Sistemas con memoria Compartida

Dr. Mario Rossainz López

Programación Concurrente y Paralela

Otoño 2022

NRC: 60898

Exclusión Mutua

- Consiste en asegurar que dos procesos concurrentes no accedan a un mismo recurso común al mismo tiempo, ya que ello puede llevar a incoherencias en la información que se procesa.

Exclusión Mutua

- ***Problema de los jardines:*** 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 2 puertas. Se desea poder conocer en cualquier momento el número de visitantes a los jardines, por lo que se dispone de un dispositivo de cálculo con conexión en cada uno de los dos puntos de entrada que le informan cada vez que se produce una entrada o una salida.

Exclusión Mutua

(variable común) X = Num. de visitantes en los jardines

P1

P2

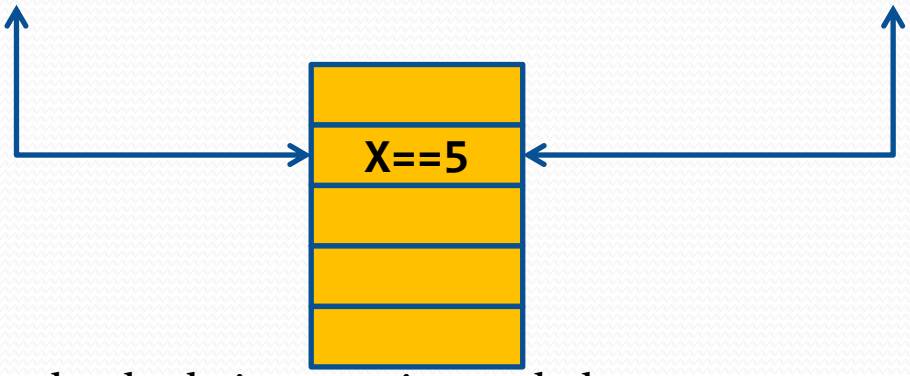
```
x= x+1;  // Un visitante entra a los jardines
x= x-1;  // Un visitante sale de los jardines
```

```
x= x+1;  // Un visitante entra a los jardines
x= x-1;  // Un visitante sale de los jardines
```

Exclusión Mutua

```
P1(x): {
     Si (persona entra)
     entonces x=x+1;
     Si (persona sale)
     entonces x=x-1;
    }
```

```
P2(x): {
     Si (persona entra)
     entonces x=x+1;
     Si (persona sale)
     entonces x=x-1;
    }
```



Situaciones de entrelazdo de instrucciones de los procesos que puede producir errores cuando se ejecutan

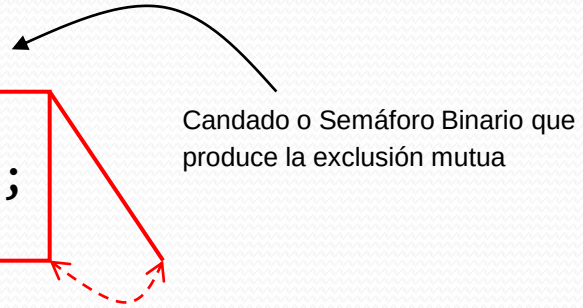
Exclusión Mutua

- **Región Crítica:** Es la zona de código de un proceso concurrente donde se utiliza un recurso compartido.
- La exclusión mutua se consigue impidiendo que dos o más procesos concurrentes ejecuten a la vez su región crítica.
- Una región crítica solo puede ser ejecutada por un único proceso concurrente siguiendo una única secuencia de interfoliación (entrelazado).
- Si algún otro proceso desea ejecutar su región crítica, deberá esperar a que aquel que actualmente ejecuta la suya termine y lo indique.
- Se necesita de algún mecanismo (*protocolos software*) que indique a los procesos cuando uno de ellos entra o sale de la región crítica

