Ejercicios de Programación Concurrente

Dpto. LSIIS. Unidad de Programación **2006-03-03**

Índice

1. Avisador de correo

Un *avisador de correo* es una pequeña aplicación que (generalmente en un entorno de ventanas) avisa al usuario de si ha recibido correo electrónico. La figura muestra el aspecto de la ventanita del avisador, en los dos estados posibles: sin o con correo.

Figura 1: El avisador en sus dos estados posibles

Lo que tiene que hacer el programa es, más o menos, lo siguiente:

- Al empezar, si el fichero de correo no está vacío se muestra el icono de correo; en caso contrario el de sin correo.
- Cada 30 segundos (aproximadamente: el tiempo no es crítico aquí) se mira si ha variado el tamaño del fichero de correo: si ha crecido desde la última vez ha de pintarse el icono de correo; si ha decrecido se asume que el usuario está leyendo ya los mensajes con algún programa y se pintará el icono de sin correo.
- El usuario puede quitar el aviso de correo en cualquier momento haciendo *clic* con el ratón en el icono.

Se asumen como ya programados los siguientes procedimientos:

```
procedure Fichero_Correo() return TipoFichero;
-- Devuelve el fichero donde se almacena el correo del usuario
-- que lo ejecuta.

procedure Tam(f: in TipoFichero) return Natural;
-- Devuelve el tamaño del fichero que se le pasa como argumento.

procedure Esperar(seg : in Natural);
-- No retorna hasta que han pasado seg! segundos.

procedure Leer_Clic();
-- No retorna hasta que el usuario hace clic con el ratón en la
-- ventana de la aplicación.

procedure Pintar_Icono_Correo();
-- Pinta el icono de que ha llegado correo en la ventana
-- de la aplicación.

procedure Pintar_Icono_Sin_Correo();
-- Pinta el icono de que no hay correo en la
-- ventana de la aplicación.
```

Las operaciones de pintado no se pueden ejecutar de manera concurrente.

2. Control de las puertas de entrada a un aparcamiento

El esquema de la figura 2 corresponde a un aparcamiento gratuito con capacidad para N coches, cuyo acceso se realiza a través de E barreras automáticas de entrada y S de salida. Dichas barreras están numeradas, desde la barrera 0 hasta la $E-1$ son de entrada y desde la barrera E hasta la $E+S-1$ son de salida.

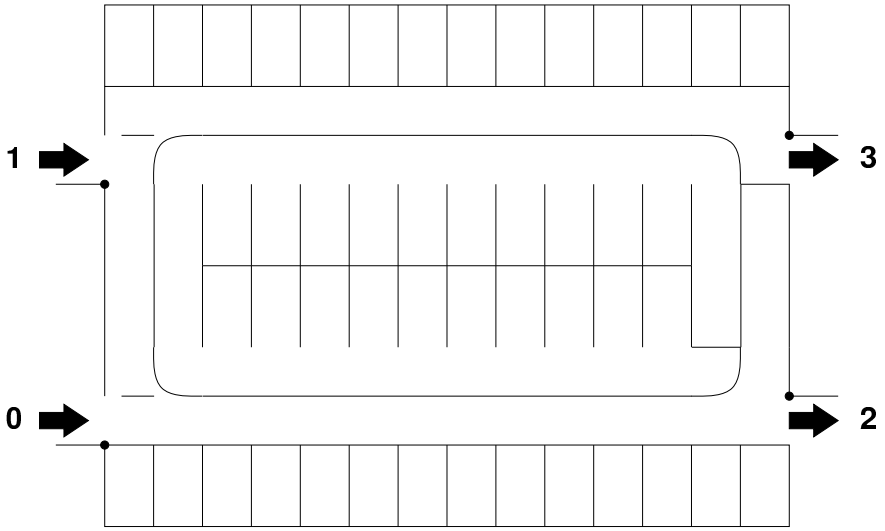


Figura 2: Aparcamiento con dos entradas y dos salidas

Para poder controlar el acceso al área común de carga y descarga un equipo ha programado el paquete `Barreras` en el que están definidas las constantes `E` y `S`, así como el tipo

```
type Tipo_Num_Puerta is Natural range 0..E+S-1
```

y los procedimientos

```
Esperar_Llegada(i: in Tipo_Num_Puerta)

Elevar_Barrera(i: in Tipo_Num_Puerta)
```

con la siguiente semántica:

- Cuando el procedimiento `Esperar_Llegada(i)` es ejecutado por un proceso, este queda bloqueado hasta que se detecta la llegada de un coche a la puerta `i`.
- Al ejecutar `Elevar_Barrera(i)` la barrera se eleva, permanece elevada mientras está pasando el coche y luego desciende automáticamente, momento en el que termina la llamada.

Se necesita desarrollar un programa que gestione el acceso al aparcamiento cumpliendo los siguientes requisitos:

- Todas las barreras deben poder funcionar simultaneamente.

- Sólo se permitirá la entrada de coches si hay plazas libres.
- Si el aparcamiento se llena, y se detectan coches esperando en más de una entrada, se irá dando paso sucesivamente a los coches de cada entrada, es decir, se dará paso a la entrada n, después a la $n+1$, etc.

NOTA: Se trata de desarrollar el programa real de control, no de simular el sistema. No existirán, por tanto, procesos dedicados a simular el comportamiento de los coches. La interacción del programa de control con su entorno se realiza exclusivamente mediante los procedimientos `Esperar_Llegada` y `Elevar_Barrera`.

3. Máquina expendedora

El objetivo de este ejercicio es diseñar un sistema concurrente para controlar una máquina genérica expendedora de productos (hasta 16 numerados de 0 al 15). Recordemos parte de la funcionalidad de una de estas máquinas:

- Inicialmente el saldo para obtener un producto es 0.
- Cuando se introduce una moneda: si **hay cambio** se actualiza el saldo y se visualiza en el display, si **no hay cambio** se devuelve la moneda.
- Cuando se pulsa el botón de devolución hay que devolver la cantidad del saldo y actualizarlo a 0.
- Cuando se selecciona un producto, si el saldo **es suficiente** se sirve el producto y se devuelve la cantidad restante dejando el saldo de nuevo a 0, si el saldo **no es suficiente** se informa en el display de la cantidad que falta y la máquina olvida la selección.

Para desarrollar dicho software se dispone de un paquete `Maquina` ya implementado para controlar los dispositivos de la máquina.

```
type Tipo_Producto is Natural range 0..15;

procedure Detectar_Moneda(Valor: out Integer);
-- Detectar_Moneda(Valor) provoca que el proceso que lo invoca
-- quede bloqueado hasta que un usuario introduzca una moneda
-- (Valor es su valor)

function Hay_Cambio return Boolean;
-- Hay_Cambio <=> no hay problemas para devolver dinero
-- a los usuarios

procedure Detectar_Devolucion;
-- Detectar_Devolucion provoca que el proceso que lo invoca quede
-- bloqueado hasta que el usuario pulse el botón de devolución

procedure Devolver(Cantidad: in Integer);
-- Devolver(c) provoca la devolución de una cantidad c de dinero
-- (la operación resuelve el numero de monedas y el tipo
-- de las mismas).

function Precio(Producto: in Tipo_Producto) return Integer;
```

```
-- Precio(p) es el precio del producto p.

procedure Detectar_Seleccion(producto: out Tipo_Producto);
-- Detectar_Seleccion(p) provoca que el proceso que lo invoca quede
-- bloqueado hasta que un usuario seleccione un producto p.

procedure Servir(producto: in Tipo_Producto)
-- Servir(p) provoca que la máquina sirva una unidad del producto p.

procedure Mostrar(cantidad: in Integer)
-- Mostrar(c) muestra una cantidad c en el display de la máquina.
```

4. Sistema de climatización

Se pretende desarrollar un sistema capaz de controlar la temperatura de un local. Para ello ya se dispone de un *selector* que permite elegir la temperatura deseada en cada momento, un *termómetro* que permite medir la temperatura ambiente y un *aparato climatizador* con el que generar aire caliente o frío.

Los requisitos de funcionamiento del sistema son los siguientes:

1. **La temperatura debe mantenerse dentro de unos márgenes** en torno a la última selección realizada. El rango de temperatura admisible oscila entre U grados por debajo y U grados por encima de la última selección.
2. Por otra parte, una nueva selección de temperatura **deberá anular inmediatamente a la anterior**, es decir, es necesario ocuparse de la nueva selección sin retrasos.
3. No podemos asumir nada sobre la velocidad de selección de temperaturas distintas (puede ser **muy rápida** y es necesario atender dicha selección sin esperas).
4. Para conseguir que la temperatura ambiente sea la deseada (dentro del rango antes mencionado) el sistema puede activar o parar la emisión de aire frío o caliente. **No es admisible** una solución que emita aire frío y aire caliente simultáneamente.
5. **No deberán pasar más de P segundos** sin tomar medidas de control cuando la temperatura ambiente se salga de sus márgenes.

