Sincronización con Semáforos

PROGRAMACIÓN CONCURRENTE – TEMA 3.2

Universidad
Rey Juan Carlos

Sincronización con Semáforos

- ¿Qué es un Semáforo?
- Sincronización Condicional
- Exclusión Mutua
- Metodología de Desarrollo
- **Sincronización Avanzada**
  - Sincronización de barrera
  - Comunicación con buffer
  - Exclusión mutua generalizada
- Conclusiones

Sincronización avanzada

Sincronización con semáforos

- Todo programa concurrente se puede implementar con **sincronizaciones condicionales** y **exclusiones mutuas**
- Existen **variantes** de estas formas de sincronización que se utilizan habitualmente en la programación concurrente
- En este apartado se estudiarán estas variantes

Sincronización con Semáforos

- ¿Qué es un Semáforo?
- Sincronización Condicional
- Exclusión Mutua
- Metodología de Desarrollo
- Sincronización Avanzada
  - **Sincronización de barrera**
  - Comunicación con buffer
  - Exclusión mutua generalizada
- Conclusiones

Sincronización de Barrera

- La **Sincronización de Barrera** (*Barrier*) es una sincronización condicional en la que los procesos tienen que esperar a que el resto de procesos lleguen al mismo punto para poder continuar su ejecución
- Vamos a estudiar este tipo de sincronización con los siguientes requisitos
  - Programa con N procesos
  - Cada proceso escribe la letra **A** y luego la **B**
  - Los procesos tienen que esperar que todos hayan escrito la letra **A** antes de escribir la **B**

Sincronización de b

- Iª aproximación incorrecta

```
public class SincBarreraMal1 {
  private static final int NPROCESOS = 3;
  private static volatile int nProcesos;
  private static Semaphore sb;
  private static void proceso() throws
                                  InterruptedException{
     System.out.println("A");
     nProcesos++;
     if(nProcesos < NPROCESOS)
        sb.acquire();
     else
        for(int i=0; i<NPROCESOS-1;i++){
           sb.release();
     System.out.println("B");
  }

  public static void main(String[] args) {
     sb = new Semaphore(0);
     List<Thread> ths = new ArrayList<Thread>();
     for (int i = 0; i < NPROCESOS; i++) {
        Thread th = new Thread(new Runnable() {
           public void run() {
           // TODO try/catch
              proceso();
        }
        }, "proceso"+ i);
        th.start();
        ths.add(th);
     }
  }
}
```

Sincronización de barrera

- 1ª aproximación incorrecta

  Puede provocar interbloqueo (*deadlock*) porque **`nProcesos`** no está bajo exclusión mutua y se pueden producir errores al contar

```
public class SincBarreraMal1 {
  private static final int NPROCESOS = 3;
  private static volatile int nProcesos;
  private static Semaphore sb;
  private static void proceso() throws
                                  InterruptedException{
     System.out.println("A");
     nProcesos++;
     if(nProcesos < NPROCESOS)
        sb.acquire();
     else
        for(int i=0; i<NPROCESOS-1;i++){
           sb.release();
     System.out.println("B");
  }

  public static void main(String[] args) {
     sb = new Semaphore(0);
     List<Thread> ths = new ArrayList<Thread>();
     for (int i = 0; i < NPROCESOS; i++) {
        Thread th = new Thread(new Runnable() {
           public void run() {
           // TODO try/catch
              proceso();
        }
        }, "proceso"+ i);
        th.start();
        ths.add(th);
     }
  }
}
```

Sincronización de barrera

- 2ª aproximación incorrecta

```
public class SincBarreraMal1 {
  private static final int NPROCESOS = 3;
  private static volatile int nProcesos;
  private static Semaphore sb;
  private static Semaphore emNProcesos;
  private static void proceso() throws
                                  InterruptedException{
     System.out.println("A");
     emNProcesos.acquire();
     nProcesos++;
     emNProcesos.release();
     if(nProcesos < NPROCESOS)
        sb.acquire();
     else
        for(int i=0; i<NPROCESOS-1;i++){
           sb.release();
     System.out.println("B");
  }

  public static void main(String[] args) {
     nProcesos = 0;
     sb = new Semaphore(0);
     emProcesos = new Semaphore(1);
     List<Thread> ths = new ArrayList<Thread>();
     for (int i = 0; i < NPROCESOS; i++) {
        Thread th = new Thread(new Runnable() {
           public void run() {
            // TODO try/catch
              proceso();
         }
        }, "proceso"+ i);
        th.start();
        ths.add(th);
     }
  }
}
```

