20151026

dit
UPM

Programación concurrente — Datos compartidos y exclusión mutua

Juan Antonio de la Puente <jpuente@dit.upm.es>

ETSIT UPM

Referencias

- Scott Oaks & Henry Wong
  Java Threads
  O'Reilly Media; 3rd ed (2004)

- Kathy Sierra & Bert Bates
  Head First Java, ch. 15
  O'Reilly Media; 2nd ed (2005)

Datos compartidos

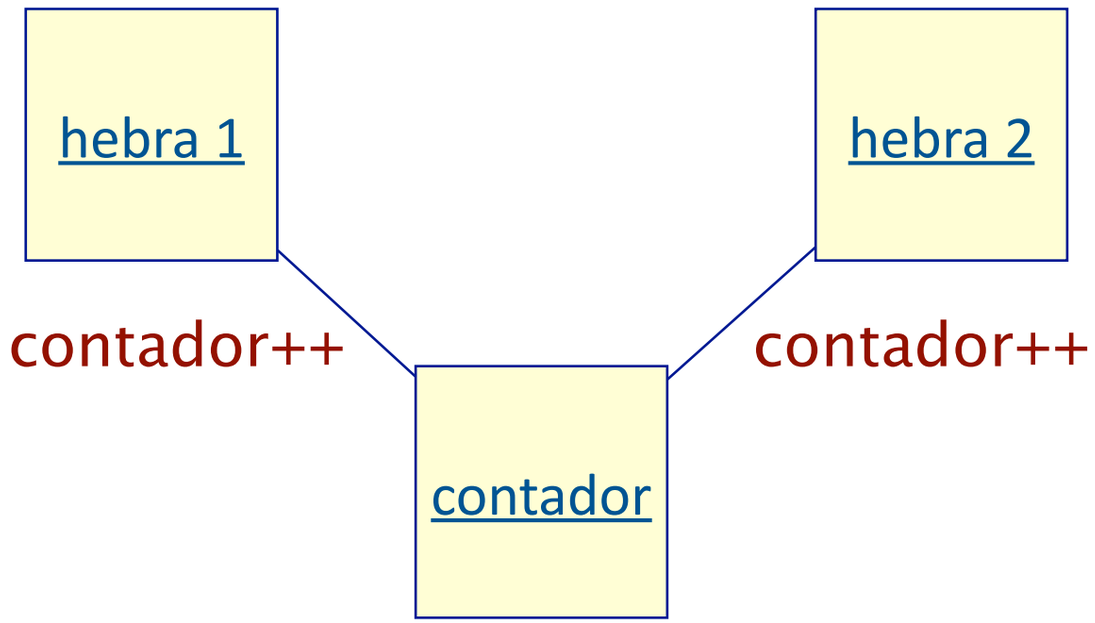
Variables compartidas

- Las hebras de un programa pueden ser independientes
  - la ejecución de una hebra no afecta a lo que hagan las demás hebras
- … pero a menudo es más interesante que varias hebras cooperen para realizar una función común

- Un mecanismo de cooperación muy corriente consiste en usar variables compartidas
  - variables a las que tienen acceso varias hebras
  - las hebras pueden consultar (leer) el valor de la variable en un momento determinado o modificarlo (escribir)

Ejemplo

- Dos hebras que incrementan concurrentemente el valor de un contador



- Problema: el resultado depende de en qué orden se entrelace la ejecución de las operaciones de cada hebra
  - esto se llama condición de carrera

