, la procesa.
- El **proceso servidor** no hace nada hasta que recibe una petición, momento en el que suma 1 y devuelve el valor.
- El **proceso cliente** muestra un número por pantalla, se lo envía al servidor y cuando le llega la respuesta, la muestra por pantalla

Ejercicio 4 – Cliente Servidor N Peticiones

- Se desea ampliar el programa anterior de forma que el **proceso cliente** esté constantemente haciendo peticiones y el **proceso servidor** atendiendo a las mismas

Ejercicio 4B – N Clientes Servidor N Peticiones

- Se desea ampliar el programa anterior de forma que existan varios clientes que acceden al mismo servidor
- Para facilitar la depuración, cada proceso cliente debe empezar los números consecutivos en un valor diferente (de 10 en 10)

Memoria Compartida

- Sincronización con Espera Activa
  - Relaciones entre procesos
  - Sincronización Condicional
  - **Exclusión Mutua**

Exclusión Mutua

- **¿Qué es la Exclusión Mutua?**
  - Es un tipo de **sincronización** entre procesos que se tiene cuando **varios procesos** compiten por un **recurso** común de **acceso exclusivo**
  - **Sólo uno** de los procesos puede estar accediendo al **recurso a la vez** y los demás tienen que esperar
  - Cuando un **proceso libera** el recurso, otro proceso que estuviera **esperando podrá acceder a él**

Exclusión Mutua

- **¿Qué es el Problema de la Exclusión Mutua?**
  - Es un problema **histórico**, definido por los primeros **investigadores** en Programación Concurrente
  - El problema se utilizó para **investigar la mejor forma de implementar** la sincronización de tipo exclusión mutua usando únicamente variables compartidas (espera activa)

Exclusión Mutua

code URJC

- **El problema de la Exclusión Mutua**
  - Se tienen dos o más procesos concurrentes, que ejecutan indefinidamente una secuencia de instrucciones dividida en dos secciones
    - Sección bajo exclusión mutua
    - Sección sin exclusión mutua

```
public static void p1() {

    while(true) {
        // Sección bajo EM
        printlnI("P1_EM1");
        printlnI("P1_EM2");

        // Sección sin EM
        printlnI("P1_1");
        printlnI("P1_2");
    }
}
```

Download this code

El método ***printlnI*** imprime en una columna diferente la información de cada proceso. Esto permite diferenciar de forma muy sencilla qué instrucciones ejecuta cada proceso.

Exclusión Mutua

code URJC

- **Sección bajo Exclusión Mutua (Sección crítica)**
  - Secuencia de instrucciones que acceden a un recurso compartido de **acceso exclusivo**
  - Puesto que el recurso es de acceso exclusivo y solo un proceso puede acceder a él al mismo tiempo, cada proceso debe ejecutar las **instrucciones de la sección bajo EM** sin que haya **intercalación** de **instrucciones de la sección bajo EM** de otros procesos
  - Se pueden **intercalar instrucciones** que no hagan uso del recurso, es decir, instrucciones de la sección sin EM de otros procesos

Exclusión Mutua

- **Sección sin Exclusión Mutua**
  - Secuencia de instrucciones que pueden ser ejecutadas concurrentemente por todos los procesos
  - Sus instrucciones se pueden intercalar con las instrucciones de la sección bajo EM de otros procesos

Exclusión Mutua

```
P1_EM1
P1_EM2
P1_1
          P2_EM1
P1_2
          P2_EM2
          P2_1
P1_EM1
          P2_2
P1_EM2
P1_1
          P2_EM1
          P2_EM2
          P2_1
P1_2
P1_EM1
...
```

- Ejecución de ejemplo con dos procesos ejecutando instrucciones de la **sección bajo EM** y a continuación instrucciones de la **sección sin EM**
- **No se intercalan** sentencias de **las secciones bajo EM** de los dos procesos
- **Sí se intercalan** instrucciones de las **secciones sin EM**

Exclusión Mutua

- **Implementación de la Exclusión Mutua con espera activa**
  - No es fácil de implementar
  - Veremos dos **intentos de implementación** pero que **no funcionan correctamente**
  - El estudio de estos intentos nos permitirá **entender los errores habituales** de los programas concurrentes (y más adelante veremos cómo evitarlos)

Primer intento

- Se puede usar una **variable booleana** que indique si hay algún proceso ejecutando las secciones que deben estar bajo exclusión mutua (para que los demás se esperen)
- Un proceso, antes de ejecutar las instrucciones bajo exclusión mutua, **comprueba si ya hay otro proceso mirando esa variable**
- Si hay otro proceso, **espera** hasta que termine
- Si no hay ningún proceso en, **pone a true la variable** y empieza a ejecutar las instrucciones
- Cuando termina, vuelve a **poner a false la variable**

Primer intento

code URJC

```
static volatile boolean ocupado;
```

```
for (int i = 0; i < 5; i++) {

  while (ocupado);
  ocupado = true;

  // Sección bajo EM
  println("P1_EM1");
  println("P1_EM2");

  ocupado = false;

  // Sección sin EM
  println("P1_1");
  println("P1_2");
}
```

```
for (int i = 0; i < 5; i++) {

  while (ocupado);
  ocupado = true;

  // Sección bajo EM
  println("P2_EM1");
  println("P2_EM2");

  ocupado = false;

  // Sección sin EM
  println("P2_1");
  println("P2_2");
}
```

Download this code

	p1	p2	ocupado
1	`while (ocupado);`		false
2	`ocupado = true;`		true
3		`while (ocupado);`	true
4	`println("P1_EM1");`		true
5		`while (ocupado);`	true
6	`println("P1_EM2");`		true
7		`while (ocupado);`	true
8	`ocupado = false;`		false
9		`while (ocupado);`	false
10	`println("P1_1");`		false
11		`ocupado = true;`	true
12		`println("P2_EM1");`	true
13	`println("P1_2");`		true
14	`while (ocupado);`		true
15		`println("P2_EM2");`	true
16	`while (ocupado);`		true
17		`ocupado = false;`	false
18	`while (ocupado);`		false
19	…		

Primer intento

- La intercalación que hemos visto se comporta **correctamente**
- El problema es que existen intercalaciones que no funcionan correctamente
- Los dos procesos pueden ejecutar las instrucciones que deberían estar **bajo exclusión muta de forma intercalada**

Primer intento

- **Intercalación problemática**
  - Se permite que dos procesos ejecuten las sentencias de la EM de forma intercalada

	p1	p2	ocupado
1	`while (ocupado);`		`false`
2		`while (ocupado);`	`false`
3	`ocupado = true;`		`true`
4		`ocupado = true;`	`true`
5	`println("P1_EM1");`		`true`
6		`println("P2_EM1");`	`true`
7	`println("P1_EM2");`		`true`
8		`println("P2_EM2");`	`true`
9	…		

Segundo intento

- El **problema** del intento anterior es que los dos procesos **miran y después entran**, pudiendo entrar los **dos a la vez**
- En el segundo intento **antes de comprobar** si hay alguien dentro, el proceso **pide** ejecutar instrucciones bajo exclusión mutua
- Si alguien lo ha pedido ya, el **proceso se espera**

Segundo intento

```
static volatile boolean p1p;
static volatile boolean p2p;
```

```
for (int i = 0; i < 5; i++) {

  p1p = true;
  while (p2p);

  // Sección bajo EM
  println("P1_EM1");
  println("P1_EM2");

  p1p = false;

  // Sección sin EM
  println("P1_1");
  println("P1_2");
}
```

```
for (int i = 0; i < 5; i++) {

  p2p = true;
  while (p1p);

  // Sección bajo EM
  println("P2_EM1");
  println("P2_EM2");

  p2p = false;

  // Sección sin EM
  println("P2_1");
  println("P2_2");
}
```

Download this code

	p1	p2	p1p	p2p
1	`p1p = true;`		true	false
2	`while (p2p);`		true	false
3		`p2p = true;`	true	true
4		`while (p1p);`	true	true
5	`println("P1_EM1");`		true	true
6		`while (p1p);`	true	true
7	`println("P1_EM2");`		true	true
8	`p1p = false;`		false	true
9		`while (p1p);`	false	true
10	`println("P1_1");`		false	true
11	`println("P1_2");`		false	true
12		`println("P2_EM1");`	false	true
13	`p1p = true;`		true	true
14	`while (p2p);`		true	true
15		`println("P2_EM2");`	true	true
16		`p2p = false;`	true	false
17	`while (p2p);`		true	false
18	`println("P1_EM1 ");`		true	false
19	…			

Segundo intento

- **Intercalación problemática**
  - Si los dos procesos anotan su petición a la vez, se quedan **esperando para siempre (interbloqueo)**

	p1	p2	c.p1p	c.p2p
1	`p1p = true;`		`true`	`false`
2		`p2p = true;`	`true`	`true`
3	`while (p2p);`		`true`	`true`
4		`while (p1p);`	`true`	`true`
5		`while (p1p);`	`true`	`true`
6		`while (p1p);`	`true`	`true`
7	`while (p2p);`		`true`	`true`

Exclusión Mutua

- Los **problemas** que hemos visto son **típicos** al empezar a programar de forma concurrente
- Hay que tener muy presentes **todas** las posibles **intercalaciones de las instrucciones**
- Por muy **baja sea la probabilidad** de que algo malo ocurra, puede **ocurrir** en cualquier momento
- **Ejecutar** el programa **varias veces** sin problemas **no es garantía** de que no tenga problemas

Exclusión Mutua

- Existen varios algoritmos para implementar sincronización de **Exclusión Mutua** con **espera activa**
  - El algoritmo de **Dekker**: se utiliza para dos procesos
  - El algoritmo de **Lamport**: es un algoritmo genérico diseñado para N procesos
- Estos algoritmos son bastante **complejos** y no los vamos a estudiar
- Información sobre ellos:

Ben-Ari, M. *Principles of Concurrent and Distributed Programming*. 2nd Ed. Prentice Hall, 2.006.

