Lección 7: Sincronización de procesos mediante monitores

- Introducción
- ¿Qué es un monitor?
- Características y funcionamiento de un monitor
- Implementación de un monitor en C++
- Algunos ejemplos de aplicación:
  - El caso de los productores/consumidores
  - El problema de la cena de los filósofos
  - El caso de los lectores/escritores

Introducción

- Los semáforos tienen algunas características que pueden generar inconvenientes:
  - 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

se necesita encapsulación

¿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* fue acuñado por **C.A.R. Hoare** [1973].
- Posteriormente, **Brinch Hansen** incorpora los monitores al lenguaje Pascal Concurrente [1975]

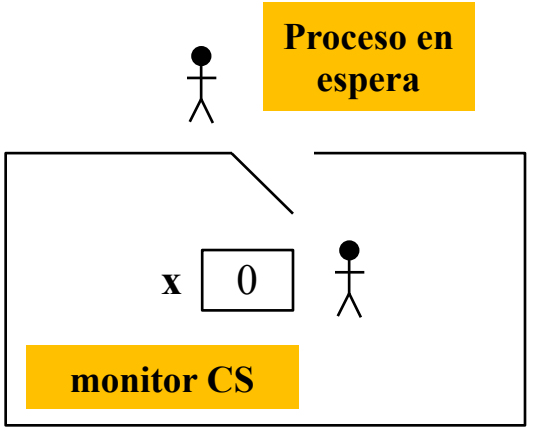
¿Qué es un monitor?

- 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
  - las operaciones de un monitor se ejecutan, *por definición*, en **exclusión mutua**
  - dispone de mecanismos específicos para la sincronización: ***variables "condición"***

Un sencillo ejemplo

```
monitor CS
    integer x := 0
    operation increment()
        x := x + 1
    end

end
```

Process P	*Process Q*
`CS.increment()`	`CS.increment()`



Características de un monitor

```
monitor CS
    integer x := 0
    operation increment()
        x := x + 1
    end

end
```

Process P	*Process Q*
`CS.increment()`	`CS.increment()`

- 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: **nombreMonitor.operación(listaParámetros)**

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**
    - con frecuencia es necesaria una sincronización explícita entre procesos
    - 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"

- Representa una *condición* de interés para los procesos que se sincronizan por medio del monitor
  - cada variable tiene asociada una ***cola FIFO para los procesos que están bloqueados***

- Ofrece dos operaciones atómicas básicas:
  - ***waitC(variable_condicion)***
    - el proceso es bloqueado en la cola de la variable condición"
  - ***signalC(variable_condicion)***
    - "el primer proceso de la cola es desbloqueado"

No confundirlas con las operaciones de semáforos!

Sobre las variables "condición"

- instrucción ***waitC(c)***:
  - el proceso invocador queda "bloqueado" y pasa a la cola FIFO asociada a la variable *c*, en espera de ser despertado
  - el cerrojo que garantiza la exclusión mutua del monitor queda libre
- instrucción ***signalC(c)***:
  - si la cola de la señal está vacía: no pasa nada y la operación sigue con su ejecución
    - al terminar, el monitor está disponible para otro proceso
  - si la cola no está vacía:
    - se saca el primer proceso de la cola y se "desbloquea"
    - *políticas de reanudación* determinan qué proceso continúa su ejecución
- instrucción ***signalC_all(c)***
- instrucción ***emptyC(c)***

Sobre las variables "condición"

- **Ejemplo**: diséñese un programa concurrente en el que
  - 10 procesos incrementan 1000 veces cada uno una variable **x** que inicialmente vale 0
  - Un proceso espera a que la variable llegue al valor 3000, e informa por la salida estándar del hecho

```
Process P(i:1..10):

    //incrementar la variable

end
```

```
Process info:

    //informar cuando llegue a 3000

end
```

