JN: An Operating System for an Embedded Java Network Computer

UCSC-CRL-96-29

Bruce R. Montague [†]
Computer Science Department
University of California, Santa Cruz
brucem@cse.ucsc.edu

9 December 1996

Abstract

The implementation of an embedded operating system explicitly designed to support the Java Virtual Machine on the National Semiconductor NS486 embedded PC is described. This system, the Java Nanokernel (JN), supports an Internet web server written in Java and a web Camera that can be controlled from a remote web browser. JN in principle can currently run any Java program that does not use a local display device. This paper is primarily a system overview and a summary of lessons learned.

keywords: Java, JavaOS, embedded systems, soft–real-time, OS implementation, kernel software architecture, JN, TCP/IP, KA9Q, NS486SXF.

1 Introduction

This paper reviews lessons learned from implementing a custom embedded operating system designed specifically to support the Java Virtual Machine (JVM) on a small 'single-chip' embedded PC attached to the Internet. JVM interprets Java, a multithreaded language that typically runs on top of a host operating system [Gos95]. We use the term JVM to refer to the Java Interpreter and its runtime, that is, the complete body of code needed to execute Java programs. The JVM is written entirely in C. Our current system, the Java Nanokernel (JN), supports a simple Web server written in Java and provides a Java interface to a color Connectix camera.

JN is of interest because it is, to the best of our knowledge, the first system designed and developed in an academic environment specifically to support Java. Indeed, it may be the first OS other than JavaSoft's JavaOS that has been custom designed to run the complete JVM.

[†] Supported in part by a gift from National Semiconductor.

The components of this system are shown in Figure 1. The Java Nanokernel runs on evaluation boards assembled by National Semiconductor for their NS486SXF part. National describes this chip as a *'32-bit 486-class controller with on-chip peripherals for embedded systems'* [Nat96].

Above JN an application interface (API) has been implemented which provides only: 1) those services that we found necessary to run the JVM as an application; 2) services necessary to port the KA9Q TCP/IP stack.

JavaSoft's JavaOS currently implements as much functionality as possible in interpreted Java [MKKS96]. Unlike JavaOS, JN, its drivers, and the JN TCP/IP stack are implemented in C. The JVM runs as an application thread. However, unlike Java hosted on Unix or Windows, the only functions included in JN are those required to run Java threads.

JVM was originally written using a multi-threading C runtime called *Greenthreads*. This runtime coordinates activities, such as I/O, between interpreted Java threads and the host system. A large part of porting Java consists of porting or reimplementing this Greenthreads functionality, which was not originally documented or specified. Our initial approach to determining this functionality was to link Java without Greenthreads, locate and inspect the JVM calls to the missing routines, and guess at the required functionality. This reverse engineering approach resulted in a system that almost worked and was not unduly tainted by the Greenthreads implementation. This process also identified a few services that did require inspecting the Greenthread source. We were thus able to implement a small OS providing the Greenthread functionality without ever becoming expert in the existing Greenthreads implementation.

Our determination of this API was a significant part of this effort; our document describing the resulting API served as the initial basis for the corresponding JavaSoft

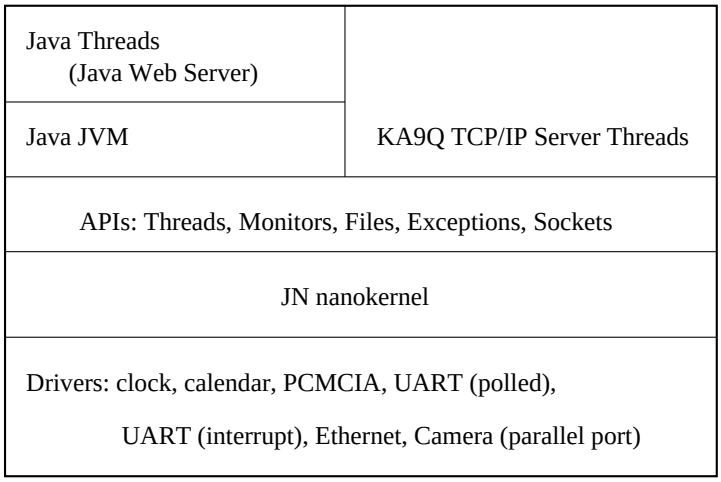

Figure 1: Java Nanokernel and Components

specification.

The current system consists of the JN nanokernel; a modified subset of the KA9Q TCP/IP stack; emulation API's providing the low-level system functionality required by both JVM and KA9Q; a simple utility suite (including the ability to format and load a simple filesystem onto a PCMCIA RAM disk); a modified JVM runtime; a Java environment with all classes needed to run both the Java compiler and compiled Java applications that do not use the Abstract Windowing Toolkit (AWT); a simple Java CLI (Command Line Interpreter, i.e., shell) which uses a VT100 terminal connected via the UART; a very simple Java editor and a simple Java Web server; a test suite (consisting of low-level JN tests, TCP tests, and Java tests); and drivers for the following: clock, calendar, PCMCIA controller, NS16550 serial-port UART, National Semiconductor AT/LANTIC Ethernet chip, and a bidirectional parallel port supporting a 640x480 24-bit Connectix Color QuickCam camera.

