Universidad de Oriente

Núcleo de Anzoátegui

Escuela de Ingeniería y Cs. Aplicadas

Departamento de Computación y Sistemas

Cátedra: Sistemas Operativos

Sección: 01

Semáforos

Proyecto: Simulador Barbería Justa

Profesor:

Cortínez, Claudio

Integrantes:

Brazón, Eulises

Rodríguez, Eliana

Barcelona, junio de 2021.

INDICE

INTRODUCCIÓN

El problema de la Barbería es un problema de sincronización; consiste en una barbería en la que trabajan 3 barberos, 3 sillas de atención al cliente, 4 puestos en un sofá, un área de espera y una caja. El problema consiste en realizar la actividad del barbero sin que ocurran condiciones de carrera. La solución implica el uso de semáforos y objetos de exclusión mutua para proteger la sección crítica. Un semáforo es una variable protegida (o tipo abstracto de datos) que constituye el método clásico para restringir o permitir el acceso a recursos compartidos en un entorno de multiprocesamiento. Fueron inventados por Edsger Dijkstra y se usaron por primera vez en el sistema operativo THEOS. En electrónica y en programación concurrente, se conoce como condición carrera el error que se produce en programas o circuitos lógicos que no se han construido adecuadamente para su ejecución simultánea con otros procesos.

Proyecto: Simulador del Problema de la Barbería

Funcionalidades del Problema

El cuerpo del programa principal activa 50 clientes, 3 barberos y el proceso cajero. Ahora se considerará el propósito y la colocación de los diferentes operadores de sincronización:

- **Capacidad de la tienda y del sofá.** La capacidad de la tienda y la capacidad del sofá se gobiernan a través de los semáforos max_capacidad y sofá, respectivamente. Cada vez que un cliente intenta entrar en la tienda, el semáforo max_capacidad se decrementa en 1; cada vez que un cliente abandona la barbería, el semáforo se incrementa. Si un cliente encuentra la tienda llena, el proceso de dicho cliente se bloquea en max_capacidad mediante la función semWait. Análogamente, las operaciones semWait y semSignal están alrededor de las acciones de sentarse y levantarse del sofá.
- **Capacidad de la silla del barbero.** Hay tres sillas de barbero y hay que tener cuidado con utilizarlas apropiadamente. El semáforo silla_barbero asegura que no hay más de tres clientes intentando obtener servicio a la vez, intentando evitar la indigna situación de tener un cliente sentado en el regazo de otro. Un cliente no se levantará del sofá, a menos que una silla esté libre [semWait(silla_barbero)] y cada barbero señaliza la acción de que un cliente ha dejado su silla [semSignal(silla_barbero)]. Se asegura el acceso justo a las sillas de los barberos debido a la organización de cola del semáforo: el primer cliente que se bloquea será el primero que ocupe una silla disponible. Obsérvese que, en el procedimiento del cliente, si semWait (silla_barbero) ocurriese después de semSignal(sofá), cada cliente brevemente se sentaría en el sofá y permanecería de pie en la línea de las sillas de los barberos, creando congestión y dejando a los barberos poco espacio de maniobra.
- **Asegurar que los clientes están en la silla del barbero**. El semáforo cliente_listo envía una señal que despierta a un barbero que se encuentre durmiendo, indicándole que un cliente acaba de ocupar una silla. Sin este semáforo, un barbero nunca dormiría sino que empezaría a cortar el pelo tan pronto como un cliente dejara la silla; si no hubiese ningún cliente nuevo sentado, el barbero estaría cortando el aire.

