Sesión $16\frac{3}{4}$: Métodos sincronizados

Julio Mariño
abril 2020

Concurrencia 2019/2020

Universidad Politécnica de Madrid

Grado en Ingeniería Informática
Grado en Matemáticas e Informática
2ble. grado en Ing. Informática y ADE

`http://babel.upm.es/teaching/concurrencia`

introducción/motivación

buscando mecanismos de más alto nivel

- Como hemos visto, los semáforos son un mecanismo de sincronización relativamente sencillo y fácil de implementar, que permite crear protocolos de moderada complejidad, pero sobre los cuales no siempre es fácil razonar. Esto puede crear todo tipo de problemas, incluyendo interbloqueos.
- Entre los mecanismos propuestos para añadir una cierta *modularidad* a la programación de la sincronización cabe destacar los monitores (Brinch-Hansen, Hoare).
- El lenguaje Java, en sus primeras versiones, incluyó una versión simplificada del concepto de monitor, buscando dar una imagen de lenguaje sencillo de usar. Este mecanismo es al que nos referimos como synchronized methods. Algunas personas se siguen refiriendo a ellos como "Java monitors", lo cual es confuso, puesto que con posterioridad se dotó a Java de una librería de monitores "de verdad".
- Como veremos, los métodos sincronizados sufren de carencias serias que desaconsejan su uso para desarrollar software concurrente de manera eficaz.
- No obstante, usaremos esta pequeña sesión para hacer un puente que nos permita motivar mejor el concepto de monitor, al que dedicaremos el siguiente vídeo y para presentar la idea de generar código a partir de los recursos compartidos.

POLITÉCNICA

introducción/motivación

buscando mecanismos de más alto nivel

- Como hemos visto, los semáforos son un mecanismo de sincronización relativamente sencillo y fácil de implementar, que permite crear protocolos de moderada complejidad, pero sobre los cuales no siempre es fácil razonar. Esto puede crear todo tipo de problemas, incluyendo interbloqueos.
- Entre los mecanismos propuestos para añadir una cierta *modularidad* a la programación de la sincronización cabe destacar los monitores (Brinch-Hansen, Hoare).
- El lenguaje Java, en sus primeras versiones, incluyó una versión simplificada del concepto de monitor, buscando dar una imagen de lenguaje sencillo de usar. Este mecanismo es al que nos referimos como synchronized methods. Algunas personas se siguen refiriendo a ellos como "Java monitors", lo cual es confuso, puesto que con posterioridad se dotó a Java de una librería de monitores "de verdad".
- Como veremos, los métodos sincronizados sufren de carencias serias que desaconsejan su uso para desarrollar software concurrente de manera eficaz.
- No obstante, usaremos esta pequeña sesión para hacer un puente que nos permita motivar mejor el concepto de monitor, al que dedicaremos el siguiente vídeo y para presentar la idea de generar código a partir de los recursos compartidos.

POLITÉCNICA

introducción/motivación

buscando mecanismos de más alto nivel

- Como hemos visto, los semáforos son un mecanismo de sincronización relativamente sencillo y fácil de implementar, que permite crear protocolos de moderada complejidad, pero sobre los cuales no siempre es fácil razonar. Esto puede crear todo tipo de problemas, incluyendo interbloqueos.
- Entre los mecanismos propuestos para añadir una cierta *modularidad* a la programación de la sincronización cabe destacar los monitores (Brinch-Hansen, Hoare).
- El lenguaje Java, en sus primeras versiones, incluyó una versión simplificada del concepto de monitor, buscando dar una imagen de lenguaje sencillo de usar. Este mecanismo es al que nos referimos como synchronized methods. Algunas personas se siguen refiriendo a ellos como "Java monitors", lo cual es confuso, puesto que con posterioridad se dotó a Java de una librería de monitores "de verdad".
- Como veremos, los métodos sincronizados sufren de carencias serias que desaconsejan su uso para desarrollar software concurrente de manera eficaz.
- No obstante, usaremos esta pequeña sesión para hacer un puente que nos permita motivar mejor el concepto de monitor, al que dedicaremos el siguiente vídeo y para presentar la idea de generar código a partir de los recursos compartidos.

POLITÉCNICA

introducción/motivación

buscando mecanismos de más alto nivel

- Como hemos visto, los semáforos son un mecanismo de sincronización relativamente sencillo y fácil de implementar, que permite crear protocolos de moderada complejidad, pero sobre los cuales no siempre es fácil razonar. Esto puede crear todo tipo de problemas, incluyendo interbloqueos.
- Entre los mecanismos propuestos para añadir una cierta *modularidad* a la programación de la sincronización cabe destacar los monitores (Brinch-Hansen, Hoare).
- El lenguaje Java, en sus primeras versiones, incluyó una versión simplificada del concepto de monitor, buscando dar una imagen de lenguaje sencillo de usar. Este mecanismo es al que nos referimos como synchronized methods. Algunas personas se siguen refiriendo a ellos como "Java monitors", lo cual es confuso, puesto que con posterioridad se dotó a Java de una librería de monitores "de verdad".
- Como veremos, los métodos sincronizados sufren de carencias serias que desaconsejan su uso para desarrollar software concurrente de manera eficaz.
- No obstante, usaremos esta pequeña sesión para hacer un puente que nos permita motivar mejor el concepto de monitor, al que dedicaremos el siguiente vídeo y para presentar la idea de generar código a partir de los recursos compartidos.

POLITÉCNICA

introducción/motivación

buscando mecanismos de más alto nivel

- Como hemos visto, los semáforos son un mecanismo de sincronización relativamente sencillo y fácil de implementar, que permite crear protocolos de moderada complejidad, pero sobre los cuales no siempre es fácil razonar. Esto puede crear todo tipo de problemas, incluyendo interbloqueos.
- Entre los mecanismos propuestos para añadir una cierta *modularidad* a la programación de la sincronización cabe destacar los monitores (Brinch-Hansen, Hoare).
- El lenguaje Java, en sus primeras versiones, incluyó una versión simplificada del concepto de monitor, buscando dar una imagen de lenguaje sencillo de usar. Este mecanismo es al que nos referimos como synchronized methods. Algunas personas se siguen refiriendo a ellos como "Java monitors", lo cual es confuso, puesto que con posterioridad se dotó a Java de una librería de monitores "de verdad".
- Como veremos, los métodos sincronizados sufren de carencias serias que desaconsejan su uso para desarrollar software concurrente de manera eficaz.
- No obstante, usaremos esta pequeña sesión para hacer un puente que nos permita motivar mejor el concepto de monitor, al que dedicaremos el siguiente vídeo y para presentar la idea de generar código a partir de los recursos compartidos.

POLITÉCNICA

métodos sincronizados

sincronización para todos los públicos en Java

- En su versión inicial, los métodos sincronizados buscan proporcionar una solución muy simple al problema de la exclusión mutua y mecanismos muy básicos para programar sincronización por condición.
- Exclusión mutua: Se introduce el modificador **synchronized** para las definiciones de métodos de objetos. Por ejemplo:
  public synchronized int getValue () {...}
  El uso de **synchronized** garantiza:
  - Que solo un thread puede ejecutar el método sincronizado sobre un mismo objeto en un momento dado, y
  - que cualquier llamada posterior desde otro thread a ese u otro método sincronizado de dicho objeto necesariamente reflejará los cambios de estado provocados por ejecuciones previas de métodos sincronizados. (En otras palabras, se asegura que no va a haber incoherencia de caches entre hilos.)

  Ojo!!: Java permite que coexistan métodos sincronizados y no sincronizados en un mismo objeto. El constructor nunca se declara sincronizado y debemos tener cuidado de no compartir un objeto entre varios threads antes de que su constructor haya concluido.

POLITÉCNICA

métodos sincronizados

sincronización para todos los públicos en Java

- En su versión inicial, los métodos sincronizados buscan proporcionar una solución muy simple al problema de la exclusión mutua y mecanismos muy básicos para programar sincronización por condición.
- Exclusión mutua: Se introduce el modificador **synchronized** para las definiciones de métodos de objetos. Por ejemplo:
  public synchronized int getValue () $\{...\}$
  El uso de **synchronized** garantiza:
  - Que solo un thread puede ejecutar el método sincronizado sobre un mismo objeto en un momento dado, y
  - que cualquier llamada posterior desde otro thread a ese u otro método sincronizado de dicho objeto necesariamente reflejará los cambios de estado provocados por ejecuciones previas de métodos sincronizados. (En otras palabras, se asegura que no va a haber incoherencia de caches entre hilos.)

  Ojo!!: Java permite que coexistan métodos sincronizados y no sincronizados en un mismo objeto. El constructor nunca se declara sincronizado y debemos tener cuidado de no compartir un objeto entre varios threads antes de que su constructor haya concluido.

POLITÉCNICA

ejemplo: contador compartido

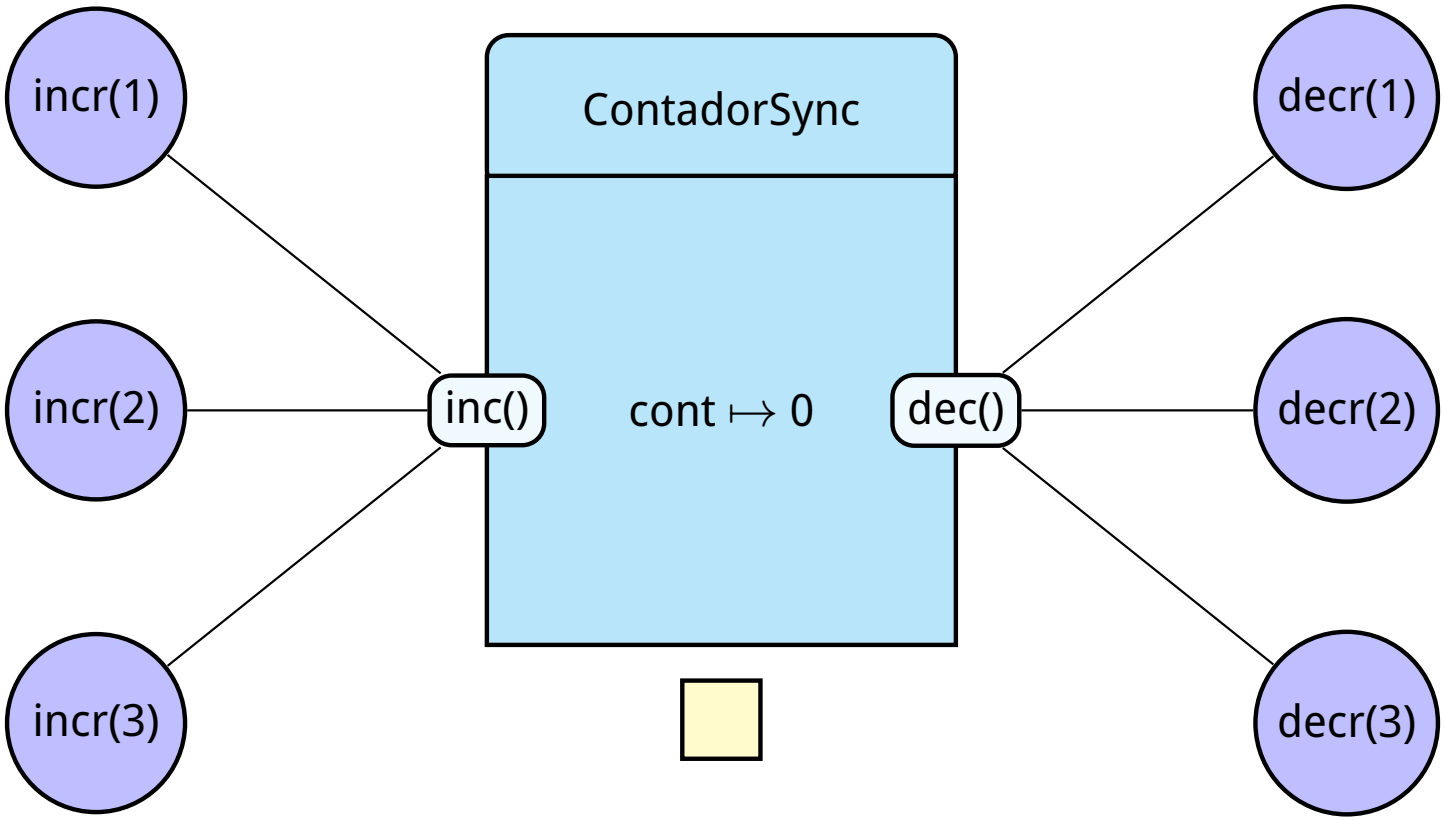
solo exclusión mutua

```
class ContadorSync {

    private int cont;

    public ContadorSync (int n) {
        this.cont = n;
    }

    public synchronized int getValue() {
        return this.cont;
    }

    public synchronized void inc () {
        this.cont++;
    }

    public synchronized void dec () {
        this.cont--;
    }

}
```

