

Examen de Programación Concurrente

Dpto. LSIS. Unidad de Programación

Febrero 2001

Este documento está disponible en HTML, PostScript, PostScript comprimido con zip, PostScript comprimido con gzip, PDF, PDF comprimido con zip y PDF comprimido con gzip.

1 Controlador de un ascensor

Nota importante: el texto de descripción de este problema es **exactamente** el mismo que el dejado anteriormente en fotocopiadora y en WWW, al igual que la primera pregunta de este ejercicio. La segunda pregunta es nueva, y debe realizarse durante este examen.

Tenemos un ascensor que atiende P pisos, numerados del 1 al P ; se supone la existencia de un tipo $TipoNumPlanta = \{1..P\}$. En cada piso hay un único botón de llamada al ascensor, y en la cabina del ascensor hay un botón para cada piso. En cada piso, a media altura respecto de la puerta del ascensor, hay un sensor que detecta la llegada de la cabina; el sensor no conoce en qué dirección va el ascensor, y tampoco da ninguna señal que nos indique que el ascensor abandona el piso: se limita a recoger cambios de estado de no haber ascensor a haber ascensor (ver la figura

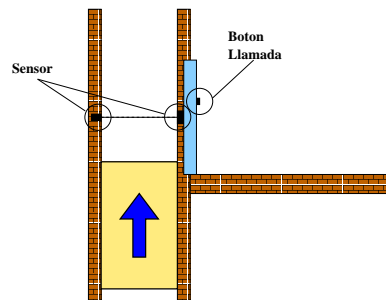


Figura 1: Esquema del ascensor

Se trata de diseñar un *controlador del ascensor* que atienda a las llamadas desde los pisos o la cabina, haciendo que el ascensor suba o baje en función de las mismas, y ocupándose también de que el ascensor se detenga en los pisos en que se haya solicitado parada y se abran y cierren las puertas. Los dispositivos físicos que el controlador debe manejar son las puertas y el motor que mueve el ascensor.

El controlador recibirá la información de los sensores para determinar la posición del ascensor y deberá ocuparse de la parada del mismo cuando corresponda, siempre en función de las llamadas pendientes. Si el ascensor debe detenerse en un piso p se debe ordenar la parada en el momento en que se detecte el paso por el sensor correspondiente a ese piso.

Las puertas deben abrirse automáticamente al llegar a un piso en que realice una parada. Las puertas se cierran automáticamente cuando se pulsa un botón de petición de piso desde el panel del ascensor, o, si no se pulsa ningún botón dentro del ascensor, cuando hayan transcurrido T segundos desde la apertura de las puertas. De haber alguna petición pendiente, el ascensor debe ponerse en marcha inmediatamente después del cierre de las puertas. Una llamada desde un piso cuando el ascensor está detenido con las puertas abiertas **no** debe cerrarlas, pero debe quedar registrada. El ascensor no debe estar en movimiento a menos que haya alguna petición pendiente.

Interfaz con los dispositivos externos

Se dispone de los siguientes procedimientos ya implementados para actuar sobre los distintos dispositivos involucrados:

Control del motor:

MOTOR.Subir() Pone en marcha el motor causando la subida del ascensor. El ascensor debe encontrarse detenido cuando se llama a esta operación.

MOTOR.Bajar() Ídem para bajar.

MOTOR.Parar() Para el motor, deteniendo el movimiento de la cabina. El ascensor se para automáticamente en el piso inmediatamente próximo según la dirección que llevase en el momento de la llamada. Es decir, si un sensor de un piso N detecta el paso del ascensor (ya sea subiendo o bajando) y en ese momento se llama a **MOTOR.Parar**, el ascensor tiene capacidad para detenerse en el piso N , y lo hará.

El controlador debe asegurarse de que en ningún momento se llama a dos o más de estas operaciones simultáneamente, pues su implementación no garantiza la exclusión mutua. Cada una de ellas retorna cuando el ascensor se ha puesto en marcha o se ha detenido.

Control de las puertas:

PUERTA.Abrir() Abre las puertas del ascensor. Retorna cuando las puertas están completamente abiertas. La cabina debe estar detenida y las puertas completamente cerradas en el momento de llamar a esta operación.

PUERTA.Cerrar() Cierra las puertas del ascensor. Retorna cuando las puertas están completamente cerradas. La cabina debe estar detenida y las puertas completamente abiertas en el momento de llamar a esta operación.

De nuevo se requiere garantizar que no habrá dos operaciones simultáneamente activas; la exclusión mutua no está dada por las operaciones en sí. El tiempo necesario para abrir o cerrar las puertas está acotado, y es relativamente corto, pero no podemos hacer suposiciones acerca de su duración con respecto a la de otros sucesos de este problema.

Lectura de peticiones de pisos:

BOTON.LeerCabina(VAR numPlanta: TipoNumPlanta) La llamada se bloquea y retorna cuando desde el interior de la cabina se pulsa el botón correspondiente a un piso, devolviendo éste en el parámetro de salida del procedimiento. Supondremos que no se pulsan varios botones simultáneamente.

BOTON.LeerPlanta(numPlanta: TipoNumPlanta) La llamada retorna cuando se pulsa el botón de llamada de la planta `numPlanta`. El parámetro `numPlanta` es entrada, **no** de salida.

No se puede suponer ningún límite de tiempo para el retorno de ninguna de estas llamadas. En particular no se puede suponer que en algún momento alguien hará una petición para un piso dado.

Detección de paso del ascensor por los sensores:

SENSOR.DetectarPaso(VAR numPlanta: TipoNumPlanta) La llamada se queda bloqueada y retorna cuando el sensor detecta la llegada del ascensor a una planta, y devuelve el número de planta por la que está pasando.

Control del tiempo:

TIEMPO.ahora(): **CARDINAL** Devuelve la hora actual en segundos contados desde un momento en el pasado.

TIEMPO.espera(lapso: CARDINAL) Se bloquea durante el tiempo, expresado en segundos, determinado por su argumento.

De cara a la especificación, las llamadas a operaciones que actúan sobre un dispositivo externo pueden ponerse como **POST(Subir)**, si ello es necesario. Las operaciones que devuelven un resultado se supondrán especificadas como acciones.

Aclaración: el algoritmo del ascensor

Aunque no es estrictamente necesario, se sugiere utilizar el llamado algoritmo del ascensor, aplicado frecuentemente en el acceso a discos. La idea de este algoritmo es que el ascensor sube y baja alternativamente. Mientras va en una dirección se efectúan las paradas necesarias en esa dirección. Cuando no quedan más peticiones en la dirección actual, se invierte el sentido de la marcha. Si no hay ninguna petición pendiente, el ascensor se detiene.

Resultados a entregar

1. **Este apartado es exactamente igual al ya dejado para su realización.** Sobre el problema anteriormente descrito, **se pide:**

- Análisis del problema, identificando y justificando qué procesos y recursos (pueden ser uno o más de uno) se han de utilizar. No es necesario considerar ni especificar las operaciones correspondientes al **MOTOR**, **PUERTA**, **SENSOR**, **BOTON** o **TIEMPO**. Se suponen dadas, y son operaciones externas.
- Grafo de procesos y recursos, aclarando:
 - qué operaciones y variables de estado tiene cada uno de los recursos diseñados,

- qué operaciones de cada recurso necesita cada uno de los procesos identificados,
- qué operaciones externas usa cada uno de los procesos y de los recursos.
- Código de cada uno de los procesos anteriormente identificados.
- Especificación formal de los recursos, donde se modele claramente la sincronización entre las operaciones, anadiendo una explicación de esa sincronización en lenguaje natural. La selección del piso a atender debe garantizar la ausencia de inanición. No es necesario hacer las tablas de bloqueos ni de desbloqueos.

