JIKESNODE: A JAVA OPERATING SYSTEM

A thesis submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Science and Engineering

2005

By
Georgios I. Gousios
Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

Operating system kernel development has been an active area of research since almost the birth of computer science. There are currently two major architectural designs for kernels, namely monolithic and microkernels. This thesis examines the potential of a Java operating system that theoretically combines the strong points of the aforementioned designs. The proposed architecture merges the Jikes Research Virtual Machine with the JNode operating system in order to demonstrate the feasibility of such an approach and to provide the Jamaica project with a tool to further continue the study of parallelism.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

First of all, I would like to thank Ian Rogers for his invaluable help throughout the project. Ian gave my project the initial kickstart, promptly gave up what he was up to when I asked for help and acted as an intermediate between me and my supervisor whenever my language skills were not up to the level to be able to explain what I was doing. Finally, he was something like the group's mother by preparing and sharing coffee, opening the windows and/or switching on the air-conditioning when it was hot (well, it never got really hot, but anyway), and buying us doughnuts from time to time.

Also, many thanks should go to my supervisor, Chris Kirkham for the motivation to work and moral support he provided me, especially at the beginning of the project when the project turned out to be much tougher than I have expected.

Thanks to the Greek guys from the ACS course for keeping me sane during the project, to Andrew Dinn for the two essential 2-hour chats, to Ian Watson for obtaining a licence for VMWare in just 1 day and to the Greek national football team for cheering us up and giving us the chance to get revenge from our English friends.

The following people have contributed comments and ideas on how to improve the present document and have gone through the tedious process of proofreading: Chris Kirkham, Ian Rogers, Diomidis Spinellis. I would like to thank them for their time and effort.

Last, but of course not least, to Fenia Aivaloglou for keeping me company, and for being the closest person I had in my whole year in Manchester.

Chapter 1

Introduction

This report, by its very length, defends itself against the risk of being read.

— Winston Churchill

In this thesis, I present the implementation of the JikesNode operating system, an attempt to create an operating system using the Java programming language. The current implementation relies heavily on the Jikes Research Virtual Machine (JikesRVM) Java Virtual Machine (JVM) [2, 26] and the JNode operating system [17]. The work was carried out in the Jamaica group at the University of Manchester.

1.1 Motivation and objectives

The Jamaica group at the University of Manchester has long worked on chip multiprocessor research. Chip multiprocessors support hardware level parallelism, thus allowing the execution of multiple threads simultaneously by utilising several processing cores on a single die. In order to be able to test and further develop the design and the implementation of the Jamaica chip, the group has put a significant amount of effort into porting the JikesRVM to their chip simulator and also on trying to improve the core of JikesRVM itself.

With the basic software infrastructure already implemented, what was really needed were applications that would expose the chip design to realistic workloads. Operating systems, unlike benchmarks, show a high degree of unpredictabilty in their operation which could set forth design or implementation flaws in the Jamaica platform. Also, an operating system could be the base of running massively

multithread applications which in turn could be a proof of concept for the platform. The JNode operating system presented an ideal platform because it was written in a high level thread-aware language and also included a functioning driver model and an Input/Output (I/O) subsystem. The challenge presented was to separate the core Operating System (OS) from its proprietary virtual machine and port it to the JikesRVM.

Since the port of JikesRVM to the Jamaica simulator was at an early stage of development and the simulator itself was slow enough to not allow for fast compile-and-test cycles, the decision was made to base the project on the i386 architecture. The goals that were set at the beginning of the project were the following:

- Create a standalone environment that offers to JikesRVM just the necessary functions to be able to work without an operating system.
- Separate the core of the JNode operating system from its Virtual Machine (VM) and port it to the standalone version of the JikesRVM.
- Create a build system that integrates the `make`-based build system of the GNU is Not Unix (GNU) classpath, the `ant`-based build system of JNode and the custom made build system of JikesRVM.

1.2 Related work

Operating system research is almost as old as computing itself. There are two major architectures that have emerged: monolithic kernels and microkernels. The monolithic kernel was the first architecture to be put into practice and still is the basis of the majority of the operating systems. Examples include Linux [6], the Berkeley Software Distribution (BSD) family [24] and the Solaris operating systems. Microkernels were supposed to be the next big thing in operating systems, but unfortunately due to early performance problems they never really gained the required momentum. Many systems were developed, the most important of which are Mach [1] by the Carnegie-Mellon University and SPIN [4]. Later efforts [9, 21] managed to overcome the infancy problems of the first implementations.

Work on Java operating systems started by the creators of the Java language, Sun Microsystems. From the scarce documentation available, the JavaOS [28] appears to provide a single network-enabled virtual machine with the ability to

display graphics on special framebuffers. The TOS [25] operating system, developed in Brock University, Canada, was an educational operating system along the lines of Minix [30]; it provided a set of servers for common operating system tasks but it required a JVM running on a host OS to run.

The KaffeOS [3] and the Alta [32] systems, both developed at University of Utah, share the assumption that a Java OS must support the process abstraction. A process is presented with a virtual instance of the JVM; the underlying GNU acts as a resource manager and is responsible for initiating and scheduling processes and providing InterProcess Communication (IPC) mechanisms. In KaffeOS, each process has its own Java heap and its own Garbage Collection (GC). The notion of a process is built on top of classloaders, each process having its own classloader for objects that are not shared. Objects that could be shared are loaded by the system wide classloader. In Alta, each process is offered the illusion of a complete copy of the GNU having separate root thread groups and private copies of static member data . Both systems are built on top of the KaffeVM and are not directly bootable.

The JX operating system [11] developed at the University of Erlangen, Germany is a real Java operating system in the sense of it can be directly booted on a machine and provides device drivers. The JX system includes a custom-developed lightweight VM, a classpath implementation and the analogue to a process model, called a domain. A domain is an instance of the virtual machine that runs independently but can communicate with other domains with a custom Remote Method Invocation (RMI) like mechanism and share read-only instances of classes. This kind of architecture offers application protection and isolation while still maintaining good performance, according to the JX developers. The results presented are not terribly convincing though, because they are not produced using any kind of globally accepted benchmarking method and only the strong points of the system are presented.

Finally, the JNode OS, written from scratch by Ewout Pransgma and released under the General Public Licence (GPL), is a Java operating system that aims to run all Java applications natively. It includes a custom resource-aware JVM implementation, a nanokernel that provides a thin abstraction layer between the JVM and the hardware and the GNU classpath implementation. Everything in JNode, from device drivers to applications are plugins to the system managed by a central plugin manager. More details on the JNode architecture can be found

in section 5.1.

Considering the options available, the decision was made to base this work on the JNode operating system for the following reasons:

- The project is relatively new, thus its size is not overwhelming.
- It is designed and developed using modern software engineering practices.
- Although JNode includes its own JVM it is not too tightly integrated with it, so porting it to JikesRVM would not be a resource consuming exercise.
- Its source code was available under the GPL licence.

1.3 Organisation of the thesis

Chapter 2 is about kernel architectures. The basic architectures are described and their strong points and weakness are detailed. Later, we describe the general idea and the architecture of a Java operating system.

Chapter 3 describes the bits of the i386 architecture needed to understand the nanokernel implementation and also discusses the development of the nanokernel itself.

Chapter 4 describes, without going into detail, the JikesRVM architecture and the changes made to the JikesRVM subsystems in order to make it work on top of the nanokernel.

Chapter 5 discusses the processes of bringing together the three distinct system components (nanokernel, JikesRVM and JNode).

Chapter 2

Operating system architectures

> *Operating systems are like underwear – nobody really wants to look at them.*
>
> — Bill Joy

Any computer system includes a basic set of programs called the operating system. The most important part of an operating system is its *kernel*. A kernel, in traditional operating-system terminology, is a small nucleus of software that provides only the minimal facilities necessary for implementing additional operating system services [24, Chapter 2.1]. The kernel is the first component of the operating system to be loaded in memory when the computer starts and stays there until the computer is shut down. The kernel is mainly responsible for communicating with the hardware and providing a basic set of services in order to allow user programs to run. A more complete list of responsibilities for a kernel can be found in [30, Chapter 1].

2.1 Established architectures

There are two major kernel architectures: *monolithic* kernels and *microkernels*. Monolithic kernels form the basis for almost all Unix-like operating systems in use today. Examples include Linux, the BSD family and Solaris. There were some efforts for commercial microkernel operating systems, mainly in the form of OSF/1 and NextStep, but the fact remains that microkernels are mostly used for research purposes. On the other hand, the two most widely used operating systems today, WindowsNT and MacOSX, take a hybrid approach by either running a large portion of a monolithic kernel as a kernel task (MacOSX) or multiple

processes in kernel space (WindowsNT), both on top of a microkernel.

2.1.1 Monolithic kernels

Monolithic systems, the most common architecture for kernels, are those whose functionality is contained into a single executable lump of code that runs in executive/kernel mode. All operating system services run in kernel mode and can be invoked by user space processes via interrupt-driven system calls. Although the kernel might be designed as a sum of components and layers, in practice even high level components can directly access low level functions.

As an example of tight integration of subsystems in monolithic kernels, the Linux kernel scheduler provides the `set_cpus_allowed` function to restrict processes to particular CPUs. The Athlon64 Powernow driver[1] directly uses this function to make a certain physical processor run the kernel execution context that the driver is currently in, a required step prior to deciding the power saving capabilities of the processor. A non-monolithic system would require many more interactions to achieve the same result, such as requesting the processor list, locking each one of them in turn to decide if the driver runs on the specific processor and finally triggering a reschedule after finishing.

Some of the strong points the monolithic design offers are the following:

- Efficiency: System calls in monolithic systems are implemented as regular function calls using commonplace calling conventions (e.g. the stack and registers). There are only two context switches taking place for servicing a system call.

- Optimisations: High level components can bypass intermediate layers and directly interoperate with low level functions. The example presented above is quite usual practice in open source kernels. Although from a software engineering point of view bypassing layers in a layered system is not acceptable, in practise it may be the only way to get acceptable performance or add features not present in the system's first design.

- Dynamicaly Loaded Extensions: Most monolithic kernels today can be extended at runtime using dynamically linked programs called modules. In order to achieve that, kernels provide a linking mechanism that imports

[1] linux/arch/i386/kernel/cpu/cpufreq/powernow-k8.c:415–430

the module's symbols and code into the kernel space and a corresponding module interface (e.g. the Linux module mechanism [6, Appendix B]). After being linked, the module can access the kernel structures as if it was directly compiled into the kernel. Modules allow for layered system design, platform independence (e.g. the SCSI disk driver is exactly the same for all architectures in Linux) and efficient memory usage which is essential for embedded systems, while having almost no effect on system performance.

The monolithic design's achilles heel is security and stability. A badly written kernel module can lead to security holes that can be used to compromise the whole system. Also, in case a kernel component misbehaves, for example if is crashes, the operating system will collapse. Being many years in development has minimised those problems for the major operating systems, though.

2.1.2 Microkernels

The basic idea driving microkernel development is to reduce the kernel space provided services to the absolute minimum and implement the rest of the OS services as a set of user space programs, usually referred to as servers. Microkernels, as their name might suggest, offer only a small subject of the services that a normal kernel does; in fact, many microkernels only provide IPC, trap handling and basic scheduling services. The advantages of the microkernel design are clear from a software engineering standpoint:

- Modularity: Almost all the system components can be replaced with new implementations while the system is running. The kernel can provide application interfaces to many established standards even if these standards' hardware requirements overlap without the cost of emulation on top of the operating system. An example is the Windows implementations of both the WIN32 and the Portable Operating System Interface for uniX (POSIX) environments using dynamically loaded servers.

- Stability: The various kernel processes run in isolated memory areas. A crash in, for example, the network card driver cannot affect the filesystem operation.

