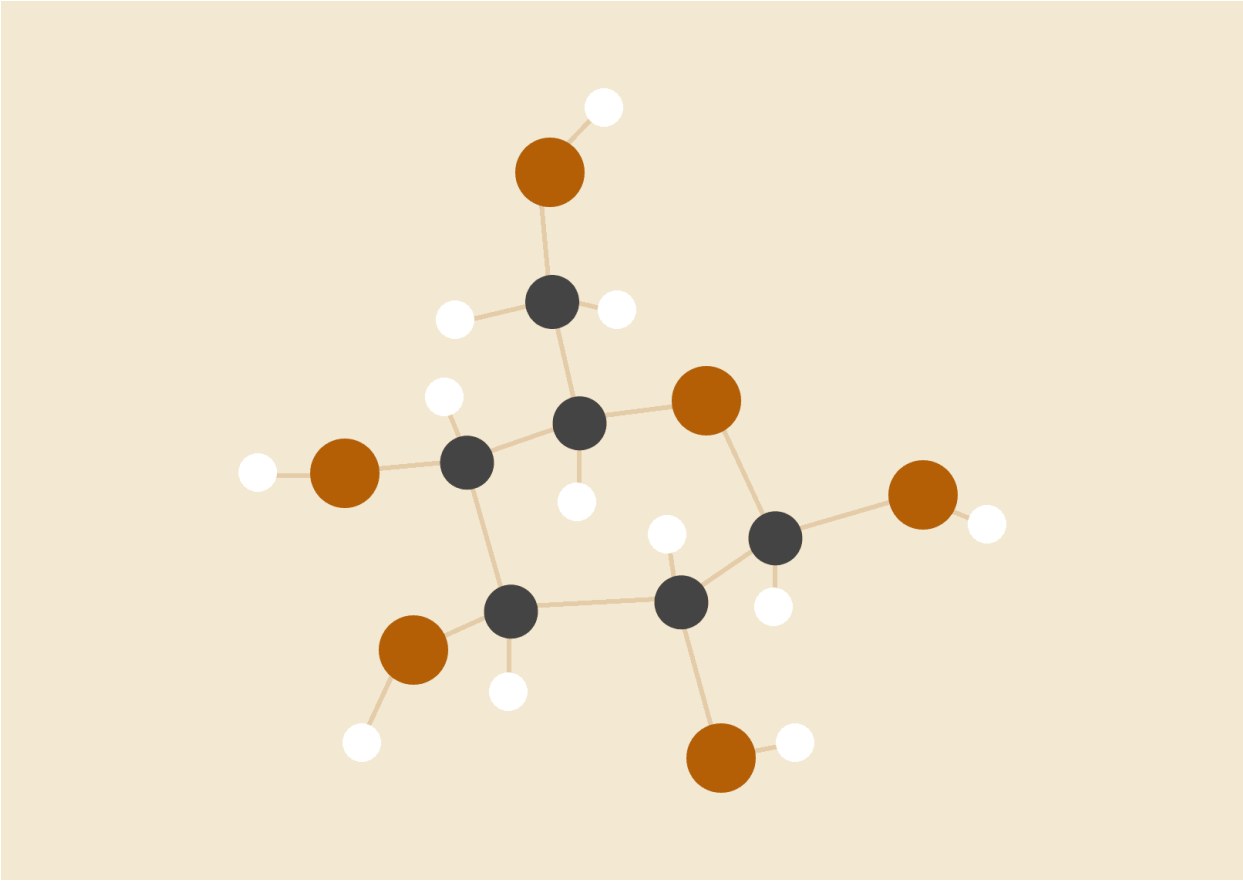
MONITORES



Barajas Ruíz Leslie

Benites Onofre Fernando

Chaparro Sicardo Tanibeth

Lerín Hernández Natalia

Cómputo Concurrente

INTRODUCCIÓN

Un proceso cooperativo es aquel que puede afectar o verse afectado por otros procesos que estén ejecutándose en el sistema. Estos procesos pueden compartir directamente un espacio de direcciones lógico o compartir los datos sólo a través de archivos o mensajes. El acceso concurrente a datos compartidos puede ocasionar incoherencia de los datos, por lo que se necesita hacer uso de mecanismos que aseguren la ejecución ordenada de los procesos cooperativos.

