Programación Concurrente 2017

Clases 5 y 6

Facultad de Informática
UNLP

Resumen de la clase anterior

Variables compartidas →
-Sincronización barrier
- **Contador compartido**
- **Flags y coordinador**
- **Arbol**
- **Barreras simétricas → Butterfly**

Semáforos →
-Definición
-Sintaxis y semántica. Operaciones.
-Resolución de casos:
- **Sección crítica**
- **Barreras**
- **Productores y consumidores**
- **Problema de los filósofos**
- **Lectores y escritores**

Lectores y Escritores como problema de Exclusión mutua

```
int nr = 0;   sem rw = 1;
process Lector [i = 1 to M] {
    while(true) { …
        ⟨ nr = nr + 1;
          if (nr == 1) P(rw); ⟩
        lee la BD;
        ⟨ nr = nr - 1;
          if (nr == 0) V(rw); ⟩  } }
process Escritor [j = 1 to N] {
    while(true) { …
        P(rw);
        escribe la BD;
        V(rw);    } }
```

```
int nr = 0;          # número de lectores activos
sem rw = 1;          # bloquea acceso a la BD
sem mutexR= 1; # bloquea acceso de los lectores a nr

process Lector [i = 1 to M] {
    while(true) { …
        P(mutexR);
          nr = nr + 1;
          if (nr == 1) P(rw);   # si es el primero, toma el bloqueo
        V(mutexR);
        lee la BD;
        P(mutexR);
          nr = nr - 1;
          if (nr == 0) V(rw);   # si es el último, libera el bloqueo
        V(mutexR);   } }

process Escritor [j = 1 to N] {
    while(true) { …
        P(rw);    # toma bloqueo de acceso exclusivo
        escribe la BD;
        V(rw);    # libera el bloqueo              } }
```

Lectores y Escritores usando Sincronización por Condición

Solución anterior ⇒ preferencia a los lectores ⇒ no es *fair*

Además, no es sencillo modificarla …

Otro enfoque ⇒ introduce la técnica *passing the baton*: emplea SBS para brindar exclusión y despertar procesos demorados

Puede usarse para implementar *await* arbitrarios, controlando de forma precisa el orden en que los procesos son despertados

En este caso, pueden contarse los procesos de cada clase intentando acceder a la BD, y luego restringir el valor de los contadores

Lectores y Escritores usando Sincronización por Condición

nr y nw enteros no negativos que registran el número de lectores y escritores accediendo a la BD

Estados malos a evitar: nr y nw positivos, o nw mayor que 1:
BAD: (nr > 0 AND nw > 0) OR nw > 1

⇒ estados buenos caracterizados por NOT BAD
RW: (nr == 0 OR nw == 0) AND nw <= 1

```
int nr = 0, nw = 0;
process Lector [i = 1 to M] {
    while(true) { …
        ⟨ await (nw == 0) nr = nr + 1; ⟩
        lee la BD;
        ⟨ nr = nr – 1; ⟩      } }
process Escritor [j = 1 to N] {
    while(true) { …
        ⟨ await (nr == 0 and nw == 0) nw = nw + 1; ⟩
        escribe la BD;
        ⟨ nw = nw – 1; ⟩              } }
```

La técnica *Passing the Baton*

En algunos casos, await puede ser implementada directamente usando semáforos u otras operaciones primitivas. *Pero no siempre…*

En el caso de las guardas de los await en la solución anterior, se superponen en que el protocolo de E/ para escritores necesita que tanto **nw** como **nr** sean 0, mientras para lectores sólo que **nw** sea 0.

Ningún semáforo podría discriminar entre estas condiciones $\Rightarrow$ ***Passing the baton***: *técnica general para implementar sentencias* ***await***

La técnica *Passing the Baton*

La sincronización se expresa con sentencias atómicas de la forma:

$F_1 : \langle S_i \rangle$ **o** **$F_2 : \langle$ await (B_j) $S_j \rangle$**

Puede hacerse con semáforos binarios divididos (SBS)

e sem. binario inicialmente 1 (controla la E/ a sentencias atómicas).

Utilizamos un semáforo **b_j** y un contador **d_j** cada uno con guarda diferente **B_j**; todos inicialmente 0.

b_j se usa para demorar procesos esperando que **B_j** sea true
d_j es un contador del número de procesos demorados sobre **b_j**

e y los b_j se usa para formar un SBS: a lo sumo uno a la vez es 1, y c/ camino de ejecución empieza con un P y termina con un único V

La técnica *Passing the Baton*

F_1: **P(e);**
S_i;
SIGNAL

F_2: **P(e)**
if (not B_j) {$d_j = d_j + 1$; V(e); P(b_j); }
S_j;
SIGNAL

SIGNAL: if (B_1 and $d_1 > 0$) {$d_1 = d_1 - 1$; V(b_1)}
- **...**
- **(B_n and $d_n > 0$) {$d_n = d_n - 1$; V(b_n)}**
- **else V(e);**

fi

La técnica *Passing the Baton*

***Passing the baton*:** cuando un proceso está dentro de una SC mantiene el *baton* (testimonio, token) que significa **permiso para ejecutar**

Cuando el proceso llega a un **SIGNAL**, pasa el *baton* (control) a otro proceso.

Si ningún proceso está esperando una condición que sea ***true,*** el baton se pasa ***al próximo proceso que trata de entrar a su SC*** por primera vez (es decir, uno que ejecuta P(e)).

Lectores y escritores:
- **e** semáforo p/ controlar acceso a las vbles compartidas, **r** semáforo asociado a la guarda en procesos lectores, y **w** asociado a la guarda en escritores
- **dr** y **dw** contadores de lectores y escritores esperando

Passing the Baton en Lectores y Escritores

```
int nr = 0, nw = 0;
sem e = 1, r = 0, w = 0:        # siempre  0 <= (e+r+w) <= 1
int dr = 0, dw = 0;
process Lector [i = 1 to M] {
    while(true) {
        # ⟨ await (nw == 0) nr = nr + 1; ⟩
        P(e);
        if (nw > 0) {dr = dr+1; V(e); P(r); }
        nr = nr + 1;
        SIGNAL1 ;
        lee la BD;
        # ⟨ nr = nr – 1; ⟩
        P(e);
        nr = nr – 1;
        SIGNAL2 ;
   }
}
```

```
F1:   P(e);
      Si;
      SIGNAL
F2:   P(e)
      if (not Bj) {dj = dj + 1; V(e); P(bj); }
      Sj;
      SIGNAL
```

Passing the Baton en Lectores y Escritores

```
process Escritor [j = 1 to N] {
    while(true) {
        # ⟨ await (nr == 0 and nw == 0) nw = nw + 1; ⟩
        P(e);
        if (nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
        nw = nw + 1;
        SIGNAL3 ;
        escribe la BD;
        # ⟨ nw = nw – 1; ⟩
        P(e);
        nw = nw – 1;
        SIGNAL4 ;
   }
}
```

```
F1:   P(e);
      Si;
      SIGNAL
F2:   P(e)
      if (not Bj) {dj = dj + 1; V(e); P(bj); }
      Sj;
      SIGNAL
```

El rol de los SIGNAL ¡ es el de señalizar *exactamente* a uno de los semáforos ⇒ los procesos se van pasando el *baton*

Passing the Baton en Lectores y Escritores

```
F1:   P(e);
      Si;
      SIGNAL
F2:   P(e)
      if (not Bj) {dj = dj + 1; V(e); P(bj); }
      Sj;
      SIGNAL
```

$SIGNAL_i$ es una abreviación de:

```
if (nw == 0 and dr > 0) {
   dr = dr - 1; V(r);          # despierta un lector, o
   }
elseif (nr == 0 and nw == 0 and dw > 0) {
   dw = dw - 1; V(w);          # despierta un escritor, o
   }
else
   V(e);                       # libera el bloqueo de E/
```

e, w y r forman un SBS

Antes de $SIGNAL_1$, nr > 0 y nw = 0 son true ⇒ puede simplificarse a:

```
if (dr > 0) {dr := dr -1; V(r); }
   else V(e);
```

Algo similar sucede con los otros SIGNAL ⇒

Passing the Baton en Lectores y Escritores

```
int nr = 0, nw = 0;
sem e = 1, r = 0, w = 0:     # siempre  0 <= (e+r+w) <= 1
int dr = 0, dw = 0;
process Lector [i = 1 to M] {
    while(true) {
            # ⟨ await (nw == 0) nr = nr + 1; ⟩
            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);
            lee la BD;
            # ⟨ nr = nr – 1; ⟩
            P(e);
            nr = nr – 1;
            if (nr == 0 and dw > 0) {dw = dw – 1; V(w); }
              else V(e);
   }
}
```

Passing the Baton en Lectores y Escritores

```
process Escritor [j = 1 to N] {
    while(true) {
            # ⟨ await (nr == 0 and nw == 0) nw = nw + 1; ⟩
            P(e);
            if (nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
            nw = nw + 1;
            V(e);
            escribe la BD;
            # ⟨ nw = nw – 1; ⟩
            P(e);
            nw = nw – 1;
            if (dr > 0) {dr = dr - 1; V(r); }
              elseif (dw > 0) {dw = dw - 1; V(w); }
              else V(e);
             }
}
```

Da preferencia a los lectores ⇒ cómo puede modificarse??

Alocación de Recursos y Scheduling

Problema ⇒ decidir cuándo se le puede dar a un proceso determinado acceso a un recurso

Recurso ⇒ cualquier objeto, elemento, componente, dato, SC, por la que un proceso puede ser demorado esperando adquirirlo

Hasta ahora aplicamos la política más simple: si algún proceso está esperando y el recurso está disponible, se lo asigna

La política de alocación más compleja fue en R/W, aunque daba preferencia a *clases* de procesos, no a procesos individuales

Cómo se pueden implementar políticas de alocación de recursos generales y cómo controlar explícitamente cuál proceso toma un recurso si hay más de uno esperando??

Alocación de Recursos y Scheduling. Definición del problema y patrón de solución general

Procesos que compiten por el uso de unidades de un recurso compartido (cada unidad está *libre* o *en uso*)

request (parámetros):
⟨await (request puede ser satisfecho) tomar unidades;⟩
release (parámetros):
⟨retornar unidades;⟩

Puede usarse Passing the Baton
request tiene la forma de F_2
request (parámetros): P(e);
if (request no puede ser satisfecho) DELAY;
tomar las unidades;
SIGNAL;
release tiene la forma de F_1
release (parámetros): P(e);
retornar unidades;
SIGNAL;

Alocación *Shortest-Job-Next*

Varios procesos que compiten por el uso de un recurso compartido *de una sola unidad*

request (tiempo,id)
Si el recurso está libre, es alocado inmediatamente al proceso *id*; sino, el proceso *id* se demora

release ( )
Cuando el recurso es liberado, es alocado al proceso demorado (si lo hay) con el mínimo valor de *tiempo*.
Si dos o más procesos tienen el mismo valor de *tiempo*, el recurso es alocado al que esperó más

Ejemplos:
- alocación de procesador (*tiempo* es el tiempo de ejecución)
- spooling de una impresora (*tiempo* tiempo de impresión)

Alocación *Shortest-Job-Next*

SJN minimiza el tiempo promedio de ejecución, aunque *es unfair* (por qué ?)
Puede mejorarse con la técnica de *aging* (dando preferencia a un proceso que esperó mucho tiempo)

Para el caso general de alocación de recursos (*NO SJN*):

bool libre = true;
request (tiempo,id): ⟨await (libre) libre = false;⟩
release (): ⟨libre = true;⟩

En SJN, un proceso que invoca a *request* debe demorarse hasta que el recurso esté libre *y* su pedido sea el próximo en ser atendido de acuerdo a la política.

El parámetro *tiempo* entra en juego sólo si un pedido debe ser demorado (esto es, si *libre* es falso).

Alocación *Shortest-Job-Next*

```
request (tiempo, id):
        P(e);
        if (not libre) DELAY;
        libre = false;
        SIGNAL;

release ( ):
        P(e);
        libre = true;
        SIGNAL;
```

Qué hay que hacer en DELAY y SIGNAL ????

Alocación *Shortest-Job-Next*

En DELAY un proceso:

- inserta sus parámetros en un conjunto, cola o lista de espera (*pares*)
- libera la SC ejecutando V(e)
- se demora en un semáforo hasta que request puede ser satisfecho.

Cuando el recurso es liberado, si *pares* no está vacío, el recurso es asignado a un proceso de acuerdo a SJN

Cada proceso tiene una condición de demora distinta, dependiendo de su posición en *pares*.

***b*[1:n]** arreglo de semáforos, donde cada entry es inicialmente 0
$\Rightarrow$ el proceso *id* se demora sobre el semáforo b[id]

Alocación *Shortest-Job-Next*

```
bool libre = true;
sem e = 1, b[n] = ([n] 0);      # para entry y demora
typedef Pares = set of (int, int);
Pares =  ∅ ;
# SJN: Pares es un conjunto ordenado ∧ libre ⇒ ( pares== ∅ )

request(tiempo,id): P(e);
                    if (! free) {
                        insertar (tiempo,id) en Pares;
                        V(e);       # libera el bloqueo
                        P(b[id]);  # espera en SU semáforo a ser despertado }
                     libre = false;    # Se está tomando el recurso
                     V(e);        # libera el bloqueo p/ encolar otros procesos

release( ):         P(e);
                    libre = true;
                    if (Pares ≠ ∅ )  {
                        remover el primer par (tiempo,id) de Pares;
                        V(b[id]);      # pasa el baton al proceso id      }
                    else V(e);
```

Alocación *Shortest-Job-Next*

Los semáforos b[id] son ejemplos de *semáforos privados*.

