Tema 6. Monitores

1. Introducción

- Los semáforos nos permiten resolver problemas de sincronización y exclusión mutua, pero…
- Tienen difícil aplicación en problemas complejos:
  - Basados en variables globales, impiden diseño modular adecuado.
  - ¿Qué semáforos se están utilizando para sincronización y cuáles para exclusión mutua?
  - Operaciones wait y signal dispersas. La supresión/adición de una nueva puede provocar *interbloqueos*.
  - No nos protege sobre el uso indebido de las variables en tiempo de compilación.
  - Mantenimiento costoso.
  - En definitiva: **propensos a errores**

1.1. Comparativa entre Semáforos y Monitores

- **Semáforos:**
  - Recursos globales
  - Código disperso por los procesos
  - Las operaciones sobre los recursos no están restringidas

- **Monitores:**
  - Recursos locales
  - Código concentrado
  - Los procesos invocan los procedimientos públicos

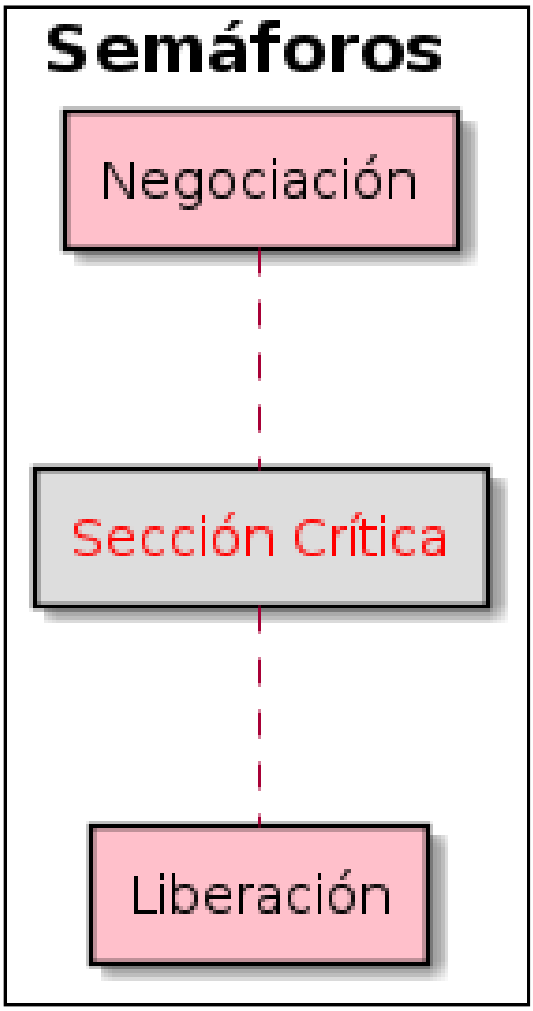

Figure 1: Comparativa semáforo/monitor.