POLITÉCNICA

modelo de ejecución

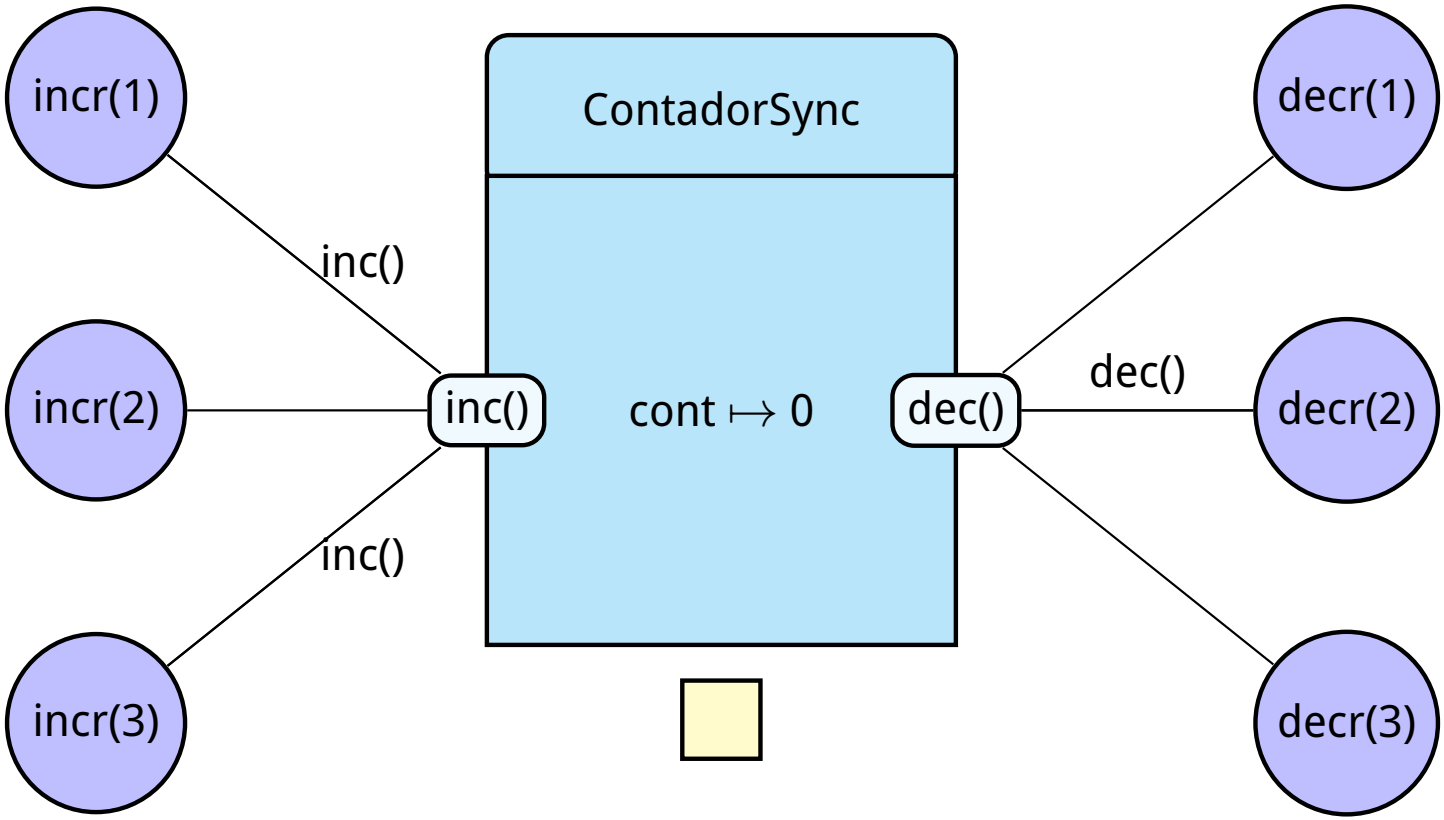
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

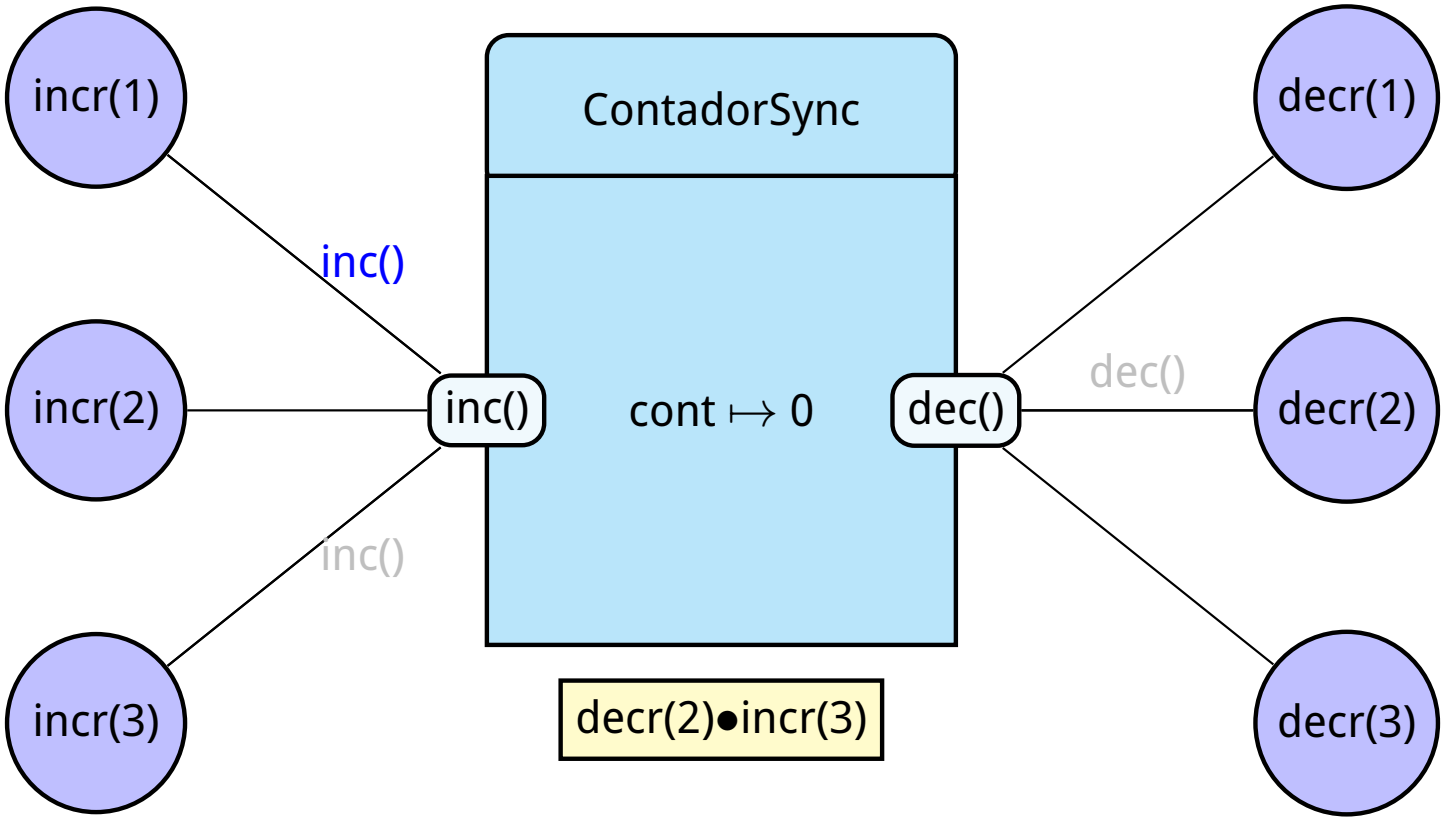
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

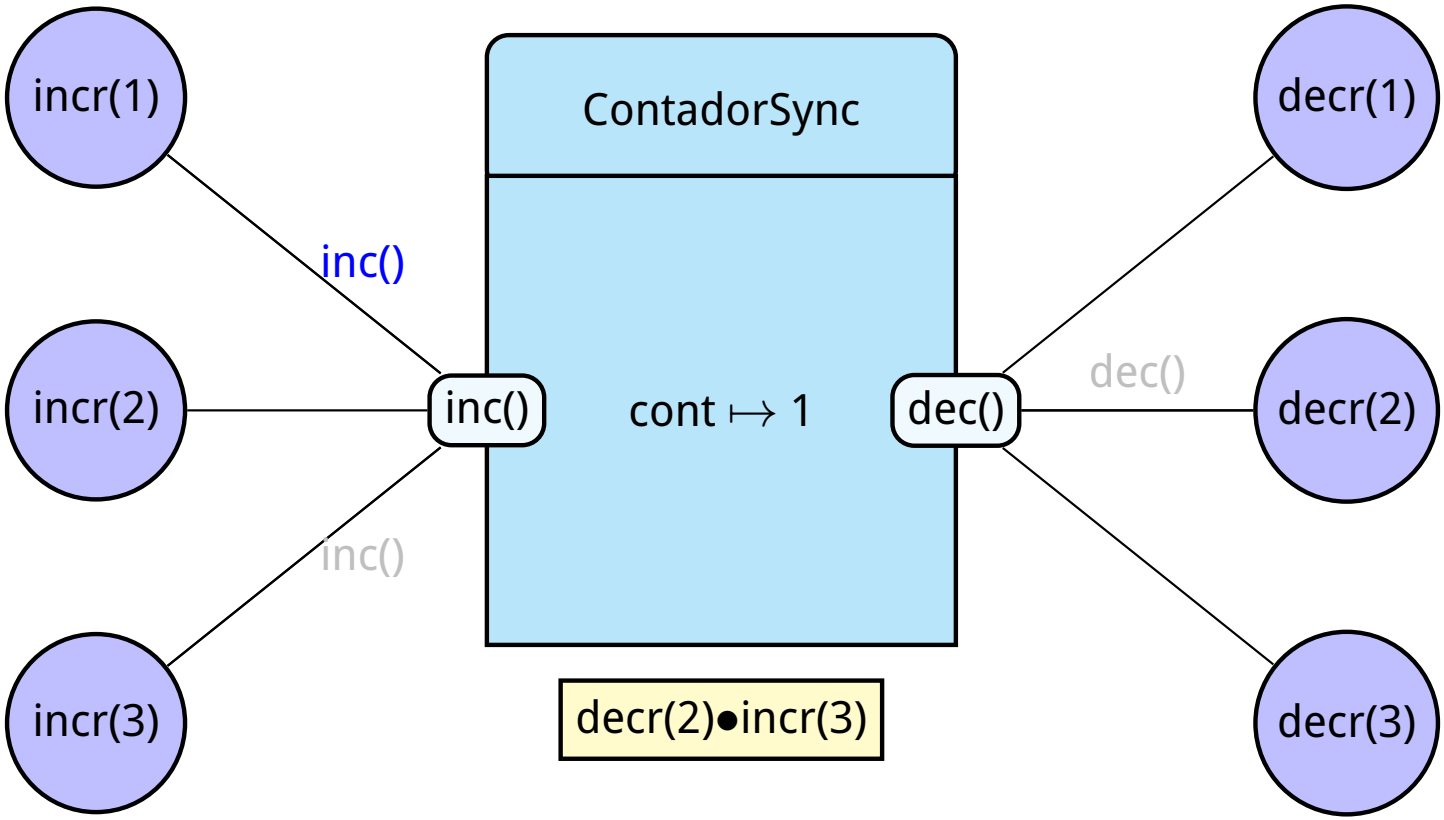
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

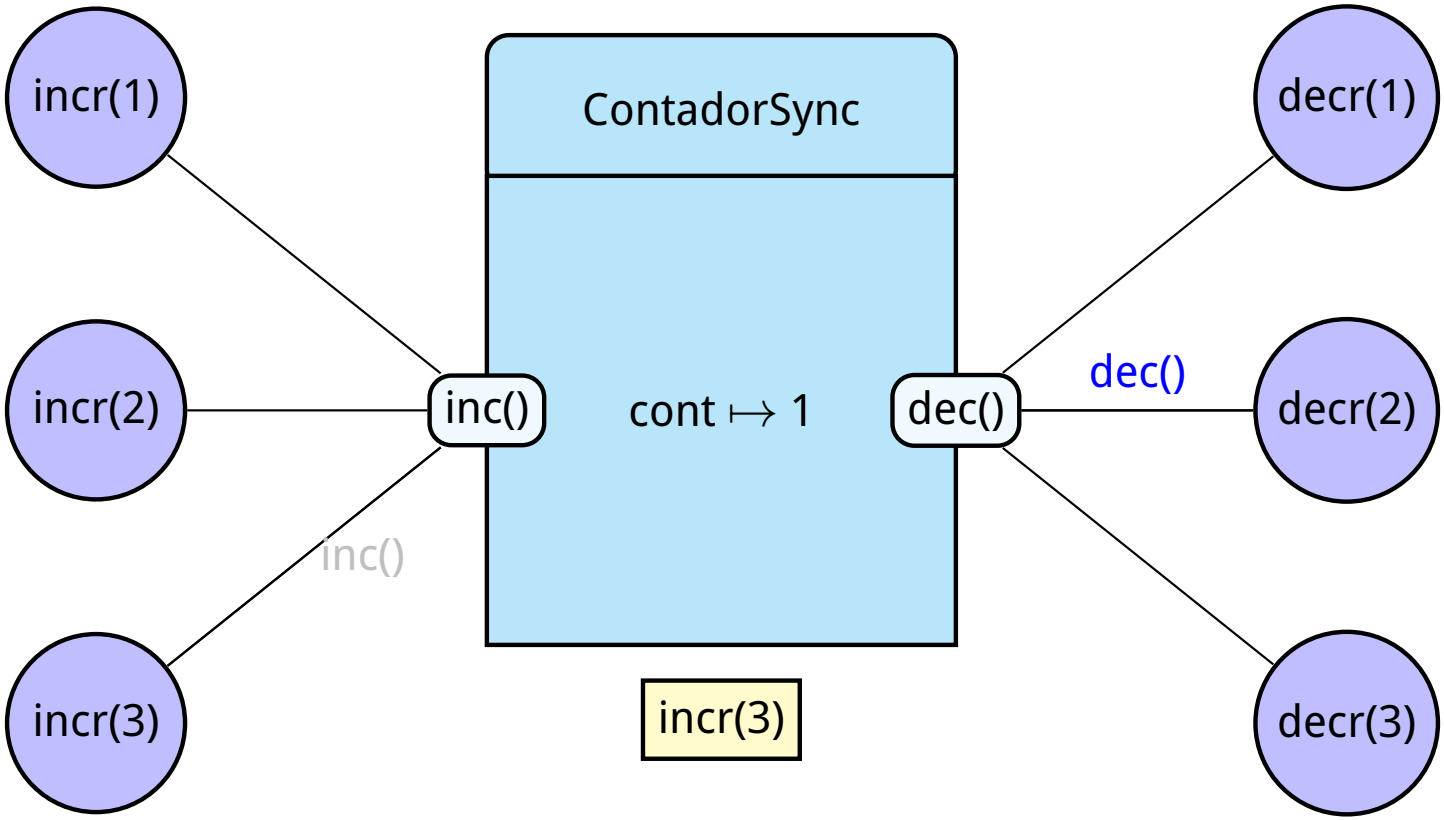
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