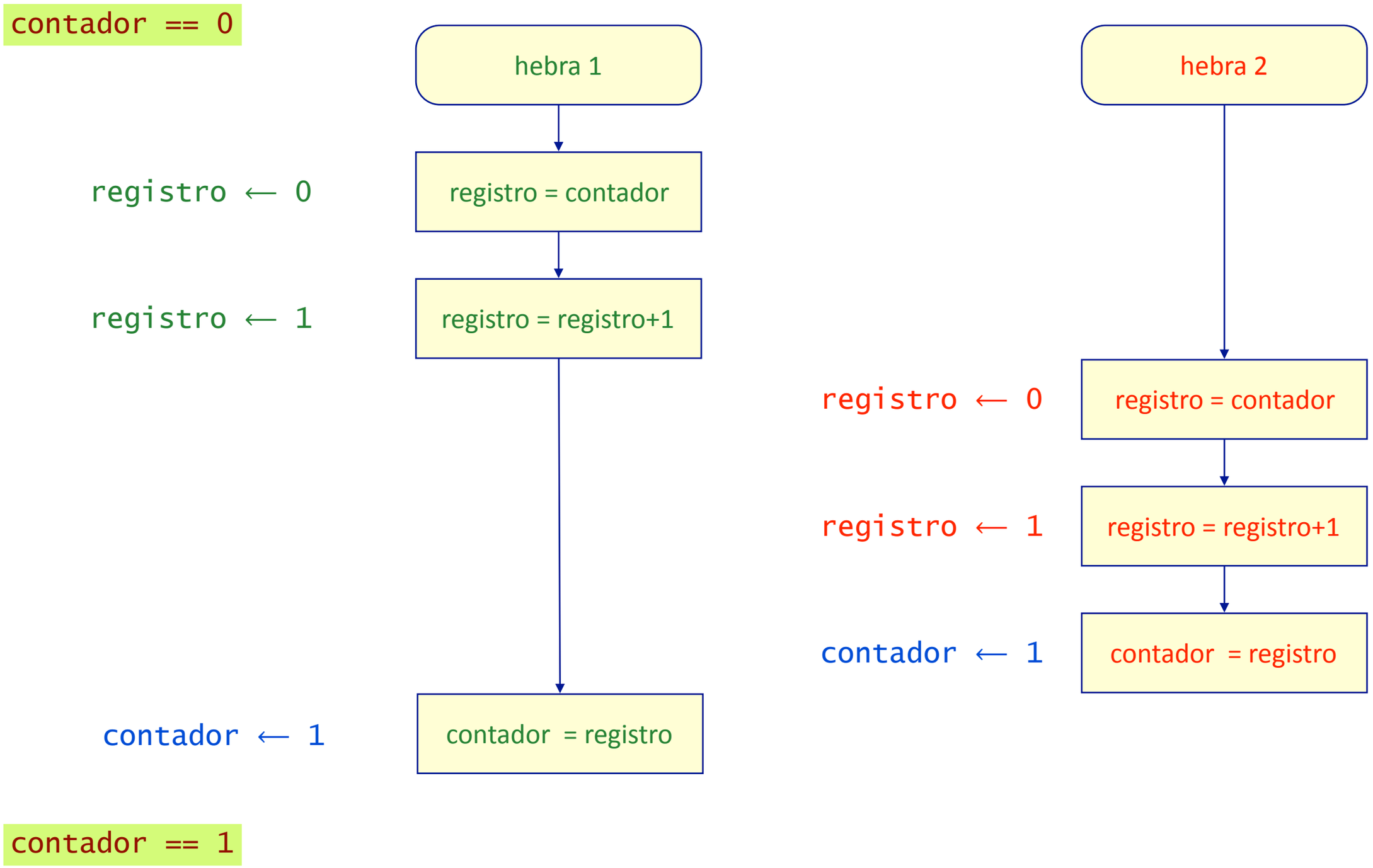
Operaciones atómicas

- La operación contador++ no se ejecuta de una sola vez
  - se descompone en

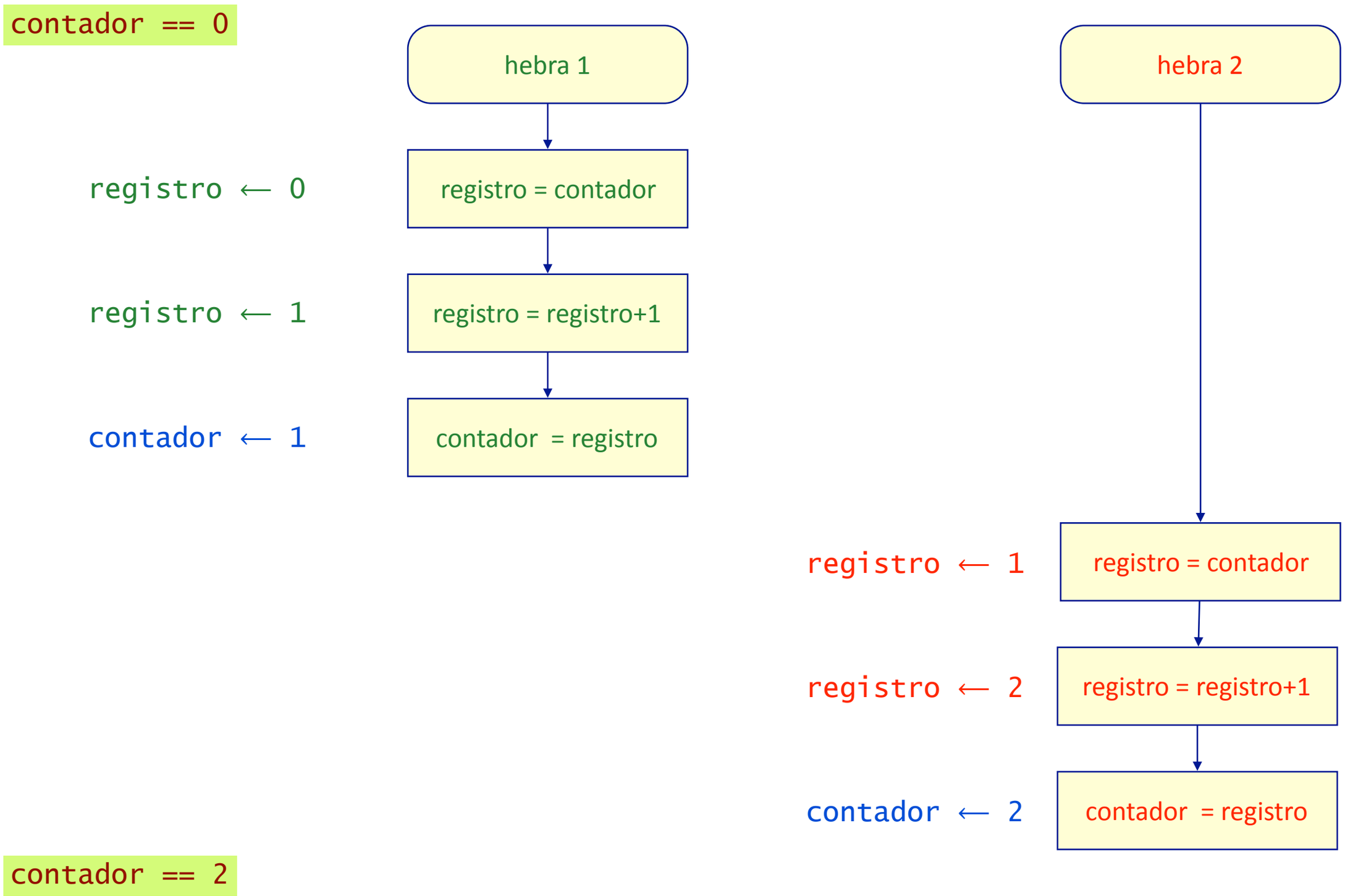
```
registro = contador       // registro es una variable temporal local de cada thread
registro = registro + 1   // incrementar el valor del registro
contador = registro       // actualizar el valor del contador
```

