Lección 11: Sincronización mediante monitores

- ¿Qué es un monitor?
- Características y funcionamiento de un monitor
- Algunos ejemplos
- Verificación de programas con monitores
- Sobre seguridad y vivacidad
- Lectores-escritores de nuevo
- Lectores-escritores en Ada: varias soluciones
- Implementación de un recurso de acceso por lotes

¿Qué es un monitor?

- Los semáforos tienen algunas características que pueden generar incovenientes:
  - las variables compartidas son globales a todos los procesos
  - las acciones que acceden y modifican dichas variables están diseminadas por los procesos
  - para poder decir algo del estado de las variables compartidas, es necesario mirar todo el código
  - la adición de un nuevo proceso puede requerir verificar que el uso de las variables compartidas es el adecuado

poca "escalabilidad" de los programas

¿Qué es un monitor?

- **E. Dijkstra** [1972]: propuesta de una unidad de programación denominada *secretary* para encapsular datos compartidos, junto con los procedimientos para acceder a ellos.
- **Brinch Hansen** [1973]: propuesta de las *clases compartidas* ("shared class"), una construcción análoga a la anterior.
- El nombre de *monitor* fué acuñado por **C.A.R. Hoare** [1973].
- Posteriormente, Brinch Hansen incorpora los monitores al lenguaje Pascal Concurrente [1975]

¿Qué es un monitor?

- componente *pasivo*
  - frente a un proceso, que es activo
- constituye un ***módulo*** de un programa concurrente
  - proporcionan un mecanismo de abstracción
  - encapsulan la representación de recursos abstractos junto a sus operaciones
    - con las ventajas inherentes a la encapsulación
  - dispone de mecanismos específicos para la sincronización: variables "condición"
  - las operaciones de un monitor se ejecutan***, por definición,*** en exclusión mutua

¿Qué es un monitor?

- Sintaxis:

```
Monitor elMonitor
  variables "permanentes"
  --IM: invariante
  Accion Accion1(pars1)
     ...
  Fin
  ...
  Accion Accionk(parsk)
     ...
  Fin
FMonitor
```

```
Monitor recurso
  Vars ocupado:bool := FALSE
       libre:condicion
  Accion tomar
  Principio
     Si ocupado ent wait(libre) FSi
     ocupado := TRUE
  Fin
  Accion dejar
  Principio
     ocupado := FALSE
     signal(libre)
  Fin
FMonitor
```

Características de un monitor

- Variables permanentes
  - "permanentes" porque existen y mantienen su valor mientras existe el monitor
  - describen el estado del monitor
  - han de ser inicializadas antes de usarse
- Las acciones:
  - son parte de la interfaz, por lo que pueden ser usadas por los procesos para cambiar su estado
  - sólo pueden acceder a las variables permanentes y sus parámetros y variables locales
  - son la única manera posible de cambiar el estado del monitor
- Invocación por un proceso: `nombMonitor.nombAcc(listaParsActs)`

Características de un monitor

- Es un TAD:
  - uso independiente de la implementación
  - el usuario no conoce nada de la implementación
  - nada se sabe del orden en que se van a invocar acciones del monitor
  - por lo que necesita una correcta especificación
- Asociado a un monitor: **invariante del monitor**
  - especifica el estado (cuando ninguna acción se está ejecutando)
  - la inicialización de las variables permanentes debe hacer cierto el invariante
  - invariante a la ejecución de cada acción

Funcionamiento de un monitor

- Respecto a la sincronización:
  - la exclusión mutua se asegura por definición
    - por lo tanto, sólo un proceso puede estar ejecutando acciones de un monitor en un momento dado
    - aunque varios procesos pueden en ese momento ejecutar acciones que nada tengan que ver con el monitor
  - la sincronización condicionada
    - cada proceso puede requerir una sincronización distinta, por lo que hay que programar cada caso
    - para ello, se usarán las variables "***condición***"
      - se usan para hacer esperar a un proceso hasta que determinada condición sobre el estado del monitor se "anuncie"
      - también para despertar a un proceso que estaba esperando por su causa