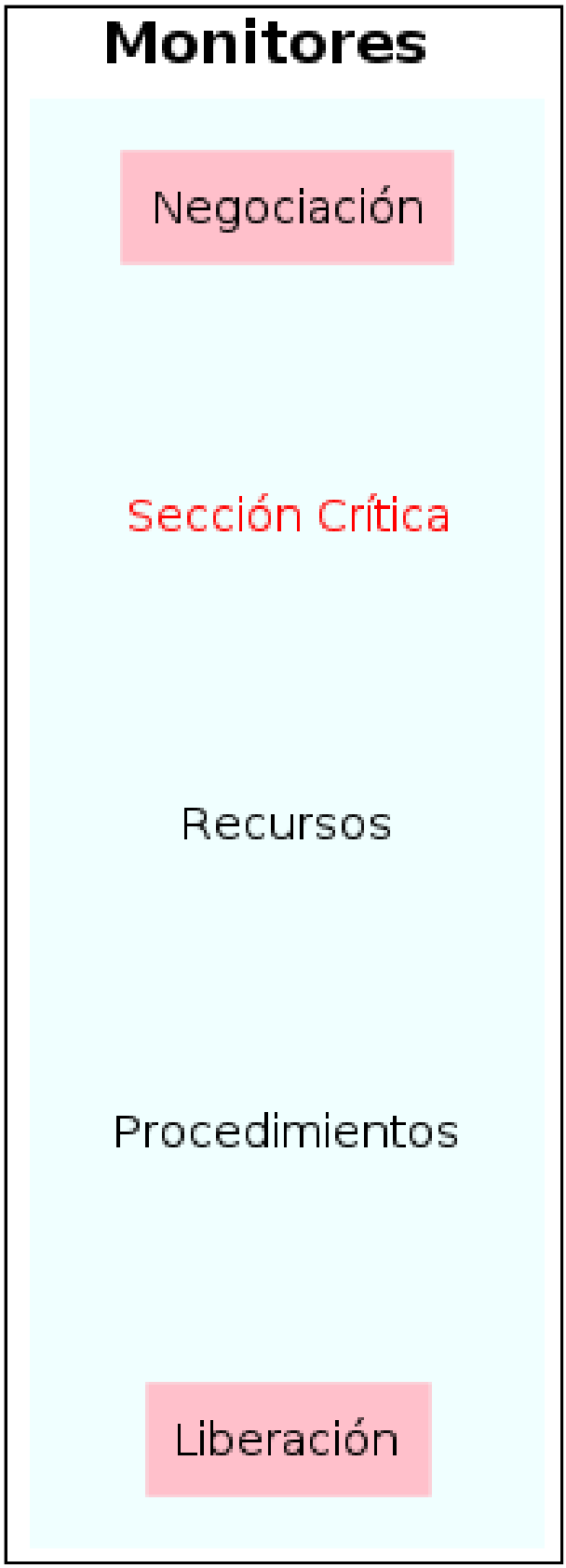

Figure 2: Comparativa semáforo/monitor.

2. Concepto y Funcionamiento

Monitor :

Mecanismo de abstracción de datos: agrupan o encapsulan las representaciones de recursos abstractos y proporcionan un conjunto de operaciones que son las únicas que se pueden realizar sobre dichos recursos.

Operaciones :

Un proceso sólo puede acceder a las variables de monitor usando los procedimientos exportados por el monitor.

Exclusión Mutua :

Garantizada. Un *monitor* se construye de forma que la ejecución de los procedimientos del monitor (exportados o no) no se solapa.

