Konzepte von Betriebssystem-Komponenten

WS 2002/2003

JavaOS

Stefan Gabriel
Stefan.M.Gabriel@stud.uni-erlangen.de

16. Januar 2003

1 Einleitung

Um Java Applikationen auszuführen benötigt man gewöhnlich ein Hostbetriebssystem, auf dem eine Java Virtual Machine (JVM) läuft. Diese sorgt dann indirekt über das Hostsystem für die Ausführung des Bytecodes.

Ein neues Konzept wurde 1996 von Sun vorgestellt, welches ab 1998 von IBM mitentwicklet wurde. Das Hostsystem sollte durch das speziell auf die Ausführung von Java Code abestimmtes Betriebssystem JavaOS ersetzt werden. Dadurch kann dieser schneller ausgeführt, die Hardwareanforderungen jedoch minimiert werden. JavaOS stellt aber die volle Funktionalität einer gewöhnlichen Java Virtual Machine (JVM) zur Verfügung, die schon vorhandenen Applikationen sind also ausnahmslos lauffähig. Außerdem kann die Mächtigkeit der Programmiersprache Java auch bei der Treiberentwicklung genutzt werden. Dadurch sind Treiber plattformunabhängig, müßen also nur einmal entwickelt werden. Zudem ist JavaOS als Client-Betriebsystem konzepiert, der Administrationsaufwand und die damit verbundenen Kosten sinken folglich.

Mit der Begründung, es gebe nun hinreichen performance-optimierte Virtual Machines für Java, wurde dennoch 1999 die Entwicklung von JavaOS eingestellt. Das durchaus interessante Konzept wurde jedoch von anderen Entwicklern aufgegriffen und Projekte wie JX wurden gestartet.

2 Betriebssystem-Architektur

JavaOS ist von grundauf daraus ausgelegt eine Basis für das Java Development Kit JDK zu sein, ist aber dennoch selbst zu 85% in Java geschrieben. Die JavaOS Runtime Environment schließt die Standardelemente JVM, Garbage Collector (GC), Class Loader, Just in Time Compiler (JIT) und Basisklasses ein. Zusätzlich enthält es aber Hosting Klassen, Gerätetreiber und Systemdienste. Die wichtigsten Systemdienste sind die Systemdatenbank, das Ereigniss System, Service Loader und die Treiber. Noch grundlegendere Funktionen wie Threads, Interrupts, Monitors, low-level Speichermanagment oder JVM Systemfunktionen stellt der Microkernel zur Verfügung. Dies wird mit nativem Code (C und Asembly) realisiert, und ist dadurch plattformabhängig. Eine Abstraktionsschicht genannt JavaOS Plattform Interface (JPI) macht es jedoch möglich, daß die Runtime sehr einfach auf eine Vielzahl von Microkernel portiert werden kann. Da JavaOS für die Verwendung auf verschiedenartigen Plattformen konzipiert ist, kann es das komplette JDK, eine verkleinerte Version, das Personal JDK, oder auch nur das Embedded JDK für eingebettete System zur Verfügung stellen.

Im folgenden sollen die verschieden Schichten von JavaOS genauer erläutert werden: Beginnend bei der API, der Schnittstelle, die dem Anwendungsprogrammierer auch schon für die JVM mit Hostbetriebsystem zur Verfügung stand, schrittweise tiefer ins Innere des Betriebsystem, bis schließlich ganz hinab zum Kernel.

2.1 Java API / Hot Java

Die Java API (Application Programmming Interface) ist exakt die selbe wie auf anderen Systemen, womit sichergestellt wurde, daß die schon verhandene Software auch unter JavaOS benutzt werden kann. Auch die Hosting Klassen für Netzwerk und Windowing (AWT) sind present. Sie mußten natürlich anders implementiert