Exclusión Mutua

- **SimpleConcurrent** implementa la sincronización de Exclusión Mutua:
  - Para entrar en la sección bajo EM:

```
enterMutex();
```

  - Para salir de la sección bajo EM:

```
exitMutex();
```

Exclusión Mutua

```
public static void p1() {

   for (int i = 0; i < 5; i++) {

      // Sección bajo EM
      enterMutex();
      printlnI("P1_EM1");
      printlnI("P1_EM2");
      exitMutex();

      // Sección sin EM
      printlnI("P1_1");
      printlnI("P1_2");
   }
}
```

```
public static void p2() {

   for (int i = 0; i < 5; i++) {

      // Sección bajo EM
      enterMutex();
      printlnI("P2_EM1");
      printlnI("P2_EM2");
      exitMutex();

      // Sección sin EM
      printlnI("P2_1");
      printlnI("P2_2");
   }
}
```

Download this code

Exclusión Mutua

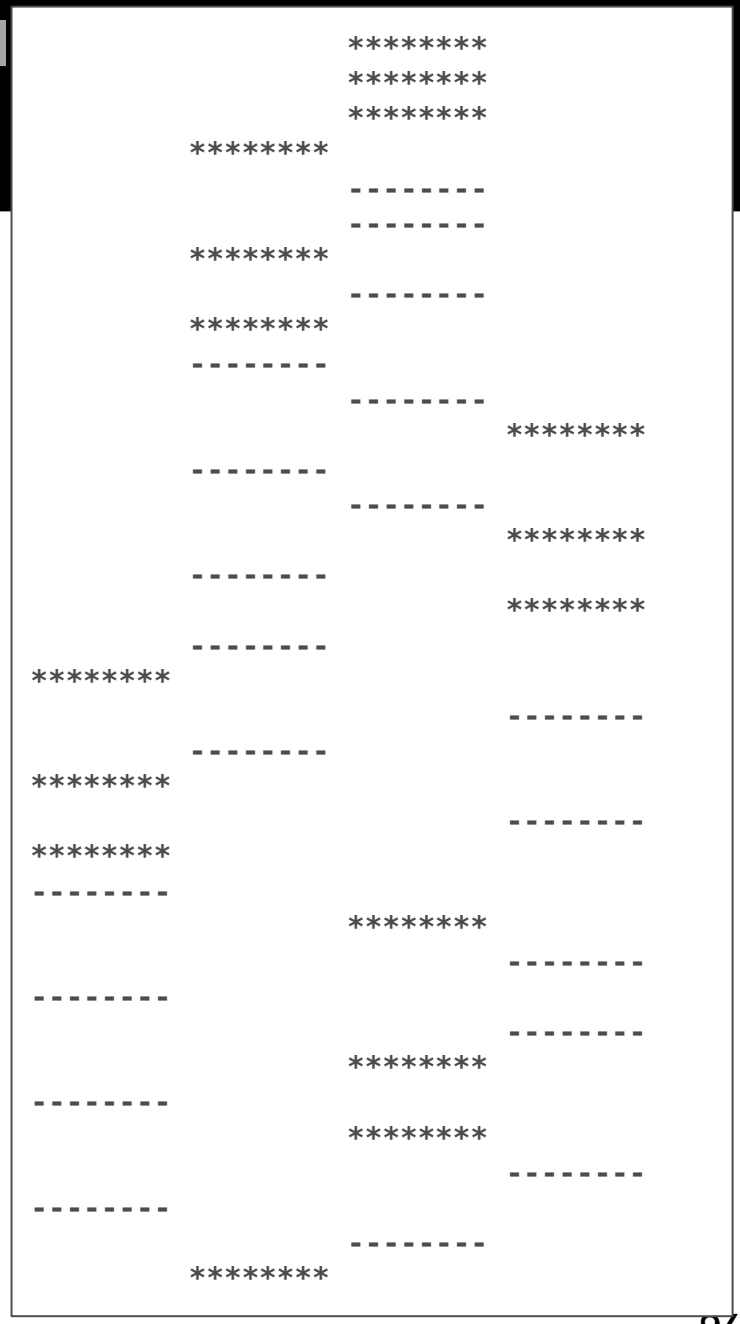
- Programa que muestra **El problema de la Exclusión Mutua** de forma textual al ejecutarse

```
public class ExcMutuaEA2 {

   public static void p() {
     for (int i=0; i<5; i++) {
       enterMutex();
       printlnI("********");
       printlnI("********");
       printlnI("********");
       exitMutex();

       printlnI("--------");
       printlnI("--------");
       printlnI("--------");
       printlnI("--------");
       printlnI("--------");
     }
   }

   public static void main(String[] args){
     createThreads(4,"p");
     startThreadsAndWait();
   }
}
```

Download this code

Exclusión Mutua

- Salida por **pantalla** de una de las posibles ejecuciones del programa
- Se puede observar como los **asteriscos nunca se intercalan** porque corresponden a la sección crítica (están bajo **exclusión mutua**)



Exclusión Mutua

- Normalmente la **sección bajo exclusión mutua** se usa para acceder de forma **exclusiva** a una **variable compartida**
- De esa forma **no aparecen interferencias entre varios hilos accediendo a la misma vez**
- Una misma **variable compartida** se puede usar en **varios partes del programa**

Exclusión Mutua

- Variable **x** bajo exclusión mutua con dos secciones diferentes (**incremento** y **decremento**)
- El resultado de este programa siempre será **correcto (cero)**
- **No** se pueden producir **errores** al contar porque **no se pueden intercalar de forma anómala** instrucciones que usan la **variable x**

```
public class IncDecEM {

   static volatile double x;

   public static void inc() {

      for (int i = 0; i < 10000000; i++){
         enterMutex();
         x = x + 1;
         exitMutex();
      }
   }

   public static void dec() {
      for (int i = 0; i < 10000000; i++){
         enterMutex();
         x = x - 1;
         exitMutex();
      }
   }
}
```

Download this code

Exclusión Mutua

- **Varios recursos** diferentes pueden estar **bajo exclusión mutua** en un mismo programa
- Las **secciones críticas** de cada recurso son completamente **independientes** entre sí. Incluso pueden **intercalarse** sus instrucciones
- Las secciones críticas de diferentes recursos pueden **anidarse si es necesario**
- Se debe indicar el nombre del recurso como un **parámetro** de ***enterMutex(…)*** y ***exitMutex(…)***

SINCRONIZACIÓN CON E
Exclusión Mutua

- Las variables **x** e **y** están bajo exclusión mutua con **dos secciones independientes**
- **Se pueden intercalar** instrucciones que usan **variables diferentes**
- **No se pueden intercalar** instrucciones que usen la **misma variable**
- El resultado final en cada variable es el **correcto (cero)**

```
public class IncDecEM2 {

   static volatile double x, y;

   public static void incX() {
      enterMutex("x");
      x = x + 1;
      exitMutex("x");
   }
   public static void decX() {
      enterMutex("x");
      x = x - 1;
      exitMutex("x");
   }
   public static void incY() {
      enterMutex("y");
      y = y + 1;
      exitMutex("y");
   }
   public static void decY() {
      enterMutex("y");
      y = y - 1;
      exitMutex("y");
   }
}
```

Download this code

Ejercicio 5 - Museo

- Existen 3 personas en el mundo, 1 museo, y sólo cabe una persona dentro del museo
- Las personas realizan cuatro acciones dentro del museo
  - Cuando entran al museo saludan: “hola!”
  - Cuando ven el museo se sorprenden: “qué bonito!” y “alucinante!”
  - Cuando se van del museo se despiden: “adiós”
- Cuando salen del museo se van a dar un “paseo”
- Después del paseo, les ha gustado tanto que vuelven a entrar

Ejercicio 5 - Museo

- **Se pide:**
  - Tipos de procesos
  - Número de procesos del programa concurrente y de qué tipo son
  - Escribir lo que hace cada proceso
  - Identificar los recursos compartidos
  - Identificar la sección o secciones críticas
  - Escribir el programa completo

Ejercicio 6 – Museo con infinitas personas

- Considerar que caben infinitas personas dentro del museo
- Cada persona al entrar tiene que saludar diciendo cuántas personas hay en el museo: "hola, somos 3"
- Al despedirse tiene que decir el número de personas que quedan tras irse: "adiós a los 2"

Ejercicio 7 – Museo con regalo

- Para incentivar las visitas, cuando una persona entre en el museo estando vacío, será obsequiado con un regalo
- Las personas, después de saludar, dicen si les han dado un regalo (“Tengo regalo”) o si no (“No tengo regalo”)
- Las personas deben permitir que otras personas hablen entre el saludo y el comentario sobre el regalo

Exclusión Mutua

- Una **instrucción atómica** es aquella que se ejecuta como una **unidad indivisible**
- El lenguaje de programación y el hardware definen las **instrucciones atómicas** en las que se divide cada sentencia

