Resumen

Semáforos

Malváez Flores Axel Daniel

Peralta Rionda Gabriel Zadquiel

Luis Darío Ortiz Izquierdo

28 de Septiembre, 2022

Introducción

Desde un punto de vista formal, la **concurrencia** no se refiere a dos o más eventos que ocurren a la vez sino a dos o más eventos cuyo orden es **no determinista**, esto es, eventos acerca de los cuales no se puede predecir el orden relativo en que ocurrirán. La concurrencia abarca varios aspectos, entre los cuales están la comunicación entre procesos y la compartición de, o competencia por recursos, la **sincronización** de actividades de múltiples procesos y la reserva de tiempo de procesador para los procesos.

No obstante, cuando múltiples procesos o hilos leen y escriben datos de manera que el resultado final depende del orden de ejecución de las instrucciones en los múltiples procesos, se le conoce como **condición de carrera**. El lugar en dónde se puede dar una condición de carrera el cual es una sección de código dentro de un proceso que requiere acceso a recursos compartidos y que no puede ser ejecutada mientras otro proceso esté en una sección de código correspondiente se le conoce como **sección crítica**. Por lo tanto, al proceso de evitar que múltiples procesos o hilos accedan a una sección crítica al mismo tiempo lo identificamos como **exclusión mutua**.

Existen diversos **mecanismos de sincronización** que permiten forzar a un proceso o hilo a detener su ejecución hasta que ocurra un evento en otro proceso y en esta presentación les introduciremos uno de estos mecanismos de sincronización que son los **semáforos**.

Soluciones del sistema

¿Qué soluciones nos brinda el sistema operativo para resolver los problemas de sincronización de procesos y por lo tanto poder resolver el problema de lograr la exclusión mutua para una sección crítica?

La primera respuesta que daremos serán los **semáforos**.

Definición

Herramienta de sincronización que provee el sistema operativo que no requiere espera ocupada. De esta manera un semáforo ***S*** es una variable que, aparte de la inicialización, sólo se puede acceder por medio de 2 operaciones *atómicas* y *mutuamente exclusivas*.

- Wait(S)
    - P(s), Down(s) —-----> **P**erate
- Signal(S)
    - V(s), Up(s), Post(s) o Release(s) —------> **V**ete

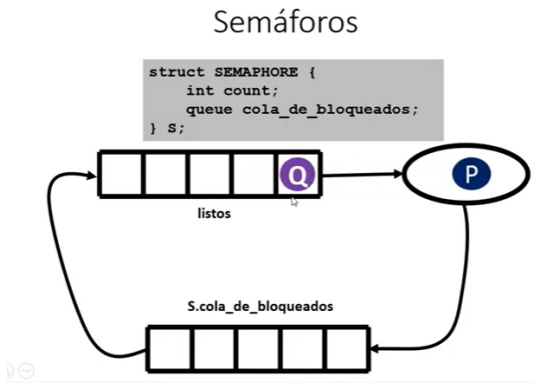
Para evitar la espera ocupada lo que hace un semáforo es que cuando un proceso tiene que esperar, se pondrá en una cola de procesos bloqueados esperando un evento.

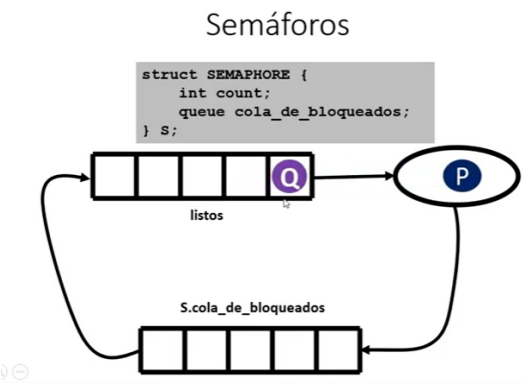
Estructura

Un tipo de dato semáforo consiste de un contador y de una cola que almacena los procesos, los pasos que sigue un semáforo (es decir, el algoritmo a *grosso modo*) se describe a continuación:

1.- Tenemos dos procesos que están listos para ejecutarse.
2.- El proceso P pasa a ejecución.

