PROGRAMACION CONCURRENTE Y DISTRIBUIDA

II.5 Sincronización basada en memoria compartida: Monitores

J.M. Drake

Notas:

Monitor

- Son módulos que encierran los recursos o variables compartidas como componentes internos privados y ofrece una interfaz de acceso a ellos que garantiza el régimen de exclusión mutua.
- La declaración de un monitor incluye:
  - Declaración de las constantes, variables, procedimientos y funciones que son **privados** del monitor (solo el monitor tiene visibilidad sobre ellos).
  - Declaración de los procedimientos y funciones que el monitor **exporta** y que constituyen la interfaz a través de las que los procesos acceden al monitor.
  - **Cuerpo del monitor**, constituido por un bloque de código que se ejecuta al ser instanciado o inicializado el monitor. Su finalidad es inicializar las variables y estructuras internas del monitor.
- El monitor garantiza el acceso al código interno en régimen de exclusión mutua. El monitor tiene asociada una lista en la que se incluyen los procesos que al tratar de acceder al monitor son suspendidos.

Notas:

Los monitores constituyen un avance respecto de las regiones críticas condicionales en los que se eliminan algunos de los problemas que estas presentan. Por un lado, en los monitores, se localiza todo el código de acceso a las variables compartidas en un solo bloque del código, y por otro lado, esta localidad va a permitir utilizar implementaciones mas eficientes.

Las **declaraciones de un monitor** incluye la declaración de un conjunto de constantes, tipos, variables, procedimientos, y funciones (quedan excluidos otros monitores o procesos) que son locales al propio monitor (solo desde el monitor se tiene visibilidad sobre los objetos declarados en él).

El monitor incluye una **lista de los procedimientos exportados**, y en consecuencia pueden ser utilizados en el código del programa , de procedimientos, o procesos externos al monitor. Los procedimientos son llamados desde el código externo con el formato:

Identificador_monitor. Identificador_procedimiento(lista de parámetros actuales);

El acceso a los componentes declarados dentro del monitor solo puede hacerse a través de la ejecución de los procedimientos exportados.

El **cuerpo del monitor** es un conjunto de sentencias que se ejecutan al comienzo del bloque en que está declarado (si está declarado en el programa principal) se ejecuta antes de que comience a ejecutarse el propio programa principal. La función del cuerpo del monitor es inicilizar las variables y estructuras internas del monitor.

Lo más característico es que el monitor **garantiza el acceso al código interno en un régimen de exclusión mutua**. Esto implica que el monitor tiene asociada una lista en la que se incluyen todos los procesos que tratan de acceder a procedimientos del monitor y que por el regimen de exclusividad, tienen que esperar a que este quede libre.

Sintaxis del monitor.

```
monitor Identificador;
        -- Lista de procedimientos y funciones exportadas
   export Identificador_procedimiento_1;
   export Identificador_procedimiento_2;
   ....
        -- Declaración de constantes, tipos y variables internas
   ....
        -- Declaración de procedimientos y funciones
   ....
begin
        -- Sentencias de inicialización
....
end;
```


Notas:

Aspectos característicos del monitor.

- Las estructuras de datos internas del monitor cuya finalidad es ser compartidas por múltiples procesos concurrentes, solo pueden ser inicializadas, leídas y actualizadas por código propio del monitor.
- Los únicos componentes del monitor públicos (visibles desde módulos externos) son los procedimientos y funciones exportadas.
- El monitor garantiza el acceso mutuamente exclusivo a los procedimientos y funciones de la interfaz. Si son invocados concurrentemente por varios procesos, solo la ejecución de un procedimientos del monitor es permitido. Los procesos no atendidos son suspendidos hasta que la ejecución del procedimiento atendido acabe.
- Dado que todo el código relativos a un recurso o a una variable compartida está incluido en el módulo del monitor, su mantenimiento es mas fácil y su implementación es mas eficiente.

Notas:

Los **aspectos característicos** de la estructura monitor son:

•Las estructuras de datos compartidas son definidas dentro del monitor, y son inicializadas y gestionadas por componentes definidos dentro del propio monitor.

•Las estructuras de datos no son directamente visibles desde el exterior del monitor, y solo pueden ser accedidas a través de los procedimientos exportados por el propio monitor.