Para la interacción con los elementos del entorno (selector, termómetro, etc.) se dispone de las siguientes operaciones **ya implementadas**:

```
procedure Esperar_Seleccion(Temperatura: out Tipo_Temperatura);
-- El proceso que la invoca permanece bloqueado hasta que se
-- selecciona una nueva temperatura con el selector. El parámetro
-- de salida toma el valor de la selección realizada.

procedure Activar_Frio();
-- provoca la emisión de aire frío.  Este procedimiento
-- y los siguientes de control del climatizador pueden tardar
-- un tiempo en ejecutarse (su ejecución no puede ser inmediata
-- puesto que implica la parada y arranque de motores y
-- elementos mecánicos). Además el código de dichas operaciones
-- para el control del climatizador no es reentrante!,
-- lo que quiere decir que podría dar problemas el que distintos
```

```
-- procesos invocaran operaciones de este tipo con simultaneidad.

procedure Parar_Frio();
-- Provoca que cese la emisión de aire frío.

procedure Activar_Calor();
-- Provoca la emisión de aire caliente.

procedure Parar_Calor();
-- Provoca que cese la emisión de aire caliente.

procedure Medir_Temperatura(Temperatura: out TipoTemperatura);
-- Realiza una medición de la temperatura ambiente a través del
-- termómetro. El parámetro de salida contiene dicha lectura.

procedure Retardo(T: in Natural);
-- El proceso que la invoca queda bloqueado durante
-- los segundos indicados en el parámetro t!.
```

5. Editor interactivo

Un editor interactivo de documentos se ha diseñado de manera que el documento en edición se mantiene en memoria. El usuario suministra repetidamente órdenes elementales de edición, mediante el teclado. Las órdenes consisten en una o varias pulsaciones de tecla seguidas. Estas órdenes provocan modificaciones en el documento almacenado en memoria y, posteriormente, se actualiza su presentación en pantalla, a partir de una copia de la parte visible del documento. Esta copia también reside en memoria.

Para permitir en lo posible que un usuario experto teclee a la máxima velocidad de que sea capaz, se establecen los siguientes requisitos:

1. El editor debe estar listo para aceptar en todo momento los caracteres que se vayan tecleando.
2. Las órdenes que se vayan reconociendo deben ejecutarse a la velocidad a la que la máquina sea capaz de hacerlo. La ejecución de una orden debe proseguir hasta completarse, aunque dure algún tiempo.
3. La actualización de la pantalla no se hará mientras queden órdenes pendientes de ejecutar, pero puede realizarse de forma simultánea al tratamiento de las siguientes órdenes, en caso de llegar durante una actualización—recuerda que la pantalla se guarda separada del documento.
4. Asimismo, no se debe actualizar la pantalla si no se han ejecutado órdenes nuevas desde la anterior actualización. El resto del editor permanecerá inactivo a la espera de nuevos comandos.

La actualización no se hará, por tanto, tras cada orden de edición. Esto permite reducir la carga de trabajo de actualización, de manera que si se ejecutan varias órdenes de edición seguidas, la actualización presenta de una vez el resultado final de la serie de órdenes. Por otra parte, el punto 4 exige que la solución no sufra de un problema de espera activa con el consiguiente gasto inútil de CPU.

Para resolver el problema se cuenta con las siguientes operaciones **que no tienes que implementar**:

```
function Leer_Comando() return Tipo_Comando;
-- Se encarga de leer un comando de teclado. Retorna cuando la
-- secuencia de teclas pulsadas se entiende sin ambigüedad
```

```
-- como un comando del editor.

procedure Ejecutar_Comando(Cmd:  in Tipo_Comando;
                           Doc: out Tipo_Documento);
-- Toma un documento y lo modifica de acuerdo con un comando.

procedure Extrae_Pantalla(Doc:   in Tipo_Documento;
                          Pant: out Tipo_Pantalla);
-- Obtiene una copia de la parte visible del documento.

procedure Mostrar_Pantalla(Pant: in Tipo_Pantalla);
-- Muestra en pantalla el documento.

procedure Abrir_Documento(Fich:     Tipo_Fichero;
                           Doc: out Tipo_Documento);
-- Crea un nuevo documento a partir de un fichero de texto.
```

6. Accesos a disco

Los discos de cabezas móviles pueden leer o grabar sectores en el dispositivo físico. Para ello es necesario situar la cabeza sobre la pista correspondiente al sector, acción que dura un tiempo bastante grande, y luego realizar el acceso al sector deseado, con un tiempo considerablemente menor que el anterior. Si el disco es usado en un entorno concurrente, resulta ventajoso realizar las operaciones solicitadas por los procesos de manera que se minimice, en lo posible, el movimiento de la cabeza de unas pistas a otras.

Suponemos que tenemos discos sin esta optimización, a los que se puede acceder mediante el paquete `Disk`, que tiene la siguiente interfaz:

```
package Disk is

type Tipo_Buffer;

procedure Read(Sec: in Natural; Buffer: out Tipo_Buffer);
-- Deja en Buffer el contenido del sector sec del disco disk

procedure Write(Sec: in Natural; Buffer: in Tipo_Buffer);
-- Copia al sector sec del disco disk el contenido de Buffer

private
...
end Disk;
```

Estas operaciones leen o graban un sector cada vez. Los sectores se consideran numerados correlativamente, una pista tras otra. Esto quiere decir que si hay S sectores por pista, la pista en que se encuentra un sector s será el resultado de la división entera de s por S.

El objetivo del programa que tienes que realizar es programar un par de procedimientos

```
procedure Leer(Sec: in Natural; Buffer: out Tipo_Buffer);
procedure Escribir(Sec: in Natural; Buffer: in Tipo_Buffer);
```

que:

1. proporcionen acceso concurrente a las operaciones correspondientes de paquete `Disk` y,
2. optimicen el acceso de varios procesos a un disco, intentando que se cambie de pista lo menos posible.

Esto se consigue leyendo o grabando los sectores que se soliciten, tratando de mantener el movimiento de las cabezas en la misma dirección, mientras sea posible. Para ello no realizará las operaciones en el orden en que se soliciten, sino que en cada momento elegirá con preferencia aquella que corresponda a la misma pista en la que se encuentra la cabeza o a la más próxima, en sentido creciente de número de pista. Si no hay ninguna petición pendiente la pista actual o superiores, se retrocedería a la menor pista en la que hubiera peticiones.

Algunos requisitos de funcionamiento del sistema son los siguientes:

1. **Las operaciones del paquete** `Disk` **no son concurrentes**, es decir, no puede haber dos procesos ejecutando simultáneamente `Disk.Read` o `Disk.Write`.
2. Si se están atendiendo peticiones – escrituras o lecturas – en una pista determinada, no se pasará a atender otra pista **hasta que no queden peticiones pendientes en la pista en curso**.
3. El **orden de atención** a las peticiones **dentro de una misma pista** es irrelevante.
4. La operación de lectura terminará **tras completarse la operación física en el disco**, y recibirse los datos leídos. La operación de escritura se dará por terminada en cuanto se inicia la operación física en el disco, **sin esperar a que se complete**.
5. El programa debe funcionar correctamente para un número indeterminado de procesos lectores y grabadores.

Tras un análisis preliminar del problema, se opta por la estructura de procesos e interacciones mostrada en la figura 3.

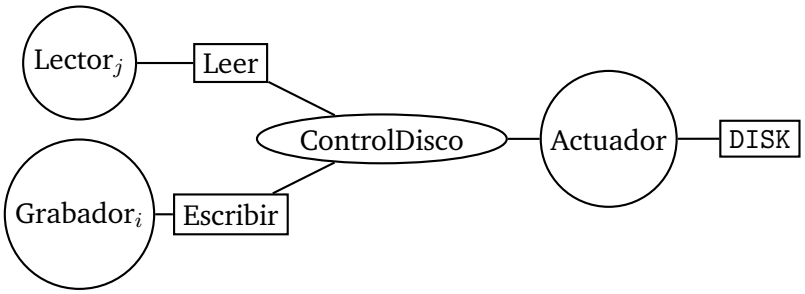


Figura 3: Procesos e interacciones

Es decir, se delega en un proceso `Actuador` el trabajo de llamar a las operaciones del paquete `Disk`. Los procedimientos `Leer` y `Escribir` interactúan con el proceso `Actuador` a través del recurso compartido `ControlDisco`, cuya función es gestionar las peticiones de manera que el actuador las atienda una por una, y en el orden apropiado.

7. Cache de disco

Muchos sistemas operativos mantienen lo que habitualmente se denomina memoria *cache* de disco. Esto consiste en un conjunto de *buffers* de memoria, cada uno del tamaño de un bloque de disco. Cuando algún proceso de usuario quiere acceder a un bloque de disco, el sistema de ficheros —parte del sistema operativo— busca primero si existe una copia de tal bloque en la cache, en cuyo caso devuelve al proceso el buffer correspondiente. En caso contrario, el sistema de ficheros busca un buffer desocupado o, de no haber espacios libres, selecciona el buffer que lleva más tiempo sin ser usado y lo sustituye por el bloque pedido, que habrá sido leído del disco. El buffer reemplazado ha de ser previamente grabado a disco. La utilidad de este sistema a dos niveles es manifiesta: la memoria es mucho más rápida que el disco y el esquema garantiza que los bloques a los que se accede con más frecuencia van a ser leídos, salvo la primera vez, de memoria principal.

