

# Modelos Teóricos de Control de la Concurrencia en Sistemas de Memoria Común

## Tema 3 - Programación Concurrente y de Tiempo Real

Antonio J. Tomeu<sup>1</sup>   Manuel Francisco<sup>2</sup>

<sup>1</sup>Departamento de Ingeniería Informática  
Universidad de Cádiz

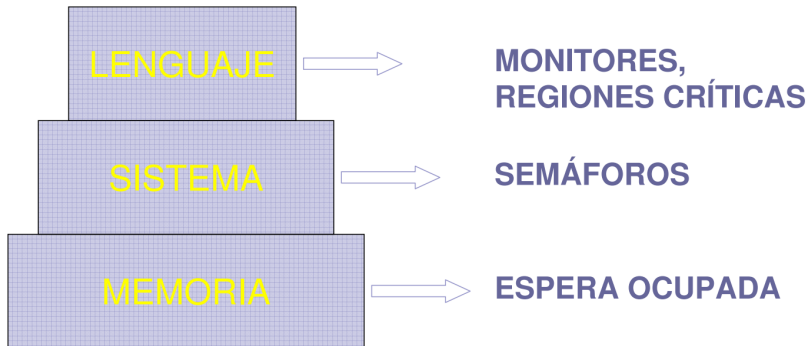
<sup>2</sup>Departamento de CC. de la Computación e I.A.  
Universidad de Granada

PCTR, 2019

# Contenido

1. El Problema de la Exclusión Mutua.
2. Concepto de Sección Crítica.
3. Algoritmos de Espera Ocupada.
4. Semáforos. Protocolos de Exclusión Mutua, Sincronización y Barrera.
5. Regiones Críticas.
6. Monitores.

# Niveles de Exclusión Mutua-Sincronización



# El Problema de la Exclusión Mutua

- ▶  $N$  procesos concurrentes ejecutan un lazo infinito de instrucciones, **divididas en sección crítica y resto de código**. Los programas satisfacen la **exclusión mutua**.
- ▶ Sólo un proceso ejecuta su **sección crítica**. Para ello, se introducen protocolos de entrada y salida, que suelen requerir variables adicionales.
- ▶ Un proceso puede pararse en su zona no crítica, pero no durante la ejecución de los protocolos o de la sección crítica.
- ▶ No habrá **bloqueos**. Si varios procesos desean acceder a **sección crítica**, alguno lo conseguirá eventualmente.
- ▶ No habrá permanencia indefinida de procesos en el **pre-protocolo**, sino que todos deberán tener éxito en el acceso a sección crítica. No habrá **procesos ansiosos**.
- ▶ Si no hay contenciones, un único proceso deseoso de acceder a **sección crítica** lo logrará. Progreso en la ejecución.

- ▶ Se logra la exclusión mutua mediante protocolos de entrada que **consumen ciclos** de CPU.
- ▶ Podemos agrupar las soluciones que utilizan esta técnica en dos grandes grupos:
  - ▶ **Soluciones software**. Las únicas instrucciones atómicas de bajo nivel que consideran son load/store.
  - ▶ Algoritmos de Dekker, Peterson, Knuth, Kesell, Eisenberg-McGuire, Lamport.
  - ▶ **Soluciones hardware**. Utilizan instrucciones específicas de lectura-escritura o intercambio de datos en memoria común cuya ejecución es garantizada como de carácter atómico.

# Intento Incorrecto: Tomando Turnos

```
Turno: integer range 1..2 := 1;
```

```
task body P2 is
begin
  loop
    Resto_código_2;
    loop exit when Turno = 2;
  end loop;
  Sección_Crítica_2;
  Turno := 1;
end loop;
end P2;
```

```
task body P1 is
begin
  loop
    Resto_código_1;
    loop exit when Turno = 1;
  end loop;
  Sección_Crítica_1;
  Turno := 2;
end loop;
end P1;
```

- ▶ Cree una condición de carrera entre dos hilos utilizando una variable común (emVC.java).
- ▶ Controle la exclusión mutua utilizando el protocolo anterior.
- ▶ Verifique que la exclusión mutua se preserva.