Sentencia alto nivel

```
x = x+1
```

Instrucciones Atómicas

```
LOAD R,x
ADD R,#1
STR R,x
```

Exclusión Mutua

- Supongamos que **dos procesos** quieren usar una **variable común** para contar las acciones realizadas
- Si dos procesos quieren **incrementar la misma variable** existen **intercalaciones** de las instrucciones atómicas que producen **errores** en la cuenta
- Para el desarrollador sería muy interesante que el **incremento** de una **variable** fuese una **instrucción atómica**

Exclusión Mutua

- **Tipos de instrucciones atómicas**
  - **De grano fino**
    - Ofrecidas por el lenguaje de programación y el hardware al desarrollador
    - Habitualmente se corresponden con las instrucciones máquina del procesador
  - **De grano grueso**
    - Conjunto de sentencias que ejecuta un proceso sin interferencias de otros procesos
    - Los lenguajes de programación y el sistema hardware disponen de mecanismos para hacer que un grupo de sentencias se ejecuten como una instrucción atómica de grano grueso

Exclusión Mutua

- Una **sección bajo Exclusión Mutua** es una instrucción **atómica** de **grano grueso**
  - **Indivisible**
    - Ningún otro proceso puede **interferir** en el uso del recurso **compartido**
    - Si dentro de la sección crítica se **incrementa una variable**, ningún otro proceso podrá interferir y no se producirán errores en la cuenta
  - **Divisible**
    - Pero es **divisible** en el sentido de que se pueden **intercalar instrucciones** de la sección no crítica de otros procesos

Memoria Compartida

- SimpleConcurrent
- Intercalación de instrucciones: Indeterminismo
- Variables Compartidas
- Sincronización con Espera Activa
- **Propiedades de Corrección**
- Espera Activa vs Herramientas de sincronización
- Introducción a los Semáforos
- Sincronización con Semáforos
- Sincronización avanzada
- Conclusiones

Propiedades de Corrección

- La programación concurrente es tan compleja que se han definido un conjunto de **propiedades de corrección** que todo programa concurrente debe cumplir para considerarse correcto
- Estas propiedades deben cumplirse en cualquier posible **intercalación de las instrucciones atómicas**

Propiedades de Corrección

- **Exclusión Mutua**
  - Hay que poner correctamente bajo **exclusión mutua** los recursos **compartidos**
  - De no ser así, se pueden producir **interferencias indeseadas** entre los procesos que tengan como consecuencia la **corrupción de los datos** compartidos
  - Cuando esto ocurre se dice que se ha producido una **"condición de carrera"** (***race condition***)

Propiedades de Corrección

- **Ausencia de Interbloqueos**
  - Dos procesos se bloquean mutuamente cuando se están esperando el uno al otro y no pueden continuar su ejecución mientras esperan
  - **Interbloqueo activo (*livelock*):**
    - Cuando los procesos ejecutan instrucciones que no producen un avance real del programa, sólo sirven para sincronizarse entre sí
  - **Interbloqueo pasivo (*deadlock*)**
    - Cuando todos los procesos están bloqueados esperando indefinidamente
    - Este interbloqueo sólo se produce con herramientas de sincronización (se verán más adelante)

Propiedades de Corrección

- **Ausencia de Retrasos Innecesarios**
  - Los procesos deben **progresar en su ejecución** si no hay otro proceso compitiendo con él para entrar en las secciones bajo exclusión mutua
  - Generalmente las primitivas concurrentes garantizan que esto no ocurra. El desarrollador no tiene que preocuparse de ello.

Propiedades de Corrección

- **Ausencia de Inanición (*starvation*)**
  - Todo proceso que quiera **acceder a un recurso compartido** deberá poder **hacerlo** (en algún momento)
  - Generalmente las primitivas concurrentes garantizan que esto no ocurra. El desarrollador no tiene que preocuparse de ello.

Propiedades de Corrección

- **Tipos de propiedades de corrección:**
  - **De seguridad (*safety*):** Si alguna de estas propiedades se incumple en alguna ocasión, el programa se comportará de forma errónea
  - **De Vida (*liveness*):** Si alguna de estas propiedades se incumple en "alguna" ocasión, el programa se comportara de forma correcta pero será más lento de lo que podría ser y desaprovechará los recursos

Propiedades de Corrección

- **Propiedades de seguridad (*safety*)**
  - Exclusión Mutua
  - Ausencia de Interbloqueo pasivo
- **Propiedades de Vida (*liveness*)**
  - Ausencia de Retrasos innecesarios
  - Ausencia de inanición (*starvation*)
  - Ausencia de interbloqueo activo (*livelock*)

Propiedades de Corrección

- **Justicia** en el acceso a recursos compartidos (***fariness***):
  - **Espera FIFO:** Si un proceso quiere acceder a un recurso, lo hará antes de que lo haga otro proceso que lo solicite después que él **(cola del pan)**
  - **Espera lineal:** Si un proceso quiere acceder a un recurso, lo hará antes de que otro proceso lo haga **más de una vez (se cuela una vez, pero no dos)**
  - **Aleatorio:** No se considera justa, pero es la más eficiente (se usa cuando no se necesita justicia)

Memoria Compartida

code URJC

- SimpleConcurrent
- Intercalación de instrucciones: Indeterminismo
- Variables Compartidas
- Sincronización con Espera Activa
- Propiedades de corrección
- **Espera Activa vs Herramientas de sincronización**
- Introducción a los Semáforos
- Sincronización con Semáforos
- Sincronización avanzada
- Conclusiones

Memoria Compartida

- **Espera Activa vs Herramientas de sincronización**
  - **Problemas de la Espera Activa**
  - Herramientas de sincronización
  - Implementación de la Multiprogramación
  - Algoritmos no bloqueantes

Problemas de la Espera Activa

- Se han estudiado técnicas básicas de sincronización de procesos en el modelo de **Memoria Compartida con hilos y cerrojos** usando únicamente variables compartidas para la sincronización
- Se denomina **Espera Activa** porque los procesos están ejecutando instrucciones (están activos) incluso cuando **tienen que esperar** para poder continuar su ejecución

```
while (!continuar);
```

- También se la conoce como ***Busy waiting*** o ***spinning*** *o* ***polling*** (aunque este último término es más usado en entrada/salida)

Problemas de la Espera Activa

- La **Espera Activa** presenta los siguientes **problemas**
  - **Multiprogramación**
    - Los procesos que están esperando están malgastando el procesador que podría usarse por otros procesos que realmente estén realizando un trabajo útil
  - **Multiproceso**
    - Un procesador ejecutando instrucciones consume energía y por tanto disipa calor
    - Si las instrucciones no son útiles, el procesador podría estar en reposo

La Espera Activa es muy ineficiente y NO se debe usar en programas concurrentes

Problemas de la Espera Activa

- Para solucionar los problemas de la espera activa, se desarrollaron **herramientas de sincronización de procesos** en los procesadores, lenguajes de programación y librerías
- Con estas herramientas, cuando un proceso **no puede continuar** ejecutando las sentencias se **bloquea** y deja de ejecutar sentencias hasta que **otro proceso lo desbloquea** cuando se cumplen las **condiciones** para que siga ejecutando
- Esto permite **aprovechar** de forma mucho mas adecuada los recursos (capacidad de cómputo, energía, …)

Memoria Compartida

- **Espera Activa vs Herramientas de sincronización**
  - Problemas de la Espera Activa
  - **Herramientas de sincronización**
  - Implementación de la Multiprogramación
  - Algoritmos no bloqueantes

Herramientas de sincronización

- Puesto que la **espera activa** en general es **ineficiente**, se han diseñado herramientas que permiten **sincronizar** procesos de forma más **eficiente**
- Existen algunas herramientas de **bajo nivel** de abstracción y otras de **mayor nivel** de abstracción dependiendo de las necesidades del desarrollador
- En general lo mejor es usar las herramientas **de mayor nivel** que permiten implementar los **requisitos**

Herramientas de sincronización

- En la **programación funcional** y en la **programación orientada a objetos**, la gran mayoría de los lenguajes de programación implementan los mismos conceptos
  - **Funcional:** Funciones, listas, patrones…
  - **Orientación a Objetos:** Clases, objetos, métodos, atributos…
- En la **programación concurrente**, incluso en lenguajes que implementan el **mismo modelo**, pueden existir **diferencias sustanciales** entre las herramientas que ofrecen
- Algunas **herramientas básicas** suelen estar disponibles en todos los lenguajes

Herramientas de sincronización

- Herramientas de sincronización de procesos en el modelo de **Memoria Compartida**
  - **Semáforos**
  - Monitores
  - Regiones Críticas
  - Regiones Críticas Condicionales
  - Sucesos
  - Buzones
  - Recursos

Memoria Compartida

- **Espera Activa vs Herramientas de sincronización**
  - Problemas de la Espera Activa
  - Herramientas de sincronización
  - **Implementación de la Multiprogramación**
  - Algoritmos no bloqueantes

Implementación de la Multiprogramación

- Cuando se ejecuta un programa concurrente en **multiprogramación**, sólo un proceso puede estar ejecutándose en el procesador a la vez
- Para coordinar a todos los procesos que comparten el procesador
  - **Planificación (*Scheduling*)**
  - **Despacho (*Dispatching*)**

