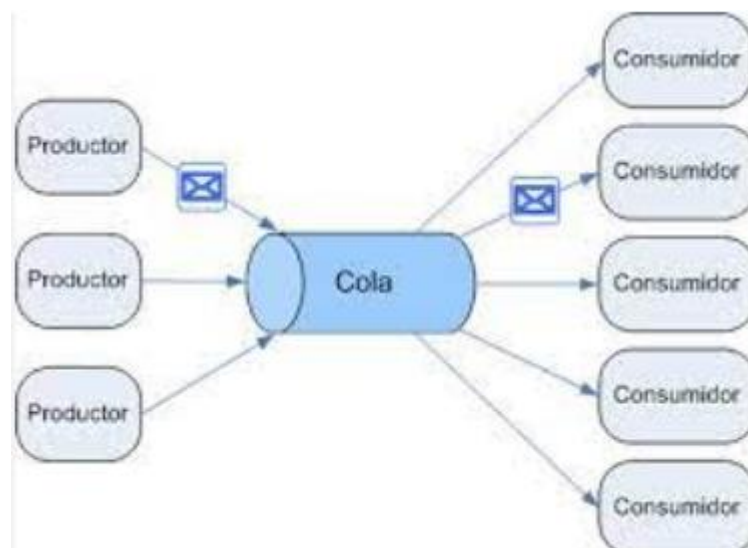


# Practica de Laboratorio: “El problema de Productor y Consumidor usando semáforos”

## Problema de productor consumidor

El problema del productor-consumidor es conocido como un problema de sincronización de múltiples procesos, aplicándose exclusión mutua mediante la compartición de un búfer común de tamaño fijo que usa cola:

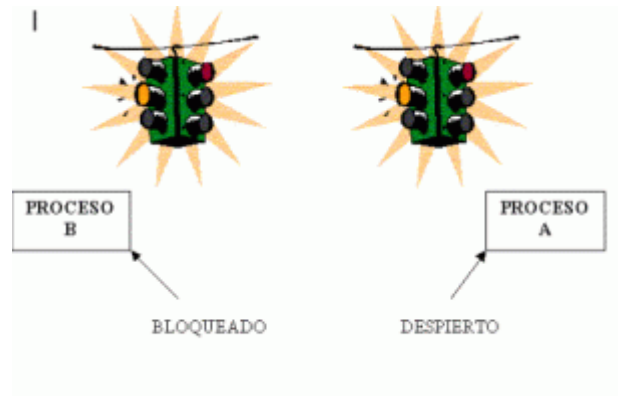
- El trabajo del productor es generar datos, ponerlos en el búfer y comenzar de nuevo.
  - El Productor introduce los datos a la sección crítica, al mismo tiempo el consumidor no debería consumir ningún elemento.
- Al mismo tiempo, el consumidor consume los datos (es decir, los elimina del búfer), una pieza a la vez.
  - El Consumidor extrae los datos desde la sección crítica, evitando de que el productor no añada elementos.



## Semáforos

Los semáforos son una herramienta de sincronización que ofrece una solución al problema de la sección crítica para dos o más procesos o hilos, controlando el acceso de múltiples procesos a un recurso común de forma ordenada y sin conflictos, también bajo entornos multiusuarios (multivariables).

Los semáforos resuelven el siguiente problema: *“¿Cómo coordinamos a los productores y consumidores, para que los productores no produzcan más ítems de los que se pueden almacenar en el momento, y los consumidores no adquieran más ítems de los que hay disponibles?”*.



En resumen, un semáforo es una solución con un bajo nivel de abstracción (contiene un bloque de código igual en todos los consumidores y productores), por lo cual se debe implementar en todo el programa, pero puede generar graves problemas de seguridad e implementación cuando más recursos son agregados, y si solo uno de estos no aplica la metodología del semáforo de manera acertada la *información* se corromperá.