2 History

We did not set out to make a custom Java Network Computer. Our interests included obtaining code fragments of real system code to study alternative concurrent system programming techniques, *thin-OS* support for high-performance network servers, enhancing embedded computing research capability, and investigating Java. Because the embedded processor market is an important National Semiconductor market, National was interested in evaluating Java's effectiveness in embedded systems. Thus, it was decided to obtain the JVM and implement a small custom OS that would provide the minimal support required to run the JVM stand-alone. This would provide a platform to evaluate Java interpreter size and performance.

2.1 The NS CR32 Version

We initially obtained access to the Java sources from Sun, removed Greenthreads, and compiled the JVM using gcc under SunOS. The version of the JVM was a very early version, Java Developer's Kit (JDK) version 1.0.1, which came with warnings from Sun that it was not yet easy to port or understand.

Our initial target system was the National Semiconductor CR32A, a RISC-like CPU that is a new member of National's CompactRisc family [Nat95]. The 16-bit version of this CPU is currently used extensively to form custom cores for special embedded processors. The 32-bit CR32 is expected to be likewise used across all National product lines.

The initial JN kernel was designed and implemented using the CR32 toolset available from National, which included a gcc-derived C compiler, an assembler, a linker, and a set of conventional embedded programming support utilities. Because we did not have real hardware, JN initially ran under the CR32 simulator which is integrated with the CompactRisc Debugger.

After approximately a 2 month effort in the spring of 1996, we had an API specification, a nanokernel design, a version of the nanokernel implemented and running on the CR32 simulator, about 25 simple test programs running under the simulator, and a version of the JVM that would compile and link. It took about another month to get our first trivial Java program to run and produce output under the simulator.

We had to modify JVM to eliminate dynamic linking (all required code was linked in) and implemented a sim-

ple Java CLI (in Java) with required native methods. A fair amount of run-time interface code was required, and all JVM dependencies resulting from assuming a virtual memory environment eliminated. Changes were made to National's C run-time, Java source, and the nanokernel as needed. There was a small amount of assembler (on the order of 2 pages), which was primarily used for context switch, but the remainder of the system was implemented in C. The initial file system used a preallocated array in (simulated) RAM as its file storage space. Naturally, this file system was volatile.

An interesting problem encountered in this original implementation was that the single C switch statement which forms the core of the JVM interpreter was too large to be compiled by the CR32 C compiler. The Java interpreter loop is essentially a single C switch statement which switches on the current interpreted opcode to the code block corresponding to the instruction. The CR32 compiler, in attempting to reduce the size of the address tables associated with large switch statements, used 16-bit offsets in the statement's jump tables. The JVM switch statement, which contains a code case for every Java opcode, generated output machine code which exceeded 32K in size, resulting in a C switch statement that jumped backwards from the location of the switch statement!

Another problem, which was never solved, resulted from Java's need to support 64-bit integer operations. The gcc-derived compiler supporting the CR32 did not support 64-bit integers, so this problem was ignored.

2.2 The NS486 Port

2.2.1 Motivation

At this point the system had out-paced the availability of hardware. Running a simple Java `'hello, world'` program in the simulated CR32 environment was taking 6 to 8 hours. Although we had a CR32 CPU on a board intended for use with logic analyzers, we did not have peripheral hardware. To continue the pace of development over the summer of 1996, a decision was made to port to a National NS486SXF preliminary evaluation board.

2.2.2 The NS486SXF

The NS486SXF is intended to be a 'single-chip' 32-bit PC, that is, to reduce the motherboard part count to a minimum [Nat96]. Peripheral chip logic incorporated directly on-chip includes a PCMCIA (removable PC card) controller, a NS16550 UART serial port, an enhanced (bidirectional) parallel port, an LCD display controller, infrared serial input and output control, a real-time clock/calendar, a watchdog timer, programmable interval timers, 2 PICs (peripheral interrupt controllers), a serial high-speed synchronous bus interface (Microwire), a degree of power management, 29 CPU pins that can be used for arbitrary bidirectional I/O under software control, a DMA controller, a DRAM controller, and a bus interface unit. The bus interface unit can be used to control the standard PC ISA bus or the embedded system variant of the ISA bus, PC-104. The bus interface unit is programmable and the programmer has considerable control over the configuration of resources and the layout of NS486 I/O space. For instance, the programmer can specify the location of device control registers and the interrupt IRQ levels for many of the above on-chip controllers, unlike many current systems which require manually setting DIP switches on PC controllers or motherboards.

To make space for all these peripherals on the same silicon as the CPU, the NS486SXF has no floating point unit, does not have virtual memory or associated page tables (it does have segmentation, however), and does not support Virtual 8086 Mode or 16-bit Real Mode, that is, the NS486 cannot run 8086 code. This means that the NS486SXF cannot run DOS or a standard PC BIOS. The NS486 boots in flat 32-bit Protected Mode and is intended to be used for embedded control applications using a conventional commercial real-time OS. A number of popular real-time operating systems have been ported to the NS486.

2.2.3 The NS486 Port

In preparation for working with real NS486 hardware, a preliminary port of the CR32 system was made to 16-bit 8086 DOS using Borland Turbo-C and the TASM assembler. This was primarily done due to the availability of tools and to simply get started on an x86-affinity port. The system compiled, linked, and ran the nanokernel.