Exclusión Mutua

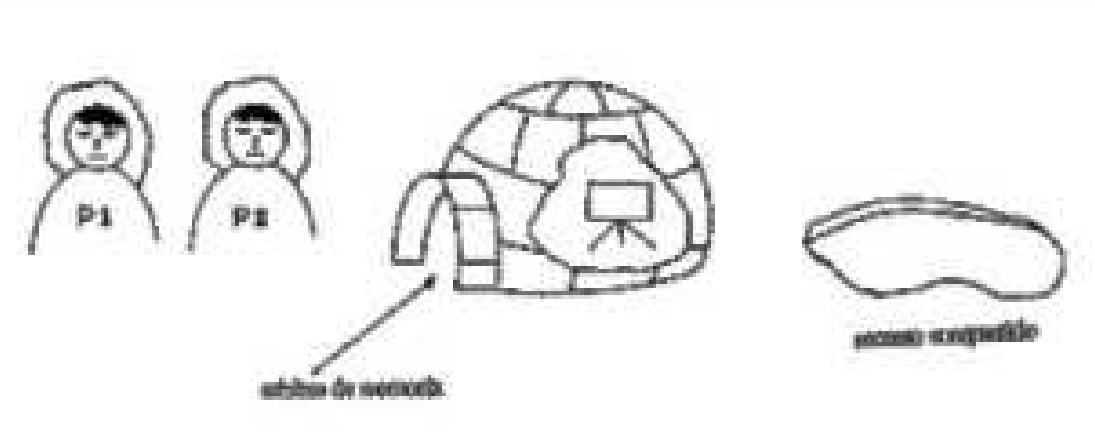
CANDADO o SEMÁFORO BINARIO: Esto se consigue con dos protocolos de software, uno de entrada y otro de salida que todo proceso debe forzosamente ejecutar antes y después de ejecutar su región crítica, y que constituyen un mecanismo de seguridad (***bloqueo***).

```
Task body P is
begin
   loop
        resto_de_código;
        pre_Protocolo;
              Región Crítica;
        post-Protocolo;
   end loop;
end P.
```



El problema de los Esquimales

ALTERNANCIA: Imaginemos 2 esquimales y un agujero para pescar como recurso compartido. Existe un iglú con un pizarrón. Solo uno de los esquimales puede acceder a la vez al pizarrón a la vez. Cuando uno de los esquimales quiere acceder al agujero para pescar debe consultar si tiene permiso para hacerlo en el pizarrón, si en el pizarrón se indica que el permiso lo tiene el otro esquimal espera un tiempo y lo vuelve a probar de nuevo más tarde. Si en el pizarrón se indica que tiene permiso, irá a pescar. Una vez que termine con el agujero, irá al pizarrón y cederá el permiso al otro esquimal.



El problema de los Esquimales

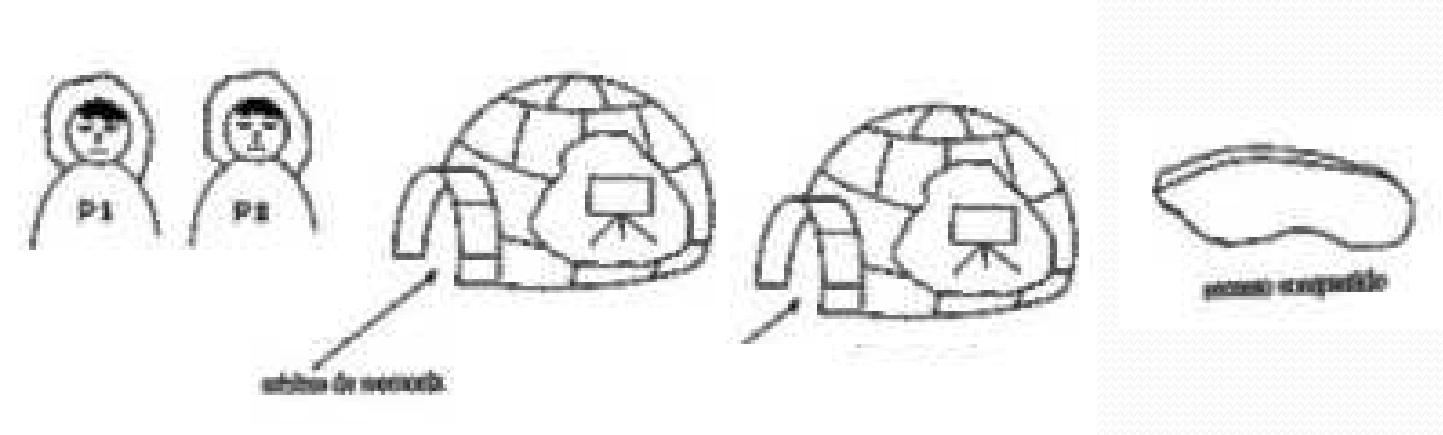
```
PROGRAM PrimerIntento;
    VAR turno [1..2];

   PROCEDURE P1;
      BEGIN
        REPEAT
           WHILE turno = 2 DO ;     (* pasea *)
           ******  usa agujero de pesca  ******
           turno := 2;
           otras cosas;
        FOREVER
      END;

   PROCEDURE P2;
      BEGIN
        REPEAT
           WHILE turno = 1 DO ; (* pasea *)
           ******  usa agujero de pesca  ******
           turno := 1;
           otras cosas;
        FOREVER
      END;

    BEGIN
      turno := 1
      COBEGUIN
            P1; P2
      COEND
 END.
```

