dit
UPM

Tema 2: Sincronización condicional /wait if ... /java

José A. Mañas

5.4.2017

ETSIT UPM

índice

- espera activa
- espera a la señal
  - wait()
  - notify()
  - notifyAll()

motivación

- podemos esperar a que se de una condición
  - durmiendo
  - dando vueltas un bucle
  - ¡ojo con esperar dentro de una zona crítica!
- ¿no sería mejor poner un despertador?
  - que nos avise de que la situación ha cambiado
  - controlando las zonas críticas

espera activa vs. sincro. condicional

```
public void run() {
      while (true) {
          Packet packet;
          while (true) {
              packet = router.get();
              if (packet != null) break;
              Nap.sleep(100);
          }
          answerTo(packet);
      }
  }
```

sincronización condicional

```
while (true) {
          Packet packet = router.get();
          answerTo(packet);
      }
```

zonas críticas condicionales

- es una zona crítica = exclusión mutua
  - el que llega espera si está ocupado
- métodos de acceso protegidos por condiciones de forma que una tarea que quiere entrar tiene que satisfacer la condición
  - si no cumples, sigues esperando

java : monitores

- variables críticas → private fields
- métodos de acceso con exclusión mutua
  - public synchronized method()
- añadimos la parte condicional
  - wait()
  - notify(), notifyAll()

espera a que ...

- dentro de zonas synchronized
  - wait()
    detiene la ejecución del thread
  - notify()
    reactiva 1 thread que esté haciendo wait()
  - notifyAll()
    reactiva todos los threads que estén haciendo wait()

estructura típica

- synchronized method(...) {
  while(!condicion())
  wait();
  hago mis cosas;
  notifyAll();
  }
- java se encarga de manejar el cerrojo asociado garantizando la exclusión mutua

notify() notifyAll()

- es más eficiente notify()
  - porque sólo despierta 1 tarea
  - pero si la condición no funciona, mal vamos
- es más general notifyAll()
  - porque las despierta a todas
  - si una tarea no encuentra su condición, vuelve a esperar
    - BLOCKED → RUNNABLE → BLOCKED

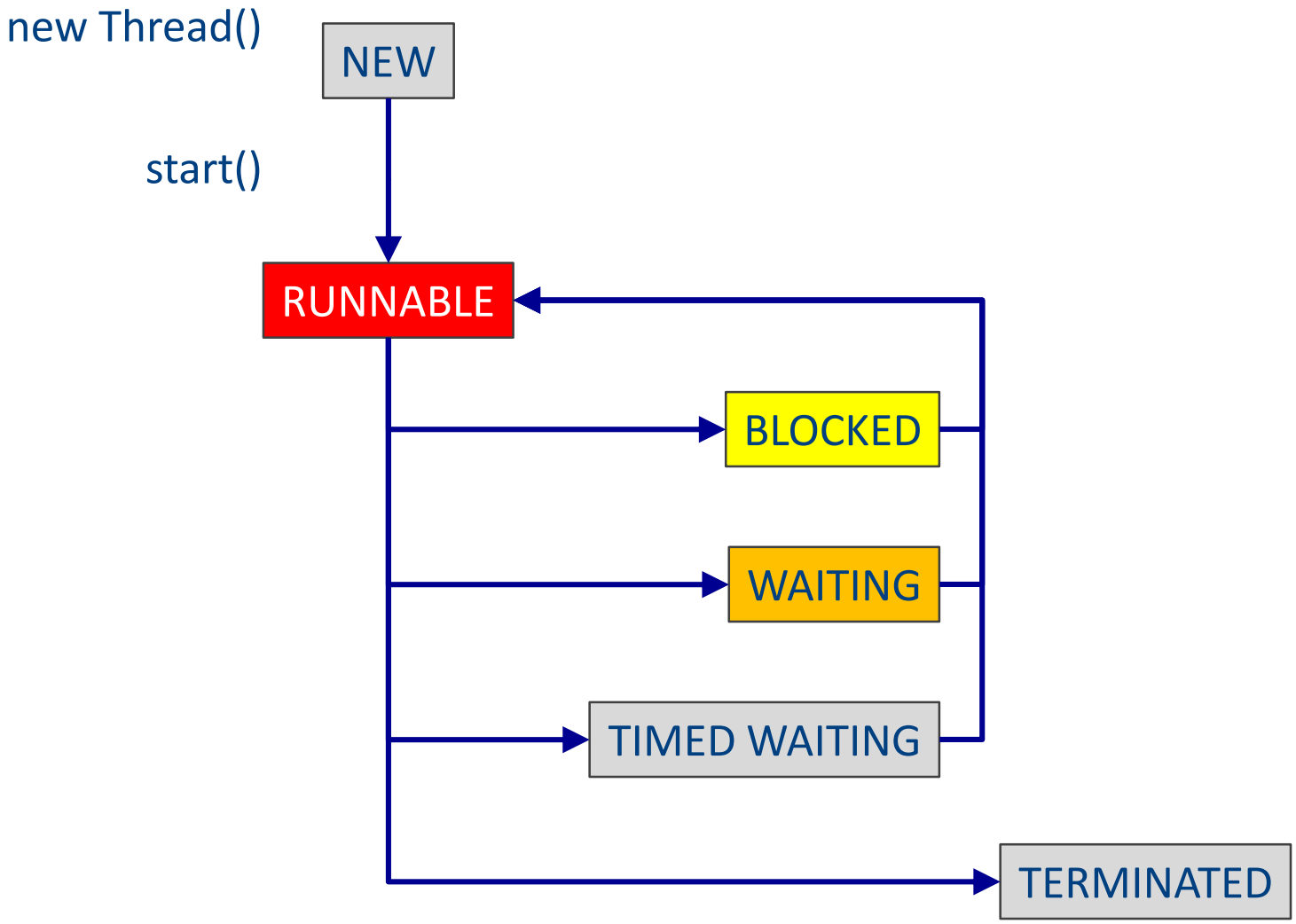
while (!condicion) wait();