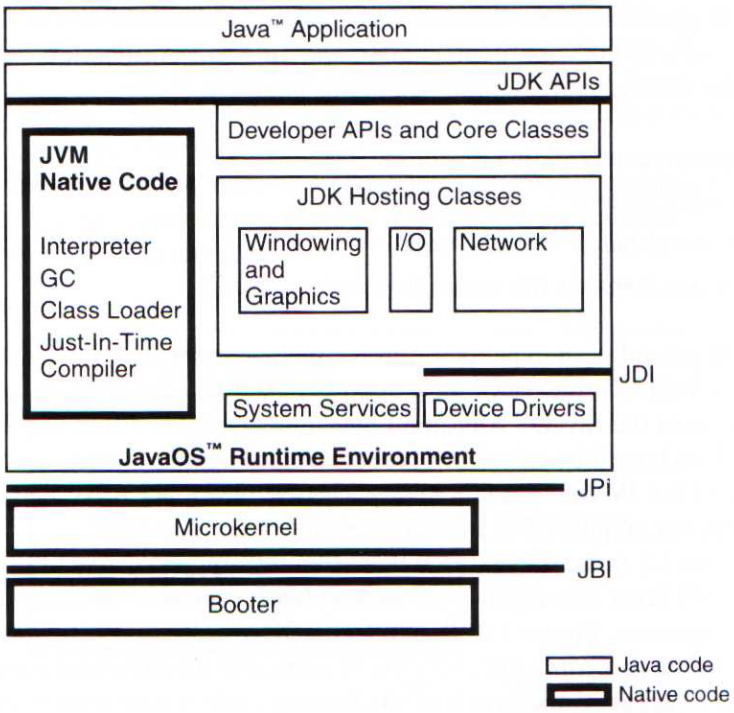

Abbildung 1: JavaOS Architektur

und an JavaOS angepaßt werden, doch sie bieten die gewohnten Schnittstellen. Folglich kann man jedes Java Applet und jede Application sowohl mit einer JVM unter JavaOS als auch unter einem anderen Hostsystem ausführen. Es kann jedoch nur eine JVM und damit eine Application gleichzeitg in JavaOS existieren, daher ist als Standardprogramm Hot Java, ein Webbrowser von Sun gedacht. Dieser ermöglicht simultanes ausführen von Applets in verschiedenen Fenstern, stellt also die Desktopumgebung dar. Dadurch wird JavaOS zum multitaskingfähigen, grafisch orientierten Betriebssystem.

2.2 Java Virtual Machine (JVM)

Wie in anderen Java unterstützenden Systemen dient die Virtual Machine zum ausführen des Byte-Codes, aber hier ist sie zusätzlich die Infrastruktur des JavaOS. Sie interpretiert auch den Byte-Code der System-Klassen, kümmert sich um Ausnahmebehandlung, managt fast den gesammten RAM des Computers und handhabt die gleichzeitige Ausführung von Threads. Dies ganze wird durch eine JVM realisiert, die fast der des JDK für Solaris entspricht. Nur der Speicherverwaltung wurde angepaßt.

2.3 JavaOS System Database (JSD)

Die Systemdatenbank dient zur Konfiguration von JavaOS, aber auch von den Treibern und Anwendungen. Sie ist eine als Baum angeordnete Sammlung von Java Objekten, die über einen Pfadnamen angesprochen werden können. Die persistenten Teile der Datenbank, darunter Informationen über die Benutzer oder über Konfiguration des Computers werden bei Bedarf vom Server geladen. Nicht beständige Einträge, z.B. wer momentan eingeloggt, oder welche Hardware gerade angeschloßen ist, werden lokal erzeugt. Es stehen standardmäßig sechs Teilbäume zur Verfügung: Software, Config (Benutzer- und Computerinforamtionen), Temp, Device (Topologie der Busse und Geräte), Alias (Referenzen mit anschaulichen Namen) und Interface (zur Verfügung stehende Schnittstellen, z.B. für eine Maus). Die Systemdatenbank unterstützt Suchfunktionen, die es unter anderem ermöglichen zu einem Systemdienst wie Drucken, die passende Schnittstelle, über die man dann darauf zugreifen kann, zu finden.

Bei jedem Verändern der Datenbank wird dies durch einen Event bekanntgemacht. So kann z.B. nach dem Einfügen eines neu entdeckten Gerätes sofort darauf reagiert werden und etwa durch den JavaOS Service Loader (JSL) die entsprechende Schnittstelle eingetragen werden.

2.4 Event System

Das Event System ist ein mächtiges Mittel, mit dem Objekte miteinander kommunizieren können. Im gewöhnlichen JDK wird dazu ein peer-to-peer Mechanismus verwendet, bei dem der Erzeuger die Information über den Aufruf einer Methode beim Empfänger übergibt. Die Emfänger müssen sich dazu vorher beim Erzeuger registriert haben.