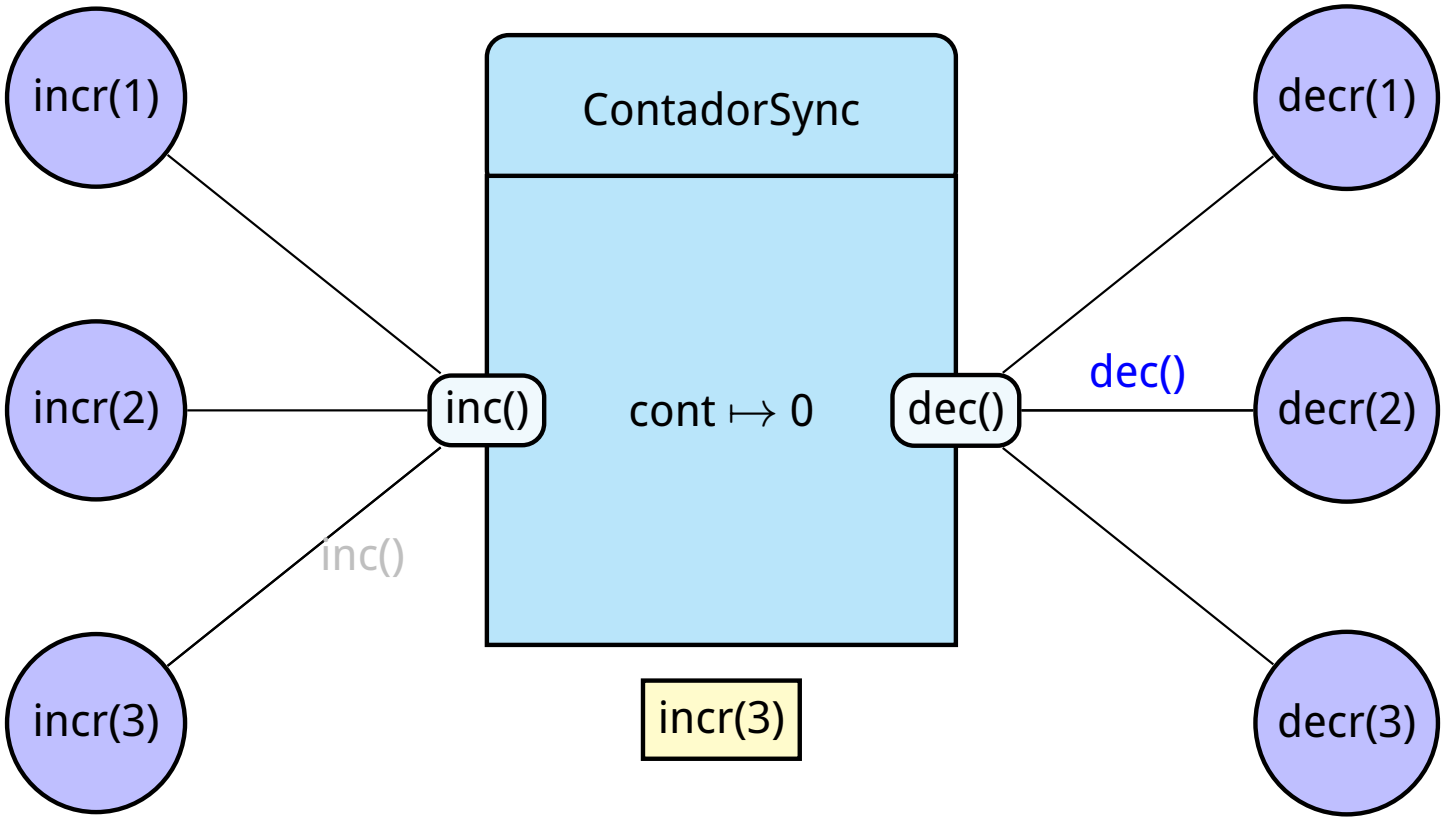
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

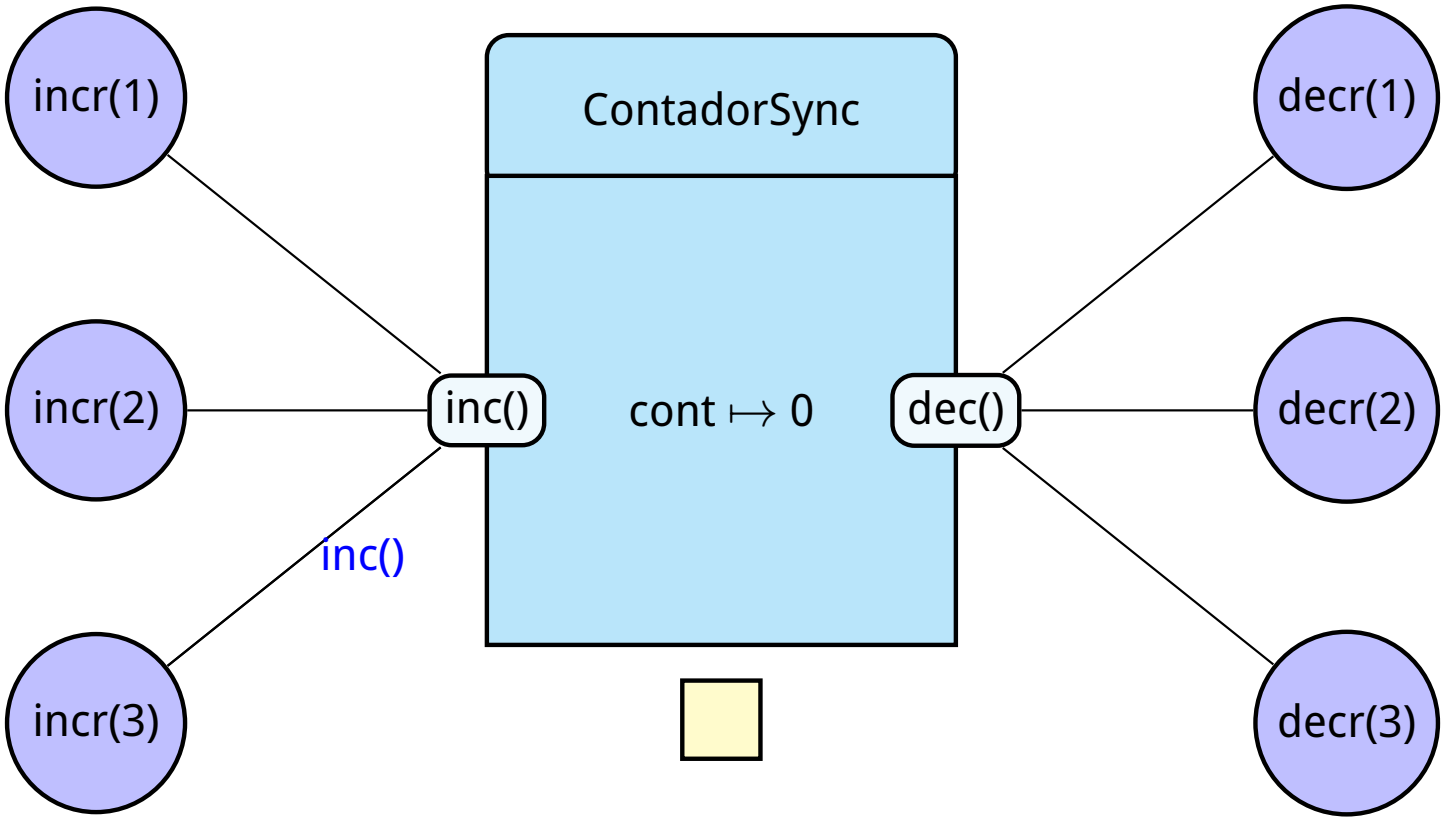
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

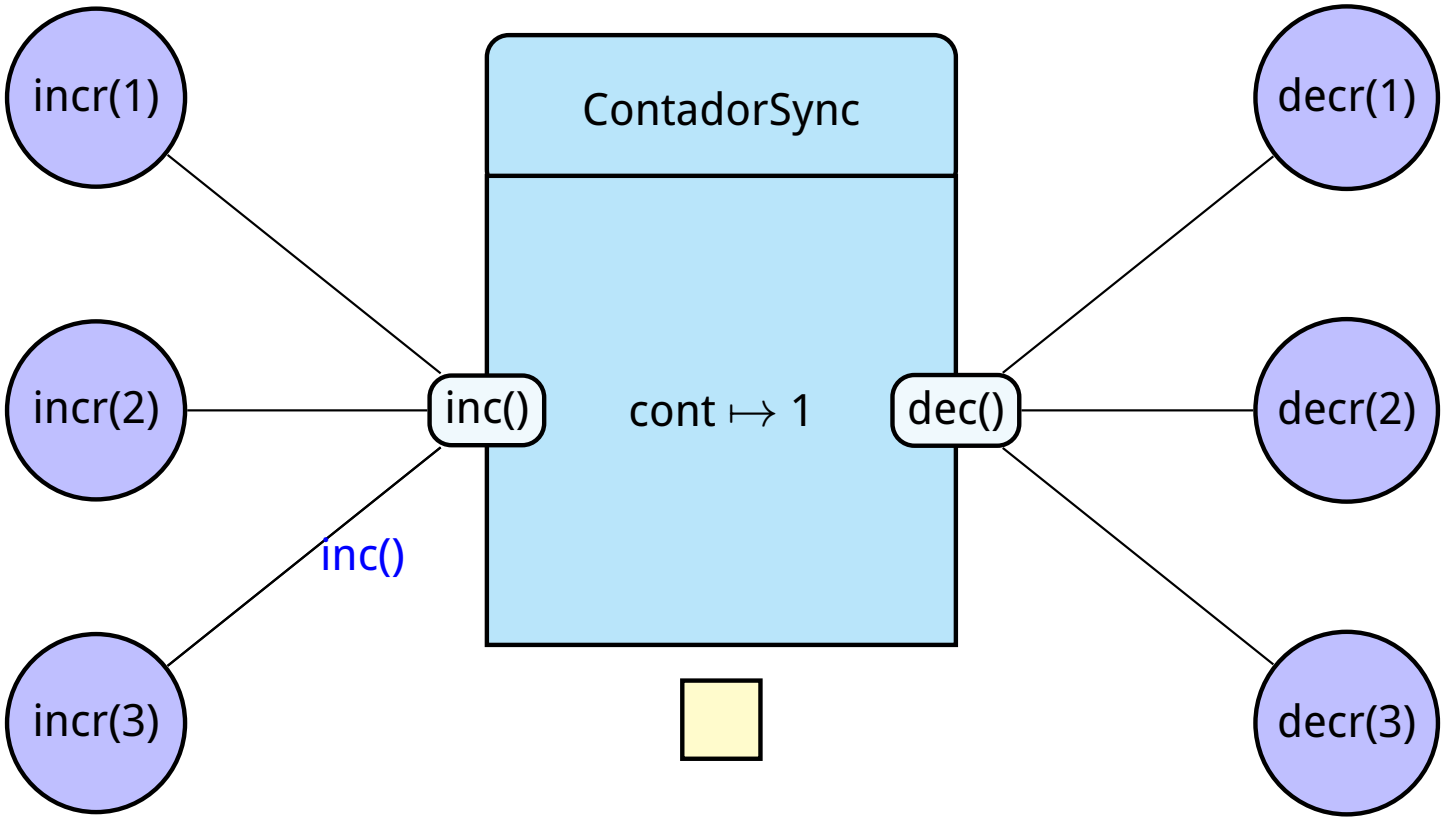
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

