Bossa – Aspektorientierte Scheduler Entwicklung

Marc Lörner

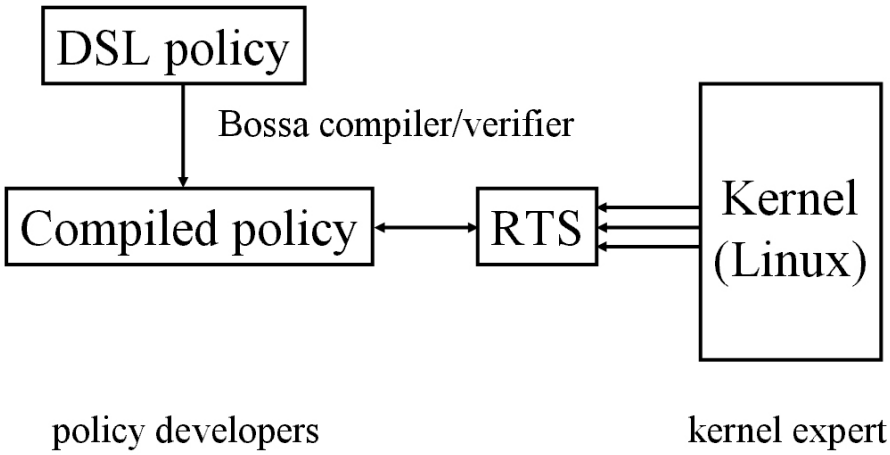
Gerd.loerner@t-online.de

1. Motivation

Einen Scheduler zu schreiben, ist eine schwierige Aufgabe, nicht die Implementation des Algorithmus, sondern die Verteilung des Codes über das gesamte Betriebssystem, sowie die Integration in ein bereits existierendes System macht die Sache so schwierig. Durch die Verteilung wird es einem erschwert bzw. fast unmöglich gemacht, einfach mal „schnell“ einen neuen Scheduling-Algorithmus in das System einzubringen. Daher war es unerlässlich, Bossa, eine Domain Specific Language (DSL), zu entwickeln, deren dahinter stehende Theorie und Implementierung nachfolgend beschrieben wird.

2. Überblick über das Bossa-System

Die Entwicklung der Scheduling-Strategien eines Betriebssystems wird im Bossa-System in die drei verschiedene Komponenten Bossa-ready-Kernel, Runtime-System und dem kompilierten Scheduling-Code aufgeteilt, welche im nachstehenden Bild in Relation gesetzt sind.



Der Bossa-ready-Kernel ist ein ganz normaler Standard-Linux-Kernel, der mit Hilfe eines Kernel-Experten dahingehend verändert wurde, dass er alle Aktivitäten, die sich auf das Scheduling beziehen, wie z. B. Prozessblockierung, oder –deblockierung, direkt an das Runtime-System weiterleitet. Diese Events werden dort dann je nach Art auf den Scheduling-Code verteilt.
Der Scheduling-Code kann von Jedermann in der für das Bossa-System entwickelten DSL geschrieben werden, der dann mit dem Bossa-Compiler übersetzt, verifiziert und schließlich als C-Datei in das Runtime-System gebunden wird.
Die Vorteile dieses Verfahrens sind ganz klar die Zentralisierung des Scheduling-Codes, anstatt der normalerweise eingesetzten, verteilten Implementierung, und die Verwendung einer DSL, anstatt von Low-Level-Programmiersprachen, bzw.

Assemblerbefehlen. Außerdem beinhaltet der Bossa-Compiler einen eingebauten Verifizierungs-Algorithmus, der dafür sorgt, das weniger bzw. keine unbeabsichtigten Fehler vom Entwickler in den Quellcode gelangen. Durch all diese Verbesserungen wird insgesamt für einen übersichtlicheren, leichter verständlicheren Entwicklungsprozess im Feld der Scheduler-Entwicklung gesorgt.

3. Die Bossa-DSL

Die DSL von Bossa setzt sich aus den drei Hauptelementen Deklarationen, Eventhandler und Interface-Funktionen zusammen, welche nachfolgend eingehender, auch an Hand von Beispielcode erläutert werden sollen.