The nanokernel was next ported, as soon as an NT system was available, to flat 32-bit Microsoft C and the MASM assembler. This port went fairly smooth, and within about 3 weeks of starting x86-based work we had a version of JN and the tests running on the NS486. This system did not include JVM and thus could not run Java. This port was significantly expedited by a suite of National hardware test and 'demonstration' routines which effectively could be used as a simple BIOS.

The NS486 evaluation board was supported with a suite of free third-party x86 embedded system tools available for evaluation. This software was 16-bit DOS based (although the vendors had 32-bit versions of the software available). We used the SSI embedded system linker and remote debugger that was included in this kit for all subsequent NS486 development. Initially we compiled JN on the NT development system, transfered the resulting file to the DOS machine hosting the NS486, downloaded the file to the NS486, and executed JN.

Porting the JVM and Java code to Microsoft C went reasonably well until we realized that Microsoft C in flat 32-bit mode did not provide floating-point emulation. Java required support for both IEEE floating-point and 64-bit integers. Rather than use a resident floating-point instruction trap handler, we obtained the 32-bit Metaware High C/C++ compiler. This compiler was known to work with the SSI toolset, explicitly supported in-line floating point in flat 32-bit mode, and supported 64-bit integer operations. This compiler ran under DOS using the PharLap DOS extender. Unfortunately, this combination proved excruciatingly slow when running under NT, so we adopted DOS as the development environment for the remainder of the project. One unfortunate side effect of this decision was that all the sources of the JVM (and all .h include file references) had to be changed to 8-character DOS filenames.

All development was performed using DOS and 2 33MHz 486 machines (essentially 'surplus' machines). These machines were completely compatible with the free evaluation software. As an aside, we later had the opportunity to compare our download times with a high-end NT system using the 32-bit tools and found that its UART performance at 115K baud was on the order of 2-3 times as slow, leading to (for us) excruciating long download times.

Once we adopted this toolset, the JVM port went reasonably well. In around 2 weeks we had Java up and in another month we had it reasonably solid. The biggest single source change required to the JVM was explicitly initializing around a dozen global variables scattered throughout the source to zero, since JN does not pre-initialize such memory. Thus, by the middle of August we had a system that achieved our original goal of 'providing the Java prompt'. The simple CLI could be used to enter the name of a class stored on the RAM disk, which would then be loaded and interpreted by the JVM.

2.2.4 Adventures with TCP/IP

Since things seemed to be going well, it was decided to add a TCP/IP stack for Internet connectivity and demo the resulting system at an Embedded System trade show in the middle of September 1996. To this end we obtained another NS486 evaluation board and a $130 Linksys 8-port network hub. We connected the 2 DOS development machines and the 2 NS486 boards to this hub to form our own small intranet.

Kona (later named JavaOS) had become available from JavaSoft by this time [MKKS96]. As much of Kona as possible was written in Java, including device drivers and the TCP/IP stack. We started a side project to obtain and evaluate the Kona code, with the intent of using the TCP/IP stack written in Java on top of JN. However, the initial Kona TCP/IP stack was still very preliminary, and JavaSoft advised us to obtain the next release. The TCP/IP stack was clearly changing rapidly and so, it appeared, were details of the low-level hardware interface and driver model that we would have to emulate. The most serious problem was getting a device driver working for the National DP83905 AT/LANTIC PC-104 Ethernet controller we were using on the NS486 systems. No such driver existed in Kona. For all these reasons we abandoned the attempt to use or leverage Kona code.

At this point we wanted to get Java TCP/IP applications running in under a month. KA9Q is a shareware TCP/IP stack freely available to educational institutions. KA9Q was written by Phil Karn and originally used primarily for amateur packet radio; it has also been used as a TCP/IP stack for mobile laptop computers accessing the Internet via a digital cellular system [Wad92] [Kar93]. KA9Q is stable, has been in use since 1991, and has been widely used as a non-commercial TCP/IP stack. KA9Q is written in C and a number of variants (such as JNOS and TNOS) are in use.

The KA9Q source was downloaded and KA9Q executables installed on the development machines. KA9Q provides a single DOS executable task that contains a non-preemptive multi-threading kernel that runs on top of DOS. Inside the single DOS executable run a number of TCP/IP server threads. The resulting system includes a simple CLI (shell), standard utilities such as Telnet and FTP, and a simple API resembling UNIX sockets. This socket API can be used by KA9Q threads running in the DOS executable.

KA9Q has a simple non-preemptive voluntary-dispatching kernel based on `kwait()` and `ksignal()` primitives, with different runtime semantics than the preemptive JN provided the JVM. Another set of emulator API's was thus required to provide KA9Q look-alike APIs on top of JN, thus enabling KA9Q server threads to run as JN threads. This API also included support for the socket-like API defined by KA9Q, allowing JN threads written in C to use TCP/IP.

Porting a subset of KA9Q to JN closely resembled porting the JVM. Required code was identified and extracted from KA9Q. The API required to replace the KA9Q kernel was determined by linking without the KA9Q kernel. The functionality of the API routines was determined both by examining calls in the source code and by inspecting the KA9Q kernel implementation. The KA9Q kernel itself was not ported.