- **Mantenimiento de los clientes en una silla de barbero**. Una vez sentado, un cliente permanece en la silla hasta que el barbero da la señal de que el corte ha finalizado, utilizando el semáforo terminado.
- **Limitar un cliente por silla de barbero**. El semáforo silla_barbero se utiliza para limitar a tres el número de clientes que hay en las sillas de los barberos. Sin embargo, por sí mismo, el semáforo silla_barbero no realiza adecuadamente su misión. Un cliente que no obtiene el procesador inmediatamente después de que su barbero ejecuta semSignal (terminado) (es decir, que se cae o se para a hablar con un vecino) podría estar todavía en la silla cuando al próximo cliente se le permita sentarse. El semáforo dejar_silla_b se utiliza para corregir este problema, evitando que el barbero invite a un nuevo cliente a la silla hasta que el cliente no se haya ido efectivamente.
- **Pagos y recibos**. Naturalmente, hay que ser cuidadoso cuando se trata de dinero. El cajero debe asegurar que cada cliente paga antes de abandonar la tienda y el cliente quiere la verificación de que se ha recibido el pago (un recibo). Esto se realiza, en efecto, mediante una transferencia monetaria cara a cara. Cada cliente, justo después de levantarse de la silla del barbero, paga, avisando al cajero que se ha pasado el dinero [semSignal (pago)] y entonces espera por un recibo [semWait (recibo)]. El proceso cajero se dedica de forma repetida a recibir los pagos: espera a que se le señalice un pago, acepta el dinero y entonces señaliza la aceptación del dinero. Aquí se necesitan evitar varios errores de programación. Si semSignal (pago) ocurriera justo antes de la acción pagar, un cliente podría verse interrumpido después de tal señalización; esto dejaría libre al cajero para aceptar pagos incluso de alguien que no se lo hubiera ofrecido. Un error aún más serio sería invertir las posiciones de las sentencias semSignal (pago) y semWait (recibo). Esto llevaría a un interbloqueo ya que provocaría que todos los clientes y el cajero se bloquearan en sus respectivos operadores semWait.

Coordinación entre las funciones de barbero y de cajero. Para ahorrar dinero, esta barbería no emplea un cajero independiente. Cada barbero puede llevar a cabo esta tarea si no está cortando el pelo. El semáforo coord asegura que sólo un barbero lleva a cabo esta tarea en un momento determinado.

Semáforo	Operación *wait*	Operación *signal*
`max_capacidad`	El cliente espera a que haya espacio en la tienda.	Un cliente que abandona la tienda señaliza a un cliente que espera por entrar.
`sofa`	El cliente espera para sentarse en el sofá.	Un cliente que abandona el sofa señaliza a un cliente que espera por sentarse en el sofá.
`silla_barbero`	El cliente espera por una silla de barbero vacía.	El barbero señaliza cuando su silla está vacía.
`cliente_listo`	El barbero espera a que el cliente se siente en la silla.	El cliente señaliza al barbero para indicarle que está en la silla.
`terminado`	El cliente espera hasta que se completa su corte de pelo.	El barbero señaliza cuando ha finalizado de cortar el pelo a su cliente.
`dejar_silla_b`	El barbero espera hasta que el cliente se levanta de la silla.	El cliente señaliza al barbero cuando se ha levantado de la silla.
`pago`	El cajero espera a que el cliente pague.	El cliente señaliza al cajero que ha pagado.
`recibo`	El cliente espera un recibo de su pago.	El cajero señaliza que se ha aceptado el pago.
`coord`	Espera por un barbero que esté libre para llevar a cabo un corte de pelo o tareas de cajero.	Señaliza que un barbero está libre.

