University of Freiburg

Department of Computer Science

Bachelor Thesis

Introducing Threads into the TakaTuka Java Virtual Machine

Author:
Gidon Marian Ernst

Supervisors:
Prof. Christian Schindelhauer
Faisal Aslam

September 29, 2008

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Arbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

`Ort, Datum` `Unterschrift`

Acknowledgement

I wish to thank all the people who supported me during my work on this bachelor thesis.

First of all, I would like to thank Prof. Christian Schindelhauer for his support for the whole TakaTuka project and his valuable feedback.
Faisal Aslam, who was my supervisor, always had an open ear and took the time to discuss outstanding issues, which was very productive.
I also wish to thank everybody who has provided me feedback on this thesis itself, especially Jan Meyer and Benedikt Waldvogel for their suggestions on the text.

Abstract

Goal of this bachelor thesis is to introduce multithreading support into the TakaTuka Java virtual machine. The TakaTuka project is an effort to provide a Java environment and development platform for sensor motes. As these motes offer only a limited amount of resources, special care has to be taken in order to minimize the requirements of the virtual machine in terms of memory and computation.

The work is focused on basic threads functionality instead of implementing novel algorithms related to scheduling or locking. Subsequent projects will improve TakaTuka on the current thread infrastructure developed in this work. Therefore, several aspects of the presented implementation are evaluated and discussed, pointing out weaknesses and alternatives.

This thesis also takes a step forward to real application of the TakaTuka virtual machine by an integration into TinyOS. First successful attempts are made to adapt Java level threads to the underlying event-based driver architecture.

The most important demands to the implementation provided by this work are:

- Realization of Java compliant interfaces and semantics of the Thread class and Java's synchronization constructs.
- Low memory and computation overhead, as the target devices offer only restricted resources.
- A stable base for improving the capabilities and design of the virtual machine with regard to library and driver support.

Zusammenfassung

Das Ziel dieser Bachelorarbeit besteht darin, Unterstützung für Multithreading für die TakaTuka Java Virtual Machine bereitzustellen. Das TakaTuka-Projekt will die Programmiersprache Java im Bereich der Sensornetzwerke etablieren. Die dort verwendeten Hardwarekomponenten, auch "Motes" genannt, stellen nur begrenzte Speicher- und Rechenkapazitäten bereit. Deshalb liegt ein besonderes Augenmerk darin, die Anforderungen der virtuellen Maschine möglichst gering zu halten.

Diese Arbeit konzentriert sich auf die grundlegende Funktionalität von Threads, anstatt neue Algorithmen für Zeitplanung und Sperrmechanismen zu entwickeln. Weiterführende Projekte werden TakaTuka auf Basis der hier vorgestellten Umsetzung verbessern. Zu deren Unterstützung wird die vorgestellte Umsetzung kritisch diskutiert und Schwachstellen sowie Alternativen ausgezeichnet.

Außerdem zielt die Arbeit auf eine zukünftige reale Anwendung der TakaTuka-Platform ab, indem die virtuelle Maschine in das Betriebssystem TinyOS integriert wird. Eine Schnittstelle zwischen den Java-Threads und der zugrundeliegenden Treiberarchitektur wird erfolgreich umgesetzt und soll als Anknüpfungspunkt für zukünftige Entwicklungen dienen.

Die wichtigsten Anforderungen an die vorgestellte Implementierung sind folgende:

- Unterstützung von Threads wie sie im Java Standard definiert sind, einschließlich der Synchronisationsmechanismen.
- Der Resourcenverbrauch der Umsetzung sollte möglichst gering gehalten werden.
- Eine stabile Grundlage soll geschaffen werden, damit TakaTuka im Bezug auf Treiber- und Bibliotheksunterstützung erweitert und verbessert werden kann.

Contents

1 Introduction

1.1 Motivation

The field of wireless sensor networks is an active research area with increasing popularity. As wireless sensor nodes, also called motes, are getting smaller and cheaper, the application of such devices in real world settings becomes more and more convenient. A typical mote usually has less than 10kB of RAM, a few hundreds of kB flash memory and an 8-bit microcontroller. An example is Crossbow's mica2 [9], which has 4kb RAM and 128kb Flash memory. This is currently the primary test platform for the virtual machine, besides the PC.

Commonly, motes are programmed in low level languages, like C or nesC to encounter their resource constraints and the requirements of wireless sensor networks. A variety of operating systems have been developed, supporting different programming paradigms and environments. A programmer must make himself familiar with their features and peculiarities before he is able to use these systems efficiently. This may not only affect productivity, but also the quality of the code.

A popular operating system for wireless sensor networks is TinyOS [17]. It supports many hardware platforms and follows an event driven component based design approach. TinyOS is programmed in nesC, a language specifically designed for the application in wireless sensor networks. It provides clever abstraction and aims towards high reusability by enforcing the specification of interfaces for every component. However, nesC lacks support for objects and the use of dynamic memory allocation is discouraged. Also, the event-driven programming style is different from the synchronous multiprocessing environments with blocking calls, which many programmers know from desktop operating systems. Section 5.2 contains more details about the TinyOS internals.

Java is a popular programming language, which supports object-oriented design with run-time polymorphism, automatic garbage collection and well organized standard libraries. It was created with platform independence in mind and relies on a virtual machine to execute Java programs. The virtual machine can abstract over the different interfaces and glitches of hardware platforms and operating systems. It always provides the same environment, viewed from a Java program. These benefits ease the programming effort, whereas they also introduce significant overhead in memory and computation [21].

The goal of the TakaTuka project is to provide an implementation of a machine and Java environment, which is able to run on such constrained devices as sensor motes, by reducing the resource requirements inherent to Java systems as much as possible.

Java has built-in support for sophisticated multithread programming. The *Connected Limited Device Configuration* (CLDC) [22] [2] is a standard declared by Sun Microsystems. It defines a subset of the Java language and features, which a virtual machine has to provide in order to be CLDC compliant. This standard mandates thread support, hence TakaTuka

must implement concurrency in order to fully support CLDC, which is our aim. Once implemented, threads offers a flexible organization of concurrent tasks within the Java sensor mote application.

1.2 Related Work

Currently, there are several efforts to introduce Java as a sensor networks programming language. It can be taken as evidence of a growing market and future perspective of Java in this field.

Sun Microsystems has introduced their Java virtual machine called *Squawk* [21] [23], which currently runs on a mote named *SunSPOT* [5]. The source code of the Squawk system has been put under the GPL in January 2008. Squawk is has an unusual design, as much of its core functionality is actually implemented in Java itself. Some of this Java code is translated to C and directly compiled into the virtual machine, however, most of it is interpreted at runtime. The interpreter is written in C and provides the necessary hooks for the Java part to operate. This approach allows the developers to test much of the virtual machine code as a Java progam, supported by the existing toolset and the stricter typing rules of Java. Although Squawk aims at resource-constrained devices, it needs around 14kB of RAM and around 500kB of Flash to execute a "Hello World"-program [12]. The SunSPOT has a total 512kb RAM and 4mb Flash memory. Typical motes do not offer this much resources, hence it is impossible to port Squawk to such devices.

Sentilla Corp. sells their own custom Java virtual machine [13] which is limited to their own motes. It uses less memory than Squawk and is CLDC compliant, but it is not openly available.

Multithreading is an active research area in wireless sensor networks:

The MANTIS operating system [7] supports a subset of the posix threading interface, which allows scheduling of stateful machine level threads.

The RETOS operating system [15] also offers posix threads with realtime capabilities. It incorporates several optimizations to reduce the overhead of a thread implementation, like stack size analysis [16].

The protothreads library [11] provides cooperative pseudothreads implemented by a few C macros, requiring minimal additional memory for its operation. However, it does not support real thread preemption.

Finally, the TinyOS scheduler has been extended to support preemption of tasks to meet timing deadlines of high priority [8].

1.3 TakaTuka

The TakaTuka project aims to create a Java virtual machine for sensor applications. It should be able to run on small devices with less than 10kB of ram and around 30kb of Flash memory. TakaTuka should be fully CLDC compliant.

