TITELSTORY

Die Programmiersprache Java hat sich bereits in wichtigen Feldern gegen C++ durchsetzen können. Wichtige Eigenschaften sind die Objektorientiertheit und eine einfache Syntax. Vorangetrieben wird die Verbreitung von Java auch durch die Tatsache, dass Embedded-Systeme zunehmend komplexer werden. Beispiele sind die industrielle Automation, die Medizintechnik, Luft- und Raumfahrt oder die Automobilindustrie. Auf all diesen Systemen läuft eine Vielzahl von unterschiedlichen Betriebssystemen und Prozessorarchitekturen. Der Einsatz von Java als eine plattformunabhängige Programmiersprache macht sie gerade für Echtzeitanwendungen besonders interessant.

Echtzeit mit Java

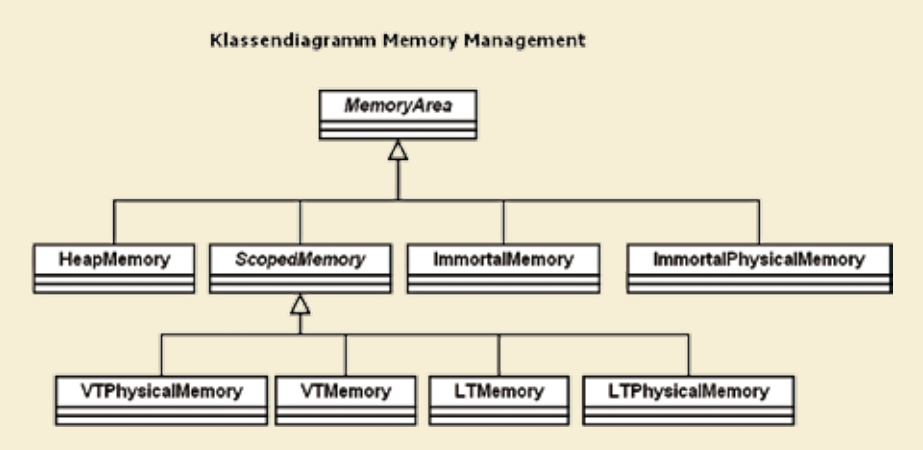
Eingeschränkte Ressourcen und Echtzeitfähigkeit: Im Automobilbau oder der Luftfahrt kommen vermehrt Echtzeitbetriebssysteme zum Einsatz. In unserem Beitrag stellen wir Ihnen die Erweiterungen der Real Time Specification for Java (RTSJ) vor.

STEFAN KUNTSCHAR *

Bereits 1999 wurden die Requirements für eine Echtzeiterweiterung der Java Spezifikation vom „National Institute of Standards and Technology“ (NIST) veröffentlicht. Daraufhin machten sich zwei unterschiedliche Gruppierungen an die Arbeit, eine geeignete Spezifikation zu erschaffen. Der erste Lösungsansatz wurde vom „J Consortium“ vorgestellt. Die Spezifikation wurde unter dem Namen „The Real-Time Core Extension“ veröffentlicht. Diese ist bis heute immer noch im Entwicklungsstadium und es existiert keine bekannte Referenzimplementierung.

Der zweite Lösungsansatz ist die RTSJ der „Real-Time for Java Expert Group“ (RTJEG). Diese Spezifikation definiert eine neue API, die auf „Java Virtual Machine“ (JVM) aufsetzt. Eine Referenzimplementierung ist bereits in die Spezifikation aufgenommen worden. Die RTSJ wurde als Erweiterung der „Java Standard Editon“ (J2SE) entwickelt und nicht für die „Java Micro Edition“ (J2ME). Theoretisch sollten alle Applikationen, die bisher für die J2SE geschrieben wurden sind, auf einer Implementierung der RTSJ laufen können.