All KA9Q functionality was removed except support for the low-level `IP`, `ICMP`, `PING`, `ARP/RARP`, and `RIP` protocols; support for the `TCP` and `UDP` protocols; and support for the corresponding socket APIs (`connect()`, `send()`, `recv()`, etc). All built-in utilities (such as

`ftp`) were removed except for a minimal version of `ping`.

All initial development was performed on our own internal 4-node network, off-line from our operational network. The TCP `ping` program (used to verify network connection) worked about a week before the trade show, a simple 1-page web server written in C worked 4-5 days before the show, and the Java web server (which required adding a Java native method interface to the socket-like API functions) worked 2 days before show-floor setup started. The Java web server was on the Internet at the trade show in the middle of September in National's booth.

2.2.5 From Alpha to Beta

Although the system had performed the demo, it had been a dirty port, typical of *demoware*. The AT/LANTIC Ethernet controller's interrupt routine was being called (polled) by the clock interrupt, rather then generating interrupts itself, and there was no driver error handling. Only the code paths in KA9Q used by the Java web server were working or had even been attempted. The web server itself was a 100-line toy, replying to any TCP connection with a simple Web page that included a dump of NS486 and AT/LANTIC registers. Nonetheless, the KA9Q code had been added to the system in around 3 weeks.

As a follow on, an additional month and a half was spent enhancing TCP/IP support before it could be considered adequate. Most of this work went into the AT/LANTIC driver, experimenting with interrupt modes, handling error conditions, and the like. The final serious TCP/IP bug proved to be a bug in the original KA9Q version of `recv()`. In practice, KA9Q application threads use an internal call that accesses input data directly within network buffers and thus avoids a copy operation into user buffer space (as it is running on DOS, KA9Q provides no memory protection). It appears the Unix-compatible `recv()` call had never been used, as it discarded all received data after returning any portion of the data to the caller!

The original KA9Q port used JN versions of `kwait()` and `ksignal()` and protected a few key KA9Q data structures by using the JN call to disable the scheduler (this call is also used by JVM when performing garbage collection). KA9Q initialization had been replaced and an additional thread had been added offloading TCP input processing from the JN Ethernet driver. This thread was aware of KA9Q internal data structures and capable of allocating KA9Q managed memory. This approach worked essentially by accident, and because the system was not stressed.

Correctly supporting a non-preemptive event-driven multi-threaded system application, such as KA9Q, with the preemptive JN was mildly interesting. The approach adopted was to consider all TCP/IP code non-preemptive real-time code and require a `kstart()` API routine to be called upon entrance to any such code, with a corresponding `kstop()` routine called upon exit from such code. The `kstart()` routine indicates that, whenever the calling thread is selected by the scheduler, scheduling is disabled until the running thread explicitly blocks via `kwait()`. Calling `kstop()` causes the thread to revert to normal preemptive scheduling treatment. When a non-preemptive thread blocks, preemptive threads may run. Non-preemptive threads are selected based on priority as are all threads, but once selected non-preemptive threads run until they explicitly yield. In practice, the 5 TCP server threads call `kstart()` once and never call `kend()`, while `kstart()` and `kend()` are used to bracket TCP/IP API code called by applications.

Another month was spent working with an interrupt driven UART driver. This month (and the driver) was also a familiarization exercise for someone new to the project and driver programming. During this period significant scaffolding for debugging drivers was developed. A real-time event log was embellished and a crash dump routine added. The event log provides a means to trace system behavior over a reasonable period of time without unduly effecting system timing, while the crash dump provides a means of obtaining a formated system snapshot, including log contents, in a DOS file for offline analysis. The crash dump can be triggered by a Nonmaskable Interrupt (NMI), in which case the PC location reported in the crash dump can subsequently be examined using the SSI debugger. This is useful when the system has hung.

While getting the TCP/IP stack to work reliably, low-level native methods were developed to manipulate the Connectix camera attached to the NS486 parallel port. Java native methods were then defined to access these functions. Once the TCP/IP stack was working reliably, it was only a matter of a few days until the Connectix camera was generating images, under control of a Java server running on the NS486, and shipping the images to a Java camera control application running on a Unix workstation.

3 System Size

The size of the initial system, in total lines of code, is shown in Figure 2. The JN nanokernel size includes the nanokernel work-loop, initialization, and assembler support routines, such as the primary interrupt routines.

The JN test/debug code and the hardware test code is kept in the system image for convenience.

JN	Lines of Code
Nanokernel	2.5K
APIs	4.5K
Drivers	3K
Test/Debug	7K
TCP/IP	19K
Hardware Tests/Initialization	8K
TOTAL	45K

Figure 2: System Size - Lines of Code

4 System Software Architecture

JN is a soft–real-time kernel because the JVM API has no hard–real-time requirements. Therefore, JN uses a classic software architecture for a soft–real-time kernel. This architecture has informally been called a *Cutler kernel*, as it is the architecture used by the successful line of kernels implemented by teams lead by David Cutler, namely RSX-11, VMS, and NT [Cus93]. This basic kernel software architecture was used on earlier systems, including the kernel of IBM's massive mid-60's TSS OS effort [Com65]. The TSS system architect noted that the basic kernel model was adopted from IBM's TSM supervisor, which adopted the model from the Mercury Programming System developed to support the soft–real-time requirements of the project Mercury space program [Kin64] [SH61].

