.UBAfiuba
FACULTAD DE INGENIERÍA

Técnicas de Programación Concurrente I

Monitores

Ing. Pablo A. Deymonnaz
pdeymon@fi.uba.ar

Facultad de Ingeniería
Universidad de Buenos Aires

Índice

Condition Variables

Una condition variable C:

- No guarda ningún valor
- Tiene asociado un FIFO
- Consta de tres operaciones atómicas:
  - waitC(cond)
  - signalC(cond)
  - empty(cond)

Condition Variables

- waitC(cond)

```
cond.append (p)
p.state := blocked
monitor.releaseLock ()
```

- signalC(cond)

```
if ( cond <> empty )
begin
q := cond.remove ()
q.state := ready
end
```

- empty(cond)

```
return cond = empty
```

Monitores

Herramientas de sincronización que permite a los hilos tener exclusión mutua y la posibilidad de esperar (*block*) por que una condición se vuelva falsa.
Tienen un mecanismo para señalizar otros hilos cuando su condición se cumple.

Monitores

Un monitor consta de:

- Nombre
- Variables internas
- Procedimientos del monitor: rutinas que acceden directamente a las variables internas
- Una interfaz pública para que los procesos puedan acceder a las variables internas
- Inicialización de las variables internas
- Un conjunto de condition variables que incorporan sincronismo al monitor

Monitores

Los procesos pueden tomar distintos estados:

- Esperando para entrar al monitor
- Ejecutando el monitor (sólo un proceso a la vez - exclusión mutua)
- Bloqueado en FIFO de variable de condición
- Recién liberado de la wait condition
- Recién completó una operación signalC

Comparación con Semáforos

Semáforo	**Monitor**
wait puede o no bloquear	**waitC** siempre bloquea
signal siempre tiene efecto	**signalC** no tiene efecto si la cola está vacía
signal desbloquea un proceso arbitrario	**signalC** desbloquea el proceso del tope de la cola
un proceso desbloqueado con signal, puede continuar la ejecución inmediatamente	un proceso desbloqueado con signalC debe esperar que el proceso señalizador deje el monitor

Índice

Creación de hilos (I)

- Los hilos se pueden crear de dos formas:
  - Heredando de la clase `Thread`
  - Implementando la interfaz `Runnable`
- ¿Cuándo se usa cada una?
  - Desición de diseño, depende de la estructura de clases de la aplicación
  - Al implementar la interfaz `Runnable` se puede extender otra clase

Creación de hilos (II)

- Ejemplo heredando de `Thread`

```
public class HiloThread extends Thread {

public void run() {
  System.out.println("Hola, soy el hilo " +
    Thread.currentThread().getId() + " y heredo
    de Thread");
  System.out.println("Termine");
}
}
```

Creación de hilos (III)

- Ejemplo implementando `Runnable`

```
public class HiloRunnable implements Runnable {

public void run() {
  System.out.println("Hola, soy el hilo "
  + Thread.currentThread().getId() + " e impleme
  Runnable");
  System.out.println("Termine");
}
}
```

Creación de hilos (III)

- Ejemplo: programa principal

```
public class Ejemplo1 {

public static void main(String[] args) {
  HiloThread hilo1 = new HiloThread();
  Thread hilo2 = new Thread(new HiloRunnable());

  hilo1.start();
  hilo2.start();
}
}
```

Índice

Secciones críticas (I)

- Bloques *synchronized*: mecanismo propio de Java
- Dos partes:
  - Un objeto que servirá como lock
  - Un bloque de código a ejecutar en forma atómica
- Métodos *synchronized*: si un bloque de código es un método completo
- ¿Cómo funciona?
  - Cada objeto tiene un *lock* o *monitor*
  - Sólo un hilo a la vez puede tomar el lock
  - El lock es reentrante

Secciones críticas (II)

- Ejemplo de bloque *synchronized*

```
public void incrementar(int cantidad) {
  synchronized(this) {
    this.valor += cantidad;
    System.out.println("Contador: valor
        actual = " + this.valor);
  }
}
```

- Ejemplo de método *synchronized*

```
public synchronized void incrementar(int cantida
  this.valor += cantidad;
  System.out.println("Contador: valor
      actual = " + this.valor);
}
```

Secciones críticas (III)

- Ejemplo de bloque *synchronized* en método estático

```
public static void escribirMensaje(String mensaj
  synchronized(Contador.class) {
    System.out.println("Mensaje del contador");
  }
}
```

- Ejemplo de método estático *synchronized*

```
public static synchronized void escribirMensaje(
  System.out.println("Mensaje del contador");
}
```

Índice

Exclusión mutua (I)

- Los hilos participantes deben sincronizarse con el mismo objeto
- Ejemplo:

```
public static void main(String[] args) {
    Contador contador = new Contador();
    Thread hilo1 = new Thread(new Hilo(contador));
    Thread hilo2 = new Thread(new Hilo(contador));

    hilo1.start();
    hilo2.start();
}
```

Exclusión mutua (II)

- Ejemplo donde no hay exclusión mutua

```
public static void main(String[] args) {
  Contador contador1 = new Contador();
  Contador contador2 = new Contador();

  Thread hilo1 = new Thread(new Hilo(contador1))
  Thread hilo2 = new Thread(new Hilo(contador2))

  hilo1.start();
  hilo2.start();
}
```

Índice

Señalización de hilos (I)

- Se debe tener el monitor adquirido para poder llamar a los siguientes métodos:
  - Método *wait()*: libera el monitor adquirido y suspende el hilo hasta que otro hilo llame a *notify()* o *notifyAll()*
  - Método *notify()*: despierta alguno de los hilos que espera por el monitor
  - Método *notifyAll()*: despierta todos los hilos que esperan por el monitor

Señalización de hilos (II) SEÑAL PERDIDA

- Ejemplo de buffer con sincronismo:

```
public class Buffer {
  private int valor = 0;

  public synchronized int getValor() {
    try {
      wait();
    } catch (InterruptedException e) {}

    return valor;
  }

  public synchronized void setValor(int valor) {
    this.valor = valor;
    notifyAll();
  }
}
```

qué pasa si ponen un valor y después lo piden —> deadlock

Variables volatile

- Los hilos guardan los valores de las variables compartidas en sus caches
- La palabra clave *volatile* indica al compilador que el valor de la variable no debe cachearse y debe leerse siempre de la memoria principal
- De este modo, los hilos verán siempre el valor más actualizado de la variable
- La declaración de una variable como *volatile* no realiza ningún lockeo en dicha variable

Bibliografía

- **Principles of Concurrent and Distributed Programming**, M. Ben-Ari, Segunda edición (capítulo 7)
- Monitors: An Operating System Structuring Concept, C.A.R. Hoare, Communications of the ACM, 1974
- "Java Concurrency in Practice", Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes y Doug Lea
- "Effective Java", Joshua Bloch, segunda edición
- Tutorial de concurrencia de Oracle, http://docs.oracle.com/javase/tutorial/essential/concurrency/