A diferencia de los vistos anteriormente, se asocian con un único proceso

s es un semáforo privado si exactamente un proceso ejecuta operaciones P sobre s

Resultan útiles para señalar procesos individuales.

TAREA: qué modificaciones deberían realizarse para generalizar la solución a recursos de múltiple unidad??

Librería para uso de semáforos - Conceptos de Pthreads

Thread $\rightarrow$ proceso "liviano" que tiene su propio contador de programa y su pila de ejecución, pero no controla el "contexto pesado" (por ejemplo, las tablas de página).

Algunos sistemas operativos y lenguajes proveen mecanismos para permitir la programación de aplicaciones "multithreading".

En principio estos mecanismos fueron heterogéneos y poco portables $\Rightarrow$ a mediados de los 90 la organización POSIX auspició el desarrolló de una biblioteca en C para multithreading **(*Pthreads*)**

La biblioteca está actualmente en varias versiones de Unix, y con ella se pueden crear threads, asignarles atributos, darlos por terminados, identificarlos, etc.

Conceptos de Pthreads

```
# include <pthread.h>
```

Declaración de variables para un descriptor de atributos de thread y uno o más descriptores de thread:

```
pthread_attr_t attr;        /* atributos del thread */
pthread_t pid;              /* descriptor del thread */
```

Los atributos incluyen tamaño del stack del thread, prioridad de scheduling, y alcance del scheduling (local o global)

Si es global, el thread compite con todos los otros por el uso de procesador y no sólo con los creados por el mismo thread padre

Inicialización de atributos:

```
pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
```

Conceptos de Pthreads

Creación de thread:

```
pthread_create(&tid, &attr, start_func, arg);
```

donde

- &tid es la dirección de un descriptor que se llena si la creación tiene éxito
- &attr es la dirección de un descriptor inicializado previamente
- el thread comienza la ejecución llamando a start_func con un argumento arg

Un thread termina su propia ejecución llamando a

```
pthread_exit(value);
```

Un thread padre puede esperar a que termine un hijo con

```
pthread_join(tid, value_ptr);
```

donde tid es un descriptor y value_ptr es la dirección de una posición para el valor de retorno (que se llena cuando el hijo llama a exit)

Semáforos con Pthreads

Los threads pueden sincronizar por semáforos (librería semaphore.h)

Declaración y operaciones:

- ***sem_t semaforo*** (se declaran globales a los threads)
- ***sem_init (&semaforo, alcance, inicial)***

Inicializa el semáforo ***semaforo***. ***Inicial*** es el valor con que comienza. ***Alcance*** indica si es compartido por threads de un mismo proceso (0) o por los de todos los procesos (1)

- ***sem_wait(&semaforo)*** equivale a P
- ***sem_post(&semaforo)*** equivale a V

Existen funciones de wait condicional, para obtener el valor de un semáforo y para destruirlo.

Productores/Consumidores con Pthreads

Las funciones de Productor y Consumidor serán ejecutadas por threads independientes.

Acceden a un buffer compartido (*datos*).

El productor deposita una secuencia de enteros de 1 a *numItems* en el buffer.

El consumidor busca estos valores y los suma.

Los semáforos *vacio* y *full* garantizan el acceso alternativo de productor y consumidor sobre el buffer.

Productores/Consumidores con Pthreads

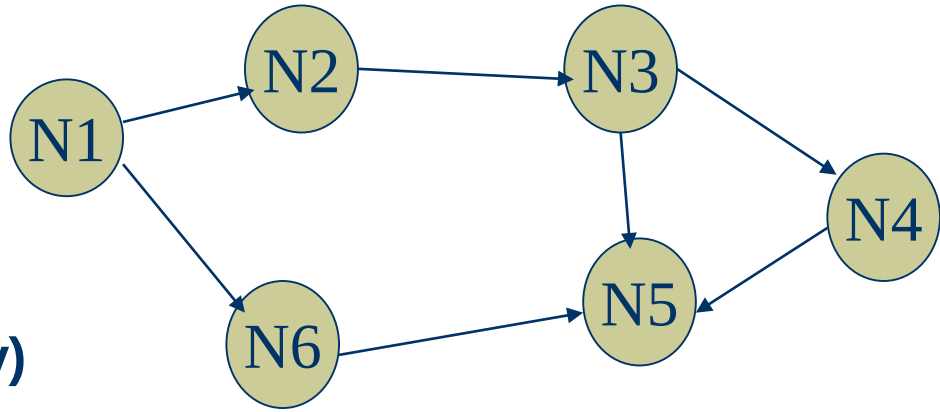
```
#include <pthread.h>
#include <semaphore.h>
#define SHARED 1
#include <stdio.h>

void *Productor(void *); /* los dos threads */
void *Consumidor(void *);
sem_t vacio, lleno; /* semáforos globales */
int datos; /* buffer compartido */
int numItems;

int main(int argc, char * argv[ ]) {
    ....
    sem_init(&vacio, SHARED, 1); /* sem vacio = 1 */
    sem_init(&lleno, SHARED, 0);  /* sem lleno = 0 */
    numItems = atoi(argv[1]);
    ....
   pthread_create(&pid, &attr, Productor, NULL);
   pthread_create(&cid, &attr, Consumidor, NULL);
   pthread_join(pid, NULL);
   pthread_join(cid, NULL);        }
```

Productores/Consumidores con Pthreads

```
/* deposita 1, ..., numItems en el buffer */
void *Productor(void *arg) {
    int item;
    for (item = 1; produced <= numItems; item++) {
        sem_wait(&vacio);
        datos = item;
        sem_post(&lleno);
        pthreads_exit();        }
}

/* busca numItems items en el buffer y los suma */
void *Consumidor(void *arg) {
    int total = 0, item, aux;
    for (item = 1; item <= numItems; item++) {
        sem_wait(&flleno);
        aux = dato;
        sem_post(&vacio);
        total = total + datos; }
     printf("TOTAL: %d\n", total);
     pthreads_exit();
}
```

Casos problema de interés: Grafo de precedencia

Analizar el problema de modelizar N procesos que deben sincronizar, de acuerdo a lo especificado por un grafo de precedencia arbitrario (con semáforos)



Process N [i:1..6] {
esperar a predecesores (si hay)
ejecutar la tarea
señalizar a sucesores (si hay) }

Ej: en este caso N4 hará P(N3) y V(N5), N1 hará V(N2) y V(N6)

Casos problema de interés: Productores/Consumidores con broadcast

En el problema del buffer atómico, sea UN proceso productor y N procesos consumidores.
El Productor DEPOSITA y debe esperar que TODOS los consumidores consuman el mismo mensaje (broadcast).
Notar la diferencia entre una solución por memoria compartida y por mensajes.

Versión más compleja: buffer con K lugares, UN productor y N consumidores.
El productor puede depositar hasta K mensajes, los N consumidores deben leer cada mensaje para que el lugar se libere y el orden de lectura de cada consumidor debe ser FIFO.
Analizar el tema con semáforos.

Casos problema de interés: Variantes del problema de los filósofos

Si en lugar de administrar tenedores, cada filósofo tiene un estado (comiendo, pensando, con hambre) y debe consultar a sus dos filósofos laterales para saber si puede comer, tendremos una solución distribuida. Se podría usar la técnica de "passing the baton" para resolverlo?

Otra alternativa es tener una solución de filósofos centralizada, en la que un scheduler administra por ejemplo los tenedores y los asigna con posiciones fijas o variables (notar la diferencia).

En la solución que vimos de filósofos, para evitar deadlock utilizamos un código distinto para un filósofo (orden de toma de los tenedores). Otra alternativa es la de la elección al azar del primer tenedor a tratar de tomar... Cómo?

Casos problema de interés: El problema de los baños

Un baño único para varones o mujeres (excluyente) sin límite de usuarios.

Un baño único para varones o mujeres (excluyente) con un número máximo de ocupantes simultáneos (que puede ser diferente para varones y mujeres)

Dos baños utilizables simultáneamente por un número máximo de varones (K1) y de mujeres (K2), con una restricción adicional respecto que el total de usuarios debe ser menor que K3 (K3 < K1 + K2).

Casos problema de interés:
El problema de la molécula de agua

Existen procesos O (oxígeno) y H (hidrógeno) que en un determinado momento toman un estado "listo" y se buscan para combinarse formando una molécula de agua (HH0).

Puede pensarse en un esquema C/S, donde el servidor recibe los pedidos y cuando tiene 2 H *listos* y 1 O *listo* concreta la molécula de agua y libera los procesos H y O.

También puede pensarse como un esquema "passing the baton" que puede iniciarse por cualquiera de los dos H o por el O, pero tiene un orden determinado (analizar con cuidado).

Podría pensar la solución con un esquema de productores-consumidores?
Quiénes serían los productores y quienes los consumidores?

Casos problema de interés: El puente de una sola vía

Suponga un puente de una sola vía que es accedido por sus extremos (procesos *Norte* y procesos *Sur*).
Cómo especificaría la exclusión mutua sobre el puente, de modo que circulen los vehículos (procesos) en una sola dirección.

Es un caso típico donde es difícil asegurar fairness y no inanición. Por qué? Cuál podría ser un método para asegurar no inanición con un scheduler? Qué ideas propone para tener fairness entre Norte y Sur ?

Suponga que cruzar el puente requiere a lo sumo 3 minutos y Ud. quiere implementar una solución tipo "time sharing" entre los procesos Norte y Sur, habilitando alternativamente el puente 15 minutos en una y otra dirección. Cómo lo esquematizaría?

Casos problema de interés: Search – Insert – Delete sobre una lista con procesos concurrentes

Una generalización de la EM selectiva e/ clases de procesos visto con lectores-escritores es el problema de procesos que acceden a una lista enlazada para *Buscar* (search), *Insertar* al final o *Borrar* un elemento en cualquier posición.

Los procesos que BORRAN se excluyen entre sí y además excluyen a los procesos que buscan e insertan.
Sólo un proceso de borrado puede acceder a la lista.

Los procesos de inserción se excluyen entre sí, pero pueden coexistir con los de búsqueda. A su vez, los de búsqueda NO se excluyen entre sí

Casos problema de interés: Modificaciones a SJN

Suponga que se quiere cambiar el esquema de alocación SJN por uno LJN (*longest JOB next*). Analice el código y comente los cambios. Cuál de los dos esquemas le parece más fair? Cuál generará una cola mayor de procesos pendientes?

En el esquema SJN, suponga que lo quiere cambiar por un esquema FIFO. Analice el código y comente los cambios. Cuál de los dos esquemas le parece más eficiente? Más Fair ?

Casos problema de interés: Drinking Philosophers

Investigar el tema definido por Chandy y Misra en 1984.
Se trata de una generalización del problema de los dining philosophers.

Analizar las diferencias de los dos problemas, estudiar los mecanismos de sincronización y discutir las ventajas/dificultades de utilizar semáforos en la sincronización.

Monitores

Semáforos ⇒
-Variables compartidas globales a los procesos
-Sentencias de control de acceso a la SC dispersas en el código
-Al agregar procesos, se debe verificar acceso correcto a las VC
-Aunque EM y SxC son conceptos ≠, se programan de forma similar

Monitores **⇒ módulos de pgm con más estructura, y que pueden ser implementados tan eficientemente como los semáforos.**

Mecanismo de abstracción de datos:
• encapsulan las representaciones de objetos (recursos).
• brindan un conjunto de operaciones que son *los únicos* medios para manipular la representación

Contiene variables que almacenan el estado del recurso y procedimientos que implementan las operaciones sobre él

Monitores

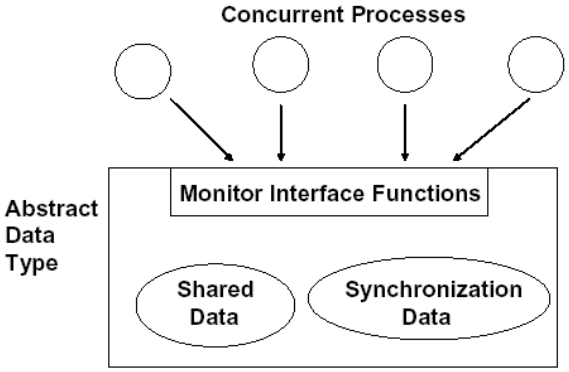
EM ⇒ **implícita asegurando que los procedures en el mismo monitor no ejecutan concurrentemente**

SxC ⇒ **explícita con variables condición**

Programa Concurrente ⇒ procesos activos y monitores pasivos

Dos procesos interactúan invocando procedures de un monitor.

Ventajas:
- un proceso que invoca un procedure puede ignorar cómo está implementado
- el programador del monitor puede ignorar cómo o dónde se usan los procedures



Monitores. Notación

Un monitor agrupa la representación y la implementación de un recurso compartido (clase). Lo que distingue a un monitor de un TAD es que puede ser *compartido* por procesos concurrentes.

- Tiene interfaz y cuerpo.
  * La interfaz especifica operaciones (métodos) que brinda el recurso.
  * El cuerpo tiene vbles que representan el *estado* del recurso y procedures que implementan las operaciones de la interfaz.