## Semáforos con múltiples variables

Se emplean semáforos basados en productor y consumidor para entablar comunicación entre sub procesos (**despertar, notificar y dormir**) para asegurarse de que el productor no intente agregar datos en el búfer si está lleno y que el consumidor no intente eliminar datos de un búfer vacío:

- El productor debe ir a **dormir** o descartar datos si el búfer está lleno.
- La próxima vez que el consumidor retire un artículo del búfer, **notifica** al productor, quien comienza a llenar el búfer nuevamente.
- El consumidor puede irse a **dormir** si encuentra que el búfer está vacío.
- La próxima vez que el productor pone datos en el búfer, **despierta** al consumidor dormido.

```
type semáforo = record
    contador: entero;
    cola: list of proceso
end;

var s: semáforo;

wait(s):
    s.contador := s.contador - 1;
    if s.contador < 0
    then begin
        poner este proceso en s.cola;
        bloquear este proceso
    end;

signal(s):
    s.contador := s.contador + 1;
    if s.contador ≤ 0
    then begin
        quitar un proceso P de s.cola;
        poner el proceso P en la cola de listos
    end;
```

## Las operaciones con semáforo

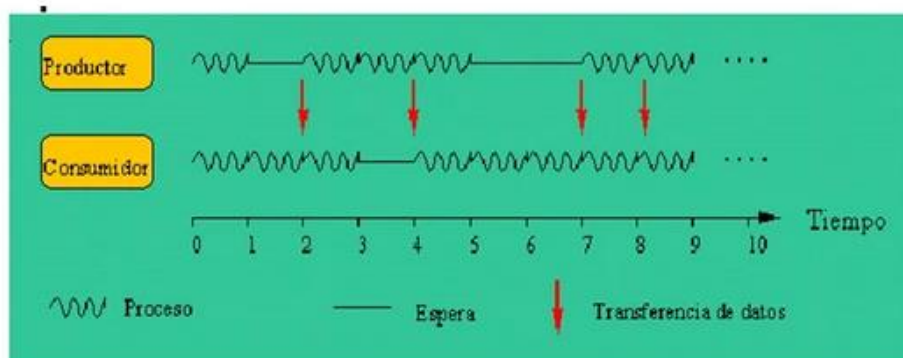
En código, existen 3 operaciones útiles para semáforos: WAIT, SIGNAL e INITIAL.

- La operación INITIAL inicializa el valor del semáforo a un mayor o igual a 0.
- La operación WAIT decrementa el valor del semáforo. Si el valor se hace negativo, el proceso que ejecuta WAIT queda bloqueado.
- La operación SIGNAL incrementa el valor del semáforo. Si el valor no es positivo, se desbloquea a un proceso bloqueado previamente por una operación WAIT.

El semáforo se inicializa con el número total de recursos disponibles (n) y las operaciones de WAIT y SIGNAL se diseñan de modo que se impida el acceso al recurso protegido por el semáforo cuando el valor de éste es menor o igual que cero.

De esta forma, cuando un hilo adquiere permiso al entrar a la sección crítica, al finalizar la respectiva sección crítica libera el permiso.

**Ejemplo:** El problema del productor-consumidor es un problema concurrente clásico con semáforos con múltiples variables.



**Problema:** El productor envía un dato cada vez, y el consumidor consume un dato cada vez. Si uno de los dos procesos no está listo, el otro debe esperar.

Un semáforo está formado por una posición de memoria y dos instrucciones, una para reservarlo (wait) y otra para liberarlo (signal). A esto se le puede añadir una cola de threads para recordar el orden en que se hicieron las peticiones.

Cada vez que se solicita y obtiene un recurso, el semáforo se decrementa y se incrementa cuando se libera uno de ellos.

Si la operación de espera se ejecuta cuando el semáforo tiene un valor menor que uno, el proceso debe quedar en espera de que la ejecución de una operación señal libere alguno de los recursos.

En CodeBlocks, los semáforos son creados usando las siguientes funciones:

Esta función de operación INITIAL posee 3 parametros:

- El primer parámetro inicializa un objeto semáforos apuntando a la variable "sem" para darle un valor inicial de tipo entero.

