Lección 7: Sincronización de procesos mediante monitores

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

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]

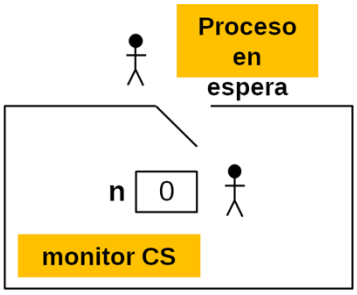
¿Qué es un monitor?

- 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 n := 0
   operation increment()
      n := n + 1
   end

end
```

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



Características de un monitor

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

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"

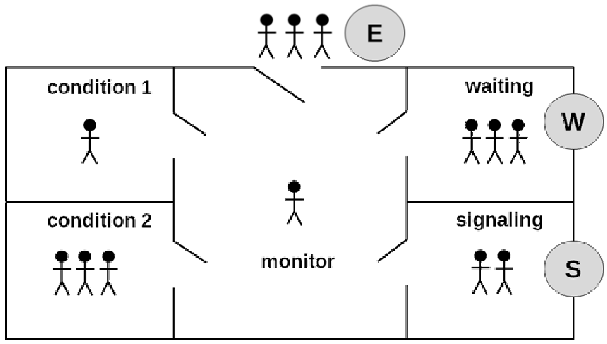
No confundirlas con las operaciones de semáforos!

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"

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



Sobre las variables "condición"

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

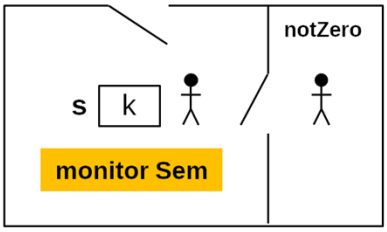
Semáforos	Monitores
wait puede bloquearse	***wait*C** siempre se bloquea
signal siempre tiene un efecto	***signal*C** no tiene efecto si la cola está vacía
signal puede desbloquear un proceso cualquiera de la cola	***signal*C** siempre desbloquea al primer proceso en la cola
Un proceso desbloqueado por la instrucción ***signal*** puede continuar su ejecución inmediatamente	Dependerá de la política de reanudación

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



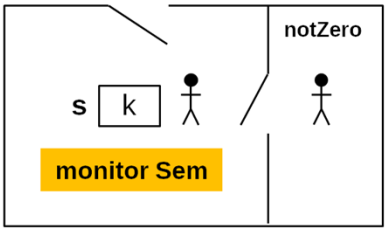
Process P	*Process Q*
loop forever	loop forever
SNC	SNC
Sem.wait()	Sem.wait()
SC	SC
Sem.signal()	Sem.signal()

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	*Process Q*
loop forever	loop forever
SNC	SNC
Sem.wait()	Sem.wait()
SC	SC
Sem.signal()	Sem.signal()

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**

queue el_tipo buffer := []
...

Process productor	*Process consumidor*
el_tipo d	el_tipo d
loop forever	loop forever
produce(d)	consume(d,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
  loop forever
     preparar d
     almacen_limitado.append(d)
  end
```

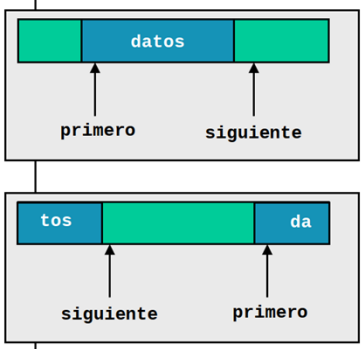
```
process consumidor
  el_tipo d
  loop forever
     d:= almacen_limitado.consume()
     usa(d)
  end
```

```
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
```

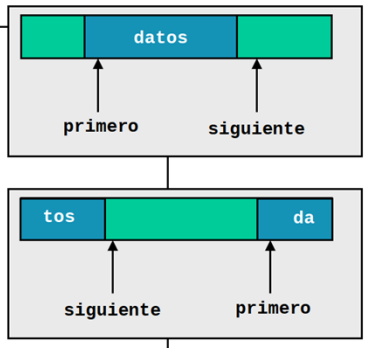
El problema de los productores/consumidores

```
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>
```

Implementación de un monitor en C++

```
class monitorLoQueSea{
public:
   monitorLoQueSea(...); //constructor
   void operación_1(...);
   ...    //resto de operaciones
private:
   tipo_1 var_1; //TODAS PRIVADAS
   ...           //resto de variables
   mutex mtxMonitor; //FUNDAMENTAL: mutex usarán las funcs
   condition_variable unaCondicion;
   ...
   void operación_k(...); //operaciones privadas
};
```

```
//implementación
monitorLoQueSea::monitorLoQueSea(){ //constructor
   ... //resto del código
};
```

```
void monitorLoQueSea::operación_1(...){

   unique_lock<mutex> lck(mtxMonitor);

   //esperar a que se cumpla lo necesario
   while(«no se dan las condiciones requeridas»){
      unaCondicion.wait(lck);
   }
   ... //resto del código
};
```