- Las operaciones que no se pueden descomponer se llaman atómicas
  - contador++ no es atómica
  - pero las operaciones en que se descompone sí lo son
    - ✓ en realidad depende de la implementación (JVM y lenguaje de máquina)
- Cuando varias hebras se ejecutan concurrentemente se entrelaza la ejecución de sus instrucciones atómicas
- Veamos dos secuencias de ejecución posibles
  - suponemos que inicialmente contador == 0

Secuencias de ejecución (1)



Secuencias de ejecución (2)



Condición de carrera

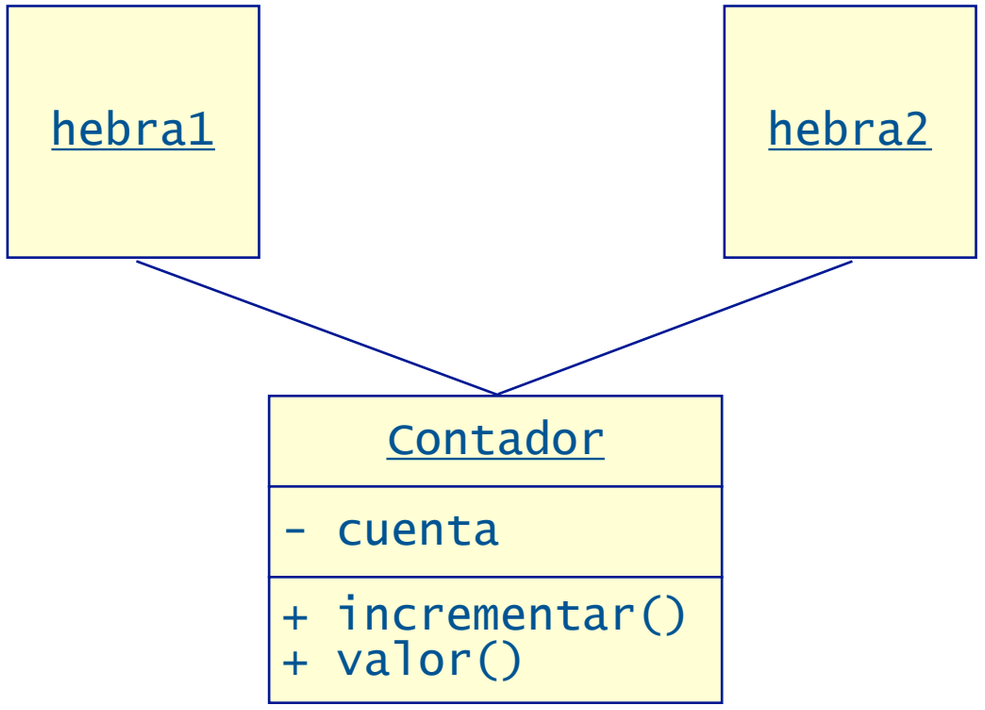
- ¡El resultado puede ser 1 o 2!
  - depende de las velocidades de ejecución relativas de las hebras
    - "quién corre más"
- El resultado de un programa no debe depender de estos detalles
  - imposible de prever de antemano
  - cada ejecución es distinta
    - comportamiento indeterminado
  - imposible hacer ensayos
    - cada vez un resultado diferente

