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 una cantidad cualquiera 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?

Recordemos

En el caso de los productores y consumidores, al usar un monitor conseguimos que

- los productores sólo tienen que llamar al método **`deposit()`** del monitor, y
- los consumidores sólo tienen que llamar al método **`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
```

Similarmente, ahora buscamos un monitor **`r&wM`** tal que:

- los lectores llamen sólo a los métodos **`requestRead()`** y **`releaseRead()`** del monitor
- y los escritores a **`requestWrite()`** y **`releaseWrite()`**



```
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&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

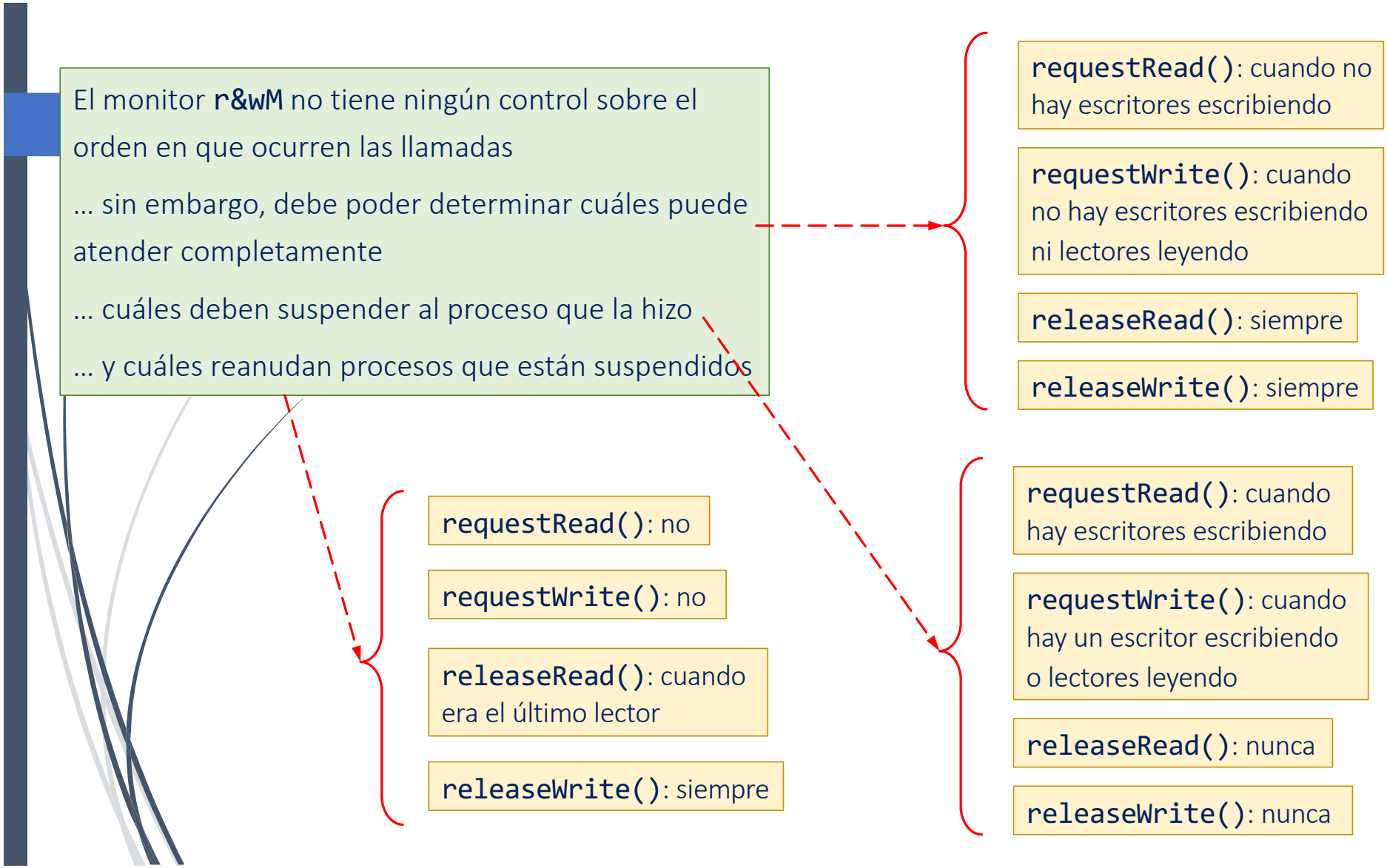
… pueden ocurrir en cualquier orden

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


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
requestRead(): cuando no hay escritores escribiendo
requestWrite(): cuando no hay escritores escribiendo ni lectores leyendo
releaseRead(): siempre
releaseWrite(): siempre
requestRead(): no
requestWrite(): no
releaseRead(): cuando era el último lector
releaseWrite(): siempre
requestRead(): cuando hay escritores escribiendo
requestWrite(): cuando hay un escritor escribiendo o lectores leyendo
releaseRead(): nunca
releaseWrite(): nunca

```
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&wM:
    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()
```

nr: # lectores leyendo

nw: # escritores escribiendo

El monitor debe actualizar correctamente los contadores **nr** y **nw**

signalAll(): Reanuda a todos los procesos suspendidos en la cola correspondiente

```
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&wM:
    int nr ← 0, nw ← 0
    condition OKtoRead, OKtoWrite

    requestRead():
        while (nw > 0 ∨ ¬OKtoWrite.empty()):
            OKtoRead.wait()
        nr ← nr+1
        OKtoRead.signal()

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

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

    releaseWrite():
        nw ← nw-1
        if OKtoRead.empty() then OKtoWrite.signal()
        else OKtoRead.signal()
```

`nr`: # lectores leyendo

`nw`: # escritores escribiendo

`empty()`: `true` si la cola correspondiente está vacía; en otro caso, `false`

Disciplinas de señalización

El proceso que ejecuta **`signal`** está ejecutando dentro del monitor

… y tiene el control del *lock* implícitamente asociado con el monitor

Si **`signal`** reanuda otro proceso, entonces ahora hay dos procesos que podrían ejecutar

… pero a lo más un proceso puede ejecutar a continuación

Disciplinas de señalización

El proceso que ejecuta **`signal`** está ejecutando dentro del monitor

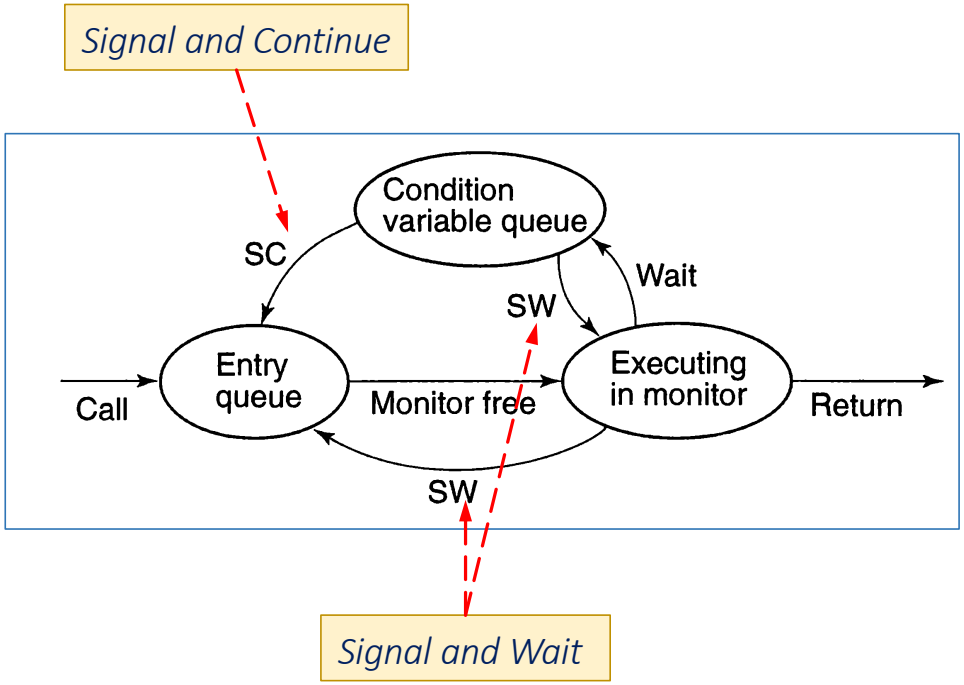
… y tiene el control del *lock* implícitamente asociado con el monitor