•El acceso mutuamente exclusivo es forzado automáticamente por el código del monitor generado por el compilador. No es posible acceder a los datos fuera del régimen de exclusión mutua. Observese que estructuras que ofrecen otros lenguajes (tal como los paquetes ADA) con información oculta y que solo es accesible salvo por los procedimientos de la interfase, no son equivalentes al monitor por no garantizar a los mismos un acceso en régimen de exclusión mutua.

•Los monitores localiza en su código todos los componentes del programa relativos a una variable compartida. El mantenimiento de los programas basados en monitores es mucho más simple que en los basados en regiones críticas.

Problema del parque público con monitor (1).

```
program Control_acceso_parque;
 const NUM_TORNOS= 10;

 monitor Censo
     export incrementa;
     export print;
     var cuenta: Natural;
     procedure incrementa; begin cuenta:= cuenta +1; end;
     procedure print; begin writeln(cuenta); end;
 begin
    cuenta:=0;
 end;
    ....              -- Continúa
```


Notas:

Problema del parque público con monitor (2).

```
process type Tipo_Tornos;
  var visitante: Natural;
 begin
    for visitante:=1 to 20 do censo.incrementa;
 end;

 var torno: array [1..NUM_TORNOS of Tipo_Torno;
      entrada: 1..NUM_TORNOS;
begin
 cobegin
    for entrada:=1 to NUM_TORNOS do torno[entrada];
 coend;
 censo.print;
end;
```


Notas:

Para imprimir el resultado final hay que acudir a un procedimiento ofrecido por el monitor, ya que a la variable Cuenta que es interna al monitor no es posible acceder directamente desde el código del programa principal.

Variables “Condition”

- El monitor resuelve el acceso seguro a recursos y variables compartidos, pero no tiene las capacidades plenas de sincronización.
- Las variables tipos **condition** que solo pueden declararse dentro de un monitor proporciona a los monitores la capacidad de sincronización plena.
- **condition** es un tipo de variable predefinido que solo puede intanciarse en los monitores

  var inviables: **condition**;

  - **Valores**: No toma ningún tipo de valor, pero tiene asociada una lista de procesos suspendidos
  - **Operaciones**:
    - **delay** suspende el proceso que lo ejecuta y lo incluye en la lista de una variable Condition.
    - **resume** Reactiva un proceso de la lista asociada a una variable Condition.
    - **empty** Función que retorna True si la lista de procesos de la variable está vacía.

Procodis’08: II.5- Monitores José M.Drake 7

Notas:

El monitor, al igual que ocurría con las regiones críticas, es un componente que permite de forma inmediata resolver el problema de las secciones críticas de exclusión mutua, pero por si solas no son suficientes para simular un semáforo y establecer condiciones de sincronización. Para dotar al monitor de esta capacidad es necesario introducir un nuevo tipo de variables predefinidas que se denomina "condition".

Las variables del tipo predefinido **condition** no tienen valores asignados, pero si tienen asociada una cola FIFO de procesos. Tras la inicialización las variables tipo condition tienen su cola vacía.

Ejemplos de declaración de este tipo de variables son:

```
var     C : condition;
        Obstaculos :  array [1..10] of condition;
```

Las variables de este tipo son declaradas dentro de los monitores, y se utilizan para suspender y activar procesos que han accedido al monitor, y que por el estado en que se encuentran, no pueden continuar su ejecución.

Operación delay.

- Suspende el proceso que la ejecuta (que ha invocado el procedimiento del monitor en que se encuentra) y lo introduce en la cola asociada a la variable Condition.

  delay (variableCondition) ;

- Semántica de la operación:
  - Cuando un proceso ejecuta la operación delay, se suspende **incondicionalmente**.
  - Cuando un proceso ejecuta la operación delay, se libera la capacidad de acceso que dispone el proceso.
  - En la cola de una variables Condition pueden encontrarse suspendidos un número ilimitado de procesos.

Notas:

Operación **delay:** que suspende al proceso que la ejecuta y lo introduce en la cola asociada a la variable condition sobre la que se ejecuta. El formato de esta sentencia es:

delay (C);

Características fundamentales de esta operación son:

•Cuando se ejecuta esta sentencia, el proceso que la ejecuta **se suspende incondicionalmente** (al contrario de la operación wait de un semáforo que solo se suspendía si el valor del semáforo era 0).

