By Marius Gafen,
Executive Vice President of Marketing,
NSIcom Ltd.

Alternative Java Threading Designs for Real-Time Environments

When using Java in embedded real-time applications, software developers must consider the ways that Java Virtual Machines (JVMs) handle Java thread mapping. Thread mapping affects JVM robustness, maintainability, and system resource utilization including memory requirements. The industry's three leading Java thread mapping techniques – task-per-thread, encapsulation, and thread-to-thread – each have very different impacts on application development. Encapsulation technology, engineered from inception for embedded environments, demonstrates excellent performance in the embedded environment: it simplifies many aspects of integrating a JVM into an embedded system, reduces memory requirements, and isolates applications in native code from unintended influence by Java application code.

Successful integration of Java Virtual Machines (JVMs) with native applications in embedded systems requires skillful handling of Java threads. Embedded system designers face a number of issues when selecting a JVM. One of the most important of these involves the JVM's technique for mapping Java threads to Real-Time Operating System (RTOS) tasks. Thread mapping affects thread semantics, memory allocation policies, garbage collection behavior, and the management of the C stacks associated with threads.

Today, different Java Virtual Machines offer application developers three varying approaches to Java thread mapping: task-per-thread, thread encapsulation, and thread-to-thread. These approaches have trade-offs and advantages that programmers are well advised to consider when planning to incorporate Java into embedded systems.

The task-per-thread model (see Figure 1), as found in several JVM implementations, maps a Java thread to an RTOS task, so that there is one RTOS task for each Java thread. Under this approach, no separate Java thread scheduler exists, so the RTOS treats tasks and threads as equivalent, and retains responsibility for context switching between Java threads, user tasks, system tasks, and interrupt service routines. Additionally, the RTOS must manage the C stacks of Java threads as well as the stacks of native tasks.

Relying upon the RTOS scheduler to perform Java thread scheduling may pose some problems. Can Java thread semantics and locking semantics be maintained, for example? Are there any unwanted interactions between Java threads and RTOS tasks, or between locks on Java objects and RTOS locks?

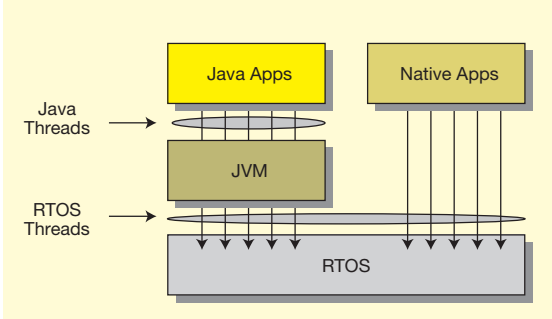

Figure 1. The task-per-thread mapping model used by some Java Virtual Machines

Another area of concern arises from the fact that RTOS tasks generally have different semantics than Java threads. This means the JVM designer must ensure that Java thread semantics are fully and faithfully supported by the underlying RTOS.

The thread encapsulation model (see Figure 2), as found in NSIcom's JSCP (Software Co-Processor for Java), eliminates a number of potential problems associated with the task-per-thread approach. By placing the entire JVM, and all Java threads, into a single RTOS task, thread encapsulation makes all Java threads invisible to the RTOS. Some UNIX operating systems that support threads also use this approach wherein Java threads are encapsulated within a UNIX process.

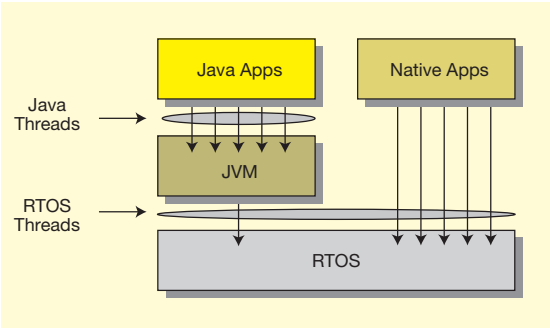

Figure 2. The encapsulation technique used in NSIcom's Java Virtual Machine

Encapsulation methodology frees the RTOS from Java thread scheduling, as well as the need to manage the C stacks of Java threads. The two-level Java thread/RTOS task model considerably simplifies the management of Java threads. For instance, the schedulers are independent of each other and can be tuned for their particular operations: In JSCP, the Java thread scheduler is not interrupt-driven, and so it need not be reentrant, whereas the RTOS scheduler must be interruptible and reentrant. When Java threads are

independent of the RTOS, and thus from RTOS task primitives, Java thread behavior is consistent under different RTOSs.

