Sincronización: capítulo 1

En este apartado te presentamos cómo abordar los problemas de sincronización entre procesos que habitualmente nos pueden surgir cuando desarrollamos programas concurrentes. Para resolver la sincronización vamos a utilizar unas herramientas básicas: control de sección críticas y manejar una cola de espera de procesos.

La solución general consiste en unos patrones de código muy sencillos que puedes aplicar en cada escenario problemático (sección crítica o necesidad de bloqueo condicional). Para poder entender cómo surge esta solución general, te vamos a ir presentando un algoritmo muy básico sobre el que vamos a aplicar mejoras sucesivas hasta llegar a la solución definitiva.

Al final del documento te proponemos varios ejercicios para que practiques la técnica aquí descrita. *¡¡No dejes de hacerlos!!*

Un problema básico: esperar por un evento

Uno de los problemas más básicos de sincronización es la espera por un evento. Por ejemplo, esperar a que un proceso finalice un trabajo. Esto lo podemos resolver con una variable booleana, compartida por el proceso en el que ocurre el evento y por los que necesiten esperar por él:

```
// variable compartida
bool eventoSucedio = false;
```

`// donde ocurre el evento`
`eventoSucedio = true;`
`...`
	`// donde hay que esperar por el evento`
	`while ( ! eventoSucedio ) {}`
	`...`

Otro problema: camareros y cocineros

Para introducirte en problemas más complejos de sincronización, usaremos el sistema que hemos visto en clase. En este sistema tenemos dos clases de procesos: camareros y cocineros. Cada cocinero se dedica a preparar platos, que va colocando en una bandeja. Cada camarero espera a que haya un plato sobre la bandeja, lo retira y se lo sirve a algún cliente. Esto lo hacen una y otra vez. Lo que queremos resolver es que un camarero se bloquee mientras no haya platos disponibles en la bandeja; y que un cocinero se bloquee si cuando quiere colocar un plato en la bandeja se la encuentra llena.

Este es el algoritmo básico del sistema, pendiente de resolver los problemas de sincronización.

```
class Bandeja { … }
Bandeja bandeja = …;
```

```
void hiloCocinero() {
  while (true) {
    cocinarUnPlato();
    … bloquear si bandeja llena
    bandeja.colocarPlato();
    fregarPlatos();
  }
}
```

```
void hiloCamarero() {
  while (true) {
    servirMesa();
    … bloquear si bandeja vacía
    bandeja.retirarPlato();
    servirPlato();
  }
}
```

Primer paso: usar una variable de estado

Tenemos que implementar un algoritmo para bloquear a los procesos cuando sea necesario. Tal y como vimos en clase, esta coordinación la podemos realizar con una variable compartida que nos indique en cada momento cuántos platos hay en la bandeja. Llamémosla **platosEnBandeja**. Este tipo de objetos se suelen llamar *variables de estado.*
Para bloquearse, el camarero puede ejecutar un bucle de espera mientras *platosEnBandeja==0*. Por su parte, el cocinero puede mantenerse en espera continua si *platosEnBandeja==MAX*, siendo MAX la capacidad máxima de la bandeja de los platos.
El algoritmo quedaría así:

```
class Bandeja { … }
Bandeja bandeja = …;
int platosEnBandeja = 0;
```

```
void hiloCocinero() {
  while (true) {
    cocinarUnPlato();
    while (platosEnBandeja==MAX) {}
    platosEnBandeja++;
    bandeja.colocarPlato();
    fregarPlatos();
  }
}
```

```
void hiloCamarero() {
  while (true) {
    servirMesa();
    while (platosEnBandeja==0) {}
    platosEnBandeja--;
    bandeja.retirarPlato();
    servirPlato();
  }
}
```

Problemas de este algoritmo

El principal problema de este algoritmo es que está lleno de riesgos ocasionados por el acceso concurrente a **platosEnBandeja**. Si un cocinero y un camarero modifican simultáneamente esta variable se puede corromper su valor, tal y como hemos visto en clase. Además, si **platosEnBandeja** vale 1 (hay un plato en la bandeja) y dos camareros llegan simultáneamente al bucle *while*, podrán decidir al mismo tiempo tomar el único plato disponible, lo cual viola el comportamiento que queremos para el sistema.

