Hauptseminar
Aspektorientierte Systemprogrammierung

DSLs für Systemsoftware:
DEVIL und BOSSA

Inhalt

1. DSLs: Einführung

Unter dem Begriff *DSL (**D**omain **S**pecific **L**anguage)* werden in der Informatik diejenigen Programmiersprachen zusammengefasst, die jeweils für einen ganz bestimmten Problembereich entworfen worden sind. Sie stellen damit eine Spezialisierung der allseits bekannten *general purpose languages* (wie z.B. *C, C++, Pascal,...*) dar, mit dem Ziel, eine geforderte Problemstellung innerhalb eines ganz bestimmten Felds der Informatik (der *Domäne*) möglichst einfach und intuitiv zu bearbeiten.
Bekannte Beispiele von domänenspezifischen Sprachen sind die aus der Unix-Welt bekannten Vertreter wie *bash* (für die Shell-Programmierung), *make* (für Beschreibung von Makefiles von Compilierungs-Prozessen) oder *postscript* (für die Beschreibung von druckerspezifischen Vorgängen).
Im folgenden sollen nun zwei *DSLs* aus dem Bereich der Systemsoftwareentwicklung vorgestellt werden: *DEVIL* (mit dem Ziel einer vereinfachten Gerätetreiberentwicklung) sowie *BOSSA* (eine DSL für die einfache Implementierung eines Schedulers unter Linux).

2. Devil

2.1. Motivation:

Das Problem ist nur allzu gut bekannt: Hardwarehersteller bringen in immer kürzeren Abständen immer mehr Produkte auf den Markt. Die Hardwarefunktionalität wird zunehmend komplexer, moderne Grafikcontroller (z.B. von Nvidia, ATI) haben inzwischen im Bereich der Komplexität von Funktion und Verarbeitung ihre CPU-Kollegen um ein Vielfaches überholt. So schnell die Technologie bezogen auf die Hardware aber auch fortschreitet, genauso konstant ist dabei auch immer ein Problembereich geblieben: Die Treiberentwicklung.
Die Entwicklung eines vollständigen Treibers für eine bestimmte Hardware unter einem bestimmten Betriebsystem hat in der Tat im Laufe der Zeit keine wesentliche Erleichterung erfahren. Im Gegenteil: Im Zuge von neuen Anwendungen wie Hardware/Software Co-Design ist es heutzutage die Politik vieler Hardwarehersteller, Funktionalität aus der Hardware letztendlich in den Treiber auszulagern und somit die Hardwarekosten so schlank wie nur möglich zu halten (Bsp: AMD-Alchemy Chipsets, Nvidia-Controller). Die Folgen davon sind sowohl für die Treiberentwickler als auf für die Anwender deutlich spürbar: Der Codeumfang des Treibers wird groß, unübersichtlich und nur schwer wartbar. Fehler schleichen sich relativ leicht ein und haben auf der low-level Ebene oft fatale Folgen (Betriebsystemabsturz). Entwickler freier Software schließlich müssen sich oft mit (wenn überhaupt) nur unvollständiger Dokumentation herumschlagen, und die Fehlersuche und das Debugging in einem Treiber ist alles andere als nervenschonend.
Kurzum: Erleichterungen bei der Treiberentwicklung sind sicherlich mehr als gefragt.

2.2. Lösungsansatz: DEVIL

Eine Entwicklergruppe vom *Laboratoire Bordelais de Recherche en Informatique* in Frankreich hat sich in diesem Kontext mit der Thematik Treiberentwicklung auseinandergesetzt und ist – bei der Suche nach Konzepten für Erleichterungen von Treibern – auf eine eigentlich nicht weiter verwunderliche Tatsache gestoßen:
Die gegenwärtigen Treibercodes bestehen zu einem großen Teil aus low-level Bit-Operationen (Port öffnen, Bits in Register schreiben, Bits in Register lesen,...). Der Linux-Kern in der 2.2.12er Version hat nach Angaben der Devil-Entwickler im Treibercode einen Anteil von 30% nur für eben jene Bit-Operationen.

Diese Code-Teile mit den Bitoperationen sind aber im Gegenzug auch sehr anfällig für Fehler jeglicher Art: Sei es nun ein falsch angegebener Offset bei einem best. Port, oder einfach nur ein kleiner Tipp/Denkfehler. Auf alle Fälle haben diese kleinen Fehler auf der low-level Ebene fatale Folgen und können im wachsenden Code im Allgemeinen nur noch sehr schwer entdeckt werden.
Genau unter diesem Hintergrund wurde nun schließlich Devil entwickelt: Durch Einführung einiger Abstraktionsebenen soll hierbei Treibercode nicht nur einfacher zu entwickeln sein, sondern auch portabler und sicherer. Das Prinzip ist dabei folgendes: Anstelle, dass der Treiberentwickler wie bisher direkt mit low-level Bitoperationen arbeitet, beschreibt er die Schnittstellenspezifikation des Gerätes in der *C*-ähnlichen DEVIL-Hochsprache. Der DEVIL-Compiler generiert daraus in Folge eine gewöhnliche C-Header-Datei, in der einige Inline-Funktionen *(stubs)* definiert sind, welche dem Treiberprogrammierer dann einen halbwegs komfortablen Zugriff auf die Hardware bereitstellen.

