JavaOS

David Burhans
2/3/2003
CS384
Dr. Taylor

Table of Contents

JavaOS 1
Table of Contents i
Table of Figures ii
Background 1
Java 1
Bytecode 2
JavaOS 2
Supported Computing Models 3
Non-Runtime Components 4
JavaOS Platform Interface (JPI) 4
Microkernel 5
Booter 7
Runtime Components 7
System Database 7
Event System 8
Service Loader 9
Device Drivers 10
Conclusion 12

Table of Figures

Figure 1: Flow from code to execution (Teach yourself Java)............................ 2
Figure 2: JPI Interaction 5
Figure 3: JSD Diagram (Mirho, 10).. 8
Figure 4: Event System distributing an event 9
Figure 5: Low-Level Device Communication (JavaOS for Business) 11
Figure 6: Device Manager Handling I/O 12

Background

Java

In 1991, Sun began development of a programming language for internal use. They had many different machines and needed a language that could be used on all of them. Thus was the birth of Oak. Oak took the developers at Sun eighteen months to complete. It was decided to change the language's name from Oak to Java because there was already a language name Oak.

Java is a high-level programming language released in May of 1995 by Sun Microsystems. When released, it sent ripples through the development world. The reason for the splash -- applications written in Java are able to run on any platform that has a Java Virtual Machine (JVM) Interpreter on it. This means that a program written in a Windows environment could be executed on a Unix platform with no need to modify the source code or even recompile. It is able to do this because when Java programs are compiled, rather than the compiler translating the program into machine code, it translates it into bytecode. ("Teach yourself Java")

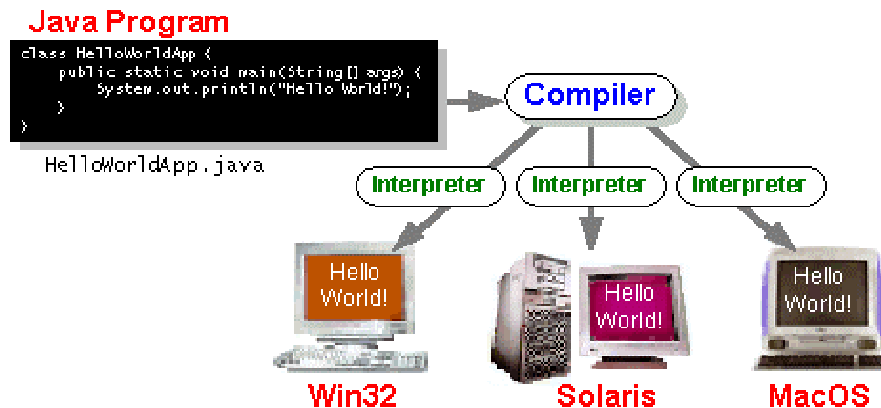

Figure 1: Flow from code to execution (Teach yourself Java)

Bytecode

Bytecode is what enables Java applications to be cross-platform. If one were to look at the bytecode generated by a Java compiler, they might mistake it for machine code. In actuality, it is this code that the JVM interprets into machine code for the operating system to execute. This results in very small files, which increases portability. ("Bytecode Basics")

JavaOS

JavaOS was created to enable many different types of computers to run in the same environment -- same operating system (OS), same programs, same configurations. This would greatly decrease the amount of support required for a technologically diverse company. This would create a hybrid working environment and testing can be done on many different platforms with relative ease. This does not come without a price.

The JavaOS uses the same JVM that the Java programming language uses. The additional interpretation layer causes the operating system to run slower than its competitors.

Also included in the operating system is a garbage collector. This reclaims unused system resources. The garbage periodically polls the system and looks for obsolete data and frees up the memory that it was using.

Another component of the OS is the class loader. This component loads and prepares programs for execution on the virtual machine. Without this component the bytecode would not be sent to the interpreter for translation.

Supported Computing Models

The models that JavaOS supports are:

- Operating system cached on the client, executed on the client
- Operating system stored on the server, executed on the client
- Data files cached on the client
- Data files stored on the server
- Applications cached on the client, executed on the client
- Applications stored on the server, executed on the server

These lend themselves particularly well to a distributed computing environment, as both applications and operating systems can be run from remote computers. This is advantageous because it allows for network booting and storing everything on a central server. Storing all operating system and

program information at a centralized location allows for easier upgrading and maintenance of the entire network.

This centralization has a downside; it greatly increases network traffic and is also much slower than local execution. As a result of the possible lower accessibility and speed, productivity may decrease.