- Portability: In theory, the only part of a microkernel OS that needs to be ported on a different architecture is the microkernel itself. The portability

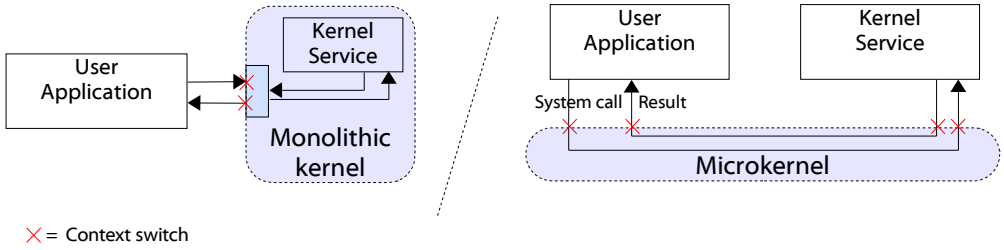

Figure 2.1: Monolithic kernel vs microkernel in system service call handling

work mainly consists of porting the kernel and the memory management servers. While this statement is also in part true for the monolithic systems, the clean separation of system parts and the existence of well defined interfaces significantly reduce the amount of effort required in the case of microkernels.

The first microkernel implementations were received with enthusiasm by the research community. After the microkernels evolved into real operating systems, the enthusiasm gave its place to scepticism, the cause being performance. In order to service an IPC call, be it a system call or a user space process to process call, the design of a microkernel OS does not allow it to employ traditional IPC mechanisms, such as shared memory or stack-based function calls. The cost of the associated Remote Procedure Call (RPC) seemed to hurt IPC performance in a microkernel environment. The cost should not only be attributed to context switching; there are a lot of performance issues discovered by [19, 7, 21], described netxt.

First of all, the cost of RPC for servicing systems calls is significant by itself. As shown in figure 2.1, in order for an RPC to be served, four process switches must be made compared to two in the case of monolithic kenrels. A process switch involves, among others, saving processor registers, invalidating caches and possibly rescheduling. The cost of process switching may differ from architecture to architecture, but it is at least twice in microkernels compared to traditional kernels [19].

Also, the memory accessing cost seems to be much higher in microkernel systems. The cost, measured in Memory Cycles Per Instruction (MCPI), involves the invalidation of caches and accesses to memory while servicing system calls. The fact that it is higher in microkernels is a consequence of intermodule copying during RPC. Furthermore, frequent context switches lead to worse locality properties

than monolithic kernels [7].

The fact that microkernels have evolved from traditional designs and had to support existing software poses an additional performance penalty [21]. The Mach microkernel offered more than 150 system calls, roughly the amount of system calls provided by monolithic kernels of its time. The additional complexity of decoding the call in kernel mode and also redirecting it to user space is a limiting factor for performance. Other microkernel designs (e.g. the Exokernel [9]) that relied more on direct IPC communications between user processes and kernel servers performed significantly better.

For the reasons listed above, microkernels never really left the researcher's laboratories to get used into production environments. Although the second generation of microkernels, such as the Exokernel mentioned above and the L4 microkernel [20], offer comparable performance to monolithic kernels on certain workloads, the fact remains that the abstraction rule enforced by microkernels render the possible synergies that could be developed between kernel parts that know each others' operation almost non-existent. Even people who have fought in favour of microkernels [29] are now much more open-minded [30, Chapter 10.1.7].

2.2 The JavaOS

Throughout many years of operating system development there has been little change on the way computer scientists understand the operating system, the common conception being that the operating system is a service provider. The last change was when computers stopped being regarded as batch systems and multiprogramming and multiuser operating systems took over.

Java operating systems (JavaOS) are going to be a major change in the way computing is done.[2] For the first time a service provider on whom programs are going to depend on in order to function is not strictly necessary. Services and programs will be all part of a system, communicating with each other with simple function calls. Resource management will be inherent part of the system. Many security headaches will simply disappear using the type safety and bounds checking features of the core language. The executed code will be able to be dynamically optimised according to load characteristics of the system and the underlying architecture. Legacy code will be able to be executed through dynamic

[2] If they manage to prevail, of course...

binary translation, even with on the fly optimisations, as shown by work done in the Jamaica group [23].

2.2.1 Basic architecture

As with all operating systems, the primary role of a JavaOS is to manage resources in such a way so as to allow more than one program to run concurrently. As resources we define the set of computational power, memory space and I/O bandwidth that running programs compete for. Current JavaOS research, as presented in Section 1.2, focuses on two approaches to organising resources:

Process based: All executed programs run on the same JVM. The virtual machine builds the notion of a process by using seperate classloaders for each loaded program. Each program is tricked to believe it is running in its own virtual machine by being with private copies of bootstrap classes and its private memory space and garbage collector. Changes in the JVM are required to support this model.

JVM based: Each executed program consists of one or more Java threads. A supervising thread is responsible for spawning new programs as needed using a well defined interface. Language-based memory protection and thread synchronisation mechanisms are used to protect shared resources. Loaded programs become part of the system's classpath and so other programs can directly access their methods, but namespaces and method access control can be used to protect the program internals. No changes are required to the JVM.

The process-based approach is based upon the traditional operating system architecture. Figure 2.2 shows an overview of the process-based JavaOS . The JVM has been modified to include additional components:

- the resource manager is responsible to allocate CPU time, memory and bandwidth to processes
- The scheduler is responsible to start, stop and pre-empt processes
- the process accounting component monitors the runtime behaviour of processes and gathers resource usage data

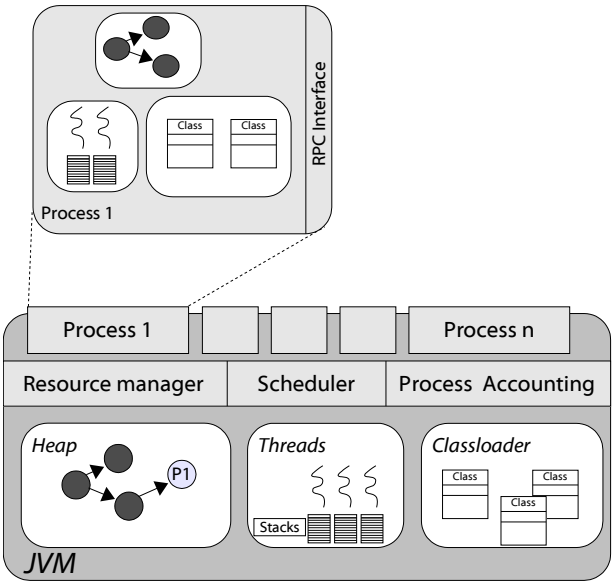

Figure 2.2: The process-based JavaOS architecture

The GC process can take place in both the JVM global heap and the process space heap. Depending on the implementation, IPC can be done using explicit object sharing [32], thread sharing between processes [32, 3] and local RPC invocations using the callee's interface [11].

A critique of the process-based JavaOS model

The process based JavaOS architecture adds an extra overhead to the virtual machine. Does it offer any added value to justify the oveehead? In our opinion it does not. The process abstraction was conceived when operating systems had no means of protection between concurrently executing programs. A program, at the time often written in a low level language such as assembly, could directly access all system available memory. There was no language level support for information hiding. The Java programming language offers both memory protection between threads, by completely eliminating pointers and enforcing bounds checking on array access, and disallows access to fields or methods private to the called class. Memory resource management is an inherent part of the JVM and for most JVMs it can be tuned for either performance or space usage. If a thread misbehaves with respect to memory, an `OutOfMemoryException` can be thrown by the JVM to stop it.[3] The problem of CPU resource sharing can be directly solved using

[3]Although the `OutOfMemoryException` is never caught in application programming and causes the JVM to exit, in the context of operating systems can be used e.g. by the supervisor

advanced thread scheduling algorithms in the JVM, a feasible approach since the JVM specification [22] defines no default scheduling strategy.

2.2.2 System components

Another role of an operating system, apart from sharing resources, is to interact with the hardware. Some types of interactions are described in the following list:

- Accept and process processor generated signals such as traps, faults and errors.
- Receive hardware interrupts.
- Provide a unified software view of classes of hardware devices for programs to use.
- Initialise the system during the boot process.

In the case of a JavaOS , the JVM is the component responsible to interact with the hardware. Since the JVM expects an operating system to do the hardware interaction in its place, a compatibility layer between the JVM and the hardware must be built. This layer must do little more than initialising the hardware, setting up a context switching environment for servicing interrupts and branching into the JVM. Device drivers would better be kept out of the compatibility layer and be directly implemented in Java, because this will make them portable and enable them to use the classpath facilities.

The basic architectural components of a JavaOS can be seen in figure 2.3[4]. The nanokernel represents the intermediate layer discussed above. When the computer boots, the nanokernel configures the hardware protection levels (usually kernel mode and user mode) and installs those parts of the interrupt handlers that execute in kernel mode. Depending on the hardware platform, the nanokernel can also be responsible for I/O operations, for example by providing a call that allows the program to break into kernel mode to do port-based I/O. The final state of its execution is to switch the hardware to user mode and branch into the JVM boot method.

thread to stop a resource consuming child.

[4]Original idea `http://jnode.sourceforge.net/portal/book/view/175`

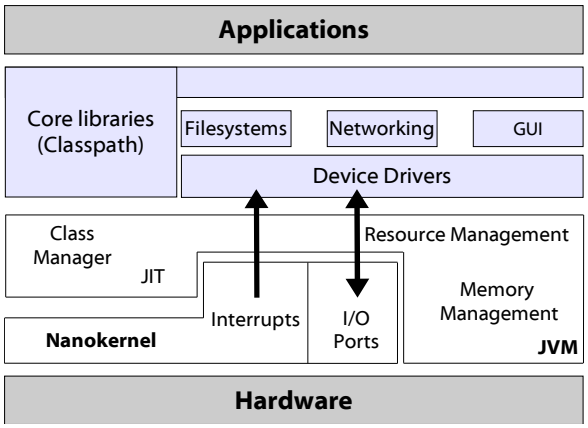

Figure 2.3: JavaOS components.

In the context of a JavaOS , a JVM is responsible for more than just executing bytecodes. First of all, it must be extended to include resource management capabilities. As resources, apart from CPU time, I/O bandwidth and memory space, we also consider interrupts and memory areas reserved for I/O. Basic resource management was discussed in section 2.2.1; hardware resource management can be implemented using a subscription/notification scheme in cooperation with the JVM's memory management subsystem. Memory management is a simple process in a JavaOS ; most JVMs just require a single non-segmented address space to use as heap, which is the state of the free computer memory after booting the OS. In order to implement a more advanced memory management scheme, such as virtual memory, support from the nanokernel, is required.

Device drivers allow an operating system to utilise hardware. In order for a device driver to talk to hardware, it needs to register for an interrupt to get notified when something interesting happens to the device and a set of memory ranges or I/O ports to read and write data to it. The interrupt registration and notification facilities are provided by the JVM along with the reservation of I/O ports. The nanokernel interrupt handlers are responsible to propagate interrupts to the JVM. The device driver layer should also support a hardware enumeration and detection facility, which maintains a database of supported hardware and assigns drivers to hardware. Device drivers in a JavaOS run in user mode and can use the Java libraries. Also, they can implicitly use the multithreading facilities provided by JVM, which in turn can have a positive impact on performance on a multiprocessor system and can help avoid pauses caused by I/O.

The classpath is the core library of the Java language. It provides programs with a set of classes ranging from simple data structures to complex graphics. In order to provide acceptable speed, most classpath implementations, including the GNU classpath, use native methods for the advanced components, through the Java Native Interface (JNI). In the case of the JavaOS , the classpath has to be modified to use the OS's subsystems instead of native calls. Finally, although the size of the classpath seems too big to be used as the basic JavaOS library, the classpath is the only library most Java applications will ever need to run. Also, the size is comparable with a set of native libraries that offer the same capabilities. [5]

2.2.3 Non-functional requirements

What we present below are high level non-functional requirements common to operating systems and how a JavaOS implementation addresses them.

Security

Security is a strong point for Java implementations in general. A JavaOS benefits from language-wide security features to provide a good set of security features.

Bytecode inspection The JVM includes a code verifier that can check if loaded bytecodes conform to certain specifications. The verification process includes stack overflow,opcode and method argument type checks. The great majority of attacks is then discovered and isolated even before the offending component starts to execute. Java goes even further and allows the development of trust relationships between the bytecode supplier and the bytecode user through digital signatures. Important system components, such as device drivers and system servers, can then be trusted, a significant prerequisite in security aware sites. Finally, Java requires that the types of variables being passed to methods conform to the method declaration which can help with pointer indirection attacks [8].