- El parámetro “pshared” controla el tipo de semáforo, si el valor es 0, entonces el semáforo está ubicado solo en el proceso actual, si el valor no es 0, el semáforo estará compartida entre procesos.

```
#include<semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

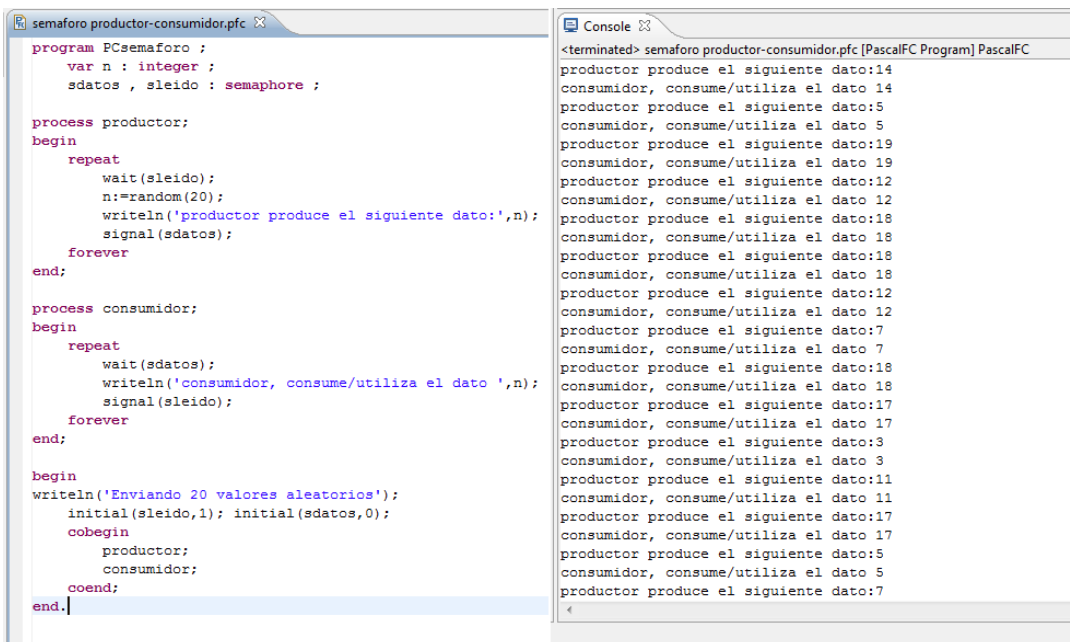
Estas funciones de operaciones WAIT y SIGNAL controlan el valor del semáforo a travez del valor inicial apuntado al parámetro “sem”.

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

## Casos Resueltos

### Ejemplo 1

Se muestra 20 elementos aleatorios de que pasan por el productor y el consumidor



```
program PCsemaphore ;
var n : integer ;
sdatos , sleido : semaphore ;

process productor;
begin
    repeat
        wait(sleido);
        n:=random(20);
        writeln('productor produce el siguiente dato:',n);
        signal(sdatos);
    forever
end;

process consumidor;
begin
    repeat
        wait(sdatos);
        writeln('consumidor, consume/utiliza el dato ',n);
        signal(sleido);
    forever
end;

begin
    writeln('Enviando 20 valores aleatorios');
    initial(sleido,1); initial(sdatos,0);
    cobegin
        productor;
        consumidor;
    coend;
end.
```

```
<terminated> semaphore productor-consumidor.pfc [PascalFC Program] PascalFC
productor produce el siguiente dato:14
consumidor, consume/utiliza el dato 14
productor produce el siguiente dato:5
consumidor, consume/utiliza el dato 5
productor produce el siguiente dato:19
consumidor, consume/utiliza el dato 19
productor produce el siguiente dato:12
consumidor, consume/utiliza el dato 12
productor produce el siguiente dato:18
consumidor, consume/utiliza el dato 18
productor produce el siguiente dato:18
consumidor, consume/utiliza el dato 18
productor produce el siguiente dato:12
consumidor, consume/utiliza el dato 12
productor produce el siguiente dato:7
consumidor, consume/utiliza el dato 7
productor produce el siguiente dato:18
consumidor, consume/utiliza el dato 18
productor produce el siguiente dato:17
consumidor, consume/utiliza el dato 17
productor produce el siguiente dato:3
consumidor, consume/utiliza el dato 3
productor produce el siguiente dato:11
consumidor, consume/utiliza el dato 11
productor produce el siguiente dato:17
consumidor, consume/utiliza el dato 17
productor produce el siguiente dato:5
consumidor, consume/utiliza el dato 5
productor produce el siguiente dato:7
```

## Ejemplo 2

Se imprime letras desde la letra "A" hasta la letra "Z" múltiples veces

```
program CasoPracticoSemaforo;
var n : integer ;
    sletra , simpreso : semaphore ;
    letra:char;
process productor;
begin
    repeat
        wait(simpreso);
        FOR letra := 'A' TO 'Z' DO
            begin
                writeln('se mandó a imprimir la letra: ',letra);
                signal(sletra);
            end;
        forever
    end;
process impresora;
begin
    repeat
        wait(sletra);
        writeln('se imprimio la letra: ',letra);
        signal(simpreso);
    forever
end;
begin
    initial(simpreso,1); initial(sletra,0);
    cobegin
        productor;
        impresora;
    coend;
end.
```

