CFGS DAM

Primer Curso

Programación de Servicios y Procesos

UT2: PROGRAMACIÓN CONCURRENTE

Índice

- Programación concurrente
  - Beneficios
  - Problemas
- Comunicación y sincronización entre procesos
  - Exclusión mutua.
  - Mecanismos de comunicación y sincronización entre procesos.
    - Semáforos
    - Monitores
    - Paso de mensajes
- Problemas típicos de concurrencia.

Introducción a la Programación concurrente

- Tipo de programación empleado por programadores que quieren/pueden dividir un programa en trozos/bloques con trazas de ejecución independientes.

- Puede ser:
  - **Aparente** → multiprogramación con una CPU
  - **Real** → multiprocesador o multicomputador
    - (uno o varios ordenadores formando un sistema de 2 o más CPU's)

Introducción a la Programación concurrente

- Con su aparición cambian algunas definiciones:
  - Hasta ahora un proceso era "un programa en ejecución".
  - Pero en realidad un programa puede dar lugar a mas de un proceso.
    - Ej: Navegador web (Cada proceso ejecuta una parte del programa):
      - A) Gestiona las acciones (eventos) del usuario
      - B) Gestiona las peticiones al servidor
      - Y esto por cada pestaña abierta.
- Por lo tanto a partir de ahora usaremos una **definición más acertada para proceso**:

  "Actividad asíncrona susceptible de ser asignada a un procesador"

Introducción a la Programación concurrente

- **Dos procesos serán concurrentes cuando** existe un solapamiento (o intercalado) en la ejecución de instrucciones:
  - Es decir que se ejecutan 2 instrucciones de uno, luego 3 del otro y así sucesivamente.
  - Nota: si las M instrucciones de los N procesos no se solapan, si-no que se ejecutan de forma **simultánea** en varios procesadores, se llama programación paralela.
    - Es decir: concurrente simultáneo = paralelo
- Hoy en día, casi todo se programa de forma concurrente.

Programación concurrente. Orden de ejecución

- **Programación Secuencial**
  - Determinismo: Se conoce de forma exacta que instrucción será ejecutada después de cual otra.
- **Programación Concurrente**
  - Indeterminismo: No se conoce la secuencia en la que serán ejecutadas la instrucciones.

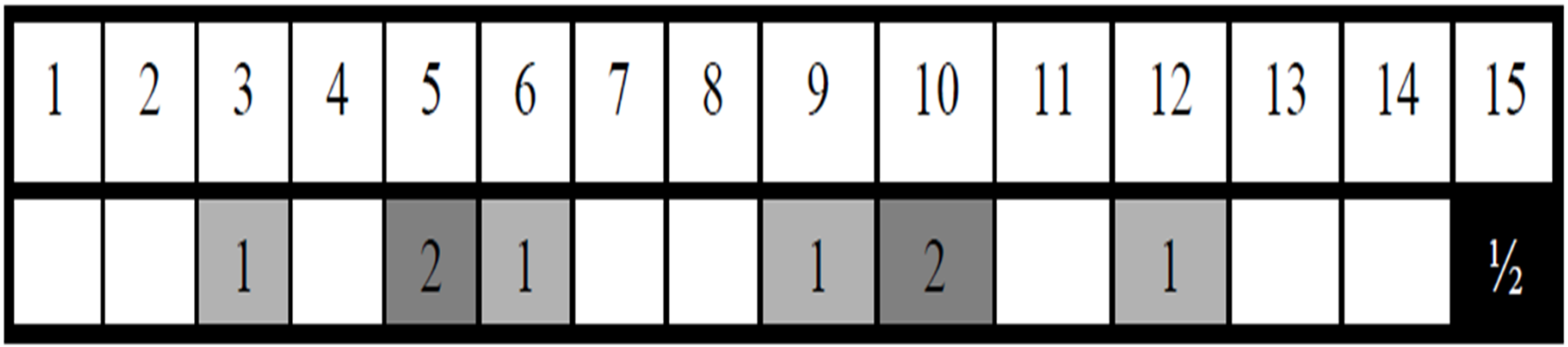
Ejemplo: P || Q

P: < p1 → p2 → p3 → ... → pm >
Q: < q1 → q2 → q3 → ... → qn >

Secuencial ➔ < p1 → p2 → ... → pm → q1 → q2 → ... → qn >
Concurrente ➔ Múltiples posbilidades.
p.e. < p1 → p2 → q1 → p3 → q2 → ... → qn → pm >
< p1 → q1 → q2 → q3 → p2 → ... → pm → qn >
< q1 → q2 → p1 → q3 → p2 → ... → pm → qn >
etc...

Programación concurrente. Ejemplo

- Supongamos que debemos realizar un programa que cada 3 seg. muestre por pantalla el mensaje "Hola" y cada 5 seg. el mensaje "Mundo".
- Pensando de forma secuencial quedaría así:



Programación concurrente. Ejemplo

Este programa se podría programar de forma secuencial y concurrente, siendo mucho más sencillo de la segunda forma como demuestra el siguiente código.
Nota: P1 y P2 serían los procesos concurrentes o secuenciales a ejecutar.

Programación secuencial

```
s1=p1;
s2=p2;
Repetir
   Si  s1 < s2  Entonces
                  Espera(s1);
                  Escribe("Hola");
                  s2 = s2 – s1;
                  s1 = p1;
   Sino Si  s1 > s2 Entonces
                  Espera(s2);
                  Escribe("Mundo");
                  s1 = s1 – s2;
                  s2 = p2;
   Sino Si s1=s2 Entonces
                  Espera(s1);
                  Escribe ("Hola Mundo");
                  s1 = p1;
                  s2 = p2;
   FinSi
FinRepetir
```

Programación concurrente

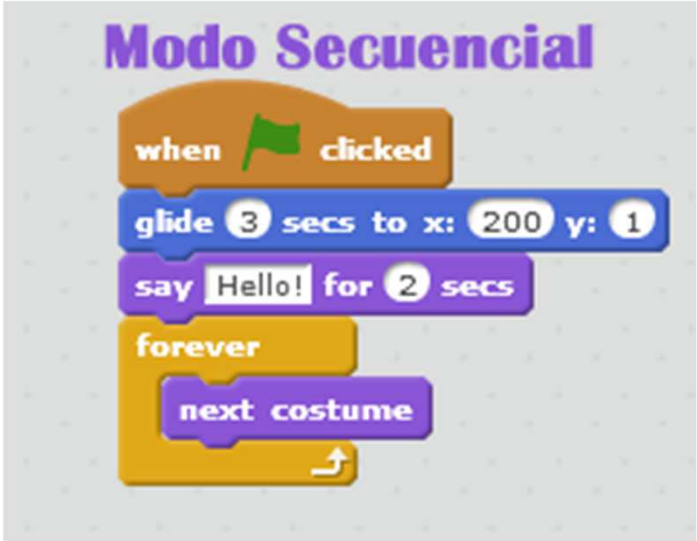
```
Proceso P1
      Esperar(3);
      Escribe("Hola");

Proceso P2
      Esperar(5);
      Escribe("Mundo");
```

Programación concurrente. Ejemplo II

- Hay cosas que no puedo programar de forma secuencial.

Ej: No podrías deslizar
al personaje y a la vez hacer
que dijera algo e hiciese una
animación de forma secuencial:



- Sin embargo si se puede hacer de forma concurrente:

Programación concurrente. Beneficios

- **Con Programación Secuencial puedo:**
  - ***Ejecutar un único trabajo muy rápido***
    - N instrucciones secuenciales son mejor que N instrucciones concurrentes, ya que no hay cambios de contexto
- **Con Programación Concurrente puedo**:
  - ***Hacer varias cosas a la vez.***
    - Por ejemplo GUI y lectura de un fichero
  - ***Aumentar la velocidad de ejecución → Divide y vence***
    - Al dividir mi programa en trozos, hay momentos en los que estos se ejecutan de forma simultánea, de forma que el programa acabará antes su ejecución.
    - Problema: No siempre se puede hacer esto
  - ***Aprovechar mejor a la CPU***
    - No se pierden "los tiempos muertos"
      - Si el proceso se para a una E/S y es "único" se queda colgado. Si está dividido entra otro trozo a la CPU, aprovechando ese tiempo muerto.
      - Además, al haber más elementos esperando siempre habrá alguien dispuesto a coger la CPU.
  - ***Solucionar problemas de naturaleza concurrente***
    - Hay determinados problemas que sólo se resuelven utilizando esta técnica, como:
      - Servidores de internet
      - Simuladores, Robots, etc.

