2.5 Mecanismos para sincronización

Una vez discutidos los principales problemas a enfrentar para coordinar procesos/hilos que comparten espacio de direccionamiento, conviene presentar los mecanismos más importantes para establecer sincronización.

2.5.1 Semáforos

Un semáforo general (contador) es un tipo de datos abstracto que tiene 2 operaciones para acceso: P y V, de los nombres originales que les puso Dijkstra (palabras holandesas passeren, pasar, y vrygeven, liberar. Tannenbaum las denomina down (abajo) y up (arriba) respectivamente). En términos de la orientación a objetos un semáforo S se crea a partir de una clase que tiene un atributo privado de tipo número entero y una pareja de métodos públicos P y V. El funcionamiento de tales operaciones sobre S se define a continuación:

Si el valor del contador S es positivo, un hilo al invocar P(S) decrementa a S en una acción atómica; de otra forma (S <= 0) el hilo espera (se bloquea).

Si el hilo invoca a V(S) y no hay hilo alguno esperando por ese semáforo entonces el valor del contador S se incrementa atómicamente; de otra forma uno de los hilos en espera se libera y se le permite continuar ejecutando su código en las siguientes instrucciones de donde fue interrumpido (i.e. inmediatamente después de P(S)). El hilo liberado no necesariamente debe ser el que lleva más tiempo esperando.

Las operaciones sobre el contador del semáforo deben realizarse atómicamente para evitar condiciones de contención.

Un semáforo binario sólo puede tener valores de su contador entre 0 y 1. Cuando es iniciado con valor 1 permite resolver el problema de la exclusión mutua:

```
Semáforo binario compartido mutex con valor inicial 1 en cada hilo:
P(mutex);          //preprotocolo
      Sección crítica
V(mutex);          //postprotocolo
```

Un semáforo binario también guarda la señal `wakeup()` de tal suerte que no se pierde si la lista de hilos en espera está vacía.

Por tanto, los semáforos binarios sirven para dos propósitos:

- Sincronización de exclusión mutua
- Condición de sincronización: bloquear a los hilos hasta que alguna condición se haga verdadera u ocurra un evento:

```
Semáforo binario compartido S
En un hilo:            en el otro:
If(!cond) P(S);              V(S);
```

Ejemplo:

```
//variables compartidas por el consumidor y el productor:
int N, count = 0;
semaphore S=0, mutex=1; //semáforos binarios compartidos

producer(){
      while(true) {
            produce_Elemento();
            if (count == N) P(S); //delay
            introduce_en_buffer();
            P(mutex); count++; V(mutex);
            If (count == 1) V(S); //wakeup
      }
}

consumer(){
      while(true) {
            if (count == 0 P(S); //delay
            quita_Elemento();
            P(mutex); count--; V(mutex);
            If (count == N-1) V(S); //wakeup
            Consume_Elemento();
      }
}
```

Figura 2.1 Pseudocódigo para el productor-consumidor utilizando semáforos binarios.

Por su parte, un semáforo contador puede representar el número de recursos disponibles para asignamiento: un hilo invoca P(S) para solicitar una unidad de recurso e invoca a V(S) para regresar dicho recurso.

```
Semáforo contador compartido S iniciado al número total de recursos en
cada hilo:

P(S);       //solicita un recurso
      Utiliza el recurso
V(S);       //libera al recurso
```

Semáforos en Java

Los semáforos pueden implantarse a nivel del usuario (usando espera ocupada, por ejemplo) o a nivel del SO (inhabilitando las interrupciones a nivel del *kernel, Unix* ofrece semáforos como parte de los llamados al sistema para comunicación interprocesos (IPC)). Los semáforos binarios o contadores no se encuentran disponibles como tales en Java, pero su construcción no es difícil ya que Java ofrece como mecanismo para sincronización entre hilos a los monitores (que se verán en la siguiente sección).