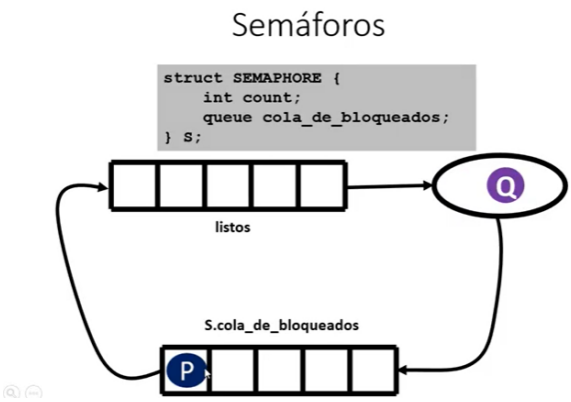
3.- El proceso P hace una llamada wait(S).
¿Qué pasa cuando P hace la llamada wait(S)?
Lo que va a suceder es que si el proceso debe de esperar en el semáforo se bloquea y se pone en la cola del semáforo S. De esta manera P ya está bloqueado y el proceso Q puede pasar a ejecución.

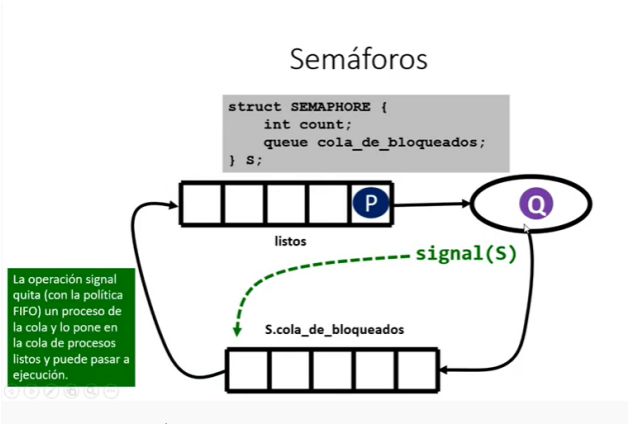




¿Cuánto tiempo permanecerá el proceso P bloqueado?

Hasta que otro proceso (por ejemplo el proceso Q) ejecute una llamada al sistema Signal(S) al semáforo, en ese momento cuando se hace un signal(s) a ese semáforo esta operación quita con la política FIFO (Primero en llegar primero en salir) un proceso de la cola del semáforo y lo pone en la cola de procesos listos y puede pasar a ejecución.





Tipos de semáforos

Semáforos binarios:

- Solo puede tener dos valores, 0 y 1.
- En windows se llaman mutex.

Semáforos binarios

Primero veremos como se forma un semáforo, es decir cual es el tipo de dato que se requiere para un semáforo, aquí lo veremos como una estructura que tiene dos campos; principalmente un valor que en este caso como son semáforos binarios solamente tomará valores de 0 a 1, después tiene una cola de bloqueados, que es la cola que es donde se van a bloquear los procesos cuando no puedan entrar a una sección crítica es decir cuando tengan que esperar en un semáforo.

```
struct SEMAPHORE {
    int valor; (0,1)
    queue cola_de_bloqueados;
} s;
```

Operaciones que vamos a utilizar con los semáforos binarios:

- **Wait(S):** Sii el campo valor del semáforo *S* es 1, este campo cambia a 0, si no lo que se tiene que hacer es poner el proceso que está invocando la función en *s.cola_de_bloqueado* es decir si el semáforo no es 1 el proceso se tiene que bloquear primero yendo a la cola de bloqueados y después bloqueándose.

```
WaitB(s):
  if s.valor=1
    s.valor=0
  else {
    poner este proceso en s.cola_de_bloqueados;
    bloquear este proceso
  };
```

- **Signal(S):** Lo que va a hacer es primero examinar si la cola del proceso de semáforos está vacía, es decir *s.cola_de_bloqueados* está vacía, es decir no hay nadie esperando, entonces lo que pasará es que pondremos el valor del semáforo.