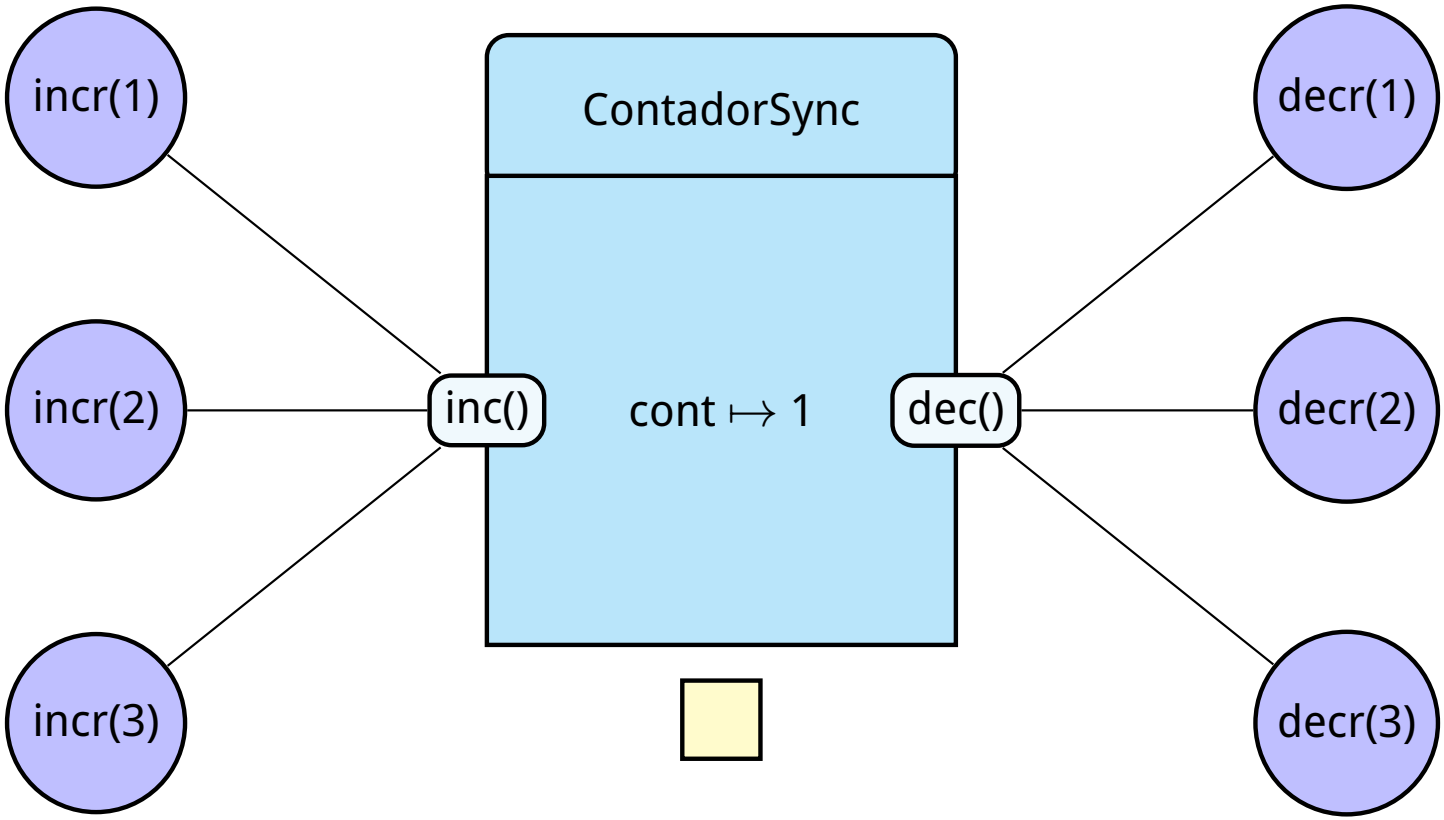
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

visualizando la exclusión mutua



POLITÉCNICA

métodos sincronizados (cont.)

sincronización para todos los públicos en Java

- Sincronización por condición: Se añade un método wait() a la clase Object que bloquea al thread que lo invoca. Típicamente se invoca wait() desde dentro de un método sincronizado. El thread que se bloquea libera la exclusión mutua del objeto – lo cual permite que otro hilo ejecute métodos sincronizados de dicho objeto. (wait(), al igual que join() puede lanzar InterruptedException.)
- Los hilos bloqueados por ejecutar wait() sobre un objeto son incluidos en una estructura de datos propia de dicho objeto que llamaremos waitset o "sala de espera" del objeto.
- ¿Cómo se desbloquean los hilos de la sala de espera? Otro thread que haya ganado acceso exclusivo al objeto puede ejecutar sobre él notify() o notifyAll() – métodos de Object. En el primer caso *uno* de los hilos del waitset se "extrae" de ahí y se lo pone en espera en la cola de acceso a la exclusión mutua (la que impide la ejecución simultánea de varios métodos sincronizados). En el caso de notifyAll() se hace lo mismo con todos los hilos del waitset.
- Ojo!!: Java no garantiza nada sobre qué hilo es el elegido al ejecutar notify(), en qué orden son insertados los threads desbloqueados en la cola (si es que es una cola) de acceso a la ejecución exclusiva, etc.

POLITÉCNICA

métodos sincronizados (cont.)

sincronización para todos los públicos en Java

- Sincronización por condición: Se añade un método wait() a la clase Object que bloquea al thread que lo invoca. Típicamente se invoca wait() desde dentro de un método sincronizado. El thread que se bloquea libera la exclusión mutua del objeto – lo cual permite que otro hilo ejecute métodos sincronizados de dicho objeto. (wait(), al igual que join() puede lanzar InterruptedException.)
- Los hilos bloqueados por ejecutar wait() sobre un objeto son incluidos en una estructura de datos propia de dicho objeto que llamaremos waitset o "sala de espera" del objeto.
- ¿Cómo se desbloquean los hilos de la sala de espera? Otro thread que haya ganado acceso exclusivo al objeto puede ejecutar sobre él notify() o notifyAll() – métodos de Object. En el primer caso *uno* de los hilos del waitset se "extrae" de ahí y se lo pone en espera en la cola de acceso a la exclusión mutua (la que impide la ejecución simultánea de varios métodos sincronizados). En el caso de notifyAll() se hace lo mismo con todos los hilos del waitset.
- Ojo!!: Java no garantiza nada sobre qué hilo es el elegido al ejecutar notify(), en qué orden son insertados los threads desbloqueados en la cola (si es que es una cola) de acceso a la ejecución exclusiva, etc.

POLITÉCNICA

métodos sincronizados (cont.)

sincronización para todos los públicos en Java

- Sincronización por condición: Se añade un método wait() a la clase Object que bloquea al thread que lo invoca. Típicamente se invoca wait() desde dentro de un método sincronizado. El thread que se bloquea libera la exclusión mutua del objeto – lo cual permite que otro hilo ejecute métodos sincronizados de dicho objeto. (wait(), al igual que join() puede lanzar InterruptedException.)
- Los hilos bloqueados por ejecutar wait() sobre un objeto son incluidos en una estructura de datos propia de dicho objeto que llamaremos waitset o "sala de espera" del objeto.
- ¿Cómo se desbloquean los hilos de la sala de espera? Otro thread que haya ganado acceso exclusivo al objeto puede ejecutar sobre él notify() o notifyAll() – métodos de Object. En el primer caso *uno* de los hilos del waitset se "extrae" de ahí y se lo pone en espera en la cola de acceso a la exclusión mutua (la que impide la ejecución simultánea de varios métodos sincronizados). En el caso de notifyAll() se hace lo mismo con todos los hilos del waitset.
- Ojo!!: Java no garantiza nada sobre qué hilo es el elegido al ejecutar notify(), en qué orden son insertados los threads desbloqueados en la cola (si es que es una cola) de acceso a la ejecución exclusiva, etc.

POLITÉCNICA

métodos sincronizados (cont.)

sincronización para todos los públicos en Java

- Sincronización por condición: Se añade un método wait() a la clase Object que bloquea al thread que lo invoca. Típicamente se invoca wait() desde dentro de un método sincronizado. El thread que se bloquea libera la exclusión mutua del objeto – lo cual permite que otro hilo ejecute métodos sincronizados de dicho objeto. (wait(), al igual que join() puede lanzar InterruptedException.)
- Los hilos bloqueados por ejecutar wait() sobre un objeto son incluidos en una estructura de datos propia de dicho objeto que llamaremos waitset o "sala de espera" del objeto.
- ¿Cómo se desbloquean los hilos de la sala de espera? Otro thread que haya ganado acceso exclusivo al objeto puede ejecutar sobre él notify() o notifyAll() – métodos de Object. En el primer caso *uno* de los hilos del waitset se "extrae" de ahí y se lo pone en espera en la cola de acceso a la exclusión mutua (la que impide la ejecución simultánea de varios métodos sincronizados). En el caso de notifyAll() se hace lo mismo con todos los hilos del waitset.
- Ojo!!: Java no garantiza nada sobre qué hilo es el elegido al ejecutar notify(), en qué orden son insertados los threads desbloqueados en la cola (si es que es una cola) de acceso a la ejecución exclusiva, etc.

POLITÉCNICA

tratando la sincronización por condición

esqueletos de código

- Nuestro objetivo es tener una manera relativamente sencilla y *trazable* de traducir especificaciones de recurso compartido a código.
- Generalmente, asociaremos especificaciones de recurso compartido con *interfaces Java*, e implementaciones de dichos recursos con clases Java que implementan dichas interfaces. Por ejemplo, tendremos la interfaz Almacen1 y la clase Almacen1Sync, etc.
- El constructor deberá establecer el estado inicial del recurso.
- En el caso de implementar un recurso compartido con métodos sincronizados tendremos un método declarado **synchronized** por cada acción del recurso.
- La estructura interna del código de cada uno de esos métodos será la siguiente:

```
public synchronized tipo accion (...) {

  while (! <CPRE>) { wait() };

  // si estamos aqui es que se cumple la CPRE
  <codigo que establece la POST>

  // podemos desbloquear a otro hilo?
  notifyAll();

  // opcionalmente:
  return (<lo que sea>);
}
```

POLITÉCNICA

tratando la sincronización por condición

esqueletos de código

- Nuestro objetivo es tener una manera relativamente sencilla y *trazable* de traducir especificaciones de recurso compartido a código.
- Generalmente, asociaremos especificaciones de recurso compartido con *interfaces Java*, e implementaciones de dichos recursos con clases Java que implementan dichas interfaces. Por ejemplo, tendremos la interfaz Almacen1 y la clase Almacen1Sync, etc.
- El constructor deberá establecer el estado inicial del recurso.
- En el caso de implementar un recurso compartido con métodos sincronizados tendremos un método declarado **synchronized** por cada acción del recurso.
- La estructura interna del código de cada uno de esos métodos será la siguiente:

```
public synchronized tipo accion (...) {

  while (! <CPRE>) { wait() };

  // si estamos aqui es que se cumple la CPRE
  <codigo que establece la POST>

  // podemos desbloquear a otro hilo?
  notifyAll();

  // opcionalmente:
  return (<lo que sea>);
}
```

POLITÉCNICA

tratando la sincronización por condición

esqueletos de código

- Nuestro objetivo es tener una manera relativamente sencilla y *trazable* de traducir especificaciones de recurso compartido a código.
- Generalmente, asociaremos especificaciones de recurso compartido con *interfaces Java*, e implementaciones de dichos recursos con clases Java que implementan dichas interfaces. Por ejemplo, tendremos la interfaz Almacen1 y la clase Almacen1Sync, etc.
- El constructor deberá establecer el estado inicial del recurso.
- En el caso de implementar un recurso compartido con métodos sincronizados tendremos un método declarado **synchronized** por cada acción del recurso.
- La estructura interna del código de cada uno de esos métodos será la siguiente:

```
public synchronized tipo accion (...) {

  while (! <CPRE>) { wait() };

  // si estamos aqui es que se cumple la CPRE
  <codigo que establece la POST>

  // podemos desbloquear a otro hilo?
  notifyAll();

  // opcionalmente:
  return (<lo que sea>);
}
```

POLITÉCNICA

tratando la sincronización por condición

esqueletos de código

- Nuestro objetivo es tener una manera relativamente sencilla y *trazable* de traducir especificaciones de recurso compartido a código.
- Generalmente, asociaremos especificaciones de recurso compartido con *interfaces Java*, e implementaciones de dichos recursos con clases Java que implementan dichas interfaces. Por ejemplo, tendremos la interfaz Almacen1 y la clase Almacen1Sync, etc.
- El constructor deberá establecer el estado inicial del recurso.
- En el caso de implementar un recurso compartido con métodos sincronizados tendremos un método declarado **synchronized** por cada acción del recurso.
- La estructura interna del código de cada uno de esos métodos será la siguiente:

```
public synchronized tipo accion (...) {

  while (! <CPRE>) { wait() };

  // si estamos aqui es que se cumple la CPRE
  <codigo que establece la POST>

  // podemos desbloquear a otro hilo?
  notifyAll();

  // opcionalmente:
  return (<lo que sea>);
}
```

POLITÉCNICA

tratando la sincronización por condición

esqueletos de código

- Nuestro objetivo es tener una manera relativamente sencilla y *trazable* de traducir especificaciones de recurso compartido a código.
- Generalmente, asociaremos especificaciones de recurso compartido con *interfaces Java*, e implementaciones de dichos recursos con clases Java que implementan dichas interfaces. Por ejemplo, tendremos la interfaz Almacen1 y la clase Almacen1Sync, etc.
- El constructor deberá establecer el estado inicial del recurso.
- En el caso de implementar un recurso compartido con métodos sincronizados tendremos un método declarado **synchronized** por cada acción del recurso.
- La estructura interna del código de cada uno de esos métodos será la siguiente:

```
public synchronized tipo accion (...) {

  while (! <CPRE>) { wait() };

  // si estamos aqui es que se cumple la CPRE
  <codigo que establece la POST>

  // podemos desbloquear a otro hilo?
  notifyAll();

  // opcionalmente:
  return (<lo que sea>);
}
```