This kernel architecture can be considered a *Serially-Reusable Interrupt-Extension*. Some of its characteristics include:

- The kernel is not reentrant. It can be thought of as a single task that runs as an interrupt-enabled follow-on extension to all interrupt routines. The kernel runs from start to completion every invocation. Although the kernel can be interrupted, the kernel cannot block, and kernel CPU state is not saved across kernel invocations. Rather, the kernel always starts at a known code and stack location. The kernel uses a single kernel stack (on a uniprocessor).

- The kernel consists of a single work-loop driven by a queue of control blocks which form a real-time subroutine or co-routine dispatch scheduler. Each control block contains the address of a routine which the kernel must execute. Historically, these routines have been called *fork* routines in the Cutler kernels. Other common names for such routines, especially as found in I/O managers, include *Second Level Interrupt Handler* (SLIH) and *Deferred Procedure Call* (DPC). Since fork routines cannot block and must have a bounded execution time, they have many characteristics of routines written for a hard–real-time environment. Neither the fork routines or the kernel need to use explicit mutual exclusion to access global data structures (on a uniprocessor), reducing the need for explicit synchronization.

- The kernel is activated by an interrupt, either hardware or software. If an interrupt occurs when the kernel is not active, the kernel is started at the end of the interrupt routine, typically after the interrupt routine queues a control block designating a fork routine to handle the events triggered by the interrupt. If the kernel is active when an interrupt occurs, the interrupt routine simply queues a control block for follow-on processing and returns from the interrupt back into the kernel.

- After executing all routines corresponding to control blocks, the process scheduler is run if a global flag indicates that one of the routines has altered the scheduling status of a process. The selected process is then dispatched.

Although we adopted a well-known kernel architecture, a full kernel was not implemented. We refer to the implemented subset as a *nanokernel*. Although the term has been disparaged, we believe it is warranted [Lie96]. Although similar to a microkernel, a nanokernel has the following identifying characteristics:

- There exists only one mandatory nanokernel API routine: run a specified subroutine serialized in kernel context, that is, as a fork routine. There are no other API's defined by the nanokernel. The specified routine is required to meet a few minimal requirements. Any application thread can define and run such a routine. There is no concept of protection or security. Implementing such a routine defines a new API routine available to all threads.

  In the case of JN, the single *thrd_kcall()* API routine runs a specified C subroutine as a kernel fork routine. One argument, a void *, is passed to the fork routine when it is called by the nanokernel work-loop. This argument is often a pointer to a C structure containing arguments required by the routine.

  In practice, we write fork routines routinely as part of JN development. This leads to a style of system programming in which the API is custom extended for individual applications with the same degree of difficulty as defining new application subroutines.

- There is no general purpose I/O database or I/O manager. A microkernel usually has a general scheme for describing, locating, and managing devices and device drivers. This is useful for supporting general purpose I/O APIs that can be used with multiple device types. A nanokernel such as JN provides no device I/O management. There is no required driver model. Device drivers are written as collections of interrupt and fork routines. Although the kernel fork mechanism is used to coordinate events, there are no restrictions or requirements placed on a device driver. Each device driver is free to do whatever it needs – defining its own suspend queues, implementing its own data structures, etc..

- There is no generic name space support.

- There is no implication that a message-based IPC mechanism exists. Such mechanisms, intended to provide a means of offloading kernel functionality to user-level servers, are a hallmark of traditional microkernel architecture.

- Construction of interrupt vectors and required memory management tables (e.g., the GDT and IDT) is performed by the linker, not by API routines. There are no API routines for connecting to an interrupt vector or for constructing page table entries. The entire system image is constructed by the linker, with all required binary code linked into the single image. There is no loader capable of dynamically loading executable binary code.

- JN supports only light-weight threads. There are no heavyweight processes, no memory protection, no process specific page tables, and no concept of resources private to a thread (other than the mandatory thread stack and thread control block). There is also no concept of thread hierarchy, that is, threads have no parent-child relationship.

Although JN was not especially designed for small size, the resulting nanokernel is quite small, consisting of some 2000 lines of C code. A small kernel-only system can be built that does not include the Java JVM or the KA9Q TCP stack.

The JN file system is very simple. Although it appears to support a Unix-like directory structure, it is a flat file system supporting long filenames which can contain the '/' directory delimiter. Providing a hierarchical file system is necessary because the hierarchical class structure of a Java program must be directly expressed in the filenames constituting the compiled Java program; each compiled Java class must reside in its own `.class` file.

JN files are simply named queues that can reside in either volatile RAM or persistent RAM on a PCMCIA RAM card. An interesting feature of this file system is that Unix *sparse file* semantics are supported, that is, bytes internal to a file that have not been written need not be allocated.

The JN routines that manipulate thread context and establish initial stack frame contents are specific to the conventions of a single compiler, currently the Metaware C compiler.

4.1 The API

This section documents the API required by the JDK version 1.0.1 of the JVM for which we built JN. A more complete specification can be found in [Mon96].

The API is divided into 4 classes: threading, monitor, file, and exception.