2.3. Praxisbeispiel Busmouse

2.3.1. Logitech Busmouse Spezifikation

Betrachten wir uns aber nun erstmal die Entwicklung eines Treibers anhand eines einfachen Beispiels, der Logitech Busmouse Spezifikation:
Aus funktionaler Sicht gesehen stellt die Logitech Busmousehardware wie erwartet die drei wesentlichen Funktionen einer Maushardware zur Verfügung [1]:

- Die relative X-Koordinate
- Die relative Y-Koordinate
- Den momentanen Zustand der Maustasten

Angesprochen wird diese Funktionalität wie auch bei anderer Hardware über entsprechende Hardwareports, hinter welchen sich entsprechende Register verbergen.
Im Falle der Logitech Maus gibt es vier derartige Ports bzw. Register:

- Data-Port
- Signature-Port
- Control-Port
- Config-Port

Diese 4 Ports sind jeweils über die Basis-Adresse (Data-Port) plus einen jeweiligen Offsetwert (Basisadresse+1 = Signature-Port, Basisadresse+2 = Control-Port,...) ansprechbar.
Die wesentliche Funktionalität verbirgt sich dabei hinter dem Data- und dem Control-Port. Aus dem Data-Port kann je nach gesetzten Bits im Control-Port jeweils die X-Koordinate (low value oder high value), Y-Koordinate (low-value oder high value) und der Button-Zustand herausgelesen werden.
Im folgenden konkret jeweils die Werte, die ins Control-Register geschrieben werden müssen, sowie der Wert, der dann jeweils aus dem DATA-Register herausgelesen werden kann:

Bits in CONTROL-Register (write)	Bits in DATA-Register (read)
10000000	X-Koordinate (Low-Wert, 4 niederwertigsten Bits)
10100000	X-Koordinate (High-Wert, 4 niederwertigsten Bits)
11000000	Y-Koordinate (Low-Wert, 4 niederwertigsten Bits)
11100000	Y-Koordinate (High-Wert, 4 niederwertigsten Bits) Buttons (3 Höchstwertigsten Bits)

2.3.2. Busmousedriver unter Linux

Unter diesem Hintergrund ist nun auch der existierende Linuxtreiber, z.B. In der 2.6er Kern-Serie zu betrachten.
Hier als Beispiel ein Ausschnitt aus dem Interrupthandler des Treibers (direkt entnommen aus dem Linux Quelltext unter `~/drivers/input/mouse/logibm.c`):

```
outb(LOGIBM_READ_X_LOW, LOGIBM_CONTROL_PORT);
dx = (inb(LOGIBM_DATA_PORT) & 0xf);
outb(LOGIBM_READ_X_HIGH, LOGIBM_CONTROL_PORT);
dx |= (inb(LOGIBM_DATA_PORT) & 0xf) << 4;
outb(LOGIBM_READ_Y_LOW, LOGIBM_CONTROL_PORT);
dy = (inb(LOGIBM_DATA_PORT) & 0xf);
outb(LOGIBM_READ_Y_HIGH, LOGIBM_CONTROL_PORT);
```

Wie gut zu sehen ist, besteht ein Großteil des Treibercodes aus low-level Bit und Port-Operationen. Nachteilig dabei ist, dass die Lesbar- und somit auch die Wartbarkeit des Treibercodes massiv unter eben jenen low-level Operationen leidet. Auch wenn dieses Beispiel noch recht einfach ist, lässt sich doch bereits jetzt erahnen, welcher Aufwand bei der Treiberentwicklung z.B. bei modernen Grafik- oder IO-Chipsätzen zu bewerkstelligen ist, um die volle Funktionalität der Hardware zu gewährleisten. Schlimmer noch: Ein Großteil des Treibercodes ist meist derartig in Bit- und Portoperationen verwickelt, dass nur noch mit erheblichen Aufwand Entwicklung oder gar Wartung des Treibercodes möglich sein wird. Eine Kapselung oder gar Typensicherheit, wie es die meisten Hochsprachen bieten, kommt bei dieser Art der Treiberentwicklung gar nicht mehr zum Vorschein.

2.3.3. Busmousedriver in DEVIL

Betrachten wir uns nun die gleiche Treiberfunktionalität, allerdings implementiert unter DEVIL. Man erstellt also eine neue Datei und beschreibt nun mittels der anfangs erwähnten DEVIL-DSL die Hardwareschnitstelle:
Als erstes muss das Gerät selber, respektive seine zur Verfügung gestellten Ports, definiert werden (folgende DEVIL Code-fragmente entnommen aus [2]):

```
device logitech_busmouse (base : bit[8] port @ {0..3})
```

Der Name des Gerätes wird mit *logitech_busmouse* angegeben. *base* stellt quasi einen Parameter dar, und wird später einmal die Hardwareadresse der Busmousehardware annehmen. *Bit[8]* gibt an, dass es sich bei den Ports um 8-Bit Register handelt, und *port@{0..3}* impliziert, dass sich ab der *base*-Adresse plus jedem Offset von 0 – 3 jeweils ein Port der Hardware befindet.
Als nächstes wird - ausgehend von den Ports - ein Register, d.h. eine Registervariable definiert:

```
register index_reg = write base @ 2, mask '1..00000' : bit[8];
```

Unsere Registervariable trägt den Namen *index_reg*, mit dem Schlüsselwort *write* wird dabei sichergestellt, dass auf diese Variable später nur schreibend zugegriffen werden kann. Außerdem wird definiert, wo sich dieses Register befindet: An der Basisadresse plus dem Offset 2, es wird also auf das CONTROL-Register gezeigt. Über das Schlüsselwort *mask* wird dabei angegeben, welche Bits im Register letztendlich variabel beschrieben werden dürfen, und welche Bits bei jedem Schreibvorgang einen fixen Wert zugewiesen bekommen: In unserem Fall bezeichnet *1..00000*, dass das höchstwertigste Bit bei jedem Schreibvorgang automatisch auf “1”, die fünf niederwertigsten Bit jeweils auf “0” gesetzt werden, sowie die beiden verbleibenden Bits die eigentliche variable Information dieser Registervariablen darstellen. Es könnte dieser Variablen theoretisch also im Grunde nur ein 2-Bit-Wert zugewiesen werden.
Auf Registervariablen alleine darf man allerdings im Kontext von DEVIL nicht direkt zugreifen,

deshalb wird nun zuguterletzt die eigentliche Arbeitsvariable angelegt:

```
private variable index = index_reg[6..5] : int(2);
```