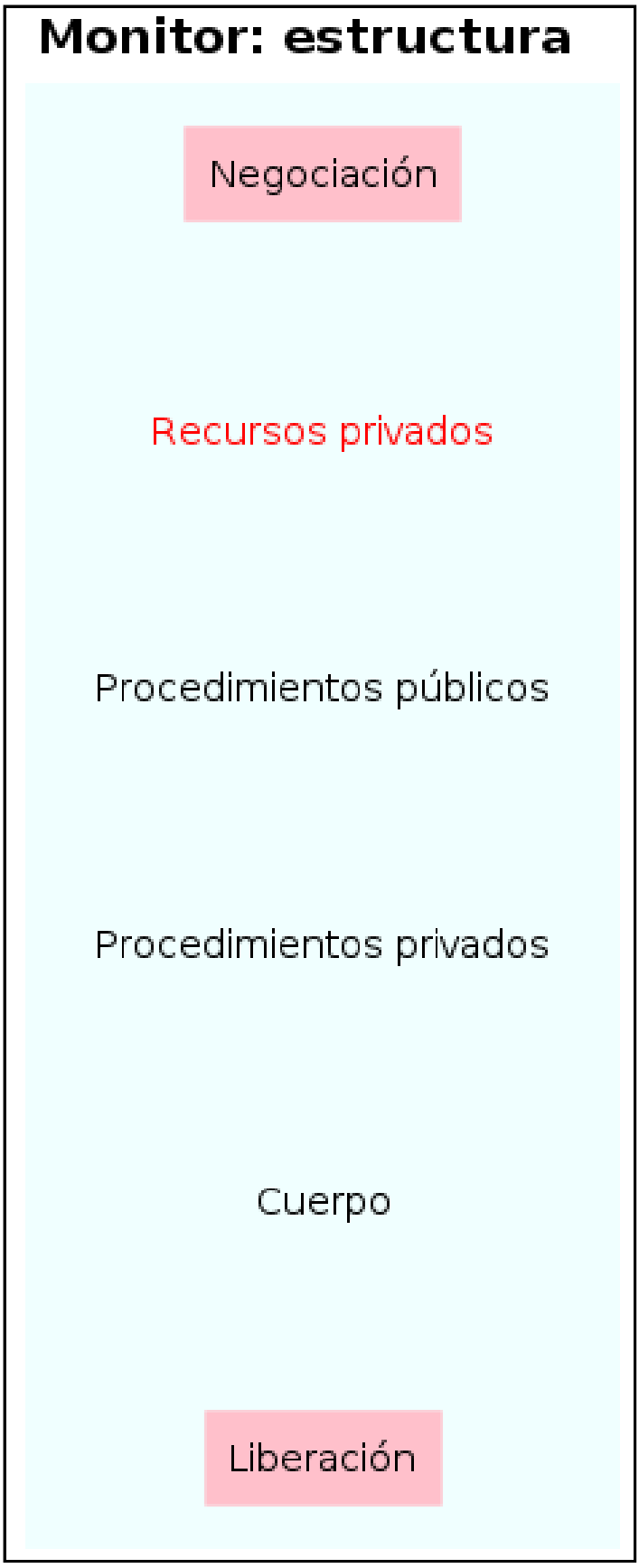
Monitor: estructura
Negociación
Recursos privados
Procedimientos públicos
Procedimientos privados
Cuerpo
Liberación

Figure 3: La estructura de un monitor.

Sintáxis en pseudocódigo

```
monitor  <nombre>:

   (* Variables permanentes, utilizadas para almacenar el estado
   interno del recurso, así como el estado de algún procedimiento
   interno *)
   variables locales

   (* Lista de los procedimientos que pueden invocar los procesos
   activos que accedan al monitor. *)
   export procedimientos exportados

   (* Implementación de los Procedimientos públicos y privados del
   monitor *)
   procedure proc1(parámetros):
       variables locales;
       # código del procedimiento

   procedure proc2(parámetros):
       variables locales;
       # código del procedimiento

```

Ejemplo

- Supongamos que varios procesos deben incrementar el valor de una variable compartida (en exclusión mutua) y poder examinar su valor en cualquier momento. Para ello definiremos un monitor que encapsulará dicha variable compartida.

```
1: monitor incremento:
2:    integer counter = 0
3:    export add, value;
4:    procedure add:
5:         counter = counter + 1
6:    procedure  value:
7:         return counter
8:    ...
9:
10: incremento.add()
11: ...
12: v = incremento.value()
```

- Los procesos, cuando desean comunicarse o sincronizarse utilizan los recurso privados del Monitor mediante invocación de los procedimientos públicos (exportados).
- Casos:
  Monitor libre :
  Si un proceso invoca un procedimiento de un *monitor* y nadie posee el *monitor*, éste proceso bloquea y ocupa el *monitor*, (ejecuta el procedimiento)

  Monitor ocupado :
  Si un proceso invoca algún procedimiento del *monitor* y éste está ocupado entonces el proceso queda bloqueado en una cola asociada al *monitor*. Cuando el proceso poseedor del *monitor* finaliza la ejecución de un procedimiento del *monitor*, se libera el primer proceso bloqueado en ella.

Exclusión Mutua:
Está garantizada por la propia definición del *monitor*

Sincronización (I) :
Se consigue mediante el uso de *Variables de condición* declaradas dentro del propio *monitor*.