POLITÉCNICA

ejemplo: aparcamiento

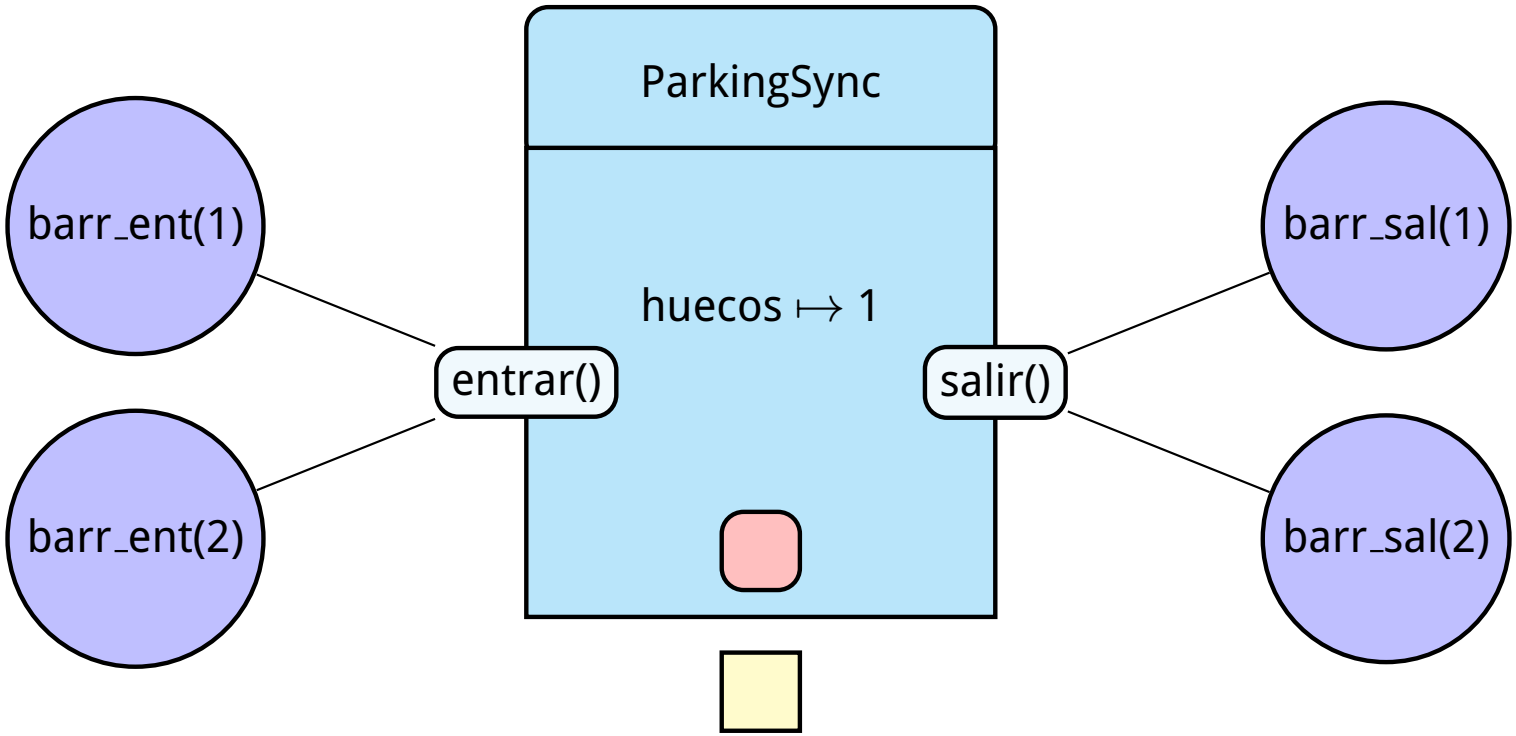
añadimos sincronización por condición al contador compartido

```
public class ParkingSync {
    private static int CAP = 5;
    private int huecos;

    public Parking() {
        this.huecos = CAP;
    }

    public synchronized void entrar() {
        while(this.huecos == 0) {
          try { wait(); } catch (InterruptedException exc) {}
        }
        this.huecos = this.huecos - 1;
        notifyAll();
    }

    public synchronized void salir() {
        this.huecos = this.huecos + 1;
        notifyAll();
    }
}
```

POLITÉCNICA

modelo de ejecución

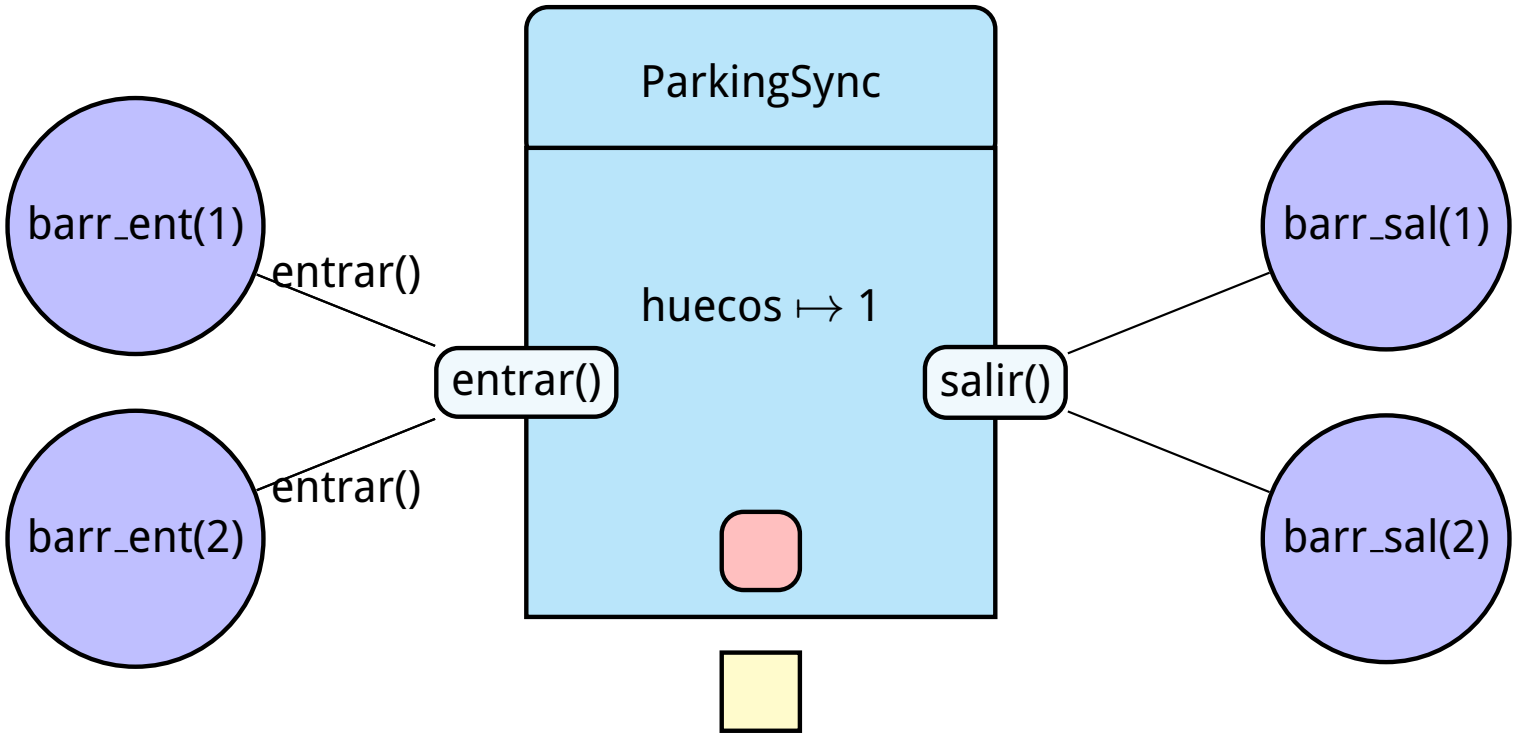
aquí, el waitset funciona como esperamos



POLITÉCNICA

modelo de ejecución

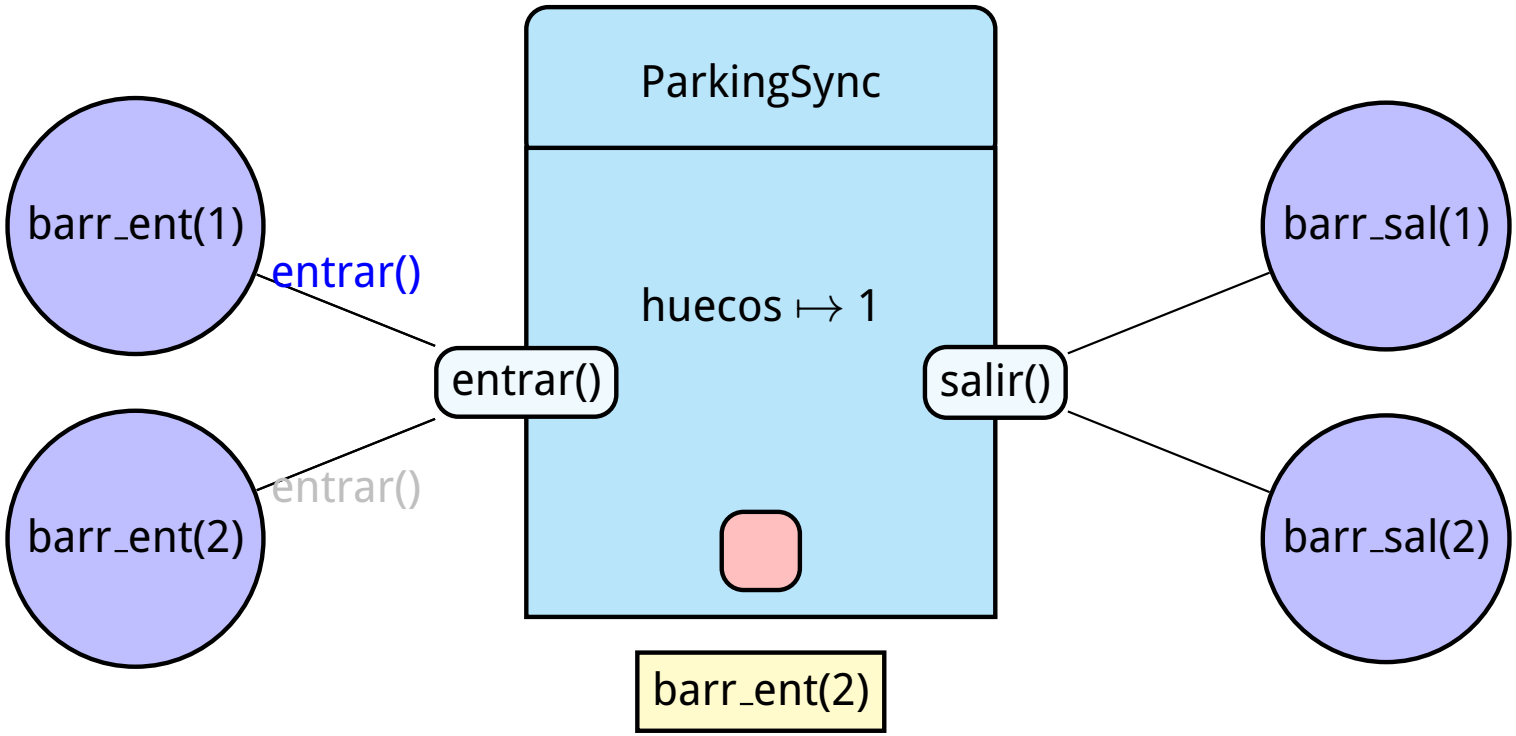
aquí, el waitset funciona como esperamos



