Semáforos: Útiles, pero aún de bajo nivel

⟶ Para entender cómo se usa un semáforo, hay que mirar el programa completo

⟶ Si agregamos un nuevo proceso al programa, entonces hay que asegurarse de que el proceso use bien los semáforos

⟶ Si un proceso olvida llamar a *V* después de una sección crítica, entonces el programa puede bloquearse:

- la causa del problema es difícil de identificar

⟶ El orden relativo de ejecución de las operaciones *P* y *V* importa

Semáforos: Útiles, pero aún de bajo nivel

⟶ Para entender cómo se usa un semáforo, hay que mirar el programa completo

⟶ Si agregamos un nuevo proceso al programa, entonces hay que asegurarse de que el proceso use bien los semáforos

⟶ Si un proceso olvida llamar a *V* después de una sección crítica, entonces el programa puede bloquearse:

- la causa del problema es difícil de identificar

⟶ El orden relativo de ejecución de las operaciones *P* y *V* importa

Solución al problema de múltiples productores y consumidores y un *buffer* de capacidad $k > 1$

```
bufferType[k] buf
int front ⟵ 0, rear ⟵ 0
sem empty ⟵ n, full ⟵ 0
sem mutexD ⟵ 1, mutexF ⟵ 1

process Producer[i⟵1 … m]:
   while (true):
      produce data
      P(empty)
      P(mutexD)
      buf[rear].deposit(data)
      rear ⟵ (rear+1)%k
      V(mutexD)
      V(full)
```

```
process Consumer[j⟵1 … n]:
   while (true):
      P(full)
      P(mutexF)
      result ⟵ buf[front].fetch()
      front ⟵ (front+1)%k
      V(mutexF)
      V(empty)
      consume result
```

Solución al problema de múltiples productores y consumidores y un *buffer* de capacidad > 1

```
bufferType[n] buf
int front ⟵ 0, rear ⟵ 0
sem empty ⟵ n, full ⟵ 0
sem mutexD ⟵ 1, mutexF ⟵ 1

process Producer[i⟵1 … m]:
   while (true):
     produce data
     P(empty)
     P(mutexD)
     buf[rear].deposit(data)
     rear ⟵ (rear+1)%n
     V(mutexD)
     V(full)
```

```
process Consumer[j⟵1 … n]:
   while (true):
     P(full)
     P(mutexF)
     result ⟵ buf[front].fetch()
     front ⟵ (front+1)%n
     V(mutexF)
     V(empty)
     consume result
```

Solución al problema de múltiples productores y consumidores y un *buffer* de capacidad > 1

```
bufferType[n] buf
int front ⟵ 0, rear ⟵ 0
sem empty ⟵ n, full ⟵ 0
sem mutexD ⟵ 1, mutexF ⟵ 1

process Producer[i⟵1 … m]:
   while (true):
      produce data
      P(empty)
      P(mutexD)
      buf[rear].deposit(data)
      rear ⟵ (rear+1)%n
      V(mutexD)
      V(full)
```

```
process Consumer[j⟵1 … n]:
   while (true):
      P(full)
      P(mutexF)
      result ⟵ buf[front].fetch()
      front ⟵ (front+1)%n
      V(mutexF)
      V(empty)
      consume result
```

… preferiríamos …



```
monitor bufferM

process Producer[j⟵1 … n]:
   while (true):
      produce data
      bufferM.deposit(data)

process Consumer[j⟵1 … n]:
   while (true):
      result ⟵ bufferM.fetch()
      consume result
```

Un **monitor** es un objeto estructurado

Encapsula las variables compartidas que requieren acceso protegido (por el monitor)

Ofrece una interfaz compuesta por métodos:

- puede ser programado sin que se sepa el orden en que sus métodos van a ser llamados

Responsable de su propia corrección:

- la corrección está concentrada en los métodos

Un **monitor** es un objeto estructurado

Encapsula las variables compartidas que requieren acceso protegido (por el monitor)

Ofrece una interfaz compuesta por métodos:

- puede ser programado sin que se sepa el orden en que sus métodos van a ser llamados

Responsable de su propia corrección:

- la corrección está concentrada en los métodos

```
monitor bufferM

process Producer[j⟵1 … n]:
   while (true):
      produce data
      bufferM.deposit(data)

process Consumer[j⟵1 … n]:
   while (true):
      result ⟵ bufferM.fetch()
      consume result
```

Un **monitor** es un objeto estructurado

Encapsula las variables compartidas que requieren acceso protegido (por el monitor)

Ofrece una interfaz compuesta por métodos:

- puede ser programado sin que se sepa el orden en que sus métodos van a ser llamados

Responsable de su propia corrección:

- la corrección está concentrada en los métodos

`bufferM`
protege al *buffer*

```
monitor bufferM

process Producer[j⟵1 … n]:
   while (true):
      produce data
      bufferM.deposit(data)

process Consumer[j⟵1 … n]:
   while (true):
      result ⟵ bufferM.fetch()
      consume result
```

