Comunicación y Sincronización entre Procesos

UTN FRMDP - Arquitectura de Sistemas Operativos

Parte 1: Concurrencia

Concurrencia de Procesos

Se refiere a las situaciones en las que 2 o más procesos pueden coincidir en el acceso a un recurso compartido o, que requieren coordinarse en su ejecución.

Procesos Concurrentes

Los temas centrales del diseño de sistemas operativos están relacionados con la gestión de procesos e hilos:

- **Multiprogramación:** Gestión de múltiples procesos dentro de un sistema monoprocesador.
- **Multiprocesamiento:** Gestión de múltiples procesos dentro de un multiprocesador.
- **Multiprocesamiento distribuido:** Gestión de múltiples procesos que ejecutan sobre múltiples sistemas de cómputo distribuidos.

Procesos Concurrentes

La concurrencia aparece en tres contextos:

- **Múltiples aplicaciones:** La multiprogramación fue ideada para permitir compartir dinámicamente el tiempo de procesamiento entre varias aplicaciones.
- **Aplicaciones estructuradas:** Algunas aplicaciones pueden ser programadas como un conjunto de procesos concurrentes.
- **Estructura del Sistema Operativo:** Los sistemas operativos son muchas veces implementados como un conjunto de procesos o hilos.

Procesos Concurrentes

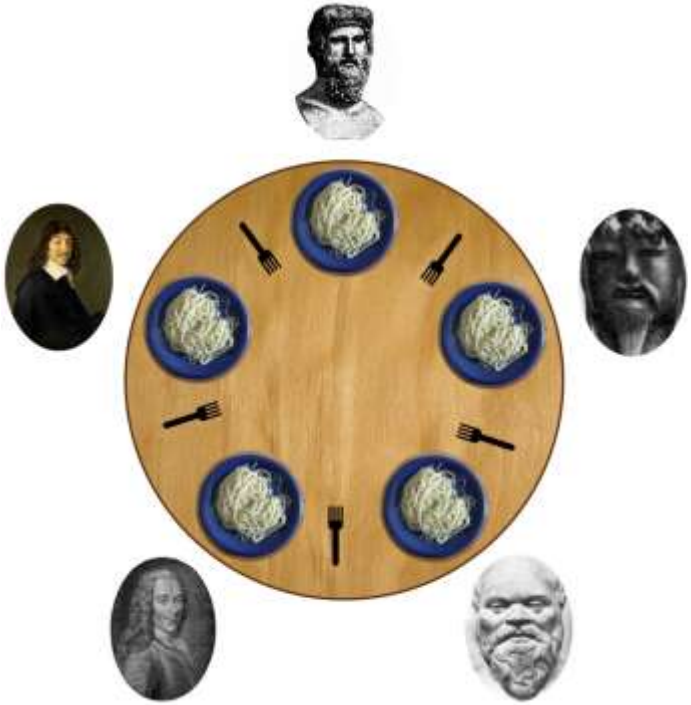
Se plantean las siguientes dificultades:

1. La compartición de recursos globales está llena de peligros.
2. Para el S.O. es complicado gestionar la asignación de recursos de manera óptima.
3. Llega a ser muy complicado localizar errores de programación porque los resultados son no deterministas y no reproducibles.