Es wird hierbei eine "normale", d.h. im folgenden Kontext direkt benutzbare Variable vom Typ 2-Bit Integer angelegt. Der Speicherplatz dieser Variablen ist nun aber unsere zuvor definierte 2-Bit-Registervariable: *index_reg[6..5]* gibt dabei an, welche Bits der Registervariablen für unsere neue Arbeitsvariable verwendet werden. In dem Fall logischerweise die einzigen zwei variablen Bits aus *index_reg*, nämlich Bits 5 und 6.
Private legt fest, dass diese Variable nur innerhalb des folgenden DEVIL-Kontextes verwendet werden kann, die Variable später also nicht über die C-Schnittstelle des kompilierten DEVIL-Codes exportiert wird.
Betrachtet man sich bisher die Variablendefinition von Ports über Registervariablen bis hin zu den Arbeitsvariablen, so stellt man fest, dass über diese unterschiedlichen Abstraktionsebenen schon im Vorfeld eine Art Typenprüfung der einzelnen Variablen sichergestellt wird. Der DEVIL-Compiler würde sich später z.B. weigern, einen Code zu kompilieren, der Lesezugriff auf ein write-only Register erfordert oder einige Bits zu setzen, die explizit als "fix" markiert sind.
Die eigentliche Anwendung unserer neuen Variable *index* steckt nun im folgenden Code-Fragment:

```
register x_low  = read base @ 0, pre {index = 0}, mask '****....' : bit[8];
register x_high = read base @ 0, pre {index = 1}, mask '****....' : bit[8];
register y_low  = read base @ 0, pre {index = 2}, mask '****....' : bit[8];
register y_high = read base @ 0, pre {index = 3}, mask '...*....' : bit[8];
```

Hier werden nun erstmal wieder nur Registervariablen definiert, und zwar vier Stück. Die entscheidende Sache ist jedoch, dass diese vier Registervariablen nicht wie vielleicht zuerst vermutet auf vier unterschiedliche Register der Hardware zeigen, sondern letztendlich immer auf ein und dasselbe, nämlich auf das DATA-Register am Port *base* (plus Offset 0). Wir erinnern uns, dass das DATA-Register je nach Zustand des Control-Registers jeweils unterschiedliche Informationen bereithält. Genau hier liegt nun der Sinn des *pre*-Statements:
pre bedeutet, dass die Funktion innerhalb des „pre-Blocks" zuallererst vor dem weiteren Zugriff auf die Registervariable ausgewertet werden muss. Im Konkreten Fall: *register x_low = read base @ 0, pre {index = 0}* bedeutet: Lege eine read-only Registervariable an, die auf das Register beim Port *base*+0 zeigt (=DATA-Register), aber vor jedem Zugriff auf diese Registervariable setze die Variable *index* auf einen bestimmten Wert, in diesem Fall auf 0. Denn das Setzen der Variable *index* auf 0 hat wiederum den Effekt, dass das CONTROL-Register seinerseits auf den Wert "10000000" gesetzt wird, was wiederum zur Folge hat, dass das DATA-Register nun den momentanen low Wert der X-Koordinate der Maus einblendet, was sich auch im Namen der Variablen "x_low" widerspiegelt. Analog dazu werden nun auch die übrigen Register *x_high*, *y_low* und *y_high* definiert. Mittels *mask* wird zur Sicherheit noch eine Maske über diejenigen Bits definiert, die im Register relevant, d.h. definiert sind: Z.B. sollen bei *x_low* später nur die unteren vier Bits betrachtet werden. Ein versehentlicher späterer Zugriff auf die undefinierten oberen vier Bits würde vom BOSSA-Compiler dann bemängelt werden.
Zusammenfassend kann man sagen, dass das "multifunktionale" und vom CONTROL-Register abhängige DATA-Register somit entzerrt bzw. "demultiplext" wurde auf 4 unabhängige Register (-variablen).
Prinzipiell könnte man nun schon mittels dieser vier Koordinaten-Register vier öffentliche Variablen definieren und diese für die spätere Verwendung im eigentlichen Treibercode freigeben. Allerdings wird man dann später an dieser Stelle vor einem kleineren Problem stehen: Unsere vier Register mit den Koordinaten tragen ja immer die relative Koordinatenposition vom letzten Aufruf, und sind somit flüchtig (*volatile)*. Das bedeutet, dass ein zweiter Lesezugriff auf das Register normalerweise stets einen anderen Wert (z.B. 0) liefert. Da man aber z.B. im Kontext der Interruptbehandlung im späteren Treibercode auf diese Variablen meist noch öfters zugreifen muss, man aber gleichzeitig auch stets konsistente Werte erwartet, wäre ein Mechanismus wünschenswert, der die Variablenwerte automatisch sichert, und nur bei einem expliziten

Kommando diese neu aus dem Register einliest. Die Lösung dieses Problems findet sich in den folgenden Zeilen:

```
structure mouste_state = {
     variable dx = x_high[3..0] # x_low[3..0], volatile : signed int(8);
     variable dy = y_high[3..0] # y_low[3..0], volatile : signed int(8);
     variable buttons = y_high[7..5], volatile : int(3);
  };
```

Es wird eine Struktur (ähnlich wie in C) angelegt, die Variablen innerhalb dieser Struktur werden aber als *volatile*, d.h. flüchtig, gekennzeichnet. Gleichzeitig wird über den Konkatenationsoperator # der high und der low Wert der jeweiligen X- und Y-Registervariablen zu einen kompletten Koordinatenwert zusammengeführt. Die Variablen dx, dy und buttons sind dann übrigens (da sie nicht als *private* deklariert sind) die endgültigen Variablen, die nun auch später automatisch über die C-Inline-Interface-Funktionen exportiert und in den eigentlichen Treibercode direkt verwendet werden können. *volatile* bewirkt nun, dass später bei jeden Zugriff auf die Strukturvariable stets immer der gleiche („gecachete“) Wert zurückgegeben wird, solange bis die komplette Struktur über einen speziellen “refresh”-Befehl neu aus den Registern eingelesen wird (Siehe Praxisbeispiel weiter unten).
Schließlich stellt DEVIL auch noch weitere Sprachkonstrukte für den fortgeschritteneren Gebrauch zur Verfügung, wie z.B. Trigger oder Enumerations, auf die jedoch an dieser Stelle mit einem Verweis auf das DEVIL-Referenzhandbuch [3] nicht explizit eingegangen werden soll.

