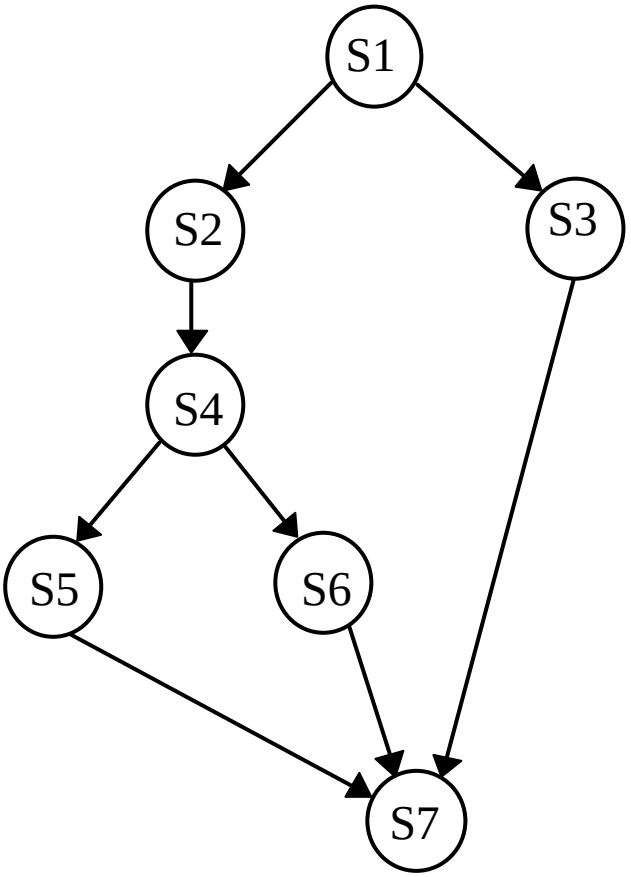
GRAFOS DE PRECEDENCIA

Grafo acíclico orientado cuyos nodos corresponden a sentencias individuales.

Un arco de un nodo Si al nodo Sj significa que la sentencia Sj puede ejecutarse sólo cuando ha acabado Si.

Ejemplo:



ESPECIFICACIÓN DE LA CONCURRENCIA

Los grafos no se pueden usar en programación.

Necesitamos otras herramientas:

FORK / JOIN

FORK L

Genera dos ejecuciones concurrentes en un programa:

1. Una se inicia en la instrucción siguiente a FORK
2. Otra empieza en la instrucción etiquetada L

JOIN

Permite recombinar varias ejecuciones paralelas en una sola. La rama que ejecuta primero la instrucción JOIN termina su ejecución.

Para saber el número de ramas que se deben reunir se usa un parámetro con JOIN (una variable entera no negativa que se inicializa con el número de ejecuciones paralelas a reunir).

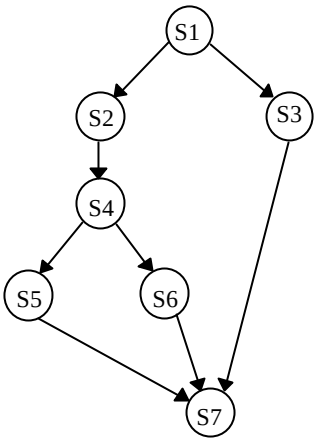
La ejecución de una instrucción JOIN CONT tiene el siguiente efecto:

```
CONT := CONT -1;
IF CONT ≠ 0 THEN <TERMINAR RAMA>;
```

La instrucción JOIN tiene que ejecutarse **indivisiblemente** es decir, la ejecución concurrente de dos instrucciones JOIN es equivalente a la ejecución secuencial en un orden indeterminado.

EJEMPLO:

Implementar, usando FORK/JOIN, el grafo de precedencia de la figura.