Implementación de la Multiprogramación

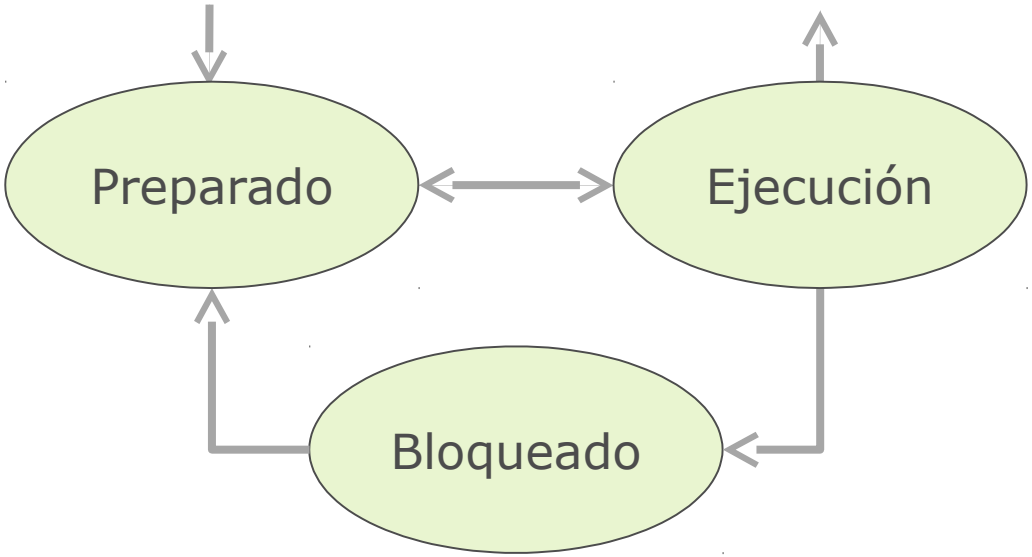
- **Planificación (*Scheduling*)**
  - Política que determina la asignación, en cada instante, de los procesos a los procesadores disponibles
    - **FIFO:** El primero que llegó
    - **Lineal:** No puede volver el mismo proceso hasta que no hayan ejecutado los demás
    - **Prioridades:** El más prioritario
    - **Aleatoria**

Implementación de la Multiprogramación

- **Despacho (*Dispatching*)**
  - **Configuración del procesador** para que ejecute el proceso que ha determinado la planificación
  - El proceso debe ejecutarse con las **mismas condiciones** en las que abandonó el procesador
  - Para ello el sistema dispone de un **Descriptor de Proceso (*Process Control Block*)** por cada proceso que alberga toda la información necesaria
    - Identificador del proceso (Process ID, PID)
    - Estado
    - Entorno, Memoria, …

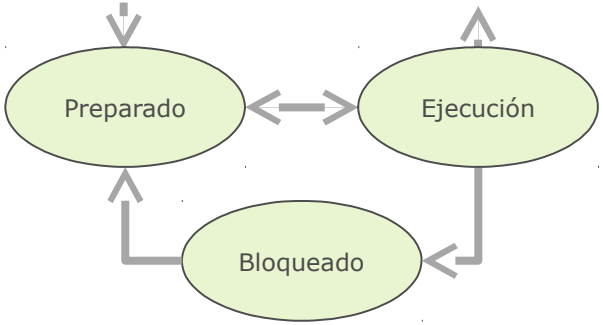
Implementación de la Multiprogramación

- Para implementar las **Herramientas de sincronización**, un proceso en un sistema en **multiprogramación** puede estar en los siguientes estados



Implementación de la Multiprogramación

- Un proceso estará en el **estado bloqueado** debido a diversos motivos:
  - Está **esperando por una operación de entrada/salida** y no puede continuar su ejecución hasta que finalice la operación
  - Ha ejecutado una instrucción de tipo **sleep()**
  - Usa una **herramienta de sincronización** (por ejemplo un semáforo) y está esperando a que se cumpla cierta condición para volver a estar preparado



Memoria Compartida

- **Espera Activa vs Herramientas de sincronización**
  - Problemas de la Espera Activa
  - Herramientas de sincronización
  - Implementación de la Multiprogramación
  - **Algoritmos no bloqueantes**

Algoritmos no bloqueantes

- Cada vez que un **proceso A** intenta acceder a un recurso bajo exclusión mutua y ya hay otro proceso usando el recurso, el proceso A se tiene que **bloquear**
- Cuando hay muchos procesos intentando acceder al mismo recurso, se dice que **tiene mucha competencia** (***contended***)
- Cuando eso ocurre, ese recurso compartido se convierte en un **cuello de botella** e impide que el programa pueda aprovechar todos los recursos (**escalar**)

Algoritmos no bloqueantes

- Cuando **muchos procesos** intentan acceder a **recursos compartidos** bajo **exclusión mutua** se pueden producir demasiados **bloqueos** y **desbloqueos** de procesos y esto también puede ser demasiado **ineficiente**
- Para evitarlo se **reduce al máximo el tamaño de la sección bajo exclusión mutua**, para reducir la probabilidad de que la sección bajo EM esté ocupada

Algoritmos no bloqueantes

- **Algoritmos no bloqueantes**
  - Otra técnica para evitar los bloqueos consiste en usar algoritmos que **no ponen los recursos bajo EM**
  - Para **evitar condiciones de carrera** al acceder a los recursos compartidos, usan instrucciones atómicas hardware de lectura y escritura de variables (***Comparse and Set***) **en bucles de espera activa**
  - A estos algoritmos se les llama: ***non-blocking algorithms*** y ***lock-free algorithms***

Algoritmos no bloqueantes

- Estos algoritmos se consideran avanzados y **no se van a estudiar en clase**
- Muchas **herramientas concurrentes** de Java están implementadas internamente usando esta **técnica avanzada**

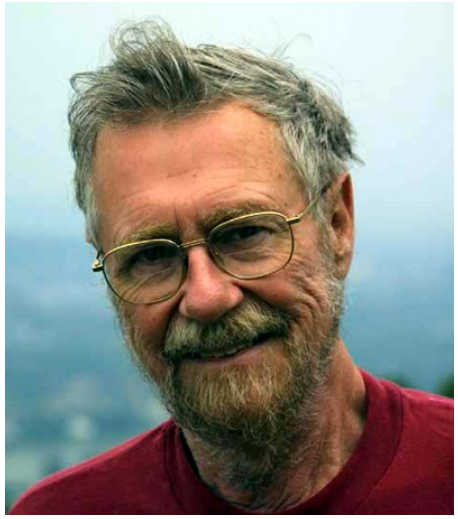
Memoria Compartida

code URJC

- SimpleConcurrent
- Intercalación de instrucciones: Indeterminismo
- Variables Compartidas
- Sincronización con Espera Activa
- Propiedades de corrección
- Espera Activa vs Herramientas de sincronización
- **Introducción a los Semáforos**
- Sincronización con Semáforos
- Sincronización avanzada
- Conclusiones

Introducción a los semáforos

- **Dijkstra** introdujo la primera herramienta de sincronización de procesos en 1968 y la denominó **Semáforo**
- Es una herramienta de **sincronización** de procesos de **bajo nivel** que permite implementar de forma sencilla la **sincronización condicional** y la **exclusión mutua**



Dijkstra
(1930-2002)

Introducción a los semáforos

- Un **semáforo** es una clase
- El estado interno está formado por un **contador de permisos** y un conjunto de **procesos bloqueados**
- Las operaciones permiten **bloquear y desbloquear** procesos dependiendo del número de **permisos**

Introducción a los semáforos

- En la librería **SimpleConcurrent** los semáforos se representan como objetos de la clase **SimpleSemaphore**
- Métodos públicos
  - **`public SimpleSemaphore(int permits)`**
    - Constructor que inicializa el número de permisos iniciales del semáforo con el valor indicado
  - **`void acquire()`** y **`void release()`**
    - Métodos invocados por los procesos para sincronizarse entre sí
    - Se ejecutan de forma atómica
    - Su comportamiento depende del número de permisos

Introducción a los semáforos

- **`void acquire()`**
  - El **proceso** que invoca este método intenta "**adquirir"** un permiso del semáforo. Si lo consigue, **continúa la ejecución**. Si no, queda **bloqueado**
  - Internamente el semáforo tiene un **contador de permisos** (**permits**).
  - Si el número de permisos del semáforo es mayor que cero (**permits>0**), se **decrementa** una unidad el número de permisos y el proceso **continúa** su ejecución

```
permits = permits - 1
```

  - Si el número de permisos del semáforo es cero (**permits=0**), el proceso suspende su ejecución, pasa al estado **bloqueado** y se añade al conjunto de procesos bloqueados en el semáforo
  - **No es posible** que un semáforo tenga un número **negativo de permisos**

Introducción a los semáforos

- `void release();`
  - El **proceso** que invoca este método **"libera"** un permiso **previamente adquirido**. Si otro proceso estaba **esperando** un permiso, lo **consigue** en la misma operación y se **desbloquea**.
  - El hilo que ejecuta este método **nunca queda bloqueado**, siempre continúa la ejecución
  - Si **no existen** procesos **bloqueados** en el semáforo, se **incrementa** el número de permisos.

```
permits = permits + 1
```

  - Si **existen** procesos **bloqueados** en el semáforo, **se desbloquea** aleatoriamente a uno cualquiera.