Runtime security Apart from link time code inspection, Java offers runtime security checks. Array accesses are checked against array bounds and additionaly there is no type of variable which can directly access memory. In the context

[5] On a Linux system, the size of the GNU classpath is 16MB. The size of an equivalent set of native libraries (libc, Xlib, Qt, XML,) is almost double (30MB).

of the JavaOS , this behaviour helps protect against accidental or intentional access and modification of kernel structures. In addition, a JavaOS can use the Java language security infastructure to implement security domains and access policies for those domains, a feature only recently added to Linux. For example, it could implement seperate domains for network and system servers and thus isolate the core system from a potential network security breach.

Efficiency

Core language features will allow a JavaOS implementation to be efficient. Although efficiency cannot be judged without having actual implementations and current benchmark results, albeit promising, are debatable [11], two characteristics of JavaOS will allow it to be very efficient, given a good overall architectural design. These are the lack of system calls and lightweight locking. System calls are not needed in JavaOS because there is no need for the system to switch to kernel mode to service a request. Kernel mode was traditionaly used to restrict access to kernel internals from malicious code; as noted before, in Java the same functionality can be achieved through protection domains. Code with the appropriate permisions can still access low-level system functionality though. On the other hand, Java's lock monitoring functionality can be used by the system to provide fine grained locking of critical code paths. The JavaOS does not need to globally lock the system when an important event, like an interrupt, occurs. Traditional monolithic systems, e.g. FreeBSD/DragonFlyBSD, are trying hard to overcome the burden of the "giant kernel lock", as they name it.

Scalability

As scalability, we define the ability of a system to adapt to its executing environment. Because the JavaOS runs in a virtual machine and knows little, if any, about the executing harware, scalability of JavaOS depends on the scalability of the JVM. Most current implementations of multiprocessor capable JVMs use `pthreads` to model an abstraction layer to physical processors. There is no JVM implementation that can directly take advantage of multiple processors without relying on services provided by the underlying operating system (such as `pthreads`). Building a multiprocessing capable JavaOS would require significant amount of work on the nanokernel and the JVM, but the operating system itself would be able to directly take advantage of the multiple processors, even

in an efficient manner, due to the multithreading programming model used and to fine-grained locking. On 64-bit architectures, a 64-bit JVM would be able to use all the available memory directly, while no effort has to be put in porting the JavaOS from 32-bit to 64-bit due to the lack of direct memory access (pointers). On the other hand, memory usage would be the major constraint to downsizing a JavaOS , because of the dependency on the classpath functionality.

Modularity

Because of the language it is written in, the JavaOS can take modularity to the extreme. All of the subsystems (except maybe from the classpath) shown in figure 2.3 can be interchanged and extended directly, most of them even when the system is running, in system designed with basic adherence to object-oriented software engineering methods. The JavaOS can thus support dynamically loaded device drivers (modules), alternative subsystem implementations running in parallel, such as multiple filesystem servers and network stacks and dynamically loaded libraries through the use of the system classloader.

Chapter 3

The Nanokernel

```
/* You are not expected to understand this */
```

— comment in the context-switching
code of the V6 Unix kernel

An important portion of the work on JikesNode has gone into the development of the system's nanokernel. *Nanokernel* is a term devised to describe operating system cores that are usually very constrained in size and/or functionality. Most nanokernels are custom implemented to fit the needs of the system and hardware they are designed for. Because nanokernels usually act as an abstraction layer between the hardware and the operating system, they are written in a low-level language, such as assembly, to directly interact with the target processor. In most cases, nanokernels do not include a boot mechanism, but rely on an external boot loader to load them to memory and provide information about the computer hardware. Here, the GRUB boot loader was used.

Currently, the JikesNode operating system is designed to run on the Intel i386 architecture. The many years in development, while maintaining backwards compatibility, has made this architecture quite complicated. All the subsystems that were present in the first implementations of the architecture are still present in today's systems, but are extended or work in parallel with more modern alternatives.

What follows is a detailed description of the nanokernel implementation in JikesNode. We first present the bits of the i386 architecture and the GRUB boot loader that are important to understand the nanokernel operation. Then, the nanokernel design and implementation is described.

3.1 The i386 architecture

Overall architecture

The x86 architecture [15], a generic term describing all generations of the Intel processor family, was first introduced more than 25 years ago. The current generation, the 7^{th}, still maintains compatibility with all its ancestors.

The i386 family are 32-bit CISC processors with an address bus 32 bits wide (36-bit width mode available) . They feature 7 general purpose registers (E[A-D]X, ESI, EDI, EBP) and 2 special registers for the stack pointer (ESP) and the instruction pointer (EIP). The EFLAGS register is 32-bit register used for storing various flags (1-bit fields) such as the zero flag or the carry flag, whose values depend on the result of the executed instructions. On later processors, the control registers (CR[0-4]) are used for controlling advanced processor features such as the Memory Management Unit (MMU). Newer models are equipped with a floating point co-processor (FPU) which introduces another 8 stack-allocated registers (ST[0-7]) along with special floating point instructions. Finally, another recent addition was the support for integer 'single instruction, multiple data' (SIMD) instructions (MMX) which was later expanded to support floating point operations by adding a new set of 128-bit dedicated registers (XMM[0-7]) [1] and the required instructions (SSE), which in turn was expanded to support a wider range of floating point and data movement instructions (SSE2) and finally expanded to enhance thread synchronisation and number conversions (SSE3).

The i386 processor features 3 modes of operation, which drastically change the processor's behaviour. The *real* mode is entered when the computer starts; it restricts the processor to accessing the first megabyte of memory and to using the bottom 16-bit of its registers. The *protected* mode,the normal i386 operating mode, enables the processor subsystems such as the MMU for virtual memory support and full 32-bit operation. The *system management* mode was targeted to power saving, but was quickly obsoleted by more competent technologies. Finally, the virtual-x86 mode allows the processor to emulate the real mode while in protected mode, which is useful to run code written for real mode in modern systems (e.g. the system BIOS).

In order to support address space protection and multitasking, the processor can run programs in 4 privilege levels and also differentiates instruction privilege

[1] MMX uses the FPU registers, which prevents mixing of MMX and FPU instructions

levels from I/O privilege levels. Most operating systems only use level 0 (least restrictive) for kernel mode and level 3 for user mode. In order to switch modes, the processor has to save or restore the full set of registers on the stack the current mode uses.

Memory management

Memory management is arguably the most complicated aspect of i386 programming. There are 2 memory models for an i386 processor which can be used in parallel, segmentation and paging [16, Chapter 3]. As a consequence, three address spaces can be used to access the same memory cell: the *logical* space when segmentation is enabled , the *linear* space when paging is enabled and the *physical* space which represents raw memory addresses. An overview of memory addressing in i386 is presented in figure 3.1 [2].

Segmentation The segmentation model allows memory addressing using the `segment_selector:offset` form (logical addresses). The `segment_selector` field is a pointer to a special system-wide memory construct called the Global Descriptor Table (GDT). A GDT entry consists of a pointer to the beginning of a memory segment and information about the privilege levels required to access the segment. Each process can have up to 6 segment selectors in the GDT; if more are required, the process can have its own segment selector table (Local Descriptor Table (LDT)) which in turn is pointed to by a special entry in the GDT. In order to speed up memory address conversions, there are exactly 6 segment selector registers in the CPU to store the running process's selectors. To get the exact memory cell, the `offset` value is added to the segment's start address, which is pointed to by the GDT entry.

In order for segmentation to work, the GDT table, the segment selectors for each process and the possible LDT tables have to be hand-generated. If more than one process is run, a special GDT entry for each process is created, which points to a task-state segment (TSS) structure. The TSS reserves space to save process context (register values) during task switching. The TSS selector for the current process is loaded to the corresponding processor register.

[2] Source: Intel Architecture Software Developer's Manual, Volume 3: System programming, page 3-2 [16]

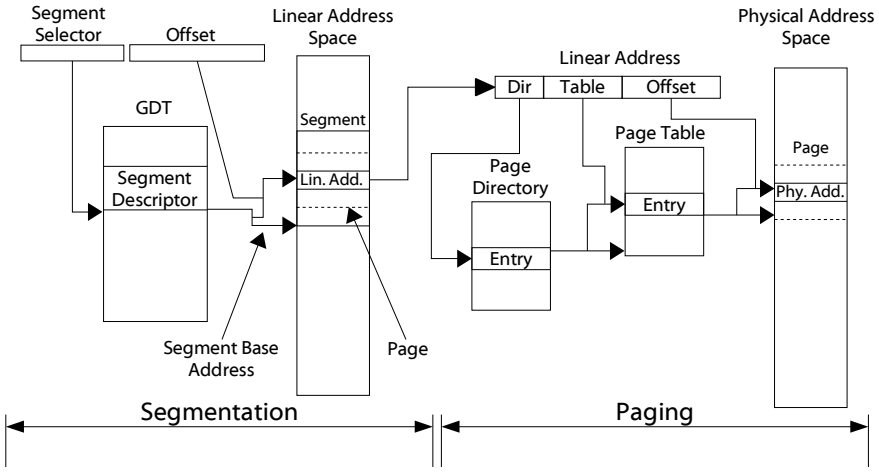

Figure 3.1: Protected mode memory management in i386

Paging Paging allows an operating system to utilise more memory than the host system's physical memory and to isolate the memory spaces used by processes. When using paging, the memory address space is organised hierarchically as a series of directories, tables and pages. A directory holds pointers to 1024 tables and a table holds pointers to 1024 pages. A 32-bit memory address (linear address) represents a memory cell using the `dir:table:offset` encoding; the first 10 bits denote the page directory entry, the next 10 bits denote the page table entry and the final 12 bits are used as a displacement indicator in the 4k page. Address translations from linear to physical addresses are transparently performed by the processor's MMU. Each process can have its own page directory; the CR3 register holds the page directory address for the current process. In case a non-existant page is accessed, a page fault exception is generated and the offending virtual memory address is returned. It is the OS's job to either create the required page by freeing memory or find the required page in the set of pages swapped to the hard disk. The virtual memory subsystem allows page sizes of either 4KB or 4MB. In the latter case, there is no need for page tables and the `offset` value of a linear address uses the last 22 bits of the address.

Interrupts

Interrupts are forced transfers of execution from the currently running program to a special procedure called the *handler* [16, Chapter 5]. Interrupts can be generated by processor-detected exceptional conditions, such as a division by zero,

external device event notifications (in that case called Interrupt Requests (Interrupt ReQuest (IRQ)s)) and can also be software-generated. A i386 processor can handle up to 255 interrupts. A memory construct, called the Interrupt Descriptor Table (IDT), is filled in with specially formatted entries, called task-gate descriptors which point to the routine or task that is executed when the interrupt occurs. The first 16 interrupts are reserved for processor detected errors and are called *exceptions*, the next 16 interrupts are reserved for future use by Intel, while interrupts 32-48 are assigned to IRQ[0-15]. Interrupts from 49 to 255 are free to be used for software generated events. When an interrupt occurs, the processor saves the important registers' values (such as EIP and EFLAGS) on the stack, searches the IDT for the handler whose table entry matches the interrupt number and switches the context to the task handler. In order to service an interrupt, the handler must be run in protection level 0.

IRQs are generated outside the processor and are routed to its interrupt pin by the 8259A Programmable Interrupt Controller (PIC) [13]. The PIC is responsible for handling 8 IRQs, so two PICs are present in each personal computer's motherboard. After an IRQ occurs, the PIC suspends all IRQs until the operating system or the interrupt handler re-enables them.

Time keeping

The i386 architecture defines the existence of 3 clocks: The Real Time Clock (RTC) which is battery operated and keeps track of real world time, the Time Stamp Counter (TSC) which is a high frequency (400 MHz) counter provided by the processor and used for precise timing and the Programmable Interval Timer (PIT). The PIT is a programmable alarm clock usually implemented by a 8254 CMOS chip [14]. It can be programmed to issue interrupts on IRQ0 at a fixed rate and is normally used by operating systems to invoke the process scheduler at regular intervals.