Un **monitor** es un objeto estructurado

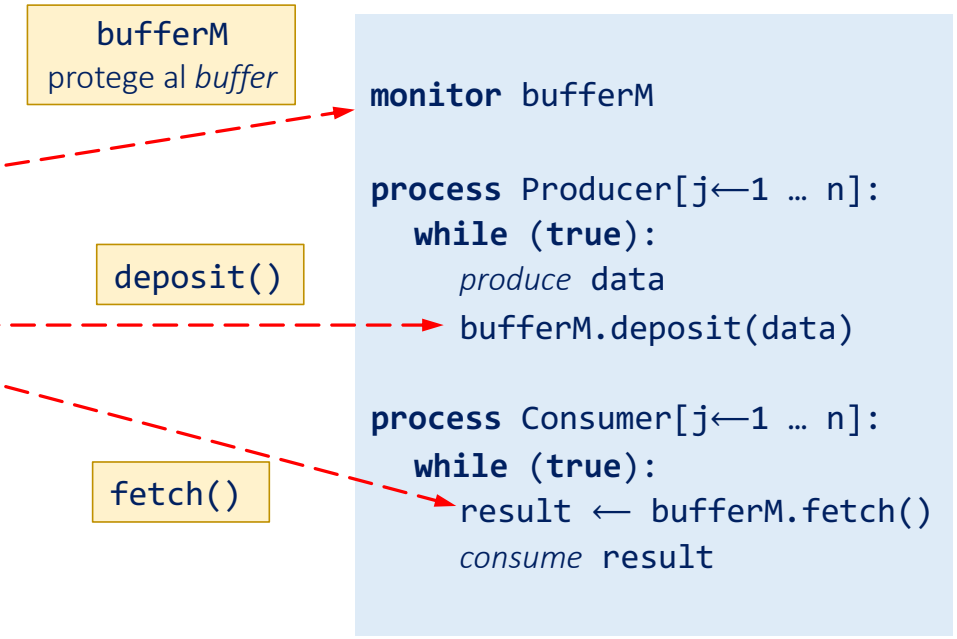
Encapsula las variables compartidas que requieren acceso protegido (por el monitor)

Ofrece una interfaz compuesta por métodos:

- puede ser programado sin que se sepa el orden en que sus métodos van a ser llamados

Responsable de su propia corrección:

- la corrección está concentrada en los métodos



Un **monitor** es un objeto estructurado

Encapsula las variables compartidas que requieren acceso protegido (por el monitor)

Ofrece una interfaz compuesta por métodos:

- puede ser programado sin que se sepa el orden en que sus métodos van a ser llamados

Responsable de su propia corrección:

- la corrección está concentrada en los métodos

`bufferM` protege al *buffer*

`deposit()`

`fetch()`

```
monitor bufferM

process Producer[j←1 … n]:
  while (true):
    produce data
    bufferM.deposit(data)

process Consumer[j←1 … n]:
  while (true):
    result ← bufferM.fetch()
    consume result
```

El orden relativo de las múltiples llamadas a `deposit()`, por productores, y a `fetch()`, por consumidores, puede ser completamente arbitrario

… y no debería incidir en el funcionamiento correcto del *buffer*

Un **monitor** es un objeto estructurado

Encapsula las variables compartidas que requieren acceso protegido (por el monitor)

Contiene variables permanentes —representan su estado— y *sentencias de inicialización* —definen el estado inicial

Ofrece una *interfaz* compuesta por métodos (sólo las firmas de los métodos forman la interfaz):

- puede ser programado sin que se sepa el orden en que sus métodos van a ser llamados

Responsable de su propia corrección:

- la corrección está concentrada en los métodos

```
monitor bufferM:
    someType[n] buffer
    int front ← 0, rear ← 0, count ← 0
    condition notFull, notEmpty
    deposit(someType data):
        while (count = n):
            notFull.wait()
        buffer[rear] ← data
        rear ← (rear+1)%n; count ← count+1
        notEmpty.signal()
    someType fetch():
        while (count = 0):
            notEmpty.wait()
        someType result ← buffer[front]
        front ← (front+1)%n; count ← count-1
        notFull.signal()
        return result
```

Un **monitor** es un objeto estructurado

Encapsula las variables compartidas que requieren acceso protegido (por el monitor)

Contiene variables permanentes —representan su estado— y *sentencias de inicialización* —definen el estado inicial

Ofrece una *interfaz* compuesta por métodos (sólo las firmas de los métodos forman la interfaz):

- puede ser programado sin que se sepa el orden en que sus métodos van a ser llamados

Responsable de su propia corrección:

- la corrección está concentrada en los métodos

```
monitor bufferM:
    someType[n] buffer
    int front ← 0, rear ← 0, count ← 0
    condition notFull, notEmpty
    deposit(someType data):
        while (count = n):
            notFull.wait()
        buffer[rear] ← data
        rear ← (rear+1)%n; count ← count+1
        notEmpty.signal()
    someType fetch():
        while (count = 0):
            notEmpty.wait()
        someType result ← buffer[front]
        front ← (front+1)%n; count ← count-1
        notFull.signal()
        return result
```