2. **Este apartado es nuevo.** Corregir el diseño anterior, adaptándolo a la siguiente situación:

En lugar de un solo ascensor, se controlarán conjuntamente N ascensores. Para ello se modifica el interfaz con los módulos MOTOR, PUERTA, BOTON y SENSOR de manera que tengan un parámetro que indique el ascensor al que se refieren, de tipo `TipoNumAscensor` (`TYPE TipoNumAscensor = [1..N]`):

`MOTOR.Subir/Bajar/Parar(k: TipoNumAscensor)`

`PUERTA.Abrir/Cerrar(k: TipoNumAscensor)`

`BOTON.LeerCabina(k: TipoNumAscensor; VAR numPlanta: TipoNumPlanta)`

`SENSOR.DetectarPaso(k: TipoNumAscensor; VAR numPlanta: TipoNumPlanta)`

El controlador debe asegurar que no se dan dos órdenes simultáneas para el mismo ascensor, pero sí se admiten órdenes simultáneas si van dirigidas a ascensores diferentes.

Se mantiene un solo botón de llamada en cada piso, pero se simplifica la operación de lectura de esos botones para hacerla similar a la lectura de los botones de una cabina:

`BOTON.LeerPlanta(VAR numPlanta: TipoNumPlanta)` La llamada se bloquea y retorna cuando se haya pulsado algún botón de llamada desde un piso, devolviendo éste en el parámetro de salida del procedimiento. Supondremos que no se pulsan varios botones al mismo tiempo.

Con estos supuestos, **se pide**:

- Rehacer el diseño, indicando los procesos y recursos que se van a utilizar en la nueva situación.
- Dibujar el nuevo diagrama de procesos y recursos.
- Especificar los procesos, en pseudocódigo.
- Especificar los recursos, indicando formalmente las variables de estado necesarias y los parámetros de cada operación. Las precondiciones y postcondiciones de las operaciones se indicarán de manera informal (en lenguaje natural).

Solución propuesta — Parte 1

Para determinar qué procesos se necesitan, se puede analizar qué operaciones no instantáneas puede que tengan que realizarse al mismo tiempo. Esto ocurre con las siguientes:

- Lectura de pulsaciones de los botones de la cabina (de uno en uno).
- Lectura de pulsaciones de los botones de las plantas (todos a la vez).
- alguna de las siguientes: arrancar o parar el motor, abrir o cerrar las puertas, esperar a que se llegue al siguiente piso (sólo una de ellas al tiempo).
- Dependiendo de cómo se realice la temporización de la apertura de puertas, puede ser necesaria otra operación simultánea más, que es mantener la temporización hasta el tiempo límite aunque ya se estén cerrando las puertas porque se ha pulsado un botón de la cabina.

La comunicación y sincronización entre los procesos puede hacerse mediante un recurso compartido que gestione las peticiones pendientes y el estado del ascensor en un momento dado. Este recurso contendrá información sobre:

- Peticiones realizadas para los diferentes pisos.
- Sentido de movimiento actual.
- Posición actual del ascensor.
- Indicación de si se pueden cerrar ya las puertas.

Esta última información podría plantearse también como correspondiente a un recurso separado que gestione solamente la temporización de la apertura y cierre de las puertas.

También es posible realizar la temporización directamente en el proceso que ordena el movimiento del ascensor, si el tiempo se va contando por periodos pequeños, y no con una espera única hasta el tiempo límite.

A continuación se describe la solución que usa un proceso separado para temporizar, y un recurso único que gestiona las peticiones y el permiso para cerrar las puertas. El diagrama de procesos y recursos se muestra en la figura