Sincronización de barrera

- 2ª aproximación incorrecta

  Si se deja la consulta del contador fuera de la Exclusión Mutua, puede ocurrir que dos procesos hagan **`release`**

- Ojo que no está mal, sino que deja al semáforo en un estado **impredecible**

```
public class SincBarreraMal1 {
  private static final int NPROCESOS = 3;
  private static volatile int nProcesos;
  private static Semaphore sb;
  private static Semaphore emNProcesos;
  private static void proceso() throws
                                  InterruptedException{
     System.out.println("A");
     emNProcesos.acquire();
     nProcesos++;
     emNProcesos.release();
     if(nProcesos < NPROCESOS)
        sb.acquire();
     else
        for(int i=0; i<NPROCESOS-1;i++){
           sb.release();
     System.out.println("B");
  }

  public static void main(String[] args) {
     nProcesos = 0;
     sb = new Semaphore(0);
     emProcesos = new Semaphore(1);
     List<Thread> ths = new ArrayList<Thread>();
     for (int i = 0; i < NPROCESOS; i++) {
        Thread th = new Thread(new Runnable() {
           public void run() {
            // TODO try/catch
              proceso();
         }
        }, "proceso"+ i);
        th.start();
        ths.add(th);
     }
  }
}
```

Sincronización de barrera

- 1ª solución correcta

  En cada rama del **`if`** el proceso libera la exclusión mutua y se bloquea o no dependiendo de la condición

- **NOTA**: Esta implementación no permite usar una única barrera en un bucle ya que se podría producir inanición

```
public class SincBarreraBien1 {
  private static final int NPROCESOS = 3;
  private static volatile int nProcesos;
  private static Semaphore sb;
  private static Semaphore emNProcesos;
  private static void proceso() throws
                                InterruptedException{
     System.out.println("A");
     emNProcesos.acquire();
     nProcesos++;
     if(nProcesos < NPROCESOS)
        emNProcesos.release();
        sb.acquire();
     else
        emNProcesos.release();
        for(int i=0; i<NPROCESOS-1;i++){
           sb.release();
     System.out.println("B");
  }

  public static void main(String[] args) {
     nProcesos = 0;
     sb = new Semaphore(0);
     emProcesos = new Semaphore(1);
     List<Thread> ths = new ArrayList<Thread>();
     for (int i = 0; i < NPROCESOS; i++) {
        Thread th = new Thread(new Runnable() {
           public void run() {
            // TODO try/catch
              proceso();
         }
        }, "proceso"+ i);
        th.start();
        ths.add(th);
     }
  }
}
```

Sincronización de barrera

- 2ª solución correcta

  En la rama del **`if`** el proceso deja el semáforo preparado para cuando él mismo ejecute un **`acquire`**

- **NOTA**: Esta implementación no permite usar una única barrera en un bucle ya que se podría producir inanición

```
public class SincBarreraBien2 {
  private static final int NPROCESOS = 3;
  private static volatile int nProcesos;
  private static Semaphore sb;
  private static Semaphore emNProcesos;
  private static void proceso() throws
                                  InterruptedException{
     System.out.println("A");
     emNProcesos.acquire();
     nProcesos++;
     if(nProcesos == NPROCESOS){
         for(int i = 0; i < NPROCESOS; i++)
              sb.release();
     }
     emNProcesos.release();
     sb.acquire();
     System.out.println("B");
  }

  public static void main(String[] args) {
     nProcesos = 0;
     sb = new Semaphore(0);
     emNProcesos = new Semaphore(1);
     List<Thread> ths = new ArrayList<Thread>();
     for (int i = 0; i < NPROCESOS; i++) {
        Thread th = new Thread(new Runnable() {
           public void run() {
            // TODO try/catch
              proceso();
         }
        }, "proceso"+ i);
        th.start();
        ths.add(th);
     }
  }
}
```