In the third methodology, the thread-to-thread technique (see Figure 3), as used by JVMs running under Microsoft Windows NT, each Java thread is mapped to an operating system (OS) thread. These threads, in turn, are contained within a process – a task that does not share the address space of other tasks. The OS schedules threads irrespective of the process with which they are associated. Like the task-per-thread model, the thread-to-thread approach makes it necessary to match Java thread semantics with OS primitives.

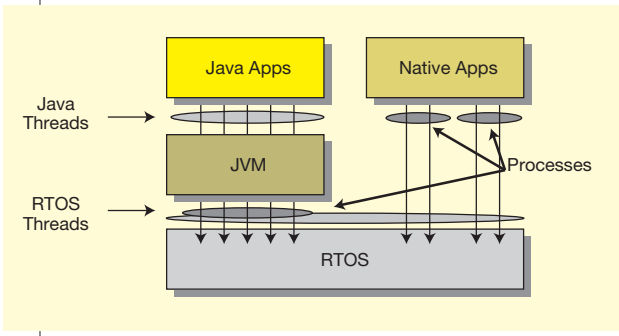

Figure 3. The thread-to-thread mapping approach

HANDLING C STACKS AND JAVA STACKS

No matter which threading model is used, every Java thread has a C stack and a Java stack. The Java stack contains the execution frames of Java methods. These Java methods invoke C code, directly or indirectly. Java calls C code directly via native methods, and indirectly through the interpreter. Whatever is running, C code is ultimately executed, and the thread's C stack is used.

Java stacks are handled in the same way in all implementations. A buffer large enough to contain 500 stack items (about 2 KB) is initially allocated, but unlike the C stack, may be increased as needed.

The Java stack is allocated from this buffer and the buffer may be extended. If the buffer becomes full when a new method is executed, a new buffer is allocated and linked to the existing buffer. This processing occurs in the functions invokeJavaMethod and variants. These buffers are allocated from the non-garbage-collected heap.

DIFFERENT WAYS TO MANAGE C STACKS

Depending upon the underlying threading methodology, different JVMs handle C stacks differently. C stacks contain several kinds of frames: Such functions as `ExecuteJava`, `FindClassFromClass`, and `VerifyClass` may mutually call themselves recursively. Each `ExecuteJava` frame represents the execution of a single Java method by a single thread. Recursive functions may also call non-recursive functions. Frames may also have functions that are the implementations of native methods.

All JVMs check for actual or potential C stack overflow in various places within the code. When such an overflow occurs, these JVMs throw a `StackOverflowException` and abort the thread. JSCP, however, allocates a new page for the C stack and then allows the application to continue.

JVMs do, however, vary significantly in the way they implement C stacks. For example, in Sun's PersonalJava™, which uses the task-per-thread model, C stacks are initialized by default to 128 KB. (A different value may be specified on the invocation command line). C stacks cannot be grown, so they must be large enough to accommodate the maximum memory usage of any thread. To protect against stack overflow, `ExecuteJava` and related functions check the current stack depth. If they detect actual or potential stack overflow, they abort the thread and throw a `StackOverflowException`.

By contrast, the JSCP JVM, based upon encapsulation technology, allows C stacks to be sized dynamically. The initial size begins as one page (16 KB) of memory. In the same places where the JVM in the common model checks for stack overflow, the JSCP VM also checks for stack overflow. Whenever the JSCP VM detects a potential stack overflow, instead of aborting the thread, it allocates another page of memory, links this new page to the existing C stack, and then allows processing to continue. These new pages are allocated from the non-garbage-collected heap.

SCHEDULING JAVA THREADS UNDER ENCAPSULATED JVM ARCHITECTURE

Since the thread scheduler in the JSCP JVM encapsulation environment operates independently of the RTOS task scheduler, it can possess properties that an RTOS scheduler cannot have. For instance, the JSCP scheduler is not reentrant, which greatly simplifies its design. A simpler design is more robust, reliable and efficient. In addition, the JSCP scheduler drives the various mechanisms that are responsible for the real-time portions of the Java application.

Under JSCP's encapsulation technology, its Java thread scheduler (SCHD) is a C function that runs as part of an RTOS task. SCHD's two main purposes are to schedule Java threads by priority and to respond to events generated by external devices. SCHD faithfully and fully implements all of the thread scheduling properties specified in *The Java Virtual Machine Specification*. However, unlike some other Java thread schedulers, JSCP schedules threads on the basis of time slices, in a non-interruptible fashion.

In operation, JSCP activates the scheduler when a timer expires and causes the preemption flag to be set. This user-settable timer is typically given a value between 20 and 50 milliseconds. JSCP checks to see if the preemption flag is set in any of several key places, including several places within the garbage collector. If the preemption flag is set, JSCP calls the function `btPpreempt` that causes SCHD to run. SCHD then chooses the highest priority runnable thread, and passes control to it.