The virtual machine currently runs on the mica2 mote and on the PC. Already implemented features include garbage collection and support for almost the complete Java bytecode set. Besides this, effort has been put into reducing the size of the interpreter and the Java program. Depending on the application, the size of the interpreter can be configured between around 5kb to over 30kb for the full featured, threaded version.

An integration into TinyOS is in the process, taking advantage of its broad hardware support: an abstraction layer between the virtual machine and TinyOS components provides their functionality in the Java environment, without the need to implement low level and hardware dependent drivers. As shown in chapter 5, it is possible to adapt the two different programming paradigms of nesC and Java by the support the multithreading established during this work.

1.4 Organization

The rest of this thesis is organized as follows:
Chapter 2 introduces the basics of concurrency in computer systems in general. It also discusses several aspects not directly related to the current implementation in TakaTuka, however, possible directions to address these issues are covered in section 7.
Chapter 3 explains how threads are embedded in the Java programming language and how they work from a conceptual point of view.
Chapter 4 describes in detail the implementation of the thread semantics in the TakaTuka virtual machine in detail. It points out issues concerning the design and discusses the decisions made.
Chapter 5 shows the use of threads to provide a blocking interface on top of the event-driven code of TinyOS.
In Chapter 6 evaluates the implementation in terms of memory usage and scheduling performance.
Finally, in Chapter 7, conclusions from this work are drawn. TakaTuka is compared to the Squawk virtual machine and future improvement opportunities of threads in TakaTuka are discussed.

2 Theoretical Background

2.1 Processes and Multiplexing

A modern computer system has the capability to run several programs simultaneously. It lies in the system's responsibility to multiplex all resources that the different programs may access. This includes I/O channels and memory regions, but the most important shared resource is the CPU. Although we have multiprocessor systems nowadays, it is still necessary to split the computation power of one processor to run multiple programs in a pseudo-parallel way. This will be referred to as *multiprocessing* here. Traditionally, a program running in such a system is called a *process*.

Time division multiplexing is a technique to allow several several processes sharing a resource, each having predefined cyclic time slots assigned, in which a given process may actually use the resource. This technique is also used in the case of CPU management.

In turn, each process is assigned a time slot to run in. When this slot is exhausted, another process is chosen to run. The component which manages the lengths of the time slots and chooses which process may run next, is called the *scheduler*.

Conceptually, a process views the machine as if there were no other programs running at the same time. In particular, it has its own address space in the memory. Support for this is usually built into the hardware. Each process has an associated table of information, which stores the state of its execution. This is called *context*. Also, this table stores the memory mapping. The act of pausing one process and resuming another is called *context switch*.

2.2 Threads

Sometimes, it is desirable to share memory between programs, since they operate in a cooperative way. To support this, lightweight processes called *threads* are introduced. They share most of the information of the process table and consequently they are grouped into one process. So the difference between threads and processes lies in the cooperation level, resulting in more permissive memory accesses. For scheduling and locking concepts there is no significant distinction.

2.3 Scheduling Strategy

There exist two approaches to multiprocessing concerning the decision how a context switch should occur. In *preemptive* multiprocessing, the scheduler has the possibility to interrupt a process at any point in time and immediately select a new one. However, a process can give up its use of the CPU in advance. In *non-preemptive*, or cooperative multiprocessing, a process may decide when to give up the CPU. If it chooses not to do so, it can run arbitrarily long.

2.4 Shared resource access

Preemptive multiprocessing has the clear advantage, that CPU time can be distributed equitably among the processes by an independent instance. However, at any time, a process must be prepared for a context switch. The process might be in the middle of a critical operation which could affect other processes. Then, it must be assured that all processes depending on the result of this critical operation wait, until the process has finished the operation in the future. Such operations include using a unique resource other than the CPU (classical example: printer). Generally, a *resource* is a part of the system, which provides functionality or information. Variables in a program are also considered as resources.

Example: Failed concurrency In this example, two processes running at the same time affect the operation of each other. Both have access to a shared resource, the variable x in this case. Process A tries to compute the square root of a value stored in x, first checking if x is positive. Process B just assigns a negative value to x. In a preemptive system, B can run anytime between two lines of A, and before or after A. Between line 1 and 2, A has already decided to compute the square root, relying on a positive value of x. However, if B changes x to -1 at this time, A will run into an error. This example demonstrates, that atomicity of operations can be critical for correct operation.

```
process A:

1:   if(x>0) {
2:       y = sqrt(x);
3:   }

process B:
1:   x = -1;
```

To control the correct access to resources, a concept called *lock* is introduced. A lock can have the status *acquired* or *free* and is used to protect one individual resource. If the lock is free, any process may acquire the lock and use the resource. When the process does not need it anymore, it frees the associated lock. If the lock is already acquired by another, a process desiring the use of the resource has to wait until it is unlocked. The process which has acquired the lock *owns* the lock and may use the resource. The part of a program, which lies between a lock acquirement and the corresponding free operation is called a *critical section*. The task of managing access to resources is called *synchronization*. If two processes wish to acquire a lock at the same time, we say they *contend* for the lock. The above example could be improved by introducing a critical section around both programs, protected by the same lock. Consequently, B could only modify x before or after A runs.

2.5 Deadlocks

A deadlock is a situation, where two processes wait on a lock, which is owned by the other. Both can not execute until the other releases the lock, which will never happen, thus effectively disabling the system's function.

Deadlocks can be detected by analyzing the lock dependency graph, but although there exist algorithms to avoid deadlock situations in advance, there is no general solution to remove them completely.

2.6 Priorities

Modern systems also know the concept of priorities. For example, an audio player should be able to run frequently, otherwise the playpack will be disturbed. Another process might copy a big file on disk at the same time, which is usually not a time critical operation. Therefore, each process is has a priority, indicating how preferably it should be run. This may influence the length and frequency of time slices. Also, a higher prioritized process might be preferred during the contention of a lock.

A system with priorities and locks can run into a situation called *priority inversion*. If a highly prioritized process is waiting for a lock held by a low prioritized process, the latter will be executed, eventually releasing the lock. However, if a medium prioritized process is present, it will be preferably scheduled over the low prioritized one. This effectively degrades the performance of the highly prioritized process further by delaying its execution. Common solutions involve boosting the priority of the task holding a lock, either to a fixed high value or to the highest priority of any waiting process.

2.7 Implementation of Multiprocessing

As mentioned before, support for multiple processes is often built into the hardware. The processor executes the machine instructions of the current process, until an event is signaled by an external component, such as a timer. These events are called *interrupts* and each has an associated piece of code, the *interrupt handler*, which is part of the operating system (OS). This mechanism is commonly used to implement multiprocessing: A timer interrupts the running process periodically, allowing the OS to interfere and select another process to run. In the interrupt handler, the context switch is performed: all CPU registers are backed up to the process's context, so that its current execution state can later be restored. Then, the context of the next scheduled process is activated, so that it resumes execution.

In a virtual machine, the realization of concurrent execution is less constrained than on a real CPU. Consider a (hypothetical) processor with a built-in scheduler: the operating system would not be concerned with context switching at all. Similarly, an interpreting virtual machine

can run arbitrary code in between the dispatch of two virtual machine instructions, multiplexing itself to several virtual machine threads. Such threads do not have a context in the operating system and are not scheduled in the classical way, since the virtual machine provides its own scheduler, registers and context specification. Threads, which are managed by a virtual machine without the operating system's support, are sometimes referred to as *Green Threads* [21].

2.8 Use of Multiprocessing, Blocking Calls

Traditional calls to the operating system, which require an event such as user input, will wait for this event before they return. In the meantime, the calling process is put aside and paused. This guarantees, that data is available when the process resumes. This behavior is often referred to as *blocking* or synchronous calls.

If an application wants to perform other tasks while waiting for a blocking call, a common solution introduces a new thread for each task, so that blocking does not affect the whole system. A classical example is a webserver, which delegates each connection to a new thread. This thread may then use blocking calls to read from the connection, handling the data and terminating when its job is done, while the main thread keeps active, accepting and delegating new connections.

3 Threads and Java

3.1 Overview

Java is a popular object-oriented programming language. It has built-in support for preemptive multithreading by design, providing a small number of concepts: *Monitors, synchronized sections, waiting and notifications and finally classes to spawn and manage threads.* On top of these features, many common multithreaded patterns, like task pools, can be implemented. The design of Java threads heavily relies on the object-oriented nature of the language.