Introducción a los semáforos

code URJC

- Se puede pensar en un **semáforo** como una **caja llena de bolas**
  - El número de **permisos** representa el número de **bolas**
  - **acquire()**
    - El proceso que ejecuta **acquire()** tiene que **coger** una **bola** de la caja
    - Si hay **una o más** bolas, coge una y **continúa**
    - Si **no hay bolas**, se **bloquea** hasta que estén disponibles
  - **release()**
    - **Echa** una nueva bola a la caja
    - Si algún proceso estaba **esperando bola**, la **coge** y se desbloquea
    - Si no había **ningún proceso esperando**, la bola se **queda** en la caja

Introducción a los semáforos

Operación que ejecuta un proceso	Antes		Después	
	Permisos	Procesos Bloqueados	Permisos	Procesos Bloqueados
`acquire()`	3	Ninguno	2	Ninguno
`acquire()`	0	P1	0	P1,P2
`release()`	1	Ninguno	2	Ninguno
`release()`	0	Ninguno	1	Ninguno
`release()`	0	P1,P3	0	P1*

* Podría haberse desbloqueado cualquiera de los procesos

Introducción a los semáforos

- **Diferentes nombres** para las operaciones **acquire** y **release**
  - Las **operaciones** de gestión del semáforo reciben diferentes nombres dependiendo del **sistema operativo, lenguaje de programación** y/o **librería**.

acquire	release	Descripción
P	**V**	Los nombres que Dijkstra propuso originalmente a las operaciones en idioma holandés. V proviene de verhogen (incrementar) y P proviene de portmanteau prolaag (intentar reducir)
Down	**Up**	ALGOL 68, el kernel de Linux kernel y algunos libros de texto
Wait	**Signal**	PascalFC y algunos libros de texto
Pend	**Post**	
Procure	**Vacate**	Procure significa obtener y vacate desocupar

Introducción a los semáforos

- **Tipos de Semáforos**
  - Según el **número de permisos**
    - **Semáforos Binarios:** Como máximo sólo pueden gestionar un permiso
    - **Semáforos Generales:** Pueden gestionar cualquier número de permisos
  - Según la **política de desbloqueo** de procesos
    - **FIFO (*First In, First Out*):** Los procesos se desbloquean en orden de llegada
    - **Aleatorio:** Los procesos se desbloquean aleatoriamente

Los semáforos **SimpleSemaphore** son **generales aleatorios**

Memoria Compartida

- SimpleConcurrent
- Intercalación de instrucciones: Indeterminismo
- Variables Compartidas
- Sincronización con Espera Activa
- Propiedades de corrección
- Espera Activa vs Herramientas de sincronización
- Introducción a los Semáforos
- **Sincronización con Semáforos**
- Sincronización avanzada
- Conclusiones

Sincronización Condicional

- Se **produce** cuando un proceso debe **esperar** a que se cumpla una cierta **condición** para proseguir su ejecución
- Esta **condición** sólo puede ser **activada** por otro **proceso**

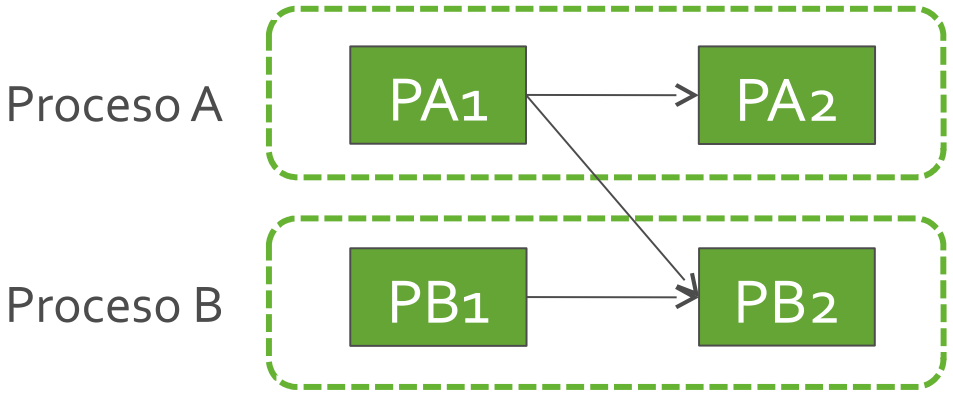
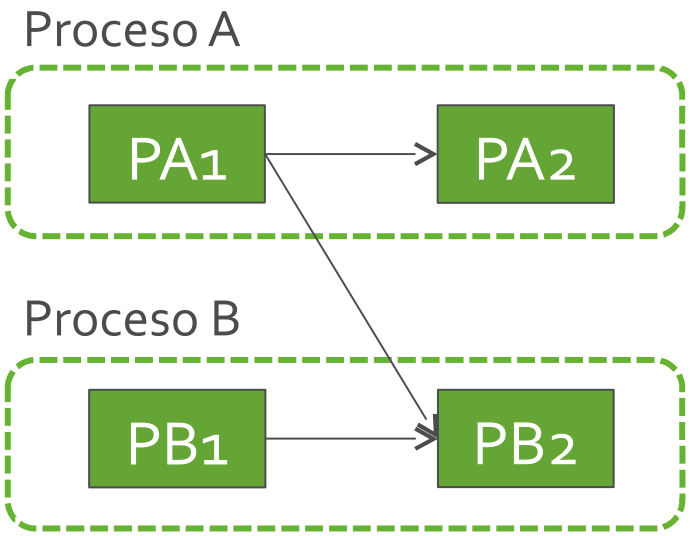


Diagrama de Precedencia

Sincronización Condicional

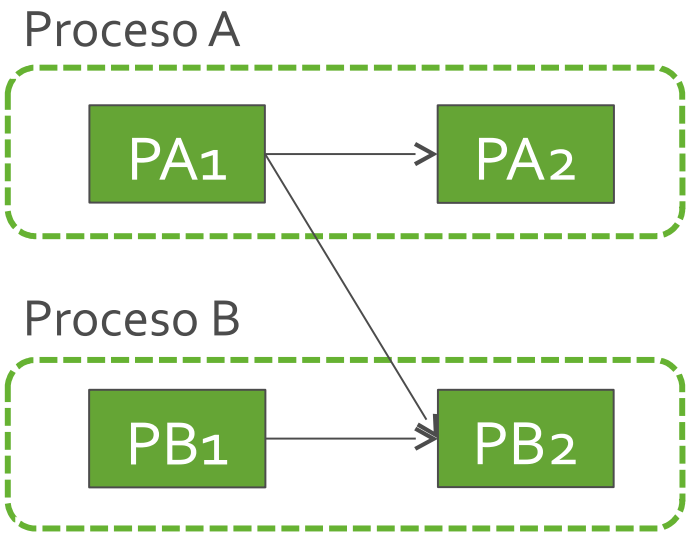
- Implementación con **Espera Activa**




```
public class SincCond {

   static volatile boolean continuar;

   public static void a() {
      print("PA1 ");
      continuar = true;
      print("PA2 ");
   }

   public static void b() {
      print("PB1 ");
      while (!continuar);
      print("PB2 ");
   }

   public static void main(String[] args){

      continuar = false;

      createThread("a");
      createThread("b");

      startThreadsAndWait();
   }
}
```

Sincronización Condicional

- Implementación con **Semáforos**




```
public class SincCondSem {

   static SimpleSemaphore continuar;

   public static void a() {
      print("PA1 ");
      continuar.release();
      print("PA2 ");
   }

   public static void b() {
      print("PB1 ");
      continuar.acquire();
      print("PB2 ");
   }

   public static void main(String[] args){

      continuar = new SimpleSemaphore(0);

      createThread("a");
      createThread("b");

      startThreadsAndWait();
   }
}
```

Download this code

Espera Activa (Ineficiente)

```
public class SincCond {

   static volatile boolean continuar;

   public static void a() {
      print("PA1 ");
      continuar = true;
      print("PA2 ");
   }

   public static void b() {
      print("PB1 ");
      while (!continuar);
      print("PB2 ");
   }

   public static
      void main(String[] args){

      continuar = false;

      createThread("a");
      createThread("b");

      startThreadsAndWait();
   }
}
```

Semáforos (Eficiente)

```
public class SincCondSem {

   static SimpleSemaphore continuar;

   public static void a() {
      print("PA1 ");
      continuar.release();
      print("PA2 ");
   }

   public static void b() {
      print("PB1 ");
      continuar.acquire();
      print("PB2 ");
   }

   public static
      void main(String[] args){

      continuar = new SimpleSemaphore(0);

      createThread("a");
      createThread("b");

      startThreadsAndWait();
   }
}
```

Sincronización Condicional

- Comportamiento de los **procesos** con **herramientas de sincronización (bloqueantes)**
  - **Un proceso se bloquea a sí mismo**
    - Un proceso se **bloquea a sí mismo** si no puede proseguir su ejecución
    - Un proceso **nunca bloquea a otro proceso**
  - **Un proceso desbloquea a otro**
    - Un proceso **nunca se desbloquea** a sí mismo
    - Un proceso **desbloquea a otro** proceso cuando ese otro proceso puede **proseguir** su ejecución

Exclusión Mutua

- **El problema de la Exclusión Mutua**
  - Se tienen dos o más procesos concurrentes, que ejecutan indefinidamente una secuencia de instrucciones dividida en dos secciones: **sección bajo Exclusión Mutua y la sección sin EM**

```
while(true) {

   // Sección bajo EM
   printlnI("P1_EM1");
   printlnI("P1_EM2");

   // Sección sin EM
   printlnI("P1_1");
   printlnI("P1_2");
}
```