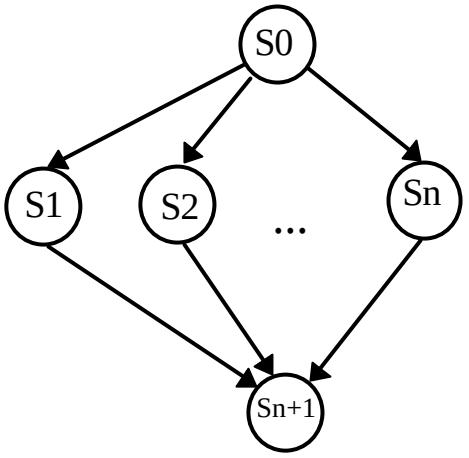



```
        S1;
        CONT := 3;
        FORK L1;
        S2;
        S4;
        FORK L2;
        S5;
        GOTO L3;
L2:     S6;
        GOTO L3;
        L1: S3;
L3:     JOIN CONT;
        S7;
```

COBEGIN / COEND

Es de mayor nivel que la pareja FORK/JOIN y tiene la forma siguiente:

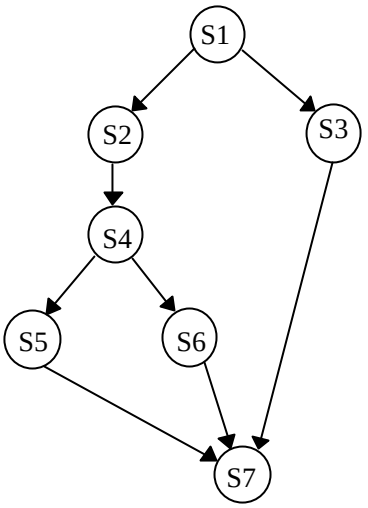
```
COBEGIN
    S1;
    S2;
    ...
    Sn;
COEND;
```



Todas las instrucciones insertadas entre las palabras clave COBEGIN y COEND se ejecutarán concurrentemente.

EJEMPLO:

Implementar, usando COBEGIN/COEND, el grafo de precedencia de la figura adjunta.




```
S1;
COBEGIN
        S3;
        BEGIN
                S2;
                S4;
                COBEGIN
                        S5;
                        S6;
                COEND
        END
COEND;
S7;
```

Implementación de COBEGIN/COEND con FORK/JOIN

```
COBEGIN
    S1;
    S2;
    ....
    Sn;
COEND;
```

```
        CONT := n;
        FORK L2;
        FORK L3;
        ...
        FORK Ln;
        S1;
        GOTO L1;
L2:     S2;
        GOTO L1;
L3:     S3;
        GOTO L1;
        ...
Ln:     Sn;
L1: JOIN CONT;
```

EJERCICIO:

Dada la expresión (A + B) * (C + D) - (E/F)

Establecer el grafo correspondiente que extraiga el máximo grado de paralelismo e implementar dicho grafo utilizando:

A) La pareja COBEGIN/COEND

B) La construcción FORK/JOIN

LIBERAR (SECCIÓN_CRÍTICA);

REQUISITOS DE LAS SOLUCIONES

Una solución al problema de la sección crítica debe cumplir los tres requisitos siguientes:

1. **Exclusión mutua.**

   Si un proceso Pi se está ejecutando en su sección crítica, entonces ningún otro proceso se puede estar ejecutando en la suya.

2. **Progresión.**

   Ningún proceso suspendido fuera de su sección crítica debe impedir progresar a otros procesos.

3. **Espera limitada.**

   Si un proceso ha solicitado entrar en su SC, debe haber un límite al número de veces que otros procesos entren en sus respectivas SC, antes de que el primero lo consiga.

Se supone que cada proceso se ejecuta a una velocidad no nula, aunque no se puede asegurar nada sobre las velocidades relativas de los procesos.

Al presentar los algoritmos se definen sólo las variables utilizadas para sincronización.

Cada proceso tendrá la siguiente estructura:

```
Pi: repeat
        ...
    CÓDIGO DE ENTRADA

        SECCIÓN CRÍTICA

    CÓDIGO DE SALIDA

        SECCIÓN RESIDUAL

    until false;
```

Lo que cambiará de una solución a otra es la forma de llevar a cabo los recuadros marcados.