  Y si no está vacía lo que vamos a hacer es vamos a desbloquear uno de los procesos que están bloqueados en la cola del semáforo, es decir en *s.cola_de_bloqueados.*

```
SignalB(s):
  If s.cola_de_bloqueados está vacía
    s.valor=1
  else {
    quitar un proceso P de s.cola_de_bloqueados;
    poner el proceso P en la cola de listos
  };
```

Nota importante: Para que estas primitivas funcionen es que deben de ejecutarse sin alguna división de sus instrucciones, es decir tanto Wait(S) como Signal(S) se deben de poder ejecutar como si fueran una sola instrucción (es decir que sean atómicas y mutuamente exclusivas), por ejemplo mientras un proceso está ejecutando Wait(S) no se debe permitir que otro proceso esté ejecutando Signal(S).

Semáforos generales o enteros

Son aquellos que pueden tomar muchos valores positivos o incluso valores negativos. Primero notemos que la estructura del semáforo es muy parecida a la presentada en los semáforos binarios, el cambio es que ahora en vez de un valor tenemos un contador, un contador que puede tomar muchos valores positivos e incluso negativos.

```
Wait(s):
  s.contador--;
  if s.contador<0 then
  {
    poner este proceso en s.cola_de_bloqueados;
    bloquear este proceso
  }
```

```
Signal(s):
  s.contador++;
  if s.contador <=0
  {
    quitar un proceso P de s.cola_de_bloqueados;
    poner el proceso P en la cola de listos
  }
```

Aquí tenemos la definición de la primitiva Wait y la definición de la primitiva Signal:

- En el caso de Wait(s) es que cuando un proceso quiera entrar a la sección lo primero que va a ocurrir es decrementar el contador en 1 y si después de hacer la resta ese contador es negativo entonces ese proceso se va a poner en la cola de bloqueados de ese semáforo (es decir el proceso se va a bloquear).

- Signal(s) hará incrementará el contador y si después de incrementar el contador tiene un valor menor o igual a cero quiere decir que hay procesos bloqueados esperando en la cola de bloqueados, por lo tanto lo que va a hacer es quitar un proceso de la cola de bloqueados de ese semáforo y lo va a poner en la cola de listos.

- De la misma manera que en el caso de los semáforos binarios, para que Wait y Signal funcionen correctamente deben de ser atómicas y mutuamente exclusivas, es decir se tiene que ejecutar como si fueran una sola operación.

- Si el contador es mayor o igual a cero, el número de procesos que pueden ejecutar Wait(S) sin que se bloqueen es igual al contador.

- Si el contador es menor que cero, el número de procesos que están esperando en el semáforo es el valor absoluto del contador. Por ejemplo, si el contador es -3 lo que significa es que tenemos tres procesos que están bloqueados en el semáforo.

Problema de los borrachos con semáforos binarios

Lo que sucede ahora es que cuando un borracho quiera entrar a un baño va a hacer una llamada Wait(s) y dentro de esta función lo que va a comprobar es si el valor del semáforo es 1 (si es 1 lo que quiere decir es que el baño está libre y por lo tanto puede pasar). Entonces el otro que quiere entrar al baño se va a preguntar ¿es *S.valor* = 1?, en este caso no es igual a uno porque ya pensamos que uno de los borrachos está en el baño, así que el borracho se tiene que ir a la cola y ahí se va a dormir hasta que el borracho que se encuentra en el baño salga.

Lo siguiente que sucederá es que cuando el borracho que está dentro del baño salga, lo primero que va a checar es que ver si la cola del baño está vacía, si esa cola no está vacía, tiene que despertar al borracho que está hasta enfrente de la cola, es decir primero tiene que salir y despues mandarle una señal al que está enfrente de la cola que le diga que ya puede pasar.

El problema de los borrachos con semáforos enteros (Utilizando valores negativos).

Iniciamos el problema de los borrachos que ya conocemos, inicia con un solo baño y en este se tendrá un contador el cual iniciará en 1. Lo primero que ocurrirá es que un borracho necesitará acceder al baño, por lo que lo primero que hará es decrementar el contador en 1. Ahora el borracho se preguntará ¿es *s.count* < 0? al no serlo por ser el primero que requiere ir al baño puede entrar. Posteriormente el otro borracho que quiere entrar ejecuta Wait y lo primero que hace es decrementar *s.count* en 1 (es decir el valor actual del contador es -1), ahora se pregunta ¿El contador es menor que cero? Al ser menor que cero el borracho se tiene que ir a la cola, es decir se tiene que bloquear esperando a que alguien salga del baño y le avise que ya puede pasar. (es bueno destacar que al tener el contador como -1 tendremos que solo un borracho está esperando en la cola). Cuando el borracho que está en el baño salga, lo primero que va a hacer será incrementar el contador en 1, una vez que está afuera será checar si el contador tiene un valor menor o igual a 0, si es así lo que hará el borracho es mandarle una señal al primer borracho de la cola para despertarlo y decirle que ya puede pasar (es decir aplica la operación Signal)