2.4. Praktische Anwendung

Wurde die Hardware erst einmal vollständig in DEVIL beschrieben, kann man die erstellte DEVIL-Datei mittels des dazugehörigen Compilers übersetzen. Solange dann beim Compilieren keine Fehler aufgetreten sind (hier kommt eine der klaren Vorteile von DEVIL zum Vorschein, nämlich die implizite Typ und Logikprüfung), wird aus dem DEVIL-Quellcode eine normale C-Include-Datei generiert, welche man nun in seinem zukünftigen Treibercode in der Regel ohne weitere Probleme einsetzen kann.
Die praktische Anwendung, z.B. im Linux-Busmouse-driver, ist so gesehen dann auch denkbar einfach: Die vom DEVIL-Compiler erstellte Include-Datei in den Quelltext einbinden, die Hardware mittels der von DEVIL automatisch bereitgestellten Inline-Stub-Funktion initialisieren (zu diesem Zeitpunkt wird dann auch der BASE-Port mit übergeben), und dann kann man schließlich direkt, einfach und vor allem sicher auf die Funktionalität der Hardware zugreifen:

- **bm_get_mouste_state** // Einlesen der „gecacheten“ Variablen
- **char dx = bm_get_dx()** // Stub-Funktion, um auf die DEVIL-Variable dx zuzugreifen
- **char dy = bm_get_dy()** // Stub-Funktion, um auf die DEVIL-Variable dy zuzugreifen

Dank der Cache-Struktur liefert nun ein wiederholtes Aufrufen von bm_get_dx immer wieder denselben Wert, solange bis mittels bm_get_mouste_state die Variablen direkt aus der Hardware neu eingelesen werden.

Zusammenfassend kann man sagen, dass, wenn man erst einmal die DEVIL-Spezifikation erstellt hat, das Erstellen des Treibers wesentlich einfacher, sicherer und übersichtlicher geworden ist. An dieser Stelle liegt auch eine der Visionen der Entwickler von DEVIL: Der Wunsch, dass Hardwarehersteller irgendwann einmal die Spezifikation ihrer Hardware in Form von DEVIL-Dateien zur Verfügung stellen: Der Treiberentwickler müsste dann - unabhängig vom Betriebsystem - nur noch die vorhandene DEVIL Spezifikation kompilieren, die generierte Include-Datei in seinen Treibercode einhängen und könnte sich dann direkt mit der eigentlichen Treiberfunktionalität befassen, ohne erst mühsam Makro-Schnittstellendefinitionen zu erstellen oder mit fehleranfälligen Bitoperationen zu arbeiten.

2.5. Kritische Betrachtung

Die Vorteile von DEVIL liegen sicherlich auf der Hand:
Die Strukturiertheit und Einfachheit des Treibercodes werden wesentlich erhöht. Ein schnelles Einlesen in den Treibercode ist auch für bisher nicht involvierte Entwickler in der Regel wesentlich einfacher möglich als mit dem konventionellen Entwicklungsweg. Positiv fallen auch die unter DEVIL ermöglichten Sicherheitsmechanismen auf: Konzepte wie Typenprüfungen oder Tests auf logische Plausibilität (Stichwort "Bitdreher") waren bisher auf dem herkömmlichen Entwicklungsweg mit low-level Code nur sehr schwer bis gar nicht zu beherrschen. Im Gegensatz zu den bei der herkömmlichen Methode verwendeten Makros bietet DEVIL dafür eine saubere Kapselung der Schnittstellenbelange quasi in einem eigenen Modul. Schließlich bleibt noch anzumerken, dass DEVIL auch zu einem gewissen Grad Plattformunabhängigkeit ermöglicht: Zwar ist der bisherige DEVIL-Compiler im Moment nur unter Linux (und seit kurzem auch Solaris) verfügbar, doch ist es prinzipiell kein allzu großes Problem, auch für andere Plattformen oder gar Architekturen den DEVIL-Compiler entsprechend anzupassen.
Der gravierendste Nachteil von DEVIL dürfte im Moment noch im Bereich der Performance liegen: In der Regel ist der durch die Inline-Stubs verursachte Overhead im Bereich von ca. 10% Performanceverlust anzusiedeln. Jedoch sind laut Entwicklerangaben momentan noch einige Optimierungen geplant, sowie spezielle Sprachkonstrukte für Zeitkritische Abschnitte, so dass man in diesem Bereich durchaus noch auf einige Verbesserung hoffen darf.

2.6. Ausblick:

Neben der weiteren Geschwindigkeitsoptimierung liegt ein weiteres Augenmerk der Entwickler auf den Aufbau einer Art public domain Bibliothek von DEVIL-Hardwarespezifikationen. Auf diese Weise soll die Verbreitung gefördert und das oft gebrauchte Argument von Treiberentwicklern, man müsse erst mühsam eine DEVIL-Spezifikation für eine Hardware schreiben, entkräftet werden. Der Vision der DEVIL-Entwickler schlechthin wäre freilich, dass die Hardwarehersteller schon von sich aus zusammen mit der Dokumentation ihrer Hardware auch gleich die komplette Schnittstellenspezifikation ihrer Hardware in DEVIL-Code offenlegen: Die verschiedenen Treiberentwickler für die diversen Betriebsysteme könnten dann ausgehend von dieser Spezifikation bequem ihren Treibercode entwickeln. Ob jedoch der Trend in der Treiberentwicklung wirklich in Richtung DEVIL gehen wird, bleibt offen. Sinnvoll wären Innovationen auf dem Gebiet der low-level Programmierung aber allemal.