•Cuando un proceso ejecuta esta operación dentro del código de algún procedimiento del monitor, el proceso no solo se suspende, sino que también **libera la autorización de ejecución en régimen de exclusión mutua del monitor.**

•En la cola asociada a una variable "condition" pueden encontrarse suspendidos un **número ilimitado de procesos.**

Operación resume.

- Ejecutada sobre una variable tipo Condition, se reactiva uno de los procesos de la lista asociada a la variables.

  resume(variableCondition);

- Semántica:
  - Si la cola de la variable está vacía, equivale a una operación null.
  - Esta operación tiene en su definición la inconsistencia de que tras su ejecución existen dos procesos activos dentro del monitor, lo que contradice su principio de de operación. Diferentes modelos del procedimiento resume, que han sido utilizados:
    - **Reactiva y Continua**: El proceso activado permanece suspendido hasta que el proceso que ha invocado resume libera el monitor.
    - **Reactivación_Inmediata**: La ejecución de la operación reasume, supone la liberación del monitor por parte del proceso que la realiza. El único proceso activo dentro del monitor es el proceso que se ha reactivado.
    - **Reactiva y espera**:El proceso que ejecuta resume se bloquea en una cola especial denominada “urgent”, y el proceso activado es el que tiene el acceso sobre el recurso. Cuando el proceso suspendido sobre la cola “urgent” se activa ejecuta la sentencia que sigue a la sentencia resume que la bloqueó.

Notas:

Características fundamentales de esta operación son:

- Cuando se ejecuta la operación resume sobre una variable que tiene su cola asociada vacía, equivale a una operación **null.**
- Un problema potencial que lleva consigo la ejecución de esta operación es que de ella resultan dos procesos activos dentro del monitor, uno el que la ejecuta que obviamente debe estar con acceso al monitor para poder llevarla a cabo, y otro el proceso de la cola asociada a la variable tipo "condition" que se activa. A este problema se le han dado dos soluciones alternativas:

  1. **Reactiva_y_continua**: En este caso se supone que el proceso que se activa permanece aún suspendido hasta que justamente el proceso que ejecuto "resume" concluya el uso del monitor. Esta alternativa fue utilizada por Mitchell (1979) en la implementación del lenguaje concurrente Mesa.
  2. **Reactivación_inmediata**: En este caso el proceso que realiza la operación "resume" abandona inmediatamente del monitor, y deja que el proceso que se ha reactivado, sea el único que permanece de forma exclusiva dentro del monitor. Esta es la variante que mas frecuentemente se ha adoptado, y es la que consideraremos en lo que sigue. En este caso, la sentencia resume **debe ser siempre la última del procedimiento** del monitor en que se encuentra.

De esta última alternativa existen diferentes variantes que han sido utilizadas en diferentes lenguajes: Pascal Concurrente (Brinch Hansen, 1975), Pascal Plus (Welsh y Bustard, 1979). Andrews (1991) demostró todos estos protocolos para la operación "resume" son equivalentes, ya que con cada uno de ellos se pueden simular la operación del otro.

Implementación de un semáforo mediante monitores

```
monitor Semaforo;
    export wait, signal, initial;
     var value: Natural;
         bloqueados: condition;

    procedure initial(v:Natural); begin value:=v; end;

    procedure wait;
    begin
     if (value=0) then
          delay(bloqueados);
     value:= value-1;
    end;

    procedure Signal;
    begin
      value:= value+1;
       resume(Bloqueados);
    end;
begin
  value:=1;    (* Por defecto semáforo abierto *)
end;
```


Notas:

Los semáforos se consideran los componentes básicos de sincronización para programas concurrentes. Comprobar que los monitores son capaces de implementar semáforos representa comprobar que los monitores resuelven todos los problemas requeridos en programación concurrente.

Productor Consumidor con buffer limitado.

