Procesos concurrentes y memoria compartida.

- Si los diferentes procesos de un programa concurrente tienen acceso a **variables globales o secciones de memoria** comunes, la transferencia de datos a través de ella es una vía habitual de comunicación y sincronización entre ellos.
- Las primitivas para programación concurrente basada en memoria compartida resuelven los problemas de sincronización entre procesos y de exclusión mutua utilizando la semántica de acceso a memoria compartida.
- En esta familias de primitivas, la semántica de las sentencias hace referencia a la exclusión mutua, y la implementación de la sincronización entre procesos resulta de forma indirecta.

Notas:

Mecanismos basados en memoria compartida

- **Semáforos:** Son componentes pasivos de bajo nivel de abstracción que sirven para arbitrar el acceso a un recurso compartido.
- **Secciones críticas**: Son mecanismos de nivel medio de abstracción orientados a su implementación en el contexto de un lenguaje y que permiten la ejecución de un bloque de sentencias de forma segura.
- **Monitores**: Son módulos de alto nivel de abstracción orientados a la gestión de recursos que van a ser usados concurrentemente.

Notas:

Los mecanismos de sintronización que se introducen en este tema son tres:

* **Los semáforos** que fueron introducidos inicialmente por Dijkstra (1968). Son componentes de muy bajo nivel de abstracción, de fácil comprensión y con una gran capacidad funcional. Por el contrario, son unos componentes que por su bajo nivel de abstracción resultan muy peligrosos de manejar y frecuentemente son causa de muchos errores.

* **Las secciones críticas condicionales** (Brinch Hansen, 1972) son unos mecanismos introducidos como componentes de los lenguajes de programación concurrente. Tienen un nivel de abstracción mucho mas alto que los semáforos, y en consecuencia son más fáciles y seguros de manejar.

* **Los monitores** fueron introducidos por Hoare (1974), y son unos módulos de programación de alto nivel de abstracción que resuelven internamente, el acceso de forma segura a una variable o a un recurso compartido por múltiples procesos concurrentes.

Definición de semáforo.

- Un semáforo es un **tipo de datos**.
- Como cualquier tipo de datos, queda definido por:
  - Conjunto de **valores** que se le pueden asignar.
  - Conjunto de **operaciones** que se le pueden aplicar.
- Un semáforo tiene asociada una **lista de procesos**, en la que se incluyen los procesos suspendidos a la espera de su cambio de estado.

Notas:

Un semáforo es un tipo abstracto de dato, y como tal, su definición requiere especificar sus dos atributos básicos:

* Conjunto de valores que puede tomar.
* Conjunto de operaciones que admite.

Un semáforo tiene también asociada una **lista de procesos,** en la que se incluyen todos los procesos que se encuentra suspendidos a la espera de acceder al mismo.

Valores de un semáforo.

- En función del rango de valores que puede tomar, los semáforos se clasifican en:
  - Semáforos **binarios**: Pueden tomar solo los valores 0 y 1.

    var mutex: **BinSemaphore**;
  - Semáforos **general**: Puede tomar cualquier valor Natural (entero no negativo).

    var escribiendo: **Semaphore**;
- Un semáforo que ha tomado el valor **0** representa un **semáforo cerrado**, y si toma un valor **no nulo** representa un **semáforo abierto**.
- Mas adelante demostraremos que un semáforo General se puede implementar utilizando semáforos Binarios.
- Los sistemas suelen ofrecer como componente primitivo semáforos generales, y su uso, lo convierte de hecho en semáforo binario.

Notas:

Un semáforo puede tomar **valores** enteros no negativos (esto es, el valor 0 o un valor entero positivo). La semántica de estos valores es: 0 semáforo cerrado, y >0 semáforo abierto.

En función del rango de valores positivos que admiten, los semáforos se pueden clasificar en:

* **Semáforos binarios**: Son aquellos que solo pueden tomar los valores 0 y 1.
* **Semáforos generales**: Son aquellos que pueden tomar cualquier valor no negativos.