Sobre las variables condición

- instrucción *wait*:
  - el proceso invocador de la acción que la contiene queda "dormido"
  - y pasa a la cola FIFO asociada a la variable, en espera de ser despertado
- instrucción *signal*:
  - si la cola de la señal está vacía: no pasa nada, y la acción que la ejecuta sigue con su ejecución
    - al terminar, el monitor está disponible para otro proceso
  - si la cola no está vacía:
    - el primer proceso de la cola se despierta (pero no avanza)
    - se saca de la cola
    - el proceso que la ejecuta sigue con su ejecución, hasta terminar el procedimiento

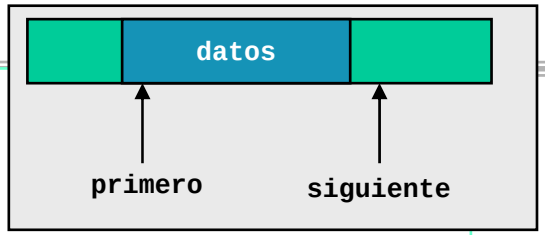
scheduler

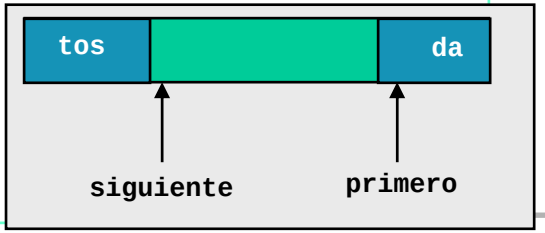
Sobre "wait" y "signal"

- Se parecen a "*wait*" y "*send*" en semáforos, y se comportan parecido, pero:
  - si no hay proceso dormido, "*signal*" no hace nada
    - es como "seguir" (semántica "*signal-and-continue*")
  - "*wait*" en monitor siempre espera a un "*signal*" posterior
  - el proceso que ejecuta el procedimiento "*signal*" de un monitor sigue ejecutando la acción del monitor antes de que el que dormía pueda avanzar
    - hasta que termina o queda dormido

Un ejemplo de monitor

```
Monitor almacenLimitado
  Vars almacen: vector(1..n) de D
       siguiente:Ent :=1; primero: Ent:=1
       nDatos:=0; noLleno,noVacio: cond
  Accion depositar(E dato:D) Principio
    Mq nDatos=n wait(noLleno) FMq        --¿Para qué?
    almacen(siguiente):= dato; nDatos:=nDatos+1
    siguiente:=siguiente mod n +1
    signal(noVacio)
  Fin
  Accion retirar(S v:D) Principio
    Mq nDatos=0 wait(noVacio) FMq        --¿Para qué?
    v:=almacen(primero); nDatos:=nDatos-1
    primero:=primero mod n +1
    signal(noLleno)
  Fin
FMonitor
```





Operaciones en variables condición

- Lista de operaciones sobre variables condición:
  - *wait(c)*: el proceso entra en la cola
  - *wait(c,orden)*: pasa a la cola, que se ordena por orden creciente
    - "*orden*" es entero
    - son despertados en orden creciente a "*orden*"
    - para una señal, usar siempre/nunca prioridades
    - en caso de "empate", FIFO
  - *signal(c)*: despertar el primer proceso de la cola
  - *signal_all(c)*: despertar todos los procesos de la cola
  - *empty(c)*: ¿Está vacía la cola?
  - *minrank(c)*: valor de la prioridad del primer proceso en la cola