Si **`signal`** reanuda otro proceso, entonces ahora hay dos procesos que podrían ejecutar

… pero a lo más un proceso puede ejecutar a continuación

Dos disciplinas de señalización posibles:

- *Signal and Continue*: El proceso que señaliza continúa y el proceso reanudado ejecuta más tarde (va a la cola de procesos esperando adquirir el *lock* del monitor) —es *no-expropiatoria*
- *Signal and Wait*: El proceso que señaliza espera (va a la cola de procesos esperando adquirir el *lock* del monitor) y el proceso reanudado ejecuta ahora—es *expropiatoria*



```
monitor Semaphore:
   int s ← 0
   condition positive
   P():
      while (s = 0):
         positive.wait()
      s ← s-1
   V():
      s ← s+1
      positive.signal()
```

Un monitor que implementa un semáforo

(Algo que probablemente nadie hace en la práctica)

`monitor Semaphore` :

- Bajo *Signal and Continue*: El proceso reanudado ejecuta algún tiempo después del señalizador y por lo tanto *debe volver a verificar el valor de* `s`
- Bajo *Signal and Wait*: El proceso reanudado ejecuta ahora y decrementa `s` —es un semáforo FIFO
- Bajo *Signal and Wait*: El ciclo *while* de `p()` puede cambiarse por un *if* —ya que el proceso reanudado ejecuta inmediatamente

```
monitor Semaphore:
   int s ← 0
   condition positive
   P():
      while (s = 0):
         positive.wait()
      s ← s-1
   V():
      s ← s+1
      positive.signal()
```

Un monitor que implementa un semáforo

(Algo que probablemente nadie hace en la práctica)

`monitor Semaphore` :

- Bajo *Signal and Continue*: El proceso reanudado ejecuta algún tiempo después del señalizador y por lo tanto *debe volver a verificar el valor de* `s`
- Bajo *Signal and Wait*: El proceso reanudado ejecuta ahora y decrementa `s` —es un semáforo FIFO
- Bajo *Signal and Wait*: El ciclo *while* de `p()` puede cambiarse por un *if* —ya que el proceso reanudado ejecuta inmediatamente

`monitor Semaphore_2` :

- emplea la técnica de *paso de la condición* —el señalizador implícitamente pasa la condición de que `s` > 0 al proceso que reanuda; como `s` > 0 no es visible, ningún otro proceso puede verla como tal

```
monitor Semaphore_2:
   int s ← 0
   condition positive
   P():
      if (s = 0):
         positive.wait()
      else:
         s ← s-1
   V():
      if (positive.empty()):
         s ← s+1
      else:
         positive.signal()
```

Semáforo	Monitor
P() puede o no bloquear al proceso	**wait()** siempre bloquea al proceso
V() siempre tiene un efecto	**signal()** no tiene efecto si la cola está vacía
V() desbloquea un proceso bloqueado arbitrario	**signal()** desbloquea el proceso al comienzo de la cola
Un proceso desbloqueado por **V()** reanuda su ejecución inmediatamente	Un proceso desbloquedo por **signal()** debe esperar a que el proceso que ejecutó **signal()** deje el monitor

Invariantes

El programador de un monitor no sabe el orden en que los métodos del monitor van a ser llamados

Es útil definir un **invariante**:

- un predicado que sea verdadero independientemente del orden de ejecución
- especifica los estados válidos de las variables permanentes cuando ningún proceso tiene acceso a ellas
- las sentencias de inicialización establecen el invariante
- cada método debe mantenerlo

Invariantes

El programador de un monitor no sabe el orden en que los métodos del monitor van a ser llamados

Es útil definir **invariante**:

- un predicado que sea verdadero independientemente del orden de ejecución
- especifica los estados válidos de las variables permanentes cuando ningún proceso tiene acceso a ellas
- las sentencias de inicialización establecen el invariante
- cada método debe mantenerlo

Para los monitores `Semaphore` y `Semaphore_2` el invariante es: `s` ≥ 0

Para `bufferM`, es: `rear` = (`front` + `count`) % `n`

Para `r&wM`, es: (`nr` = 0 $\wedge$ `nw` = 0) $\wedge$ `nw` ≤ 1

Solución al problema de los lectores y escritores usando semáforos


```
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)
```