Sistemas Operativos

Monitores

Curso 2025

Facultad de Ingeniería, UDELAR

Agenda

1. Monitores
2. Variables de condición
3. Dos semánticas para las variables de condición
4. Equivalencia entre monitores y semáforos
5. Problemas de sincronización con monitores
6. Monitores en Java

Monitores

Introducción

- Los semáforos son herramientas de bajo nivel. Es muy factible cometer errores al utilizarlos.
- Algunos aspectos complejos de los semáforos:
  - No se vinculan con los datos que controlan.
  - Se usan para exclusión mutua y para sincronización
  - La lógica depende del orden de invocación de P y V
  - Cualquier omisión u orden incorrecto puede generar un deadlock o un problema de acceso a una región crítica
- Es interesante contar con herramientas de sincronización más estructuradas.

Monitores

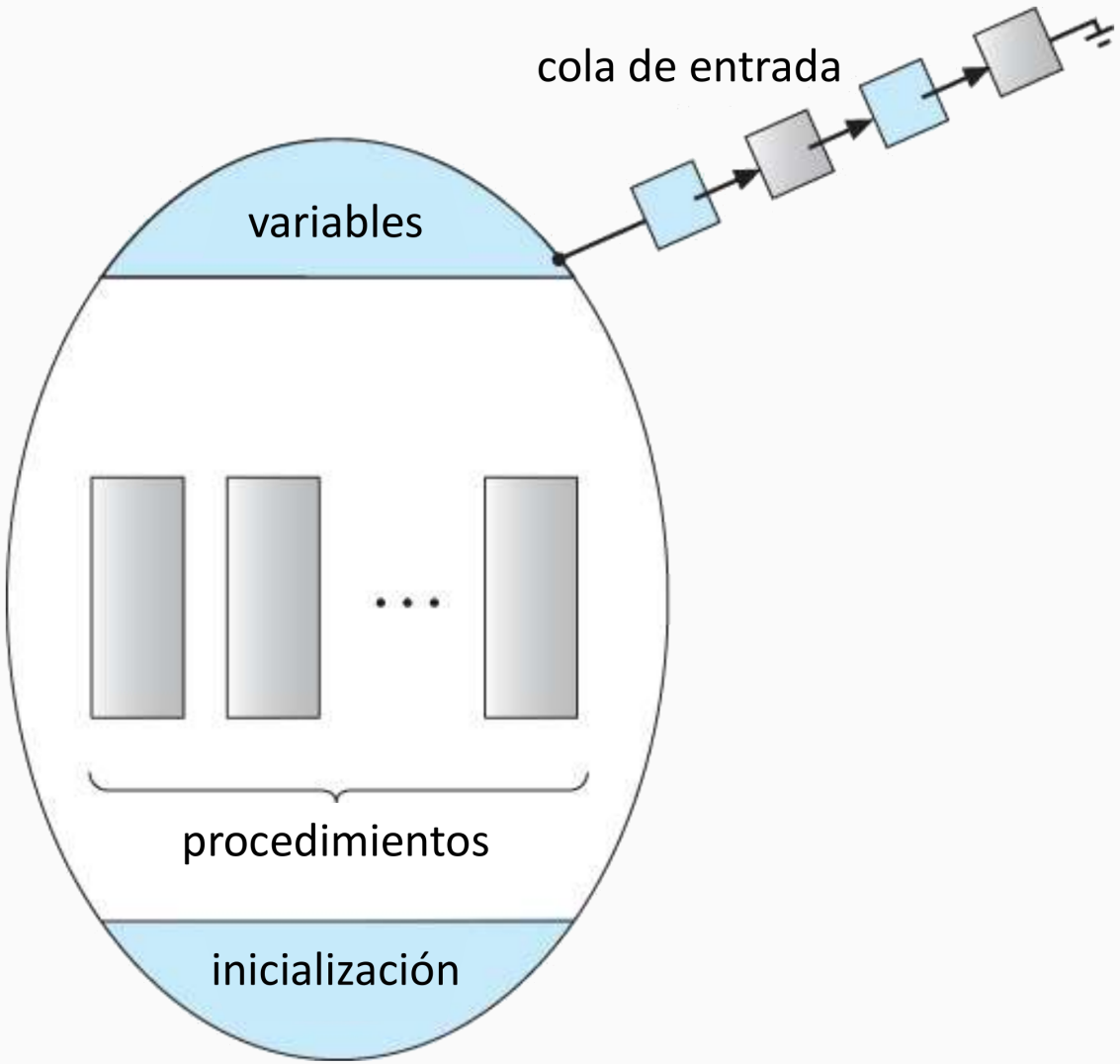
- Es interesante contar con herramientas de sincronización más estructuradas.
- Los monitores son un tipo abstracto de datos diseñado para resolver problemas de sincronización.
  - Encapsulan los datos en forma privada, que se acceden mediante métodos públicos.
  - Los monitores son un tipo de datos, no un hilo de ejecución. Debe implementarse código que utilice el monitor

Formulación

Componentes

- Un conjunto de variables privadas del monitor, globales/compartidas dentro del monitor.
- Un conjunto de procedimientos (públicos, que permiten modificar las variables).
- Un cuerpo de programa (para inicializar las variables).