3.2 The GRUB boot loader

A boot loader is the first program that is run by the computer BIOS. It is responsible for loading the operating system kernel in memory from an external resource and transferring control to it. On i386, the boot loader is installed on the first 512 bytes of the first floppy disk or the first hard drive, which is referred to as

the Master Boot Record (MBR). Because this space is often too restrictive, only an initial part of the boot loader is installed on the MBR; the remaining parts are usually installed on the first sectors of the hard disk partition that contains the operating system. Because the boot loader must know the internals of the operating system, such as the address of the start-up method, each operating system uses its own boot loader.

The GRUB boot loader is an effort aiming to produce a boot loader that can directly boot all existing operating systems. GRUB is a two stage boot loader: a bootstrap 512 byte file (`stage1`) is installed on the MBR and the main boot loader executable (`stage2`) is installed on the boot sector of any hard disk partition. The boot loader uses a configuration file (`menu.lst`) to get information about the kernel to load, possible additional files and the kernel command line, if any. The boot loader can access the operating system's filesystem to load the kernel and the configuration file by means of filesystem drivers embedded into `stage2` when the boot loader is installed. At boot time, GRUB reads the configuration file and loads the specified kernel and the additional modules on consecutive memory addresses starting from `0x100000` (1MB) and transfers execution to the kernel entry point. GRUB supports many advanced features, such as network booting and ELF format support, but for our system its the most important feature is support for the Multiboot specification.

The Multiboot specification

The Multiboot protocol [10] aims to unify the different boot mechanisms found in open source operating systems. The specification requires that a special data structure (Multiboot header) is present at the beginning of the kernel executable file. The Multiboot header includes fields that can force the boot loader to provide information about the computer hardware; the information that can be provided varies from probed memory areas to video adapter capabilities. Also, the kernel entry point and the text and data segments can be described in the Multiboot header. The boot loader recognises the Multiboot header by a 'magic number' found at its beginning and, after loading the kernel, sets the EBX processor register to the address containing the information requested by the kernel.

3.3 Implementation

The JikesNode nanokernel has a specific target: to provide the runtime environment for the JikesRVM to run. The nanokernel should only initialise and use those bits of the i386 architecture that are vital for the system. The functional requirements are following:

- Incorporate the Multiboot information header.
- Initialise the memory management subsystem by providing separate segments for user and kernel code. Initialise the kernel space and the user space execution context.
- Initialise the IDT table with interrupt handlers for the required interrupts and convert the interrupts to signals supported by the JikesRVM.
- Initialise other hardware components.
- Use the system console for output and provide a debug mechanism.
- Link the code to the appropriate base address.

The JNode operating system includes a nanokernel that can satisfy some of the requirements set above. The decision was made to base our work on JNode's nanokernel and to extend it to support the features our system needed. The JNode kernel was entirely written in assembly language. We tried, however, to avoid the use of assembler as much as possible in our system and implement the architecture independent pieces in the C language. Table 3.1 describes the distribution of C and assembler in the nanokernel's implementation.

Multiboot information header

As mentioned before, the Multiboot information header has to be directly linked in the first bytes of the kernel. The NASM i386 assembler used for the project compiles the assembly code sequentially, meaning that the output object file's symbols appear in the same order as in the assembly file. Therefore, implementation was straightforward: the Multiboot information header forms the beginning of the `start.s` file, the first file processed by the assembler. The actions requested from the boot loader through the header's `flags` field was to map the usable memory ranges and to align the kernel code on a 4k boundary so its size is a multiple of the memory page size.

Files	Lines of code	Size (source)	Size (compiled)
ASSEMBLER: console.s, cpuid.s, debug.s, heap.s, ints.s, mm.s, start.s, i386.h	1617	34.2Kb	50Kb
C: console.[ch], interrupts.c, kernel.[ch]	669	16.1Kb	16.4Kb
TOTAL: 13 files	2286	50.3kb	66.4kb

Table 3.1: Nanokernel code distribution and sizes

Memory management initialisation

Both segmentation and paging are used in JikesNode. Basic protection between kernel space and user space is provided by employing different memory segments. Paging is not used as a memory management mechanism but as an additional protection mechanism for the kernel code when executing an interrupt handler in kernel space.

All memory management initialisation code is included in the `mm.s` assembly file. The GDT table is initialised with entries for the kernel code and data segments, the user space code and data segments and a TSS entry. The code segments are read-only and the data segments are read-write. The TSS struct is mainly used for preserving the stack pointer state when switching between kernel mode and user mode. The GDT table address is loaded to the GTDR processor register and then a long jump is performed to the kernel code segment. Afterwards, a page directory, a page table and the required pages are created. We use 4k pages for the first 4MB of paged memory and 4MB pages for the rest of the memory. The pages that fall into the area used by the kernel (as denoted by the `kernel_begin` and `kernel_end` assembly labels) are marked read-only. Finally, the processor's ESP register, the stack pointer, is set to the bottom of the space specifically reserved for use as a stack in the `heap.s` file, labeled `kernel_stack`. An overview of the memory status after the memory has been initialised and the system components loaded is shown in figure 3.2.

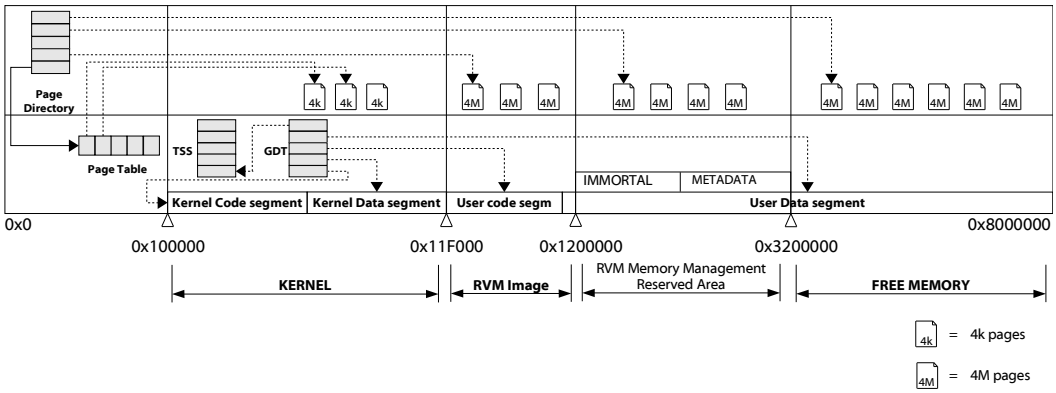

Figure 3.2: System memory after initialisation

Interrupt handler initialisation

The interrupt handler code in JikesNode works in three steps: i) an interrupt specific assembly routine accepts the interrupt, ii) a C function (`default_interrupt_handler`) which maps the interrupt to the appropriate signal is called, iii) and the JikesRVM signal handler is directly invoked by the C callback. The interrupt handler setup code uses a set of assembler macros to prepare the interrupt handler code (`int_{no}error`) and setup the appropriate entry into the IDT table (`intport`).

`int_{no}error`

: For each interrupt, the `int_error` macro generates a `stub_<int_name>` routine which pushes the interrupt specific handler address and the interrupt number (both arguments to the macro) to the stack and calls the generic interrupt handler routine (`inthandler`). The `int_noerror` version is used for interrupts that do not produce an error code and pushes a dummy error code on the stack in addition to what the `int_error` does because the generic interrupt handler expects the same task layout in both cases.

`intport`

: The `intport` macro uses the `stub_<int_name>` autogenerated routine and the interrupt number to produce an entry for the IDT table.

The `inthandler` routine pushes the general registers on the kernel stack, checks whether it is run in kernel mode[3], loads the kernel's data segment descriptor to the DS register and jumps to the interrupt specific handler. Because the JikesRVM signal handler code requires write access to the stack, a copy of the interrupt handler stack is saved in the memory area labelled `int_regs`. When the interrupt handler returns, the stack is restored from the `int_regs` copy, the user code registers are restored from the stack and an `iret` instruction is issued to signal the end of the interrupt handler and switch back to user mode.

IRQs are handled in almost the same way as interrupts, the main difference being that there is no need to generate a stack copy. The C callback function in case of IRQs is `default_irq_handler`. A notable exception is IRQ0 (used by the PIT) whose interrupt handler directly calls the JikesRVM thread scheduler function for efficiency. Also, care is taken to re-enable the PIC after an IRQ handler has finished processing an IRQ, which by default is suspended after delivering an IRQ. Currently, there is no support for transferring received IRQs to the JavaOS IRQ handler. The code for the interrupt handlers is in the `ints.s` and `interrupts.c` files.

Hardware component initialisation

Several hardware components are initialised prior to jumping into user mode. We should notice that we do not provide drivers for generic usage into the nanokernel for those components.

Serial port The serial port is used to redirect the console output to it. A virtual terminal can then be connected to a physical serial port or an emulator can directly dump the serial port to a file. The first serial port is initialised by writing special control words to the `0x3f8` I/O port.

PIT As in most OSs, in JikesNode the PIT is used to trigger reschedule sessions. The PIT is programmed by writing to the `0x43` I/O port. As an initial value, the PIT was programmed to issue 100 IRQ0 interrupts per second. The actual value should be subject to careful experimentation.

FPU The JikesRVM can produce code that uses the floating point unit of the

[3]If it is, then an interrupt has arrived while executing the interrupt handler, a situation which we are not prepared to handle.

processor. Prior to usage, the FPU is initialized using the `fninit` instruction.

System console and debugging support

Console writing support is implemented by directly writing to the video memory address 0xB8000. Each written character consists of its 8-bit ASCII value and an 8-bit attribute value which controls how the character is displayed. In JikesNode, the attribute value is fixed to display a white character on black background. Some basic functions such as `printf` and `putchar` were implemented and the console can scroll text if the text buffer is full. Console output support is only supposed to work in kernel mode, during early system initialisation. A full console implementation is provided by the JikesNode console driver, written in Java. Console functionality is implemented in `console.[ch]`.

The nanokernel features a limited debugging facility that is constrained to printing the processor register values when executing an interrupt handler. The debug function (`dump_registers`) also takes advantage of the disassembler that was plugged into the kernel for use by the JikesRVM signal handler to print a disassembled version of the stack contents. The function is declared in the `interrupts.c` file. An assembly function (`sys_print_intregs`) that prints the register contents, but can work when in user mode, is provided in the `debug.s` file. The function overwrites screen contents.

Linking the nanokernel

The nanokernel uses the ELF [31] binary file format which is directly supported by the GRUB boot loader. The ELF format divides the executable file into a number of sections, two of which are required to be present in every ELF binary: i) the text section for read only executable code and ii) the bss and/or text sections for read-write runtime data. Linkers typically use the data section for initialised variables, for example strings, and leave uninitialised variables for the bss section. Other sections used are the rodata section for variables declared as constants and the dynamic section which holds dynamic linking information.

We used the GNU `ld` linker to link the nanokernel. The link base address was set to `0x100000`, this address where GRUB loads the nanokernel. The GNU C compiler was used to compile the C language files and the NASM assembler compiled the assembly files. In the assembly source code, two sections (code and data)

were declared. The user mode stack and the kernel mode stack declared in the data section. A linker script was used to relocate the two sections appropriately in the final ELF file. The build procedure was intergrated to the JikesRVM build process, analysed in page 45.

Runtime operation

The kernel and the JikesNode Java code module are loaded to memory by the boot loader. After running the memory management and interrupt handling initialisation routines, the kernel executes an interrupt return (`iret`) instruction to switch to user mode. In user mode, the kernel does little more than calculate the beginning and end adresses for the JikesNode module and pass them as arguments to the JikesRVM initialisation code.

Chapter 4

The Jikes Research Virtual Machine

> *If Java had true garbage collection, most programs would delete themselves upon execution.*
>
> — Robert Sewell

Perhaps the most important subsystem of a JavaOS is its Java Virtual Machine. It is responsible for providing core services, such as bytecode compilation to native code and for acting as a resource manager. In general, a JVM that adheres to the specification [22] cannot act as a JavaOS core as the specification specifically assumes that JVMs should be run on top of a real operating system. A significant amount of effort has been put into modifying the JikesRVM to make it run on top of the JikesNode nanokernel.

The JikesRVM emerged as an open source version of an IBM internal project called Japaleño [2]. Since then, it has become the testbed for many institutions that conduct research on virtual machines. As a result, it contains some advanced features such as an optimising bytecode to machine code recompilation system, several GC strategies and a sophisticated thread execution mechanism. The VM itself is written in Java.