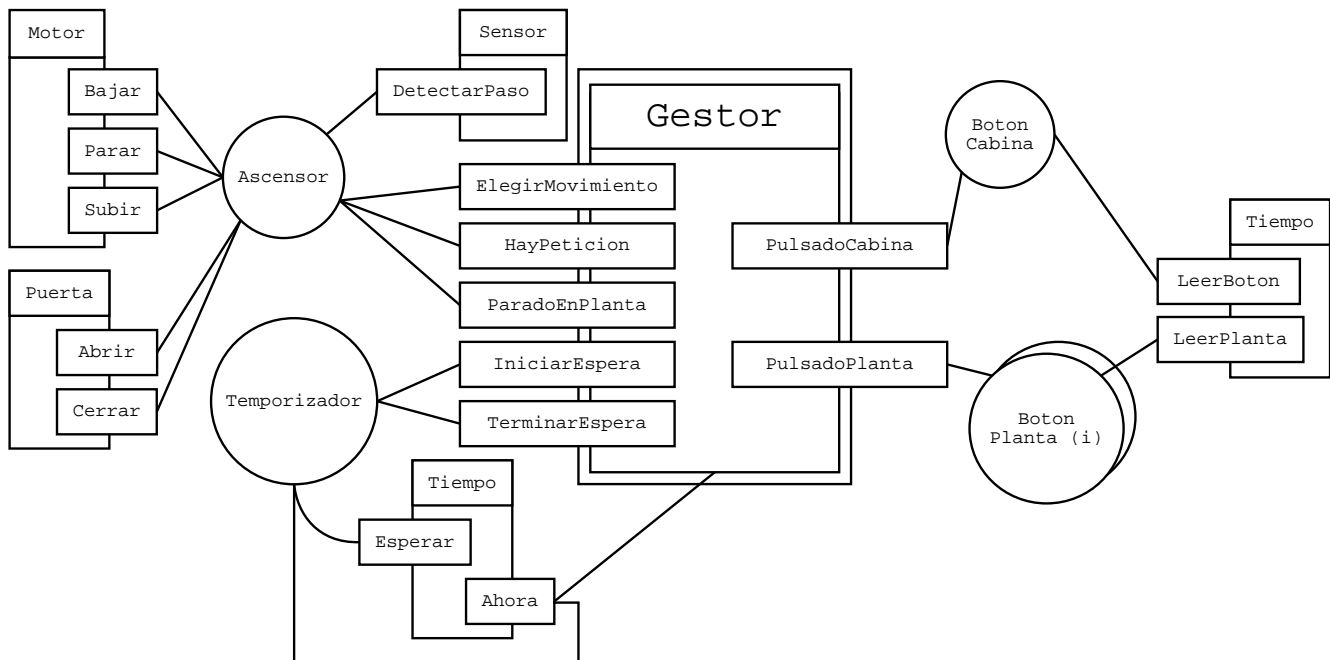


Figura 2: Diagrama de procesos y recursos — primer supuesto

Código de las tareas

```

TASK BotonCabina ;
  VAR p: TipoNumPlanta ;
BEGIN
  LOOP
    BOTON. LeerCabina (p);
    GESTOR. PulsadoCabina (p);
  END
END BotonCabina ;

TASK BotonPlanta (p: TipoNumPlanta);
BEGIN
  LOOP
    BOTON. LeerPlanta (p);
    GESTOR. PulsadoPlanta (p);
  END
END BotonPlanta ;

TASK Temporizador ;
  VAR t: CARDINAL;
BEGIN
  LOOP
    GESTOR. IniciarEspera (t);
    TIEMPO. Espera (t – TIEMPO. Ahora );
    GESTOR. TerminarEspera ;
  END
END Temporizador ;

TASK Ascensor ;
  VAR p: TipoNumPlanta ;
  movimiento: TipoMovimiento ;
  parar: BOOLEAN;
BEGIN
  <<inicializar el ascensor parado en alguna
  planta 'p' y con las puertas abiertas>>
  GESTOR. ParadoEnPlanta (p);
  LOOP
    GESTOR. ElegirMovimiento (movimiento);
    PUERTA. Cerrar ;
    IF movimiento = SUBIR THEN
      MOTOR. Subir ;
    ELSE
      MOTOR. Bajar ;
    END;
    REPEAT
      SENSOR. DetectarPaso (p);
      GESTOR. HayPeticion (p, parar);
    UNTIL parar ;
    MOTOR. Parar ;
    PUERTA. Abrir ;
    GESTOR. ParadoEnPlanta (p);
  END
END Ascensor ;

```

Especificación del recurso

C-TADSOL *TipoGestor*

C-Operaciones

ACCION *PulsadoCabina*: $TipoGestor \times TipoNumPlanta[ent]$
ACCION *PulsadoPlanta*: $TipoGestor \times TipoNumPlanta[ent]$
ACCION *HayPeticion*: $TipoGestor \times TipoNumPlanta[ent] \times \mathcal{B}[sal]$
ACCION *ParadoEnPlanta*: $TipoGestor \times TipoNumPlanta[ent]$
ACCION *IniciarEspera*: $TipoGestor \times \mathcal{N}[sal]$
ACCION *TerminarEspera*: $TipoGestor$
ACCION *ElegirMovimiento*: $TipoGestor \times TipoMovimiento[sal]$

TRANSACCIONES

PulsadoCabina
PulsadoPlanta
HayPeticion
ParadoEnPlanta
IniciarEspera
TerminarEspera
ElegirMovimiento

CONCURRENCIA

ninguna

Dominio

Tipo: $TipoGestor = (peticion: Secuencia(\mathcal{B}) \times sentido: TipoMovimiento \times posicion: TipoNumPlanta \times$
 $pulsadaCabina: \mathcal{B} \times tiempoActual: \mathcal{N} \times tiempoLimite: \mathcal{N})$
 $TipoNumPlanta = 1..P$
 $TipoMovimiento = (SUBIR \mid BAJAR)$
Invariante: $\forall t \in TipoGestor \bullet Longitud(t.peticion) = P$

CPRE: cierto

PulsadoCabina(g, p)**POST**: Anota la petición y marca pulsación desde cabina**POST**: $g^{ent} = g^{sal} / g^{sal}.peticion_p \wedge g^{sal}.pulsadaCabina$

CPRE: cierto

PulsadoPlanta(g, p)

POST: Anota la petición

POST: $g^{ent} = g^{sal} / g^{sal}.peticion_p$

CPRE: cierto

HayPeticion(g, p, hay)

POST: Indica si hay petición para esa planta

POST: $g^{ent} = g^{sal} \wedge hay^{sal} = peticion_p$

CPRE: cierto

ParadoEnPlanta(g, p)

POST: Anota que se ha atendido la petición de esa planta, y comienza el plazo de apertura de puertas

POST: $g^{ent} = g^{sal} / \neg g^{sal}.peticion_p \wedge g^{sal}.posicion = p \wedge \neg g^{sal}.pulsadaCabina \wedge g^{sal}.abierto \wedge g^{sal}.tiempoActual = TIEMPO.Ahora \wedge g^{sal}.tiempoLimite = g^{sal}.tiempoActual + T$

CPRE: $g^{ent}.tiempoActual < g^{ent}.tiempoLimite$

IniciarEspera(g, t)

POST: Indica el tiempo límite hasta el que hay que esperar

POST: $g^{ent} = g^{sal} \wedge t^{sal} = g^{ent}.tiempoLimite$

CPRE: cierto

TerminarEspera(g)

POST: Anota el nuevo tiempo actual

POST: $g^{ent} = g^{sal} / g^{sal}.tiempoActual = TIEMPO.Ahora$

CPRE: Hay peticiones para otra planta y se han mantenido las puertas abiertas el tiempo suficiente

CPRE: $\exists p \in TipoNumPlanta \bullet p \neq g^{ent}.posicion \wedge g^{ent}.peticion_p \wedge p \neq g^{ent}.posicion \wedge (g^{ent}.pulsadaCabina \vee g^{ent}.tiempoActual < g^{ent}.tiempoLimite)$

ElegirMovimiento(g, mov)

POST: Indica que hay que subir si hay peticiones superiores y, o bien está subiendo o no hay peticiones inferiores. Indica que hay que bajar en caso contrario. Además anota el nuevo sentido de movimiento y que las puertas están cerradas. y anula peticiones para el piso en que estaba.

POST: $g^{ent} = g^{sal} / g^{sal}.sentido = g^{sal}.mov \wedge \neg g^{sal}.abierto \wedge g^{sal}.peticion_p \wedge g^{sal}.mov = \begin{cases} \text{SUBIR} & \exists p \in TipoNumPlanta \bullet p > g^{ent}.posicion \wedge g^{ent}.peticion_p \wedge (sentido = SUBIR \vee \neg \exists k \in TipoNumPlanta \bullet k < posicion \wedge peticion_k) \\ \text{BAJAR} & e.o.c. \end{cases}$

Solución propuesta — Parte 2

Se puede mantener en esencia el diseño anterior, con los siguientes cambios.

- La lectura de los botones de las plantas se hará con un solo proceso.
- Habrá un proceso Cabina y un proceso Ascensor por cada ascensor
- Se mantiene un solo proceso Temporizador, que irá marcando en cada momento al final del plazo de espera más corto de los que estén pendientes (también se podría haber repetido el proceso Temporizador por cada ascensor).
- Se mantiene un solo recurso que gestiona el estado de todos los ascensores, cambiando lo siguiente:
 - Las acciones PulsadoCabina, HayPeticion, ParadoEnPlanta y ElegirMovimiento tendrán un parámetro adicional para indicar a qué ascensor se refieren.
 - Se repiten los elementos del dominio para cada uno de los ascensores.
 - Además habrá un vector de peticiones desde los pisos, común a todos los ascensores.

Estas peticiones tendrán tres estados: inactiva, pedida y asignada. Esto es necesario para evitar que varios ascensores acudan a la misma llamada de un piso. Una petición de piso no asignada se asigna a un ascensor cuando está en su sentido de movimiento y no hay peticiones de su cabina en ese sentido, o bien cuando el ascensor se para en ese piso por el motivo que sea.

