Konzepte von Betriebssystem-Komponenten

JX

Michael Schmidt
swmeschm@cip.informatik.uni-erlangen.de

23. Januar 2003

1 Überblick

JX ist ein Single-Address-Space-Betriebssystem, welches fast vollständig in Java geschrieben ist. Es basiert auf einem Mikrokernel, dessen Größe bei ungefähr 100kBytes liegt und der in C und Assembler verfasst ist. Das Betriebssystem ist in sogenannte “Domains” unterteilt, welche die Ressourcenverwaltung und Sicherheitsmerkmale implementieren. Dabei gibt es eine spezielle Domain, die sogenannte “DomainZero”, die den Kern darstellt und einige Grunddienste zur Verfügung stellt, z.B. Namensdienst, Scheduler.

2 Allgemeines

2.1 Entstehung

Das Java-basierte Betriebssystem JX wird in Erlangen am Lehrstuhl 4 “Verteilte Systeme und Betriebssysteme” von einer kleinen Pojektgruppe entwickelt.

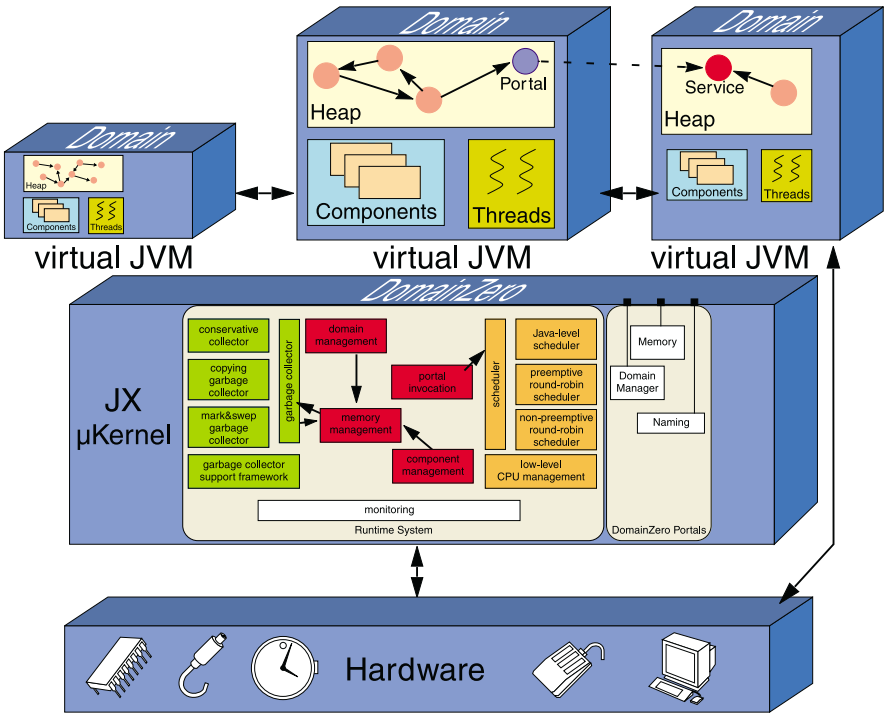
Die Entwicklung des Betriebssystems wurde 1996 von Michael Golm mit der Entwicklung einer Meta-Architektur für die JVM begonnen, welche schließlich 1999 zu den ersten Grundzügen von “JX” führte. In der Zwischenzeit hat das Betriebsystem im Zuge von Studien- und Diplomarbeiten stetig Veränderungen und Ergänzungen erfahren, wobei unter anderem Treiber für gängige Hardwarekomponenten, ein Window Manager und optimierte Compiler und Übersetzer entwickelt wurden.

2.2 Ziele

Die Entwicklung von JX ist den Unzulänglichkeiten heutiger Betriebssysteme zu verdanken, die meist vollständig in C und Assembler implementiert sind. Mit JX sollte ein Betriebssystem auf reiner Typsicherheit entwickelt werden, welches trotz der unterschiedlichen Architektur gegenüber heute gängigen Betriebssystemen, leistungsmäßig mit diesen konkurrieren kann. Hierfür war der von Java gebotene Sicherheitsmechanismus basierend auf einem typsicheren Zwischencode und dessen Verifizierung zur Laufzeit sehr gut geeignet.

