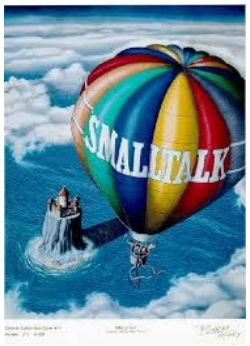
Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue

Programación Concurrente



Repaso

Sección crítica

- El problema de la sección crítica consiste en diseñar algún tipo de solución para garantizar que los procesos involucrados puedan operar sin generar ningún tipo de inconsistencia.

> El segmento de código en el que un proceso puede modificar variables compartidas con otros procesos se denomina **sección crítica**

- **Sección de entrada**, se solicita el acceso a la sección crítica.
- **Sección crítica**, en la que se realiza la modificación efectiva de
- los datos compartidos.
- **Sección de salida**, en la que típicamente se hará explícita la salida de la sección crítica.
- **Sección restante**, que comprende el resto del código fuente.ión.

Sección crítica

- El código se divide en las siguientes secciones

SECCION DE ENTRADA

SECCION CRITICA

SECCION DE SALIDA

SECCION RESTANTE

- Cualquier solución al problema de la sección crítica debe verificar:
  - **Exclusión mutua**, si un proceso está en su sección crítica, entonces ningún otro proceso puede ejecutar su sección crítica.
  - **Progreso**, todos los procesos que no estén en su sección de salida podrán participar en la decisión de quién es el siguiente en ejecutar su sección crítica.
  - **Espera limitada**, todo proceso debería poder entrar en algún momento a la sección crítica

Exclusión mutua

- Existen partes de código con **variables compartidas** y que deben ejecutarse en **exclusión mutua -MUTEX-**

Los algoritmos de exclusión mutua **evitan** que más de un **proceso** a la vez ingrese a las **secciones críticas.**

- Propiedades de seguridad **(safety)**
  (aseguran que nada malo va a pasar)=
  - Exclusión mutua
  - Condición de sincronización

Mecanismos de sincronización vistos

- **Semáforos**: - Arquitecturas no estructuradas - -
  - Bajo nivel de abstracción, de fácil comprensión y con una gran capacidad funcional.
  - Resultan peligrosos de manejar y son causa de muchos errores.
- **Bloques y métodos sincronizados -** Arquitecturas estructuradas -
  - Módulos de programación de alto nivel de abstracción
- **Monitores -** Arquitecturas estructuradas -
  - Módulos de programación de alto nivel de abstracción
  - Todos los métodos están sincronizados

Exclusión mutua

Las **zonas críticas** son secciones del programa que sólo debe ejecutar un *thread* en un momento dado, o habría problemas

- **Protocolo**
  - - si está ocupado, espero
  - – si está libre:
    - Entro
    - cierro por dentro
    - hago mis cosas
    - salgo
      - si hay alguien esperando, que entre
      - si no, queda abierto

Ejemplo: vble compartida

```
public class ContadorCompartido {
   private int n = 0;
   private int id = 0;

 public int getN(String id) {
   return n;
 }

 public void setN(String id, int n) {
   this.n = n;
   this.id = id;
  }
 }
```

Semáforos

```
ALGORITMO ejemploSem
   …..
         semaforo wait
         //código sección crítica
         semaforo signal
   …….
FIN ALGORITMO proceso2
```

```
Semaphore semaforo = new Semaphore (1);
….
         semaforo.acquire();
               int valor = cc.getN(id);
               valor++;
               cc.setN(id, valor);
          semaforo.release();
   ……
```

```
|semaforo |
semaforo := Semaphore new signal.
[….
        semaforo wait.
         "sección crítica "
         semaforo signal.
…..].
```