3.BOSSA

3.1. Motivation:

Dass der Scheduler in moderneren Betriebsystemen, wie z.B. Linux, nicht gerade sehr modular in den OS-Code eingebettet ist, dürfte schon einmal jedem klar geworden sein, der einen Linux-Patch für einen neueren Scheduler (z.B. O(1)-Scheduler) eingespielt hat, oder gar selber versucht hat, an den entsprechenden Codestellen Veränderungen vorzunehmen. Der Schedulercode ist in der Regel quer über etliche Dateien des Betriebsystemquelltextes verteilt, weswegen generell Änderungen an diesen Codebestandteilen selten innerhalb weniger Zeilen Quelltext getan sind. Die praktische Implementierung eines Schedulers wird deswegen in der Regel auch als ein Problemfall des *cross-cutting-concerns* angesehen.
"Einfach mal den Schedulercode austauschen" ist demnach eine Aufgabe, die bisher wirklich nur routinierten Kernel-Entwicklern vorbehalten war. Dabei wäre es in vielen Situationen im Grunde nur von Vorteil, wenn es einen durchdachten Mechanismus gäbe, der eine schnelle und unproblematische Manipulation des Schedulers ermöglichen würde. Die Einsatzgebiete wären vielfältig, angefangen von Forschung bei Scheduler und Warteschlangenstrategien bis hin zu Spezialimplementierung von Schedulern für zeitkritische Applikationen im Privat- und Industrieeinsatz.

3.2. Lösungsansatz: BOSSA

Auch zu diesem Problembereich existiert wiederum einen vielversprechender Lösungsansatz: *BOSSA*.
BOSSA gehört wie vorher schon DEVIL ebenso zu den *DSLs*, also zu einer Klasse von Spezialprogrammiersprachen, die für spezielle Belange entworfen wurden. Im konkreten Fall eben für die "einfache" und sichere Entwicklung eines Schedulers. Der Hauptteil von BOSSA bildet dabei eine speziell angepasste Version des Linux-Kerns (gegenwärtig 2.4.24), der in der Hinsicht angepasst worden ist, dass alle Belange der Schedulerverwaltung in einer zentralen Schnittstelle zusammengefasst sind, dem sog. BOSSA-Runtime-System (*BOSSA-RTS*). Das BOSSA-RTS verwaltet somit alle scheduler-spezifischen Belange des Betriebsystemkerns und bietet zudem eine event-basierte Schnittstelle an, mittels der die Strategie des jeweiligen Schedulers feingranular beschrieben werden kann. In der Praxis wird das BOSSA-RTS dabei durch ein Kernel-Modul gesteuert, welches nachgeladen werden kann und somit die gewünschte neue Schedulerfunktionalität zur Verfügung stellt. Erzeugt wird dieses Kernel-Modul aus dem in der BOSSA-Sprache erstellten sog. *Policy-File*, in welchem die Funktionalität des Schedulers ereignisbasiert beschrieben wird. Der spezielle BOSSA-Compiler so wie der Standard C-Compiler (gcc) des Betriebsystems übersetzen dann dieses Policy-File in einem automatisierten Ablauf direkt in ein ein entsprechendes Linux-Kernel-Modul.

3.3. Die Bossa-DSL

Die Bossa-spezifische Sprache selbst ist – wie bereits erwähnt - eine event-basierte, C-ähnliche Hochsprache. Sie ermöglicht das Programmieren eines Schedulers, ohne dabei auf die speziellen Eigenschaften des darunterliegenden Betriebsystems (in diesem Fall Linux) eingehen zu müssen. Dennoch ist mit dieser Sprache fast jede erdenkliche Schedulerstrategie implementierbar.

Das Grundgerüst eines jeden in BOSSA geschriebenen Policy-Files sieht dabei wie folgt aus:

- Deklarationsteil
- Eventhandler
- Schnittstellenfunktionen

3.3.1 Deklarationsteil

Hier werden (wie der Name schon sagt) hauptsächlich verschiedene Variablen, Warteschlangen und Prozesszustände deklariert. Im Grunde alle Informationen, die ein Scheduler später einmal zum erfolgreichen arbeiten benötigt. Im folgenden sollen nun die entsprechenden Codezeilen der Deklaration des Standard Linux 2.4 Schedulers, den die BOSSA-Entwickler zu Demonstrationszwecken in BOSSA-reimplementiert haben, vorgestellt werden [4].

```
1:    default scheduler Linux = {
2:      type policy_t = enum {BOSSA_SCHED_FIFO, BOSSA_SCHED_RR, BOSSA_SCHED_OTHER};
3:
4:      process = {
5:        policy_t policy;
6:        int      rt_priority; // real int priority (0 if non RT)
7:        int      priority;   // normal priority (for RR and OTHER)
8:        int      ticks;     // clock ticks left on the current intslice
9:        system struct mm_struct system active_mm;  // process virtual address space
10:
11:   system struct mm_struct running_active_mm = NULL;
12:
13:     states = {
14:       RUNNING running : process;
15:       READY ready : sorted queue;
16:       READY expired : queue; // process which has expired its intslice
17:       READY yield : process;
18:       BLOCKED blocked : queue;
19:       TERMINATED terminated;
20:     }
21:
22:     ordering_criteria = {
23:       highest rt_priority, highest (true ? (priority + ticks) : 0),
24:       highest ((active_mm == running_active_mm) ? 1:0)
25:     }
26:    }
27:   [...]
28:   }
```