ADVANTAGES OF JVM ENCAPSULATION TECHNOLOGY

Encapsulation technology offers a number of interesting advantages for embedded systems, including reduced memory consumption and unlimited lightweight threads that are safe to stop or suspend. JSCP's dynamic C stacks permit significantly smaller memory usage than task-per-thread mapping methodologies. JSCP dynamically expands the amount of memory allocated to a thread and then reclaims it when it is no longer needed.

Reducing Memory Usage

One user application initially developed without JSCP, involving a policy-based network management system with 50 policies, each needing two threads, was run repeatedly with fixed C stacks for the Java threads. On each run, the size of the fixed C stack was reduced until a stack overflow occurred. This technique revealed that a minimum of 36 KB per thread was needed. Further analysis revealed that the maximum stack size was reached during initialization of a policy. Therefore, the total amount of memory needed would have been 3.6 MB when a fixed C stack size was used. Under JSCP, each thread was allocated three pages (48 KB) at startup, and after the threads had finished initialization, the C stack was cut back to 16 KB, thereby saving the user 2 MB of memory.

Benefits of Lightweight Threads

JSCP threads bear the same relationship to RTOS tasks that threads on many desktop operating systems bear to processes. For example, user threads in Solaris are not visible to the underlying operating system scheduler, just as JSCP threads are not visible to the scheduler of the underlying RTOS.

The motivation for using threads this way in JSCP parallels the rationale that originally drove operating system designers to develop threads. For one thing, thread scheduling is much faster than process scheduling because a thread context is significantly smaller than a process context. Additionally, threads share the same address space, so caches do not need to be flushed when a thread context switch occurs.

Overall, the advantage of JSCP's lightweight threads is that are less costly in terms of CPU cycles and memory usage than the way threads are handled in task-per-thread methodology.

No Limit to the Number of Java Threads with JSCP

Some real-time operating systems constrain the number of tasks that can be created due to limitations in the number of control blocks they allocate. This would limit the number of Java threads that could be created if each thread were mapped to a task. Since Java is naturally thread intensive, it is not unusual for applications to create large numbers of threads. Standard Java class libraries, for example, implicitly allocate many threads, as when an HTTP connection is created or a request is made to load an image. Because JSCP's threading operates independently of the underlying RTOS, JSCP does not suffer from this limitation.

Safely Killing/Suspending Threads Versus Ownership of System Resources

Because system resources may be allocated to RTOS tasks, if Java threads are mapped to tasks, these threads may also own system resources. This makes stopping or suspending these threads an unsafe operation. Although it might be possible to use a flag to tell a thread to kill itself gracefully after deallocating all system resources it owns, it is not possible for a thread to know of the resources consumed by class libraries in its behalf.

Threads in JSCP do not allocate system resources. All resources external to the JSCP Virtual Machine (VM) are owned by JSCP itself. No Java thread owns the resource directly and thus it never can be in an RTOS critical section. The JSCP VM is free to suspend or kill Java threads without any side effects.

When the interpreter loop is running, instead of a timer, JSCP counts the number of bytecodes that have been executed. First, the system guesses an initial value for the number of bytecodes that can be executed in the user-specified time-slice period. This value is used in a counter that is decremented within the interpreter's main loop. When the counter reaches zero, the preemption function is called. In addition, the other scheduling functions are performed, a check is made to see how much time has elapsed, and an adjustment is made to the new initial value of the bytecode counter that better approximates the desired time-slice interval. The bytecode count converges rapidly to a good approximation of the time-slice interval.

The JSCP thread scheduler (SCHD) can call a user-specified idle function to support a device's power saving function. SCHD calls an idle function whenever there are no runnable threads. The user-defined idle function then blocks for a time specified by SCHD or until it detects an external event. When either of the above occurs, the idle function returns to SCHD.

CONCLUSION

Thread mapping methodology plays an important role in the use of Java in embedded systems. The encapsulation technology in JSCP™, an enhanced Java™ Virtual Machine (JVM), has been specifically engineered for embedded applications. JSCP's encapsulated design, its memory management system and its native interface techniques alleviate many concerns about embedded Java. With JSCP, a complete Java environment can be added to an existing embedded application without compromising that application.

Marius Gafen is responsible for all marketing activities at NSIcom. He previously held senior marketing positions with telecommunications provider Arel and held management and R&D positions with the Israel Ministry of Defense for telecommunications and software projects.
Mr. Gafen holds a Bachelor of Science Degree in Electrical Engineering from the Technical Institute of Israel in Haifa and is a marketing graduate of Tel-Aviv University.

