

# Tema 6. Monitores

Programación Concurrente

14 de diciembre de 2021

## 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**

## 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

### **Semáforos**



Figura 1: Comparativa semáforo/monitor.

### **Monitores**



Figura 2: Comparativa semáforo/monitor.

## 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.

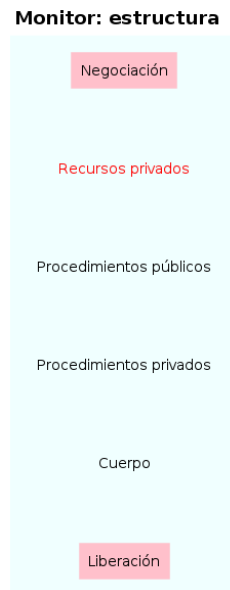


Figura 3: La estructura de un monitor.

### Sintáxis en pseudocódigo

```
1 monitor <nombre>:  
2  
3 (* Variables permanentes, utilizadas para almacenar el estado  
4 interno del recurso, así como el estado de algún procedimiento  
5 interno *)
```

```

6     variables locales
7
8     (* Lista de los procedimientos que pueden invocar los procesos
9     activos que accedan al monitor. *)
10    export procedimientos exportados
11
12    (* Implementación de los Procedimientos públicos y privados del
13    monitor *)
14    procedure proc1(parámetros):
15        variables locales;
16        # código del procedimiento
17
18    procedure proc2(parámetros):
19        variables locales;
20        # 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 recursos 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 dispone de una *cola de bloqueo* de procesos (aparte de la del monitor)

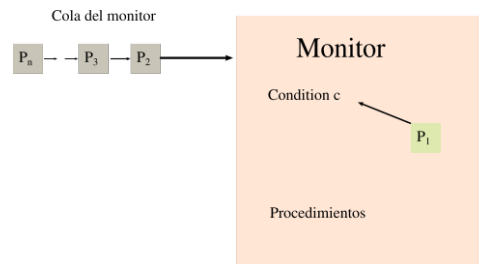


Figura 4: Funcionamiento del monitor.

**Sincronización (II) :** Esta cola es una cola **FIFO**:

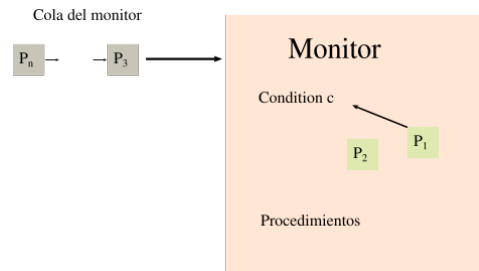


Figura 5: Funcionamiento del monitor.

## 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

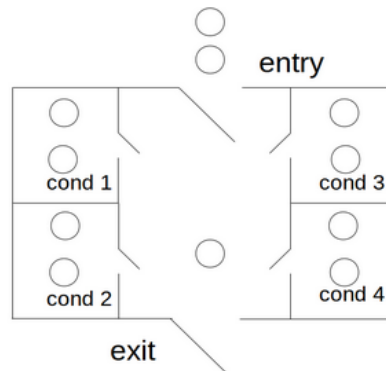


Figura 6: Funcionamiento del monitor.

## 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`.

## 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 señala (**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:  $E = W < S$

## Ejemplos clásicos: Barreras

- Monitores con Python:

---

```
1 mutex = threading.Lock()
2 allArrived = threading.Condition(mutex)
3 arrived = 0
4
5 def barrier(n):
6     with mutex:
7         arrived += 1
```

```

8         if arrived == n:
9             arrived = 0
10            allArrived.notify_all()
11        else:
12            allArrived.wait()

```

---

- Código completo: `barrier.py`

## Ejemplos clásicos: Productor/Consumidor

- Código completo: `producer-consumer.py`

---

```

1  # Productor
2
3  def append(self, data):
4      with mutex:
5          while len(buffer) == buffer.maxlen:
6              untilNotFull.wait()      # condición buffer no lleno
7              buffer.append(data)
8              whileEmpty.notify()
9
10 # Consumidor
11
12 def take(self):
13     with mutex:
14         while not buffer:
15             whileEmpty.wait()      # condición buffer no vacío
16             data = buffer.popleft() # extrae el primero de la lista
17             untilNotFull.notify()
18         return data

```

---

## Ejemplos clásicos: Lectores/Escritores (prioridad Lectores)

- Código completo: `rw_lock.py`

---

```

1  # Lectores
2
3  def reader_lock():
4      with mutex:

```



```

5         while writing:
6             canRead.wait()      # condición hay escritores
7             readers += 1
8             canRead.notify()    # pueden entrar otros lectores
9
10    def reader_unlock():
11        with mutex:
12            readers -= 1
13            if not readers:
14                canWrite.notify() # desbloquea escritor si eres el último
15
16    # Escritores
17
18    def writer_lock():
19        with mutex:
20            while writing or readers:
21                canWrite.wait()  # condición hay lectores o escritores
22                writing = True
23
24    def writer_unlock():
25        with mutex:
26            writers = False
27            canRead.notify()    # desbloquea a lectores
28            canWrite.notify()   # y escritores

```

---

### 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):

---

```

1    def reader_lock():
2        with mutex:
3            while writing or not empty(canWrite):
4                canRead.wait() # condición hay escritores
5            readers += 1

```

## Ejemplos clásicos: Filósofos

---

```
1 def pick():
2     with mutex:
3         while picks[i] != 2:
4             canEat[i].wait()
5         picks[left] -= 1
6         picks[right] -= 1
7
8 def release():
9     with mutex:
10        picks[left] += 1
11        picks[right] += 1
12        if picks[left] == 2:      # sólo desbloquean vecinos con ...
13            canEat[left].notify()
14        if picks[right] == 2:     # ... los dos palillos libres
15            canEat[right].notify()
```

---

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

## 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)**

## 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:

---

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

---

- Operaciones de sincronización (sobre variables de condición)

---

```
1 empty (m_x):
2     if m_x_count == 0:
3         return (True)
4     else:
5         return (False)
6
7 resume(m_x):
8     if m_x_count != 0:
9         m_next_count = m_next_count + 1
10
11     signal(m_x_sem) # Libera al siguiente proceso en la cola de la
12     wait(m_next)    # variable de condición y se bloquea en la cola
13                     # de cortesía
14
15     m_next_count = m_next_count - 1
16
```

```

17 delay(m_x):
18     m_x_count = m_x_count + 1
19
20     # Alguien puede ocupar el monitor: puede ser de la cola de cortesía
21     # primero o de la entrada normal al monitor
22     if m_next_count != 0:
23         signal(m_next)
24         # Libera al siguiente proceso: tienen
25         # preferencia los de la cola de cortesía
26     else:
27         signal(m_mutex)
28
29     wait(m_x_sem)      # Se bloquea en la variable de condición m_x
30     m_x_count = m_x_count - 1

```

---

## 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.

## 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.

### 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.