5 Grundlegende Thread-Steuerung

In der Praxis trifft man oft auf die Anforderung, dass mehrere Threads miteinander kooperieren müssen. Ein Paradebeispiel ist das Erzeuger-Verbraucher-Muster (*producer consumer pattern*), bei dem zwei oder mehrere Threads über eine geeignete Datenstruktur Daten austauschen. Dabei wird häufig ein FIFO-Puffer (*First In First Out*) mit einer begrenzten Kapazität benutzt. Einer der beiden Threads, der Erzeuger, füllt Information in den Puffer und der andere, der Verbraucher, entnimmt sie. Der sogenannte Ringpuffer wird typischerweise mithilfe eines Arrays und zwei Markern für den Anfang und das Ende der Schlange realisiert. Es ist offensichtlich, dass das Hinzufügen oder Entfernen eines Elements aus dem Puffer aus mehreren Schritten besteht. Während deren Ausführung darf dann keine andere nebenläufige Änderung erfolgen. Mit dem `synchronized`-Konzept kann der Puffer geschützt werden. Ein Problem tritt dabei aber auf. Wenn der Erzeuger ein neues Element in den bereits vollen Puffer einfügen will, muss er warten, bis ein Platz frei wird. Die Lösung für den Erzeuger besteht darin, sich schlafen zu legen, bis er von einem Verbraucher nach einer Entnahme geweckt wird. Umgekehrt soll sich der Verbraucher auch schlafen legen, wenn kein Element abzuholen ist. Erst wenn ein Erzeuger etwas liefert, wird er geweckt. Zu diesem Zweck werden sogenannte Bedingungsvariablen (*condition variables*) benötigt.

5.1 Bedingungsvariablen und Signalisieren

Eine Bedingungsvariable repräsentiert ein Objekt, das zusammen mit einem Lock-Objekt arbeitet. Während ein Lock-Objekt zum Schutz eines kritischen Bereichs benutzt wird, zeigt eine Bedingungsvariable ein anderes Verhalten. Zusammen mit einem Lock-Objekt dient sie zum Warten und zum Signalisieren, wenn man unter Umständen länger auf die Zuteilung einer Ressource, wie z.B. den Zugriff auf den Puffer, warten muss.

Die Methoden wait, notify und notifyAll

Zusätzlich zum reinen Ausschlussprinzip (`synchronized`) hat Java auch die Möglichkeit, Threads über Ereignisse zu synchronisieren. Die beteiligten Threads kommunizieren hierbei über einen *Vermittler*. In dem Erzeuger-Verbraucher-Beispiel ist es der Puffer. Der Puffer benutzt intern seine von `Object` geerbten Methoden `wait`, `notify` und `notifyAll`, die mit seiner Sperre zusammenarbeiten. Ein Thread muss im Besitz der zugehörigen Sperre sein, um eine dieser Methoden aufrufen zu dürfen, sonst wird eine `IllegalMonitorStateException` ausgeworfen. Diese Methoden können somit nur innerhalb von `synchronized`-Blöcken bzw. -Methoden verwendet werden.

Ein Aufruf von `wait` bewirkt Folgendes: Der aufrufende Thread trägt sich in die Warteliste (`wait`-Warteraum) des entsprechenden Objekts ein und gibt die Sperre frei. Er wechselt in den `WAITING`-Zustand und bleibt so lange darin, bis ihn ein anderer Thread durch `notify` oder `notifyAll` aufweckt. Auch ein `interrupt`-Aufruf bewirkt, dass er diesen Zustand verlässt (vgl. Abschnitt 2.3.5).

Mit `notify` wird genau ein (vom Scheduler bestimmter) Thread aus der Warteliste geweckt; mit `notifyAll` werden alle wartenden in den `RUNNABLE`-Zustand überführt. Ein wieder aktivierter Thread muss zunächst die Sperre erwerben, bevor er mit den Anweisungen (nach der `wait`-Methode) fortfahren kann.

Als Anwendungsbeispiel betrachten wir den oben beschriebenen Ringpuffer, über den Threads Daten austauschen können. Den *Ringpuffer* realisieren wir in dem Beispiel durch ein Array und drei Variablen:

```
private final Object[] data;
private int head;
private int tail;
private int count;
```

Die Variable `count` entspricht der aktuellen Anzahl der Elemente. Die Variable `tail` zeigt auf den ersten freien Platz zum Einfügen und `head` entspricht dem Index des nächsten verfügbaren Objekts. Für die Indexverwaltung wird hier die Modulo-Arithmetik eingesetzt (vgl. Abb. 5-1).