- Sólo los nombres de los procedures son visibles desde afuera. Sintácticamente, los llamados al monitor tienen la forma:

call NombreMonitor.op$_i$ (argumentos)

- Los procedures pueden acceder sólo variables permanentes, sus vbles locales, y parámetros que le sean pasados en la invocación

- El pgmdor de un monitor no puede conocer a priori el orden de llamado $\Rightarrow$ ***puede definirse un invariante del monitor que especifica los estados "razonables" de las vbles permanentes cuando ningún proceso las está accediendo***

Monitores. Notación

```
monitor NombreMonitor {
   declaraciones de variables permanentes;
   código de inicialización

   procedure op1 (par. formales1)
            { cuerpo de op1  }
   .......

   procedure opn (par. formalesn)
            { cuerpo de opn  }

}
```

Monitores. Sincronización

La *exclusión mutua* en monitores se provee *implícitamente.*

La *SxC* es programada explícitamente con *variables condición*

cond cv;

El valor asociado a cv es una cola de procesos demorados, *no visible directamente* al programador.

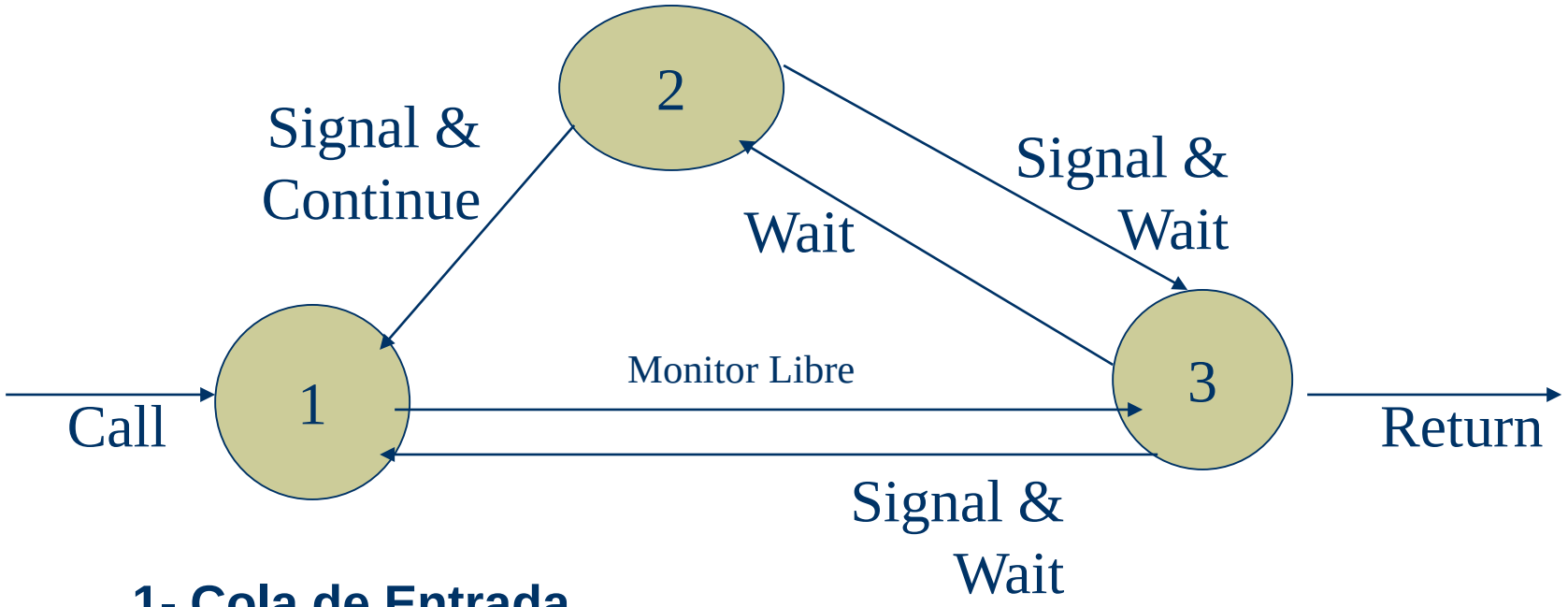
wait(cv)→ el proceso se demora al final de la cola de cv y deja el acceso exclusivo al monitor.

signal(cv)→ despierta al proceso que está al frente de la cola y lo saca de ella (cola vacía = skip). *Ese proceso puede ejecutar cuando readquiera el acceso exclusivo al monitor (depende de la disciplina de señalización)*

signal_all(cv) → despierta todos los procesos demorados en cv. El tiempo en que cada uno reinicie efectivamente la ejecución dependerá de las condiciones de exclusión mutua.

Monitores. Sincronización

Signal and continue vs. Signal and Wait



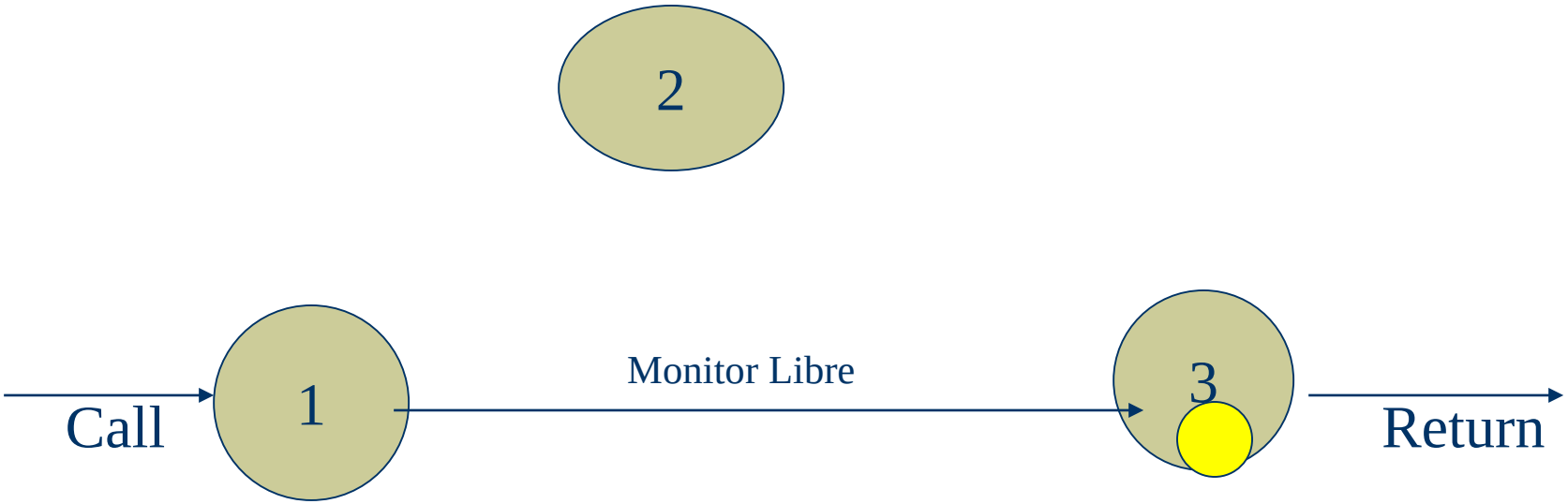
1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización

Llamado - Monitor libre



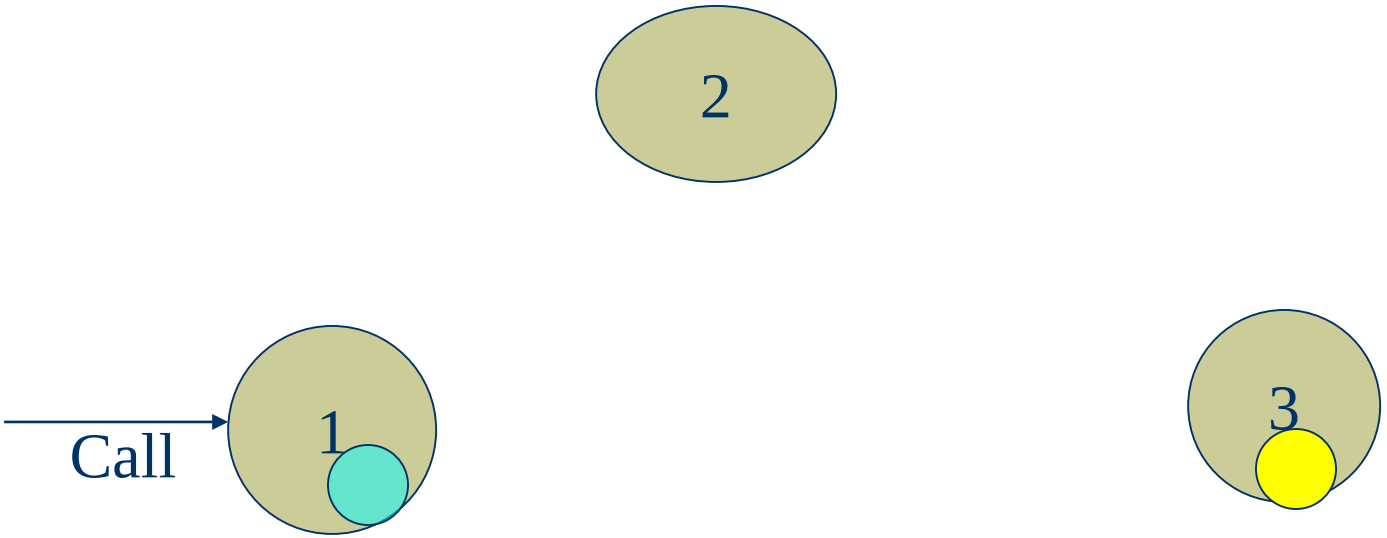
1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización

Llamado – monitor ocupado

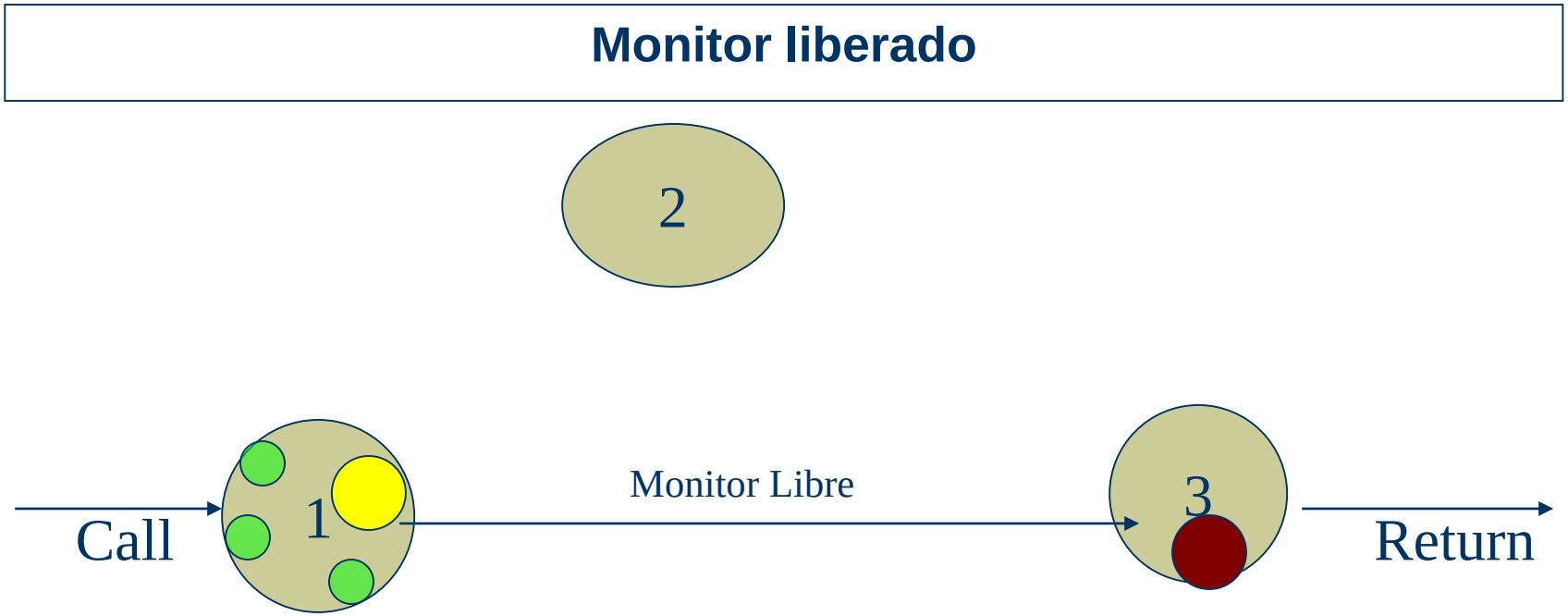


1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización



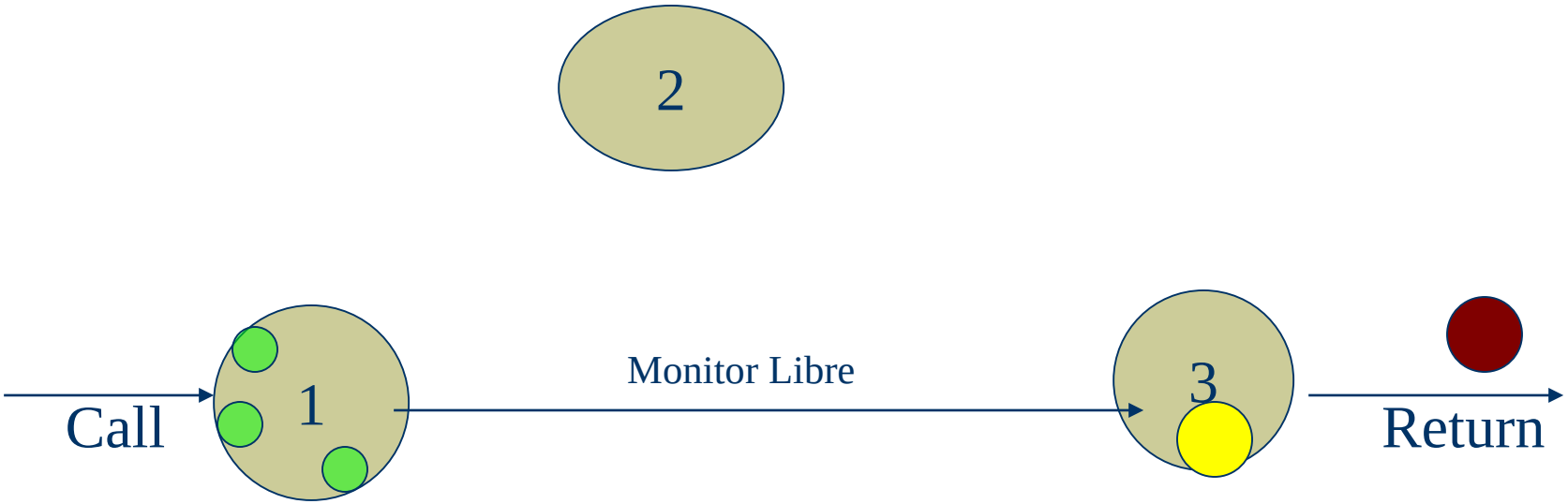
1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización

Monitor liberado

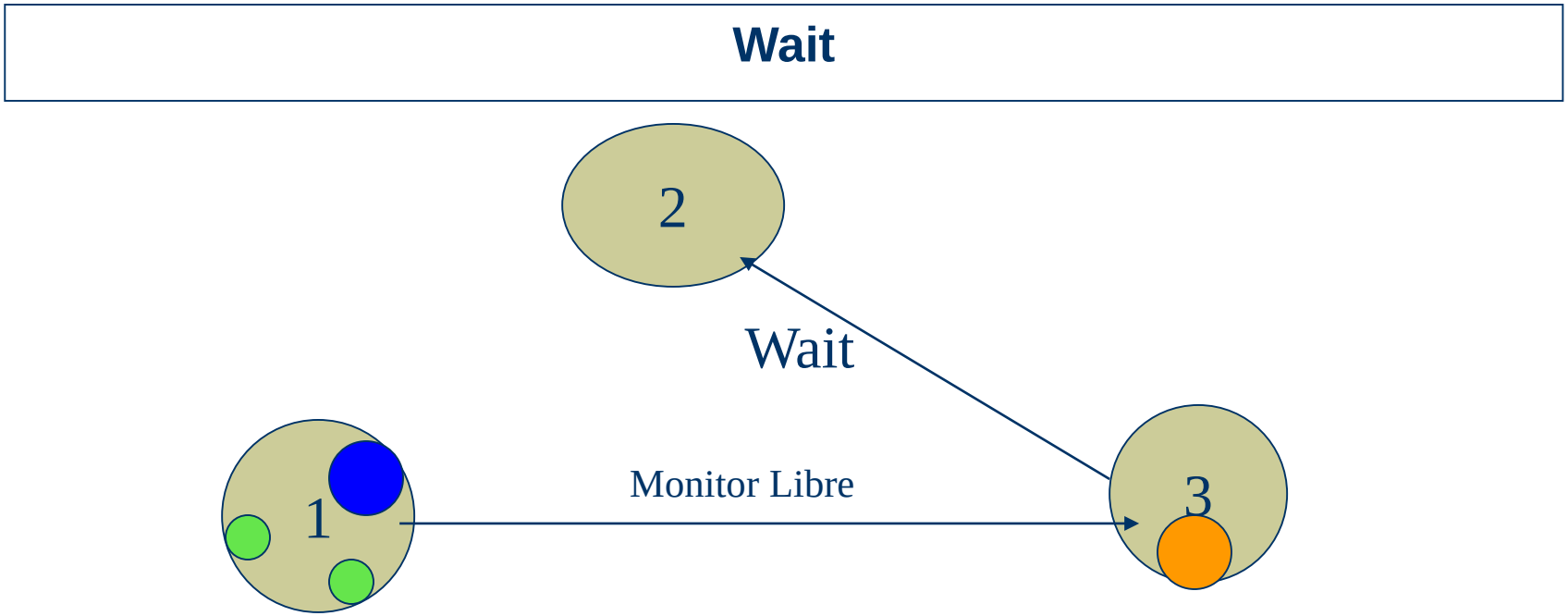


1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización

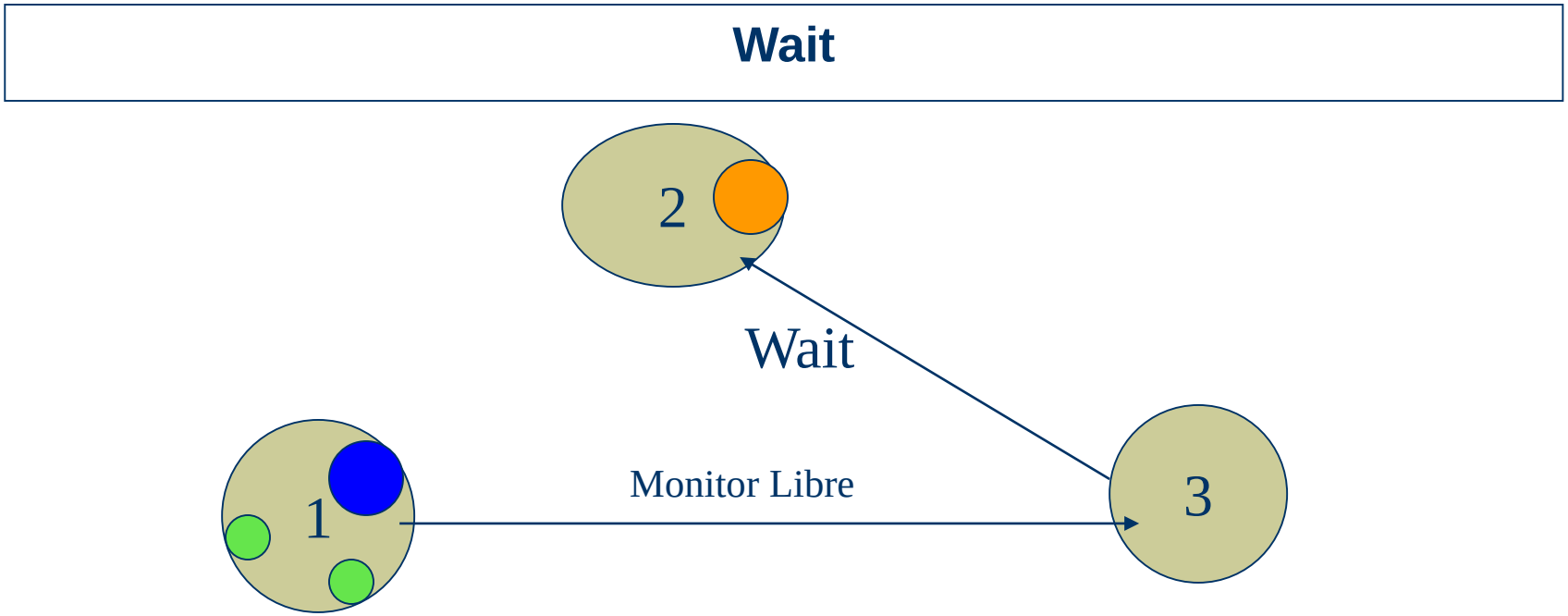


1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización

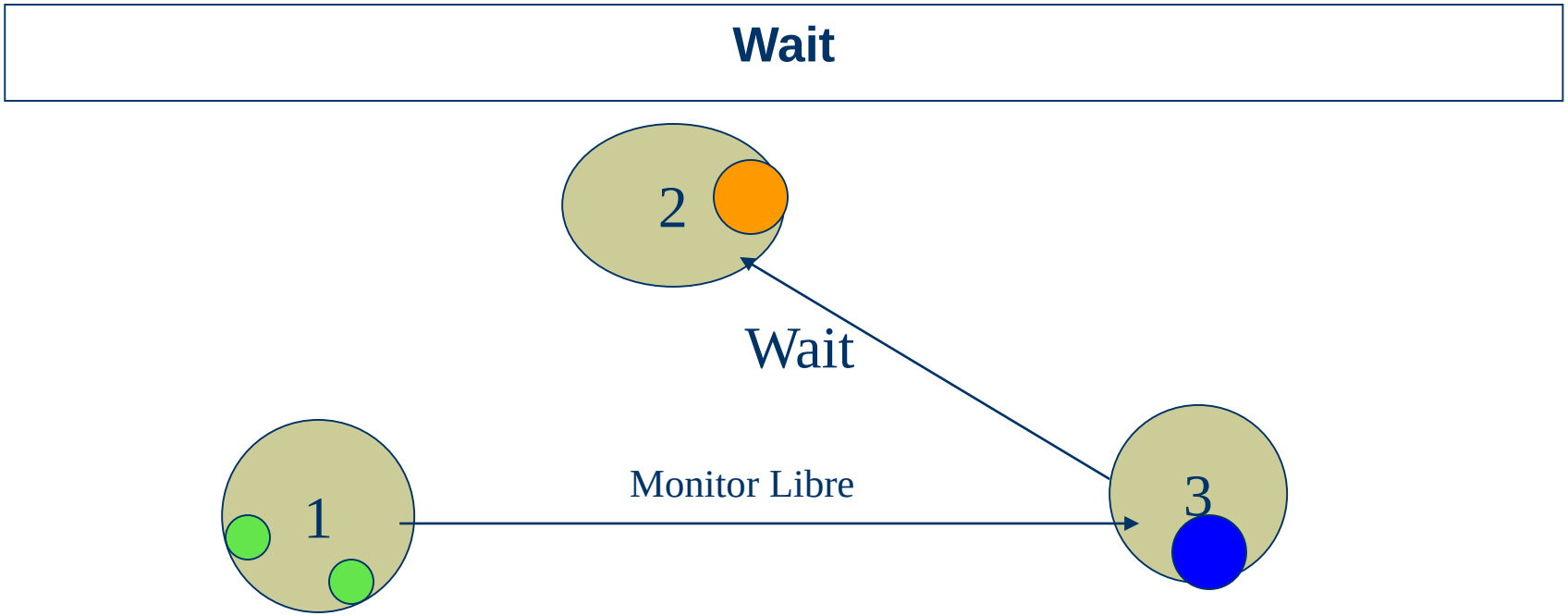


1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización



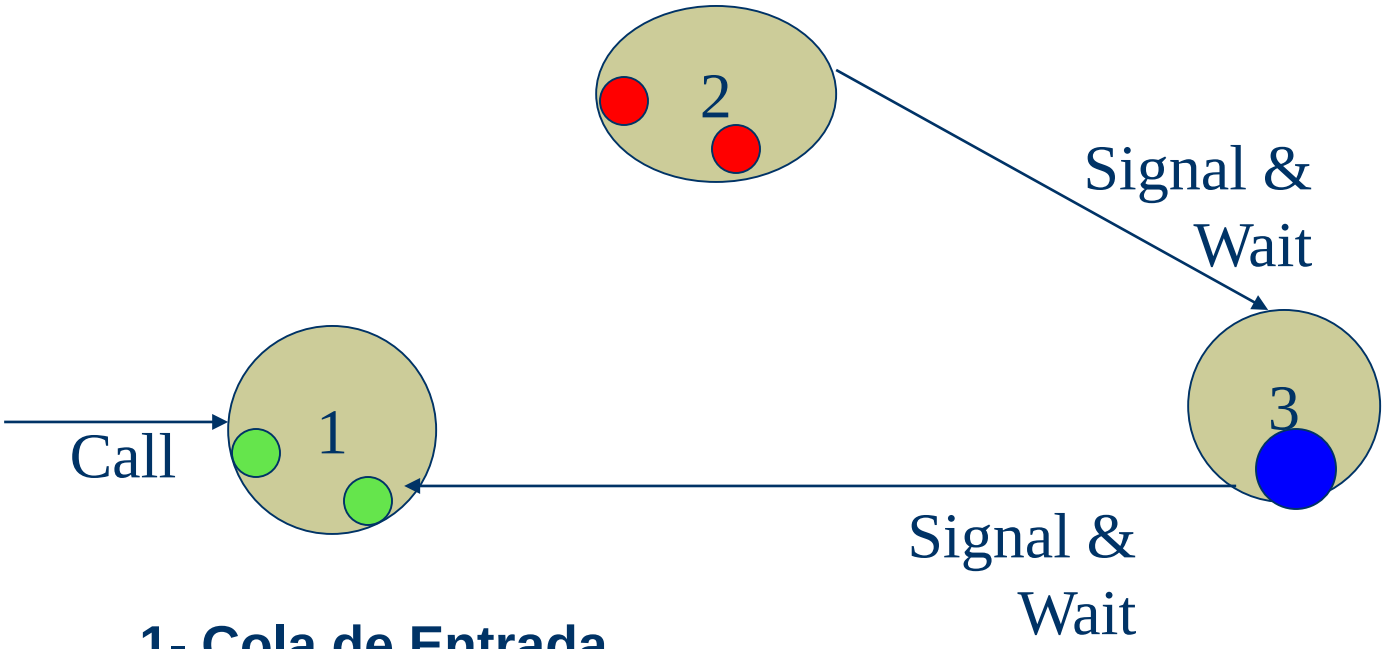
1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización

Disciplina Signal and Wait



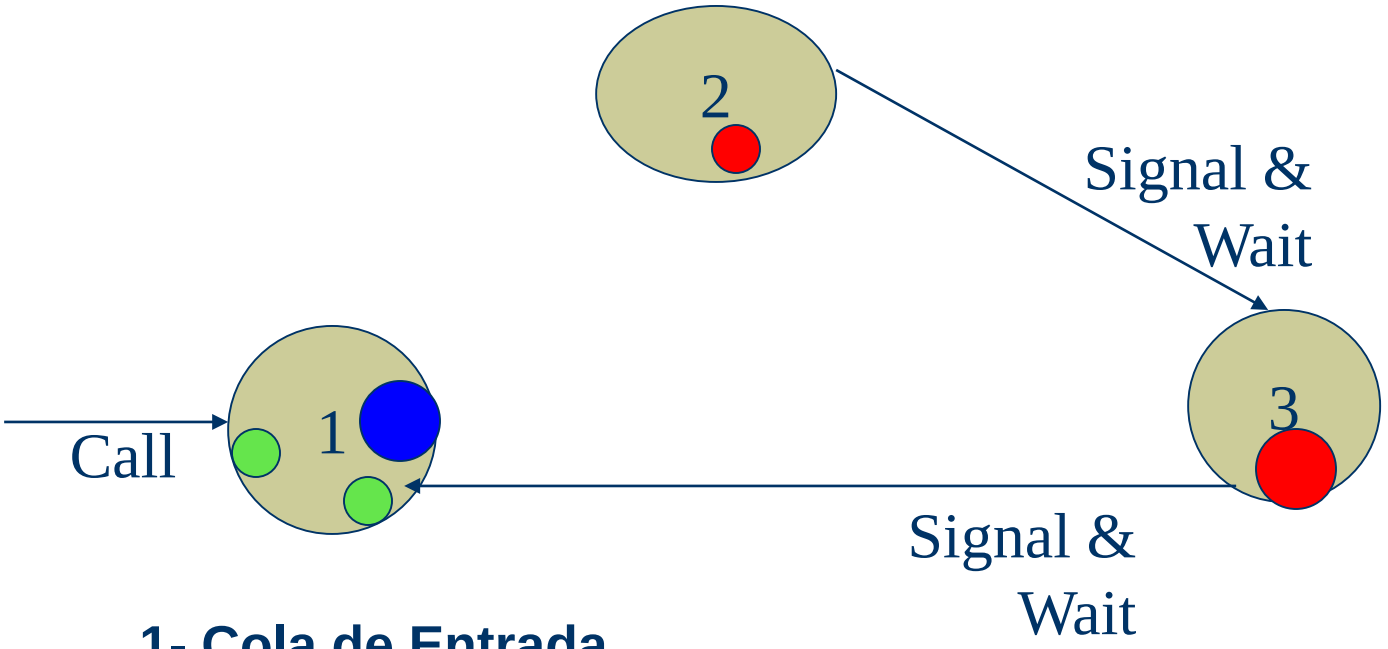
1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización

Disciplina Signal and Wait



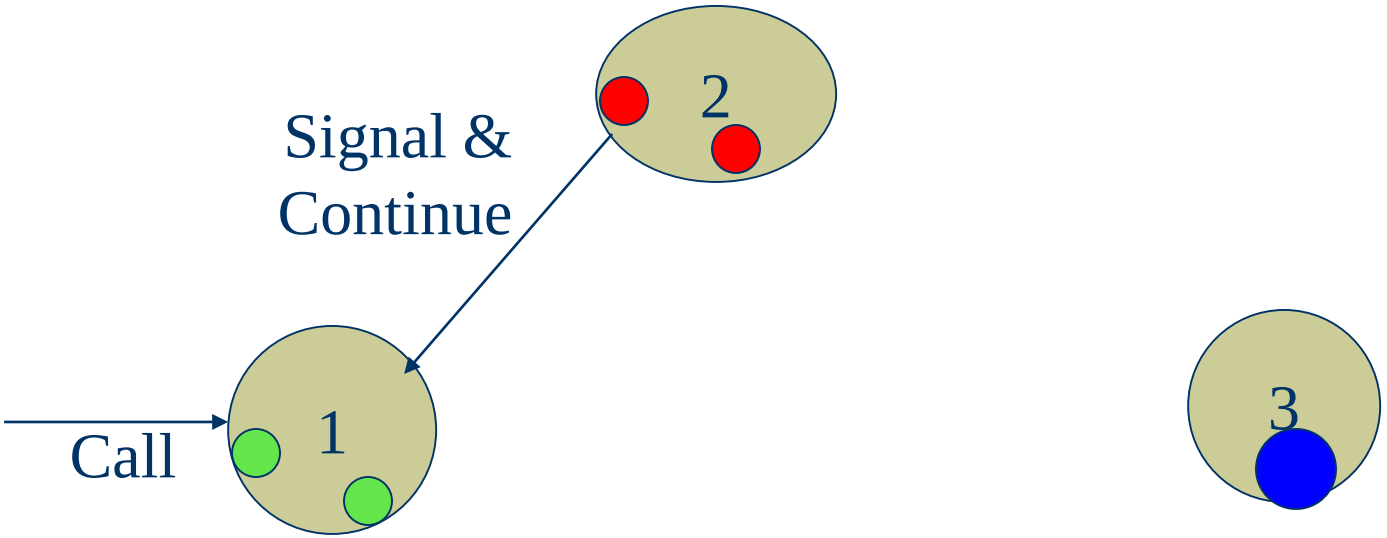
1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización

Disciplina Signal and continue



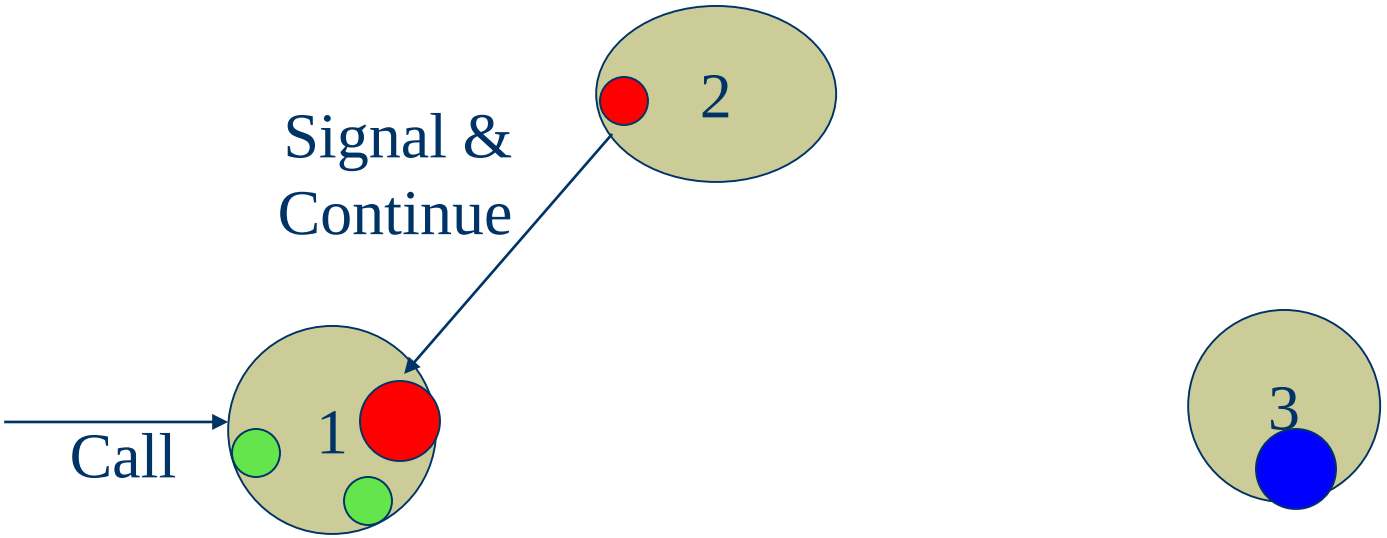
1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización

Disciplina Signal and continue



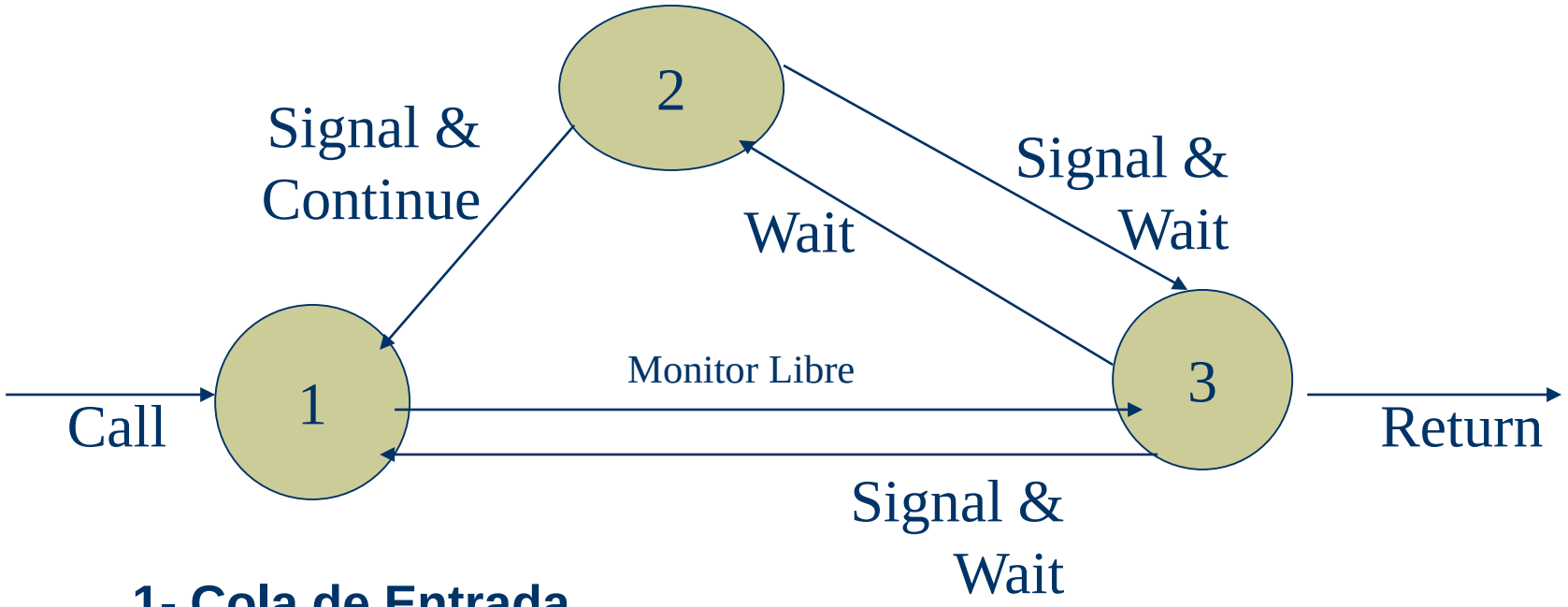
1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización

Signal and continue vs. Signal and Wait



1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

Monitores. Sincronización

Signal and continue: el proceso que ejecuta SIGNAL retiene el control exclusivo del monitor y puede seguir ejecutando, mientras el proceso despertado pasa a competir por acceder nuevamente al monitor y continuar su ejecución en la instrucción siguiente al wait (Unix, Java, Pthreads)

Signal and wait: el proceso que hace SIGNAL pasa a competir por acceder nuevamente al monitor, mientras que el despertado pasa a ejecutar dentro del monitor en la instrucción siguiente al wait.

WAIT y SIGNAL son similares a P y V, pero hay diferencias:

WAIT	**P**
El proceso siempre se duerme	El proceso sólo se duerme si el semáforo es 0.

SIGNAL	**V**
Si hay procesos dormidos despierta al primero de ellos. En caso contrario no tiene efecto posterior.	Incrementa el semáforo para que un proceso dormido o que hará un P continúe. No sigue ningún orden al despertarlos.

Monitores. Operaciones adicionales

empty(cv)→ retorna true si la cola controlada por cv está vacía.

wait(cv,rank) → wait con prioridad: permite tener más control sobre el orden en que los procesos son encolados y despertados

Los procesos demorados en cv son despertados en orden ascendente de rank.

minrank(cv) → función que retorna el mínimo ranking de demora

Con semántica S&C, signal_all es lo mismo que
while (not empty(cv)) signal(cv);

Estas operaciones no son usadas en la práctica de la materia

Ejemplo – Simulación de semáforos: *condición básica*

```
monitor Semaforo
{  int s = 1;   cond pos;

   procedure  P ()
      { if (s == 0) wait(pos);
        s = s-1;
      };

   procedure V ()
     { s = s+1;
       signal(pos);
     };
};
```

Puede quedar el semáforo con un valor menor a 0 (no cumple las propiedades de los semáforos).

```
monitor Semaforo
{  int s = 1;   cond pos;

   procedure  P ()
      { while (s == 0) wait(pos);
        s = s-1;
      };

   procedure V ()
     { s = s+1;
       signal(pos);
     };
};
```

¿Qué diferencia hay con los semáforos?

