Ausgewählte Kapitel eingebetteter Systeme

Java Real-Time Specification

Tobias Distler

05.07.2006

Problem

Nichtdeterministisches Verhalten der Garbage Collection

Weitere Nachteile

- Scheduling mangelhaft
- kein physikalischer Speicherzugriff
- ungenaue Timer

JRTS – Die Eckpfeiler

- Scheduling
- Speicherverwaltung
- Physikalischer Speicherzugriff
- Synchronisation
- Asynchrones Eventhandling
- Asynchroner Kontrolltransfer
- Asynchrone Thread-Beendigung

- Basis-Scheduler
  - Klasse: „PriorityScheduler“
  - 28 Echtzeit-Prioritäten
  - präemptiv
- schedulebare Objekte
  - Implementierung des Interfaces „Schedulable“
  - Verwaltung der eigenen Ausführungs-Parameter
  - Einplanung durch den Scheduler

- Thread
  - übernommen aus Java
  - Einsatzbereich: Nicht-Echtzeit-Aufgaben
- RealtimeThread
  - niedrigere Priorität als die Garbage Collection
  - Einsatzbereich: weiche Echtzeit
- NoHeapRealtimeThread
  - höhere Priorität als die Garbage Collection
  - Einsatzbereich: harte Echtzeit

NoHeapRealtimeThread

- Unterklasse von RealtimeThread
- evtl. Verdrängung der Garbage Collection

➔ strikte Heap-Unabhängigkeit erforderlich

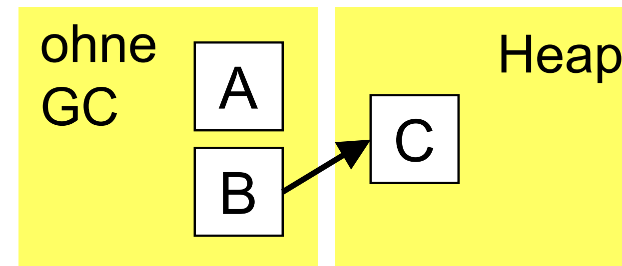
➔ **VERBOTEN**:

- `this` im Heap
- Aufruf-Parameter im Heap
- Allokation von Objekten im Heap
- Referenzierung von Objekten im Heap
- Veränderung von Referenzen auf Objekte im Heap

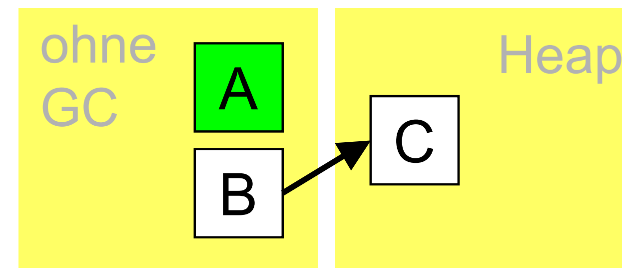
Problemfall: Manipulation von Heap-Referenzen

```
A.pointer = B.pointer;
B.pointer = null;
```

1. Ausgangssituation



2. GC überprüft A auf Referenz zu C

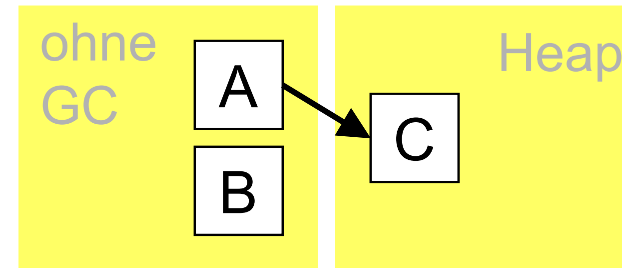


3. GC wird von NHRT verdrängt

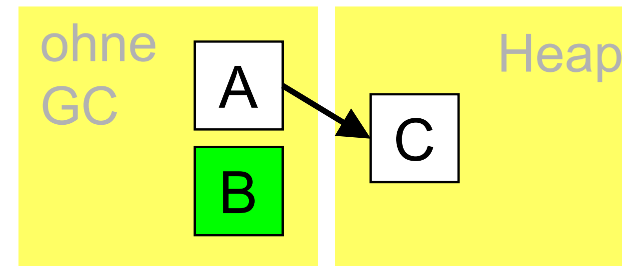
Problemfall: Manipulation von Heap-Referenzen