SOLUCIONES PARA DOS PROCESOS

Algoritmo 1

```
var turno: 0..1;
```

```
Pi: repeat

    while turno ≠ i do no-op;

        SECCIÓN CRÍTICA

    turno := j;

        SECCIÓN RESIDUAL

    until false;
```

- 👍 Se garantiza la exclusión mutua
- 👎 No se garantiza la progresión
- 👎 Hay espera improductiva

Algoritmo 2

```
var indicador: array [0..1] of boolean;
```

```
Pi: repeat

        indicador[i] := true;
        while indicador[j] do no-op;

                SECCIÓN CRÍTICA

        indicador[i] := false;

                SECCIÓN RESIDUAL

    until false;
```

- Se garantiza la exclusión mutua
- No se garantiza la progresión
- Hay espera improductiva

Algoritmo 2 (variante)

```
var indicador: array [0..1] of boolean;
```

```
Pi: repeat

      while indicador[j] do no-op;
      indicador[i] := true;

            SECCIÓN CRÍTICA

      indicador[i] := false;

            SECCIÓN RESIDUAL

    until false;
```

- No se garantiza la exclusión mutua
- Se garantiza la progresión
- Hay espera improductiva

Algoritmo 3 (Peterson)

```
var  indicador: array [0..1] of boolean;
     turno: 0..1;
```

```
Pi: repeat

        indicador[i] := true;
        turno := j;
        while (indicador[j] and turno=j) do no-op;

                SECCIÓN CRÍTICA

        indicador[i] = false;

                SECCIÓN RESIDUAL
    until false;
```

- Se garantiza exclusión mutua (ver)
- Se garantiza la progresión
- Se garantiza la espera limitada
- Hay espera improductiva

SOPORTE HARDWARE PARA SINCRONIZACIÓN

La solución más simple es INHABILITAR/HABILITAR interrupciones.

Inconvenientes:

- Peligroso en manos del usuario.
- No sirve si se tienen dos o más CPUs.

Existen otras soluciones (software) que requieren algo de ayuda del hardware. Instrucciones especiales:

- Test-and-Set
- Swap

Test-and-Set (Evaluar-y-asignar)

Puede definirse (de forma algorítmica) como una función:

```
function TAS (var objetivo: boolean): boolean;
    begin
        TAS := objetivo;
        objetivo := true;
    end;
```

La característica esencial es que la instrucción se ejecuta , es decir, como una unidad ininterrumpible (en un sólo ciclo de memoria).

Si se ejecutan dos TAS simultáneamente lo harán siguiendo algún orden arbitrario.

Solución con Test-and-Set

```
var cerradura: boolean (:= false);
```

```
Pi: repeat

      while TAS(cerradura) do no-op;

            SECCIÓN CRÍTICA

      cerradura := false;

            SECCIÓN RESIDUAL

    until false;
```

- 👍 Se garantiza la exclusión mutua
- 👍 Se garantiza la progresión
- 👎 No se garantiza la espera limitada
- 👎 Hay espera improductiva

Solución con Swap

```
var cerradura:  boolean (:= false);
```

```
Pi: var clave: boolean;
    repeat
        llave := true;
        repeat
            SWAP(cerradura, llave);
        until llave = false;

            SECCIÓN CRÍTICA

        cerradura := false;

            SECCIÓN RESIDUAL
    until false;
```

- 👍 Se garantiza la exclusión mutua
- 👍 Se garantiza la progresión
- 👎 No se garantiza la espera limitada
- 👎 Hay espera improductiva

Algoritmo de Burns (para n procesos)

```
var  esperando: array[0..n-1] of boolean (:= false);
     cerradura:  boolean (:= false);
```

```
Pi: var  j: 0..n-1;
         llave: boolean;
    repeat
        esperando[i] := true;
        llave := true;
        while (esperando[i] and llave) do
              llave := TAS(cerradura);
        esperando[i] := false;
              SECCIÓN CRÍTICA
        j := i +1 mod n;
        while (j≠i) and (not esperando[j]) do
              j := j+1 mod n;
        if j=i then cerradura := false
              else esperando[j] := false;
              SECCIÓN RESIDUAL
    until false;
```