¿Que pasa si se quiere que los procesos pasen el P en el orden en que llegan?

Técnicas de sincronización – Simulación de semáforos: *passing the condition*

Simulación de Semáforos

```
monitor Semaforo
{  int s = 1;   cond pos;

   procedure  P ()
      { if (s == 0) wait(pos)
         else s = s-1;
      };

   procedure V ()
     { if (empty(pos) ) s = s+1
         else signal(pos);
      };
};
```

Como resolver este problema al no contar con la sentencia *empty*.

```
monitor Semaforo
{  int s = 1, espera = 0;  cond pos;

   procedure  P ()
      { if (s == 0) { espera ++; wait(pos);}
         else  s = s-1;
      };

   procedure V ()
     { if (espera == 0 ) s = s+1
         else { espera --; signal(pos);}
      };
};
```

Monitores. Alocación SJN

```
monitor Shortest_Job_Next {
bool libre = true;
cond turno; # Signal cuando recurso está disponible
procedure request(int tiempo) {
  if (libre)
     libre = false;
  else # lo duermo ordenado x tiempo
     wait(turno, tiempo);
  }

procedure release() {
  if (empty(turno))
     libre = true
  else
      signal(turno);
 }
}
```

- ***wait*** con prioridad para ordenar los procesos demorados por la cantidad de tiempo que usarán el recurso.
- ***empty*** para determinar si hay procesos demorados.
- Cuando el recurso es liberado, si hay procesos demorados se despierta al que tiene mínimo ***rank***.
- ***Wait*** no se pone en un loop pues la decisión de cuándo puede continuar un proceso la hace el proceso que libera el recurso.

Cómo se resuelve sin Wait con prioridad?

Monitores. Alocación SJN

Manejo del orden explícitamente usando una cola ordenada y variables condición privadas

```
monitor Shortest_Job_Next
{ bool libre = true;
  cond turno[N];
  cola  espera;

  procedure request (int id, int tiempo)
     { if (libre) libre = false
       else { insertar_ordenado(espera, id, tiempo);
              wait (turno[id]);
            };
     };

  procedure release ()
    { if (empty(espera)) libre = true
      else   { sacar(espera, id);
               signal(turno[id]);
             };
    };
}
```

Monitores. Buffer limitado

```
monitor Buffer_Limitado {
  typeT buf[n];              # array de algún tipo T
  int ocupado = 0,           # índice del primer slot lleno
      libre = 0;             # índice del primer slot vacío
      cantidad = 0;          # cantidad de slots llenos
                             # rear == (front + count) mod n
  cond not_lleno,            # signal cuando count < n
          not_vacio;         # signal cuando count > 0
  procedure depositar(typeT datos) {
       while (cantidad == n) wait (not_lleno);
       buf[libre] = datos; libre = (libre+1) mod n; cantidad++;
       signal(not_vacio);    }
  procedure retirar(typeT &resultado) {
       while (cantidad == 0) wait(not_vacio);
       resultado=buf[ocupado]; ocupado=(ocupado+1) mod n; cantidad--;
       signal(not_lleno);        }
}
```

Lectores y Escritores con monitores. *Broadcast signal*

Qué abstrae el monitor en este caso???

El monitor arbitra el *acceso* a la BD

Los procesos dicen cuándo quieren acceder y cuándo terminaron $\Rightarrow$

requieren un monitor con 4 procedures: pedido_leer, libera_leer, pedido_escribir y libera_escribir

nr número de lectores y nw número de escritores (*variables permanentes*).

Invariante: RW: $(nr = 0 \vee nw = 0) \wedge nw \leq 1$

Lectores y Escritores con monitores. *Broadcast signal*

```
monitor Controlador_RW {
        int nr = 0, nw = 0
        cond okleer             # signal cuando nw = 0
        cond okescribir         # signal cuando nr = 0 ∧ nw = 0
procedure pedido_leer( ) {
        while (nw > 0) wait(okleer);
        nr = nr + 1;    }
procedure libera_leer( )   {
        nr = nr - 1;
        if (nr == 0) signal(okescribir);   }
procedure pedido_escribir( ) {
        while (nr > 0 OR nw > 0) wait(okescribir);
        nw = nw + 1;     }
procedure libera_escribir( )   {
        nw = nw - 1;
        signal(okescribir);
        signal_all(okleer);    }
}
```

Diseño de un Reloj Lógico

Timer que permite a los procesos dormirse una cantidad de unidades de tiempo.

Se plantean dos soluciones:
- covering conditions (vble para la cual la condición booleana asociada "cubre" las condiciones por las cuales esperan los distintos procesos)
- wait con prioridad

Ejemplo de *controlador de recurso* (reloj lógico) con dos operaciones:

1) demorar(intervalo)
Efecto: demora al llamador durante *intervalo* ticks de reloj
Es llamada por los procesos de aplicación

2) tick
Efecto: incrementa el valor del reloj lógico
Es llamada por un proceso que es despertado periódicamente por un timer de hardware y tiene alta prioridad de ejecución

Diseño de un Reloj Lógico. Solución 1 – *covering conditions*

```
monitor Timer {
      int hora_actual = 0;  # invariante CLOCK: hora_actual>=0 AND tod
                            crece monótonamente de a 1
      cond chequear         # signal cuando hora_actual fue incrementada

procedure demorar(int intervalo)  {
       int hora_de_despertar;
       hora_de_despertar = hora_actual + intervalo;
      while (hora_de_despertar > hora_actual) wait(chequear);
      }

procedure tick( )   {
      hora_actual = hora_actual + 1;
      signal_all(chequear);
      }
  }

INEFICIENTE ...
```

Diseño de un Reloj Lógico. Solución 2 – wait con prioridad

```
monitor Timer {
      int hora_actual = 0;  #  invariante  CLOCK:  hora_actual>=0  AND
                            hora_actual crece monótonamente de a 1
      cond espera           # signal cuando minrank(chequear) <=
                            hora_actual

procedure demorar(int intervalo)  {
       int hora_de_despertar;
       hora_de_despertar = hora_actual + intervalo;
       If  (hora_de_despertar > hora_actual)
                     wait(espera, hora_de_despertar);
       }

procedure tick( )   {
     hora_actual = hora_actual + 1;
     while (not empty(espera) AND minrank(espera) <= hora_actual)
            signal(espera);
       }
  }
```

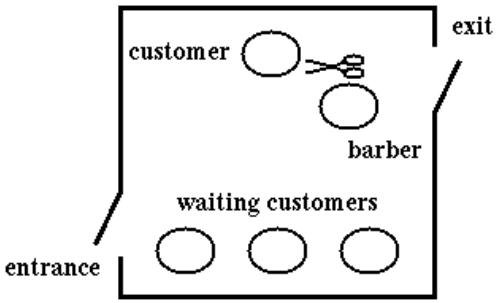
Diseño de un Reloj Lógico. Solución 2 – (sin wait con prioridad pero con vbles condition privadas)

```
monitor Timer {
        int hora_actual = 0;
        cond espera[N];
        cola_ordenada dormidos;

procedure demorar(int intervalo, int id)  {
        int hora_de_despertar;
        hora_de_despertar = hora_actual + intervalo;
        If  (hora_de_despertar > hora_actual)
                        { insertar(dormidos, id, hora_de_despertar);
                         wait(espera[id]);    }
        }

procedure tick( )   {
     hora_actual = hora_actual + 1; idaux = verPrimero(dormidos);
     while (idaux <= hora_actual)
              { sacar (dormidos, idaux);
               signal(espera[idaux]); idaux = verPrimero(dormidos); }
        }
   }
```

Monitores. El problema del peluquero dormilón (*Rendezvous*)

***Problema del peluquero dormilón* (*sleeping barber*).**

Una ciudad tiene una peluquería con 2 puertas y unas pocas sillas. Los clientes entran por una puerta y salen por la otra. Como el negocio es chico, a lo sumo un cliente o el peluquero se pueden mover en él a la vez. El peluquero pasa su tiempo atendiendo clientes, uno por vez. Cuando no hay ninguno, el peluquero duerme en su silla. Cuando llega un cliente y encuentra que el peluquero está durmiendo, el cliente lo despierta, se sienta en la silla del peluquero, y duerme mientras el peluquero le corta el pelo. Si el peluquero está ocupado cuando llega un cliente, éste se va a dormir en una de las otras sillas. Después de un corte de pelo, el peluquero abre la puerta de salida para el cliente y la cierra cuando el cliente se va. Si hay clientes esperando, el peluquero despierta a uno y espera que se siente. Sino, se vuelve a dormir hasta que llegue un cliente.



Monitores. El problema del peluquero dormilón (*Rendezvous*)

Procesos → clientes y peluquero

Monitor → administrador de la peluquería

Relación cliente/servidor.

Tres procedures:

✓**corte_de_pelo:** llamado por los clientes, que retornan luego de recibir un corte de pelo

✓**proximo_cliente:**
llamado por el peluquero para esperar que un cliente se siente en su silla, y luego le corta el pelo

✓**corte_terminado:**
llamado por el peluquero para que el cliente deje la peluquería

Monitores. El problema del peluquero dormilón (*Rendezvous*)

El peluquero y un cliente necesitan ***rendezvous***: el peluquero tiene que esperar que llegue un cliente, y este tiene que esperar que el peluquero esté disponible.

El cliente necesita esperar que el peluquero termine de cortarle el pelo, indicado cuando le abre la puerta de salida.

Antes de cerrar la puerta de salida, el peluquero necesita esperar hasta que el cliente haya dejado el negocio.

$\Rightarrow$ el peluquero y el cliente atraviesan una serie de *etapas sincronizadas*, comenzando con un *rendezvous* similar a una barrera de dos procesos pues ambas partes deben arribar antes de que cualquiera pueda seguir.

Monitores. El problema del peluquero dormilón (*Rendezvous*)

Podemos usar contadores incrementales para especificar las etapas de sincronización.

Etapas del cliente:
- sentarse en la silla del peluquero (*c_ensilla*)
- dejar la peluquería (*c_salir*)

Etapas del peluquero:
- estar disponible (*p_disponible*)
- estar cortando el pelo (*p _ocupado*)
- terminar de cortar el pelo (*p _listo*)

Condiciones:

C1: c _ensilla $\geq$ c _salir $\wedge$ p_disponible $\geq$ p_ocupado $\geq$ p_listo

C2: c_ensilla $\leq$ p_disponible $\wedge$ p_ocupado $\leq$ c_ensilla

C3: c_salir $\leq$ p_listo

$\Rightarrow$ Invariante: PELUQUERO: { C1 AND C2 AND C3 }

Monitores. El problema del peluquero dormilón (*Rendezvous*)

Problema: los valores de los contadores pueden crecer sin límite
$\rightarrow$ cambio de variables

Podemos hacerlo siempre que, como aquí, la sincronización depende sólo de las diferencias entre valores de los contadores

peluquero = p_disponible – c_ensilla
silla = c_ensilla – p_ocupado
abierto = p_listo – c_salir

Las nuevas variables satisfacen el invariante:

PELUQUERO': $\{ 0 \leq \text{peluquero} \leq 1 \ \wedge \ 0 \leq \text{silla} \leq 1 \ \wedge \ 0 \leq \text{abierto} \leq 1\}$

Monitores. El problema del peluquero dormilón (*Rendezvous*)

peluquero es 1 cuando el peluquero está esperando a que un cliente se siente en su silla

silla es 1 cuando el cliente se sentó en la silla pero el peluquero aún no está ocupado

abierto es 1 cuando la puerta de salida fue abierta pero el cliente todavía no se fue

Necesitamos implementar sincronización con vbles condición:

• Usamos 4 variables condición, una para cada una de las cuatro distintas condiciones booleanas.

