Sesión 17: Monitores – Introducción

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

- En los años 70 se hicieron varias propuestas de mecanismos de sincronización que fueran más allá de los semáforos y que encajasen mejor en las ideas de programación estructurada que se iban imponiendo.
- Per Brinch-Hansen en 1973 y C.A.R. Hoare en 1974 hicieron propuestas que tenían en común el tratar de manera separada la exclusión mutua y la sincronización por condición.
- El concepto de monitor influyó en el diseño de los mecanismos de sincronización originales de Java si bien la librería más fiel a la propuesta original es la llamada Locks & Conditions.
- En este curso usaremos nuestra propia librería de monitores (es.upm.babel.cclib.Monitor), con pequeñas – pero relevantes – diferencias con la que viene con Java.

POLITÉCNICA

introducción/motivación

buscando mecanismos de más alto nivel

- En los años 70 se hicieron varias propuestas de mecanismos de sincronización que fueran más allá de los semáforos y que encajasen mejor en las ideas de programación estructurada que se iban imponiendo.
- Per Brinch-Hansen en 1973 y C.A.R. Hoare en 1974 hicieron propuestas que tenían en común el tratar de manera separada la exclusión mutua y la sincronización por condición.
- El concepto de monitor influyó en el diseño de los mecanismos de sincronización originales de Java si bien la librería más fiel a la propuesta original es la llamada Locks & Conditions.
- En este curso usaremos nuestra propia librería de monitores (es.upm.babel.cclib.Monitor), con pequeñas – pero relevantes – diferencias con la que viene con Java.

POLITÉCNICA

introducción/motivación

buscando mecanismos de más alto nivel

- En los años 70 se hicieron varias propuestas de mecanismos de sincronización que fueran más allá de los semáforos y que encajasen mejor en las ideas de programación estructurada que se iban imponiendo.
- Per Brinch-Hansen en 1973 y C.A.R. Hoare en 1974 hicieron propuestas que tenían en común el tratar de manera separada la exclusión mutua y la sincronización por condición.
- El concepto de monitor influyó en el diseño de los mecanismos de sincronización originales de Java si bien la librería más fiel a la propuesta original es la llamada Locks & Conditions.
- En este curso usaremos nuestra propia librería de monitores (es.upm.babel.cclib.Monitor), con pequeñas – pero relevantes – diferencias con la que viene con Java.

POLITÉCNICA

introducción/motivación

buscando mecanismos de más alto nivel

- En los años 70 se hicieron varias propuestas de mecanismos de sincronización que fueran más allá de los semáforos y que encajasen mejor en las ideas de programación estructurada que se iban imponiendo.
- Per Brinch-Hansen en 1973 y C.A.R. Hoare en 1974 hicieron propuestas que tenían en común el tratar de manera separada la exclusión mutua y la sincronización por condición.
- El concepto de monitor influyó en el diseño de los mecanismos de sincronización originales de Java si bien la librería más fiel a la propuesta original es la llamada Locks & Conditions.
- En este curso usaremos nuestra propia librería de monitores (es.upm.babel.cclib.Monitor), con pequeñas – pero relevantes – diferencias con la que viene con Java.

POLITÉCNICA

monitores

no tan diferentes de los semáforos, pero separando usos

- La idea fundamental de los monitores consiste en tener dos tipos de entidades. Por un lado, los monitores en sí son los responsables de garantizar la ejecución en exclusión mutua de ciertos fragmentos de código. Por otra parte, las conditions o *condition queues* son las encargadas de manejar la sincronización por condición. En nuestra librería Java los monitores son una clase, pero otros lenguajes (incluso SSOO) pueden implementar el concepto de maneras muy diferentes. Lo que sigue, mientras no se indique lo contrario, se refiere a es.upm.babel.cclib.Monitor.
- Exclusión mutua: Los objetos de la clase Monitor proporcionan un método para solicitar permiso de ejecución en exclusión mutua (enter()) y otro método para liberar ese permiso (leave()). Por ejemplo:

```
mutex = new Monitor();
mutex.enter();
<seccion critica>
mutex.leave();
```

- A simple vista, un monitor se parece mucho a un semáforo inicializado a uno como los que usábamos para el protocolo de exclusión mutua, pero los monitores solo tienen dos estados: *libre* y *ocupado*. Cuando se crea, el monitor está libre. Al hacer enter() sobre un monitor libre pasa a estar ocupado. El proceso que hace enter() sobre un monitor ocupado se bloquea. Hacer leave() sobre un monitor ocupado lo libera si no hay procesos bloqueados en enter() – en caso contrario se desbloquea al más antiguo. No se puede hacer leave() sobre un monitor no adquirido previamente.

POLITÉCNICA

monitores

no tan diferentes de los semáforos, pero separando usos

- La idea fundamental de los monitores consiste en tener dos tipos de entidades. Por un lado, los monitores en sí son los responsables de garantizar la ejecución en exclusión mutua de ciertos fragmentos de código. Por otra parte, las conditions o *condition queues* son las encargadas de manejar la sincronización por condición. En nuestra librería Java los monitores son una clase, pero otros lenguajes (incluso SSOO) pueden implementar el concepto de maneras muy diferentes. Lo que sigue, mientras no se indique lo contrario, se refiere a es.upm.babel.cclib.Monitor.
- Exclusión mutua: Los objetos de la clase Monitor proporcionan un método para solicitar permiso de ejecución en exclusión mutua (enter()) y otro método para liberar ese permiso (leave()). Por ejemplo:

```
mutex = new Monitor();
mutex.enter();
<seccion critica>
mutex.leave();
```

- A simple vista, un monitor se parece mucho a un semáforo inicializado a uno como los que usábamos para el protocolo de exclusión mutua, pero los monitores solo tienen dos estados: *libre* y *ocupado*. Cuando se crea, el monitor está libre. Al hacer enter() sobre un monitor libre pasa a estar ocupado. El proceso que hace enter() sobre un monitor ocupado se bloquea. Hacer leave() sobre un monitor ocupado lo libera si no hay procesos bloqueados en enter() – en caso contrario se desbloquea al más antiguo. No se puede hacer leave() sobre un monitor no adquirido previamente.

POLITÉCNICA

monitores

no tan diferentes de los semáforos, pero separando usos

- La idea fundamental de los monitores consiste en tener dos tipos de entidades. Por un lado, los monitores en sí son los responsables de garantizar la ejecución en exclusión mutua de ciertos fragmentos de código. Por otra parte, las conditions o *condition queues* son las encargadas de manejar la sincronización por condición. En nuestra librería Java los monitores son una clase, pero otros lenguajes (incluso SSOO) pueden implementar el concepto de maneras muy diferentes. Lo que sigue, mientras no se indique lo contrario, se refiere a es.upm.babel.cclib.Monitor.
- Exclusión mutua: Los objetos de la clase Monitor proporcionan un método para solicitar permiso de ejecución en exclusión mutua (enter()) y otro método para liberar ese permiso (leave()). Por ejemplo:

```
mutex = new Monitor();
mutex.enter();
<seccion critica>
mutex.leave();
```

- A simple vista, un monitor se parece mucho a un semáforo inicializado a uno como los que usábamos para el protocolo de exclusión mutua, pero los monitores solo tienen dos estados: *libre* y *ocupado*. Cuando se crea, el monitor está libre. Al hacer enter() sobre un monitor libre pasa a estar ocupado. El proceso que hace enter() sobre un monitor ocupado se bloquea. Hacer leave() sobre un monitor ocupado lo libera si no hay procesos bloqueados en enter() – en caso contrario se desbloquea al más antiguo. No se puede hacer leave() sobre un monitor no adquirido previamente.

POLITÉCNICA

ejemplo: contador compartido

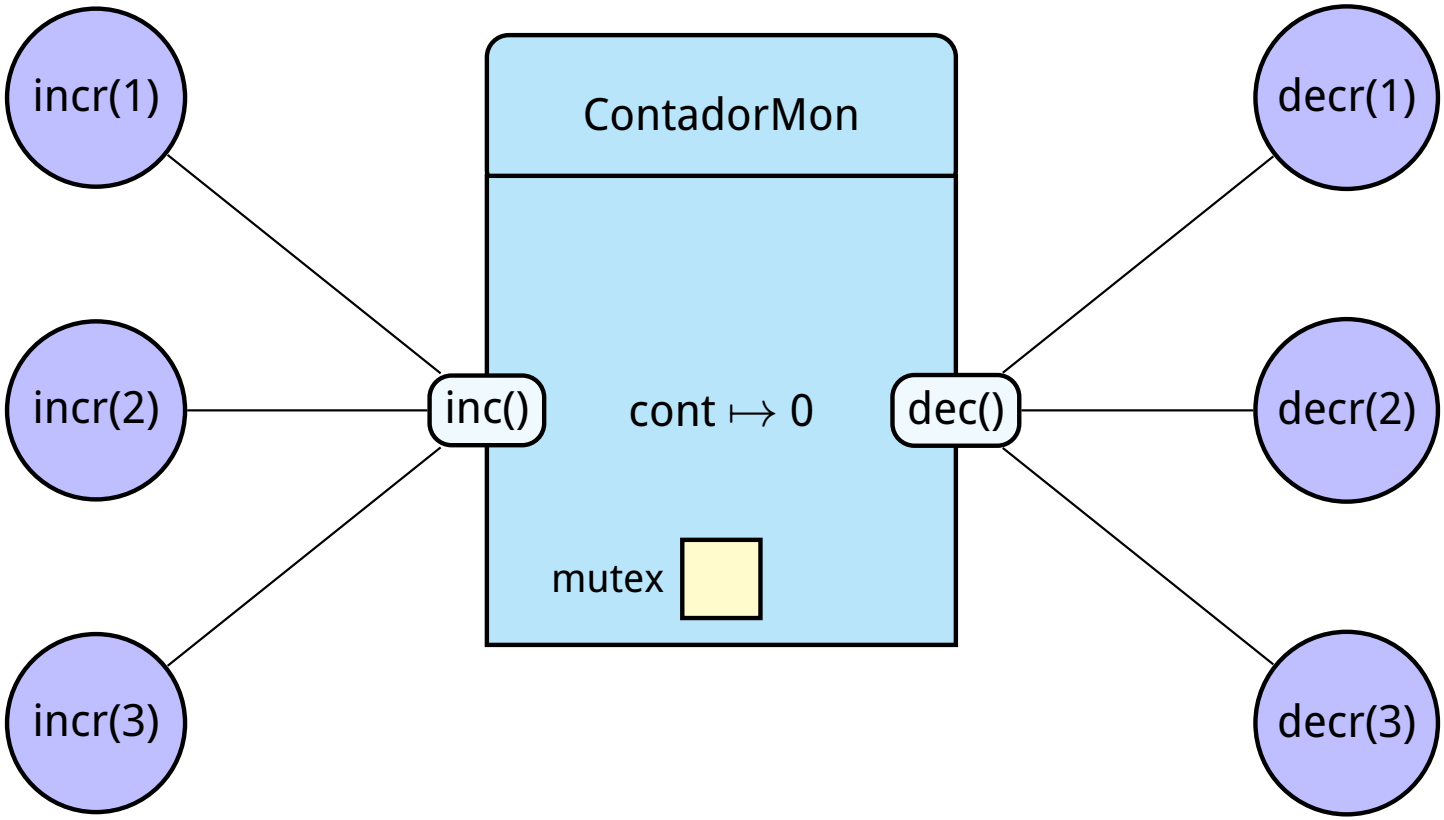
solo exclusión mutua

```
class ContadorMon {
    private int cont;
    private Monitor mutex;

    public ContadorMon (int n) {
        this.cont = n;
    }
    public int getValue() {
        int value;
        mutex.enter();
        value = this.cont;
        mutex.leave();
        return value;
    }
    public void inc () {
        mutex.enter();
        this.cont++;
        mutex.leave();
    }
    public void dec () {
        mutex.enter();
        this.cont--;
        mutex.leave();
    }
}
```

POLITÉCNICA

modelo de ejecución

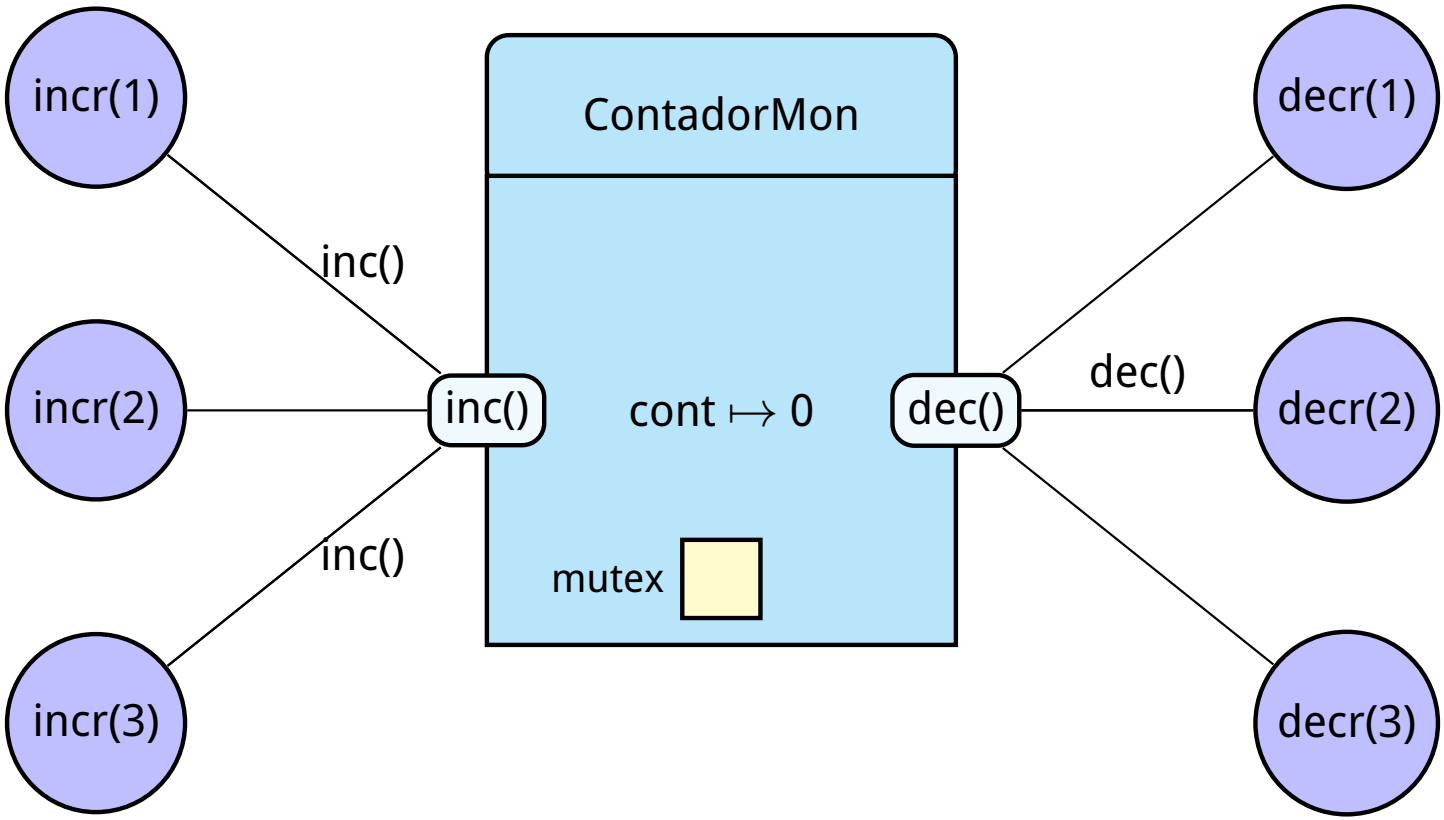
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