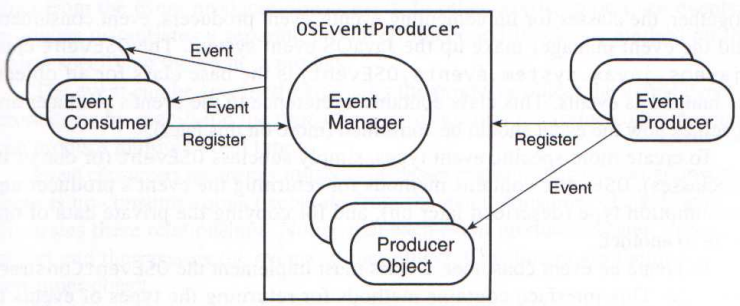

Abbildung 2: Event Manager

In JavaOS dagegen existiert ein Eventmanager. An dieser zentralen Stelle registrieren die Erzeuger, welchen Typ von Event sie produzieren. Standardtypen sind z.B. buffer empty, data ready, man kann Typen aber auch selbst definieren. Die Empfänger melden dem Eventmanager, welchen Typ sie empfangen möchten, und so kann er die Events weiterleiten. Will das Empfängerobjekt die Events jedoch nicht typspezifisch sondern erzeugerspezifisch empfangen, so kann es sich die `OSEventProducerEnumeration`, eine Liste mit allen Erzeugern, anfordern und den entsprechenden heraussuchen. Bei diesem kann er sich dann anmelden, bei Bedarf auch exklusiv. Ein anderes Mittel ist die Auswahl der Events durch einen Filter. Empfänger können dazu einen String beim Eventmanager registrieren. Beim Auftreten eines Events überprüft dieser dann ob es einen solchen Vergleichsstring hat. Wenn ja, überprüft er ob der bei ihm registrierte String ein Substring ist und stellt das Event entsprechend zu oder nicht. Für jedes Event gibt es eine Liste, die die Reihenfolge des Empfangs regelt. Will der Empfänger an einer bestimmten Position in der Liste der Empfänger stehen, so muß er das Interface `OSEventOrderdConsumer` implementieren. Dann teilt der Eventmanager dem Objekt eine bestimmte Position zu und fragt, ob diese aktzeptabel ist. Wenn nicht, versucht er einen anderern Platz zu finden. Ist keine geeignete Stelle mehr frei, so wird der neue Empfänger zurückgewiesen. Ein Event wird an jeden dafür registrierten Empfänger geschickt, dabei ist jedoch zu beachten, daß eine Veränderung des Eventobjekts sich auf alle nachfolgenden Empfänger auswirkt. Daher gibt es zusätzlich exklusiven und konkurierenden Empfang. Beim ersteren gibt es nur einen Empfänger, das zweite ermöglicht Empfängern, die `OSOrderdEventConsumption` implementieren, den Eventmanager an der weiteren Auslieferung zu hindern. Für die eigentlich nachfolgenden Emfänger bleibt das Event unbemerkt.

2.5 JavaOS Service Loader (JSL)

Der Service Loader dient zum Laden und Entladen von Softwarekomponenten eines *bundle*. Ein *bundle* ist eine Ansammlung von Javakomponenten wie Packages, Applikationen, Gerätetreiber und Dateien deren Eigenschaften von einem Objekt genannt *business card* beschrieben werden. Ein *business card* ist ein Eintrag in der JSD, der beim Booten vom Server geladen wird, ein *bundle* ist typischerweise ein .JAR oder .ZIP Archiv auf einem Server. Es gibt mehrere Wege wie es zu einem Laden eines Dienstes kommen kann. *Loading on discovery* reagiert auf das nachträgliche Eintragen eines Dienstes in die JSD und versucht diesen zu starten. Bei der Entdeckung einer neuen Hardwarekomponente lädt der JSL den passenden Dienst zusammen mit dem Treiber per *loading on matching* und macht einen ensprechenden Eintrag in die JSD. Schließlich kann ein bestimmter Dienst auch erst bei einer wirklichen Nachfrage geladen werden. Dieses *loading on demand* hat den Vorteil, daß der Garbage Collector diesen Speicherplatz wieder freigeben kann, wenn der Dienst nicht mehr benötigt wird.