El diagrama de procesos y recursos se muestra en la figura

Código de las tareas

```

TASK BotonCabina(a: TipoNumAscensor);
  VAR p: TipoNumPlanta;
BEGIN
  LOOP
    BOTON.LeerCabina(a, p);
    GESTOR.PulsadoCabina(a, p);
  END
END BotonCabina;

TASK BotonPlanta;
BEGIN
  LOOP
    BOTON.LeerPlanta(p);
    GESTOR.PulsadoPlanta(p);
  END
END BotonPlanta;

TASK Temporizador;
  VAR t: CARDINAL;
BEGIN
  LOOP
    GESTOR.IniciarEspera(t);
    TIEMPO.Espera(t - TIEMPO.Ahora);
    GESTOR.TerminarEspera;
  END
END Temporizador;

```

```

TASK Ascensor(a: TipoNumAscensor);
  VAR p: TipoNumPlanta;
      movimiento: TipoMovimiento;
      parar: BOOLEAN;
BEGIN
  <<inicializar el ascensor parado en alguna
    planta 'p' y con las puertas abiertas>>
  GESTOR.ParadoEnPlanta(a, p);
  LOOP
    GESTOR.ElegirMovimiento(a, movimiento);
    PUERTA.Cerrar;
    IF movimiento = SUBIR THEN
      MOTOR.Subir(a);
    ELSE
      MOTOR.Bajar(a);
    END;
    REPEAT
      SENSOR.DetectarPaso(a, p);
      GESTOR.HayPeticion(a, p, parar);
    UNTIL parar;
    MOTOR.Parar(a);
    PUERTA.Abrir(a);
    GESTOR.ParadoEnPlanta(a, p);
  END
END Ascensor;

```

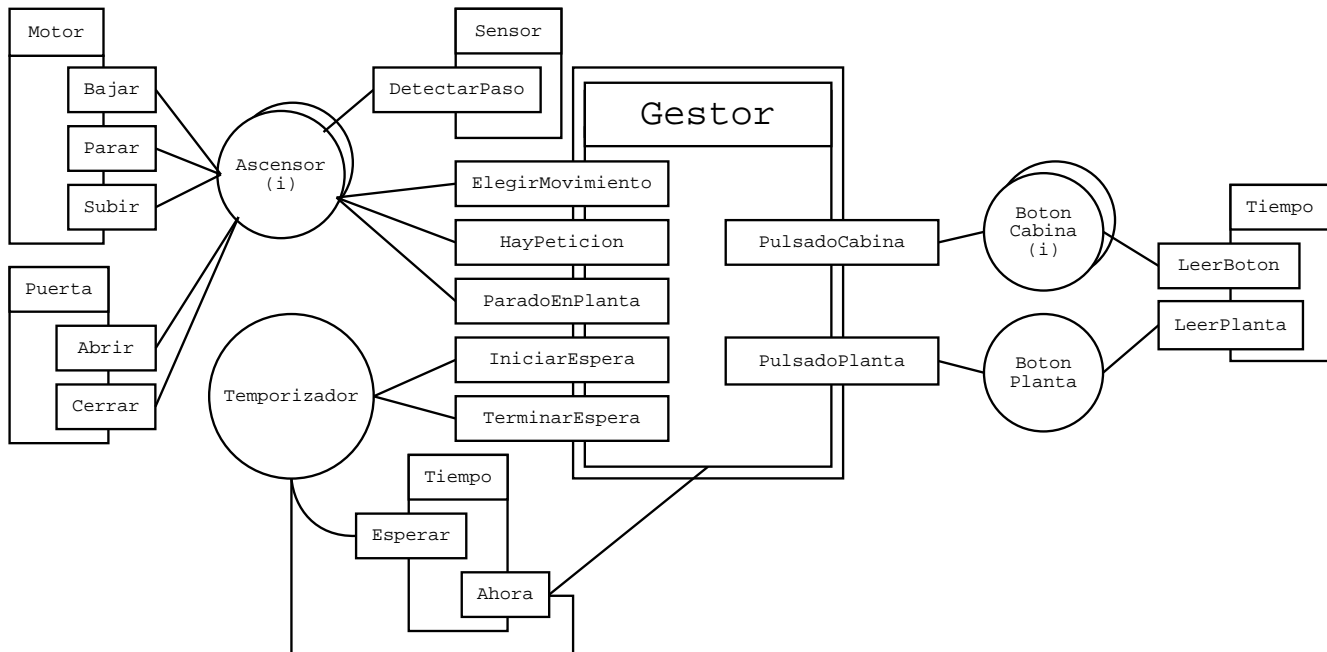


Figura 3: Diagrama de procesos y recursos — segundo supuesto

Especificación del recurso

C-TADSOL TipoGestor

C-Operaciones

ACCION PulsadoCabina: TipoGestor \times TipoNumAscensor[ent] \times TipoNumPlanta[ent]ACCION PulsadoPlanta: TipoGestor \times TipoNumPlanta[ent]

ACCION *HayPeticion*: $TipoGestor \times TipoNumAscensor[ent] \times TipoNumPlanta[ent] \times \mathcal{B}[sal]$

ACCION *ParadoEnPlanta*: $TipoGestor \times TipoNumAscensor[ent] \times TipoNumPlanta[ent]$

ACCION *IniciarEspera*: $TipoGestor \times \mathcal{N}[sal]$

ACCION *TerminarEspera*: $TipoGestor$

ACCION *ElegirMovimiento*: $TipoGestor \times TipoNumAscensor[ent] \times TipoMovimiento[sal]$

TRANSACCIONES

PulsadoCabina

PulsadoPlanta

HayPeticion

ParadoEnPlanta

IniciarEspera

TerminarEspera

ElegirMovimiento

CONCURRENCIA

ninguna

DOMINIO

TIPO: $TipoGestor = (\text{petPlanta: } Secuencia(TipoPeticion) \times \text{peticion: } Secuencia(Secuencia(\mathcal{B})) \times$
 $\text{asignada: } Secuencia(\mathcal{N}) \times \text{sentido: } Secuencia(TipoMovimiento) \times$
 $\text{posicion: } Secuencia(TipoNumPlanta) \times \text{pulsadaCabina: } Secuencia(\mathcal{B}) \times$
 $\text{tiempoActual: } \mathcal{N} \times \text{tiempoLimite: } Secuencia(\mathcal{N}))$

$TipoNumPlanta = (1..P)$

$TipoMovimiento = (\text{SUBIR} \mid \text{BAJAR})$

$TipoPeticion = (\text{INACTIVA, PEDIDA, ASIGNADA})$

INVARIANTE: $\forall g \in TipoGestor \bullet \text{Longitud}(g.\text{petPlanta}) = P \wedge \text{Longitud}(g.\text{sentido}) = NumAscensores \wedge$
 $\text{Longitud}(g.\text{asignada}) = NumAscensores \wedge \text{Longitud}(g.\text{posicion}) = NumAscensores \wedge$
 $\text{Longitud}(g.\text{pulsadaCabina}) = NumAscensores \wedge$
 $\text{Longitud}(g.\text{tiempoActual}) = NumAscensores \wedge$
 $\text{Longitud}(g.\text{tiempoLimite}) = NumAscensores \wedge$
 $\forall i \in \{1..NumAscensores\} \bullet \text{Longitud}(g.\text{peticion}_i) = P$

CPRE: cierto

PulsadoCabina(g, a, p)

POST: Anota la petición y marca pulsación desde cabina

CPRE: cierto

PulsadoPlanta(g, p)

POST: Anota la petición, si no lo estaba todavía

CPRE: cierto

HayPeticion(g, a, p, hay)

POST: Indica si hay petición desde cabina o planta para ese ascensor y planta. Si hay petición PEDIDA para esa planta, se le asigna a ese ascensor, deshaciendo la asignación anterior, si la hubiera.