Sincronización de barrera

- Se desea implementar un bucle en el que todos los procesos muestran A y se esperan. El ultimo proceso muestra * y desbloquea a los demás
- Si se necesita usar una única barrera en un bucle, debe tenerse control de los desbloqueos de los procesos

Sincronización de barrera

- Se desea implementar un bucle en el que todos los procesos muestran A y se esperan. El ultimo proceso muestra * y desbloquea a los demás
- Si se necesita usar una única barrera en un bucle, debe tenerse control de los desbloqueos de los procesos

```
public class SincBarreraBien2 {
    //Los mísmo atributos más …
   private static Semaphore desbloqueo;
   private static void proceso() throws
                                    InterruptedException{
      System.out.println("A");
      emNProcesos.acquire();
      nProcesos++;
      if(nProcesos < NPROCESOS)
          emNProcesos.release();
          sb.acquire();
          desbloqueo.release();
      else // último proceso
         System.out.println("*");
         nProcesos = 0;
         for(int i = 0; i < NPROCESOS-1;i++){
               sb.release();
         for(int i =0; i < NPROCESOS-1;i++){
               desbloqueo.acquire();
   }

   public static void main(String[] args) {
      //La mima inicialización más …
      desbloqueo = new Semaphore(0);
      for (int i = 0; i < NPROCESOS; i++) {
         Thread th = new Thread(new Runnable() {
            public void run() {
             // TODO try/catch
               proceso();
          }
         }, "proceso"+ i);
         th.start();
         ths.add(th);
      }
   }
}
```

Sincronización con Semáforos

- ¿Qué es un Semáforo?
- Sincronización Condicional
- Exclusión Mutua
- Metodología de Desarrollo
- Sincronización Avanzada
  - Sincronización de barrera
  - **Comunicación con buffer**
  - Exclusión mutua generalizada
- Conclusiones

Comunicación con Buffer

- Hemos visto cómo los procesos se comunican con una variable
- Es habitual que los procesos se comuniquen con un **buffer** con varias posiciones para almacenar temporalmente información
- El **buffer** permite que se pueda ir insertando información aunque no esté preparado el proceso encargado de usarla
- El problema típico con el que se estudia la comunicación con buffer es el **problema de los Productores Consumidores**

Productor Consumidor

- **Procesos**
  - Los Productores son procesos que generan datos
  - Los Consumidores son procesos que consumen los datos en el orden en que se generan
- **Restricciones**
  - Cada productor genera un único dato cada vez
  - Un consumidor sólo puede consumir un dato cuando éste haya sido generado por el productor
  - Todos los productos se consumen

Productor Consumidor

- **Sincronización**
  - Los consumidores deben de bloquearse cuando no tengan datos que consumir
  - Los productores deben bloquearse cuando el buffer esté lleno
- **Comunicación**
  - Se utilizará un buffer para almacenar los datos producidos antes de ser consumidos