Los semáforos utilizados para asegurar exclusión mutua son iniciados a 1 y sólo toman 2 valores: 1 indica cuando la SC se encuentra libre y 0 cuando está siendo ocupada. Por tanto, pueden verse como candados (*locks*) con 2 operaciones: cerrar el candado (*lock*) y abrir el

candado (*unlock*) (obviamente correspondiendo a las operaciones P y V respectivamente). Además de asegurar exclusión mutua (sólo se permite que un hilo se apodere de un candado a la vez), también se puede utilizar a un semáforo binario para bloquear a un hilo hasta que ocurra un evento que será causado por otro hilo:

```
Semáforo binario compartido S
en un hilo:       en el otro:
P(S);             V(S);
```

Es decir, no tienen la restricción que el mismo hilo que realizó P(S) deba hacer también el correspondiente V(S).

Cada objeto en Java tiene su candado, el cual puede utilizarse como semáforo binario implícito en un bloque de código indicado por el modificador *synchronized* en el objeto. Para entrar al bloque, un hilo debe adquirir primero el candado del objeto, para ello el hilo espera (se bloquea) si es que el candado está en uso por otro hilo. Cuando el hilo sale del bloque libera automáticamente al candado. Los candados son implantados en la JVM: si obj denota la referencia a un objeto, entonces se puede utilizar como un semáforo binario implícito de la siguiente forma:

```
Synchronized (obj){
      //sección crítica
}
```

cualquier objeto puede utilizarse en un bloque sincronizado.

Si todas las secciones críticas de código que necesitan exclusión mutua están en los métodos de un único objeto, entonces los bloques sincronizados deben usar la referencia *this*:

```
synchronized (this) {…}
```

incluso el cuerpo completo de un método puede indicarse como sincronizado:

```
synchronized tipo metodo(…) {
      //sección critica
}
```

Si las secciones críticas que requieren exclusión mutua están dispersas sobre varias clases distintas u objetos diferentes creados a partir de una clase, entonces un objeto común para los bloques de sincronización deben crearse fuera de las clases y pasar una referencia a sus constructores.

A continuación se muestra cómo se elimina la condición de contención para el ejemplo de:

```
Private Object mutex=null;
…
mutex=this;  //en el constructor
…
public void run(){
```

```
        for(int m=1; m<=M; m++)
              synchronized (mutex){
                    sum=fn(sum, m);
              }
}
```

Las soluciones propuestas a algunos problemas de sincronización clásicos se muestran enseguida.

Productor – consumidor

En el problema productor – consumidor, los productores envían mensajes que son recibidos por los consumidores. Los hilos se comunican usando un solo buffer compartido, que es manipulado por dos operaciones: deposit y fetch. Los productores insertan mensajes en el buffer al llamar a deposit; los consumidores reciben mensajes al llamar a fetch. Para asegurar que los mensajes no se sobrescriben y que solo se reciba uno a la vez, la ejecución de deposit y fetch debe alternar, con deposit ejecutándose primero. Para su solución se usan dos semáforos binarios que permiten la alternancia entre los hilos. El siguiente pseudo código ilustra esto.

Solución para un productor y un consumidor con semáforos binarios

```
var buf[1:n]:T          #Para algún tipo T
var empty:=sem:=1, full:sem:=0

Producer[i: 1..M]:: do true ->
                        Produce message m
                        Deposit:        P(empty)
                                            buf := m;
                                        V(full)
                    od

Consumer[j:1..N]:: do true ->
                        Fetch:          P(full)
                                            m:= buf
                                        V(empty)
                        Consume message m
                    od
```

Productor – consumidor con buffer limitado

A continuación se desarrolla una solución para el problema del buffer limitado, es decir, el problema de implementar un buffer con multislot. También se ilustra el uso de semáforos generales (contadores) como contadores de recursos.

Supongamos que hay un solo productor y un solo consumidor. El productor deposita mensajes en un buffer compartido; el consumidor lo retira. Esta cola puede representarse como una lista ligada o por un arreglo. En particular utilizaremos el arreglo y lo representaremos como buf[1:n], donde n es mayor que 1. Sea front el índice del mensaje que esta en el frente de la cola, y sea rear el índice del primer slot vacio. Inicialmente, front y rear toman el valor 0. Entonces el productor deposita mensajes m en el buffer al ejecutar:

Deposit: buf[rear] := m; rear:= rear **mod** n+1

Y el consumidor retira un mensaje y lo coloca en la variable local m al ejecutar:

Fetch: m:= buf[front]; front := front mod n+1

El operador mod (modulo) se usa para asegurar que el valor de front y rear estén siempre entre 1 y n. En el siguiente pseudocodigo los semáforos sirven como contador de recursos. En este caso, empty cuenta el número de slots vacíos del buffer, y full cuenta el número de slots ocupados. Los semáforos contadores son útiles cuando los hilos compiten por el acceso a múltiples recursos, como los slots del buffer o bloques de memoria.

```
var buf[1:n]:T          #Para algún tipo T
var front:=0, rear:=0
var empty:=sem:=n, full:sem:=0

Producer:: do true ->
                    Produce message m
                    Deposit:        P(empty)
                                        buf[rear] := m; rear:= rear mod n+1
                                    V(full)
                od

Consumer:: do true ->
                    Fetch:          P(full)
                                        m:= buf[front]; front := front mod n+1
                                    V(empty)
                    Consume message m
                od
```

Solución para múltiples productores y múltiples consumidores, usando semáforos generales (contadores)

Supongamos ahora que hay dos o más productores. Entonces cada uno podría ejecutar deposit al mismo tiempo si hay al menos dos slots vacíos. En este caso, ambos procesos tratarían de depositar su mensaje en el mismo slot (asignar el mensaje a buf[rear] y también incrementar rear). Similarmente, si hay dos o más consumidores, ambos podrían ejecutar fetch al mismo tiempo y recuperar el mismo mensaje. Así, deposit y fetch llegan a ser secciones críticas. Cada uno debe ejecutarse con exclusión mutua, pero no pueden ejecutarse concurrentemente con cualquier otro.

Así se usan los semáforos empty y full para que los productores y consumidores accedan a distintos slots del buffer.

Hemos solucionado dos problemas: primero la sincronización entre el productor y consumidor, y la sincronización entre productores y entre consumidores. El siguiente pseudocodigo ilustra esto:

var buf[1:n]:T #Para algún tipo T
var front:=0, rear:=0
var empty:=**sem**:=n, full:**sem**:=0
var mutexD: **sem**:=1, mutexF:**sem**:=1

Producer[1:n]:: **do** true ->
Produce message m
Deposit: **P**(empty) # Espacios libres
P(mutexD)
buf[rear] := m; rear:= rear mod n+1
V(mutexD)
V(full) #Num. items
od

Consumer[1:n]:: **do** true ->
Fetch: **P**(full)
P(mutexF)
m:= buf[front]; front := front mod n+1
V(mutexF)
V(empty)
Consume message m
od

El siguiente segmento de código (programa bbou.java) resuelve el problema de sincronización entre un productor y un consumidor que utilizan un buffer limitado.

```
...
   private int count = 0;
   private BinarySemaphore mutex = null;  //1
   private CountingSemaphore elements = null;   //0
   private CountingSemaphore spaces = null;  //numSlots
...

public void deposit(double value) {
      P(spaces);
      buffer[putIn] = value;
      putIn = (putIn + 1) % numSlots;
      P(mutex);
         count++;
      V(mutex);
      V(elements);
 }

 public double fetch() {
```

```
      double value;
      P(elements);
      value = buffer[takeOut];
      takeOut = (takeOut + 1) % numSlots;
      P(mutex);
         count--;
      V(mutex);
      V(spaces);
      return value;
}
```

El semáforo binario *mutex* se utiliza para asegurar exclusión mutua en las operaciones de actualización ó consulta sobre la variable compartida *count*. Los semáforos contadores `spaces` y `elements` se utilizan para sincronización de eventos entre el productor y el consumidor: el productor no puede producir si el buffer está lleno, será despertado por el consumidor avisándole que ya hay espacios vacíos disponibles; y viceversa, el consumidor no puede consumir si el buffer está vacío y será despertado por el productor cuando haya al menos un elemento disponible.

Productor – consumidor con buffer ilimitado

Requerimientos:

1.- El productor puede producir un item en cualquier momento, independientemente del estado del buffer.

2.- El consumidor podrá consumir items siempre que el buffer no esté vacío

3.- El orden en que los items se colocan en el buffer es el mismo que en el que se recuperan

4.- Todos los items producidos se consumen

Cuando el buffer esté vacío, el semáforo deberá tener el valor de 0, de forma que el proceso consumidor quede bloqueado por el P. Cuando el productor genere un nuevo item, se ejecutará el V.