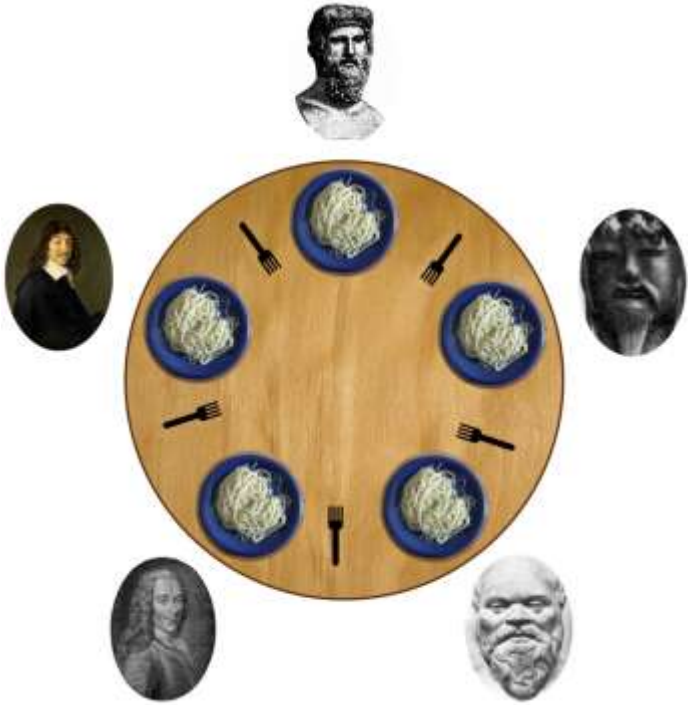
Algunos términos relacionados con la concurrencia

- **Sección Crítica (Critical Section):** sección de código de un proceso que requiere acceso a recursos compartidos.
- **Interbloqueo (Deadlock):** situación donde dos o más procesos son incapaces de actuar porque cada uno está esperando que alguno haga algo.
- **Círculo Vicioso (Live Lock):** dos o más procesos cambian continuamente su estado en respuesta a cambios en los otros procesos, sin realizar trabajo útil.
- **Exclusión Mutua (Mutual Exclusion):** requisito fundamental que evita que si un proceso está en su sección crítica, ningún otro proceso pueda entrar en su sección crítica.
- **Condición de Carrera (Race Condition):** situación donde múltiples hilos o procesos leen y escriben un dato compartido y el resultado final depende de la coordinación de sus ejecuciones.
- **Inanición (Starvation):** Un proceso preparado para avanzar es soslayado indefinidamente por el planificador, aunque puede avanzar nunca se escoge.

Ejemplo: CENA DE FILÓSOFOS

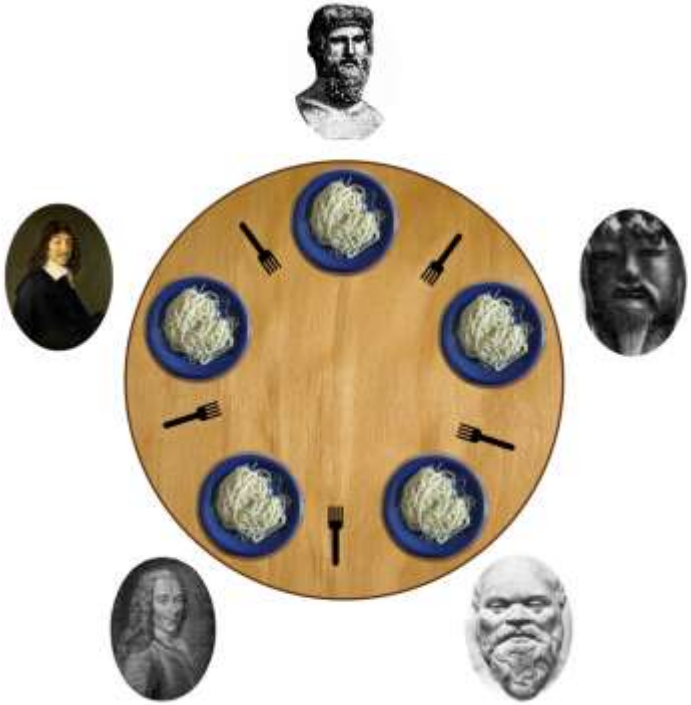


Cinco filósofos se sientan alrededor de una mesa a comer y pensar. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Para comer los fideos son necesario dos tenedores y cada filósofo puede tomar los que están a su izquierda o derecha. Si un filósofo toma un tenedor y el otro está ocupado, se quedará esperando con el tenedor en la mano hasta que pueda tomar el otro.



¿Qué pasa si dos filósofos intentan tomar el mismo tenedor a la vez?

Se produce una *condición de carrera* o *race condition*: ambos compiten y uno de ellos se queda sin comer.