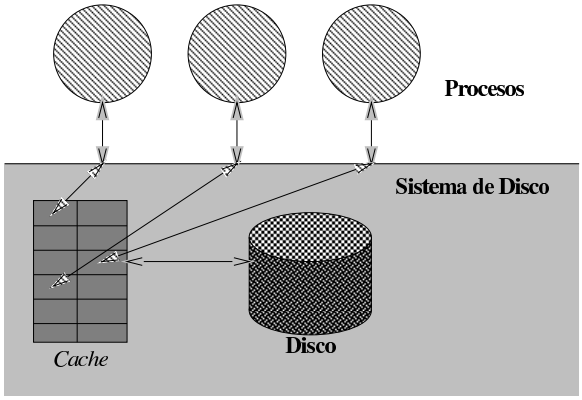


Figura 4: Esquema de sistema de disco y caché

Diseña un sistema concurrente para manejar una cache de bloques de disco, de acuerdo con los siguientes requisitos:

- La copia de bloques de disco en respuesta a una petición deberá ser concurrente con la recepción de otras peticiones.
- Inicialmente —al arrancar el sistema— la cache estará vacía.
- Supóngase, para el control del disco, de la existencia del TAD `Tipo_Disk`:

```
package Disk is

  type Tipo_Disk;

  type Tipo_Buffer;

  procedure Read_Block(Disk:    in  Tipo_Disk;
                       Num_Blk: in  Integer;
                       Buf:     out Tipo_Buffer);

  procedure Write_Block(Disk:    in out Tipo_Disk;
                        Num_Blk: in     Integer;
                        Buf:     in Tipo_Buffer);
```

```
  procedure Init(Disk: out Tipo_Disk);

private
    ...
end Disk;
```

- El tamaño de la cache será de 8 buffers.
- El sistema de disco (`System`) ofrecerá a los procesos los procedimientos

```
Read(Num_Blk: in Integer; Buf: out Tipo_Buffer);
```

y

```
Write(Num_Blk: in Integer; Buf: in Tipo_Buffer);
```

8. Gestión de memoria

En un sistema operativo multiprogramado, un Gestor se encarga de la asignación de páginas de memoria a los distintos procesos de usuario que se encuentran en ejecución en cada momento. Las peticiones de memoria se hacen mediante la ejecución del procedimiento `Solicitar` con cabecera

```
type Tipo_Paginas is Natural range 1..N;

procedure Solicitar(Num: in  Natural;
                    Ind: out Tipo_Paginas);
```

donde `Num` indica el número de páginas solicitadas, `N` es el número total de páginas de memoria y el índice `Ind` la dirección de la primera de las páginas concedidas, que han de ser *tantas como las solicitadas y ocupando posiciones contiguas*.

Los procesos liberan memoria ejecutando el procedimiento `Liberar` con cabecera

```
procedure Liberar(Num: in Natural;  Ind: in Tipo_Paginas);
```

donde `Num` denota el número de páginas liberadas e `Ind` la dirección de la primera de ellas (de nuevo las páginas deben ser *contiguas*).

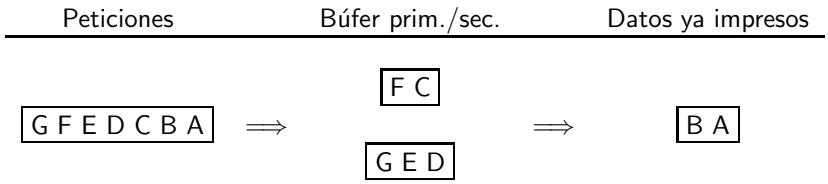
En principio los procesos pueden liberar una cantidad cualquiera de páginas con la única restricción de no liberar más de la que solicitaron, aunque a la hora de ejecutar una simulación se recomienda que ambas cantidades sean coincidentes. El ejercicio propuesto es diseñar e implementar dicho gestor, teniendo en cuenta la posibilidad de accesos concurrentes.

9. Doble *spooler* de impresora

Se trata de organizar un *spooler* de impresora con doble sistema de búfer. Existirá un búfer primario rápido en memoria, con capacidad limitada, y un búfer secundario en disco, de capacidad prácticamente ilimitada pero mucho más lento. Cuando un proceso usuario solicite una operación de escritura los datos a escribir se copiarán a un búfer, y posteriormente un proceso servidor de impresión irá tomando e imprimiendo los datos de los búfers.

Diseña un sistema concurrente que cumpla los siguientes criterios:

1. Cuando se pida una operación de escritura los datos a escribir se copiarán al búfer primario, si hay sitio, y si no al secundario.
2. Cuando un dato se copia a un búfer, permanece en él hasta el momento de imprimirlo, es decir, no hay trasiego de datos de un búfer a otro.
3. Los datos han de imprimirse en el orden en que llegaron las peticiones. Al iniciar una acción de impresión hay que determinar de qué búfer hay que tomar el siguiente dato. Obsérvese la situación en el siguiente ejemplo, que muestra cómo los datos pueden estar repartidos entre uno y otro búfer de manera irregular:



4. Cada búfer requiere acceso exclusivo. No pueden hacerse dos operaciones a la vez sobre el mismo búfer.
5. Durante el acceso al búfer lento en disco no debe estar bloqueada la aceptación de peticiones ni el acceso al búfer primario.
6. Comentar las decisiones que se pueden tomar y justificar las soluciones elegidas (basandose en criterios como pueden ser tiempos de espera, acceso simultaneo de dos procesos cada uno a un *buffer*, simplicidad de la solución, etc.).

10. Buffer basculante

En algunos sistemas de *buffering* las llamadas a las operaciones de *Insertar* y *Extraer* se realizan por rachas. En situaciones como ésta resulta obvio que la posibilidad de insertar y extraer datos simultáneamente sobre el mismo *buffer* es bastante deseable.

En general, no es posible asumir que la ejecución simultánea (sin asegurar exclusión mutua) de dos operaciones cualesquiera de un recurso del tipo *TipoCola* vaya a llevarnos a un estado correcto de recurso, por lo tanto los accesos sobre dicho recurso han de ser excluyentes.

¿Que ocurriría si nuestro *buffer* estuviera formado por dos colas independientes? Parece que existe la posibilidad de insertar en una y extraer de la otra de forma simultánea, sólo es necesario establecer un protocolo que asegure que el *buffer*, en su conjunto, respeta una política FIFO. Dicho protocolo consiste en decidir ("bascular") entre una cola u otra en cada momento. Veamos un escenario de ejecución:

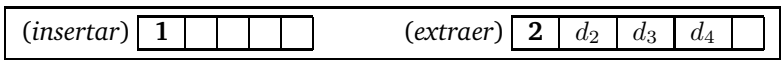
1. Supongamos las dos colas inicialmente vacías, una de ellas etiquetada para *insertar* y otra para *extraer*.

2. Si un proceso quiere extraer datos no puede hacerlo porque no hay ningún dato (y más precisamente porque no hay datos en la cola de extraer).

3. Si un proceso quiere insertar datos tendrá que hacerlo en la cola etiquetada para ello (la cola **1**). Supongamos la inserción de un dato d_1. Además de insertar d_1 en la cola **1** el proceso debería "bascular" las colas, es decir, cambiar las etiquetas de insertar y extraer para posibilitar una extracción y una inserción simultáneamente:

4. Supongamos que llegan dos procesos que quieren insertar datos (d_2 y d_3), tendrán que insertar en la cola etiquetada para ello y lo tendrán que hacer en exclusión mutua:

5. Si ahora un proceso quiere insertar y otro extraer pueden realizar la operación simultáneamente y, finalmente . . . , hay que volver a "bascular":



6. Como puede observarse la situación permite que de nuevo puedan ejecutarse simultáneamente una operación de inserción y una de extracción.

11. Buffer selectivo

Se tiene un sistema con un número indeterminado de procesos productores y consumidores donde la operación de consumir es más lenta que la de producir, y se trata de que los consumidores no retarden la ejecución de los productores. Además, se tienen algunas características particulares:

- Hay dos clases de datos, que llamaremos A y B.
- Hay 2 clases de productores, los que sólo producen datos de clase A y los que sólo producen datos de clase B.
- Hay 3 tipos de consumidores:
  - Los que sólo consumen datos de clase A.
  - Los que sólo consumen datos de clase B.
  - Los que consumen datos de clase A o B indistintamente.
- Los productores generan datos en rachas de 1 a 3, que deben ser almacenados "de golpe".
- Se tiene capacidad para almacenar N_A y N_B datos de clase A o B, respectivamente.
- La lectura de los datos por parte de los consumidores debe hacerse respetando el orden en que han sido depositados por los productores.

Las operaciones de producción y consumo de los datos responden a las siguientes cabeceras:

```
package Buffer_Selectivo is

type Tipo_Datos_A is ...;
type Tipo_Datos_B is ...;
type Array_A is
   record
      Cuantos: Natural;
      Datos: array(1..3) of Tipo_Datos_A;
   end record;
type Array_B is
   record
      Cuantos: Natural;
      Datos: array(1..3) of Tipo_Datos_B;
   end record;


procedure Producir_A (Datos: in Array_A);
procedure Producir_B (Datos: in Array_B);
-- Escribe los datos de tipo A y B que esten en el array

procedure Consumir_A (Dato : in out Array_A);
procedure Consumir_B (Dato : in out Array_B);
-- Consume tantos datos como se haya señalado en el campo ‘‘Cuantos’’

end Buffer_Selectivo;
```

12. Lectores/escritores

Un ejemplo paradigmático de la programación concurrente es el de los lectores/escritores. Muchas veces nos enfrentamos a situaciones en las que un recurso compartido por varios procesos es accedido sólo para leer información del mismo o bien para modificar su/s contenido/s (típica situación en bases de datos). Todos sabemos que varias operaciones de lectura (la estructura no va a sufrir ningún cambio de estado) sobre el recurso compartido pueden realizarse simultaneamente sin efectos no deseados, independientemente de cual sea el entrelazado de operaciones atómicas, sin embargo, en el instante en que se inicia una operación que modifique la estructura ninguna otra operación, ya sea de lectura o de modificación, puede ejecutar simultaneamente con ésta.