In TinyOS, the synchronization is managed by the `atomic` keyword [20]. It protects a code region from being preempted by other code using the same variables. The data, which is the resource that should be protected, is not explicitly given by the programmer. Java takes the opposite approach: synchronization must be associated with a *monitor*, provided with each object. Monitors indicate, if an object is currently used by a thread, ensuring exclusive access to this object. However, it still lies in the programmer's responsibility to identify possible concurrent data access. The relation between monitors and the protected data is not specified by the language itself, it is a protocol which the programmer defines to guarantee safe operation. The concept of monitors is further explained below in section 3.2. In addition to monitors, Java provides a way to communicate ownership of an object between threads, in order to manage cooperated access. This is explained in section 3.4.

Modern Java virtual machines, like Sun Microsystem's JRE, rely on the operating system as back-end for their thread implementation. Native OS threads have several advantages over green threads. Green threads do not allow preemption while a thread is executing native code. The virtual machine does not have control over the execution while a thread is inside a call to a native function, thus preventing virtual machine code to run and perform scheduling. This also limits the use of a JIT. As it is undecidable how long loops would run, the JIT could not fully compile loops into native code. As indicated in section 2.7, the implementation of green threads is easier though, because it can be done entirely inside the high level implementation language.

3.2 Monitors

Each object in a Java program has an associated lock, also called *monitor*. Like the traditional semantics of locks, only one thread can own a monitor at any time. A thread can acquire the same monitor several times recursively. The virtual machine counts, how many times the owner has acquired the monitor. This number is referred to as the *entry count*. Conversely, the monitor is only freed, if the owner has released it as often as acquired.

The Java virtual machine specification [18] (see chapter 8) does not mandate the format and allocation policy of monitors.

Monitors support the following operations:

- `monitorenter`: Entering a monitor is equal to acquiring the monitor, locking the object for exclusive access. The operation is implemented as a bytecode, the monitor reference is passed on the Java stack.
- `monitorexit`: The operation decrements the entry count by one, releasing the ownership if equal to zero. Monitorexit is also initiated via a bytecode.
- Waiting and notifications as explained below.

3.3 The Synchronized Statement and Synchronized Methods

The Java language includes the *synchronized* statement, which is used to denote critical sections. It takes a Java object as argument, indicating that this section should be protected by the object's monitor. The synchronized statement is compiled into a `monitorenter` bytecode instruction at the beginning and a `monitorexit` instruction at the end. It is syntactically ensured, that both operations are always used symmetrically. The Java compiler enforces this property, even when exceptions are thrown past the end of the synchronized block. The corresponding `monitorexit` instruction is then placed inside a dummy exception handler. The programmer is freed from the tedious task of keeping locking and unlocking symmetric, and it is impossible to break the mechanism intentionally.

Java provides a shortcut for the frequently applicable pattern of protecting the whole body of an instance method with the object reference, on which the method is called: the *synchronized* method attribute. The virtual machine enters the monitor of the object implicitly, without the use of the bytecode, analogously releasing it as soon as the method returns. As special case, static methods can also be declared synchronized, locking a monitor associated with the method's class.

Example: Use of synchronized blocks

```
class Vector {
    // insert o into the vector
    // alters the internal state
    public void insert(Object o) {
        synchronized(this) {
            // code block
            // protected by the monitor attached
```

```
            // to the object referenced by "this"
        }
    }
}
```

In this case, the code block is protected by the monitor of the Vector represented by `this`. It is equivalent to declaring the method as `public void synchronized insert(Object o)`. For all objects, only one Java thread can enter a synchronized block, which is protected by the monitor of a specific object. However, two threads can enter the same code block at the same time, if they synchronize on two *different* objects. On the other hand, no two threads can enter two different critical sections, if both are protected by the same monitor. If a thread owns a monitor, it can enter any other critical section protected by this monitor. Note that the value of `this` (the object which it references) cannot be determined at compile time, but depends on the flow of execution during runtime. Summing up, it can be said that a monitor protects data and not code.

3.4 wait/notify mechanism

In addition to the synchronization of code blocks, Java provides a more sophisticated way for communication between threads. The concept is to allow threads to send each other event notifications, using an object as channel. In order to receive such an event, a thread pauses its execution, waiting for the event to occur.

The class `Object` has built-in methods for *waiting* and *notifications*, as illustrated in a (schematic) code excerpt of the Object implementation.

```
class Object {
    ...
    public final void wait()
                throws InterruptedException;

    public final void wait(long timeout)
                throws InterruptedException;

    public final void notify();
    public final void notifyAll();
    ...
}
```

`O.wait()` causes the current thread to give up O's monitor and to suspend its execution, waiting for a notification on this object. The set of threads waiting on an object O is called the

wait set of O. `O.notify()` and `O.notifyAll()` causes threads in the wait set of O to wake up. The difference between these methods is, that `notify` only wakes up a single thread, while its choice is not defined but dependent on the implementation. `NofityAll()`, in contrast, resumes all threads which are waiting on the object.

Both commands, wait and notify, require the current thread to own the object's monitor. A call to wait completely releases the thread's ownership, thus allowing other threads to acquire it. The state of the ownership is stored inside the thread to later recover it. Otherwise, notification would not be possible, since the notifier must also synchronize on the object. A call to notify causes one or more threads to be awaken. They will try to restore their previous state, for which they will have to contend in the usual manner. The execution of a thread continues, as if `wait` had just returned.

`Wait` can be given a timeout, defining the maximum time to wait. After that, the method simply returns as if notified. Java defines no mechanism to detect if there was a timeout or a real notification, however, a programmer can provide own code to distinguish the situations. If no timeout or a zero timeout is given, the thread would wait forever until notified. The `wait` methods may also throw an `InterruptedException`. Interruption is explained in section 3.5.3.

Example: use of wait/notify: A webserver has one thread for accepting connections. Whenever it receives a request, it delegates the new connection to a thread designed to handle the request. These are organized in a thread pool, waiting on a communication object, in this case a queue containing the open connections. The main thread puts new connections into the queue and notifies one thread of the pool, so that it can answer the request. The example is somewhat incomplete as it does not guarantee an unoccupied handler thread in the pool, but the concept should be clear.

```
/* Main Thread: */
while(true) {
    Connection connection = accept();
    synchronized(queue) {
        queue.put(connection);
        queue.notify();
    }
}

/* Handler Threads: */
while(true) {
    synchronized(queue) {
        queue.wait();
```

```
        Connection connection = queue.get();
    }
    // handle the connection here
}
```

3.5 The Thread class

Besides synchronization, the `Thread` class is the most visible part of multithreading to the Java programmer. Objects of this class correspond to running threads inside the virtual machine. The programmer can implement the behavior of a thread by subclassing and overriding the `run` method. This serves as the entry point to the thread's operation. The thread exits as soon as this method returns. Alternatively, a class implementing the Runnable interface can be instantiated and given an existing thread object to run. A thread will be scheduled after the `start` method was called. Other methods affecting the state and operation of a thread are consequently grouped into this class.

A schematic subset of the class is presented, as defined by the *Java (TM) 2 Platform Standard Ed. 5.0* [3].

```
interface Runnable {
    void run();
}

class Thread implements Runnable {
    Thread();
    Thread(Runnable target);

    void run();
    void start();

    static Thread currentThread();

    static void sleep(long timeout);
    static void yield();

    void join();

    void setPriority(int priority);
    int getPriority();
```

```
    Thread.State getState();

    void interrupt();
    boolean isInterrupted();
    static boolean interrupted();

    static boolean holdsLock(Objecet obj);
}
```

3.5.1 Runnable target

Java restricts inheritance to a single base class and an arbitrary number of interfaces. If the thread's main class already has a determined base, it cannot derive from `Thread` anymore. Hence, threads can also take an object implementing the `Runnable` interface as target, invoking the objects run method. The default implementation of `Thread.run` checks, if a runnable target was given to the threads constructor, and runs that one. Otherwise it simply returns.

3.5.2 Thread States

The state of a Java thread is modeled by the nested `Thread.State` [14] class with six constants.