4.1.1 Thread APIs

The threading API routines are conventional. Thread priorities range from 0 to 10, as defined by Java. Thread stacks are fixed-size and cannot dynamically expand.

A *sysThreadCreate()* API routine creates a thread in suspended state, *sysThreadExit()* can terminate any thread, and *sysThreadYield()* rotates the threads at the current priority level. Any thread can be removed from scheduling consideration by *sysThreadSuspend()* and restored to scheduling eligibility by *sysThreadResume()*. The unique integer thread ID of the caller can be obtained, the priority of any thread can be set and read, and the stack base and current stack pointer value obtained. This last

allows the JVM interpreter to check if a stack overflow is likely to occur.

The JN scheduler can be disabled by a call to *sysThreadSingle()* and later re-enabled by calling *sysThreadMulti()*. These calls are used to eliminate all thread-based concurrency when garbage collection is being performed on the single Java Heap. They can also be used if a single thread wants to eliminate any possibility of contention while in some time-critical section of code. In this case, preemptive context switching will not occur, although all API calls will still work as normal.

There is a 1:1 correspondence between low-level JN threads and high-level interpreted threads managed by the JVM. Associated with each low-level JN thread is an integer *cookie* field that can be accessed by *sysThreadSetBackPtr()* and *sysThreadGetBackPtr()*. These APIs are used by the JVM interpreter to link each high-level interpreted thread context to a low-level JN thread.

4.1.2 Monitor APIs

Java uses monitors as its fundamental concurrent programming mechanism. Monitors are exposed to Java programmers via *synchronized methods* or *synchronized objects*.

A monitor, as defined by Greenthreads and implemented in JN, can be considered a critical section associated with two semaphores. One semaphore guards external entrance into the critical section, and the other, which starts with a value of 0 (unavailable), guards internal access. To enter a critical section, a thread specifies the external semaphore in a *sysMonitorEnter()* API call. Only one thread can be inside a critical section at a time. Once inside the critical section, if the active thread must wait for some occurrence (an I/O completion or a change in the content of a data structure, etc.), it uses API routine *sysMonitorWait()* to put itself on the internal semaphore queue, while releasing its hold, atomically, on the external semaphore. Thus an external thread that was forced to wait can enter the critical section. Whenever a thread uses *sysMonitorExit()* to leave the critical section, as with any semaphore operation, another thread waiting on the external semaphore proceeds into the critical section.

Threads waiting on the internal semaphore are only reactivated by an explicit *sysMonitorNotify()* or *sysMonitorNotifyAll()* call. Essentially this is a semaphore V operation on the internal semaphore. An active routine that completes an activity upon which some thread may be internally waiting issues these API calls. These API routines simply move the thread waiting internally to the external wait queue. When each waiting thread moves to the head of the external wait queue, its execution resumes at the point inside the critical section where it waited for the needed resource or event.

A *sysMonitorNotify()* or *sysMonitorNotifyAll()* can only be performed by a thread that is inside the critical section. Race conditions in which *sysMonitorNotify()* is called before what should be the corresponding *sysMonitorWait()* can be avoided by correct programming. However, a newly activated thread that has become unblocked should always recheck the condition on which it was waiting. Such a thread cannot tell if it has activated due to a *sysMonitorNotify()*, a *sysMonitorNotifyAll()*, or the expiration of a timeout specified in the original *sysMonitorWait()*. In case of timeout or *sysNotifyAll()*, the desired resource may not be available when the thread actually resumes running in the critical section.

JVM monitor structures are stored outside of the kernel in user-space and thus are not, strictly speaking, secure from the perspective of the operating system. In JN, of course, no protection exists, so the issue is moot.

4.1.3 File APIs

The file API required by the JVM is a Unix subset. There are *open*, *close*, *read*, *write* and *lseek()* calls similar to their Unix counterparts. In addition, there is an *available()* call that can be used as an alternative to *lseek()* to determine the number of bytes between the current file position and the end of the file.

4.1.4 Exception APIs

The exception API routines provide support for per-task software interrupt handlers, that is, *signals*. The signal set is basically a subset of that provided by Unix. A software interrupt handler can be specified for a particular software interrupt, and the handler can be removed. The reception of software interrupts can be enabled and disabled by the *intrLock* and *intrUnlock* APIs, and a thread can determine if any software interrupts are pending.

Although these facilities are not heavily used, they considerably complicate the nanokernel. Each thread actually maintains 2 contexts, one the normal non-interrupt context, and the other the software interrupt context. When a thread is dispatched, if its interrupt-level context is valid, that context is always executed in preference to the normal context.

If a thread is executing normally and a software interrupt is queued to the thread, the nanokernel saves the normal-level context, decrements the stack by a small pad, forms a pseudo-call frame for the designated interrupt handler, and dispatches the software interrupt context. The software interrupt handler thus runs with full ability to perform all API calls as if it was executing normally, in fact, the interrupt handler code is externally indistinguishable from normal code. Upon completion of

the software interrupt routine, interrupted normal-level execution continues.

5 Lessons Learned

5.1 Traditional OS Engineering Lessons Revisited

- *Drivers*. The drivers took longer to develop than the kernel, especially the AT/LANTIC (Ethernet) driver. This was largely because the kernel was a deterministic program that could be designed before programming began, while the drivers, to a degree, were initially 'exploratory' tools used to determine actual hardware operation by trial-and-error. This illustrates the importance of good data-sheets targeted at the device driver programmer, in addition to conventional data-sheets written for the hardware designer.