Se utilizará un semáforo contador: num-items, cuyo valor inicial será 0. Para garantizar la exclusión mutua en el acceso al buffer, se utilizará el semáforo binario mutex, cuyo valor inicial es 1.

Ejercicio: Diseñar el algoritmo y el programa en Java
Algoritmo:

var buf[1:_]:T #Para algún tipo T
var front:=0, rear:=0
var num-items:**sem**:=0 #Semáforo contador
var mutex: **sem**:=1

```
Producer[1:n]:: do true ->
                    Produce message m
                    Deposit:    P(mutex)
                                   buf[rear] := m; rear:= rear+1
                                V(mutexD)
                                V(num-items)          #Num. items
               od

Consumer[1:n]:: do true ->
                    Fetch:      P(num-items)
                                P(mutex)
                                   m:= buf[front]; front := front +1
                                V(mutex)
                    Consume message m
                od
```

Otro ejemplo del uso de semáforos binarios y contadores: ABC

El programa ABCs.java inicia tres hilos, Pa, Pb y PC. El hilo Pa imprime la letra "A", el hilo Pb la letra "B", y Pc la letra "C". Utilizando dos semáforos binarios y un semáforo contador, los hilos sincronizan sus salidas satisfaciendo las siguientes condiciones:

1. Una "B" debe escribirse antes que una "C".
2. Las letras "B" y "C" deben alternar en la cadena de salida, es decir, después de la primera "B" en la salida, ninguna "B" puede escribirse hasta que una "C" se escriba. Similarmente para "C", si "C" esta en la cadena de salida, no se puede escribir otra "C" hasta que una "B" se escriba.
3. El número total de "B´s" y "C´s" que están en la salida no debe exceder el número de "A's" escritos.

Ejemplo de cadenas:

```
ACB          -- Invalida, viola 1.
ABACAC       -- Invalida, viola 2.
AABCABC      -- Invalida, viola 3.
AABCAAABC    -- valida
AAAABCBC     -- valida
AB           -- valida
```

Solución:

Algoritmo:

```
var sum:sem:=0       #Semáforo contador
var B:sem:=0, C:sem:=1,mutex: sem:=1     #Semáforos binarios

A :: do true ->
                                P(mutex)
                                   Write("A");
```

V(mutex)
V(sum) #Num. letras A's

od

B :: **do** true ->

P(C)
P(sum)
P(mutex)
Write("B");
V(mutex)
V(B) #Num. letras A's

od

C :: **do** true ->

P(B)
P(sum)
P(mutex)
Write("B");
V(mutex)
V(C) #Num. letras A's

od

Programa:

```
class ABCs extends MyObject {

   protected static final BinarySemaphore B       // these semaphores
      = new BinarySemaphore(0);                   // are static
   protected static final BinarySemaphore C       // so subclasses
      = new BinarySemaphore(1);                   // Pa, Pb,
   protected static final CountingSemaphore sum   // and Pc share
      = new CountingSemaphore(0);                 // them
…
}
class Pa extends ABCs implements Runnable {  // extend ABCs to
                                              // access semaphore sum
   public void run () {
      while (true) { nap(1+(int)(random(500)));
         P(mutex);
               System.out.print("A"); //System.out.flush();
         V(mutex);
         V(sum);
      }
   }
}

class Pb extends ABCs implements Runnable {

   public void run () {
```

```
      while (true) { nap(1+(int)(random(800)));
         P(C); P(sum);
         P(mutex);
            System.out.print("B");
            //System.out.flush();
         V(mutex);
         V(B);
      }
   }
}

class Pc extends ABCs implements Runnable {

   public void run () {
      while (true) { nap(1+(int)(random(800)));
         P(B); P(sum);
         P(mutex);
               System.out.print("C"); //System.out.flush();
         V(mutex);
         V(C);
      }
   }
}
```

Los semáforos binarios B y C son usados para forzar a los hilos Pb y Pc para alternar sus caracteres a imprimir. El semáforo contador sum lleva el numero de caracteres impresos por el hilo Pa. Así, Pb y Pc deben ejecutar P(sum) antes de escribir, el número total de caracteres que ellos escriben no debe exceder el número de A´s escritas por Pa.