Primer Intento: Alternancia

Inconveniente: Alternancia Estricta obligada

El problema de los Esquimales

FALTA DE EXCLUSIÓN: Haremos uso de dos iglúes con su correspondiente pizarrón. Los pizarrones tienen dos indicadores, "*pescando*" o "*NoPescando*", cada uno de los pizarrones corresponden a cada uno de los esquimales. Si queremos hacer uso del agujero para pescar deberemos consultar en el pizarrón del otro esquimal, si indica que está "*pescando*" saldremos y esperaremos para volverlo a intentar más tarde. Si en el pizarrón pone "*NoPescando*" iremos a nuestro pizarrón y pondremos "*pescando*". Cuando terminemos de usar el agujero de pescar iremos a nuestro pizarrón y pondremos "*NoPescando*".



El problema de los Esquimales

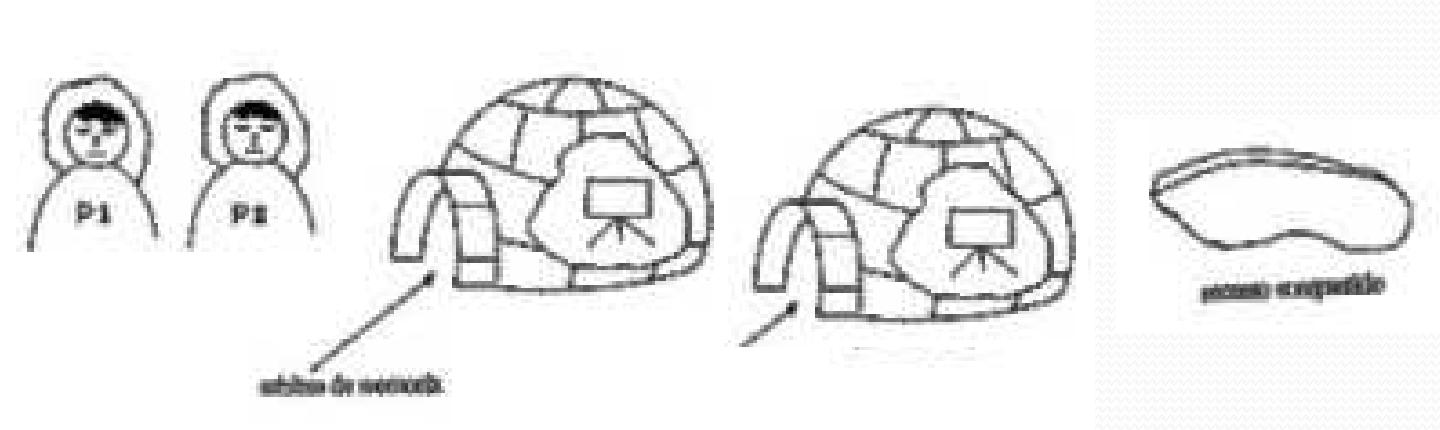
```
PROGRAM SegundoIntento;
  VAR pizarra1, pizarra2: (pescando NOpescando);

  PROCEDURE P1;
    BEGIN
      REPEAT
        WHILE pizarra2 = pescando DO;  (* pasea *)
        pizarra1 := pescando;
        *** usa agujero de pesca ***
        pizarra1 := NOpescando;
        otras cosas
      FOREVER
    END;

  PROCEDURE P2;
    BEGIN
      REPEAT
        WHILE pizarra1 = pescando DO;   (* pasea *)
        pizarra2 := pescando;
        *** usa agujero de pesca ***
        pizarra2 := NOpescando;
        otras cosas
      FOREVER
    END;

BEGIN
  pizarra1 := NOpescando;
  pizarra2 := NOpescando;
  COBEGIN
     P1; P2
  COEND
END.
```

Segundo Intento: Falta de Exclusión

La falta de exclusión mutua se genera si los dos procesos tienen libre el recurso, a la vez consultan el estado de los oponentes y al comprobar que está libre los dos indican estado de uso del recurso y lo usan.