POLITÉCNICA

modelo de ejecución

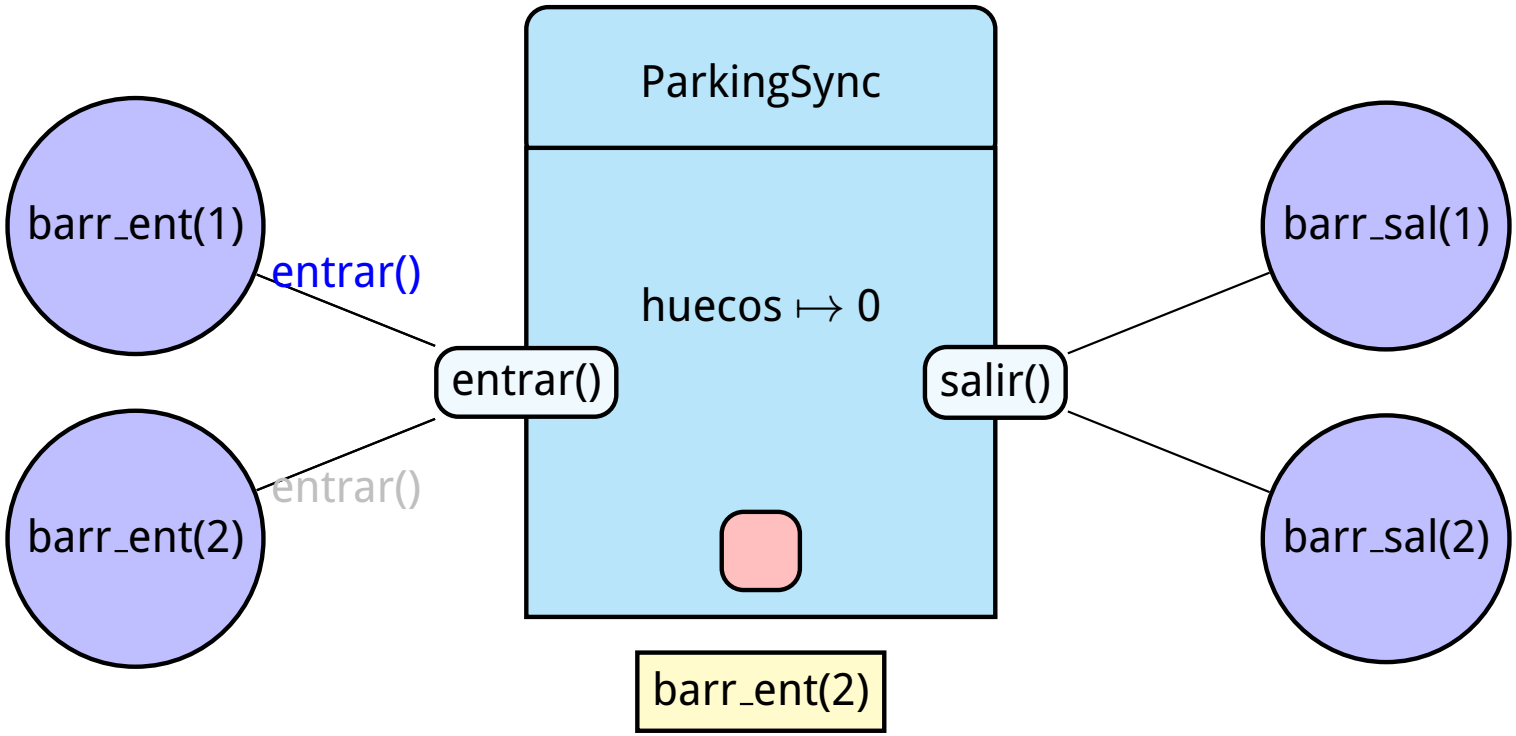
aquí, el waitset funciona como esperamos



POLITÉCNICA

modelo de ejecución

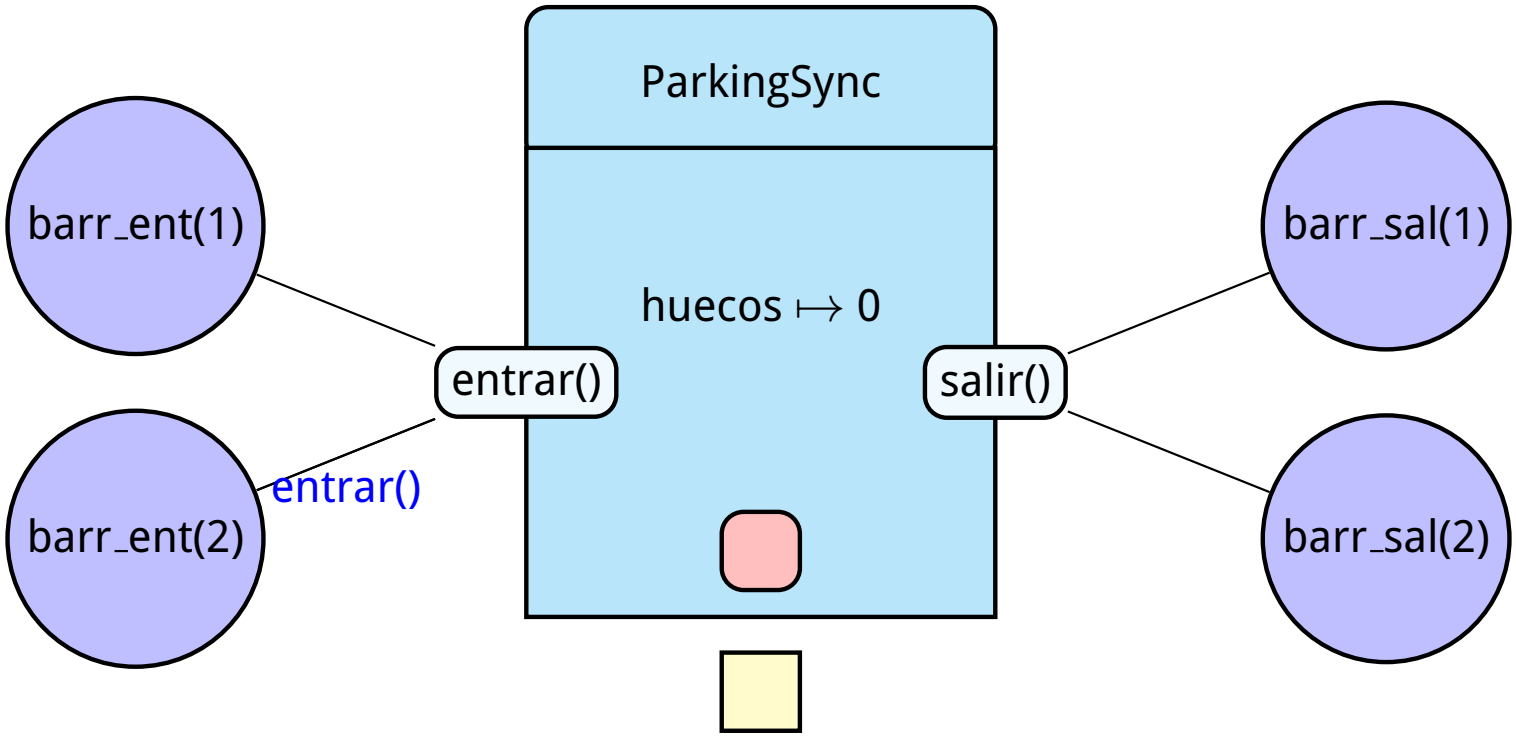
aquí, el waitset funciona como esperamos



POLITÉCNICA

modelo de ejecución

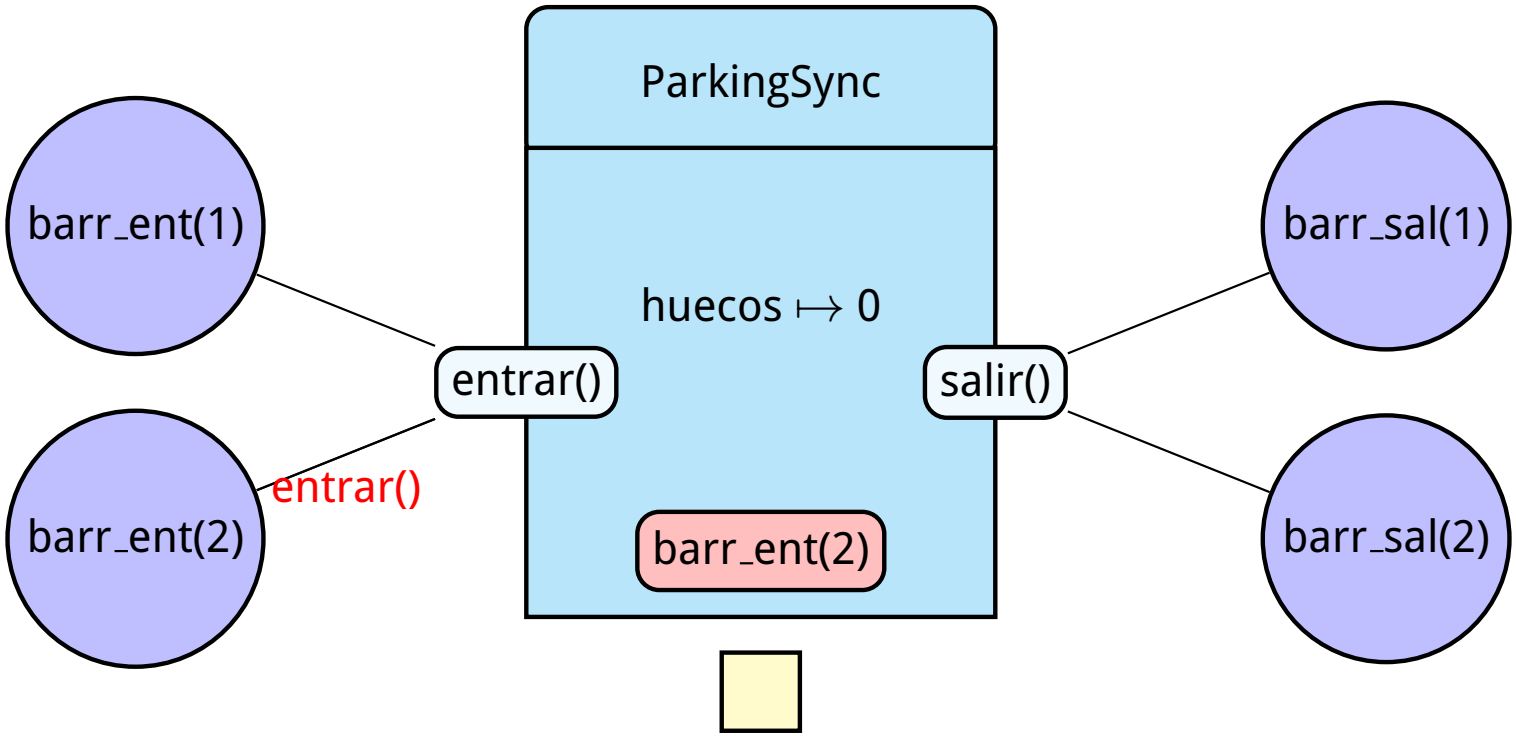
aquí, el waitset funciona como esperamos



POLITÉCNICA

modelo de ejecución

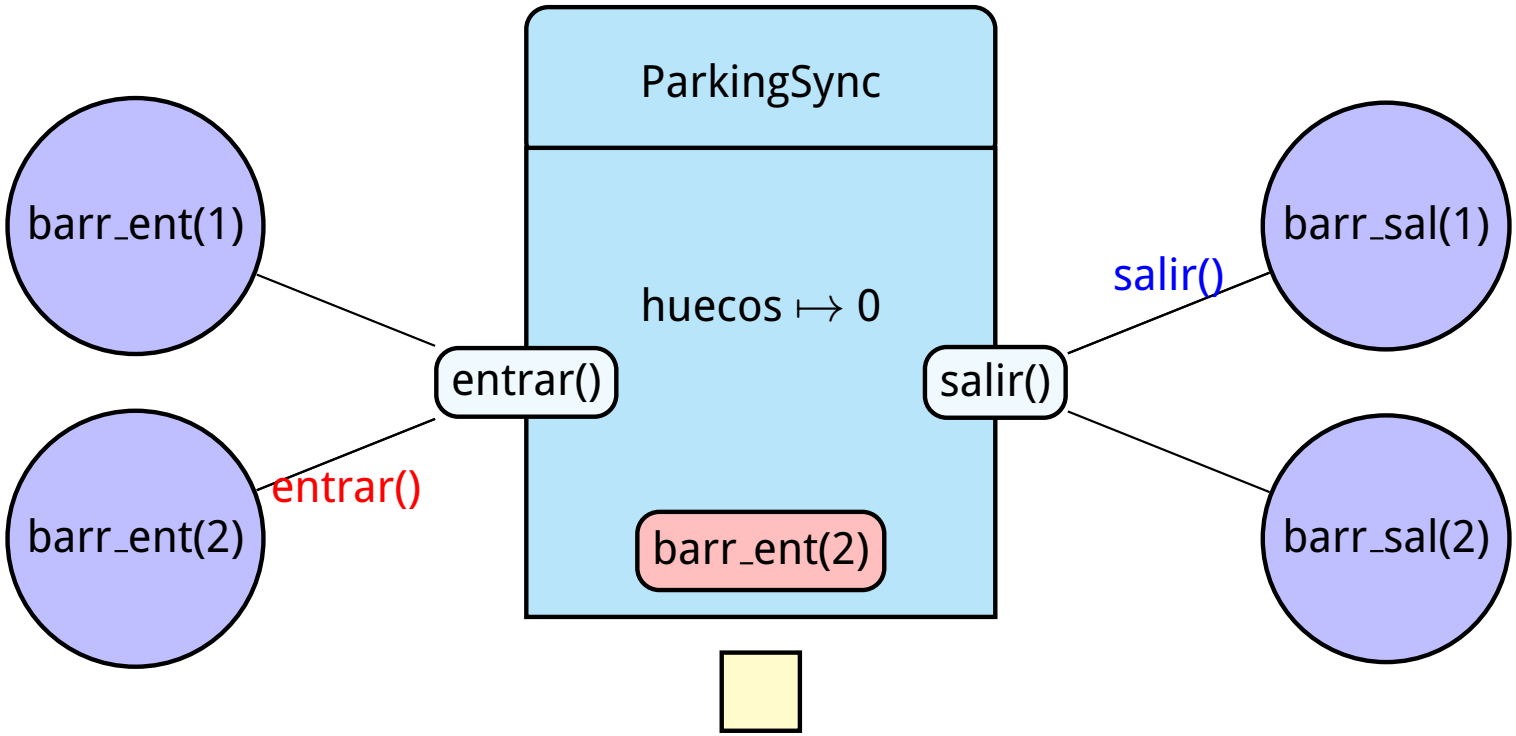
aquí, el waitset funciona como esperamos