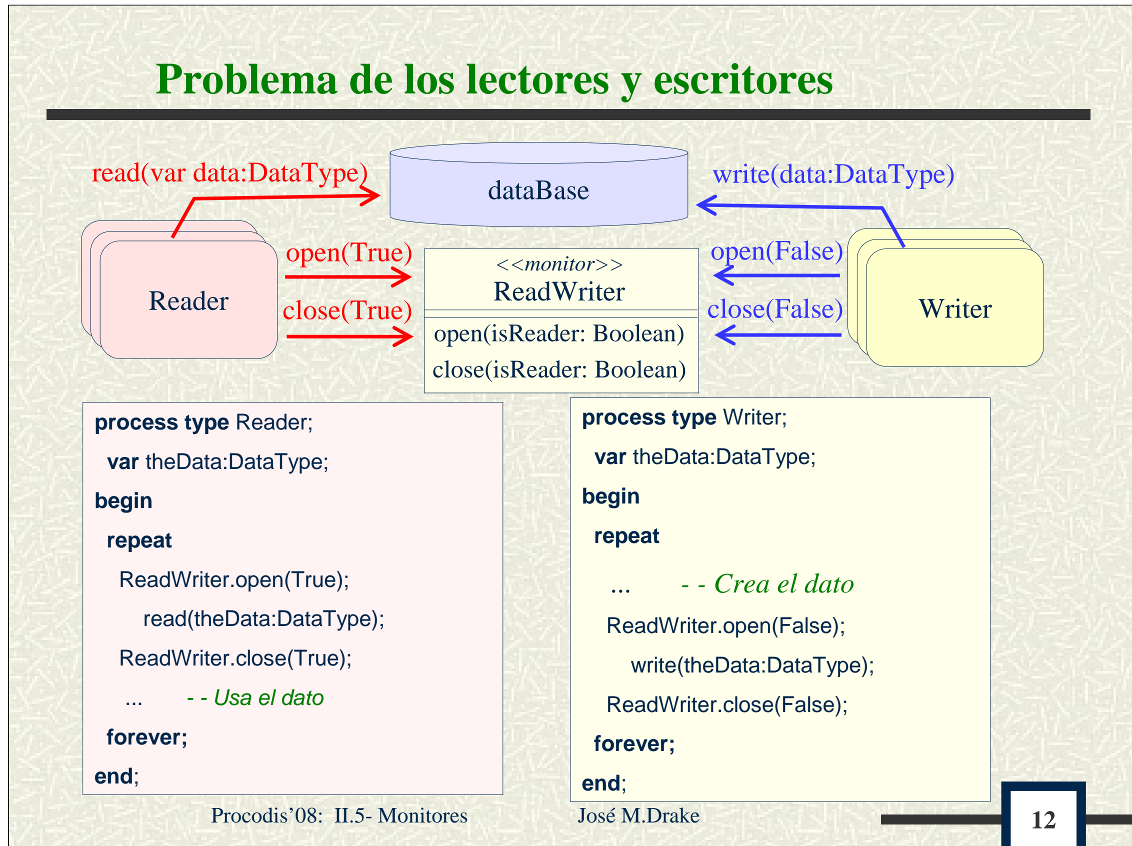
```
monitor Buffer;
export pone, toma;
const SIZE=5;
var datos: array[0..SIZE-1] of  TipoDato;
      cuenta:Natural;
      lleno,vacio: condition;
      nextIn,nextOut:0..SIZE-1;
```

```
procedure pone(d:Tipo_Dato);
begin
  if (cuenta>=SIZE) then delay(lleno);
  datos[NextIn]:= d;
  cuenta:= cuenta+1;
  nextIn:= (nextIn+1)mod SIZE;
  resume(vacio);
end;
```

```
procedure toma(var d:TipoDato);
begin
  if (cuenta=0) then delay(vacio);
  d:= datos[nextOut];
  cuenta:= cuenta-1;
  nextOut:=(nextOut+1) mod SIZE;
  resume(lleno);
end;
```

```
begin cuenta:=0; nextIn:=0; nextOut:=0; end;
```


Notas:



Notas:

Problema Lectores-Escritores: Un conjunto de procesos concurrentes de longitud indeterminada requieren acceder a una sección de una base de datos. Los procesos se clasifican en dos grupos: Los procesos lectores acceden a la base de datos para leer el contenido, pero sin cambiarlo. Los procesos escritores acceden a la sección de la base de datos para modificar su contenido. En este problema el acceso concurrente de múltiples lectores es seguro. Sin embargo, el acceso de un proceso escritor no es compatible con el acceso concurrente de otro proceso ya sea lector o escritor.

Existen dos pólitica de acceso a la sección de la base de datos:

Política de priorización de Lectores: Permite el acceso de cualquier lector a la base de datos, si la base de datos no está siendo accedida o si el proceso que está operando dentro de ella es un lector. Esta política da máximo throughput, pero puede provocar un bloqueo de los escritores por inanición si los lectores se alternan sin dejar nunca la base de datos libre.