¿Y si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo?

Se quedarán esperando eternamente, por lo tanto se produce un *interbloqueo* o *deadlock*.

Posible solución: **Exclusión Mutua**

- ★ Los **algoritmos de exclusión mutua** se usan en programación concurrente para evitar que entre más de un proceso a la vez en la *sección crítica*.

- ★ La **sección crítica** es el fragmento de código donde puede modificarse un recurso compartido.

Requisitos de la Exclusión Mutua

- Sólo se permite un proceso al mismo tiempo dentro de su sección crítica para el mismo recurso u objeto compartido.
- Un proceso que no está en su sección crítica no debe impedir que otro proceso acceda a la suya.
- No puede pasar que un proceso que solicite acceso a la sección crítica sea postergado indefinidamente (ni inanición ni interbloqueo).
- Cuando ningún proceso esté en su sección crítica, cualquier proceso que solicite entrar, debe permitírselo sin demora.
- No se hacen suposiciones sobre las velocidades relativas de los procesos ni sobre el número de procesadores.
- Un proceso debe permanecer dentro de su sección crítica por un tiempo finito.

Exclusión Mutua

Un proceso se lo puede ver de la siguiente forma:

```
sección_residual

entrada_sección_crítica

seccion_crítica

salida_sección_crítica
```

Dónde:

- **seccion_residual**: es el fragmento del proceso que no necesita acceso exclusivo.
- **entrada_sección_crítica** y **salida_sección_crítica**: se debe garantizar el acceso exclusivo, por eso se los conoce como primitivas de exclusión mutua.

Exclusión Mutua para dos procesos

Técnicas de Software

Veremos distintas tentativas de solución que surgieron para manejar la exclusión mutua, aunque son incorrectas sirvieron para poder llegar a una solución completa y sin errores.

Consideraremos 4 intentos de solución donde se planteará su algoritmo y posibles errores.

Exclusión Mutua para dos procesos – Primer intento

Se utiliza una variable global *turno* que es compartida por dos procesos:

```
int turno = 0;

void proceso0(){
   ...
   while(turno != 0);
   /* sección crítica */
   turno = 1;
   ...
}
```

Espera Activa

```
void proceso1(){
   ...
   while(turno != 1);
   /* sección crítica */
   turno = 0;
   ...
}
```

Exclusión Mutua para dos procesos – Primer intento

Garantiza la propiedad de exclusión mutua pero tiene dos desventajas:

- ➔ Los procesos deben alternarse en el uso de su sección crítica por lo tanto el ritmo de ejecución viene dictado por el proceso más lento.
- ➔ Si un proceso falla, el otro quedará permanentemente bloqueado.

Esta solución es una *corruptiva*. Están diseñadas para pasar el control de ejecución entre ellas mismas.

Exclusión Mutua para dos procesos – Segundo intento

Cada proceso debe llevar su propia *«llave»* para entrar en la sección crítica, de modo que si uno falla, el otro puede continuar accediendo:

```
int estado[2] = {0, 0};

void proceso0(){
  ...
  while(estado[1]);
  estado[0] = 1;
  /* sección crítica */
  estado[0] = 0;
  ...
}
```

```
void proceso1(){
  ...
  while(estado[0]);
  estado[1] = 1;
  /* sección crítica */
  estado[1] = 0;
  ...
}
```

Exclusión Mutua para dos procesos – Segundo intento

Si un proceso falla fuera de la sección crítica, el otro proceso no queda bloqueado. Sin embargo, si un proceso falla dentro de su sección crítica o después de establecer su estado, el otro proceso quedará permanentemente bloqueado.