- ¿por qué no basta …?

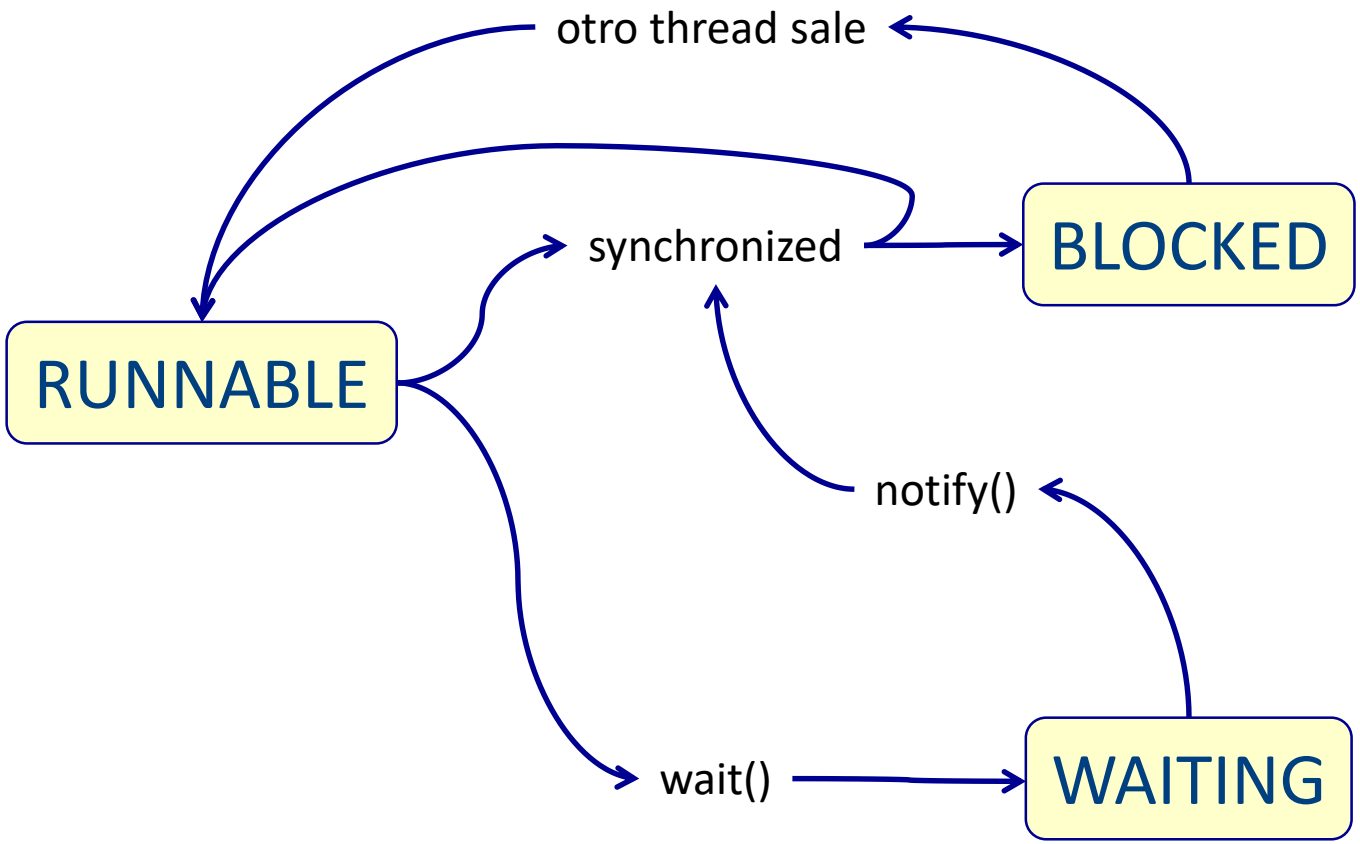
  if (!condicion) wait();

1. varias tareas con diferentes condiciones
2. si la condición es dinámica,
   puede haber variado para cuando
   este thread pase de BLOCKED a RUNNABLE
3. puede salirse del wait() por otras razones
   - misterio dependiendo de la plataforma

state


new Thread()
NEW
start()
RUNNABLE
BLOCKED
WAITING
TIMED WAITING
TERMINATED

estado del thread



estado del thread

```
synchronized ... {
   sentencias previas;
   wait();
   sentencias posteriores;
```