Der Zugriff auf den Ringpuffer wird durch `synchronized` geschützt. Verbraucher (Aufrufer von `get`) und Erzeuger (Aufrufer von `put`) werden über `wait` und `notifyAll` koordiniert. Eine Implementierung des Ringpuffers ist im Codebeispiel 5.1 zu sehen.

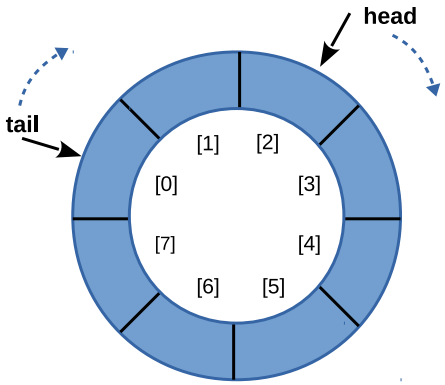

Abbildung 5-1: Ein einfacher Ringpuffer mit acht Plätzen

```
public class BoundedFIFOQueue<T>
{
  private final Object[] data;
  private int head;
  private int tail;
  private int count;

  public BoundedFIFOQueue(int cap)
  {
    data = new Object[cap];
    head = 0;
    tail = 0;
    count = 0;
  }

  public synchronized void put(T elem) throws InterruptedException
  {
    while (count == data.length)                                    ❶
    {
      wait();
    }

    count++;                                                        ❷
    data[tail] = elem;
    tail = (tail+1)%data.length;

    if (count == 1)                                                 ❸
    {
      notifyAll();
    }
  }

  public synchronized T get() throws InterruptedException
  {
    while (count == 0)                                              ❹
    {
      wait();
    }
```

```
    count--;                                                    ❺
    T obj = (T) data[head];
    data[head] = null;
    head = (head+1)%data.length;

    if (count == data.length-1)                                 ❻
    {
      notifyAll();
    }

    return obj;
  }
}
```

Codebeispiel 5.1: Ringpuffer zum Datenaustausch von Erzeugern und Verbrauchern

Ein Aufrufer der `put`-Methode prüft zuerst, ob noch Platz im Ringpuffer verfügbar ist. Falls die Kapazität erschöpft ist (`count == data.length`), wird er (der zugehörige Thread) in den Wartezustand versetzt (❶). Ist noch ein Platz vorhanden, wird der interne Zähler `count` um eins erhöht, das übergebene Element `elem` eingefügt und der Einfügeindex `tail` vorgerückt (❷). Bevor die Methode verlassen wird, werden ggf. blockierte Threads benachrichtigt, falls der Kapazitätszähler `count` den Wert 1 besitzt (❸). Das ist immer dann der Fall, wenn ein Element in einen leeren Ringpuffer abgelegt wird.

Bei der `get`-Methode prüft der Abholer zuerst, ob Elemente vorhanden sind. Ist dies nicht der Fall (`count == 0`), wird er in den Wartezustand versetzt (❹). Sind Elemente verfügbar, wird `count` entsprechend angepasst, das am längsten gehaltene Objekt entnommen und der Leseindex `head` vorgerückt (❺). Es werden ggf. blockierte Threads benachrichtigt, falls aus einem vollen Ringpuffer ein Element entfernt wurde (❻).

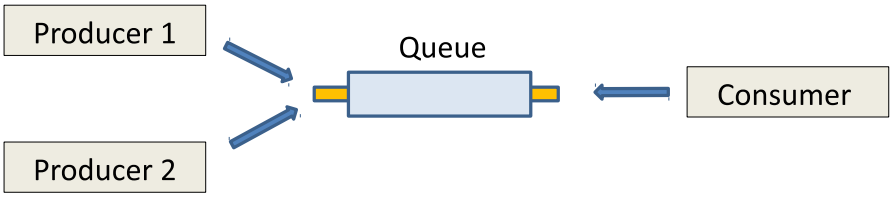

Abbildung 5-2: Ein Ringpuffer (Queue) mit zwei Erzeugern und einem Verbraucher

Das folgende Beispiel zeigt die Verwendung der Klasse `BoundedFIFOQueue`. Der Erzeuger schreibt 100 `Integer` in die ihm übergebene `BoundedFIFOQueue` (Codebeispiel 5.2).

```
class Producer implements Runnable
{
  private BoundedFIFOQueue<Integer> queue;

  public Producer(BoundedFIFOQueue<Integer> queue)
  {
    this.queue = queue;
  }

  @Override
  public void run()
  {
    try
    {
      for (int i = 0; i < 100; i++)
        queue.put(ThreadLocalRandom.current().nextInt(100));
    }
    catch (InterruptedException e)
    {
      // kann ignoriert werden
    }
  }
}
```