- ▶ Son poco eficientes y muy complejas
- ▶ En arquitecturas modernas pueden funcionar incorrectamente
- ▶ Actualmente no se utilizan
- ▶ Se concretan en los siguientes algoritmos:
  - ▶ Algoritmo de Dekker.
  - ▶ Algoritmo de Kesell.
  - ▶ Algoritmo de Peterson.
  - ▶ Algoritmo de Lamport.
  - ▶ Algoritmo de Eisenberg-McGuire.



# Inconvenientes de los Algoritmos de Exclusión Mutua

- ▶ Utilizan **espera ocupada**.
- ▶ Requieren un análisis y programación muy **cuidadosos**.
- ▶ Están muy **ligados a la máquina** en que se implementan.
- ▶ No son transportables.
- ▶ No dan una interfaz directa al programador.
- ▶ Son **poco estructurados**.
- ▶ Son **difíciles de entender** a un número arbitrario de entidades concurrentes.

## Definición

Un semáforo es una variable  $S$  entera que toma valores no negativos y sobre la que se pueden realizar dos operaciones. Son introducidos inicialmente por *Dijkstra* en 1965.

- ▶ Operaciones soportadas:
  - ▶ Wait ( $S$ ): Si  $S > 0$ , entonces  $S := S - 1$ . En otro caso, la entidad concurrente es suspendida sobre  $S$ , en una cola asociada.
  - ▶ Signal ( $S$ ): Si hay una entidad concurrente suspendida, se le despierta. En otro caso,  $S := S + 1$ .
  - ▶ Notación:  
 $\text{Wait } (S) = P(S)$   
 $\text{Signal } (S) = V(S)$

# Semáforos: Generalidades

- ▶ Wait y Signal son **atómicas**.
- ▶ El valor inicial de un semáforo es no negativo.
- ▶ Signal despierta a algún proceso, no especificado por la definición (aunque es habitual un FIFO).
- ▶ Hipótesis de corrección.
- ▶ Semáforos generales:  $S \geq 0$
- ▶ Semáforos binarios:  $S = \{0, 1\}$
- ▶ Ecuaciones de Invariancia: deben ser satisfechas por cualquier implementación del concepto de semáforo:

$$S \geq 0$$

$$S = S_0 + |Signals| - |Waits|$$

## Implementación

```
Type semaforo=record of  
  S: integer;  
  L: lista_de_procesos;  
end;
```

- ▶ La variable  $S$  mantiene el valor actual del semáforo.
- ▶  $L$  es una estructura de datos, en principio dinámica.
- ▶ Cuando  $S = 0$  y un proceso llama a `Wait` es bloqueado y mantenido en la lista  $S$ .
- ▶ Cuando otro proceso señala sobre  $S$ , alguno de los bloqueados sale de  $L$  según algún algoritmo de prioridad.

# Semáforos: Implementación II

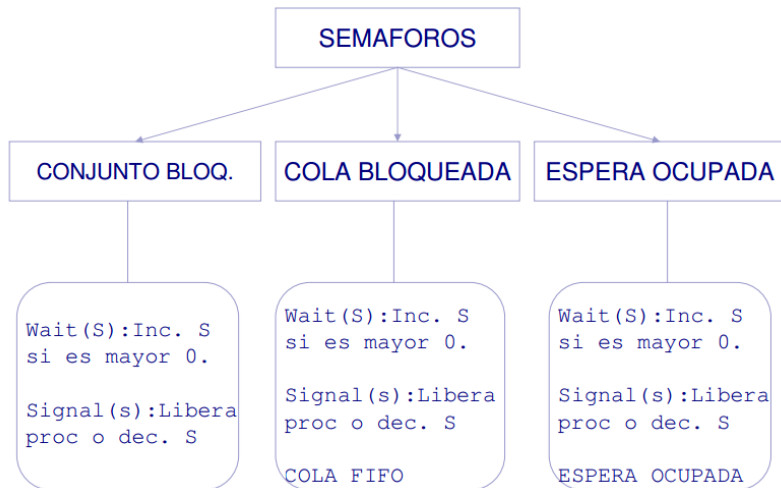
```
procedure inic(var sem:semaphore; s0:integer);  
begin  
    sem.s:=s0;  
    inicializar(sem.L);  
end;
```

```
procedure wait(var sem:semaphore);  
begin  
    if sem.s>0  
    then  
        sem.s:=sem.s-1;  
    else  
        begin  
            sem.L.insertar(proceso);  
            bloquear (proceso);  
        end;  
    end;  
end;
```