- `NEW`: A thread has this state, until it was started successfully by calling `Thread.start`.
- `RUNNABLE`: The thread is initialized and may be scheduled.
- `BLOCKED`: The thread is blocking in a monitorenter instruction to gain ownership of an object.
- `WAITING`: The thread is waiting for another thread's notification.
- `TIMED_WAITING`: The thread is waiting for a notification with a given timeout.
- `TERMINATED`: A thread has exited the run method either normally or by an uncaught exception. Indicates, that it should not be scheduled.

In Java, the term *blocking* is used more restrictedly as compared to section 2.8. Both refer to a state where the current thread has to wait for a condition, but in Java it specifically means waiting for a monitor in contrast to waiting for an event notification.

3.5.3 Interruption

Threads can be interrupted while in an operation, for example to indicate that the operation will not terminate successfully. If the `interrupted` method is called on a thread, the action taken depends on its current state. If it is waiting on a notification or timeout, the corresponding method call will signal an `InterruptedException` in that thread. If the thread is runnable, the interrupted flag is set, which can be checked by the thread periodically. While the wait and notify mechanism provides a way to communicate normal events, interruption can be used to signal errors.

3.5.4 Further operations

Java's model of application security involves a security manager class, providing a security context. It specifies, what actions can be taken by the Java program in a given context. Operations, which modify a thread from another one (like interrupting) depend on such policies. If access is not granted, a security exception is thrown.

A thread may invoke the *yield* operation to give up its remaining time slice. The scheduler selects another thread to run. A call to *sleep* pauses the current thread for a given time. *Joining* another thread acts like waiting on some designated object, which is notified as soon as the other thread terminates.

3.6 Shared variables

The Java virtual machine specification [18] chapter 8 defines terminology and semantics of the different thread constructs. Java allows threads to have a private copy of variables and to work on them without updating the main memory. When performing synchronization, the virtual machine also writes back the private copies. These rules mostly apply to compiled Java code or special optimizations, for example when caching variables or objects in registers or on the stack. Since the TakaTuka virtual machine does not use such techniques, many rules do not have to be implemented explicitly.

3.6.1 Java Stack

The Java stack is organized in *activation records*, one for each method call. An activation record stores the *local variables* and the *operand stack*. The local variables also include the parameters passed to the function, hence all variables which can be referred to with a locally scoped identifier in Java. The operand stack is used for computation and temporary results, as the Java virtual machine uses a stack based computation model. Stack-frames do not need to be contiguously aligned in memory and the virtual machine is free to allocate them in a convenient manner (see [18], 3.5 Runtime Data Areas). The stack itself is organized in *slots*.

A slot must be large enough to hold all Java types, including references, except for `long` and `double`, which occupy two slots. Knowledge about the stack layout will prove valuable when implementing multiple stacks for multithreading.

4 Implementation Details

4.1 Multiplexing the Virtual Machine

4.1.1 General Design Deliberations

Green Threads or Native Threads TakaTuka implements green threads instead of native threads. The disadvantages have already been discussed in section 3.1. The decision for green threads was made for a couple of reasons. The virtual machine should run on many different hardware platforms. The low level threading code, especially the context switch and the stack allocation, has to be implemented partly in assembler for native threads. This is very target specific and would have to be redone for each platform. Synchronization concerns are thereby pushed into the virtual machine, as management code may run in an arbitrary thread. Several threads could enter the same virtual machine routine concurrently, so all code must be *reentrant*. This implies restrictions on global variables and enforces locking of data structures inside the virtual machine. The resulting code and memory overhead overweights the benefits of the possible preemption in native methods. Using the thread implementation of another wireless sensor networks operating system like MANTIS or RETOS would restrict TakaTuka to such an environment, which is not our desire. TinyOS integration would be harder to do with native threads, as the system is not thread-safe, although support for preemption has been built into it recently [8].

Speed vs. Size Efficiency of threads is the major concern, RAM and CPU usage should be kept as low as possible, leading to trade-off decisions between speed and size. Currently, we aim at reducing RAM usage as much as possible before optimizing calculation time. This manifests in the organization of threads in one queue, as outlined in section 4.5 on the scheduler. However, the overall system will be revised from an algorithmic point of view soon.

4.1.2 Stack

The stack is organized by activation records, also called *stack-frames* in TakaTuka. In general, an activation record carries the data which is necessary to resume its method invokation. A stack-frame in TakaTuka concretely consists of the current stack pointer, the program counter and a reference to the current method. Additionally, a reference to the caller's stack-frame is stored, providing the necessary information to return from the call. The virtual machine maintains a pointer to the currently running stack-frame for its execution. Memory for local variables and the operand stack is adjacent to this information, so that stack-frames are completely self-contained and not dependent on a particular organization in memory. Usually, a stack grows down downwards, towards the heap memory. The stack and the heap are abstracted in TakaTuka and due to stack-frames, the stack growth direction is not fixed by the design.

In TakaTuka, the stack inside a frame growth upwards. Please note that the total number of parameters and local variables and also the size of the operand stack can be determined at compile time. This value is provided to the virtual machine for every method and stored in the class file. Being self-contained is a feature that is paid by one extra register in the activation record: the stack pointer after returning is known in a continuous stack implementation and can be kept in a global register. Our approach is compared to Squawk in section 7.2.

4.1.3 Context Cache

In order to optimize several operations, the virtual machine caches some pointers related to the executing method.

- A pointer to the local variables to speed up their lookup, since they are used frequently by the bytecode. The local's memory region is below the operand stack. Parameters are expected to be the first part of the local variables and are also referenced by the pointer.
- A pointer to the beginning of the bytecode array of the current method. For example, the `jsr` bytecode (jump to subroutine, as in [18] chapter 6) depends on the offset of the program counter.
- A pointer to the base of the operand stack of the current method. This is used for clearing the operand stack when catching exceptions and for stack pointer checks in the debug-enabled version of the virtual machine.

The latter two pointers are not strictly necessary, as they mainly speed up the use of exceptions and debugging, which is not the general case. On the other hand, their memory usage is comparable to an empty Java object and therefore acceptable.

4.1.4 Thread extension

With the current stack design, several adaptions to the virtual machine had to be made to support concurrently running execution flows:

- Stack-frames have to be grouped by threads. This is easy to implement, as they already form a linked list with the current one as its head. Hence it is sufficient to keep a pointer to the top stack frame in each thread object.
- The allocation of stack-frames becomes more complicated, since they are not created in *first in first out* order (FIFO) any more. This property is only true as long as one thread runs, but can be invalidated by context switches. Usually, each thread is assigned its own private stack memory, where the FIFO order can be restored locally. Since the maximum

total stack requirement can not be determined in advance, it has to be overestimated, wasting memory.

In TakaTuka, frames are currently allocated by the system library's dynamic memory management. This slows down the stack considerably (see section 6.2), but bypasses the overestimation of each stack's size.

- The virtual machine keeps a reference to the current thread. To perform a context switch, two operations are needed: saving the current context in the executing thread and activating another thread. Saving is trivial, as all registers are directly used from the stack-frame and are always up to date. Hence it is sufficient to update the frame pointer in the thread. Restoring involves the recalculation of the context cache, reading out some method properties and pointer arithmetic. The procedure is similar to invoking or exiting a method.

- Rescheduling hast to be initiated regularly even if no thread blocks, to allow concurrency. The virtual machine counts the number of backward and return branches and switches threads if the sum has exceeded a certain threshold. This guarantees linear code to execute non preemptively. The strategy reduces the number of counter checks, as only certain bytecodes can initiate switches. Possibly, this reduces the number of locked monitors when a context switch is performed, leading to less contention overhead. This aspect could be investigated in the future (see section 7.3).

4.2 Implementation of the Thread class

4.2.1 Native Code

A Thread in TakaTuka carries the following internal information, which is private to the virtual machine and not directly visible in Java.

- The Thread objects inherit from the Object class all management information of the virtual machine. Also, the support for monitors on a thread object itself is given.

- A state, indicating if the thread is runnable, blocked, waiting, during initalization or has terminated. The scheduler has special private native methods to read and modify this state. The state is defined by the Java specification, see section 3.5.