- Clientes esperando que el peluquero esté libre
- Clientes esperando que el peluquero abra la puerta
- Peluquero esperando que lleguen clientes
- Peluquero esperando que se vaya un cliente

• Luego, agregamos signal donde las condiciones se hacen true

Monitores. El problema del peluquero dormilón (*Rendezvous*)

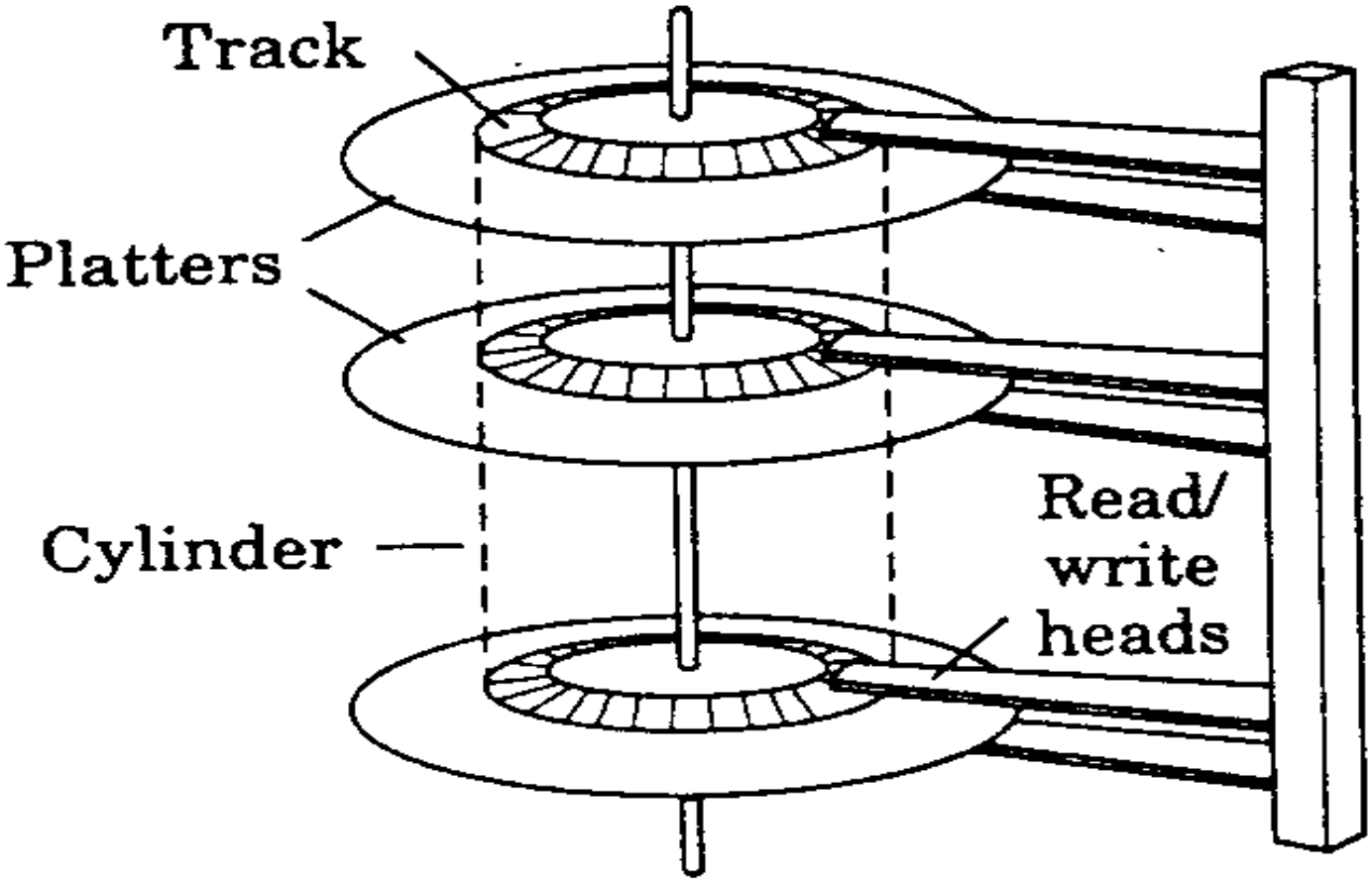
```
monitor Peluqueria {
    int peluquero = 0, silla = 0, abierto = 0;
    cond peluquero_disponible;   # signal cuando peluquero > 0
    cond silla_ocupada;          # signal cuando silla > 0
    cond puerta_abierta;         # signal cuando abierto > 0
    cond salio_cliente;          # signal cuando abierto == 0
    procedure corte_de_pelo() {
        while (peluquero == 0) wait(peluquero_disponible);
        peluquero = peluquero - 1;
        silla = silla + 1; signal(silla_ocupada);
        while (abierto == 0) wait(puerta_abierta);
        abierto = abierto - 1; signal(salio_cliente);  }
    procedure proximo_cliente() {
          peluquero = peluquero + 1; signal(peluquero_disponible);
          while (silla == 0) wait(silla_ocupada);
          silla = silla - 1;                              }
    procedure corte_terminado() {
          abierto = abierto + 1; signal(puerta_abierta);
          while (abierto > 0) wait(salio_cliente);              }
     }
```

Monitores. El problema del peluquero dormilón (*Rendezvous*)

El procedure corte_de_pelo tiene dos sentencias wait, ya que el proceso cliente pasa por dos etapas: esperar a que el peluquero esté libre, y después esperar a que le abra la puerta cuando terminó de cortarle el pelo

TAREA PROPUESTA:
Analizar cómo se modificaría la solución si fueran DOS peluqueros.

Monitores. Scheduling de disco



Monitores. Scheduling de disco

El disco contiene “platos” conectados a un eje central y que rotan a velocidad constante. Las pistas forman círculos concéntricos
$\Rightarrow$ **concepto de cilindro de información.**

Los datos se acceden posicionando una cabeza lectora/escritora sobre la pista apropiada, y luego esperando que el plato rote hasta que el dato pase por la cabeza.

dirección física $\rightarrow$ cilindro, número de pista, y desplazamiento

Para acceder al disco, un programa ejecuta una instrucción de E/S específica
Los parámetros para esa instrucción son:
dirección física del disco, el número de bytes a transferir, el tipo de transferencia a realizar (read o write), y la dirección de un buffer.

Monitores. Scheduling de disco

El tiempo de acceso al disco depende de tres cantidades:

a) seek time para mover una cabeza al cilindro apropiado

b) rotational delay

c) transmission time (depende solo del número de bytes)

a) y b) dependen del estado del disco (seek time >> rotational delay).

⇒ para reducir el tiempo de acceso promedio conviene minimizar el movimiento de la cabeza (reducir el tiempo de seek)

Monitores. Scheduling de disco

El scheduling de disco puede tener distintas políticas:

Shortest-Seek-Time (SST):
Selecciona siempre el pedido pendiente que quiere el cilindro más cercano al actual. Es unfair

SCAN, LOOK, o algoritmo del ascensor:
Se sirven pedidos en una dirección y luego se invierte. Es fair.
Problema: un pedido pendiente justo detrás de la posición actual de la cabeza no será servido hasta que la cabeza llegue al final y vuelva (gran varianza del tiempo de espera).

CSCAN o CLOOK:
se atienden pedidos en una sola dirección. Es fair y reduce la varianza del tiempo de espera.

Scheduling de disco con un monitor separado

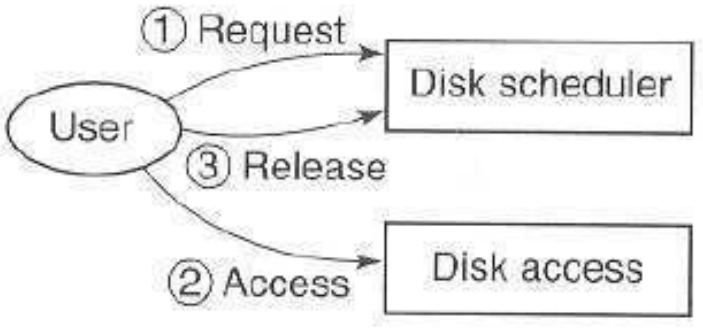
El scheduler es implementado por un monitor para que los datos sean accedidos solo por un proceso usuario a la vez.

El monitor provee dos operaciones: pedir y liberar.

Un proceso usuario que quiere acceder al cilindro *cil* llama a ***pedir(cil)***, y retoma el control cuando el scheduler seleccionó su pedido. Luego, el proceso usuario accede al disco (llamando a un procedure o comunicándose con un proceso manejador del disco).

Luego de acceder al disco, el usuario llama a ***liberar.***

Scheduler_Disco.pedir(cil)
Accede al disco
Scheduler_Disco.liberar( )



Scheduling de disco con un monitor separado

Suponemos cilindros numerados de 0 a MAXCIL y scheduling CSCAN.
A lo sumo un proceso a la vez puede tener permiso para usar el disco, y los pedidos pendientes son servidos en orden CSCAN.

posicion es la variable que indica posición corriente de la cabeza (cilindro que está siendo accedido por el proceso que está usando el disco).

Para implementar CSCAN, hay que distinguir entre los pedidos pendientes a ser servidos en el scan corriente y los que serán servidos en el próximo scan.

Scheduling de disco con un monitor separado

```
monitor Scheduler_Disco {
     int posicion = -1, v-actual = 0, v-proxima = 1;
     cond scan[2];          # scan[v-actual] signa cuando el disco es
                            liberado

procedure pedir(int cil) {
    if (posicion == -1)     # disco libre, entonces ok para acceder
        posicion = cil;
    elseif (posicion != -1 && cil > posicion)
        wait(scan[v-actual],cil);
    else
        wait(scan[v-proxima],cil);
}
```

Scheduling de disco con un monitor separado

```
procedure liberar() {

    if (!empty(scan[v-actual]))
       posicion = minrank(scan[v-actual]);

    elseif (empty(scan[v-actual]) && !empty(scan[v-proxima])) {
          v-actual :=: v-proxima;     # swap v-actual y v-proxima
          posicion = minrank(scan[v-actual]);
     }
     else
       posicion = -1;
     signal(scan[v-actual]);
 }
}
```

Scheduling de disco con un monitor intermediario

Problemas de la solución anterior:
- la presencia del scheduler es visible al proceso que usa el disco. Si se borra el scheduler, los procesos usuario cambian
- todos los procesos usuario deben seguir el protocolo de acceso. Si alguno no lo hace, el scheduling falla
- luego de obtener el acceso, el proceso debe comunicarse con el driver de acceso al disco a través de 2 instancias de buffer limitado

$\Rightarrow$ 3 monitores (scheduler y 2 buffers limitados) y 4 llamados

Posible solución: relación C/S $\Rightarrow$ 2 monitores: uno para scheduling y otro para interacción entre procesos usuario y el driver de disco

MEJOR: usar un monitor como intermediario entre los procesos usuario y el disk driver. El monitor envía los pedidos al disk driver en el orden de preferencia deseado

Scheduling de disco con un monitor intermediario

Mejoras:

-La interfase al disco usa un solo monitor, y los usuarios hacen un solo llamado al monitor x acceso al disco

-La existencia o no de scheduling es transparente

-No hay un protocolo multipaso que deba seguir el usuario y en el cual pueda fallar

Transformamos la solución del peluquero en una interfase disk driver que brinda comunicación entre clientes y el driver, e implementa scheduling CSCAN

- Se agregan parámetros en los procedures para enviar pedidos desde los usuarios (clientes) y el driver (peluquero), y p/ recibir resultados
⇒ Las puertas de E/ y S/ se convierten en buffers de comunicación

- Se agrega scheduling al rendezvous usuario/driver para atender los pedidos en el orden deseado

Scheduling de disco con un monitor intermediario

monitor Interfase_al_Disco
Variables permanent para estado, scheduling y transferencia de datos

procedure usar_disco(int cil, parámetros de transferencia y resultados**) {**
esperar turno para usar el manejador
almacenar parámetros de transferencia en variables permanentes
esperar que se complete la transferencia
recuperar resultados desde las variables permanentes
}
procedure buscar_proximo_pedido(algunType &resultados) {
seleccionar próximo pedido
esperar a que se almacenen los parámetros de transferencia
setear **resultados** a los parámetros de transferencia
}
procedure transferencia_terminada(algunType resultados) {
almacenar los resultados en variables permanentes
esperar a que **resultados** sean recuperados por el cliente
}
}

Scheduling de disco con un monitor intermediario

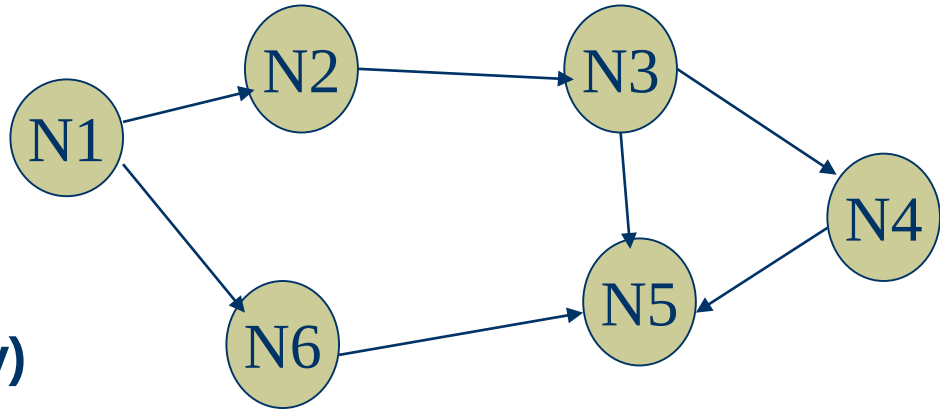
```
monitor Interfase_al_disco {
   int posicion = -2, v-actual = 0, v-proxima = 1, args = 0, resultados = 0;
   cond scan[2];
   cond args_almacenados, resultados_almacenados, resultados_recuperados;
   argType area_arg; resultadoType area_resultado;

   procedure usar_disco(int cil; argType parametros_transferencia;
     resultType &parametros_resultado) {
     if (posicion == -1)
        posicion = cil;
     elseif (posicion != -1 and cil > posicion)
        wait(scan[v-actual],cil);
     else
        wait(scan[v-proxima],cil);
     area_arg = parametros_transferencia;
     args = args+1; signal(args_almacenados);
     while (resultados == 0) wait(resultados_almacenados);
    parametros_resultado = area_resultado;
    resultados = resultados-1; signal(resultados_recuperados);
   }
```

Scheduling de disco con un monitor intermediario

```
procedure buscar_proximo_pedido(argType &parametros_transferencia) {
    int temp;
    if (!empty(scan[v-actual]))
        posicion = minrank(scan[v-actual]);
    elseif (empty(scan[v-actual]) && !empty(scan[v-proxima])) {
        v-actual :=: v-proxima;    # swap v-actual and v-proxima
        posicion = minrank(scan[v-actual]);
     }
    else
         posicion = -1;
    signal(scan[v-actual]);
    while (args == 0) wait(args_almacenados);
    parametros_transferencia = area_arg; args = args-1;  }
procedure transferencia_terminada(resultType valores_resultado) {
       area_resultado := valores_resultado;
       resultados = resultados+1;
       signal(resultados_almacenados);
       while (resultados > 0) wait(resultados_recuperados);  }
 }
```

Casos problema de interés: Grafo de precedencia

Analizar el problema de modelizar N procesos que deben sincronizar, de acuerdo a lo especificado por un grafo de precedencia arbitrario (con semáforos)



Process N [i:1..6] {
esperar a predecesores (si hay)
ejecutar la tarea
señalizar a sucesores (si hay) }

Ej: en este caso N4 hará P(N3) y V(N5), N1 hará V(N2) y V(N6)

Casos problema de interés: Productores/Consumidores con broadcast

En el problema del buffer atómico, sea UN proceso productor y N procesos consumidores.
El Productor DEPOSITA y debe esperar que TODOS los consumidores consuman el mismo mensaje (broadcast).
Notar la diferencia entre una solución por memoria compartida y por mensajes.