Semáforos enteros positivos

La diferencia de los semáforos anteriores es que ahora estos no van a tomar valores negativos. Ahora se va a formar de dos contadores, el primero será el contador normal y el segundo va a ser de bloqueados, así mismo también estará formado por la cola de bloqueados.

```
struct SEMAPHORE {
    unsigned int contador;
    unsigned int bloqueados;
    queue cola_de_bloqueados;
} s;
```

```
Wait(s):
  if s.contador==0 then
  {
    s.bloqueados++;
    poner este proceso en s.cola_de_bloqueados;
    bloquear este proceso;
  }
  else
    s.contador--;
```

```
Signal(s):
 if s.bloqueados==0 then
     s.contador++;
  else
 {
    quitar un proceso P de s.cola_de_bloqueados;
    poner el proceso P en la cola de listos
    s.bloqueados--;
 }
```

Ahora lo primero que se ejecutará en **Wait** será que si el contador es cero entonces incrementará el contador de bloqueados en 1 (es decir, indicará que hay un bloqueado más) y pondrá al proceso en la cola de bloqueados. Si el contador no es igual a cero entonces el contador se decrementa en 1.

Por otro lado para la primitiva **Signal** signal tendremos que se empezará con la comprobación de ver si el contador de bloqueados es igual a 0, si si lo es entonces lo único que va a hacer es aumentar el contador en 1. En el caso en que el contador de bloqueados no sea 0 lo que se realizará entonces será quitar el proceso de la cola de bloqueados y poner el proceso en la cola de listos y de esta manera se pueda volver a ejecutar y decrementamos el contador de bloqueados en 1.

De la misma forma que todos los semáforos anteriores tendremos que Wait y Signal tienen que ser atómicas y mutuamente exclusivas para que estas funcionen correctamente.El contador es el número de procesos que pueden ejecutar **Wait** sin que se bloqueen. Y en este caso el contador de bloqueados es el número de procesos que están esperando en el semáforo.

Pseudocódigo

```
main()
{
      cobegin {
            P(0);P(1);P(2)… P(N);
      }
}
```

```
Semaphore S;

Process P(int i)
{
      while(1)
      {
            wait(S);
            CS
            signal(S);
            RS
      }
}
```

Pasos

P(0)	P(1)	P(2)	S.count
			2
wait(S);			1
	wait(S);		0
		wait(S);	-1
CS			
	CS		
signal(S);			0
		CS	
	signal(S);		1
		signal(S);	2

Ejemplo (*k* procesos)

Podemos ejecutar *k* procesos en una cierta sección crítica con la ayuda de semáforos, pero ¿esto para que me puede servir?.

Imaginemos la siguiente situación:

Vamos a suponer que yo tengo un ancho de banda limitado a 100 MB, y yo tengo procesos que para que se ejecuten bien necesitan 50 MB. Si tenemos *k* procesos en ejecución, ¿qué necesito hacer para asegurar que todos los procesos que se van a ejecutar tendrán sus 50 MB? Lo que necesito hacer es limitar la cantidad de procesos, en este caso necesitaría limitar a que dos procesos están ocupando la sección crítica.

Nuestra solución es usar un semáforo entero que inicializamos con el valor de 2, con esto permitimos que dos procesos compartan la sección crítica al mismo tiempo

Ejemplo 2

Tenemos un restaurante donde existen *cocineros* y *meseros* que modelamos con procesos, estos inician concurrentemente.

Tenemos los procesos de cocinero() y mesero() que inician concurrentemente. Y los pasos serán:

- El cocinero preparará la comida.
- El mesero servirá la comida.

```
Process cocinero()
{
    preparar_comida();

}
```

```
Process mesero()
{

    servir_comida();
}
```

Lo que haremos será que los procesos iniciarán concurrentemente es decir al mismo tiempo arrancan los dos, es decir el *mesero* puede ir a *servir* la comida mientras el *cocinero*, *cocina* la comida, pero hay un detalle: el mesero no debe repartir la comida hasta que ésta no esté preparada.