Esta situación es peor que la anterior ya que no garantiza la exclusión mutua, evaluemos la siguiente traza de ejecución:

Exclusión Mutua para dos procesos – Segundo intento

	Proceso0	*Proceso1*
0	`while(estado[1]); ⊘`	…
1	…	`while(estado[0]); ⊘`
2	`estado[0] = 1;`	…
3	…	`estado[1] = 1;`
4	`/* sección crítica */`	…
5	…	`/* sección crítica */`
6		

Exclusión Mutua para dos procesos – Tercer intento

El segundo intento falla debido a que un proceso puede cambiar su estado después de que otro proceso lo haya cambiado. En este intento se trata de solucionar intercambiando dos sentencias:

```
void proceso0(){
  ...
  estado[0] = 1;
  while(estado[1]);
  /* sección crítica */
  estado[0] = 0;
  ...
}
```

```
void proceso1(){
  ...
  estado[1] = 1;
  while(estado[0]);
  /* sección crítica */
  estado[1] = 0;
  ...
}
```

Exclusión Mutua para dos procesos – Tercer intento

Este intento de solución garantiza la exclusión mutua. Desde el punto de vista del P0, una vez que establece su estado a verdadero, P1 no puede entrar a su sección crítica hasta que P0 lo haya abandonado.

Pero este intento genera un nuevo problema:

Exclusión Mutua para dos procesos – Tercer intento

	Proceso0	*Proceso1*
0	`estado[0] = 1;`	…
1	…	`estado[1] = 1;`
2	`while(estado[1]);` ✓	…
3	…	`while(estado[0]);` ✓

DEADLOCK

Exclusión Mutua para dos procesos – Cuarto intento

En esta tentativa cada proceso se vuelve más *«respetuoso»*. Cada uno establece si estado en verdadero pero está preparado a cambiar su estado si otro decide entrar.

```
void proceso0(){
  ...
  estado[0] = 1;
  while(estado[1]){
    estado[0] = 0;
    //retraso
    estado[0] = 1;
  }
  /* sección crítica */
  estado[0] = 0;
  ...
}
```

```
void proceso1(){
  ...
  estado[1] = 1;
  while(estado[0]){
    estado[1] = 0;
    //retraso
    estado[1] = 1;
  }
  /* sección crítica */
  estado[1] = 0;
  ...
}
```

Exclusión Mutua para dos procesos – Cuarto intento

Está cerca a una solución correcta, pero todavía falla. La exclusión mutua está garantizada.

Sin embargo siguiendo la siguiente secuencia de eventos:

Exclusión Mutua para dos procesos – Cuarto intento

	Proceso0	*Proceso1*
0	`estado[0] = 1;`	…
1	…	`estado[1] = 1;`
2	`while(estado[1])` ✓	…
3	…	`while(estado[0])` ✓
4	`estado[0] = 0;`	…
5	…	`estado[1] = 0;`
6	`estado[0] = 1;`	…
7	…	`estado[1] = 1;`
8	`while(estado[1])` ✓	…

LIVE LOCK

Exclusión Mutua para dos procesos
Algoritmo de Dekker

Es necesario observar el estado de ambos procesos, que se consigue mediante la variable *estado*, pero no es suficiente. Se debe imponer un *orden*.

Se puede utilizar la variable *turno* del primer intento, e indica que proceso tiene el derecho a insistir en entrar a su sección crítica.

Algoritmo de Dekker

```
int estado[2] = {0, 0};
int turno = 1;

void proceso0(){
  while(true){
    estado[0] = 1;
    while(estado[1])
    if(turno == 1){
      estado[0] = 0;
      while(turno == 1);
      estado[0] = 1;
   }
  /* sección crítica */
  turno = 1; estado[0] = 0;
... }}
```

```
void proceso1(){
  while(true){
    estado[1] = 1;
    while(estado[0])
    if(turno == 0){
      estado[1] = 0;
      while(turno == 0);
      estado[1] = 1;
   }
  /* sección crítica */
  turno = 0; estado[1] = 0;
... }}
```

Exclusión Mutua para dos procesos
Algoritmo de Peterson

El **algoritmo de Dekker** resuelve el problema de exclusión mutua pero con un programa bastante complejo.

Peterson proporcionó en 1981 una solución más simple.