Un **monitor** es un objeto estructurado

Encapsula las variables compartidas que requieren acceso protegido (por el monitor)

Contiene variables permanentes —representan su estado— y *sentencias de inicialización* —definen el estado inicial

Ofrece una *interfaz* compuesta por métodos (sólo las firmas de los métodos forman la interfaz):

- puede ser programado sin que se sepa el orden en que sus métodos van a ser llamados

Responsable de su propia corrección:

- la corrección está concentrada en los métodos

```
monitor bufferM:
    someType[n] buffer
    int front ← 0, rear ← 0, count ← 0
    condition notFull, notEmpty
    deposit(someType data):
        while (count = n):
            notFull.wait()
        buffer[rear] ← data
        rear ← (rear+1)%n; count ← count+1
        notEmpty.signal()
    someType fetch():
        while (count = 0):
            notEmpty.wait()
        someType result ← buffer[front]
        front ← (front+1)%n; count ← count-1
        notFull.signal()
        return result
```

Un **monitor** es un objeto estructurado

Encapsula las variables compartidas que requieren acceso protegido (por el monitor)

Contiene variables permanentes —representan su estado— y *sentencias de inicialización* —definen el estado inicial

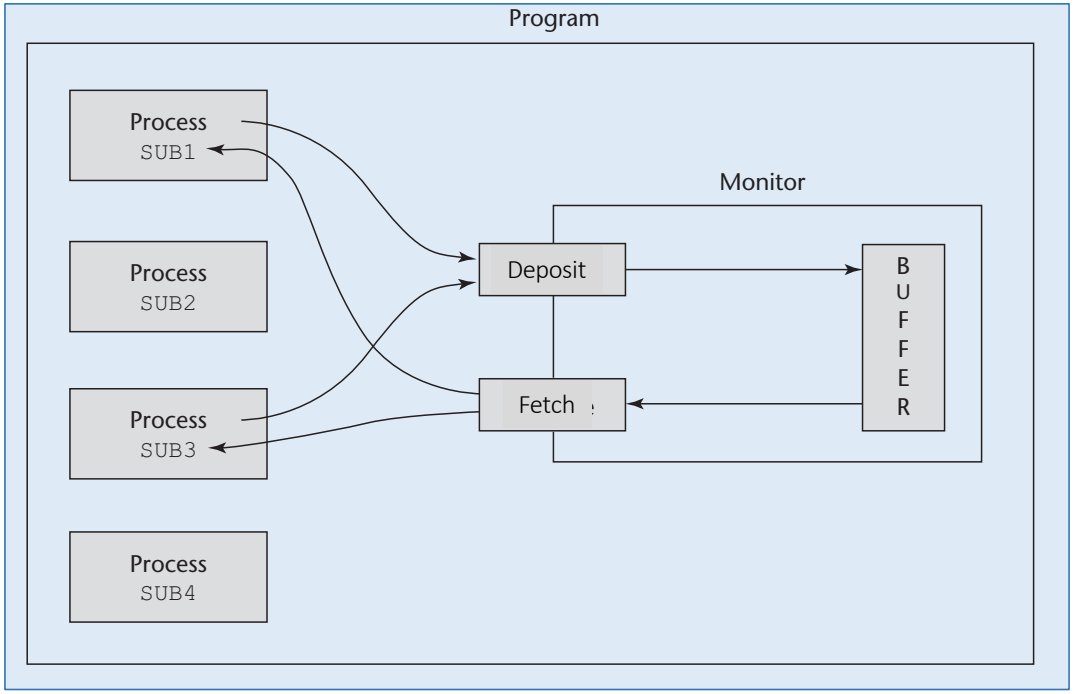
Ofrece una *interfaz* compuesta por métodos (sólo las firmas de los métodos forman la interfaz):

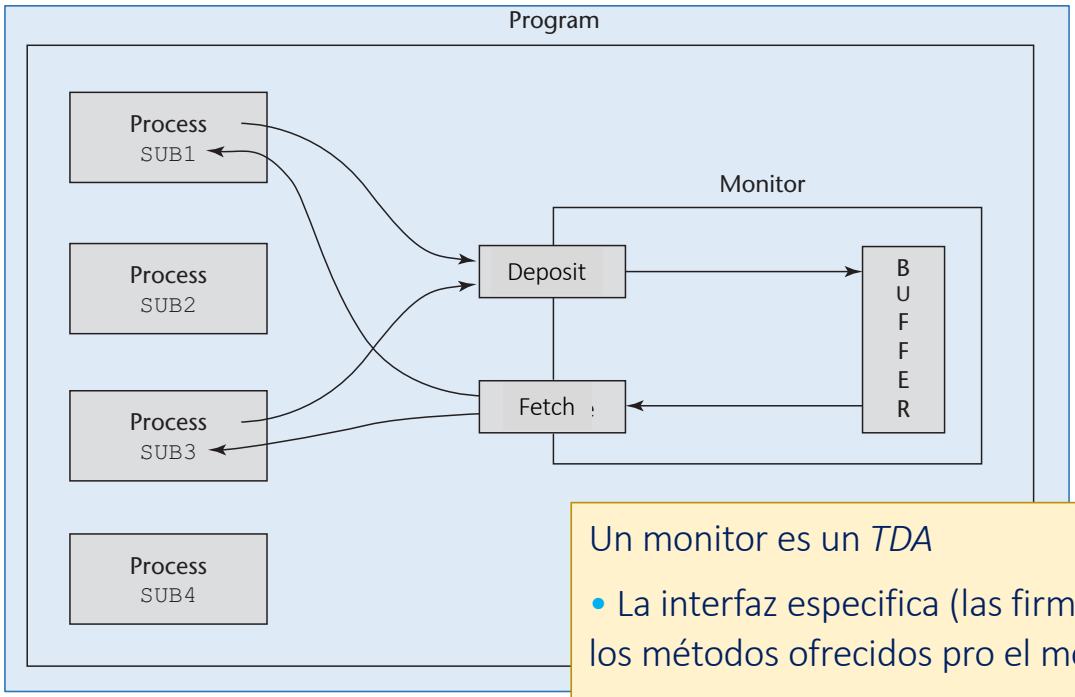
- puede ser programado sin que se sepa el orden en que sus métodos van a ser llamados