This chapter describes those bits of the JikesRVM architecture that were of interest to this project and how the JikesRVM was modified to allow it to run without a supporting operating system.

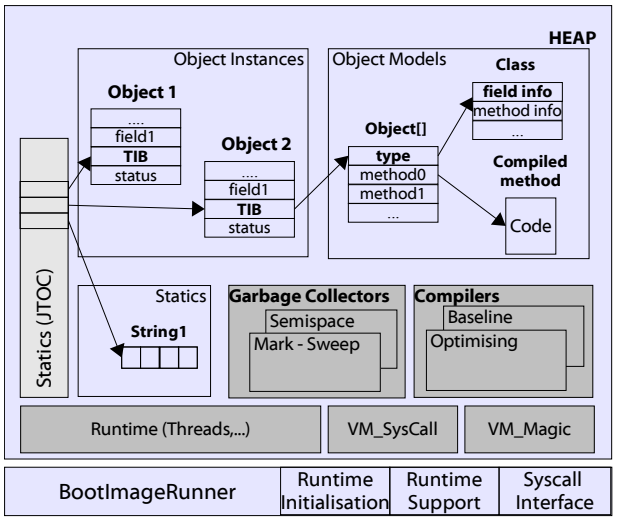

Figure 4.1: High level view of the JikesRVM architecture

4.1 The JikesRVM architecture

An overview of the JikesRVM architecture can be seen in figure 4.1. There are 4 important subsystems in the JikesRVM architecture [12, Chapter 6]:

Core Runtime Provides services to execute applications and to interfacing libraries. Its subsystems include the classloader, the thread mechanism, the bytecode verifier and the system call interface along with the magic interface which are explained in 4.1.1.1 .

Compilers Includes the bytecode to native code compilers and the adaptive optimisation framework. The JikesRVM includes two families of compilers, the baseline compiler which is fast but does not implement any optimisations and the optimising compiler which only compiles the parts of the running program that consume the most time of the program's execution. Runtime information about the program is collected and retrofitted to the compiler by the adaptive subsystem.

Memory Managers The JikesRVM uses a unified, extensible memory management framework called MMTk [5]. It accommodates several GC algorithms and memory layout plans, which are selectable at compile time.

Native Runtime The native runtime is the only part of the JikesRVM that is not written in Java. It is used for three reasons: i) to load the Java image of

the JikesRVM in memory and branch into it (4.1.2), ii) to handle exceptional conditions before they are propagated to the VM (4.1.1.3), iii) to provide a system call interface to the underlying operating system (4.1.1.2).

Because the JikesRVM is written in Java, all its subsystems run in the same memory space as the executed programs. As a consequence, they share global structures such as the Java Table of Contents (JTOC) that stores references to static fields and to all loaded classes' type information blocks (TIB). The memory management subsystem knows about the memory resources the VM uses and does not perform GC on that memory. The information on the memory layout is stored in the MMTk's memory usage plan; the plan divides the available memory into sections, which specifically declare a memory area as a 'garbage-collectible' area or not.

4.1.1 Runtime

JikesRVM uses `pthreads` to multiplex Java threads on physical processors. Usually, for each physical processor a `pthread` is spawn and is encapsulated by the `VM_Processor` class. Each virtual processor holds a set of thread queues to store threads in various processing states, such as the `ioQueue` and the `runQueue`. Using a FIFO scheduling policy, the scheduler always selects the first thread in the `runQueue` and assigns its context to the underlying `pthread`. There is no time slicing policy for the scheduler; a thread stops executing when it voluntarily calls the `yield` method or when it is blocked by a lock. A load-balancing mechanism distributes the threads among virtual processors.

4.1.1.1 Bypassing the type system

Occasionally, subsystems of the JikesRVM must use functionality that is not directly supported by the Java language, e.g. pointer arithmetic or unsafe casts. As an example the thread switching mechanism needs to save the running thread context (processor registers and the stack) before switching to the new thread. The implementing class, `VM_Magic`, uses function prototypes declared as `native` whose existence is known by the bytecode compiler. When the compiler encounters calls to these methods, it inlines the required assembly code for the magic method into the caller method. A special class that implements pointers and pointer arithmetic is also provided (`VM_Address`).

4.1.1.2 System calls

The JikesRVM must use system calls to communicate with the underlying OS and to implement the thread multiplexing functionality described in 4.1.1. The native runtime component implements stub functions for invoking system calls that convert the JikesRVM system call parameters to native call parameters. The boot image header (4.1.2) contains pointer placeholders in preconfigured locations for each implemented system call. These placeholders get filled in with the addresses of the corresponding stub functions. The `VM_SysCall` class provides a set of static functions that use the `VM_Magic` conventions to invoke the native system call.

4.1.1.3 Exception handling

Exceptions in the JikesRVM context can either be normal Java exceptions or processor generated synchronous or asynchronous exceptions. The first case is handled internally by the JikesRVM; in the latter cases, support from the native runtime is required. The exceptions reach the JikesRVM in the form of Unix signals. The signal handler code differentiates between synchronous exceptions (SIGSEGV, SIGFPE, SIGTRAP) which are delivered when the code performs an illegal operation (null pointer access, division by zero, stack overflow respectively), and asynchronous exceptions (SIGALRM, SIGQUIT, SIGTERM) which are usually generated by user intervention. Synchronous signals are handled by creating an artificial stack frame on the current thread's stack with the appropriate values and setting the instruction pointer to the Java exception handler. Depending on the signal, the next instruction to be executed might be needed to resume execution after the signal handling process; a disassembler is then required[1] to examine the current code frame. In case of asynchronous signals, the native runtime manipulates the stack in a similar way, using the JikesRVM exit method as the signal handler.

4.1.2 The boot image

JikesRVM cannot bootstap itself because Java bytecodes are not executable under any operating system. To address this problem, the JikesRVM developers introduced the concept of the boot image, a structured lump of compiled bytecodes whose header is used to identify the location of the Java bootstap code in it.

[1] Only required on the i386 architecure, where the instruction length is not fixed.

The boot image is generated after compiling the Java source files by the `BootImageWriter` program. The `BootImageWriter` uses an external JVM to load and initialise the JikesRVM in a special boot image mode. The initialised JikesRVM loads a set of pre-defined classes (referred to as *primordials*), compiles them to native code and outputs the compiled code in a byte array. It then appends the boot image header to the beginning of the byte array and fills it in with the offsets of important system components such as the JTOC and the initial processor object.

4.1.3 The build system

The JikesRVM build system is a collection of Unix shell scripts. It is quite customizable but hugely complicated. The configuration script (`jconfigure`) uses two files which contain configuration parameters for a) the VM runtime environment, such as the underlying operating system (from now on HOST_CONFIG) b) the combination of compiler and GC implementation to use (VM_CONFIG) [12, Chapter 2.3].

Among others, the HOST_CONFIG file specifies some parameters especially important for the project:

- RVM_FOR_SINGLE_VIRTUAL_PROCESSOR: Controls whether JikesRVM should use `pthreads` to map Java threads to physical processors or not.
- BOOTIMAGE_LOAD_ADDRESS: The physical address where the bootimage should be loaded by the operating system. The JikesRVM load address has to be specified in order to avoid conflicts with dynamically loaded shared libraries on the target system.
- MAXIMUM_MAPPABLE_ADDRESS: Controls the maximum physical address the JikesRVM can access.

The `jconfigure` script outputs a set of build scripts whose contents are based on parameter values defined in the configuration files. The build process is divided into four phases which: i) collect the required files into the build directory, ii) compile the required Java files, iii) build the boot image, iv) compile the native runtime C++ files. Additional phases can be inserted, for example to compile an additional module, but the build system does not support incremental compilation. Finally, the build system features a preprocessor that can include or

exclude Java code from the compilation stage, similarly to the C preprocessor.

4.2 Implementation

As discussed in 2.2.1, a JVM has two roles in the context of the JavaOS : to provide the basic execution environment and to manage low-level shared resources, namely memory and interrupts. JikesRVM, and in particular the native runtime environment, had to be extensively modified in order to be able to run without the support of the operating system. Some minor modifications are also needed to implement resource management functionality.

4.2.1 Changes to the runtime

The native runtime from the i386 Linux JikesRVM distribution was used as a basis for our native runtime implementation. The source code for the system call interface (`sys.C`), the signal handlers (`libVM.C`) and the command line parser (`cmdline.h`) was imported. A major concern throughout the process was the JikesRVM's interdependencies with the GNU classpath. Because both JikesRVM and Jnode use the GNU classpath, though in a different manner, the changes made to the classpath are described in section 5.2.

To simplify the development of an initial prototype and also to speedup the build cycles, the simplest configuration possible was chosen for JikesRVM. The RVM_FOR_SINGLE_VIRTUAL_PROCESSOR parameter was set to disable the dependency on the `pthreads` library (which our nanokernel does not provide) and the JikesRVM was built with the prototype configuration set. This set provides the baseline compiler and a simple mark-sweep GC.

System calls JikesRVM uses system call stub functions to implement access to files, network connectivity and threading. This functionality is exported to the Java programs through the classpath. In the context of JavaOS though, these services should normally be provided to the system by higher level components, but also be exported through the classpath. Therefore, the JikesRVM runtime only has to implement a small subset of the system call interface that will allow it to operate until the Jnode operating system is loaded.

The first step was to decide which system calls were needed to boot the JikesRVM until it could be able to use its own classloader to load external classes.

Group	Implemented	Comment
Console Output	`sysWrite*`	Use the `printf` function provided by the kernel
File manipulation	-	Provided by the Jnode filesystem layer
Math Functions	`*Float*`, `*Double*`	Implemented by importing code from FreeBSD `libc`. Mainly used to speed up type conversions.
Memory management	`sysMmap`, `sysCopy`, `bzero`	Used GCC assembly built-in functions for `*Copy, *bzero`. `sysMmap` always returns true
Networking	-	Provided by the Jnode networking layer
Thread Manipulation	-	Not applicable
Thread Switching	`processTimerTick`, `*TimeSlicer*`	`processTimerTick` called directly by the timer interrupt handler, `*TimeSlicer*` used by the VM boot method.

Table 4.1: Implemented system call stubs.

The first system calls implemented were the `sysWrite*` functions that write messages to the console by directly calling the kernel's `printf` function. For the rest of the system calls dummy implementations were provided which do little more than writing their name and arguments to the console. That allowed to easily identify all the functions needed when JikesRVM boots. As it turned out, only a very small subset of system calls were actually used. Table 4.1 lists the system call groups that were implemented.

Signal Handlers The logic of signal handling has not changed in the JikesNode JikesRVM native runtime. However, the semantics did change as a result of the differences in signal generation in the nanokernel. There is no formal support for Unix like signals in the nanokernel; signal handlers are directly called from the corresponding interrupt handlers. The correct signal identification number, the interrupt memory address and the interrupt handler stack (section 3.3) are passed as arguments to the signal handler in order to minimise the amount of changes to it. Some stack bound checks were also removed due to differences in the location of the currently used stack; in normal operation, the signal handler runs on the user space stack whereas, in JikesNode, the kernel space stack is used

because the caller, the nanokernel's interrupt handler, is executed in kernel mode.

Command line parser The command line parser is used by the JikesRVM to convert its command line to runtime options. The command line parser was imported along with some string related C functions needed for its operation. The initial plan was to use the command line parser to exploit the command line feature the GRUB boot loader offers in order to dynamically set options to the JikesRVM through the GRUB configuration file at boot time. Unfortunately, problems in the version used for the project prevented GRUB from returning the command line passed to it. However, the command line parser can still be used by assigning it to a static string and recompiling the kernel.

4.2.2 The build system

The JikesRVM build system was changed to accommodate an extra compilation stage that processes both the JikesRVM native runtime sources and the nanokernel sources. The integration was necessary as long as there are direct function calls across the boundaries of the two subsystems. The `jconfigure` script was modified to output an extra builder script, named `jbuild.jnode`. The new builder first copies the native source files in the JikesRVM build directory and then compiles them into a single binary file. The linking stage is described in section 36.