Algoritmo de Peterson

```
void proceso0(){
  while(true){
    estado[0] = 1;
    turno = 1;
    while(estado[1] && turno == 1);
    /* sección crítica */
    estado[0] = 0;
    …
  }
}
```

```
void proceso1(){
  while(true){
    estado[1] = 1;
    turno = 0;
    while(estado[0] && turno == 0);
    /* sección crítica */
    estado[1] = 0;
    …
  }
}
```

DEKKER	booleano estado[2] = {0, 0}; int turno = 1;
P0	**P1**
while(true)	
	while(true)
estado[0] = true;	
	estado[1] = true;
while(estado[1])	
	while(estado[0])
if(turno==1)	
	if(turno==0)
estado[0] = false;	
	/* sección crítica */
while(turno == 1)	
	turno = 0;
while(turno == 1)	
	estado[1] = false;
estado[0] = true;	
	while(true)
/* sección crítica */	
	estado[1] = true;
turno = 1;	
	while(estado[0])
estado[0] = false;	
	if(turno==0)
while(true)	
	/* sección crítica */

PETERSON	booleano estado[2] = {0, 0}; int turno = 1;
P0	**P1**
while(true)	
	while(true)
estado[0] = true;	
	estado[1] = true;
turno = 1;	
	turno = 0;
while(estado[1]&&turno==1)	
	while(estado[0]&&turno==0)
/* sección crítica */	
	while(estado[0]&&turno==0)
estado[0] = false;	
	while(estado[0]&&turno==0)
while(true)	
	/* sección crítica */
estado[0] = true;	
	estado[1] = false;
turno = 1;	
	while(true)
while(estado[1]&&turno==1)	
	estado[1] = true;
/* sección crítica */	
	turno = 0;
estado[0] = false;	
	while(estado[0]&&turno==0)

Comparación de Algoritmos de Dekker y Peterson (Enlace)

Exclusión Mutua para dos procesos

Soporte Hardware

La solución de software es fácil que tenga una alta sobrecarga de procesamiento y el significativo el riesgo de errores lógicos.

Consideraremos varias soluciones de hardware a la exclusión mutua.

Deshabilitar Interrupciones

En una máquina monoprocesador, los procesos concurrentes no pueden solaparse, solo pueden entrelazarse. Un proceso continuará ejecutándose hasta que se invoque un servicio del sistema operativo o hasta que sea interrumpido.

Por tanto para garantizar la exclusión mutua, hay que impedir que un proceso sea interrumpido.

Un proceso puede cumplir la exclusión mutua de este modo:

```
while(true){
  /* deshabilitar interrupciones */
  /* sección crítica */
  /* habilitar interrupciones */
  /* resto */
}
```

Deshabilitar Interrupciones

Dado que la sección crítica no puede ser interrumpida, se garantiza la exclusión mutua, aunque el precio de esta solución es alto…

- ➔ La eficiencia de ejecución podría degradarse notablemente porque se limita la capacidad del procesador.
- ➔ Esta solución no funcionará sobre una arquitectura multiprocesador.

Instrucciones máquina especiales

Los diseñadores de procesadores han propuesto varias instrucciones máquina que llevan a cabo dos acciones atómicamente, como leer o escribir o leer y comprobar, sobre una única posición de memoria con un único ciclo de búsqueda de instrucción. Durante la ejecución de la instrucción, se bloquea el acceso a toda otra instrucción que referencia esa posición.

Veremos la instrucción *test and set* y la instrucción *exchange*.

Instrucciones máquinas especiales – Test and set

La instrucción comprueba el valor del argumento **i**. Si el valor es **0**, entonces la instrucción reemplaza el valor por **1** y devuelve *true*. En caso contrario, el valor no se cambia y devuelve *false*. Toda la instrucción se garantiza que se ejecute de manera atómica.

La instrucción se define de la siguiente manera:

```
boolean testset(int i){
  if(i == 0){
    i = 1;
    return true;
  }else{
    return false;
  }
}
```

Instrucciones máquinas especiales– Instrucción Exchange

La instrucción **intercambia** los contenidos de un registro con los de una posición de memoria.