Algunos mecanismos de sincronización en memoria compartida tienen dificultades para resolver el problema de los lectores/escritores al asegurar exclusión mutua, esto hace que la solución al problema sea más complicada de lo esperado.

Además es encesario tomar decisiones extra para evitar que no se cumpla la propiedad de ausencia de inanición pero seguir manteniendo un grado de eficiencia aceptable en el acceso al recurso.

Analizar el problema bajo las siguientes restricciones:

- Prioridad para los lectores (esto puede provocar la inanición de los escritores).
- Prioridad para escritores (esto puede provocar la inanición de los lectores).
- Prioridad por estricto orden de invocación de las operaciones.
- Prioridad cambiante mediante el manejo de un turno de prioridad que cambia. ¿Cómo debería cambiar el turno para que la solución mantenga una eficiencia aceptable?

13. Servicio de impresoras

En un centro de cálculo se cuenta con dos tipos distintos de impresoras para dar servicio a los usuarios: impresoras de tipo *A* e impresoras de tipo *B*. Obviamente, el número de impresoras de cada tipo es limitado: `NumA` impresoras de tipo *A* y `NumB` impresoras de tipo *B*, numeradas desde 1 hasta `NumA` o `NumB` según corresponda.

Para imprimir un trabajo en una impresora de tipo *A* es necesario ejecutar la operación con cabecera:[1]

```
type Tipo_Impresora_A is Natural range 1..NumA;

procedure Imprimir_A(Num:     in Tipo_Impresora_A;
                     Trabajo: in TipoTrabajo);
```

mientras que para imprimir en una de tipo *B* se utiliza

```
type Tipo_Impresora_B is Natural range 1..NumB;

procedure Imprimir_B(Num:     in Tipo_Impresora_B;
                     Trabajo: in TipoTrabajo);
```

A la hora de imprimir los trabajos podemos considerar tres grupos distintos de procesos:

1. Los que requieren una impresora de tipo *A*.
2. Los que requieren una impresora de tipo *B*.
3. Los que pueden utilizar una impresora de uno cualquiera de los dos tipos.

La labor de los procesos es generar trabajos («`Generar Trabajo`») e imprimirlos. Como restricción tenemos que dos procesos no pueden ejecutar simultáneamente las operaciones de impresión (`Imprimir_A` o `Imprimir_B`) sobre una misma impresora.

De esta forma cuando un proceso quiera imprimir un trabajo deberá hacerlo sobre una impresora compatible con él y que no esté siendo utilizada por otro proceso. En otro caso el proceso deberá esperar.

14. Impresión tolerante a fallos

Nuestro sistema de impresión está formado por dos impresoras idénticas. El objetivo es programar un servidor, que estará compuesto por un conjunto de tareas, con la interfaz que se muestra en la figura 5. El comportamiento del servidor deberá cumplir las restricciones siguientes:

- Cuando un cliente ejecuta `Imprimir(Trabajo)`, la llamada retorna inmediatamente si hay alguna impresora libre. En caso contrario — las dos en uso o estropeadas — la llamada se bloqueará hasta que cambie la situación.
- Cuando el servidor atiende una petición de `Imprimir(Trabajo)` lo que hace es enviar el trabajo de impresión a la impresora elegida, mediante una llamada a `Enviar_Impresora(Que_Impresora, Trabajo)`[2].

[1] Las operaciones `Imprimir_A` e `Imprimir_B` son operaciones **ya implementadas**.

[2] Los procesos cliente **no** llaman directamente a `EnviarImpresora`

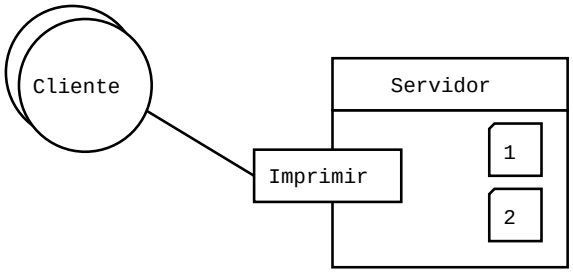


Figura 5: Servidor de impresión

- Un trabajo raramente tarda más de un minuto en imprimirse, por lo que si una llamada a `Enviar_Impresora` tarda más de un minuto en retornar, el servidor considera que dicha impresora está estropeada.
- Un trabajo enviado a una impresora pero no impreso en el tiempo habitual deberá ser enviado a la otra si es que se halla operativa.

Se dispone de los siguientes tipos y procedimientos ya definidos para la realización de este sistema:

```
type Tipo_Impresora is natural range 1..2;

procedure Enviar_Impresora(Que_Impresora: in Tipo_Impresora;
                           Trabajo: in Tipo_Trabajo);

-- Envía un trabajo de impresión a una impresora dada. Retorna cuando
-- el trabajo ha sido impreso con éxito.  Puede no retornar si la
-- impresora se estropea.

procedure Hora_Sistema(Hora: out Tipo_Hora);
-- Proporciona la hora del sistema operativo

procedure Esperar_Hora(Hora: in Tipo_Hora);
-- Se queda bloqueada hasta que se cumpla la hora que se pasa como
-- argumento
```

Se suponen también definidas funciones para trabajar con el tipo `Tipo_Hora` (como `Inc_Minuto` y $>=$).

15. Cintas transportadoras

La nueva terminal de Barajas estará conectada con la actual mediante un par de cintas transportadoras —una en cada sentido— para hacer menos fatigoso el desplazamiento de los viajeros (figura 6). Las cintas poseen un mecanismo de control para que estén en marcha sólo cuando se necesitan. Esto se consigue mediante unos sensores a la entrada de cada cinta que detectan la llegada de usuarios. El criterio seguido es:

- Si una cinta está parada, debe ponerse en marcha al detectarse la presencia de un usuario.
- Si una cinta está en marcha, se para si pasa un minuto sin que llegue nadie.

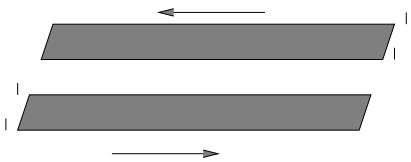


Figura 6: Cintas transportadoras

Para implementar este sistema se dispone de:

- un procedimiento de acceso al sensor

```
Esperar_Sensor(N: in Integer)
-- Se queda esperando hasta que un objeto es detectado
-- por el sensor N!
```

- procedimientos de manejo del motor

```
Start(N: in Integer)

Stop(N: in Integer)
```

  que ponen en marcha y detienen la cinta *N*, respectivamente, y
- un único proceso `Reloj` que cada segundo envía un mensaje a través del canal `CanalReloj`.

Esto constituye toda la interfaz de vuestro programa con el exterior. Además, resulta que si surge algún problema en el motor de una de las cintas, no está garantizado el retorno de los procedimientos textsf Start y `Stop`, requiriéndose que si falla un motor al menos la otra cinta siga funcionando.

Extra: ¿Qué modificaciones habría que realizar en vuestro sistema si ahora queremos que en vez de tener prefijados los sentidos de la marcha, estos sean variables? Se supone que disponemos de sensores en ambos extremos de cada cinta y el requisito adicional sería que las cintas no deben nunca desplazarse en el mismo sentido.

16. Sistema de supervisión y registro

Un sistema distribuido de supervisión y registro de datos se compone de varias unidades de adquisición de datos (UADs) y una unidad central de registro, como se indica en la figura 7.

En cada unidad de adquisición de datos se realiza continuamente un bucle de exploración de un sensor y se anota el valor actual de esa medida. Cuando se detecten ciertos cambios críticos se enviará un registro de alarma a la unidad de registro.

Las unidades de adquisición de datos atenderán también a órdenes periódicas de muestreo desde la unidad de registro, a las que responderán con los valores actuales anotados de las medidas.

Se asume que las unidades de adquisición de datos pueden tomar las medidas a una cadencia mucho más rápida que las solicitudes de muestreo, es decir, pueden repetir varias veces la lectura de los sensores entre dos órdenes de muestreo sucesivas.

La unidad de registro recibirá las alarmas enviadas espontáneamente por las unidades de adquisición de datos, y las almacenará en disco. Periódicamente enviará órdenes de muestreo a esas mismas

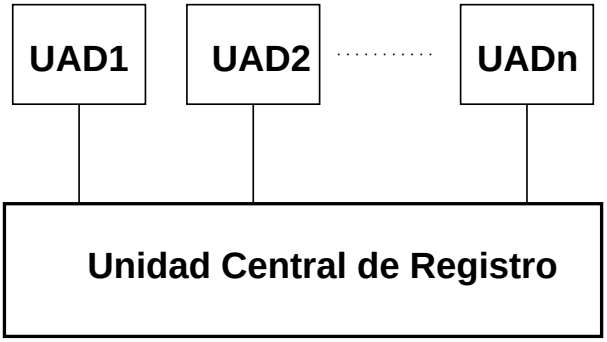


Figura 7: Unidades de adquisición de datos

unidades de adquisición y almacenará en disco las colecciones de medidas recibidas. Sólo se dispone de un disco, por lo que las operaciones de registro en él sólo se podrán hacer de una en una, sean alarmas o medidas originadas por los muestreos. El registro de las alarmas debe tener prioridad sobre el de las medidas. Si el registro de medidas se retrasa, se puede suprimir el registro de las más antiguas. Por ejemplo, la nueva lectura de un sensor reemplaza a la antigua, si no había sido grabada todavía (ie. sólo se guarda una medida por unidad de adquisición).