Programación concurrente. Problemas

- Este tipo de programación implica que varios procesos compartan y compitan por el uso de recursos.
- **Si no** existe la **adecuada comunicación** y **sincronización entre procesos**, pueden aparecer **problemas de concurrencia**.
- Ejemplos:
  - Cuenta banco / Puerta de acceso
  - Productor-consumidor
  - Filósofos
  - Etc.
- Estos problemas son inherentes a la programación concurrente, por lo que **siempre debemos tenerlos en cuenta**.

Programación concurrente. Problemas

Si el sistema operativo/nuestro programa no gestionase estos problemas se producirían:

- Interbloqueos (**Deadlock**):
  - Un proceso/trozo espera un evento que no ocurrirá.
  - Por ejemplo:
    - Un proceso espera que una variable tome un valor que nunca es modificada.
    - Un proceso espera que otro le envíe un aviso que nunca llega.
    - Todos los procesos, de formar circular, esperan que otro use un recurso:
      - Considera dos procesos y dos recursos. Supón que cada proceso necesita acceder a ambos recursos para llevar a cabo una parte de su función.
      - Puede suceder que el sistema operativo asigne R1 a P1 y R2 a P2. Cada proceso está esperando uno de los dos recursos. Ninguno liberará el recurso que posee hasta que adquiera el otro y realice su tarea.
- Postergación indefinida o Inanición (**Starvation**):
  - Un proceso es desestimado a la hora de ejecutarse.
  - Por ejemplo:
    - Un proceso espera usar la CPU y nunca es seleccionado
    - Un proceso espera usar un recurso y otros se lo impiden
- Violación de la Exclusión Mutua
  - Dos o mas procesos ejecutan la Sección Critica a la vez.
  - Dos o mas procesos acceden a la vez a un recurso compartido.

Programación concurrente. Soluciones

- Para evitar estos problemas, los procesos "competidores" necesitan **mecanismos o técnicas que les faciliten la**:
  - **Comunicación**
    - Para transmitirse información entre sí
  - **Sincronización**
    - Para poder esperar y continuar cuando sea necesario, de forma que se acceda de forma ordenada a los recursos, siguiendo una secuenciación temporal, conocida y establecida entre ellos.
- Estos mecanismos se pueden garantizar por:
  - Hardware (Inhabilitando a la CPU (interrupciones))
  - Software (Con instrucciones en nuestro prog)

Comunicación y sincronización entre procesos

- Un programa concurrente será correcto si garantiza la:
- **Exclusión Mutua**.
  - Asegurar que solo un proceso tiene acceso a un recurso compartido.
- **Condición de sincronización**.
  - Asegurar que un proceso no progrese hasta que se cumpla una determinada condición.

Comunicación y sincronización entre procesos

- **Exclusión mutua**:
  - **Dos o más trozos intentan acceder a un recurso común**, como por ejemplo:
    - Una variable o trozo de memoria
    - Un fichero de disco
    - Una tabla de una base de datos
  - **Se soluciona estableciendo secciones o regiones críticas**
    - Son zonas a las que sólo puede acceder un proceso a la vez. El resto: a esperar
    - En java se hace mediante: synchronized

Comunicación y sincronización entre procesos

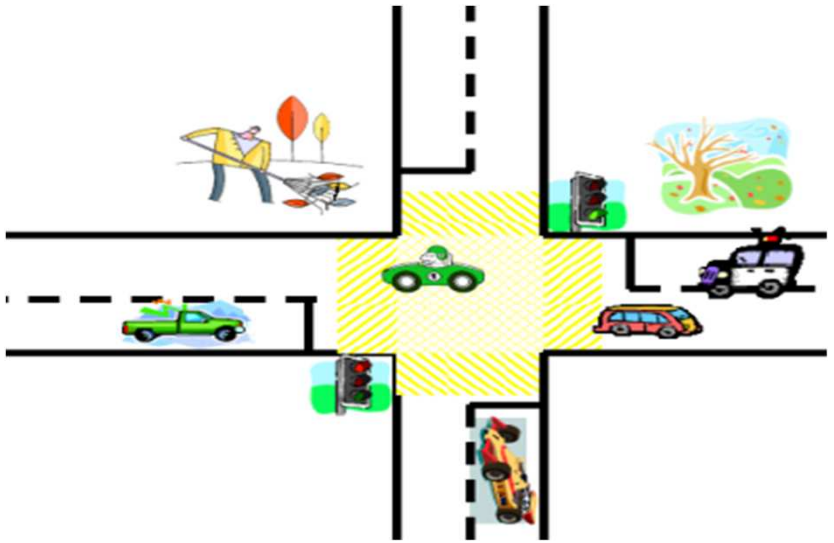
- **Condición de sincronización:**
  - Ejecución de líneas/trozos de código en orden diferente al adecuado.
    - Puede ocurrir que un proceso P1 en estado X no pueda continuar su ejecución hasta que el proceso P2 llegue al estado Y, por lo que deberá esperar.
    - Ej: Estas líneas no se pueden ejecutar en diferente orden:

      1→ **x=**x+1;

      2→ y=**x**+y
    - Ej: Receta de cocina con 2 cocineros
  - **Se soluciona mediante** la sincronización entre procesos
    - A través de **bloqueos y desbloqueos** que les detengan cuando quieran hacer algo para lo que necesiten una acción/operación previa no finalizada.
    - En java se hace mediante: wait(), notify() y notifyAll()

Exclusión mutua.

- Capacidad de un sistema o necesidad de asegurar que un único proceso acceda a un conjunto de instrucciones que acceden a un recurso compartido (por ejemplo una variable global/compartida, un fichero, etc.).
- A este conjunto de instrucciones se le denomina **Sección Crítica**.
- Cuando un proceso está **accediendo a datos compartidos** se dice que el proceso se encuentra en **sección critica**.
- La ejecución de las secciones críticas debe ser mutuamente exclusiva para evitar inconsistencia de datos.



Exclusión mutua.

- Estar dentro de una sección crítica es un estado muy especial asignado a un proceso, ya que el proceso tiene acceso exclusivo a los datos compartidos.
  - Por ello, cuando un proceso esté en su sección crítica, todos los demás procesos (o al menos aquellos que tengan acceso a los mismos datos compartidos) serán excluidos de sus propias secciones criticas.
- Las secciones críticas deben ser codificadas con todo cuidado:
  - Las SC's deben ser ejecutadas lo más rápido posible.
  - Un proceso no debe bloquearse dentro de su SC.
- Los algoritmos de exclusión mutua se usan en programación concurrente para evitar el acceso simultáneo a esas secciones críticas.
- La estructura general de cualquier mecanismo para implementar la sección crítica es la siguiente:

Barrera de entrada a la sección crítica

<<código de la sección crítica>>

Barrera de salida de la sección crítica

Exclusión mutua. Requisitos

- Cualquier sistema, servicio o capacidad que dé soporte para la exclusión mutua debe cumplir los requisitos siguientes:
  - Debe cumplirse la exclusión mutua:
    - Solo un proceso, de entre todos los que poseen secciones críticas sobre el mismo recurso u objeto compartido, debe tener permiso para entrar en ella en un instante dado.
  - No puede permitirse el interbloqueo o la inanición.
    - Un proceso no debe poder solicitar acceso a una sección crítica para después ser demorado indefinidamente.
  - Un proceso permanece en su sección crítica solo por un tiempo finito.
  - Cuando ningún proceso está en su sección crítica, cualquier proceso que solicite entrar en la suya debe poder hacerlo sin dilación.
  - Mientras un proceso se encuentra en su sección crítica, los demás procesos pueden continuar su ejecución fuera de sus secciones críticas.
  - Un proceso que se interrumpe en una sección no crítica debe hacerlo sin interferir con los otros procesos.
  - Si un proceso dentro de una sección critica termina, tanto de forma voluntaria como involuntaria, entonces al realizar su limpieza de terminación, el S.O debe liberar la exclusión mutua para que otros procesos puedan entrar en sus secciones críticas. Proceso lineal: se ejecuto el proceso, termina y empieza otro.

Exclusión mutua. Ejemplo I

- Para entender mejor la EM veamos un sencillo ejemplo ilustrativo, en el que 2 procesos acceden a una variable compartida llamada "ent" en 2 instrucciones diferentes:

```
void echo(){
        ent= getchar();
        int sal = ent;          //ent es global y sal es local
        putchar(sal);
}
```

- Esto podría perfectamente provocar la siguiente traza de ejecución:

```
/*Proceso P1*/
...
ent = getchar();
...
sal = ent;
putchar(sal);
...
...
```

```
/*Proceso P2*/
...
...
ent = getchar ();
sal = ent;
...
putchar(sal);
...
```

Nota: los puntos suspensivos indican intervalos de espera

- Resultado:
  - Se pierde el primer carácter y el segundo se visualiza dos veces. ¿Por qué?
  - Si observas bien, verás que la variable ent puede tomar un valor incorrecto fácilmente si las instrucciones se realizan en el orden equivocado, provocando la salida del segundo valor leído en el primer proceso.