- A record to backup a monitor's state while a thread is waiting: an object reference and the entry count. This is entirely managed by the virtual machine and not exported to Java, like the monitors' state itself.

- A pointer to the current activation record and the stack space of the thread, as described in section 4.1.2.

- To signal exceptions in a thread which is not executing, a reference to a pending exception can be stored in each thread. A Java interface is provided to set this exception. When the thread's context is activated, this reference is evaluated. If an exception is pending, it is thrown immediately after switching to the thread.

4.2.2 Java Code

Some parts of threads are directly modeled in Java.

- The runnable target is easily implementable in Java, as it does not require any support by the virtual machine.

- All threads currently subject to the scheduler are organized in a doubly linked list. Pointers to the next and previous element are directly stored in the thread class as public variables. This design is not very clean, since the list can be corrupted by anyone. On the other hand, it is straightforward and also the fastest possible implementation besides having the scheduler entirely as native code. Section 7.4 provides more thoughts on this.

- The methods for interrupting, yielding, joining and sleeping require direct support by the virtual machine. They are exposed via the `Scheduler` class, and the thread class simply delegates these operations. The rationale for this is, that they can be brought down on a common denominator, mainly through a generalized wait/notification system.

- Starting a thread is also delegated to the scheduler, simply by adding the thread to the list. It is initalized later when running for the first time.

- Interruptions are implemented in Java, the interrupted flag is a field inside the thread class. Special handling is only required in the case that the interrupted thread is currently in a wait operation, which is covered by throwing a remote exception.

- Priorities are currently not implemented. Most of the priority support can be done in Java, with one exception: notifying a single thread should choose an approrpiate, high prioritized one. Since the implementation is done in native code, it would have to be adapted to the scheduler's logic. This should improve, when more of the wait/notify code moves to Java.

4.2.3 Deprecated Methods

The methods `stop`, `suspend` and `resume` have been deprecated by the Java standard [3], hence they are not implemented in TakaTuka. If the need arises, they can be modeled in Java by modifying the Scheduler.

4.3 Implementation of Monitors

4.3.1 Data Structures

The monitors, associated with Java objects, are natively implemented by the virtual machine, which means they do not depend on Java code. A monitor references the owner thread if locked and stores the entry count. In TakaTuka, each monitor has two fields storing this information. The fields are private to the virtual machine and not exposed to the Java environment. An object having an owner equals to `null` is not locked and must have an entry count of zero.

Monitors are allocated on demand. The virtual machine keeps a global list of all currently used monitors in last recently used order. A small number of unlocked monitors is kept for reuse. This has the advantage no memory overhead inside objects is introduced.

In the future, more functionality of the threads might be implemented in Java, eventually modelling monitors as proper objects. This will be discussed in section 7.1.

4.3.2 Operations

Entering a Monitor The actions taken when entering a monitor depends on the current owner. If the monitor is unlocked or already owned by the current thread, it may claim ownership and increase the entry count by one. If the monitor has already been acquired by another thread, the current thread enters a blocking state. It saves the monitor, which it failed to enter, and the virtual machine initiates a context switch, running another thread.

Exiting a Monitor Exiting a monitor is done by decreasing the entry count by one. If the count reaches zero, the monitor is unlocked, setting the owner to null. The current thread may continue its execution unless it loses ownership of the monitor, to allow other threads to acquire it. This guarantees fairness among all threads.

Waiting The wait operation on a monitor first checks, whether the current thread owns the monitor. If so, the monitor's state is stored in the thread, and the monitor is freed, according to the Java specification. Otherwise, an exception is thrown. The thread's state is set to waiting and if a timeout is given, the wakeup time is associated with the thread. Then, rescheduling is initiated.

TakaTuka supports anonymous waiting without a specific monitor. The thread is simply put into the waiting state. This is useful for the implementation of the `sleep` method. Such a thread can not be notified, however, it can be awaken by a timeout.

Notifications Notifications also first check ownership in the same way as waiting. The virtual machine traverses the list of threads to find one, which is waiting on the monitor, setting its state to blocked. The thread will then take part in the contention of the monitor, and is

eventually resumed, just returning from its call to wait. If all threads on the monitor should be notified, the search continues until all threads have been examined.

A special version of notify exists, which is mainly a workaround for drivers: Native code can call notifications without being the owner of an object. As it is not clear what thread is currently running, it is not suitable to manipulate its locking status. Ownership for wait and notify is usually necessary to prevent race conditions between threads and to protect the internal state of the monitor from concurrently being manipulated by several threads. If the Java interface takes care of not exposing the object on which the driver notifies the waiting thread, it is guaranteed that the monitor is free anyway when the event occurs, resulting in a safe system.

4.4 Components written in Java

Parts of the virtual machine are written in Java itself. This includes the scheduler and the garbage collector, for example. These components will be called *services* from now on.

4.4.1 Initialization and Services

The virtual machine is represented by a special Java class named `VM`, which is responsible to coordinate the initialization and invocation of the TakaTuka runtime and its services. After the interpreter has started up, it creates a thread inside the Java heap for the operation of the `VM` class and the services, which will be referred to as the *service thread*. Control is transferred to the initialization procedure written in Java, running in this thread, which sets up storage for static variables, the input/output streams, and calls service initializers. In the next step, a second thread is created and started. In this thread, the main method of the application is invoked. The concept of having a service thread is taken from the Squawk virtual machine and discussed in section 7.6.

4.4.2 Dispatching Service Requests

After the initialization has completed, the service thread enters an infinite loop. The thread is paused and not run again until a virtual machine service is requested, either by some Java code or by the virtual machine itself. The `VM` class has a variable, indicating the desired service. So, whenever a service is demanded, this variable is set accordingly and the virtual machine switches to the service thread, which evaluates the request and performs the corresponding actions.

The service thread does not use wait and notify to pause and resume itself, since this would result in a circular dependency: the scheduler is required to switch to the service thread, and at the same time, this same switch is required to run the scheduler. Instead, the scheduler explicitly excludes the service thread from the list of runnable candidates. Only in case of a

service request the virtual machine directly runs the service thread. This is triggered either by a call to the special `VM.requestService(id)` method, or internally from the C code.

4.5 Scheduler

The core scheduler of TakaTuka is written in Java, assisted by several native methods. It is a simple round robin scheduler, selecting the next runnable process in turn.

If a thread has exhausted its time quantum, the virtual machine posts a scheduling request and then switches to the service thread. The service thread dispatches the request, which in this case runs the scheduling algorithm. The scheduler iterates over the list of all threads, deciding its action depending on the state of a thread.

- If the thread is runnable, the scheduler calls back to the virtual machine, specifying this thread as the next one to run.
- If the thread is blocked on a monitor, there are two cases: 1) The monitor has been released by the owner since the last thread switch, then allow the current thread to acquire the monitor. The saved monitor state is recovered and ownership is declared. 2) The monitor is still locked by another thread, then continue the search with the next thread in the list.
- If the thread is waiting on a notification without a timeout, continue the search, since the notification is performed internally. If a timeout is set and it has expired, then set the thread's status to blocked, indicating it should contend for the monitor. The ownership status holding before the call to wait is restored as described in the preceeding bullet-point. The current implementation allows the thread to contend immediately in the same scheduler loop iteration.

If there are no runnable threads, the scheduler has traversed the list without determining a new runnable thread. All threads are either blocked or waiting. If all threads block on a monitor, then we ran into a deadlock and the virtual machine terminates with an error. If there is at least one thread in a timed waiting or depending on a driver notification, the whole system can pause and enter an energy conserving sleep state. TakaTuka relies on a library function for this operation. Other power management like turning off the radio should not be automated and are left to the application's programmer.

5 Case Study: Mapping TinyOS Events to TakaTuka Threads

5.1 Event Style vs. Threads

TinyOS adapts to the event style programming, where calls to the operating system are split into a command (request) and an event, which carries the response. Normal TinyOS code can only be preempted by interrupts. The relation between commands and events may be arbitrary and each component has to keep track of its current state manually.

Classical read and write commands to the operating system block the calling process (see section 2.8). A major benefit of threads is the automatic state management. TakaTuka will use threads to provide a blocking interface on top of the TinyOS event based architecture. Purpose of this case study is to show the application of TakaTuka's threads inside driver implementations.