# Semáforos: Implementación III

```
procedure signal(var sem:semaphore);  
begin  
  if not sem.L.vacia()  
  then  
    begin  
      sem.L.eliminar (proceso);  
      desbloquear (proceso);  
    end;  
  else  
    sem.s:=sem.s+1;  
  end;
```

# Semáforos: Modalidades de Implementación



- ▶ C dispone de la biblioteca `sem.h` en el marco de las facilidades IPC.
  - ▶ Define conjuntos de semáforos.
  - ▶ Semántica muy diferente a la estándar de Dijkstra.
  - ▶ Son más expresivos.
  - ▶ Funciones `semget`, `semctl` y `semop`.



# Semáforos en C II

```
1  #include<stdio.h>
2  #include<sys/stat.h>
3  #include<sys/types.h>
4  #include<sys/ipc.h>
5  #include<sys/sem.h>
6  #include<string.h>
7
8  #define PERMISOS S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
9  #define TAM 3
10 #define CLAVE (key_t) 666
11
12 void ajustar (struct sembuf *s, int valor, int operacion, int
    flags)
13 {
14     s->sem_num = (short) valor;
15     s->sem_op = operacion;
16     s->sem_flg = flags;
17 }
18
19 int main(void)
20 {
21     int semid;
```

# Semáforos en C III

```
22     struct sembuf mioper[TAM-1];
23     semid = semget(CLAVE, TAM, IPC_CREAT | PERMISOS);
24
25     /*Creacion*/
26     if(semid==-1)
27         printf("Error");
28
29     semctl(semid, 0, SETVAL, 1);
30     ajustar(&(mioper[0]), 0, -1, 0);
31     semop(semid, mioper,1);
32
33     /*Wait=>Decremento del semaforo*/
34     semop(semid, mioper,1);
35
36     /*Wait=>Bloqueo del proceso*/
37     /*Eliminacion del conjunto de semaforos*/
38     semctl(semid, NULL, IPC_RMID);
39 }
```

- ▶ Java no dispuso de semáforos como primitivas hasta 1.5.
- ▶ Java proporciona primitivas de control de la exclusión mutua en el acceso concurrente a objetos (cerrojos).
- ▶ Java permite forzar la ejecución de métodos en exclusión mutua.

# Protocolo de Exclusión Mutua con Semáforos

```
S: semaforo := 1;
```

```
Task body P1 is
```

```
begin
```

```
  loop
```

```
    Resto_1;
```

```
    Wait (S);
```

```
    Seccion_Critica_1;
```

```
    Signal (S);
```

```
  end loop;
```

```
end P1;
```

```
Task body P2 is
```

```
begin
```

```
  loop
```

```
    Resto_2;
```

```
    Wait (S);
```

```
    Seccion_Critica_2;
```

```
    Signal (S);
```

```
  end loop;
```

```
end P2;
```

- ▶ Veamos una implementación del protocolo anterior escrita en Java.
- ▶ Descargue `emSem.java`.
- ▶ Verifique la preservación de la exclusión mutua.

# Protocolo de Sincronización con Semáforos

- Sincronizar P1 y P2 para que P2 espere a la señal de P1.

```
S: semaforo := 0;
```

```
Task body P1 is  
begin  
  loop  
   Codigo;  
    Signal (S);  
   Codigo;  
  end loop;  
end P1;
```

```
Task body P2 is  
begin  
  loop  
   Codigo;  
    Wait (S);  
   Codigo;  
  end loop;  
end P2;
```

# Barreras con Semáforos

## Barrera

Una barrera es un punto del código que ninguna entidad concurrente sobrepasa hasta que **todas** han llegado a ella.