Exclusión mutua. Ejemplo I. Solución

- El problema es que **todos los procesos que llaman a echo() tienen acceso a las variables "ent" y "sal".**
  - Aunque cómo solo el acceso a ent es compartido, sólo son "sensibles" esas líneas.
- Impongamos la **restricción** de que, aunque **echo()** sea un procedimiento global, **sólo puede estar ejecutándolo un proceso cada vez.**
  - **Con esto indicamos que las tres líneas de echo() son ahora sección crítica.**
  - La secuencia anterior quedaría ahora:
    - •P1 llama a echo() y es interrumpido inmediatamente después de leer la entrada.
    - •P2 se activa y llama a echo(). Como P1 estaba usando echo(), **P2 se bloquea** al entrar al procedimiento y se le suspende mientras espera que quede libre echo().
    - •P1 se reanuda y completa la ejecución de echo(). Se visualiza el carácter correcto.
    - •Cuando P1 abandona echo() se retira el bloqueo sobre P2, por lo que reanudará su ejecución y se ejecutará con éxito.
- Pero esa restricción **podría mejorarse**, ya que la última línea del método no es susceptibles de error.
  - **Ya que solo trabaja con la variable local(sal),** y cada hilo tiene la suya, por lo que no hay problemas de concurrencia.
- **Solución óptima**: poner únicamente las 2 primeras líneas como sección crítica.

Exclusión mutua. Ejemplo II

- Otro error menos visible pero muy típico se produce cuando:
  - dos procesos P1 y P2 utilizan una ***variable compartida X***, y ambos desean realizar alguna ***actualización de ésta*** **en una sola línea.**
    - Aunque a primera vista parece que no puede haber problema, si lo hay *debido a que internamente las instrucciones de alto nivel se subdividen en varias de bajo.*
  - Ej: *El "Problema de los jardines"*:
    - En este problema se supone que se desea controlar el número de visitantes a unos jardines. La entrada y la salida a los jardines se pueden realizar por dos puntos que disponen de puertas giratorias.
    - Se desea poder conocer en cualquier momento el número de visitantes a los jardines, por lo que se dispone de un ordenador con conexión en cada uno de los dos puntos de entrada que le informan cada vez que se produce una entrada o una salida.
  - **Asociamos el proceso P1 a un punto de entrada y el proceso P2 al otro punto de salida**.
  - **Ambos procesos se ejecutan de forma concurrente y utilizan una única variable x** para llevar la cuenta del número de visitantes que hay dentro del jardín.
  - Así, la **entrada** de un visitante por una de las puertas hace que se ejecute la instrucción **x:=x+1** mientras que la **salida** de un visitante hace que se ejecute la instrucción **x:=x-1**.

Exclusión mutua. Ejemplo II

- El **problema** aquí sería el **acceso a la variable x** por parte de los 2 procesos que controlan las puertas de entrada y salida al jardín:
  - Bien porque se ejecuten en paralelo por parte de 2 procesadores a la vez
  - Bien porque esa instrucción se subdivida a nivel interno en "subinstrucciones" y se ejecute en el orden incorrecto, como vemos a continuación:
    - Aunque no nos demos cuenta, internamente, al ser ésta una instrucción de alto nivel, se suele dividir en varias instrucciones de bajo nivel, que podrían llegar a ejecutarse de forma incorrecta.

```
Process P1                                   Process P2
...                                          ...
X := X+1   ← Sección Crítica →               X:=X-1
...                                          ...
```

Si ambos ejecutan esa instrucción a la vez, podría ocurrir el escenario:

P1: load R_{P1},X
P2: load R_{P2},X
P1: Add R_{P1}, 1
P2: Add R_{P2}, 1
P1: Store R_{P1},X
P2: Store R_{P2},X

Recordatorio

Un lenguaje de alto nivel facilita la programación mediante la *abstracción* de operaciones máquinas en operaciones de alto nivel. Así, una operación

X<-X+1

se traduce en el siguiente conjunto de operaciones *atómicas*:

LOAD X,R
ADD R,1
STORE R,X

Exclusión mutua. Ejemplo II.
Solución

- El problema de los jardines se podría extrapolar a cualquier problema de concurrencia, ya que incluye el caso más simple: la actualización de una variable por parte de 2 o más procesos.
  - Por eso extrapolaremos la solución a cualquier problema de concurrencia

- Solución:
  - Poner la línea que accede y modifica la variable compartida por los dos procesos (X) como región crítica.
    - *Es decir, poner el x<-x+1 y el x<-x-1 dentro de una sección crítica.*

Exclusión mutua. Soluciones Generales

- Existen 3 tipos de soluciones para garantizar la exclusión mutua en un sistema concurrente:
  - Soluciones por hardware
    - Aunque estas soluciones no están al alcance del programador normalmente, es interesante conocerlas. Hay 2:
      - ***Inhabilitación de Interrupciones***
      - *Instrucciones máquina especiales*
  - Soluciones por software
    - En el código se escriben instrucciones que nos permitan el paso controlado a las secciones críticas.
    - Se pueden implementar en:
      - El sistema operativo:
        - Se implementa un algoritmo que la garantice por parte del SSOO
      - Los procesos:
        - Se implementa un algoritmo que la garantice por parte del programador
    - Hay 4 principales:
      - **Semáforos, secciones críticas condicionales, monitores y paso de mensajes**

Exclusión mutua. Soluciones HW

- **Inhabilitación de interrupciones:**
  - Por defecto, un proceso continuará ejecutándose hasta que solicite un recurso al SO o hasta que sea interrumpido.
  - Para garantizar la EM es suficiente con impedir que un proceso sea interrumpido cuando quiera acceder a un recurso compartido.
    - Se impide que llegue otro y lo use cuando el primero se encuentra en la "mitad".
  - Problemas:
    - Se limita la capacidad del procesador para intercalar programas.
    - En sistemas Multiprocesador, Inhabilitar las interrupciones de un procesador no garantiza la Exclusión Mutua.

      Si lo hiciera en todos, y a la vez, si, pero ralentizaría mucho el sistema.

Exclusión mutua. Soluciones HW

- **Instrucciones especiales de máquina:**
  - Consiste en crear unas **instrucciones nuevas que agrupen las sentencias de acceso a memoria y registros**, y de forma que no se puedan segmentar (o ejecutar por separado.)
    - Por ejemplo acceso a memoria y a registro simultáneamente.
  - Notas:
    - Se realizan en un único ciclo de instrucción, lo cual es bueno, pero impiden la segmentación o pipeling, ya que no se podrían realizar instrucciones relacionadas de forma simultánea.
    - No están sujetas a injerencias por parte de otras instrucciones.
  - Ventajas:
    - Es aplicable a cualquier nº de procesos en sistemas con memoria compartida, tanto de monoprocesador como de multiprocesador.
      - Nota: Todos los sistemas multiprocesador actuales tienen un mecanismo de HW interno que impide a 2 procesos de 2 procesadores diferentes acceder de forma simultánea a la misma dirección de memoria, por lo que no habría problemas adicionales en sistemas multiprocesador.
    - Es simple y fácil de verificar.
  - Desventajas:
    - La espera activa consume tiempo del procesador (al ser HW).
    - Puede producirse inanición cuando un proceso abandona la sección crítica y hay más de un proceso esperando.

Soluciones SW

- Nos valen para resolver problemas de exclusión mutua y sincronización.
- Los principales mecanismos para resolver los problemas de sincronización a nivel SW son:
  - Para procesos que se ejecutan en *sistemas de memoria compartida:*
    - Semáforos (Dijkstra, 1965)
    - Secciones críticas condicionales (Brinch Hansen, 1972)
    - Monitores (Hoare, 1974)
  - Para procesos que se ejecutan en *sistemas distribuidos*:
    - Paso de mensajes

Soluciones SW

- Estos mecanismos buscan que **los procesos/hilos se sincronicen** de forma que lleven a cabo la tarea de forma adecuada y sin conflictos.
- Los lenguajes de programación que admitan concurrencia *(llamados Leng Prog concurrente)* deberán proveer de primitivas adecuadas para la especificación/implementación de los mismos.
  - Es decir deberán proveer funciones que permitan al programador comunicar/sincronizar procesos/hilos entre si.

- Entre ellos tenemos:
  - Java, q proporciona:
    - las primitivas: synchronized, wait y notify / notifyAll
    - la clase Semaphore
  - C/C++:
    - Wait, signal y la librería semaphore.h
  - C#:
    - Lock (equivalente a synchronized) y las clases Monitor y Semaphore.
    - Nota: No tiene wait() y notify() como en Java xq xa eso están las clases anteriores.
    - Por ej, semaphore tiene wait() y release()

Semáforos

- Se definieron originalmente por Dijkstra en 1965.
  - Año en el que él mismo formuló y resolvió el problema de los filósofos.
- Son componentes pasivos que sirven para arbitrar el acceso a un recurso compartido.