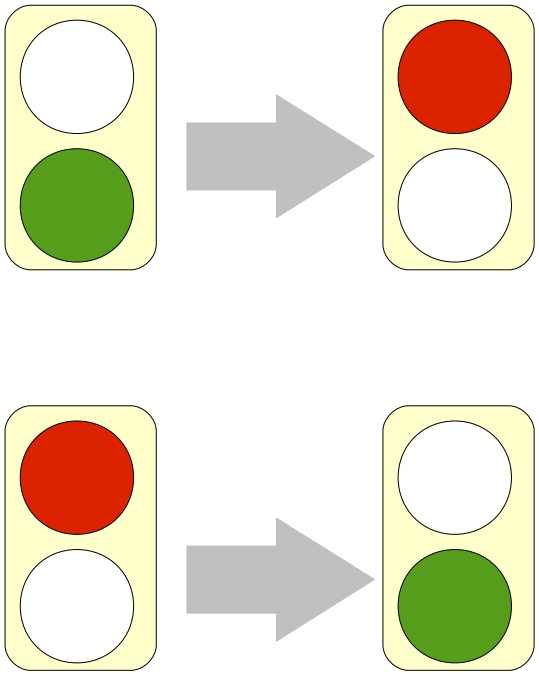
Mecanismo del semáforo

- Contexto para la sincronización de procesos

- Típicamente los semáforos se utilizan tanto para gestionar el sincronismo entre procesos como para controlar el acceso a fragmentos de memoria compartida, como por ejemplo el buffer de productos

Semáforos

```
Semaphore semaforo = new Semaphore (1);
….
        semaforo.acquire();
              int valor = cc.getN(id);
              valor++;
              cc.setN(id, valor);
         semaforo.release();
   ……
```



OJO: hay que asegurarse de que se libera el semáforo; por ejemplo, si hay excepciones

try { s.acquire(); … } finally { s.release(); }

Semáforos

package java.util.concurrent
class Semaphore - Semaphore (int permisos)

- semáforo binario: gestiona 1 permiso de acceso
  - void ***acquire()*** - void ***release()***
- semáforo general: gestiona N permisos
  - void ***acquire(int n)*** (solicita n permisos del semáforo si no hay bastantes, espero y cuendo los haya, sigo)
  - void ***release(int n)***
    - devuelvo n permisos al seméforo si hay alguien esperando, se intenta satisfacerle

Ejemplo de Prod/cons con buffer n sincronizados por semáforos

```
while (1) {
    /* produce*/
    vacio.acquire()
    mutex.acquire();
    /* guarda en el buffer*/
    mutex.release()
    lleno.release();
}
```

```
while (1) {
    lleno.acquire()
    mutex.acquire();
    /* rellena */
    mutex.release()
    vacio.release();
    /* consume*/
}
```

- **mutex**, semáforo binario, proporciona exclusión mutua para el acceso al buffer de productos. - se inicializa a 1.
- **Vacio** semáforo contador para controlar los huecos vacíos del buffer. - se inicializa a **n**, tamaño del buffer
- **lleno** semáforo contador para controlar el número de huecos llenos del buffer. - se ***inicializa a 0.***

Bloques sincronizados

- El objeto es el que se utiliza como llave para acceder a la sección crítica

```
synchronized (objeto) {
// zona de exclusión mutua
}
```

Objeto compartido

```
synchronized (cc) {
    int valor = cc.getN(id);
    valor++;
    sleep(1000);
    cc.setN(id, valor);
    }
```

Bloques y métodos Sincronizados

synchronized

- Son de alto nivel
- Delimitan una zona
- Requieren que todos colaboren
- Un método synchronized puede llamar a otro método synchronized

Métodos sincronizados

- Sincroniza todo el método

Utiliza el objeto de esa clase para sincronizar

```
public synchronized void incrementar {
throws InterruptedException {

   int valor = cc.getN(id);
   valor++;
   sleep(1000);
   cc.setN(id, valor);
 }
```

Monitores

- Mecanismo de sincronización de más alto nivel
  - protege la sección crítica
  - Garantiza que solamente pueda existir un hilo activo dentro de la misma.
- Un monitor permite suspender un hilo dentro de la sección crítica posibilitando que otro hilo pueda acceder a la misma.
- Este segundo hilo puede abandonar el monitor, liberándolo, o suspenderse dentro del monitor.
- De cualquier modo, el hilo original se despierta y continua su ejecución dentro del monitor.
- Este esquema es escalable a múltiples hilos

  (varios hilos pueden suspenderse dentro de un monitor).

Características bloques Synchronized

- Bloques synchronized
  - No garantizan la secuencia del acceso de que los hilos que esperan
  - No pueden pasar parámetros .
  - No puede hacerlo por un tiempo determinado (no tiene parámetros).
  - Debe estar contenido dentro de un método

Lectores/escritores con sincronización

Clase Libro
num_lectores *estado*
+ ***syncronized*** *leer()* +***syncronized*** *escribir()* +***syncronized*** *desbloq_lectores()* +***syncronized*** *desbloq_escritores()*

```
public synchronized void escribir(){
 if (estado==VACIO)
    estado=ESCRIBIENDO;
 /*Comprobamos si hay escritores*/
 else {
   while(estado!=VACIO)
      {try {wait();}
       catch(InterruptedException e){}
       }
 }
```