Para dos procesos:

```
Barrera1: semaforo := 0;
```

```
Barrera2: semaforo := 0;
```

Task body P1 is

```
begin
```

```
  loop
```

```
    Signal (Barrera1);
```

```
    Wait (Barrera2);
```

```
   Codigo_Restante;
```

```
  end loop;
```

```
end P1;
```

Task body P2 is

```
begin
```

```
  loop
```

```
    Signal (Barrera2);
```

```
    Wait (Barrera1);
```

```
   Codigo_Restante;
```

```
  end loop;
```

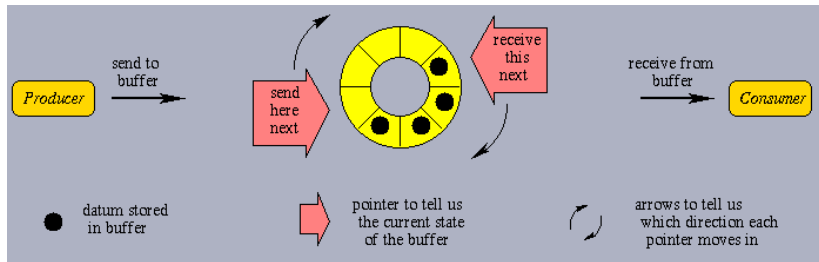
```
end P2;
```

# Sincronización compleja: Productor-Consumidor

- ▶ Idealmente el buffer de comunicación es infinito.
- ▶ El productor puede insertar tantos datos como desee.
- ▶ El consumidor solo puede extraer de un buffer con datos presentes.
- ▶ Usamos un semáforo Elements para controlar al consumidor.
- ▶ Variables de puntero que indican las posiciones donde se inserta o se extrae: In\_Ptr, Out\_Ptr.
- ▶ Acceso a las variables de puntero en exclusión mutua mediante el uso de un semáforo binario: em.



# Sincronización compleja: Productor-Consumidor



# Aproximación Inicial

```
B: array (0..infinity) of integer;  
In_Ptr, Out_Ptr: integer:=0;
```

Task body Productor is

```
I:Integer;  
begin  
  loop  
    producir (I);  
    B(In_Ptr):=I;  
    In_Ptr:= In_Ptr+1;  
  end loop;  
end Productor;
```

Task body Consumidor is

```
I:Integer;  
begin  
  loop  
    I:=B(Out_Ptr);  
    Out_Ptr:= Out_Ptr+1;  
    consumir (I);  
  end loop;  
end Consumidor;
```

# Análisis de la Aproximación Inicial

- ▶ In\_Ptr cuenta el número de elementos insertados en el *buffer*.
- ▶ Out\_Ptr cuenta el número de elementos extraídos del *buffer*.
- ▶ Podemos definir el estado del *buffer* de acuerdo a:

$$E = In\_Ptr - Out\_Ptr$$

- ▶ Puesto que, claramente,  $E$  es siempre mayor o igual que 0, se tiene que:

$$E \geq 0$$

$$E = 0 + In\_Ptr - Out\_Ptr$$

- ▶ Que coinciden con las **ecuaciones de invariancia** de un semáforo.
- ▶ Cabe plantear entonces la sincronización de ambos procesos con el uso de un semáforo que controle las condiciones de acceso al buffer de los procesos productor y consumidor.

# Aproximación con *Buffer* Infinito

```
B: array (0..infinity) of integer;  
In_Ptr, Out_Ptr: integer:=0;  
Elements: semaphore:=0;  
em: semaphore:=1;
```

Task body Productor is

```
I:Integer;  
begin  
  loop  
    producir (I);  
    Wait (em);  
    B(In_Ptr):=I;  
    In_Ptr:= In_Ptr+1;  
    Signal (em);  
    Signal (Elements);  
  end loop;  
end Productor;
```

Task body Consumidor is

```
I:Integer;  
begin  
  loop  
    Wait (Elements);  
    Wait (em);  
    I:=B(Out_Ptr);  
    Out_Ptr:= Out_Ptr+1;  
    Signal (em);  
    consumir (I);  
  end loop;  
end Consumidor;
```

# Aproximación con *Buffer* Finito

```
B: array (0..N-1) of integer;  
In_Ptr, Out_Ptr: integer:=0;  
Elements: semaphore:=0;  
Spaces: semaphore:=N;  
em: semaphore:=1;
```

Task body Productor is

```
I:Integer;  
begin  
  loop  
    producir (I);  
    Wait (Spaces);  
    Wait (em);  
    B(In_Ptr):=I;  
    In_Ptr:= (In_Ptr+1)modN;  
    Signal (em);  
    Signal (Elements);  
  end loop;  
end Productor;
```

Task body Consumidor is

```
I:Integer;  
begin  
  loop  
    Wait (Elements);  
    Wait (em);  
    I:=B(Out_Ptr);  
    Out_Ptr:= (Out_Ptr+1)modN;  
    Signal (em);  
    Signal (Spaces);  
    consumir (I);  
  end loop;  
end Consumidor;
```

- ▶ Veamos una implementación del protocolo anterior escrita en Java.
- ▶ Descargue `prodCon.java`.
- ▶ Añada un semáforo de control de recursos comunes en exclusión mutua.
- ▶ Añada semáforos de sincronización.