Ejemplo: Barbero Dormilón

El siguiente segmento de código (correspondiente al programa `slba.java`) resuelve el problema del barbero dormilón: un barbero en una peluquería espera a los clientes durmiendo, cuando llega uno, lo despierta para que le corte el cabello, si continúan llegando clientes toman asiento en las sillas disponibles en la sala de espera; si llega un cliente y todas las sillas están ocupadas, se va. Los tiempos de duración del corte de cabello por cada cliente son aleatorios, así como las llegadas de los mismos a la peluquería.

```
class Salon extends MyObject {

   private int numChairs = 0;//5
   private CountingSemaphore customers = null;  //0
   private CountingSemaphore barber = null;//0
   private BinarySemaphore mutex = null;//1
   private CountingSemaphore cutting = null;//0
   private int waiting = 0;

…
public void wantToCut() {  //Barber
      P(customers);  //espera al sig. cliente
      P(mutex);
      waiting--;
      V(barber); //tiene cliente
      V(mutex);
      P(cutting); //corta cabello
```

```
}

public void wantHairCut(int i, int cutTime) { //Cliente
   int napping;
   P(mutex); //llega al salon
   if (waiting < numChairs) { //si hay asiento
      waiting++;
      V(customers); // toma asiento
      V(mutex);
      P(barber); //aguarda su turno para cortarse el cabello
      napping = 1 + (int)random(cutTime);
      nap(napping);
      V(cutting); //termina su corte de cabello
   } else {
      V(mutex);
   }
}
```

El (hilo) barbero espera que lleguen clientes indicado por el semáforo customers, entonces decrementa el contador de clientes en espera (en exclusión mutua). El barbero indica al semáforo contador barber que pase el siguiente cliente de la sala de espera al sillón del barbero para cortarle el cabello. Si hay asiento en la sala de espera, un cliente indica su llegada incrementando el semáforo contador customers, el cual a su vez permite despertar al barbero si es que es el primer cliente que llega; a su vez, el cliente espera su turno para cortarse el cabello mediante el semáforo barber. La duración del corte depende del cliente y se indica por el semáforo cutting.

Este ejemplo ilustra el uso de semáforos tanto para exclusión mutua como para condición de sincronización. La interacción entre los clientes y el barbero es un ejemplo de relación cliente-servidor, los cuales coinciden en ciertos puntos en el tiempo. A pesar de su simplicidad, éste problema ilustra muchos de los problemas de sincronización que ocurren en muchas aplicaciones (concurrentes) en el mundo real.

Ejemplo: Filosofos

A continuación, se presenta un fragmento del programa dphi.java que resuelve el problema de los filósofos comensales de Dijkstra: cinco filósofos están sentados en una mesa circular. En el centro hay un plato de spaghetti, el cual, para comerlo se necesitan 2 tenedores. Hay un tenedor por cada filósofo, cuando uno quiere comer tiene que tomar prestado el tenedor del filósofo vecino. La vida de los filósofos se pasa en un ciclo continúo entre los estados sucesivos: tras un rato (aleatorio) pensando, al filosofo le da hambre y entonces procura apropiarse de los tenedores a su alcance, si lo logra, come un rato (también aleatorio), de lo contrario puede suceder que suelte su tenedor o que se encapriche y pueda provocar que ni él ni sus vecinos coman. Cuando termina de comer, vuelve a pensar.

```
private static final int
   THINKING = 0, HUNGRY = 1, EATING = 2;
private int numPhils = 0;
private int[] state = null;
private BinarySemaphore[] self = null;
private BinarySemaphore mutex = null;
```

```
...
      state = new int[numPhils];
      for (int i = 0; i < numPhils; i++) state[i] = THINKING;
      self = new BinarySemaphore[numPhils];
      for (int i = 0; i < numPhils; i++)self[i] = new BinarySemaphore(0);
      mutex = new BinarySemaphore(1);
…
  private final int left(int i) { return (numPhils + i - 1) % numPhils; }

  private final int right(int i) { return (i + 1) % numPhils; }

  public void takeForks(int i) {
      P(mutex);
      state[i] = HUNGRY;
      test(i);
      V(mutex);
      P(self[i]);
   }

   public void putForks(int i) {
      P(mutex);
      state[i] = THINKING;
      test(left(i));
      test(right(i));
      V(mutex);
   }

   private void test(int k) {
      if (state[left(k)] != EATING && state[right(k)] != EATING
            && state[k] == HUNGRY) {
         state[k] = EATING;
         V(self[k]);
      }
   }
```