La *instrucción exchange* puede definirse de la siguiente manera:

```
void exchange(int registro, int memoria){
  int temp;
  temp = memoria;
  memoria = registro;
  registro = temp;
}
```

Ejemplo de Uso – Instrucción Exchange

Una variable compartida *cerrojo* se inicializa en 0, cada proceso utiliza una variable local *llave* que se inicializa en 1.

El único proceso que puede entrar en su sección crítica es aquel que encuentra *cerrojo* igual a 0, y al cambiar *cerrojo* en 1 se excluyen a todos los otros procesos de la sección crítica.

Cuando un proceso abandona la sección crítica, se restaura *cerrojo* al valor 0.

```
int cerrojo = 0;


void P(int cerrojo){
  int llave = 1;
  while(true){
    exchange(llave, cerrojo);
    while(llave != 0);
    /* sección crítica */
    exchange(llave, cerrojo);
    ...
  }
}
```

Instrucciones máquinas especiales – Propiedades

La utilización de una instrucción máquina especial para conseguir la exclusión mutua tiene ciertas ventajas:

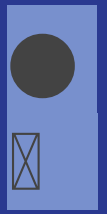
- ➔ Es aplicable a cualquier número de procesos sobre un procesador único o multiprocesador de memoria compartida.
- ➔ Es simple, y por tanto, fácil de verificar.
- ➔ Puede ser utilizado para dar soporte a múltiples secciones críticas: cada sección crítica puede ser definida por su propia variable.

Instrucciones máquinas especiales – Propiedades

Hay algunas desventajas serias:

- **Se emplea espera activa.** Así, mientras un proceso está esperando para acceder a una sección crítica, continúa consumiendo tiempo de procesador.
- **Es posible la inanición.** Cuando un proceso abandona su sección crítica y hay más de un proceso esperando, la selección del proceso en espera es arbitraria. Así a algún proceso podría negársele indefinidamente acceso.
- **Es posible el interbloqueo.** Supongamos que el P1 ejecuta una instrucción especial y entra a su sección crítica. Entonces el P1 es interrumpido para darle el procesador a P2, que tiene más prioridad. Si P2 intenta usar el mismo recurso que P1, se le denegara el acceso y caerá en una espera activa. P1 nunca será escogido por ser de menor prioridad que P2.

Parte 2: Semáforos



Un poco de historia…

El primer avance fundamental en el tratamiento de los problemas concurrentes ocurre en 1965 con el *tratado de Dijkstra*, que estaba involucrado en el diseño de un sistema operativo como una colección de procesos secuenciales cooperantes y con el desarrollo de mecanismos eficientes y fiables para dar soporte a la cooperación. Estos mecanismos podrían ser fácilmente usados por procesos de usuario si el procesador y el sistema operativo colaborasen en hacerlos disponibles.

Semáforos

El principio fundamental es: dos o más procesos pueden cooperar por medio de simples señales, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una señal específica.

Para la señalización se utilizan variables especiales llamadas *«semáforos»*.

Para transmitir una señal vía el semáforo ‘s’, el proceso ejecuta la primitiva *semSignal(s)*. Para recibir una señal vía el semáforo ‘s’, el proceso ejecutará la primitiva *semWait(s)*.

Semáforos con Contador

Las primitivas *semWait* y *semSignal* se asumen atómicas. Podemos ver una definición formal de las primitivas del semáforo:

```
struct semaforo{
  int contador;
  queue fila;
}
```

```
void semWait(semaforo s){
  s.contador--;
  if(s.contador < 0){
    agregarA(fila, proceso);
    bloquearProceso();
  }
}
```

```
void semSignal(semaforo s){
  s.contador++;
  if(s.contador >= 0){
    quitarDe(fila, proceso);
    ponerProcesoEnListos();
  }
}
```

Este tipo de semáforos se los conoce como *semáforos con contador*.

Semáforos con Contador

Para conseguir el efecto deseado, el semáforo puede ser visto como una variable que tiene un valor entero sobre el cual sólo están definidas tres operaciones:

- ➔ Un semáforo puede ser inicializado a un valor no negativo.
- ➔ La operación **semWait** decrementa el valor del semáforo. Si el valor pasa a ser negativo entonces el proceso que está ejecutando el **semWait** se bloquea.
- ➔ La operación **semSignal** incrementa el valor del semáforo.