3.1 Deklaration

In diesem Teil der Sprache werden Scheduling-Algorithmen, Prozesseigenschaften, Prozesszustände, sowie Sortierungskriterien definiert.

```
Type policy_t = enum {SCHED_FIFO, SCHED_RR, SCHED_OTHER};

Process = {
   Policy_t  policy;
   Int       rt_priority;
   Time      priority;
   Time      ticks;
   System struct ctx mm;
}

states = {
   RUNNING        running   :    process,
                                 previous old_running;
   READY          ready     :    sorted_queue;
   READY          expired   :    queue;
   BLOCKED        yield     :    process;
   BLOCKED        blocked   :    queue;
   TERMINATED     terminated;
}

ordering_criteria = {
   highest rt_priority, highest ticks,
   highest ((!empty(old_running) &&
             mm == old_running.mm) ? 1 : 0)
}
```

Zuerst definiert man sich einen policy_t-Typ, der alle verwendeten Scheduling-Algorithmen enthält. Danach werden dem Prozess verschiedene Attribute zugeordnet, welche entweder Zeitmessungen, Speicherinformationen oder einfach nur einen Prioritätswert repräsentieren. Danach müssen die verschiedenen Statuseigenschaften definiert werden, welche entweder einer Schlange oder einem Prozess entsprechen und folgende Attribute haben können: RUNNING(entspricht dem laufenden Prozess), READY (gibt die Prozesse an, die

gescheduled werden dürfen), BLOCKED(alle Prozesse, die blockiert sind) und schließlich TERMINATED(welches alle beendeten Prozess enthält). Weiterhin muss mindestens bei einem Attribut „sorted“, also sortiert angegeben werden, da nur Prozesse welche in einer sortierten Schlange sind, den Prozessor erhalten dürfen, um gravierende Fehler im Algorithmus zu vermeiden. Dann gibt es noch das Schlüsselwort „previous“, mit dem man sich den vorhergehenden Wert eines Attributs merken kann. Zum Schluss muss man noch das Sortierkriterium festlegen, nach dem sortiert werden soll. „highest“ gibt hier jeweils den höchsten Wert zurück und „lowest“ den Niedrigsten. Die Reihenfolge dort gibt an, was gewählt werden soll, wenn es mehrere höchste oder niedrigste Werte gibt.

3.2 Eventhandler

Eventhandler beinhalten den Code, der beim Auftreten eines Events ausgeführt werden soll. Daher auch die Namen „on block“, „on unblock“, „on system clocktick“ … Dabei gibt „e“ das aufgetretene Event an.

```
On block.* {
   e.target => blocked;
}

On unblock.* {
   If(e.target in blocked) {
        e.target => ready;
        if(!empty(running) && e.target > running) {
             running => ready;
        }
   }
}
```

“.*” stellt hier jeweils das Wildcard-Symbol dar und „=>“ ist die Schreibweise dafür, dass dem links von dem Pfeil stehendem Ziel oder Prozess, der rechts stehende Status zugeordnet wird.

3.3 Interface-Funktion

Interface-Funktionen sind Schnittstellen zwischen verschiedenen Scheduling-Strategien, und können wie folgt aussehen.

```
Void sched_setscheduler(process p, policy_t policy,
                        System struct sched_param param) {
   Int prio = param.sched_priority;
   If(policy == SCHED_OTHER && prio != 0) {
        Error(“non-RT process: invalid priority”);
   } else {
        if(prio < 0 || prio > 99) {
             error(“RT process: invalid priority”);
        }
   }
   p.policy = policy;
   p.rt_priority = prio;
}
```

Die Error-Funktion hier bricht die Funktion ab und gibt den angegebenen String zurück.

4. Scheduling-Hierarchien

Bei Scheduling-Hierarchien handelt es sich um Baumstrukturen von Virtuellen- und Prozess-Schedulern. Virtuelle-Scheduler sitzen dabei an der Wurzel und allen anderen Knoten, die keine Blätter des Baumes sind, und die Prozess-Scheduler sitzen an den Blättern.
Es ist ersichtlich, dass alle auftretenden Events, vom Wurzelknoten aus, durch den kompletten Baum, bis zu einem Prozess-Scheduler, geschickt und verarbeitet werden müssen und auch die Ergebnisse des Schedules müssen durch den gesamten Baum verteilt werden. Weshalb die Events nun einen Verweis auf den nächsten Scheduler und zusätzlich die Funktionen „forwardImmediate“ und „forwardDeferred“ bereitstellen. Dabei gibt „forwardImmediate“ an, dass das Event gleich weitergeleitet werden soll. Bei „forwardDeferred“ hingegen warten die Events solange an dem Knoten, bis irgendwann ein „forwardImmediate“ auftaucht.

```
On block.* {
   e.next => forwardImmediate();
}

On unblock.* {
   If(e.next in blocked) {
        /* preemption possible */
        e.next => forwardImmediate();
        if(e.next in ready && !empty(running) &&
           e.next > running) {
             running => ready;
        }
   } else {
        /* preemption not possible */
        e.next => forwardImmediate();
   }
}
```