- Por su bajo nivel de abstracción resultan muy peligrosos de manejar y frecuentemente son causa de muchos errores.
- Usan un sistema de señales que hacen que los procesos paren y continúen su ejecución en base a ellas:
  - Para recibir una señal del semáforo s, los procesos ejecutan la primitiva **wait(s) / down(s)**.
  - Para transmitir una señal por el semáforo s, los procesos ejecutan la primitiva(operación atómica) **signal(s) / up(s)**.

Semáforos

- Un semáforo es un TAD(tipo abstracto de dato)/Objeto, y por lo tanto tendrá un:
  - Conjunto de valores que puede tomar (atributos).
    - Un semáforo tendrá 2 atributos:
      - un número entero
      - una lista de procesos, en la que se incluyen todos los procesos que se encuentra suspendidos a la espera de acceder al mismo
  - Conjunto de operaciones que admite.
    - Se permiten tres operaciones sobre un semáforo:
      - Inicializar
      - Espera (wait)
      - Señal (signal)
- Esto es común para todas las variaciones o implementaciones de semáforos existentes.
  - Otra cosa ya es como se manejen → De ahí las múltiples variantes

Semáforos

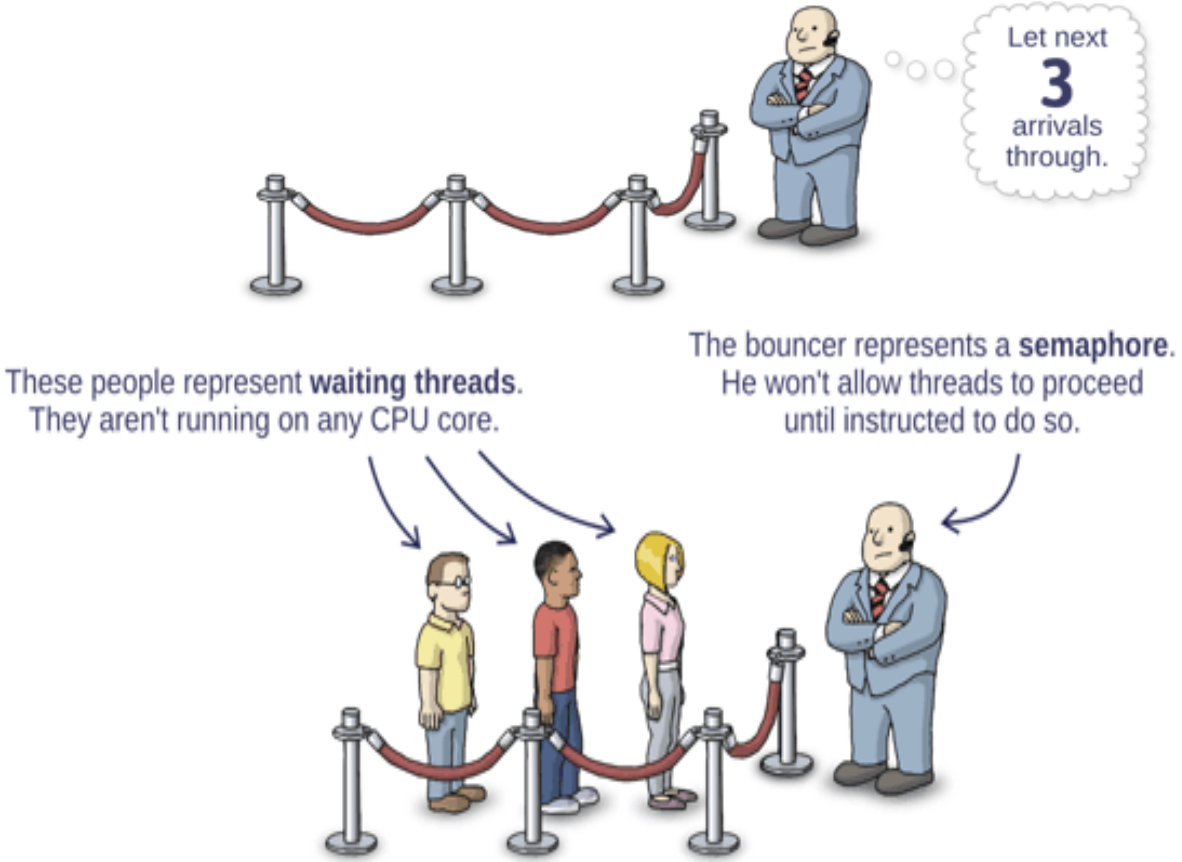
- Implementación
  - La estructura de los semáforos es fija, pero no su implementación.
  - Se pueden implementar de muchas maneras (variantes):
    - Por ejemplo, hay semáforos que bloquean al llegar a 0 y otros en -1.
    - Hay semáforos que restan al llegar un wait aunque estén en cero→permitiendo valores negativos
      Otros no: simplemente sacan de la cola.
    - Etc. (aunque estas son las más habituales)
  - Pero por regla general:
    - Un semáforo que ha tomado el valor 0 representa un semáforo cerrado, y
    - si toma un valor >0 representa un semáforo abierto.
  - Hay muchos lenguajes que los implementan como clases/Tipos para que los empleemos de forma nativa en nuestro programa.
  - En caso contrario deberemos ser nosotros los que los implementemos.
    - Para ello deberemos usar las primitivas proporcionadas por el lenguaje en cuestión.
    - Y programarlo nosotros como si de una clase / Tipo más de nuestro programa se tratase.

Semáforos

- Hay 2 tipos de semáforos:
  - General
    - El semáforo puede tomar cualquier número positivo.
    - Y respecto a los negativos:
      - Si se reduce (es decir, admite negativos): lo llamaremos normal
      - Si no se reduce a partir de cero(no acepta negativos) → lo llamaremos normal-binario
  - Binario
    - Si la variable entera solo puede tomar valores 0 y 1, el semáforo se denomina binario.
    - Asi, para un semáforo binario:, si
      - S = 1, entonces el recurso está disponible y la tarea lo puede utilizar
      - S = 0 el recurso no está disponible y el proceso debe esperar
- Los sistemas suelen ofrecer como componente primitivo semáforos generales ya que un semáforo General se puede utilizar como un semáforo binario.

Semáforos

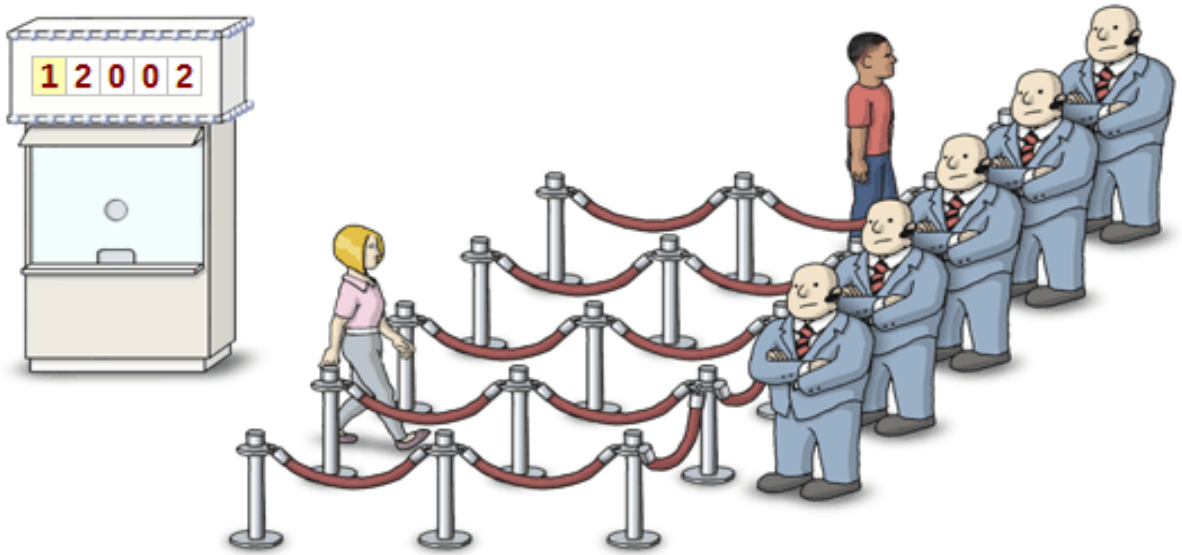
- Como una imagen vale + q mil palabras:
- Sol con semáforo General (N=3):



- El portero dejará pasar un máximo de 3 personas.