Codebeispiel 5.2: Erzeuger mit einem Ringpuffer

Der Verbraucher liest so lange `Integer` aus der ihm übergebenen Queue, bis er durch ein Interrupt-Signal gestoppt wird. Wartet der `Consumer`-Thread in der `get`-Methode (❶), wird er durch den Interrupt aus der `wait`-Methode herausgeholt (vgl. Codebeispiel 5.3).

```
class Consumer implements Runnable
{
  private BoundedFIFOQueue<Integer> queue;
  private int count = 0;

  Consumer(BoundedFIFOQueue<Integer> queue)
  {
    this.queue = queue;
  }

  @Override
  public void run()
  {
    try
    {
      while (Thread.currentThread().isInterrupted() == false)
      {
        System.out.println(queue.get());                    ❶
        count++;
      }
    }
```

```
    catch (InterruptedException exce)
    {
      // kann ignoriert werden
    }
    System.out.println("Anzahl: " + count );
  }
}
```

Codebeispiel 5.3: Verbraucher mit einem Ringpuffer

Das folgende Codebeispiel 5.4 erzeugt ein `BoundedFIFOQueue`-, zwei `Producer`- und ein `Consumer`-Objekt (vgl. Abb. 5-2).

```
public class BoundedFIFOQueueTest
{
  public static void main(String[] args) throws InterruptedException
  {
    BoundedFIFOQueue<Integer> queue = new BoundedFIFOQueue<>(10);

    Producer producer1 = new Producer(queue);
    Producer producer2 = new Producer(queue);
    Consumer consumer  = new Consumer(queue);

    Thread p1 = new Thread(producer1,"Producer1");
    Thread p2 = new Thread(producer2,"Producer2");
    p1.start();
    p2.start();

    Thread c = new Thread(consumer,"Consumer");
    c.start();

    // Warte auf das Ende der Erzeuger
    p1.join();
    p2.join();

    // Warte kurz , dann wird der Verbraucher gestoppt
    TimeUnit.MILLISECONDS.sleep(100);
    c.interrupt();
  }
}
```

Codebeispiel 5.4: Test mit zwei Erzeugern und einem Verbraucher

Java stellt jedem Objekt nur eine Bedingungsvariable zur Verfügung. In manchen Situationen wäre es aber für eine effiziente Implementierung von Vorteil, wenn man pro Sperre mehrere Bedingungsvariablen hätte. Wir werden in Kapitel 8 sehen, dass die mit Java 5 eingeführten `Lock`-Objekte diese Limitierung aufheben.

5.2 Regeln zum Umgang mit wait, notify und notifyAll

Beim Einsatz von `wait`, `notify` und `notifyAll` sollten folgende Punkte unbedingt beachtet werden:

- Sind mehrere Threads im `wait`-Warteraum, ist nicht festgelegt, welcher der wartenden nach `notify` bzw. `notifyAll` als nächster die Sperre erhält.
- Nach `notify` und `notifyAll` wird der aufrufende Thread seine Arbeit fortsetzen.
- Warten mehrere Threads auf dieselbe Bedingungsvariable, kann es vorkommen, dass sich die Situation für den wieder aktivierten Thread geändert hat (zum Beispiel hat ein Erzeuger die Queue wieder voll geschrieben). Daher muss die Bedingung erneut geprüft werden, d.h., `wait` sollte immer in einer Schleife aufgerufen werden:

```
while (Bedingung nicht erfüllt)
{
  wait();
}
```

  und nicht in einer einfachen Abfrage:

```
if(Bedingung nicht erfüllt)  // FALSCH !!
{
  wait();
}
```

  Die Prüfung der Bedingung in einer `while`-Schleife sollte auch deshalb gemacht werden, da JVM-Implementierungen auch ein *spurious wakeup*, also ein zufällig unerwünschtes Wecken, durchführen können. Obwohl weder `notifyAll` noch `interrupt` aufgerufen wurde, kann die JVM einen Thread wecken [17].
- Wird ein Thread, der aufgrund von einem `sleep`-, `join`- oder `wait`-Aufruf blockiert wurde, durch `interrupt` geweckt, wird eine `InterruptedException` ausgeworfen und das Interrupt-Flag wird gelöscht. Dagegen gibt es keine Ausnahme, wenn der Thread bei `synchronized` auf die Freigabe der Sperre wartet. Zu beachten ist, dass der Thread ggf. die Sperre erlangen muss, bevor der `catch`-Block ausgeführt wird.