Figura 1. Funcionalidades de los semáforos del Programa

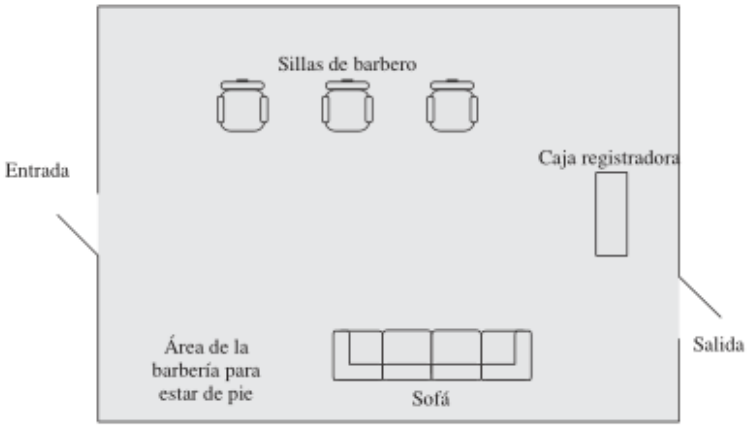

Figura 2. La barbería

```
/* programa barberia2 */
semaforo max_capacidad = 20;
semaforo sofa = 4;
semaforo silla_barbero = 3, coord = 3;
semaforo mutex1 = 1, mutex2 = 1;
semaforo cliente_listo = 0, dejar_silla_b = 0, pago = 0; recibo = 0;
semaforo terminado [50] = {0};
int count;

void cliente ()
{
    int numcliente;
    wait(max_capacidad)
    Entrar_tienda();
    wait(mutex1);
    numcliente = cuenta;
    cuenta++;
    signal(mutex1);
    wait(sofa);
    sentarse_en_sofa();
    wait(silla_barbero);
    levantarse_del_sofa();
    signal(sofa);
    sentarse_en_silla_de_barbero();
    wait(mutex2);
    encola1(numcliente);
    signal(cliente_listo);
    signal(mutex2);
    wait(terminado[numcliente]);
    dejar_silla_barbero();
    signal(dejar_silla_b);
    pagar();
    signal(pago);
    wait(recibo);
    salir_tienda();
    signal(max_capacidad)
}

void barbero ()
{
    int cliente_b
    while (true)
    {
        wait(cliente_listo);
        wait(mutex2);
        fcola1(cliente_b);
        signal(mutex2);
        wait(coord);
        cortar_pelo();
        signal(coord);
        signal(terminado[cliente_b]);
        wait(dejar_silla_b);
        signal(silla_barbero);
    }
}

void cajero ()
{
    while (true)
    {
        wait(pago);
        wait(coord);
        aceptar_pago();
        signal(coord);
        signal(recibo);
    }
}

void main ()
{   count :=0;
    parbegin (cliente,… 50 veces,… cliente, barbero, barbero, barbero, cajero);
}
```

Figura A6. Una barbería justa.

Se plantea dar solución al apartado A.6. Se mantienen varios problemas en el caso de la barbería justa. Modifique el problema para corregir los siguientes problemas.

Algoritmos Implementados.

a) El cajero podría aceptar el pago de un cliente y liberar a otro si dos o más clientes están esperando por pagar. Afortunadamente, una vez que un cliente presenta un pago, no hay forma para él de retroceder, así que al final, en la caja registradora se guarda la cantidad exacta. No obstante, es deseable liberar al cliente correcto tan pronto como haya realizado su pago.

```
1   /* programa barberia2 a */
2   semaforo max_capacidad = 20;
3   semaforo sofa = 4;
4   semaforo silla_barbero = 3, coord = 3;
5   semaforo mutex1 = 1, mutex2 = 1;
6   semaforo cliente_listo = 0, dejar_silla_b = 0,pago = 0, recibo = 0;
7
8   int cuenta_c=0;
9
10  semaforo mutex3 = 1,mutex4 = 1,mutex5 = 1, mutex6 = 1,mutex7 = 1;
11  boolean semaforoDecrementado = False
12
```

