Análisis y diseño de software

dit
UPM

Tema 2: Concurrencia /exclusión mutua /java

José A. Mañas

5.4.2017

ETSIT UPM

memoria compartida: carreras

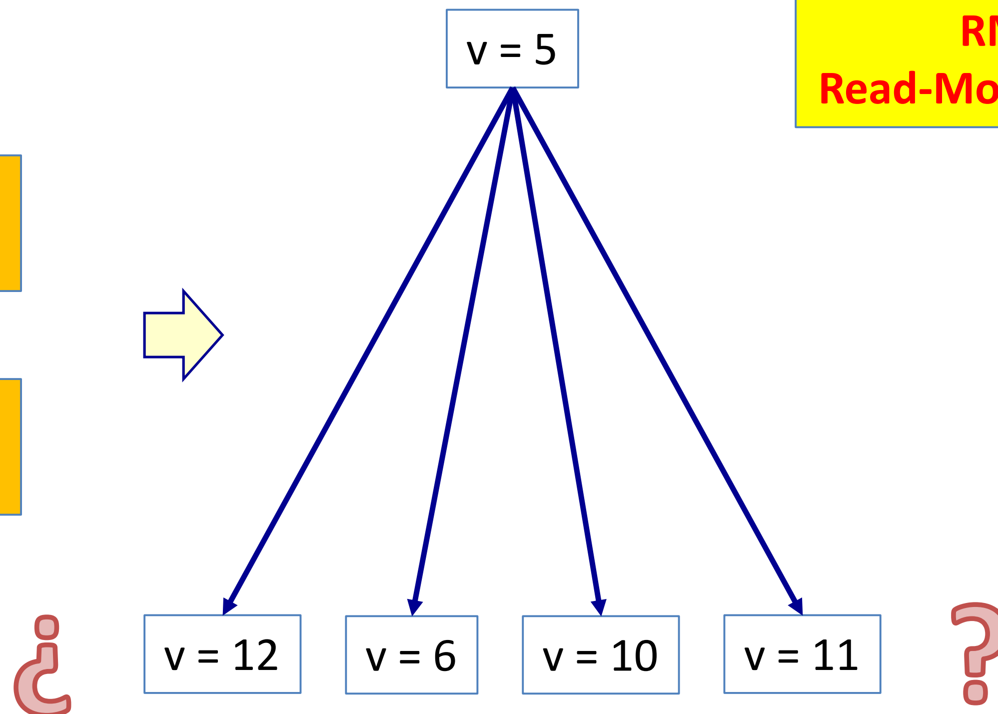
- carreras (race conditions)
  - yo me traigo el dato a mi memoria local
  - tu te llevas el dato a tu memoria local
  - modificamos nuestros datos
  - yo lo devuelvo, tu lo devuelves
  - ¿quién tiene la última palabra?

variables compartidas: carreras

RMW issues
Read-Modify-Write

thread A:
run() { v = v + 1; }

thread B:
run() { v = v * 2; }

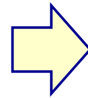


variables compartidas: carreras

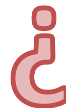
RMW issues
Read-Modify-Write

```
thread A:
run() { v = v + 1; }
```

```
thread B:
run() { w = v * 2; }
```



v = 5



v = 6	v = 6
w = 12	w = 10

índice

- acceso a variables compartidas
- carreras
- exclusión mutua
  - cerrojos (locks)
  - semáforos (semaphores)
  - synchronized
  - monitores (monitors)

variable compartida desprotegida

RMW issues
Read-Modify-Write

```
public class Contador {
   private int cuenta = 0;

   public int incrementa(int v) {
      cuenta += v;
      return cuenta;
   }

   public int decrementa(int v) {
      cuenta -= v;
      return cuenta;
   }
}
```

```
public int incrementa(int v) {
      int registro = cuenta;
      registro = registro + v;
      cuenta = registro;
      return cuenta;
   }
```

alguien puede meterse por en medio

se dice que el estado del objeto (en este caso el campo ‘cuenta’) puede verse estropeado

exclusión mutua

- zonas críticas
  - secciones del programa que sólo debe ejecutar una *thread* en un momento dado, o habría problemas
- protocolo
  - si está ocupado, espero
  - si está libre
    - entro
      - **cierro** por dentro
    - hago mis cosas – yo solito
    - salgo
      - dejo **abierto**
      - si hay alguien esperando, que entre
      - si no, queda abierto