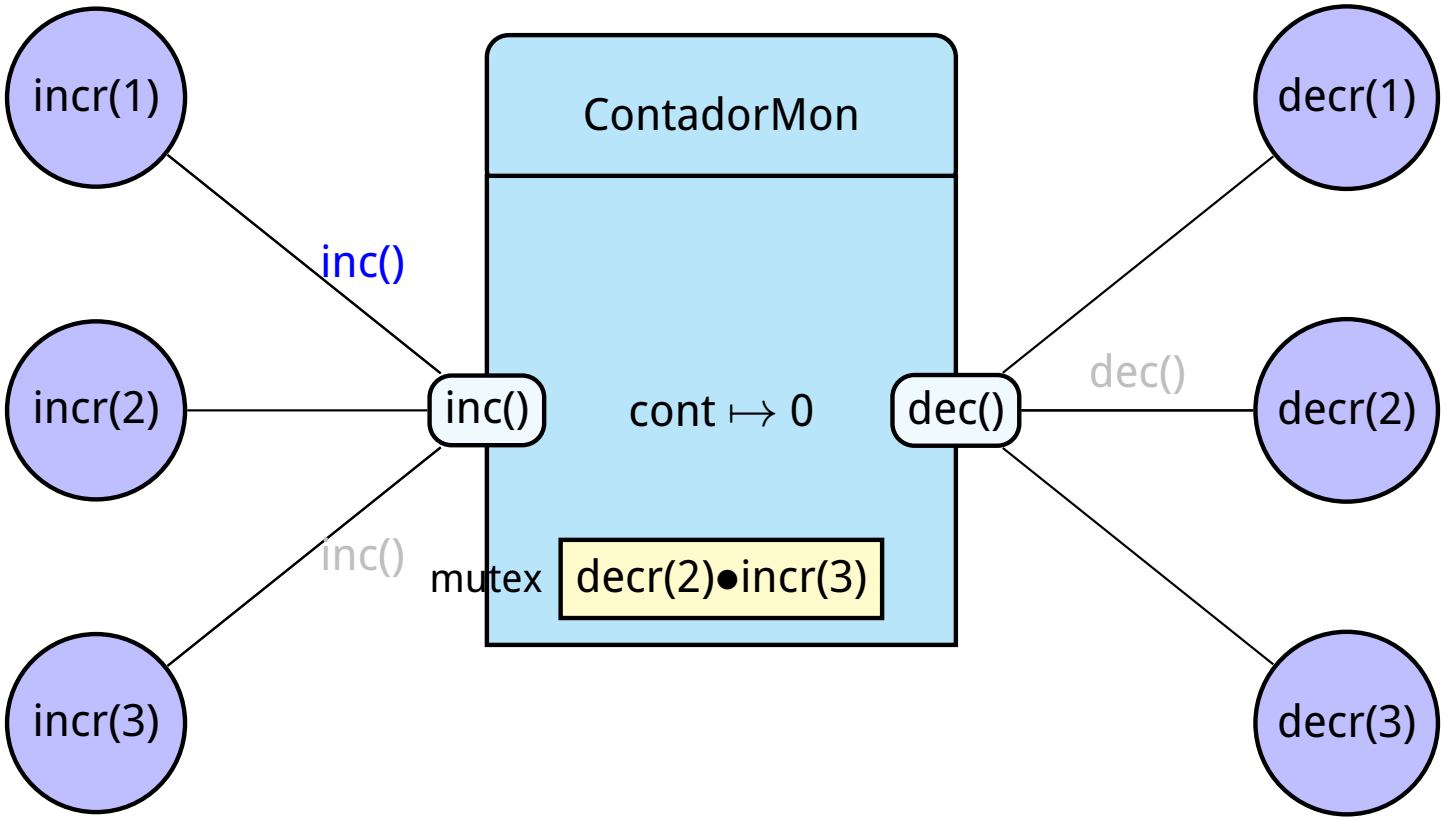
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

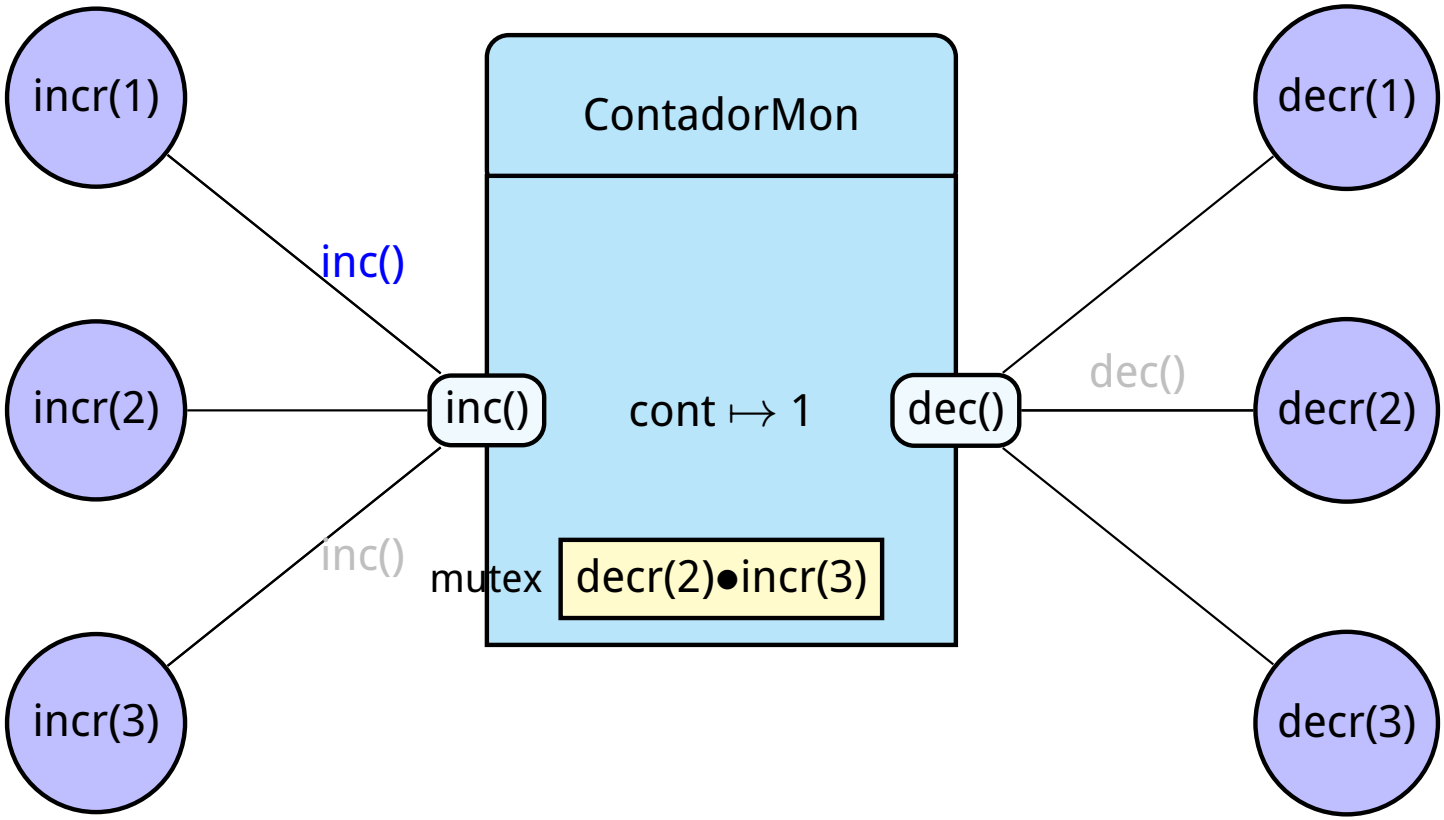
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

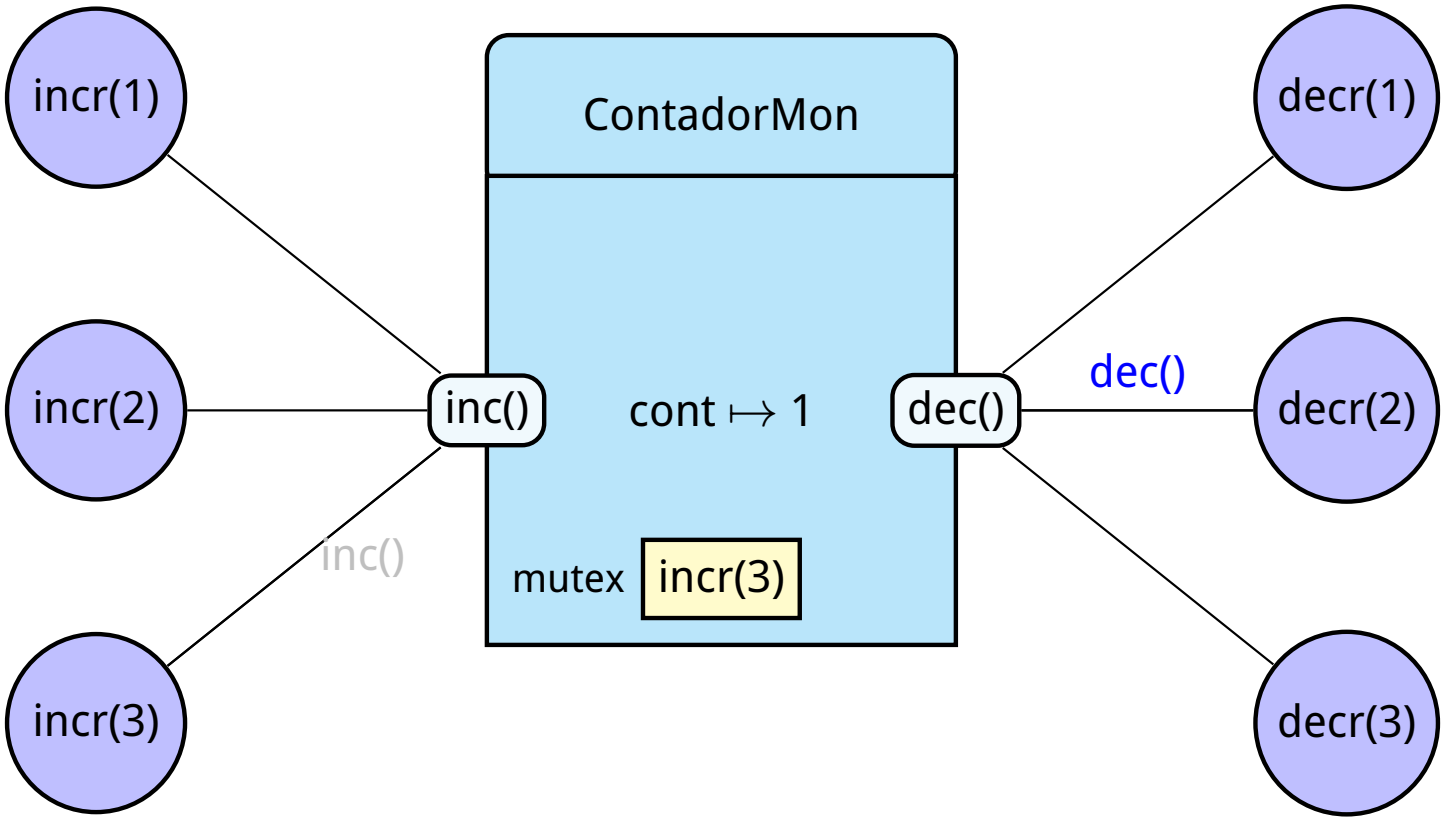
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