SEMÁFOROS

Un semáforo es una variable entera protegida que, aparte de la **inicialización**, sólo puede ser accedida por medio de dos operaciones **indivisibles** estándar:

- P(s) ≡ WAIT(s) ≡ ESPERA(s)
- V(s) ≡ SIGNAL(s) ≡ SEÑAL(s)

Las definiciones clásicas de estas operaciones son:

- WAIT(s):

```
while s ≤ 0 do no-operación;
s := s - 1;
```

- SIGNAL(s):

```
s := s + 1;
```

USO DE SEMÁFOROS PARA EXCLUSIÓN MUTUA

```
var mutex: semáforo;
```

```
Pi: repeat
        wait(mutex);
            SECCIÓN CRÍTICA
        signal(mutex);
            SECCIÓN RESIDUAL
    until false;
```

Esta solución es aplicable a **n** procesos.

Se asocia un semáforo a cada recurso de acceso exclusivo. El valor inicial del semáforo se establece a 1 de modo que sólo un proceso pueda ejecutar la operación wait con éxito.

La inicialización es responsabilidad del proceso padre.

REVISIÓN DE LA DEFINICIÓN DE SEMÁFORO

Problema de la espera activa:
Cuando un proceso ejecuta la operación wait sobre un semáforo con valor no positivo, tiene que hacer una espera que es improductiva.

Para evitarlo, el proceso debería **bloquearse** a sí mismo. El bloqueo sitúa al proceso en estado de espera y transfiere el control al Sistema Operativo, que puede seleccionar a otro proceso de la cola de preparados.

Un proceso bloqueado en un semáforo es reactivado por otro proceso que ejecute la operación signal sobre el mismo semáforo.

El proceso “despierta” por una operación que cambie el estado del proceso bloqueado y lo ponga en la cola de preparado para ejecución.

Por ello se redefine el concepto de semáforo como un entero más una lista de procesos.

```
type semáforo =   record
                      valor: integer;
                      L: lista de proceso;
                  end;
```

Las operaciones wait y signal se redefinen como:

```
wait(S):  S.valor := S.valor - 1;
          if S.valor < 0 then
              begin
                  <añadir proceso a S.L>;
                  bloquear;
              end;
```

```
signal(S): S.valor := S.valor + 1;
           if S.valor ≤ 1 then
               begin
                   <sacar proceso P de S.L>;
                   despertar(P);
               end;
```

Con esta definición, el valor del semáforo puede ser negativo. En tal caso, su magnitud equivale al número de procesos que están esperando en el semáforo.

El aspecto clave es que las operaciones sobre un semáforo se ejecuten de forma **indivisible**.

Esta es una nueva situación de sección crítica que puede resolverse usando cualquiera de las soluciones vistas.

No se ha eliminado completamente el problema de la espera activa.

Lo que se consigue es:

- Eliminarla del programa de aplicación.
- Limitarla a las S.C. de wait y signal, que son bastante cortas.

Así, la espera improductiva no se produce casi nunca y, cuando se produce, ocurre durante un periodo muy breve.

USO DE SEMÁFOROS PARA SINCRONIZACIÓN

Sean P_1 y P_2 dos procesos que se están ejecutando concurrentemente y supongamos que P_1 incluye una instrucción (S1) que sólo puede ser ejecutada después de que se haya ejecutado una instrucción (S2) de P_2.

Se usa un semáforo compartido inicializado a cero (0).

```
var S: semáforo (S.valor := 0);
```

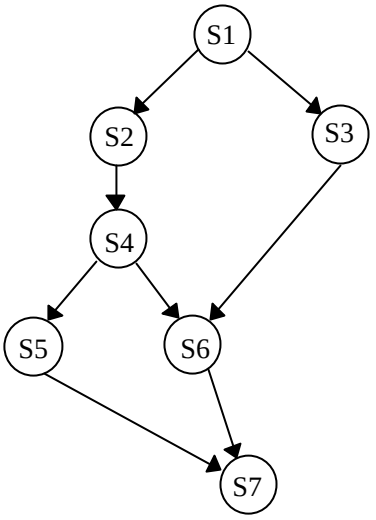
```
P1: repeat                    P2: repeat
        ...                           ...
        wait(S);                      S2;
        S1;                           signal(S);
        ...                           ...
    until false;                  until false;
```

- Si P_1 ejecuta wait antes que P_2 haga signal ⇨ P_1 espera a P_2
- Si P_2 ejecuta signal antes de que P_1 haga wait ⇨ P_1 hace wait sin bloquearse

Usando semáforos para sincronización en combinación de COBEGIN/COEND, puede resolverse cualquier grafo de precedencia.