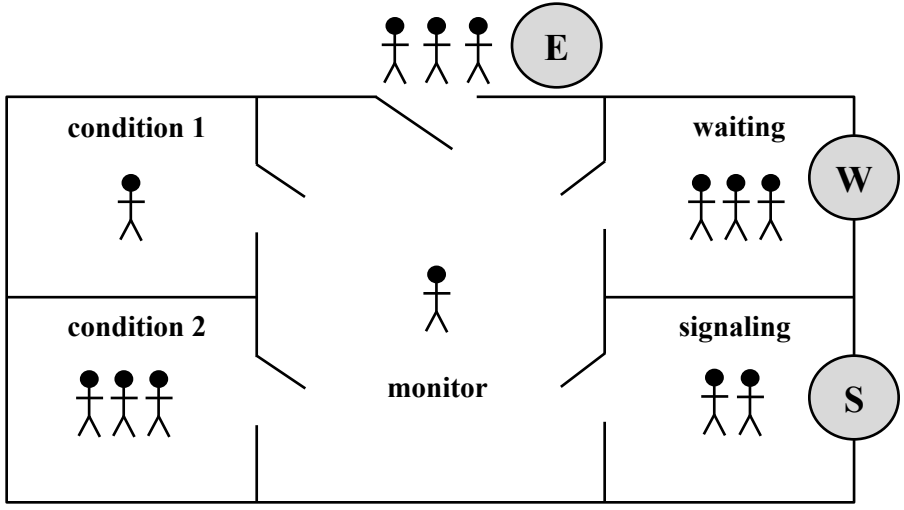
Sobre las variables "condición"

- Diferencias entre las instrucciones de un monitor y las de un semáforo con nombre similar:

Semáforos	Monitores
wait puede bloquearse, o no	***waitC*** siempre se bloquea
signal siempre tiene un efecto	***signalC*** no tiene efecto si la cola está vacía
signal puede desbloquear un proceso cualquiera de la cola (depende del tipo de semáforo)	***signalC*** siempre desbloquea al primer proceso en la cola
wait/signal en cualquier parte del programa	***waitC/signalC*** solo dentro de monitores

Sobre las variables "condición"

- *Políticas de reanudación*:
  - versión clásica de un monitor: **E < S < W**
    - "Immediate Resumption Requirement" (IRR)
  - versión implementada en Java: **E = W < S**

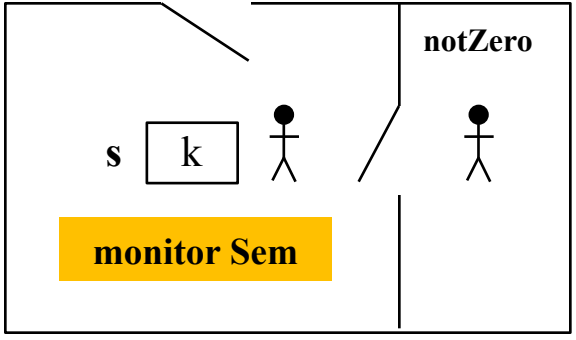


Implementando un semáforo con un monitor

```
monitor Sem
    integer s := 1
    condition notZero
    operation wait()
        if s = 0
            waitC(notZero)
        end
        s := s - 1
    end
    operation signal()
        s := s + 1
        signalC(notZero)
    end
end
```

E<S<W

E=W<S



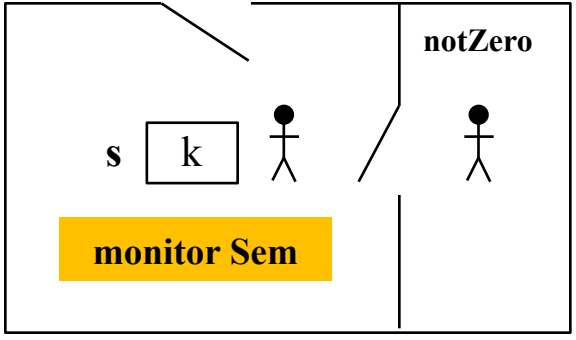

```
Process P(i:1..N)::
while true
    SNC
    Sem.wait()
    SC
    Sem.signal()
end
```

Implementando un semáforo con un monitor

```
monitor Sem
    integer s := 1
    condition notZero
    operation wait()
        while s = 0
            waitC(notZero)
        end
        s := s - 1
    end
    operation signal()
        s := s + 1
        signalC(notZero)
    end
end
```

E<S<W

E=W<S




```
Process P(i:1..N)::
while true
    SNC
    Sem.wait()
    SC
    Sem.signal()
end
```

El problema de los productores/consumidores

- **Ejemplo:**
  - tenemos un sistema con un proceso productor y un consumidor
  - Caso 1: buffer intermedio de **capacidad infinita**
  - Caso 2: buffer intermedio de **capacidad finita**

el_tipo queue buffer := []
...