Ejemplo: un monitor "el más corto primero"

```
Monitor elMasCortoPrimero
  Vars libre: Booleano := Verdad -- ¿Recurso libre?
       turno: cond    -- avisa cuando recurso libre
                  -- eMCP: (turno ordenado por tiempo) ∧
                  --       (libre → turno es vacio)
  Accion tomar(E duracion: Ent) -- tiempo estimado de uso
  Principio
    Si libre
      libre := Falso
    Si_No
      wait(turno,duracion)
    FSi
  Fin

FMonitor
```

```
Accion dejar
Principio
  Si empty(turno)
    libre := Verdad
  Si_No
    signal(turno)
  FSi
Fin
```

Ejemplo: un monitor para lectores/escritores

```
...
Lector(i:1..m)::
Mq true
  controlaLyE.pideLeer
  leeBaseDatos
  controlaLyE.dejaDeLeer
FMq
Escritor(i:1..n)::
Mq true
  controlaLyE.pideEscribir
  escribeBaseDatos
  controlaLyE.dejaDeEscribir
FMq
```

Ejemplo: un monitor para lectores/escritores

```
Monitor controlaLyE
  Vars nLec: Ent:= 0;  nEsc: Ent :=0
       okLeer,                  -- señala nEsc=0
       okEscribir: cond         -- señala nEsc=0 ∧ nLec=0
                  -- LyE: (nLec=0 ∨ nEsc=0) ∧ (nEsc<=1)
  Accion pideLeer
  Principio
     Mq (nEsc>0) wait(okLeer) FMq
     nLec := nLec+1
  Fin
  Accion dejaDeLeer
  Principio
    nLec := nLec-1
    Si (nLec=0) Ent signal(okEscribir) FSi -- ¿signal(okLeer)?
  Fin
  ...
FMonitor
```

Ejemplo: un monitor para lectores/escritores

```
...
Accion pideEscribir
Principio
  Mq (nLec>0) ∨ (nEsc>0)
    wait(okEscribir)
  FMq
  nEsc := nEsc+1
Fin
Accion dejaDeEscribir
Principio
  nEsc := nEsc-1
  signal(okEscribir)
  signal_all(okLeer)
Fin
...
```

Verificación de programas con monitores

- La corrección de un programa con monitores requiere:
  - probar la corrección de cada monitor
  - probar la corrección de cada proceso aislado
  - probar la corrección de la ejecución concurrente
- Respecto a los monitores:
  - el programador desconoce el orden de ejecución de sus procs/funcs
  - un buen enfoque: buscar un *invariante de monitor*
- Invariante de monitor:
  - cierto cuando un proc/func empieza a ejecutarse
  - cierto cuando termina de ejecutarse
  - cierto cuando se llegue a cualquier *wait*

Verificación de programas con monitores

- Sea ***IM*** una aserción referente a las variables de un monitor
  - ***IM*** se ha de cumplir después de la inicialización:

    {TRUE} inicialización {IM}

    Inicialización: valores iniciales
  - ***IM*** se ha de cumplir antes y después de cada acción:

    {IM ∧ Pre} cuerpoDeLaAcción {IM ∧ Post}
  - Axioma del *wait*:

    {IM ∧ L} *wait*(cond) {IM ∧ L}

    L: referente a parámetros y vars. locales de la acción
  - Axioma del *signal*: **{Q} *signal*(cond) {Q}**

    ¿Algo que decir de las colas asociadas a las señales?

Verificación de programas con monitores

- ¿Qué pasa con *empty, minrank, signal_all*?
- El estado de las colas también es parte del estado del sistema
  - ¿Qué pasa con ellas?
- **Regla de la invocación a una acción**:

```
Accion miAcc(E x:tX;ES y:tY;S z:tZ)
--Pre:  Q(x,y)
--Post: R(x,y,z)
        A
```

Considerar **miAcc(a,b,c)** (b,c distintos))
Sea **I** un predicado que no depende de b,c .
Entonces

$$\{Q^{a,b}_{x,y} \wedge I\}\ \texttt{miAcc(a,b,c)}\ \{R^{a,b,c}_{x,y,z} \wedge I\}$$