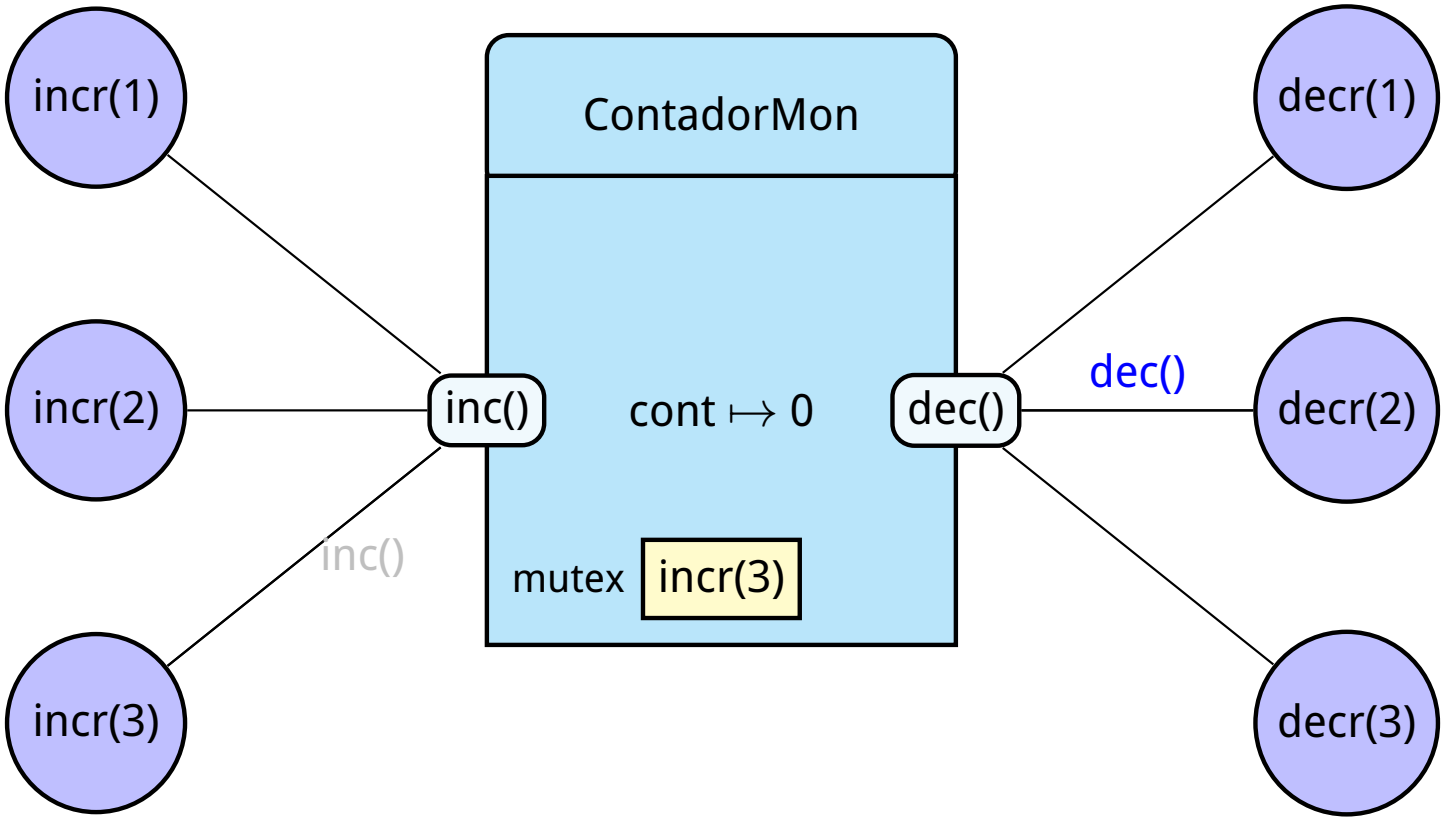
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

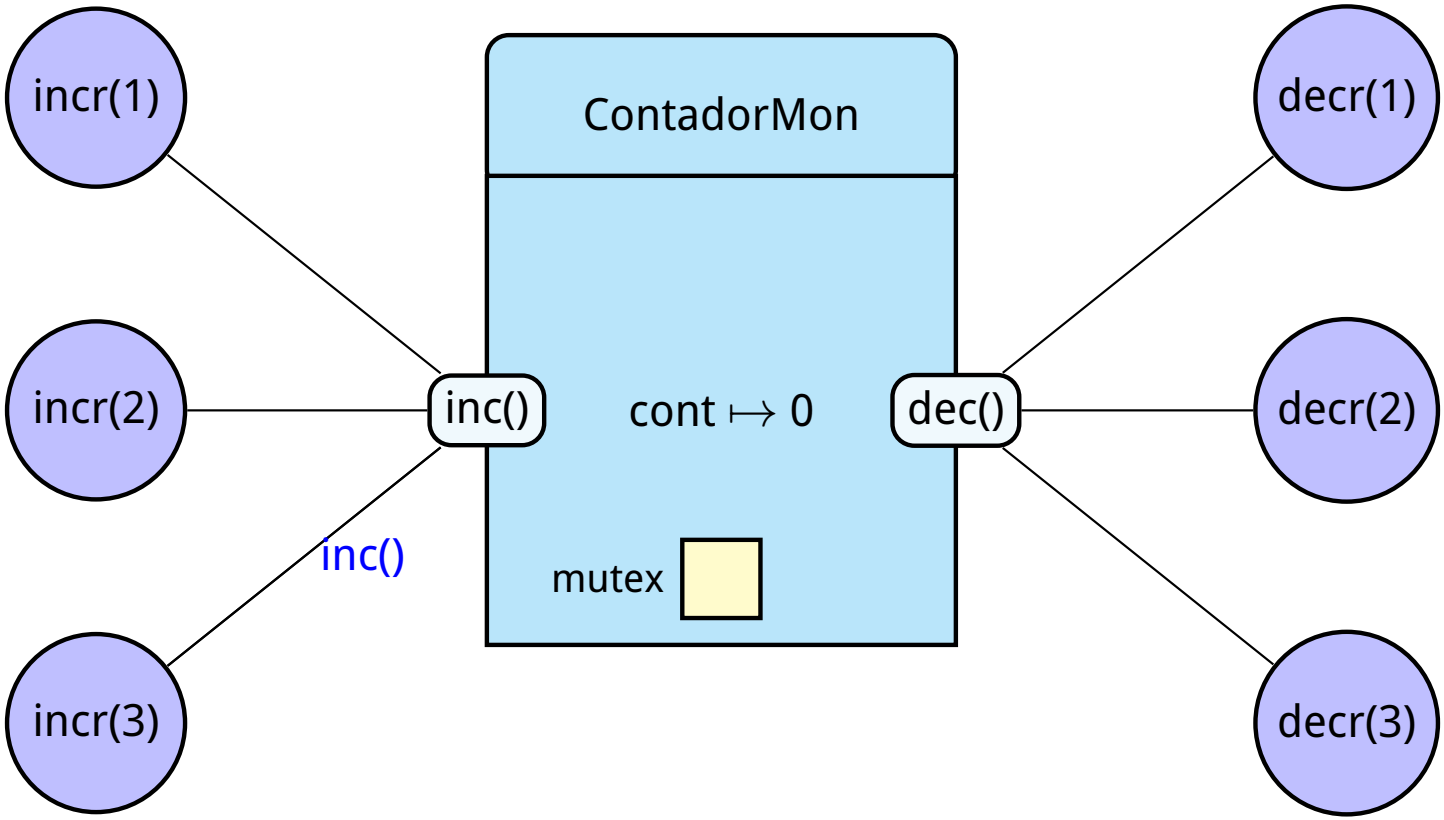
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

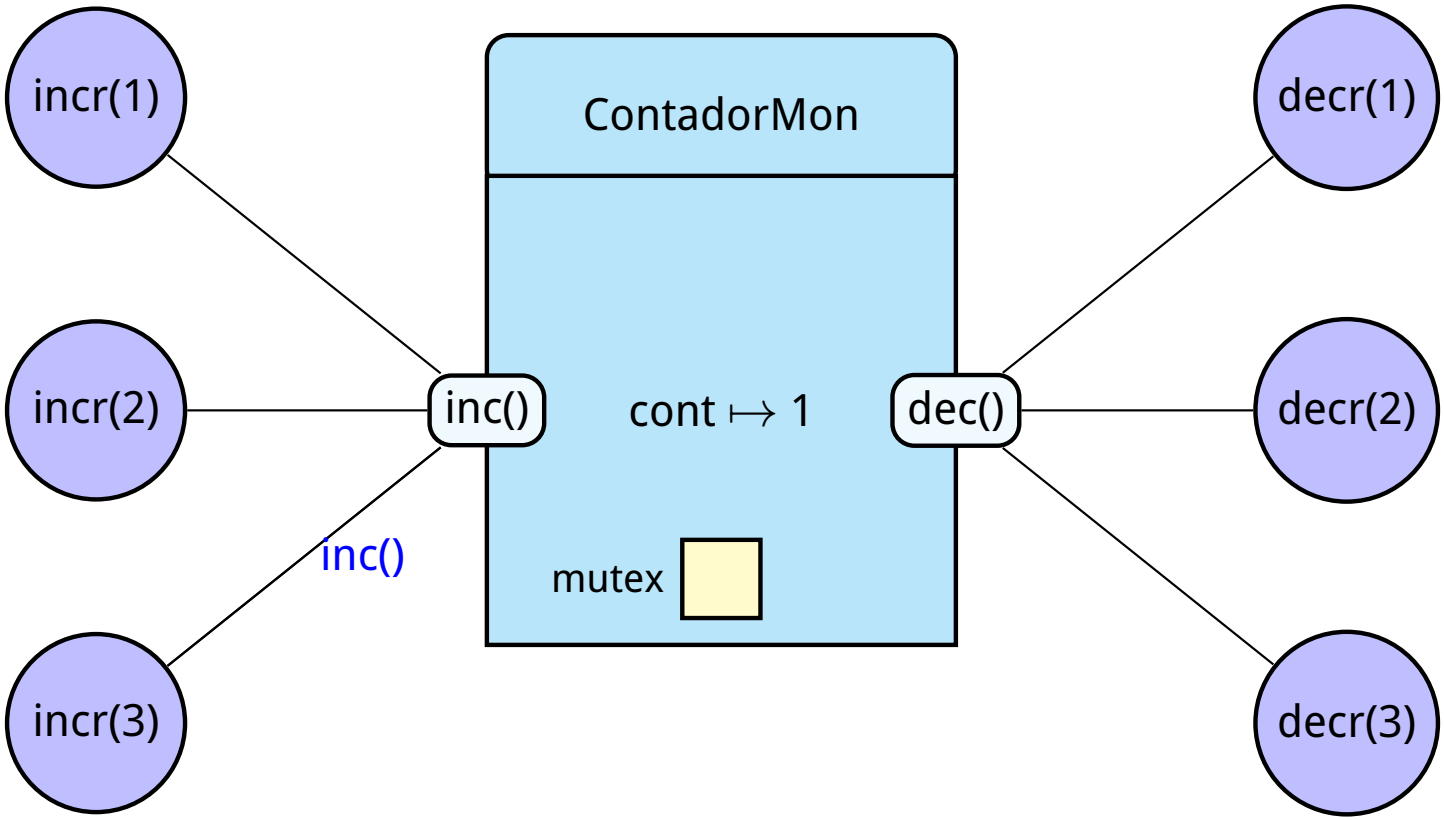
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

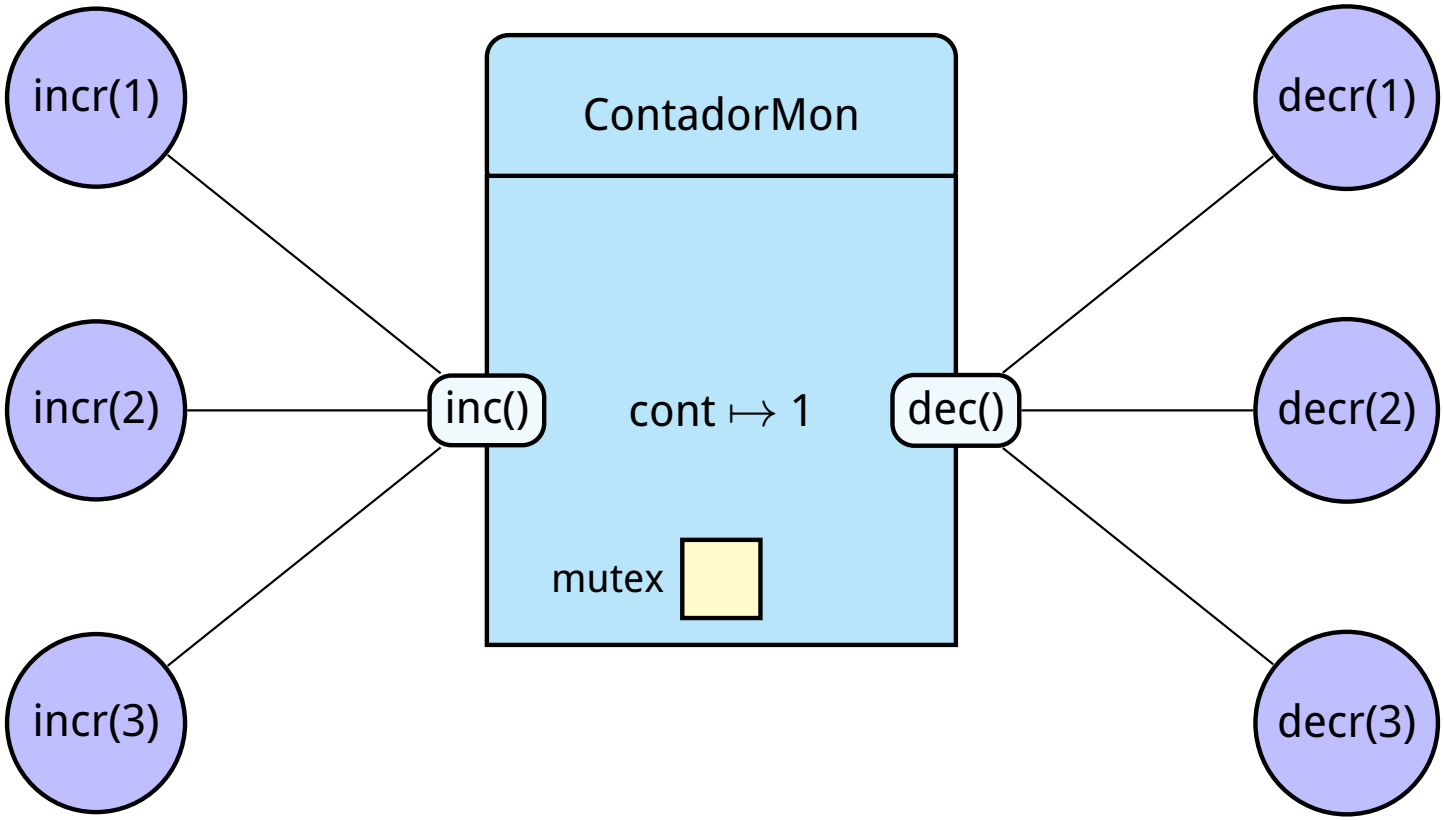
visualizando la exclusión mutua



POLITÉCNICA

modelo de ejecución

visualizando la exclusión mutua



POLITÉCNICA

monitores (cont.)

tratamos ahora la sincronización por condición

- Sincronización por condición: Con cada objeto de clase Monitor podemos asociar objetos (en número variable) de la clase Monitor.Cond que son *waitsets* con política FIFO. Estos objetos son las condition queues.
- Las condition queues se construyen a partir de monitores usando el método newCond() de la clase Monitor. Esto garantiza la asociación de cada cola con un único monitor.
- El bloqueo de hilos se realiza llamando al método await() de la clase Monitor.Cond. El proceso bloqueado se apunta al final de la cola interna de la condition queue. Para ejecutar await() es imprescindible haber adquirido el monitor asociado con la condition en cuestión, y el bloqueo conlleva la liberación automática de dicho monitor.
- ¿Cómo se desbloquean los hilos en una condition queue? Otro thread que haya ganado acceso exclusivo al monitor asociado puede ejecutar el método signal() de la clase Monitor.Cond. Esto saca al primer hilo de la cola y lo coloca al principio de la cola del monitor. En caso de no haber hilos bloqueados en una condition el signal() no tiene efecto.