Otra cuestión es que estamos actualizando el valor de **platosEnBandeja** antes de que realmente hayamos colocado o extraído el plato de la bandeja. Un camarero puede observar que **platosEnBandeja** vale 1 cuando aún el cocinero no ha terminado de ejecutar **bandeja.colocarPlato()**. La actualización de la variable de estado debería ocurrir *después* de completar la actualización real de la bandeja.

Además de lo anterior, es posible que la implementación de las operaciones de la clase **Bandeja** no permita la ejecución concurrente de los métodos **colocarPlato**() y **retirarPlato**(). Por tanto estas dos operaciones deberían también estar controladas.

Controlar las secciones críticas

Para resolver con eficacia los problemas del primer algoritmo necesitamos controlar el acceso a las secciones críticas. Con este fin, supongamos que la API de nuestro SO nos ofrece sendas operaciones para entrar y salir de sección crítica: **ENTRARSC**() y **SALIRSC**(). En su interior estas operaciones implementan alguna de las técnicas que hemos visto en clase.

Haciendo uso de estas dos operaciones podríamos llegar a un algoritmo similar a este:

```
class Bandeja { … }
Bandeja bandeja = …;
int platosEnBandeja = 0;
```

```
void hiloCocinero() {
  while (true) {
    cocinarUnPlato();
    ENTRARSC();
    while (platosEnBandeja==MAX) {}
    bandeja.colocarPlato();
    platosEnBandeja++;
    SALIRSC();
    fregarPlatos();
  }
}
```

```
void hiloCamarero() {
  while (true) {
    servirMesa();
    ENTRARSC();
    while (platosEnBandeja==0) {}
    bandeja.retirarPlato();
    platosEnBandeja--;
    SALIRSC();
    servirPlato();
  }
}
```

Con este algoritmo se está protegiendo el acceso a **platosEnBandeja**, que se realiza en exclusión mutua.

Pero... tal y como ha quedado el algoritmo, surge un nuevo problema igual o más grave que el que hemos resuelto. Si un camarero se queda bloqueado en el bucle **while** porque no hay platos, impedirá a cualquier cocinero colocar un nuevo plato: el camarero bloqueado estará reteniendo el derecho de uso de la sección crítica. Con eso se llegará a una situación de bloqueo permanente e irreversible. El camarero espera por los cocineros (a que pongan platos nuevos), y los cocineros esperan por el camarero (a que desbloquee la sección crítica). Esta *espera circular* se conoce como **interbloqueo** (*deadlock*).

Evitar el interbloqueo

Para evitar el interbloqueo del algoritmo anterior tenemos que lograr que no ocurra el estado de *retención y espera*: un proceso bloqueado está reteniendo recursos que otros necesitan. Esta retención y espera la podemos romper haciendo que el camarero (o el cocinero) abandone la sección crítica en cuanto vea que se tiene que bloquear. Periódicamente el camarero volverá a mirar la bandeja (para ello tiene que ganar el acceso a la sección crítica). Esta misma técnica la podemos aplicar en el lado del cocinero.