Semáforos

- Solución Binaria: El portero solo deja pasar a un@.
  - El resto esperan en la cola.
- Solución General: El portero deja pasar un máximo de N personas.
  - Cada vez que entra 1 resta 1 al contador del semáforo (=de huecos disponibles)
  - Si ese contador llega a 0, cualquiera que llegue se encola.
- Solución N-aria.
  - En muchas ocasiones se usan varios semáforos, ya que hay que proteger múltiples recursos, o un recurso con muchos/diferentes accesos.
  - Ejemplo: (Los porteros utilizan los números que ven en la cabina para dejar pasar o no)



En este caso el portero 1,2 y 5 dejarán pasar, pero el 3 y 4 no.

Semáforos

- En todos los tipos de semáforos **existe una cola** a la cual se añaden los procesos que están en espera del recurso.
- El orden en que se retiran los procesos de la cola define dos categorías más de semáforos:
  - a) Semáforos robustos:
    - funcionan como una cola FIFO, el que ha estado bloqueado más tiempo es el que sale de la cola.
  - b) Semáforos débiles:
    - no se especifica el orden en el que se retiran los semáforos.
- Los semáforos robustos cumplen todas las condiciones de la exclusión mutua, mientras que los semáforos débiles no previenen la inanición.

Semáforos

Ejemplos de Implementación: *(normal / binario / normal-binario)*

Semáforos:

```
struct semaforo{
   int contador;
   tipoCola cola;
}
void wait(semaforo s){
   s.contador--;
   if(s.contador<0){
      poner el proceso en s.cola;
      bloquear este proceso;
   }
}

void signal(semaforo s){
   s.contador++;
   if(s.contador<=0){
      quitar el proceso de s.cola;
      pasar el proceso a la cola listos;
   }
}
```

Semáforos binarios:

```
struct semaforoB{
   enum(cero,uno)valor;
   tipoCola cola;
}
void wait(semaforoB s){
   if(s.valor==1)
      s.valor=0;
   else{
      poner este proceso en s.cola;
      bloquear este proceso;
   }
}
void signalB(semaforoB s){
   if(s.cola.esvacia())
      s.valor=1;
   else{
      quitar el proceso de s.cola;
      pasar el proceso de s.cola a la cola de listos;
   }
```

NORMAL-BINARIO:

Sin este método wait lo codificáramos:

SI s.contador==0 ENT{
poner el proceso en s.cola
bloquear el proceso
}SI-NO{
s.contador–;
}FIN-SI

Idem con signal:

SI NO s.cola.esVacia()) ENT{
quitar el proceso de s.cola
y ponerlo en la de listos
}SI-NO{
s.contador++;
}FIN-SI

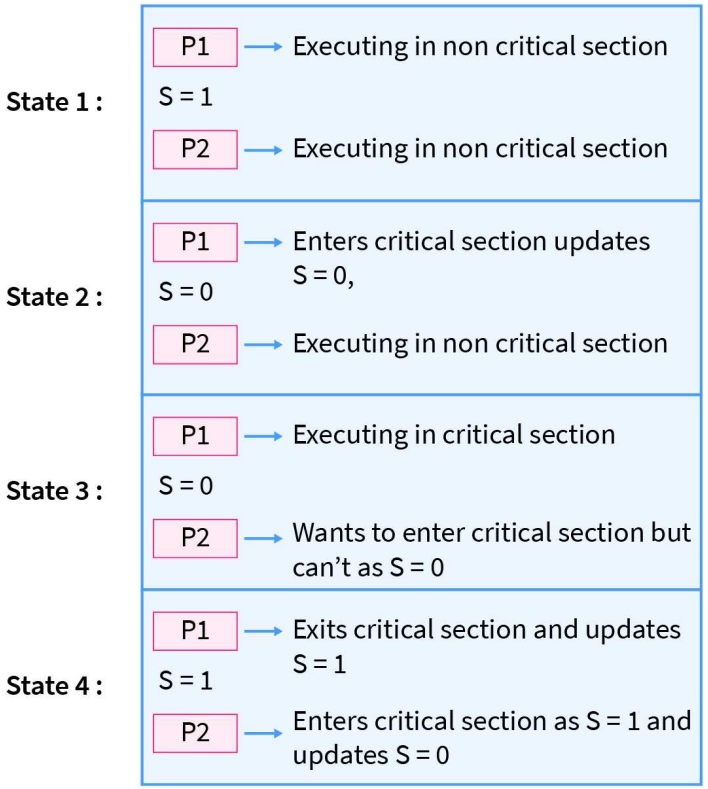
Semáforos. Ejemplo Semáforo binario

```
/* Ej de como usar un semáforo binario q garantiza la Exclusión Mutua */
Semáforo s=1; /*Inicio con el semáforo activo*/
void main(){
    parbegin(P(1), P(2), ..., P(n)); //para que lance a la par todos los procesos
                    /*n= nº de procesos q quieren ejecutar */
}
void P(int i){  //Un proceso cualquiera…
    while(cierto){
        wait(s); /*entrada a la sección crítica*/
            <<Se ejecuta la sección crítica, aquella q accede/modifica la var compartida >>
        signal(s); /*salida de la sección crítica*/

        /*resto del código no crítico*/
    }
}
```

Semáforos. Ejemplo semáforo binario

- Sean dos procesos P1 y P2 y un semáforo "s" inicializado a 1.
- Supongamos que P1 ingresa en su sección crítica, entonces el valor del semáforo s se convierte en 0.
- Si ahora P2 deseara ingresar a su sección crítica, deberá esperar hasta que s > 0.
  - Y esto solo puede suceder cuando P1 termina su sección crítica y hace un signal (s=1), lo que libera a P2 q puede pasar a ejecución.



Semáforos. Ej Semáforo no binario

En este ejemplo vemos como usar un semáforo que se inicia en N=3, de forma que deja pasar hasta 3 procesos/hilos a la sección crítica

```
item[3] buffer; // initially empty
semaphore empty; // initialized to +3
semaphore full; // initialized to 0
binary_semaphore mutex; // initialized to 1
```

```
void producer()
{
      …
      while (true) {
          item = produce();
p1:       wait(empty);
  /       wait(mutex);
p2:       append(item);
  \       signal(mutex);
p3:       signal(full);
      }
}
```

```
void consumer()
{
      …
      while (true) {
c1:       wait(full);
  /       wait(mutex);
c2:       item = take();
  \       signal(mutex);
c3:       signal(empty);
          consume(item);
      }
}
```

Semáforos. Conclusiones

- Como vemos, un semáforo se puede considerar como una variable sobre la que se pueden ejecutar las siguientes operaciones:
  - initial(s,Valor_inicial):
    - Asigna al semáforo s el valor inicial que se pasa como argumento.
    - Siempre se debe inicializar con un valor no negativo que indicará el número máximo de procesos que pueden entrar en la sección crítica.
  - wait(s):
    - Se ejecutará antes de entrar en la sección crítica.
    - El nombre de la operación wait es equívoco. En contra de su significado semántico natural, su ejecución a veces provoca una suspensión pero en otros caso no implica ninguna suspensión.
  - signal(s):
    - Se ejecutará cuando salgamos de la sección crítica.

Semáforos. Conclusiones

- Los semáforos son (ventajas)
  - estructuras muy simples y fáciles de comprender,
  - que pueden implementarse de forma muy eficiente, y
  - que permiten resolver todos los problemas que se presentan en programación concurrente.
- Sin embargo, siempre han sido muy criticados:
  - Dejan demasiados aspectos a decidir por parte del programador.
- Peligros que introducen los semáforos son (desventajas):
  - Un procedimiento wait y signal pueden olvidarse accidentalmente y ello conduce a una mal función catastrófica en el programa. En general un olvido de una sentencia wait conduce a un error de seguridad, como que no se respete una región de exclusión mutua, así mismo, un olvido de una sentencia signal, conduce a un bloqueo.
  - Pueden incluirse todas las sentencias wait y signal necesarias, pero en puntos no correctos del programa o en orden no adecuado.
  - Los semáforos dan lugar a un código donde las operaciones wait y signal relativas a un mismo semáforo, están muy dispersas por el código del programa, lo que constituye una grave dificultad para comprender el programa y para mantenerlo.

Monitores

- Los **semáforos** tienen algunas características que pueden generar **inconvenientes**:
  - Las variables compartidas son globales a todos los procesos
  - Las acciones que acceden y modifican dichas variables están dispersas por todo el programa
    - Para poder decir algo del estado de las variables compartidas, es necesario mirar todo el código
  - La adición de un nuevo proceso puede requerir verificar que el uso de las variables compartidas es el adecuado
  - *Todo esto redunda en la poca escalabilidad de los programas (que usan semáforos como método de sincronización).*
- **Los monitores se crean con el objetivo de solventar estos problemas.**

Monitores

- **Un monitor es un conjunto de procedimientos/métodos** que proporciona el acceso con exclusión mutua a un recurso o conjunto de recursos compartidos por un grupo de procesos.
- **Los procedimientos van encapsulados dentro de un módulo** que tiene la propiedad especial de que sólo un proceso puede estar activo cada vez para ejecutar un procedimiento del monitor.
- **El monitor se puede ver como una valla alrededor del recurso** (o recursos), de modo que los procesos que quieran utilizarlo deben entrar dentro de la valla, pero en la forma que impone el monitor, es decir, que solo uno puede estar a la vez dentro del recinto.