POLITÉCNICA

monitores (cont.)

tratamos ahora la sincronización por condición

- Sincronización por condición: Con cada objeto de clase Monitor podemos asociar objetos (en número variable) de la clase Monitor.Cond que son *waitsets* con política FIFO. Estos objetos son las condition queues.
- Las condition queues se construyen a partir de monitores usando el método newCond() de la clase Monitor. Esto garantiza la asociación de cada cola con un único monitor.
- El bloqueo de hilos se realiza llamando al método await() de la clase Monitor.Cond. El proceso bloqueado se apunta al final de la cola interna de la condition queue. Para ejecutar await() es imprescindible haber adquirido el monitor asociado con la condition en cuestión, y el bloqueo conlleva la liberación automática de dicho monitor.
- ¿Cómo se desbloquean los hilos en una condition queue? Otro thread que haya ganado acceso exclusivo al monitor asociado puede ejecutar el método signal() de la clase Monitor.Cond. Esto saca al primer hilo de la cola y lo coloca al principio de la cola del monitor. En caso de no haber hilos bloqueados en una condition el signal() no tiene efecto.

POLITÉCNICA

monitores (cont.)

tratamos ahora la sincronización por condición

- Sincronización por condición: Con cada objeto de clase Monitor podemos asociar objetos (en número variable) de la clase Monitor.Cond que son *waitsets* con política FIFO. Estos objetos son las condition queues.
- Las condition queues se construyen a partir de monitores usando el método newCond() de la clase Monitor. Esto garantiza la asociación de cada cola con un único monitor.
- El bloqueo de hilos se realiza llamando al método await() de la clase Monitor.Cond. El proceso bloqueado se apunta al final de la cola interna de la condition queue. Para ejecutar await() es imprescindible haber adquirido el monitor asociado con la condition en cuestión, y el bloqueo conlleva la liberación automática de dicho monitor.
- ¿Cómo se desbloquean los hilos en una condition queue? Otro thread que haya ganado acceso exclusivo al monitor asociado puede ejecutar el método signal() de la clase Monitor.Cond. Esto saca al primer hilo de la cola y lo coloca al principio de la cola del monitor. En caso de no haber hilos bloqueados en una condition el signal() no tiene efecto.

POLITÉCNICA

monitores (cont.)

tratamos ahora la sincronización por condición

- Sincronización por condición: Con cada objeto de clase Monitor podemos asociar objetos (en número variable) de la clase Monitor.Cond que son *waitsets* con política FIFO. Estos objetos son las condition queues.
- Las condition queues se construyen a partir de monitores usando el método newCond() de la clase Monitor. Esto garantiza la asociación de cada cola con un único monitor.
- El bloqueo de hilos se realiza llamando al método await() de la clase Monitor.Cond. El proceso bloqueado se apunta al final de la cola interna de la condition queue. Para ejecutar await() es imprescindible haber adquirido el monitor asociado con la condition en cuestión, y el bloqueo conlleva la liberación automática de dicho monitor.
- ¿Cómo se desbloquean los hilos en una condition queue? Otro thread que haya ganado acceso exclusivo al monitor asociado puede ejecutar el método signal() de la clase Monitor.Cond. Esto saca al primer hilo de la cola y lo coloca al principio de la cola del monitor. En caso de no haber hilos bloqueados en una condition el signal() no tiene efecto.

POLITÉCNICA

ejemplo: aparcamiento

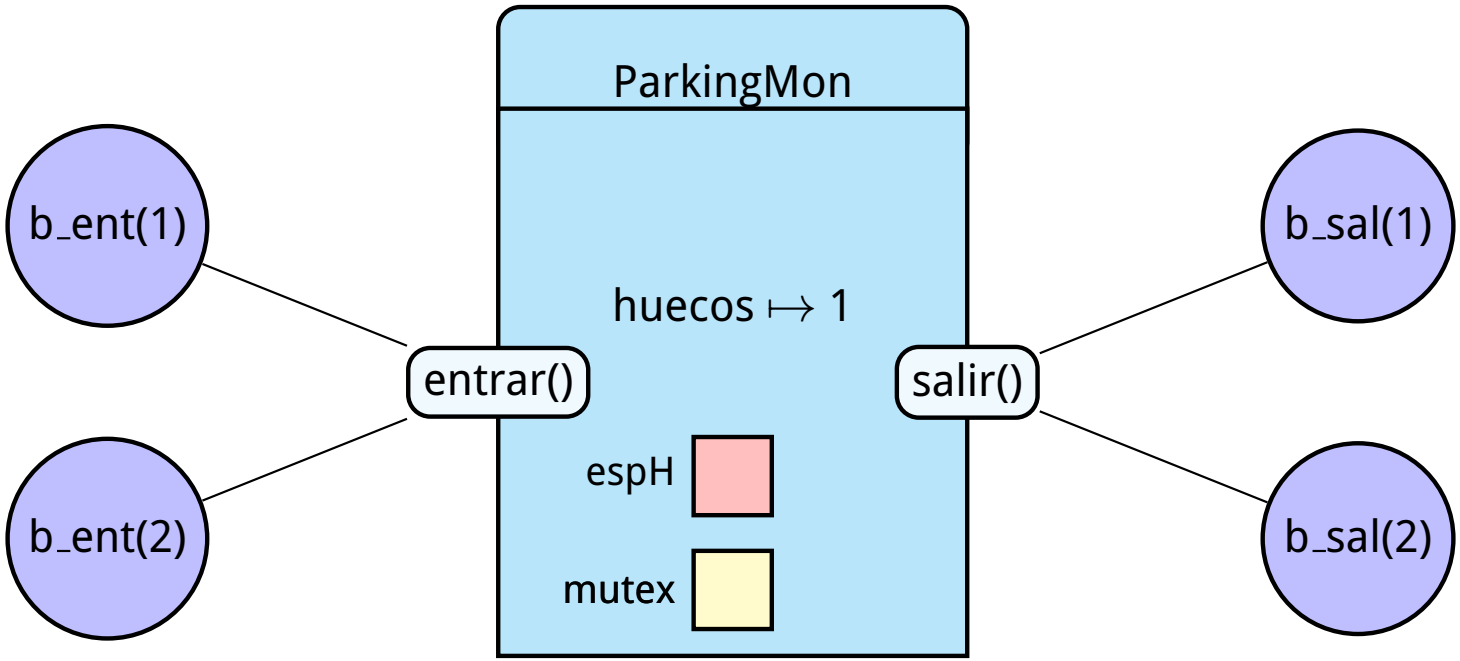
añadimos sincronización por condición al contador compartido

```
public class ParkingMon {
    private Monitor mutex;
    private Monitor.Cond esperarHueco;

    private int huecos;
    private static int CAP = 5;

    public Parking() {
        mutex = new Monitor();
        esperarHueco = mutex.newCond();
        this.huecos = CAP;
    }
    public void entrar() {
        mutex.enter();
        if(this.huecos == 0) {
            esperarHueco.await();
        }
        this.huecos = this.huecos - 1;
        mutex.leave();
    }
    public void sal() {
        mutex.enter();
        this.huecos = this.huecos + 1;
        esperarHueco.signal();
        mutex.leave();
    }
}
```

POLITÉCNICA

modelo de ejecución

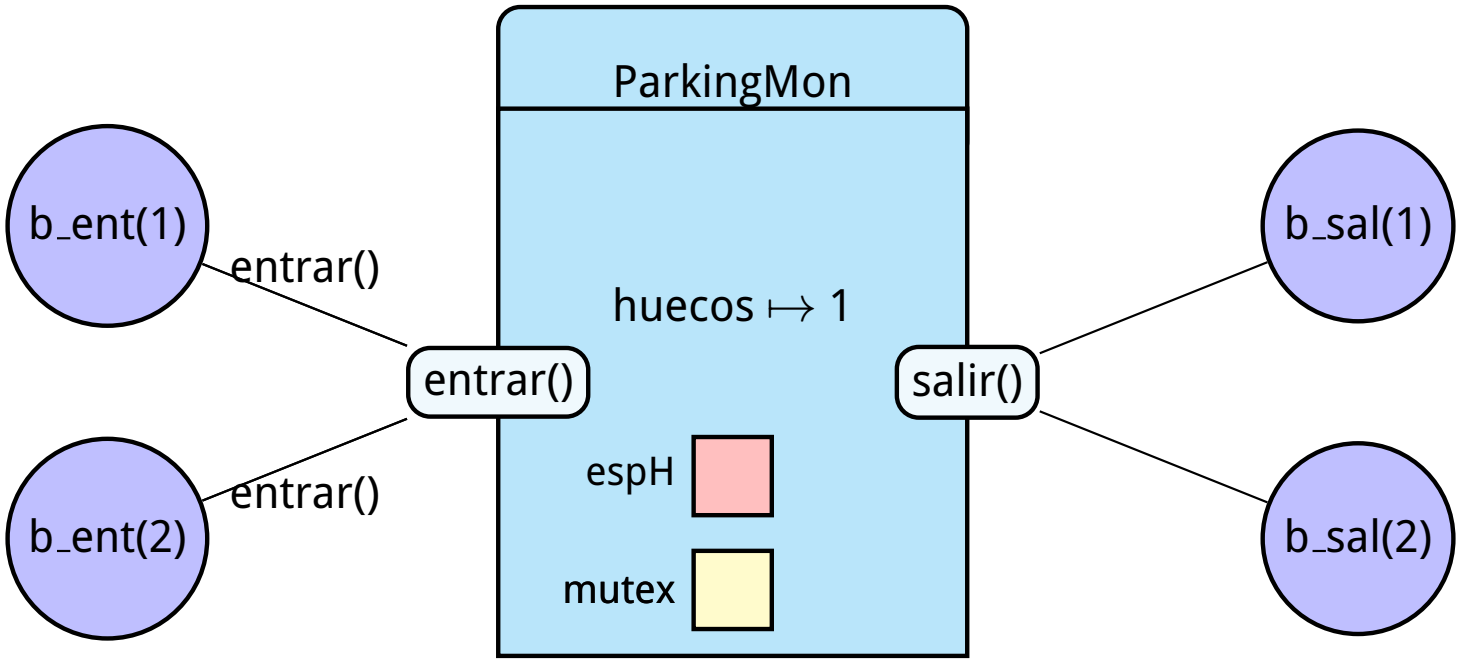
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

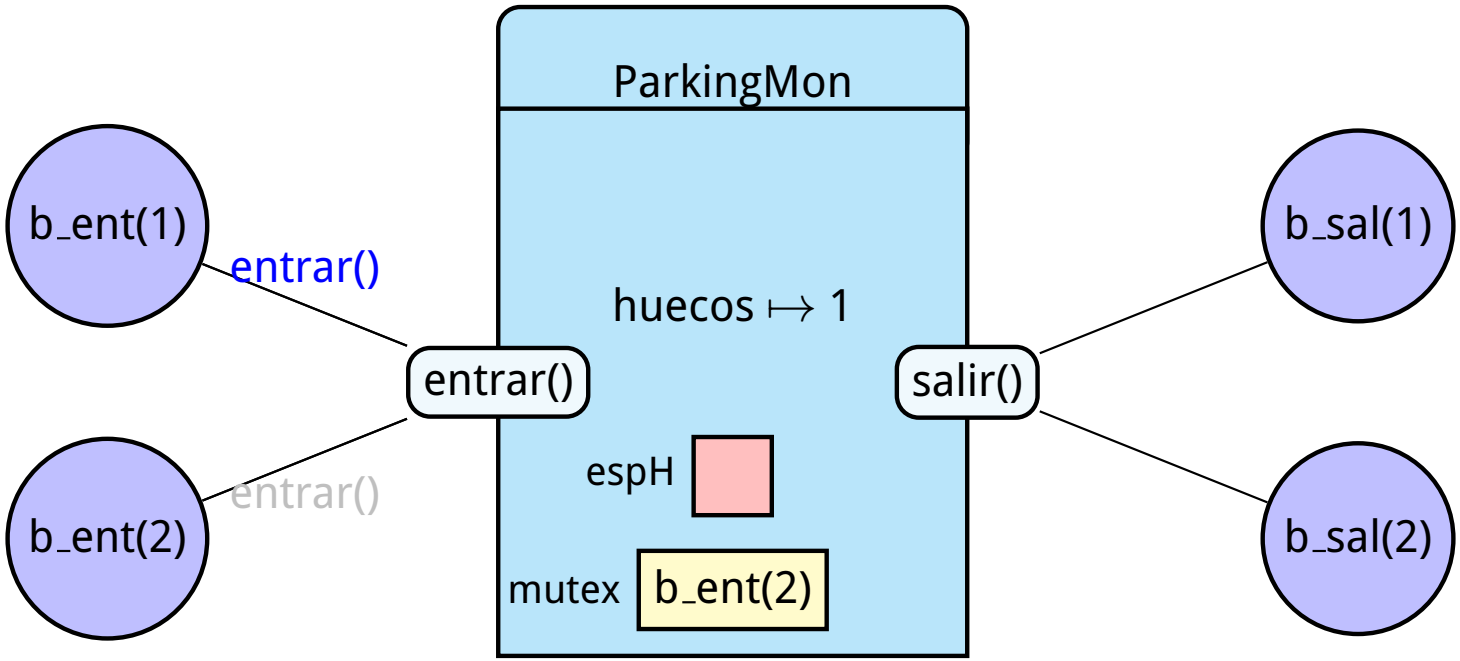
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