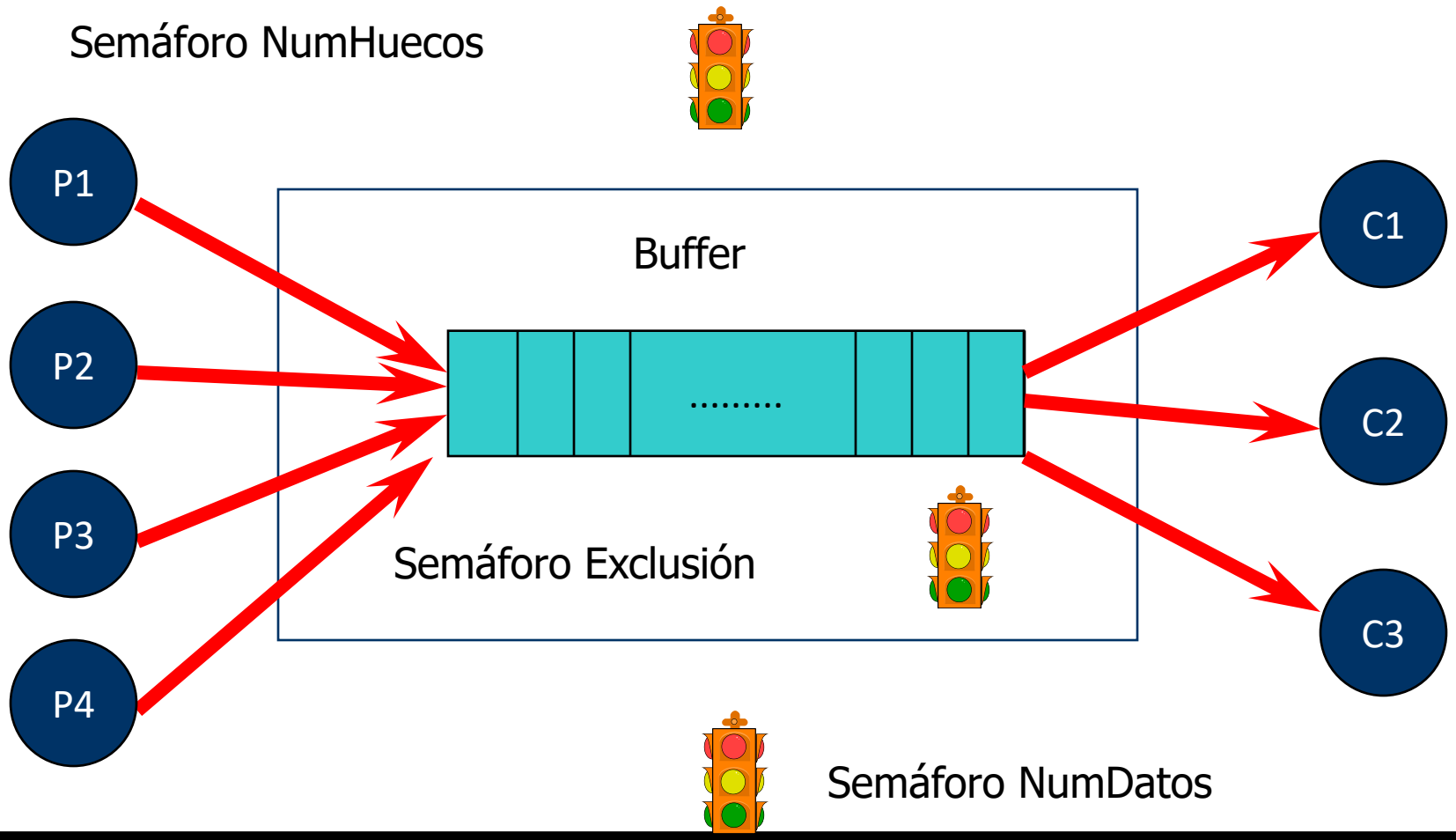
Productor Consumidor

- Cuando el buffer está lleno el productor se bloquea (**`acquire`**), hasta que el consumidor deja un hueco (**`release`**)