5.2 TinyOS Structure

TinyOS is highly modular. It has a strict distinction between interface and implementation. Components provide and use interfaces, which is the only way for interaction. Configurations group several components, specifying what concrete component should be selected as the implementation of an interface, which another uses. This is called *wiring*.

The nesC compiler traces the use of components, starting with the main application configuration, following all referenced components and configurations recursively. This allows it to remove much of the TinyOS code, eg. implementations of drivers for other platforms and unused features. During this step, it translates components to C functions and performs aggressive optimizations and inlining. As result, a big C file, containing the whole application and operating system code, is given to the appropriate C compiler. The nesC compiler can be instructed to generate an application binary, or an object file, which can be further linked with other code.

5.3 Embedding TakaTuka inside TinyOS

5.3.1 TakaTuka running as a TinyOS component

The main interpreter loop of TakaTuka originally runs until the Java program exits. For perpetual systems, this is equal to "forever". As TinyOS does not support preemption, the main loop has to be adapted: it runs for a specified time and periodically returns to TinyOS, leaving the virtual machine in its current state. This allows the TinyOS system to signal events and do its own bookkeeping. Later, the main loop is resumed.

5.3.2 Component-Class Mapping

In the simplest case, each driver component has a corresponding Java class, which exposes the functionality as methods. The first decision is to determine whether to use class or instance methods. If the driver has to keep a state for every client, like a timer, it can easily be stored in instances of the Java class. A reference to the object is automatically passed as parameter to the native method. The driver does not have to worry about allocating and releasing this object, which is done by the programmer in Java.

5.3.3 Calling TinyOS from Java

By default, the nesC compiler declares all functions as static in the generated output, so that they can not be referenced from outside this file. We link TakaTuka with the object file generated by nesC, and every function which is called from TakaTuka but implemented in a TinyOS component must be annotated with the special attribute `__attribute__((C,spontaneous))`, to prevent it from being hidden [10].

TakaTuka calls its native functions using an ID, stored in the bytecode. These IDs are consecutive, so that the corresponding function pointer can easily be looked up by indexing a table. This table is automatically generated by the PC side class loader, referencing functions by a special naming convention. The function can be placed in any C file, as long as it is linked with the virtual machine. Exploiting this feature, the implementation of a function inside TinyOS components is straightforward: Placing it inside a TinyOS component gives it access to the components data and code, while being directly interfacable with the virtual machine.

5.3.4 Calling Java from TinyOS

Calling Java methods from TinyOS is more challenging. The main problem is, that an arbitrary thread is running when the component in TinyOS wants to issue the call. Although it is possible to manually switch to the desired thread, perform the call then and switch back, this implies a big overhead, hence this approach is avoided. Also this falls into the category of event driven programming, as the driver issues a callback.

The Java semantics provide a very convenient way to solve the problem: The wait and notify mechanism. If a thread issues a call to a driver, the driver may decide to let the thread wait on a determined communication object. When the data arrives, the driver can notify on the communication object, so that the thread resumes its execution. Preferably, the communication object can also be used for data passing. By this technique, the driver does not even have to know the thread, only the object, which can be stored inside its private data. The wait/notify system automatically selects the correct thread and resumes it.

5.3.5 Passing Data to Java

Data is copied into Java objects as soon as it is available in the driver. This frees it from keeping buffers or queues, leaving such management to Java, which has better capabilities of dealing with dynamic data.

5.4 Radio Communication Integration

Equipped with the strategies outlined in this chapter, a driver for the TinyOS message abstraction has been developed during this work. It can send and receive packets over the radio as well as any other communication channel, which provides the `AMSend` and `Receive` interfaces in TinyOS.

5.4.1 Java Side

In Java, the `Radio` class exposes the driver interface, while the `Packet` class mirrors the message data structure of TinyOS. Radio has one method for each sending and receiving, respectively. A thin abstraction between the native methods and the public interface translates internal errors to exceptions and assigns source and destination for convenience. It enters the packet's monitor before calling the native back-end, allowing the driver to call wait on the packet in the current thread's context and protecting the same packet from being passed again to the same function by another thread. The Radio class also serves as a Packet factory, returning packet objects containing a preallocated data buffer with the maximum payload size. All objects for packet transmission and retrieval are constructed in Java, so that the driver does not need to perform these operations. This also holds for reception, as the `receive` method does not return a new packet, instead it writes into an existing one which it has been given as argument. The programmer may reuse the same packet object after processing its data, resulting in less garbage objects.

5.4.2 TinyOS Side

Driver Design In TinyOS, packets are represented by the `message_t` structure. They carry source and destination, as well as a payload and some other internal data. For both sending and receiving, the driver employs a similar strategy. It keeps a reference to the Packet object, passed to the native function, while serving a Java thread. This bridges the gap between the command and the event, as TinyOS does not have a cookie for such operations. The component is forced to keep track of the information associated with a split phase operation. When the event finally occurs, the driver knows on which object the notification should be done. If the driver is idle, the reference to the communication object is set to null. This property can be used to detect attempts for concurrent use of the driver which is not supported.

Sending Besides synchronization, the sending code copies the payload from Java to an internal message buffer and reads out the destination address. The thread is paused and resumed via the described mechanism.

Receiving Unlike sending, packet reception has no direct relation between the request and the corresponding event. There is no active command to receive a packet, just turning on the radio. Hence, the packet reception can occur at any time. If a thread is waiting for a packet at the time of the arrival, it is copied to Java and the thread is awaken by a notification. Otherwise, the driver keeps the last received packet in an internal buffer, which can be immediately delivered to Java without pausing the requesting thread. Only one packet is queued, so information is lost if the Java program does not actively read it. Future improvements might use a designated thread to read out all messages or implement a longer queue inside the driver.

5.4.3 Power Management

The embedded version of TakaTuka has to deal differently with the sleep state, since TinyOS is still running under the hood, which might have further work to do, even when TakaTuka can pause. To enter the sleep mode, a flag is set, and the main loop exits, returning control to TinyOS, which does not post the main loop task again. TinyOS automatically puts power down if no tasks are in the queue, waking only on interrupts. Inside the resulting events, the driver code may resume the virtual machine, which is necessary if a thread was notified.

6 Evaluation

In this section, several aspects of the implementation of threads in TakaTuka are evaluated with respect to their memory usage and computation overhead. The majority of the statistics is descriptive and does not compare different approaches. This evaluation gives an overview of the resources used by the current implementation in order address improvements precisely.

6.1 Memory Usage

The TakaTuka virtual machine can be configured to use either four or two bytes for the Java `int` type, while `long` is always configured to occupy 8 bytes. As two bytes is the size for the `int` type in C on the mote, TakaTuka uses this setting as default for Java, too. Therfore, the results are related to two byte sized integers.

6.1.1 Virtual Machine Memory

Thread support significantly increases the size of the virtual machine binary. The memory used by thread and TinyOS support is presented for two architectures:

- AVR: 8 bit *Atmel ATmega128 AVR* microcontroller [6], compiled with `avr-gcc` and linked to the `avr-libc` [1]. This architecture is used for the mica2 mote.

- I686: Intel i686 compatible, compiled with `gcc` (the GNU C compiler) running on Linux and linked to the `glibc` C library.

VM Code and Data Sizes		
Version	Code	RAM
AVR no threads	17280	331
AVR threads, no TinyOS	20046	345
AVR threads and TinyOS, Radio drivers	55150	459
I686 non threaded	15686	956
I686 threaded	18277	1012
AVR overhead	2766	14
TinyOS overhead	35104	114
I686 overhead	2591	56

Values are in bytes and were retrieved by the `size (1)` unix utility, `avr-size` respectively. Code is the size of the `text` section of the executable. RAM is the sum of the `data` and `bss`

sections [1]. The measured overhead is the difference between the threaded and non-threaded version. The TinyOS overhead is relative to the non-TinyOS but threaded version.

The greater code size for the AVR architecture results from the 8 bit processor. It needs multiple instructions for copying and arithmetic of 16 and 32 bit types. It has a reduced instruction set (RISC) with many registers. This leads to longer function prologue and epilogue code to save and restore these registers, as well as more instructions for certain tasks, as compared to i686 using a complex instruction set (CISC). The greater RAM usage for i686 is originated in the glibc library.