POLITÉCNICA

modelo de ejecución

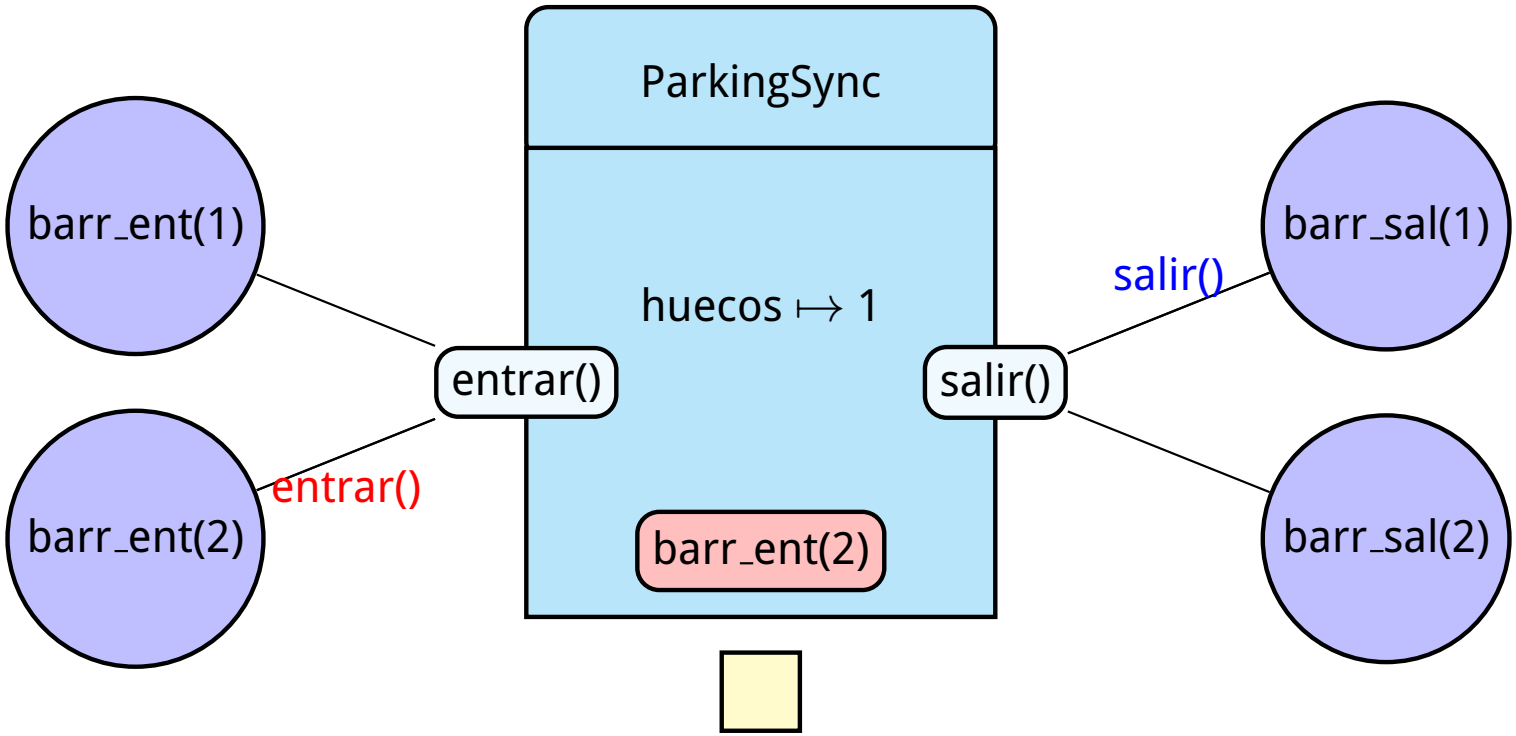
aquí, el waitset funciona como esperamos



POLITÉCNICA

modelo de ejecución

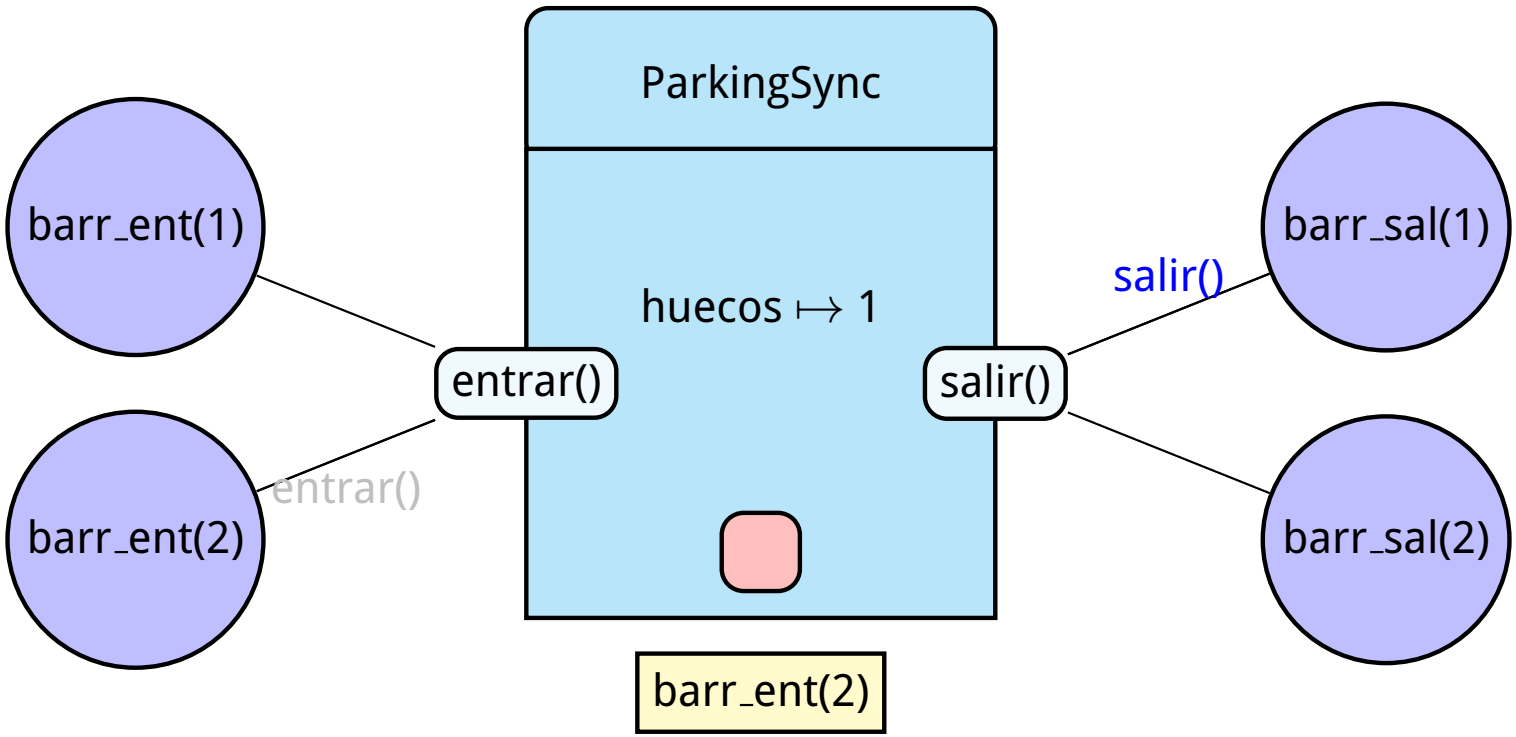
aquí, el waitset funciona como esperamos



POLITÉCNICA

modelo de ejecución

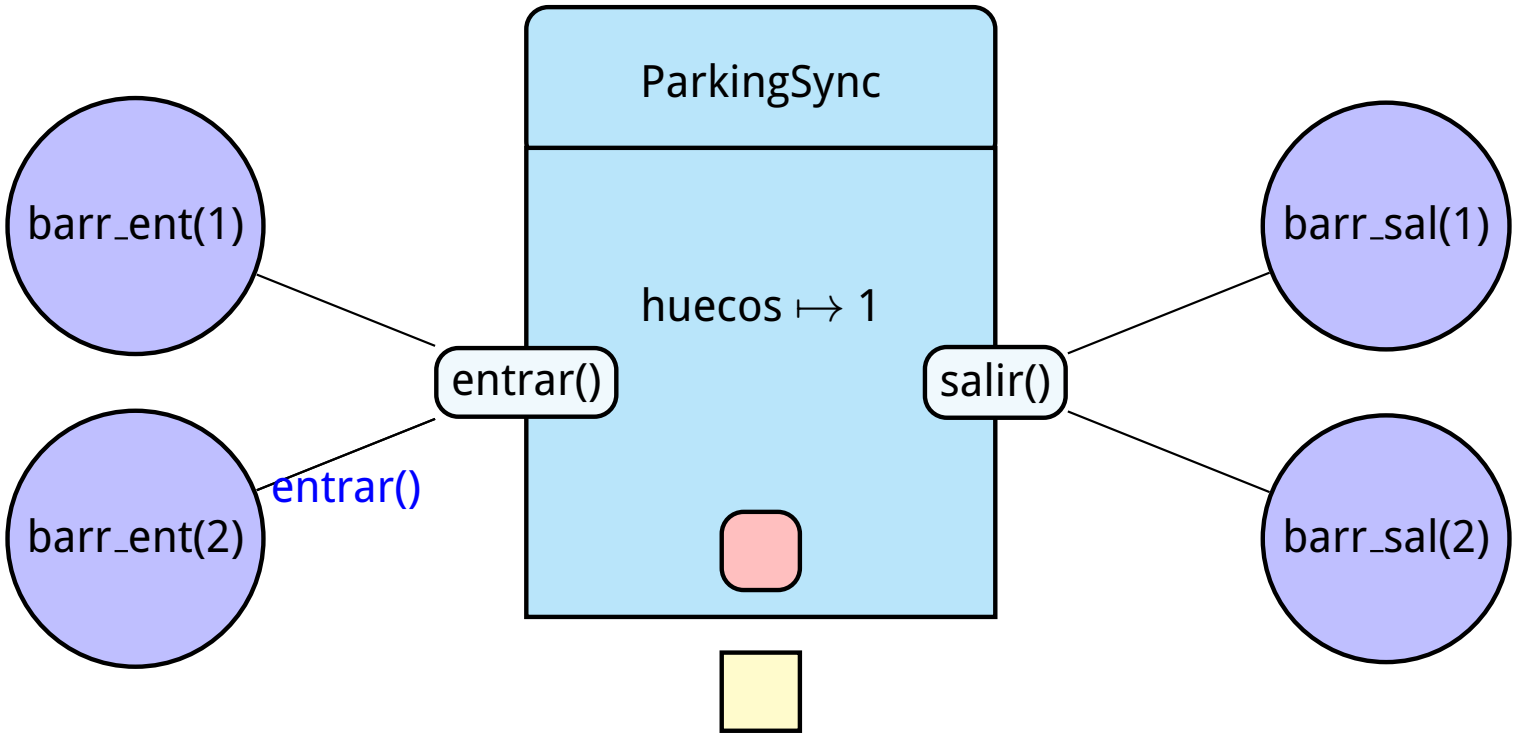
aquí, el waitset funciona como esperamos



POLITÉCNICA

modelo de ejecución

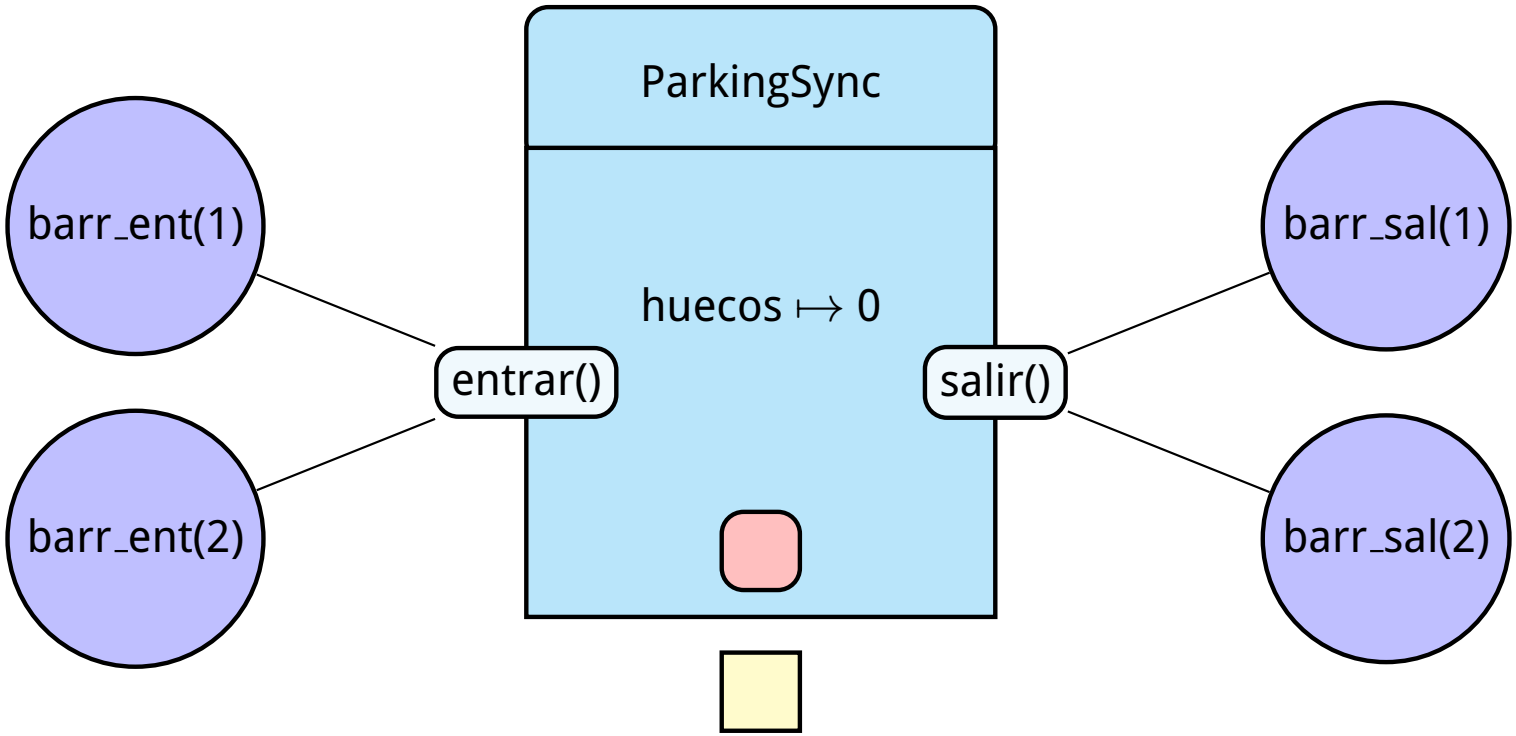
aquí, el waitset funciona como esperamos