17. Centralita

La práctica consiste en escribir un programa concurrente que simule el funcionamiento de una centralita con los N teléfonos conectados a ella.

Los teléfonos tienen una interfaz con el usuario y otra con la centralita:

1. El usuario puede realizar las siguientes acciones sobre el teléfono[3]:
   - **Descolgar**: se simulará pulsando la tecla `D`.
   - **Marcar**: se simulará pulsando uno de los dígitos del 0 al 9 del teclado (por tanto, el número N de teléfonos es menor o igual que 10 y cada teléfono tiene asignado un número del 0 a N).
   - **Colgar**: se simulará pulsando la tecla `C`.
2. El teléfono comunicará al usuario (simulándolo a través de la pantalla) la situación en la que se encuentra:
   - **Colgado y en espera**: `Colgado`
   - **Colgado y recibiendo llamada**: `Ring-Ring`.
   - **Descolgado y dando señal**: `Piii...`.
   - **Descolgado y marcando**: `Marcado` x.
   - **Descolgado y llamando al otro extremo**: `Piii-Piii`.
   - **Descolgado y el otro extremo comunica**: `Tuu-Tuu-Tuu`.
   - **Descolgado y conectado al otro extremo**: `Hablando con` x.
3. El teléfono debe comunicar a la centralita que el usuario
   - ha descolgado,

[3]Deberá haber procesos diferentes (incluso más de 1) para cada teléfono.

- ha marcado un número o
- ha colgado.

4. La centralita debe comunicar al teléfono lo que éste posteriormente "muestra" al usuario:

   - hay una llamada entrante (`Ring`),
   - hay señal (`Piii...`),
   - está sonando el otro extremo (`Piii-Piii`),
   - el otro extremo comunica (`Tuu-Tuu-Tuu`) o
   - conectado al otro extremo (`Hablando con x`).

En la figura 17 se puede observar un esquema de las comunicaciones que existen entre el usuario, el teléfono (uno de ellos, el resto son idénticos) y la centralita

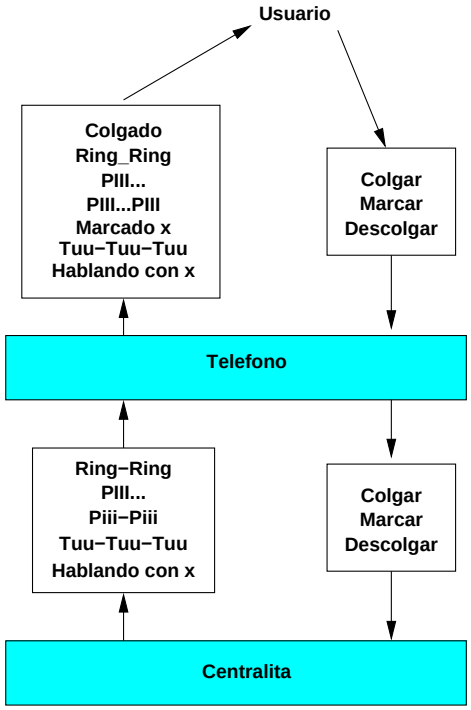


Figura 8: Esquema de comunicaciones

Las especificaciones que debe cumplir la simulación son similares a las del funcionamiento real de un teléfono, no siendo necesario simular el intercambio de información entre los usuarios. Recordemos algunas de las reglas que habrá que respetar:

- Cuando un usuario descuelga el teléfono, éste tiene que comunicarlo a la centralita este hecho y la centralita deberá comunicar al teléfono que hay linea (`Piii...`).
- Cuando un usuario marca después de descolgar, el teléfono debe comunicar a la centralita el número marcado a la vez que se lo muestra al usuario (`Marcado` x); cuando la centralita contacte con el otro teléfono deberá hacerlo sonar (`Ring-Ring`) si está colgado y comunicar al teléfono origen que se está llamando al otro lado (`Piii-Piii`). Si el otro teléfono está descolgado (probablemente alguien está hablando o va a hacerlo) deberá comunicar al teléfono que origina la llamada que el otro lado está comunicando (`Tuu-Tuu-Tuu`).

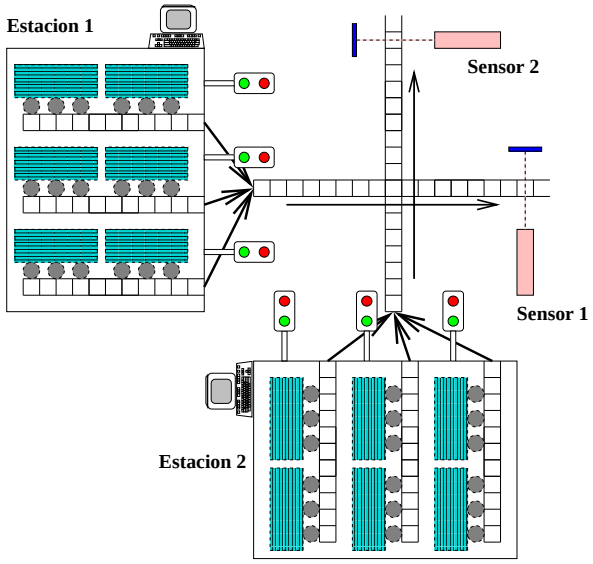
- Cuando se está llamando a un teléfono (`Ring-Ring`) y se descuelga, en ambos lados (el llamante y el llamado) la centralita deberá advertir que están conectados (`Hablando con` x).
- En general, cuando la centralita comunica al teléfono un mensaje, este deberá comunicar al usuario la situación (en nuestro caso basta con visualizar el `Ring-Ring`, el `Piii...`, el `Piii-Piii`, el `Tuu-Tuu-Tuu` o el `Hablando con` x).
- Cuando un usuario cuelga después de haberse establecido el contacto con otro teléfono, este último debe dar señal de comunicando (`Tuu-Tuu-Tuu`) y el primero mostrará su situación (`Colgado`).

Los dos siguientes apartados son opcionales:

- Cuando un usuario descuelga y espera demasiado tiempo (un *time-out*), la centralita debe comunicar al teléfono un `Tuu-Tuu-Tuu` y no hacerle caso hasta que cuelgue.
- Cuando un usuario lleva demasiado tiempo (otro *time-out*) intentando comunicar con otro teléfono (`Piii-Piii`) la centralita debe dejar de intentarlo y comunicar al teléfono el `Tuu-Tuu-Tuu`.

18. Diseño de un controlador para un cruce de trenes

Se pretende diseñar un controlador que regule las salidas de dos estaciones de tren cuyas vías se cruzan. El aspecto de las estaciones y las vías se muestra a continuacion:



En cada estación se cargan trenes con mercancías y se solicita la salida de un tren cuando está lleno. La forma de solicitar la salida es mediante un terminal de ordenador en que se introduce el número de vía en la que está estacionado el tren y qué tipo de tren es (ver más abajo). Un tren arranca cuando su semáforo se pone en verde. Tras salir de la estación el tren entra en el cruce, en el que no debe haber dos trenes simultáneamente. A la salida del cruce hay un sensor en cada vía que detecta cuándo el tren ya ha abandonado el cruce.

Hay trenes de dos tipos: urgentes y normales. Dentro de cada estación, los trenes urgentes tienen siempre preferencia sobre los normales y la salida de estos últimos debe retrasarse hasta que no haya ningún tren urgente esperando. Dentro de la misma categoría de trenes y en una estación dada las peticiones de salida deben ser atendidas en el mismo orden en que fueron realizadas.

Tenemos el siguiente interfaz para leer datos acerca de los sensores de las vías y de las peticiones de salida de trenes, y para notificar la salida de los trenes:

```
Sensor.EsperarPasoTren(NumeroEstacion: TipoNumEst)
(* Se bloquea hasta que se detecte el paso del último vagón de un tren*)
(* por la vía que sale de la estación NumeroEstacion.                 *)
(* Puede consultarse concurrentemente el estado de varios sensores.   *)

Semaforo.DarSalida(NumeroEstacion: TipoNumEst;
                   NumeroVia:      TipoNumVia);
(* Pone a verde el semáforo de la vía NumeroVia en la estación        *)
(* NumeroEstacion y lo vuelve a poner a rojo tras el tiempo necesario *)
(* para que el tren arranque (este tiempo es mucho menor que el que   *)
(* emplea el tren en llegar al cruce).  No pueden realizarse          *)
(* concurrentemente operaciones sobre semáforos de la misma estación. *)
(* Es bloqueante.                                                     *)

Terminal.LeerPeticion(    NumeroEstacion: TipoNumEst;
                      VAR Clase:          TipoTren;
                      VAR NumeroVia:      TipoNumVia);
(* Lee una petición de salida de la estación NumeroEstacion.  La vía  *)
(* en la que está el tren que quiere salir se devuelve en NumeroVia   *)
(* y la clase de tren se devuelve en Clase.  Es bloqueante.           *)
```

Se suponen las definiciones de tipos

```
  TYPE TipoNumEst = [1..2];
       TipoNumVia = 1..MaxVias;
       TipoTren   = (urgente, normal);
```

y se asume que no hay que comprobar la corrección de los datos que llegan del terminal (por ejemplo, nunca se va a decir que hay un tren esperando en una vía cuando en realidad no lo hay) ni preocuparse por eventualidades ajenas al funcionamiento correcto de la estación (por ejemplo, que un tren no pueda arrancar).