CPRE: cierto

ParadoEnPlanta(g, a, p)

POST: Anota que se ha atendido la petición de cabina y/o planta ASIGNADA de esa planta, y comienza el plazo de apertura de puertas

CPRE: Hay alguna temporización pendiente

CPRE: $\exists a \in TipoNumAscensor \bullet g.\text{tiempoActual} < g.\text{tiempoLimite}_a$

IniciarEspera(g, t)

POST: Indica el mínimo tiempo límite hasta el que hay que esperar

CPRE: cierto

TerminarEspera(g)

POST: Anota el nuevo tiempo actual

CPRE: Hay alguna petición para atender por ese ascensor (desde cabina, o bien desde planta, no asignada) y se ha cumplido el plazo para mantener las puertas abiertas.

ElegirMovimiento(g, a, mov)

POST: Indica que hay que subir si hay peticiones superiores y, o bien está subiendo o no hay peticiones inferiores. Indica que hay que bajar en caso contrario. Además anota el nuevo sentido de movimiento. La asignación de petición desde una planta a ese ascensor se hace si no hay peticiones de cabina pero sí peticiones de planta en el sentido de su movimiento.

2 El almacén de las vacas locas

Debido al problema de contaminación priónica en vacas, el número de sacrificios está causando escasez de recintos que puedan ser dedicados a almacenarlas. El gobierno decide autorizar el uso de salas de mataderos para almacenar reses sanas y enfermas, con la restricción de que una sala sólo debe albergar, en cada momento, vacas de una clase.

Con objeto de optimizar el uso de las salas no se hace una asignación estricta de salas a tipos de vacas, sino que cuando una sala queda vacía (porque todas las vacas que había allí ha sido llevada a cremar o a una carnicería) se puede reutilizar para guardar indistintamente vacas sanas o contaminadas (tras una desinfección de la que no tenemos que ocuparnos). Previendo la posibilidad de fallos humanos, se decide automatizar el sistema de asignación de salas con un sistema de control. El funcionamiento del sistema es el siguiente: se senala, al llegar, si se quiere depositar o retirar una vaca y si ésta está contaminada o no. El sistema asignará una sala de la que extraer o recoger. El diseño del recurso al que se ha llegado es el siguiente:

C-TADSOL *Almacen Vacas*

C-Operaciones

ACCION *DejarLoca*: $TipoAlm \times TipoNumSala$

ACCION *RetirarLoca*: $TipoAlm \times TipoNumSala$

ACCION *DejarNormal*: $TipoAlm \times TipoNumSala$

ACCION *RetirarNormal*: $TipoAlm \times TipoNumSala$

Operaciones

ACCION *Iniciar*: $TipoAlm$

CONCURRENCIA:

ninguna

TRANSACCIONES:

DejarLoca

RetirarLoca

DejarNormal

RetirarNormal

Dominio:

Tipo: $TipoAlm = Secuencia(TipoSala)$

$TipoNumSala = \{1..NumSalas\}$

Donde:

$TipoSala = (nvacas: TipoTam \times tipo: TipoVaca)$

$TipoVaca = loca \mid normal$

$TipoTam = \{0..TamSala\}$

$TamSala = \dots$

$NumSalas = \dots$

Invariante: $\forall a \in TipoAlm \bullet (Longitud(a) = NumSalas)$

CPRE: *cierto*

Iniciar(alm)

POST: *Todas las salas estan inicialmente vacías*

POST: $\forall i \in TipoNumSala \bullet (alm_i^{sal}.nvacas = 0)$

CPRE: *Hay una sala de vacas locas con espacio u otra cualquiera vacía*

CPRE: $\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas = 0 \vee (alm_i^{ent}.nvacas < TamSala \wedge alm_i^{ent}.tipo = loca))$

DejarLoca(alm, sala)

POST: *Se utiliza la sala de vacas locas o la sala vacía*

POST: $\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas = 0 \vee (alm_i^{ent}.nvacas < TamSala \wedge alm_i^{ent}.tipo = loca) \wedge sala^{sal} = i \wedge alm^{sal} = alm^{ent} \setminus alm_i^{sal}.nvacas = alm_i^{ent}.nvacas + 1 \wedge alm_i^{sal}.tipo = loca)$

CPRE: *Hay una sala con vacas locas*

CPRE: $\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas > 0 \wedge alm_i.tipo = loca)$

RetirarLoca(alm, sala)

POST: *Se retira una vaca de una sala con vacas locas*

POST: $\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas > 0 \wedge alm_i^{ent}.tipo = loca \wedge sala^{sal} = i \wedge alm^{sal} = alm^{ent} \setminus alm_i^{sal}.nvacas = alm_i^{ent}.nvacas - 1)$

CPRE: *Hay una sala de vacas normales con espacio u otra cualquiera vacía*

CPRE: $\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas = 0 \vee (alm_i^{ent}.nvacas < TamSala \wedge alm_i^{ent}.tipo = normal))$

DejarNormal(alm, sala)

POST: *Se utiliza la sala de vacas normales o la sala vacía*

POST: $\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas = 0 \vee (alm_i^{ent}.nvacas < TamSala \wedge alm_i^{ent}.tipo = normal) \wedge sala^{sal} = i \wedge alm^{sal} = alm^{ent} \setminus alm_i^{sal}.nvacas = alm_i^{ent}.nvacas + 1 \wedge alm_i^{sal}.tipo = normal)$

CPRE: *Hay una sala con vacas normales*

CPRE: $\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas > 0 \wedge alm_i^{ent}.tipo = normal)$

RetirarNormal(alm, sala)

POST: *Se retira una vaca de una sala con vacas normales*

POST: $\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas > 0 \wedge alm_i^{ent}.tipo = normal \wedge sala^{sal} = i \wedge alm^{sal} = alm^{ent} \setminus alm_i^{sal}.nvacas = alm_i^{ent}.nvacas - 1)$

Notas

- Con objeto de agilizar la entrada y salida de vacas se impone una condición adicional, no contemplada en el recurso por razones de claridad: no se deben ocupar todas las salas con vacas de un solo tipo.
- Asimismo es necesario tener en cuenta la posibilidad de un despertar en cascada al retirar una vaca loca o normal.
- No es necesario codificar operaciones rutinarias, como buscar una sala vacía. Pueden darse **nombres significativos** a esas operaciones y dejar escrito en lenguaje natural (pero preciso) el significado que se le asume.

Se pide:

- Realizar la tabla de bloqueos y desbloqueos del recurso, tal y como se ha especificado formalmente.
- Estudiar cómo se puede asegurar la vivacidad del sistema, respetando la restricción introducida en la nota anterior.
- Codificar las operaciones anteriores como procedimientos **Cc Modula** usando monitores, basándose en la especificación, las tablas de bloqueos y desbloqueos y las consideraciones sobre vivacidad anteriores.