Frecuentemente, el que un semáforo sea binario o general, no es función de su estructura interna sino de como el programador lo maneja.

Operaciones seguras de un semáforo.

- Un semáforo

var p: **semaphore**;

admite dos **operaciones seguras**:

- **wait**(p): Si el semáforo no es nulo (abierto) decrementa en uno el valor del semáforo. Si el valor del semáforo es nulo (cerrado), el thread que lo ejecuta se suspende y se encola en la lista de procesos en espera del semáforo.

- **signal**(p); Si hay algún proceso en la lista de procesos del semáforo, activa uno de ellos para que ejecute la sentencia que sigue al wait que lo suspendió. Si no hay procesos en espera en la lista incrementa en 1 el valor del semáforo.

Notas:

Los semáforos admiten dos **operaciones**:

* Operación **Wait (P)**: Si el valor del semáforo no es nulo, esta operación decrementa en uno el valor del semáforo. En el caso de que su valor sea nulo, la operación suspende el proceso que lo ejecuta y lo ubica en la lista del semáforo a la espera de que deje de ser nulo el valor.

* Operación **Signal (V)**: Incrementa el valor del semáforo, y en caso de que haya procesos en la lista de espera del semáforo, se activa uno de ellos para que concluya su operación Wait.

Lo importante del semáforo es que se garantiza que la operación de chequeo del valor del semáforo, y posterior actualización según proceda, es siempre **segura** respecto a otros accesos concurrentes.

Operación no segura de un semáforo.

- Un semáforo

  var p: **semaphore**;

  admite una **operación no segura**:

  - **initial(p, Valor_inicial)**: Asigna al semáforo p el valor inicial que se pasa como argumento.

- Esta operación es no segura y por tanto debe ser ejecutada en una fase del programa en la que se tenga asegurada que se ejecuta sin posibilidad de concurrencia con otra operación sobre el mismo semáforo.

Notas:

Operación wait.

- Pseudocódigo de la operación: **wait**(p);

  if p>0
  then p:= p-1;
  else Suspende el proceso y lo encola en la lista del semáforo.

- Lo característico de esta operación es que está **protegida contra interrupción** entre el chequeo del valor del semáforo y la asignación del nuevo valor.

- 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.

Notas:

La operación **wait** lleva a cabo la siguiente secuencia de operaciones:

if Semáforo > 0
then Semaforo:= Semaforo -1
else Suspende al proceso en la lista del semáforo;

La operación wait (como la semántica de su nombre indica) es una potencial causa de retraso en la ejecución de un proceso. Sin embargo, es importante resaltar que no siempre que se ejecute una sentencia wait se produce el retraso, sino solo cuando al ejecutarla, el semáforo este cerrado, esto es tiene el valor 0.

Operación signal.

- Pseudocódigo de la operación: **signal**(p);

  if Hay algún proceso en la lista del semáforo
  then Activa uno de ellos
  else p:= p+1;

- La ejecución de la operación signal(p) nunca provoca una suspención del thread que lo ejecuta.

- Si hay varios procesos en la lista del semáforo, la operación signal solo activa uno de ellos. Este se elige de acuerdo con un criterio propio de la implementación (FIFO, LIFO, Prioridad, etc.).

Notas:

Ejemplo: Exclusión mutua.

```
program Exclusion_Mutua;
      var mutex: binsemaphore;
      process type Proceso;
      begin
        repeat
              wait(mutex);
                 (* Código de la sección crítica *)
              signal(mutex);
        forever;
      end;
var p, q, r: Proceso;
begin
   initial(mutex,1);
   cobegin p; q; r; coend;
end;
```


Notas:

Se plantea una solución del problema de exclusión mutua entre tres procesos P, Q y R, respecto de una sección crítica dada. Se va a utilizar un semáforo "Seccion_Libre" para controlar el acceso a la misma.