Exclusión Mutua

- La exclusión mutua con **espera activa** con **SimpleConcurrent** se implementa usando **enterMutex()** y **exitMutex()**
- Implementada con **semáforos** es **mucho más eficiente**

```
public class ExcMutuaSem {


   public static void p() {

      for (int i = 0; i < 5; i++) {

         // Sección bajo EM
         enterMutex();
         println("P1_EM1");
         println("P1_EM2");
         exitMutex();

         // Sección sin EM
         println("P1_1");
         println("P2_2");
      }
   }

   public static void main(String[] args) {

      createThreads(2,"p");
      startThreadsAndWait();
   }
}
```

Exclusión Mutua

- Para **entrar en la sección bajo EM** hay que **coger una bola** de la caja, y **dejarla al salir** para que la pueda coger el próximo proceso que quiera entrar
- Cuando el **semáforo tiene 1 permiso** (*permits*=1), la sección crítica **está libre**
- Cuando el **semáforo no tiene permisos** (*permits*=0), la sección crítica **está ocupada** por un proceso

```
public class ExcMutuaSem {

   private static SimpleSemaphore em;

   public static void p() {

      for (int i = 0; i < 5; i++) {

         // Sección bajo EM
         em.acquire();
         printlnI("P1_EM1");
         printlnI("P1_EM2 ");
         em.release();

         // Sección sin EM
         printlnI("P1_1");
         printlnI("P1_2");
      }
   }

   public static void main(String[] args) {
      em = new SimpleSemaphore(1);
      createThreads(2,"p");
      startThreadsAndWait();
   }
}
```

Download this code

Ejercicio 13A

- Una **línea del metro** está formada por **varios tramos de vía** que son recorridos **secuencialmente** en un único sentido por **diferentes** trenes
- Cada **tren** está representado por un **hilo** que realiza las siguientes **operaciones**:

```
public static void tren(int numTren) {

   sleepRandom(500);
   recorrerTramoA(numTren);

   sleepRandom(500);
   recorrerTramoB(numTren);

   sleepRandom(500);
   recorrerTramoC(numTren);
}
```

Ejercicio 13A

```
public class Metro {

  private static final int NUM_TRENES = 5;

  public static void tren(int numTren) {
    ...
  }

  private static void recorrerTramoA(int numTren) {
    printlnI("Entra TA T" + numTren);
    sleepRandom(500);
    printlnI("Sale TA T" + numTren);
  }

  public static void main(String args[]) {
    for (int i = 0; i < NUM_TRENES; i++) {
      createThread("tren", i);
    }
    startThreadsAndWait();
  }
}
```

Ejercicio 13A

- El funcionamiento actual es:
  - **Varios trenes** pueden estar en el **mismo tramo** a la vez
  - Unos trenes pueden **adelantar** a otros
- Se quiere el siguiente funcionamiento:
  - Cada tramo sólo pueda estar ocupado por **un tren en cada instante**
  - Los trenes **nunca pueden adelantarse** unos a otros
- Para ello únicamente hay que añadir **herramientas de sincronización** de procesos (semáforos)

Ejercicio 13B

- Se quiere hacer el **código genérico** para que el **número de tramos** se pueda especificar con una constante

Memoria Compartida

- SimpleConcurrent
- Intercalación de instrucciones: Indeterminismo
- Variables Compartidas
- Sincronización con Espera Activa
- Propiedades de corrección
- Espera Activa vs Herramientas de sincronización
- Introducción a los Semáforos
- Sincronización con Semáforos
- **Sincronización avanzada**
- Conclusiones

Sincronización Avanzada

- Todo programa concurrente se puede implementar con **sincronizaciones condicionales** y **exclusiones mutuas**
- En algunas ocasiones este tipo de sincronizaciones son **muy básicas**, de **muy bajo nivel**
- Existen otras formas de sincronización **más avanzadas**, de más **alto nivel**

Sincronización Avanzada

- Veremos varios ejemplos de sincronizaciones avanzadas:
  - **Exclusión Mutua Generalizada:** En la zona bajo exclusión mutua puede haber varios procesos (no sólo 1)
  - **Comunicación con Buffer:** Varios procesos se comunican entre sí usando una estructura de datos intermedia para no bloquearse.

Exclusión Mutua Generalizada

- **Tipos de Exclusión Mutua generalizada:**
  - **Exclusión Mutua con varios procesos:** Sin ninguna restricción salvo el número de ellos
  - **Lectores – Escritores:** Pueden ejecutar instrucciones múltiples procesos lectores o un único proceso escritor (se verá como ejercicio)
  - **Específicos del problema:** Cada problema puede tener sus propias restricciones

Exclusión Mutua Generalizada

- **Exclusión Mutua con varios procesos:**
  - Se tiene cuando **más de un proceso** puede ejecutar la **sección bajo exclusión mutua**
  - Si el número de procesos es **K**, se implementa con un semáforo con un **valor inicial de K**

Exclusión Mutua con varios proc

Para entrar en la **sección bajo EM** hay que **coger una bola** de la caja, y **dejarla al salir** para que la pueda coger **otro proceso**

El **contador** del semáforo indica los **huecos libres** de la sección crítica

```
public class ExcMutuaGenSem {

private static SimpleSemaphore em;

   public static void p() {

      for (int i = 0; i < 5; i++) {

         // Sección bajo EM
         em.acquire();
         printlnI("P_EM1");
         printlnI("P_EM2");
         em.release();

         // Sección sin EM
         printlnI("P_1");
         printlnI("P_2");
      }
   }

   public static void main(String[] args) {
      em = new SimpleSemaphore(3);
      createThreads(5,"p");
      startThreadsAndWait();
   }
}
```

Comunicación con Buffer

- En los ejercicios de **productores y consumidores** los procesos se comunicaban con **una variable** de tipo primitivo
- Para **adaptar las velocidades diferentes** de los productores y consumidores y que **no haya esperas innecesarias** se utilizan **buffers** para almacenar **temporalmente** información
- Un **buffer** permite que se pueda ir **insertando información** aunque no esté preparado el proceso encargado de consumirla
- Los **buffers** se usan mucho en informática:
  - **Sistemas operativos**: teclado, red, escritura en disco, etc.
  - **Aplicaciones distribuidas:** Comunicación entre diferentes nodos de la red

Productores Consumidores

- Para estudiar la comunicación con buffers implementaremos el **"Problema de los Productores y los Consumidores"**
- **Procesos**
  - Los **Productores** son procesos que generan datos
  - Los **Consumidores** son procesos que consumen los datos en el orden en que se generan

Productores Consumidores

- **Restricciones**
  - Cada **productor** genera un **único** dato cada vez
  - Un **consumidor** consume un **dato** generado por el productor cada vez
  - Todos los productos **se consumen**
  - Si hay **varios consumidores**, ningún producto se consume dos veces

Productores Consumidores

- **Comunicación**
  - Se utilizará un **array** o una **lista** para almacenar **temporalmente** los datos producidos antes de ser consumidos
- **Sincronización**
  - Los **consumidores** deben de **bloquearse** cuando **no tengan datos** que consumir (**array vacío**)
  - Los **productores** deben **bloquearse** cuando no puedan insertar más datos en el buffer (**array lleno**)

Productores Consumidores

Esquema de la aplicación

Hay 5 hilos **productores** que generan 20 productos y finalizan

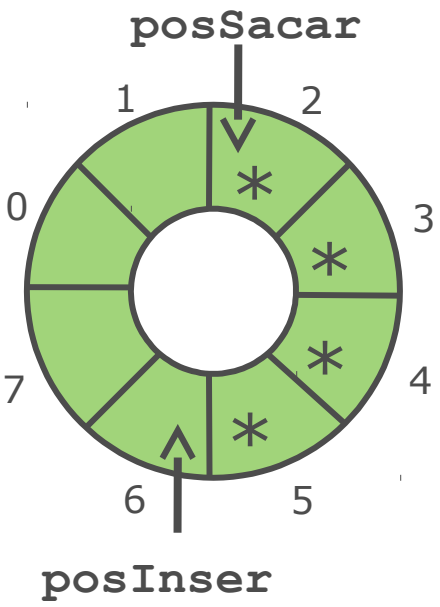
Hay 3 hilos **consumidores** que consumen 20 productos y finalizan

```
public class ProdConsBufferMal {

   //Atributos y métodos del buffer
   ...

   //Hilos y main
   public static void productor() {
      for(int i=0; i<20; i++){
         sleepRandom(500);
         insertarBuffer(i);
      }
   }

   public static void consumidor() {
      for(int i=0; i<20; i++){
         String dato = sacarBuffer();
         sleepRandom(500);
         print(dato);
      }
   }

   public static void main(String[] args) {
   createThreads(5, "productor");
      createThreads(3, "consumidor");
      startThreadsAndWait();
   }
}
```

Download this code

Productores Consumidores

Implementación del buffer con un array circular




```
class ProdConsBufferMal {

   //Atributos y métodos del buffer
   int[] datos = new int[10];
   int posInser, posSacar = 0;

   public static void insertarBuffer(int dato) {
      datos[posInser] = dato;
      posInser = (posInser+1) % 10;
   }

   public static int sacarBuffer() {
      int dato = datos[posSacar];
      posSacar = (posSacar+1) % 10;
      return dato;
   }

   //Hilos y main
   ...

}
```