  Driver development would be greatly facilitated by including complete stand-alone code examples in data-sheets. In our case, such information would have benefited system development more than any software engineering technique of which we are aware, especially in the case of low-end PC devices, which often trade low cost for driver complexity. Data-sheets currently tend to informally mix English, flow-charts, and code fragments.

- *Kernel architecture.* The serially-reusable interrupt-extension kernel architecture (the *Cutler kernel*) works well. This architecture is well understood and appropriate for a kernel such as JN. How such an architecture compares with a reentrant semaphore-based kernel architecture remains a question of interest.

  Because we used a traditional software architecture, writing the kernel, seemingly among the harder things we attempted, proved to be one of the easier aspects of the project. However, application of this traditional architecture was greatly facilitated by implementation experience; this was the eighth OS implementation in which the principle implementor was involved, and the fourth as principal architect.

- *Redundancy due to multiple software architectures*. Difficulties merging code from different software architectures have recently been discussed within the software reuse community [GAO95]. In our case, we experienced redundant code problems such as the following:

  There are currently 3 memory managers: 1) JN main memory and dynamic pool allocation (*malloc()*); 2) the JVM garbage collected heap; and 3) KA9Q memory allocation, which provides a garbage collected cache reflecting its use for, among other things, best-effort reassembly of TCP segments.

  As another example, redundant synchronization mechanisms and corresponding API routines exist. To support Java, JN provides Java-style monitor support (*Enter*, *Exit*, *Wait*, *Notify*, and *NotifyAll*), an explicit *Suspend* and *Resume*, and the ability to define a critical region by explicitly disabling and re-enabling the scheduler (*ThreadSingle* and *ThreadMulti*). To support KA9Q we added the *kstart*, *ksignal*, *kwait*, and *kend*. Additional mechanisms used include the kernel fork mechanism and routines to define critical sections by disabling interrupts. In practice, we have 2 pairs of these hardware interrupt masking routines, one set of which is a cover function for the corresponding machine instructions, and the other set which nests, that is, only re-enables interrupts when all disables have been matched by a corresponding enable. And of course, strictly speaking, one must consider the interaction of all of these methods with software interrupts and the *Lock* and *Unlock* calls...

  It would be difficult to get rid of any of these routines without significant source modifications and commensurate re-testing. However, the API's involved in thread synchronization have clearly become distressingly redundant, and the idea of combining all of these into a single mechanism all the more appealing. Additionally, these redundant mechanisms clearly contribute to total system size.

- *Software Reuse is Difficult*. The KA9Q source was frozen, stable, and of high quality. In addition, National had provided us with source for 2 drivers to the Ethernet device used with the NS486 boards. Yet more effort was spent establishing a robust TCP/IP stack then any other single aspect of the system, including implementing the kernel and porting the JVM. This was due to 1) the size of the TCP/IP stack; 2) the software architecture mismatch, both with respect to KA9Q and the National drivers; and 3) assumptions regarding the quality of the reused source. Not all source deserves to be reused, and this was especially true in regard to one of the drivers, but this was not obvious from a simple preliminary examination of the source.

- *Situational awareness*, that is, *build plenty of scaffolding*. Seeing what you are doing makes a tremendous difference in system programming. A line analyzer (datascope) was extremely helpful in writing the UART driver. Likewise, after we had TCP/IP

running, access to a Network General network analyzer proved a useful tuning and debugging tool.

For debugging complex concurrent system problems, event logs and crash dumps are both nearly essential. Use of the SSI remote interactive debugger was very useful for solving a few key problems (for instance, when porting the TCP/IP assembler checksum code from 16-bit to 32-bit code and neglecting to clear the high 16 bits of the accumulator), but in general interactive debugging was not used, as many problems occured in heavily used code paths with timing dependencies. Code that did not have timing dependencies tended to be debugged using a polling version of `printf()` on a debug serial port (such output was both displayed on the monitor of the host DOS machine and captured in a DOS file). If we could add a feature to the SSI debugger, it would have been data watchpoints, that is, the ability to set a breakpoint that would occur when a particular memory location changed value.

To support the crash dump and on-line debugging, format routines exist for all data structures, and these format routines are kept linked into the kernel.

- *Software Estimation.* Software estimation by analogy has been claimed to be as accurate as any other technique for estimating development time and cost [SSK96]. With respect to estimation by analogy, 2 months seems a typical development time for a kernel such as JN, recognizing that driver development, not the kernel itself, is likely to require considerably more time. This reemphasizes a point Bill Wulf made in his description of *CM-star*, which was that utility, driver, and application development was more important to the success of the system than the operating system itself [Wul81].

- *Avoid using demo schedules to drive development*. For software above a certain size, demo driven development leads to fragile non-engineering solutions (*demoware*). The difference between a system being 'up', doing 1 thing, and the system being complete needs constant emphasis. Some 20 years ago Brooks made a famous observation that a factor of 9 exists between the effort required to write a program for self-use versus the same program as a product [Bro75]. The difference between demoware and a robust engineered product seems at least this large, if not greater.