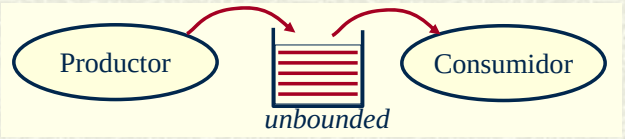
El valor del semáforo indica el número de procesos que se encuentran ejecutando la sección crítica. Cuando toma el valor 0, significa que algún proceso está en su sección crítica, mientras que cuando toma el valor 1 significa que no hay ningún proceso ejecutándola. La estructura del programa garantiza que el semáforo es binario, y nunca puede tomar un valor superior a 1.

De acuerdo con la semántica expuesta, el semáforo debe ser inicializado al valor 1, ya que al comenzar, ningún proceso se encuentra en la zona crítica.

Cuando uno de los procesos va a iniciar su sección crítica, ejecuta una sentencia wait. Si no hay otro proceso ejecutando la sección crítica, el proceso accede directamente a ejecutarla sin necesidad de ninguna espera. En caso contrario, se incorpora a la lista asociada al semáforo en espera de que le toque su turno de ejecución de la sección crítica.

Cuando uno de los procesos concluye la ejecución de su sección crítica, ejecuta una sentencia signal y continua la ejecución de las sentencias posteriores a esta, sin esperar mayor confirmación. Con la ejecución de la sentencia signal el semáforo (o mejor el sistema operativo que lo soporta) analiza la lista de procesos en espera, y si hay al menos uno en ella, le activa y le permite concluir la sentencia wait que lo introdujo en la lista.

Sincronización basada en semáforos.




```
var datoDisponible: Semaphore:= 0;

process Productor;
var dato: Tipo_Dato;
begin
  repeat
    produceDato(var dato);
    dejaDato(dato);
    signal(datoDisponible);
  forever;
end:

process Consumidor;
var dato: Tipo_Dato;
begin
  repeat
      wait(datoDisponible);
      tomaDato(var dato);
      consume(dato);
  forever;
end;
```


Notas:

Este representa una implementación del problema de productor consumidor con buffer ilimitado.

La condición de buffer ilimitado solo no introduce límite a la operación del productor. La única restricción es la del consumidor (este no puede tomar el dato si este no se ha producido). Esta restricción se formula en función de la condición de sincronización:

$\forall$ e $\bullet$ (producción^i $\rightarrow$ consumisión^i)

El problema que se plantea es que el consumidor debe esperar sin leer el buffer hasta tanto que el dato que corresponda haya sido previamente colocado. Obviamente, el consumidor es el que tiene que esperar en este caso, luego debe contener una sentencia wait.

Esta versión resuelve el problema productor consumidor entre dos procesos. Sin embargo no prevé la posibilidad de que en el caso de que existan varios productores y varios consumidores, estos no pueden dejar simultáneamente el dato en el buffer o no puedan tomar simultáneamente el dato del buffer. Un problema se va a producir si las actividades Dejar_dato y Tomar_dato no son atómica.

Productor-Consumidor (múltiples productores y consumidores)

```
datoDisponible: Semaphore:=0;
mutex: BinSemaphore:=1;

process type Productor;
var dato:Tipo_Dato;
begin
  repeat
       dato:=Produce_Dato;
       wait(mutex);
       dejaDato(dato);
       signal(mutex);
       signal(datoDisponible);
  forever;
end;
```

```
process type Consumidor;
   var dato:Tipo_Dato;
begin
   repeat
     wait(datoDisponible);
     wait(mutex);
     dato:=tomaDato;
     signal(mutex);
     consume(dato);
   forever;
end;
```


Notas:

El semáforo Dato_Disponible suspende a los consumidores mientras que no haya datos disponibles.

El semáforo Nop garantiza que el acceso al buffer por parte de múltiples productores y consumidores se haga con exclusión mutua.