Realizar un diseño para el controlador del cruce que garantice que no habrá choques y que respete las prioridades antedichas entre trenes. Se sugiere seguir los siguientes pasos:

1. Dar el esquema de recursos y procesos. Incluir los nombres de las operaciones en el/los recurso(s) y señalar claramente qué operaciones usa cada proceso.

2. Redactar el código de los procesos que aparezcan en el punto anterior. La comunicación con el personal de la estación y con el sensor del cruce debe utilizar únicamente el interfaz expuesto anteriormente.

3. Especificar formalmente con un C-TAD el/los recurso(s) que aparezcan en el primer apartado de esta pregunta.

Aunque el texto del problema habla únicamente de dos estaciones, pensar en el problema como si tuviese n estaciones puede ayudar a tener una solución más elegante.

19. Lonja *Online*

En una subasta de pescado, el que subasta anuncia el lote de pescado que se va a subastar, junto con un precio de salida que será una cota superior del precio que realmente se va a pagar. Tras el anuncio inicial irá cantando precios cada vez menores hasta que alguno de los clientes puje, siéndole adjudicado el lote por el último precio cantado.

La globalización también ha llegado a las subastas de pescado y se está diseñando un sistema para poder participar en ellas a través de Internet. Tras un primer análisis, se han identificado dos tipos de procesos, *Vendedor* y *Comprador*, cuyo comportamiento viene dado por el siguiente pseudocódigo:

```
task body Vendedor is
  Adjudicado : Boolean := False;
begin
  <iniciar Lote>
  Gestor.IniciarSubasta(L.pescado,
                        L.precio);
  loop
    <calcular NuevoPrecio>
    <esperar un cierto tiempo>
    Gestor.ActualizarPrecio(NuevoPrecio,
                            Adjudicado);
    exit when Adjudicado;
  end loop;
end Vendedor;
```

```
task body Comprador is
  Ps : Pescado;
  Pr : Precio;
  Concedido : Boolean;
begin
  Gestor.UnirseASubasta(Hay, Ps, Pr);
  if Hay and then
     <interesa Ps y Pr> then
    loop
      Gestor.OirValor(Pr,
                      NuevoPrecio,
                      Hay);
      exit when not Hay;
      if Hay and then
        <queremos comprar> then
        Gestor.Pujar (Concedido,
                      PrecioFinal)
        exit when True;
      end if;
    end loop;
    if Concedido then
      <comprar lote>
    end if;
  end if;
end Comprador;
```

Como se puede ver, una característica fundamental del sistema es el escaso acoplamiento entre vendedor y compradores: aquél ignora cuántos compradores hay atendiendo a la subasta en un momento determinado y éstos ignoran si la subasta está o no abierta. La interacción con el gestor se ha pensado para que las operaciones proporcionen en todo momento esa información — variables *Hay*, etc. Las operaciones que el gestor ha de proporcionar son, pues, las siguientes:

```
IniciarSubasta (Ps : in Pescado; Pr : in Precio);
-- Usada por el vendedor para notificar la apertura de una subasta

UnirseASubasta (Hay : out Boolean;
                Ps : out Pescado; Pr : Precio);
-- Usada por un comprador para saber si la subasta está activa y,
-- en tal caso, las características de lo subastado

OirValor (PrecioAnterior : in  Precio;
          NuevoPrecio    : out Precio;
          Hay            : out Boolean);
-- Usada por un comprador para conocer el nuevo precio
```

```
-- ‘‘Hay’’ sirve para saber si la subasta sigue abierta

Pujar (Concedido : out Boolean; Pr : out Precio);
-- ‘‘Concedido’’ indica si la puja tuvo éxito y, si es así,
-- ‘‘Pr’’ el precio a que se ha adjudicado el lote

ActualizarPrecio (NuevoPrecio : in Precio;
                  Adjudicado  : out Boolean);
-- La llama el vendedor, y pasa en ‘‘NuevoPrecio’’ el nuevo valor
-- de la oferta. Si algún comprador ha pujado desde la última
-- actualización, ‘‘Adjudicado’’ se hará cierto.
```

Un requisito que debe cumplir el sistema es la ausencia de bucles de espera activa, tanto en el vendedor como en los compradores. En el vendedor esto se consigue mediante una pequeña espera entre actualizaciones consecutivas del precio, pero en los compradores esto no es posible, ya que introducir retardos entre llamadas consecutivas a *OirValor* podría resultar en la pérdida de precios intermedios. Es, por tanto, responsabilidad del gestor el sincronizar las llamadas a *OirValor* para que los compradores no llamen varias veces sin haberse producido variación en el precio del pescado. En otras palabras, la llamada a *OirValor* no debe retornar hasta que no se haya producido una variación de precio con respecto al anterior que el comprador conocía — de ahí la necesidad del parámetro de entrada *PrecioAnterior*.

- Especificar formalmente un CTADSOL *GestorSubasta* que permita realizar la interacción entre vendedor y compradores.
- Implementación en Ada 95 del sistema (procesos *Vendedor*, *Comprador* y gestor) usando procesos distribuidos y *rendez-vous* y/o paso de mensajes.
- Implementación mediante objetos protegidos (sólo el código del tipo protegido).

20. Sistema de retransmisión de vídeo

Un sistema de retransmisión de vídeo en tiempo real (figura 9) consta de un *emisor* que es capaz de recoger imágenes, codificarlas para lograr que ocupen menos espacio, y enviarlas a una serie de *receptores*. Se desea que los receptores tengan una imagen que sea lo más fiel posible al desarrollo temporal de las escenas a retransmitir (por ejemplo, que no acumulen retrasos).

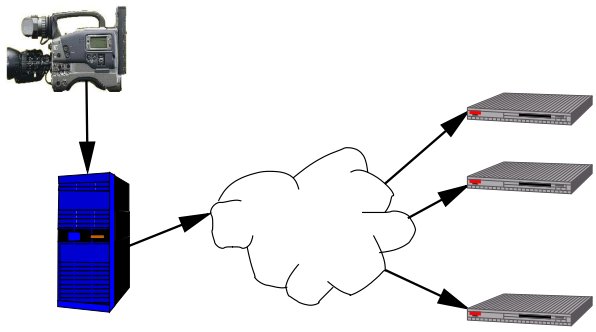


Figura 9: Esquema de sistema de retransmisión

El emisor intenta producir un cierto número de imágenes por segundo. Sin embargo, debido a la tecnología usada para la codificación de imágenes, algunos fotogramas tardan más y otros menos en estar preparados para su difusión. Por lo tanto, el emisor elabora fotogramas a una cadencia no

fija, tardando, en media, R segundos por fotograma. Dichos fotogramas se dejan inmediatamente disponibles para los receptores. Como número orientativo, los sistemas reales de televisión, vídeo o DVD muestran entre 24 y 50 fotogramas por segundo, dependiendo del estándar.

Por otro lado, los receptores pueden ser de distinta calidad y velocidad. Cada receptor puede tardar un tiempo variable en mostrar cada fotograma (sin relación directa con el tiempo empleado por el emisor en codificarlo), y distintos receptores pueden necesitar un tiempo diferente para el mismo fotograma. Los receptores pueden conectarse al sistema en **cualquier momento** (su número, por tanto, **no está fijo** a lo largo de la historia del recurso), y mostrar las mismas imágenes que los demás receptores. Se puede asumir que la retransmisión **no termina** una vez que ha comenzado, y los receptores **no se desconectan** del sistema una vez conectados a él.

De todo lo anterior se desprende que el emisor y los receptores se comportan de forma asíncrona. Sin embargo, se quiere asegurar que la visualización progresa adecuadamente y de forma lo más simultánea posible: **no debe suceder** que un receptor con más potencia de proceso adelante sin límite a otros más lentos. Para conseguirlo, algunos receptores pueden no mostrar algún fotograma (bien porque lo desechen, bien porque directamente no lo reciban) para conseguir alcanzar la simultaneidad necesaria. Si bien la falta de un fotograma representa una disminución en la calidad de la reproducción, ello debe ser causado sólo por la falta de potencia de un reproductor en particular, que no es capaz de alcanzar la velocidad a la que el emisor genera fotogramas. El diseño del sistema **no debe** producir pérdidas innecesarias de fotogramas.

Es deseable desacoplar en lo posible las velocidades del emisor y los receptores, para intentar que, aun en el caso de que el emisor tarde más de R unidades de tiempo en codificar una imagen determinada, los receptores no perciban este retraso. Asimismo debe intentarse reducir en lo posible el gasto de memoria del sistema, eliminando del mismo fotogramas en cuanto estos no sean ya necesarios y evitando la duplicación de la información (un segundo de película en formato MPEG, como el usado en DVDs, que utiliza con algoritmos de compresión y predicción del movimiento muy elaborados, necesita alrededor de 0.6 Mb de almacenamiento).