Die komplette RTSJ ist sehr komplex. Dadurch ist allerdings auch der Memory Footprint gewachsen. Eine JVM nach RTSJ mit einem darunterliegenden RT-Linux als Echtzeitbetriebssystem hat eine Größe von 5 MByte. Damit ist die RTSJ für die meisten Systeme mit eingeschränkten Ressourcen erst einmal zu groß. Aber die Hersteller der Real-Time Java VMs haben oftmals eigene Lösungen parat, um dieses Problem in den Griff zu bekommen. Hierfür werden unterschiedliche Compiler- und Linkerverfahren eingesetzt sowie oftmals nur eingeschränkte Standardbibliotheken zur Verfügung gestellt. In der RTSJ werden sieben Leitlinien definiert. Diese sollen die Ziele der Real-Time Spezifikation aufzeigen:

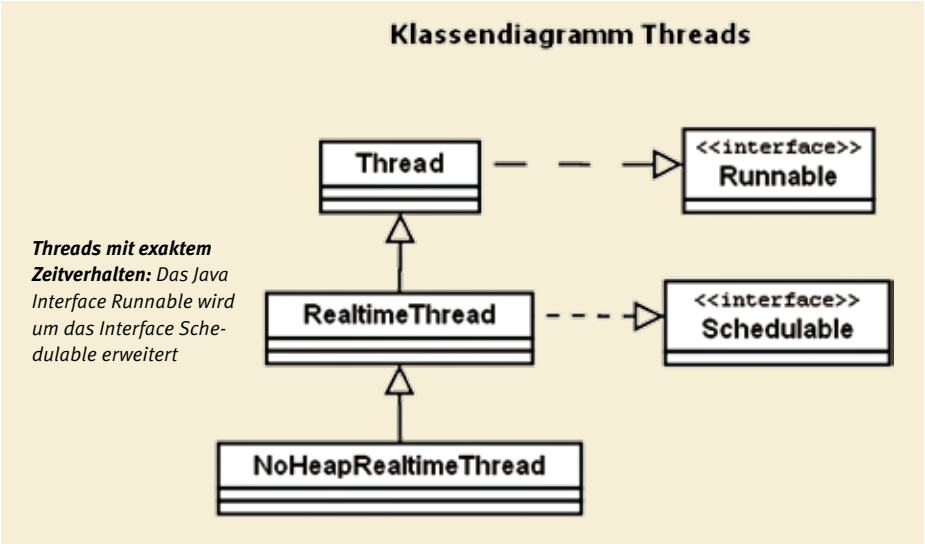


- *Applicability to Particular Java Environments:* Die RTSJ soll in allen Java Umgebungen einsatzfähig sein und darf deswegen keine Bibliotheken und Spezifikationen nutzen, die sie an ein bestimmtes „Java Developement Kit“ (JDK) oder eine Edition binden würde.
- *Backward Compatibility:* Alle nach dem Java Standard geschriebenen Applikationen sollen auf einer Implementierung der RTSJ lauffähig sein.
- *Write once, Run Anywhere:* Die Portabilität verliert im Bereich Embedded Systems nicht ihre grundsätzliche Bedeutung, muss aber auf Grund der Plattformvielfalt manchmal zurückstecken.
- *Current Practice Vs. Advanced Features:* Die RTSJ muss aktuelle Standards unterstützen, aber auch Platz für Erweiterungen bieten.
- *Predictible Execution:* Die Sicherheit und Vorhersagbarkeit hat in der RTSJ höchste Priorität. Dies kann zu Lasten der Performance gehen.
- *No Syntactic Extension:* Die RTSJ soll keine neuen Schlüsselwörter oder syntaktische Erweiterungen einführen um die Arbeit und die Einarbeitungszeit der Tool-Entwickler und Programmierer zu minimieren.
- *Allow Variation in Implementation Decisions:* Die RTSJ versucht dem Entwickler nicht vorzuschreiben, wie er die Applikation umzusetzen hat.

Die erste große Erweiterung betrifft die Zeit/Timer. Java selbst bietet eine Auflösung in Millisekunden an. Viele Systeme brauchen eine deutlich bessere Auflösung. Die RTSJ erweitert diese auf den Nanosekundenbereich. Hierfür wird der long Wert (64 Bit) für Millisekunden um einen integer Wert (32 Bit) für Nanosekunden erweitert. Dabei wird der Millisekundenwert alle 1.000.000 Nanosekunden inkrementiert. Das stellt die Kompatibilität zur Auflösung von Standard Java sicher.