It also creates a heavy reliance on network connectivity. If the server is loses connection from the network, it would make it impossible for the clients to boot or access any required data. (Mirho, 6)

Non-Runtime Components

JavaOS Platform Interface (JPI)

Simply speaking, the JPI is the liaison between the microkernel and the JavaOS Runtime Environment (JRE). The JPI facilitates the replacement of the microkernel layer. This is very important because from here is where much of the platform independence comes.

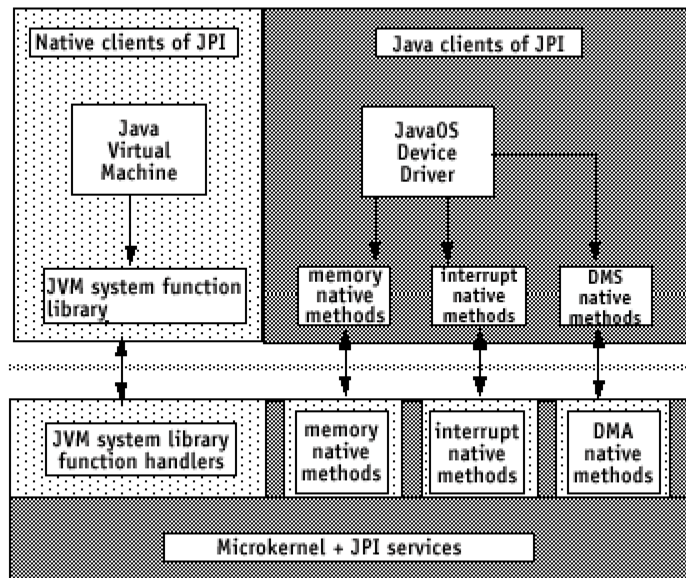

Figure 2: JPI Interaction

Note: A different JPI must be made for every type of microkernel since it uses commands specific to the microkernel.

Microkernel

The kernel is the meat of the operating system. It is responsible for paging, threads, interrupts, monitors, exceptions, timers, memory and more. Much of this is handled by the JPI, but there are some things that only the microkernel can control.

The microkernel only communicates with two other components, the JVM and the JPI's native methods. With so few clients to worry about, the kernel can be more efficient and easily ported.

The JavaOS adheres to three principles.

- Run all software in supervisor mode.

Most operating systems run software in user mode, but when the microkernel requests a service, an expensive switch to supervisor mode must be made. This is avoided in JavaOS because the microkernel is always in supervisor mode. Because JavaOS follows the same standards as Java, the safety features built into the language make in impossible to crash the system.

- Run all software in a single virtual address space.

  This reduces the need for constant swapping of virtual address spaces. This is not too costly when it is happening, but it is very expensive in terms of the amount of code required to do this. Most operating systems use this for security reasons, a program can very easily crash a system if it tries to access memory it does not own. This is inherently prevented in JavaOS.

- No inter-process communication service is necessary.

  There is no reason to have this with a single virtual address space. This is usually the bridge through which programs in different address spaces communicate.

Memory management is another very important duty of the microkernel. The objective of JavaOS' memory management function is to remove the hardware details from it. It does this by providing a common interface for the allocation and deallocation of memory.

The Virtual Memory Manager (VMM) depends on a platform adaptor to convert the virtual memory addresses into the physical addresses. The platform adaptor also handles page faults. (“Java 1.2 Unleashed”)

Booter

The booter is the lowest level of JavaOS. The booter’s job is to load the operating system into executable memory and activate the microkernel. The JavaOS Boot Interface (JBI) provides a standard interface that enables JavaOS to boot from virtually any device.

The JBI is a bidirectional communication interface between the booter and the microkernel. First the booter starts JavaOS by invoking its entry point. The microkernel in turn gets system specific information from the booter. The booter then terminates itself and is removed from memory. (“JavaOS: A Standalone Environment”)

Runtime Components

System Database

The JavaOS System Database (JSD) contains all of the configuration information for the OS. Things entered into this database are called entries. Possible entries include users, devices, services, etc.

The JSD stores two types of information: transient and persistent. The persistent information is cached on the local hard drive for future retrieval.

