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 la herramienta que vamos a utilizar en la parte práctica de la asignatura para implementar la sincronización entre hilos. Esta herramienta se llama **semáforo**. Las técnicas que hemos visto en el capítulo 1 son las mismas, pero con ciertas variaciones para adaptarlas a los semáforos, que serán las que usemos tanto en las prácticas como en los exámenes de teoría.

Semáforos: cómo funcionan

Los semáforos fueron propuestos por Edsger Dijkstra en 1965 como un mecanismo simple y a la vez potente de sincronización. Un semáforo es una variable entera que sólo se puede manipular mediante dos operaciones: una denominada «P» que lo decrementa en uno; y otra llamada «V» que lo incrementa en uno. Si al intentar realizar una operación «P» el semáforo tiene valor cero (o negativo), el proceso se queda bloqueado hasta que el semáforo adquiera un valor superior a cero. Aparte de «P» y «V», no se puede manipular el semáforo con ninguna otra operación.

Los nombres «P» y «V» provienen de palabras holandesas de la especificación original. En el mundo angloparlante se suelen nombrar como «wait» y «signal» respectivamente.

Estas son las especificaciones detalladas de las operaciones de un semáforo:

Nombre original	Nombre en inglés	Descripción algorítmica
P (sem)	**wait** (sem)	`while (sem<=0) {}` `sem = sem-1`
V (sem)	**signal** (sem)	`sem = sem+1`

Un semáforo debe inicializarse con un valor igual o superior a cero.

Para que un semáforo pueda servir de herramienta de sincronización, su implementación debe cumplir estrictamente esta propiedad: las operaciones deben ejecutarse siempre de forma **atómica**. En la operación «P», los pasos de comprobar el valor y luego decrementar se tienen que realizar en exclusión mutua respecto a cualquier otro acceso al semáforo; igualmente, el decremento realizado por la operación «V» debe ejecutarse en exclusión mutua.

Cumpliendo estas especificaciones, los semáforos permiten resolver problemas típicos de sincronización con apenas unas líneas de código. Es lo que veremos a continuación.

Sección crítica con semáforos

Si disponemos de semáforos, el problema de sección crítica se resuelve de forma trivial:

```
// Variable compartida por todos los procesos afectados
Semáforo mutex = 1;
```

```
// código de la sección crítica de un proceso
P(mutex);
  … instrucciones de la sección crítica …
V(mutex);
```

El algoritmo emplea un semáforo inicializado a 1, compartido por todos los procesos. Cualquier proceso que intente acceder a una sección crítica realizará una operación **P()** sobre el semáforo. Si el semáforo vale 1, pasará a valer 0 y el proceso avanzará sin bloquearse. Mientras el proceso se encuentre en su sección crítica, cualquier otro proceso que intente entrar se encontrará con el semáforo a 0 y la operación **P()** lo dejará bloqueado. Cuando el proceso que está disfrutando de la sección crítica la abandone, ejecutará una operación **V()** que restituirá el valor 1 en el semáforo. Si hay uno o varios procesos bloqueados en la operación **P()**, alguno de ellos -y solamente uno- conseguirá desbloquarse y decrementar el valor.

El nombre **mutex** que hemos elegido para el semáforo no es casual. Se suelen llamar así a estos semáforos que protegen secciones críticas. **«Mutex»** es una abreviatura del inglés *mutual exclusion*.

Esperar por un evento con semáforos

Si hay que esperar a que ocurra un evento, la solución general con semáforos es bastante simple:

```
// variable compartida
Semáforo eventoSucedio = 0;
```

`// donde ocurre el evento`
`V(eventoSucedio);`
`...`
	`// donde hay que esperar por el evento`
	`P(eventoSucedio);`
	`...`

Ojo, esta solución solamente funciona para un único proceso que espera por el evento. Si hubiera múltiples procesos a la escucha del evento, solamente uno de ellos se desbloquearía cuando ocurriera el evento: sería el primero de ellos que consiguiera decrementar el valor del semáforo, que pasaría a valer 0 y mantendría bloqueado a cualquier proceso actual o futuro que quisiera ejecutar la operación **P()**.

Para lograr que todos los procesos sean avisados, podemos usar un pequeño *truco*:

```
// variable compartida
Semáforo eventoSucedio = 0;
```

`// donde ocurre el evento`
`V(eventoSucedio);`
`...`
	`// donde hay que esperar por el evento`
	`P(eventoSucedio);`
	`V(eventoSucedio);`
	`...`

El truco consiste en que cada proceso que consiga desbloquearse de la operación **P()** vuelve a poner a 1 el valor del semáforo, con lo cual consigue desbloquear a un compañero. Y este hará lo mismo, con lo cual se desbloqueará a otro compañero, y así hasta que no haya más procesos bloqueados. Tras esta cadena de desbloqueos, el valor final del semáforo será 1: gracias a ello, si en un momento futuro un proceso ejecuta el código de espera por el evento, podrá ejecutar la operación **P()** sin bloquearse.