Veamos el algoritmo transformado con esta técnica:

```
class Bandeja { … }
Bandeja bandeja = …;
int platosEnBandeja = 0;
```

```
void hiloCocinero() {
  while (true) {
    cocinarUnPlato();
    ENTRARSC();
    while (platosEnBandeja==MAX) {
      SALIRSC();
      ...esperar un poco…
      ENTRARSC();
    }
    bandeja.colocarPlato();
    platosEnBandeja++;
    SALIRSC();
    fregarPlatos();
  }
}
```

```
void hiloCamarero() {
  while (true) {
    servirMesa();
    ENTRARSC();
    while (platosEnBandeja==0) {
      SALIRSC();
      ...esperar un poco…
      ENTRARSC();
    }
    bandeja.retirarPlato();
    platosEnBandeja--;
    SALIRSC();
    servirPlato();
  }
}
```

Con este arreglo ya no se nos va a producir un interbloqueo, porque quien está en el bucle de espera no retiene el derecho de uso de la sección crítica. Podemos dar el algoritmo por bueno, aunque también podemos aspirar a mejorar su rendimiento. Es lo que haremos en el siguiente apartado.

Mejora del rendimiento: cola de espera

Desde la primera versión del algoritmo tenemos un problema no resuelto: la pérdida de rendimiento ocasionada por la espera activa (*busy waiting*) de los bucles **while**. Para zanjar de forma definitiva esta cuestión, lo que deberíamos hacer es DORMIR al proceso que veamos que está en una situación de bloqueo: que salga fuera de la cola de preparados y se mantenga en una cola de espera, de la cual solamente podrá salir si otro proceso lo despierta y lo coloca de nuevo en la cola de preparados.

Si conseguimos implementar este mecanismo, se acaban las esperas activas: un proceso que está en una situación de bloqueo no consume CPU; en vez de eso, permanece inactivo en una cola, sin malgastar recursos.

La buena noticia es que no es complicado implementar estas operaciones de «bloqueo» y «desbloqueo» dentro del núcleo de un SO. De hecho, cualquier SO multiprogramado las utiliza internamente para resolver los bloqueos por operaciones de E/S. Solamente le tenemos que ofrecer a los usuarios en la API estas dos operaciones. Vamos a llamarlas así:

- **DORMIR()** → deja al proceso actual en un estado de bloqueo (en una cola aparte)
- **DESPERTAR()** → coloca en la cola de preparados a *todos* los procesos bloqueados

La operación DESPERTAR(), cuando desbloquea a un proceso, lo reanuda justo en la siguiente instrucción después de DORMIR().

Aprovechando estas operaciones, el algoritmo nos podría quedar finalmente así:

```
class Bandeja { … }
Bandeja bandeja = …;
int platosEnBandeja = 0;
```

```
void hiloCocinero() {
  while (true) {
    cocinarUnPlato();
    ENTRARSC();
    while (platosEnBandeja==MAX) {
      SALIRSC();
      DORMIR();
      ENTRARSC();
    }
    bandeja.colocarPlato();
    platosEnBandeja++;
    DESPERTAR();
    SALIRSC();
    fregarPlatos();
  }
}
```

```
void hiloCamarero() {
  while (true) {
    servirMesa();
    ENTRARSC();
    while (platosEnBandeja==0) {
      SALIRSC();
      DORMIR();
      ENTRARSC();
    }
    bandeja.retirarPlato();
    platosEnBandeja--;
    DESPERTAR();
    SALIRSC();
    servirPlato();
  }
}
```

La única precaución que debe tener en cuenta el programador es que debe insertar unas operaciones de DESPERTAR() cuando varía el estado del sistema; en nuestro caso, cuando cambia el valor de *platosEnBandeja*. Si no añadimos estas operaciones, los procesos dormidos nunca podrán reanudar su ejecución.

Capítulo 1: resumen final

La discusión anterior sirve para extraer unas técnicas básicas para solventar los problemas de sincronización que nos pueden aparecer en un programa concurrente:

- espera por un evento
- acceso a un objeto compartido por varios procesos → sección crítica
- espera/bloqueo si no se cumple una condición

Para resolver estos escenarios, supondremos que el sistema operativo nos ofrece cuatro operaciones:

ENTRARSC()	El proceso actual entra en la sección crítica. Si el recurso está ocupado, el proceso se bloquea hasta que quede libre.
SALIRSC()	El proceso actual sale de la sección crítica. Si hay otros procesos esperando por ella, alguno de ellos se desbloquea.
DORMIR()	El proceso actual se bloquea y pasa a una cola de espera.
DESPERTAR()	Se desbloquean todos los procesos que estaban en la cola de espera y vuelven a competir por el procesador.