Cada una de estas variables dipone de una *cola de bloqueo* de procesos (aparte de la del monitor)

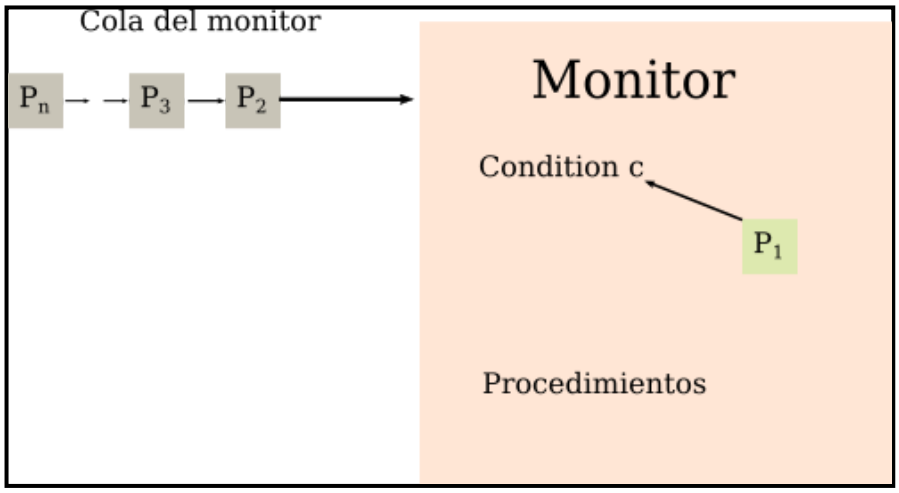

Figure 4: Funcionamiento del monitor.

Sincronización (II) :

Esta cola es una cola **FIFO**:

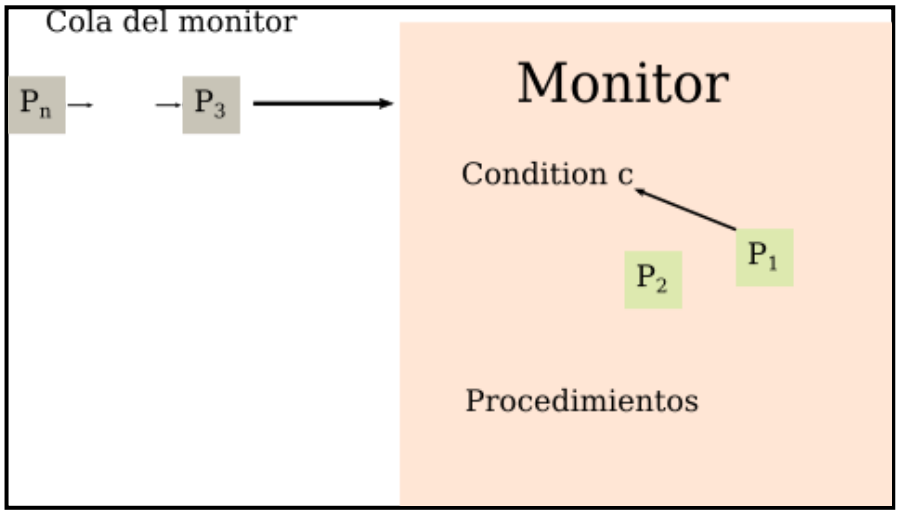

Figure 5: Funcionamiento del monitor.

