Threads in Java

Übersicht

Threads sind parallele Abläufe, die auf einen gemeinsamen Prozessraum zugreifen. Um mit Threads arbeiten zu können, müssen folgende Mechanismen bereitgestellt werden:

1. Zustandsmodell für Threads
2. Kommunikation zwischen Threads
3. Koordination des Zugriffs auf den gemeinsamen Prozessraum

Java stellt in seinem Sprachumfang alle Mechanismen zur Verfügung.

Ein Thread ist ein aktives Objekt. Zur Erzeugung gibt es zwei Möglichkeiten. Entweder wird ein Objekt einer von `java.lang.Thread` abgeleiteten Klasse erzeugt. Oder es wird ein Objekt der Klasse `java.lang.Thread` erzeugt, dem als Parameter ein Interface-Objekt vom Typ `java.lang.Runnable` übergeben wird. In beiden Fällen wird eine Methode `run` definiert, die den Programmablauf des Thread darstellt.

Bei Aufruf der `run` Methode des Thread Objektes wird in der JVM ein neuer Abwickler (Execution Engine) für diesen Thread erzeugt. Dieser wird (quasi) nebenläufig zu den bereits bestehenden Abwicklern ausgeführt[1]. Unterstützt die Plattform, auf der die JVM läuft die Verteilung auf mehrere Rechnerkerne, so laufen die Threads tatsächlich parallel.

Zustandsmodell für Threads

Die folgende Graphik stellt das Zustandsmodell für Threads dar (s. Abbildung 1). Grün gekennzeichnete Übergänge werden vom Thread selbst ausgelöst; die anderen Übergänge werden durch Aktionen von außerhalb gesteuert. Fett gedruckte Ausdrücke sind implizite Ereignisse in der JVM (also nicht durch Aufruf einer Methode hervorgerufen).

Ein Thread ist nach Erzeugung zunächst im Zustand `New`. Durch Aufruf der Methode `start` aus einem anderen Thread heraus (z.B. in der `main`-Methode, typischerweise durch den Thread, der ihn erzeugt hat) wird der Thread gestartet. Dies bedeutet, dass er in den Zustand `Ready to run` versetzt wird und am Scheduling teilnimmt.

Die Priorität von Threads kann mit Hilfe der Methoden `setPriority` verändert werden.

Die JVM ist solange aktiv, wie noch mindestens ein Thread aktiv ist. Eine Ausnahme bilden die Hintergrundprozesse (Daemon Thread). Diese stellen Hilfsprogramme dar, die anderen Threads zuarbeiten. Diese Eigenschaft kann einem Thread mit Hilfe der Methode `setDaemon` zugewiesen werden. Hintergrundprozesse laufen i.a. als Endlosschleifen, die von anderen Threads via Auftragswarteschlagen o.ä. Arbeitsaufträge zugewiesen bekommen.

[1] Beim Start eines Java Programms wird die `main` Methode aufgerufen. Für diesen Ablauf wird der erste Abwickler angelegt. Ein Thread entspricht einem neuen Hauptprogramm, das seine `run` Methode anstelle eines `main` ausführt.