Monitor


cola de entrada
variables
procedimientos
inicialización

Formulación

- Los monitores aseguran que solo un proceso a la vez está activo en los procedimientos del monitor.
- El monitor funciona como una región crítica: el acceso a las variables del monitor está mutuo-excluido.
- Los procesos que esperan por el acceso al monitor lo hacen en una cola FIFO.

Formulación

- El monitor funciona como una región crítica: el acceso a las variables del monitor está mutuo-excluido.
- Con esta única propiedad no alcanza para resolver los diferentes problemas de sincronización.
- Diferentes procesos/hilos de ejecución pueden necesitar esperar hasta que se cumpla una condición.
- Con este objetivo se utiliza la construcción condition (variables de condición).

Variables de condición

Variables de condición: motivación

- Diferentes hilos de ejecución pueden necesitar esperar hasta que se cumpla una condición.
- Un loop de busy waiting (`while not (cond) do nop`) no puede usarse porque la mutua exclusión impide a otros procesos acceder al monitor para cambiar el valor de la condición.
- Soluciones de espera parcial son complejas de implementar, malgastan recursos y pueden conducir a errores en la lógica.
- Se necesita un mecanismo para señalar al proceso que la condición es verdadera.
- Con este objetivo de implementar la sincronización se utiliza la construcción condition (variables de condición).

Variables de condición

Se definen variables de tipo condition (solo válidas dentro de los monitores), que permiten usar las funciones:

wait() El proceso invocante se bloquea y queda en una cola (FIFO) de procesos bloqueados.

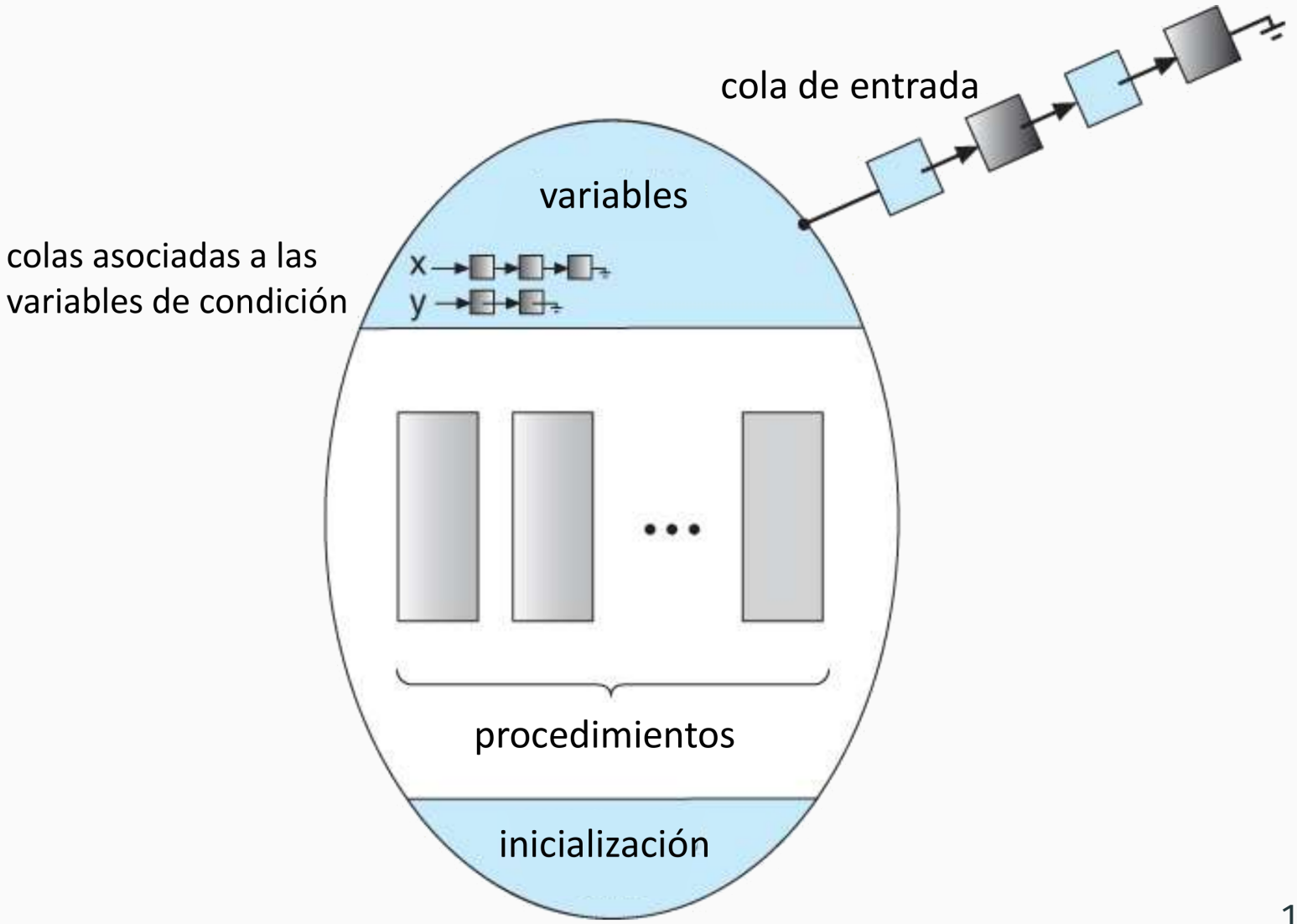
signal() Si la cola de procesos bloqueados no es vacía, despierta al primer proceso bloqueado.