Responsable de su propia corrección:

- la corrección está concentrada en los métodos

```
monitor bufferM:
    someType[n] buffer
    int front ← 0, rear ← 0, count ← 0
    condition notFull, notEmpty
    deposit(someType data):
        while (count = n):
            notFull.wait()
        buffer[rear] ← data
        rear ← (rear+1)%n; count ← count+1
        notEmpty.signal()
    someType fetch():
        while (count = 0):
            notEmpty.wait()
        someType result ← buffer[front]
        front ← (front+1)%n; count ← count-1
        notFull.signal()
        return result
```


Program
Process
SUB1
Process
SUB2
Process
SUB3
Process
SUB4
Monitor
Deposit
Fetch
B
U
F
F
E
R



Un monitor es un *TDA*

- La interfaz especifica (las firmas de) los métodos ofrecidos pro el monitor
- El cuerpo contiene las variables permanentes y (el código de) los métodos

La corrección está concentrada en los métodos del monitor (y *no es responsabilidad* de los procesos que usan el monitor)

Dos tipos de sincronización (como ya hemos visto):

- *exclusión mutua* es garantizada por el *tipo de datos* `monitor` (es *implícita*)
- *sincronización por condición* debe ser programada explícitamente en el código de los métodos del monitor

La corrección está concentrada en los métodos del monitor (y *no es responsabilidad* de los procesos que usan el monitor)

Diferencia importante con los semáforos

Dos tipos de sincronización (como ya hemos visto):

- *exclusión mutua* es garantizada por el *tipo de datos* **`monitor`** (es *implícita*)

Otra diferencia importante con los semáforos

- *sincronización por condición* debe ser programada explícitamente en el código de los métodos del monitor

… usando variables de tipo **`condition`**

```
monitor bufferM

process Producer[j⟵1 … n]:
   while (true):
      produce data
      bufferM.deposit(data)

process Consumer[j⟵1 … n]:
   while (true):
      result ⟵ bufferM.fetch()
      consume result
```

```
monitor bufferM:
    someType[n] buffer
    int front ← 0, rear ← 0, count ← 0
    condition notFull, notEmpty
    deposit(someType data):
        while (count = n):
            notFull.wait()
        buffer[rear] ← data
        rear ← (rear+1)%n; count ← count+1
        notEmpty.signal()
    someType fetch():
        while (count = 0):
            notEmpty.wait()
        someType result ← buffer[front]
        front ← (front+1)%n; count ← count-1
        notFull.signal()
        return result
```

Exclusión mutua es garantizada por el tipo de datos **monitor**

- Un método de un monitor es llamado (explícitamente) por un proceso
- Si varios métodos de un mismo monitor son llamados por varios procesos, *la implementación del monitor asegura que estos métodos son ejecutados bajo exclusión mutua*

Es reponsabilidad del lenguaje, librería o sistema operativo

Sincronización por condición debe ser programada explícitamente en el código de los métodos del monitor usando variables de tipo **`condition`**

Variables de tipo **`condition`**

`condition` ***`c`*** :

- **`condition`** es un (nuevo) tipo de datos
- el valor de ***`c`*** es una cola de procesos suspendidos

Dos propósitos:

- suspender un proceso hasta que el estado del monitor satisfaga alguna condición — `c.wait()`
- reanudar un proceso suspendido cuando la condición se hace verdadera — `c.signal()`

```
monitor bufferM

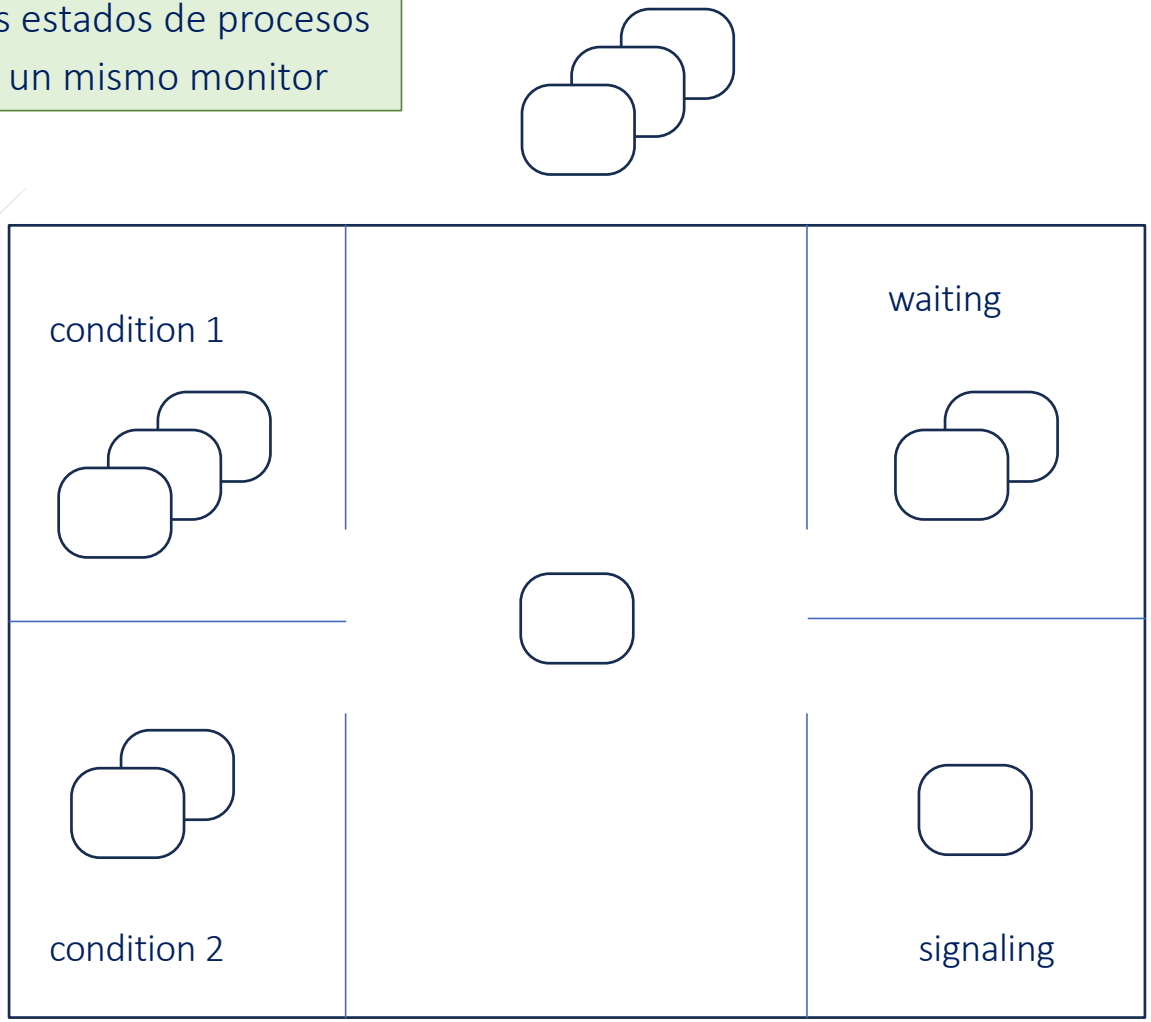
process Producer[j⟵1 … n]:
   while (true):
      produce data
      bufferM.deposit(data)

process Consumer[j⟵1 … n]:
   while (true):
      result ⟵ bufferM.fetch()
      consume result
```

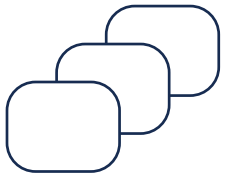
```
monitor bufferM:
    someType[n] buffer
    int front ← 0, rear ← 0, count ← 0
    condition notFull, notEmpty
    deposit(someType data):
        while (count = n):
            notFull.wait()
        buffer[rear] ← data
        rear ← (rear+1)%n; count ← count+1
        notEmpty.signal()
    someType fetch():
        while (count = 0):
            notEmpty.wait()
        someType result ← buffer[front]
        front ← (front+1)%n; count ← count-1
        notFull.signal()
        return result
```

- Si durante la ejecución de **`deposit()`** se ejecuta **`notFull.wait()`**, entonces el proceso productor que llamó a **`deposit()`** queda suspendido en la cola de **`notFull`**
- Cuando durante la ejecución de **`deposit()`** se ejecute **`notEmpty.signal()`**, se reanudará el primer proceso consumidor que esté suspendido en la cola de **`notEmpty`**