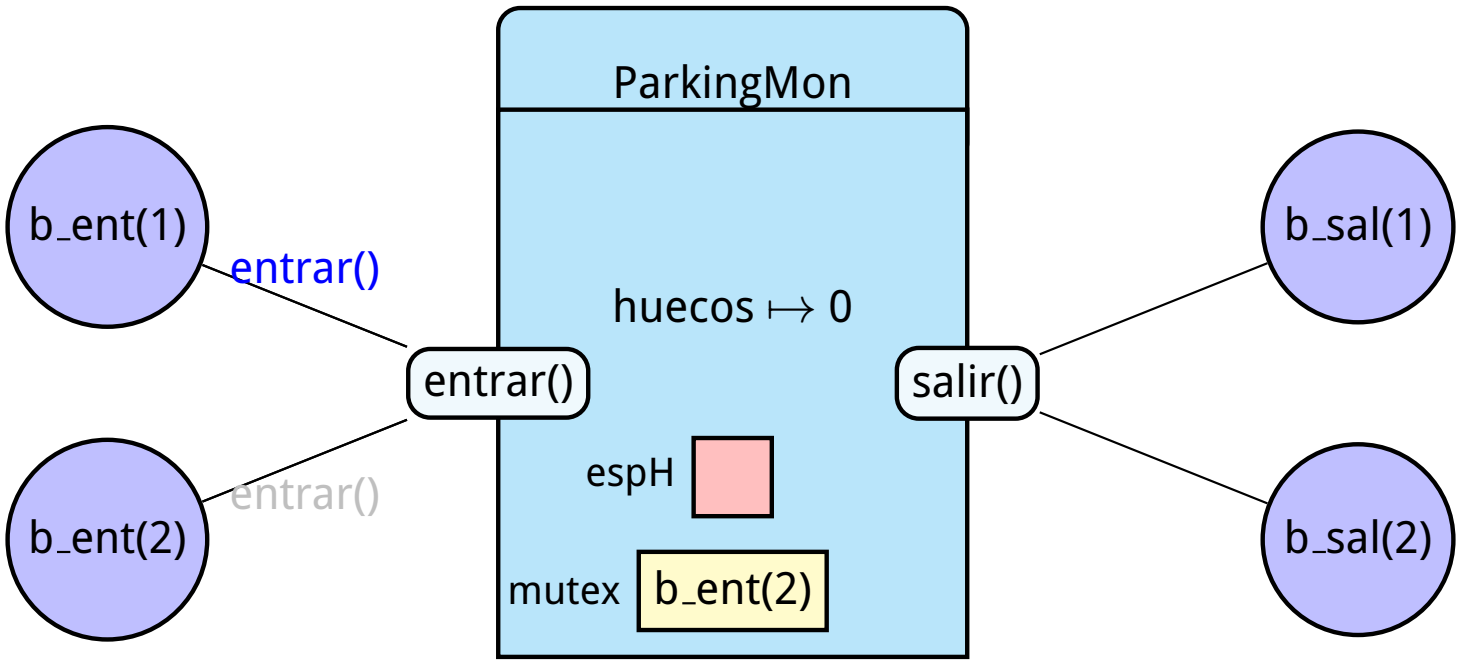
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

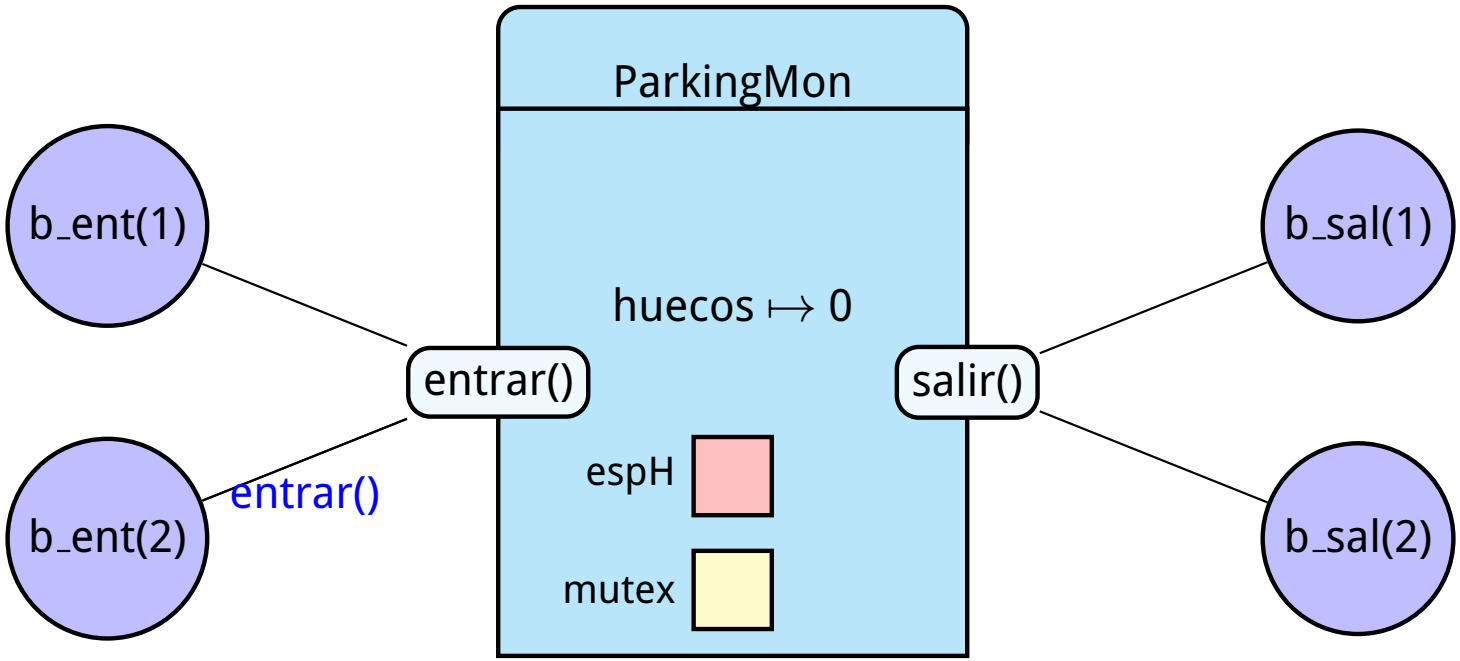
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

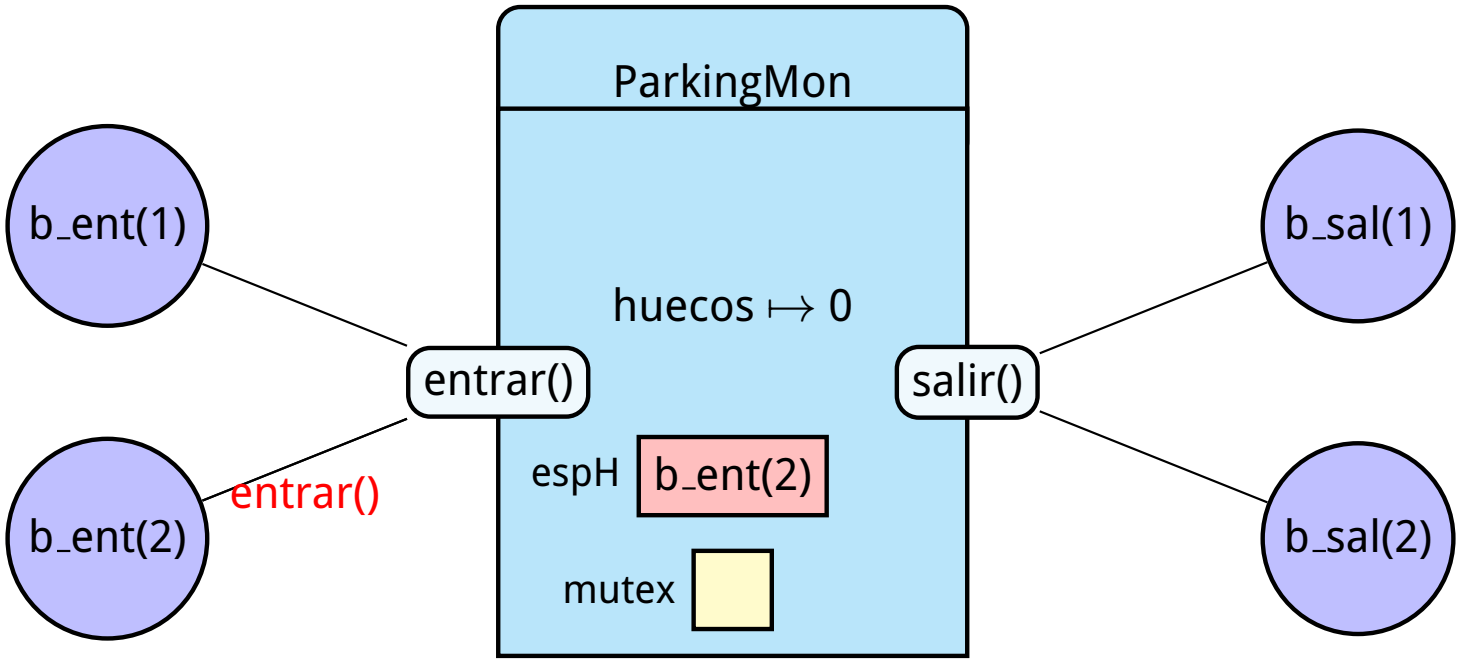
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

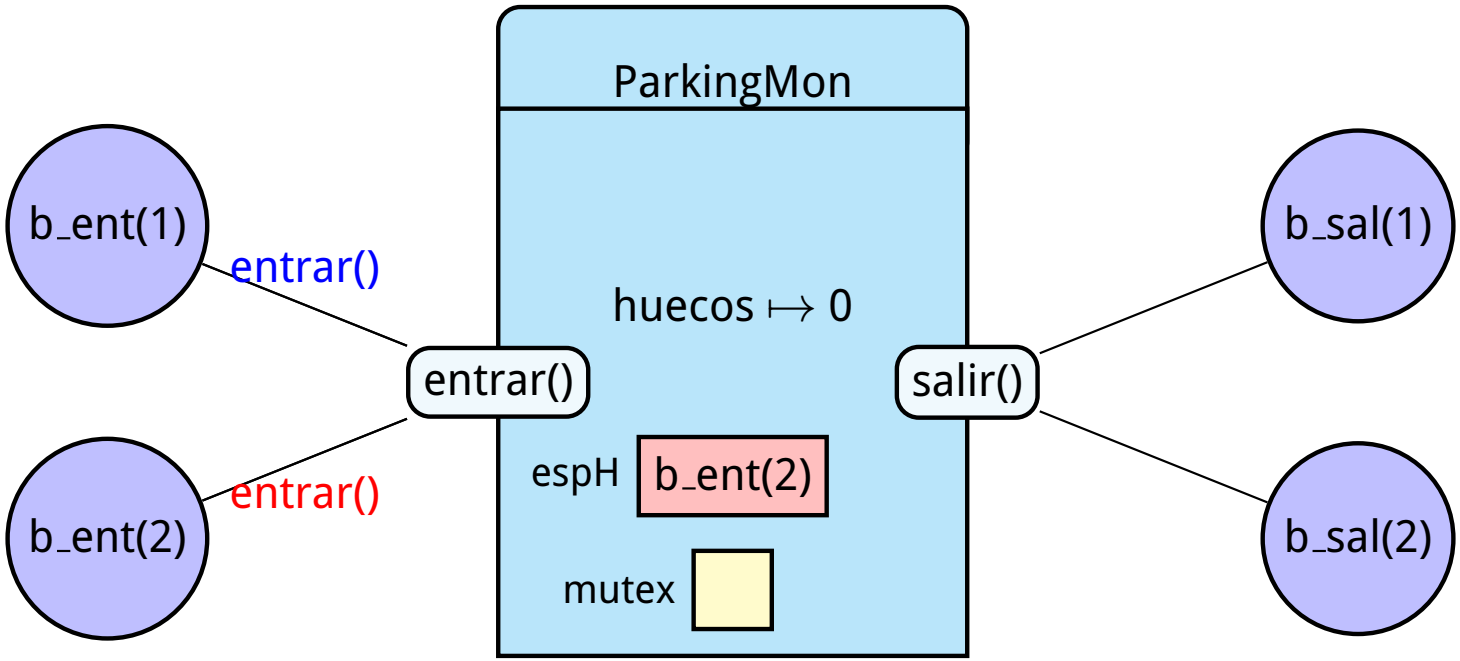
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

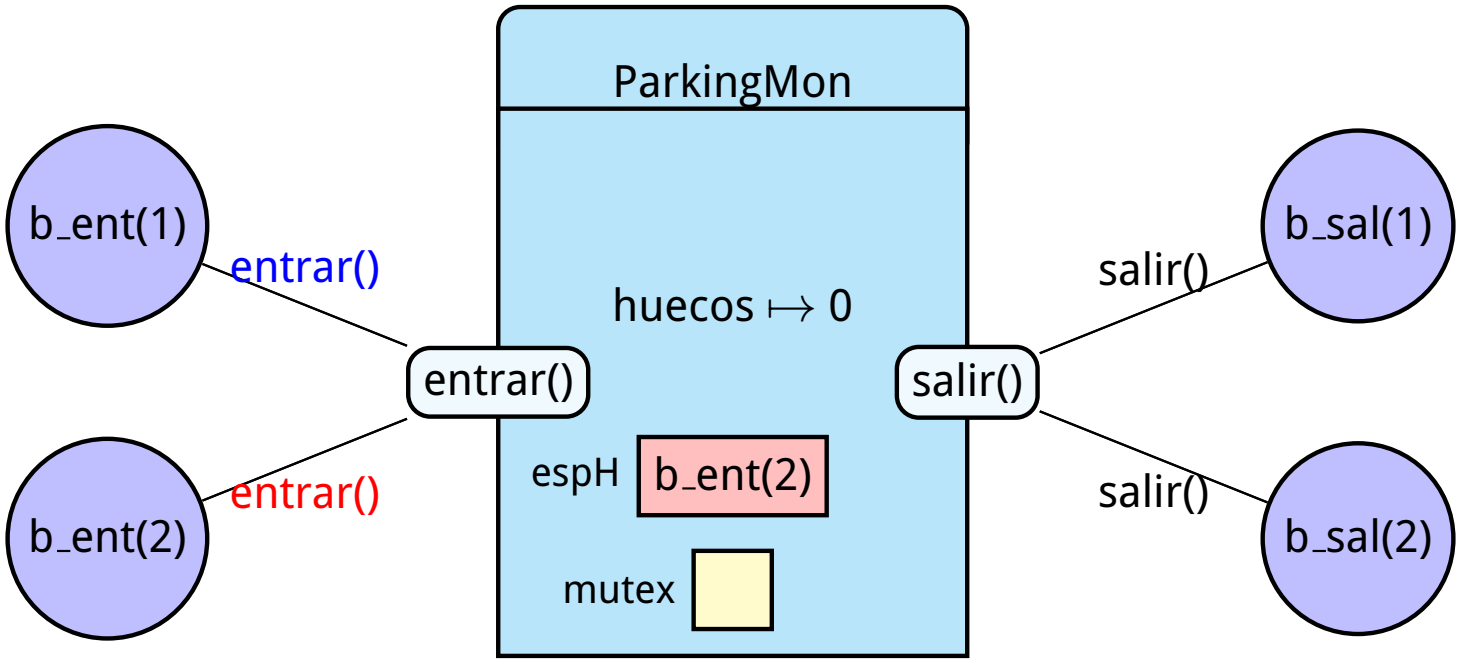
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