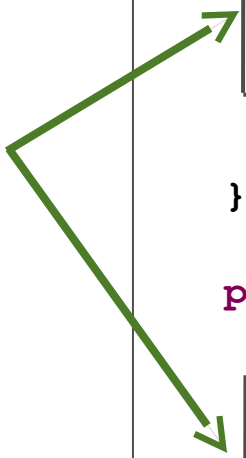
Download this code

Productores Consumidores

- El programa no es correcto porque falta incorporar los **puntos de sincronización**
  - **Sincronización Condicional**
    - Un productor se **bloqueará** antes de insertar un dato si el buffer está **lleno**
    - Un consumidor se **bloqueará** antes de leer un dato si el buffer está **vacío**
  - **Exclusión Mutua**
    - Las **variables de control del buffer** deben estar bajo exclusión mutua

Productores Consumidores

Cada **contador** se protege bajo su **propio** semáforo de **exclusión mutua**



```
public class ProdConsBuffer {

   //Atributos y métodos del buffer
   SimpleSemaphore nHuecos = new SimpleSemaphore(10);
   SimpleSemaphore nProductos = new SimpleSemaphore(0);
   SimpleSemaphore emPosInser = new SimpleSemaphore(1);
   SimpleSemaphore emPosSacar = new SimpleSemaphore(1);
   int[] datos = new int[10];
   int posInser, posSacar = 0;

   public static void insertarBuffer(int dato) {
      nHuecos.acquire();
      emPosInser.acquire();
      datos[posInser] = dato;
      posInser = (posInser+1) % 10;
      emPosInser.release();
      nProductos.release();
   }

   public static int sacarBuffer() {
      nProductos.acquire();
      emPosSacar.acquire();
      int dato = datos[posSacar];
      posSacar = (posSacar+1) % 10;
      emPosSacar.release();
      nHuecos.release();
      return dato;
   }
   //Hilos y main...
}
```

Download this code

Productores Consumidores

El proceso que quiere **sacar** queda **bloqueado** si no hay productos

Hay tantos **permisos** como **productos**

```
public class ProdConsBuffer {

   //Atributos y métodos del buffer
   SimpleSemaphore nHuecos = new SimpleSemaphore(10);
   SimpleSemaphore nProductos = new SimpleSemaphore(0);
   SimpleSemaphore emPosInser = new SimpleSemaphore(1);
   SimpleSemaphore emPosSacar = new SimpleSemaphore(1);
   int[] datos = new int[10];
   int posInser, posSacar = 0;

   public static void insertarBuffer(int dato) {
      nHuecos.acquire();
      emPosInser.acquire();
      datos[posInser] = dato;
      posInser = (posInser+1) % 10;
      emPosInser.release();
      nProductos.release();
   }

   public static int sacarBuffer() {
      nProductos.acquire();
      emPosSacar.acquire();
      int dato = datos[posSacar];
      posSacar = (posSacar+1) % 10;
      emPosSacar.release();
      nHuecos.release();
      return dato;
   }
   //Hilos y main...
}
```

Download this code

Productores Consumidores

El proceso que quiere **insertar** queda **bloqueado** si no hay **huecos**

Hay tantos **permisos** como **huecos**

```
public class ProdConsBuffer {

   //Atributos y métodos del buffer
   SimpleSemaphore nHuecos = new SimpleSemaphore(10);
   SimpleSemaphore nProductos = new SimpleSemaphore(0);
   SimpleSemaphore emPosInser = new SimpleSemaphore(1);
   SimpleSemaphore emPosSacar = new SimpleSemaphore(1);
   int[] datos = new int[10];
   int posInser, posSacar = 0;

   public static void insertarBuffer(int dato) {
      nHuecos.acquire();
      emPosInser.acquire();
      datos[posInser] = dato;
      posInser = (posInser+1) % 10;
      emPosInser.release();
      nProductos.release();
   }

   public static int sacarBuffer() {
      nProductos.acquire();
      emPosSacar.acquire();
      int dato = datos[posSacar];
      posSacar = (posSacar+1) % 10;
      emPosSacar.release();
      nHuecos.release();
      return dato;
   }
   //Hilos y main...
}
```

Download this code

Memoria Compartida

- SimpleConcurrent
- Intercalación de instrucciones: Indeterminismo
- Variables Compartidas
- Sincronización con Espera Activa
- Propiedades de corrección
- Espera Activa vs Herramientas de sincronización
- Introducción a los Semáforos
- Sincronización con Semáforos
- Sincronización avanzada
- **Conclusiones**

Conclusiones

- **Espera activa vs Herramientas de sincronización**
  - La **espera activa** es una técnica que no debe usarse **nunca** para desarrollar programas concurrentes porque es **ineficiente**
  - Deben usarse **herramientas de sincronización** bloqueantes para **aprovechar** mejor los **recursos** de cómputo.
  - En general es mejor usar las herramientas de **alto nivel** que proporcionan las tecnologías de desarrollo (lenguajes + librerías) que pueden estar implementadas con **algoritmos no bloqueantes**

Conclusiones

- **Ventajas de los Semáforos**
  - Son muy **fáciles de entender**
  - Permiten **sincronizar** procesos de forma **sencilla** y **eficiente**
  - Están **presentes** en la mayoría de las tencologías de desarrollo **(lenguajes+librerías)**

Conclusiones

- **Desventajas de los Semáforos**
  - Son primitivas de muy **bajo nivel**
  - Omitir una llamada a **`release()`** puede provocar **interbloqueo**
  - Omitir una llamada a **`acquire()`** puede provocar condiciones de carreara **(por no estar bajo exclusión mutua)**
  - No **estructuran el código** del programa, lo que hace que los códigos sean difíciles de mantener y de eliminar los errores que se produzcan

Conclusiones

- Para solventar los **problemas de los semáforos**, existen otras herramientas de sincronización de procesos con **mayor nivel de abstracción**
  - **Monitores (presentes en Java)**
  - Regiones Críticas
  - Regiones Críticas Condicionales
  - Sucesos
  - Buzones
  - Recursos

Conclusiones

- **Programación concurrente con memoria compartida:** Difícil de implementar programas concurrentes y difícil de corregir errores
  - **Condiciones de carrera:** Por no poner bajo EM los recursos compartidos.
  - **Interbloqueos:** Cuando dos procesos se bloquean esperando que el otro los desbloquee.
  - **Aprovechamiento de recursos:** Cuando el programa no es lo suficientemente concurrente y no aprovecha todos los procesadores disponibles o emplea demasiado tiempo sincronizando los procesos.

Conclusiones

- Popularización de **otros modelos de programación concurrente**:
  - Paso de mensajes:
    - Actores: Librerías para Java (Akka, Vert.x, Reactor)
    - CSP: Lenguaje de programación Go!
  - Modelos asíncronos (sin bloqueo explícito) :
    - JavaScript, Vert.x, Akka…

14. Sincronización de Barrera

- La **Sincronización de Barrera** (*Barrier*) es una sincronización condicional en la que los procesos tienen que esperar a que el resto de procesos lleguen al mismo punto para poder continuar su ejecución
- Vamos a estudiar este tipo de sincronización con los siguientes requisitos
  - Programa con N procesos
  - Cada proceso escribe la letra **`A`** y luego la **`B`**
  - Los procesos tienen que esperar que todos hayan escrito la letra **`A`** antes de escribir la **`B`**

Sincronización de Barrera

1ª Aproximación Incorrecta

Puede provocar interbloqueo (*deadlock*) porque **`nProcesos`** no está bajo exclusión mutua y se pueden producir errores al contar

```
public class Ejer14_SincBarrera_Mal1 {

   private static final int NPROCESOS = 3;

   private static volatile int nProcesos;
   private static SimpleSemaphore sb;

   public static void proceso() {
      print("A");
      nProcesos++;
      if (nProcesos < NPROCESOS) {
         sb.acquire();
      } else {
         for (int i=0; i<NPROCESOS - 1; i++) {
            sb.release();
         }
      }
      print("B");
   }

   public static void main(String[] args) {
      nProcesos = 0;
      sb = new SimpleSemaphore(0);
      createThreads(NPROCESOS, "proceso");
      startThreadsAndWait();
   }
}
```

Download this code

Sincronización de Barrera

2ª Aproximación Incorrecta

Si se deja la consulta del contador fuera de la Exclusión Mutua, puede ocurrir que dos procesos hagan **`release()`**

```
private static final int NPROCESOS = 3;
private static volatile int nProcesos;
private static SimpleSemaphore sb;
private static SimpleSemaphore emNProcesos;

public static void proceso() {
      print("A");
      emNProcesos.acquire();
      nProcesos++;
      emNProcesos.release();

      if (nProcesos < NPROCESOS) {
         sb.acquire();
      } else {
         for (int i=0; i<NPROCESOS - 1; i++) {
            sb.release();
         }
      }
      print("B");
   }

   public static void main(String[] args) {
      nProcesos = 0;
      sb = new SimpleSemaphore(0);
      emNProcesos = new SimpleSemaphore(1);
      createThreads(NPROCESOS, "proceso");
      startThreadsAndWait();
   }
}
```

Download this code

Sincronización de Barrera

Solución Correcta

Si el proceso no es el último, libera la EM y se bloquea

Si es el último, sale de la EM y desbloquea a los demás procesos

```
public class Ejer14_SincBarrera {
    private static final int NPROCESOS = 3;
    private static volatile int nProcesos;
    private static SimpleSemaphore sb;
    private static SimpleSemaphore emNProcesos;

    public static void proceso() {
        print("A");
        emNProcesos.acquire();
        nProcesos++;
        if (nProcesos < NPROCESOS) {
            emNProcesos.release();
            sb.acquire();
        } else {
            emNProcesos.release();
            for (int i=0; i< NPROCESOS-1; i++) {
                sb.release();
            }
        }
        print("B");
    }

    public static void main(String[] args) {
        nProcesos = 0;
        sb = new SimpleSemaphore(0);
        emNProcesos = new SimpleSemaphore(1);
        createThreads(NPROCESOS, "proceso");
        startThreadsAndWait();
    }
}
```