Als erstes wird der Name des Schedulers vergeben (*Linux*) und mitgeteilt, dass es sich hierbei um einen Default-Scheduler handeln soll, d.h. um denjenigen Scheduler, in welchen später automatisch alle weiteren neu erzeugten Prozesse eingehängt werden sollen (Zeile 1). Gleich danach erfolgt eine Definition eines aufzählenden Datentyps *policy_t* mit drei verschiedenen Bezeichern (Zeile 2). Diese drei Bezeichner stellen die unter dem standard Linuxkern verfügbaren Prozessklassen da (*FIFO* und *RR* für zeitkritische “Echtzeit”-Prozesse, *OTHER* für alle anderen normalen Prozesse).
Als nächstes wird in den Zeilen 4 – 9 die Prozessstruktur selbst definiert: Ein Prozess unter Linux zeichnet sich für den Scheduler demnach aus durch die Prozessklasse, zu welcher er gehört (FIFO, RR, OTHER), der Echtzeitpriorität (falls der Prozess zu einer Echtzeitklasse gehört) oder der Standard-Priorität (falls der Prozess zu der Standard-Prozessklasse, d.h. OTHER, gehört). Die Variable *ticks* gibt dabei die verbleibenden clock-ticks des jeweiligen Prozesses an, d.h. das Maß, welche Zeit der Prozess für die laufende Zeitscheibe noch übrig hat. Schließlich wird noch eine Struktur definiert, die den aktuellen virtuellen Adressraum des Prozesses widerspiegelt und mit *NULL* initialisiert (Zeile 11).
Schließlich müssen noch die verschiedenen Prozesszustände definiert werden (Zeilen 13-20): Die unter Linux üblichen Prozesszustände *RUNNING, READY* und *BLOCKED* werden hier auf einige

Variablen aufgeteilt: Da auf einem Einprozessorsystem immer nur genau ein Prozess gleichzeitig laufen kann, wird eine Variable *running* angelegt, vom Typ *process* (aus der Zustandsklasse *RUNNING*). Diese Variable wird später somit immer genau den Prozess repräsentieren, der aktuell von der CPU ausgeführt wird. Die Prozesse im Zustand READY werden hier durch drei Prozessvariablen abgedeckt: *ready* vom Typ *sorted queue* enthält die Liste aller laufbereiten Prozesse. *expired* enthält die Liste aller Prozesse, die zwar ebenfalls bereit sind, aber gerade ihre „Clockticks" verbraucht haben. *yield* schliesslich stellt den einen gerade von der CPU abgegeben Prozess dar, ist also wiederum nur vom Typ *process*. Schliesslich wird noch eine queue *blocked* aus der Zustandsklasse *BLOCKED* erstellt, die alle Prozesse enthält, welche momentan blockiert sind. *Terminated* zuguterletzt stellt die beendeten Prozesse dar, wird für den eigentlichen Schedulercode aber im Grunde nicht benötigt.
Als nächstes wird mittels *ordering_criteria* eine Ordnung auf die oben definierte sorted queue defniniert (Zeile 22-25): Da der Scheduler grundsätzlich im weiteren Verlauf, z.B. wenn der laufende Prozess blockiert oder verdrängt wurde, sich seinen nächsten Prozess immer aus der READY-Klasse holt, muss sichergestellt sein, dass diese Prozesse auch gemäß der aktuellen Scheduler-Strategie sortiert sind. Deshalb wurde die ready-queue im obigen Beispiel als *sorted* angelegt. Nur aus dieser sorted-queue wird sich der Scheduler dann den nächsten zu schedulenden Prozess holen. Im weiteren Schedulercode wird also immer nur festgelegt, wann und wie ein Prozess die CPU verliert. Den nächsten Prozess holen tut sich bei BOSSA der Scheduler grundsätzlich automatisch aus der sortierten ready-queue.

3.3.2 Eventhandler

Der eigentliche Code für den Scheduler wird bei BOSSA in den sog. *Eventhandlern* beschrieben, das sind die Abschnitte des Quelltextes, die beschreiben, wie der Scheduler auf bestimmte Ereignisse reagieren soll (z.B. Timerinterrupts, blocks, unblocks,..)
Betrachten wir uns nun exemplarisch einen Ausschnitt des Eventhandlercodes, wiederum aus der Linux 2.4 Strategie:

```
1:   handler (event e) {
2:
3:    On block.* {
4:       e.target => blocked;
5:       if (empty(READY)) {running_active_mm = NULL;}
6:     }
7:
8:     On unblock.* {
9:       if (e.target in blocked) {
10:         if ((!empty(running)) && (e.target > running))
11:            {running => ready; }
12:         e.target => ready;
13:       }
14:     }
15:
16:    [...]
17:
18:     On system.clocktick {
19:         if (running.policy != BOSSA_SCHED_FIFO) {
20:           running.ticks--;
21:           if (running.ticks <= 0) { // intslice expired ?
22:             running.ticks = (((running.priority)>>2)+1); // reload ticks
23:             if (running.policy == BOSSA_SCHED_RR) {
24:               running => ready;
25:             }
26:             else {
27:               running => expired;       // preempt running
28:             }
29:           }
30:         }
31:
```

In den Eventhandlern wird beschrieben, wie der Scheduler bei Auftreten von bestimmten (für das Scheduler-Verhalten relevanten) Events verfahren soll. Die Event-Variable *e* enthält hierbei immer Informationen über das aktuelle Ereignis. Die wichtigsten Ereignisse dürften dabei die Block- (Zeile 3-6), UnBlock- (Zeile 8-14) und Clocktick-Ereignisse (Zeile 18-30) darstellen. Bezogen auf das Beispiel wird *Block.** bei allen Ereignissen ausgeführt, bei denen der aktuell ausgeführte Prozess aus welchen Gründen auch immer blockiert wird. Der Code zum Block-Eventhandler ist so gesehen auch dementsprechend einfach: Der frisch blockierte Prozess, dargestellt durch *e.target*, wird einfach mittels des =>-Operators in die blocked-queue verschoben und somit dem Scheduler erst einmal entzogen (Zeile 4). Der Scheduler wird sich dann im Gegenzug sofort den nächsten Prozess aus der sortierten ready-Queue abholen. Das Ereignis für die Deblockierungen läuft dementsprechend ab: Der nicht mehr blockierte Zielprozess wird somit sofort in die ready-queue eingereit (Zeile 12) . Hat der deblockierende Zielprozess zudem eine höhere Priorität als der aktuell laufende Prozess, wird dem aktuell laufenden Prozess auch sofort die CPU entzogen, d.h. Er wird von running zurück auf die ready-queue gesetzt (Zeile 10+11). Auf diese Weise lassen sich preemptive Schedulerverfahren implementieren. Statt Wildcards wie *block.** lassen sich übrigens auch speziellere Ereignisse abfangen, wie z.B. *block.network.**: Dieser Eventhandler würde dann z.B. nur bei Prozessblockierungen innerhalb des Network-Subsystems anspringen. Auf diese Weise könnte man den Scheduler noch weiter spezialisieren.
Der Eventhandler für den Timer-Baustein (*system.clocktick)* stellt schließlich den wesentlichen Bestandteil des Zeitscheibenbasierten Schedulings dar (d.h. für alle Prozessklassen außer FIFO): Dieses Event wird in der Regel bei jedem "Tick" des internen Timerbausteins aufgerufen und hat zur Folge, dass dem aktuell ausgeführten Prozess jedesmal ein Tick von seiner zu Anfangs definierten *ticks*-variable abgezogen wird. Sind alle Ticks eines Prozesses aufgebraucht, werden dessen Ticks entsprechend dessen Priorität neu aufgefüllt und dem Prozess die CPU entzogen, d.h. Je nach Prozessklasse entweder gleich zurück in die ready-list oder in die expired-list gesteckt.