El problema de los Esquimales

INTERBLOQUEO (Espera Infinita): Ahora actuaremos a la inversa para tratar de evitar la falta de exclusión mutua, cuando un proceso quiera utilizar el recurso, primero indica que quiere hacerlo y luego espera a que esté libre.



El problema de los Esquimales

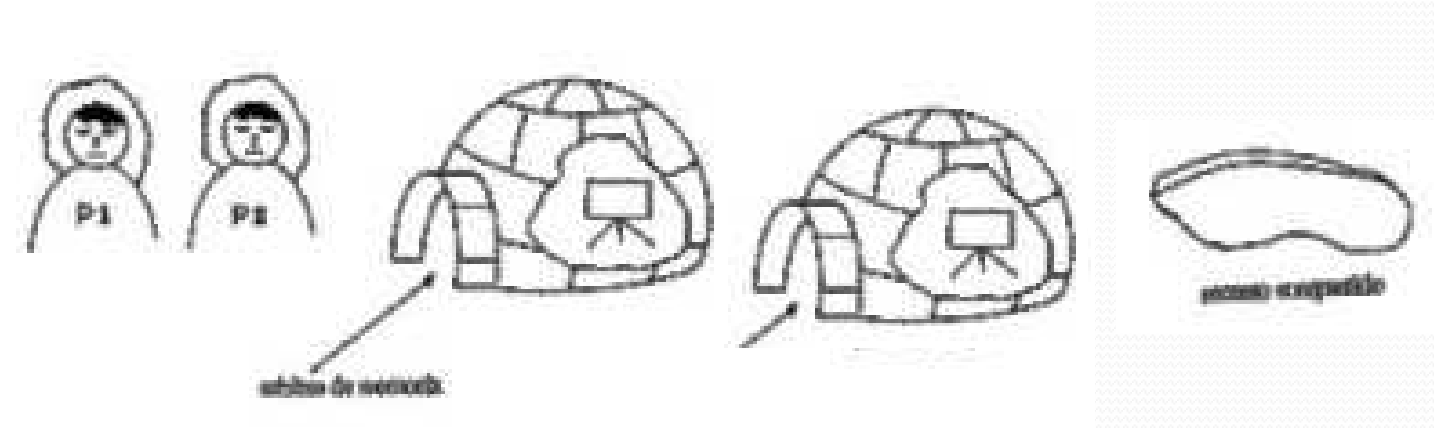
```
PROGRAM TercerIntento;
VAR pizarra1, pizarra2: (pescando NOpescando);

PROCEDURE P1;
  BEGIN
    REPEAT
      pizarra1 := pescando;
      WHILE pizarra2 = pescando DO;  (* pasea *)
      *** usa el agujero de pesca ***
      pizarra1 := NOpescando;
      otras cosas
     FOREVER
  END;

PROCEDURE P2;
  BEGIN
    REPEAT
      pizarra2 := pescando;
      WHILE pizarra1 = pescando DO;  (* pasea *)
      *** usa el agujero de pesca ***
      pizarra2 := NOpescando;
      otras cosas
    FOREVER
  END;

BEGIN
  pizarra1 := NOpescando;
  pizarra2 := NOpescando;
  COBEGIN
     P1; P2
  COEND
END.
```

Tercer Intento: Interbloqueo (espera infinita)

En el caso que los dos esquimales estén sin pescar (NOpescando) y los dos pretendan acceder al agujero al mismo instante, los dos activarán en su pizarrón el estado de intención de pescar (pescando), al ir a consultar la pizarra del oponente comprobarán que está pescando (o en intención de hacerlo), en este caso ninguno de los dos esquimales podrá acceder al recurso ni podrá cambiar su estado.

El problema de los Esquimales

ESPERA INFINITA: Para solucionar el problema anterior añadiremos el trato de cortesía, si un proceso ve que su oponente quiere hacer uso del recurso, se lo cede.