Process productor	*Process consumidor*
el_tipo d	el_tipo d
while true	while true
produce(d)	d:=consume(buffer)
append(d,buffer)	usa(d)

El problema de los productores/consumidores

```
monitor almacen_limitado
  ...
  operation append(el_tipo d)
     ...

  operation consume() return el_tipo
     ...
```

```
process productor
  el_tipo d
  while true
     preparar d
     almacen_limitado.append(d)
  end
```

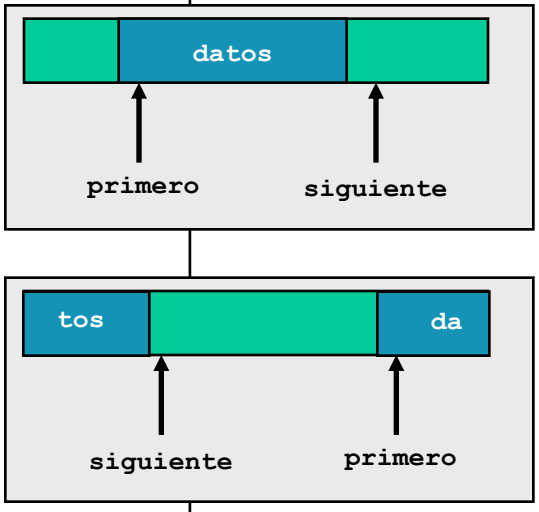
```
process consumidor
  el_tipo d
  while true
     d:= almacen_limitado.consume()
     usa(d)
  end
```

```
monitor almacen_limitado
     integer n := ... --capacidad, >=1
     ...
     condition no_lleno,no_vacio

  operation append(el_tipo d)
     ...
     while "esta lleno"
         waitC(no_lleno)
     end
     ...

  operation consume() return el_tipo
     el_tipo d
     ...
     while "esta vacío"
         waitC(no_vacio)
     end
     ...
     return d
```

```
monitor almacen_limitado
    integer n := ... --capacidad, >=1
    ...
    condition no_lleno,no_vacio

  operation append(el_tipo d)
    ...
    while "esta lleno"
        waitC(no_lleno)
    end
    ...
    signalC(no_vacio)

  operation consume() return el_tipo
    el_tipo d
    ...
    while "esta vacío"
        waitC(no_vacio)
    end
    ...
    signalC(no_lleno)
    return d
```

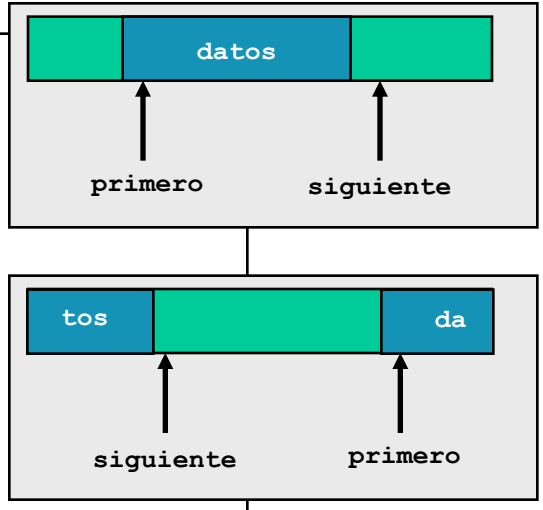
El problema de los productores/consumidores

```
monitor almacen_limitado
    integer n := .... //n>=1
    integer primero := 1
            siguiente := 1
    el_tipo array[1..n] almacen
    natural n_datos := 0

    condition no_lleno,no_vacio

    operation append(el_tipo d)
       while n_datos=n
          waitC(no_lleno)
       end
       almacen[siguiente] := d
       siguiente := (siguiente mod n) + 1
       n_datos := n_datos+1
       signalC(no_vacio)
```