Zusammenfassend kann JX als ein anpassungsfähiges, flexibles offenes Betriebssystem mit hohen Robustheitsansprüchen beschreiben.

3 Aufbau des Betriebssystems



3.1 Komponenten des Betriebssystems

3.1.1 DomainZero / Kern

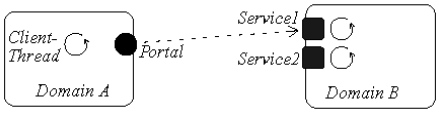
DomainZero wird automatisch mit dem Start des System geladen und fungiert als Schnittstelle für alle Systemaufrufe anderer Domains im Mikrokernel, welcher dem Betriebssystem JX zugrunde liegt. Der Kern ist verantwortlich für die Initialisierung des Systems, die Kontextumschaltung bei Prozesswechsel und schließlich für das low-level Domain-Sicherheitsmanagement. Es existieren grundsätzlich zwei Anforderungsmöglichkeiten der Anwendungen gegenüber dem Kern: direkt über den Aufruf von Diensten und indirekt über Anforderungen an das Java-Laufzeit-System. DomainZero implementiert einen preemptiven Scheduler zur Verwaltung der einzelnen Domains. Innerhalb jeder Domain kommt ein zweiter Scheduler zum Einsatz, der für das Scheduling der Threads innerhalb der Domain verantwortlich ist.

3.1.2 Domains

Domains sind eine Art Container, in denen die Anwendungen in JX gekapselt ablaufen, sie sind einem Prozess sehr ähnlich und bestehen zu 100% aus Java-Code. Da JX auf einem Microkernel basiert, müssen die in monolithischen Systemen vorhandenen und in JX erwünschten Dienste in eigenen Domains ausgeführt werden. Jede Domain kann folgende Objekte enthalten: Komponenten (Java-Klassen), Heap, Speicherbereich für Code und Stack, Threads, Dienste, Portale und Interrupt-Handler.
Jede Domain kann ihren eigenen Speichermanager für den Heap, einen eigenen Scheduler für die Threads der Domain und einen eigenen Gargabe Collector implementieren. Hieraus bietet sich die Möglichkeit, die Verwaltungsdienste einer Domain für bestimmte Aufgaben zu optimieren und somit eine effizientere Bearbeitung zu erzielen.

3.1.3 Portale

JX bietet zur Interprozesskommunikation ein eigenes Konzept basierend auf “Portalen” an. Diese Portale bestehen aus einem Objekt, einer Portalschnittstelle, einem Dienstzähler und einem Servicethread. Es ist möglich, den angebotenen Dienst beim Namensdienst von DomainZero zu propagieren, um einen Zugriff auf selbigen über den Servicenamen zu ermöglichen.



Führt ein Thread einen Methodenaufruf an einem Portal aus, so wird der lokale Scheduler der dienstanbietenden Domain ausgeführt, wobei dieser über die Ausführung des Aufrufes entscheidet. Der aufrufende Thread wird während der Verarbeitung des Portalaufrufes durch den Dienstthread blockiert. Hierfür werden die Parameter nach erfolgter Sicherheitsprüfung bezüglich Validität an den Dienst übergeben und der Dienstzähler wird inkremenitert. Nach Beendigung der Bearbeitung durch den Dienstthread wird der aufrufende Thread deblockiert und der Dienstzähler dekrementiert. Der angebotene Dienst kann erst dann komplett beendet werden, wenn der Dienstzähler gleich 0 ist, wodurch eine komplette Abarbeitung einmal akzeptierter Anfragen gewährleistet wird.

Portale können ebenfalls als Parameter eines Portalaufrufes übergeben werden. Dies ermöglicht sehr flexible Kommunikationswege zwischen verschiedenen Diensten, da diese gemeinsame Ressourcen, z.B. Memory Objekte, einfach und sehr effizient nutzen können.