Ejemplo: variable compartida

```
public class PruebaCuenta {

  static long cuenta = 0;       /* variable compartida */

  private static class Incrementa extends Thread {
    public void run () {
        for (int i = 0; i < 1000000; i++) {
          cuenta++;                    /* región crítica */
        }
    }
  }

  public static void main(String[] args) {
    new Incrementa().start(); /* hebra 1 */
    new Incrementa().start(); /* hebra 2 */
    System.out.println("contador = " + contador);
  }
}
```

Datos compartidos en objetos

- Los campos de datos de objetos a los que se accede desde varias hebras también son datos compartidos
  - también pueden dan lugar a condiciones de carrera
  - da igual que se acceda a los datos directamente (si son visibles) o mediante métodos



Ejemplo: objetos con estado (1)

```
public class Contador {

  private long cuenta = 0;        /* estado */

  public Contador (long valorInicial) {
    cuenta = valorInicial;
  }

  public void incrementar () {
    cuenta++;                      /* modifica el estado */
  }

  public long valor () {       /* devuelve el valor */
    return cuenta;
  }
}
```

Ejemplo: objetos con estado (2)

```
public class PruebaContador {

  private static class Incrementa extends Thread {
    Contador contador;

    public Incrementa (Contador c) {
      contador = c;
    }

    public void run () {
      ...
      contador.incrementar();   /* región crítica */
      ...
    }
  }

  // continúa
```

Ejemplo: objetos con estado (3)

```
// continuación

public static void main(String[] args) {
   Contador contador = new Contador(0);
   Thread hebra1 = new Incrementa(contador);
   Thread hebra2 = new Incrementa(contador);
   /* las dos hebras comparten el objeto contador */
   hebra1.start();
   hebra2.start();
   ...
}
```

Regiones críticas y exclusión mutua

- Los segmentos de código en que se accede a variables compartidas se llaman regiones críticas
- Para evitar condiciones de carrera es preciso asegurar la exclusión mutua entre las hebras en las regiones críticas
- Cuando una hebra está en una región crítica ninguna otra puede acceder a los mismos datos
  - primero una hebra efectúa todas las operaciones de su región crítica, luego la otra
  - el orden no importa, siempre que no se entrelacen las operaciones elementales
- De esta forma se consigue que las operaciones con variables compartidas sean atómicas