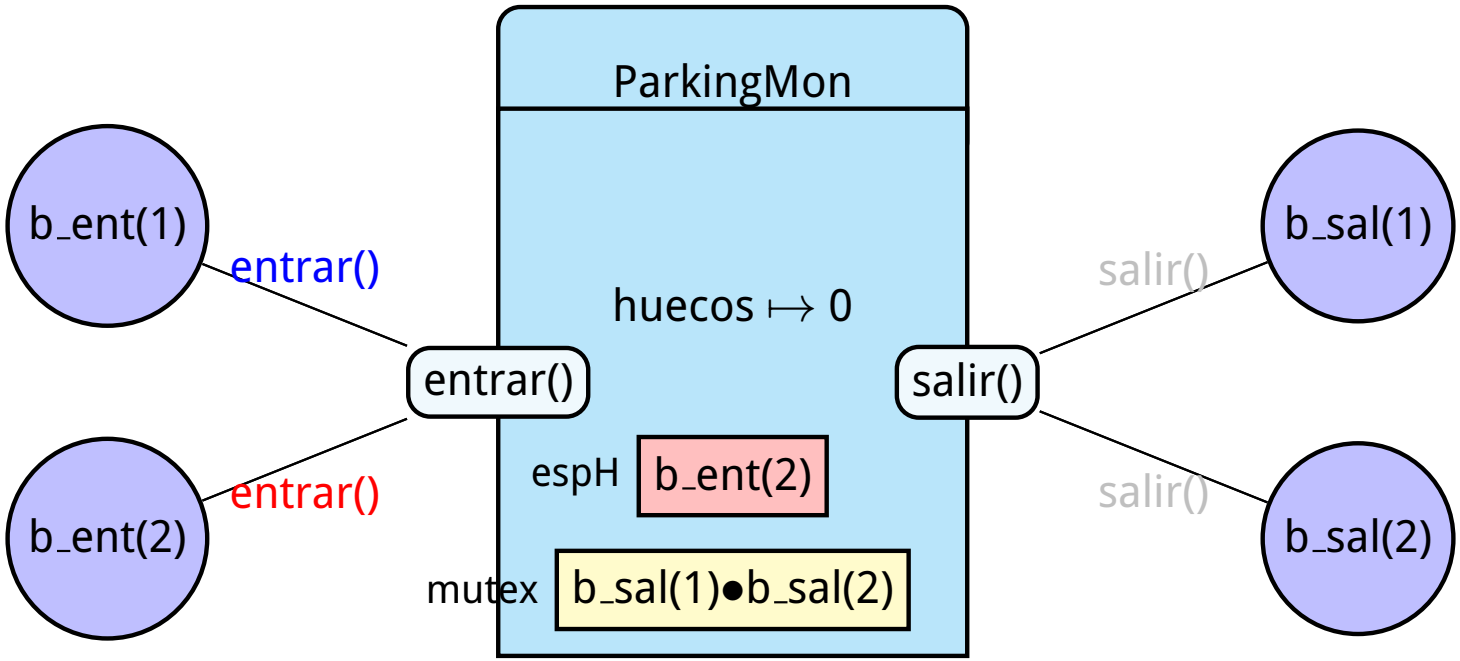
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

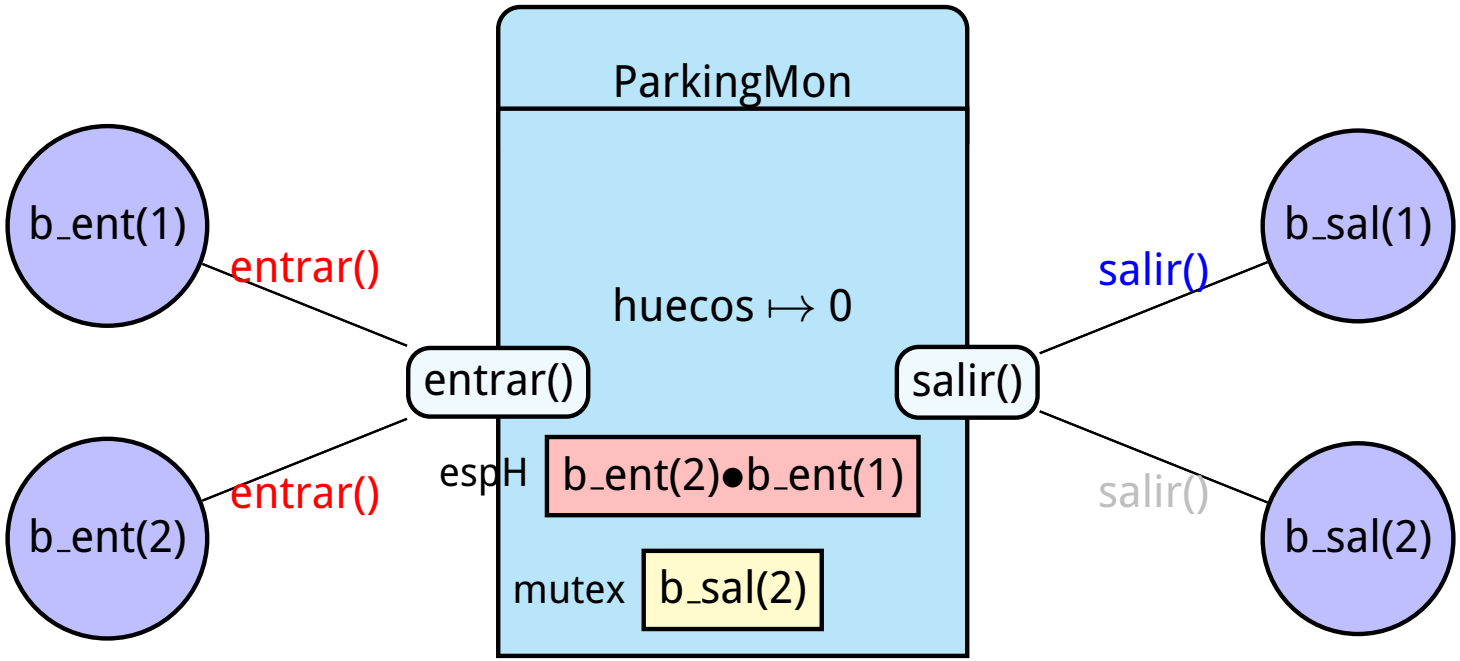
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

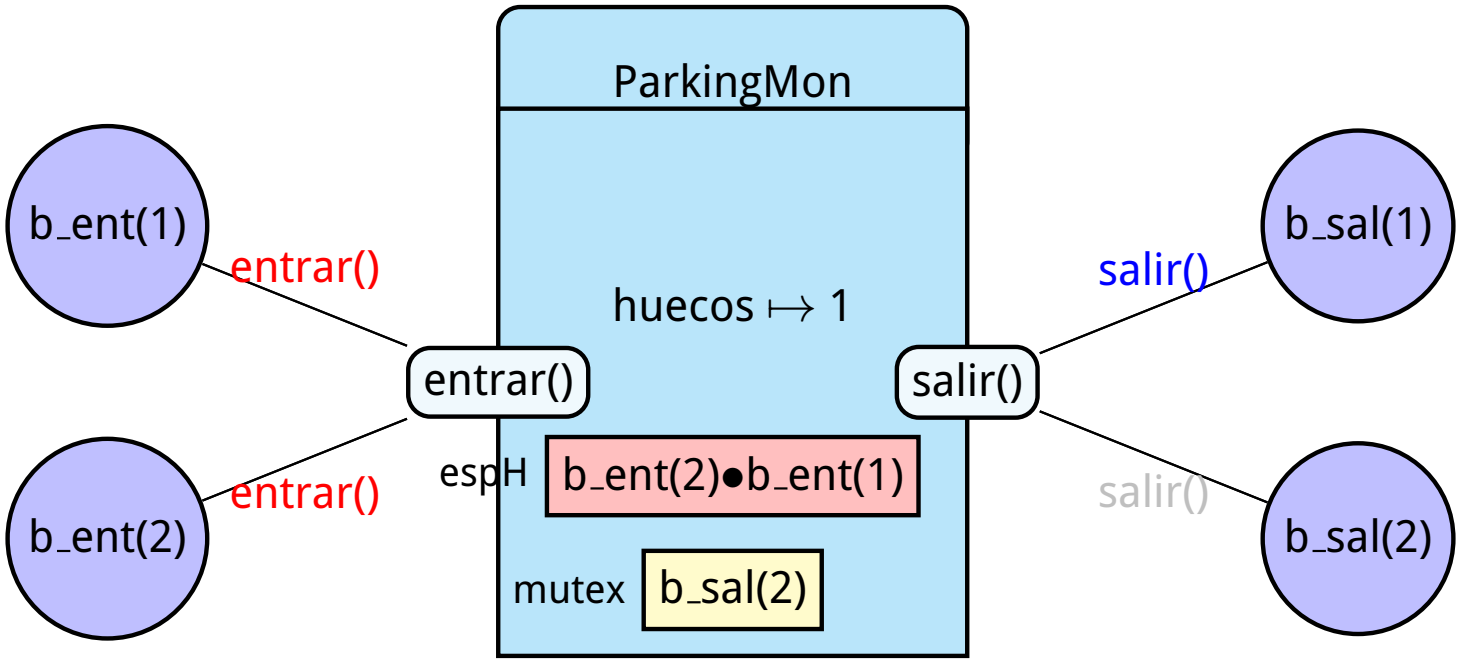
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

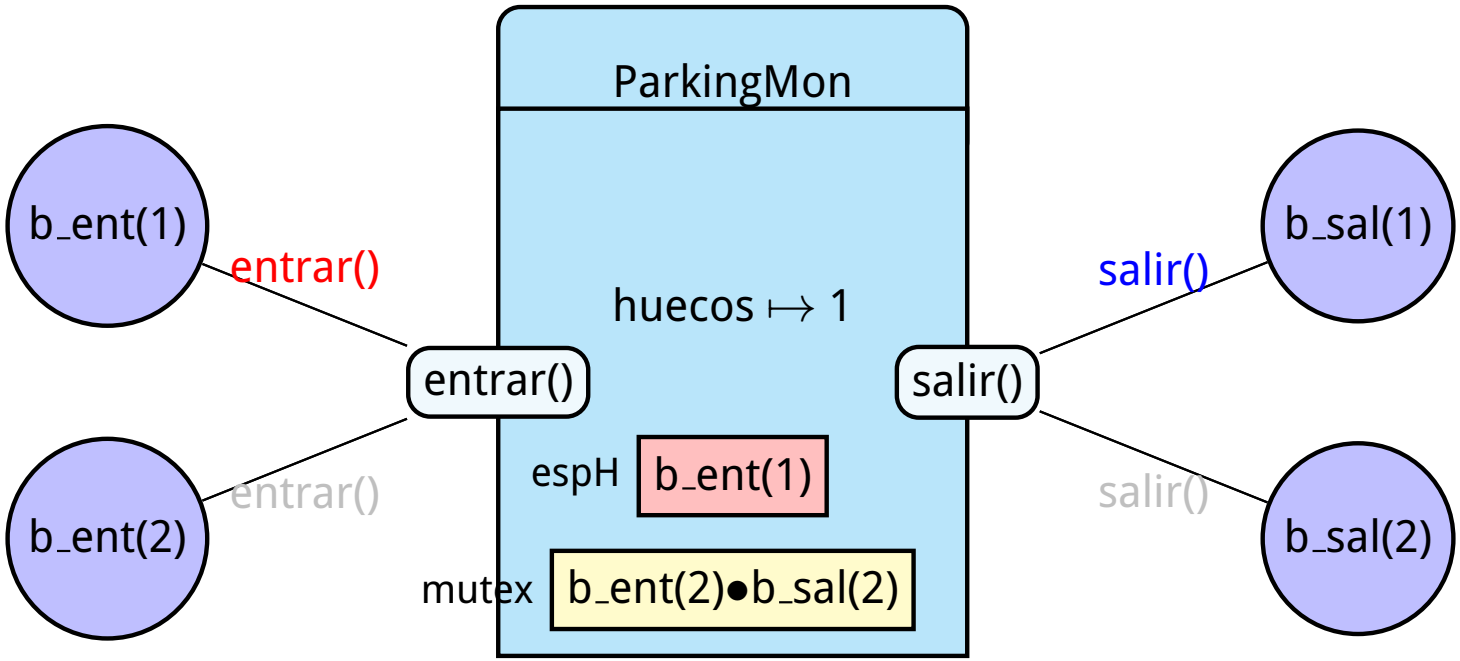
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

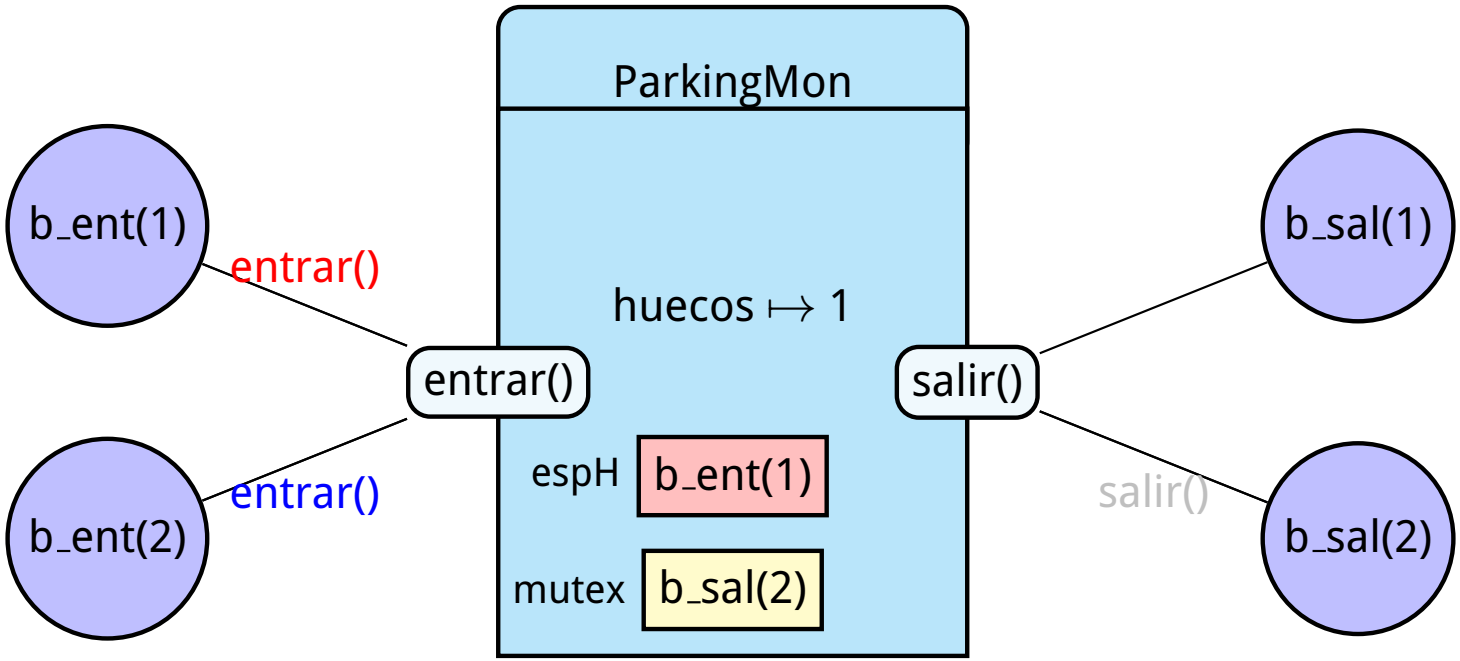
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