Weiterhin kommen neue Interface-Funktionen hinzu, die gewährleisten, dass ein neuer Scheduler unten am Baum angehängt werden kann, was zu einer dynamischen Erweiterung des Baumes führen könnte, um auf auftretende Events besser reagieren zu können. Das Beispiel unten zeigt, wie eine solche Funktion aussehen könnte.

```
void attach(process p, policy_t policy, int rt_priority,
            time priority) {
   p.policy = policy;
   p.rt_priority = rt_priority;
   p.priority = priority;
   p => ready;
}
```

5. Bossa und AOP?

Da sich der Scheduling-Code normalerweise quer durch ein Betriebssystem zieht und dadurch nicht gerade verständlicher wird, hat man in Bossa den Versuch gemacht, den komplette Quellcode der Scheduler an einen Ort zentral zu binden, welcher dann mit Hilfe von AOP-Techniken über das Betriebssystem verteilt werden kann. Jedoch hat es sich herausgestellt, dass normales AOP in diesem Zusammenhang zu statisch ist, deshalb hat man sich das neue Konzept des Event-based model of AOP (EAOP) ausgedacht, welches auf Events basiert und es dadurch ermöglicht, während der Ausführung, dynamisch die Scheduling-Strategie zu verändern.
Wenn ein Event auftritt, dann wird das gerade laufende Hauptprogramm sofort gestoppt und der entsprechende Eventhandler aufgerufen. Nach dessen Ausführung wird das unterbrochene Hauptprogramm dann wieder fortgesetzt.

6. Performanzuntersuchung

Die nachfolgende Tabelle zeigt die Performanz von einem, mit Bossa erstellten, Scheduler zu dem, der im Linux 2.2.16 Kernel implementiert ist. Die Werte wurden mit Hilfe eines Pentium III-700 Mhz Compaq Armada M700 mit 256 MB Hauptspeicher und der Benchmark-Suite HBench-OS ermittelt.

Benchmark	parameter	Linux	Bossa	Bossa average/Linux average
lat_pipe		4.221 ± 0.008	4.404 ± 0.057	104%
lat_proc	null dummy	90.22 ± 0.16	93.87 ± 0.22	104%
	sh dynamic	15558 ± 23	14036 ± 26	90%
	sh static	14457 ± 10	12887 ± 17	89%
	simple dynamic	1253.3 ± 6.3	1259.0 ± 3.7	100%
	simple static	195.4 ± 4.3	183.3 ± 1.7	94%
lat_fs	create 0	9463 ± 25	9307 ± 14	98%
lat_ctx	0k 2	1.130 ± 0.015	1.240 ± 0.024	109%
	0k 20	1.916 ± 0.033	2.147 ± 0.193	112%
lat_ctx2	4k 20	2.625 ± 0.201	2.492 ± 0.098	95%
	32k 20	52.141 ± 0.292	52.971 ± 0.199	102%
	64k 20	103.256 ± 0.091	104.510 ± 0.062	101%

Wie die Tabelle zeigt, stehen Bossa-Scheduler den Linux-Schedulern in nichts nach. Sie sind zwar manchmal ein bisschen langsamer, aber das gleicht sich mit der hohen Benutzerfreundlichkeit von Bossa während der Entwicklung wieder aus.

Literatur:

- Richard A. Aberg, Julia L. Lawall, Mario Südholt und Gilles Muller: “Evolving an OS Kernel using Temporal Logic and Aspect-Oriented Programming”, http://www.emn.fr/x-info/bossa/bossa-3-2-info.ps
- Gilles Muller und Julia L. Lawall: “Towards a Scheduling Framework for Dynamically Downloaded Multimedia Applications”, http://www.emn.fr/x-info/bossa/ms2002.ps.gz

- Gilles Muller, Julia L. Lawall, Luciano Porto Barreto und Jean-Ferdinand Susini: “A Framework for Simplifying the Development of Kernel Schedulers: Design and Performance Evaluation”, http://www.cs.ubc.ca/~ycoady/acp4is03/papers/aaberg.pdf
- Julia L. Lawall: “Capturing OS Expertise in an Event Type System: the Bossa Experience”, http://www.emn.fr/x-info/bossa/lawall.ppt
- Luciano Porto Barreto, Rémi Douence, Gilles Muller und Mario Südholt: “Programming OS Schedulers with Domain-Specific Languages and Aspects: New Approaches for OS Kernel Engineering”, http://www.emn.fr/x-info/bossa/asp4is-scheduling.pdf
- Julia L. Lawall, Gilles Muller und Luciano Porto Barreto: “Capturing OS Expertise in an Event Type System: the Bossa Experience”, http://www.emn.fr/x-info/bossa/lawall-muller-barreto.pdf
- http://www.emn.fr/x-info/bossa/