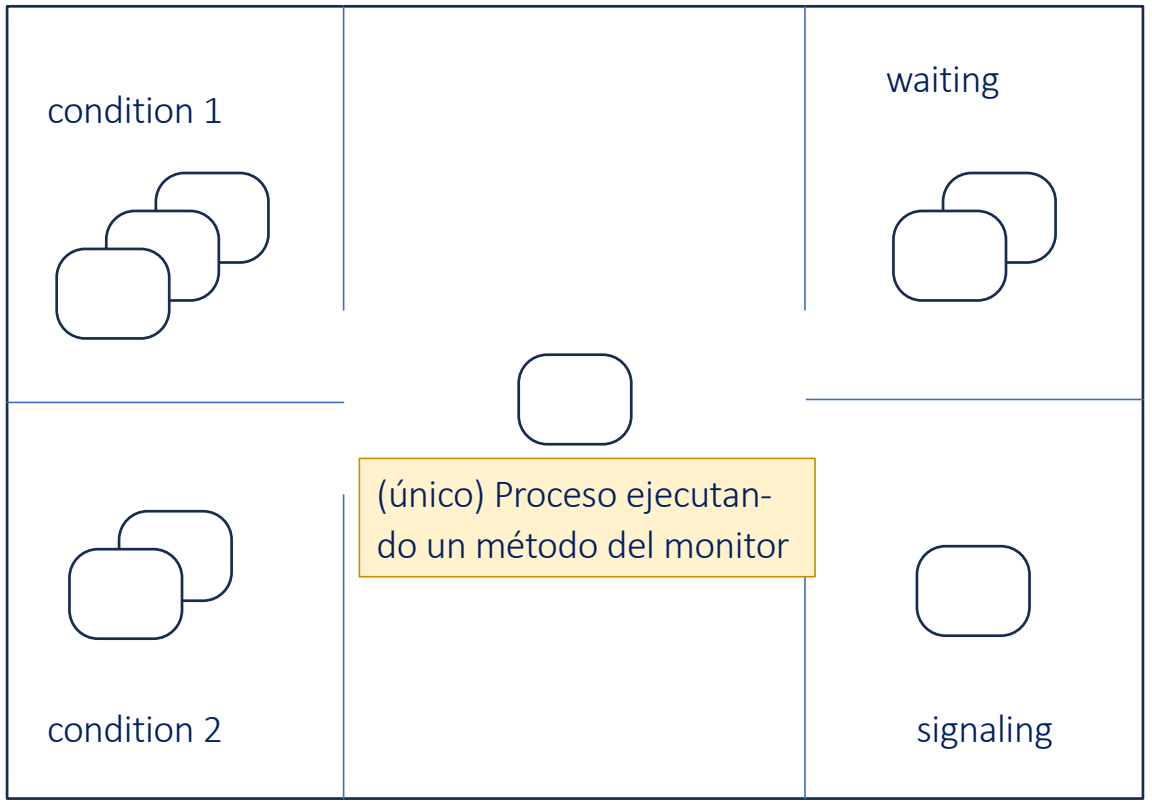
Posibles estados de procesos usando un mismo monitor



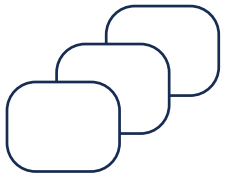
Posibles estados de procesos usando un mismo monitor



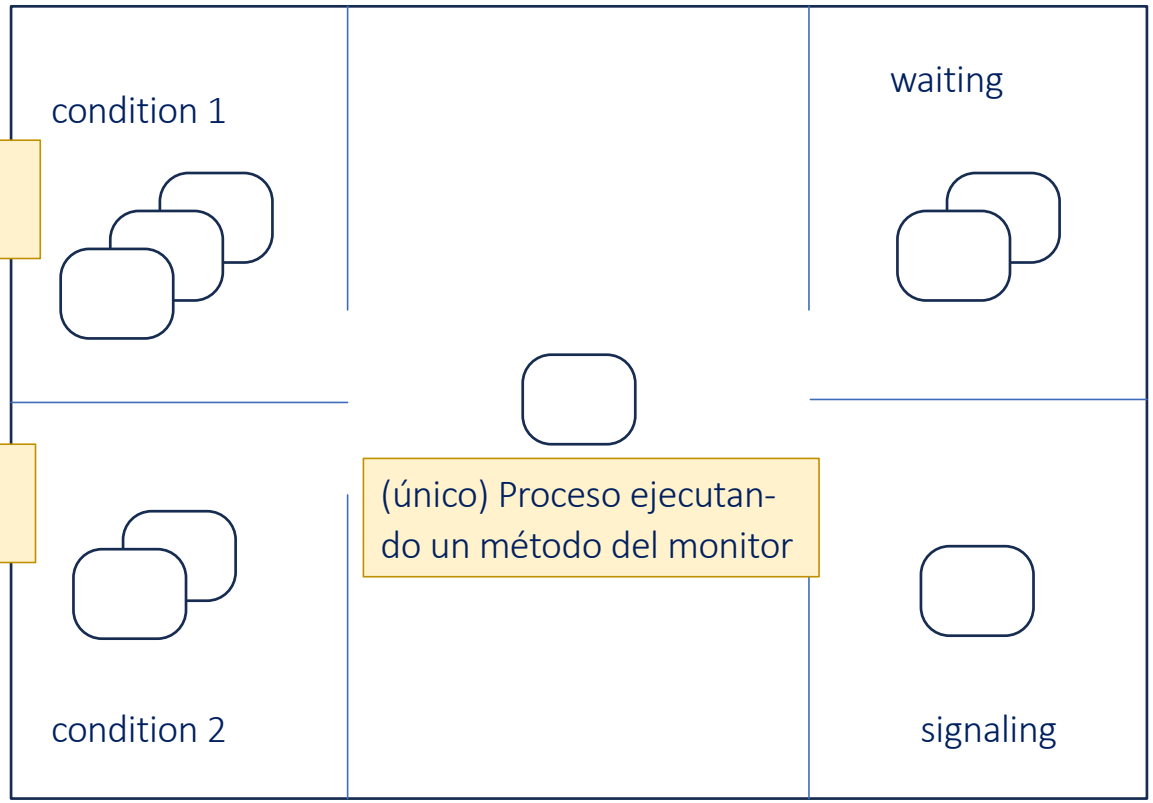
Procesos que han llamado a un método del monitor, pero aún no tienen acceso