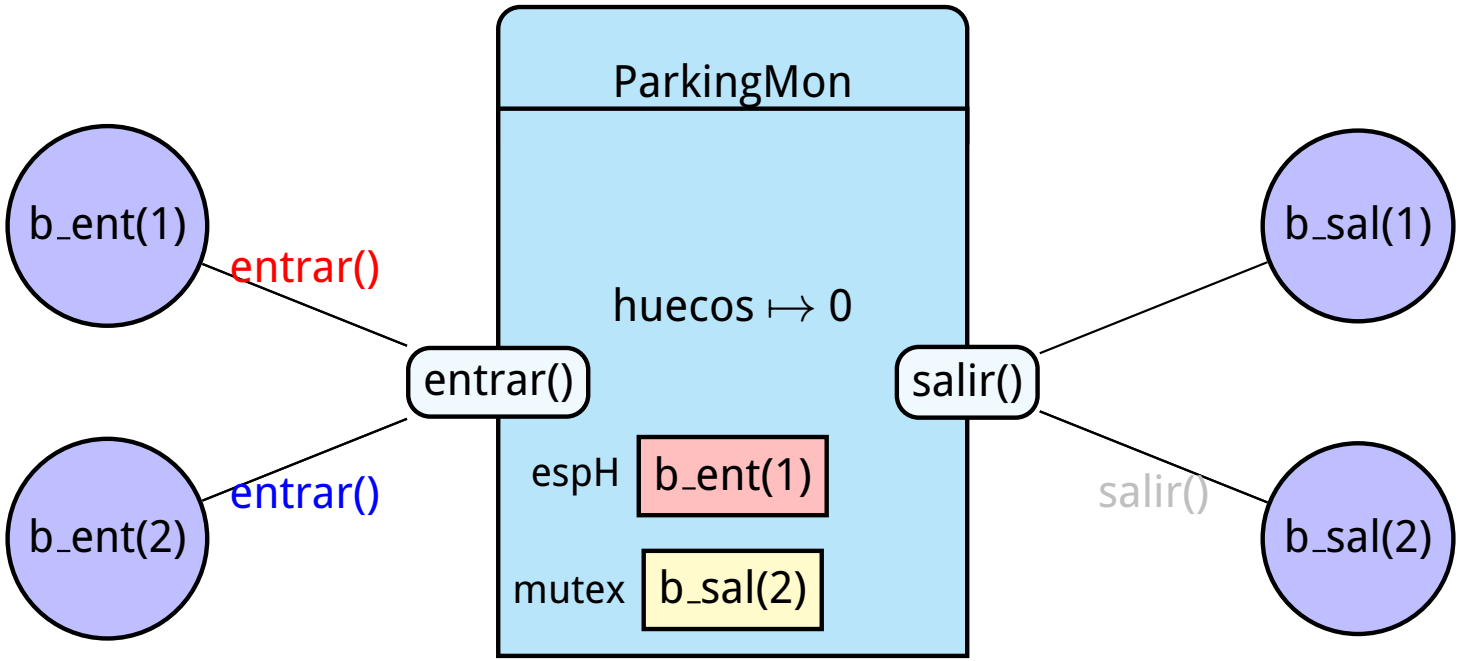
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

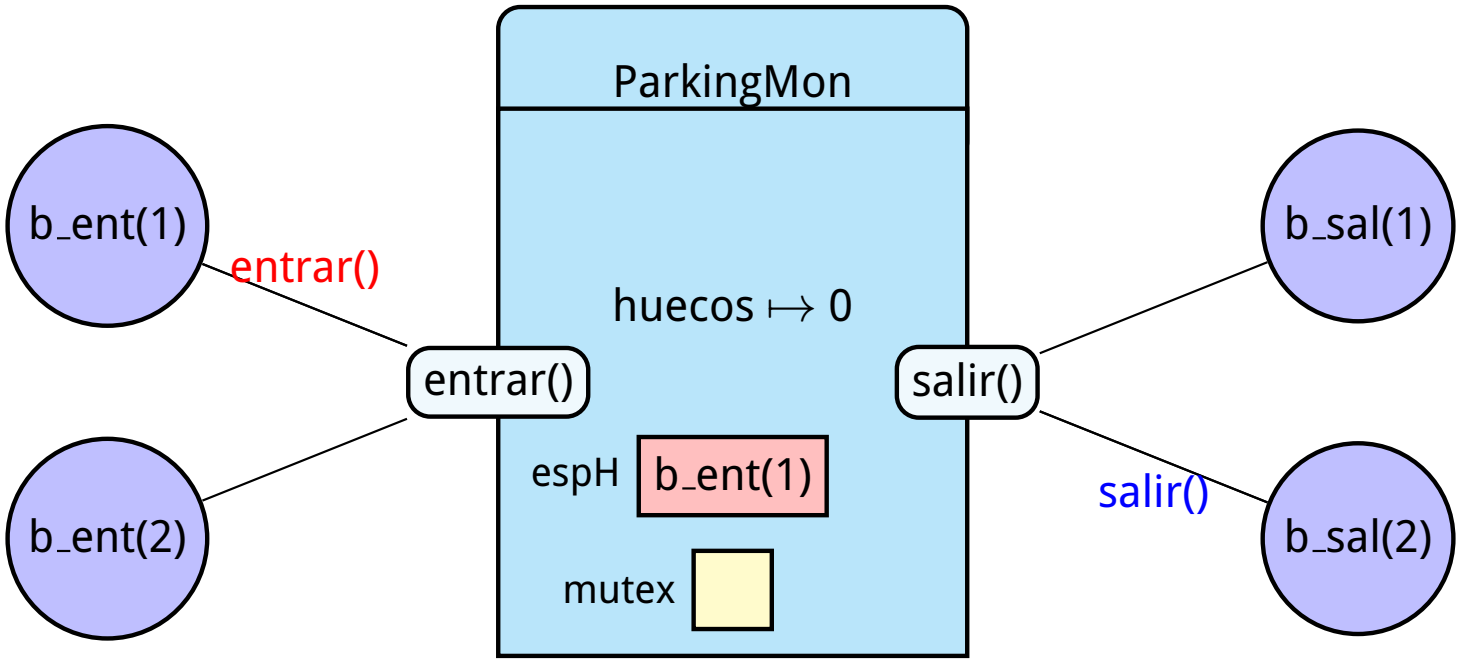
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

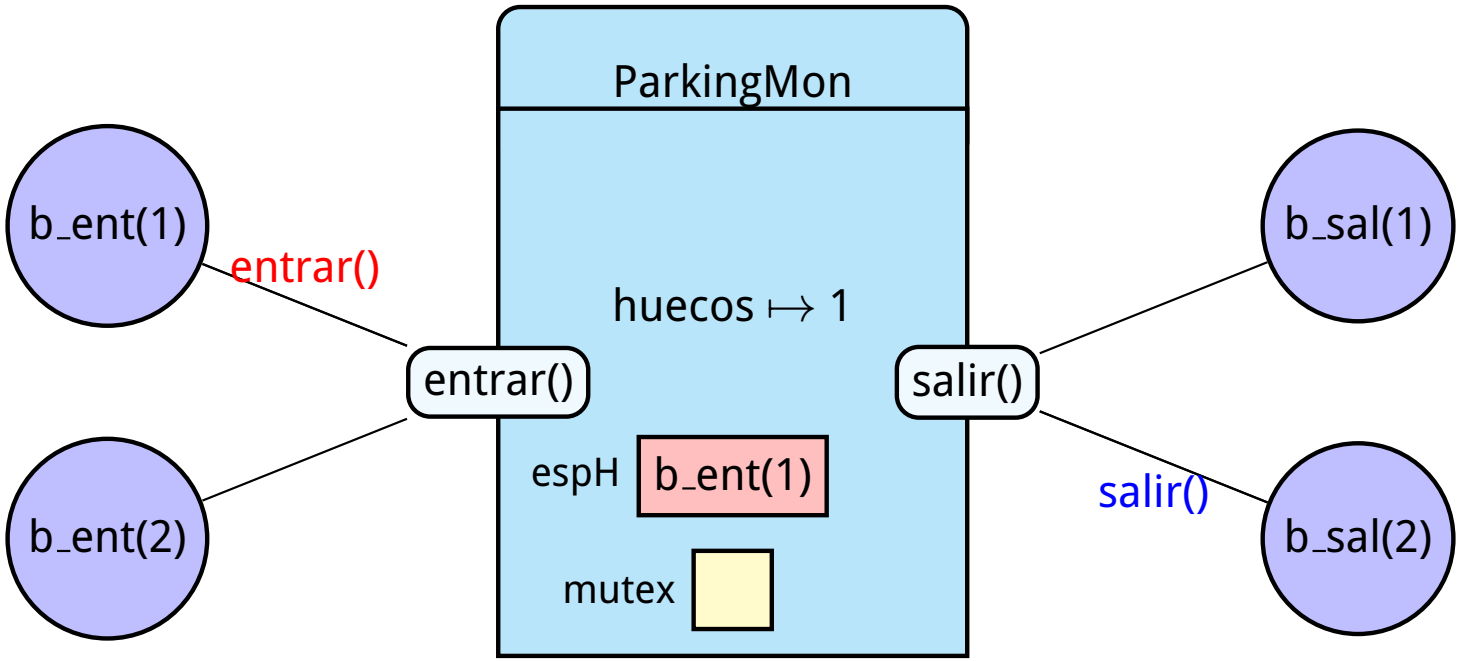
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

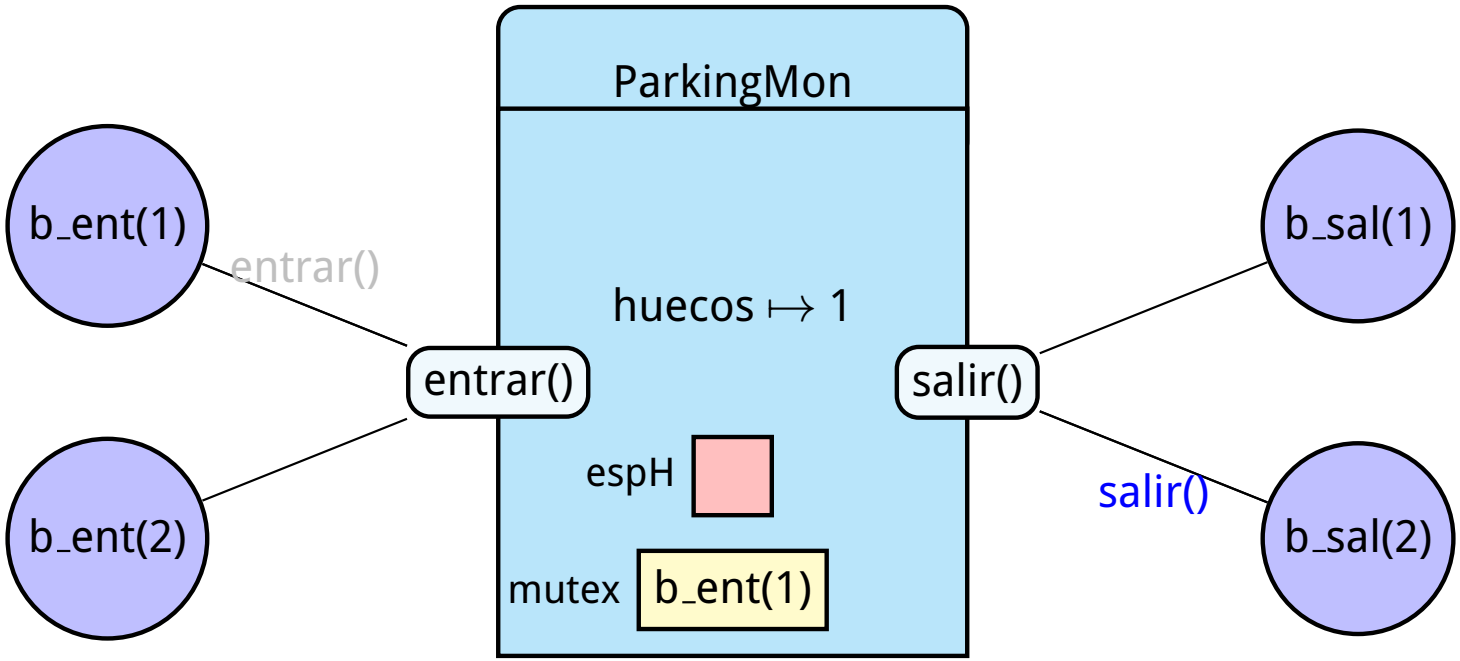
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