Transient information could possibly change often and is typically retrieved during the boot sequence. Depending on the information, it may need to be updated on a regular basis. (Mirho, 10)

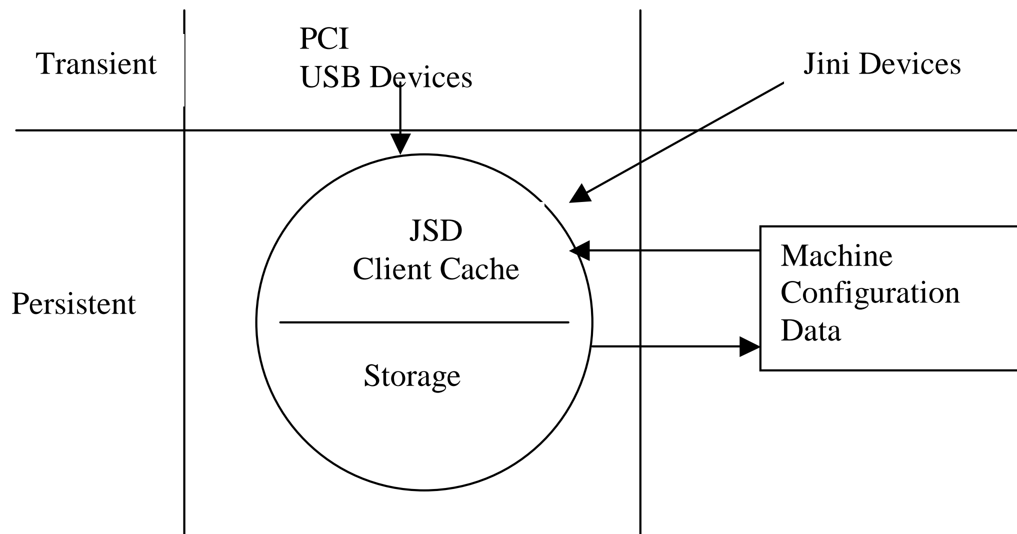

Figure 3: JSD Diagram (Mirho, 10)

Event System

The event system controls the routing of events in JavaOS. An event contains information of something that happened in the OS. A description is contained in the event so the OS knows what to do with it. The thing from which the event originated is called the producer. The producer's event is then sent to consumer(s). Consumers will perform an action when the event is received, be it saving data or logging off the current user.

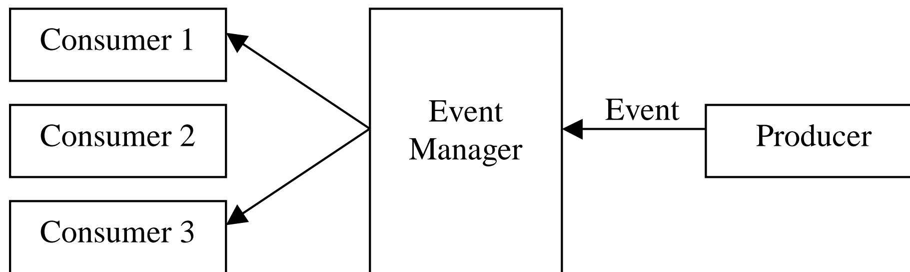

Figure 4: Event System distributing an event

Because of the interface the Event System uses, a database of valid producers and consumers must be maintained. Both of these components must register with the event manager. Registering consists of letting the manager know it exists and for which kind of events it is looking. This manager is sometimes called a broker because it handles all transactions in the OS. (Mirho, 11)

Service Loader

A service is a collection of java language packages that tell the JSD where they are, how they can be loaded, and what they can do. Typical services include device drivers, file systems, and printer communications. The JavaOS Service Loader (JSL) loads and unloads services in response to events given to it by the event system.

There are a few ways the JSL can load services. The first is it can load on boot. The booter reads the Master Configuration File (MCF) from the disk to obtain the list of services it should load. After the OS has booted services can be loaded in other ways.

A service can also *load on discovery*. That happens when a new service is inserted into the JSD. This new service will then be available to all clients using the same configuration file as the server computer.

Another type of loading occurs only with device drivers. This happens when JavaOS detects new hardware and needs to load drivers for it. This is called *load on matching* because it loads a driver that matches the hardware.

The last way the JSL can load a service is to have it requested. This happens when an application runs and needs a service not currently loaded. The JSL will load the appropriate service for the applications use. When the application is finished with the service, the JSL will unload it and the garbage collect reclaims any resources that are no longer in use. (“JavaOS: A Standalone Environment”)

Device Drivers

Device drivers are used for the implementation of input and output operations. They communicate using the JavaOS Device Interface and the event system. A device driver formats the data of a device into something the operating system can understand. They are a software abstraction of the hardware and therefore can make it appear that all hardware has essentially the same interface.