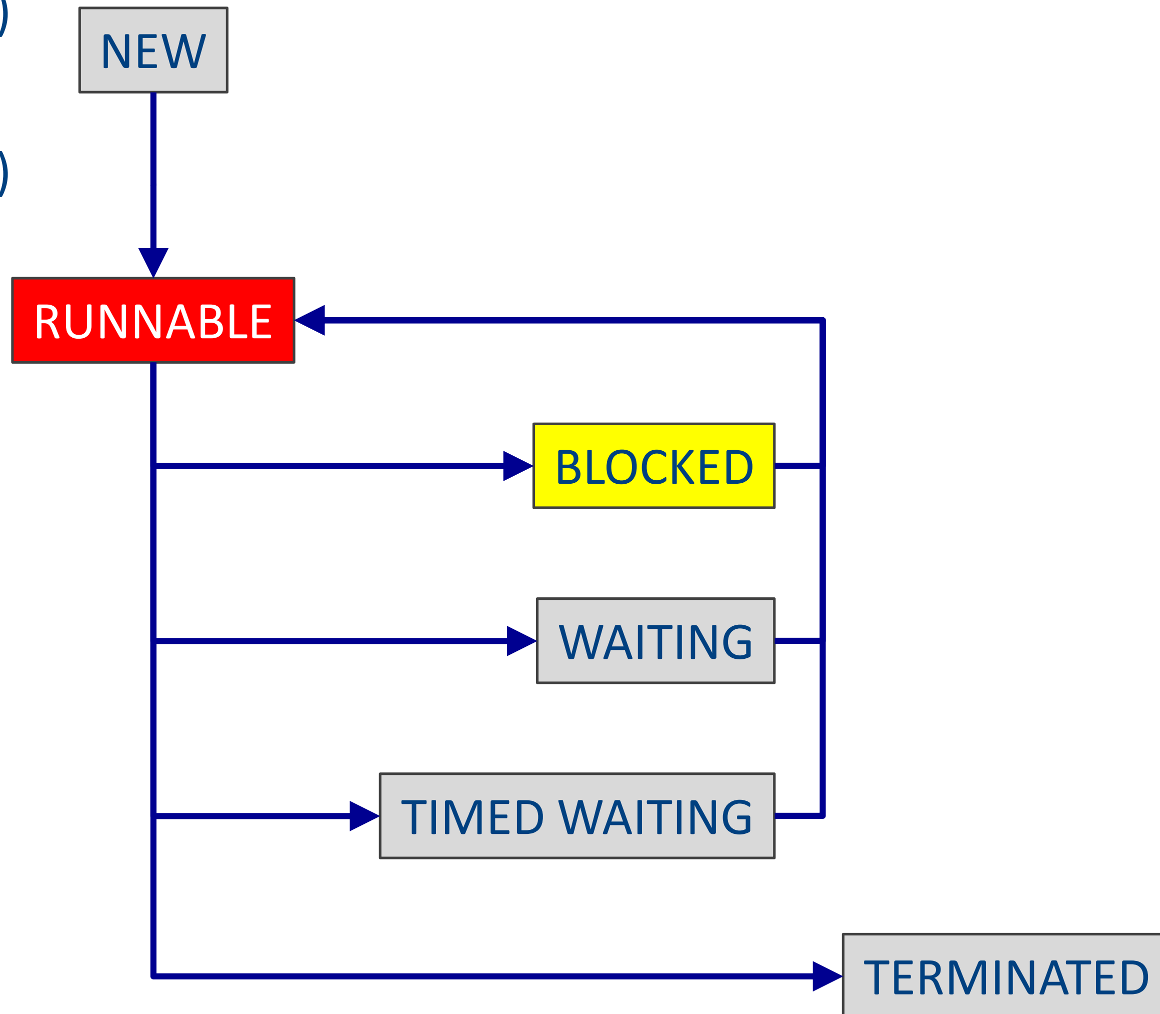
java mantiene una cola de threads esperando a que el cerrojo se abra

exclusión mutua

- zonas críticas
  - secciones del programa que sólo debe ejecutar una *thread* en un momento dado, o habría problemas
- protocolo
  - si está ocupado, espero
  - si está libre
    - entro
      - **cierro** por dentro
    - hago mis cosas – yo solito
    - salgo
      - dejo **abierto**
      - si hay alguien esperando, que entre
      - si no, queda abierto

todo esto
es indivisible
ATÓMICO

state



arquitecturas

- no estructuradas
  - *locks*
  - semáforos
- estructuradas
  - bloques *synchronized*
  - métodos *synchronized*
  - monitores

Lock

- package java.util.concurrent.locks
  - interface Lock
  - class ReentrantLock implements Lock

```
lock.lock();
  ... operaciones ...
lock.unlock();
```

```
try {
  lock.lock();
    zona exclusiva
} finally {
  lock.unlock();
}
```