Si se invoca **signal()** pero no existen procesos bloqueados, la operación no tiene efecto.

Variables de condición

- Conceptualmente, una variable de condición está asociada
  a un mutex y a la cola en la que esperan los procesos por la condición.
- Mientras un proceso espera en la cola por la condición, no se considera que ocupe el monitor.
- Otros procesos pueden acceder al monitor y eventualmente cambiar la condición (señalizan que la variable de condición cambió su valor).

Monitor


cola de entrada
variables
colas asociadas a las
variables de condición
x
y
procedimientos
inicialización

Dos semánticas para las variables de condición

Cómo se bloquean las variables de condición

Cuando un procedimiento invoca a un **signal()**, hay dos procesos que pueden seguir activos en el monitor:

1. El proceso que invoca al **signal()**.
2. El proceso que se despierta.

Bloqueo de variables de condición

1. Semántica de Hoare y Hansen
    - El proceso que invoca el **signal()** devuelve el monitor y espera en una cola.
    - El proceso despertado inicia su trabajo en forma inmediata.
2. Semántica de Mesa
    - El proceso que invoca el **signal()** pone el proceso despertado en una cola de listos y sigue trabajando.
    - Cuando termina el proceso invocante, empieza a trabajar el primer proceso de la cola de listos.

Si es posible, ubicar los **signal()** como última instrucción en el código permite igualar ambas semánticas.

- El proceso invocante deja el monitor

Posibles estados de los procesos

- La semántica del monitor queda definida por las prioridades entre las diferentes colas
- Entry queue: formada por los procesos que quieren entrar al monitor. Su prioridad es E.
- Waiting queue: formada por los procesos que ejecutaron **wait()** y ya recibieron un **signal()**. Su prioridad es W.
- Signaller queue: formada por los procesos que ejecutaron **signal()**. Su prioridad es S.
- $E < S < W \rightarrow$ Hoare (prioridad a los que esperan)
- $E < W < S \rightarrow$ Mesa (prioridad a los que ejecutan)
- $E = W < S \rightarrow$ Java

Posibles estados de los procesos

- Si se asume que un proceso ejecuta **wait()** esperando por una condición que se cumple al invocar **signal()**, tiene sentido despertar al proceso inmediatamente para garantizar que la condición se cumple.
- Si se permite continuar la ejecución al proceso que invoca **signal()**, éste podría invalidar la condición en posteriores instrucciones.
- Como contrapartida, se podría bloquear al proceso que invoca **signal()** innecesariamente.

Equivalencia entre monitores y semáforos

Semáforos con monitores

- Implementar las funciones **P()** y **V()** de semáforos con monitores.
- Los procesos se despiertan en orden FIFO.