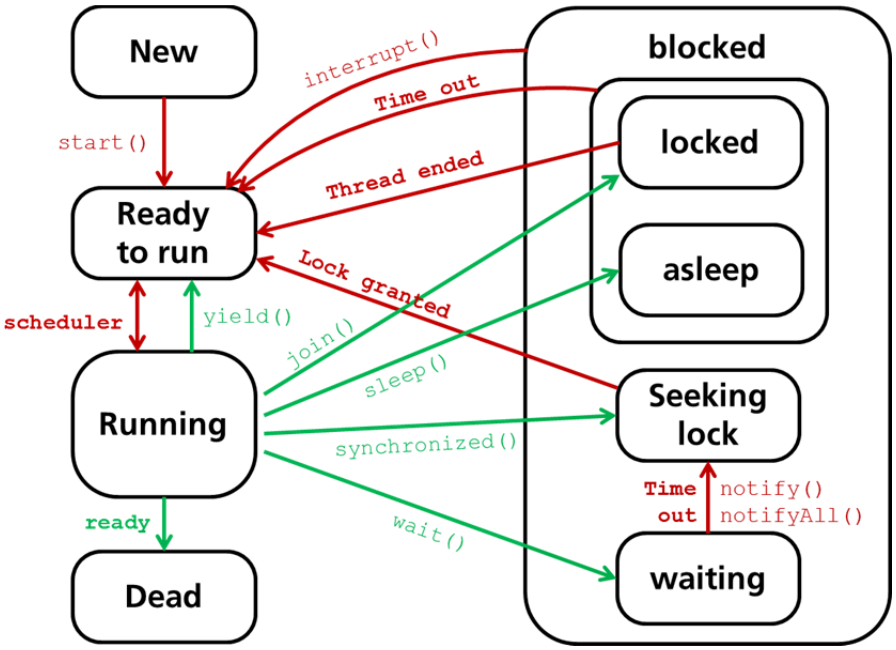

Abbildung 1: Zustandsmodell für Threads

Der Scheduler wählt einen laufbereiten Thread aus[2] und teilt ihm Rechenzeit zu. Der Thread ist dann im Zustand `Running` und arbeitet seinen Code ab. Diesen Zustand kann der Thread auf verschiedene Arten verlassen.

1. Der Thread wird beendet.
   Sobald der Thread beendet wird (d.h. die Methode `run` wird verlassen) ist der Thread beendet. Das Thread-Objekt ist weiterhin im Speicher vorhanden und befindet sich im Zustand `Dead`. Ein Thread in diesem Zustand kann nicht reaktiviert werden. Ein Versuch, die Methode `start` aufzurufen führt zum Werfen einer `IllegalThreadStateException`.
2. Der Thread möchte auf einen geschützten Bereich zugreifen.
   Wie im Folgenden noch ausgeführt wird stellt Java Mittel bereit, den Zugriff auf Ressourcen aus verschiedenen Threads heraus zu koordinieren. Dies geschieht durch das Monitorkonzept; konkret bedeutet das, dass der Thread einen als `synchronized` gekennzeichneten Code auszuführen versucht. Hierzu muss er zunächst die entsprechende Ressource (das Synchronisationsobjekt) zugeteilt bekommen und wird daher so lange in den Zustand `seeking lock` versetzt, bis die Ressource freigegeben wurde.
3. Der Thread wartet auf ein Ereignis.
   Dieser Mechanismus stellt eine Erweiterung des Monitorkonzepts dar. Der Thread befindet sich in einem Monitor (Code, der als `synchronized` gekennzeichnet ist) und möchte den Monitor zurückgeben, bis ein anderer Thread ihm diesen wieder zurückgibt. Dazu ruft der Thread die Methode `wait` des Synchronisationsobjektes auf. Er wird daraufhin dem Wait-Pool des Synchronisationsobjektes hinzugefügt und in den Zustand `waiting` versetzt. Diesen verlässt er dann, wenn ein anderer Thread die `notify` oder `notifyAll` Methode des Synchronisationsobjektes aufruft. Jetzt kann der Thread wieder um das Synchronisationsobjekt konkurrieren und wird entsprechend in den Zustand `seeking lock` versetzt.

[2] Im Fall von Mehrkernsystemen können dies auch mehrere Threads parallel sein.

4. Der Thread legt sich schlafen.
   Durch den Aufruf der Klassenmethode `Thread.sleep()` kann sich ein Thread schlafen legen. Nach Ablauf der (in Millisekunden anzugebenden) Zeit wird der Thread wieder reaktiviert.
5. Der Thread wartet auf die Beendigung eines anderen Thread.
   Hierzu ruft der Thread die Methode `join` des anderen Thread auf und wird solange blockiert, bis der entsprechende Thread beendet wurde. Die Übergangsbedingung in der Graphik (`Thread ended`) bezieht sich also nicht auf den Thread selbst, sondern auf den Thread, auf dessen Beendigung gewartet wurde.

Die drei Zustände `locked`, `asleep` und `waiting` können auch durch einen Timeout verlassen werden. Die entsprechende Zeitspanne muss jeweils bei Aufruf spezifiziert werden.