Productor consumidor con buffer limitado (1).

```
program Productor_Consumidor;
const LONG_BUFFER = 5;
type Tipo_Dat= ....;
var datoDisponible:Semaphore;
      hueco:Semaphore;
      mutex: BinSemaphore;
      buffer: record datos:array[0..LONG_BUFFER-1]of  Tipo_Dato;
                     nextIn, nextOut: Natural:=0; end;

procedure dejaDato(d:Tipo_Dat);
begin
  buffer.dato[buffer.nextIn]:=D;
  buffer.nextIn:=(buffer.nextIn+1)
              mod LONG_BUFFER;
end;

procedure tomaDato(d:Tipo_Dat);
 begin
  d:=buffer.dato[buffer.nextOut];
  buffer.nextOut:= (buffer.nextOut+1)
                  mod LONG_BUFFER;
end;
```


Notas:

En este caso, existen condiciones de sincronismo tanto para el consumidor como para el productor:

* El consumidor no puede consumir el dato i_esimo, antes de que haya sido producido:

$$\forall e \bullet (\text{producción}^i \rightarrow \text{consumisión}^i)$$

* El productor no puede producir un nuevo dato, si el buffer está lleno:

$$\forall e \bullet (\text{consumisión}^i \rightarrow \text{producción}^{\{i + Long\_buffer\}})$$

Productor consumidor con buffer limitado (2).

```
process productor;
var dato:Tipo_Dat;
 begin
   repeat
     dato:=produceDato;
     wait(hueco);
     wait(mutex);
     dejaDato(dato);
     signal(mutex);
     signal(datoDisponible);
   forever;
 end;
```

```
process consumidor;
var dato: Tipo_Dat;
begin
  repeat
     wait(datoDiponible);
     wait(mutex);
     tomaDato(dato);
     signal(mutex);
     signal(hueco);
     Consume(dato);
  forever;
 end;
```

```
begin
   initial(mutex,1); initial(hueco, LONG_BUFFER); initial(datoDisponible,0);
   cobegin productor; consumidor; coend;
end.
```


Notas:

Semáforos generales y semáforos binarios

```
type SemGral = record
          mutex, espera: BinSemaphore;
          cuenta: Natural;  end;
procedure initialGral(var s:SemGral, v:Natural);
begin initial(s.mutex,1); v.cuenta:= v;
        if (v=0) then initial(s.espera,0) else initial(s.espera,1);
end;
procedure waitGral(var s:SemGral);
begin
  wait(s.espera);
  wait(s.mutex);
  s.cuenta:= s.cuenta - 1;
  if (s.cuenta>0) then signal(s.espera);
  signal(s.mutex);
end;
```

```
procedure signalGral(var s:Sem_Gral);
begin
  wait(s.mutex)
   s.cuenta:= s.cuenta +1;
   if (s.cuenta=1) then signal(s.espera);
   signal(s.mutex);
end;
```


Notas:

Un semáforo General puede implementarse utilizando semáforos binarios. Para ello, vamos a demostrar como utilizando una estructura de datos compuesta de una variable entera y dos semáforos binarios, se puede obtener una estructura equivalente a un semáforo general, y así mismo, como se pueden construir operaciones wait y signal seguras sobre esta estructura utilizando sentencias estándar, y las operaciones wait y signal sobre los semáforo binario.

MutEx es un semáforo binario que garantiza la exclusión mutua en las operaciones que se ejecutan sobre el semáforo general. Su valor de inicial debe ser 1. La variable entera "Cuenta" contiene el valor del semáforo general. "Espera" es también un semáforo binario que se utiliza para encolar los procesos que traten de realizar la operación wait cuando "Cuenta" sea 0. Su valor inicialde delay debe ser 0, si el valor inicial de Cuenta es 0, y debe ser 1 si el valor inicial de Cuenta es superior a 0.

Cena de filósofos chinos (1)

```
program Cena_filósofos_chinos;
const    N = 5;
var      palillo: array [1..N] of  BinSemaphore;
         sillasLibres: Semaphore;

process type TipoFilosofo(nombre: Integer); begin ... end;
var filosofo: array [1..N] of TipoFilosofo;
      i: Integer;

begin
  for i:=1 to N do initial(palillo[I],1);
  initial(sillasLibres,N-1);
  cobegin for i:=1 to N do filosofo[I](I); coend;
end.
```


Notas:

Se propone una solución libre de bloqueos, basada en permitir que solo N-1 de los N filósofos se puedan sentar simultáneamente para comer.