Zugriff und Berechnungen von Zeiten werden in der abstrakten Klasse HighResolutionTime umgesetzt. Diese bietet die Klassen AbsoluteTime und RelativeTime sowie das Interface Comparable zum einfachen Vergleichen von Zeiten an. Mit AbsoluteTime wird die exakte Zeitangabe dargestellt: 11.06.2010, 19:30 Uhr. Das erste darstellbare Datum ist, wie in der IT-Branche üblich, der 1.1.1970, 00:00:00 Uhr GMT.

*** Stefan Kuntschar**
... ist Softwareentwickler bei Mixed Mode in München.



Threads mit exaktem Zeitverhalten: *Das Java Interface Runnable wird um das Interface Schedulable erweitert*

Von diesem Datum an lassen sich sämtliche Zeitpunkte bis in etwa 292 Millionen Jahren auf die Nanosekunde genau angeben. AbsoluteTime ist mit der Standard Klasse java.util.date kompatibel. RelativeTime repräsentiert Zeiträume, wie Stunden. Da der vorher schon beschriebene Datentyp verwendet wird, lassen sich Zeiträume zwischen einer Nanosekunde und 292 Millionen Jahren festlegen.

Die zwei Schwächen bei der Zeitenverarbeitung

Früher konnten zusätzlich mit der Klasse RationalTime Zeitintervalle vorgegeben werden. Diese Klasse wurde jedoch in der RTSJ 1.0.1 nicht weitergeführt, da das zugrundeliegende Konzept mit der RTSJ keine geeignete Umsetzung erlaubt. Die Klasse ist inzwischen als „deprecated“ gekennzeichnet. Insgesamt offenbart die Klasse HighResolutionTime zwei Schwächen.

Erstens müssen Zeiten mit Bedacht bearbeitet werden. Der Nanosekundenwert darf nicht >999.999 sein. Dann erfolgt das Inkrementieren des Millisekundenwertes. Da der Nanosekundenwert ein Integer ist, kann er allerdings unbemerkt mit größeren Werten initialisiert werden. Zweitens werden zur Bearbeitung von Zeiten keine Multiplikation und Division angeboten. Dies kann zu Einschränkungen der Implementierung von Applikationen führen. Neben HighResolutionTime wird die Klasse Clock angeboten. Sie bietet Zugriff auf die Systemuhr und kann die Genauigkeit dieser Uhr abfragen. Die Genauigkeit gibt das kleinste Zeitintervall an und damit die kleinste von der Systemuhr unterstützte Genauigkeit. Viele Systeme erreichen die theoretische Nanosekundengenauigkeit von Real-Time Java (RTJ) nicht.

Als dritte Zeitklasse neben HighResolutionTime und Clock wird die Klasse Timer mit OneShotTimer und PeriodicTimer angeboten. Ein OneShotTimer läuft nach dem Start exakt einmal und wird nach dem Auslösen eines asynchronen Events beendet. Ein PeriodicTimer läuft, sofern einmal gestartet, bis zum expliziten Beenden weiter und wirft nach jedem Ablauf der eingestellten Zeit ein asynchrones Event. Dadurch können einfache zyklische Tasks ersetzt und Ressourcen eingespart werden.

Speicherverwaltung und Speicherzugriff

Standard Java verfügt über Heap Memory, das durch den Java Garbage Collector, kurz JGC, regelmäßig gesäubert wird. Es verwirft Objekte im Speicher, auf die es zurzeit keine Referenzen gibt. Viele Objekte müssen in einem Echtzeitsystem dauerhaft angelegt werden, damit dynamische Speicherallokation vermieden wird. Außerdem ist ein JGC hinderlich, der ohne vorhersagbares Zeitverhalten mit hoher Priorität läuft und das System durcheinander bringt. RTSJ bietet zwei Möglichkeiten: Die abstrakte Basisklasse heißt MemoryArea. Sie unterteilt sich in das bekannte HeapMemory, ImmortalMemory, ScopedMemory und ImmortalPhysicalMemory. Zur einfacheren Speicherverwaltung und -allokation wird noch die Klasse SizeEstimator definiert.

Das HeapMemory ist die geläufige Speichervariante in Java mit automatischer Speicherverwaltung und JGC Unterstützung. Es wird exakt einmal angelegt und hat eine statische Größe. Es ist nicht für harte Echtzeitanforderungen geeignet, kann aber für Programmteile ohne Echtzeitanforderung komfortabel Speicher zur Verfügung stellen.