Sämtliche blockierenden Zustände werden durch den Empfang eines Interrupt-Ereignisses verlassen. Ein Thread empfängt ein Interrupt-Ereignis, sobald ein anderer Thread die Methode `interrupt` aufruft. Der empfangende Thread wird (falls er in einem blockierten Zustand ist) sofort reaktiviert und setzt seinen Ablauf mit der Behandlung der ausgelösten `InterruptedException` fort. Empfängt ein Thread einen Interrupt, bevor er blockiert wird, so führt der nächste Aufruf einer blockierenden Aktion (`join`, `sleep`, `wait`) nicht zum Blockieren. Stattdessen wird direkt die Exception ausgelöst.

Kommunikation zwischen Threads

Threads können untereinander durch Ereignisse, durch Interrupts oder mittels (normaler) Variablen und Methoden kommunizieren.

Der Ereignismechanismus mittels `wait`-Methode stellt eine Erweiterung des Monitorkonzeptes dar. Das (präzisierte) Zustandsmodell für Threads ist in Abbildung 2 dargestellt und soll anhand eines Beispiels erläutert werden.

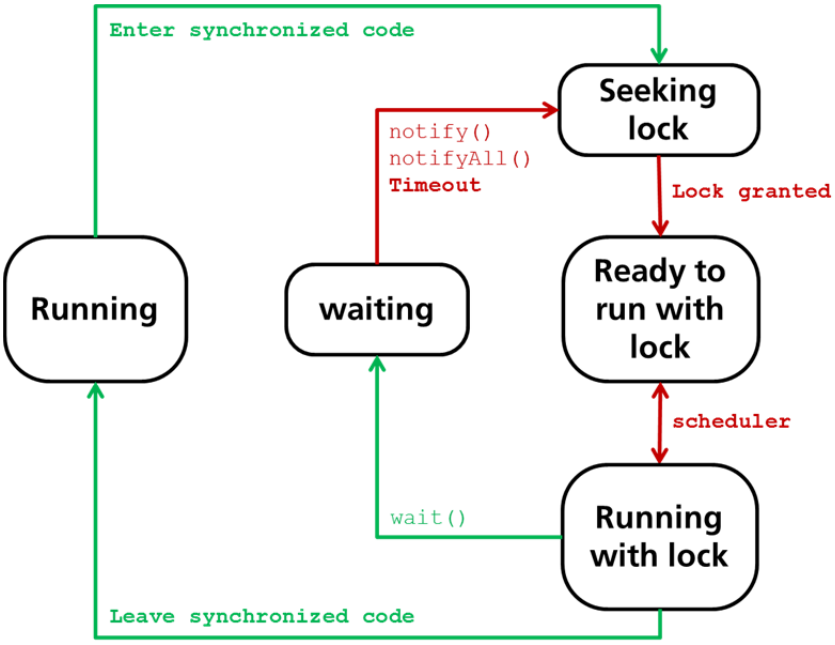

Abbildung 2: Zustandsmodell im Monitor

Gegeben seien eine Warteschlage (Message-Queue) und zwei Threads. Der eine Thread stellt in der Warteschlange Nachrichten bereit, die der andere Thread dort entnimmt und verarbeitet (Producer/Consumer). Der Zugriff auf die Warteschlange erfolgt durch die beiden Methoden `addItem()` und `readItem()`. Diese müssen im Monitor ausgeführt werden, da beide modifizierend auf die innere Datenstruktur der Warteschlange zugreifen. Eine Beispielimplementierung findet sich im Anhang.

Es wird nun die Situation betrachtet, dass die Warteschlange leer ist (d.h. es liegt keine zu verarbeitende Nachricht im Puffer). Der lesende Thread greife jetzt lesend auf die Warteschlange zu. Dazu ruft er die (synchronisierte) Methode `readItem` auf. Entsprechend konkurriert er jetzt um die Ressource Warteschlange (Zustand `Seeking Lock`). Sobald er die Ressource zugeteilt bekommt (Zustand `Ready to run with lock`) und vom Scheduler aktiviert wird (Zustand `Running with lock`) befindet er sich im Monitor und kann die Methode ausführen. Da der Inhalt der Warteschlange leer ist, kann er nicht weiterlaufen und muss warten. Problematisch ist nun, dass er erst dann weiterlaufen kann, wenn der schreibende Thread eine Nachricht hinterlegt hat. Dazu benötigt dieser aber den Zugriff auf die Ressource Warteschlange, die der lesende Thread gerade hält.