An important consideration during the development of the build script was to allow the build system to construct the JikesRVM image without depending on the compilation of the runtime. This approach had a strong prerequisite: the link address of the JikesRVM image should be known prior to its compilation. When running on top of an OS, JikesRVM does not need to care about its link address; the OS virtual memory subsystem can load it to any address. In fact, the JikesRVM developers have set the BOOTIMAGE_LOAD_ADDRESS configuration parameter to a rather high number (`0x4300000`) to avoid conflicts with shared libraries. In our case, the GRUB boot loader loads the JikesRVM image directly after the nanokernel. Since the nanokernel's load address is known (`0x100000`), the JikesRVM load address could be guessed by adding the nanokernel size to its load address. The resulting address is then assigned to BOOTIMAGE_LOAD_ADDRESS parameter and the build process is restarted. The proposed solution does not always work; the problem seems to be in the algorithm used by GRUB to decide the address to load the JikesRVM image. Most of the times, this address is the

nanokernel end address plus a padding to align it to a 4k boundary; however, sometimes, GRUB adds another 4k to the padding, as if it had to align the nanokernel address to an 8k boundary. The approach used in the project always aligns to a 4k boundary.

4.2.3 Changes to the VM

A small amount of changes had to be done to the VM. For all changes made, the simplest possible approach was followed. In case a VM subsystem had to be disabled, the subsystem's initialisation method was removed from the JikesRVM initialisation method (`VM.boot()`). We relied on the Java exception mechanism to receive errors in case of using an uninitialised subsystem. First, the JNI support was disabled because, obviously, there is no need of running native methods in JikesNode. Some class initialisation methods that depended on the JNI subsystem were also disabled. A few system calls were added to support reading to and writing from I/O ports and to read the result of the CPUID processor instruction. The required native system calls were added to the native runtime.

4.2.4 Not implemented functionality

Lack of adequate time prevented the addition of the following required functionality to the VM:

- A native runtime method for handling IRQs and propagating them to the JNode IRQ manager. This change would probably require the development of an asynchronous notification mechanism, similar to the exception handlers.

  Implementation suggestion A new method, for example `deliverIRQ`, has to be added in the `VM_Runtime` class. This method should disable the GC and call the corresponding static method in the JNode IRQ manager. Because the `deliverIRQ` method has to be called from the native runtime, a JTOC entry is required in the `VM_BootRecord` class. The native counterpart should only call the Java method with the IRQ number as an argument.

- Support for managing Direct Memory Access (DMA) areas in the MMTk.

- We did not check the optimising compiler or advanced memory managers with the current runtime implementation.

4.2.5 Runtime operation

After the nanokernel has initialised the computer components, it calls the `checkMultiboot` kernel function to calculate the size and the initial address for the loaded JikesRVM boot image. It then branches into the JikesRVM native code component, which calls the `setLinkage` method to link the system call stub pointers to the boot image header. An assembly routine takes over and transfers the control to the Java code boot thread. The `VM.boot` method is called and after initialising the memory manager and the scheduler, JikesRVM is ready to run the class specified in its command line.

Chapter 5

Merging the components

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.

— Gerald M Weinberg

5.1 The JNode operating system

The JNode operating system is a recent effort to create an environment in pure Java that will be able to execute Java bytecodes natively. It includes a VM implementation, a custom version of the GNU classpath and a device driver infrastructure with device drivers for many widely used hardware components. It can be directly booted on the i386 architecture and execute simple programs. It does not support multiprocessing or multiple simultaneous users yet.

The JNode distribution is split into 6 packages, whose contents can be seen in table 5.1. Not all sub-packages are at the same level of development; in particular, while the core and shell packages are fairly complete, the net and especially the gui package are still very immature. For JikesNode, the core, shell and fs packages were used.

5.1.1 Components of the JNode architecture

5.1.1.1 The plug-in architecture

All components of JNode, except for the VM, the classpath and the plug-in architecture itself, are plug-ins to the system. A plug-in has an associated XML

Package	Description
core	Includes the classpath implementation, the JNode VM, the resource managers, the plug-in manager and the device driver infrastructure and the device drivers.
fs	Devices drivers for block devices and implementations of various filesystems.
shell	Command line implementation and various shell commands.
net	Device drivers for network interfaces, network protocol implementations and command line clients for various protocols.
gui	Attempt to implement the java.awt package, in a very early stage.
builder	`ant` based builder tasks for building the system and creating the boot disks.

Table 5.1: The JNode project packages

descriptor and is contained in a Java ARchive (JAR) file. Plug-ins are assembled during the build process and are contained in the initial boot image. Plug-ins can define extension points which are subsystem or domain specific implementations of functionality that can be accessed with the same interface. As an example, all filesystem drivers are hooked to the `org.jnode.fs.types` extension point and thus their existence is known to the system and their functionality can be accessed through the `FileSystemType` interface. Other types of plug-ins are libraries, which provide shared implementations of common functionality (e.g. the XML parser), and runtime plug-ins which export commands that can be started through the shell. Each plug-in has its own class loader and can access a limited set of classes, namely system plug-ins and predefined (in the descriptor file) library plug-ins.

The central part of the plug-in architecture is the plug-in manager. It is the container object of the plug-in registry and the plug-in loader and defines methods to start, stop and load plugins. The registry holds information about the loaded plug-ins which is derived from the plug-in descriptor files and is responsible to use the provided plug-in loaders to load plug-ins and resolve dependencies. A plug-in loader is used to load a plug-in from a resource, such as a URL or the boot image JAR file. Different instances of plug-in loaders are managed by the loader manager.

5.1.1.2 The device driver infrastructure

All device drivers in JNode share a common infrastructure. A driver controls a device and is an implementation of an API specific to the device hardware. An API is a common method of controlling a class of devices that share common operating characteristics, for example block or character devices. A bus is a resource manager for a set of devices that are connected to it; it can register for receiving interrupts to the resource manager, execute commands specific to the hardware bus it encapsulates and probe the hardware for a device. All buses are organised below the system bus, which is the root of the system device hierarchy tree. A device is a software representation of a hardware device and holds references to its bus, to the controlling driver and to the APIs it implements and can emit notifications for specific events, such as start and stop events.

The controller of the device driver infrastructure in JNode is the device manager, which is a system plug-in. Associated with the device manager are the device finders and the device mappers, which are defined as plug-in extension points. A device finder searches for and reports devices found on a particular bus. Device mappers search for device drivers for a particular device. The device manager holds references to all devices, the device finders and the device mappers in the system. When a new device is found by a device finder, it is registered to the device manager and device mappers are queried to return a device driver for the device. For buses that support asynchronous notifications when a new device is connected to them (e.g. USB), an event mechanism notifies the device manager which then undertakes to map a driver to the device.

5.1.1.3 Resource management

The JNode OS can share 3 distinct types of resources among Java threads: i) Memory regions, ii) IRQ's and iii) I/O ports. Each resource can have a resource owner, and depending on the resource, it can be shared among owners. Memory regions and I/O port resources are handled in the same manner. A system-wide table for each resource associates each resource unit with a resource owner. Synchronised methods provide access to that table by the associated resource manager. In order to claim an I/O or memory resource, a system component first tries to discover the specified resource manager through the system-wide naming service and then calls the `claimIOResource` or `claimMemoryResource` methods.

These methods return resource-specific interfaces for accessing the claimed resource, for example writing to I/O ports.

IRQ resource management is somewhat different because IRQs can be shared and the resource manager must respond to external events. A system global table counts the number of IRQs occurred since the beginning of the system operation and is directly updated by the nanokernel's generic IRQ handler function. Each IRQ is associated with a service thread, which holds references to the IRQ handler methods. At regular intervals, the thread scheduler awakes the IRQ service threads, which call the respective IRQ handlers if the IRQ count table entry is greater than the count recorded by the handler.

Finally, a system-wide resource manager offers uniform access to all resources. The resource manager is part of the JNode VM and is directly used by its GC to mark system-only read-only and user allocable read-write memory areas and to provide an interrupt handler (for IRQ0) that drives its scheduler.

5.1.2 Changes to JNode

The main change that had to be done to JNode was to remove the VM it incorporates and adapt it to work with JikesRVM. The VM in JNode is resource aware and includes the functionality discussed in Section 5.1.1.3. The goal was to completely remove the VM's garbage collector, the code generation classes and the classloader implementation while preserving the interface to the nanokernel, the resource management functionality and the architecture description classes.

Removing the VM As there was no documentation on the JNode VM architecture, a trial-and-error approach had to be followed. The org.jnode.assembler and the bytecode, classmgr, compiler and memmgr sub-packages of the org.jnode.vm package were removed at the beginning of the process, as they clearly implemented parts of the VM. A class diagram (automatically generated by the UMLgraph tool) was used to identify some important classes, while others were considered as important just because of their name. The "important" classes were a very small percentage (about 10%) of the total VM classes. The Java compiler's error output was used to identify missing dependencies which were added back to the compilation after careful inspection and if and only if they did not contained undesired functionality.

The nanokernel interface The nanokernel interface in JNode is mainly provided by two classes, `Unsafe` and `Address`. These classes play the same role as `VM_Magic` and `VM_Address` in the case of JikesRVM, the main difference being that in case of JikesRVM the compiler knows their existence and treats them differently while in JNode a pseudo-JNI implementation is used. Not all methods of `Unsafe` were needed for JikesNode, as many of them were used as support for the JNode VM, for example to circumvent the JNode type system. The implementation used the JikesRVM equivalents of the specified methods and also added some system calls to JikesRVM, as discussed in section 4.2.3.

5.2 The classpath

The classpath is the base of the Java language. Both JNode and JikesRVM use the GNU version of the classpath. The GNU classpath has been developed for quite a long period of time, but since the exponential increase of the core Java API's in Java version 1.4, its evolution no longer represents the current state of the language. Nevertheless, it supports the basic language packages (java.lang, java.util, java.net) well enough to enable complex applications, such as JikesRVM, to work.

The classpath implementation is divided into two namespaces: the standard java and javax namespaces which implement the classpath classes as described in the specification and the gnu namespace that specifies internally-used classes. The classpath is not written in pure Java, as one might expect, because it has to interface many OS subsystems in order for programs to work. Examples include the windowing system and the network stack. For that reason, the classpath uses native code libraries, which are dynamically loaded through the JNI mechanism. Also, some language features need special support by the VM and cannot be implemented in the library, for example the reflection and thread API's. In order to be VM independent, the classpath uses abstract classes which are expected to be replaced by the implemented VM with concrete equivalents at runtime. It is however possible to configure the classpath to not directly load and use external native libraries but to expect that the runtime will load the libraries on demand by some external mechanism. In that case, even a runtime that does not support JNI can use the classpath by losing a little of its functionality.

JikesRVM and JNode take different approaches on how they use the classpath. In JNode, all classes that use native methods have either been replaced by pure Java classes that use the JNode operating system services or completely removed. The classpath's `make`-based build system has been replaced by an `ant`-based equivalent. In JikesRVM, the classpath build is controlled by the standard `jbuild` builder and is performed by using the classpath's own builder. There was no need to modify the classes because JikesRVM supports JNI; only the required abstract classes are reimplemented by JikesRVM.

The differences described above presented a strong challenge in the effort to merge the two systems. In addition, the two systems used different versions of the classpath; in open-source programs, especially those whose code has not stabilised, even a minor revision version can lead to many changes in the source code, and unfortunately this was the case with the classpath. Two important decisions had to be made at the beginning of the effort:

1. Which version of the classpath should be used.

2. Which should be the preferred method to build the classpath.

The solution should i) minimize the changes needed to the source code and ii) minimize the changes needed to the build system. The decision was mainly based on the fact that the classpath should be the first component to be built as both JNode and JikesRVM depend on it. Also, the JikesRVM should be compiled before JNode because JNode uses functionality present in internal JikesRVM classes. Thus, the default implementation of the classpath was used as a basis. This decision was further reinforced by the fact that the JikesRVM build process makes a number of assumptions about the location of files produced by the classpath build process. Breaking those assumptions would require a fair amount of work on the JikesRVM build system.