El problema de los Esquimales

```
PROGRAMA CuartoIntento;
  VAR pizarra1, pizarra2: (pescando, NOpescando);

  PROCEDURE P1;
    BEGIN
      REPEAT
        pizarra1 := pescando;
        WHILE pizarra2 = pescando DO
          BEGIN
            (* tratamiento de cortesía *)
            pizarra1 := NOpescando;
            (* date una vuelta *)
            pizarra1 := pescando
          END;

         *** usa el agujero de pesca ***
         pizarra1 := NOpescando;
         otras cosas
      FOREVER
     END;
```

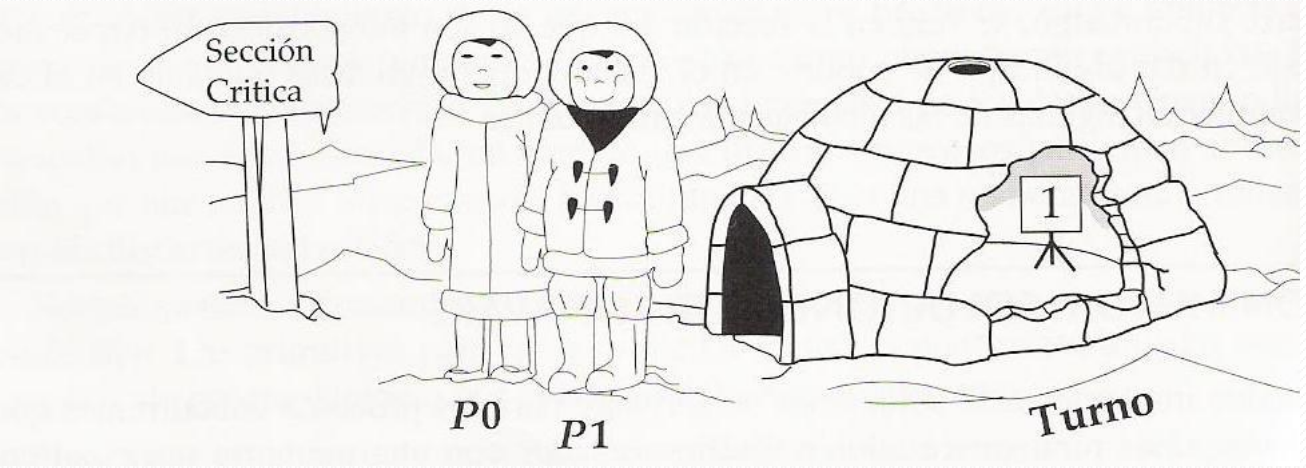
```
PROCEDURE P2;
     BEGIN
       REPEAT
         pizarra2 := pescando;
         WHILE pizarra1 = pescando DO
           BEGIN
             (* tratamiento de cortesía *)
             pizarra2 := NOpescando;
             (* date una vuelta *)
             pizarra2 := pescando
           END;
         *** usa el agujero de pesca ***
         pizarra2 := NOpescando;
         otras cosas
       FOREVER
     END;

BEGIN
  pizarra1 := NOpescando;
  pizarra2 := NOpescando;
  COBEGIN
    P1; P2
  COEND
END.
```

Cuarto Intento: Espera infinita

Este tratamiento de cortesía puede conducir a que los procesos se queden de manera indefinida cediéndose mutuamente el paso. Esta solución no asegura que se acceda al recurso en un tiempo finito.

El problema de los Esquimales

ALGORITMO DE DEKKER: La solución al problema de la exclusión mutua se atribuye al matemático holandés T. Dekker y fue presentada por Dijkstra en 1968. Se utiliza al igual que en otro algoritmo llamado Algoritmo de Peterson, una variable turno que sirve para establecer la prioridad relativa de los dos procesos, es decir, en caso de conflicto ésta variable servirá para saber, a quien se le concede el recurso y su actualización se realiza en la región crítica, lo que evita que pueda haber interferencias entre los procesos.