Speicherbereich über die gesamte Laufzeit

Das ImmortalMemory ist ein Speicherbereich, der für die gesamte Laufzeit erhalten bleibt. Er wird nicht vom JGC gesäubert und damit auch nicht automatisch verwaltet, sondern muss vom Softwareentwickler bewusst genutzt werden.

Hier sind zwei Nachteile bekannt. Erstens gestaltet sich das Anlegen eines neuen Objektes für den herkömmlichen Javaentwickler umständlich und zweitens kann ein im ImmortalMemory angelegtes Objekt während der gesamten Laufzeit nicht mehr gelöscht werden. Das ImmortalMemory existiert ähnlich dem HeapMemory exakt einmal und hat eine statische Größe. Die dort angelegten Objekte können auch aus allen anderen Speicherbereichen angesprochen werden.

Das ImmortalPhysicalMemory ist in der Anwendungsweise dem ImmortalMemory sehr ähnlich, besitzt allerdings die Möglichkeit, Speicher mit speziellen Eigenschaften zu nutzen. Das ImmortalPhysicalMemory ist weder als Singleton definiert, noch statisch. Zur Verwaltung dieser Speicherbereiche wird der PhysicalMemoryManager zur Verfügung gestellt. Die Spezifikation des Immortal PhysicalMemory ist allerdings noch nicht ganz ausgereift und wird von einigen VMs entweder noch nicht oder nur ansatzweise unterstützt.

Eingeschränkte Nutzung von Referenzen

Sollen Objekte ohne Einfluss des JGC und mit begrenzter Lebensdauer angelegt werden, empfiehlt sich das ScopedMemory. Diese Speicherbereiche bleiben nur erhalten, solange sie von einem SchedulableObject, wie einem Echtzeit Thread, referenziert werden. Danach wird der Speicher wieder freigegeben und damit alle enthaltenen Objekte gelöscht. Dadurch kommt es zu Einschränkungen in der Nutzung von Referenzen. Ob-

„Die Zeit zwischen dem Auslösen eines Systems und dessen Reaktion muss so gering wie möglich sein.“

Greg Bollella, Sun Microsystems

jekte in einem ScopedMemory dürfen nur von Objekten im gleichen oder in einem untergeordneten ScopedMemory referenziert werden, keinesfalls jedoch von HeapMemory oder ImmortalMemory.

Das ScopedMemory wird noch einmal in vier Klassen unterteilt. Dabei unterscheidet man Memory und PhysicalMemory, das für speziellen Speicher konzipiert ist, sowie Speicher mit variabler (VT) und linearer Allokationszeit (LT). Daraus ergeben sich die vier Kombinationen VTPhysicalMemory, VTMemory, LTPhysicalMemory und LTMemory.

Direkter Zugriff auf Speicher und Register

Echtzeitsysteme benötigen teilweise einen direkten Zugriff auf Speicher und Register, was mit Standard Java nicht möglich ist. Die RTSJ stellt hierfür die Klasse RawMemoryAccess mit der Unterklasse RawMemoryFloatAccess zur Verfügung.

Durch die Klasse RawMemoryAccess kann ein Speicherbereich, der durch eine Startadresse (Offset) und eine Größenangabe (Size) fest definiert ist, für den direkten Zugriff freigegeben werden. Der definierte Speicherbereich wird je nach Größe als byte, short, int oder long interpretiert. Mehrere Speicherbereiche hintereinander können als Array dieser Datentypen angesprochen werden.

Die Klasse RawMemoryFloatAccess nutzt den gleichen Mechanismus, interpretiert die Speicherbereiche allerdings mit den Datentypen float oder double. Bei direktem Speicherzugriff sind drei Punkte zu beachten:

- Die Reihenfolge der Bytes ist plattformabhängig (Byte Order) und muss richtig interpretiert werden.
- In diesen Speicherbereichen dürfen keine Objekte angelegt werden.
- Nur der direkte Zugriff auf Speicherbereiche außerhalb der VM ist erlaubt, um Inkonsistenzen zu vermeiden.