Solución propuesta:

Tabla de bloqueos

Operación	CPRE informal	CPRE codificada	Condition
DejarLoca (alm, sala)	Hay una sala de vacas locas con espacio u otra cualquiera vacía	$\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas = 0 \vee (alm_i^{ent}.nvacas < TamSala \wedge alm_i^{ent}.tipo = loca))$	dejaesperando[loca]
RetirarLoca (alm, sala)	Hay una sala con vacas locas	$\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas > 0 \wedge alm_i.tipo = loca)$	retiraresperando[loca]
DejarNormal (alm, sala)	Hay una sala de vacas normales con espacio u otra cualquiera vacía	$\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas = 0 \vee (alm_i^{ent}.nvacas < TamSala \wedge alm_i^{ent}.tipo = normal))$	dejaesperando[normal]
RetirarNormal (alm, sala)	Hay una sala con vacas normales	$\exists i \in TipoNumSala \bullet (alm_i^{ent}.nvacas > 0 \wedge alm_i^{ent}.tipo = normal)$	retiraresperando[normal]

	<i>DejarLoca</i> (alm, sala) CPRE(<i>DejarLoca</i>) dejaresperando [loca]	<i>DejarNormal</i> (alm, sala) CPRE(<i>DejarNormal</i>) dejaresperando [normal]	<i>RetirarLoca</i> (alm, sala) CPRE(<i>RetirarLoca</i>) retiraresperando [loca]	<i>RetirarNormal</i> (alm, sala) CPRE(<i>RetirarNormal</i>) retiraresperando [normal]
<i>DejarLoca</i> (alm, sala) POST(<i>DejarLoca</i>)	$alm^{sal}_{sala^{sal}} < TamSala$ Despertar en cascada CPRE(<i>DejarLoca</i>) No aplicable cierto	CPRE(<i>DejarNormal</i>) No aplicable $alm^{sal}_{sala^{sal}} < TamSala$ Despertar en cascada $alm^{sal}_{sala^{sal}} = 0$	cierto	CPRE(<i>RetirarNormal</i>) No aplicable cierto
<i>DejarNormal</i> (alm, sala) POST(<i>DejarNormal</i>)				
<i>RetirarLoca</i> (alm, sala) POST(<i>RetirarLoca</i>)				
<i>RetirarNormal</i> (alm, sala) POST(<i>RetirarNormal</i>)	$alm^{sal}_{sala^{sal}} = 0$	cierto	CPRE(<i>RetirarLoca</i>) No aplicable	CPRE(<i>RetirarNormal</i>) No aplicable

Para asegurar la vivacidad se debe respetar en todo momento la restricción introducida en la primera nota. Esta restricción se respetará si en todo momento el número de salas dedicadas a un tipo de vaca es menor o igual a $NumSalas - 1$. Como el número de salas dedicadas a un tipo de vaca puede aumentar como consecuencia de ejecutar un operación **Dejar**, debemos modificar las CPREs de las operaciones **Dejar** de manera que recojan la restricción. Al modificar las CPREs, vamos a introducir dos contadores en el estado del recurso, **nosalaslocas** y **nosalasnormales**, que indicarán el número de salas dedicadas a vacas locas y a vacas normales, respectivamente. Lógicamente, estos contadores deben ser inicializados a cero. Dado que estos dos contadores deben formar parte del tipo **TipoAlm**, tenemos que modificar la definición del tipo *TipoAlm*:

$$TipoAlm = (salas : Secuencia(TipoSala) \times nosalaslocas : \mathcal{N} \times nosalasnormales : \mathcal{N})$$

Las CPREs de las operaciones **Dejar** quedarán así:

CPRE(DejarLoca): Hay una sala de vacas locas con espacio, u otra cualquiera vacía y el número de salas dedicadas a vacas locas es menor o igual que $NumSalas - 2$.

$$\exists i \in TipoNumSala \bullet ((alm^{ent}.salas_i.nvacas = 0 \wedge alm^{ent}.nosalaslocas \leq NumSalas - 2) \vee (alm^{ent}.salas_i.nvacas < TamSala \wedge alm^{ent}.salas_i.tipo = loca))$$

CPRE(DejarNormal): Hay una sala de vacas normales con espacio, u otra cualquiera vacía y el número de salas dedicadas a vacas normales es menor o igual que $NumSalas - 2$.

$$\exists i \in TipoNumSala \bullet ((alm^{ent}.salas_i.nvacas = 0 \wedge alm^{ent}.nosalasnormales \leq NumSalas - 2) \vee (alm^{ent}.salas_i.nvacas < TamSala \wedge alm^{ent}.salas_i.tipo = normal))$$

Dado que estas operaciones pueden incrementar el número de salas dedicadas a las vacas de un tipo, debemos también modificar las postcondiciones de las operaciones **Dejar**:

POST(DejarLoca): Se utiliza la sala de vacas locas, o la sala vacía y se incrementa el número de salas dedicadas a vacas locas.

$$\begin{aligned} \exists i \in TipoNumSala \bullet & ((alm^{ent}.salas_i.nvacas = 0 \wedge alm^{ent}.nosalaslocas \leq NumSalas - 2) \vee \\ & (alm^{ent}.salas_i.nvacas < TamSala \wedge alm^{ent}.salas_i.tipo = loca)) \wedge sala^{sal} = i \wedge \\ & alm^{sal} = alm^{ent} \setminus alm^{sal}.salas_i.nvacas = alm^{ent}.salas_i.nvacas + 1 \\ & \wedge alm^{sal}.salas_i.tipo = loca \wedge (alm^{ent}.salas_i.nvacas = 0 \rightarrow \\ & alm^{sal}.nosalaslocas = alm^{ent}.nosalaslocas + 1) \end{aligned}$$

POST(DejarNormal): Se utiliza la sala de vacas normales, o la sala vacía y se incrementa el número de salas dedicadas a vacas normales.

$$\begin{aligned} \exists i \in TipoNumSala \bullet & ((alm^{ent}.salas_i.nvacas = 0 \wedge alm^{ent}.nosalasnormales \leq NumSalas - 2) \vee \\ & (alm^{ent}.salas_i.nvacas < TamSala \wedge alm^{ent}.salas_i.tipo = normal) \wedge sala^{sal} = i \wedge \\ & alm^{sal} = alm^{ent} \setminus alm^{sal}.salas_i.nvacas = alm^{ent}.salas_i.nvacas + 1 \wedge \\ & alm^{sal}.salas_i.tipo = normal \wedge (alm^{ent}.salas_i.nvacas = 0 \rightarrow \\ & alm^{sal}.nosalasnormales = alm^{ent}.nosalasnormales + 1)) \end{aligned}$$

Asimismo, también debemos modificar las postcondiciones de las operaciones **Retirar**, ya que puede suceder que tras retirar una vaca de un tipo, una sala se queda vacía, y por lo tanto se decremente el número de salas dedicadas a dicho tipo.