```
se mandó a imprimir la letra: A
se mandó a imprimir la letra: Bse imprimio la letra: B

se mandó a imprimir la letra:
se mandó a imprimir la letra: _
se imprimio la letra: Dse mandó a imprimir la letra: E
se imprimio la letra: E
se mandó a imprimir la letra: F
```

## Ejemplo 3

Asegurarse de que el proceso 2 no pueda ejecutar "d" hasta que el proceso 1 haya ejecutado "a"

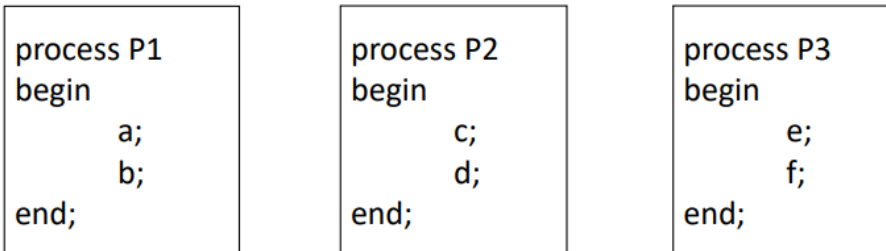
```
process P1
begin
    a;
    b;
end;
```

```
process P2
begin
    c;
    d;
end;
```

```
1  #include <iostream>
2  #include <thread>
3  #include <semaphore.h>
4  using namespace std;
5
6  sem_t s;
7
8  void procesol() {
9      while(true) {
10
11          cout << "A" << endl;
12          sem_post(&s); //signal
13          cout << "B" << endl;
14
15      }
16  }
17
18  void proceso2() {
19      while(true) {
20
21          cout << "C" << endl;
22          sem_wait(&s);
23          cout << "D" << endl;
24
25      }
26  }
27
28  int main()
29  {
30      sem_init(&s,0,0);
31
32      thread P1(procesol);
33      thread P2(proceso2);
34
35      P1.join();
36      P2.join();
37      return 0;
38  }
39
```

## Ejemplo 4

Asegurarse de que en el proceso 2 no pueda ejecutar “d”, si el proceso 3 ya ejecuto “e” o si el proceso 1 ya ejecuto “a”.



```
1  #include <iostream>
2  #include <thread>
3  #include <semaphore.h>
4  using namespace std;
5
6  sem_t s;
7
8  void proceso1(){
9      while(true){
10
11          cout << "A" << endl;
12          sem_post(&s);
13          cout << "B" << endl;
14      }
15  }
16
17
18  void proceso2(){
19      while(true){
20
21          cout << "C" << endl;
22          sem_wait(&s);
23          cout << "D" << endl;
24      }
25  }
26
27
28  void proceso3(){
29      while(true){
30
31          cout << "E" << endl;
32          sem_post(&s);
33          cout << "F" << endl;
34      }
35  }
36
37
38  int main()
39  {
40      sem_init(&s,0,0);
41
42      thread P1(proceso1);
43      thread P2(proceso2);
44      thread P3(proceso3);
45
46      P1.join();
47      P2.join();
48      P3.join();
49      return 0;
50  }
```

## Ejercicio 5

Se requiere que el proceso 2 ejecute "d" si el proceso 3 ejecuto "e" y el proceso 1 ejecuto "a".

```
process P1
begin
    a;
    b;
end;
```

```
process P2
begin
    c;
    d;
end;
```

```
process P3
begin
    e;
    f;
end;
```

```
1  #include <iostream>
2  #include <thread>
3  #include <semaphore.h>
4  using namespace std;
5
6  sem_t s, t;
7
8  void procesol(){
9      while(true){
10
11          cout << "A" << endl;
12          sem_post(&s);
13          cout << "B" << endl;
14      }
15  }
16
17
18  void proceso2(){
19      while(true){
20
21          cout << "C" << endl;
22          sem_wait(&s);
23          sem_wait(&t);
24          cout << "D" << endl;
25      }
26  }
27
28
29  void proceso3(){
30      while(true){
31
32          cout << "E" << endl;
33          sem_post(&t);
34          cout << "F" << endl;
35      }
36  }
37
38
39  int main()
40  {
41      sem_init(&s,0,0);
42      sem_init(&t,0,0);
43
44      thread P1(procesol);
45      thread P2(proceso2);
46      thread P3(proceso3);
47
48      P1.join();
49      P2.join();
50      P3.join();
51      return 0;
52  }
```



## Ejemplo 6

Mostrar los elementos producidos por el productor y consumidos por el consumidor usando semáforos.