El problema de los Esquimales

```
PROGRAMA AlgoritmoDeDekker;
  VAR pizarra1, pizarra1: (pescando, NOpescando);
  Turno:[1..2];

  PROCEDURE P1;
    BEGIN
      REPEAT
        pizarra1 := pescando
        WHILE pizarra2 = pescando DO
          BEGIN
            IF turno = 2 THEN
               BEGIN
                 (* tratamiento de cortesía *)
                 pizarra1 := NOpescando;
                 WHILE turno = 2 DO; (* date una vuelta *)
                   pizarra1 := pescando
                END;
           END;
        *** usa el agujero de pesca ***
        turno := 2;
        pizarra1 := NOpescando;
        otras cosas
      FOREVER
    END;
```

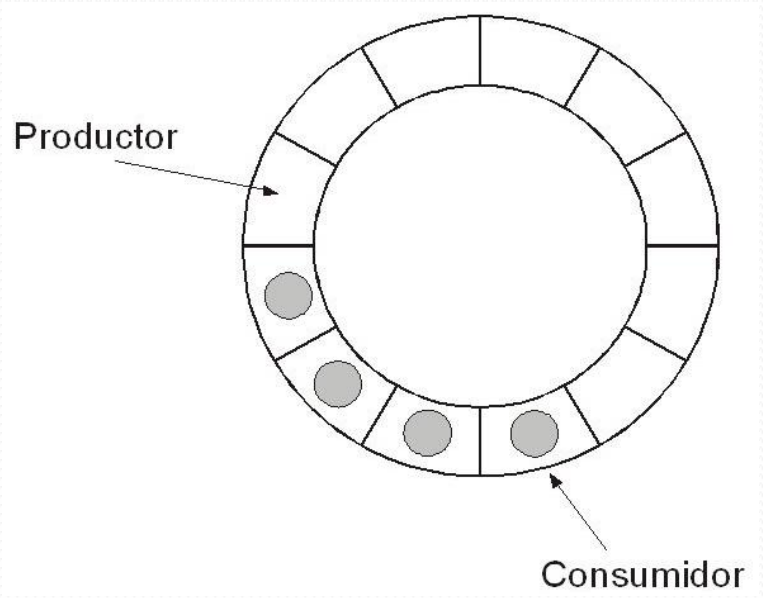
```
PROCEDURE P2;
    BEGIN
      REPEAT
        pizarra2 := pescando
        WHILE pizarra1 = pescando DO
          BEGIN
            IF turno = 1 THEN
              BEGIN
               (* tratamiento de cortesía *)
               pizarra2 := NOpescando;
               WHILE turno = 1 DO;(* date una vuelta *)
               Pizarra2 := pescando
              END;
           END;
        *** usa el agujero de pesca ***
        turno := 1;
        pizarra2 := NOpescando;
        otras cosas
      FOREVER
    END;

BEGIN
  pizarra1 := NOpescando;
  pizarra2 := NOpescando;
  turno := 1;
  COBEGIN
    P1; P2
  COEND
END.
```

Quinto Intento: Dekker

Este algoritmo asegura la exclusión mutua y está libre de interbloqueos.

Sincronización entre Procesos

- **El Problema del Buffer Circular:**
- Supongamos que tenemos dos procesos concurrentes ocupados uno de ellos (el proceso Productor) en llenar un buffer circular de datos y el otro (el proceso Consumidor) en vaciarlo.



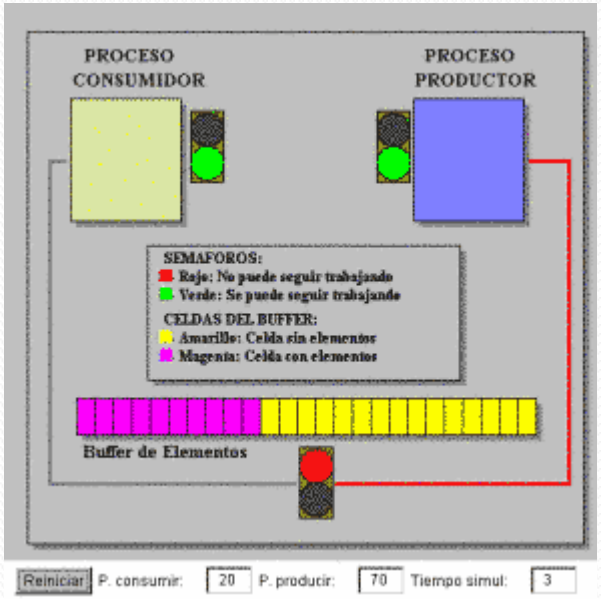
Sincronización entre Procesos

- El proceso Productor no puede poner más datos en el buffer si está lleno,
- el proceso Consumidor no puede tomar datos del buffer cuando está vacío
- Por tanto es necesario sincronizar la ejecución de ambos procesos en función del estado del buffer, consiguiendo:

1. Que el proceso productor se bloquee cuando el buffer esta lleno
2. Que el proceso consumidor se bloquee cuando el buffer esta vacío
3. que el productor desbloquee al consumidor (¿Cuándo?)
4. que el consumidor desbloquee al productor (¿Cuándo?)