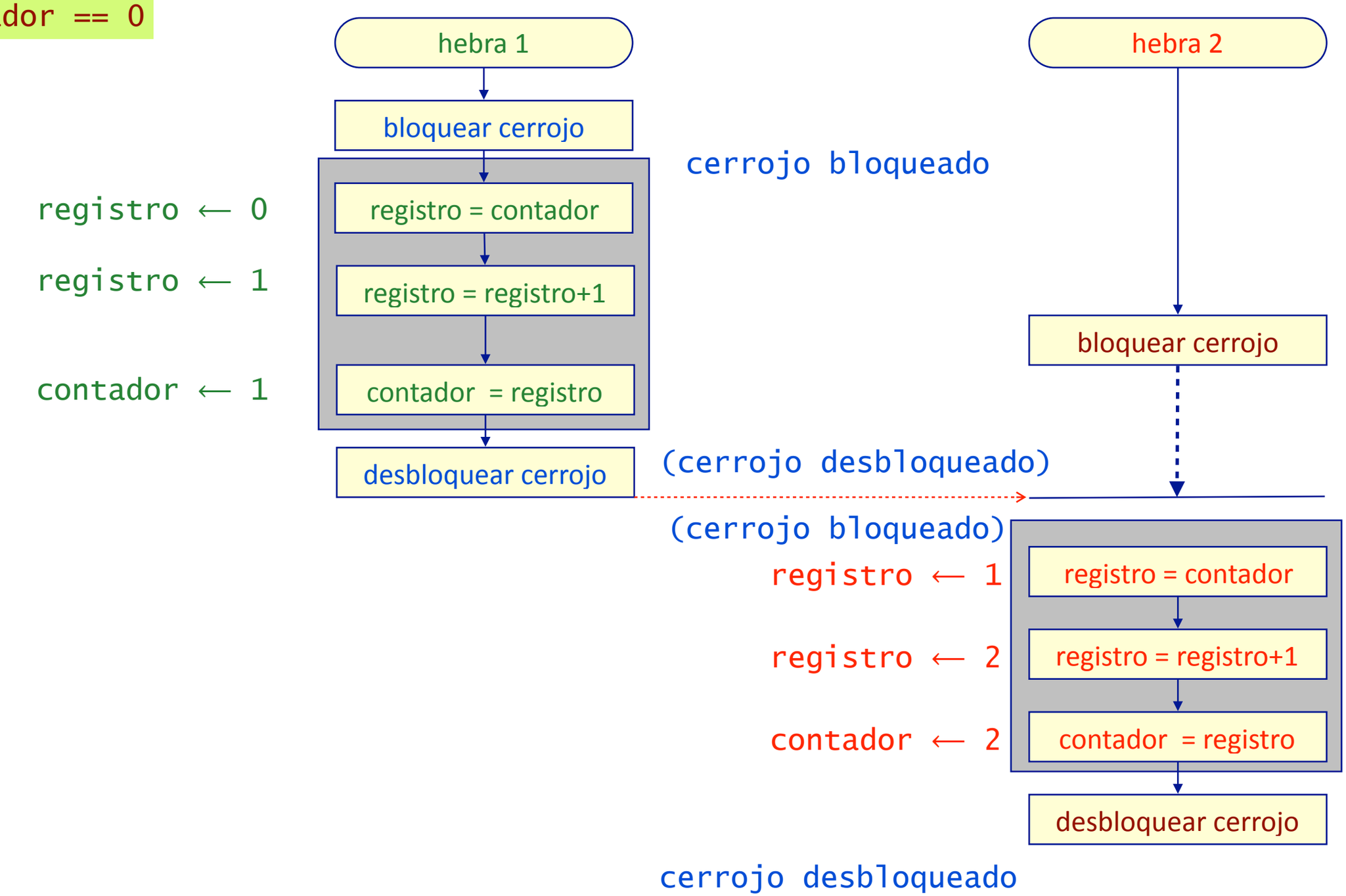
Cerrojos

Cerrojos

- Un cerrojo (*lock*) es un mecanismo de sincronización para asegurar la exclusión mutua
- Puede estar en uno de estos dos estados:
  - bloqueado (cerrado)
  - desbloqueado (abierto)
- Dos operaciones atómicas
  - bloquear (*lock*)
    - si ya está bloqueado, la hebra se queda esperando
  - desbloquear (*unlock*)
    - si hay hebras esperando, continúa una de ellas
- Para asegurar la exclusión mutua
  - bloquear cerrojo // espera si está ocupado
  - región crítica
  - desbloquear cerrojo // si alguien esperaba puede continuar

Ejemplo

`contador == 0`



`contador == 2`

Problemas de los cerrojos

- Es muy mecanismo de muy bajo nivel que no es aconsejable utilizar directamente
  - hay que hacer siempre lock() y unlock(), en ese orden
  - puede ser complicado con estructuras de programa complejas
  - es fácil equivocarse, muchos problemas

- Es mejor dejar que el compilador lo haga por nosotros
  - integrar exclusión mutua con objetos
  - métodos y bloques sincronizados en Java

Monitores

Exclusión mutua en Java

- Java proporciona mecanismos de sincronización abstractos
  - métodos sincronizados
  - bloques sincronizados
- Definen partes del programa que se ejecutan en exclusión mutua (regiones críticas)
- El compilador y la JVM se encargan de gestionar los cerrojos de forma implícita

Métodos sincronizados

- Un método sincronizado se ejecuta en exclusión mutua con los demás métodos sincronizados del mismo objeto

```
public class ContadorSincronizado {

  private long cuenta = 0;        /* estado */

  public ContadorSincronizado (long valorInicial) {
    cuenta = valorInicial;
  }

  public synchronized void incrementar () {
    cuenta++;
  }

  public synchronized long valor () {
    return cuenta;
  }
}
```

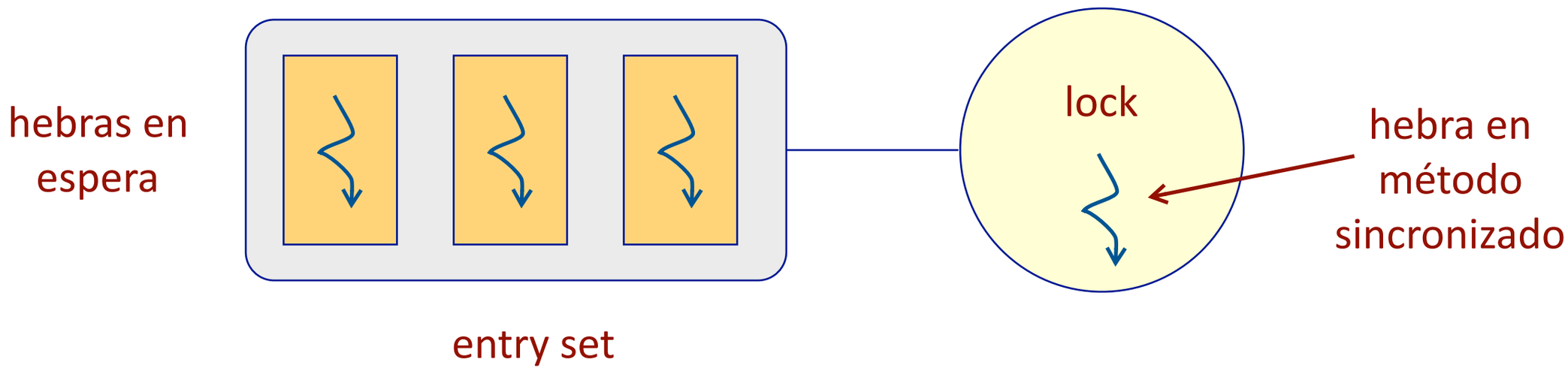
Métodos sincronizados (continuación)