- Der folgende Code:

```
while (!condition)
{
  try
  {
    wait();
  }
  catch (InterruptedException ex)
  {
  }
}
```

  ist nicht zu empfehlen, weil die Unterbrechung ignoriert wird. Es ist besser, die Ausnahme weiter an den Aufrufer zu geben, wie das in den Methoden `put` und `get` im Codebeispiel 5.1 gemacht wird.
- Auch wenn die `InterruptedException` behandelt wird, kann es zu Fehlverhalten kommen. Das folgende Beispiel zeigt die Implementierung einer (fehlerhaften) »Startlinie«, an der Threads durch den Aufruf von `halt` gestoppt werden können. Durch den Aufruf von `go` werden alle an der Startlinie wartenden Threads geweckt und sie können danach loslaufen. Man beachte, dass die Unterbrechung eines Threads hier (falsch) behandelt wird, weil der Interrupt-Status wieder gesetzt wird.

```
class BadStartingLine
{
  private boolean haltCondition = true;

  public synchronized void halt()
  {
    while( this.haltCondition )
    {
      try
      {
        this.wait();
      }
      catch (InterruptedException e)
      {
        Thread.currentThread().interrupt();
      }
    }
  }

  public synchronized void go()
  {
    haltCondition = false;
    notifyAll();
  }
}
```

Die Startlinie funktioniert nicht zuverlässig! Wurde bei einem Thread bereits vor dem Eintritt in `halt` seine `interrupt`-Methode aufgerufen, kommt es zu einer Endlosschleife. Beim Aufruf von `wait` wird aufgrund des Unterbrechungsstatus (`true`) sofort die `InterruptedException` ausgelöst. Nach der falschen Behandlung beginnt die Schleife von vorne. Die Methode `halt` sollte hier die `InterruptedException` nicht abfangen, sondern sie an den Aufrufer weitergeben.

- Wenn ein Thread nach `wait(milliseconds)` wieder aktiv ist, muss die Bedingung nicht erfüllt sein. Daher muss sie erneut geprüft werden und die Methode wird ggf. verlassen. Zum Beispiel wird ein Element weggeworfen, wenn es nach einer bestimmten Dauer nicht bearbeitet wird oder wenn es nach einem einmaligen Warten nicht erfolgreich abgeliefert wird. Eine Beispielimplementierung hierfür ist folgende:

```
public synchronized boolean put(T elem, long millis)
                                    throws InterruptedException
{
  if (count == data.length)
  {
    // Queue ist voll, warten mit wait
    // Dabei wird die Sperre implizit freigegeben
    wait(millis);
    // Entweder timeout oder der Thread wurde geweckt
    if (count == data.length)
    {  // Kein Erfolg
      return false;
    }
  }

  // Ab hier ist garantiert, dass
  // count < data.length UND das Objekt geschützt ist
  count++;
  data[tail] = elem;
  tail = (tail+1)%data.length;

  if (count == 1)
  {
    notifyAll();
  }
  return true; // erfolgreich abgeliefert
}
```

- Wenn mehrere Threads aufgrund eines `wait`-Aufrufs warten, soll `notify` vermieden werden. Ansonsten kann es zu einer Wettlaufsituation zwischen `notify` und `interrupt` kommen. Nach dem Wecken durch `notify` wird der Thread versuchen, in den Besitz der Sperre zu gelangen. Wenn aber in der Zwischenzeit ein `interrupt` stattfindet, wird er eventuell aufgrund seiner Implementierung die Sperre freigeben und die

Methode verlassen: Keiner der noch wartenden Threads wird geweckt. Die Wirkung von `notify` geht verloren. Ein Deadlock ist dadurch möglich:

```
public synchronized void method() throws InterruptedException
{
  while( this.haltCondition )
  {
     this.wait();
  }

  // Weitere Aktionen

  // Kann verloren gehen
  this.notify();
}
```

5.3 Zusammenfassung

Mit `wait`, `notify` und `notifyAll` ist das Konzept der Bedingungsvariablen in Java umgesetzt. Damit können Threads auf bestimmte Ereignisse warten, ohne dabei die Rechenressourcen zu belegen. Threads können durch `wait` schlafen gelegt werden und schlafende Threads können mit `notifyAll` geweckt werden. Die Methode `notify` sollte in der Regel nicht benutzt werden. Alle dieser Methoden dürfen nur in Zusammenhang mit der durch `synchronized` erlangten Sperre verwendet werden.

Für einfache Anwendungen ist das fest in der Sprache definierte Konzept ausreichend. Für komplexere Kommunikationen sind feinere Abstimmungen notwendig, die wir in Kapitel 8 besprechen werden.