Basierend auf dem Domain-Konzept von JX gibt es zwei unterschiedliche Arten von IPC-Vorgängen in JX, die sich vor allem in der Art der Parameterübergabe unterscheiden.

IPC bei JX

	Parameterübergabe
Intradomain-Aufruf	call-by-reference
Interdomain-Aufruf	call-by-value (Kopieren)

Basierend auf dem Portal-Prinzip gibt es in JX noch die von DomainZero zur Verfügung gestellten “Fast Portals”, die sich nur durch den Ausführungskontext der Anfragen unterscheiden. Bei normalen Portalen wird die Bearbeitung der Anfrage in der dienstanbietenden Domain ausgeführt, wohingegen bei einem “Fast Portal” die Anfrage immer im Kontext der dienstaufrufenden Domain verarbeitet wird. Ein solches “Fast Portal” wird zum Beispiel verwendet, um ein neues Memory Objekt (siehe Punkt 3.3 auf Seite 4) zu erstellen. Sie werdenzum Beispiel ebenfalls verwendet, wenn ein Domainscheduler den Prozessor für die Bearbeitung durch andere Domains freigeben möchte und hierfür die yield()-Methode aufruft. Hierbei ist es nötig, die yield()-Methode, welche über ein “Fast Portal” von DomainZero zur Verfügung gestellt wird, im Kontext des Aufrufers auszuführen, da ansonsten nicht der aufrufende, sondern der bearbeitende Thread blockiert würde.

3.2 Der JX-Translator

Bei JX handelt es sich, wie bereit erwähnt, um ein Single-Address-Space Betriebssystem, sprich alle Speicherbereiche der einzelnen Domains, auch DomainZero, liegen im selben Adressraum. Bedingt durch diesen Umstand ist die Einhaltung der Speichergrenzen einzelner Domains unumgänglich, um einen stabilen Ablauf des Systems zu gewährleisten.

In Java werden die einzelnen Objekte erst zur Laufzeit aus dem Bytecode in Maschinensprache übersetzt, um Plattformunabhängigkeit zu gewährleisten. Hierzu prüft der Bytecode-Verifizierer zuerst den Code auf Einhaltung der Java-Regeln. Dieser Vorgang garantiert grundsätzlich Typsicherheit. Außerdem führt der in JX verwendete Verifizierer zusätzliche spezifische Prüfungen bezüglich Interrupthandlern, Memory Objekten und Schedulern durch.

Nachdem der Bytecode erfolgreich verifiziert wurde, führt der JX-Translator Sicherheitsprüfungen und Code-Optimierungen durch, die maßgeblichen Einfluss auf die Performance von JX haben. Es ist möglich, für jede Domain einen eigenen Translator zu implementieren, wodurch anwendungsspezifische Übersetzungsstrategien ermöglicht werden. Dies hat jedoch zur Folge, dass ein Objekt in verschiedenen Domains unterschiedlich übersetzt werden kann.

Neben der Code-Optimierung ist der JX-Translator maßgeblich für den Speicherschutz im Betriebssystem JX verantwortlich, dabei muss er den durch die JVM spezifizierten Speicherschutz durch entsprechende Verfahren zur Übersetzungszeit nachbilden. Geschieht dies nicht, ist offensichtlich kein Speicherschutz im Betriebssystem vorhanden.

Der JX-Translator ist neben den obigen Aufgaben ebenfalls noch für die Garbage Collection und den direkten Speicherzugriff (DMA) von großer Wichtigkeit. Bedingt durch das Sicherheitssystem in JX wäre ein Direct Memory Access (DMA) normalerweise nicht möglich, da bei diesem Verfahren auf Speicher außerhalb der zum Gerät gehörenden Treiber-Domain zugegriffen werden muss. Jedoch kann der JX-Translator während der Übersetzung des Codes durch das Verwenden von sogenannten Plugins, die Methodenaufrufe durch entsprechende Maschinenbefehle ersetzen, solche Zugriffe ermöglichen. Diese Plugins unterliegen jedoch nicht dem Speicherschutzkonzept von JX und müssen somit als vertrauenswürdig angesehen werden.