- Las llamadas concurrentes a métodos sincronizados se ejecutan en exclusión mutua

```
...
public void run () {
  ...
  contador.incrementar();   /* región crítica */
  ...
}
...
```

Implementación

- Cada objeto tiene asociado un cerrojo
  - Al empezar a ejecutar un método sincronizado se bloquea el cerrojo
    - si ya estaba bloqueado, la hebra que invoca el método se suspende
    - las hebras que esperan están en una lista asociada al objeto (*entry set*)
  - Al terminar se desbloquea
    - si hay hebras esperando, se reanuda la ejecución de una de ellas
  - El compilador genera el cerrojo y los bloqueos y desbloqueos



Algunos detalles

- Sólo se ejecutan en exclusión mutua los métodos sincronizados
  - si se olvida sincronizar algún método se pueden producir condiciones de carrera
- Los métodos de clase también se pueden sincronizar
  - se usa un cerrojo para la clase y uno por cada objeto
- Los constructores no se pueden sincronizar
  - pero sólo se invocan al crear un objeto

Monitores

- Un monitor es una clase que encapsula datos compartidos y operaciones con los mismos
  - todos los campos de datos se hacen privados
    - sólo se puede acceder a ellos a través de métodos
  - todos los métodos están sincronizados
- Es un esquema clásico para construir programas concurrentes
  - inventado por Per Brinch Hansen y C. Anthony Hoare (1974)
- Ventajas
  - la sincronización se especifica al mismo tiempo que los datos
  - las hebras que acceden a los datos no tienen que incluir ningún mecanismo especial
  - consistente con los principios de la programación con objetos

Bloques sincronizados

- Permiten definir regiones críticas mutuamente exclusivas con una granularidad menor

```
synchronized(objeto) {
  instrucciones
}
```

- Las instrucciones del cuerpo se ejecutan en exclusión mutua
- Se usa el cerrojo asociado al objeto
- Puede ser útil para mejorar las prestaciones
- Pero es más difícil de usar que los métodos sincronizados

Precauciones

- Hay que identificar bien las regiones críticas
  - secuencia de instrucciones donde se lee y se modifica el estado interno de los objetos compartidos
  - no siempre es evidente dónde están
  - puede haber regiones críticas fuera de los datos encapsulados si no se han diseñado bien los métodos

- Hay que sincronizar todas las regiones críticas
  - identificar las operaciones con datos compartidos y realizarlas como métodos sincronizados

Ejemplo (incorrecto)

```
public class Variable {
  private long valor = 0;

  public Variable (long valorInicial) {
    valor = valorInicial;
  }

  public synchronized void modificar(long v) {
    valor = v;
  }

  public synchronized long valor() {
    return valor;
  }
}
```

```
…
Variable v = new Variable(0);
…
long x = v.valor();  // ¡La región crítica es
x +=1;               //  todo esto!!
v.modificar(x);      // ¡No está sincronizada!
...
```

Ejemplo (correcto)

```
public class Variable {
  private long valor = 0;

  …

  public synchronized void incrementar() {
    valor++;
  }

  …
}
```

```
…
v.incrementar ();       // Ahora está sincronizado
…
```

Datos volátiles

Datos volátiles

- Algunos campos de datos se pueden declarar `volatile`
  - indica que el valor puede cambiar desde otras hebras
  - en multiprocesadores no se hacen copias privadas
  - se asegura que las lecturas y escrituras son atómicas
    - ¡nada más!

- Puede ser eficiente si lo único que se hace es cambiar el valor en una hebra y leerlo en otra
  - por ejemplo, para parar la ejecución de una hebra

Ejemplo

```
public class Tarea extends Thread {

  private volatile boolean vivo = true;

  public void run() {
    while (vivo) {
      // actividad concurrente
    }
  }

  public void para() {
    vivo = false;
  }
}
```