Semáforos Binarios

Existe una versión restringida de los semáforos, conocida como *semáforo binario* o *mutex*, definido de la siguiente manera:

```
struct semaforoB{
  enum {0, 1} valor;
  queue fila;
}
```

```
void semWaitB(semaforoB s){
  if(s.valor == 1){
    s.valor = 0;
  }else{
    agregarA(fila, proceso);
    bloquearProceso();
  }
}
```

```
void semSignalB(semaforoB s){
  if(estaVacia(s.fila)){
    s.valor = 1;
  }else{
    quitarDe(fila, proceso);
    ponerProcesoEnListos();
  }
}
```

Semáforos Binarios

Un *semáforo binario* sólo puede tomar los valores 0 y 1 y se puede definir las siguientes operaciones:

- ➔ Un semáforo binario sólo puede ser inicializado en 0 o 1.
- ➔ Las operaciones *semWaitB* comprueba el valor de semáforo. Si el valor es 0, entonces el proceso que está ejecutando el *semWaitB* se bloquea.
- ➔ La operación *semSignalB* comprueba si hay algún proceso bloqueado en el semáforo. Si lo hay, entonces se desbloquea uno de los procesos bloqueados en la operación *semWaitB*. Si no hay, entonces el valor del semáforo se pone a uno.

Ejemplo: Semáforos

Consideremos n procesos, los cuales necesitan todos acceder al mismo recurso. Cada proceso ejecuta el *semWait(s)* justo antes de entrar a su sección crítica. Si el valor de s pasa a ser negativo, el proceso se bloquea. Si el valor es 1, entonces decrementa a 0 y el proceso entra a su sección crítica inmediatamente.

```
/* programa exclusión mutua */
semaforo s = 1;
void p(){
  while(true){
    semWait(s);
    /* sección crítica */
    semSignal(s);
    /* resto */
  }
}
```

Ejemplo: **Semáforos**

Este ejemplo puede servir igualmente si el requisito es que se permita más de un proceso en su sección crítica a la vez. Este requisito se cumple simplemente inicializando el semáforo al valor especificado. Así, el valor de *s.contador* puede ser interpretado como sigue:

- **s.contador >= 0:** s.cuenta es el número de procesos que pueden ejecutar *semWait(s)* sin suspensión. Tal situación permitirá a los semáforos admitir sincronización así como exclusión mutua.
- **s.contador < 0:** la magnitud de *s.contador* es el número de procesos suspendidos en *s.fila*.

Problema Productor - Consumidor

Problema Productor - Consumidor

*«Hay uno o más procesos generando algún tipo de dato y poniéndolos en un **buffer**. Hay un único consumidor que está extrayendo datos de dicho **buffer** de uno en uno.»*

Este es uno de los problemas más comunes en la programación concurrente.

El sistema está obligado a impedir la superposición de las operaciones sobre los datos. Es decir, sólo un agente (productor o consumidor) puede acceder al *buffer* en un momento dato.

Veremos algunas soluciones.

Problema Productor - Consumidor

```
#define BUFFER_SIZE 100;
int cantItems = 0;

void productor(){
  while(1){
    int item = producirItem();
    if(cantItems == BUFFER_SIZE)
      sleep();
    ponerItemBuffer(item);
    cantItems++;
    if(cantItems == 1)
      wakeup(consumidor);
  }
}
```

```
void consumidor(){
  while(1){
    if(cantItems == 0)
      sleep();
    int item = quitarItemBuffer();
    cantItems--;
    if(cantItems == BUFFER_SIZE-1)
      wakeup(productor);
  }
}
```

Problema Productor - Consumidor

Si analizamos el código podemos ver como el *productor* introduce datos en un *buffer* y cómo el *consumidor* los extrae. Tenemos dos situaciones:

- ➔ Cuando *cantItems* valga **0**, el consumidor no tendrá items para extraer y consumir, por lo tanto ejecutará la instrucción *sleep* y se bloqueará. Cuando el productor ingrese un nuevo item al *buffer* ejecutará la instrucción *wakeup* despertando así, al consumidor.
- ➔ Cuando *cantItems* valga el tamaño máximo del *buffer* (**100**), el productor no tendrá más espacio para seguir produciendo por lo que ejecutará la instrucción *sleep* y se bloqueará. Cuando el consumidor extraiga un elemento del *buffer*, mandará una señal a través de *wakeup* para despertar al productor y que éste pueda seguir produciendo.