- Solución: un semáforo que marque el número de huecos disponibles en el buffer

Productor Consumidor

Sincronización Avanzada



Productor Consumidor

Sincronización Avanzada

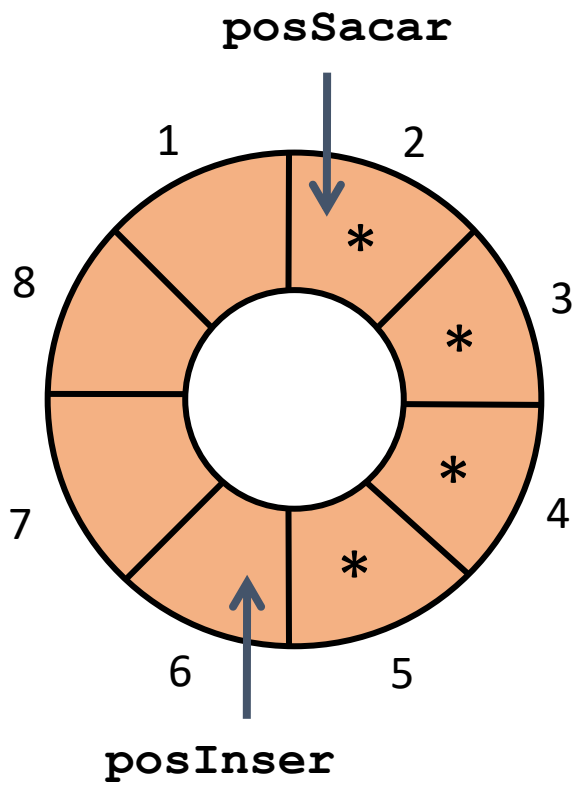
- Esquema de la aplicación
- Ojo que no tiene ningún tipo de sincronización

```
public class Buffer {

   private static final int BUFFER_SIZE = 10;
   private int[] datos = new int[BUFFER_SIZE];

   private int posInser = 0;
   private int posSacar = 0;

   public Buffer(){
      for(int i = 0; i < BUFFER_SIZE; i++){
         this.datos[i] = -1; //Sólo para mostrar que saca basura
      }
   }

   public void insertar(int dato) {
      datos[posInser] = dato;
      posInser = (posInser+1) % datos.length;
   }

   public int sacar() {
      int dato = datos[posSacar];
      datos[posSacar] = -1;//Sólo para mostrar que saca basura
      posSacar = (posSacar+1) % datos.length; return dato;
   }
}
```

Productores Consumidores

Comunicación con Buffer

Buffer circular




```
public class ProdConsBufferMal {
   private static Buffer buffer;
   public static void productor(){
      for(int i=0; i<20; i++){
         sleepRandom(500);
         buffer.insertar(i);
      }
   }
   public static void consumidor(){
      while(true){
         int data = buffer.sacar();
         sleepRandom(500);
         print(data+" ");
      }
   }
   public static void main(String[] args) {
      buffer = new Buffer();
      // Crear hilos
      List<Thread> ths = new ArrayList<Thread>();
      for (int i = 0; i < numThreads; i++) {
         …
      // Arrancar hilos
      for (int i = 0; i < numThreads; i++) {
         …
      // Esperar a que todos los hilos hayan terminado
      for (int i = 0; i < numThreads; i++) {
         …
      System.out.println("Programa finalizado");
   }
}
```

Productores Consumidores

Comunicación con Buffer

- Incorporar los puntos de sincronización
  - Sincronización Condicional
    - Un productor se bloqueará antes de insertar un dato si el buffer está lleno
    - Un consumidor se bloqueará antes de leer un dato si el buffer está vacío
  - Exclusión Mutua
    - Las variables de control del buffer deben estar bajo exclusión mutua

Productores Consumidores

Comunicación con Buffer

- Insertar

```
public void insertar(int dato) throws InterruptedException{
    this.nHuecos.acquire();

    this.excMutPosInsert.acquire();
    datos[posInser] = dato;
    posInser = (posInser+1) % datos.length;
    this.excMutPosInsert.release();

    this.nProductos.release();
  }
```

Productores Consumidores

Comunicación con Buffer

- Sacar

```
public int sacar() throws InterruptedException {
    this.nProductos.acquire();

    this.excMutPosSacar.acquire();
    int dato = datos[posSacar];
    datos[posSacar] = -1;
    posSacar = (posSacar+1) % datos.length;
    this.excMutPosSacar.release();

    this.nHuecos.release();

    return dato;
}
```

Productores Consumidores

Comunicación con Buffer

- Declaración e inicialización de semáforos

```
private Semaphore nHuecos = new Semaphore(BUFFER_SIZE);
private Semaphore nProductos = new Semaphore(0); //Sinc Cond
private Semaphore excMutPosInsert = new Semaphore(1); //Exc. Mutua
private Semaphore excMutPosSacar = new Semaphore(1); //Exc. Mutua
```