- x no es asignable
- actuales distintos

Verificación de programas con monitores

- **Regla de la invocación a una función**:

```
Función f(E x:tX) Dev (z:tZ)
--Pre_f(x)
--Post_f(x,z)
```

$$\frac{Q \rightarrow Pre_f\,{}^{E}_{x}, Post_f\,{}^{E,b}_{x,z} \rightarrow R}{\{Q\}\ b:= f(E)\ \{R\}}$$

```
Si f(E) aparece en una
expresión X,

    e:= X(f(E))

es equivalente a

    TEMP := f(E)
    e := X(TEMP)
```

Un ejemplo de monitor

- Probar la corrección de este monitor, que implementa un semáforo

```
Monitor semaforo
  Vars s: Ent:= K --K>=0
       esPos: cond  -- señala s>0
       nW,nW: Ent := 0 --auxiliares
        -- IM: s>=0 ∧ s=K+nS-nW
  Accion wait
  --Pre: IM ∧ nS=NS∧ nW=NW
  --Post:IM ∧ nS=NS∧ nW=NW+1
  Principio
     Mq (s=0) wait(esPos) FMq
     s := s-1
  Fin
  …
FMonitor
```

```
Accion send
--Pre: IM ∧ nS=NS∧ nW=NW
--Post:IM ∧ nS=NS+1∧ nW=NW

Principio
   s := s+1
   signal(esPos)
Fin
```

Sobre seguridad y vivacidad

- Los monitores se usan de una manera natural para comunicación y sincronización
  - se pueden usar para compartir variables
  - el acceso es, por definición, en exclusión mutua
- **Regla de los monitores**:

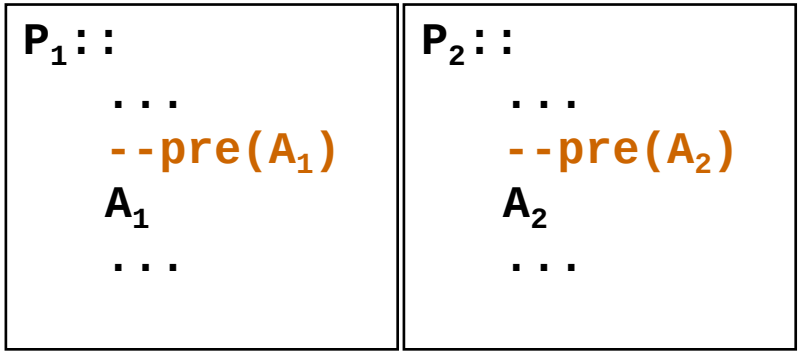
- m monitores
- n procesos
- variables compartidas: en monitores
- interacciones: mediante monitores

$$\forall i \in \{1..n\}.\{Q_i\}\ A_i\ \{R_i\},$$
$$\forall j \in \{1..n\}.j <> i, A_j \text{ no modifica variables de } Q_i, R_i$$
$$\text{las variables de } IM_i \text{ son locales a } m_i$$

$$\{IM_1 \wedge ... \wedge IM_m \wedge Q_1 \wedge ... \wedge Q_n\}$$
$$PCo\ A_1 \,||\, A_2 \,||\, ... \,||\, A_n\ FCo$$
$$\{IM_1 \wedge ... \wedge IM_m \wedge R_1 \wedge ... \wedge R_n\}$$

Sobre seguridad y vivacidad

- **Regla para la exclusión mutua con monitores**



A_1 y A_2 fuera de monitores

$(IM_1 \wedge ... \wedge IM_m \wedge pre(A_1) \wedge pre(A_2)) = Falso$

A_1 y A_2 están en exlusión mutua

Sobre seguridad y vivacidad

- Bloqueo
  - todos los procesos bloqueados o terminados
  - al menos uno bloqueado
- **Regla para la ausencia de bloqueos con monitores**

un monitor no usa otro monitor

$(IM_1 \wedge ... \wedge IM_m \wedge BT \wedge B) = Falso$

---

el programa es libre de bloqueos

- BT: todos los procesos bloqueados o terminados
- B: al menos un proceso bloqueado