Eine mögliche Lösung wäre die, dass der lesende Thread die Ressource unverrichteter Dinge wieder frei gibt und zyklisch nachsieht, ob eine Nachricht vorhanden ist (Polling). Diese Lösung ist sehr ineffizient.

Die in Java vorgesehene Lösung ist der `wait` Mechanismus. Hierzu legt sich der lesende Thread schlafen, indem er die `wait` Methode aufruft. Dies bedeutet, dass

- der aufrufende Thread die zum Monitor gehörende Ressource (hier: die Warteschlange) freigibt,
- der aufrufende Thread angehalten wird (Zustand `waiting`) und
- der aufrufende Thread wieder laufbereit wird, sobald ein anderer Thread ihn durch Aufruf von `notify` über eine Änderung an der Ressource benachrichtigt (hier: Änderung an der Warteschlange).

Das Warten findet also immer bezüglich einer Ressource statt. Threads können an verschiedenen Ressourcen warten; die Benachrichtigung geschieht analog auch immer nur bezüglich der gerade belegten Ressource statt.

Kritisch ist die Tatsache, dass der Thread mitten in einem Monitor die Ressource freigibt. Um Inkonsistenzen zu vermeiden wird der wartende Thread nach Empfang von `notify` nicht direkt in Laufbereitschaft versetzt. Vielmehr konkurriert er jetzt wieder mit anderen Threads um die Ressource wie bei Betreten des Monitors (Aufruf einer synchronisierten Methode).

Ruft der schreibende Thread die Methode `addItem` auf während der lesende Thread wartet, so erhält er die Ressource. Er kann eine Nachricht in die Warteschlange einfügen. Anschließend ruft er (noch im Monitor!) die Methode `notify` auf. Dadurch wird wie oben ausgeführt der wartende Thread benachrichtigt. Befindet sich zu diesem Zeitpunkt kein Thread im Zustand `waiting`, so geht die Benachrichtigung ins Leere; sie wird nicht gespeichert.

Der Aufruf von `notify` führt dazu, dass genau ein wartender Thread reaktiviert wird. Welcher Thread genau reaktiviert wird liegt im Belieben der JVM. Möchte man sicherstellen, dass alle wartenden Threads benachrichtigt werden, so muss `notifyAll` aufgerufen werden.

Der `wait` Mechanismus kann mit einer maximalen Wartezeit (Timeout) versehen werden. Erfolgt innerhalb der angegebenen Zeitspanne keine Benachrichtigung, so geht der Thread automatisch in den Zustand `Seeking Lock` über.

Der Interruptmechanismus wurde im vorhergehenden Abschnitt bereits dargestellt.

Die dritte Möglichkeit der Kommunikation zwischen Threads ist der Austausch mittels (normaler) Variablen. Da alle Java-Threads einer JVM auf demselben Prozessraum arbeiten, stehen alle öffentlichen Objekte im Zugriff der Threads. Dies gilt insbesondere auch für Methoden und Variablen der (abgeleiteten) Threadklassen bzw. des Runnables selbst.

Ressourcenschutz

Für den koordinierten Zugriff auf Ressourcen sieht Java das Monitorkonzept vor. Ein Monitor wird immer bezüglich eines Objektes definiert. Ein Objekt kann hierbei jedes beliebige Objekt im Prozessraum der JVM sein, insbesondere auch ein Klassenobjekt.