Sincronización con Semáforos

- ¿Qué es un Semáforo?
- Sincronización Condicional
- Exclusión Mutua
- Metodología de Desarrollo
- Sincronización Avanzada
  - Sincronización de barrera
  - Comunicación con buffer
  - **Exclusión mutua generalizada**
- Conclusiones

Exclusión Mutua Generalizada

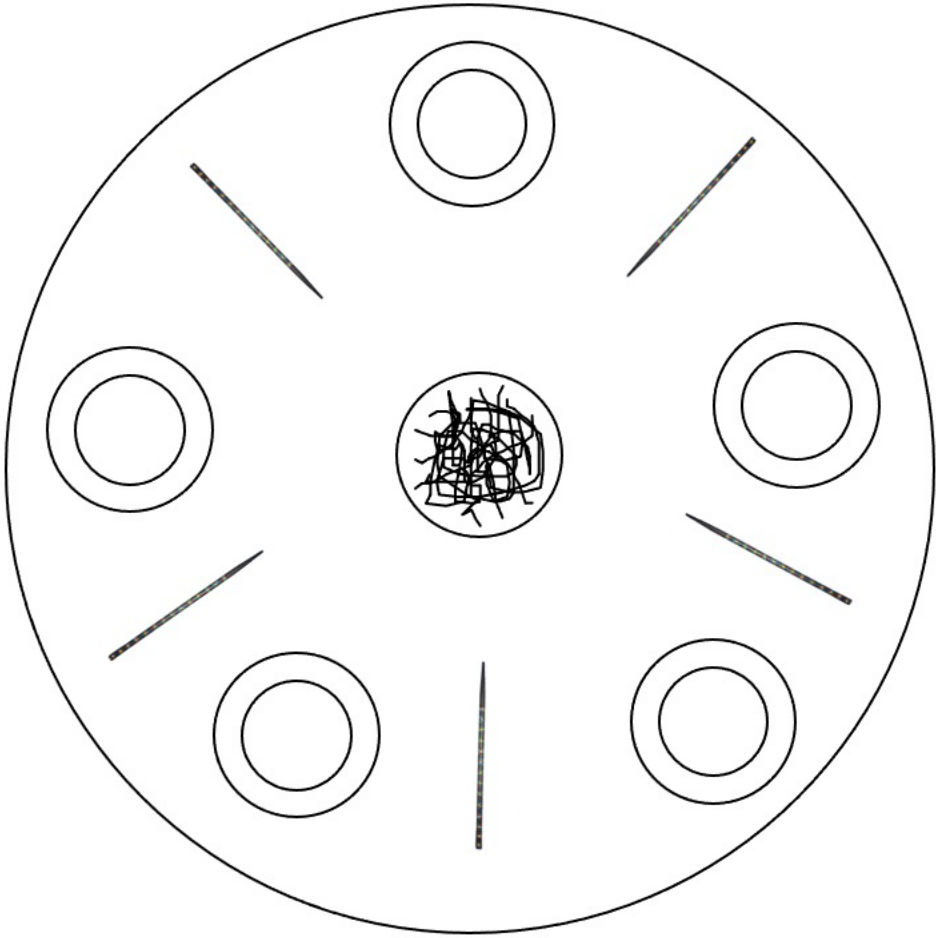
- Se tiene cuando el número de procesos que pueden ejecutar la sección crítica a la vez es *K> I*
- Se implementa con semáforos asignando inicialmente un valor *K al semáforo*

- Para entrar en la sección crítica hay que coger una bola de la caja, y dejarla al salir para que la pueda coger otro proceso
- El contador del semáforo indica los huecos libres de la sección crítica

```
public class ExcMutuaGenSem {

   private static Semaphore excMutua;
   private static int numThreads = 5;
   private static int numThInSC = 3;

   public static void p() throws InterruptedException {
      for (int i = 0; i < 5; i++) {
         excMutua.acquire();           // Preprotocol
         System.out.println("SC1 ");   // Sección Crítica
         System.out.println("SC2 ");   // Sección Crítica
         excMutua.release();           // Postprotocolo
         System.out.println("SNC1 ");  // Sección No Crítica
         System.out.println("SNC2 ");  // Sección No Crítica

      }
   }

   public static void main(String[] args) throws
                                          InterruptedException{
      excMutua = new Semaphore(numThInSC);

      // Crear hilos
      List<Thread> ths = new ArrayList<Thread>();
      for (int i = 0; i < numThreads; i++) {
         …
      // Arrancar hilos
      for (int i = 0; i < numThreads; i++) {
         …
      // Esperar a que todos los hilos hayan terminado
      for (int i = 0; i < numThreads; i++) {
         …
      System.out.println("Programa finalizado");
   }
}
```