Das Scheduling im Echtzeitbetriebssystem

Ein wichtiger Punkt für Echtzeitsysteme ist das Scheduling. Standard Java benutzt einen nicht ganz konsequenten Prioritätsscheduler mit wenigen Prioritätsstufen. Deshalb führt die RTSJ die Klasse Scheduler ein. Hier kann ein eigener Scheduler implementiert werden oder es kann der nach RTSJ zwingend vorhandene PriorityScheduler eingesetzt werden.

Dieser arbeitet nach dem gängigsten Algorithmus und führt ein Fixed Priority Preemptive Scheduling durch. Jeder Thread bekommt eine Priorität zugewiesen. Dabei bekommt der Thread mit der höchsten Priorität die CPU. Aktuell ausgeführte Threads mit niedriger Priorität werden unterbrochen.

Auch die Prioritätsstufe des JGC kann eingestellt werden. Der Scheduler ist als Singleton implementiert. Dadurch werden Inkonsistenzen und ein nicht vorhersagbares Zeitverhalten vermieden, da nicht mehrere Scheduler gleichzeitig aktiv sein können. Die RTSJ schreibt mindestens 38 unterschiedliche Prioritätsstufen vor. Manche VMs implementieren mehr Stufen. Dabei muss beim Wechsel oder Update einer VM darauf geachtet werden, dass noch alle Prioritäten stimmen, vor allem beim Arbeiten mit den Methoden getMaxPriority() und getMinPriority().

Threads ein exaktes Zeitverhalten mitgeben

Um den Threads ein exaktes Zeitverhalten zu geben, wird das Java Interface Runnable um das Interface Schedulable erweitert. Neben den SchedulingParameters werden ProcessingGroupParameters, ReleaseParameters und MemoryParameters definiert. Da eine Auflistung und Erklärung sämtlicher Parameter jeden Rahmen sprengen würde, soll hier nur auf die RTSJ verwiesen werden. Es sei erwähnt, dass diese Parameter ein System hochflexibel konfigurieren lassen.

Passend zum Echtzeitscheduler sind zwei weitere Threadklassen spezifiziert, die von der Klasse Thread abgeleitet sind. Der RealtimeThread nutzt das Interface Schedulable. Für den RealtimeThread kann ein Zeitverhalten definiert werden. Dieses wird allerdings nicht zwingend eingehalten, da dieser Threadtyp mit dem HeapMemory arbeiten darf. Der HeapMemory wird vom JGC überwacht. Nach Aktivierung des JGCs muss auf dessen Ende gewartet werden, um Inkonsistenzen zu vermeiden. Dies verzögert auch den erneuten Aufruf von RealtimeThread. In Anbetracht der genannten Punkte ist die Klasse nur für weiche Echtzeitanforderungen einzusetzen, in der Zeitvorgaben nicht zwingend eingehalten werden müssen.

Für harte Echtzeitbedingungen gibt es die Ableitung von RealtimeThread: den so genannten NoHeapRealtimeThread. Dieser verwendet kein HeapMemory und unterbricht einen laufenden JGC. Zum Datenaustausch zwischen den unterschiedlichen Threads werden verschiedene Queues verwendet. Eine WaitFreeWriteQueue kann ohne Verzögerung beschrieben, aber nur synchronisiert gelesen werden.

Eine WaitFreeReadQueue funktioniert genau umgekehrt. Eine WaitFreeDequeue besteht aus beiden oben genannten Queues. Allerdings wird sie in der RTSJ als veraltet gekennzeichnet, da sie durch eine WaitFreeWriteQueue und eine WaitFreeReadQueue abgebildet werden kann. Ein Echtzeit Thread muss immer auf der unsynchronisierten Seite mit Sofortzugriff auf die Daten implementiert werden. (...)

Im letzten Abschnitt lesen Sie, wie einem Thread unterschiedliche Prioritäten mitgegeben werden können. Dazu erläutern wir zwei Mechanismen die verhindern, dass Threads mit hoher Priorität beim Zugriff auf exklusive Ressourcen auf Threads mit niedriger Priorität warten müssen. Über unseren InfoClick gelangen Sie direkt zum Online-Beitrag. // **HEH**

Mixed Mode: +49 (0)89 89868200

InfoClick

- 1. Teil: Grundlegendes zur Programmiersprache Realtime Java
- Programmierung mit der Real Time Specification for Java

www.elektronikpraxis.de **InfoClick** 2665782