6.1.2 Java Code Size

The following table shows the Java code size, as generated by the loader in our optimized format, sizes in bytes. The program used in the example prints a number on the standard output or the serial communication respectively. There is no difference in size between the mote and the PC as the platform specific code is embedded in the virtual machine.

Java Code Size	
Version	Java Code Size
No threads	2507
Thread support total	4758
Thread support and Radio driver total	5292
Thread support increase	2251
Radio driver increase	534

6.1.3 Stack

Stack-frames are organized as described in section 4.1.2. A stack slot is modeled by a tagged union named `stackv_t`. It uses a one byte tag to identify the type of the value currently stored in the slot, which is needed to identify reference values for garbage collection. The tag information is also used for debugging, as it can identify invalid stack use resulting from virtual machine bugs. Single precision floating point values occupy four bytes in the underlying implementation. A slot must be big enough to hold any Java type, except long and double. The resulting size of a stack slot is thereby five bytes.

[1] Data contains program variables which are explicitly initialized or shared between several C files. Bss contains static variables, which are automatically initialized to 0 by the runtime. Text contains program code, which is stored in flash memory on the AVR.

stackv_t		
Variable	Type	Size in Bytes
`type`	`byte`	1
`i`	`int`	4
`f`	`float`	
`a`	`reference`	
Sum		5

StackFrame		
Variable	Type	Size in Bytes
`parent_frame`	`pointer`	2
`method_id`	`Method`	2
`stack_pointer`	`pointer`	2
`program_counter`	`pointer`	2
`stack_slots[]`	`array of stackv_t`	5·(`locals+stack`)
Sum		8 + 5·(`locals+stack`)

As an example, the following instance method would allocate a stack-frame with a size of 28 bytes: two local variables, two operand stack slots and the frame header. It also shows the computation memory overhead of stack based virtual machines.
Java Code:

```
class Object {
    public boolean equals(/* Object this; */ Object other) {
        return this == other;
    }
}
```

Bytecode:

```
1:  aload_0    push this onto the stack
2:  aload_1    push other
3:  acmpeq     compare: pop 2 values, push result
4:  ireturn    pop result and return it
```

6.1.4 Thread Objects

Constructing a thread has constant memory costs. Each thread occupies 27 bytes in memory plus an entry in the object ID to address map.

Thread		
Variable	Type	Size in Bytes
Java Variables		
`Thread.runnable`	`Runnable`	2
`Thread.interruptedFlag`	`boolean`	1
`Thread.prev`	`Thread`	2
`Thread.next`	`Thread`	2
Native Variables		
`class_id`	`Class`	2
`flags`	`byte`	1
`state`	`byte`	1
`monitor_id`	`Object`	2
`monitor_count`	`int`	2
`wakeup_time`	`long`	8
`pending_exception`	`Throwable`	2
`stack_frame`	`pointer`	2
Sum		27

6.1.5 Java State and Service Thread

The virtual machine class and the scheduler class store themselves some data only related to threads:

VM & Scheduler		
Variable	Type	Size in Bytes
`VM.serviceRequest`	`int`	2
`VM.serviceThread`	`Thread`	2
`Scheduler.previousThread`	`Thread`	2
Sum		6

Most resource requirements are introduced by the additional service thread. The service thread itself runs inside an infinite loop, which requires a permanent stack-frame in memory. This frame has no arguments or local variables, but uses an operand stack of three slots. Summed up, the memory usage increase generated by the Java code for threading in terms of RAM is as follows:

Threads: Java Memory	
Runtime Data	Size in Bytes
VM & Scheduler state	6
Extra service thread	27
Permanent stack-frame	8+15 = 23
Sum	56

6.2 Stack Speed

This section compares the speed of the stack for two different allocation schemes. Firstly, the mechanism relying on the memory management of the C library and secondly a first in first out stack implementation inside the virtual machine. The program used as benchmark is a recursive calculation of the fibonacci function. Implemented in this way, the time complexity for calculating fibonacci numbers grows exponentially. This leads to a high number of method calls and stack-frame allocations and deallocations.

```
int fib(int n) {
    if(n <= 2) return 1;
    return fib(n-1) + fib(n-2);
}
```

The test was performed on the PC to facilitate time measurement. The result for running the benchmark on the mote is expected to be comparable. The memory allocator implemented in the `avr-libc` will not be faster than the one on the PC as the resource constraints discourage trading memory for speed ([1], documentation of `malloc()`).

The above function was called with a parameter of n=30 and the time was measured with the unix `time (1)` utility.

Stack Speed	
Allocation Scheme	Time
FIFO	5.158s
Malloc/Free	9.245s

Obviously, the implementation using the general purpose memory allocator is slower. This result is not astonishing, but demonstrates the need for a proper stack implementation inside TakaTuka.

6.3 Scheduling Overhead

In order to evaluate the overhead of the scheduler in Java, the ratio between instructions executed in the service thread is related to the number of instructions executed in the application.

Two threads run concurrently, each printing numbers to the standard output in an infinite loop. The print operation involves locking the corresponding output stream object, so rescheduling is also initiated my monitor contention. The test runs a total of 100000 instructions.

Scheduling Overhead			
Ticks for each thread	Service Thread	Application	Ratio
100	41054	58946	0.7
1000	4719	95281	0.05
10000	701	99299	0.007

"Ticks for each thread" is the maximum number of instructions which a thread may execute before rescheduling is initiated. The instructions executed in the service thread is given as well as the instructions executed in the application code. The rescheduling operation takes approximately 30 bytecodes if the next thread in the list is runnable. If threads contend for locks, the scheduler has more work to do, which explains the high ratio for 100 ticks. With 1000 ticks before rescheduling, the overhead is already quite low with 5 percent. Executing 10000 instructions before rescheduling might not be desired, because it can lead to high latencies when reacting on events. From a theoretical approach, the current algorithm used for selecting a runnable thread has a worst case time complexity of $O(n)$, where n is the number of started threads.

7 Discussion and Outlook

In this chapter, several design decisions are reviewed and discussed regarding to their advantages and disadvantages. Solutions found in the Squawk virtual machine are presented and compared to the implementation in TakaTuka. Based on the evaluation results, future improvements are suggested and outlined. Furthermore, the purpose of this chapter is to document the current status of TakaTuka and its implementation of threads, and support ongoing work on controversial issues.

7.1 Java VM parts and Monitors

It is generally a good idea to implement parts of a JVM in Java itself. Testing is easier because Java code is protected by many runtime checks. For example, null pointer accesses, invalid casts or array indices are caught by the virtual machine. On the other hand, all functionality written in Java has to be interpreted (at least in our case) and is orders of magnitudes slower. Java objects and variables require more code and data memory than native their counterparts.

Taking the realization of monitors in Squawk as example, it is illustrated, how to efficiently implement parts of the virtual machine in Java.

Squawk accepts no compromise, as most data structures are visible in Java. Monitors are allocated on demand and attached inside a so called `ObjectActivation` to the object. An object activation has the ability to serve as a proxy to class objects. Hence, it can replace the objects class, saving an extra reference variable. Squawk uses a queue of pending monitors to indicate synchronization on them. Monitor operations are executed lazily inside the virtual machine. Contentions are detected at the time of a context switch. [2] In such a case a Java method is invoked, which performs the monitor operations. This effectively reduces the computation needed for uncontended monitor operations, because they do not call Java code. However, the RAM usage of monitors is increased, as they are full blown objects.

TakaTuka aims at devices significantly smaller than supported by Squawk. The choice of native monitors is based on this fact. The interface between the virtual machine and Java together with the actual implementation would introduce a considerable overhead and make the monitor bytecodes more complicated. Native code has more freedom in dealing with monitors, possibly providing a more effective allocation strategy. On the other hand, a better integration into the scheduling algorithm can be established, if monitor states are accessible from Java. A good solution should find a way to accommodate both aspects. A possible starting point for improvements is the code for wait/notify, which is more loosely coupled to the virtual machine internals.