El problema de los productores/consumidores

```
monitor almacen_limitado
    ...

    operation consume() return el_tipo
       el_tipo d

       while n_datos=0
          waitC(no_vacio)
       end
       d := almacen[primero]
       primero := (primero mod n) + 1
       n_datos := n_datos-1
       signalC(no_lleno)
       return d
```



Implementación de un monitor en C++

- Un monitor como instancia de una clase que implemente el comportamiento deseado
- Usando variables condición y mutex
- El acceso en exclusión mutua lo gestionaremos explícitamente
  - declarar un mutex dentro del objeto
  - bloquearlo al iniciar cada función
- Declarar todas las variables del monitor como atributos privados

```
#include <mutex>
#include <condition_variable>
```

Ejemplo

```
Process P(i: 1..10)::
    for j:=1..1000
        var_x.inc()
    end
end

Process informador::
    var_x.informa()
end
```

Diseño

```
Monitor var_x
    int x := 0
    condition alMenos3000

    operation inc()
        x++
        if x=3000
            signalC(alMenos3000)
        end
    end

    operation informa()
        if x<3000
            waitC(alMenos3000)
        end
        write("Al menos hay 3000")
    end
end
```

Implementación de un monitor en C++

```
Monitor var_x
    int x := 0
    condition alMenos3000

    operation inc()
        x++
        if x=3000
            signalC(alMenos3000)
        end
    end

    operation informa()
        if x<3000
            waitC(alMenos3000)
        end
        write("Al menos hay 3000")
    end
end
```

monitorVar_x.hpp

```
class MonitorVar_x {
public:
    ...
private:
    ...
};
```

monitorVar_x.cpp

```
...
void MonitorVar_x::inc() {
    ...
}

void MonitorVar_x::informar(){
    ...
}
...
```

Implementación de un monitor en C++

```
Monitor var_x
    int x := 0
    condition alMenos3000

    operation inc()
        x++
        if x=3000
            signalC(alMenos3(
        end
    end

    operation informa()
        if x<3000
            waitC(alMenos300(
        end
        write("Al menos hay 1
    end
end
```

monitorVar_x.hpp

```
#include <mutex>
#include <condition_variable>

class MonitorVar_x {
public:
    MonitorVar_x();        //constructor
    ~MonitorVar_x();       //destructor
    void inc();
    void informar();
private:
    std::mutex mtxMonitor;
    std::condition_variable alMenos3000;
    int x;
};
```

Implementación de u

```
Monitor var_x
    int x := 0
    condition alMenos3000

    operation inc()
        x++
        if x=3000
            signalC(alMenos30
        end
    end

    operation informa()
        if x<3000
            waitC(alMenos3000
        end
        write("Al menos hay 3
    end
end
```

¿E<S<W?

¿E=W<S?

monitorVar_x.cpp

```
MonitorVar_x::MonitorVar_x() {
    x = 0;
}
MonitorVar_x::~MonitorVar_x() {

}
void MonitorVar_x::inc() {
    unique_lock<mutex> lck(mtxMonitor);

    x = x + 1;
    if (x == 3000) {
        alMenos3000.notify_one();
    }
} //aquí se libera mtxMonitor
void MonitorVar_x::informar() {
    unique_lock<mutex> lck(mtxMonitor);

    if (x < 3000) {
        alMenos3000.wait(lck);
    }
    cout << "Al menos hay 3000: "
         << x << endl;
} //aquí se libera mtxMonitor
```

Implementaci

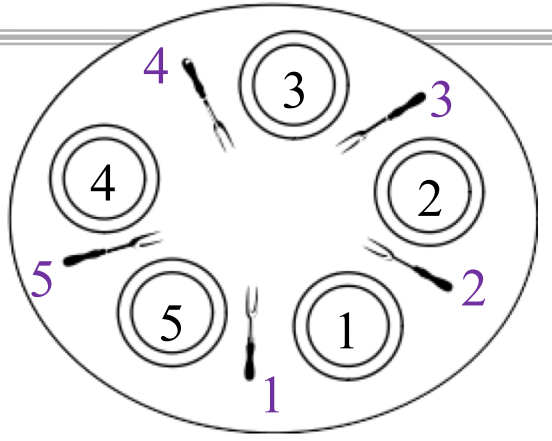
```
Process P(i: 1..10):
    for j:=1..1000
        var_x.inc()
    end
end

Process informador::
    var_x.informa()
end
```

main.cpp

```
const int N = 10;

void informar(MonitorVar_x &mE){
    mE.informar();
}

void incrementar(MonitorVar_x &mE) {
    for (int i=0; i<1000; ++i) {
        mE.inc();
    }
}

int main() {
    MonitorVar_x monX;
    thread P[N];
    thread informador = thread(informar, ref(monX));
    for (int i=0; i<N; ++i) {
        P[i] = thread(incrementar, ref(monX));
    }
    ...
    return 0;
}
```