Lectores/escritores sincronizados

```
public synchronized void leer(){
 if (estado==VACIO)
    estado=LEYENDO;
 /*Comprobamos si hay escritores*/
 else if (estado!=LEYENDO)
    while(estado==LEYENDO)
      {try {wait();}
       catch(InterruptedException e){}
      }
     estado=LEYENDO;
     num_lectores++;
 }
```

```
public synchronized void escribir(){
 if (estado==VACIO)
    estado=ESCRIBIENDO;
 /*Comprobamos si hay escritores*/
 else {
   while(estado!=VACIO)
      {try {wait();}
       catch(InterruptedException e){}
       }
 }
```

Lectores/escritores sincronizados

Clase Pasiva (ej. libro)
estado
num_lectores
+ ***syncronized*** *leer()*
+***syncronized*** *escribir()*
+***syncronized*** *terminaLeer()*
+***syncronized*** *terminaEscribir()*

```
public synchronized void terminaEscribir(){
      estado=VACIO)
      notify();
```

```
public synchronized void terminaLeer(){
   num_lectores--;
   if (num_lectores ==0){
      estado=VACIO;
      notify();
   }
}
```

Lectores/escritores sincronizados

```
public synchronized void leer(){
 if (estado==VACIO)
    estado=LEYENDO;
 /*Comprobamos si hay escritores*/
 else if (estado!=LEYENDO)
    while(estado==LEYENDO)
      {try {wait();}
       catch(InterruptedException e){}
      }
     estado=LEYENDO;
     num_lectores++;
 }
```

```
public synchronized void
   terminaLeer(){
    num_lectores--;
    if (num_lectores ==0){
       estado=VACIO;
       notify();
    }
}
```

```
public synchronized void escribir(){
 if (estado==VACIO)
    estado=ESCRIBIENDO;
 /*Comprobamos si hay escritores*/
 else {
   while(estado!=VACIO)
      {try {wait();}
       catch(InterruptedException e){}
       }
```

```
public synchronized void terminaEscribir(){
      estado=VACIO)
       notify();
 }
```

BlockingQueue

- Estructuras de datos para programar en forma concurrente
- Es una interfaz de Java que contiene varios tipos de colas bloqueantes.
- Tienen flexibilidad para gestionar la cola correctamente porque:
  - Permiten poner un número determinado máximo de elementos
  - Permiten poner un timeout para las esperas en cola llena y cola vacía,

Interfaz BlockingQueue

- Clases que la implementan:
  - ArrayBlockingQueue basado en la conocida estructura de Arrays,
  - DelayQueue que permite extraer un objeto después de cierto tiempo definido,
  - LinkedBlockingQueue basado en nodos enlazados,
  - SynchronousQueue que permite que cada elemento introducido, espere a ser extraído antes de meter el siguiente. (misma idea de Rendezvouz)

BlockingQueue

- Se importa el tipo de dato correspondiente

```
import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.SynchronousQueue;

import java.util.concurrent.LinkedBlockingQueue;
```

- Se crean los objetos de esos tipos

```
BlockingQueue<Integer> deque = new LinkedBlockingDeque<Integer>(5);

BlockingQueue<Integer> deque = new ArrayBlockingQueue<Integer>(5,true);

BlockingQueue<Integer> deque = new SynchronousQueue<Integer>();
```

- Se utilizan los tipos convenientes

```
Runnable producer = new ProdColaBloqueo("ProductorCola", deque);
Runnable consumer = new ConsColaBloqueo("ConsumidorCola", deque);
```

BlockingQueue - Forma de uso

- **Productor**

```
import java.util.concurrent.BlockingDeque;
private BlockingQueue<Integer> bloCola;

run()
…..      //pone en la cola de bloqueo
  bloCola.put(i);
...
}
```

- **Consumidor**

```
import java.util.concurrent.BlockingDeque;
private BlockingQueue<Integer> bloCola;

run()
…for (int i=0;i<10;i++){
  try { j= bloCola.take();//saca de la cola de bloqueo
….}
```

- **Test**

```
import java.util.concurrent.BlockingQueue;
BlockingQueue<Integer> deque = new LinkedBlockingDeque<Integer>(5);

Runnable producer = new ProdColaBloqueo("ProductorCola", deque);
Runnable consumer = new ConsColaBloqueo("ConsumidorCola", deque);
```