Se agregan nuevos mutex para asegurar la exclusion en los accesos a variables globales, al momento de inicializar los cajeros el primero cajero en ejecutarse decrementará el semaforo pago en -1, haciendo que sea necesario que dos cliente realicen el llamado a signal(pago), antes que el barbero pueda pasar a cobrar (minimo dos personas esperando), para controlar que solo se de cremente el semaforo pago en -1 y no en -3 se hace uso de mutex7 junto con un boolean semaforoDrecremantado que una vez que el primero proceso entra dentro del condicional de la linea 77 modifica ese boolean para que los procesoso posteriosres a el no decrementen el contador mas de los necesario. Es importante tener en cuanta que para que el ultimo cliente no se quede esperando por siempre en la caja, este debe realizar un signal adicional, haciendo que los ultimos clientes son procesaco cuando hay minimo una persona esperando, y que puedan liberarse el ultimo cliente.

```
13  void cliente()
14  {
15      int numcliente
16      wait(max_capacidad)
17      Entrar_tienda()
18      wait(mutex1)
19      numcliente=cuenta_c
20      cuenta_c++
21      signal(mutex1)
22      wait(sofa)
23      sentarse_en_sofa()
24      wait(silla_barbero)
25      levantarse_del_sofa()
26      signal(sofa)
27      sentarse_en_silla_de_barbero()
28      wait(mutex2)
29      encola1(numcliente)
30      signal(cliente_listo)
31      signal(mutex2)
32      wait((terminado[numcliente]))
33      dejar_silla_barbero()
34      signal(dejar_silla_b)
35      pagar()
36      wait(mutex4)
37      encola2(numcliente)
38      //si llego el ultimo cliente...
39      // es necesario incrementar
40      //el semaforo pago para que no
41      //quede esperando en la cola
42      //de pagar hasta el proximo dia
43      if(numcliente==49)
44          {
45              signal(pago)
46          }
47      signal(pago)
48      signal(mutex4)
49      wait(recibo[numcliente])
50      salir_tienda()
51      signal(max_capacidad)
52  }
```

```
53  void barbaero()
54  {
55      int cliente_b
56
57      int numBarbero;
58
59      while(True)
60      {
61          wait(cliente_listo)
62          wait(mutex2)
63          fcola1(cliente_b)
64          signal(mutex2)
65          wait(coord)
66          cortar_pelo()
67          signal(coord)
68          signal(terminado[cliente_b])
69          wait(dejar_silla_b)
70          signal(silla_barbero)
71      }
72  }
73  void cajero()
74  {
75      int cliente_b
76      wait(mutex7)
77      if(!semaforoDecrementado) {
78          wait(pago)
79          semaforoDecrementado= True }
80      signa(mutex7)
81
82      while(true)
83      {
84          wait(mutex6)
85          wait(pago)
86
87          wait(mutex5)
88          fcola2(cliente_b)
89          signal(mutex5)
90
91          wait(coord)
92          aceptar_pago()
93          signa(coord)
94
95          signal(recibo[cliente_b])
96          signal(mutex6)
97      }
98  }
```

b) El semáforo dejar_silla_b evita supuestamente que múltiples clientes accedan a una única silla de barbero. Desafortunadamente, este semáforo no realiza su tarea adecuadamente en todos los casos. Por ejemplo, supóngase que los tres barberos han cortado el pelo y se han quedado bloqueados en semWait(dejar_silla_b). Dos de los clientes se encuentran en un estado interrumpido justo antes de dejar la silla del barbero. El tercer cliente deja su silla y ejecuta semSignal (dejar_silla_b). ¿Qué barbero se libera? Debido a que la cola dejar_silla_b es FIFO, se libera el primer barbero que se bloqueó. ¿Es este barbero el que estaba cortando el pelo del cliente que señalizó? Tal vez, pero tal vez no. Si no es así, un nuevo cliente vendrá y se sentará en el regazo de otro cliente que estaba a punto de levantarse.

```
1   /* programa barberia2 b*/
2   semaforo max_capacidad = 20;
3   semaforo sofa = 4;
4   semaforo silla_barbero = 3, coord = 3;
5   semaforo mutex1 = 1, mutex2 = 1;
6   semaforo cliente_listo = 0, dejar_silla_b [3]= 0,pago = 0, recibo = 0;
7   semaforo terminado [50] = {0};
8
9   int cuenta=0, cuenta_b=0;
10  semaforo mutex3=1, mutex4 = 1, mutex5 = 1;
11
```

Se agregan nuevos mutex para asegurar la exclusion en los accesos a variables globales, se crea una nueva cuenta_b, para asignarle un numero unico a cada berbero al ser inicializados, se remplaza el semaforo dejar_silla_b por un arreglo de semaforo con 3 posiciones (una por cada barbero), de esta manera el cliente debe enviar un signal al barbero correcto (dejar_silla_b[numBarbero]) cuyo dato es cargado desde la cola2 (fcola2), que previamente es cargado en la funcion barbero (encola2)

```
void cliente()
{

    int numcliente
    int numBarbero
    wait(max_capacidad)
    Entrar_tienda()
    wait(mutex1)
    numcliente=cuenta
    cuenta++
    signal(mutex1)
    wait(sofa)
    sentarse_en_sofa()
    wait(silla_barbero)
    levantarse_del_sofa()
    signal(sofa)
    sentarse_en_silla_de_barbero()
    wait(mutex2)
    encola1(numcliente)
    signal(cliente_listo)
    signal(mutex2)
    wait((terminado[numcliente]))

    wait(mutex5)
    fcola2(numBarbero)
    signal(mutex5)

    dejar_silla_barbero()
    signal(dejar_silla_b[numBarbero])
    pagar()
    signal(pago)
    wait(recibo)
    salir_tienda()
    signal(max_capacidad)


}
```

```
void barbaero()
{
    int cliente_b

    int numBarbero;
    wait(mutex3)
    numBarbero=cuenta_b
    cuenta_b++
    signal(mutex3)

    while(True)
    {
        wait(cliente_listo)
        wait(mutex2)
        fcola1(cliente_b)
        signal(mutex2)
        wait(coord)
        cortar_pelo()
        signal(coord)

        wait(mutex4)
        encola2(numBarbero)
        signal(mutex4)

        signal(terminado[cliente_b])
        wait(dejar_silla_b[numBarbero])
        signal(silla_barbero)
    }
}
void cajero()
{

    while(true)
    {
        wait(pago)
        wait(coord)
        aceptar_pago()
        signa(coord)
        signal(recibo)
    }
}
```