Si hay que esperar a que ocurra un evento, el patrón general podría ser:

```
// variable compartida
bool eventoSucedio = false;
```

`// donde ocurre el evento`
`eventoSucedio = true`
`...`
	`// donde hay que esperar por el evento`
	`while ( ! eventoSucedio ) {}`
	`...`

Para el acceso a un objeto compartido (sin esperas condicionales) nos basta con este código:

```
ENTRARSC();
… acciones sobre el objeto compartido …
SALIRSC();
```

Cuando nos encontremos con un problema de bloqueo condicionado a que se cumpla alguna expresión lógica, el patrón general del código sería así:

```
ENTRARSC();
while ( condición de bloqueo ) {
  SALIRSC();
  DORMIR();
  ENTRARSC();
}
...acciones sobre las variables de estado…
… acciones sobre el objeto compartido …
DESPERTAR();
SALIRSC();
```

Capítulo 1: ejercicios

1 En una tienda de pájaros están teniendo problemas para tener a todos sus canarios felices. Los canarios comparten una jaula en la que hay un plato con alpiste y un columpio para hacer ejercicio. Todos los canarios quieren inicialmente comer del plato y después columpiarse. Pero se encuentran con el inconveniente de que sólo tres de ellos pueden comer del plato al mismo tiempo y sólo uno puede columpiarse.

Escribe el código de un proceso que ejecuta cada uno de los canarios de forma que esté sincronizado con todos los demás.

2 En una fábrica se tienen dos procesos que modelan una planta embotelladora de bebidas, y que trabajan en paralelo:

- Un proceso «Embotellador» se encarga de preparar botellas de un litro.
- Otro proceso «Empaquetador» se encarga de empaquetar y reponer las cajas donde se van colocando las botellas.

Cada vez que el embotellador prepara una botella, ésta se coloca en una caja, que tiene una capacidad de 10 litros. Si al colocar la botella la caja queda llena, se envía una señal al empaquetador, que toma la caja, la sella y la guarda en un almacén. El empaquetador deposita una nueva caja de 10 litros, totalmente vacía. Mientras el empaquetador está haciendo su labor, el embotellador no puede colocar sus botellas, ya que en esos momentos no hay una caja disponible.

Escribe el algoritmo de estos dos procesos, solucionando los problemas de sincronización.

3 El algoritmo que hemos propuesto para solucionar la espera por un evento (ver página 1) utiliza un bucle de espera activa. Aprovecha las operaciones DORMIR() y DESPERTAR() para modificar el algoritmo y que la espera no consuma tiempo de CPU.

4 Se trata de resolver el problema de la pajarería (problema 1) haciendo una jaula un poco más entretenida. Vamos a permitir que se puedan columpiar dos canarios, siempre y cuando uno sea macho y otro hembra. **Nota**: No respondemos del uso que le den los canarios al columpio ☺

Sincronización: capítulo 2

En este segundo capítulo te presentamos las herramientas que vamos a utilizar en la parte práctica de la asignatura para implementar la sincronización entre hilos: los **cerrojos** (también llamados **mutex**) y las **variables condición**. Estas herramientas son tipos abstractos de datos que encapsulan los fragmentos de código que hemos presentado en el capítulo 1 para resolver problemas básicos de sincronización.

Cerrojos/mutex

Un cerrojo o *mutex* es un tipo abstracto de datos que se emplea para regular el acceso en exclusiva a un recurso compartido. La palabra «mutex» proviene del inglés *mutual exclusion* (exclusión mutua).