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

estado compartido protegido

```
public class Contador {
    private int cuenta = 0;
    private final Lock LOCK = new ReentrantLock();
```

```
    public int incrementa(int v) {
        try {
            LOCK.lock();
            cuenta += v;
            return cuenta;
        } finally {
            LOCK.unlock();
        }
    }
```

```
    public int decrementa(int v) {
        try {
            LOCK.lock();
            cuenta -= v;
            return cuenta;
        } finally {
            LOCK.unlock();
        }
    }
```

nadie puede meterse por enmedio

estado protegido

atomicidad

- las operaciones lock() y unlock() son indivisibles
  - no se permite que una thread se vea desplazada por otra mientras se están ejecutando

- esta misma propiedad sirve para todas las formas de sincronización:
  - semáforos y bloques sincronizados

semáforo

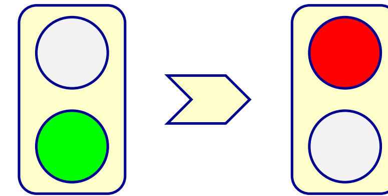
- package java.util.concurrent
  - class Semaphore
    - Semaphore (int permisos)
- semáforo binario: gestiona 1 permiso de acceso
  - void acquire()
  - void release()
- semáforo general: gestiona N permisos
  - void acquire(int n)
    - solicita N permisos del semáforo
      si no hay bastantes, espero
    - cuando los haya, sigo
  - void release(int n)
    - devuelvo N permisos al semáforo
    - si hay alguien esperando, se intenta satisfacerle

uso semáforo binario

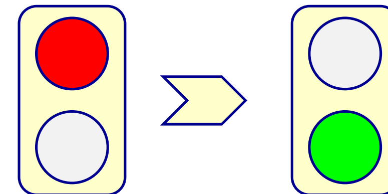
```
semaforo.acquire();

    ... operaciones ...

semaforo.release();
```



```
semaforo.acquire();
try {
    ... operaciones ...
} finally {
  semaforo.release();
}
```



uso de los semáforos

1. limitar el número de threads en la zona crítica

```
Semaphore sm = new Semaphore(5) # Max: 5-threads

InputStream fetch_page(String ref):
   sm.acquire();
   try {
      URL url = new URL(ref);
      return url.openStream();
   } finally {
      sm.release();
   }
```

```
Semaphore semaphore = new Semaphore(N)

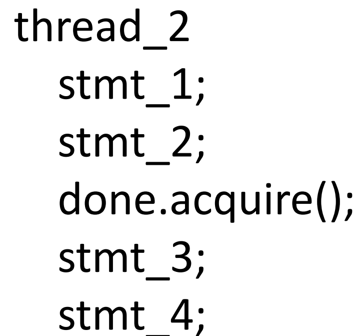
semaphore.acquire();
try {
   … zona crítica …
} finally {
   semaphore.release();
}
```

uso de los semáforos

2. coordinar threads

```
Semaphore done = new Semaphore(0)
```

```
thread_1
    stmt_1;
    stmt_2;
    done.release();
    stmt_3;
    stmt_4;
```



```
thread_2
    stmt_1;
    stmt_2;
    done.acquire();
    stmt_3;
    stmt_4;
```

ejemplo: semáforo

ejemplo

```
public class Parking {
    private final int capacidad; // número de coches que caben
    private final Semaphore semaphore;

    // constructor
    public ParkingSemaphore(int capacidad) {
        this.capacidad = capacidad;
        semaphore = new Semaphore(capacidad);
    }
```

```
    // barreras de entrada
    public void entra() throws InterruptedException {
        semaphore.acquire(1);
    }
```

```
    // barreras de salida
    public void sale() {
        semaphore.release(1);
    }
```

ejemplo: semáforo con N permisos

ejemplo

```
public class EsperaNTareas {
  public static void main(String[] args)
   throws InterruptedException {

   Semaphore contador = new Semaphore(0);
   List<Tarea> tareas = new ArrayList<Tarea>();
   tareas.add(new Tarea(contador));
   // ... N veces

   for (Tarea tarea : tareas)
       tarea.start();

   // espera a que todas acaben
   contador.acquire(tareas.size());
  }
}
```

ejemplo: semáforo con N permisos

ejemplo

```
public class Tarea extends Thread {
   private Semaphore contador;

   public Tarea( Semaphore contador ){
    this.contador = contador;
   }

   public void run() {
    // hace su tarea
    contador.release();
   }
}
```

comentarios

locks y semáforos

- son de bajo nivel
- primitivas independientes
- requieren que todos colaboren
- se puede hacer cualquier cosa, estén bien o no
  – abrir y cerrar no están necesariamente emparejados **sintácticamente**

bloques *synchronized*

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

cc: *Object* compartido

```
synchronized (cc) {
    ... operaciones ...
}
```

el cerrojo se libera al salir del bloque:

- última sentencia
- return interno
- excepción
```

estado compartido protegido

```
public class Contador {
   private int cuenta = 0;
   private final Object LOCK = new Object();
```