4. NHRT verändert Referenzen, beendet sich

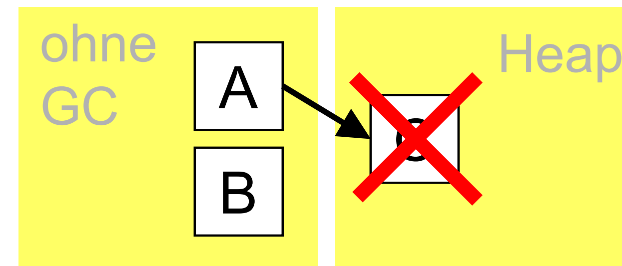
```
A.pointer = B.pointer;
B.pointer = null;
```



5. GC überprüft B auf Referenz zu C



6. GC löscht C



MemoryArea

- abstrakte Basisklasse für alle Speicherbereich-Klassen
- Funktionsweise:

```
MemoryArea ma;
[...] // Initial. mit Konstruktor bzw. instance()
ma.enter(
    new Runnable() {
        public void run() {
            /* alle mit ‚new‘ erzeugten Objekte
             * werden in der entsprechenden
             * MemoryArea abgelegt
             */
        }
    }
);
```

Unterklassen von MemoryArea

- HeapMemory
  - Heap im klassischen Sinn
  - Garbage Collection

```
HeapMemory hm =
    HeapMemory.instance();
```

- ImmortalMemory
  - Objekte leben bis zum Ende der Anwendung
  - **kein** Löschen möglich
  - **keine** Garbage Collection

```
ImmortalMemory im =
    ImmortalMemory.instance();
```

- ScopeMemory

ScopeMemory

- Objekte mit beschränkter Lebenszeit
- **keine** Garbage Collection
- Funktionsweise:

1. Scope erzeugen

```
ScopeMemory scope = new ScopeMemory(size);
```

2. Scope betreten

```
scope.enter( /* Runnable-Code */ );
```

3. Scope verlassen: Zurückkommen aus `enter()`
4. Scope aufräumen: bei allen im Scope enthaltenen Objekten wird `finalize()` aufgerufen

Eckpfeiler – Physikalischer Speicherzugriff *Übersicht*

- Vorteile
  - direkter Zugriff auf die Hardware
  - z.B. Programmierung von Treibern möglich
- spezifizierte Klassen
  - VTPhysicalMemory, LTPhysicalMemory
  - ImmortalPhysicalMemory
  - **RawMemoryAccess**, RawMemoryFloatAccess

RawMemoryAccess

- physikalischer Speicher als Byte-Sequenz
- Zugriff durch Offsets
- keine Referenzen auf Java-Objekte erlaubt

```
public class RawMemoryAccess {
    public RawMemoryAccess(Object type, long size);
    public byte getByte(long offset);
    public void setByte(long offset, byte value);
    public void getInts(long offset, int[] ints,
                        int low, int number);
    public void setLongs(long offset, long[] longs,
                         int low, int number);
    [...]
}
```

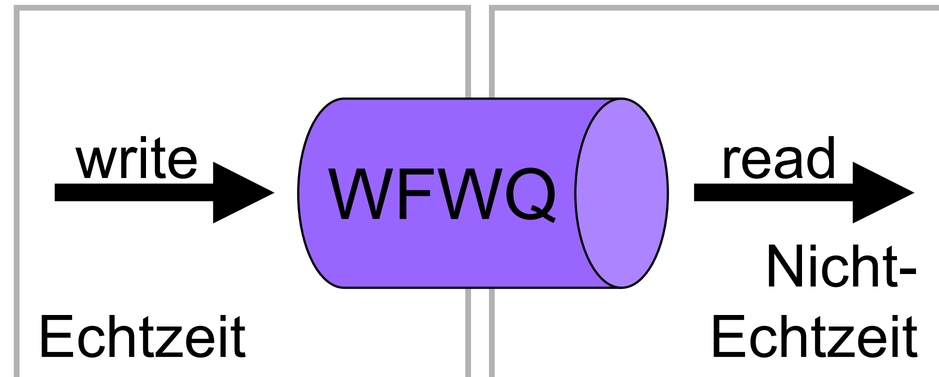
Eckpfeiler – Synchronisation

- Konzepte
  - Java-Monitore (`synchronized`)
  - Priority-Inheritance-Protokoll verpflichtend
  - Priority-Ceiling-Protokoll optional
- Mechanismen
  - Warteschlangen für Belegung synchronisierter Blöcke

    zuerst Sortierung nach Prioritäten

    dann innerhalb einer Priorität: FIFO
  - WaitFree-Queues zur Kommunikation zwischen Threads

- WaitFreeWriteQueue
  - blockierendes Lesen
  - nicht-blockierendes Schreiben



- WaitFreeReadQueue
  - nicht-blockierendes Lesen
  - blockierendes Schreiben
- WaitFreeDequeue