Monitores

- Características básicas:
  - Las variables de datos locales están solo accesibles para el monitor.
  - Un proceso entra en el monitor invocando a uno de sus procedimientos.
  - Solo un proceso se puede estar ejecutando en el monitor en un instante dado.

- Sincronización
  - La sincronización se consigue mediante variables de condición accesibles solo desde el interior del monitor.
  - Los monitores NO proporcionan por si mismos un mecanismo para la sincronización de tareas, sólo para la exclusión mutua.
    - Por ello su construcción debe completarse permitiendo, por ejemplo, que se puedan usar señales para sincronizar los procesos.

Monitores

- Las variables que dejan pasar o no a la sección crítica (mecanismo de sincronización) en un monitor se conocen como **variables de condición**.
  - A cada causa diferente por la que un proceso deba esperar se asocia una variable de condición.
  - Sobre ellas sólo se puede actuar con **dos primitivas:**

    espera y señal (wait / signal en C o wait / notify en Java).
- Cuando un proceso ejecuta una **operación de espera** se suspende y se coloca en una ***cola asociada a dicha variable de condición***.
  - La diferencia con el semáforo radica en que ahora la ejecución de esta operación el monitor siempre suspende el proceso que la emite.
  - La suspensión del proceso hace que se libere la posesión del monitor, lo que permite que entre otro proceso.
- Cuando un proceso ejecuta una **operación de señal** se libera un proceso suspendido en la ***cola de la variable de condición utilizada***.
  - Si no hay ningún proceso suspendido en la cola de la variable de condición invocada, la operación señal no tiene ningún efecto.

Monitores. Implementación

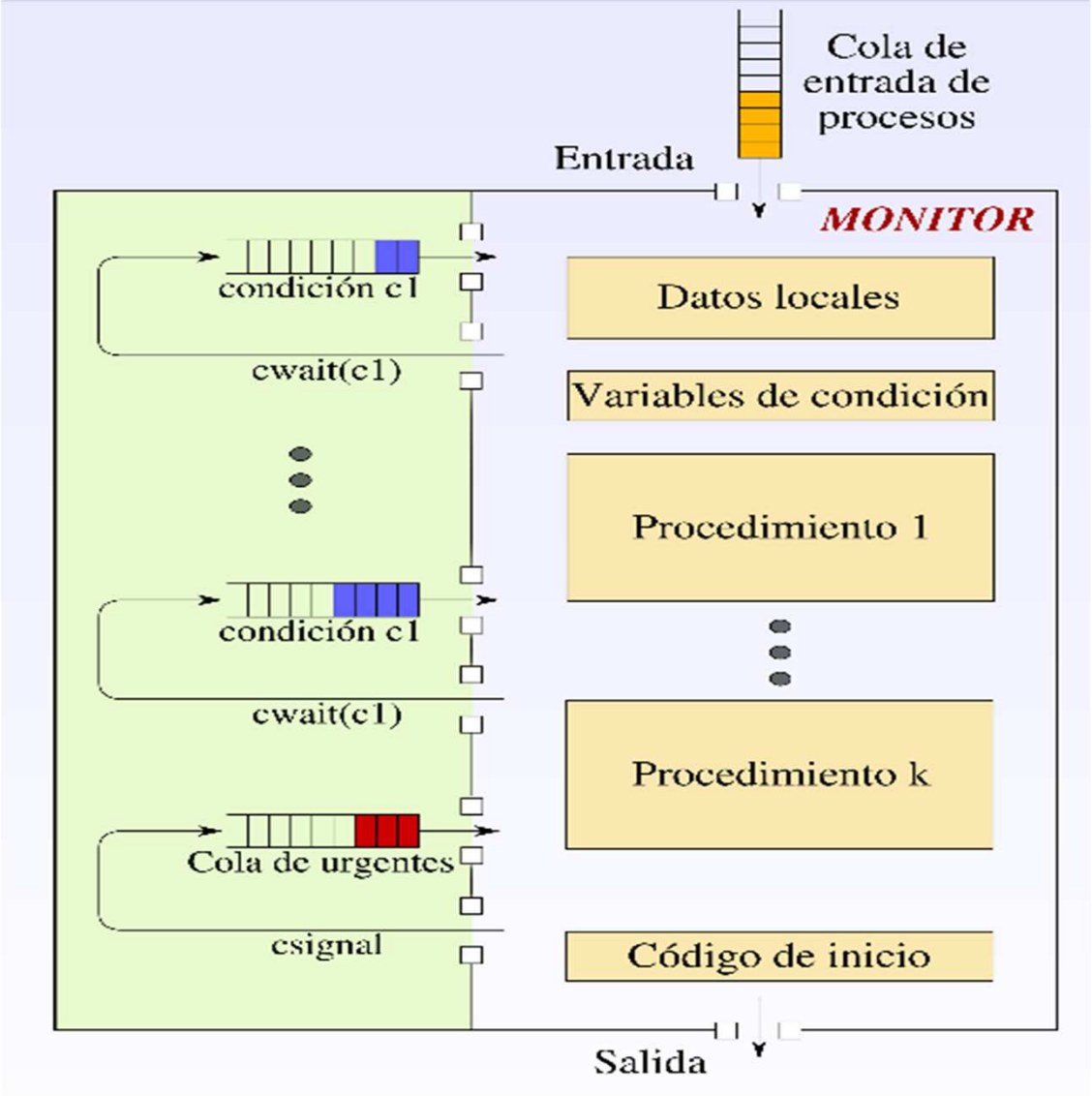
- Sintaxis general (Pseudocódigo):

```
monitor nombre-del-monitor {

    // declaración de variables compartidas
    condition c;

    //Métodos
    procedure P1 (…) { ….}
    …
    procedure Pn (…) {……}

    //Código de inicalización
    Initialization code ( ….) { … }
    …
}
```



Monitores. Conclusión

- La ventaja para gestionar la exclusión mutua que presenta un monitor (frente a los semáforos u otro mecanismo), es que *está implícita:*

la única acción que debe realizar el programador del código que usa un recurso es invocar una entrada del monitor.

- Ejemplos y mas detalle en:

Monitores.pdf

Paso de mensajes. Definición

- Los mensajes proporcionan una solución al problema de la concurrencia de procesos que integra la sincronización y la comunicación entre ellos.
- Resulta adecuado tanto para sistemas centralizados como distribuidos.
  - Esto hace que se incluyan en prácticamente todos los sistemas operativos modernos.

Paso de mensajes. Definición

- La comunicación mediante mensajes necesita siempre de un proceso emisor y de uno receptor así como de información que intercambiarse.
  - Por ello, las operaciones básicas para comunicación mediante mensajes que proporciona todo sistema operativo son: **enviar (*mensaje*) y *recibir* (*mensaje*)**.
  - *Ambas son primitivas del sistema*
- Se utilizan como:
  - Refuerzo de la exclusión mutua, para sincronizar los procesos.
  - Medio de intercambio de información, tanto en SSOO como en comunicaciones de red.

Paso de mensajes. Identificación

- **Comunicación simétrica**
  - Los procesos tanto receptor como emisor necesitan nombrar al otro para comunicarse
- **Comunicación asimétrica**
  - Sólo el emisor nombra al destinatario.