By Jan Baan,
INtime Application Consultant,
RadiSys BV

Distributed Real Time Computing with Windows NT

Windows NT by itself may not be the optimal solution for real-time computing, but there are various options to extend the operating system. One of these, RadiSys Corporation's INtime real-time extension for Windows NT has been enhanced to provide distributed options, resulting in a highly scaleable feature set. This article discusses why the feature set has been extended and how it effects scalability.

WHY DOES WINDOWS NT NEED A REAL-TIME EXTENSION?

Many researchers have investigated the real-time capabilities of Microsoft's standard Windows NT operating system (at the time this article was written version 4.0 with SP4 was the current version). All have arrived at the same conclusion: although Windows NT may be able to provide a timely response to events, the system architecture can never guarantee fully deterministic behaviour [1].

One reason for the lack of guaranteed deterministic behaviour has to do with the way Windows NT handles interrupts. Windows NT interrupts cannot be controlled by an application; instead, the Windows NT kernel uses a FIFO mechanism to deal with interrupt requests. Each interrupt request (IRQ) results in a call to a small Interrupt Service Routine (ISR), which has all interrupts disabled; this ISR generates an NT kernel request to perform a Deferred Procedure Call (DPC), which is placed into a FIFO queue of all outstanding DPC requests. This method of deferring interrupts via a FIFO queue can be disastrous. For example, when a system is flooded with mouse interrupts (which are rarely important for stable system operation) the FIFO queuing of pending interrupts (DPCs) can cause uncontrollable delays in the handling of other more critical interrupts. Avoiding the FIFO queue by doing all processing inside the ISR is not acceptable either, because it makes the system unresponsive to all interrupts (due to the fact that all interrupts are disabled during a Windows NT ISR), resulting in a system with proper behaviour for one interrupt only.

Some system designers may claim that they have tuned the total Windows NT system, including device drivers and application programs, in order to achieve acceptable determinism. But with the current rate of Service Packs and new releases of the operating system, plus the need for system feature adjustment typically required by customers, it becomes highly improbable that such tuning can be effective in the long term, after a few cycles of updating and modifications.

There are essentially two methods for correcting the lack of determinism in Windows NT:

- Use a separate computer system running a proper real-time operating system (RTOS) and connect it by a networking link to the Windows NT system.
- Extend the Windows NT system in such a way that at least part of the system provides the desired level of determinism.

The first approach may look attractive because there are many reliable and powerful RTOSes available from various suppliers. A disadvantage is that the application developer has to work with two different operating systems and their development environments The most significant disadvantage is cost: even the smallest application requires two complete hardware systems.

Extending Windows NT is an approach taken by only a few suppliers. Modifying the Windows NT kernel would be the best method, but is not a realistic option because the Windows NT kernel source code is not available and Microsoft has never expressed any intent to make this change to it's kernel – such a kernel change could even effect current Windows NT applications in a negative sense! Thus, another approach must be used to extend the determinism of Windows NT.

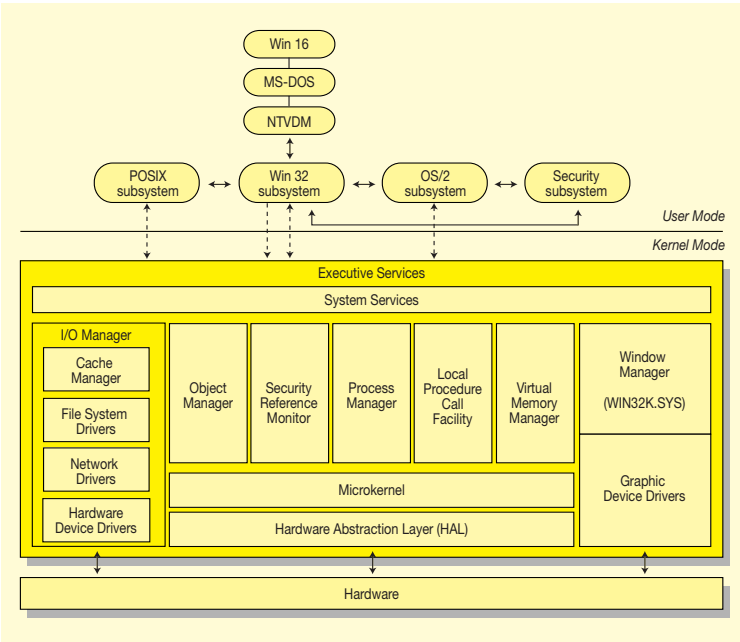

Figure 1. Windows NT architecture