EJEMPLO:

El grafo de precedencia adjunto no tiene un programa correspondiente usando sólo la instrucción concurrente.



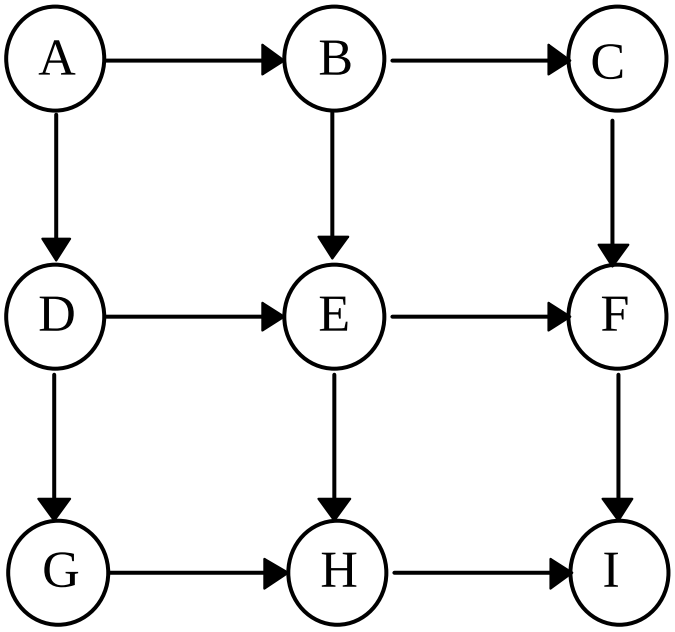

```
var A, B, C, D, E, F, G: semáforo (:= 0, 0, 0, 0, 0, 0, 0);
```

```
cobegin
    begin S1; signal(A); signal(B); end;
    begin wait(A); S2; S4; signal(C); signal(D); end;
    begin wait(B); S3; signal(E); end;
    begin wait(C); S5; signal(F); end;
    begin wait(D); wait (E); S6; signal(G); end;
    begin wait(F); wait (G); S7; end;
coend
```

EJERCICIOS

1. Resolver el problema anterior usando el menor número posible de semáforos.

2. Sincronizar, usando el menor número de semáforos posible, el siguiente grafo de precedencia:



3. Resolver el ejercicio anterior usando semáforos binarios (un semáforo binario sólo puede tomar dos valores, 1 o 0, es decir que si se hacen dos signal consecutivos sobre él, el segundo no tendrá efecto).

PROBLEMA PRODUCTORES/CONSUMIDORES

Sea un conjunto de procesos:

- ➔ unos “producen” datos que
- ➔ otros “consumen” con
- ➔ diferentes ritmos de producción y consumición

Se trata de diseñar un **protocolo de sincronización** que:

> Permita a productores y consumidores funcionar de forma concurrente a sus ritmos respectivos de forma los elementos producidos sean consumidos en el orden exacto en el que son producidos

Ejemplos de productores/consumidores:

- Controlador de impresora que produce caracteres que son consumidos por la impresora
- Compilador que produce código ensamblador que es consumido por el ensamblador

Dependiendo de la capacidad de la memoria intermedia usada (finita o infinita), tendremos diferentes problemas.

