# Capítulo 3
# Aproximación a soluciones concurrentes

Como se hablo anteriormente las reglas de sincronización necesarias entre procesos concurrentes son: Exclusión mutua y Condición de Sincronización.

**Tipos de sincronización:**

**Exclusión mutua:** Cuando los procesos desean utilizar un recurso no compartible, la sincronización necesaria entre ellos se denomina exclusión mutua. Un proceso que está accediendo a un recurso no compartible se dice que está en su sección crítica. Por lo tanto, la sección crítica es la parte del código que utiliza un proceso para acceder a un recurso no compartible, y por lo tanto, debe ejecutarse en exclusión mutua.

*Condiciones para resolver exclusión mutua:*
*Exclusión mutua:* no pueden acceder dos procesos a la vez a la sección crítica.
*Limitación en la espera:* ningún proceso debe esperar indefinidamente para ingresar a la sección crítica.
*Progreso en la ejecución:* Cuando un proceso quiera ejecutar su sección crítica, este pueda hacerlo si está libre la misma y avanzar en la ejecución de la sección crítica.
*Condiciones de sincronización:* La podemos definir como la propiedad de un proceso de no realizar un evento hasta que otro proceso haya llevado a cabo una acción determinada.

Mecanismos para implementar sincronización son:
- Inhibición de interrupciones
- Espera ocupada
- Semáforos
- Regiones críticas
- Regiones críticas condicionales
- Monitores
- Paso de mensajes

**Inhibición de interrupciones:** Puesto que la única instrucción que hace posible la interrupción de la ejecución de un proceso son las interrupciones, entonces deshabilitamos las mismas, es decir el proceso lo hace:

```
     process P1                              process Pi
...                                     ...
deshabilitar                            deshabilitar
interrupciones;                         interrupciones;
Sección Crítica                  ...    Sección Crítica
habilitar interrupciones;               habilitar interrupciones;
...                                     ...
```

No es correcto hacer esto, ya que delegamos el acceso a las interrupciones a procesos de usuario, si un proceso no vuelve a habilitar las interrupciones sería el fin del sistema.

**Espera ocupada:** Solución de software: El proceso queda en espera mediante la comprobación de una variable compartida.
Aquí suponemos que las instrucciones load y store son atómicas.

*Primer intento:* Una sola variable global compartida. No garantiza mutex, dos procesos pueden ingresar a la sección crítica a la vez. La v se inicia en sclibre.

```
     process P0                          process P1
repeat                              repeat
   /*protocolo de entrada*/            /*protocolo de entrada*/
   a) while v=scocupada do;            a) while v=scocupada do;
   b) v: =scocupada;                   b) v: =scocupada;
   /*ejecuta la sección crítica*/      /*ejecuta la sección crítica*/
   c) Sección Crítica0;                c) Sección Crítica1;
   /*protocolo de salida*/             /*protocolo de salida*/
   d) v: =sclibre;                     d) v: =sclibre;
   Resto0                              Resto1
forever                             forever
```

*Segundo intento:* Alternancia. Garantiza **mutex**, si un proceso quiere entrar dos veces seguidas no puede, y si Resto0 es más largo que Resto1, P1 se queda esperando, esto significa que no satisface **progreso en la ejecución**, turno puede ser iniciada en 0 ó 1.

```
     process P0                          process P1
repeat                              repeat
   while turno=1 do;                   while turno=0 do;
   Sección Crítica0;                   Sección Crítica1;
   turno: =1;                          turno: =0;
   Resto0                              Resto1
forever                             forever
```

*Tercer Intento:* Falta de exclusión mutua.

```
     process P0                          process P1
repeat                              repeat
   a) while C1=enSC do;                a) while C0=enSC do;
   b) C0: =enSC;                       b) C1: =enSC;
   c) Sección Crítica0;                c) Sección Crítica1;
   d) C0: =restoproceso;               d) C1: =restoproceso;
   Resto0                              Resto1
forever                             forever
```

No garantiza exclusión mutua. Observar que hay dos variables compartidas, cada proceso puede leer pero no escribir la del otro. Ambas variables c0 y c1 se inicializan con valor restoproceso.

*Cuarto Intento:* Espera infinita

```
     process P0                          process P1
repeat                              repeat
   C0: =quiereentrar;                  C1: =quiereentrar;
   while C1=quiereentrar do;           while C0=quiereentrar do;
   Sección Crítica0;                   Sección Crítica1;
   C0: =restoproceso;                  C1: =restoproceso;
   Resto0                              Resto11
forever                             forever
```

Garantiza mutex, pero si ambos llegan al mismo tiempo al while quedan esperando infinitamente.

**Algoritmo de Peterson**:

```
     process P0                          process P1
repeat                              repeat
   C0: =quiereentrar;                  C1: =quiereentrar;
   turno: =1;                          turno: =0;
   while (C1=quiereentrar) and         while (C0=quiereentrar) and
(turno=1) do;                       (turno=0) do;
   Sección Crítica0;                   Sección Crítica1;
   C0: =restoproceso;                  C1: =restoproceso;
   Resto0                              Resto1
forever                             forever
```

Variables compartidas: c0, c1 y turno.
Garantiza mutex (usa turno para la simultaneidad).
Garantiza progreso en la ejecución (si un Proceso quiere entrar, lo logra si el otro no quiere entrar).
Garantiza espera limitada (cuando sale de la crítica asigna Ci a restoproceso permitiendo entrar al otro).
Sirve para dos procesos

**Algoritmo de Lamport** *(no es objetvo de la materia demostrar el mismo):*

El siguiente algoritmo soluciona el problema de la exclusión mutua para n procesos, es conocido también como el *algoritmo de la panadería*, debido a su algoritmo es utilizado corrientemente en panaderías, heladería, etc.. Al entrar al la tienda, cada cliente recibe un número. El cliente con el número más bajo es el primero en ser atendido. Para describir el algoritmo usamos la siguiente notación:

- < es una relación de orden sobre pares de enteros definida como sigue:
  [(a,b) < (c,d)] si [a<c o (a=c y b<d)]

- max(a1, ….., an) obtiene el máximo de sus argumentos

- C: array[0..n-1] of {cognum, nocognum}. Con la variable C[i]=cognum se indica que el proceso Pi está tomando un número y con C[i]=nocognum que ha terminado de tomarlo. Inicialmente este array está inicializado en nocognum.

- numero: array[0..n-1] of integer. Cada elemento de este array indica el número tomado por cada proceso. Inicialmente este array se inicializa a 0.

```
     process Pi
repeat
   C[i]: =cognum;
   numero[i]: =1+max(numero[0],...,numero[n-1]);
   C[i]: =nocognum;
   for j: =0 to n-1 do
      begin
         while (C[j]=cognum) do;
         while ((numero[j]≠0) and ((numero[i],i)>(numero[j],j))) do;
      end;
   Sección Crítica_i;
   numero[i]: =0;
   Resto_i
forever
```

Se verifica la **exclusión mutua** porque cuando un proceso Pi está en su sección crítica, éste ha pasado por el segundo while al poseer el número menor o el meno identificador entre los de menor número, y si otro proceso intenta pasar a su sección crítica tendría que cumplir lo mismo, lo que no es posible debido a que el número que extraiga será siempre mayor que el que posee el proceso que está dentro de la sección crítica. Se garantiza el **progreso de ejecución** según el orden en que se toma el número; además la entrada a la sección crítica es según el número tomado asegurando **limitación de espera**.

**Espera ocupada**: Solución de hardware (como por ejemplo instrucciones especiales de procesadores) no lo veremos.