POST(RetirarLoca): Se retirar una vaca de una sala con vacas locas

$$\begin{aligned} \exists i \in TipoNumSala \bullet & (alm^{ent}.salas_i.nvacas > 0 \wedge alm^{ent}.salas_i.tipo = loca \wedge \\ & sala^{sal} = i \wedge alm^{sal} = alm^{ent} \setminus alm^{sal}.salas_i.nvacas = alm^{ent}.salas_i.nvacas - 1 \wedge \\ & (alm^{sal}.salas_i.nvacas = 0 \rightarrow alm^{sal}.nosalaslocas = alm^{ent}.nosalaslocas - 1)) \end{aligned}$$

POST(RetirarNormal): Se retirar una vaca de una sala con vacas normales

$$\begin{aligned} \exists i \in TipoNumSala \bullet & (alm^{ent}.salas_i.nvacas > 0 \wedge alm^{ent}.salas_i.tipo = normal \wedge sala^{sal} = i \wedge \\ & alm^{sal} = alm^{ent} \setminus alm^{sal}.salas_i.nvacas = alm^{ent}.salas_i.nvacas - 1 \wedge \\ & (alm^{sal}.salas_i.nvacas = 0 \rightarrow alm^{sal}.nosalasnormales = alm^{ent}.nosalasnormales - 1)) \end{aligned}$$

Estos cambios introducidos en las CPREs y las postcondiciones de las operaciones **Dejar**, y en las postcondiciones de las operaciones **Retirar** van a afectar a dos condiciones de desbloqueo. Si al retirar una vaca de un tipo, se queda vacía la sala, entonces para poder desbloquear una operación **Dejar** del otro tipo de vacas, debemos comprobar $alm^{ent}.nosalas_{tipo} \leq NumSalas - 2$.

Código del programa

```

CONST
  normal = 1;
  loca = 2;
  NUMSALAS = ...;
  TAMSALA = ...;

(* Tipos de uso general *)

TYPE
  TipoVaca = [normal..loca];
  TipoNumSala = [1..NUMSALAS];

(*****
(* Implementacion del gestor del almacen *)
*****)

TYPE TipoGestorAlmacen = MONITOR

(* Constantes, tipos y nombres de procedimientos a usar dentro del monitor *)

IMPORT
  normal, loca, TipoVaca, TipoNumSala, NUMSALAS, TAMSALA;

(* Procedimientos públicos *)

PUBLIC
  DejarNormal,
  DejarLoca,
  RetirarNormal,
  RetirarLoca;

(* Tipos y variables privadas del monitor. *)

TYPE
  TipoTamSala = [0..TAMSALA];
  TipoSala = RECORD
    npiezas : TipoTamSala;
    tipo : TipoVaca;
  END;

VAR (* Estas variables se utilizan en la inicialización *)
  i: TipoNumSala;
  j: TipoVaca;

(* Variables que definen el estado interno del recurso *)
STATE
  almacen: ARRAY TipoNumSala OF TipoSala;
  nodejaesperando: ARRAY TipoVaca OF CARDINAL;
  dejaresperando: ARRAY TipoVaca OF CONDITION;
  retiraresperando: ARRAY TipoVaca OF CONDITION;
  saladisponible: TipoNumSala;
  nosalas: ARRAY TipoVaca OF CARDINAL;

(* Implementación de las operaciones del recurso *)

PROCEDURE BuscarSalaNoLlena (tipo: TipoVaca) : CARDINAL;
(**
* PRE: nosalas(tipo) ≤ NUMSALAS - 1
  BuscarSalaNoLlena (almacen, sala, nosalas)
  POST: Devuelve la sala no llena que contiene más vacas de la clase "tipo".
* Si no existe ninguna sala con vacas de la clase "tipo",
* se devuelve cualquier sala vacía siempre que el número de salas asignadas a
* las vacas de la clase "tipo" no supere el tope NUMSALAS-1. Si tampoco
* existe una sala vacía que satisfaga el requisito anterior, se devuelve
* cero.
* POST:

```

```

*      resultado = {
0  (∀i ∈ TipoNumSala.
    almacen(i).tipo = tipo → almacen(i).npiezas ≠ TAMSALA) ∧
    (∃i ∈ TipoNumSala.
    almacen(i).tipo = tipo → nosalas(tipo) = NUMSALAS -1)
i  ∃i ∈ TipoNumSala.
    almacen(i).npiezas = 0 ∧
    ∀j ∈ TipoNumSala.
    almacen(j).tipo = tipo → almacen(j).npiezas = TAMSALA
maxim i ∈ TipoNumSala |
    almacen(i).tipo = tipo ∧ almacen(i).npiezas < TAMSALA .
    almacen(i).npiezas
e.o.c.
**)

```

```

:
END BuscarSalaNoLlena;

```

```

PROCEDURE BuscarSalaOcupada (tipo: TipoVaca): CARDINAL;
(**
* PRE: cierto
* BuscarSalaOcupada (almacen, sala)
* POST: Devuelve la sala que contenga un menor numero de vacas de la clase
*       "tipo". Si no existe ninguna sala con vacas de la clase "tipo",
*       se devuelve cero.
* POSTresultado = {
0  ∀i ∈ TipoNumSala.almacen(i).npiezas ≠ 0 → almacen(i).tipo ≠ tipo
minim i ∈ TipoNumSala|almacen(i).tipo = tipo . almacen(i).npiezas
e.o.c.
**)

```

```

:
END BuscarSalaOcupada;

```

```

PROCEDURE ElOtro(tipo: TipoVaca):TipoVaca;
(**
* PRE: cierto
* ElOtro (tipo)
* POST: resultado = {
normal  tipo = loca
loca    tipo = normal
**)

```

```

:
END ElOtro;

```

```

PROCEDURE Dejar (tipo : TipoVaca; sala:TipoNumSala); (**
* PRE: almacen(sala).npiezas < TAMSALA
* Dejar (tipo, sala, almacen, nosalas)
* POSTDeja una vaca en la sala indicada actualizando el almacén y el contador de salas
* POST∀i ∈ TipoNumSala. almacensal(i) = { almacenent(sala) + 1  i = sala
    almacenent(i)      e.o.c.
    ∧ (almacenent(sala) = 0 → (almacensal(sala) = tipo
    ∧ nosalassal(tipo) = nosalasent(tipo) + 1))
**)

```

```

:
END Dejar;

```

```

PROCEDURE Retirar (tipo : TipoVaca, sala:TipoNumSala); (**
* PRE: almacen(sala).npiezas > 0
* Dejar (tipo, sala, almacen, nosalas)
* POSTDeja una vaca en la sala indicada actualizando el almacén y el contador de salas
* POST∀i ∈ TipoNumSala. almacensal(i) = { almacenent(sala) - 1  i = sala
    almacenent(i)      e.o.c.
    ∧ (almacenent(sala) = 1) → nosalassal(tipo) = nosalasent(tipo) - 1)
**)

```

```

:
END Retirar;

```

```

PROCEDURE DejarNormal (VAR sala: TipoNumSala);