- Nota: Existe otra forma donde nadie nombre a nadie: **Los tablones (o bulletin boards)**

Paso de mensajes. Identificación

- **Comunicación directa**
  - Cada proceso que desea comunicarse debe nombrar explícitamente el destinatario o el remitente de la comunicación
  - enviar(P, mensaje) → *Envía un mensaje al proceso P*
  - recibir(Q, mensaje)→ *Recibe un mensaje del proceso Q*

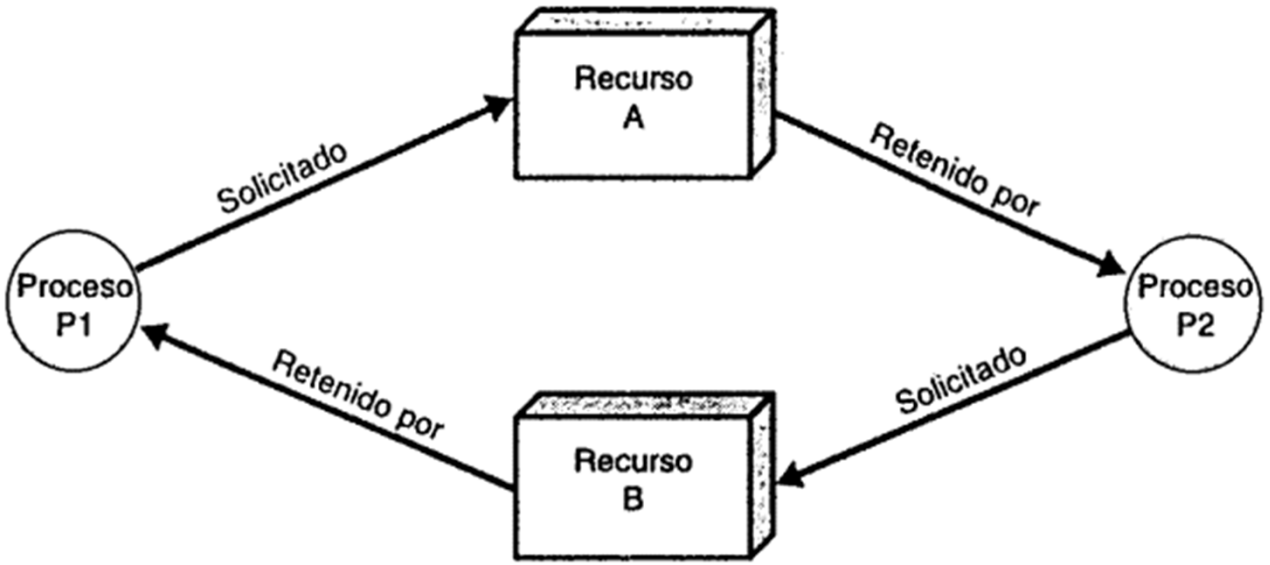
- **Comunicación indirecta**
  - Con la comunicación indirecta, los mensajes se envían a, y se reciben de, buzones (también llamados puertos)
  - enviar(A, mensaje) → *Enviar un mensaje al buzón A*
  - recibir(A, mensaje) → *Recibir un mensaje del buzón A*

Paso de mensajes. Sincronización

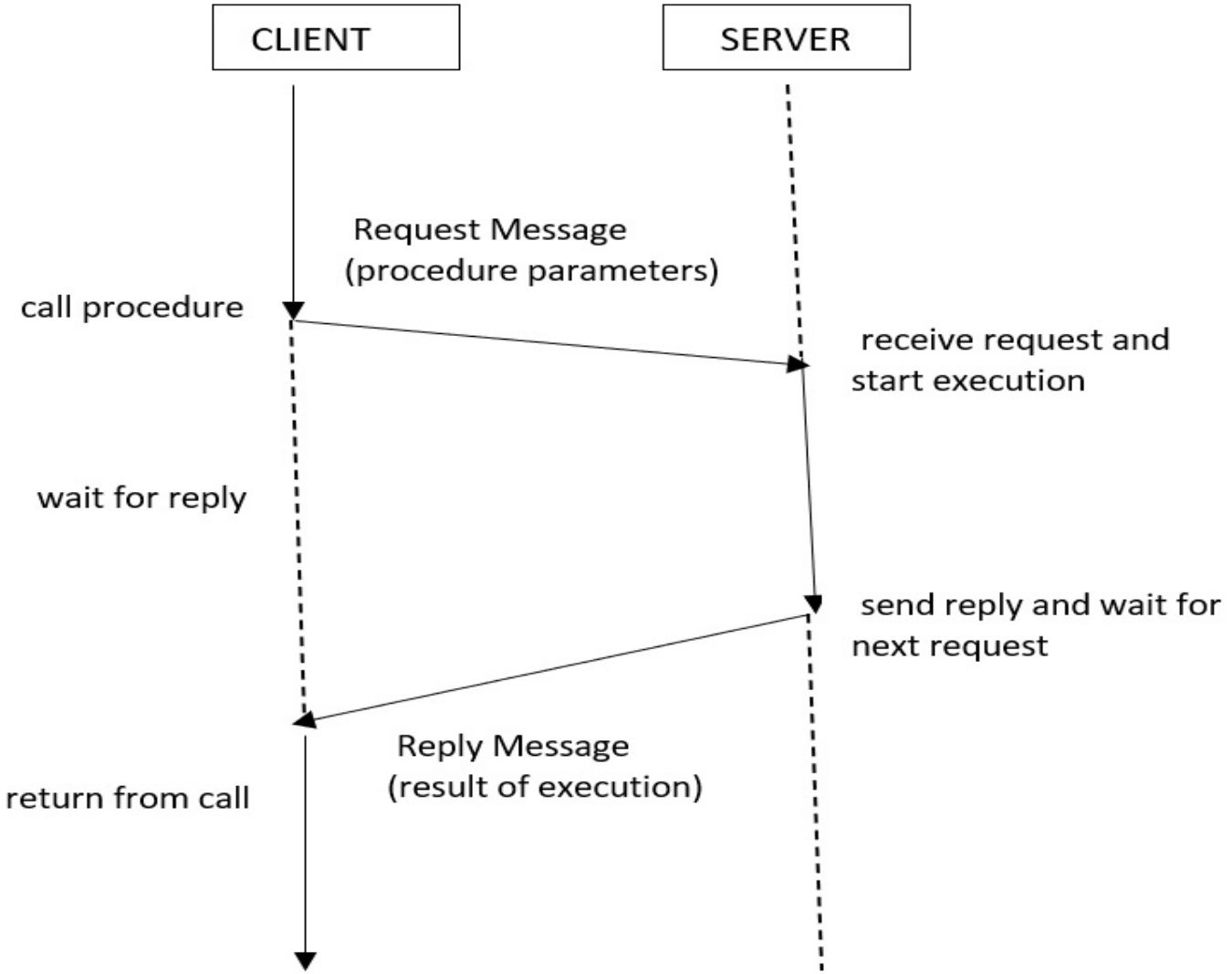
- El emisor y el receptor pueden ser **bloqueantes o no bloqueantes**
  - Esperando a que se lea un mensaje o a que se escriba un nuevo mensaje.
- Hay varias combinaciones posibles:
  - Envío bloqueante, recepción bloqueante:
    - Tanto el emisor como el receptor se bloquean hasta que se entrega el mensaje.
    - Esta técnica se conoce como rendezvous.
  - Envío no bloqueante, recepción bloqueante:
    - Permite que un proceso envíe uno o mas mensajes a varios destinos tan rápido como sea posible.
    - El receptor se bloquea hasta que llega el mensaje solicitado.
  - Envío no bloqueante, recepción no bloqueante:
    - Nadie debe esperar.

Paso de mensajes. Sincronización

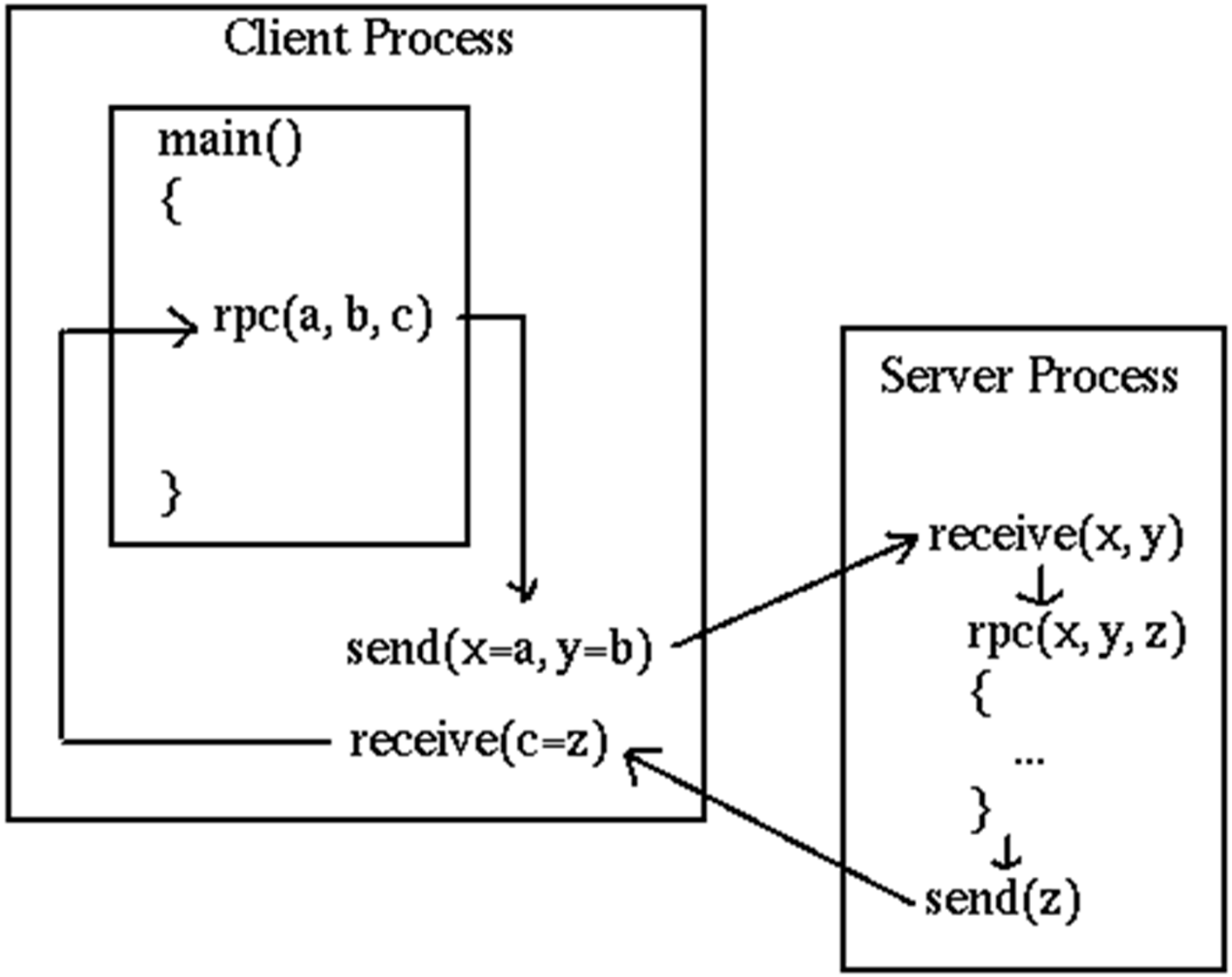
- Interbloqueo:



Paso de mensajes. Ejemplos

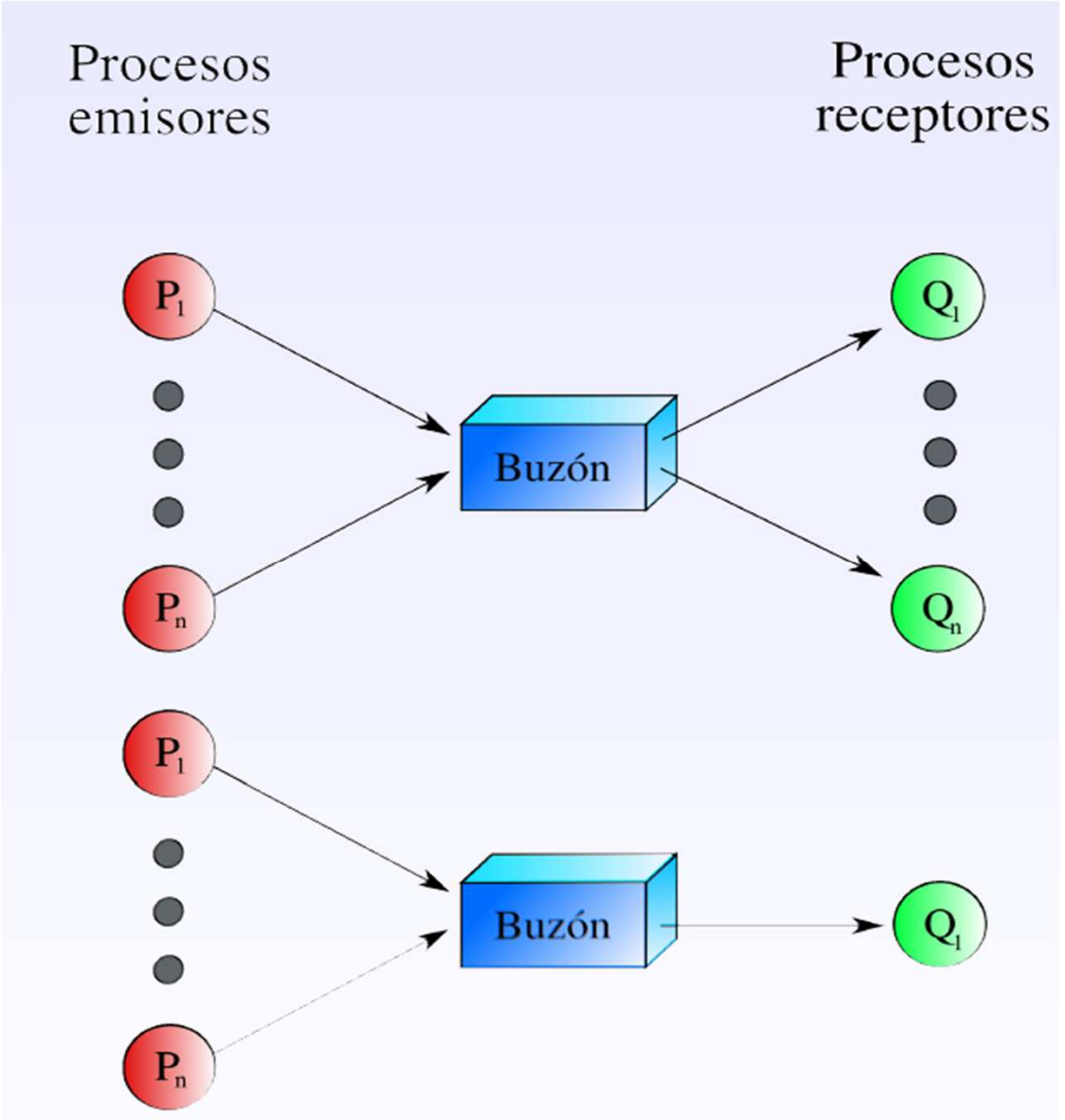


Paso de mensajes. Ejemplos



Paso de mensajes. Canales y Mensajes

- 2 mecanismos de comunicación:
- **Directo**:
  - Se mandan mensajes directamente unos a otros.
  - Es mas lenta ya que requiere siempre de espera y sincronización entre los 2 procesos implicados en la comunicación.
- **Indirecto**:
  - Los mensajes se envían a una estructura de datos compartida formada por **colas**.
  - Estas colas se denominan **buzones** (mailboxes).
  - Un proceso envía mensajes al buzón apropiado y el otro los coge del buzón.
  - En general es mejor que el anterior, ya que evita problemas de sincronización, pudiendo los procesos trabajar a "su" velocidad.



Paso de mensajes. Canales y Mensajes

- Flujo de Datos.
  - Una vez establecido un canal de comunicación entre emisor y receptor, éste, de acuerdo al flujo de datos que pasan por él puede ser de dos tipos: unidireccional o bidireccional.
  - Para el primero, la información fluye siempre en un sentido entre los dos interlocutores mientras que para el segundo la información fluye en ambos sentidos.
- Capacidad del Canal.
  - Es la posibilidad que tiene el enlace de comunicación de almacenar los mensajes enviados por el emisor cuando éstos no son recogidos de forma inmediata por el receptor.
  - Se tienen en general canales de capacidad
    - cero (en donde no existe un buffer donde se vayan almacenando los mensajes),
    - finita (en donde el buffer existente tiene un tamaño fijo)
    - infinita (donde el buffer asociado al enlace de comunicación se supone infinito en su capacidad).

Paso de mensajes. Canales y Mensajes

- Tamaño de los mensajes.
  - Los mensajes que viajan por el canal pueden ser de longitud fija o de longitud variable.
- Canales con tipo o sin tipo.
  - Algunos esquemas de comunicación exigen definir el tipo de datos que va a fluir por el canal, imponiendo la restricción del envío de datos únicamente del tipo para el que fue declarado el canal.
- Paso por copia o por referencia.
  - El paso de mensajes exige enviar información entre los procesos implicados en la comunicación. Esto puede realizarse de dos maneras:
  - A) efectuar una copia exacta de los datos (mensaje) que el emisor quiere enviar desde su espacio de direcciones al espacio de direcciones del proceso receptor (paso por copia o valor) o
  - B) simplemente enviarle al receptor la dirección en el espacio de direcciones del emisor donde se encuentra el mensaje (paso por referencia).
    Nota: Esto último requiere que los procesos interlocutores compartan memoria.

Paso de mensajes. Condiciones de Error

- Pérdida de mensajes entre los procesos que se comunican mediante canales
- Alteración de mensajes como consecuencia por ejemplo de ruidos en la transmisión del mensaje por el canal
- Bloqueo de procesos (emisor y/o receptor)
- El estudio a fondo de soluciones a los errores de comunicación entre procesos concurrentes y cooperantes por medio de paso de mensajes en sistemas distribuidos o de red cae dentro del ámbito de los llamados Sistemas Tolerantes a Fallos.