El problema de los filósofos

- Dijkstra 65, Hoare 85
  - Prototipo de sistema en que los procesos comparten recursos conservativos
- Hay 5 filósofos sentados alrededor de una mesa para comer espaguetis
- Hay 5 tenedores, e infinita pasta
- Hacen falta dos tenedores para comer
  - Cada filósofo puede usar los que tiene más cerca
  - **Cada filósofo coge ambos tenedores a la vez**
- Se pide un programa que simule el comportamiento del sistema




```
Process filosofo(i:1..5):
   while true
      //piensa
      //coge tenedores
      //come
      //deja tenedores
   end
end
```

El problema de los filósofos

Diseño

```
Process filosofo(i:1..N)::
   while true
      //pensar
      tenedores.coger(i)
      //comer
      tenedores.dejar(i)
   end
end
```

```
Monitor tenedores
   boolean array[1..N] ocupado := (1..N,false)
   condition liberado //espero a que se libere alguno

   operation coger(integer i)
   . . .
   operation dejar(integer i)
   . . .
```

El problema de los filósofos

Diseño

```
Monitor tenedores
   boolean array[1..N] ocupado := (1..N,false)
   condition liberado //espero a que se libere alguno

   operation coger(integer i)
      while(ocupado[i] OR ocupado[i⊕1])
          waitC(liberado)
      end
      ocupado[i] := true
      ocupado[i⊕1] := true
   end operation

   . . .
end
```

El problema de los filósofos

Diseño

```
Monitor tenedores
   boolean array[1..N] ocupado := (1..N,false)
   condition liberado //espero a que se libere alguno

   . . .

   operation dejar(integer i)
      ocupado[i] := false
      ocupado[i⊕1] := false
      signalC_all(liberado)
   end operation
end monitor
```

El problema de los filósofos

```
Process filosofo(i:1..N)::
   while true
      //pensar
      tenedores.coger(i)
      //comer
      tenedores.dejar(i)
   end
end
```

Implementación C++

```
#include "monitorTenedores.hpp"
MonitorTenedores tenedores;
void filosofo(unsigned i){
   while (true) {
      //pensar
      tenedores.coger(i)
      //comer
      tenedores.dejar(i)
   }
}
int main() {
    thread filosofo[N];
    ...
```

El problema de los filósofos

```
Process filosofo(i:1..N)::
   while true
      //pensar
      tenedores.coger(i)
      //comer
      tenedores.dejar(i)
   end
end
```

Implementación C++

```
#include "monitorTenedores.hpp"
void filosofo(unsigned i, MonitorTenedores &tenedores){
   while (true) {
      //pensar
      tenedores.coger(i)
      //comer
      tenedores.dejar(i)
   }
}
int main() {
    MonitorTenedores tenedores;
    thread filosofo[N];
    ...
```

El problema de los filósofo

```
Monitor tenedores
   boolean array[1..N] ocupado := (1..N,false)
   condition liberado

   operation coger(integer i)
   . . .
   operation dejar(integer i)
   . . .
```

monitorTenedores.hpp

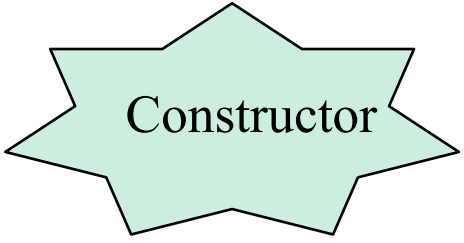
```
#include <mutex>
#include <condition_variable>

class MonitorTenedores {
public:
   MonitorTenedores();
   void coger(int i);
   void dejar(int i);
private:
   bool  ocupado[N]; //asumir N declarado
   mutex mtxMonitor; //para la condición
   condition_variable liberado;
};
```