  Kombination: WaitFreeWriteQueue + WaitFreeReadQueue

Eckpfeiler – Asynchrones Eventhandling *Übersicht*

- Verwendung
  - Reaktion auf äußere Ereignisse
  - Reaktion auf innere Ereignisse
  - Timer
- spezifizierte Klassen
  - AsyncEvent
  - AsyncEventHandler

AsyncEvent

- Repräsentation eines asynchronen Ereignisses
- Alarmierung von mehreren Handlern möglich
- selbstständige Verwaltung seiner AsyncEventHandler

```
public class AsyncEvent {
    public void addHandler(AsyncEventHandler handler);
    public void fire();
    [...]
}
```

AsyncEventHandler

- Behandlung eines asynchronen Ereignisses
- schedulebares Objekt
- Laufzeitlänge, Blockierung: keine Beschränkungen

```
public class AsyncEventHandler implements Schedulable {
    public void handleAsyncEvent();
    [...]
}
```

Eckpfeiler – Asynchrones Eventhandling *Beispiel*

```
public class AEExample {
    public static void main(String[] args) {
        AsyncEventHandler handler = new AsyncEventHandler() {
            public void handleAsyncEvent() {
                System.out.println(„tick");
            }
        };
        PeriodicTimer timer =
                  new PeriodicTimer(null,
                                    new RelativeTime(1000, 0),
                                    handler);
        timer.start();
        try {
            Thread.sleep(60000);
        } catch(Exception e) {
            System.exit(1);
        }
    }
}
```

Eckpfeiler – Asynchroner Kontrolltransfer *Übersicht*

- Verwendung
  - Abbruch von Berechnungen
  - Reaktion auf besondere Ereignisse
- Umsetzung
  - Erweiterung von `interrupt()`
  - Abwicklung über *AsynchronouslyInterruptedExceptions*
  - Kennzeichnung von unterbrechbaren Bereichen

```
void foo()
throws AsynchronouslyInterruptedException {
     /* Code */
}
```

Eckpfeiler – Asynchroner Kontrolltransfer *Beispiel*

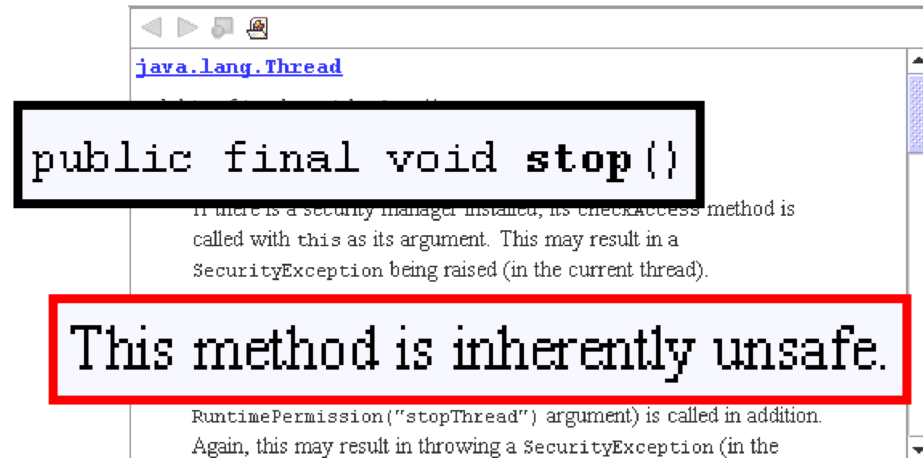
```
public class AKTBeispiel extends RealtimeThread {
    static AsynchronouslyInterruptedException intExc =
           AsynchronouslyInterruptedException.getGeneric();

    public void run() {
        try {
            foo();
        } catch(AsynchronouslyInterruptedException aie) {
            if(intExc.happened(false)) {
                /* Code */
            }
        }
    }
}
```

Aufruf:

```
AKTBeispiel rt;
[...]
rt.interrupt();
```

Eckpfeiler – Asynchrone Thread-Beendigung



- Notwendigkeit
  - `stop()` unsicher
  - `destroy()` nicht implementiert
- Forderung

  sicheres, sauberes Beenden eines Threads
- Umsetzung
  - Grundsatz: ein Thread kann nicht direkt beendet werden
  - Verwendung von `interrupt()`
  - Thread mit Hilfe von AIE-catch-Block sauber beenden

Zusammenfassung

- Anpassung an Echtzeitanforderungen
  - Beherrschbarkeit der Garbage Collection
  - Vorhersagbarkeit (z.B. Dauer der Speicherallokation)
- Anpassung an Echtzeitprogrammierung
  - Möglichkeit des physikalischen Speicherzugriffs
  - Mechanismus für Ereignissteuerung
  - sichere Thread-Beendigung