2.6 Standard-Geräteunterstützung

Ein großer Teil der Javaprogramme erwartet standardmäßig die Existenz von Hostingklassen für Netzwerkadapter, Grafikkarte, Maus und Tastatur. JavaOS muß also Schnittstellen anbieten, die diese unterstützen,

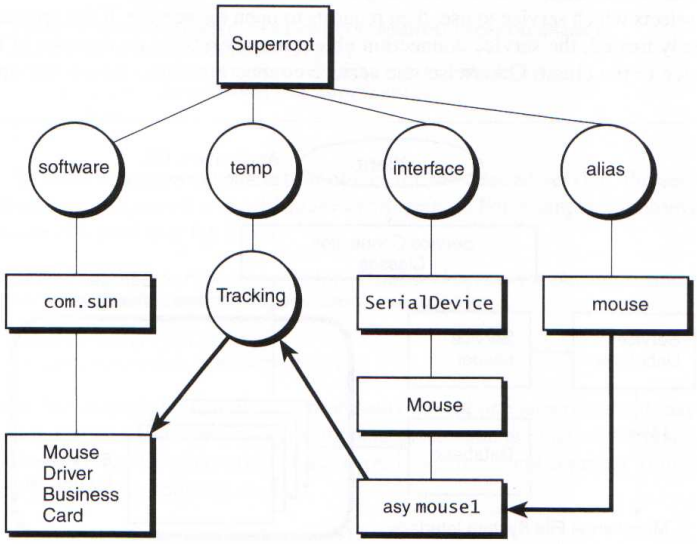

Abbildung 3: Ein Beispiel für einen Eintrag des JSL in der JSD

aber unter der Berücksichtigung, daß solche Standardkomponenten gegen Produkte eines anderes Herstellers austauschbar sind.

Für das Netzwerk wird durch eine Implementation von java.net die Standard-API unterstützt. Ein Protokollstapel, den die zu übertragenden Daten durchlaufen, verbindet die java.net Klassen mit dem Netwerkkartentreiber. Alle Komponeneten sind in Java geschrieben, also plattformunabhängig. Nur der Treiber beinhaltet eine Aufruf von nativem Code zur effizienten Berechnung von Checksummen. Da aber diese native Methode von JavaOS auf jeder Plattform zur Verfügung gestellt wird, ist der Treiber dennoch plattformunabhängig. Er ist aber sehrwohl abhängig vom Gerät und dem Bustyp (PCI, ISA, etc.) an den es angeschlossen ist.

Die Grafikunterstützung bietet die Standard-API durch die Implementation von java.awt und es stehen Windowing- und Grafikklassen zur Verfügung. Der Videogerätetreiber ist auch Javacode, aber aus Performancegründen wird sehr häufig auf native Videobeschleunigungsmethoden zurückgegriffen. Diese sind nur zum Teil hardwareunabhängig, zum Teil sind sie aber hardwarespezifisch optimiert. Dieser muß dann für die Verwendung auf mehreren Plattformen portiert werden.

Für Maus und Tastatur stehen ebenfalls die Standardschnittstellen des JDK zur Verfügung.

2.7 Gerätetreiber

Gerätetreiber sind ein Teil der Runtime Environment, also plattformunabhängig und greifen nicht direkt auf den Speicher und die Interrupts zu. Sattdessen verwenden sie die Speicher und Interrupt Objekte vom JavaOS Plattform Interface (JPI). Sehr wohl sind sie aber von den konkreten Geräten und den Bustypen, mit denen diese verbunden sind, abhängig. Treiber können zum einen statisch sein, d.h. sie werden beim Booten gestartet und können nicht beendet werden, aber auch dynamisch, was bedeutet sie werden erst bei Bedarf geladen.