Posibles estados de procesos usando un mismo monitor



Procesos que han llamado a un método del monitor, pero aún no tienen acceso

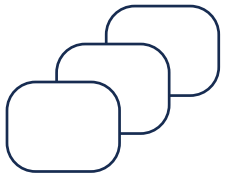


Procesos esperando que se cumpla una condición

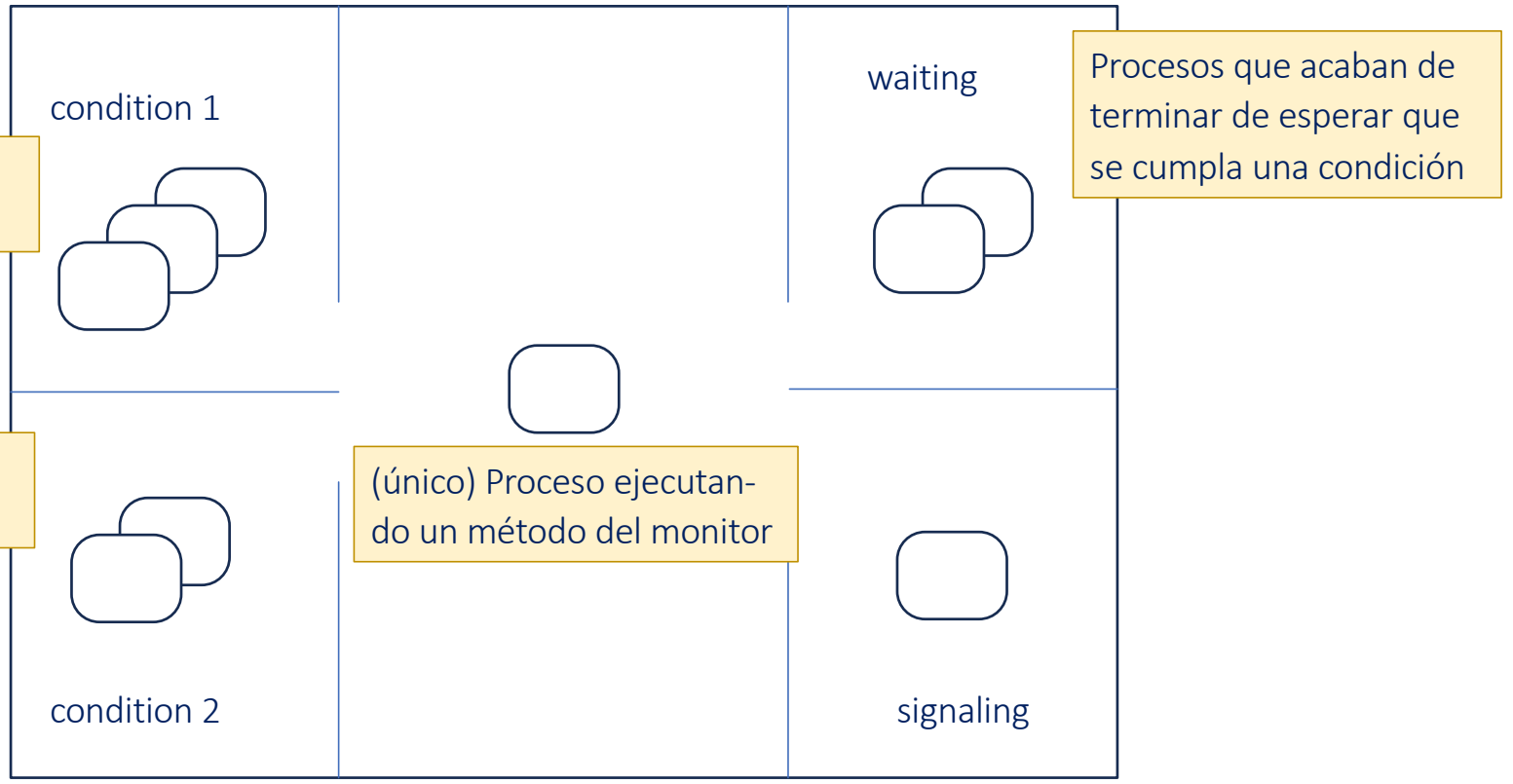
Procesos esperando que se cumpla una condición