Lectores-escritores de nuevo

```
Tipos vectLect=Vector(1..m) De Ent
      vectEscr=Vector(1..n) De Ent
Vars lee: vectLect := (1..m,0)
     escribe: vectEscr := (1..n,0)
     controlaLyE: monitor  --def después

Lector(i:1..m)::
Mq true
   ....
FMq
Escritor(i:1..n)::
Mq true
   ....
FMq
```

Lectores-escritores de r

IM: $AV \wedge LyE$

AV: $nLect=\sum_{\alpha\in\{1..m\}} lee(\alpha) \wedge nEsc= \sum_{\beta\in\{1..n\}} escribe(\beta)$

LyE: $(nLec=0 \vee nEsc=0) \wedge (nEsc<=1)$

```
Lector(i:1..m)::
Mq true
      --lee(i)=0 ∧IM
  controlaLyE.pideLeer(lee,i)
     --lee(i)=1 ∧IM
  leeBaseDatos
  controlaLyE.dejaDeLeer(lee,i)
FMq
Escritor(i:1..n)::
Mq true
      --escribe(i)=0 ∧IM
  controlaLyE.pideEscribir(escribe,i)
     --escribe(i)=1 ∧IM
  escribeBaseDatos
  controlaLyE.dejaDeEscribir(escribe,i)
FMq
```

```
Monitor controlaLyE            --IM: LyE ∧ AV
  Vars nLec: Ent:= 0;  nEsc: Ent :=0
       okLeer,                 -- señala nEsc=0
       okEscribir: cond        -- señala nEsc=0 ∧ nLec=0

  Accion pideLeer(ES lee: vectLect; E i: 1..m)
  --Pre:IM ∧ lee(i)=0
  --Post:IM ∧ lee(i)=1
  Principio
     Mq (nEsc>0) wait(okLeer) FMq
     nLec:=nLec+1;lee(i):=1
  Fin
  Accion dejaDeLeer(ES lee: vectLect; E i: 1..m)
  --Pre: IM ∧ lee(i)=1
  --Post: IM ∧ lee(i)=0
  Principio
    nLec:=nLec-1;
    escribe(i):=0
    Si (nLec=0) signal(okEscribir) FSi
  Fin
```

AV: $nLect=\sum_{\alpha\in\{1..m\}} lee(\alpha) \wedge nEsc= \sum_{\beta\in\{1..n\}} escribe(\beta)$

LyE: $(nLec=0 \vee nEsc=0) \wedge (nEsc<=1)$

Lectores-escritores de nuevo

```
...
Accion pideEscribir(ES escribe: vectEscr; E i: 1..n)
--Pre: IM ∧ escribe(i)=0
--Post: IM ∧ escribe(i)=1
Principio
  Mq (nLec>0) ∨ (nEsc>0)
    wait(okEscribir)
  FMq
  nEsc:=nEsc+1;escribe(i):=1
Fin
Accion dejaDeEscribir(ES escribe: vectEscr; E i: 1..n)
--Pre: IM ∧ escribe(i)=1
--Post: IM ∧ escribe(i)=0
Principio
  nEsc:=Esc-1;escribe(i):=0
  signal(okEscribir)
  signal_all(okLeer)
Fin
```

Lectores-escritores de nuevo

- ¿Se cumplen las especificaciones?
  - si uno escribiendo, nadie más accede
  - puede haber varios leyendo
- ¿Se bloquea la base de datos?