2.1. Variables Condición: Operaciones

- **Delay(c)** → *Bloqueo de procesos*:

- Bloquea el proceso que realiza la llamada al final de la cola asociada a la *variable condición c*
- Libera la exclusión mutua antes de bloquearse

- **Resume(c)** → *Desbloqueo de procesos*:
  - Extrae el proceso que se encuentra en la cabeza de la cola y lo prepara para su ejecución
  - Cola vacía: operación nula (null)

- **Empty(c)** → *Comprobación de la cola*:
  - Devuelve un valor booleano indicando si la cola de la *variable condición* está vacía o no

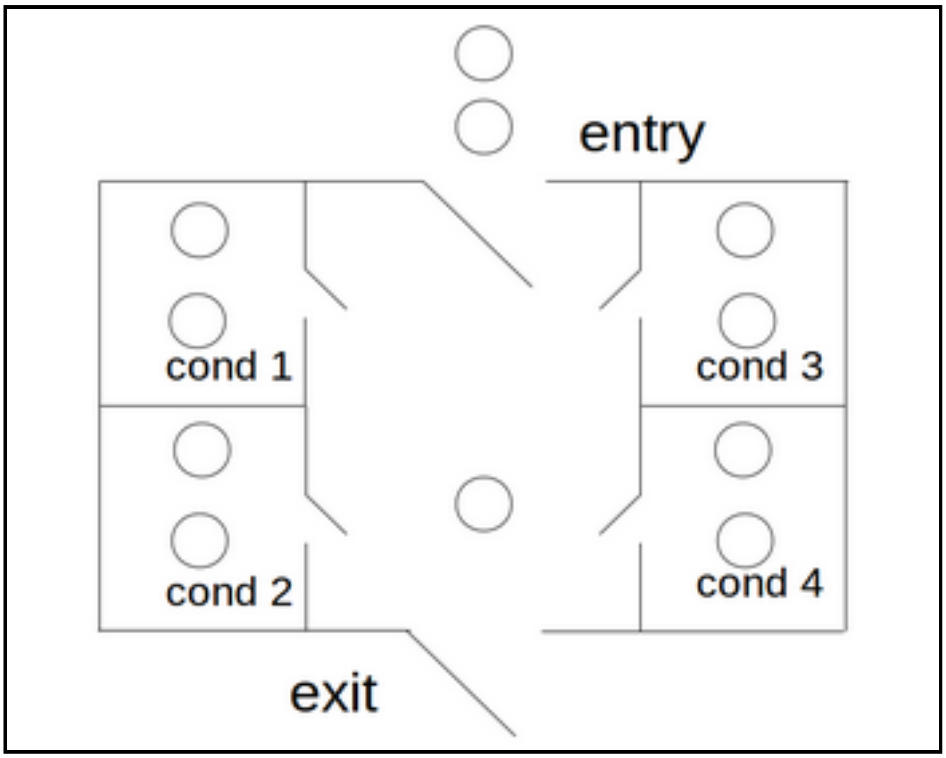

Figure 6: Funcionamiento del monitor.

2.2. Funcionamiento del monitor

Variables de condición

- Si un proceso ejecutando un procedimiento de un *monitor* invoca una operación `delay(c)`, libera el *monitor* y **se bloquea en la cola asociada a esa variable de condición**.
- Cuando un proceso ejecutando un procedimiento de un *monitor* invoca una operación `resume(c)`, se analiza la cola asociada a esa *variable de condición*, seleccionando al primer proceso bloqueado en ella. Si no hubiese procesos bloqueados la operación no tendrá efecto.
- ¿Qué pasa si se desbloquea a un proceso? ¿Hay entonces 2 procesos ejecutando en el monitor?: Distintos lenguajes implementan distintos comportamientos (políticas) para la función `resume`.

2.3. Especificación de prioridades

Hay tres alternativas habituales:

- Requerimiento de reanudación inmediata (**IRR**): el *proceso bloqueado en la variable condición se debe reanudar inmediatamente*:
  - El proceso que señaliza (**S**) se bloquea inmediatamente y cede el *monitor*
  - La mayor prioridad la tiene el proceso bloqueado en la condición señalizada (**W**)
  - Los bloqueados en la entrada del *monitor* (**E**) deben esperar
  - Abreviadamente: **E** $<$ **S** $<$ **W**

- El proceso **S** acaba y sale del monitor, luego los procesos bloqueados en la condición (**W**) y finalmente los procesos de la entrada (**E**):
  - Abreviadamente: **E** $<$ **W** $<$ **S**