Download this code

15. Sincronización de Barrera Cíclica

code URJC

- Se pide implementar un programa concurrente con las siguientes características:
  - El programa tendrá **4 procesos** que escriben una única letra indefinidamente.
  - Un proceso escribirá la letra A, otra la B, otra la C y otro la D.
  - Cada proceso **escribe su letra y espera a que los demás** hayan escrito también su letra para poder continuar.
  - Uno de los procesos tiene que escribir un **guión** – después de que todos hayan escrito su letra y antes de que empiecen a escribirlas de nuevo.
- La salida por pantalla del programa sería:
  - **ACDB-BACD-DCBA-ABCD-BCAD …**

Sincronización de Barrera Cíclica

1ª Aproximación Incorrecta

Un proceso puede ser desbloqueado, mostrar su letra, volver a entrar en **sincronizacion()** ejectuar **sb.acquire**(), ser desbloqueado de nuevo (con un permiso que no estaba destinado para él) y volver a escribir su letra. Todo ello antes de que todos lo procesos de la primera iteración se hayan desbloqueado

```
public static void procesoA() {
    while(true){
        print("A");
        sincronizacion();
    }
}

public static void sincronizacion(){

    emNProcesos.acquire();
    nProcesos++;
    if (nProcesos < 4) {
        emNProcesos.release();
        sb.acquire();

    } else {

        println("-");

        nProcesos = 0;
        emNProcesos.release();

        for (int i = 0; i < 3; i++) {
            //sleepRandom(500); Condición carrera
            sb.release();
        }
    }
}
```

Download this code

Sincronización de Barrera Cíclica

2ª Aproximación Incorrecta

Se protege la entrada de los procesos a la segunda iteración hasta que todos los de la primera han sido desbloqueados.

El problema es que no podemos garantizar que han llegado a bloquearse en "sb" antes de liberar la exclusión mutua.

```
public static void procesoA() {
    while(true){
        print("A");
        sincronizacion();
    }
}

public static void sincronizacion(){

    emNProcesos.acquire();
    nProcesos++;
    if (nProcesos < 4) {
        emNProcesos.release();
        //sleepRandom(500); Condición carrera
        sb.acquire();

    } else {

        println("-");
        nProcesos = 0;

        for (int i = 0; i < 3; i++) {
            sb.release();
        }
        emNProcesos.release();
    }
}
```

Download this code

Sincronización de Barrera Cíclica

Solución

Obligando a que los procesos desbloqueados señalizen que les ha dado tiempo a bloquearse y desbloquearse.

De esta forma se garantiza que un proceso no se adelanta a los demás

El nuevo semáforo **desbloqueo** se inicializa a o porque es de sincronización condicional

```
public static void procesoA() {
    while(true){
        print("A");
        sincronizacion();
    }
}

public static void sincronizacion() {

    emNProcesos.acquire();
    nProcesos++;
    if (nProcesos < 4) {
        emNProcesos.release();
        sleepRandom(500);
        sb.acquire();
        desbloqueo.release();
    } else {

        println("-");
        nProcesos = 0;

        sb.release(3);
        desbloqueo.acquire(3);
        emNProcesos.release();
    }
}
```

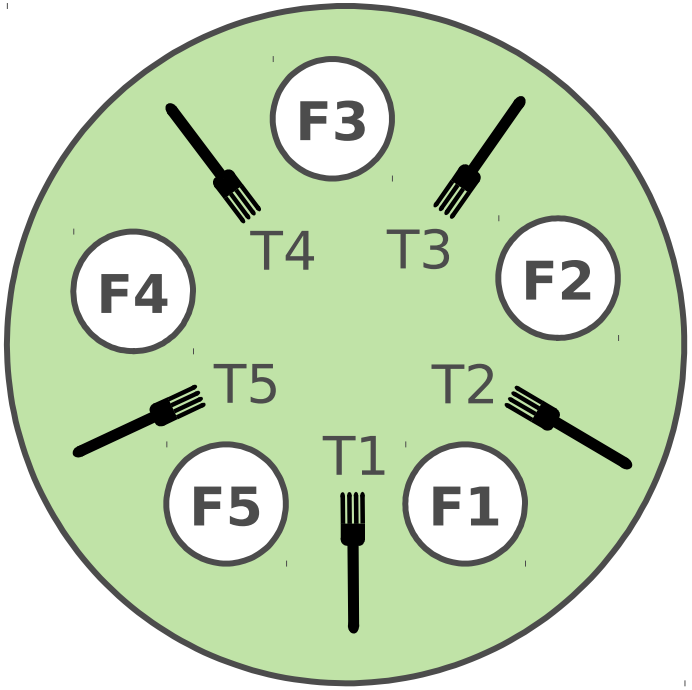
Download this code

16. Descarga de Ficheros

- Se desea ampliar el ejercicio de descarga de ficheros para descargar 10 ficheros.
- Cuando los procesos hayan terminado de descargar un fichero, se esperen a que el último proceso muestre por pantalla el fichero para comenzar a descargar un nuevo fichero

17. Filósofos Comilones

- 5 Filósofos que piensan libremente y comen en una mesa con 5 platos
- Cada plato está asignado a un filósofo
- Los platos se reponen continuamente
- Hay 5 tenedores, uno entre cada par de platos adyacentes



17. Filósofos Comilones

- **Procesos (Filósofos)**
  - El filósofo inicialmente piensa
  - Se sienta delante de su plato y toma de uno en uno los tenedores situados a ambos lados de su plato
  - Come
  - Cuando finaliza, deja los dos tenedores en su posición original
  - Todo filósofo que come, en algún momento se harta y termina

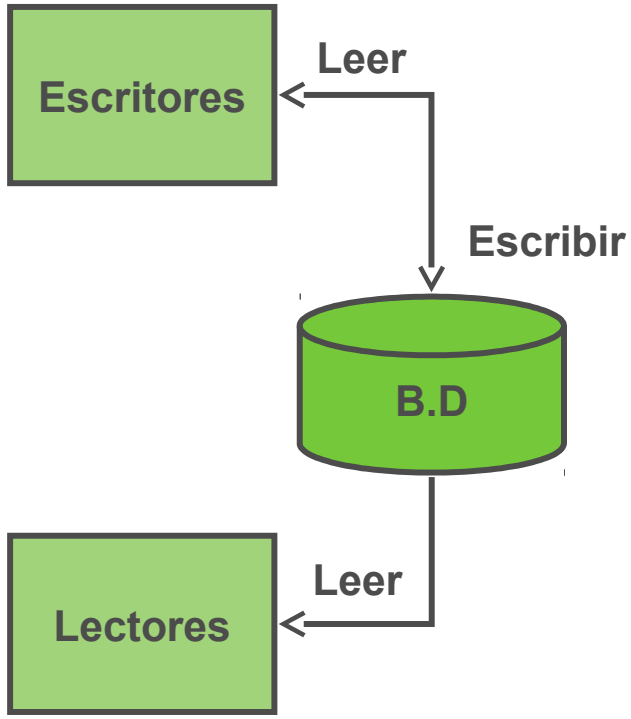
```
public static void
    filosofo(int numFilosofo) {

    while (true) {
        printlnI("Pensar");
        // Obtener tenedores
        printlnI("Comer");
        // Liberar tenedores
    }
}
```

17. Filósofos Comilones

- **Restricciones**
  - Un filósofo sólo puede comer cuando tenga los dos tenedores
  - Los tenedores se cogen y se dejan de uno en uno
  - Dos filósofos no pueden tener el mismo tenedor simultáneamente (EM de acceso al tenedor)
  - Si varios filósofos tratan de comer al mismo tiempo, uno de ellos debe conseguirlo (Ausencia Interbloqueo)
  - Si un filósofo desea comer y tiene competencia, en algún momento lo deberá poder hacer (Ausencia de Inanición)
  - En ausencia de competencia, un filósofo que quiera comer deberá hacerlo sin retrasos innecesarios (Ausencia retrasos innecesarios)

19. Lectores Escritores

- **Acceso Combinado (Concurrente-Exclusivo) a variables compartidas**
- Dos tipos de procesos que comparten una misma Base de Datos: los Lectores y los Escritores
- Los Lectores pueden leer registros de la BD
- Los Escritores pueden leer y escribir los registros de la BD



19. Lectores Escritores

- **Restricciones**
  - Cualquier número de lectores puede acceder a la vez a la BD, si no hay escritores accediendo
  - El acceso a la BD de los escritores es exclusivo. Mientras haya algún lector leyendo, ningún escritor puede acceder a la BD, pero otros lectores sí.
  - Se puede tener varios escritores trabajando, aunque estos se deberán sincronizar para que la escritura se lleve a cabo de uno en uno
  - Se da prioridad a los escritores. Ningún lector puede acceder a la BD cuando haya escritores que desean hacerlo (Inanición de Lectores)

19. Lectores Escritores

- Procesos

```
public static void lector() {
   while(true){
      inicioLectura();
      println("Leer datos");
      finLectura();
      println("Procesar datos");
   }
}
```

```
public static void escritor()
{
   while (true) {
      println("Generar datos");
      inicioEscritura();
      println("Escribir datos");
      finEscritura();
   }
}
```