Pareciera una solución válida, pero tiene un problema muy grande. Veamos la siguiente traza de ejecución…

Problema Productor - Consumidor

`cantItems = 0;`

	Productor	*Consumidor*
0	…	`if(cantItems == 0)` ✓
1	`item = producirItem();`	…
2	`if(cantItems == BUFFER_SIZE)` ⊘	…
3	`ponerItemBuffer(item);`	…
4	`cantItems++;`	…
5	`if(cantItems == 1)` ✓	…
6	`wakeup(consumidor);`	…
7	…	`sleep();`
8	…	…

Ambos quedan bloqueados

Problema Productor - Consumidor

Se podría solucionar el problema planteado anteriormente implementando *semáforos*.

Consideremos *3 semáforos* que son declarados de manera global. Los semáforos *elementos* y *huecos* funcionan como *semáforos con contador* y el semáforo *mutex* como *semáforo binario*.

Problema Productor - Consumidor

```
#define BUFFER_SIZE 100;
semaphore huecos =
BUFFER_SIZE;
semaphore elementos = 0;
semaphore mutex = 1;
```

```
void productor(){
  while(1){
    item = producirItem();
    semWait(huecos);
    semWait(mutex);
    ponerItemBuffer(item);
    semSignal(mutex);
    semSignal(elementos);
  }
}
```

```
void consumidor(){
  while(1){
    semWait(elementos);
    semWait(mutex);
    item = quitarItemBuffer();
    semSignal(mutex);
    semSignal(huecos);
    consumirItem(item);
  }
}
```

Semáforos en C

Los semáforos pueden ser implementados en *C* junto con los *hilos* para esto, debemos agregar una nueva librería: *semaphore*.

```
#include <semaphore.h>
```

Para que los semáforos sirvan para todos los hilos, deben ser declarados de manera global. Para eso usamos el tipo de dato *sem_t*:

```
sem_t mutex;
```

Semáforos en C

Los semáforos deben ser inicializados para poder ser utilizados, esta inicialización generalmente se realiza en el *main*:

sem_init(sem_t sem, int pshared, unsigned int value);*

Donde:

- ➔ *sem_t* sem:* Es la dirección de memoria del semáforo que va a ser inicializado.
- ➔ *int pshared:* Si su valor es 0 el semáforo sólo puede ser usado por threads del mismo proceso. Si no es 0, se puede compartir entre diferentes procesos.
- ➔ *unsigned int value:* Valor en el cual va a ser inicializado el semáforo:
  - *Valor 0:* Indica que el semáforo comienza cerrado. ●
  - *Valor 1:* El semáforo comienza abierto. ▯
  - *Valor >1:* Se considera un semáforo con *contador*.

Semáforos en C

Funciones para manejo de *semáforos*:

- ➔ *sem_wait(sem_t* sem):* Equivalente al *semWait()* visto anteriormente. Verifica el valor del semáforo, si vale 0 bloquea al proceso/hilo impidiéndole continuar. Si vale 1, cambia el valor del semáforo a 0 y lo deja continuar. Si vale >1, le resta uno al semáforo y deja pasar al proceso/hilo.
- ➔ *sem_post(sem_t* sem):* Equivalente al *semSignal()* visto anteriormente. Incrementa el valor del semáforo.
- ➔ *sem_destroy(sem_t* sem):* Destruye el semáforo pasado por parámetro. Sólo un semáforo que fue inicializado con el *sem_init()* puede ser destruido.