Mecanismos Software para sincronizar y comunicar procesos

- **Candados o Semáforos Binarios** (lock): Constituye el método clásico para restringir o permitir el acceso a recursos compartidos en un entorno en el que se ejecutan varios procesos concurrentemente



Mecanismos Software para sincronizar y comunicar procesos

- **Variables de Condición:** Son variables comunes a los procesos concurrentes que comparten recursos, donde dependiendo del valor de la variable de condición se produce una sincronización del tipo *wait-notify*, *espera-notificación*, de manera que los procesos se comunican y sincronizan al acceder a sus recursos compartidos mediante mensajes que indican el bloqueo o desbloqueo de uno a otro u otros procesos vía la variable de condición.

Mecanismos Software para sincronizar y comunicar procesos

- **Variables de Condición:**

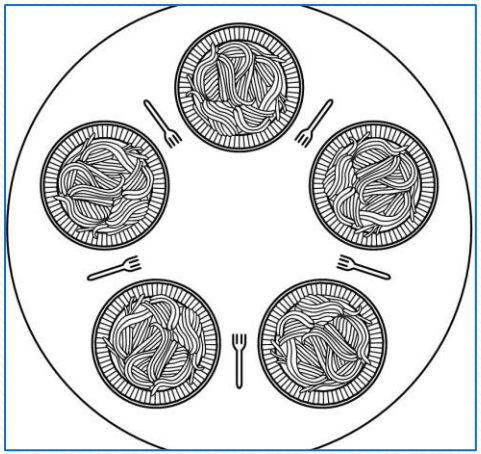
```
action()
   {
     . . .
     lock();
     while (x != 0)
          wait(s);
     unlock();
     take_action();
     . . .
    }
```

```
counter()
   {
     . . .
     lock();
     x--;
     if (x == 0)
          signal(s)
     unlock();
     . . .
    }
```

Mecanismos Software para sincronizar y comunicar procesos

- **Monitor:** Un monitor es una unidad de programación propuesta por dijsktra que encapsula datos compartidos, junto con los procedimientos u operaciones para acceder a ellos.
  - Dispone de variables de condición para la sincronización de los procesos que utilizan el monitor
  - Todas las operaciones de un monitor se ejecutan por definición en exclusión mutua

Ejemplos clásicos



Los filósofos comelones

Ejemplos clásicos

El barbero dormilón

Ejemplos clásicos

Los fumadores compulsivos

Correctitud de los Programas

- **PROPIEDADES:**

- **Seguridad (safety):** Decimos que un sistema o programa concurrente es seguro cuando dicho programa o sistema preserva la exclusión mutua. Esta propiedad se relaciona con el comportamiento estático del sistema concurrente.

- **Vivacidad (Liveness):** Un programa concurrente tiene la propiedad de vivacidad cuando éste previene los interbloqueos (deadlocks), que se producen cuando una serie de recursos protegidos por exclusión mutua están distribuidos de tal manera entre los procesos que, todo proceso posee un recurso y espera a otro que esta en posesión de un proceso distinto.

- **No Inanición (Not Lockout):** Es aquella propiedad que asegura que el conjunto de procesos que integran al sistema o programa concurrente, siempre accederán al procesador. En otras palabras, la inanición se da cuando hay un conjunto de procesos que por ciertas circunstancias nunca acceden al procesador. La inanición (lockout) es menos grave que un interbloqueo (deadlock), ya que al menos otros procesos siguen en ejecución.

- **Equidad (Fairness):** Un sistema sin interbloqueos y sin inaniciones es equitativo cuando todos los procesos disponen de una fracción proporcionada de tiempo de procesador y no hay procesos marginados para acceder a dicha fracción de tiempo.