- *Engineering requires testing*. JN development would have been extremely different without a test harness. Around 100 tests exist, including Java and TCP tests. All tests are always linked in and available. After a new kernel rebuild, approximately 50 of these tests are always immediately rerun. This test suite exercises each API call.

  This is an old lesson, but bears repeating since students typically do not find test code of interest. Perhaps it would be worth adopting a pedagogic attitude that code that is written without corresponding test code is not ever finished code.

- *Reading becomes more important than writing*. Software engineering in a project such as JN, in which large existing programs are integrated, requires skill at reading large bodies of code. However, student training tends to emphasize writing code, rather than reading code.

- *Assembler programming is still required.* Even though total assembler line count is small, understanding and getting this code correct is of critical importance and often the cause of disproportionate programming time.

5.2 JAVA

We have no final results on the applicability of Java to embedded systems. Clearly, a complete Java system with TCP/IP support is not as small as a traditional embedded system using Forth, Basic, or assembler.

It required considerable effort to get Java programs running on an embedded system. However, the effort was not large compared to many industry projects. We found it necessary to modify the JVM runtime, but this did not prove overly burdensome. The portability of the JVM based on JDK 1.0.1 could, however, be significantly enhanced. We have not attempted to keep up with the rapidly changing Java source code. We continue to use the original source derived from version 1.0.1 of the JDK.

We have not attempted to optimize the JVM for real-time or embedded systems, that is, we are using Java 'as is'.

It is not clear at this time that Java is a portable soft–real-time system. Clearly, Java code on different platforms has different threading behaviors, since scheduling behavior is not specified as part of the language. This resembles the somewhat unfortunate ADA experience. Multithreading Java code currently cannot be expected to have the same behavior on Unix, MacOS, Windows, and JN. A working program on one platform may deadlock on another.

6 Conclusion

A working Java network computer with a custom Java operating system has been implemented for a 'single-chip'

32-bit PC, the NS486SXF. This system is used as a web server and a web camera. The design and implementation of this system somewhat resembled a typical industry implementation rather than a research project, largely because the requirements were predefined by the JVM. This system is reasonably robust and can serve as a testbed for future work. The JN kernel has now been stable with few changes for a number of months.

We developed a Java API supporting the JVM and its required concurrent programming runtime without requiring any other system software, and can use this environment to run Java applications that do not use the AWT window class.

We have shown it is possible to do TCP/IP network development and related research and education for very low cost.

The serially-reusable interrupt-extension kernel architecture is well understood and works well for a kernel such as JN. However, driver development remains a difficult and time consuming task. When drivers and utilities are considered, developing and maintaining even a modest system such as JN approaches the limit of what can reasonably be developed in a typical university research environment.

Acknowledgments: I would like to express my gratitude for continuously support of this work to Professor Charlie McDowell of UCSC and Bijoy Chatterjee of National Semiconductor. In addition, special thanks are owed Elizabeth Baldwin and Mike Allen for much hard implementation work.

References

[Bro75] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[Com65] Webb T. Comfort. A computing system design for user service. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 619–626. Spartan Books, 1965.

[Cus93] Helen Custer. *Inside Windows NT*. Microsoft Press, 1993.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 17(6):17–26, November 1995.

[Gos95] James Gosling. Java intermediate bytecodes. *SIGPLAN Notices*, 30(3):111–118, March 1995.

[Kar93] Phil Karn. The Qualcomm CDMA digital cellular system. In *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*, pages 35–39. USENIX Association, 1993.

[Kin64] H.A. Kinslow. The time-sharing monitor system. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 443–454. Spartan Books, 1964.

[Lie96] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[MKKS96] Peter Madany, Susan Keohan, Douglas Kramer, and Tom Saulpaugh. JavaOS: A standalone Java environment. 'http:// java. sun. com/ products/ javaos/ javaos. white. html', December 1996.

[Mon96] Bruce R. Montague. The API of the UCSC Java Nanokernel (JN). Computer Science Technical Report UCSC–CRL–96–28, UCSC, December 1996.

[Nat95] National Semiconductor. *Preliminary CR32A Core Architecture Specification Revision 1.0*, October 1995.

[Nat96] National Semiconductor. *NS486SXF Optimized 32-bit 486-class Controller With On-Chip Peripherals for Embedded Systems*, March 1996.

[SH61] Marilyn B. Scott and Robert Hoffman. The Mercury programming system. In *Computers – Key to Total Systems Control: Proceedings of the Eastern Joint Computer Conference (AFIPS)*, pages 47–53. Spartan Books, 1961. Part B of Project Mercury Real-Time Computational and Data-Flow System.

[SSK96] Martin Shepperd, Chris Schofield, and Barbara Kitchenham. Effort estimation using analogy. In *Proceedings of the 18th International Conference on Software Engineering*, pages 170–178. IEEE Computer Society Press, March 1996.

[Wad92] Ian Wade. *NOSintro: TCP/IP Over Packet Radio; An introduction to the KA9Q Network Operating System*. Dowermain, 1992. Dustjacket, annotated table of contents, and related links at 'http:// www. netro. co. uk/ nosintro. html'.

[Wul81] William Allan Wulf. *HYDRA/C.mmp, an Experimental Computer System*. McGraw-Hill, 1981.