P/C CON BUFFER ILIMITADO

```
var producido, mutex: semáforo (:= 0, 1);
```

```
Prod:   repeat
            <Produce mensaje>
            wait(mutex);
            <Deposita mensaje>
            signal(mutex);
            signal(producido);
            ...
        until false;
```

```
Cons.:  repeat
            ...
            wait(producido);
            wait(mutex);
            <Recoge mensaje>
            signal(mutex);
            <Consume mensaje>
        until false;
```

P/C CON BUFFER LIMITADO (Capacidad n)

```
var lleno, vacío, mutex: semáforo (:= 0, n, 1);
```

```
P: repeat
       <Produce mensaje>
       wait(vacío);
       wait(mutex);
       <Deposita mensaje>
       signal(mutex);
       signal(lleno);
   until false;
```

Bloquea al productor en espera de huecos

Incrementa el número de llenos

```
C: repeat
       wait(lleno);
       wait(mutex);
       <Recoge mensaje>
       signal(mutex);
       signal(vacio);
       <Consume mensaje>
   until false;
```

Bloquea al consumidor en espera de mensajes

Incrementa el número de huecos

PROBLEMA DE LOS LECTORES/ESCRITORES

Sean dos categorías de procesos que usan una estructura compartida de datos:

- ➔ un lector nunca modifica la estructura de datos (puede haber acceso simultáneo de varios lectores)
- ➔ un escritor puede leer y escribir de ella (es necesario que tengan acceso exclusivo)

Se trata de diseñar un **protocolo de sincronización** que:

> Asegure la consistencia de los datos comunes a la vez que se mantiene el mayor grado de concurrencia que sea posible

Ejemplo:

- Control de acceso a ficheros en sistemas de bases de datos multiusuario

El problema de Lectores/Escritores tiene distintas variantes, según se dé prioridad a unos o a otros:

- ➔ Primer problema de L/E (prioridad a lectores).
  Ningún lector espera, a menos que un escritor haya obtenido ya permiso para modificar.
- ➔ Segundo problema de L/E (prioridad a escritores).
  Una vez que se detecta una petición de escritura, ésta se realiza tan pronto como sea posible. Es decir, si un escritor está esperando, ningún lector puede comenzar a leer.

Ambas estrategias pueden provocar espera indefinida (inanición de escritores o lectores, respectivamente).

Existen estrategias más complejas que garantizan el avance de lectores y escritores en un tiempo finito (Hoare).

PRIMER PROBLEMA DE L/E

```
var  escribe, mutex: semáforo (:= 1, 1);
     cont_lect: integer(:= 0);
```

```
L:  repeat
        wait(mutex);
        cont_lect := cont_lect + 1;
        if cont_lect = 1 then wait(escribe);
        signal(mutex);
        Lectura
        wait(mutex);
        cont_lect := cont_lect -1;
        if cont_lect = 0 then signal(escribe);
        signal(mutex);
    until false;
```

```
E:  repeat
        wait(escribe);
        Escritura
        signal(escribe);
    until false;
```

EJEMPLO: **EL BUFFER CIRCULAR (semáforos)**

```
var  buzon: array[o..n-1] of T;
     p, c: o..n-1 (:=0, 0);
     lleno, vacío: semáforo (:= 0, n);
     mutexp, mutexc: semáforo (:= 1, 1);
```

```
P:  repeat
        <Produce mensaje>
        wait(vacío);
        wait(mutexp);
            buzon[p] := mensaje;
            p := (p+1) mod n;
        signal(mutexp);
        signal(lleno);
    until false;
```

```
C:  repeat
        wait(lleno);
        wait(mutexc);
            mensaje := buzon[c];
            c := (c+1) mod n;
        signal(mutexc);
        signal(vacio);
        <Consume mensaje>
    until false;
```

☞ mensaje es una variable local de tipo T en productores y en consumidores.

PROBLEMAS DE LOS SEMÁFOROS

Supongamos que se está usando semáforos para resolver el problema de la sección crítica. Si no se respeta el protocolo (aunque sólo sea un proceso), se pueden presentar varias situaciones problemáticas:

	PROBLEMA
Un proceso intercambia las operaciones wait y signal	Se viola el principio de la exclusión mutua
Un proceso cambia signal por wait	Situación de interbloqueo
Un proceso omite wait, signal o ambas	Se violará la exclusión mutua o se producirá una situación de interbloqueo
Dos procesos usan wait y signal adecuadamente sobre dos semáforos de exclusión mutua, en orden opuesto	Situación de posible interbloqueo