Lectores-escritores en Ada

- Solución fácil: encapsular los datos compartidos en un objeto protegido
- Inconvenientes:
  - normalmente, nos interesa dar preferencia a escritura
    - lo que no es fácilmente implementable en un objeto protegido
  - si la lectura o escritura son potencialmente bloqueantes, no se pueden (deben) hacer desde un objeto protegido
- Solución alternativa 1:
  - usar un objeto protegido para controlar el acceso, pero no la propia lectura o escritura (una especie de "driver")

Lectores-escritores en Ada

```
task type lector(i: rangoLect);
task type escritor(i: rangoEsc);
procedure Read(d: out integer) is ...
procedure Write(d: in integer) is ...
protected control is ...
task body lector is
   d: integer;
begin
   loop
      Read(d);
      put_line("Lector " & Integer'Image(i) & " lee “ &
                Integer'Image(d));
   end loop;
end lector;

task body escritor is
   d: integer;
begin
   loop
      d := i; --lo que sea
      write(d);
      put_line("Escritor " & Integer'Image(i) & " escribe "&
                  Integer'Image(d));
   end loop;
end escritor;
```

```
...
Escritor  1 escribe  1
Escritor  3 escribe  3
Escritor  2 escribe  2
Lector  3 lee  2
Lector  2 lee  2
Lector  1 lee  2
Escritor  1 escribe  1
Lector  4 lee  1
Escritor  4 escribe  4
Escritor  2 escribe  2
Lector  2 lee  2
Escritor  3 escribe  3
Lector  3 lee  3
Lector  1 lee  3
Escritor  1 escribe  1
Escritor  2 escribe  2
...
```

Lectores-escritores en Ada

```
protected control is
   entry Start_Read;
   procedure Stop_Read;
   entry Request_Write;
   entry Start_Write;
   procedure Stop_Write;
private
   Writers: boolean := false; --¿alguien escribiendo?
   Readers: integer := 0;     --cuántos leyendo
end control;
```

Burns and Wellings
pág. 153-

```
procedure Read(d: out integer) is
begin
   control.Start_Read;
   d := laBBDD;
   control.Stop_Read;
end Read;

procedure Write(d: in integer) is
begin
   control.Request_Write;
   control.Start_Write;
   laBBDD := d;
   control.Stop_Write;
end Write;
```

Lectores-escritores en Ada

```
protected body control is
   entry Start_Read when not Writers and Request_Write'Count=0 is
   begin
      Readers := Readers+1;
   end Start_Read;

   procedure Stop_Read is
   begin
      Readers := Readers-1;
   end Stop_Read;

   entry Request_Write when not Writers is
   begin
      Writers := true;
   end Request_Write;


end control;
```

```
   entry Start_Write when Readers=0 is
   begin
      null;
   end Start_Write;


   procedure Stop_Write is
   begin
      Writers := false;
   end Stop_Write;
```

Lectores-escritores en Ada

Burns and Wellings
pág. 155-

- Solución alternativa 2:
  - cuando la operación de escritura es no bloqueante, se puede incorporar dentro del objeto protegido

```
protected control is
   entry Start_Read;
   procedure Stop_Read;
   entry Write(d: in integer);
private
   Readers: integer := 0;  --leyendo
end control;

procedure Read(d: out integer) is
begin
   control.Start_Read;
   d := laBBDD;
   control.Stop_Read;
end Read;

procedure Write(d: in integer) is
begin
   control.Write(d);
end Write;
```

```
protected body control is
   entry Start_Read when Write'Count=0 is
   begin
      Readers := Readers+1;
   end Start_Read;

   procedure Stop_Read is
   begin
      Readers := Readers-1;
   end Stop_Read;

   entry Write(d: in integer) when Readers=0 is
   begin
      laBBDD := d;
   end Write;
end control;
```

Implementación de un recurso de acceso por lotes

```
Monitor recurso
  Vars capMax: Ent:= k
       libres:0..capMax := k
       liberados: cond
            -- señala
            -- liberación
            -- de unidades
  Accion liberar(E n: 1..capMax)
  Principio
     libres := libres + n
     signal_all(liberados)
  Fin
  ...
Fin
```

```
Accion tomar(E n: 1..capMax)
  Vars conseguido: bool :=
                   false
Principio
   Si n<=libres Ent
      libres := libres - n
   Si_No
      Mq no conseguido
         wait(liberados)
         Si n<=libres Ent
           libres := libres - n
           conseguido := true
         FSi
      FMq
   FSi
Fin
```