# Inconvenientes de los Semáforos

- ▶ Bajo nivel.
- ▶ No estructurados.
- ▶ Balanceado incorrecto de operaciones.
- ▶ Semántica poco ligada al contexto.
- ▶ Mantenimiento complejo de códigos que hacen uso exhaustivo de semáforos.

## Definición

Una región crítica es una sección crítica cuya ejecución bajo exclusión mutua está garantizada (Hoare y Brinch-Hansen, 1972).

- ▶ Más alto nivel que los semáforos.
- ▶ Agrupa acceso a recursos comunes (variables) en regiones (recursos).
- ▶ Variables comunes **etiquetadas** como compartidas o recurso.
- ▶ Un proceso no puede entrar en una región donde ya hay otro proceso. Debe esperar.
- ▶ Son una notación de carácter sintáctico.
- ▶ Permiten detectar accesos indebidos en **tiempo de compilación**.



```
var V: shared T;  
.  
.  
.  
region V do  
S; /* acceso en e.m. a V */
```

- ▶ Las entidades concurrentes sólo pueden acceder a variables compartidas dentro de una región crítica.
- ▶ Una entidad concurrente que desee entrar en una región lo logrará en tiempo finito.
- ▶ En tiempo  $t$ , sólo puede haber un proceso dentro de una región crítica.
- ▶ Una entidad concurrente pasa un tiempo finito dentro de una región crítica y posteriormente la abandona.
- ▶ Un proceso que no puede entrar a una región es puesto en espera.
- ▶ En general, la gestión de la cola de procesos en espera es equitativa.

# Regiones Críticas: Inconvenientes

- ▶ Su anidamiento puede producir interbloqueos.

```
P: region x do  
    region y do S1;
```

```
Q: region y do  
    region x do S2;
```

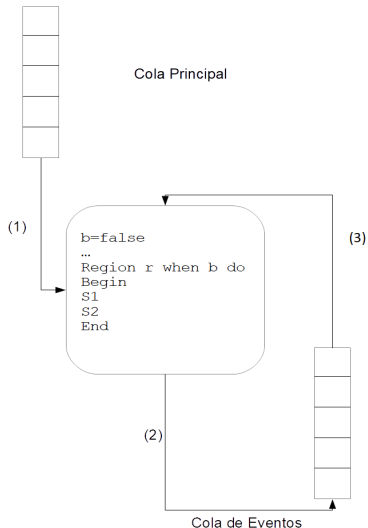
- ▶ No dan soporte para resolver sincronización.
- ▶ Mejora: regiones críticas condicionales.

# Regiones Críticas Condicionales: Sintaxis

- ▶ Es una construcción sintáctica bastante sencilla...
- ▶ ...aunque su semántica no lo es tanto

```
region identificador when condicion do  
begin  
s1;  
...  
sn;  
end;
```

# Regiones Críticas Condicionales: Arquitectura



# Regiones Críticas Condicionales: Semántica

- ▶ Un proceso que desea ejecutar la región crítica debe conseguir el acceso en exclusión mutua a la misma. Si ya está ocupada, espera sobre la cola principal
- ▶ El proceso que logra la exclusión mutua a la sección crítica entra en ella y evalúa la condición (Etapa (1) del diagrama). Si la condición es cierta, se ejecuta la lista de instrucciones ( $s_1, s_2, \dots, s_n$ ) que forman la región en exclusión mutua. Si la condición es falsa, el proceso es enviado (Etapa (2) del diagrama) a la cola de eventos, en espera de que la condición sea cierta. La exclusión mutua sobre la región se libera.
- ▶ Cuando un proceso termina de ejecutar la sección crítica, la región comprueba si hay procesos en la cola de eventos para los cuáles la re-evaluación de la condición es cierta. Si no hay ninguno (esto implica reevaluar las condiciones para todos los procesos de dicha cola), la región da paso al proceso que está en el frente de la cola principal. Si hay alguno, se le da acceso a la región.