- La "*opción Java*":
  - Abreviadamente: **E = W** $<$ **S**

2.4. Ejemplos clásicos: Barreras

- Monitores con Python:

```
mutex = threading.Lock()
allArrived = threading.Condition(mutex)
arrived = 0

def barrier(n):
                  with mutex:
                                          arrived += 1
                                          if arrived == n:
                                                                  arrived = 0
                                                                  allArrived.not
                                          else:
                                                                  allArrived.wai
```

- Código completo: `barrier.py`

2.5. Ejemplos clásicos: Productor/Consumidor

- Código completo: `producer-consumer.py`

```
# Productor

def append(self, data):
                  with mutex:
                                          while len(buffer) == buffer
                                                                  untilNotFu
                                          buffer.append(data)
                                          whileEmpty.notify()

# Consumidor

def take(self):
                  with mutex:
                                          while not buffer:
                                                                  whileEmpty
                                          data = buffer.popleft() # e
                                          untilNotFull.notify()
                                          return data
```

2.6. Ejemplos clásicos: Lectores/Escritores (prioridad Lectores)

- Código completo: `rw_lock.py`

```
# Lectores

def reader_lock():
                with mutex:
                                        while writing:
                                                        canRead.wa
                                        readers += 1
                                        canRead.notify()       # pu

def reader_unlock():
                with mutex:
                                        readers -= 1
                                        if not readers:
                                                        canWrite.n

# Escritores

def writer_lock():
                with mutex:
                                        while writing or readers:
                                                        canWrite.w
                                        writing = True

def writer_unlock():
                with mutex:
                                        writers = False
                                        canRead.notify()  # desbloq
                                        canWrite.notify()  # y escr
```

2.7. Ejemplos clásicos: Lectores/Escritores (prioridad Escritores)

- El problema anterior presenta un problema de *inanición* → los **escritores podrían no entrar nunca a la sección crítica**
- Una posible solución es dar preferencia al escritor que esté bloqueado (la función `empty` habría que definirla usando, por ejemplo, un contador para los escritores en espera):

```
def reader_lock():
 with mutex:
                        while writing or not empty(canWrite):
                                        canRead.wait() # condición
                        readers += 1
                        canRead.notify()    # pueden entrar otros le
```

2.8. Ejemplos clásicos: Filósofos

```
def pick():
          with mutex:
                                  while picks[i] != 2:
                                                  canEat[i].wait()
                                  picks[left] -= 1
                                  picks[right] -= 1

def release():
                  with mutex:
                                          picks[left] += 1
                                          picks[right] += 1
                                          if picks[left] == 2:        # s
                                                          canEat[left].r
                                          if picks[right] == 2:       # .
                                                          canEat[right].
```

- Código completo: `philosophers.py`
- Una solución similiar en Java: `PhilosopherConditions.java`

3. Implementación de monitores en Posix

- Variables condición: la cabecera `pthread.h` proporciona el tipo `pthread_cond_t`.
- Para hacer un uso sencillo disponemos de:
  - `PTHREAD_COND_INITIALIZER`
  - `int` `pthread_cond_wait` `(pthread_cond_t *cond, pthread_mutex_t *mutex)`
  - `int` `pthread_cond_signal` `(pthread_cond_t *cond)`

- Existen funciones para inicializar con otro tipo de atributos y para destruir las condiciones:
  - `int` `pthread_cond_init` `(...)`

- También disponemos de:
  - `int` `pthread_cond_timedwait` `(...)`
  - `int` `pthread_cond_broadcast` `(...)`