Versión más compleja: buffer con K lugares, UN productor y N consumidores.
El productor puede depositar hasta K mensajes, los N consumidores deben leer cada mensaje para que el lugar se libere y el orden de lectura de cada consumidor debe ser FIFO.
Analizar el tema con semáforos.

Casos problema de interés: Variantes del problema de los filósofos

Si en lugar de administrar tenedores, cada filósofo tiene un estado (comiendo, pensando, con hambre) y debe consultar a sus dos filósofos laterales para saber si puede comer, tendremos una solución distribuida. Se podría usar la técnica de "passing the baton" para resolverlo?

Otra alternativa es tener una solución de filósofos centralizada, en la que un scheduler administra por ejemplo los tenedores y los asigna con posiciones fijas o variables (notar la diferencia).

En la solución que vimos de filósofos, para evitar deadlock utilizamos un código distinto para un filósofo (orden de toma de los tenedores). Otra alternativa es la de la elección al azar del primer tenedor a tratar de tomar... Cómo?

Casos problema de interés: El problema de los baños

Un baño único para varones o mujeres (excluyente) sin límite de usuarios.

Un baño único para varones o mujeres (excluyente) con un número máximo de ocupantes simultáneos (que puede ser diferente para varones y mujeres)

Dos baños utilizables simultáneamente por un número máximo de varones (K1) y de mujeres (K2), con una restricción adicional respecto que el total de usuarios debe ser menor que K3 (K3 < K1 + K2).

Casos problema de interés:
El problema de la molécula de agua

Existen procesos O (oxígeno) y H (hidrógeno) que en un determinado momento toman un estado "listo" y se buscan para combinarse formando una molécula de agua (HH0).

Puede pensarse en un esquema C/S, donde el servidor recibe los pedidos y cuando tiene 2 H *listos* y 1 O *listo* concreta la molécula de agua y libera los procesos H y O.

También puede pensarse como un esquema "passing the baton" que puede iniciarse por cualquiera de los dos H o por el O, pero tiene un orden determinado (analizar con cuidado).

Podría pensar la solución con un esquema de productores-consumidores?
Quiénes serían los productores y quienes los consumidores?

Casos problema de interés: El puente de una sola vía

Suponga un puente de una sola vía que es accedido por sus extremos (procesos *Norte* y procesos *Sur*).
Cómo especificaría la exclusión mutua sobre el puente, de modo que circulen los vehículos (procesos) en una sola dirección.

Es un caso típico donde es difícil asegurar fairness y no inanición. Por qué? Cuál podría ser un método para asegurar no inanición con un scheduler? Qué ideas propone para tener fairness entre Norte y Sur ?

Suponga que cruzar el puente requiere a lo sumo 3 minutos y Ud. quiere implementar una solución tipo "time sharing" entre los procesos Norte y Sur, habilitando alternativamente el puente 15 minutos en una y otra dirección. Cómo lo esquematizaría?

Casos problemas de interés: semáforos/monitores.

Administrador de páginas de memoria.

Los pedidos (Request) y las liberaciones (Release) tienen una CANTIDAD de páginas como parámetro.

Analizar la resolución con semáforos, y monitores.
La condición para satisfacer el Request es tener el número de páginas disponible.

Ahora el problema "contiene" una política de scheduling.
A- SJN (menor pedido primero)
B- FIFO
C- GJN (mayor pedido primero).

Discuta las modificaciones a las soluciones anteriores.

Casos problemas de interés: semáforos/monitores.

N procesos P[i] comparten 2 impresoras identificadas r1, r2.

Los pedidos de impresora (Request) pueden tener la identificación de la impresora o xx (cualquiera).

Las liberaciones (Release) tiene la identificación de la impresora.

Analizar la resolución con semáforos y monitores.
La condición de atender el Request es la impresora disponible.

Ahora el problema "contiene" una política de scheduling.
A- Por prioridad de proceso.
B- Por menor número de hojas prometidas a imprimir.
Discuta las modificaciones a las soluciones anteriores.

Casos problemas de interés: semáforos/monitores.

El problema de los "baby birds", "parent bird" y el plato de F porciones de comida.

N consumidores con dos actividades.
Un productor múltiple.

El problema de las "abejas productoras de miel", el "oso" y el pote de H porciones de miel.

N productores con dos actividades.
Un consumidor múltiple.

Casos problema de interés: Search – Insert – Delete sobre una lista con procesos concurrentes

Una generalización de la EM selectiva e/ clases de procesos visto con lectores-escritores es el problema de procesos que acceden a una lista enlazada para *Buscar* (search), *Insertar* al final o *Borrar* un elemento en cualquier posición.

Los procesos que BORRAN se excluyen entre sí y además excluyen a los procesos que buscan e insertan.
Sólo un proceso de borrado puede acceder a la lista.

Los procesos de inserción se excluyen entre sí, pero pueden coexistir con los de búsqueda. A su vez, los de búsqueda NO se excluyen entre sí

Casos problema de interés: Modificaciones a SJN

Suponga que se quiere cambiar el esquema de alocación SJN por uno LJN (*longest JOB next*). Analice el código y comente los cambios. Cuál de los dos esquemas le parece más fair? Cuál generará una cola mayor de procesos pendientes?

En el esquema SJN, suponga que lo quiere cambiar por un esquema FIFO. Analice el código y comente los cambios. Cuál de los dos esquemas le parece más eficiente? Más Fair ?

Casos problema de interés: Drinking Philosophers

Investigar el tema definido por Chandy y Misra en 1984.
Se trata de una generalización del problema de los dining philosophers.

Analizar las diferencias de los dos problemas, estudiar los mecanismos de sincronización y discutir las ventajas/dificultades de utilizar semáforos en la sincronización.

Pthreads

Además de semáforos, Pthreads soporta locks y variables condición. Los locks pueden usarse por sí solos p/ proteger secciones críticas, o en combinación con variables condición para simular monitores.

Los locks en Pthreads se llaman *mutex locks,* pues se usan para implementar exclusión mutua. Declaración e inicialización:

pthread_mutex_t mutex; ... pthread_mutex_init(&mutex,NULL);

Una sección crítica usa mutex de la siguiente manera:

pthread_mutex_lock(&mutex); SC; pthread_mutex_unlock(&mutex);

El unlock puede ser ejecutado *solo* por el thread que mantiene el lock

Declaración e inicialización de una variable condición:

pthread_cond_t cond; ... pthread_cond_init(&cond,NULL);

Pthreads

Operaciones principales: wait, signal y broadcast. Deben ejecutarse cuando se mantiene un mutex lock. Un procedure de monitor puede simularse bloqueando un mutex al inicio del procedure y unlocking el mutex al final.

Los parámetros de *pthread_cond_wait* son una vble condición y un mutex lock. Un thread que quiere esperar primero debe lograr el lock.

Ejemplo: si un thread ya ejecutó *pthread_mutex_lock(&mutex)* y luego ejecuta *pthread_cond_wait(&cond, &mutex)*, esto hace que el thread libere *mutex* y espere en *cond*.
Cuando reanuda la ejecución luego de un signal o broadcast, el thread nuevamente obtiene *mutex* y estará bloqueado.
Cuando otro thread ejecuta *pthread_cond_signal(&cond)*, despierta a un thread (si hay alguno bloqueado), pero continúa ejecutando y manteniendo *mutex*.

Pthreads. Suma paralela de los elementos de una matriz

Un número de threads (***numWorkers***) suman los elementos de una matriz compartida con ***size*** filas y columnas. Matrix ([size], [size])

Caso típico de paralelismo iterativo con memoria compartida.

TAREA PROPUESTA: Ver la solución en Pthreads con monitores (Andrews)

Pthreads. Suma paralela de los elementos de una matriz

```
#include <pthread.h>
#include <stdio.h>
#define SHARED 1
#define MAXSIZE 2000         /* maximum matrix size */
#define MAXWORKERS 4         /* maximum number of workers */
pthread_mutex_t barrier;     /* lock for the barrier */
pthread_cond_t go;           /* condition variable */
int numWorkers;              /* number of worker threads */
int numArrived = 0;          /* number who have arrived */
```

Pthreads. Suma paralela de los elementos de una matriz

```
/* a reusable counter barrier */

void Barrier() {
    pthread_mutex_lock(&barrier);
    numArrived++;
    if (numArrived < numWorkers)
          pthread_cond_wait(&go, &barrier);
    else {
        numArrived = 0;    /* last worker awakens others */
        pthread_cond_broadcast(&go);
     }
pthread_mutex_unlock(&barrier);
}
```

Pthreads. Suma paralela de los elementos de una matriz

```
void *Worker(void *);
int size, stripSize;                /* size == stripSize*numWorkers */
int sums[MAXWORKERS];    /* sums computed by each worker */
int matrix[MAXSIZE][MAXSIZE];
/* read command line, initialize, and create threads */
int main(int argc, char *argv[ ]) {
     int i, j;
     pthread_attr_t attr;
     pthread_t workerid[MAXWORKERS];
/* set global thread attributes */
     pthread_attr_init(&attr);
     pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* initialize mutex and condition variable */
     pthread_mutex_init(&barrier, NULL);
     pthread_cond_init(&go, NULL);
```

Pthreads. Suma paralela de los elementos de una matriz

```
/* read command line */
     size = atoi(argv[1]);
     numWorkers = atoi(argv[2]);
     stripSize = size/numWorkers;
/* initialize the matrix */
     for (i = 0; i < size; i++)
          for (j = 0; j < size; j++)
          matrix[ i ] [ j ] = 1;
/* create the workers, then exit main thread */
       for (i = 0; i < numWorkers; i++)
            pthread_create(&workerid[ i ], &attr, Worker, (void *) i);
            pthread_exit(NULL);     }
/* Each worker sums the values in one strip.
   After a barrier, worker(0) prints the total */
void *Worker(void *arg) {
      int myid = (int) arg;    int total, i, j, first, last;
```

Pthreads. Suma paralela de los elementos de una matriz

```
/* determine first and last rows of my strip */
      first = myid*stripSize;
      last = first + stripSize - 1;
/* sum values in my strip */
total = 0;
for (i = first; i <= last; i++)
      for (j = 0; j < size; j++)
            total += matrix[ i ][ j ];
sums[myid] = total;
Barrier();
if (myid == 0)   { /* worker 0 computes the total */
      total = 0;
      for (i = 0; i < numWorkers; i++)
            total += sums[ i ];
      printf("the total is %d\n", total);
 }      }
```

Concurrencia en Java

JAVA ⇒ lenguaje OO con soporte para concurrencia, mediante ***Threads***. Un thread es un proceso "liviano" (lightweight process) que tiene un contexto privado mínimo

Cada Thread en JAVA tiene su propio contador de programa, su pila de ejecución (stack) y su conjunto de registros (working set), pero la zona de datos es compartida por todos los threads, exigiendo sincronización.

Métodos sincronizados ⇒ Monitores

Threads en Java

Exclusión mutua implícita:

- El mecanismo de sincronización o EM básico es el provisto por la palabra clave "synchronized" declarada sobre un método.

- C/ objeto tiene un "lock" y p/ ganar acceso a éste se lo hace en un bloque protegido por la palabra "synchronized“; p/ obtenerlo debe esperar a tener acceso exclusivo, que retendrá hasta salir del bloque

Threads en Java

Exclusión mutua explícita

- JAVA permite la sincronización por condición con los métodos *wait, notify y notifyAll,* similares al “wait” y “signal” que vimos en monitores.

- Estos métodos están definidos en la superclase "Object" y siempre deben ser ejecutados en porciones de código "synchronized" (es decir, cuando un objeto está *locked*)

- El método *wait* libera el lock de un objeto y demora al thread. Hay una *única* cola de demora por objeto.

Threads en Java

Exclusión mutua explícita

- Si bien no existen las variables "condition" los "lock" implícitos en cada objeto cumplen esta función.
- El método *notify* despierta al thread que está al frente en la cola de demora, si hay.
- La semántica que tienen es de SC (Signal and Continue).
- Si un método sincronizado en un objeto contiene un llamado a otro método en otro objeto, el lock del primer objeto es retenido mientras se ejecuta el llamado.

TAREA PROPUESTA: Ver la solución de lectores-escritores en JAVA con monitores (Andrews)

Implementaciones

Cómo se provee la concurrencia a nivel de S.O?

De qué manera se representan los procesos, cómo se manejan, etc?

Cuál es la forma en que se implementan los semáforos, monitores, etc?

(Referencia: Capítulo 6 del libro de Andrews)

Tareas propuestas

Investigar los monitores como herramienta de sincronización entre procesos

Buscar información sobre problemas clásicos de sincronización entre procesos y su resolución con monitores.