Sección crítica

Consideremos un sistema que consta de *n* procesos {P_0, P_1, …, P_{n-1}}. Cada proceso tiene un segmento de código llamado **sección crítica**, en el cual el proceso puede modificar las variables compartidas. Es importante que dos procesos del sistema no ejecuten su sección crítica simultáneamente, por lo que se debe diseñar un protocolo para que los procesos cooperen de forma correcta.

Cualquier solución al problema de la sección crítica debe satisfacer los siguientes requisitos:

- **Exclusión mutua.** Si el proceso P_i está ejecutando su sección crítica, los demás procesos no pueden estar ejecutando sus secciones críticas.
- **Progreso.** Si ningún proceso está ejecutando su sección crítica y algunos procesos desean entrar a sus correspondientes secciones críticas, sólo aquellos procesos que no estén ejecutando sus secciones restantes pueden participar en la decisión de cuál será el siguiente que entre en sección crítica, esta selección no se puede posponer indefinidamente.
- **Espera limitada**. Existe un límite en el número de veces que se permite que otros procesos entren en sus secciones críticas después de que un proceso haya hecho una solicitud para entrar en su sección crítica y antes de que la misma haya sido concedida.

MONITORES

El problema de los semáforos.

Los semáforos pueden ser un mecanismo adecuado de sincronización, pero su uso incorrecto puede ocasionar errores de temporización difíciles de detectar, pues estos sólo ocurren si se producen secuencias de ejecución concretas y esas secuencias no siempre se producen.

Revisemos el problema de la sección crítica con semáforos: En esta solución todos los procesos comparten la variable `mutex`, inicializada en 1. Cada proceso ejecuta la operación `wait(mutex)` antes de entrar a la sección crítica y ejecutar `signal(mutex)` después de esta. Si esta secuencia no se lleva a cabo, dos procesos podrían estar en la sección crítica al mismo tiempo.

Supongamos que un proceso intercambia el orden de ejecución de las operaciones `wait()` y `signal()`, es decir:

```
signal(mutex);
    . . .
    sección crítica
    . . .
wait(mutex);
```

En esta situación, varios procesos pueden ejecutar sus secciones críticas simultáneamente, violando el requisito de exclusión mutua. Sin embargo, el error sólo puede producirse si varios procesos están activos simultáneamente en sus secciones críticas y esto no siempre se produce.

Ahora supongamos que se reemplaza a `signal(mutex)` por `wait(mutex)`. Esto producirá un interbloqueo.

```
wait(mutex);
    . . .
    sección crítica
    . . .
wait(mutex);
```

Supongamos que se omite la operación `wait(mutex)` o `signal(mutex)` o ambas. Esto violaría la exclusión mutua o produciría un interbloqueo.

Para abordar este tipo de errores, se han desarrollado estructuras de lenguaje de alto nivel. Una de estas estructuras de sincronización de alto nivel es el tipo monitor.

Un tipo monitor tiene un conjunto de operaciones definidas por el programador que

tienen la característica de exclusión mutua dentro del monitor. El tipo monitor también contiene la declaración de variables cuyos valores definen el estado de una instancia de dicho tipo, junto con los cuerpos de los procedimientos o funciones que operan sobre dichas variables. Un procedimiento definido dentro del monitor sólo puede acceder a las variables declaradas dentro del monitor y sus parámetros formales. Las variables dentro del monitor solo pueden acceder a los procedimientos locales.

```
monitor nombre del monitor
{
    // declaraciones de variables compartidas
    procedimiento P1 ( . . . )  {
       . . .
    }
    procedimiento P2 ( . . . )  {
       . . .
    }
         .
         .
         .
    procedimiento Pn ( . . . )  {
       . . .
    }
    código de inicialización ( . . . )  {
       . . .
    }
}
```

La estructura del monitor asegura que solo un proceso esté activo dentro de él, por lo que no es necesario codificar explícitamente esta restricción de sincronización. También es necesario definir mecanismos de sincronización adicionales, proporcionados por la estructura `condition`. Se pueden definir una o más variables de tipo `condition`, y las únicas operaciones que se pueden invocar en una variable de condición son `wait()` y `signal()`.

```
condition x, y;
```

La operación `x.wait();` indica que el proceso que invoca esta operación queda suspendido hasta que otro proceso invoque la operación `x.signal();` que hace que reanude exactamente uno de los procesos suspendidos. Si no hay un proceso suspendido, entonces `signal()` no tiene efecto.