The changes made by the JNode team to the classpath had to be backported to the version we used in order for the JNode services to be available to the JikesNode OS. The JNode builder offered a method to identify the differences between its own classpath version and the stock classpath. Its output was rather complicated, but offered good hints as to which classes the porting effort should focus on. Most differences, as expected, were in the java.io, java.net and java.util packages. The differences in the java.lang and java.lang.reflect packages were not backported because most of these classes form the VM specific part of the classpath, which is

Class	C	Comment
java.io		
DataOutputStream File FileDescriptor FileInputStream FileOutputStream RandomAccessFile	M	Completely replaced with JNode version. Possible conflicts due to different versions. New versions use the VM* classes to interface with the JNode filesystem API.
FileNotFound-Exception	M	Added `FileNotFoundException (String, Throwable)` and `FileNotFoundException (Throwable)` constructors.
IOException	M	Added `IOException (String, Throwable)` and `IOException (String, Throwable)` constructors.
VMFileHandle VMFileSystemAPI VMIOUtils VMObjectStreamClass VMOpenMode	A	Interfaces that are implemented by the JNode filesystem API.
java.util		
jar.JarFile zip.ZipFile	M	Use the `RandomAccessBuffer` for processing memory-mapped JAR files.
zip.RandomAccess-Buffer	A	Provides methods for the manipulation of memory-mapped files. Used to access the initial JAR classpath file.

A	Added
M	Modified

Table 5.2: Changes to the classpath.

already implemented by JikesRVM. The java.net package was also not backported because it would depend on the jnode-net module which was outside the initial porting plan. Once again, the Java compiler's error output was used to identify which classes were missing or had unsatisfied dependencies. Table 5.2 summarises the changes made to the classpath.

5.3 The build system

The build system is where the three components are brought together. In its current state, it incorporates the three different build systems of the system

components and supports component-level build dependencies.

Before actually creating the build system, the build procedure and the non-functional requirements had to be decided:

- The JikesNode build system should use the existing component build systems with the least possible changes.
- The build system should be able to identify the dependencies at a component level, in order to avoid building an already up-to-date subsystem.
- The dependencies for the system's boot image should be automatically extracted and the boot image should be generated by the build system.

The step-by-step approach followed for the development of JikesNode has generated important preconditions that had to be taken into consideration when designing the build system. First, the JikesRVM build system incorporates an automated build for the classpath, using the classpath's original `make`-based mechanism. The JNode project does also include a classpath build procedure, but it is not based on the original classpath build system. As discussed above, the classpath was the first component to be built, because both JikesRVM and JNode depend on it. Also, the JikesRVM build system had been already modified to include the nanokernel build (Chapter 4.2.2), and, since it was working acceptably, no changes should be made to it. Finally, the build system should include a phase that builds the system's boot disk in order to ease the build-test cycles.

5.3.1 Implementation

The subprojects were imported into separate directories. Each directory also contained the build script each project used. The top level directory was added to a CVS tree to ensure proper backups and code versioning. The `ant` build tool was used to coordinate the build process. We did not use the commonly used `make` tool for three reasons: i) The JNode build system is based on `ant`; while it is trivial to use other build systems from `ant`, it is not so from `make`, and rewriting or adapting the JNode build system is not trivial either. ii) There are strong indications (in the JikesRVM development lists) that the JikesRVM build system would be moved towards `ant` sometime in the future. iii) The author was more acquainted with `ant` than with `make`. A top-level `ant` script was written that recursively invokes the build system in each component subdirectory. A component level dependency

mechanism was developed to avoid rebuilding components that were up to date, using the `ant`'s conditional building feature. Finally, the UMLgraph tool was included to the build script to automatically extract UML class diagrams that helped understand the structure of the subsystems written in Java.

5.3.2 The boot image

The JikesNode boot image is used to assemble the system components in a format suitable for loading them to memory and starting the system. Both JNode and JikesRVM include a boot image build step in their build process; the two systems are similar in principle but differ considerably in the implementation. The JNode boot disk builder builds the jnode-core package and the modified classpath by booting the JNode VM in a special build mode, loading and compiling all the classes. The main difference is that the JikesRVM image builder tries to minimize the created image size by only compiling the necessary classes for the VM to load (the primordials).

For the JikesNode build system, the JikesRVM image builder was used. The default build process for JikesRVM had to be stopped in order to allow the JNode JAR files to be built prior to creating the boot image. To achieve that, a custom switch is implemented (`-compile`) into the `jbuild` script. A custom shell script is used to parse the debugging output of the Sun's Java Compiler in order to extract a list composed of the compiled JNode classes and the classpath depedencies of those classes. This list is then appended to the primordials file of the JikesRVM build process. Care is taken to avoid duplicate entries as they confuse the image builder. The JikesRVM build is then resumed and the image builder is given the new primordials file as input. The resulting image file should be directly bootable, but the described system is not thoroughly tested yet.

In order to load the nanokernel and the JikesRVM image to the system memory when the computer boots, the typical PC boot conventions had to be used: The system should be able to find the boot loaded at the start of the boot partition and the boot loader would then be responsible to load the images in memory and branch into the kernel. On a normal PC, the operating system and the boot loader usually reside on the hard drive. The need for fast compilation-testing cycles during the development of JikesNode prohibited the use of an external computer for testing; instead, the `VMWare`, `bochs` and `qemu` emulators were used. All three emulators support booting from exact copies of hard disks, usually

referred to as disk images. A disk image is a file whose contents are formatted exactly like a hard disk platter meaning that it is divided in sectors and contains a master boot record and a master file table. The process of creating a boot disk file and installing a boot loader on it is described in appendix B.

The boot image creation step is the last step of the build process and is performed into the `jbuild.jnode` (Section 4.2.2) script developed for the nanokernel builder. The script uses the `mtools` package to access the disk image and copies the JikesNode boot image and the GRUB configuration file into it. Finally, the `mkisofs` tool creates a bootable CDROM image containing the boot disk.

5.3.3 Not implemented

Lack of adequate time prevented the complete integration of the JikesNode components. In the author's opinion, the following changes have to be made before the system is fully functional.

- The JNode boot image, apart from the nanokernel and the compiled VM and classpath, contains an archive of the system plug-ins to be used, referred to as `initJar`. The current JikesNode build process does not build plug-ins, although it compiles the required Java files. The plug-in builder from JNode has to be ported to the JikesNode build system. In order for the system to use the plug-ins, two system calls have to be implemented in the JikesRVM native runtime and then exported through the `VM_SysCall` facility to the JNode `Unsafe` class.

- The built JikesNode image has not been tested. Moreover, only the necessary bits of the classpath are included in it. The JNode boot image contains the full classpath compiled in it. The author's opinion is that the same should go for JikesNode. The problem is that the boot image size is predicted to be more than 45MB. A solution to this problem is not to include parts of the classpath that are really unnecessary, such as `java.awt` or `javax.swing`.

- The JNode main class has to be known to JikesRVM in order to be called after the VM initialisation. There are two solutions to that: i) Directly call the JNode init method from the JikesRVM init method and ii) use the JikesRVM command line to specify the method to load first. In the first

case, the JikesRVM primordial classloader is going to be used while in the second a normal application classloader is needed. The correct solution might only be found by experimentation.

- A mechanism that accepts IRQs and directly calls the required interrupt handlers could replace the polling mechanism used in JNode. Alternatively, the generic IRQ handler can be used to update the IRQ manager tables and a seperate thread should check for recently received IRQs in regular intervals. Changes to the JikesRVM are required, as discussed in section 4.2.4.

- A small number of minor tweaks might be needed to the JNode core code, especially in the `Unsafe` class, which is the main entry point to JikesRVM.

Chapter 6

Conclusions

Multithreaded Java programs are perfectly suited to thread-aware chip multiprocessors [34]. Java operating systems, being massively multithreaded, can be the platforms for developing and deploying applications that exploit the capabilities of such architectures.

This thesis examined the feasibility of merging an advanced Java operating system (JNode) with a high performance Java virtual machine (JikesRVM) to form a new operating system called JikesNode. The engineering work was performed on an established platform (Intel i386), to avoid the infancy problems of chip multiprocessor architectures. The project was successful in providing a native runtime environment for the virtual machine to run on and in making the majority of the required changes to JikesRVM and JNode in order to work together. Time constraints in conjunction with the, sometimes, overwhelming complexity of the project subsystems prevented the completion of the merging effort and the process of system testing.

First, a thin abstraction layer, the nanokernel, sitting between the hardware and JikesRVM, was built. The existing JNode nanokernel was leveraged and several new features were added to it. Special effort was put into minimizing the effect it has on system performance by providing shortcuts for frequently called functions, such as the timer interrupt handler (section 3.3), a feature not existing in the original JNode nanokernel. Also, the decision to integrate the nanokernel with the JikesRVM native runtime component allowed for minimising the functionality that needed to be implemented in the nanokernel to support JikesRVM; subsystems such as Unix style asynchronous notification schemes (signals) were replaced by direct function calls from the signal source. In its current state,

the nanokernel runs stably enough for a JVM to run on top of it. Changes are required to it to support multiple processors or advanced memory management schemes.

Furthermore, the JikesRVM was successfully modified to make it boot without the need of an operating system. Both the native code and the Java code needed a significant amount of change, mostly concerning the integration with the nanokernel. Fortunately, concepts in the design of the core of JikesRVM, mainly the JTOC table, allowed us to change the way JikesRVM operates with respect to signal handling and loading to memory, without needing any changes in the core system. The output of a sample run of the modified JikesRVM on top of the nanokernel can be seen in Appendix A.

The final stage of the work proved to be the most challenging due to the size and complexity of the code that was involved. It included the merging of three distinct projects featuring three different build systems. A number of different tools were used to check the differences and mark the pieces of code that needed to be changed or imported. Two custom tools were used to identify dependencies of the JNode classes to the main library classpath and to build the disk boot image. The project integrated the classpath, JikesRVM and the core JNode functionality to a single build system. A small amount of further change was left to be done in order for the system to fully function.

Lessons learned

The project was a worthwhile exercise. Having never worked on systems programming before, we gained valuable experience and also came in touch with emerging technologies and future trends in operating systems and hardware design. The most important lessons learned throughout the process were the following:

- The project proved the capabilities of the Java language as a generic programming tool. With careful design and a clever bootstrapping mechanism both JNode and JikesRVM were able to combine the advanced type safety and memory management features of Java with accessing low-level hardware, only requiring a tiny compatibility layer. Device drivers and protocols are much easier to implement in Java, mostly due to the advanced structures available in the library, and the resulting code is sometimes smaller that writing it using a low-level language like C.

- The assertion made in [27, Chapter 6.6] that the complexity of large projects cannot be dealt without employing a significant amount of meta-programming was proven valid. Many of the used tools, for example `ant`, `grep` and `awk`, required declaring program structures in their custom domain-specific languages. Tackling large projects without being familiar with these languages or relying on an IDE to provide complex search and build functions can only lead to significant time losses or even loss of interest from the developer's part. The Unix programming paradigm of combining tools to perform complex tasks [18] was the preferred method of dealing with problems during this project. When needed, we did not hesitate to build our own custom tool to help us perform a particular complex task easily.

- Working with a small group doing advanced research was probably the most important experience gained. We had the opportunity to communicate and cooperate with knowledgeable people and to learn the methods employed and the tools used to coordinate a large project.

Future work

A working prototype of the JikesNode operating system is about 85% complete. The remaining work to be done is highlighted in various points throughout the thesis (sections 4.2.4, 5.3.3). A conservative estimate for the time required for the successful completion of the project would be about one man month. This includes the time to understand the project's background and the work carried out so far.

Java operating systems is a relatively new research area in computer science. Current implementations tend to provide solutions to well defined problems rather than focus on building a platform for running applications on. JikesNode is, to the best of our knowledge, the first implementation that is based on a solid well-performing, continuously evolving virtual machine and that provides an extensible architecture and working implementations of the crucial subsystems.

Further work on the JikesNode platform can include:

- Implementation of a process model in order to allow multiple programs to run concurrently. The isolation primitives and the application loading and initiating semantics have to be examined. The currently existing approaches are discussed in section 2.2.1.

- Support for multi-user operation. The notion of user does not exist in the current implementation. The research can include, among others, the implementation of security properties (perhaps based on the Java security domains), the possibility to assign users to running threads, implementation of user databases and more.

- Support for legacy applications. Work could be done to embed the dynamic binary translator developed in the Jamaica group[23] for running native code applications.

- Testing. The system could be tested by running real world applications which are best suited for multithreaded operation such as web servers or J2EE servers.

JikesNode could provide a good starting point for all these usefull projects and in general can be considered as a testbed for novel research on operating systems architectures.