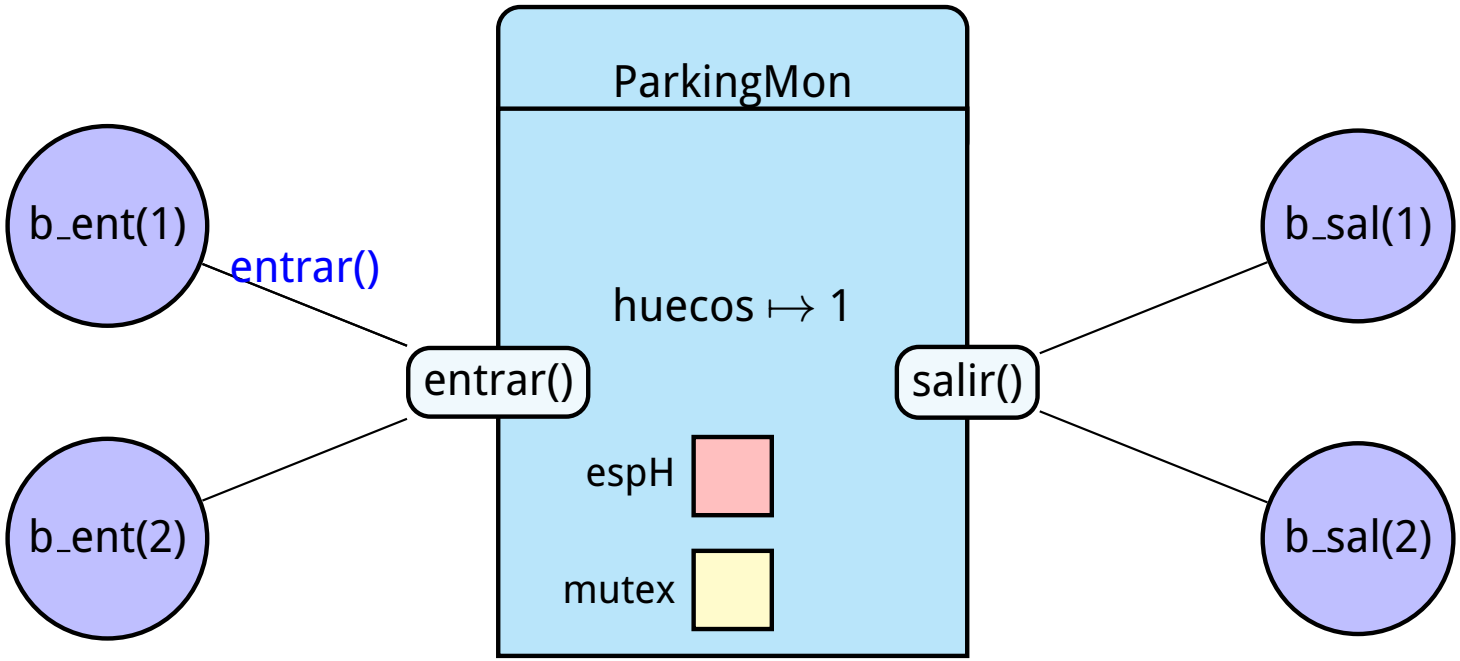
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

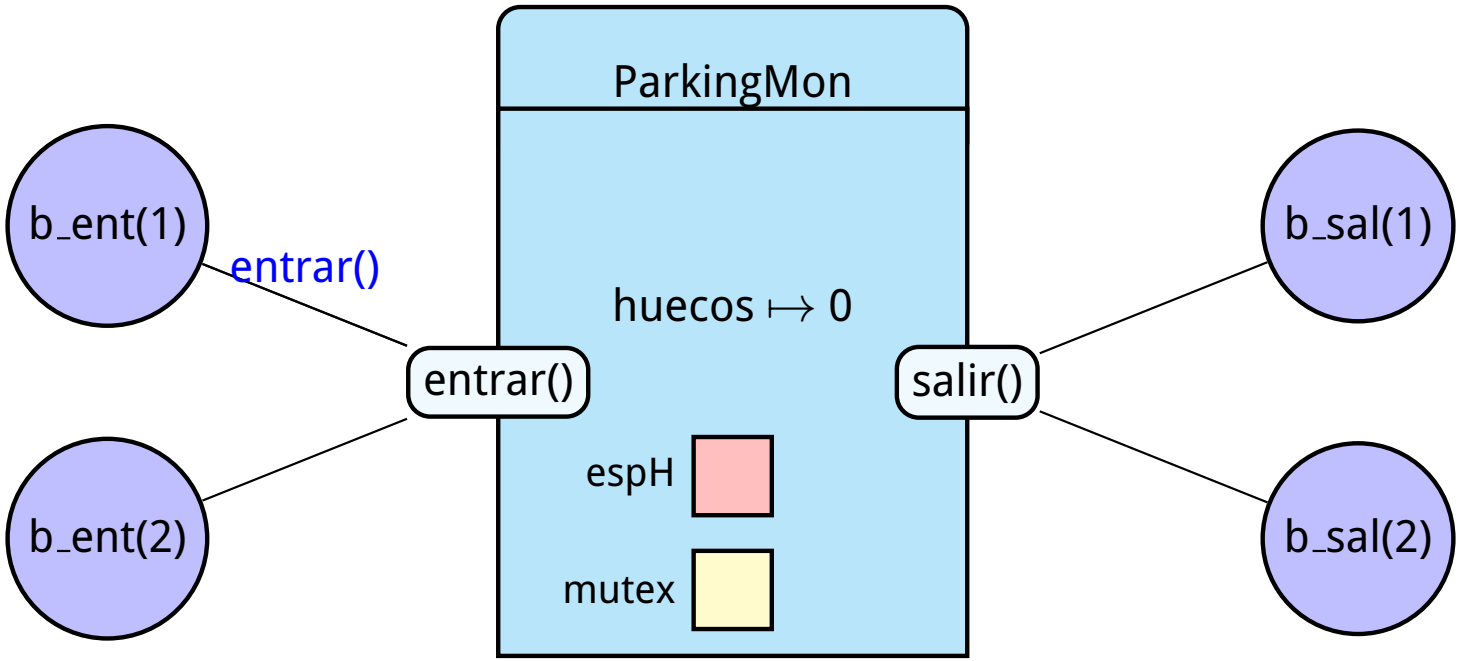
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

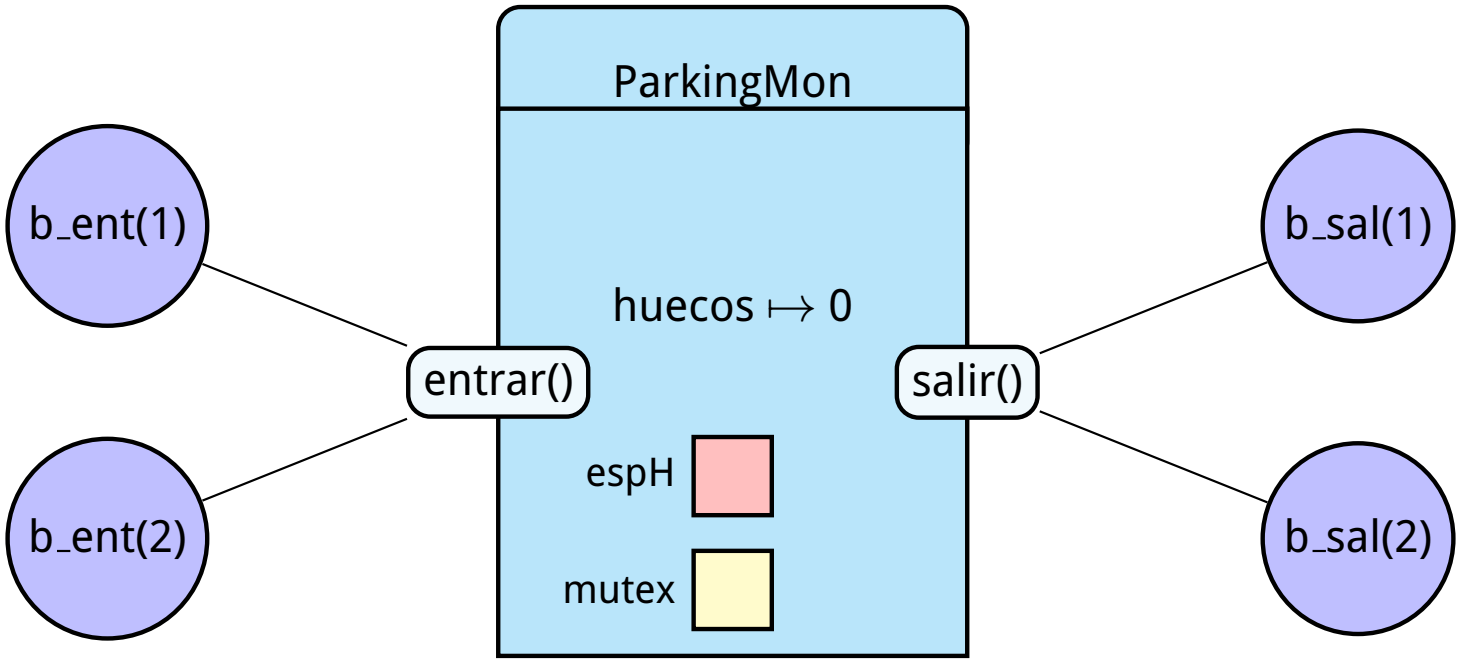
ojo al trasiego entre colas



POLITÉCNICA

modelo de ejecución

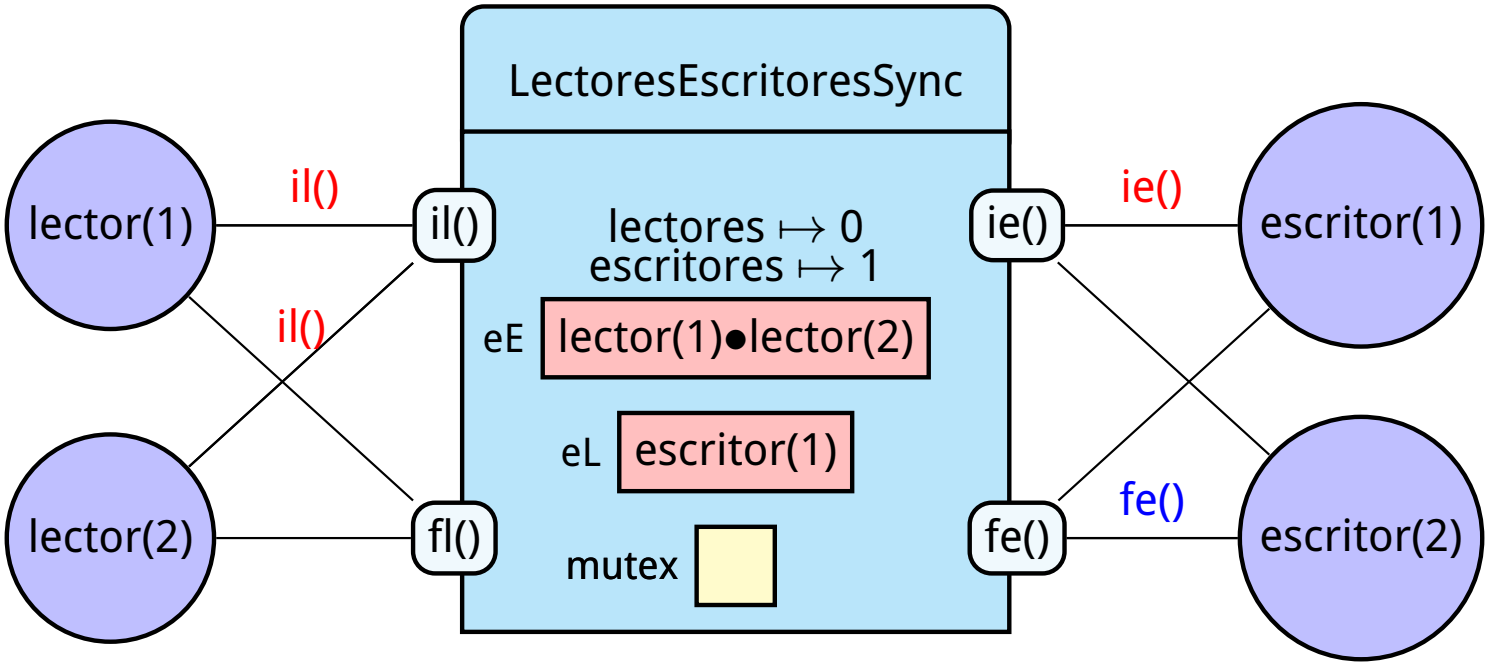
ojo al trasiego entre colas



POLITÉCNICA

lectores/escritores con monitores

las *condition queues* permiten diferenciar CPREs



POLITÉCNICA

en próximos vídeos...

- Estudiaremos un patrón general para traducir especificaciones de recurso compartido a clases de objetos sincronizados mediante monitores y condition queues.
- Habrá que tratar el caso de múltiples condition queues y cómo decidir qué llamadas a signal () hacer en caso de que se pueda desbloquear a varios tipos de procesos.
- También habrá que tratar el caso – no considerado en estos ejemplos sencillos – de CPREs que dependen de parámetros de entrada de una acción. Este problema es el tema del ejercicio de las entregas E8 – E10.
- Explicaremos las diferencias entre nuestra implementación, la que viene con Java (*locks & conditions*) y otras propuestas.

POLITÉCNICA

en próximos vídeos...

- Estudiaremos un patrón general para traducir especificaciones de recurso compartido a clases de objetos sincronizados mediante monitores y condition queues.
- Habrá que tratar el caso de múltiples condition queues y cómo decidir qué llamadas a signal() hacer en caso de que se pueda desbloquear a varios tipos de procesos.
- También habrá que tratar el caso – no considerado en estos ejemplos sencillos – de CPREs que dependen de parámetros de entrada de una acción. Este problema es el tema del ejercicio de las entregas E8 – E10.
- Explicaremos las diferencias entre nuestra implementación, la que viene con Java (*locks & conditions*) y otras propuestas.

POLITÉCNICA

en próximos vídeos...

- Estudiaremos un patrón general para traducir especificaciones de recurso compartido a clases de objetos sincronizados mediante monitores y condition queues.
- Habrá que tratar el caso de múltiples condition queues y cómo decidir qué llamadas a signal () hacer en caso de que se pueda desbloquear a varios tipos de procesos.
- También habrá que tratar el caso – no considerado en estos ejemplos sencillos – de CPREs que dependen de parámetros de entrada de una acción. Este problema es el tema del ejercicio de las entregas E8 – E10.
- Explicaremos las diferencias entre nuestra implementación, la que viene con Java (*locks & conditions*) y otras propuestas.

POLITÉCNICA

en próximos vídeos...

- Estudiaremos un patrón general para traducir especificaciones de recurso compartido a clases de objetos sincronizados mediante monitores y condition queues.
- Habrá que tratar el caso de múltiples condition queues y cómo decidir qué llamadas a signal () hacer en caso de que se pueda desbloquear a varios tipos de procesos.
- También habrá que tratar el caso – no considerado en estos ejemplos sencillos – de CPREs que dependen de parámetros de entrada de una acción. Este problema es el tema del ejercicio de las entregas E8 – E10.
- Explicaremos las diferencias entre nuestra implementación, la que viene con Java (*locks & conditions*) y otras propuestas.

POLITÉCNICA