Su interfaz en Java vendría a ser así:

```
public class Mutex {
      public void lock();     // adquiere el mutex
      public void unlock();   // libera el mutex
}
```

Un cerrojo o mutex puede estar en dos estados: libre o bloqueado. Inicialmente se encuentra libre. Las operaciones `lock()` y `unlock()` se comportan de forma muy similar a las operaciones ENTRARSC y SALIRSC del Capítulo 1:

operación	funcionamiento
`lock()`	Si el cerrojo está libre, pasa al estado 'ocupado'. El proceso que ha hecho la llamada se convierte en *propietario* del cerrojo. Si el cerrojo está bloqueado, el proceso que hace la llamada se bloquea y espera hasta que pueda adquirir la propiedad del cerrojo.
`unlock()`	Libera el cerrojo. Esta operación solo la debería poder invocar el actual propietario del cerrojo. Si existen procesos esperando en el cerrojo, el sistema escoge a alguno de ellos y lo desbloquea. El proceso desbloqueado pasa a ser el nuevo propietario del cerrojo. Si no hay procesos esperando, el cerrojo se libera sin más.

Variables condición

Una variable condición abstrae una cola de espera similar a la que hemos implementado en el Capítulo 1 mediante las operaciones DORMIR y DESPERTAR. Su definición en Java es la siguiente:

```
public class Condition {
      public Condition (Mutex m);
      public void wait();
      public void signal();
      public void broadcast();
}
```

Una variable condición siempre estará vinculada a un cerrojo (el parámetro *m* del constructor de la clase). Las operaciones funcionan de la siguiente manera:

operación	funcionamiento
`wait()`	Libera el cerrojo asociado a la variable condición y bloquea al proceso. Cuando el proceso se desbloquee, recupera de nuevo el cerrojo.
`signal()`	Desbloquea a exactamente uno de los procesos que hay en la cola de espera de la variable condición. Si no hay procesos esperando, esta operación no hace nada.
`broadcast()`	Desbloquea a todos los procesos que hay en la cola de espera de la variable condición. Si no hay procesos esperando, esta operación no hace nada.

Ejemplo de uso: sección crítica

El acceso a una sección crítica se resuelve de forma trivial mediante mutex:

```
// Variable compartida por todos los procesos afectados
Mutex mutex = new Mutex();
```

```
// código de la sección crítica de un proceso
mutex.lock();
  … instrucciones de la sección crítica …
mutex.unlock();
```

El algoritmo emplea un cerrojo compartido por todos los procesos. Cualquier proceso que intente acceder a una sección crítica tratará de adquirir el cerrojo compartido. Si el cerrojo está libre, el proceso avanzará sin bloquearse. Mientras el proceso se encuentre en su sección crítica, cualquier otro proceso que intente entrar se encontrará con el cerrojo bloqueado y la operación **lock()** lo dejará bloqueado. Cuando el proceso que está disfrutando de la sección crítica la abandone, ejecutará una operación **unlock()** que liberará el cerrojo. Si hay uno o varios procesos bloqueados en la operación **lock()**, alguno de ellos -y solamente uno- conseguirá desbloquearse.

Ejemplo de uso: esperar por un evento

Si hay que esperar a que ocurra un evento, esta podría ser el esquema si utilizamos mutex y variables condición.

```
// variable compartida
bool eventoSucedio = false;

// objetos para la sincronización
Mutex mutex = new Mutex();
Condition cola = new Condition(mutex);
```

`// donde ocurre el evento`
`mutex.lock();`
`  eventoSucedio = true;`
`  cola.broadcast();`
`mutex.unlock();`
	`// donde hay que esperar por el evento`
	`mutex.lock();`
	`  if (!eventoSucedio) {`
	`  cola.wait();`
	`  }`
	`mutex.unlock();`

La idea es utilizar una variable condición para sincronizar a los procesos. Si un proceso observa que el evento aún no ha sucedido, se bloquea con **wait()** en la variable condición. Cuando un proceso quiere notificar que ha ocurrido el evento, invoca a **broadcast()** para desbloquear a todos los procesos que pudieran estar esperando por el evento.