- ▶ No existen como tales.
- ▶ Pueden simularse a partir de otras primitivas, como los monitores.

- ▶ Existen mediante bloques de código `synchronized`.
- ▶ Requieren de un objeto para proveer el bloqueo (Tema 4).
- ▶ El modelo RCC se puede simular mediante un monitor construido con el API de concurrencia estándar (Tema 4)



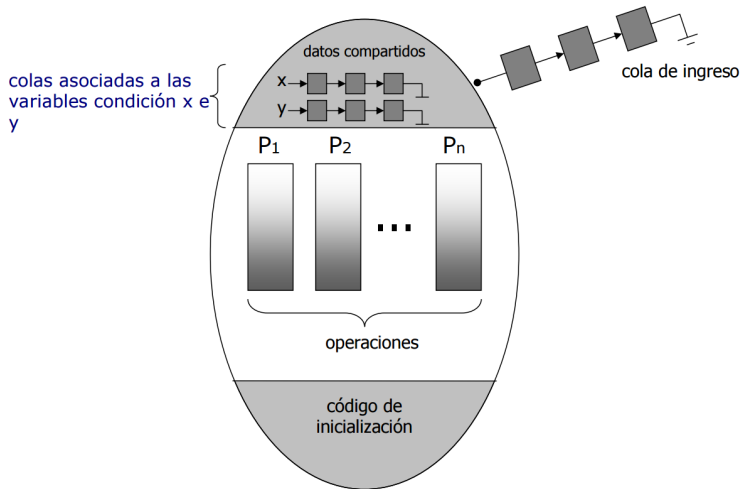
- ▶ Descargue `regCritica.java`
- ▶ Verifique la preservación de la exclusión mutua.

## Definición

Un monitor es una **construcción sintáctica** de un lenguaje de programación concurrente que encapsula un recurso crítico a gestionar en exclusión mutua, junto con los procedimientos que lo gestionan. Por tanto, se da una **centralización** de recursos y una **estructuración** de los datos [Hoare, 1974].

- ▶ **Encapsulación** de los datos.
- ▶ Alta **estructuración**.
- ▶ Exclusión mutua de procesos internos por definición.
- ▶ Sincronización mediante el uso del concepto de **señal**.
- ▶ Compacidad y eficacia.
- ▶ Transparencia al usuario.
- ▶ Tienen las mismas capacidades que los semáforos.
- ▶ Son **más expresivos**.

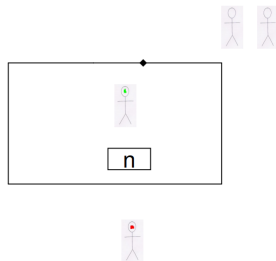
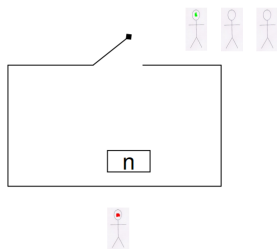
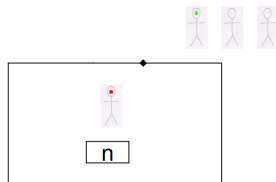
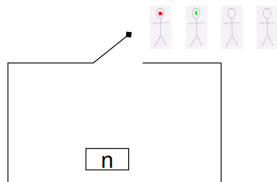
# Monitores: Idealización Gráfica



```
type nombre-monitor=monitor  
  declaraciones de variables  
  
  procedure entry P1(...)  
  begin ... end;  
  ...  
  procedure entry PN(...)  
  begin ... end;  
  
begin  
  codigo de inicializacion  
end;
```

- ▶ El acceso de las entidades concurrentes al monitor es en exclusión mutua.
- ▶ Las variables de condición (señales) proveen sincronización.
- ▶ Una entidad concurrente puede quedar en espera sobre una variable de condición. En ese momento, la exclusión mutua se libera y otras entidades pueden acceder al monitor.
- ▶ Disciplinas de señalización más utilizadas:
  - ▶ Señalar y salir: la entidad que señala abandona el monitor.
  - ▶ Señalar y seguir: la entidad que señala sigue dentro del monitor.