En caso de haber varios procesos suspendidos en la condición `x` y se ejecuta `x.signal()`, una de las soluciones es usar el orden FCFS, haciendo que el proceso que

lleve más tiempo en espera se reanude primero. Puede usarse también la estructura de espera condicional, que tiene la forma

```
x.wait(c);
```

donde `c` es una expresión entera que se evalúa cuando se ejecuta la operación `wait()`. A `c` se le denomina número de prioridad, se almacena junto con el nombre del proceso suspendido. Al ejecutarse `x.signal()`, se reanuda el proceso que tenga asociado el número de prioridad más bajo.

Los monitores son estructuras de un lenguaje de programación que ofrecen una funcionalidad equivalente a la de los semáforos, pero son más fáciles de controlar. Constan de uno o más procedimientos, una secuencia de inicialización y datos locales.

Sus características son:

- Las variables locales sólo son accesibles para los procedimientos dentro del monitor. Podemos proteger los datos compartidos situándolos dentro del monitor.
- Un proceso entra al monitor invocando uno de los procedimientos.
- Solo un proceso puede ejecutarse en el monitor en un instante dado. Cualquier otro objeto que invoque al monitor queda suspendido mientras espera que esté disponible el monitor. Si esta norma se cumple, obtenemos exclusión mutua.

Monitores con señales

Un monitor proporciona sincronización por medio de las variables de condición incluidas del monitor, y como se ha mencionado anteriormente, estas solo son accesibles para el proceso dentro del monitor. Existen dos funciones que operan con las variables de condición:

- `wait(c)`: Suspende la ejecución del proceso bajo la condición c. El monitor puede ser usado por otro proceso.
- `signal(c)`: Reanuda la ejecución de algún proceso suspendido con un wair bajo la misma condición c. En caso de haber varios procesos, elige uno de ellos, si no hay ninguno, no hace nada.

Las operaciones de `wait` y `signal` son diferentes a aquellas de semáforos. Notemos que si un proceso dentro del monitor ejecuta signal y no hay tareas en la cola de espera de la variable de condición, el `signal` se pierde.

Como se ha mencionado anteriormente, los procesos pueden entrar al monitor llamando a cualquiera de los procedimientos, y puede considerarse que el monitor tiene un único punto de entrada que solo permite que un proceso se encuentre dentro de él en cada instante. Otros procesos que intenten entrar al monitor se añadirán a una cola de procesos suspendidos donde esperan a que el monitor esté disponible. Cuando un proceso se encuentra dentro del monitor, puede suspenderse temporalmente bajo la condición `x` ejecutando `wait(x)`; de esta manera se sitúa en una cola de procesos que esperan volver a entrar al monitor cuando la condición x cambie.

Si un proceso que se ejecuta dentro del monitor detecta un cambio en la condición `x`, ejecuta `signal(x)`, avisando a la cola de condición correspondiente que la condición ha cambiado.

Problema del productor/consumidor con monitores con señales

Problemática. Uno o más productores generan cierto tipo de datos (caracteres) y los sitúan en un buffer. Un único consumidor saca elementos del buffer de uno en uno. El sistema debe impedir la superposición de operaciones sobre el buffer. Es decir, el añadir o tomar elementos del buffer es una sección crítica. Debemos asegurar que si el productor está añadiendo un elemento al buffer, el consumidor no debe tener acceso al buffer.

```
buffer_acotado buffer=new buffer_acotado();
```

```
Productor()
{
    char x;
    while(forever)
    {
        producir(&x);
        buffer.añadir(x);
    }
}
```

```
Consumidor()
{
    char x;
    while(forever)
    {
        buffer.tomar(&x);
        consumir(x);
    }
}
```

```
main()
{
    cobegin {
        Productor();
        Consumidor();
    }
}
```

```
monitor buffer_acotado
{
        char buffer[TAM_BUFFER];        //Espacio para N elementos
        int sigent, sigsal;             //Apuntadores al buffer
        int contador;                   // Número de elementos del buffer
        condition no_lleno, no_vacio;   //Para la sincronización

        añadir(char x) {
            if (contador == TAM_BUFFER) cwait(no_lleno);
                                        //Buffer lleno; se impide producir
            buffer[sigent]=x;
            sigent = sigent + 1 % TAM_BUFFER;
            contador++;                 //Un elemento más en el buffer
            csignal(no vacio);          //Reanudar un consumidor en espera
        }
```

```
        tomar(char x)
        {
            if (contador == 0) cwait(no_vacio);
                                //Buffer vacio; se impide consumir
            x=buffer[sigsal];
            sigsal=(sigsal+1) % TAM_BUFFER;
            contador--;
                        //Un elemento menos en el buffer
            csignal(no_lleno);    //Reanudar un productor en espera
        }
    initialize
    {       //Cuerpo del monitor
            sigent=0;sigsal=0;contador=0;    //Buffer inicia vacio
    }
} // Termina el monitor
```