Se bloquea hasta poder cogerlo

mtxMonitor se libera automáticamente al cerrar el bloque

Implementación de un monitor en C++

```
void monitorLoQueSea::operación_2(...){
   unique_lock<mutex> lck(mtxMonitor);
   ...
   unaCondicion.notify_one();
   ...
   unaCondicion.notify_all();
   ... //resto del código
};
```

Se bloquea hasta poder cogerlo

mtxMonitor se libera automáticamente al cerrar el bloque

¿E<S<W? **¿E=W<S?**

El problema de l

```
Process filosofa_V1(i:1..N)::
   while(true)
      //pensar
      tenedores.coger(i)
      //comer
      tenedores.dejar(i)
   end while
```

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

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

El problema de los productores/consumidores

```
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 while
      ocupado[i] := true
      ocupado[i+1] := true
   end operation

   . . .
end monitor
```

El problema de los productores/consumidores

```
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
```

Implementación de un monitor en C++

```
//especificación
class controlaTenedores{
public:
   controlaTenedores();
   void coger(int i);
   void dejar(int i);
private:
   bool  ocupado[N]; //asumir N declarado
   mutex mtxMonitor; //para la condición
   condition_variable liberado;
                        //versión con una única condición
   void informa(string mens);
};
```

Implementación de un monitor en C++

```
//implementación
controlaTenedores::controlaTenedores(){ //constructor

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

Implementación de un monitor en C++

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

   unique_lock<mutex> lck(mtxMonitor);

   //esperar tenedores libres
   while(ocupado[i] || ocupado[(i + 1) % N]){
      informa(to_string(i) + " me bloqueo\n");
      //para la condición
      liberado.wait(lck);
   }
   ocupado[i] = true;
   ocupado[(i + 1) % N] = true;
   informa("\t" + to_string(i) + " eating\n");
};
```

Se bloquea hasta poder cogerlo

Se libera automáticamente al cerrar el bloque

Implementación de un monitor en C++

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

   unique_lock<mutex> lck(mtxMonitor);

   ocupado[i] = false;
   ocupado[(i + 1) % N] = false;
   informa("\t\t" + to_string(i) + " out\n");
   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);
```

Implementación de un monitor en Java. V1

- Un monitor se puede implementar en Java como instancia de una clase que cumple:
  - todos los atributos (variables permanentes) son *private*
  - todos los métodos (operaciones) son declarados *synchronized* (exclusión mutua a nivel de acción)
  - usa las operaciones *wait()*, *notify()* y *notifyAll()* para implementar la sincronización condicionada
- Política de reanudación de los monitores Java: **E = W < S**
- El paquete *java.util.concurrent* dispone de constructores de concurrencia más potentes e, incluso, de variables "condición"

Implementación de un monitor en Java. V1

- Todo objeto Java tiene asociado un *lock* que a su vez tiene asociado un *wait set*
  - conjunto de threads bloqueados a nivel de objeto
- Un método *synchronized* automáticamente toma y deja el *lock*
- Semántica de las operaciones:
  - *wait()* :: el proceso que invoca el método se bloquea y es añadido al *wait set* del objeto liberando inmediatamente su *lock*
    - *Interrupted exception* debe ser capturada
  - *notify()* :: desbloquea un proceso cualquiera del *wait set* del objeto
  - *notifyAll()* :: desbloquea todos los procesos del *wait set* del objeto
- Para invocar cualquiera de las operaciones anteriores el proceso invocador debe tener el *lock* del objeto
  - sólo desde métodos *synchronized*

Implementación de un monitor en Java

- Cuando un proceso es desbloqueado por una operación *notify()* o *notifyAll()* debe:
  - y volver a evaluar la condición que provocó su bloqueo
    - ¡su valor ha podido cambiar desde que se produjo el desbloqueo!
    - por ser política **E = W < S**

```
synchronized método(…)
{   …
    while(!condición)
        wait();
    //condición TRUE
    …
}
```

Implementación de un monitor en Java

- Problema: ¿cómo bloquear en un mismo objeto a procesos en base a dos o más condiciones diferentes?

```
synchronized método1(…) {
   while(x==0)
      wait();
}
synchronized método2(…) {
   while(y==0)
      wait();
}
synchronized método3(…) {
   if (…) x = 1;
   else y = 1;
   notifyAll();
}
```

Implementación de un monitor en Java. V2

- Alternativa: usando variables condición
- Es una implementación más próxima a la semántica de monitores que manejamos
  - aunque exige que el acceso en exclusión mutua lo gestinemos explícitamente

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
```

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class monitorLoQueSea {
    //variables privadas
    ...
    //lock: ha  de ser "reentrant" para que un método
    //del objeto pueda invocar otro método del mismo
    private final Lock mutex = new ReentrantLock();
    ...
    //las variables condición están asociadas a los locks
    private final Condition c1 = mutex.newCondition();
    private final Condition c2 = mutex.newCondition();
    ...
}
```