c) El programa requiere que un cliente se siente primero en el sofá, incluso si la silla del barbero está vacía. Éste es un problema menor y resolver este problema implica hacer un código aún más complejo. No obstante, intente resolver este problema.

```
1   /* programa barberia2 c*/
2   semaforo max_capacidad = 20;
3   semaforo sofa = 4;
4   semaforo silla_barbero = 3, coord = 3;
5   semaforo mutex1 = 1, mutex2 = 1;
6   semaforo cliente_listo = 0, dejar_silla_b = 0,pago = 0, recibo = 0;
7   semaforo terminado [50] = {0};
8
9   semaforo mutex3 = 1;
```

Si hay un barbero disponible y no hay nadie esperando, no debería de ser necesario pasar por el sofá, el cliente puede ir directamente a sentarse en la silla_barbero, para esto se utilizó un mutex, dentro de él se evalúa si se cumple esta condición, y el resultado es almacenado en una variable, para posteriormente, ejecutar o no el bloque de código asociado con el proceso de sofá, es necesario almacenar este dato, porque de realizarse esta evaluación dos veces, (dos bloques de código asociados al proceso sofá), puede que arrojen resultados distintos en instantes de tiempo distintos generando interbloqueo.

```
void cliente()
{
    boolean condicion
    //si esta libre alguna silla
    //de barbaero y no hay personas esperando
    int numcliente

    wait(max_capacidad)
    Entrar_tienda()
    wait(mutex1)
    numcliente=cuenta
    cuenta++
    signal(mutex1)

    wait(mutex3)
    condicion = sofa.cuenta==4 && silla_barbero>0
    //guardo el valor de la ejecucion
    //en un momento determinado
    //(cuando el cliente entra)
    signal(mutex3)

    //si hay espacio libre
    //no es necesario pasar por el sofa
    if(condicion!)
    {
        wait(sofa)
        sentarse_en_sofa()
    }

    wait(silla_barbero)

    if(condicion!)
    {
        levantarse_del_sofa()
        signal(sofa)
    }

    sentarse_en_silla_de_barbero()
    wait(mutex2)
    encola1(numcliente)
    signal(cliente_listo)
    signal(mutex2)
    wait((terminado[numcliente]))
    dejar_silla_barbero()
    signal(dejar_silla_b)
    pagar()
    signal(pago)
    wait(recibo)
    salir_tienda()
    signal(max_capacidad)
}
```

```
void barbaero()
{
    int cliente_b

    while(True)
    {
        wait(cliente_listo)
        wait(mutex2)
        fcola1(cliente_b)
        signal(mutex2)
        wait(coord)
        cortar_pelo()
        signal(coord)
        signal(terminado[cliente_b])
        wait(dejar_silla_b)
        signal(silla_barbero)
    }
}
void cajero()
{

    while(true)
    {
        wait(pago)
        wait(coord)
        aceptar_pago()
        signa(coord)
        signal(recibo)
    }
}
```

CONCLUSIÓN

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 estas 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.

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.

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.

BIBLIOGRAFÍA

- Sistemas Operativos. Aspectos internos y principios de diseño. William Stallings. Pearson educación. 5ª edición. 2005. Fundamentos de sistemas operativos.

- Editor Visual Studio Code