El monitor `buffer_acotado` controla el buffer empleado para almacenar y retirar caracteres. Las variables de condición son:

- **`no_lleno`.** Cierta cuando hay sitio para añadir al menos un carácter al buffer
- **`no_vacío`.** Cierta cuando hay al menos un carácter en el buffer.

Un productor sólo puede añadir caracteres al buffer con el procedimiento `añadir` del monitor, no tiene un acceso directo al buffer. El procedimiento `añadir` comprueba la condición `no_lleno` para determinar si hay espacio libre en el buffer. Existen dos casos:

1. No hay espacio en el buffer: El proceso que se está ejecutando se suspende y pasa a la cola de condición `no_lleno`. En ese momento, cualquier otro proceso, ya sea productor o consumidor, puede entrar al monitor. Cuando el buffer ya no esté lleno el proceso suspendido puede ser retirado de la cola y continuar con su tarea.
2. Hay espacio en el buffer: El productor añade un carácter en el buffer y activa la condición `no_vacío`.

La tarea del consumidor es similar a la descripción anterior.

Monitores con notificación

En la definición anterior de monitores se exige que si hay al menos un proceso en una cola de condición, un proceso de la cola deberá ejecutarse en cuanto otro proceso ejecute `signal` para dicha condición, por lo que el proceso que ejecuta el `signal` debe salir del

monitor inmediatamente o suspenderse. Esto puede dar lugar a inconvenientes:

1. Si el proceso que ejecuta `signal` no abandona el monitor hacen falta cambios de contexto: uno que suspenda el proceso y otro para reanudarlo cuando quede disponible.
2. Cuando se ejecuta `signal` debe activarse inmediatamente un proceso de la cola de condición correspondiente y el planificador debe asegurarse de que ningún otro proceso entra al monitor antes de la activación del proceso en espera, pues un proceso nuevo puede cambiar la condición bajo la cual se ha activado el proceso en espera.

En los monitores con notificación, la primitiva `signal` es reemplazada con `notify`. Cuando un proceso dentro del monitor ejecuta `notify`, origina una notificación a la cola de condición `x`, pero el mismo proceso que ejecuta la notificación puede continuar ejecutándose. La notificación solo nos asegura que el proceso de la cabeza de la cola de notificación se reanudará en el futuro cercano cuando el monitor esté disponible. Sin embargo, como no garantiza que ningún otro proceso entre al monitor antes que el proceso en espera, el proceso en espera debe volver a comprobar la condición `x`.

Problema del productor/consumidor con monitores con notificación

Tomando en cuenta lo anterior, el código del buffer acotado cambia a:

```
monitor buffer_acotado
{
        char buffer[TAM_BUFFER];        //Espacio para N elementos
        int sigent, sigsal;               //Apuntadores al buffer
        int contador;                     // Número de elementos del buffer
        condition no_lleno, no_vacio;     //Para la sincronización

        añadir(char x) {
                while (contador == TAM_BUFFER) cwait(no_lleno);
                                        //Buffer lleno; se impide producir
                buffer[sigent]=x;
                sigent = sigent +  1 % TAM_BUFFER;
                contador++;             //Un elemento más en el buffer
                cnotify(no_vacio);      //Envía notificación a consumidor en espera
        }
```