Drivers for JavaOS are written in Java and are platform independent. Even though the drivers are platform independent, they are specific to a certain device and bus combination. ("JavaOS for Business")

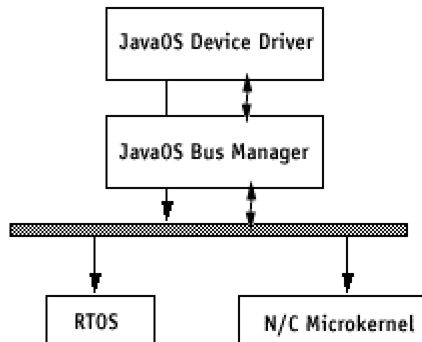

Figure 5: Low-Level Device Communication (JavaOS for Business)

The JavaOS also supports virtual devices. A virtual device is a piece of software that emulates hardware. The software is able to take residence in the CPU address bus and behave like any other hardware except it does not physically exist outside of the computer.

JavaOS uses what is called a "shallow driver" architecture. In a shallow architecture there are only two layers of abstraction over the hardware. The lowest layer is the actual device driver. This communicates directly to the device. Above the drivers is a device manager. The device manager acts in much the same way as the JSL. An application asks for a certain type of input or output and the device manager decides which device driver to use to achieve the desired result. (Mirho, 91)

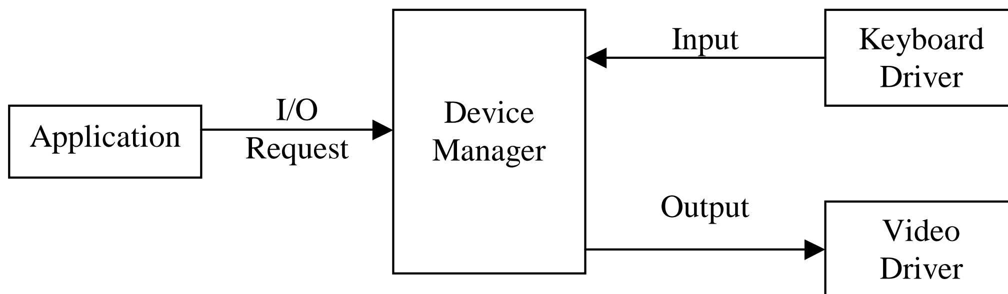

Figure 6: Device Manager Handling I/O

Conclusion

JavaOS brings the benefits from Java into a standalone operating system. The advantages of have a platform-independent OS are great, especially when distributed computing is involved. The added type checking and security of Java also allow the OS to save in many places where other systems would spend large amounts of resources. On the downside, it is still an interpreted language, so it is slower than most other operating systems. Also, it does not have the support to become a major player in the OS industry anytime soon.

References

"Bytecode Basics." 1996. [Internet, WWW] ADDRESS: http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html [Accessed: Feb 2003]

"Java 1.2 Unleashed." 1998 [Internet, WWW] ADDRESS: http://www.szptt.net.cn/9810dnwl/new/jdk1.2/ch52/ch52.htm [Accessed: Feb 2003]

"JavaOS: A Standalone Java Environment." Technical White Paper. 2001. [Internet, WWW] ADDRESS: ftp://ftp.javasoft.com/docs/papers/JavaOS.cover.ps [Accessed: Jan 2003]

"JavaOS for Business: Technical Summary." 1999 [Internet, WWW] ADDRESS: http://www.engr.uconn.edu/~steve/Cse258/JavaOS/overview.html [Accessed: Feb 2003]

Lindholm, Tim. Yellin, Frank. *The Java Virtual Machine Specification 2nd Edition.* Addison-Wesley. Copyright 1999, Sun Microsystems, Inc.

Poulson, Lisa. May 29, 1996. "JavaSoft Announces JavaOS." [Internet, WWW] ADDRESS: http://www.sun.com/smi/Press/sunflash/9605/sunflash.960529.11819.html [Accessed: Feb 2003]

Saulpaugh, Tim. Mirho, Charles. *Inside the JavaOS Operating System*. Addison-Wesley. Copyright 1999

Silberschatz. Galvin. Gagne. "Operating System Concepts." 6th Edition. John Wiley & Sons, Inc. New York. 2002.

Sun Microsystems, Inc. 1995-2002. "java.sun.com: The Source for Java™ Technology." [Internet, WWW]. ADDRESS: http://java.sun.com/ [Accessed: Jan 2003].

"Teach Yourself Java." 2002. [Internet, WWW] ADDRESS: http://www.duke.edu/~trc7/cps/background/what_is_java.html [Accessed: Feb 2003]