El problema de la cena de los filósofos

- **5 Filósofos piensan** libremente y **comen** en una mesa con 5 platos.
- Cada plato está asignado a un filósofo.
- Los platos se reponen continuamente
- Hay 5 palillos, uno entre cada par de platos adyacentes



El problema de la cena de los filósofos

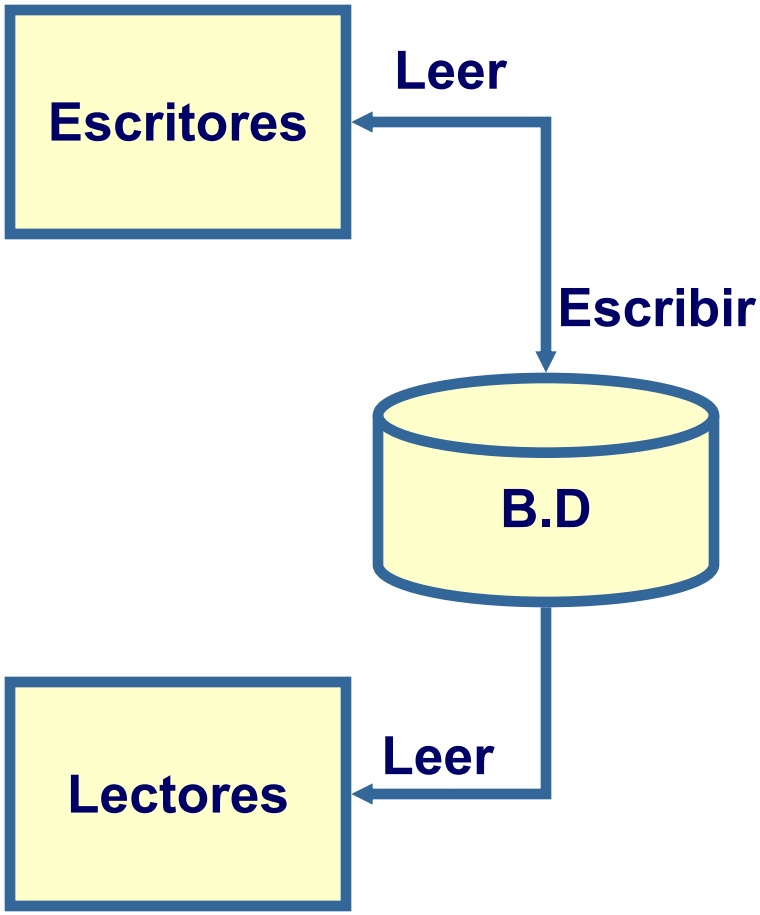
- Comer:
  - El filósofo se sienta delante de su plato y toma de uno en uno los tenedores situados a ambos lados de su plato (Preprotocolo)
  - Come
  - Cuando finaliza, deja los dos tenedores en su posición original (Postprotocolo)
  - Todo filósofo que come, en algún momento se sacia y termina

El problema de la cena de los filósofos

- Requisitos:
  - Un filósofo sólo puede comer cuando tenga los dos tenedores
  - Los tenedores se cogen y se dejan de uno en uno
  - Dos filósofos no pueden tener el mismo tenedor simultáneamente (EM de acceso al tenedor)
  - Si varios filósofos tratan de comer al mismo tiempo, uno de ellos debe conseguirlo. (Ausencia Interbloqueo)
  - Si un filósofo desea comer y tiene competencia, en algún momento lo deberá poder hacer. (Ausencia de Inanición)
  - En ausencia de competencia, un filósofo que quiera comer deberá hacerlo sin retrasos innecesarios (Ausencia retrasos)