A new aspect is introduced by the *vm2c* module of Squawk. It can translate a subset of Java

[2] This information is taken from the Squawk sourcecode [24].

code to C. This enables the virtual machine programmer to choose between a native implementation and the interpreted one. For example, he might use the latter during debugging, while the compiled version is used for the production virtual machine. Squawk uses this approach for its garbage collector. Future work could adapt vm2c to TakaTuka, as it is open source.

7.2 Stack Design

Squawk uses a list of stack chunks for each thread [21], which themselves are modeled as arrays in the Java heap. Each has a pointer to the next chunk. This creates a dynamic size stack with moderate overhead. It is possible that a method is called subsequently several times when the current chunk is full, requiring to allocate the next chunk for the method's calls. For optimization, the virtual machine keeps one unused chunk at the end of the list for this case, as the described situation would otherwise lead to excessive allocation/deallocation.

TakaTuka should use a similar technique in the future, as the current design involves the computation overhead of the underlying memory manager. A more sophisticated approach could incorporate ideas from the RETOS operating system [16], which has a shared kernel stack. The system could have one main stack on which all threads run and move parts of it to the heap if needed. Research into this direction has already been made with the multi stack sharing technique [19] or the Knots system [4].

7.3 Scheduler Improvements

The virtual machine can be extended to initiate context switches at times, when the current thread is in a low resource state: for example when it owns less monitors or when the call depth is small. Such a strategy could also use grace times similar to [8]. Resource sharing will be more cooperative, resulting in better performance and memory usage.

Currently, the scheduler has only one list for all threads disregarding their states. Starting with three different lists for the states runnable, blocked and (timed) waiting, the scheduling algorithm can be improved to finally run in constant time. In order to support this, the implementation of synchronization and wait/notify has to be adapted to move threads between these lists

7.4 Thread List

The current solution used for the linked list of threads exposes the prev and next references as public variables. This is fast and simple, but access should be limited to the scheduler class.

Squawk separates threads in a public interface provided by the `Thread` class and a private interface, provided by the `VMThread` class. For a new thread, one object of each is constructed with bilateral references. The scheduler operates mainly on the vm-threads by keeping them

in a list. This solution hides all implementation details from the application interface with the cost of an extra object per thread.

7.5 Priorities

Priorities can almost entirely implemented in Java with the only limitation that the native code for `notify` would have to be adapted to prefer processes with the high priority. The scheduler might keep a separate list of threads for each priority, first executing each runnable thread of a higher priority before considering the next priority level. Generally, thread starvation is an issue: low prioritized processes are never chosen to run because of activity in a higher prioritized process. Priority inversion detection would also need support by the virtual machine, either by exposing the set of locked monitors of a thread or by implementing it natively.

7.6 Service Thread Design Aspects

The decision for a separate service thread was chosen for several reasons, which will be explained in detail.

- Running Java code in the virtual machine requires some registers, including the instruction pointer, the stack pointer and the current method. Each thread has its private copy of these registers and also its assigned stack space. Running a service in an arbitrary thread would act like an artificial method call. This means that the thread's stack and registers are used for the evaluation of the service. As a consequence, each thread's stack must be large enough to run any service at any time. Also, if the service throws an uncaught exception, it will propagate down the call hierarchy, disrupting the operation of the previously running code.

- The virtual machine can not assume safely, that the current thread, in which a service might run, is in fact runnable. For example, it could just have called wait, resulting in the need to reschedule. The current thread's state would have to be forcibly set to runnable, allowing operation in its context, later restoring its previous state.

Summarized, the use of a designated service thread frees the virtual machine from all concerns of mixing up two execution flows in one context. Besides being less bug-prone, this results in a cleaner and more logical design.

On the other hand, there are several drawbacks with this approach:

- The additional thread increases the memory requirements of the virtual machine, especially in terms of valuable RAM.

- Running a service requires a context switch, which produces some overhead. The switch between two application threads results in two low level context switches, since the scheduler is required to run in its own thread.

In the non-threaded version of the virtual machine, most of the rationale for a separate service thread are void. If the program blocks on an operation, there is no need to keep the system alive (i.e. running other threads). It can simply wait for the operation to terminate. The whole concept of locks and waiting is not used, so the state of the current flow of execution is always runnable.

As conclusion, having a separate service thread results in a cleaner design of the virtual machine internals when running multiple threads. The non-threaded interpreter uses a different implementation of the `VM` class and services to save resources.

7.7 Code Size and RAM Usage

The results from section 6 show two sides of the resource requirements of the presented thread implementation. On the virtual machine side, the memory increase introduced by threading is clearly visible but still moderate. The memory usage for maintaining thread objects and the service stack is quite high. The increase of Java code size is almost the same as compared to the native threading code, although it provides much less functionality. TinyOS integration is costly, but one has to keep in mind, that it includes a complete radio driver with a full protocol stack. Overall, the requirements seem low enough to run medium sized Java applications on motes lice the mica2.

8 Conclusion

This work successfully introduces threads into the TakaTuka virtual machine. The majority of thread mechanisms has been implemented, including synchronization, notifications and preemptive concurrency.

Integration of TakaTuka into TinyOS was established and makes use of the provided hardware abstraction. A convenient blocking interface can be implemented on-top of the underlying event driven architecture with the current status of multithreading in TakaTuka. Several open issues have been pointed out together with suggestions for improvements.

Threads do use a considerable amount of memory, in particular in terms of valuable RAM. The results also indicate, that running Java code is generally costly. Using Java code for the program logic and native methods for computation intensive tasks is suggested as a trade-off between ease of programming and efficiency.

`:wq`

References

[1] AVR Libc Homepage.
`http://www.nongnu.org/avr-libc/`.

[2] J2ME CLDC and K virtual machine: Frequently Asked Questions.
`http://java.sun.com/products/cldc/faqs.html`.

[3] Java 2 Platform Standard Edition 5.0: API Reference: Thread.
`http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html`.

[4] Knots: An Effcient Single Stack Preemption Mechanism for Resource Constrained Devices.
`http://nesl.ee.ucla.edu/fw/documents/reports/2007/akhi07knots.pdf`.

[5] SunSPOT Documentation.
`http://www.sunspotworld.com/docs/index.php`.

[6] Atmel. ATmega128 Datasheet.
`http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`.

[7] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[8] Cormac Duffy and Utz Roedig and John Herbert and Cormac J. Sreenan. Adding Preemption to TinyOS. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 88–92, New York, NY, USA, 2007. ACM.

[9] Crossbow. Mica2 Datasheet.
`http://www.xbow.com/products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf`.

[10] David Gay, David Culler, Philip Levis. nesC Language Reference Manual.
`http://nescc.sourceforge.net/papers/nesc-ref.pdf`.

[11] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM.

[12] Eric Arseneau. Squawk Forum, about memory requirements.
`http://forums.java.net/jive/thread.jspa?threadID=38680&tstart=0`.

[13] Janice J. Heiss. Sentilla's Pervasive Computing – The Universe Is the Computer. 2008 JavaOne Conference, `http://java.sun.com/javaone/sf/2008/articles/gen_sentilla.jsp`.

[14] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *Java(TM) Language Specification (2nd Edition)*. Addison-Wesley Pub Co, June 2000.

[15] Hyoseung Kim and Hojung Cha. Towards a resilient operating system for wireless sensor networks. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 2006. USENIX Association.

[16] Hyoseung Kim and Hojung Cha. Multithreading optimization techniques for sensor network operating systems. In *EWSN*, pages 293–308, 2007.

[17] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. pages 115–148. 2005.

[18] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[19] Bhuvan Middha, Matthew Simpson, and Rajeev Barua. Mtss: multi task stack sharing for embedded systems. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 191–201, New York, NY, USA, 2005. ACM.

[20] Philip Levis. TinyOS Programming. `http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf`.

[21] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88, New York, NY, USA, 2006. ACM.

[22] Sun Microsystems, Inc. The Connected Limited Device Configuration. `http://java.sun.com/products/cldc/`.

[23] Sun Microsystems, Inc. The Squawk Project. `http://research.sun.com/projects/squawk/`.

[24] Sun Microsystems, Inc. The Squawk Sourcecode Repository. `https://squawk.dev.java.net/svn/squawk/trunk`.