Die Existenz eines Gerätetreibers beginnt mit der Entdeckung eines neuen Gerätes durch einen Busmanager. Dabei wird ein Eintrag in die JSD gemacht, der unter anderem ein Objekt beeinhaltet, das Speicher- und Interruptobjekt für die Kontrolle des Gerätes erstellen kann. Wie im Abschnitt über JSL erläutert wird dieser über einen Event darüber informiert, sucht den geeigneten Treiber und übergibt ihm eine Referenz des eben erwähnten Objekts. Mit dieser kann sich der Treiber Speicher- und Interruptobjekte zur Interaktion mit dem Gerät schaffen. Zusätzlich zum Gerätetreiber kann auch noch ein Gerätemanager existieren. Er faßt mehrere Geräte einer bestimmten Kategorie zusammen und bietet Methoden an, die alle seine Geräte unterstützen. Dies erleichtert das Schreiben von Treiber für die schon ein solcher Manager vorhanden ist.

Ebenfalls zur Entlastung der Programmmier dient das JavaOS Device Interface (JDI). Es beinhaltet Schnittstellen für das Erzeugen von Gerätetreibern und -manager sowie von Bustreibern und -manager. Ebenso stellt es Basisfunktionen, wie z.B. Powermanagment, zur Verfügung, die viele Geräte unabhängig von ihrer Kategorie brauchen. Außerdem bietet das JDI noch Methoden für dem Umgang mit bestimmten Datentypen, beispielsweise zeichenorientierte Daten, blockorientierte Daten oder Audiodaten. Es werden auch einige

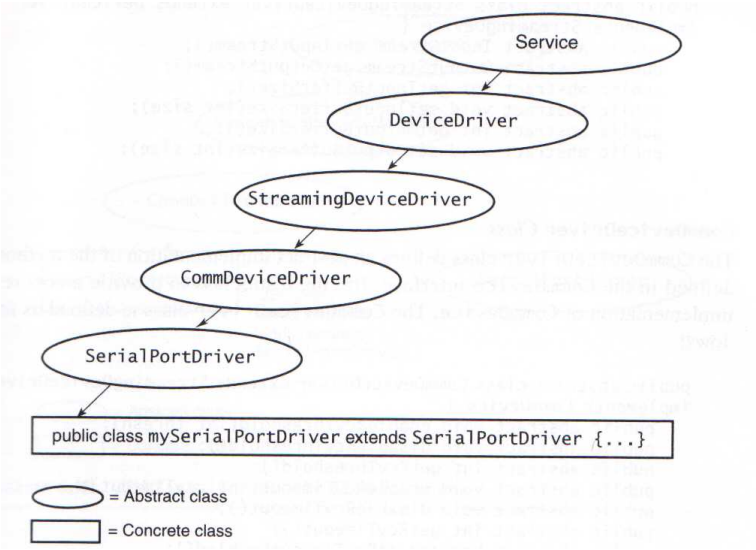

Abbildung 4: Ein Beispiel wie sich Treiber von Standardklassen der JDI ableitet lassen

Exceptions definiert, zum Beipiel die `DeviceOpNotSuppException`, die geworfen wird, wenn ein Treiber eine Operation nicht unterstützt.

2.8 Speicher

Einige Systemkomponenten wie z.B Treiber benötige low-level Zugriff auf I/O Adressen, Direct Memory Access (DMA) Bereiche, oder spezifische Adressen. Um diese trotzdem plattformunabhängig gestalten zu können, kapselt das JavaOS Plattform Interface (JPI), welches nicht mehr Teil der Java Runtime ist, Adressen und Adressbereiche in Objekte. Diese Objekte liefern auch native Methoden, die dann den Speicherzugriff ermöglichen. JavaOS stellt dabei zeitgleich einen physikalischen, einen DMA und beliebig viele virtuelle Adressbereiche zur Verfügung. Für jede JVM, d.h. für jede Applikation im System, ist ein eigener virtueller Adressbereich vorgesehen. Momentan wird jedoch nur eine Applikation gleichzeitig erlaubt und so laufen die Applikation und die Systemthreads im gleichen Adressraum. Die Speicherklassen prüfen bei jedem Zugriff auf die dafür nötige Berechtigung und der Kernel implementiert die konkrete Adressumsetzung, den Addressbereichsschutz und das Seitenmanagment.