POLITÉCNICA

modelo de ejecución

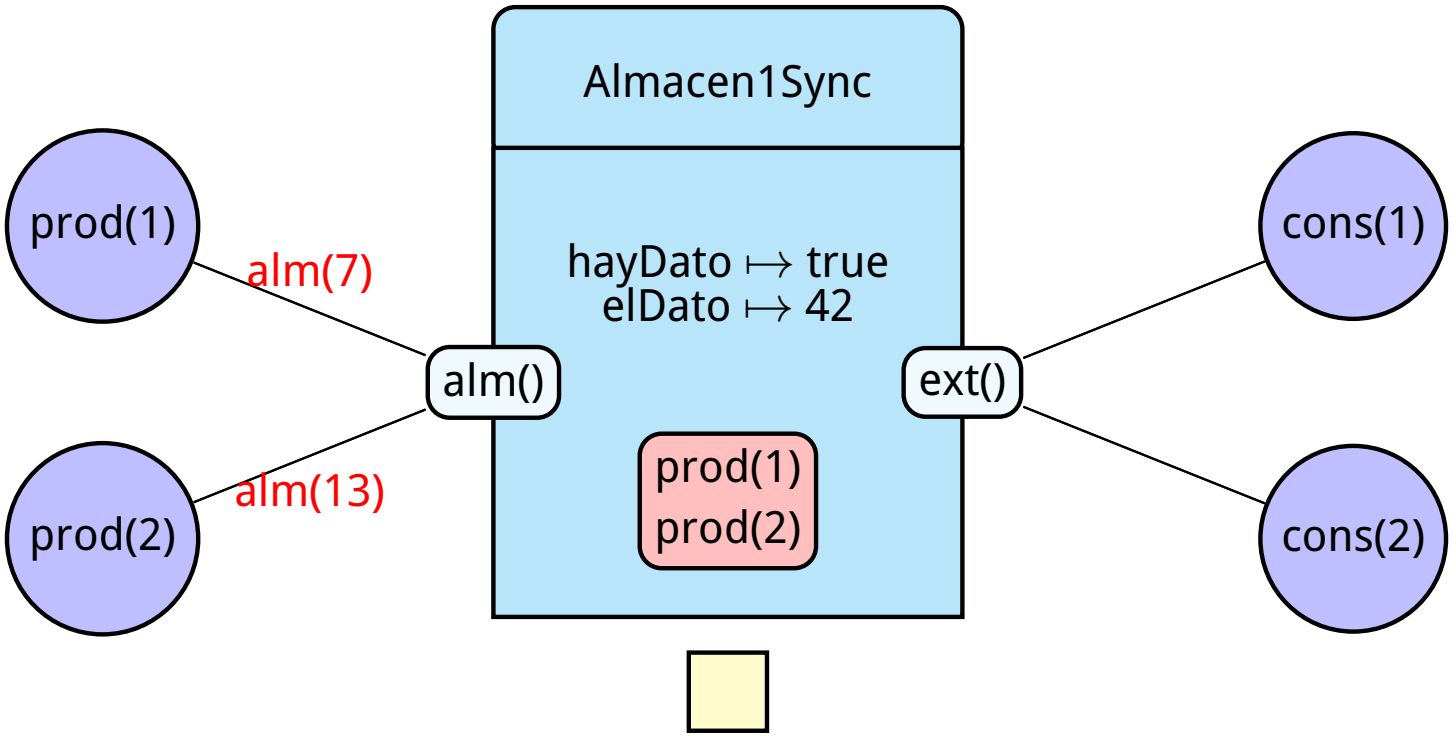
aquí, el waitset funciona como esperamos



POLITÉCNICA

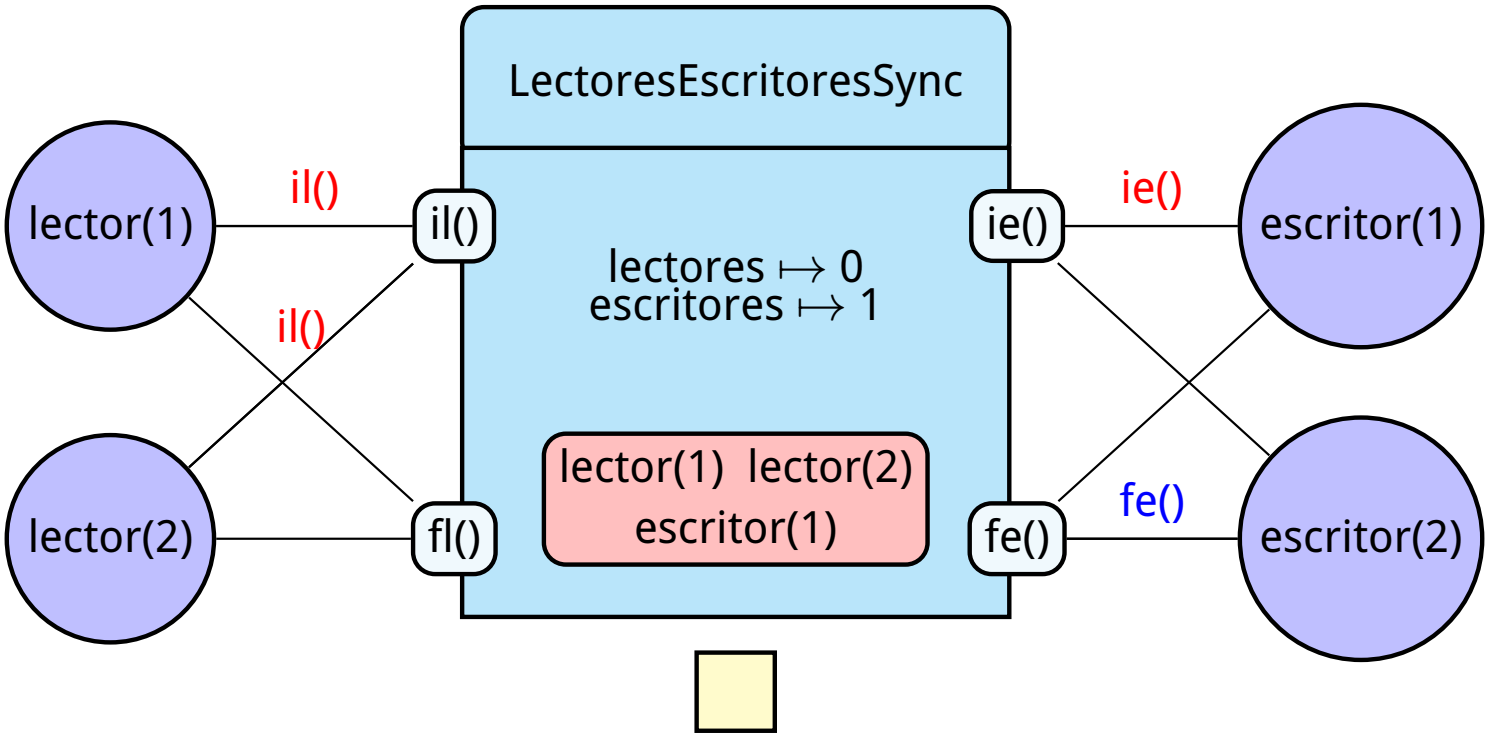
almacén de un dato con métodos sincronizados

¿a quién se desbloquea? ¿en qué orden se reingresa en el objeto?



lectores/escritores con métodos sincronizados

promiscuidad en el waitset



POLITÉCNICA

En conclusión...

- Los métodos sincronizados son una manera muy sencilla de resolver el acceso en exclusión mutua a objetos en Java.
- A la hora de programar la sincronización por condición se opta por unos mecanismos demasiado pobres y nunca bien justificados ni documentados que pueden acabar provocando ineficacia y otros problemas, como inanición.
- Esto ha dado a Java una mala fama de "lenguaje donde la concurrencia no escala bien", a pesar de que la máquina virtual de Java puede manejar un número considerable de threads.
- En el año 1999, Per Brinch-Hansen – uno de los proponentes del concepto de monitor – publicó un artículo titulado *Java's Insecure Parallelism* criticando alguna de las decisiones de diseño tomadas.
- Versiones posteriores de Java incluyeron librerías con implementaciones de monitores más fieles al espíritu original y que, como veremos en los próximos vídeos, permiten dar respuesta a muchas de las cuestiones planteadas hoy.

POLITÉCNICA

En conclusión...

- Los métodos sincronizados son una manera muy sencilla de resolver el acceso en exclusión mutua a objetos en Java.
- A la hora de programar la sincronización por condición se opta por unos mecanismos demasiado pobres y nunca bien justificados ni documentados que pueden acabar provocando ineficacia y otros problemas, como inanición.
- Esto ha dado a Java una mala fama de "lenguaje donde la concurrencia no escala bien", a pesar de que la máquina virtual de Java puede manejar un número considerable de threads.
- En el año 1999, Per Brinch-Hansen – uno de los proponentes del concepto de monitor – publicó un artículo titulado *Java's Insecure Parallelism* criticando alguna de las decisiones de diseño tomadas.
- Versiones posteriores de Java incluyeron librerías con implementaciones de monitores más fieles al espíritu original y que, como veremos en los próximos vídeos, permiten dar respuesta a muchas de las cuestiones planteadas hoy.

POLITÉCNICA

En conclusión...

- Los métodos sincronizados son una manera muy sencilla de resolver el acceso en exclusión mutua a objetos en Java.
- A la hora de programar la sincronización por condición se opta por unos mecanismos demasiado pobres y nunca bien justificados ni documentados que pueden acabar provocando ineficacia y otros problemas, como inanición.
- Esto ha dado a Java una mala fama de "lenguaje donde la concurrencia no escala bien", a pesar de que la máquina virtual de Java puede manejar un número considerable de threads.
- En el año 1999, Per Brinch-Hansen – uno de los proponentes del concepto de monitor – publicó un artículo titulado *Java's Insecure Parallelism* criticando alguna de las decisiones de diseño tomadas.
- Versiones posteriores de Java incluyeron librerías con implementaciones de monitores más fieles al espíritu original y que, como veremos en los próximos vídeos, permiten dar respuesta a muchas de las cuestiones planteadas hoy.

POLITÉCNICA

En conclusión...

- Los métodos sincronizados son una manera muy sencilla de resolver el acceso en exclusión mutua a objetos en Java.
- A la hora de programar la sincronización por condición se opta por unos mecanismos demasiado pobres y nunca bien justificados ni documentados que pueden acabar provocando ineficacia y otros problemas, como inanición.
- Esto ha dado a Java una mala fama de "lenguaje donde la concurrencia no escala bien", a pesar de que la máquina virtual de Java puede manejar un número considerable de threads.
- En el año 1999, Per Brinch-Hansen – uno de los proponentes del concepto de monitor – publicó un artículo titulado *Java's Insecure Parallelism* criticando alguna de las decisiones de diseño tomadas.
- Versiones posteriores de Java incluyeron librerías con implementaciones de monitores más fieles al espíritu original y que, como veremos en los próximos vídeos, permiten dar respuesta a muchas de las cuestiones planteadas hoy.

POLITÉCNICA

En conclusión...

- Los métodos sincronizados son una manera muy sencilla de resolver el acceso en exclusión mutua a objetos en Java.
- A la hora de programar la sincronización por condición se opta por unos mecanismos demasiado pobres y nunca bien justificados ni documentados que pueden acabar provocando ineficacia y otros problemas, como inanición.
- Esto ha dado a Java una mala fama de "lenguaje donde la concurrencia no escala bien", a pesar de que la máquina virtual de Java puede manejar un número considerable de threads.
- En el año 1999, Per Brinch-Hansen – uno de los proponentes del concepto de monitor – publicó un artículo titulado *Java's Insecure Parallelism* criticando alguna de las decisiones de diseño tomadas.
- Versiones posteriores de Java incluyeron librerías con implementaciones de monitores más fieles al espíritu original y que, como veremos en los próximos vídeos, permiten dar respuesta a muchas de las cuestiones planteadas hoy.

POLITÉCNICA