```
public int incrementa(int v) {
    synchronized (LOCK) {
        cuenta += v;
        return cuenta;
    }
}
```

```
public int decrementa(int v) {
    synchronized (LOCK) {
        cuenta -= v;
        return cuenta;
    }
}
```

métodos *synchronized*

- usando *this* como cerrojo
  - es lo mismo
    pero escribes menos y te equivocas menos

```
public synchronized int incrementa(int v) {
    cuenta += v;
    return cuenta;
}
```

```
public synchronized int decrementa(int v) {
    cuenta -= v;
    return cuenta;
}
```

comentarios

bloques y métodos *synchronized*

- son de alto nivel
- delimitan una zona
- requieren que todos colaboren
- un thread que ya posee un cerrojo puede entrar en zonas protegidas por el mismo cerrojo
  - esto permite que un método synchronized llame a otro método synchronized

monitores

- clases donde
  - se monitoriza toda modificación del estado
    - el estado es privado
    - todos los accesos son zonas críticas
- en java
  - todos las campos son privados
  - todos los métodos son synchronized

monitor

ejemplo

```
public class Contador {
    private int cuenta = 0;

    public synchronized int getCuenta() {
        return cuenta;
    }

    public synchronized int incrementa(int v) {
        cuenta += v;
        return cuenta;
    }

    public synchronized int decrementa(int v) {
        cuenta -= v;
        return cuenta;
    }
}
```

synchronized garantiza

- acceso exclusivo
- variables actualizadas

interbloqueo - *deadlock*

- todos los mecanismos de exclusión mutua pueden provocar situaciones de bloqueo si varios threads intentan acaparar recursos de forma desordenada
- ejemplo típico

```
thread 1:
synchronized(recurso1) {
  synchronized(recurso2) {
    // hago mis cosas;
  }
}
```

```
thread 2:
synchronized(recurso2) {
  synchronized(recurso1) {
    // hago mis cosas;
  }
}
```

deadlock prevention

- reservar de 1 vez y en orden

```
void get_in(Lock... locks) {
    Arrays.sort(locks);
    for (Lock lock : locks)
        lock.lock();
}

void get_out(Lock... locks) {
    for (Lock lock : locks)
        lock.unlock();
}
```

conceptos adicionales

- operación **atómica**
  - la que se ejecuta sin cambiar de thread
  - ejemplo
    - int n = v++;
      NO es atómica => posibles carreras
  - solución: envolverla

```
IntAtomico va = new IntAtomico();

int n = va.inc();
```

```
class IntAtomico {
  private int v;

  synchronized int inc() {
    v++;
    return v;
  }
}
```

  - véase **AtomicInteger**

Atomic...

```
public class Contador {
    private AtomicInteger saldo = new AtomicInteger(0);

    public int incrementar(int v) {
        return saldo.addAndGet(v);
    }

    public int decrementar(int v) {
        return saldo.addAndGet(-v);
    }
}
```

thread-safe objects

- se dice de clases protegidas para poder ser utilizadas por varios threads sin problemas
- o sea, que controlan su estado para que
  - si varias threads comparten descuidadamente el mismo objeto,
  - su estado interno nunca se corrompe

error ¿qué es zona crítica?

```
class TSV {
   private int v = 0;

   synchronized int get() {
      return v;
   }

   synchronized set(int v) {
      this.v = v;
   }
}
```

independientemente de que TSV
sea thread-safe

```
TSV cc = new TSV();

int valor = cc. get();
valor+= 1;
cc.set(valor);
```

```
TSV cc = new TSV()
Object lock = new Object()

synchronized (lock) {
   int valor = cc. get();
   valor+= 1;
   cc.set(valor);
}
```

cerrojos

- el mecanismo de cerrojos alrededor de zonas de acceso exclusivo
  - es un mecanismo pesimista
    - siempre me protejo por si acaso
  - puede hacer que una thread quede esperando a que otra termine
    - si la otra casca, puede esperar eternamente
    - puede ser que una tarea de alta prioridad quede esperando a que termine otra de menor prioridad
      - *priority inversión*

priority inversion

- C (prioridad baja) cierra un lock
- A (prioridad alta) quiere el lock que tiene C
- B (prioridad media) monopoliza la cpu
- resultado
  - B se lo lleva todo mientras A espera tontamente

ej: https://en.wikipedia.org/wiki/Mars_Pathfinder

wait-free algorithms

- es una alternativa al uso de cerrojos
- mecanismo optimista
    1. no pido permiso para entrar
    2. si hay alguien, pido perdón y salgo
- es más eficiente si hay pocos fallos (paso 2)
    - few contending threads
- es menos eficiente si hay que pedir perdón muchas veces
    - el compilador de java puede optimizar dinámicamente