Política de priorización de Escritores: Permite el acceso de cualquier lector a la base de datos, si la base de datos no está siendo accedida, o si el proceso que esta operando dentro de ella es un lector y sólo si no hay ningún escritor esperando para acceder. Esta política proporciona un throughput mas bajo, y además puede producir un bloqueos de los lectores si siempre hay un escritor esperando para acceder.

Se desea diseñar un protocolo que permita el acceso seguro a un número indeterminado de procesos lectores y escritores que acceden a la sección de la base de datos en cualquier orden arbitrario.

La solución que se propone se basa en un protocolo de acceso a la base de datos. Todo proceso antes de acceder debe requerir el acceso lopen(tipo:TipoProceso) y cuando sale lo comunica invocando el protocolo de salida close(tipo:TipoProceso). En ambas funciones de acceso y abandono el proceso pasa como parámetro su naturaleza {LECTOR, ESCRITOR}

Monitor ReadWriter (1)

```
monitor ReadWriter;
export
  open(iswriter: Boolean);
  close(isReader:Boolean);
var
  readerCount:Integer;
  activeW: Boolean
  okeyRead, okeyWrite: condition;

procedure open(isReader: Boolean);begin ... end;   -- se define en la siguiente pgna.
procedure open(isReader: Boolean);begin ... end;   -- se define en la siguiente pgna.

begin
  activeW:=False;
  readCount:=0;
end;
```


Notas:

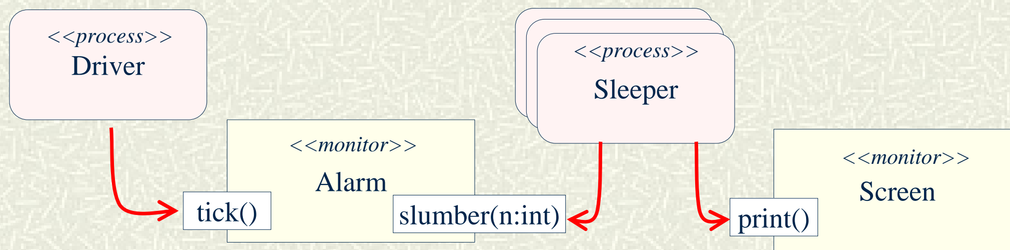
Monitor ReadWriter (2)

```
procedure open(isReader:Boolean);
begin
 if isReader then begin
   if activeW or not empty(okeyWrite)
   then delay(okeyRead);
   readCount:=readCount+1;
   resume(okeyRead)
 end else begin  -- is writer
   if activeW or (readCount<>0)
   then delay(okeyWrite);
   activeW:=True;
 end;
end;
```

```
procedure close(isReader: Boolean);
begin
 if isReader then begin
     readCount:=readCount-1;
     if (readCount=0)
     then resume(okeyWrite);
 end else begin
    activeW:=False;
    if not empty(okeyRead)
    then resume(okeyRead);
    else resume(okeyWrite);
 end;
end;
```


Notas:

Ejemplo de despertador programable




```
program AlarmClock;
  const NUM_SLEEPER=3;
  monitor alarm; ...
  monitor screen ...
  process driver ...
  process type SleeperType ...
var sleepers: array [1..NUM_SLEEPER] of SleeperType;
     i:Integer
begin
  cobegin driver; for i:=1 to NUM_SLEEPER do sleepers[i](i); coend;
end;
```


Notas:

Un conjunto de procesos que denominamos sleepers va a dormirse y desean que se les despierte en diferentes tiempos futuros. Para ello se supenden en el despertador Alarma, ejecutando el procedimiento slumber() y pasando como argumento el tiempo que desea estar dormido. El despertador opera haciendo uso de un temporizador que denominamos Driver. Este cada segundo ejecuta sobre alarm el procedimiento tick(). A esta acción Alarm responde despertando temporalmente y sucesivamente a todos los sleepers, los cuales observan si es su hora. Caso positivo pasan a su actividad que consiste en imprimir un mensaje. En caso negativo se vuelven a dormirhasta el siguiente Tick.

Monitor Alarm y Screen

```
monitor alarm;
  export
    slumber(n:Integer);
    tick();
  var
     now: Integer;
     wake: Condition;
  procedure slumber(n:Integer);
    var alarmCall: Integer
  begin
    alarmCall:=now+n;
    while now<alarmCall do begin
       delay(wake);
       resume(wake);
    end;
   end;

procedure tick;
begin
   now:=now+1;
   resume(wake);
end;

begin        -- Alarm
  now:=0;
end;
```

```
monitor screen;
  export
    print(n: Integer);
  procedure print(n:Integer);
  begin
    writeln("process", n-1, "awakes");
  end;
end;
```


Notas:

Comparación entre modelos de concurrencia.