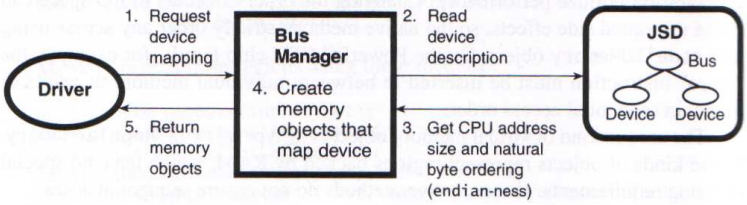

Abbildung 5: Ein Beispiel wie sich Treiber Speicherbereiche schaffen

2.9 Interrupts

Ebenso wie den Speicher abstrahiert das JPI auch die Interrupts. In der Systemdatenbank existiert dafür ein Teilbaum der die physikalische Interrupttopologie darstellt. In den Einträgen kann einem Interrupt ein *interrupt enabler*, ein *disabler*, ein *acknowledger* und ein *handler* zugewiesen werden. Dies sind Softwarekomponenten die einen Interrupt maskieren, demaskieren oder dem Interruptcontroller den Empfang des Interrupts signalisieren. Der Handler ist dann schließlich für die Behandlung des Interrupts zuständig. Dabei kann es jedoch wichtig sein, innerhalb einer gewissen Zeit zu reagieren, damit es nicht zu einem Datenverlust kommt. Java Code ist dafür nur bedingt geeignet, da der Garbage Collector diese Behandlung für einen ungewissen Zeitraum

unterbrechen könnte. Daher gibt es drei verschiedene Level von Interrupthandlern: First-level Handler sind hochpriorisiert und unterbrechen die Ausführung jeglicher Threads. Sie können einen second-level Handler bestimmen, dem der Scheduler im Anschluß die Ausführung ermöglicht. Second-level Handler werden als native Threads ausgeführt, verdrängen also Java Threads. Third-level Handler sind gewöhnliche Java Threads, haben alle Mittel von Java Software zur Verfügung, aber unter Umständen sehr hohe Latenzzeiten. Ist kein first-level Handler installiert, so wird der second-level, oder falls auch nicht existent, der third-level Handler in die Warteschlange des entsprechenden Schedulers eingefügt.

2.10 Microkernel

Der Microkernel ist die unterste Schicht in JavaOS. Die Virtual Machine (JVM) und das Plattform Interface (JPI) sind die einzigen Komponeneten die unmittelbaren Zugriff auf die Dienste des Kernels haben. Auch native Treiber und native Programme unterstützt er nicht direkt. Damit aber JVM und JPI nicht von einem bestimmten Microkernel abhängig sind, muß dieser das Kernel Service Interface (KSI) implementieren, durch das auf seine Dienste zugegriffen werden kann. Innerhalb des Kernels existiert noch ein Interface, das Platform Adaptor Interface (PAI), welches die plattformunabhänigen Teile des Kernels mit den plattformabhängigen verbindet. Beim Portieren muß also nur noch der Code unterhalb des PAI angepaßt werden. Die Dienste die der Kernel anbietet bestehen aus Paging, Threads, Interrupts, Monitoren, Exceptions und Zeitgebern.

Beim Ausführen von Code kann der Kernel die Sicherheitsmechanismen der Programmiersprache Java voll ausnutzen. Sie sind so weitreichend, daß die ganze Software im Supervisor Modus der CPU ausgeführt werden kann, ohne daß eine Gefahr für das System besteht. Damit entfällt der aufwendige Moduswechsel, der nötig ist, wenn Betriebsystem und Anwenderprogramme in verschiedenen Modi laufen. Ebenso wird auf getrennte virtuelle Adressräume für einzelne Programme verzichtet. Dadurch wird auch der Inter-Process Communication Service (IPC) und das damit verbundene Mapping und Unmapping von Daten zu und von verschiedenen Adressräumen überflüssig.

Der Kernel ist also wirklich so minimalistisch gestaltet, so daß er gerade die Ausführung einer Virtual Machine unterstützt, und die low-level Funktionen realisiert, die für Treiber zwingend nötig sind.

Literatur

[1] Tom Saulpaugh, Charles Mirho. Inside the JavaOS™ Operating System. 1999

[2] Peter W. Madany. A Standalone Java™ Environment. Mai 1996.
http://java.sun.com/docs/white/

[3] SUN and IBM. Ein Whitepaper über Treiber. 1998
http://www.geocities.com/diltiwari/solaris/java.html

[4] Jean Lainé. Java Operating System. 1999
http://taurus.ubishops.ca/jlaine/jos.html