# Monitores: Control de la exclusión mutua



# Monitores: Variables de Condición (señales)

- ▶ Permiten sincronizar a las entidades concurrentes que acceden al monitor.
- ▶ Declaración: `nombre_variable: condition;`
- ▶ Operaciones soportadas:
  - ▶ `wait(variable_condición)`: la entidad concurrente que dentro del monitor hace la llamada es suspendida en una cola FIFO asociada a la variable en espera de que se cumpla la condición. Exclusión mutua liberada.
  - ▶ `send(variable_condición)`: proceso situado al frente de la cola asociada a la variable despertado.
  - ▶ `non_empty(variable_condición)`: devuelve verdadero si la cola asociada a la variable no está vacía.

NOTA: En ocasiones, la sintaxis podrá ser `c.wait`, `c.send` para una variable de condición `c`.

# Monitores: Disciplinas de Señalización I

Tipo	Carácter	Clase de Señal
SA	Señales automáticas	Señales implícitas, incluidas por el compilador. No hay que programar send.
SC	Señalar y continuar	Señal explícita no desplazante. El proceso señalador no sale.
SX	Señalar y salir	Señal explícita desplazante. El proceso señalador sale. Send debe ser la última instrucción del monitor.
SW	Señalar y esperar	Señal explícita desplazante. El proceso señalador sale y es enviado a la cola de entrada del monitor.
SU	Señalar con urgencia	Señal explícita desplazante. El proceso señalador sale y es enviado a una cola de procesos urgentes, prioritaria sobre la de entrada.