Bibliography

[1] Acetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for Unix development. In *Proceedings of Summer USENIX 1986 Technical Conference* (Atlanta, Georgia, June 1986), USENIX, pp. 93–112.

[2] Alpern, B., Attanasio, C. R., and Burton, J. J. The Jalapeño virtual machine. *IBM Systems Journal* (2000).

[3] Back, G., Hsieh, W. C., and Lepreau, J. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation* (San Diego, CA, Oct. 2000), USENIX.

[4] Bershad, B. N., Chambers, C., Eggers, S. J., Maeda, C., McNamee, D., Pardyak, P., Savage, S., and Sirer, E. G. SPIN - an extensible microkernel for application-specific operating system services. In *ACM SIGOPS European Workshop* (1994), pp. 68–71.

[5] Blackburn, S. M., Cheng, P., and McKinley, K. S. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of ICSE 2004, 26th International Conference on Software Engineering* (Edinburgh, Scotland, May 2004).

[6] Bovet, D. P., and Cesati, M. *Undestanding the Linux kernel*, first edition ed. O' Reilly, 2001.

[7] Chen, J., and Bershad, B. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM symposium on Operating systems principles* (1993), ACM Press, pp. 120–133.

[8] Drossopoulou, S., and Eisenbach, S. Java is type safe — probably. In *European Conference On Object Oriented Programming* (Berlin, June 1997), M. Aksit, Ed., vol. 1241 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pp. 389–418.

[9] Engler, D., Kaashoek, M., and O'Toole, J. Exokernel, an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)* (Copper Mountain Resort, Colo., Dec. 1995), ACM, ACM Press, pp. 251–266.

[10] Free Software Foundation. *The Multiboot protocol*, Jan. 2004. http://www.gnu.org/software/grub/manual/multiboot/.

[11] Golm, M., Felser, M., Wawersich, C., and Kleinoder, J. The JX operating system. In *USENIX Annual Technical Conference* (June 2002), pp. 175–188.

[12] IBM. *The Jikes Research Virtual Machine User's guide*, 2004. Manual accompanying the JikesRVM source distribution.

[13] Intel Corporation. *8259A Programmable Interrupt Controller*, Dec. 1988.

[14] Intel Corporation. *8254 Programmable Interval Timer*, Sept. 1993.

[15] Intel Corporation. *Intel Architecture Software Developers Manual, Volume 1: Basic Architecture*, 1997.

[16] Intel Corporation. *Intel Architecture Software Developers Manual, Volume 3: System programming*, 1999.

[17] The Jnode operating system, 2004. http://jnode.sourceforge.net.

[18] Kernighan, B. W., and Pike, R. *The UNIX Programming Environment.* Prentice-Hall, 1984.

[19] Liedtke, J. Improving IPC by kernel design. In *Proceedings of the 14th ACM symposium on Operating systems principles* (1994), ACM Press, pp. 175–188.

[20] Liedtke, J. On micro-kernel construction. In *Proceedings of the 15th ACM symposium on Operating systems principles* (1995), ACM Press, pp. 237–250.

[21] Liedtke, J. Towards real microkernels. *Commun. ACM 39*, 9 (1996), 70–77.

[22] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification*, 2nd ed. Addison Wesley Professional, Apr. 1999. Also available online at ftp://ftp.javasoft.com/docs/specs/vmspec.2nded.html.tar.gz.

[23] Mattley, R. Native code execution within a jvm. Master's thesis, University of Manchester, Sept. 2004.

[24] McKusick, M. K., Bostic, K., Karels, M. J., and Quaterman, J. S. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, 1996.

[25] Nicholas, T., and Barchanski, J. A. Overview of TOS: a distributed educational operating system in Java. *SIGOPS Oper. Syst. Rev. 34*, 1 (2000), 2–10.

[26] The Jikes Research Virtual Machine (RVM), 2004. http://oss.software.ibm.com/developerworks/oss/jikesrvm/.

[27] Spinellis, D. *Code reading: The open-source perspective.* Effective Software Development. Addison Wesley, 2003.

[28] Sun Microsystems. *JavaOS: A standalone Java environment*, Feb. 1997. http://www.javasoft.com/products/javaos/-javaos.white.html.

[29] Tanenbaum, A. S. Linux is obsolete. Message sent to comp.os.minix newsgroup, Jan. 1992.

[30] Tanenbaum, A. S. *Modern operating systems*, 2nd ed. Prentice-Hall, 2001.

[31] TIS comitee. *Tool Interface Standard Portable Formats Specification*, Oct. 1993.

[32] Tullmann, P. The Alta operating system. Master's thesis, University of Utah, 1999.

[33] WikiPedia, the free encyclopedia. Online. http://en.wikipedia.org/wiki/Main_Page.

[34] Wright, G. M. *A single-chip multiprocessor architecture with hardware thread support*. PhD thesis, University of Manchester, Jan. 2001.

Acronyms

Definitions for most acronyms were taken from the WikiPedia project [33].

BSD Berkeley Software Distribution An open source reimplementation of the Unix operating system from the University of California at Berkeley.

GC Garbage Collection. A system of automatic memory management which seeks to reclaim memory used by objects which will never be referenced in the future

GNU GNU is Not Unix. Open source project that was launched in 1983 by Richard Stallman with the goal of creating a complete free operating system.

GPL General Public Licence. The most common licence accompanying open-source projects. Grants the user the right to distribute, use, copy and modify the source code of a program, provided that the modifications are released under the same licence.

IPC InterProcess Communication. The exchange of data between one process and another, either within the same computer or over a network.

IRQ Interrupt ReQuest. Interrupts used by peripherals as a way to bring the processor into attention.

I/O Input/Output. Generic term used to describe the mechanisms of communication between computer software and hardware.

JAR Java ARchive. A gzip-compatible file format that is used to store compiled Java classes and associated metadata that constitute a program.

JikesRVM Jikes Research Virtual Machine. A JVM implemented by IBM as a research program and released under an open source licence [2, 26].

JNI Java Native Interface A programming framework that allows Java code running in the Java virtual machine to call and be called by programs specific to a hardware and operating system platform.

JVM Java Virtual Machine. A VM that executes Java bytecodes.

OS Operating System. A basic set of programs that communicate with the computer harware and share resources in order to enable user programs to run.

POSIX Portable Operating System Interface for uniX. Attempt to provide a standardised programming interface for Unix environments.

RPC Remote Procedure Call. A remote procedure call is a method that allows a computer program running on one host to cause code to be executed on the same host or another host, using the network as a transport medium, without the programmer needing to explicitly code for this.

RMI Remote Method Invocation. Allows an object running in one JVM to invoke methods on another object running in another JVM. Provides for remote communication between products written in the Java language.

VM Virtual Machine. An environment between the computer platform and the end user which allows the execution of programs not designed for the current architecture. In the text, used interchangably with JVM.

Appendix A

A sample run output

```
Multiboot flags = 0x7ef
cmdline =
mods_count = 1, mods_addr = 0x284a0
 RVMmodule: mod_start = 0x11f000, mod_end = 0xda9494
            mod_size=12841kb cmdline=@
Kernel end: 0xdaa000
Memory map provided by grub
 base_addr=0x0, length_low=0x9fc00, type = 0x1
 base_addr=0x100000, length_low=0x7f00000, type = 0x1
FreeMem: start=0xdaa000 end=0x8400000 size=121176kb usable pages=2474
   Processor id = 0x8a1e20
Booting
Setting up current VM_Processor
Doing thread initialization
Setting up write barrier
Setting up memory manager: bootrecord = 0x11f1c0
mmap succeeded at chunk 19  0x01300000 with len = 1048576
mmap succeeded at chunk 20  0x01400000 with len = 1048576
mmap succeeded at chunk 21  0x01500000 with len = 1048576
mmap succeeded at chunk 22  0x01600000 with len = 1048576
mmap succeeded at chunk 23  0x01700000 with len = 1048576
mmap succeeded at chunk 103  0x06700000 with len = 1048576
Stage one of booting VM_Time
SYS:sysGetTimeOfDay
```

```
Initializing baseline compiler options to defaults
Creating class objects for static synchronized methods
mmap succeeded at chunk 35  0x02300000 with len = 1048576
Fetching command-line arguments
SYS:sysArg argno=-1 buf=0x671f20c buflen=512
Early stage processing of command line
Collector processing rest of boot options
Initializing class loader
Stage two of booting VM_Time
SYS:sysGetTimeOfDay
Running various class initializers
[Loaded [Ljava/util/Hashtable$HashEntry;]
[Loaded superclasses of [Ljava/util/Hashtable$HashEntry;]
[Initializing java.security.ProtectionDomain]
[Initialized java.security.ProtectionDomain]
[Loaded [Ljava/util/Locale;]
[Loaded superclasses of [Ljava/util/Locale;]
Booting VM_Lock
Booting scheduler
mmap succeeded at chunk 51  0x03300000 with len = 1048576
SYS:sysVirtualProcessorEnableTimeSlicing: timeSlice=10
SYS:setTimeSlicer timerDelay=10
Running late class initializers
[Loaded [Ljava/lang/System;]
[Loaded superclasses of [Ljava/lang/System;]
VM is now fully booted
Initializing runtime compiler
Late stage processing of command line
[VM booted]
SYS: sysCPUID id=0x6752254
Extracting name of class to execute
vm: Please specify a class to execute.
vm:   You can invoke the VM with the "-help" flag for usage information.
SYS:Exit 100
Halted.
```

Appendix B

Creating a boot disk image

Creating a boot disk image and istalling a boot loader on it is not a trivial task; Most boot images for open source operating systems that can be found on the Internet are snapshots of a hard drive on which the operating system has been installed. The following method can be used to create a boot image using only standard command line tools on the Linux operating system. Assuming that the GRUB bootloader is installed to `/boot/grub`, and the user knows how to use an i386 emulator, the required steps are the following:

1. Decide the size of the disk to be created. A size that is equal to a power of 2 is preferable because calculations tend to be easier. A disk size of 16MB ($= 2^{24}$) is used throughout as an example throughout the described process.

2. Create an empty file with the exact size:
   `dd if=/dev/zero of=disk.img bs=1k count=16k`

3. Setup a loop device for the created file (root access required):
   `losetup /dev/loop0 disk.img`

4. Call `fdisk` on the loop device:
   `fdisk -C 64 -H 16 -S 32 /dev/loop0`
   Arbitrary values can be used for the `C/H/S` switches as long as `C*H*S*512` equals to the exact size of the disk. At the `fdisk` command line (in paranthesises the required keystrokes), create a new(`n`), primary(`p`) first(`1`) partition, set its type(`t`) to FAT16<32M (`4`) or to FAT16(`6`) and activate it(`a`,`1`).

5. Delete the current loop device (`losetup -d /dev/loop0`) and recreate it with an offset matching the size of the disk master file table (MFT). The

offset can be calculated as:
`offset = blocks/sector * sectors to start of partition * 512`
where `blocks/sector` is the S parameter specified to `fdisk` and `sectors to start of partition` equals to 1. The command to create the loop device is:
`losetup -o 32256 /dev/loop0 disk.img`

6. Create a DOS filesystem on the loop device
   `mkdosfs /dev/loop0`
   and mount it to an empty directory
   `mount -t vfat /dev/loop0 /mnt/scratch`

7. Copy the required GRUB files on the disk image and unmount it:
   `mkdir -p /mnt/scratch/boot/grub`
   `cp /boot/grub/stage* /mnt/scratch/boot/grub`
   `umount /mnt/scratch`

8. Prepare a bootable GRUB floppy:
   `dd if=/boot/grub/stage1 of=floppy`
   `dd if=/boot/grub/stage1 of=floppy seek=1`

9. Setup the emulator to boot from the floppy disk image and use the hard disk image as it hard disk. Then start the emulator and the GRUB command line should appear. On the command line enter the following commands:
   `root (hd0,0)`
   `setup (hd0)`

The boot image is now ready. The user can use the `mtools` package to access the contents of the image e.g.
`mdir z:/` to list the files
after adding the following line to `~/.mtoolsrc`
`drive z:  file="/path/to/diskimage" partition=1`
Alternatively, the user could loop-mount the disk image with the offset option to `losetup` as discussed previously.