(único) Proceso ejecutan-do un método del monitor

Posibles estados de procesos usando un mismo monitor



Procesos que han llamado a un método del monitor, pero aún no tienen acceso



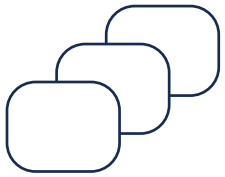
Procesos esperando que se cumpla una condición

Procesos que acaban de terminar de esperar que se cumpla una condición

Procesos esperando que se cumpla una condición

(único) Proceso ejecutan-do un método del monitor

Posibles estados de procesos usando un mismo monitor



Procesos que han llamado a un método del monitor, pero aún no tienen acceso

Procesos esperando que se cumpla una condición

Procesos que acaban de terminar de esperar que se cumpla una condición

Procesos esperando que se cumpla una condición

(único) Proceso ejecutando un método del monitor

Procesos que acaban de ejecutar la operación *signal* y cedieron su lugar en el monitor

Variables permanentes:

- existen y mantienen sus valores mientras exista el monitor
- inicializadas antes de que se llama a cualquier método
- compartidas por todos los métodos del monitor

Sentencias de inicialización:

- ejecutadas al crear el monitor

Sentencias del monitor —de inicialización y de los métodos:

- no tienen acceso a las variables de los procesos

Lectores y escritores

… dos tipos de procesos que comparten una base de datos (BD):

- los lectores ejecutan transacciones que sólo examinan (e.g., leen) la BD
- los escritores la actualizan (e.g., escriben)

La BD está inicialmente en un estado consistente:

- las transacciones, ejecutadas aisladamente, la transforman de un estado consistente a otro

Un escritor debe tener acceso exclusivo a la BD

Si ningún escritor tiene acceso a la BD, entonces cualquier número de lectores puede ejecutar transacciones concurrentemente

La BD es compartida

… pero no conviene protegerla encerrándola en un monitor:

- los lectores no podrían tener acceso concurrente a la BD

Usamos un monitor para dirimir el acceso a la BD:

- los procesos avisan al monitor cuando quieren tener acceso a la BD

… y cuando han terminado de usarla

… y eso es todo lo que a los procesos les interesa y tienen que saber

Los lectores llaman al método `requestRead()` del monitor

… y los escritores a `requestWrite()`

Los lectores llaman a `releaseRead()`

… y los escritores a `releaseWrite()`

¿Qué código deben ejecutar estos métodos?

```
monitor r&wM

process reader[j←1 … n]:
  while (true):
    r&wM.requestRead()
    leer la BD
    r&wM.releaseRead()

process writer[j←1 … m]:
  while (true):
    r&wM.requestWrite()
    escribir la BD
    r&wM.releaseWrite()
```

Estas cuatro llamadas a los métodos del monitor

… por parte de múltiples procesos lectores y múltiples procesos escritores

… pueden ocurrir en cualquier orden

El monitor **r&wM** no tiene ningún control sobre el orden en que ocurren las llamadas

… sin embargo, debe poder determinar cuáles puede atender completamente

… cuáles deben suspender al proceso que la hizo

… y cuáles reanudan procesos que están suspendidos

```
monitor r&wM

process reader[j←1 … n]:
  while (true):
    r&wM.requestRead()
    leer la BD
    r&wM.releaseRead()

process writer[j←1 … m]:
  while (true):
    r&wM.requestWrite()
    escribir la BD
    r&wM.releaseWrite()
```

```
monitor r&rM:
    int nr ← 0, nw ← 0
    condition OKtoRead, OKtoWrite

    requestRead():
        while (nw > 0):
            OKtoRead.wait()
        nr ← nr+1

    releaseRead():
        nr ← nr-1
        if (nr = 0):
            OKtoWrite.signal()

    requestWrite():
        while (nr > 0 ∨ nw > 0):
            OKtoWrite.wait()
        nw ← nw+1

    releaseWrite():
        nw ← nw-1
        OKtoWrite.signal()
        OKtoRead.signalAll()
```

Solución al problema de los lectores y escritores

```
int nr ← 0, nw ← 0,
  dr ← 0, dw ← 0
sem e ← 1, r ← 0, w ← 0

process Reader[i]:
  while (true):
    P(e)
    if (nw > 0):
      dr ← dr+1;
      V(e); P(r)
    nr ← nr+1
    if (dr > 0):
      dr ← dr-1; V(r)
    else: V(e)
    leer
    P(e)
    nr ← nr-1
    if (nr = 0 ∧ dw > 0):
      dw ← dw-1; V(w)
    else: V(e)
```

```
process Writer[j]:
  while (true):
    P(e)
    if (nr > 0 V nw > 0):
      dw ← dw+1;
      V(e); P(w)
    nw ← nw+1
    V(e)
    escribir
    P(e)
    nw ← nw-1
    if (dr > 0):
      dr ← dr-1; V(r)
    else if (dw > 0):
      dw ← dw-1; V(w)
    else: V(e)
```