Implementación Ada de u

```
protected recurso is
   entry tomar(n: natural);
   procedure liberar(n: natural);
private
   entry again(n: natural);
   free: natural := ....;
   waiters: integer := 0; --esperando
end recurso;
```

```
protected body recurso is

end recurso;
```

ver
Barnes
pág. 434-

```
procedure liberar(n: natural) is
begin
   free := free+n;
   waiters := again'Count;
end liberar;

--¿entry tomar(n: natural) when n<=free?
--¿procedure tomar(n: natural)?
entry tomar(n: natural) when true is
begin
   if n>free then
      requeue again;
   else
      free := free-n;
   end if;
end tomar;

entry again(n: natural) when waiters>0 is
begin
   waiters := waiters-1;
   if n>free then
      requeue again;
   else
      free := free-n;
   end if;
end again;
```

Ejercicio

- Un sistema con:
  - un proceso productor de mensajes
  - "l" procesos consumidores de mensajes
  - un *buffer* compartido con capacidad para "n" mensajes
- Tal que:
  - no se pierden mensajes
  - todos los consumidores
    - leen todos los mensajes
    - en orden de llegada

```
Monitor controlaBuffer
   ....
   Accion escribeMensaje(E m: mensaje)
   Accion leeMensaje(E i: 1..l; S m: mensaje)
```

```
Productor::
   Vars m: mensaje
Principio
   Mq TRUE
      preparaMensaje(m)
      controlaBuffer.escribeMensaje(m)
   FMq
Fin
```

```
Consumidor(i:1..l)::
   Vars m: mensaje
Principio
   Mq TRUE
      controlaBuffer.leeMensaje(i,m)
      procesaMensaje(m)
   FMq
Fin
```

Ejercicio

```
Constantes
   l: Natural := .... --# de clientes
   n: Natural := .... --capacidad del buffer

Monitor controlaBuffer   --IM: ........
   Tipos
      infoMens = Registro
         elId:   Entero
         elMens: mensaje
         vecesLeido: Entero
      FReg

      bufMensajes= Registro
         numMens: Natural
         losMens: Vector(1..n) de infoMens
      FReg
   Vars
      buffer: bufMensajes := (0,....)
      sigMensajes= Vector(1..l) de Entero := (1..n,1)
      sigID: Natural := 1
      hayHueco,           --¡se ha creado un hueco en el buffer!
      nuevoMensaje: cond  --¡envío de nuevo mensaje!
```

Ejercicio

```
Accion escribeMensaje(E m: mensaje)
--Pre:
--Post:
Principio
   Mq buffer.numMens=n   --¿Mq?
      wait(hayHueco)
   FMq

   insertaMensaje((sigID,m,0),buffer) --como quiera que se haga
   sigID := sigID+1
   signal_all(nuevoMensaje)
Fin
```

```
Accion leeMensaje(E i: natural, S m: mensaje)
--Pre:
--Post:
    Vars pos: Entero
Principio
   pos := posMensaje(sigMensaje(i)) --0: no ha llegado

   Mq pos=0
      wait(nuevoMensaje)
      pos := posMensaje(sigMensaje(i))
   FMq  --¿iteraré muchas veces?
   sigMensaje(i) := sigMensaje(i)+1
   m := buffer.losMens(pos)
   buffer.losMens(pos).vecesLeido := l+buffer.losMens(pos).vecesLeido
   --incrementar en 1 el campo "vecesLeido"
   --del mensaje leído
   Si buffer.losMens(pos).vecesLeido=l Ent
      borrarMensajeEnPosDelBuffer(pos)
      signal(hayHueco)
   FSi

Fin
```