- Utilizamos dos criterios complementarios de comparación:
  - **Poder expresivo**: Es la capacidad de la primitiva para implementar algoritmos o resolver problemas de sincronización en programas concurrentes.
    - Si una primitiva tiene capacidad de implementar otra, tiene al menos su capacidad expresiva.
    - Todas las primitivas que hemos estudiado tienen la misma capacidad expresiva,
  - **Facilidad de uso**: Criterio subjetivo que se refiere a aspectos, tales como:
    - Como de natural es el uso de la primitiva.
    - Como de fácil es combinar la primitiva con otras sentencias del lenguaje.
    - Cuan proclive es la primitiva para que el programador cometa errores.

Notas:

En los capítulos anteriores, y por razones históricas, se han introducido un gran número de primitivas para que sean seguras y generales las operaciones de sincronización entre procesos dentro de un programa concurrente. Cualquiera de ellas permiten resolver todos los problemas de sincronización y exclusión mutua entre procesos concurrentes que habían sido planteados.

El poder expresivo, y la facilidad de uso suelen estar en conflicto. Las primitivas de bajo nivel, suelen tener mayor poder expresivo, pero son más difíciles de utilizar. Así por ejemplo, los semáforos, tienen una capacidad expresiva máxima, y permiten abordar cualquier situación de sincronización, pero su uso es peligroso, ya que es fácil que el programador cometa errores por olvido de alguna de las llamadas, en alguno de los procesos que hagan referencia al semáforo. Por el contrario, el monitor que es un concepto de alto nivel, es muy fácil de utilizar y mantener, pero sin embargo requieren una utilización muy forzada para resolver ciertos problemas de sincronización tal como el problema de "múltiples escritores y lectores".

Facilidad de uso de las primitivas de sincronización.

- La sincronización a través de semáforos presenta la capacidad plena, pero su facilidad de uso es muy pobre por su bajo nivel de abstracción y su gestión distribuida.
- Los Monitores incrementan la abstracción y concentran su gestión, pero al final requieren variables "Condition" que son también de muy bajo nivel de abstracción aunque permanecen localizadas en el monitor.
- La invocación de procedimientos remotos es de alto nivel de abstracción y de uso muy seguro, aunque presenta dos problemas:
  - Implementa los objetos pasivos con un modelo de módulo activo que no corresponde a lo que el programador espera.
  - Conduce a implementaciones ineficientes como consecuencia del gran número de cambios de contextos.
- No hay una solución absolutamente favorable. Para una programación segura, mantenible y eficiente se requiere combinar varios tipos de mecanismos.

Notas:

1. **Los semáforos y las variables de condición son de un nivel muy bajo, a efectos de ser consideradas adecuadas para un lenguaje de programación de propósito general.** Esta afirmación no es contraria al hecho de reconocer a los semáforos su carácter de componente básico que todas los sistemas lo van a utilizar a bajo nivel. El semáforo juega en este aspecto un papel equivalente al de la sentencia GO TO en programación secuencial.
2. **Los método de paso de mensajes (incorporando la facilidad de condiciones de guarda) representan un mecanismo más abstracto y unificado para la programación concurrente.** El modelo de lenguaje a que dá lugar estas primitivas es más natural y seguro, y además simplifica considerablemente el lenguaje al eliminar el uso de variables compartidas.
3. **La invocación remota de procedimientos es la abstracción de alto nivel más efectiva del tipo de paso de mensajes.** La invocación remota tiene un gran número de ventajas respecto de la comunicación sincrona:
   - No requiere introducir el concepto intermedio de canal o buzón.
   - No requiere simetría en las sentencias Select.
   - El lenguaje resulta de mas simple al utilizar una sintaxis similar para la invocación de procedimiento remotos y de procedimientos locales.
4. **Las estructuras tipo Monitor constituyen un método efectivo de encapsular recursos compartidos**. Constituye la mejor abstracción de un recurso si excluimos su modelado mediante un proceso. No obstante, presenta el problema de requerir variables condicionadas, para que cubra la totalidas de las posibilidades de sincronización.