3.3 Memory Objekte

Java bietet zur Verwaltung großer Datenmengen Arrays an, welche jedoch für die Verwendung in einem Betriebssystem, bedingt durch ihren Aufbau, nicht geeignet sind, da sie keinerlei Zugriffsschutz und keine Methoden, um Teilbereiche in eigene Memory Objekte auszugliedern, bieten. Aus diesem Grund hat man für JX spezielle Speicherstrukturen, die so genannten “memory objects” entwickelt, die genau diese Unzulänglichkeiten umgehen. Die Memory Objekte werden über den Aufruf eines “Fast Portal” erstellt und können zwischen Domains als Parameter in Form eines Portals übergeben werden. Dieses Vorgehen ermöglicht es mehreren Threads mit geringem Aufwand auf den selben physikalischen Speicher zuzugreifen und fungiert somit als Grundlage für viele im System verfügbare Dienste.
Der Zugriff auf die Memory Objekte funktioniert faktisch über Methodenaufrufe, wobei der JX-Translator diese Aufrufe durch Maschinenbefehle ersetzt, um einen möglichst schnellen Zugriff auf den zugrunde liegenden Speicher zu erzielen.

Interessant ist dieses Konzept vor allem für die Vermeidung von Speicheroverhead bei der Datenübergabe über Aufrufparameter und zur Interdomainkommunikation über gemeinsame Speicherobjekte, wobei dieses Verfahren dem des Shared Memory prinzipiell sehr ähnelt.

3.4 Interrupt-Handling

In JX wurde das Interrupt-Handling als zweistufiges Konzept implementiert. Tritt ein Interrupt ein, so wird dieser durch Aufruf der handleInterrupt-Methode eines vorher installierten Interrupthandlers bearbeitet. Diese Methode wird in einem eigenen Thread ausgeführt während andere Interrupt auf der entsprechenden CPU deaktiviert sind. Bisher ähnelt das Vorgehen dem eines first-level-Interrupthandlers in anderen Betriebssystemen. Da die Bearbeitung des Interrupts unter Umständen zu einem Blockieren des Systems auf unbegrenzte Zeit führen kann, wird vom Verifizierer in allen Klassen, die eine handleInterrupt-Schnittstelle implementieren, vor Ausführung des Codes auf eine entsprechende zeitliche Begrenzung geprüft - es wird vorher also eine Abschätzung der maximalen Laufzeit durchgeführt. Im allgemeinen erscheint eine solche Abschätzung eher schwer realisierbar, da Schleifen, Sprünge etc. eine solche Angabe schier unmöglich machen. Daher sollten in den handleInterrupt-Methoden nur einfache Codesequenzen verwendet werden, bei denen ein Stillstand des Systems ausgeschlossen werden kann: lineare Anweisungen oder Schleifen usw. Aus diesem Problem heraus wird die handleInterrupt-Methode normalerweise den entsprechenden Geräteinterrupt benachrichtigen und dann einen Thread starten, der den Interrupt asynchron abarbeitet. Dadurch wird eine Deblockierung des Systems nach der Abarbeitung des 1st-level Interrupt-Handlers ermöglicht.

4 Performance

Im Folgenden soll nunmehr ein IPC-Performancevergleich zwischen JX und anderen Betriebssystemen erfolgen. Die gemessenen IPC Zyklen der einzelnen Betriebssysteme lassen sich jedoch nicht gänzlich miteinander vergleichen, da die Betriebssysteme zum Teil unterschiedliche Architekturen verwenden und auf unterschiedlichen Systemen arbeiten:

IPC-Performance im Vergleich

System	IPC Zyklen
L4KA (PIII 450)	800
Fiasco/L4 (PIII 450 MHz)	2610
J-Kernel (Pentium Pro 200 MHz)	440
JX (hosted, Linux 2.2.14, PIII 500MHz)	7100
JX (native, PIII 500MHz)	650

Trotz der unterschiedlichen Testbedingungen kann man JX im nativen Modus im Vergleich zu Fiasco/L4, der aus reinem C-Code besteht, einen durchaus guten Wert bescheinigen. Ein Vergleich zum L4KA ist nicht angebracht, da dieser sehr viel optimierten Assembler-Code enthält.