Las únicas operaciones disponibles en las interfaces del servidor y los receptores son:

```
procedure Mostrar_Imagen(I: in Tipo_Imagen);
-- Muestra en el receptor llamante la imagen I.  Tarda un tiempo
-- en ejecutarse, que varía dependiendo de la imagen I.

procedure Codificar_Imagen(I: out Tipo_Imagen);
-- Accede a la cámara y devuelve en I el resultado de codificarla y comprimirla.
-- Tarda un tiempo en ejecutarse, que varía dependiendo de la imagen I.
```

Adicionalmente se dispone de una operación de bloqueo durante un tiempo dado, que puede utilizarse en cualquier proceso del sistema:

```
procedure Esperar(T: in Tipo_Tiempo);
-- Espera (bloqueando el proceso) durante T unidades de tiempo.
```

Supondremos que los valores de `Tipo_Tiempo` pueden asignarse, compararse (con igualdad y desigualdad) y operarse (por ejemplo, para sumar una duración a un punto en el tiempo). De modo similar, las variables de tipo `Tipo_Imagen` puede asignarse entre sí.

Realizar la fase de análisis y diseño de la concurrencia, proporcionando:

- **Grafo** de procesos (incluyendo los de emisor, receptores, y cualesquiera otros que se consideren necesarios) y recursos.
- **Especificación formal** del recurso necesario para la sincronización de los procesos.
- **Seudocódigo** de los procesos en función de la interfaz del recurso especificado.

- **Implementación** en Ada 95 del sistema (procesos *Emisor*, *Receptor*, *Reloj* y recurso *TransmisiónVídeo*) usando procesos distribuidos y *rendez-vous* y/o paso de mensajes.
- **Implementación** del recurso mediante objetos protegidos (sólo el código del tipo protegido).

21. Bolsa en red

Se trata de diseñar un sistema de gestión de compra y venta de títulos bursátiles. Se supone la existencia de usuarios compradores y usuarios vendedores que se conectan a un servidor que gestiona sus ofertas y demandas. Se supone, además, una clasificación de los títulos por su nivel de riesgo (bajo, medio o alto), que se puede calcular mediante la función predefinida `Nivel_Riesgo(Titulo)`.

Cuando un vendedor desea utilizar este sistema para poner a la venta un título, publica un anuncio en un tablón virtual ubicado en el servidor y a continuación queda esperando a que algún comprador le notifique su interés en adquirir el título. Tras realizarse la adquisición, se retirará el anuncio del tablón.

Por otro lado, cuando un comprador desee adquirir un título, deberá comunicarle al servidor el nivel máximo de riesgo que se desea asumir. Al realizar esta petición, el comprador le indicará también al servidor qué anuncios ha descartado ya, con el fin de que el servidor no le proporcione un anuncio que ya ha sido consultado. Seguidamente, el comprador examinará el anuncio con la intención de averiguar si le interesa o no (función predefinida `Interesa(Cliente,Titulo)`). Si no le interesa, lo descartará y vuelve a pedir un anuncio *con idéntico riesgo* al servidor. Si, por el contrario, el anuncio le interesa, el comprador notificará su interés al servidor y éste, a su vez, se lo hará saber al vendedor. Si la notificación llega tarde porque otro comprador se le ha adelantado, el comprador pedirá al servidor otro anuncio (de nuevo, con el mismo nivel de riesgo).

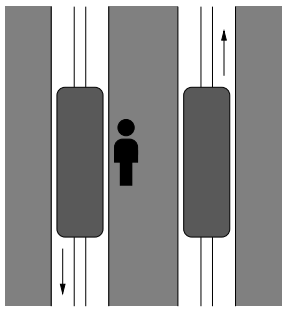
Para simplificar, supondremos que:

- Los anuncios no caducan, sino que son expuestos hasta que se realiza la venta del título.
- El tablón tiene capacidad para un máximo de K anuncios.
- Hay un máximo de V vendedores y C compradores, siendo estos valores *relativamente pequeños*.
- La cuestión del pago por el título es ajena a este sistema.

22. People mover

Se da el nombre de *people mover* a aquellos sistemas de ferrocarril ligero o monorraíl carentes de conductor y que suelen usarse para salvar pequeñas distancias, p.ej. las distintas terminales de un aeropuerto o los edificios de un campus.

El sistema que vamos a considerar consiste en una línea circular con N estaciones y 2 vías, una en cada sentido. Por cada vía viaja un único tren. La figura muestra una vista aérea de una estación.



Existe un andén central, de entrada, por el que se puede acceder indistintamente a los trenes de ambas vías. En la figura, un pasajero se dispone a montar en uno de los trenes. La salida ha de realizarse necesariamente por los andenes exteriores. Para ello los trenes disponen de puertas de entrada y salida que se accionan de manera independiente.

Para ahorrar energía, los trenes se encuentran detenidos en alguna estación hasta que un pasajero se monte o hasta que algún pasajero de otra estación lo solicite mediante el botón de llamada presente en todos los andenes centrales.

Cada tren dispone de un sensor capaz de detectar que se está aproximando a una estación, pero sólo se debe detener si es necesario, es decir, si uno de los pasajeros que viajan en el tren ha pulsado el botón de *parada solicitada* o si se ha llamado al tren desde la estación a la que éste se acerca. La solicitud de parada se desactiva al detenerse el tren y no puede volverse a activar hasta que se reanuda la marcha.

El tren posee un detector de carga que le permite conocer si lleva pasajeros en un momento dado. Esto sirve, entre otras cosas, para detectar que un pasajero ha subido a un tren vacío y detenido en una estación.

Para realizar el control de este sistema se dispone de los siguientes procedimientos **ya programados**:

```
procedure Ocupacion_Vagon(in  T: Tipo_Tren;
                          out P: Boolean);
```

Se encarga de ver si la carga del tren T permite deducir que hay algún pasajero. Esta operación es relativamente lenta (tarda unos segundos).

```
procedure Lectura_Anden(in  E: Tipo_Estacion);
```

Se queda bloqueado hasta que se detecta la llamada desde una estación.

```
procedure Lectura_Parada(in  T: Tipo_Tren);
```

Se queda bloqueado hasta que se detecta la petición de parar en la próxima estación.

```
procedure Detector(in T: Tipo_Tren);
```

Se queda bloqueado hasta que se detecta la proximidad a una estación.

```
procedure Arrancar_Tren(in  T: Tipo_Tren);
procedure Detener_Tren (in  T: Tipo_Tren);
procedure Apertura_Puertas_Salida(in  T: Tipo_Tren);
procedure Cierre_Puertas_Salida  (in  T: Tipo_Tren);
procedure Apertura_Puertas_Entrada(in  T: Tipo_Tren);
procedure Cierre_Puertas_Entrada  (in  T: Tipo_Tren);
```

Actúan sobre diferentes dispositivos del tren. Deben ser llamados de forma no simultánea para un tren dado y en una secuencia coherente con los requisitos planteados anteriormente. Una llamada a una de estas operaciones puede tardar unos segundos en retornar: p.ej., *Cierre_Puertas_Salida(*t*)* incluye todo el proceso de cerrado, con aviso sonoro, pequeña espera y cierre de las puertas en sí. Para simplificar, supondremos que el intento de cerrar unas puertas ya cerradas o abrir unas puertas ya abiertas carece de efecto alguno.

Se pide:

- Grafo de procesos/recursos que permita controlar el sistema arriba descrito.
- Código esquemático de las tareas propuestas.
- Especificación formal (CTADSOL) del recurso o recursos propuestos.
- Una implementación en Ada 95 del **C-TADSOL** utilizando objetos protegidos.
- Una implementación en Ada 95 del **C-TADSOL** utilizando *Rendez-Vous*, con paso de mensajes explícito si se considera necesario.
- La comunicación de la ocupación (o no) de un vagón sólo incumbe a los procesos controlador del tren y al proceso que recoge dicha información. ¿Podría extraerse esa comunicación del recurso presentado y delegarse a otro recurso?

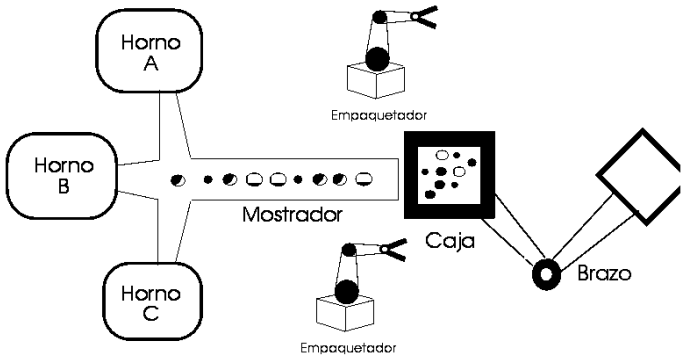
23. Pastas de té

Una fábrica de pastas de té tiene tres hornos que producen tres tipos diferentes de pastas: A, B y C, con pesos diferentes: $pesoA$, $pesoB$ y $pesoC$. Procedentes de los hornos, las pastas se van situando en un *mostrador* común.

Las pastas son empaquetadas en cajas. Uno o varios robots *Empaquetador* toman pastas del mostrador y las introducen en la caja (ver figura). Cada caja puede contener un número diferente de pastas siempre y cuando no se sobrepase un peso límite, denominado $PesoMaximo$. Por este motivo, antes de incluir una pasta en la caja, cada empaquetador debe asegurarse de que con su inclusión no se sobrepasa el peso máximo. Si no se sobrepasa el peso se incluye la pasta en la caja; en otro caso, un *Brazo* mecánico se encarga de retirar la caja que se estaba llenando y posteriormente la sustituye por una caja vacía.

Tened en cuenta de que se trata de llenar la caja lo más posible, lo cual puede ser conseguido por uno cualquiera de los robots que intentan depositar simultáneamente alguna pasta, de pesos que pueden variar. Se considera que no hay interferencia física entre robots que intentan soltar pastas al mismo tiempo en la caja.