Ejemplo de uso: espera condicional

Para los problemas de esperas condicionales, podemos aplicar el mismo algoritmo propuesto en el Capítulo 1, adaptado a la interfaz de cerrojos y variables condición:

```
// objetos para la sincronización
Mutex mutex = new Mutex();
Condition cola = new Condition(mutex);
```

```
mutex.lock();
while ( condición de bloqueo ) {
  cond.wait();
}
… acciones sobre las variables de estado…
… acciones sobre el objeto compartido …
cond.broadcast();
mutex.unlock();
```

Como puede verse, nos queda un código bastante compacto, sobre todo porque la operación **wait()** de la variable condición resuelve lo que en el Capítulo 1 requería una secuencia de tres operaciones.

En caso de que queramos despertar solamente a un proceso de la cola, usaríamos **signal()** en lugar de **broadcast()**. Si nuestro algoritmo funciona igual con signal() que con broadcast(), siempre será preferible el **signal()**, ya que evita estar desbloqueando innecesariamente a un conjunto de procesos que entrarán a competir al mismo tiempo por el cerrojo. Ante la duda de si usar una u otra, nos habremos de decantar por **broadcast()**.

El restaurante con mutex y variables condición

Veamos cómo se podría aplicar este esquema al problema de los cocineros y camareros:

```
class Bandeja { … }
Bandeja bandeja = …;
int platosEnBandeja = 0;
Mutex mutex = new Mutex();
Condition cocinero = new Condition(mutex);
Condition camarero = new Condition(mutex);
```

```
void hiloCocinero() {
  while (true) {
    cocinarUnPlato();
    mutex.lock();
    while (platosEnBandeja==MAX) {
      cocinero.wait();
    }
    bandeja.colocarPlato();
    platosEnBandeja++;
    camarero.signal();
    mutex.unlock();
    fregarPlatos();
  }
}
```

```
void hiloCamarero() {
  while (true) {
    servirMesa();
    mutex.lock();
    while (platosEnBandeja==0) {
      camarero.wait();
    }
    bandeja.retirarPlato();
    platosEnBandeja--;
    cocinero.signal();
    mutex.unlock();
    servirPlato();
  }
}
```

En este ejemplo hemos usado dos variables condición distintas para los cocineros y los camareros. Usamos **signal()** en lugar de **broadcast()** porque sabemos que si colocamos o retiramos un plato, solamente se tendría que despertar un proceso de la otra clase.

Interfaz de Java

Los mutex/cerrojos y variables condición se utilizan con sintaxis diversas en muchos lenguajes de programación y bibliotecas. En el lenguaje Java, todo objeto derivado de la clase Object tiene asociado un mutex. Un método de una clase puede tener la propiedad **synchronized**, que significa que ese método ha de ejecutarse en exclusión mutua. El compilador inserta automáticamente código para adquirir y liberar el cerrojo del objeto cuando se ejecuta el método. Además, la clase Object ofrece unos métodos de sincronización llamados **wait()**, **notify()** y **notifyAll()** que son prácticamente iguales a las operaciones **wait()**, **signal()** y **broadcast()** que hemos descrito para las variables condición.

Interfaz de pthreads

En las prácticas de la asignatura usaremos lenguaje C y la biblioteca pthreads. Esta biblioteca ofrece una API para cerrojos y variables condición, que tienen un comportamiento idéntico al que hemos descrito, aunque adaptado a las características del lenguaje C:

Código de pthreads	Equivalente en este documento
`pthread_mutex_t mutex;` `pthread_mutex_init(&mutex,NULL);`	mutex = new Mutex()
`pthread_mutex_lock(&mutex);`	mutex.lock()
`pthread_mutex_unlock(&mutex);`	mutex.unlock()
`pthread_cond_t cond;` `pthread_cond_init(&cond,NULL);`	Cond = new Condition(), pero sin mutex asociado
`pthread_cond_wait(&cond,&mutex);`	cond.wait(), aunque el mutex hay que pasarlo como parámetro adicional
`pthread_cond_signal(&cond);`	cond.signal()
`pthread_cond_broadcast(&cond);`	cond.broadcast()