```

```

BEGIN
(* Sincronización de entrada *)
sala := BuscarSalaNoLlena(normal);
IF sala = 0 THEN
    INC(nodejaesperando[normal]);
    Delay(dejaesperando[normal]);
    DEC(nodejaesperando[normal]);
    sala := saladisponible;
END;

(* Actualización del almacen *)
Dejar (normal, sala);
(* Sincronización de salida *)
saladisponible := sala;
IF nodejaesperando[normal] > 0 AND (almacen[sala].npiezas < TAMSALA) THEN
    Continue(dejaesperando[normal]); (* despertar en cascada *)
ELSE
    Continue(retiraresperando[normal]);
END; END DejarNormal;

PROCEDURE DejarLoca (VAR sala: TipoNumSala);
BEGIN
(* Sincronización de entrada *)
sala := BuscarSalaNoLlena(loca);
IF sala = 0 THEN
    INC(nodejaesperando[loca]);
    Delay(dejaesperando[loca]);
    DEC(nodejaesperando[loca]);
    sala := saladisponible;
END;

(* Actualización del almacen *)
Dejar (loca, sala);

(* Sincronización de salida *)
saladisponible := sala;
IF nodejaesperando[loca] > 0 AND (almacen[sala].npiezas < TAMSALA) THEN
    Continue(dejaesperando[loca]); (* despertar en cascada *)
ELSE
    Continue(retiraresperando[loca]);
END;
END DejarNormal;

PROCEDURE RetirarNormal (VAR sala: TipoNumSala);
BEGIN
(* Sincronización de entrada *)
sala := BuscarSalaOcupada(normal);
IF sala = 0 THEN
    (* No es necesario notificar el bloqueo *)
    Delay(retiraresperando[normal]);
    sala := saladisponible;
END;

(* Actualización del almacen *)
Retirar (normal, sala);

(* Sincronización de salida *)
saladisponible := sala;
IF nodejaesperando[loca] > 0 AND (almacen[sala].npiezas = 0) AND
(nosalas[loca] < NUMSALAS-2) THEN
    Continue(dejaesperando[loca]);
ELSE
    Continue(dejaesperando[normal]);
END;
END RetirarNormal;

PROCEDURE RetirarLoca (VAR sala: TipoNumSala);
BEGIN
(* Sincronización de entrada *)

```

```

sala := BuscarSalaOcupada(loca);
IF sala = 0 THEN
  (* No es necesario notificar el bloqueo *)
  Delay(retiraresperando[loca]);
  sala := saladisponible;
END;

(* Actualización del almacén *)
Retirar (loca, sala);

(* Sincronización de salida *)
saladisponible := sala;
IF nodejaresperando[normal] > 0 AND (almacen[sala].npiezas = 0) AND
  (nosalas[normal] < NUMSALAS-2) THEN
  Continue(dejaresperando[normal]);
ELSE
  Continue(dejaresperando[loca]);
END;
END RetirarLoca;

BEGIN
  (* El almacén está inicialmente vacío *)
  FOR i := 1 TO NUMSALAS DO
    almacen[i].npiezas := 0
  END ;

  FOR j := normal TO loca DO
    nodejaresperando[j] := 0; (* No hay operaciones bloqueadas *)
    nosalas[j] := 0;
  END;
END (* MONITOR TipoGestorAlmacen *);

```

3 Preguntas cortas

Las respuestas a esta pregunta deben entregarse en esta misma hoja del enunciado.

Nombre:

Número de matrícula: Grupo:

Cuestión 1

Tras un análisis de una aplicación concurrente se diseña un sistema con dos procesos A y B y un recurso compartido C . Cuando se traduce a paso de mensajes se añade un proceso `Gestor.C` para controlar el recurso C . Este proceso incluye una sentencia `SELECT` ¿Qué procesos pueden enviar mensajes a un canal `c` que aparece en la recepción de una rama de dicha `SELECT`?

- ☐ A , B y C
- ☐ O bien A o bien B , según se haga el `GetChannel (c)`.
- ☒ A y B indistintamente.

Cuestión 2

Complétese la siguiente tabla que equipara los diferentes elementos de la implementación de un recurso mediante monitores y paso de mensajes:

Monitores	Paso de mensajes
Nombre de operaciones del monitor	Canal para petición de ejecución de operaciones
Entrada a monitor en exclusión mutua	Verificación de la guarda booleana tras un <code>WHEN</code> y recepción del mensaje correspondiente a la operación
Llamada a una operación del monitor para la que se usa <code>canal_op</code> suministrando <code>datos_entrada</code> y recibiendo <code>datos_salida</code>	<code>Send (canal_op, datos_entrada);</code> <code>Receive (canal_retorno, resultados)</code>
Variables del monitor accesibles en exclusión mutua por todos los procesos	Variables locales del proceso que implementa el recurso activo
Precondición de una operación que no se cumple y que provoca un <code>Delay</code>	Condición de rama <code>WHEN</code> que no se cumple
<code>Continue (variable_condition)</code>	Actualización de variables de manera que en la siguiente ejecución de la <code>SELECT</code> se verifica la condición de una de sus ramas
Cuerpo dentro de una operación entre la comprobación de precondiciones y el código que realiza los <code>Continue</code> necesarios	Cuerpo dentro de una rama de la <code>SELECT WHEN</code> <code>cond, Receive (canal, d) DO</code> <code><< cuerpo >></code>

Cuestión 3

Estúdiase el siguiente programa concurrente con paso de mensajes, y descríbase en el recuadro inferior vacío las situaciones (esto es, los valores admisibles para v_1, v_2, v_3, v_4) en las que no se producirán errores, ni interbloqueos, ni inanición en el programa, y el programa terminará correctamente. **Nota:** la rama **ELSE** de la **SELECT** se ejecuta cuando todas las expresiones booleanas de las guardas son falsas, independientemente de que haya o no mensajes encolados en los canales por los que reciben los **Receive**.

```
MODULE Programa;
TYPE T = [1..3];
VAR c1, c2 : CHANNEL (* OF T *);
```

```
TASK P1;
VAR a : T;
BEGIN
  a := v1;
  IF NOT ODD (a) THEN DEC (a) END;
  IF a > 2 THEN
    Send (c1, a)
  END;
END P1;
```

```
TASK P2;
VAR a : T;
BEGIN
  a := v2;
  Send (c2, a);
END P2;
```

```
TASK G_R;
VAR a, b : T;
VAR i, c : INTEGER;
BEGIN
  GetChannel(c1);
  GetChannel(c2);
  a := v3;
  c := 0;
  FOR i := 1 TO 2 DO
    SELECT
      WHEN a < 2, Receive (c1, b) DO
        c := c + b;
      WHEN a > 2, Receive (c2, b) DO
        c := c + b;
      ELSE
        c := b;
    END;
    a := v4;
  END
END G_R;
```

```
BEGIN
  COBEGIN
    P1; P2; G_R
  COEND;
END Programa;
```

Valores admisibles:

$v_i \in \{1..3\} \wedge$	<i>por declaración de tipos</i>
$((v_1 = 3 \wedge (v_3, v_4) \in \{(1, 3), (3, 1)\}) \vee$	<i>cuando P1 envía dato</i>
$(v_1 \neq 3 \wedge (v_3, v_4) \in \{(2, 3), (3, 2)\}))$	<i>cuando P1 no envía ningún dato</i>

Si se considera que puede haber un error de ejecución en el caso de que se realice la asignación $c := b$ en el proceso gestor antes de la inicialización de b, la última línea debe ser

$$v_1 \neq 3 \wedge v_3 = 3 \wedge v_4 = 2$$

que fuerza a que se ejecute la recepción por el canal c2 antes que la rama **ELSE** de la **SELECT**.