Asumiremos que inicialmente hay una caja vacía junto al mostrador.



Para desarrollar el software de control se parte de paquetes Robot y Brazo – ya programados – que proporcionan, entre otras, los siguientes procedimientos de acceso a los dispositivos mecánicos (nos olvidamos del horno; no es asunto del software que debemos diseñar):

```
type Tipo_Empaquetador is Natural range 0..NumEmpaquetadores;

procedure Retirar_Caja;
-- Retirar_Caja hace que el proceso que lo invoca
-- quede bloqueado hasta que la caja que estaba siendo llenada es
-- retirada por el brazo auxiliar.
-- Requiere que haya una caja en la zona de rellenado.

procedure Reponer_Caja;
-- Reponer_Caja hace que el proceso que lo invoca
-- quede bloqueado hasta que el brazo auxiliar coloque una caja
-- vacía en el área de rellenado.
-- No debe haber ninguna caja en la zona de rellenado.

procedure Tomar_Pasta(Empaquetador: in Tipo_Empaquetador;
                      Peso: out TipoPeso);
-- Tomar_Pasta(Empaq, Peso) provoca que el proceso que lo invoca quede
-- bloqueado hasta que el el empaquetador con identificador Empaq
-- toma la pasta más cercana del mostrador, registrando Peso
-- el peso de la misma.

procedure Soltar_Pasta(Empaquetador: in Tipo_Empaquetador);
-- Soltar_Pasta(Empaq) provoca que el proceso que lo invoca quede
-- bloqueado hasta que el el empaquetador con identificador Empaq
-- suelta la pasta que acaba de tomar en la caja del área de
-- rellenado.
-- Es necesario que haya una caja en el área de rellenado, que el
-- robot empaquetador en cuestión tenga una pasta y que la inclusión
-- de esa pasta en la caja no haga sobrepasar el peso máximo permitido.
-- Físicamente, no hay interferencia entre dos robots que intentan
-- depositar pastas simultáneamente en una misma caja.
```

Se pide:

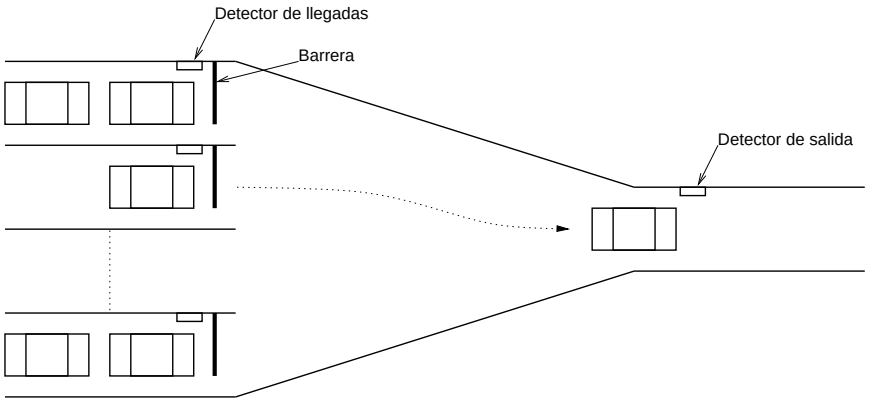
- Grafo de procesos/recursos que permita controlar el sistema arriba descrito.
- Código esquemático de las tareas propuestas.
- Especificación formal (CTADSOL) del recurso o recursos propuestos.
- Una implementación en Ada 95 utilizando objetos protegidos.
- Una implementación en Ada 95 utilizando *Rendez-Vous* y con paso de mensajes explícito si se considera necesario.
- Una variación del diseño en la que se considere que el brazo robot no puede calcular el peso de las pastas, sino que la plataforma en que está la caja tiene una balanza con una operación

  PesoBalanza(Peso: **out** TipoPeso)

  que es **no bloqueante** y de ejecución **prácticamente inmediata**. El nuevo diseño debe cumplir en lo posible los mismos requisitos que el inicial.

24. Zona de estacionamiento

A una zona de estacionamiento para la visita de un importante monumento llegan varios carriles que terminan en barreras que controlan la entrada de los coches. Tras dichas barreras, al aparcamiento se accede por un único carril. La siguiente figura es un esquema del acceso descrito:



El sistema de control de acceso realiza la apertura simultánea de varias barreras y el efecto que provoca es un embotellamiento cuando los coches llegan al final del *embudo*. Para evitar este problema se ha instalado un detector de salida y se va a rediseñar el programa concurrente que controla el acceso de tal forma que:

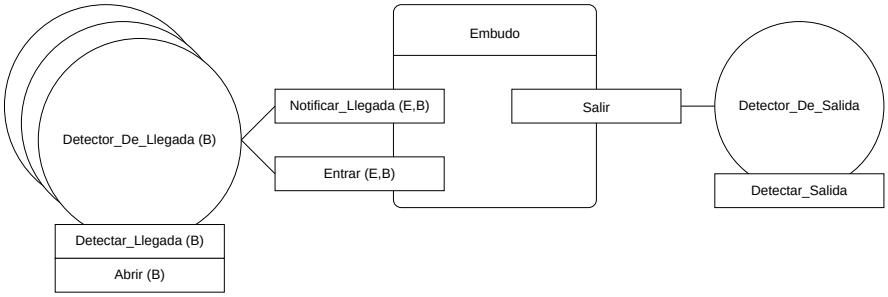
- Nunca haya más de un coche en el embudo.
- En el caso en el que haya varias barreras que puedan ser abiertas, las aperturas deben realizarse en el orden en el que se detectaron los coches.

Para el control de las barreras y de los detectores se dispone del paquete Barreras:

```
package Barreras is
   ------------------------------------------------------------------
   N_Barreras : constant Natural := N;
   ------------------------------------------------------------------
   subtype Barrera is Natural range 0 .. N_Barreras - 1;
   ------------------------------------------------------------------
   -- El proceso que la invoca queda bloqueado hasta que un coche
   -- llega a la barrera B
   procedure Detectar_Llegada (B : in Barrera);
   ------------------------------------------------------------------
   -- Abre la barrera B, espera el paso de un coche y vuelve a cerrarla.
   procedure Abrir (B : in Barrera);
   ------------------------------------------------------------------
   -- El proceso que la invoca queda bloqueado hasta que en la salida
   -- del embudo se detecta el paso de un coche.
   procedure Detectar_Salida;
   ------------------------------------------------------------------
end Barreras;
```

Diseño

A continuación se presenta un diseño en el que hay un proceso controlando cada barrera y otro controlando la salida del embudo. Dichos procesos se comunican mediante un recurso compartido tal y como indica el siguiente grafo de procesos y recursos:



El código de los procesos es el siguiente:

```
-- Recurso compartido
E : Embudo;
```

```
-- Tarea para controlar la salida
-- del embudo
task type Detector_De_Salida;
```

```
task body Detector_De_Salida is
begin
   loop
      Detectar_Salida;
      Salir (E);
   end loop;
end Detector_De_Salida;
```

```
-- Tarea para controlar la barrera B
task type Detector_De_Llegada
          (B : Barrera);
```

```
task body Detector_De_Llegada is
begin
   loop
      Detectar_Llegada (B);
      Notificar_Llegada (E, B);
      Entrar (E, B);
      Abrir (B);
   end loop;
end Detector_De_Llegada;
```

La especificación (incompleta) del recurso compartido es la siguiente:

```
-- CTAD Embudo
-- OPERACIONES
--   ACCION Notificar_Llegada : Embudo[es] × Barrera[e]
--   ACCION Entrar : Embudo[es] × Barrera[e]
--   ACCION Salir : Embudo[es]
-- SEMÁNTICA
--   DOMINIO
--     TIPO Embudo = (Ocupado : Boolean × Llegadas : Secuencia (Barrera))
--     INV: ∀ e ∈ Embudo. ∀ i,j ∈ 1 .. Longitud(e.Llegadas).
--               e.Llegadas(i) = e.Llegadas(j) ⇒ i = j
--     INV (informal): no hay entradas repetidas en la secuencia
--     INICIAL(e): ¬ e.Ocupado ∧ e.Llegadas = ⟨⟩
type Embudo is private;
```

```
-- CPRE: Cierto
-- POST: INICIAL(resultado)
function Crear_Embudo return Embudo;
```

```
-- CPRE: Cierto
-- POST: E^sal.Ocupado = E^ent.Ocupado ∧ E^sal.Llegadas = E^ent.Llegadas + ⟨B⟩
procedure Notificar_Llegada (E : in out Embudo; B : in Barrera);
```

```
-- Especificación de Entrar a completar
procedure Entrar (E : in out Embudo; B : in Barrera);
```

```
-- CPRE: Cierto
```

```
-- POST: ¬ E^sal.Ocupado ∧ E^sal.Llegadas = E^ent.Llegadas
procedure Salir (E : in out Embudo);
```

Completar especificación

Completar la especificación de la operación Entrar.

Objetos protegidos

Completar toda la implementación del recurso compartido Embudo utilizando objetos protegidos como mecanismo de concurrencia, aplicando la metodología propia de la asignatura y siguiendo el esquema que se ofreció anteriormente.

Rendez-Vous / Paso de mensajes

Completar toda la implementación del recurso compartido Embudo utilizando *rendez-vous* y paso de mensajes como mecanismos de concurrencia, aplicando la metodología propia de la asignatura y siguiendo el esquema que se ofreció anteriormente.