Semáforos con monitores

```
monitor Semaforo
var cont: int
var bloq: condition

procedure P()
   if cont = 0 then
      bloq.wait();
   end if
   cont := cont - 1;
end procedure

procedure V()
   cont := cont + 1;
   bloq.signal();
end procedure

Begin
   cont := N;
End
End Monitor
```

Monitores con semáforos

- Implementar las variables de condición con semáforos.
- Idea básica:
  - **wait()** análogo a **P()**, el proceso se bloquea hasta que otro proceso ejecute **signal()** sobre la variable de condición
  - **signal()** análogo a **V()**, desbloquea a otro proceso esperando en la variable de condición
- El acceso al monitor debe mutuo-excluirse por un semáforo.
- Se necesita un semáforo para mutex del monitor y un semáforo por cada variable de condición.

Monitores con semáforos

cond cuenta procesos esperando en la variable de condición

```
cond := 0;
INIT(mutex, 1);
INIT(condM, 0);

procedure
Pro_Mon_i()
   P(mutex);
   código Pro_Mon_i;
   V(mutex);
end procedure
```

```
procedure wait()
   cond := cond + 1;
   V(mutex);    ▷ libera el monitor
   P(condM);
   P(mutex);
end procedure

procedure signal()
   if cond > 0 then
      cond := cond - 1;
      V(condM);
   end if
end procedure
```

Problemas de sincronización con monitores

Productor – Consumidor

- Problema Productor – Consumidor con buffer finito (dimensión DBUFF).
- Con el acceso (mutuo-excluido) a los monitores se resuelve el problema del acceso al buffer.
- Con las variables de condición se manejan los casos de borde: no sacar de un buffer vacio (bloqueo de consumidores) y no agregar a un buffer lleno (bloqueo de productores).

Productor - Consumidor con monitores

```
monitor buffer
n, in, out: int
vacio, lleno: condition
b: buffer_t
```

```
n: #elementos en el buffer
in/out: punteros para
        agregar/sacar
```

```
procedure agregar(v:int)
   if (n = DBUFF) then
      lleno.wait();
   end if
   b[in] := v;
   in := (in + 1) mod DBUFF;
   n := n + 1;
   vacio.signal();
end procedure
```

```
procedure sacar(v:int)
   if (n = 0) then
      vacio.wait();
   end if
   v := b[out];
   out := (out + 1) mod DBUF;
   n := n - 1;
   lleno.signal();
end procedure
```

Productor – Consumidor con monitores

```
Begin
   in := out := n := 0;
End

procedure productor()
   repeat
      producir(v);
      buffer.agregar(v);
   until False
end procedure
```

```
procedure consumidor()
   repeat
      buffer.sacar(v);
      consumir(v);
   until False
end procedure
```

Los procedimientos principales tienen un código simple, porque la lógica de sincronización se encapsula en el monitor

Lectores – Escritores

- Solución sin prioridad exclusiva para un grupo de procesos
- Funciona para las dos semánticas (**signal()** al final)

Lectores – Escritores I

```
monitor lect_esc
cantLect: int
lectBloq, escBloq: int
escribiendo: boolean
okLeer, okEscribir: condition


procedure terminar_leer
   cantLect := cantLect - 1;
   if cantLect = 0 then
      okEscribir.signal();
   end if
end procedure
```

El último lector despierta a un escritor

```
procedure empezar_leer
   if escribiendo OR (escBloq > 0) then
      lectBloq := lectBloq + 1;
      okLeer.wait();
      lectBloq := lectBloq - 1;
   end if
   cantLect := cantLect + 1;
   okLeer.signal();
end procedure
```

`okLeer.signal()` despierta a otro lector que pueda estar esperando (despiertan en cascada). Se podría hacer un for para que el primer lector despierte a todos los lectores bloqueados y que el resto no ejecuten el signal.

Lectores – Escritores II

```
procedure empezar_escribir
   if escribiendo OR (cantLect > 0) then
      escBloq := escBloq + 1;
      okEscribir.wait();
      escBloq := escBloq - 1;
   end if
   escribiendo := true;
end procedure
```

Prioridades cruzadas, lectores esperan por (y liberan) escritores y viceversa.

Balancea los procesos de cada tipo.

```
procedure terminar_escribir
   escribiendo := false;
   if lectBloq > 0 then
      okLeer.signal();
   else
      okEscribir.signal();
   end if
end procedure

Begin
   cantLect := 0
   escBloq := lectBloq := 0;
   escribiendo := false;
End
End Monitor
```

Lectores – Escritores III

```
procedure lector
   repeat
      lect_esc.empezar_leer();
      leer();
      lect_esc.terminar_leer();
   until false
end procedure
```

```
procedure escritor
   repeat
      lect_esc.empezar_escribir();
      escribir();
      lect_esc.terminar_escribir();
   until false
end procedure
```

```
Begin
   Cobegin
      lector();
      ...                          ▷ no se puede usar for !
                  ▷ (for secuencializa las ejecuciones)
      lector();
      escritor();
      ...
      escritor();
   Coend
End
```

Filósofos comensales

- Solución que ordena el acceso a los recursos: primero tenedor izquierdo, luego tenedor derecho.