El problema de los lectores/escritores

- Acceso Combinado (Concurrente-Exclusivo) a variables compartidas
- Dos tipos de procesos que comparten una misma memoria compartida: los Lectores y los Escritores
- Los Lectores pueden leer de la memoria compartida
- Los Escritores pueden leer y escribir en la memoria compartida



El problema de los lectores/escritores

- Requisitos
  - Cualquier número de lectores puede acceder a la BD, si no hay escritores accediendo
  - El **acceso a la BD de los escritores es exclusivo**. -> Mientras haya algún lector leyendo, ningún escritor puede acceder a la BD, pero otros lectores sí
  - Puede haber varios escritores trabajando, aunque estos se deberán sincronizar para que la escritura se lleve a cabo de uno en uno
  - Se da prioridad a los escritores. Ningún lector puede acceder a la BD cuando haya escritores que desean hacerlo (Inanición de Lectores)

Estado de los procesos

El problema de los Lectores/Escritores

- **LectoresActivos:** nº de lectores que han iniciado su Preprotocolo. Están activos hasta que finalizan el Postprotocolo.
  - Un lector activo puede estar esperando a obtener el acceso a los datos, que se le concederá cuando no haya escritores activos
- **LectoresTrabajando**: nº de lectores que han conseguido el acceso a la BD (Han terminado su Preprotocolo). Están trabajando hasta que finalizan el Postprotocolo.
- LectoresActivos $\geq$ LectoresTrabajando

Estado de los procesos

El problema de los Lectores/Escritores

- **EscritoresActivos:** nº de escritores que han iniciado su Preprotocolo. Están activos hasta que finalizan el postprotocolo.
- **EscritoresTrabajando**: nº de escritores que han conseguido el acceso a la BD (han terminado Preprotocolo). Están trabajando hasta que finalizan el postprotocolo.
- EscritoresActivos ≥ EscritoresTrabajando
- Puede ocurrir que EscritoresTrabajando ≥ 1

Estado de los procesos

- Para bloquear a los lectores y escritores:
  - PermisoLectura: bloquear lectores cuando hay escritores activos.
  - Permiso Escritura: bloquear escritores cuando haya lectores trabajando.
  - ExclusiónEscritura: bloquear escritores si hay uno escribiendo
- INICIALMENTE:
  - ExclusionEstado, ExclusionEscritura a 1.
  - PermisoLectura, PermisoEscritura a 0

Sincronización con Semáforos

- ¿Qué es un Semáforo?
- Sincronización Condicional
- Exclusión Mutua
- Metodología de Desarrollo
- Sincronización Avanzada
- **Conclusiones**

Conclusiones

- Ventajas de los Semáforos
  - Permiten resolver de forma sencilla y eficiente la sincronización de procesos concurrentes
  - Se pueden usar para implementar de forma muy sencilla el problema de la Exclusión Mutua
  - Están presentes en la mayoría de lenguajes y librerías de soporte a la concurrencia

Conclusiones

- Desventajas de los Semáforos
  - Son primitivas de muy bajo nivel
  - Omitir un simple **`release`** lleva a interbloqueo
  - Omitir un simple **`acquire`** lleva a la violación de la Exclusión Mutua
  - Colocar las operaciones **`acquire/release`** en los lugares no adecuados puede llevar a comportamientos erróneos del programa
  - No estructuran el código del programa (libertad total para acceder a las variables compartidas), lo que hace que los códigos sean difíciles de mantener y los errores difíciles de rastrear

Conclusiones

Sincronización con Semáforos

- Otras herramientas de sincronización de procesos con mayor nivel de abstracción:
  - Modelo de Memoria Compartida
    - Monitores
    - Regiones Críticas
    - Regiones Críticas Condicionales
    - Sucesos
    - Buzones
    - Recursos
  - Modelo de Paso de Mensajes
    - Envío asíncrono
    - Envío síncrono o cita simple
    - Invocación Remota o cita extendida