# Monitores: Disciplinas de Señalización II

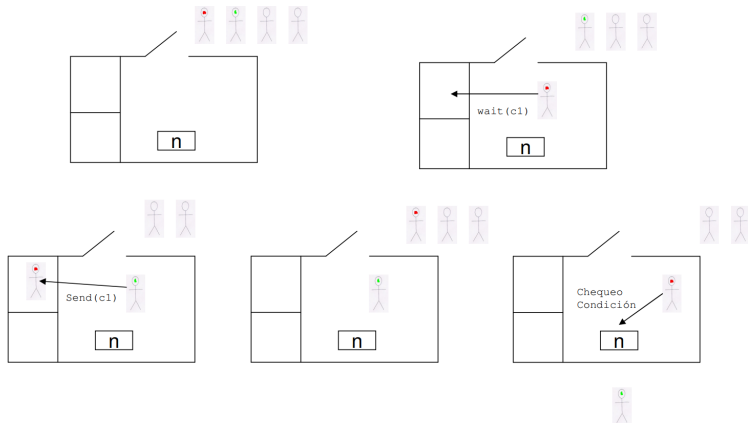


Figura: Señalar y Continuar

# Monitores: Disciplinas de Señalización III

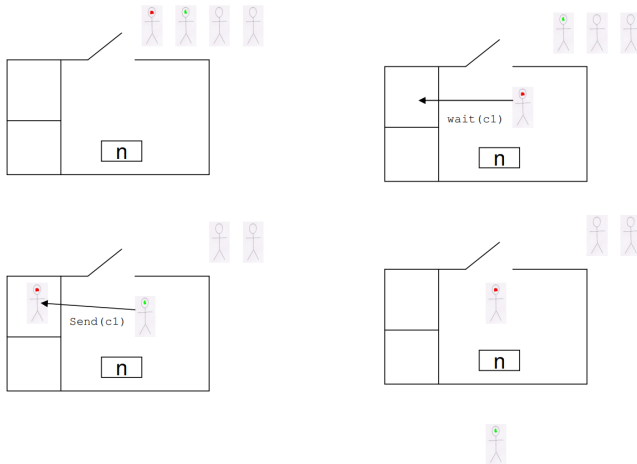


Figura: Señalar y Salir

# Monitores: Disciplinas de Señalización IV

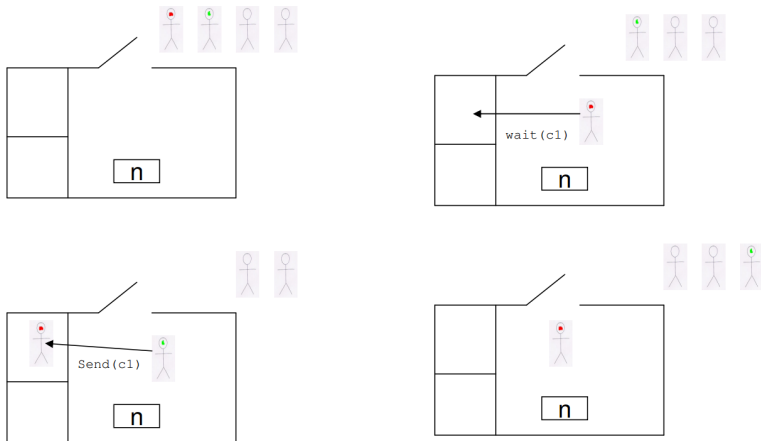


Figura: Señalar y Esperar

# Monitores: Disciplinas de Señalización V

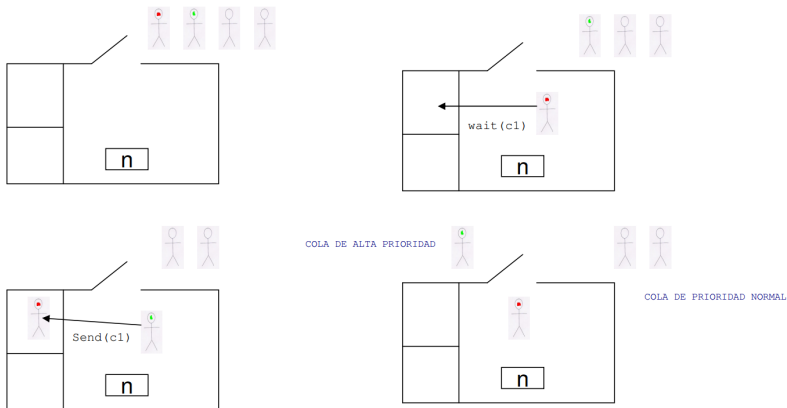


Figura: Señalar con Urgencia

# Monitor Productor-Consumidor

```
monitor prod_con is
  B: array(0..N-1) of Integer;
  In_Ptr, Out_Ptr: Integer:=0;
  Count: Integer:=0;
  Not_Full, Not_Empty: Condition;
```

```
Procedure Añadir(I:in Integer) is
begin
  if Count=N then wait(Not_Full);
  end if;

  B(In_Ptr):=I;
  In_Ptr:=(In_Ptr+1) mod N;
  Send(Not_Empty);
end;
```

```
Procedure Coger(I:out Integer) is
begin
  if Count=0 then wait(Not_Empty);
  end if;

  I:=B(Out_Ptr);
  Out_Ptr:=(Out_Ptr+1) mod N;
  Send(Not_Full);
end;

end prod_con;
```

- ▶ C++11 proporciona (¡por fin!) clases que dan soporte a:
  - ▶ Cerrojos.
  - ▶ Variables de condición.
- ▶ Es posible implantar monitores mediante su uso combinado.

- ▶ Todo objetos es un monitor potencial.
- ▶ Clase Object: métodos wait, notify, notifyAll.
- ▶ Clases con métodos synchronized.
- ▶ Sólo soporta variables de condición a partir de Java 1.5.
- ▶ En otro caso, hay que *simular* la sincronización con los métodos de la clase Object.
- ▶ Equivalen a una única variable de condición.

- ▶ Descargue `incDecMonitor.java`.
- ▶ Escriba una condición de carrera sobre un objeto de la clase anterior.
- ▶ Verifique la preservación de la exclusión mutua.



# En el Próximo Tema...

- ▶ Control de la Concurrency en Java.
- ▶ API estándar.
- ▶ Códigos y Métodos `synchronized`.
- ▶ Protocolos de exclusión mutua.
- ▶ Sincronización.
- ▶ Diseño de Monitores en Java.



Ben-Ari, M.

*Principles of Concurrent and Distributed Programming*

Add. Wesley, 2006



Raynal, M.

*Algorithms for Mutual Exclusion*

MIT Press, 1986



Palma, J.

*Programación Concurrente*