Filósofos comensales I

Se utiliza un monitor por recurso (tenedor).

```
monitor tenedor
enUso: boolean
esperar: condition

procedure levantar
   if enUso then
      esperar.wait();
   end if
   enUso := true;
end procedure

procedure dejar
   enUso := false;
   esperar.signal();
end procedure

Begin
   enUso := false;
End
End Monitor
```

Filósofos comensales II

Se utiliza un monitor para limitar el acceso concurrente a cuatro filósofos.

```
monitor comedor
cantFilosofos: int
lleno: condition


procedure entrar
   if cantFilosofos = 4 then
      lleno.wait();
   end if
   cantFilosofos := cantFilosofos + 1;
end procedure
```

```
procedure salir
   cantFilosofos := cantFilosofos - 1;
   lleno.signal();
end procedure


Begin
   cantFilosofos := 0;
End
End Monitor
```

Tenedor y comedor tienen la misma lógica, pero tenedor es booleano y comedor de conteo.

Filósofos comensales III

```
 var tenedores: array[1..5] of monitor tenedor;

procedure filósofo(i: int)
   izq := i;
   der := (i + 1) MOD 5;
   repeat
      pensar();
      comedor.entrar();
      tenedores[izq].levantar();
      tenedores[der].levantar();
      comer();
      tenedores[izq].dejar();
      tenedores[der].dejar();
      comedor.salir();
   until false
end procedure
```

```
Begin
   Cobegin
      filósofo(1);
      filósofo(2);
      filósofo(3);
      filósofo(4);
      filósofo(5);
   Coend
End
```

Filósofos comensales IV

- Solución que controla el acceso a los recursos: un monitor para la manejar los tenedores.

Filósofos comensales V

```
monitor Controlador
enum
{PENSAR,HAMBRE,COMER}estado[5];
condition self[5];
procedure levantar(i: int)
   estado[i] = HAMBRE;
   test(i);
   if estado[i] != COMER then
      self[i].wait();
   end if
end procedure

procedure bajar(i: int)
   estado[i] = PENSAR;
   test((i+1)%5);
   test((i+4)%5);
end procedure
```

```
procedure test(i: int)
   if (estado[(i+1)%5] != COMER)
&& (estado[(i+4)%5] != COMER) &&
(estado[i] == HAMBRE) then
      estado[i] = COMER;
      self[i].signal();
   end if
end procedure


Begin
   for i = 0 to 4 do
      estado[i] = PENSAR;
   end for
End
End Monitor
```

self permite al filósofo postergar su comida cuando tiene hambre pero no tiene los tenedores disponibles

Filósofos comensales VI

```
procedure filósofo(i: int)
   repeat
      pensar();
      controlador.levantar(i);
      comer();
      controlador.bajar(i);
   until false
end procedure


Begin
   Cobegin
      filósofo(1);
      filósofo(2);
      filósofo(3);
      filósofo(4);
      filósofo(5);
   Coend
End
```

Monitores en Java

Monitores en Java

- Los métodos synchronized de una clase en Java tienen el comportamiento de una función de un monitor.
- Solo un hilo de ejecución puede ejecutar un método (cualquiera) synchronized a la vez.
- Toda clase/objeto tiene un monitor implícito (mutex) que utlizan sus métodos synchronized.
- No existen variables de condición explícitas.
- Los objetos tienen métodos wait y notify (signal) que permiten ser usados como condition.
- wait y notify solo se pueden usar si se está dentro del monitor (dentro de los métodos synchronized del objeto actual).

Monitores en Java

- La implementación es similar a Mesa: el objeto que hace signal sigue en el monitor.
- Al despertar un proceso que espera no hay garantías de que la condición siga siendo válida.
- El orden en que se despiertan los procesos depende de la implementación (no se garantiza orden FIFO).

Productor - Consumidor en Java

```
class Buffer {
    private int n = 0, in = 0, out = 0;
    private int [] b = new int [DBUFF];
    public synchronized void agregar(int v)
                                    throws InterruptedException {
        while (n == DBUFF)
            wait();
        b[in] = v;
        in = (in + 1) % DBUFF;
        n = n + 1;
        notifyAll();
    }
    public synchronized int sacar() throws InterruptedException{
        while (n == 0)
            wait();
        int v = b[out];
        out = (out + 1) % DBUFF;
        n = n - 1;
        notifyAll();
        return v;
    }
}
```