Este programa es una traducción directa en Java de la solución clásica presentada por ejemplo en el libro de Tannenbaum de sistemas operativos. Se asume un servidor (mesa) al cual se le solicitan los tenedores. El servidor central verifica la disponibilidad de los mismos examinando los estados de los filósofos vecinos al que hizo la solicitud. Si ningún vecino está comiendo, entonces el filósofo en cuestión come; de otra forma debe esperar, lo cual se indica mediante `P(self[i])`. Cada vez que un filósofo termina de comer coloca sus tenedores y el servidor verifica si sus vecinos quieren comer (invocando al método `test()`), en cuyo caso un filósofo hambriento es desbloqueado mediante `V(self[k])`.

Ejercicios 1:

1. Se tiene una cuenta única en un banco. Se desean realizar depósitos y extracciones de la misma, garantizando que estas no se realicen de manera simultánea, se tiene una cantidad inicial de 1000, el depósito no tiene restricciones pero el retiro sí, no se puede retirar una cantidad mayor a la existente en la cuenta. Use semáforos binarios para dar el paso entre los hilos, uno que deposita y el otro que retira.

2. Se tiene una cuenta bancaria en un banco, el banco le proporciona al usuario una tarjeta de crédito, que tiene un límite en saldo ($30 000) cada vez que el cliente hace una compra y/o un retiro, se decrementa el límite y aumenta el saldo por pagar, el cliente debe pagar (depositar) en el banco las compras y retiros hechos antes de su fecha de corte (el día 20 de cada mes), de lo contrario su saldo por pagar sufre un incremento del 4%. Considere que los retiros en efectivo hechos (retiros), sufren un interés del 2% inmediatamente.

3. Se desean crear n moléculas de dióxido de carbono, cada molécula está formada por dos átomos de oxigeno y un átomo de carbono (CO_2). Resuelva el problema usando semáforos.

4. Entrada de la 14 sur. En la puerta de acceso de la 14 sur, se tiene un policía que pide una credencia a cada persona que desea ingresar a las instalaciones de la FCC; el sólo puede atender a uno a la vez. Después de cierto tiempo las personas que entraron abandonan las instalaciones. Resuelva el problema utilizando semáforos.

5. Se desea simular el ascenso y descenso de pasaje en la parada de la 14 sur, se cuentan con 3 rutas (3 estrellas, 16 y 33), cada ruta tiene un número fijo de asientos vacios, cada ruta tiene que esperar a que por lo menos suba la mitad de los asientos vacios para que pueda salir de la parada. Use semáforos para resolver este problema, considere que tiene n hilos pasajeros y 3 hilos rutas.

6. El problema de la guardería: Suponga que se tiene una guardería en la que se tienen *n* cuidadores y *m* pequeños. Cada cuidador debe cuidar a 3 niños, la repartición entre los cuidadores es equitativa, después de cierta hora los pequeños abandonan la guardería.

7. Simule el llenado de refrescos de la empresa "πCola": Se tienen 3 empleados (hilos) y un agente (hilo). Se cuenta además con tres ingredientes, botellas, líquido y corcholatas; el agente proporciona 2 de los ingredientes para formar el refresco. Cada empleado posee un ingrediente, el cual podrá incorporar cuando sea necesario para formar el refresco. Diseñe un programa que sincronice las acciones, usando semáforos.

8. En la "Fonda Doña Clarita" los días viernes se realiza un venta de buffet, como plato fuerte se tienen 5 platillos puestos en tazones, cada vez que un tazón se termina Doña Clarita vuelve a llenarlo. Resuelva el problema para *n* clientes, utilizando semáforos.

9. El salón de fiestas "*Tribilin*" se reservo para la fiesta de cumpleaños de Manuel, sus papás pidieron que se sirviera en platos desechables "lomo adobado, ensalada y espagueti"; además contrataron el servicio de meseros y un cocinero que sirve los platos en la cocina y los meseros los reparten a los invitados. Cada vez que se terminan los platos con alimento, el cocinero tiene que volver a servirlos. Sincronice las actividades usando semáforos. Modifique el problema para que sean *n* cocineros y *m* meseros.