3.3.3 Schnittstellenfunktionen

Jeder Scheduler muss schließlich noch einige Interface-Funktionen implementieren, damit er für das BOSSA-RTS benutzbar wird. Die wichtigsten Funktionen dürften dabei *attach*, *detach* und *set_priority* sein.
Im Kontext des Beispiels der Linux 2.4 Scheduler-Strategie:

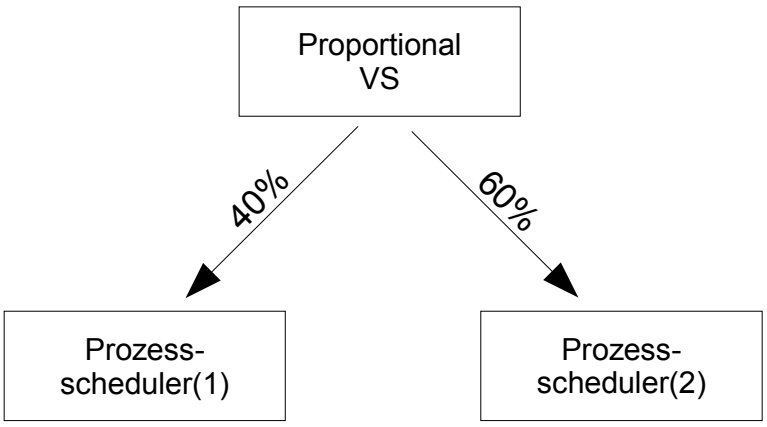
```
Interface = {
    void attach (process p, policy_t policy, int rt_priority, int niceval) {
        // insert checks here...
      p.policy = policy;
      p.rt_priority = rt_priority;
      p.priority = 20 - niceval;
      p.ticks = (p.priority>>2)+1;
      p => ready;
    }

    void detach (process p) {
      if (running_active_mm == p.active_mm) {
        running_active_mm = NULL;
      }
    }


    void set_priority(process p, int niceval) {
        // insert checks here
      p.priority = 20 - niceval;
    }
```

attach und *detach* werden ausgeführt, wenn ein neuer Prozess in den Scheduler eingehängt bzw.

ausgehängt (terminiert) wird, *set_priority* wird benötigt, um die Priorität eines vorhandenen Prozesses zu verändern.

3.4. Virtuelle Scheduler

Die bisherigen Methoden geben einen recht guten Einblick, wie man mit BOSSA einen einfachen Standard-Scheduler implementieren kann.
Seine kompletten Fähigkeiten entfaltet BOSSA jedoch erst bei einem Konzept, welches als "virtuelle Scheduler" aufgeführt ist, und mit dem es möglich ist, komplette Baumartige Schedulerhierarchien zu entwerfen.
Unter einem virtuellen Scheduler versteht man im BOSSA-Kontext einen Scheduler, der die CPU-Zeit zwischen mehreren vorhandenen Schedulern aufteilt (im Gegensatz zu den normalen Prozessscheduler). BOSSA liefert in diesem Kontext beispielsweise den sog. *virtual porportional scheduler* mit. Mit diesem virtuellen Scheduler ist es möglich, mehrere andere Scheduler zu verwalten und diesen jeweils einen fixen Anteil an CPU-Zeit zur Verfügung zu stellen. In der Praxis kann man sich den Ablauf dafür ungefähr so vorstellen: Mittels modprobe wird ein vorher mit BOSSA kompilierter Prozessscheduler in den Kernel geladen (zusätzlich zum bisher vorhandenen Defaultscheduler). Dieser neue Scheduler ist dann aber logischerweise ersteinmal deaktiviert, da das System vorerst nicht mit zwei parallel arbeitenden Schedulern umgehen kann. Erst der virtuelle Scheduler macht daher das Szenario sinnvoll: Nachdem man auch den virtuellen Scheduler als Modul in den Kern geladen hat, kann man nun mittels eines speziellen Konfigurationstools sowohl den standard-Scheduler als auch den zweiten Scheduler in den virtuellen Scheduler einhängen, wobei den beiden Prozessscheduler jeweils ein eigener Anteil an CPU-Zeit zugewiesen wird. Die folgende Grafik veranschaulicht dieses Szenario:



In diesem Beispiel wurden in den virtuellen Proportionalscheduler die beiden Prozessscheduler mit einer jeweils zugeteilten CPU-Zeit von 40- bzw. 60% eingehängt. Prozessscheduler 1 kann also z.B. wie gehabt seine Prozesse mit seiner ganz eigenen Schedulingstrategie verwalten, erhält aber insgesamt nie mehr als 40% der gesamten CPU-Zeit zu seiner Verfügung.
Auf die gleiche Weise lässt sich nun auch der virtuelle Scheduler selber wieder von anderen virtuellen Schedulern verwalten, so dass sich letztendlich eine komplette Baumhierarchie herausarbeiten lässt. Die einzige Voraussetzung ist, dass an den Blättern des Baumes jeweils die Prozessscheduler, und an den inneren Knoten jeweils immer die virtuellen Scheduler angesiedelt sind.
Die Implementierung eines virtuellen Schedulers unterscheidet sich im Prinzip nur wenig von dem

eines herkömmlichen Prozessschedulers. Deklariert wird der virtuelle Scheduler zu Beginn mit dem Keyword “virtual_scheduler”, Variablen von Typ 'process' sind nun logischerweise vom Typ 'scheduler', außerdem müssen Prozessschedulerrelevante Ereignisse wie Blockierungen mittels des “forwardImmediate”-Statements an die darunterliegenden Prozessscheduler weitergereicht werden.
Das folgende Beispiel aus dem BOSSA-Code des virtuellen Proportional-Schedulers veranschaulicht dies:

```
On system.clocktick {
  scheduler s = running;
  s => forwardImmediate ();
  s.ticks--;
  if (!(s in BLOCKED) && s.ticks <= 0) { // no more ticks so suspend the scheduler
    s.ticks = (s.proportion * period) / MaxProp;
    s => expired;
  }
}

On unblock.*, block.*, yield.*, process.* {
  next(e.target) => forwardImmediate();
}
```

Jedes Clocktick und alle (un-)blocking events werden dabei erst einmal mittels *forwardImmediate ()* an den jeweiligen, von diesem virtuellen Scheduler gerade bearbeiteten Kindscheduler weitergereicht (was im normalen Prozessscheduler dem aktuell laufenden Prozess entsprechen würde), danach aber werden beim clocktick-event zusätzlich noch die Restticks des aktuellen Kindschedulers dekrementiert, damit im Falle des Nullwertes der Restticks der nächste Kindscheduler an die Reihe kommen kann (nachdem zuvor der abgebende Scheduler noch gemäß seines proportionalen Anteils neu mit Ticks aufgefüllt wurde).
Zusammenfassend kann man sagen, dass sich mittels der virtuellen Schedulerfunktionalität von BOSSA auch komplexe Schedulerabläufe mit vielen unterschiedlich priorisierten Prozessklassen recht übersichtlich gestalten lassen.

3.5. Praktische Verwendung

Im praktischen Einsatz zeigt sich die Realisierung einer Schedulerhierarchie wie folgt:

- die einzelnen, mittels BOSSA kompilierten Kernelmodule in den BOSSA-Kern laden
- mittels eines mitgelieferten tools die einzelnen Prozessscheduler in die jeweiligen virtuellen Scheduler einhängen
  (Alternativ kann dieser Vorgang auch über die angebotene C-Schnittstelle im Kontext des jeweiligen Programms erfolgen)
- die jeweilige Anwendung über den attach-Aufruf der C-Schnittstelle des jeweiligen Schedulermoduls in den gewünschten Scheduler einhängen
  Bsp: *myscheduler_attach(0,BOSSA_SCHED_OTHER,0,20)* verschiebt die laufende Anwendung vom Standard-Scheduler in den Scheduler *myscheduler*

3.6. Kritische Betrachtung

Abschließend noch kurz ein Wort über die Vor- und Nachteile der gesamten BOSSA-DSL:
Positiv anzumerken ist sicherlich die Tatsache, dass mit BOSSA der Bereich Schedulerentwicklung auch für jene Personen erschlossen wird, die kein explizites Wissen über die internen Eigenheiten des darunterliegenden Betriebsystems haben. Insbesondere für Studenten und Forscher, die sich mit den Theorien von Schedulern und Warteschlangen beschäftigen, bietet

BOSSA so eine schnelle und relativ unkomplizierte Methode, “mal eben” eine neue Schedulerstrategie zu testen. Im Bereich der Lehre und zu Demonstrationszwecken besitzt BOSSA somit sicherlich eine berechtigte Existenz. Aber auch in praktischen Einsatzszenarien, in welchem für bestimmte Situationen ein speziell angepasster Scheduler benötigt wird, kann BOSSA zum Tragen kommen: Spezielle Anforderungen für zeitkritische Prozesse wie z.B. In der Industrie lassen sich mit BOSSA doch relativ feingranular auf die jeweilige Situation zurechtschneiden.
Hauptnachteil von BOSSA auf der anderen Seite dürften der Performanceverlust sowie der speziell angepasste Linux-Kern sein. Während der Performanceverlust von bis zu 10% in den meisten Fällen sogar noch verschmerzbar sein dürfte, führt die Tatsache, dass man auf einen speziell angepassten Kernel angewiesen ist, in vielen Situationen zu hohen Hindernissen: Viele Patches – gerade bei kritischen Sicherheitslücken – werden nicht mehr einwandfrei auf den BOSSA-Kern anwendbar sein, ebenso ist man darauf angewiesen, dass die BOSSA-Entwickler ihren spezialisierten Kern jeweils den jeweiligen Entwicklungsstand anpassen. Momentan (Stand Mai 2004) ist der neueste BOSSA-angepasste Kern ein Linux 2.4.24 Kern, Unterstützung für die 2.6er Serie ist momentan noch nicht in Sicht. Letztendlich ist es deshalb mehr als fraglich, in welchem Umfang sich derartig spezialisierte DSL generell durchsetzen werden.

4.Quellen

Allgemeine Informationen

Devil Projekthomepage
http://compose.labri.fr/prototypes/devil/

Bossa - A Framework for Scheduler Development
http://www.emn.fr/x-info/bossa/

Referenzen

[1] Devil: An IDL for Hardware programming - slides
http://compose.labri.fr/documentation/talks/osdi2000.pdf

[2] Devil: An IDL for Hardware programming
http://compose.labri.fr/documentation/papers/osdi00.pdf

[3] Devil: Reference manual
http://compose.labri.fr/prototypes/devil/doc/devil-0.4-refman.pdf

[4] Reimplementierung des Linux-Schedulers in Bossa:
http://www.emn.fr/x-info/bossa/policies/Linux.bossa