El problema de los fi

```
Monitor tenedores
   boolean array[1..N] ocupado := (1..N,false)
   condition liberado

   operation coger(integer i)
   . . .
   operation dejar(integer i)
   . . .
```

monitorTenedores.cpp

```
MonitorTenedores::MonitorTenedores() {

    for(int i=0; i<N; i++) {
        ocupado[i] = false;
    }
};
```



El problema de los filósofos

```
operation coger(integer i)
   while(ocupado[i] OR ocupado[i+1])
      waitC(liberado)
   end
   ocupado[i] := true
   ocupado[i⊕1] := true
end
```

monitorTenedores.cpp

```
void MonitorTenedores::coger(int i) {

   unique_lock<mutex> lck(mtxMonitor);

   //esperar tenedores libres
   while(ocupado[i] || ocupado[(i + 1) % N]) {
      liberado.wait(lck);
   }
   ocupado[i] = true;
   ocupado[(i + 1) % N] = true;
};
```

Se bloquea hasta poder cogerlo

Se libera automáticamente al cerrar el bloque

El problema de los filósofos

```
operation dejar(integer i)
   ocupado[i] := false
   ocupado[i⊕1] := false
   signalC_all(liberado)
end
```

monitorTenedores.cpp

```
void MonitorTenedores::dejar(int i) {

   unique_lock<mutex> lck(mtxMonitor);

   ocupado[i] = false;
   ocupado[(i + 1) % N] = false;
   liberado.notify_all();
};
```

Implementación de un monitor en C++

- ¿Qué pasa si una función del monitor quiere invocar otra función del mismo?
- ¿Puedo implementar funciones recursivas?

```
recursive_mutex mtxMonitor
condition_variable_any liberado;
```

```
unique_lock<recursive_mutex> lck(mtxMonitor);
```

El problema de los lectores/escritores

- **Problema**:
  - dos tipos de procesos para acceder a una base de datos:
    - lectores: consultan la BBDD
    - escritores: consultan y modifican la BBDD
  - cualquier transacción aislada mantiene la consistencia de la BBDD
  - cuando un escritor accede a la BBDD, es el único proceso que la puede usar
  - varios lectores pueden acceder simultáneamente

El problema de los lectores/escritores

Diseño

```
Process lector
  while true
    ...
    controlaLyE.pideLeer()
    ...
    controlaLyE.dejaDeLeer()
    ...
end
```

```
monitor controlaLyE
  ...
  operation pideLeer()
  operation dejaDeLeer()
  operation pideEscribir()
  operation dejaDeEscribir()
end
```

```
Process escritor
  while true
    ...
    controlaLyE.pideEscribir()
    ...
    controlaLyE.dejaDeEscribir()
    ...
end
```

El problema de los lectores/escritores

Diseño

```
monitor controlaLyE
  integer nLec := 0
          nEsc :=0
  condition okLeer  --señala nEsc=0
            okEscribir --señala nEsc=0 AND nLec=0
  operation pideLeer()
     while (nEsc>0)
         waitC(okLeer)
     nLec := nLec+1

  operation dejaDeLeer()
    nLec := nLec-1
    if (nLec=0)
       signalC(okEscribir) -- ¿signalC(okLeer)?
```

El problema de los lectores/escritores

```
monitor controlaLyE
  ...
  operation pideEscribir()
    while (nLec>0) OR (nEsc>0)
       waitC(okEscribir)
    nEsc := nEsc+1

  operation dejaDeEscribir()
    nEsc := nEsc-1
    signalC(okEscribir)
    signalC_all(okLeer)
```

Un ejercicio de tra

- Diseñar, mediante monitores, un programa que se comporte como el siguiente

```
int x := . . . //lo que sea
    y := . . . //lo que sea
    z := . . . //lo que sea
    w := . . . //lo que sea

process cliente(i:1..300)::
   int miX
   miX := . . .  //lo que sea

   <await x+y>0
      y := y+1
      x := miX
   >
   <await z>0
     z := z+3+miX-w
   >
   <y := 2*x+y>
end
```