- Veamos un uso habitual de estas funciones para simular *monitores*:

```
1: pthread_mutex_t crj = PTHREAD_MUTEX_INITIALIZER;
2: pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
3: ...
4: tipoOperacion nombreOperacion(argumentos){
5:    ...
6:    pthread_mutex_lock(&crj);
7:    while(predicado)
8:         pthread_cond_wait(&cond,&crj);
9:
10:   // Inicio sección crítica
11:     ...
12:   // Señalar condiciones si las hubiera
13:   // Fin de la sección crítica
14:
15:   pthread_mutex_unlock (&crj);
16:   ...
17:   return valorDevuelto;  // de tipoOperacion
18: }
```

Ejemplo: cola con prioridad (ver código)

3.1. Implementación de monitores con semáforos

Elementos :

son los siguientes

- Para cada monitor:

  semáforo `m_mutex` **:**

  inicializado a 1 para la *entrada al monitor*.

  semáforo `m_next` **:**

  para la *cola de cortesía* inicializado a 0.

entero `m_next_count` **:**

número de procesos bloqueados en la *cola de cortesía* (0 inicialmente)

- Para cada variable de condición (`m_x`):

  semáforo `m_x_sem` **:**

  para la cola de la variable de condición inicializado a 0.

  entero `m_x_count` **:**

  número de procesos bloqueados en la variable de condición (0 inicialmente)

Procedimientos exportados por el monitor:

Cada uno de ellos seguirá este patrón:

```
wait(m_mutex)
#  cuerpo_procedimiento
if m_next_count > 0:
        # Al salir del monitor se debe desbloquear en primer lugar
        # procesos de la cola de cortesía si los hay.
        signal(m_next)
        # Fijate que no hay signal(m_mutex) aquí porque hay cesión
        # exclusión mutua del proceso que sale al proceso que entra
        # cola de cortesía. Dicho de otro modo, las llamadas a wait
        # han de estar equilibradas.
else:
        signal(m_mutex)

```

- Operaciones de sincronización (sobre variables de condición)

```
empty (m_x):
        if m_x_count == 0:
                return (True)
        else:
                return (False)

resume(m_x):
        if m_x_count != 0:
                m_next_count = m_next_count + 1

                signal(m_x_sem) # Libera al siguiente proceso en la
                wait(m_next)    # variable de condición y se bloque


                m_next_count = m_next_count - 1

delay(m_x):
        m_x_count = m_x_count + 1

        # Alguien puede ocupar el monitor: puede ser de la cola de
        # primero o de la entrada normal al monitor
        if m_next_count != 0:
                 signal(m_next)
                 # Libera al siguiente proceso: tienen
                 # preferencia los de la cola de cortesía
        else:
                signal(m_mutex)

        wait(m_x_sem)     # Se bloquea en la variable de condición
        m_x_count = m_x_count - 1
```

4. Ejercicio Propuesto

Una cuenta de ahorros:

- Es compartida entre distintas personas (procesos).
- Cada persona puede sacar o depositar dinero en la cuenta.
- El balance actual de la cuenta es la suma de los depósitos menos la suma de las cantidades sacadas.
- El balance nunca puede ser negativo.

Se pide :

Construir un *monitor* en pseudocódigo para resolver este problema con las operaciones `depositar(cantidad)` y `devolver(cantidad)`.

- El cliente que deposita debe despertar a los clientes que están esperando para sacar
- El cliente que llega para sacar dinero lo saca si existe saldo, independientemente de que haya algún otro proceso esperando porque no hay suficiente dinero para él.

Variante :

Implementad el mismo problema suponiendo que ningún cliente puede colarse para sacar dinero.

5. Ejercicio Propuesto

- Implementad el *problema de la tribu* con *monitores*
  - Los miembros de la tribu cenan en comunidad de una gran olla que contiene **M** misioneros.
  - Cuando un miembro de la tribu quiere comer, él mismo se sirve de la olla, a menos que esté vacía.
  - Si la olla está vacía, entonces despierta al cocinero y espera a que éste llene la olla.
  - Sólo se debe despertar al cocinero cuando la olla está vacía.

5.1. Aclaraciones

- **En ningún caso estas transparencias son la bibliografía de la asignatura**, por lo tanto debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.