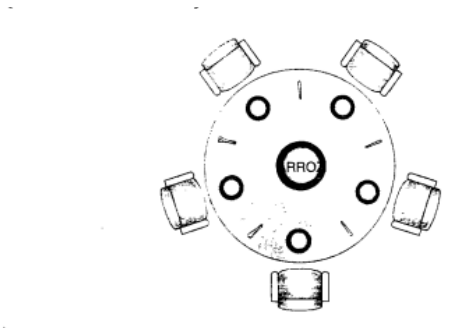
```
        tomar(char x)
        {
            while (contador == 0) cwait(no_vacio);
                                    //Buffer vacio; se impide consumir
            x=buffer[sigsal];
            sigsal=(sigsal+1) % TAM_BUFFER;
            contador--;
                            //Un elemento menos en el buffer
            cnotify(no_lleno);    //Envía notificación a un productor en espera
        }
    initialize
    {           //Cuerpo del monitor
            sigent=0;sigsal=0;contador=0;    //Buffer inicia vacio
    }
} // Termina el monitor
```

Además, la sentencia **`if`** fue reemplazada por un bucle **`while`**. Así generamos al menos una evaluación extra de la variable de condición.

Este método es menos propenso a errores: Un proceso realice una notificación incorrecta no generará un error en el programa pues, si después de la comprobación de la variable la condición deseada no se cumple, el proceso en la cola de condición continuará esperando.

CENA DE LOS FILÓSOFOS: SOLUCIÓN CON MONITORES

Consideremos a 5 filósofos que gastan sus vidas en pensar y comer. Los filósofos comparten una mesa redonda con 5 sillas, una para cada uno. En el centro de la mesa hay arroz y solo se han puesto 5 palillos en la mesa. Cuando un filósofo piensa, no se relaciona con los demás. De vez en cuando tiene hambre y trata de tomar los palillos que se encuentran entre él y sus vecinos de la izquierda y la derecha. Un filósofo puede tomar un palillo a la vez. Cuando un filósofo consigue dos palillos, come sin soltarlos. Cuando termina de comer, los coloca de nuevo en la mesa y vuelve a pensar.



Introducimos la estructura de datos siguiente para diferenciar entre los tres estados en los que se puede hallar un filósofo.

```
enum {pensar, hambre, comer} state[5];
```

El filósofo `i` puede configurar la variable `state[ i ] = comer` sólo si sus vecinos no están comiendo: `(state[(i+4)%5] != comer) y (state[(i+1)%5] != comer)`. También se declara

```
condition self[5];
```

donde el filósofo `i` tiene que esperar cuando tiene hambre pero no puede conseguir los palillos.

La distribución de los palillos se controla mediante el monitor dp. Antes de empezar a comer, cada filósofo invoca la operación `pickup()` , la cual puede dar lugar a la suspensión del proceso filósofo. Después de completar esta operación, el filósofo puede comer. A continuación el filósofo invoca la operación `putdown()`. El filósofo `i` debe invocar las operaciones anteriores en la secuencia:

```
dp.pickup(i);
    ...
    comer
    ...
dp.putdown(i);
```

Esta solución asegura que nunca dos vecinos estarán comiendo simultáneamente y que no se producirán interbloqueos. Sin embargo, es posible que un filósofo se muera de hambre.

```
monitor dp
{
    enum {PENSAR, HAMBRE, COMER}state[5];
    condition self[5];

    void pickup(int i) {
      state[i] = HAMBRE;
      test(i);
      if (state[i] != COMER)
         self[i].wait();
    }

    void putdown(int i) {
      state[i] = PENSAR;
      test((i + 4) % 5);
      test((i + 1) % 5);
    }

    void test(int i) {
      if ((state[(i + 4) % 5] != COMER) &&
          (state[i] == HAMBRE) &&
          (state[(i + 1) % 5] != COMER)) {
             state[i] = COMER;
             self[i].signal();
    }
}

    initialization code() {
      for (int i = 0; i < 5; i++)
        state[i] = PENSAR;
    }
}
```

REFERENCIAS

Silberschatz, Abraham, et. al. *Fundamentos de Sistemas Operativos.* 7ma edición. McGraw-Hill, 2005.

Stallings, William. *Sistemas Operativos*. 2da edición. Prentice Hall, 1997.