Capítulo 2: ejercicios

5 Resuelve con mutex y variables condición el problema de la pajarería (el ejercicio 1 y la variante del ejercicio 4).

6 Resuelve con mutex y variables condición el problema de la embotelladora (el ejercicio 2).

7 Supongamos tres procesos concurrentes P1, P2 y P3 con el código que se muestra a continuación:

```
Proceso P1:
while (true) {
  a;
  b;
}
```

```
Proceso P2:
while (true) {
  c;
  d;
}
```

```
Proceso P3:
while (true) {
  e;
  f;
}
```

Empleando cerrojos y variables condición, se pide introducir las modificaciones necesarias para que por cada ejecución de **«a»** o de **«e»** se permita ejecutar una iteración de **«d»**.

8 Tenemos una cuenta bancaria compartida, en la que hay procesos que retiran dinero o lo ingresan, mediante las funciones:

```
void ingresa ( float euros );
void retira ( float euros );
```

La cuenta tiene un saldo inicial de E euros, que se actualiza con las rutinas **ingresa()** y **retira()**. El saldo nunca puede quedar negativo: si un proceso intenta retirar más dinero del que hay actualmente, la rutina **retira()** dejará al proceso bloqueado hasta que haya saldo suficiente.

El acceso a la cuenta debe estar protegido: los accesos deben hacerse en exclusión mutua.

Se trata de implementar las operaciones **ingresa()** y **retira()** asegurando la exclusión mutua y el bloqueo condicional mediante mutex y variables condición.

9 Se ha inaugurado un nuevo parque en el barrio y debido a la gran aceptación se ha establecido un aforo máximo de usuarios (N) por razones de seguridad. Para ello se han instalado unos torniquetes a la entrada y a la salida que controlan en todo momento el número de personas en el parque. Cuando el aforo está completo, se debe esperar en la entrada hasta que alguien salga del parque. La afluencia de gente es tal que se forman largas colas de espera, por lo que se ha decidido crear una entrada preferente para los vecinos residentes en la zona, de forma que cuando el parque está lleno, tienen preferencia sobre los no residentes a la hora de acceder a las instalaciones.

Se pide realizar el código de entrada y salida del parque, de forma que cuando alguien abandona el parque, en caso de que existan personas esperando debido a que el cupo esté agotado, tendrán preferencia los residentes frente a los no residentes. La sincronización se deberá realizar con cerrojos y variables condición.

10 Un **semáforo** es una herramienta de sincronización clásica, propuesta por Edsger Dijkstra en 1965. Muchas API de concurrencia ofrecen semáforos como mecanismo de sincronización. Un semáforo es una variable entera que solo puede ser modificada mediante dos operaciones: **P()** y **V()**. La definición de estas operaciones es la siguiente:

operación	funcionamiento
`init(sem,N)`	Inicializa el valor de **sem** a **N**. N debe ser igual o mayor que cero. Esta operación debe ejecutarse al principio de la vida del semáforo, solamente una vez (es un constructor).
`P(sem)`	Mientras el valor de **sem** sea cero, deja al proceso bloqueado hasta que cambie el valor. Cuando el valor sea superior a cero, lo decrementa en uno.
`V(sem)`	Incrementa en uno el valor de **sem**. Si hubiera procesos bloqueados por haber ejecutado una operación P(), desbloquea a uno de ellos.

La API debe garantizar que estas operaciones se ejecutan de forma atómica: mientras un proceso está evaluando o modificando el semáforo, lo hace en exclusión mutua respecto a los demás.

a) Imagina que en lugar de mutex y variables condición tuvieras semáforos como mecanismo de sincronización. ¿Cómo resolverías el acceso a una sección crítica? ¿Y la espera por un evento?

b) Implementa un semáforo mediante mutex y variables condición.