1. RUNNABLE → cuando llega al synchronized
   - si no puede pasar → BLOCKED
   - si puede pasar, entra
2. cuando llega al wait() → WAITING
   - otra tarea puede aprovechar para pasar de BLOCKED → RUNNABLE
3. cuando llega un notify() o notifyAll()
   - WAITING → BLOCKED
   - cuando pueda entrar,
     - BLOCKED → RUNNABLE
     - ejecuta donde se quedó: tras el wait()

Si estamos BLOCKED:
- se pasa a RUNNABLE
  cuando el thread que está dentro sale del synchronized

parking

ejemplo

```
public class Parking {
   private final int capacidad; // número de coches que caben
   private int n = 0; // número de coches que hay dentro

   // constructor
   public Parking(int capacidad) {
      this.capacidad = capacidad;
   }
```

n
estado que debemos proteger

si se corrompe
- pueden no entrar coches habiendo plazas vacías
- pueden entrar coches sin haber plazas

```
   // consulta
   public synchronized int ocupado() {
      return n;
   }
```

parking

ejemplo

```
private int n = 0; // número de coches dentro
```

```
// entra un coche por una de las puertas
public synchronized void entra(String puerta) {
    while (n >= capacidad)
        waiting();
    n++;
}
```

```
// sale un coche por una de las puertas
public synchronized void sale(String puerta) {
    n—;
    notifyAll();
}
```

```
private void waiting() {
    try {
        wait();
    } catch (InterruptedException ignored) {
    }
}
```

parking++

ejemplo

```
private int n = 0; // número de coches dentro
private boolean cerrado = false;
```

```
// cerramos el parking; los coches esperando a entrar, que se vayan
public synchronized void cerrar() {
    cerrado = true;
    notifyAll();
}
```

```
// un coche quiere entrar por una de las puertas
// devuelve TRUE si le dan plaza; FALSE si esta cerrado
public synchronized boolean entrar(String puerta) {
    while (!cerrado && n >= capacidad)
        waiting();
    if (cerrado)
        return false;
    n++;
    return true;
}
```

uso de monitores

ejemplo

- limitar el número de threads en la zona crítica

```
public class MaxIn {
   private final int MAX;
   private int n;

   public MaxIn(int max) {
      MAX = max;
      n = 0;
   }

   private void waiting() {
      try { wait(); } catch (InterruptedException ignored) { }
   }
}
```

```
public synchronized void getIn() {
   while (n >= MAX)
      waiting();
   n++;
}

public synchronized void getOut() {
   n--;
   notifyAll();
}
```

uso de monitores

ejemplo

- coordinar threads

```
public class Event {
   private boolean ready = false;

   public synchronized void waitSignal() {
      while (ready == false)
         waiting();
   }

   public synchronized void signal() {
      ready = true;
      notifyAll();
   }

   private void waiting() {
      try { wait(); } catch (InterruptedException ignored) { }
   }
}
```

uso de monitores (wait N)

ejemplo

```
class Monitor {
        private int ready;

        synchronized void done() {
            ready++;
            notifyAll();
        }

        void waitAllDone(int expected) {
            while (expected < ready)
                waiting();
        }

        private void waiting() {
            try {
                wait();
            } catch (InterruptedException ignored) {
            }
        }
    }
```

uso de monitores (wait N)

ejemplo

```
public static void main(String[] args) {
      Monitor monitor = new Monitor();

      List<Task> taskList = new ArrayList<>();
      for (int i = 0; i < 100; i++)
         taskList.add(new Task(monitor));

      for (Task task : taskList)
         task.start();

      monitor.waitAllDone(taskList.size());
   }
```

```
class Task extends Thread {
      private final Monitor monitor;

      Task(Monitor monitor) {
         this.monitor = monitor;
      }

      @Override
      public void run() {
         // ... y al acabar ...
         monitor.done();
      }
   }
```

wait() interrumpible

wait

```
public final void wait()
                throws InterruptedException
```

Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.

- mecanismo para que otra thread "supervisora" puede interrumpirla
  - th.interrumpt()
  - ... y salir de una situación indeseada

lo bueno de los monitores

- es un concepto muy limpio
- encapsula las variables críticas y
  protege todos los accesos,
  convirtiéndose en un thread-safe object

limitaciones de los monitores

- es un concepto básicamente centralizado
  - 1 cola para todas las tareas esperando
  - todas despiertan para ver si ya les toca
- choca con el principio de encapsulación
  - ¿por qué tengo que enseñar mis datos para que el monitor los proteja?
- si no protegemos 100% las variables críticas, la seguridad del conjunto depende de la buena educación de los demás

limitaciones de los monitores

- en ejercicios académicos 1 (uno) monitor es perfecto
- el programas grandes
  - 1 monitor único ➔ cuello de botella
  - N monitores pueden convertirse en una pesadilla
- nested monitor call
  - es un riesgo si un monitor llama a otro y el segundo queda bloqueado esperando que el primero libere el wait()