Precauciones

- Sólo son atómicas la lectura y la escritura de datos primitivos

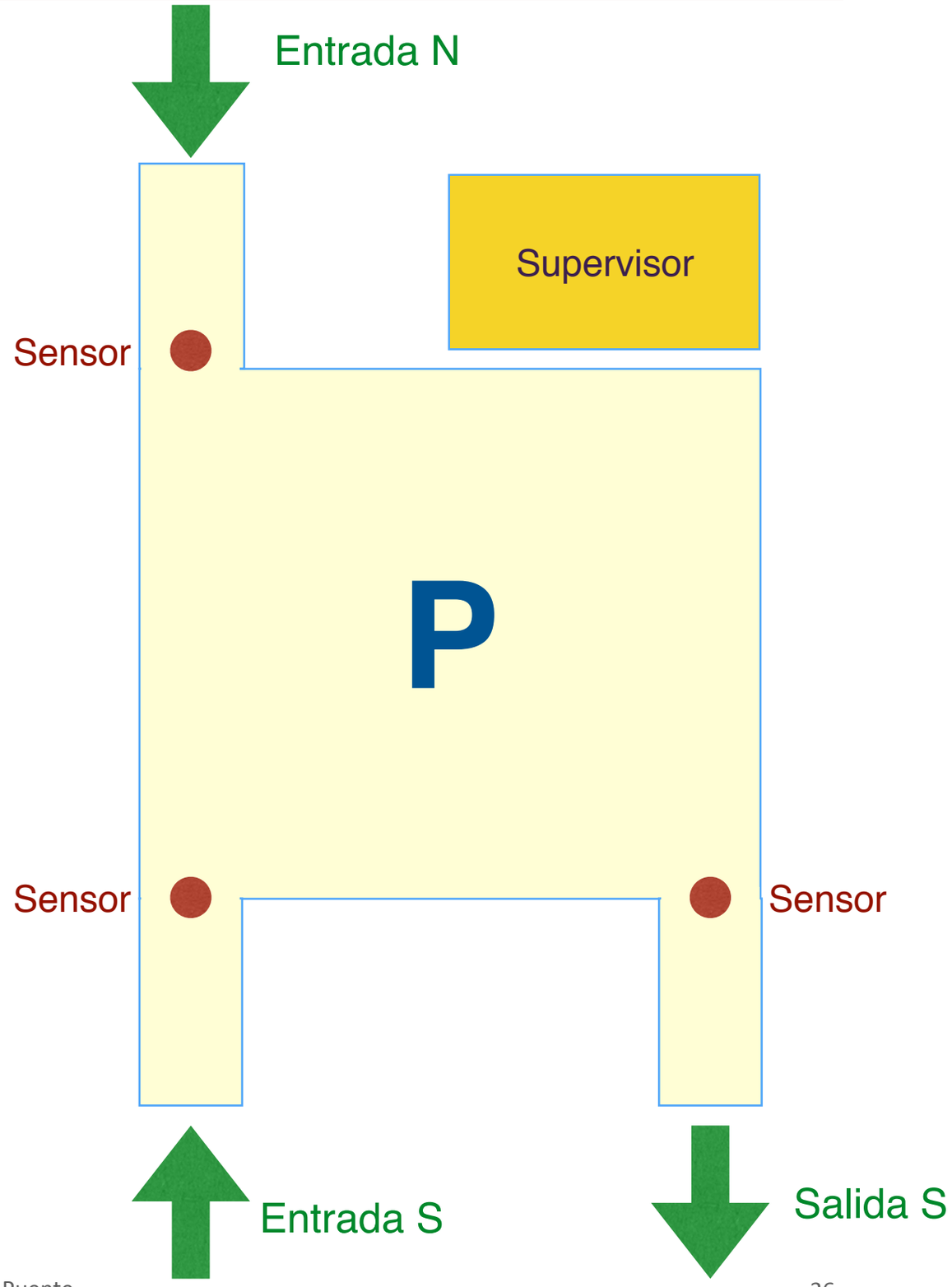
```
volatile int i;
…
i++; // ¡no es atómico, puede haber carreras
```

- Si se declaran como volátiles arrays u objetos, los elementos individuales no están protegidos