```
 ...
public class monitorLoQueSea {
    //variables privadas
    ...
    public void operacion(...)
                    throws InterruptedException {

        mutex.lock(); //se toma al principio del método

        try {
            ...
        } finally { //finally: desbloquear siempre
                    //haya excepción, o no
            mutex.unlock();
        }
    }
}
```

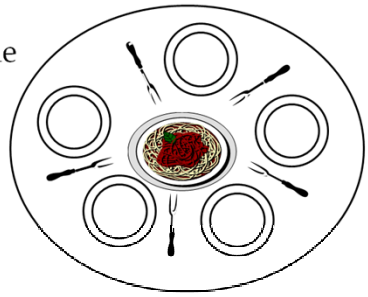
```
public void operacion(...)
                throws InterruptedException {
    ...
    mutex.lock(); //se toma al principio del método

    try {
        ...
        c1.await();
        ...
        c2.signal();
        ...
        c1.signalAll();
        ...
    } finally { //finally: desbloquear siempre
                //haya excepción, o no
        mutex.unlock();
    }
}
```

```
public void operacion(...)
                throws InterruptedException {
    ...
    mutex.lock(); //se toma al principio del método

    try {
        ...
        c1.await();
        ...
        c2.signal();
        ...
        c1.signalAll();
        ...
    } finally { //finally: desbloquear siempre
                //haya excepción, o no
        mutex.unlock();
    }
}
```

```
while (! cond){
     c1.await();
}
```

El problema de la cena de los filósofos

- El problema:
  - cada filósofo, alternativamente, piensa y come
  - se necesitan dos tenedores para comer
    - el de la izquierda y el de la derecha
  - la cantidad de espaguetis ¡es infinita!
  - hay que evitar que se mueran de hambre
    - por la cabezonería de los filósofos
- Salta a la vista que:
  - dos filósofos vecinos no pueden comer a la vez
  - no más de dos filósofos pueden comer a la vez
- Objetivo: desarrollar un programa concurrente que simule el sistema



```
Process Filósofo::
loop forever
    piensa
    coge tenedores
    come
    deja tenedores
```

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

```
process lector
  loop forever
    ...
    controlaLyE.pideLeer()
    ...
    controlaLyE.dejaDeLeer()
    ...
```

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

```
process escritor
  loop forever
    ...
    controlaLyE.pideEscribir()
    ...
    controlaLyE.dejaDeEscribir()
    ...
```

El problema de los lectores/escritores

```
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)
```

Ejercicios

- **Ejercicio 1**: Control de puente con sólo un carril
  - Un puente que cruza un río de norte a sur sólo dispone de un carril. Varios coches en una misma dirección pueden cruzar el puente a la vez, pero no coches en direcciones contrarias
  - Se pide modelar el sistema, de manera que cada coche es un proceso y el control del puente se realiza mediante un monitor.
- **Ejercicio 2**: Mejorar la solución anterior para que se cumpla la siguiente propiedad de equidad: cuando un coche llega al puente, como mucho K coches en dirección contraria cruzarán el puente antes que él

Ejercicios

- **Ejercicio 3**: Dos tipos de procesos, A y B, entran y salen de una habitación. Un proceso A que entra no puede salir hasta que se encuentre en la habitación con dos procesos B, mientras que un B sólo puede salir si se ha encontrado con un A
  - escribir un monitor para la sincronización de los procesos
  - escribir el código de los procesos de tipo A y B
    - **Nota**: hacer una versión para cada una de las dos posibles interpretaciones

Ejercicios

- **Ejercicio 4**: Un sistema dispone de tres tipos de recursos, que vamos a denominar T1, T2 y T3, de los que el sistema dispone de 10, 5 y 8 unidades, respectivamente. Un proceso cliente ejecuta el siguiente bucle
  - escribir un monitor para el control del acceso a los recursos por n procesos
  - escribir el código de los procesos cliente.
  - se valorará, además, que los procesos cliente sean atendidos en el orden en que realizan las peticiones

```
loop forever
   obtener valores(n1,n2,n3)
     --1<=n1<=10,1<=n2<=5,1<=n3<=8
   reservar (n1,n2,n3) recursos
   usar recursos pedidos
   liberar(n1,n2,n3)
```

Ejercicios

- **Ejercicio 5**: Escribir un programa en el que "n" procesos cliente (n>=2) acceden concurrentemente a una matriz de reales `m` de dimensión **dxd**, de acuerdo al esquema de código en el recuadro
  - Hay que tener en cuenta que la función **f** es de tipo real, de la que no sabemos nada
- la matriz debe ser inicializada con todas sus componentes igual a 0.0 antes de que ningún proceso cliente pueda acceder a ella

```
loop forever
  leer(i);leer(j);leer(val)
  . . .
  asignar a matriz(i,j):=f(val)
```

Ejercicios

- **Ejercicio 6**: Implementar un programa concurrente correspodiente a 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