Als zweites soll noch ein Vergleich mit einem monolithischen Betriebssystem (Linux) im Bereich des Dateisystems erfolgen. Für diesen Test wird JX so umkonfiguriert, dass es als quasi-monolithisches System arbeitet, hierzu werden alle Dienste in einer Domain co-located.

Dateisystem-Vergleich

System	Durchsatz (Mbyte/s)	Latenzzeit (mikrosek/4kB)
Linux	400	10.0
JX	201	19.9
JX co-located	213	18.7

Der vorliegende Unterschied sieht noch relativ groß aus, er ist jedoch durch den nicht optimierten Compiler für das Dateisystem zu erklären, mit entsprechenden Optimierungen scheint eine Annäherung an Linux durchaus möglich.

5 Zusammenfassung

JX ist der Beweis dafür, dass es möglich ist, ein Betriebssystem auf dem heutigen Stand der Technik und Forschung basierend auf Typsicherheit in nativem Java ohne Spracherweiterungen zu realisieren. Durch die Verwendung von Java und dem Domain-Konzept gewinnt JX im Vergleich zu anderen Betriebssystemen, die in C oder Assembler programmiert sind, an Sicherheit und Robustheit. Außerdem bietet die objektorientierte Programmierung eine sehr hohe Modularität, Erweiterbarkeit und Flexibilität in der Systemkonfiguration, die es ermöglicht, JX je nach Anwendungsgebiet äußerst individuell zu gestalten.

JX läuft auf Standard-PCs, bietet Treiber für gängige Geräte, verwendet ein erprobtes Dateisystem (ext2), bietet Netzwerk- und Mehrprozessorunterstützung und ist insgesamt gesehen nicht nur ein Forschungssystem, sondern durchaus für den realen Einsatz geeignet.

6 Literatur

Golm, Michael; Felser, Meik; Wawersich, Christian; Kleinöder, Jürgen: The JX Operating System
In: USENIX Association (Hrsg.): General Track 2002 USENIX Annual Technical Conference (2002 USENIX Annual Technical Conference Monterey, CA 10-15 June 2002). 1. Aufl. 2002, S. 45-58. ISBN 1-880446-00-6

Golm, Michael; Bellosa, Frank; Kleinöder, Jürgen: Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System .
In: Hotos (Hrsg.): HotOS 2001 (The 8th Workshop on Hot Topics in Operating Systems Elmau/ Oberbayern, Germany 20-23 May 2001). 2001

Golm, Michael; Kleinöder, Jürgen: Ubiquitous Computing and the Need for a New Operating System Architecture. Erlangen: Universiät Erlangen-Nürnberg. 2001 (TR-I4-01-09).

Felser, Meik; Golm, Michael; Wawersich, Christian; Kleinöder, Jürgen:
Execution Time Limitation of Interrupt Handlers in a Java Operating System.
In: ACM (Hrsg.): Tenth ACM SIGOPS European Workshop (Tenth ACM SIGOPS European Workshop Saint-Emilion, France 22-25 Sept. 2002)

http://www4.informatik.uni-erlangen.de/Projects/JX/DA/DA-I4-01-05-Wawersich.pdf
Diplomarbeit: Design und Implementierung eines Profilers und optimierenden Compilers für das Betriebssystem JX, Christian Wawersich, April 2001, DA-I4-2001-05

http://www4.informatik.uni-erlangen.de/Projects/JX/DA/DA-I4-01-10-Alt.ps
Diplomarbeit: Ein Bytecode-Verifier zur Verifikation von Betriebssystemkomponenten, Martin Alt, July 2001, DA-I4-2001-10

Weitere Informationen sind unter www.jxos.org verfügbar.