Entonces necesitamos:

- Preparar_comida() en cocinero() sea ejecutado antes que servir_comida() en mesero().

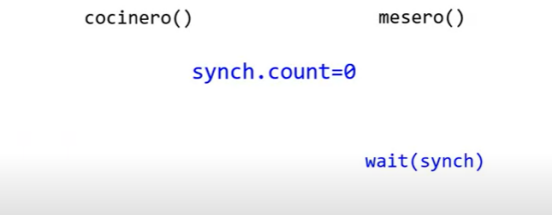
¿Cómo podemos hacer esto con semáforos?

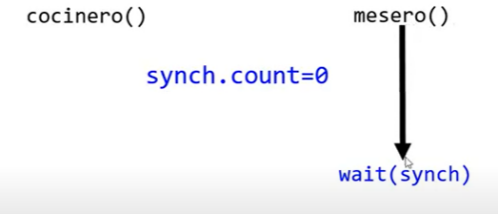
En el ***main*** del programa, lo que haremos es que antes de que inicien el *cocinero* y el *mesero* inicializamos el semáforo *synch* con valor de 0 y lo que hacemos es añadir la señal a los procesos:

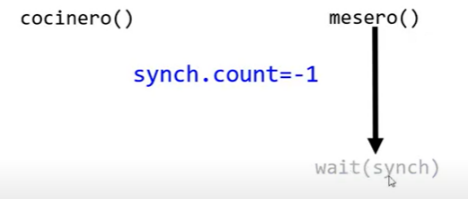
```
Process cocinero()
{
    preparar_comida();
    signal(synch);
}
```

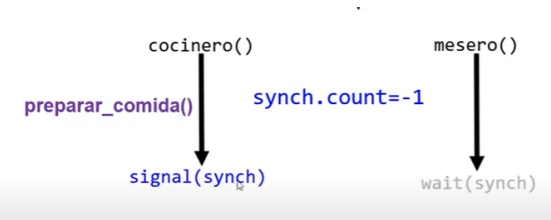
```
Process mesero()
{
    wait(synch);
    servir_comida();
}
```

Ahora tenemos que el *mesero* va a esperar en el semáforo *synch* y el *cocinero* cuando termine de preparar la comida va a hacer un signal al semáforo synch.









Glosario

- **Concurrencia**: Eventos cuyo orden de ejecución es no determinista

- **Sincronización:** es un método de administración de los procesos que garantiza la exclusión mutua en la concurrencia.

- **Condición de carrera:** Categoría de errores de programación que involucra a dos procesos que fallan al comunicarse su estado mutuo, llevando a resultados inconsistentes. Es uno de los problemas más frecuentes y difíciles de depurar, y ocurre típicamente por no considerar la no atomicidad de una operación.

- **Operación Atómica:** Manipulación de datos que requiere la garantía de que se ejecutará como una sóla unidad de ejecución, o fallará completamente, sin resultados o estados parciales observables por otros procesos o el entorno. Esto no necesariamente implica que el sistema no retirará el flujo de ejecución en medio de la operación, sino que el efecto de que se le retire el flujo no llevará a un comportamiento inconsistente.

- **Sección Crítica:** El área de código que requiere ser protegida de accesos simultáneos donde se realiza la modificación de datos compartidos.

- **Recurso compartido:** Es un recurso al cual se puede tener acceso desde mas de un proceso. En muchos casos esto es una variable en memoria pero podría ser archivos, periféricos, etc.

Conclusion

En conclusión los semáforos limitan a los procesos para que accedan uno a la vez a una sección crítica . Tienen como ventaja que siguen el principio de exclusión mutua y son mucho más eficientes que algunos otros métodos de sincronización. Sin embargo los semáforos son complicados de implementar, debido a que las primitivas Wait y Signal deben ser implementadas en el órden correcto para prevenir *deadlocks,* además podemos caer en un problema de priorización debido a que procesos de menor prioridad pueden acceder antes a la sección crítica que otros procesos de mayor prioridad.