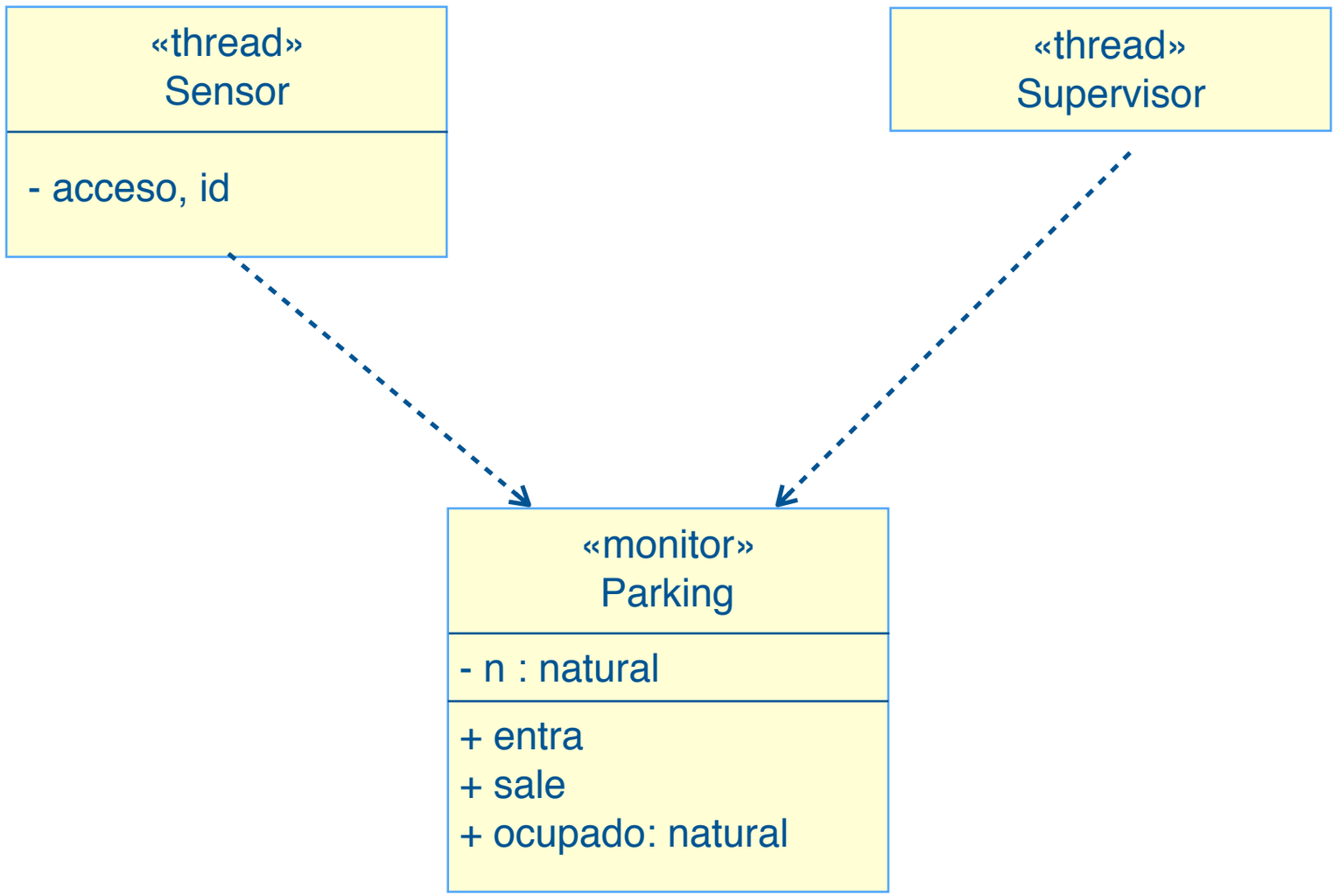
Ejemplo

Gestión de un estacionamiento

- Varias entradas y salidas
  - sensores que avisan cuando pasa un coche
- Supervisor
  - muestra la ocupación del estacionamiento (número de coches)



Diseño



Parking

```
public class Parking {

    private int n = 0; // número de coches en el parking

    // entra un coche por una de las puertas
    public synchronized void entra (String puerta) {
        n++;
    }

    // sale un coche por una de las puertas
    public synchronized void sale (String puerta) {
        n--;
    }

    // consulta
    public synchronized int ocupado() {
        return n;
    }
}
```

Sensor

```
public class Sensor extends Thread {

    private String id;
    private Parking p;
    private Acceso acceso;  // ENTRADA o SALIDA

    public Sensor(Parking p, String id, Acceso acceso) {
        this.p = p;
        this.id = id;
        this.acceso = acceso;
    }

    @Override
    public void run() {
        while (true) {
            …  // detecta un coche
            switch (acceso) {
                case ENTRADA:   p.entra(id);
                                break;
                case SALIDA:    p.sale(id);
                                break;
            }
        }
    }
}
```

Supervisor

```
public class Supervisor extends Thread {

    private Parking p;

    public Supervisor(Parking p) {
        this.p = p;
    }

    @Override
    public void run() {
        // comprueba el estado del parking cada 10 s
        while (true) {
            System.out.println("Número de plazas ocupadas: "
                               + p.ocupado());
            try {
                sleep(10*1000);
            } catch (InterruptedException e) {
                System.err.println(e.toString());
            }
        }
    }
}
```

Crítica

- El monitor protege el estado interno del estacionamiento
  - no hay carreras al actualizar o leer
- Pero no se asegura que el estado sea correcto
  - no deberían entrar coches si el estacionamiento está lleno
- Hace falta un mecanismo que permita imponer estas condiciones
  - sincronización condicional

Resumen

Resumen

- El acceso concurrente a variables compartidas puede dar lugar a resultados erróneos
  - carreras
- La forma de evitar este problema es asegurar la exclusión mutua entre las zonas donde se usan esas variables
  - regiones críticas
- El mecanismo básico para conseguir la exclusión mutua es el cerrojo
  - bloqueo y desbloqueo atómicos
- En Java se usan métodos y bloques sincronizados
  - usan un cerrojo asociado implícitamente a cada objeto

Resumen (continuación)

- Los monitores son clases que aseguran un acceso correcto a objetos compartidos
  - todos los campos de datos son privados
  - todos los métodos están sincronizados
    - excepto los constructores
- Una clase que se puede usar sin problemas desde varias hebras se dice que es segura (*thread safe)*