Recurso que admite N procesos concurrentes

Otro problema que se resuelve con facilidad mediante semáforos es controlar el acceso a un recurso que admite hasta N procesos trabajando en él de forma simultánea. Por ejemplo, una sala con capacidad para N personas, un búfer que contiene N entradas, etc.

```
// Variable compartida
Semáforo puedoPasar = N;
```

```
// código para acceder al recurso compartido
P(puedoPasar);
  … trabajar con el recurso …
V(puedoPasar);
```

Como puede verse, el código es idéntico al de sección crítica, salvo que el valor inicial del semáforo es un número arbitrario. De hecho, el problema de la sección crítica es un caso particular del aquí descrito, cuando N=1.

Espera condicional con semáforos

El mismo algoritmo que hemos propuesto en el Capítulo 1 lo podemos adaptar para utilizar semáforos como mecanismo básico de bloqueo y desbloqueo. Esto se logra utilizando un semáforo *mutex* para implementar la sección crítica y otro semáforo *cola* inicializado a cero para gestionar la cola de procesos que se van bloqueando en espera de una condición lógica.

Las operaciones ENTRARSC() y SALIRSC() las resolvemos directamente con operaciones P y V del semáforo *mutex.* Las operaciones DORMIR() y DESPERTAR() necesitan apoyarse en un contador entero para saber en todo momento cuántos procesos están bloqueados.

```
// Estructura para resolver problemas de
// sincronización que tienen esperas condicionales
struct Monitor {
  Semáforo mutex;
  Semáforo cola;
  int esperando;
}

void inicializa_monitor (Monitor* m) {
  m->mutex = 1;
  m->cola = 0;
  m->esperando = 0;
}
```

```
// ** equivalente a la operación DORMIR
// Nos bloqueamos en el semáforo «cola».
// Previamente liberamos el mutex para no producir interbloqueos
// e incrementamos «esperando» para registrar que hay un proceso
// más en espera;
// cuando nos desbloqueen, intentamos recuperar el mutex.
void dormir (Monitor* m) {
  V(m->mutex);
  m->esperando++;
  P(m->cola);
  P(m->mutex);
}

// ** equivalente a la operación DESPERTAR
// desbloqueamos a todos los procesos que están esperando en «cola»
void despertar (Monitor* m) {
  while (m->esperando > 0) {
    m->esperando--;
    V(m->cola);
  }
}
```

Una vez que hemos adaptado a semáforos las operaciones básicas del capítulo 1, este es el algoritmo para resolver cualquier situación de espera por condición lógica:

```
// Variable compartida por todos los procesos afectados
// Contiene dos semáforos y un contador entero
Monitor mio;
inicializa_monitor(&mio);
```

```
// código para mantener el bloqueo
P(mio.mutex);
while ( condición de bloqueo ) {
  dormir(&mio);
}
...acciones sobre las variables de estado…
… acciones sobre el objeto compartido …
despertar(&mio);
V(mio.mutex);
```

Como nota histórica, el nombre «monitor» que hemos usado tampoco es casual. En 1972 C.A.R. Hoare propuso una herramienta para sincronizar procesos llamada justamente «monitor», que incluía las dos operaciones de «dormir» y «despertar» que hemos estado tratando a lo largo de este documento. El concepto de monitor ha tenido gran influencia en el diseño de lenguajes de programación y de hecho se ha aplicado en Java, el cual ofrece en la clase Object unos métodos de sincronización llamados **wait()** y **notifyAll()** con un funcionamiento casi idéntico al que hemos descrito para las dos operaciones de bloqueo y desbloqueo.

En las prácticas de la asignatura usaremos lenguaje C, la biblioteca pthreads y una pequeña biblioteca que implementa los semáforos, con sus operaciones P y V tal y como las hemos definido aquí.

Capítulo 2: ejercicios con semáforos

5 Reescribe con semáforos el algoritmo del sistema camarero-cocinero.

6 Resuelve con semáforos el problema de la pajarería (el ejercicio 1 y la variante del ejercicio 4).

7 Resuelve con semáforos el problema de la embotelladora (el ejercicio 2).

8 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 semáforos, se pide introducir las modificaciones necesarias para que por cada ejecución de **«a»** o de **«e»** se permita ejecutar una iteración de **«d»**.

9 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 semáforos.

10 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 semáforos.