```
1  #include <iostream>
2  #include <thread>
3  #include <semaphore.h>
4  using namespace std;
5
6  const int n = 3;
7
8  sem_t libres, mutex, ocupados;
9  int fin, frente, elemento;
10 int cola[n];
11
12 void Productor(){
13     while(true){
14
15         elemento = rand() % 10+1;
16         sem_wait(&libres);
17         sem_wait(&mutex);
18         cola[fin] = elemento;
19         fin = (fin + 1) % n;
20         cout << "ELEMENTO PRODUCIDO: " << elemento << endl;
21         sem_post(&mutex); //signal
22         sem_post(&ocupados);
23     }
24 }
25
26
27
28 void Consumidor(){
29     while(true){
30
31         sem_wait(&ocupados);
32         sem_wait(&mutex);
33         elemento = cola[frente];
34         frente = (frente + 1) % n;
35         sem_post(&mutex);
36         sem_post(&libres);
37         cout << "ELEMENTO CONSUMIDO: " << elemento << endl;
38     }
39 }
40
41
42
43 int main()
44 {
45     fin = n;
46     frente = 0;
47
48     sem_init(&mutex, 0, 1);
49     sem_init(&ocupados, 0, 0);
50     sem_init(&libres, 0, n);
51
52     thread productor(Productor);
53     thread consumidor(Consumidor);
54
55     productor.join();
56     consumidor.join();
57
58     return 0;
59 }
60
```

## Laboratorio

En código Pascal FC o en CodeBlocks resolver:

Mostrar conteo de veces en que ejecuta en productor y consumidor, iniciándolo desde productor, deberá seguir la siguiente secuencia:

- Aprobar o denegar acceso a una variable compartida, inicializarse con un valor no negativo.

**DATOS COMPARTIDOS**

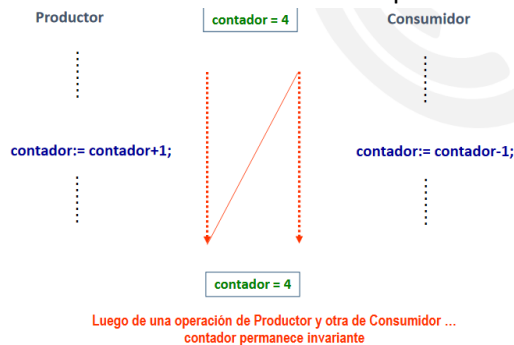
```

type item = ... ;
var buffer array [0..n-1] of item;
in, out: 0..n-1;
contador : 0..n;
in, out, contador := 0;
    
```

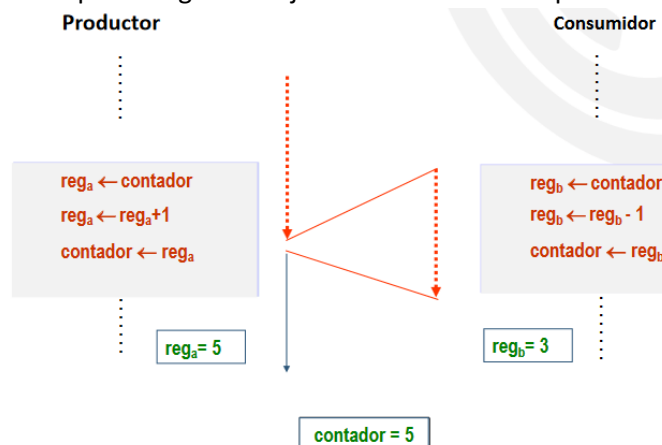
- Garantizar que los procesos compartidos sean usados y los datos sean actualizados.

| Proceso Productor   | Proceso Consumidor   |
|---|--|
| <pre> repeat   produce un item en nextp   ... while contador = n do no-op;   buffer[in] := nextp;   in := in + 1 mod n;   contador := contador + 1; until false;                     </pre> | <pre> repeat   while contador = 0 do no-op;     nextc := buffer[out];     out := out + 1 mod n;     contador := contador - 1;   consume el item en nextc   ... until false;                     </pre> |

- Ejecutar un conjunto de sentencias sobre variables compartidas.



- Emplear mecanismos para asegurar la ejecución ordenada de procesos cooperativos.



El resultado deberá ser mostrado similar a la siguiente manera:

```
Producer produced item : 0
Consumer consumed item : 0
Producer produced item : 1
Consumer consumed item : 1
Producer produced item : 2
Consumer consumed item : 2
Producer produced item : 3
Consumer consumed item : 3
Producer produced item : 4
Consumer consumed item : 4
```