Cena de filósofos chinos (2)

```
process type TipoFilosofo(nombre: Natural);
  derecho=nombre;
  izquierdo=nombre mod (N+1);
begin
  repeat
     sleep(Random(5));                    -- Está pensando
     wait(sillaslibres);
     wait(palillo[derecho]);
     wait(palillo[izquierdo]);
      sleep(Random(5));                   -- Está comiendo
      signal(palillo[derecho]);
      signal(palillo[izquierdo]);
      signal(pillasLibres);
   forever
end;
```


Notas:

Implementación de los semáforos.

- La clave para implementar semáforos es disponer de un mecanismo(lock y unlock) que permita garantizar las secciones críticas de las primitivas wait y signal.

Wait	*Signal*
lock **if** s>0 **then** s:= s-1; **else** Suspende el proceso; **unlock**;	**lock** **if** Hay procesos suspendidos **then** Desbloquea uno de los procesos; **else** s:= s+1; **unlock**;

- Alternativas para implementarlos mecanismos **lock** y **unlock**:
  - Inhibición de la interrupciones de planificación.
  - Utilizar las instrucciones especiales (TAS) de que están dotadas la CPU.

Notas:

El mecanismo clave, para poder realizar semáforos, es disponer de algún mecanismo que garantice que las secciones críticas de las primitivas wait y signal, se hacen en exclusión mutua con otros procesos que ejecuten estas primitivas sobre el mismo semáforo.Para ello introducimos dos primitivas "lock" y "unlock" que realizan esta función.

Tres alternativas para implementar las operaciones básicas lock y unlock son:

* Un **método software** basado en un protocolo tal como los indicados en el capítulo II. Estos métodos presentan el problema de utilizar una espera activa, y en consecuencia son muy poco eficiente.

* **Inhibir las interrupciones** que en el sistema operativo atiende los eventos de reloj y que inducen los cambios de contexto por tiempo.

* Utilizar **instrucciones especiales** de las que están dotadas el hardware de la CPU, tales como TAS (Test and set), que permiten en una operación indescomponible, que verificar el estado y establecer el nuevo valor de acuerdo con él.

Ejemplo:

```
LOOP:
    TAS lockbyte  ; Si es 0, lo pone a 1.
    BNZ LOOP      ; Vuelve a intentarlo si era 0
    Sección crítica
    CLR lockbyte  ; unlock
```

Critica de los semáforos.

- Ventajas de los semáforos:
  - Resuelven todos los problemas que presenta la concurrencia.
  - Estructuras pasivas muy simples.
  - Fáciles de comprender.
  - Tienen implementación muy eficientes.
- Peligros de los semáforos:
  - Son de muy bajo nivel y un simple olvido o cambio de orden conducen a bloqueos.
  - Requieren que la gestión de un semáforo se distribuya por todo el código. La depuración de los errores en su gestión es muy dificil.
- Los semáforos son los componentes básicos que ofrecen todas las plataformas hardware y software como base para construir los mecanismos de sincronización. Las restantes primitivas se basan en su uso implícito.

Notas:

Los semáforos son 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, y los programadores consideran que introducen un peligro de fallo potencial muy alto. Estos se debe a que los semáforos son unas estructuras de muy bajo nivel de abstracción, y en consecuencia dejan demasiados aspectos a decidir por parte del programador.

- Peligros que introducen los semáforos son:

* 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.

* La solución de los mecanismos de exclusión mutua o de sincronización mediante semáforos, dan lugar a un código con operaciones wait y signal relativas a un mismo semáforo, muy dispersas por el código del programa, lo que constituye una grave dificultad para comprender el programa y para mantenerlo.