Ein geschützter Code-Abschnitt wird geklammert und als `synchronized` gekennzeichnet. Dabei muss die Ressource angegeben werden, bezüglich derer die Koordination stattfindet. Im folgenden Code findet eine Synchronisation auf das Objekt `someObject` statt. Sobald ein Thread an dieser Stelle anlangt konkurriert er um die Ressource `someObject`. Hält ein anderer Thread bereits diese Ressource, so wird der aktuelle Thread blockiert. Er wird erst dann wieder lauffähig, wenn er die Ressource erhält. Sobald er den `synchronized` Bereich betreten hat hält der Thread die Ressource. Bei Verlassen des Bereichs gibt er sie wieder zurück; ein eventuell wartender Thread kann jetzt aktiviert werden.

```
synchronized (someObject) {
        /**
         * Now we have access to the lock and can do our modifications.
         */
        someObject.doSomeModification();
}
```

Muss eine Methode vollständig hinsichtlich des Objektes, zu dem sie gehört synchronisiert werden, so deklariert man die Methode als `synchronized` (s. Anhang `addItem` und `readItem`).

Klassenmethoden können sich nur auf globale Objekte synchronisieren. Im Allgemeinen wird dies das entsprechende Klassenobjekt sein. Eine `static synchronized` Methode blockiert also das Klassenobjekt – nicht jedoch Instanzen!

Anhang A

Listing für das Beispiel Message Box

```
import java.util.Vector;

public class PostBox<T> {
	private Vector<T> queue;
	private int readIdx;
	private int writeIdx;

	public PostBox(int c) {
		/**
		 * First we check if the capacity is at least 2. If not we set it to 2.
		 * This is not very safe code! We implicitly change some property
		 * without giving feedback to the user!
		 */
		if (c < 2)
			c = 2;
		queue = new Vector<T>();
		queue.setSize(c);
		/**
		 * Initialize the read and write position. The write idx marks the
		 * element that will be written next. The read idx marks the element
		 * that will be read next. Both indices must never overtake each other.
		 */
		readIdx = 0;
		writeIdx = 0;

	}

	public synchronized void addItem(T item) {
		while (!canWrite()) {
			/**
			 * The queue is full. We need to wait until someone tells us to wake
			 * up. We then still have to check if the write access is available
			 * because some other thread might have come in between.
			 */
			try {
				wait();
			} catch (InterruptedException e) {
				/**
				 * Do nothing if interrupted
				 */
				;
			}
		}

		/**
		 * Now we have access to the queue and it is not full. So we write.
		 */
		queue.set(writeIdx, item);
		/**
		 * We notify that the queue has something to read in. If we wouldn't do
		 * other threads would wait endlessly.
		 */
		notify();
		writeIdx = (writeIdx + 1) % (queue.size());
	}

	public synchronized T readItem() {
		T result = null;
		while (!canRead()) {
			/**
			 * The queue is empty. We need to wait until some tells other to
			 * wake up. We then still have to check if the read access is
			 * available because some other thread might have come in between.
			 */
			try {
				wait();
			} catch (InterruptedException e) {
				/**
				 * Do nothing if interrupted
				 */
				;
			}
		}
```

```
                /**
                 * Now we have access to the queue and it is not empty. So we take the
                 * next item.
                 */
                result = queue.get(readIdx);
                /**
                 * We notify that the queue has space to write in again. If we wouldn't
                 * do other threads would wait endlessly.
                 */
                notify();
                readIdx = (readIdx + 1) % (queue.size());

                return result;
        }

        /**
         * This method checks if the queue has elements to be read. It must be
         * called from a synchronized method.
         *
         * @return true if the current thread can read, false if not.
         */
        private boolean canRead() {
                boolean res;
                if (readIdx == writeIdx) {
                        /**
                         * In this case the read idx would overtake the write idx. So we are
                         * not allowed to read.
                         */
                        res = false;
                } else {
                        res = true;
                }
                return res;
        }

        /**
         * This method checks if an item can be added to the queue. It must be
         * called from a synchronized method.
         *
         * @return true if an item can be added, false if not.
         */
        private boolean canWrite() {
                boolean res;
                int nextPos;
                /**
                 * Virtually go to the next read position.
                 */
                nextPos = (writeIdx + 1) % (queue.size());
                if (nextPos == readIdx) {
                        /**
                         * In this case the write idx would overtake the read idx. So we are
                         * not allowed to write.
                         */
                        res = false;
                } else {
                        res = true;
                }
                return res;
        }

}
```